

Kubernetes изнутри



Джей Вьяс
Крис Лав



MANNING



Kubernetes изнутри

Развертывание Kubernetes в реальных условиях — сложная задача. Даже небольшие ошибки в конфигурации или в архитектуре могут серьезно навредить вашей системе. Поэтому всегда полезно знать, как работает каждый компонент, чтобы уметь быстро устранять неполадки и переходить к следующей задаче. Эта единственная в своем роде книга включает подробные сведения об устройстве Kubernetes и советы профессионалов, которые помогут вам поддерживать работоспособность ваших приложений. Исследуется внутреннее устройство Kubernetes, от управления iptables до настройки динамически масштабируемых кластеров, реагирующих на изменение нагрузки. Каждая страница несет новую информацию о настройке и управлении Kubernetes, а также о том, как справляться с неизбежно возникающими проблемами.

Для разработчиков и администраторов Kubernetes со средним уровнем подготовки.

Рассматриваемые темы:

- основные компоненты Kubernetes;
- хранилище и интерфейс контейнерного хранилища;
- безопасность Kubernetes;
- способы создания кластеров Kubernetes;
- подробное описание плоскости управления, сети и других основных компонентов.

Джей Вьяс и Крис Лав — опытные разработчики Kubernetes.

«Эта книга поможет понять основы внутреннего устройства Kubernetes».

Убальдо Пескаторе, PagaPA

«Детально исследует наиболее важные компоненты Kubernetes. Написана понятным языком, содержит множество подробных примеров и диаграмм».

Роб Рюэтч, adesso SE

«Подробное руководство, с помощью которого я смог быстро приступить к работе. Настоятельно рекомендую!»

Эл Кринкер, USPTO

«Идеальное руководство по Kubernetes».

*Ганди Раджан,
Software Dell Technologies*

ISBN 978-5-93700-153-5



9 785937 001535 >

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru



Джей Вьяс
Крис Лав

Kubernetes изнутри

Core Kubernetes

JAY VYAS
CHRIS LOVE



MANNING
Shelter Island

Kubernetes изнутри

ДЖЕЙ ВЪЯС
КРИС ЛАВ



Москва, 2023

УДК 004.042Kubernetes
ББК 32.372
В96

Вьяс Дж., Лав К.

В96 Kubernetes изнутри / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. – 378 с.: ил.

ISBN 978-5-93700-153-5

В этой книге подробно рассказывается о настройке и управлении платформой Kubernetes, а также о том, как быстро и эффективно устранять неполадки. Исследуется внутреннее устройство Kubernetes – от управления iptables до настройки динамически масштабируемых кластеров, реагирующих на изменение нагрузки. Советы профессионалов помогут вам поддерживать работоспособность ваших приложений. Особое внимание уделяется теме безопасности.

Книга адресована разработчикам и администраторам Kubernetes со средним уровнем подготовки.

УДК 004.042Kubernetes
ББК 32.372

Copyright © DMK Press 2022. Authorized translation of the English edition © 2022 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

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.

ISBN 978-1-6172-9755-7 (англ.)
ISBN 978-5-93700-153-5 (рус.)

© Manning Publications, 2022
© Перевод, оформление, издание, ДМК Пресс, 2022

Амиму Кнаббену (Amim Knabben), Рикардо Кацу (Ricardo Katz), Мэту Фенвику (Matt Fenwick), Антонио Охеа (Antonio Ojea), Раджасу Какодару (Rajas Kakodar) и Микаэлю Клюзо (Mikael Cluseau) за многочасовые исследования K8s по ночам и выходным и увлекательные соревнования по крику. Эндрю Стойокосу (Andrew Stoyocos), возглавлявшему группу политик в SIG Network. Моей жене и семье, позволившим мне писать эту книгу по субботам. Гари (Gary), Роне (Rona), Норе (Nora) и Джинджину (Gingin) за помощь моей маме.

– Джей

Кейт (Kate) и всем моим близким, поддерживавшим меня в этом путешествии. Спасибо команде LionKube, особенно Одри (Audrey) за организацию работы и Шарифу (Sharif) за помощь и поддержку. Также моему соавтору Джею (Jay), предложившему мне принять участие в работе над этой книгой вместе с ним, я благодарю тебя за это! Без твоей целеустремленности и упорства у нас ничего не получилось бы.

– Крис

Оглавление

1	■ Почему появился Kubernetes.....	24
2	■ Зачем нужны модули Pod?.....	40
3	■ Создание модулей Pod.....	68
4	■ Использование контрольных групп для управления процессами в модулях Pod.....	103
5	■ Интерфейсы CNI и настройка сети в модулях Pod	132
6	■ Устранение проблем в крупномасштабных сетях.....	154
7	■ Хранилища в модулях Pod и CSI.....	179
8	■ Реализация и моделирование хранилищ.....	198
9	■ Запуск модулей Pod: как работает kubelet.....	221
10	■ DNS в Kubernetes	243
11	■ Плоскость управления	257
12	■ etcd и плоскость управления.....	272
13	■ Безопасность контейнеров и модулей Pod	296
14	■ Безопасность узлов и Kubernetes	312
15	■ Установка приложений.....	343

Содержание

Оглавление	6
Предисловие	14
Благодарности	15
О книге	17
Об авторах	21
Об иллюстрации на обложке	23

1 Почему появился Kubernetes	24
1.1 Предварительный обзор некоторых основных терминов	25
1.2 Проблема дрейфа инфраструктуры и Kubernetes	26
1.3 Контейнеры и образы	27
1.4 Базовая основа Kubernetes	29
1.4.1 Все инфраструктурные правила в Kubernetes определяются в обычных файлах YAML	31
1.5 Возможности Kubernetes	32
1.6 Компоненты и архитектура Kubernetes	34
1.6.1 Kubernetes API	35
1.6.2 Пример первый: интернет-магазин	37
1.6.3 Пример второй: онлайн-решение для благотворительности	37
1.7 Когда не стоит использовать Kubernetes	38
Итоги	38

2 Зачем нужны модули Pod?	40
2.1 Пример веб-приложения	42
2.1.1 Инфраструктура нашего веб-приложения	44
2.1.2 Эксплуатационные требования	45
2.2 Что такое Pod?	46
2.2.1 Пространства имен в Linux	47
2.2.2 Kubernetes, инфраструктура и Pod	49
2.2.3 Объект Node	51
2.2.4 Наше веб-приложение и плоскость управления	55
2.3 Создание веб-приложения с помощью kubectl	56
2.3.1 Сервер Kubernetes API: kube-apiserver	57
2.3.2 Планировщик Kubernetes: kube-scheduler	58

2.3.3	Контроллеры инфраструктуры	59
2.4	Масштабирование, высокодоступные приложения и плоскость управления	63
2.4.1	Автоматическое масштабирование	65
2.4.2	Управление затратами	66
Итоги	67

3	Создание модулей Pod	68
3.1	Общий обзор примитивов Kubernetes	71
3.2	Что такое примитивы Linux?	72
3.2.1	Примитивы Linux – это инструменты управления ресурсами	73
3.2.2	Все сущее является файлом (или файловым дескриптором)	74
3.2.3	Файлы можно комбинировать	75
3.2.4	Настройка kind	76
3.3	Использование примитивов Linux в Kubernetes	78
3.3.1	Предварительные условия для запуска модуля Pod	78
3.3.2	Запуск простого модуля Pod	79
3.3.3	Исследование зависимостей модуля Pod от Linux	81
3.4	Создание модуля Pod с нуля	86
3.4.1	Создание изолированного процесса с помощью chroot	87
3.4.2	Использование mount для передачи данных процессам	89
3.4.3	Защита процесса с помощью unshare	91
3.4.4	Создание сетевого пространства имен	92
3.4.5	Проверка работоспособности процесса	93
3.4.6	Ограничение потребления процессора с помощью cgroups	94
3.4.7	Создание раздела resources	95
3.5	Использование модуля Pod в реальном мире	96
3.5.1	Проблема сети	97
3.5.2	Как kube-проху реализует сервисы Kubernetes с помощью iptables	98
3.5.3	Использование модуля kube-dns	98
3.5.4	Другие проблемы	100
Итоги	102

4	Использование контрольных групп для управления процессами в модулях Pod	103
4.1	Модули Pod простаивают до завершения подготовительных операций	104
4.2	Процессы и потоки в Linux	106
4.2.1	Процессы systemd и init	108
4.2.2	Контрольные группы для процессов	109
4.2.3	Реализация контрольных групп для обычного модуля Pod	112
4.3	Тестирование контрольных групп	114
4.4	Как kubelet управляет контрольными группами	115
4.5	Как kubelet управляет ресурсами	116
4.5.1	Почему ОС не может использовать подкачку в Kubernetes?	117
4.5.2	Хак: настройка приоритета «для бедных»	118
4.5.3	Хак: настройка HugePages с помощью контейнеров инициализации	119
4.5.4	Классы QoS: почему они важны и как они работают	120
4.5.5	Создание классов QoS путем настройки ресурсов	121

4.6	Мониторинг ядра Linux с помощью Prometheus, cAdvisor и сервера API.....	122
4.6.1	Публикация метрик обходится недорого и имеет большую ценность	124
4.6.2	Почему Prometheus?	125
4.6.3	Создание локального сервиса мониторинга Prometheus	126
4.6.4	Исследование простоев в Prometheus.....	129
Итоги	131

5 Интерфейсы CNI и настройка сети в модулях Pod..... 132

5.1	Зачем нужны программно-определяемые сети в Kubernetes	134
5.2	Реализация Kubernetes SDN на стороне сервиса: kube-proxy.....	136
5.2.1	Плоскость данных в kube-proxy	138
5.2.2	Подробнее о NodePort	140
5.3	Провайдеры CNI	141
5.4	Два плагина CNI: Calico и Antrea	143
5.4.1	Архитектура плагинов CNI	143
5.4.2	Давайте поэкспериментируем с некоторыми CNI	144
5.4.3	Установка провайдера CNI Calico	146
5.4.4	Организация сети в Kubernetes с OVS и Antrea.....	149
5.4.5	Замечание о провайдерах CNI и kube-proxy в разных ОС	152
Итоги	153

6 Устранение проблем в крупномасштабных сетях..... 154

6.1	Sonobuoy: инструмент подтверждения работоспособности кластера.....	155
6.1.1	Трассировка движения данных модулей Pod в кластере	156
6.1.2	Настройка кластера с CNI-провайдером Antrea	157
6.2	Исследование особенностей маршрутизации в разных провайдерах CNI с помощью команд <code>arp</code> и <code>ip</code>	158
6.2.1	Что такое IP-туннель и почему его используют провайдеры CNI?	159
6.2.2	Сколько пакетов проходит через сетевые интерфейсы CNI?.....	160
6.2.3	Маршруты	161
6.2.4	Инструменты для CNI: Open vSwitch (OVS).....	163
6.2.5	Трассировка движения данных активных контейнеров с помощью <code>tcpdump</code>	164
6.3	kube-proxy и iptables.....	166
6.3.1	iptables-save и diff.....	166
6.3.2	Как сетевые политики изменяют правила CNI	167
6.3.3	Как реализуются политики?.....	170
6.4	Входные контроллеры.....	172
6.4.1	Настройка Contour и кластера kind для изучения входных контроллеров	173
6.4.2	Настройка простого модуля Pod с веб-сервером	174
Итоги	178

7	Хранилища в модулях Pod и CSI	179
7.1	Небольшое отступление: виртуальная файловая система (VFS) в Linux	181
7.2	Три вида хранилищ для Kubernetes	182
7.3	Создание PVC в кластере kind	184
7.4	Интерфейс контейнерного хранилища (CSI)	188
7.4.1	Проблема внутреннего провайдера	189
7.4.2	CSI как спецификация, работающая внутри Kubernetes	191
7.4.3	CSI: как работает драйвер хранилища	193
7.4.4	Привязка точек монтирования	193
7.5	Краткий обзор действующих драйверов CSI	194
7.5.1	Контроллер	194
7.5.2	Интерфейс узла	195
7.5.3	CSI в операционных системах, отличных от Linux	196
	Итоги	196
8	Реализация и моделирование хранилищ	198
8.1	Микрокосм в экосистеме Kubernetes: динамическое хранилище	199
8.1.1	Оперативное управление хранилищем: динамическое выделение ресурсов	200
8.1.2	Локальное хранилище в сравнении с emptyDir	201
8.1.3	Тома PersistentVolume	203
8.1.4	Интерфейс контейнерного хранилища (CSI)	204
8.2	Динамическая подготовка выигрывает от CSI, но не зависит от него	205
8.2.1	Классы хранилищ (StorageClass)	206
8.2.2	Вернемся к центрам обработки данных	207
8.3	Варианты организации хранилищ в Kubernetes	209
8.3.1	Секреты: эфемерная передача файлов	209
8.4	Как выглядит типичный провайдер динамического хранилища?	212
8.5	hostPath для управления системой и/или доступа к данным	214
8.5.1	hostPath, CSI и CNI: канонический вариант использования	214
8.5.2	Cassandra: пример реального хранилища в Kubernetes	217
8.5.3	Дополнительные возможности и модель хранения в Kubernetes	218
8.6	Дополнительная литература	219
	Итоги	220
9	Запуск модулей Pod: как работает kubelet	221
9.1	kubelet и узел	222
9.2	Основы kubelet	223
9.2.1	Среда выполнения контейнеров: стандарты и соглашения	224
9.2.2	Конфигурационные параметры и API агента kubelet	225
9.3	Создание модуля Pod и его мониторинг	228
9.3.1	Запуск kubelet	229
9.3.2	После запуска: жизненный цикл узла	230
9.3.3	Механизм аренды и блокировки в etcd, эволюция аренды узла	230
9.3.4	Управление жизненным циклом Pod в kubelet	231
9.3.5	CRI, контейнеры и образы: как они связаны	233
9.3.6	kubelet не запускает контейнеры: это делает CRI	233
9.3.7	Приостановленный контейнер: момент истины	235

9.4	Интерфейс времени выполнения контейнеров (CRI).....	235
9.4.1	Сообщаем Kubernetes, где находится среда выполнения контейнеров	235
9.4.2	Процедуры CRI	236
9.4.3	Абстракция kubelet вокруг CRI: GenericRuntimeManager	236
9.4.4	Как вызывается CRI?	237
9.5	Интерфейсы kubelet	237
9.5.1	Внутренний интерфейс среды выполнения	237
9.5.2	Как kubelet извлекает образы: интерфейс ImageService	239
9.5.3	Передача ImagePullSecret в kubelet	240
9.6	Дополнительная литература.....	241
Итоги	241

10	DNS в Kubernetes	243
10.1	Краткое введение в DNS (и CoreDNS).....	243
10.1.1	NXDOMAIN, записи A и записи CNAME	244
10.1.2	Модулям Pod нужен внутренний DNS	246
10.2	Почему StatefulSet, а не Deployment?.....	248
10.2.1	DNS и автономные сервисы	248
10.2.2	Постоянные записи DNS в StatefulSet	250
10.2.3	Развертывание с несколькими пространствами имен для изучения свойств модуля DNS	250
10.3	Файл resolv.conf	252
10.3.1	Краткое примечание о маршрутизации.....	252
10.3.2	CoreDNS: вышестоящий сервер имен для ClusterFirst DNS.....	254
10.3.3	Разбор конфигурации плагина CoreDNS.....	255
Итоги	256

11	Плоскость управления	257
11.1	Плоскость управления	258
11.2	Особенности сервера API	259
11.2.1	Объекты API и пользовательские ресурсы	259
11.2.2	Определения пользовательских ресурсов (CRD)	261
11.2.3	Планировщик	261
11.2.4	Краткий обзор фреймворка планирования	267
11.3	Диспетчер контроллеров	267
11.3.1	Хранилище.....	268
11.3.2	Учетные данные сервисов и токены	269
11.4	Облачные диспетчеры контроллеров Kubernetes (CCM)	269
11.5	Дополнительная литература.....	271
Итоги	271

12	etcd и плоскость управления	272
12.1	Заметки для нетерпеливых	273
12.1.1	Мониторинг производительности etcd с помощью Prometheus	274
12.1.2	Когда нужно настраивать etcd	278
12.1.3	Пример: быстрая проверка работоспособности etcd	280
12.1.4	etcd v3 и v2	280
12.2	etcd как хранилище данных	281
12.2.1	Можно ли запустить Kubernetes в других базах данных?	281

12.2.2	Строгая согласованность	283
12.2.3	Согласованность в etcd обеспечивают операции fsync	283
12.3	Обзор интерфейса Kubernetes с etcd	285
12.4	Задача etcd – надежное хранение фактов.....	285
12.4.1	Журнал упреждающей записи etcd	287
12.4.2	Влияние на Kubernetes	287
12.5	Теорема CAP.....	287
12.6	Балансировка нагрузки на уровне клиента и etcd	289
12.6.1	Ограничения по размеру: о чем (не) следует беспокоиться	289
12.7	Шифрование хранимых данных в etcd	291
12.8	Производительность и отказоустойчивость etcd в глобальном масштабе	292
12.9	Интервал отправки контрольных сообщений в высокораспределенной etcd	292
12.10	Настройка клиента etcd в кластере kind	293
12.10.1	Запуск etcd в окружении, отличном от Linux.....	294
Итоги	295

13 Безопасность контейнеров и модулей Pod.....296

13.1	Радиус взрыва	297
13.1.1	Уязвимости	298
13.1.2	Вторжение	298
13.2	Безопасность контейнера	298
13.2.1	Планирование обновления контейнеров и пользовательского программного обеспечения	299
13.2.2	Контроль контейнеров	299
13.2.3	Пользователи в контейнерах – не запускайте ПО от имени root	300
13.2.4	Используйте наименьшие возможные контейнеры.....	300
13.2.5	Происхождение контейнера	301
13.2.6	Линтеры для контейнеров	302
13.3	Безопасность модулей Pod.....	302
13.3.1	Контекст безопасности	303
13.3.2	Расширение привилегий и возможностей	305
13.3.3	Политики безопасности Pod (PSP)	307
13.3.4	Не внедряйте автоматически токен учетной записи сервиса	309
13.3.5	Модули Pod с привилегиями root	309
13.3.6	Граница безопасности	310
Итоги	311

14 Безопасность узлов и Kubernetes312

14.1	Безопасность узла.....	312
14.1.1	Сертификаты TLS.....	313
14.1.2	Неизменяемые ОС и применение исправлений на узлах	314
14.1.3	Изолированные среды выполнения контейнеров	315
14.1.4	Атаки на ресурсы	316
14.1.5	Единицы измерения потребления процессора	317
14.1.6	Единицы измерения объема памяти	317
14.1.7	Единицы измерения объема хранилища	318
14.1.8	Сети хостов и модулей Pod	318
14.1.9	Пример модуля Pod	319

14.2	Безопасность сервера API	320
14.2.1	Управление доступом на основе ролей (RBAC)	320
14.2.2	Определение RBAC API	321
14.2.3	Ресурсы и подресурсы	323
14.2.4	Субъекты и RBAC	325
14.2.5	Отладка RBAC	326
14.3	Authn, Authz и Secret	326
14.3.1	Учетные записи сервисов IAM: защита облачных API	327
14.3.2	Доступ к облачным ресурсам	328
14.3.3	Частные серверы API	329
14.4	Безопасность сети	329
14.4.1	Сетевые политики	330
14.4.2	Балансировщики нагрузки	334
14.4.3	Агент открытой политики (OPA)	335
14.4.4	Коллективная аренда	338
14.5	Советы по Kubernetes	341
Итоги	341

15	Установка приложений	343
15.1	Размышления о приложениях в Kubernetes	344
15.1.1	Масштаб приложения влияет на выбор инструментов	345
15.2	Приложения на основе микросервисов обычно требуют тысячи строк определения конфигурации	345
15.3	Переосмысление установки приложения Guestbook в реальных условиях	346
15.4	Установка набора инструментов Carvel	347
15.4.1	Часть 1: разделение ресурсов на отдельные файлы	347
15.4.2	Часть 2: исправление файлов приложения с помощью ytt	349
15.4.3	Часть 3: развертывание приложения Guestbook и управление им	351
15.4.4	Часть 4: создание оператора kapp для упаковки приложения и управления им	355
15.5	И снова об операторах Kubernetes	359
15.6	Tanzu Community Edition: пример комплексного набора инструментов Carvel	362
Итоги	363

Предметный указатель	365
----------------------------	-----

Предисловие

Мы написали эту книгу для всех, кто хочет получить новые знания о K8s (Kubernetes) и сразу же углубиться в различные темы, связанные с хранением, сетевыми взаимодействиями и использованием разнообразных инструментов.

Мы не ставили перед собой цель написать исчерпывающее руководство по всем возможностям K8s API (это просто невозможно), но искренне верим, что, прочитав эту книгу, пользователи обретут понимание, которое поможет им по-новому взглянуть на сложные задачи организации инфраструктуры в промышленных кластерах и увидеть общее развитие ландшафта Kubernetes в более широком контексте.

Конечно, есть немало книг, позволяющих пользователям изучить основы Kubernetes, но мы хотели написать книгу, рассказывающую об основных технологиях, составляющих Kubernetes. Здесь мы расскажем вам все тонкости организации сети и плоскости управления, а также некоторые другие темы, чтобы вы смогли понять внутренние особенности работы Kubernetes. Понимание этих деталей сделает вас лучшим инженером DevOps и программистом.

Мы также надеемся вдохновить новых пользователей Kubernetes. Свяжитесь с нами в Твиттере (@jayunit100, @chrislovcnm), если решите принять участие в жизни сообщества Kubernetes или помочь нам добавить больше примеров для этой книги.

Благодарности

Мы хотим поблагодарить сообщество и компании, поддерживающие Kubernetes. Без них и их постоянных усилий не было бы этого замечательного программного обеспечения. Мы хотели бы упомянуть всех причастных, но боимся, что кого-то упустим, поэтому извиняемся заранее.

Спасибо нашим друзьям и наставникам в SIG Network (Микаэлю Клюзо (Mikael Cluseau), Халеду Хендиаку (Khaled Hendiak), Тиму Хокинсу (Tim Hockins), Антонио Охеа (Antonio Ojea), Рикардо Кацу (Ricardo Katz), Мэтту Фенвику (Matt Fenwick), Дэну Уиншипу (Dan Winship), Дэну Уильямсу (Dan Williams), Кейси Календеро (Casey Calendero), Кейси Девенпорт (Casey Davenport), Эндрю Си (Andrew Sy) и многим, многим другим); неутомимым разработчикам открытого исходного кода в сообществах SIG Network и SIG Windows (Марку Розетти (Mark Rosetti), Джеймсу Стареванту (James Sturevant), Клаудио Белу (Claudio Belu), Амиму Кнаббену (Amim Knabben)); первым основателям Kubernetes (Джо Беду (Joe Beda), Брендану Барнсу (Brendan Burns), Вилле Айкасу (Ville Aikas) и Крейгу Маклаки (Craig McLuckie)); а также инженерам из Google, включая Брайана Гранта (Brian Grant) и Тима Хокина (Tim Hockin), присоединившимся к ним вскоре после этого.

Мы благодарны духовным лидерам сообщества: Тиму Сент-Клеру (Tim St. Clair), Джордану Лиггиту (Jordan Liggitt), Бриджит Кромхаут (Bridget Kromhaut) и многим другим. Мы также хотели бы поблагодарить Раджаса Какодара (Rajas Kakodar), Анушу Хедж (Anusha Hedge) и Неху Лохию (Neha Lohia) за создание потрясающей команды SIG Network India, внесших колоссальное количество предложений, которые мы надеемся учесть в следующем издании (или возможном продолжении) этой книги, где мы предполагаем подробнее рассказать о приемах организации сети и о прокси-сервере kube-proxy.

Джей также хотел бы поблагодарить Клинта Китсона (Clint Kitson) и Аарти Ганесана (Aarthi Ganesan), давших возможность работать над этой книгой, будучи сотрудником VMware, а также его коллег в VMware (Амима (Amim) и Зака (Zac)), постоянно остававшихся рядом и по-

могавших советами. И конечно же, спасибо Фрэнсису Бурану (Frances Buran), Карен Миллер (Karen Miller) и многим другим сотрудникам Manning Publications, помогавшим готовить эту книгу к печати.

Наконец, спасибо всем нашим рецензентам: Алу Кринкеру (Al Krinker), Алессандро Кампейсу (Alessandro Campeis), Александру Эрциу (Alexandru Herciu), Аманде Деблер (Amanda Debler), Андреа Косентино (Andrea Cosentino), Андреесе Сакко (Andres Sacco), Анупаму Сенгупте (Anupam Sengupta), Бену Фенвику (Ben Fenwick), Борко Джурковичу (Borko Djurkovic), Дарье Василенко (Daria Vasilenko), Элиасу Рангелю (Elias Rangel), Эрике Хоулу (Eric Hole), Эриксу Зеленке (Eriks Zelenka), Ойгену Кокалеа (Eugen Cocalea), Ганди Раджану (Gandhi Rajan), Ирине Романенко (Iryna Romanenko), Джареду Дункану (Jared Duncan), Джеффу Лиму (Jeff Lim), Джиму Амрайну (Jim Amrhein), Хуану Хосе Дурильо Баррионуэво (Juan José Durillo Barrionuevo), Мэтту Фенвику (Matt Fenwick), Мэтту Велке (Matt Welke), Михалю Рутке (Michał Rutka), Риккардо Маротти (Riccardo Marotti), Робу Пачеко (Rob Pacheco), Робу Рютчу (Rob Ruetsch), Роману Левченко (Roman Levchenko), Райану Бартлетту (Ryan Bartlett), Убальдо Пескаторе (Ubaldo Pescatore) и Уэсли Рольнику (Wesley Rolnick). Ваши предложения помогли сделать эту книгу лучше.

О книге

Кому адресована эта книга

Всем желающим узнать больше о внутреннем устройстве Kubernetes, о том, как рассуждать о его режимах отказа и возможности расширения под нужды пользователей. Если вы не знаете, что такое Pod, то, конечно, можете приобрести эту книгу, но прежде прочтите какую-нибудь другую книгу, где вы сможете поближе познакомиться с терминологией.

Также книга пригодится операторам, желающим общаться на едином языке с сотрудниками ИТ-отделов, техническими директорами и другими руководителями о том, как внедрить Kubernetes, сохранив при этом основные принципы построения инфраструктуры, существовавшие до появления контейнеров. Эта книга действительно поможет преодолеть разрыв между новыми и старыми решениями по проектированию инфраструктуры. По крайней мере, мы на это надеемся!

Организация книги

Эта книга состоит из 15 глав:

- глава 1. Дает общий обзор Kubernetes для новичков;
- глава 2. Рассматривает идею модуля Pod как атомарного строительного блока для приложений и закладывает основы для последующих глав, подробно рассматривающих низкоуровневые детали Linux;
- глава 3. Углубляется в детали использования в Kubernetes низкоуровневых примитивов Linux для реализации концепций более высокого уровня, включая модули Pod;
- глава 4. Здесь начинается изучение внутренних особенностей процессов и их изоляции в Linux, которые являются одними из менее известных деталей ландшафта Kubernetes;
- глава 5. После подробного знакомства с модулями Pod углубляется в организацию сетевых взаимодействий между ними и рассказывает, как они соединяются друг с другом через разные узлы;

- глава 6. Вторая глава, посвященная сетевым взаимодействиям, описывающая более широкие аспекты работы модулей Pod и прокси-сервера (kube-proxy), а также приемы их настройки и сопровождения;
- глава 7. Первая глава, посвященная вопросам организации хранилища. Здесь дается широкое введение в теоретические основы хранилищ Kubernetes, контейнерный интерфейс хранилища (Container Storage Interface, CSI) и его взаимодействия с kubelet;
- глава 8. Вторая глава, посвященная вопросам организации хранилища. Здесь рассматриваются некоторые более практические аспекты хранения данных, включая особенности emptyDir, Secrets и PersistentVolumes/Dynamic storage;
- глава 9. Углубляется в kubelet и рассматривает некоторые детали запуска модулей Pod и управления ими, включая такие понятия, как CRI, жизненный цикл узла и ImagePullSecrets;
- глава 10. DNS в Kubernetes – сложный механизм, используемый почти всеми контейнерными приложениями для локального доступа к внутренним сервисам. Здесь рассматривается CoreDNS – реализация сервиса DNS для Kubernetes – и порядок выполнения DNS-запросов разными модулями Pod;
- глава 11. Подробно обсуждает плоскость управления, упоминавшуюся в предыдущих главах, включая тонкости работы планировщика, диспетчера контроллеров и сервера API. Они образуют «мозг» Kubernetes и объединяют вместе все низкоуровневые концепции, обсуждавшиеся в предыдущих главах;
- глава 12. После обсуждения логики плоскости управления мы углубимся в etcd, надежный механизм консенсуса для Kubernetes, и особенности его настройки для удовлетворения потребностей плоскости управления Kubernetes;
- глава 13. Представляет обзор NetworkPolicies, RBAC и безопасности на уровне модулей Pod и узлов, о которых должен знать каждый администратор. В этой главе также обсуждается общее развитие политик безопасности модулей Pod;
- глава 14. Здесь рассматривается настройка безопасности на уровне узла и облака, а также другие аспекты безопасности Kubernetes, ориентированные на инфраструктуру;
- глава 15. Эта заключительная глава дает общий обзор прикладных инструментов на примере Carvel, набора инструментов для управления файлами YAML, создания приложений и долгосрочного управления жизненным циклом приложений.

О примерах программного кода

Для этой книги мы подготовили несколько примеров, которые вы найдете в репозитории GitHub (<https://github.com/jayunit100/k8sprototypes/>), в том числе примеры:

- использования kind для настройки реалистичной сети в локальных кластерах с помощью Calico, Antrea или Cilium;
- анализа метрик Prometheus в реальном мире;
- создания приложений с помощью набора инструментов Carvel;
- различных экспериментов, связанных с RBAC.

Эта книга также содержит множество примеров программного кода. Они оформлены шрифтом фиксированной ширины, чтобы вам было проще отличать его от основного текста.

Во многих случаях исходный код переформатирован, чтобы уместить его по ширине книжной страницы. В частности, мы добавили разрывы строк и отступы. В редких случаях даже этого было недостаточно, и мы добавили маркеры продолжения строки (↪). Многие листинги сопровождаются комментариями в тексте, подчеркивающими важные понятия. Получить выполняемые фрагменты кода можно из электронной версии книги по адресу <https://livebook.manning.com/book/core-kubernetes> и в репозитории GitHub <https://github.com/jayunit100/k8sprototypes/>.

Отзывы и пожелания

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

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

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

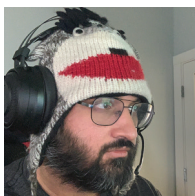
Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно от-

носятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторах



Джей Вьяс, д-р наук (Jay Vyas, PhD), в настоящее время – штатный инженер в VMware. Работал над несколькими коммерческими дистрибутивами и платформами Kubernetes с открытым исходным кодом, включая OpenShift, VMware Tanzu, внутренними коллективными платформами Black Duck для Kubernetes и установкой Kubernetes для клиентов его

консалтинговой компании Rocket Rudolf, LLC. В течение нескольких лет был членом комитета по управлению проектами (Project Management Committee, PMC) в Apache Software Foundation, где работал над несколькими проектами в области больших данных. Он был связан с Kubernetes на различных должностях с момента его создания и в настоящее время уделяет большое внимание сообществам SIG-Windows и SIG-Network. Начинал с создания распределенных систем, одновременно защитив докторскую диссертацию по витринам данных в сфере биоинформатики (объединявшим базы данных в платформы для анализа человеческих и вирусных белковых карт – протеомов). Это привело его в мир больших данных и масштабируемых систем обработки данных и, наконец, в Kubernetes.

Связаться с Джейем можно по адресу @jayunit100 в Твиттере, если вы заинтересованы в сотрудничестве... по какой угодно теме. Уделяет большое внимание спорту, ежедневно пробегает одну милю в спринтерском темпе и подтягивается до отказа. Также увлекается музыкой и имеет несколько синтезаторов, в том числе Prophet-6, который звучит как космический корабль.



Крис Лав (Chris Love) – сертифицированный сотрудник Google Cloud и соучредитель Lionkube. Больше 25 лет занимался разработкой программного обеспечения в таких компаниях, как Google, Oracle, VMware, Cisco, Johnson & Johnson и др. Как идейный лидер в Kubernetes и в сообществе DevOps, Крис Лав участвовал во многих проектах с открытым исход-

ным кодом, включая Kubernetes, kops (в должности руководителя AWS SIG), Bazel (внес вклад в разработку правил для Kubernetes) и Terraform (один из первых разработчиков плагина VMware). В число его профессиональных интересов входят: трансформация ИТ-культуры, технологии контейнеризации, методы и средства автоматизированного тестирования, Kubernetes, Golang (он же Go) и другие языки программирования. Лав обожает заниматься популяризацией DevOps, Kubernetes и технологий, а также обучать людей в сфере ИТ и программного обеспечения.

Вне работы любит кататься на лыжах, играть в волейбол, заниматься йогой и участвовать в мероприятиях на свежем воздухе. Кроме того, вот уже 20 лет занимается боевыми искусствами.

Если вы решите пообщаться с Крисом и задать ему свои вопросы, то сможете связаться с ним по адресу @chrislovenm в Twitter или LinkedIn.

Об иллюстрации на обложке

Изображение на обложке книги называется «Штерн, матрос у руля корабля». Оно взято с гравюры картины Альфредо Луксоро (Alfredo Luxoro), опубликованной в журнале «L'Illustrazione Italiana», № 19, от 9 мая 1880 года.

В те дни по одежде было легко определить, где живет человек, чем занимается и какое положение занимает в обществе. Мы в издательстве Manning славим изобретательность, предприимчивость и радость компьютерного бизнеса обложками книг, изображающими богатство региональных различий многовековой давности, оживших благодаря иллюстрациям, таким как эта.

Почему появился Kubernetes

В этой главе:

- почему появился Kubernetes;
- основные термины Kubernetes;
- конкретные примеры использования Kubernetes;
- высокоуровневые функции Kubernetes;
- когда нежелательно использовать Kubernetes.

Kubernetes – это платформа с открытым исходным кодом для размещения контейнеров и определения прикладных API для управления облачной семантикой обеспечения этих контейнеров хранилищами данных, сетевыми услугами, поддержкой безопасности и другими ресурсами. Kubernetes обеспечивает непрерывную синхронизацию всего пространства состояний ваших приложений, в том числе способов доступа к ним из внешнего мира.

Зачем внедрять Kubernetes в свое окружение? Не проще ли выделить все необходимые ресурсы вручную с помощью инструмента управления инфраструктурой, связанного с DevOps? Ответ зависит от того, как мы определяем процесс DevOps и его интеграцию в общий жизненный цикл приложения. DevOps продолжает расширяться и включает в себя процессы, инженеров и инструменты, которые поддерживают более автоматизированное администрирование приложений в центре обработки данных. Одно из условий успешного

решения этой задачи – воспроизводимость инфраструктуры: изменение, внесенное для устранения инцидента в одном компоненте, которое не воспроизводится полностью во всех других идентичных компонентах, означает, что один или несколько компонентов отличаются.

В этой книге мы подробно рассмотрим передовые практики использования Kubernetes в DevOps, обеспечивающие репликацию компонентов по мере необходимости и уменьшение частоты сбоев системы. Мы также исследуем внутренние процессы, чтобы лучше понять, как работает Kubernetes и как с его применением получить максимально эффективную систему.

1.1 Предварительный обзор некоторых основных терминов

В 2021 году Kubernetes стала одной из наиболее широко используемых облачных технологий. Из-за этого мы не всегда полностью определяем новые термины, прежде чем ссылаться на них. Для новичков в Kubernetes или недостаточно хорошо знакомых с некоторыми терминами далее мы приводим несколько ключевых определений, к которым вы можете периодически обращаться, читая несколько первых глав этой книги и осваивая эту новую вселенную. Мы определим эти понятия более подробно и в более широком контексте, когда углубимся в них позже в этой книге:

- *CNI (Container Networking Interface)* и *CSI (Container Storage Interface)* – сетевой интерфейс контейнеров и интерфейс хранилища для контейнеров соответственно; позволяют подключать к сетям и хранилищам модули Pod (с контейнерами), работающие в Kubernetes;
- *контейнер (Container)* – образ Docker или OCI (Open Container Initiative), который обычно запускает приложение;
- *плоскость управления (Control plane)* – мозг кластера Kubernetes, осуществляющий планирование контейнеров и управляющий всеми объектами Kubernetes (которые иногда называют мастер-объектами);
- *набор демонов (DaemonSet)* – аналог развертывания (Deployment), но выполняется на каждом узле кластера;
- *развертывание (Deployment)* – набор модулей, которыми управляет Kubernetes;
- *kubectl* – инструмент командной строки для взаимодействия с панелью управления Kubernetes;
- *kubelet* – агент Kubernetes, работающий на узлах кластера. Обеспечивает поддержку плоскости управления;
- *узел (Node)* – машина, на которой запущен процесс kubelet;

- *OCI (Open Container Initiative)* – общий формат образа для создания выполняемых автономных приложений. Также называется *образами Docker*;
- *Pod (модуль)* – объект Kubernetes, инкапсулирующий действующий контейнер.

1.2 Проблема дрейфа инфраструктуры и Kubernetes

Управление инфраструктурой – это воспроизводимый способ управления «дрейфом» конфигурации этой инфраструктуры по мере изменения аппаратного обеспечения, нормативов и других требований, действующих в центре обработки данных. Это относится и к *определению* приложений, и к *управлению* хостами, на которых выполняются эти приложения. ИТ-инженеры слишком хорошо знакомы с типичными проблемами, такими как:

- обновление версии Java на нескольких серверах;
- выявление причин отказа некоторых приложений в определенных местах;
- замена или масштабирование старого или сломанного оборудования и перенос приложений;
- ручное управление маршрутами для балансировки нагрузки;
- отсутствие описания новых изменений инфраструктуры в документации из-за отсутствия общего обязательного языка конфигурации.

В процессе управления и обновления серверов в центре обработки данных или в облаке увеличивается вероятность «отклонения» их исходных определений от предполагаемой архитектуры. Приложения могут запускаться в неправильных местах, с неправильным набором ресурсов или с доступом к неправильным хранилищам.

Kubernetes дает возможность централизовать управление пространством состояний всех приложений с использованием одного удобного инструмента: `kubectl` (<https://kubernetes.io/docs/tasks/tools/>) – клиента командной строки, выполняющего вызовы REST API к серверу Kubernetes API. Также есть возможность использовать клиентов Kubernetes API для выполнения этих же задач программно. Установить `kubectl` и протестировать его в кластере довольно просто, что мы и сделаем в начале этой книги.

Предыдущие подходы к управлению сложным пространством состояний приложений основывались на таких технологиях, как Puppet, Chef, Mesos, Ansible и SaltStack. Kubernetes заимствовал лучшие черты этих подходов и использует возможности управления состоянием таких инструментов, как Puppet, а также идеи некоторых приложений

и примитивов планирования, предоставляемых таким программным обеспечением, как Mesos.

Ansible, SaltStack и Terraform играли важную роль в настройке инфраструктуры (определяя требования, специфичные для ОС, такие как брандмауэры или установка двоичных файлов). Kubernetes тоже поддерживает эту идею, но использует *привилегированные контейнеры* в среде Linux (в Windows v1.22 они известны как *HostProcess Pods*). Например, привилегированный контейнер в системе Linux может управлять правилами iptables для организации маршрутизации трафика к приложениям, что, собственно, и делает прокси-сервер Kubernetes Service (известный как *kube-proxy*).

Google, Microsoft, Amazon, VMware и многие другие компании взяли на вооружение контейнеризацию как основную стратегию, позволяющую клиентам запускать сотни и тысячи приложений в различных облачных окружениях и окружениях без системного программного обеспечения. Соответственно, контейнеры оказываются фундаментальным примитивом *и* для запуска приложений, *и* для управления инфраструктурой (например, для предоставления контейнерам IP-адресов), которые запускают сервисы, необходимые приложениям (например, специализированные хранилища или брандмауэры с определенными настройками), и, что особенно важно, сами приложения.

Kubernetes (на момент написания этой книги) практически бесспорно считается современным стандартом для организации и запуска контейнеров в любом облачном окружении, на сервере или в центре обработки данных.

1.3 Контейнеры и образы

Приложения имеют зависимости, которые должны удовлетворяться хостом, на котором они выполняются. В доконтейнерную эпоху разработчики решали эту задачу в произвольном порядке (например, приложению Java требовалось действующая виртуальная машина JVM вместе с настроенным брандмауэром, поддерживающим возможность доступа к базе данных).

По своей сути Docker можно рассматривать как способ запуска контейнеров, где *контейнер* – это работающий образ OCI (<https://github.com/opencontainers/image-spec>). Спецификация OCI – это стандартный способ определения образа, который может быть запущен такой программой, как Docker, и в конечном счете представляет собой архив с различными слоями. Слои в архиве образа содержат такие компоненты, как двоичные файлы Linux и файлы приложений. Соответственно, когда вы запускаете контейнер, среда выполнения контейнеров (такая как Docker, containerd или CRI-O) берет образ, распаковывает его и запускает процесс в хост-системе, который, в свою очередь, запускает содержимое образа.

Контейнеры добавляют слой изоляции, устраняющий необходимость управления библиотеками на сервере или предварительной загрузки инфраструктуры другими зависимостями приложений (рис. 1.1). Например, если есть два приложения Ruby, которым требуются разные версии одной и той же библиотеки, то можно использовать два контейнера. Каждое приложение Ruby будет изолировано внутри своего контейнера и использовать определенную версию библиотеки.

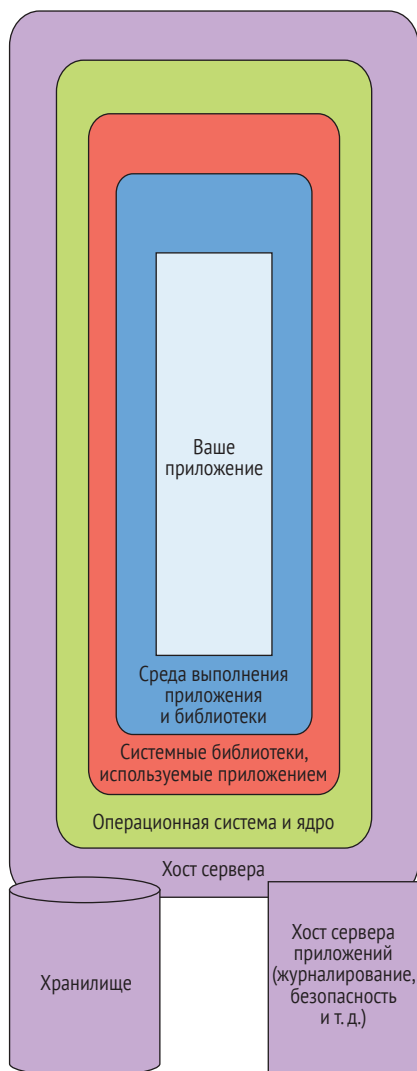


Рис. 1.1 Приложения, выполняющиеся в контейнерах

Разработчикам хорошо известна фраза: «Ну, это работает на моей машине». Программное обеспечение часто может работать в одном окружении или на одном компьютере, но не работать в другом. Обра-

зы упрощают запуск одного и того же программного обеспечения на разных серверах. Подробнее об образах и контейнерах мы поговорим в главе 3.

Объединение образов с платформой Kubernetes, позволяющей запускать неизменяемые серверы, дает небывалый уровень простоты. Поскольку контейнеры быстро становятся отраслевым стандартом развертывания программных приложений, стоит привести некоторые цифры:

- по результатам опроса более 88 000 разработчиков, *Docker* и *Kubernetes* заняли третье место среди самых популярных технологий разработки в 2020 году, сразу после *Linux* и *Docker* (<http://mng.bz/nY12>);
- служба *Datadog* недавно обнаружила, что *Docker* охватывает 50 или более процентов рабочего процесса среднего разработчика. Аналогично более 25 % компаний полностью перешли на использование *Docker* (<https://www.datadoghq.com/docker-adoption/>).

Однако использование контейнеров немислимо без автоматизации, и именно этой цели служит *Kubernetes*. *Kubernetes* доминирует в пространстве контейнеров так же, как доминировали в своих сферах СУБД *Oracle* и платформа виртуализации *vSphere* во времена расцвета. Спустя годы базы данных *Oracle* и *vSphere* все еще пользуются большой популярностью; такое же долголетие мы прогнозируем и для *Kubernetes*.

Мы начнем эту книгу знакомством с базовыми особенностями *Kubernetes*. Ее цель в том, чтобы вывести вас за пределы базовых принципов и познакомить с низкоуровневым ядром. Давайте начнем наше погружение и посмотрим на чрезвычайно упрощенный рабочий процесс *Kubernetes* (также называемый «K8s»), демонстрирующий, как некоторые из пользователей создают и запускают микросервисы.

1.4 Базовая основа Kubernetes

Все сущее в *Kubernetes* определяется в виде простых текстовых файлов в формате *YAML* или *JSON*, и платформа запускает образы ОСИ декларативным способом. Тот же подход (с текстовыми файлами в формате *YAML* или *JSON*) можно использовать для настройки сетевых правил, аутентификации и авторизации на основе ролей (*RBAC*) и т. д. Изучив один синтаксис и его структуру, можно настраивать, управлять и оптимизировать любые системы *Kubernetes*.

Давайте рассмотрим короткий пример запуска простого приложения в *Kubernetes*. Не волнуйтесь, если что-то покажется вам непонятным; далее в книге мы рассмотрим еще множество реальных примеров, которые проведут нас через весь жизненный цикл приложения. Считайте, что это всего лишь наглядная демонстрация наших пассов руками, которые мы делали до сих пор. Начнем с конкретного примера – микросервиса. Следующий фрагмент кода генерирует файл *Dockerfile*, который определяет образ для запуска *MySQL*:

```
FROM alpine:3.15.4
RUN apk add --no-cache mysql
ENTRYPOINT ["/usr/bin/mysqld"]
```

Обычно этот образ создается (с помощью `docker build`) и сохраняется (с помощью, например, `docker push`) в *реестре OCI* (место, где образ может храниться и извлекаться контейнером в момент запуска). Общедоступный реестр с открытым исходным кодом, в котором вы можете размещать свои образы, доступен по адресу <https://github.com/goharbor/harbor>. Еще один похожий реестр, тоже широко используемый для хранения миллионов образов приложений, находится по адресу <https://hub.docker.com/>. Для целей нашего примера предположим, что мы отправили образ в реестр и теперь запускаем его. Нам также может понадобиться создать контейнер для взаимодействия с этой службой (например, с приложением на Python, которое используется базой данных MySQL). Мы могли бы определить его образ Docker так:

```
FROM python:3.7
WORKDIR /myapp
COPY src/requirements.txt ./
RUN pip install -r requirements.txt
COPY src /myapp
CMD [ "python", "mysql-custom-client.py" ]
```

Чтобы запустить клиента и сервер MySQL в виде контейнеров в окружении Kubernetes, нужно создать два объекта типа Pod. Каждый из них может выполнять один из контейнеров, например:

```
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
  spec:
    containers:
      - name: my-mysql-server
        image: myregistry.com/mysql-server:v1.0
---
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s-mysql
  spec:
    containers:
      - name: my-sqlclient
        image: myregistry.com/mysql-custom-client:v1.0
        command: ['tail', '-f', '/dev/null']
```

Обычно такие фрагменты YAML сохраняются в текстовых файлах (например, `myapp.yaml`) и выполняются с помощью клиента Kubernetes (например, `kubectl create -f my-app.yaml`). Этот инструмент под-

ключается к серверу Kubernetes API и передает определение в формате YAML для сохранения. Затем Kubernetes автоматически извлекает определения двух модулей Pod, имеющиеся на сервере API, и проверяет, где они должны быть запущены.

Это происходит не мгновенно, потому что узлам в кластере требуется время, чтобы среагировать на постоянно происходящие события и обновить состояние своих объектов Node через агента kubelet, взаимодействующего с сервером API. Также требуется, чтобы образы ОС присутствовали и были доступны для узлов в кластере Kubernetes. В любой момент может что-то пойти не так, поэтому мы называем Kubernetes системой, *стабильной в конечном счете*, в которой согласование желаемого состояния с течением времени является ключевой философией дизайна. Эта модель согласованности (по сравнению с моделью гарантированной согласованности) обеспечивает возможность постоянного мониторинга изменений в общем пространстве состояний всех приложений в кластере и позволяет платформе Kubernetes определять, как эти приложения приводятся в движение с течением времени.

Этот подход естественным образом укладывается в сценарии реального мира. Например, если вам нужно, чтобы «пять приложений были распределены по трем зонам в облаке», этого легко добиться, определив несколько строк YAML с примитивами планирования Kubernetes. Конечно, вам придется гарантировать существование этих трех зон и их доступность для планировщика, но, даже если вы этого не сделаете, Kubernetes, по крайней мере, запланирует некоторые рабочие нагрузки в доступных зонах.

Проще говоря, Kubernetes позволяет определить желаемое состояние всех приложений в кластере, их подключение к сети, место работы, используемое хранилище и т. д., делегируя базовую реализацию этих деталей самой платформе Kubernetes. Соответственно, вам редко придется выполнять однократное обновление Ansible или Puppet в сценарии промышленного использования Kubernetes (если только вы не переустанавливаете саму платформу Kubernetes, но даже в этом случае можно воспользоваться такими инструментами, как Cluster API, позволяющими управлять платформой Kubernetes с помощью самой Kubernetes (как бы не запутаться во всем этом)).

1.4.1 Все инфраструктурные правила в Kubernetes определяются в обычных файлах YAML

Kubernetes автоматизирует все аспекты стека с помощью Kubernetes API, которым можно полностью управлять как ресурсами YAML и JSON. К их числу относятся традиционные правила инфраструктуры (которые так или иначе применяются к микросервисам), такие как:

- конфигурация портов или IP-маршрутов;
- постоянная доступность хранилища для приложений;

- размещение программного обеспечения на определенных или произвольных серверах;
- обеспечение безопасного доступа приложений друг к другу с использованием, например, RBAC или сетевых правил;
- конфигурация DNS для каждого приложения и глобально.

Все эти компоненты определяются в конфигурационных файлах, представляющих объекты в Kubernetes API. Kubernetes использует эти стандартные блоки и контейнеры, применяет изменения, отслеживает эти изменения и устраняет сбои или нарушения, пока не будет достигнуто желаемое конечное состояние. Когда «ночью что-то идет не так», Kubernetes автоматически обрабатывает множество сценариев и избавляет нас от решения проблем вручную. Правильная настройка сложных систем с применением средств автоматизации позволяет команде DevOps сосредоточиться на решении важных задач, планировать будущее и находить лучшие в своем классе решения для бизнеса. Давайте далее рассмотрим некоторые возможности, предлагаемые платформой Kubernetes для поддержки модулей Pod.

1.5 Возможности Kubernetes

Платформы оркестрации контейнеров позволяют разработчикам автоматизировать процесс запуска экземпляров, подготовки хостов, связывания контейнеров для оптимизации процедур оркестрации и продления жизненных циклов приложений. Далее мы перечислим основные возможности платформы оркестрации контейнеров, потому что контейнерам нужны модули Pod, а модулям Pod нужна платформа Kubernetes для:

- предоставления доступа, не зависящего от используемой облачной технологии, ко всем возможностям сервера API;
- интеграции со всеми основными облачными платформами и гипервизорами в диспетчере контроллеров Kubernetes (Kubernetes Controller Manager, KCM);
- обеспечения отказоустойчивости для хранения и определения состояния всех сервисов, приложений и конфигураций центров обработки данных или других инфраструктур, поддерживаемых Kubernetes;
- управления развертыванием, чтобы минимизировать время простоя отдельных узлов, сервисов или приложений;
- автоматизации масштабирования хостов и приложений с поддержкой постоянного обновления;
- создания внутренних и внешних соединений (известных как типы ClusterIP, NodePort или LoadBalancer Service) с балансировкой нагрузки;
- предоставления возможности планирования запуска приложений на определенном виртуализированном оборудовании на

основе его метаданных с помощью маркировки узлов и планировщика Kubernetes;

- обеспечения высокой доступности с помощью DaemonSets и других технологических инфраструктур, в которых приоритет отдается контейнерам, работающим на всех узлах кластера;
- обнаружения сервисов через службу доменных имен (Domain Name Service, DNS), ранее реализованную как KubeDNS, а совсем недавно – CoreDNS, которая интегрируется с сервером API;
- запуска пакетных процессов (известных как задания), которые используют хранилище и контейнеры так же, как обычные приложения;
- расширения API и создания собственных программ, управляемых API, с помощью пользовательских определений ресурсов и без создания каких-либо сопоставлений портов или подключений;
- проверки сбойных процессов на уровне кластера, включая удаленное выполнение в любом контейнере в любое время с помощью `kubectl exec` и `kubectl describe`;
- подключения локального и/или удаленного хранилища к контейнеру и декларативного управления томами хранилища с помощью StorageClass API и PersistentVolumes.

На рис. 1.2 показана схема простого кластера Kubernetes. То, что делает Kubernetes, отнюдь не тривиально. Она стандартизирует управление жизненным циклом нескольких приложений, работающих в одном кластере. Основой Kubernetes является кластер, состоящий из узлов. Сложность Kubernetes – это, по общему признанию, одна из претензий, предъявляемых инженерами к Kubernetes. Сообщество усиленно работает над тем, чтобы сделать платформу проще, но вы должны понимать, что Kubernetes решает сложные задачи, которые нельзя реализовать просто с первой попытки.

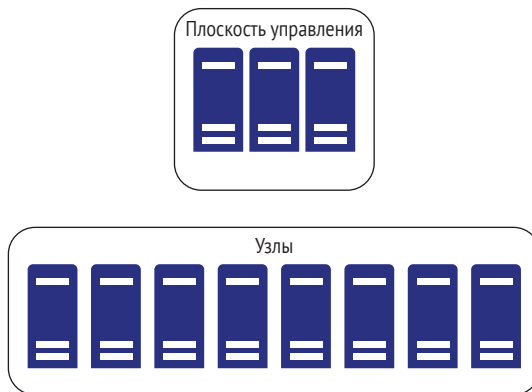


Рис. 1.2 Пример кластера Kubernetes

Если вам не нужны высокая доступность, масштабируемость и оркестрация, то, возможно, вам не нужна Kubernetes. Теперь рассмотрим типичный сценарий сбоя в кластере.

- 1 Узел перестает отвечать плоскости управления.
- 2 Плоскость управления перепланирует запуск модулей Pod, работавших на отключившемся узле, на другом узле или узлах.
- 3 Когда пользователь вызывает API сервера через `kubectl`, сервер сообщает об отключившемся узле и новом местоположении модулей Pod.
- 4 Все клиенты, взаимодействующие с сервисом в модуле Pod, переадресуются в новое местоположение.
- 5 Тома хранилища, подключенные к модулям Pod на неисправном узле, подключаются к новому местоположению модуля Pod, благодаря чему прежние данные остаются доступными для чтения.

Цель этой книги – дать более глубокое понимание особенностей работы основных механизмов и показать, как базовые примитивы Linux дополняют высокоуровневые компоненты Kubernetes для решения многих задач. Kubernetes опирается на сотни технологий в стеке Linux, которые часто трудно освоить и которым не хватает подробной документации. Мы надеемся, что, прочитав эту книгу, вы поймете многие тонкости Kubernetes, часто упускаемые из виду в учебных пособиях, описывающих приемы запуска контейнеров и управление ими.

Обычно Kubernetes запускается поверх неизменяемых операционных систем, когда имеется базовая ОС, обновляемая только при обновлении всей ОС (и, следовательно, является неизменной), и вы устанавливаете свои узлы/Kubernetes, используя эту ОС. Неизменяемая ОС дает много преимуществ, которые мы не будем рассматривать здесь. Вы можете запускать Kubernetes в облаке, на физических серверах или даже на Raspberry Pi. Более того, в настоящее время Министерство обороны США исследует возможность запуска Kubernetes на некоторых своих истребителях. А компания IBM реализовала поддержку запуска кластеров на своих мейнфреймах PowerPC следующего поколения.

По мере развития облачной экосистемы, окружающей Kubernetes, она будет продолжать позволять организациям находить лучшие приемы, активно вносить изменения для предотвращения проблем и поддерживать согласованность окружения, чтобы избежать дрейфа, когда некоторые машины ведут себя немного иначе, чем другие, из-за того, что на них какие-то исправления не применялись или применялись неправильно.

1.6 Компоненты и архитектура Kubernetes

Теперь рассмотрим архитектуру Kubernetes на высоком уровне (рис. 1.3). Если вкратце, то она включает ваше оборудование и ту часть вашего оборудования, на котором выполняются плоскость управления и рабочие узлы Kubernetes:

- *аппаратная инфраструктура* – включает компьютеры, сетевую инфраструктуру, инфраструктуру хранения и реестр контейнеров;
- *рабочие узлы Kubernetes* – базовая вычислительная единица в кластере Kubernetes;
- *плоскость управления Kubernetes* – основа Kubernetes. Она включает сервер API, планировщика, диспетчера контроллеров и другие контроллеры.

1.6.1 Kubernetes API

Самое важное, что можно вынести из этой главы и о чем следует помнить на протяжении чтения всей книги, – это то, что администрирование микросервисов и других контейнерных приложений на платформе Kubernetes сводится к объявлению объектов Kubernetes API. Все остальное делается автоматически самой платформой.

В этой книге мы подробно рассмотрим сервер API и его хранилище данных etcd. Почти все, что можно попросить `kubectl` сделать, сводится к чтению или записи в определенный и версионированный объект на сервере API. (Исключением являются такие операции, как использование `kubectl` для сбора журналов из действующего модуля Pod, соединение с которым проксируется через узел.) Сервер Kubernetes API – `kube-apiserver` – позволяет выполнять CRUD-операции (Create, Read, Update, Delete – создавать, читать, обновлять, удалять) со всеми объектами и предоставляет интерфейс передачи репрезентативного состояния RESTful (REpresentational State Transfer). Некоторые команды `kubectl`, такие как `describe`, возвращают комбинированное представление нескольких объектов. Как правило, все объекты Kubernetes API имеют:

- именованную версию API (например, `v1` или `rbac.authorization.k8s.io/v1`);
- тип (например, `kind: Deployment`);
- раздел метаданных.

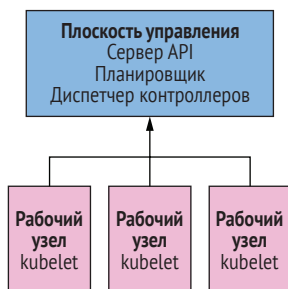


Рис. 1.3 Плоскость управления и рабочие узлы

Мы можем поблагодарить Брайана Гранта (Brian Grant), одного из первых основателей Kubernetes, за предложенную им схему управления версиями API, которая на деле доказала свою надежность. Такая

организация может показаться сложной и, честно говоря, иногда немного неудобной, но она позволяет производить обновления и устанавливать контракты, определяющие изменения в API. Изменения и миграция API часто нетривиальны, и Kubernetes предоставляет четко определенный контракт для этого. Взгляните на документы, описывающие особенности версионирования API на веб-сайте Kubernetes (<http://mng.bz/voP4>), где можно найти описание контрактов для версий API Alpha, Beta и GA.

Далее в этой книге мы сосредоточимся на Kubernetes, но постоянно будем возвращаться к основной теме: практически все в Kubernetes направлено на поддержку модулей Pod. В частности, мы подробно рассмотрим несколько элементов API:

- модули Pod с окружением времени выполнения и развертывания;
- детали реализации API;
- Ingress Services и балансировку нагрузки;
- хранилища PersistentVolumes и PersistentVolumeClaims;
- сетевые политики и сетевую безопасность.

Существует около 70 различных типов API, с которыми вы можете экспериментировать, создавая, изменяя и удаляя соответствующие ресурсы в стандартном кластере Kubernetes. Вы можете просмотреть их командой `kubectl api-resources`, вывод которой выглядит примерно так:

```
$ kubectl api-resources | head
```

NAME	SHORTNAMES	NAMESPACED	KIND
bindings		true	Binding
componentstatuses	cs	false	ComponentStatus
configmaps	cm	true	ConfigMap
endpoints	ep	true	Endpoints
events	ev	true	Event
limitranges	limits	true	LimitRange
namespaces	ns	false	Namespace
nodes	no	false	Node
persistentvolumeclaims	pvc	true	PersistentVolumeClaim

Как видите, каждый ресурс в Kubernetes API имеет:

- краткое имя;
- полное имя;
- признак ограничения пространством имен.

Пространства имен в Kubernetes позволяют объектам в файле существовать внутри определенного... эм-м... пространства имен. Это дает разработчикам простую форму иерархической группировки. Например, для приложения, запускающего 10 различных микросервисов, можно создать все его модули Pod, сервисы Service и запросы PersistentVolumeClaims (также называемые PVC) внутри одного пространства имен. При такой организации, когда придет время удалить приложение, можно просто удалить пространство имен. В главе 15 мы рассмотрим более высокоуровневые и более сложные способы ана-

лиза и организации жизненного цикла приложений. Но во многих случаях пространства имен являются наиболее очевидным и интуитивно понятным средством разделения объектов Kubernetes API, связанных с приложениями.

1.6.2 Пример первый: интернет-магазин

Представьте крупный интернет-магазин, который должен иметь возможность быстро масштабироваться в соответствии с колебаниями спроса, например, в праздничные дни. Масштабирование и прогнозирование масштабирования всегда были одной из их самых больших проблем – возможно, самой большой. Kubernetes берет на себя решение многих проблем, связанных с созданием высокодоступной и масштабируемой распределенной системы. Представьте, что у вас всегда под рукой имеются возможности масштабирования, распространения и создания высокодоступных систем. Это не только лучший способ ведения бизнеса, но и наиболее эффективная и действенная платформа для управления системами. Сочетание Kubernetes и облачных услуг позволяет задействовать чужие серверы, когда возникает потребность в дополнительных ресурсах, вместо покупки и обслуживания дополнительного оборудования на всякий случай.

1.6.3 Пример второй: онлайн-решение для благотворительности

Вторым примером из реальной жизни, о котором стоит упомянуть, может служить веб-сайт, позволяющий передавать пожертвования благотворительным организациям по выбору пользователя. Допустим, что этот конкретный сайт изначально был основан на WordPress, но с течением времени бизнес-транзакции привели к полномасштабной зависимости от фреймворков JVM (таких как Grails) с настраиваемым пользовательским интерфейсом, промежуточным уровнем и уровнем базы данных. Требования для этого бизнес-цунами включали машинное обучение, показ рекламы, обмен сообщениями, Python, Lua, NGINX, PHP, MySQL, Cassandra, Redis, Elastic, ActiveMQ, Spark, львов, тигров и медведей... да остановитесь вы уже.

Первоначальная инфраструктура была организована как созданная вручную облачная виртуальная машина, использующая Puppet для настройки всего и вся. Проект предусматривал возможность масштабирования с ростом компании, но для этого требовалось все больше и больше виртуальных машин, на которых размещались одно-два приложения. Тогда владельцы сайта решили перейти на Kubernetes. Количество виртуальных машин уменьшилось примерно с 30 до 5, и их стало легче масштабировать. Благодаря переходу на Kubernetes они полностью устранили Puppet и настройку сервера, а значит, и необходимость вручную управлять инфраструктурой.

Перейдя на Kubernetes, этой компании удалось решить целый класс проблем, связанных с администрированием виртуальных машин, нагрузкой на DNS из-за публикации сложных сервисов, и многие другие. Кроме того, время восстановления системы в случае катастрофических сбоев оказалось гораздо более предсказуемым с точки зрения инфраструктуры. Почувствовав преимущества перехода на надежную и стандартизированную методологию на основе API, способную быстро вносить массовые изменения, вы начинаете ценить декларативный характер Kubernetes и его облачный подход к оркестрации контейнеров.

1.7 Когда не стоит использовать Kubernetes

Следует признать, что в некоторых случаях Kubernetes оказывается не лучшим выбором. Вот некоторые из них:

- *высокопроизводительные вычисления (High-Performance Computing, HPC)* – контейнеры добавляют дополнительные сложности, а наличие нового уровня бьет по производительности. С развитием контейнерных технологий задержки, создаваемые контейнерами, постепенно уменьшаются, но если ваше приложение считает каждую нано- или микросекунду, то использование Kubernetes может оказаться не лучшим вариантом;
- *унаследованные приложения* – некоторые приложения имеют требования к оборудованию, программному обеспечению и поддержке, что затрудняет их контейнеризацию. Например, у вас могут быть приложения, приобретенные у компании-разработчика программного обеспечения, которые официально не поддерживают работу в контейнере или в кластере Kubernetes;
- *миграция* – реализации унаследованных систем могут быть настолько жесткими, что их миграция в Kubernetes не дает особых преимуществ, кроме возможности громко заявить: «Мы используем Kubernetes». Некоторые из наиболее значительных преимуществ достигаются только после миграции, когда монолитные приложения разбиваются на логические компоненты, способные масштабироваться независимо друг от друга.

Поэтому изучайте основы и овладевайте ими. Kubernetes решает многие проблемы, описанные в этой главе, надежно и недорого.

Итоги

- Kubernetes делает жизнь проще!
- Платформа Kubernetes может работать в инфраструктуре любого типа.

- Kubernetes создает экосистему компонентов, работающих вместе. Объединение компонентов позволяет компаниям предотвращать сбои, восстанавливать и масштабировать системы в режиме реального времени, когда требуются срочные изменения.
- Все, что делается в Kubernetes, можно сделать с помощью одного простого инструмента: `kubectl`.
- Kubernetes создает кластер из одного или нескольких компьютеров и использует его как платформу для развертывания и размещения контейнеров. Kubernetes обеспечивает оркестрацию контейнеров, управление хранилищами и распределенную сеть.
- Платформа Kubernetes родилась на основе предыдущих подходов, основанных на конфигурации и контейнерах.
- Pod – это основной строительный блок Kubernetes. Поддержка модулей Pod предлагает множество возможностей: масштабирование, обработку отказов, поиск DNS и обеспечение безопасности на основе правил RBAC.
- Приложения Kubernetes управляются простыми обращениями к серверу Kubernetes API.

2

Зачем нужны модули Pod?

В этой главе:

- что такое Pod?
- пример веб-приложения и зачем нужны модули Pod;
- как Kubernetes поддерживает модули Pod;
- плоскость управления Kubernetes.

В предыдущей главе мы в общих чертах познакомились с платформой Kubernetes, ее возможностями, основными компонентами и архитектурой. Мы также увидели пару примеров использования Kubernetes в бизнесе и некоторые определения контейнеров. Абстракция Pod в Kubernetes, предназначенная для гибкого запуска тысяч контейнеров, стала фундаментальной частью перехода предприятий на использование контейнеров. В этой главе мы расскажем о модулях Pod и о том, как Kubernetes поддерживает их в роли основного строительного блока приложений.

Как кратко упоминалось в главе 1, Pod – это объект в Kubernetes API, как и большинство ресурсов в Kubernetes. Pod – это наименьшая атомарная единица, которую можно развернуть в кластере Kubernetes, и сама платформа Kubernetes построена на основе определения Pod. Определение Pod (рис. 2.1) описывает модуль, способный включать несколько контейнеров, что позволяет Kubernetes создать несколько контейнеров на узле.

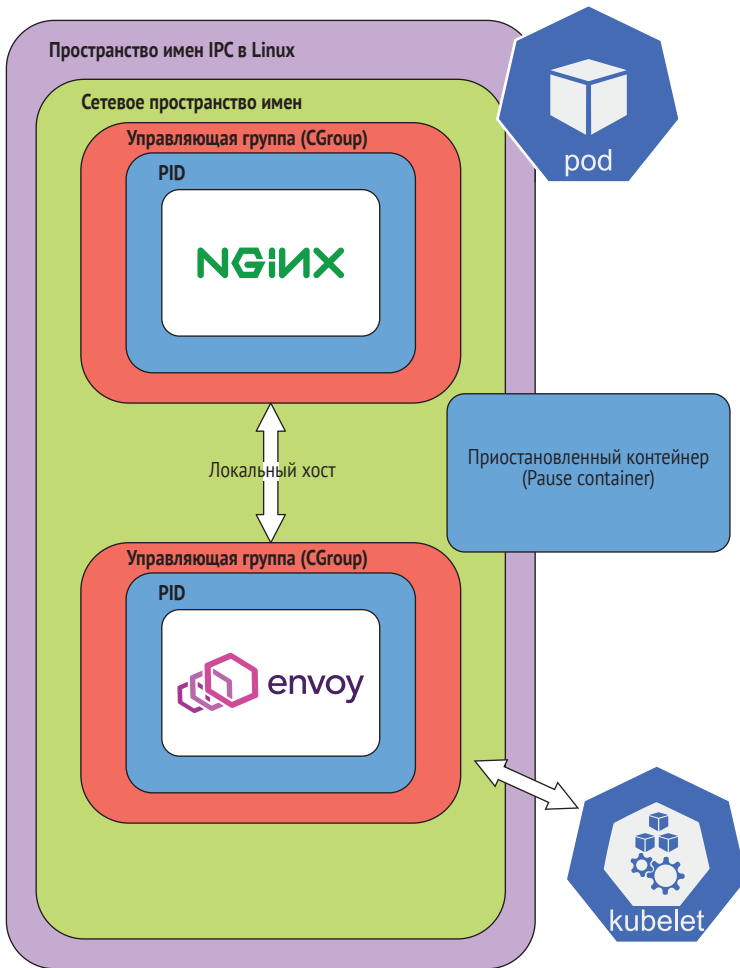


Рис. 2.1 Pod

Многие другие объекты Kubernetes API либо используют Pod напрямую, либо являются объектами API, поддерживающими Pod. Например, объект разворачивания Deployment использует Pod, а также StatefulSet и DaemonSet. Некоторые контроллеры Kubernetes высокого уровня запускают модули Pod и управляют ими. Контроллеры – это программные компоненты, работающие в плоскости управления. Примерами встроенных контроллеров могут служить: диспетчер контроллеров, диспетчер облачных вычислений и планировщик. Но давайте немного отвлечемся и создадим веб-приложение, а затем вернемся к Kubernetes, модулям Pod и плоскости управления.

ПРИМЕЧАНИЕ Возможно, вы заметили, что под плоскостью управления мы подразумеваем группу узлов, на которых работают контроллеры, диспетчер контроллеров и планировщик.

Их также называют *мастерами* или *ведущими узлами*, но в этой книге, говоря об этих компонентах, мы будем использовать термин *плоскость управления*.

2.1 Пример веб-приложения

Давайте рассмотрим пример веб-приложения, чтобы понять, зачем нужны Pod и как Kubernetes поддерживает их и контейнерные приложения. Чтобы лучше понять, зачем нужны Pod, мы на протяжении большей части этой главы используем следующий пример.

Компания по производству энергетических напитков Zeus Zap имеет свой веб-сайт, на котором потребители могут приобретать различные линейки газированных напитков. Веб-сайт имеет три уровня: пользовательский интерфейс (UI), уровень бизнес-логики (различные микросервисы) и серверную базу данных. Имеются также протоколы обмена сообщениями и очереди. Такие компании, как Zeus Zap, обычно имеют несколько веб-интерфейсов, в том числе для клиентов и для администраторов, набор различных микросервисов, образующих уровень бизнес-логики, и одну или несколько серверных баз данных. Вот структура одного фрагмента веб-приложения Zeus Zap (рис. 2.2):

- пользовательский интерфейс со сценариями на JavaScript, обслуживаемый веб-сервером NGINX;
- два веб-контроллера, реализующих микросервисы на Python с поддержкой Django;
- база данных CockroachDB, обслуживающая запросы на порту 6379, с хранилищем.

Теперь представим, что эти приложения запускаются в четырех разных контейнерах. Запуск может выполняться следующими командами `docker run`:

```
$ docker run -t -i ui -p 80:80
$ docker run -t -i microservice-a -p 8080:8080
$ docker run -t -i microservice-b -b 8081:8081
$ docker run -t -i cockroach:cockroach -p 6379:6379
```

Спустя короткое время после начала эксплуатации сервисов в компании обнаружили, что:

- они не могут запустить несколько копий контейнера поддержки пользовательского интерфейса, не предусмотрев распределение нагрузки перед портом 80, потому что на компьютере, где работает образ, есть только один порт 80;
- они не могут перенести контейнер с базой данных CockroachDB на другой сервер, не изменив IP-адрес в веб-приложении (или

не добавив DNS-сервер, динамически обновляющийся при перемещении контейнера с CockroachDB);

- они должны запускать каждый экземпляр CockroachDB на отдельном сервере, чтобы обеспечить высокую доступность;
- если экземпляр CockroachDB на одном сервере потерпит аварию, им нужна возможность переместить данные на другой узел и освободить неиспользуемое пространство в хранилище.

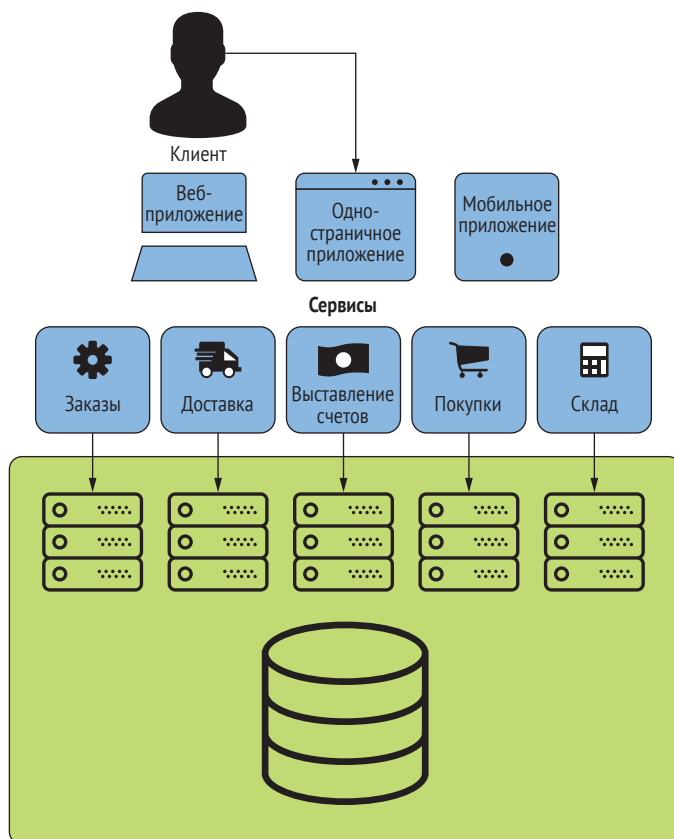


Рис. 2.2 Архитектура веб-приложения Zeus Zap

В Zeus Zap также понимают, что существуют определенные требования к платформе оркестрации контейнеров. В том числе:

- наличие общей сети, связывающей сотни процессов, привязанных к одному и тому же порту;
- возможность миграции и отделения томов хранилища от двоичных файлов без загрязнения локальных дисков;
- оптимизация использования доступных вычислительных ресурсов и памяти для снижения затрат.

ПРИМЕЧАНИЕ Запуск большого количества процессов на сервере часто приводит к явлению, получившему название *шумные соседи*: скопление большого количества приложений приводит к чрезмерной конкуренции за дефицитные ресурсы (процессор, память). Система должна смягчать эту проблему.

Масштабируемые контейнерные приложения (большие и малые) должны также поддерживать возможности планирования и управления балансировкой нагрузки. Поэтому система должна также осуществлять:

- *планирование с учетом хранилища* – для планирования процессов с одновременным обеспечением доступности его данных;
- *балансировку сетевой нагрузки с учетом сервисов* – для переключения трафика на разные IP-адреса при перемещении контейнеров с одной машины на другую.

Откровения, которыми мы только что поделились, нашли отклик у создателей инструментов распределенного планирования и оркестрации, включая Mesos и Borg, еще в 2000-х годах. Borg – это внутренняя система оркестрации контейнеров в Google, а Mesos – приложение с открытым исходным кодом. Оба инструмента обеспечивают управление кластером и предшествовали появлению Kubernetes.

2.1.1 Инфраструктура нашего веб-приложения

Без программного обеспечения оркестрации контейнеров, такого как Kubernetes, организациям требуется добавить множество компонентов в их инфраструктуру. Чтобы запустить приложение, нужны различные виртуальные машины (ВМ) в облаке или физические компьютеры, действующие как серверы, и, как упоминалось выше, сервисы должны иметь постоянные идентификаторы, чтобы их легко было отыскать.

Нагрузка на разные серверы может быть разной. Например, для баз данных могут понадобиться серверы с большим объемом памяти, а для микросервисов – серверы с меньшим объемом памяти, но с большим количеством процессоров. Также может потребоваться хранилище с малой задержкой для обслуживания базы данных, такой как MySQL или Postgres, и более медленное хранилище для резервного копирования и других приложений, которые обычно загружают данные в память и потом не обращаются к диску. Кроме того, серверам непрерывной интеграции, таким как Jenkins или CircleCI, требуется полный доступ ко всем вашим серверам, но системе мониторинга достаточно иметь доступ только для чтения лишь к некоторым вашим приложениям. Теперь добавьте сюда авторизацию и аутентификацию. В итоге вам понадобятся:

- виртуальная машина или физический сервер в качестве платформы для развертывания;

- балансировка нагрузки;
- служба обнаружения приложений;
- хранилище;
- система безопасности.

Для поддержки системы ваш персонал DevOps должен будет сопровождать следующие подсистемы (в дополнение ко многим другим):

- централизованную подсистему журналирования;
- подсистему мониторинга и оповещения;
- систему непрерывной интеграции / непрерывной доставки (CI/CD);
- подсистему резервного копирования;
- подсистему управления конфиденциальной информацией.

В отличие от большинства самодельных платформ доставки приложений, Kubernetes включает встроенные инструменты ротации журналов, мониторинга и управления. Далее перечисляются основные бизнес-задачи: эксплуатационные требования.

2.1.2 Эксплуатационные требования

У компании по производству энергетических напитков Zeus Zap нет типичных сезонных колебаний спроса, как у большинства интернет-магазинов, но они спонсируют различные мероприятия, связанные с киберспортом, привлекающие большой трафик. Это связано с тем, что отдел маркетинга и различные стримеры онлайн-игр проводят конкурсы, которые рекламируются во время этих мероприятий. Такие колебания пользовательского трафика с большими всплесками особенно сложны для управления. Масштабирование онлайн-приложений – сложная задача, и команда DevOps должна запланировать взрывной рост числа посетителей! Кроме того, из-за онлайн-кампаний в социальных сетях, разворачивающихся вокруг киберспортивных мероприятий, бизнес волнуют потенциальные перебои в работе сайта. Потери, вызываемые простоями, огромны.

Согласно исследованию Gartner, проведенному в 2018 году (<http://mng.bz/PWNn>), одна минута простоя ИТ-инфраструктуры обходится в среднем в 5600 долл. США. Есть случаи, когда двухчасовой простой приложения приводил к потере 672 000 долл. США. Деньги – это одно, но есть еще и кадровые потери. Инженеры DevOps неизбежно сталкиваются с перебоями в работе; это часть жизни, но они также создают стрессовую обстановку для персонала и приводят к выгоранию. Выгорание сотрудников в США обходится промышленности примерно в 125–190 млрд долл. в год (<http://mng.bz/4j6j>).

Многим компаниям требуется некоторый уровень высокой доступности и возможность отката своих производственных систем. Эти требования вынуждают поддерживать некоторый резерв приложений и оборудования. Однако, чтобы сэкономить на затратах, эти же компании могут увеличивать или уменьшать доступность приложений в менее напряженные периоды времени. Таким образом, управление

затратами часто противоречит более широким бизнес-требованиям в отношении времени безотказной работы. Напомним основные требования к простому веб-приложению:

- масштабирование;
- высокая доступность;
- управление версиями для поддержки возможности отката;
- управление затратами.

2.2 Что такое Pod?

Если говорить в общем, то *Pod* – это модуль, содержащий один или несколько образов OCI, которые выполняются в контейнерах на одном узле кластера Kubernetes. Узел Kubernetes – это отдельная единица вычислительной инфраструктуры (сервер), на которой выполняется kubelet. Как и все остальное в Kubernetes, узел тоже является объектом API. Чтобы развернуть Pod, достаточно следующего:

```
$ cat << EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
spec:
  container:
    - name: busybox
      image: mycontainerregistry.io/foo
EOF
```

Идентификатор версии API,
соответствующий версии на сервере API

kind объявляет тип объекта API
(в данном случае Pod)

Имя образа в реестре

Команда kubectl

Это пример команд, выполняемых в командной оболочке Linux Bash. Команда `kubectl` – это двоичный выполняемый файл, реализующий интерфейс командной строки к серверу Kubernetes API.

В большинстве случаев модули Pod развертываются не напрямую, а создаются автоматически другими объектами API, такими как `Deployment`, `Job`, `StatefulSet` и `DaemonSet`, которые тоже определяем мы:

- *Deployment* – наиболее часто используемый объект API в кластере Kubernetes. Это типичный объект API, выполняющий развертывание, например, микросервиса;
- *Job* – запускает Pod как пакетный процесс;
- *StatefulSet* – приложения с особыми требованиями; обычно это приложения с состоянием, такие как базы данных;
- *DaemonSet* – используется для запуска одного Pod в роли «агента» на каждом узле кластера (обычно для системных служб, обеспечивающих поддержку сети, хранилища или журналирования).

Ниже перечислены некоторые возможности, предлагаемые объектом `StatefulSet`:

- порядковое именование модулей Pod для получения уникальных сетевых идентификаторов;
- постоянное хранилище, которое всегда подключается к одному и тому же Pod;
- упорядоченный запуск, масштабирование и обновление.

СОВЕТ Имена образов Docker поддерживают использование метки *latest*. Не используйте в промышленном окружении такие имена образов, как `mycontainerregistry.io/foo`, потому что в этом случае из реестра всегда будет извлекаться образ с тегом *latest*, т. е. самая последняя его версия. Всегда для установки образа используйте версионированное имя или, что еще лучше, контрольную сумму SHA. Имена образов непостоянны, в отличие от контрольной суммы SHA. Многие системы терпят сбой из-за непреднамеренной установки более новой версии контейнера. Друзья, не позволяйте коллегам запускать образы с тегом *latest*!

После запуска модулей Pod можно посмотреть, какие из них действуют в пространстве имен по умолчанию, выполнив простую команду `kubectl get po`. Теперь, создав работающий контейнер, осталось развернуть в нем компоненты веб-приложения Zeus Zap, что совсем несложно (рис. 2.3). Просто используйте свой любимый инструмент для работы с образами, такой как Docker или CRI-O, и объедините двоичные файлы и их зависимости в образы, которые представляют собой обычные архивы с определениями файлов. В следующей главе мы покажем, как создавать собственные образы и модули Pod.

Вместо использования системного планировщика для выполнения различных команд `docker run` при запуске сервера мы определяем четыре объекта API более высокого уровня, которые создают модули Pod и вызывают сервер Kubernetes API. Как уже упоминалось, модули Pod редко используются для установки приложений в Kubernetes. Для этого обычно применяются абстракции более высокого уровня, такие как `Deployment` и `StatefulSet`. Но мы по-прежнему возвращаемся к Pod, потому что `Deployment` и `StatefulSet` создают объекты-реплики, которые затем создают модули Pod.

2.2.1 Пространства имен в Linux

Пространства имен Kubernetes (что создаются командой `kubectl create ns`) – это не то же самое, что пространства имен Linux. Пространства имен Linux – это функция ядра Linux, позволяющая разделять процессы. На базовом уровне Pod – это набор пространств имен в определенной конфигурации. Pod имеет следующие пространства имен Linux:

- одно или несколько пространств имен *PID*;
- единое сетевое пространство имен;

- пространство имен `IPC`;
- пространство имен `sgroup` (управляющая группа);
- пространство имен `mnt` (монтирование);
- пространство имен `user` (ID пользователя).

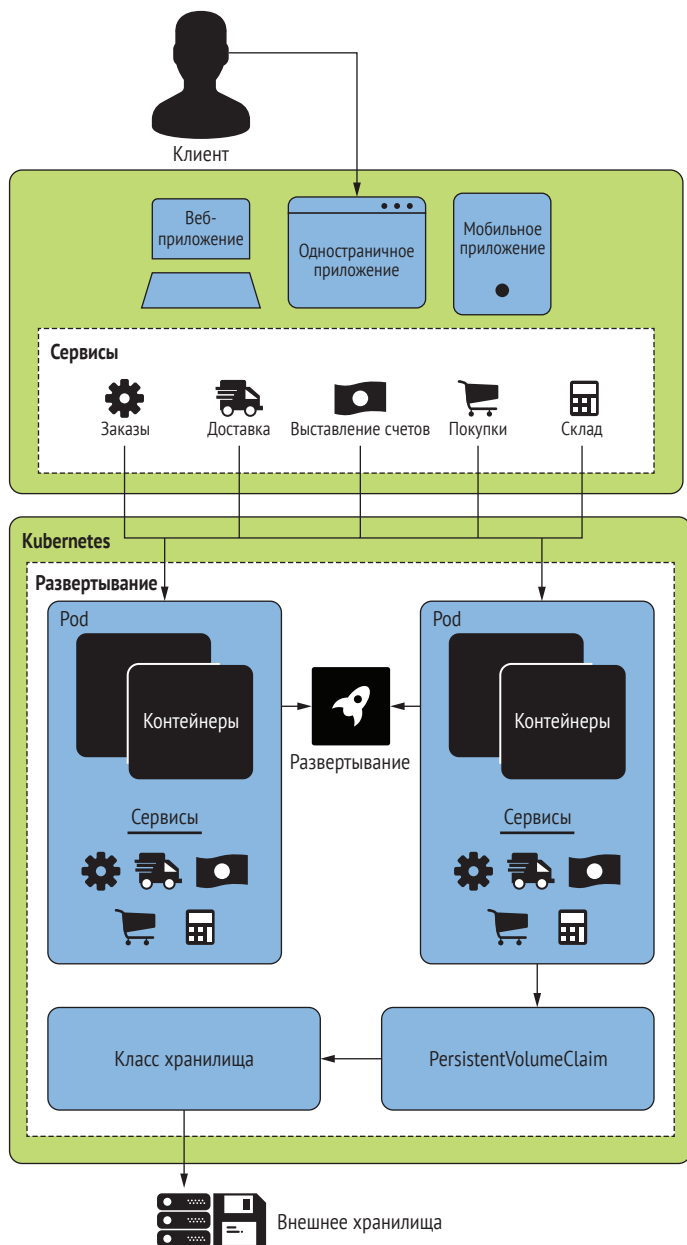


Рис. 2.3. Запуск приложения Zeus Zap в Kubernetes

Пространства имен Linux – это компоненты файловой системы ядра Linux, обеспечивающие базовую функциональность для получения образа и создания работающего контейнера. Почему это важно? Давайте вернемся к паре требований для запуска примера веб-приложения.

Как определено в требованиях, для веб-приложения важна возможность масштабирования. Pod не только дает нам и платформе Kubernetes возможность развернуть контейнер, но также позволяет масштабировать обработку объемного трафика. Для сокращения затрат и вертикального масштабирования необходимо иметь возможность регулировать настройки ресурсов для контейнера. Чтобы микросервисы Zeus Zap могли взаимодействовать с сервером CockroachDB, необходимо развернуть общую сеть и механизм поиска.

Модули Pod и их основа – пространства имен Linux – обеспечивают поддержку всех этих возможностей. В сетевом пространстве имен существует виртуальный сетевой стек, подключенный к системе программно-определяемой сети (Software-Defined Networking, SDN), охватывающей кластер Kubernetes. Потребность в масштабировании часто удовлетворяется за счет балансировки нагрузки между несколькими модулями Pod с приложением. SDN в кластере Kubernetes – это сетевая структура, поддерживающая балансировку нагрузки.

2.2.2 *Kubernetes, инфраструктура и Pod*

Серверы зависят от работы Kubernetes и Pod. Единица вычислительной мощности в Kubernetes представлена объектом узла Node. Узел может работать на множестве платформ, но фактически это просто сервер с определенными компонентами. Вот некоторые требования к узлу:

- сервер;
- установленная операционная система (ОС), Linux или Windows, с необходимыми зависимостями;
- systemd (диспетчер системных служб, в Linux);
- kubelet (агент узла);
- среда выполнения контейнеров (например, Docker);
- сетевой прокси-сервер (kube-проху), обслуживающий сервисы Kubernetes;
- провайдер сетевого интерфейса контейнеров (Container Network Interface, CNI).

Узел может работать на Raspberry Pi, виртуальной машине в облаке и множестве других платформ. На рис. 2.4 показано, какие компоненты образуют узел, работающий в Linux.

kubelet – это двоичная программа, играющая роль агента и взаимодействующая с сервером Kubernetes API посредством поддержки цикла управления. Она работает на каждом узле; без этой программы узел Kubernetes недоступен планировщику и не может считаться

частью кластера. Знание возможностей kubelet помогает диагностировать низкоуровневые проблемы, такие как отказ узла присоединиться к кластеру или ошибки развертывания модулей Pod. Программа kubelet гарантирует:

- запуск любых модулей Pod, запланированных для выполнения на данном хосте механизмом управления, который следит за распределением модулей Pod между узлами;
- регулярное уведомление сервера API об исправной работе kubelet отправкой контрольных сообщений (Kubernetes 1.17+) с помощью механизма в пространстве имен kube-node-lease кластера;
- своевременное освобождение ресурсов, выделенных для Pod, включая эфемерные хранилища или сетевые устройства.

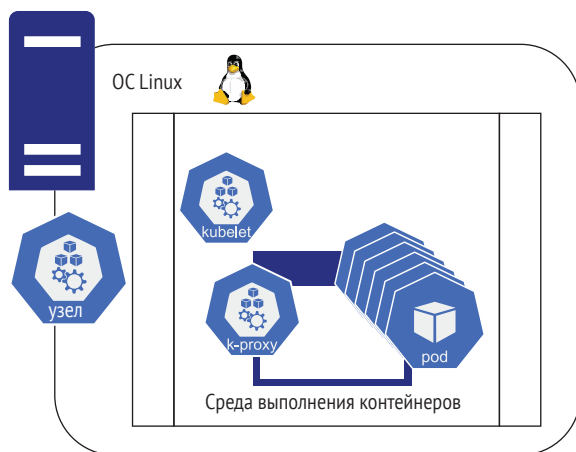


Рис. 2.4. Узел

Однако программа kubelet не может выполнять свои обязанности без провайдера CNI и среды выполнения, доступной через интерфейс среды выполнения контейнеров (Container Runtime Interface, CRI). CNI обслуживает потребности CRI, который затем запускает и останавливает контейнеры. kubelet использует CRI и CNI для согласования состояния узла с состоянием плоскости управления. Например, когда плоскость управления решает, что NGINX будет работать на втором, третьем и четвертом узлах кластера, состоящего из пяти узлов, то задача kubelet – гарантировать, что провайдер CRI извлечет соответствующий контейнер из реестра образов и запустит его с IP-адресом в диапазоне podCIDR. В главе 9 мы расскажем, как принимаются такие решения. Для CRI важно наличие механизма запуска контейнеров. Типичными примерами таких механизмов могут служить: Docker, CRI-O и LXC.

Сервис (Service) – это объект API, определяемый платформой Kubernetes. Двоичный файл сетевого прокси Kubernetes (kube-проxy) создает на каждом узле сервисы ClusterIP и NodePort. Вот некоторые типы сервисов:

- *ClusterIP* – внутренний балансировщик, распределяющий нагрузку между модулями Pod в кластере Kubernetes;
- *NodePort* – открытый порт на узле Kubernetes, распределяющий нагрузку между несколькими модулями Pod;
- *LoadBalancer* – внешний сервис, создающий балансировщик нагрузки, внешний по отношению к кластеру.

Сетевой прокси-сервер Kubernetes может не устанавливаться, потому что некоторые провайдеры сетевых услуг заменяют его своим сетевым компонентом, осуществляющим управление сервисами. Kubernetes позволяет использовать один объект Service для проксирования нескольких модулей Pod одного типа. Для этого каждый узел в кластере должен иметь информацию о каждом сервисе и каждом модуле Pod. Сетевой прокси-сервер Kubernetes управляет всеми сервисам на каждом узле, как определено для конкретного кластера. Он поддерживает сетевые протоколы TCP, UDP и STCP, а также переадресацию и балансировку нагрузки.

ПРИМЕЧАНИЕ Сетевые взаимодействия в Kubernetes поддерживаются с помощью программного решения, называемого провайдером CNI. Некоторые провайдеры CNI создают компоненты, заменяющие сетевой прокси Kubernetes собственной программной инфраструктурой. Благодаря этому они могут использовать разные сети без iptables.

2.2.3 Объект Node

Как отмечалось выше, узлы поддерживают модули Pod, а плоскость управления определяет группу узлов, на которых работают контроллеры, диспетчер контроллеров и планировщик. Получить список узлов кластера можно с помощью этой простой команды `kubectl`:

```
$ kubectl get no
```

NAME	STATUS	ROLES	AGE	VERSION
kind-control-plane	NotReady	master	25s	v1.17.0

Полная команда имеет вид: `kubectl get nodes`. Извлекает объекты Node, зарегистрированные в кластере Kubernetes

Список узлов в кластере. Обратите внимание на версию 1.17.0 – вероятно, она немного старше той, которую вы используете

Теперь давайте взглянем на объект Node, описывающий узел, где размещена плоскость управления Kubernetes:

```
$ kubectl get no kind-control-plane -o yaml
```

В следующем примере показано полное представление объекта Node. (В этом примере файл YAML разбит на несколько разделов, потому что он довольно длинный.)

```

apiVersion: v1
kind: Node
metadata:
  annotations:
    kubeadm.alpha.kubernetes.io/cri-socket:
      /run/containerd/containerd.sock ← Используемый сокет CRI.
                                         Здесь (как и в большинстве кластеров)
                                         используется сокет containerd
    node.alpha.kubernetes.io/ttl: "0"
    volumes.kubernetes.io/controller-managed-attach-detach: "true"
  creationTimestamp: "2020-09-20T14:51:57Z"
  labels: ← Стандартные метки, включая имя узла
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: kind-control-plane
    kubernetes.io/os: linux
    node-role.kubernetes.io/master: ""
  name: kind-control-plane
  resourceVersion: "1297"
  selfLink: /api/v1/nodes/kind-control-plane
  uid: 1636e5e1-584c-4823-9e6b-66ab5f390592
spec:
  podCIDR: 10.244.0.0/24 ← IP-адрес CNI, который определяет CIDR
                           (диапазон адресов) сети объектов Pod
  podCIDRs:
    - 10.244.0.0/24
# продолжение в следующем разделе

```

Теперь перейдем к разделу `status`. Он содержит информацию об узле и его содержимом.

```

status: ← Различные поля состояния, получаемые
          сервером API от агента kubelet,
          работающего на узле
  addresses:
    - address: 172.17.0.2
      type: InternalIP
    - address: kind-control-plane
      type: Hostname
  allocatable:
    cpu: "2"
    ephemeral-storage: 61255492Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 2039264Ki
    pods: "110"
  capacity:
    cpu: "2"
    ephemeral-storage: 61255492Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 2039264Ki
    pods: "110"
  conditions:
    - lastHeartbeatTime: "2020-09-20T14:57:28Z"
      lastTransitionTime: "2020-09-20T14:51:51Z"

```

```

message: kubelet has sufficient memory available
reason: KubeletHasSufficientMemory
status: "False"
type: MemoryPressure
- lastHeartbeatTime: "2020-09-20T14:57:28Z"
  lastTransitionTime: "2020-09-20T14:51:51Z"
  message: kubelet has no disk pressure
  reason: KubeletHasNoDiskPressure
  status: "False"
  type: DiskPressure
- lastHeartbeatTime: "2020-09-20T14:57:28Z"
  lastTransitionTime: "2020-09-20T14:51:51Z"
  message: kubelet has sufficient PID available
  reason: KubeletHasSufficientPID
  status: "False"
  type: PIDPressure
- lastHeartbeatTime: "2020-09-20T14:57:28Z"
  lastTransitionTime: "2020-09-20T14:52:27Z"
  message: kubelet is posting ready status
  reason: KubeletReady
  status: "True"
  type: Ready
daemonEndpoints:
  kubeletEndpoint:
    Port: 10250

```

Теперь посмотрим, какие образы запущены на узле:

```

images:
- names:
  - k8s.gcr.io/etcd:3.4.3-0
  sizeBytes: 289997247
- names:
  - k8s.gcr.io/kube-apiserver:v1.17.0
  sizeBytes: 144347953
- names:
  - k8s.gcr.io/kube-proxy:v1.17.0
  sizeBytes: 132100734
- names:
  - k8s.gcr.io/kube-controller-manager:v1.17.0
  sizeBytes: 131180355
- names:
  - docker.io/kindest/kindnetd:0.5.4
  sizeBytes: 113207016
- names:
  - k8s.gcr.io/kube-scheduler:v1.17.0
  sizeBytes: 111937841
- names:
  - k8s.gcr.io/debian-base:v2.0.0
  sizeBytes: 53884301
- names:

```

Различные образы, работающие на узле

Сервер etcd, играющий роль базы данных для Kubernetes

Сервер API и другие контроллеры (такие как kube-controller-manager)

Поставщик CNI. Нам нужна программно-определяемая сеть, и этот контейнер обеспечивает необходимую функциональность

```

- k8s.gcr.io/coredns:1.6.5
  sizeBytes: 41705951
- names:
  - docker.io/rancher/local-path-provisioner:v0.0.11
    sizeBytes: 36513375
- names:
  - k8s.gcr.io/pause:3.1
    sizeBytes: 746479

```

Наконец, добавим блок `nodeInfo`. Он включает управление версиями для системы Kubernetes:

nodeInfo: ← Сообщает информацию об узле, включая версии ОС, kube-proxy и kubelet

```

architecture: amd64
bootID: 0c700452-c292-4190-942c-55509dc43a55
containerRuntimeVersion: containerd://1.3.2
kernelVersion: 4.19.76-linuxkit
kubeProxyVersion: v1.17.0
kubeletVersion: v1.17.0
machineID: 27e279849eb94684ae8c173287862c26
operatingSystem: linux
osImage: Ubuntu 19.10
systemUUID: 9f5682fb-6de0-4f24-b513-2cd7e6204b0a

```

Для запуска узла нам нужен механизм запуска контейнеров, сетевой прокси (`kube-proxy`) и `kubelet`. Но вернемся к этому чуть позже.

