Контейнеры без демона и привилегий

Podman
В ДЕЙСТВИИ

Дэниэл Уолш

MANNING

Podman in Action

SECURE, ROOTLESS CONTAINERS FOR KUBERNETES, MICROSERVICES, AND MORE

DANIEL WALSH

MANNING
SHELTER ISLAND

Podman в действии

ДЭНИЭЛ УОЛШ

Выпущено
при поддержке
КРОК

ПИТЕР®

Санкт-Петербург • Москва • Минск

2024

ББК 32.988.02-18
УДК 004.738.5+004.9
У63

Уолш Дэниэл

У63 Podman в действии. — СПб.: Питер, 2024. — 352 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-2111-3

Пришло время обновить свой контейнерный движок! Менеджер контейнеров Podman обеспечивает гибкое управление слоями образов и полную совместимость с Kubernetes, а также дает возможность пользователям без прав администратора создавать, запускать непривилегированные контейнеры и управлять ими. OCI-совместимая поддержка Docker API позволяет перевести существующие контейнеры на Podman, не ломая свои скрипты и не меняя привычного порядка работы.

«Podman в действии» познакомит вас с менеджером контейнеров Podman. Простые объяснения и примеры позволят быстро разобраться с тем, что такое контейнеры, как они работают и как управлять ими. Вы получите глубокие знания об используемых Podman компонентах Linux и даже узнаете больше о Docker. Особенно ценны соображения автора Дэна Уолша по поводу безопасности контейнеров.

Для разработчиков и системных администраторов, имеющих опыт работы с Linux и Docker.

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

ББК 32.988.02-18
УДК 004.738.5+004.9

ISBN 978-1633439689 англ.

ISBN 978-5-4461-2111-3

Краткое содержание

Часть 3
РАСШИРЕННЫЕ ВОЗМОЖНОСТИ PODMAN

Часть 4
БЕЗОПАСНОСТЬ КОНТЕЙНЕРОВ

ПРИЛОЖЕНИЯ

Оглавление

Часть 2
АРХИТЕКТУРА

Часть 3
РАСШИРЕННЫЕ ВОЗМОЖНОСТИ PODMAN

ПРИЛОЖЕНИЯ

В память о моей матери, Джоан П. Уолш

Предисловие

Я работаю в области компьютерной безопасности почти сорок лет, а в последние двадцать сосредоточился на контейнерных технологиях. Когда около десяти лет назад появился Docker, он вызвал революцию в распространении и запуске приложений в интернете. Используя Docker, я думал о том, что его можно было бы спроектировать лучше. Работа с демоном, запускаемым с правами root, а затем добавление все новых и новых демонов казались мне неправильным подходом. По моему мнению, вместо этого следовало бы использовать низкоуровневые концепции операционных систем и сделать инструмент, который запускал бы те же контейнерные приложения таким же образом, но более безопасно и требуя меньшего количества привилегий. Поэтому моя команда в Red Hat задалась целью создать инструменты, которые помогут разработчикам и администраторам запускать контейнеры самым безопасным способом. В результате этой работы появился Podman.

В начале 2000-х я начал вести блог на такие темы, как SELinux, и с тех пор пишу разные статьи. За эти годы я написал сотни статей о контейнерах и безопасности, но мне хотелось объединить все свои идеи и описать технологию Podman в одной книге, к которой я мог бы отсылать пользователей и заказчиков.

В этой книге рассказывается о Podman и о том, как с ним работать. Кроме того, она глубоко погружает читателя в изучение технологии и различных частей операционной системы Linux, которыми мы пользуемся. Поскольку я являюсь инженером по безопасности, я также посвятил пару глав описанию безопасности контейнеров. Прочитав эту книгу, вы станете лучше понимать, что такое контейнеры, как они действуют и как применять различные функции Podman. Вы даже узнаете гораздо больше о Docker. Так как популярность Podman растет и он все шире внедряется в инфраструктуру, эта книга станет для вас полезным справочником, который стоит держать под рукой.

Благодарности

Я выражаю благодарность всем людям, которые помогли мне написать эту книгу. В их числе члены команды Podman — они написали статьи, послужившие мне подспорьем в понимании технологий, в которых я не до конца разбирался, и помогли создать отличный продукт. Спасибо вам, Брент Боде (Brent Baude), Мэтт Хеон (Matt Heon), Валентин Ротберг (Valentin Rothberg), Джузеппе Скривано (Giuseppe Scrivano), Урваши Мохнани (Urvashi Mohnani), Налин Дахьябхай (Nalin Dahyabhai), Локеш Мандвекар (Lokesh Mandvekar), Милослав Трмач (Miloslav Trmac), Джейсон Грин (Jason Greene), Джон Хонс (Jhon Honce), Скотт Маккарти (Scott McCarty), Том Суини (Tom Sweeney), Эшли Куи (Ashley Cui), Эд Сантьяго (Ed Santiago), Крис Эвич (Chris Evich), Адитья Раджан (Aditya Rajan), Пол Холцингер (Paul Holzinger), Прити Томас (Preethi Thomas) и Чарли Доерн (Charlie Doern). Я также хочу поблагодарить бесчисленных разработчиков открытого исходного кода, которые сделали возможными Linux-контейнеры и Podman.

Я благодарю всю команду Manning, но особенно Тони Арритола (Toni Arritola). Тони научила меня, как лучше раскрыть мои идеи, и была отличным партнером в этом процессе. Ей пришлось иметь дело со мной, старым математиком, который никогда не умел писать, но она помогла этой книге увидеть свет.

Спасибо вам, мои рецензенты: Ален Ломп (Alain Lompo), Алессандро Кампеис (Alessandro Campeis), Аллан Макура (Allan Makura), Аманда Деблер (Amanda Debler), Андерс Бьёрклунд (Anders Björklund), Андреа Монакки (Andrea Monacchi), Камал Какар (Camal Cakar), Клиффорд Тёрбер (Clifford Thurber), Конор Редмонд (Conor Redmond), Дэвид Паккуд (David Paccoud), Дипак Шарма (Deepak Sharma), Федерико Кирхайс (Federico Kircheis), Франс Оилинки (Frans Oilinki), Гоутам Садасивам (Gowtham Sadasivam), Ибрагим Аккулак (Ibrahim Akkulak), Джеймс Лю (James Liu), Джеймс Ньика (James Nyika), Джереми Чен (Jeremy Chen), Кент Спилнер (Kent Spillner), Кевин Этьен (Kevin Etienne), Кирилл Ширинкин (Kirill Shirinkin), Космас Чатзимихалис (Kosmas

Chatzimichalis), Кшиштоф Камычек (Krzysztof Kamyczek), Ларри Кай (Larry Cai), Майкл Брайт (Michael Bright), Младен Кнежич (Mladen Knežic´), Оливер Кортен (Oliver Korten), Ричард Майнсен (Richard Meinsen), Роман Жужа (Roman Zhuzha), Руй Лю (Rui Li), Сатадру Рой (Satadru Roy), Сын Чжин Ким (Seung-jin Kim), Симеон Лейзерзон (Simeon Leyzerzon), Симона Сгуацца (Simone Sguazza), Сайед Ахмед (Syed Ahmed), Томас Пеклак (Thomas Peklak) и Вивек Вираппан (Vivek Veerappan) — ваши предложения помогли сделать эту книгу лучше.

Об этой книге

В книге «Podman в действии» описывается, как пользователи могут создавать, запускать контейнеры и управлять ими. При написании этой книги моей целью было объяснить, как без особых сложностей перенести полученные в Docker навыки на Podman, а также как просто использовать Podman, если вы никогда раньше не работали с контейнерными движками. «Podman в действии» научит вас использовать такие расширенные возможности, как поды (pod), и поможет вам на пути к созданию приложений, готовых к работе в Kubernetes. Наконец, «Podman в действии» расскажет все о функциях безопасности ядра Linux, которые применяются для изоляции контейнеров как от системы, так и от других контейнеров.

ДЛЯ КОГО ЭТА КНИГА

Книга «Podman в действии» написана для разработчиков программного обеспечения, которые хотят разобраться в том, что такое контейнеры, создавать и применять их, а также для системных администраторов, использующих контейнеры в выпускаемых продуктах. Прочитав эту книгу, вы получите более глубокое представление о том, что такое контейнеры. Для получения максимальной пользы от книги необходимо понимание процессов Linux и умение работать с его командной строкой.

Каждый, кто хочет использовать контейнеры, найдет что-то для себя в этой книге. Пользователи, хорошо знакомые с Docker, узнают о расширенных возможностях Podman, недоступных в Docker, и еще лучше разберутся в том, как работает Docker. Начинающие пользователи изучат основы работы с контейнерами и подами.

СТРУКТУРА ИЗДАНИЯ: ДОРОЖНАЯ КАРТА

«Podman в действии» состоит из четырех частей и шести приложений.

- Часть 1 «Основы» состоит из четырех глав и знакомит читателей с Podman. В главе 1 объясняется, что делает Podman, зачем он был создан и почему он важен. В следующих двух главах рассказывается об интерфейсе командной строки и о том, как использовать тома в контейнерах. В главе 4 вводится понятие подов и объясняется, как Podman работает с ними. Здесь каждый найдет что-то полезное для себя, но если у вас большой опыт работы с Docker, вам будет достаточно лишь бегло просмотреть бóльшую часть содержимого главы 2.
- Часть 2 «Архитектура» состоит из двух глав, в которых я глубоко погружаюсь в устройство Podman. Вы узнаете о непривилегированных (rootless) контейнерах и о том, как они работают. Прочитав эти главы, вы станете лучше понимать, что такое пользовательские пространства имен и безопасность непривилегированных контейнеров. Вы также узнаете, как настроить конфигурацию вашего Podman-окружения.
- Часть 3 «Расширенные возможности Podman» состоит из трех глав и выходит за рамки основ Podman. В главе 7 показано, как Podman может работать в продакшене благодаря интеграции с systemd. В ней рассказывается о запуске systemd внутри контейнера и о том, как использовать его в качестве менеджера контейнеров. Вы узнаете, как настроить серверы с контейнерами Podman, где systemd управляет жизненным циклом контейнера. Podman позволяет легко генерировать файлы systemd-юнитов, помогая отправить в продакшен ваши контейнеризованные приложения. В главе 8 вы узнаете, как с помощью Podman можно перенести контейнеры в Kubernetes. Podman поддерживает запуск контейнеров с помощью тех же YAML-файлов, которые использует Kubernetes, а также дает возможность генерировать YAML для Kubernetes из ваших контейнеров. В главе 9 я покажу, как Podman работает в качестве службы, обеспечивая удаленный доступ к контейнерам Podman. Использование Podman в качестве службы позволяет вам применять другие языки программирования и инструменты для управления контейнерами. Вы увидите, как `docker-compose` может работать с контейнерами Podman. Вы также узнаете, как использовать для взаимодействия с сервисом Podman при управлении контейнерами такие библиотеки Python, как podman-py и docker-py.
- Часть 4 «Безопасность контейнеров» состоит из двух глав, в которых речь идет о важных аспектах безопасности. В главе 10 описаны функции, используемые для обеспечения изоляции контейнеров. В этой главе анализируются такие подсистемы безопасности Linux, как SELinux, seccomp, привилегии Linux (Linux capabilities), файловые системы ядра и пространства имен

(namespaces). Затем в главе 11 рассматриваются соображения безопасности, которые я считаю лучшими практиками для максимально безопасного запуска контейнеров.

Кроме того, в книге шесть приложений, посвященных связанным с Podman темам:

- В приложении A описаны все связанные с Podman инструменты, включая Buildah, Skopeo и CRI-O.
- Приложение B подробно рассматривает различные среды выполнения OCI, доступные как для Podman, так и для Docker. В нем обсуждаются `runc`, `crun`, Kata и gVisor.
- Приложение C описывает, как установить Podman на вашу локальную систему, будь то Linux, Mac или Windows.
- В приложении D рассказывается о сообществе разработчиков открытого исходного кода Podman и о том, как к нему присоединиться.
- Приложения E и F посвящены запуску Podman на компьютерах с операционными системами Mac и Windows.

ФОРУМ LIVEBOOK

Приобретая книгу «Podman в действии», вы получаете бесплатный доступ к закрытому веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу book/podman-in-action/discussion сайта https://livebook.manning.com/. Информация о форумах Manning и правилах поведения на них размещена на https://livebook.manning.com/#!/discussion.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

АВТОР ОНЛАЙН

Вы можете подписаться на Дэна Уолша (Dan Walsh) в Twitter и GitHub @rhatdan. Он регулярно ведет блог на https://www.redhat.com/sysadmin/users/dwalsh и на нескольких других сайтах. На YouTube также много видео с выступлениями Дэна.

Об авторе

Дэниэль Уолш (Daniel Walsh) возглавляет команду, которая создала Podman, Buildah, Skopeo, CRI-O и связанные с ними инструменты. Дэн — ведущий инженер компании Red Hat, работает там с августа 2001 года. Более 40 лет он занимается компьютерной безопасностью. Дэна иногда называют «мистер SELinux», так как он руководил разработкой SELinux в Red Hat до того, как возглавил группу, занимающуюся контейнерами. Дэн получил степень бакалавра математики в Колледже Святого Креста и степень магистра компьютерных наук в Вустерском политехническом институте. В Twitter и на GitHub его можно найти по нику @rhatdan. Вы можете написать ему по адресу dwalsh@redhat.com.

Иллюстрация на обложке

Иллюстрация на обложке книги «Podman в действии» озаглавлена как «La vandale», или «Вандал», и позаимствована из коллекции Жака Грассе де Сен-Совера (Jacques Grasset de Saint-Sauveur), опубликованной в 1797 году. Каждая иллюстрация из коллекции тщательно прорисована и раскрашена вручную.

В те времена по одежде человека можно было легко определить, где он живет и каково его ремесло или положение в обществе. Издательство Manning отмечает изобретательность и инициативность IT-бизнеса обложками книг, которые основаны на богатом разнообразии региональной культуры многовековой давности, возвращаемой к жизни изображениями из коллекций, подобными этому.

О переводчике на русский язык

Дмитрий Иванов — ведущий системный инженер компании КРОК, руководитель группы системных инженеров ДРПО (департамента разработки программного обеспечения). Имеет большой практический опыт работы с решениями контейнеризации и оркестрации контейнеров.

От издательства

Мы выражаем огромную благодарность компании КРОК за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть 1

Основы

В части 1 книги я познакомлю вас с несколькими способами работы с Podman из командной строки. В главе 2 объясню, как создавать контейнеры и применять их, а также как контейнеры работают с образами. Вы узнаете, в чем разница между контейнером и образом, как сохранить контейнер в образ, а затем отправить образ в реестр, чтобы им можно было поделиться с другими пользователями.

В главе 3 я ввожу понятие тома. Тома — это механизмы, которые большинство пользователей ваших контейнерных приложений используют для хранения своих данных отдельно от приложения. В первых двух главах основное внимание уделяется использованию контейнеров и образов, что очень напоминает работу контейнеров в Docker.

В главе 4 добавляется концепция подов, аналогичная подам Kubernetes, которую Docker не поддерживает. Поды предоставляют возможность совместно использовать один или несколько контейнеров в рамках одного и того же ресурса, пространства имен и ограничений безопасности. Поды позволяют писать более сложные приложения и управлять ими как единой сущностью.

1 Podman: контейнерный движок нового поколения

В ЭТОЙ ГЛАВЕ

- ✓ Что такое Podman
- ✓ Преимущества применения Podman по сравнению с Docker
- ✓ Примеры использования Podman

Непросто приступать к этой книге, ведь имеющие разный опыт читатели подойдут к ней с разными ожиданиями. Скорее всего, у вас есть некоторый опыт работы с контейнерами, Docker или Kubernetes — или, по крайней мере, вам хочется узнать больше о Podman, потому что вы о нем слышали. Если вы пользовались Docker или знакомы с ним, вы обнаружите, что Podman в большинстве случаев работает так же, но он решает некоторые проблемы, присущие Docker. Самое главное, Podman предлагает повышенную безопасность и возможность запускать команды без привилегий root. Это означает, что вы можете управлять контейнерами с помощью Podman, не имея root-доступа или привилегий. Благодаря своему устройству Podman по умолчанию обеспечивает гораздо большую безопасность, чем Docker.

Помимо того что Podman имеет открытый исходный код (и является бесплатным), его команды, выполняемые из интерфейса командной строки (CLI), очень похожи на команды Docker. В этой книге показано, как можно использовать

Podman в качестве локального контейнерного движка для запуска контейнеров на одном узле локально или через удаленный REST API. Вы также узнаете, как находить, запускать и собирать контейнеры, используя Podman и такие инструменты с открытым исходным кодом, как Buildah и Skopeo.

1.1. О ТЕРМИНОЛОГИИ

Прежде чем продолжить, я считаю важным определиться с терминологией, которую вы встретите в книге. В мире контейнеров такие термины, как *оркестратор контейнеров, контейнерный движок* и *среда выполнения контейнеров*, часто используются как взаимозаменяемые, что обычно приводит к путанице. В следующем списке я кратко обобщаю, к чему относится каждый из этих терминов в контексте данной книги.

- *Оркестраторы контейнеров*. Программные проекты и продукты, которые управляют контейнерами на нескольких машинах или узлах. Для запуска контейнеров оркестраторы взаимодействуют с контейнерными движками. Основным оркестратором контейнеров сейчас является Kubernetes. Изначально он был разработан для взаимодействия с контейнерным движком Docker daemon, но использование Docker становится все менее актуальным, поскольку Kubernetes теперь в основном применяет CRI-O или containerd в качестве движка. CRI-O и containerd специально созданы для запуска управляемых Kubernetes контейнеров (CRI-O рассматривается в приложении A). Другие примеры оркестраторов контейнеров — Docker Swarm и Apache Mesos.
- *Контейнерные движки*. В основном используются с целью настройки контейнерных приложений для запуска на одном локальном узле. Их могут запускать непосредственно пользователи, администраторы и разработчики. Они могут запускаться из файлов systemd-юнитов при загрузке, а также контейнерными оркестраторами, такими как Kubernetes. Как уже упоминалось ранее, CRI-O и containerd — это контейнерные движки, применяемые Kubernetes для локального управления контейнерами. Они не предназначены непосредственно для пользователей. Docker и Podman — это основные контейнерные движки для разработки, управления и запуска контейнерных приложений на одной машине. Podman редко используется для запуска контейнеров для Kubernetes, поэтому Kubernetes в этой книге в общем случае не рассматривается. Buildah — еще один контейнерный движок, но он применяется только для создания образов контейнеров.
- *Среды выполнения контейнеров Open Container Initiative (OCI)*. Настраивают различные части ядра Linux и запускают контейнерное приложение. Наиболее часто используемыми средами выполнения контейнеров являются

`runc` и `crun`. Другие примеры таких сред — Kata и gVisor. Чтобы разобраться в различиях между средами выполнения контейнеров OCI, обращайтесь к приложению B.

На рис. 1.1 показано, к каким категориям относятся некоторые проекты с открытым исходным кодом.

Рис. 1.1. Различные проекты с открытым исходным кодом, связанные с контейнерами в категориях оркестраторов, движков и сред выполнения

Podman — это сокращение от *Pod Manager*. *Под*, концепция которого была популяризирована в проекте Kubernetes, представляет собой один или несколько контейнеров, использующих одни и те же пространства имен и `cgroups` (ограничения ресурсов). Поды более подробно рассматриваются в главе 4. Podman управляет как отдельными контейнерами, так и подами. Логотип Podman,

приведенный на рис. 1.2, представляет собой группу шелки (Selkies), русалок из ирландской мифологии. Группы шелки называются подами.

Проект Podman описывает Pod Manager как «контейнерный движок без демона (daemonless) для разработки, управления и запуска контейнеров OCI в вашей системе Linux. Контейнеры можно запускать как с правами root, так и в rootless-режиме» (https://podman.io). Суть Podman часто представляют простой строкой *alias Docker = Podman*, потому что из командной строки он делает почти все, что может Docker. Но как вы узнаете из этой книги, Podman может гораздо больше. Знание Docker полезно, но не критично для понимания Podman.

Рис. 1.2. Логотип Podman

ПРИМЕЧАНИЕ Open Container Initiative (OCI) — это организация по стандартизации, основной целью которой является создание открытых отраслевых стандартов для форматов контейнеров и сред их выполнения. Чтобы получить дополнительную информацию, обращайтесь к сайту https://opencontainers.org.

Проект Podman находится на github.com в показанном на рис. 1.3 разделе Containers (https://github.com/containers/podman) вместе с другими библиотеками и инструментами управления контейнерами, такими как Buildah и Skopeo (описание некоторых из этих инструментов приводится в приложении A).

Podman запускает образы с новым форматом OCI, описанным в разделе 1.2.4, а также устаревшие (legacy) образы формата Docker (v2 и v1). Podman работает с любыми образами, доступными в docker.io и quay.io, а также в сотнях других реестров контейнеров. Podman загружает эти образы на хост Linux и запускает их так же, как Docker и Kubernetes. Podman поддерживает все среды выполнения OCI, включая `runc`, `crun`, `kata` и `gvisord` (представленные в приложении B), здесь он не отличается от Docker.

Эта книга призвана помочь администраторам Linux оценить преимущества использования Podman в качестве основного контейнерного движка. Вы узнаете, как настроить свои системы максимально безопасно, при этом давая пользователям возможность работать с контейнерами. Одним из основных вариантов

применения Podman является запуск контейнерных приложений в средах с одним узлом (однонодовых), таких как пограничные устройства (edge devices). Podman и systemd позволяют управлять всем жизненным циклом приложения на нодах без вмешательства человека. Цель Podman — запускать контейнеры на компьютере с Linux естественным образом, используя все возможности платформы Linux.

Рис 1.3. Containers — это раздел разработчиков Podman и других связанных с ним контейнерных инструментов (https://github.com/containers)

ПРИМЕЧАНИЕ Podman доступен для многих дистрибутивов Linux, а также для платформ Mac и Windows. Чтобы узнать, как установить Podman на вашей платформе, обращайтесь к приложению C.

К целевой аудитории этой книги также относятся разработчики приложений. Podman — отличный инструмент для разработчиков, желающих контейнеризировать свои приложения безопасным способом. Podman позволяет разработчикам создавать контейнеры Linux на всех дистрибутивах. Кроме того, Podman доступен на платформах Mac и Windows, в этом случае он может взаимодействовать со службой Podman, работающей на виртуальной машине или на компьютере с Linux, доступными в сети. «Podman в действии» показывает, как работать с контейнерами, создавать образы контейнеров, а затем преобразовывать контейнерные приложения либо в однонодовые сервисы, либо в микросервисы на базе Kubernetes.

Podman и упомянутые выше инструменты для работы с контейнерами являются проектами с открытым исходным кодом, вклад в которые вносят представители различных компаний, университетов и организаций. К работе подключаются люди со всего мира. Проекты постоянно ищут новых контрибьюторов. Чтобы узнать, как присоединиться к этой работе, обратитесь к приложению D.

В этой главе я сначала даю краткий обзор контейнеров, а затем объясняю, какие ключевые особенности Podman делают его отличным инструментом для работы с ними.

1.2. КРАТКОЕ ОПИСАНИЕ КОНТЕЙНЕРОВ

Контейнеры — это группы работающих в системе Linux процессов, которые изолированы друг от друга. Контейнеры гарантируют, что ни одна группа процессов не будет мешать другим процессам в системе. Зловредные процессы не могут завладеть системными ресурсами, препятствуя другим процессам выполнять свои задачи. Кроме того, враждебные контейнеры не могут атаковать другие контейнеры, красть данные или вызывать атаки типа «отказ в обслуживании». Конечная цель контейнеров — предоставить возможность устанавливать приложения с их собственными версиями общих библиотек, которые не конфликтуют с приложениями, требующими других версий тех же самых библиотек. Они позволяют приложениям «жить» в виртуализованной среде так, будто они владеют всей системой.

Контейнеры изолируются следующим образом:

- *Ограничения ресурсов (cgroups).* Справочная страница cgroups (https://man7.org/linux/man-pages/man7/cgroups.7.html) определяет cgroups следующим обра-

зом: «Контрольные группы (control groups), обычно называемые cgroups, — это функция ядра Linux, позволяющая организовывать процессы в иерархические группы, использование которых для различных типов ресурсов может быть затем ограничено и отслежено»[1].

Примеры ресурсов, контролируемых cgroups:

- объем памяти, который может использовать группа процессов;
- количество CPU, которое могут использовать процессы;
- объем сетевых ресурсов, которые может использовать процесс.

Основная идея cgroups заключается в том, чтобы предотвратить доминирование одной группы процессов над определенными системными ресурсами таким образом, чтобы и другая группа процессов могла работать в системе.

- *Ограничения безопасности.* Контейнеры изолируются друг от друга с помощью различных доступных в ядре инструментов безопасности. Цель состоит в том, чтобы заблокировать повышение привилегий и предотвратить совершение зловредной группой процессов враждебных действий против системы. Например:
  - отключение привилегий Linux ограничивает возможности root;
  - SELinux контролирует доступ к файловой системе;
  - доступ к файловым системам имеется только на чтение;
  - Seccomp ограничивает системные вызовы, доступные в ядре;
  - пользовательское пространство имен для сопоставления одной группы UID на хосте с другой позволяет получить доступ к ограниченному root-окружению.

В табл. 1.1 приведена дополнительная информация и даны ссылки с более подробным описанием некоторых из этих функций безопасности.

Таблица 1.1. Расширенные функции безопасности Linux

Компонент	Описание	Справка
Linux привилегии (Linux capabilities)	Linux capabilities разделяют возможности root на отдельные привилегии	Страница документации «capabilities» предоставляет обзор доступных привилегий (https://bit.ly/3A3Ppeg)

[1] Текст справки на английском, здесь приведен перевод. — *Примеч. ред.*

Компонент	Описание	Справка
SELinux	Security-Enhanced Linux (SELinux) — это механизм ядра Linux, который маркирует каждый процесс и каждый объект файловой системы. Политика SELinux определяет правила взаимодействия промаркированных процессов с промаркированными объектами. Ядро Linux обеспечивает соблюдение этих правил	Я сделал книгу-раскраску, чтобы таким забавным способом помочь вам разобраться в SELinux (https://bit.ly/33plEbD). Если вы действительно хотите изучить предмет, посмотрите страницу SELinux Notebook (https://bit.ly/ 3GxGhkm)
Seccomp	seccomp — это механизм ядра Linux, ограничивающий количество системных вызовов (syscalls) для группы процессов в системе. Вы можете исключить вызов потенциально опасных системных вызовов процессами	Страница руководства seccomp — хороший источник дополнительной информации (https://bit.ly/3rnnim1)
Пользовательское пространство имен (User namespace)	Пользовательское пространство имен позволяет вам иметь свои Linux-привилегии в пределах группы UID и GID, назначенных этому пространству имен, не имея при этом возможностей root на хосте	Пользовательское пространство имен рассматривается в главе 3

- *Технологии виртуализации (пространства имен)*. В ядре Linux используется концепция, называемая пространством имен. Это означает, что создаются виртуальные среды, в которых один набор процессов видит один набор ресурсов, а другой набор процессов видит отличный от первого набор ресурсов. Эти виртуальные среды не дают процессам увидеть остальные части системы, создавая тем самым эффект виртуальной машины (VM) без дополнительных затрат ресурсов. Примерами пространств имен являются:
  - *пространство имен Network*, ограничивающее доступ к сети хоста, но дающее доступ к виртуальным сетевым устройствам;
  - *пространство имен Mount*, исключающее возможность просмотра всей файловой системы и оставляющее доступ только к файловой системе контейнера;
  - *пространство имен PID*, исключающее просмотр других процессов в системе; процессы контейнера видят только процессы внутри контейнера.

Эти технологии контейнеризации существуют в ядре Linux уже много лет. Инструменты безопасности для изоляции процессов появились в Unix еще в 1970-х годах, а SELinux — в 2001 году. Пространства имен были введены примерно в 2004 году, а cgroups — в 2006-м.

ПРИМЕЧАНИЕ Существуют также образы контейнеров для Windows, но в этой книге основное внимание уделяется контейнерам на базе Linux. Даже если вы запускаете Podman в Windows, вы все равно работаете с контейнерами Linux. Podman на платформе Mac рассматривается в приложении E, а Podman на Windows — в приложении F.

1.2.1. Образы контейнеров: новый способ доставки программного обеспечения

Контейнеры не были столь популярны, пока проект Docker не представил концепцию образа контейнера и реестра контейнеров. По сути, был создан новый способ доставки программного обеспечения (ПО).

Установка нескольких приложений в системе Linux традиционно приводила к проблеме управления зависимостями. До появления контейнеров мы упаковывали ПО с помощью менеджеров пакетов, таких как RPM или DPKG. Эти пакеты устанавливаются на хост и совместно используют его содержимое, включая общие библиотеки. Когда разработчики тестируют свой код, при запуске на данном хосте все может работать нормально. Затем команда QA-инженеров будет тестировать это ПО на другой машине с другими пакетами и может столкнуться с ошибками. Чтобы выработать надлежащие требования к окружению, обе команды должны будут работать вместе. Наконец, программное обеспечение поставляется заказчикам, у которых имеется множество различных конфигураций и установленного ПО, что приводит к дальнейшим сбоям.

Образы контейнеров решают проблему управления зависимостями, объединяя в единое целое всё ПО, необходимое для запуска приложения. Вы поставляете все библиотеки, исполняемые файлы и файлы конфигурации вместе. Программное обеспечение изолировано от хоста с помощью технологии контейнеризации. Ядро обычно оказывается единственной частью системы хоста, с которой взаимодействует ваше приложение.

Разработчик, QA-инженеры и заказчик запускают одно и то же контейнерное окружение вместе с приложением. Это гарантирует согласованность с окружением и ограничивает количество ошибок, вызванных неправильной конфигурацией.

Контейнеры часто сравнивают с виртуальными машинами, поскольку и те и другие могут запускать несколько изолированных приложений на одном узле. При использовании виртуальных машин вам необходимо управлять всей операционной системой виртуальной машины, а также самим изолированным приложением. Приходится управлять жизненным циклом различных ядер, системой инициализации, ведением журнала (логированием), обновлениями безопасности, резервным копированием и т. д. Кроме того, существуют накладные расходы всей работающей ОС, а не только приложения. В мире контейнеров все,

что вы запускаете, — это приложение в контейнере, тут нет никаких накладных расходов и необходимости управления операционной системой. На рис. 1.4 показаны три приложения, запущенные на трех разных виртуальных машинах.

Рис. 1.4. Физическая машина с тремя приложениями на трех виртуальных машинах

При использовании виртуальных машин вам в итоге придется управлять четырьмя ОС, тогда как при использовании контейнеров три приложения работают только с необходимыми пользовательскими пространствами.

1.2.2. Как образы контейнеров способствуют развитию микросервисов

Упаковка приложений в образы позволяет устанавливать несколько приложений с конфликтующими требованиями на одном хосте. Например, одно приложение может требовать иную версию библиотеки C, чем другое приложение, что не позволяет установить их одновременно. На рис. 1.6 показано традиционное приложение, работающее в операционной системе без использования контейнеров.

Рис. 1.5. Физическая машина, на которой работают три приложения в трех контейнерах

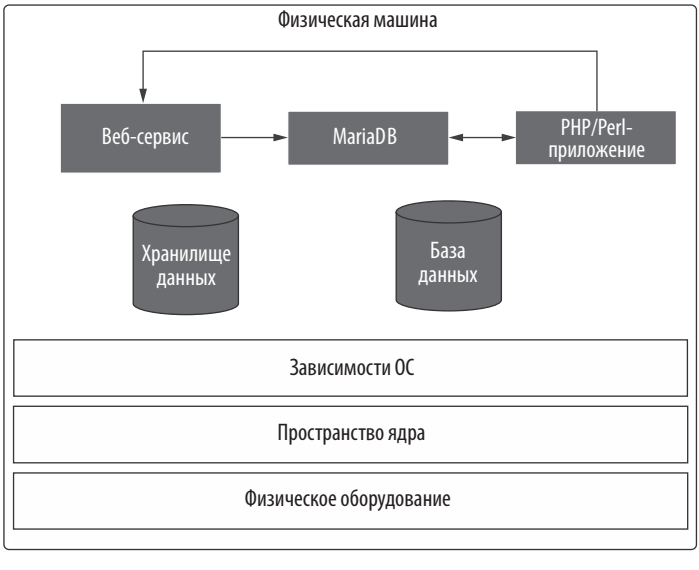


Рис. 1.6. Классический LAMP стек (Linux, Apache, MariaDB и приложение PHP/PERL), работающий на сервере

Контейнеры имеют правильную библиотеку C в своем образе, причем в каждом образе потенциально могут быть разные версии библиотеки, специфичные для приложения внутри этого контейнера. Вы можете запускать приложения из совершенно разных дистрибутивов.

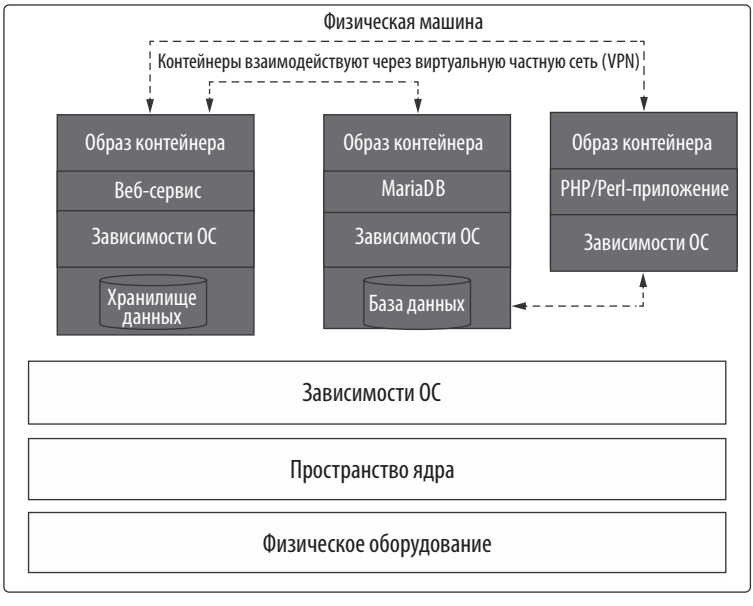


Рис. 1.7. Стек LAMP, упакованный в контейнеры микросервисов. Поскольку контейнеры взаимодействуют по сети, их можно легко перемещать на другие виртуальные машины, что значительно упрощает повторное использование

Контейнеры позволяют легко запускать несколько экземпляров одного и того же приложения, что продемонстрировано на рис. 1.7. Рекомендуется упаковывать один сервис или одно приложение в отдельный образ. Контейнеры также позволяют легко связывать несколько приложений по сети.

Вместо того чтобы разрабатывать монолитные приложения, в которых есть веб-фронтенд, балансировщик нагрузки и база данных, вы можете создать три разных образа, а затем соединить их вместе, получая тем самым микросервисы. Микросервисы позволяют вам и другим пользователям экспериментировать с запуском нескольких баз данных и веб-интерфейсов, а затем организовывать их совместную работу. Контейнерные микросервисы делают возможным совместное и повторное использование программного обеспечения.

1.2.3. Формат образа контейнера

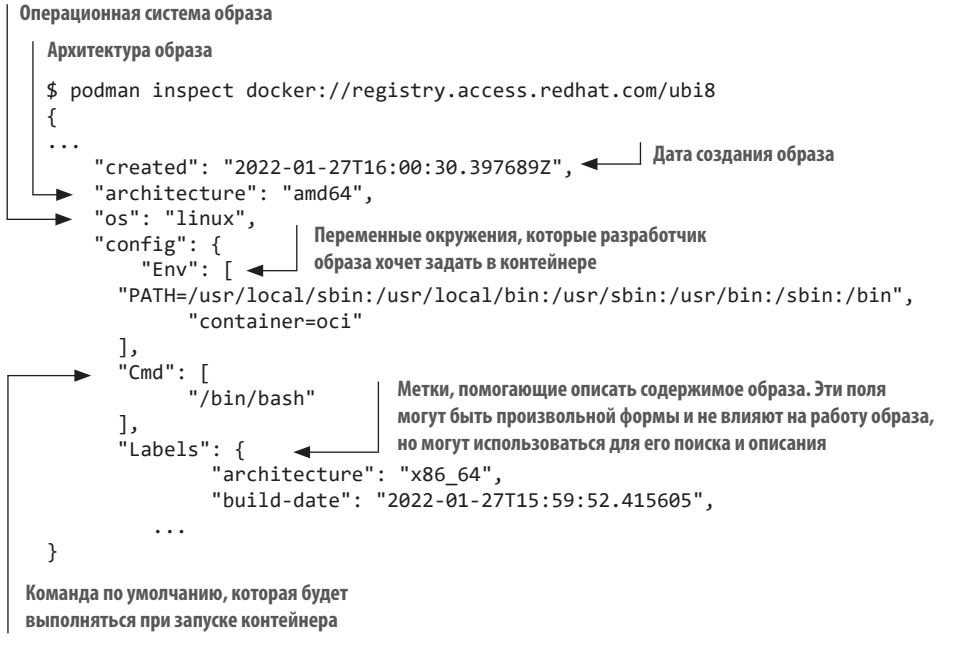
Образ контейнера состоит из трех компонентов:

- дерево каталогов, содержащее все программное обеспечение, необходимое для запуска вашего приложения;

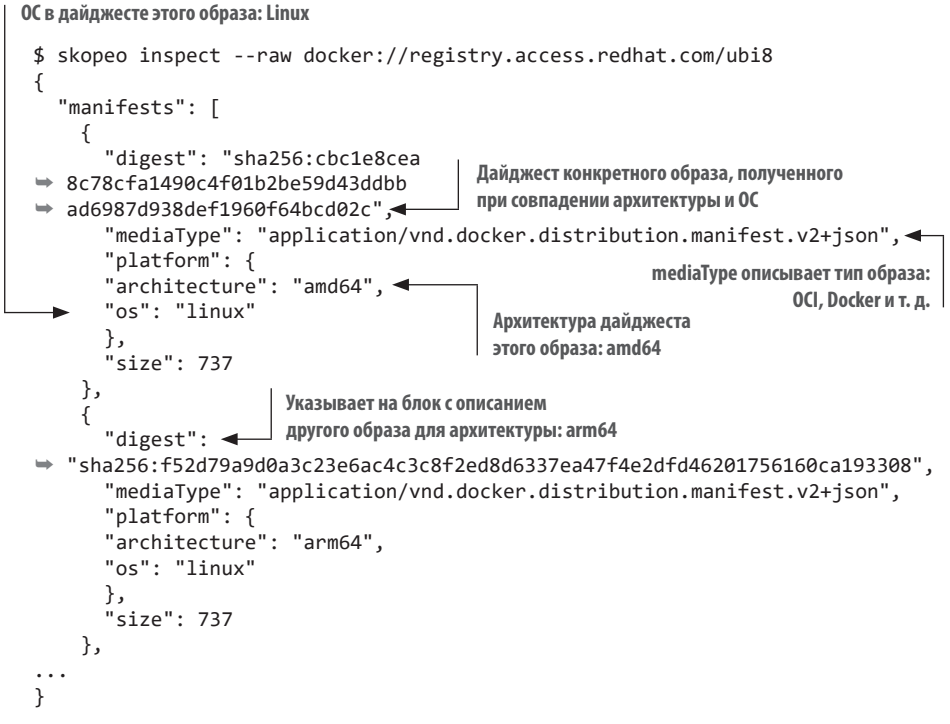
- JSON-файл, описывающий содержимое rootfs;
- еще один файл JSON, называемый manifest list, который связывает несколько образов вместе для поддержки различных архитектур.

Дерево каталогов называется *rootfs* (корневая файловая система). Оно располагается так, как если бы это был корень (`/`) системы Linux.

Исполняемый файл, который будет запущен в rootfs, рабочий каталог, используемые переменные окружения, мейнтейнер исполняемого файла и другие метки, помогающие узнать о содержимом образа, определены в первом JSON-файле. Для просмотра этого JSON-файла воспользуйтесь командой `podman inspect`:



Второй JSON-файл, manifest list, позволяет пользователям на машине arm64 получить образ с таким же именем, как если бы они были на машине amd64. Podman загружает образ на основе архитектуры машины по умолчанию, используя этот список манифестов. Skopeo — инструмент, применяющий те же базовые библиотеки, что и Podman, он доступен по адресу github.com/containers/skopeo (подробнее рассматривается в приложении A). Skopeo предоставляет более низкоуровневый вывод для изучения структуры образа контейнера. В следующем примере используется команда `skopeo` с параметром `–raw`, чтобы получить спецификацию манифеста образа `registration.access.redhat.com/ubi8`:




```
$ skopeo inspect --raw docker://registry.access.redhat.com/ubi8
{
  "manifests": [
    {
      "digest": "sha256:cbc1e8cea
➥ 8c78cfa1490c4f01b2be59d43ddbb
➥ ad6987d938def1960f64bcd02c",
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "platform": {
      "architecture": "amd64",
      "os": "linux"
      },
      "size": 737
    },
    {
      "digest":
➥ "sha256:f52d79a9d0a3c23e6ac4c3c8f2ed8d6337ea47f4e2dfd46201756160ca193308",
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "platform": {
      "architecture": "arm64",
      "os": "linux"
      },
      "size": 737
    },
...
}
```

Образы используют утилиту Linux tar для упаковки rootfs и файлов JSON вместе. Затем эти образы помещаются на хранение на веб-серверы, называемые реестрами контейнеров (например, docker.io, quay.io и Artifactory). Контейнерные движки типа Podman могут копировать эти образы на хост и распаковывать их в файловую систему. Затем движок объединяет JSON-файл образа, встроенные параметры движка по умолчанию и данные, введенные пользователем, создавая новый JSON-файл спецификации контейнера для среды выполнения OCI. В этом JSON-файле описывается, как запустить контейнерное приложение.

На последнем этапе движок запускает небольшую программу, называемую средой выполнения контейнера (например, `runc`, `crun`, `kata` или `givisord`). Среда выполнения считывает JSON контейнера и настраивает cgroups ядра, ограничения безопасности и пространства имен, прежде чем окончательно запустить основной процесс контейнера.

1.2.4. Контейнерные стандарты

Организация по стандартизации OCI определяет стандартные форматы для хранения и описания образов контейнеров. Она также определила стандарт для

движков, запускающих контейнеры. OCI создала OCI Image Format, который стандартизирует формат образов и их JSON-файлов. Кроме того, была разработана спецификация OCI Runtime Specification, которая стандартизировала JSON-файл контейнера для среды выполнения OCI. Стандарты OCI позволяют различным контейнерным движкам, таким как Podman[1], работать со всеми образами, хранящимися в реестрах контейнеров, и запускать их точно так же, как и все другие контейнерные движки, включая Docker (см. рис. 1.7).

1.3. ЗАЧЕМ ИСПОЛЬЗОВАТЬ PODMAN, ЕСЛИ ЕСТЬ DOCKER

Мне часто задают вопрос: «Зачем вам нужен Podman, если уже есть Docker?». Одна из причин в том, что *открытый исходный код всегда предполагает выбор*. В операционных системах больше одного редактора, больше одной оболочки, больше одной файловой системы и больше одного веб-браузера. Я считаю, что Podman устроен принципиально лучше, чем Docker, и предлагает функции, повышающие безопасность и содействующие более широкому применению контейнеров.

1.3.1. Почему есть только один способ запуска контейнеров

Одним из преимуществ Podman является то, что он был создан намного позже Docker. Разработчики Podman рассматривали возможности по улучшению устройства Docker с совершенно другой точки зрения. Поскольку Docker имеет открытый исходный код, Podman использует часть его кода и одновременно преимущества таких новых стандартов, как Open Container Initiative. Сосредоточиваясь на разработке новых функций, Podman взаимодействует с open-source сообществом.

Далее в этом разделе я расскажу о некоторых из упомянутых улучшений. Таблица 1.2 сравнивает функции, доступные в Podman и Docker.

Таблица 1.2. Сравнение возможностей Podman и Docker

Функции	Podman	Docker	Описание
Поддержка всех OCI и Docker-образов	✓	✓	Загружает и запускает образы контейнеров из реестров контейнеров, например quay.io и docker.io (глава 2)

[1] К контейнерным движкам относятся Buildah, CRI-O, containerd и многие другие.

Функции	Podman	Docker	Описание
Запуск контейнерных движков OCI	✓	✓	Запускает `runc`, `crun`, Kata, gVisor и т. д. (приложение B)
Простой интерфейс командной строки	✓	✓	Podman и Docker используют один и тот же CLI (глава 2)
Интеграция с systemd	✓	✗	Podman поддерживает запуск systemd внутри контейнера, а также многие функции systemd (глава 7)
Модель fork/exec	✓	✗	Контейнер является дочерним объектом команды
Полная поддержка пользовательского пространства имен	✓	✗	Только Podman поддерживает запуск контейнеров в отдельных пользовательских пространствах имен (глава 6)
Клиент-серверная модель	✓	✓	Docker — это демон REST API. Podman поддерживает REST API через активируемый сокетами сервис systemd (глава 9)
Поддержка `docker-compose`	✓	✓	Compose-скрипты работают с обоими REST API. Podman работает в режиме без root (глава 9)
Поддержка docker-py	✓	✓	Связка docker-py Python работает с обоими REST API. Podman работает в режиме без root. Podman также поддерживает podman-py для запуска расширенных функций (глава 9)
Daemonless	✓	✗	Команда Podman работает как классический инструмент командной строки, в то время как Docker требует запуска нескольких демонов, работающих от root
Поддержка Kubernetes-подобных подов	✓	✗	Podman поддерживает запуск нескольких контейнеров в рамках одного пода (глава 4)
Поддержка Kubernetes YAML	✓	✗	Podman может запускать контейнеры и поды на основе Kubernetes YAML. Он генерирует Kubernetes YAML из запущенных контейнеров (глава 8)
Поддержка Docker Swarm	✗	✓	Podman считает, что будущее управляемых многоузловых контейнеров за Kubernetes, и не планирует внедрять Swarm

Таблица 1.2 (*окончание*)

Функции	Podman	Docker	Описание
Настраиваемые реестры	✓	✗	Podman позволяет настраивать реестры для расширения использования коротких имен. Docker жестко привязан к docker.io при указании короткого имени (глава 5)
Настраиваемые значения по умолчанию	✓	✗	Podman поддерживает полную настройку всех параметров по умолчанию, включая безопасность, пространства имен и тома (глава 5)
Поддержка macOS	✓	✓	Podman и Docker поддерживают запуск контейнеров на Mac через виртуальную машину под управлением Linux (приложение E)
Поддержка Windows	✓	✓	Podman и Docker поддерживают запуск контейнеров на Windows WSL 2 или на виртуальной машине под управлением Linux (приложение F)
Поддержка Linux	✓	✓	Podman и Docker поддерживаются во всех основных дистрибутивах Linux (приложение C)
Контейнеры не останавливаются при обновлении	✓	✗	Podman не должен оставаться запущенным, когда запущены контейнеры. Поскольку демон Docker следит за контейнерами, когда он останавливается, по умолчанию останавливаются все контейнеры

1.3.2. Непривилегированные (rootless) контейнеры

Вероятно, самая важная особенность Podman — то, что он способен работать в режиме rootless. Есть множество ситуацией, в которых вам не хочется предоставлять своим пользователям полный root-доступ, но и пользователям и разработчикам нужно запускать контейнеры и создавать образы. Требование root-доступа препятствует широкому внедрению Docker компаниями, которые заботятся о своей безопасности. Podman, напротив, может запускать контейнеры без использования каких-либо дополнительных функций безопасности в Linux, не считая стандартной учетной записи для входа.

Вы можете запустить клиент Docker как обычный пользователь, добавленный в группу пользователей Docker (/etc/group), но я считаю предоставление такого

доступа одной из самых опасных ошибок, которые можно совершить на компьютере с Linux. Доступ к docker.sock позволяет вам получить полный root-доступ на хосте, выполнив приведенную ниже команду. В этой команде вы монтируете всю операционную систему хоста / в каталог /host внутри контейнера. Флаг `--privileged` отключает безопасность контейнера, а затем вы выполняете `chroot` в каталоге /host. После `chroot` вы оказываетесь в оболочке root в / операционной системы с полными привилегиями root:

```
$ docker run -ti --name hacker --privileged -v /:/host ubi8 chroot /host
#
```

На этом этапе у вас есть полные привилегии root на машине, и вы можете делать все, что захотите. Когда вы закончите со взломом машины, нужно просто выполнить команду `docker rm`, чтобы удалить контейнер и все записи о том, что вы сделали:

```
$ docker rm hacker
```

Если Docker настроен по умолчанию на логирование в файл, все записи о запуске контейнера стираются. Я считаю, что это гораздо хуже, чем установка `sudo` без root, поскольку при использовании `sudo` у вас по крайней мере есть шанс увидеть в файлах журнала, что `sudo` был запущен.

При использовании Podman запущенные в системе процессы всегда принадлежат пользователю и не обладают возможностями, превосходящими таковые обычного пользователя. Даже если вы вырветесь за пределы контейнера, процесс все равно будет продолжаться как запущенный под вашим UID, а все действия в системе запишутся в журналы аудита. Пользователи Podman не могут просто удалить контейнер и скрыть следы своего пребывания. Более подробную информацию вы найдете в главе 6.

> **ПРИМЕЧАНИЕ** Сейчас Docker имеет возможность запускаться без root, подобно Podman, но почти никто не использует его таким образом. Запуск нескольких служб в домашнем каталоге только для того, чтобы запустить один контейнер, не прижился.

1.3.3. Модель fork/exec

Docker построен как сервер REST API. В основе Docker лежит клиент-серверная архитектура, включающая несколько демонов. Когда пользователь запускает клиент Docker, вызывается инструмент командной строки, который подключается к демону Docker. Демон Docker загружает образы в свое хранилище, а затем подключается к демону containerd, который, наконец, запускает среду

выполнения OCI, создающую контейнер. Получается, что демон Docker — это коммуникационная платформа, которая передает данные чтения и записи `stdin`, `stdout` и `stderr` от начального процесса (PID1), созданного в контейнере. Демон ретранслирует все данные вывода обратно клиенту Docker. Пользователям процессы контейнера представляются просто дочерними элементами текущей сессии, но под капотом происходит множество взаимодействий. На рис. 1.8 показана клиент-серверная архитектура Docker.

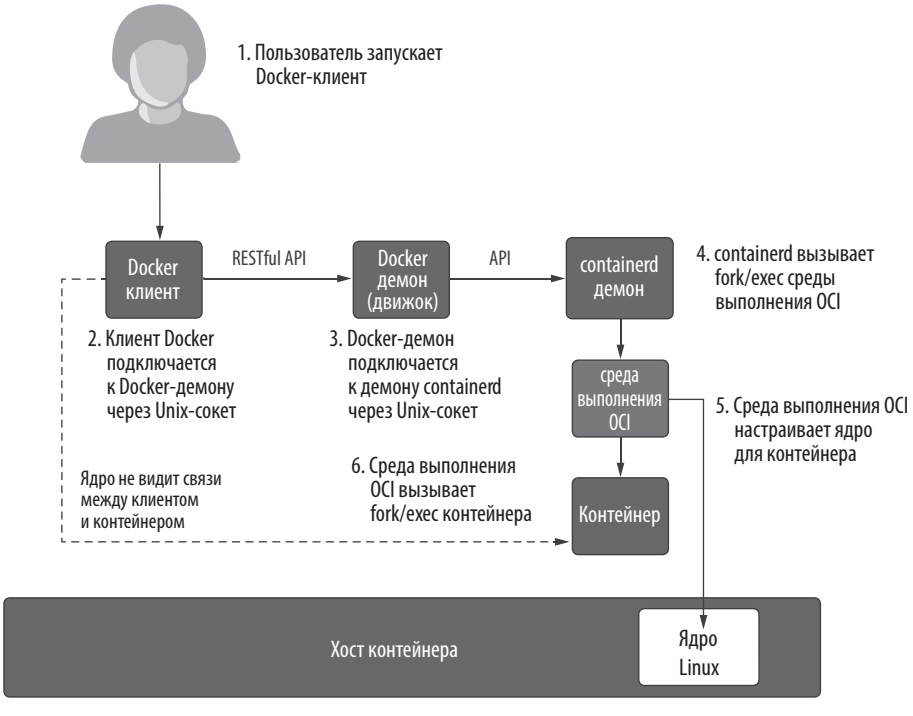


Рис. 1.8. Клиент-серверная архитектура Docker. Контейнер является прямым потомком containerd, а не Docker-клиента. Ядро не видит связи между клиентской программой и контейнером

Суть в том, что клиент Docker взаимодействует с демоном Docker, который, в свою очередь, взаимодействует с демоном containerd, а последний в итоге и запускает среду выполнения OCI, например `runc`, для запуска PID1-контейнера. Запуск контейнеров таким образом сопряжен со многими сложностями. Годами сбои в любом из демонов приводили к остановке всех контейнеров, и часто было трудно диагностировать, что же произошло.

Основная команда инженеров Podman состоит из специалистов в области операционных систем, основанных на философии Unix. Unix и язык C были разработаны в модели fork/exec. В сущности, при выполнении новой программы родительская программа (например, оболочка bash) создает новый процесс, а затем выполняет новую программу как дочернюю программу старой. Команда инженеров Podman решила упростить работу с контейнерами, создав инструмент для загрузки образов из реестра контейнеров, настройки хранилища контейнеров и затем запуска среды выполнения OCI, которая запускает контейнер как дочернюю программу вашего контейнерного движка.

В операционной системе Unix процессы могут обмениваться содержимым через файловую систему и механизмы межпроцессного взаимодействия (IPC). Эти возможности операционной системы позволяют нескольким контейнерным движкам совместно использовать хранилище без необходимости запуска демона для контроля доступа и обмена содержимым. Контейнерам не нужно взаимодействовать друг с другом, если не считать использование механизмов блокировки, предоставляемых файловой системой ОС. В следующих главах будут рассмотрены преимущества и недостатки этого механизма. На рис. 1.9 показана архитектура Podman и коммуникационный поток в ней.

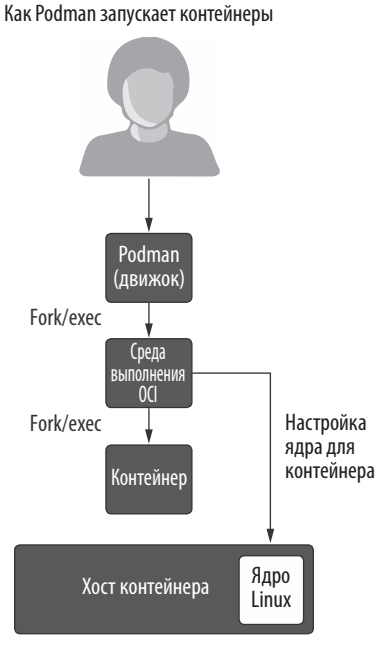


Рис. 1.9. Архитектура fork/exec в Podman. Пользователь запускает Podman, запускающий среду выполнения OCI, которая затем запускает контейнер. Контейнер является прямым потомком Podman

1.3.4. Podman без демонов

Podman принципиально отличается от Docker тем, что он не нуждается в демонах. Podman может запускать все те же образы контейнеров, что и Docker, и запускать контейнеры с той же средой выполнения. Однако Podman делает это без постоянно запущенных с правами root демонов.

Представьте, что у вас есть веб-сервис, запускаемый во время загрузки системы. Веб-сервис упакован в контейнер, поэтому вам нужен контейнерный движок. В случае с Docker вам нужно настроить его так, чтобы он действовал на вашем компьютере, и при этом каждый из демонов работал и принимал соединения. Вы запустите клиент Docker, чтобы запустить веб-сервис. Теперь у вас запущено контейнерное приложение, а также все демоны Docker. В случае с Podman вы просто выполняете команду для запуска вашего контейнера, и Podman сам завершится. Ваш контейнер будет продолжать работать, не затрачивая дополнительных ресурсов на запуск нескольких демонов. Меньшие затраты дополнительных ресурсов очень популярны на низкопроизводительных машинах, таких как устройства интернета вещей (internet-of-things, IoT) и пограничные серверы.

1.3.5. Дружественная командная строка

Одна из замечательных особенностей Docker — простой интерфейс командной строки. Есть также `RKT`, `lxc` и `lxcd`, но у них есть свои собственные интерфейсы командной строки. Команда разработчиков Podman быстро поняла, что не сможет завоевать рынок, если у Podman будет собственный интерфейс командной строки. Docker был доминирующим инструментом, и почти все, кто когда-либо работал с контейнерами, делали это с помощью командной строки (CLI) Docker. Кроме того, что бы вы ни искали в интернете о работе с контейнером, вы неизменно получите пример с использованием командной строки Docker. С самого начала Podman должен был соответствовать командной строке Docker. Был быстро разработан слоган для замены Docker на Podman: `alias Docker = Podman`.

С помощью этой команды вы можете продолжать вводить команды Docker, но запускать ваши контейнеры будет Podman. Если командная строка Podman отличается от таковой в Docker, это считается багом Podman, и пользователи требуют его исправления, чтобы инструменты совпадали. Есть несколько команд (например, Docker Swarm), которые Podman не поддерживает, но в остальном Podman является полноценной заменой Docker CLI.

Многие дистрибутивы поставляют пакет под названием `podman-docker`, который меняет псевдоним с *docker* на *podman* и содержит ссылку на страницу man. Псевдоним означает, что когда вы вводите `docker ps`, запускается команда `podman ps`. Если вы выполните `man docker ps`, отобразятся справочные страницы podman ps.

На рис. 1.10 показано сообщение в Twitter пользователя Podman, который работал с таким псевдонимом и с удивлением обнаружил, что использовал Podman в течение двух месяцев, думая, что применяет Docker.

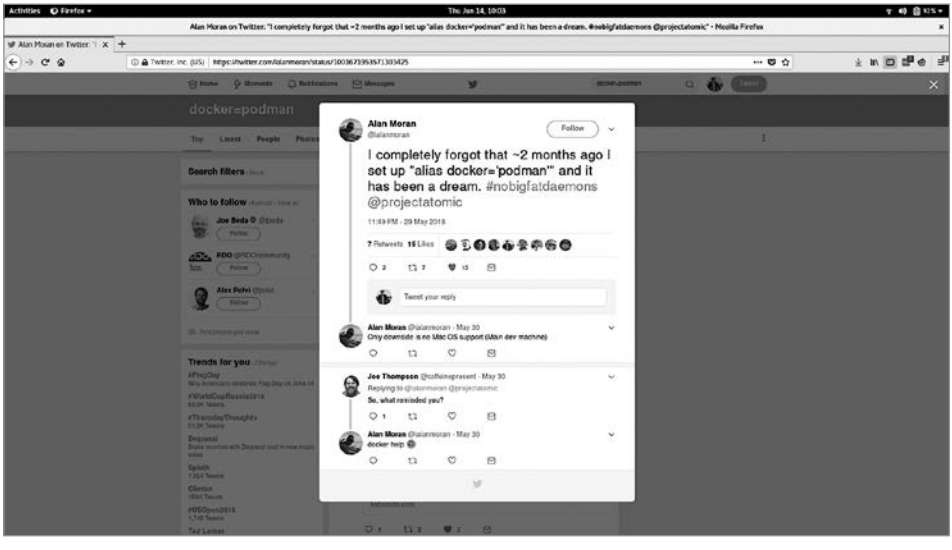


Рис. 1.10. Твит про «alias docker='podman'»

Еще в 2018 году Алан Моран (Alan Moran) написал в Твиттере: «Я совсем забыл, что примерно два месяца назад я установил 'alias docker=«podman»', и это была мечта». «И что же тебе напомнило об этом?» — спросил Джо Томсон (Joe Thomson), на что Алан ответил: «docker help». И тут появилась страница помощи Podman.

1.3.6. Поддержка REST API

Podman можно запустить как REST API-сервис, активируемый через сокет. Это позволяет удаленным клиентам управлять контейнерами Podman и запускать их. Podman поддерживает API Docker, а также API Podman для расширенных возможностей. Благодаря использованию Docker API, Podman поддерживает `docker-compose` и других пользователей Python-обвязки docker-ру. Это означает, что даже если вы построили свою инфраструктуру на использовании сокета Docker для запуска контейнеров, вы можете просто заменить Docker на службу Podman и продолжать работу с существующими сценариями и инструментами. Глава 9 посвящена службе Podman.

Podman REST API также позволяет удаленным клиентам Podman на системах Mac, Windows и Linux взаимодействовать с контейнерами Podman на машине Linux. В приложениях E и F рассказывается об использовании Podman на машинах с Mac и Windows.

1.3.7. Интеграция с systemd

Systemd — основная система инициализации в операционных системах. Процесс init в системе Linux — это первый процесс, запускаемый ядром при загрузке. Таким образом, система инициализации является прародителем всех процессов и может отслеживать их. Podman планирует полностью интегрировать работу контейнеров с системой инициализации. Пользователям нужна возможность использовать systemd для запуска и остановки контейнеров во время загрузки. Контейнеры должны делать следующее:

- поддерживать systemd внутри контейнера;
- поддерживать активацию через сокет;
- поддерживать уведомления systemd о том, что контейнерное приложение полностью активировано;
- позволять systemd полностью управлять cgroups и временем жизни контейнерного приложения.

По сути, контейнеры работают как службы в юнит-файлах systemd. Многие разработчики хотят запускать systemd в контейнере, чтобы внутри него запускать несколько определенных системой служб.

Однако сообщество разработчиков Docker не согласно с таким подходом и отклоняет все запросы на внесение изменений, интегрирующих systemd в Docker. Они считают, что Docker должен сам управлять жизненным циклом контейнера, и не желают подстраиваться под пользователей, которые хотят запускать systemd в контейнере.

Сообщество разработчиков Docker полагает, что Docker-демон, а не systemd должен быть контроллером процессов, управлять жизненным циклом контейнеров, запускать и останавливать их во время загрузки. Проблема в том, что в systemd гораздо больше функций, чем в Docker, среди них упорядочивание запуска, активация через сокет, уведомления о готовности сервисов и т. д. На рис. 1.11 показан реальный бейдж сотрудника Docker на первой конференции DockerCon, иллюстрирующий неприязненное отношение к systemd.

Когда создавался Podman, разработчики попытались сделать его полностью интегрированным с systemd. При запуске systemd внутри контейнера Podman настраивает контейнер так, как ожидает systemd, и позволяет ему просто

запускаться как PID1 контейнера с ограниченными привилегиями. Podman позволяет запускать службы внутри контейнера так же, как они запускаются в системе или на виртуальной машине, — через файлы systemd-юнитов. Podman поддерживает активацию через сокет, уведомления сервисов и многие другие возможности файлов systemd-юнитов. Podman упрощает генерацию таких файлов с использованием лучших практических приемов для запуска контейнеров в рамках службы systemd. Для получения дополнительной информации обращайтесь к главе 7 «Интеграция с systemd».

Рис. 1.11. Бейдж сотрудника Docker на DockerCon EU. Надпись на бейдже: «Я говорю "нет" характерным для systemd пулл-реквестам»

Проект Containers (https://github.com/containers), в котором обитают Podman, библиотеки контейнеров и другие инструменты управления контейнерами, стремится охватить все возможности операционной системы и полностью интегрировать их.

1.3.8. Поды

Одно из преимуществ Podman описано в его названии. Как упоминалось ранее, *Podman* — это сокращение от *Pod Manager*. Официальная документация Kubernetes сообщает: «Pod (как стая тюленей pod of seals — отсюда логотип, или как стручок гороха pea pod) — это группа из одного или нескольких контейнеров с общими ресурсами хранилища/сети и спецификацией того, как запускать эти контейнеры». Podman либо работает с одним контейнером за раз, как Docker, либо управляет группами контейнеров, объединенных в под. Одной из целей

разработки контейнеров является разделение сервисов на отдельные контейнеры — микросервисы. Затем контейнеры объединяются вместе для создания более крупных сервисов. Поды позволяют группировать несколько сервисов вместе, чтобы сформировать еще более крупный сервис, управляемый как единое целое. Одна из целей Podman — предоставить возможность экспериментировать с подами. На рис. 1.12 показаны два пода, запущенные в системе, каждый из них содержит три контейнера.

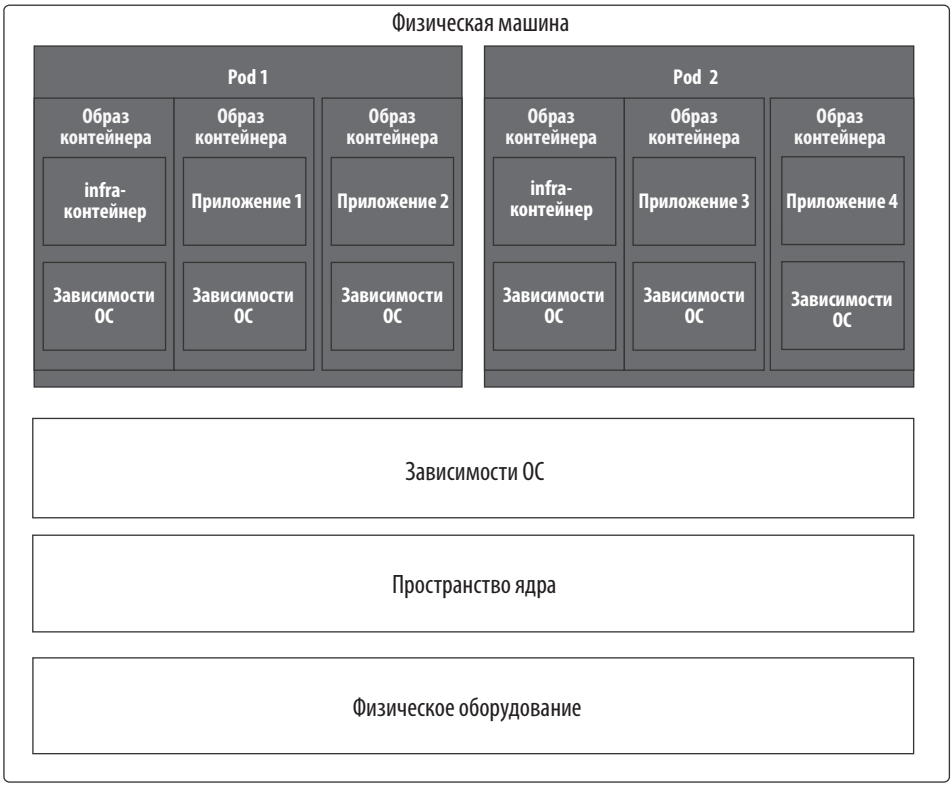


Рис. 1.12. Два пода, запущенные на одном хосте. Каждый под запускает два разных контейнера вместе с infra-контейнером

В Podman есть команда `podman generate kube`, которая позволяет генерировать YAML-файлы Kubernetes из запущенных контейнеров и подов, что будет показано в главе 7.

В Podman есть также команда `podman play kube`, позволяющая запускать YAML-файлы Kubernetes и создавать поды и контейнеры на вашем хосте.

Я предлагаю использовать Podman для запуска подов и контейнеров на одном хосте, а Kubernetes — для запуска подов и контейнеров на нескольких машинах и во всей вашей инфраструктуре. Другие проекты, такие как kind (https://kind.sigs.k8s.io/docs/user/rootless), экспериментируют с запуском подов с помощью Podman под управлением Kubernetes.

1.3.9. Настраиваемые реестры

Такие контейнерные движки, как Podman, поддерживают концепцию загрузки образов с использованием коротких имен, например ubi8, без указания реестра, в котором они находятся: registry.access.redhat.com. Полные имена образов включают имя реестра контейнеров, из которого они были извлечены: registry.access.redhat.com/library/ubi8:latest. В табл. 1.3 показано имя этого образа в разбивке по компонентам.

Таблица 1.3. Таблица соответствия короткого имени полному имени образа контейнера

Имя	Реестр	Репозиторий	Имя	Тег
Короткое имя			ubi8	
Полное имя	registry.access.redhat.com	library	ubi8	`latest`

В Docker жестко запрограммировано, что при использовании короткого имени образ всегда берется из источника по адресу https://docker.io. Если вы хотите взять образ из другого реестра контейнеров, вы должны полностью указать его имя. В следующем примере я пытаюсь извлечь ubi8/httpd-24, и это не удается, потому что образ контейнера не находится на docker.io. Образ находится на сайте registry.access.redhat.com:

```
# docker pull ubi8/httpd-24
Using default tag: latest
Error response from daemon: pull access denied for ubi8/httpd-24,
repository does not exist or may require 'docker login': denied: requested
access to the resource is denied
```

Поэтому если я хочу использовать ubi8/httpd-24, я вынужден вводить полное имя, включая имя реестра:

```
# docker pull registry.access.redhat.com/ubi8/httpd-24
```

Движок Docker дает docker.io преимущество, предпочитая его другим контейнерным реестрам. Podman был разработан так, чтобы вы могли указать несколько реестров, подобно тому как это можно сделать с помощью инструментов `dnf`,

yum и apt для установки пакетов. Вы даже можете удалить docker.io из списка. Если вы попытаетесь извлечь ubi8/httpd-24 с помощью Podman, он представит вам список реестров на выбор:

```
$ podman pull ubi8/httpd-24
? Please select an image:
    registry.fedoraproject.org/ubi8/httpd-24:latest
              ▸ registry.access.redhat.com/ubi8/httpd-24:latest
    docker.io/ubi8/httpd-24:latest
    quay.io/ubi8/httpd-24:latest
```

Как только вы примете решение, Podman запишет псевдоним короткого имени и больше не будет запрашивать ранее выбранный реестр. Podman поддерживает множество других функций, таких как блокировка реестров, загрузка только подписанных образов, настройка зеркал образов и указание жестко закодированных коротких имен, дающее возможность напрямую сопоставлять конкретные короткие имена с определенными длинными именами (глава 5).

1.3.10. Множественный транспорт

Podman поддерживает множество различных источников и целей для образов контейнеров, называемых *транспортами* (табл. 1.4). Podman может не только загружать образы из реестров и из локального хранилища контейнеров, но и поддерживать образы, хранящиеся в формате OCI, формате OCI TAR, устаревшем формате Docker TAR, формате каталогов, а также образы непосредственно из Docker-демона. Команды Podman без проблем запускают образы из каждого из этих форматов.

Таблица 1.4. Транспорты, поддерживаемые Podman

Транспорт	Описание
Реестр контейнеров (docker)	Ссылается на образ контейнера, хранящийся на удаленном веб-сайте реестра образов контейнеров. Реестры хранят образы контейнеров и делятся ими (например, docker.io и quay.io)
oci	Ссылается на образ контейнера, соответствующий спецификациям макета OCI. Архивные файлы манифеста и слоев находятся в локальном каталоге в виде отдельных файлов
dir	Ссылается на образ контейнера, совместимый с макетом образа Docker, аналогичный транспорту oci, но сохраняющий файлы в устаревшем формате docker
docker-archive	Ссылается на образ контейнера, соответствующий макету образа Docker, который упакован в архив TAR

Транспорт	Описание
`oci-archive`	Ссылается на образ контейнера, соответствующий спецификациям макета OCI, который упакован в архив TAR
`docker-daemon`	Ссылается на образ, хранящийся во внутреннем хранилище docker-демона
`container-storage`	Ссылается на образ контейнера, расположенный в локальном хранилище. Podman по умолчанию использует контейнерное хранилище для локальных образов

1.3.11. Полная настраиваемость

Контейнерные движки, как правило, имеют множество встроенных констант, определяющих, например, с какими пространствами имен работает движок, включен ли SELinux или с какими полномочиями запускаются контейнеры. В Docker большинство этих значений жестко закодированы и не могут быть изменены по умолчанию. У Podman, напротив, легко настраиваемая конфигурация.

Podman имеет встроенные настройки по умолчанию, но определяет три места для хранения файлов конфигурации:

- */usr/share/containers/containers.conf* — здесь дистрибутив может определить настройки, которые он предпочитает использовать;
- */etc/containers/containers.conf* — здесь можно установить системные переопределения;
- *$HOME/.config/containers/containers.conf* — задается только в непривилегированном режиме.

Конфигурационные файлы позволяют настроить Podman, чтобы он по умолчанию работал так, как вам нужно. Вы даже можете запустить его с более строгими настройками безопасности по умолчанию, если возникнет такая необходимость.

1.3.12. Поддержка пользовательского пространства имен

Podman полностью интегрирован с пользовательским пространством имен. Работа в режиме rootless зависит от пользовательских пространств имен, что позволяет назначать пользователю несколько UID. Пространства имен пользователей обеспечивают изоляцию пользователей в системе, поэтому может быть несколько пользователей без полномочий root, каждый из которых

запускает контейнеры с несколькими UID, и все они будут изолированы друг от друга.

Пользовательское пространство имен можно использовать для изоляции контейнеров друг от друга. Podman упрощает запуск нескольких контейнеров, каждый из которых имеет уникальное пространство имен пользователей. Затем ядро изолирует процессы от пользователей хоста и друг от друга с помощью разделения UID.

Docker поддерживает запуск контейнеров только в одном, отдельном пользовательском пространстве имен, это означает, что все контейнеры работают в одном и том же пространстве имен. Root в одном контейнере тот же самый, что и root в другом контейнере. Если не поддерживается запуск каждого контейнера в отдельном пространстве имен, следовательно, контейнеры могут атаковать друг друга через пользовательское пространство имен. Даже несмотря на поддержку такого режима Docker, почти никто не запускает контейнеры с Docker в отдельном пользовательском пространстве имен.

1.4. КОГДА НЕ СЛЕДУЕТ ИСПОЛЬЗОВАТЬ PODMAN

Как и Docker, Podman не является контейнерным оркестратором. Podman — это инструмент для запуска контейнеров на одном хосте либо в привилегированном (rootful), либо в непривилегированном (rootless) режиме. Если вы хотите организовать запуск контейнеров на нескольких машинах, требуются инструменты более высокого уровня.

Я считаю, что лучшим инструментом для этого сейчас является Kubernetes. Если говорить об узнаваемости, Kubernetes выиграл войну контейнерных оркестраторов. У Docker есть оркестратор под названием Swarm, который пользовался определенной популярностью, но сейчас, похоже, впал в немилость. Podman не поддерживает функциональность Swarm, поскольку команда Podman считает, что Kubernetes — лучший путь к организации работы контейнеров на нескольких машинах. Podman использовался для различных оркестраторов и применяется для grid/HPC-вычислений, а разработчики открытого исходного кода даже добавили его в Kubernetes.

РЕЗЮМЕ

- Технологии контейнеризации уже много лет, но внедрение образов и реестров позволило разработчикам улучшить способ поставки программного обеспечения.

- Podman — отличный контейнерный движок, подходящий практически для всех ваших однонодовых контейнерных проектов. Он полезен для разработки, создания и запуска контейнерных приложений.
- Podman так же прост в использовании, как и Docker, и имеет точно такой же интерфейс командной строки.
- Podman поддерживает REST API, что позволяет работать с контейнерами Podman через различные языки программирования и удаленные инструменты, например `docker-compose`.
- В отличие от Docker, у Podman есть такие замечательные особенности, как поддержка пользовательского пространства имен, несколько транспортов, настраиваемые реестры, интеграция с systemd, модель fork/exec и встроенный режим rootless.
- Podman лучше обеспечивает безопасность при работе с контейнерами.

2 Командная строка

В ЭТОЙ ГЛАВЕ

- ✓ Командная строка Podman
- ✓ Запуск приложения OCI
- ✓ Сравнение контейнеров и образов
- ✓ Примеры использования Podman
- ✓ Создание OCI-образа

Podman — отличный инструмент для создания и запуска контейнерных приложений. В этой главе вы начнете с создания простого веб-приложения, чтобы познакомиться с часто используемыми функциями командной строки Podman.

Если на вашей машине не установлен Podman, обратитесь сначала к приложению C, а затем вернитесь к этой главе — ее изучение предполагает, что Podman 4.1 или более поздней версии уже установлен. По всей вероятности, более ранние версии Podman тоже будут работать нормально, но все приведенные в книге примеры были протестированы с Podman 4.1. В качестве примера базового образа я использую образ registry.access.redhat. com/ubi8/httpd-24.

В главе 2 показано, что Podman является превосходным инструментом для работы с контейнерами. В этой главе мы вместе выполним сценарий создания

контейнерного приложения. Вы запустите контейнер, измените его содержимое, создадите образ и отправите его в реестр. Затем я объясню, как сделать это автоматически. На этом пути вы познакомитесь со многими командами, запускаемыми из командной строки Podman, и получите полное представление о работе с ним.

ПРИМЕЧАНИЕ Универсальные базовые образы (UBI, universal base images) используются повсеместно, но полностью поддерживается программное обеспечение контейнеров, разработанное и проверенное Red Hat, а также работающее в операционной системе Red Hat. Существуют сотни образов Apache, работающих подобно этому образу, их вы тоже можете опробовать.

Опытным пользователям Docker, возможно, захочется просто пролистать эту главу. Многое, о чем в ней говорится, вам уже известно. Но есть немало уникальных для Podman особенностей, таких как монтирование образов контейнеров (раздел 2.2.10) и различные транспорты (раздел 2.2.4). Начнем же с запуска нашего первого контейнера.

ПРИМЕЧАНИЕ Podman — проект с открытым исходным кодом, находящийся в стадии активной разработки. Podman поставляется в комплекте многих дистрибутивов Linux, а также Mac и Windows. Дистрибутивы могут комплектоваться более старыми версиями Podman без некоторых современных функций, о которых рассказывается в этой книге. Некоторые представленные в книге примеры рассчитаны на использование Podman 4.1 или более поздней версии. Если код из примера не работает, пожалуйста, обновите Podman до последней версии. Дополнительную информацию об установке Podman вы найдете в приложении С.

2.1. РАБОТА С КОНТЕЙНЕРАМИ

В реестрах контейнеров хранятся тысячи различных образов. Разработчики, администраторы, QA-инженеры и обычные пользователи в основном применяют команду `podman run` для загрузки и запуска, тестирования или изучения этих образов. Чтобы приступить к созданию контейнерных приложений, вам прежде всего надо начать работать с базовым образом. В наших примерах вы извлечете образ registry.access.redhat.com/ubi8/httpd-24 в контейнерное хранилище в вашем домашнем каталоге, запустите его и приступите к исследованию внутренней среды контейнера.

2.1.1. Исследование контейнеров

В этом разделе рассмотрим шаг за шагом стандартную команду Podman. Выполните команду `podman run`, которая обратится к реестру контейнеров

registry.access.redhat.com, загрузит образ и сохранит его локально в вашем домашнем каталоге:

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
```

Теперь я подробно разберу эту команду. По умолчанию `podman run` выполняет контейнерную команду на первом плане, пока контейнер не завершится. В этом случае вы попадаете в командную строку bash, работающую внутри контейнера и отображающую строку `bash-4.4$`. Когда вы выходите из этой командной строки bash, Podman останавливает контейнер.

В этом примере вы указали как `-ti` два параметра: `-t` и `-i`, которые дают Podman указание соединиться с терминалом. При этом потоки ввода, вывода и ошибок процесса bash в контейнере подключаются к вашему экрану, что позволяет вам действовать внутри контейнера:

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
```

Параметр `--rm` дает указание Podman удалить контейнер, как только тот завершится, освобождая все хранилище данного контейнера:

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
```

Далее вы указываете образ контейнера, с которым работаете, registry.access.redhat.com/ubi8/httpd-24. Команда `podman` обращается к реестру контейнеров registry.access.redhat.com и начинает скачивать образ ubi8/httpd-24:latest. Podman копирует несколько слоев (blobs), как показано в следующем листинге, и сохраняет их в локальном контейнерном хранилище. Вы видите ход выполнения процесса, пока слои образа копируются. Некоторые образы довольно большие, и их копирование может занимать много времени. Если позже вы запустите другой контейнер на том же образе, Podman пропустит этот шаг извлечения, поскольку у вас уже есть нужный образ в локальном хранилище контейнеров.

Листинг 2.1. Извлечение и запуск образа контейнера из реестра

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
Trying to pull registry.access.redhat.com/
➥ ubi8/httpd-24:latest...  <-- Установление контакта с реестром
Getting image source signatures
Checking if image destination supports signatures
Copying blob 296e14ee2414 skipped: already exists
Copying blob 356f18f3a935 skipped: already exists
Copying blob 359fed170a21
➥ [=======================>---------] 11.8MiB / 16.2MiB
Copying blob 226cafc3a0c6
➥ [=====>----------------------------]
➥ 10.1MiB / 61.1MiB
```

Извлечение слоя пропускается

Наконец, укажите исполняемый файл для запуска внутри контейнера, в данном случае bash:

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
...
bash-4.4$
```

> **ПРИМЕЧАНИЕ** У образов почти всегда есть команды по умолчанию, которые они выполняют. Команду нужно указывать только в том случае, если вы хотите переопределить приложение по умолчанию, с которым запускается образ. В случае с образом registry.access.redhat.com/ubi8/httpd-24 запускается веб-сервер Apache.

Находясь внутри оболочки bash контейнера, выполните cat /etc/os-release и обратите внимание, что это, скорее всего, другая ОС или другая ее версия, чем в /etc/os-release вне контейнера. Осмотритесь в контейнере и обратите внимание на то, насколько он отличается от вашего окружения:

```
bash-4.4$ grep PRETTY_NAME /etc/os-release
PRETTY_NAME="Red Hat Enterprise Linux 8.4 (Ootpa)"
```

На моем хосте в другом терминале та же команда выводит:

```
$ grep PRETTY_NAME /etc/os-release
PRETTY_NAME="Fedora Linux 35 (Workstation Edition Prerelease)"
```

Вернувшись в контейнер, вы заметите, что здесь доступно гораздо меньше команд:

```
bash-4.4$ ls /usr/bin | wc -l
525
```

Тем временем на хосте вы увидите:

```
$ ls -l /usr/bin | wc -l
3303
```

Выполните команду ps, чтобы узнать, какие процессы запущены внутри контейнера:

```
$ ps
PID TTY TIME CMD
1 pts/0 00:00:00 bash
2 pts/0 00:00:00 ps
```

Вы видите только два процесса: bash и команду ps. Излишне говорить, что на моей хост-машине запущены сотни процессов (включая эти два). Вы можете дополнительно изучить внутреннюю часть контейнера, чтобы получить представление о том, что происходит внутри него.

Закончив, вы выйдете из bash, и контейнер завершит свою работу. Поскольку вы выполняли скрипт с параметром --rm, Podman очистит все хранилище контейнера и удалит сам контейнер. Образ контейнера при этом остается в хранилище. Теперь, когда вы изучили внутреннюю работу контейнера, пришло время начать работу с приложением по умолчанию внутри контейнера.

2.1.2. Запуск контейнерного приложения

В предыдущем примере вы вызвали и запустили bash в контейнере, но вы запустили не то приложение, которое планировал разработчик. В следующем примере вы запустите актуальное приложение, удалив эту команду и запустив контейнер с несколькими новыми параметрами.

Для начала удалите параметры -ti и --rm, поскольку вам нужно, чтобы контейнер оставался запущенным после завершения работы команды podman. Вы не находитесь в оболочке, работающей в контейнере в интерактивном режиме, поскольку она просто запускает контейнеризованный веб-сервис:

```
$ podman run -d -p 8080:8080 --name myapp registry.access.redhat.com/ubi8/httpd-24
37a1d2e31dbf4fa311a5ca6453f53106eaae2d8b9b9da264015cc3f8864fac22
```

Первый параметр, на который следует обратить внимание, это параметр -d (--detach). Он указывает Podman запустить контейнер, а затем отсоединиться от него, то есть по сути запустить контейнер в фоновом режиме. Команда Podman фактически завершает работу и оставляет контейнер запущенным. В главе 6 более подробно рассмотрено, что при этом происходит «за кулисами»:

```
$ podman run -d -p 8080:8080 --name myapp
     registry.access.redhat.com/ubi8/httpd-24
```

Параметр -p (--publish) указывает Podman «публиковать» или связывать порт контейнера 8080 с портом хоста 8080, когда контейнер запущен. При использовании параметра -p поле перед двоеточием относится к порту хоста, а поле после двоеточия — к порту контейнера. В данном случае видно, что порты одинаковые. Если вы указываете только один порт, Podman считает его портом контейнера и случайным образом выбирает порт хоста, к которому привязывает порт контейнера. С помощью команды podman port можно узнать, какие порты привязаны к контейнеру.

Листинг 2.2. Пример команды podman port

```
$ podman port myapp
8080/tcp -> 0.0.0.0:8080
```

Показывает, что порт 8080/tcp внутри контейнера привязан ко всем сетям хоста (0.0.0.0) через порт 8080

По умолчанию контейнеры создаются в собственном сетевом пространстве имен, то есть они привязаны не к сети хоста, а к своей виртуализированной сети. Предположим, я запускаю контейнер без параметра `-p`. В этом случае сервер Apache в контейнере привязывается к сетевому интерфейсу в сетевом пространстве имен контейнера, но Apache при этом не связан с сетью хоста.

Только процессы внутри контейнера могут подключаться к порту `8080` для связи с веб-сервером. Выполняя команду с параметром `-p`, Podman подключает порт изнутри контейнера к сети хоста через указанный порт. Это соединение позволяет таким внешним процессам, как веб-браузер, считывать данные с веб-сервиса.

> **ПРИМЕЧАНИЕ** Если вы запускаете контейнеры в режиме rootless, о котором говорится в главе 3, пользователям Podman ядро запрещает по умолчанию связываться с портами, номера которых меньше 1024. Некоторые контейнеры стремятся привязываться к портам с меньшими номерами, например к порту 80, который разрешен внутри контейнера, но `-p 80:80` не работает, так как 80 меньше 1024. Использование `-p 8080:80` заставляет Podman привязать порт `8080` хоста к порту `80` внутри контейнера. Репозиторий Podman содержит информацию по устранению таких проблем, как привязка к портам с номерами меньше 1024 и многих других (http://mng.bz/69ry).

Параметр `-p` сопоставит номера портов внутри контейнера с различными номерами портов вне контейнера:

```
$ podman run -d -p 8080:8080 --name myapp
     registry.access.redhat.com/ubi8/httpd-24
```

В этом примере контейнер `myapp` имеет параметр `--name myapp`. Указание имени упрощает поиск контейнера; имя в дальнейшем можно использовать для других команд (например, `podman stop myapp`). Если вы не укажете имя, Podman автоматически создаст уникальное имя контейнера вместе с его идентификатором (ID). Все команды Podman, взаимодействующие с контейнерами, могут использовать либо имя, либо ID:

```
$ podman run -d --name myapp -p 8080:8080
     registry.access.redhat.com/ubi8/httpd-24
```

Когда команда `podman run` завершится, контейнер все еще будет запущен. Поскольку этот контейнер работает в фоновом режиме, Podman выводит идентификатор контейнера и завершает работу, но контейнер остается запущенным:

```
$ podman run -d -p 8080:8080 --name myapp
     registry.access.redhat.com/ubi8/httpd-24
37a1d2e31dbf4fa311a5ca6453f53106eaae2d8b9b9da264015cc3f8864fac22
```

Теперь, когда контейнер работает, вы можете запустить веб-браузер для связи с веб-сервером внутри контейнера через локальный порт 8080 (рис. 2.1):

```
$ web-browser localhost:8080
```

Поздравляем! Вы запустили свое первое контейнерное приложение.

Теперь представьте, что вы хотите запустить другой контейнер. Вы можете выполнить похожую команду всего лишь с парой изменений:

```
$ podman run -d -p 8081:8080 --name myapp1 \
➥ registry.access.redhat.com/ubi8/httpd-24
fa41173e4568a8fa588690d3177150a454c63b53bdfa52865b5f8f7e4d7de1e1
```

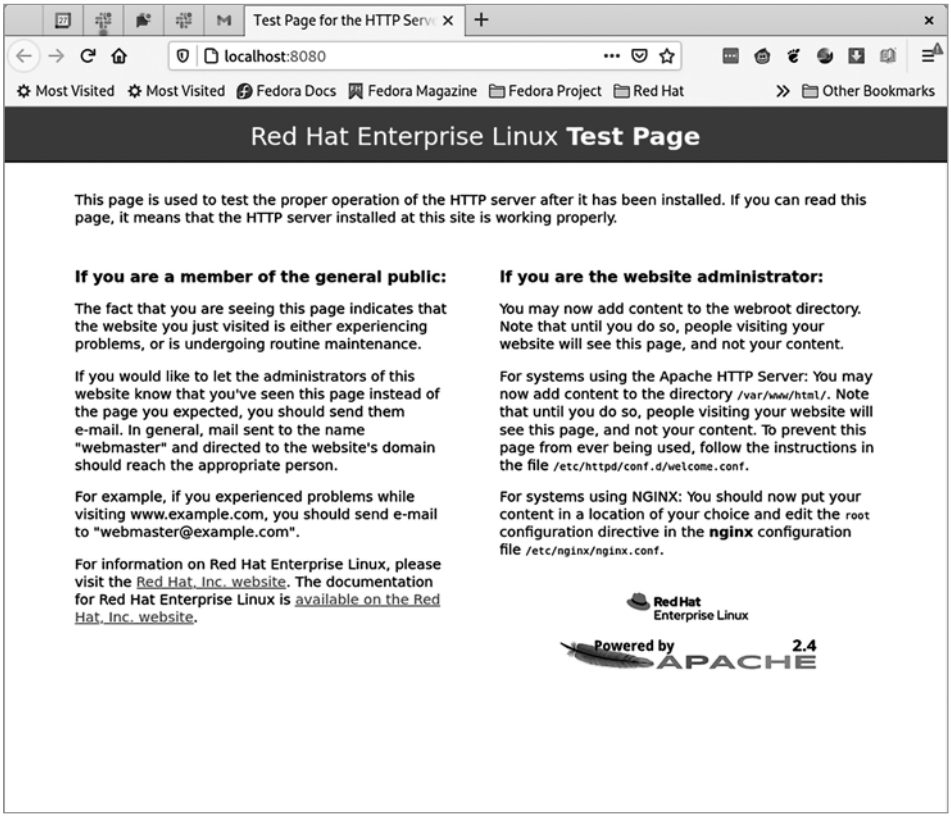


Рис. 2.1. Окно веб-браузера, которое подключается к контейнеру ubi8/httpd-24, работающему в Podman

Обратите внимание, вам нужно изменить имя контейнера на `myapp1`; в противном случае команда `podman run` завершится ошибкой, поскольку попробует использовать прежде указанное имя `myapp`, а такой контейнер уже существует. Также вам нужно изменить параметр `-p`, указав `8081` для порта хоста, потому что предыдущий контейнер, `myapp`, в настоящее время работает и привязан к порту `8080`. Второй контейнер не может быть привязан к порту `8080`, пока первый контейнер не завершится:

```
$ podman run -d -p 8081:8080 --name myapp1
     registry.access.redhat.com/ubi8/httpd-24
```

Команда `podman create` почти идентична команде `podman run`. Команда `create` загружает образ, если он не находится в хранилище контейнеров, и настраивает информацию о контейнере, делая его готовым к запуску, но никогда не запускает контейнер. Она часто используется вместе с командой `podman start`, описанной в разделе 2.1.4. Возможно, вам понадобится создать контейнер, а затем использовать файл systemd-юнита для запуска и остановки этого контейнера.

Ниже приводятся некоторые важные параметры команды `podman run`:

- `--user USERNAME`. Указывает Podman на запуск контейнера от имени определенного пользователя, указанного в образе. Если в образе контейнера не указан пользователь, то по умолчанию Podman будет запускать контейнер от имени root.
- `--rm`. Автоматически удаляет контейнер, завершивший свою работу.
- `--tty -(t)`. Выделяет псевдо `-tty` и подключает его к стандартному вводу контейнера.
- `--interactive (-i)`. Подключает `stdin` к основному процессу контейнера. Эти параметры обеспечивают интерактивную оболочку внутри контейнера.

ПРИМЕЧАНИЕ У команды `podman run` есть десятки параметров, позволяющих изменять характеристики безопасности, пространства имен, томов и т. д. Некоторые из них я использую и объясняю по ходу книги. Описание всех параметров смотрите на man-странице `podman run`. Большинство параметров `podman create`, указанных в табл. 2.1, также доступны для `podman run`.

Для получения информации обо всех параметрах используйте команду `man podman-run`. Теперь, когда контейнер запущен и работает, пришло время его остановить и перейти к следующему шагу.

2.1.3. Остановка контейнеров

Вы запустили два контейнера и протестировали их через веб-браузер. Чтобы продолжить разработку и добавить некое содержимое на веб-страницу, остановите контейнеры с помощью команды `podman stop`:

```
$ podman stop myapp
```

Команда `stop` останавливает контейнер, запущенный предыдущей командой `podman run`.

При остановке контейнера Podman проверяет работающий контейнер и отправляет сигнал остановки, обычно `SIGTERM`, основному процессу (PID1) контейнера, а затем по умолчанию ждет 10 секунд, пока контейнер остановится. Сигнал остановки сообщает основному процессу внутри контейнера о необходимости корректного выхода. Если контейнер не останавливается в течение 10 секунд, Podman отправляет процессу сигнал `SIGKILL`, принудительно заставляя контейнер остановиться. Десятисекундное ожидание дает процессам в контейнере время для очистки и фиксации изменений.

Сигнал остановки по умолчанию для контейнера можно изменить с помощью параметра `podman run --stop-signal`. Иногда основной процесс или init-процесс контейнера игнорирует `SIGTERM` (например, контейнеры, использующие systemd в качестве основного процесса внутри контейнера). systemd игнорирует `SIGTERM` и указывает, что он выключается, используя сигнал `SIGRTMIN+3` (сигнал #37). Сигнал остановки может быть встроен в образы контейнеров, как показано в разделе 2.3.

Некоторые контейнеры игнорируют стоп-сигнал `SIGTERM`, а это означает, что вам придется подождать 10 секунд, пока контейнер завершится. Если известно, что контейнер игнорирует сигнал остановки по умолчанию, и вас не волнует очистка контейнера, просто добавьте параметр `-t 0` к команде `podman stop`, чтобы сразу отправить сигнал `SIGKILL`:

```
$ podman stop -t 0 myapp1
myapp1
```

У Podman есть похожая команда `podman kill`, которая отправляет такой же сигнал (kill — уничтожить, аннулировать). Команда `podman kill` может быть полезна, если нужно отправить сигналы в контейнер, фактически его не останавливая.

Ниже приводятся некоторые часто используемые параметры команды `stop` в Podman:

- `--timeout (-t)`. Устанавливает тайм-аут. `t -0` отправляет `SIGKILL`, не ожидая, пока контейнер остановится.

- `--latest` (`-l`). Полезный параметр, который позволяет остановить последний созданный контейнер без использования его имени или идентификатора. Большинство команд Podman, требующих указания имени или ID контейнера, также принимают параметр `--latest`. Параметр доступен только на машинах Linux.
- `--all`. Дает указание Podman остановить все запущенные контейнеры. Аналогично `--latest`, команды Podman, требующие указания параметра имени контейнера или его ID, также принимают параметр `--all`.

Для получения информации обо всех параметрах используйте команду `man podman-stop`.

В конечном итоге в вашей системе окажется много остановленных контейнеров, и иногда вам нужно будет перезапустить их (например, после перезагрузки системы). Другой распространенный вариант действий — сначала создать контейнер, а затем запустить его. В следующем разделе объясняется, как запустить контейнер.

2.1.4. Запуск контейнеров

На данном этапе созданный вами контейнер остановлен. Вы можете запустить его снова, используя команду из следующего листинга.

Листинг 2.3. Пример запуска контейнера

```
$ podman start myapp
myapp
```

Команда start выводит имена контейнеров, которые были запущены

Команда `podman start` запускает один или несколько контейнеров. Показывая, что ваш контейнер запущен и работает, эта команда выводит ID контейнера. Теперь вы можете подключиться к нему с помощью веб-браузера. Одним из распространенных вариантов использования `podman start` является запуск контейнера после перезагрузки. Так запускают все контейнеры, которые были остановлены во время завершения работы.

Ниже приводятся некоторые часто используемые параметры запуска:

- `--all`. Запускает все остановленные контейнеры в контейнерном хранилище.
- `--attach`. Подключает ваш терминал к выводу контейнера.
- `--interactive` (`-i`). Присоединяет ввод терминала к контейнеру.

Для получения информации обо всех параметрах используйте команду `man podman-start`.

Вы уже некоторое время используете Podman, вы собрали и запустили множество различных образов контейнеров, и теперь настала пора выяснить, какие контейнеры запущены или какие контейнеры у вас находятся в локальном хранилище. Вам понадобится возможность выводить списки этих контейнеров.

2.1.5. Список контейнеров

Существует возможность отобразить запущенные контейнеры и все контейнеры, которые были созданы ранее. Для вывода списка контейнеров используйте команду `podman ps`:

```
$ podman ps
CONTAINER ID IMAGE                          COMMAND        CREATED \
➥    STATUS         PORTS           NAMES
b1255e94d084 registry.access.redhat.com/ubi8/httpd-24:latest /usr/bin/run-\
➥ http... 6 minutes ago Up 4 minutes ago 0.0.0.0:8080->8080/tcp myapp
```

Обратите внимание, что команда `podman ps` по умолчанию выводит список только запущенных контейнеров. Чтобы увидеть все контейнеры, используйте параметр `--all`:

```
$ podman ps --all
CONTAINER ID IMAGE                          COMMAND        CREATED \
➥    STATUS          PORTS           NAMES
b1255e94d084 registry.access.redhat.com/ubi8/httpd-24:latest /usr/bin/run-\
➥ http... 9 minutes ago Up 8 minutes ago 0.0.0.0:8080->8080/tcp myapp
3efee4d39965 registry.access.redhat.com/ubi8/httpd-24:latest /usr/bin/run-\
➥ http... 7 minutes ago Exited (0) 3 minutes ago 0.0.0.0:8081->8080/tcp myapp1
```

Ниже приводятся некоторые часто используемые параметры команды `podman ps`:

- `--all`. Дает указание Podman вывести список всех контейнеров, а не только запущенных.
- `--quiet`. Дает указание Podman показать только идентификаторы контейнеров.
- `--size`. Дает указание Podman вывести объем дискового пространства, используемого в настоящее время для каждого контейнера, а не для образов, на которых они основаны.

Для получения информации обо всех параметрах используйте команду `man podman-ps`.

Теперь вам известны все контейнеры, имеющиеся в системе, и можно обратиться к их внутреннему устройству.

2.1.6. Инспектирование контейнеров

Чтобы полностью разобраться в контейнере, иногда не обойтись без информации о том, на каком образе он был основан, какие переменные окружения получает по умолчанию или какие настройки безопасности используются для этого контейнера. Команда `podman ps` предоставляет нам некоторые данные о контейнерах, но если вам действительно нужно узнать всё о контейнере, используйте команду `podman inspect`.

Команда `podman inspect` также применяется для инспектирования образов, сетей, томов и подов. Есть и команда `podman container inspect`, специально для контейнеров. Но большинство пользователей просто вводят более короткую команду `podman inspect`:

```
$ podman inspect myapp
[
  {
    "Id":"240271ae90480d3836b1477e5c0b49fbd3883846ca474e3f6effdfb271f4ff54",
    "Created": "2021-09-27T05:27:47.163828842-04:00",
    "Path": "container-entrypoint",
    "Args": [
      "/usr/bin/run-httpd"
    ],
...
]
```

Как видите, команда `podman inspect` выводит большой JSON-файл — 307 строк на моей машине. Вся эта информация в конечном итоге передается в среду выполнения OCI для запуска контейнера. При работе с командой `inspect` бывает удобнее направить ее вывод в `less` или `grep`, чтобы найти именно те поля, которые представляют для вас интерес. В качестве альтернативы можно использовать параметр `--format`. Если вы хотите изучить команду, выполняемую при запуске контейнера, выполните код из листинга 2.4.

Листинг 2.4. Просмотр команды, заданной для запуска контейнера

```
$ podman inspect --format '{{ .Config.Cmd }}' myapp
[/usr/bin/run-httpd]
```

inspect отображает данные из спецификации OCI образа

Если же вам нужно увидеть сигнал остановки, выполните код из листинга 2.5.

Листинг 2.5. Проверка сигнала остановки, который будет использоваться при остановке контейнера

```
$ podman inspect --format '{{ .Config.StopSignal }}' myapp
15
```

Сигнал остановки по умолчанию для всех контейнеров — 15 (SIGTERM)

Ниже приводятся некоторые часто используемые параметры команды `podman inspect`:

- `--latest (-l)`. Позволяет быстро проверить последний созданный контейнер без указания имени или идентификатора контейнера.
- `--format`. Полезен для извлечения определенных полей из JSON.
- `--size`. Добавляет в вывод объем дискового пространства, используемого контейнером. Сбор этой информации занимает много времени, поэтому по умолчанию он не выполняется.

Для получения информации обо всех параметрах используйте команду `man podman-inspect`.

После инспектирования контейнера вы могли прийти к выводу, что он вам больше не понадобится. Ненужный контейнер занимает место в хранилище, поэтому его необходимо удалить.

2.1.7. Удаление контейнеров

Закончив работу с контейнером, следует удалить его, чтобы освободить место на диске или повторно использовать имя этого контейнера. Помните, как вы запустили второй контейнер под названием `myapp1`? Он вам больше не нужен, поэтому удалим его. Обязательно остановите контейнер (раздел 2.1.3) перед его удалением. Затем используйте команду `podman rm` для удаления контейнера:

```
$ podman rm myapp1
3efee4d3996532769356ffea23e1f50710019d4efc704d39026c5bffd6aa18be
```

Ниже приводятся некоторые часто используемые параметры команды `podman rm`:

- `-all`. Этот параметр полезен, если вы хотите удалить все ваши контейнеры.
- `--force`. Параметр передает указание Podman остановить все запущенные контейнеры при удалении.

Для получения информации обо всех параметрах используйте команду `man podman-rm`.

Вы разобрались с несколькими командами, и теперь пора приступать к модификации работающего контейнера.

2.1.8. Выполнение команд внутри контейнера

Когда контейнер запущен, нередко требуется запустить другой процесс внутри контейнера для отладки или проверки того, что происходит внутри. В некоторых случаях может потребоваться изменить часть содержимого, используемого контейнером.

Представьте, что вам нужно зайти в свой контейнер и изменить веб-страницу, которую он показывает. Вы можете зайти в контейнер с помощью команды `podman exec`. Используйте параметр `--interactive (-i)`, чтобы иметь возможность выполнять команды внутри контейнера. Вам нужно указать имя контейнера `myapp` и выполнить bash-скрипт, находясь в контейнере. Если вы остановили контейнер `myapp`, придется перезапустить его, так как `podman exec` работает только с запущенными контейнерами.

В следующем примере вы запустите процесс bash в контейнере, чтобы создать файл /var/www/html/index.html. Вы запишете HTML-содержимое, которое заставит веб-сайт в контейнере отобразить Hello World:

```
 $ podman exec -i myapp bash -c 'cat > /var/www/html/index.html' << _EOF
<html>
  <head>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
_EOF
```

Выполнив `exec` во второй раз, вы увидите, что файл был успешно изменен. Это показывает, что изменения, внесенные в контейнер с помощью `exec`, являются постоянными для контейнера и сохранятся, даже если вы остановите и перезапустите его. Ключевое различие между `podman run` и `podman exec` заключается в том, что `run` создает новый контейнер на основе образа с запущенными внутри процессами, а `exec` запускает процессы внутри существующих контейнеров:

```
 $ podman exec myapp cat /var/www/html/index.html
<html>
  <head>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Давайте подключим веб-браузер к контейнеру, чтобы посмотреть, изменилось ли содержимое (рис. 2.2):

```
$ web-browser localhost:8080
```

Рис. 2.2. Окно веб-браузера, который подключается к контейнеру ubi8/httpd-24, работающему в Podman, с обновленным HTML-содержимым «Hello World»

Ниже приводятся некоторые часто используемые параметры команды `podman exec`:

- `--tty -(t)`. Параметр подключает `-tty` к сессии команды `exec`.
- `--interactive`. Параметр `-i` указывает Podman на запуск в интерактивном режиме, давая вам возможность взаимодействовать с исполняемой программой как с оболочкой.

Для получения информации обо всех параметрах используйте команду `man podman-exec`.

После создания приложения у вас может возникнуть желание поделиться им с другими. Для начала вам нужно зафиксировать контейнер в образ.

2.1.9. Создание образа из контейнера

Разработчики часто запускают контейнеры из базового образа для создания нового контейнерного окружения. После завершения работы они упаковывают это окружение в образ, чтобы поделиться им с другими пользователями. Эти пользователи затем запускают контейнерное приложение, используя Podman. Вы можете сделать это с помощью Podman, зафиксировав контейнер в OCI-образ.

Сначала остановите или поставьте на паузу контейнер, чтобы убедиться, что ничего не будет изменено во время коммита:

```
$ podman stop myapp
```

Теперь выполните команду `podman commit`, чтобы зафиксировать контейнер приложения `myapp`, создав новый образ с именем `myimage`:

```
$ podman commit myapp myimage
Getting image source signatures
Copying blob e39c3abf0df9 skipped: already exists
Copying blob 8f26704f753c skipped: already exists
Copying blob 83310c7c677c skipped: already exists
Copying blob 654b3bf1361e skipped: already exists
Copying blob 9e816183404c done Copying config e38084bb8a done
Writing manifest to image destination
Storing signatures
e38084bb8a76104a7cac22b919f67646119aff235bb1cfcba5478cc1fbf1c9eb
```

Можно продолжить работу существующего контейнера `myapp`, вызвав `podman start`, или создать новый контейнер на основе `myimage`:

```
$ podman run -d --name myapp1 -p 8080:8080 myimage
0052cb32c8e63b845ac5dfd5ba176b8204535c2c6cafa3277453424de601263f
```

> **ПРИМЕЧАНИЕ** Использование команды `podman commit` не является общепринятым методом для создания образа. Весь процесс создания образов контейнеров программируется и автоматизируется с помощью `podman build`. Дополнительная информация об этом процессе приведена в разделе 2.3.

Ниже приводятся некоторые часто используемые параметры команды `podman commit`:

- `--pause`. Приостанавливает работающий контейнер во время коммита (фиксации). Обратите внимание, что я остановил контейнер перед выполнением коммита, хотя мог бы просто поставить его на паузу. Команды `podman pause` и `podman unpause` позволяют напрямую останавливать и возобновлять работу контейнеров.
- `--change`. Параметр позволяет зафиксировать в образе следующие инструкции: `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`, `LABEL`, `ONBUILD`, `STOPSIGNAL`, `USER`, `VOLUME` и `WORKDIR`. Эти инструкции совпадают с директивами в Containerfile или Dockerfile.

Для получения информации обо всех параметрах используйте команду `man podman-commit`. В табл. 2.1 перечислены все команды Podman container.

Теперь, когда вы закоммитили свой контейнер в образ, пришло время показать, как Podman может работать с образами.

> **ПРИМЕЧАНИЕ** Вы изучили лишь несколько команд работы с контейнерами Podman, но их гораздо больше. Используйте справочные страницы `podman-container(1)`, чтобы ознакомиться со всеми командами, а также получить более полное описание команд, о которых мы говорили в этом разделе.

Таблица 2.1. Команды podman container

Команда	man-страница	Описание
attach	podman-container-attach(1)	Присоединяется к работающему контейнеру
checkpoint	podman-container-checkpoint(1)	Создает контрольную точку контейнера
cleanup	podman-container-cleanup(1)	Очищает сети и точки монтирования контейнера
commit	podman-container-commit(1)	Фиксирует контейнер в образ
cp	podman-container-cp(1)	Копирует файлы или каталоги в контейнеры и из них
create	podman-container-create(1)	Создает новый контейнер
diff	podman-container-diff(1)	Проверяет изменения в файловой системе контейнера
exec	podman-container-exec(1)	Запускает процесс в контейнере
exists	podman-container-exists(1)	Проверяет, существует ли контейнер
export	podman-container-export(1)	Экспортирует файловую систему контейнера в виде архива TAR
init	podman-container-init(1)	Инициализирует контейнер
inspect	podman-container-inspect(1)	Отображает подробную информацию о контейнере
kill	podman-container-kill(1)	Отправляет сигнал основному процессу в контейнере
list (ps)	podman-container-list(1)	Выводит список всех контейнеров
logs	podman-container-logs(1)	Извлекает логи контейнера
mount	podman-container-mount(1)	Монтирует корневую файловую систему контейнера
pause	podman-container-pause(1)	Приостанавливает контейнер
port	podman-container-port(1)	Выводит список сопоставления портов для контейнера
prune	podman-container-prune(1)	Удаляет все незапущенные контейнеры
rename	podman-container-rename(1)	Переименовывает существующий контейнер
restart	podman-container-restart(1)	Перезапускает контейнер

Команда	man-страница	Описание
`restore`	`podman-container-restore(1)`	Восстанавливает контейнер из контрольной точки
`rm`	`podman-container-rm(1)`	Удаляет контейнер
`run`	`podman-container-run(1)`	Запускает команду в новом контейнере
`runlabel`	`podman-container-runlabel(1)`	Выполняет команду, описанную в метке образа
`start`	`podman-container-start(1)`	Запускает контейнер
`stats`	`podman-container-stats(1)`	Выводит статистику контейнера
`stop`	`podman-container-stop(1)`	Останавливает контейнер
`top`	`podman-container-top(1)`	Отображает запущенные процессы в контейнере
`unmount`	`podman-container-unmount(1)`	Размонтировывает корневую файловую систему контейнера
`unpause`	`podman-container-unpause(1)`	Возобновляет работу одного или нескольких контейнеров
`wait`	`podman-container-wait(1)`	Ожидает завершения работы контейнера

2.2. РАБОТА С ОБРАЗАМИ КОНТЕЙНЕРОВ

В предыдущем разделе вы пробовали совершать базовые операции с контейнерами, включая инспектирование и коммит образа контейнера. В этом разделе вы попробуете работать с образами, узнаете, чем они отличаются от контейнеров и как делиться ими через реестры контейнеров.

2.2.1. Разница между контейнером и образом

Одна из проблем сферы IT заключается в том, что одни и те же имена постоянно используются для разных целей. В мире контейнеров нет более часто используемого термина, чем *контейнер*. Часто *контейнер* связывают с действующими процессами, запущенными Podman. Но *контейнер* также может относиться к данным контейнера как к неработающим объектам, находящимся в хранилище контейнера. Как было сказано в предыдущем разделе, `podman ps --all` показывает работающие и неработающие контейнеры.

Другой пример — термин *пространство имен*. Я часто путаюсь, когда говорят о пространствах имен в Kubernetes. Некоторые люди, услышав этот термин, представляют себе *виртуальные кластеры*, а я в таких случаях думаю о пространствах имен Linux, используемых подами и контейнерами. А *образ* может относиться к образу виртуальной машины, образу контейнера, образу OCI или образу Docker, хранящемуся в реестре контейнеров.

Я считаю контейнеры выполняющимися процессами в некотором окружении либо чем-то, что готовится к запуску. Напротив, образы — это *зафиксированные* (committed) *контейнеры*, которые подготовлены для передачи. Другие пользователи или системы могут использовать эти образы для создания новых контейнеров.

Образы контейнеров — это просто зафиксированные контейнеры. OCI определяет формат образа. Podman использует библиотеку containers/image (https://github.com/containers/image) для всех взаимодействий с образами. Образы контейнеров могут храниться в различных типах хранилищ или транспортов, как они называются в библиотеке containers/image. Этими транспортами могут быть реестры контейнеров, архивы Docker, архивы OCI, `docker-daemon`, а также containers/storage. Транспорты более подробно рассмотрены в разделе 2.2.4.

В контексте Podman под образами я обычно подразумеваю содержимое, хранящееся локально в контейнерном хранилище или в реестрах контейнеров, таких как docker.io и quay.io. Podman использует библиотеку GitHub containers/storage (https://github.com/containers/storage) для работы с локально хранимыми образами. Давайте рассмотрим ее подробнее.

Библиотека containers/storage представляет концепцию контейнера хранения (storage container). По сути, контейнеры хранения — это промежуточное содержимое хранилища, которое еще не было зафиксировано. Представьте их в виде файлов на диске и некоторого JSON, описывающего содержимое. У Podman есть собственное хранилище данных, связанных с контейнерами Podman, и он должен иметь дело с несколькими пользователями своих контейнеров одновременно. Он полагается на блокировку файловой системы, обеспечиваемую библиотекой containers/storage, чтобы сотни исполняемых файлов Podman могли надежно использовать одно и то же хранилище данных.

Когда вы коммитите контейнер в хранилище, Podman копирует хранилище контейнера в хранилище образов. Образы хранятся в виде ряда слоев, и каждый коммит создает новый слой.

Мне нравится думать об образе как о свадебном торте (рис. 2.3). В нашем предыдущем примере у нас был образ ubi8/httpd-24, состоящий из двух слоев: базовый слой — ubi8, а затем в этот образ был добавлен пакет `httpd` и несколько других пакетов для создания ubi8/httpd-24. Когда вы фиксируете свой контейнер как

описано в предыдущем разделе, Podman добавляет поверх образа ubi8/httpd-24 еще один слой, который называется `myimage`.

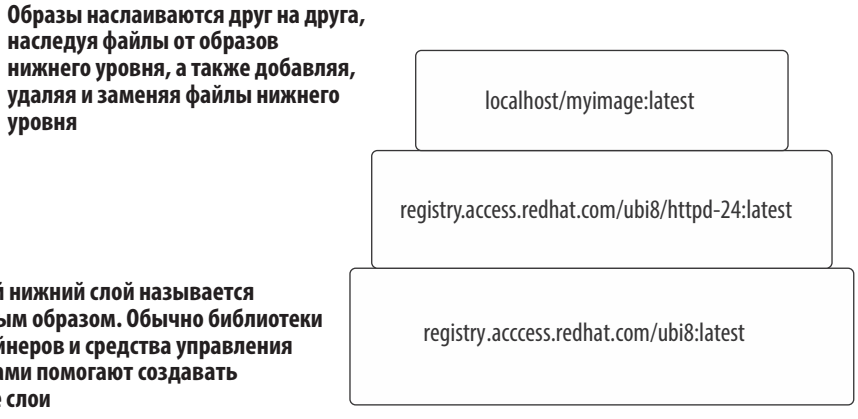


Рис. 2.3. Изображение свадебного торта, показывающее образы, из которых состоит наше приложение «Hello World»

В Podman есть полезная команда для отображения слоев образа `podman image tree`:

```
$ podman image tree myimage
Image ID: 2c7e43d88038
Tags: [localhost/myimage:latest]
Size: 461.7MB
Image Layers
├── ID: e39c3abf0df9 Size: 233.6MB
├── ID: 42c81bd2b468 Size: 20.48kB Top Layer of:
    [registry.access.redhat.com/ubi8:latest]
├── ID: 51a7beaa0b88 Size: 57.43MB
├── ID: 519e681b5702 Size: 170.6MB Top Layer of:
     [registry.access.redhat.com/ubi8/httpd-24:latest]
└── ID: bc3dcdefdac3 Size: 69.63kB Top Layer of:[localhost/myimage:latest
localhost/myapp:latest]
```

Как видите, образ `myimage` состоит из пяти слоев.

Другая полезная команда, `podman image diff`, позволяет увидеть фактические файлы и каталоги, которые были изменены (C), добавлены (A) или удалены (D) по сравнению с другим образом или нижним слоем:

```
$ podman image diff myimage ubi8/httpd-24
C /etc/group
C /etc/httpd/conf
C /etc/httpd/conf/httpd.conf
C /etc/httpd/conf.d
```

```
C /etc/httpd/conf.d/ssl.conf
C /etc/httpd/tls
C /etc
C /etc/httpd
A /etc/httpd/tls/localhost.crt
A /etc/httpd/tls/localhost.key
...
```

Образы — это просто TAR-diffs, применяемые на образах нижнего слоя, а содержимое контейнера — незафиксированный слой. Когда контейнер зафиксирован, вы создаете другие контейнеры поверх своего образа. Вы также можете поделиться образом с другими пользователями, чтобы они могли создавать другие контейнеры на основе вашего образа.

Теперь давайте посмотрим на все образы в вашем хранилище контейнеров.

2.2.2. Вывод списка образов

В разделе о контейнерах вы работали с образами и использовали команду `podman images`, чтобы вывести список образов в локальном хранилище:

```
$ podman images
REPOSITORY                   TAG       IMAGE ID        CREATED      SIZE
localhost/myimage          latest    2c7e43d88038   46 hours ago    462 MB
registry.access.redhat
➥.com/ubi8/httpd-24       latest    8594be0a0b57    5 weeks ago    462 MB
registry.access.redhat
➥.com/ubi8                latest    ad42391b9b46    5 weeks ago    234 MB
```

Давайте рассмотрим поля в выводе по умолчанию. В табл. 2.2 описаны поля и данные, доступные с помощью команды `podman images`. Команду `podman images` вы будете использовать на протяжении всего этого раздела.

Таблица 2.2. Поля по умолчанию, выводимые командой podman images

Заголовок	Описание
`Repository`	Полное имя образа
`TAG`	Версия (тег) образа. Тегирование образов рассматривается в разделе 2.2.6.
`IMAGE ID`	Уникальный идентификатор образа. Он генерируется Podman как хеш SHA256 JSON-объекта конфигурации этого образа
`CREATED`	Время, прошедшее с момента создания образа. По умолчанию образы сортируются по этому полю
`SIZE`	Объем хранилища, используемого образом

ПРИМЕЧАНИЕ Со временем объем хранилища, используемый всеми скачанными образами, увеличивается. Довольно часто у пользователей заканчивается свободное место на диске, поэтому вам стоит следить за размером образов и контейнеров, удаляя те, которые вы больше не используете. Для получения дополнительной информации об очистке используйте команду `man podman-system-prune`.

Одним из важных параметров команды `podman image` является `--all`. Этот параметр полезен для вывода списка всех образов. По умолчанию `podman-images` выводит список только тех образов, которые используются в данный момент. Когда образ заменяется более новым с тем же тегом, предыдущий образ помечается как `<none><none>`; такие образы называются висячими, или dangling, образами. О висячих образах рассказывается в разделе 2.3.1.

Для получения сведений обо всех параметрах используйте команду man `podman-images`. Скорее всего вы также захотите изучить информацию о конфигурации, связанную с образом, проинспектировав его, подобно тому, как это было сделано с контейнерами.

2.2.3. Инспектирование образов

В предыдущих разделах я упоминал несколько команд для исследования образов. Я использовал `podman image diff` для изучения файлов и каталогов, созданных или удаленных между образами. Я также показал, как увидеть структуру иерархии образов или слои «свадебного торта» из образов с помощью команды `podman image tree`.

Возможно, вам понадобилось изучить конфигурацию образа. Для этого служит команда `podman image inspect`. Команда `podman inspect` также может быть использована для исследования образов, но имена могут конфликтовать с контейнерами, поэтому я предпочитаю применять специальную команду с image:

```
$ podman image inspect myimage
[
  {
    "Id": "3b8fcf9081b4c4e6c16d763b8d02684df0737f3557a1e03ebfe4cc7cd6562135",
    "Digest": "sha256:ff49aa6253ae47569d5aadbd73d70e7d0431bcf3a2f57b1b56feecdb
➥ 531029a3",
    "RepoTags": [
      "localhost/myimage:latest"
    ],
    "RepoDigests": [ "localhost/myimage@sha256:ff49aa6253ae47569d5aadbd73d70e7d0
➥ 431bcf3a2f57b1b\56feecdb531029a3"
    ],
...
]
```

Как видите, эта команда выводит большой массив JSON — 153 строки в предыдущем примере, — который включает данные, используемые для спецификации OCI Image Format. Эта информация берется в качестве одного из входных данных для создания контейнера из образа.

При использовании команды `inspect` ее вывод лучше направить в `less` или `grep` для поиска интересующих вас полей. В качестве альтернативы можно добавить параметр `--format`.

Если вам нужно выяснить, какая команда по умолчанию будет выполняться из этого образа, используйте следующий код:

```
$ podman image inspect --format '{{ .Config.Cmd }}' myimage
[/usr/bin/run-httpd]
```

Если же вы хотите увидеть сигнал остановки, выполните такой код:

```
$ podman image inspect --format '{{ .Config.StopSignal }}' myimage
```

Как видите, ничего не выводится, значит, разработчик приложения не указал `STOPSIGNAL`. При сборке контейнера на основе этого образа `STOPSIGNAL` по умолчанию будет равен 15, если вы не переопределите его через командную строку.

Одним из важных параметров команды `podman image inspect` является `–format` — параметр, как было показано выше, полезен для извлечения определенных полей из json.

Для получения информации о `podman image inspect` используйте команду `man podman-image-inspect`.

Как только вы будете удовлетворены состоянием контейнера и зафиксируете его в образ, следующим шагом будет или поделиться им с другими пользователями, или, возможно, запустить его в другой системе. Вам нужно передать образ в другие типы хранилищ контейнеров, обычно в реестр контейнеров.

2.2.4. Передача образов

В Podman вы используете команду `podman push`, чтобы скопировать образ и все его слои из контейнерного хранилища и отправить в другие формы хранения образов, например в реестр контейнеров. Podman поддерживает несколько различных типов контейнерных хранилищ, которые он называет транспортами.

Библиотека containers/image (https://github.com/containers/image) служит для загрузки и отправки образов в Podman. Я описываю проект containers/image как библиотеку для копирования образов из различных типов контейнерных хранилищ. Одно из таких хранилищ — containers/storage.

При передаче образа место назначения указывается в формате `transport:ImageName`. Если транспорт не указан, по умолчанию используется `docker` (реестр контейнеров).

Одним из новаторских решений Docker, как я уже объяснял ранее, было изобретение концепции реестра контейнеров — по сути, веб-сервера, содержащего образы контейнеров. Веб-серверы docker.io, quay.io и Artifactory являются примерами контейнерных реестров. Команда разработчиков Docker определила протокол для получения и передачи этих образов из реестров контейнеров; я называю этот протокол реестром контейнеров или транспортом `docker`.

Запуская контейнер из образа, я могу полностью указать имя образа, включая транспорт, как в следующей команде:

```
$ podman run docker://registry.access.redhat.com/ubi8/httpd-24:latest echo hello
hello
```

В Podman транспорт `docker://` используется по умолчанию, и для удобства его можно опустить:

```
$ podman run registry.access.redhat.com/ubi8/httpd-24:latest echo hello
hello
```

Образ `myimage`, который вы создали в предыдущем разделе, был создан локально, и это означает, что он не связан ни с каким реестром. По умолчанию локально созданные образы связаны с реестром localhost. Вы можете просмотреть образы в containers/storage с помощью команды `podman images`:

```
$ podman images
REPOSITORY                TAG           IMAGE ID        CREATED      SIZE
localhost/myimage         latest    2c7e43d88038   46 hours ago    462 MB
registry.access.redhat
➥.com/ubi8/httpd-24      latest    8594be0a0b57   5 weeks ago     462 MB
registry.access.redhat
➥.com/ubi8               latest    ad42391b9b46   5 weeks ago     234 MB
```

Если образ имеет связанный с ним удаленный реестр (например, registry.access.redhat.com/ubi8), его можно отправить без указания поля «место назначения» или `[destination]`. И напротив, поскольку localhost/myimage не имеет связанного с ним реестра, удаленный реестр (например, quay.io/rhatdan) необходимо указать:

```
$ podman push myimage quay.io/rhatdan/myimage
Getting image source signatures
Copying blob 164d51196137 done
Copying blob 8f26704f753c done
Copying blob 83310c7c677c done
Copying blob 654b3bf1361e [==================>--------------------] 82.0MiB /
162.7MiB
Copying blob e39c3abf0df9 [================>----------------------] 100.0MiB /
222.8MiB
```

ПРИМЕЧАНИЕ Перед выполнением команды `podman push` я вошел в учетную запись quay.io/rhatdan с помощью команды `podman login`, которая рассматривается в следующем разделе.

После завершения команды `push` образ смогут извлекать для загрузки другие пользователи, имеющие доступ к этому реестру контейнеров. В табл. 2.3 описаны поддерживаемые транспорты для различных типов хранилищ контейнеров.

Таблица 2.3. Транспорты, поддерживаемые Podman

Транспорт	Описание
Реестр контейнеров (`docker`)	Транспорт по умолчанию. Ссылается на образ контейнера, хранящийся в удаленном реестре образов контейнеров. Реестр контейнеров — это место для хранения и обмена образами контейнеров (например, docker.io или quay.io)
`oci`	Ссылается на образ контейнера, соответствующий спецификации Open Container Image Layout Specification. Архивные файлы манифеста и слоев находятся в локальном каталоге в виде отдельных файлов
`dir`	Ссылается на образ контейнера, совместимый с макетом образа Docker. Он очень похож на транспорт `oci`, но хранит файлы, используя устаревший формат Docker. Это нестандартный формат, который в первую очередь полезен для отладки или неинвазивной (безопасной) инспекции контейнеров
`docker-archive`	Ссылается на образ контейнера, соответствующий макету образа Docker, который упакован в архив TAR
`oci-archive`	Ссылается на образ, соответствующий спецификации Open Container Image Layout Specification, который упакован в архив TAR. Он очень похож на транспорт `docker-archive`, но хранит образ в формате OCI
`docker-daemon`	Ссылается на образ, хранящийся во внутреннем хранилище Docker-демона. Поскольку демон Docker требует root-привилегий, Podman должен быть запущен от пользователя root
`container-storage`	Ссылается на образ, расположенный в локальном контейнерном хранилище. Это не транспорт, а скорее механизм для хранения образов. Его можно использовать для преобразования других транспортов в `container-storage`. По умолчанию Podman использует `container-storage` для локальных образов

При попытке отправить свой образ в реестр контейнеров ваш push окажется отклоненным, пока вы не предоставите информацию об авторизации для входа. Нужно выполнить `podman login`, чтобы пройти авторизацию.

2.2.5. podman login: аутентификация в реестре контейнеров

В предыдущем разделе я поместил образ в реестр контейнеров, выполнив следующее:

```
$ podman push myimage quay.io/rhatdan/myimage
```

Однако я упустил ключевой шаг: вход в реестр контейнеров с использованием правильных учетных данных. Это необходимый этап для отправки образа контейнера. Он требуется также для загрузки образа контейнера из частного реестра.

Чтобы следовать инструкциям в этом разделе, вам необходимо настроить учетную запись в реестре контейнеров. На выбор доступно несколько реестров. Реестры https://quay.io и https://docker.io предоставляют бесплатные учетные записи и хранилище. Если у вашей компании есть частный реестр, вы также можете получить учетную запись в нем.

В примерах я продолжу использовать свою учетную запись rhatdan на quay.io. Войдите в систему, чтобы получить нужные права:

```
$ podman login quay.io
Username: rhatdan
Password:
Login Succeeded!
```

Обратите внимание, что команда Podman запрашивает имя пользователя и пароль в реестре. У команды `podman login` есть параметры для передачи информации об имени пользователя/пароле в командной строке, чтобы обойтись без запроса. Это позволяет автоматизировать процесс входа.

Для хранения информации об аутентификации пользователя команда `podman login` создает файл auth.json. По умолчанию он хранится в каталоге /run/user/$UID/containers:

```
cat /run/user/3267/containers/auth.json
{
  "auths": {
    "quay.io": {
      "auth": "OBSCURED-BASE64-PASSWORD"
    }
  }
}
```

Файл auth.json содержит пароль вашего реестра в виде строки в кодировке Base64, шифрование не используется. Поэтому файл auth.json должен быть защищен. По умолчанию Podman хранит его в каталоге /run, так как файловая

система — временная, она уничтожается при выходе из системы или ее перезагрузке. Каталог /run/user/$UID/containers недоступен для других пользователей системы.

Для переопределения расположения файла указывается параметр `--auth-file`. В качестве альтернативы для изменения его местоположения можно использовать переменную окружения `REGISTRY_AUTH_FILE`. Если указаны оба параметра, то применяется параметр `--auth-file`. Все контейнерные инструменты используют этот файл для доступа к реестру контейнеров.

Команду `podman login` можно запускать несколько раз для входа в несколько реестров, сохраняя информацию для входа в разные разделы одного и того же файла авторизации.

> **ПРИМЕЧАНИЕ** Podman поддерживает и другие механизмы для хранения информации о пароле. Они называются *credential helpers*.

Закончив работу с реестром, выйдите из системы, выполнив `podman logout`. Эта команда удаляет кэшированные учетные данные, хранящиеся в файле auth.json:

```
$ podman logout quay.io
Removed login credentials for quay.io
```

Ниже приводятся некоторые часто используемые параметры команд `podman login` и `logout`:

- `--username,(-u)`. Позволяет указать имя пользователя Podman при входе в реестр.
- `--authfile`. Дает Podman указание хранить файл авторизации в другом месте. Переменная окружения `REGISTRY_AUTH_FILE` также используется для изменения местоположения файла авторизации.
- `--all`. Позволяет выйти из всех реестров.

Для получения информации обо всех параметрах используйте команды `man podman-login` и `man podman-logout`.

Обратите внимание: когда вы размещали образ в реестре контейнеров, вы переименовали `myimage` в quay.io/rhatdan/myimage:

```
$ podman push myimage quay.io/rhatdan/myimage
```

Неплохо было бы иметь локальный образ с именем quay.io/rhatdan/myimage — в этом случае вы бы просто выполнили:

```
$ podman push quay.io/rhatdan/myimage
```

В следующем разделе вы узнаете, как добавлять имена к образам.

2.2.6. Тегирование образов

Ранее в этой главе я указывал, что локально созданные образы связаны с реестром localhost. Образы создаются с реестром localhost, когда вы фиксируете контейнер в образ или используете `podman build` для создания образа. В Podman есть механизм для добавления дополнительных имен к образам. Такие имена называются тегами, поэтому для их добавления применяется команда `podman tag`.

Используя команду `podman images`, выведите образ или список образов в container/storage:

```
$ podman images
REPOSITORY                  TAG         IMAGE ID          CREATED        SIZE
localhost/myimage           latest    2c7e43d88038   46 \hours ago     462 MB
registry.access.redhat
➥.com/ubi8/httpd-24        latest    8594be0a0b57      5 weeks ago    462 MB
registry.access.redhat
➥.com/ubi8                 latest    ad42391b9b46      5 weeks ago    234 MB
```

Вам нужно, чтобы конечный образ, который вы планируете отправить, назывался quay.io/rhatdan/myimage. Чтобы добиться этого, добавьте это имя с помощью следующей команды `podman tag`:

```
$ podman tag myimage quay.io/rhatdan/myimage
```

Теперь снова запустите `podman images`, чтобы проверить образы. Вы увидите, что имя теперь quay.io/rhatdan/myimage. Обратите внимание, что localhost/myimage и quay.io/ rhatdan/myimage имеют одинаковый идентификатор образа 2c7e43d88038:

```
$ podman images
REPOSITORY                     TAG      IMAGE ID        CREATED      SIZE
localhost/myimage           latest  2c7e43d88038   46 hours ago    462 MB
quay.io/rhatdan/myimage     latest  2c7e43d88038   46 hours ago    462 MB
registry.access.redhat
➥.com/ubi8/httpd-24        latest  8594be0a0b57    5 weeks ago    462 MB
registry.access.redhat
➥.com/ubi8                 latest  ad42391b9b46    5 weeks ago    234 MB
```

Поскольку образы имеют одинаковый ID, это один и тот же образ с несколькими именами. Теперь вы можете напрямую взаимодействовать с quay.io/rhatdan/myimage. Сначала вам нужно снова войти в quay.io:

```
$ podman login --username rhatdan quay.io
Password:
Login Succeeded!
```

Сделайте push без необходимости указания имени места назначения (транспорта):

```
$ podman push quay.io/rhatdan/myimage
Getting image source signatures
...
Storing signatures
```

Получилось гораздо проще.

Давайте протегируем ранее использованный образ версией 1.0:

```
$ podman tag quay.io/rhatdan/myimage quay.io/rhatdan/myimage:1.0
```

Еще раз исследуйте образы. Обратите внимание, что у `myimage` теперь три разных имени/тега. Все три имеют одинаковый идентификатор 2c7e43d88038:

```
$ podman images
REPOSITORY                    TAG         IMAGE ID       CREATED       SIZE
localhost/myimage          latest      2c7e43d88038   46 hours ago     462 MB
quay.io/rhatdan/myimage       1.0      2c7e43d88038   46 hours ago     462 MB
quay.io/rhatdan/myimage    latest      2c7e43d88038   46 hours ago     462 MB
registry.access.redhat
➥ .com/ubi8/httpd-24         latest    8594be0a0b57   5 weeks ago      462 MB
registry.access.redhat
➥ .com/ubi8                  latest    ad42391b9b46   5 weeks ago      234 MB
```

Теперь отправим версию 1.0 `myimage` (приложения) в реестр:

```
$ podman push quay.io/rhatdan/myimage:1.0
Getting image source signatures
Copying blob 8f26704f753c skipped: already exists
Copying blob e39c3abf0df9 skipped: already exists
Copying blob 654b3bf1361e skipped: already exists
Copying blob 83310c7c677c skipped: already exists
Copying blob 164d51196137 [--------------------------------] 0.0b / 0.0b
Copying config 2c7e43d880 [--------------------------------] 0.0b / 4.0KiB
Writing manifest to image destination
Storing signatures
```

Пользователи будут извлекать либо образ latest, либо версию 1.0. Позже, когда вы создадите версию 2.0 своего приложения, вы сможете хранить оба образа в реестре и одновременно запускать на хосте версии 1.0 и 2.0 вашего приложения.

ПРИМЕЧАНИЕ Вопреки здравому смыслу, тег `latest` (англ. «самый последний») не относится к самому последнему образу в хранилище. Это просто еще один тег без каких-либо магических свойств. Хуже того, поскольку он используется в качестве тега по умолчанию для образов, загружаемых без указания тега, он может ссылаться на любую случайную версию образа. В реестре контейнеров могут быть более новые образы, чем в вашем локальном хранилище с этим тегом. Поэтому всегда лучше указывать конкретную версию образа, которую вы хотите использовать, а не полагаться на `latest`.

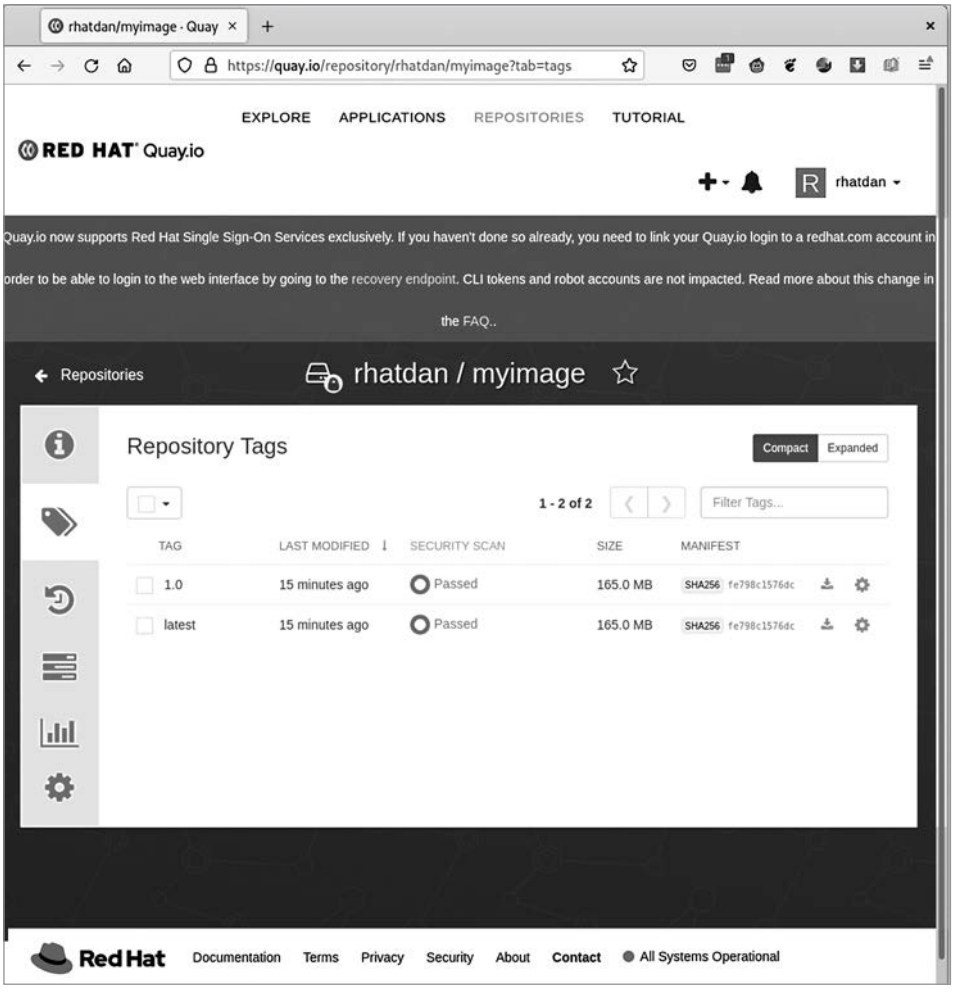


Рис. 2.4. Список тегов myimage на quay.io (https://quay.io/repository/rhatdan/myimage/?tab=tags)

Используйте веб-браузер (Firefox, Chrome, Safari, Internet Explorer или Microsoft Edge) для просмотра образов на сайте quay.io. На рис. 2.4 представлены образы 1.0 и latest.

```
$ web-browser quay.io/repository/rhatdan/myimage?tab=tags
```

Образ помещен в реестр контейнеров, и теперь следует освободить место в домашнем каталоге, удалив оттуда образы.

2.2.7. Удаление образов

Со временем образы начинают занимать много места на диске. Поэтому удалять те, которые вы больше не используете, — правильная идея. Давайте сначала выведем список локальных образов:

```
$ podman images
REPOSITORY                    TAG         IMAGE ID       CREATED      SIZE
localhost/myimage             1.0     2c7e43d88038   46 hours ago    462 MB
quay.io/rhatdan/myimage       1.0     2c7e43d88038   46 hours ago    462 MB
quay.io/rhatdan/myimage    latest     2c7e43d88038   46 hours ago    462 MB
registry.access.redhat
➥.com/ubi8/httpd-24       latest     8594be0a0b57   5 weeks ago     462 MB
registry.access.redhat
➥.com/ubi8                latest     ad42391b9b46   5 weeks ago     234 MB
```

Используйте команду `podman rmi` для удаления локальных образов:

```
$ podman rmi localhost/myimage
Untagged: localhost/myimage:latest
```

При повторном выводе списка локальных образов вы увидите, что команда на самом деле удалила не образ, а только тег `localhost` из этого образа. У Podman по-прежнему две ссылки на один и тот же ID образа: фактическое содержимое образа не было удалено. Дисковое пространство не освободилось:

```
$ podman images
REPOSITORY                    TAG         IMAGE ID       CREATED      SIZE
quay.io/rhatdan/myimage       1.0     2c7e43d88038   46 hours ago    462 MB
quay.io/rhatdan/myimage    latest     2c7e43d88038   46 hours ago    462 MB
registry.access.redhat
➥.com/ubi8/httpd-24       latest     8594be0a0b57    5 weeks ago    462 MB
registry.access.redhat
➥.com/ubi8                latest     ad42391b9b46    5 weeks ago    234 MB
```

Остальные теги можно удалить с помощью короткого имени (раздел 2.2.8). Podman использует короткое имя, находит первое имя в локальном хранилище, соответствующее короткому имени без реестра, и удаляет его. Вот почему нужно удалить его дважды, чтобы избавиться от обоих образов. Теги, отличные от `latest`, должны быть указаны явно:

```
$ podman rmi myimage
Untagged: quay.io/rhatdan/myimage:latest
$ podman rmi myimage:1.0
Untagged: quay.io/rhatdan/myimage:1.0
Deleted: 2c7e43d88038669e8cdbdff324a9f9605d99697215a0d21c360fe8dfa8471bab
```

Только после удаления последнего тега освобождается реальное дисковое пространство:

```
$ podman images
REPOSITORY                  TAG         IMAGE ID       CREATED     SIZE
registry.access.redhat
➥.com/ubi8/httpd-24     latest   8594be0a0b57    5 weeks ago   462 MB
registry.access.redhat
➥.com/ubi8              latest   ad42391b9b46    5 weeks ago   234 MB
```

Другой вариант — попробовать удалить образы, указав ID образа:

```
$ podman rmi 14119a10abf4
Error: unable to delete image\
➥ "2c7e43d88038669e8cdbdff324a9f9605d99697215a0d21c360fe8dfa8471bab" by\
➥ ID with more than one tag ([quay.io/rhatdan/myimage:1.0\
➥ quay.io/rhatdan/myimage:latest]): please force removal
```

Но это не удается, поскольку для одного и того же образа существует несколько тегов. Добавление параметра `--force` удаляет образ и все его теги:

```
$ podman rmi 14119a10abf4 --force
Untagged: quay.io/rhatdan/myimage:1.0
Untagged: quay.io/rhatdan/myimage:latest
Deleted: 2c7e43d88038669e8cdbdff324a9f9605d99697215a0d21c360fe8dfa8471bab
```

По мере увеличения размеров и количества образов, а также растущего числа контейнеров становится все труднее определить, какие образы больше не нужны. В Podman есть еще одна полезная команда для удаления всех висячих (dangling) образов — `podman image prune`. *Висячие образы* — это образы, которые больше не имеют связанного с ними тега и не используются каким-либо контейнером. У команды `prune` также имеется параметр `--all`, удаляющий все образы, не используемые в настоящее время никакими контейнерами, включая висячие:

```
$ podman image prune -a
WARNING! This command removes all images without at least one container \
➥ associated with them.
Are you sure you want to continue? [y/N] y
6d633c2626113fb4e5aa75babb2af39268948497893f7bb5b4c2043d7a986ba0
B9097177b416944cabdcfcab0e74a319223ad1acaed38ac57a262b2421732355
```

ПРИМЕЧАНИЕ При отсутствии запущенных контейнеров команда `podman image prune` удаляет все локальные образы. Это освобождает все дисковое пространство в домашнем каталоге. С помощью команды `podman system df` можно показать все дисковое пространство в вашем домашнем каталоге, используемое Podman.

```
$ podman images
REPOSITORY          TAG      IMAGE       ID        CREATED        SIZE
```

Ниже приводятся некоторые часто используемые параметры команды `podman image prune`:

- `--all`. Дает указание Podman удалить все образы, чтобы освободить дисковое пространство. Образы, на которых в данный момент запущены контейнеры, не удаляются.
- `--force`. Дает указание Podman остановить и удалить все запущенные на образе контейнеры и удалить все образы, зависящие от того образа, который вы пытаетесь удалить.

Для получения информации обо всех параметрах используйте команду `man podman-image-prune`.

Образы, помещенные в реестр, также могут быть извлечены для разных целей, например обмена приложениями с другими пользователями, тестирования различных версий, восстановления удаленных локальных версий, работы над новой версией образа и т. д.

2.2.8. Загрузка образов

Хотя вы уже удалили все локальные образы, можно извлечь образ, ранее размещенный по адресу quay.io/rhatdan/myimage. В Podman есть команда `podman pull` для загрузки («вытягивания») образов из реестров контейнеров (транспортов) в локальное хранилище контейнеров:

```
$ podman pull quay.io/rhatdan/myimage
Trying to pull quay.io/rhatdan/myimage:latest...
Getting image source signatures
Copying blob dfd8c625d022 done
Copying blob e21480a19686 done
Copying blob 68e8857e6dcb done
Copying blob 3f412c5136dd done
Copying blob fbfcc23454c6 done
Copying config 2c7e43d880 done
Writing manifest to image destination
Storing signatures
2c7e43d880382561ebae3fa06c7a1442d0da2912786d09ea9baaef87f73c29ae
```

Вывод выглядит знакомым? Вероятно, вы помните похожий вывод команды `podman run` из раздела 2.1.2:

```
$ podman run -d -p 8080:8080 --name myapp\
➥ registry.access.redhat.com/ubi8/httpd-24
Trying to pull registry.access.redhat.com/ubi8/httpd-24:latest...
Getting image source signatures
Checking if image destination supports signatures
```

```
Copying blob 296e14ee2414 skipped: already exists
Copying blob 356f18f3a935 skipped: already exists
Copying blob 359fed170a21 done
Copying blob 226cafc3a0c6 done
Writing manifest to image destination
Storing signatures
37a1d2e31dbf4fa311a5ca6453f53106eaae2d8b9b9da264015cc3f8864fac22
```

Многие команды Podman автоматически выполняют `podman pull`, если требуемый образ не присутствует в локальном хранилище.