Контроллеры и циклы управления

Слово *управление* имеет множество значений в контексте Kubernetes; все они связаны между собой, и это вносит некоторую путаницу. Существуют циклы управления, контроллеры (управляющие механизмы) и плоскость управления. Окружение Kubernetes состоит из нескольких выполняемых двоичных файлов, называемых *контроллерами*. Они известны как `kubelet`, сетевой прокси Kubernetes, планировщик и др. Контроллеры написаны с использованием шаблона проектирования с названием *циклы управления*. Плоскость управления содержит определенные контроллеры. Эти узлы и контроллеры являются основой, мозгом Kubernetes. В этой главе мы еще поговорим об этом.

Узлы, составляющие плоскость управления, иногда называют *мастерами* или *ведущими узлами*, но мы в этой книге будем использовать термин *плоскость управления*. По своей сути Kubernetes – это машина согласования состояний с различными циклами управления, подобно кондиционеру. Однако вместо регулирования температуры Kubernetes контролирует и регулирует многие аспекты, связанные с управлением распределенными приложениями:

- привязку хранилища к процессам;
- запуск контейнеров и масштабирование их количества;

- остановку и перенос контейнеров на другие узлы при потере работоспособности;
- создание IP-маршрутов к портам;
- динамическое обновление конечных точек с балансировкой нагрузки.

Вернемся к требованиям, перечисленным выше: модули Pod реализуют возможность развертывания образов. Образы развертываются на узле, а их жизненным циклом управляет kubelet. Объекты сервисов управляются сетевым прокси Kubernetes. Система DNS, такая как CoreDNS, обеспечивает поиск приложений и позволяет микросервисам, действующим в пределах одного модуля Pod, находить и взаимодействовать с другими модулями, например с CockroachDB.

Сетевой прокси Kubernetes обеспечивает возможность балансировки нагрузки внутри кластера, а также аварийное переключение, обновление, высокую доступность и масштабирование. Для удовлетворения потребностей в долговременном хранилище комбинация из пространства имен `mnt` в Linux, агента kubelet и объекта узла Node позволяет подключить диск к Pod. Когда kubelet создает Pod, к нему автоматически подключается назначенное хранилище.

Кажется, нам не хватает еще кое-чего. Что делать, если узел выходит из строя? Как вообще загружаются модули Pod на узлы? Все это обеспечивает плоскость управления.

2.2.4 *Наше веб-приложение и плоскость управления*

Теперь, узнав, как создаются модули Pod и узлы, давайте посмотрим, как обеспечивается удовлетворение сложных требований, таких как высокая доступность. Высокая доступность – это не просто отказоустойчивость, а соответствие требованиям соглашения об уровне обслуживания (Service-Level Agreement, SLA). Доступность системы часто измеряется количеством девяток времени безотказной работы. Это количество позволяет оценить, какую долю от общего времени может простаивать приложение или набор приложений. Четыре девятки соответствуют 52 мин и 36 с простоя в год; пять девяток (время безотказной работы 99,999 %) соответствуют 5 мин и 15 с возможного простоя в год, или 26,25 с в месяц. Соглашение с пятью девятками требует, чтобы приложение, размещенное в Kubernetes, оставалось недоступным не более полуминуты в месяц. Это невероятно сложно! Все остальные требования тоже весьма непростые. К их числу относятся, например:

- масштабирование;
- экономия средств;
- управление версиями контейнеров;
- безопасность пользователей и приложений.

ПРИМЕЧАНИЕ Да, Kubernetes предоставляет все эти возможности, но приложения тоже должны учитывать особенности работы Kubernetes. Подробнее о проектировании таких приложений мы поговорим в последней главе этой книги.

Итак, первый шаг – подготовка Pod. Однако, помимо этого, у нас есть система, обеспечивающая не только отказоустойчивость и масштабируемость, но также возможность экономить деньги и, соответственно, контролировать расходы. На рис. 2.5 показан состав типичной плоскости управления.

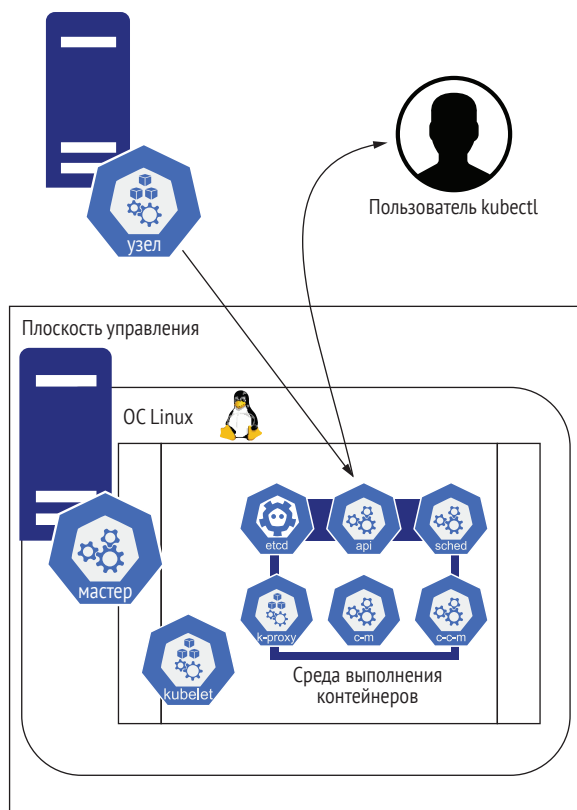


Рис. 2.5 Плоскость управления

2.3 Создание веб-приложения с помощью kubectl

Чтобы понять, как плоскость управления обеспечивает масштабирование и отказоустойчивость, рассмотрим простую команду: `kubectl apply -f deployment.yaml`. Файл `deployment.yaml`, описывающий развёртывание, показан ниже:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

После запуска команды `kubectl apply` программа `kubectl` связывается с первым компонентом в плоскости управления – сервером API.

2.3.1 Сервер Kubernetes API: kube-apiserver

Сервер Kubernetes API (`kube-apiserver`) – это HTTP REST-сервер, экспортирующий различные объекты API для кластера Kubernetes, такие как Pod, Node или HorizontalPodAutoscaler. Сервер API предоставляет веб-интерфейс для выполнения операций CRUD с состоянием кластера. В промышленных окружениях плоскости управления Kubernetes обычно обеспечивают высокую доступность сервера Kubernetes API, запуская его на каждом узле, входящем в плоскость управления, или в облачной службе, использующей какой-то другой механизм обеспечения отказоустойчивости и высокой доступности. На практике это означает, что доступ к серверу API осуществляется через конечную точку HAProxy или облачный балансировщик нагрузки.

Компоненты плоскости управления тоже взаимодействуют с сервером API. В момент запуска узла kubelet гарантирует его регистрацию в кластере, взаимодействуя с сервером API. Все компоненты плоскости управления имеют функцию мониторинга для отслеживания изменений в объектах сервера API.

Сервер API – единственный компонент в плоскости управления, взаимодействующий с etcd, базой данных Kubernetes. В прошлом некоторые другие компоненты, такие как провайдеры CNI, тоже взаимодействовали с etcd, но в настоящее время все манипуляции с данными в хранилище выполняются через сервер API. По сути, сервер API предоставляет интерфейс для всех операций, изменяющих состояние кластера Kubernetes. По этой причине безопасность сервера API и его конечных точек HTTPS имеет решающее значение.

При работе плоскости управления в режиме высокой доступности все серверы Kubernetes API активны и получают трафик. Сервер API не имеет состояния и может работать на нескольких узлах одновременно. Перед серверами API в плоскости управления, состоящей из нескольких узлов, размещается балансировщик нагрузки HTTPS. Было бы крайне нежелательно, чтобы кто-то посторонний имел возможность связаться с сервером API, поэтому в состав сервера API входят *контроллеры доступа*, обеспечивающие аутентификацию и авторизацию клиентов.

Нередко в сервер API через веб-обработчики интегрируются внешние системы аутентификации и авторизации. *Веб-обработчик* (webhook) – это HTTP PUSH API, дающий возможность выполнять обратные вызовы. Вызов, отправленный командой `kubectl`, проходит аутентификацию, а затем сервер API сохраняет новый объект развертывания в etcd. Следующий шаг – уведомление планировщика о необходимости запустить модуль Pod на узле. Новым модулям Pod нужен новый дом, поэтому планировщик связывает их с определенными объектами узлов Node.

2.3.2 Планировщик Kubernetes: kube-scheduler

Распределенное планирование – непростая задача. Планировщик Kubernetes (kube-scheduler) предлагает чистую и простую реализацию планирования, идеально подходящую для такой сложной системы, как Kubernetes. При планировании модулей Pod он учитывает несколько факторов, таких как аппаратная конфигурация узла, доступные вычислительные ресурсы и объем памяти, ограничения политик планирования и др.

Планировщик также следует правилам соответствия/несоответствия (affinity/anti-affinity), определяющим порядок планирования и размещения модулей Pod. По сути, правила соответствия определяют силу притяжения модулей Pod к узлам, отвечающим требованиям, тогда как правила несоответствия определяют силу отталкивания. Ограничения (Taint), дополняющие правила, позволяют узлам отклонять определенные типы модулей, а это означает, что планировщик может определить, какие модули и на каких узлах не должны находиться. В примере с Zeus Zap (раздел 2.1.2) определено, что модуль Pod может иметь три реплики (копии). Запуск дополнительных реплик обеспечивает как отказоустойчивость, так и возможность масштабирования. Планировщик выбирает узлы для размещения реплик, а затем планирует развертывание модулей Pod на них.

Жизненным циклом модуля Pod управляет агент kubelet. Он действует как мини-планировщик для узла. Как только планировщик Kubernetes обновит nodeName в определении Pod, kubelet развернет этот модуль на своем узле. Плоскость управления полностью отделена от узлов, на которых отсутствуют ее компоненты. Даже при отключении

плоскости управления приложение Zeus Zap не потеряет свою информацию. В случае сбоя плоскости управления ничего нового не удастся развернуть, но сам веб-сайт продолжит работать.

А что случится, если после выхода из строя плоскости управления приложению потребуется обратиться к подключенному к нему хранилищу? В этом приложения, скорее всего, продолжат работать как обычно, пока не возникнут проблемы на узлах, где они запущены. Но даже в этом случае после восстановления работоспособности плоскости управления можно ожидать безопасной миграции как данных, так и приложения в новый дом благодаря соответствующим возможностям плоскости управления Kubernetes.

2.3.3 Контроллеры инфраструктуры

Одним из требований Zeus Zap к инфраструктуре является наличие CockroachDB – распределенной базы данных, совместимой с Postgres и работающей в собственном облачном окружении. Приложения с состоянием, такие как базы данных, часто имеют особые требования, вследствие чего для управления ими необходим контроллер или некоторая реализация шаблона проектирования Operator (Оператор). Поскольку шаблон Operator считается стандартным механизмом развертывания сложных приложений в Kubernetes, мы советуем отказаться от применения простого YAML для установки и использовать реализацию оператора. Следующий пример демонстрирует установку оператора для CockroachDB:

```
$ kubectl apply -f https://raw.githubusercontent.com/cockroachdb/cockroach-operator/master/install/crds.yaml
$ kubectl apply -f https://raw.githubusercontent.com/cockroachdb/cockroach-operator/master/install/operator.yaml
```

Установка собственного определения ресурса, используемого оператором

Установка оператора в пространство имен по умолчанию

Собственные определения ресурсов

Собственные определения ресурсов (Custom Resource Definition, CRD) – это объекты API, определяющие новые объекты API. Пользователь создает определение CRD, обычно в формате YAML, а затем применяет к существующему кластеру Kubernetes и фактически позволяет создать другой объект API. CRD обычно используются для определения новых пользовательских объектов, представляющих ресурсы.

После установки оператора CockroachDB можно загрузить файл `example.yaml` командой `curl`, как показано ниже:

```
$ curl -LO https://raw.githubusercontent.com/cockroachdb/cockroach-operator/master/examples/example.yaml
```

Вот фрагмент этого файла:

```
apiVersion: crdb.cockroachlabs.com/v1alpha1
kind: CrdbCluster
metadata:
  name: cockroachdb
spec:
  dataStore:
    pvc:
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: "60Gi"
          volumeMode: Filesystem
resources:
  requests:
    cpu: "2"
    memory: "8Gi"
  limits:
    cpu: "2"
    memory: "8Gi"
tlsEnabled: true
image:
  name: cockroachdb/cockroach:v21.1.5
nodes: 3
additionalLabels:
  crdb: is-cool
```

← Образ, используемый для запуска базы данных

Этот собственный ресурс использует шаблон Operator (Оператор) для создания следующих ресурсов и управления ими (обратите внимание, что определения этих элементов легко могут занимать собой сотни строк YAML), это:

- ключи безопасности транспортного уровня (Transport Layer Security, TLS), хранящиеся вместе с другой конфиденциальной информацией в базе данных;
- StatefulSet, в котором находится CockroachDB, включая хранилище PersistentVolume и PersistentVolumeClaim;
- сервисы;
- бюджет отключения модуля Pod (PodDisruptionBudget, PDB).

Давайте возьмем за основу только что приведенный пример и углубимся в инфраструктуру поддержки контроллеров, которая называется диспетчером контроллеров Kubernetes (Kubernetes Controller Manager, КСМ) или компонентом kube-controller-manager и облачным диспетчером контроллеров (Cloud Controller Manager, ССМ). Имея развернутый объект StatefulSet, построенный на модулях Pod, нам теперь нужно хранилище для StatefulSet.

Объекты API PersistentVolume (PV) и PersistentVolumeClaim (PVC) определяют хранилища и воплощаются в реальность диспетчерами

KCM и CCM. Одна из ключевых особенностей Kubernetes – возможность работать на множестве платформ: в облаке, на «голом железе» или на ноутбуке. Однако хранилища и другие компоненты различаются на разных платформах. KCM – это набор циклов управления, запускающих различные компоненты, называемые контроллерами, на узле, находящемся в плоскости управления. Это один двоичный файл, но он управляет несколькими циклами и, соответственно, контроллерами.

Рождение облачного диспетчера контроллеров (CCM)

В команду разработчиков Kubernetes входят инженеры со всего мира, объединенные под эгидой фонда CNCF (Cloud Native Computing Foundation), куда входят сотни корпоративных членов. Удовлетворение потребностей такого широкого круга предприятий невозможно без выделения функциональности конкретных производителей в четко определенные подключаемые компоненты.

Диспетчер KCM исторически имел тесно связанную и сложную в сопровождении реализацию в основном из-за сложности технологий различных провайдеров. Например, для предоставления новых IP-адресов или томов хранилищ используются совершенно разные пути в коде в зависимости от того, используете ли вы Google Kubernetes Engine (GKE) или Amazon Web Services (AWS). Учитывая наличие нескольких специализированных предложений для Kubernetes (vSphere, Openstack и т. д.), с первых дней существования Kubernetes сохраняется расползание кода, специфичного для облачных провайдеров.

Реализация KCM находится в репозитории github.com/kubernetes/kubernetes, который часто называют *kk*. Нет ничего плохого в наличии огромного монолитного репозитория. У Google есть только один репозиторий для всей компании, но монолитный репозиторий Kubernetes перерос GitHub и вариант использования одной компании. В какой-то момент инженеры, работающие над Kubernetes, осознали, что им требуется выделить функциональность, зависящую от конкретного провайдера, как упоминалось выше. Одним из этапов этого крестового похода стало создание CCM, который обобщенным образом использовал функциональные возможности любого провайдера, реализующего интерфейс облачного провайдера (<http://mng.bz/QWRv>). Более того, тот же шаблон теперь используется в реализации планировщика Kubernetes и его плагинов.

Целью CCM было ускорение разработки и создания облачных провайдеров. CCM определяет интерфейс, позволяющий разрабатывать облачные провайдеры, такие как DigitalOcean, вне основного репозитория Kubernetes на GitHub. Такая реструктуризация репозитория позволила владельцам облачных провайдеров управлять своим кодом и двигаться вперед с большей скоростью. Каждый облачный провайдер теперь хранится в своем репозитории за пределами хоста Kubernetes.

После выхода Kubernetes v1.6 началась работа по переносу функциональности из KCM в CCM. CCM обещает сделать Kubernetes полностью независимым от конкретной реализации облака. Его архитектура следует в русле развития архитектуры Kubernetes, полностью отделенной от реализации любых подключаемых технологий.

При развертывании в облачном окружении платформа Kubernetes напрямую взаимодействует с API общедоступного или частного облака, и большинство соответствующих вызовов API выполняет CCM. Цель этого компонента – запускать контроллеры, специфичные для облака, и выполнять вызовы к API облака. Вот список этих контроллеров:

- *контроллер узлов* – выполняет тот же код, что и KCM;
- *контроллер маршрутизации* – настраивает маршруты в используемой облачной инфраструктуре;
- *контроллер сервисов* – создает, обновляет и удаляет балансировщики нагрузки облачного провайдера;
- *контроллер томов* – создает, подключает и монтирует тома, а также взаимодействует с облачным провайдером, осуществляя управление томами.

Эти контроллеры постепенно переводятся на работу с интерфейсом облачного провайдера, и эта тенденция широко распространена в Kubernetes. Также разрабатываются другие интерфейсы для поддержки более модульного и независимого от провайдеров услуг будущего Kubernetes:

- *сетевой интерфейс контейнеров* (Container Network Interface, CNI) – предоставляет IP-адреса модулям Pod;
- *интерфейс среды выполнения контейнеров* (Container Runtime Interface, CRI) – определяет и подключает различные механизмы выполнения контейнеров;
- *интерфейс хранилища для контейнеров* (Container Storage Interface, CSI) – модульный способ поддержки новых типов хранилищ без необходимости изменять код Kubernetes.

Теперь вернемся к нашему примеру. Для подключения к модулям Pod с базой данных CockroachDB необходимо хранилище. Когда Pod запланирован для запуска на узле kubelet, контроллер KCM (или CCM) выявляет необходимость в новом хранилище и создает его, учитывая особенности платформы, на которой он работает. Затем хранилище монтируется на узле. Когда kubelet создает модуль Pod, он определяет нужное хранилище и подключает его к контейнеру через пространство имен `mnt` Linux. После этого приложение может пользоваться хранилищем.

Наше веб-приложение Zeus Zap также нуждается в балансировщике нагрузки, чтобы иметь возможность масштабировать общедоступный веб-сайт. При создании сервиса LoadBalancer вместо ClusterIP облачный провайдер Kubernetes «наблюдает» за запросом балансировщика и выполняет его (например, вызывая облачный API для

предоставления внешнего IP-адреса и связывая его с внутренней конечной точкой сервиса Kubernetes). Однако с точки зрения конечного пользователя запрос выглядит довольно просто:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  type: LoadBalancer
```

Контроллер КСМ, выполняя цикл наблюдения, обнаруживает потребность в новом балансировщике нагрузки и выполняет вызовы API, необходимые для создания балансировщика в облаке, или вызывает аппаратный балансировщик нагрузки, внешний по отношению к кластеру Kubernetes. В процессе выполнения этих вызовов API базовая инфраструктура выясняет, какие узлы являются частью кластера, а затем направляет им трафик. Как только вызов достигает узла, программно-определяемая сеть, предоставляемая провайдером CNI, направляет трафик в нужный модуль Pod.

2.4 Масштабирование, высокодоступные приложения и плоскость управления

Масштабирование приложений вверх и вниз – это основной механизм поддержки высокой доступности приложений в облаке и особенно в Kubernetes. Суть масштабирования заключается в создании дополнительных модулей Pod или повторном развертывании, когда обнаруживается их нехватка или когда какой-то модуль Pod или узел терпит аварию. Выполняя масштабирование, `kubectl` может увеличивать и уменьшать количество модулей Pod в кластере. Он работает непосредственно с наборами реплик, состояний и другими объектами API, которые используют модули Pod, в зависимости от входных данных, передаваемых в команде. Например:

```
$ kubectl scale --replicas 300 deployment zeus-front-end-ui
```

Эта команда не влияет на объекты DaemonSet. Несмотря на то что объекты DaemonSet создают модули Pod, они не масштабируются, потому что по определению они запускают один модуль Pod на каждом узле кластера: их масштаб определяется количеством узлов в кластере. Применительно к Zeus эта команда увеличивает или уменьшает

количество модулей Pod, поддерживающих развертывание пользовательского интерфейса Zeus, следуя тому же шаблону, что и планировщик, КСМ или kubelet в предыдущем примере. На рис. 2.6 показана типичная последовательность выполнения команды `kubectl scale`.

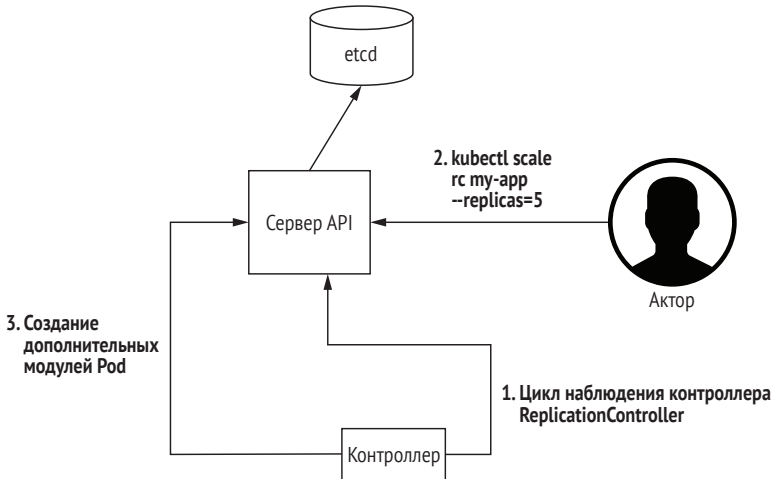


Рис. 2.6 Последовательность операций, выполняемых командой `kubectl scale`

Что происходит, когда что-то идет не так? Сбой можно разбить на три основные категории: сбой модуля Pod, сбой узла и сбой обновления программного обеспечения.

Первой разберем ситуацию со сбоем модуля Pod. За жизненный цикл модулей Pod отвечает агент kubelet, который выполняет запуск, остановку и перезапуск модулей. Выход модуля Pod из строя определяется по отсутствию контрольных сообщений от него или по аварийному завершению процесса. В этом случае kubelet пытается перезапустить модуль. Более подробно работу kubelet мы рассмотрим в главе 9.

Вторая ситуация – сбой узла. Один из циклов управления в kubelet постоянно сообщает серверу API об исправности узла (посылая контрольные сообщения). В Kubernetes 1.17+ поддержку этих контрольных сообщений можно увидеть, заглянув в пространство имен `kube-node-lease` кластера. Если узел присылает контрольные сообщения недостаточно часто, то контроллер КСМ меняет его статус на «вне сети» и планировщик перестает планировать модули Pod для запуска на этом узле. Модули, действовавшие на узле, планируются для удаления, а затем переносятся на другие узлы.

Этот процесс можно наблюдать, запустив вручную команду `kubectl cordon node-name`, а затем команду `kubectl drain node-name`. Узел может находиться в различных состояниях: вне сети, частые повторные запуски `docker`, kubelet готов к работе и др. Любое из этих состояний, сообщаящих о неработоспособности узла, останавливает планирование запуска новых модулей Pod на этом узле.

Наконец, из-за сбоев в обновлении программного обеспечения многие веб-сайты и другие сервисы планируют простои, но крупные игроки в интернете, такие как Facebook и Google, никогда этого не делают. Обе эти компании используют специальное программное обеспечение, появившееся до Kubernetes. Kubernetes создан для развертывания обновлений как самой платформы Kubernetes, так и новых модулей Pod без простоев. Однако важно понимать, что программное обеспечение, работающее на платформе Kubernetes, должно быть надежным и поддерживать возможность повторного запуска. Если они недостаточно устойчивы к сбоям, может произойти потеря данных.

Приложения, размещенные на платформе Kubernetes, должны поддерживать корректное завершение работы и последующий запуск. Например, если приложение выполняет какую-то транзакцию, оно должно поддерживать либо продолжение выполнения транзакции в другой реплике, либо перезапуск транзакции после повторного запуска приложения. Обновление развертываний осуществляется простым изменением версии образа в определении YAML и обычно выполняется одним из трех способов:

- `kubectl edit` – принимает объект Kubernetes API на входе и открывает локальный терминал для редактирования объекта API на месте;
- `kubectl apply` – принимает файл на входе и отыскивает объект API, соответствующий этому файлу, автоматически заменяя его;
- `kubectl patch` – применяет небольшой файл с исправлениями, определяющий различия для объекта.

В главе 15 мы рассмотрим полноценные инструменты исправления YAML и управления жизненным циклом приложений. Там мы рассмотрим эту широкую тему более комплексно.

Обновление кластера Kubernetes – нетривиальная задача, но Kubernetes поддерживает несколько разных способов обновления. Мы обсудим их в последней главе книги, так как эта глава посвящена плоскости управления, а не оперативным задачам.

2.4.1 Автоматическое масштабирование

Масштабирование развертываний вручную – это прекрасно, но как быть, если кластер вдруг начнет получать по 10 000 новых веб-запросов в минуту? В таком случае вам поможет автоматическое масштабирование. Есть три формы автоматического масштабирования:

- создание большого количества модулей Pod (горизонтальное автоматическое масштабирование с помощью `HorizontalPodAutoscaler`);
- предоставление модулям Pod большого количества ресурсов (вертикальное автоматическое масштабирование с помощью `VerticalPodAutoscaler`);
- создание дополнительных узлов (с помощью `ClusterAutoscaler`).

ПРИМЕЧАНИЕ Механизмы автоматического масштабирования могут быть недоступны на некоторых платформах без операционной системы.

2.4.2 Управление затратами

При автоматическом масштабировании в кластер добавляются дополнительные узлы, а это означает увеличение стоимости облачных услуг. Большое количество узлов позволяет приложениям создать больше реплик и справляться с большей нагрузкой, но тогда ваше руководство, получив счет за услуги, захочет найти решение, позволяющее сэкономить деньги. Одно из таких решений – увеличение плотности модулей Pod на узлах.

Модули Pod – это самые маленькие единицы любого приложения в Kubernetes. Каждый модуль – это группа из одного или нескольких контейнеров, использующих одну и ту же сеть. Узлы, на которых размещаются модули Pod, – это либо виртуальные машины, либо физические серверы. Чем больше модулей Pod размещается на одном узле, тем ниже затраты на дополнительные серверы. Kubernetes поддерживает *возможность плотного размещения модулей Pod*, что позволяет запускать узлы с избыточными ресурсами и высокой плотностью модулей. Плотность Pod контролируется с помощью следующих шагов.

- 1 *Масштабируйте и профилируйте свои приложения.* Приложения должны быть протестированы и тщательно проверены как на предмет потребления памяти, так и центрального процессора. После профилирования можно правильно определить потребности приложения в ресурсах.
- 2 *Выберите размер узла.* Это позволит упаковать несколько приложений на одном узле. Запуск виртуальных машин разных размеров или серверов без системного программного обеспечения с разной емкостью позволяет сэкономить деньги и развернуть на них больше модулей Pod. При этом вы должны обеспечить достаточно большое количество узлов для поддержания высокой доступности в соответствии с требованиями SLA.
- 3 *Сгруппируйте определенные приложения вместе на определенных узлах.* Это обеспечит наибольшую плотность. Если поместить в банку много-много шариков, в ней все равно останется много пустого пространства. Добавив песка (небольших приложений), можно заполнить некоторые пробелы. Ограничения и допуски позволяют шаблону Operator (Оператор) группировать модули и управлять их развертыванием.

Еще один фактор, который необходимо учитывать, – это *шумные соседи*. В зависимости от рабочей нагрузки некоторые из настроек могут оказаться неподходящими. Однако вы можете более равномерно распределить «шумные» приложения в своем кластере Kubernetes, используя определения соответствия/несоответствия (affinity/anti-

affinity) модулей Pod. А используя автоматическое масштабирование и эфемерные облачные виртуальные машины, можно еще больше сократить расходы. Также иногда помогает простое нажатие выключателя. Многие компании имеют отдельные кластеры для разработки и контроля качества. Если не требуется, чтобы среда разработки работала на выходных, то зачем оставлять ее работающей? Просто уменьшите количество рабочих узлов в плоскости управления до нуля, а когда потребуется выполнить резервное копирование кластера, увеличьте его.

Итоги

- Pod – это базовый объект Kubernetes API, который использует пространства имен Linux с целью создания среды выполнения для одного или нескольких контейнеров.
- Платформа Kubernetes была создана для запуска модулей Pod с применением различных шаблонов, которые тоже являются объектами API: Deployment, StatefulSet и т. д.
- Контроллеры – это программные компоненты, управляющие жизненным циклом модулей Pod. К ним относятся kubelet, облачный диспетчер контроллеров (CCM) и планировщик.
- Плоскость управления – это мозг Kubernetes. С ее помощью Kubernetes может подключать хранилища к процессам, запускать контейнеры, масштабировать количество контейнеров, останавливать и перемещать контейнеры, обнаружив их неработоспособность, создавать IP-маршруты к портам, обновлять конечные точки с балансировкой нагрузки и регулировать многие другие аспекты управления распределенными приложениями.
- Сервер API (компонент kube-apiserver) проверяет и предоставляет веб-интерфейс для выполнения операций CRUD с общим состоянием кластера. Большинство плоскостей управления имеют сервер API, работающий на каждом узле, входящем в плоскость управления, благодаря чему обеспечивается высокая доступность кластера для сервера API.

3

Создание модулей Pod

В этой главе:

- основы примитивов Linux;
- использование примитивов Linux в Kubernetes;
- создание собственного модуля Pod без использования Docker;
- почему некоторые плагины Kubernetes эволюционировали со временем.

В этой главе вы познакомитесь с приемами создания модулей Pod с помощью примитивов Linux, уже существующих в вашей ОС. Это основные строительные блоки в ОС Linux, предназначенные для управления процессами, и вы скоро узнаете, что их можно использовать для создания сложных административных программ или выполнения основных повседневных задач, требующих доступа к функциональным возможностям уровня ОС. Особенность этих примитивов заключается в том, что они послужили основой для реализации многих важных аспектов Kubernetes.

Мы также рассмотрим причины необходимости *провайдеров CNI* – выполняемых программ, предоставляющих IP-адреса модулям Pod. Наконец, мы посмотрим, какую роль играет kubelet в запуске контейнера. Начнем с небольшого вступления, чтобы определить контекст для дальнейшего обсуждения.

Пробное приложение гостевой книги

В главе 15 мы рассмотрим реалистичное приложение Kubernetes с интерфейсом, сетью и серверной частью. Вы можете смело перейти к этой главе, чтобы получить общее представление о том, как работают модули Kubernetes с точки зрения приложения.

Схема на рис. 3.1 иллюстрирует процесс создания и запуска модуля Pod в Kubernetes. Это очень упрощенная схема, и в следующих главах мы будем постепенно развивать некоторые детали, изображенные на этом рисунке. А пока отметим лишь, что между моментом создания модуля Pod и объявлением о его переходе в рабочее состояние проходит большой промежуток времени. Мы предполагаем, что вы сами уже пробовали запустить несколько модулей Pod и хорошо знаете об этой задержке. Что происходит в этот период времени? Для создания так называемого контейнера вызывается множество примитивов Linux, которые вы, вероятно, не используете в повседневной работе. Проще говоря:

- kubelet выясняет, что должен запустить контейнер;
- затем kubelet (обращением к среде выполнения контейнеров) запускает *приостановленный контейнер* (так называемый контейнер pause), что дает ОС Linux время на создание сети для контейнера. Этот приостановленный контейнер является предшественником фактического приложения, которое будет запущено. Его цель – создать дом для начальной загрузки нового сетевого процесса контейнера и его идентификатора процесса (PID);
- во время запуска состояние различных компонентов меняется, как показано на каждом треке на рис. 3.1. Например, провайдер CNI в основном бездействует, за исключением времени, необходимого для привязки приостановленного контейнера к сетевому пространству имен.

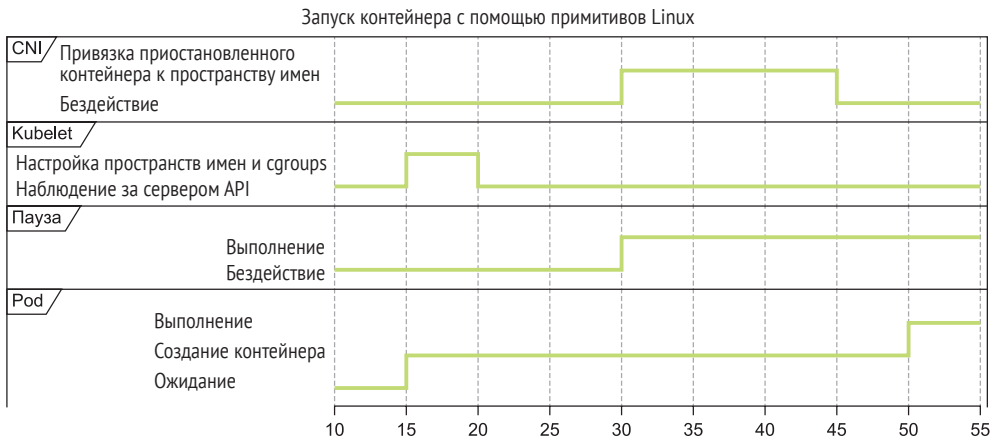


Рис. 3.1 Запуск контейнера с помощью примитивов Linux

Ось *x* на рис. 3.1 представляет относительную шкалу времени запуска модуля Pod. С течением времени выполняется несколько операций, включая монтирование *подпути* (связывание внешних каталогов хранилища, доступных контейнеру для чтения и записи). Это происходит в период времени, предшествующий 30-секундной отметке, когда модуль Pod переходит в рабочее состояние. Как уже упоминалось, различные команды Linux используют базовые примитивы, запускаемые kubelet, чтобы перевести модуль Pod в конечное рабочее состояние.

Хранилище, привязка точек монтирования и подпути

Хранилище в модулях Pod Kubernetes обычно монтируется с привязкой (когда папки, находящиеся в одном месте, подключаются к другому месту в дереве каталогов). Это позволяет контейнерам «видеть» каталог по определенному пути в своем дереве каталогов. Это базовая функция Linux, которая часто используется при монтировании ресурсов NFS.

Привязка точек монтирования используется «за кулисами» во всех системах для реализации многих важных функций Kubernetes. В том числе и для предоставления модулям Pod доступа к хранилищу. Для исследования каталогов, доступных изолированным процессам, можно использовать такие инструменты, как *nsenter*, не прибегая к использованию среды выполнения контейнеров (такой как Docker или *crictl*).

Поскольку *nsenter* – это простой выполняемый файл Linux, работающий с API базовой ОС, его всегда можно использовать независимо от того, находитесь ли вы в конкретном дистрибутиве Kubernetes или нет. Этот инструмент можно использовать, даже если Docker или *crictl* недоступны. Однако *nsenter* нельзя использовать для исследования кластеров Windows, где выше зависимость от инструментов среды выполнения контейнеров.

В качестве примера фундаментальной природы этих примитивов рассмотрим комментарии к следующему коду, который находится в файле `pkg/volume/util/subpath/subpath_linux.go` самой платформы Kubernetes, расположенном здесь: <http://mng.bz/8Ml2>. Они демонстрируют широкое применение этих примитивов в реализации Kubernetes:

```
func prepareSubpathTarget(mounter mount.Interface, s Subpath)
(bool, string, error) { ... }
```

← Создает цель для привязки точки
монтирования к подпути

После того как функция `prepareSubpathTarget` создаст цель для привязки точки монтирования подпути, этот подпуть становится доступным внутри контейнера, даже если он создан в kubelet. Ранее эту функциональность предоставляла функция `NsEnterMounter`, предна-

значенная для выполнения различных операций с каталогами внутри контейнеров. Возможно, вам никогда не придется читать этот код, и тем не менее мы считаем, что полезно знать о наличии ссылок на `nsenter` в самой платформе Kubernetes.

В прошлом `nsenter` использовался в Kubernetes для устранения ошибок или выяснения особенностей управления хранилищами средами выполнения контейнеров. Точно так же если вам доведется столкнуться с проблемой, связанной с хранилищем или монтированием каталогов, то вы будете знать, что в Linux есть инструменты, способные делать то же самое, что и `kubelet` или ваша среда выполнения контейнеров, и сможете с их помощью выяснить причины; `nsenter` – это всего лишь один пример простой команды Linux.

3.1 Общий обзор примитивов Kubernetes

Начнем с простого кластера, с помощью которого можно исследовать некоторые из этих концепций. Нет более простого способа создать ссылку на среду Kubernetes, чем использовать `kind` (<https://kind.sigs.k8s.io>), инструмент разработчика для Kubernetes. Этот кластер станет нам основой для многих экспериментов в этой книге.

Во-первых, нужно настроить простую среду Linux для экспериментов. Для этого мы будем использовать `kind` – инструмент, позволяющий запускать кластеры Kubernetes (с одним или несколькими узлами) внутри среды выполнения контейнеров, такой как Docker. `kind` работает в любой ОС, где создает новый контейнер для каждого узла. Мы можем сравнить кластер `kind` с реальным кластером, как показано в табл. 3.1.

Таблица 3.1 Сравнение `kind` с настоящим кластером

Тип кластера	Администратор	Тип kubelet	Поддержка подкачки
<code>kind</code>	Вы (пользователь Docker)	Контейнер Docker	Да (не подходит для промышленного использования)
GKE (Google Kubernetes Engine)	Google	Узел GCE (Google Compute Engine)	Нет
Cluster API	Действующий мастер кластера	Виртуальная машина в облаке по вашему выбору	Нет

В табл. 3.1 проводится сравнение кластера `kind` с обычными кластерами, которые используются в промышленном окружении. Кластер `kind` имеет много черт, делающих его непригодным для промышленного использования. Например, он допускает возможность подкачки ресурсов, а это означает, что контейнеры могут использовать дисковое пространство для хранения страниц виртуальной памяти. Это приводит к снижению производительности, если множеству контейнеров внезапно потребуется больше памяти. Однако `kind` очень удобен для обучения, потому что:

- не требует затрат на оплату облачных услуг;
- устанавливается за считанные секунды;
- перестраивается, если потребуется, за считанные секунды;
- поддерживает базовые функции Kubernetes;
- способен работать практически с любым провайдером сети или хранилища, поэтому может использоваться для работы с Kubernetes на начальном этапе.

В нашем случае мы будем использовать `kind` и контейнеры Docker, чтобы не только запустить несколько примеров, но и получить легковесную виртуальную машину Linux для экспериментов. Ближе к концу главы мы углубимся в сетевые возможности Kubernetes и посмотрим, как использовать команду `iptables` для маршрутизации трафика внутри кластера Kubernetes.

Настройка компьютера

Прежде чем начать, затронем вопрос настройки вашего компьютера. Мы предполагаем, что вы имеете опыт использования веб-сайта <https://kubernetes.io> и различных поисковых систем, умеете находить самую свежую информацию об установке программ и устанавливали пакеты в Linux. Мы не будем давать вам конкретных инструкций, которые нужно выполнить, но имейте в виду, что вам понадобится ОС Linux, чтобы следовать за этой главой:

- если вы используете Windows, то установите виртуальную машину с Linux с помощью VMware Fusion, VirtualBox или Hyper-V;
- если вы используете Linux, то сможете опробовать многие из примеров в этой главе с кластером `kind` или без него;
- если вы используете Mac, то просто загрузите рабочий стол Docker.

Если вы еще не настроили окружение для экспериментов, то рекомендуем потратить время на это. Если вы пользователь Linux, то, вероятно, уже имеете опыт самостоятельной настройки различных инструментов программирования. Всем остальным мы советуем поискать в интернете по фразе «запуск контейнеров Linux в Windows» или «как запустить Docker в OS X», чтобы найти инструкции, которые помогут вам начать работу в течение нескольких минут. Почти каждая современная ОС так или иначе поддерживает Docker.

3.2 Что такое примитивы Linux?

Как упоминалось выше, примитивы Linux являются основными строительными блоками ОС Linux. Примерами таких примитивов могут служить такие инструменты, как `iptables`, `ls`, `mount` и многие другие базовые программы, доступные в большинстве дистрибутивов Linux. Вы почти наверняка использовали хотя бы некоторые из этих команд

раньше, если занимались разработкой программного обеспечения. Например, команда `ls` – один из первых инструментов, которые изучает каждый, кто работает с терминалом в Linux. Она выводит список файлов, находящихся в текущем каталоге. Если передать ей аргумент (например, `/tmp`), то она выведет список файлов в этом каталоге.

Знание основ этих инструментов дает мощный толчок в понимании множества новых плагинов и надстроек в экосистеме Kubernetes, потому что все они построены на основе одного и того же набора фундаментальных строительных блоков:

- *сетевой прокси-сервер kube-проху создает правила iptables, и эти правила часто приходится проверять для устранения проблем с сетью контейнеров в больших кластерах.* Получить эти правила можно, запустив команду `iptables -L` в узле Kubernetes. Провайдеры Container Network Interface (CNI) тоже используют этот сетевой прокси-сервер (например, для различных задач, связанных с реализацией NetworkPolicies);
- *интерфейс Container Storage Interface (CSI) определяет сокет для взаимодействий kubelet с технологиями хранения.* Сюда входят такие ресурсы, как Pure, GlusterFS, vSAN, Elastic Block Store (EBS), Network File System (NFS) и т. д. Например, с помощью команды `mount` можно увидеть смонтированные в кластере контейнеры и тома, управляемые платформой Kubernetes, не привлекая `kubectl` и другие инструменты, не входящие в стандартную конфигурацию ОС. Как следствие, она часто используется для отладки и устранения низкоуровневых ошибок, возникающих в хранилищах Kubernetes;
- *команды среды выполнения контейнеров, такие как unshare и mount, используются при создании изолированных процессов.* Они широко используются технологиями создания контейнеров. Доступность этих команд (которым для работы часто требуются привилегии суперпользователя) является важной границей безопасности в сценариях моделирования угроз в кластере Kubernetes.

Команда `ls` обычно вызывается из командной оболочки напрямую или из сценариев командной оболочки. Многие команды Linux можно запускать из оболочки, и часто они возвращают текстовый вывод, который можно передать на вход следующей команде. Причина, по которой мы часто обращаемся к `ls` в сценариях, состоит в возможности сочетать ее с другими программами (например, чтобы применить некоторую команду ко всем файлам в каталоге).

3.2.1 Примитивы Linux – это инструменты управления ресурсами

Системное администрирование предполагает управление ресурсами на машине. Команда `ls` кажется простой программой, но, с точки зре-

ния администратора, это – мощный инструмент управления ресурсами. Администраторы ежедневно используют эту программу, отыскивая файлы большого размера или проверяя права доступа к файлам, нехватка которых мешает пользователям выполнять их работу. Команда `ls` позволяет проверить:

- возможность доступа к определенному файлу;
- права доступа к файлам в произвольном каталоге;
- разрешения, назначенные конкретному файлу (например, разрешение на выполнение).

Разговор о файлах подводит нас к следующему аспекту примитивов Linux: в этой ОС *все сущее является файлом*. Это ключевое отличие Linux от других ОС, таких как Windows. Фактически, когда в кластере Kubernetes запускаются узлы Windows, возможности проверки и мониторинга состояния сужаются, и для выяснения нужных сведений порой приходится прикладывать значительные усилия, потому что отсутствует единообразное представление объектов. Например, многие объекты Windows хранятся в памяти, доступны только через Windows API и недоступны через файловую систему.

«Все сущее является файлом» – уникальная особенность Linux

В Windows администрирование машин часто сопряжено с редактированием реестра. Это требует запуска пользовательских программ, причем нередко с графическим интерфейсом. Конечно, есть возможность решения многих задач системного администрирования с помощью PowerShell и других инструментов, однако невозможно полноценно администрировать всю ОС Windows, просто читая и записывая файлы.

Администрирование Linux, напротив, почти целиком заключается в управлении текстовыми файлами. Например, администраторам хорошо известен каталог `/proc`, в котором находится информация о запущенных процессах, меняющаяся в режиме реального времени. Во многих отношениях этим каталогом можно управлять, как если бы он был простым каталогом с обычными файлами, даже при том что это совсем не «обычный» каталог.

3.2.2 Все сущее является файлом (или файловым дескриптором)

Примитивы Linux почти всегда представляют свои операции как операции с каким-либо файлом, потому что *все*, что вам нужно построить с помощью Kubernetes, изначально создавалось для работы в Linux, а Linux изначально разрабатывалась для использования файловой абстракции в качестве примитива управления.

Например, команда `ls` работает с файлами. Она просматривает файл (который является каталогом) и читает имена файлов в ката-

логе из этого файла. Затем она выводит эти имена в другой файл, известный как *стандартный вывод*. Стандартный вывод – это не обычный файл в широком представлении; это файл, при записи в который информация волшебным образом появляется в терминале. Когда мы говорим, что в Linux все сущее является файлом, то именно это мы и имеем в виду!

- *Каталог* – это файл, содержащий имена других файлов.
- *Устройства тоже представлены файлами*. Поскольку устройства доступны в виде файлов, это означает, что можно использовать такие команды, как `ls`, чтобы проверить, например, подключено ли устройство Ethernet внутри контейнера.
- *Сокеты и каналы также являются файлами, которые процессы могут использовать локально для взаимодействий*. Позже вы увидите, что CSI интенсивно использует эту абстракцию для определения способа взаимодействий kubelet с провайдерами томов, предоставляющими хранилище для модулей Pod.

3.2.3 Файлы можно комбинировать

Объединив предыдущие концепции управления файлами и ресурсами, мы подошли к самой важной черте примитивов Linux: их можно комбинировать в операции более высокого уровня. Используя канал (`|`), можно получить вывод одной команды и передать его для обработки другой команде. Одним из самых популярных применений канала является объединение `ls` с командами `grep` для фильтрации списка файлов.

Например, типичная задача администрирования Kubernetes – проверка работоспособности `etcd` внутри кластера. Внутри контейнера на узле, где запущены компоненты плоскости управления Kubernetes (которые почти всегда запускают критический процесс `etcd`), можно выполнить следующую команду:

```
$ ls /var/log/containers/ | grep etcd
etcd-kind-control-plane_kube-system_etcd-44daab302813923f188d864543c...log
```

Точно так же в произвольном кластере Kubernetes можно узнать, где находятся ресурсы конфигурации, связанные с `etcd`, выполнив примерно такую команду:

```
$ find /etc | grep etcd; find /var | grep etcd
```

На этом мы закончим теоретические изыскания и далее погрузимся в практику, занявшись исследованием множества команд Linux, позволяющих нам создать собственный модуль Pod с нуля. Но, прежде чем приступить к экспериментам, создадим кластер с помощью `kind`.

etcd в контейнере?

Возможно, вам интересно, почему etcd работает в контейнере. Чтобы у вас не сложилось неправильного представления, сразу отметим, что в промышленных кластерах принято запускать etcd отдельно от остальных контейнеров, чтобы исключить вероятность конкуренции за драгоценные ресурсы диска и процессора. Однако во многих небольших кластерах или в кластерах для разработки все компоненты плоскости управления для простоты запускаются в одном месте. Тем не менее многие решения Kubernetes демонстрируют, что etcd может прекрасно работать в контейнере, если тома для этого контейнера находятся на локальном диске, чтобы они не терялись при перезапуске контейнера.

3.2.4 Настройка kind

kind – это уникальный инструмент, поддерживаемый сообществом Kubernetes. Он создает кластеры Kubernetes внутри контейнеров Docker без любых других зависимостей. Это позволяет разработчикам моделировать реалистичные кластеры с множеством узлов на локальном компьютере без создания виртуальных машин или использования других тяжеловесных конструкций. Он не подходит для промышленного использования и применяется только для разработки или исследований. Чтобы продолжить, создадим несколько кластеров (инструкции по созданию нашего первого кластера для этой главы вы найдете по адресу <http://mng.bz/voVm>).

kind устанавливается за несколько секунд в любой ОС и позволяет запускать Kubernetes внутри Docker. Мы будем рассматривать каждый контейнер Docker как виртуальную машину и использовать эти контейнеры для исследования различных свойств Linux в целом. Процесс настройки kind в качестве базовой среды Kubernetes выглядит просто.

- 1 Установить Docker.
- 2 Установить kubect`l` в `/usr/local/bin/kubectl`.
- 3 Установить kind в `/usr/local/bin/kind`.
- 4 Проверить установку, выполнив команду `kubectl get pods`.

Почему мы используем kind? В этой книге приводится много примеров, поэтому если вы захотите опробовать их (что мы настоятельно рекомендуем, хотя для работы с книгой это необязательно), то вам понадобится какая-то среда Linux. А поскольку мы говорим о Kubernetes, мы выбрали kind по причинам, описанным выше. Однако использование kind не является обязательным требованием. Если вы опытный пользователь, знакомы с основами и просто хотите углубиться в исследование более сложных аспектов, то можете опробовать многие из представленных команд в любом кластере Kubernetes. Конечно же, мы предполагаем, что при этом вы будете использовать некоторую разновидность Linux, потому что контрольные группы (cgroups), пространства имен Linux и другие фундаментальные при-

митивы Kubernetes по умолчанию недоступны в коммерческих ОС, таких как Windows и Mac OS X.

Пользователям Windows

В Windows kind устанавливается как выполняемый файл. Мы рекомендуем прочитать инструкции по адресу <https://kind.sigs.k8s.io/docs/user/quick-start/>. Он даже поддерживает команды `choco install`, которые вы можете попробовать запустить. Пользователи Windows смогут выполнять все команды из этой книги в подсистеме Windows для Linux (Windows Subsystem for Linux, WSL2) – облегченной виртуальной машине Linux, которую легко запустить на любом компьютере с Windows.

Обратите внимание, что `kubectl` тоже можно запустить как выполняемый файл Windows, чтобы подключиться к удаленному кластеру в любом облаке. И хотя в этой книге отдается предпочтение Linux и OS X, мы готовы поддержать ваше решение опробовать эти команды на компьютере с Windows!

Следующий шаг после установки `kind` – создание кластера. Для этого выполните следующие команды:

```
$ kind delete cluster --name=kind
Deleting cluster "kind" ...
```

← Удалит прежний кластер kind, если имеется

```
$ kind create cluster
Creating cluster "kind" ...
? Ensuring node image (kindest/node:v1.17.0) ?
? Preparing nodes ?
? Writing configuration ?
?? Starting control-plane ?
```

← Запустит новый кластер kind

Теперь можно посмотреть, какие модули Pod выполняются в кластере. Чтобы получить список модулей, введите команду, как показано в примере ниже, где также приводится пример вывода:

```
$ kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	AGE
kube-system	coredns-6955-6bb2z	1/1	Running	3m24s
kube-system	coredns-6955-82zzn	1/1	Running	3m24s
kube-system	etcd-kind-control-plane	1/1	Running	3m40s
kube-system	kindnet-njvrs	1/1	Running	3m24s
kube-system	kube-proxy-m9gf8	1/1	Running	3m24s

Если вам интересно, где находится ваш узел Kubernetes, вы легко сможете это узнать, просто спросив Docker:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
776b91720d39	kindest/node:v1.17.0	"/usr/local/bin/entr..."

CREATED	PORTS	NAMES
4 minutes ago	127.0.0.1:32769->6443/tcp	kind-control-plane

Наконец, если вы – системный администратор и вам нужна возможность подключаться к своим узлам по ssh, то можно это сделать, используя команду:

```
$ docker exec -t -i 776b91720d39 /bin/sh
```

вместе с командами, подобными перечисленным выше в этом разделе. Они будут выполняться внутри узла (который на самом деле является контейнером).

Кстати, возможно, вам интересно, как Kubernetes может работать в Docker. Означает ли это, что контейнеры Docker могут запускать другие контейнеры Docker? Да, это так. Если заглянуть в определение образа Docker для kind (<http://mng.bz/nYg5>), можно увидеть, как именно он работает. Здесь, в частности, можно увидеть все устанавливаемые примитивы Linux, в том числе обсуждавшиеся нами. Чтение этого кода может стать отличным самостоятельным упражнением, которое стоит выполнить после прочтения этой главы.

3.3 Использование примитивов Linux в Kubernetes

Особенности работы основных функций в Kubernetes часто прямо или косвенно связаны с работой базовых примитивов Linux. Эти примитивы образуют основу для запуска контейнеров, к которой вы будете постоянно возвращаться. Со временем вы обнаружите, что многие технологии, использующие такие модные словечки, как «сервисная сетка» или «собственное хранилище контейнеров», сводятся к искусно подобранной коллекции одних и тех же фундаментальных возможностей ОС.

3.3.1 Предварительные условия для запуска модуля Pod

Напоминаем, что *Pod* – это основная единица выполнения в кластере Kubernetes: именно так мы определяем контейнеры, действующие в нашем центре обработки данных. Конечно, существуют сценарии использования Kubernetes для решения задач, не связанных с запуском контейнеров, но в этой книге мы не будем касаться их. В конце концов, мы предполагаем, что основной интерес для вас представляют использование и понимание Kubernetes в традиционном контексте.

Для создания модуля Pod мы полагаемся на возможности изоляции, сетевых взаимодействий и управления процессами. Все это может быть реализовано с помощью утилит, уже доступных в ОС Linux. На самом деле некоторые из этих утилит можно считать *обязатель-*

ными, потому что без них kubelet не сможет выполнить действия, необходимые для запуска модуля Pod. Давайте кратко рассмотрим некоторые программы (или *примитивы*), на которые мы полагаемся в повседневной работе с кластерами Kubernetes:

- `swapon` – команда, отключающая подкачку памяти, что является известным предварительным условием для запуска Kubernetes, позволяющим исключить конкуренцию за процессор и память;
- `iptables` – основное требование (обычно) сетевого прокси-сервера, который создает правила `iptables` для отправки служебного трафика модулям Pod;
- `mount` – эта команда (упоминавшаяся выше) проецирует ресурс в определенное место в файловой системе (например, она позволяет отобразить устройство как папку в домашнем каталоге);
- `systemd` – эта команда обычно запускает kubelet – основной процесс, управляющий всеми контейнерами в кластере;
- `socat` – эта команда позволяет установить двунаправленную связь между процессами; `socat` обеспечивает правильное функционирование команды `kubectl port-forward`;
- `nsenter` – инструмент для входа в различные пространства имен процесса и просмотра происходящего (с точки зрения сети, хранилища или процесса). Точно так же, как пространство имен в Python имеет определенные модули с локальными именами, пространство имен Linux имеет определенные ресурсы, к которым невозможно обратиться из внешнего мира. Например, уникальный IP-адрес модуля Pod в кластере Kubernetes не используется другими модулями, даже на том же узле, потому что каждый модуль (обычно) работает в отдельном пространстве имен;
- `unshare` – команда, позволяющая запускать дочерние процессы, выполняющиеся изолированно. В этой главе мы воспользуемся ею для изучения знаменитого феномена *Pid 1* в контейнерах, когда каждый контейнер в кластере Kubernetes думает, что он является единственной программой во всем мире;
- `unshare` также может изолировать точки монтирования (каталог `/`) и сетевые пространства имен (IP-адреса), и поэтому ее можно считать прямым аналогом команды `docker run`, имеющимся в ОС Linux;
- `rs` – программа, отображающая список запущенных процессов. Агент kubelet должен постоянно следить за процессами, чтобы всегда быть в курсе, например, когда они завершаются. С помощью команды `rs` можно определить, имеются ли в кластере процессы-зомби, не начал ли привилегированный контейнер проявлять неуправляемое поведение (создавая много новых подпроцессов) и т. д.

3.3.2 Запуск простого модуля Pod

Прежде чем переходить к использованию этих команды, давайте взглянем на модуль Pod, используя наш кластер `kind` для его соз-

дания. Обычно модули Pod создаются не вручную, а с помощью объектов Deployment, DaemonSet или Job. Но пока оставим эти высокоуровневые конструкции в стороне и создадим простой одиночный модуль Pod. Используем для этого команду `kubectl create -f pod.yaml`, предварительно создав YAML-файл, представленный ниже. Но перед созданием модуля давайте кратко пробежимся по этому YAML-файлу:

- для тех, кому интересно, что содержит образ *BusyBox*: это минимальный образ с ОС Linux, который можно запустить для изучения поведения контейнеров по умолчанию. В примерах часто используется модуль Pod NGINX, но мы выбрали BusyBox, потому что он включает команду `ip` и другие основные утилиты. Часто из контейнеров с микросервисами промышленного уровня удаляются лишние двоичные файлы, чтобы максимально уменьшить вероятность вторжения;
- для тех, кому интересно, почему мы определяем `webapp-port`: этот модуль Pod служит единственной цели – знакомству с синтаксисом определения модулей Pod, но в нем не запускается никаких служб, прослушивающих порт 80, однако если вы замените этот образ чем-то вроде NGINX, то этот порт станет конечной точкой с балансировкой нагрузки, которую вы сможете использовать для доступа к службе Kubernetes.

```
$ cat << EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
  labels:
    role: just-an-example
    app: my-example-app
    organization: friends-of-manning
    creator: jay
spec:
  containers:
    - name: any-old-name-will-do
      image: docker.io/busybox:latest
      command: ['sleep', '10000']
      ports:
        - name: webapp-port
          containerPort: 80
          protocol: TCP
EOF
$ kubectl create -f pod.yaml
```

Метаданные метки позволяют выбрать этот модуль Pod в качестве цели балансировщика нагрузки или фильтра в запросе к серверу Kubernetes API

Имя образа `docker.io` должно быть реальным и либо доступным в интернете, либо отмечено как образ локального контейнера

Создаст модуль Pod в пространство имен Kubernetes по умолчанию

Не путайте инструмент `kind`, который мы использовали для создания этого кластера, с полем `kind` в определении модуля Pod, которое сообщает серверу API тип создаваемого объекта. Если в этом поле указать другой тип (например, `Service`), то сервер API попытается создать

объект другого типа. То есть в самом начале определения модуля Pod мы указываем тип (или *вид* – *kind*) Pod. Ниже в этом разделе мы запустим команду, которая покажет нам настроенные маршруты и IP-конфигурацию этого модуля, так что держите это определение под рукой!

Теперь, создав модуль, давайте посмотрим, можно ли увидеть его процессы со стороны нашей ОС. Как показывает следующий пример, процессы действительно видны. Команда `ps -ax` дает простой и быстрый способ получить список всех процессов в системе, включая не имеющие терминала. Параметр `-x` особенно важен, потому что мы имеем дело с программным обеспечением на уровне системы, а не пользователя, соответственно, нам нужно знать глобальное количество запущенных программ, чтобы проиллюстрировать видимость процесса:

```
$ ps -ax | wc -l  ← Количество выполняющихся
706              процессов до создания модуля Pod

$ kubectl create -f pod.yml  ← Создание модуля
pod "core-k8s" deleted

$ ps -ax | wc -l  ← Количество выполняющихся процессов
707              после создания модуля Pod
```

3.3.3 Исследование зависимостей модуля Pod от Linux

После запуска модуля Pod в кластере он запускает программу, которой требуется доступ к основным вычислительным ресурсам, таким как процессор, память, диск и т. д. Чем Pod отличается от обычной программы? С точки зрения конечного пользователя – ничем. Например, как любая нормальная программа, модуль Pod:

- использует общие библиотеки или низкоуровневые утилиты конкретной ОС для поддержки ввода с клавиатуры, получения списков файлов и т. д.;
- получает доступ к клиенту, способному использовать реализацию стека TCP/IP для выполнения сетевых вызовов и получения уведомлений (их часто называют *системными вызовами*);
- требует выделения некоторого изолированного адресного пространства в памяти, чтобы другие программы не могли выполнять запись в его память.

При создании этого модуля kubelet выполняет множество действий, которые выполняет любой другой пользователь, пытающийся запустить компьютерную программу:

- создает изолированный дом для запуска программы (с ограничениями на доступ к процессору, памяти и пространствам имен);
- гарантирует наличие в доме работающего соединения Ethernet;
- предоставляет доступ к некоторым основным файлам для разрешения имен или для доступа к хранилищу;

- сообщает программе, что она может безопасно войти в этот дом и начать работу;
- ожидает завершения программы;
- прибирает дом за программой и освобождает использованные ею ресурсы.

Мы обнаружим, что почти все действия в Kubernetes повторяют самые обычные административные задачи, которые мы выполняли десятилетиями. Другими словами, kubelet просто следует по пунктам руководства системного администратора Linux.

Этот процесс можно представить как *жизненный цикл* модуля Pod – циклический процесс, отражающий фундаментальный цикл управления, который определяет, что делает сам агент kubelet во время работы (рис. 3.2). Поскольку контейнеры в модуле Pod могут выйти из строя в любой момент, на этот случай существует цикл управления, возвращающий их к жизни. Такие циклы управления распространяются в Kubernetes «фрактально». На самом деле можно сказать, что сам Kubernetes – это всего лишь сложно организованный набор циклов управления, которые позволяют автоматически запускать и управлять контейнерами в больших масштабах.

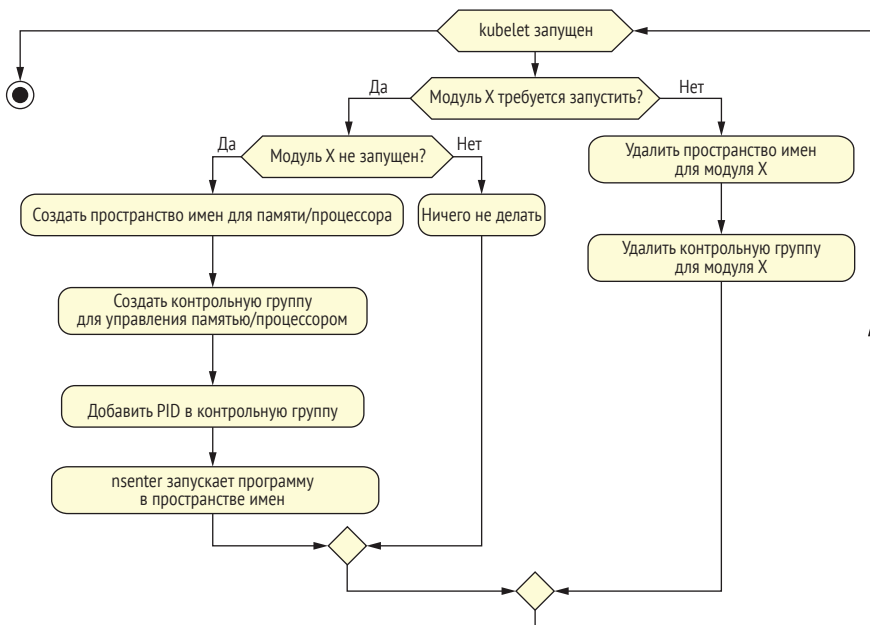


Рис. 3.2 Цикл управления жизненным циклом модуля в kubelet

Одним из самых низкоуровневых циклов управления является сам жизненный цикл kubelet/Pod. На рис. 3.2 завершение работы kubelet изображено в виде точки. Пока kubelet работает, он выполняет непрерывный цикл согласования, в котором проверяются и запускаются модули Pod. Здесь мы снова ссылаемся на nsenter как на одно из ни-

жестоящих действий kubelet, но обратите внимание, что nsenter (запускающий и управляющий контейнерами в Linux) нельзя перенести в другие окружения выполнения контейнеров или ОС.

Как мы вскоре покажем, модуль Pod можно создать, выполнив множество команд в правильном порядке и в нужное время, и получить в свое распоряжение все те замечательные возможности, что обсуждались в предыдущей главе. Схема на рис. 3.2 показывает несколько важных моментов, которые мы должны исследовать.

Какое место во всем этом занимает Docker?

Если вы новичок в Kubernetes, то вас, возможно, уже снедает вопрос, когда же мы наконец начнем говорить о Docker. На самом деле мы мало будем много говорить о Docker, так как это в большей степени инструмент разработчика, не относящийся к контейнерным решениям на стороне сервера.

Да, начиная с версии 1.20, Kubernetes поставляется с собственной поддержкой Docker, но в настоящее время полным ходом идет процесс отказа от явной поддержки Docker, и в конечном итоге сама платформа Kubernetes ничего не будет знать о среде выполнения контейнеров. Таким образом, несмотря на то что kubelet поддерживает жизненный цикл модулей Pod, предоставляя все необходимые ресурсы, он зависит только от интерфейса среды выполнения контейнеров (Container Runtime Interface, CRI), осуществляющей запуск и остановку модулей Pod, а также извлекающей образы для запуска в контейнерах.

Наиболее распространенной реализацией CRI является *containerd*, и на самом деле сама платформа Docker использует *containerd* за кулисами. Интерфейс CRI представляет доступ к некоторым (но не ко всем) функциям *containerd*, что упрощает реализацию собственных сред выполнения контейнеров для Kubernetes. Стандартный выполняемый файл *containerd* представляет kubelet с интерфейсом CRI, и, когда вызывается этот интерфейс, *containerd* (сервис) вызывает такие программы, как *runc* (в Linux) или *hcsshim* (в Windows).

После запуска модуля Pod в Kubernetes (или, по крайней мере, когда платформа Kubernetes знает, что модуль должен быть запущен) объект Pod на сервере API хранит полную информацию о состоянии модуля. Попробуйте самостоятельно выполнить следующую команду, чтобы увидеть эту информацию:

```
$ kubectl get pods -o yaml
```

Она выведет большой файл в формате YAML. Поскольку мы пообещали нашему модулю Pod собственный IP-адрес и место для запуска его процессов, давайте теперь убедимся в доступности этих ресурсов. Для этого используем *jsonpath* в запросе, чтобы отыскать конкретные детали.

ИССЛЕДОВАНИЕ МОДУЛЯ Pod С ИСПОЛЬЗОВАНИЕМ JSONPATH

Отфильтровав атрибуты из раздела `status`, рассмотрим некоторые из них. При этом мы будем использовать поддержку JSONPath в Kubernetes для фильтрации конкретной информации. Например:

```
$ kubectl get pods -o=jsonpath='{.items[0].status.phase}'
Running
$ kubectl get pods -o=jsonpath='{.items[0].status.podIP}'
10.244.0.11
$ kubectl get pods -o=jsonpath='{.items[0].status.hostIP}'
172.17.0.2
```

← Запросить состояние модуля Pod

← Запросить IP-адрес модуля

← Запросить IP-адрес хоста, на котором выполняется модуль

ПРИМЕЧАНИЕ Обратите внимание, что, кроме обновленной информации о состоянии модуля в `pod.yaml`, присутствует некоторая новая информация в разделе `spec`. Это обусловлено тем, что серверу API может потребоваться исправить некоторые настройки модуля перед отправкой. Например, такие параметры, как `terminationMessagePath` и `dnsPolicy`, которые часто не требуют внимания со стороны пользователя, но могут быть добавлены автоматически после того, как вы определите свой модуль Pod. В некоторых организациях также может иметься настраиваемый контроллер, «просматривающий» входящие объекты и изменяющий их перед передачей на сервер API (например, добавляя жесткие ограничения на потребление процессора или памяти).

В любом случае предыдущие команды помогают увидеть, что модуль Pod:

- контролируется операционной системой и имеет статус `Running`;
- находится в отдельном от хоста IP-пространстве (сетевом пространстве имен) `10.244.0.11/16`.

ИССЛЕДОВАНИЕ ДАННЫХ, СМОНТИРОВАННЫХ В МОДУЛЬ Pod

Одним из параметров, которые Kubernetes любезно определяет для всех своих модулей Pod, является `default-token`. Он предоставляет модулям сертификат, позволяющий связываться с сервером API и «звонить домой». В дополнение к томам Kubernetes мы передаем нашим модулям информацию о DNS. Чтобы увидеть ее, можно выполнить внутри модуля команду `mount` с помощью `kubectl exec`. Например:

```
$ kubectl exec -t -i core-k8s mount | grep resolv.conf
/dev/sda1 on /etc/resolv.conf type ext4 (rw,relatime)
```

Как видно из этого примера, команда `mount`, запущенная внутри контейнера, на самом деле показывает файл `/etc/resolv.conf` (кото-

рый сообщает Linux, где находятся DNS-серверы), смонтированный из другого места. Это место (/dev/sda1) – том на нашем хосте, где находится соответствующий файл resolv.conf. На самом деле в нашем примере mount выводит также другие местоположения с другими файлами, многие из которых на самом деле являются символическими ссылками, ведущими в каталог /dev/sda1 на хосте. Этот каталог обычно соответствует папке в /var/lib/containerd. Вы можете найти этот файл командой:

```
$ find /var/lib/containerd/ -name resolv.conf
```

Не заикивайтесь на этой детали. Это лишь часть базовой реализации kubelet, однако приятно знать, что эти файлы существуют в системе и их можно найти с помощью стандартных инструментов Linux (например, если вдруг кластер начнет проявлять неожиданное поведение).

Гипервизор (программный компонент, который создает виртуальные машины) понятия не имеет, какие процессы выполняются в созданных им виртуальных машинах. Однако в контейнерной среде все процессы, запущенные средой выполнения containerd (или Docker, или любой другой), активно управляются самой ОС. Это позволяет kubelet выполнять множество мелких задач по очистке и управлению, а также предоставлять серверу API важную информацию о состоянии контейнера.

Что особенно важно, этот пример показывает, что вы, как администратор, и агент kubelet способны управлять процессами и запрашивать информацию о них, исследовать тома, доступные этим процессам, и даже останавливать их, если потребуется. Много из этого может быть очевидным для вас, и все же мы хотим подчеркнуть, что в большинстве окружений Linux то, что мы называем контейнерами, – это просто процессы, созданные с помощью нескольких механизмов изоляции, позволяющих им ужиться с сотнями других процессов в кластере. Модули Pod и процессы почти не отличаются от программ, которые вы можете запускать обычным способом. В общем, наш модуль Pod:

- *имеет том хранилища с сертификатом для доступа к серверу API.* Благодаря этому модули Pod получают возможность обращаться к Kubernetes API. Кроме того, этот том служит основой для операторов и контроллеров, позволяющих модулям Pod создавать другие модули, сервисы и т. д.;
- *имеет IP-адрес в подсети 10.* Это не то же самое, что IP-адрес хоста в подсети 172;
- *способен обслуживать трафик через порт 80 в своем внутреннем пространстве имен по IP-адресу 10.244.0.11.* Другие модули Pod в кластере тоже могут получить к нему доступ;
- *успешно работает в кластере.* Теперь у него есть контейнер с уникальным PID, полностью управляемый и доступный нашему хосту.

Обратите внимание, что нам еще предстоит определить правила iptables для маршрутизации трафика нашему модулю Pod. При создании модуля без сервисов сеть обычно не настраивается в определении объекта Pod, потому что это, как правило, делает Kubernetes. Несмотря на то что IP-адрес нашего модуля Pod в кластере можно узнать, у нас все равно нет возможности балансировать нагрузку между репликами нашего модуля, потому что для этого нужны метки и селекторы меток, связанные с сервисами. О настройках сети в Kubernetes мы поговорим позже, в главе 4.

Теперь, познакомившись с базовыми возможностями простого модуля Pod в реальном кластере, давайте посмотрим, как можно получить те же возможности самостоятельно, используя базовые примитивы Linux. Пройдя процесс создания тех же возможностей без кластера, вы обретете множество новых знаний, которые улучшат ваше понимание особенностей работы Kubernetes, администрирования больших кластеров и приемов устранения неполадок в контейнерах в дикой природе. Пристегнитесь, поездка обещает быть захватывающей!

3.4 Создание модуля Pod с нуля

В этом разделе мы вернемся в прошлое и создадим систему управления контейнерами, подобную тем, что существовали до появления Kubernetes. С чего начнем? Выше мы рассмотрели четыре основных аспекта нашего модуля Pod, в частности:

- хранилище;
- IP-адрес;
- изолированную сеть;
- идентификатор процесса.

Для реализации всего этого придется использовать нашу базовую ОС Linux. К счастью, в нашем кластере уже есть базовая ОС Linux, которую можно использовать локально, поэтому нам не придется создавать выделенную виртуальную машину Linux для этого упражнения.

Будет ли этот модуль Pod работать?

Этот модуль немного иного типа, отличный от используемых в реальном мире. Он слабо защищен, и процесс, которому мы следуем, не поддерживает возможность масштабирования рабочих нагрузок. Но в основе своей это тот же процесс, которому следует kubelet, создавая контейнеры в любом облаке или центре обработки данных.

Итак, вернемся в наш кластер kind и приступим! Для этого нужно узнать идентификатор контейнера, что легко сделать, выполнив команду `docker ps | grep kind | cut -d ' ' -f 1`, а затем выполнить вход

на один из узлов командой `docker exec -t -i container_id /bin/sh`. Поскольку мы будем править текстовые файлы, нам понадобится текстовый редактор. Давайте установим редактор Vim (вы можете установить любой другой редактор, привычный вам):

```
root@kind-control-plane:/# sudo apt-get update -y
root@kind-control-plane:/# apt-get install vim
```

Если вы новичок в Kubernetes и прежде не использовали kind (и у вас голова идет кругом), то да, вы правы, мы сейчас работаем на метауровне. По сути, мы моделируем реальный кластер, а также отладку SSH, которую мы все знаем и любим. Методология запуска `docker exec` в кластерном kind (примерно) эквивалентна ssh-подключению к реальному узлу Kubernetes в реальном кластере.

3.4.1 Создание изолированного процесса с помощью chroot

Для начала создадим контейнер в самом прямом смысле – папку, в которой есть именно все, что нужно для запуска командной оболочки Bash, и больше ничего (как показано на рис. 3.3). Это делается с помощью известной команды `chroot`. (Когда технология Docker появилась, ее называли «chroot на стероидах».)

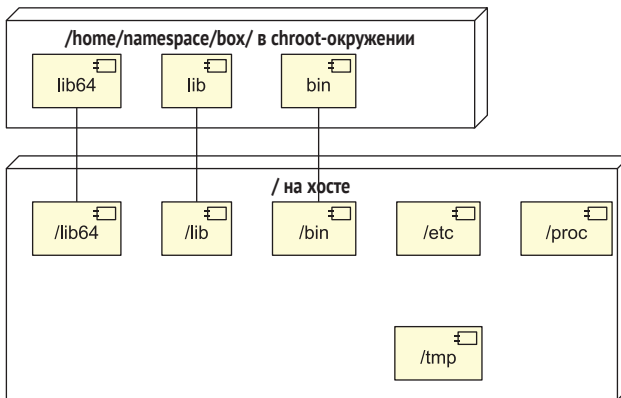


Рис. 3.3 Окружение `chroot` можно сравнить с корневой файловой системой

Назначение `chroot` – создать изолированную корневую файловую систему для процесса. Делается это в три шага.

- 1 Выбор программы, которую требуется запустить, и где в файловой системе она должна работать.
- 2 Создать среду для запуска процесса. В каталоге `lib64` в Linux находится множество программ, которые необходимы даже для запуска простой командной оболочки, такой как Bash. Их нужно загрузить в новую корневую файловую систему.
- 3 Скопировать программу для запуска в `chroot`-окружение.

Наконец, можно запустить программу, и она будет полностью изолирована от исходной файловой системы: она не сможет видеть или изменять другую информацию в вашей файловой системе (например, она не сможет изменять файлы в `/etc/` или `/bin/`).

Звучит знакомо? Так и должно быть! Когда мы запускаем контейнеры Docker в Kubernetes или где-то еще, мы всегда имеем такую же чистую изолированную среду выполнения. На самом деле, заглянув в список рассылки Kubernetes, вы гарантированно найдете множество прошлых и настоящих проблем и вопросов, связанных с функциональностью на основе `chroot`. Теперь давайте посмотрим, как работает `chroot`, запустив терминал Bash в `chroot`-окружении.

Следующий сценарий создает `chroot`-окружение, в котором можно запустить сценарий Bash или другую программу Linux. Этому сценарию недоступна вмещающая система, а это означает, что, запустив команду `rm -rf /` внутри `chroot`-окружения, мы не уничтожим никаких файлов в фактической ОС. Конечно, мы не рекомендуем пробовать это у себя, если только вы не используете компьютер, информацию на котором не жалко потерять, потому что одна крошечная ошибка может привести к большим потерям. Мы сохраним этот сценарий в локальной системе как `chroot.sh` на случай, если захотим использовать его повторно:

```
#/bin/bash
mkdir /home/namespace/box

mkdir /home/namespace/box/bin
mkdir /home/namespace/box/lib
mkdir /home/namespace/box/lib64

cp -v /usr/bin/kill /home/namespace/box/bin/
cp -v /usr/bin/ps /home/namespace/box/bin
cp -v /bin/bash /home/namespace/box/bin
cp -v /bin/ls /home/namespace/box/bin

cp -r /lib/* /home/namespace/box/lib/
cp -r /lib64/* /home/namespace/box/lib64/

mount -t proc proc /home/namespace/box/proc

chroot /home/namespace/box /bin/bash
```

Создают структуру каталогов для `chroot`-окружения, необходимую для нашей программы Bash

Копируют все программы из базовой ОС в это окружение, чтобы можно было запустить Bash в новом корневом каталоге

Копируют библиотечные зависимости программ в каталоги `lib/`

Монтирует каталог `/proc` сюда

Это важная команда: она запускает изолированный процесс Bash в изолированном каталоге

Напомним, что наш корневой каталог (`/`) не будет содержать никаких программ, которые не были загружены явно. Это означает, что `/` не имеет глобального доступа к обычному дереву каталогов Linux, где находятся обычные программы, запускаемые каждый день. По этой причине в предыдущем примере мы скопировали `kill` и `ps` (две основные программы) непосредственно в наш каталог `/home/namespace/box/bin`. А так как мы смонтировали каталог `/proc` в `chroot`-окружение,

можно видеть процессы на нашем хосте. Это позволяет использовать `ps` для изучения границ безопасности `chroot`-окружения. На данный момент вы должны понимать, что:

- некоторые команды, такие как `cat` или `cp`, недоступны в `chroot`-окружении, тогда как `ps` и `kill` будут выполняться, как в любой ОС Linux;
- другие команды (например, `ls /`) возвращают совершенно другие результаты, чем те, которые выводятся в полноценной ОС;
- в отличие от виртуальных машин, запускаемых на хосте, `chroot`-окружение не дает накладных расходов на производительность или задержек, потому что это обычная команда Linux. Если вы не пробовали запустить этот сценарий у себя, то вот его вывод:

```
root@kind-control-plane:/# ./chroot0.sh
'/bin/bash' -> '/home/namespace/box/bin/bash'
'/bin/ls' -> '/home/namespace/box/bin/ls'
'/lib/x86_64-linux-gnu/libtinfo.so.6' ->
'/home/namespace/box/lib/libtinfo.so.6'
'/lib/x86_64-linux-gnu/libdl.so.2' ->
'/home/namespace/box/lib/libdl.so.2'
'/lib64/ld-linux-x86-64.so.2' ->
'/home/namespace/box/lib/ld-linux-x86-64.so.2'

bash-5.0# ls
bin lib lib64
```

Будет весьма поучительно, если для сравнения запустить `ls` в корневом каталоге ОС Linux. Закончив исследовать бесплодную пустыню `chroot`, введите команду `exit`, чтобы вернуться в обычную ОС. Уф-ф, это было страшно!

Изолированное окружение `chroot` – один из основных строительных блоков контейнерной революции, продолжающейся и по сей день, хотя в течение некоторого времени она была известна как «виртуальная машина для бедных». Команда `chroot` часто используется администраторами Linux и разработчиками на Python для изолированного тестирования и запуска определенных программ. Если вы читали предыдущий раздел, где упоминалась фраза про «`chroot` на стероидах», то, возможно, теперь начинаете понимать ее смысл.

3.4.2 Использование `mount` для передачи данных процессам

Контейнерам обычно требуется доступ к хранилищу, находящемуся где-то в другом месте: в облаке или на хосте. Команда `mount` позволяет взять устройство и отобразить его в любой каталог в вашей ОС. Обычно `mount` используется для отображения дисков в виде папок. Например, в нашем кластере `kind` имеется несколько папок, управляемых Kubernetes, которые с помощью команды `mount` были сделаны доступными определенным контейнерам в определенном месте.

В простейшем случае, с точки зрения администратора, команда `mount` используется для отображения в некоторую постоянную папку диска, находящегося в каком-то другом произвольном месте. Например, представьте, что нам нужно запустить предыдущую программу, но так, чтобы она записывала данные во временное место, которое можно очистить позже. Мы могли бы выполнить простую команду `mount --bind /tmp/ /home/namespace/box/data` и создать каталог `/data` в предыдущем сценарии `chroot0.sh`. Тогда любой пользователь, создавший `chroot`-окружение с помощью этого сценария, получит каталог `/data` и сможет использовать его для доступа к файлам в нашем каталоге `/tmp`.

Обратите внимание, что этот шаг создает дыру в системе безопасности! После монтирования содержимого `/tmp` в контейнеры любой сможет читать или изменять его содержимое. Именно поэтому функциональность `hostPath` томов Kubernetes часто отключается в промышленных кластерах. В любом случае давайте убедимся, что действительно есть возможность передать некоторые данные в контейнер, созданный в предыдущем разделе, используя несколько основных примитивов Linux:

```
root@kind-control-plane:/# touch /tmp/a
root@kind-control-plane:/# touch /tmp/b
root@kind-control-plane:/# touch /tmp/c
root@kind-control-plane:/# ./chroot0.sh
'/bin/bash' -> '/home/namespace/box/bin/bash'
'/bin/ls' -> '/home/namespace/box/bin/ls'
'/lib/x86_64-linux-gnu/libtinfo.so.6' ->
'/home/namespace/box/lib/libtinfo.so.6'
'/lib/x86_64-linux-gnu/libdl.so.2' ->
'/home/namespace/box/lib/libdl.so.2'
'/lib64/ld-linux-x86-64.so.2' ->
'/home/namespace/box/lib/ld-linux-x86-64.so.2'
```

```
bash-5.0# ls data
a b c
```

Вот и все! Мы создали нечто похожее на контейнер, к тому же имеющее доступ к хранилищу. Позже рассмотрим более сложные аспекты контейнеризации, включая пространства имен для защиты процесса, памяти и сетевых ресурсов. Теперь просто ради удовольствия выполним команду `ps` и посмотрим, какие еще процессы присутствуют в нашем контейнере. Обратите внимание, что мы увидим наш процесс `bash` и еще несколько других:

```
bash-5.0# ps
  PID TTY          TIME CMD
  5027 ?           00:00:00 sh
 79455 ?           00:00:00 bash
 79557 ?           00:00:00 ps
```


3.4.3 Защита процесса с помощью unshare

Отлично! К данному моменту мы создали изолированное окружение для нашего процесса и подключили к нему папку. Очень похоже на модуль Pod, верно? Не совсем. Действительно, наше chroot-окружение изолировано от файловой системы хост-системы (например, команда `ls` внутри этого окружения видим только файлы, явно скопированные сценарием `chroot0.sh`). Но безопасно ли это окружение? Как оказывается, надеть шоры на процесс – это не совсем то же самое, что защитить его. Чтобы убедиться в этом:

- 1 выполните команду `ps -ax` внутри кластера, предварительно запустив в нем Docker, как мы делали это ранее;
- 2 найдите идентификатор процесса `kubelet` (например, 744). Чтобы упростить задачу, можно использовать команду `ps -ax | grep kubelet`;
- 3 снова запустите сценарий `chroot0.sh`;
- 4 выполните команду `kill 744` в приглашении к вводу `bash-5.0#`.

Вы сразу увидите, что только что остановили `kubelet`! Несмотря на то что процесс в `chroot`-окружении не имеет доступа к другим папкам (потому что мы переместили / в новый корень), он может найти и остановить критически важные системные процессы. Таким образом, если бы это был наш контейнер, мы бы точно создали уязвимость из общего перечня уязвимостей и рисков (Common Vulnerabilities and Exposures, CVE), позволяющую вывести из строя весь кластер.

Если хотите ощутить последствия на себе, можете действительно остановить процесс `bash` командой `kill`. Это приведет к тому, что строка с приглашением `bash-5.0#` будет напечатана на последнем издыхании вашим ничего не подозревающим процессом `chroot0.sh`. Как видите, другие процессы могут не только видеть процесс `chroot0`, но и остановить его. И здесь в игру вступает команда `unshare`.

Помните, как мы, «оглядевшись», увидели несколько модулей Pod с большими номерами PID? Это говорит о том, что наш процесс может получить доступ к каталогу `/proc` и увидеть происходящее вокруг него. Одной из первых проблем, которые вам, возможно, придется решить при создании рабочей среды контейнеризации, является *изоляция*. Если выполнить `ps -ax` из этого процесса, сразу станет понятна важность изоляции; контейнер, имеющий полный доступ к хосту, сможет вывести его из строя, например, остановив процесс `kubelet` или удалив критически важные для системы файлы.

Однако с помощью команды `unshare` мы можем создать `chroot`-окружение для запуска Bash в изолированном терминале с по-настоящему ограниченным пространством процесса. То есть на этот раз остановить `kubelet` не получится. Следующий пример демонстрирует использование команды `unshare` для такой изоляции:

```
# Обратите внимание: эта команда завершится ошибкой, если
# вы уже отмонтировали этот каталог
```

```

root@kcp:/# unshare -p -f
--mount-proc=/home/namespace/box/proc
chroot /home/namespace/box /bin/bash

```

Запустит новый сеанс командной оболочки в chroot-окружении

```

bash-5.0# ps -ax
PID TTY STAT TIME COMMAND
1 ? S 0:00 /bin/bash
2 ? R+ 0:00 ps -ax

```

Проверка доступности для наблюдения других процессов; похоже, видно только то, что выполняется в kind

Посмотрите-ка! На этот раз наш процесс, который прежде получил PID 79455, работает в том же контейнере, но, будучи запущенным командой `unshare`, он полагает, что имеет PID, равный 1. Обычно PID 1 присваивается процессу, который первым запускается в ОС (systemd). Таким образом, используя `unshare` для запуска `chroot`, мы имеем:

- изолированный процесс;
- изолированную файловую систему;
- возможность изменять определенные файлы в дереве каталогов `/tmp`.

Эта конфигурация больше похожа на модуль Pod в Kubernetes. На самом деле, выполнив команду `ps -ax` в любом запущенном модуле Kubernetes, вы получите похожую таблицу процессов.

3.4.4 Создание сетевого пространства имен

Предыдущая команда изолировала наш процесс от других, но он все еще использует общую сеть. Чтобы запустить ту же программу в новой сети, можно снова использовать команду `unshare`:

```

root@kind-control-plane:/# unshare -p -n -f
--mount-proc=/home/namespace/box/proc chroot /home/namespace/box /bin/bash
bash-5.0# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default...
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default...
   link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN group default...
   link/tunnel6 :: brd ::

```

Если сравнить этот модуль Pod с настоящим, работающим в обычном кластере Kubernetes, то можно заметить одно важное отличие: отсутствие устройства `eth0`. Если запустить предыдущий модуль Pod с `BusyBox` и выполнить в нем команду `ip a` (мы проделаем это в следующем разделе), то можно увидеть активную сеть с пригодным для использования устройством `eth0`. В этом разница между контейнером, имеющим сеть (в мире Kubernetes больше известен как CNI) и процессом в `chroot`-окружении. Как отмечалось выше, процессы в `chroot`-окружении – это основа контейнеризации, Docker и в конечном счете самой платформы Kubernetes, но само по себе `chroot`-окружение бес-

полезно для запуска контейнерных приложений из-за необходимости в передаче им дополнительных изолированных средств.

3.4.5 Проверка работоспособности процесса

В качестве упражнения выполните `exit` и вернитесь в обычный терминал в нашем кластере `kind`. Заметили разницу в выводе `ip a`? Уверены, что да! На самом деле, если вы работаете в изолированной сети, программа `cURL` (которую можно скопировать так же, как `kill` и `ls`) не сможет извлекать информацию с внешних узлов, даже при том что она прекрасно работает внутри `chroot`-окружения. Причина в том, что при создании нового сетевого пространства имен мы потеряли информацию о маршрутизации и адресе IP для нашего контейнера, которая была унаследована от пространства имен `hostNetwork`. Чтобы убедиться в этом, выполните команду `curl 172.217.12.164` (это статический IP-адрес `google.com`) после:

- запуска сценария `chroot0.sh`;
- предыдущей команды `unshare`.

В обоих случаях процесс запускается в `chroot`-окружении; однако в последнем случае он получает новую сеть и пространство имен процесса. Если с пространством имен процесса все в порядке, то сетевое пространство имен выглядит пустым по сравнению с типичным реальным процессом (например, как показано в предыдущем примере, отсутствует значимая информация об IP-адресе). Давайте посмотрим, как выглядит «настоящий» сетевой стек контейнера.

Воссоздадим наш оригинальный `pod.yaml`, с помощью которого запускался контейнер `BusyBox`. На этот раз посмотрим, как определена в нем информация о сети, чтобы затем удалить ее снова. Обратите внимание, что были случаи, когда провайдеры CNI обнаруживали ошибки, возникающие при быстром перезапуске контейнеров. В таких ситуациях контейнеры могли запускаться без IP-адреса. Об этом важно помнить при отладке сетевых ошибок контейнера в промышленных окружениях. Такое, в частности, особенно характерно при перезапуске контейнеров `StatefulSet`, поддерживающих сохранение IP-адресов. Следующий код иллюстрирует, что контейнер имеет сеть (в отличие от нашего процесса в `chroot`-окружении):

```
$ kubectl delete -f pod.yaml ;
  kubectl create -f pod.yaml
$ kubectl exec -t -i core-k8s ip a
```

Создает оригинальный пример `pod.yaml`,
как было показано выше в этой главе

Запускает команду для получения
списка сетевых интерфейсов

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue 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: tunl0@NONE: <NOARP> mtu 1480 qdisc noop qlen 1
   link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop qlen 1
   link/tunnel6 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
   brd 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
5: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN>
   mtu 1500 qdisc noqueue
   link/ether 4a:9b:b2:b7:58:7c brd ff:ff:ff:ff:ff:ff
   inet 10.244.0.7/24 brd 10.244.0.255 scope global eth0
       valid_lft forever preferred_lft forever
   inet6 fe80::489b:b2ff:feb7:587c/64 scope link
       valid_lft forever preferred_lft forever

```

Здесь мы видим резкий контраст. В предыдущем примере, где мы не могли запустить даже простую команду `curl`, у нас не было устройства `eth0`, а в этом контейнере оно есть. Вы можете удалить этот файл `pod.yaml`, если хотите. И, как обычно, не чувствуйте себя уязвленными – не нужно слишком серьезно воспринимать контейнеры `BusyBox`.

Мы еще вернемся к некоторым сетевым настройкам в контексте `iptables` и IP-маршрутизации далее в этой главе. А сейчас завершим наш вступительный обзор примитивов Linux для создания контейнеров. Далее рассмотрим параметр, наиболее часто переключаемый в настройках контрольных групп в промышленном окружении для типичных приложений Kubernetes: `limits`.

3.4.6 Ограничение потребления процессора с помощью `cgroups`

Контрольные группы (`control groups`, `cgroups`) – это те самые рычажки и рукоятки настройки, которые мы так любим. Они позволяют выделять больше или меньше процессорного времени и памяти приложениям, работающим в наших кластерах. Если вы используете Kubernetes на работе, то, вероятно, уже пробовали использовать эти рычажки и рукоятки. Мы можем легко изменить объем процессорного времени, доступный нашему предыдущему контейнеру `BusyBox`:

```

$ cat << EOF > greedy-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s-greedy
spec:
  containers:
  - name: any-old-name-will-do
    image: docker.io/busybox:latest
    command: ['sleep', '10000']
    resources:
      limits:
        memory: "200Mi"

```

Сообщает платформе Kubernetes, что та должна создать контрольную группу для ограничения доступного процессорного времени

```
requests:
  memory: "100Mi"
EOF
```

3.4.7 Создание раздела *resources*

Мы вручную пройдем этапы, которые проходит kubelet, когда определяются ограничения контрольных групп. Обратите внимание, что фактический способ определения этого параметра можно настроить в любом дистрибутиве Kubernetes с помощью флага `--cgroup-driver`. (*Драйверы контрольных групп* – это архитектурные компоненты в Linux, которые используются для выделения ресурсов. Обычно в роли драйвера в Linux используется `systemd`.) Тем не менее основные логические этапы запуска контейнера в Kubernetes, включая создание изолированного окружения для процесса, по сути, одинаковы, даже при отклонении от традиционной архитектуры `containerd/Linux`. На самом деле, в kubelet для Windows тот же самый раздел `resources` обслуживается с использованием совершенно другого набора деталей реализации.

Чтобы определить ограничения контрольной группы, нужно выполнить следующие действия:

- создать PID (мы это уже сделали). В Kubernetes это называется песочницей модуля Pod;
- передать ограничения для этого PID в операционную систему.

Для этого, во-первых, внутри окружения, запущенного сценарием `chroot0`, нужно получить его PID, выполнив команду `echo $$`. Запишите полученное число. В нашем случае это было число 79455. Затем пройдем ряд шагов, чтобы ограничить этот конкретный процесс и позволить ему использовать строго ограниченный объем памяти. При этом мы сможем оценить, сколько памяти требуется для выполнения команды `ls`:

```
root@kind-control-plane:/# mkdir
/sys/fs/cgroup/memory/chroot0
root@kind-control-plane:/# echo "10" >
/sys/fs/cgroup/memory/chroot0/
memory.limit_in_bytes
root@kind-control-plane:/# echo "0" >
/sys/fs/cgroup/memory/chroot0/memory.swappiness
root@kind-control-plane:/# echo 79455 >
/sys/fs/cgroup/memory/chroot0/tasks
```

Создаст контрольную группу

Выделит контейнеру только 10 байт памяти, что сделает его неспособным выполнять основную работу

Гарантирует, что контейнер не сможет выделить место в пространстве подкачки (Kubernetes почти всегда работает без поддержки подкачки)

Сообщает ОС, что этот процесс (chroot0 с PID 79455) управляется этой контрольной группой

Обратите внимание, что в этом примере создание каталога `/chroot0` вызывает действие ОС по созданию полной контрольной группы,

ограничивающей память, процессорное время и т. д. Теперь вернемся к терминалу, где был запущен сценарий `chroot0.sh`. Попытка выполнить простую команду, такую как `ls`, терпит неудачу. В зависимости от ОС можете получить другой, но не менее мрачный ответ, как показано ниже; в любом случае эта команда должна завершиться ошибкой:

```
bash-5.0# ls
bash: fork: Cannot allocate memory
```

Вот и все! Теперь вы создали свой собственный процесс, изолированный от других файлов, имеющий ограниченный объем памяти и действующий в изолированном пространстве, где ему кажется, что он единственный процесс во всем мире. Это естественное состояние любого модуля Pod в кластере Kubernetes. С этого момента мы начнем изучать, как Kubernetes расширяет эту базовую функциональность и создает сложную, динамичную и отказоустойчивую распределенную систему.

3.5 Использование модуля Pod в реальном мире

На рис. 3.4 изображен реальный контейнер. Мы уже многое узнали в этой главе, но имейте в виду, что обычному микросервису может потребоваться взаимодействовать с большим количеством других сервисов, что часто означает подключение новых сертификатов. Кроме того, обнаружение других сервисов с использованием внутреннего DNS всегда сложнее, и в этом заключается огромное преимущество модели масштабируемого управления микросервисами в Kubernetes. У нас не было возможности добавить поддержку запросов API для внутренних сервисов, и мы пока не исследовали автоматическое внедрение учетных данных для безопасного взаимодействия с такими сервисами, поэтому мы не можем сказать, что реализованную нами контрольную группу и `chroot`-окружение можно использовать в реальном мире.

Обратите внимание, что контейнер на рис. 3.4 способен взаимодействовать с другими контейнерами в кластере. Для этого ему нужен IP-адрес. Однако, поскольку наш модуль Pod был создан без настроенного сетевого пространства имен и отдельного IP-адреса, он не сможет устанавливать какие-либо прямые TCP-соединения с нижестоящими службами.

Теперь мы чуть подробнее обсудим, что значит иметь сетевой контейнер. Напомним, что ранее, исследуя этот аспект нашего процесса Bash, мы видели, что он имеет только один IP-адрес, не имеющий отношения к нашему контейнеру. Это означает, что нет никакой возможности направить входящий трафик какому-либо сервису, запущенному в этом процессе. (Обратите внимание, что мы не пред-

лагаем запускать веб-сервер в командной оболочке Bash, мы использовали ее лишь как метафору произвольной программы, в том числе и сервиса TCP.)

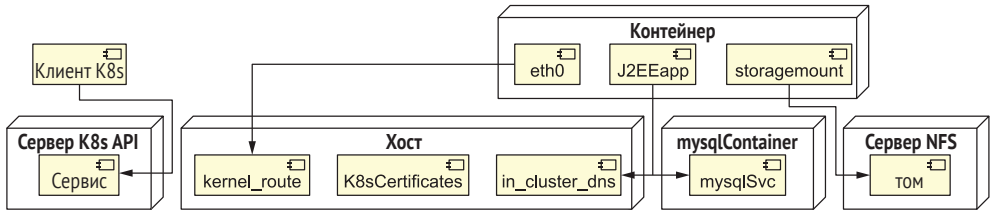


Рис. 3.4 Пример реального контейнера

Прежде чем перейти к освещению этой части головоломки, давайте рассмотрим некоторые основные сетевые примитивы Linux, чтобы подготовить почву для дальнейшего знакомства с различными аспектами сети в Kubernetes.

3.5.1 Проблема сети

Любому контейнеру Kubernetes могут потребоваться:

- маршрутизация трафика внутри кластера для сетевых взаимодействий между модулями Pod;
- маршрутизация трафика наружу для доступа к другим модулям Pod в интернете;
- балансировка трафика между конечными точками за сервисом со статическим IP-адресом.

Для поддержки всего этого необходимо, чтобы метаданные о модулях Pod публиковались где-то в Kubernetes (эту задачу решает сервер API), а также постоянно наблюдать за их состоянием (эту задачу решает kubelet) и своевременно обновлять эту информацию. Таким образом, модули Pod имеют не только команду запуска контейнера и образ Docker, но также *метки* и четко определенные *спецификации*, определяющие порядок публикации их состояния, благодаря чему их можно повторно создавать на лету вместе с набором возможностей, предоставляемых агентом kubelet. Это гарантирует неизменную актуальность IP-адресов и правил DNS. Метки можно видеть в схеме модулей, а под спецификациями подразумеваются четко определенные состояния модулей, логика повторного запуска и гарантии доступности IP-адресов в кластере.

Обратите внимание, что для изучения этих аспектов мы вновь используем среду Linux. Вы можете перестроить свой кластер kind, если хотите или если вы что-то нарушили в ходе экспериментов в предыдущих разделах. Для этого выполните команду `kind delete cluster --name=kind`, а затем `kind create cluster`.

3.5.2 Как kube-proxy реализует сервисы Kubernetes с помощью iptables

Сервисы Kubernetes определяют контракт API, в котором говорится: «При обращении к некоторому IP-адресу ваш трафик автоматически пересылается одной из многих возможных конечных точек». Таким образом, контракты являются основой Kubernetes, когда дело доходит до развертывания микросервисов. В большинстве кластеров эти сетевые правила полностью реализуются прокси-сервером kube-proxy, который чаще всего настроен на использование iptables для организации маршрутизации сетевых пакетов.

Программа iptables добавляет в ядро правила, которые затем используются для обработки сетевого трафика, и это наиболее распространенный способ реализации сервисов Kubernetes и маршрутизации трафика. Обратите внимание, что iptables не требуется для работы базовой сети модуля Pod (эту функцию выполняет CNI; однако почти любой реальный кластер Kubernetes обслуживает конечных пользователей через сервисы). Таким образом, iptables и соответствующие правила являются одним из самых фундаментальных примитивов, необходимых для рассуждений о сети Kubernetes. В традиционном окружении (вне Kubernetes) каждое правило iptables добавляется в сетевой стек ядра с использованием синтаксиса `-A . . .`, как показано ниже:

```
iptables -A INPUT -s 10.1.2.3 -j DROP
```

Эта команда говорит, что любой трафик, поступающий с IP-адреса 10.1.2.3, должен отбрасываться. Однако модулю Pod нужно нечто большее, чем набор правил брандмауэра. Ему нужны по крайней мере:

- возможность принимать трафик в конечной точке сервиса;
- возможность отправлять трафик во внешний мир из своей конечной точки;
- возможность отслеживать текущие соединения TCP (в Linux это делается с помощью модуля conntrack, входящего в состав ядра Linux).

Давайте посмотрим, как реализуются сетевые правила в действующем сервисе (работающем под управлением kind). Мы больше не будем использовать наш модуль Pod, потому что для его привязки к IP-адресу нам действительно необходима действующая, маршрутизируемая сеть, определяемая программно. Мы постараемся обеспечить максимальную простоту.

Итак, давайте посмотрим, какую информацию о нашей сети выводит команда `iptables-save | grep hostnames`.

3.5.3 Использование модуля kube-dns

Модуль Pod kube-dns – отличный пример для изучения, потому что является типичным модулем Pod, запускаемым в приложении Kubernetes. Модуль Pod kube-dns:

- работает в любом кластере Kubernetes;
- не имеет особых привилегий и использует обычную сеть модулей Pod, а не сеть хоста;
- отправляет трафик в порт 53, широко известный как стандартный порт DNS;
- уже работает в вашем кластере по умолчанию.

Модуль Pod, созданный нами ранее, не имел доступа к интернету, а также не мог получать трафик. Когда мы запускали команду `ip a`, то видели, что модуль Pod не имел собственного IP-адреса. В Kubernetes провайдер CNI предоставляет уникальный IP-адрес и правила маршрутизации для доступа к этому адресу. Мы можем исследовать эти маршруты с помощью команды `ip route`:

```
root@kind-control-plane:/# ip route
default via 172.18.0.1 dev eth0
10.244.0.2 dev vethfc5287fa scope host
10.244.0.3 dev veth31aba882 scope host
10.244.0.4 dev veth1e578a9a scope host
172.18.0.0/16 dev eth0 proto kernel scope link src 172.18.0.2
```

Согласно этому выводу мы имеем predefined IP-маршруты для отправки трафика в определенные устройства veth. Эти устройства создаются сетевым плагином. Как сервисы Kubernetes направляют к ним трафик? Чтобы выяснить это, посмотрим вывод программы `iptables`. Если выполнить команду `iptables-save`, то в ее выводе можно найти адреса `10.244.0.*` (конкретные адреса будут различаться в зависимости от особенностей вашего кластера и от того, сколько в нем может быть модулей Pod) и увидеть, что существуют правила для исходящего трафика, позволяющие создавать исходящие TCP-соединения.

МАРШРУТИЗАЦИЯ ТРАФИКА В МОДУЛИ DNS

С ПРАВИЛАМИ ОБСЛУЖИВАНИЯ

Внутренний трафик передается модулям DNS, согласно следующим правилам, использующим параметр `-j`, который сообщает ядру: «если что-то пытается получить доступ к правилу KUBE-SVC-ERIFX, то передать трафик правилу KUBE-SEP-IT2Z». Параметр `-j` в правиле `iptables` означает *jump* (переход – «переход к другому правилу»). Правило перехода пересылает сетевой трафик конечной точке сервиса (Pod), как показано в следующем примере:

```
-A KUBE-SVC-ERIFXISQEP7F70F4 -m comment --comment
    "kube-system/kube-dns:dns-tcp"
    -m statistic
    --mode random
    --probability 0.500000000000
    -j KUBE-SEP-IT2ZTR26T04XFPT0
```

ОПРЕДЕЛЕНИЕ ПРАВИЛ ДЛЯ ОТДЕЛЬНЫХ МОДУЛЕЙ Pod С ПРАВИЛАМИ КОНЕЧНЫХ ТОЧЕК

Трафик, полученный от сервиса, маршрутизируется с использованием следующих правил KUBE-SEP. Эти модули Pod получают доступ к внешнему интернету или принимают трафик извне. Например:

```
-A KUBE-SEP-IT2Z.. -s 10.244.0.2/32
  -m comment --comment "kube-system/kube-dns:dns-tcp"
  -j KUBE-MARK-MASQ

-A KUBE-SEP-IT2Z.. -p tcp
  -m comment --comment "kube-system/kube-dns:dns-tcp"
  -m tcp -j DNAT --to-destination 10.244.0.2:53
```

Как следует из примера, конечным портом для любого трафика, направляемого на эти IP-адреса, является порт 53. Это конечная точка, посредством которой модуль Pod kube-dns обслуживает свой трафик (IP-адрес модуля Pod CoreDNS, обрабатывающего этот трафик). Если один из этих модулей выйдет из строя, будет выполнено согласование конкретного правила для KUBE-SEP-IT2Z с сетевым прокси-сервером kube-проху, чтобы трафик направлялся только исправным экземплярам наших модулей DNS. Обратите внимание, что kube-dns – это имя сервиса, а CoreDNS – это модуль Pod, реализующий конечную точку сервиса kube-dns.

Главная цель сетевого прокси состоит в постоянном управлении наборами этих простых правил, чтобы любой узел в кластере Kubernetes мог передавать трафик сервисам Kubernetes, поэтому мы часто называем просто сетевым прокси-сервером Kubernetes или прокси-сервисом Kubernetes.

3.5.4 Другие проблемы

Хранение, планирование и повторный запуск – ничего из этого мы еще не обсуждали. Все эти проблемы характерны для любого корпоративного приложения. Например, в традиционном центре обработки данных может потребоваться произвести перенос базы данных с одного сервера на другой, а затем перенастроить серверы приложений, подключающиеся к этой базе данных, в соответствии с новой топологией центра обработки данных. В Kubernetes тоже нужно учитывать эти аспекты.

ХРАНИЕНИЕ

Помимо сети, нашему модулю Pod также может потребоваться доступ к различным типам хранилищ. Например, представьте, что у нас имеется большое сетевое хранилище (Network Attached Storage, NAS), которое должны использовать все контейнеры, и нам нужно перио-

дически менять способ подключения к этому хранилищу NAS. В предыдущем примере это означало бы изменение команд и монтирование томов по одному. Делать это для сотен или тысяч процессов без дополнительных инструментов и средств автоматизации было бы нецелесообразно по очевидным причинам. Однако даже с такими инструментами нужен какой-то способ, с помощью которого можно определить типы хранилищ и фиксировать в журнале неудачные попытки монтирования этих томов хранилища. Такой способ в Kubernetes предоставляют объекты StorageClass, PersistentVolume и PersistentVolumeClaim.

ПЛАНИРОВАНИЕ

В предыдущей главе мы отметили, что планирование модулей Pod – сложный процесс. Помните, как мы настраивали контрольные группы cgroups? Представьте, что может произойти, если выделять контейнерам чересчур много памяти или запускать их в окружении с ограниченным объемом памяти, недостаточным, чтобы удовлетворить его запросы. В любом из этих случаев мы можем вызвать остановку целого узла в нашем кластере Kubernetes. Наличие планировщика, достаточно умного, чтобы поместить модуль Pod в место, где иерархия контрольных групп соответствует требованиям модуля к ресурсам, – еще одна важнейшая функция, которую берет на себя Kubernetes.

Планирование – это общая проблема в информатике, поэтому следует отметить, что существуют альтернативные инструменты планирования, такие как Nomad (<https://www.nomadproject.io/>), решающие задачу в манере независимости от провайдера, свойственной Kubernetes. Планировщик Kubernetes специализируется на использовании простых и предсказуемых критериев выбора узлов для запуска модулей Pod и опирается на такие параметры, как сходство, требования к процессору и памяти, доступность хранилища, топология центра обработки данных и т. д.

ОБНОВЛЕНИЕ И ПОВТОРНЫЙ ЗАПУСК

Команды Bash, которые мы запускали для создания модуля Pod, не смогут работать должным образом, если PID модуля будет постоянно меняться или если мы вообще забудем его записать. Как вы помните, нам пришлось записать PID, чтобы потом использовать его в некоторых операциях. Если потребуется запустить более сложное приложение, чем bash или ls, может обнаружиться, что необходимо удалить данные из папки, добавить новые данные в папку, а затем перезапустить сценарий. И снова этот процесс практически не масштабируется с помощью сценариев командной оболочки из-за большого объема операций управления каталогами, процессами и монтированием томов, которые должны выполняться одновременно для нескольких приложений.

Управление процессами и/или контрольными группами, связанными с модулем Pod, который может перестать работать, является важной частью организации масштабных контейнерных рабочих нагрузок, особенно в контексте микросервисов, которые должны быть переносимыми и эфемерными. Модель данных приложений в Kubernetes, о которой чаще всего думают с точки зрения объектов развертывания Deployment, приложений с состоянием StatefulSet, заданий Job и наборов демонов DaemonSet, поддерживает возможность надежного обновления.

Итоги

- Платформа Kubernetes – это объединение различных примитивов Linux.
- Создать конструкцию, похожую на модуль Pod, можно с помощью chroot в любом дистрибутиве Linux.
- Механизмы хранения, планирования и сетевых взаимодействий должны поддерживать самые разные возможности управления, чтобы обеспечить надежную работу наших модулей Pod в промышленном окружении.
- iptables – это примитив Linux, который можно использовать для гибкого перенаправления трафика или создания брандмауэров.
- Сервисы Kubernetes реализуются прокси-сервером kube-proxy, который обычно опирается на работу iptables.

Использование контрольных групп для управления процессами в модулях Pod

В этой главе:

- основы контрольных групп;
- идентификация процессов Kubernetes;
- создание контрольных групп и управление ими;
- исследование иерархии контрольных групп с командами Linux;
- сравнение cgroup v2 и cgroup v1;
- установка Prometheus и взгляд на потребление ресурсов модулями Pod.

Предыдущая глава была перегружена деталями, и вы, возможно, сочли ее слишком теоретической. В конце концов, в настоящее время нет необходимости создавать свои модули Pod с нуля (если вы не работаете в Facebook). Но не волнуйтесь, с этого момента мы перейдем на более высокий уровень в стеке.

В этой главе мы немного углубимся в контрольные группы cgroups: управляющие структуры, которые изолируют ресурсы друг от друга в ядре. В предыдущей главе мы на самом деле реализовали простую границу модуля в виде контрольной группы, которую создали сами. На этот раз создадим «настоящий» модуль Pod на платформе Kubernetes и посмотрим, как ядро управляет контрольной группой этого модуля. Попутно мы рассмотрим несколько поучительных примеров,

объясняющих существование контрольных групп. И в заключение исследуем Prometheus, агрегатор метрик, ставший фактическим стандартом мониторинга в облачном окружении.

Самое главное, о чем следует помнить, читая эту главу: контрольные группы и пространства имен Linux не являются черной магией. В действительности это простые реестры, поддерживаемые ядром, связывающие процессы с IP-адресами, объемами выделяемой памяти и т. д. Поскольку задача ядра – предоставлять эти ресурсы программам, становится совершенно очевидно, почему эти структуры тоже управляются самим ядром.

4.1 Модули Pod простаивают до завершения подготовительных операций

В предыдущей главе мы кратко коснулись происходящего при запуске модуля Pod. Давайте вернемся к этому сценарию и посмотрим, что *на самом деле* должен сделать kubelet, чтобы создать настоящий модуль Pod (рис. 4.1). Обратите внимание, что наше приложение бездействует, пока приостановленный контейнер не будет добавлен в наше пространство имен. После этого приложение наконец запустится.

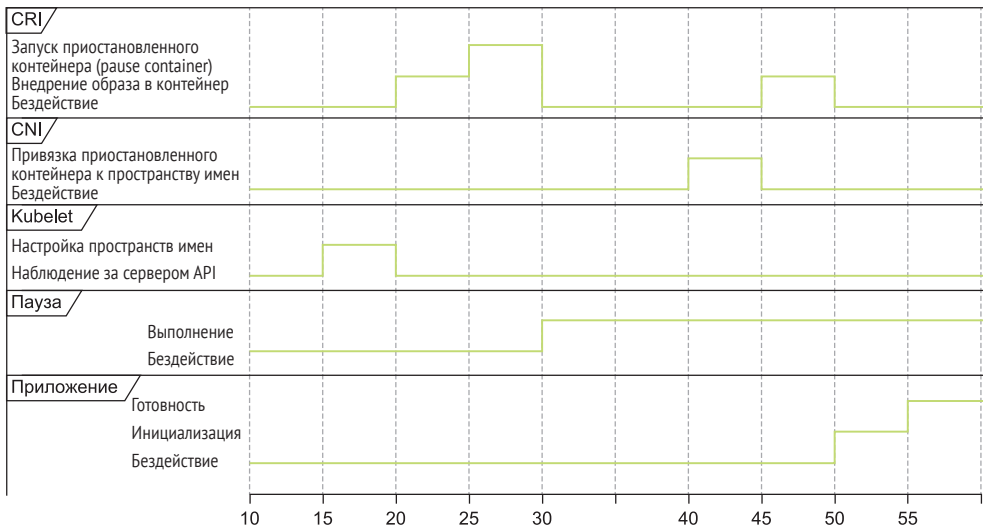


Рис. 4.1 Процессы, вовлеченные в запуск контейнера

На рис. 4.1 показано, как меняется состояние различных частей kubelet во ходе создания контейнера. Каждый агент kubelet имеет установленный интерфейс среды выполнения (Container Runtime Interface, CRI), отвечающий за запуск контейнеров, и сетевой интерфейс (Container Network Interface, CNI), отвечающий за предоставле-

ние контейнерам IP-адресов, и будет запускать один или несколько *приостановленных контейнеров* (заготовок, в которых kubelet создаст пространства имен и контрольные группы cgroups для запуска контейнеров). Чтобы подготовить приложение к балансировке нагрузки со стороны Kubernetes, необходимо запустить несколько эфемерных процессов и обеспечить высокий уровень координации их действий:

- если бы CNI запускался до приостановленного контейнера с CNI, то он не смог бы иметь сеть;
- в отсутствие доступных ресурсов kubelet не сможет завершить подготовку модуля Pod к запуску, и ничего не произойдет;
- перед запуском каждого модуля Pod создается приостановленный контейнер, служащий заготовкой для процессов в модуле Pod.

Мы решили проиллюстрировать этот замысловатый танец, чтобы подчеркнуть тот факт, что программам нужны ресурсы, а ресурсы конечны: управление ресурсами – сложный, упорядоченный процесс. Чем больше программ мы запускаем, тем сложнее выглядит переплетение запросов на ресурсы. Рассмотрим несколько примеров. Каждая из следующих программ имеет разные требования к объему процессорного времени, памяти и хранилища:

- *расчет числа пи* – программе расчета числа пи требуется доступ к выделенному ядру процессора, чтобы иметь возможность непрерывно выполнять вычисления;
- *кеширование содержимого «Википедии» для быстрого поиска* – программе кеширования «Википедии» в хеш-таблицу, в отличие от программы расчета числа пи, требуется меньше процессорного времени, но может потребовать около 100 Гбайт памяти;
- *резервное копирование базы данных объемом 1 Тбайт* – программа резервного копирования базы данных в холодное хранилище практически не требует памяти, ей достаточно небольшого объема процессорного времени, но необходимо хранилище огромной емкости, роль которого может играть медленный вращающийся диск.

Если представить, что у нас есть один компьютер с двухъядерным процессором, 101 Гбайт памяти и хранилищем с емкостью 1,1 Тбайт, то теоретически можно запустить каждую программу, удовлетворив требования к процессорному времени, памяти и емкости хранилища. Однако:

- программа расчета числа пи, если написана неправильно (например, сохраняет промежуточные результаты на диск), может в конечном итоге занять на диске пространство, необходимое программе резервного копирования;
- программа кеширования «Википедии», если она написана неправильно (например, функция хеширования слишком интенсивно использует процессор), может помешать программе расчета числа пи быстро выполнять математические вычисления;

- программа резервного копирования базы данных, если написана неправильно (например, журналирует слишком много информации), может помешать программе расчета числа пи выполнить свою работу, потратив все процессорное время.

Вместо запуска всех процессов с полным доступом ко всем (ограниченным) ресурсам системы можно сделать следующее (если предположить, что у нас есть возможность разделить ресурсы процессора, памяти и дискового пространства):

- запустить процесс расчета числа пи, выделив ему 1 ядро и 1 Кбайт памяти;
- запустить кеширование «Википедии», выделив половину ядра и 99 Гбайт памяти;
- запустить резервное копирование базы данных, выделив 1 Гбайт памяти и оставшиеся половину ядра процессора и том хранилища, недоступные для других приложений.

Чтобы это можно было сделать предсказуемым образом для программ, управляемых нашей ОС, контрольные группы позволяют определить иерархически разделенные ячейки для памяти, процессора и других ресурсов. Все потоки, созданные программой, используют один и тот же пул ресурсов, изначально предоставленный родительскому процессу. Другими словами, никто не может распоряжаться чужим пулом.

Это само по себе является аргументом в пользу контрольных групп для модулей Pod. В кластере Kubernetes можно запустить 100 программ на одном компьютере, большая часть из которых имеет низкий приоритет или полностью простаивает в какие-то моменты времени. Если эти программы зарезервируют большие объемы памяти, они сделают стоимость работы такого кластера неоправданно высокой. Создание новых узлов для предоставления памяти «голодающим» процессам приведет к административным издержкам и затратам на инфраструктуру, которые со временем будут только увеличиваться. Поскольку перспективность контейнеров (увеличение использования центров обработки данных) в значительной степени основана на возможности расходовать меньше ресурсов для каждого сервиса, в основе запуска приложений в виде микросервисов лежит тщательное использование контрольных групп.

4.2 Процессы и потоки в Linux

Любой процесс в Linux может создать один или несколько потоков выполнения. *Поток выполнения* – это абстракция, которую программы могут использовать для создания новых процессов, имеющих общую память. Для примера давайте посмотрим с помощью команды `ps -T`, сколько независимых потоков планирования используется в Kubernetes:


```

root@kind-control-plane:/# ps -ax | grep scheduler
631 ? Ssl 60:14 kube-scheduler
--authentication-kubeconfig=/etc/kubernetes/...

```

Получить PID модуля Pod планировщика в Kubernetes

```

root@kind-control-plane:/# ps -T 631

```

Вывести список потоков выполнения в модуле Pod

```

root@kind-control-plane:/# ps -T 631
  PID  SPID  TTY      STAT   TIME COMMAND
  631    631 ?        Ssl    4:40 kube-scheduler --authentication-kube..
  631    672 ?        Ssl   12:08 kube-scheduler --authentication-kube..
  631    673 ?        Ssl    4:57 kube-scheduler --authentication-kube..
  631    674 ?        Ssl    4:31 kube-scheduler --authentication-kube..
  631    675 ?        Ssl    0:00 kube-scheduler --authentication-kube..

```

Эта команда выводит список потоков выполнения планировщика, использующих общую память. Потоки имеют свои идентификаторы подпроцессов, и для ядра Linux все они являются обычными процессами. И все же у них есть одна общая черта: родитель. Связь родитель/потомок можно исследовать с помощью команды `pstree`:

```

/# pstree -t -c | grep sched
|-containerd-sh-+-kube-scheduler-+-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}
|                               |-{kube-}

```

Родитель планировщика – containerd-shim, следовательно, он работает в контейнере

Все потоки планировщика имеют одного и того же родителя – главный поток самого планировщика

containerd и Docker

Мы не тратили время на сравнение containerd и Docker, но настал момент, и мы должны отметить, что кластеры kind не используют Docker в роли среды выполнения контейнеров. Они используют Docker для создания узлов, а затем на каждом узле используется среда выполнения containerd. Современные кластеры Kubernetes редко используют Docker как среду выполнения контейнеров в Linux. Тому есть несколько причин. Docker был отличной отправной точкой для запуска Kubernetes разработчиками, но центрам обработки данных нужны более легкие решения, глубже интегрированные в ОС. В большинстве кластеров на самом низком уровне используется среда выполнения контейнеров `glibc`, которая

вызывается из containerd, CRI-O или другого выполняемого файла командной строки более высокого уровня, установленного на узлах. Это заставляет systemd быть родителем ваших контейнеров, а не демоном Docker.

Одна из особенностей, делающих контейнеры такими популярными, особенно в Linux, – они не создают искусственной границы между программой и хостом, позволяя планировать программы и управлять ими с использованием более простых и легковесных механизмов, чем виртуальные машины.

4.2.1 Процессы *systemd* и *init*

Теперь, кратко познакомившись с иерархией процессов, сделаем шаг назад и зададимся вопросом: что такое процесс *на самом деле*? Вернувшись в наш надежный кластер kind, мы выполнили следующую команду, чтобы узнать, с чего все началось (посмотрите на первые несколько строк в журнале состояния systemd). Помните, что наш узел kind (который мы запустили, чтобы получить все это) на самом деле всего лишь контейнер Docker; в противном случае вывод следующей команды может вас немного напугать:

```
root@kind-control-plane:/# systemctl status | head
kind-control-plane
State: running
Jobs: 0 queued
Failed: 0 units
Since: Sat 2020-06-06 17:20:42 UTC; 1 weeks 1 days
CGroup: /docker/b7a49b4281234b317eab...9
├─ init.scope
│ └─ 1 /sbin/init
└─ system.slice
   └─ containerd.service
      └─ 126 /usr/local/bin/containerd
```

Эта единственная контрольная группа является родительской для нашего узла kind

Сервис containerd является потомком контрольной группы Docker

На обычном компьютере с Linux можно получить увидеть следующий, более показательный вывод:

```
State: running
Jobs: 0 queued
Failed: 0 units
Since: Thu 2020-04-02 03:26:27 UTC; 2 months 12 days
cgroup: /
├─ docker
│ └─ ae17db938d5f5745cf343e79b8301be2ef7
│   └─ init.scope
│     └─ 20431 /sbin/init
│       └─ system.slice
```

А в `system.slice` вы увидите:

```
└─ containerd.service
└─ 3067 /usr/local/bin/containerd-shim-runc-v2
    -namespace k8s.io -id db70803e6522052e
└─ 3135 /pause
```

На стандартном компьютере с Linux или на узле кластера `kind` корнем всех контрольных групп является `/` – родительская контрольная группа для всех процессов в системе, которая создается при запуске. Сам процесс `Docker` является потомком этой контрольной группы, и если запустить кластер `kind`, то узлы `kind` станут потомками этого процесса `Docker`. Если запустить обычный кластер `Kubernetes`, то, скорее всего, вместо контрольной группы `Docker` мы увидим, что `containerd` является потомком корневого процесса `systemd`. Если у вас под рукой есть узел `Kubernetes`, к которому можно подключиться по `ssh`, то его проверка может стать хорошим дополнительным упражнением.

Если углубиться вниз по такому дереву достаточно далеко, то можно найти процессы, запущенные контейнерами в нашей ОС. Обратите внимание, что идентификаторы процессов (PID), полученные при исследовании на хосте, такие как 3135 в предыдущем фрагменте, на самом деле являются довольно большими числами. Это связано с тем, что PID процесса *вне контейнера* не совпадает с PID процесса *внутри* контейнера. Причину мы объяснили в первой главе, когда демонстрировали применение команды `unshare` для разделения пространств имен наших процессов. Это означает, что процессы, запущенные контейнерами, не могут видеть, идентифицировать или уничтожать процессы, запущенные в других контейнерах. Это важная функция безопасности любого развертывания программного обеспечения.

Вам также может быть интересно узнать назначение приостановленных процессов (`pause processes`). Для каждой из программ `containerd-shim` имеется соответствующая ей приостановленная программа, которая используется как заготовка для создания сетевого пространства имен. Приостановленные контейнеры также помогают очищать процессы и служат заготовками для интерфейса CRI, помогающими осуществлять базовые операции управления процессами и избежать появления процессов-зомби.

4.2.2 Контрольные группы для процессов

Теперь мы получили довольно хорошее представление о том, чем занимается планировщик: он порождает несколько потомков и часто сам создается платформой `Kubernetes`, потому что является потомком `containerd`, – среды выполнения контейнеров, – которую `Kubernetes` использует в `kind`. В качестве эксперимента можете попробовать завершить процесс `containerd` и понаблюдать, как планировщик и его потоки возвращаются к жизни. Это делается самим агентом `kubelet`,

у которого есть каталог /manifests. В этом каталоге определяются процессы, которые всегда должны выполняться, даже до того, как сервер API сможет планировать контейнеры. Собственно, так Kubernetes и устанавливает себя через kubelet. Жизненный цикл установки Kubernetes, который реализует kubeadm (самый распространенный в настоящее время инструмент установки), выглядит примерно так:

- в kubelet есть каталог manifests, в который включены сервер API, планировщик и диспетчер контроллеров;
- system запускает kubelet;
- kubelet сообщает процессу containerd (или любой другой среде выполнения контейнеров) о необходимости запустить все процессы, перечисленные в каталоге manifests;
- сразу после запуска сервера API агент kubelet подключается к нему и запускает все контейнеры, которые запросит сервер API.

Зеркальные модули Pod подкрадываются к серверу API

В kubelet есть секретное оружие: каталог /etc/kubernetes/manifests. Агент kubelet постоянно сканирует этот каталог, и, когда в нем появляются новые модули Pod, он немедленно создает и запускает их. Поскольку запуск этих модулей Pod производится в обход сервера Kubernetes API, им необходимо зеркалировать себя, чтобы сервер API узнал об их существовании. По этой причине модули Pod, созданные за пределами плоскости управления Kubernetes, называются *зеркальными модулями Pod* (mirror Pod).

Зеркальные модули, как и любые другие, можно увидеть, выполнив команду `kubectl get pods -A`, но они создаются и управляются агентом kubelet на независимой основе. Это позволяет kubelet в одиночку запустить целый кластер Kubernetes, работающий внутри модулей Pod. Довольно неожиданно!

Вы можете спросить: «Какое отношение все это имеет к контрольным группам?» Как оказывается, планировщик, показанный выше, на самом деле идентифицируется как зеркальный модуль Pod, и контрольные группы, за которыми он закреплен, именуются с использованием этого идентификатора. Причина использования этой особой идентификации заключается в том, что изначально сервер API ничего не знает о зеркальном модуле Pod, потому что он был создан агентом kubelet. Давайте добавим немного конкретики: рассмотрим следующее определение объекта Pod и найдем его идентификатор:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  annotations:
```

```
    kubernetes.io/config.hash: 155707e0c1919c0ad1
```

```
    kubernetes.io/config.mirror: 155707e0c19147c8
```

Идентификатор зеркального модуля Pod планировщика

```
kubernetes.io/config.seen: 2020-06-06T17:21:0
kubernetes.io/config.source: file
creationTimestamp: 2020-06-06T17:21:06Z
labels:
```

Идентификатор зеркального модуля Pod планировщика можно использовать для поиска соответствующих контрольных групп. Получить доступ к этим модулям, чтобы просмотреть их содержимое, можно, отправив команду `edit` или `get action` модулю Pod плоскости управления (например, `kubectl edit Pod -n kube-system kube-apiserver-calico-control-plane`). Теперь давайте посмотрим, сможем ли мы найти контрольные группы, ограничивающие процессор, выполнив:

```
$ cat /proc/631/cgroup
```

В этой команде мы использовали найденный ранее PID, чтобы узнать, какие контрольные группы определены для планировщика. Результат получился довольно пугающим (как показано ниже). Не беспокойтесь о папке *burstable*; мы объясним идею *burstable* – класса качества обслуживания (Quality of Service, QoS) – после знакомства с некоторыми внутренними особенностями kubelet. А пока отметим, что *модули Pod с классом burstable*, как правило, не имеют жестких ограничений на использование ресурсов. Планировщик – это пример модуля Pod, который может иметь всплески потребления процессора (например, когда необходимо быстро запланировать 10 или 20 модулей Pod для выполнения на узле). Каждый элемент списка имеет чрезвычайно длинный идентификатор контрольной группы и идентификатора модуля Pod, например:

```
13:name=systemd:/docker/b7a49b4281234b31
➔ b9/kubepods/burstable/pod155707e0c19147c../391fbfc..
➔ a08fc16ee8c7e45336ef2e861ebef3f7261d
```

Как видите, ядро отслеживает все эти процессы в каталоге `/proc`, и мы можем продолжать копать, чтобы увидеть, какие ресурсы получает каждый процесс. Список контрольных групп для процесса 631 можно вывести, передав команде `cat` файл `cgroup`, как показано ниже. Обратите внимание, что мы сократили очень длинные идентификаторы для удобства чтения:

```
root@kind-control-plane:/# cat /proc/631/cgroup
```

```
13:name=systemd:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
12:pids:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
11:hugetlb:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
10:net_prio:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
9:perf_event:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
8:net_cls:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
7:freezer:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
```

```
6:devices:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
5:memory:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
4:blkio:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
3:cpuacct:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
2:cpu:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
1:cpuset:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
0::/docker/b7a49b42.../system.slice/containerd.service
```

Мы поочередно заглянем внутрь этих папок. Не беспокойтесь слишком о папке `docker`. Мы экспериментируем в кластере `kind`, поэтому папка `docker` является родительской для всего и вся. Но обратите внимание, что на самом деле все наши контейнеры работают в `containerd`:

- `docker` – контрольная группа `cgroup` для демона Docker, работающего на нашем компьютере и напоминающего виртуальную машину, в которой работает `kubelet`;
- `b7a49b42...` – имя контейнера Docker с кластером `kind`. Эту контрольную группу создает Docker;
- `kubepods` – подраздел контрольных групп, который Kubernetes выделяет для своих модулей Pod;
- `burstable` – специальная контрольная группа для Kubernetes, определяющая класс качества обслуживания, который получает планировщик;
- `pod1557...` – идентификатор модуля Pod, который отражается внутри ядра Linux с собственным идентификатором.

На момент написания этой книги фреймворк Docker был признан устаревшим в Kubernetes. Папку `docker` в этом примере можно рассматривать не как понятие из мира Kubernetes, а как «виртуальную машину, запускающую `kubelet`», потому что сам кластер `kind` на самом деле запускает лишь одного демона Docker, играющего роль узла Kubernetes, а затем внутри этого узла размещает `kubelet`, `containerd` и т. д. Проще говоря, продолжая изучать Kubernetes, продолжайте повторять себе «кластер `kind` не использует Docker для запуска контейнеров». Он использует Docker для создания узлов, на которые устанавливает среду выполнения контейнеров `containerd`.

Теперь вы знаете, что каждый процесс в Kubernetes (на компьютере с Linux) попадает в каталог `/proc`. Давайте далее посмотрим, какую информацию можно извлечь из этого каталога для более традиционного модуля Pod: контейнера NGINX.

4.2.3 Реализация контрольных групп для обычного модуля Pod

Модуль Pod планировщика представляет особый случай, потому что работает на всех кластерах и не поддерживает прямой возможности настройки или исследования. Более реалистичным выглядит сценарий, когда требуется убедиться в правильной реализации

контрольных групп для обычного приложения (например, NGINX). Чтобы попробовать свои силы в таком сценарии, можно создать модуль Pod, похожий на оригинальный `pod.yaml` с запросами ресурсов, в котором запускается веб-сервер NGINX. Спецификация этой части определения Pod приводится ниже (вероятно, она покажется вам знакомой):

```
спес:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources:
      requests:
        cpu: "1"
        memory: 1G
```

В этом случае спецификация объекта Pod определяет количество ядер (1) и объем памяти (1 Гбайт). Оба требования определяются контрольными группами в каталоге `/sys/fs` и применяются ядром. Чтобы убедиться в этом, нужно подключиться к узлу по `ssh` или, если вы используете `kind`, выполнить команду `docker exec -t -i 75 /bin/sh`, чтобы получить доступ к командной оболочке узла `kind`.

В результате контейнер NGINX получит выделенный доступ к 1 ядру и 1 Гбайт памяти. После создания этого модуля Pod можно исследовать его иерархию контрольных групп, соответствующих требованию в поле `memory` (снова выполнив команду `ps -ax`). При этом можно увидеть, что Kubernetes *на самом деле* удовлетворяет наш запрос на объем памяти. Мы предлагаем вам самим поэкспериментировать с другими ограничениями и посмотреть, как их выражает ОС.

Если теперь заглянуть в каталог `/proc`, то можно увидеть, что там имеется информация об объеме памяти, выделенной нашему модулю Pod. Он примерно равен 1 Гбайт. В момент создания предыдущего модуля Pod наша базовая среда выполнения контейнеров находилась в контрольной группе с ограниченным объемом памяти. Это решает проблему изоляции ресурсов памяти и процессора, которую мы изначально обсуждали в этой главе:

```
$ sudo cat /sys/fs/memory/docker/75../kubepods/pod8a58e9/d176../
memory.limit_in_bytes
999997440
```

Таким образом, волшебство изоляции в Kubernetes на компьютере с Linux можно рассматривать как обычное иерархическое распределение ресурсов, организованное с помощью простой структуры каталогов. В ядре имеется немало логики, помогающей «сделать все правильно», но то же самое доступно любому, у кого хватит смелости заглянуть под покров.

4.3 Тестирование контрольных групп

Мы уже знаем, как убедиться, что контрольные группы созданы правильно. Но как проверить, соблюдаются ли ограничения, устанавливаемые контрольными группами? Общеизвестно, что среда выполнения контейнеров и ядро Linux могут иметь ошибки и несоответствия с нашими ожиданиями в реализации механизмов изоляции. Например, иногда ОС может позволить контейнеру потреблять больше выделенного ему процессорного времени, если другие процессы не нуждаются в этом ресурсе. Давайте запустим простой процесс с помощью следующего кода и проверим, правильно ли работают наши контрольные группы:

```
$ cat /tmp/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
  labels:
    role: just-an-example
    app: my-example-app
    organization: friends-of-manning
    creator: jay
spec:
  containers:
    - name: an-old-name-will-do
      image: busybox:latest
      command: ['sleep', '1000']
      resources:
        limits:
          cpu: 2
        requests:
          cpu: 1
      ports:
        - name: webapp-port
          containerPort: 80
          protocol: TCP
```

Гарантировать модулю Pod возможность использовать много вычислительных ресурсов

Гарантирует, что модуль Pod не запустится, пока не получит полное ядро процессора

Теперь можно запустить в этом модуле команду, интенсивно потребляющую процессорное время. Команда `top` поможет увидеть происходящее:

```
$ kubectl create -f pod.yaml
$ kubectl exec -t -i core-k8s /bin/sh
#> dd if=/dev/zero of=/dev/null
$ docker exec -t -i 75 /bin/sh
root@kube-control-plane# top
```

Запуск командной оболочки в контейнере

Запуск команды `dd`, впуская расходующей процессорное время

Запуск команды `top` для оценки потребления процессора


```
PID USER      PR  NI   VIRT   RES    SHR S  %CPU  %MEM  TIME+  COMMAND
91467 root       20   0   1292     4      0 R   99.7   0.0  0:35.89 dd
```

Что произойдет, если ужесточить ограничения и повторить эксперимент? Давайте для проверки изменим раздел `resources`:

```
resources:
  limits:
    cpu: .1
  requests:
    cpu: .1
```

Ограничить использование процессора до 0,1 ядра

Зарезервировать 0,1 долю ядра процессора для этого модуля Pod

Повторно запустив ту же команду, можно наблюдать менее напряженный сценарий работы узла `kind`:

```
root@kube-control-plane# top
```

На этот раз узел использует только 10 % процессорного времени

```
PID USER      PR  NI   VIRT   RES    SHR S  %CPU  %MEM  TIME+  COMMAND
93311 root       20   0   1292     4      0 R   10.3   0.0  0:03.61 dd
```

4.4 Как kubelet управляет контрольными группами

Выше в этой главе мы не затронули некоторые другие контрольные группы, такие как `blkio`. Безусловно, существует множество видов контрольных групп, и, конечно же, важно понимать их назначения, хотя в 90 % случаев основной интерес представляют только изоляция процессора и памяти.

На более низком уровне примитивы контрольных групп, перечисленные в `/sys/fs/cgroup`, предоставляет средства управления выделения соответствующих ресурсов процессам. Некоторые такие группы не особенно полезны для администраторов Kubernetes. Например, контрольная группа `freezer` связывает группы родственных задач с единственной контрольной точкой приостановки («заморозки»). Этот примитив изоляции позволяет эффективно планировать и приостанавливать группы процессов (по иронии судьбы некоторые критикуют Kubernetes за то, что он плохо справляется с этим видом планирования).

Другой пример – контрольная группа `blkio`, менее известный ресурс, используемый для управления вводом/выводом. Заглянув в `/sys/fs/cgroup`, можно увидеть все контролируемые ресурсы, которые в Linux можно распределить иерархически:

```
$ ls -ld /sys/fs/cgroup/*
/sys/fs/cgroup/blkio freezer perf_event
/sys/fs/cgroup/cpu hugetlb pids
/sys/fs/cgroup/cpuacct memory rdma
/sys/fs/cgroup/cpu,cpuacct net_cls systemd
```

```
/sys/fs/cgroup/cpuset net_cls,net_prio unified  
/sys/fs/cgroup/devices net_prio
```

Узнать больше о первоначальном предназначении контрольных групп можно по адресу <http://mng.bz/vo8p>. Некоторые из статей могут оказаться устаревшими и неактуальными, но они содержат много интересных сведений об истории развития контрольных групп и их предназначении. Для опытных администраторов Kubernetes понимание особенностей интерпретации этих структур данных может очень пригодиться, особенно при использовании различных технологий контейнеризации.

4.5 Как kubelet управляет ресурсами

Теперь, представляя, как формируются контрольные группы, можно взглянуть на особенности их использования в kubelet, точнее в структуре данных `allocatable`. Выполнив команду `kubectl get nodes -o yaml` на узле Kubernetes (или в кластере `kind`), можно увидеть следующий раздел в выводе:

```
...  
  allocatable:  
    cpu: "12"  
    ephemeral-storage: 982940092Ki  
    hugepages-1Gi: "0"  
    hugepages-2Mi: "0"  
    memory: 32575684Ki  
    pods: "110"
```

Эти параметры выглядят подозрительно знакомыми. Так и должно быть. Эти ресурсы представляют суммарный бюджет контрольных групп, доступный для выделения ресурсов модулям Pod. Агент kubelet вычисляет его, определяя общую емкость узла. Затем определяет пропускную способность процессора, необходимую для него самого и базового узла, и вычитает ее из доступного объема ресурсов. Соответствующие уравнения можно найти по адресу <http://mng.bz/4jJR> и настраивать с помощью параметров, таких как `--system-reserved` и `--kubelet-reserved`. Полученные значения используются планировщиком Kubernetes при принятии решения о возможности размещения контейнера на этом конкретном узле.

Как правило, в `--kubelet-reserved` и `--system-reserved` можно передать 0,5 ядра, оставляя 1,5 ядра двухъядерного процессора свободными для выполнения рабочих нагрузок, потому что kubelet потребляет не так много вычислительных ресурсов (за исключением периодов планирования или запуска большого количества задач). В больших кластерах эти числа зависят от множества факторов, связанных с типами рабочих нагрузок и оборудования, задержками в сети и т. д.

Для планирования используется следующее уравнение (член «ресурсы_для_системы» определяет количество ресурсов, необходимых для работы ОС):

$$\text{Доступно_для_распределения} = \text{емкость_узла} - \text{ресурсы_для_kube} - \text{ресурсы_для_системы}$$

Например, если:

- имеется узел с 16-ядерным процессором;
- 1 ядро зарезервировано для kubelet и системных процессов в кластере,

то для распределения будет доступно 15 ядер. В таком случае:

- systemd запустит kubelet, который периодически передает информацию о доступных ресурсах в Kubernetes API;
- systemd запустит также среду выполнения контейнеров (containerd, CRI-O или Docker);
- kubelet создаст контрольные группы при запуске модулей Pod, чтобы ограничить потребление ими ресурсов;
- среда выполнения контейнеров запустит процесс внутри контрольных групп, что гарантирует предоставление запрошенных ресурсов, указанных в спецификации модуля.

Вместе с kubelet запускается встроенная родительская логика. Соответствующий параметр настраивается с помощью флага командной строки (который следует оставить включенным), в результате чего kubelet становится родительской контрольной группой cgroup верхнего уровня для дочерних контейнеров. Предыдущее уравнение вычисляет общее количество контрольных групп, доступных для kubelet, которое называется *бюджетом распределяемых ресурсов*.

4.5.1 Почему ОС не может использовать подкачку в Kubernetes?

Чтобы разобраться в этом вопросе, нужно немного глубже исследовать некоторые контрольные группы, уже упоминавшиеся выше. Помните, как наши модули Pod размещались в специальных папках, таких как *guaranteed* и *burstable*? Если бы мы разрешили операционной системе вытеснять неактивные области памяти на диск, то могли бы столкнуться с ситуацией, когда простаивающий процесс внезапно начал медленно выделять память. Это нарушило бы *гарантированный* доступ к памяти, указанный в спецификации модуля, и сделало бы производительность сильно изменчивой.

Поскольку предсказуемость планирования большого количества процессов важнее работоспособности любого отдельного процесса, мы полностью отключаем подкачку в Kubernetes. Чтобы избежать путаницы, программы установки Kubernetes, такие как *kubeadm*, мно-

венно перестают работать, если обнаруживается попытка загрузить и запустить kubelet на машине с включенной подкачкой.

Почему бы не включить подкачку?

В некоторых случаях включение механизма подкачки памяти может принести пользу (например, позволит плотнее упаковывать контейнеры в системе). Однако семантическая сложность, связанная с таким подходом, невыгодна большинству пользователей. Разработчики kubelet еще не решили (пока), стоит ли поддерживать это более сложное понятие памяти, а кроме того, необходимые для этого изменения будет сложно внести в API такой системы, как Kubernetes, используемой миллионами пользователей.

Конечно, эта технология, как и многие другие, быстро развивается, и на самом деле в Kubernetes 1.22 уже имеется возможность запуска с включенной подкачкой памяти (<http://mng.bz/4jY5>). Однако в большинстве промышленных систем делать это не рекомендуется, потому что есть риск получить очень неустойчивые характеристики производительности рабочих нагрузок.

На уровне среды выполнения контейнеров есть много тонкостей, связанных с использованием ресурсов, таких как память. Например, контрольные группы делятся на мягкие и жесткие ограничения:

- процесс с мягкими ограничениями памяти может иметь разные объемы ОЗУ в разные моменты времени в зависимости от нагрузки на систему;
- процесс с жесткими ограничениями памяти уничтожается, если в течение длительного времени будет удерживать объем памяти больше установленного лимита.

Обратите внимание, что при завершении процессов по этим причинам Kubernetes возвращает код выхода и статус OOMKilled. Вы можете увеличить объем памяти, выделенной высокоприоритетному контейнеру, чтобы уменьшить вероятность проблем, обусловленных «шумными соседями». Давайте продолжим рассмотрение этой темы.

4.5.2 Хак: настройка приоритета «для бедных»

Изначально концепция огромных страниц *HugePages* не поддерживалась в Kubernetes, потому что это была веб-технология. Но по мере изменения базовой технологии центров обработки данных актуальными стали более тонкие стратегии планирования и распределения ресурсов. Конфигурация *HugePages* позволяет модулю Pod получать страницы памяти с размером, превышающим размер страниц памяти в ядре Linux, который обычно составляет 4 Кбайт.

Память, как и процессор, можно явно выделять модулям Pod и обозначать с использованием единиц измерения в килобайтах, мегабай-

тах и гигабайтах (Ki, Mi и Gi соответственно). Многие приложения, активно использующие память, такие как Elasticsearch и Cassandra, поддерживают технологию HugePages. Если узел поддерживает HugePages и способен распределять память страницами по 2048 КиБ, то он предоставляет планируемый ресурс: HugePages-2Mi. В Kubernetes планирование HugePages возможно с помощью стандартной директивы `resources`:

```
resources:
  limits:
    hugepages-2Mi: 100Mi
```

Технология *Transparent HugePages* – это оптимизация стандартной технологии HugePages. Она может оказывать очень разное влияние на модули Pod, которым требуется высокая производительность. В некоторых случаях желательно отключить ее, особенно при использовании высокопроизводительных контейнеров, которым требуются большие непрерывные блоки памяти на уровне загрузчика или ОС.

4.5.3 Хак: настройка HugePages с помощью контейнеров инициализации

Мы прошли полный круг. Помните, как в начале этой главы мы рассматривали применение каталога `/sys/fs` для управления различными ресурсами для контейнеров? Настройку HugePages можно выполнить в контейнерах инициализации `init`, если есть возможность запустить их от имени пользователя `root` и смонтировать `/sys`, чтобы отредактировать эти файлы.

Конфигурацию HugePages можно переключать, просто записывая файлы в каталог `sys` и удаляя их из него. Например, отключить поддержку Transparent HugePages, влияющую на производительность в определенных ОС, можно командой `echo 'never' > /sys/kernel/mm/redhat_transparent_hugepage/enabled`. Настроить HugePages определенным образом можно полностью из спецификации Pod.

- 1 Объявить объект Pod, предъявляющий особые требования к производительности, основанные на HugePages.
- 2 Объявить контейнер `init` с модулем, соответствующим этому объекту Pod, который запускается в привилегированном режиме и монтирует каталог `/sys`, используя тип тома `hostPath`.
- 3 Выполнить в контейнере `init` необходимые команды Linux (например, команду `echo`, показанную выше) в качестве шагов выполнения.

Как правило, контейнеры `init` можно использовать для настройки определенных особенностей Linux, которые могут потребоваться модулю Pod. Но имейте в виду, что для монтирования тома типа `hostPath` нужны особые привилегии в кластере, которые ваш администратор может предоставить только после неоспоримого обоснования. Неко-

торые дистрибутивы, такие как OpenShift, по умолчанию запрещают монтирование томов `hostPath`.

4.5.4 Классы QoS: почему они важны и как они работают

В этой главе мы то там, то тут использовали такие термины, как *guaranteed* и *burstable*, но пока не дали им определений. Но, чтобы определить эти понятия, сначала нужно определить понятие качества обслуживания (Quality of Service, QoS).

Направляясь в модный ресторан, вы ожидаете получить там блюда высочайшего качества, а также учтивость и отзывчивость официантов. Эти учтивость и отзывчивость известны как *качество обслуживания* (Quality of Service, QoS). Мы упоминали QoS выше, когда говорили, что подкачка памяти в Kubernetes отключается ради гарантий высокой производительности доступа к памяти. Под QoS понимается доступность ресурсов в любой момент. Любой центр обработки данных, гипервизор или облако предполагают компромиссы, связанные с доступностью ресурсов для приложений:

- платить больше за гарантии бесперебойной работы критически важных сервисов, потому что необходимо иметь больше оборудования, чем нужно;
- платить меньше и рисковать бесперебойностью обслуживания.

QoS позволяет идти по тонкой грани, когда многие службы работают неоптимально в часы пик, не жертвуя качеством критически важных сервисов. На практике такими критическими сервисами могут быть системы обработки платежей, задачи машинного обучения или искусственного интеллекта, перезапуск которых обходится очень дорого, или процессы, осуществляющие коммуникации в масштабе реального времени, которые нельзя прервать. Имейте в виду, что вытеснение модуля Pod во многом зависит от того, насколько он превышает лимит ресурсов. В общем случае:

- приложения с предсказуемым потреблением памяти и процессора подвергаются вытеснению с меньшей вероятностью;
- жадные приложения с большей вероятностью будут вытеснены в периоды пиковых нагрузок, если попытаются использовать больше процессорного времени или памяти, чем выделено Kubernetes, и при условии, что они не принадлежат к классу *Guaranteed*;
- приложения с классом *BestEffort* – первые кандидаты на вытеснение и перепланирование в периоды пиковых нагрузок.

Возможно, вам интересно, как выбирается правильный класс QoS. Обычно класс определяется не напрямую, а опосредованно, через определение необходимости приложению гарантированного доступа к ресурсам в разделе `resources`. Как это делается, мы рассмотрим в следующем разделе.

4.5.5 Создание классов QoS путем настройки ресурсов

Burstable, Guaranteed и BestEffort – это три класса QoS, которые создаются в зависимости от определения объекта Pod. Эти параметры могут помочь увеличить количество контейнеров, запускаемых в кластере, часть из которых затем может отключаться в периоды высокой загрузки и вновь запускаться позже. Заманчиво определить глобальные политики, управляющие выделением процессорного времени или памяти конечным пользователям, но имейте в виду, нет универсальных политик, подходящих под все случаи жизни:

- если все контейнеры отнести к классу гарантированного качества обслуживания (Guaranteed QoS), вам будет трудно обслуживать динамические рабочие нагрузки с изменяющимися потребностями в ресурсах;
- если на серверах не будет контейнеров с классом Guaranteed QoS, то kubelet не сможет поддерживать работу некоторых важных процессов.

Вот основные правила определения класса QoS (вычисляется и отображается в поле status модуля Pod):

- *BestEffort* – к этому классу относятся модули Pod, не определяющие требуемые количества памяти или ядер процессора. Они первые кандидаты на вытеснение (и перезапуск на другом узле), когда образуется нехватка ресурсов;
- *Burstable* – к этому классу относятся модули Pod, определяющие требуемые количества памяти или ядер процессора, но не устанавливающие ограничений для обоих ресурсов. Они вытесняются с меньшей вероятностью, чем модули с классом BestEffort;
- *Guaranteed* – к этому классу относятся модули Pod, определяющие требуемые количества памяти и ядер процессора и устанавливающие ограничения для обоих ресурсов. Они вытесняются и перемещаются в последнюю очередь.

Давайте посмотрим, как в действительности определяются эти классы. Создайте новое развертывание, выполнив команду `kubectl create ns qos; kubectl -n qos run --image=nginx myapp`. Затем отредактируйте развертывание, включив спецификацию контейнера, определяющую требуемые объемы ресурсов, но не определяющую ограничений. Например:

```
спес:
containers:
- image: nginx
  imagePullPolicy: Always
  name: nginx
resources:
  requests:
    cpu: "1"
    memory: 1G
```


Теперь вы увидите, что команда `kubectl get pods -n qos -o yaml` вернет класс `Burstable` в поле `status` определения модуля `Pod`, как показано ниже. Вы можете использовать этот прием, чтобы гарантировать присваивание всем наиболее важным процессам класса качества обслуживания `Guaranteed` или `Burstable`.

```
hostIP: 172.17.0.3
phase: Running
podIP: 192.168.242.197
qosClass: Burstable
startTime: "2020-03-08T08:54:08Z"
```

4.6 Мониторинг ядра Linux с помощью Prometheus, cAdvisor и сервера API

В этой главе мы рассмотрели множество низкоуровневых понятий Kubernetes и сопоставили их с понятиями в ОС, но в своей практике вам едва ли придется контролировать все это вручную. Вместо этого вся информация уровня контейнера и системы в целом обычно собирается в единой панели мониторинга, чтобы в случае чрезвычайных ситуаций можно было определить момент появления проблемы и исследовать ее с различных точек зрения (приложения, ОС и т. д.).

В заключение этой главы мы поднимемся уровнем выше и воспользуемся Prometheus – популярным инструментом мониторинга облачных приложений и самой платформы Kubernetes, ставшего отраслевым стандартом. Мы посмотрим, как можно количественно оценить использование ресурсов модулем `Pod` путем прямого анализа контрольных групп. Этот подход имеет несколько преимуществ, когда возникает необходимость в организации сквозной наблюдаемости системы:

- позволяет видеть скрытые процессы, создающие избыточную нагрузку и невидимые для Kubernetes;
- позволяет напрямую отображать ресурсы, доступные Kubernetes, с помощью инструментов изоляции уровня ядра, которые могут обнаруживать ошибки взаимодействия кластера с ОС;
- дает отличную возможность получить дополнительную информацию о масштабировании контейнеров с помощью `kubelet` и среды выполнения контейнеров.

Прежде чем перейти к Prometheus, поговорим о метриках. *Метрика* – это некое количественное значение; например, количество чизбургеров, съеденных за истекший месяц. Во вселенной Kubernetes мириады контейнеров, запускающихся и останавливающихся в центре обработки данных, делают метрики приложений важными для администраторов, позволяя им объективно и независимо судить о состоянии приложений и сервисов центра обработки данных.

Если придерживаться метафоры с чизбургерами, у нас может быть ряд метрик, как показано в следующем фрагменте кода, которые можно зафиксировать в журнале. Рассмотрим три основных типа метрик – гистограммы, датчики и счетчики:

- *датчики*: сообщают количество запросов в секунду, получаемых в любой момент времени;
- *гистограммы*: отображают количества событий с разбивкой на временные интервалы (например, сколько запросов было обработано менее чем за 500 мс);
- *счетчики*: сообщают постоянно увеличивающееся количество событий (например, общее количество полученных запросов).

В качестве конкретного примера, более близкого к повседневной реальности, можно вывести метрики Prometheus о нашем ежедневном потреблении калорий, как показано ниже:

```
meals_today 2
cheeseburger 1
salad 1
dinner 1
lunch 1
calories_total_bucket_bucket[le=1024] 1
```

Общее количество приемов пищи на сегодня

Количество съеденных чизбургеров

Количество потребленных калорий, разбитое на группы по 2, 4, 8, 16 и т. д., до 2048

Вы можете опубликовать общее количество приемов пищи за день. Эта метрика называется *датчиком*, она постоянно обновляется и может увеличиваться или уменьшаться. Количество чизбургеров, съеденных сегодня, будет *счетчиком*, который постоянно увеличивается с течением времени. Метрика, сообщающая количество потребленных калорий, говорит о том, что в один из приемов пищи было потреблено менее 1024 калорий. Эта гистограмма дает способ дискретно определить количество потребленных калорий, не увязая в деталях (все, что выше 2048, вероятно, слишком много, а все, что ниже 1024, скорее всего, слишком мало).

Обратите внимание, что такая дискретизация часто используется для мониторинга etcd. Количество операций записи, длящихся более 1 с, – важный показатель, помогающий прогнозировать продолжительность периодов недоступности etcd. Со временем, агрегировав ежедневные записи в журнале, можно выявить некоторые интересные корреляции. Например:

```
meals_today 2
cheeseburger 50
salad 99
dinner 101
lunch 99

calories_total_bucket_bucket[le=512] 10
calories_total_bucket_bucket[le=1024] 40
calories_total_bucket_bucket[le=2048] 60
```

Если построить графики, отложив значения метрик по оси *y* и время по оси *x*, можно увидеть, что:

- количество дней, когда вы ели чизбургеры, обратно пропорционально количеству дней, когда вы завтракали;
- количество съеденных чизбургеров неуклонно растет.

4.6.1 Публикация метрик обходится недорого и имеет большую ценность

Метрики важны для контейнерных и облачных приложений, но управление ими должно быть легковесным и независимым. Система мониторинга *Prometheus* предлагает инструменты для масштабирования метрик без ненужных шаблонов или фреймворков, которые только мешают. Она проектировалась с учетом следующих требований:

- обслуживание стен и тысяч процессов, публикующих схожие метрики, а это означает, что каждая метрика должна снабжаться меткой, помогающей различать процессы;
- приложения должны иметь возможность публиковать метрики независимо от языка, на котором они написаны;
- приложения должны иметь возможность публиковать метрики, не обременяя себя знанием, как эти метрики используются;
- публикация метрик должна реализовываться легко и просто и не зависеть от используемого языка программирования.

Решив реализовать журналирование метрик из предыдущей аналогии, мы могли бы объявить экземпляры *cheeseburger*, *meals_today* и *calories_total* с типами *counter* (счетчик), *gauge* (датчик) и *histogram* (гистограмма) соответственно. Это типы *Prometheus API*, и они поддерживают автоматическое сохранение локальных значений в памяти, которые можно извлечь в виде CSV-файла из локальной конечной точки. Обычно это делается путем добавления обработчика *Prometheus* на сервер *REST API*, который обслуживает только одну значимую конечную точку: *metrics/*. Для управления этими данными можно использовать клиента *Prometheus API*:

- периодически проверять значение количества приемов *meals_today*, выполняя вызов *Gauge API*;
- периодически увеличивать значение *cheeseburger* – количество съеденных чизбургеров;
- ежедневно агрегировать значение *calories_total*, которое может быть получено из другого источника данных.

Со временем мы могли бы попытаться определить, связано ли употребление чизбургеров с более высоким количеством калорий, потребленных за день, и с другими метриками (например, с весом тела). Такую возможность может обеспечить любая база данных, способная хранить временные ряды, но *Prometheus*, как легковесный механизм метрик, прекрасно подходит для использования в контейнерах, по-

тому что независим, не имеет состояния и стал фактическим стандартом поддержки метрик в любых приложениях.

Не ждите, пока публикация метрик будет настроена централизованно

Prometheus часто ошибочно считают тяжеловесной системой, которая должна устанавливаться централизованно. На самом деле это просто *инструмент учета* с открытым исходным кодом и API, который можно встроить в любое приложение. Конечно, тот факт, что ведущий узел Prometheus может собирать и интегрировать эту информацию, играет немаловажную роль, но он не является обязательным требованием, чтобы начать публикацию и сбор метрик для вашего приложения.

Любой микросервис может начать публиковать метрики через конечную точку, импортировав клиента Prometheus. Ваш кластер может не использовать эти метрики, но нет причин не сделать их доступными на стороне контейнера, хотя бы ради возможности использовать эту конечную точку для проверки вручную различных количественных аспектов вашего приложения. А позднее вы сможете развернуть специальный ведущий узел Prometheus для организации непрерывного наблюдения за работой приложения.

Клиенты Prometheus существуют для всех основных языков программирования. Благодаря этому любой микросервис сможет легко и недорого вести ежедневный журнал различных событий в виде метрики Prometheus.

4.6.2 Почему Prometheus?

В этой книге мы сосредоточимся на Prometheus, потому что эта система стала стандартом де-факто в облачном окружении. Но мы постараемся убедить вас, что она заслужила этот статус, на простом и убедительном примере, демонстрирующем, насколько легко и быстро организовать проверку работоспособности сервера API. Например, с ее помощью можно посмотреть, сильно ли нагружен сервер Kubernetes API запросами на запуск новых модулей Pod, выполнив следующие команды в терминале (при условии, что у вас уже запущен и работает кластер такого типа). В отдельном терминале запустите команду `kubectl proxy`, а затем используйте `curl` для обращения к конечной точке `metrics` на сервере API:

```
$ kubectl proxy
```

Позволяет получить доступ к серверу
Kubernetes API на localhost:8001

```
$> curl localhost:8001/metrics |grep etcd
```

Вызов конечной точки
metrics сервера API

```
etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.005"}
```

174

```

etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.01"}
194
etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.025"}
201
etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.05"}
203

```

Любой, имеющий клиента `kubectl`, может немедленно использовать команду `curl` для получения метрики с временем отклика определенной конечной точки API. В предыдущем фрагменте можно видеть, что почти все вызовы `get` конечной точки API модуля `Pod` возвращаются менее чем за 0,025 с, что обычно считается хорошей производительностью. В оставшейся части этой главы мы настроим систему мониторинга Prometheus для нашего кластера с нуля.

4.6.3 *Создание локального сервиса мониторинга Prometheus*

Службу мониторинга Prometheus можно использовать для проверки контрольных групп и потребления системных ресурсов. Архитектура системы мониторинга Prometheus (рис. 4.2) в `kind` включает следующие компоненты:

- мастер (ведущий узел) Prometheus;
- сервер Kubernetes API, за которым наблюдает мастер;
- множество агентов `kubelet` (в нашем случае один), каждый из которых играет роль источника метрик для сервера API.

Обратите внимание, что мастер Prometheus может собирать метрики из множества разных источников, включая серверы API, аппаратные узлы, автономные базы данных и даже автономные приложения. Однако не всегда удобно агрегировать все данные на сервере Kubernetes API. В этом простом примере мы хотим показать, как использовать Prometheus для конкретной цели – мониторинга ограничения потребления ресурсов в Kubernetes с помощью контрольных групп, поэтому для удобства мы собираем данные со всех узлов прямо на сервере API. Также обратите внимание, что в этом примере наш кластер `kind` имеет только один узел. Даже если бы у нас было больше узлов, мы все равно могли бы собрать все данные непосредственно на сервере API, добавив дополнительные поля `target` в файл YAML (который мы покажем чуть ниже).

Запустим Prometheus с помощью следующего конфигурационного файла `prometheus.yaml`:

```

$ mkdir data
$ ./prometheus-2.19.1.darwin-amd64/prometheus \
  --storage.tsdb.path=./data --config.file=./prometheus.yaml

```

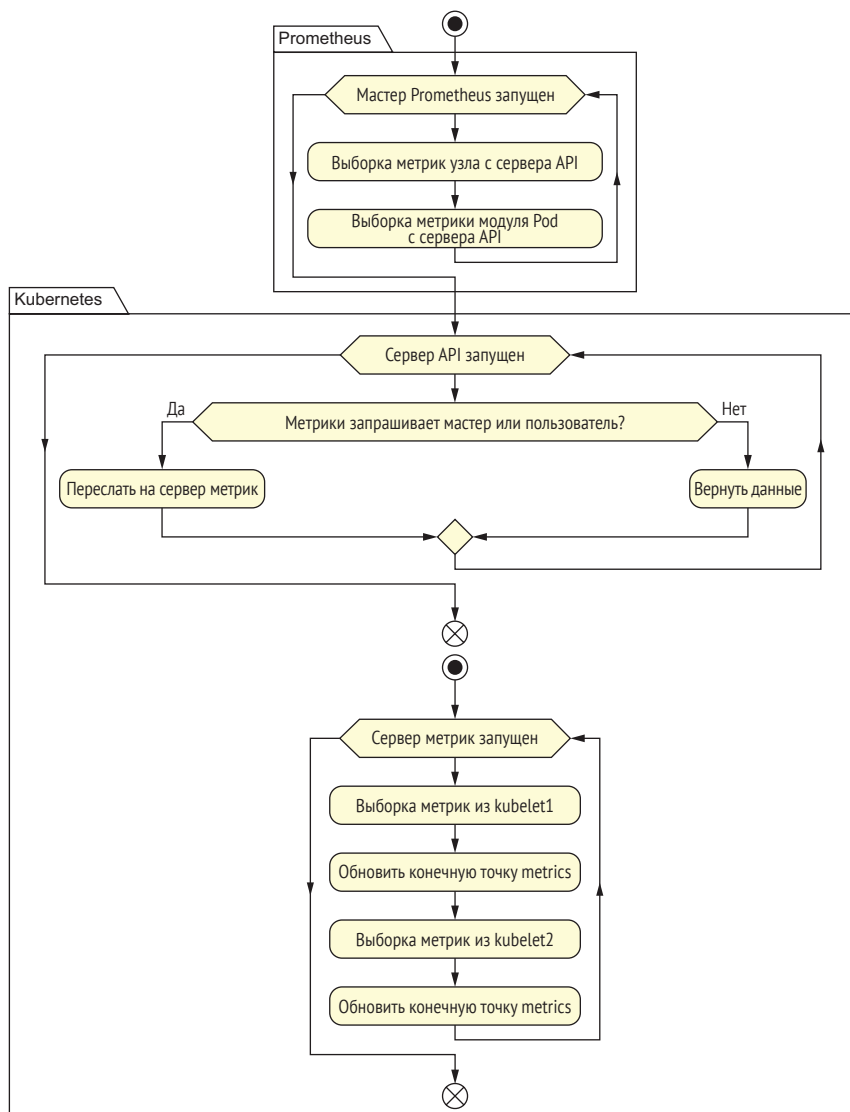


Рис. 4.2 Архитектура мониторинга Prometheus

Для мониторинга контрольных групп и сбора количественных данных (например, доли процессорного времени и объема памяти потребляемых модулем Pod в конкретной группе) kubelet использует библиотеку cAdvisor. Вы уже знаете, как просматривать иерархии файловой системы контрольных групп, поэтому вывод метрик, собранных с помощью cAdvisor, должен быть вам понятен. Для сбора метрик мы потребуем от Prometheus запрашивать сервер API каждые 3 с, например так:

```

global:
  scrape_interval: 3s
  evaluation_interval: 3s

scrape_configs:
  - job_name: prometheus
    metrics_path:
      /api/v1/nodes/kind-control-plane/
      proxy/metrics/cadvisor
    static_configs:
      - targets: ['localhost:8001']

```

Узел с плоскостью управления kind – единственный узел в нашем кластере

Сюда можно добавить дополнительные узлы для сбора метрик

Действующие конфигурации Prometheus должны учитывать ограничения реального мира. К ним относятся объем данных, безопасность и протоколы оповещения. Обратите внимание, что базы данных для хранения временных рядов известны своей расточительностью в отношении потребления дискового пространства, и соответствующие метрики могут многое рассказать о возможных угрозах для вашей организации. Как отмечалось выше, они могут быть неважны на первых этапах проектирования, но вы можете начать с публикации метрик на уровне приложений, а полноценную конфигурацию системы мониторинга Prometheus развернуть позже. В нашем простом примере это все, что нужно для настройки Prometheus с целью изучения контрольных групп.

Напомним еще раз, что сервер API периодически получает данные от kubelet, поэтому стратегия, основанная на использовании единственной конечной точки, более чем оправдана. В ином случае мы могли бы собирать данные прямо из kubelet или даже запустить свой сервис sAdvisor. А теперь давайте посмотрим на общую метрику *потребления процессора контейнером в секундах*, выполнив следующую команду.

ВНИМАНИЕ Эта команда создает большой сетевой трафик и пик потребления процессора на компьютере.

```

$ kubectl apply -f \
https://raw.githubusercontent.com/
➤ giantswarm/kube-stresscheck/master/examples/node.yaml

```

Эта команда запускает серию ресурсоемких контейнеров, отличающихся значительным потреблением сетевых ресурсов, памяти и процессорного времени. Если выполнить эту команду на ноутбуке, то гигантский рой контейнеров, созданный ею, наверняка вызовет сильную нагрузку на процессор, и вы можете услышать шум вентилятора.

На рис. 4.3 показано, как выглядит наш кластер kind под нагрузкой. Мы оставляем вам как самостоятельное упражнение сопоставление различных контрольных групп и метаданных контейнеров (их можно увидеть, наведя указатель мыши на метрики Prometheus) с процессами и контейнерами в вашей системе. В частности, советуем обратить

внимание на следующие метрики, чтобы получить представление о мониторинге в Prometheus на уровне процессора. Изучение этих метрик с сотнями других при запуске рабочих нагрузок или контейнеров поможет вам создать надежные протоколы мониторинга и оценки ваших внутренних системных конвейеров:

- `container_memory_usage_bytes`;
- `container_fs_writes_total`;
- `container_memory_cache`.

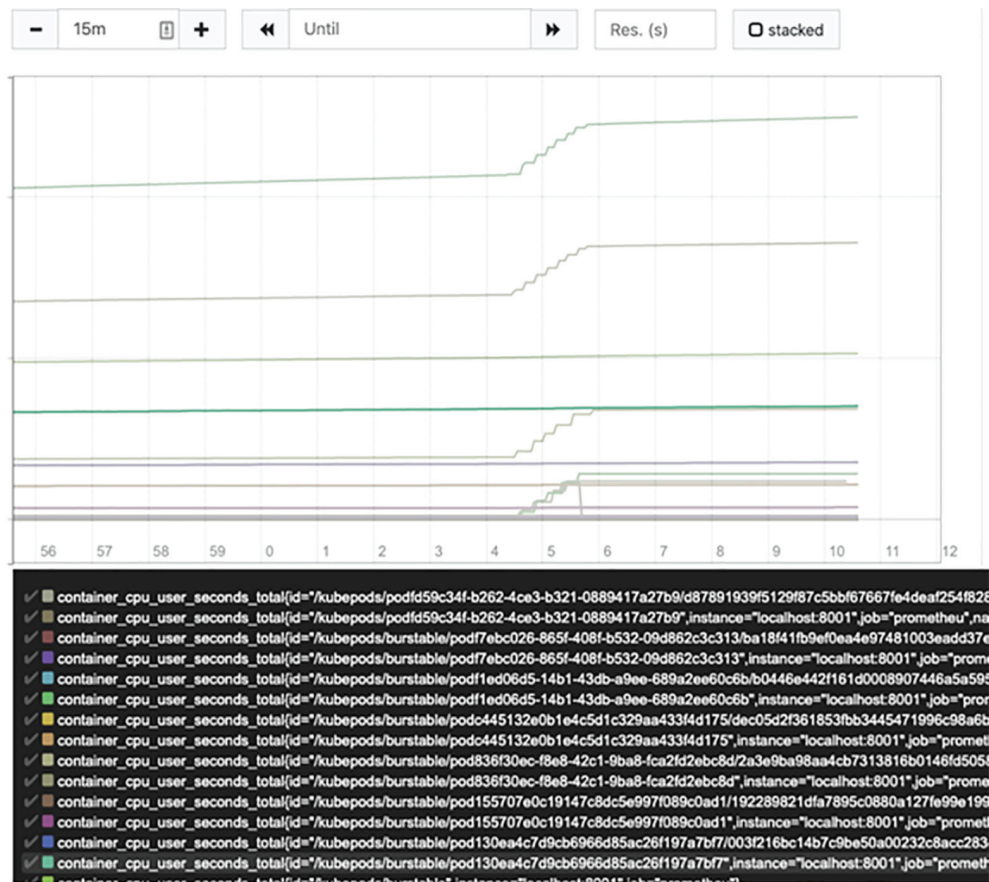


Рис. 4.3 Графики изменения метрик в высоконагруженном кластере

4.6.4 Исследование простоев в Prometheus

Прежде чем завершить эту главу, рассмотрим чуть подробнее три типа метрик. На рис. 4.4 представлены графики изменения трех метрик, помогающие по-разному взглянуть на одну и ту же ситуацию, возникшую в центре обработки данных. В частности, датчик дает логическое значение – признак работоспособности кластера. Гисто-

грамма показывает распределение запросов до полной потери приложения. А счетчик показывает общее количество транзакций, приведших к сбою:

- показания датчика представляют наибольшую ценность для дежурного персонала, наблюдающего за работой приложения;
- гистограмма может пригодиться инженерам, которые «на следующий день» займутся расследованием причин, приведших к длительному простоя микросервиса;
- счетчик может помочь определить, сколько запросов было успешно обработано до сбоя. Например, если в приложении имеется утечка памяти, то ее можно обнаружить по предсказуемому сбою веб-сервера после определенного количества запросов (скажем, 15 000 или 20 000).

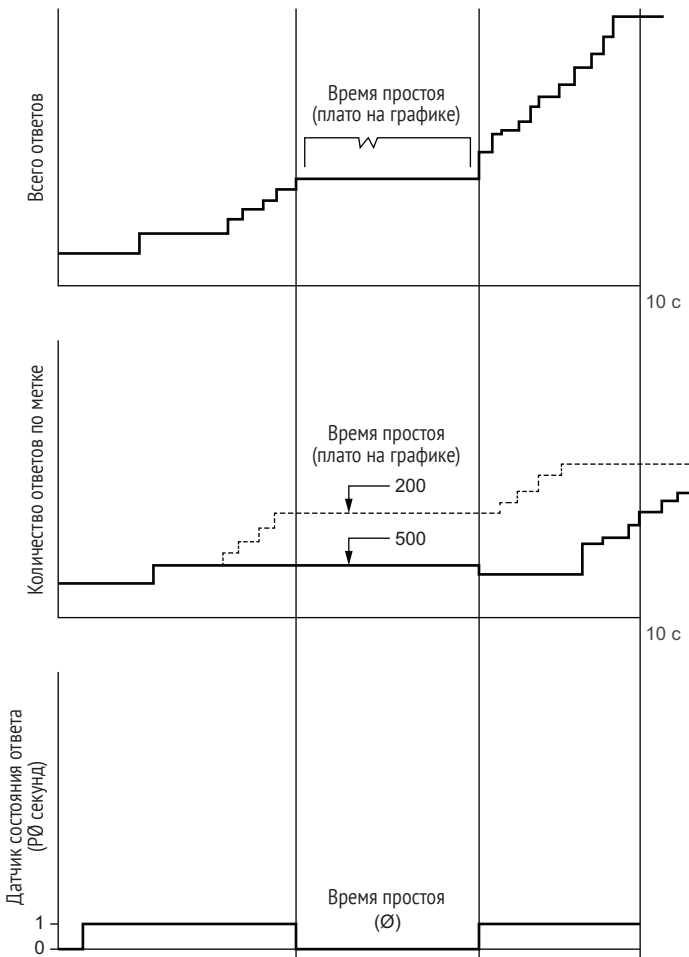


Рис. 4.4 Сравнение графиков изменения датчика, гистограммы и счетчика, отражающих одну и ту же ситуацию в кластере

В конечном счете вам решать, какие метрики использовать для принятия решений, но имейте в виду, что набор метрик не должен быть просто свалкой информации. Они должны помогать видеть, как развивались события в ваших сервисах, как они вели себя и взаимодействовали друг с другом. Обобщенные метрики редко бывают полезны для отладки сложных проблем, поэтому найдите время, чтобы встроить клиента Prometheus в свои приложения и собрать некоторые количественные метрики. Ваши администраторы будут вам благодарны! Мы еще вернемся к метрикам в главе, посвященной etcd, так что не волнуйтесь – обсуждение Prometheus на этом не заканчивается!

Итоги

- Ядро выражает ограничения для контейнеров в форме контрольных групп.
- Агент kubelet запускает процессы планировщика и зеркалирует их для сервера API.
- Для проверки особенностей ограничения потребления памяти контрольными группами можно использовать простые контейнеры.
- Агент kubelet поддерживает классы QoS, определяющие квоты ресурсов для процессов в модулях Pod.
- Для записи и просмотра метрик, характеризующих работу кластера в режиме реального времени, можно использовать Prometheus.
- Prometheus поддерживает три основных типа метрик: датчики, гистограммы и счетчики.

Интерфейсы CNI и настройка сети в модулях Pod

В этой главе:

- определение Kubernetes SDN с точки зрения kube-proxy и CNI;
- традиционные инструменты SDN в Linux и плагины CNI;
- использование технологий с открытым исходным кодом для управления работой CNI;
- CNI-провайдеры Calico и Antrea.

Для управления балансировкой нагрузки, изоляцией и безопасностью виртуальных машин в облаке, а также во многих локальных центрах обработки данных традиционно используется программно-определяемая сеть (Software-Defined Networking, SDN). SDN избавляет системных администраторов от излишней нагрузки, позволяя перенастраивать сети больших центров обработки данных каждую неделю или даже каждый день, при создании новых или уничтожении старых виртуальных машин. С наступлением эпохи контейнеров идея SDN приобрела совершенно новый смысл, потому что в контейнерных окружениях сети постоянно меняются и по определению должны поддерживаться с помощью автоматизированного программного обеспечения. Сеть Kubernetes полностью определяется программно и постоянно меняется из-за эфемерного и динамического характера модулей Pod в Kubernetes и конечных точек сервисов.

В этой главе мы рассмотрим организацию сетевых взаимодействий между модулями Pod и, в частности, как сотни и тысячи контейнеров, размещенные на одной машине, могут иметь уникальные IP-адреса, маршрутизируемые кластером. Kubernetes предоставляет эту функциональность модульным и расширяемым способом с использованием стандарта сетевого интерфейса контейнеров (Container Network Interface, CNI), который может быть реализован с применением самых разных технологий и способный предоставить каждому модулю Pod уникальный маршрутизируемый IP-адрес.

Спецификация CNI не определяет деталей контейнерной сети

Спецификация CNI – это общее определение высокоуровневых операций добавления контейнера в сеть. Углубление во все тонкости, если походить к чтению с точки зрения провайдера Kubernetes CNI, может вызвать некоторые затруднения в понимании. Например, некоторые плагины CNI, такие как IPAM (<https://www.cni.dev/plugins/current/ipam/>), отвечают только за поиск действительного IP-адреса для контейнера, тогда как другие, такие как Antrea или Calico, работают на более высоком уровне, делегируя выполнение низкоуровневых функций другим плагинам. Некоторые плагины CNI вообще не подключают модули Pod к сети, а скорее играют незначительную роль в более широком рабочем процессе «добавления контейнера в сеть». (В свете этого плагин IPAM служит отличным примером, помогающим понять идею.)

Имейте в виду, что любой плагин CNI, с которым вам доведется столкнуться, – это шестеренка в общем механизме подключения контейнера к сети. Кроме того, некоторые плагины CNI имеют смысл только в комплексе с другими плагинами.

Давайте вернемся к нашим предыдущим модулям Pod и к их основным требованиям к сети. Выше мы обсуждали способы управления правилами iptables для nftables, IPVS (IP virtual servers – виртуальные IP-серверы) и других реализаций сетевых прокси с помощью kube-проху. Мы также рассмотрели различные правила KUBE-SER для «маскарадинга» (masquerade) трафика, исходящего из контейнеров, чтобы он выглядел как исходящий трафик узла. Этот трафик пересылается работающему модулю Pod, который может находиться на другом узле в кластере.

kube-проху отлично подходит для маршрутизации сервисов, действующих в модулях Pod, и обычно является первой ступенью программно-определяемой сети, с которой взаимодействуют пользователи. Например, впервые запуская простое приложение Kubernetes и открывая к нему доступ через порт узла, вы получаете доступ к модулю Pod через правило маршрутизации, созданное прокси-сервером kube-проху, работающим на узлах Kubernetes. Однако kube-проху

не особенно полезен, если в кластере нет надежной сети, связывающей модули Pod, потому что его единственная задача – отобразить IP-адрес службы в IP-адрес модуля Pod. Если IP-адрес модуля Pod не маршрутизируется между двумя узлами, то никакой kube-проху не сможет сделать приложение доступным для конечного пользователя. Другими словами, балансировщик нагрузки настолько надежен, насколько надежен его самый медленный конечный пункт.

Проект kpng и будущее kube-проху

С развитием Kubernetes расширяется и ландшафт CNI и предпринимаются попытки реализовать функциональность маршрутизации службы kube-проху на уровне CNI. Это позволяет провайдерам CNI, таким как Antrea, Calico и Cilium, обеспечивать высокую производительность и расширенные возможности прокси-сервера Kubernetes (например, мониторинг и встроенную интеграцию с другими технологиями балансировки нагрузки).

Чтобы удовлетворить потребность в «подключаемом» сетевом прокси-сервере, который может взять на себя часть базовой логики Kubernetes и позволить провайдерам расширять другие части, был создан проект kpng (<https://github.com/kubernetes-sigs/kpng>). Он разрабатывается как новая альтернатива kube-проху. Это чрезвычайно модульная реализация, целиком и полностью находящаяся за пределами кодовой базы Kubernetes. Если вас интересуют сервисы балансировки нагрузки в Kubernetes, то этот проект послужит отличным примером, в котором можно покопаться и узнать больше, но имейте в виду, что на момент написания этой книги он еще не был готов к промышленному использованию.

Примером альтернативного сетевого прокси-сервера, предоставляемого CNI, который когда-нибудь можно будет полностью реализовать как расширение kpng, может служить такой проект, как прокси-сервер Antrea, который можно включать и отключать, в зависимости от предпочтений пользователя. Дополнительную информацию вы найдете по адресу <http://mng.bz/AxGQ>.

5.1 Зачем нужны программно-определяемые сети в Kubernetes

Главная проблема контейнерных сетей, часто насчитывающих сотни модулей Pod с экземплярами одного и того же сервиса, состоит в последовательной маршрутизации трафика в кластер и из него, чтобы этот трафик всегда попадал в нужное место, даже если модули Pod перемещаются между узлами. Это вторая очевидная операционная проблема, с которой сталкивается каждый, пытающийся внедрить

контейнерное решение, отличное от Kubernetes (например, на основе Docker). Для решения этой проблемы Kubernetes предоставляет два основных сетевых инструмента:

- *прокси-сервер сервисов* – обеспечивает балансировку нагрузки модулей Pod с постоянными IP-адресами и маршрутизирует объекты Kubernetes Service;
- *CNI* – гарантирует возрождение модулей Pod в плоской сети, к которой легко получить доступ из кластера.

В основе этого решения лежит Kubernetes Service с типом ClusterIP – объект, маршрутизируемый внутри кластера Kubernetes, но недоступный за его пределами. Это фундаментальный примитив, на основе которого можно строить другие сервисы. Это также простой способ организации взаимодействий приложений внутри кластера без прямой маршрутизации по IP-адресу модуля Pod (не забывайте, что IP-адреса модулей Pod могут меняться при перемещении с одного узла на другой).

Например, если создать три экземпляра одного и того же сервиса в кластере kind, то они получат три случайных IP-адреса в пространстве IP-адресов 10.96. Чтобы убедиться в этом, можно выполнить команду `kubectl create service clusterip my-service-1 --tcp="100:100"` три раза (разумеется, изменив имя сервиса `my-service-1`), а затем вывести список IP-адресов сервиса:

```
$ kubectl get svc -o wide
svc-1 ClusterIP 10.96.7.53      80/TCP 48s app=MyApp
svc-2 ClusterIP 10.96.152.223 80/TCP 33s app=MyApp
svc-3 ClusterIP 10.96.43.92   80/TCP 5s  app=MyApp
```

Аналогично для модулей Pod имеется своя единая сеть и подсеть. При создании новых модулей Pod им выделяются новые IP-адреса. В нашем кластере kind уже есть два работающих модуля CoreDNS, поэтому можно проверить их IP-адреса, чтобы убедиться в этом:

```
$ kubectl get pods -A -o wide | grep coredns
kube-system coredns-74ff55c5b-nlxrs 1/1 Running 0 4d16h 192.168.71.1
➡ calico-control-plane <none> <none>
kube-system coredns-74ff55c5b-t4p6s 1/1 Running 0 4d16h 192.168.71.3
➡ calico-control-plane <none> <none>
```

Мы только что увидели первые важные уроки Kubernetes SDN: IP-адреса назначаются объектам Pod и Service автоматически и находятся в разных IP-подсетях. Это верно почти для всех кластеров, встречающихся в реальном мире. На самом деле, если встретится кластер, в котором это *не* так, есть вероятность, что какой-то из механизмов Kubernetes был скомпрометирован. Это может быть способность kube-ргоху маршрутизировать трафик или способность узла маршрутизировать трафик между модулями Pod.

Диапазоны IP-адресов для объектов Pod и Service определяет плоскость управления Kubernetes

В Kubernetes бытует распространенное заблуждение, что ответственность за распределение IP-адресов между объектами Pod и Service несут провайдеры CNI. На самом деле, создавая новый объект Service с типом ClusterIP, плоскость управления Kubernetes создает новый IP-адрес из диапазона CIDR, заданного при запуске в виде параметра командной строки (например, `--service-cluster-ip-range`), который используется с параметром `--allocate-node-cidrs`. Провайдеры CNI часто полагаются на диапазоны CIDR узлов, которые выделяются сервером API, если они указаны. Таким образом, интерфейс CNI и сетевой прокси действуют на строго локализованном уровне, выдавая директивы конфигурации кластера, определяемые плоскостью управления Kubernetes.

5.2 Реализация Kubernetes SDN на стороне сервиса: kube-proxy

Существует три основных типа объектов Service: ClusterIP, NodePort и LoadBalancer. Они определяют модули Pod для подключения с помощью *меток*. Например, в предыдущем кластере мы имели объекты Service с типом ClusterIP в подсети 10, направляющие трафик в модули Pod в подсети 192. Как трафик, адресованный на IP-адрес сервиса, пересылается в другую подсеть? Эту функцию выполняет kube-proxy (прокси-сервер сети или сервиса Kubernetes).

В предыдущем примере мы трижды выполнили команду `kubectl create service my-service-1 --tcp="100:100"` и получили три экземпляра сервиса типа ClusterIP. Если бы эти сервисы имели тип NodePort, то они могли бы получить IP-адреса любых узлов в нашем кластере. Если бы они имели тип LoadBalancer, то наше облако (если бы кластер размещался в облаке) предоставило бы внешний IP-адрес, например 35.1.2.3, доступный из интернета или из сети за пределами диапазона IP-адресов модулей Pod, узлов или сервисов, в зависимости от облачного провайдера.

kube-proxy – это действительно прокси-сервер?

На заре Kubernetes компонент kube-proxy был реализован как процедура на Golang, т. е. как процесс пространства пользователя, обслуживающий трафик. Создание прокси-сервера Kubernetes на основе iptables (а затем прокси-сервера IPVS) и прокси-сервера ядра Windows привело к увеличению масштабируемости и эффективности kube-proxy.

В настоящее время еще продолжают существовать подобные варианты проксирования в пространстве пользователя, но с каждым годом их остается все меньше. Например, VMware Tanzu Kubernetes Grid использует

проксирование в пространстве пользователя для поддержки кластеров в Windows, потому что не может положиться на проксирование в пространстве ядра. Это связано с отличиями от архитектур на основе виртуального коммутатора Open vSwitch (OVS). В любом случае kube-proxy использует другие инструменты проксирования конечных точек Kubernetes, но сам не является прокси-сервером в традиционном смысле.

На рис. 5.1 показано течение трафика от балансировщика нагрузки LoadBalancer в кластер Kubernetes. Здесь видно, что:

- kube-proxy использует технологию низкоуровневой маршрутизации, такую как iptables или IPVS, для передачи трафика от сервисов в модули Pod и обратно;
- сервис типа LoadBalancer получает трафик с внешнего IP-адреса и пересылает его на внутренний IP-адрес.

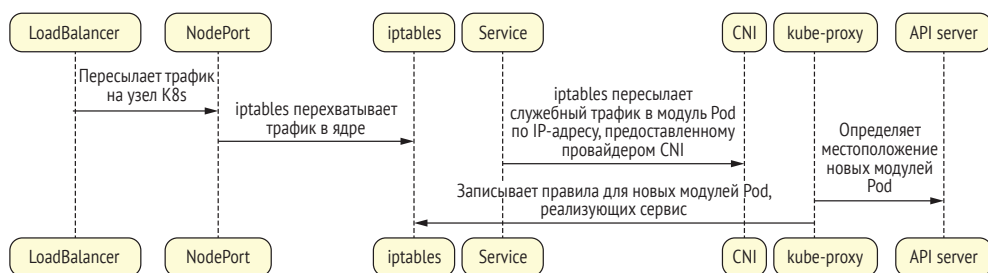


Рис. 5.1 Течение трафика от LoadBalancer в кластер Kubernetes

Объекты Service с типами NodePort и ClusterIP

NodePort – это тип объектов Service в Kubernetes, доступных на всех портах за пределами внутренней сети модуля Pod. Они позволяют построить самый простой балансировщик нагрузки.

Например, представьте, что имеется веб-приложение, объявленное как сервис ClusterIP, скажем 100.1.2.3:443. Чтобы получить доступ к этому приложению из-за пределов кластера, можно организовать перенаправление в эту службу через NodePort. Значение NodePort выбирается произвольно; например, это может быть число 50491. Соответственно, ваше веб-приложение будет доступно по адресам node_ip_1:50491, node_ip_2:50491, node_ip_3:50491 и т. д.

Существуют и другие, более оптимальные способы настройки маршрутизации, например путем аннотирования сервисов с помощью аннотации *externalTrafficPolicy*, но этот подход применим не во всех операционных системах и типах облаков. Обязательно изучите все подробности, если решите увлечься маршрутизацией сервисов.

Сервисы NodePort основаны на сервисах *ClusterIP*. Сервисы ClusterIP имеют внутренний IP-адрес, не пересекающийся (обычно) с сетью модуля Pod, которая синхронизирована с сервером API.

Пример набора правил iptables, создаваемых kube-proxy

Если вам интересно увидеть набор правил iptables из реального кластера с подробным описанием, загляните в файл iptables-save-calico.md по адресу <http://mng.bz/enV9>. Мы подготовили его, чтобы показать правила iptables, создаваемые типичным кластером Kubernetes.

В этом файле определяются правила для трех основных таблиц iptables, самой важной из которых в Kubernetes является таблица NAT. Именно она подвержена наиболее частым изменениям в больших кластерах с динамично изменяющимся ландшафтом сервисов. Как упоминается в других частях этой книги, существуют другие варианты организации проксирования с kube-proxy, но чаще других используется сочетание kube-proxy с iptables.

5.2.1 Плоскость данных в kube-proxy

Прокси-сервер kube-proxy должен иметь возможность обрабатывать входящий и исходящий TCP-трафик модулей Pod с сервисами. Каждый IP-пакет имеет определенные свойства, включая IP-адрес отправителя и получателя. В сложной сети адреса могут меняться, пока пакет пересекает последовательность маршрутизаторов, и узел Kubernetes тоже следует рассматривать как один из маршрутизаторов (из-за kube-proxy). В общем случае изменение IP-адресов в пакете маршрутизатором называется *трансляцией сетевых адресов* (Network Address Translation, NAT) и считается фундаментальным аспектом практически любой сетевой архитектуры на том или ином уровне. Под названиями *SNAT* и *DNAT* подразумеваются трансляция IP-адресов отправителя (source) и получателя (destination) соответственно.

Плоскость данных в kube-proxy может решать эту задачу разными способами, которые определяются конфигурацией. Углубившись в детали, можно обнаружить, что сам kube-proxy реализован в виде двух отдельных компонентов управления: `server_windows.go` и `server_others.go` (оба можно найти здесь: <http://mng.bz/EWxl>). Компонент `server_windows.go` компилируется в двоичный файл `kube-proxy.exe` и в процессе работы использует низкоуровневые Windows API (такие как команда `netsh` для проксирования в пространстве пользователя и интерфейсы контейнеризации `hcsshim` и `HCN` [<http://mng.bz/N6x2>] для проксирования на уровне ядра Windows).

Но чаще kube-proxy запускается в Linux. В этой ОС используется другая программа (которая называется `kube-proxy`). Она не содержит кода, использующего Windows API. В Linux для проксирования обычно используется iptables. В кластерах `kind` прокси kube-proxy по умолчанию использует iptables. В этом легко убедиться, если получить конфигурацию kube-proxy командой `kubectl edit cm kube-proxy -n kube-system` и посмотреть значение в поле `mode`:

- `ipvs` – для описания правил маршрутизации сервисов используется балансировщик нагрузки ядра (Linux);
- `iptables` – для описания правил маршрутизации сервисов используется брандмауэр ядра (Linux);
- `userspace` – запускается процесс командой `go func`, который вручную проксирует трафик в модуль Pod (Linux);
- ядро Windows использует для балансировки нагрузки интерфейсы `hcsshim` и HCN, несовместимые с реализациями CNI на основе OVS, но совместимые с другими CNI, такими как Calico (аналог варианта `userspace` в Linux);
- для проксирования в пространстве пользователя в Windows также используется `netsh`. Этот вариант может пригодиться, когда по какой-то причине нельзя использовать обычные Windows API. Обратите внимание, что если в Windows установлено расширение OVS, то может потребоваться использовать проксирование в пространстве пользователя, потому что HCN API в ядре действуют совершенно иначе.

ПРИМЕЧАНИЕ На протяжении всей книги мы будем упоминать такие понятия, как *информеры*, *контроллеры* и *операторы*, и отмечать, что они не всегда одинаково реагируют на изменения в конфигурации. Сетевой прокси реализован на основе контроллера Kubernetes, однако он не реагирует на изменения в конфигурации. Поэтому, чтобы поэкспериментировать со способами балансировки нагрузки, нужно сначала отредактировать `configMap` для сетевого прокси, а затем перезапустить его `DaemonSet`. (Можно просто остановить модуль Pod в своем `DaemonSet`, а затем понаблюдать за журналом модулей, чтобы увидеть, как он запускается повторно. После этого `kube-proxy` должен запуститься в новом режиме.)

Однако `kube-proxy` – это лишь один из компонентов, управляющих трафиком в Kubernetes SDN. Маршрутизацию трафика в Kubernetes можно представить в виде трех отдельных уровней:

- *внешние балансировщики нагрузки или входные/шлюзовые маршрутизаторы* – направляют трафик в кластер Kubernetes;
- *kube-proxy* – управляет маршрутизацией трафика между сервисами в модулях Pod. Выше уже отмечалось, что термин *прокси* (`proxy`) несколько неточен, потому что `kube-proxy` просто управляет статическими правилами маршрутизации, которые реализуются ядром или другой технологией в плоскости данных, такой как `iptables`;
- *провайдеры CNI* – направляют трафик в модули Pod и от них, независимо от способа обращения, – через конечную точку службы или напрямую.

В конечном счете провайдер CNI (например, `kube-proxy`) тоже настраивает некоторый механизм (например, таблицу маршрутизации)

или переключатель OVS, чтобы гарантировать передачу в модули Pod трафика, пересылаемого между узлами или из внешнего мира. Если вам интересно, почему используются две разные технологии – kube-проху и CNI, – то вы не одиноки! Многие провайдеры CNI пытаются реализовать внутри полноценный kube-проху, чтобы дать возможность отказаться от kube-проху, входящего в состав Kubernetes.

5.2.2 Подробнее о NodePort

В первой части этой главы мы рассмотрели сервисы ClusterIP, поэтому для полноты картины нам нужно поближе познакомиться с сервисами NodePort. Для этого создадим новый сервис Kubernetes и на его примере посмотрим, насколько легко добавлять и изменять правила балансировки нагрузки. В этом примере мы создадим сервис типа NodePort, указывающий на контейнер CoreDNS, который выполняется внутри модуля Pod в нашем кластере. Для простоты можно получить конфигурацию командой `kubectl get svc -o yaml kube-dns -n kube-system` и затем изменить тип ClusterIP сервиса на NodePort:

```
# сохраните следующий код в файле my-nodeport.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/port: "9153"
    prometheus.io/scrape: "true"
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: CoreDNS
name: kube-dns-2
namespace: kube-system
spec:
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: dns
    port: 53
    protocol: UDP
    targetPort: 53
  - name: dns-tcp
    port: 53
    protocol: TCP
    targetPort: 53
  - name: metrics
    port: 9153
    protocol: TCP
    targetPort: 9153
  selector:
```

← Дайте сервису имя kube-dns-2, чтобы отличить его от уже существующего сервиса kube-dns

```

k8s-app: kube-dns
sessionAffinity: None
type: NodePort ← Измените тип сервиса на NodePort
status:
  loadBalancer: {}

```

Если теперь выполнить команду `kubectl create -f my-nodeport.yaml`, то можно увидеть, что для нового сервиса был зарезервирован некоторый случайный порт. Теперь этот сервис будет пересылать трафик в CoreDNS:

```

kubectl get pods -o wide -A
kube-system  kube-dns  ClusterIP  10.96.0.10
              53/UDP,53/TCP,9153/TCP  k8s-app=kube-dns
kube-system  kube-dns-2  NodePort    10.96.80.7
              53:30357/UDP,53:30357/TCP,9153:31588/TCP
              2m33s  k8s-app=kube-dns ←

```

Отображает произвольно
выбранные порты 30357
и 31588 в порт 53

Произвольно выбранные порты 30357 и 31588 отображаются в порт 53 наших модулей Pod со службой DNS и открыты на всех узлах нашего кластера, потому что на всех узлах работает kube-проxy. Эти произвольные порты не были зарезервированы ранее, когда мы создавали сервисы ClusterIP.

Желающие могут попробовать выполнить команду `iptables-save` на узлах кластера `kind` и попытаться понять, какие правила добавил kube-проxy для поддержки вновь созданных сервисов. (Если вас интересовали сервисы NodePort, то вам определенно понравится наша последняя глава, где рассказывается, как устанавливать и тестировать приложения Kubernetes локально. Там мы создадим несколько сервисов для тестирования известного приложения Guestbook в Kubernetes.)

Теперь, освежив в памяти, как сервисы устанавливают правила маршрутизации между внутренними портами модулей Pod и внешним миром, перейдем к провайдерам CNI. Они образуют уровень управления, лежащий ниже прокси в сетевом стеке Kubernetes SDN. На самом деле наш сервис просто реализует маршрутизацию трафика с адреса 10.96.80.7 в модули Pod, находящиеся внутри кластера. Но как эти модули подключаются к действительному IP-адресу и как получают этот трафик? Ответ: с помощью интерфейса CNI.

5.3 Провайдеры CNI

Провайдеры CNI (CNI providers) реализуют спецификацию CNI (<http://mng.bz/RENK>), определяющую контракт, который позволяет окружениям выполнения контейнеров запрашивать рабочий IP-адрес для процесса при запуске. Они также добавляют другие уникальные возможности, выходящие за рамки этой спецификации (такие как реализация сетевых политик или интеграция мониторинга сети). На-

пример, пользователи VMware могут свободно использовать Antrea в качестве прокси CNI и подключать его к таким окружениям, как платформа VMware NSX, для мониторинга контейнеров в реальном времени с возможностью журналирования, которая поддерживается некоторыми современными провайдерами CNI с открытым исходным кодом. Теоретически от провайдера CNI требуется маршрутизировать только трафик в модулях Pod, но многие из них предоставляют дополнительные функции. Вот краткий перечень основных локальных провайдеров CNI:

- *Calico* – провайдер CNI на основе протокола пограничного шлюза (Border Gateway Protocol, BGP), который реализует плоскость данных, создавая новые правила маршрутизации BGP. Calico дополнительно поддерживает технологии маршрутизации XDP, NAND и VXLAN (например, в Windows нередко запускают Calico в режиме VXLAN). Может заменить kube-проxy, используя технологию, аналогичную Cilium;
- *Antrea* – провайдер CNI плоскости данных OVS, использующий мост для маршрутизации всего трафика модулей Pod. Подобно Calico, поддерживает множество дополнительных функций маршрутизации и способен заменить kube-проxy (AntreaProxy);
- *Flannel* – провайдер CNI на основе моста; в настоящее время вышел из употребления. Это был один из первых провайдеров CNI для промышленных кластеров Kubernetes;
- *Google, EC2 и NCP* – облачные провайдеры CNI, использующие патентованное программное обеспечение для маршрутизации трафика с применением облачных технологий. Например, они могут создавать правила, направляющие трафик напрямую между контейнерами, в обход сетевых путей узлов;
- *Cilium* – провайдер CNI на основе технологии XDP, использующий современные Linux API без привлечения механизмов ядра для управления трафиком. В некоторых случаях обеспечивает более быструю и безопасную IP-связь между контейнерами. Cilium предлагает дополнительные средства управления маршрутизацией данных, позволяющие заменить kube-проxy;
- *KindNet* – плагин CNI, который по умолчанию используется в кластерах kind. Предназначен исключительно для использования в простых кластерах с одной подсетью.