Итак, выполнение команды `podman images` показывает, что образ снова в контейнерном хранилище и готов к использованию контейнерами:

```
$ podman images
REPOSITORY TAG IMAGE ID CREATED SIZE
quay.io/rhatdan/myimage latest 2c7e43d88038 2 days ago 462 MB
```

До сих пор вы вводили образы с полными именами registry.access.redhat.com/ubi8/httpd-24 или quay.io/rhatdan/myimage, но если вы, как и я, медленно набираете текст, это мучительно. Вам необходим способ обращения к образам через короткие имена.

Когда Docker только появился, ссылка на образ в нем была определена как комбинация из реестра контейнеров, где хранится образ, а также репозитория, имени образа и тега или версии образа. В табл. 2.4 показана эта разбивка имени образа; обратите внимание, что тег `latest` применен имплицитно, так как версия образа не задана.

Таблица 2.4. Таблица имени образа

Реестр	Репозиторий	Имя	Тег
quay.io	rhatdan	myimage	`latest`

В командной строке Docker реестр docker.io установлен как единственный, что заставляет каждое короткое имя образа ссылаться на образы на docker.io. Существует также специальный репозиторий library, который используется для сертифицированных образов.

Поэтому вместо того, чтобы набирать:

```
# docker pull docker.io/library/alpine:latest
```

вы можете просто выполнить:

```
# docker pull alpine
```

И наоборот, если вы хотите извлечь образ из другого реестра, вам нужно указать полное имя образа:

```
# docker pull registry.access.redhat.com/ubi8/httpd-24:latest
```

В табл. 2.5 показана разница между именем образа в сокращенном виде и полным именем образа. Обратите внимание, что при использовании короткого имени реестр, хранилище и тег не были указаны.

Таблица 2.5. Таблица соответствия короткого имени образа полному

Реестр	Репозиторий	Имя	Тег
		alpine	
docker.io	library	alpine	`latest`

Поскольку я ленив и не люблю набирать лишние символы, я почти всегда использую короткие имена. В Podman разработчики не стали встраивать в инструмент только один реестр, docker.io. Podman позволяет дистрибутивам, компаниям и вам самим контролировать, какие реестры использовать, а также настраивать несколько реестров. В то же время Podman обеспечивает поддержку более простых в использовании коротких имен.

Podman обычно поставляется с несколькими заданными реестрами, которые контролируются дистрибутивом, упаковавшим Podman. Чтобы узнать, какие реестры определены для вашей инсталляции Podman, используйте команду `podman info`:

```
$ podman info
...
registries:
  search:
  - registry.fedoraproject.org
  - registry.access.redhat.com
  - docker.io
  - quay.io
```

Список реестров изменяется в файле registries.conf, который описан в разделе 5.2.1.

Давайте теперь обсудим аспекты безопасности на примере следующих команд:

```
$ podman pull rhatdan/myimage
$ podman pull quay.io/rhatdan/myimage
```

С точки зрения безопасности всегда лучше указывать полное имя образа при извлечении его из реестра. При таком способе Podman гарантирует, что он

загружает образ из указанного реестра. Представьте, что вы пытаетесь получить образ rhatdan/myimage. При использовании предыдущего порядка поиска есть шанс, что кто-то создаст учетную запись на docker.io/rhatdan и эта уловка заставит вас по ошибке получить docker.io/rhatdan/myimage.

Для защиты от подобных ситуаций при первой загрузке образа Podman предлагает вам выбрать точный образ из списка найденных в настроенных реестрах:

```
$ podman create -p 8080:8080 ubi8/httpd-24
? Please select an image:
   registry.fedoraproject.org/ubi8/httpd-24:latest
 ▸ registry.access.redhat.com/ubi8/httpd-24:latest
   docker.io/ubi8/httpd-24:latest
   quay.io/ubi8/httpd-24:latest
```

После того как вы однажды успешно выбрали и загрузили образ, Podman записывает сопоставление с ним короткого имени. В будущем, когда вы запустите контейнер с таким коротким именем, Podman использует это сопоставление для выбора правильного реестра без выдачи лишних подсказок.

Дистрибутивы Linux также содержат сопоставления часто используемых коротких имен, поскольку это способствует тому, чтобы вы брали образы из поддерживаемых ими реестров. Вы найдете эти файлы конфигурации в каталоге /etc/containers/registries.conf.d на хосте Linux. Компании тоже могут поместить файлы псевдонимов коротких имен в этот каталог:

```
$ cat /etc/containers/registries.conf.d/000-shortnames.conf
[aliases]
# centos
"centos" = "quay.io/centos/centos"
# containers
"skopeo" = "quay.io/skopeo/stable"
"buildah" = "quay.io/buildah/stable"
"podman" = "quay.io/podman/stable"
...
```

Ниже приводятся некоторые часто используемые параметры команды `podman pull`:

- `--arch`. Дает указание Podman, что нужно извлечь образ для другой архитектуры. Например, на моей машине x86_64 я могу получить образ arm64. По умолчанию `podman pull` загружает образы для своей собственной архитектуры.
- `--quiet (-q)`. Дает указание Podman не выводить всю информацию о ходе выполнения, а просто сообщить идентификатор образа, завершив работу.

Для получения информации обо всех параметрах используйте команду `man podman-pull`.

В этой книге я упомянул лишь несколько образов, но существуют тысячи и тысячи доступных образов. Необходим механизм, позволяющий искать среди них нужные.

2.2.9. Поиск образов

Не всегда вам известно имя конкретного образа, который вы собираетесь запустить или использовать в качестве основы для создания собственного образа. Podman предоставляет команду `podman search`, позволяющую искать в реестрах контейнеров подходящие имена:

```
$ podman search registry.access.redhat.com/httpd
INDEX NAME
➥ DESCRIPTION redhat.com
➥ registry.access.redhat.com/rhscl/httpd-24-rhel7
➥ Apache HTTP 2.4\ Server
redhat.com registry.access.redhat.com/ubi8/httpd-24\
➥ Platform for running Apache httpd 2.4 or bui...
redhat.com registry.access.redhat.com/rhscl/varnish-6-rhel7 Varnish\
➥ available as container is a base pla...
...
```

В этом примере в репозитории registry.access.redhat.com мы ищем образы, содержащие в своем имени строку *httpd*.

Ниже приводятся некоторые часто используемые параметры команды `podman search`:

- `--no-trunc` — дает указание Podman показывать полное описание образа
- `--format` — позволяет вам настроить, какие поля будут отображаться.

Для получения информации обо всех параметрах используйте команду `man podman-search`.

В предыдущих разделах вы узнали несколько способов управления контейнерами и работы с образами контейнеров, включая их инспектирование, передачу, загрузку и поиск. Но ознакомиться с содержимым образа вы могли, только запустив его в качестве контейнера. Одним из способов упростить этот процесс является монтирование образа контейнера.

2.2.10. Монтирование образов

Время от времени возникает необходимость изучить содержимое образа. Одним из способов это сделать является запуск оболочки внутри контейнера,

работающего на основе этого образа. Проблема в том, что инструменты, которые вы используете для изучения образа контейнера, могут быть недоступны внутри контейнера. Возникает также угроза безопасности: если приложение в контейнере окажется вредоносным, использование данного контейнера нежелательно.

Для решения этих проблем в Podman имеется команда `podman image mount`, которая монтирует корневую файловую систему образа в режиме только для чтения без создания контейнера. Смонтированный образ сразу же становится доступным на системе хоста, что позволяет изучить его содержимое.

Попробуйте смонтировать образ, который вы загрузили ранее:

```
$ podman mount quay.io/rhatdan/myimage
Error: cannot run command "podman mount" in rootless mode, must execute
       'podman unshare' first
```

Причина этой ошибки в том, что режим rootless не позволяет монтировать образы. Вам необходимо ввести пользовательское пространство имен и отдельное пространство имен mount. В главе 5 объясняется, как большинство команд Podman в режиме rootless при выполнении переходят в пространство имен пользователя и пространство имен mount. Пока же достаточно знать, что команда `podman unshare` входит в пользовательское пространство имен и пространство имен mount и завершает работу, когда вы выполняете команду `exit` вашей оболочки.

> **ПРИМЕЧАНИЕ** Название `unshare` происходит от системного вызова Linux `unshare (man 2 unshare)`. Linux также включает инструмент `unshare (man 1 unshare)`, позволяющий создавать пространства имен вручную. Другой низкоуровневый инструмент, называемый `nsenter` или `namespace enter (man 1 nsenter)`, позволяет присоединять процессы к различным пространствам имен. `unshare` в Podman использует те же возможности ядра. Он упрощает процесс создания и настройки пространств имен и присоединения процессов к пространствам имен.

Команда `podman unshare` переводит вас в командную строку `#`, где вы можете фактически смонтировать образ:

```
$ podman unshare
#
```

Смонтируйте образ и сохраните расположение смонтированной файловой системы в переменной окружения:

```
# mnt=$(podman image mount quay.io/rhatdan/myimage)
```

Теперь можно изучить содержимое образа. Давайте выведем содержимое файла в терминал:

```
# cat $mnt/var/www/html/index.html
<html>
  <head>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Когда вы закончите, размонтируйте образ и выйдите из сеанса unshare:

```
# podman image unmount quay.io/rhatdan/myimage
# exit
```

> **ПРИМЕЧАНИЕ** Мы рассмотрели примерно половину подкоманд команды `podman image` — пожалуй, наиболее часто используемых. Полная информация об этих и других подкомандах `podman image` приводится на страницах руководства Podman `man podman-image`.

Теперь, когда вы получили больше знаний о контейнерах и образах, следующим важным шагом будет обновление вашего образа. Это связано с тем, что приложение необходимо обновлять, а также с появлением новых версий для используемого базового образа. Можно создать скрипты для ручного выполнения команд для сборки образа, но к счастью, Podman оптимизировал этот процесс.

2.3. СБОРКА ОБРАЗОВ

До сих пор вы работали с образами, которые уже были созданы и загружены в реестр контейнеров. Процесс создания образа называется *сборкой* (building).

При сборке образов контейнеров вы управляете не только своим приложением, но и содержимым образа, используемого этим приложением. Во времена до появления контейнеров приложения поставлялись в виде пакетов RPM или DEB, а затем дистрибутив должен был следить за тем, чтобы другие части ОС поддерживались в актуальном и безопасном состоянии. Но в мире контейнеров образ включает в себя приложение вместе с частью ОС. Ответственность за поддержание актуальности и безопасности всего содержимого образа лежит на разработчиках.

Мой коллега Скотт Маккарти (Scott McCarty, smccarty@redhat.com, @fatherlinux) как-то сказал: «Образы контейнеров стареют не как вино, а скорее как сыр. Когда образ стареет, он начинает вонять».

Это означает, что если разработчик не следит за обновлениями безопасности, количество уязвимостей в образе будет расти с угрожающей скоростью. К счастью для разработчиков, Podman имеет специальный механизм, который помогает в создании образов для ваших приложений. Команда `podman build` использует инструмент Buildah (https://github.com/containers/buildah) в качестве библиотеки для создания образов контейнеров. Buildah рассматривается в приложении А.

`podman build` использует специальный текстовый документ под названием Containerfile или Dockerfile, чтобы автоматизировать создание образов контейнеров. В этом документе перечислены команды для сборки образа контейнера.

ПРИМЕЧАНИЕ Концепция Dockerfile и его синтаксис были первоначально созданы для инструмента Docker, разработанного компанией Docker. В Podman имя файла — Containerfile, а синтаксис в нем точно такой же. Имя Dockerfile также поддерживается в целях обратной совместимости. Команда сборки Docker по умолчанию не поддерживает Containerfile, но использовать может. Для этого следует указать параметр `-f`: `# docker build -f Containerfile`.

2.3.1. Формат Containerfile или Dockerfile

В файлах Containerfile имеется множество директив. Я разделяю их на две категории: одна связана с добавлением содержимого к образу контейнера, а другая относится к описанию и документированию того, как использовать образ.

Добавление содержимого к образу

Напомню, в разделе 1.1.2 я описал образ контейнера как каталог на диске, который выглядит как корень в системе Linux. Этот каталог называется rootfs. Некоторые директивы в процессе выполнения сборки контейнера добавляют содержимое в rootfs. Этот rootfs в конечном итоге включает в себя все содержимое, используемое для создания образа контейнера.

Каждый Containerfile должен содержать строку `FROM`. В строке `FROM` указывается образ, на котором будет основан новый образ, его часто называют базовым. Команда `podman build` поддерживает специальный образ под названием `scratch`, что означает запуск образа без содержимого. Когда Podman видит директиву `FROM scratch`, он просто выделяет место в containers/storage для пустого rootfs, а затем директива `COPY` используется для наполнения этого rootfs. Чаще всего директива `FROM` применяется к существующему образу. Например, `FROM registry.access.redhat.com/ubi8` дает указание Podman загрузить образ ubi8 из реестра контейнеров registry.access.redhat.com и скопировать его в контейнерное хранилище. `podman build` загружает тот же образ, что и команда `podman pull`, о которой вы узнали в разделе 2.2.8. Когда образ загружен, Podman использует контейнерное

хранилище для монтирования этого образа в каталог rootfs, используя файловую систему «Copy-on-write», например OverlayFS, куда другие директивы способны добавлять содержимое. Этот образ становится базовым слоем rootfs.

Директива `COPY` используется для копирования файлов, каталогов или tar-архивов с локального хоста во вновь созданный rootfs. Директива `RUN` — одна из часто используемых в Containerfile. `RUN` дает указание Podman фактически запустить контейнер на этом образе. Инструменты управления пакетами, такие как DNF/YUM и `apt-get`, запускаются для установки пакетов из дистрибутивов на ваш новый образ. Директива `RUN` запускает любую команду внутри образа как контейнер. Команда `podman build` запускает команды с теми же ограничениями безопасности, что и команда `podman run`.

Предположим, например, что вам нужно добавить команду `ps` в образ контейнера. Для этого создайте директиву, подобную описанной ниже. Команда `RUN` запускает контейнер, который обновляет все пакеты из базового образа, а затем устанавливает пакет `procps-ns`, включающий команду `ps`. Наконец, запущенная в контейнере команда выполняет `yum`, чтобы убрать за собой, в результате чего из образа контейнера удаляется мусор:

```
RUN yum -y update; yum -y install procps-ng; yum -y clean all
```

Добавление содержимого в образ контейнера — это лишь половина того, что вам предстоит сделать при создании образа. Вам также понадобится описать и задокументировать, как должен использоваться ваш образ, когда другие пользователи станут его загружать и запускать.

Документация — как использовать образ

В разделе 1.1.2 я привел JSON-файл, содержащий спецификацию образа. Эта спецификация описывает, как образ контейнера должен быть запущен, под каким UID его запускать, какая команда будет выполняться по умолчанию и другие требования. Containerfile также поддерживает множество директив, которые указывают Podman, как запускать контейнеры. К ним относятся следующие:

- Директивы `ENTRYPOINT` и `CMD`. Позволяют установить в образе команду по умолчанию, которая будет выполняться, когда пользователи запускают образ с помощью podman `run`. `CMD` — это сама команда для выполнения. `ENTRYPOINT` позволяет запустить весь образ как одну команду.
- Директива `ENV`. Устанавливает переменные окружения по умолчанию, когда Podman запускает контейнер на основе образа.
- Директива `EXPOSE`. Записывает сетевые порты, которые Podman должен открыть в контейнерах на основе этого образа. Если вы выполните `podman run`

`--publish-all`..., Podman ищет внутри образа сетевые порты `EXPOSE` и подключает их к хосту.

В табл. 2.6 описаны директивы, используемые в Containerfile для добавления содержимого в образ контейнера.

Таблица 2.6. Директивы Containerfile, вносящие изменения в образ

Примеры директив	Пояснение
`FROM quay.io/rhatdan/myimage`	Устанавливает базовый образ для последующих инструкций. Containerfile должны иметь FROM в качестве первой инструкции. FROM может появляться несколько раз в одном файле Containerfile для создания нескольких этапов сборки
`ADD start.sh /usr/bin/start.sh`	Копирует новые файлы, каталоги или удаленные файлы по URL-адресу в файловую систему контейнера по указанному пути
`COPY start.sh /usr/bin/start.sh`	Копирует файлы в файловую систему контейнера по указанному пути
`RUN dnf -y update`	Выполняет команды в новом слое поверх текущего образа и фиксирует результаты. Зафиксированный образ используется для следующего шага в Containerfile
`VOLUME /var/lib/mydata`	Создает точку монтирования с указанным именем и помечает ее как содержащую тома, смонтированные извне с собственного хоста или из других контейнеров. Подробнее о томах рассказывается в главе 3

Таблица 2.7 описывает директивы, используемые в Containerfile для заполнения спецификаций OCI Runtime Specifications данными, которые сообщают таким контейнерным движкам, как Podman, информацию об образе и о том, как запустить этот образ. Более подробную информацию о Containerfile можно найти на странице man `containerfile(5)`.

Таблица 2.7. Директивы Containerfile, определяющие спецификацию OCI Runtime Specification

Примеры директив	Пояснение
`CMD /usr/bin/start.sh`	Указывает команду по умолчанию, которая будет выполняться при запуске контейнера из этого образа. Если CMD не указан, наследуется CMD родительского образа. Обратите внимание, что RUN и CMD сильно отличаются друг от друга. RUN выполняет команды в процессе сборки, а CMD используется только тогда, когда пользователь запускает образ без указания команды

Таблица 2.7 (*окончание*)

Примеры директив	Пояснение
`ENTRYPOINT "/bin/sh -c"`	Позволяет настроить контейнер на запуск в виде исполняемого файла. Инструкция `ENTRYPOINT` не перезаписывается при передаче аргументов в `podman run`. Это позволяет передавать аргументы в точку входа, например, `podman run <image> -d` передает аргумент `-d` в `ENTRYPOINT`
`ENV foo="bar"`	Добавляет переменную окружения, которая будет использоваться как при сборке образа, так и при выполнении контейнера
`EXPOSE 8080`	Объявляет порт, который будут использовать контейнерные приложения. На самом деле эта директива не сопоставляет и не открывает какие-либо порты
`LABEL Description="Web browser which displays Hello World"`	Добавляет метаданные к образу
`MAINTAINER Daniel Walsh`	Устанавливает поле `Author` для создаваемых образов
`STOPSIGNAL SIGTERM`	Устанавливает сигнал остановки по умолчанию, посылаемый контейнеру для завершения работы. Сигналом может быть число без знака или имя сигнала в формате `SIGNAME`
`USER apache`	Устанавливает имя пользователя (или UID) и имя группы (или GID), которые будут использоваться для всех `RUN`, `CMD` и `ENTRYPOINT`, указанных после него
`ONBUILD`	Добавляет в образ инструкцию запуска, выполняемую позже, когда образ будет использоваться в качестве базового для другой сборки
`WORKDIR /var/www/html`	Устанавливает рабочий каталог для директив `RUN`, `CMD`, `ENTRYPOINT` и `COPY`. Каталог будет создан, если он не существует

Сохранение образа

Когда `podman build` завершает обработку Containerfile, он фиксирует образ, используя тот же код, что и `podman commit`, о котором вы узнали в разделе 2.1.9. По сути, Podman собирает в TAR все различия между новым содержимым в rootfs и базовым образом, извлеченным директивой `FROM`. Podman также фиксирует JSON-файл и сохраняет его как образ в контейнерном хранилище. Теперь вы

можете автоматизировать все шаги, используемые для создания контейнерных приложений, с помощью Containerfile и Podman build.

> **СОВЕТ** Используйте параметр `--tag` для присвоения имени новому образу, который вы создаете с помощью `podman build`. Это заставит Podman добавить заданный тег или имя к образу в контейнерном хранилище так же, как это делает команда `podman tag`.

2.3.2. Автоматизация сборки нашего приложения

Сначала создайте каталог, в который будут помещены ваш Containerfile и любое другое содержимое для образа контейнера. Этот каталог называется контекстным:

```
mkdir myapp
```

Затем создайте файл index.html, который вы планируете использовать в контейнере, в каталоге `myapp`:

```
$ cat > myapp/index.html << _EOF
<html>
  <head>
  </head>
  <body>
  <h1>Hello World</h1>
  </body>
</html>
_EOF
```

Далее создайте простой Containerfile для сборки вашего приложения в каталоге `myapp`. Первая строка Containerfile — это директива `FROM` для извлечения образа ubi8/httpd-24, который вы будете использовать в качестве базового. Затем добавьте команду `COPY` для копирования файла index.html в образ. Директива `COPY` указывает Podman взять файл index.html из контекстного каталога (./myapp) и скопировать его в файл /var/www/html/index.html внутри образа:

```
$ cat > myapp/Containerfile << _EOF
FROM ubi8/httpd-24
COPY index.html /var/www/html/index.html
_EOF
```

Наконец, используйте `podman build` для сборки вашего контейнерного приложения. Укажите `--tag (-t)`, чтобы назвать образ quay.io/rhatdan/myimage. Также необходимо указать контекстный каталог ./myapp:

```
$ podman build -t quay.io/rhatdan/myimage ./myapp
STEP 1/2: FROM ubi8/httpd-24
```

```
STEP 2/2: COPY index.html /var/www/html/index.html
COMMIT quay.io/rhatdan/myimage
--> f81b8ace4f1
Successfully tagged quay.io/rhatdan/myimage:latest
F81b8ace4f134d08cedb20a9156ae727444ae4d4ec1ceb3b12d3aff23d18128b
```

Когда команда `podman build` завершает работу, она фиксирует образ и тегирует (`-t`) его именем quay.io/rhatdan/myimage. Теперь он готов к отправке в реестр контейнеров с помощью команды `podman push`.

Далее вы можете настроить CI/CD-систему или даже простое задание cron для регулярной сборки и замены `myapplication`:

```
$ cat > myapp/automate.sh << _EOF
#!/bin/bash
podman build -t quay.io/rhatdan/myimage ./myapp
podman push quay.io/rhatdan/myimage
_EOF
$ chmod +x myapp/automate.sh
```

Добавьте также несколько тестовых скриптов, чтобы убедиться, что ваше приложение работает так, как было задумано перед заменой предыдущей версии. Давайте посмотрим на созданные образы:

```
$ podman images
REPOSITORY                   TAG         IMAGE ID          CREATED       SIZE
quay.io/rhatdan/myimage    latest    f81b8ace4f13    2 minutes ago     462 MB
<none>                     <none>    2c7e43d88038       2 days ago     462 MB
registry.access.redhat
➥ .com/ubi8/httpd-24      latest    8594be0a0b57      5 weeks ago     462 MB
```

Обратите внимание, что старая версия quay.io/rhatdan/myimage с идентификатором `2c7e43d88038` все еще существует в контейнерном хранилище, но теперь имеет `REPOSITORY` и `TAG` `<none> <none>`. Такие образы называются висячими. Поскольку я создал новую версию quay.io/rhatdan/myimage с помощью команды `podman build`, предыдущий образ утратил это имя. Вы все еще можете использовать команды Podman с ID-образа или, если новый образ не работает, просто примените `podman tag`, чтобы переименовать старый образ обратно в quay.io/rhatdan/myimage. Если новый образ работает правильно, следует удалить старый образ с помощью `podman rmi`. Эти `<none><none>`-образы имеют тенденцию накапливаться со временем, расходуя место, но вы можете периодически использовать команду `podman image prune` для их удаления.

О `podman build` следовало было бы написать отдельную главу или даже книгу. Люди создают образы тысячами различных способов, используя команды, кратко описанные ниже.

`--tag` — важный параметр `podman build`, который задает тег или имя образа. Помните, что у вас всегда есть возможность добавить дополнительные имена после создания образа с помощью команды `podman tag`, которую вы использовали в разделе 2.2.6. Для получения информации обо всех параметрах используйте команду `man podman-build` (табл. 2.8).

Таблица 2.8. Команды podman image

Команда	man-страница	Описание
`import`	`podman-image-import(1)`	Импортирует tar-архив для создания образа файловой системы
`inspect`	`podman-image-inspect(1)`	Отображает конфигурацию образа
`list`	`podman-image-list(1)`	Выводит список всех образов
`load`	`podman-image-load(1)`	Загружает образ(ы) из tar-архива
`mount`	`podman-image-mount(1)`	Монтирует корневую файловую систему образа
`prune`	`podman-image-prune(1)`	Удаляет неиспользуемые образы
`pull`	`podman-image-pull(1)`	Загружает образ из реестра
`push`	`podman-image-push(1)`	Отправляет образ в реестр
`rm`	`podman-image-rm(1)`	Удаляет образ
`save`	`podman-image-save(1)`	Сохраняет образ(ы) в архив
`scp`	`podman-image-scp(1)`	Осуществляет безопасное копирование образов в другие containers/storage
`search`	`podman-image-search(1)`	Осуществляет поиск образа в реестре
`sign`	`podman-image-sign(1)`	Подписывает образ
`tag`	`podman-image-tag(1)`	Добавляет дополнительное имя к локальному образу
`tree`	`podman-image-tree(1)`	Выводит иерархию слоев образа в формате дерева
`trust`	`podman-image-trust(1)`	Управляет политикой доверия к реестрам образов контейнеров
`unmount`	`podman-image-unmount(1)`	Размонтировывает корневую файловую систему образа
`untag`	`podman-image-untag(1)`	Удаляет имя локального образа

РЕЗЮМЕ

- Простой интерфейс командной строки Podman облегчает работу с контейнерами.
- Команды Podman `run`, `stop`, `start`, `ps`, `inspect`, `rm` и `commit` — это команды для работы с контейнерами.
- Команды Podman `pull`, `push`, `login` и `rmi` — это инструменты для работы с образами и обмена ими через реестры контейнеров.
- Podman `build` — это замечательная команда для автоматизации сборки контейнерных образов.
- Командная строка Podman основана на Docker CLI и полностью поддерживает ее, что позволяет нам говорить всем о Podman как о псевдониме Docker.
- В Podman есть дополнительные команды и параметры для поддержки более продвинутых концепций, например `podman image mount`.

3 Тома

В ЭТОЙ ГЛАВЕ

- ✓ Использование томов для изоляции данных от приложения в контейнере
- ✓ Передача содержимого с хоста в контейнеры через тома
- ✓ Использование томов с пользовательским пространством имен и SELinux
- ✓ Встраивание томов в образы контейнеров
- ✓ Изучение различных типов томов и команды volume

Контейнеры, с которыми вы работали до сих пор, включали все свое содержимое в образ контейнера. Как я писал в главе 1, единственное, что требуется использовать совместно традиционным контейнерам, — ядро Linux. Данные приложения необходимо изолировать от самого приложения по нескольким причинам, среди которых:

- Предотвращение включения фактических данных для таких приложений, как базы данных.

- Использование одного и того же образа контейнера для запуска нескольких окружений.
- Снижение накладных расходов и повышение производительности чтения/записи в хранилище. Тома записываются непосредственно в файловую систему, а контейнеры используют файловую систему overlay или fuse-overlayfs для монтирования своих слоев. Overlay — это многослойная файловая система, то есть ядру необходимо полностью скопировать предыдущий слой, чтобы создать новый, а fuse-overlayfs переключает каждое чтение и запись из пространства ядра в пространство пользователя и обратно. Все это создает значительные накладные расходы.
- Совместное использование содержимого, доступного через сетевое хранилище.

ПРИМЕЧАНИЕ `bind` перемонтировывает части файловой иерархии в другое место файловой системы. Файлы и каталоги в `bind mount` те же, что и в оригинале (`bind mount` объясняется на странице руководства команды `mount`). `bind mount` позволяет получить доступ к одному и тому же содержимому в двух местах без дополнительных накладных расходов. Важно понимать, что `bind` не копирует данные и не создает новые данные.

Поддержка томов также не облегчает ситуацию, особенно в том, что касается безопасности. Многие функции безопасности контейнеров не позволяют их процессам получить доступ к файловой системе за пределами образа контейнера. В этой главе вы узнаете, как Podman позволяет обойти эти ограничения.

3.1. ИСПОЛЬЗОВАНИЕ ТОМОВ В КОНТЕЙНЕРАХ

Вернемся к нашему контейнерному приложению. До сих пор вы просто встраивали веб-приложения в файловую систему контейнера напрямую. Напомню, что в разделе 2.1.8 команда `podman exec` использовалась для изменения данных Hello World index.html внутри контейнера:

```
$ podman exec -i myapp bash -c 'cat > /var/www/html/index.html' << _EOF
<html>
  <head>
  </head>
  <body>
  <h1>Hello World</h1>
  </body>
</html>
_EOF
```

Вы сделали образ контейнера более гибким, позволив пользователям предоставлять собственное содержимое для веб-службы или обновлять эту веб-службу на лету. Хотя такой метод возможен, в то же время он подвержен ошибкам и не масштабируется. Вот тут-то и пригодятся тома.

Podman позволяет монтировать содержимое файловой системы хоста в контейнеры с помощью команды `podman run` через параметр `--volume (-v)`.

Параметр `--volume HOST-DIR:CONTAINER-DIR` дает указание Podman связать `HOST-DIR` на хосте с `CONTAINER-DIR` в контейнере. Podman поддерживает и другие виды томов, но в этом разделе я сосредоточусь на томах `bind mount`.

Возможно монтирование и файлов, и каталогов с помощью одного параметра. Изменения содержимого на хосте будут видны внутри контейнера. Подобным же образом, если процессы контейнера изменяют содержимое внутри контейнера, изменения будут видны на хосте.

Рассмотрим пример. Создайте каталог html в своем домашнем каталоге, а затем создайте в нем новый файл html/index.html:

```
$ mkdir html
$ cat > html/index.html << _EOF
<html>
  <head>
  </head>
  <body>
  <h1>Goodbye World</h1>
  </body>
</html>
_EOF
```

Теперь запустите контейнер с параметром `-v ./html:/var/www/html`:

```
$ podman run -d -v ./html:/var/www/html:ro,z -p 8080:8080
     quay.io/rhatdan/myimage
94c21a3d8fda740857abc571469aaaa181f4db27a464ceb6743c4a37fb875772
```

Обратите внимание на дополнительные поля `:ro,z` в параметре `--volume`. Значение `ro` дает указание Podman смонтировать том в режиме только для чтения. Монтирование только для чтения означает, что процессы внутри контейнера не могут изменять содержимое /var/www/html, в то время как процессы на хосте все еще могут изменять его содержимое. По умолчанию Podman монтирует все тома в режиме чтения/записи. Значение `z` дает указание Podman перемаркировать содержимое общедоступной меткой для SELinux (раздел 3.1.2).

Запустив контейнер, откройте веб-браузер и перейдите на localhost:8080, чтобы убедиться, что изменения произошли (рис. 3.1).

```
$ web-browser localhost:8080
```

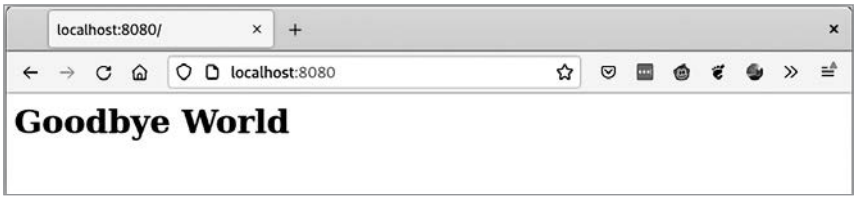


Рис. 3.1. Окно веб-браузера, подключающегося к контейнеру Podman myimage со смонтированным томом

Теперь вы можете остановить и удалить только что созданный контейнер. Удаление контейнера никак не влияет на содержимое. Следующая команда удаляет самый последний (`--latest`) контейнер — ваш. Параметр `--force` дает указание Podman остановить контейнер, а затем удалить его:

```
$ podman rm --latest --force
```

Наконец, удалите содержимое с помощью этой команды:

```
$ rm -rf html
```

> **ПРИМЕЧАНИЕ** Параметр `--latest` недоступен на платформах Mac и Windows. Вы должны указать имя или ID контейнера. Удаленный режим рассматривается в главе 9, а работа с Podman в Mac и Windows — в приложениях E и F.

3.1.1. Именованные тома

В предыдущем примере вы создали каталог на диске, а затем смонтировали его в контейнер. Подобным же образом вы можете взять любой существующий файл или каталог и смонтировать его в контейнер, если у вас есть к нему доступ на чтение.

Другим механизмом для хранения данных контейнеров Podman является именованный том. Создется он с помощью команды `podman volume create`. В следующем примере вы создадите том с именем `webdata`:

```
$ podman volume create webdata
webdata
```

По умолчанию Podman создает тома с локальными именами, а хранилище выделяется в каталогах хранилища контейнеров. Следующая команда позволяет исследовать том и найти его точку монтирования:

```
$ podman volume inspect webdata
[
  {
      "Name": "webdata",
      "Driver": "local",
      "Mountpoint":
➥ "/home/dwalsh/.local/share/containers/storage/volumes/webdata/_data",
      "CreatedAt": "2021-10-11T14:10:48.741367132-04:00",
      "Labels": {},
      "Scope": "local",
      "Options": {}
  }
]
```

Для хранения содержимого тома Podman фактически создает каталог в вашем локальном хранилище контейнеров, /home/dwalsh/.local/share/containers/storage/volumes/webdata/_data. Действуя с хоста, вы можете создавать содержимое в этом каталоге:

```
$ cat > /home/dwalsh/.local/share/containers/storage/volumes/webdata/_
   data/index.html << _EOL
<html>
  <head>
  </head>
  <body>
  <h1>Goodbye World</h1>
  </body>
</html>
_EOL
```

Теперь запустите приложение `myimage`, используя этот том:

```
$ podman run -d -v webdata:/var/www/html:ro,z -p 8080:8080
     quay.io/rhatdan/myimage
0c8eb612831f8fe22438d73d801e5bb664ec3b1d524c5c10759ee0049061cb6b
```

Обновите окно веб-браузера, чтобы убедиться, что файл, созданный в каталоге на хосте, отображает «Goodbye World» (рис. 3.2).

Рис. 3.2. Окно веб-браузера, подключающегося к контейнеру myimage Podman со смонтированным именованным томом

Именованные тома можно использовать для нескольких контейнеров одновременно, и они сохраняются даже после удаления контейнера. Если вы закончили работу с именованным томом и контейнером, остановите контейнер, не дожидаясь завершения процессов:

```
$ podman stop -t 0 0c8eb61283
```

Затем удалите том с помощью команды `podman volume rm`. Обратите внимание на параметр `--force`, дающий указание Podman удалить том и все контейнеры, которые используют этот том:

```
$ podman volume rm --force webdata
```

Убедитесь, что том удален, выполнив команду `volume list`:

```
$ podman volume list
```

Если именованный том не существовал до выполнения команды `podman run`, он будет создан автоматически. В следующем примере вы укажете `webdata1` в качестве имени именованного тома, а затем выведете список томов:

```
$ podman run -d -v webdata1:/var/www/html:ro,z -p 8080:8080\
➥ quay.io/rhatdan/myimage
58ccaf37958496322e34cd933cd4dd5a61ab06c5ba678beb28fdc29cfb81f407

$ podman volume list
DRIVER VOLUME NAME
local webdata1
```

Разумеется, этот том пуст. Удалите том `webdata1` и контейнер:

```
$ podman volume rm --force webdata1
```

Podman также поддерживает другие типы томов. В нем используется концепция плагинов томов, позволяющая сторонним разработчикам предоставлять свои типы томов. Более подробную информацию можно найти на странице руководства команды `podman-volume-create`.

В Podman есть и другие интересные возможности работы с томами. Команда `podman volume export` экспортирует все содержимое тома во внешний TAR-архив. Этот архив копируется с помощью команды `podman volume import` для воссоздания данного тома на другой машине.

Мы разобрались, как работать с томами, пришло время углубиться в изучение их параметров.

3.1.2. Параметры монтирования томов

На протяжении всей этой главы вы использовали параметры монтирования томов. Параметр `ro` дает указание Podman смонтировать том только для чтения, а параметр `z` в нижнем регистре — перемаркировать содержимое с помощью меток SELinux, позволяющих нескольким контейнерам читать из тома и записывать в него:

```
$ podman run -d -v ./html:/var/www/html:ro,z -p 8080:8080
     quay.io/rhatdan/myimage
```

Podman поддерживает и другие интересные параметры томов.

Параметр тома U

При запуске контейнера без root иногда возникает необходимость сделать так, чтобы том принадлежал пользователю этого контейнера. Представьте, что вашему приложению нужно разрешить веб-серверу записывать данные в том. В вашем контейнере процесс веб-сервера Apache (`httpd`) запущен от имени пользователя `apache` (`UID==60`). Каталог html в вашем домашнем каталоге принадлежит вашему UID, то есть внутри контейнера он принадлежит root. Ядро не позволяет процессу, запущенному под `UID==60` внутри контейнера, вносить изменения в каталог, принадлежащий root. Нужно задать владельца тома как `UID==60`.

В непривилегированных контейнерах UID контейнера смещается относительно пользовательского пространства имен. Сопоставление моего пользовательского пространства имен выглядит следующим образом:

```
$ podman unshare cat /proc/self/uid_map
       0    3267        1
       1  100000      65536
```

`UID==0` внутри контейнера — это мой UID 3267, а `UID 1==100000, UID 2==10000 ... UID60==100059` означает, что мне нужно установить владельца каталога html как `100059`.

Я могу сделать это довольно просто, используя команду `podman unshare` следующим образом:

```
$ podman unshare chown 60:60 ./html
```

Все заработало. Единственная проблема заключается в том, что мне придется изрядно пораскинуть мозгами, чтобы выяснить, с каким UID будет запущен контейнер.

Многие образы контейнеров уже имеют UID, определенный по умолчанию. Одним из примеров является образ mariadb, запускающийся с пользователем `mysql`, `UID=999`:

```
$ podman run docker.io/mariadb grep mysql /etc/passwd
mysql:x:999:999::/home/mysql:/bin/sh
```

Если вы создали том, который будет использоваться для базы данных, нужно выяснить, к чему привязан `UID=999` в пользовательском пространстве имен. В моей системе это `UID=100998`.

Именно для таких случаев в Podman есть параметр `U`. Параметр `U` указывает Podman на рекурсивное изменение владельца (`chown`) исходного тома, чтобы по умолчанию он соответствовал UID, с которым выполняется контейнер.

Попробуйте применить этот параметр, предварительно создав каталог для базы данных. Обратите внимание, что каталог в домашней директории принадлежит вашему пользователю:

```
$ mkdir mariadb
$ ls -ld mariadb/
drwxrwxr-x. 1 dwalsh dwalsh 0 Oct 23 06:55 mariadb/
```

Теперь запустите контейнер `mariadb` с параметром `--user mysql` и смонтируйте каталог ./mariadb в /var/lib/mariadb с параметром `:U`. Обратите внимание, что теперь каталог принадлежит пользователю `mysql`:

```
$ podman run --user mysql -v ./mariadb:/var/lib/mariadb:U \
➥ docker.io/mariadb ls -ld /var/lib/mariadb
drwxrwxr-x. 1 mysql mysql 0 Oct 23 10:55 /var/lib/mariadb
```

Снова обратившись к каталогу mariadb на хосте, вы увидите, что теперь он принадлежит `UID 100998` или любому другому, который сопоставлен с `UID 999` в вашем пользовательском пространстве имен:

```
$ ls -ld mariadb/
drwxrwxr-x. 1 100998 100998 0 Oct 23 06:55 mariadb/
```

Пользовательское пространство имен — не единственный механизм безопасности, с которым приходится иметь дело при работе с rootless-контейнерами. SELinux, хоть и отлично подходит для обеспечения безопасности контейнеров, вызывает некоторые проблемы при работе с томами.

Параметры SELINUX для томов

На мой взгляд, SELinux — лучший механизм защиты файловой системы от вредоносных контейнерных процессов. С помощью SELinux за прошедшие

годы было предотвращено несколько случаев «побега из контейнера» (более подробная информация о SELinux приведена в разделе 10.8).

Как я уже объяснял ранее, тома позволяют пропускать файлы из ОС в контейнер, и с точки зрения SELinux эти файлы и каталоги должны быть правильно помечены, чтобы доступ не блокировался ядром.

Параметр `z` в нижнем регистре, который вы использовали в этой главе, дает Podman указание рекурсивно перемаркировать все содержимое исходного каталога меткой, которая с точки зрения SELinux может быть прочитана и записана всеми контейнерами. Если том не будет использоваться более чем одним контейнером, перемаркировка с помощью параметра `z` в нижнем регистре — не совсем то, что вам нужно. Ведь если другой вредоносный контейнер выйдет из-под контроля, у него будет возможность получить доступ к этим данным и прочитать/записать их. Есть также параметр `Z` в верхнем регистре, который дает Podman указание рекурсивно перемаркировать содержимое таким образом, чтобы только процессы внутри данного контейнера могли читать/записывать содержимое.

В обоих предыдущих случаях вы перемаркировали содержимое каталога. Перемаркировка отлично работает, если каталог определен для использования контейнерами. Но в некоторых ситуациях вам может понадобиться применить контейнер для изучения содержимого системного каталога, например, с целью запустить контейнер, который будет просматривать все журналы в /var/log или изучать все ваши домашние каталоги (/home/dwalsh).

> **ПРИМЕЧАНИЕ** Использование этого параметра в домашнем каталоге может иметь катастрофические последствия для системы, поскольку она рекурсивно перемаркирует все содержимое каталога, как если бы данные были приватными для контейнера. Другие ограниченные (confined) домены лишатся возможности использовать неправильно помеченные данные.

В таких случаях необходимо отключить SELinux enforcement, чтобы позволить контейнерам использовать том. Podman предоставляет параметр `--security-opt label=disable`, отключающий поддержку SELinux для отдельного контейнера. По сути, с точки зрения SELinux контейнер запускается с меткой *unconfined*:

```
$ podman run --security-opt label=disable -v /home/dwalsh:/home/dwalsh -p\
➥ 8080:8080 quay.io/rhatdan/myimage
```

В табл. 3.1 перечислены и описаны все параметры монтирования, доступные в Podman.

Таблица 3.1. Параметры монтирования тома

Параметр	Описание
`nodev`	Запрещает процессам контейнера использовать символьные или блочные устройства в томе
`noexec`	Запрещает контейнерным процессам напрямую выполнять любые бинарные файлы в томе
`nosuid`	Запрещает SUID-приложениям изменять свои привилегии в томе
`O`	Монтирует каталог с хоста в качестве временного хранилища с использованием оверлейной файловой системы. Изменения точки монтирования уничтожаются, когда контейнер завершает работу. Этот параметр полезен для совместного использования кэша пакетов из хоста в контейнер, чтобы ускорить сборку

`[r]shared|`
`[r]slave|`
`[r]private|`
`[r]unbindable`
	Задает режим mount propagation. Mount propagation управляет тем, как изменения монтируемых объектов распространяются за границы монтирования. Префикс `r` означает «рекурсивный», то есть все монтирования, расположенные под данной точкой монтирования, будут обработаны таким же образом:
	• `shared` — монтирования, выполненные с таким томом внутри контейнера, будут видны на хосте, и наоборот;
	• `slave` — монтирования, сделанные на хосте с этим томом, будут видны в контейнере, но не наоборот;
	• `private` (по умолчанию) — все монтирования, выполненные внутри контейнера, не будут видны на хосте, и наоборот;
	• `unbindable` — версия режима private, при котором нельзя сделать bind-монтирование

`rw|ro`	Монтирует том в режиме только для чтения (`ro`) или чтения-записи (`rw`). По умолчанию используется режим чтения/записи
`U`	Устанавливает правильные UID и GID хоста на основе UID и GID внутри контейнера. Используйте с осторожностью, так как это приведет к изменению файловой системы хоста
`z|Z`	Перемаркировывает файловые объекты на общих томах. Выбирайте параметр z, чтобы пометить содержимое тома как общее для нескольких контейнеров. Выбирайте параметр Z, чтобы пометить содержимое как не подлежащее совместному использованию и приватное

Для получения дополнительной информации обращайтесь к страницам man для `mount` и `mount_namespaces(7)`.

В большинстве случаев для монтирования томов в контейнеры достаточно простого параметра `--volume`. Со временем запросы на новые параметры монтирования усложняются, поэтому был добавлен новый параметр `--mount`.

3.1.3. Параметр --mount команды podman run

Параметр `podman run --mount` очень близок к лежащей в основе команде монтирования Linux. Он позволяет указать все параметры монтирования, которые понимает данная команда. Podman передает их непосредственно в ядро.

В настоящее время поддерживаются только следующие типы монтирования: `bind`, `volume`, `image`, `tmpfs` и `devpts`. Дополнительную информацию содержит страница man `podman-mount(1)`.

Тома и монтирование — отличный способ хранить данные отдельно от образа контейнера. В большинстве случаев образ контейнера рассматривается как доступный только для чтения, а любые данные, которые требуют записи или не являются специальными для приложения, должны храниться вне образа контейнера с помощью томов. Зачастую удается добиться гораздо более высокой производительности, храня данные отдельно, поскольку при чтении и записи не будет издержек, связанных с copy-on-write файловой системой. Монтирование также облегчает использование одних и тех же образов контейнеров с разными данными (табл. 3.2).

Таблица 3.2. Команды podman volume

Команда	man-страница	Описание
`create`	`podman-volume-create(1)`	Создание нового тома
`exists`	`podman-volume-exists(1)`	Проверка существования тома
`export`	`podman-volume-export(1)`	Экспорт содержимого тома в tar-архив
`import`	`podman-volume-import(1)`	Разархивация tar-архива в том
`inspect`	`podman-volume-inspect(1)`	Вывод подробной информации о томе
`list`	`podman-volume-list(1)`	Вывод списка всех томов
`prune`	`podman-volume-prune(1)`	Удаление всех неиспользуемых томов
`rm`	`podman-volume-rm(1)`	Удаление одного или более томов

РЕЗЮМЕ

- Тома полезны для отделения данных, используемых контейнером, от приложения внутри образа.
- Тома монтируют части файловой системы в окружение контейнера. Это подразумевает, что такие инструменты безопасности, как SELinux и пользовательское пространство имен, должны быть настроены на разрешение доступа.

4 Поды

В ЭТОЙ ГЛАВЕ

- ✓ Знакомство с подами
- ✓ Управление несколькими контейнерами внутри одного пода
- ✓ Использование томов с подами

Podman — это сокращение от Pod Manager («менеджер подов»). Концепция подов популяризирована проектом Kubernetes. Под — группа из одного или нескольких контейнеров, работающих вместе для достижения общей цели и разделяющих одни и те же пространства имен и cgroups (ограничения ресурсов). Podman в дополнение к этому гарантирует, что на машинах с SELinux все контейнерные процессы внутри пода имеют одинаковые метки SELinux. Это означает, что с точки зрения SELinux все они могут работать вместе.

4.1. УПРАВЛЕНИЕ ПОДАМИ

Поды Podman (рис. 4.1), как и поды Kubernetes, всегда содержат контейнер, называемый *infra*-контейнером. Иногда его именуют контейнером *pause*,

поэтому не спутайте с процессом pause rootless контейнера, упомянутым в разделе 6.2. infra-контейнер лишь держит открытыми пространства имен и cgroups, позволяя контейнерам входить в под и выходить из него. Когда Podman добавляет контейнер в под, он добавляет процесс контейнера в cgroups и пространства имен. Обратите внимание, что у infra-контейнера есть процесс мониторинга `conmon`, который следит за ним. Каждый контейнер в поде имеет свой собственный `conmon`.

Conmon — это легковесная программа на языке C, которая следит за контейнером вплоть до его завершения, позволяя исполняемой программе Podman заканчивать свою работу и снова подключаться к контейнеру. В процессе мониторинга контейнера Conmon выполняет следующие действия:

1. Запускает среду выполнения OCI, передает ей путь к файлу спецификации OCI, а также указывает на точку монтирования слоя контейнера в containers/storage. Эта точка монтирования называется rootfs.
2. Следит за контейнером до его завершения и сообщает код его завершения.
3. Обслуживает присоединение пользователя к контейнеру, предоставляя сокет для потоковой передачи STDOUT и STDERR контейнера.
4. Записывает STDOUT и STDERR в файл для журналов Podman.

ПРИМЕЧАНИЕ infra-контейнер (или контейнер pause; рис. 4.1) аналогичен процессу pause rootless контейнера. Его единственная цель — держать открытыми пространства имен и cgroups, пока контейнеры появляются и исчезают. Однако у каждого пода будет свой infra-контейнер.

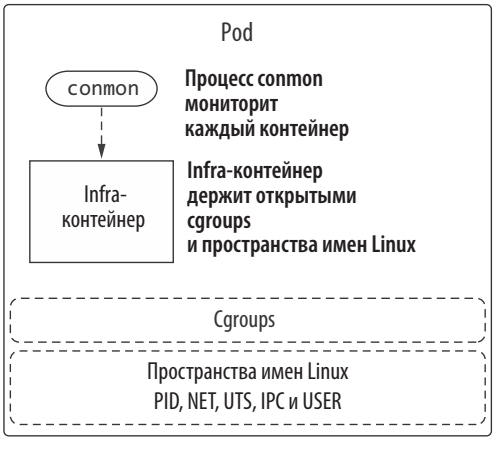


Рис. 4.1. Под Podman запускает conmon с infra-контейнером, который будет удерживать cgroups и пространства имен Linux открытыми

Поды в Podman также поддерживают init-контейнеры, как показано на рис. 4.2. Эти контейнеры запускаются до выполнения основных контейнеров в поде. Примером init-контейнера является инициализация базы данных в томе, позволяющая использовать ее основному контейнеру. Podman поддерживает следующие два класса init-контейнеров:

- *once* — запускается только при первом создании пода;
- *always* — выполняется при каждом запуске пода.

В основном контейнере запускается приложение.

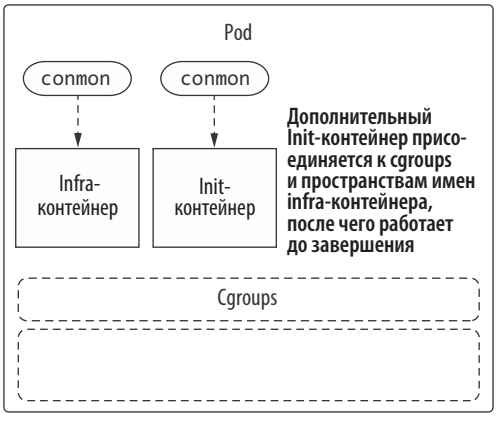


Рис. 4.2. Podman запускает все init-контейнеры с conmon. Эти init-контейнеры исследуют infra-контейнер и присоединяются к его cgroups и пространствам имен

Поды также поддерживают дополнительные контейнеры, называемые *sidecar*-контейнерами (рис. 4.4). Sidecar-контейнеры часто мониторят основной контейнер, как показано на рис. 4.3, или окружение, в котором работает основной контейнер. Документация Kubernetes (https://kubernetes.io/docs/concepts/workloads/pods) описывает поды с sidecar-контейнерами следующим образом:

> *В под может быть помещено приложение, состоящее из нескольких расположенных рядом контейнеров, которые тесно связаны между собой и нуждаются в совместном использовании ресурсов. Эти контейнеры образуют единую целостную единицу сервиса: например, один контейнер предназначен для публикации данных, хранящихся в общем томе, в то время как отдельный sidecar-контейнер обновляет эти файлы. Под объединяет эти контейнеры, ресурсы хранилища и эфемерную сетевую сущность в единое целое.*

Для желающих глубже погрузиться в изучение sidecar-контейнеров есть несколько полезных статей на сайте https://www.magalix.com/blog/the-sidecar-pattern.

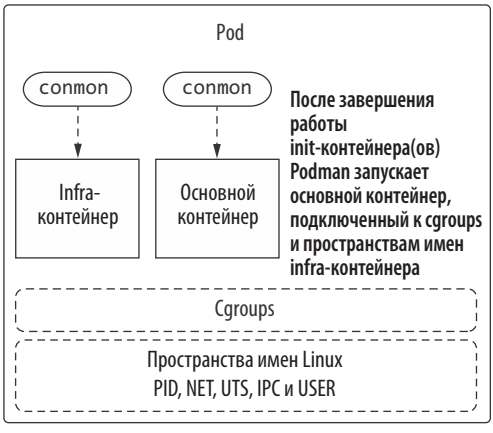


Рис. 4.3. Podman ожидает завершения работы init-контейнеров перед запуском в поде основных контейнеров с их conmon

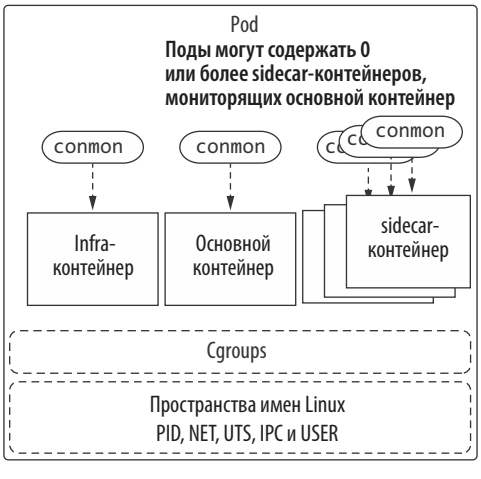


Рис. 4.4. Podman может запускать дополнительные контейнеры, называемые sidecar-контейнерами

ПРИМЕЧАНИЕ Хотя поды и могут поддерживать более одного sidecar-контейнера, я рекомендую использовать только один. Существует соблазн злоупотребить предоставляемой возможностью, особенно в Kubernetes, но это приведет к большему расходованию ресурсов и может стать трудноуправляемым.

Большое преимущество использования подов заключается в том, что вы можете управлять ими как отдельными единицами. При запуске пода запускаются все находящиеся в нем контейнеры, а при остановке пода все контейнеры останавливаются.

4.2. СОЗДАНИЕ ПОДОВ

В этом разделе вы создадите под, в котором приложение myimage будет основным контейнером. Чтобы показать совместную работу двух контейнеров в одном поде, вы добавите в этот под sidecar-контейнер, и он обновит веб-контент, используемый вашим приложением.

Вы можете создать под с именем `mypod` с помощью команды `podman pod create`, как показано ниже:

```
$ podman pod create -p 8080:8080 --name mypod --volume ./html:/var/www/html:z
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

У команды `podman pod create` многие из параметров совпадают с параметрами команды `podman container create`. Когда вы создаете внутри пода контейнер, он наследует эти параметры по умолчанию (рис. 4.5).

Обратите внимание, что, как и в предыдущих примерах, вы привязываете под к порту `-p 8080:8080`:

```
$ podman pod create -p 8080:8080 --name mypod --volume ./html:/var/www/html:z
```

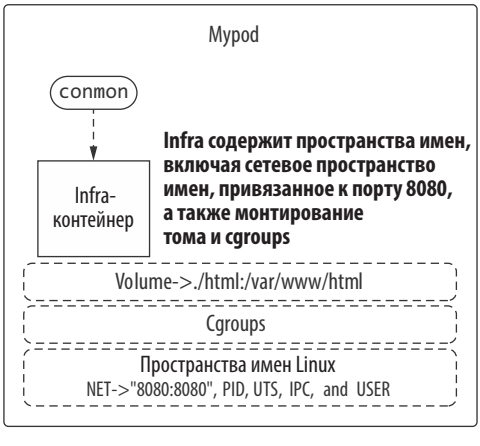


Рис. 4.5. Podman создает сетевое пространство имен и связывает порт 8080 внутри контейнера с портом 8080 на хосте. Podman создает infra-контейнер с каталогом /var/www/html с хоста в этом контейнере и присоединяет cgroups и сетевое пространство имен

Так как контейнеры внутри пода используют одно и то же сетевое пространство имен, эта привязка к порту является общей для всех контейнеров. Ядро позволяет только одному процессу слушать порт `8080`. Обратите внимание, что каталог ./html был смонтирован в под через том, `--volume ./html:/var/www/html:z`:

```
$ podman pod create -p 8080:8080 --name mypod --volume ./html:/var/www/html:z
```

Параметр `:z` заставляет Podman перемаркировать содержимое каталога. Podman будет автоматически монтировать этот каталог в каждый контейнер, который присоединится к данному поду. Контейнеры в подах имеют одну и ту же метку SELinux, это означает, что они могут совместно использовать одни и те же тома.

4.3. ДОБАВЛЕНИЕ КОНТЕЙНЕРА В ПОД

Вы создаете контейнер внутри пода с помощью команды `podman create` (рис. 4.6). Добавьте контейнер quay.io/rhatdan/myimage в под с помощью параметра `--pod mypod`:

```
$ podman create --pod mypod --name myapp quay.io/rhatdan/myimage
Cec045acb1c2be4a6e4e88e21275076fb1de5519a25fb5a55f192da70708a640
```

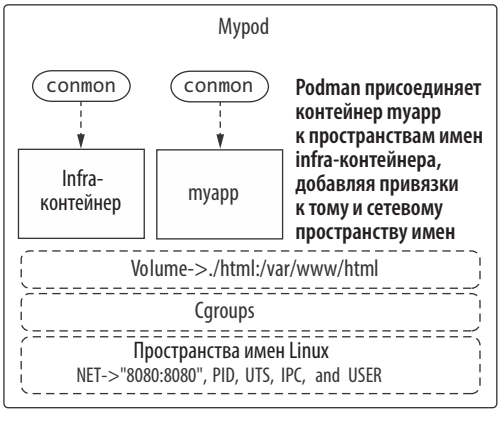


Рис. 4.6. Поскольку в поде нет init-контейнеров, первым запускается контейнер myapp

Когда вы добавляете первый контейнер в под, Podman считывает информацию, связанную с infra-контейнером, и добавляет монтирование тома к контейнеру `myapp`. Затем Podman присоединяет его к пространствам имен, занимаемым infra-контейнером. Следующим шагом является добавление в под sidecar-контейнера. Этот sidecar-контейнер будет обновлять файл index.html в томе /var/www/html, добавляя новую временную отметку каждую секунду.

Создайте простой bash-скрипт под названием html/time.sh для обновления index.html, используемого контейнером `myapp`. Чтобы он был доступен для процессов внутри пода, его нужно создать в каталоге ./html:

```
$ cat > html/time.sh << _EOL
#!/bin/sh
data() {
  echo "<html><head></head><body><h1>"; date;echo "Hello World</h1></body></html>"
```

```
  sleep 1
}
while true; do
  data > index.html
done
_EOL
```

Убедитесь, что скрипт является исполняемым. В Linux это можно сделать с помощью команды `chmod`:

```
$ chmod +x html/time.sh
```

Теперь создайте второй контейнер (`--name time`), на этот раз используя другой образ — ubi8. Контейнеры внутри подов могут использовать совершенно разные образы, даже образы из разных дистрибутивов. Напомним, что по умолчанию образы контейнеров используют совместно только ядро хоста:

```
$ podman create --pod mypod --name time --workdir /var/www/html ubi8 ./time.sh
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/000-
    shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest...
...
1be0b2fae53029d518e75def71c0d6961b662d0e8b4a1082edea5589d1353af3
```

Не забывайте о концепции коротких имен, рассмотренной в главе 2. Можно ввести длинное имя, registry.access.redhat.com/ubi8, но придется много печатать. Хорошо, что короткое имя ubi8 уже имеет псевдоним, сопоставленный с его длинным именем, а значит, вам не придется выбирать его из списка реестров. Podman показывает в выводе, где он нашел псевдоним для длинного имени:

```
$ podman create --pod mypod --name time --workdir /var/www/html ubi8 ./time.sh
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/
000-shortnames.conf)
```

Вы также использовали параметр `--workdir`, чтобы по умолчанию установить каталог для контейнера /var/www/html. Когда контейнер запускается, файл ./time.sh начнет работать в каталоге `workdir`, и на самом деле это будет /var/www/html/time.sh (рис. 4.7):

```
$ podman create --pod mypod --name time --workdir /var/www/html ubi8 ./time.sh
```

Поскольку этот контейнер будет запущен внутри пода `mypod`, он унаследует параметр `-v` ./html:/var/www/html от этого пода, то есть команда ./html/time.sh в каталоге хоста будет доступна каждому контейнеру внутри этого пода.

Podman анализирует infra-контейнер, монтирует том /var/www/html и присоединяет пространства имен при запуске sidecar-контейнера. Теперь пришло время запустить pod и посмотреть, что произойдет.

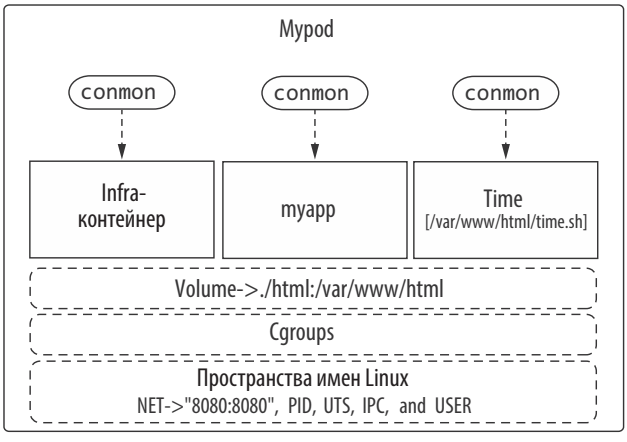


Рис. 4.7. Последним Podman запускает sidecar-контейнер с именем time

4.4. ЗАПУСК ПОДОВ

Под запускается с помощью команды `podman pod start`:

```
$ podman pod start mypod
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

Используйте команду `podman ps`, чтобы узнать, какие контейнеры запустил под:

```
$ podman ps
CONTAINER ID     IMAGE          COMMAND         CREATED         STATUS
     PORTS            NAMES
b9536ea4a8ab     localhost/podman-pause:4.0.3-1648837314                      14
     minutes ago  Up 5 seconds ago 0.0.0.0:8080->8080/tcp 8920b1ccd8b0-infra
a978e0005273 quay.io/rhatdan/myimage:latest     /usr/bin/run-http...         14
     minutes ago  Up 5 seconds ago 0.0.0.0:8080->8080/tcp myapp
be86937986e9 registry.access.redhat.com/ubi8:latest ./time.sh                13
     minutes ago  Up 5 seconds ago 0.0.0.0:8080->8080/tcp time
```

Обратите внимание, что теперь запущены три контейнера: infra-контейнер основан на образе k8s.gcr.io/pause, ваше приложение основано на quay.io/rhatdan/myimage:latest, а контейнер для обновления основан на образе registry.access.redhat.com/ubi8:latest.

Когда запускается sidecar-контейнер ubi8, он начинает изменять файл index.html с помощью скрипта time.sh. Поскольку контейнер `myapp` совместно использует монтируемый том /var/www/html, ему видны изменения в файле /var/www/html/index.html. Запустите веб-браузер, с которым вы обычно работаете, и перейдите на страницу http://localhost:8080, чтобы убедиться в работоспособности приложения, как показано на рис. 4.8.

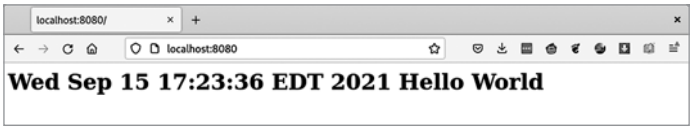


Рис. 4.8. Веб-браузер взаимодействует с myapp, запущенным в поде

Через несколько секунд нажмите кнопку обновления страницы. Вы заметите, что дата изменилась. Это указывает на то, что sidecar-контейнер работает и обновляет данные, используемые веб-сервером `myapp`, который запущен в основном контейнере (рис. 4.9).

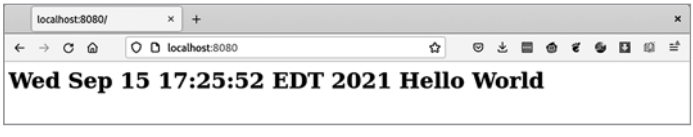


Рис. 4.9. Веб-браузер показывает, что содержимое myapp было изменено вторым контейнером, запущенным в поде

Ниже приводятся некоторые часто используемые параметры команды `podman pod start`:

- `--all`. Дает указание Podman запустить все поды.
- `--latest`. `-l`. Дает указание Podman запустить последний созданный под. Этот параметр недоступен в операционных системах Mac и Windows.

Вы запустили приложение в поде, и возможно, теперь вам понадобится его остановить.

4.5. ОСТАНОВКА ПОДА

Теперь, когда приложение успешно запущено, можно остановить под с помощью команды `podman pod stop`, как показано ниже:

```
$ podman pod stop mypod
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

Используйте команду `podman ps`, чтобы убедиться, что Podman остановил все контейнеры внутри пода:

```
$ podman ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Ниже приводятся некоторые часто используемые параметры команды

`podman pod stop`:

- `--all`. Указывает Podman остановить все поды.
- `--latest`. `-l`. Указывает Podman остановить последний запущенный под.
- `--timeout`. `-t`. Указывает Podman установить тайм-аут, когда выполняется остановка контейнеров внутри пода.

Вы создали, запустили и остановили под, и теперь пора приступить к его изучению. Начнем с вывода списка всех подов в вашей системе.

4.6. ВЫВОД СПИСКА ПОДОВ

Список подов выводится с помощью команды `podman pod list`:

```
$ podman pod list
POD ID         NAME     STATUS    CREATED        INFRA ID       # OF CONTAINERS
790fefe97b28   mypod    Exited    22 minutes ago b9536ea4a8ab   3
```

Ниже приведены наиболее часто используемые параметры команды `podman pod list`:

- `--ctr*`. Указывает Podman выводить информацию о контейнерах внутри подов.
- `--format`. Указывает Podman изменить формат вывода списка подов.

Закончив с этим списком, пора очистить поды и контейнеры.

4.7. УДАЛЕНИЕ ПОДОВ

В главе 8 я расскажу, как можно генерировать YAML-файлы Kubernetes, которые позволят запускать ваш под на других системах с помощью Podman или в Kubernetes. А сейчас вы можете удалить под с помощью команды `podman pod rm`.

Перед этим выведите список всех (`--all`) контейнеров в системе. Применив параметр `--format` для отображения только образа, ID контейнера и ID пода, вы увидите три контейнера, составляющих ваш под:

```
$ podman ps    --all --format "{{.ID}} {{.Image}} {{.Pod}}"
b9536ea4a8ab     k8s.gcr.io/pause:3.5 790fefe97b28
a978e0005273     quay.io/rhatdan/myimage:latest 790fefe97b28
be86937986e9     registry.access.redhat.com/ubi8:latest 790fefe97b28
```

Удалите под с помощью следующей команды:

```
$ podman pod rm mypod
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

Убедитесь, что он действительно исчез:

```
$ podman pod ls
POD ID         NAME     STATUS    CREATED        INFRA ID       # OF CONTAINERS
```

Прекрасно! Похоже, что ваш под удален. Выполнив следующую команду, проверьте, удалил ли Podman все контейнеры:

```
$ podman ps -a --format "{{.ID}} {{.Image}}"
```

Система полностью очищена.

Ниже приведены наиболее часто используемые параметры команды `podman pod rm` (см. также табл. 4.1):

- `--all`. Дает указание Podman удалить все поды.
- `--force`. Дает указание Podman остановить все запущенные контейнеры перед попыткой их удаления. Без этого параметра Podman будет удалять только незапущенные контейнеры.

Таблица 4.1. Команды podman pod

Команда	man-страница	Описание
`create`	`podman-pod-create(1)`	Создание нового пода
`exists`	`podman-pod-exists(1)`	Проверка существования пода
`inspect`	`podman-pod-inspect(1)`	Вывод подробной информации о поде
`kill`	`podman-pod-kill(1)`	Отправка сигнала основным процессам контейнеров в поде
`list`	`podman-pod-list(1)`	Вывод списка всех подов
`logs`	`podman-pod-logs(1)`	Получение логов для пода с одним или несколькими контейнерами
`pause`	`podman-pod-pause(1)`	Приостановка всех контейнеров в поде
`prune`	`podman-pod-prune(1)`	Удаление всех незапущенных подов и их контейнеров
`restart`	`podman-pod-restart(1)`	Перезапуск пода

Таблица 4.1 (*окончание*)

Команда	man-страница	Описание
`rm`	`podman-pod-rm(1)`	Удаление одного или более подов
`stats`	`podman-pod-stats(1)`	Вывод статистики контейнеров в поде
`start`	`podman-pod-start(1)`	Запуск пода
`stop`	`podman-pod-stop(1)`	Остановка пода
`top`	`podman-pod-top(1)`	Отображение запущенных процессов в поде
`unpause`	`podman-pod-unpause(1)`	Возобновление работы всех контейнеров в поде

РЕЗЮМЕ

- Поды — это способ объединения контейнеров в более сложные приложения, совместного использования пространств имен и ограничений ресурсов.
- Большая часть параметров, используемых подами и контейнерами, совпадает. Когда вы добавляете контейнер в под, он разделяет параметры со всеми контейнерами в данном поде.

Часть 2

Архитектура

В части 2 книги рассматривается базовое устройство Podman. В главе 5 описываются все конфигурационные файлы, используемые в Podman. Podman разработан с применением множества различных контейнерных библиотек, у каждой из которых свой способ конфигурирования. Вы узнаете, как настроить контейнерное хранилище и где хранить ваши контейнеры, а также их образы. Затем я расскажу, как настраивать реестры контейнеров, которые используются для загрузки и передачи образов. Наконец, вы узнаете о файле containers.conf, который позволяет полностью настроить работу Podman. По существу, он дает возможность изменить значения по умолчанию, используемые Podman CLI для каждого создаваемого вами контейнера.

Далее в главе 6 мы подробно рассмотрим, как работают rootless-контейнеры. Rootless (или непривилегированные) контейнеры — это ключевая особенность Podman, которая позволяет вам полноценно работать с контейнерами и подами как обычному пользователю без каких-либо дополнительных привилегий. Эта глава также познакомит вас с работой пользовательского пространства имен и использованием более одного UID в контейнере без прав root. В заключение вы узнаете о некоторых проблемах с rootless-контейнерами и способах их решения.

5 Настройка и файлы конфигурации

В ЭТОЙ ГЛАВЕ

- ✓ Применение конфигурационных файлов Podman на основе используемых библиотек
- ✓ Конфигурирование файла storage.conf
- ✓ Использование файлов registries.conf и policy.json для конфигурации
- ✓ Использование файла containers.conf для настройки других параметров по умолчанию
- ✓ Использование конфигурационных файлов для предоставления доступа к пространству имен пользователям без root-привилегий

Такие контейнерные движки, как Podman, имеют десятки встроенных в них жестко закодированных настроек по умолчанию. Эти параметры по умолчанию определяют многие аспекты функционального и нефункционального поведения Podman, в том числе настройки сети и безопасности. Разработчики Podman стараются обеспечить максимальный уровень безопасности, при этом позволив большинству контейнеров успешно выполнять свою работу. Вот и я хочу изолироваться от хоста настолько, насколько это возможно.

Параметры безопасности по умолчанию касаются того, какие Linux-привилегии использовать, какие метки SELinux установить, какой набор системных вызовов доступен для контейнеров. Существуют значения по умолчанию для ограничений ресурсов, таких как использование памяти и максимальное количество процессов, разрешенных в контейнере. Другие параметры по умолчанию включают локальный путь для хранения образов, список реестров контейнеров и даже конфигурацию системы, позволяющую работать в непривилегированном режиме. Разработчики Podman хотели дать пользователям полный контроль над этими параметрами, поэтому файлы конфигурации контейнерного движка предоставляют механизм для настройки работы Podman и других контейнерных движков.

Проблема с настройками по умолчанию заключается в том, что они представляют собой оценки на основе предположений разработчиков. Хотя большинство пользователей запускают Podman с настройками по умолчанию, иногда возникает необходимость изменить эти настройки. Не каждая среда имеет такую же конфигурацию, и возможно, вам понадобится установить на некоторых машинах другие режимы безопасности и конфигурации реестра. Ведь даже непривилегированным пользователям могут потребоваться иные настройки, чем пользователям с правами root. В этой главе я покажу вам, как настраивать различные части Podman, и объясню, где найти больше информации обо всех доступных вам «рукоятках».