Существует множество других реализаций сетевого интерфейса CNI, как с открытым исходным кодом, так и проприетарных для различных облачных окружений, таких как VMware, Azure, EKS и т. д. Проприетарные реализации CNI работают только внутри инфраструктуры соответствующего производителя и, соответственно, менее переносимы, зато часто более эффективны или лучше интегрированы с облачными функциями. Некоторые CNI, такие как Calico и Antrea, имеют зависящие и не зависящие от производителя функции (например, интеграция с Tigera или NSX).

5.4 Два плагина CNI: Calico и Antrea

На рис. 5.2 показано, как работают CNI-плагины Calico и Antrea. Оба достигают одного и того же конечного состояния, используя набор правил маршрутизации и технологий с открытым исходным кодом. Интерфейс CNI определяет несколько основных функциональных аспектов сетевых решений для контейнеров, и все плагины CNI (например, на основе BGP и OVS) реализуют их по-разному. Как показано на рис. 5.2, разные CNI используют разные базовые технологические стеки.

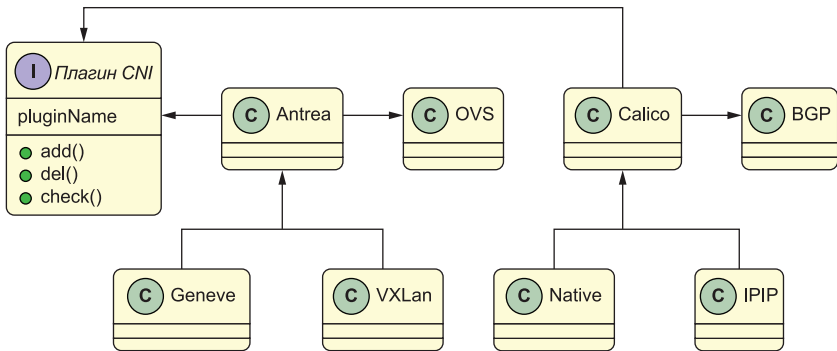


Рис. 5.2 Работа CNI-плагинов Calico и Antrea

Использование kube-proxy – обязательное требование?

Мы говорим об использовании kube-проху как об обязательном требовании, но все больше провайдеров сетевых услуг начинает предлагать такие технологии, как расширенный пакетный фильтр Беркли (Extended Berkeley Packet Filter, eBPF), поддерживаемый в Cilium CNI, или прокси-сервер OVS – в Antrea CNI, избавляющие от необходимости использовать kube-проху. По большей части они повторяют логику kube-проху, но при этом используют другую плоскость данных. Однако на момент публикации этой книги большинство кластеров продолжало использовать традиционные iptables или механизм проксирования в ядре Windows. По этой причине мы описываем kube-проху как неотъемлемую часть современного кластера Kubernetes. Но на горизонте уже видны интересные альтернативы!

5.4.1 Архитектура плагинов CNI

Оба плагина, Calico и Antrea, имеют схожую архитектуру: объект DaemonSet и координирующий контейнер. Их установка CNI выполняется в четыре этапа (обычно полностью автоматизированные провайдером CNI, благодаря чему в простых кластерах Linux ее можно выполнить одной командой).

- 1 Установка kube-проху, потому что контроллеру координации провайдера CNI почти наверняка потребуется возможность запрашивать сервер Kubernetes API. Этот шаг обычно выполняется автоматически установщиком Kubernetes.
- 2 Установка на узле скомпилированной программы CNI (обычно в таком каталоге, как `/opt/cni/bin`), которую может вызвать среда выполнения контейнеров для создания модуля Pod с IP-адресом, предоставленным интерфейсом CNI.
- 3 Развертывание объекта DaemonSet в кластере, в ходе которого в один контейнер устанавливаются сетевые примитивы для резидентного узла. Этот объект DaemonSet выполняет предыдущий этап установки для своего хоста при запуске.
- 4 Развертывание в кластере координирующего контейнера, который агрегирует или проксирует метаданные из Kubernetes, например объединяет информацию NetworkPolicy в одном месте, чтобы упростить ее использование модулями Pod, создаваемыми объектом DaemonSet.

Для плагинов CNI нет какой-то обязательной архитектуры, но многие используют устоявшийся шаблон на основе DaemonSet и контроллера. В Kubernetes такой шаблон хорошо подходит почти для любого агентно-ориентированного процесса, предназначенного для интеграции с Kubernetes API.

ПРИМЕЧАНИЕ Провайдеры CNI предоставляют IP-адреса модулям Pod, но многие предположения о работе этого процесса изначально базировались на особенностях ОС Linux. Поэтому, знакомясь с Calico и Antrea, учитывайте, что поведение этих интерфейсов CNI различается в разных операционных системах. Например, в Windows оба плагина, Calico и Antrea, обычно запускаются не как модули Pod, а как службы Windows с использованием таких инструментов, как `nssm`. В настоящее время наиболее проверенными плагинами CNI с открытым исходным кодом, поддерживающими и Linux, и Windows, являются Calico и Antrea, но есть и многие другие.

Спецификация CNI реализуется двоичной программой, установленной агентом. Она, кроме всего прочего, реализует три основные операции CNI: ADD, DELETE и CHECK, которые вызываются, когда `containerd` запускает новый модуль Pod или удаляет его. Эти операции:

- добавляют контейнер в сеть;
- удаляют контейнер из сети;
- проверяют правильность установки контейнера.

5.4.2 Давайте поэкспериментируем с некоторыми CNI

Наконец-то настал момент, когда можно попрактиковаться! Начнем с установки провайдера Calico CNI в наш кластер `kind`. Calico исполь-

зует маршрутизацию уровня 3 (в отличие от моста, который является технологией уровня 2) для широковещательной рассылки маршрутов модулям Pod в кластере. Конечные пользователи обычно не замечают этой разницы, но для администраторов она важна, потому что некоторым администраторам может понадобиться использовать концепции уровня 3 (например, пиринг BGP) или уровня 2 (например, мониторинг трафика на основе OVS) для более широких целей проектирования инфраструктуры кластера:

- **BGP** – протокол пограничного шлюза (Border Gateway Protocol), технология маршрутизации уровня 3, широко используемая в интернете;
- **OVS** – Open vSwitch, прикладной программный интерфейс (API) на основе ядра Linux для создания внутри ОС программного коммутатора виртуальных IP-адресов.

Первый шаг на пути к созданию нашего кластера `kind` – отключение CNI по умолчанию. Затем мы воссоздадим его из спецификации YAML. Например:

```
$ cat << EOF > kind-Calico-conf.yaml
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha4
networking:
  disableDefaultCNI: true      ← Отключает kind-net CNI
  podSubnet: 192.168.0.0/16   ← Разделяет подсеть 192.168 так, чтобы она не пересекалась с нашей служебной подсетью
nodes: ← Добавляет второй узел в кластер
- role: control-plane
- role: worker
EOF
$ kind create cluster --name=calico --config=./kind-Calico-conf.yaml
```

`kind-net CNI` – это минимальный CNI, работающий только в кластере с одним узлом. Мы отключаем его, чтобы использовать настоящего провайдера CNI. Все наши модули Pod будут находиться в большой подсети 192.168. Calico делит ее для каждого узла, и она не должна пересекаться с нашей служебной подсетью. Кроме того, наличие второго узла в нашем кластере поможет нам понять, как Calico отделяет локальный трафик от трафика, предназначенного для другого узла.

Процесс настройки кластера `kind` с использованием настоящего плагина CNI не сильно отличается от того, что мы уже видели. После запуска кластера стоит приостановиться ненадолго, чтобы посмотреть, что происходит, когда CNI в модуле Pod еще недоступен. Эти модули недоступны для планирования и не определены в каталоге `kubelet/manifests`, в чем можно убедиться, выполнив следующие команды `kubectl`:

```
$ kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-66bff467f8-86mgh	0/1	Pending	0	7m47s
kube-system	coredns-66bff467f8-nfzhz	0/1	Pending	0	7m47s


```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
Calico-control-plane	NotReady	master	2m4s	v1.18.2
Calico-worker	NotReady	<none>	85s	v1.18.2

5.4.3 Установка провайдера CNI Calico

На этом этапе наш модуль Pod CoreDNS не сможет запуститься, потому что планировщик Kubernetes видит все узлы в состоянии NotReady, как показывают предыдущие команды. Это обусловлено тем, что провайдер CNI еще не установлен. Интерфейсы CNI настраиваются после того, как контейнер CNI запишет файл /etc/cni/net.d в локальную файловую систему, где работает агент kubelet. Чтобы запустить наш кластер, установим Calico:

```
$ wget https://docs.projectcalico.org/manifests/Calico.yaml
$ kubelet create -f Calico.yaml
```

Безопасность Kubernetes имеет значение

Эта книга сосредоточена на изучении внутренних механизмов Kubernetes, но мы почти ничего не предпринимаем для защиты кластера. Предыдущая команда, например, извлекает файл манифеста из интернета и устанавливает в кластер несколько контейнеров. Не запускайте эти команды в действующем промышленном кластере, если не до конца понимаете их последствия!

В главах 13 и 14 вы найдете краткое руководство по безопасности модулей Pod и узлов. Кроме того, заинтересованным в безопасности приложений рекомендуем обратить внимание на такие проекты, как <https://sigstore.dev/> и <https://github.com/bitnami-labs/sealed-secrets>, созданные для решения различных проблем безопасности двоичных файлов, артефактов, манифестов и даже секретов Kubernetes. За дополнительной информацией об общих концепциях безопасности в Kubernetes обращайтесь по адресу <https://kubernetes.io/docs/concepts/security/> или подписывайтесь на список рассылки по безопасности Kubernetes (<http://mng.bz/QWz1>).

На предыдущем шаге создаются контейнеры двух типов: модуль Pod Calico-node на каждом узле и модуль Pod Calico-kube-controllers на некоторых произвольно выбранных узлах. Как только эти контейнеры запустятся, узлы должны перейти в состояние Ready, и вы также увидите, что заработал модуль Pod CoreDNS:

```
$ kubectl get pods --all-namespaces
```

NAMESPACE	NAME
kube-system	Calico-kube-cntrlrs-57-m5
kube-system	Calico-node-4mbc5
kube-system	Calico-node-gpvxm

Координирует контейнеры узлов Calico

Настраивает различные маршруты BGP и IP для каждого узла

kube-system	coredns-66bff467f8-98t8j
kube-system	coredns-66bff467f8-m7lj5
kube-system	etcd-Calico-control-plane
kube-system	kube-apiserver-Calico-control-plane
kube-system	kube-controller-mgr
kube-system	kube-proxy-8q5zq
kube-system	kube-proxy-zgrjf
kube-system	kube-scheduler-Calico-control-plane
local-path-storage	local-path-provisioner-b5-fsr

В этом примере контейнер контроллера координирует контейнеры узлов Calico. Каждый контейнер узла Calico устанавливает различные маршруты BGP и IP для всех контейнеров, работающих на данном узле. В этом примере их два – по числу узлов.

Оба плагина, Calico и Antrea, монтируют тома типа *hostPath* в каталог `/etc/cni/net.d/`, к которому затем обращаются двоичные программы. Агент `kubelet` использует эти двоичные программы для вызова CNI API, чтобы получить IP-адрес для нового модуля Pod, и, соответственно, его можно рассматривать как *механизм установки* провайдера CNI хоста. В общем случае создание томов типа *hostPath* считается антишаблоном, исключением является настройка низкоуровневой функциональности ОС, такой как CNI.

На рис. 5.2 мы рассмотрели возможности `DaemonSet` как интерфейса, реализованного в обоих плагинах, Calico и Antrea. Теперь давайте посмотрим, что создает плагин Calico, выполнив команду `kubectl get ds -n kube-system`. Мы увидим, что Calico определяет объект `DaemonSet` для запуска модуля Pod с CNI на всех узлах. Ниже, запустив Antrea, мы увидим, что этот плагин определяет аналогичный объект `DaemonSet`.

Поскольку плагины CNI для Linux обычно помещают двоичный файл CNI в системный путь поиска программ хоста, их можно рассматривать как реализацию метода `MountCniBinary`, который не является частью формального интерфейса CNI, но имеется почти во всех плагинах CNI, встречающихся в дикой природе.

Итак, теперь у нас есть интерфейс CNI. Давайте посмотрим, что было создано плагином Calico, выполнив команду `docker exec`, чтобы получить доступ к нашим узлам и исследовать их. После запуска `docker exec -t -i <узел_кластера_kind> /bin/bash` можно посмотреть, какие маршруты были созданы плагином Calico. Например:

```
root@Calico-control-plane:/# ip route
default via 172.18.0.1 dev eth0
172.18.0.0/16 dev eth0 proto kernel scope
    link src 172.18.0.3
192.168.9.128/26 via 172.18.0.2 dev tunl0
    proto bird onlink
blackhole 192.168.71.0/26 proto bird
192.168.71.1 dev cali38312ba5f3c scope link
192.168.71.2 dev califcbd6ecdce5 scope link
```

Трафик, предназначенный для другого узла, идентифицируется по его подсети

Трафик, не соответствующий этому узлу, но относящийся к подсети 71, будет отбрасываться

Здесь можно видеть два IP-адреса: 192.168.71.1 и 71.2. Они связаны с двумя устройствами, имена которых начинаются с префикса *cali*, созданными нашими контейнерами Calico-node. Как работают эти устройства? Чтобы увидеть, как они определяются, можно выполнив команду `ip a`:

```
root@Calico-control-plane:/# ip a | grep califc
5: califcbd6ecdce5@if4: <BROADCAST,MULTICAST,UP,LOWER_UP>
➔ mtu 1440 qdisc noqueue state UP group default
```

Теперь мы видим, что узел имеет интерфейс, созданный для модулей Pod, имеющих отношение к Calico, с узнаваемым именем. В следующем примере:

```
root@Calico-control-plane:/# apt-get update -y;
➔ apt-get install tcpdump ← Установить tcpdump в контейнер
root@Calico-control-plane:/# tcpdump -s 0
➔ -i cali38312ba5f3c -v | grep 192 ← Выполнить tcpdump для устройства Calico
tcpdump: listening on cali38312ba5f3c, link-type EN10MB (Ethernet),
➔ capture size 262144 bytes
```

```
10.96.0.1.443 > 192.168.71.1.59186: Flags [P.],
    cksum 0x14d2 (incorrect -> 0x7189),
    seq 520038628:520039301, ack 2015131286, win 502,
    options [nop,nop,TS val 1110809235 ecr 1170831911],
    length 673
192.168.71.1.59186 > 10.96.0.1.443: Flags [.],
    cksum 0x1231 (incorrect -> 0x9f10),
    ack 673, win 502,
    options [nop,nop,TS val 1170833141 ecr 1110809235],
    length 0
10.96.0.1.443 > 192.168.71.1.59186:
    Flags [P.], cksum 0x149c (incorrect -> 0xa745),
    seq 673:1292, ack 1, win 502,
    options [nop,nop,TS val 1110809914 ecr 1170833141],
    length 619
192.168.71.1.59186 > 10.96.0.1.443:
    Flags [.], cksum 0x1231 (incorrect -> 0x9757),
    ack 1292, win 502,
    options [nop,nop,TS val 1170833820 ecr 1110809914],
    length 0
192.168.71.1.59186 > 10.96.0.1.443:
    Flags [P.], cksum 0x1254 (incorrect -> 0x362c),
    seq 1:36, ack 1292, win 502,
    options [nop,nop,TS val 1170833820 ecr 1110809914],
    length 35
10.96.0.1.443 > 192.168.71.1.59186:
    Flags [.], cksum 0x1231 (incorrect -> 0x9734),
    ack 36, win 502, options [nop,nop,TS val 1110809914
    ecr 1170833820],
    length 0
```

можно видеть входящий трафик на IP-адрес 71.1 из подсети 10.96. На самом деле это подсеть нашего сервиса CoreDNS, через который осуществляется связь с нашими контейнерами DNS посредством CNI. Предыдущее устройство `cali3831...` напрямую подключено (как и любое другое устройство) через кабель Ethernet (виртуальный) к нашему узлу. Это известно как пара *veth*, в которой один конец виртуального Ethernet-кабеля (с именем `cali3831`) подключен к нашим контейнерам, а другой – напрямую к `kubelet`. Это означает, что любой легко сможет получить доступ к этому устройству из `kubelet`.

Теперь давайте вернемся назад и рассмотрим таблицу IP-маршрутов. Записи `dev` теперь выглядят более понятными. Они соответствуют маршрутам подключения к нашим контейнерам. А как же «черная дыра» `blackhole` и маршруты `192.168.9.128/26`? Эти маршруты соответствуют:

- контейнерам, принадлежащие другому узлу (маршрут `192.168.9.128/26`);
- контейнерам, вообще не принадлежащим ни одному узлу (маршрут «черной дыры» `blackhole`).

Это работа протокола BGP. Каждый узел в кластере, где выполняется демон `Calico-node`, имеет диапазон IP-адресов, ведущих к нему. С появлением новых узлов в таблицу добавляются новые IP-маршруты. Выполнив команду `kubectrl scale deployment coredns -n kube-system --replicas=6`, вы обнаружите, что все IP-адреса относятся к одной из двух подсетей:

- некоторые модули *Pod* находятся в подсети `192.168.9`; они соответствуют одному из наших узлов;
- другие модули *Pod* находятся в подсети `192.168.71`; они соответствуют другому узлу.

Чем больше узлов в кластере, тем больше подсетей в нем будет. Каждый узел имеет свой диапазон IP-адресов, и провайдер CNI использует этот диапазон для распределения IP-адресов между модулями *Pod* на данном узле, чтобы избежать конфликтов между модулями *Pod* на разных узлах. Кроме того, такой подход является оптимизацией производительности, избавляя от необходимости глобальной координации пространства IP-адресов модулей *Pod*. Таким образом, `Calico` автоматически управляет диапазонами IP-адресов, выделяя пулы для отдельных узлов и координируя эти пулы с таблицами маршрутов в ядре.

5.4.4 Организация сети в Kubernetes с OVS и Antrea

Обычному пользователю может показаться, что плагины `Antrea` и `Calico` действуют совершенно одинаково: маршрутизируют трафик между контейнерами в кластере с несколькими узлами. В общем и целом так и есть, но, заглянув под капот, можно обнаружить множество тонких отличий.

Для расширения своих возможностей Antrea использует OVS. В отличие от BGP, Antrea не использует IP-адреса для прямой маршрутизации между узлами, как мы видели это на примере Calico, а создает мост, работающий локально на узле Kubernetes, используя OVS. OVS – это в буквальном смысле программно-определяемый коммутатор (подобный аппаратным коммутаторам, которые можно купить в любом компьютерном магазине). Кроме того, OVS служит интерфейсом между модулями Pod и остальным миром.

Обсуждение плюсов и минусов маршрутизации на основе моста (уровень 2) и IP-адресов (уровень 3) выходит за рамки этой книги и является предметом горячих споров как среди ученых, так и среди компаний-разработчиков программного обеспечения. Мы же просто отметим, что это разные технологии, работающие достаточно хорошо и легко масштабируемые для обработки тысяч модулей Pod.

Давайте снова создадим кластер kind, но на этот раз в роли провайдера CNI используем Antrea. Сначала удалите предыдущий кластер командой `kind delete cluster --name=calico`, а затем воссоздайте его, как показано ниже:

```
$ cat << EOF > kind-Antrea-conf.yaml
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
networking:
  disableDefaultCNI: true
  podSubnet: 192.168.0.0/16
nodes:
- role: control-plane
- role: worker
EOF
$ kind create cluster --name=Calico --config=./kind-Antrea-conf.yaml
```

Затем, когда кластер будет создан, выполните команду:

```
kubectl apply -f https://github.com/vmware-tanzu/Antrea/
➔ releases/download/v0.8.0/Antrea.yml -n kube-system
```

Теперь запустим `docker exec` и исследуем IP-маршруты. На этот раз, как можно видеть ниже, было создано несколько разных интерфейсов. Обратите внимание, что здесь мы опустили интерфейс `tap0`, который присутствует в обоих CNI. Это сетевой интерфейс, через который течет инкапсулированный трафик между узлами.

Интересно отметить, что если выполнить команду `ip route`, то она не отметит появления новых маршрутов для вновь запускаемых модулей Pod. Это связано с тем, что OVS использует мост, и поэтому все виртуальные кабели Ethernet подключены непосредственно к локальному экземпляру OVS. Выполнив следующую команду, можно увидеть, что в Antrea используется практически та же логика организации подсети, что мы видели в Calico:

```

root@Antrea-control-plane:/# ip route
172.18.0.0/16 dev eth0 proto kernel scope link src 172.18.0.3
192.168.0.0/24 dev Antrea-gw0 proto kernel scope link
    src 192.168.0.1
192.168.1.0/24 via 192.168.1.1 dev
    Antrea-gw0 onlink

```

Определяет трафик, предназначенный для локальной подсети, суффиксом 0.0

Шлюз Antrea управляет трафиком, направляемым в другую подсеть с суффиксом 1.0

Чтобы подтвердить это, выполним команду `ip a`. Она покажет все IP-адреса, известные нашей машине:

```

$ docker exec -t -i ba133 /bin/bash
root@Antrea-control-plane:/# ip a
# ip a
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state
    DOWN group default qlen 1000
    link/ether 2e:24:a8:d8:a3:50 brd ff:ff:ff:ff:ff:ff
4: genev_sys_6081: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65000 qdisc
    noqueue master ovs-system state
    UNKNOWN group default qlen 1000
    link/ether 76:82:e1:8b:d4:86 brd ff:ff:ff:ff:ff:ff
5: Antrea-gw0:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state
    UNKNOWN group default qlen 1000
    link/ether 02:09:36:d3:cf:a4 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.1/24 brd 192.168.0.255 scope global Antrea-gw0
        valid_lft forever preferred_lft forever

```

Интересно отметить, что, выполнив команду `ip a`, можно увидеть несколько незнакомых устройств:

- `genev_sys_6081` – интерфейс для протокола туннелирования Genev, который использует Antrea;
- `ovs-system` – интерфейс OVS;
- `Antrea-gw0` – интерфейс Antrea, через который отправляется трафик в модули Pod.

В отличие от Calico, Antrea пересылает трафик на IP-адрес шлюза, находящийся в подсети модулей Pod, используя диапазон адресов `podCIDR` кластера. Соответственно, алгоритм, используемый плагином Antrea для назначения IP-адреса модулям Pod на данном узле, выглядит примерно так:

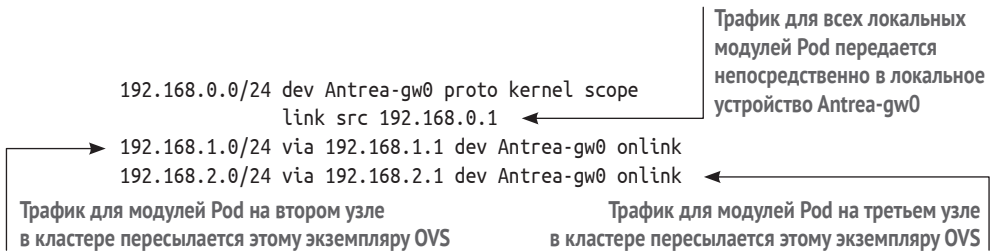
- выделить каждому узлу подсеть IP-адресов модулей Pod;
- выделить первый IP-адрес в подсети для коммутатора OVS на данном узле;
- распределить оставшиеся свободные IP-адреса в подсети между модулями Pod.

Таблица маршрутизации в таком кластере соответствует хронологическому порядку подключения узлов. Обратите внимание, что каждый узел получает трафик на IP-адрес `x.y.z.1` (первый Pod в подсети, выделенной для узла). Способ определения подсети для каждого мо-

дуля Pod зависит как от реализации Kubernetes, так и от логики работы используемого провайдера CNI. Некоторые CNI могут не выделять отдельную подсеть каждому узлу, но в целом это простой и понятный способ управления IP-адресами с течением времени, поэтому он довольно распространен.

Имейте в виду, что *оба* плагина, Calico и Antrea, создают отдельные подсети для модулей Pod на узлах, откуда модули получают IP-адреса. Если вам когда-нибудь понадобится заняться отладкой маршрутизации в CNI, знание распределения модулей Pod по узлам может здорово помочь в выборе машин, которые нужно перезагрузить или вообще выключить, в зависимости от используемой у вас методики сопровождения.

Ниже показано устройство `antrea-gw0` – IP-адрес шлюза для всех модулей Pod в кластере:



Как видите, мостовая модель сети имеет несколько отличий в типах создаваемых устройств:

- отсутствует маршрут «черной дыры» `blackhole`, так как он обрабатывается OVS;
- ядро управляет только маршрутами к самому шлюзу Antrea (`Antrea-gw0`);
- весь трафик данного модуля Pod направляется непосредственно в устройство `Antrea-gw0`. Нет глобальной маршрутизации к другим устройствам, как в протоколе BGP, используемом Calico CNI.

5.4.5 Замечание о провайдерах CNI и kube-proxy в разных ОС

Стоит отметить, что трюк с использованием `DaemonSet` для настройки сети для модулей Pod характерен для Linux. В других ОС (например, в Windows) при запуске `containerd` требуется установить провайдера CNI с помощью диспетчера служб, при этом провайдер CNI запускается как процесс хоста. В будущем ситуация может измениться (в настоящее время ведется работа по включению привилегированных контейнеров для узлов Windows Kubernetes), тем не менее важно отметить, что сетевой стек Linux идеально подходит для сетевой модели Kubernetes. Во многом это связано с архитектурой контрольных групп, пространств имен и концепцией *суперпользователя root*, позволяющей запускать процессы с повышенными привилегиями даже в контейнере.

Поначалу сложность сети Kubernetes может показаться пугающей из-за быстрого развития сервисных сеток, CNI и сетевых прокси, но только до тех пор, пока вы не поймете основы маршрутизации между модулями Pod, остающиеся неизменными во многих реализациях CNI.

Итоги

- Сетевая архитектура Kubernetes имеет много параллелей с общими идеями программно-определяемых сетей (SDN).
- Antrea и Calico – провайдеры CNI, накладывающие кластерную сеть на реальную сеть для модулей Pod.
- Для определения структуры сети модулей Pod можно использовать базовые команды Linux (например, `ip a`).
- Провайдеры CNI обычно управляют сетями модулей Pod в наборах DaemonSet, которые запускают привилегированный контейнер Linux на каждом узле.
- Протокол пограничного шлюза (Border Gateway Protocol, BGP) и виртуальный коммутатор Open vSwitch (OVS) – это основные технологии организации сетевого интерфейса контейнеров CNI, которые решают одни и те же фундаментальные задачи широко-вещательной передачи и совместного использования информации о маршрутизации трафика модулей.
- Другие ОС, такие как Windows, в настоящее время не имеют всех встроенных механизмов, упрощающих организацию сети модулей Pod, которые имеются в Linux.

Устранение проблем в крупномасштабных сетях

В этой главе:

- подтверждение работоспособности кластера с помощью Sonobuoy;
- трассировка движения данных модулей Pod;
- использование команд `arp` и `ip` для проверки маршрутизации CNI;
- подробнее о `kube-proxy` и `iptables`;
- введение в сетевой уровень 7 (входной ресурс).

В этой главе мы рассмотрим несколько приемов устранения проблем в крупномасштабных сетях, а также познакомимся с многофункциональным инструментом *Sonobuoy*, предназначенным для сертификации, диагностики и тестирования кластеров, который широко используется для диагностики Kubernetes.

Sonobuoy и тесты Kubernetes e2e

Sonobuoy запускает набор тестов Kubernetes e2e в контейнере и упрощает получение, хранение и архивирование результатов.

Опытные пользователи Kubernetes, использующие версию Kubernetes, собранную из исходных кодов, могут напрямую использовать каталог

test/e2e/ (доступный по адресу <http://mng.bz/Dgx9>) взамен Sonobuoy. Мы рекомендуем этот инструмент как отправную точку желающим узнать больше о том, как запускать тесты Kubernetes.

Sonobuoy основан на библиотеке тестирования Kubernetes e2e и используется для проверки выпускаемых версий Kubernetes и их соответствия спецификации Kubernetes API. В конце концов, Kubernetes – это просто библиотека, поэтому мы определяем кластер Kubernetes как набор узлов, который может успешно пройти тестирование.

Опробование kind-local-up.sh

Изучение различных CNI – отличный способ попрактиковаться в устранении неполадок в реальных сетях, с которыми можно столкнуться в реальной жизни. Вы можете использовать рецепты (<http://mng.bz/2jg0>) запуска различных вариантов кластеров Kubernetes с разными провайдерами CNI. Например, клонировав этот проект, можно выполнить команду `CLUSTER=calico CONFIG=calico-conf.yaml ./kind-local-up.sh`, чтобы создать кластер на основе Calico. Другие варианты поддержки CNI (например, Antrea и Cillium) также доступны в сценарии `kind-local-up.sh`. Вот как можно создать кластер на основе Antrea, чтобы следовать за примерами в этой главе:

```
CLUSTER=antrea CONFIG=kind-conf.yaml ./kind-local-up.sh
```

Вы можете изменить параметр `CLUSTER` и использовать другой тип интерфейса CNI, например `calico` или `cillium`. Если после создания кластера вывести список всех модулей Pod в пространстве имен `kube-system`, вы должны увидеть, что модули CNI успешно работают.

ПРИМЕЧАНИЕ Не стесняйтесь сообщать о проблемах в репозитории (<https://github.com/jayunit100/k8sprototypes>), если, например, хотели бы увидеть поддержку нового интерфейса CNI или конкретная версия вызывает у вас проблемы в окружении `kind`.

6.1 Sonobuoy: инструмент подтверждения работоспособности кластера

Набор тестов включает сотни проверок, помогающих подтвердить работоспособность всех аспектов кластера, от томов хранилища и поддержки сети до планирования модулей Pod и возможности запуска некоторых основных приложений. Проект Sonobuoy (<https://sonobuoy.io/>) содержит набор тестов Kubernetes e2e, которые можно запускать на любом кластере, чтобы узнать, какие части наших кластеров могут

работать неправильно. В общем случае можно загрузить Sonobuoy, а затем выполнить следующую команду:

```
$ wget https://github.com/vmware-tanzu/sonobuoy/releases/
    download/v0.51.0/sonobuoy_0.51.0_darwin_amd64.tar.gz
$ tar -xvf sonobuoy
$ chmod +x sonobuoy ; cp sonobuoy /usr/local/bin/
$ sonobuoy run e2e --focus=Conformance
```

Этот пример выполняет установку Sonobuoy в MacOS, поэтому используйте файл, соответствующий вашей операционной системе. Выполнение тестов обычно занимает от 1 до 2 ч на исправном кластере. По завершении можно выполнить команду `sonobuoy status` и получить отчет о работоспособности кластера. Компоненты кластера можно тестировать по отдельности, например, вот как можно проверить одну только сеть:

```
$ sonobuoy run e2e --e2e-focus=intra-pod
```

Этот тест проверит возможность взаимодействий модулей Pod на разных узлах взаимодействовать друг с другом и тем самым подтвердить правильную работу основных функций интерфейса CNI и сетевого прокси (`kube-proxy`). Например:

```
$ sonobuoy status


| PLUGIN | STATUS   | RESULT | COUNT |
|--------|----------|--------|-------|
| e2e    | complete | passed | 1     |


```

6.1.1 Трассировка движения данных модулей Pod в кластере

NetworkPolicy API позволяет создавать в родной для Kubernetes манере правила брандмауэра, ориентированные на приложения, и является основой безопасности взаимодействий в кластере. Этот API действует на уровне модуля Pod, что позволяет разрешать или блокировать подключение модулей друг к другу правилами NetworkPolicy, существующими в определенном пространстве имен. Сетевые политики NetworkPolicy, сервисы и провайдеры CNI тесно взаимодействуют друг с другом, как показано на рис. 6.1. Логический путь данных между любыми двумя модулями в кластере можно обобщить, как показано на рисунке, где:

- модуль Pod с адресом 100.96.1.2 отправляет трафик на IP-адрес сервиса, который он получает через DNS-запрос (не показан на рисунке);
- затем сервис пересылает трафик на IP-адрес, определяемый правилами `iptables`;
- правила `iptables` направляют трафик модулю Pod на другом узле;
- узел получает пакет, и правила `iptables` (или OVS) определяют, нарушает ли он политику сети;
- пакет доставляется в конечную точку 100.96.1.3.

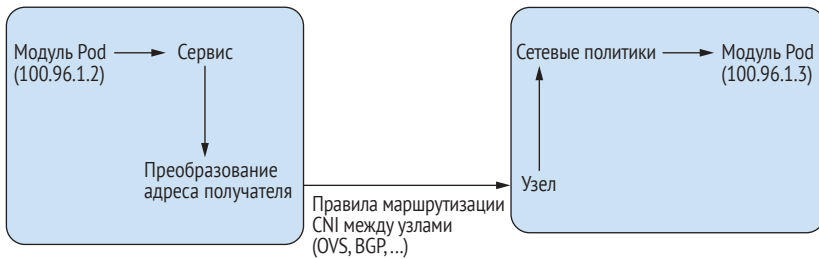


Рис. 6.1 Логический путь данных между любыми двумя модулями Pod в кластере

В этом примере не учитываются некоторые особенности поведения реальных кластеров. Например, в реальном мире:

- на первый модуль Pod тоже могут распространяться правила сетевой политики;
- интерфейс между узлами 10.1.2.3 и 10.1.2.4 может контролироваться брандмауэром;
- CNI может быть отключен или работать со сбоями, т. е. маршрутизация пакета может осуществляться в ином месте;
- часто в реальном мире для доступа одного модуля Pod к другому могут потребоваться сертификаты mTLS.

Как вы, вероятно, уже знаете, правила iptables состоят из *цепочек* и *правил*. Каждая таблица iptables имеет разные цепочки, состоящие из правил, которые определяют движение пакетов. Следующие цепочки управляются сервисом kube-proxy и определяют правила передачи пакета через кластер. (В следующем разделе вы узнаете, что именно мы подразумеваем под *устройством маршрутизации*.)

```
KUBE_MARK_MASQ -> KUBE-SVC -----> KUBE_MARK_DROP
|-----> KUBE_SEP -> KUBE_MARK_MASK -> NODE -> устройство маршрутизации
```

6.1.2 Настройка кластера с CNI-провайдером Antrea

В предыдущей главе мы говорили об особенностях управления сетевым трафиком с помощью Calico. В этой главе мы снова используем этот плагин, а также рассмотрим некоторые тонкости работы CNI-провайдера Antrea, использующего Open vSwitch (OVS) в качестве альтернативы технологиям маршрутизации, которые использует Calico. Это означает:

- вместо широковещательной рассылки IP-адресов, маршрутизируемых на всех узлах, как это делает BGP, OVS запускает коммутатор на каждом узле, который управляет трафиком по мере его поступления;
- в маршрутизаторах OVS сетевые политики определяются не правилами iptables, а правилами для Kubernetes NetworkPolicy API.

Мы еще раз рассмотрим некоторые понятия, потому что считаем, что обзор одних и тех же сведений под другим углом значительно

упрощает понимание работы сети в реальном мире. Однако на этот раз мы пойдем немного быстрее, потому что предполагаем, что вы понимаете некоторые концепции, представленные в предыдущих главах, такие как сервисы, правила iptables, провайдеры CNI и назначение IP-адресов модулям Pod.

Чтобы настроить кластер с провайдером Antrea, используем kind, как уже делали это, когда знакомились с провайдером Calico; однако на этот раз напрямую используем «рецепты», предоставляемые проектом Antrea. Чтобы создать кластер kind с провайдером Antrea, выполните следующие шаги:

```
$ git clone https://github.com/vmware-tanzu/antrea/  
$ cd antrea  
$ cd ci/kind  
$ ./kind-setup.sh
```

ВНИМАНИЕ В этой главе дается несколько расширенный пример. Чтобы излишне не нагружать вас, мы будем предполагать, что вы в состоянии переключаться между кластерами с разными провайдерами. Если у вас нет действующих кластеров с Antrea и Calico, то мы советуем опробовать только некоторые из приводимых команд и не стараться следовать за всеми примерами в этой главе. Как всегда, при исследовании сетевых механизмов вам может понадобиться выполнить `apt-get update`; `apt-get install net-tools` в кластере kind, если вы еще этого не сделали.

6.2 Исследование особенностей маршрутизации в разных провайдерах CNI с помощью команд `arp` и `ip`

На этот раз мы оставим kind в стороне

Вы можете запустить Antrea в кластере kind, но в этой главе мы покажем примеры, полученные в кластере VMware Tanzu. Желающие опробовать эти примеры в kind могут воспользоваться сценарием, доступным по адресу <http://mng.bz/2jg0> и поддерживающим плагины Calico, Cilium и Antrea. Но имейте в виду, что для правильной работы провайдеров Cilium и Antrea в кластере kind требуется выполнить дополнительные настройки сети Linux (eBPF и OVS соответственно).

Вся работа IP-сетей основана на идее, что IP-адреса в конечном итоге приведут вас к какому-то аппаратному устройству, которое работает уровнем ниже (на уровне 2) абстракции IP (уровень 3) и, соответ-

ственно, поддерживает возможность MAC-адресации. Часто первым шагом в тестировании сети является запуск `ip a`. Эта команда позволяет получить общее представление о сетевых интерфейсах, имеющих на хосте, и устройствах, которые являются конечными точками сети в вашем кластере.

В кластере с CNI Antrea можно зайти на любой узел с помощью той же команды `docker exec`, как не раз было показано в предыдущих главах, и ввести команду `arp -n`, чтобы посмотреть, какие устройства известны данному узлу. В этой главе мы покажем результаты, полученные в реальных виртуальных машинах, чтобы вы могли использовать их как эталон при обзоре сетей Antrea в своих локальных кластерах.

Для начала зайдём на узел и посмотрим список известных ему IP-адресов, запустив команду `arp`. Чтобы получить адреса модулей Pod, доступных узлу, выберем IP-адреса с фильтром `grep 100`. Этот пример мы выполнили в кластере с машинами, находящимися в подсети 100:

```
antrea_node> arp -n | grep 100
? (100.96.26.15) at 86:55:7a:e3:73:71 [ether] on antrea-gw0
? (100.96.26.16) at 4a:ee:27:03:1d:c6 [ether] on antrea-gw0
? (100.96.26.17) at <incomplete> on antrea-gw0
? (100.96.26.18) at ba:fe:0f:3c:29:d9 [ether] on antrea-gw0
? (100.96.26.19) at e2:99:63:53:a9:68 [ether] on antrea-gw0
? (100.96.26.20) at ba:46:5e:de:d8:bc [ether] on antrea-gw0
? (100.96.26.21) at ce:00:32:c0:ce:ec [ether] on antrea-gw0
? (100.96.26.22) at e2:10:0b:60:ab:bb [ether] on antrea-gw0
? (100.96.26.2) at 1a:37:67:98:d8:75 [ether] on antrea-gw0
Адреса, локальные для узла:
antrea_node> arp -n | grep 192
? (192.168.5.160) at 00:50:56:b0:ee:ff [ether] on eth0
? (192.168.5.1) at 02:50:56:56:44:52 [ether] on eth0
? (192.168.5.207) at 00:50:56:b0:80:64 [ether] on eth0
? (192.168.5.245) at 00:50:56:b0:e2:13 [ether] on eth0
? (192.168.5.43) at 00:50:56:b0:0f:52 [ether] on eth0
? (192.168.5.54) at 00:50:56:b0:e4:6d [ether] on eth0
? (192.168.5.93) at 00:50:56:b0:1b:5b [ether] on eth0
```

6.2.1 Что такое IP-туннель и почему его используют провайдеры CNI?

Возможно, вам интересно, что это за устройство `antrea-gw0`. Если запустить эти команды в кластере Calico, то также можно увидеть устройство `tun0`. Они известны как *туннели* и обеспечивают возможность создания *плоской* сети, соединяющей модули Pod в кластере. Устройства `antrea-gw0`, которые можно видеть в предыдущем примере, соответствуют шлюзу OVS, управляющему трафиком. Этот шлюз достаточно умен, чтобы «замаскировать» трафик от одного модуля Pod к другому и передать его сначала узлу. В кластерах Calico вы увидите аналогичную схему маскировки с использованием протокола

(например, IPIP). Оба CNI-провайдера, Calico и Antrea, достаточно интеллектуальны, чтобы знать, когда маскировать трафик для лучшей производительности.

Теперь давайте посмотрим, где CNI-интерфейсы Antrea и Calico начинают различаться. В нашем кластере с Calico команда `ip a` показывает наличие интерфейса `tunl0`. Он создается контейнером `calico_node` через сервис `brd`, который отвечает за маршрутизацию трафика через туннель IPIP в кластере. Сравним полученный вывод с выводом `ip a` в кластере с Antrea, что показан во втором фрагменте кода.

```
calico_node> ip a
2: tunl0@NONE: <NOARP,UP,LOWER_UP>
mtu 1440 qdisc noqueue state UNKNOWN
group default qlen 1000

antrea_node> ip a
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
group default qlen 1000
link/ether 7e:de:21:4b:88:46 brd ff:ff:ff:ff:ff:ff
5: antrea-gw0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
noqueue state UNKNOWN group default qlen 1000
link/ether 82:aa:a9:6f:02:33 brd ff:ff:ff:ff:ff:ff
inet 100.96.29.1/24 brd 100.96.29.255 scope global antrea-gw0
valid_lft forever preferred_lft forever
inet6 fe80::80aa:a9ff:fe6f:233/64 scope link
```

Теперь в обоих кластерах выполним `kubectl scale deployment coredns --replicas=10 -n kube-system`, затем еще раз – предыдущие команды. Теперь должны появиться новые записи для контейнеров.

6.2.2 Сколько пакетов проходит через сетевые интерфейсы CNI?

Мы знаем, что пакеты нужно протолкнуть в специальные туннели, чтобы они попали на нужные узлы, где находятся модули-получатели. Поскольку весь локальный трафик модулей Pod доступен на уровне узла, для его мониторинга можно использовать стандартные инструменты Linux, и для этого не нужно полагаться на знание самого Kubernetes. Команда `ip` имеет параметр `-s`, позволяющий выводить подробную информацию о текущем трафике. Выполнив эту команду на узле кластера с CNI-провайдером Calico или Antrea, можно узнать объем трафика, поступающего в модуль Pod через интерфейс CNI. Вот пример вывода:

```
10: cali3317e4b4ab5@if5: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 1440 qdisc noqueue state UP group default
link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
link-netns cni-abb79f5f-b6b0-f548-3222-34b5eec7c94f
RX: bytes packets errors dropped overrun mcast
```

```

150575    1865    0      2      0      0
TX: bytes packets errors dropped carrier collsns
839360    1919    0      0      0      0

5: antrea-gw0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
➔ noqueue state UNKNOWN group default qlen 1000
   link/ether 82:aa:a9:6f:02:33 brd ff:ff:ff:ff:ff:ff
   inet 100.96.29.1/24 brd 100.96.29.255 scope global antrea-gw0
       valid_lft forever preferred_lft forever
   inet6 fe80::80aa:a9ff:fe6f:233/64 scope link
       valid_lft forever preferred_lft forever
RX: bytes packets errors dropped overrun mcast
89662090 1089577 0      0      0      0
TX: bytes packets errors dropped carrier collsns
108901694 1208573 0      0      0      0

```

Теперь у нас есть общее представление о работе сетевых соединений в наших кластерах. Если на узле отсутствует трафик, текущий через интерфейс, созданный плагином Calico или Antrea, то это явно свидетельствует о неисправности CNI, потому что в большинстве кластеров Kubernetes между действующими модулями Pod всегда течет некоторый трафик. Даже в отсутствие прикладных модулей Pod модули kube-проху и CoreDNS в кластере kind будут активно обмениваться данными о сетевом трафике через конечную точку сервиса CoreDNS. Наблюдение за модулями Pod, находящимися в состоянии «Работает» – хороший тест на работоспособность (особенно наблюдение за сервисом CoreDNS, для работы которого нужна сеть модулей Pod) и способ убедиться в исправности провайдера CNI.

6.2.3 Маршруты

Следующая остановка в нашем путешествии – обзор особенностей присваивания IP-адресов этим устройствам. На рис. 6.2 снова изображена архитектура сети Kubernetes. Но на этот мы добавили информацию о туннелировании, полученную предыдущими командами.

Calico Pods route to Calico devices for individual Pods.

Antrea Pods route to the OVS gateway, either local or remote, in the Pod network.

Маршруты от отдельных модулей Pod к устройствам Calico в кластере с CNI-провайдером Calico

```

192.168.9.130 -----> tun0
192.168.173.65 -----> calic2f...
192.168.173.66 -----> calicaa57...

```

Маршруты к шлюзу OVS, локальному или удаленному, в сети модулей Pod в кластере с CNI-провайдером Antrea

```

100.96.0.0 -----> 100.96.0.1 -----> antrea-gw0
100.96.21.0 -----> 100.96.21.1 ----->
100.96.26.0 -----> 100.96.26.1 ----->

```

Рис. 6.2 Архитектура сети Kubernetes с информацией о туннелировании

Теперь, узнав о существовании туннелей, давайте посмотрим, как провайдер CNI направляет трафик в туннели с помощью таблицы маршрутизации Linux. Выполнив команду `route -n` в нашем кластере Calico, мы получили следующую таблицу маршрутизации, где интерфейсы `cali` – это локальные модули Pod, находящиеся на узле, а интерфейсы `tunl0` – это специальные интерфейсы, созданные самим плагином Calico для отправки трафика на узел шлюза:

```
# route -n
Kernel IP routing table
Destination    Gateway        Genmask         Flags Metric Ref Use Iface
0.0.0.0        172.18.0.1    0.0.0.0         UG    0     0   0   eth0
172.18.0.0     0.0.0.0       255.255.0.0     U     0     0   0   eth0
192.168.9.128  172.18.0.3    255.255.255.192 UG    0     0   0   tunl0
192.168.71.0   172.18.0.5    255.255.255.192 UG    0     0   0   tunl0
192.168.88.0   172.18.0.4    255.255.255.192 UG    0     0   0   tunl0
192.168.143.64 172.18.0.2    255.255.255.192 UG    0     0   0   tunl0
192.168.173.64 0.0.0.0       255.255.255.192 U     0     0   0   *
192.168.173.65 0.0.0.0       255.255.255.255 UH    0     0   0   calicd2f3
192.168.173.66 0.0.0.0       255.255.255.255 UH    0     0   0   calibaa57
```

Как можно заметить в этой таблице:

- узлы в подсети 172 являются шлюзами для некоторых модулей Pod;
- IP-адреса 192 в определенных диапазонах (показаны в столбце Genmask) маршрутизируются на определенные узлы.

А теперь перейдем к CNI-провайдеру Antrea. В кластере с этим провайдером мы не увидим новых IP-адресов получателей, присваиваемых каждому устройству, но увидим шлюз .1:

```
root [ /home/capv ]# route -n
Kernel IP routing table
Destination    Gateway        Genmask         Flags Ref Use Iface
0.0.0.0        192.168.5.1    0.0.0.0         UG    0     0   0   eth0
100.96.0.0     100.96.0.1    255.255.255.0   UG    0     0   0   antrea-gw0
100.96.21.0    100.96.21.1    255.255.255.0   UG    0     0   0   antrea-gw0
100.96.26.0    100.96.26.1    255.255.255.0   UG    0     0   0   antrea-gw0
100.96.28.0    100.96.28.1    255.255.255.0   UG    0     0   0   antrea-gw0
```

Как можно заметить в этой таблице:

- любой трафик для получателей с IP-адресами из диапазона 100.96.0.0 направляется непосредственно на IP-адрес 100.96.0.1. Это зарезервированный IP-адрес в сети CNI, который Antrea использует для OVS-маршрутизации. То есть провайдер отправляет трафик не на IP-адрес узла непосредственно, а на IP-адрес службы коммутатора Antrea в сети Pod;
- в отличие от Calico, Antrea направляет весь трафик (включая локальный) непосредственно в шлюз. Единственное наблюдаемое отличие: конечный пункт назначения – это IP-адрес шлюза.

Из вышесказанного можно заключить следующее:

- Antrea создает одну запись в таблице маршрутизации для каждого узла;
- Calico создает одну запись в таблице маршрутизации для каждого модуля Pod.

6.2.4 Инструменты для CNI: Open vSwitch (OVS)

Плагины Antrea и Calico работают как модули Pod. Это верно не для всех провайдеров CNI, но если это так, то для отладки маршрутизации данных можно использовать множество полезных особенностей Kubernetes. Приступая к исследованию работы внутренних механизмов CNI, обратите внимание на такие инструменты, как `ovs-vsctl`, `antctl`, `calicoctl` и т. д. Мы не будем подробно рассматривать их все, а познакомим только с инструментом `ovs-vsctl`, который легко запустить в контейнере Antrea внутри кластера. С его помощью можно получить массу интересной информации об этом интерфейсе. Чтобы воспользоваться этим инструментом, можно в контейнере Antrea запустить командную оболочку командой `kubectrl exec -t -i antrea-agent-1234 -n kube-system /bin/bash`, а затем вызвать команду:

```
# ovs-vsctl list interface|grep -A 5 antrea
name                : antrea-gw0
ofport              : 2
ofport_request      : 2
options             : {}
other_config        : {}
statistics          : {collisions=0, rx_bytes=1773391201,
                        rx_crc_err=0, rx_dropped=0, rx_errors=0,
                        rx_frame_err=0, rx_missed_errors=0, rx_over_err=0,
                        rx_packets=16392260, tx_bytes=6090558410,
                        tx_dropped=0, tx_errors=0, tx_packets=17952545}
```

Есть несколько инструментов командной строки, позволяющих диагностировать низкоуровневые проблемы в CNI. Для отладки можно использовать `antctl` или `calicoctl`:

- `antctl` выводит перечень активных функций Antrea, получает отладочную информацию об агентах и выполняет детальный анализ объектов NetworkPolicy;
- `calicoctl` тоже анализирует объекты NetworkPolicy, выводит диагностическую информацию о сети и позволяет отключать сетевые функции (как альтернатива ручному редактированию файлов YAML).

Один из универсальных подходов к отладке кластеров в Linux предлагает Sonobuoy – инструмент для запуска набора тестов e2e в кластере. Также обратите внимание на инструмент <https://github.com/sarun87/k8snetlook>, который производит детальную диагности-

ку функционирования сети кластера (например, проверяет возможность подключения к серверу API, к модулям Pod и т. д.).

В зависимости от степени сложности топологии сети, объем действий по устранению неполадок в реальном кластере может изменяться. Довольно часто на один узел приходится более 100 модулей Pod, поэтому особую важность приобретает возможность проверки или рассуждений.

6.2.5 Трассировка движения данных активных контейнеров с помощью tcpdump

Теперь, получив некоторое представление о том, как пакеты перемещаются из одного места в другое при использовании разных провайдеров CNI, вернемся обратно и рассмотрим приемы использования одного из традиционных инструментов диагностики сетей: tcpdump. Мы проследили взаимосвязь между хостом и базовыми средствами маршрутизации в Linux и теперь взглянем на трафик с точки зрения контейнера. Чаще всего для этой цели используется tcpdump. Давайте возьмем один из наших контейнеров CoreDNS и исследуем его трафик. В Calico пакеты можно перехватывать непосредственно на устройствах cali:

```
192.168.173.66 0.0.0.0 255.255.255.255 UH 0 0 0 calibaa5769d671
calico_node> tcpdump -i calicd2f389598e
listening on calicd2f389598e,
link-type EN10MB (Ethernet),
capture size 262144 bytes
20:13:07.733139 IP 10.96.0.1.443 > 192.168.173.65.60684:
  Flags [P.],
  seq 1615967839:1615968486,
  ack 1173977013, win 264,
  options [nop,nop,TS val 296478
```

IP-адрес 10.96.0.1 – это адрес внутреннего сервиса Kubernetes (сервера API), присутствие которого в выводе подтверждает получение запроса от сервера CoreDNS на получение записи DNS. Если посмотреть на типичный узел в кластере, где действует модуль Pod с сервисом CoreDNS, то модули с Antrea будут называться так:

```
30: coredns--e5cc00@if3: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 1450 qdisc noqueue master ovs-system state UP
group default
link/ether e6:8a:27:05:d7:30 brd ff:ff:ff:ff:ff:ff
link-netns cni-2c6b1bc0-cf36-132c-dfcb-88dd158f51ca
inet6 fe80::e48a:27ff:fe05:d730/64 scope link
valid_lft forever preferred_lft forever
```

Это означает возможность напрямую перехватывать пакеты, поступающие на этот узел, подключившись к veth-устройству с помощью `tcpdump`. Вот как это сделать:

```
calico_node> tcpdump -i coredns--29244a -n
```

Запустив эту команду, вы должны увидеть трафик от разных модулей Pod к серверу DNS в Kubernetes. Мы сами часто используем параметр `-n` для вывода наших IP-адресов при использовании `tcpdump`.

Чтобы узнать, взаимодействует ли один Pod с другим, можно зайти на узел, где находится Pod-приемник, и перехватить весь TCP-трафик, включающий один из IP-адресов модуля Pod. Допустим, что Pod, управляющий трафиком, имеет адрес 100.96.21.21. Следующая команда даст полный дамп трафика, например, с адресом 19 и номером порта 9153:

```
calico_node> tcpdump host 100.96.21.21 -i coredns--29244a
listening on coredns--29244a, link-type EN10MB (Ethernet),
capture size 262144 bytes
```

```
21:59:36.818933 IP 100.96.21.21.45978 > 100.96.26.19.9153:
  Flags [S], seq 375193568, win 64860, options [mss 1410,sackOK,TS
    val 259983321 ecr 0,nop,wscale 7], length 0
```

```
21:59:36.819008 IP 100.96.26.19.9153 > 100.96.21.21.45978: Flags [S.],
  seq 3927639393, ack 375193569, win 64308, options [mss 1410,
    sackOK,TS val 2440057191 ecr 259983321,nop,wscale 7], length 0
```

```
21:59:36.819928 IP 100.96.21.21.45978 > 100.96.26.19.9153:
  Flags [.], ack 1, win 507, options [nop,nop,TS val
    259983323 ecr 2440057191], length 0
```

`tcpdump` часто используется для оперативного исследования трафика между парой контейнеров. В частности, если вы не видите подтверждения (пакета `ack`), возвращаемого принимающим модулем отправителю, то это может означать, что принимающий модуль Pod не получает трафик, например, из-за действия сетевых политик или правил `iptables`, мешающих нормальной пересылке информации в kube-проху.

ПРИМЕЧАНИЕ В традиционных вычислительных центрах для настройки и управления правилами `iptables` часто используются такие инструменты, как `Puppet`. Трудно комбинировать kube-проху с правилами `iptables`, которые управляются другими сетевыми правилами, и зачастую лучше запускать узлы в среде, изолированной от обычных правил, поддерживаемых вашими сетевыми администраторами.

6.3 *kube-proxy и iptables*

Самое важное, что нужно помнить о сетевом прокси: его действия в общем случае не зависят от действий провайдера CNI. Конечно, как и все остальное в Kubernetes, это утверждение верно лишь с оговоркой: некоторые провайдеры CNI реализуют свой сервис прокси как альтернативу проксированию с помощью iptables (или IPVS), реализованному в Kubernetes. Однако использование этих сервисов нетипично для большинства кластеров, где желательно концептуально отделить проксирование, осуществляемое сервером kube-проху, от маршрутизации трафика, выполняемым провайдером CNI (например, OVS), который управляет примитивами Linux.

В этом разделе мы вновь погрузимся в основные сетевые концепции Kubernetes. К настоящему моменту мы видели:

- как хост отображает трафик модулей Pod в IP-адреса и маршруты;
- как исследовать входящий трафик модуля Pod и получить информацию об IP-туннелировании;
- как перехватить трафик с определенными IP-адресами с помощью tcpdump.

Теперь давайте обратим внимание на kube-проху. Несмотря на то что он не является частью CNI, понимание kube-проху совершенно необходимо для диагностики сетевых проблем.

6.3.1 *iptables-save и diff*

Самое простое, что можно сделать при поиске всех конечных точек сервиса, – запустить iptables-save в кластере. Эта команда сохраняет каждое правило iptables в определенный момент времени. Затем, используя такие инструменты, как diff, можно оценить различия между двумя состояниями сети Kubernetes и поискать комментарии, сообщающие, какие сервисы связаны с тем или иным правилом. Типичный запуск iptables-save приводит к появлению нескольких строк таких правил:

```
-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
  --comment "kube-system/kube-dns:dns" -m statistic --mode random
  --probability 0.10000000009 -j KUBE-SEP-QIVPDYSUOLQYQCAA

-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
  --comment "kube-system/kube-dns:dns" -m statistic --mode random
  --probability 0.11111111101 -j KUBE-SEP-N76EJY3A4RTXTN2I

-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
  --comment "kube-system/kube-dns:dns" -m statistic --mode random
  --probability 0.12500000000 -j KUBE-SEP-LSGM2AJGRPG672RM
```

Выяснив сервисы, можно найти соответствующие им правила SEP, например, с помощью `grep`. В данном случае `SEP-QI`... соответствует контейнеру CoreDNS в нашем кластере.

ПРИМЕЧАНИЕ Во многих примерах для иллюстрации мы используем CoreDNS, потому что это стандартный модуль, который можно масштабировать и который, вероятно, имеется практически в любом кластере. Описываемый пример можно опробовать с любым другим модулем Pod, доступным через внутренний сервис Kubernetes и получающим IP-адрес с помощью плагина CNI (т. е. не использующий сеть хоста).

```
calico_node> iptables-save | grep SEP-QI
:KUBE-SEP-QIVPDYSUOLQYQCAA - [0:0]
### Здесь к исходящему трафику применяется маскардинг...
-A KUBE-SEP-QIVPDYSUOLQYQCAA -s 192.168.143.65/32
  -m comment
  --comment "kube-system/kube-dns:dns" -j KUBE-MARK-MASQ

-A KUBE-SEP-QIVPDYSUOLQYQCAA -p udp -m comment
  --comment "kube-system/kube-dns:dns" -m udp -j DNAT
  --to-destination 192.168.143.65:53

-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
  --comment "kube-system/kube-dns:dns" -m statistic
  --mode random --probability 0.100000000009 -j KUBE-SEP-QIVPDYSUOLQYQCAA
```

Этот шаг одинаков для любого провайдера CNI, поэтому мы не будем приводить сравнение Antrea/Calico.

6.3.2 Как сетевые политики изменяют правила CNI

Правила фильтрации входящего трафика и сетевые политики NetworkPolicy – две самые важные функции сети Kubernetes, потому что обе определяются интерфейсом API, но реализуются внешними сервисами, которые считаются необязательными в кластере. По иронии судьбы сетевые политики и маршрутизация входящего трафика являются важнейшими для большинства администраторов, поэтому, несмотря на теоретическую необязательность этих функций, вы почти наверняка будете их использовать.

Сетевые политики NetworkPolicy в Kubernetes поддерживают блокировку входящего и исходящего трафика для любого модуля Pod. Как правило, модули в кластере Kubernetes вообще никак не защищены, поэтому сетевые политики считаются неотъемлемой частью системы безопасности промышленных кластеров Kubernetes. Интерфейс NetworkPolicy API выглядит довольно сложным для новичков, поэтому несколько упростим его для начала:

- сетевые политики NetworkPolicy создаются в определенном пространстве имен и воздействуют на модули Pod, определяемые заданными метками;
- сетевые политики NetworkPolicy должны определять тип трафика (по умолчанию входящий);
- сетевые политики – аддитивные, работают по принципу «что не разрешено, то запрещено» и могут быть многоуровневыми, разрешая прохождение разнообразного трафика;
- оба плагина, Calico и Antrea, по-разному реализуют Kubernetes NetworkPolicy API. Calico создает новые правила iptables, а Antrea – правила OVS;
- некоторые сетевые интерфейсы CNI, такие как Flannel, вообще не реализуют NetworkPolicy API;
- некоторые сетевые интерфейсы CNI, такие как Cilium и OVN (Open Virtual Network – открытая виртуальная сеть) Kubernetes, реализуют не весь набор возможностей Kubernetes NetworkPolicy API (например, Cilium не реализует недавно добавленную политику PortRange, которая на момент публикации находилась в состоянии бета-версии, а OVN Kubernetes не реализует политику NamedPort).

Важно понимать, что Calico использует iptables только для реализации сетевых политик. Вся остальная маршрутизация осуществляется с помощью правил BGP, как было показано в предыдущем разделе. В этом разделе мы определим сетевую политику и посмотрим, как она влияет на правила маршрутизации в Calico и Antrea. Чтобы понять, как сетевые политики могут влиять на трафик, применим политику NetworkPolicy, блокирующую весь трафик к модулю Pod с именем web:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
  ingress: []
```

Эта политика применяется к контейнеру app:web в пространстве имен по умолчанию

Запрещает весь трафик, потому что фактически здесь отсутствуют какие-либо правила для применения к входящему трафику

Если бы мы хотели определить правило, разрешающее прохождение некоторого входящего трафика, то политика могла бы выглядеть примерно так:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web
spec:
  podSelector:
    matchLabels:
```

```

    app: web
  ingress:
  - ports:
    - port: 80
  - from:
    - podSelector:
        matchLabels:
          app: web2

```

Разрешаем трафик, но только адресованный в порт 80, который обслуживает наш веб-сервер

Позволяет модулю web реагировать на трафик, отправленный модулем web2

Обратите внимание, что второе определение NetworkPolicy разрешает также модулю web2 получать трафик от модуля web. Это объясняется тем, что для модуля web не определены никакие политики, управляющие исходящим трафиком, вследствие чего любой исходящий трафик будет разрешен по умолчанию. Таким образом, чтобы полностью оградить модуль web, необходимо:

- определить политику для *исходящего трафика*, которая пропускала бы исходящий трафик только к основным сервисам;
- определить политику для *входящего трафика*, которая пропускала бы входящий трафик только от основных сервисов;
- добавить номера портов в предыдущие политики, чтобы было разрешено прохождение трафика только через основные порты.

Определение подобных политик в YAML-файле может потребовать значительных усилий. Желающим глубже изучить эту тему мы рекомендуем заглянуть в репозиторий <http://mng.bz/XWEI>, где приводятся рекомендации по созданию конкретных сетевых политик для различных вариантов использования.

Хороший способ проверить политики, созданные провайдером CNI, – определить DaemonSet, запускающий один и тот же контейнер на всех узлах. Обратите внимание, что создание правил NetworkPolicy провайдером CNI является функцией самого провайдера, – это не часть интерфейса CNI. Большинство провайдеров CNI создается для Kubernetes, поэтому реализация Kubernetes NetworkPolicy API является очевидным дополнением, которое они предоставляют.

Теперь проверим нашу политику, создав модуль Pod, который может послужить целью. Следующий DaemonSet запускает указанный модуль Pod на каждом узле. Каждый модуль Pod защищен политикой, что определена выше, и это приводит к созданию определенного набора правил iptables CNI-провайдером Calico (или правил OVS CNI-провайдером Antrea). Протестировать нашу политику можно с помощью следующего кода:

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-ds
spec:
  selector:
    matchLabels:

```

```

app: web ← Модуль Pod для запуска на каждом узле
template:
  metadata:
    labels:
      app: web
  spec:
    containers:
      - name: nginx
        image: nginx

```

6.3.3 Как реализуются политики?

Для сравнения правил iptables до и после создания политики в Calico можно использовать diff или git diff. Давайте, например, реализуем правило drop. Для этого:

- создайте DaemonSet, определение которого приведено выше, а затем запустите iptables-save > a1 на любом узле;
- создайте сетевую политику, блокирующую трафик, запустив iptables-save > a2, чтобы сохранить правила в другом файле;
- запустите команду, например git diff a1 a2, и оцените различия.

В данном случае вы увидите следующие новые правила, созданные для политик:

```

> -A cali-tw-calic5cc839365a -m comment
--comment "cali:Uv2zkaIvaVnFWYI9" -m comment
--comment "Start of policies" -j MARK --set-xmark 0x0/0x20000

> -A cali-tw-calic5cc839365a -m comment
--comment "cali:70LyCb9i6s_CPjbu" -m mark --mark 0x0/0x20000
-j cali-pi_IDb4Gbl3P1MtRtVzfEP

> -A cali-tw-calic5cc839365a -m comment --comment "cali:DBkU9PXyu2eCwkJC"
-m comment --comment "Return if policy accepted" -m mark
--mark 0x10000/0x10000 -j RETURN

> -A cali-tw-calic5cc839365a -m comment --comment "cali:tioNk8N7f4P5Pzf4"
-m comment --comment "Drop if no policies passed packet" -m mark
--mark 0x0/0x20000 -j DROP

> -A cali-tw-calic5cc839365a -m comment --comment "cali:wcGG1iiHvTXsj5lq"
-j cali-pri-kns.default

> -A cali-tw-calic5cc839365a -m comment --comment "cali:gaGDuGQkGckLPa4H"
-m comment --comment "Return if profile accepted" -m mark
--mark 0x10000/0x10000 -j RETURN

> -A cali-tw-calic5cc839365a -m comment --comment "cali:B6l_lueEhRWiWwnn"
-j cali-pri-ksa.default.default

> -A cali-tw-calic5cc839365a -m comment --comment "cali:McPS2ZHishhYyFnw"
-m comment --comment "Return if profile accepted" -m mark

```



```
--mark 0x10000/0x10000 -j RETURN
> -A cali-tw-calic5cc839365a -m comment --comment "cali:lThI2kHuPODjvF4v"
  -m comment --comment "Drop if no profiles matched" -j DROP
```

Antrea тоже реализует сетевые политики, но использует потоки OVS и записывает их в таблицу 90. Запустив аналогичную рабочую нагрузку в Antrea, можно убедиться, что политики действительно созданы, вызвав `ovs-ofctl`. Обычно эта команда запускается внутри контейнеров, где размещаются агенты Antrea со всеми утилитами OVS, но ее можно запустить и на уровне хоста, нужно лишь установить утилиты OVS. Чтобы опробовать следующий пример в кластере Antrea, можно использовать клиента `kubectnl`. Следующая команда показывает, как Antrea реализует сетевые политики:

```
$ kubectnl -n kube-system exec -it antrea-agent-2kksz
➡ ovs-ofctl dump-flows br-int | grep table=90
...
Defaulting container name to antrea-agent.
cookie=0x20000000000000, duration=344936.777s, table=90, n_packets=0,
n_bytes=0, priority=210,ct_state=new+est,ip actions=resubmit(,105)

cookie=0x20000000000000, duration=344936.776s, table=90, n_packets=83160,
n_bytes=6153840, priority=210,ip,nw_src=100.96.26.1 actions=resubmit(,105)

cookie=0x20500000000000, duration=22.296s, table=90, n_packets=0,
n_bytes=0, priority=200,ip,reg1=0x18 actions=conjunction(1,2/2)

cookie=0x20500000000000, duration=22.300s, table=90, n_packets=0, n_bytes=0,
priority=190,conj_id=1,ip actions=load:0x1->NXM_NX_REG6[],resubmit(,105)

cookie=0x20000000000000, duration=344936.782s, table=90, n_packets=149662,
n_bytes=11075281, priority=0 actions=resubmit(,100)
```

OVS, так же как `iptables`, определяет правила, регламентирующие потоки пакетов. В OVS есть несколько потоковых таблиц, которые использует Antrea, и каждая поддерживает программную логику для разных модулей Pod. Если предполагается использовать Antrea в больших кластерах и необходима возможность получать оперативную информацию об использовании OVS, то количество активных потоков в OVS, например, можно определять в режиме реального времени с помощью таких инструментов, как Prometheus.

Не забывайте, что OVS и `iptables` интегрированы в ядро Linux, поэтому вам не придется делать что-то особенное, чтобы использовать эти технологии. Дополнительную информацию о трассировке OVS с помощью Prometheus можно найти в статье, опубликованной в блоге, посвященном этой книге и доступной по адресу <http://mng.bz/1jaj>. В ней подробно рассказывается, как настроить Prometheus для мониторинга Antrea.

Cyclonus и e2e-тесты для NetworkPolicy

Желающие узнать больше о сетевых политиках NetworkPolicy могут заняться их исследованием, запуская e2e-тесты Kubernetes с помощью Sooboo. Вы получите прекрасно оформленный список таблиц, явно указывающих, какие модули Pod могут или не могут взаимодействовать друг с другом согласно установленным политикам. Еще один мощный инструмент для изучения возможностей NetworkPolicy провайдера CNI – Cyclonus. Его легко получить из исходного кода (<https://github.com/mattfenwick/cyclonus>).

Cyclonus генерирует сотни сетевых политик и проверяет правильность их реализации провайдером CNI. Иногда в провайдерах CNI могут встречаться ошибки в реализации сложного NetworkPolicy API, поэтому мы рекомендуем запустить тестирование с помощью Cyclonus в промышленном окружении, чтобы проверить соответствие используемого провайдера CNI спецификации Kubernetes API.

6.4 Входные контроллеры

Входные контроллеры (ingress controllers) позволяют направить в кластер весь трафик через один IP-адрес (и это отличный способ сэкономить деньги на облачных IP-адресах). Но в некоторых сценариях они довольно сложны в отладке, потому что являются дополнительными компонентами. Чтобы исправить эту проблему, в сообществе Kubernetes было решено выпустить входной контроллер по умолчанию.

NGINX, Contour и Gateway API

Оригинальный Ingress API, предназначенный для управления входным трафиком в Kubernetes, был реализован в NGINX и стал каноническим стандартом. Однако вскоре после этого произошли два важных события:

- появился альтернативный входной контроллер Contour (<https://projectcontour.io/>), созданный в фонде облачных вычислений CNCF (Cloud Native Computing Foundation);
- появился Gateway API как альтернативное многопользовательское решение проблемы представления маршрутов из кластера Kubernetes.

На момент публикации этой книги Ingress API предполагалось заменить на Gateway API – гораздо более описательный и позволяющий гибко определять различные типы ресурсов уровня 7. По этой причине мы рекомендуем изучить сведения в этом разделе, но рассматривать их как трамплин перед исследованием Gateway API и его возможностей, способных удовлетворить ваши потребности в будущем. Узнать больше о Gateway API можно по адресу <https://gateway-api.sigs.k8s.io/>.

Чтобы реализовать входной контроллер (или Gateway API), нужно решить, как направлять в него трафик, потому что контроллер не может быть обычным сервисом ClusterIP. Если входной контроллер выйдет из строя, то передача трафика в кластер прервется, поэтому предпочтительнее запускать его как DaemonSet (если он работает в кластере) на всех узлах.

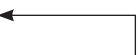
Для проксирования своих услуг Contour использует технологию Envoy, с помощью которой можно создавать входные контроллеры, сервисные сетки и другие сетевые компоненты, прозрачно управляющие трафиком. Обратите внимание, что Kubernetes Services API – это постоянная область инноваций в сообществе Kubernetes, и в ближайшие несколько лет с увеличением масштабов кластеров возникнет потребность в еще более сложных моделях маршрутизации трафика.

6.4.1 *Настройка Contour и кластера kind для изучения входных контроллеров*

Задача входных контроллеров – дать внешнему миру возможность обращаться к множеству сервисов в вашем кластере Kubernetes. Если вы пользуетесь услугами облачных вычислений с неограниченным количеством общедоступных IP-адресов, то ценность входных контроллеров может оказаться не такой очевидной, главная задача которых – дать возможность настроить сквозную передачу HTTPS, осуществлять мониторинг всех экспортируемых сервисов и определять политики доступа к URL извне.

Чтобы показать, как добавить входной контроллер в существующий кластер Kubernetes, мы создадим кластер kind, но на этот раз настроим его для перенаправления входящего трафика в порт 80. Этот трафик будет обрабатываться входным контроллером Contour, который позволяет привязать к порту 80 несколько сервисов по именам:

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
networking:
  disableDefaultCNI: true # запретить kindnet
  podSubnet: 192.168.0.0/16 # настроить подсеть Calico по умолчанию
nodes:
- role: control-plane
- role: worker
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    listenAddress: "0.0.0.0"
  - containerPort: 443
    hostPort: 443
    listenAddress: "0.0.0.0"
```



Определяет дополнительную карту отображения портов extraPortMappings для доступа к порту 80 из локального терминала и для пересылки трафика в порт 80 на узлах kind

Дополнительная карта отображения портов extraPortMappings в этом фрагменте позволяет получить доступ к порту 80 из локально-

го терминала и пересылать трафик в порт 80 на узлах `kind`. Обратите внимание, что эта конфигурация работает только с кластерами, имеющими единственный узел, потому что при запуске узлов Kubernetes на основе Docker на локальном компьютере имеется только один порт для предоставления доступа. После создания кластера `_kind_` нужно установить Calico, как показано ниже. В результате получится базовая сеть модулей Pod:

```
$ kubectl create -f
  https://docs.projectcalico.org/archive/v3.16/manifests/
  tigera-operator.yaml

$ kubectl -n kube-system set env daemonset/calico-node
  FELIX_IGNORELOOSERPF=true
$ kubectl -n kube-system set env daemonset/calico-node
  FELIX_XDPENABLED=false
```

Покончив с подготовкой инфраструктуры, можно начинать изучать управление входящим трафиком! В этом разделе мы представим сервис Kubernetes снизу вверх. Как обычно, используем для этого наш верный кластер `kind`. Однако на этот раз мы будем:

- обращаться к сервису внутри кластера;
- использовать входной контроллер Contour для управления этим сервисом по имени хоста.

6.4.2 Настройка простого модуля Pod с веб-сервером

Для начала создадим кластер `kind`, как мы делали это в предыдущих главах, а затем запустим простое веб-приложение. Поскольку в роли входного контроллера часто используется NGINX, на этот раз мы создадим веб-приложение на Python:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    service: example-pod ← Наш сервис будет выбираться по этой метке
spec:
  containers:
    - name: frontend
      image: python
      command:
        - "python"
        - "-m"
        - "SimpleHTTPServer"
        - "8080"
      ports:
        - containerPort: 8080
```

Далее экспортируем `containerPort` через стандартный сервис `ClusterIP`. Это самый простой из всех сервисов Kubernetes; он просто сообщает `kube-проху` о необходимости создать один виртуальный IP-адрес (конечные точки `KUBE_SEP`, которые мы видели выше) в одном из модулей с Python:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    service: example-pod ← Этот Pod является конечной
                           точкой нашего сервиса
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Теперь у нас есть небольшое веб-приложение, получающее трафик от сервиса. Оно обслуживает внутренний трафик через порт 8080, и наш сервис тоже использует этот порт. Попробуем получить к нему доступ локально. Создадим простой образ Docker для работы с сервисами в нашем кластере (этот образ получен из <https://github.com/arunvelsriram/utlis>):

```
apiVersion: v1
kind: Pod
metadata:
  name: sleep
spec:
  containers:
    - name: check
      image: jayunit100/ubuntu-utils
      command:
        - "sleep"
        - "10000"
```

Теперь посмотрим, можно ли получить доступ к нашему сервису изнутри этого образа. Следующая команда `curl` выводит содержимое файла `/etc/passwd` в контейнере. При желании вы можете сохранить файл, например `hello.html`, в корневой каталог вашего контейнера, чтобы сгенерировать более дружелюбное сообщение:

```
$ kubectl exec -t -i sleep curl my-service:8080/etc/passwd
root:x:0:0:root:/root:/bin/bash ← Выведет содержимое файла /etc/passwd
```

Все получилось! Теперь мы знаем, что:

- модуль `Pod` работает и позволяет просмотреть содержимое любых файлов в ОС через порт 8080;

- любой модуль Pod в кластере может обратиться к этому сервису через порт 8080 благодаря созданному выше сервису `my-service`;
- `kube-proxy` пересылает трафик из `my-service` в `example-pod` и создает соответствующие правила для `iptables`;
- провайдер CNI способен создавать необходимые правила маршрутизации (которые мы рассмотрели выше в этой главе) и пересылать трафик между IP-адресами модулей Pod `check` и `example-pod` после переадресации правилами `iptables`.

Теперь допустим, что нам потребовалось получить доступ к этому сервису из внешнего мира. Для этого нужно:

- 1 добавить его как входной ресурс, чтобы Kubernetes API мог настроить входной контроллер для передачи трафика;
- 2 запустить входной контроллер, передающий трафик из внешнего мира во внутренний сервис.

Есть несколько видов входных контроллеров. Наибольшей популярностью пользуются NGINX и Contour. В данном примере мы используем Contour:

```
$ git clone https://github.com/projectcontour/contour.git
$ kubectl apply -f contour/examples/contour
```

Теперь у нас установлен входной контроллер, который будет управлять всем внешним трафиком. Далее добавим запись в файл `/etc/hosts` на локальном компьютере, определяющую IP-адрес предыдущего сервиса:

```
$ echo "127.0.0.1 my-service.local" >> /etc/hosts
```

И создадим входной ресурс:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
    - host: my-service.local ← Имя сервиса на локальном
                               компьютере с адресом 127.0.0.1
      http:
        paths:
          - path: /
            backend:
              serviceName: my-service ← Имя внутреннего сервиса
              servicePort: 8080
                                     Kubernetes
```

Проверить доступность сервиса в кластере `kind` можно, выполнив команду `curl` на локальном компьютере. Она будет работать следующим образом:

- локальный клиент `curl` пытается обратиться к сервису `my-service.local` на порту 80. Имя сервиса разрешается в IP-адрес 127.0.0.1;
- трафик к хосту `localhost` перехватывается узлом Docker в кластере `kind`, который прослушивает порт 80;
- узел Docker пересылает трафик входному контроллеру Contour, который определяет его как попытку получить доступ к `my-service.local`;
- входной контроллер Contour пересылает трафик, адресованный `my-service.local`, внутреннему сервису `my-service`.

Когда этот процесс завершится, мы увидим тот же результат, что и в предыдущем примере. Следующий фрагмент кода иллюстрирует этот процесс, используя сервер Envoy для прослушивания на другом конце. Это связано с тем, что входной контроллер использует Envoy (прокси, используемый Contour) в качестве шлюза для входа в кластер:

```
curl -v http://my-service.local/etc/passwd
* Trying 127.0.0.1...
* TCP_NODELAY set
* Conn to my-service.local (127.0.0.1) port 80
> GET / HTTP/1.1
> Host: my-service.local
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< server: envoy
< date: Sat, 26 Sep 2020 18:32:36 GMT
< content-type: text/html; charset=UTF-8
< content-length: 728
< x-envoy-upstream-service-time: 1
<
root:x:0:0:root:/root:/bin/bash
```

Имя `my-service.local` разрешается в адрес `localhost`

Сервер Envoy отвечает на HTTP-запрос

Теперь мы можем получить доступ к содержимому, обслуживаемому сервером SimpleHTTPServer из стандартной библиотеки Python, используя команды `curl` как на ClusterIP так и на локальном компьютере, запустив сервис входного контроллера, который за кулисами пересылает трафик на ClusterIP. Как отмечалось выше, оригинальный Ingress API в конечном итоге будет заменен более новым Gateway API.

Gateway API в Kubernetes позволяет разделить пользователей в кластере, заменив входной ресурс шлюзами, классами шлюзов и маршрутами, которые могут настраиваться индивидуально. Тем не менее идеи Gateway API и оригинального API управления входным трафиком функционально схожи, и большая часть из того, что мы узнали в этой главе, естественным образом относится и к Gateway API.

Итоги

- Передача трафика в плагинах CNI предполагает маршрутизацию трафика модулей Pod между узлами через сетевые интерфейсы.
- Плагины CNI могут иметь мостовую и немостовую архитектуру, и в обоих случаях способ передачи трафика отличается.
- Сетевые политики реализуются с использованием ранних базовых технологий, таких как Antrea OpenVSwitch (OVS) и Calico iptables.
- Сетевые политики уровня 7 реализуются с помощью входных контроллеров.
- Contour – это входной контроллер, решающий те же задачи, что и интерфейс CNI для модулей Pod на уровне 7, и способный работать с любым провайдером CNI.
- В будущем на смену Ingress API придет более гибкой Gateway API, но все, что вы узнали в этой главе, сохранит свою актуальность.

Хранилища в модулях Pod и CSI

В этой главе:

- введение в виртуальную файловую систему (VFS);
- исследование внутренних (in-tree) и внешних (out-of-tree) провайдеров для Kubernetes;
- запуск динамического хранилища в кластере kind с несколькими контейнерами;
- определение интерфейса контейнерного хранилища (CSI).

Организация хранилищ – сложная тема, и мы не ставили своей целью описать в этой книге все типы хранилищ, доступные современному разработчику приложений. Вместо этого мы возьмем для примера конкретную задачу – реализация возможности хранения файлов в модуле Pod – и решим ее. Файл должен сохраняться в период от остановки до повторного запуска контейнера и должен быть доступен новым узлам в кластере. Встроенные тома хранилища по умолчанию, с которыми мы уже познакомились в этой книге, не подходят для данной задачи:

- наш модуль Pod не может полагаться на `hostPath`, потому что сам узел может не иметь уникального доступного для записи каталога на своем диске;
- наш модуль Pod также не может полагаться на `emptyDir`, потому что фактически ему требуется база данных, а для баз данных непозволительно терять информацию, хранящуюся в эфемерном томе;

- наш модуль Pod может хранить сертификаты или пароли для доступа к таким сервисам, как базы данных, в секретах Secret, но этот модуль не считается томом, когда речь идет о приложениях, работающих в Kubernetes;
- наш модуль Pod может записывать данные на уровне своей файловой системы контейнера, но такой подход работает очень медленно и не рекомендуется, когда предполагается записывать большие объемы информации. А кроме того, этот подход тоже не годится, потому что данные исчезают с перезапуском модуля Pod!

Таким образом, мы наткнулись на совершенно новый вид хранилищ в Kubernetes, способных удовлетворить потребности разработчика. Приложениям Kubernetes, как и обычным облачным приложениям, часто нужна возможность монтировать тома EBS, ресурсы NFS или корзины S3 внутри контейнеров, а также читать или записывать в них данные. Для решения этой задачи нам понадобится облачная модель данных и соответствующий API. Kubernetes позволяет представить эту модель данных, предлагая такие понятия, как том хранилища PersistentVolume (PV), запрос на хранилище PersistentVolumeClaim (PVC) и класс хранилища StorageClass:

- тома хранилищ PV дают возможность управлять дисковыми томами в среде Kubernetes;
- запросы на хранилище PVC определяют требования приложений (модулей Pod) к этим томам и обрабатываются Kubernetes API;
- класс хранилища StorageClass дает возможность получать тома, не зная, как они реализованы. Это позволяет определять запросы PVC, не зная точно, какой тип тома PersistentVolume используется за кулисами.

Классы хранилищ StorageClass дают приложениям возможность запрашивать тома или типы хранилищ, отвечающие требованиям конечного пользователя, *декларативно*, что позволяет определять классы, удовлетворяющие различным потребностям, таким как:

- сложные требования к уровню обслуживания (что хранить, как долго хранить и что не хранить);
- высокие требования к производительности (приложения пакетной обработки или приложения с малой задержкой);
- безопасность в многопользовательских окружениях (доступ пользователей к определенным томам).

Имейте в виду, что многим контейнерам (например, серверу CFSSL для управления сертификатами приложений) может потребоваться совсем небольшой объем хранилища, но им может понадобится дополнительное пространство на случай перезапуска или для кеширования сертификатов. Более подробно о высокоуровневых концепциях управления классами хранения мы поговорим в следующей главе. Если вы только начинаете осваивать Kubernetes, то вам может быть интересно, могут ли модули Pod поддерживать свое состояние без тома.

Модули Pod сохраняют состояние?

В общем случае – нет. Не забывайте, что модуль Pod почти всегда является эфемерной конструкцией. В некоторых случаях (например, когда определен объект StatefulSet) некоторые аспекты модуля (например, IP-адрес или локально смонтированный каталог хоста) могут сохраняться после перезапуска.

Если Pod по какой-либо причине выходит из строя, то диспетчер контроллеров Kubernetes (Kubernetes Controller Manager, КСМ) воссоздаст его. Создавая новый Pod, планировщик Kubernetes выбирает узел, удовлетворяющий требованиям. Следовательно, эфемерный характер хранилища позволяет принимать решения в режиме реального времени, что совершенно необходимо для гибкого управления большими парками приложений.

7.1 Небольшое отступление: виртуальная файловая система (VFS) в Linux

Прежде чем перейти к абстракциям хранилищ модулей Pod в Kubernetes, стоит отметить, что сама операционная система тоже поддерживает эти абстракции. Фактически сама *файловая система* является абстракцией сложной схемы, соединяющей приложения с простым набором API-интерфейсов, которые мы видели раньше. Возможно, вы уже знаете, что доступ к файлу подобен доступу к любому другому API. Управление файлами в Linux реализовано в форме набора простых команд, в том числе:

- `read()` – читает несколько байтов из открытого файла;
- `write()` – записывает несколько байтов в открытый файл;
- `open()` – создает и/или открывает файл для чтения и записи;
- `stat()` – возвращает некоторые основные сведения о файле;
- `chmod()` – меняет права доступа к файлу для пользователя или группы.

Все эти команды передаются так называемой *виртуальной файловой системе* (Virtual Filesystem, VFS), которая в большинстве случаев является оберткой вокруг BIOS системы. В облаке и в случае с FUSE (Filesystem in Userspace – файловая система в пространстве пользователя) Linux VFS является оберткой вокруг механизмов, которые в конечном счете ведут к сетевым вызовам. Даже если данные записываются на внешний диск, доступ к этим данным все равно осуществляется через VFS. Единственная разница в том, что для записи на удаленный диск VFS использует своего клиента NFS, FUSE или другой файловой системы, как показано на рис. 7.1, где различные операции записи, выполняемые в контейнере, фактически следуют через VFS API:

- вызываемая хранилищем Docker или CRI, VFS посылает запросы модулю отображения устройств или OverlayFS, который в свою очередь передает трафик локальным устройствам через BIOS системы;
- вызываемая хранилищем в инфраструктуре Kubernetes, VFS посылает запросы локальным дискам на узле;
- вызываемая приложениями, VFS часто посылает запросы по сети, особенно в «настоящих» кластерах Kubernetes, работающих в облаке или в центре обработки данных с большим количеством компьютеров. Именно поэтому нежелательно использовать тома локальных типов.

А как это происходит в Windows?

На узлах Windows агент kubelet монтирует и предоставляет хранилище для контейнеров так же, как в Linux. На узлах с Windows обычно запускается CSI Proxy (<https://github.com/kubernetes-csi/csi-proxy>), выполняющий низкоуровневые вызовы ОС Windows, которая монтирует и размонтирует тома по требованию kubelet. В экосистеме Windows существуют те же понятия и абстракции файловой системы (https://ru.wikipedia.org/wiki/Installable_File_System).

В любом случае не требуется разбираться в деталях Linux API доступа к хранилищам, чтобы смонтировать PersistentVolume в Kubernetes. Однако понимать основы файловых систем полезно, потому что в конечном итоге ваши модули Pod будут взаимодействовать с этими низкоуровневыми API. Теперь вернемся к Kubernetes-ориентированному представлению хранилищ для модулей Pod.

7.2 Три вида хранилищ для Kubernetes

Термин *хранилище* слишком широкий, поэтому, прежде чем углубиться в обсуждение, выделим типы хранилищ, которые обычно вызывают проблемы в Kubernetes:

- *хранилище Docker/containerd/CRI* – файловая система с копированием при записи, используется контейнерами. Контейнерам требуются специальные файловые системы во время выполнения, потому что запись выполняется на уровне VFS (именно поэтому, например, можно запустить `rm -rf /tmp` в контейнере, не рискуя удалить что-то на хосте). Обычно в окружении Kubernetes используются такие файловые системы, как btrfs, overlay или overlay2;
- *хранилище в инфраструктуре Kubernetes* – тома hostPath или Secret, которые используются на отдельных узлах для локального обмена информацией (например, для хранения секрета, кото-

рый будет смонтирован в модуле Pod или в каталоге, откуда вызывается плагин хранилища или сети);

- *хранилище для приложений* – тома хранилища, которые модули Pod используют в кластере Kubernetes. Если модуль Pod должен записать данные на диск, то ему необходимо смонтировать том хранилища, для чего в объявление Pod добавляются соответствующие определения. Обычно для томов хранилищ используются файловые системы OpenEBS, NFS, GCE, EC2, постоянные диски vSphere и т. д.

На рис. 7.1, дополненном рис. 7.2, мы постарались как можно нагляднее показать все три типа хранилищ и основные этапы запуска модуля Pod. Выше мы рассматривали только этапы запуска Pod, связанные с CNI. Не забывайте, что планировщик выполняет ряд проверок перед тем, как Pod подтвердит готовность хранилища. Затем, перед запуском Pod, агент kubelet и провайдер CSI монтируют внешние тома приложений на узле для использования модулем Pod. В процессе запуска Pod может записать данные в свою собственную OverlayFS. Например, он может использовать каталог /tmp для временного хранения данных. Наконец, по завершении процедуры за-

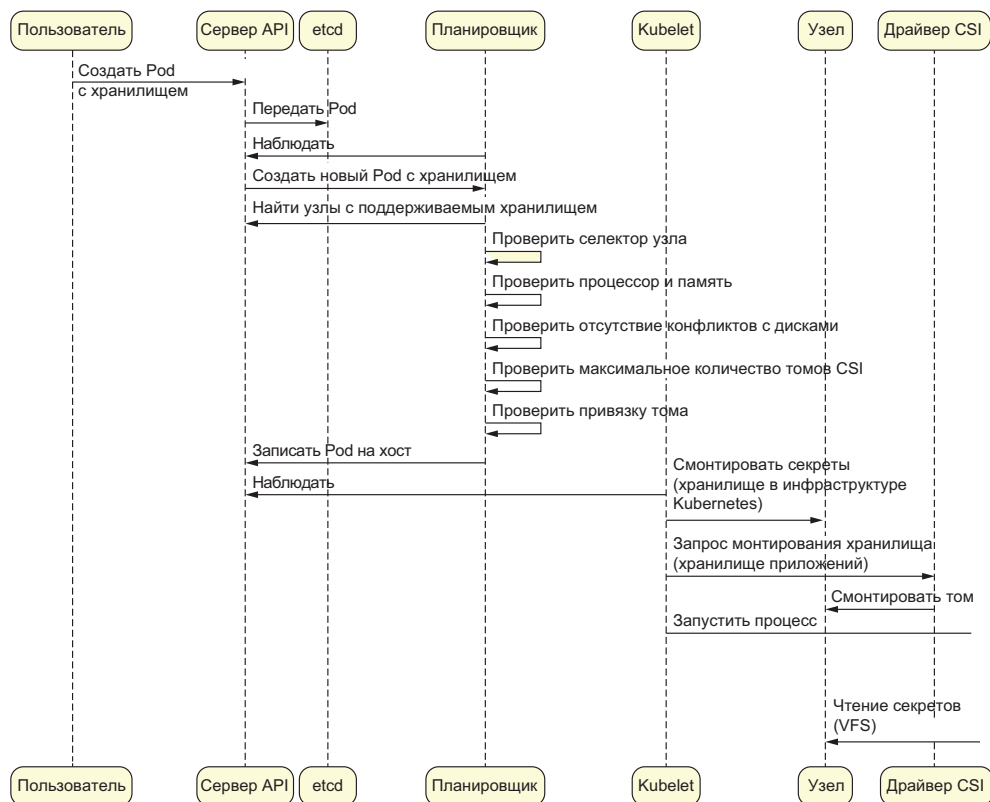


Рис. 7.1 Три типа хранилищ во время запуска модуля Pod

пуска Pod может читать локальные тома и записывать данные в другие удаленные тома.

Первая диаграмма заканчивается драйвером CSI, но в последовательности запуска Pod с хранилищами есть множество других уровней. На рис. 7.2 мы видим, что CSIDriver, containerd, многоуровневая файловая система и сам том CSI используются процессами в Pod. В частности, когда kubelet запускает процесс, он посылает сообщение демону containerd, который затем создает новый слой в файловой системе, доступный для записи. После запуска процессу в контейнере необходимо прочитать секреты из смонтированных файлов. Таким образом, в одном модуле Pod выполняется множество видов вызовов хранилищ, каждый из которых имеет свою семантику и назначение в жизненном цикле приложения.

Монтирование тома CSI – одно из последних событий, происходящих перед запуском модуля Pod. Чтобы понять, как выполняется этот шаг, нужно сделать небольшое отступление и посмотреть, как организованы файловые системы в Linux.

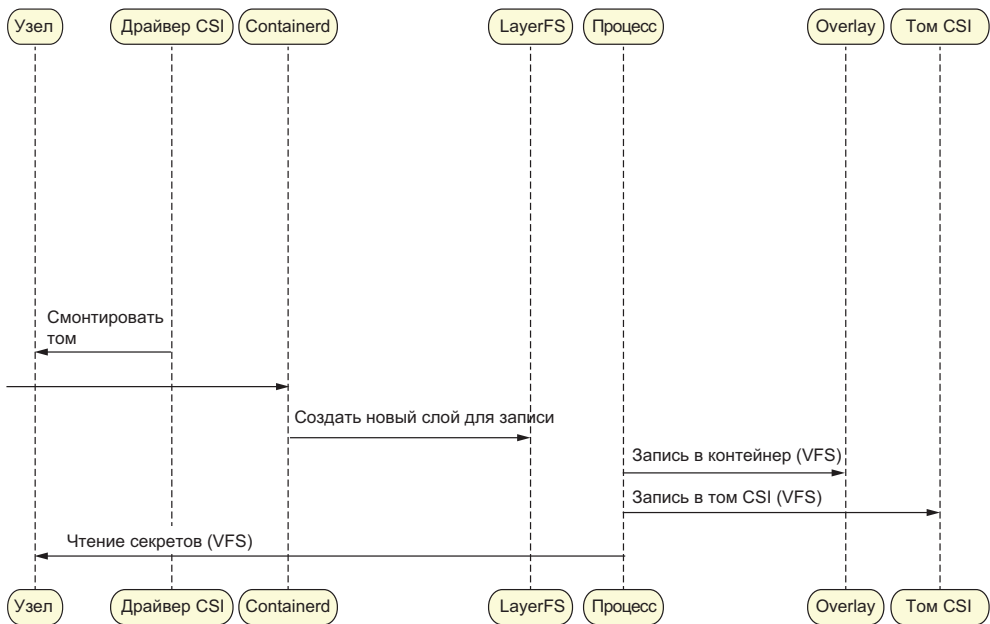


Рис. 7.2 Три типа хранилищ во время запуска модуля Pod, часть 2

7.3 Создание PVC в кластере kind

Но хватит теории; давайте добавим хранилище для использования приложениями в простом модуле Pod с NGINX. Выше мы определили понятия PV, PVC и StorageClass. Теперь посмотрим, как с их помощью

создать каталог, который модуль Pod мог бы использовать для хранения некоторых файлов:

- PV (PersistentVolume – том хранилища) создается провайдером динамических хранилищ, работающим в нашем кластере kind. Это контейнер, который создает хранилище для Pod, выполняя запрос PVC;
- PVC (PersistentVolumeClaim – запрос на хранилище) недоступен до готовности тома PersistentVolume, потому что планировщик должен убедиться, что сможет смонтировать хранилище в пространстве имен модуля Pod перед его запуском;
- kubelet не запустит Pod, пока VFS не смонтирует доступный для записи PVC в пространство имен файловой системы Pod.

К счастью, кластер kind поставляется вместе с провайдером хранилищ. Давайте посмотрим, что произойдет, если запросить запуск Pod с новым PVC, еще не созданным и не имеющим связанного тома в нашем кластере. Проверить, какие провайдеры хранилищ доступны в кластере Kubernetes, можно командой `kubectl get sc`:

```
$ kubectl get sc
NAME                                PROVISIONER              RECLAIMPOLICY
standard (default)                 rancher.io/local-path    Delete

VOLUMEBINDINGMODE                 ALLOWVOLUMEEXPANSION     AGE
WaitForFirstConsumer             false                    9d
```

Чтобы показать, как модули Pod поддерживают хранение общих данных для контейнеров и как смонтировать несколько точек хранения с разной семантикой, мы запустим Pod с двумя контейнерами и двумя томами. В итоге:

- контейнеры в Pod смогут совместно использовать общую информацию;
- постоянное хранилище может быть создано в kind по запросу с помощью динамического поставщика `hostPath`;
- любой контейнер сможет использовать несколько томов, смонтированных в модуле Pod.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100k
  ---
apiVersion: v1
kind: Pod
```

Папка, совместно используемая со вторым контейнером

```

metadata:
  name: nginx
spec:
  containers:
    - image: busybox
      name: busybox
      volumeMounts:
        - mountPath: /shared
          name: shared
    - image: nginx
      imagePullPolicy: Always
      name: nginx
      ports:
        - containerPort: 80
          protocol: TCP
      volumeMounts:
        - mountPath: /var/www
          name: dynamic1
        - mountPath: /shared
          name: shared
  volumes:
    - name: dynamic1
      persistentVolumeClaim:
        claimName: dynamic1
    - name: shared
      emptyDir: {}

```

Определяет том динамического хранилища для второго контейнера в дополнение к совместно используемой папке в первом контейнере

Монтирует ранее созданный том

Поскольку раздел тома находится за пределами раздела контейнера, одни и те же данные доступны для чтения нескольким модулям Pod

Общий том, доступный обоими контейнерам

Объем запрашиваемого хранилища; определяется запросом PVC

```

$ kubectl create -f simple.yaml
pod/nginx created

```

```

$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   0/1     Pending   0           3s

```

Первое состояние Pending (ожидание) обусловлено отсутствием тома для нашего модуля Pod

```

$ kubectl get pods
NAME    READY   STATUS             RESTARTS   AGE
nginx   0/1     ContainerCreating   0           5s

```

Конечное состояние Running (работает) означает, что том существует и доступен для Pod; теперь kubelet может запустить Pod

```

$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          13s

```

Теперь можно создать файл в первом контейнере, выполнив простую команду, например `echo a > /shared/ASDF`, и посмотреть результат во втором контейнере в папке `emptyDir` с именем `/shared/`, доступной в обоих контейнерах:

```

$ kubectl exec -i -t nginx -t busybox -- /bin/sh
Defaulting container name to busybox.
Use kubectl describe pod/nginx -n default to see the containers in this pod.
# cat /shared/ASDF
a

```


Теперь у нас есть модуль Pod с двумя томами: один временный и один постоянный. Как это получилось? Если заглянуть в журнал в кластере kind для local-path-provisioner, то все станет понятно:

```
$ kubectl logs local-path-provisioner-77..f-5fg2w
-n local-path-storage
controller.go:1027] provision "default/dynamic2" class "standard":
    volume "pvc-ddf3ff41-5696-4a9c-baae-c12f21406022"
    provisioned
controller.go:1041] provision "default/dynamic2" class "standard":
    trying to save persistentvolume "pvc-ddf3ff41-5696-4a9c-baae-
    c12f21406022"
controller.go:1048] provision "default/dynamic2" class "standard":
    persistentvolume "pvc-ddf3ff41-5696-4a9c-baae-c12f21406022" saved
controller.go:1089] provision "default/dynamic2" class "standard": succeeded
event.go:221] Event(v1.ObjectReference{Kind:"PersistentVolumeClaim",
    Namespace:"default", Name:"dynamic2",
    UID:"ddf3ff41-5696-4a9c-baae-
    c12f21406022", APIVersion:"v1", ResourceVersion:"11962",
    FieldPath:""})
): type: 'Normal' reason:
    'ProvisioningSucceeded'
Successfully provisioned volume
pvc-ddf3ff41-5696-4a9c-baae-c12f21406022
```

Контейнер все время продолжает работать как контроллер в нашем кластере. Когда он видит, что нам нужен том с именем dynamic2, он создает его. В случае успеха Kubernetes сам привязывает том к PVC. Если существует том, удовлетворяющий требованиям PVC, в ядре Kubernetes возникает событие привязки.

После этого планировщик Kubernetes проверяет возможность вернуть на узле этот конкретный PVC и в случае успеха Pod переходит из состояния Pending (ожидание) в состояние ContainerCreating (контейнер создан), как было показано выше. Как вы уже знаете, после перехода в состояние ContainerCreating агент kubelet настраивает контрольные группы и точки монтирования для Pod, после чего переводит его в состояние Running (выполнение). Автоматическое создание тома (мы не определяли PersistentVolume вручную) может служить наглядным примером динамического создания хранилища в кластере. Вот как можно получить список динамически созданных томов:

```
$ kubectl get pv
```

NAME	CAPACITY	ACCESS
pvc-74879bc4-e2da-4436-9f2b-5568bae4351a	100k	RWO

RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS
Delete	Bound	default/dynamic1	standard

Присмотревшись, можно заметить, что для этого тома используется класс хранилища StorageClass standard. Фактически класс хранили-

ща определяет, как Kubernetes смог создать этот том. Когда определен класс `standard` или `default`, PVC, для которого не определен класс хранилища, автоматически настраивается на получение PVC по умолчанию, если он существует. На самом деле это происходит с помощью *контроллера доступа* (admission controller), который предварительно модифицирует новые модули Pod, поступающие на сервер API, добавляя к ним метку класса хранилища `default`. При наличии этой метки инструмент подготовки томов (provisioner), работающий в кластере (в нашем случае он называется `local-path-provisioner` и входит в состав `kind`), автоматически обнаруживает запрос модуля Pod на получение хранилища и немедленно создает том:

```
$ kubectl get sc -o yaml
apiVersion: v1
items:
- apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"storage.k8s.io/v1",
          "kind":"StorageClass", "metadata":{"
            "annotations":{"
              "storageclass.kubernetes.io/is-default-class": "true"}
            },
            "name":"standard"
          },
          "provisioner":"rancher.io/local-path",
          "reclaimPolicy":"Delete",
          "volumeBindingMode":"WaitForFirstConsumer"}
        storageclass.kubernetes.io/is-default-class: "true"
  name: standard
  provisioner: rancher.io/local-path
  kind: List
```

Класс `is-default-class` делает этот том доступным для модулей Pod, которым требуется хранилище, без необходимости явно запрашивать класс хранилища

Кластер может поддерживать несколько классов хранилищ

Выяснив, что модули Pod могут иметь много разных типов хранилищ, становится ясно, что для их поддержки нужен подключаемый поставщик хранилищ для Kubernetes. Эту задачу решает интерфейс CSI (<https://kubernetes-csi.github.io/docs/>).

7.4 Интерфейс контейнерного хранилища (CSI)

Kubernetes CSI определяет интерфейс (рис. 7.3), чтобы поставщики решений для организации хранилищ могли легко подключаться к любому кластеру Kubernetes и предоставлять приложениям широкий спектр возможностей хранения данных. Это альтернатива внутренним хранилищам, драйверы которых агент `kubelet` добавляет сам в процессе запуска Pod.

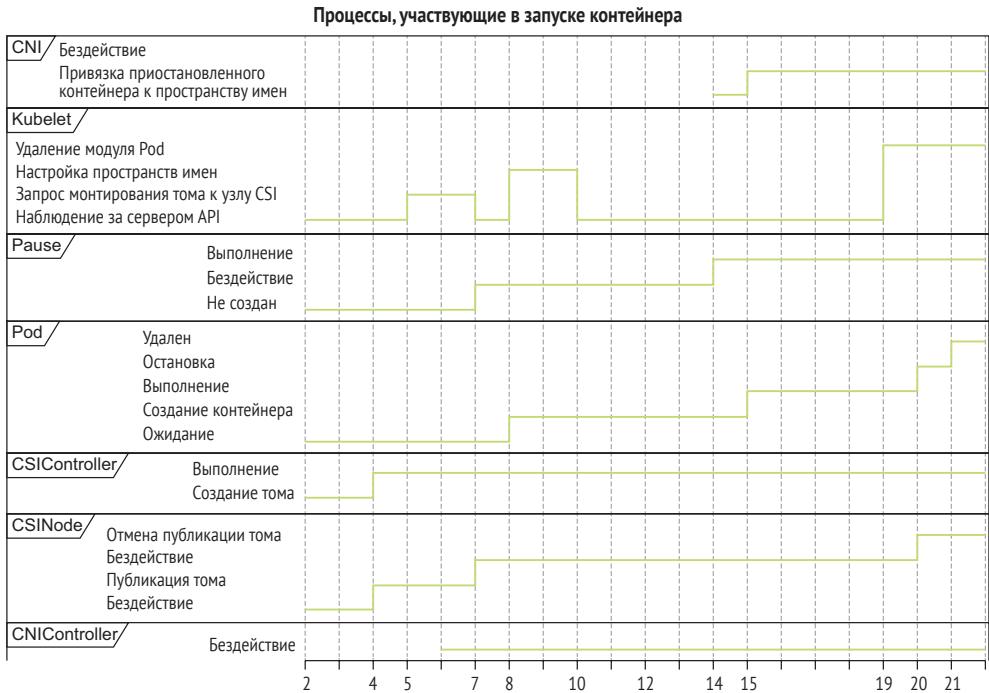


Рис. 7.3 Архитектура модели Kubernetes CSI

Цель определения CSI – упростить управление решениями хранения данных с точки зрения производителя. Чтобы сформулировать эту задачу, рассмотрим базовую реализацию хранилища для нескольких Kubernetes PVC:

- драйвер vSphere CSI может создавать объекты PersistentVolume на основе VMFS или vSAN;
- файловые системы, такие как GlusterFS, имеют драйверы CSI, позволяющие запускать тома распределенным образом в контейнерах;
- в Pure Storage есть драйвер CSI, который напрямую создает тома в дисковом массиве Pure Storage.

Многие другие поставщики тоже предоставляют решения на основе CSI для Kubernetes. Прежде чем описать, как CSI упрощает эту задачу, мы кратко осветим проблему внутреннего провайдера, входящего в состав Kubernetes. Этот CSI был в значительной степени ответом на проблемы, связанные с управлением томами хранилищ, возникающие во внутренней модели хранения.

7.4.1 Проблема внутреннего провайдера

С момента создания Kubernetes поставщики потратили много времени на включение функциональной совместимости в его кодовую базу.

Как результат, поставщики различных типов хранилищ должны были вносить код поддержки в само ядро Kubernetes! В кодовой базе Kubernetes все еще есть остатки такого кода, как можно видеть на примере GlusterFS (<http://mng.bz/J1NV>):

```
package glusterfs
```

```
import (
    "context"
    ...
    gcli "github.com/heketi/heketi/client/api/go-client"
    gapi "github.com/heketi/heketi/pkg/glusterfs/api"
```

Импорт пакета GlusterFS API (Heketi – это REST API для Gluster) фактически подразумевает, что Kubernetes знает о GlusterFS и зависит от него. Заглянув немного дальше, можно увидеть, как проявляется эта зависимость:

```
func (p *glusterfsVolumeProvisioner) CreateVolume(gid int)
(r *v1.GlusterfsPersistentVolumeSource, size int,
 volID string, err error) {
    ...
    // GlusterFS/heketi создает тома, объем которых измеряется гигабайтами.
    sz, err := volumehelpers.RoundUpToGiBInt(capacity)
    ...
    cli := gcli.NewClient(p.url, p.user, p.secretValue)
    ...
}
```

Пакет volume в Kubernetes вызывает GlusterFS API для создания нового тома. Также можно увидеть аналогичный код других поставщиков, таких как VMware vSphere. Фактически многие поставщики, включая VMware, Portworx, ScaleIO и др., имеют собственные каталоги в пакете pkg/volume Kubernetes. Это очевидный антишаблон для любого проекта с открытым исходным кодом, потому что имеет место объединение кода конкретного поставщика с кодом фреймворка. Это накладывает очевидные ограничения:

- пользователи должны согласовать свою версию Kubernetes с версиями драйверов хранилищ;
- поставщики должны постоянно передавать свой код в репозиторий Kubernetes, чтобы обеспечить актуальность своих решений.

Эти два сценария весьма неустойчивы во времени, из-за чего возникла потребность в стандарте, определяющем возможность создания, мониторинга и управления жизненным циклом внешних томов. По аналогии с интерфейсом CNI, рассматривавшимся выше, стандарт CSI гарантирует нормальную работу DaemonSet на всех узлах, монтирующих тома (во многом подобно агентам CNI, которые реализуют внедрение IP для пространства имен). Кроме того, CSI позволяет легко заменить хранилище одного типа другим и даже одновременно запустить несколько хранилищ разных типов (что не так

просто сделать с сетями), потому что определяет конкретное соглашение об именовании томов.

Обратите внимание, что проблема внутренних провайдеров характерна не только для хранилищ. Интерфейсы CRI, CNI и CSI появились из внешнего кода, который долгое время включался в ядро Kubernetes. В первых версиях в ядро Kubernetes входили такие инструменты, как Docker, Flannel и многие другие файловые системы. Они постепенно удаляются, и CSI является лишь одним из ярких примеров, как после создания надлежащих интерфейсов код может исключаться из ядра и перемещаться во внешние плагины. Однако в Kubernetes существует еще довольно много кода, зависящего от поставщика, и могут потребоваться годы, чтобы полностью отделить эти технологии.

7.4.2 CSI как спецификация, работающая внутри Kubernetes

На рис. 7.4 показан процесс инициализации PVC с драйвером CSI. Он выглядит гораздо более прозрачным, чем то, что мы видим в GlusterFS, где разные компоненты выполняют разные задачи дискретным образом.

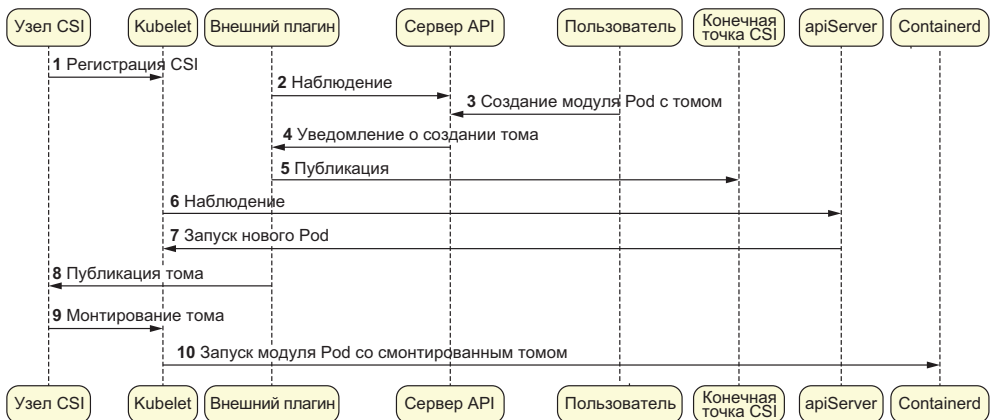


Рис. 7.4 Инициализация PVC с драйвером CSI

Спецификация CSI определяет общий набор функций, позволяет определить службу поддержки хранилища без указания конкретной реализации. В этом разделе мы рассмотрим некоторые аспекты этого интерфейса в контексте самого Kubernetes. Определяемые им операции делятся на три основные категории: сервисы идентификации, сервисы контроллеров и сервисы узлов. В основе всего этого лежит понятие контроллера, который согласовывает потребность в хранилище с провайдером и плоскостью управления Kubernetes, выполняя запрос на динамическое создание хранилища. Давайте кратко рассмотрим эти три категории:

- *сервисы идентификации* – позволяют плагину идентифицировать себя (предоставить метаданные о себе), чтобы плоскость управления Kubernetes могла подтвердить, что плагин хранилища определенного типа запущен и доступен для данного типа томов;
- *сервисы узлов* – позволяют агенту kubelet взаимодействовать с локальным сервисом, выполняющим операции, специфичные для провайдера хранилища. Например, сервис узла провайдера CSI может запустить двоичную программу конкретного поставщика, получив запрос на монтирование хранилища определенного типа. Запрос выполняется через сокет по протоколу GRPC;
- *сервисы контроллеров* – реализуют обработку событий создания, удаления и других, связанных с управлением жизненным циклом тома хранилища. Имейте в виду: чтобы NodeService имел какую-либо ценность, используемая система хранения должна сначала *создать* том, который в нужный момент сможет подключить kubelet. Таким образом, сервисы контроллеров играют связующую роль, соединяя Kubernetes с поставщиком решения хранения данных. Как и следовало ожидать, это реализуется путем перехвата запросов к Kubernetes API на выполнение операций с томами.

В следующем фрагменте кода приводится краткий обзор спецификации CSI. Здесь показаны не все методы. Более полную информацию вы найдете в репозитории (<http://mng.bz/y4V7>):

```

service Identity {
  rpc GetPluginInfo(GetPluginInfoRequest)
  rpc GetPluginCapabilities(GetPluginCapabilitiesRequest)
  rpc Probe (ProbeRequest)
}

service Controller {
  rpc CreateVolume (CreateVolumeRequest)
  rpc DeleteVolume (DeleteVolumeRequest)
  rpc ControllerPublishVolume (ControllerPublishVolumeRequest)
}

service Node {
  rpc NodeStageVolume (NodeStageVolumeRequest)
  rpc NodeUnstageVolume (NodeUnstageVolumeRequest)
  rpc NodePublishVolume (NodePublishVolumeRequest)
  rpc NodeUnpublishVolume (NodeUnpublishVolumeRequest)
  rpc NodeGetInfo (NodeGetInfoRequest)
  ...
}

```

Сервис идентификации сообщает Kubernetes, какие типы томов могут создаваться контроллерами, работающими в кластере

Методы Create и Delete вызываются до того, как узел сможет смонтировать том в Pod, реализуя динамическое хранилище

Сервис Node – это часть CSI, которая работает в kubelet и монтирует ранее созданный том в определенный Pod по запросу

7.4.3 CSI: как работает драйвер хранилища

Плагин хранилища CSI разбивает действия, выполняемые для подключения хранилища к Pod, на три этапа: регистрацию драйвера хранилища, запрос тома и его публикацию.

Регистрация драйвера хранилища выполняется через Kubernetes API. Фреймворку сообщается, как обращаться с этим конкретным драйвером (должны ли произойти определенные события, прежде чем том хранилища станет доступен для записи) и когда агенту kubelet будет доступен определенный тип хранилища. Имя драйвера CSI играет важную роль, как мы вскоре увидим:

```
type CSIDriverInfoSpec struct {  
    Name string `json:"name"`
```

Получив запрос на создание тома (например, через вызов API решения NAS за 200 000 долл.), механизм поддержки хранилищ приступает к созданию тома, вызывая функцию CreateVolume, представленную выше. На самом деле CreateVolume вызывается (обычно) отдельным сервисом, известным как *внешний инструмент подготовки* (external provisioner), который, скорее всего, никак не связан с DaemonSet. Скорее, это стандартный Pod, наблюдающий за сервером Kubernetes API и отвечающий на запросы создания томов, который вызывает API поставщика решения хранилищ. Этот сервис просматривает созданные объекты PVC, а затем вызывает функцию CreateVolume зарегистрированного драйвера CSI. Он знает, какой драйвер вызвать, потому что эта информация включена в имя тома. (Именно поэтому так важно правильно заполнить поле name.) В этом случае запрос тома в драйвере CSI выполняется отдельно от его монтирования.

В момент публикации том подключается (монтируется) к модулю Pod. Эта операция выполняется драйвером хранилища CSI, который обычно находится на каждом узле кластера. Публикация тома – это просто необычное название операции монтирования тома в место, указанное агентом kubelet, чтобы Pod смог записать в него данные. kubelet отвечает за запуск контейнера в Pod с правильными пространствами имен монтирования для доступа к этому каталогу.

7.4.4 Привязка точек монтирования

Возможно, вы помните, что выше мы определили монтирование как простую операцию в Linux, связывающую монтируемый том с каталогом в дереве /. Это фундаментальная часть контракта между плагином и kubelet, который определяется интерфейсом CSI. В Linux конкретная операция, делающая каталог доступным для модуля Pod (или любого другого процесса посредством зеркалирования каталога), называется *привязкой точки монтирования* (bind mount). Таким образом, в любой среде хранения, предоставляемой CSI, Kubernetes

получает несколько запущенных сервисов, которые координируют взаимодействия вызовов API для достижения конечной цели – монтирования внешних томов в модули Pod.

Драйверы CSI – это набор контейнеров, часто поддерживаемых поставщиками, поэтому сам kubelet должен поддерживать возможность монтирования изнутри контейнера. Это известно как *распространение монтирования* (mount propagation) и является важной частью низкоуровневых требований к Linux, соответствие которым необходимо для правильной работы Kubernetes.

7.5 Краткий обзор действующих драйверов CSI

В заключение приведем несколько конкретных примеров реальных провайдеров CSI. Для этого необходим действующий кластер, поэтому вместо создания пошагового руководства, описывающего, как воспроизвести поведение CSI (как мы делали это с провайдерами CNI), мы просто поделимся фрагментами журналов различных компонентов провайдера CSI. Это поможет вам увидеть, как реализованы интерфейсы, описанные в этой главе, и происходит их мониторинг в режиме реального времени.

7.5.1 Контроллер

Контроллер – это мозг любого драйвера CSI, пересылающий запросы к хранилищу внутренним провайдерам, такими как vSAN, EBS и т. д. Интерфейс, который он реализует, должен позволять создавать, удалять и публиковать тома на лету для использования нашими модулями Pod. Мы можем увидеть, как осуществляется непрерывный мониторинг сервера Kubernetes API, если заглянуть непосредственно в журнал работающего контроллера vSphere CSI:

```
I0711 05:38:07.057037      1 controller.go:819] Started provisioner
    controller csi.vsphere.vmware.com_vsphere-csi-controller-...-
I0711 05:43:25.976079      1 reflector.go:389] sigs.k8s.io/sig-
    storage-lib-external-provisioner/controller/controller.go:807:
    Watch close - *v1.StorageClass total 0 items received
I0711 05:45:13.975291      1 reflector.go:389] sigs.k8s.io/sig-
    storage-lib-external-provisioner/controller/controller.go:804:
    Watch close - *v1.PersistentVolume total 3 items received
I0711 05:46:32.975365      1 reflector.go:389] sigs.k8s.io/sig-
    storage-lib-external-provisioner/controller/controller.go:801:
    Watch close - *v1.PersistentVolumeClaim total 3 items received
```

После получения запроса PVC контроллер может запросить хранилище непосредственно из vSphere. Затем тома, созданные vSphere, могут синхронизировать метаданные между PVC и PV, чтобы убедиться в возможности монтирования PVC. После этого в игру вступают

ет узел CSI (планировщик сначала убедится в готовности узла CSI для vSphere в месте назначения Pod).

7.5.2 Интерфейс узла

Интерфейс узла отвечает за взаимодействие с агентом kubelet и мониторинг хранилища в модули Pod. Убедиться в этом можно, взглянув в текущие журналы используемых томов. Выше мы пытались запустить драйвер NFS CSI, чтобы выявить низкоуровневые взаимодействия с VFS в Linux. Теперь, познакомившись с интерфейсом CSI, вернемся назад и еще раз посмотрим, как драйвер NFS CSI действует в рабочем окружении.

Первым делом посмотрим, как CSI-плагины NFS и vSphere используют сокет для связи с kubelet, точнее, как вызываются компоненты интерфейса узла. Изучая детали контейнера узла CSI, мы должны увидеть примерно такую картину:

```
$ kubectl logs
➡ csi-nodeplugin-nfsplugin-dbj6r -c nfs
I0711 05:41:02.957011 1 nfs.go:47]
➡ Driver: nfs.csi.k8s.io version: 2.0.0
I0711 05:41:02.963340 1 server.go:92] Listening for connections on address:
  &net.UnixAddr{
    Name: "/plugin/csi.sock",
    Net: "unix"}
    Имя драйвера CSI
    Канал, посредством которого kubelet
    взаимодействует с плагинами CSI

$ kubectl logs csi-nodeplugin-nfsplugin-dbj6r
-c node-driver-registrar
I0711 05:40:53.917188 1 main.go:108] Version: v1.0.2-rc1-0-g2edd7f10
I0711 05:41:04.210022 1 main.go:76] Received GetInfo call: &InfoRequest{}
```

Имена драйверов CSI играют важную роль, потому что они являются частью протокола CSI. `csi-nodeplugin` выводит свою версию при запуске. Обратите внимание, что каталог плагинов `csi.sock` служит общим каналом, который kubelet использует для взаимодействий с плагинами CSI:

```
$ kubectl logs -f vsphere-csi-node-6hh7l -n kube-system
➡ -c vsphere-csi-node
{"level": "info", "time": "2020-07-08T21:07:52.623267141Z",
 "caller": "logger/logger.go:37",
 "msg": "Setting default log level to :\\\"PRODUCTION\\\""}
{"level": "info", "time": "2020-07-08T21:07:52.624012228Z",
 "caller": "service/service.go:106",
 "msg": "configured: \\\"csi.vsphere.vmware.com\\\"
 with clusterFlavor: \\\"VANILLA\\\"
 and mode: \\\"node\\\"",
 "TraceId": "72fff590-523d-46de-95ca-fd916f96a1b6"}
level=info msg="identity service registered"
level=info msg="node service registered"
    Показывает, что драйвер
    зарегистрирован
```

```
level=info msg=serving endpoint=
➡ "unix:///csi/csi.sock" ←
```

Показывает, что используется
сокет CSI

На этом мы завершаем рассмотрение интерфейса CSI и причин его появления. В отличие от других компонентов Kubernetes этот интерфейс сложно обсуждать, не имея под рукой действующего кластера с реальными рабочими нагрузками. В качестве самостоятельного упражнения мы настоятельно рекомендуем установить CSI-провайдера NFS (или любой другой драйвер CSI) в любом имеющемся у вас кластере и попробовать оценить, замедляется ли создание томов с течением времени, и если да, то выяснить, в чем причина такого замедления.

Мы не включили в эту главу живой пример драйвера CSI, потому что большинство современных драйверов CSI, использующихся в промышленных кластерах, не могут работать в kind. Однако если вы поняли, что подготовка томов отличается от их монтирования, то можете считать, что вы достаточно хорошо подготовлены к отладке сбоев CSI в производственной системе, рассматривая эти две независимые операции как разные источники сбоев.

7.5.3 CSI в операционных системах, отличных от Linux

Как и CNI, интерфейс CSI не зависит от операционной системы; однако его реализация более естественна для Linux, где поддерживается возможность запуска привилегированных контейнеров. Как и в случае с сетями, способ реализации CSI в Linux немного отличается. Например, пользующиеся Kubernetes в Windows могут найти полезным проект CSI-прокси (<https://github.com/kubernetes-csi/csi-proxy>), запускающий на каждом узле кластера сервис, который абстрагирует многие команды PowerShell, реализующие функциональные возможности узла CSI. Это связано с тем, что концепция *привилегированных* контейнеров в Windows является совершенно новой и поддерживается только последними версиями containerd.

Мы полагаем, что со временем многие, использующие Kubernetes в Windows, тоже смогут запускать свои реализации CSI как наборы демонов DaemonSet с поведением, аналогичным поведению DaemonSet в Linux, продемонстрированным в этой главе. Потребность в абстрагировании хранилища возникает на многих уровнях вычислительного стека, и Kubernetes – это всего лишь еще одна абстракция поверх постоянно растущей экосистемы хранения данных для приложений.

Итоги

- Модули Pod могут получать хранилища прямо во время выполнения, используя операции монтирования, которые выполняет kubelet.

- Самый простой способ поэкспериментировать с провайдерами хранилищ в Kubernetes – создать PVC в модуле Pod в кластере kind.
- Провайдер CSI для NFS – один из множества. Все провайдеры соответствуют одному и тому же стандарту CSI в отношении монтирования хранилищ для контейнеров, который позволяет отделить исходный код Kubernetes от исходного кода поставщика решений хранения данных.
- Контроллер CSI и сервисы узлов включают несколько абстрактных функций, которые позволяют провайдерам динамически предоставлять хранилище модулям Pod через CSI API.
- Интерфейс CSI может также работать в ОС, отличных от Linux. Ярким примером реализации этого типа может служить CSI-прокси для Windows.
- Виртуальная файловая система (VFS) Linux поддерживает все, что можно открыть, читать и записывать. Операции с дисками происходят под управлением ее API.

Реализация и моделирование хранилищ

В этой главе:

- знакомство с работой динамических хранилищ;
- использование томов `emptyDir` в рабочих нагрузках;
- управление хранилищем с помощью провайдеров CSI;
- использование значений `hostPath` с CSI и CSI;
- реализация шаблонов `storageClassTemplate` для Cassandra.

Моделирование хранилища в кластере Kubernetes – одна из самых важных задач, лежащих на плечах администратора, которые он должен решить перед передачей кластера в эксплуатацию. Для этого он должен определить, какой объем хранилища требуется приложению, и этот вопрос имеет свои нюансы. Как правило, готовясь запустить в кластере приложение, которому требуется хранилище, вы должны задать себе следующие вопросы:

- должно ли хранилище гарантировать сохранность данных? Для организации хранилища с гарантиями сохранности часто приходится использовать такие решения, как NAS, NFS или что-то вроде GlusterFS. Все они имеют свои достоинства и недостатки, которые вы должны оценить и взвесить;
- должно ли хранилище работать быстро? Является ли скорость ввода/вывода критически важным параметром? Если важна высокая скорость, то часто хорошим выбором оказывается храни-

лище `emptyDir`, работающее в памяти, или хранилище специального типа с подходящим контроллером;

- какой объем хранилища потребуется каждому контейнеру и сколько контейнеров планируется запустить? При большом количестве контейнеров может потребоваться контроллер хранилища;
- нужен ли выделенный диск для обеспечения безопасности? Если да, то вашим потребностям могут лучше соответствовать локальные тома;
- используются ли рабочие нагрузки искусственного интеллекта (ИИ) с кешами для моделей и обучающих данных? Для них могут потребоваться быстродействующие тома, остающиеся в памяти в течение нескольких часов;
- потребности в хранилищах укладываются в диапазон от 1 до 10 Гбайт? Если да, то в большинстве случаев можно с успехом использовать локальное хранилище или `emptyDir`;
- используется ли что-то вроде распределенной файловой системы Hadoop (Hadoop Distributed File System, HDFS) или Cassandra, выполняющей репликацию и резервное копирование данных? Если да, то в таком случае вы сможете использовать только тома на локальном диске, но это усложняет восстановление;
- допускается ли приостановка хранилища? Если да, то, возможно, вам подойдет модель хранения объектов поверх распределенных томов, например, с применением таких технологий, как NFS или GlusterFS.

8.1 Микрокосм в экосистеме Kubernetes: динамическое хранилище

Определив, что приложению необходимо хранилище, можно взглянуть, какие примитивы, предоставляет Kubernetes. Существует довольно много хранилищ, преследующих разные цели. Это связано с тем, что хранилище, в отличие от сети, является чрезвычайно ограниченным и дорогостоящим ресурсом из-за физических ограничений (одно требование к сохранности данных между перезагрузками чего стоит) и различных юридических и процедурных аспектов хранения данных на предприятиях. Чтобы удержать все это в голове, давайте кратко рассмотрим общую схему технологий хранения, изображенную на рис. 8.1.

На рис. 8.1 можно видеть, что пользователи *запрашивают* хранилище, администраторы *определяют* хранилище с помощью классов хранилищ, а инструменты подготовки CSI обычно отвечают за *подготовку* хранилища, доступного пользователю для записи. Если вернуться к главе о сети, то это представление хранилищ можно сравнить с недавно появившимся Gateway API для балансировки нагрузки уровня 7:

- классы GatewayClass в некотором роде являются аналогами StorageClass в CSI, потому что они определяют тип точки входа в сеть;
- шлюзы подобны PersistentVolume (PV) в CSI в том смысле, что они представляют подготовленные балансирующие нагрузки уровня 7;
- маршруты аналогичны PersistentVolumeClaim (PVC) в CSI в том смысле, что позволяют отдельным разработчикам запрашивать экземпляры GatewayClass для конкретного приложения.

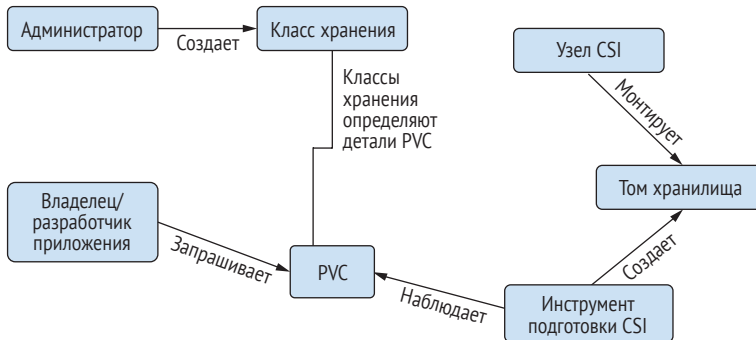


Рис. 8.1 Обобщенное представление хранилища

Таким образом, углубляясь в изучение хранилищ, полезно помнить, что значительная часть самого Kubernetes с течением времени все больше подстраивается под идею создания API, независимых от поставщиков, для доступа к ресурсам инфраструктуры, которыми разработчики и администраторы могут управлять асинхронно и независимо. А теперь, памятуя о сказанном, перейдем к знакомству с идеей динамического хранилища и того, что она означает для разработчиков на практике.

8.1.1 Оперативное управление хранилищем: динамическое выделение ресурсов

Возможность оперативного управления хранилищем в кластере означает также необходимость поддержки оперативного выделения томов. Эта поддержка называется *динамической подготовкой* (dynamic provisioning), или *динамическим выделением ресурсов*. В самом общем смысле динамическая подготовка поддерживается многими кластерными решениями (например, Mesos уже некоторое время предлагает PersistentVolume, резервирующий постоянное, восстанавливаемое локальное хранилище для процессов). Любой, использовавший такой продукт, как VSan, знает, что облачный провайдер EBS должен поддерживать некоторую модель хранения на основе API.

Динамическая подготовка в Kubernetes отличается широкими возможностями подключения сменных модулей (PVC, CSI) и декларатив-

ным характером (PVC с динамической подготовкой). Это позволяет определять свою семантику для различных типов решений хранения данных и обеспечивает возможность косвенных взаимодействий между PVC и соответствующим PersistentVolume.

8.1.2 Локальное хранилище в сравнении с *emptyDir*

Том *emptyDir* хорошо известен большинству новичков в Kubernetes. Он предлагает самый простой способ смонтировать каталог в Pod и практически не требует затрат на безопасность или ресурсы, за которыми необходимо внимательно следить. И все же его использование сопряжено с множеством тонкостей, которые могут способствовать повышению безопасности и производительности при переносе приложения в промышленное окружение.

В табл. 8.1 приводятся сравнительные характеристики локальных томов и томов *emptyDir*. Эти тома имеют совершенно разный жизненный цикл, несмотря на то что все данные хранятся локально. Например, в случае аварии локальный том можно использовать для восстановления данных из работающей базы данных, тогда как *emptyDir* не поддерживает этого. Использование сторонних томов и контроллеров для PVC – это третий вариант применения, который обычно подразумевает, что хранилище может быть переносимым и монтироваться на новых узлах в случае миграции модуля Pod.

Таблица 8.1 Сравнение хранилищ: локального, *emptyDir* и PVC

Тип хранилища	Продолжительность жизни	Постоянное	Реализация	Типичные потребители
Локальное	Располагается на локальном диске	Да	Локальный диск узла	Приложения, обрабатывающие большие объемы данных, или устаревшие приложения
<i>emptyDir</i>	Пока Pod находится на узле	Нет	Локальная папка на узле	Любые модули Pod
PVC	Бесконечная	Да	Сторонние поставщики услуг хранения	Легковесные приложения баз данных

Как правило, мы PVC применяются для приложений, для которых сохранность данных важнее всего. В случае комплексных требований можно использовать том локального хранилища (например, когда приложение должно работать в одном и том же месте и подключаться к огромному диску). Тома *emptyDir* не имеют конкретного варианта использования и служат своего рода швейцарским армейским ножом. Обычно они используются, когда двум контейнерам требуется небольшое пространство для временного хранения данных. Возможно, вам интересно, зачем использовать тип *emptyDir* вместо простого монтирования реального PersistentVolume непосредственно в два контейнера. Тому есть несколько причин:

- *тома PersistentVolume обычно обходятся довольно дорого.* Им требуется контроллер распределенного хранилища, который предоставит том определенного объема, и этот объем может быть ограничен. Если нет нужды хранить данные для модуля Pod, то и нет смысла тратить ресурсы;
- *производительность томов PersistentVolume может быть на порядок ниже производительности томов emptyDir.* Это связано с тем, что часто сохраняемые данные должны передаваться по сети и сохраняться на диске. Тома emptyDir, напротив, могут использовать временное файловое хранилище (tmpfs) и находиться в оперативной памяти, которая по определению работает намного быстрее;
- *тома PersistentVolume по своей природе менее безопасны, чем emptyDir.* Данные, хранящиеся в томе PersistentVolume, могут сохраняться и читаться в разных местах кластера. Тома emptyDir, напротив, недоступны за пределами модулей Pod, объявивших их;
- *emptyDir можно использовать с пустыми контейнерами для создания каталогов, включая /var/log и /etc/, когда приложению требуется выполнять запись данных в файлы журналов или в конфигурационные файлы;*
- *для большей производительности или поддержки определенной функциональности желательно добавить в контейнер каталог /tmp или /var/log.*

Том emptyDir можно использовать для оптимизации производительности или для манипуляций со структурами каталогов контейнеров. В функциональном смысле контейнеру может потребоваться том emptyDir, если в нем самом отсутствует каталог по умолчанию, содержащий только один выполняемый файл. Иногда контейнер создается с временным образом, чтобы уменьшить влияние на его безопасность, но в этом случае исключается возможность кеширования или хранения простых файлов.

Даже если в образе контейнера есть каталог /var/log, вы все равно сможете использовать emptyDir для оптимизации производительности записи данных на диск. Это распространенный прием, потому что контейнеры, записывающие файлы в предопределенный каталог (например, /var/log), могут терять производительность из-за медлительности, свойственной операциям с файловой системой, выполняющим копирование при записи. Слои контейнеров обычно имеют такие файловые системы, которые позволяют процессу записывать данные на верхний уровень файловой системы, фактически не затрагивая нижележащий образ контейнера. Это позволяет делать в работающем контейнере практически все, что угодно, не опасаясь повредить базовый образ Docker, но это отрицательно сказывается на производительности. Файловые системы с копированием при записи часто работают медленно (и потенциально сильно нагружают про-

цессор). Впрочем, многое зависит от драйвера хранилища, который используется в среде выполнения контейнеров.

Как видите, тома `emptyDir` имеют довольно сложную организацию с точки зрения их использования в промышленном окружении. Но в целом если вам нужно хранилище в Kubernetes, то, скорее всего, вы потратите гораздо больше времени на решение проблем, связанных с `PersistentVolume`, чем на изучение особенностей эфемерных хранилищ.

8.1.3 Тома *PersistentVolume*

PersistentVolume – это ссылка Kubernetes на том хранилища, который можно подключить к модулю Pod. Хранилище монтируется агентом `kubelet`, который создает и монтирует различные типы томов и/или вызывает драйвер CSI (о нем мы поговорим далее). Соответственно, запрос `PersistentVolumeClaim` (PVC) является именованной ссылкой на `PersistentVolume`. Этот запрос связывает том, если он имеет тип `RWO` (что означает `read-write-once` – однократное чтение–запись), так что другие модули Pod не могут использовать его, пока он не будет смонтирован снова. Когда создается Pod, которому требуется постоянное хранилище, обычно возникает следующая цепочка событий:

- 1 запрашивается создание Pod, которому требуется PVC;
- 2 планировщик Kubernetes начинает искать место для него – узел с соответствующей топологией хранилища, количеством процессоров и объемом памяти;
- 3 создается действующий PVC, к которому Pod может получить доступ;
- 4 запрошенный объем выделяется плоскостью управления Kubernetes.

Контроллер динамического хранилища создает `PersistentVolume`. В большинстве промышленных кластеров Kubernetes имеется как минимум один такой контроллер, а чаще несколько (различающихся именем `StorageClass`). Контроллеры просто наблюдают за созданием стандартных объектов PVC на сервере API, а затем создают тома, соответствующие им;

- 5 после проверки требований к хранилищу планировщик решает, что Pod готов;
- 6 Pod, зависящий от этого запроса PVC, планируется и запускается;
- 7 во время запуска Pod агент `kubelet` монтирует локальные каталоги, соответствующие PVC;
- 8 локально смонтированный том становится доступным для записи в Pod;
- 9 модуль Pod начинает работу и выполняет чтение или запись в хранилище, существующее внутри `PersistentVolume`.

Планировщик Kubernetes и подключение томов к модулям Pod

Планировщик Kubernetes неразрывно связан с логикой подключения томов к модулям Pod. Он определяет несколько расширений, позволяя реализовать дополнительную логику для удовлетворения различных требований модулей Pod, например, к хранилищам. К числу таких расширений относятся: PreFilter, Filter, PostFilter, Reserve, PreScore, PreBind, Bind, PostBind, Permit и QueueSort. Точка расширения PreFilter – это одно из мест, где планировщик реализует логику хранения.

Способность принимать решение о готовности модуля Pod к запуску отчасти определяется параметрами хранилища, известными планировщику. Например, планировщик избегает планирования модуля Pod, зависящего от тома, который может использоваться только одним читателем и уже подключен к другому модулю Pod. Это предотвращает появление ошибок на запуске модуля Pod, когда привязка тома не происходит по непонятной для вас причине.

Возможно, вам интересно, зачем планировщику информация о хранилище. (В конце концов, как многие представляют, за подключение хранилища отвечает kubelet.) Причина – гарантии производительности и предсказуемости. Например, вам может понадобиться ограничить количество томов на узле. Кроме того, если какие-то узлы имеют ограничения, касающиеся хранилища, планировщик может отказаться от размещения модулей Pod на этих узлах, чтобы не создавать «зомби-Pod», которые, хоть и запланированы, но не могут запуститься должным образом из-за отсутствия доступа к ресурсам хранилища.

Благодаря недавним нововведениям в Kubernetes API для поддержки логики определения емкости хранилищ в CSI API появилась возможность описывать ограничения хранилищ, которые планировщик может запросить и использовать, выбирая узлы, лучше всего соответствующие требованиям модулей Pod к хранилищам. Дополнительную информацию можно найти по адресу <http://mng.bz/M2pE>.

8.1.4 Интерфейс контейнерного хранилища (CSI)

Вам, наверное, интересно узнать, как агенту kubelet удается монтировать хранилища произвольных типов. Например, такая файловая система, как NFS, требует предварительной установки клиента NFS. Действительно, монтирование хранилищ во многом зависит от платформы, и kubelet не решает эту проблему как по волшебству.

До версии Kubernetes 1.12 поддержка распространенных файловых систем, таких как NFS, GlusterFS, Ceph и многих другие, включалась непосредственно в kubelet. Однако появление интерфейса CSI изменило ситуацию, и теперь kubelet практически ничего не знает о файловых системах для конкретных платформ. Вместо этого пользователи, которым требуется монтировать хранилища определенного типа, должны запускать в своих кластерах соответствующие драйверы CSI

в DaemonSet. Эти драйверы используют сокет для связи с kubelet и выполняют низкоуровневые операции монтирования файловой системы. Переход на CSI позволяет поставщикам развивать клиентов своих хранилищ и публиковать обновления, не добавляя свою специфическую логику в конкретные версии Kubernetes.

ПРИМЕЧАНИЕ В фонде CNCF (Cloud Native Computing Foundation) типичной практикой считается сначала опубликовать проект с открытым исходным кодом, включающий множество зависимостей, а затем постепенно удалять эти зависимости. Это помогает привлечь первых пользователей простотой. Однако, как только внедрение новых механизмов становится обычным явлением, выполняется работа по отделению таких зависимостей. Примерами такого подхода могут служить интерфейсы CNI, CSI и CRI.

CSI – это интерфейс контейнерного хранилища (container storage interface), который разработан так, что код поддержки разных типов томов больше не нужно компилировать в ядро Kubernetes. Модель хранилища CSI требует только реализовать некоторые концепции Kubernetes (DaemonSet и контроллер хранилища), чтобы с их помощью kubelet мог предоставить любое хранилище. CSI не зависит от Kubernetes. Справедливости ради следует отметить, что Mesos тоже поддерживает CSI, как и Kubernetes, поэтому мы здесь никого не выделяем.

8.2 Динамическая подготовка выигрывает от CSI, но не зависит от него

Динамическая подготовка (dynamic provisioning), как возможность волшебным образом создать PersistentVolume при появлении PVC, никак не связана с интерфейсом CSI, позволяющим динамически монтировать в контейнер любые хранилища. Однако эти две технологии прекрасно дополняют друг друга. Комбинируя их, разработчик может продолжать использовать одни и те же объявления классов StorageClass (описываются ниже) для монтирования потенциально разных типов хранилищ, но предоставляющих одинаковую высокоуровневую семантику. Например, класс хранилища fast может быть первоначально реализован с использованием твердотельных дисков, доступных через NAS:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
parameters:
  type: pd-ssd
```

Спустя время вы можете приобрести (например, у компании Datera) быстрое хранилище в другом массиве. В любом случае, используя средства динамической подготовки, ваши разработчики смогут продолжать использовать одни и те же запросы API для получения новых томов хранилища, заменяя только драйверы CSI и контроллеры хранилищ, работающие в вашем кластере.

В любом случае динамическая подготовка реализуется большинством облачных провайдеров для Kubernetes с простым типом диска, подключенным к облаку, по умолчанию. Для многих небольших приложений вполне достаточно медленного PersistentVolume, который автоматически выбирается облачным провайдером. Однако для него все рабочие нагрузки одинаковы, и иногда важно иметь возможность выбирать между различными моделями хранения и реализациями политик в отношении PVC.

8.2.1 Классы хранилищ (StorageClass)

Классы *StorageClass* позволяют определять сложную семантику хранения декларативным способом. Несмотря на наличие уникальных параметров, которые можно определять для хранилищ разных типов, общим для всех является *режим привязки*. Именно здесь создание пользовательского динамического средства подготовки может быть чрезвычайно важным.

Динамическая подготовка – это не только способ подготовить простое хранилище к использованию, но и мощный инструмент поддержки высокопроизводительных рабочих нагрузок в центре обработки данных с разнородными требованиями к хранилищам. Любая рабочая нагрузка может выиграть от применения другого класса *StorageClass* с другим режимом привязки, поддержкой восстановления и показателями производительности (о чем мы поговорим ниже).

ГИПОТЕТИЧЕСКИЙ ПРОВАЙДЕР КЛАССА ХРАНИЛИЩ ДЛЯ ЦЕНТРА ОБРАБОТКИ ДАННЫХ

Классы *StorageClass* кажутся довольно абстрактными, но это представление меняется, стоит только рассмотреть вариант использования, когда администратор Kubernetes отбивается от множества разработчиков, жаждущих развернуть свои приложения в промышленном окружении, но мало что знающих о том, как работает хранилище. Давайте представим сценарий с приложениями, относящимися к трем категориям: пакетной обработке данных, транзакционным веб-приложениям и приложениям ИИ. В этом сценарии можно создать единое средство подготовки томов с тремя классами хранилищ. В таком случае приложение сможет декларативно запрашивать определенные типы хранилищ следующим образом:

```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: my-big-data-app-vol
spec:
  storageClassName: bigdata
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100G
```

Этот запрос PVC может использоваться внутри Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-big-data-app
spec:
  volumes:
    - name: myvol
      persistentVolumeClaim:
        claimName: my-big-data-app-vol
  containers:
    - name: my-big-data-app
      image: datacruncher:0.1
      volumeMounts:
        - mountPath: "/mybigdata-app-volume"
          name: myvol
```

КРАТКОЕ НАПОМИНАНИЕ О РАБОТЕ PVC

Kubernetes просматривает метаданные PVC (например, запрашиваемый объем), а затем отыскивает объекты PV, соответствующие запросу. Соответственно, вы не можете явно связать определенное хранилище с запросом PVC. Вы лишь определяете характеристики (например, объем 100 Гбайт) и асинхронно создаете том, соответствующий этим атрибутам.

8.2.2 Вернемся к центрам обработки данных

Что происходит в нашем средстве динамической подготовки, который представлен здесь? Давайте разберемся.

- 1 Мы пишем цикл управления, который отслеживает запросы на получение томов.
- 2 Обнаружив запрос, мы выделяем на жестком диске том размером 100 Гбайт, выполняя вызов API (например, провайдера хранилища на нашем NAS). Обратите внимание, что также можно было бы заранее создать множество каталогов в NAS или NFS.
- 3 Затем определяем объект PV для поддержки PVC. Тип этого тома может быть любым, например NFS или `hostPath`.

С этого момента в работу вступает Kubernetes, и, как только запрос PVC будет выполнен и создан `PersistentVolume`, наши модули Pod смо-

гут планироваться планировщиком. В этом сценарии упоминаются три шага: цикл управления, выделение тома и его создание. Решение о том, какие тома создавать, зависит от типа хранилища, запрашиваемого разработчиками. В предыдущем фрагменте кода мы использовали `bigdata` как тип `StorageClass`. В центрах обработки данных обычно можно поддерживать три класса хранилищ:

- `bigdata` (упоминавшийся выше);
- `postgres`;
- `ai`.

Почему три класса? Нет никаких особых причин для реализации именно трех классов хранилищ. Их может быть и четыре, и пять, и даже больше.

Для рабочих нагрузок в стиле `BigData/HDFS/ETL`, интенсивно использующих хранилище, важна локальность данных. В таких случаях желательно хранить данные на аппаратном диске и читать с него, как если бы это был смонтированный том хоста. Режим привязки для этого типа может выиграть от применения стратегии `WaitForFirstConsumer`, позволяющей создавать и подключать том непосредственно к узлу, где выполняется рабочая нагрузка, а не создавать его заранее в месте с худшей локальностью данных.

Поскольку узлы данных в `Nadoor` гарантируют сохранность данных, а `HDFS` сама поддерживает репликацию, то для этой модели привлекательнее всего выглядит политика хранения `Delete`. Для рабочих нагрузок, поддерживающих работу с «холодными» хранилищами (например, в `GlusterFS`), можно автоматизировать политику внедрения трансляторов для томов хранилищ в рабочие нагрузки, работающие в определенных пространствах имен. В любом случае вся подготовка может выполняться по запросу на самых дешевых дисках, доступных в тот момент времени.

Для рабочих нагрузок в стиле `Postgres/RDBMS` желательно иметь выделенные твердотельные накопители емкостью в несколько терабайт. После создания запроса на хранилище потребуется узнать, где работает ваш `Pod`, чтобы можно было зарезервировать `SSD` в той же стойке или на том же узле. Поскольку местоположение дисков и планирование могут значительно влиять на производительность таких рабочих нагрузок, класс хранилища для `Postgres` может использовать стратегию `WaitForFirstConsumer`. Поскольку база данных `Postgres` часто имеет историю транзакций, для нее можно выбрать политику хранения `Retain`.

Наконец, работая с моделями ИИ, специалисты по данным могут не заботиться о сохранности данных; например, им может быть достаточно обработать некоторый числовой массив, и для этого им необходимо некоторое пространство в хранилище. Вы можете установить связь между разработчиками и типом хранилища, которое им предоставляется, чтобы иметь возможность время от времени изменять `StorageClass` и типы томов в кластере, не затрагивая такие вещи, как

определения YAML API, диаграммы Helm или код приложения. Так же как в сценариях использования «холодных» хранилищ, рабочие нагрузки ИИ загружают большие объемы данных в память в течение короткого периода времени, прежде чем выгрузить их, поэтому локальность данных не всегда важна. Для ускорения запуска модулей Pod можно выполнить немедленную привязку и использовать политику хранения Delete как более уместную.

Учитывая сложность этих процессов, может понадобиться специальная логика для выполнения запросов на тома. Соответствующие типы томов можно назвать просто как `hdfs`, `coldstore`, `pg-perf` и `ai-slow` соответственно.

8.3 *Варианты организации хранилищ в Kubernetes*

Мы только что рассмотрели важность моделирования сценариев использования хранилища конечными пользователями. Теперь исследуем некоторые другие аспекты, чтобы получить более широкое представление об использовании томов хранилищ в Kubernetes для работы с секретами `Secret` и сетевыми функциями.

8.3.1 *Секреты: эфемерная передача файлов*

Шаблон проектирования передачи файлов как способ переноса учетных данных в контейнеры или виртуальные машины получил широкое распространение. Например, фреймворк `cloud-init`, загружающий виртуальные машины в облачные окружения, такие как AWS, Azure и vSphere, имеет директиву `write_files`, обычно используемую за пределами Kubernetes, как показано ниже:

```
# Пример взят из https://cloudinit.readthedocs.io/en/latest/topics
# /examples.html#writing-out-arbitrary-files
write_files:
- encoding: b64
  content: CîMgVGhpcyBmaWxlIGVbnRyb2xzIHROZSBzdGF0ZSBvZiBTRUxpbmV4...
  owner: root:root
  path: /etc/sysconfig/selinux
  permissions: '0644'
- content: |
  # Мой новый файл /etc/sysconfig/samba
  SMBDOPTIONS="-D"
  path: /etc/sysconfig/samba
```

Подобно тому, как системные администраторы используют инструменты, такие как `cloud-init`, для загрузки виртуальных машин, Kubernetes использует сервер API и `kubelet` для загрузки секретов `Sec-`

ret или файлов в модули Pod, применяя почти идентичный шаблон проектирования. Если вам приходилось администрировать облачную среду, обращающуюся к базе данных любого типа, то, скорее, решали эту задачу одним из трех способов:

- *внедрением учетных данных через переменные окружения* – для этого необходимо иметь некоторый контроль над контекстом, в котором выполняется процесс;
- *внедрением учетных данных через файлы* – этот подход позволяет перезапускать процесс, используя другие параметры или аргументы контекста, без обновления переменных окружения с паролями;
- *использованием объекта Secret API* – это конструкция Kubernetes, предназначенная для выполнения практически тех же действий, что и с ConfigMap, с учетом некоторых отличий от ConfigMaps:
 - для шифрования и расшифровывания секретов Secret можно использовать разные типы алгоритмов, что недопустимо для ConfigMap;
 - в отличие от ConfigMap сервер API может шифровать секреты Secret с etcd в состоянии покоя; это упрощает чтение или отладку секретов, но делает их менее безопасными;
 - по умолчанию любые данные в секрете Secret преобразуются в формат Base64. Это связано с распространенным вариантом хранения сертификатов и других сложных типов данных в секретах (а также с очевидным преимуществом, заключающимся в сложности чтения строк в формате Base64).

Предполагается, что со временем поставщики предоставят сложные API ротации секретов, ориентированные на тип Secrets API в Kubernetes. Однако на момент написания этой книги объекты Secret и ConfigMap практически не отличались друг от друга с точки зрения их использования в Kubernetes.

КАК ВЫГЛЯДИТ SECRET?

Вот как выглядит определение объекта Secret в Kubernetes:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  val1: YXNkZgo=
  val2: YXNkZjIK
stringData:
  val1: asdf
```

В этом секрете есть пара значений: val1 и val2. Поле StringData на самом деле хранит val как простую текстовую строку, которую легко прочитать. Многие ошибочно полагают, что секретные данные в Ku-

bernetes защищены кодировкой Base64. Это не так, потому что кодировка Base64 вообще *ничего не защищает!* Безопасность секретов в Kubernetes обеспечивается проверкой и ротацией секретов администратором. В любом случае секреты в Kubernetes хранятся в полной безопасности, потому что они передаются только агенту kubelet для внедрения в модули Pod, имеющие разрешение читать их через RBAC. Значение `val1` может быть позже внедрено в Pod следующим образом:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: my-webapp
    volumeMounts:
    - name: myval
      mountPath: "/etc/myval"
      readOnly: true
  volumes:
  - name: myval
    secret:
      secretName: val1
```

Согласно этим определениям Pod получит файл `/etc/myval`, содержащий значение `asdf`. Этого можно добиться созданием с помощью kubelet специального эфемерного тома `tmpfs`, предназначенного для контейнеров, которым требуется доступ к этому секрету. Агент kubelet также может обновлять этот файл при изменении значения в определении Secret в Kubernetes API, потому что в действительности это самый обычный файл, который находится на хосте и передается модулям Pod с помощью магии пространств имен файловой системы.

Создание простого Pod с пустым томом для быстрой записи

Каноническим примером модуля Pod с томом `emptyDir` может служить приложение, хранящее временные файлы в `/var/tmp`. Эфемерное хранилище обычно монтируется в Pod как:

- том с одним или несколькими файлами, что характерно для ConfigMap, содержащих конфигурационные данные (например, с различными настройками приложения);
- переменные окружения со значениями из секретов Secret.

Для приложения, использующего файл в роли блокировки или семафора для синхронизации нескольких контейнеров или для внедрения в приложение какой-то эфемерной конфигурации (например, через ConfigMap), достаточно локальных томов хранилища, управляемых агентом kubelet. Секреты могут использовать том `emptyDir` для внедрения в контейнер пароля (например, в виде файла). Точно так же

том `emptyDir` может одновременно использоваться двумя модулями `Pod`, благодаря чему можно создать простую очередь заданий или сигналов между двумя контейнерами.

`emptyDir` – это самый простой в реализации тип хранилищ. Ему не требуется физический том на диске, и он гарантированно будет работать в любом кластере. Например, в базе данных `Redis`, от которой не требуется долговременное хранение данных, можно смонтировать эфемерное хранилище в виде тома, как показано ниже:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: redis-storage
          mountPath: /data/redis
  volumes:
    - name: redis-storage
      emptyDir: {}
```

Почему мы так много говорим о `emptyDir`? Потому что, как упоминалось выше, `emptyDir` имеет более высокую производительность, чем контейнеризованный каталог. Не забывайте, что среда выполнения контейнеров выполняет запись в файл, используя файловую систему с копированием при записи, отличающуюся от операций записи в обычные файлы на диске. Поэтому в папки, при работе с которыми важна высокая производительность, можно намеренно монтировать тома `emptyDir` или `hostPath`. В некоторых средах выполнения контейнеров такой шаг нередко дает ускорение до десяти раз по сравнению с записью в файловую систему хоста.

8.4 Как выглядит типичный провайдер динамического хранилища?

В отличие от томов `emptyDir` реализации провайдеров хранилищ находятся за пределами `Kubernetes` и обычно предоставляются поставщиками решений. Для их внедрения требуется подготовить спецификацию `CSI`. Например, мы могли бы создать провайдера хранилища `NAS`, который циклически перебирает список предопределенных папок. В следующем примере определяется только шесть томов, чтобы сделать код более простым и конкретным. Однако в реальном мире может потребоваться более сложный способ управления базовыми каталогами хранилища. Например:

```

var storageFolder1 = "/opt/NAS/1"
var storageFolder2 = "/opt/NAS/2"
var storageFolder3 = "/opt/NAS/3"
var storageFolder4 = "/opt/NAS/4"
var storageFolder5 = "/opt/NAS/5"
var storageFolder6 = "/opt/NAS/6"
var storageFoldersUsed = 0

// Функция Provision создает ресурс хранилища
// и возвращает объект PV, представляющий его.
func (p *hostPathProvisioner) Provision
(options controller.VolumeOptions) (*v1.PersistentVolume, error) {
    glog.Infof("Provisioning volume %v", options)
    path := path.Join(p.pvDir, options.PVName)

    // Реализация нашего искусственного ограничения простейшим способом...
    if storageFoldersUsed == 0 {
        panic("Cant store anything else !")
    }
    if err := os.MkdirAll(path, 0777); err != nil {
        return nil, err
    }

    // Явный вызов chmod создаст каталог, для которого точно известно,
    // что его набор разрешений будет иметь вид 0777, независимо от значения umask
    if err := os.Chmod(path, 0777); err != nil {
        return nil, err
    }

    // Пример вызова NAS
    folders := []string{
        storageFolder1, storageFolder2, storageFolder3,
        storageFolder4, storageFolder5, storageFolder6
    }

    // Теперь собственно создание папки ...
    mycompany.ProvisionNewNasResourceToLocalFolder
        (folders[storageFoldersUsed++]);

    // Этот код практически целиком взят из контроллера minikubes ...
    pv := &v1.PersistentVolume{
        ObjectMeta: metav1.ObjectMeta{
            Name: options.PVName,
            Annotations: map[string]string{
                // Замените это значение своим
                "myCompanyStoragePathIdentity": string(p.identity),
            },
        },
        Spec: v1.PersistentVolumeSpec{
            PersistentVolumeReclaimPolicy:
                options.PersistentVolumeReclaimPolicy,
            AccessModes: options.PVC.Spec.AccessModes,
            Capacity: v1.ResourceList{

```

Шесть разных точек монтирования

Поместить точки монтирования в массив для циклического перебора

Создает PV YAML, подобно тому как мы делали это вручную

```

v1.ResourceName(v1.ResourceStorage):
    options.PVC.Spec.Resources.Requests[
        v1.ResourceName(v1.ResourceStorage)...
    ],
PersistentVolumeSource: v1.PersistentVolumeSource{
    HostPath: &v1.HostPathVolumeSource{
        Path: storageFolder,
    },
},
},
}
return pv, nil
}

```

Использовать hostPath, чтобы смонтировать в каталог в нашем NAS

Имейте в виду, что этот код – лишь гипотетический пример, как можно написать свой механизм подготовки, заимствуя логику подготовки hostPath в minikube. Остальной код контроллера хранилища в minikube можно найти по адресу <http://mng.bz/wn5P>. Если вы хотите глубже понять, как работают PersistentVolumeClaim или StorageClass, то обязательно прочитайте этот код или, что еще лучше, попробуйте скомпилировать его у себя!

8.5 hostPath для управления системой и/или доступа к данным

Томы hostPath в Kubernetes похожи на тома Docker, позволяя контейнерам записывать данные непосредственно в файловую систему хоста. Это мощная функция, которой часто злоупотребляют новички, осваивающие разработку микросервисов, поэтому будьте осторожны, используя ее. Тип томов hostPath имеет широкий спектр применений. Обычно их делят на две категории:

- для реализации вспомогательных функций, предоставляемых контейнерами, которые можно реализовать, только имея доступ к файловой системе хоста (мы рассмотрим это на примере);
- для долговременного хранения файлов, чтобы, когда Pod исчезнет, его данные сохранялись в предсказуемом месте. Обратите внимание, что это применение – почти всегда антишаблон, потому что может приводить к изменению поведения приложения, когда Pod останавливается и затем переназначается на новый узел.

8.5.1 hostPath, CSI и CNI: канонический вариант использования

Интерфейсы CNI и CSI, составляющие основу для сетевых взаимодействий и организации хранилищ в Kubernetes, в значительной мере за-

висят от использования hostPath. Сам kubelet запускается на каждом узле и монтирует и размонтирует тома хранилищ, используя драйвер CSI и сокет домена UNIX, который, как вы уже наверняка догадались, основан на применении тома hostPath. Существует также второй сокет домена UNIX, который использует node-driver-registrar для регистрации драйвера CSI в kubelet.

Как уже упоминалось, многие варианты применения hostPath в приложениях считаются антишаблонами. Однако одним из распространенных и важных применений hostPath является реализация плагина CNI. Давайте рассмотрим этот вариант дальше.

ПРИМЕР CNI HOSTPATH

В качестве примера, насколько сильно провайдеры CNI могут зависеть от поддержки hostPath, давайте посмотрим, как монтируются тома на работающем узле Calico. Calico Pod отвечает за многие действия на системном уровне, такие как управление правилами XDP, правилами iptables и т. д. Кроме того, эти модули Pod должны убедиться, что таблицы BGP в ядре Linux правильно синхронизированы. То есть, как можно видеть, существует множество объявлений томов hostPath для доступа к различным каталогам хоста. Например:

```
volumes:
  - hostPath:
      path: /run/xtables.lock
      type: FileOrCreate
      name: xtables-lock
  - hostPath:
      path: /opt/cni/bin
      type: ""
...
```

Провайдеры CNI в Linux устанавливаются на узел с kubelet, буквально записывая свои двоичные файлы из контейнера в файловую систему узла, обычно в каталог /opt/cni/bin. Это один из самых популярных вариантов применения hostPath – использование контейнеров для выполнения административных действий на узле Linux. Эту возможность используют многие приложения, имеющие административный характер, в том числе:

- Prometheus – решение для сбора метрик и мониторинга, монтирующее /proc и другие системные ресурсы с целью получения параметров потребления ресурсов;
- Logstash – решение для интеграции журналов, подключающее различные каталоги с журналами к контейнерам;
- провайдеры CNI, которые, как уже упоминалось, самостоятельно устанавливают двоичные файлы в /opt/cni/bin;
- провайдеры CSI, использующие hostPath для монтирования своих хранилищ.

Провайдер Calico CNI – один из многих таких низкоуровневых системных процессов Kubernetes, которые не могли бы выполняться без возможности монтировать устройства или каталоги хоста в контейнер напрямую. Фактически другим CNI (таким как Antrea или Flannel) и даже драйверам хранилищ CSI тоже требуется эта функциональность для запуска и управления хостами.

Поначалу такой способ самостоятельной установки может показаться нелогичным, поэтому вам понадобится немного времени, чтобы разобраться с этим. Тимоти Сент-Клер (Timothy St. Claire), один из первых разработчиков Kubernetes, назвал такое поведение «проникновением внутрь через собственный пупок». Но, как бы то ни было, этот способ лежит в основе работы Kubernetes в Linux. (Мы говорим «в Linux», потому что в других ОС, таких как Windows, такой уровень контейнерных привилегий пока недоступен. С появлением контейнеров Windows HostProcess в Kubernetes 1.22 можно заметить, как эта парадигма укореняется и в окружениях, отличных от Linux.) Таким образом, тома `hostPath` – это не просто функция поддержки контейнерных рабочих нагрузок, она фактически позволяет контейнерам администрировать сложные аспекты сервера Linux за пределами области контейнерных приложений.

Когда следует использовать тома `hostPath`?

В своих путешествиях по хранилищам помните, что `hostPath` можно использовать для самых разных целей, и, хотя это считается антишаблоном, применение томов этого типа может довольно легко вытащить вас из многих затруднительных ситуаций. `hostPath` позволяет, к примеру, реализовать простое и быстрое резервное копирование, осуществление политик соответствия (когда узлы авторизованы для хранения, а распределенные тома – нет) и предоставление высокопроизводительных хранилищ без опоры на глубокую облачную интеграцию. В общем случае, рассматривая пути реализации хранилищ для каждого конкретного случая, учитывайте следующее:

- существует ли встроенный провайдер томов Kubernetes? Если да, то его применение может оказаться самым простым решением, требующим наименьших усилий по автоматизации с вашей стороны;
- если нет, то предоставляет ли ваш поставщик реализацию CSI? Если предоставляет, то можно использовать его, и, скорее всего, он будет иметь в своем составе средство динамической подготовки томов.

Если ни один из этих вариантов не подходит, попробуйте использовать такие инструменты, как тома `hostPath` или `Flex`, чтобы настроить хранилище как том, который можно привязать к любому модулю `Pod`. Возможно, при этом придется добавить в `Pod` информацию о планировании, если только определенные хосты в кластере имеют доступ

к этому провайдеру хранилища, поэтому первый вариант часто оказывается идеальным.

8.5.2 *Cassandra: пример реального хранилища в Kubernetes*

Приложения в Kubernetes, использующие хранилища, должны поддерживать динамическое масштабирование, однако все еще существуют предсказуемые способы доступа к именованным томам с критически важными данными. Давайте рассмотрим усложненный вариант использования хранилища – Cassandra.

Модули Pod с Cassandra обычно управляются посредством StatefulSet. Идея StatefulSet заключается в том, что Pod постоянно воссоздается на одном и том же узле. В этом случае вместо простого определения тома имеется шаблон VolumeClaimTemplate. Эти шаблоны имеют разные имена для разных томов.

volumeClaimTemplates – это конструкция в Kubernetes API, сообщающая Kubernetes, как объявлять PersistentVolume для StatefulSet, чтобы их можно было создавать динамически, в зависимости от размера StatefulSet, оператором, который устанавливает этот StatefulSet в первый раз, или тем, кто его масштабирует. Например:

```
volumeClaimTemplates:
- metadata:
  name: cassandra-data
```

Модуль Pod с именем cassandra-1, например, будет иметь шаблон volumeClaimTemplate с именем cassandra-data-1. Он находится на том же узле, и StatefulSet снова и снова будет назначаться на один и тот же узел.

Не путайте объекты StatefulSet с DaemonSet. Последний гарантирует, что один и тот же модуль Pod будет выполняться на всех узлах кластера, а первый – что модули будут повторно запускаться на том же узле, но ничего не говорит о том, сколько таких модулей Pod будет запущено и где. Чтобы сделать это отличие более явным, отметим, что объекты DaemonSet используются для поддержки безопасности, контейнерных провайдеров сети и хранилищ и т. д. А теперь посмотрим, как выглядит StatefulSet для Cassandra вместе с его volumeClaimTemplate:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  serviceName: cassandra
  replicas: 1
  selector:
```

```

matchLabels:
...
volumeMounts:
- name: cassandra-data
  mountPath: /cassandra_data
# Контроллер преобразует их в запросы томов
# и монтирует в каталоги, упомянутые в нашем обсуждении, и не
# использует в промышленном окружении, пока не будет установлен
# ssd GCEPersistentDisk или другой ssd pd
volumeClaimTemplates:
- metadata:
  name: cassandra-data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: fast
    resources:
      requests:
        storage: 1Gi
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
parameters:
  type: pd-ssd

```

С этого момента при каждом повторном запуске на этом же узле модуль Pod с Cassandra будет обращаться к тому же тому с заранее известным именем. Это позволяет добавить в кластер дополнительные реплики Cassandra и гарантировать, например, что восьмой Pod всегда будет запускаться на восьмом узле в кворуме Cassandra. Без такого шаблона вам пришлось бы вручную определять уникальные имена volumeClaimTemplate при каждом увеличении количества модулей Pod в кворуме Cassandra. Обратите внимание, что если потребуется перепланировать модуль Pod на другой узел и хранилище можно подключить к этому узлу, то это хранилище будет перемещено, и Pod запустится на этом узле.

8.5.3 *Дополнительные возможности и модель хранения в Kubernetes*

К сожалению, всю функциональность конкретного типа хранилища невозможно полностью выразить в Kubernetes. Например, разные типы томов могут иметь разную семантику чтения и записи, когда речь заходит о низкоуровневых операциях с хранилищем. Еще одним примером может служить концепция *моментальных снимков*. Многие поставщики облачных услуг позволяют создавать резервные копии, восстанавливать их или делать моментальные снимки дисков. Если поставщик решения предусмотрел поддержку моментальных сним-

ков и надлежащим образом реализовал их семантику в своей спецификации CSI, то вы сможете использовать эту функцию.

Начиная с Kubernetes 1.17, моментальные снимки и клонирование (которые могут быть полностью реализованы в Kubernetes) включены в Kubernetes API как новые операции. Например, следующий PVC определяется как производная от источника данных. Сам источник данных в свою очередь является объектом VolumeSnapshot, т. е. это определенный том, загружаемый с определенного момента времени:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restore-pvc
spec:
  storageClassName: csi-hostpath-sc
dataSource:
  name: new-snapshot-test
  kind: VolumeSnapshot
  apiGroup: snapshot.storage.k8s.io
accessModes:
  - ReadWriteOnce
resources:
  requests:
    storage: 10Gi
```

Мы уже познакомились с важностью спецификации CSI, поэтому вы, скорее всего, догадались, что подключение клиента Kubernetes к логике моментальных снимков конкретного решения совершенно не нужно: для поддержки этой возможности поставщику нужно просто реализовать несколько вызовов CSI API, например:

- CreateSnapshot;
- DeleteSnapshot;
- ListSnapshots.

После их реализации CSI-контроллеры в Kubernetes смогут управлять моментальными снимками. Если вас заинтересовала тема создания моментальных снимков для томов в промышленном окружении, обратитесь к конкретным драйверам CSI или провайдерам хранилищ, которые вы используете в своих кластерах Kubernetes, и проверьте – реализуют ли они компоненты поддержки моментальных снимков CSI API.

8.6 *Дополнительная литература*

J. Eder. «The Path to Cloud-Native Trading Platforms». <http://mng.bz/p2nE> (доступно по состоянию на 24.12.2021).

От авторов Kubernetes. «PV controller changes to support PV Deletion protection finalizer». <http://mng.bz/g46Z> (доступно по состоянию на 24.12.2021).

От авторов Kubernetes. «Remove docker as container runtime for local-up». <http://mng.bz/enaw> (доступно по состоянию на 24.12.2021).
Документация Kubernetes. «Create static Pods». <http://mng.bz/g4eZ> (доступно по состоянию на 24.12.2021).
Документация Kubernetes. «Persistent Volumes». <http://mng.bz/en9w> (доступно по состоянию на 24.12.2021).
«PostgreSQL DB Restore: unexpected data beyond EOF». <http://mng.bz/aDQx> (доступно по состоянию на 24.12.2021).
«Shared Storage». https://wiki.postgresql.org/wiki/Shared_Storage (доступно по состоянию на 24.12.2021).
Z. Zhuang and C. Tran. «Eliminating Large JVM GC Pauses Caused by Background IO Traffic». <http://mng.bz/5KJ4> (доступно по состоянию на 24.12.2021).

Итоги

- Классы хранилищ StorageClass подобны другим многопользовательским концепциям в Kubernetes, таким как GatewayClass.
- Администраторы моделируют требования к хранилищу, используя классы StorageClass, и приспособливают стандартные сценарии обобщенным образом.
- Фреймворк Kubernetes сам использует тома emptyDir и hostPath для выполнения повседневных задач.
- Для получения предсказуемых имен томов при перезапуске модулей Pod можно использовать шаблоны volumeClaimTemplate, определяющие именованные тома для модулей Pod в StatefulSet. Это позволяет гарантировать высокую производительность рабочих нагрузок с состоянием, например, кластера Cassandra.
- Создание моментальных снимков томов и клонирование становятся популярными вариантами использования хранилищ, которые можно осуществить с помощью новых реализаций CSI.

Запуск модулей Pod: как работает kubelet

В этой главе:

- что делает kubelet и как он настраивается;
- инициализация среды выполнения и запуск контейнеров;
- управление жизненным циклом модулей Pod;
- знакомство с CRI;
- обзор интерфейсов Go внутри kubelet и CRI.

kubelet – это рабочая лошадка кластера Kubernetes; в центрах обработки данных могут работать тысячи агентов kubelet, потому что он запускается на каждом узле. В этой главе мы посмотрим, что делает kubelet и как он использует интерфейс выполнения контейнеров (Container Runtime Interface, CRI) для запуска контейнеров и управления жизненным циклом рабочих нагрузок.

Одной из задач kubelet является запуск и остановка контейнеров, а CRI – это интерфейс, посредством которого kubelet взаимодействует со средой выполнения контейнеров. Например, *containerd* классифицируется как среда выполнения контейнеров, которая получает образ и создает действующий контейнер. Движок *Docker* – это среда выполнения контейнеров, но в настоящее время сообщество Kubernetes отказывается от него в пользу *containerd*, *runC* и других окружений.

ПРИМЕЧАНИЕ Мы хотим поблагодарить Дону Чен (Dawn Chen), любезно согласившуюся ответить на ряд вопросов о работе kubelet. Дона – первый автор kubelet и в настоящее время руководит группой «Node Special Interest Group for Kubernetes», которая осуществляет поддержку кодовой базы kubelet.

9.1 *kubelet и узел*

kubelet – это двоичная программа, запускаемая демоном systemd на каждом узле. Она играет роль планировщика модулей Pod и агента локального узла. kubelet хранит и поддерживает информацию о сервере, на котором выполняется, для своего узла и при обнаружении изменений обновляет объект Node с помощью сервера API.

Начнем наше путешествие с обзора объекта Node, который можно получить командой `kubectl get nodes <укажите_здесь_имя_узла> -o yaml` в действующем кластере. Ниже приводится пример кода, созданного командой `kubectl get nodes`. Вы можете последовать за нашими примерами, предварительно выполнив команды `kind create cluster` и `kubectl`. Например, команда `kubectl get nodes -o yaml` вернет следующий результат (здесь приводится несколько сокращенный вариант):

```
kind: Node
metadata:
  annotations:
    kubeadm.alpha.kubernetes.io/cri-socket:
      /run/containerd/containerd.sock ← Kubelet использует этот
                                         сокет для связи со средой
                                         выполнения контейнеров
    node.alpha.kubernetes.io/ttl: "0"
    volumes.kubernetes.io/controller-managed-attach-detach: "true"
  labels:
    beta.kubernetes.io/arch: amd64
    kubernetes.io/hostname: kind-control-plane
    node-role.kubernetes.io/master: ""
name: kind-control-plane
```

Метаданные в этом объекте Node помогают понять, что представляет собой его среда выполнения контейнеров и какую архитектуру Linux она использует. Для получения этой информации kubelet обращается к провайдеру CNI. Как упоминалось выше, задача провайдера CNI заключается в выделении IP-адресов и создании сети модулей Pod, обеспечивающей возможность сетевых взаимодействий внутри кластера Kubernetes. Объект Node API определяет CIDR (диапазон IP-адресов) для всех модулей Pod. Важно отметить, что также указывается внутренний IP-адрес для самого узла, обязательно отличающийся от диапазона адресов CIDR модулей Pod. Следующий фрагмент представляет часть кода YAML, созданного командой `kubectl get node:`

```
spec:
  podCIDR: 10.244.0.0/24
```

Теперь мы подошли к разделу `status` в определении. Все объекты Kubernetes API имеют поля `spec` и `status`:

- `spec` – определяет характеристики объекта (каким он должен быть);
- `status` – представляет текущее состояние объекта.

Раздел `status` – это данные, которые kubelet поддерживает для кластера. Помимо всего прочего, он включает список условий, среди которых можно увидеть время последнего обмена контрольными сообщениями (heartbeat) с сервером API. В момент запуска узел автоматически получает всю необходимую системную информацию. Информация о состоянии отправляется на сервер Kubernetes API и постоянно обновляется. Следующий фрагмент представляет часть кода YAML, созданного командой `kubectl get node`, с полем `status`:

```
status:
  addresses:
    - address: 172.17.0.2
      type: InternalIP
    - address: kind-control-plane
      type: Hostname
```

Далее в документе YAML для этого узла можно найти поля `allocatable`, содержащие информацию о процессоре и памяти:

```
allocatable:
  ...
  capacity:
    cpu: "12"
    ephemeral-storage: 982940092Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 32575684Ki
    pods: "110"
```

В объекте `Node` имеются также другие поля, поэтому мы рекомендуем вам самим получить документы YAML и посмотреть, какую информацию сообщают узлы. Количество узлов может варьироваться от 0 до 15 000 (15 000 считаются текущим пределом из-за чрезмерного роста накладных расходов на поддержку конечных точек и выполнение других операций, интенсивно использующих метаданные). Информация в объекте `Node` имеет решающее значение, например, для планирования модулей `Pod`.

9.2 Основы kubelet

Мы знаем, что kubelet – это двоичная программа, которая устанавливается и запускается на каждом узле. Поэтому давайте погрузимся в мир kubelet и посмотрим, что он делает. Узлы и агенты kubelet бес-

полезны без среды выполнения контейнеров, на которую они полагаются, запуская контейнерные процессы. Далее мы кратко рассмотрим среду выполнения контейнеров.

9.2.1 *Среда выполнения контейнеров: стандарты и соглашения*

Для разворачивания образов контейнеров, которые являются tar-архивами, агент kubelet использует четко определенный API, позволяющий распаковать эти архивы и запустить двоичные файлы в них. Существует две стандартные спецификации, CRI и OCI, определяющие, как и что должен делать kubelet, чтобы запустить контейнер:

- *интерфейс CRI определяет, как.* Он предлагает ряд удаленных вызовов для запуска, остановки и управления контейнерами и образами. Любая среда выполнения контейнеров предлагает этот интерфейс как удаленный сервис;
- *интерфейс OCI определяет, что.* Это стандарт форматов образов контейнеров. Запуская или останавливая контейнер с помощью CRI, вы полагаетесь на то, что формат образа этого контейнера будет определенным образом стандартизирован. OCI определяет архив, содержащий другие архивы с метаданными.

Если у вас есть такая возможность, то создайте кластер, чтобы вместе с нами пройтись по примерам, что приводятся далее. Реализация CRI, как основная зависимость, должна передаваться агенту kubelet через аргумент командной строки или альтернативные настройки. Пример конфигурации containerd вы найдете в файле `/etc/containerd/config.toml` внутри действующего кластера kind, где можно посмотреть различные входные конфигурационные параметры, включая обработчики, определяющие провайдера CNI. Например:

```
# использовать v2 формата конфигурации
version = 2

# установка обработчиков времени выполнения для каждого модуля Pod
[plugins."io.containerd.grpc.v1.cri".containerd]
  default_runtime_name = "runc"
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  runtime_type = "io.containerd.runc.v2"

# настройка среды выполнения с магическим именем ("test-handler")
# для тестовых классов k8s ...
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.test-handler]
  runtime_type = "io.containerd.runc.v2"
```

В следующем примере используется kind для создания кластера Kubernetes v1.20.2. Обратите внимание, что в разных версиях Kubernetes результаты могут отличаться. Чтобы просмотреть файл в кластере kind, выполните следующие команды:

```

$ kind create cluster ← Создает кластер Kubernetes

$ export \
  KIND_CONTAINER=\
$(docker ps | grep kind | awk '{ print $1 }') ← Находит идентификатор контейнера
                                                Docker с контейнером kind
$ docker exec -it "$KIND_CONTAINER" /bin/bash ← Выполняется
                                                в запущенном контейнере
                                                и запускает интерактивную
                                                командную строку

root@kind-control-plane:/# \
cat /etc/containerd/config.toml ← Выводит конфигурационный
                                файл containerd

```

Мы не будем углубляться в детали реализации контейнера. Однако важно знать, что это базовая среда выполнения, от которой зависит агент kubelet. На входе он принимает провайдера CRI, реестр образов и значения среды выполнения, а это означает, что kubelet поддерживает множество разных реализаций механизмов контейнеризации (контейнеры VM, контейнеры gVisor и т. д.). Находясь в командной оболочке, работающей внутри контейнера kind, можно выполнить следующую команду:

```

root@kind-control-plane:/# ps axu | grep /usr/bin/kubelet
root          653 10.6  3.6 1881872 74020 ?
              Ssl  14:36   0:22 /usr/bin/kubelet
              --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
              --kubeconfig=/etc/kubernetes/kubelet.conf
              --config=/var/lib/kubelet/config.yaml
              --container-runtime=remote
              --container-runtime-endpoint=unix:///run/containerd/containerd.sock
              --fail-swap-on=false --node-ip=172.18.0.2
              --provider-id=kind://docker/kind/kind-control-plane
              --fail-swap-on=false

```

Она выведет список конфигурационных параметров и флагов командной строки, полученных агентом kubelet, работающим внутри контейнера kind. Эти параметры рассматриваются далее; однако мы охватим не все варианты, потому что их слишком много.

9.2.2 Конфигурационные параметры и API агента kubelet

kubelet – это точка интеграции для широкого спектра примитивов в ОС Linux. Некоторые из его структур данных отражают историю его развития. kubelet поддерживает более 100 различных параметров командной строки в двух разных категориях:

- *параметры* – управляют поведением низкоуровневых функций Linux, используемых с Kubernetes, таких как правила, связанные с использованием iptables, или конфигурация DNS;
- *варианты* – определяют жизненный цикл и работоспособность kubelet.

kubelet имеет множество крайних случаев, касающихся, например, обработки рабочих нагрузок Docker и containerd или управления ра-

бочими нагрузками Linux и Windows и т. д. Обсуждение каждого из этих крайних случаев может занять недели или даже месяцы. Поэтому так важно понимать организацию кодовой базы kubelet, чтобы вы могли копаться в ней и отыскивать необходимую информацию, столкнувшись с ошибкой или иным непредвиденным поведением.

ПРИМЕЧАНИЕ В версии Kubernetes v1.22 в kubelet было внесено довольно много изменений, в том числе были удалены встроенные провайдеры хранилищ, добавлена поддержка новых параметров безопасности по умолчанию с помощью флага `--seccomp-default`, реализована поддержка подкачки памяти (известная как NodeSwap) и улучшено качество обслуживания памяти. Если вам интересно узнать больше об усовершенствованиях в версии Kubernetes v1.22, мы настоятельно рекомендуем прочитать примечания по адресу <http://mng.bz/2jy0>. Что касается этой главы, то ошибка, недавно обнаруженная в kubelet, может привести к тому, что статические изменения манифеста модуля Pod нарушат работу долгоживущих модулей.

Файл `kubelet.go` является основной точкой входа для запуска kubelet. Папка `cmd` содержит определения флагов kubelet. (Определения флагов и параметров командной строки можно найти в файле `options.go`, доступном по адресу <http://mng.bz/REVK>.) Ниже приводится объявление структуры `kubeletFlags`. Она предназначена для флагов командной строки, но кроме нее имеются также значения API:

```
// kubeletFlags содержит конфигурационные флаги для kubelet.
// Конфигурационные значения должны записываться в kubeletFlags, а не в
// kubeletConfiguration, если выполняется хотя бы одно из следующих условий:
// - значение никогда не изменяется или не может быть безопасно изменено
//   на протяжении всего жизненного цикла узла, или
// - значение не может безопасно использоваться одновременно несколькими
//   узлами (например, имя хоста);
// структура kubeletConfiguration предназначена для настроек,
// общих для нескольких узлов.
// Не добавляйте сюда новые флаги или параметры, их и так слишком много.
type kubeletFlags struct {
```

Выше была представлена команда, с помощью которой мы искали файл `/usr/bin/kubelet`, и в ее выводе можно было увидеть строку `--config=/var/lib/kubelet/config.yaml`. Флаг `--config` определяет конфигурационный файл. Следующая команда выводит содержимое этого файла:

```
$ cat /var/lib/kubelet/config.yaml ← Выведет содержимое файла config.yaml
```

А далее показан результат ее выполнения:

```
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
```



```
anonymous:
  enabled: false
webhook:
  cacheTTL: 0s
  enabled: true
x509:
  clientCAFile: /etc/kubernetes/pki/ca.crt
authorization:
  mode: Webhook
  webhook:
    cacheAuthorizedTTL: 0s
    cacheUnauthorizedTTL: 0s
clusterDNS:
- 10.96.0.10
clusterDomain: cluster.local
cpuManagerReconcilePeriod: 0s
evictionHard:
  imagefs.available: 0%
  nodefs.available: 0%
  nodefs.inodesFree: 0%
evictionPressureTransitionPeriod: 0s
fileCheckFrequency: 0s
healthzBindAddress: 127.0.0.1
healthzPort: 10248
httpCheckFrequency: 0s
imageGCHighThresholdPercent: 100
imageMinimumGCAge: 0s
kind: kubeletConfiguration
logging: {}
nodeStatusReportFrequency: 0s
nodeStatusUpdateFrequency: 0s
rotateCertificates: true
runtimeRequestTimeout: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
syncFrequency: 0s
volumeStatsAggPeriod: 0s
```

Все API-значения для kubelet определены в файле `types.go`, доступном по адресу <http://mng.bz/wnJP>. В этом файле определяется структура данных API, содержащая входные конфигурационные параметры для kubelet, в том числе многие настраиваемые аспекты kubelet, на которые ссылается код в файле `kubelet.go`, доступном по адресу <http://mng.bz/j1YV>.

ПРИМЕЧАНИЕ В приведенных URL мы ссылаемся на версию Kubernetes 1.20.2, однако, читая эти строки, имейте в виду, что, несмотря на довольно частое изменение местоположения кода, сами объекты API изменяются очень редко.

Kubernetes API – это механизм или стандарт определения объектов API в Kubernetes и в его исходном коде.

В файле `types.go` можно заметить, что многие параметры, управляющие низкоуровневыми сетевыми операциями и процессами, передаются непосредственно в `kubelet` в виде входных данных. В следующем примере показана конфигурация `ClusterDNS`. Она чрезвычайно важна для нормальной работы кластера `Kubernetes`:

```
// ClusterDNS -- это список IP-адресов для сервера DNS кластера.
// Если это значение установлено, то the kubelet будет настраивать
// все контейнеры, используя для разрешения имен в DNS этот список
// вместо серверов DNS хостов.

ClusterDNS []string
```

Вместе с модулем `Pod` автоматически создается несколько файлов. Один из таких файлов: `/etc/resolv.conf`. Он используется сетевым стеком `Linux` для поиска в `DNS`, потому что этот файл определяет серверы `DNS`. Подробнее о создании модулей `Pod` мы поговорим в следующем разделе.

9.3 Создание модуля Pod и его мониторинг

Выполните следующие команды, чтобы создать модуль `Pod` с сервером `NGINX`. После запуска вы сможете просмотреть содержимое файла командой `cat`, как показано ниже:

```
$ kubectl run nginx --generator=run-pod/v1 \
  --image nginx ← Запуск модуля Pod

$ kubectl exec -it nginx -- /bin/bash ← Вход в командную оболочку
                                     работающего контейнера с NGINX

root@nginx:/# cat /etc/resolv.conf ← Запуск команды cat для просмотра
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
                                     содержимого файла resolv.conf
```

Теперь должно быть понятно, как при создании модуля `Pod` `kubelet` создает и монтирует файл `resolv.conf`. Теперь `Pod` может выполнять поиск в `DNS` и вы, если хотите, можете попробовать выполнить команду `ping google.com`. Вот еще несколько интересных структур в файле `types.go`:

- `ImageMinimumGCAge` (для удаления устаревших образов) – в долго работающих системах образы могут со временем заполнить дисковое пространство;
- `kubeletCgroups` (для корневых контрольных групп и драйверов `Pod`) – пул ресурсов `Pod` может управляться `systemd`, и эта структура унифицирует администрирование всех процессов с администрированием контейнеров;

- EvictionHard (для жестких ограничений) – эта структура определяет, когда удалять модули Pod при увеличении нагрузки на систему;
- EvictionSoft (для мягких ограничений) – эта структура определяет, как долго kubelet должен ждать, прежде чем вытеснить Pod, потребляющий больше всех ресурсов.

Это лишь некоторые из параметров в файле types.go; в kubelet их сотни. Все эти параметры можно передавать через аргументы командной строки в виде значений по умолчанию или определять в конфигурационных файлах YAML.

9.3.1 Запуск kubelet

Когда запускается узел, возникает целая череда событий, которые в конечном итоге приводят узел в состояние доступности для планирования. Обратите внимание, что порядок событий может меняться в зависимости от изменений в кодовой базе kubelet и из-за асинхронной природы Kubernetes в целом. На рис. 9.1 показана схема запуска kubelet. Глядя на шаги, можно заметить, что:

- выполняется несколько простых проверок работоспособности, помогающих убедиться, что kubelet сможет запускать модули Pod (контейнеры), – проверяются значения в NodeAllocatable, определяющие, сколько процессоров и памяти доступно;
- запускается процедура containerManager – основной цикл обработки событий в kubelet;

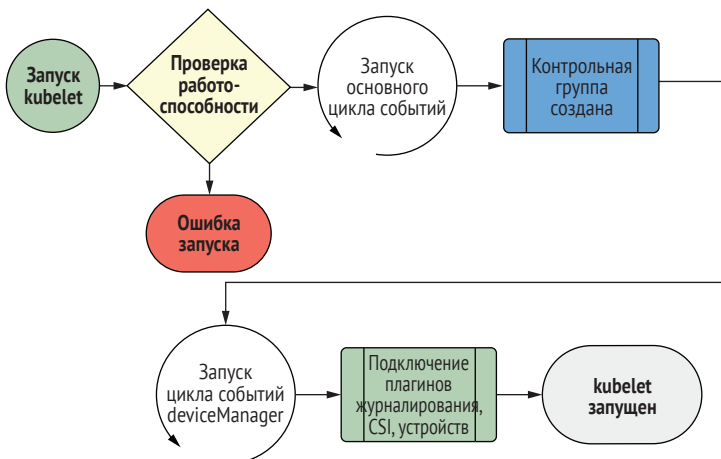


Рис. 9.1 Цикл запуска kubelet

- добавляется контрольная группа. При необходимости она создается с помощью функции `setupNode`. Планировщик и диспетчер контроллеров «замечают», что в системе появился новый узел.

Посредством сервера API они «наблюдают» за его состоянием – способностью запускать процессы и периодичностью отправки контрольных сигналов серверу API. Если kubelet пропустит отправку очередного контрольного сообщения, то в какой-то момент диспетчер контроллеров исключит узел из кластера;

- запускается цикл обработки событий диспетчера устройств `deviceManager`, который подключает внешние устройства к kubelet. Затем информация об этих устройствах передается как часть непрерывных обновлений (упомянутых на предыдущем шаге);
- к kubelet подключаются и регистрируются плагины журналирования, CSI и устройств.

9.3.2 *После запуска: жизненный цикл узла*

В более ранних версиях Kubernetes (до 1.17) объект `Node` обновлялся каждые 10 с вызовами сервера API из kubelet. По своей природе kubelet довольно плотно взаимодействует с сервером API, потому что плоскость управления в кластере должна знать, исправны узлы или нет. Если понаблюдать за запуском кластера, то можно заметить, что kubelet пытается связаться с плоскостью управления и будет повторять попытки, пока они не увенчаются успехом. В процессе своей работы плоскость управления иногда может оказываться недоступной, и узлы знают об этом. В процессе запуска kubelet также настраивает сетевой уровень, заставляя провайдера CNI инициализировать надлежащие сетевые функции, необходимые для работы сети.

9.3.3 *Механизм аренды и блокировки в etcd, эволюция аренды узла*

Для оптимизации производительности больших кластеров и уменьшения сетевого трафика в Kubernetes 1.17 была реализована конечная точка сервера API для управления узлами через *механизм аренды* в etcd. В etcd реализована концепция аренды, чтобы компоненты с высокой доступностью (Highly Available, HA), которым может потребоваться аварийное переключение, могли положиться на центральный механизм аренды и блокировки вместо реализации собственного механизма.

Любой, знакомый с семафорами, легко поймет, почему создатели Kubernetes не хотели впадать в зависимость от множества доморощенных реализаций блокировок для различных компонентов. Состояние kubelet поддерживается двумя независимыми циклами управления:

- *агент kubelet обновляет объект `NodeStatus` каждые 5 мин, чтобы сообщить серверу API о своем состоянии.* Например, если перезагрузить узел после изменения объема доступной ему памяти, то через 5 мин это обновление можно увидеть в объекте `Node-`

Status на сервере API. Если вам интересно, насколько велика эта структура данных, запустите `kubectl get nodes -o yaml` на большом промышленном кластере. Скорее всего, вы увидите десятки тысяч строк текста, не менее 10 Кбайт на узел;

- дополнительно каждые 10 с kubelet обновляет (относительно небольшой) объект Lease. Эти обновления позволяют контроллерам в плоскости управления Kubernetes вытеснить узел в течение нескольких секунд, если он отключился, без больших затрат на отправку большого объема информации о состоянии.

9.3.4 Управление жизненным циклом Pod в kubelet

После завершения всех предварительных проверок kubelet запускает большой цикл синхронизации: процедуру `containerManager`. Она поддерживает жизненный цикл модуля Pod, состоящий из цикла управления действиями. На рис. 9.2 показан жизненный цикл модуля Pod и шаги, связанные с управлением им.

- 1 Начало жизненного цикла модуля Pod.
- 2 Проверка возможности запуска Pod на узле.
- 3 Настройка хранилища и сети (CNI).
- 4 Запуск контейнеров посредством CRI.
- 5 Мониторинг модуля Pod.
- 6 Перезапуск.
- 7 Остановка.

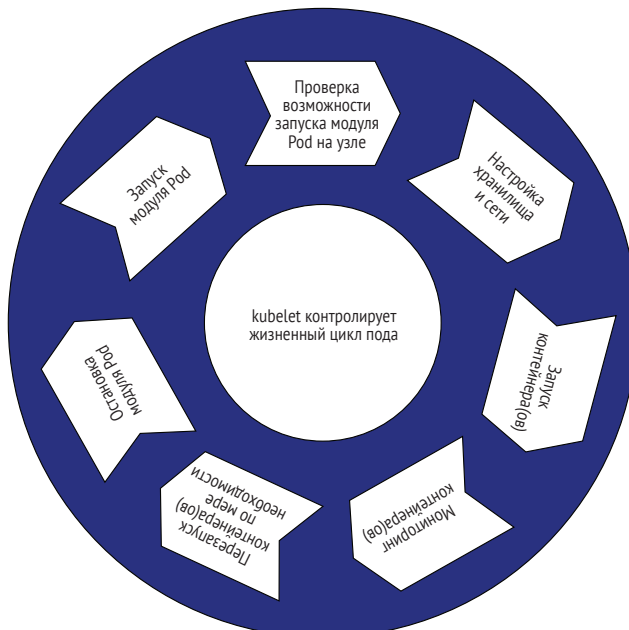


Рис. 9.2 Жизненный цикл модуля Pod

На рис. 9.3 изображен жизненный цикл контейнера на узле Kubernetes. Как показано на рисунке:

- 1 пользователь или контроллер набора реплик принимает решение создать Pod с помощью Kubernetes API;
- 2 планировщик отыскивает подходящий узел (например, хост с IP-адресом 1.2.3.4);

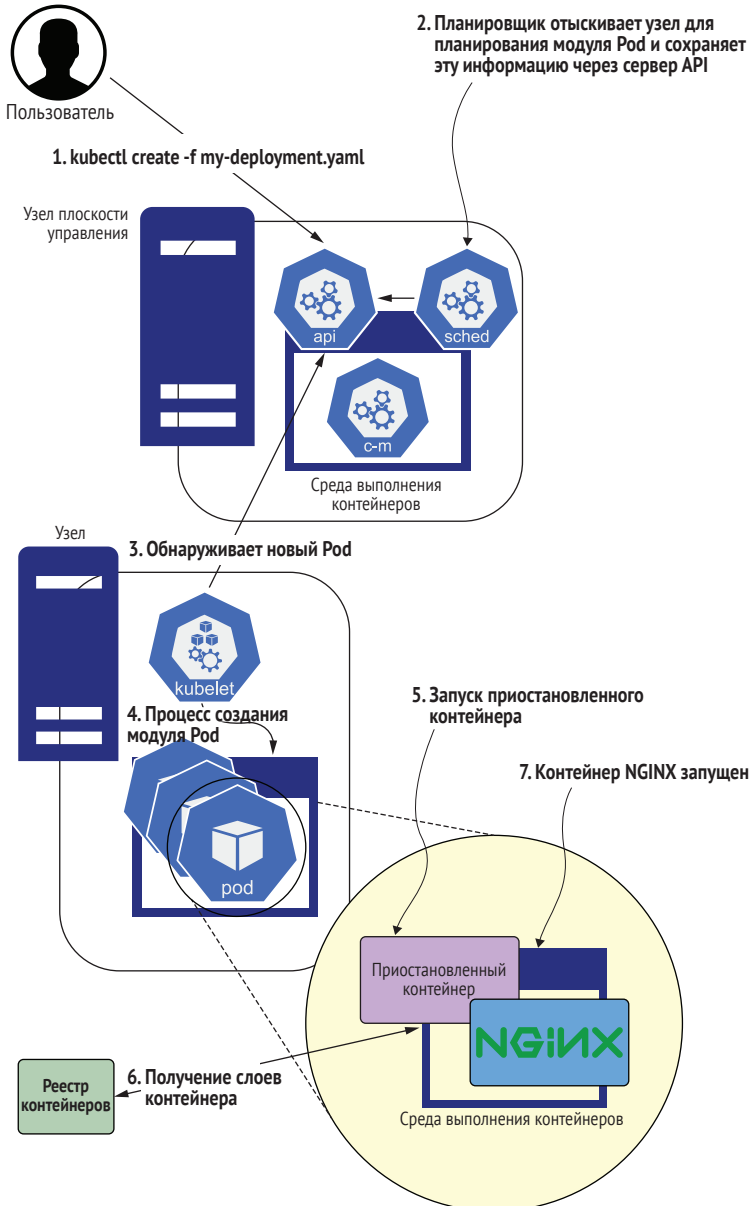


Рис. 9.3 Создание модуля Pod

- 3 kubelet на хосте 1.2.3.4 получает от сервера API новые данные о модулях Pod, размещенных на его узле, и замечает, что Pod еще не запущен;
- 4 запускается процесс создания модуля Pod;
- 5 приостановленный контейнер (контейнер pause) имеет защищенное окружение, куда будут помещены запрошенные контейнеры, и определяет пространства имен Linux и IP-адреса, созданные для него агентом kubelet и провайдером CNI (сетевой интерфейс контейнера);
- 6 kubelet взаимодействует со средой выполнения контейнеров, извлекает слои контейнера и запускает фактический образ;
- 7 запускается контейнер NGINX.