Таблица 5.1. Контейнерные библиотеки, используемые Podman

Библиотека	Описание
containers/storage	Определяет хранилище образов контейнеров и другие основные хранилища, используемые контейнерными движками
containers/image	Определяет механизмы, используемые для перемещения образов контейнеров из различных типов хранилищ. Обычно используется при перемещении между реестрами контейнеров и локальными хранилищами контейнеров
containers/common	Определяет все параметры конфигурации по умолчанию для контейнерных движков, не заданные в containers/storage или containers/image
containers/buildah	Как объяснялось в главе 2, используется для создания образов контейнеров в локальном хранилище с помощью директив, определенных в Containerfile или Dockerfile; более подробная информация о Buildah приведена в приложении A

Каждая из этих библиотек имеет отдельные конфигурационные файлы, используемые для установки параметров по умолчанию, за исключением Buildah.

Контейнерные движки Podman и Buildah используют общий конфигурационный файл библиотеки containers/common, описанный в разделе 5.3 containers.conf.

ПРИМЕЧАНИЕ Все несистемные конфигурационные файлы, используемые Podman, задаются в формате TOML. Синтаксис TOML состоит из пар имя = «значение», [имен секций] и #-комментариев. Формат TOML можно упрощенно представить следующим образом:

- `[table]`
- `option = value`
- `[table.subtable1]`
- `option = value`
- `[table.subtable2]`
- `option = value`

Более полное описание языка TOML представлено на сайте https://toml.io.

При настройке Podman обычно одной из первых задач является определение того, где вы собираетесь хранить контейнеры и образы.

5.1. КОНФИГУРАЦИОННЫЕ ФАЙЛЫ ДЛЯ ХРАНИЛИЩА

Podman использует библиотеку github.com/containers/storage, которая обеспечивает методы хранения слоев файловой системы, образов контейнеров и самих контейнеров. Настройка этой библиотеки осуществляется с помощью конфигурационного файла storage.conf, он может храниться в различных директориях.

В дистрибутивах Linux часто содержится файл /usr/share/containers/storage.conf, который можно переопределить, создав другой файл /etc/containers/storage.conf. Пользователи без прав root хранят свою конфигурацию в файле $XDG_CONFIG_HOME/containers/storage.conf. Если переменная окружения $XDG_CONFIG_HOME не установлена, то используется файл $HOME/.config/containers/storage.conf. Большинство пользователей не изменяют файл storage.conf, но в некоторых случаях опытным пользователям требуется внести какие-либо персональные настройки. Самая распространенная причина для этого — необходимость изменить местоположения хранилища контейнеров.

ПРИМЕЧАНИЕ При использовании Podman в удаленном режиме (например, на компьютере с операционной системой Mac или Windows) служба Podman

использует файлы storage.conf, расположенные на компьютере Linux. Для их изменения необходимо войти в виртуальную машину. При работе с машиной Podman для входа в виртуальную машину выполните команду `podman machine ssh`. Дополнительная информация приведена в приложениях E и F.

Podman читает только один файл storage.conf и игнорирует все последующие. Сначала Podman пытается использовать файл storage.conf из вашего домашнего каталога, затем файл /etc/storage/storage.conf. Наконец, если этих двух файлов не существует, Podman читает файл /usr/share/containers/storage.conf. Файл storage.conf, который использует ваша команда Podman, просматривается с помощью команды `podman info`:

```
$ podman info --format '{{ .Store.ConfigFile }}'
/home/dwalsh/.config/containers/storage.conf
```

5.1.1. Расположение хранилища

По умолчанию в режиме rootless Podman настроен на хранение ваших образов в каталоге $HOME/.local/share/containers/storage. Местом хранения в режиме rootful по умолчанию является /var/lib/containers/storage.

Это стандартное местоположение иногда бывает необходимо изменить. Если у вас недостаточно места на диске /var или в домашнем каталоге пользователя, вам понадобится хранить ваши образы на другом диске. В файле storage.conf место хранения называется `graphRoot`, и оно переопределяется для контейнеров с правами root в файле /etc/containers/storage.conf.

В этом разделе вы измените расположение драйвера Graph Driver на /var/mystorage. Сначала войдите как пользователь root и убедитесь, что файл /etc/containers/storage.conf существует. Если он не существует, просто скопируйте туда файл /usr/share/containers/storage.conf:

```
$ sudo cp /usr/share/containers/storage.conf /etc/containers/storage.conf
```

ПРИМЕЧАНИЕ Некоторые дистрибутивы уже содержат файл /etc/containers/storage.conf.

Теперь сделайте резервную копию и откройте файл /etc/containers/storage.conf для редактирования:

```
$ sudo cp /etc/containers/storage.conf /etc/containers/storage.conf.orig
$ sudo vi /etc/containers/storage.conf
```

Замените переменную `graphroot = "/var/lib/containers/storage"` на `graphroot = "/var/mystorage"` и сохраните файл.

Ваш файл storage.conf должен содержать следующие строки:

```
$ grep -B 1 graph /etc/containers/storage.conf
# Primary Read/Write location of container storage
graphroot = "/var/mystorage"
```

Выполните `podman info`, чтобы убедиться, что изменение произошло:

```
$ sudo podman info
...
Store:
  configFile: /etc/containers/storage.conf
...
  graphDriverName: overlay
  graphOptions:
    overlay.mountopt: nodev,metacopy=on
  graphRoot: /var/mystorage
...
  volumePath: /var/mystorage/volumes
```

Обратите внимание, что в разделе storage `graphRoot` теперь — это /var/mystorage. Все образы и контейнеры будут храниться в этом каталоге.

Выполните команду `podman info` в режиме rootless. Расположение хранилища не изменилось; это по-прежнему /home/dwalsh/.local/share/containers/storage:

```
$ podman info
store:
  configFile: /home/dwalsh/.config/containers/storage.conf
  containerStore:
    number: 27
    paused: 0
    running: 0
    stopped: 27
  graphDriverName: overlay
  graphOptions: {}
  graphRoot: /home/dwalsh/.local/share/containers/storage
```

Можно создать файл $HOME/.config/containers/storage.conf и изменить его, но в системах с несколькими пользователями это не очень хорошо масштабируется. Ключ `rootless_storage_ path` позволяет изменить расположение для всех пользователей системы.

На этот раз раскомментируйте и измените строку `rootless_storage_path`:

```
$ sudo vi /etc/containers/storage.conf
```

Найдите параметр `rootless_storage_path` в файле storage.conf:

```
# rootless_storage_path = "$HOME/.local/share/containers/storage"
```

Измените его следующим образом:

```
rootless_storage_path = "/var/tmp/$UID/var/mystorage"
```

Сохраните файл storage.conf. Когда вы закончите, он должен выглядеть следующим образом:

```
$ grep -B 3 rootless_storage_path /etc/containers/storage.conf
# Storage path for rootless users
#
rootless_storage_path = "/var/tmp/$UID/var/mystorage"
```

Теперь запустите `podman info`, чтобы увидеть, что изменилось. Обратите внимание, `graphRoot` теперь указывает на каталог /var/tmp/3267/var/mystorage:

```
$ podman info
...
store:
  configFile: /home/dwalsh/.config/containers/storage.conf
...
  graphOptions: {}
  graphRoot: /var/tmp/3267/var/mystorage
```

Библиотека containers/storage поддерживает переменные окружения $HOME и $UID для этого пути. Для отмены изменений скопируйте и восстановите исходный файл storage.conf:

```
$ sudo cp /etc/containers/storage.conf.orig /etc/containers/storage.conf
```

> **ПРИМЕЧАНИЕ** Если вы работаете в системе с SELinux и меняете расположение хранилища по умолчанию, вам необходимо дать знать об этом SELinux с помощью команды `semanage`. Она сообщит SELinux, что новое местоположение должно быть помечено так, как если бы оно находилось на прежнем месте. Затем нужно изменить маркировку на диске с помощью команды `restorecon`:
>
> ```
> sudo semanage fcontext -a -e /var/lib/containers/storage /var/mystorage
> sudo  restorecon -R -v /var/mystorage
> ```

В режиме rootless вам нужно сделать следующее:

```
sudo semanage fcontext -a -e $HOME/.local/share/containers/storage/
➥ var/tmp/3267/var/mystorage
sudo restorecon -R -v /var/tmp/3267/var/mystorage
```

Иногда бывает нужно изменить драйвер хранилища или, что более вероятно, конфигурацию драйвера хранилища.

5.1.2. Драйверы хранилища

Вспомните иллюстрацию в виде свадебного торта из главы 2. Эта иллюстрация показывает, что образы часто состоят из нескольких слоев. Слои хранятся на диске в containers/storage, но когда вы запускаете на их основе контейнер, каждый слой должен быть смонтирован на предыдущий слой (рис. 5.1).

Рис. 5.1. Образы, расположенные слоями друг на друге, собираются и монтируются с помощью containers/storage

Containers/storage использует для этого концепцию файловых систем ядра Linux, называемую *многоуровневой файловой системой*. Podman, используя containers/storage, поддерживает несколько различных типов многоуровневых файловых систем. В Linux эти файловые системы называются файловыми системами *copy-on-write (CoW)*. В containers/storage они называются *драйверами*. По умолчанию Podman использует драйвер хранилища *overlay*.

> **ПРИМЕЧАНИЕ** Docker поддерживает два типа драйверов: `overlay` и `overlay2`. В `overlay2` были внесены улучшения по сравнению с `overlay`, и оригинальный драйвер теперь применяется редко. Podman использует более новый драйвер `overlay2` и называет его просто `overlay`. Если вы выбираете драйвер `overlay` в Podman, то это будет просто псевдоним для `overlay2`.

В табл. 5.2 перечислены все драйверы хранилищ, поддерживаемые Podman и containers/storage. Я рекомендую вам придерживаться использования драйвера `overlay`, поскольку именно так поступает подавляющее большинство пользователей.

Таблица 5.2. Драйверы контейнерных хранилищ

Драйвер	Описание
`overlay` (`overlay2`)	Драйвер по умолчанию, я настоятельно рекомендую использовать именно его. Он основан на оверлейной файловой системе ядра Linux. `overlay` и `overlay2` в Podman абсолютно идентичны. Это самый проверенный драйвер, применяемый подавляющим большинством пользователей
`vfs`	Самый простой драйвер, он создает полные копии каждого нижнего слоя на следующем слое. Он работает везде, но работает медленно и требует больших затрат дискового пространства
`devmapper`	Драйвер активно использовался, когда Docker только стал популярным, до появления драйвера `overlay`. Он перераспределяет размер каждого слоя до максимального. В настоящее время его использование не рекомендуется
`aufs`	Драйвер так и не был включен в состав ядра, поэтому он доступен только на нескольких дистрибутивах Linux
`btrfs`	Драйвер позволяет хранить данные на снапшотах btrfs, используя файловую систему Btrfs. Некоторые пользователи успешно используют эту файловую систему
`zfs`	Драйвер использует файловую систему ZFS, которая является проприетарной файловой системой и недоступна в большинстве дистрибутивов

У драйвера `overlay` есть несколько параметров для настройки, заслуживающих рассмотрения. Эти параметры находятся в файле storage.conf в таблице [storage.options.overlay]. Я упомяну несколько дополнительных параметров для настройки драйвера, чтобы описать сценарии их использования.

Параметр `mount_program` позволяет указать исполняемый файл для использования вместо оверлейного драйвера. Обычно Podman поставляется с исполняемым файлом `fuse-overlayfs`, который предоставляет оверлейный драйвер `FUSE` (filesystem in userspace). Podman автоматически переходит на `mount_program` `fuse-overlayfs`, если он установлен в системах без поддержки нативного оверлея без root. Обычно поддерживается нативный оверлей, однако в некоторых случаях вам может потребоваться настройка `mount_program`. Имеющиеся у `fuse-overlayfs` расширенные возможности в настоящее время не поддерживаются нативным оверлеем.

Podman активно внедряется сообществом высокопроизводительных вычислений (high performance computing, HPC). В сообществе HPC не разрешены rootful-контейнеры, и во многих случаях работать можно только с одним UID. Это означает, что некоторые HPC-системы не позволяют использовать пользовательские пространства имен с несколькими UID. Поскольку многие образы поставляются с несколькими UID, Podman добавил параметр `ignore_chown_errors` в containers/storage. Если у файлов в образах разные UID, этот параметр

позволяет объединять их в один UID. В табл. 5.3 перечислены все параметры, поддерживаемые в настоящее время контейнерными хранилищами.

ПРИМЕЧАНИЕ Вы рассмотрели лишь несколько полей storage.conf, но на самом деле их гораздо больше. Для изучения всех параметров воспользуйтесь man-страницей containers-storage.conf:

```
https://github.com/containers/storage/blob/main/docs/
    containersstorage.conf.5.md
$ man containers-storage.conf
```

Таблица 5.3. Параметры контейнерных хранилищ

Параметр	Описание
`ignore_chown_errors`	Игнорирование изменения UID файлов для контейнеров без root с одним UID. Нет записи в /etc/subuid
`mount_program`	Путь к вспомогательной программе, которую нужно использовать для монтирования файловой системы вместо использования оверлея ядра для этого. Старые ядра не поддерживали оверлей без root
`mountopt`	Список параметров монтирования через запятую, передаваемых ядру. По умолчанию используется значение `"nodev,metacopy=on"`
`skip_mount_home`	Отказ от создания `PRIVATE` bind-монтирования в домашнем каталоге хранилища
`inode`	Максимальное количество inodes в образе контейнера
`size`	Максимальный размер образа контейнера
`force_mask`	Маска разрешений для новых файлов и каталогов в образе. Значения могут быть следующие: • `private`. Устанавливает для всех объектов файловой системы значение `0700`. Другие пользователи системы не получают доступа к файлам; • `shared`. Соответствует `0755`. Каждый пользователь системы может получать доступ к файлам в образе, читать и запускать их. Это удобно при использовании контейнерного хранилища совместно с другими пользователями. Все файлы в образе доступны для чтения и выполнения любому пользователю системы. Даже /etc/shadow в вашем образе теперь может быть прочитан любым пользователем. Когда установлено значение `force_mask`, исходная маска разрешений сохраняется в `xattr`, а программа, указанная в `mount_program` (например, /usr/bin/fuse-overlayfs), представляет `xattr`-разрешения процессам внутри контейнеров

Теперь вы знаете, как настроить хранилище контейнеров! Следующая настройка — доступ к реестру контейнеров.

5.2. КОНФИГУРАЦИОННЫЕ ФАЙЛЫ РЕЕСТРОВ

Для получения и отправки образов контейнеров, обычно из реестров контейнеров, Podman использует библиотеку github.com/containers/image. Реестры указываются в конфигурационном файле registries.conf, проверка подписи образов осуществляется с помощью файла policy.json. Как и в случае с хранилищем контейнеров storage.conf, большинство пользователей никогда не изменяет эти файлы, полагаясь на значения по умолчанию.

5.2.1. registries.conf

registries.conf — это общесистемный файл конфигурации для реестров образов контейнеров. Podman использует существующий $HOME/.config/containers/registries.conf; если же его не существует, то /etc/containers/registries.conf.

> **ПРИМЕЧАНИЕ** При использовании Podman в удаленном режиме (например, на компьютере с операционной системой Mac или Windows) файлы registries.conf хранятся на стороне сервера Linux. Для внесения изменений необходимо войти по ssh на этот Linux-сервер. На машине Podman вы можете выполнить команду `podman machine ssh`. Дополнительная информация приведена в приложениях E и F.

Важнейшим в файле registries.conf является `unqualified-searchregistries`. Это поле задает массив реестров `host[:port]`, которые перебираются по порядку при поиске по коротким именам. Если в параметре `unqualified-searchregistries` указать только один реестр, Podman будет работать аналогично Docker и принудительно установит для пользователя единственный доступный реестр.

В качестве упражнения предлагаю вам изменить стандартные реестры для поиска, которые будут использоваться Podman. Сначала необходимо сделать резервную копию файла /etc/containers/registries.conf, а затем удалить docker.io и добавить example.com:

```
$ sudo cp /etc/containers/registries.conf
     /etc/containers/registries.conf.orig
$ sudo vi /etc/containers/registries.conf
```

Найдите следующую строку:

```
unqualified-search-registries = ["registry.fedoraproject.org",
     "registry.access.redhat.com", "docker.io", "quay.io"]
```

Измените ее, чтобы получилось следующее:

```
unqualified-search-registries = ["registry.fedoraproject.org",
     "registry.access.redhat.com", "example.com", "quay.io"]
```

Сохраните файл и выполните `podman info` для проверки изменений:

```
$ podman info
registries:
search:
  - registry.fedoraproject.org
  - registry.access.redhat.com
  - example.com
  - quay.io
```

Если после этого вы попытаетесь выполнить загрузку образа через неизвестное короткое имя, то увидите следующую подсказку:

```
$ podman pull foobar
? Please select an image:
  ▸ registry.fedoraproject.org/foobar:latest
    registry.access.redhat.com/foobar:latest
    example.com/foobar:latest
    quay.io/foobar:latest
```

Скопируйте оригинал в файл registries.conf:

```
$ sudo cp /etc/containers/registries.conf.orig /etc/containers/registries.conf
```

В табл. 5.4 описаны все параметры, доступные в файлах registries.conf.

Таблица 5.4. Глобальные поля registries.conf

Поле	Описание
`unqualified-search-registries`	Содержит массив реестров в формате `host[:port]`, перебираемых по порядку при загрузке образа по короткому имени
`short-name-mode`	Определяет, как Podman должен обрабатывать короткие имена. Возможные значения: • `enforcing`. Если существует один неквалифицированный реестр поиска, то будет использоваться он. Если определены два или более реестра и вы запускаете Podman в терминале, пользователю предлагается выбрать один из реестров поиска; в противном случае выдается сообщение об ошибке; • `permissive`. Действует подобно `enforcing`, но не приводит к сообщению об ошибке при отсутствии терминала, просто используя для поиска каждую запись в неквалифицированных реестрах до достижения успеха; • `disabled`. Используются все неквалифицированные реестры поиска без запросов выбора

Поле	Описание
`credential-helpers`	Определяет, что массив credential-helpers по умолчанию используется в качестве внешних хранилищ учетных данных. Обратите внимание, containers-auth.json — это зарезервированное значение для использования файлов аутентификации, как указано в `containers-auth.json(5)`. credential helpers устанавливаются в `["containers-auth.json"]`, если они не указаны

5.2.2. Запрет на загрузку из реестров контейнеров

В registries.conf можно настроить еще одну интересную возможность — запрещение загрузки из реестра контейнеров пользователями. В следующем примере вы настроите registries.conf на блокировку получения образов из docker.io. В файле registries.conf есть специальная запись в таблице `[[registry]]`, указывающая, как работать с отдельными реестрами контейнеров. Таблицу можно добавлять несколько раз — по одному на каждый реестр:

```
$ sudo vi /etc/containers/registries.conf
```

Добавьте следующее:

```
[[registry]]
Location = "docker.io"
blocked=true
```

Сохраните файл. Проверьте настройки с помощью `podman info`:

```
$ podman info
...
registries:
 Docker.io:
  Blocked: true
  Insecure: false
  Location: docker.io
  MirrorByDigestOnly: false
  Mirrors: null
  Prefix: docker.io
  search:
  - registry.fedoraproject.org
  - registry.access.redhat.com
  - docker.io
  - quay.io
```

Теперь попытайтесь загрузить образ из docker.io:

```
$ podman pull docker.io/ubuntu
Trying to pull docker.io/library/ubuntu:latest...
Error: initializing source docker://ubuntu:latest: registry docker.io is
```

```
blocked in /etc/containers/registries.conf or
/home/dwalsh/.config/containers/registries.conf.d
```

Это показывет, что у администраторов есть возможность блокировать содержимое определенных реестров. В табл. 5.5 описаны параметры, доступные для таблицы `[[registry]]` в файле registries.conf.2.

ПРИМЕЧАНИЕ Скопируйте исходный registries.conf, чтобы использовать docker.io при дальнейшем изучении этой книги:

```
$ sudo cp /etc/containers/registries.conf.orig/
➥ etc/containers/registries.conf
```

Таблица 5.5. Поля таблицы [[registry]]

Поле	Описание
`location`	Имя реестра/репозитория, к которому нужно применить фильтры
`prefix`	Выбор указанной конфигурации при попытке загрузить образ, соответствующий определенному префиксу
`insecure`	При значении true — разрешение незашифрованных HTTP-соединений, а также TLS-соединений с ненадежными сертификатами
`blocked`	При значении true — запрещение загрузки образов с совпадающими именами

Некоторые пользователи работают в системах, полностью изолированных от интернета. И все же им приходится использовать приложения, основанные на образах из интернета. В качестве примера такой ситуации рассмотрим приложение, которое ожидает использования registry.access.redhat.com/ubi8/httpd-24:latest, но не имеет доступа к registry.access.redhat.com в интернете. Поместив загруженный образ во внутренний реестр, настройте зеркалирование в registries.conf. Если вы настроите запись в registries.conf, она будет выглядеть следующим образом:

```
[[registry]]
location="registry.access.redhat.com"
[[registry.mirror]]
location="mirror-1.com"
```

Ваши пользователи будут применять команду `podman pull`:

```
$ podman pull registry.access.redhat.com/ubi8/httpd-24:latest
```

Теперь Podman фактически работает с mirror-1.com/ubi8/httpd-24:latest, но пользователи не заметят разницы.

Вы узнали, как настраивать хранилище и реестры. Пришло время перейти к настройке всех ключевых параметров Podman.

ПРИМЕЧАНИЕ Мы рассмотрели лишь несколько полей registries.conf, но их гораздо больше. Для изучения всех полей воспользуйтесь man-страницей `containers-registries.conf(5)`:

```
$ man containers-registries.conf
https://github.com/containers/image/blob/main/docs/
    containersregistries.conf.5.md
```

5.3. КОНФИГУРАЦИОННЫЕ ФАЙЛЫ КОНТЕЙНЕРНЫХ ДВИЖКОВ

Podman и другие контейнерные движки используют библиотеку github.com/containers/common для обработки настроек по умолчанию, не связанных с хранением контейнеров или их реестрами. Эти параметры берутся из файла containers.conf. Podman считывает файлы, приведенные в табл. 5.6 (если они существуют).

Таблица 5.6. Файлы containers.conf, считываемые в режимах rootful и rootless в Podman

Файл	Описание
/usr/share/containers/containers.conf	Обычно поставляется с дистрибутивом по умолчанию
/etc/containers/containers.conf	Системный администратор использует этот файл для задания и изменения различных значений по умолчанию
/etc/containers/containers.conf.d/*.conf	Некоторые пакетные инструменты могут помещать в этот каталог дополнительные файлы настроек, отсортированные по порядку

При работе в режиме rootless Podman также читает файлы, указанные в табл. 5.7 (если они существуют).

Таблица 5.7. Файлы containers.conf, читаемые Podman в режиме rootless

Файл	Описание
$HOME/.config/containers/containers.conf	Пользователи создают этот файл, чтобы переопределить системные настройки по умолчанию
$HOME/.config/containers/containers.conf.d/*.conf	При желании пользователи могут поместить сюда файлы, которые будут отсортированы по порядку

В отличие от storage.conf и registries.conf, файлы containers.conf объединяются вместе, и они не отменяют полностью предыдущие версии. Отдельные поля переопределяют аналогичные поля в файле containers.conf вышестоящего уровня. Podman не требует существования какого-либо файла containers.conf, поскольку имеет встроенные значения по умолчанию. Большинство систем поставляются только с переопределениями по умолчанию значений дистрибутива в файле /usr/share/containers/containers.conf.

> **ПРИМЕЧАНИЕ** Podman поддерживает переменную окружения CONTAINERS_CONF. Все остальные файлы containers.conf при этом игнорируются. Это полезно для тестирования окружения, а также придает уверенности, что никто не изменил настройки Podman по умолчанию.

В настоящее время containers.conf поддерживает пять различных таблиц, как показано в табл. 5.8. При изменении параметров необходимо убедиться, что вы находитесь в правильной таблице.

Таблица 5.8. Таблицы containers.conf

Таблица	Описание
`[containers]`	Конфигурация для запуска отдельных контейнеров. Примерами служат пространства имен, в которые следует помещать контейнеры, включение или отключение SELinux, а также переменные окружения по умолчанию для контейнеров
`[engine]`	Конфигурации по умолчанию. Примерами являются система логирования по умолчанию, пути к среде выполнения OCI и расположение conmon
`[service_destinations]`	Данные для удаленного подключения с командой `podman --remote`. Удаленное подключение рассматривается в главе 9
`[secrets]`	Информация о драйвере плагина `secrets` для контейнеров
`[network]`	Специальные настройки для конфигурации сети, включая имя сети по умолчанию, расположение плагинов CNI и подсети по умолчанию

Многим пользователям Podman бывает нужно изменить стандартные способы запуска контейнеров в среде. Ранее я рассказывал, как участники сообщества HPC стремятся использовать Podman для запуска своих рабочих нагрузок, но при этом они очень требовательны к тому, какие тома присоединяются к контейнерам, какие переменные окружения добавляются и какие пространства имен включаются.

Допустим, вам нужно, чтобы все ваши контейнеры имели одинаковые переменные окружения. Давайте попробуем сделать это на следующем примере. Запустите podman, чтобы узнать окружение по умолчанию в образе ubi8:

```
$ podman run --rm ubi8 printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
container=oci
HOME=/root
HOSTNAME=ba4acf180386
```

> **ПРИМЕЧАНИЕ** При использовании Podman в удаленном режиме (например, на компьютере с операционной системой Mac или Windows) большинство настроек файлов containers.conf используются с Linux-сервера. Файл containers.conf в домашнем каталоге пользователя используется для хранения данных о подключении, этот вопрос рассматривается в главе 9. Клиенты Mac и Windows рассматриваются в приложениях E и F.

Теперь создайте файл env.conf в домашнем каталоге с параметром env="[foo=bar]":

```
$ mkdir -p $HOME/.config/containers/containers.conf.d
$ cat << _EOF > $HOME/.config/containers/containers.conf.d/env.conf
[containers]
env=[ "foo=bar" ]
_EOF
Run any container and you see the foo=bar environment set.
$ podman run --rm ubi8 printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
container=oci
foo=bar
HOME=/root
HOSTNAME=406fc182d44b
```

Я использую containers.conf при настройке запуска Podman внутри контейнера. Многим пользователям нужно запускать Podman в контейнере для CI/CD-систем или просто для тестирования более новых версий Podman, чем это позволяет их дистрибутив. Поскольку часто возникают трудности с запуском Podman в контейнере, я решил помочь им, попробовав создать образ по умолчанию quay.io/podman/stable. Создавая этот образ, я осознал, что в Podman некоторые параметры по умолчанию не очень хорошо работают при запуске в контейнере, поэтому для изменения этих параметров я использовал файл containers.conf. Мой файл containers.conf доступен по этой ссылке: http://mng.bz/o5DM.

Вы можете ознакомиться с этим файлом, запустив образ:

```
$ podman run quay.io/podman/stable cat /etc/containers/containers.conf
[containers]
```

```
netns="host"
userns="host"
ipcns="host"
utsns="host"
cgroupns="host"
cgroups="disabled"
log_driver = "k8s-file"
[engine]
cgroup_manager = "cgroupfs"
events_logger="file"
runtime="crun"
```

Во время написания этого файла я обдумал некоторые моменты. Прежде всего, поскольку Podman работает внутри контейнера, я решил отключить все cgroups и пространства имен, за исключением mount и пространства имен пользователя. Если пользователи зададут cgroups или сконфигурированные пространства имен, то контейнер, запущенный через Podman в другом контейнере, будет следовать правилам родительского Podman:

```
[containers]
netns="host"
userns="host"
ipcns="host"
utsns="host"
cgroupns="host"
cgroups="disabled"
```

Во многих дистрибутивах `log_driver` event logger и cgroup manager — это, по умолчанию, journald и systemd соответственно. Но внутри контейнера journald и systemd не запущены, поэтому контейнерный движок будет использовать файловую систему:

```
[containers]
log_driver = "k8s-file"
[engine]
cgroup_manager = "cgroupfs"
events_logger="file"
```

И последнее: выбор среды выполнения OCI `crun`, а не `runc`, прежде всего потому, что она намного меньше:

```
[engine]
runtime="crun"
```

Теперь попробуйте запустить контейнер внутри контейнера. Чтобы это сработало, необходимо запустить образ podman/stable с параметром `--user podman`, что заставит Podman внутри контейнера работать в режиме rootless. Поскольку образ podman/stable использует драйвер `fuse-overlay` внутри контейнера, необходимо также добавить устройство /dev/fuse:

```
$ podman run --device /dev/fuse --user podman quay.io/podman/stable podman
➥ run ubi8-micro echo hi
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/
➥ 000-shortnames.conf
Trying to pull registry.access.redhat.com/ubi8:latest...
Getting image source signatures
Copying blob sha256:5368f457acd16b337e2b150741f727c46f886c69eea
➥ 1a4d56d0114c88029ed87
...
hi
```

ПРИМЕЧАНИЕ Мы рассмотрели лишь несколько полей файла container.conf, но их гораздо больше. Для изучения всех полей воспользуйтесь man-страницей `container.conf(5)`:

```
$ man containers.conf
https://github.com/containers/common/blob/main/docs/containers.conf.5.md
```

Вы близко познакомились со средствами конфигурации, используемыми в таких контейнерных инструментах, как Podman. Далее вы узнаете о некоторых других системных конфигурационных файлах, необходимых Podman.

5.4. СИСТЕМНЫЕ ФАЙЛЫ КОНФИГУРАЦИИ

При запуске Podman без root вы используете файлы /etc/subuid и /etc/subgid, чтобы указать диапазоны UID для ваших контейнеров. Как я объяснял в разделе 3.1.2, чтобы найти выделенные для учетной записи вашего пользователя диапазоны UID и GID, Podman читает файлы /etc/subuid и /etc/subgid. После этого Podman запускает /usr/bin/newuidmap и /usr/bin/newgidmap, которые проверяют, что указанные Podman диапазоны UID и GID действительно выделены для вас. В некоторых случаях необходимо изменить эти файлы, чтобы добавить UID. Такие инструменты, как `useradd`, автоматически обновляют /etc/subuid и /etc/subgid при добавлении новых пользователей в систему. Например, когда я устанавливал систему на свой ноутбук, `useradd` настроил мою учетную запись на использование UID `3267` и добавил сопоставление `dwalsh:100000:65536` в /etc/subuid и /etc/subgid. На рис. 5.2 показано, как выглядят в моей системе контейнеры, основанные на этом сопоставлении.

ПРИМЕЧАНИЕ Необходимо сохранять диапазоны UID уникальными для каждого пользователя и следить за тем, чтобы они не пересекались с системными UID. Podman и система не проверяют отсутствие пересечения. Если бы у двух разных пользователей оказались одинаковые UID в их диапазоне, процессы в контейнерах смогли бы атаковать друг друга через пространство имен пользователей. Проверка проводится ручным процессом. Инструмент `useradd` автоматически выбирает уникальные диапазоны.

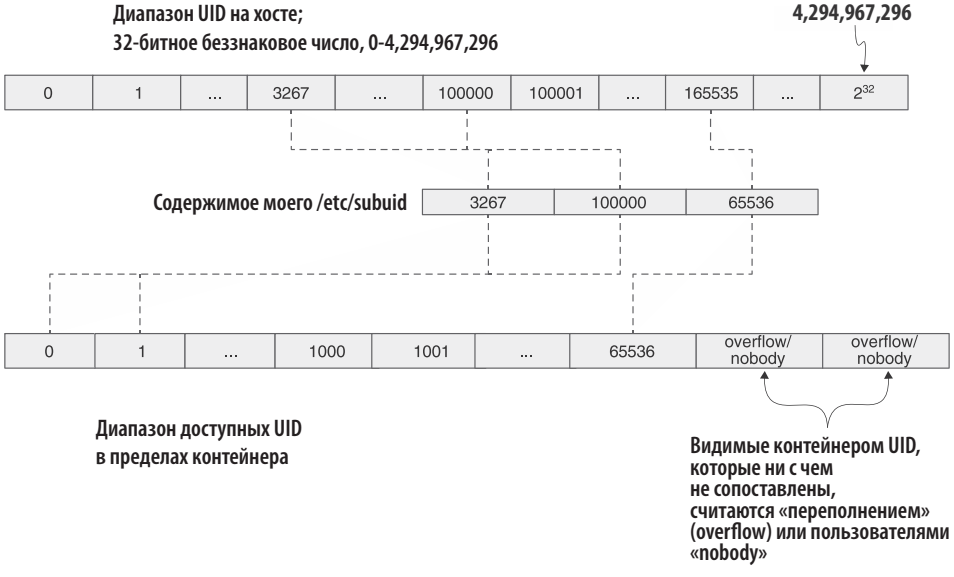


Рис. 5.2. Сопоставление пользовательского пространства имен с контейнерами

Как объясняется на man-страницах subuid(5) и subgid(5), каждая строка в /etc/subuid и /etc/subgid содержит имя пользователя и диапазон, соответственно, подчиненных идентификаторов пользователя или GID, разрешенных пользователю. Запись задается тремя полями, разделенными двоеточием:

- имя пользователя или UID;
- числовой ID подчиненного пользователя или ID группы;
- количество ID подчиненных пользователей или ID групп.

> **СОВЕТ** Изменения в /etc/subuid и /etc/subgid не обязательно сразу отражаются в учетной записи пользователя. С этой проблемой часто сталкиваются пользователи, изменившие эти файлы уже после того, как запустили Podman. Запомните: когда Podman начинает работу, он запускает процесс `podman pause` в пользовательском пространстве имен, а затем все остальные контейнеры присоединяются к этому пользовательскому пространству имен процесса Podman. Чтобы изменение пространства имен вступило в силу, необходимо выполнить команду `podman system migrate`. Она останавливает процесс `podman pause` и заново создает пользовательское пространство имен.

Новые версии операционной системы, в частности поставляющие /usr/bin/newuidmap и /usr/bin/newgidmap пакеты, получают возможность передавать

содержимое этих файлов по сети с сервера LDAP. В Fedora эти исполняемые файлы поставляются в пакете shadow-utils. Эта функция есть в версиях 4.9 и более поздних.

РЕЗЮМЕ

- В Podman есть несколько конфигурационных файлов, основанных на библиотеках, которые он использует.
- Конфигурационные файлы совместно используются в средах rootful и rootless.
- Файл storage.conf используется для настройки containers/storage, в том числе настройки драйвера хранилища и места, где будут храниться контейнеры и их образы.
- Файлы registries.conf и policy.json применяются для настройки библиотеки containers/image. Прежде всего это касается доступа к реестрам контейнеров, коротких имен и зеркалирования.
- Файл containers.conf нужен для настройки всех остальных параметров по умолчанию, используемых в Podman.
- Системные файлы конфигурации /etc/subuid и /etc/subgid применяются для настройки пользовательского пространства имен, необходимого для работы Podman без root.

6 Непривилегированные (rootless) контейнеры

В ЭТОЙ ГЛАВЕ

- ✓ Почему режим rootless безопаснее
- ✓ Как Podman работает с пространствами имен пользователя и mount
- ✓ Архитектура rootless режима Podman

Эта глава глубоко погрузит вас в то, что происходит при запуске Podman в режиме rootless. Я считаю, полезно понимать, что случается при этом, а также знать и о проблемах, которые может вызвать запуск в таком режиме. Контейнеризованные приложения внедряются последние несколько лет, а некоторым высокозащищенным средам не удалось воспользоваться преимуществами новой технологии.

Системы высокопроизводительных вычислений (high performance computing, HPC) работают на самых быстрых компьютерах в мире. Они, как правило, находятся в национальных лабораториях или университетах и работают с информацией, имеющей высокий уровень защиты. Здесь строго запрещено использование привилегированных (rootful) контейнеров, так как обрабатываются самые охраняемые данные в мире. HPC-системы работают с огромными датасетами, включая искусственный интеллект, ядерное оружие, глобальные погодные условия и медицинские исследования. Обычно в таких системах тысячи компьютеров

с многопользовательским доступом, и они должны быть защищены. Считается, что в HPC-системах запуск демонов от имени root слишком небезопасен. Если вредоносный процесс в контейнере, запущенный от имени root, выйдет из-под контроля, он может получить доступ к особо важной информации. В связи с этим администраторы HPC-окружений не использовали контейнеры Open Container Initiative (OCI), пока не появился Podman. И сейчас сообщество HPC работает над переходом на непривилегированный Podman.

Администраторы крупных финансовых компаний не разрешают пользователям и разработчикам получать root-доступ к своим системам также из опасений за финансовую информацию. Крупнейшие финансовые фирмы испытывали трудности с полноценным внедрением контейнеров OCI. На рис. 6.1 показано, как клиент Docker, хотя он и может быть запущен без полномочий root, подключается к работающему в привилегированном режиме демону, который и предоставляет ему полный доступ к операционной системе хоста.

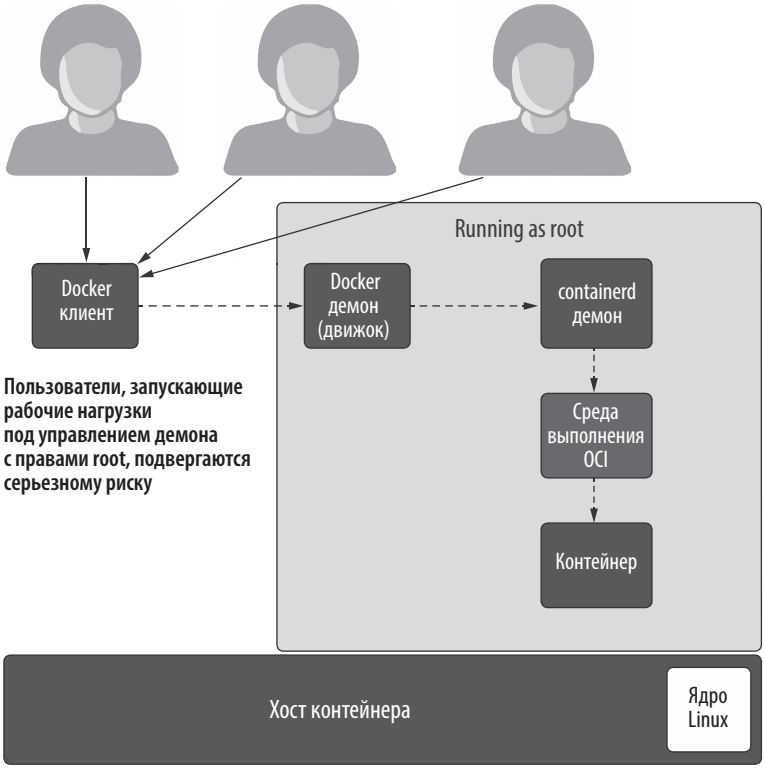


Рис. 6.1. Рабочие нагрузки пользователей, использующий один демон, который работает как root, — это небезопасно

Получается, что позволять пользователям в общей вычислительной системе запускать контейнеры, обращаясь к одному и тому же демону с правами root, слишком небезопасно. Безопаснее запускать контейнеры каждого пользователя в режиме rootless под учетными записями самих пользователей.

На рис. 6.2 показано, как несколько пользователей запускают Podman независимо друг от друга без прав суперпользователя.

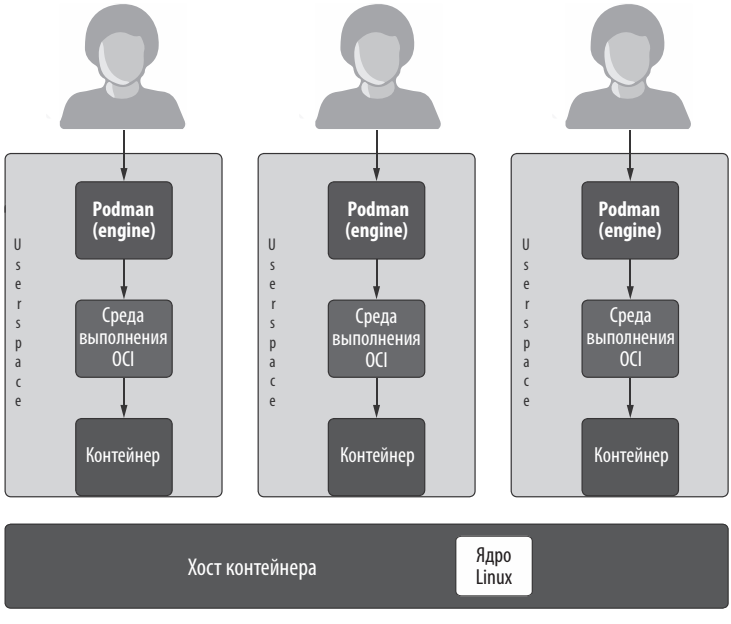


Рис. 6.2. Гораздо более безопасно, когда каждый процесс работает в своем отдельном пользовательском пространстве

Система Linux была изначально разработана с разделением между привилегированным (rootful) и непривилегированным (rootless) режимами. В Linux почти все задачи выполняются без привилегий root. Привилегированные операции требуются только для модификации ядра операционной системы. Почти все приложения, работающие в контейнерах, веб-серверы, базы данных и пользовательские инструменты запускаются без root. Эти приложения не изменяют основные части системы. К сожалению, большинство образов, которые вы найдете в репозиториях, требуют root-прав — или, по крайней мере, запускаются от имени root, а затем понижают уровень привилегий.

В корпорациях администраторы очень неохотно предоставляют root-доступ своим пользователям. Если вы получаете корпоративный ноутбук от своего

работодателя, root-доступ вам обычно не предоставляется. Администраторам необходимо контролировать, что установлено в их системах огромных масштабов, — они должны иметь возможность обновлять сотни и тысячи машин одновременно, поэтому контроль над операционной системой приобретает решающее значение. Если другой специалист администрирует вашу машину, ему необходимо держать под контролем, кто получает root-доступ.

Как человек, заботящийся о безопасности, я до сих пор вздрагиваю, увидев sudo без пароля. Когда я только начал работать с Docker, я был потрясен тем, что в нем поощряется использование группы Docker, что дает пользователям полный root-доступ на хосте без пароля. Святой Грааль хакеров — применить эксплойт для получения root-доступа; это дает злоумышленникам полный контроль над системой.

Случившийся «побег из контейнера», как бы плохо это ни было, все же не самое страшное, если он произошел в режиме без root. В этом случае хакеры контролируют только непривилегированные процессы. А вот в ситуации с root-эксплойтом они получают полный контроль над системой и всеми данными (игнорируя другие механизмы безопасности, например SELinux). Среди целей разработки Podman — возможность запуска максимального количества рабочих нагрузок без прав root и упрощение запуска контейнеров в безопасном для основной операционной системы режиме.

6.1. КАК РАБОТАЕТ PODMAN В РЕЖИМЕ ROOTLESS?

Вы когда-нибудь задумывались, что происходит «под капотом» rootless контейнера Podman? В главе 2 все примеры Podman работали в режиме без root. Давайте взглянем, что происходит внутри таких контейнеров. Я расскажу о каждом компоненте, а затем разберу все необходимые этапы процесса.

> **ПРИМЕЧАНИЕ** Часть этого раздела скопирована и переписана из статьи «What Happens behind the Scenes of a Rootless Podman Container?» (https://www.redhat.com/sysadmin/behind-scenes-podman), которую я написал вместе со своими коллегами Мэтью Хоном (Matthew Heon) и Джузеппе Скриваном (Guiseppe Scrivano).

Давайте сначала очистим систему. В чистом окружении запустим контейнер из образа quay.io/rhatdan/myimage (напоминаю, что команда `podman rmi –all --force` удаляет все образы и контейнеры из системы).

```
$ podman rmi --all --force
Untagged: registry.access.redhat.com/ubi8/httpd-24:latest
Untagged: registry.access.redhat.com/ubi8-init:latest
```

```
Untagged: localhost/myimage:latest
Untagged: quay.io/rhatdan/myimage:latest
Deleted:d2244a4379d6f1981189d35154beaf4f9a17666ae3b9fba680ddb014eac72adc
Deleted:82eb390304938f16dd707f32abaa8464af8d4a25959ab342e25696a540ec56b5
Deleted:8773554aad01d4b8443d979cdd509e7b8fa88ddbc966987fe91690d05614c961
```

Теперь, в очищенной системе, нужно получить образ приложения — quay.io/rhatdan/myimage — из репозитория, в который вы отправили его в главе 2. С помощью следующей команды заново запустите приложение на своем компьютере. Команда извлечет образ из репозитория и запустит контейнер `myapp` на вашем хосте.

```
$ podman run -d -p 8080:8080 --name myapp quay.io/rhatdan/myimage
Trying to pull quay.io/rhatdan/myimage:latest...
...
2f111737752dcbf1a1c7e15e807fb48f55362b67356fc10c2ade24964e99fa09
```

Разберемся поглубже, что произошло, когда вы запустили контейнер Podman без root. Прежде всего, Podman нужно было настроить пользовательское пространство имен. В следующем разделе я объясню, как это работает.

6.1.1. Образы с содержимым, принадлежащим нескольким идентификаторам пользователей (UIDs)

В Linux идентификаторы пользователей (UID) и идентификаторы групп (GID) назначаются процессам и хранятся в объектах файловой системы. Объекты файловой системы также имеют назначенные им разрешения. Linux контролирует доступ процессов к файловой системе на основе UID и GID. Это называется *дискреционным управлением доступом* (discretionary access control, DAC). Когда вы входите в систему Linux, ваши пользовательские процессы без полномочий root запускаются с одним UID — скажем, `1000`, — но образы контейнеров обычно имеют несколько разных UID в своих слоях. Давайте проверим, какие UID необходимы для запуска нашего образа. В этом примере вы исследуете все UID, определенные в образе контейнера, запустив другой контейнер.

Следующая команда запустит контейнер с образом quay.io/rhatdan/myimag. Вам нужно запустить его от имени пользователя root (`--user=root`) внутри контейнера, чтобы исследовать каждый файл в образе.

```
$ podman run --user=root --rm quay.io/rhatdan/myimage -- bash -c "find /
➥ -mount -printf \"%U=%u\n\" | sort -un" 2>/dev/null
```

Поскольку это всего лишь временный контейнер, используйте параметр `--rm`, чтобы контейнер был удален после завершения его работы. Контейнер запускает bash-скрипт, который находит все UID и пользователей, связанных с каждым файлом/каталогом в контейнере. Этот скрипт передает выходные данные,

отображая уникальные вхождения, и перенаправляет `stderr` в /dev/null для устранения каких-либо ошибок.

```
$ podman run --user=root --rm quay.io/rhatdan/myimage -- bash -c "find /
➥-mount -printf \"%U=%u\n\" | sort -un" 2>/dev/null
0=root
48=apache
1001=default
65534=nobody
```

Из вывода видно, что наш образ контейнера использует четыре разных UID, показанных в табл. 6.1.

Таблица 6.1. Уникальные UID, необходимые для запуска контейнера

UID	Имя	Описание
`0`	`root`	Владеет большей частью содержимого в образе контейнера
`48`	`apache`	Владеет всем содержимым Apache
`1001`	`default`	Пользователь по умолчанию, от имени которого работает контейнер
`65634`	`nobody`	Назначается каждому UID, который не сопоставлен с контейнером

Чтобы вы могли загрузить образ контейнера в свой домашний каталог, Podman должен хранить по крайней мере три разных UID: `0`, `48` и `1001`. Поскольку ядро Linux не позволяет непривилегированным учетным записям использовать более одного UID, вам запрещено создавать файлы с разными UID. Нужно воспользоваться пользовательским пространством имен.

6.1.2. Пользовательское пространство имен (user namespace)

Linux поддерживает концепцию пользовательских пространств имен, которая основана на сопоставлении UID/GID хоста с различными UID и GID внутри этого пространства имен. Вот как описывает это справочное руководство Linux:

```
$ man user namespaces
...
```

Пользовательские пространства имен изолируют связанные с безопасностью идентификаторы и атрибуты, в частности идентификаторы пользователей (user ID, UID) и идентификаторы групп (group ID, GID) (см. справочное руководство, `credentials(7)`), корневой каталог, ключи (см. `keyrings(7)`) и привилегии (см. `capabilities(7)`). Идентификаторы пользователя и группы процесса могут быть разными внутри и вне пользовательского пространства имен. Процесс может

иметь обычный, непривилегированный ID пользователя вне пользовательского пространства имен и в то же время иметь ID пользователя `0` внутри этого пространства имен. Другими словами, процесс имеет полные привилегии для операций внутри пользовательского пространства имен, но не имеет привилегий для операций за его пределами.

Поскольку вашему контейнеру требуется более одного UID, процесс Podman сначала создает пространство имен пользователя, где он имеет доступ к большему количеству UID, и входит в него. Podman должен также смонтировать несколько файловых систем для запуска контейнера. Команды монтирования не допускаются вне пространства имен пользователя (наряду с пространством имен mount). На рис. 6.3 показаны UID, используемые в пространстве имен пользователя.

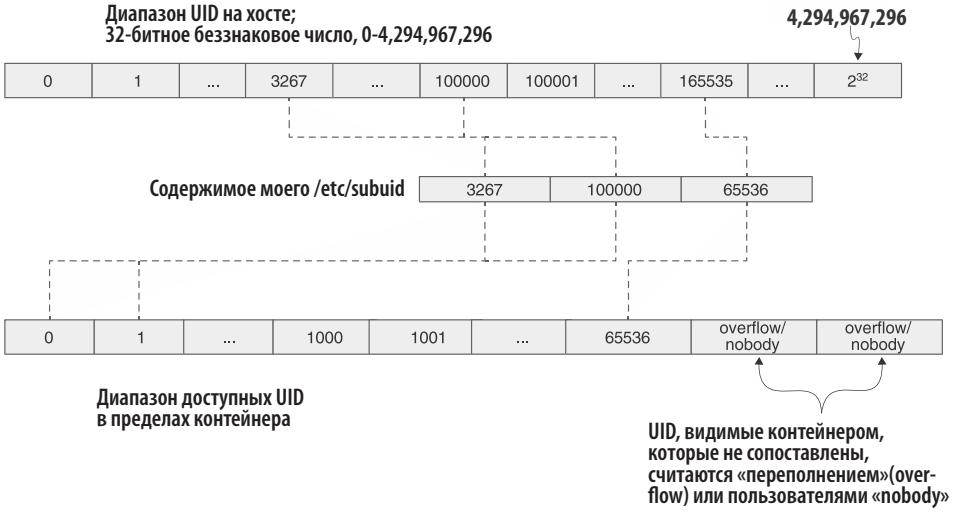


Рис. 6.3. Сопоставление UID пользовательского пространства имен и контейнера

После установки системы я использовал утилиту `useradd` для создания своей учетной записи. Эта утилита назначила `3267` в качестве моих UID и GID, определенных в /etc/passwd и /etc/group. Она также выделила для меня диапазон `100000-1065535` — это дополнительные UID и GID, определенные в /etc/subuid и /etc/subgid. Посмотрим содержимое этих файлов:

```
$ cat /etc/subuid
dwalsh:100000:65536
Testuser:165536:65536
$ cat /etc/subgid
dwalsh:100000:65536
Testuser:165536:65536
```

Если вы выведете содержимое этих файлов в своей системе, то увидите нечто похожее. В моей системе также есть учетная запись `testuser`. Утилита `useradd` добавила диапазон UID/GID для этого пользователя, который начинается сразу после диапазона пользователя `dwalsh`.

В пространстве имен пользователя у меня есть доступ к UID `3267` (мой UID), а также к `100000`, `100001`, `100002`, ..., `165535` — всего к `65 537` UID. Пользователь root может изменить файлы /etc/subuid и /etc/subgid, чтобы увеличить или уменьшить это число.

Команда `useradd` стартует с UID `100000`, что позволяет иметь около 99 000 обычных пользователей плюс 1000 UID, зарезервированных для системных служб в системе Linux. Ядро поддерживает более 4 миллиардов UID (2^{32} = 4 294 967 296). Поскольку `useradd` выделяет 65 537 UID для каждого пользователя, Linux может поддерживать более 60 000 пользователей. Число 65 536 (2^{16}) было выбрано потому, что до ядра Linux 2.4 это было максимальным количеством пользователей в системе.

Теперь рассмотрим подробнее пространство имен пользователя.

Каждый процесс в Linux, включая init и systemd, находится в каком-либо пространстве имен. Соответственно, каждый процесс принадлежит также и одному из пользовательских пространств имен. Сопоставление пользовательского пространства имен с вашим процессом можно найти, изучив содержимое файловой системы /proc. Файлы /proc/PID/uid_map и /proc/PID/gid_map содержат сопоставления пространств имен пользователей для каждого процесса в ОС. /proc/self/uid_map содержит соответствие UID текущего процесса:

```
$ cat /proc/self/uid_map
    0       0 4294967295
```

Это означает, что UIDы, начинающиеся с UID `0`, сопоставляются с UID `0` для диапазона из 4 294 967 295 UID.

Можно взглянуть на это отображение другим способом:

```
UID 0->0, 1->1, ...3267->3267, ..., 4294967294->4294967294.
```

Разницы при сопоставлении, в сущности, нет — root есть root. И мой UID `3267` сопоставлен с `3267`.

Давайте войдем в это пользовательское пространство имен и посмотрим, что в нем отображается. В Podman есть специальная команда `podman unshare`, которая позволяет войти в пространство имен пользователя без запуска контейнера. Таким образом вы исследуете, что происходит в пространстве имен пользователя, продолжая выполнять обычный процесс в вашей системе.

Следующей командой я запускаю `podman unshare` и вывожу `cat /proc/self/uid_map` в пространстве имен пользователя по умолчанию для своей учетной записи:

```
$ podman unshare cat /proc/self/uid_map
       0     3267        1
       1   100000    65536
```

Видно, что UID `0` сопоставляется с UID `3267` (мой UID) для диапазона `1`. Далее UID `1` сопоставляется с UID `100000` для диапазона из `65536` UID.

Любой UID, не сопоставленный с пространством имен пользователя, определяется в этом пространстве имен как пользователь `nobody`. Вы сталкивались с этим ранее, когда искали UID внутри образа контейнера:

```
$ podman run --user=root --rm quay.io/rhatdan/myimage -- bash -c "find /
➥ -mount -exec stat -c %u=%U {} \; | sort -un" 2>/dev/null
0=root
48=apache
1001=default
65534=nobody
```

Посмотрите на корневой каталог / на хосте — вы увидите, что он принадлежит настоящему root:

```
$ ls -l -ld /
dr-xr-xr-x. 18 root root 242 Sep 21 22:32 /
```

Если вы проверите тот же каталог в пространстве имен пользователя, окажется, что он принадлежит пользователю `nobody`:

```
$ podman unshare ls -ld /
dr-xr-xr-x. 18 nobody nobody 242 Sep 21 22:32 /
```

Поскольку UID хоста `0` не сопоставлен с пользовательским пространством имен, ядро сообщает об этом UID как о пользователе `nobody`. Процессы в этом пространстве имен имеют доступ только к файлам пользователя `nobody` на основе разрешений «для всех» (`other` или `world`). В следующем примере вы запустите сценарий bash — он показывает, что пользователь является root в этом пространстве имен, но при этом /etc/passwd принадлежит пользователю `nobody`. С помощью команды grep вы можете прочитать файл, так как /etc/passwd доступен для чтения всем. Но команда touch не работает, потому что даже root не может изменять файлы, принадлежащие UID, который не сопоставлен с этим пространством имен:

```
$ podman unshare bash -c "id ; ls -l /etc/passwd; grep dwalsh
➥ /etc/passwd; touch /etc/passwd"
uid=0(root) gid=0(root) groups=0(root),65534(nobody)
-rw-r--r--. 1 nobody nobody 2942 Sep 28 07:08 /etc/passwd
dwalsh:x:3267:3267:Dan Walsh:/home/dwalsh:/bin/bash
touch: cannot touch '/etc/passwd': Permission denied
```

Посмотрите на свой домашний каталог на хосте и сравните его с тем же каталогом в пользовательском пространстве имен. Вы увидите, что на хосте файлы принадлежат вашему UID:

```
$ ls -ld /home/dwalsh
drwx------. 365 dwalsh dwalsh 24576 Sep 28 07:30 /home/dwalsh
```

В пространстве имен они принадлежат пользователю root:

```
$ podman unshare ls -ld /home/dwalsh
drwx------. 365 root root 24576 Sep 28 07:30 /home/dwalsh
```

По умолчанию Podman сопоставляет ваш UID с root в этом пространстве имен. Podman использует root, потому что, как я указал в начале этой главы, большинство образов контейнеров предполагают, что они запускаются с правами root.

Рассмотрим последний пример. Находясь в пространстве имен пользователя, создайте каталог и файл в этом каталоге. Используйте команду `chown`, чтобы изменить UID содержимого на `1:1`:

```
$ podman unshare bash -c "mkdir test;touch test/testfile; chown -R 1:1 test"
```

Вне пространства имен пользователя видно, что тестовый файл принадлежит UID `100000`:

```
$ ls -l test
total 0
-rw-r--r--. 1 100000 100000 0 Sep 28 07:53 testfile
```

Когда вы создаете тестовый файл и изменяете его UID/GID на `1:1` в пространстве имен пользователя, владельцем на диске фактически является UID `100000`/`100000`. Запомните, что в пространстве имен пользователя UID `1` сопоставляется с UID `100000`, поэтому при создании вами файла UID `1` в пространстве имен пользователя ОС на самом деле создает UID `100000`.

Если вы попытаетесь удалить файл за пределами пространства имен пользователя, то получите сообщение об ошибке:

```
$ rm -rf test
rm: cannot remove 'test/testfile': Permission denied
```

Вне пространства имен пользователя у вас есть доступ только к вашему UID; к дополнительным UID у вас доступа нет.

> **ПРИМЕЧАНИЕ** В разделе 3.1.2 я показал, как сопоставление пользовательских пространств имен с томами контейнера может вызывать определенные проблемы, и рассмотрел способы их решения.

Повторно войдя в пространство имен, вы можете удалить файл:

```
$ podman unshare rm -rf test
```

Надеюсь, вы начинаете проникаться пониманием того, как работает пользовательское пространство имен. Команда `podman unshare` упрощает изучение вашей системы внутри пользовательского пространства имен и позволяет понять, что происходит в rootless-контейнерах. При запуске контейнера без root Podman нужно не просто работать с правами root — ему также необходим доступ к некоторым особым возможностям root, называемым Linux capabilities[1].

Не все процессы root одинаково эффективны в Linux. Linux разбивает полномочия root на ряд Linux-привилегий. Корневой процесс со всеми привилегиями обладает всеми полномочиями, в то время как root-процесс без таких привилегий не может управлять большей частью системы. Он, например, не может читать не относящиеся к root файлы, если только эти файлы не имеют флагов, которые разрешают чтение всем UID в системе («доступно для чтения всем»).

Посмотрим, как привилегии работают с пространством имен пользователя:

```
$ man capabilities
...
DESCRIPTION
For the purpose of performing permission checks, traditional UNIX
implementations distinguish two categories of processes: privileged
processes (whose effective user ID is 0, referred to as superuser or root),
and unprivileged processes (whose effective UID is nonzero). Privileged
processes bypass all kernel permission checks, while unprivileged processes
are subject to full permission checking based on the process's credentials
(usually: effective UID, effective GID, and supplementary group list).
Starting with kernel 2.2, Linux divides the privileges traditionally
associated with superuser into distinct units, known as capabilities, which
can be independently enabled and disabled. Capabilities are a per-thread
attribute.
```

(Перевод с английского:

```
ОПИСАНИЕ
Для проверки разрешений в традиционных реализациях UNIX различают две
категории процессов: привилегированные процессы (чей действующий идентификатор
(effective UID) пользователя равен 0, называются суперпользователями или root)
и непривилегированные процессы (чей действующий идентификатор пользователя не
равен нулю). Привилегированные процессы обходят все проверки разрешений ядра,
в то время как непривилегированные процессы подлежат полной проверке разрешений
на основе учетных данных процесса (обычно это действующий UID, действующий GID
и вспомогательный список групп). Начиная с ядра 2.2, Linux делит привилегии,
традиционно связанные с суперпользователем, на отдельные единицы, известные
как capabilities, которые можно независимо включать и отключать. Capabilities
назначаются каждому потоку отдельно.)
```

[1] Далее будем называть их Linux-привилегии или просто привилегии. — *Примеч. пер.*

В настоящее время в Linux имеется около 40 привилегий. Среди них CAP_SETUID и CAP_SETGID, позволяющие процессам изменять свои UID и GID. CAP_NET_ADMIN дает возможность управлять сетевым стеком.

Еще одна привилегия, называемая CAP_CHOWN, позволяет процессам изменять UID/GID файлов на диске. В предыдущем примере, применив команду chown к каталогу test, вы использовали CAP_CHOWN в пользовательском пространстве имен:

```
$ podman unshare bash -c "mkdir test;touch test/testfile; chown -R 1:1 test"
```

Работая в пользовательском пространстве имен, вы используете привилегии пространства имен. Помимо определенных в пространстве имен UID и GID, пользователь root имеет свои привилегии в вашем пространстве имен. Процессам с привилегией пространства имен CAP_CHOWN разрешено менять владельца файлов, принадлежащих вашему пользовательскому пространству имен, на UID, также находящиеся в этом пространстве имен. Если процесс в пользовательском пространстве имен пытается получить доступ к файлу, который не сопоставлен с этим пространством и принадлежит пользователю nobody, он получит отказ. Подобным же образом будет отказано в доступе процессу, пытающемуся выполнить chown на файле с UID, который не определен в этом пространстве имен. CAP_SETUID также позволяет процессам изменять UID только на те, которые определены в пространстве имен пользователя.

Когда Podman запускает контейнер, ему необходимо смонтировать несколько файловых систем. В Linux для монтирования файловых систем требуется привилегия CAP_SYS_ADMIN. Монтирование файловых систем в Linux может быть связано с рисками для безопасности. Ядро включает дополнительные элементы управления тем, какие типы файловых систем могут быть смонтированы, и требует, чтобы процессы пользовательского пространства имен также находились в уникальном пространстве имен mount. В главе 10 вы узнаете, как Podman ограничивает количество Linux-привилегий, доступных пользователю root внутри контейнера.

6.1.3. Пространство имен mount (mount namespace)

Пространства имен mount позволяют процессам внутри них монтировать файловые системы, при этом точки монтирования не видны процессам вне текущего пространства имен. Внутри пространства имен mount вы можете смонтировать tmpfs в /tmp, что препятствует процессам в этом пространстве просматривать содержимое оригинального /tmp. За пределами этого пространства имен процессы по-прежнему видят исходное монтирование и файлы в /tmp, но не видят произведенное вами монтирование.

В непривилегированных контейнерах Podman нужно монтировать как содержимое образов контейнеров, так и /proc, /sys, устройства из /dev и некоторых

файловых систем `tmpfs`. Для этого Podman необходимо создать пространство имен mount:

```
$ man mount namespaces
...
Mount namespaces provide isolation of the list of mount points seen by the
processes in each namespace instance. Thus, the processes in each of the mount
namespace instances see distinct single-directory hierarchies.
```

(Перевод с английского:

```
Пространства имен mount обеспечивают изоляцию списка точек монтирования, видимых
процессами в каждом экземпляре пространства имен. Таким образом, процессы в каждом
из экземпляров пространства имен mount видят разные иерархии в одном каталоге.)
```

Когда вы выполняете команду `podman unshare`, вы фактически входите и в другое пространство имен mount, и в другое пользовательское пространство имен.

Вы можете изучить пространства имен процесса, получив список содержимого каталога /proc/self/ns/ следующим образом:

```
$ ls -l /proc/self/ns/user /proc/self/ns/mnt
lrwxrwxrwx. 1 dwalsh dwalsh 0 Sep 28 09:17 /proc/self/ns/mnt ->
➥ 'mnt:[4026531840]'
lrwxrwxrwx. 1 dwalsh dwalsh 0 Sep 28 09:17 /proc/self/ns/user ->
➥ 'user:[4026531837]'
```

Обратите внимание: при входе в пользовательское пространство имен и пространство имен mount идентификаторы меняются:

```
$ podman unshare ls -l /proc/self/ns/user /proc/self/ns/mnt
lrwxrwxrwx. 1 root root 0 Sep 28 09:17 /proc/self/ns/mnt ->
➥ 'mnt:[4026533087]'
lrwxrwxrwx. 1 root root 0 Sep 28 09:17 /proc/self/ns/user ->
➥ 'user:[4026533086]'
```

Для теста вы можете создать файл в /tmp, а затем попытаться привязать его к /etc/shadow. Вне пространств имен ядро правомерно запрещает вам монтировать файл, как показано в следующем выводе:

```
$ echo hello > /tmp/testfile
$ mount --bind /tmp/testfile /etc/shadow
mount: /etc/shadow: must be superuser to use mount.
```

Как только вы войдете в пространство имен пользователя и пространство имен mount, ваш процесс сможет успешно выполнить монтирование поверх файла /etc/shadow. Чтобы убедиться, что /etc/shadow действительно изменен, запустите команду:

```
$ podman unshare bash -c "mount -o bind /tmp/testfile /etc/shadow; cat
/etc/shadow"
hello
```

Как только вы выйдете из `unshare`, все вернется на круги своя.

6.1.4. Пользовательское пространство имен и пространство имен mount

Как вы видели ранее, при перемонтировании файла /etc/shadow можно обмануть некоторые `setuid`-приложения, например /bin/su или /bin/sudo, и получить права полного root. В предотвращении такого рода атак и заключается причина, по которой пользователям без права root не разрешается монтировать файловые системы.

Итак, отдельное пространство имен mount не позволяет вам влиять на то, как систему видит хост; все, что вы монтируете, видно только в пределах этого пространства имен. В пользовательском пространстве имен у контейнера уже есть root этого пространства имен. Атаки на ваши точки монтирования могут перейти на уровень root только в пределах текущего пространства имен, а не реального root на хосте. Контейнерные процессы не могут изменить свой UID (`setuid`) на реальный root или любой другой UID, не сопоставленный с текущим пространством имен.

Даже с учетом пространств имен ядро Linux позволяет монтировать только определенные типы файловых систем. Многие типы файловых систем слишком опасно предоставлять пользователям без полномочий root, потому что они получат доступ к важным частям ядра. Я работаю вместе с инженерами, занимающимися вопросами файловых систем в ядре Linux, мы стараемся понять, есть ли способы ограничить другие типы файловых систем так, чтобы их можно было монтировать в режиме rootless без влияния на безопасность системы.

Начиная с версии ядра 5.13, инженеры добавили собственные оверлейные монтирования в список разрешенных. Допустимые в настоящее время типы файловых систем перечислены в табл. 6.2.

Таблица 6.2. Типы монтирования файловой системы, поддерживаемые в настоящее время в режиме rootless

Тип монтирования	Описание
`bind`	Активно используется в непривилегированных контейнерах. Поскольку пользователям без полномочий root не разрешено создавать устройства, Podman с помощью `bind` mount привязывает /dev на хосте к контейнеру. Podman также использует `bind` монтирования, `bind` mount /dev/null поверх файлов в /proc и /sys, чтобы скрыть содержимое файловой системы хоста от контейнеров. Монтирование томов, описанное в главе 3, тоже использует монтирование типа `bind`

Таблица 6.2 (*окончание*)

Тип монтирования	Описание
`binderfs`	Файловая система для механизма IPC binder Android. Не поддерживается Podman
`devpts`	Виртуальная файловая система, смонтированная в /dev/pts. Содержит файлы устройств, используемые для эмуляторов терминала
`cgroupfs`	Файловая система ядра, используемая для управления cgroups. Rootless-контейнеры могут использовать `cgroupfs` для управления cgroups в cgroups v2. В v1 не поддерживается. Смонтирован в /sys/fs/cgroups
`FUSE`	Используется для монтирования образов контейнеров с помощью `fuse-overlayfs` в режиме rootless. До ядра 5.13 это был единственный способ использовать оверлейную файловую систему в режиме без root
`procfs`	Монтируется в /proc внутри контейнера. Дает возможность исследовать процессы внутри контейнера
`mqueue`	Реализует API очередей сообщений POSIX. Podman монтирует эту файловую систему в /dev/mqueue
`overlayfs`	Используется для монтирования образа. Лучше работает в файловой системе `fuse-overlayfs`. В некоторых случаях дает преимущества по сравнению с нативным оверлеем, как, например, домашние каталоги NFS
`ramfs`	Файловая система Linux на основе оперативной памяти с динамически изменяемым размером, в настоящее время не используется с Podman
`sysfs`	Монтируется в /sys
`tmpfs`	Используется, чтобы скрыть каталоги файловой системы ядра /proc и /sys от контейнеров

6.2. «ПОД КАПОТОМ» PODMAN БЕЗ ROOT

Вы уже получили некоторое представление о том, зачем нужны и как работают пользовательское пространство имен и пространство имен mount. Теперь давайте углубимся в изучение того, что делает Podman при запуске контейнера. При первом запуске контейнера и входе в систему Podman читает файлы /etc/subuid и /etc/subgid в поисках вашего имени пользователя или UID. Как только Podman находит запись, он использует ее содержимое, а также ваш текущий UID/GID, чтобы сгенерировать для вас пространство имен пользователя. Затем Podman запускает процесс `podman pause`, чтобы удерживать открытыми пространства имен пользователя и mount (рис. 6.4).

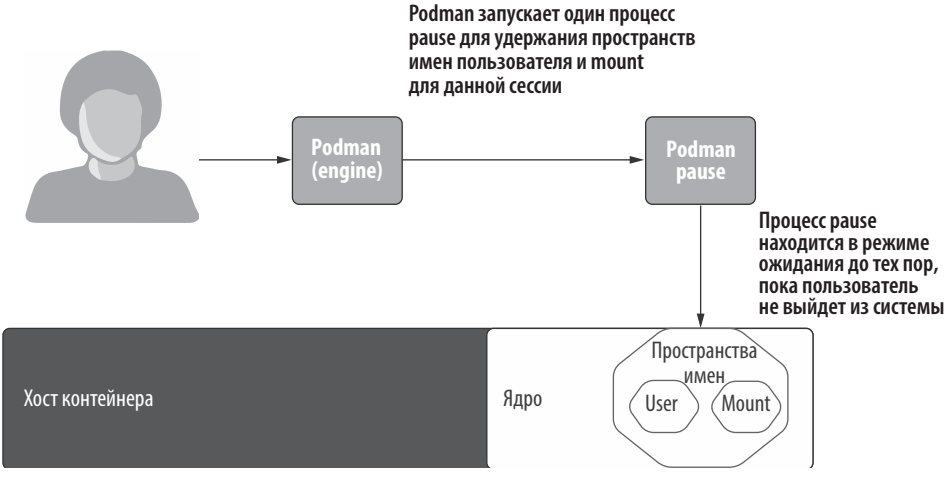


Рис. 6.4. Podman запускает процесс pause, чтобы держать открытыми пространства имен пользователя и mount

От пользователей часто можно услышать, что после запуска контейнеров Podman они видят процесс `podman` все еще работающим, когда выполняют следующую команду:

```
$ ps -e | grep podman
2541 ? 00:00:00 podman pause
```

При последовательном выполнении команд Podman присоединяется к пространствам имен процесса `podman pause`. Это необходимо, чтобы избежать состояния гонки (race conditions), при котором пользовательские пространства имен появляются и исчезают. Процесс `pause` остается работающим до тех пор, пока вы не выйдете из системы. Чтобы удалить этот процесс, можно также выполнить команду `podman system migrate`. Роль процесса `pause` состоит в поддержании активности пользовательского пространства имен, поскольку все rootless-контейнеры должны запускаться в одном и том же пространстве имен. В противном случае совместное использование содержимого и других пространств имен (например, совместное использование сетевого пространства имен из другого контейнера) было бы невозможным.

ПРИМЕЧАНИЕ Пользователи часто сообщают мне, что при изменении файлов /etc/subuid и /etc/subgid их контейнеры не сразу отражают эти изменения. Поскольку процесс pause был запущен с настройками предыдущего пространства имен, его необходимо удалить. Выполнение команды `podman system migrate` перезапускает процесс pause в пространстве имен пользователя.

Вы можете завершить процесс `pause` в любое время, но Podman пересоздаст его при следующем запуске. По умолчанию у каждого пользователя без полномочий root есть собственное пользовательское пространство имен, и все их контейнеры работают в одном и том же пространстве имен. Это пространство имен можно разделить и запускать контейнеры с разными пространствами имен, но имейте в виду, что для этого по умолчанию у вас есть только 65 000 UID. Запуск нескольких контейнеров в разных пользовательских пространствах имен намного проще при работе с привилегированными (rootful) контейнерами. Когда пользовательское пространство имен и пространство имен mount созданы, Podman создает хранилище для образа контейнера и точку монтирования для хранения этого образа.