Если что-то пойдет не так, например если контейнер потерпит аварию или проверка его работоспособности завершится неудачей, то модуль Pod может быть перемещен на новый узел. Это называется *перепланированием*. Мы упомянули здесь о приостановленном контейнере, который используется для создания общих пространств имен Linux. Подробнее о нем мы поговорим позже в этой главе.

9.3.5 CRI, контейнеры и образы: как они связаны

Одна из задач kubelet – управление образами. Возможно, вы знакомы с этим процессом, если когда-либо запускали `docker gm -a -q` или `docker images --prune` на своем ноутбуке. kubelet запускает только контейнеры, но эти контейнеры в конечном итоге полагаются на *базовые образы*, которые извлекаются из реестров образов. Один из таких реестров – Docker Hub.

Новый слой поверх существующих образов создает контейнер. Обычно образы используют одни и те же слои, которые кешируются средой выполнения контейнеров, работающей под управлением kubelet. Время хранения в кеше зависит от настроек сборщика мусора в самом kubelet. Старые образы периодически удаляются из постоянно растущего кеша реестра. Этот процесс оптимизирует запуск контейнера и предотвращает заполнение дискового пространства образами, которые больше не используются.

9.3.6 kubelet не запускает контейнеры: это делает CRI

Среда выполнения контейнеров (CRI) предоставляет средства управления контейнерами, которые kubelet должен запускать. Не забывайте, что сам агент kubelet не может запускать контейнеры: в этом он полагается на среду выполнения, такую как containerd или runC.

Скорее всего, независимо от версии Kubernetes у вас установлена среда runC. С ее помощью можно эффективно запустить любой образ вручную. Например, выполните команду `docker ps`, чтобы получить список контейнеров, работающих локально. Также можно экспор-

тировать образы в виде архива. В нашем случае мы можем сделать следующее:

```
$ docker ps ← Получение идентификатора образа
d32b87038ece kindest/node:v1.15.3
"/usr/local/bin/entr..." kind-control-plane
$ docker export d32b > /tmp/whoneedsdocker.tar ← Экспорт образа в архив
$ mkdir /tmp/whoneedsdocker
$ cd /tmp/whoneedsdocker
$ tar xf /tmp/whoneedsdocker.tar ← Распаковка архива
$ runc spec ← Запуск runC
```

Эти команды создадут файл config.json. Например:

```
{
  "ociVersion": "1.0.1-dev",
  "process": {
    "terminal": true,
    "user": {
      "uid": 0,
      "gid": 0
    },
    "args": [
      "sh"
    ]
  },
  "namespaces": [
    {
      "type": "pid"
    },
    {
      "type": "network"
    },
    {
      "type": "ipc"
    },
    {
      "type": "uts"
    },
    {
      "type": "mount"
    }
  ]
}
```

Часто желательно определить в разделе args команду, которую по умолчанию будет запускать runC, чтобы сделать что-то полезное (например, `python mycontainerizedapp.py`). Мы опустили большую часть кода в предыдущем примере файла config.json, но сохранили важную его часть: раздел namespaces.

9.3.7 Приостановленный контейнер: момент истины

Каждый контейнер в модуле Pod соответствует действию `runC`. Поэтому нам нужен приостановленный контейнер, являющийся прародителем всех контейнеров. Приостановленный контейнер:

- ждет, пока станет доступно сетевое пространство имен, чтобы все контейнеры в модуле Pod могли использовать один IP-адрес и взаимодействовать друг с другом, используя IP-адрес 127.0.0.1;
- ждет, пока станет доступна файловая система, чтобы все контейнеры в модуле Pod могли обмениваться данными через `emptyDir`.

После настройки Pod каждый вызов `runC` принимает одни и те же параметры из пространства имен. Несмотря на то что `kubelet` не запускает контейнеры, в нем сосредоточено довольно много логики, связанной с созданием модулей Pod, которыми `kubelet` должен управлять. Например, он проверяет готовность сети и хранилища для контейнеров. Это упрощает реализацию распределенных сценариев. Запуску контейнера предшествуют другие действия, такие как извлечение образов, которые мы рассмотрим далее в этой главе. Но прежде мы должны отступить немного назад и поближе познакомится с CRI, чтобы более четко понять, где проходит граница между средой выполнения контейнеров и `kubelet`.

9.4 Интерфейс времени выполнения контейнеров (CRI)

Программа `runC` – это лишь часть общей схемы, участвующей в запуске контейнеров в Kubernetes. Основная магия заключена в интерфейсе CRI, который абстрагирует `runC` вместе с другими функциями и обеспечивает планирование на более высоком уровне, управление образами и деятельность среды выполнения контейнеров.

9.4.1 Сообщаем Kubernetes, где находится среда выполнения контейнеров

Как сообщить Kubernetes, где находится сервис CRI? Заглянув внутрь действующего кластера `kind`, можно увидеть, что `kubelet` запускается со следующими двумя параметрами:

```
--container-runtime=remote  
--container-runtime-endpoint=/run/containerd/containerd.sock
```

Для взаимодействий с конечной точкой среды выполнения контейнеров `kubelet` использует gRPC, фреймворк вызова удаленных про-

цедур (Remote Procedure Call, RPC); сам `containerd` имеет встроенный плагин CRI. Значение `remote` подразумевает, что Kubernetes может использовать сокет `containerd` в качестве минимальной реализации интерфейса для создания и управления модулями Pod и их жизненными циклами. CRI – это минимальный интерфейс, который должна реализовать любая среда выполнения контейнеров. Этот интерфейс был разработан, чтобы сообщество могло быстро внедрять различные среды выполнения контейнеров (кроме Docker), а также подключать их к Kubernetes.

ПРИМЕЧАНИЕ Несмотря на модульную организацию Kubernetes в смысле запуска контейнеров, он по-прежнему имеет хранимое состояние. Нельзя на ходу отключить среду выполнения контейнеров от действующего кластера Kubernetes, не опустошив (и потенциально удалив) узел из работающего кластера. Это ограничение связано с метаданными и контрольными группами, которые создает и поддерживает kubelet.

CRI – это интерфейс gRPC, поэтому в идеале параметр `container-runtime` в Kubernetes должен определяться со значением `remote`. CRI описывает создание всех контейнеров через интерфейс, и по аналогии с хранилищами и сетью разработчики Kubernetes предполагают со временем вынести логику среды выполнения контейнеров из ядра Kubernetes.

9.4.2 *Процедуры CRI*

CRI состоит из четырех высокоуровневых интерфейсов Go, включающих все основные функции, необходимые Kubernetes для запуска контейнеров. Вот эти интерфейсы:

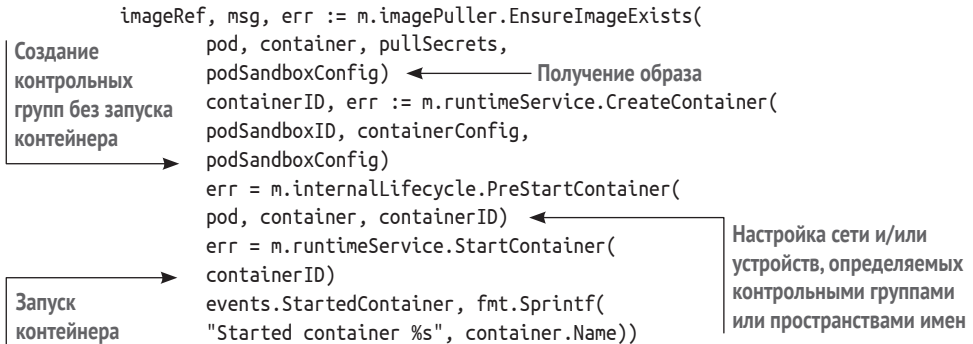
- *PodSandboxManager* – создает окружение установки для модулей Pod;
- *ContainerRuntime* – запускает, выполняет и останавливает контейнеры;
- *ImageService* – извлекает, перечисляет и удаляет образы;
- *ContainerMetricsGetter* – сообщает количественную информацию о запущенных контейнерах.

Эти интерфейсы обеспечивают функции приостановки, извлечения, а также создания изолированного окружения («песочницы»). Kubernetes ожидает, что все эти интерфейсы будут реализованы любым удаленным CRI, и взаимодействует с ними посредством gRPC.

9.4.3 *Абстракция kubelet вокруг CRI: GenericRuntimeManager*

От CRI не требуется охватывать все возможности оркестровки контейнеров, такие как поиск и удаление устаревших образов, управле-

ние журналами контейнеров и поддержка жизненного цикла. Агент kubelet предоставляет интерфейс среды выполнения, реализованный `kuberuntime.NewKubeGenericRuntimeManager` – оберткой вокруг произвольного провайдера CRI (`containerd`, `CRI-O`, `Docker` и т. д.). Диспетчер среды выполнения (<http://mng.bz/lxaM>) управляет вызовами всех четырех основных интерфейсов CRI. Давайте для примера посмотрим, что происходит при создании нового модуля Pod:



Возможно, вам интересно, зачем производится вызов `PreStartContainer`. Эта функция, кроме всего прочего, подключает конкретные сетевые плагины и драйверы GPU, которые настраиваются с использованием информации, относящейся к контрольным группам, до запуска процесса, использующего сеть или GPU.

9.4.4 Как вызывается CRI?

Удаленные вызовы CRI в предыдущем фрагменте кода прячутся за несколькими строками кода, из которого, кстати, мы удалили много лишнего. Чуть ниже мы подробно рассмотрим функцию `EnsureImageExists`, но прежде предлагаем посмотреть, как Kubernetes абстрагирует низкоуровневую функциональность CRI за двумя основными API, которые используются внутри kubelet для работы с контейнерами.

9.5. Интерфейсы kubelet

В исходном коде kubelet определены различные интерфейсы Go. Мы рассмотрим их в следующих нескольких разделах, чтобы вы могли получить представление о внутренней работе kubelet.

9.5.1 Внутренний интерфейс среды выполнения

CRI в Kubernetes делится на три части: `Runtime`, `StreamingRuntime` и `CommandRunner`. Интерфейс `KubeGenericRuntime` (в файле `kuberuntime_manager.go`, доступном по адресу <http://mng.bz/BMxg>) исполь-

зуется внутри Kubernetes и служит оберткой для основных функций среды выполнения CRI. Например:

```
type KubeGenericRuntime interface {
    kubecontainer.Runtime
    kubecontainer.StreamingRuntime
    kubecontainer.CommandRunner
}
```

Определяет интерфейс, заданный провайдером CRI

Определяет функции для обработки потоковых вызовов (например, exec/attach/port-forward)

Определяет функцию, которая выполняет команду в контейнере и возвращает результат

Для поставщиков это означает, что сначала нужно реализовать интерфейс Runtime, а затем интерфейс StreamingRuntime, потому что интерфейс Runtime описывает большую часть основных функций Kubernetes (<http://mng.bz/1jXj> и <http://mng.bz/PWdn>). Клиенты gRPC – это функции, помогающие понять, как kubelet взаимодействует с CRI. Они определены в структуре kubeGenericRuntimeManager. В частности, runtimeService internalapi.RuntimeService взаимодействует с провайдером CRI.

Внутри RuntimeService имеется ContainerManager, реализующий главное волшебство. Этот интерфейс является частью фактического определения CRI. В следующем фрагменте показаны примеры вызовов функций провайдера CRI для запуска, остановки и удаления контейнеров:

```
// ContainerManager содержит методы для манипулирования контейнерами,
// работающими под управлением среды выполнения. Все методы потокобезопасные.

type ContainerManager interface {
    // CreateContainer создает новый контейнер в указанном окружении PodSandbox.
    CreateContainer(podSandboxID string, config
        *runtimeapi.ContainerConfig, sandboxConfig
        *runtimeapi.PodSandboxConfig) (string, error)
    // StartContainer запускает контейнер.
    StartContainer(containerID string) error
    // StopContainer останавливает запущенный контейнер.
    StopContainer(containerID string, timeout int64) error
    // RemoveContainer удаляет контейнер.
    RemoveContainer(containerID string) error
    // ListContainers выводит отфильтрованный список контейнеров.
    ListContainers(filter *runtimeapi.ContainerFilter)
        ([]*runtimeapi.Container, error)
    // ContainerStatus возвращает состояние контейнера.
    ContainerStatus(containerID string)
        (*runtimeapi.ContainerStatus, error)
    // UpdateContainerResources обновляет ресурсы cgroup для контейнера.
    UpdateContainerResources(
        containerID string, resources *runtimeapi.LinuxContainerResources)
        error
```

```
// ExecSync выполняет команду в контейнере.
// Если команда завершится с ненулевым кодом выхода,
// то возвращается ошибка error.
ExecSync(containerID string, cmd []string, timeout time.Duration)
    (stdout []byte, stderr []byte, err error)
// Exec подготавливает конечную точку потоковой передачи к выполнению...,
// возвращает адрес.
Exec(*runtimeapi.ExecRequest) (*runtimeapi.ExecResponse, error)
// Attach подготавливает конечную точку потоковой передачи к подключению
// к работающему контейнеру и возвращает адрес.
Attach(req *runtimeapi.AttachRequest)
    (*runtimeapi.AttachResponse, error)
// ReopenContainerLog требует от среды выполнения повторно открыть
// файл журнала stdout/stderr для контейнера.
// Если вернет ошибку, то новый файл журнала контейнера НЕ ДОЛЖЕН
// создаваться.
ReopenContainerLog(ContainerID string) error
}
```

9.5.2 Как kubelet извлекает образы: интерфейс ImageService

За подпрограммами среды выполнения контейнеров скрывается интерфейс ImageService, определяющий несколько основных методов: PullImage, GetImage, ListImages и RemoveImage. Идея извлечения образа, исходящая из семантики Docker, является частью спецификации CRI. Определение этого интерфейса можно увидеть в том же файле (runtime.go), что и другие интерфейсы. Таким образом, каждая среда выполнения контейнеров реализует следующие функции:

```
// Интерфейс ImageService позволяет работать с сервисом образов.
type ImageService interface {
    PullImage(image ImageSpec, pullSecrets []v1.Secret,
        podSandboxConfig *runtimeapi.PodSandboxConfig)
        (string, error)
    GetImageRef(image ImageSpec) (string, error)
    // Возвращает список всех образов, находящихся в настоящее время на машине.
    ListImages() ([]Image, error)
    // Удаляет указанный образ.
    RemoveImage(image ImageSpec) error
    // Возвращает статистику по образам.
    ImageStats() (*ImageStats, error)
}
```

Среда выполнения контейнеров может вызывать `docker pull` для извлечения образа. Точно так же она может вызвать `docker run` для создания контейнера. Среда выполнения контейнеров, как вы наверняка помните, может быть установлена в kubelet при его запуске с помощью флага `container-runtime-endpoint`:

```
--container-runtime-endpoint=unix:///var/run/crio/crio.sock
```

9.5.3 Передача *ImagePullSecret* в *kubelet*

Давайте конкретизируем связь между *kubectl*, *kubelet* и интерфейсом CRI. Для этого посмотрим, как можно передать информацию агенту *kubelet*, чтобы он мог безопасно загружать образы из частного реестра. Ниже приводится определение YAML объектов Pod и Secret. Pod ссылается на безопасный реестр, требующий передачи учетных данных, а Secret хранит эти учетные данные:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: my.secure.registry/container1:1.0
    imagePullSecrets:
    - name: my-secret
---
apiVersion: v1
data:
  .dockerconfigjson: sojosaidjfwoeij2f0ei8f...
kind: Secret
metadata:
  creationTimestamp: null
  name: my-secret
selfLink: /api/v1/namespaces/default/secrets/my-secret
type: kubernetes.io/dockerconfigjson
```

Для этого фрагмента вам нужно самостоятельно сгенерировать значение `.dockerconfigjson`. Это можно сделать в интерактивном режиме, используя *kubectl*, например:

```
$ kubectl create secret docker-registry my-secret
--docker-server my.secure.registry
--docker-username my-name --docker-password 1234
--docker-email jay@apache.org
```

Или воспользоваться эквивалентной командой, если у вас уже есть конфигурационный JSON-файл Docker:

```
$ kubectl create secret generic regcred
--from-file=.dockerconfigjson=<path/to/.docker/config.json>
--type=kubernetes.io/dockerconfigjson
```

Эта команда создаст полную конфигурацию Docker и поместит ее в файл `.dockerconfigjson`, а затем использует его при извлечении образов через *ImageService*. Еще более важно, что этот сервис в конечном

итоге вызывает функцию `EnsureImageExists`. Затем можно запустить `kubectl get secret -o yaml`, чтобы просмотреть определение секрета `Secret` и скопировать его. Затем используйте Base64 для декодирования, чтобы увидеть, как выглядит токен входа в Docker, который использует kubelet.

Теперь, узнав, как демон Docker использует `Secret` при извлечении образов, вернемся к обзору механизма в Kubernetes, обеспечивающего возможность работы через секреты `Secret`, управляемые фреймворком Kubernetes. Ключом является интерфейс `ImageManager`, реализующий эту функциональность в методе `EnsureImageExists`. Этот метод вызывает внутреннюю функцию `PullImage`, если необходимо, в зависимости от значения `ImagePullPolicy` в определении объекта `Pod`. Следующий фрагмент демонстрирует, как передаются секреты, необходимые для извлечения:

```
type ImageManager interface {
    EnsureImageExists(pod *v1.Pod, container *v1.Container,
        pullSecrets []v1.Secret,
        podSandboxConfig *runtimeapi.PodSandboxConfig)
    (string, string, error)
}
```

Функция `EnsureImageExists` получает секреты `Secret`, созданные в документе YAML выше и необходимые для извлечения образов. Затем выполняется защищенное извлечение командой `docker pull` путем десериализации значения `dockerconfigjson`. Как только демон загрузит образ, Kubernetes сможет двинуться дальше и запустить `Pod`.

9.6 Дополнительная литература

- M. Crosby. «What is containerd?» Docker blog. <http://mng.bz/Nxq2> (доступно по состоянию на 27.12.21).
- J. Jackson. «GitOps: 'Git Push' All the Things». <http://mng.bz/6Z5G> (доступно по состоянию на 27.12.21).
- «How does copy-on-write in fork() handle multiple fork?». Документация Stack Exchange. <http://mng.bz/Exql> (доступно по состоянию на 27.12.21).
- «Deep dive into Docker storage drivers». Видео на YouTube. https://www.youtube.com/watch?v=9oh_M11-foU (доступно по состоянию на 27.12.21).

Итоги

- kubelet запускается на каждом узле и контролирует жизненный цикл модулей `Pod`.

- kubelet взаимодействует со средой выполнения контейнеров для создания, запуска, остановки и удаления контейнеров.
- kubelet поддерживает возможность настройки различных функций (например, срок удаления модулей Pod).
- При запуске kubelet выполняет различные проверки работоспособности на узле, создает контрольные группы и запускает различные плагины, такие как CSI.
- kubelet управляет жизненным циклом модуля Pod: запуском, поддержкой его выполнения, созданием хранилища и сети, мониторингом, перезапуском и остановкой.
- CRI определяет порядок взаимодействий kubelet с используемой средой выполнения контейнеров.
- kubelet основан на различных интерфейсах Go. К ним относятся интерфейсы для взаимодействий с CRI, извлечения образов и собственные интерфейсы kubelet.

10

DNS в Kubernetes

В этой главе:

- обзор сервиса DNS в кластерах Kubernetes;
- иерархия DNS;
- исследование сервиса DNS по умолчанию в Pod;
- настройка CoreDNS.

DNS существует ровно столько же, сколько существует интернет. Микросервисы усложняют управление записями DNS, потому что требуют резкого увеличения использования доменных имен в центре обработки данных. Стандарты Kubernetes делают разрешение имен модулей Pod чрезвычайно простым, поэтому отдельным приложениям редко требуется следовать сложным правилам для поиска нижестоящих сервисов. Обычно разрешение имен обеспечивается сервисом CoreDNS (<https://github.com/coredns/coredns>), которому посвящена эта глава.

10.1 Краткое введение в DNS (и CoreDNS)

Работа любого сервера DNS заключается в отображении доменных имен (например, www.google.com) в соответствующие им IP-адреса (например, 142.250.72.4). Серверы DNS поддерживают множество

записей, которые мы используем каждый день при просмотре веб-страниц. Давайте рассмотрим некоторые из них.

10.1.1 NXDOMAIN, записи A и записи CNAME

В Kubernetes разрешение имен происходит автоматически, по крайней мере в кластерах. Однако нам все же нужно определить некоторые термины, чтобы сделать эту главу более конкретной, особенно в ситуациях, когда имеет место нестандартное поведение DNS (например, в отношении автономных сервисов, как показано в этой главе). Вот некоторые определения, которые желательно знать:

- *ответы NXDOMAIN* – ответы DNS, которые возвращаются, если для заданного доменного имени не найден IP-адрес;
- *отображения A и AAAA* – получают имя хоста на входе и возвращают адрес IPv4 или IPv6 (например, они получают имя google.com и возвращают 142.250.72.4);
- *отображения CNAME* – возвращают псевдоним для некоторых доменных имен (например, получают www.google.com и возвращают google.com).

В собственных окружениях CNAME имеют решающее значение для обратной совместимости клиентов API и других приложений, зависящих от сервисов. В следующем фрагменте показан пример, как смешиваются имена A и записи CNAME. Эти записи находятся в так называемых *файлах зон*. Файл зоны напоминает длинный CSV-файл записей (только без запятых):

```
my.very.old.website CNAME my.new.site.  
my.old.website. CNAME my.new.site.  
my.new.site. A 192.168.10.123
```

Немного похоже на содержимое файла /etc/hosts, верно? Файл /etc/hosts в ОС Linux – это всего лишь локальная конфигурация DNS, которая проверяется перед тем, как компьютер пошлет запрос в интернет, чтобы найти другие хосты, соответствующие DNS-именам, которые вы вводите в браузере, а сервер DNS вернет записи ANAME и CNAME. Еще до Kubernetes существовало множество различных реализаций DNS-серверов, в том числе:

- рекурсивные, т. е. способные разрешить почти любые имена, имеющиеся в интернете, начиная с корневой записи DNS (например, .edu или .com) и спускаясь вниз. Один из таких серверов – BIND. Он часто используется в центрах обработки данных;
- облачные и интегрированные в облако (например, Route53 в AWS). Они не устанавливаются конечными пользователями;

- в большинстве кластеров Kubernetes используется CoreDNS – внутрикластерный сервер DNS, обслуживающий модули Pod;
- наборы тестов Kubernetes Conformance, подтверждающие обладание определенными чертами DNS и включающие:
 - файлы `/etc/hosts` в модулях Pod, чтобы они могли автоматически обращаться к серверу API через внутреннее имя хоста `kubernetes.default`;
 - модули Pod, которым разрешено внедрять свои записи DNS;
 - произвольные сервисы, в том числе автономные, имена которых должны преобразовываться модулями Pod в записи A;
 - модули Pod со своими DNS-записями.

Для реализации такого поведения в Kubernetes необязательно использовать CoreDNS, но его применение упрощает задачу. Все, что действительно важно, – дистрибутив Kubernetes должен соответствовать спецификации DNS для Kubernetes. Но, как бы то ни было, вы почти наверняка будете использовать CoreDNS в своих кластерах, и тому есть веские причины. Это единственный общедоступный сервис DNS с открытым исходным кодом и встроенной поддержкой Kubernetes. Он способен:

- подключаться к серверу Kubernetes API и получать IP-адреса для модулей Pod и сервисов Service;
- преобразовывать записи DNS в IP-адреса модулей Pod и сервисов внутри кластера;
- кешировать записи DNS для эффективной работы больших кластеров, где работают сотни модулей Pod, которым требуется разрешать имена сервисов;
- подключать плагины с новыми возможностями во время компиляции (не во время выполнения);
- масштабироваться и гарантировать чрезвычайно низкие задержки даже в окружениях с высокой нагрузкой;
- перенаправлять запросы вышестоящим серверам DNS (через плагин <https://coredns.io/plugins/forward/>) для разрешения внешних имен в кластере.

CoreDNS способен на многое, но он не перенаправляет запросы для разрешения внешних имен другим вышестоящим серверам, которые предоставляют возможности рекурсивного DNS. CoreDNS позволяет разрешать IP-адреса сервисов, находящихся в сети кластера, а также модулей Pod (как мы увидим чуть ниже).

На рис. 10.1 показаны отношения между CoreDNS и другими серверами DNS (такими как BIND). Любой сервер DNS должен реализовать базовые функции разрешения имен. Сервер CoreDNS был создан после Kubernetes и поэтому тоже имеет явную поддержку Kubernetes DNS.

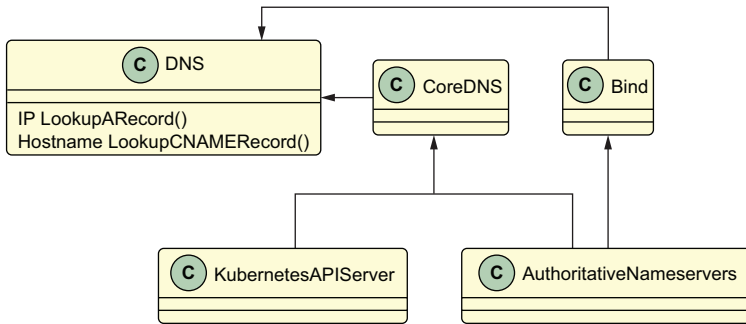


Рис. 10.1 Отношения между CoreDNS и другими серверами DNS

10.1.2 Модулям Pod нужен внутренний DNS

Доступ ко всем модулям Pod в микросервисной среде обычно осуществляется через сервис, а модули могут появляться и исчезать (т. е. у них меняются IP-адреса), поэтому DNS – это основной способ доступа к любому сервису. Это верно и для облака, и для интернета в целом. Прошли те времена, когда вам давался IP-адрес определенного сервера или базы данных. Давайте посмотрим, как модули Pod в кластере могут связываться друг с другом через DNS, для чего запустим многоконтейнерный сервис и опробуем его:

```

apiVersion: v1
kind: Service
metadata:
  name: nginx4
  labels:
    app: four-of-us
spec:
  ports:
    - port: 80 ← Предоставляет порт нашего сервиса
      name: web
  clusterIP: None
  selector:
    app: four-of-us
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-ss
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: four-of-us
  template:
    metadata:

```

```

labels:
  app: four-of-us
spec:
  containers:
    - name: nginx
      image: nginx:1.7
    ports:
      - containerPort: 80
        name: web
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-dep
spec:
  replicas: 2
  selector:
    matchLabels:
      app: four-of-us
  template:
    metadata:
      labels:
        app: four-of-us
    spec:
      containers:
        - name: nginx
          image: nginx:1.7
          ports:
            - containerPort: 80
              name: web

```

Старая версия NGINX, позволяющая использовать командную оболочку внутри модуля NGINX

Для сравнения работы DNS в разных типах модулей Pod

Порт сервиса играет важную роль в нашем примере, потому что нам интересно, как DNS разрешает имена. Также обратите внимание, что в примере используется старая версия NGINX, позволяющая использовать командную оболочку внутри модуля NGINX. Более новые контейнеры NGINX не включают командную оболочку из соображений безопасности. Наконец, на этот раз мы используем StatefulSet, чтобы сравнить работу DNS в разных типах модулей Pod.

ПРИМЕЧАНИЕ Используемый контейнер NGINX позволяет использовать командную оболочку, чтобы заглянуть внутрь. Более новые контейнеры NGINX не дают такого удобства. В этой книге мы несколько раз упоминали такие минималистичные контейнеры (действительно компактные, не имеющие полноценной операционной системы и потому более безопасные, потому что не имеют командной оболочки, с помощью которой можно было бы осуществить взлом). В настоящее время все чаще встречаются контейнеры без командной оболочки, в которую можно было бы войти. Также в практике все чаще встречаются базовые образы контейнеров – контейнеры без *дистрибутива*. Для создания безопасных контейнеров с мини-

мальным набором программного обеспечения по умолчанию мы рекомендуем использовать такие образы без дистрибутива с микросервисами, необходимыми приложению, не имеющие лишнего программного обеспечения, способного увеличить уязвимость с точки зрения общего перечня уязвимостей и рисков (Common Vulnerabilities and Exposures, CVE). Эта идея рассматривается в главе 13. Чтобы узнать больше о создании приложений на основе базовых образов без дистрибутива, посетите страницу <https://github.com/GoogleContainerTools/distroless>.

Прежде чем приступить к экспериментам, рассмотрим кратко суть объектов StatefulSet и где они используются в Kubernetes. Часто они определяют интересные свойства и требования к DNS.

10.2 Почему StatefulSet, а не Deployment?

В этой главе мы создадим Pod, работающий в так называемом StatefulSet. Объекты StatefulSet обладают интересными свойствами, когда речь заходит о DNS, поэтому мы используем этот Pod для исследования возможностей и ограничений Kubernetes в отношении запуска процессов высокой доступности с надежными конечными точками DNS. Объекты StatefulSet чрезвычайно важны для приложений с четко установленной идентичностью, таких как:

- Apache ZooKeeper;
- MinIO или другие приложения, связанные с хранением данных;
- Apache Hadoop;
- Apache Cassandra;
- приложения для майнинга биткоинов.

Объекты StatefulSet (наборы модулей Pod с состоянием) тесно связаны с использованием DNS в Kubernetes, потому что оба обычно используются в сценариях, когда каноническая модель микросервисов начинает разрушаться, а внешние объекты (сервисы, приложения, устаревшие системы) начинают влиять на способ развертывания приложений. Теоретически вам редко придется использовать объекты StatefulSet для современных приложений без состояния, если только они не предъявляют критических требований к производительности, которые нельзя удовлетворить иным способом. StatefulSet сложнее администрировать, расширять и масштабировать, в отличие от «простых» объектов Deployment, не имеющих состояния, которое необходимо сохранять между перезапусками модулей Pod.

10.2.1 DNS и автономные сервисы

StatefulSet обычно используются для развертывания автономных сервисов. *Автономный сервис* – это сервис, не имеющий поля `clus-`

terIP и напрямую возвращающий запись A DNS-сервера. Это решение имеет некоторые важные последствия для DNS. Чтобы взглянуть на такой сервис, запустите следующий фрагмент кода:

```
$ kubectl create -f https://github.com/jayunit100/k8sprototypes/
  ➔ blob/master/smoke-tests/nginx-pod-svc.yaml
```

Эта команда вернет определение сервиса в формате YAML, например:

```
apiVersion: v1
kind: Service
metadata:
  name: headless-svc
spec:
  clusterIP: None
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  # Измените это значение на true, чтобы НИКОГДА не получать ответ NXDOMAIN!
  publishNotReadyAddresses: false
```

publishNotReadyAddresses решает, будете ли вы получать записи NXDomain или нет

←

Этот сервис выбирает из набора модулей Pod веб-сервер, который также определен в этом файле. После запуска сервиса:

- вы сможете выполнить запрос `wget headless-svc:80` в модуле BusyBox, который разворачивается вместе с сервисом;
- модуль BusyBox обратится к CoreDNS (о котором рассказывается в этой главе), чтобы получить IP-адрес автономного сервиса;
- CoreDNS проверит, работает ли автономный сервис (на основе `readinessProbe`), и вернет IP-адреса соответствующих модулей Pod.

ПРИМЕЧАНИЕ Если в `publishNotReadyAddresses` установить значение `true`, то всегда будут возвращаться внутренние модули Pod с веб-сервером NGINX, даже если они не готовы. Это означает, что если Pod с веб-сервером NGINX не готов, согласно его `readinessProbe`, то сервисы CoreDNS вернут записи NXDOMAIN вместо IP-адресов. Начиная осваивать Kubernetes часто неверно называют это ошибкой DNS, но на самом деле такое поведение указывает на потенциальные проблемы в kubelet или в приложении.

Когда используются автономные сервисы? Как оказывается, многие приложения создают кворумы и реализуют другое поведение, зависящее от сети, напрямую подключаясь друг к другу через IP-адрес, не полагаясь на kube-проxy, обеспечивающий балансировку нагрузки.

В общем случае старайтесь использовать сервисы с полем ClusterIP, когда это возможно, потому что с ними гораздо проще работать с точки зрения DNS, если только вам действительно не нужно какое-то особое поведение, связанное с сохранением IP, принятием решений кворумом или конкретными гарантиями в отношении IP-адресов.

Если вам интересно узнать больше о работе автономных сервисов и DNS, то загляните на страницу <http://mng.bz/q2Rz>.

10.2.2 Постоянные запуски DNS в StatefulSet

Давайте воссоздадим исходный пример StatefulSet. Для простоты выполните команду `kubectl create -f https://raw.githubusercontent.com/jayunit100/k8sprototypes/master/smoke-tests/four-of-us.yaml`. Имя этого сервиса можно использовать для просмотра его конечных точек, как показано далее:

```
$ kubectl get endpoints -o yaml | grep ip
- ip: 172.18.0.2
  ip: 10.244.0.13
- ip: 10.244.0.14
- ip: 10.244.0.15
  ip: 10.244.0.16
```


Здесь можно видеть четыре конечных точки в диапазоне IP-адресов 13–16. Это обусловлено наличием двух реплик StatefulSet и двух реплик Deployment.

10.2.3 Развертывание с несколькими пространствами имен для изучения свойств модуля DNS

В этом разделе мы рассмотрим два способа использования Kubernetes DNS. Затем сравним свойства DNS наших модулей StatefulSet и Deployment.

Для начала посмотрим, как работает DNS в этих модулях Pod. Самый очевидный тест – проверить конечные точки сервиса. Сделаем это внутри кластера, чтобы не пришлось беспокоиться об открытии или перенаправлении каких-либо портов. Прежде всего создадим Pod `bastion`, в котором мы сможем использовать утилиту `wget` для отправки запросов нашим приложениям:

```
$ cat << EOF > bastion.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
  namespace: default
spec:
  containers:
    - name: bastion
```



Пространство имен по умолчанию


```

image: docker.io/busybox:latest
command: ['sleep', '10000']
EOF
$ kubectl create -f bastion.yml

```

Обратите внимание, что в этом примере проще использовать пространство имен по умолчанию, но при желании вы можете создать Pod в другом пространстве имен. В таком случае вам придется использовать полные DNS-имена при обращении к нашим четырем сервисам. Теперь запустим этот Pod и используем его для экспериментов в оставшейся части этой главы:

```

$ kubectl get pods

```

NAME	READY	STATUS	AGE
core-k8s	1/1	Running	9m56s
web-dep-58db7f9644-fjtp6	1/1	Running	12h
web-dep-58db7f9644-gxddt	1/1	Running	12h
web-ss-0	1/1	Running	12h
web-ss-1	1/1	Running	12h

Этот Pod мы будем использовать для экспериментов с DNS внутри нашего кластера

```

$ kubectl exec -t -i core-k8s /bin/sh

```

Первое, что можно сделать, – обратиться к конечным точкам с помощью `wget`, например, так:

```

#> wget nginx4:80
Connecting to nginx4:80 (10.96.123.164:80)
saving to 'index.html'

```

Как все просто! Теперь мы знаем, что наш сервис работает. Далее, если внимательно посмотреть на IP-адрес, то можно заметить, что он находится вне диапазона 10.244. Причина в том, что мы обращаемся к сервису, а не к модулю Pod. Обычно для доступа к сервису внутри кластера используется DNS-имя сервиса, но что, если нам понадобится обратиться к определенному модулю Pod? Это можно сделать так:

```

#> wget nginx:80
Connecting to nginx:80 (10.96.123.164:80)
saving to 'index.html'

#> wget web-ss-0.nginx
Connecting to web-ss-0.nginx (10.244.0.13:80)

#> wget web-dep-58db7f9644-fjtp6
bad address 'web-dep-58db7f9644-fjtp6'

```

Комбинация из имен модуля Pod и сервиса

Возвращается IP-адрес модуля Pod в Deployment по его имени

Как видите, также можно использовать комбинацию из имен модуля Pod и сервиса, но для модулей Pod из Deployment нет эквивалентных DNS-имен.

Внутри контейнера можно получить доступ не только к модулям Pod через их сервисы, но также к некоторым из модулей, созданных в StatefulSet, непосредственно через DNS.

При выполнении запроса к конечной точке `web-ss-0.nginx` (и вообще к любой конечной точке `<name>-0.<serviceName>`) с помощью `wget`, ее имя напрямую преобразуется в IP-адрес первой реплики в данном `StatefulSet`. Чтобы получить доступ ко второй реплике, можно заменить 0 на 1 и т. д. По результатам первого эксперимента с DNS в кластере, можно сделать вывод: сервисы и модули Pod в `StatefulSet` являются типичными, стабильными конечными точками DNS в кластере Kubernetes. А теперь посмотрим, как разрешается чрезвычайно удобное имя `web-ss-0.nginx`?

10.3 Файл `resolv.conf`

Давайте посмотрим, как разрешаются (или не разрешаются, в некоторых случаях) эти различные DNS-запросы. И начнем мы с обзора файла `resolv.conf`, который в конечном итоге приведет нас к сервису CoreDNS.

10.3.1 Краткое примечание о маршрутизации

Эта глава посвящена не IP-сетям модулей Pod, но дает хороший шанс убедиться, что у вас имеется четкое представление о связи между DNS и сетевой инфраструктурой Pod – двух аспектах кластера, тесно связанных между собой. После разрешения имени хоста:

- если IP-адрес соответствует сервису, то сетевой прокси должен убедиться, что этот IP-адрес ведет к конечной точке модуля Pod;
- если IP-адрес соответствует модулю Pod, то провайдер CNI должен обеспечить прямую маршрутизацию IP-адреса;
- если хост находится в интернете, то исходящий трафик из модуля Pod должен пройти процедуру трансляции сетевых адресов (NAT) в `iptables`, чтобы TCP-соединение, установленное с внешним миром, возвращалось к вашему модулю Pod с узла, которому был отправлен запрос.

На рис. 10.2 показано, как работает DNS для входящего имени хоста. Ключевая особенность состоит в том, что на адрес 10.96.0.10 будет отправлено несколько версий DNS-запроса, пока обнаружится совпадение.

Файл `resolv.conf` – это стандартный способ настройки DNS для контейнера. В любой ситуации, когда вы пытаетесь выяснить настройки DNS в вашем модуле Pod, это первое место, куда следует заглянуть. Если вы используете современный сервер Linux, то можете использовать `resolvectl`, но суть та же. Теперь посмотрим на настройки DNS в нашем модуле Pod, выполнив следующую команду:

```
/ # cat /etc/resolv.conf  
search
```

Следующие строки
добавляются в конец запроса

```

default.svc.cluster.local
svc.cluster.local
cluster.local
nameserver 10.96.0.10 ← Адрес DNS-сервера
options ndots:5

```

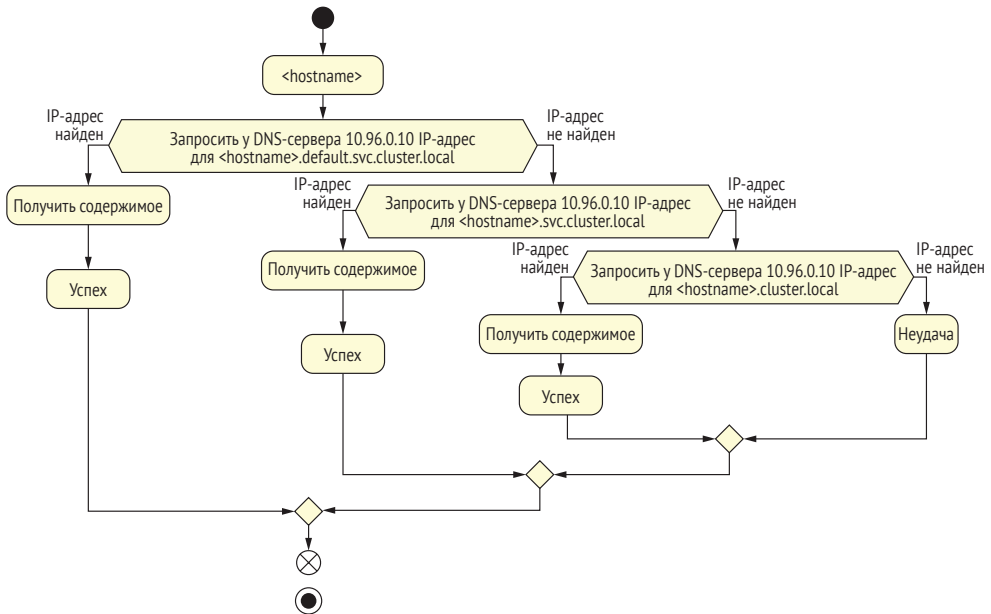


Рис. 10.2 Разрешение входящего имени хоста в DNS

В этом фрагменте поле `search` говорит, что «нужно добавлять эти атрибуты в конец запроса, пока он не увенчается успехом». Другими словами, сначала проверяется, можно ли разрешить URL без каких-либо изменений. В случае неудачи предпринимается попытка добавить `default.svc.cluster.local`. Если это не помогло, выполняется попытка добавить `svc.cluster.local` и т. д. Обратите также внимание на поле `nameserver`. Оно сообщает вашему DNS-серверу, что тот может исправлять внешние DNS-имена (которых нет в `/etc/hosts`), запрашивая DNS-сервер по адресу `10.96.0.10` – ваш сервис `kube-dns`.

Посмотрим, например, как DNS разрешает имена модулей Pod из `StatefulSet` внутри кластерной сети, запустив `wget`. Запустим `kubectl exec` внутри модуля `NGINX` и затем выполним следующую команду:

```

/ # wget web-ss-0.nginx.default.svc.cluster.local
Connecting to web-ss-0.nginx.default.svc.cluster.local
(10.244.0.13:80)

```

Мы оставим читателям в качестве упражнения попробовать проделать то же самое при использовании другого пространства имен, чтобы убедиться, что имя `web-ss-0` правильно разрешается из любого


```

}
kind: ConfigMap

```

Первое, что пытается сделать этот пример, – разрешить имена локальных узлов в кластере, добавляя плагин Kubernetes для CoreDNS. Затем используется *resolv.conf* в kubelet для разрешения имен в интернете, если плагин Kubernetes потерпит неудачу.

Возможно, вам интересно: не будет ли *resolv.conf* зависеть от CoreDNS, если CoreDNS работает в контейнере? Ответа на этот вопрос нет! Чтобы понять причину, посмотрим на поле *dnsPolicy* кластера, которое имеется в любом модуле Pod в кластере Kubernetes:

```

> kubectl get pod coredns-66bff467f8-cr9kh
  -o yaml | grep dnsPolicy
  dnsPolicy: ClusterFirst

```

Использует CoreDNS в качестве основного сервера имен

```

> kubectl get pods -o yaml | grep dnsPolicy
  dnsPolicy: Default

```

Запускает модули Pod со значением по умолчанию в поле *dnsPolicy*

Политика *ClusterFirst* использует CoreDNS в качестве основного сервера имен, поэтому файл *resolv.conf* в нашем модуле Pod содержит только CoreDNS. Модули Pod, запущенные со значением *Default* в *dnsPolicy*, фактически получают внедренный в них файл */etc/resolv.conf*, получающий записи из kubelet. Таким образом, в большинстве кластеров Kubernetes вы обнаружите, что:

- несмотря на то что CoreDNS работает в обычной сети модулей Pod, ему назначается другая политика DNS, отличная от политики для других «обычных» Pod в кластере;
- модули Pod в кластере сначала пытаются связаться с внутренним сервисом Kubernetes, прежде чем выходить в интернет через поток, настроенный в Corefile;
- CoreDNS в контейнере перенаправляет внекластерные внутренние IP-адреса туда же, куда они перенаправляются его хостом. Другими словами, он наследует настройки разрешения имен в интернете от kubelet.

10.3.3 Разбор конфигурации плагина CoreDNS

Плагин кеша сообщает сервису CoreDNS, что может хранить результаты в кеше в течение 30 с. Это означает, что если:

- уменьшить масштаб StatefulSet (командой `kubectl scale statefulset web-ss --replicas=0`);
- запустить `wget` для подключения к модулю Pod `web-ss-0.nginx`;
- масштабировать резервную копию StatefulSet (командой `kubectl scale statefulset web-ss --replicas=3`),

то команда `wget` может надолго зависнуть, даже притом, что почти сразу будет запущено три реплики веб-сервера. Причина в том, что по умолчанию CoreDNS, который должен запускать свой плагин кеша

с 30-секундной емкостью, в течение нескольких секунд будет терпеть неудачу, отправляя DNS-запросы к `web-ss-0.nginx` даже после успешного запуска этого модуля.

Чтобы исправить проблему, можно выполнить команду `kubectl edit cm coreDNS -n kube-system` и изменить значение продолжительности кеширования на меньшее число, например 5. Это гарантирует быстрое обновление результатов DNS-запросов в кеше. Чем больше это число, тем меньшую нагрузку будет испытывать базовая плоскость управления Kubernetes, но, как вы понимаете, в нашем небольшом кластере эти накладные расходы не важны.

Обратите внимание, что настройка DNS – серьезная тема в любом центре обработки данных и не зависит от применения или не-применения Kubernetes. Для дальнейшей настройки DNS в больших кластерах можно запустить kubelet с политиками NodeLocalDNS (для этого нужна одна из последних версий Kubernetes). Эта политика существенно ускоряет работу DNS, запуская DaemonSet на всех узлах в кластере, который кеширует все DNS-запросы для всех модулей Pod. Также можно исследовать множество других настроек плагина CoreDNS и организовать мониторинг метрик Prometheus.

Итоги

- Kubernetes предоставляет модулям внутренний механизм разрешения имен для доступа к сервисам Service.
- Объекты StatefulSet чрезвычайно важны для приложений с четко установленной идентичностью.
- Автономные сервисы напрямую возвращают IP-адреса модулей Pod и не имеют постоянного значения ClusterIP, т. е. иногда они могут возвращать NXDOMAIN, если модуль Pod не работает.
- сервисы и модули Pod в StatefulSet являются типичными, стабильными конечными точками DNS в кластере Kubernetes.
- Файл `resolv.conf` – это стандартный способ настройки DNS для контейнера. В любой ситуации, когда вы пытаетесь выяснить настройки DNS в вашем модуле Pod, это первое место, куда следует заглянуть.
- CoreDNS поддерживается плагинами. Читать конфигурацию CoreDNS следует сверху вниз, причем каждый следующий плагин определяется в новой строке.
- Плагин кеша сообщает сервису CoreDNS, что может хранить результаты в кеше в течение 30 с.

11

Плоскость управления

В этой главе:

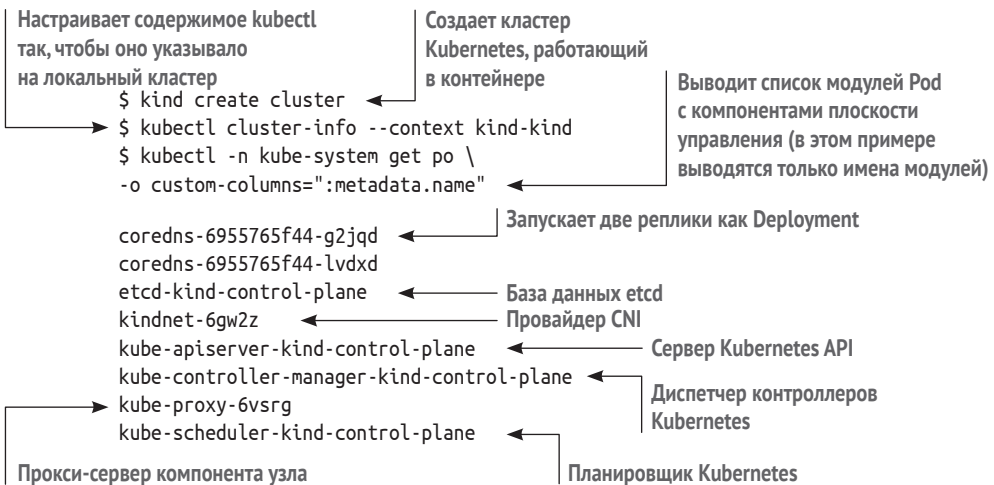
- основные компоненты плоскости управления;
- обзор особенностей сервера API;
- исследование интерфейсов планировщика и его работы;
- знакомство с диспетчерами контроллеров и облачных вычислений.

Выше в этой книге мы предоставили общий обзор модулей Pod и рассказали, зачем нужны модули, а также как строятся кластеры Kubernetes с их помощью. Теперь давайте углубимся в детали плоскости управления. Как правило, все компоненты плоскости управления устанавливаются в пространстве имен `kube-system`, причем в это пространство имен устанавливается минимальное количество компонентов.

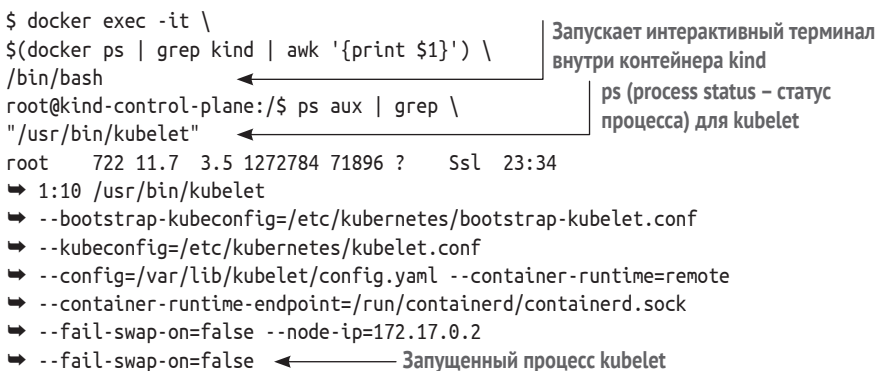
ПРИМЕЧАНИЕ Вы просто не должны использовать `kube-system` для прикладных нужд! Одна из основных причин состоит в том, что приложения, не являющиеся контроллерами и работающие внутри `kube-system`, создают дыру в системе безопасности, увеличивая риск вторжения. Кроме того, работая в такой системе, как GKE или EKS, нет возможности видеть все компоненты плоскости управления. Подробнее о методах обеспечения безопасности мы поговорим в главе 13.

11.1 Плоскость управления

Один из самых простых способов запустить и настроить плоскость управления – использовать `kind`, кластер Kubernetes в контейнере (инструкции по установке вы найдете по ссылке: <http://mng.bz/1alM>). Чтобы использовать `kind` для исследования плоскости управления, выполните следующие команды:



Обратите внимание, что `kubelet` работает не в модуле Pod. Некоторые системы запускают `kubelet` внутри контейнера, но в таких системах, как `kind`, агент `kubelet` запускается как обычная программа. Чтобы увидеть, как `kubelet` работает в кластере `kind`, выполните следующие команды:



Выполните `exit`, чтобы выйти из интерактивного терминала в контейнере. Чтобы по-настоящему понять, что такое плоскость управления, попробуйте получить информацию о разных модулях Pod. Например, вот как можно получить сведения о модуле сервера API:

```
$ kubectl -n kube-system get po kube-apiserver-kind-control-plane -o yaml
```


11.2 Особенности сервера API

Теперь пришло время углубиться в особенности сервера API, потому что это не только веб-сервер, но и критически важный компонент плоскости управления. Следует отметить, что сервер API обслуживает не только объекты плоскости управления, но и пользовательские объекты. Далее в этой книге мы рассмотрим контроллеры аутентификации, авторизации и допуска, но сначала более подробно рассмотрим объекты Kubernetes API и пользовательские ресурсы.

11.2.1 Объекты API и пользовательские ресурсы

Kubernetes – открытая платформа, т. е. открытый API. Открытость платформы обеспечивает появление инноваций и открывает путь для творчества. Ниже перечислены некоторые ресурсы API, связанные с кластером Kubernetes. В них вы увидите часть этих объектов API (например, Deployment и Pod):

```
$ kubectl api-resources -o name | head -n 20
bindings
componentstatuses
configmaps
endpoints
events
limitranges
namespaces
nodes
persistentvolumeclaims
persistentvolumes
pods
podtemplates
replicationcontrollers
resourcequotas
secrets
serviceaccounts
services
mutatingwebhookconfigurations.admissionregistration.k8s.io
validatingwebhookconfigurations.admissionregistration.k8s.io
customresourcedefinitions.apiextensions.k8s.io
```

Выводит первые 20 доступных API с помощью head

Когда определяется манифест YAML с ClusterRoleBinding, частью определения является версия API. Например:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: cockroach-operator-default
  labels:
    app: cockroach-operator
roleRef:
```

apiVersion соответствует группе API в предыдущем фрагменте

```
apiGroup: rbac.authorization.k8s.io
kind: ClusterRole
name: cockroach-operator-role
subjects:
- name: cockroach-operator-default
  namespace: default
kind: ServiceAccount
```

Раздел `apiVersion` в предыдущем фрагменте YAML определяет версию API. Версионирование API – сложная задача. Чтобы дать возможность перемещаться между разными версиями, Kubernetes поддерживает версии и уровни. Например, в предыдущем определении можно заметить, что значение `apiVersion` содержит `v1beta1`. Это означает, что `ClusterRoleBinding` является объектом *бета*-версии API.

Объекты API имеют следующие уровни: `alpha`, `beta` и `GA` (*general availability* – доступность для всех). Объекты с уровнем `alpha` никогда не должны использоваться в промышленном окружении, потому что могут вызвать серьезные проблемы с обновлением. Объекты API с уровнем `alpha` обязательно будут изменяться и предназначены только для разработки и экспериментов. Уровень `beta` в действительности не обозначает бета-версию! Бета-версии программного обеспечения часто считаются нестабильными и не предназначены для промышленного использования, но объекты Kubernetes API с уровнем `beta` готовы к эксплуатации, и их поддержка гарантирована в отличие от объектов на уровне `alpha`. Например, наборы `DaemonSet` находились на уровне `beta` в течение многих лет, и практически все использовали их в промышленных окружениях.

Префикс `v1` позволяет разработчикам Kubernetes нумеровать версии объектов API. Например, в Kubernetes `v1.17.0` API автоматически масштабирования включает:

- `/apis/autoscaling/v1`;
- `/apis/autoscaling/v2beta1`;
- `/apis/autoscaling/v2beta2`.

Обратите внимание, что элементы этого списка имеют вид URI. Вы можете просматривать объекты API в формате URI, предварительно запустив кластер `kind` локально:

```
$ kind cluster start
```

После этого выполните команду `kubect1` в системе, где имеется веб-браузер. Например:

```
$ kubect1 proxy --port=8181
```

Затем откройте страницу `http://127.0.0.1:8181/`. Для краткости мы будем показывать 120-строчный ответ сервера API, но если вы сделаете это у себя, то получите графическое представление конечных точек API.

11.2.2 Определения пользовательских ресурсов (CRD)

В примере с ClusterRoleBinding мы представили CRD (Custom Resource Definition – определение пользовательского ресурса) для связи с базой данных cockroach. Теперь самое время обсудить, зачем нужны CRD. В Kubernetes v1.17.0 имеется 54 объекта API. Следующая команда позволит получить некоторое представление о них:

```
$ kubectl api-resources | wc  
54 230 5658
```

Результаты, возвращаемые командой `kubectl`, передаются через конвейер утилите `wc`, которая подсчитывает количество строк, слов и символов

Нетрудно понять, сколько требуется времени на разработку для поддержки системы, содержащей 54 различных объекта (честно говоря, нам нужно больше). Чтобы отделить второстепенные объекты API от сервера API, были придуманы пользовательские определения ресурсов CRD. Они позволяют разработчикам определять свои объекты API, а затем с помощью `kubectl` внедрять их на сервер API. Следующая команда создает объект CRD на сервере API:

```
$ kubectl apply -f https://raw.githubusercontent.com/cockroachdb/  
➤ cockroach-operator/v2.4.0/config/crd/bases/  
➤ crdb.cockroachlabs.com_crdbclusters.yaml
```

По аналогии с Pod и другими стандартными объектами API объекты CRD расширяют платформу Kubernetes API без участия программиста. Операторы, пользовательские контроллеры допуска, Istio, Envoy и другие технологии теперь используют сервер API, определяя свои CRD. Но эти пользовательские объекты слабо связаны с реализацией объектов Kubernetes API. Более того, многие новые компоненты Kubernetes добавляются не как стандартные определения, а как CRD. Это и есть сервер API. Далее мы обсудим первый контроллер: планировщик Kubernetes.

11.2.3 Планировщик

Планировщик, подобно другим контроллерам, реализует несколько циклов управления, обрабатывающих разные события. Начиная с версии Kubernetes 1.15.0, планировщик был реорганизован для использования фреймворка планирования и поддержки пользовательских плагинов. Kubernetes позволяет использовать пользовательские планировщики, которые запускаются не в реальном планировщике, а в отдельном модуле Pod. Однако пользовательские планировщики часто страдают проблемой низкой производительности.

Первый компонент фреймворка планировщика – QueueSort. Он сортирует модули Pod, требующие планирования, и ставит их в очередь. Затем фреймворк разбивается на два цикла: цикл планирования и цикл связывания. Сначала цикл планирования выбирает узлы, до-

ступные для запуска модулей Pod. После завершения цикла планирования в работу включается цикл связывания. Он выбирает конкретный узел и проверяет, можно ли разместить Pod на нем. Это может занять некоторое время. Например, модулю Pod нужен том хранения, а значит, этот том необходимо создать. Что произойдет, если создать требуемый том не удастся? В таком случае Pod нельзя будет запустить на этом узле, и он ставится в очередь повторно.

Мы рассмотрим этот процесс, чтобы лучше понять, например, когда планировщик обрабатывает Pod NodeAffinity. Каждый из циклов имеет отдельные компоненты, представленные в следующей структуре в Scheduler API. Код взят из версии Kubernetes v1.22, а начиная с версии 1.23, он подвергся реорганизации с целью разрешить подключение плагинов к нескольким точкам. На момент написания этой книги основа самого планировщика и плагинов не изменилась. Этот код (доступный по адресу <http://mng.bz/d2oX>) определяет различные наборы плагинов, зарегистрированных в действующем экземпляре планировщика. Вот базовое определение API:

```
// Плагины имеют несколько точек расширения. Если точка расширения
// определена в конфигурации, к ней подключается заданный список плагинов.
// Если не определена, то к ней подключается набор плагинов по умолчанию.
// Подключенные плагины вызываются в порядке, указанном здесь, после
// плагинов по умолчанию. Если они должны вызываться перед плагинами
// по умолчанию, то последние должны быть отключены и повторно подключены
// здесь в нужном порядке.
type Plugins struct {
    // QueueSort -- список плагинов, которые должны
    // вызываться при помещении модулей pod в очередь.
    QueueSort *PluginSet Sorts the Pods in a Queue
    // PreFilter -- это список плагинов, вызываемых точкой расширения
    // PreFilter фреймворка планирования.
    PreFilter *PluginSet
    // Filter -- это список плагинов, вызываемых во время фильтрации
    // узлов, на которых не может быть запущен Pod.
    Filter *PluginSet
    // PostFilter -- это список плагинов, вызываемых после этапа
    // фильтрации, независимо от его успеха.
    PostFilter *PluginSet
    // PreScore -- это список плагинов, вызываемых перед ранжированием.
    PreScore *PluginSet
    // Score -- это список плагинов, вызываемых при ранжировании узлов,
    // прошедших через этап фильтрации.
    Score *PluginSet
    // Reserve -- это список плагинов, вызываемых при
    // резервировании ресурсов после выбора узла для запуска Pod.
    Reserve *PluginSet
}
```

Помещает модули Pod в очередь

Здесь начинаются плагины цикла планирования и заканчиваются плагином Permit

```
// Permit -- это список плагинов, управляющих привязкой Pod.
// Они могут предотвратить или задержать привязку Pod.
Permit *PluginSet

// PreBind -- это список плагинов, вызываемых перед привязкой Pod.
PreBind *PluginSet

// Bind -- это список плагинов, вызываемых в точке расширения
// Bind фреймворка планирования. Планировщик вызывает эти плагины
// по порядку и прекращает их выполнение, как только первый
// из них вернет признак успеха.
Bind *PluginSet

// PostBind -- это список плагинов, вызываемых после успешной привязки Pod.
PostBind *PluginSet
}
```

Эти три последних плагина
вызываются в цикле привязки

Структура в предыдущем фрагменте создается в <http://mng.bz/rJaZ> (после выпуска версии 1.21 этот код был реорганизован и перемещен). В следующем фрагменте вы увидите плагины планирования, которые обрабатывают такие конфигурации, как Pod NodeAffinity, влияющие на планирование модулей Pod. На первом этапе этого процесса выполняются плагины из списка QueueSort, но обратите внимание, что QueueSort можно расширить и даже заменить:

```
func getDefaultConfig() *schedulerapi.Plugins { ← Вызывается из getDefaultConfig()
    return &schedulerapi.Plugins{
        QueueSort: &schedulerapi.PluginSet{ ← Вызывается из getDefaultConfig()
            Enabled: []schedulerapi.Plugin{
                {Name: queuesort.Name},
            },
        },
    },
}
```

Приватная функция getDefaultConfig() вызывается функцией NewRegistry, которая определена в том же файле Go. Она возвращает экземпляр реестра с провайдерами алгоритмов. Следующие возвращаемые элементы определяют цикл планирования. Первый из них, Prefilter, — это список плагинов, выполняемых последовательно:

```
Prefilter: &schedulerapi.PluginSet {
    Enabled: []schedulerapi.Plugin {
        {Name: noderesources.FitName}, ← Проверяет, достаточно ли ресурсов на узле
        {Name: nodeports.Name}, ← Проверяет наличие на узле свободных портов для размещения Pod
        {Name: podtopologyspread.Name}, ← Проверяет соответствие PodTopologySpread, что позволяет равномерно распределять модули Pod по зонам
        {Name: interpodaffinity.Name}, ← Обработывает совместимость модулей Pod. Если на узле выполняется несовместимый модуль (согласно правилам, определенным пользователем), то планируемый модуль «отталкивается» от этого узла
    }
}
```

```

    {Name: volumebinding.Name},
  },
},

```

На самом деле это не фильтр, этот плагин создает кеш, используемый позже на этапах резервирования и предварительной привязки

Следующий этап – фильтрация. Обратите внимание, что *Filter* – это список плагинов, определяющих возможность запуска Pod на определенном узле:

```

Filter: &schedulerapi.PluginSet {
  Enabled: []schedulerapi.Plugin {
    {Name: nodeunschedulable.Name},
    {Name: noderesources.FitName},
    {Name: nodename.Name},
    {Name: nodeports.Name},
    {Name: nodeaffinity.Name},
    {Name: volumerestrictions.Name},
    {Name: tainttoleration.Name},
    {Name: nodevolumelimits.EBSName},
    {Name: nodevolumelimits.GCEPDName},
    {Name: nodevolumelimits.CSIName},
    {Name: nodevolumelimits.AzureDiskName},
    {Name: volumebinding.Name},
    {Name: volumezone.Name},
    {Name: podtopologyspread.Name},
    {Name: interpodaffinity.Name},
  },
},

```

Плагин запускается повторно →

Этот плагин тоже запускается повторно →

Проверяет соблюдение различных ограничений, связанных с томом хранилища →

Повторно запускается плагин фильтра из PreFilter →

Гарантирует невозможность запланировать Pod на узле, отмеченном как непредназначенный для планирования (например, на узле в плоскости управления)

PodSpec API позволяет установить поле nodeName, идентифицирующее узел, на котором должен размещаться Pod

Проверяет, соответствует ли селектор узла в определении Pod метке в определении узла

Проверяет совместимость Pod с данным узлом

Проверяет возможность добавления тома хранилища на узел (например, в GCP узел может подключить только до 16 томов)

Проверяет наличие тома в зоне, в которой находится узел

Эти два фильтра выполняются повторно

На этапе фильтрации планировщик проверяет различные ограничения на подключение томов, имеющиеся в GCP, AWS, Azure, ISCI и RBD. Например, несовместимость модулей Pod гарантирует, что модули Pod из StatefulSet будут размещаться на разных узлах. Возможно, вы уже заметили, что фильтры планируют модули Pod на основе настроек, которые вы уже определили. Теперь перейдем к PostFilter. Плагины из этого списка выполняются, даже если фильтрация потерпела неудачу:

```

PostFilter: &schedulerapi.PluginSet{
  Enabled: []schedulerapi.Plugin{
    {Name: defaultpreemption.Name},
  },
},

```

Выполняет вытеснение модулей Pod

Пользователь может назначить модулю Pod класс приоритета. В таком случае плагин `defaultpreemption` позволяет планировщику определить, можно ли вытеснить другой модуль Pod, чтобы освободить место для более приоритетного планируемого модуля Pod. Обратите внимание, что эти плагины повторно выполняют всю фильтрацию, чтобы определить, сможет ли Pod выполняться на определенном узле.

Далее выполняется ранжирование. Планировщик составляет список узлов, на которых можно разместить Pod, и теперь он должен их упорядочить путем оценки, чтобы выбрать наиболее подходящий. Поскольку компонент ранжирования является частью этапа фильтрации, вы заметите в нем много повторяющихся плагинов. Планировщик сначала вычисляет предварительные оценки, чтобы создать общий список для плагинов:

```
PreScore: &schedulerapi.PluginSet{
  Enabled: []schedulerapi.Plugin{
    {Name: interpodaffinity.Name},
    {Name: podtopologyspread.Name},
    {Name: tainttoleration.Name},
  },
},
```

← Все плагины в этом списке уже выполнялись на этапе фильтрации

В следующем фрагменте повторно используются различные плагины и несколько новых. Планировщик определяет вес, влияющий на планирование. Все узлы, получившие оценку, прошли различные этапы фильтрации:

```
Score: &schedulerapi.PluginSet{
  Enabled: []schedulerapi.Plugin{
    {Name: noderesources.BalancedAllocationName,
      Weight: 1},
    {Name: imagelocality.Name, Weight: 1},
    {Name: interpodaffinity.Name, Weight: 1},
    {Name: noderesources.LeastAllocatedName,
      Weight: 1},
    {Name: nodeaffinity.Name, Weight: 1},
    {Name: nodepreferavoidpods.Name,
      Weight: 10000},
    {Name: podtopologyspread.Name, Weight: 2},
    {Name: tainttoleration.Name, Weight: 1},
  },
},
```

← Приоритезация узлов со сбалансированным использованием ресурсов

← Узлы, на которые уже загружен образ Pod, оцениваются выше

← Повторно выполняется плагин для оценки построенного кеша

← Еще раз выполняется плагин для оценки построенного кеша

← Снижает оценку узла, если установлен параметр `preferenceAvoidPods`

← Эти два плагина выполняются повторно

Предпочтение отдается узлам с меньшим количеством запросов

// Вес увеличивается на два, потому что:
 // - эта оценка обусловлена предпочтениями пользователя.
 // - делает этот сигнал сопоставимым с `NodeResourcesLeastAllocated`

При приоритизации узлов со сбалансированным использованием ресурсов планировщик учитывает количество доступных процессов, памяти и томов. Вот как выглядит используемый алгоритм:

$$(\text{cpu}((\text{capacity} \cdot \text{sum}(\text{requested})) * \text{MaxNodeScore}/\text{capacity}) + \text{memory}((\text{capacity} \cdot \text{sum}(\text{requested})) * \text{MaxNodeScore}/\text{capacity})) / \text{weightSum}$$

Этот алгоритм оценивает выше узлы с меньшим количеством запросов. Метка узла, `preferenceAvoidPods`, говорит о том, что этот узел должен исключаться из планирования.

Последний шаг в процессе фильтрации – этап резервирования. На этом этапе резервируется том для модуля Pod, который будет использоваться в цикле привязки. Обратите внимание, что плагин `volume-binding` в следующем фрагменте выполняется повторно:

```
Reserve: &schedulerapi.PluginSet{
  Enabled: []schedulerapi.Plugin{
    {Name: volumebinding.Name}, ← Кеш резервирует том для Pod
  },
},
```

Цикл планирования, выполняя в основном фильтрацию, определяет подходящий узел. Но подготовка всех ресурсов на узле, необходимых модулю Pod, – это гораздо более длительный процесс, в течение которого Pod пребывает в очереди планирования. Давайте теперь посмотрим на цикл привязки, начав с этапа предварительной привязки. В следующем фрагменте показан код плагина `PreBind`:

```
PreBind: &schedulerapi.PluginSet{
  Enabled: []schedulerapi.Plugin{
    {Name: volumebinding.Name}, ← Привязывает том к модулю Pod
  },
},

Bind: &schedulerapi.PluginSet{
  Enabled: []schedulerapi.Plugin{
    {Name: defaultbinder.Name}, ← Сохраняет объект Bind, обращаясь к серверу
                                   API, и обновляет информацию об узле,
                                   где должен запуститься Pod
  },
},
```

У планировщика есть несколько очередей: активная очередь, куда помещаются модули Pod, планируемые для запуска, и очередь простаивающих модулей Pod, не предназначенных для планирования. Реестр в планировщике не создает экземпляры плагинов для двух разных этапов: `Permit` и `PostBind`. Эти точки расширения используются другими плагинами, например пакетным планировщиком, который вскоре станет внешним плагином. Поскольку теперь у нас есть фреймворк планирования, мы можем регистрировать и использовать другие плагины. Примеры таких пользовательских плагинов можно найти в репозитории GitHub по адресу <http://mng.bz/oaBN>.

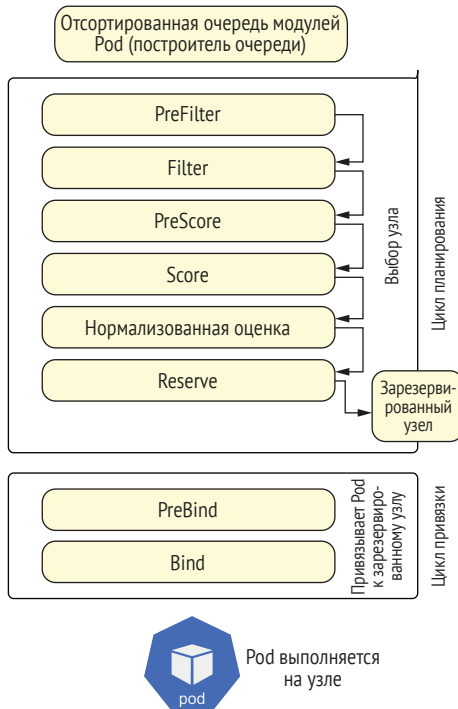


Рис. 11.1 Планировщик Kubernetes

11.2.4 Краткий обзор фреймворка планирования

На рис. 11.1 показаны три компонента, составляющих фреймворк планирования:

- *построитель очереди* – поддерживает очередь модулей Pod;
- *цикл планирования* – фильтрует и отыскивает узлы для запуска модулей Pod;
- *цикл привязки* – сохраняет данные и информацию о привязке на сервере API.

11.3 Диспетчер контроллеров

Значительная часть функциональности диспетчера контроллеров (Kubernetes Controller Manager, KCM) была перемещена в облачный диспетчер контроллеров (Cloud Controller Manager, CCM). Этот двоичный файл включает четыре компонента, которые сами являются контроллерами или просто циклами управления. Мы рассмотрим их в следующих разделах.

11.3.1 Хранилище

Поддержка хранилищ в Kubernetes – это своего рода движущаяся цель. Одновременно с перемещением функциональности из КСМ в ССМ происходят серьезные изменения в работе хранилищ внутри плоскости управления Kubernetes. До перемещения адаптеры хранилища КСМ находились в основном репозитории `kubernetes/kubernetes`. Пользователь создавал PVC (`PersistentVolumeClaim`) в облаке, а КСМ вызвал код, находящийся внутри проекта Kubernetes. Затем появились гибкие контроллеры томов, которые существуют и по сей день. КСМ управляет созданием объектов хранилищ, начиная с Kubernetes v1.18.x.

Когда пользователь создает PV или PVC или комбинацию PVC/PV, необходимые для создания `StatefulSet`, компонент плоскости управления должен инициировать и управлять созданием тома хранилища. Этот том может размещаться в облаке или в другой виртуальной среде. Но важно помнить, что создание и удаление хранилища контролирует КСМ. Давайте пройдемся по контроллерам, составляющим КСМ.

Контроллер узла наблюдает за работоспособностью узла и своевременно обновляет его статус в объекте `Nodes API`.

Контроллер репликации поддерживает заданное количество модулей `Pod` для каждого объекта контроллера репликации в системе. Объекты контроллера репликации по большей части были заменены разворачиваниями, использующими наборы реплик `ReplicaSet`.

Контроллер конечной точки – это последний контроллер, управляющий объектами `Endpoint`, которые определяются в `Kubernetes API`. Эти объекты обычно обслуживаются автоматически и создаются для передачи прокси-серверу `kube-proxy` информации, необходимой для подключения модуля `Pod` к сервису. Сервис `Service` может иметь один или несколько модулей `Pod`, обрабатывающих трафик от указанного сервиса. Вот пример конечных точек, созданных для `kube-dns` в кластере `kind`:

```
$ kubectl -n kube-system describe endpoints kube-dns
Name:         kube-dns
Namespace:    kube-system
Labels:       k8s-app=kube-dns
              kubernetes.io/cluster-service=true
              kubernetes.io/name=KubeDNS
Annotations:  endpoints.kubernetes.io/last-change-trigger-time:
➔ 2020-09-30T00:21:28Z
Subsets:
  Addresses:          10.244.0.2,10.244.0.4
  NotReadyAddresses:  <none>
  Ports:
    Name  Port  Protocol
    ----  -
    dns    53   UDP
    dns-tcp 53   TCP
    metrics 9153  TCP
```

IP-адреса модулей Pod, составляющих сервис kube-dns

11.3.2 Учетные данные сервисов и токены

Когда создается новое пространство имен Namespace, диспетчер контроллеров Kubernetes создает для него учетную запись сервиса ServiceAccount по умолчанию и токены доступа к API. Если в определении модуля Pod не указана конкретная учетная запись для сервиса, то будет использоваться учетная запись ServiceAccount по умолчанию, созданная в пространстве имен Namespace. Неудивительно, что ServiceAccount используется, когда Pod обращается к серверу API кластера. В момент запуска модуля Pod передается токен доступа к API, если только пользователь не запретит это.

СОВЕТ Если модулю Pod не нужен токен ServiceAccount, запретить его передачу можно, указав в automountServiceAccountToken значение false.

11.4 Облачные диспетчеры контроллеров Kubernetes (CCM)

Представьте, что нас есть кластер Kubernetes, работающий в облаке или использующий решение виртуализации. В любом случае эти разные платформы хостинга поддерживают набор облачных контроллеров, взаимодействующих со слоем API, где размещен Kubernetes. Если у вас появится желание написать новый облачный контроллер, то вам потребуется включить функции для следующих компонентов:

- *узлов* – для обслуживания виртуальных экземпляров;
- *маршрутизации* – для обслуживания трафика между узлами;
- *внешних балансировщиков нагрузки* – для создания балансировщика нагрузки, внешнего по отношению к узлам в кластере.

Код взаимодействия с этими компонентами внутри облачного провайдера зависит от API провайдера. Интерфейс облачного контроллера теперь определяется общим интерфейсом для разных облачных провайдеров. Например, для создания облачного провайдера для Kubernetes нужно создать контроллер, реализующий следующий интерфейс:

```
// Абстрактный интерфейс для подключения интерфейсов облачных провайдеров.
type Interface interface {

    // Initialize предоставляет облаку строителя клиентов Kubernetes и
    // может создавать сопрограммы для обслуживания или запуска
    // пользовательских контроллеров, характерных для облачного провайдера.
    // Любые задачи, запущенные здесь, должны останавливаться с
    // закрытием канала stop.
    Initialize(clientBuilder ControllerClientBuilder, stop <-chan struct{})
```

```

// LoadBalancer возвращает интерфейс балансировщика, а также true, если
// интерфейс поддерживается; иначе возвращает false.
LoadBalancer() (LoadBalancer, bool)

// Instances возвращает интерфейс экземпляров, а также true, если
// интерфейс поддерживается; иначе возвращает false.
Instances() (Instances, bool)

// InstancesV2 -- это реализация для экземпляров. Должна реализовываться
// только внешними облачными провайдерами. По своему поведению
// InstancesV2 идентична Instances, но оптимизирована для сокращения
// вызовов API облачного провайдера при регистрации и синхронизации узлов.
// Возвращает true, если интерфейс поддерживается, и false в противном случае.
// ВНИМАНИЕ: InstancesV2 -- это экспериментальный интерфейс, который
// может измениться в версии 1.20.
InstancesV2() (InstancesV2, bool)

// Zones возвращает интерфейс зон, а также true, если
// интерфейс поддерживается; иначе возвращает false.
Zones() (Zones, bool)

// Clusters возвращает интерфейс кластеров, а также true, если
// интерфейс поддерживается; иначе возвращает false.
Clusters() (Clusters, bool)

// Routes возвращает интерфейс маршрутов, а также true, если
// поддерживается.
Routes() (Routes, bool)

// ProviderName возвращает идентификатор облачного провайдера.
ProviderName() string

// HasClusterID возвращает true, если ClusterID используется и установлен.
HasClusterID() bool
}

```

При проектировании ССМ рекомендуется реализовать три цикла управления (контроллера), которые обычно развертываются как один двоичный файл.

Чтобы подключить облачный том к узлу, нужно найти узел в облаке. Эту задачу решает контроллер узлов. Он должен знать, какие узлы имеются в кластере, и ему нужна не только информация, предоставляемая агентом kubelet при запуске узла. При работе в облачной среде фреймворку Kubernetes требуется определенная информация об узле и как он развернут в облачной среде (например, о зоне). Кроме того, есть слой, который определяет, был ли узел удален из облачной среды. Контроллер узла обеспечивает мост с уровнем облачного API и сохраняет эту информацию на сервере API.

Kubernetes должен направлять трафик между узлами, и этим занимается контроллер маршрутов. Если облаку потребуются информация для маршрутизации данных между узлами, то ССМ выполнит вызовы API для сбора информации обо всем сетевом трафике между узлами.

Название *контроллер сервисов* немного неправильное. Контроллер сервисов просто облегчает создание сервисов типа LoadBalancer в кластере, но никак не связан с сервисами ClusterIP в кластере Kubernetes.

11.5 Дополнительная литература

Acetozi. «Kubernetes Master Components: Etcd, API Server, Controller Manager, and Scheduler». <http://mng.bz/doKX> (доступно по состоянию на 29.12.2021).

Итоги

- Плоскость управления Kubernetes предоставляет функциональные возможности для организации и размещения модулей Pod в кластере Kubernetes.
- Планировщик реализует несколько циклов управления, обрабатывающих разные события.
- Цикл планирования, выполняя в основном фильтрацию, определяет подходящий узел.
- Объекты Kubernetes API на уровне beta готовы к использованию в промышленном окружении. Поддержка этих объектов гарантирована в отличие от объектов на уровне alpha.
- КСМ и ССМ вместе предоставляют ресурсы хранения, сервисы, балансировщики нагрузки и другие компоненты с помощью различных контроллеров, входящих в состав КСМ и ССМ.

12

etcd и плоскость управления

В этой главе:

- сравнение etcd v2 и v3;
- хронометраж в Kubernetes;
- важность поддержки строгой согласованности;
- балансировка нагрузки на узлы с etcd;
- модель безопасности etcd в Kubernetes.

Как обсуждалось в главе 11, etcd – это хранилище ключей/значений с надежными гарантиями согласованности. Оно похоже на хранилище ZooKeeper, которое используется в таких популярных технологиях, как HBase и Kafka. Кластер Kubernetes по своей сути состоит из:

- агента kubelet;
- планировщика;
- диспетчеров контроллеров (KCM и CCM);
- сервера API.

Все эти компоненты взаимодействуют друг с другом, обновляя информацию на сервере API. Например, если планировщик собирается запустить Pod на определенном узле, то для этого он изменяет определение Pod на сервере API. Если в процессе запуска агенту kubelet необходимо передать событие, он отправляет сообщение серверу API. Поскольку планировщик, kubelet и диспетчер контроллеров взаимодействуют через сервер API, они оказываются *слабо связанными* друг

с другом. Например, планировщик не знает, как kubelet запускает модули Pod, а kubelet не знает, как сервер API планирует запуск этих модулей. Другими словами, Kubernetes – это гигантская машина, которая постоянно хранит состояние всей инфраструктуры на сервере API.

Когда узлы, контроллеры или сервер API выходят из строя, возникает необходимость согласовать приложения в центре обработки данных, чтобы получить возможность запланировать контейнер для выполнения на другом узле, привязать тома к этому контейнеру и т. д. Все изменения состояния, сделанные через Kubernetes API, на самом деле сохраняются в etcd. В этом нет ничего нового в мире масштабируемых вычислений. Вы, наверное, слышали о таких инструментах, как ZooKeeper, которые используются аналогичным образом. На самом деле HBase, Kafka и многие другие распределенные платформы внутри используют ZooKeeper. База данных etcd – это просто современная версия ZooKeeper с некоторыми своими особенностями, касающимися хранения особо важных данных и согласования записей в случае сбоев.

12.1 Заметки для нетерпеливых

Теоретические аспекты сценариев распределенного консенсуса и аварийного восстановления в базах данных etcd могут вызвать потрясение у начинающих их изучать. Прежде чем окунуться в эту вселенную, рассмотрим некоторые особенности etcd в Kubernetes с практической точки зрения.

- Если данные, хранящиеся в etcd, потеряются, то кластер выйдет из строя. Делайте резервные копии etcd!
- Для запуска etcd v3 в промышленном окружении желательно использовать быстрые твердотельные накопители и высокопроизводительную сеть.

Единственная операция записи на диск в etcd, длящаяся более 1 с, может постепенно вывести из строя огромный кластер. В любой момент времени может выполняться множество операций записи, и это подразумевает жесткие требования к сети и диску – сеть с пропускной способностью не ниже 10 Гбит/с и твердотельные накопители. Вот выдержка из документации etcd (<https://etcd.io/docs/v3.3/op-guide/hardware/>): «Обычно должно выполняться не менее 50 последовательных операций ввода/вывода в секунду, что требует применения производительных дисков (например, со скоростью вращения 7200 об/мин)». Но нередко etcd требуется выполнять гораздо больше операций.

- В большинстве центров обработки данных и облачных окружений неизбежно возникают сбои, затрагивающие данный узел, поэтому в запасе должны иметься резервные узлы с etcd. Это означает, что в развертывании желательно иметь три или больше узлов etcd.

- Использующие etcd в кластерном окружении должны понимать, как работает реализация Raft, почему дисковый ввод/вывод важен для консенсуса Raft и как etcd использует процессор и память.
- В etcd сохраняется не только состояние кластера, но и все события. Поэтому следует подумать о хранении событий (которых много) в другой конечной точке etcd, чтобы операции с данными ядра кластера не конкурировали с менее важными операциями с метаданными событий.
- Инструмент командной строки etcdctl, предназначенный для взаимодействия с сервером etcd, имеет свой тест, позволяющий быстро проверить производительность etcd: `etcdctl check perf`.
- Если потребуется восстановить экземпляр etcd, следуйте инструкциям на <http://mng.bz/6Ze5>, где описывается, как вручную восстановить моментальный снимок etcd.

12.1.1 *Мониторинг производительности etcd с помощью Prometheus*

Большая часть информации в этом разделе основана на примерах. В Kubernetes и без того слишком много теории, связанной с настройкой и управлением etcd, поэтому, чтобы как-то компенсировать это, мы начнем с прикладных аспектов настройки и мониторинга etcd. Примеры, представленные здесь, довольно сложные. Вы можете следовать им, но не обязательно создавать все данные независимо, чтобы извлечь пользу из этого раздела.

На рис. 12.1 показан канонический поток процессов, возникающий при любом событии в кластере Kubernetes. Все операции записи в конечном итоге завершаются кворумом нескольких серверов etcd, согласившихся, что запись завершена. Изучение этой схемы даст нам основу для понимания практического сценария, который мы рассмотрим чуть ниже.

Каждое действие сервера API (например, каждый раз, когда создается простой Pod командой `kubectl create -f mypod.yaml`) приводит к синхронной записи в etcd. Это гарантирует, что запрос на создание Pod будет храниться на диске, если вдруг сервер API в какой-то момент выйдет из строя (что по закону больших чисел случится рано или поздно). Сервер API отправляет информацию серверу etcd, играющему роль «лидера», после чего начинается волшебство распределенного консенсуса, чтобы запечатлеть это в камне. На рис. 12.1 видно, что:

- в этом кластере есть три экземпляра etcd. Обычно используется три, пять или семь экземпляров. Количество экземпляров etcd всегда нечетное, чтобы всегда была возможность голосования по выбору нового лидера (лидерство в etcd мы рассмотрим ближе к концу этой главы);

- единственный сервер API может получить запрос на выполнение записи, после чего он сохранит данные в указанной конечной точке etcd, назначенной серверу etcd при запуске как `--etcd-servers`;
- скорость операции записи зависит от быстродействия самого медленного узла etcd. Если на одном из узлов вывод на диск происходит медленно, то это время доминирует над общим временем транзакции.

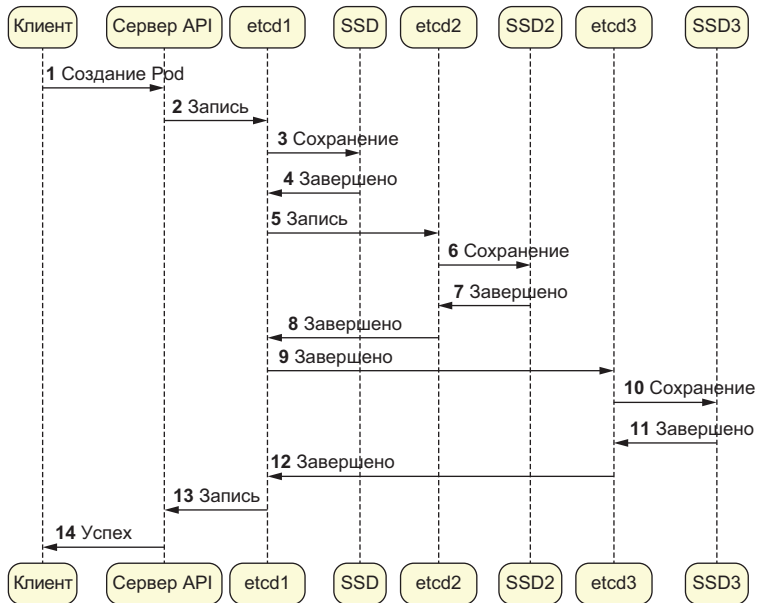


Рис. 12.1 Поток процессов, возникающий при появлении события в кластере Kubernetes

Теперь посмотрим на происходящее в исправном кластере с точки зрения etcd. Прежде всего нужно установить Prometheus. (Мы уже обсуждали этот вопрос, но в данном случае есть небольшое замечание: мы должны настроить Prometheus для работы в Docker и сбора информации из экземпляров etcd.) Возможно, вы помните, что при запуске сервиса Prometheus ему нужно передать файл YAML, чтобы сообщить, откуда должны собираться информация. Вот как выглядят настройки в этом файле, определяющие особенности анализа трех кластеров etcd, изображенных на рис. 12.1:

```
global:
  scrape_interval:    15s
  external_labels:
    monitor: 'myetcdscraper'
scrape_configs:
  - job_name: 'prometheus'
```

```
scrape_interval: 5s
static_configs:
  - targets: ['10.0.0.217:2381']
  - targets: ['10.0.0.251:2381']
  - targets: ['10.0.0.141:2381']
```

Для этого процесса ключевой является метрика *fsync*. Она сообщает, сколько времени занимает запись в etcd (на диск). Метрика разделена на сегменты (это гистограмма). Любая запись, занимающая около 1 с, может служить признаком, что производительность находится под угрозой. Увидев тенденцию роста количества операций записи, длящихся более чем, скажем, 0,25 с, можно начинать беспокоиться о замедлении кластера etcd, потому что это может замедлять работу кластера Kubernetes.

Запустив Prometheus с этой конфигурацией, можно создать несколько красивых графиков. Давайте взглянем на кластер Kubernetes, в котором все узлы etcd работают нормально. Гистограммы Prometheus вначале могут показаться нелогичными. Но важно помнить, что, если график конкретного сегмента изменит наклон, у вас могут возникнуть проблемы! На первом графике (рис. 12.2) видно, что:

- количество операций записи, длящихся более 1 с, незначительно;
- количество операций записи, длящихся более 0,5 с, незначительно;
- единственное отклонение в общей скорости записи имеет место в сегментах с высокой производительностью;
- самое главное, наклон линий не меняется.

Некоторые кластеры менее удачливы. Если развернуть тот же кластер на оборудовании с медленными дисками, мы в конечном итоге получим гистограмму, изображенную на рис. 12.3. В отличие от рис. 12.2 здесь видно резкое изменение наклона некоторых сегментов гистограммы с течением времени. Сегмент с наиболее сильно колеблющимся наклоном представляет операции записи, длящиеся менее 0,5 с. Поскольку обычно ожидается, что почти все записи будут длиться меньше этого предела, можно сделать вывод, что со временем кластер окажется под угрозой; однако не факт, что этот кластер испытывает проблемы или движется к катастрофе.

Теперь вы знаете, что можно организовать мониторинг etcd в кластере. Но как соотнести результаты мониторинга с реальными проблемами? Производительность операций записи на серверах etcd может привести к проблемам, связанным с частыми выборами лидера. Каждое событие выбора лидера в кластере Kubernetes означает, что в течение определенного времени *kubectl* будет практически бездействовать, пока сервер API ожидает возврата etcd в рабочий режим. Рассмотрим еще одну метрику: избрание лидера (рис. 12.4).

Чтобы увидеть последствия ситуации, представленной на рис. 12.3, рассмотрим метрику события выбора лидера: *etcd_server_is_leader*. Построив график этого события во времени (рис. 12.4), легко заме-

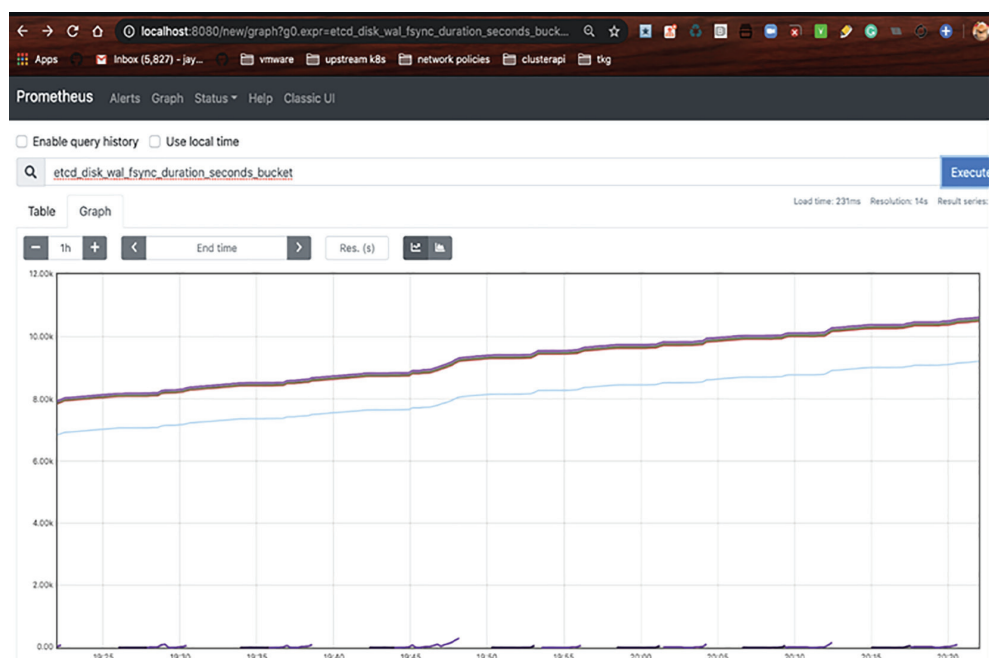


Рис. 12.2 График метрик etcd в исправно работающем кластере

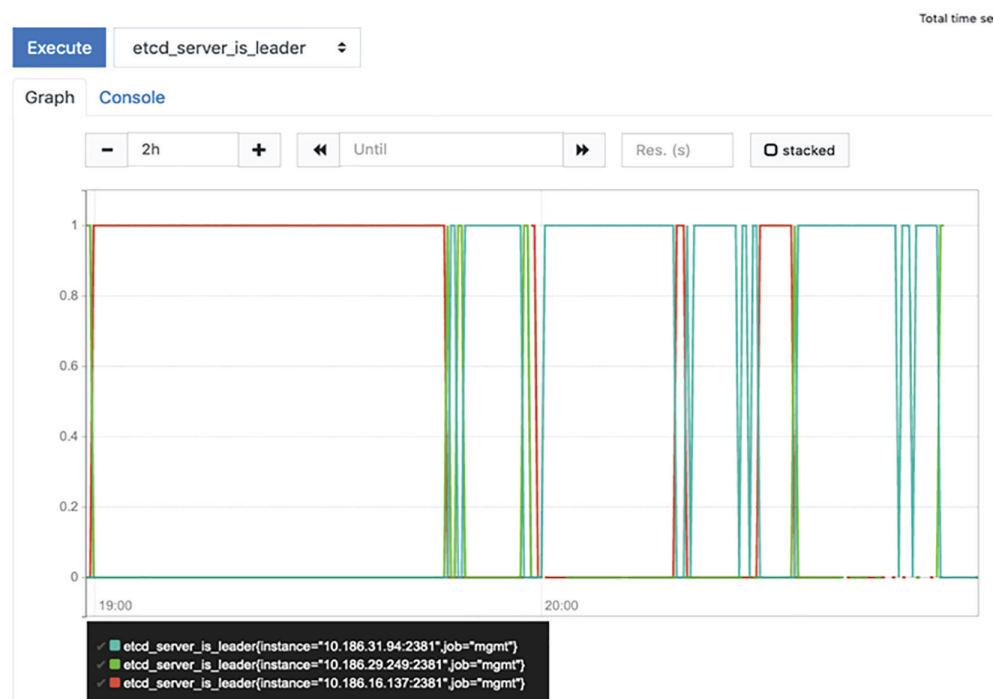


Рис. 12.3 График метрик etcd в кластере, испытывающем проблемы с производительностью

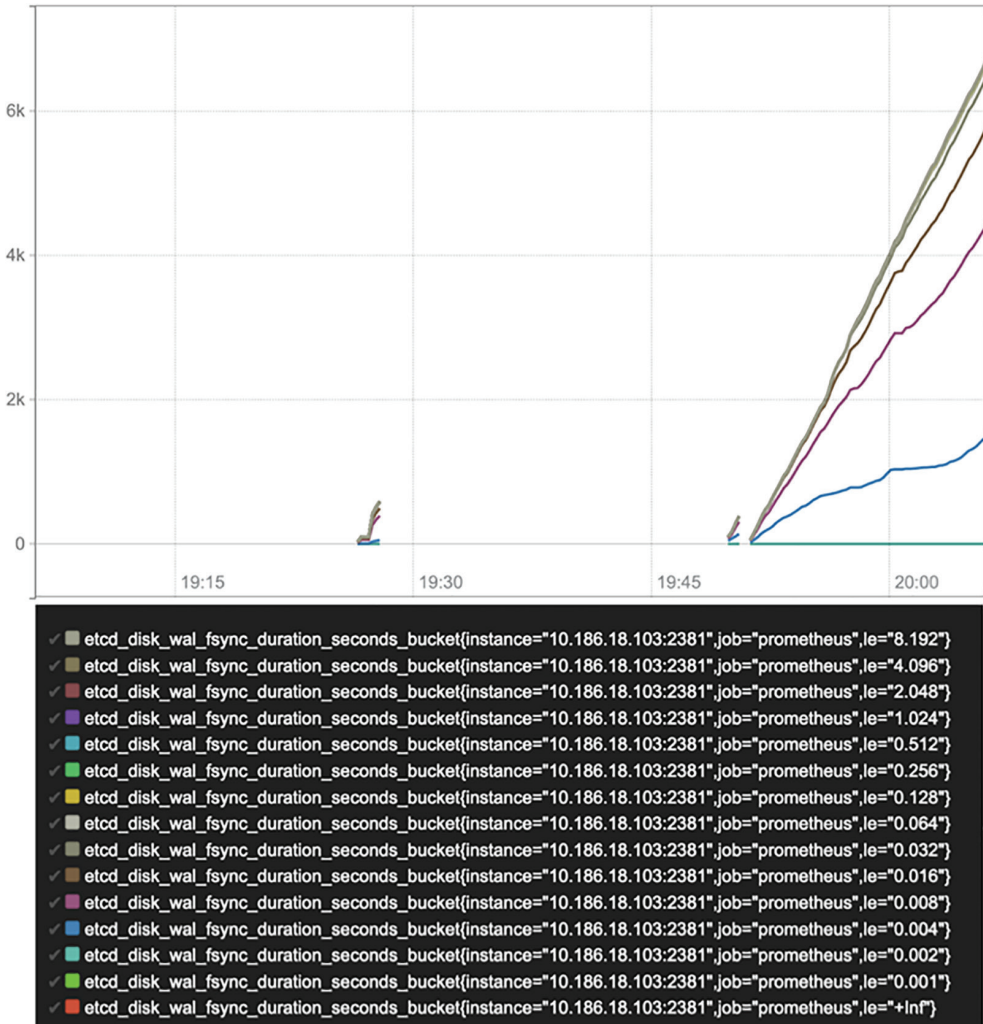


Рис. 12.4 График событий выбора лидера и метрики etcd

тить, когда в центре обработки данных происходят всплески голосований. Далее мы рассмотрим несколько простых дымовых тестов, которые можно запустить с помощью таких инструментов, как `etcdctl`, чтобы быстро продиагностировать отдельные узлы etcd.

12.1.2 Когда нужно настраивать etcd

Как отмечалось в предыдущем разделе, вам может понадобиться выполнить дополнительную настройку экземпляра etcd в промышленном окружении. Существуют сотни сценариев, которые могут привести к этой мысли. Рассмотрим пару примеров для конкретики. Есть много поставщиков услуг Kubernetes, которые сами управляют хра-

нищами etcd (развертывания на основе кластерного API позволяют держать etcd на отдельных узлах, которые можно воссоздать) или полностью скрывают работу etcd от вас (как, например, в GKE). В других случаях может понадобиться подумать о том, как устанавливается etcd и в каких условиях работает. В числе наиболее интересных вариантов можно назвать вложенную виртуализацию и установку Kubernetes на основе kubeadm. Рассмотрим их по порядку.

Вложенная виртуализация

Вложенная виртуализация распространена в окружениях разработки и тестирования. Например, для моделирования гипервизора vSphere можно использовать такую технологию, как VMware Fusion. В этом случае используются виртуальные машины, работающие внутри других виртуальных машин. Узлы в наших кластерах можно рассматривать как аналог вложенной виртуализации: у нас есть контейнеры Docker, которые запускаются внутри демона Docker, имитирующего виртуальные машины, а затем внутри этих узлов Docker запускаются контейнеры Kubernetes. В любом случае, как нетрудно догадаться, вложение одной виртуальной машины в другую приводит к огромным потерям производительности и не рекомендуется для промышленного использования. Основная причина, почему этот подход считается таким опасным, — из-за наличия нескольких уровней виртуализации добавляется задержка к операциям записи на жесткий диск. Эта задержка может сделать etcd крайне ненадежным.

Вложенная виртуализация ограничивает количество операций ввода/вывода в единицу времени и приводит к частым ошибкам записи. Сам кластер Kubernetes восстанавливается после них, но многие модули Pod будут постоянно терять статус лидера, если в Kubernetes используется Lease API, что становится все более распространенным явлением. В результате могут возникать ложные срабатывания и/или остановка работы долгоживущих приложений, основанных на консенсусе. Например, сам Cluster API (о котором мы поговорим ниже) сильно зависит от нормальной работы Lease API. Если вы используете Cluster API в качестве провайдера Kubernetes и etcd испытывает проблемы, то возникает риск невыполнения запросов кластера Kubernetes. Даже без кластерного решения, такого как Cluster API, которое зависит от etcd, все равно будут возникать проблемы, затрагивающие способность вашего сервера API отслеживать состояние узлов и получать обновления от контроллеров.

KUBEADM

Многие провайдеры Kubernetes используют kubeadm в роли инструмента установки по умолчанию и строительного блока для распространения Kubernetes. Однако он не имеет сквозной поддержки etcd по умолчанию. Для промышленного использования вам придется перенести собственное хранилище данных etcd в kubeadm вместо на-

строек по умолчанию, которые, несмотря на разумность, могут нуждаться в корректировке с учетом требований масштабируемости. Например, вы можете создать внешний кластер etcd с выделенным диском и назначить его для установки kubeadm.

12.1.3 Пример: быстрая проверка работоспособности etcd

Мы начали эту главу с обзора графиков производительности fsync в Prometheus, но в промышленных окружениях не всегда есть возможность построить красивые графики и поразглагольствовать. Один из самых простых способов быстро убедиться, что etcd работает, – использовать инструмент командной строки etcdctl, включающий встроенный тест производительности. Для примера зайдите по ssh (или выполнив команду `docker exec`, если вы экспериментируете с кластером kind) на узел, где работает etcd. Запустите `find`, чтобы отыскать местоположение двоичного файла `etcdctl`:

```
$> find / -name etcdctl # Это явно нестандартный способ,
                        # но он работает на любой машине
/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/
snapshots/13/fs/usr/local/bin/etcdctl
```

После этого используйте найденный двоичный файл и передайте ему необходимые сертификаты `casert`, которые, скорее всего, будут храниться в `/etc/kubernetes/pki/`, если используется кластер kind или Cluster API:

```
$> /var/lib/containerd/io.containerd.snapshotter.v1
➡ .overlayfs/snapshots/13/fs/usr/local/bin/etcdctl \
--endpoints="https://localhost:2379" \
--cacert="/etc/kubernetes/pki/etcd/ca.crt" \
--cert="/etc/kubernetes/pki/etcd/server.crt" \
--key="/etc/kubernetes/pki/etcd/server.key" \
  check perf
0 / 60 B
60 / 60 Boooooooooooooooooooooooooooo!
    100.00% 1m0s
PASS: Throughput is 150 writes/s
PASS: Slowest request took 0.200639s
PASS: Stddev is 0.017681s
PASS
```

Этот вывод сообщает, что etcd достаточно быстр для промышленного использования. О том, что конкретно означает вся эта информация, мы поговорим в оставшейся части главы.

12.1.4 etcd v3 и v2

При попытке запустить любую версию Kubernetes выше 1.13.0 с etcd v2 или ниже вы получите следующее сообщение об ошибке: «etcd2

больше не поддерживается реализацией хранилищ». Соответственно, в большинстве промышленных окружений используется etcd v3. Это хорошая новость, потому что с ростом размеров кластера количество пользователей может достичь сотен или даже тысяч, и в этом случае вам не обойтись без etcd v3:

- etcd v3 намного производительнее, чем etcd v2;
- etcd v3 использует gRPC для ускорения транзакций;
- etcd v3 имеет плоское (не иерархическое) пространство ключей, что обеспечивает более высокую скорость обслуживания тысяч клиентов;
- поддержка мониторинга в etcd v3, являющаяся основой для контроллера Kubernetes, позволяет наблюдать за множеством различных ключей по одному TCP-соединению.

Мы уже обсуждали etcd в других частях этой книги, поэтому будем считать, что вы знаете, почему это важно, и просто сосредоточимся на внутренней реализации etcd.

12.2 etcd как хранилище данных

Алгоритмы консенсуса всегда были ключевой частью распределенных систем, с самых первых дней. Еще в 1970-х правила Теда Кодда (Ted Codd) для баз данных широко использовались для упрощения транзакционного программирования, чтобы любой компьютерной программе не приходилось тратить время на разрешение противоречий и избыточности данных. Kubernetes в этом отношении ничем не отличается.

Архитектурные решения в плоскости данных, реализованной через etcd, и в плоскости управления (планировщик, диспетчеры контроллеров и сервер API) основаны на одном и том же принципе *согласованности любой ценой*. Поэтому etcd решает общую проблему согласования глобальных знаний. К числу базовых функций, лежащих в основе сервера Kubernetes API, относятся:

- создание пар ключ/значение;
- удаление пар ключ/значение;
- ключи наблюдения (с фильтрами выбора, которые могут предотвратить получение ненужных данных).

12.2.1 Можно ли запустить Kubernetes в других базах данных?

Механизм наблюдения в Kubernetes позволяет «наблюдать» за ресурсом API – за одним, а не за несколькими сразу. Это важно отметить, потому что реальному приложению может потребоваться создать несколько объектов-наблюдателей, чтобы реагировать на новые входящие события Kubernetes. Обратите внимание, что под ресурсами API подразумеваются объекты определенного типа, например Pod или

Service. Запустив наблюдение за ресурсом, можно организовать получение событий, влияющих на него (например, можно организовать получение событий от своего клиента, когда новый Pod добавляется или удаляется из кластера). Шаблон, используемый в Kubernetes для создания приложений на основе механизма наблюдения, известен как *шаблон контроллера*. Как рассказывалось выше в этой книге, контроллеры – это основа управления равновесием в кластерах Kubernetes.

Теперь давайте сосредоточимся на последней операции, отличающей Kubernetes от других приложений, работающих с базами данных. Большинство баз данных не поддерживает возможность наблюдения. В этой книге мы много раз упоминали о важности наблюдения. Поскольку сам фреймворк Kubernetes – это всего лишь набор контроллеров, поддерживающих баланс в распределенной группе компьютеров, необходим механизм мониторинга изменений. Вот некоторые базы данных, поддерживающие возможность наблюдения, о которых вы, возможно, слышали:

- Apache ZooKeeper;
- Redis;
- etcd.

Протокол Raft – это способ управления распределенным консенсусом. Он был разработан как продолжение протокола Paxos, используемого в Apache ZooKeeper. О Raft проще рассуждать, чем о Paxos. Он просто определяет надежный и масштабируемый способ согласования состояния базы данных ключей/значений в распределенной группе компьютеров. Проще говоря, Raft можно определить так:

- 1 в базе данных имеется лидер и несколько подчиненных узлов. Общее число узлов – нечетное;
- 2 клиент запрашивает операцию записи в базу данных;
- 3 сервер получает запрос и передает его нескольким подчиненным узлам;
- 4 после того как половина подчиненных узлов получит и подтвердит запрос на запись, сервер фиксирует его;
- 5 клиент получает ответ с признаком успешной записи;
- 6 если лидер выходит из строя, подчиненные узлы выбирают нового лидера и процесс продолжается, при этом старый лидер исключается из кластера.

Из вышеупомянутых баз данных только etcd поддерживает модель строгой согласованности, основанную на протоколе Raft, и конкретно создана для координации центров обработки данных. Именно поэтому она была выбрана для использования в Kubernetes. Тем не менее Kubernetes можно запустить с другой базой данных. Внутри etcd нет функций, специально предназначенных для Kubernetes. В любом случае основным требованием Kubernetes является возможность наблюдения за источником данных, чтобы выполнять такие задачи, как планирование модулей Pod, создание балансировщиков нагруз-

ки, предоставление хранилища и т. д. Однако семантика наблюдения в базе данных имеет такое же значение, как и качество согласуемых данных.

Как уже упоминалось, etcd v3 имеет возможность организовать наблюдение за множеством различных ключей по одному TCP-соединению. Это делает etcd v3 мощным компаньоном для больших кластеров Kubernetes. Таким образом, в Kubernetes есть второе требование к своей базе данных: согласованность.

12.2.2 Строгая согласованность

Представьте, что в канун Рождества вы запускаете сайт интернет-магазина, который должен работать максимально безотказно. Теперь представьте, что один из узлов etcd «решил», что нужно запустить два модуля Pod для масштабирования критической службы, хотя на самом деле нужны 10. В этом случае может произойти событие сокращения масштаба, противоречащее требованиям к доступности приложения. В этом случае стоимость перехода с правильного узла etcd на неправильный выше стоимости отказа вообще! Таким образом, etcd строго согласован. Такая согласованность достигается с помощью ключевых архитектурных констант etcd:

- в кластере etcd есть только один лидер, и точка зрения лидера на 100 % верна;
- в случае потери узла лидера нечетный кворум экземпляров etcd всегда может проголосовать и выбрать нового лидера;
- операция записи не считается осуществленной, пока она не будет подтверждена кворумом;
- узел etcd, не имеющий актуальной информации обо всех транзакциях, никогда не будет передавать данные. Это обеспечивается протоколом согласования под названием Raft, который мы обсудим ниже;
- кластер etcd в любой момент имеет одного и только одного лидера, которому передаются все запросы на запись;
- все операции записи блокируются в etcd до каскадной передачи их как минимум половине узлов в кворуме.

12.2.3 Согласованность в etcd обеспечивают операции fsync

Операции fsync блокируют операции записи на диск, что гарантирует согласованность etcd и запись данных на диск до возврата ответа. Это обстоятельство также может замедлить некоторые операции API, зато вы никогда не потеряете данные о состоянии кластера Kubernetes в случае сбоя. Чем быстрее ваши диски (да, именно диски, а не память или процессор), тем быстрее будет выполняться операция fsync:

- в промышленных кластерах обычно наблюдается снижение производительности (или сбой), если продолжительность выполнения операции fsync превышает 1 с;

- в типичном облаке можно ожидать, что эта операция завершится в течение 250 мс или около того.

Самый простой способ оценить работу etcd – посмотреть на производительность `fsync`. Давайте сделаем это в одном из кластеров `kind`, которые вы наверняка создали в предыдущих главах. В окне терминала выполните команду `docker exec -t -i <контейнер kind> /bin/bash`:

```
$ docker ps
CONTAINER ID   IMAGE                                COMMAND
ba820b1d7adb   kindest/node:v1.17.0               "/usr/local/bin/entr..."

$ docker exec -t -i ba /bin/bash
```

Теперь давайте посмотрим на скорость `fsync`. Метрики Prometheus, извлекаемые из etcd, можно получить с помощью `curl` или отобразить в виде графиков с помощью Grafana. Эти метрики сообщают продолжительность в секундах выполнения блокирующих операций `fsync`. В локальном кластере на SSD вы увидите, что они выполняются быстро. Например, в локальном кластере, работающем на ноутбуке с твердотельным накопителем, можно увидеть что-то вроде этого:

```
root@kind-control-plane:/#
curl localhost:2381/metrics|grep fsync
# TYPE etcd_disk_wal_fsync_duration_seconds histogram
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.001"} 1239
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.002"} 2365
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.004"} 2575
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.008"} 2587
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.016"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.032"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.064"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.128"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.256"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.512"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="1.024"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="2.048"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="4.096"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="8.192"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="+Inf"} 2588
etcd_disk_wal_fsync_duration_seconds_sum 3.181597084000007
etcd_disk_wal_fsync_duration_seconds_count 2588
```

Сегменты гистограммы в этом выводе говорят, что:

- 1 239 из 2 588 операций записи на диск продолжались менее 0,001 с;
- 2 587 операций вывода или 2 588 операций записи на диск продолжались менее 0,008 с;
- одна операция записи длилась 0,016 с;
- ни одна операции записи не потребовала больше 0,016 с.

Обратите внимание, что границы сегментов следуют экспоненциальному закону, потому что если операции записи занимают больше 1 с, то ваш кластер, вероятно, работает ненормально. В любой момент времени в Kubernetes могут запускаться сотни событий, выполняющих свою работу, и все они зависят от скорости ввода/вывода etcd.

12.3 Обзор интерфейса Kubernetes с etcd

Интерфейс хранилища данных в Kubernetes – это вполне конкретная абстракция, которую сам Kubernetes использует для доступа к базовому хранилищу данных. Единственной популярной и хорошо отлаженной реализацией хранилища данных для Kubernetes является etcd. Сервер API в Kubernetes абстрагирует некоторые базовые операции с etcd – Create, Delete, WatchList, Get, GetToList и List, – как показано в следующем фрагменте:

```
type Interface interface {  
    Create(ctx context.Context, key string, obj, out runtime.Object, ...  
    Delete(ctx context.Context, key string,  
        out runtime.Object, preconditions...  
    Get(ctx context.Context, key string,  
        resourceVersion string, objPtr runtime.Object,  
    GetToList(ctx context.Context, key string,  
        resourceVersion string, p SelectionPredicate, ...  
    List(ctx context.Context, key string,  
        resourceVersion string, p SelectionPredicate ...  
    ...  
}
```

Далее мы остановимся на WatchList и Watch. Эти функции являются особенностью etcd, отличающей ее от других баз данных (даже при том, что другие базы данных, такие как ZooKeeper и Redis, тоже реализуют этот API):

```
WatchList(ctx context.Context, key string, resourceVersion string ...  
Watch(ctx context.Context, key string, resourceVersion string, ...
```

12.4 Задача etcd – надежное хранение фактов

Строгая согласованность – ключевой аспект работы Kubernetes в промышленном окружении. Но как несколько узлов базы данных могут иметь одинаковое представление о системе в любой момент времени? Ответ прост: никак, это невозможно. Информации распространяется с ограниченной скоростью, и, хотя эта скорость высока, она не бесконечна. Всегда существует задержка между записью данных в разных местах.

На эту тему было написано много докторских диссертаций, и мы не будем пытаться объяснять теоретические ограничения консенсуса и строгой согласованности во всех деталях, но определим несколько концепций с конкретными особенностями Kubernetes, которые в конечном итоге зависят от способности etcd поддерживать согласованное представление о кластере. Например:

- за один раз можно принять только один новый факт, и эти факты должны передаваться на один узел, где работает плоскость управления;
- состояние системы в любой момент времени есть сумма всех текущих фактов;
- сервер Kubernetes API обеспечивает 100%-ную достоверность операций чтения и записи в отношении существующего потока фактов;
- поскольку в любой базе данных сущности могут меняться со временем, то могут быть доступны более старые их версии, и etcd поддерживает понятие версионирования.

Строгая согласованность обеспечивается в два этапа: установление лидерства в определенное время, чтобы каждый факт в потоке был принят всеми членами системы, а затем запись этого факта членам. Это (грубое) представление так называемого алгоритма консенсуса Paxos.

Предыдущая логика довольно сложна, поэтому представьте сценарий, в котором лидеры кластеров постоянно меняются и морят друг друга голодом. Под «морят голодом» мы подразумеваем сокращение времени безотказной работы etcd в сценарии выбора лидера, который осуществит запись. Если выборы лидера происходят часто, то страдает производительность операций записи. В Raft вместо блокировки выбора лидера для каждой новой транзакции факты постоянно отправляются только одним лидером. С течением времени лидер может оказаться недоступным, и тогда выбирается новый лидер.

etcd гарантирует, что недоступность лидера не приведет к несогласованному состоянию базы данных, прерывая операцию записи, если лидер пропадает во время транзакции, до того как запись будет произведена на 50 % узлов в кластере etcd. Это показано на рис. 12.1, где последовательность возвращается из запроса после того, как второй узел в кластере подтвердил запись. Обратите внимание, что в этот момент третий узел etcd может не спешить обновить свое внутреннее состояние, так как это не повлияет на общую скорость базы данных.

Это важно учитывать, рассматривая распределение узлов etcd, особенно если они распределены по разным сетям. В этом случае выборы могут быть более частыми, потому что лидеры могут теряться из-за задержек в сети. Таким образом, в любой момент большинство всех баз данных в кластере etcd будет иметь актуальный журнал всех транзакций.

12.4.1 Журнал упреждающей записи etcd

Надежность etcd обеспечивается еще и сохранением всех транзакций в журнале упреждающей записи (Write-Ahead Log, WAL). Чтобы понять важность WAL, посмотрим внимательнее, что происходит, когда выполняется запись.

- 1 Клиент отправляет запрос на сервер etcd.
- 2 Сервер etcd использует протокол консенсуса Raft для записи транзакции.
- 3 Raft окончательно подтверждает, что все узлы etcd, являющиеся членами кластера Raft, имеют синхронизированные файлы WAL. Таким образом, данные в кластере etcd всегда согласованы, даже если записи отправляются на разные серверы etcd в разное время. Это связано с тем, что все узлы etcd в кластере с течением времени приходят к единому консенсусу Raft о точном состоянии системы.

Фактически мы можем балансировать нагрузку клиента etcd даже притом, что etcd поддерживает строгую согласованность. У вас может возникнуть вопрос: как клиент может отправлять запросы на запись множеству серверов, не вызывая некоторую несогласованность хотя бы на короткие периоды времени. Причина в том, что реализация Raft в etcd пересылает запросы на запись лидеру независимо от источника. Запись считается незавершенной, пока лидер и половина других узлов в кластере не обновят свое состояние.

12.4.2 Влияние на Kubernetes

etcd реализует протокол консенсуса Raft, и, как следствие, мы в любой момент точно знаем, где сохраняется вся информация о состоянии Kubernetes. Никакое состояние не изменится в Kubernetes, если лидер etcd не принял запрос на запись и не переслал его большинству других узлов в кластере. В результате, когда etcd выходит из строя, сервер Kubernetes API тоже фактически отключается, когда дело доходит до большинства важных операций, которые он реализует.

12.5 Теорема CAP

Теорема CAP (Consistency, Availability, Partition tolerance – согласованность, доступность, устойчивость к разделению) – основополагающая теория в информатике. Узнать больше о ней можно по адресу https://ru.wikipedia.org/wiki/Теорема_CAP. Фундаментальный вывод теоремы CAP: невозможно получить в базе данных идеальную согласованность, доступность и устойчивость к разделению одновременно. etcd выбирает согласованность как наиболее важный аспект. Как результат, если один лидер в кластере etcd выходит из строя, то база данных оказывается недоступной, пока не будет выбран новый лидер.

Для сравнения: есть базы данных, такие как Cassandra, Solr и т. д., обеспечивающие более высокий уровень доступности и устойчивости к разделению; однако они не гарантируют согласованное представление данных на всех узлах в базе данных в каждый конкретный момент времени. В etcd, напротив, мы всегда имеем точное представление состояния базы данных. По сравнению с ZooKeeper, Consul и другими подобными хранилищами ключей/значений, производительность etcd чрезвычайно стабильна при больших масштабах, а предсказуемая задержка является ее самой привлекательной особенностью:

- Consul подходит для обнаружения сервисов и, как правило, хранит мегабайты данных, но при масштабировании увеличиваются задержки и снижается производительность;
- etcd подходит для надежного хранения ключей и значений, имея предсказуемую задержку, и способна обрабатывать гигабайты данных;
- ZooKeeper можно использовать взамен etcd, но с одной оговоркой: ее API слишком низкоуровневый, не поддерживает версионирование записей и труднее поддается масштабированию.

Теоретическая основа всех этих компромиссов называется *теоремой CAP*¹, которая диктует необходимость выбора между согласованностью данных, доступностью и устойчивостью к разделению. Например, если у нас есть распределенная база данных, то ее узлы должны обмениваться информацией о транзакциях. Мы можем сделать это сразу и строго, в таком случае мы всегда будем иметь непротиворечивые данные.

Почему невозможно обеспечить идеальную запись данных в распределенной системе? Потому что узлы могут выходить из строя, и, когда это происходит, требуется некоторое время на восстановление. Тот факт, что это время не равно нулю, означает, что базы данных с несколькими узлами, которые всегда должны быть согласованы друг с другом, иногда могут оказываться недоступными. Например, транзакции должны блокироваться, пока другие хранилища данных не смогут их выполнить.

Что произойдет, если мы решим, что у нас все в порядке с некоторыми базами данных, которые время от времени не получают транзакции (например, если происходит сбой в сети)? В этом случае можно пожертвовать согласованностью. Проще говоря, происходит выбор между двумя сценариями работы распределенной системы в реальном мире, т. е., когда сети работают медленно или машины работают неправильно, база данных:

- прекращает принимать транзакции (жертвует доступностью);
- или продолжает принимать транзакции (жертвует согласованностью).

Реальность этого выбора (опять же, теорема CAP) ограничивает способность распределенной системы достичь «идеала». Например,

¹ Еще ее называют теоремой Брюера. – Прим. перев.

реляционные базы данных обычно обеспечивают согласованность и устойчивость к разделению, тогда как другие базы данных, такие как Solr или Cassandra, обеспечивают устойчивость к разделению и высокую доступность.

CoreOS (компания, приобретенная компанией RedHat) разработала etcd для управления большими парками машин, создав хранилище ключей/значений, обеспечивающее согласованное представление состояния кластера для всех узлов. Благодаря ей появилась возможность обновлять серверы, просто просматривая состояние самой etcd. Как следствие, в Kubernetes было принято решение использовать etcd в качестве основы для сервера API, обеспечивающей строгую согласованность хранилища ключей/значений, где Kubernetes может хранить состояние кластера. В заключительном разделе этой главы мы рассмотрим несколько существенных аспектов использования etcd в промышленном окружении и, в частности, балансировку нагрузки, мониторинг и ограничения размера.

12.6 Балансировка нагрузки на уровне клиента и etcd

Как уже упоминалось, кластер Kubernetes включает плоскость управления, поэтому серверу API необходим доступ к etcd, чтобы реагировать на события от различных компонентов этой плоскости. В предыдущих примерах мы использовали curl для получения необработанных данных в формате JSON. Это удобно, но реальный клиент etcd должен иметь доступ ко всем членам кластера etcd, чтобы распределять нагрузку между узлами:

- клиент etcd пытается установить соединения со всеми конечными точками, и первое ответившее соединение остается открытым;
- etcd поддерживает TCP-соединение с конечной точкой, выбранной клиентом;
- в случае сбоев может произойти переход к другим конечным точкам.

Это распространенная идиома в фреймворке gRPC. Он основан на схеме мониторинга активности HTTPS.

12.6.1 Ограничения по размеру: о чем (не) следует беспокоиться

Сама база данных etcd накладывает некоторые ограничения по размеру и не предназначена для хранения терабайтов и петабайтов данных. Основной вариант ее использования – координирование и согласование распределенных систем (подсказка: /etc/ – это имя

каталога, где хранятся конфигурационные файлы программного обеспечения на ваших компьютерах с Linux). В рабочем кластере Kubernetes чрезвычайно грубой, но надежной начальной оценкой объема памяти и диска, приходящегося на одно пространство имен, является 10 Кбайт. Это означает, что кластер с 1000 пространств имен, вероятно, будет достаточно хорошо работать с 1 Гбайт ОЗУ. Однако, поскольку etcd использует большой объем памяти для управления механизмом наблюдения, что является доминирующим фактором в его требованиях к ОЗУ, эта минимальная оценка теряет смысл. В промышленном кластере Kubernetes с тысячами узлов следует подумать о возможности запуска etcd с 64 Гбайт ОЗУ, чтобы гарантировать эффективное обслуживание всех агентов kubelet и других клиентов сервера API.

Отдельные пары ключ/значение обычно занимают меньше 1,5 Мбайт (размер запроса операции обычно должен быть меньше этого значения). Это довольно распространено в хранилищах типа ключ/значение, потому что возможность дефрагментации и оптимизации хранилища зависит от фиксированного объема дискового пространства, занимаемого отдельными значениями. Однако это значение можно настроить параметром `max-request-bytes`.

Kubernetes явно не запрещает хранить произвольно большие объекты (например, ConfigMap с объемом данных > 2 Мбайт), но, в зависимости от настроек etcd, это может быть или не быть возможным. Имейте в виду, это очень важно, особенно учитывая, что каждый член кластера etcd хранит полную копию всех данных, поэтому распределение данных по сегментам невозможно.

РАЗМЕРЫ ОГРАНИЧЕНИЙ

Kubernetes не предназначен для хранения бесконечно больших типов данных, как и etcd: оба предназначены для обработки небольших пар ключ/значение, типичных для конфигурации и метаданных состояния в распределенных системах. Из-за этого конструктивного решения etcd имеет некоторые благонамеренные ограничения:

- запросы большего размера будут обрабатываться, но могут вызывать задержку обработки других запросов;
- по умолчанию максимальный размер любого запроса равен 1,5 Мбайт;
- это ограничение настраивается с помощью флага `--max-request-bytes` сервера etcd;
- общий размер базы данных ограничен:
 - максимальный размер хранилища по умолчанию равен 2 Гбайт, при этом желательно, чтобы размер базы данных etcd не превышал 8 Гбайт;
 - максимальный размер полезной нагрузки в etcd по умолчанию равен 1,5 Мбайт. Объем текста, описывающего модуль Pod, обычно составляет менее одного килобайта, поэтому, если вы не создаете CRD или другие объекты, размер которых в тысячу

раз больше размера обычного YAML-файла в Kubernetes, это ограничение не должно повлиять на вас.

Из вышеперечисленного можно сделать вывод, что сам фреймворк Kubernetes не предназначен для бесконечного роста с точки зрения объема хранилища. В этом есть определенный смысл. В конце концов, даже в кластере из 1000 узлов, если на каждом запустить по 300 модулей Pod и каждый модуль будет потреблять 1 Кбайт для хранения своей конфигурации, у вас все равно получится меньше мегабайта данных. Даже если с каждым модулем связано 10 карт ConfigMap одинакового размера, они все равно займут меньше 50 Мбайт.

Обычно вам не придется беспокоиться об общем размере etcd как факторе, ограничивающем производительность Kubernetes. Но придется позаботиться о скорости частоте запросов мониторинга и запросов с балансировкой нагрузки, особенно при большом количестве приложений. Причина в том, что для обслуживания конечных точек сервисов и внутренней маршрутизации требуется оперативно обновлять сведения об IP-адресах, назначенных модулям Pod, и если эта информация устареет, то способность маршрутизировать трафик кластера может быть потеряна.

12.7 Шифрование хранимых данных в etcd

Зайдя так далеко, вы теперь должны понимать, что в etcd хранится много информации, компрометация которой может привести к катастрофе в масштабе предприятия. Действительно, такие секреты, как пароли к базам данных, идиоматически хранятся в Kubernetes и в конечном счете в etcd.

Поскольку трафик между сервером API и различными клиентами защищен, то возможность украсть секрет из Pod, как правило, есть у тех, кто имеет доступ к клиенту kubectl, который поддается аудиту (и, соответственно, трассировке). По этим причинам etcd является самой ценной добычей для любого хакера в кластере Kubernetes. Давайте посмотрим, как Kubernetes API решает проблему шифрования:

- сам сервер Kubernetes API *поддерживает шифрование*; он принимает аргумент `--encryption-provider-config`, описывающий типы объектов API, которые должны шифроваться (как минимум секреты Secret);
- значение `--encryption-provider-config` представляет файл YAML, в котором перечисляются типы объектов API (например, Secret) и провайдеры шифрования. Их три: AES-GCM, AES-CBC и Secret Box;
- ранее перечисленные провайдеры применяются для расшифровывания в порядке убывания по алфавиту, а для шифрования используется первый элемент в списке провайдеров.

Таким образом, сервер Kubernetes API является наиболее важным инструментом управления безопасностью etcd в кластере Kubernetes. etcd – это один из множества инструментов, и важно не думать о нем как о высокозащищенной коммерческой базе данных. Возможно, в будущем технология шифрования будет внедрена в саму базу данных etcd, но на данный момент хранение данных на зашифрованном диске и шифрование непосредственно на стороне клиента – лучший способ защитить свои данные.

12.8 Производительность и отказоустойчивость etcd в глобальном масштабе

Под *глобальным развертыванием etcd* подразумевается возможность запускать etcd в режиме георепликации. Понимание возможных последствий этого требует пересмотра особенностей работы операций записи в etcd.

Как вы наверняка помните, etcd выполняет каскадную запись консенсуса через протокол Raft, а это означает, что более половины всех узлов etcd должны принять записываемые данные, прежде чем они станут официальными. Как уже упоминалось, консенсус в etcd является наиболее важным атрибутом в любом масштабе. По умолчанию etcd предназначена для поддержки локального, а не глобального развертывания, а это означает, что вам придется определить дополнительные настройки etcd для развертывания в глобальном масштабе. Таким образом, если у вас узлы etcd распределены по разным сетям, вам придется настроить несколько параметров, чтобы:

- смягчить процедуру выбора лидера;
- контрольные сообщения для мониторинга работоспособности посылались реже.

12.9 Интервал отправки контрольных сообщений в высокораспределенной etcd

Что делать, если используется единый кластер etcd, распределенный по разным центрам обработки данных? В этом случае нужно умерить свои ожидания в отношении пропускной способности операций записи, которая будет намного ниже. Согласно документации etcd, «разумное время прохождения запроса/ответа в пределах континентальной части США составляет 130 мс, а между США и Японией – около 350–400 мс». (Подробнее об этом на странице <https://etcd.io/docs/v3.4/tuning/>.)

Основываясь на этих данных, следует для начала увеличить интервал отправки контрольных сообщений, а также тайм-аут выбора лидера. При коротком интервале тратятся дополнительные такты процессора на отправку по сети избыточных данных. При слишком длинном интервале повышается вероятность, что может потребоваться избрание нового лидера. Ниже приводится пример, как настроить параметры, управляющие выборами лидера, для геораспределенного развертывания etcd:

```
$ etcd --heartbeat-interval=100 --election-timeout=500
```

12.10 Настройка клиента etcd в кластере kind

Одним из самых сложных аспектов доступа к etcd в работающем кластере Kubernetes – простое и безопасное выполнение запросов. Чтобы решить эту задачу, можно использовать следующий файл YAML (первоначальный вариант взят на <https://mauilion.dev>) для быстрого создания Pod, который можно использовать для выполнения команд etcd. В качестве примера сохраните следующий код в файле (с именем cli.yaml) и запустите кластер kind (или любой другой кластер Kubernetes). Возможно, вам придется изменить значение hostPath, указав местоположение учетных данных etcd:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    component: etcdclient
    tier: debug
  name: etcdclient
  namespace: kube-system
spec:
  containers:
    - command:
      - sleep
      - "6000"
      image: ubuntu
      name: etcdclient
      volumeMounts:
        - mountPath: /etc/kubernetes/pki/etcd
          name: etcd-certs
      env:
        - name: ETCDCCTL_API
          value: "3"
        - name: ETCDCCTL_CACERT
          value: /etc/kubernetes/pki/etcd/ca.crt
        - name: ETCDCCTL_CERT
          value: /etc/kubernetes/pki/etcd/healthcheck-client.crt
        - name: ETCDCCTL_KEY
```

← Замените это имя образа на etcd, если хотите использовать etcdctl для запросов к серверу API вместо curl

```

    value: /etc/kubernetes/pki/etcd/healthcheck-client.key
  - name: ETCDCTL_ENDPOINTS
    value: "https://127.0.0.1:2379"
hostNetwork: true
volumes:
  - hostPath:
      path: /etc/kubernetes/pki/etcd
      type: DirectoryOrCreate
    name: etcd-certs

```

Использование такого файла в активном кластере позволяет быстро и просто настроить контейнер, который можно использовать для отправки запросов в etcd. Например, с его помощью можно запускать команды, как показано ниже (после запуска `kubectrl exec -t -i etcd-client -n kube-system /bin/sh`, чтобы открыть терминал `bash`):

```

#/ curl --cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/peer.crt \
--key /etc/kubernetes/pki/etcd/peer.key https://127.0.0.1:2379/health

```

Чтобы вернуть etcd в работоспособное состояние или получить различные метрики Prometheus, выполните следующие команды:

```

#/ curl
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/peer.crt \
--key /etc/kubernetes/pki/etcd/peer.key \
https://127.0.0.1:2379/metrics

```

12.10.1 *Запуск etcd в окружении, отличном от Linux*

На момент написания этой книги etcd полностью поддерживала macOS и Linux, и лишь частично Windows. По этой причине кластеры Kubernetes, охватывающие несколько операционных систем (кластер Kubernetes с узлами Linux и Windows), обычно имеют плоскость управления кластером, размещенную в Linux, которая запускает etcd в модуле Pod. Кроме того, сервер API, планировщик и диспетчеры контроллеров Kubernetes тоже могут работать только в Linux и лишь в некоторых случаях в macOS. Таким образом, несмотря на способность Kubernetes поддерживать распределение рабочих нагрузок в операционных системах, отличных от Linux (в основном это относится к возможности запустить kubelet в Windows), вам все равно понадобится Linux, чтобы запустить сервер Kubernetes API, планировщика и диспетчера контроллеров (и, конечно же, etcd).

Итоги

- etcd – основа конфигурации почти всех современных кластеров Kubernetes.
- etcd – это база данных с открытым исходным кодом, относящаяся к одному семейству с ZooKeeper и Redis, с точки зрения общих шаблонов использования. Она не предназначена для хранения больших наборов данных.
- Kubernetes API абстрагирует пять основных вызовов API, поддерживаемых etcd и, что особенно важно, включает возможность просмотра отдельных элементов или списков.
- etcdctl – мощный инструмент командной строки для проверки пар ключ/значение, а также стресс-тестирования и диагностики проблем на узле кластера.
- etcd имеет ограничение по умолчанию 1,5 Мбайт для транзакций и, как правило, менее 8 Гбайт для наиболее распространенных сценариев.
- etcd, как и другие компоненты плоскости управления кластера Kubernetes, на самом деле поддерживается только в Linux, что является одной из причин, почему большинство кластеров Kubernetes, даже те, что предназначены для выполнения рабочих нагрузок Windows, включают по крайней мере один узел с Linux.

15 Безопасность контейнеров и модулей Pod

В этой главе:

- обзор основ безопасности;
- знакомство с приемами обеспечения безопасности контейнеров;
- ограничение модулей Pod с помощью контекста безопасности и лимитами ресурсов.

Для пущей безопасности компьютеры можно запереть в защищенном здании внутри охраняемого хранилища в клетке Фарадея, настроить систему с биометрическим входом без подключения к интернету... и все равно этих мер будет недостаточно, чтобы по-настоящему обезопасить компьютеры. Мы, как практикующие специалисты в области Kubernetes, должны принимать обоснованные решения по обеспечению безопасности, исходя из потребностей бизнеса. Если закрыть все кластеры Kubernetes в клетке Фарадея и отключить их от интернета, то они станут непригодными для использования. Но, если не уделить безопасности должного внимания, мы позволим посторонним (например, майнерам биткоинов) вторгаться в наши кластеры.

По мере развития и распространения Kubernetes вскрываются новые уязвимости. Рассуждая о безопасности, следует помнить, что всегда есть риск взлома вашей системы! И если такое (не дай бог) случится, задайте себе следующие вопросы.

- Что взломщик сможет взломать?
- Что он может сделать?
- Какие данные он сможет получить?

Безопасность – это ряд компромиссов, которые часто трудно найти. Используя Kubernetes, мы представляем систему, которая может отпугнуть людей, когда они узнают, что есть возможность открыть доступ к балансировщику нагрузки из интернета всего одним вызовом API. Но, используя простые методы, можно уменьшить влияние возможных рисков. Следует признать, что большинство компаний не планирует такие базовые операции, как обновление системы безопасности в контейнерах.

Безопасность – это компромисс. Безопасность может замедлить развитие бизнеса. Безопасность – это хорошо, но, если забраться в кроличью нору безопасности слишком глубоко, можно потерять деньги и замедлить работу предприятий и организаций. Мы должны сбалансировано применять меры безопасности и принимать решение о том, стоит ли еще глубже залезть в эту кроличью нору.

Нет необходимости реализовывать все меры безопасности самостоятельно. Существует расширяющийся набор инструментов, которые наблюдают за контейнерами внутри Kubernetes и определяют возможные бреши в безопасности; например, Open Policy Agent (OPA), о котором мы поговорим в главе 14. В конце концов, компьютер, подключенный к интернету, просто не может быть в полной безопасности.

Как отмечалось в первой главе этой книги, процесс DevOps базируется на автоматизации, как и Kubernetes. В этой главе мы поговорим о том, что нужно сделать, чтобы автоматизировать безопасность Kubernetes. Но для начала рассмотрим некоторые понятия безопасности, чтобы сформировать правильное мышление.

ПРИМЕЧАНИЕ Следующие две главы можно было бы развернуть и написать целую книгу, однако мы намерены сделать эти главы кратким справочником, а не исчерпывающим руководством. Прочитав справочник, вы сможете пойти дальше и поближе познакомиться с каждой темой.

13.1 Радиус взрыва

Когда что-то взрывается, расстояние от центра взрыва до его края называют радиусом взрыва. Но какое отношение это имеет к компьютерной безопасности (или кибербезопасности)? Когда компьютерная система скомпрометирована, происходит взрыв, и обычно не один, а несколько. Представьте, что у вас есть несколько контейнеров, работающих на нескольких узлах с несколькими компонентами безопасности. Как далеко распространится взрыв?

- Сможет ли скомпрометированный Pod получить доступ к другому Pod?
- Можно ли использовать скомпрометированный Pod для создания другого Pod?
- Можно ли использовать скомпрометированный Pod для управления узлом?
- Можно ли с узла node01 проникнуть в узел node02?
- Можно ли из Pod проникнуть во внешнюю базу данных?
- Можно ли проникнуть, скажем, в систему LDAP?
- Сможет ли хакер получить доступ к системе управления версиями или секретам Secret?

Точка вторжения в систему безопасности – это эпицентр большого взрыва. Расстояние проникновения злоумышленника от точки вторжения – это радиус взрыва. Внедрив простые стандарты безопасности, например запрет на запуск процессов от имени суперпользователя root или использование RBAC, можно существенно ограничить радиус взрыва, когда он случится.

13.1.1 Уязвимости

Уязвимость – это слабое место. (Например, трещина в плотине.) Стремясь обезопасить свои системы, мы пытаемся предотвратить уязвимость (появление трещины в плотине), а не исправить ее. Следующие две главы посвящены уязвимостям Kubernetes изнутри, а также методам усиления безопасности.

13.1.2 Вторжение

Вторжение – это самое нежелательное явление! Это взлом, когда злоумышленник использует уязвимость и проникает в нашу систему. Например, злоумышленник обретает контроль над модулем Pod и получает доступ к `curl` или `wget`, затем создает еще один Pod, работающий как корень в сети хоста, и ваш кластер оказывается скомпрометированным целиком.

13.2 Безопасность контейнера

Наиболее очевидной начальной точкой для работы с безопасностью в Kubernetes является уровень контейнера, потому что практически каждый кластер Kubernetes работает в нескольких контейнерах. Поэтому первой линией обороны кластера являются контейнеры. Важно помнить, что пользовательское программное обеспечение, работающее внутри контейнера, уязвимо для атак.

Запуская приложение и открывая доступ к нему из внешнего мира, имейте в виду, что некоторые люди могут иметь злой умысел. Напри-

мер, однажды мы заметили, что уровень потребления процессора на узле сошел с ума. Разработчик развернул приложение с известной ошибкой, и майнеры использовали модуль Pod для майнинга, и все это произошло в течение нескольких часов. Никогда не забывайте, что, независимо от предпринимаемых мер безопасности на уровне контейнера, если приложение имеет критическую уязвимость, то контейнер оказывается широко открытым для атак. Хакеру понадобится совсем немного времени, может быть, несколько минут, чтобы найти кластер, и несколько секунд, чтобы поместить в контейнер программное обеспечение для майнинга. А теперь познакомимся с некоторыми методами защиты контейнеров.

13.2.1 *Планирование обновления контейнеров и пользовательского программного обеспечения*

Обновление – это первое, чего *не* делают в компаниях, с которыми мы сталкивались. Честно говоря, это страшно, страшнее самого жуткого фильма ужасов. Крупные компании допускают утечки данных, просто потому что не обновили зависимости программного обеспечения или не пересобрали базовый образ.

Желательно заранее заложить в проект затраты на обновление программного обеспечения в случае обнаружения уязвимости в системе безопасности. Получив отказ, вы сможете ненавязчиво напомнить о плохой рекламе компаниям, допустившим утечку информации о клиентах. Кроме того, утечки часто обходятся компаниям в миллионы долларов.

Обновляйте контейнеры по мере выпуска новых базовых версий и появления сообщений об новых уязвимостях в CVE. Программа CVE регулярно публикует уведомления об обнаруженных уязвимостях на сайте <https://cve.mitre.org/>. Так обязательно запланируйте обновление зависимостей прикладного программного обеспечения – обновлять нужно не только контейнер, вмещающий прикладное программное обеспечение, но и само это программное обеспечение.

13.2.2 *Контроль контейнеров*

Контроль контейнеров – это система, выявляющая уязвимости в контейнерах (например, вспомните, когда в 2014 году в реализации OpenSSL обнаружилась ошибка Heartbleed). Система, просматривающая ваши образы, *совершенно необходима* в современных условиях. Программное обеспечение в контейнерах обязательно должно проверяться на наличие уязвимостей и обновляться.

Под программным обеспечением подразумевается все, что установлено, включая OpenSSL и Bash. Обновляться должно не только прикладное программное обеспечение, но также базовые образы, определяемые в разделе FROM. Это тяжелый труд. Мы не понаслыш-

ке знаем, сколько времени и денег это требует. Настройте системы непрерывной интеграции/доставки (CI/CD) и другие инструменты быстрой сборки, тестирования и развертывания контейнеров. Многие коммерческие реестры контейнеров имеют системы, которые могут проверить ваши контейнеры, но если никто не просматривает их уведомления, то некому будет на них реагировать. Создайте систему трассировки таких уведомлений или приобретите коммерческое программное обеспечение, которое поможет вам в этом.

13.2.3 Пользователи в контейнерах – не запускайте ПО от имени root

Не запускайте программы внутри контейнера от имени суперпользователя root. Существует такое понятие, как «выпадение в командную оболочку» (popping a shell), означающее выход из пространства имен контейнера Linux и получение доступа к командной оболочке. Из оболочки можно получить доступ к серверу API и, возможно, к узлу. Иногда оболочка может иметь привилегии, необходимые для загрузки сценария из интернета и его запуска. Запуск от имени пользователя root дает привилегии суперпользователя в контейнере и, возможно, в хост-системе, если удастся выйти за границы контейнера.

Определяя контейнер, можно добавить нового пользователя и группу с помощью `adduser` и запускать приложение от имени этого пользователя. Вот пример создания нового пользователя в контейнере Debian:

```
$ adduser --disabled-password --no-create-home --gecos '' \  
--disabled-login my-app-user
```

Теперь можно запустить приложение от имени этого пользователя:

```
$ su my-app-user -c my-application
```

Работа с привилегиями root в контейнере чревата теми же последствиями, что и работа с привилегиями root в хост-системе: root – это царь и бог. Кроме того, в определении модуля Pod можно определить настройки `runAsUser` и `fsGroup`. Мы рассмотрим их позже.

13.2.4 Используйте наименьшие возможные контейнеры

Рекомендуется использовать легковесную контейнерную операционную систему, предназначенную для работы только в качестве контейнера. Это ограничивает количество программ в контейнере и, следовательно, количество уязвимостей. Проекты без дистрибутива от Google предоставляют легковесные контейнеры, зависящие от языка.

Включение в контейнер только программного обеспечения, необходимого вашему приложению, – это передовая практика, используемая Google и другими технологическими гигантами, использую-

щими контейнеры вот уже много лет. Это улучшает отношение сигнал/шум в сканерах (например, CVE) и облегчает бремя выбора только того, что вам нужно.

– Open Web Application Foundation Security Cheat Sheet
(<http://mng.bz/g42v>)

Проект без дистрибутива включает базовый слой, а также контейнеры для запуска различных языков программирования, таких как Java. Ниже показан пример приложения Go, использующего контейнер `golang` для сборки программного обеспечения, за которым следует контейнер без дистрибутива:

```
# Запуск сборки приложения.
FROM golang:1.17 as build

WORKDIR /go/src/app
COPY . .

RUN go get -d -v ./...
RUN go install -v ./...

# Копирование приложения в базовый образ.
FROM gcr.io/distroless/base
COPY --from=build /go/bin/app /
CMD ["/app"]
```

А еще есть вспомогательное ПО. Представьте, что вы устанавливаете cURL для загрузки двоичного файла в процессе создания контейнера. Затем cURL необходимо удалить. Дистрибутив Alpine реализует такую возможность, автоматически удаляя компоненты, используемые в сборке, но в Debian и в других дистрибутивах такая возможность не предусмотрена. Если вашему приложению не нужно какое-то ПО, то не устанавливайте его. Чем больше программ вы установите, тем больше возможных уязвимостей получите. Обратите также внимание, что в примере отсутствует создание нового пользователя для запуска двоичного файла, – запускайте программы от имени `root`, только если это действительно необходимо.

13.2.5 Происхождение контейнера

Одной из ключевых проблем безопасности в кластерах Kubernetes – знание, какие образы контейнеров выполняются внутри каждого модуля Pod, и учет их происхождения. Под знанием *происхождения контейнера* подразумевается наличие достоверной информации об источнике контейнера и гарантий следования четким правилам в процессе создания артефакта (контейнера).

Не развертывайте контейнеры из реестров, не контролируемых вашей компанией. Проекты с открытым исходным кодом прекрасны, но желательно собрать эти контейнеры локально и поместить в свой репо-

зиторий. Теги контейнера можно менять, нельзя изменить только контрольную сумму SHA. Нет никакой гарантии, что контейнер на самом деле является именно тем, за кого себя выдает. Возможность отследить происхождение контейнера позволяет гарантировать, что развернутый контейнер можно идентифицировать и источник пользуется доверием.

Команда Kubernetes создала базовый слой образа для всех контейнеров, работающих внутри кластера Kubernetes. Это дает командам уверенность, что образ имеет безопасное происхождение и является согласованным и проверенным. Безопасность происхождения также означает, что все образы поступают из известного источника. А согласованность гарантирует выполнение определенных шагов при создании образа и более герметичную среду. Наконец, известность происхождения гарантирует известность источника и целостность контейнера перед запуском.

13.2.6 *Линтеры для контейнеров*

Автоматизация способствует уменьшению трудозатрат на совершенствование систем. Безопасность требует больших усилий, но, честно говоря, и их можно уменьшить, воспользовавшись инструментом статического анализа (линтером), таким как hadolint, для поиска общих проблем с контейнерами и пользовательским программным обеспечением, которые могут вызвать уязвимости в системе безопасности. Ниже приводится краткий список линтеров, которые мы использовали в прошлом:

- hadolint для файлов Dockerfile (<https://github.com/hadolint/hadolint>);
- команда go vet (<https://golang.org/cmd/vet/>);
- Flake8 для Python (<http://flake8.pycqa.org/en/latest/>);
- ShellCheck для Bash (<https://www.shellcheck.net/>).

Теперь, поговорив о безопасности контейнеров, перейдем на следующий уровень и обратим внимание на модули Pod.

13.3 *Безопасность модулей Pod*

Kubernetes позволяет определять разрешения для пользователей в модуле Pod и для модулей Pod за пределами пространства имен Linux (например, возможность монтирования тома на узле). Злоумышленник, взломавший модуль Pod, может взломать кластер! Используя такие команды, как `nsenter`, можно войти в корневой процесс (`/proc/1`), создать командную оболочку и выполнить операции с привилегиями root на фактическом узле, где выполняется скомпрометированный модуль Pod. Kubernetes API позволяет определять разрешения для модулей Pod и обеспечивает дополнительную защиту модулей, узлов и кластера в целом.

ПРИМЕЧАНИЕ Некоторые дистрибутивы Kubernetes, такие как OpenShift, добавляют дополнительные уровни безопасности, и вам может потребоваться определить дополнительные конфигурационные параметры для использования таких API, как контекст безопасности.

13.3.1 Контекст безопасности

Выше уже упоминалось, что контейнеры не должны запускаться от имени пользователя root. Для поддержки этого правила Kubernetes позволяет определить идентификатор пользователя для Pod. В определении объекта Pod можно указать три идентификатора:

- `runAsUser` – идентификатор пользователя для запуска процесса;
- `runAsGroup` – идентификатор группы для запуска процесса;
- `fsGroup` – идентификатор второй группы, используемый для монтирования любых томов и файлов, созданных процессами в модуле Pod.

Если есть контейнер, работающий от имени root, его можно принудительно запустить с другим идентификатором пользователя. Повторим еще раз: ни один контейнер не должен запускаться от имени root, иначе пользователь может случайно пропустить определение `securityContext`. Ниже представлен фрагмент определения Pod с контекстом безопасности:

```
apiVersion: v1
kind: Pod
metadata:
  name: sc-Pod
spec:
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
    fsGroupChangePolicy: "OnRootMismatch"
  volumes:
  - name: sc-vol
    emptyDir: {}
  containers:
  - name: sc-container
    image: my-container
    volumeMounts:
    - name: sc-vol
      mountPath: /data/foo
```

При запуске модуля Pod веб-сервер NGINX будет работать с идентификатором пользователя 3042

Пользователь с идентификатором 3042 принадлежит к группе 4042

Запись в файлы процесс NGINX будет осуществлять с идентификатором группы 5042

Изменяет владельца тома перед его подключением к модулю Pod

Точка монтирования

Посмотрим влияние этих настроек, запустив модуль в кластере `kind`. Сначала запустим сам кластер:

```
$ kind create cluster
```

Затем развернем NGINX, используя контейнер по умолчанию:

```
$ kubectl run nginx --image=nginx
```

После запуска NGINX выполним команду `exec`, чтобы войти в контейнер Docker:

```
$ docker exec -it a62afaadc010 /bin/bash
root@kind-control-plane:/# ps a | grep nginx
2475 0:00 nginx: master process
➔ nginx -g daemon off;
2512 22:36 0:00 nginx: worker process
```

Процесс NGINX запускается с привилегиями root

Как видите, процесс NGINX запущен с привилегиями root, и это чревато опасностями. Чтобы предотвратить их, остановим этот Pod и запустим другой. Следующая команда останавливает и удаляет модуль Pod с веб-сервером NGINX:

```
$ kubectl delete po nginx
```

Теперь создадим Pod с контекстом безопасности следующей командой:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: sc-Pod
spec:
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
  volumes:
  - name: sc-vol
    emptyDir: {}
  containers:
  - name: sc-container
    image: nginx
    volumeMounts:
    - name: sc-vol
      mountPath: /usr/share/nginx/html/
EOF
```

Догадайтесь, что произойдет? Модуль Pod просто не запустится. Заглянем в журнал:

```
$ kubectl logs sc-Pod
```

Эта команда выведет:

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to
➔ perform configuration
```

```
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/
➔ 10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: error: can not modify /etc/nginx/conf.d/
➔ default.conf (read-only file system?)
/docker-entrypoint.sh: Launching /docker-entrypoint.d/
➔ 20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2020/11/08 22:44:59 [warn] 1#1: the "user" directive makes sense only if
➔ the master process runs with super-user privileges, ignored
➔ in /etc/nginx/nginx.conf:2
nginx: [warn] the "user" directive makes sense only if the master process
➔ runs with super-user privileges, ignored in
➔ /etc/nginx/nginx.conf:2
2020/11/08 22:44:59 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp"
➔ failed (13: Permission denied)
nginx: [emerg] mkdir() "/var/cache/nginx/client_temp" failed
➔ (13: Permission denied)
```

Заметили проблему? Она состоит в том, что NGINX требует определенных настроек для запуска от имени простого непривилегированного пользователя, тогда как большинство других приложений требует специальных настроек для запуска от имени root. Дополнительную информацию, касающуюся этого конкретного случая, вы найдете на странице <http://mng.bz/ra4Z> или <http://mng.bz/Vl4O>.

Отсюда следует вывод: не запускайте контейнеры от имени root и используйте контекст безопасности, чтобы гарантировать невозможность запуска от имени root.

СОВЕТ SSL-сертификаты – это проблема, и код, использующий их, может быть очень сложным. Вы можете столкнуться с проблемой, когда код проверяет соответствие пользователя сертификата идентификатору процесса. Попытка внедрить сертификат TLS как секрет, который не использует fsGroup, может вызвать неразбериху. К сожалению, в настоящее время эта проблема еще не решена в Kubernetes.

13.3.2 Расширение привилегий и возможностей

В модели безопасности Linux традиционные разрешения UNIX делятся на две категории: привилегированные и непривилегированные:

- *привилегированный пользователь* – это пользователь root или другой, чей эффективный идентификатор пользователя установлен равным нулю с помощью `sudo`, чтобы выполнить действия от имени root;
- *непривилегированный пользователь* – обычный пользователь, идентификатор которого не равен нулю.

Когда действия выполняет привилегированный пользователь, ядро Linux не проверяет никаких разрешений. Вот почему можно запустить

жуткую команду `rm -rf /` от имени пользователя `root`. Большинство современных дистрибутивов в ответ на эту команду хотя бы спрашивают: действительно ли вы желаете удалить всю файловую систему. При выполнении действий непривилегированным пользователем производится проверка разрешений безопасности, основанная на идентификаторе процесса.

Определяя непривилегированного пользователя и присваивая ему идентификатор, можно дать ему определенные *возможности*, позволяющие непривилегированному пользователю выполнять определенные действия, например вносить изменения в UID и GID файлов. Имена всех этих возможностей начинаются с префикса `CAP`; только что упомянутая возможность называется `CAP_CHOWN`. Замечательно! Но какое отношение все это имеет к нам?

Помните, как мы предупреждали, – не запускать ничего с привилегиями `root`? Представьте, что у нас есть модуль Pod, который должен управлять правилами `iptables` узла сети или BPF (Berkley Packet Filter – фильтр пакетов Berkley), затрагивающими провайдера CNI, и для нас нежелательно запускать этот модуль от имени `root`. Kubernetes позволяет установить контекст безопасности, определить идентификатор пользователя, а затем добавить определенные возможности. Например:

```
apiVersion: v1
kind: Pod
metadata:
  name: net-cap
spec:
  containers:
  - name: net-cap
    image: busybox
    securityContext:
      runAsUser: 3042
      runAsGroup: 4042
      fsGroup: 5042
      capabilities:
        add: ["NET_ADMIN", "BPF"]
```

Предоставляет пользователю
возможности `CAP_NET_ADMIN` и `CAT_BPF`

Обратите внимание на отсутствие префикса `CAP`. Вместо возможности `CAP_NET_ADMIN` указана `NET_ADMIN`. Разрешения `CAP` позволяют модулям Pod выполнять многие привилегированные операции, в том числе перезагружать узлы (при наличии разрешения `CAP_SYS_BOOT`). Кроме того, внутри `CAP` существует подмножество разрешений, имена которых начинаются с префикса `CAP_SYS`. Это очень мощные разрешения. Например, `CAP_SYS_ADMIN` выдает права `root`.

У нас есть наборы демонов `DaemonSet`, модули Pod и развертывания `Deployment`, управляющие кластером Kubernetes, изменяющие правила `iptables`, загружающие компоненты Kubernetes и т. д. Вариантов использования очень много. Но повторим еще раз – если нет явной необходимости, не запускайте ничего от имени `root`, дайте процессу как можно меньше разрешений, определив разрешения `CAP`. Правда,

эти разрешения недостаточно точны, как хотелось бы. Например, нет отдельного разрешения на монтирование файловой системы. Для этого приходится использовать `CAP_SYS_ADMIN`.

13.3.3 Политики безопасности Pod (PSP)

ПРИМЕЧАНИЕ Начиная с Kubernetes v1.21, политики безопасности Pod (Pod Security Police, PSP) объявлены устаревшими и запланированы для удаления в версии v1.25. Им на смену должны прийти разрешения безопасности Pod Security Admission (<http://mng.bz/5QQ4>). Мы включили в книгу этот раздел, потому что многие ранее использовали PSP, а на момент написания этой книги стандарты безопасности Pod еще не вышли из стадии бета-тестирования.

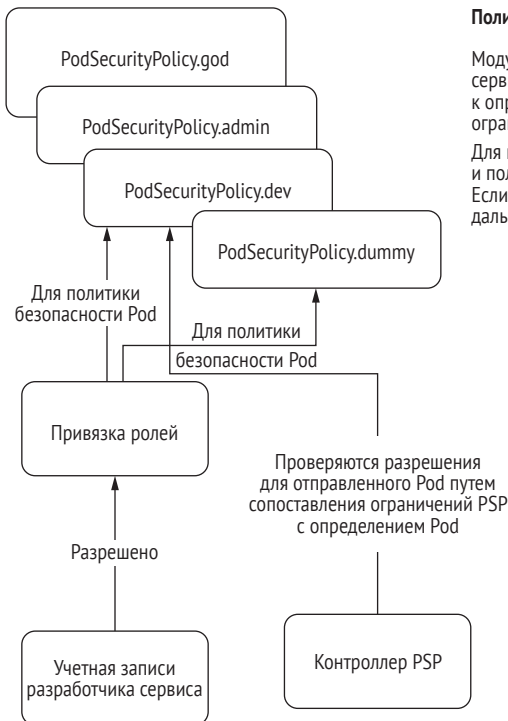
Чтобы гарантировать создание надлежащих контекстов безопасности, можно определить политику безопасности PSP, которая обеспечит заданную настройку контекста безопасности. Как и все другие конструкции Kubernetes, `PodSecurityPolicy` является объектом API:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false # Запретить запуск привилегированных модулей Pod!
  # Далее следуют определения некоторых обязательных полей.
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

Этот фрагмент кода устанавливает ряд произвольных правил для SELinux, пользователей и т. д. Обратите внимание, что сделать каждый контейнер безопасным просто невозможно. Если посмотреть на гиперкубы, сети, плагины хранилищ и т. д., то можно заметить, что эти инструменты административной инфраструктуры системного уровня выполняют привилегированные действия (например, установку правил iptables). После добавления PSP многие модули Pod могут перестать работать, причем в самый неподходящий момент. Это связано с тем, что PSP реализуются при анализе привилегий контроллерами допуска. Давайте посмотрим на жизненный цикл аудита безопасности для контейнеров в кластере Kubernetes:

- день 0 – модули Pod могут выполнять любые операции, включая запуск от имени root и создание файлов на хостах;
- день 1 – разработчики создают контейнеры на работавших в день 0;
- день 2 – специалисты по безопасности приобретают эту книгу;
- день 3 – в кластер добавляется аутентификация и авторизация на основе ролей (RBAC). Теперь вызовы API могут выполнять только учетные записи с административными привилегиями;
- день 4 – в кластер добавляются политики безопасности PSP;
- день 5 – половина узлов отключается на техническое обслуживание;
- день 6 – несколько модулей Pod не смогли запуститься надлежащим образом.

Причина, почему проблемы с модулями Pod проявляются только на шестой день после добавления PSP, заключается в том, что после того, как Pod остановится, его PSP будет протестирован в промышленном окружении. Вспомним, как работают контейнеры. Действующий контейнер уже имеет PID, он выполнил все необходимые команды и связан с сетевым устройством хоста. Поэтому изменение политик во время работы контейнера не устраняет векторы угроз, а лишь предотвращает их появление в новых местах. На рис. 9.1 показан процесс PSP для Kubernetes.



Политика безопасности модулей Pod (PSP) и RBAC

Модули Pod рано или поздно передаются в учетную запись сервиса, которая имеет авторизацию RBAC для доступа к определенным PSP. Если авторизованный PSP проходит ограничения, то Pod передается на сервер API.

Для иллюстрации мы добавили фиктивную политику dummy и политику разработчика dev для учетной записи разработчика. Если какая-то из них разрешает доступ Pod, то он передается дальше; иначе – блокируется

Рис. 13.1 Политики безопасности модулей Pod (PSP)

Это важная концепция, которую следует помнить на протяжении оставшейся части этой главы. Новые политики безопасности могут не исправить огрехи прошлого. Открытые IP-адреса, скомпрометированные SSL-сертификаты и открытые точки монтирования NFS никуда не исчезнут ни в центре обработки данных Kubernetes, ни в vSphere, и правила безопасности не становятся значительно проще в реализации, просто потому что вы запускаете свои приложения в контейнерах.

13.3.4 Не внедряйте автоматически токен учетной записи сервиса

По умолчанию токен учетной записи сервиса автоматически внедряется в Pod и используется для аутентификации на сервере API. Это плохо! Настолько плохо, что один из авторов книги использовал даже ненормативную лексику, чтобы выразить свое отношение к этому.

Зачем нужен этот токен API? Иногда модулям Pod требуется доступ к серверу API, например, чтобы оператор мог выполнять административные действия с базой данных. Это вполне допустимый вариант использования. Но на самом деле 99,999 % модулей Pod, выполняющих пользовательское программное обеспечение, не нуждаются в доступе к серверу API. Поэтому следует отключить автоматическое внедрение токена учетной записи. Для этого достаточно добавить всего одну строку в определение Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: no-sa
spec:
  automountServiceAccountToken: false ← Запрет автоматического внедрения
```

Другое решение – отключить автоматическое внедрение учетной записи сервиса для всех модулей Pod. Учетная запись сервиса (о которой мы поговорим ниже) тоже поддерживает поле `automountServiceAccountToken`. С его помощью можно настроить любую учетную запись так, чтобы она не подключалась по умолчанию.

13.3.5 Модули Pod с привилегиями root

Учитывая все вышесказанное, настройка модулей Pod превращается в балансирование между привилегированным пользователем, непривилегированным пользователем и непривилегированным пользователем с определенными возможностями. Но зачем все это? Почему бы просто не запустить все модули Pod с разрешениями непривилегированного пользователя? Дело в том, что многие компоненты Kubernetes действуют от имени системного администратора.

Многие компоненты Kubernetes действуют от имени root, и в основном это сетевые модули Pod. Агент kubelet выполняется с привилегиями root, но редко запускается внутри Pod. На каждом узле есть модули CNI, настраивающие сеть кластера, и этим модулям требуются соответствующие привилегии. Мы не можем полностью устранить все риски, но можем уменьшить их с помощью модулей Pod с привилегиями root, включая операторы:

- ограничьте доступ к этим модулям Pod, поместив их в kube-system или другое пространство имен;
- предоставьте им как можно меньше прав root;
- контролируйте их.

13.3.6 Граница безопасности

Kubernetes поддерживает еще три уровня безопасности модулей Pod, которые используют функциональные возможности ядра Linux через модули ядра или другие встроенные функции. В том числе:

- *AppArmor* – профили, работающие на уровне модулей ядра Linux и обеспечивающие контроль на уровне процессов;
- *seccomp* – использование функциональных возможностей ядра Linux защищает процесс, разрешая ему выполнять только определенные вызовы, иначе процесс получает сигнал SIGKILL;
- *SELinux* – Security-Enhanced Linux (Linux с улучшенной безопасностью), еще один модуль ядра Linux, обеспечивающий поддержку политик безопасности, включая обязательный контроль доступа.

Мы лишь кратко затронем эти функции, не вдаваясь в подробности.

Если поддерживаете магазин RHEL, то использование SELinux понятно и оправданно, но, как известно, это доставит немало головной боли авторам. Если вы используете популярную базу данных с открытым исходным кодом или программный компонент с поддерживаемым профилем AppArmor, то, возможно, имеет смысл использовать этот профиль. Технология seccomp предлагает весьма широкие возможности, но ее обслуживание требует немалых усилий. Следует признать, что профили AppArmor и seccomp слишком сложны, а сложности часто приводят к ненадежной безопасности.

В практике часто встречаются ситуации, требующие иного уровня безопасности процессов, но, как и в большинстве случаев, мы стараемся следовать базовым рекомендациям: KISS (*Keep it simple, stupid* – «будь проще, глупыш»), закону убывающей отдачи и правилу 80/20 (прежде чем вы начнете внедрять одну из этих мер, добейтесь 80 % уровня безопасности).

Итоги

- Если не уделять должного внимания безопасности, высока вероятность вторжения злоумышленников в наши кластеры. Безопасность – это набор компромиссов, часто сопряженных с трудными решениями, но, используя простые и базовые методы, можно уменьшить риски.
- Не нужно самостоятельно реализовывать все меры безопасности. Существует постоянно расширяющийся набор инструментов, помогающих исследовать контейнеры и выявлять возможные дыры в безопасности.
- Наиболее очевидной начальной точкой для работы с безопасностью в Kubernetes является уровень контейнера. Владение информацией о происхождении контейнера позволяет отследить источник и убедиться в его надежности.
- Не запускайте контейнеры от имени root, особенно если используются контейнеры, созданные не в вашей организации.
- Для поиска распространенных проблем, которые могут привести к уязвимостям безопасности, используйте линтер (статический анализатор), такой как hadolint.
- Не устанавливайте дополнительное программное обеспечение без явной необходимости. Чем больше дополнительных программ установлено, тем больше возможных уязвимостей.
- Для защиты отдельных модулей Pod следует отключить автоматическое внедрение токена учетной записи сервиса по умолчанию. Кроме того, можно принудительно отключить автоматическое внедрение учетной записи сервиса по умолчанию для всех модулей Pod.
- Дайте процессу минимальные разрешения, используя привилегии CAP.
- Процесс DevOps построен на автоматизации, как и Kubernetes. Автоматизация способствует уменьшению трудозатрат на совершенствование систем.
- Своевременно обновляйте контейнеры и программные зависимости.

14

Безопасность узлов и Kubernetes

В этой главе:

- усиление защиты узлов и объявлений модулей Pod;
- безопасность сервера API, включая RBAC;
- аутентификация и авторизация пользователей;
- Open Policy Agent (OPA);
- режим коллективной аренды в Kubernetes.

Мы только что познакомились с приемами защиты модулей Pod и теперь перейдем к способам защиты узлов Kubernetes. В этой главе мы исследуем дополнительные приемы безопасности, связанные с возможными атаками на узлы и модули Pod, и представим примеры конфигураций.

14.1 Безопасность узла

Защита узлов в Kubernetes подобна защите любых других виртуальных машин или серверов в центре обработки данных. Для начала мы рассмотрим сертификаты безопасности транспортного уровня (Transport Layer Security, TLS), позволяющие защитить узлы, а также проблемы, связанные с неизменностью образов, рабочими нагрузками, сетевыми политиками и т. д. Считайте эту главу своеобразным

аналогом меню, в котором предлагаются возможные варианты обеспечения безопасности, которые следует хотя бы учитывать, запуская Kubernetes в промышленном окружении.

14.1.1 Сертификаты TLS

Все внешние взаимодействия в Kubernetes обычно осуществляются через TLS, хотя этот аспект можно настроить иначе. Однако существует множество разновидностей TLS. Благодаря этому есть возможность выбора наборов механизмов шифрования для использования сервером Kubernetes API. Многие дистрибутивы Kubernetes автоматически создают сертификаты TLS в процессе установки. *Набор алгоритмов шифрования* – это набор алгоритмов, которые в совокупности обеспечивают безопасную работу TLS. Алгоритм TLS предполагает:

- *обмен ключами* – определяет согласованный способ обмена ключами для шифрования/дешифрования;
- *аутентификацию* – подтверждение личности отправителя сообщения;
- *шифрование* – изменение сообщений, чтобы посторонние не могли их прочитать;
- *аутентификацию сообщений* – подтверждение принадлежности сообщений допустимым источникам.

В Kubernetes можно найти следующий набор алгоритмов шифрования: `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384`. Давайте исследуем его. Каждое подчеркивание (`_`) в этой строке отделяет один алгоритм от другого. Например, если выбрать `--tls-cipher-suites` на сервере API, то будет установлен набор:

```
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
```

Описание этого конкретного набора можно найти на <http://mng.bz/nYZ8> и определить, как он работает. Например:

- *протокол* – Transport Layer Security (TLS);
- *обмен ключами* – протокол Диффи–Хеллмана на эллиптических кривых (Elliptic Curve Diffie-Hellman Ephemeral, ECDHE);
- *аутентификация* – алгоритм цифровой подписи на эллиптических кривых (Elliptic Curve Digital Signature Algorithm, ECDSA);
- *шифрование* – расширенный стандарт шифрования с 256-битным ключом в режиме цепочки блоков шифрования (AES 256 CBC).

Обсуждение специфики этих протоколов выходит за рамки нашей книги, но мы хотели бы отметить, что вы должны следить за состоянием безопасности TLS, особенно если в вашей организации оно устанавливается вышестоящим органом по стандартизации. Это позволит вам убедиться, что ваша модель безопасности в Kubernetes соответствует стандартам TLS для вашей организации. Например, чтобы обновить наборы шифров, используемые каким-либо сервисом Kubernetes, передайте ему аргумент `tls-cipher-suites` при запуске:

```
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,  
➔ TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
```

Передача этого аргумента серверу API гарантирует, что он будет подключаться к другим сервисам, используя только этот набор шифров. Как уже говорилось, есть возможность использовать несколько наборов алгоритмов шифрования, перечисляя их через запятую. Полный список наборов можно найти на странице справки для любого сервиса (например, на странице <http://mng.bz/voZq> вы найдете справку для планировщика Kubernetes, kube-scheduler). Также важно отметить, что:

- при обнаружении уязвимости в шифре TLS необходимо обновить наборы шифров на сервере Kubernetes API, в планировщике, диспетчере контроллеров и kubelet. Все эти компоненты так или иначе используют протокол TLS;
- если в вашей организации запрещены определенные наборы шифров, то их следует исключить из списка.

ПРИМЕЧАНИЕ При чрезмерном сужении наборов шифров, разрешенных на сервере API, возникает риск, что некоторые типы клиентов не смогут подключиться к нему. В качестве примера можно привести балансировщики Amazon ELB, которые иногда используют проверки работоспособности HTTPS, чтобы убедиться в доступности конечной точки перед передачей ей трафика, но они не поддерживают некоторые распространенные шифры TLS, используемые на сервере Kubernetes API. Балансировщик нагрузки AWS версии 1 поддерживает только неэллиптические алгоритмы шифрования, такие как TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256. Результат может оказаться плачевным; весь кластер вообще не будет работать! Обычно за одним балансировщиком ELB находится несколько серверов API, поэтому имейте в виду, что проверка работоспособности TCP (а не HTTPS) может оказаться проще в поддержке, особенно если на серверах API требуется использовать специальные шифры безопасности.

14.1.2 Неизменяемые ОС и применение исправлений на узлах

Неизменяемость означает невозможность изменить. Неизменяемая ОС состоит из компонентов и двоичных файлов, доступных только для чтения, которые нельзя исправить или обновить. Но вместо исправления и обновления программного обеспечения можно заменить всю ОС, удалив ее с сервера или из облака, а затем удалив виртуальную машину и создав новую. Kubernetes упрощает работу с неизменяемыми ОС, позволяя администратору перемещать рабочие нагрузки на другие узлы (это встроенная функция).

Вместо системы, автоматически применяющей исправления с помощью диспетчера пакетов, используйте неизменяемую ОС. Наличие

кластера Kubernetes устраняет понятие настраиваемых *серверов-снежинок*. Серверы, на которых выполняются определенные приложения, заменяются стандартными узлами с неизменяемой ОС. Один из самых простых способов внедрить уязвимость в изменяемую систему – заменить какую-нибудь системную библиотеку Linux.

Неизменяемые ОС доступны только для чтения и, соответственно, не позволяют вносить какие-либо изменения, запрещая запись этих изменений на диск, что снижает уязвимость. Использование неизменяемого дистрибутива избавляет от множества проблем. Как правило, узел плоскости управления Kubernetes (сервер API, диспетчер контроллеров и планировщик) будет иметь:

- двоичный файл kubelet;
- двоичный файл kube-proxy;
- двоичный файл среды выполнения containerd или какой-то другой;
- образ etcd.

Все это обеспечивает быстрый запуск кластера. Любой рабочий узел Kubernetes будет иметь те же компоненты, за исключением etcd. Это важное отличие, потому что etcd не поддерживает операционную систему Windows, тогда как некоторые пользователи запускают рабочие узлы с Windows.

Создание пользовательских образов для рабочих узлов с Windows занимает особое место, потому что лицензия на ОС Windows запрещает ее вторичное распространение, и конечные пользователи должны сами создавать образы Windows с агентом kubelet, если хотят использовать неизменяемую модель развертывания. Чтобы узнать больше о неизменяемых образах, посетите домашнюю страницу проекта Tanzu Community Edition (<https://tanzu.vmware.com/tanzu/community>). Его цель – реализовать для широкого использования подход «все включено» к использованию неизменяемых образов вместе с Cluster API для создания кластеров промышленного уровня. Многие другие поставщики услуг Kubernetes, в том числе Google GKE, тоже используют неизменяемые операционные системы.

14.1.3 Изолированные среды выполнения контейнеров

Контейнеры – хорошая штука, но они не могут полностью изолировать процесс от операционной системы. Docker Engine (и другие контейнерные движки) не способны полностью изолировать работающий контейнер от ядра Linux. Между контейнером и хостом нет строгой границы безопасности, поэтому, если в ядре хоста есть уязвимость, то контейнер сможет получить доступ к ней. Для отделения процессов, скажем от прямого доступа к сетевому стеку Linux, Docker Engine использует пространства имен Linux, но этот механизм не перекрывает все обходные пути. Например, файловые системы хоста /sys и /proc по-прежнему доступны процессам в контейнере.

Некоторые проекты, такие как gVisor, IBM Nabra, Amazon Firecracker и Kata, предоставляют виртуальное ядро Linux, отделяющее процесс контейнера от ядра хоста, обеспечивая тем самым настоящую изоляцию. Эти проекты все еще относительно новые, по крайней мере, с точки зрения открытого исходного кода, и пока не получили широкого распространения в среде Kubernetes. Некоторые из них, впрочем, достаточно зрелые. Например, gVisor используется как часть Google Cloud Platform, а Firecracker – как часть платформы Amazon Web Services. Возможно, к тому времени, когда вы будете читать эти строки, выполнение контейнеров в кластерах Kubernetes поверх виртуального ядра получит более широкое распространение! Также можно подумать о развертывании облегченных виртуальных машин в качестве модулей Pod. В такое веселое время мы живем!

14.1.4 Атаки на ресурсы

Узел Kubernetes имеет ограниченное количество ресурсов, таких как процессор, память и диск. В кластере обычно работает несколько модулей Pod, прокси-серверов kube-proxy, агентов kubelets и других процессов Linux. На каждом узле обычно действует провайдер CNI, демон регистрации и другие процессы поддержки кластера. Вы должны убедиться, что контейнеры в модулях не расходуют все ресурсы узла. Если не задать ограничений, то контейнер может создать большую нагрузку на узел и повлиять на остальные системы. По сути, *вышедший из-под контроля процесс-контейнер* может произвести атаку типа «отказ в обслуживании» (DoS) на узле. Обязательно определите ограничения на потребление ресурсов...

Ограничения ресурсов контролируются тремя различными объектами и конфигурациями уровня API. Объекты Pod API могут иметь настройки, управляющие каждым из ограничений. Например, ниже показано, как ограничить потребление процессора, памяти и дискового пространства:

```
apiVersion: v1
kind: Pod
metadata:
  name: core-kube-limited
spec:
  containers:
  - name: app
    image:
    resources:
      requests:
        memory: "42Mi"
        cpu: "42m"
        ephemeral-storage: "21Gi"
      limits:
        memory: "128Mi"
```

Начальный объем ресурсов:
процессор, память или хранилище

Максимально допустимый объем
потребления ресурсов: процессора,
памяти и хранилища

```
cpu: "84m"
ephemeral-storage: "42Gi"
```

Если какое-либо из этих значений будет превышено, то модуль Pod будет остановлен и запущен снова. А если ограничения снова будут превышены, то после повторной остановки модуль Pod больше не будет запускаться.

Еще одна интересная деталь: запросы ресурсов и ограничения влияют также на планирование модулей Pod на узле. Узел, где находится Pod, должен иметь объем свободных ресурсов не меньше запрашиваемого начального объема, чтобы планировщик мог выбрать узел для размещения модуля Pod. Обратите внимание на единицы измерения, которые мы использовали для определения объемов ресурсов – памяти, процессора и временного хранилища.

14.1.5 Единицы измерения потребления процессора

Для измерения потребления процессора в Kubernetes используется базовая единица, соответствующая одному гиперпотoku на «голом железе» или одному ядру / виртуальному процессору в облаке. Разрешается также выражать потребление процессора дробными значениями; например, можно назначить 0,25 единицы. Кроме того, API позволяет преобразовать 0,25 единицы процессора ЦП в 250 m. Все следующие обозначения считаются допустимыми:

```
resources:
  requests:
    cpu: "42"      ← 42 процессора (это очень мощный сервер!)

resources:
  requests:
    cpu: "0.42"    ← Доля 0.42 одного процессора

resources:
  requests:
    cpu: "420m"    ← Та же самая доля 0.42 одного процессора
```

14.1.6 Единицы измерения объема памяти

Объем потребляемой памяти измеряется в байтах, в формате целых или вещественных чисел с использованием следующих суффиксов: E, P, T, G, M и K. Также можно использовать суффиксы Ei, Pi, Ti, Gi, Mi и Ki, представляющие эквивалентные степени двойки. Все следующие разделы определяют примерно одинаковые объемы памяти:

```
resources:
  requests:
    memory: "128974848" ← Объем в байтах (128 974 848 байт)

resources:
```

requests:
memory: "129e6" ← 129e6 иногда записывается как 129e+6
в научной записи: $129e+6 = 129000000$.
Это значение соответствует 129 000 000 байтам

В следующем разделе определяется значение в мегабайтах:

resources:
requests:
memory: "129M" ← 129 мегабайт = 129000000 байт,
что эквивалентно значению 129e+6

А здесь в мебибайтах:

resources:
requests:
memory: "123Mi" ← 123 мебибайт = 128 974 848 байт,
что довольно близко к значению 129e+6

14.1.7 Единицы измерения объема хранилища

Новейшие конфигурационные параметры – объем временного (эффемерного) хранилища, запрашиваемый и максимальный. Ограничения объема временного хранилища применяются к трем типам хранилищ:

- томам emptyDir, кроме tmpfs;
- каталогам с журналами на уровне узла;
- доступным для записи слоям контейнера.

При превышении ограничения kubelet вытесняет Pod. Для каждого узла определяется максимальный объем временного хранилища, что опять же влияет на выбор узлов при планировании модулей Pod. Существует еще одно ограничение, с помощью которого пользователь может задать ограничения для конкретных узлов, называемое *расширенный ресурс*. Более подробную информацию о расширенных ресурсах можно найти в документации Kubernetes.

14.1.8 Сети хостов и модулей Pod

В разделе 14.4 мы поближе рассмотрим сетевые политики, позволяющие заблокировать связь с Pod с помощью провайдера CNI, который обычно реализует эти политики автоматически. Однако есть гораздо более фундаментальный тип сетевой безопасности, который следует учитывать: не запускайте модули Pod в той же сети, что и ваши хосты. В частности:

- ограничьте доступность вашего модуля Pod для внешнего мира;
- ограничьте доступ вашего модуля Pod к сетевым портам хоста.

Подключение модуля Pod сети хоста упрощает доступ из него к узлу и тем самым увеличивает радиус взрыва при атаке. Если Pod не должен работать в сети хоста, то закройте такую возможность! Если Pod должен работать в сети хоста, то закройте к нему доступ из интернета.

Ниже показан фрагмент определения модуля Pod, который работает в сети хоста. Такие модули Pod обычно выполняют административные задачи, такие как журналирование или поддержка сети (провайдеры CNI):

```
apiVersion: v1
kind: Pod
metadata:
  name: host-Pod
spec:
  hostNetwork: true
```

14.1.9 Пример модуля Pod

Мы рассмотрели различные конфигурации модулей Pod: токены учетной записи сервиса, способы ограничения потребления ресурсов, контекст безопасности и т. д. Вот пример определения модуля Pod, содержащий все настройки:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-Pod
spec:
  automountServiceAccountToken: false
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
    capabilities:
      add: ["NET_ADMIN"]
  hostNetwork: true
  volumes:
    - name: sc-vol
      emptyDir: {}
  containers:
    - name: sc-container
      image: my-container
      resources:
        requests:
          memory: "42Mi"
  cpu: "42m"
  ephemeral-storage: "1Gi"
  limits:
    memory: "128Mi"
    cpu: "84m"
  ephemeral-storage: "2Gi"
  volumeMounts:
    - name: sc-vol
      mountPath: /data/foo
  serviceAccountName: network-sa
```

Отключает автоматическое монтирование токена учетной записи сервиса

Настраивает контекст безопасности и включает возможность NET_ADMIN

Работает в сети хоста

Настраивает ограничения ресурсов

Определяет для модуля Pod конкретную учетную запись сервиса

14.2 Безопасность сервера API

Для аутентификации компонентов используются веб-обработчики (webhooks), предоставляемые контроллерами доступа. Контроллеры являются частью сервера Kubernetes API и создают веб-обработчики, которые служат точками входа для событий. Например, ImagePolicy-Webhook – один из плагинов, который позволяет системе реагировать на события и принимать решения о допуске контейнеров. Если модуль Pod не соответствует стандартам допуска, он удерживается в состоянии ожидания – приостанавливается его развертывание в кластере. Контроллеры доступа могут проверять объекты API, создаваемые в кластере, изменять их или делать и то и другое. С точки зрения безопасности такой подход предоставляет огромные возможности контроля и аудита для кластера.

14.2.1 Управление доступом на основе ролей (RBAC)

Прежде всего в кластере необходимо включить управление доступом на основе ролей (Role-Based Access Control, RBAC). В настоящее время RBAC автоматически включается большинством дистрибутивов и облачных провайдеров. Для включения RBAC на сервере Kubernetes API используется флаг `--authorization-mode=RBAC`. Если вы пользуетесь услугами хостинга Kubernetes, такими как GKE, то в них поддержка RBAC уже включена. Авторы уверены, что существуют ситуации, когда использование RBAC не отвечает потребностям бизнеса. Однако в остальных 99 % случаев следует включить RBAC.

RBAC – это механизм безопасности на основе ролей, управляющий доступом пользователей и систем к ресурсам и разрешающий пользоваться ресурсами только авторизованным пользователям и сервисам с учетом их ролей и привилегий. Как это применимо к Kubernetes? Одним из наиболее важных компонентов, который желательно защитить с помощью Kubernetes, является сервер API. Если у системного пользователя есть административный доступ к кластеру через сервер API, он сможет удалять объекты и вызывать иные серьезные нарушения. Администраторы в Kubernetes – это пользователи `root` в контексте кластера.

RBAC – мощный компонент безопасности, обеспечивающий большую гибкость в ограничении доступа к API кластера. Но он также имеет обычный побочный эффект – большую сложность в управлении и отладке.

ПРИМЕЧАНИЕ Типичный модуль Pod, работающий в Kubernetes, не должен иметь доступа к серверу API, поэтому следует запретить внедрение токена учетной записи сервиса.

14.2.2 Определение RBAC API

RBAC API определяет следующие типы:

- *Role* – набор разрешений, ограниченных пространством имен;
- *ClusterRole* – набор разрешений для всего кластера;
- *RoleBinding* – определяет пользователя или группу, которому присваивается указанная роль *Role*;
- *ClusterRoleBinding* – определяет пользователя или группу, которому присваивается указанная роль *ClusterRole*.

Определения *Role* и *ClusterRole* включают несколько компонентов. Первый из них – *verbs*, определяющий глаголы API и HTTP. Объекты на сервере API могут выполнять запросы на получение данных, следовательно, для них определяется глагол *get*. При создании служб REST мы часто строим свои рассуждения, основываясь на глаголах CRUD: *create* (создать), *read* (читать), *update* (обновить) и *delete* (удалить). В определениях ролей можно использовать:

- глаголы для запроса ресурсов: *get* (получить), *list* (перечислить), *create* (создать), *update* (обновить), *patch* (исправить), *watch* (наблюдать), *проху* (проксировать), *redirect* (перенаправить), *delete* (удалить) и *deletecollection* (удалить коллекцию);
- глаголы HTTP-запросов, не связанных с ресурсами: *get* (получить), *post* (отправить), *put* (поместить) и *delete* (удалить).

Например, чтобы оператор мог просматривать и обновлять модули *Pod*, можно:

- 1 определить ресурс (в данном случае *Pod*);
- 2 определить глаголы, к которым данная роль *Role* дает доступ (скорее всего, *list* и *patch*);
- 3 определить группы API (пустая строка обозначает основную группу API).

Вы уже знакомы с группами API, представляющими *apiVersion* и *kind* в определениях Kubernetes. Группам API соответствуют пути REST на самом сервере API (*/apis/\$GROUP_NAME/\$VERSION*) и используют *apiVersion \$GROUP_NAME/\$VERSION* (например, *batch/v1*). Однако давайте пока не будем усложнять; оставим в стороне группы API и начнем с основной группы API. Вот пример определения роли для конкретного пространства имен. Роли ограничиваются пространствами имен, что обеспечивает доступ для применения глаголов *list* и *patch* к ресурсу *Pod*:

```
# Определение роли в пространстве имен по умолчанию, дающей право
# применять глаголы list и patch к определениям модулей Pod
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
```

```

name: Pod-labeler
namespace: rbac-example
rules:
- apiGroups: [""]          # "" означает основную группу API
  resources: ["Pods"]      # ресурс
  verbs: ["list", "patch"] # разрешение применять глаголы list и patch

```

Предыдущую роль можно использовать в определении учетной записи сервиса, как показано ниже:

```

# Определение ServiceAccount, которая будет присваивать роль, описанную выше
apiVersion: v1
kind: ServiceAccount
metadata:
  name: Pod-labeler
  namespace: rbac-example

```

Предыдущее определение создает учетную запись сервиса, которую может использовать модуль Pod. Далее мы создадим привязку роли RoleBinding, чтобы подключить предыдущую учетную запись сервиса с ролью, которая определена выше:

```

# Связывает ServiceAccount с именем Pod-labeler и роль Role с именем Pod-labeler
# Любой Pod, использующий ServiceAccount с именем Pod-labeler, получает права
# доступа к API, определяемые ролью Role с именем Pod-labeler.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Pod-labeler
  namespace: rbac-example
subjects:
  # Список учетных записей сервисов для привязки
- kind: ServiceAccount
  name: Pod-labeler
roleRef:
  # Присваиваемая роль
  kind: Role
  name: Pod-labeler
  apiGroup: rbac.authorization.k8s.io

```

Теперь можно запустить модуль Pod в разворачивании Deployment, которому назначена учетная запись сервиса для этого модуля Pod:

```

# Разворачивает единственный Pod для выполнения кода с правами Pod-labeler
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Pod-labeler
  namespace: rbac-example
spec:
  replicas: 1

  # Может управлять любыми модулями Pod с меткой app=Pod-labeler

```



```

selector:
  matchLabels:
    app: Pod-labeler

template:
  # Гарантировать создание модулей Pod с меткой app=Pod-labeler,
  # соответствующей селектору развертывания
  metadata:
    labels:
      app: Pod-labeler

спес:
  # определить учетную запись сервиса для Pod
  serviceAccount: Pod-labeler

  # Еще одна мера безопасности: задать UID и GID в контексте безопасности,
  # которые будут присваиваться по умолчанию всем процессам контейнеров.
  # Здесь используются идентификаторы 9999, потому что они соответствуют
  # непривилегированному пользователю и группе и (по крайней мере, в
  # нашем случае) не пересекаются с идентификаторами пользователей и групп
  # на узле.
  securityContext:
    runAsUser: 9999
    runAsGroup: 9999
    fsGroup: 9999

containers:
  - image: gcr.io/pso-examples/Pod-labeler:0.1.5
    name: Pod-labeler

```

Подведем итоги. Мы создали роль, разрешающую получать список модулей Pod и исправлять их определения. Затем мы создали учетную запись сервиса, чтобы получить возможность создать Pod и заставить этот Pod использовать определенного пользователя. Затем определили привязку роли, чтобы связать роль с учетной записью сервиса. Наконец, мы запустили развертывание с модулем Pod, который использует учетную запись сервиса, определенную выше.

Управление доступом на основе ролей – непростая задача, но жизненно важная для безопасности кластера Kubernetes. Предыдущие определения были взяты из демонстрационного примера Helmsman RBAC, доступного по адресу <http://mng.bz/ZzMa>.

14.2.3 Ресурсы и подресурсы

Большинство ресурсов RBAC использует только одно имя, например Pod или Deployment. У некоторых ресурсов есть подресурсы, например:

```
GET /api/v1/namespaces/rbac-example/Pods/Pod-labeler/log
```

Здесь определяется путь API к подресурсу `log` в пространстве имен `rbac-example` для модуля Pod с именем `Pod-labeler`. В общем виде путь определяется так:

```
GET /api/v1/namespaces/{имя_пространства_имен}/Pods/{имя}/log
```

Чтобы использовать подресурс log, необходимо определить роль, например:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-and-Pod-logs-reader
rules:
- apiGroups: [""]
  resources: ["Pods", "Pods/log"]
  verbs: ["get", "list"]
```

Также есть возможность дополнительно ограничить доступ к журналам модулей Pod, манипулируя их именами. Например:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-labeler-logs
rules:
- apiGroups: [""]
  resourceNames: ["Pod-labeler"]
  resources: ["Pods/log"]
  verbs: ["get"]
```

Обратите внимание, что элемент rules в предыдущем примере – это массив. Ниже показано, как можно задать несколько разрешений. Элементы resources, resourceNames и verbs могут комбинироваться как угодно:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-labeler-logs
rules:
- apiGroups: [""]
  resourceNames: ["Pod-labeler"]
  resources: ["Pods/log"]
  verbs: ["get"]
- apiGroups: [""]
  resourceNames: ["another-Pod"]
  resources: ["Pods/log"]
  verbs: ["get"]
```

Ресурсы – это такие объекты, как Pod и Node, но сервер API включает также объекты, не являющиеся ресурсами. Они определяются фак-

тическим компонентом URI конечной точки REST API; например, вот как можно определить роль Pod-labeler-logs для доступа к конечной точке API /healthz:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-labeler-logs
rules:
- apiGroups: [""]
  resourceNames: ["Pod-labeler"]
  resources: ["Pods/log"]
  verbs: ["get"]
- apiGroups: [""]
  resourceNames: ["another-Pod"]
  resources: ["Pods/log"]
  verbs: ["get"]
- nonResourceURLs: ["/healthz", "/healthz/*"]
  verbs: ["get", "post"]
```

Звездочка (*) в URL объекта, не являющегося ресурсом, соответствует любому суффиксу

14.2.4 Субъекты и RBAC

Привязки ролей могут охватывать объекты User, ServiceAccount и Group в Kubernetes. Следующий пример определяет еще одну учетную запись сервиса с именем log-reader и добавляет ее в определение привязки роли из предыдущего раздела. В этом примере также определяется пользователь с именем james-bond и группа с именем MI-6:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Pod-labeler
  namespace: rbac-example
subjects:
  # Список учетных записей сервисов для привязки
- kind: ServiceAccount
  name: Pod-labeler
- kind: ServiceAccount
  name: log-reader
- kind: User
  name: james-bond
- kind: Group
  name: MI-6
roleRef:
  # Назначаемая роль
  kind: Role
  name: Pod-labeler
  apiGroup: rbac.authorization.k8s.io
```

ПРИМЕЧАНИЕ Пользователи и группы создаются стратегией аутентификации, настроенной в кластере.

14.2.5 Отладка RBAC

Как уже отмечалось, управление доступом на основе ролей – непростая задача и доставляет немало неудобств, но у нас всегда есть журнал аудита. Kubernetes регистрирует в журнале аудита, если эта функция включена, события безопасности, влияющие на кластер. К этим событиям относятся действия пользователя, действия администратора и/или другие компоненты внутри кластера. По сути, используя RBAC и другие компоненты системы безопасности, вы получаете полную информацию о том, «кто», «что», «откуда» и «как». Журнал аудита настраивается с помощью файла политики аудита, который передается на сервер API через `kube-apiserver --audit-policy-file`.

Итак, у нас есть журнал всех событий – круто! Но постоит-ка... теперь у нас есть кластер с сотнями ролей, массой пользователей и множеством привязок ролей. Теперь нужно объединить все эти данные вместе. Помочь в этом нам может несколько инструментов. Общей особенностью всех этих инструментов является объединение различных объектов, используемых в RBAC. Чтобы проанализировать разрешения RBAC в ролях кластера, необходимо объединить привязки ролей RoleBinding и субъекты. Субъектами могут быть пользователи, группы или учетные записи сервисов:

- компания *ReactiveOps* создала инструмент, позволяющий отыскивать текущие роли, назначенные пользователю, группе или учетной записи сервиса. `rbac-lookup` доступен по адресу <https://github.com/reactiveops/rbac-lookup>;
- определить, какие разрешения имеет пользователь или учетная запись сервиса в кластере, можно с помощью `kubectrl-rbac`. Этот инструмент доступен по адресу <https://github.com/octarine/sec/kubectrl-rbac>;
- Джордан Лиггит (*Jordan Liggitt*) создал инструмент с открытым исходным кодом `audit2rbac`. Он принимает журналы аудита и имя пользователя и создает роль и привязку роли, соответствующие выполненным запросам. Все вызовы, которые выполняются на сервере API, фиксируются в журналах, и по этим данным `audit2rbac` может воссоздать соответствующие определения RBAC (другими словами, произвести реинжиниринг).

14.3 Authn, Authz и Secret

Authn (user authentication – аутентификация пользователя) и *Authz* (authorization – авторизация) – это группа и разрешения аутентифицированного пользователя. Может показаться странным, что в на-

звании раздела упомянуты также объекты Secret, но дело в том, что при работе с секретами Secrets используются тех же инструменты, что применяются для аутентификации и авторизации, и для доступа к секретам часто необходимо выполнить аутентификацию и авторизацию.

Сразу хотим предупредить: не используйте административные сертификаты кластера по умолчанию, которые генерируются в ходе установки кластера. Используйте службу IAM (Identity and Access Management – управление идентификацией и доступом) для аутентификации и авторизации пользователей. Также не включайте на сервере API аутентификацию по имени пользователя и паролю; используйте встроенную возможность аутентификации с помощью сертификатов TLS.

14.3.1 Учетные записи сервисов IAM: защита облачных API

Контейнеры Kubernetes имеют собственные *облачные удостоверения*. Это и благо, и наказание одновременно. Без модели угроз для облака не может быть модели угроз для кластера Kubernetes.

Учетные записи облачных сервисов IAM являются основой безопасности, обеспечивая авторизацию и аутентификацию людей и систем. В центре обработки данных конфигурация безопасности Kubernetes ограничивается системами Linux, Kubernetes, сетями и развернутыми контейнерами. При запуске Kubernetes в облаке возникает новая проблема – IAM-роли узлов и модулей Pod:

- IAM – это роль для определенного пользователя или учетной записи сервиса, и тогда эта роль является членом группы;
- каждому узлу в кластере Kubernetes назначается роль IAM;
- модули Pod обычно наследуют эту роль.

Узлам в кластере, особенно работающим в плоскости управления, роли IAM нужны для работы в облаке, потому что многие облачные функции в Kubernetes учитывают тот факт, что сам Kubernetes знает, как общаться со своим облачным провайдером. В качестве примера возьмем пример из официальной документации GKE: Google Cloud Platform автоматически создает учетную запись сервиса, называемую учетной записью сервиса по умолчанию для вычислительного движка, и GKE связывает ее с создаваемыми узлами. В зависимости от настроек проекта, учетная запись сервиса по умолчанию может иметь или не иметь разрешения на использование других API облачной платформы. GKE также назначает некоторые ограниченные права доступа для вычислительных экземпляров. Обновление разрешений учетной записи сервиса по умолчанию или дополнительных прав доступа для вычислительных экземпляров не является рекомендуемым способом аутентификации в других сервисах облачной платформы из модулей Pod, работающих в GKE.

Таким образом, во многих случаях ваши контейнеры получают привилегии наравне с узлами. В будущем с развитием облачных тех-

нологий и появлением более детализированных моделей разрешений для контейнеров это решение по умолчанию, вероятно, будет усовершенствовано. Однако и в этом случае необходимо убедиться, что роли IAM имеют минимальные разрешения, и всегда есть возможность изменить эти роли IAM. Например, при использовании GKE в Google Cloud Platform необходимо создать новую роль IAM в проекте кластера. Если этого не сделать, кластер будет использовать учетную запись сервиса вычислительного движка по умолчанию, которая имеет разрешения редактора.

Разрешения редактора в *Google Cloud* позволяют учетной записи (в данном случае узлу в кластере, что потенциально означает любой модуль Pod) редактировать любой ресурс в проекте. Например, злоумышленник сможет удалить целый парк баз данных, балансировщиков нагрузки TCP или облачных записей DNS, просто скомпрометировав некоторый модуль Pod в кластере. Кроме того, необходимо удалить учетную запись сервиса по умолчанию для любого проекта, который создается в GCP. Те же проблемы существуют в AWS, Azure и других облачных окружениях. Суть в том, что каждый кластер создается с собственной уникальной учетной записью сервиса, и эта учетная запись имеет наименьшие возможные разрешения. С помощью таких инструментов, как kops (Kubernetes Operations), можно получить все разрешения, которые требуются кластеру Kubernetes, а затем создать одну роль IAM для плоскости управления и другую – для узлов.