6.2.1. Скачивание образа

При скачивании образа (рис. 6.5) Podman проверяет, существует ли образ контейнера quay.io/rhatdan/myimage в локальном хранилище контейнеров. Если он уже существует, Podman переходит к настройке сети контейнера (раздел 6.2.3). Если же образ контейнера отсутствует в локальном хранилище, Podman использует библиотеку containers/image для его скачивания. Скачивая образ, Podman предпринимает следующие шаги:

- Разрешает IP-адрес для реестра: quay.io.
- Подключается к этому IP-адресу через порт HTTPS (443).
- Начинает загружать манифест, все слои и конфигурацию образа, используя протокол HTTP.
- Находит слои, или блобы (blobs), образа quay.io/rhatdan/myimage.
- Параллельно копирует все слои из реестра контейнеров на хост.

Когда все слои скопированы на хост, Podman использует библиотеку containers/storage для повторной сборки слоев по порядку, создавая оверлейную точку монтирования для каждого из них поверх предыдущего в ~/.local/share/containers/storage. Если предыдущего слоя нет, создается начальный слой.

Затем containers/storage распаковывает содержимое слоя в новый слой хранилища. Как только слои разархивированы, containers/storage меняет UID/GID файлов архива на те же, что и у домашнего каталога пользователя. В предыдущих разделах я объяснял, что Podman использует преимущества пользовательского пространства имен `CAP_CHOWN`. Не забывайте, что Podman не удается создать содержимое, если UID или GID, указанные в файле TAR, не были сопоставлены с пространством имен пользователя.

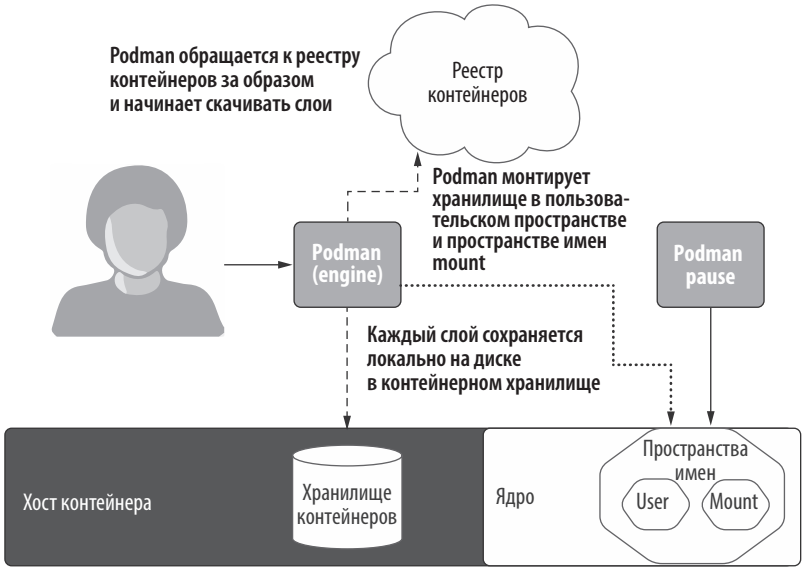


Рис. 6.5. Podman загружает образ из реестра контейнеров и сохраняет его в хранилище контейнеров

6.2.2. Создание контейнера

Как только библиотека containers/storage завершит загрузку образа и создание хранилища, Podman создает новый контейнер на основе этого образа и добавляет его в свою внутреннюю базу данных. Затем Podman дает containers/storage указание создать доступное для записи пространство на диске и использовать драйвер хранилища по умолчанию (обычно `overlayfs`) для монтирования этого пространства в качестве нового слоя контейнера. Новый слой контейнера действует как последний слой чтения/записи и монтируется поверх образа.

> **ПРИМЕЧАНИЕ** Контейнеры rootful по умолчанию используют нативное оверлейное монтирование Linux. В режиме rootless версии ядра новее 5.13 или с бэкпортированной rootless-функцией оверлея (ядра RHEL 8.5 или более поздней версии также имеют эту функцию) используют нативные оверлейные монтирования. В старых ядрах Podman использует `fuse-overlayfs` для создания слоя. В Podman `overlay` и `overlay2` — это один и тот же драйвер.

Теперь Podman необходимо настроить сеть внутри пространства имен network.

6.2.3. Настройка сети

При работе с Podman в rootless-режиме вам не удастся создать полноценную отдельную сеть для контейнеров, потому что процессам без root не разрешено организовывать сетевые устройства и изменять правила файервола. Непривилегированный Podman в режиме rootless использует slirp4netns (https://github.com/rootless-containers/slirp4netns) для настройки сети хоста и имитации VPN для контейнера. Slirp4netns обеспечивает работу сети в пользовательском режиме (slirp) для непривилегированных сетевых пространств имен (рис. 6.6).

> **ПРИМЕЧАНИЕ** В rootful-контейнерах Podman использует подключаемые модули CNI для настройки сетевых устройств. В режиме rootless, даже если пользователю разрешено создавать сетевое пространство имен и присоединяться к нему, он не может создавать сетевые устройства. Программа slirp4netns эмулирует виртуальную сеть для подключения сети хоста к сети контейнера. Для более сложных сетевых настроек требуются привилегированные контейнеры с root.

В нашем исходном примере вы указали сопоставление портов `8080:8080` следующим образом:

```
$ podman run -d -p 8080:8080 --name myapp
     registry.access.redhat.com/ubi8/httpd-24
```

Podman настраивает slirp4netns для прослушивания порта `8080` в сети хоста и позволяет процессу контейнера подключиться к этому порту. Команда slirp4netns

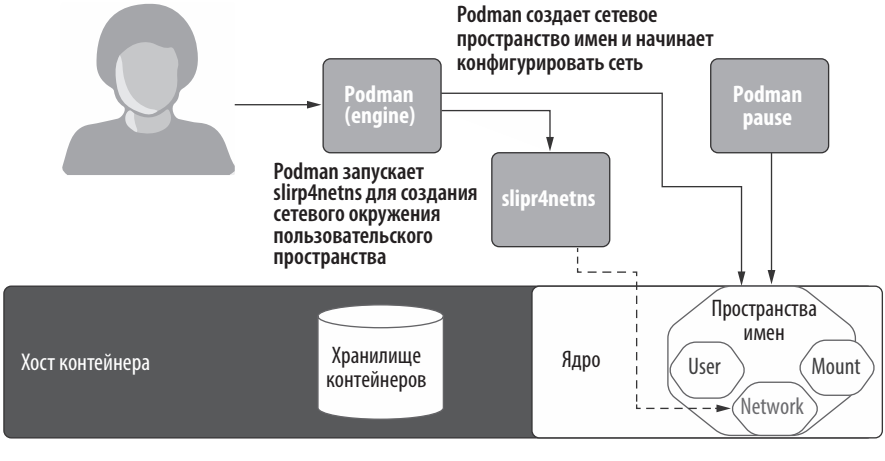


Рис. 6.6. Podman создает сетевое пространство имен и запускает slirp4netns для ретрансляции сетевых подключений

создает TAP-устройство, которое вводится в новое сетевое пространство имен, где находится контейнер. Каждый пакет считывается из slirp4netns и эмулирует стек TCP/IP в пространстве пользователя. За пределами пространства имен сети контейнера каждое соединение преобразуется в операцию сокета, которую непривилегированный пользователь может выполнять в пространстве имен сети хоста.

> **ПРИМЕЧАНИЕ** TAP-устройства Linux создают сетевой мост пространства пользователя. В пользовательском пространстве устройства TAP могут эмулировать сетевые устройства внутри сетевого пространства имен. Процессы внутри данного пространства имен взаимодействуют с этими сетевыми устройствами. Пакеты, считанные с сетевого устройства или записанные на него, направляются через устройство TUN/TAP в программу пользовательского пространства slirp4netns.

Хранилище и сеть настроены, и теперь Podman готов наконец запустить процесс контейнера.

6.2.4. Запуск мониторинга контейнера: conmon

Podman запускает conmon (мониторинг контейнера) для нашего контейнера, сообщая ему, что нужно использовать настроенную среду выполнения OCI — обычно это `crun` или `runc`. Он также выполняет команду `podman container cleanup $CTRID` при выходе из контейнера (рис. 6.7). conmon описан в разделе 4.1.

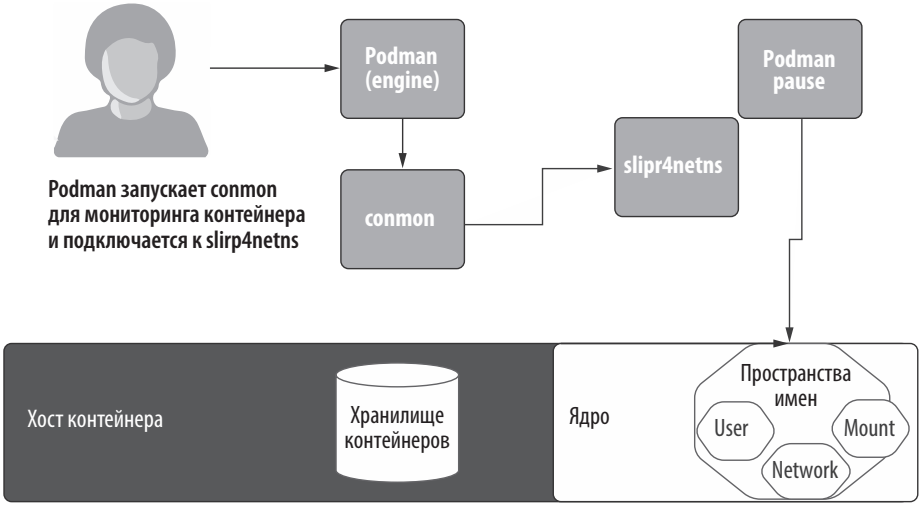


Рис. 6.7. Podman запускает мониторинг контейнера и подключается к slirp4netns

6.2.5. Запуск среды выполнения OCI

Среда выполнения OCI считывает spec-файл OCI и настраивает ядро для запуска контейнера (рис. 6.8). Среда выполнения OCI:

- Настраивает дополнительные пространства имен для контейнера.
- Настраивает cgroups v2 (cgroups v1 не поддерживается для rootless-контейнеров).
- Настраивает контекст безопасности (label) SELinux для запуска контейнера.
- Загружает правила seccomp из /usr/share/containers/seccomp.json в ядро.
- Задает переменные окружения для контейнера.
- Монтирует тома к путям в rootfs.

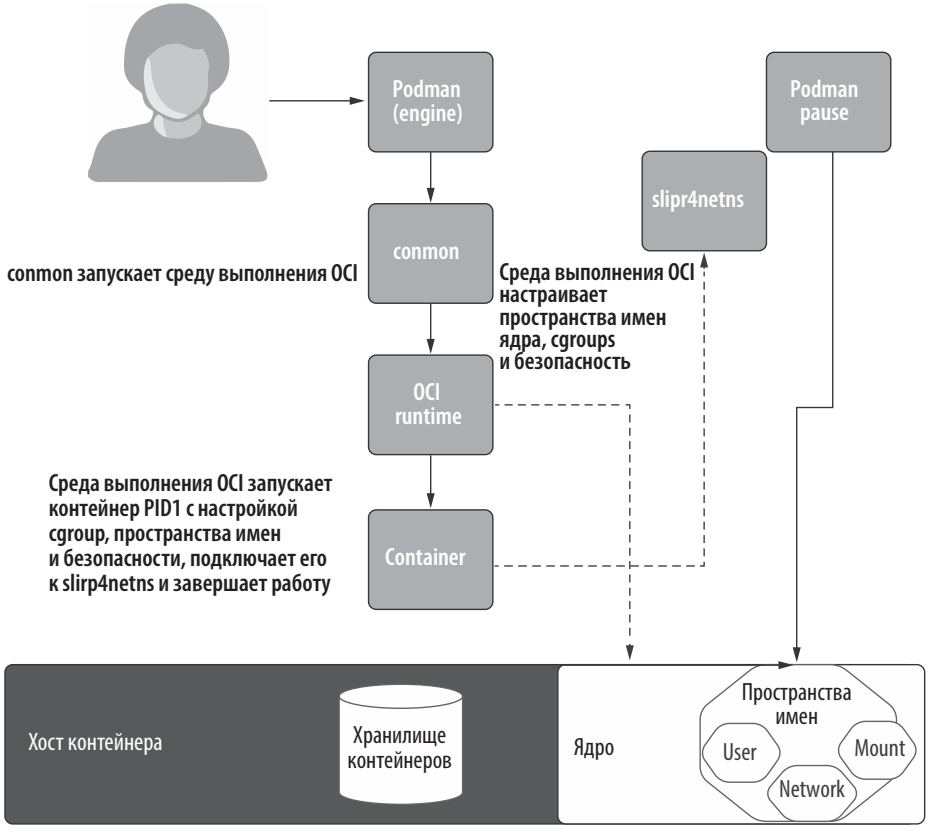


Рис. 6.8. conmon запускает среду выполнения OCI, которая настраивает ядро

- Переключает текущий / на / в rootfs.
- Создает дочерний процесс контейнера.
- Запускает OCI-хуки, передав им rootfs и PID контейнера 1.
- Выполняет команду, указанную в образе.
- Завершает работу среды выполнения OCI, оставив conmon для наблюдения за контейнером.

Наконец, conmon рапортует Podman об успехе (рис. 6.9).

Теперь команда Podman завершает работу, потому что она выполнялась в режиме `--detach (-d)`.

```
$ podman run -d -p 8080:8080 --name myapp
     registry.access.redhat.com/ubi8/httpd-24
```

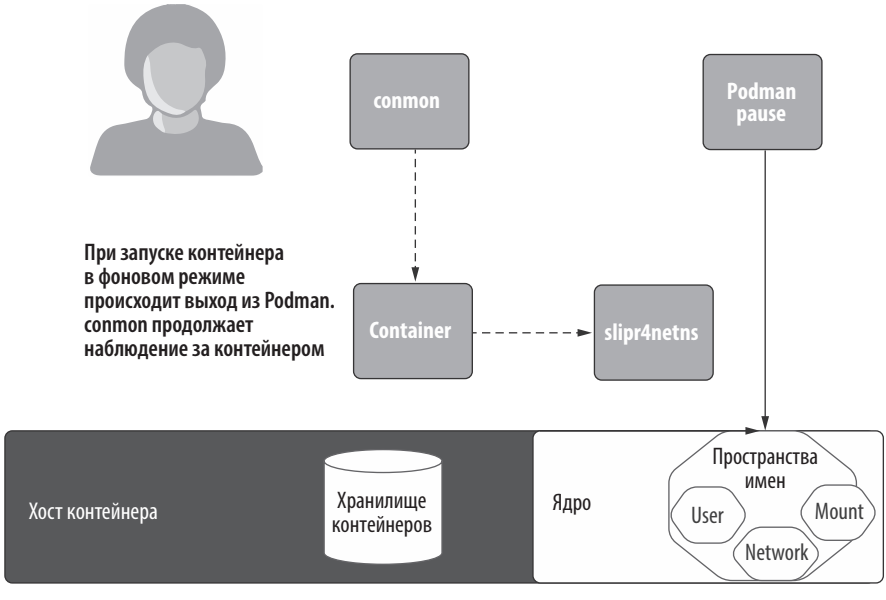


Рис. 6.9. Выход из Podman и среды выполнения OCI; при этом контейнер остается работающим, conmon отслеживает его, а slirp4netns обеспечивает работу сети

ПРИМЕЧАНИЕ Если позже вам понадобится, чтобы Podman взаимодействовал с работающим в фоновом режиме (detached) контейнером, используйте команду `podman attach`, которая подключается к сокету conmon. Это позволяет Podman взаимодействовать с процессом контейнера через файловые дескрипторы `STDIN`, `STDOUT` и `STDERR`, которые conmon отслеживает.

6.2.6. Контейнерное приложение работает до своего завершения

Процесс приложения завершается сам по себе, либо вы можете остановить контейнер, выполнив команду `podman stop`:

```
$ podman stop myapp
```

Когда процесс контейнера завершается, ядро отправляет сигнал `SIGCHLD` процессу `conmon`. В свою очередь conmon:

- Записывает код завершения контейнера.
- Закрывает лог-файл контейнера.
- Закрывает `STDOUT/STDERR` команд Podman.
- Выполняет команду `podman container cleanup $CTRID`.
- Завершается сам.

Команда `podman container cleanup` отключает сеть slirp4netns и размонтировывает все точки монтирования контейнера. Если вы укажете параметр `--rm`, контейнер будет полностью удален — слои будут убраны из containers/storage, а описание контейнера удалено из базы данных.

РЕЗЮМЕ

- Запуск rootless-контейнеров безопаснее, чем запуск контейнеров с полномочиями root.
- Пользовательское пространство имен дает обычным пользователям возможность манипулировать более чем одним UID и является ключом к запуску контейнеров.
- Пространство имен mount позволяет Podman монтировать файловые системы в пространстве имен пользователя.
- Podman использует slirp4netns для предоставления сетевого доступа к контейнерам.
- Podman запускает процесс `conmon` для мониторинга контейнера.

Часть 3

Расширенные возможности Podman

В части 3 книги вы узнаете о дополнительных возможностях использования Podman. Здесь рассматривается интеграция Podman с вашей системой, а также взаимодействие Podman с другими инструментами и оркестраторами.

В главе 7 я рассказываю об интеграции с systemd. Podman был разработан для полной интеграции с этой системой и использует преимущества init-системы systemd. Systemd несложно запускать в контейнерах Podman, и в главе будет показано, как это делать. Подобным же образом можно запускать Podman в рамках сервисов systemd. В нем предусмотрены команды, позволяющие автоматически создавать для этого конфигурационные файлы сервисов.

В главе 8 показано, как Podman работает с Kubernetes. Podman не является контейнерным движком для Kubernetes, но может работать с Kubernetes YAML-файлами. Поскольку эти YAML-файлы используются для описания выполняющихся в Kubernetes приложений, Podman позволяет легко перемещать приложения в полностью оркестрированную среду и обратно на одиночный хост. Такая функция облегчает разработку приложений, которые в перспективе будут работать под Kubernetes, а также отладку возникающих под Kubernetes проблем при помощи локального запуска этих приложений на ноутбуке. Kubernetes YAML является отличной альтернативой `docker-compose` YAML при запуске группы контейнеров на одной ноде.

В главе 9 представлена концепция Podman как службы, которая позволяет написанным с использованием RESTful API инструментам генерировать поды и контейнеры и управлять ими с помощью Podman. Такие инструменты, как docker-compose и другие средства Python, построенные на базе docker-py, могут взаимодействовать с сервисом Podman, что полностью исключает необходимость использования Docker. Служба Podman даже позволяет Podman, запущенному на удаленных системах, таких как Windows, macOS и Linux, работать с контейнерами, запущенными на Linux.

7 Интеграция с systemd

В ЭТОЙ ГЛАВЕ

- ✓ Запуск systemd внутри контейнера в качестве основного процесса
- ✓ Генерация файлов systemd-юнитов из существующих контейнеров
- ✓ Контейнерные сервисы с активацией через сокеты
- ✓ Использование контейнерных сервисов типа `sd-notify`
- ✓ Преимущества использования journald в качестве драйвера логирования и бэкенда для журнала событий
- ✓ Использование Podman и systemd для управления жизненным циклом контейнерных сервисов на граничных устройствах

Systemd де-факто является системой инициализации в Linux. Практически во всех дистрибутивах Linux по умолчанию первым процессом после ядра запускается systemd, который затем уже запускает все службы, включая сессии входа в систему текущего пользователя. Podman использует возможности systemd для запуска многих своих сервисов. При запуске контейнерных сервисов во время загрузки Podman рекомендует использовать юнит-файлы systemd вместе

с командами Podman. Юнит-файлы — это название конфигурационных файлов в systemd. Systemd поддерживает несколько различных типов юнит-файлов, в том числе сервисные файлы, определяющие службу, которой будет управлять systemd. SystemD.socket — еще один тип юнит-файла в systemd (раздел 7.6). Юнит-файлы служб systemd — это некий способ поделиться с миром своим контейнерным сервисом. Как показано на рис. 7.1, модель fork/exec в Podman предоставляет systemd возможность отслеживать процессы внутри контейнерного сервиса.

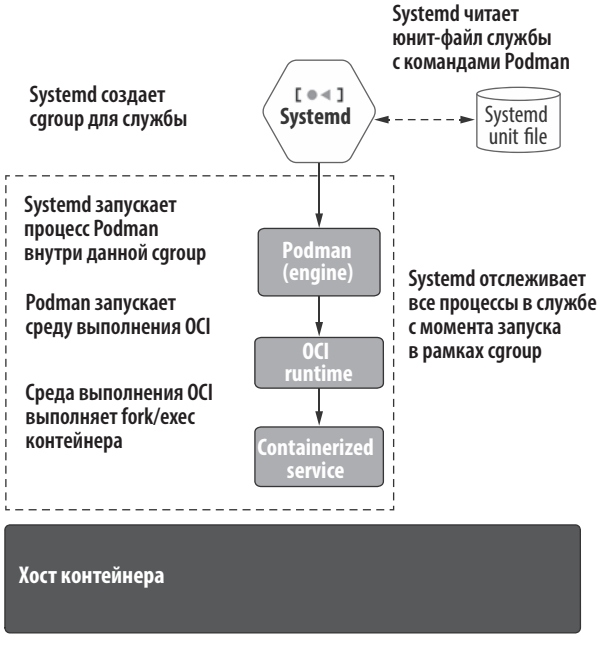


Рис. 7.1. Запуск контейнера Podman через systemd

Systemd помещает все процессы в рамках сервисного юнит-файла (называемого scope) в одну иерархию cgroup. Затем он использует PID cgroup для отслеживания всех процессов, а также использует эту информацию для управления службой. Контейнерные движки, основанные на модели клиент — сервер, не позволяют systemd отслеживать процессы в контейнере.

В этой главе вы узнаете, что Podman также использует преимущества других служб для автозапуска контейнеров, автообновления и базового управления контейнерными сервисами. Вы познакомитесь со многими возможностями Podman и systemd, но сначала вы запустите systemd в контейнере Podman.

7.1. ЗАПУСК SYSTEMD ВНУТРИ КОНТЕЙНЕРА

Когда контейнеризация только начинала набирать популярность, многие ее поборники пропагандировали концепцию микросервисов. *Микросервис* — это один специализированный сервис внутри контейнера. Этот единственный сервис запускается как начальный PID (PID 1) внутри контейнеров и пишет свои логи непосредственно в stdout и stderr. Kubernetes допускает использование микросервисов и поэтому собирает журналы из stdin/stderr запускаемых контейнеров. На рис. 7.2 показан Podman, работающий с микросервисами.

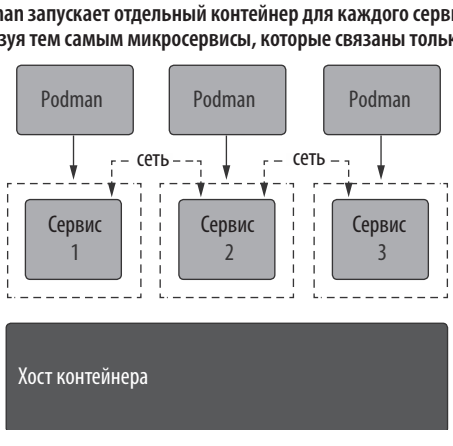


Рис. 7.2. Podman, управляющий тремя микросервисами

Альтернативный замысел состоял в том, чтобы запустить systemd в качестве начального PID в контейнере, а затем позволить systemd запустить одну или несколько служб внутри этого контейнера. Представители этой точки зрения утверждают, что контейнерные сервисы должны запускаться таким же образом, как они запускаются на виртуальной машине. Поскольку разработчики сервисных пакетов (например, RPM и APT) разрабатывают файлы модулей systemd как оптимальный способ запуска своих сервисов в ОС, разработчикам контейнеров также следует воспользоваться этими юнит-файлами. Такой подход позволяет запускать несколько сервисов в одном контейнере, использовать преимущества локальных путей передачи данных и ускорять процесс преобразования больших мультисервисных приложений в контейнер, а затем, со временем, разбивать каждый сервис на свои микросервисы.

Наконец, огромным преимуществом systemd в контейнере является то, что init-система справляется с удалением зомби-процесса. В Linux при завершении процесса ядро посылает родительскому процессу сигнал SIGCHLD, и родительский процесс должен получить код возврата завершающегося процесса. Ядро удаляет процесс из системы, когда родительский процесс считывает его код возврата. Если

родительский процесс не считывает код возврата, то завершенный процесс остается в статусе exited и называется *зомби-процессом*. Init-система systemd подчищает большинство процессов в системе. В контейнерах подчищать эти процессы должен начальный процесс, запущенный внутри этого контейнера. Иногда процессы контейнера завершаются, но если PID1 не удаляет их, то они никуда не исчезают.

ПРИМЕЧАНИЕ Команда `podman-run` поддерживает опцию `-init`, которая запускает маленькую init-программу только для того, чтобы подчищать зомби-процессы.

Podman был разработан для поддержки обоих методов — как микросервисов, так и мультисервисных контейнеров. На рис. 7.3 показан systemd, запускающий мультисервисное приложение в контейнере.

Podman проверяет параметр cmd контейнера на наличие там «init» или «systemd». После этого контейнер автоматически запускается в режиме systemd.

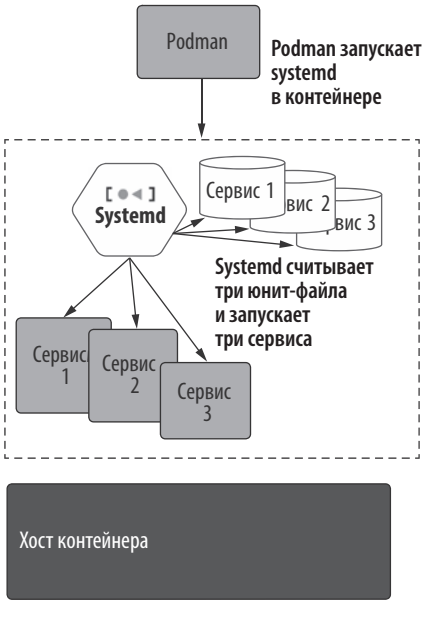


Рис. 7.3. Podman, запускающий systemd в контейнере с тремя сервисами

В следующем списке приведены все команды, заставляющие Podman запускаться в режиме systemd:

- /sbin/init;
- /usr/sbin/init;

- /usr/local/sbin/init;
- /*/systemd (любой путь, оканчивающийся командой systemd).

Образ registry.access.redhat.com/ubi8-init является примером образа, предназначенного для работы в режиме systemd.

Разверните образ ubi8-init и исследуйте команду:

```
$ podman pull ubi8-init
Resolved "ubi8-init" as an alias (/etc/containers/registries.conf.d/
➥ 000-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8-init:latest...
...
8cb83279f877a4bf3412827bf71c53188c3983194bd4663a1fc1378360844463
$ podman inspect ubi8-init --format '{{ .Config.Cmd }}'
[/sbin/init]
```

Systemd требует, чтобы окружение было сконфигурировано определенным образом; в противном случае systemd пытается исправить это окружение. В следующем разделе объясняется, как Podman обеспечивает выполнение требований systemd.

7.1.1. Требования к systemd в контейнерах

Systemd делает некоторые предположения об окружении, в котором он запускается, например, что /run и /tmp должны быть смонтированы в tmpfs. Если окружение неверно, systemd пытается исправить это, монтируя /run и /tmp в tmpfs. Для монтирования требуется привилегия `CAP_SYS_ADMIN` внутри контейнера, что недопустимо в непривилегированных контейнерах. В результате systemd просто падает.

Для решения этой проблемы Podman анализирует ENTRYPOINT и `CMD` образа контейнера на предмет запуска systemd, а затем изменяет окружение контейнера так, чтобы оно соответствовало ожиданиям systemd. Когда systemd видит эти монтирования, он пропускает их, что позволяет systemd работать в ограниченном окружении. В табл. 7.1 описаны требования, выдвигаемые systemd и обеспечиваемые Podman для успешной работы в непривилегированном контейнере.

Таблица 7.1. Требования systemd для работы в непривилегированном контейнере

Требования systemd	Описание
/run в tmpfs	Systemd требует, чтобы /run была смонтирована в tmpfs. Если /run не смонтирован в tmpfs, systemd попытается это сделать. По умолчанию ограниченный контейнер не позволит такое монтирование, поэтому systemd потерпит неудачу

Требования systemd	Описание
/tmp в tmpfs	Аналогично /run, systemd попытается смонтировать tmpfs в /tmp, если там еще нет смонтированного каталога
/var/log/journald в виде tmpfs	Systemd внутри контейнера ожидает, что сможет писать в /var/log/journald, поэтому Podman монтирует tmpfs, чтобы сделать это возможным
Переменная окружения `container`	Systemd определяет, что переменная окружения `container` установлена, чтобы изменить некоторые из своих действий по умолчанию и улучшить свою работу в контейнере
`STOPSIGNAL=SIGRTMIN+3`	В отличие от большинства процессов в системе, systemd игнорирует `SIGTERM` и корректно завершает работу с ним только при получении сигнала `SIGRTMIN+3 (37)`

7.1.2. Контейнер Podman в режиме systemd

С помощью флага `--systemd=always` можно исследовать окружение контейнера, работающего в режиме systemd. Для начала запустите контейнер с включенным режимом systemd с помощью флага `--systemd=always`. Этот параметр запускает контейнер в режиме systemd даже тогда, когда systemd не запущен, что облегчает отладку окружения. В этот момент можно выполнить systemd и запустить его с PID1:

```
$ podman create –rm –name SystemD -ti –systemd=always ubi8-init sh
774a50204204768edd73f178b6afdf975cf9353e3b90af9df77273d639f60ac3
```

С помощью `podman inspect` можно узнать `StopSignal` контейнера; Podman установил его равным `37 (SIGRTMIN+3)`:

```
$ podman inspect SystemD --format '{{ .Config.StopSignal}}'
37
```

Теперь запустите контейнер; посмотрев на монтирование /run и /tmp, вы увидите, что оба монтируются в tmpfs. Наконец, проверьте, установлена ли переменная окружения container:

```
$ podman start --attach SystemD
# mount | grep -e /tmp -e /run | head -2
tmpfs on /tmp type tmpfs
➥ (rw,nosuid,nodev,relatime,context="system_u:object_r:container_file_t:s0:
➥ c37,c965",uid=3267,gid=3267,inode64)
tmpfs on /run type tmpfs
➥ (rw,nosuid,nodev,relatime,context="system_u:object_r:container_file_t:s
➥ 0:c37,c965",uid=3267,gid=3267,inode64)
# printenv container
Oci
```

Если просто запустить контейнер на основе образа `ubi8-init`, вы увидите запущенный systemd:

```
$ podman run -ti ubi8-init
SystemD 239 (239-45.el8_4.3) running in system mode. (+PAM +AUDIT +SELINUX
➥ +IMA -APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +GNUTLS
➥ +ACL +XZ +LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD +IDN2 -IDN +PCRE2
➥ default-hierarchy=legacy)
Detected virtualization container-other.
Detected architecture x86-64.
Welcome to Red Hat Enterprise Linux 8.4 (Ootpa)!
Set hostname to <26bbf9077219>.
Initializing machine ID from random generator.
Failed to read AF_UNIX datagram queue length, ignoring:
➥ No such file or directory
[ OK ] Listening on initctl Compatibility Named Pipe.
[ OK ] Reached target Swap.
[ OK ] Listening on Journal Socket (/dev/log).
[ OK ] Listening on Journal Socket.
...
```

Отмечу, что systemd не реагирует на нажатие Ctrl-C и не отправляет сигнал остановки SYGTERM. Поэтому чтобы остановить этот контейнер, необходимо перейти в другой терминал и выполнить команду:

```
# podman stop -l
```

Это заставит Podman послать соответствующий сигнал `STOPSIGNAL (SIGRTMIN+3)` systemd в контейнере. При получении этого сигнала systemd сразу выключится.

Вы разобрались, что требуется для systemd, и пришло время создать службу, которую systemd будет запускать. В следующем разделе вы создадите на основе systemd службу Apache, которая будет работать внутри контейнера.

7.1.3. Запуск службы Apache в systemd-контейнере

В этом разделе мы создадим Containerfile, использующий ubi8-init в качестве базового образа, а затем установим Apache `httpd`. После этого вы включите эту службу и настроите скрипт Apache, с которым работали.

Создайте Containerfile:

```
$ cat << _EOF > /tmp/Containerfile
FROM ubi8-init
RUN dnf -y install httpd; dnf -y clean all
RUN systemctl enable httpd.service
_EOF
```

Напомню, что FROM ubi8-init указывает Podman на использование образа ubi8-init в качестве базового для нового образа:

```
FROM ubi8-init
RUN dnf -y install httpd; dnf -y clean all
RUN systemctl enable httpd.service
```

Строка `RUN dnf -y install httpd; dnf -y clean all` дает указание Podman запустить контейнер, который выполнит команду `dnf` и установит пакет `httpd` поверх образа `ubi8-init`. Вторая команда `dnf` удаляет лишние файлы и логи, созданные `dnf` в процессе установки, поскольку нет смысла включать их в образ:

```
FROM ubi8-init
RUN dnf -y install httpd; dnf -y clean all
RUN systemctl enable httpd.service
```

Последняя команда `RUN systemctl enable httpd.service` дает Podman указание запустить другой сборочный контейнер и выполнить команду `systemctl` для активации службы `httpd.service`. Когда systemd заработает в контейнере, запущенном из вновь созданного образа, служба `httpd` активируется:

```
FROM ubi8-init
RUN dnf -y install httpd; dnf -y clean all
RUN systemctl enable httpd.service
```

Соберите образ с помощью `podman build` и назовите его `my-systemd`:

```
$ podman build -t my-systemd /tmp
STEP 1/3: FROM ubi8-init
STEP 2/3: RUN dnf -y install httpd; dnf -y clean all
Updating Subscription Management repositories.
Unable to read consumer identity
...
COMMIT my-systemd
--> 104fa99d9a2
Successfully tagged localhost/my-systemd:latest
104fa99d9a2138404039cf15b470ab04784cdaab2226f29bd8343f8e24ec60e2
```

Теперь запустите контейнер из этого образа с томом, смонтированным с хоста. Поскольку пакет Apache по умолчанию прослушивает порт `80`, используйте `--p 8080:80`, что, как вы узнали, сопоставит порт `8080` с портом `80` внутри контейнера. Используйте каталог html с файлом index.html из раздела 3.1:

```
$ podman run -d --rm -p 8080:80 -v ./html:/var/www/html:Z my-systemd
71f1678084390925b7488f68ab58cd55e16009d69b717045b8ed5ef14e8599ce
```

Вы смонтировали том (`-v ./html/:/var/www/html:Z`) в каталоге ./html с файлом goodbye world index.html:

```
$ podman run -d --rm -p 8080:80 -v ./html:/var/www/html:Z my-systemd
```

Запустите веб-браузер для проверки работоспособности контейнерного сервиса, как показано на рис. 7.4:

```
$ web-browser localhost:8080
```

Рис. 7.4. Окно веб-браузера демонстрирует запущенный образ контейнера на базе systemd с вашим содержимым

Обратите внимание, что при создании образа не нужно было специально управлять процессами сервера HTTPD. Контейнер работает с HTTPD так же, как и виртуальная машина. Если вам необходимо включить в образ другой сервис, это просто сделать, установив пакет и активировав его юнит-файл.

Выполнив команду podman logs, вы обнаружите один из недостатков такой установки:

```
$ podman logs 71f1678084
```

Вывод отсутствует. Поскольку systemd работает под PID1 контейнера, он не записывает вывод в логи. Необходимо войти в контейнер и использовать `journalctl` либо прочитать логи `httpd` в /var/log/httpd/error_log, чтобы узнать, были ли какие-либо проблемы.

Вы узнали, как использовать systemd внутри контейнера, и пришло время выяснить, как с помощью systemd и Podman получить преимущества расширенных возможностей systemd.

7.2. JOURNALD ДЛЯ ВЕДЕНИЯ ЖУРНАЛОВ И СОБЫТИЙ

Журнал systemd (journald) — современная система логирования в Linux. Это системная служба, которая собирает и хранит данные логирования. Преимуществами использования journald является постоянное хранение записей, а также встроенная функция ротации журнала. Podman по умолчанию использует journald для хранения логов.

7.2.1. Драйвер логирования

По умолчанию Podman использует journald в качестве драйвера логирования в системах, работающих с systemd в качестве init-системы. Если вы запускаете Podman в контейнере без systemd, он возвращается к использованию файлового драйвера. При выборе драйвера журнала следует обратить внимание на то, сохраняются ли данные журнала после удаления контейнера.

Вторым важным моментом при выборе драйвера является размер файла журнала. В журнал записывается весь `stdout` и `stderr` контейнера. Контейнеры, работающие очень долгое время, могут создавать большое количество логов. Только драйвер journald имеет встроенную функцию ротации логов, которая обеспечивается systemd. Если использовать драйвер k8s-file, есть риск, что в системе закончится свободное место. В табл. 7.2 приведены доступные драйверы, а также указано, сохраняются ли данные и поддерживает ли система ротацию логов.

Таблица 7.2. Варианты драйверов логирования

Драйвер	Описание	Сохранение логов после удаления контейнера	Ротация логов
Journald	Использует журнал systemd для хранения логов	✓	✓
k8s-file	Хранит данные логирования в плоском файле формата Kubernetes	✗	✗
None	Не сохраняет никаких логов	✗	✗

Хотя я рекомендую выбирать journald в качестве драйвера журнала, некоторым пользователям без прав root не разрешается его использовать из-за конфигурации их системы. В других случаях, например при запуске Podman в контейнере, journald оказывается недоступным.

Узнать, какой драйвер журнала установлен в системе по умолчанию, можно с помощью следующей команды:

```
$ podman info --format '{{ .Host.LogDriver }}'
k8s-file
```

Допустим, по какой-то причине в системных настройках вашего хоста было установлено ведение лога в k8s-файл. Переопределить драйвер логирования по умолчанию для вашей системы очень просто с помощью файла containers.conf. Создайте файл log_driver.conf в домашнем каталоге $HOME/.config/containers/containers.conf.d с установленным параметром `log_driver`:

```
$ mkdir -p $HOME/.config/containers/containers.conf.d
$ cat > $HOME/.config/containers/containers.conf.d/log_driver.conf << _EOF
[containers]
log_driver="journald"
_EOF
$ podman info --format '{{ .Host.LogDriver }}'
journald
```

Теперь все хорошо. Далее вы убедитесь в преимуществах драйвера journald, запустив контейнер с опцией `--rm` для удаления контейнера при выходе из него:

```
$ podman run --rm --name test2 ubi8 echo "Check if logs persist"
Check if logs persist
```

Убедитесь, что в журнале ведется запись о запуске контейнера:

```
$ journalctl -b | grep "Check if logs persist"
Nov 10 06:19:54 fedora conmon[657915]: Check if logs persist
```

Если бы вы запустили контейнер с опцией `k8s_file`, то Podman удалил бы файл журнала при удалении контейнера и никаких записей в логах не осталось бы. Как и в случае с логами, Podman поддерживает использование журнала systemd для хранения событий.

7.2.2. События

События Podman фиксируют различные этапы жизненного цикла контейнера. Например, событие «старт» последнего запущенного контейнера:

```
$ podman events --filter event=start --since 1h
2021-11-10 06:35:06.780429582 -0500 EST container start
➥ ecf04c4802bb120f34533560fbfc19ab023bcce63d48945ab0e8ff06cc6eeda1
...
```

Проверьте тип журнала событий по умолчанию с помощью команды Podman info:

```
$ podman info --format '{{ .Host.EventLogger }}'
journald
```

Журнал событий изменяется с помощью параметра `events_logger` в файле containers.conf аналогично тому, как это было сделано для `log_driver`. В табл. 7.3 приведены доступные варианты журналов событий.

Если в вашей системе используется регистратор событий file, то для пользователей без root резервный файл событий хранится в каталоге `$XDG_RUNTIME_DIR`, который по умолчанию находится на tmpfs. При использовании такого драйвера файл событий постоянно растет до перезагрузки системы. Это может привести к сбоям в работе контейнеров или нехватке места в системе, поскольку события

не ротируются, если не используется journald. Кроме того, при перезагрузке журнал событий теряется. Переход на journald сохраняет события и обеспечивает ротацию журнала. Я рекомендую оставить драйвер логирования и драйвер событий с одинаковыми значениями либо в виде journald или плоского файла, либо вообще без них, если вам не нужны события и логи.

Таблица 7.3. Варианты журналов событий

Журнал событий	Описание	Сохранение логов после удаления контейнера	Ротация логов
Journald	Информация обо всех событиях записывается в журнал systemd	✓	✓
File	Информация о событиях хранится в файле, обычно в /run	✗	✗
None	Информация о событиях не хранится	✗	✗

Мы рассмотрели использование systemd в Podman, а также journald для управления файлами логов и событиями. Теперь выясним, как настроить систему на автоматический запуск контейнера при загрузке системы с помощью systemd.

7.3. ЗАПУСК КОНТЕЙНЕРОВ ПРИ ЗАГРУЗКЕ

Из главы 1 вы узнали, что Podman не работает как демон, а значит, вы не можете полагаться на демон для автоматического запуска контейнеров во время загрузки системы. Контейнерные службы часто приходится запускать через systemd. Systemd можно настроить для установки, запуска и управления контейнерными приложениями. Многие приложения поставляются в виде образов контейнеров и содержат юнит-файлы systemd для запуска. В systemd реализовано множество функций, позволяющих упростить запуск контейнерных служб в системе.

7.3.1. Перезапуск контейнеров

Podman использует systemd для запуска контейнерных сервисов с помощью загрузки Podman в юнит-файлах systemd. Команда `podman run` позволяет указать, следует ли перезапускать контейнер (`--restart`), если он не остановлен пользователем (например, при сбое контейнера или перезагрузке системы). В табл. 7.4 приведены доступные в Podman политики перезапуска.

Один из простых способов, который имеется в systemd, — это запуск контейнеров с политикой перезапуска `always`. Если вы установили опцию `always` и система перезагрузилась, Podman использует две службы systemd для автоматического перезапуска контейнеров, помеченных параметром `--restart=always`. Одна служба обслуживает rootful-контейнеры, другая — все rootless-контейнеры в системе.

Таблица 7.4. Политики перезапуска

Параметр	Описание	Перезапуск при загрузке
`no`	Не перезапускает контейнеры при завершении их работы	✗
`on-failure[:max_retries]`	Перезапускает контейнеры при завершении их работы с ненулевым кодом завершения; повторные попытки продолжаются бесконечно или до тех пор, пока не будет превышено опциональное значение `max_retries`	✗
`always` or `unless-stopped`	Перезапускает контейнеры при их завершении независимо от статуса, повторяя неограниченное количество попыток	✓

При загрузке системы systemd выполняет следующую команду Podman для запуска всех контейнеров с установленной политикой перезапуска `always`:

```
/usr/bin/podman start --all --filter restart-policy=always
```

ПРИМЕЧАНИЕ Podman поставляется с двумя файлами служб systemd, используемыми для перезапуска сервисов, — один для rootful, другой для rootless:

/usr/lib/systemd/system/podman-restart.service

/usr/lib/systemd/user/podman-restart.service

Параметр `--restart=always` работает отлично, но он требует создания контейнера в системе и перезапускает контейнеры даже в случае их сбоя. Systemd был разработан для запуска сервисов; в следующем разделе я покажу, как можно просто создать сервисный юнит-файл с помощью Podman для запуска вашего контейнеризированного сервиса.

7.3.2. Контейнеры Podman как службы systemd

Итак, systemd использует юнит-файлы для определения порядка работы сервиса. На рис. 7.5 показано, как systemd работает с Podman для запуска контейнера.

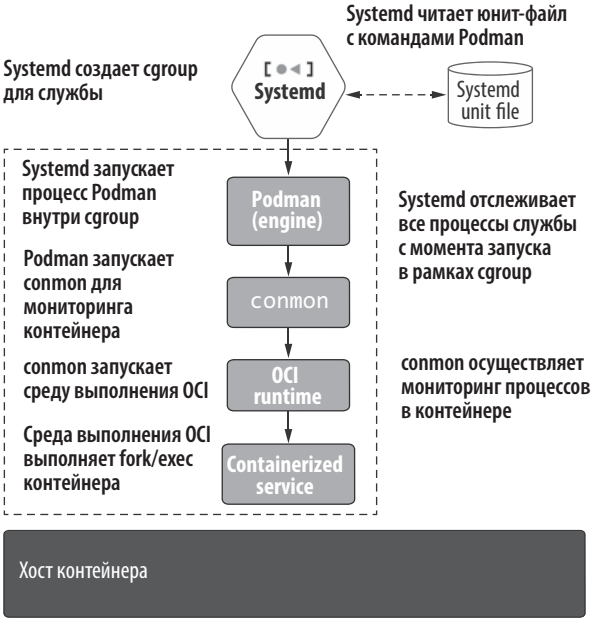


Рис. 7.5. Архитектура fork/exec в Podman идеально подходит для управления службами systemd

На рис. 7.5 я показал, что systemd может отслеживать все процессы, выполняющиеся в юнит-файле systemd. Это позволяет ему легко запускать и останавливать процессы. Процесс `conmon` также работает в рамках службы systemd, отслеживая процессы контейнера. `conmon` замечает, когда контейнер завершает работу, после этого он сохраняет код выхода и аккуратно останавливает контейнерное окружение. Systemd не знает о самом контейнере, ему известно только о процессах, выполняющихся в юнит-файле, включая процессы контейнера.

В юнит-файлах Systemd имеется множество различных способов выполнения и запуска процессов, а в Podman — множество вариантов запуска контейнеров. Конфигурирование этих файлов может быть очень сложным. Многие пользователи писали свои юнит-файлы для запуска контейнеров, но при этом сталкивались с различными проблемами. Наиболее распространенной проблемой является выполнение команды `podman run --detach` внутри юнит-файла. Когда команда Podman отсоединяется и выходит из системы, systemd отключает службу как завершенную, хотя `conmon` и контейнер все еще работают. «Как я должен запускать свой контейнер внутри юнит-файла systemd?» — один из самых распространенных вопросов, который я слышу от пользователей.

В Podman есть функция генерации юнит-файлов с оптимальными настройками по умолчанию. Сначала пересоздайте контейнер из `myimage`, а затем с помощью `podman systemd generate` создайте юнит-файл службы systemd для управления контейнером.

Создайте контейнер на основе образа, созданного в главе 2:

```
$ podman create -p 8080:8080 --name myapp quay.io/rhatdan/myimage
...
8879112805e976b4b6d97c07c9426bdde22ee4ffc7ba4daa59965ae25aa08331
```

Теперь с помощью Podman сгенерируйте из этого контейнера юнит-файл:

```
$ mkdir -p $HOME/.config/systemd/user
$ podman generate systemd myapp > $HOME/.config/systemd/user/myapp.service
```

Обратите внимание, что в скрипте myapp.service Podman создал поле `ExecStart`. При начале работы службы systemd выполнит команду `ExecStart`, которая просто запустит созданный вами контейнер:

```
ExecStart=/usr/bin/podman start 8879112805...
```

При остановке сервиса systemd выполняет команду `ExecStop`, добавленную в юнит-файл:

```
ExecStop=/usr/bin/podman stop -t 10 8879112805...
Let's take a look at the generated service file:
$ cat $HOME/.config/systemd/user/myapp.service
# container-
     8879112805e976b4b6d97c07c9426bdde22ee4ffc7ba4daa59965ae25aa08331.service
# autogenerated by Podman 3.4.1
# Wed Nov 10 08:23:06 EST 2021
[Unit]
Description=Podman container-8879112805...service
Documentation=man:podman-generate-SystemD(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/3267/containers
[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start 8879112805...
ExecStop=/usr/bin/podman stop -t 10 8879112805...
ExecStopPost=/usr/bin/podman stop -t 10 8879112805...
PIDFile=/run/user/3267/containers/overlay-containers/8879112805.../
         userdata/conmon.pid
Type=forking
[Install]
WantedBy=multi-user.target default.target
```

Чтобы все это заработало, необходимо указать systemd на перезагрузку своей базы данных, тогда она заметит изменения в юнит-файлах:

```
$ systemctl --user daemon-reload
```

Запустите службу с помощью следующей команды:

```
$ systemctl --user start myapp
```

Убедитесь в том, что служба запущена:

```
$ systemctl --user status myapp
• myapp.service - Podman container-8879112805....service
   Loaded: loaded (/home/dwalsh/.config/SystemD/user/myapp.service;
➥ disabled; vendor preset: disabled)
   Active: active (running) since Thu 2021-11-11 07:19:08 EST; 3min 9s ago
...
$ podman ps
CONTAINER ID   IMAGE                  COMMAND
➥ CREATED      STATUS    PORTS    NAMES
8879112805e9 quay.io/rhatdan/myimage:latest /usr/bin/run-http...
➥ 23 hours ago Up 5 minutes ago 0.0.0.0:8080->8080/tcp myapp
```

Запустите в браузере localhost на порте `8080`, чтобы убедиться, что все работает (рис. 7.6):

```
$ web-browser localhost:8080
```

Рис. 7.6. Окно веб-браузера с подключением к myapp

Чтобы выключить службу, выполните команду:

```
$ systemctl --user stop myapp
```

Возможность генерировать сервисные файлы systemd предоставляет пользователям значительную гибкость и умышленно стирает разницу между контейнером и любой другой программой или сервисом на хосте.

Одна из проблем с рассмотренным юнит-файлом заключается в том, что он привязан к созданному контейнеру. Необходимо сначала создать контейнер и сгенерировать конкретные файлы. Вы не можете передать юнит-файл другому пользователю, чтобы он запустил вашу службу на своей машине.

К счастью, Podman поддерживает создание переносимого юнит-файла systemd: `podman generate systemd --new`.

7.3.3. Распространение юнит-файлов systemd для управления контейнерами Podman

Как было показано ранее, команда `podman generate systemd` сгенерировала юнит-файл, который запускал и останавливал существующий контейнер. Флаг `--new` передает Podman инструкцию генерировать юнит-файлы, которые запускают, останавливают и удаляют контейнеры. Попробуйте это применить в том же контейнере:

```
$ podman generate systemd --new myapp > $HOME/.config/systemd/user/
➥ myapp-new.service
```

Обратите внимание, что при использовании параметра `--new` Podman создает несколько отличающийся юнит-файл. Посмотрите на следующую команду `ExecStart`, и вы увидите, что первоначальная команда `podman create -p 8080:8080 --name myapp quay.io/rhatdan/myimage`, которую вы использовали для создания контейнера, была изменена на команду `podman run`. Обратите внимание, что в Podman добавлены дополнительные параметры для упрощения работы под управлением systemd (`--cidfile =%t/%n.ctr-id --cgroups=no-conmon --rm --sdnotify=conmon -d --replace`).

Podman добавляет команду `ExecStop` (`/usr/bin/podman stop --ignore --cidfile=%t/%n.ctr-id`), указывающую systemd, как остановить контейнер, если кто-то выполнит команду `systemctl stop` или система выключится.

Наконец, Podman добавляет команду `ExecStopPost` (`/usr/bin/podman rm -f --ignore --cidfile=%t/%n.ctr-idType=notify`), которую systemd выполняет после завершения выполнения команды `ExecStop`. Эта команда удаляет контейнер из системы:

```
$ cat $HOME/.config/systemd/user/myapp-new.service
# container-8879112805....service
# autogenerated by Podman 3.4.1
# Thu Nov 11 07:40:34 EST 2021
[Unit]
Description=Podman container-8879112805...service
Documentation=man:podman-generate-SystemD(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=%t/containers
[Service]
Environment=PODMAN_SystemD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStartPre=/bin/rm -f %t/%n.ctr-id
ExecStart=/usr/bin/podman run --cidfile=%t/%n.ctr-id --cgroups=no-conmon –
➥ rm --sdnotify=conmon -d --replace -p 8080:8080 --name myapp
➥ quay.io/rhatdan/myimage
ExecStop=/usr/bin/podman stop --ignore --cidfile=%t/%n.ctr-id
```

```
ExecStopPost=/usr/bin/podman rm -f --ignore --cidfile=%t/%n.ctr-idType=notify
NotifyAccess=all
[Install]
WantedBy=multi-user.target default.target
```

Вы можете удалить контейнер и его образ из своей системы, а когда вы сообщите `systemctl` о запуске службы, Podman загрузит образ и создаст новый контейнер. Это означает, что юнит-файлом myapp-new.service можно поделиться с другим пользователем. Когда пользователь запустит эту службу, Podman подобным же образом загрузит образ и запустит контейнер в его системе, при этом предварительно создавать контейнер не понадобится. В табл. 7.5 приведены различные команды, добавляемые в юнит-файл в зависимости от того, использовался ли флаг `--new`.

Таблица 7.5. Различия между юнит-файлами

Параметр	Команды

С флагом `--new`	`ExecStart=/usr/bin/podman run ...--cidfile=%t/%n.ctr-id --cgroups=no-`
	`➥ conmon --rm --sdnotify=conmon -d --replace -p 8080:8080 –name`
	`➥ myapp quay.io/rhatdan/myimage`
	`ExecStop=/usr/bin/podman stop --ignore --cidfile=%t/%n.ctr-id`
	`ExecStopPost=/usr/bin/podman rm -f --ignore --cidfile=%t/%n`
	`➥ .ctr-idType=notify`

Без флага `--new`	`ExecStart=/usr/bin/podman start 8879112805...`
	`ExecStop=/usr/bin/podman stop -t 10 8879112805...`
	`ExecStopPost=/usr/bin/podman stop -t 10 8879112805...`

Так как контейнерный сервис теперь запущен на множестве машин, необходимо подумать о его поддержке. В Podman есть способ сделать это без вмешательства человека: автообновление.

7.3.4. Автоматическое обновление контейнеров Podman

В главе 2 мы говорили о том, что образы контейнеров стареют «как сыр». Если образ контейнера обновляется с помощью нового программного обеспечения или исправления уязвимостей, необходимо связаться с использующими его машинами, загрузить обновленные образы и заново создать контейнерные службы. Когда машины сами управляют обновлениями, это требует гораздо меньше усилий.

Представьте, что вы настраиваете службу для работы с образом контейнера на сотнях узлов. Спустя несколько месяцев вы добавляете новые функции в приложение в образе или, что более важно, обнаруживается новая уязвимость. Тогда необходимо обновить образ, а затем заново создать службу на всех узлах.

Podman автоматизирует этот процесс с помощью автообновления. Каждый узел следит за появлением новых образов в реестре контейнеров. Когда образ

появляется, нода загружает его и заново создает контейнер. Никакого взаимодействия с человеком при этом не требуется.

Автообновление позволяет использовать Podman в нестандартных сценариях, обновлять рабочие среды после их подключения к сети и откатывать сбои до заведомо исправного состояния. Кроме того, запуск контейнеров необходим для реализации граничных вычислений (edge computing) в удаленных центрах обработки данных или на устройствах интернета вещей. Автоматическое обновление Podman позволяет обновлять рабочие нагрузки после их подключения к сети и снижать затраты на их обслуживание.

Для реализации автообновления Podman требует, чтобы контейнеры имели специальную метку, `--label "io.containers.autoupdate=registry"`, а сам контейнер запускался в systemd-юните, сгенерированном командой `podman generate systemd --new`. В табл. 7.6 представлены доступные режимы автообновления.

Таблица 7.6. Режимы автообновления

`io.containers.autoupdate`	**Описание**
`registry`	Podman подключается к реестру контейнеров и проверяет, доступен ли образ, отличный от использовавшегося при создании контейнера; если он есть, то Podman обновляет контейнер
`local`	Podman подключается к реестру контейнеров, но сравнивает локальные образы с использовавшимся при создании контейнера; если они отличаются, Podman обновляет контейнер

Сначала остановите службу systemd, если она запущена, и удалите существующий контейнер `myapp`:

```
$ systemctl --user stop myapp-new
$ podman rm myapp --force -t 0
```

Пересоздайте контейнер myapp со специальной меткой `"io.containers.autoupdate= registry"`:

```
$ podman create --label "io.containers.autoupdate=registry" -p 8080:8080
➥ --name myapp quay.io/rhatdan/myimage
397ad15601868eb6fd77fe0b67136869cde9e0ffad90ee5095a19de5bb4b999e
```

Пересоздайте юнит-файл systemd с помощью параметра `--new`:

```
$ podman generate systemd myapp --new > $HOME/.config/systemd/user/
➥ myapp-new.service
```

Сообщите systemd об изменении юнит-файла, выполнив команду `daemon-reload`, и запустите службу:

```
$ systemctl --user daemon-reload
$ systemctl --user start myapp-new
```

Теперь служба `myapp-new` готова к автоматическому обновлению. При получении команды `podman auto-update` Podman проверяет запущенные контейнеры на наличие метки `io.containers.autoupdate` в `image`. Для каждого контейнера с такой меткой Podman обращается к реестру контейнеров и проверяет, не изменился ли образ с момента создания контейнера. Если образ изменился, Podman перезапускает соответствующий systemd-юнит. Напомним, что при перезапуске systemd выполняются следующие действия:

1. Systemd останавливает службу, выполняя команду `podman stop`:

   ```
   ExecStop=/usr/bin/podman stop --ignore --cidfile=%t/%n.ctr-id
   ```

2. Systemd выполняет скрипт `ExecStopPost`. После остановки контейнера этот скрипт удаляет контейнер с помощью команды `podman rm`:

   ```
   ExecStopPost=/usr/bin/podman rm -f --ignore --cidfile=%t/
   ➥ %n.ctr-idType=notify
   ```

3. Systemd перезапускает службы с помощью команды `podman run`, включая параметр `--label "io.containers.autoupdate=registry"`:

   ```
   ExecStart=/usr/bin/podman run --cidfile=%t/%n.ctr-id --cgroups=no-conmon –rm
   ➥ --sdnotify=conmon -d --replace –label
   ➥ io.containers.autoupdate=registry -p 8080:8080
   ➥ --name myapp quay.io/rhatdan/myimage
   ```

Команда `podman run` на третьем шаге обратится к реестру, получит обновленный образ контейнера и пересоздаст на его основе контейнерное приложение. Контейнер, его окружение и все зависимости перезапускаются.

Это можно проверить, изменив образ, разместив его в реестре и выполнив команду `podman auto-update` следующим образом:

```
$ podman exec -i myapp bash -c 'cat > /var/www/html/index.html' << _EOF
<html>
 <head>
 </head>
 <body>
  <h1>Welcome to the new Hello World<h1>
 </body>
</html>
_EOF
```

Теперь выполните коммит образа под именем `myimage-new` и добавьте его в реестр с исходным именем `myimage`. Наконец, удалите образ из локального хранилища, имитируя, что образ никогда не находился в вашей системе:

```
$ podman commit myapp quay.io/rhatdan/myimage-new
...
226ec055eef82ac185c53a26de9e98da4e6403640e72c7461a711edcbcaa2422
$ podman push quay.io/rhatdan/myimage-new quay.io/rhatdan/myimage
...
$ podman rmi quay.io/rhatdan/myimage-new
```

Как только новый образ окажется в реестре и вы удалите его из локального хранилища, можно запустить команду `podman auto-update`, которая заметит новый образ и перезапустит сервис. Это вызовет Podman для загрузки нового образа и пересоздания контейнерного сервиса:

```
$ podman auto-update
Trying to pull quay.io/rhatdan/myimage...
Getting image source signatures
Copying blob ecfb9899f4ce done
Copying config 37e5619f4a done
Writing manifest to image destination
Storing signatures
UNIT CONTAINER IMAGE
➥ POLICY UPDATED
myapp-new.service c8888d1319c4 (myapp) quay.io/rhatdan/myimage registry
➥ true
```

Ваше приложение обновлено до последней версии образа.

Ниже приводятся некоторые часто используемые параметры команды `podman auto-update`:

- `--dry-run`. Параметр полезен для проверки необходимости обновления каких-либо контейнеров без их фактического обновления.
- `--roll-back`. Параметр указывает Podman на откат к предыдущему образу в случае неудачного обновления, о чем будет рассказано в следующем разделе.

Таймеры SYSTEMD, инициирующие обновления Podman

Podman содержит два таймера автообновления systemd и два сервис-юнита автообновления — по одному для rootful- и rootless-контейнеров. Раз в сутки systemd запускает следующие таймеры:

- /usr/lib/systemd/system/podman-auto-update.timer;
- /usr/lib/systemd/user/podman-auto-update.timer.

Таймеры указывают systemd на выполнение соответствующего юнит-файла автообновления:

- /usr/lib/systemd/system/podman-auto-update.service;
- /usr/lib/systemd/user/podman-auto-update.service.

С помощью этой функции systemd запускает Podman, который ищет контейнеры с меткой `"io.containers.autoupdate=registry"` подобно тем, что вы создали в предыдущем разделе. Как только Podman находит контейнер с такой меткой, он проверяет, был ли обновлен образ контейнера в реестре. Если образ изменился, Podman запускает процесс обновления. Это означает, что вы можете эксплуатировать свои системы без постоянного сопровождения, и они будут обновляться все 24 часа, получая новейшую версию образа контейнера каждый раз, когда вы отправляете обновленный образ в реестр. Если вы поделитесь созданным юнит-файлом с другими пользователями, то они также получат автоматическое обновление.

При использовании автообновления серьезное беспокойство вызывает вопрос о том, что произойдет, если обновление будет сбойным. В этом случае сотни узлов будут обновлены до неисправной версии сервиса. В systemd есть функция `sd-notify`, которая позволяет службе сообщить, что ее инициализация завершена и она готова к использованию.

> **ПРИМЕЧАНИЕ** Часть этого раздела основана на моих ранее написанных статьях из блога «Как использовать автообновления и откаты в Podman» («How to use auto-updates and rollbacks in Podman») (https://www.redhat.com/sysadmin/podman-auto-updates-rollbacks), соавторами которых являются мои коллеги Валентин Ротберг (Valentin Rothberg) и Прити Томас (Preethi Thomas).

7.4. ЗАПУСК КОНТЕЙНЕРОВ В ЮНИТ-ФАЙЛАХ ТИПА NOTIFY

Службы в юнит-файлах могут ожидать запуска до тех пор, пока не будут запущены другие службы. Например, можно создать веб-сайт, для которого база данных должна быть запущена до того, как веб-сервис начнет принимать соединения. Systemd обычно считает службу запущенной после старта ее основного процесса. Однако многие службы требуют времени на инициализацию и не могут сразу принимать соединения. Например, упомянутой выше базе данных может потребоваться несколько минут, прежде чем веб-сервис будет готов к приему соединений.

Systemd использует специальный тип службы `notify` (или `sd-notify`), позволяющий сервисному процессу уведомлять systemd о том, что он действительно

полностью готов к работе. Systemd запускает веб-сервис только тогда, когда получает уведомление о готовности базы данных.

Systemd сообщает службе, что его необходимо уведомить о готовности, передавая переменную окружения `NOTIFY_SOCKET`. Эта переменная указывает на сокет systemd, который необходимо задействовать для уведомления. По умолчанию systemd прослушивает сокет /run/SystemD/notify. Когда Podman запускается внутри юнит-файла `NOTIFY`, ему необходимо выполнить монтирование сокета в контейнер и передать контейнеру переменную окружения (рис. 7.7).

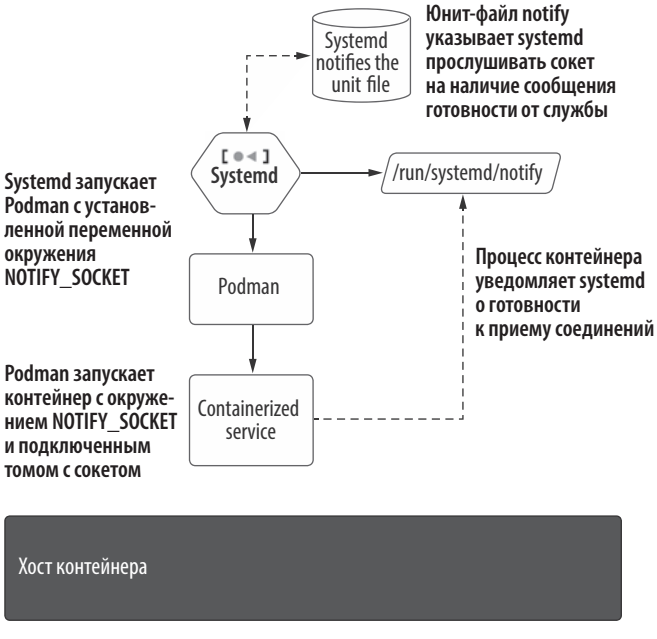


Рис. 7.7. Контейнерная служба systemd sd_notify, запущенная с помощью Podman

Если в течение указанного времени служба не уведомит systemd, systemd пометит ее как неработающую. Автообновление Podman проверяет, полностью ли запущена новая служба, и если проверка не проходит, то Podman может автоматически откатиться к предыдущему состоянию — и все без вмешательства человека.

7.5. ОТКАТ НЕИСПРАВНЫХ КОНТЕЙНЕРОВ ПОСЛЕ ОБНОВЛЕНИЯ

Если определенная вами служба поддерживает `sd-notify` и пишет в сокет notify в пределах установленного временного интервала, то команда `podman auto-update`

будет успешной. При неудаче автообновления Podman удалит новый контейнер и перетегирует исходный образ. Наконец, он создаст контейнер на предыдущем образе, и ваш сервис вернется в прежнее состояние. Можно даже настроить свой контейнерный сервис на базе systemd так, чтобы система логирования получала уведомление о неудачном обновлении. Откат дает вам время разобраться, что пошло не так, загрузить новый образ и снова запустить автообновление. Как видите, systemd можно использовать в качестве контейнерного оркестратора для одной машины.

Вы открыли для себя несколько интересных особенностей, которые предоставляет systemd для запуска контейнеров без участия человека. Еще одна удобная дополнительная функция — активация сокетов, позволяющая указать в юнит-файле контейнер, который не запустится до тех пор, пока первый пакет не поступит в его сокет.

7.6. КОНТЕЙНЕРЫ PODMAN, АКТИВИРУЕМЫЕ ЧЕРЕЗ СОКЕТ

Когда systemd только появилась, ее хвалили за ускорение загрузки системы. До появления systemd сервисы запускались последовательно, и сервисам, которые зависели от запуска других сервисов, приходилось ждать. Чтобы ускорить загрузку и улучшить распределение ресурсов, systemd использует *службы, активируемые через сокет*. Когда служба активируется через сокет, systemd устанавливает прослушивающие IP или доменные сокеты UNIX от имени вашей службы, не запуская ее саму (рис. 7.8).

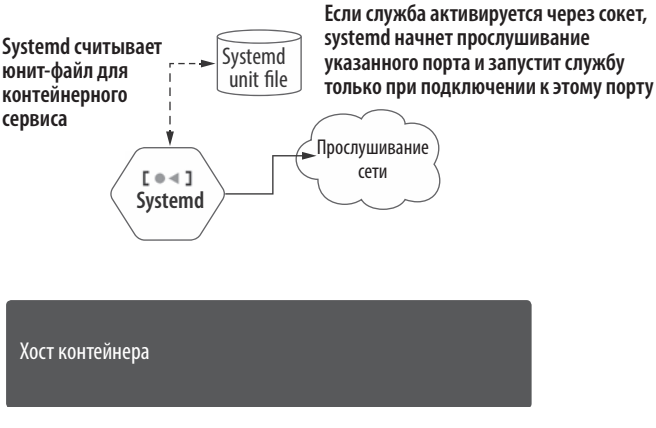


Рис. 7.8. Systemd прослушивает сокет для контейнера

При установлении соединения с сокетом systemd активирует службу и передает ей это подключение. После этого служба обрабатывает соединения. В дальнейшем она может завершить свою работу, выйдя из системы. Если поступает новое соединение, systemd принимает его и снова запускает службу.

Активация через сокет позволяет systemd мгновенно сигнализировать о запуске службы, не запуская ее на самом деле и не ожидая ее запуска, что ускоряет процесс загрузки. Активация с помощью сокетов позволяет systemd запускать больше служб в системе, поскольку многие службы простаивают и не используют системные ресурсы. В сущности, ваши службы могут быть остановлены, а запущены только тогда, когда они действительно необходимы, и они не будут простаивать в ожидании очередного соединения. При использовании контейнерных сервисов главным процессом службы является Podman, и он должен передать соединение службе, запущенной внутри контейнера (рис. 7.9).

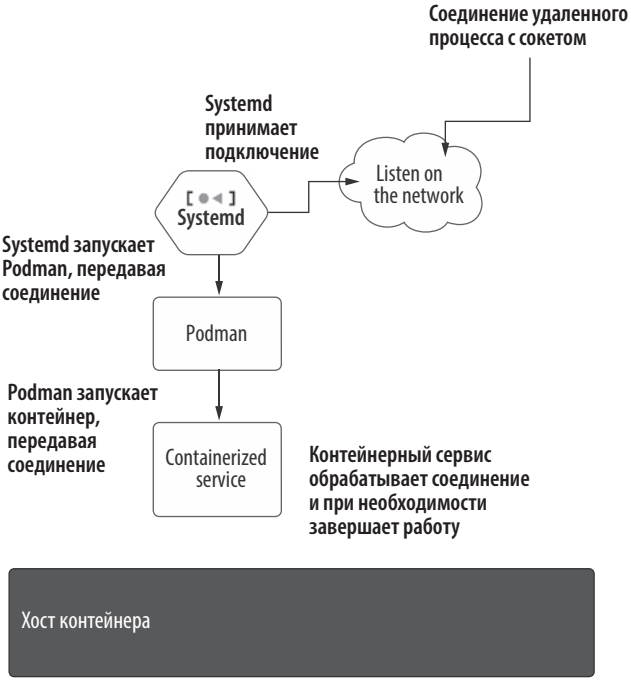


Рис. 7.9. При установлении соединения с сокетом, который прослушивает systemd, systemd активирует Podman, а он, в свою очередь, запускает контейнер, передавая ему сокет

Отключите службу myapp.service и создайте сокет myapp.socket:

```
$ systemctl --user stop myapp.service
$ cat > $HOME/.config/systemd/user/myapp.socket <<_EOF
[Unit]
Description=myapp socket service
PartOf=myapp.service
[Socket]
ListenStream=127.0.0.1:8080
[Install]
WantedBy=sockets.target
_EOF
```

Теперь включите сокет и убедитесь, что ни один контейнер не запущен:

```
$ systemctl --user enable --now myapp.socket
$ podman ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS
➥ PORTS    NAMES
```

Подключитесь к сокету через веб-браузер (рис. 7.10).

Рис. 7.10. Окно веб-браузера, подключенного к запущенному в Podman контейнеру ubi8/httpd-24, с обновленным HTML-файлом Hello World

Обратите внимание, что podman.socket запустил podman.service, который создал контейнер для обслуживания данного соединения:

```
$ podman ps
CONTAINER ID    IMAGE    COMMAND    CREATED
➥ STATUS    PORTS    NAMES
69c34949d632 quay.io/rhatdan/myimage:latest /usr/bin/run-http...
➥ 2 minutes ago Up 2 minutes ago 0.0.0.0:8080->8080/tcp myapp
```

Теперь, если остановить службу, контейнер будет не только остановлен, но и удален:

```
$ systemctl --user stop myapp.service
$ podman ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS
➥ PORTS    NAMES
```

Активация через сокет позволяет запускать службу только при необходимости, экономя системные ресурсы. В дальнейшем можно отключить эту службу, зная, что при появлении нового соединения systemd и Podman его обработают.

РЕЗЮМЕ

- Podman позволяет запускать systemd в качестве начального процесса в контейнере.
- Для ведения логов и событий Podman рекомендуется использовать journald.
- Systemd может запускать и перезапускать контейнеры во время загрузки системы.
- Автообновление Podman используется для управления жизненным циклом контейнера и его образа.
- Активируемые через сокеты службы systemd могут применяться совместно с контейнерами на базе Podman.
- Команда `podman generate systemd` позволяет без сложностей генерировать файлы служб systemd для запуска контейнеров.

8 Работа с Kubernetes

В ЭТОЙ ГЛАВЕ

- ✓ Создание YAML-файлов Kubernetes из существующих подов и контейнеров Podman
- ✓ Создание контейнеров и подов Podman из YAML-файлов Kubernetes
- ✓ Выключение и удаление подов и контейнеров с помощью YAML-файла Kubernetes
- ✓ Создание образа контейнера на лету перед запуском подов и контейнеров из YAML-файла Kubernetes
- ✓ Запуск Podman внутри контейнеров в Podman и Kubernetes

Некоторые читатели ожидают от этой главы рассказа о том, как Podman, подобно Docker, может служить в качестве контейнерного движка для Kubernetes. Несмотря на то что предпринимались попытки использовать Podman подобным образом (проект kind это поддерживает), в целом я не советую этого делать. Лучше применять CRI-O, описанный в приложении A, поскольку он был создан специально для работы с Kubernetes и имеет общие базовые библиотеки

с Podman. В настоящее время в качестве бэкенда Kubernetes рекомендуется использовать именно CRI-O или containerd, а не Docker.

В этой главе рассказывается о применении одного и того же структурированного формата в Kubernetes и Podman, а также о том, как запускать контейнеры Podman внутри кластера Kubernetes. Вы уже знаете, как создавать микросервисы в виде контейнеров и подов из командной строки с помощью Podman. Часто разработчикам и сборщикам программного обеспечения необходимо запустить свои приложения на нескольких машинах. Возможно, вам потребуется добавить к своему веб-приложению бэкенд базы данных. Если веб-приложение станет популярным, нужно будет запустить несколько экземпляров на разных узлах. Соединение различных микросервисов вместе и их оркестрация — это не совсем то, что умеет Podman. Здесь на помощь приходит Kubernetes.

В этой главе вы узнаете, как запустить те же самые контейнеры и поды в Kubernetes. На сайте kubernetes.io говорится: «Kubernetes, также известный как K8s, — это система с открытым исходным кодом для автоматизации развертывания, масштабирования контейнерных приложений и управления ими». Я рассматриваю Kubernetes как инструмент для одновременного запуска контейнеров на нескольких машинах, то есть способ организации больших кластеров контейнерных микросервисов.

Одна из проблем состоит в том, что в большинстве случаев разработка контейнеров ведется с помощью таких инструментов, как Podman и Docker, которые используют довольно простые интерфейсы командной строки для создания контейнеров и подов. В Kubernetes же используется декларативный язык, описываемый в YAML-файлах.

В этой главе я не буду подробно останавливаться на том, как работает Kubernetes, поскольку на эту тему уже существует множество книг, в том числе «Kubernetes in Action» Марко Лукши (Marko Lukša)[1] (Manning, 2020) и «Kubernetes for developers» Уильяма Деннисса (William Denniss, Manning, 2020), в которых рассказывается обо всех возможностях Kubernetes. Я же буду описывать язык разработчиков Kubernetes — YAML-файл Kubernetes.

> **ПРИМЕЧАНИЕ** На сайте yaml.org YAML сначала определяется как «YAML Ain't Markup Language», то есть «YAML — не язык разметки». Далее уточняется: «YAML — это удобный для человека язык сериализации данных для всех языков программирования».

Перевод параметров командной строки на структурированный язык YAML представляет сложность для разработчиков, переходящих от контейнеров на

[1] Марко Лукша, «Kubernetes в действии».

одном узле к контейнерам, запускаемым в больших масштабах. Как указать тома, используемый образ, ограничения безопасности, сетевые порты и т. д.? В разделе 8.2 вы научитесь использовать Podman для генерации YAML-файлов Kubernetes из созданных локальными подов и контейнеров.

После написания и развертывания приложения в поде с помощью YAML-файлов Kubernetes пользователи, скорее всего, обнаружат проблемы с работой своего приложения в Kubernetes. Тестирование приложения в масштабе может оказаться непростой задачей. Часто требуется просто запустить его локально в своей системе, без установки и настройки кластера Kubernetes. В разделе 8.3 вы познакомитесь с командой `podman play kube`. Эта команда позволяет запустить YAML-файл Kubernetes локально, без применения самого Kubernetes, что дает возможность проводить тесты и отлаживать проблемы.

В заключительной части этой главы будет рассмотрена работа Podman внутри контейнеров, в том числе в кластере Kubernetes. Администраторам, разработчикам и QA-инженерам необходимо тестировать контейнеры в системах непрерывной интеграции (CI) с помощью Podman. Часто эти CI-системы построены на кластерах Kubernetes. В разделе 8.4 приводятся различные способы запуска команд Podman внутри контейнеров, запускаемых Podman и Kubernetes.

8.1. YAML-ФАЙЛЫ KUBERNETES

YAML-файл Kubernetes — это объект, применяемый для запуска подов и контейнеров в Kubernetes. В главе 5 вы узнали, что используемые в Podman конфигурационные файлы написаны с помощью TOML, который очень похож на YAML. Оба языка конфигурации стремятся стать человекочитаемыми. YAML использует отступы, что отличается от синтаксиса TOML. За более подробной информацией об этом языке обращайтесь к сайту yaml.org.

Если вы собираетесь много работать с YAML-файлами Kubernetes, было бы неплохо иметь текстовый редактор или IDE, например Visual Studio или VS Code, которые, по крайней мере, понимают YAML. А еще лучше, если редактор или IDE знают язык Kubernetes. Kubernetes YAML является наглядным и мощным. Он позволяет моделировать желаемое состояние приложения на декларативном языке. Как было сказано во введении к этой главе, написание YAML-файлов является для разработчиков препятствием при переносе контейнеров из локальной системы в Kubernetes. Большинство разработчиков просто ищут в интернете существующий YAML-файл Kubernetes, а затем начинают вырезать и вставлять в YAML-файл команду, образ и параметры контейнера. Хотя такой метод и допустим, он может привести к непредвиденным последствиям, а зачастую к лишней работе.

Скотт Маккарти (Scott McCarty), продакт-менеджер Podman, однажды сказал: «Что я действительно хотел бы сделать, так это помочь пользователям перейти от Podman к оркестрации контейнеров с помощью Kubernetes». Это подтолкнуло разработчиков Podman к созданию новой команды Podman: `podman generate kube`.

8.2. ГЕНЕРАЦИЯ YAML-ФАЙЛОВ KUBERNETES С ПОМОЩЬЮ PODMAN

Представьте, что вам нужно взять контейнеры, созданные в предыдущих главах, и запустить их в Kubernetes. Для этого необходимо написать YAML-файл Kubernetes. С чего следует начать?

В этой главе вы познакомитесь с новой командой — `podman generate kube`. Команда берет описания локальных подов и контейнеров, а затем переводит их в формат Kubernetes YAML. Это помогает перейти к более сложной среде оркестрации, каковой является Kubernetes. Сгенерированный YAML-файл Kubernetes затем используется командами Kubernetes для запуска ваших подов и контейнеров в кластере.

Пересоздать контейнеры или поды можно локально, используя командную строку Podman с теми же командами `run`, `create` и `stop`, которые были изучены в предыдущих главах. Используя следующие команды, пересоздайте контейнер, с которым вы работали.

Сначала удалите контейнер, если он существует, с помощью команды `podman rm`. Введите новый флаг `--ignore`, который дает указание команде `podman rm` не сообщать об ошибках, если контейнер не существует. Затем заново создайте контейнер из командной строки:

```
$ podman rm -f --ignore myapp
$ podman create -p 8080:8080 --name myapp quay.io/rhatdan/myimage
9305822e6089ca28a1fdbb005c12f57f4a26be273fe5d49a1908eadbcfdcb7d4
```

Теперь выполните команду `podman generate kube myapp` для создания YAML-файла Kubernetes. Podman анализирует существующий контейнер или под в своей базе данных на предмет наличия всех полей, необходимых для запуска контейнера в Kubernetes, и затем заполняет их в YAML-файле Kubernetes:

```
$ podman generate kube myapp > myapp.yaml
```

На рис. 8.1 показан результат выполнения команды `podman generate kube`.

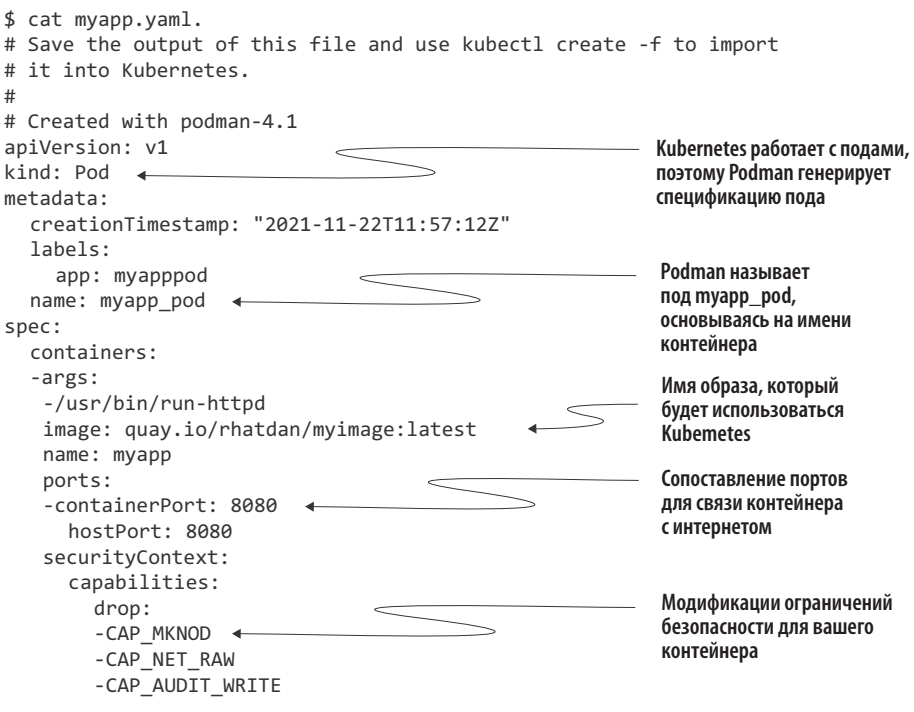


Рис. 8.1. Сгенерированный файл myapp.yaml из контейнера myapp

Рассмотрим части YAML-файла. Нужно понимать, что Kubernetes работает с подами: даже если вы создали контейнер, `podman generate kube` создает спецификацию пода. Podman называет под `myapp-pod` и контейнер `myapp` внутри данной спецификации, основываясь на имени исходного контейнера:

```
metadata:
  creationTimestamp: "2021-11-22T11:57:12Z"
  labels:
    app: myapppod
  name: myapp-pod
spec:
  containers:
- args:
  - /usr/bin/run-httpd
  image: quay.io/rhatdan/myimage:latest
  name: myapp
```

Обратите внимание, что в разделе containers записано имя образа quay.io/rhatdan/myimage:latest, оно указывает Kubernetes, откуда загружать образ для

контейнера. Оно также передает Kubernetes аргументы команды для запуска приложения внутри контейнера — /usr/bin/run-httpd:

```
spec:
containers:
  - args:
    - /usr/bin/run-httpd
    image: quay.io/rhatdan/myimage:latest
```

В той же секции порты Podman прописаны как `-p 8080:8080`:

```
containers:
- args:
  - /usr/bin/run-httpd
  image: quay.io/rhatdan/myimage:latest
  name: myapp
  ports:
  - containerPort: 8080
  hostPort: 8080
```

В конце раздела containers находится секция `securityContext`, в которой указано, что Podman по умолчанию отключает три дополнительных Linux-привилегии (capabilities): `CAP_MKNOD`, `CAP_NET_RAW` и `CAP_AUDIT_WRITE`:

```
securityContext:
capabilities:
drop:
- CAP_MKNOD
- CAP_NET_RAW
- CAP_AUDIT_WRITE
```

Большинство контейнеров прекрасно работают без Linux-привилегий, но спецификация OCI включает эти три по умолчанию. Получив сведения, что контейнер будет безопасно функционировать без указанных привилегий, Kubernetes отказывается от них. Чтобы получить более подробную информацию о Linux-привилегиях, выполните команду `man capabilities`.

На данном этапе можно запустить этот YAML-файл в любом кластере Kubernetes с помощью команды, подобной приведенной ниже:

```
kubectl create -f myapp.yml
```

К YAML-файлу часто приходится добавлять сложные элементы оркестрации и использовать расширенные функции Kubernetes. Например, сгенерированный YAML-файл Kubernetes создаст только один экземпляр вашего приложения. Если необходимо запустить несколько версий приложения на разных нодах, добавьте в YAML-файл параметр `replicas`, как показано на рис. 8.2.

```
# Save the output of this file and use kubectl create -f to import
# it into Kubernetes.
#
# Created with podman-4.1
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2021-11-22T11:57:12Z"
  labels:
    app: myapppod
  name: myapp_pod
spec:
  containers:
  - args:
    - /usr/bin/run-httpd
    image: quay.io/rhatdan/myimage:latest
    name: myapp
    ports:
    - containerPort: 8080
      hostPort: 8080
    securityContext:
      capabilities:
        drop:
        - CAP_MKNOD
        - CAP_NET_RAW
        - CAP_AUDIT_WRITE
replicas:  2
```

Дает указание Kubernetes запустить два пода с данным шаблоном

Рис. 8.2. Модифицированный YAML-файл Kubernetes, готовый к запуску двух реплик

Флаг `replicas` сообщает Kubernetes, что файл myapp.yaml предполагает постоянную работу двух подов myapp на двух разных узлах. Реплики и другие расширенные возможности Kubernetes не входят в сферу применения Podman. Команда `podman play kube` игнорирует эти поля.

Ниже приводятся некоторые часто используемые параметры команды `podman generate kube`:

- `-f`, `--filename`. Записывает вывод в указанный путь.
- `-s`, `--service`. Генерирует YAML для объекта service в Kubernetes.

Вы сгенерировали YAML-файл Kubernetes, и теперь было бы неплохо научиться обратному процессу. Возможно, вам понадобится сгенерировать Podman-поды и контейнеры из имеющегося Kubernetes YAML-файла.

8.3. ГЕНЕРАЦИЯ ПОДОВ И КОНТЕЙНЕРОВ PODMAN ИЗ KUBERNETES YAML

Представьте, что вы получили Kubernetes YAML-файл и хотите проверить его работу локально. Можно создать свой кластер Kubernetes, но было бы удобнее получить возможность просто запускать поды локально. Для этого Podman предусматривает команду `podman play kube`, которая создает поды, контейнеры и тома на основе структурированных YAML-файлов Kubernetes. Созданные поды и контейнеры запускаются автоматически. Чтобы проверить это, удалите созданный контейнер, а затем запустите сгенерированный файл myapp.yaml с помощью следующих команд:

```
$ podman rm -f --ignore myapp
$ podman play kube myapp.yaml
Pod:
b70aedd8105a6915428928a2b33fd7ecede632298088ea25d9db74ba9b16201e
Container:
a4d78fdfa5d8f751aafb06f3782e36a3aaf5b3804ca57694385de2ea1e400fe6
```

Kubernetes работает только с подами, а не с отдельными контейнерами. Команда `podman play kube` считывает YAML-файл и запускает под вместе с контейнером. На рис. 8.3 показано, что команда `play` создала под с вашим контейнером вместе с infra-контейнером.

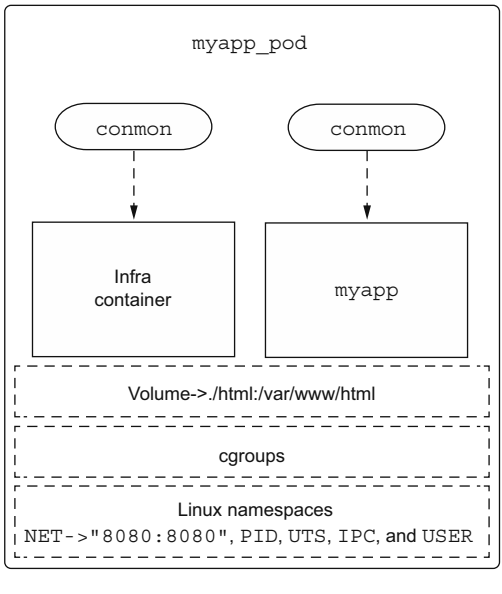


Рис. 8.3. Работающий myapp-pod с контейнером myapp и infra-контейнером

Команда `podman generate kube` создает под с именем `myapp-pod`, основанным на имени из файла myapp.yaml. Имена контейнеров генерируются путем добавления имени пода к имени контейнера: `myapp-pod-myapp`. Если в YAML-файле определены дополнительные контейнеры, то они должны быть обозначены подобным же образом:

```
$ cat myapp.yaml
...
  name: myapp-pod
spec:
  containers:
  - args:
    name: myapp
```

С помощью команды `podman pod ps` можно вывести на экран запущенные в системе поды. Добавьте параметр `--ctr-names`, чтобы также вывести список контейнеров, запущенных внутри данного пода:

```
$ podman pod ps --ctr-names
POD ID NAME STATUS CREATED INFRA ID NAMES
b70aedd8105a myapp-pod Running 1 day ago b7a276c62c1d
➥ myapp-pod-myapp,b70aedd8105a-infra
```

Используя следующую команду, сравните два запущенных контейнера:

```
$ podman ps
CONTAINER ID    IMAGE             COMMAND           CREATED
➥ STATUS             PORTS     NAMES
b7a276c62c1d k8s.gcr.io/pause:3.5
➥ 3 minutes ago Up 3 minutes ago 0.0.0.0:8080->8080/tcp b70aedd8105a-infra
a4d78fdfa5d8 quay.io/rhatdan/myimage:latest /usr/bin/run-http...
➥ 3 minutes ago Up 3 minutes ago 0.0.0.0:8080->8080/tcp myapp-pod-myapp
```

Завершите работу пода и контейнера командой `podman pod stop`:

```
$ podman pod stop myapp-pod
b70aedd8105a6915428928a2b33fd7ecede632298088ea25d9db74ba9b16201e
```

Команда `podman play kube` обрабатывает гораздо более сложные YAML-файлы, в том числе с несколькими подами, томами и контейнерами. В предыдущем несложном примере можно завершить работу пода командой `podman pod stop`, но когда `podman play kube` генерирует несколько уникальных подов, остановить их оказывается немного сложнее.

8.3.1. Завершение работы подов и контейнеров на основании Kubernetes YAML-файла

Вы можете остановить каждый под, запущенный командой `podman play kube`, однако иногда требуется не только остановить эти поды и контейнеры, но

и удалить их из системы. Команда `podman play kube --down` уничтожает поды, которые были созданы предыдущим запуском `play kube`. Поды останавливаются и затем удаляются. При этом все созданные тома остаются нетронутыми. Завершите работу пода myapp.yaml, созданного в предыдущем примере:

```
$ podman play kube myapp.yaml --down
Pods stopped:
B70aedd8105a6915428928a2b33fd7ecede632298088ea25d9db74ba9b16201e
Pods removed:
b70aedd8105a6915428928a2b33fd7ecede632298088ea25d9db74ba9b16201e
```

Обратите внимание: Podman не только остановил под, но и удалил его. В этом можно убедиться с помощью команды `podman pod ps`:

```
$ podman pod ps
POD ID   NAME      STATUS      CREATED      INFRA ID       # OF CONTAINERS
```

Произошел возврат к состоянию, в котором можно снова запустить `podman play kube`, что приведет к созданию новых подов и контейнеров:

```
$ podman play kube myapp.yaml
Pod:
302b1d2c0048a49ea32c2e6ffa0e0549af199ab2bc32de285eef5da628efe28c
Container:
b9f080dc6e13b4a4c37fa66a9b727dbeb2af30f0c3824044aba8a46eebfe15c5
```

Это имитирует работу Kubernetes с подами и контейнерами. Kubernetes всегда создает новые поды и контейнеры и уничтожает их по завершении работы. Возможность генерировать все поды и контейнеры из YAML-файла и затем удалять их с помощью флага `--down` похожа на работу `docker-compose`. Преимущество Podman заключается в использовании для запуска подов и контейнеров того же YAML-файла, что и в случае многоузловой оркестрированной среды Kubernetes. Разработчики Podman также добавили в `podman play kube` возможность сборки образов, заданных в YAML-файле, — еще одна особенность `docker-compose`.

8.3.2. Создание образов с использованием YAML-файлов Podman и Kubernetes

Пользователи, рассматривающие `podman play kube` как замену `docker-compose`, просили добавить в Podman возможность собирать образы, а не только брать их из реестра контейнеров. Хотя Kubernetes не поддерживает такую возможность, разработчики Podman решили добавить флаг `--build` в `podman play kube`. Поскольку `podman build` способен обрабатывать файлы Containerfiles или Dockerfiles, доработка `podman play kube` была несложной.

Идея заключается в том, чтобы создать контейнерное приложение с помощью образа контейнера, который собирается по требованию. Обычный рабочий процесс Kubernetes требует от разработчиков создания образа с помощью `podman build` и отправки его в реестр контейнеров с помощью `podman push`, как вы уже знаете из главы 2. Затем образ можно получить из реестра с помощью команды `podman play kube`. Параметр `podman play kube --build` позволяет выполнить `podman build` внутри системы и сгенерировать образ по требованию, не обращаясь к реестру контейнеров.

> **ПРИМЕЧАНИЕ** Параметр `--build` недоступен для удаленного клиента Podman, поэтому его нельзя использовать на платформах Mac или Windows.

В данном примере необходимо пересоздать файл Containerfile из раздела 6.1.3:

```
$ cat > ./Containerfile << _EOF
FROM ubi8-init
RUN dnf -y install httpd; dnf -y clean all
RUN systemctl enable httpd.service
_EOF
```

Напомню, что Containerfile создает образ контейнера, в котором в качестве init-системы используется systemd, а служба HTTPD работает и слушает порт `80`.

Сначала удалите все поды и контейнеры:

```
$ podman pod rm --all --force
$ podman rm --all --force
```

Теперь пересоберите образ my-systemd:

```
$ podman build -t mysystemd.
STEP 1/3: FROM ubi8-init
STEP 2/3: RUN dnf -y install httpd; dnf -y clean all
Updating Subscription Management repositories.
Unable to read consumer identity
...
Successfully tagged localhost/mysystemd:latest
bb1634ce1457f2eb70f84af33599d211eae64cb5f951e40e91481b6e58b747bf
```

Используя пример кода из раздела 3.1, создайте на основе этого образа контейнер со смонтированным в нем каталогом ./html:

```
$ podman create --rm -p 8080:80 --name myapp -v ./html:/var/www/
➥ html:Z mysystemd
fec6de5716ac246613723a4cc26407005e0bc315affdc62b56883bd94acd795e
```

Теперь сгенерируйте YAML-файл Kubernetes с помощью `podman generate kube`:

```
$ podman generate kube myapp > myapp2.yaml
```

Обратите внимание, что на этот раз Podman сгенерировал YAML-файл с секцией volume для html:

```
$ cat myapp2.yaml
...
spec:
  containers:
  - image: localhost/mysystemd:latest
    ...
    volumeMounts:
    - mountPath: /var/www/html
    name: home-dwalsh-podman-html-host-0
    volumes:
    - hostPath:
      path: /home/dwalsh/podman/html
      type: Directory
      name: home-dwalsh-podman-html-host-0
```

Восстановите чистое окружение, удалив все поды с помощью команды `podman pod rm --all --force`. Удалив все контейнеры и образы с помощью команд `podman rm` и `podman rmi`, вы сможете начать работу с чистого листа:

```
$ podman pod rm --all --force
$ podman rm --all --force
fec6de5716ac246613723a4cc26407005e0bc315affdc62b56883bd94acd795e
$ podman rmi mysystemd
Untagged: localhost/mysystemd:latest
Deleted: bb1634ce1457f2eb70f84af33599d211eae64cb5f951e40e91481b6e58b747bf
Deleted: 70e0c1a7580089420267b5928210ad59fdd555603e647b462159ea94f97946f9
```

Чтобы образы были собраны, команда `podman play kube --build` требует наличия подкаталогов, соответствующих именам образов. Podman просматривает YAML-файл Kubernetes для всех образов, а затем ищет соответствующий подкаталог. Каждая директория рассматривается как контекстная и должна содержать Containerfile или Dockerfile. Затем Podman выполняет `podman build` для каждой поддиректории. Поскольку YAML-файлу необходим образ mysystemd, следует создать каталог mysystemd и поместить в него Containerfile:

```
$ mkdir mysystemd
$ mv Containerfile mysystemd/
```

Следующая команда `podman play kube --build` пересоберет образ контейнера и запустит под и контейнеры для вашего приложения:

```
$ podman play kube myapp2.yaml --build
STEP 1/3: FROM ubi8-init
STEP 2/3: RUN dnf -y install httpd; dnf -y clean all
Updating Subscription Management repositories.
...
```

```
--> 305bb9b8da1
Successfully tagged localhost/mysystemd:latest
305bb9b8da12db682b0eae93ad492e632d2ba43e03f6a6b68467d7429a8a2664
a container exists with the same name ("myapp") as the pod in your YAML file;
➥ changing podname to myapp-pod
Pod:
30739dd554acfeab66a9767301127bab0fe994461686f45a3a89b137c3954840
Container:
ce633ac4e7a1e4d08e0428a8401fcfc4ac75fbcca4be07bc167add6093a44afa
```

Podman пересобрал образ mysystemd на основе файла mysystemd/Containerfile и затем сгенерировал под `myapp-pod` и контейнер `myapp` для вашего приложения, даже не обращаясь к реестру контейнеров.

Если вы поделитесь YAML-файлом и каталогом mysystemd с другими пользователями, они смогут собрать и запустить ваше приложение с помощью Podman. Однако помните, что если приложение будет запускаться в Kubernetes, вам необходимо направить собранный образ в реестр контейнеров, а затем отредактировать YAML-файл, чтобы указать на образ в реестре.

Вы познакомились с интеграцией Podman с Kubernetes, и я хочу рассмотреть напоследок запуск Podman внутри контейнеров Podman и Kubernetes.

8.4. ЗАПУСК PODMAN ВНУТРИ КОНТЕЙНЕРА

Запуск Podman в контейнере или в кластере Kubernetes является распространенной задачей. Пользователям необходима возможность тестировать образы контейнеров и инструменты в рамках CI/CD-систем, в которых используются контейнеры. Достаточно часто приходится создавать образы контейнеров с помощью `podman build`. А иногда пользователям просто нужно протестировать более новую версию Podman, чем та, которая была выпущена в имеющемся у них дистрибутиве.

Podman может быть настроен множеством различных способов. Из-за этого пользователи испытывают сложности при выборе наиболее подходящего метода для запуска Podman в контейнере. Поэтому я и некоторые мои коллеги решили создать образ контейнера quay.io/podman/stable, который упрощает эту задачу. Как вы знаете, Podman может работать в двух различных режимах: rootful и rootless. По умолчанию контейнеры Podman запускаются как root в своем пользовательском пространстве имен. Чтобы приблизиться к пониманию того, как запустить Podman в контейнере, давайте сначала поэкспериментируем с запуском Podman внутри Podman. В табл. 8.1 описаны различные способы запуска контейнера внутри контейнера и привилегии (capabilities), необходимые для того, чтобы внутренний Podman мог выполнить контейнер.

Таблица 8.1. Требования к запуску Podman в контейнере

Режим хоста	Режим контейнера	Привилегии	Объяснение
Rootful	Rootful	`CAP_SYS_ADMIN`	Имеет полный доступ к пространству имен пользователя хоста
Rootful	Rootless	`CAP_SETUID` `CAP_SETGID`	Работает в отдельном пространстве имен пользователя, основанном на /etc/subuid и /etc/subgid внутри контейнера
Rootless	Rootful	Namespaced `CAP_SYS_ADMIN`	Имеет полный доступ к пользовательскому пространству имен
Rootless	Rootless	Namespaced `CAP_SETUID`, `CAP_SETGID`	Запускается в отдельном пользовательском пространстве имен, основанном на /etc/subuid и /etc/subgid внутри контейнера. Это пространство имен должно быть подмножеством того пространства имен, в котором выполняется команда Podman

8.4.1. Запуск Podman внутри Podman-контейнера

В первом примере вы запустите rootful Podman внутри rootless-контейнера. Команда `--privileged` необходима, так как для успешной работы Podman должен иметь возможность монтировать файловые системы. Когда Podman запускается от имени root, для монтирования требуется привилегия `CAP_SYS_ADMIN`, которая и задается опцией `--privileged`. Попробуйте это сделать, выполнив следующую команду:

```
$ podman run --privileged quay.io/podman/stable podman version
Trying to pull quay.io/podman/stable:latest...
Getting image source signatures
Copying blob b1f89b7294d7 done
...
Version: 4.1.0
API Version: 4.1.0
Go Version: go1.18.2
Built: Mon May 30 12:03:28 2022
OS/Arch: linux/amd64
```

Образ quay.io/podman/stable также сконфигурирован для запуска rootless Podman внутри контейнера Podman. Активировать такое поведение можно, добавив запуск от имени пользователя Podman с помощью параметра `--user podman`. В этом режиме Podman внутри контейнера нуждается в `CAP_SETUID` и `CAP_SETGID` для

настройки пространства имен пользователя. К счастью, Podman предоставляет такой доступ по умолчанию:

```
$ podman run --user podman quay.io/podman/stable podman version
```

Если вы действительно хотите ограничить среду контейнера, откажитесь от всех привилегий, кроме `CAP_SETUID` и `CAP_SETGID`, используя параметры `--cap-drop=all --cap-add CAP_SETUID,CAP_SETGID`:

```
$ podman run --cap-drop=all --cap-add CAP_SETUID,CAP_SETGID
➥ --user podman quay.io/podman/stable podman version
Version:    4.1.0
API Version:    4.1.0
Go Version:    go1.18.2
Built:    Mon May 30 12:03:28 2022
OS/Arch:    linux/amd64
```

Эти примеры, демонстрирующие запуск Podman внутри контейнера Podman, также легко выполняются с помощью Docker.

Обратите внимание, что Docker работает с фильтром seccomp, который блокирует системные вызовы unshare и mount. Необходимо отключить фильтрацию seccomp в Docker:

```
docker run –security-opt seccomp=unconfined ...
```

либо запустить Docker с фильтрами seccomp Podman:

```
docker run –security-opt seccomp=/usr/share/containers/seccomp.json ...
```

В этом разделе я рассмотрел интеграцию Podman с Kubernetes. В следующем разделе вы узнаете, как настроить Podman для работы внутри подов или контейнеров Kubernetes.

8.4.2. Запуск Podman внутри пода Kubernetes

Использование Podman для запуска контейнеров в Kubernetes является распространенным вариантом для систем CI/CD. Вы уже знаете, что для запуска Podman внутри контейнера требуется либо `CAP_SYS_ADMIN` (для rootful-контейнеров), либо `CAP_SETUID` и `CAP_SETGID` (для запуска в режиме rootless). Для работы контейнеров Podman почти всегда требуется более одного UID, особенно при запуске `podman build`. Многие проблемы с Podman возникали из-за пользователей Kubernetes, пытавшихся запустить Podman в ограниченном контейнере Kubernetes, имеющем только один UID, и без Linux-привилегий. Такие контейнеры применяются по умолчанию в OpenShift и во многих облачных средах Kubernetes. Но запуск такого контейнерного движка, как Podman, невозможен в средах без каких-либо Linux-привилегий и с доступом к более чем одному UID.

Вариантом, эквивалентным запуску rootful Podman с использованием образа quay.io/podman/stable внутри привилегированного контейнера Kubernetes, может быть запуск с помощью такого YAML-файла:

```
apiVersion: v1
kind: Pod
metadata:
 name: podman-priv
spec:
 containers:
   - name: priv
     image: quay.io/podman/stable
     args:
       - podman
       - version
     securityContext:
       privileged: true
```

Подобным же образом можно запустить Podman без root в контейнере Kubernetes с помощью следующего YAML-файла. Обратите внимание, что в качестве UID указывается `runAsUser: 1000`, а не пользователь `podman`. Kubernetes не поддерживает трансляцию имен пользователей внутри контейнеров в их UID:

```
apiVersion: v1
kind: Pod
metadata:
  name: podman-rootless
spec:
  containers:
  - name: rootless
  image: quay.io/podman/stable
  args:
    - podman
    - version
    securityContext:
      capabilities:
        add:
          - "SETUID"
          - "SETGID"
      runAsUser: 1000
```

ПРИМЕЧАНИЕ Множество других примеров работы Podman в контейнерах приводится в статьях, написанных мной и моей коллегой Урваши Мохнани (Urvashi Mohnani): «How to Use Podman inside of a Container» («Как использовать Podman внутри контейнера»), (https://www.redhat.com/sysadmin/podman-inside-container) и «How to Use Podman inside of Kubernetes» («Как использовать Podman внутри Kubernetes»), (https://www.redhat.com/sysadmin/podman-inside-kubernetes).

Как видите, запуск контейнеров Podman в Kubernetes достаточно прост, если вы понимаете требования Podman. В сообществе Kubernetes ведется постоянная работа по использованию преимуществ пользовательских пространств имен, упрощающих запуск контейнеров Podman в контейнерах Kubernetes и делающих их более безопасными.

РЕЗЮМЕ

- Команда `podman generate kube` позволяет без лишних трудов перенести локально запущенные поды и контейнеры в YAML-файл Kubernetes, пригодный для запуска в кластере Kubernetes.
- Эти YAML-файлы также могут быть использованы для генерации локальных подов и контейнеров с помощью команды `podman play kube`.
- Параметр `--down` позволяет команде `podman play kube` завершить работу всех подов и контейнеров, запущенных предыдущей командой `podman play kube`.
- Параметр `--build` позволяет команде `podman play kube` сгенерировать заданный в YAML-файле Kubernetes образ контейнера на основе файла Containerfile/Dockerfile, что избавляет от необходимости размещать этот образ в реестре контейнеров.
- `podman play kube` заменяет `docker-compose`, поскольку имеет тот же формат YAML, что и Kubernetes.
- Запуск Podman внутри контейнеров Podman и Kubernetes возможен, если вы следуете требованиям Podman к работе в ограниченной среде.

9 Podman как служба

В ЭТОЙ ГЛАВЕ

- ✓ Запуск Podman в качестве службы
- ✓ Поддержка службой Podman двух REST API
- ✓ Библиотеки Python podman-py и docker-py для управления контейнерами Podman
- ✓ Поддержка `docker-compose`
- ✓ Удаленное взаимодействие со службой Podman через командную строку
- ✓ Управление взаимодействием с удаленными экземплярами Podman по SSH

В предыдущих главах вы познакомились с командной строкой Podman. Сложность заключается в том, что иногда требуется работать с контейнерами из удаленной системы. Кроме того, для взаимодействия с контейнерами может потребоваться написать код на каком-нибудь скриптовом языке. Docker, будучи создан изначально как клиент-серверное приложение, поддерживает популярный удаленный API, что привело к созданию библиотек на языках Python и JavaScript

для доступа к его демону. Docker-py — широко распространенная библиотека на языке Python, используемая для взаимодействия с Docker-демоном.

Для управления контейнерами Docker создано множество систем CI/CD, графических интерфейсов и систем удаленного управления. Такие редакторы кода, как Visual Studio, даже имеют встроенные плагины, напрямую взаимодействующие с Docker API. Развитие инструментов, например `docker-compose`, привело к появлению нового языка программирования, который используется для оркестрации нескольких контейнеров на хосте путем взаимодействия с Docker-демоном.

Podman предоставляет подобные возможности и может быть запущен как служба. Запуск службы Podman поддерживается как в режиме rootless, так и в режиме rootful. В этой главе вы познакомитесь с этой службой и узнаете, как с ней взаимодействовать. Вы напишете на языке Python простую программу, которая использует библиотеки docker-py и более новые библиотеки podman-py для взаимодействия со службой Podman. Вы узнаете, как настроить удаленные инструменты на базе Docker, включая `docker-compose`, на реальное использование службы Podman при отсутствии демона Docker.

> **ПРИМЕЧАНИЕ** Служба Podman поддерживается только в ОС Linux. Версии Podman для Windows и Mac взаимодействуют со службой Podman через REST API, так как она запускает Linux-контейнеры. Дополнительную информацию о Podman для платформы Mac вы найдете в приложении E, а для Windows — в приложении F.

У команды Podman есть параметр `--remote`, позволяющий взаимодействовать со службой Podman как на локальной, так и (в большинстве случаев) на удаленной машине. Вы научитесь настраивать соединения Podman, чтобы сделать взаимодействие с удаленными сервисами простым и безопасным. Но прежде вам необходимо узнать, как включить службу Podman.

9.1. ЗНАКОМСТВО СО СЛУЖБОЙ PODMAN

Проект Podman поддерживает REST (или RESTful) API. Команда `podman system service` создает службу для прослушивания, которая отвечает на вызовы к API Podman. Служба может быть запущена в режиме rootful или rootless. Эта команда имеет дополнительный аргумент, позволяющий указать URI, по которому будет прослушиваться служба Podman. Например, URI unix:///tmp/podman.sock указывает Podman на прослушивание сокета /tmp/podman.sock домена UNIX. URI сокета tcp:localhost:10000 дает указание Podman прослушивать TCP-сокет, порт `10000`. По умолчанию Podman прослушивает сокет домена UNIX в каталоге /run (табл. 9.1).

Таблица 9.1. Местоположение по умолчанию для podman.socket

Режим	Местоположение по умолчанию
Rootful	unix:///run/podman/podman.sock
Rootless	unix://$XDG_RUNTIME_DIR/podman/podman.sock Пример unix:///run/user/1000/podman/podman.sock

ПРИМЕЧАНИЕ Если вы не знакомы с REST API или удаленными API в целом, рекомендую прочитать статью «What is a REST API?» компании Red Hat: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

Работа Podman в качестве службы в данном случае отличается от Docker с его централизованным демоном. Самое большое отличие — в том, что команда Podman может выполняться без службы и взаимодействовать с контейнерами и образами, создаваемыми этой службой. Другие контейнерные инструменты могут взаимодействовать с хранилищем и контейнерами, не проходя через эту службу. Кроме того, служба завершает работу при отсутствии соединений с ней. Можно даже запустить несколько служб одновременно на одном и том же устройстве (хотя я не рекомендую так делать). Демон Docker заставляет все взаимодействия с контейнерами и образами проходить через него. В табл. 9.1 показаны местоположения, где служба Podman по умолчанию прослушивает входящие соединения.

Хотя служба Podman может быть настроена и на работу через TCP-сокет, я предупреждаю, что в этом случае следует действовать очень осторожно из-за отсутствия авторизации и дополнительных средств защиты, препятствующих получению доступа злоумышленниками. Для получения удаленного доступа к службе Podman рекомендуется использовать службу SSH.

Служба Podman была разработана как запускаемая по требованию и завершающаяся через 5 секунд после последнего соединения. Такое ограничение времени позволяет избежать длительной работы демона, потребляющего системные ресурсы даже тогда, когда сервис не используется. Служба Podman может запускать отдельный процесс для каждого соединения, но именно это может стать узким местом. Протестируйте сказанное, выполнив приведенную ниже команду. Через 5 секунд вы увидите, что команда завершилась. Если у вас есть активные соединения со службой, она будет продолжать работать:

```
$ podman system service
```

С помощью параметра `--time` можно указать тайм-аут для завершения работы (в секундах). Если указать параметр `--time 0`, команда `podman system service`

будет выполняться до тех пор, пока вы ее не остановите. Большинство пользователей никогда не взаимодействуют напрямую с системной службой Podman для ее активации, а полагаются для управления ею на службы systemd.

9.1.1. Службы systemd

Для запуска Podman в качестве службы предусмотрено несколько юнит-файлов systemd. Поскольку Podman не был задуман как демон и разработчики не хотели иметь постоянно работающий демон, они решили воспользоваться активацией через сокеты systemd. Это позволяет запускать службу Podman по требованию. На рис. 9.1 показано, как systemd прослушивает сокет Podman, а затем запускает службу Podman при установлении соединения.

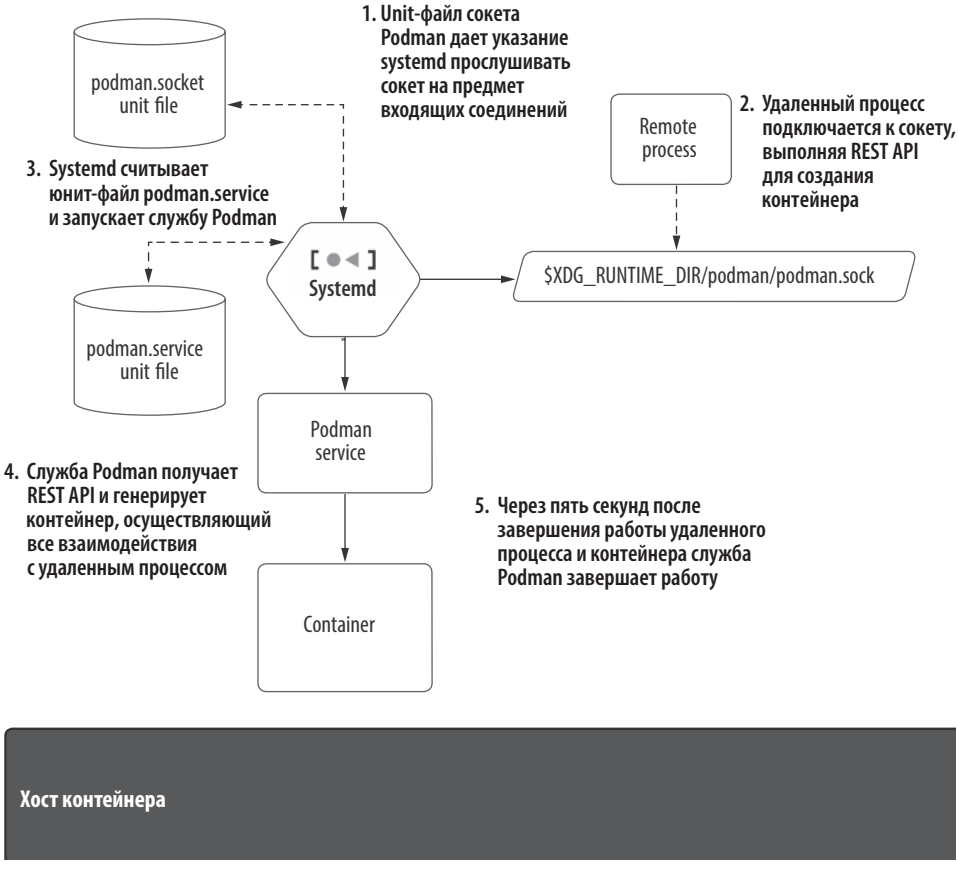


Рис. 9.1. Служба Podman, работающая под управлением systemd

В состав пакета Podman входят два юнит-файла podman.socket: один для rootful, другой для rootless-режимов. В табл. 9.2 демонстрируется расположение файлов сокетов systemd для использования в режимах rootful и rootless.

Таблица 9.2. Юнит-файлы сокетов Podman

Режим	Файл сокета Systemd
Rootful	/usr/lib/systemd/system/podman.socket
Rootless	/usr/lib/systemd/user/podman.socket

Эти две службы указывают systemd на необходимость прослушивания по умолчанию сокета UNIX домена, указанного в табл. 9.1. Когда процесс подключается к сокету, systemd запускает соответствующую службу, которая выполняет команду `podman system service`. Затем Systemd передает сокет службе. После того как служба Podman выполнит API-запрос, она ожидает новое соединение. Если в течение 5 секунд соединения не происходит, Podman завершает работу, освобождая использовавшиеся ресурсы. При появлении нового соединения systemd повторяет процесс и запускает новый экземпляр службы Podman.

На протяжении этой главы вы будете взаимодействовать со службой Podman, поэтому вам необходимо запустить ее. Включить и запустить сокет Podman на своей машине можно с помощью параметра `--user`, который дает указание systemd включить пользовательскую службу (или службу в режиме rootless):

```
$ systemctl --user enable podman.socket
Created symlink
➥ /home/dwalsh/.config/systemd/user/sockets.target.wants/podman.socket ?
➥ /usr/lib/systemd/user/podman.socket.
$ systemctl --user start podman.socket
```

В вашем `XDG_RUNTIME_DIR` был создан файл podman.sock:

```
$ ls $XDG_RUNTIME_DIR/podman/podman.sock
/run/user/3267/podman/podman.sock
```

В этот момент systemd слушает сокет, а процесс Podman не запущен. Когда службе приходит пакет, systemd запускает процесс службы Podman для обработки соединения.

Чтобы поэкспериментировать с этой службой, можно выполнить следующую команду `curl` для проверки версии:

```
$ curl -s --unix-socket $XDG_RUNTIME_DIR/podman/podman.sock
➥ http://d/v1.0.0/libpod/version | jq
{
  "Platform": {
  "Name": "linux/amd64/fedora-35"
```

```
  },
  "Components": [
  {
    "Name": "Podman Engine",
    "Version": "4.0.0-dev",
    "Details": {
      "APIVersion": "4.0.0-dev",
        "Arch": "amd64",
        "BuildTime": "2022-01-04T13:42:14-05:00",
        "Experimental": "false",
      "GitCommit": "66ffbc845d1f0fd5c29611ac3f09daa24749dc1e-dirty",
      "GoVersion": "go1.16.12",
      "KernelVersion": "5.15.10-200.fc35.x86_64",
      "MinAPIVersion": "3.1.0",
      "Os": "linux"
    }
  },
  {
    "Name": "Conmon",
    "Version": "conmon version 2.0.30, commit: ",
    "Details": {
      "Package": "conmon-2.0.30-2.fc35.x86_64"
    }
  },
  {
    "Name": "OCI Runtime (crun)",
    "Version": "crun version 1.4\ncommit:
3daded072ef008ef0840e8eccb0b52a7efbd165d\nspec: 1.0.0\n+SYSTEMD
➥ +SELINUX +APPARMOR +CAP +SECCOMP +EBPF +CRIU +YAJL",
    "Details": {
      "Package": "crun-1.4-1.fc35.x86_64"
    }
  }
  ],
  "Version": "4.0.0-dev",
  "ApiVersion": "1.40",
  "MinAPIVersion": "1.24",
  "GitCommit": "66ffbc845d1f0fd5c29611ac3f09daa24749dc1e-dirty",
  "GoVersion": "go1.16.12",
  "Os": "linux",
  "Arch": "amd64",
  "KernelVersion": "5.15.10-200.fc35.x86_64",
  "BuildTime": "2022-01-04T13:42:14-05:00"
}
```

Служба запущена, и пришло время изучить API.

9.2. API, ПОДДЕРЖИВАЕМЫЕ PODMAN

Служба Podman предоставляет два API через один и тот же сокет (табл. 9.1). Режим совместимости определяет использование последней версии Docker

API, реализующей все эндпоинты, кроме Swarm API. Разработчики Podman расценивают любую проблему, связанную с расхождением с Docker API, как баг. Если API работает с демоном Docker, то он должен работать и со службой Podman.

Podman Libpod API обеспечивает поддержку уникальных возможностей Podman, таких как поды. Хотя было бы замечательно, если бы все проекты поддерживали нативный Libpod API, переход на него требует времени, а для уже не поддерживаемых проектов, основанных на Docker API, это может оказаться невозможным.

Я рекомендую всем новым пользователям Podman работать с Libpod API. Но если у вас имеется унаследованный код или вы собираетесь разрабатывать код, который будет работать и с Podman, и с Docker, вам надо использовать совместимый API. В табл. 9.3 приведены два варианта REST API, предоставляемых Podman.

Таблица 9.3. API, поддерживаемые Podman

Режим	Описание	Документация
Compatibility (Режим совместимости)	Уровень совместимости, обеспечивающий поддержку API Docker v1.40	https://docs.docker.com/engine/api/
Libpod	Нативный для Podman слой Libpod	https://docs.podman.io/en/latest/_static/api.html

Самый простой способ взаимодействия с удаленным API — команда `curl`. Изучите список доступных образов с помощью команды `curl` и команды `jq`, получив красивый вывод JSON-кода. Также обратите внимание на поле `libpod` в URL. Это поле дает указание Podman использовать его собственный API.

Листинг 9.1. Вывод по умолчанию при подключении curl к сокету Podman

```
$ curl -s --unix-socket $XDG_RUNTIME_DIR/podman/podman.sock
➥ http://d/v1.0.0/libpod/images/json | jq
[
  {
  "Id":
"Sha256:2c7e43d880382561ebae3fa06c7a1442d0da2912786d09ea9baaef87f73c29ae",
  "ParentId": "",
  "RepoTags": [
    "quay.io/rhatdan/myimage:latest"  <-- Образ, с которым вы работаете
  ],
...
  }
]
```

Можно также запустить API Docker, исключив поле `libpod`. Для этой команды вы получите такой же вывод, поскольку у обоих API он одинаковый:

```
$ curl -s --unix-socket $XDG_RUNTIME_DIR/podman/podman.sock
➥ http://d/v1.0.0/images/json | jq
[
  {
  "Id":
"Sha256:2c7e43d880382561ebae3fa06c7a1442d0da2912786d09ea9baaef87f73c29ae",
  "ParentId": "",
  "RepoTags": [
    "quay.io/rhatdan/myimage:latest"
  ],
...
  }
]
```

API будут различаться при выводе списка подов: поскольку Docker не поддерживает концепцию подов, в `compat` API отсутствуют соответствующие интерфейсы.

Создайте под для теста, выполнив следующую команду:

```
$ podman pod create --name mypod
116291543d5691c597132ec73a428f29f2c1f71a65fdfbaca17eb5440a5d47f6
```

Затем воспользуйтесь Libpod pods или JSON API для просмотра JSON, относящегося к только что созданному поду:

```
$ curl -s --unix-socket $XDG_RUNTIME_DIR/podman/podman.sock
➥ http://d/v1.0.0/libpod/pods/json | jq
[
  {
    "Cgroup": "user.slice",
    "Containers": [
      {
      "Id": "8eeceeb4fd6aa3897e05b5361b5c27c6e98bc29707484f95994f49437536599e",
      "Names": "4b10a21c5b8c-infra",
      "Status": "running"
      }
    ],
    "Created": "2022-01-05T06:51:52.604528462-05:00",
    "Id": "4b10a21c5b8c2b4f8a598de1eace7b94918d813055891276c2472df856a7fbc1",
    "InfraId":
➥ "8eeceeb4fd6aa3897e05b5361b5c27c6e98bc29707484f95994f49437536599e",
    "Name": "test_pod",
    "Namespace": "",
    "Networks": [],
    "Status": "Running",
    "Labels": {}
  },
  {
    "Cgroup": "user.slice",
```

```
    "Containers": [
      {
      "Id": "7a7405a31917da7bde01a6000809e0ee12f40b69fc76963d87a8ae254b34d8c7",
      "Names": "e10eb9303705-infra",
      "Status": "configured"
      }
    ],
    "Created": "2022-01-05T09:18:01.648324833-05:00",
    "Id": "e10eb930370592834fc168a7460fabe9b3e0e20a54b48a2bf3236cecd75f8138",
    "InfraId":
➥ "7a7405a31917da7bde01a6000809e0ee12f40b69fc76963d87a8ae254b34d8c7",
    "Name": "mypod",
    "Namespace": "",
    "Networks": [],
    "Status": "Created",
    "Labels": {}
  }
]
```

Если вы попробуете выполнить тот же запрос к эндпоинту Docker API, он завершится с ошибкой Not Found:

```
$ curl -s --unix-socket $XDG_RUNTIME_DIR/podman/podman.sock
➥ http://d/v1.0.0/pods/json
Not Found
```

Объясняется это тем, что Docker API и сам Docker не поддерживают работу с подами. Несмотря на то что многие тесты с API можно проводить напрямую с помощью таких инструментов, как `curl`, для взаимодействия с API лучше использовать языки высокого уровня, например Python.

9.3. БИБЛИОТЕКИ PYTHON ДЛЯ РАБОТЫ С PODMAN

Python является, пожалуй, самым популярным скриптовым языком на платформах Linux. Практически в каждой системе Linux Python установлен по умолчанию. Как и в случае с API, существуют две очень похожие библиотеки Python: библиотека docker-py, работающая с библиотекой compatibility, и podman-py, которая поддерживает более новый API Libpod. В данном разделе вы встретите ряд команд Python (так что некоторое знание этого языка желательно), но вам будет достаточно легко следовать за изложением, если у вас есть хотя бы ограниченный опыт.

9.3.1. Использование docker-py с Podman API

Наиболее популярным пакетом Python для взаимодействия с контейнерами является docker-py (https://github.com/docker/docker-py). Docker-py — это библиотека

Python, изначально используемая для взаимодействия с демоном Docker, но она также может взаимодействовать со службой Podman в режиме совместимости. Библиотека Docker-py позволяет запускать те же контейнеры, что и команда Podman, только на языке Python.

Тысячи инструментов, построенных на docker-py, существуют и применяются в продакшене. Эти инструменты используются для создания CI/CD-систем, а также графических интерфейсов, средств управления и отладки. Для этих команд можно использовать Podman `compat` API, который отлично работает с docker-py.

docker-py обычно устанавливается с помощью `apt-get` или `dnf install`, он также доступен через PyPI. Ознакомьтесь с командами установки для вашей платформы Linux. В системах на базе RPM пакет называется `python-docker`. В моей системе на базе Red Hat я устанавливаю его с помощью следующей команды `dnf`:

```
$ sudo dnf install -y python-docker
```

Установив docker-py, можно приступать к его использованию для взаимодействия со службой Podman. Представьте, что вы хотите создать Python-скрипт для взаимодействия со службой Podman, чтобы получить список доступных на данный момент образов. Обратите внимание, что мне пришлось изменить URL-адрес `DockerClient`, чтобы он указывал на сокет Podman. Возможно, вам придется изменить расположение сокета podman.sock в вашей системе:

```
$ cat > images.py << _EOF
import docker
client=docker.DockerClient(base_url='unix:/run/user/1000/podman/podman.sock')
print(client.images.list(all=True))
_EOF
```

Запустите скрипт images.py, и вы увидите образы, установленные на вашем компьютере:

```
$ python images.py
[<Image: 'quay.io/rhatdan/myimage:latest'>, <Image: 'k8s.gcr.io/pause:3.5'>]
```

Полностью указывать путь к сокету Podman внутри Python-скрипта не очень удобно, но к счастью, инструменты Docker поддерживают специальную переменную окружения `DOCKER_HOST`. Вы можете установить `DOCKER_HOST` так, чтобы она указывала на сокет, обеспечивающий работу Docker API.

Сначала установите переменную окружения `DOCKER_HOST` для указания на podman.sock:

```
$ export DOCKER_HOST=unix://$XDG_RUNTIME_DIR/podman/podman.sock
```

Теперь измените скрипт так, чтобы он использовал функцию `docker.from_env()`:

```
$ cat > images.py << _EOF
import docker
client=docker.from_env()
print(client.images.list(all=True))
_EOF
```

Запустив новый скрипт, вы обнаружите, что он использует переменную окружения `DOCKER_HOST` для обнаружения сокета службы Podman:

```
$ python images.py
[<Image: 'quay.io/rhatdan/myimage:latest'>, <Image: 'k8s.gcr.io/pause:3.5'>]
```

> **ПРИМЕЧАНИЕ** Во многих дистрибутивах Linux пакет `podman-docker` доступен локально. При инсталляции пакета устанавливается скрипт, который перенаправляет команды Docker на выполнение команд Podman. Он также связывает все man-страницы Docker с man-страницами Podman. Наконец, он устанавливает символическую ссылку между docker.sock и podman.sock для rootful-контейнеров, позволяя инструментам Docker использовать /var/run/podman/podman.sock без каких-либо изменений в окружении.

Самое замечательное в том, что этот трюк с `DOCKER_HOST` применим к большинству скриптов docker-py, которые были написаны за долгие годы, и вам удастся без труда переключить свои скрипты с использования демона Docker на использование службы Podman. Если вы хотите задействовать расширенные возможности Podman, необходим пакет podman-py.

9.3.2. Использование podman-py с Podman API

Podman-py (https://github.com/containers/podman-py), как и docker-py, представляет собой библиотеку Python, применяемую для взаимодействия со службой Podman. Библиотека podman-py является более новой, чем docker-py, и поддерживает все расширенные возможности Podman через Libpod API.

Python-библиотека Podman использует стандартное расположение сокета podman.sock и подключается к нему автоматически. При запуске от имени пользователя, не являющегося root, библиотека подключается к rootless-сокету, расположенному в каталоге /run/user/$UID/podman/podman.sock. При запуске Python с библиотекой Podman от имени root происходит автоматическое подключение к /run/podman/podman.sock.

Подобно установке docker-py, я могу установить библиотеку podman-py в своей системе с помощью пакета `python-podman`:

```
$ sudo dnf install -y python-podman
Last metadata expiration check: 0:27:40 ago on Sun 19 Jun 2022 02:14:49 PM EDT.
Dependencies resolved.
...
Installed:
  python3-podman-3:4.0.0-1.fc36.noarch
Complete!
```

Создадим теперь функционально похожий скрипт podman-images.py, используя библиотеку podman-py. На этот раз вам не нужно беспокоиться о расположении сокета Podman. Библиотека podman-py подключается к расположению по умолчанию:

```
$ cat > podman-images.py << _EOF
import podman
client=podman.PodmanClient()
print(client.images.list())
_EOF
```

Запустив скрипт, вы увидите те же результаты, что и в примере с docker-py, но эта библиотека использует API Libpod:

```
$ python podman-images.py
[<Image: 'quay.io/rhatdan/myimage:latest'>, <Image: 'k8s.gcr.io/pause:3.5'>]
```

Если вам нужны расширенные функции, например информация обо всех подах из базы данных Podman, вызовите `pod.lists()` и выполните итерацию по каждому поду:

```
$ cat >> podman-images.py << _EOF
for i in client.pods.list():
    print(i.attrs)
_EOF
```

Теперь скрипт показывает и образы, и информацию о подах.

```
$ python podman-images.py
[<Image: 'quay.io/rhatdan/myimage:latest'>, <Image: 'k8s.gcr.io/pause:3.5'>]
{'Cgroup': 'user.slice', 'Containers': [{'Id':
➥ 'f8679839c25729eb422d38e505ae3a4b7ffe18942e2f77a997bd388e0f52313e',
➥ 'Names': '116291543d56-infra', 'Status': 'configured'}], 'Created':
➥ '2021-12-14T06:44:04.56055485-05:00', 'Id':
    '116291543d5691c597132ec73a428f29f2c1f71a65fdfbaca17eb5440a5d47f6',
➥ 'InfraId':
    'f8679839c25729eb422d38e505ae3a4b7ffe18942e2f77a997bd388e0f52313e',
➥ 'Name': 'mypod', 'Namespace': '', 'Networks': None, 'Status':
➥ 'Created', 'Labels': {}}
```

С помощью этих библиотек Python можно приступить к созданию Python-версии Podman, которая будет взаимодействовать с удаленным сокетом.

9.3.3. Какую библиотеку Python выбрать

Разработчики библиотеки podman-py взяли за основу библиотеку docker-py, чтобы облегчить процесс перехода на Podman. Если вам нужно создать приложение, работающее с Podman и Docker, то docker-py оказывается единственным возможным вариантом, поскольку podman-py не работает с Docker. Если вы хотите использовать расширенные возможности Podman, вам придется применять podman-py. Podman-py находится в стадии активной разработки, а docker-py имеет огромную базу инсталляций. Podman-py «без подгонки» работает с rootful-и rootless-службами Podman, в то время как при использовании docker-py необходимо установить переменную окружения `DOCKER_HOST` для указания на podman.socket. В табл. 9.4 приведено сравнение возможностей библиотек podman-py и docker-py, которое поможет вам разобраться, когда следует выбирать ту или иную библиотеку.

Таблица 9.4. Podman-py и docker-py

Поддержка	Podman-py	Docker-py
Служба Podman	✓	✓
Docker-демон	✗	✓
Поддержка подов	✓	✗
Расширенные функции Podman	✓	✗

Используя Python-библиотеки docker-py и podman-py для взаимодействия с демонами и службами контейнерных движков, инженеры разработали более высокоуровневые инструменты оркестрации и управления контейнерами. Наиболее популярным из них является `docker-compose`.

9.4. ИСПОЛЬЗОВАНИЕ DOCKER-COMPOSE СО СЛУЖБОЙ PODMAN

В предыдущих главах мы рассмотрели управление контейнерами с помощью командной строки Podman, а также управление несколькими контейнерами

с помощью Kubernetes YAML, запущенного командой `podman play kube`. Вы также познакомились с запуском контейнеров с помощью Kubernetes. В этом разделе я объясню, как работать с еще одним инструментом оркестрации — `dockercompose` (https://docs.docker.com/compose), который часто называют просто `compose`.

`Compose` — один из самых популярных инструментов для запуска контейнеров. Инструмент `compose` появился еще до Kubernetes и ориентирован на организацию работы нескольких контейнеров на одном узле, в то время как Kubernetes организует работу нескольких контейнеров на нескольких узлах. `compose`, как и Kubernetes, использует YAML-файл для описания контейнеров. Одной из причин создания `compose` было то, что построение сложных команд для запуска нескольких контейнеров вызывало затруднения. Использование такого структурированного языка, как YAML, упрощает поддержку запуска сложных приложений с несколькими контейнерами на одном узле.

У `compose` огромная пользовательская база, и вполне вероятно, что вам понадобится запустить `compose` YAML-файл в своей инфраструктуре. Если вы считаете, что такого не случится, пропустите данный раздел.

Инструмент `compose` был написан с использованием docker-py и запускает контейнеры с помощью Docker REST API. Поскольку Podman поддерживает `compat` REST API, он также поддерживает `docker-compose` для запуска контейнеров Podman. Podman работает как в режиме rootless, так и в режиме rootful, поэтому вы можете применять `docker-compose` даже для запуска контейнеров Podman без root.

В оставшейся части этого раздела вы создадите YAML-файл, чтобы получить представление о работе команды `compose` со службой Podman. Сначала необходимо установить `docker-compose`. В моей системе Fedora это можно сделать с помощью следующей команды:

```
$ sudo dnf -y install docker-compose
```

Убедитесь, что запущена служба Podman systemd с активацией через сокеты, выполнив следующую команду:

```
$ systemctl –user start Podman.socket
```

Для проверки того, что служба запущена, обратитесь к эндпоинту ping и посмотрите, будет ли получен ответ. Этот шаг должен быть успешно выполнен, чтобы можно было двигаться дальше.

```
$ curl -H "Content-Type: application/json" --unix-socket
➥ $XDG_RUNTIME_DIR/podman/podman.sock http://localhost/_ping
OK
```

Поскольку `docker-compose` поддерживает переменную окружения `DOCKER_HOST`, убедитесь, что она установлена, с помощью следующей команды:

```
$ export DOCKER_HOST=unix://$XDG_RUNTIME_DIR/podman/podman.sock
```

Как было сказано ранее в этом разделе, `compose` поддерживает собственный YAML-файл, который отличается от YAML-файла Kubernetes, описанного в главе 8.

Создайте каталог с именем example и перейдите в него. Переместите каталог html, который вы ранее использовали, в каталог example:

```
$ mkdir example
$ mv ./html example
$ cd example
```

Необходимо создать файл docker-compose.yaml в каталоге example. Этот YAML-файл создаст контейнер с именем `myapp` на основе quay.io/rhatdan/myimage:latest. Настройте контейнер на использование томов из каталога хоста ./html, а также встроенного тома `myapp_vol`, который используется только для примера:

```
cat > docker-compose.yaml << _EOF
version: "3.7"
services:
  myapp:
    image: quay.io/rhatdan/myimage:latest
    volumes:
      - ./html:/var/www/html
      - myapp_vol:/vol
    ports:
      - 8080:80
volumes:
  myapp_vol: {}
_EOF
```

Теперь очистите имеющиеся в системе образы и контейнеры, чтобы убедиться, что вы начинаете с чистого листа. Для этого выполните следующие команды:

```
$ podman pod rm --all --force
$ podman rm --all --force
$ podman rmi --all --force
$ podman volume rm --all --force
```

Чтобы выяснить, как `compose` взаимодействует со службой Podman, запустите контейнер командой `compose`. Обратите внимание, что `compose` указывает Podman на необходимость загрузки образа. Затем `compose` дает Podman указание на создание контейнера с именем `example_myapp_1` и создание тома с именем `example_myapp_vol`, который будет смонтирован в контейнер вместе с каталогом ./html.

Листинг 9.2. Результат выполнения docker-compose с сокетом Podman

```
$ docker-compose up
Pulling myapp (quay.io/rhatdan/myimage:latest)...  ◄── Загрузка образа myimage
59bf1c3509f3: Download complete
c059bfaa849c: Download complete
Creating example_myapp_1 ... done  ◄── Создание контейнера example_myapp_1
Attaching to example_myapp_1
```

На другом терминале выполните команду `podman ps`:

```
$ podman ps --format "{{.ID}} {{.Image}} {{.Ports}} {{.Names}}"
230fce823ff6 quay.io/rhatdan/myimage:latest 0.0.0.0:8080->80/tcp
➥ example_myapp_1
```

Теперь проверьте, создал ли Podman том:

```
$ podman volume ls
DRIVER   VOLUME NAME
local    example_myapp_vol
```

Вернитесь в исходное окно и нажмите Ctrl-C для остановки `docker-compose`:

```
^CGracefully stopping... (press Ctrl+C again to force)
Stopping example_myapp_1 ... done
```

Это приведет к остановке контейнера:

```
$ podman ps --format "{{.ID}} {{.Image}} {{.Ports}} {{.Names}}"
```

Выполнив команду `podman ps -a`, вы увидите, что контейнер все еще существует, но не запущен:

```
$ podman ps -a --format "{{.ID}} {{.Image}} {{.Ports}} {{.Names}}"
230fce823ff6 docker.io/library/alpine:latest 0.0.0.0:8080->80/tcp
➥ example_myapp_1
```

Если вы запустите сейчас `docker-compose down`, он даст указание Podman удалить контейнер из системы:

```
$ docker-compose down
Removing example_myapp_1 ... done
Removing network example_default
```

Убедитесь, что все контейнеры удалены, снова выполнив команду `podman ps -a`:

```
$ podman ps -a --format "{{.ID}} {{.Image}} {{.Ports}} {{.Names}}"
```

Как видите, Podman прекрасно работает с `docker-compose` при оркестрации контейнеров.

СОВЕТ Несмотря на то что `docker-compose` отлично работает с сервисом Podman, я полагаю, что начинать новый проект лучше с Kubernetes YAML и `podman play kube`, поскольку это упрощает перенос ваших контейнеров в Kubernetes.

Как вы уже убедились, служба Podman позволяет удаленным процессам управлять вашими подами и контейнерами. Даже сама команда Podman может использоваться в качестве клиента и взаимодействовать со службой Podman.

9.5. PODMAN --REMOTE

По мере масштабирования контейнерных приложений вам, вероятно, потребуется запускать их на нескольких машинах. Для управления окружением можно либо подключаться к каждой машине по `ssh` и локально запускать команды Podman, либо написать код для использования библиотеки Python, описанной в разделе 9.4. Разработчики Podman встроили поддержку клиентов в команду Podman. Команда `podman` дает возможность напрямую подключаться к удаленным сервисам Podman и управлять контейнерным окружением на удаленных машинах.

Команда `podman` имеет специальный параметр `--remote`, позволяющий ей взаимодействовать со службой Podman, активируемой через сокет. Вместо того чтобы выполнять команды и контейнеры в качестве дочернего процесса Podman, она взаимодействует с сервисом через REST API.

Поскольку Podman — это инструмент для запуска контейнеров Linux, полноценно команда `podman` может быть выполнена только в Linux. Разработчики Podman постарались обеспечить поддержку других операционных систем, по крайней мере в клиентском режиме. Для поддержки запуска Podman на машинах, отличных от Linux, Podman может быть собран двумя различными способами. До сих пор вы работали с полноценным Podman, который имеет параметр `--remote`. Исполняемый файл Podman может быть скомпилирован с поддержкой только взаимодействия со службой Podman. Скомпилированный таким образом Podman часто называют `podman-remote`. Команда `podman-remote` поставляется в некоторых операционных системах, таких как Mac и Windows (более подробно рассматривается в приложениях E и F). Если во время чтения этой книги вы тестировали Podman на машине с Mac или Windows, то вы уже использовали команду `podman-remote`, которая прозрачным образом взаимодействует со службой Podman, запущенной в виртуальной машине или на другом компьютере.

9.5.1. Локальные подключения

Я уже упоминал, что команда `podman --remote` по умолчанию подключается к локальному сокету podman.socket (рис. 9.2). Попробуйте выполнить команду `podman --remote` со службой Podman, которую вы запустили в разделе 9.1.1. Обратите внимание, что `podman --remote version` показывает версию как клиента Podman, так и сервера Podman. В данном случае это один и тот же исполняемый файл.

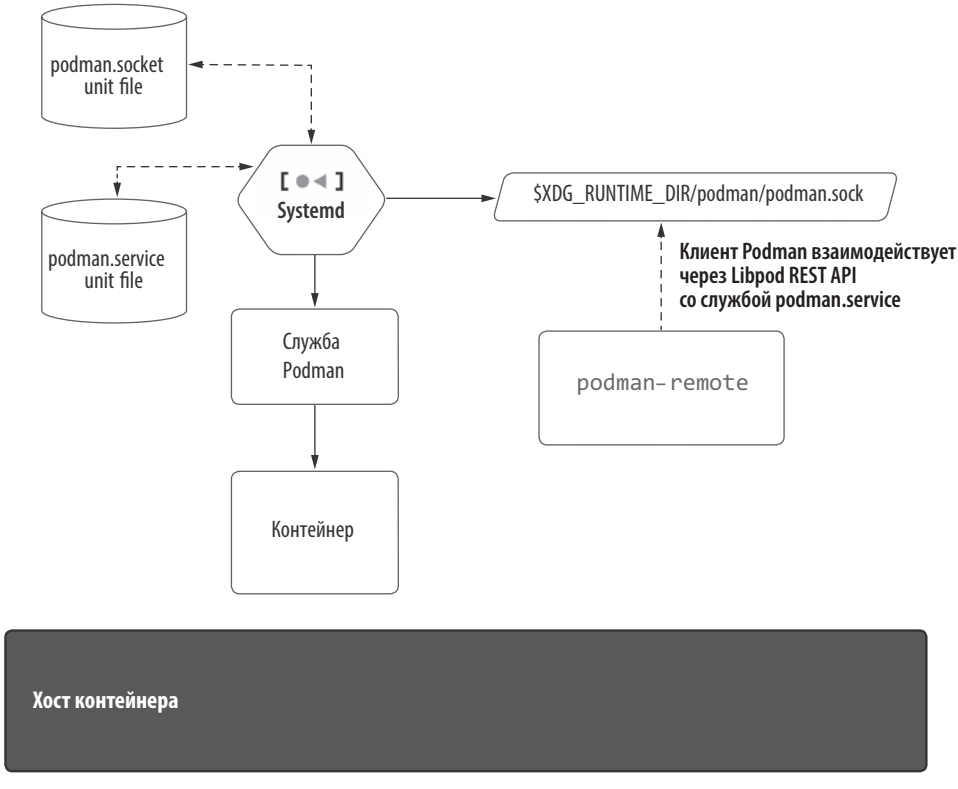


Рис. 9.2. podman --remote подключается к локальному сокету podman.socket

Листинг 9.3. Вывод команды podman --remote, которая обращается к API для получения версии

```
$ podman --remote version
Client:
Version:     4.1.0  ◄——  Версия клиента Podman
```

```
API Version: 4.1.0
Go Version:  go1.18.2
Built:       Sun Jun 19 07:35:42 2022
OS/Arch:     linux/amd64
Server:
Version:     4.1.0  ◄——  Версия сервера Podman
API Version: 4.1.0
Go Version:  go1.18.2
Git Commit:  a2b78b627f0a9deef83a5b5e4ecffc9cdb5a72b1-dirty
Built:       Sun Jun 19 07:35:42 2022
OS/Arch:     linux/amd64
```

Для запуска контейнера можно использовать те же команды:

```
$ podman --remote run ubi8 echo hi
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/
➥ 000-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest...
..
hi
```

Как вы понимаете, в этом режиме он не так уж и полезен, поскольку можно запустить Podman без параметра `--remote` и управлять тем же контейнерным окружением. Локальные соединения используются в основном для тестирования API, особенно в системах непрерывной интеграции (Continuous Integration, CI). Гораздо интереснее использовать `podman --remote` для связи с действительно удаленными машинами.