14.3.2 Доступ к облачным ресурсам

Представьте, что вы настроили свои узлы Kubernetes с самым минимумом необходимых разрешений и теперь, прочитав все это, чувствуете себя в безопасности. На самом деле, используя такое решение, как AKS (Azure Kubernetes Service – сервис Azure Kubernetes), не нужно беспокоиться о настройке плоскости управления, достаточно позаботиться только об IAM на уровне узла, но это еще не все. Например, разработчик создал сервис, который должен взаимодействовать с облачным сервисом, например с хранилищем файлов. Работающему модулю Pod теперь нужна учетная запись сервиса с правильными ролями. Существуют различные способы решения этой задачи.

ПРИМЕЧАНИЕ Сервис AKS является, пожалуй, самым простым решением, но оно создает некоторые проблемы. Вы должны ограничить количество модулей Pod на узлах, оставив только те, которым нужен доступ к облачным ресурсам, или принять риск доступности хранилища файлов для всех модулей Pod.

СОВЕТ Для реализации этого подхода можно использовать такие инструменты, как kube2iam (<https://github.com/jtblin/kube2iam>) и kiam (<https://github.com/uswitch/kiam>).

Некоторые вновь созданные операторы могут назначать определенные учетные записи сервисов отдельным модулям Pod. Компонент на каждом узле перехватывает обращения к облачному API и вместо роли IAM узла назначает модулю Pod роль, определенную в кластере, используя аннотации. Некоторые поставщики облачных услуг предлагают аналогичные решения. Например, Google предлагает дополнительные компоненты (контейнеры-прицепы – sidecar), которые могут запускаться и подключаться к облачной службе SQL. Им назначаются определенные роли, после чего они проксируют запросы приложений к базе данных.

Вероятно, наиболее сложным, но более надежным решением является использование централизованного сервера хранилища. Используя его, можно заставить приложения извлекать временные токены IAM, разрешающие доступ к облачной системе. Часто для автоматизации обновления токенов используется контейнер-прицеп (sidecar). Также можно использовать HashiCorp Vault для защиты секретов, которые не являются учетными данными IAM. Если ваш вариант использования требует надежного управления секретами и IAM, то Vault – отличное решение, но, как и в случае со всеми критически важными решениями, вам придется сопровождать и поддерживать его.

СОВЕТ Используйте HashiCorp Vault (<https://www.vaultproject.io/>) для хранения секретов.

14.3.3 Частные серверы API

Последний вопрос, который мы рассмотрим в этом разделе, – сетевой доступ к серверу API. Вы можете сделать свой сервер API недоступным из интернета или разместить его в частной сети. Однако, разместив балансировщик нагрузки сервера API в частной сети, вам придется использовать промежуточный шлюз, VPN или другую форму подключения к серверу API, поэтому это решение не особенно удобно.

Безопасность сервера API чрезвычайно важна, и он должен иметь соответствующую защиту. DoS-атаки или вторжение могут вывести кластер из строя. Более того, многие проблемы безопасности, обнаруживаемые сообществом Kubernetes, возникают на сервере API. Если есть такая возможность, то поместите свой сервер API в частную сеть или, по крайней мере, реализуйте белый список IP-адресов, которые могут подключаться к балансировщику нагрузки, находящемуся перед сервером API.

14.4 Безопасность сети

К этой сфере безопасности редко относятся с должным вниманием. По умолчанию любой Pod в сети модулей Pod может получить доступ к любому модулю Pod в любом месте кластера, включая и сервер API.

Это позволяет модулям Pod обращаться к таким системам, как DNS, для поиска сервисов. Pod, работающий в сети хоста, может получить доступ практически ко всему: ко всем модулям Pod, ко всем узлам и к серверу API. Pod в сети хоста может даже получить доступ к порту API агента kubelet, если он включен.

Сетевые политики – это объекты, которые определяются для управления сетевым трафиком между модулями Pod. Объекты NetworkPolicy позволяют настраивать доступ к входящему и исходящему трафику модуля Pod. *Входящий трафик* – это трафик, поступающий в Pod, а *исходящий* – это трафик, исходящий из Pod.

14.4.1 Сетевые политики

Создать объект NetworkPolicy можно в любом кластере Kubernetes, но для его поддержки нужен работающий провайдер безопасности, такой как Calico. Calico – провайдер CNI, который также предоставляет отдельное приложение для поддержки сетевых политик. Если создать сетевую политику без провайдера, то она не будет иметь никакого влияния. Сетевые политики имеют следующие ограничения и функции:

- применяются к модулям Pod;
- соответствие конкретным модулям Pod выявляется с помощью селекторов;
- осуществляют контроль входящего и исходящего сетевого трафика;
- управляют сетевым трафиком, определяемым диапазоном CIDR, конкретным пространством имен или соответствующими модулями Pod;
- предназначены для обработки трафика TCP, UDP и SCTP;
- поддерживают обработку конкретных портов или модулей Pod.

Давайте попробуем реализовать их. Чтобы настроить кластер kind и установить в него Calico, выполните следующую команду, но не запускайте CNI по умолчанию, потому что далее мы установим Calico:

```
$ cat <<EOF | kind create cluster --config -
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  # запретить установку CNI по умолчанию
  disableDefaultCNI: true
  PodSubnet: "192.168.0.0/16"
EOF
```

Далее установите оператор Calico и его ресурсы, выполнив следующие команды:

```
$ kubectl create -f \
https://docs.projectcalico.org/manifests/tigera-operator.yaml
```



```
$ kubectl create -f \
https://docs.projectcalico.org/manifests/custom-resources.yaml
```

Теперь можно понаблюдать за запуском Pod. Выполните следующую команду:

```
$ kubectl get Pods --watch -n calico-system
```

Затем настройте пару пространств имен, установите сервер NGINX для обслуживания тестовой веб-страницы и контейнер BusyBox, в котором можно запустить команду `wget`:

```
$ kubectl create ns web
$ kubectl create ns test-bed
$ kubectl create deployment -n web nginx --image=nginx
$ kubectl expose -n web deployment nginx --port=80
$ kubectl run --namespace=test-bed testing --rm -ti --image busybox /bin/sh
```

В командной строке, в контейнере BusyBox, отправьте запрос серверу NGINX, запущенному в пространстве имен `web`:

```
$ wget -q nginx.web -O
```

Задайте сетевую политику, отвергающую весь входящий трафик к NGINX:

```
$ kubectl create -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-ingress
  namespace: web
spec:
  PodSelector:
    matchLabels: {}
  policyTypes:
  - Ingress
EOF
```

Эта команда создаст политику, запрещающую доступ к веб-странице NGINX из тестового модуля Pod. Запустите следующую команду в тестовом модуле. Когда истечет время ожидания, она завершится с ошибкой:

```
$ wget -q --timeout=5 nginx.web.svc.cluster.local -O -
```

Затем разрешите трафик из пространства имен `test-bed` в пространство имен `web`:

```
$ kubectl create -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

```

metadata:
  name: access-web
  namespace: web
spec:
  PodSelector:
    matchLabels:
      app: nginx
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              name: test-bed
EOF

```

В командной строке в тестовом модуле введите:

```
$ wget -q --timeout=5 nginx.web.svc.cluster.local -O -
```

Команда снова потерпит неудачу. Причина в том, что сетевые политики определяют соответствие по меткам, а пространство имен `test-bed` не имеет метки. Добавьте метку:

```
$ kubectl label namespaces test-bed name=test-bed
```

Теперь в командной строке в тестовом модуле проверьте, что политика сети заработала:

```
$ wget -q --timeout=5 nginx.web.svc.cluster.local -O -
```

Первая рекомендация, которую можно дать настраивающим брандмауэр: создать правило, запрещающее все. Следующая политика запрещает любой трафик, входящий в пространство имен `test-bed` или исходящий из него:

```

$ kubectl create -f - <<EOF
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-all
  namespace: test-bed
spec:
  PodSelector:
    matchLabels: {}
  policyTypes:
    - Ingress
    - Egress
EOF

```

Соответствует всем модулям Pod в пространстве имен

Определяет два типа политик: для входного (Ingress) и выходного (Egress) трафика

Реализация этой политики вызывает некоторые забавные побочные эффекты. Модули Pod не только не смогут взаимодействовать ни

с какими другими компонентами (кроме своего пространства имен), но и с провайдером DNS в пространстве имен kube-system. Если Pod не нуждается в поддержке DNS, то и не включайте ее! Давайте применим следующую сетевую политику, чтобы разрешить доступ к DNS:

```
$ kubectl label namespaces kube-system name=kube-system
$ kubectl create -f - <<EOF
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: dns-egress
  namespace: test-bed
spec:
  PodSelector:
    matchLabels: {}
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              name: kube-system
      ports:
        - protocol: UDP
          port: 53
EOF
```

Если теперь запустить команду `wget`, то вы увидите, что она по-прежнему не работает. Мы разрешили входящий трафик в пространство имен `web`, но не разрешили исходящий трафик из пространства имен `test-bed`, направляющийся в пространство имен `web`. Выполните следующую команду, чтобы разрешить передачу трафика из `test-bed` в `web`:

```
$ kubectl label namespaces web name=web
$ kubectl create -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-bed-egress
  namespace: test-bed
spec:
  PodSelector:
    matchLabels: {}
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              name: web
EOF
```

Вы, наверное, заметили, что в NetworkPolicy можно определять весьма сложные правила. Если кластер работает в окружении с *высоким уровнем доверия*, то реализация сетевых политик может не улучшить безопасность. Используйте правило 80/20, отложите определение сетевых политик NetworkPolicy, если ваша организация не обновляет образы. Да, сетевые политики сложны, и частично поэтому использование сервисной сетки может повысить вашу безопасность.

СЕРВИСНАЯ СЕТКА

Сервисная сетка (service mesh) – это приложение, работающее поверх кластера Kubernetes и предоставляющее различные возможности, улучшающие наблюдаемость и надежность. В числе популярных сервисных сеток можно назвать Istio, Linkerd, Consul и др. Мы упоминаем сервисные сетки в главе о безопасности просто потому, что они могут помочь в двух ключевых аспектах: поддержке TLS и расширенных политиках управления сетевым трафиком. Мы коснемся этой темы очень кратко, поскольку ей посвящены целые книги.

Сервисная сетка добавляет сложный слой поверх практически каждого приложения, запускаемого в кластере, а также предоставляет множество дополнительных компонентов безопасности. И снова проанализируйте оправданность добавления сервисной сетки, не начинайте с ее внедрения в первый же день. Чтобы узнать, соответствует ли ваш кластер спецификации CNCF для NetworkPolicy API, можно запустить наборы тестов NetworkPolicy с помощью Sonobuoy (о котором мы рассказывали в предыдущих главах):

```
$ sonobuoy run --e2e-focus=NetworkPolicy
# подождать примерно 30 минут
$ sonobuoy status
```

Эти команды выведут серию таблиц с результатами тестов, которые точно показывают, как работают сетевые политики в вашем кластере. Чтобы узнать больше о концепциях соответствия NetworkPolicy API для поставщиков CNI, посетите страницу <http://mng.bz/XW7M>. Мы настоятельно рекомендуем запускать тесты для проверки соответствия NetworkPolicy при оценке вашего провайдера CNI на совместимость со спецификациями сетевой безопасности Kubernetes.

14.4.2 Балансировщики нагрузки

Важно помнить, что Kubernetes может создавать внешние балансировщики нагрузки, открывающие доступ к вашим приложениям извне, и делает это автоматически. Это общеизвестный факт, поэтому имейте в виду, что размещение неправильной службы в промышленном окружении может привести к тому, что сервис (например, административный пользовательский интерфейс) окажется открытым для посторонних. Используйте инструменты в процессе непрерывной интеграций (CI), такие как Open Policy Agent (OPA), чтобы исключить

непреднамеренное создание внешних балансировщиков нагрузки. Кроме того, по возможности используйте внутренние балансировщики нагрузки.

14.4.3 Агент открытой политики (OPA)

Выше упоминалось, что операторы могут помочь организации дополнительно защитить кластер. Агент открытой политики (Open Policy Agent, OPA) – проект CNCF – позволяет определять декларативные политики, используемые контроллерами доступа.

OPA – это облегченный механизм политик общего назначения, который можно разместить вместе с вашим сервисом. OPA можно интегрировать как вспомогательный компонент на уровне хоста или библиотеки.

Сервисы передают право принятия решений о политике агенту OPA, выполняя запросы. OPA оценивает политики и данные применительно к результатам запросов (которые отправляются обратно клиенту). Политики определяются на декларативном языке высокого уровня и могут загружаться в OPA через файловую систему или четко определенные API.

– Open Policy Agent
(<http://mng.bz/RE6O>)

OPA поддерживает два разных компонента: контроллер доступа OPA и OPA Gatekeeper. Gatekeeper не использует контейнер-прицеп (sidcar) для CRD (настраиваемые определения ресурсов), поддерживает возможность расширения и выполняет функции аудита. В следующем разделе рассматривается установка Gatekeeper в кластере Kubernetes.

Установка OPA

Для начала остановите прежний кластер, действующий под управлением Calico, а затем запустите новый:

```
$ kind delete cluster
$ kind create cluster
```

Далее, установите OPA Gatekeeper:

```
$ kubectl apply -f \
https://raw.githubusercontent.com/open-policy-agent/gatekeeper/v3.7.0/
➔ deploy/gatekeeper.yaml
```

Следующая команда выведет список установленных модулей Pods:

```
$ kubectl -n gatekeeper-system get po
```

NAME	READY	STATUS	RESTARTS	AGE
gatekeeper-audit-7d99d9d87d-rb4qh	1/1	Running	0	40s

gatekeeper-controller-manager-f94cc7dfc-j6zjv	1/1	Running	0	39s
gatekeeper-controller-manager-f94cc7dfc-mxz6d	1/1	Running	0	39s
gatekeeper-controller-manager-f94cc7dfc-rvqvj	1/1	Running	0	39s

ПРИМЕЧАНИЕ Установить OPA Gatekeeper также можно с помощью Helm.

ОПРЕДЕЛЕНИЯ РЕСУРСОВ (CRD) для GATEKEEPER

Одна из сложных сторон OPA – изучение нового языка (Rego) описания политик. Более полную информацию о языке Rego можно найти по адресу <http://mng.bz/2jdm>. С помощью Gatekeeper политики, написанные на Rego, помещаются в поддерживаемые CRD. Чтобы добавить политику, нужно создать два разных CRD:

- шаблон ограничения, определяющий политику и ее цели;
- ограничение для включения шаблона и определения того, как должна включаться политика.

Ниже приводится пример шаблона и связанного с ним ограничения. В этом примере определяются два CRD. Раздел `match` поддерживает:

- `kinds` – определяет объекты Kubernetes API;
- `namespaces` – определяет список пространств имен;
- `excludeNamespaces` – определяет список пространств имен, на которые не распространяются ограничения;
- `scope` – *, кластер или пространство имен;
- `labelSelector` – стандартный селектор меток Kubernetes;
- `namespaceSelector` – стандартный селектор пространства имен Kubernetes.

```
apiVersion: templates.gatekeeper.sh/v1beta1
```

```
kind: ConstraintTemplate
```

```
metadata:
```

```
  name: enforcespecificcontainerregistry
```

```
spec:
```

```
  crd:
```

```
    spec:
```

```
      names:
```

```
        kind: EnforceSpecificContainerRegistry
```

```
        # Схема для поля 'parameters'
```

```
        openAPIV3Schema:
```

```
          properties:
```

```
            repos:
```

```
              type: array
```

```
              items:
```

```
                type: string
```

```
targets:
```

```
  - target: admission.k8s.gatekeeper.sh
```

```
    rego: |
```

Определяет
EnforceSpecificContainerRegistry –
CRD с ограничениями

```

package enforcespecificcontainerregistry

violation[{"msg": msg}] {
    container := input.review.object.spec.containers[_]
    satisfied := [good | repo = input.parameters.repos[_] ;
➔ good = startswith(container.image, repo)]
    not any(satisfied)
    msg := sprintf("container '%v' has an invalid image repo
➔ '%v', allowed repos are %v",
➔ [container.name, container.image, input.parameters.repos])
}

violation[{"msg": msg}] {
    container := input.review.object.spec.initContainers[_]
    satisfied := [good | repo = input.parameters.repos[_] ;
➔ good = startswith(container.image, repo)]
    not any(satisfied)
    msg := sprintf("container '%v' has an invalid image repo '%v',
➔ allowed repos are %v",
➔ [container.name, container.image, input.parameters.repos])
}

---
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: EnforceSpecificContainerRegistry
metadata:
  name: enforcespecificcontainerregistrytestns
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "test-ns"
  parameters:
    repos:
      - "myrepo.com"

```

Теперь сохраните этот код в двух файлах YAML (в один сохраните шаблон, а во второй – ограничения). Сначала установите в кластер файл с шаблоном, а затем с ограничением. (Мы не будем показывать команду установки для краткости.) Теперь можно применить политику, выполнив следующие команды:

```

$ kubectl create ns test-ns
$ kubectl create deployment -n test-ns nginx --image=nginx

```

Для проверки развертывания выполните следующую команду (ожидается, что Pod не запустится):

```

$ kubectl get -n test-ns deployments.apps
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
nginx     0/1     0             0           37s

```

Команда `kubectl -n test-ns get Pods` показывает, что ни один из модулей Pod не запустился, а в журнале можно найти сообщение о сбое создания модуля Pod:

```
$ kubectl -n test-ns get events
7s          Warning   FailedCreate      replicaset/nginx-6799fc88d8
➔ Error creating: admission webhook "validation.gatekeeper.sh"
➔ denied the request: [denied by
➔ enforcespecificcontainerregistrytestns] container <nginx>
➔ has an invalid image repo <nginx>, allowed repos are
➔ ["myrepo.com"]
```

14.4.4 Коллективная аренда

Для классификации вида коллективной аренды (multi-tenancy) нужно оценить уровень доверия между арендаторами, а затем разработать соответствующую модель. Существует три основные категории или модели безопасности, которые можно отнести к коллективной аренде:

- *высокое доверие (все арендаторы находятся в одной компании)* – разные отделы одной компании выполняют вычисления в одном кластере;
- *доверие от среднего до низкого (арендаторы находятся в разных компаниях)* – внешние клиенты запускают приложения в вашем кластере в разных пространствах имен;
- *нулевое доверие (обработка данных регулируется законами)* – различные приложения используют данные, обработка которых регулируется законами, поэтому предоставление доступа к разным хранилищам данных может привести к судебному иску против вашей компании.

Сообщество Kubernetes много лет работает над решением этих вариантов использования. Джесси Фразелье (Jessie Frazelle) прекрасно резюмирует это в своей статье «Hard Multi-Tenancy in Kubernetes»:

Модели коллективной аренды подробно обсуждались в рабочей группе сообщества... Также было представлено несколько предложений по реализации каждой модели. Текущая модель аренды в Kubernetes предполагает, что границей безопасности является кластер. Вы можете создать SaaS поверх Kubernetes, но для этого нужно использовать свой доверенный API, а не просто использовать Kubernetes API. Конечно, при этом необходимо подумать о множестве мелочей, чтобы обезопасить кластер даже для SaaS.

— Джесси Фразелье (Jessie Frazelle), <http://mng.bz/1jdn>

Kubernetes API создавался без учета наличия нескольких изолированных клиентов в одном кластере. Docker Engine и другие среды выполнения контейнеров тоже имеют проблемы с запуском вредоносных или ненадежных рабочих нагрузок. Программные компоненты, такие как gVisor, добились больших успехов в этом направлении, но

на момент написания этой книги пока не имелось решений, позволяющих запускать контейнеры с нулевым уровнем доверия.

Итак, что мы имеем? Сотрудник службы безопасности сказал бы, что все зависит от доверия и модели безопасности. Выше мы перечислили три модели безопасности: высокое доверие (в одной компании), низкое доверие или его отсутствие (разные компании) и нулевое доверие (обработка данных регулируется законами). Kubernetes поддерживает коллективную аренду с высоким уровнем доверия и, в зависимости от модели, способен поддерживать модели с высоким и низким уровнем доверия. При нулевом или низком доверии между арендаторами необходимо использовать отдельные кластеры для каждого клиента. Некоторые компании управляют сотнями кластеров, соответственно, каждая небольшая группа приложений получает свой отдельный кластер, но это влечет необходимость управлять большим количеством кластеров.

Иногда, даже если клиенты принадлежат одной компании, может потребоваться изолировать модули Pod на определенных узлах из-за высокой конфиденциальности данных. Благодаря RBAC, пространствам имен, сетевым политикам и изоляции узлов можно добиться достойного уровня изоляции. Следует признать, что существует риск размещения рабочих нагрузок разных компаний в одном и том же кластере Kubernetes. Поддержка коллективной аренды со временем будет расширяться.

ПРИМЕЧАНИЕ Идеи коллективной аренды применимы также к другим окружениям, например окружениям разработки или тестирования в промышленном кластере. Однако есть возможность внести в кластер неверный код субъекта, что приведет к смешиванию окружений.

Размещение нескольких клиентов в одном кластере Kubernetes сопряжено с двумя основными проблемами: обеспечением безопасности сервера API и узла. Почему после настройки аутентификации, авторизации и RBAC может возникнуть проблема с безопасностью сервера API? Одна из причин связана с URI сервера API. В системах с несколькими клиентами распространенной практикой является использование одного и того же сервера API, а также идентификатора пользователя, идентификатора проекта или некоторого уникального идентификатора, с которого начинается URI.

Наличие URI, начинающегося с уникального идентификатора, позволяет арендатору выполнить вызов для получения всех пространств имен. Поскольку в Kubernetes отсутствует полная изоляция, для этого достаточно запустить команду `kubectl get namespaces`. Для обеспечения должной изоляции также вам также придется настроить слой API поверх Kubernetes API.

Еще одно решение, позволяющее использовать кластер несколькими арендаторами, – вложение ресурсов, а основной границей ре-

сурсов в Kubernetes являются пространства имен. Пространства имен Kubernetes не допускают вложенности. Многие ресурсы могут пересекать границы пространства имен, в том числе токен учетной записи сервиса по умолчанию. Часто арендаторы сами хотят использовать расширенные возможности RBAC, и предоставление им разрешений на создание объектов RBAC в Kubernetes может дать возможности выхода за рамки их аренды.

Проблема безопасности узлов заключена внутри. Если в одном кластере Kubernetes есть несколько арендаторов, то все они совместно используют следующие элементы (это лишь краткий список):

- плоскость управления и сервер API;
- надстройки, такие как DNS-сервер, средства журналирования или создания сертификатов TLS;
- пользовательские определения ресурсов (CRD);
- сети;
- ресурсы хоста.

ОБЗОР СЛУЧАЯ КОЛЛЕКТИВНОЙ АРЕНДЫ С ВЫСОКИМ УРОВНЕМ ДОВЕРИЯ

Многим компаниям нужна поддержка коллективной аренды, чтобы снизить затраты на управление. В некоторых случаях есть определенный смысл отказаться от поддержки трех кластеров, по одному для окружения разработки, тестирования и работы, и настроить все три окружения в одном кластере Kubernetes. Кроме того, некоторые компании не хотят иметь отдельные кластеры для разных подразделений. И снова это решение принимается бизнесом с учетом безопасности, а организации, с которыми мы работаем, обычно имеют ограниченные бюджеты и человеческие ресурсы.

Мы не собираемся давать пошаговые инструкции, касающиеся реализации коллективной аренды, а просто перечислим шаги, которые необходимо выполнить. Эти шаги будут меняться со временем и различаться в разных организациях с разными моделями безопасности.

- 1 Спроектируйте модель безопасности. На первый взгляд этот шаг может показаться очевидным, но мы видели организации, не использующие модели безопасности. Модель безопасности должна определять роли пользователей, включая администраторов кластера и пространств имен, а также одну или несколько ролей арендатора. Не менее важное значение имеет стандартизация соглашений об именах всех объектов API, пользователей и других компонентов, создаваемых вашей организацией.
- 2 Используйте различные объекты API:
 - Namespace;
 - NetworkingPolicy;
 - ResourceQuota;
 - ServiceAccount и RBACRule.

- 3 Применение таких инструментов, как сервисные сетки с расширенной поддержкой TLS и управления сетевыми политиками, может обеспечить более высокий уровень безопасности. Применение сервисной сетки добавляет дополнительные сложности, поэтому используйте ее только при необходимости.
- 4 Подумайте о возможности использования ОРА для организации управления кластером Kubernetes на основе политик.

СОВЕТ Если вы собираетесь объединить несколько окружений в один кластер, то могут возникнуть проблемы, связанные не только с безопасностью, но также с тестированием обновлений Kubernetes. Обновления лучше сначала протестировать на другом кластере.

14.5 Советы по Kubernetes

Вот краткий список различных советов по настройке кластера:

- добавьте конечную точку частного сервера API и, если возможно, не открывайте доступ к серверу API из интернета;
- используйте RBAC;
- используйте сетевые политики;
- не включайте на сервере API авторизацию по имени пользователя и паролю;
- используйте определенные учетные записи для создания модулей и не используйте учетные записи администратора по умолчанию;
- запуск модулей Pod в сети хоста допустим лишь в редких и исключительных случаях;
- используйте `serviceAccountName`, если модулю Pod требуется доступ к серверу API; в противном случае присвойте параметру `automountServiceAccountToken` значение `false`.
- используйте квоты ресурсов для пространств имен и определяйте ограничения во всех модулях Pod.

Итоги

- Безопасность узла зависит от сертификатов TLS, защищающих взаимодействия между узлами и плоскостью управления.
- Использование неизменяемых операционных систем может усилить защиту узлов.
- Ограничение ресурсов может предотвратить атаки на ресурсы.
- Используйте сеть Pod, если нет явной необходимости использовать сеть хоста. Сеть хоста позволяет модулю Pod взаимодействовать с ОС узла.

- RBAC – ключ к защите сервера API. Это непросто, но необходимо.
- Учетные записи сервиса IAM помогают изолировать разрешения модулей Pod.
- Сетевые политики – ключ к изоляции сетевого трафика. В их отсутствие все смогут общаться со всеми.
- Агент открытой политики (OPA) позволяет создавать политики безопасности и применять их в кластере Kubernetes.
- Изначально Kubernetes создавался без учета возможности коллективной аренды с нулевым доверием. Вы можете использовать разные формы коллективной аренды, но все они сопряжены с компромиссами.

15

Установка приложений

В этой главе:

- обзор средств управления приложениями в Kubernetes;
- установка прототипа приложения гостевой книги Guestbook;
- сборка версии приложения Guestbook для эксплуатации.

Управлять приложениями в Kubernetes, как правило, намного проще, чем на обычных физических серверах, потому что всю настройку приложений можно выполнить через унифицированный интерфейс командной строки. И все же, когда речь идет о перемещении десятков или сотен контейнеров в среду Kubernetes, к управлению конфигурацией, которое необходимо автоматизировать, бывает сложно подойти с единой точки зрения. ConfigMap, Secret, учетные данные сервера API и настройка типов томов – это лишь некоторые примеры, которые могут сделать администрирование Kubernetes весьма утомительным занятием.

В этой главе мы (наконец-то) отойдем от исследования внутренних деталей реализации Kubernetes и поговорим немного о настройке и администрировании приложений. Начнем с размышлений о том, что такое приложение и как приложения устанавливаются в Kubernetes.

15.1 Размышления о приложениях в Kubernetes

Для целей дальнейшего обсуждения будем называть приложения Kubernetes наборами ресурсов API, которые необходимо развернуть. Каноническим примером может служить приложение гостевой книги Guestbook, исходный код которого доступен по адресу <http://mng.bz/y4NE>. Это приложение включает:

- модуль Pod ведущей базы данных Redis;
- модуль Pod ведомой базы данных Redis;
- интерфейсный модуль Pod, взаимодействующий с ведущей базой данных Redis;
- сервис для всех этих трех модулей.

Доставка приложения включает обновление, параметризацию и настройку множества различных ресурсов Kubernetes. Этот широко обсуждаемый вопрос имеет множество различных технических решений, поэтому мы не будем пытаться решить всю головоломку, а просто представим широкий спектр инструментов, потому что большая часть решения, касающаяся развертывания модулей Pod, связана в первую очередь с особенностями установки и настройки приложения. Приложение Guestbook можно запустить в любом кластере Kubernetes, как показано ниже:

```
$ kubectl create -f https://github.com/kubernetes/examples/blob/master/
  guestbook/all-in-one/guestbook-all-in-one.yaml
```

Получение файлов из интернета

Мы уже отмечали это, но повторим еще раз: загрузка файлов YAML из интернета таит в себе опасности. В нашем случае мы загружаем наши файлы YAML непосредственно с адреса github.com/kubernetes надежного репозитория, поддерживаемого тысячами известных и проверенных членов фонда CNCF (Cloud Native Computing Foundation). К концу этой главы мы представим более реалистичный пример установки того же приложения Guestbook корпоративного уровня, так что просто подождите.

Через короткое время после ввода предыдущей команды можно заметить, что все модули Pod запустились и работают с несколькими репликами трех модулей Pod интерфейса и ведомой базы данных Redis. В самом простом случае установка приложения Kubernetes выглядит так:

```
$ kubectl get pods -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	frontend-6c-7wjx8	1/1	Running	0	3m18s
default	frontend-6c-g7z8z	1/1	Running	0	3m18s
default	frontend-6c-xd5q2	0/1	ContainerCreating	0	3m18s
default	redis-master-f46-l2d	1/1	Running	0	3m18s

default	redis-slave-797-nv9	1/1	Running	0	3m18s
default	redis-slave-797-9qc	1/1	Running	0	3m18s

15.1.1 Масштаб приложения влияет на выбор инструментов

Устанавливая приложение, мы должны задать себе несколько очевидных вопросов, касающихся масштабирования, обновления, настройки, безопасности и модульности.

- Как будет осуществляться масштабирование веб-приложения Guestbook? Автоматически или вручную?
- Будет ли веб-приложение Guestbook периодически обновляться? Будут ли при этом одновременно обновляться несколько модулей Pod? Если да, то нужно ли для этого создавать оператор Kubernetes?
- Есть ли у нас четко определенные конфигурации (например, имеется ли несколько альтернативных конфигураций Redis)? Если да, то должны ли мы использовать `ytt`, `kustomize` или какой-то другой инструмент, чтобы не приходилось копировать и вставлять большие фрагменты YAML каждый раз, когда потребуется сохранить новый вариант настроек приложения?
- Защищена ли база данных Redis? Должна ли она быть защищена? Если да, то должны ли мы добавить учетные данные RBAC для обновления или редактирования пространства имен, в котором действует приложение, и должны ли мы установить правила NetworkPolicy? Правила можно просмотреть на странице <https://redis.io/topics/security> и реализовать их с помощью Secret, ConfigMap и т. д. Кроме того, нам может понадобиться периодически менять секреты Secret, что требует автоматизации. (Это намекает на потребность в операторе.)
- Приложение можно развернуть в определенном пространстве имен, но как отслеживать происхождение образов и общее состояние работоспособности приложения с течением времени, а также обновлять его атомарным способом? Развертывание огромного списка ресурсов Kubernetes в виде приложения из большого файла – весьма неуклюжее решение по нескольким причинам, самая главная из которых – неясно, что на самом деле представляет собой приложение, когда окажется в кластере.

15.2 Приложения на основе микросервисов обычно требуют тысячи строк определения конфигурации

Микросервисная архитектура помогает разбить функциональность приложения на отдельные сервисы, каждый из которых имеет свои

уникальные настройки. Из-за этого настройка микросервисов оказывается более трудоемкой, по сравнению с монолитными приложениями, где большая часть проблем со связью и безопасностью устраняется за счет выполнения всех вычислений в памяти. Но вернемся к нашему приложению Guestbook, включающему три микросервиса и 200 строк кода – по 10–50 строк для каждого создаваемого объекта API.

Типичное приложение Kubernetes уровня предприятия имеет более сложную конфигурацию, включающую от 10 до 20 контейнеров, каждый из которых, как правило, имеет хотя бы один сервис, один объект ConfigMap и несколько секретов Secret. Объем кода для приложения в такой конфигурации легко может превысить несколько тысяч строк кода YAML, разбросанных по множеству файлов. Очевидно, что копировать тысячи строк кода при каждом развертывании приложения неудобно, поэтому рассмотрим несколько решений управления конфигурацией Kubernetes для реальных приложений.

Не бойтесь переосмысливать свои приложения

Прежде чем углубиться в бесконечность вариантов приложений, обратите внимание на одно обстоятельство: если ваши инструменты установки чрезвычайно сложны, то высока вероятность не заметить неработающие элементы в базовом приложении. В этих случаях целесообразно упростить способ управления приложением. Например, разработчики часто создают слишком много микросервисов или закладывают в приложение больше гибкости, чем необходимо (как правило, потому что нет адекватного способа оценить и выбрать правильные настройки по умолчанию). Иногда, занимаясь конфигурацией, лучшее решение – полностью исключить возможности настройки.

15.3 Переосмысление установки приложения Guestbook в реальных условиях

Теперь, определив в общих чертах предметное пространство, – управление конфигурациями приложений Kubernetes, – давайте переосмыслим наше приложение Guestbook с учетом следующих решений:

- *шаблоны Yaml* – их поддержку мы будем осуществлять с помощью *ytt*, но также можно использовать такие инструменты, как *kustomize* или *helm3*;
- *развертывание и обновление приложения* – для этой цели мы будем использовать проект *kapp-controller* компании Carvel, но также можно создать оператор Kubernetes.

Почему не helm?

Мы не хотим явно отдавать предпочтение какому-то одному инструменту: для достижения одних и тех же конечных целей можно вполне успешно использовать и `helm3`, и `kustomize`, и `ytt`. Мы выбрали `ytt` из-за его модульной и полностью программируемой природы (и он интегрируется со `Starlark`). Но вы можете выбрать другой инструмент. `helm3`, `kustomize`, `ytt` – отличные инструменты, но есть много других, ничуть не хуже решающих задачу управления шаблонами YAML. Представленные нами примеры можно реализовать с использованием множества других технологий. Если на то пошло, у вас под рукой всегда есть `sed`.

Набор Carvel (<https://carvel.dev>) содержит несколько инструментов, которые можно использовать вместе или по отдельности для управления всем пространством задач, которое мы описали до сих пор. На самом деле это основа дистрибутива VMware Tanzu.

15.4 Установка набора инструментов Carvel

Первый шаг к расширению возможностей нашей гостевой книги – установка инструментария Carvel, после которой мы сможем запускать инструменты из командной строки. Ниже показана команда установки набора. В дальнейшем мы будем использовать `ytt`, `kapp` и `kapp-controller` для постепенного улучшения и автоматизации нашего приложения Guestbook:

```
# в macOS
$ brew tap vmware-tanzu/carvel ; brew install ytt kbd kapp imgpkg kwt vendir

# в Linux
$ curl -L https://carvel.dev/install.sh | bash
```

Действительно ли нужен весь набор Carvel?

Хотя в этой главе нам не понадобятся все инструменты из набора Carvel, но мы все равно установим их, потому что они хорошо сочетаются друг с другом. Мы предлагаем изучить некоторые из них (например, `imgpkg` или `vendir`) самостоятельно. Все инструменты Carvel просты в использовании, автономны и занимают незначительное место на диске. Вы можете смело использовать их для самообучения.

15.4.1 Часть 1: разделение ресурсов на отдельные файлы

Первое, что следует сделать, знакомясь с 200 строками кода в нашем проекте, – разбить их на более мелкие фрагменты. Причины этого шага очевидны:

- использовать такие инструменты, как `grep` или `sed`, намного проще, когда нет большого количества повторяющихся строк;
- выяснить, кто и что изменил в системе управления версиями, намного проще, если файлы небольшие;
- добавление новых объектов Kubernetes в файл приведет к его раздуванию, поэтому рано или поздно обязательно встанет вопрос о модульной организации. Мы могли бы сделать это прямо сейчас.

Мы разделили Guestbook на два отдельных каталога и поместили их на <http://mng.bz/M2wm>, чтобы вы могли клонировать и поэкспериментировать с ними. Не стесняйтесь разбивать файлы логичным и понятным для вас способом.

Необязательно точно следовать шагам, что приводятся в этом разделе, потому что, если спросить 10 разных программистов, как разложить объект на части, вы получите 10 разных ответов. Однако, выполняя декомпозицию, обязательно учитывайте одно важное обстоятельство: каждый ресурс Kubernetes должен иметь *уникальное имя*. Например, развертывание ведущей БД Redis должно иметь имя, отличное от объекта сервиса Redis. Например, ниже мы добавили суффикс `-dep` в имя развертывания ведущей БД Redis:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-master-dep
---
```

Тот же прием мы применили к внешнему интерфейсу. Вслед за фрагментом с определением развертывания интерфейса показана получившаяся структура каталогов:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-dep
spec:
```

```
-> carvel-guestbook git:(master) ? tree
```

```
.
├─ v0
│   └─ original.yaml
└─ v1
    ├── fe-dep.yaml
    ├── fe-svc.yaml
    ├── redis-master-dep.yaml
    ├── redis-master-svc.yaml
    ├── redis-slave-dep.yaml
    └─ redis-slave-svc.yaml
```

Переименование ведущей БД Redis и внешнего интерфейса

Если вы не собираетесь использовать файлы, доступные по адресу <http://mng.bz/M2wm>, и решили разделить Guestbook самостоятельно, обязательно переименуйте ведущую БД Redis и внешний интерфейс, дав им имена `redis-master-dep` и `frontend-dep` в поле `metadata.name` (как показано в предыдущих фрагментах кода). Позже это позволит использовать `ytt` для поиска и замены значений в конструкциях YAML.

Теперь можно проверить эквивалентность результата декомпозиции исходному приложению, запустив `kubectl create -f v1/`. Выполните эту команду самостоятельно и убедитесь, что три модуля Pod внешнего интерфейса и два модуля Redis запустились и успешно работают. После этого можно приступить к настройке переадресации портов для локального доступа к приложению Guestbook через порт 8080. Например:

```
$ kubectl port-forward service/frontend 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Теперь введите несколько значений в поле **Messages** (Сообщения) в приложении и посмотрите, как они сохраняются в базе данных Redis. Обратите внимание, что они также отображаются на главной странице Guestbook.

15.4.2 Часть 2: исправление файлов приложения с помощью ytt

У нас есть действующее приложение с внешним интерфейсом и сервисом. Что произойдет, если подвергнуть его большой нагрузке? В таком случае, например, может появиться желание выделить ему больше вычислительных ресурсов. Для этого нужно изменить файл `fe-dep.yaml` и увеличить значение параметра `request.cpu`. Это означает, что нам нужно отредактировать некоторый файл YAML:

```
containers:
- name: php-redis
  image: gcr.io/google-samples/gb-frontend:v4
resources:
  requests:
    cpu: 100m
    memory: 100Mi
```

Одна десятая ядра – это не так много для приложения

В этом примере мы могли бы с легкостью заменить `100m` на `1`, но это всего лишь замена одной жесткой константы другой. Было бы лучше параметризовать это значение. Кроме того, позднее у нас может появиться желание увеличить требования Redis к процессору. К счастью, у нас есть `ytt`.

Механизм шаблонов YAML ytt (<https://carvel.dev/ytt/>) позволяет настраивать файлы YAML с помощью таких методов, как наложение, исправление и т. д. Он также поддерживает расширенные конструкции языка Starlark для реализации логического манипулирования текстом. Поскольку мы уже установили инструменты Carvel, давайте просто посмотрим, как можно настроить требования к процессору в нашем приложении.

YAML на входе, YAML на выходе

ytt – это инструмент, принимающий и возвращающий код YAML, и это важно. В отличие от других инструментов, которые появлялись и исчезали в экосистеме Kubernetes, ytt (как и другие инструменты Carvel) фокусируется на выполнении одной конкретной работы и делает это очень хорошо. Он манипулирует кодом YAML! Он устанавливает файлы и никак не связан с Kubernetes.

Во второй (v2) итерации мы добавим новый файл (назовем его ytt-cpu-overlay.yaml) в новый каталог (назовем его v2/). Наша цель – сопоставить раздел `cpu` в интерфейсном веб-приложении `php-redis` с модулем `Pod` ведущей базы данных `Redis`:

```

#@ load("@ytt:overlay", "overlay")
#@overlay/match by=overlay.subset(
  {"metadata":{"name":"frontend-dep"}})
---
spec:
  template:
    spec:
      containers:
        #@overlay/match by="name"
        - name: php-redis
          #@overlay/match missing_ok=False
          resources:
            requests:
              cpu: 200m

```

Оверлей ytt идентифицирует имя фрагмента YAML для сопоставления

Оказавшись внутри контейнера, он подставит контейнер именем `php-redis`

Исходное значение `cpu`, равное `100m`, теперь увеличено вдвое, до `200m`

Точно так же можно поступить с модулем `Pod` базы данных: создать новый файл с именем `v2/ytt-cpu-overlay-db.yaml`, который делает то же самое:

```

#@ load("@ytt:overlay", "overlay")
#@overlay/match by=overlay.subset({"metadata":{"name":"redis-master-dep"}})
---
spec:
  template:
    spec:
      containers:
        #@overlay/match by="name"

```

```

- name: master
  resources:
    requests:
      #@overlay/match missing_ok=True
      cpu: 300m

```

Определяет новое значение
сри (на этот раз 300m,
исключительно ради отличия)

Теперь можно выполнить описанное преобразование. Например:

```

$ tree v2
v2
├─ ytt-cpu-overlay-db.yml
└─ ytt-cpu-overlay.yml

$ ytt -f ./v1/ -f v2/ytt-cpu-overlay.yml -f v2/ytt-cpu-overlay-db.yml
...
      cpu: 200m
...
      cpu: 300m

```

Изменяет файл, увеличить объем
вычислительных ресурсов только
для ведущей БД

```

$ kubectl delete -f v0/original.yaml
$ ytt -f ./v1/ -f v2/ytt-cpu-overlay.yml \
  | -f v2/ytt-cpu-overlay-db.yml | kubectl create -f -
deployment.apps/frontend-dep created
service/frontend created
deployment.apps/redis-master-dep created
service/redis-master created
deployment.apps/redis-slave created
service/redis-slave created

```

Отлично, мы прошли полный круг! Изначально у нас был один файл, который легко упаковать, но трудно модифицировать. Мы разбили файл и, используя ytt, добавили слой настройки, чтобы затем передать результаты в команду kubectl как один ресурс YAML.

Теперь приложение готово к эксплуатации, потому что появилась возможность добавлять и изменять конфигурационные параметры. Если заглянуть в документацию по адресу <https://carvel.dev/ytt/>, то можно увидеть, что поддерживается еще множество дополнительных действий, таких как добавление значений, вставка совершенно новых конструкций YAML и т. д. Однако в нашем случае мы оставим все как есть и поднимемся вверх по стеку, чтобы посмотреть, как объединить исправленные ресурсы YAML в одно приложение с управляемым состоянием.

15.4.3 Часть 3: развертывание приложения Guestbook и управление им

Кому-то из вас может не понравиться, как часто мы запускаем kubectl delete в этой книге. Возможно, вы даже задавались вопросом: зачем мы это делаем? Причина проста – потому что мы не изолировали наше приложение от других приложений в кластере. Один из про-

стных способов сделать это – развернуть приложение в отдельном пространстве имен, которое затем можно удалить или создать. Однако, начиная рассматривать несколько ресурсов как одно приложение, мы получаем новый набор вопросов, на которые хотим ответить.

- Сколько отдельных приложений будет запускаться в этом пространстве имен?
- Насколько успешно обновились все ресурсы приложения?
- Сколько ресурсов каждого типа связано с приложением?

На каждый из этих вопросов можно ответить с помощью утилит `kubectl`, `gcr` и некоторого количества кода на `Bash`. Однако этот подход плохо масштабируется, если в приложении имеются десятки или сотни контейнеров, объектов `Secret`, `ConfigMap` и других ресурсов. Кроме того, он подходит не для всех приложений в кластере, число которых также может легко достигать сотен и даже тысяч. В такой ситуации вам поможет инструмент `kapp`.

А что насчет `helm`?

`helm` – одно из первых и наиболее успешных решений для управления приложениями в `Kubernetes`. Первоначально он сочетал аспекты обновления с сохранением состояния и установки ресурсов с шаблонами `YAML`. Проект `Carvel` использовал опыт `helm` и выделил многие из этих функций в отдельные инструменты.

`helm3` на самом деле является более модульной попыткой управления приложениями, которые можно запускать без сохранения состояния, подобно тому, что мы увидим, когда начнем знакомиться с инструментом `kapp`. В любом случае `helm3` и экосистема `Carvel` во многом пересекаются, и оба могут использоваться в одинаковых ситуациях, но они основаны на разных мнениях, философских подходах и сообществах. Мы рекомендуем изучить оба инструмента, особенно если вы считаете, что `kapp` не идеальное решение для ваших задач. В любом случае вы узнаете довольно много об управлении приложениями `Kubernetes` в целом, следя за развитием `Guestbook` с использованием `kapp`!

Инструмент `kapp` (<https://carvel.dev/kapp/>) довольно прост в использовании, особенно теперь, когда у нас есть возможность настраивать приложение с помощью `ytt`. Для начала давайте в еще раз очистим наше приложение `Guestbook`, если оно все еще работает:

<pre>\$ ytt -f ./v1/ -f v2/ytt-cpu-overlay.yml -f v2/ytt-cpu-overlay-db.yml kubectl delete -f -</pre>	<div style="border-left: 1px solid black; padding-left: 10px;">Удалит ресурсы, созданные в предыдущем разделе (на случай, если они все еще существуют)</div>
---	--

Предположим, что вы уже установили `kapp`. Теперь запустим те же команды `ytt` для создания приложения, но для установки используем `kapp` вместо `kubectl`. Обратите внимание, что в этом примере в качестве провайдера CNI используется `Antrea`. Но на данном этапе не-

важно, какого провайдера CNI вы выбрали (обратите внимание, что несколько столбцов в выводе в этом фрагменте кода опущены, чтобы уместить листинг по ширине книжной страницы):

```
$ kapp deploy -a guestbook -f <(ytt -f ./v1/
➡ -f v2/ytt-cpu-overlay.yml
➡ -f v2/ytt-cpu-overlay-db.yml) ←
```

Принимает инструкцию ytt,
генерирующую YAML,
и передает ее на вход kapp

Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)

Changes

Namespace	Name	Kind	...	Op	Op st.	Wait to ...
default	frontend	Service	...	create	-	reconcile ...
^	frontend-dep	Deployment	...	create	-	reconcile ...
^	redis-master	Service	...	create	-	reconcile ...
^	redis-master-dep	Deployment	...	create	-	reconcile ...
^	redis-slave	Deployment	...	create	-	reconcile ...
^	redis-slave	Service	...	create	-	reconcile ...

Op: 6 create, 0 delete, 0 update, 0 noop

Wait to: 6 reconcile, 0 delete, 0 noop

Continue? [yN]:

Если в ответ на вопрос ввести `y`, то инструмент `kapp` проделает большой объем работы, в том числе аннотирует ресурсы, чтобы позже он мог управлять ими автоматически. Он обновит или удалит ресурсы или сообщит общее состояние приложения. В этом примере мы назвали приложение `Guestbook`, но мы могли бы назвать его как угодно.

После ввода `y` («да») вы увидите больше информации, чем доступно в `kubectrl`. Потому что `kapp` тщательно исследует состояние приложения и делает все возможное, чтобы убедиться, что все ресурсы находятся в работоспособном состоянии. Нетрудно представить, как можно использовать `kapp` в среде непрерывной интеграции/доставки (CI/CD) для полной автоматизации обновления и управления приложением. Например:

```
11:17:40AM: ---- applying 6 changes [0/6 done] ----
11:17:40AM: create deployment/frontend-dep (apps/v1) namespace: default
11:17:40AM: create deployment/redis-master-dep (apps/v1) namespace: default
11:17:40AM: create service/redis-master (v1) namespace: default
11:17:40AM: create service/redis-slave (v1) namespace: default
11:17:40AM: create deployment/redis-slave (apps/v1) namespace: default
11:17:40AM: create service/frontend (v1) namespace: default
11:17:40AM: ---- waiting on 6 changes [0/6 done] ----
11:17:40AM: ok: reconcile service/frontend (v1) namespace: default
11:17:40AM: ok: reconcile service/redis-slave (v1) namespace: default
11:17:40AM: ok: reconcile service/redis-master (v1) namespace: default
11:17:41AM: ongoing: reconcile deployment/frontend-dep (apps/v1)
➡ namespace: default
```

```

11:17:41AM: ^ Waiting for generation 2 to be observed
11:17:41AM: L ok: waiting on replicaset/frontend-dep-7bf896bf7c (apps/v1)
➔ namespace: default
11:17:41AM: L ongoing: waiting on pod/frontend-dep-7bf896bf7c-vbn22 (v1)
➔ namespace: default
11:17:41AM: ^ Pending: ContainerCreating
11:17:41AM: L ongoing: waiting on pod/frontend-dep-7bf896bf7c-qph5b (v1)
➔ namespace: default
...
11:17:44AM: ---- waiting on 1 changes [5/6 done] ----
11:18:01AM: ok: reconcile deployment/redis-master-dep (apps/v1)
➔ namespace: default
11:18:01AM: ---- applying complete [6/6 done] ----
11:18:01AM: ---- waiting complete [6/6 done] ----
Succeeded

```

Теперь вернемся назад и еще раз взглянем на наше приложение, чтобы убедиться, что оно все еще работает. Для этого выполните следующую команду:

```
$ kapp list
```

```
Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)
```

```
Apps in namespace 'default'
```

Name	Namespaces	Lcs	Lca
guestbook	default	true	12m

Lcs: Last Change Successful

Lca: Last Change Age

```
1 apps
```

```
Succeeded
```

kapp можно также использовать для получения подробной информации о запущенном приложении:

```
$ kapp inspect --app=guestbook
```

```
Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)
```

```
Resources in app 'guestbook'
```

Name	Kind	Owner	Conds.	Age
frontend	Endpoints	cluster	-	12m
frontend	Service	kapp	-	12m
frontend-dep	Deployment	kapp	2/2 t	12m
frontend-dep-7bf7c	ReplicaSet	cluster	-	12m
frontend-dep-7bf7c-g7jlt	Pod	cluster	4/4 t	12m
frontend-dep-7bf7c-qph5b	Pod	cluster	4/4 t	12m
frontend-dep-7bf7c-vbn22	Pod	cluster	4/4 t	12m
frontend-sccps	EndpointSlice	cluster	-	12m

redis-master	Endpoints	cluster	-	12m
redis-master	Service	kapp	-	12m
redis-master-dep	Deployment	kapp	2/2 t	12m
redis-master-dep-64fcb	ReplicaSet	cluster	-	12m
redis-master-dep-64fcb-t4hjl	Pod	cluster	4/4 t	12m
redis-master-zqdv	EndpointSlice	cluster	-	12m
redis-slave	Deployment	kapp	2/2 t	12m
redis-slave	Endpoints	cluster	-	12m
redis-slave	Service	kapp	-	12m
redis-slave-dffcf	ReplicaSet	cluster	-	12m
redis-slave-dffcf-75vfq	Pod	cluster	4/4 t	12m
redis-slave-dffcf-lwch9	Pod	cluster	4/4 t	12m
redis-slave-vlnkh	EndpointSlice	cluster	-	12m

Rs: Reconcile state

Ri: Reconcile information

21 resources

Succeeded

Следует отметить, что в выводе присутствуют некоторые объекты, созданные кластером Kubernetes «за кулисами», такие как Endpoint и EndpointSlice. Объекты EndpointSlice и их доступность для балансировки нагрузки сервисов Service имеют решающее значение для доступности приложения конечным потребителям. kapp собрал эту информацию, а также сведения об успехах и сбоях всех ресурсов в приложении в единой удобной для чтения таблице.

Наконец, теперь мы с легкостью можем полностью удалить приложение, выполнив команду `kapp delete --app=guestbook`. Это обратная операция нашей операции `kapp deploy`, и мы не будем показывать ее вывод, потому что он не требует дополнительных пояснений.

15.4.4 Часть 4: создание оператора kapp для упаковки приложения и управления им

Теперь, организовав приложение в виде набора атомарных ресурсов с четко определенными именами, мы фактически создали пользовательское определение ресурса (Custom Resource Definition, CRD). Проект `kapp-controller` позволяет добавить в любое `kapp`-приложение несколько дополнительных возможностей автоматизации. Этот последний шаг завершает наш переход от «одного большого блока YAML из интернета» к автоматически управляемому приложению с состоянием, которое можно запускать на предприятии вместе с сотнями других приложений. Здесь мы также кратко познакомимся с особенностями создания операторов Kubernetes.

Для начала установим инструмент `kapp-controller` с помощью `kapp`. Мы снова приступаем к установке артефактов из интернета, поэтому не ленитесь проверять файлы YAML перед установкой:

```
$ kapp deploy -a kc -f https://github.com/vmware-tanzu/
  carvel-kapp-controller/releases/latest/
  download/release.yml
```

Установка kapp-controller с помощью kapp

```
$ kapp deploy -a default-ns-rbac -f
  https://raw.githubusercontent.com/vmware-tanzu/
  carvel-kapp-controller/
  develop/examples/rbac/default-ns.yml
```

Установка простого определения RBAC

Возможно, вам интересно, зачем настраивать правила RBAC для kapp-controller. Установка определения RBAC (default-ns.yml) позволяет kapp-controller манипулировать объектами API в пространстве имен по умолчанию, как того требует любой оператор. Операторы – это административные приложения, а модуль Pod с kapp-controller должен создавать, редактировать и обновлять различные ресурсы Kubernetes, чтобы выполнять свою работу универсального оператора для наших приложений.

Теперь, когда в кластере имеется kapp-controller, его можно использовать для автоматизации утт, причем декларативным способом. Для этого нужно создать CR (CustomResource – пользовательский ресурс) кapp. Спецификация приложения кapp описана на странице <http://mng.bz/PWqv>. Наибольший интерес для нас представляют поля:

- `git` – определяет клонируемый репозиторий Git с исходным кодом наших приложений;
- `template` – определяет местонахождение шаблонов утт для установки приложения в реальном времени.

Первым делом создадим спецификацию для нашего оригинального приложения Guestbook, которое будет выполняться под управлением kapp. Затем добавим наши шаблоны утт обратно:

```
apiVersion: kappctrl.k14s.io/v1alpha1
kind: App
metadata:
  name: guestbook
  namespace: default
spec:
  serviceAccountName: default-ns-sa
  fetch:
  - git:
      url: https://github.com/jayunit100/k8sprototypes
      ref: origin/master

  # в корне репозитория имеется каталог 'app' с файлами,
  # описывающими приложение (например, определения модулей, сервисов).
  subPath: carvel-guestbook/v1/
  template:
  - ytt: {}
  deploy:
  - kapp: {}
```

Использует учетную запись сервиса, созданную ранее для установки приложения

Определяет местонахождение приложения

Поскольку код приложения на самом деле находится в carvel-guestbook/v1/, нужно сообщить этот дополнительный путь

В этот момент у вас может вспыхнуть догадка, что при таком подходе можно организовать непрерывную доставку, и вы не ошиблись.

Это единственное объявление YAML позволяет передать все управление нашим приложением фреймворку Kubernetes и оператору kapp-controller. Давайте попробуем. Запустите `kubectl create -f`, чтобы создать приложение Guestbook, используя показанный выше фрагмент кода YAML, а затем выполните следующую команду:

```
$ kapp list
Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)

Apps in namespace 'default'
```

Name	Namespaces	Lcs	Lca
default-ns-rbac	default	true	14m
guestbook-ctrl	default	true	1m
...			

Как видите, kapp-controller автоматически создал наше приложение guestbook-ctrl. Проверим его с помощью kapp:

```
$ kapp inspect --app=guestbook-ctrl
Target cluster 'https://127.0.0.1:53756'
(nodes: antrea-control-plane, 2+)
```

```
Resources in app 'guestbook-ctrl'
```

Namespace	Name	Kind	Owner	Conds.	Rs
default	fe	Dep	kapp	2/2 t	ok
^	fe	Endpoints	cluster	-	ok
^	fe	Service	kapp	-	ok
...					

На самом деле мы интегрировали приложение в систему CI/CD, полностью управляемую внутри Kubernetes. Замечательно! Теперь легко можно представить создание произвольно сложных систем для разработчиков, позволяющих им отправлять и поддерживать CRD для их приложений, которые в конечном итоге развертываются и управляются единым оператором kapp-controller, действующим в пространстве имен по умолчанию.

При желании можно повторно развернуть это же приложение в новом пространстве имен, просто добавляя или удаляя их командой `kubectl get apps`, потому что модуль Pod kapp-controller установил CRD для приложений kapp в нашем кластере:

```
$ kubectl get apps
NAME      DESCRIPTION      SINCE-DEPLOY  AGE
guestbook  Reconcile succeeded  5m16s        6m8s
```

Мы только что реализовали полноценное развертывание оператора приложения Guestbook. Теперь попробуем добавить наши шаблоны ytt. В этом примере мы переместили вывод ytt (полученный в предыдущем примере) в конкретный каталог в репозитории k8sprototypes (вы можете использовать свой репозиторий GitHub для этого упражнения, но это необязательно):

```

apiVersion: kappctrl.k14s.io/v1alpha1
kind: App
metadata:
  name: guestbook
  namespace: default
spec:
  serviceAccountName: default-ns-sa
  fetch:
  - git:
    url: https://github.com/jayunit100/k8sprototypes
    ref: origin/master
    subPath: carvel-guestbook/v2/output/
  template:
  - ytt: {}
  deploy:
  - kapp: {}

```

Теперь можно создать новое определение для приложения Guestbook, включающее наши шаблоны `ytt`, просто записав преобразованные шаблоны `ytt` в другой каталог. Еще одна прелесть использования операторов для управления приложениями заключается в возможности создания и удаления их без применения специальных инструментов. Это связано с тем, что клиент `kubectl` знает о них как о ресурсах API. Чтобы удалить приложение Guestbook, выполните следующую команду:

```
$ kubectl delete app guestbook
```

Теперь мы можем просто использовать `kubectl` для декларативного удаления приложения Guestbook, а `kapp-controller` сделает все остальное. Также можно использовать команды, такие как `kubectl describe`, чтобы узнать состояние приложения.

Мы лишь слегка коснулись гибкости, которую дает модель операторов в отношении управления и создания определений приложений. На будущее вам также стоит изучить:

- использование `kapp-controller` для развертывания нескольких копий одного и того же приложения в нескольких пространствах имен;
- использование директивы `ytt` внутри `kapp-controller`;
- использование возможностей `kapp-controller` для развертывания и управления диаграммами Helm как приложениями;
- внедрение секретов в приложения `kapp` для безопасного развертывания рабочих процессов CI/CD.

На этом мы завершаем итеративное усовершенствование приложения Guestbook. А в завершение главы вернемся к нашим старым знакомым – провайдерам CNI Calico и Antrea – и посмотрим, как они внедряют полноценные операторы Kubernetes с детализированными CRD для администраторов.

15.5 И снова об операторах Kubernetes

Инструменты `kapp` и `kapp-controller` открыли нам возможность автоматизированного управления всеми сервисами в приложении `Guestbook`. Этот пример органично познакомил нас с идеей оператора Kubernetes. Применение универсального инструмента, такого как `kapp-controller` или `Helm` (<https://helm.sh>), может избавить вас от сложностей создания полноценного определения CRD и реализации оператора. Однако CRD широко распространены в современной экосистеме Kubernetes, и мы окажем вам медвежью услугу, если не расскажем о них хотя бы немного подробнее.

Фабричные операторы

Если вы решите, что ваше приложение достаточно продвинутое, чтобы нуждаться в собственных расширениях Kubernetes API, то вам нужно создать оператора. Под созданием оператора обычно предполагается автоматическое создание клиентов API, которые должны выполнять «правильные» действия при создании, удалении или редактировании ресурсов. В интернете есть много инструментов, таких как проект <https://github.com/kubernetes-sigs/kubebuilder>, которые упрощают создание полноценных операторов.

Давайте развернем кластер `kind` с двумя разными поставщиками CNI (`Calico` и `Antrea`), чтобы лучше понять, как использовать CRD. В данном случае, поскольку может понадобиться добавить объекты `NetworkPolicy`, создадим кластер на основе `Calico`. Для этого можно использовать сценарий `kind-local-up.sh`:

```
# установить kind и kubectl перед выполнением команды...
$ git clone \
  https://github.com/jayunit100/k8sprototypes.git
$ cd kind ; ./kind-local-up.sh
```

Разработчики приложений для Kubernetes часто создают множество CRD. CRD позволяют любому приложению использовать сервер Kubernetes API для хранения конфигурации и создания операторов (о которых мы поговорим далее в этой главе). Операторы – это контроллеры Kubernetes, следящие за изменениями на сервере API и при их появлении выполняющие определенные действия. Например, взглянув в наш недавно созданный кластер `kind`, можно заметить несколько ресурсов CRD `Calico`, которые имеют определенные конфигурации `Calico` в роли провайдера CNI:

```
$ kubectl get crd ← Выведет список всех определений CRD в кластере
NAME
bgpconfigurations.crd.projectcalico.org
bgppeers.crd.projectcalico.org
```

```

blockaffinities.crd.projectcalico.org
clusterinformations.crd.projectcalico.org
felixconfigurations.crd.projectcalico.org
globalnetworkpolicies.crd.projectcalico.org
globalnetworksets.crd.projectcalico.org
hostendpoints.crd.projectcalico.org
ipamblocks.crd.projectcalico.org
ipamconfigs.crd.projectcalico.org
ipamhandles.crd.projectcalico.org
ippools.crd.projectcalico.org
kubecollectorsconfigurations.crd.projectcalico.org
networkpolicies.crd.projectcalico.org
networksets.crd.projectcalico.org

$ kubectl get kubecontrollersconfigurations -o yaml
apiVersion: v1
items:
- apiVersion: crd.projectcalico.org/v1
  kind: KubeControllersConfiguration
  ...
spec:
  controllers:
    namespace:
      reconcilerPeriod: 5m0s
    node:
      reconcilerPeriod: 5m0s
      syncLabels: Enabled
    policy:
      reconcilerPeriod: 5m0s
    serviceAccount:
      reconcilerPeriod: 5m0s
    workloadEndpoint:
      reconcilerPeriod: 5m0s
  etcdV3CompactionPeriod: 10m0s
  healthChecks: Enabled
  logSeverityScreen: Info
  prometheusMetricsPort: 9094
status:
  environmentVars:
    DATASTORE_TYPE: kubernetes
    ENABLED_CONTROLLERS: node
  runningConfig:
    controllers:
      node:
        hostEndpoint:
          autoCreate: Disabled
          syncLabels: Disabled
        etcdV3CompactionPeriod: 10m0s
        healthChecks: Enabled
        logSeverityScreen: Info

```

Проверки работоспособности можно отключить, если в этом нет необходимости

Порт, через который контроллер Calico экспортирует свои метрики Prometheus. Номер порта можно изменить, если понадобится

Интересно отметить, что Calico хранит свою конфигурацию внутри кластера Kubernetes в виде пользовательского объекта своего собствен-

ного типа. Впрочем, Antrea действует точно так же. Мы можем проверить содержимое кластера Antrea, запустив сценарий `kind-local-up.sh`:

```
$ kind delete cluster --name=kcalico ← Удаляет предыдущий кластер
$ C=antrea CONFIG=conf.yaml ./kind-local-up.sh )
```

Создает новый кластер с Antrea в роли провайдера CNI

Через несколько секунд можно посмотреть, какие объекты конфигурации использует Antrea, подобно тому, как сделали это с Calico:

```
$ kubectl get crd
NAME
antreaagentinfos.clusterinformation.antrea.tanzu.vmware
antreacontrollerinfos.clusterinformation.antrea.tanzu.
clusternetworkpolicies.security.antrea.tanzu.vmware.co
externalentities.core.antrea.tanzu.vmware.com
networkpolicies.security.antrea.tanzu.vmware.com
tiers.security.antrea.tanzu.vmware.com
traceflows.ops.antrea.tanzu.vmware.com
```

Пользовательские типы объектов NetworkPolicy: пример любви производителей к CRD

Если посмотреть на определения CRD в Calico и Antrea, то можно увидеть много общих черт, одна из которых – сетевые политики. NetworkPolicy API в Kubernetes поддерживает не все возможные сетевые политики, предлагаемые конкретными CNI. Например, политика PortRange (добавленная в Kubernetes v1.21) в течение некоторого времени оставалась политикой, уникальной для Calico и Antrea. Однако Calico, и Antrea имеют свои определения ресурсов сетевых политик, поэтому пользователи могут создавать новые объекты NetworkPolicy специально для этих конкретных CNI. Определения CRD позволяют точно дифференцировать продукты без создания инструментов для управления механизмами конкретного производителя. Например, объект NetworkPolicy k8s.io можно отредактировать с помощью директивы `kubectl edit` точно так же, как любой другой CRD.

Желающие узнать больше о конкретных сетевых политиках, которые расширяют возможности сетевой безопасности Kubernetes, смогут найти дополнительную информацию на страницах <http://mng.bz/aD9Y> и <http://mng.bz/g4mn>. Однако если вы не знакомы с основными Kubernetes NetworkPolicy API, то мы советуем сначала изучить их на странице <http://mng.bz/enBZ>.

Обратите внимание, что создание, редактирование или удаление объектов NetworkPolicy для Calico или Antrea приводит к немедленному созданию правил брандмауэра. Однако редактирование других CRD может не вызывать немедленных изменений в конфигурациях приложений, и эти изменения не будут реализованы до перезапуска соответствующих модулей Calico или Antrea. То есть даже притом, что CRD позволяют расширить сервер Kubernetes API, они не дают никаких гарантий относительно реализации ваших новых конструкций.

В предыдущем примере мы развернули Calico CNI с помощью файла YAML, с которым связано несколько конфигурационных объектов. Как вариант, его развернуть с помощью инструмента `operator`, разработанного в Tigera (<https://github.com/tigera/operator>), который автоматически создает и обновляет YAML-манифесты Calico. Также для применения настроек в режиме реального времени мы могли бы установить инструмент `calicoctl`.

Для развертывания Antrea в предыдущих главах мы тоже использовали манифест YAML. Развертывание кластера с Antrea (так же как с Calico) включает создание нескольких компонентов конфигурации, находящихся внутри кластера (<http://mng.bz/11qa>).

Теперь мы исследовали многие аспекты управления приложениями Kubernetes. А так как для этой цели постоянно создаются новые инструменты, считайте этот обзор началом вашего собственного исследования путей масштабирования и управления большими парками приложений в промышленном окружении. Во многих случаях, когда используются несложные рабочие процессы развертывания приложений в Kubernetes, для их автоматизации может оказаться достаточно `ytt`.

15.6 *Tanzu Community Edition: пример комплексного набора инструментов Carvel*

Tanzu Community Edition (TCE) – отличный способ исследовать Cluster API и проект Image Builder. TCE широко используется в Carvel для определения невероятно сложных конфигураций кластеров и управления парками микросервисов, которые могут обновляться и модифицироваться конечными пользователями. В основе этого набора лежит семейство утилит Carvel.

Если вам интересно посмотреть, как на практике используются `kapp`, `imgpkg`, `ytt` и другие инструменты Carvel, посетите репозитории <https://github.com/vmware-tanzu/tce> и <https://github.com/vmware-tanzu/tanzu-framework>. В них вы найдете набор инструментов для установки дистрибутива VMware Tanzu Kubernetes с открытым исходным кодом. В этом дистрибутиве:

- `ytt` устанавливает и определяет сложные шаблоны кластеров, используя спецификацию Kubernetes Cluster API. Например, `ytt` заменяет файлы спецификации кластера для Windows (которые вручную устанавливают агента Antrea как процесс Windows) спецификациями кластера для Linux. Со временем может появиться поддержка других концепций Cluster API. На момент написания этих строк соответствующие примеры были доступны по адресу <http://mng.bz/p2Z0>. Утилита `ytt` применяет по одному различные файлы в этих каталогах и создает единый

- массивный файл YAML, определяющий схему всего кластера;
- `kapp` и `kapp-controller` согласовывают все и вся, от спецификаций CNI до различных дополнительных приложений, используемых в этих дистрибутивах;
- `imgpkg` и `vendir` (которые мы не рассматривали) используются для различных задач упаковки контейнеров и управления их выпуском.

Если вас заинтересовали утилиты Carvel, подключайтесь к каналу `#carvel` в Kubernetes Slack (slack.k8s.io). Там вы найдете активное сообщество разработчиков, готовых помочь вам по общим и конкретным вопросам применения этих утилит.

Обучающие видеоматериалы Antrea LIVE

Полное введение в различные аспекты инструментов Carvel, в том числе описание концепций, заимствованных из Antrea, доступно в серии обучающих видеоматериалов Antrea LIVE. Список доступных видеоматериалов доступен на antrea.io/live. Многие темы этой книги, в том числе метрики Prometheus, провайдеры CNI и т. д., были освещены в этих видеоматериалах.

Итоги

- Простейший способ управления приложениями в Kubernetes – использовать `kubectl` и большой файл YAML, но этот подход быстро становится слишком утомительным. Существует множество отличных инструментов, которые помогут вам справиться с объемными файлами YAML. Один из них, который мы рассмотрели в этой главе, – `ytt`.
- В наборе Carvel есть несколько инструментов, предлагающих высокоуровневую возможность оркестрации приложений в Kubernetes.
- С помощью `ytt` или `kustomize` можно легко и просто реализовать настройку файлов YAML.
- `ytt` может сопоставлять произвольные части файла YAML, анализируя предложения `overlay/match` в файле оверлея, применяемого после чтения исходных файлов, и строить новый файл YAML для Kubernetes.
- С коллекциями разрозненных ресурсов YAML можно обращаться как с одним приложением, используя такие инструменты, как `kapp` или `helm`.
- Чтобы упаковать приложение с состоянием без создания оператора, можно использовать такой инструмент, как `kapp-controller`, который управляет коллекциями ресурсов приложения с учетом изменения их состояния во времени. Этот подход не дает такой же

гибкости, как создание полноценного оператора, зато его можно реализовать за считанные секунды и использовать многие из тех же преимуществ.

- С помощью операторов можно определять API более высокого уровня. Эти API поддерживают конкретный жизненный цикл вашего приложения и обычно включают возможность добавления клиента Kubernetes в контейнер.
- Calico и Antrea реализуют шаблон Operator (Оператор) для создания очень сложных расширений Kubernetes API, позволяющих управлять их конфигурацией путем создания и редактирования ресурсов Kubernetes.
- Инструменты Carvel и многие другие темы из этой книги освещаются в различных видеоматериалах Antrea LIVE, выложенных на YouTube. Их список можно найти по адресу antrea.io/live.

Предметный указатель

Символы

- allocate-node-cidrs параметр, 136
- authorization-mode, 320
- cgroup-driver флаг, 95
- config флаг, 226
- .dockerconfigjson значение, 240
- encryption-provider-config параметр, 291
- j флаг, 99
- resources раздел, 95
- seccomp-default флаг, 226

A

- A записи, 244
- allocatable структура данных, 116
- antctl, 163
- Antrea, 142
 - архитектура плагина CNI, 143
 - настройка кластера, 157
 - организация сети в Kubernetes с OVS и, 149
 - провайдеры CNI и kube-proxy в разных ОС, 152
- apiVersion раздел определения, 260
- AppArmor, 310
- arp команда, 159
 - маршруты, 161
 - OVS (Open vSwitch), 163

- arp -на команда, 159
- Authn и Authz, 327
 - доступ к облачным ресурсам, 328
 - учетные записи сервисов IAM, 327
 - частные серверы API, 329

B

- blkio контрольная группа, 115
- brd сервис, 160
- burstable папка, 111

C

- Calico, 142
 - архитектура плагина CNI, 143
 - провайдеры CNI и kube-proxy в разных ОС, 152
 - установка провайдера CNI, 146
- calicoctl, 163
- calicoctl инструмент, 362
- calico_node контейнер, 160
- CAR теорема, 287
- Carvel набор инструментов, 347
 - исправление файлов приложения с помощью ytt, 349
 - развертывание приложения Guestbook и управление им, 352
 - разделение ресурсов на отдельные файлы, 347

- создание оператора karp
 - для упаковки приложения
 - и управления им, 355
- Tanzu Community Edition (TCE), 362
- Cassandra, 217
- cat команда, 226
- CCM (Cloud Controller Manager – облачный диспетчер контроллеров), 60
- CCM (Cloud Controller Managers – облачные диспетчеры контроллеров), 269
 - контроллер маршрутов, 270
 - контроллер сервисов, 271
 - контроллер узлов, 270
- cgroups
 - классы QoS, 120
 - создание путем настройки ресурсов, 121
 - модули Pod простаивают до завершения подготовительных операций, 104
 - мониторинг ядра Linux с помощью Prometheus, 122
 - исследование простоев в Prometheus, 130
 - метрики, 123
 - создание локального сервиса мониторинга, 126
 - ограничение потребления процессора, 94
 - процессы и потоки в Linux, 106
 - контрольные группы для процессов, 110
 - процессы systemd и init, 108
 - реализация контрольных групп для обычного модуля Pod, 113
 - тестирование, 114
 - управление ресурсами со стороны kubelet, 116
 - управление со стороны kubelet, 115
- chroot команда, 87
- CLUSTER параметр, 155
- ClusterFirst Pod DNS, 254
- ClusterIP поле, 249
- CNAME записи, 244
- CNCF (Cloud Native Computing Foundation), 344
- CNCF (Cloud Native Computing Foundation) фонд, 61
- CNI (Container Network Interface – сетевой интерфейс контейнеров)
 - в разных ОС, 152
 - и kube-proxy, 134
 - организация сети в Kubernetes с OVS и Antrea, 149
 - плагины, 142
 - архитектура, 143
 - установка провайдера Calico, 146
 - провайдеры, 142
 - программно-определяемая сеть (Software-Defined Networking, SDN), 132
 - kube-proxy
 - плоскость данных, 138
 - NodePort, 140
- containerManager процедура, 231
- container-runtime параметр, 236
- container-runtime-endpoint флаг, 239
- Contour, входные контроллеры, 172
- CoreDNS
 - вышестоящий сервер имен, 254
 - разбор конфигурации, 255
- CRD (Custom Resource Definition – определения пользовательских ресурсов), 261
- CRD (Custom Resource Definition – пользовательское определение ресурса), 355
- CRD (Custom Resource Definition – собственные определения ресурсов), 59
- CRI (Container Runtime Interface – интерфейс среды выполнения контейнеров), 50, 221, 224
 - вызов, 237
 - контейнеры и образы, 233
 - процедуры, 236
 - GenericRuntimeManager, 237
- CSI (Container Storage Interface), 73
- CSI (Container Storage Interface – интерфейс контейнерного хранилища), 188

динамические хранилища, 204
драйверы хранилищ, 193
как спецификация, работающая
внутри Kubernetes, 191
контроллер, 194
не в Linux, 196
обзор действующих драйверов, 194
привязка точек монтирования, 194
пример hostPath, 214
проблема внутреннего
провайдера, 190
CSI (Container Storage Interface –
интерфейс хранилища для
контейнеров), 62
curl команда, 59, 94, 177

D

diff инструмент, 166
DNS, 243
записи NXDOMAIN, A, AAAA
и CNAME, 244
и StatefulSet, 248
постоянные записи DNS, 250
развертывание с несколькими
пространствами имен для
изучения свойств модуля
DNS, 250
DNS для автономных
сервисов, 249
модулям Pod нужен внутренний
DNS, 246
CoreDNS, разбор конфигурации, 255
resolv.conf файл, 252
и маршрутизация, 252
CoreDNS, 254
docker exec команда, 159
docker run команда, 42

E

emptyDir
для быстрого доступа на запись, 211
тип, 201
EnsureImageExists функция, 237, 241
etcd
балансировка нагрузки на уровне
клиента, 289

быстрая проверка
работоспособности, 280
журнал упреждающей записи, 287
запуск в окружении, отличном от
Linux, 294
интерфейс с Kubernetes, 285
как хранилище данных, 281
механизм аренды и
блокировки, 230
мониторинг производительности
с помощью Prometheus, 274
настройка, 279
 вложенная виртуализация, 279
 kubeadm, 280
настройка клиента в кластере
kind, 293
ограничения по размеру, 290
операция fsync, 283
производительность и
отказоустойчивость в глобальном
масштабе, 292
размеры ограничений, 290
строгая согласованность, 283
шифрование хранимых данных, 291
etcd v3 и v2, 281
externalTrafficPolicy аннотация, 137

F

fast класс хранилищ, 205
Flannel, 142
freezer контрольная группа, 115
fsync операция, 283

G

Gatekeeper, 335
getDefaultConfig() функция, 263
getImage метод, 239
go vet команда, 302
grep команда, 75
Guestbook приложение, 346, 352

H

hostNetwork пространство имен, 93
hostPath, 214, 215

когда следует использовать, 216
модель хранения в Kubernetes, 219
Cassandra, 217

I

ImagePullSecret, 240
ImageService интерфейс, 239
init контейнер, 119
install команда, 77
ip команда, 159
 трассировка движения данных
 активных контейнеров с помощью
 tcpdump, 164
 IP-туннель и его использование
 провайдерами CNI, 160
 OVS (Open vSwitch), 163
IP-туннели, 160
ip a команда, 92, 151, 159
ip route команда, 99, 150
iptables, 98, 166
 как сетевые политики изменяют
 правила CNI, 167
 iptables-save и инструмент diff, 166
iptables команда, 73
iptables-save, 166
iptables-save | grep hostnames
команда, 98

J

JSONPath, 84

K

kapp, 355
kapp инструмент, 352
kapp-controller инструмент, 355
kapp-controller проект, 346
KCM (Kubernetes Controller
Manager – диспетчер контроллеров
Kubernetes), 181
kiam инструмент, 328
kind
 и примитивы Kubernetes, 71
 настройка, 76
 создание PVC в кластере kind, 185

kind кластер, 86, 108, 158, 185, 224,
258, 303, 359
kind контейнер, 225
kind окружение, 155
kind-local-up.sh сценарий, 361
kube2iam инструмент, 328
kubeadm, 280
kube-apiserver, 57
kube-controller-manager, 60
kubectrl, 56
 контроллеры инфраструктуры, 59
 kube-apiserver, 57
 kube-scheduler, 58
kubectrl клиент, 171
kubectrl команда, 46, 222, 260, 351
kubectrl apply -f deployment.yaml
команда, 56
kubectrl edit директива, 361
kubectrl get nodes команда, 222
kubectrl get nodes -o yaml команда, 222
kubectrl get po команда, 47
kubectrl get sc команда, 185
kubectrl port-forward команда, 79
kubectrl scale команда, 64
kube-dns модуль Pod, 98
kube-dns Pod правила конечных
точек, 100
kubeGenericRuntimeManager
структура, 238
kubelet, 50
 запуск как программы, 229
 интерфейсы, 237
 внутренний интерфейс
 Runtime, 237
 интерфейс ImageService, 239
 передача ImagePullSecret, 240
контрольные группы, 115
основы, 224
 конфигурационные параметры
 и API, 225
 соглашения среды выполнения
 контейнеров, 224
 стандарты среды выполнения
 контейнеров, 224
создание модулей Pod, 228
жизненный цикл узла, 230

- механизм аренды и блокировки
 - в etcd, 230
 - приостановленный контейнер, 235
 - узлы, 222
 - управление жизненным циклом Pod, 231
 - управление ресурсами, 116
 - CRI, контейнеры и образы, 233
 - kubelet программа, 229
 - kubeletFlags структура, 226
 - kube-node-lease пространство имен, 64
 - kube-proxu, 98, 134, 136
 - в разных ОС, 152
 - и iptables, 166
 - плоскость данных, 138
 - реализация политик, 170
 - iptables-save и инструмент diff, 166
 - NodePort, 140
 - Kubernetes, 209
 - базовая основа, 29
 - варианты организации хранилищ, 209
 - возможности, 32
 - и модули Pod, 49
 - и проблема дрейфа инфраструктуры, 26
 - интерфейс с etcd, 285
 - когда не стоит использовать, 38
 - компоненты и архитектура, 34
 - обзор основных терминов, 25
 - определение инфраструктурных правил в файлах YAML и JSON, 31
 - пример интернет-магазина, 37
 - пример кластера, 33
 - пример решения для благотворительности, 37
 - примитивы Linux, 78
 - зависимости модулей Pod от Linux, 81
 - предварительные условия для запуска Pod, 78
 - программно-определяемая сеть (Software-Defined Networking, SDN), 132
 - секреты, 209
 - обзор, 210
 - создание простого Pod с пустым томом для быстрой записи, 211
 - советы по безопасности, 341
 - Kubernetes API, 35
 - Kubernetes API механизм, 227
 - kube-scheduler, 58
 - kube-system пространство имен, 257, 333
-
- ## L
-
- latest ter, 47
 - Linux
 - виртуальная файловая система (VFS), 181
 - мониторинг ядра Linux с помощью Prometheus, 122
 - исследование простоев в Prometheus, 130
 - метрики, 123
 - создание локального сервиса мониторинга, 126
 - настройка kind, 76
 - пространства имен, 47
 - процессы и потоки, 106
 - контрольные группы для процессов, 110
 - реализация контрольных групп для обычного модуля Pod, 113
 - ListImages метод, 239
 - LoadBalancer тип, 137
 - ls инструмент, 73
-
- ## M
-
- match раздел, 336
 - mount команда, 73, 84, 89
 - MountCniBinary метод, 147
-
- ## N
-
- Node объект API, 51
 - nodeName, 58
 - NodePort, 140
 - nsenter инструмент, 79
 - NXDOMAIN, 244

О

OPA (Open Policy Agent – агент открытой политики), 335
Open vSwitch (OVS), виртуальный коммутатор, 137
ovs-ofctl, 171
OVS (Open vSwitch), 157, 163
ovs-vsctl, 163

Р

PersistentVolumeClaim (PVC), 60, 180, 201
PersistentVolume (PV), 60, 180, 201, 203
Pod (модуль)

- безопасность, 302
 - граница безопасности, 310
 - модули Pod с привилегиями root, 309
 - расширение привилегий и возможностей, 305
- виртуальная файловая система (VFS) в Linux, 181
- возможности, 305
- запуск, 80
- зеркальные модули Pod, 110
- интерфейс контейнерного хранилища (Container Storage Interface, CSI), 188
 - драйверы хранилищ, 193
 - как спецификация, работающая внутри Kubernetes, 191
 - контроллер, 194
 - не в Linux, 196
 - обзор действующих драйверов, 194
 - привязка точек монтирования, 194
 - проблема внутреннего провайдера, 190
- использование в реальном мире, 96
 - как kube-proxy реализует сервисы Kubernetes, 98
 - проблема сети, 97
 - проблемы, 100
 - kube-dns, 98

контекст безопасности, 303
контрольные группы

- классы QoS, 120
- модули Pod простаивают до завершения подготовительных операций, 104
- мониторинг ядра Linux с помощью Prometheus, 122
- процессы и потоки в Linux, 106
- тестирование, 114
- управление ресурсами со стороны kubelet, 116
- управление со стороны kubelet, 115

плоскость управления и пример веб-приложения, 55
политики безопасности Pod (PSP), 307
потребность во внутреннем DNS, 246
привилегии, 305
пример веб-приложения, 42

- инфраструктура, 44
- эксплуатационные требования, 45

примитивы Linux, 73

- зависимости от Linux, 81
- использование в Kubernetes, 78
- как инструменты управления ресурсами, 74
- комбинирование файлов, 75
- настройка kind, 76
- файлы, 74

пространства имен Linux, 47
сетевое пространство имен, создание, 92
создание, 86

- защита процесса с помощью unshare, 91
- изолированного процесса с командой chroot, 87
- использование mount для передачи данных, 89
- ограничение потребления процессора с cgroups, 94
- проверка работоспособности, 93
- создание раздела resources, 95

создание веб-приложения
с помощью kubectl
контроллеры
инфраструктуры, 59
kube-apiserver, 57
kube-scheduler, 58
создание в kubelet, 228
создание с помощью kubelet
приостановленный
контейнер, 235
создание PVC в кластере kind, 185
управление жизненным циклом
в kubelet, 231
хранилища в Kubernetes, 182
Kubernetes, инфраструктура, 49
Node объект API, 51
PersistentVolumeClaim (PVC), 201
PersistentVolume (PV), 201
prepareSubpathTarget функция, 70
Prometheus, 125
мониторинг производительности
etcd, 274
мониторинг ядра Linux, 122
исследование простоев
в Prometheus, 130
метрики, 123
создание локального сервиса
мониторинга, 126
ps программа, 79
PSP, политики безопасности Pod, 307
pstree команда, 107
PullImage метод, 239, 241
PVC (PersistentVolumeClaim) создание
в кластере kind, 185

Q

QoS (Quality of Service – качество обслуживания), 120

R

RBAC API, 321
RBAC (Role-Based Access Control – управление доступом на основе ролей), 320
RemoveImage метод, 239

resolv.conf файл, 252
CoreDNS, 254
Runtime внутренний интерфейс, 237

S

SDN (Software-Defined Networking – программно-определяемая сеть), 49, 132
seccomp, 310
SELinux, 310
setupNode функция, 229
socat команда, 79
Sonobuoy, 155
настройка кластера
с CNI-провайдером Antrea, 157
трассировка движения данных
модулей Pod в кластере, 156
sonobuoy status команда, 156
status раздел, 223
StorageClass, 180
StringData поле, 210
swapoff команда, 79
system процесс, 109
systemd команда, 79

T

Tanzu Community Edition (TCE), 362
tcpdump, 165
test-bed пространство имен, 331
tmpfs (временное хранилище файлов), 202

U

unshare команда, 73, 91, 109

W

web пространство имен, 331
write_files директива, 209

Y

ytt инструмент, 349
ytt команда, 352

А

Автоматическое
масштабирование, 65
Автономные сервисы, 249
Агент открытой политики (Open
Policy Agent, OPA), 335
Архитектура сетевых плагинов, 143
Атаки на ресурсы, 316

Б

Балансировщики нагрузки сети, 335
Безопасность, 297
агент открытой политики (Open
Policy Agent, OPA), 335
балансировщики нагрузки, 335
доступ к облачным ресурсам, 328
коллективной аренды
(multi-tenancy), 338
контейнеров, 298
контроль, 299
линтеры, 302
обновление, 299
происхождение, 301
модули Pod с привилегиями
root, 309
радиус взрыва, 297
вторжение, 298
уязвимости, 298
сервера API, 320
ресурсы и подресурсы, 323
субъекты и RBAC, 325
управление доступом на основе
ролей (RBAC), 320
RBAC API, 321
сервисная сетка, 334
сети, 330
узла, 313
атаки на ресурсы, 316
единицы измерения объема
памяти, 317
единицы измерения объема
хранилища, 318
единицы измерения потребления
процессора, 317
изолированные среды
выполнения контейнеров, 315

неизменяемые ОС, 314
применение изменений, 314
сертификаты TLS, 313
сети хостов и модулей Pod, 318
учетные записи сервисов IAM, 327
частные серверы API, 329
Authn и Authz, 327
Kubernetes, советы, 341
Бюджет распределяемых
ресурсов, 117

В

Веб-обработчики (webhook), 58
Веб-приложения
пример, 42
инфраструктура, 44
плоскость управления, 55
эксплуатационные требования, 45
создание с помощью kubectl, 56
контроллеры инфраструктуры, 59
kube-apiserver, 57
kube-scheduler, 58
Ведущие узлы, 54
Вложенная виртуализация, 279
Внешний инструмент подготовки
(external provisioner), 193
Входные контроллеры, 172
настройка простого модуля Pod
с веб-сервером, 174
настройка Contour и кластера
kind, 173
Высокая доступность приложений, 63
автоматическое
масштабирование, 65
управление затратами, 66

Д

Динамическая подготовка, 200, 205
Динамические хранилища, 199
интерфейс контейнерного
хранилища(CSI), 204
локальное хранилище в сравнении
с emptyDir, 201
провайдеры, 212
PersistentVolume (PV), 203

Диспетчер контроллеров
 контроллер конечной точки, 268
 контроллер репликации, 268
 контроллер узла, 268
 учетные записи сервисов
 и токены, 269
 хранилище, 267
Диспетчер контроллеров Kubernetes
(Kubernetes Controller Manager,
KCM), 181
Драйверы хранилищ, 193

Е

Единицы измерения
 объема памяти, 317
 объема хранилища, 318
 потребления процессора, 317

Ж

Журнал упреждающей записи, 287

З

Зеркальные модули Pod, 110

И

Изолированные среды выполнения
контейнеров, 315
Интерфейс контейнерного
хранилища (Container Storage
Interface, CSI), 188
 динамические хранилища, 204
 драйверы хранилищ, 193
 как спецификация, работающая
 внутри Kubernetes, 191
 контроллер, 194
 не в Linux, 196
 обзор действующих драйверов, 194
 привязка точек монтирования, 194
 пример hostPath, 214
 проблема внутреннего
 провайдера, 190
Интерфейс среды выполнения
контейнеров (Container Runtime
Interface, CRI), 50, 221, 224

 вызов, 237
 контейнеры и образы, 233
 процедуры, 236
 GenericRuntimeManager, 237
Интерфейс хранилища для
контейнеров (Container Storage
Interface, CSI), 62
Интерфейсы
 kubelet, 237
 внутренний интерфейс
 Runtime, 237
 интерфейс ImageService, 239
 передача ImagePullSecret, 240

Инфраструктура
 и модули Pod, 49
 контроллеры, 59
Исследование особенностей
маршрутизации в разных
провайдерах CNI с помощью команд
arp и ip, 159
 IP-туннели, 160

К

Качество обслуживания (Quality of
Service, QoS), 120
Классы хранилищ, 206
Классы QoS, 120
 создание путем настройки
 ресурсов, 121
Коллективная аренда
(multi-tenancy), 338
Контейнеры, 25
 без дистрибутива, 247
 безопасность, 298
 контроль, 299
 линтеры, 302
 обновление, 299
 происхождение, 301
 и образы, 27
 приостановленный контейнер, 235
 среда выполнения, 224
Контроллер
 доступа (admission controller), 188
 конечной точки, 268
 репликации, 268
 узла, 268

Контроллеры, 61
Контрольные группы
 классы QoS, 120
 создание путем настройки
 ресурсов, 121
модули Pod простаивают до
завершения подготовительных
операций, 104
мониторинг ядра Linux с помощью
Prometheus, 122
 исследование простоев
 в Prometheus, 130
 метрики, 123
 создание локального сервиса
 мониторинга, 126
процессы и потоки в Linux, 106
 контрольные группы для
 процессов, 110
 процессы systemd и init, 108
 реализация контрольных
 групп для обычного модуля
 Pod, 113
тестирование, 114
управление ресурсами со стороны
kubelet, 116
управление со стороны
kubelet, 115

Л

Линтеры, 302
Локальные хранилища, 201

М

Маршрутизация, 252
Маршруты, 161
Мастеры, 54
Масштабирование, 63
 автоматическое, 65
Механизм аренды и блокировки
в etcd, 230
Микросервисы, 346
Модули Pod с привилегиями
root, 309
Модуль Pod с веб-сервером, 174
Моментальные снимки, 218

Н

Набор алгоритмов шифрования, 313
Настройка etcd, 279
 вложенная виртуализация, 279
 kubeadm, 280
Неизменяемые ОС, 314

О

Облачные диспетчеры контроллеров
(Cloud Controller Managers, CCM), 269
 контроллер маршрутов, 270
 контроллер сервисов, 271
 контроллер узлов, 270
Облачные ресурсы, 328
Облачные API, 327
Облачный диспетчер контроллеров
(Cloud Controller Manager, CCM), 60
Обновление и повторный запуск Pod
(модули), проблемы, 101
Обновление контейнеров, 299
Образы, 233
Образы без дистрибутива, 247
Операторы Kubernetes, 359
Определения пользовательских
ресурсов (Custom Resource Definition,
CRD), 261
Основные термины Kubernetes, 25

П

Пакеты, 160
Перепланирование, 233
Планирование Pod (модули),
проблемы, 101
Планировщик
 компоненты, 267
 обзор, 261
Плоскость данных, 138
Плоскость управления, 54, 63
 автоматическое
 масштабирование, 65
 диспетчер контроллеров, 267
 контроллер конечной точки, 268
 контроллер репликации, 268
 контроллер узла, 268

- учетные записи сервисов и токены, 269
- хранилище, 268
- и пример веб-приложения, 55
- обзор, 258
- сервер API, 259
 - объекты API и пользовательские ресурсы, 259
 - определения пользовательских ресурсов (CRD), 261
 - планировщик, 261
- управление затратами, 66
- Плотное размещение модулей Pod, 66
- Подкачка, 117
 - и kubelet, 117
- Политики безопасности Pod (PSP), 307
- Пользовательское определение ресурса (Custom Resource Definition, CRD), 355
- Постоянные записи DNS, 250
- Потоки, 106
- Привилегии, 305
- Привязка точек монтирования, 70, 194
- Приложения на основе микросервисов, 346
- Применение изменений на узлах, 314
- Примитивы Linux, 73
 - использование в Kubernetes, 78
 - зависимости от Linux, 81
 - предварительные условия для запуска Pod, 78
- как инструменты управления ресурсами, 74
- файлы
 - возможность комбинирования, 75
 - все существующее является, 74
- Приостановленные контейнеры, 105
- Приостановленный контейнер, 69, 235
- Проблема дрейфа инфраструктуры, и Kubernetes, 26
- Провайдеры
 - в разных ОС, 152
 - установка провайдера Calico, 146

- Программно-определяемая сеть (Software-Defined Networking, SDN), 49, 132
- Процессы, 106
 - контрольные группы для, 110
 - systemd и init, 108

Р

- Радиус взрыва, 297
 - вторжение, 298
 - уязвимости, 298
- Развертывание с несколькими пространствами имен, 250
- Распространение монтирования (mount propagation), 194
- Ресурсы, инструменты управления, 74

С

- Секреты, 209
 - обзор, 210
 - создание простого Pod с пустым томом для быстрой записи, 211
- Серверы-снежинки, 315
- Сервер API, 259
 - безопасность, 320
 - ресурсы и подресурсы, 323
 - субъекты и RBAC, 325
 - управление доступом на основе ролей (RBAC), 320
 - RBAC API, 321
- объекты API и пользовательские ресурсы, 259
- определения пользовательских ресурсов (CRD), 261
- планировщик
 - компоненты, 267
 - обзор, 261
- Сервисная сетка, 334
- Сервисы автономные, 249
- Сертификаты TLS, 313
- Сетевой интерфейс контейнеров (Container Network Interface, CNI)
 - в разных ОС, 152
 - и kube-proxy, 134

организация сети в Kubernetes
с OVS и Antrea, 149
плагины, 142
 архитектура, 143
 установка провайдера
 Calico, 146
провайдеры, 142
программно-определяемая сеть
(Software-Defined Networking,
SDN), 132
kube-proxu
 плоскость данных), 138
 NodePort, 140
Сетевые плагины, 142
 архитектура, 143
 организация сети в Kubernetes
 с OVS и Antrea, 149
 провайдеры CNI и kube-proxu
 в разных ОС, 152
 установка провайдера Calico, 146
Сети, безопасность
 агент открытой политики (Open
 Policy Agent, OPA), 335
 балансировщики нагрузки, 335
 коллективной аренды
 (multi-tenancy), 338
Сети хостов, 318
Сеть, пространство имен,
создание, 92
Собственные определения
ресурсов (Custom Resource Definition,
CRD), 59
Согласованность etcd как хранилище
данных, 283
Соглашения, среды выполнения
контейнеров, 224
Стандарты, среды выполнения
контейнеров, 224

Т

Тестирование контрольных
групп, 114
Трассировка движения данных
активных контейнеров с помощью
tcpdump, 164
модулей Pod в кластере, 156

У

Узлы

 безопасность, 313
 атаки на ресурсы, 316
 единицы измерения объема
 памяти, 317
 единицы измерения объема
 хранилища, 318
 единицы измерения потребления
 процессора, 317
 изолированные среды
 выполнения контейнеров, 315
 неизменяемые ОС, 314
 применение изменений, 314
 сертификаты TLS, 313
 сети хостов и модулей Pod, 318
жизненный цикл, 230
интерфейс, 195
kubelet, 222

Управление доступом на основе
ролей (Role-Based Access Control,
RBAC), 320

Управление затратами, 66

Установка

 набора инструментов Carvel, 347
 исправление файлов приложения
 с помощью ytt, 349
 развертывание приложения
 Guestbook и управление им, 352
 разделение ресурсов на
 отдельные файлы, 347
 создание оператора карп
 для упаковки приложения
 и управления им, 355
 Tanzu Community Edition (TCE), 362
приложений в Kubernetes, 344
приложений на основе
микросервисов, 346
приложения Guestbook, 346
провайдера Calico, 146

Устранение проблем

в крупномасштабных сетях
входные контроллеры, 172
исследование особенностей
маршрутизации в разных
провайдерах CNI с помощью
команд `arp` и `ip`, 159

kube-proxy и iptables, 166
Sonobuoy, 154
настройка кластера
с CNI-провайдером Antrea, 157
трассировка движения данных
модулей Pod в кластере, 156
Учетные записи сервисов
и токены, 269
Учетные записи сервисов
IAM, 327
Уязвимости, 298

Ф

Файлы
возможность комбинирования, 75
все сущее является, 74

Х

Хранение, Pod (модули),
проблемы, 100
Хранилища
варианты организации хранилищ
в Kubernetes, 209
виртуальная файловая система
(VFS) в Linux, 181
в Kubernetes, 182
динамическая подготовка, 205
динамические хранилища, 199
интерфейс контейнерного
хранилища(CSI), 204
локальное хранилище
в сравнении с emptyDir, 201
провайдеры, 212

PersistentVolume (PV), 203
диспетчер контроллеров, 267
драйверы хранилищ, 193
интерфейс контейнерного
хранилища (Container Storage
Interface, CSI), 188
драйверы хранилищ, 193
как спецификация, работающая
внутри Kubernetes, 191
контроллер, 194
не в Linux, 196
обзор действующих
драйверов, 194
привязка точек
монтирования, 194
проблема внутреннего
провайдера, 190
локальные хранилища, 201
модель хранения в Kubernetes, 219
Хранилище данных, etcd, как
операция fsync, 283
строгая согласованность, 283
Хранилище данных, etcd, как , 281

Ц

Циклы управления, 54

Ч

Частные серверы API, 329

Ш

Шумные соседи явление, 44

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@aliants-kniga.ru**.

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: <http://www.galaktika-dmk.com/>.

Джей Вьяс, Крис Лав

Kubernetes изнутри

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*

Перевод *Киселев А. Н.*

Корректор *Абросимова Л. А.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.

Усл. печ. л. 30,71. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com