9.5.2. Удаленные подключения

Основное назначение команды `podman --remote` — дать возможность управлять подами и контейнерами на отдельной машине с помощью службы Podman. Установите Podman на Linux-машину или виртуальную машину, на которой также запущен демон SSH. В локальной операционной системе при запуске команды Podman происходит подключение к серверу по SSH. Затем он подключается к службе Podman через активацию сокета в systemd и взаимодействует с нашим REST API, как показано на рис. 9.3.

Интерфейс командной строки Podman с параметром `--remote` в точности совпадает с интерфейсом обычных команд Podman. При выполнении команд Podman создается впечатление, что вы запускаете контейнеры локально, однако процессы контейнеров работают на удаленной машине. В удаленном режиме не поддерживаются несколько параметров, они перечислены в табл. 9.5.

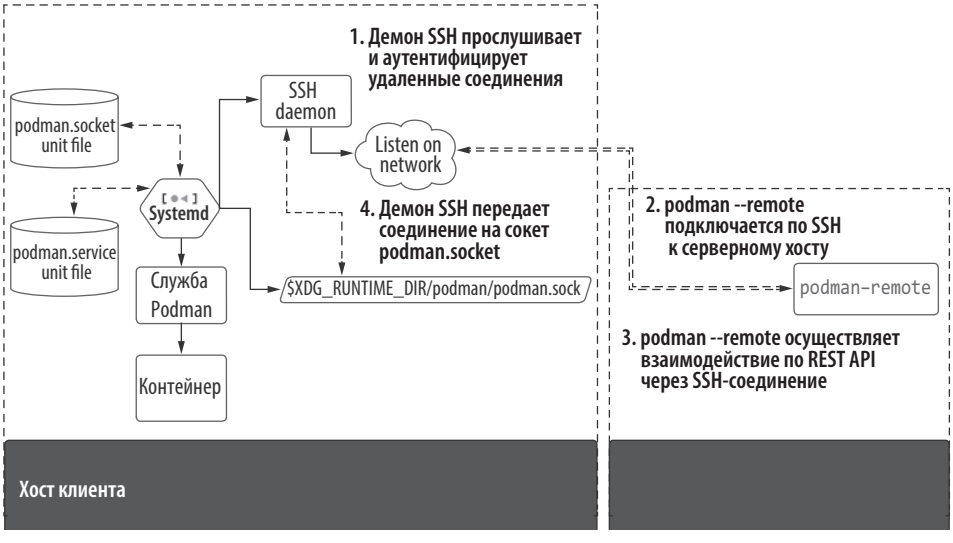


Рис. 9.3. Подключение podman --remote по SSH к серверу

Таблица 9.5. Параметры, не поддерживаемые командой podman --remote

Параметр	Описание
`--env-host`	Совместное использование переменных окружения на двух разных машинах не имеет смысла. В некоторых случаях это могут быть две разные операционные системы, например Windows и Mac, взаимодействующие с Linux-службой Podman
`--group-add=keepgroups`	Параметр `--group-add` работает в режиме `--remote`, а специальный флаг `keep-groups` — нет. Флаг `keep-groups` указывает Podman на передачу в контейнер групп, к которым имеет доступ текущий процесс. Поскольку это клиент-серверная процедура, такая передача невозможна
`--http-proxy`	Параметр `--http-proxy` указывает Podman использовать переменные окружения HTTP-прокси на клиентской машине и передавать их на сервер. Поскольку прокси обычно настраивается на сервере, параметр `--http-proxy` не разрешается использовать с параметром `--remote`
`--preserve-fds`	Параметр `--preserve-fds` обеспечивает «утечку» файловых дескрипторов из вызывающего процесса в контейнер. Поскольку это удаленное соединение, то это невозможно

Таблица 9.5 (*окончание*)

Параметр	Описание
`--volume`	Параметр поддерживается, но при этом исходный том будет получен с удаленной машины, а не с той, на которой выполняется команда `podman` (если только они не находятся на одной и той же машине). Если вы используете виртуальную машину, сначала необходимо смонтировать каталог на хост-машине в VM. Затем, увидев монтирование внутри этой VM, Podman смонтирует его в контейнер
`--latest, -l`	Поскольку с одним и тем же сервером потенциально могут одновременно общаться несколько разных пользователей, концепция `--latest` могла привести к состоянию «гонки», поэтому она не поддерживается

Команды Podman выполняются на сервере. При этом с точки зрения клиента это выглядит так, будто Podman работает локально. Вам необходимо завершить настройку службы Podman на удаленном сервере.

Включение SSHD-соединений

Чтобы клиент Podman мог взаимодействовать с сервером, необходимо включить и запустить демон SSH на машине Linux, если до сих пор он еще не был включен:

```
$ sudo systemctl enable --now -s sshd
```

После запуска демона SSHD нужно включить службу Podman на удаленной машине.

Активация службы Podman на серверной машине

Перед выполнением любых команд клиента Podman необходимо включить службу systemd podman.sock на Linux-сервере. В этих примерах вы запускаете Podman от имени обычного непривилегированного пользователя. Для корректной работы Podman без root на сервере активируйте этот сокет для постоянного подключения с помощью следующей команды:

```
$ systemctl --user enable --now podman.socket
```

В общем случае при вашем выходе из системы systemd останавливает все процессы в ней. Поэтому необходимо указать systemd, чтобы процессы удаленных пользователей оставались в режиме rootless:

```
$ sudo loginctl enable-linger $USER
```

Этим также передается распоряжение systemd начать прослушивание данного сокета во время загрузки. После запуска службы с помощью команды Podman можно убедиться в том, что сокет прослушивается:

```
$ podman --remote info
Host:
  arch: amd64
  buildahVersion: 1.16.0-dev
...
```

ПРИМЕЧАНИЕ Включить rootful-службу `podman` можно с помощью следующей команды:

```
$ sudo systemctl enable --now podman.socket
```

Предыдущая команда `enable-linger` предназначена только для режима rootless. Теперь, когда удаленная служба включена и работает вместе с демоном SSHD, мы вернемся на клиентскую машину.

9.5.3. Настройка SSH на клиентской машине

Удаленный Podman использует SSH для связи между клиентом и сервером, когда они находятся на разных машинах. По умолчанию SSH запрашивает ввод имени пользователя и пароля при каждой команде, если не настроены SSH-ключи. Чтобы настроить SSH-соединение, необходимо сгенерировать пару SSH-ключей на клиентской машине. Вы можете просто использовать существующие SSH-ключи, если они у вас есть. Еще лучше, если у вас уже есть общие ключи с сервером. В своей Linux-системе я сгенерирую SSH-ключи с помощью следующей команды:

```
$ ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/myuser/.ssh/id_ed25519):
```

Завершив генерацию ключей, следует установить доверительное соединение между клиентской и серверной машинами с помощью команды `ssh-copy-id` или другой, подобной ей. Открытый ключ по умолчанию будет находиться в домашнем каталоге $HOME/.ssh/id_ed25519.pub. Вам необходимо скопировать содержимое id_ed25519.pub и добавить его в ~/.ssh/ authorized_keys на Linux-сервере. Дополнительную информацию о настройке среды SSH вы найдете на сайте https://red.ht/3HuxPT6:

```
$ ssh-copy-id myuser@192.168.122.1
passwd:
```

Если вы не хотите использовать SSH-ключи, то при каждой команде Podman вам будет предложено ввести пароль для входа в систему. После того как вы поделились с сервером своими SSH-ключами, следующим шагом будет настройка соединения с Podman.

9.5.4. Настройка соединения

Команда `podman system connection` позволяет управлять SSH-соединениями, которые будут использоваться командой `podman --remote`. Добавляется соединение с помощью команды `podman system connection add`. Назовите соединение `server1`. Файл ключа выбирается по умолчанию, но можно применить параметр `--identity` для указания используемого SSH-ключа. Наконец, необходимо задать полный SSH-адрес для сокета Podman. Полный адрес включает в себя учетную запись пользователя `myuser` и IP-адрес, а также путь к сокету Podman для этой учетной записи:

```
$ podman system connection add server1 --identity ~/.ssh/id_ed25519
➥ ssh://myuser@192.168.122.1/run/user/1000/podman/podman.sock
```

Данная команда добавляет удаленное соединение к Podman. Поскольку это было первое добавленное соединение, Podman отмечает его как соединение по умолчанию.

Выведите список доступных соединений с помощью команды `podman system connection list`. Обратите внимание на символ * после имени соединения, указывающий, что это соединение по умолчанию:

```
$ podman system connection list
Name    Identity         URI
system1*    id_ed25519
➥ ssh://myuser@192.168.122.1/run/user/1000/podman/podman.sock
```

Проверьте соединение с помощью `podman info`:

```
$ podman --remote info
host:
  arch: amd64
  buildahVersion: 1.23.1
  cgroupControllers:
...
```

> **ПРИМЕЧАНИЕ** Параметр `--connection (-c)` применяется в том случае, если у вас есть несколько соединений и вы хотите выбрать не то, которое задано по умолчанию. Используйте команду `man podman-system-connection` для просмотра всех возможных параметров.

Для управления контейнерами, работающими на серверах или виртуальных машинах под управлением Linux, можно использовать параметр команды `podman` или клиент `podman-remote`. Связь между клиентом и сервером в значительной степени опирается на SSH-соединения, поэтому рекомендуется использовать SSH-ключи. После установки Podman на удаленном сервере необходимо настроить соединение с помощью команды `podman system connection add`, чтобы затем оно использовалось последующими командами Podman. В табл. 9.6 приведены доступные команды `podman system`.

Таблица 9.6. Команды podman system

Команда	man-страница	Описание
`connection`	`podman-system-connection(1)`	Управляет удаленными конечными точками SSH
`df`	`podman-system-df(1)`	Показывает использование диска Podman
`info`	`podman-system-info(1)`	Отображает информацию о системе
`migrate`	`podman-system-migrate(1)`	Перемещает контейнеры в новое пользовательское пространство имен
`prune`	`podman-system-prune(1)`	Удаляет неиспользуемые данные подов, контейнеров, томов и образов
`renumber`	`podman-system-renumber(1)`	Перенумеровывает блокировки (locks)
`reset`	`podman-system-reset(1)`	Сбрасывает хранилище в исходное состояние
`service`	`podman-system-service(1)`	Запускает службу API

РЕЗЮМЕ

- Podman может быть запущен как REST API-служба.
- Podman поддерживает два REST API-эндпоинта.
- Сокет Podman поддерживает два API.
- Режим совместимости или режим Docker позволяет клиентским инструментам Docker работать с Podman.
- Режим Podman позволяет удаленным клиентам использовать расширенные возможности Podman.

- Podman-py — это библиотека на языке Python, используемая для взаимодействия с сервисом Podman.
- Docker-py — это библиотека на языке Python, используемая для взаимодействия со службой совместимости Podman.
- Podman поддерживает запуск `docker-compose` со службой совместимости для оркестрации контейнеров `compose` на одном узле.
- Команда `podman --remote` взаимодействует со службой Podman по протоколу SSH при управлении контейнерами.
- Команда `podman system connect` управляет SSH-соединениями с удаленными службами Podman, облегчая управление контейнерами в вашем окружении.

Часть 4

Безопасность контейнеров

В части 4, заключительной, я рассказываю все, что знаю о безопасности контейнеров. Эта часть очень техническая, но некоторые приведенные в ней ключевые принципы помогут вам понять, в каких случаях контейнер может получить отказ в доступе. В ней также объясняются преимущества запуска приложений в контейнере с точки зрения безопасности. Контейнеризация приложений обеспечивает значительную защиту от потенциальных взломов системы вашего хоста.

В главе 10 я опишу все свойства ядра, которые Podman использует для изоляции контейнеров друг от друга, а также от хостовой системы. Я расскажу о SELinux, seccomp, Linux-привилегиях, точках монтирования read-only и о многих других возможностях.

В главе 11 мы углубимся в вопросы безопасности. Вы узнаете о лучших практиках обеспечения безопасности при запуске контейнеров в продакшен, о том, как следует проектировать приложение и как запускать контейнерное приложение в продакшен.

10 Изоляция контейнеров

В ЭТОЙ ГЛАВЕ

- ✓ Все защитные механизмы Linux, используемые для обеспечения изоляции контейнеров друг от друга
- ✓ Доступ только на чтение к файловым системам ядра, которые нужны процессам внутри контейнера, но должны быть ограничены для доступа на запись
- ✓ Маскировка файловых систем ядра для сокрытия информации от хостовой системы
- ✓ Linux-привилегии, ограничивающие полномочия root
- ✓ PID, IPC и сетевые пространства имен, которые скрывают большую часть операционной системы от процессов внутри контейнеров
- ✓ Пространство имен mount, которое вместе с SELinux дает доступ процессам контейнера только к определенному образу и томам
- ✓ Пользовательское пространство имен, которое позволяет создавать внутри контейнера root процессы, не являющиеся root вне контейнера

В главах 10 и 11 я рассматриваю и демонстрирую на примерах дополнительные аспекты безопасности Podman. Некоторые из этих вопросов были рассмотрены в других главах, но я считаю целесообразным остановиться на данных функциях отдельно.

При работе с контейнерами я часто сталкиваюсь с тем, что если процессу контейнера отказывают в каком-либо доступе, первой реакцией пользователя оказывается запуск контейнера в режиме `--privileged`, а это отключает все разграничения уровней безопасности для контейнера. Если вы разберетесь в работе с функциями безопасности, рассмотренными в этой главе, то сможете избежать подобных ситуаций.

Рассматривая контейнеры с точки зрения безопасности, я изучаю, как защитить ядро и файловую систему хоста от процессов внутри контейнера. Я написал книжку-раскраску «The Container Coloring Book» (https://red.ht/3gfVlHF) с иллюстрациями Майрин Даффи (Máirín Duffy, @marin), в которой особенности безопасности контейнеров описываются на примере трех поросят (рис. 10.1).

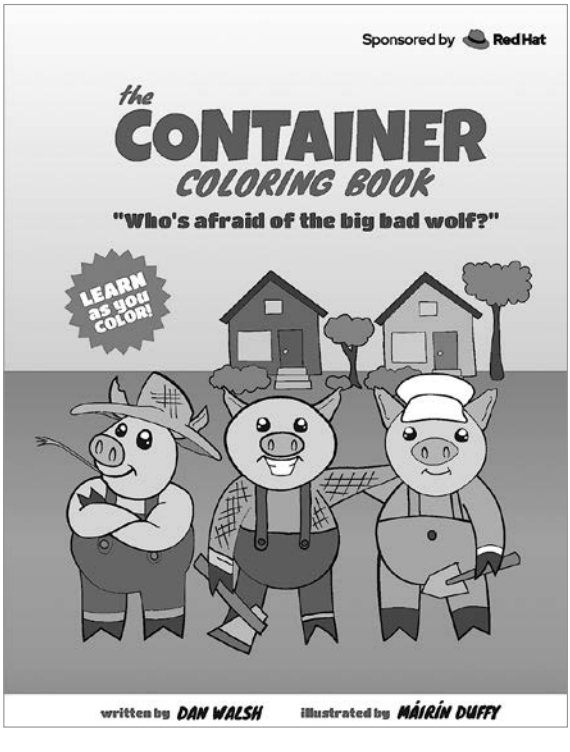


Рис. 10.1. Книга-раскраска «Контейнер» (https://red.ht/3gfVlHF)

В книге я провожу аналогию между тремя поросятами и приложениями. Я рассматриваю, где поросята живут, и их выбор жилища сравниваю с выбором архитектуры систем.

Дом на одну семью соответствует запуску одного приложения на одном изолированном узле. Проживание в двухквартирном доме эквивалентно запуску каждого приложения на отдельной виртуальной машине. Проживание в гостинице или в многоквартирном доме аналогично контейнерам, когда вы получаете собственную квартиру, но полагаетесь на безопасность службы регистрации, контролирующей доступ к вашему жилому пространству. Если ресепшн будет взломан, то и ваше жилище пострадает. Подобным же образом контейнеры полагаются на безопасность ядра. Если один контейнер сможет захватить ядро хоста, то он сможет захватить все контейнерные приложения, работающие в этой системе. Кроме того, если он получит доступ к основной файловой системе, то сможет читать и записывать данные всех контейнеров внутри системы.

С этой точки зрения задача номер один для хоста заключается в защите ядра и файловой системы от контейнерных процессов. Средства, используемые для этого, и описаны далее в этой главе.

Защита ядра от потенциально вредоносных контейнеров является основной целью обеспечения контейнерной безопасности. Если уязвимо ядро, то уязвима и остальная система, и все остальные контейнеры. Во многих случаях единственным местом контакта контейнера с хост-системой является само ядро хоста.

Процессы внутри контейнера могут взаимодействовать с ядром множеством различных способов. В данном разделе рассматриваются такие взаимодействия и функции операционной системы, используемые для защиты процессов контейнера.

Ядро Linux содержит файловые системы, которые позволяют процессам взаимодействовать и конфигурировать ядро. Защита этих файловых систем от контейнерных процессов является первой функцией безопасности, которую вы исследуете.

10.1. READ-ONLY ПСЕВДОФАЙЛОВЫЕ СИСТЕМЫ ЯДРА LINUX

Такие псевдофайловые системы ядра Linux обычно монтируются под /proc и /sys. В табл. 10.1 приведены некоторые псевдофайловые системы ядра Linux, смонтированные на моей машине.

Таблица 10.1. Файловые системы, монтируемые только для чтения

Точка монтирования	Описание
`/sys`	Файловая система sysfs позволяет манипулировать из пользовательского пространства объектами, которые создаются и уничтожаются в пространстве ядра
`/sys/kernel/security`	Псевдофайловая система безопасности используется для чтения и конфигурирования модулей безопасности. Примером служит модуль Integrity Measurement Architecture (IMA)
`/sys/fs/cgroup`	Файловая система cgroup используется для управления контрольными группами
`/sys/fs/pstore`	Файловая система pstore хранит энергонезависимую информацию, полезную для диагностики причин сбоя системы
`/sys/fs/bpf`	Файловая система Berkeley Packet Filter (BPF) — механизм, позволяющий использовать в ядре Linux пользовательские программы, раскрывающие информацию о ядре и контролирующие выполнение процессов в системе
`/sys/fs/selinux`	Файловая система SELinux используется для настройки SELinux в ядре (раздел 10.2.7)
`/sys/kernel/config`	Файловая система configfs предназначена для создания объектов ядра, управления ими и удаления объектов ядра из пользовательского пространства

Для успешной работы большинству процессов требуется доступ на чтение к этим псевдофайловым системам, но только процессам администратора требуется доступ на запись. Обычно для модификации этих файловых систем ядро полагается на отделение root от не-root или на наличие привилегии `CAP_SYS_ADMIN` (раздел 10.2.2).

Часто контейнеры должны запускаться от имени root. Это требует от системы безопасности контейнеров использовать другие средства, чтобы предотвратить запись в эти файловые системы ядра со стороны root-процесса. Podman не монтирует большинство из этих псевдофайловых систем. Однако он монтирует /sys, /sys/fs/cgroup и /sys/fs/selinux как доступные только для чтения. Когда вы находитесь в пространстве имен PID, файловая система /proc изменяется, то есть /proc внутри контейнера не соответствует /proc хоста. Процессы внутри контейнера могут влиять только на другие процессы внутри этого контейнера.

Файловая система /sys и файловая система с именем /proc иногда допускают утечку информации о хосте в контейнер. Чтобы предотвратить это, Podman монтирует /dev/null поверх файлов и монтирует файловые системы tmpfs, доступные только для чтения, поверх каталогов. Podman также монтирует некоторые

подкаталоги как доступные только для чтения, чтобы избежать записи в них процесса контейнера. Полный список файлов и каталогов, которые Podman монтирует в целях безопасности, приведен в табл. 10.2.

Таблица 10.2. Области файловой системы, маскируемые Podman

Типы маскировки	Пути
Read-only tmpfs, смонтированный поверх каталога	/proc/acpi, /proc/kcore, /proc/keys, /proc/latency_stats, /proc/timer_list, /proc/timer_stats, /proc/sched_debug, /proc/scsi, /sys/firmware, /sys/fs/selinux, /sys/dev/block
Read-only bind-монтирование поверх каталога	/proc/asound, /proc/bus, /proc/fs, /proc/irq, /proc/sys, /proc/sysrq-trigger

Я убедился, что почти все контейнерные образы прекрасно работают с такой дополнительной защитой. Но иногда контейнерному приложению может потребоваться дополнительный доступ к одному из этих замаскированных каталогов.

10.1.1. Снятие маскировки с путей

Вместо того чтобы заставлять контейнер работать в режиме `--privileged`, можно дать Podman указание снять маскировку с каталога. В следующем примере вы запустите контейнер и увидите, что в каталоге /proc/scsi нет файлов и каталогов, поскольку он смонтирован с помощью tmpfs:

```
$ podman run --rm ubi8 ls /proc/scsi
```

С помощью флага `--security-opt unmask=/proc/scsi` удалите точку монтирования и откройте исходные файлы и каталоги:

```
$ podman run --rm --security-opt unmask=/proc/scsi ubi8 ls /proc/scsi
device_info
scsi
sg
```

Используя символ * и указав определенный путь, вы можете даже размонтировать все каталоги в нем:

```
$ podman run --rm --security-opt unmask=/proc/* ubi8 ls /proc/scsi
device_info
scsi
sg
```

Снятие маскировки делает ваш контейнер менее защищенным, но это гораздо лучше, чем установка `--privileged` и отключение всех средств защиты.

В определенных ситуациях вам понадобится повысить безопасность системы, замаскировав часть псевдофайловых систем. Список маскируемых файловых систем приведен на страницах руководства `podman run`:

```
$ man podman run
...
   • unmask=ALL or /path/1:/path/2, or shell expanded paths (/proc/*):
Paths to unmask separated by a colon. If set to ALL, it will unmask all the
paths that are masked or made read only by default. The default masked
   paths are /proc/acpi, /proc/kcore, /proc/keys, /proc/latency_stats,
/proc/sched_debug, /proc/scsi, /proc/timer_list, /proc/timer_stats,
/sys/firmware, and /sys/fs/selinux.
   The default paths that are read only are /proc/asound, /proc/bus,
/proc/fs, /proc/irq, /proc/sys, /proc/sysrq-trigger, /sys/fs/cgroup.
```

10.1.2. Маскировка дополнительных путей

Если вы очень требовательны к безопасности или не доверяете какому-то контейнеру, следует добавить дополнительные маскируемые пути с помощью флага `--security-opt mask`. Например, если вы хотите запретить контейнерному процессу видеть устройства в каталоге /proc/sys/dev, выполните следующее:

```
$ podman run --rm ubi8 ls /proc/sys/dev
cdrom
hpet
i915
mac_hid
raid
scsi
tty
```

Этот путь можно замаскировать с помощью флага `--security-opt mask=/proc/sys/dev`:

```
$ podman run --rm --security-opt mask=/proc/sys/dev ubi8 ls /proc/sys/dev
```

Я показал, как Podman предотвращает чтение и, что более важно, запись root-процессов в псевдофайловые системы. На самом деле процессы контейнера могут видеть, что смонтировано внутри контейнера, просматривая /proc/self/mountinfo.

Листинг 10.1. Таблица монтирования в контейнере Podman

```
$ podman run –rm ubi8 cat /proc/self/mountinfo
...
1628 1610 0:5 /null /proc/kcore rw,nosuid –
➥ devtmpfs devtmpfs rw,seclabel,size=4096k,
➥ nr_inodes=1048576,mode=755,inode64
...
```

Показывает, что /dev/null смонтирован поверх /proc/kcore

```
1620 1595 0:86 / /sys/firmware ro,relatime - tmpfs tmpfs
rw,context="system_u:object_r:container_file_t:s0:c406,c915",size=0k,uid=32
➥ 67,gid=3267,inode64
...
```

Показывает tmpfs, смонтированный только для чтения в /sys/firmware

Вы можете спросить себя: «Если контейнер знает, что было смонтировано, то что мешает пользователю root внутри контейнера снять монтирование или перемонтировать файловые системы на чтение/запись, а затем атаковать ядро хоста?»

10.2. LINUX-ПРИВИЛЕГИИ (LINUX CAPABILITIES)

Большинство «линуксоидов» знают, что в Linux есть два типа пользователей: root (привилегированный) и все остальные (непривилегированные). Root всемогущ, а не-root имеет весьма ограниченные полномочия, особенно в том, что касается конфигурирования и модификации ядра. Иногда непривилегированному процессу требуются привилегии для выполнения определенных команд, например `ping` или `sudo`. Linux помечает эти файлы как `setuid`, и когда непривилегированный процесс выполняет их, новый процесс получает соответствующие привилегии.

Четкое противопоставление привилегированных и непривилегированных процессов в Linux стало стираться году в двухтысячном. Разработчики ядра разделили полномочия root на группу различных привилегий (privileged capabilities). В настоящее время в моей системе ядро Linux поддерживает 41 привилегию. Чтобы увидеть список привилегий в вашей системе, выполните команду `capsh`. Вы обнаружите, что текущий набор привилегий для ваших процессов пуст. Ограничивающий набор привилегий — это набор привилегий, который может получить процесс при выполнении `setuid`.

Листинг 10.2. capsh --print, в котором показаны доступные процессу пользователя привилегии

В списке действующих привилегий ничего нет

Ограничивающий набор показывает все (41) привилегии

```
$ capsh --print
Current: =
Bounding set =
cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,
➥ cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,
➥ cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,
➥ cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,
➥ cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,
```

```
➥ cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,
➥ cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,
➥ cap_block_suspend,cap_audit_read,cap_perfmon,cap_bpf,cap_checkpoint_restore
Ambient set =
...
uid=3267(dwalsh) euid=3267(dwalsh)
gid=3267(dwalsh)
```

Поскольку вы выполнили команду capsh от имени обычного пользователя, вы видите в списке свои UID и GID

Это означает, что ваш пользовательский процесс может выполнить команду `sudo` и получить полный набор привилегий как root. Информацию о том, что делает каждая из привилегий, вы найдете на man-странице, выполнив для обращения к ней команду `man capabilities`. С годами сообщество пришло к выводу, что почти всем контейнерам не требуется полный список привилегий, поскольку они редко влияют на работу ядра.

10.2.1. Отключенные Linux-привилегии

Поскольку процессы, ограниченные контейнером, не должны производить манипуляции с операционной системой и в особенности с ядром, Podman может выполнять root внутри своих контейнеров с гораздо меньшими привилегиями. Для получения списка привилегий, доступных по умолчанию в контейнере Podman, выполните ту же команду `capsh`.

Листинг 10.3. Список привилегий, доступных по умолчанию в контейнере Podman

Текущий набор показывает всего 11 привилегий, поскольку процесс контейнера запущен от имени root

Ограничивающий набор показывает те же самые 11 привилегий

```
$ podman run --rm ubi8 capsh --print
Current: =
cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
➥ cap_setuid,cap_setpcap,cap_net_bind_service,cap_sys_chroot,
➥ cap_setfcap+eip
Bounding set =
cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
➥ cap_setuid,cap_setpcap,cap_net_bind_service,cap_sys_chroot,cap_setfcap
...
uid=0(root)
gid=0(root)
groups=
```

Поскольку контейнеры по умолчанию запускаются от имени root, вы видите UID и GID как root

Как видите, при запуске контейнера Podman по умолчанию отказался от 30 привилегий — из 41 осталось 11. Несмотря на то что контейнер имеет права root, он гораздо менее привилегирован, чем root в системе.

ПРИМЕЧАНИЕ Docker также понижает количество привилегий, но оставляет 14. Podman обеспечивает более высокий уровень безопасности за счет отказа от следующих дополнительных привилегий: `CAP_ MKNOD`, `CAP_AUDIT_WRITE` и `CAP_NET_RAW`.

Список привилегий, по-прежнему разрешенных в контейнере, в основном касается управления несколькими процессами. Например, `CAP_SETUID` и `CAP_SETGID` позволяют процессам внутри контейнера менять UID. В качестве примера можно привести ситуацию, когда веб-приложение работает под `UID=60`, но при запуске процесса контейнера ему необходимо некоторое время работать под именем root, а затем сменить свой UID на `60`. Если бы Podman отказался от `CAP_SETUID`, то root-процессу внутри контейнера было бы запрещено менять UID на UID веб-сервиса.

Еще одна интересная привилегия в Podman — `CAP_NET_BIND_SERVICE`, позволяющая привязать процесс к сетевому порту, меньшему `1024`, например к порту `80`. Вспомните, как в главе 2 говорилось, что вы не можете привязать порт `80` на хосте к порту `80` внутри контейнера. Пользовательские процессы не имеют `CAP_NET_BIND_SERVICE`, поэтому они не могут привязать порт `80`. В табл. 10.3 перечислены привилегии, доступные по умолчанию для root, работающего в контейнере с Podman. Этот список может быть изменен в файле containers.conf с помощью поля `default_capabilities` в таблице containers.

Таблица 10.3. Список привилегий, разрешенных root-процессам в контейнере по умолчанию

Параметр	Описание
`CAP_CHOWN`	Внесение произвольных изменений в UID и GID файлов
`CAP_DAC_OVERRIDE`	Обход проверки прав на чтение, запись и выполнение файлов
`CAP_FOWNER`	Обход проверки прав на операции над UID файловой системы
`CAP_SETFSID`	Запрет очищения битов режимов `set-user-ID` и `set-group-ID` при модификации файла
`CAP_KILL`	Обход проверки прав доступа для отправки сигналов
`CAP_NET_BIND_SERVICE`	Привязка сокета к привилегированным портам интернет-домена (номера портов менее `1024`)
`CAP_SETFCAP`	Установка произвольных привилегий для файла
`CAP_SETGID`	Изменение группового идентификатора (GID) процесса или списка дополнительных GID

Параметр	Описание
SET_SETPCAP	Добавление и удаление любых привилегий из ограничивающего множества вызывающего потока
CAP_SETUID	Произведение произвольных манипуляций с идентификатором пользователя процесса (UID)
CAP_SYS_CHROOT	Разрешение chroot изменять пространства имен mount

Вернемся к вопросу, что мешает корневому root-процессу размонтировать или перемонтировать файловые системы, доступные только для чтения. Отвечаю: то, что Podman отключает привилегию CAP_ SYS_ADMIN.

10.2.2. Отключение привилегии CAP_SYS_ADMIN

Самой мощной привилегией Linux является CAP_SYS_ADMIN. Эту привилегию можно описать следующим образом: представьте, что вы — разработчик, добавляющий новую функцию в ядро, и эта функция требует привилегированного доступа. Вы просматриваете список привилегий и не находите ни одной привилегии, подходящей для такого доступа. Разработчики ядра могут пойти на то, чтобы создать новую привилегию. Но если это что-то, что нужно сделать системному администратору, для этого и существует CAP_SYS_ADMIN. Если вы посмотрите на информацию о привилегиях в man, то увидите, что возможностям CAP_SYS_ADMIN посвящено множество страниц.

Одна из функций, которой управляет CAP_SYS_ADMIN, — возможность монтировать и размонтировать файловые системы. Поскольку по умолчанию эта возможность отключена, root-процессы в контейнерах Podman не могут размонтировать или перемонтировать точки монтирования, доступные только для чтения.

Как вы узнали ранее, в Podman по умолчанию все еще допускается использование 11 привилегий. В большинстве случаев вашему контейнерному процессу не нужно даже столько привилегий, поэтому от некоторых вы можете отказаться.

10.2.3. Отказ от привилегий

Я рекомендую запускать свои приложения с минимально возможными привилегиями. Одним из способов повышения безопасности системы является отказ от дополнительных привилегий.

Представьте, что вашему контейнерному процессу не нужно привязывать порты < 1024. Запустив Podman с флагом --cap-drop=CAP_NET_BIND_SERVICE, вы откажетесь от этой привилегии в своем контейнере.

Листинг 10.4. Привилегии внутри контейнера при отказе от CAP_NET_BIND_SERVICE

Обратите внимание, что в списке текущих привилегий больше нет CAP_NET_BIND_SERVICE

```
$ podman run --cap-drop CAP_NET_BIND_SERVICE ubi8 capsh --print
Current: =
cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
➥ cap_setuid,cap_setpcap,cap_sys_chroot,cap_setfcap+eip
Bounding set =
cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
➥ cap_setuid,cap_setpcap,cap_sys_chroot,cap_setfcap
...
```

Обратите внимание, что в ограничивающем наборе привилегий больше нет CAP_NET_BIND_SERVICE

Можно отказаться и от всех привилегий сразу, используя флаг `--cap-drop=all`:

```
$ podman run --cap-drop all ubi8 capsh --print
Current: =
Bounding set =
```

Даже если ваш контейнер работает от имени root, у него нет привилегий для внесения изменений в ядро. Иногда контейнер не может работать с ограниченным списком привилегий, предоставляемых Podman; в этом случае следует добавить необходимые.

10.2.4. Добавление привилегий

В некоторых ситуациях контейнер может не работать из-за отсутствия определенных привилегий. В таких случаях можно, конечно, просто выполнить команду `--privileged` и отключить всю защиту, но лучшим решением будет просто добавить необходимые привилегии.

Представьте, что у вас есть контейнер, который создает необработанный (raw) IP-пакет в сети своего пространства имен, для чего требуется `CAP_NET_RAW`. Podman по умолчанию этого не позволяет. Вместо того чтобы запускать контейнер с правами `--privileged`, используйте флаг `--cap-add CAP_NET_RAW`:

```
$ podman run --cap-add CAP_NET_RAW ubi8 capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,
➥ cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_
➥ sys_chroot,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,
➥ cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_
➥ sys_chroot,cap_setfcap
...
```

Если другие привилегии вашему контейнеру не нужны, вы можете одновременно отказаться от всех привилегий и добавить только `CAP_NET_RAW`, используя флаги `--cap-drop` и `--cap-add`:

```
$ podman run --cap-drop=all --cap-add CAP_NET_RAW ubi8 capsh --print
Current: = cap_net_raw+eip
Bounding set =cap_net_raw
...
```

10.2.5. Отключение новых привилегий

В Podman есть опция `--security-opt no-new-privileges`, которая отключает возможность получения контейнерными процессами дополнительных привилегий. По сути, она фиксирует для процессов ту группу привилегий Linux, которую они имеют на момент запуска. Даже если процессы могут выполнить `setuid`, ядро не позволяет им получить дополнительные привилегии. Параметр `no-new-privileges` также влияет на SELinux, предотвращая смену меток SELinux. Если бы в базе данных правил SELinux была обнаружена ошибка, контейнерному процессу все равно не было бы позволено изменить свою метку.

10.2.6. Root без привилегий по-прежнему опасен

Отключение привилегий означает, что ваш контейнер работает гораздо безопаснее, но еще более безопасным является запуск всех ваших контейнеров без каких-либо привилегий Linux. Но есть еще одна проблема при запуске контейнера от имени root. Учтите, что даже если вы откажетесь от всех привилегий, процесс все равно выполняется от имени root, а root-процессу разрешено изменять все файлы в системе, которые принадлежат root. root-процесс может изменить системный файл и обманом заставить привилегированного администратора выполнить его. Кроме того, некоторые клиент-серверные приложения (например, Docker) доверяют клиентской стороне соединения только в том случае, если она работает от имени root. Podman решает обе эти проблемы с помощью пользовательского пространства имен.

10.3. ИЗОЛЯЦИЯ UID: ПОЛЬЗОВАТЕЛЬСКОЕ ПРОСТРАНСТВО ИМЕН

В разделе 6.1.1 я познакомил вас с понятием пользовательского пространства имен. Напомню, что для пользователя, не имеющего root, UID назначались через файлы /etc/subuid и /etc/subgid. Для моих учетных записей диапазон UID от `100000–165535` был выделен вместе с UID `3265` и использовался программой

Podman при запуске контейнеров. На рис. 10.2 показано сопоставление (mapping) пользовательского пространства имен.

Это пространство имен позволяет моей учетной записи иметь root-доступ в контейнере, не являясь root на хосте. Запуск контейнеров в пользовательском пространстве имен устраняет проблему запуска процессов от имени пользователя root, а также доступа, встроенного в некоторые демоны.

Использование rootless-пользователей затрудняет то, что по умолчанию все контейнеры работают с одним и тем же пользовательским пространством имен. С точки зрения пользовательского пространства имен теоретически один контейнер может атаковать другой контейнер, поскольку они работают с одинаковыми UID. Кроме того, если процессы контейнера «вырвались» наружу, они могут читать/записывать содержимое вашего домашнего каталога, поскольку root-процессы в контейнерах работают с вашим UID.

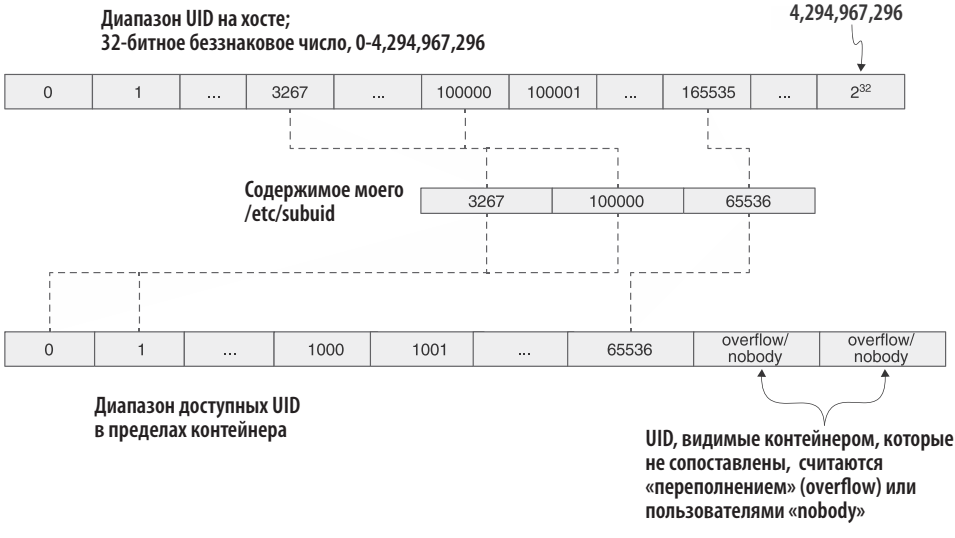


Рис. 10.2. Сопоставление UID, используемых в rootless-режиме Podman для моей учетной записи

10.3.1. Изоляция контейнеров с использованием флага --userns=auto

В Podman реализована функция выделения уникальных диапазонов UID для каждого запускаемого контейнера. Поскольку для каждой учетной записи пользователя выделяется ограниченное количество UID, эта функция лучше всего работает при запуске от имени пользователя root.

Чтобы запустить несколько контейнеров в их собственном пользовательском пространстве имен, необходимо сначала распределить UID и GID, которые будут взяты для этих контейнеров. В системе Linux доступны 4 миллиарда UID. Podman рекомендует выделять для контейнеров первые 2 миллиарда UID. Для этого в файлы /etc/subuid и /etc/subgid добавляется строка containers.

Листинг 10.5. Содержимое файлов /etc/subuid и /etc/subgid

```
# cat /etc/subuid
dwalsh:100000:65536
containers:2147483647:2147483648
# cat /etc/subgid
dwalsh:100000:65536
containers:2147483647:2147483648
```

Назначает первые 2 миллиарда UID пользователям контейнеров в Podman. Добавление этой строки дает указание другим инструментам в вашей системе, например useradd, не выделять UID и GID в этом диапазоне

С помощью параметра `--userns=auto` можно запустить контейнер в уникальном пользовательском пространстве имен. Podman выделяет для контейнера UID, начиная с `2147483647`, который вы указали в файле /etc/subuid. Затем Podman проверяет образ контейнера на наличие всех определенных в нем UID, а также файл /etc/passwd, если он существует в образе, и на его основе выделяет количество UID, необходимое для работы контейнера. Причем по умолчанию минимальное количество UID составляет `1024`:

```
# podman run --userns=auto ubi8 cat /proc/self/uid_map
0 2147483647 1024
```

Если я запускаю второй контейнер с конкретным пользователем `2000`, то распределение UID отражает это. Вы видите, что количество выделенных UID равно `2001`, это означает UID `2000` плюс один для пользователя root:

```
# podman run --user=2000 --userns=auto ubi8 cat /proc/self/uid_map
0 2147484671 2001
```

Обратите внимание, что начальный UID первого контейнера был равен `2147483647`, а начальный UID второго контейнера — `2147484671`. Вычитание первого UID `2147483647` из второго UID `2147484671` дает `1024`, то есть количество UID, выделенных для первого контейнера. Ни один UID в первом контейнере не пересекается со вторым, а значит, ни один процесс в первом контейнере не может атаковать процессы во втором контейнере, и наоборот.

Если Podman не выделяет необходимое вам количество UID или GID для контейнера, можно переопределить размер используемого в контейнере по умолчанию пользовательского пространства имен с помощью параметра `size`. В приведенном примере с помощью параметра `--userns=auto:size=5000` вы указываете Podman выделить `5000` UID для контейнера:

```
# podman run --userns=auto:size=5000 ubi8 cat /proc/self/uid_map
0 2147486672 5000
```

При удалении контейнеров Podman возвращает себе принадлежавшие им UID и использует их для следующего контейнера, созданного с флагом `--userns=auto`. Это заметно при запуске следующих друг за другом контейнеров с опцией `--rm`. Обратите внимание, что они начинаются с одного и того же UID. В следующем примере оба контейнера начинаются с UID `2147491672`:

```
# podman run --rm --userns=auto ubi8 cat /proc/self/uid_map
0 2147491672 1024
# podman run --rm --userns=auto ubi8 cat /proc/self/uid_map
0 2147491672 1024
```

Имя, используемое в файле /etc/subuid, а также минимальное и максимальное количество UID для пользовательских пространств имен задаются в файле storage.conf, описанном в табл. 10.4.

Таблица 10.4. Поля, используемые в файлах storage.conf для переопределения настроек пользовательского пространства имен

Параметр	Описание
`root-autouserns-user`	Определяет имя пользователя, применяемое для поиска одного или нескольких диапазонов UID/GID в файле /etc/subuid и /etc/subgid. Эти диапазоны разбиты на контейнеры, настроенные для автоматического создания пользовательского пространства имен. Контейнеры, настроенные на автоматическое создание пользовательского пространства имен, могут пересекаться с контейнерами с явным набором сопоставлений. Параметр `root-auto-userns-user` игнорируется пользователями без root. Он задается по умолчанию
`auto-usernsmin-size`	Определяет минимальный размер пользовательского пространства имен, создаваемого автоматически. По умолчанию он равен `1024`
`auto-usernsmax-size`	Определяет максимальный размер пользовательского пространства имен, создаваемого автоматически. По умолчанию он равен `65536`

10.3.2. Linux-привилегии пользовательского пространства имен

В разделе 10.2 вы узнали о Linux-привилегиях и о том, как они используются для разделения полномочий root. Когда контейнер запускается в пользовательском пространстве имен, он может иметь Linux-привилегии. Эти привилегии влияют только на UID и GID, сопоставленные с данным пользовательским пространством имен. Привилегии, не связанные с UID и GID, ограниченны. Обычно они влияют только на другие пространства имен, сопоставленные с данным пользовательским пространством имен.

Например, CAP_NET_ADMIN — это привилегия, позволяющая манипулировать сетевым стеком. Она дает процессу возможность настраивать правила межсетевого экрана и таблицы сетевой маршрутизации. Процессу с привилегией CAP_NET_ADMIN, относящейся к пространству имен, разрешено изменять только сеть, назначенную данному пользовательскому пространству имен, но не сетевое пространство имен хоста.

В следующем примере список привилегий в контейнере с пользовательским пространством имен такой же, как и при запуске контейнера без него. Во второй команде, с флагом --userns=auto, привилегии являются привилегиями пространства имен:

```
# podman run --rm ubi8 capsh --print | grep Current
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,
➥ cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_sys_chroot,
➥ cap_setfcap+eip
# podman run --rm --userns=auto ubi8 capsh --print | grep Current
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,
➥ cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_sys_chroot,
➥ cap_setfcap+eip
```

Чтобы убедиться, что это действительно так, попробуйте применить chown к файлу внутри контейнера с несуществующим UID. Это не удастся, поскольку привилегия CAP_CHOWN позволяет root-процессу внутри контейнера выполнять chown файлов на любой UID только при условии, что этот UID сопоставлен с данным пользовательским пространством имен:

```
# podman run --rm --userns=auto:size=5000 ubi8 chown 6000 /etc/motd
chown: changing ownership of '/etc/motd': Invalid argument
```

Но это сработает, если выполнить chown с UID, сопоставленным с данным пространством имен:

```
# podman run --rm --userns=auto:size=5000 ubi8 chown 4000 /etc/motd
```

Предположим, что вы запускаете все свои контейнеры с флагом --userns=auto. В этом случае контейнер запускается в своем уникальном пользовательском пространстве имен, изолированном от всех других контейнеров и UID на хостовой системе. Кроме того, вы получаете привилегии root с ограниченными полномочиями, и такие процессы не имеют привилегий на хосте.

10.3.3. Rootless Podman с флагом --userns=auto

Параметр --userns=auto работает с rootless-контейнерами, основываясь на количестве UID, доступных пользователю. Но их число ограниченно. В предыдущих

примерах пользовательские пространства имен начинаются с UID `1`. UID `1` относится к пространству имен непривилегированного пользователя:

```
$ podman run --userns=auto ubi8 cat /proc/self/uid_map
     0   1      1024
$ podman run --userns=auto ubi8 cat /proc/self/uid_map
     0   1025   1024
```

Исследовав ваше пользовательское пространство имен, вы обнаружите, что UID `1` в пользовательском пространстве равен `100000`:

```
$ podman run --rm ubi8 cat /proc/self/uid_map
    0    3267     1
    1    100000   65536
```

Это означает, что первый контейнер работает с UID `0`, сопоставленным с UID `1` в rootless пользовательском пространстве имен. UID `1` — это rootless UID `100000` на хост-системе. Проблема с непривилегированными пользователями `--userns=auto` заключается в том, что поскольку пользователь по умолчанию получает только 65 536 UID, максимум можно запустить 64 контейнера, а контейнеры, требующие более 65 536 UID, запускать нельзя.

> **ПРИМЕЧАНИЕ** Если запустить контейнер без использования флага `--userns=auto`, то UID, сопоставленные с пользовательским пространством имен, могут и, вероятно, будут пересекаться с UID в изолированных контейнерах с пользовательским пространством имен. Необходимо следить за тем, чтобы ни один из UID в таких контейнерах не использовал подобные UID, поскольку они уязвимы для атак. Чтобы избежать пересечений, я предлагаю использовать большой диапазон UID.

10.3.4. Пользовательские тома с флагом --userns=auto

При работе с пользовательским пространством имен трудно определить, UID какого пользователя должен принадлежать монтируемому в контейнер тóму, чтобы разрешить доступ. В следующем примере сначала создается каталог, затем том монтируется в контейнер и в нем вы пытаетесь создать файл.

Листинг 10.6. Недостатки использования томов в пользовательском пространстве имен

```
# mkdir /mnt/test
# ls -ld /mnt/test
drwxr-xr-x. 2 root root 6 Feb 8 16:23 /mnt/test
# podman run --rm -v /mnt/test:/mnt/test --userns=auto ubi8 ls -ld /mnt/test
```

На хосте этот каталог принадлежит root

```
drwxr-xr-x. 2 nobody nobody 6 Feb 8 21:23 /mnt/test
# podman run --rm -v /mnt/test:/mnt/test:Z --userns=auto ubi8 touch /mnt/test
touch: setting times of '/mnt/test':
➥ Permission denied
```

Даже root не имеет права записывать в каталог несопоставленного пользователя, если только этот каталог не является доступным для записи всем

Каталог указан как принадлежащий пользователю nobody, поскольку root UID=0 не сопоставлен с пользовательским пространством имен. Все файлы и каталоги, принадлежащие UID, не сопоставленным контейнеру, будут восприниматься как данные пользователя nobody. Параметр :Z указывает Podman на необходимость перемаркировки для SELinux

Podman поддерживает специальный параметр флага `--volume U`, который указывает на необходимость выполнить `chown` всех файлов или каталогов из исходного каталога в соответствующий UID основного процесса контейнера:

```
# ls -ld /mnt/test
drwxr-xr-x. 2 root root 6 Feb 8 16:38 /mnt/test
# podman run --rm -v /mnt/test:/mnt/test:Z,U
➥ --userns=auto ubi8 touch /mnt/test/test1
# ls -ld /mnt/test
drwxr-xr-x. 2 2147503960 2147503960
➥ 19 Feb 8 16:38 /mnt/test
```

После добавления параметра U процессы внутри контейнера могут производить запись на том

Podman присвоил исходному тому владельца 2147503960, чтобы соответствовать сопоставлению пользователя root в контейнере

Новая расширенная возможность ядра Linux называется `idmapped mounts`. Она позволяет изменять UID внутри исходного тома в соответствии с пользовательским пространством имен без фактической передачи прав на файлы с диска. В следующем примере мы заново создадим каталог /mnt/test и смонтируем его с опцией `idmap`. Когда том, смонтированный с применением `idmap`, появится в контейнере, файлы будут принадлежать root пользовательского пространства имен, и вы сможете читать и записывать их на основании стандартных разрешений. После завершения записи файлов они корректно сопоставляются с пользовательским пространством имен, в отличие от параметра U, который сопоставляет их с реальным UID процесса контейнера:

```
# chown -R root:root /mnt/test
# podman run --rm -v /mnt/test:/mnt/test:idmap,Z
➥ --userns=auto ubi8 ls -ld /mnt/test
drwxr-xr-x. 2 root root 31 Feb 9 11:56 /mnt/test
# podman run --rm -v /mnt/test:/mnt/test:idmap,Z
➥ --userns=auto ubi8 touch /mnt/test/test
# ls -l /mnt/test
total 0
-rw-r--r--. 1 root root 0 Feb 9 06:57 test
-rw-r--r--. 1 root root 0 Feb 8 17:02 test1
```

Меняем владельца исходного тома на root

Монтируем исходный том /mnt/test в контейнер с опцией idmap. Обратите внимание, что внутри контейнера этот путь принадлежит root

Создаем файл в исходном каталоге, чтобы проверить, что контейнер может записывать в этот каталог

На хостовой системе видно, что вновь созданный файл принадлежит настоящему root

ПРИМЕЧАНИЕ Функционал `idmap` являлся совершенно новым на момент написания книги и доступен не во всех файловых системах. В настоящее время он поддерживается только в привилегированном режиме, но я надеюсь, что это скоро изменится. `crun` — среда выполнения OCI, поддерживающая эту технологию уже сейчас.

Понимание того, какие преимущества в плане безопасности дает работа контейнеров с пользовательскими пространствами имен, очень важно. Далее я покажу, что можно сделать в плане безопасности в других пространствах имен.

10.4. ИЗОЛЯЦИЯ ПРОЦЕССОВ: ПРОСТРАНСТВО ИМЕН PID

Я часто упоминаю, что пространства имен не задумывались как механизм безопасности, хотя на самом деле они обеспечивают дополнительную безопасность за счет изоляции и сокрытия информации. Пространство имен PID скрывает факт наличия в системе других процессов. Знание о том, что в системе запущено определенное приложение, может оказаться ценным для взламывающего контейнер злоумышленника. Когда контейнер запускается в своем собственном пространстве имен PID, он может видеть только другие процессы, запущенные внутри этого контейнера. По умолчанию Podman запускает контейнеры в их собственных пространствах имен PID.

Некоторые приложения, поставляемые в виде образов контейнеров, требуют дополнительного доступа к системе. Если у вас есть такое приложение, которому необходимо следить за процессами на хосте, то вам придется отключить пространство имен PID, чтобы раскрыть все процессы в системе. Отключить пространство имен PID в Podman очень просто путем добавления параметра `--pid=host`. В следующих примерах показано, что при использовании пространства имен PID виден только процесс контейнера внутри самого контейнера. Вторая команда открывает контейнеру доступ ко всем процессам в системе.

Листинг 10.7. Разница между применением пространства имен pid и работой без него

```
$ podman run --rm ubi8 find /proc -maxdepth 1
➥ -type d -regex ".*/[0-9]*"
/proc/1
$ podman run --rm --pid=host ubi8 find
➥ /proc -maxdepth 1 -type d -regex ".*/[0-9]*"
/proc/1
/proc/2
/proc/3
/proc/4
...
```

Выполнив команду find, которая ищет все процессы внутри контейнера, вы увидите только один процесс

Выполнив команду find в контейнере с параметром --pid=host, вы увидите все процессы в системе

ПРИМЕЧАНИЕ В системах с SELinux раскрытие процессов хоста с помощью опции `--pid=host` имеет побочный эффект в виде отключения разделения SELinux. SELinux блокирует доступ к процессам хоста и затрудняет взаимодействие процессов внутри контейнера с этими процессами. Другие механизмы безопасности, такие как отмена привилегий и пользовательские пространства имен, действуют и могут блокировать доступ к процессам.

10.5. СЕТЕВАЯ ИЗОЛЯЦИЯ: СЕТЕВОЕ ПРОСТРАНСТВО ИМЕН

Сетевое пространство имен обеспечивает изоляцию от сети хоста. Оно позволяет Podman создавать виртуальные частные сети, чтобы контролировать, какие контейнеры могут общаться с другими контейнерами. В Podman есть возможность создавать несколько сетей и назначать контейнерам соответствующие сети. По умолчанию все контейнеры работают в сети хоста. Однако с помощью команды `podman network create` легко создавать дополнительные сети. В следующем примере вы создадите две сети — `net1` и `net2`:

```
$ podman network create net1
net1
$ podman network create net2
net2
```

При создании новых контейнеров можно назначить им определенную сеть с помощью параметра `--network`:

```
$ podman run -d --network net1 --name
➥ cnet1 ubi8 sleep 1000  ◄—— Запуск контейнера в сети net1
74ce5b2396f77fce8c499b121aeb8731f1e1b22e363a6a72d243487cf93a5897
$ podman run --network net1 alpine
➥ ping -c 1 cnet1  ◄—— Проверьте, что контейнер доступен из другого контейнера в пределах сети
PING cnet1 (10.89.0.4): 56 data bytes
64 bytes from 10.89.0.4: seq=0 ttl=42 time=0.077 ms
```

Если попытаться выполнить ping сети из сетевого пространства имен по умолчанию через имя контейнера или даже IP-адрес, то это не удастся:

```
$ podman run --rm alpine ping -c 1 cnet1
ping: bad address 'cnet1'
$ podman run alpine ping -c 1 10.89.0.4
PING 10.89.0.4 (10.89.0.4): 56 data bytes
64 bytes from 10.89.0.4: seq=0 ttl=42 time=0.073 ms
```

Убедитесь, что контейнер cnet1 по-прежнему доступен по IP-адресу

Аналогично, если попытаться выполнить ping из другой сети, `--network net2`, это также не удастся:

```
$ podman run --rm --network net2 alpine ping -c 1 cnet1
ping: bad address 'cnet1'
```

Создание частных сетей для контейнеров позволяет изолировать их друг от друга даже поверх сети, используя сетевое пространство имен.

> **ПРИМЕЧАНИЕ** Для данных примеров я использовал образ alpine, поскольку он поставляется с установленным пакетом ping, в то время как образ ubi8 его не содержит. Но исполняемый файл ping легко добавить в ubi8 с помощью Containerfile и `podman build`.

С помощью параметра `--net=host` контейнеру предоставляется доступ к сети хоста, что позволяет этому контейнеру подключаться к портам хоста. В некоторых ситуациях вы добьетесь большей производительности, исключив сетевое пространство имен.

10.6. IPC-ИЗОЛЯЦИЯ: ПРОСТРАНСТВО ИМЕН IPC

Пространство имен межпроцессного взаимодействия (inter-process communication, IPC) изолирует определенные ресурсы IPC, а именно объекты System V IPC и очереди сообщений POSIX. Оно также изолирует tmpfs /dev/shm от хоста и других контейнеров. Пространство имен IPC позволяет контейнерам создавать именованные IPC с тем же именем, что и у других контейнеров в той же системе, не вызывая конфликта.

Таким образом, IPC-изоляция не позволяет одному контейнеру атаковать другой через IPC или /dev/shm. Вы можете объединить два контейнера с пространством имен IPC с помощью команды `--ipc=container:NAME` или запустить их внутри одного пода. Они используют одно и то же пространство имен IPC. Эти контейнеры используют IPC вместе, но при этом остаются изолированными от хоста.

Листинг 10.8. Пространство имен IPC, обеспечивающее приватность /dev/shm для каждого контейнера

```
$ podman run -d --rm --name ipc1 ubi8 bash
➥ -c "touch /dev/shm/ipc1; sleep 1000"
93df44264dd4b87d24f59dfffb92a6a0b6359bc5bcf94213d5e38499a10d3f3e
$ podman run --rm ubi8 ls /dev/shm
$ podman run --rm --ipc=container:ipc1 ubi8 ls /dev/shm
ipc1
```

Создаем контейнер с именем ipc1, делаем touch /dev/shm/ipc1 и уходим в sleep

Запустив второй контейнер, можно убедиться, что /dev/shm/ipc не существует, поскольку контейнер работает в отдельном пространстве имен IPC

Запустив контейнер с общим пространством имен IPC, вы увидите, что /dev/shm является совместно используемым и файл IPC существует

Для совместного использования пространства имен IPC хоста со своим контейнером выполните `--ipc=host`.

ПРИМЕЧАНИЕ В системах с SELinux Podman модифицирует все контейнеры, разделяющие одно и то же пространство имен IPC, чтобы они использовали одну и ту же метку SELinux. В противном случае SELinux блокирует IPC-коммуникации между контейнерами при несовпадении этих меток. Использование параметра `--ipc=host` приводит к отключению разделения SELinux, в противном случае SELinux блокирует доступ к IPC хоста.

10.7. ИЗОЛЯЦИЯ ФАЙЛОВОЙ СИСТЕМЫ: ПРОСТРАНСТВО ИМЕН MOUNT

Пожалуй, самым важным средством изоляции является пространство имен mount. Пространство имен mount скрывает от контейнерных процессов всю файловую систему хоста. Контейнерные процессы видят только содержимое файловой системы, определенное в пространстве имен mount. Podman создает точку монтирования файловой системы rootfs и привязывает к ней все тома. Затем Podman запускает среду исполнения OCI, которая выполняет системный вызов pivot_root, а он, в свою очередь, изменяет корневое монтирование в пространстве имен mount вызывающего процесса. Он перемещает монтирование корня в каталог `rootfs`. В результате все содержимое операционной системы хоста исчезает, а процессы контейнера видят только выделенное содержимое. Отказ от привилегии `CAP_SYS_ADMIN` лишает процессы внутри контейнера возможности влиять на монтирование каталога `rootfs` для раскрытия основной файловой системы.

ПРИМЕЧАНИЕ Для получения дополнительной информации о системном вызове `pivot_root` почитайте man-страницу `pivot_root(2)`: `man pivot_root`.

Пространство имен mount и отсутствие `CAP_SYS_ADMIN` обеспечивают отличную изоляцию, но несмотря на это, в некоторых контейнерах случались попытки проникновения в базовую файловую систему, и здесь на помощь приходит SELinux. В качестве примера можно привести ошибку в среде выполнения OCI `runc` (CVE-2019-5736), которая позволяла контейнерным процессам перезаписывать исполняемый файл `runc` в rootful-контейнерах. Этот эксплойт позволял контейнерам выходить из-под контроля и управлять системами пользователей. Эксплойт воздействует на все контейнерные движки, включая Podman, Docker, CRI-O и containerd. Приятная новость заключается в том, что хорошо настроенный SELinux может его остановить. Podman в основном работает в режиме

rootless, a rootless Podman защищен двумя способами: SELinux и отсутствием возможности запуска от имени root. Я писал об этом эксплойте в статье «Latest container exploit (runc) can be blocked by SELinux», доступной на сайте Red Hat (https://www.redhat.com/en/blog/latest-container-exploit-runc-can-be-blocked-selinux).

10.8. ИЗОЛЯЦИЯ ФАЙЛОВОЙ СИСТЕМЫ: SELINUX

SELinux — это система маркировки, в которой каждый процесс и объект файловой системы получает метку. Затем в ядро записываются правила взаимодействия меток процессов с метками файловой системы, а также с метками других процессов. SELinux поддерживает множество различных механизмов безопасности; возможностями двух из этих механизмов пользуются контейнеры. Первый — контроль типа (type enforcement, TE), с помощью которого SELinux контролирует действия процессов в зависимости от их типа. Второй называется мультикатегорийной безопасностью (multi-category security, MCS), и в нем дополнительно используются категории, присваиваемые процессам.

SELinux поддерживается не во всех дистрибутивах. Fedora, RHEL и другие дистрибутивы Red Hat поддерживают SELinux, а дистрибутивы на базе Debian, например Ubuntu, часто не поддерживают. Если ваш дистрибутив Linux не поддерживает SELinux, то этот раздел вы можете пропустить.

10.8.1. SELinux type enforcement

Метки SELinux состоят из четырех компонентов: пользователь SELinux, роль, тип и уровень MCS (табл. 10.5).

Таблица 10.5. Примеры типов меток SELinux

Объект	Пользователь	Роль	Тип	уровень MCS
Container process	`system_u`	`system_r`	`container_t`	`s0:c1,c2`
Container process	`system_u`	`system_r`	`container_t`	`s0:c361,c871`
Container file	`system_u`	`object_r`	`container_file_t`	`s0:c1,c2`
Container file	`system_u`	`object_r`	`container_file_t`	`s0:c361,c871`
/etc/shadow label	`system_u`	`object_r`	`shadow_t`	`s0`
Container process	`system_u`	`system_r`	`spc_t`	`s0`
User process	`unconfined_u`	`unconfined_r`	`unconfined_t`	`s0-s0:c0.c1023`

В данном разделе вы сосредоточитесь на типе SELinux. Я создал книгу-раскраску «The SELinux Coloring Book», чтобы объяснить маркировку, используя сравнение с кошками и собаками (рис. 10.3).

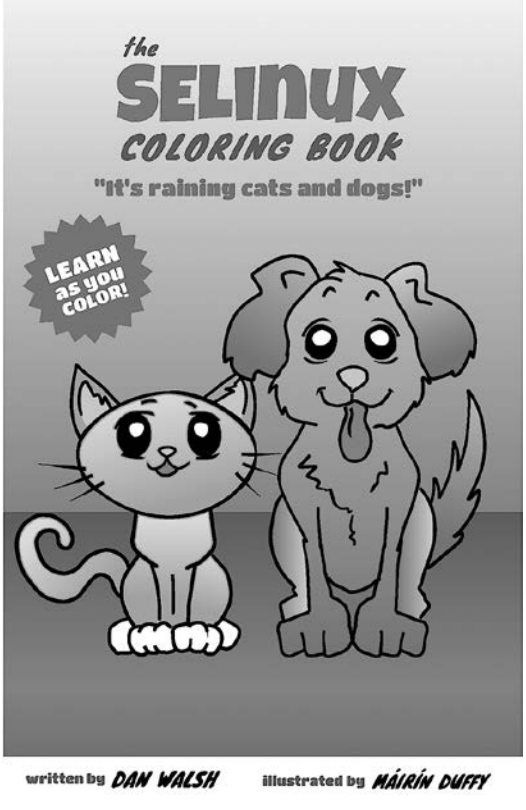


Рис. 10.3. Раскраска SELinux Coloring Book (https://people.redhat.com/duffy/selinux/selinux-coloring-book_A4-Stapled.pdf)

Объяснение материала в книжке-раскраске построено следующим образом. Представьте, что у вас есть группа процессов, обозначенных как тип «кошка», и другая группа процессов, обозначенных как тип «собака». Представьте, что в файловой системе также есть объекты, помеченные как «собачья еда» и «кошачья еда». И наконец, вообразите, что вы записали в ядро правила, согласно которым кошачьим типам разрешается есть кошачью еду, а собачьим типам — собачью еду. С помощью SELinux запрещается все, что не разрешено явно. Процессы «кошки» могут есть кошачью еду, а процессы «собаки» — собачью, но если тип «собака» попытается съесть кошачью еду, ядро Linux вмешается и заблокирует доступ.

Аналогичным образом работают и контейнеры. Podman маркирует каждый процесс контейнера типом `container_t`. Все файлы внутри контейнера помечаются как тип `container_file_t`. В ядро записываются правила, согласно которым процессам `container_t` разрешается читать, записывать и исполнять файлы, помеченные типом `container_file_t`.

> **ПРИМЕЧАНИЕ** SELinux не интересует, кто владелец файла и какие у него права доступа, поэтому можно, например, определить тип процесса, который имеет доступ ко всем типам файловой системы и не ограничивается SELinux, — так называемый неограниченный (unconfined) тип. В вашей системе Linux работает несколько неограниченных типов. Команда `id -Z` показывает, что пользовательские процессы работают с типом `unconfined_t`, а привилегированный контейнер работает с типом `spc_t`.

Когда Podman создает rootfs для контейнера, он помечает все файлы в rootfs как `container_file_t`. Это означает, что процесс контейнера может читать, записывать и выполнять все файлы в rootfs контейнера, но если они попадают в файловую систему хоста, то ядро SELinux блокирует доступ к ее объектам. В следующих примерах мы изучим, что происходит в контейнерах с SELinux. В первом примере вы видите метку контейнерного процесса. Обратите внимание, что его тип — `container_t`. Но при запуске с флагом `--privileged` Podman меняет метку на `spc_t`:

```
$ podman run --rm ubi8 cat /proc/self/attr/current
system_u:system_r:container_t:s0:c694,c944
$ podman run --rm --privileged ubi8 cat /proc/self/attr/current
unconfined_u:system_r:spc_t:s0
```

Исследуйте файлы внутри контейнера с помощью команды `ls -Z`. Все файлы помечены как `container_file_t`:

```
$ podman run --rm ubi8 ls -Z /
system_u:object_r:container_file_t:s0:c88,c191 bin
system_u:object_r:container_file_t:s0:c88,c191 boot
system_u:object_r:container_file_t:s0:c88,c191 dev
system_u:object_r:container_file_t:s0:c88,c191 etc
system_u:object_r:container_file_t:s0:c88,c191 home
system_u:object_r:container_file_t:s0:c88,c191 lib
...
```

Поскольку Podman правильно настроил среду SELinux, контейнерные процессы имеют полный доступ ко всем объектам в rootfs контейнера. SELinux практически не вмешивается в их работу, если только не происходит какой-либо сбой и контейнерный процесс не выходит из rootfs в операционную систему хоста. В этот момент SELinux начинает блокировать доступ. Представьте, что процесс контейнера, запущенный в вашей системе, «сбежал» из контейнера и попытался

прочитать SSH-ключи в вашем домашнем каталоге. Давайте проверим метки этих файлов. Оказывается, что эти файлы помечены типом `ssh_home_t`:

```
$ ls -1Z $HOME/.ssh/
unconfined_u:object_r:ssh_home_t:s0 authorized_keys
unconfined_u:object_r:ssh_home_t:s0 authorized_keys2
unconfined_u:object_r:ssh_home_t:s0 config
...
```

Поскольку в политиках SELinux нет правила, разрешающего процессу `container_t` читать файл `ssh_home_t`, ядро SELinux блокирует доступ. Это демонстрирует пример монтирования каталога .ssh в контейнере. Процесс контейнера получает `Permission denied` при попытке открыть каталог:

```
$ podman run -v $HOME/.ssh:/.ssh ubi8 ls /.ssh
ls: cannot open directory '/.ssh': Permission denied
```

Как вы узнали в разделе 3.1.2, Podman имеет параметры монтирования томов `z` и `Z`. Они указывают SELinux на необходимость переразметки содержимого исходного тома, чтобы сделать его пригодным для использования внутри контейнера. Для каталога .ssh это не очень удачная мысль.

Вместо этого давайте создадим временный файл и покажем, как действуют метки SELinux. Сначала создадим в домашнем каталоге временный файл с именем foo. Присвойте ему метку `user_home_t`. Смонтировав его в контейнер, мы увидим, что процессу контейнера отказано в доступе.

Листинг 10.9. Как SELinux работает с томами внутри контейнеров Podman

```
$ mkdir foo
$ ls -Zd foo
unconfined_u:object_r:user_home_t:s0 foo
$ podman run -v ./foo:/foo ubi8 touch /foo/bar
touch: cannot touch '/foo/bar': Permission denied
$ podman run --privileged -v ./foo:/foo ubi8 touch
➥ /foo/bar
$ ls -Z foo
unconfined_u:object_r:user_home_t:s0 bar
$ rm foo/bar
$ podman run -v ./foo:/foo:Z ubi8 touch /foo/bar
$ ls -Z ./foo
system_u:object_r:container_file_t:s0:c454,c510 bar
```

Файлы, созданные в вашем домашнем каталоге, по умолчанию имеют тип user_home_t

По умолчанию контейнерным процессам запрещено записывать содержимое в домашний каталог пользователя. Podman по умолчанию не изменяет метки томов

Файл, созданный привилегированным контейнером, имеет метку домашнего каталога пользователя (user_home_t)

Метки вновь созданного файла совпадают с меткой внутри контейнера

Параметр :Z при монтировании тома указывает Podman на необходимость перемаркировки содержимого каталога в соответствии с метками файлов в rootfs (container_file_t)

Флаг --privileged приводит к отключению разграничения SELinux и запуску контейнера с неограниченным типом (spc_t). Команда моделирует «побег» контейнера, показывая, что без SELinux сбежавший контейнер имеет право на запись в файловую систему

Контроль типов в SELinux показал свою неоценимую эффективность в блокировании побегов из контейнеров, когда все другие механизмы безопасности недоступны. В табл. 10.6 приведен список случаев побегов из контейнеров, предотвращенных с помощью SELinux.

Контроль типов SELinux отлично справляется с защитой операционной системы хоста от контейнерных процессов. Проблема заключается в том, что это не защищает от атак одного контейнера на другой.

Таблица 10.6. Основные эксплойты для контейнеров, предотвращаемые SELinux

Основные уязвимости	Описание
CVE-2019-5736	Запуск вредоносных контейнеров позволяет осуществлять побег из контейнера и получать доступ к файловой системе хоста
CVE-2015-3627	Небезопасное открытие файла-дескриптора 1, ведущее к повышению привилегий
CVE-2015-3630	Чтение/запись путей proc позволяет модифицировать систему и осуществлять доступ к информации о системе
CVE-2015-3631	При монтировании тома возможна эскалация профиля Linux Security Modules (LSM)
CVE-2016-9962	Уязвимость выполнения `runc`

10.8.2. Мультикатегорийное разделение SELinux

SELinux не блокирует процессы одного типа от атак на другие процессы того же типа. Можно вспомнить аналогию с кошками и собаками. Принудительное использование типов не позволяет `собаке` съесть `кошачий корм`, но не мешает `кошке-A` съесть корм `кошки-B`.

Помните, когда я представлял этот раздел, я сказал, что Podman пользуется двумя типами средств обеспечения безопасности SELinux. В SELinux есть механизм, обеспечивающий разделение процессов на основе значения поля level (уровень) в Multi-Category Security (MCS). SELinux определяет 1024 категории, которые могут быть объединены вместе для присвоения уровня каждому контейнеру. Podman выделяет две категории для каждого контейнера и затем убеждается, что уровень метки процесса соответствует уровню метки файловой системы. Затем ядро SELinux обеспечивает соответствие уровней MCS, иначе доступ будет запрещен.

ПРИМЕЧАНИЕ Разделение MCS — это фактически что-то из области доминирования. Каждая категория должна доминировать над уровнем MCS. Уровень `S0:C1,C2` может писать на уровни `S0:C1,C2`, `S0:C1`, `S0:C2` и `S0`. Но уровень `S0:C1,C2` не может писать на уровень `S0:C1,C3`, поскольку в исходной метке нет `C3`. На практике Podman использует только две категории или вообще не использует их. При использовании параметра тома `:z` Podman перемаркирует исходный каталог с уровнем `s0` — без категорий. Уровень `s0` позволяет процессам из любого контейнера читать и записывать объекты файловой системы с таким уровнем с точки зрения SELinux.

Вернемся к табл. 10.4, на этот раз сосредоточившись на поле уровня MCS (табл. 10.7).

Таблица 10.7. Маркировка контейнерных процессов с выделением уровня MCS

Объект	Пользователь	Роль	Тип	Уровень MCS
Container process	system_u	system_r	container_t	s0:c1,c2
Container process	system_u	system_r	container_t	s0:c361,c871
Container file	system_u	object_r	container_file_t	s0:c1,c2
Container file	system_u	object_r	container_file_t	s0:c361,c871
/etc/shadow label	system_u	object_r	shadow_t	s0
Container process	system_u	system_r	spc_t	s0
User process	unconfined_u	unconfined_r	unconfined_t	s0-s0:c0.c1023

Теперь посмотрим, как работают уровни MCS в Podman. Если запустить контейнеры один за другим и проследить за меткой SELinux, мы заметим, что уровень MCS каждого контейнера уникален:

```
$ podman run --rm ubi8 cat /proc/self/attr/current
System_u:system_r:container_t:s0:c648,c1009
$ podman run --rm ubi8 cat /proc/self/attr/current
system_u:system_r:container_t:s0:c393,c834
```

Уровень MCS не позволяет процессам атаковать друг друга. Напомню, что в разделе 10.8.2 вы создали файл foo/bar с приватной меткой контейнера. Если вы смонтируете этот файл в другой контейнер и попытаетесь произвести запись в него, то получите отказ в доступе.

Листинг 10.10. SELinux запрещает разным контейнерам совместно использовать один том

```
$ ls -Z ./foo
system_u:object_r:container_file_t:s0:c454,c510 bar
$ podman run -v ./foo:/foo ubi8 touch /foo/bar
touch: cannot touch '/foo/bar': Permission denied
$ podman run --security-opt label=level:s0:c454,c510
➥ -v ./foo:/foo ubi8 touch /foo/bar
```

Файл foo/bar имеет приватный уровень MCS, который Podman не передает другому контейнеру

Другим контейнерам доступ к файлу foo/bar запрещен на основании того, что они имеют отличающийся уровень MCS

Если принудительно установить уровень MCS контейнера, соответствующий метке предыдущего контейнера, SELinux разрешит доступ

Напомню, что параметр тома `Z` указывает Podman на то, что контейнер должен быть помечен как приватный, а параметр тома `z` — что контейнер должен быть помечен как общий для всех контейнеров. Этот параметр необходим, если у вас есть каталог, который вы разрешаете использовать нескольким контейнерам.

Листинг 10.11. Параметр тома z, указывающий Podman на перемаркировку томов общей меткой

```
$ podman run -v ./foo:/foo:z ubi8 touch /foo/bar
$ ls -Z foo/
system_u:object_r:container_file_t:s0 bar
$ podman run --rm -v ./foo:/foo ubi8 touch /foo/bar
```

Параметр -v ./foo:/foo:z указывает Podman, что том должен быть помечен как общий

Podman использует уровень MCS :s0, поскольку всем контейнерам разрешена запись в него

Другие контейнеры с различными уровнями MCS могут успешно модифицировать содержимое

ПРИМЕЧАНИЕ SELinux имеет 1024 категории, и Podman выбирает две категории для каждого контейнера. Уровень `s0:c1,c1` не допускается. Эти категории не должны совпадать, а порядок неважен. Уровень `s0:c1,c2` такой же, как `s0:c2,c1`. Существует 1024 × 1024 ÷ 2 - 1024 = ~500 000 уникальных комбинаций, то есть в системе можно создать полмиллиона уникальных контейнеров.

Иногда возникает необходимость отключить разделение SELinux для своего контейнера, например, чтобы предоставить доступ к своему домашнему каталогу внутри контейнера. Перемаркировать домашний каталог с помощью параметров `Z` или `z` — не самая лучшая мысль. Напомню, что при перемаркировке томов они должны быть приватными для контейнера. Перемаркировка домашнего каталога может привести к новым проблемам SELinux с другими ограниченными доменами. Можно запустить контейнер с флагом `--privileged`, но вы, вероятно,

захотите, чтобы другие механизмы безопасности все же были задействованы. Для этого есть флаг `--security-opt label:disable`:

```
$ podman run --rm --security-opt label=disable ubi8 cat
➥ /proc/self/attr/current
unconfined_u:system_r:spc_t:s0
$ podman run --rm -v $HOME/.ssh:/ssh --security-opt label=disable ubi8 ls /ssh
authorized_keys
authorized_keys2
config
fedora_rsa
fedora_rsa.pub
...
```

> **ПРИМЕЧАНИЕ** Целью проекта Udica (https://github.com/containers/udica) является создание политик SELinux для контейнеров. По сути, Udica изучает созданный контейнер с помощью `podman inspect`, а затем пишет политику, разрешающую доступ к томам, которые вам нужно подключить к контейнеру.

SELinux — очень мощный инструмент для защиты операционной системы хоста от контейнеров. С его помощью легко работать с контейнерами, если вы умеете обращаться с томами.

Мы разобрались с тем, как защитить файловую систему, настало время рассмотреть защиту ядра Linux от потенциально уязвимых системных вызовов.

10.9. ИЗОЛЯЦИЯ СИСТЕМНЫХ ВЫЗОВОВ SECCOMP

Системный вызов, часто называемый *syscall*, — это способ, с помощью которого программа запрашивает сервис у ядра операционной системы, в которой она выполняется. Наиболее распространенными системными вызовами являются `open`, `read`, `write`, `fork` и `exec`. В Linux более 700 системных вызовов.

В начале главы я говорил о том, что ядро Linux — это единственная точка отказа, которую могут атаковать вредоносные контейнеры, намеревающиеся выйти из-под контроля. Если в ядре Linux имеется баг, позволяющий атаковать ядро через системный вызов, то процессам удастся «сбежать» из контейнера. Благодаря функции ядра Linux seccomp процессы могут добровольно ограничить количество системных вызовов, выполняемых ими самими и их дочерними процессами. Podman по умолчанию исключает сотни системных вызовов, используя эту функцию. Предположим, что в ядре Linux есть уязвимость в одном из системных вызовов, которую контейнерный процесс может использовать для побега. Если Podman исключил его из таблицы системных вызовов, доступных данному контейнеру, то контейнер не сможет использовать эту уязвимость.

Фильтры seccomp в Podman хранятся в виде JSON-файла в /usr/share/containers/seccomp.json. Podman также изменяет список фильтров seccomp в зависимости от привилегий, которые вы даете контейнеру. При добавлении привилегии Podman добавляет системные вызовы, необходимые для этой привилегии. Привилегии и seccomp применяются отдельно, Podman просто пытается облегчить задачу пользователю. Если пользователь предоставляет свой собственный seccomp JSON-файл, чтобы модификации привилегий работали, он должен быть аналогичен файлу по умолчанию.

Фильтр seccomp можно корректировать, редактируя этот файл. В следующем примере вы удалите системный вызов `mkdir` из seccomp.json, а затем запустите контейнер, в котором пытаетесь создать каталог. Фильтр seccomp блокирует системный вызов, и контейнер завершит работу.

Листинг 10.12. Как фильтры seccomp могут блокировать системные вызовы в контейнере Podman

```
$ sed '/mkdir/d' /usr/share/containers
➥ /seccomp.json > /tmp/seccomp.json
$ diff /usr/share/containers/seccomp.json/
➥ tmp/seccomp.json
249,250d248
< "mkdir",
< "mkdirat",
$ podman run --rm --security-opt seccomp=/
➥ tmp/seccomp.json ubi8 mkdir /foo
mkdir: cannot create directory '/foo': Function not implemented
$ podman run --rm ubi8 mkdir /foo
```

С помощью команды sed удалите все записи с mkdir и создайте файл /tmp/seccomp.json

Используйте команду diff, чтобы показать удаленные записи с mkdir

Используйте флаг --security-opt seccomp=/tmp/seccomp.json для альтернативного фильтра seccomp. Команда mkdir не работает, поскольку системный вызов mkdir недоступен

Выполните ту же команду еще раз с фильтром по умолчанию, чтобы показать, что mkdir работает успешно

ПРИМЕЧАНИЕ Мало кто модифицирует фильтры seccomp, так как трудно определить количество системных вызовов, необходимых контейнеру. Существуют инструменты для генерации такого списка системных вызовов с помощью Berkeley Packet Filter (BPF). В пакете, расположенном на указанной веб-странице, находится хук, который следит за контейнером и автоматически генерирует файл seccomp.json, чтобы впоследствии использовать его для изоляции контейнера: https://github.com/containers/oci-seccomp-bpf-hook/.

Иногда стандартный файл контейнера seccomp.json оказывается слишком жестким. Ваш контейнер не заработает, если ему потребуется системный вызов, который недоступен. В этом случае отключите фильтрацию seccomp с помощью флага `--security-opt seccomp=unconfined`.

Как видите, фильтрация системных вызовов является мощным средством и действительно может ограничить доступ контейнерных процессов к ядру хоста.

Следующий уровень — использование изоляции KVM.

10.10. ИЗОЛЯЦИЯ ВИРТУАЛЬНЫХ МАШИН

В начале этой главы я сравнил процесс изоляции с тем, как три поросенка выбирали место для проживания. Они могли жить в отдельных домах, в дуплексе или в многоквартирном доме. Безопасность контейнеров по умолчанию похожа на жизнь в многоквартирном доме или отеле. Но чтобы добиться еще лучшей изоляции, можно использовать VM-изоляцию, которая, по сути, помещает контейнер в виртуальную машину.

В приложении B я рассказываю о том, как среды выполнения OCI, Kata и libkrun используют возможности виртуальных машин на базе ядра (KVM, Kernel-based Virtual Machine) для запуска своих контейнеров внутри легковесной виртуальной машины. Эти виртуальные машины запускают собственное ядро и средства инициализации для запуска контейнера. При этом почти все системные вызовы ядра хоста исключаются, что значительно усложняет выход из-под контроля.

Проблема с такой изоляцией заключается в том, что за нее приходится платить. Как и в случае с дуплексом или многоквартирным домом, вы получаете меньшее количество сервисов, совместно используемых контейнерами. Управление памятью, процессор и другие ресурсы труднее использовать совместно. Совместное использование томов в контейнере также будет работать хуже.

Итак, вы закончили изучение функций защиты Podman, используемых для изоляции контейнеров. Далее мы рассмотрим другие средства безопасности.

РЕЗЮМЕ

- Безопасность контейнеров связана с защитой ядра Linux и файловой системы хоста от вредоносных процессов контейнеров.
- Глубокая защита означает, что инструментарий контейнера использует преимущества как можно большего числа механизмов защиты. Если один механизм защиты не сработает, другие смогут защитить вашу систему.

11 Дополнительные аспекты безопасности

В ЭТОЙ ГЛАВЕ

- ✓ Обеспечение безопасности приложений, работающих на отдельных серверах, в виртуальных машинах и контейнерах
- ✓ Запуск контейнеров через службу в сравнении с их запуском как дочернего процесса контейнерного движка через fork и exec
- ✓ Средства защиты Linux, используемые для обеспечения изоляции контейнеров друг от друга
- ✓ Настройка доверия к образам контейнеров
- ✓ Подпись образов

В этой главе я демонстрирую дополнительные аспекты безопасности использования Podman для запуска контейнеров. Некоторые из этих вопросов рассматривались в других главах, но я считаю целесообразным остановиться на них именно с точки зрения безопасности.

Одна из проблем, с которыми чаще всего я сталкиваюсь при работе с контейнерами, заключается в том, что при отказе процессу контейнера в каком-либо доступе первой реакцией пользователя является запуск контейнера в режиме

`--privileged`, а это отключает все разграничения уровней безопасности для вашего контейнера. Избежать этой ситуации вам поможет понимание того, как работать с функциями безопасности, рассмотренными в этой главе.

11.1. СРАВНЕНИЕ ДЕМОНА С МОДЕЛЬЮ FORK/EXEC

Из предыдущих глав вы узнали достаточно много о проблемах при работе с демоном Docker в сравнении с моделью fork/exec, используемой в Podman.

11.1.1. Доступ к docker.sock

Напомню, что по умолчанию Docker работает под управлением демона, который принадлежит пользователю root. Это означает, что любой пользователь, имеющий доступ к демону, может запускать процессы с полным root-доступом в системе. Docker рекомендует некоторым пользователям включать свои учетные записи в группу docker в файле /etc/group. В некоторых дистрибутивах это позволяет получить доступ к файлу /run/docker.sock, не являясь пользователем root:

```
# ls -l /run/docker.sock
srw-rw----. 1 root docker 0 Jun 13 14:54 /run/docker.sock
```

Запускается Docker-контейнер подобно тому, как вы запускали контейнер Podman:

```
$ docker run registry.access.redhat.com/ubi8-micro echo hi
Unable to find image 'registry.access.redhat.com/ubi8-micro:latest' locally|
latest: Pulling from ubi8-micro
4f4fb700ef54: Pull complete
b6d5e0581b2f: Pull complete
Digest: sha256:a519ab06c0287085c352af0d2b84f2a2b257d2afb2e554b8d38a076cd6205b48
Status: Downloaded newer image for registry.access.redhat.com/
ubi8-micro:latest
hi
```

Это приводит в восторг многих пользователей, но потом они осознают, что с помощью простой команды Docker можно также запустить оболочку root shell на своей системе:

```
$ docker run -ti --name hack -v /:/host --privileged
➥ registry.access.redhat.com/ubi8-micro chroot /host
# cat /etc/shadow
...
```

На хостовой системе у вас появляется оболочка root с полными привилегиями, в которой вы можете как угодно взламывать машину. Мало того, в Docker по умолчанию все журналы ведутся в виде файлов. Закончив взлом системы, вы можете удалить файлы журнала и все записи о своей деятельности:

```
$ docker rm hack
hack
```

При использовании rootless Podman это невозможно, поскольку при запуске контейнера его процессы запускаются под UID пользователя и имеют доступ только к тем файлам, что и любой процесс в вашей учетной записи. Один из способов, с помощью которого администраторы могут определить, не взломали ли их систему, — это обращение к системе логирования и журналам аудита.

11.1.2. Логирование и аудит

Одной из ключевых особенностей системы Linux является отслеживание действий процессов во время их работы в системе. Когда вы входите в ОС Linux, ваш UID записывается ядром в данные процесса в каталоге /proc/self/loginuid. Просмотреть эти данные можно, выполнив следующую команду:

```
$ cat /proc/self/loginuid
3267
```

Это поле сохраняется у всех процессов, созданных первым процессом после входа в систему. Даже если вы используете программу `setuid`, например `su` или `sudo`, ваш `loginuid` остается неизменным:

```
$ sudo cat /proc/self/loginuid
3267
```

Даже при запуске контейнера `loginuid` остается неизменным. В следующем примере вы запускаете простой контейнер в режиме демона, который находится в спящем состоянии, затем используете `podman inspect` для получения PID спящих процессов и, наконец, изучаете `loginuid` процесса, работающего в контейнере:

```
$ podman run -d ubi8-micro sleep 20
1c55b9cfa0cd20c36da4b606415e190a6c20cc868d3486981c7713d41ee9ea6a
$ podman inspect -l --format '{{ .State.Pid }}'
119394
$ cat /proc/119394/loginuid
3267
```

Обратите внимание, что контейнерный процесс по-прежнему выполняется с вашим `loginuid`. Значит, по этому полю ядро может отслеживать, какой

пользователь запустил контейнерный процесс в системе при использовании контейнерным движком модели fork/exec. Если провести этот же тест с Docker, то результаты будут совсем другими:

```
$ docker run -d registry.access.redhat.com/ubi8-micro sleep 20
df2302cf8c6385df2b86ccd3429166e0d8dd0c9f0d0139e98e6354809a04080e
$ docker inspect df2302cf8c6 --format '{{ .State.Pid }}'
120022
$ cat /proc/120022/loginuid
4294967295
```

Вместо своего `loginuid` вы видите `4294967295`, что равно 2^{32} - 1. Именно так ядро Linux представляет `-1` — это `loginuid` по умолчанию для всех процессов, запускаемых системой, а не для пользователей, вошедших в систему. Это связано с тем, что Docker использует клиент-серверную модель и процесс контейнера является дочерним процессом Docker-демона, а не клиента Docker. Поскольку Docker-демон был запущен systemd при загрузке системы, все его дочерние процессы имеют `loginuid`, равный `-1`.

Подсистема аудита ядра записывает `loginuid` каждого процесса в системе, когда он завершает поддающееся аудиту событие. Так, когда пользователь входит в систему и выходит из нее, эти события регистрируются. Модификация файлов /etc/passwd и /etc/shadow также является регистрируемым событием.

Ниже приведена запись `USER_START` из журнала аудита, сделанная, когда я вошел сегодня в систему. Мой UID `3267` записан вместе с моим именем пользователя:

```
# ausearch -m USER_START
type=USER_START msg=audit(1651064687.963:315): pid=2579 uid=0 auid=3267
➥ ses=3 subj=system_u:system_r:xdm_t:s0-s0:c0.c1023 msg='op=PAM:session_open
➥ grantors=pam_selinux,pam_loginuid,pam_selinux,pam_keyinit,pam_namespace,
➥ pam_keyinit,pam_limits,pam_systemd,pam_unix,pam_gnome_keyring,pam_umask acct=
➥ "dwalsh" exe="/usr/libexec/gdm-session-worker" hostname=fedora addr=?
➥ terminal=/dev/tty2 res=success'UID="root" AUID="dwalsh"
```

Если контейнер был запущен с помощью команды Podman, то подсистема аудита записывает в журналы аудита ваш UID. Если контейнер был запущен с помощью Docker, то в качестве `loginuid` записывается `-1`. Представьте, что ваша система была взломана через контейнер. Вам необходимо проанализировать журнал audit.log, чтобы определить пользователя, который запустил взломавший вашу систему контейнер.

Рассмотрим пример. Сначала как root установите наблюдение за файлом /etc/passwd с помощью `auditctl`:

```
# auditctl -w /etc/passwd -p wa -k passwd
```

Теперь запустите контейнер `--privileged` с помощью Docker, который обратится к файлу /etc/passwd хоста:

```
# docker run --privileged -v /:/host registry.access.redhat.com/ubi8-
➥ micro:latest touch /host/etc/passwd
```

Таким образом имитируется ситуация, когда Docker-контейнер выходит из-под контроля и может модифицировать файл /etc/passwd хоста. Посмотрите на журнал audit.log, где должна быть запись о модификации /etc/passwd. Обратите внимание, что в журнале указано `auid=unset`. Так в журнале аудита представлен `loginuid` пользователя, изменившего файл /etc/passwd. Поскольку ни один пользователь не запускал демон Docker напрямую, в журнале аудита нет записи о пользователе, запустившем контейнер:

```
# ausearch -k passwd -i
...
type=SYSCALL msg=audit(05/03/2022 08:24:52.885:464) : arch=x86_64
➥ syscall=openat success=yes exit=3 a0=AT_FDCWD a1=0x7ffef7a9ef75
➥ a2=O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK a3=0x1b6 items=2 ppid=6723
➥ pid=6743 auid=unset uid=root gid=root euid=root suid=root fsuid=root
➥ egid=root sgid=root fsgid=root tty=(none) ses=unset comm=touch
➥ exe=/usr/bin/coreutils
```

Выполните ту же команду с помощью Podman:

```
# podman run --privileged -v /:/host registry.access.redhat.com/
➥ ubi8-micro:latest touch /host/etc/passwd
```

Изучив журнал audit.log для контейнера Podman, который модифицирует файл /etc/passwd, вы увидите, что `auid=dwalsh`. Поскольку Podman работает по модели fork/exec и был запущен пользователем, который вошел в систему и имел запись в `loginuid`, audit.log фиксирует, какой пользователь запустил контейнер, взломавший систему:

```
# ausearch -k passwd -i
...
type=SYSCALL msg=audit(05/03/2022 08:25:42.466:480) : arch=x86_64
➥ syscall=openat success=no exit=EACCES(Permission denied) a0=AT_FDCWD
➥ a1=0x7fff3d5aef59 a2=O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK a3=0x1b6
➥ items=2 ppid=6978 pid=6986 auid=dwalsh uid=root gid=root euid=root
➥ suid=root fsuid=root egid=root sgid=root fsgid=root tty=(none) ses=1
➥ comm=touch exe=/usr/bin/coreutils
➥ subj=system_u:system_r:container_t:s0:c484,c845 key=passwd
```

ПРИМЕЧАНИЕ В текущей версии Fedora подсистема аудита отключена. Включить ее можно, удалив файл `/etc/audit/rules.d/audit.rules` и заново сгенерировав правила аудита с помощью команды `augenrules --load`.

Именно поэтому еще в 2014 году я говорил, что доступ к docker.sock через процессы, не принадлежащие root, опаснее, чем предоставление доступа через root-процесс или sudo. Во втором случае записывается `loginuid`, а значит, можно отследить, что пользователь делает в системе. Когда вы предоставляете доступ к root, выполняющему docker.sock, у вас не остается никаких данных отслеживания.

В следующем разделе мы рассмотрим, как можно защитить ядро и файловую систему от процессов, запущенных в контейнере.

11.2. РАБОТА С СЕКРЕТАМИ В PODMAN

Часто при запуске контейнера требуется предоставить секретные данные сервису, работающему в контейнере. Примером служит инструмент работы с базой данных, для управления доступом к которой требуется указать имя администратора и пароль. Еще один пример — служба, требующая пароль для доступа к другой службе.

Разработчики таких приложений не стремятся жестко кодировать секретную информацию в образ. Пользователь контейнерного приложения должен предоставить этот секрет. Можно просто передать секрет приложению через переменные окружения, но это означает, что при коммите образа секрет будет зафиксирован в образе.

Podman предоставляет механизм секретов, `podman secret`, который позволяет добавлять файлы или переменные окружения в контейнер без сохранения этих секретов при фиксации контейнера в образ. Рассмотрим прежде всего создание секретов.

Листинг 11.1. Использование секретов в контейнере Podman

```
$ echo "This is my secret" > /tmp/secret
$ podman secret create my_secret /tmp/secret
b5f27b90e9b3486fb5a78d1eb
$ podman run --rm --secret my_secret ubi8 cat
/run/secrets/my_secret
This is my secret
```

Добавьте свои секретные данные в файл

Используйте команду podman secret create для создания секрета на основе этого файла

Используйте параметр --secret для передачи секрета в контейнер

Секрет также можно передать в контейнер в качестве переменной окружения, добавив флаг `--secret my_secret,type=env`:

```
$ podman run --secret my_secret,type=env --name secret_ctr ubi8 bash
➥ -c 'echo $my_secret'
This is my secret
```

Если зафиксировать этот контейнер в образ, то секрет не будет сохранен внутри этого образа.

Листинг 11.2. Секрет не сохраняется при фиксации контейнера в образ

```
$ podman commit secret_ctr secret_img
Getting image source signatures
Copying blob a9820c2af00a skipped: already exists
Copying blob 3d5ecee9360e skipped: already exists
Copying blob dc409efbefc4 done
Copying config 501812299f done
Writing manifest to image destination
Storing signatures
501812299f0c0cfbb032d144e6d2c2a41c5eadf229e7b76f6264ab74d9f6c069
$ podman image inspect secret_img --format
➥ '{{ .Config.Env }}'
[TERM=xterm container=oci PATH=/usr/local/sbin:/usr/local/
➥ bin:/usr/sbin:/usr/bin:/sbin:/bin]
```

Зафиксируйте контейнер secret_ctr в образ secret_img

Проверьте образ, чтобы увидеть зафиксированные переменные окружения, и обратите внимание, что переменная окружения my_secret не зафиксирована

В табл. 11.1 перечислены все команды `podman secret`.

Таблица 11.1. Команды podman secret

Команда	man-страница	Описание
`create`	`podman-secret-create(1)`	Создание нового секрета
`inspect`	`podman-secret-inspect(1)`	Отображение подробной информации об одном или нескольких секретах
`ls`	`podman-secret-ls(1)`	Вывод списка всех доступных секретов
`rm`	`podman-secret-rm(1)`	Удаление одного или нескольких секретов

11.3. ДОВЕРИЕ К ОБРАЗАМ PODMAN

Часто возникают ситуации, когда пользователям нужно указать, каким реестрам и образам контейнеров они доверяют. Команда `podman image trust` позволяет это сделать. Она также позволяет определить блокируемые реестры.

Расположение доверенного реестра определяется транспортом и хостом реестра образа. В примере с данным образом контейнера — docker://quay.io/podman/stable — транспортом является Docker, а хостом реестра — quay.io.

ПРИМЕЧАНИЕ Настройка доверия к образам Podman недоступна в удаленном режиме, например на компьютере с Mac или Windows. Описанные здесь команды необходимо выполнять на компьютере с ОС Linux. Если вы используете машину Podman, то для входа в VM применяйте команду `podman machine ssh`. Более подробная информация приведена в приложениях E и F.

Политика доверия определяется в файле /etc/containers/policy.json, в котором описывается доверенная область действия реестра (реестр и/или репозиторий). Политика доверия может использовать открытые ключи для подписанных образов. Команда `podman image trust` должна выполняться от имени root.

Область доверия выбирается от наиболее конкретной к наименее конкретной. Другими словами, политика может быть определена для всего реестра, для конкретного хранилища в этом реестре или даже для конкретного подписанного образа внутри реестра. В следующем примере вы запретите загрузку из docker.io, а затем дадите разрешение загружать только образы docker.io/library.

В списке ниже перечислены допустимые значения диапазона для использования в файле policy.json, начиная с наиболее конкретного и заканчивая наименее конкретным:

```
docker.io/library/busybox:notlatest
docker.io/library/busybox
docker.io/library
docker.io
```

Если ни для одного из этих диапазонов конфигурация не найдена, используется значение по умолчанию (заданное с помощью `default` вместо `REGISTRY [/REPOSITORY]`), как показано в листинге 11.3. В табл. 11.2 описаны допустимые значения настроек доверия для реестров.

Листинг 11.3. Указание Podman не загружать образы из определенного реестра

Использование podman image trust, чтобы задать более конкретный реестр/хранилище для docker.io/library

Использование podman image trust позволяет отклонять все образы из реестра контейнеров docker.io

Попытка извлечь образ alpine из реестра контейнеров приводит к тому, что Podman его отклоняет

```
$ sudo podman image trust set -t reject docker.io
$ podman pull alpine
Trying to pull docker.io/library/alpine:latest...
Error: Source image rejected: Running image docker://alpine:latest
➥ is rejected by policy.
$ sudo podman image trust set -t accept
➥ docker.io/library
$ podman pull alpine
Trying to pull docker.io/library/alpine:latest...
```

Podman может загрузить образ docker.io/library/alpine

```
Getting image source signatures
Copying blob 59bf1c3509f3 skipped: already exists
Copying config c059bfaa84 done
Writing manifest to image destination
Storing signatures
C059bfaa849c4d8e4aecaeb3a10c2d9b3d85f5165c66ad3a4d937758128c4d18
$ podman pull bitnami/nginx
Resolving "bitnami/nginx" using unqualified-search registries
➥ (/etc/containers/registries.conf.d/999-podman-machine.conf)
Trying to pull docker.io/bitnami/nginx:latest...
Error: Source image rejected: Running image docker://bitnami/nginx:latest
➥ is rejected by policy.
```

Образы, загружаемые из остальных разделов docker.io, отклоняются

Таблица 11.2. Тип доверия указывает контейнерным движкам типа Podman, каким реестрам доверять

Типы	Описание
`accept`	Разрешается загружать образы из указанных реестров
`reject`	Образы из указанных реестров загружать запрещено
`signBy`	Образы из указанных реестров должны быть подписаны определенным именем

В файле policy.json есть записи, добавленные командой `podman image trust`:

```
$ cat /etc/containers/policy.json
{
  "default": [
    {
      "type": "insecureAcceptAnything"
    }
  ],
  "transports": {
    "docker": {
      "docker.io": [
        {
          "type": "reject"
        }
      ],
      "docker.io/library": [
        {
          "type": "insecureAcceptAnything"
        }
      ]
...
```

Для отображения текущих настроек в удобном для просмотра виде используйте команду `podman image trust show`:

```
$ podman image trust show
all               default             accept
repository        docker.io           reject
repository        docker.io/library   accept
repository registry.access.redhat.com signed   security@redhat.com
https://access.redhat.com/webassets/docker/content/sigstore
repository registry.redhat.io         signed
➥ security@redhat.com https://registry.redhat.io/containers/sigstore
docker-daemon                         accept
```

Флаги `accept` и `reject` позволяют установить, каким реестрам доверять, а какие отвергать. Если вы хотите ограничить доступность образов в вашей рабочей системе, измените политику по умолчанию, чтобы она отвергала образы из любого реестра. Все разрешенные образы должны поступать из конкретного реестра:

```
$ sudo podman image trust set --type=reject default
$ podman image trust show
All               default             reject
Repository        docker.io           reject
Repository        docker.io/library   accept
Repository registry.access.redhat.com signed    security@redhat.com
https://access.redhat.com/webassets/docker/content/sigstore
repository registry.redhat.io         signed
➥ security@redhat.com https://registry.redhat.io/containers/sigstore
docker-daemon                         accept
```

При таких настройках Podman принимает образы из docker.io/library и подписанные образы из registry.redhat.io. Все образы из других реестров отклоняются. Podman также позволяет загружать образы непосредственно из `docker-daemon`.

Не забудьте восстановить стандартный policy.json:

```
$ sudo cp /tmp/policy.json /etc/containers/policy.json
```

Podman поддерживает использование подписанных образов из реестров контейнеров. Компания Red Hat подписывает и поставляет свои образы. Рассмотрим, как и вы можете подписывать образы.

11.3.1. Подписание образов в Podman

Одним из способов подписания образов является использование ключа GNU Privacy Guard (https://gnupg.org). Podman может подписывать образы перед отправкой их в удаленные реестры, это называется *простой подписью*. Podman и другие контейнерные движки настраиваются так, чтобы они требовали от образов определенной подписи. Все неподписанные образы отвергаются.

Для начала необходимо создать пару ключей GPG или выбрать уже готовую. Чтобы сгенерировать новые GPG-ключи, выполните команду `gpg --full-gen-key` и следуйте указаниям интерактивного диалога. За описанием создания ключей

обращайтесь на следующую веб-страницу: https://www.gnupg.org/gph/en/manual/c14.html#AEN25.

Ниже приведен пример создания простого ключа с параметрами по умолчанию. Убедитесь, что используете свой собственный адрес электронной почты:

```
$ gpg --batch --passphrase '' --quick-gen-key dwalsh@redhat.com default
➥ default
```

Большинство реестров контейнеров не понимают, что такое подпись образов, они просто предоставляют удаленное хранилище для образов контейнеров. Если вы хотите подписать образ, вам необходимо распространять подписи самостоятельно, обычно это делается с помощью веб-сервера. Podman и другие контейнерные движки настраиваются на получение подписей из этого веб-сервиса.

В следующих примерах для демонстрации подписания образа создается веб-сервис, работающий на локальной машине. Podman умеет отправлять и подписывать образ одной командой. Он считывает расположение подписей в конфигурационном файле реестров /etc/containers/registries.d/default.yaml.

Изучите файл default.yaml, чтобы найти флаг `sigstore-staging` и то, где Podman по умолчанию хранит подписи:

```
sigstore-staging: file:///var/lib/containers/sigstore
```

Флаг `sigstore-staging` дает указание Podman хранить подписи в каталоге /var/lib/containers/sigstore. Если вам будет нужно, чтобы другие пользователи использовали эти подписи для проверки ваших образов, поместите эти образы на веб-сервер. Итак, мы готовы протестировать простую подпись, сначала подписав образ ubi8, а затем настроив Podman на загрузку образа с использованием подписи для его проверки.

Подпись и отправка образа

Перед началом этого этапа необходимо создать резервную копию нескольких файлов настроек безопасности, чтобы впоследствии восстановить их:

```
$ sudo cp /etc/containers/registries.d/default.yaml
➥ /etc/containers/policy.json /tmp
```

Давайте загрузим образ из реестра и добавим подпись, а затем вернем его в реестр. Обязательно используйте собственную учетную запись реестра, образ и ранее созданный GPG-ключ:

```
$ sudo podman pull quay.io/rhatdan/myimage
Trying to pull quay.io/rhatdan/myimage:latest...
...
2c7e43d880382561ebae3fa06c7a1442d0da2912786d09ea9baaef87f73c29ae
$ podman login quay.io/rhatdan
```

```
Username: rhatdan
Password:
Login Succeeded!
$ sudo -E GNUPGHOME=$HOME/.gnupg \
podman push --tls-verify=false --sign-by dwalsh@redhat.com
➥ quay.io/rhatdan/myimage
...
Storing signatures
```

Найдите в каталоге sigstore-staging /var/lib/containers/sigstore репозиторий с именем rhatdan. Вы увидите, что доступна новая подпись, созданная командой `podman push`. Убедитесь, что вы используете собственное имя учетной записи реестра:

```
$ sudo ls /var/lib/containers/sigstore/rhatdan/
'myimage@sha256=0460a9d13a806e124639b23e9d6ffa1e5773f7bef91469bee6ac88
➥ a4be213427'
```

Теперь, когда образ подписан, необходимо настроить веб-сервер для предоставления подписи, а также Podman и другие контейнерные движки на использование этих подписей и подписанных образов.

Настройка Podman для загрузки подписанных образов

При настройке Podman на использование подписей для проверки образов необходимо сконфигурировать систему на получение этих подписей. Обычно подписи передаются через веб-сервис. При этом устанавливается флаг `sigstore` в файле /etc/containers/registries. d/default.yaml для определения сайта, на котором хранятся подписи. Podman загружает подписи с этого сайта.

В приведенном примере будет создан веб-сервис, работающий на localhost с портом `8000`. В файл default.yaml добавьте веб-сервер `sigstore: http://localhost:8000`. Это позволит Podman получать подписи с этого веб-сервера при загрузке образов. Podman ищет подпись по имени образа и его дайджесту:

```
$ echo " sigstore: http://localhost:8000" | sudo tee --append
➥ /etc/containers/registries.d/default.yaml
```

Для примера запустите новый сервер с использованием `python3` внутри локального хранилища подписей /var/lib/containers/sigstore:

```
$ cd /var/lib/containers/sigstore && python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

В другом окне удалите quay.io/rhatdan/myimage из локального хранилища, так как вы собираетесь выполнить загрузку с подписями:

```
$ podman rmi quay.io/rhatdan/myimage
Untagged: quay.io/rhatdan/myimage:latest
Deleted: 2c7e43d880382561ebae3fa06c7a1442d0da2912786d09ea9baaef87f73c29ae
```

Необходимо настроить доверие к образам для репозитория quay.io/rhatdan и назначить открытый ключ publickey.gpg, который будет использоваться при проверке образов, подписанных dwalsh@redhat.com:

```
$ sudo podman image trust set -f /tmp/publickey.gpg quay.io/rhatdan
```

Предыдущая команда Podman добавляет следующую строфу в файл /etc/containers/policy.json:

```
...
"transports": {
    "docker": {
        "quay.io/rhatdan": [
            {
                "type": "signedBy",
                "keyType": "GPGKeys",
                "keyPath": "/tmp/publickey.gpg"
            }
        ],
...
```

Вы еще не создали файл keyPath /tmp/publickey.gpg. Создайте его с помощью следующей команды GPG:

```
$ gpg --output /tmp/publickey.gpg --armor --export dwalsh@redhat.com
```

Теперь можно загрузить подписанный образ:

```
$ podman pull quay.io/rhatdan/myimage
Trying to pull quay.io/rhatdan/myimage:latest...
...
Writing manifest to image destination
Storing signatures
2c7e43d880382561ebae3fa06c7a1442d0da2912786d09ea9baaef87f73c29ae
```

Сработало! Тем не менее пока вы не можете с уверенностью сказать, использовались ли при этом подписи. Чтобы убедиться в этом, попытайтесь загрузить из репозитория другой образ, для которого у вас нет подписей, — ничего не получится:

```
$ podman pull quay.io/rhatdan/podman
Trying to pull quay.io/rhatdan/podman:latest...
Error: Source image rejected: A signature was required,
➥ but no signature exists
```

Не забудьте восстановить все настройки по умолчанию:

```
$ sudo cp /tmp/default.yaml /etc/containers/registries.d/default.yaml
$ sudo cp /tmp/policy.json /etc/containers/policy.json
```

Остановите веб-сервер localhost, запущенный в другом терминале. В табл. 11.3 представлена инфраструктура, которую необходимо создать для использования простой подписи в вашей среде.

Таблица 11.3. Инфраструктура, необходимая для простой подписи

Требования	Описание
Приватный ключ GPG	Вам необходима пара ключей GPG, закрытый ключ из которой используется в сервисе, подписывающем образы
Web-сервер подписей	Где-то должен работать веб-сервер, имеющий доступ к хранилищу подписей

После создания инфраструктуры для использования простой подписи необходимо выяснить требования к конфигурации каждого клиента, который будет применять и проверять подписи. В табл. 11.4 перечислены все подобные требования.

Таблица 11.4. Конфигурация клиента, необходимая для простой подписи

Требования	Описание
Открытый ключ GPG (/tmp/publickey.gpg)	Открытый ключ GPG, используемый для подписи, должен присутствовать на всех машинах, которые загружают подписанные образы
Настроен sigstore клиента	Веб-сервер подписей должен быть сконфигурирован как sigstore в файле /etc/containers/registries.d/*.yaml на всех системах, которым необходимо загружать подписанные образы
Настроено доверие к образу клиента	Доверие к образам должно быть настроено на каждой системе, использующей данные образы

11.4. СКАНИРОВАНИЕ ОБРАЗОВ

Podman не является сканером образов, он возлагает эту задачу на другие инструменты. Однако в Podman есть хорошая функция, которая облегчает сканирование. Podman может напрямую монтировать образ, который подлежит сканированию. Сканеры просматривают смонтированное содержимое образа без необходимости выполнения кода, содержащегося в нем. Напомним, что нельзя монтировать контейнеры или образы в режиме rootless без предварительного входа в пользовательское пространство имен. Выполните команду `podman image mount`, и вы увидите сообщение об ошибке:

```
$ podman image mount ubi8
Error: cannot run command "podman image mount" in rootless mode, must
➥ execute 'podman unshare' first
```

В следующем примере используется команда `podman unshare` для входа в пользовательское пространство имен, а затем монтируется образ ubi8. Далее измените каталог на директорию монтирования и выполните команду `find`, чтобы найти все бинарные файлы `setuid` в образе. Обратите внимание, что для сканирования образа используются средства операционной системы хоста:

```
$ podman unshare
# podman image mount
# mnt=$(podman image mount ubi8)
# echo $mnt
/home/dwalsh/.local/share/containers/storage/overlay/05ddfb76c5eb2146646c70
➥ e20db21a35dfec2215f130ce8bd04fce530142cfbd/merged
# cd $mnt
# /usr/bin/find . -user root -perm -4000
./usr/libexec/dbus-1/dbus-daemon-launch-helper
./usr/bin/chage
./usr/bin/mount
./usr/bin/umount
./usr/bin/newgrp
./usr/bin/gpasswd
./usr/bin/passwd
./usr/bin/su
./usr/sbin/userhelper
./usr/sbin/unix_chkpwd
./usr/sbin/pam_timestamp_check
```

Сканирование образов с помощью инструментов внутри самого образа небезопасно, поскольку злоумышленник, взломавший образ, может модифицировать и инструменты сканирования. Podman упрощает работу средств сканирования.

11.4.1. Контейнеры, доступные только для чтения

Я часто говорю о контейнерах в продакшене и контейнерах в средах разработки. Когда контейнерное приложение находится в стадии разработки, полезно иметь возможность записывать образ контейнера и впоследствии фиксировать его. Хотя такая практика довольно распространена, большинство людей переключаются на использование Containerfiles, когда речь заходит о реальной сборке образов. Разработчики, передавая свои программы на тестирование, ожидают, что содержимое будет использоваться только для чтения.

При эксплуатации контейнера в продакшене, на мой взгляд, имеет смысл запускать образ в режиме «только чтение». Представьте, что вы запускаете приложение, которое подверглось взлому. Первое, что захочет сделать злоумышленник, — записать в приложение бэкдор. Тогда при следующем запуске контейнера или приложения в контейнере уже будет установлен эксплойт. Если образ был доступен только для чтения, хакер не сможет оставить в нем бэкдор.

Параметр `--read-only` не позволяет приложениям записывать содержимое в образ и заставляет их записывать содержимое только в файловую систему tmpfs или тома, добавленные в контейнер. Иногда требуется запретить контейнеру писать куда-либо в вашей системе, а читать или выполнять код разрешить только внутри контейнера. Еще одним преимуществом запуска контейнеров в режиме read-only является отслеживание ошибок, связанных с тем, что контейнер вел запись в образ,

о которой вы не знали. Наконец, запись поверх copy-on-write файловых систем, таких как overlayfs, почти всегда медленнее, чем запись в том или в tmpfs:

```
$ podman run --read-only ubi8 touch /foo
touch: cannot touch '/foo': Read-only file system
```

Одна из проблем работы в режиме rootless заключается в том, что приложения часто рассчитывают на запись в /run, /tmp и /var/tmp. Podman решает эту проблему путем автоматического монтирования файловых систем tmpfs в этих каталогах:

```
$ podman run --read-only ubi8 touch /run/foo
```

Некоторые пользователи считают, что разрешение контейнерным приложениям вести запись в любые места, даже в tmpfs, слишком небезопасно, поэтому в Podman есть параметр `--read-only-tmpfs`. Параметр `--read-only-tmpfs` добавляет tmpfs /run, /tmp и /var/tmp при запуске в режиме `--read-only`. Если вы хотите отключить эту опцию, используйте флаг `--readonly-tmpfs=false`:

```
$ podman run --read-only-tmpfs=false --read-only ubi8 touch /run/foo
touch: cannot touch '/run/foo': Read-only file system
```

11.5. ГЛУБОКАЯ ЗАЩИТА

В области безопасности распространена идея *глубокой защиты*. В соответствии с ней, для защиты объектов необходимо использовать несколько уровней или инструментов. Классическая аналогия — защита древнего замка, который обычно строится на возвышенности, имеет множество стен, рвов и других защитных сооружений. Чтобы добраться до хозяина, злоумышленнику необходимо преодолеть все эти слои защиты.

Защита контейнеров работает примерно так же. Podman использует все предоставляемые Linux механизмы безопасности, обеспечивая глубокую защиту.

11.5.1. Podman использует все механизмы безопасности одновременно

Контейнеры Podman могут работать со всеми механизмами безопасности, о которых говорилось в этой главе. Это означает, что для получения доступа к системе взломанный контейнер должен найти способ обойти read-only файловые системы, пространства имен, отключенные привилегии, SELinux, seccomp и т. д.

Бывают случаи, когда для запуска контейнера требуется ослабить некоторые механизмы защиты. Разумнее грамотно работать с рассмотренными в этой главе средствами безопасности, чем просто запускать контейнеры с флагом `--privileged`, который отключает все средства защиты.

Podman стремится к разумному уровню защиты контейнеров, но при этом он должен обеспечивать успешную работу универсальных контейнеров. Знание требований к безопасности контейнерного приложения и функций безопасности Podman позволяет усилить защиту контейнеров. Если вы уверены, что ваш контейнер не должен запускаться с правами root, то и не запускайте его с правами root. Если контейнеру не нужны какие-то привилегии Linux, откажитесь от них. rootless-контейнеры лучше, чем rootful-контейнеры. Рассмотрите также возможность запуска контейнеров в режиме read-only или в разделенных пользовательских пространствах имен. Используя эти меры, вы можете сделать стены замка вокруг своих контейнерных приложений более надежными.

11.5.2. Где следует запускать контейнеры?

Напоследок я хочу поделиться с вами одним соображением. В начале этой главы я рассказал о трех поросятах, живущих в различных типах жилищ — отдельно стоящих домах, дуплексе и многоквартирном доме, — каждый из которых был чуть менее защищен, чем предыдущий. Контейнерная безопасность может быть лучше, чем у поросят, живущих в отдельных жилых домах, поскольку «дома» можно соединять вместе.

Представьте, что у вас есть два разных контейнера: веб-фронтенд и база данных с данными кредитных карт. Чтобы быть уверенным, что они разделены, вы можете поместить их в одну систему внутри контейнеров, или, что еще лучше, в контейнеры, но при этом в отдельные VM, и даже разместить эти VM на отдельные машины. Вы сможете разместить свой веб-фронтенд на машине, работающей под управлением VM внутри контейнера в DMZ, выходящей в интернет. Все это можно сделать, разместив базу данных в частной сети, не ограничивая сетевой доступ к веб-фронтенду. Возможности практически безграничны.

РЕЗЮМЕ

- У безопасности контейнеров множество различных аспектов, включая разделение запущенных контейнеров, доверие к образам и реестрам, сканирование образов и т. д.
- Глубокая защита подразумевает использование как можно большего числа механизмов защиты. Если один из механизмов защиты не сработает, остальные смогут защитить вашу систему.
- Безопасность контейнеров обеспечивается защитой ядра Linux и файловой системы хоста от вредоносных контейнерных процессов.
- Настройка и контроль образов контейнеров, запускаемых в ваших системах, очень важны. Не позволяйте пользователям запускать случайные приложения из интернета.

Приложения

A Инструменты, связанные с Podman

В этом приложении описаны три инструмента, использующие библиотеки containers/storage и containers/image. Эти инструменты реализуют следующие функциональные возможности:

- перемещение образов контейнеров между различными реестрами и хранилищами контейнеров;
- сборка образов контейнеров;
- разработка, тестирование и запуск контейнеров в продакшен на одном узле;
- запуск контейнеров в продакшен с масштабированием.

Будучи создателем Podman, я осознал, что необходимы скорее специализированные инструменты, каждый из которых выполняет определенную функцию, чем универсальное монолитное решение.

С точки зрения безопасности каждая из этих функциональных возможностей требует различных ограничений. Контейнеры, используемые в продакшене, должны работать в более защищенной среде, чем контейнеры, применяемые в разработке и тестировании. Перемещение образов контейнеров между реестрами не требует привилегированного доступа к хосту, на котором выполняется команда, нужен только удаленный доступ к реестрам. Наименее защищенной будет система с монолитным демоном. Если моим контейнерам требуется больше прав доступа во время сборки, то в продакшене они получают такой же доступ, как и во время сборки.

Монолитный демон порождает еще одну критическую проблему: он препятствует тому, чтобы экспериментировать с инструментами, и не позволяет им развиваться своим собственным путем. Одним из примеров служит ситуация,

когда мы предложили внести изменения в Docker-демон, чтобы пользователи могли загружать различные типы содержимого OCI из реестров контейнеров. Это изменение было отклонено как не имеющее отношения к контейнерам Docker.

Модификация монолитного демона для одного продукта может негативно сказаться на возможностях другого продукта, использующего этот демон, привести к снижению производительности или полной неработоспособности. Так произошло при разработке Kubernetes, в качестве контейнерного движка опиравшегося на Docker-демон. Так как Docker-демон является монолитным и разрабатывается для различных проектов, многие его изменения сказывались на Kubernetes, что приводило к нестабильности. Стало очевидно, что Kubernetes нужен специализированный контейнерный движок для своих рабочих нагрузок. В декабре 2020 года было объявлено, что Kubernetes в перспективе будет использовать новый разработанный стандарт Container Runtime Interface (CRI, https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/) для улучшения взаимодействия между оркестраторами и различными средами выполнения контейнеров. Я написал книжку-раскраску «The Container Commandos» (рис. A.1; https://red.ht/3gfVlHF), проиллюстрированную Майрин Даффи (Máirín Duffy), в которой контейнерные инструменты, рассматриваемые в этом приложении, сравниваются с супергероями.

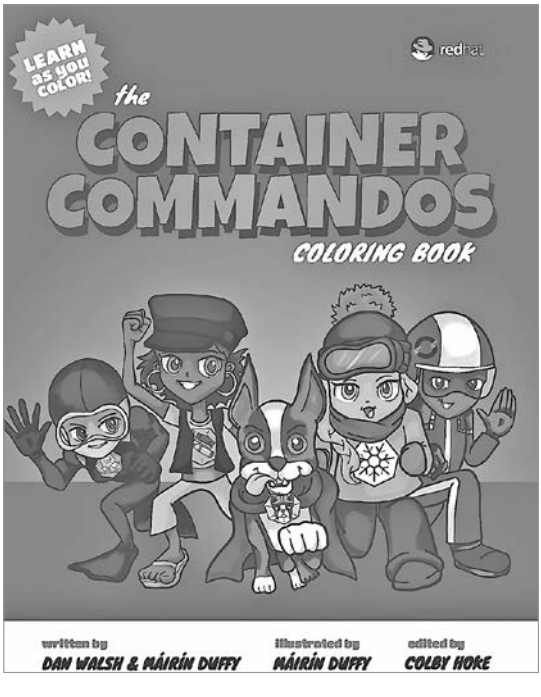


Рис. A.1. Книжка-раскраска «Контейнерные коммандос» (https://red.ht/3gfVlHF)

Наконец, иногда возникают конфликты интересов или графиков релизов. Наличие отдельных, самостоятельных инструментов позволяет развертывать релизы независимо от всех остальных, гарантируя новые возможности их клиентам. Для выполнения различных функций, описанных в табл. А.1, было создано четыре проекта.

Таблица А.1. Основные инструменты для работы с контейнерами, основанные на containers/storage и containers/image

Инструмент	Описание
`Skopeo`	Выполняет различные операции с образами контейнеров и репозиториями образов (https://github.com/containers/skopeo)
`Buildah`	Облегчает выполнение широкого спектра операций с образами контейнеров (https://github.com/containers/buildah)
`Podman`	Универсальный инструмент управления подами, контейнерами и образами (https://github.com/containers/podman)
`CRI-O`	Основанная на OCI реализация интерфейса Kubernetes Container Runtime Interface (https://github.com/cri-o/cri-o)

Узнав многое о Podman, вы теперь понимаете, почему он включен в этот список. Podman — отличный инструмент для изучения и разработки контейнеров, подов и образов. Он инкапсулирует в себе все то, что делает Docker CLI, но не замыкает все это в рамках использования одного централизованного демона. Поскольку Podman работает без демона и использует операционную систему для обмена данными, другие инструменты могут работать с теми же хранилищами данных и библиотеками.

Далее в этом приложении описаны такие инструменты, начиная со Skopeo (рис. А.2).

А.1. SKOPEO

При использовании таких контейнерных движков, как Docker или Podman, для просмотра образа контейнера в реестре необходимо загрузить этот образ из реестра в локальное хранилище. Только после этого его можно исследовать. Этот образ может быть огромным, и после его изучения вполне может оказаться, что это не то, что вы ожидали, и вы лишь зря потратили время на загрузку образа. Поскольку протокол, используемый для загрузки образа и его исследования, является обычным веб-протоколом, был создан простой инструмент Skopeo, позволяющий получить подробную информацию об образе и вывести ее на экран. *Skopeo* — это греческое слово, означающее *смотреть на расстоянии*.

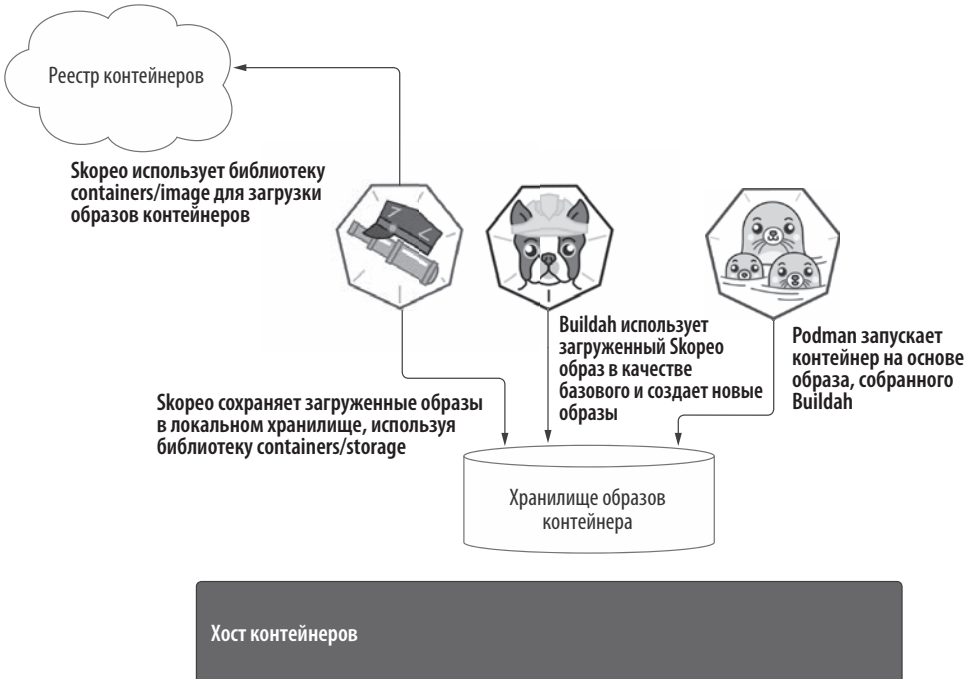


Рис. A.2. Skopeo, Buildah и Podman работают вместе, используя одни и те же образы containers/storage и библиотеку containers/image для загрузки и отправки образов

Выполните команду `skopeo inspect`, чтобы получить подробную информацию об образе в формате JSON:

```
$ skopeo inspect docker://quay.io/rhatdan/myimage
{
  "Name": "quay.io/rhatdan/myimage",
  "Digest": "sha256:fe798c1576dc7b70d7de3b3ab7c72cd22300b061921f052279d887297080
92d8",
  "RepoTags": [
    "Latest",
    "1.0"
  ],
...
```

Skopeo был доработан для копирования образов из реестров. В конечном итоге Skopeo стал инструментом для копирования образов и перемещения копий между

различными типами хранилищ (транспортов). Такими типами хранилищ стали транспорты, приведенные в табл. А.2.

Таблица А.2. Транспорты, поддерживаемые Podman

Транспорт	Описание
Реестр контейнеров (`docker`)	Используется по умолчанию. Ссылается на образ контейнера, хранящийся на удаленном веб-сайте реестра образов контейнеров. Реестры (например, docker.io и quay.io) хранят образы контейнеров и делятся ими
`oci`	Ссылается на образ контейнера, соответствующий спецификации Open Container Initiative Format. Архивные файлы манифеста и слоев находятся в локальном каталоге в виде отдельных файлов
`dir`	Ссылается на образ контейнера. Соответствует схеме размещения образов Docker. Он очень похож на транспорт oci, но хранит файлы в устаревшем формате Docker. Как нестандартизированный формат, он полезен в первую очередь для отладки или неинвазивной диагностики контейнеров
`docker-archive`	Ссылается на образ контейнера, соответствующий макету образа Docker, который упакован в архив TAR
`oci-archive`	Ссылается на образ контейнера, соответствующий спецификации Open Container Initiative Format, который упаковывается в TAR-архив. Он очень похож на транспорт docker-archive, но хранит образ в формате OCI
`docker-daemon`	Ссылается на образ, хранящийся во внутреннем хранилище docker-демона. Поскольку демон Docker требует привилегий root, Podman должен быть запущен от имени пользователя root
`container-storage`	Ссылается на образ контейнера, расположенный в локальном хранилище. Это не транспорт, а скорее механизм для хранения образов. Он может быть использован для преобразования других транспортов в container-storage. По умолчанию Podman использует container-storage для локальных образов

Разработанная в Skopeo функциональность для копирования образов подходила и для использования другими контейнерными движками и инструментами, поэтому Skopeo был разделен на две части: командную строку и базовую библиотеку containers/image. Выделение функциональности в отдельную библиотеку позволило создать другие контейнерные инструменты, в том числе Podman.

Команда `skopeo copy` очень популярна для копирования образов в различных типах контейнерных хранилищ. Ее отличие от Podman и Buildah, как будет показано в разделе А.2, заключается в том, что Skopeo требует от пользователя

указания транспорта для источника и места назначения. В Podman и Buildah по умолчанию используется транспорт docker или containers-storage (в зависимости от контекста и команды). В следующем примере вы скопируете образ из реестра контейнеров с помощью транспорта `docker` и сохраните его локально с помощью транспорта `container-storage`:

```
$ skopeo copy docker://quay.io/rhatdan/myimage containers-storage:quay.io/
     rhatdan/myimage
Getting image source signatures
Copying blob dfd8c625d022 done
Copying blob 68e8857e6dcb done
Copying blob e21480a19686 done
Copying blob fbfcc23454c6 done
Copying blob 3f412c5136dd done
Copying config 2c7e43d880 done
Writing manifest to image destination
Storing signatures
```

Многими пользователями Skopeo применяется команда `skopeo sync`, позволяющая синхронизировать образы в реестрах контейнеров и локальном хранилище.

Skopeo используется в основном в инфраструктурных проектах для обеспечения работы нескольких реестров контейнеров, например, для копирования образов из публичного реестра в частный. В табл. А.3 представлены часто используемые команды Skopeo.

Таблица А.3. Основные команды Skopeo и их описание

Команда	Описание
`skopeo copy`	Копирование образа (манифеста, слоев файловой системы или подписей) из одного места в другое
`skopeo delete`	Пометка имени образа для последующего удаления сборщиком мусора данного реестра
`skopeo inspect`	Возвращение низкоуровневой информации об образе в реестре
`skopeo list-tags`	Вывод списка тегов в хранилище образов для конкретного транспорта
`skopeo login`	Логин в реестр контейнеров (аналогично логину в podman)
`skopeo logout`	Выход из реестра контейнеров (аналогично выходу из реестра podman)
`skopeo manifest digest`	Вычисление дайджеста манифеста для файла манифеста и передача его в стандартный вывод
`skopeo sync`	Синхронизация образов между реестрами контейнеров и локальными каталогами

A.2. BUILDAH

Как вы узнали из раздела 1.1.2, создание образа контейнера означает создание на диске каталога rootfs и добавление в него содержимого, чтобы он выглядел как корневой каталог на Linux-машине. Первоначально это можно было сделать только с помощью `docker build`, используя Dockerfile. Хотя Dockerfiles и Containerfiles и являются прекрасным способом создания «рецептов» для образов контейнеров, требовался низкоуровневый инструмент, позволяющий создавать образы другими методами, — инструмент, позволяющий разбить процесс сборки образа на отдельные команды и использовать для сборки другие, более мощные скриптовые инструменты и языки. Так появился Buildah (https://buildah.io).

Buildah разработан как простой инструмент для создания образов контейнеров. Как Podman и Skopeo, он создавался на основе библиотек containers/storage и containers/image. Функциональность инструмента аналогична функциональности Podman. Он дает возможность загрузить, отправить, зафиксировать образы и даже запустить контейнеры на основе этих образов. Основным отличием Podman от Buildah является базовая концепция *контейнера*. Контейнер Podman — это долгоживущий, действующий контейнер, а контейнер Buildah — это временный рабочий контейнер, который будет использоваться для создания образа OCI.

> **ПРИМЕЧАНИЕ** Buildah — инструмент только для Linux, он недоступен для Mac или Windows. Однако Buildah встроен в команду `podman build` в Podman. Podman на платформах Mac и Windows использует код Buildah на стороне сервера, что позволяет этим платформам выполнять сборку с использованием файлов Containerfiles и Dockerfiles. Более подробная информация приведена в приложениях E и F.

Buildah был задуман так, чтобы сделать определенные в Dockerfile шаги доступными из командной строки. Buildah стремится упростить создание образа контейнера, позволяя использовать для наполнения образа все инструменты, доступные в ОС. Вы можете добавить данные в каталог с помощью стандартных инструментов Linux, таких как `cp`, `make`, `yum install` и т. д. Затем зафиксировать rootfs в файле tar, далее добавить JSON для описания того, что должен делать

образ по мнению его создателя, и наконец, отправить все это в реестр контейнеров. По сути, Buildah разбивает шаги, о которых вы узнали из Containerfile, на отдельные команды, выполняемые из командной строки.

> **ПРИМЕЧАНИЕ** Название *Buildah* — шуточное; это то, как я произношу слово *builder* (строитель). Если бы вы слышали, как я говорю, то заметили бы, что у меня сильный бостонский акцент. Когда команда спросила, как я хочу назвать инструмент, я ответил: «Мне все равно, просто назовите его *builder*». И они услышали «*Buildah*».

Первым шагом при создании нового образа контейнера является загрузка базового образа. В файле Containerfile это делается с помощью инструкции `FROM`.

A.2.1. Создание рабочего контейнера из базового образа

Прежде всего следует обратить внимание на команду `buildah from`. Она эквивалентна инструкции `FROM` файла Containerfile. При выполнении команды `buildah from IMAGE` указанный образ из реестра контейнеров загружается и сохраняется в локальном контейнерном хранилище, а затем создается рабочий контейнер на основе этого образа. Как уже говорилось, этот контейнер похож на контейнер Podman, за исключением того, что он существует временно — только для того, чтобы стать образом контейнера. В следующем примере рабочий контейнер создается на основе образа ubi8- init.

Листинг A.1. Загрузка образа и создание контейнера Buildah

```
$ buildah from ubi8-init
Resolved "ubi8-init" as an alias (/etc/containers/registries.conf.d/
➥ 000-shortnames.conf)
Trying to pull registry.access.redhat.com/
➥ ubi8-init:latest...  ◄──  Загрузка образа из реестра контейнеров
Getting image source signatures
Checking if image destination supports signatures
Copying blob adffa6963146 done
Copying blob 29250971c1d2 done
Copying blob 26f1167feaf7 done
Copying config 4b85030f92 done
Writing manifest to image destination
Storing signatures
ubi8-init-working-container  ◄──  Вывод нового имени контейнера
```

Обратите внимание, что вывод команды `buildah from` выглядит так же, как и вывод команды `podman pull`, за исключением последней строки, в которой выводится

имя контейнера: `ubi8-init-working-container`. Если выполнить команду `buildah from` еще раз, мы получим другое имя контейнера:

```
$ buildah from ubi8-init
ubi8-init-working-container-1
```

Buildah ведет учет своих контейнеров и генерирует имя каждого из них с увеличением счетчика. Конечно, вы можете переопределить имя контейнера с помощью параметра `--name`.

Следующим шагом будет добавление содержимого в образ контейнера.

А.2.2. Добавление данных в рабочий контейнер

Для копирования содержимого файла, URL или каталога в рабочий каталог контейнера в Buildah есть две команды, `buildah copy` и `buildah add`. По своим функциональным возможностям они соответствуют командам `COPY` и `ADD` в Containerfile.

> **ПРИМЕЧАНИЕ** Наличие двух команд, выполняющих практически одно и то же действие, слегка сбивает с толку. В большинстве случаев я рекомендую использовать только `buildah copy` и `COPY` внутри Containerfile. Основное различие между ними заключается в том, что `COPY` копирует в образ контейнера только локальные файлы и каталоги с хоста. Команда `add` поддерживает использование URL-адресов, чтобы загрузить удаленное содержимое и вставить его в контейнер. Команда `ADD` также поддерживает получение файлов TAR и ZIP и их распаковку при копировании в образ контейнера.

Синтаксис команды `buildah copy` предполагает указание имени контейнера, ранее созданного командой `buildah from`, затем источника и, опционально, места назначения. Если пункт назначения не указан, то исходные данные будут скопированы в рабочий каталог контейнера. Каталог назначения будет создан, если до сих пор такого каталога не существовало.

В следующем примере локальный файл html/index.html (созданный ранее в разделе 3.1) копируется в каталог /var/lib/www/html в контейнере:

```
$ buildah copy ubi8-init-working-container html/index.html
➥ /var/lib/www/html/
```

Если для добавления содержимого в контейнеры вы предпочитаете более сложные инструменты, например менеджеры пакетов, это возможно, так как Buildah поддерживает выполнение команд внутри контейнеров.

A.2.3. Выполнение команд в рабочем контейнере

Чтобы выполнить команду внутри контейнера, необходимо выполнить команду `buildah run`. Эта команда работает точно так же, как и инструкция `RUN`: она запускает новый контейнер поверх текущего, выполняет заданную команду и фиксирует результат в работающем контейнере. Синтаксис команды `buildah run` требует указания имени контейнера, за которым следует команда. В следующем примере внутри контейнера устанавливается служба `httpd`:

```
$ buildah run ubi8-init-working-container dnf -y install httpd
Updating Subscription Management repositories.
Unable to read consumer identity
This system is not registered with an entitlement server. You can use
➥ subscription-manager to register.
...
Complete!
```

Чтобы убедиться, что после запуска контейнера у вас будет работающий вебсервер, выполните команду, включающую службу Apache HTTP Server:

```
$ buildah run ubi8-init-working-container systemctl enable httpd.service
Created symlink /etc/systemd/system/multi-user.target.wants/httpd.service ?
➥ /usr/lib/systemd/system/httpd.service.
```

В табл. A.4 показана взаимосвязь между инструкциями Containerfile и командами Buildah.

Таблица A.4. Инструкции Containerfile, сопоставленные с командами Buildah

Инструкция	Команда	Описание
`ADD`	`buildah add`	Добавляет содержимое файла, URL или каталога в контейнер
`COPY`	`buildah copy`	Копирует содержимое файла, URL или каталога в рабочий каталог контейнера
`FROM`	`buildah from`	Создает новый рабочий контейнер с нуля или используя указанный образ в качестве исходной точки
`RUN`	`buildah run`	Выполняет команду внутри контейнера

A.2.4. Добавление содержимого в рабочий контейнер напрямую с хоста

До сих пор я демонстрировал, что Buildah может выполнять те же команды, что и Containerfile, но одной из целей Buildah является предоставление rootfs образа контейнера напрямую хосту. Это позволяет использовать доступные

на хост-машине команды для добавления содержимого в образ контейнера, не требуя при этом присутствия этих команд внутри образа.

Команда `buildah mount` позволяет смонтировать корневую файловую систему рабочего контейнера непосредственно в своей системе и затем использовать такие инструменты, как `cp`, `make`, `dnf` и даже редакторы, для работы с содержимым корневых файлов контейнера.

Если вы запускаете Buildah с правами root, то можете просто выполнить команду `buildah mount`. Но в режиме rootless это недопустимо. Вспомните, в разделе 2.2.10 о применении команды `podman mount` говорилось, что сначала необходимо войти в пользовательское пространство имен. Подобно этому, команда `buildah unshare` создает оболочку, работающую в пользовательском пространстве имен. Войдя в пространство имен, можно монтировать контейнер. В следующем примере, используя полученные ранее знания, вы примените команды `grep` операционной системы хоста для добавления содержимого в контейнер:

```
$ buildah unshare
# mnt=$(buildah mount ubi8-init-working-container)
# echo $mnt
/home/dwalsh/.local/share/containers/storage/
overlay/133e1728eac26589b07984
➥ e3bdf31b5e318159940c866d9e0493a1d08e1d2f6a/merged
# grep dwalsh /etc/passwd >> $mnt/etc/passwd
# exit
```

Проверьте, действительно ли изменения были применены внутри рабочего контейнера:

```
$ buildah run ubi8-init-working-container grep dwalsh /etc/passwd
dwalsh:x:3267:3267:Daniel J Walsh:/home/dwalsh:/bin/bash
```

Вы закончили заполнять содержимое рабочего контейнера, настало время задать другие инструкции из Containerfile. Они будут описывать ваши намерения как создателя образа контейнера.

A.2.5. Конфигурирование рабочего контейнера

Вы, вероятно, заметили, что в табл. А.3 очень много отсутствующих инструкций Containerfile. Такие инструкции в Containerfile, как `LABEL`, `EXPOSE`, `WORKDIR`, `CMD` и `ENTRYPOINT`, применяются для заполнения спецификации образа OCI.

С помощью команды `buildah config` теперь вы можете добавить порт для открытия (`EXPOSE`) и отметить место внутри rootfs-контейнера как том (`VOLUME`), который будет использоваться в качестве корневого каталога веб-сайта:

```
$ buildah config --port=80 --volume=/var/lib/www/html
➥ ubi8-init-working-container
```

Проверьте соответствующие поля спецификации образа OCI с помощью команды `buildah inspect`:

```
$ buildah inspect --format '{{ .OCIv1.Config.ExposedPorts }} {{
➥ .OCIv1.Config.Volumes }}' ubi8-init-working-container
map[80:{}] map[/var/lib/www/html:{}]
```

В табл. A.5 показана связь между инструкциями Containerfile и параметрами конфигурации Buildah.

Таблица A.5. Инструкции Containerfile, сопоставленные с параметрами конфигурации Buildah

Инструкция	Параметр	Описание
`MAINTAINER`	`--author`	Задает контактную информацию автора образа
`CMD`	`--cmd`	Устанавливает команду по умолчанию для запуска внутри контейнера
`ENTRYPOINT`	`--entrypoint`	Задает команду для контейнера, которая будет выполняться как исполняемый файл
`ENV`	`--env`	Устанавливает переменную окружения для всех последующих инструкций
`HEALTHCHECK`	`--healthcheck`	Указывает команду для проверки того, что контейнер все еще работает
`LABEL`	`--label`	Добавляет метаданные в формате ключ — значение
`ONBUILD`	`--onbuild`	Задает команду, которая будет выполняться при использовании данного образа в качестве базового для другого образа
`EXPOSE`	`--port`	Задает порт, который контейнер будет прослушивать во время выполнения
`STOPSIGNAL`	`--stop-signal`	Устанавливает сигнал остановки, который будет передаваться при завершении работы контейнера
`USER`	`--user`	Устанавливает пользователя, который будет использоваться при запуске контейнера и для всех последующих инструкций `RUN`, `CMD` и `ENTRYPOINT`
`VOLUME`	`--volume`	Добавляет точку монтирования и помечает ее как том для внешних данных
`WORKDIR`	`--workingdir`	Устанавливает рабочий каталог для всех последующих инструкций `RUN`, `CMD`, `ENTRYPOINT`, `COPY` и `ADD`

Добавив содержимое в образ контейнера Buildah и конфигурацию в спецификацию образа OCI, нужно создать образ из рабочего контейнера.

A.2.6. Создание образа из рабочего контейнера

Рабочий контейнер, который вы собирали до этого, может быть использован для создания OCI-совместимого образа с помощью команды `buildah commit`. Эта команда работает так же, как и команда `podman commit`, которая рассматривалась в разделе 2.1.9. Входными данными для нее являются имя рабочего контейнера и необязательный тег образа; если тег не указан, образ не будет иметь имени:

```
$ buildah commit ubi8-init-working-container quay.io/rhatdan/myimage2
Getting image source signatures
Copying blob 352ba846236b skipped: already exists
Copying blob 3ba8c926eef9 skipped: already exists
Copying blob 421971707f97 skipped: already exists
Copying blob 9ff25f020d5a done
Copying config 5e47dbd9b7 done
Writing manifest to image destination
Storing signatures
5e47dbd9b7b7a43dd29f3e8a477cce355e42c019bb63626c0a8feffae56fcbf9
```

Образ можно увидеть, выполнив `buildah images`:

```
$ buildah images
REPOSITORY                   TAG     IMAGE ID       CREATED        SIZE
quay.io/rhatdan/myimage2  latest    5e47dbd9b7b7  2 minutes ago   293 MB
registry.access.redhat
➥ .com/ubi8-init         latest    4b85030f924b   5 weeks ago    253 MB
```

Поскольку Podman и Buildah используют одно и то же хранилище образов контейнеров, те же самые образы покажет `podman images`:

```
$ podman images
REPOSITORY               TAG      IMAGE ID      CREATED       SIZE
quay.io/rhatdan/myimage2 latest   5e47dbd9b7b7  4 minutes ago 293 MB
registry.access.redhat
➥ .com/ubi8-init        latest   4b85030f924b  5 weeks ago   253 MB
```

На этом образе можно даже запустить Podman-контейнер:

```
$ podman run quay.io/rhatdan/myimage2 grep dwalsh /etc/passwd
dwalsh:x:3267:3267:Daniel J Walsh:/home/dwalsh:/bin/bash
```

A.2.7. Отправка образа в реестр контейнеров

Так же как и в Podman, в Buildah есть команды `buildah login` и `buildah push`, позволяющие передавать образы в реестры контейнеров, как показано в следующем примере:

```
$ buildah login quay.io
Username: rhatdan
Password:
Login Succeeded!
$ buildah push quay.io/rhatdan/myimage2
Getting image source signatures
Copying blob 3ba8c926eef9 done
Copying blob 421971707f97 done
Copying blob 9ff25f020d5a done
Copying blob 352ba846236b done
Copying config 5e47dbd9b7 done
Writing manifest to image destination
Copying config 5e47dbd9b7 done
Writing manifest to image destination
Storing signatures
```

> **ПРИМЕЧАНИЕ** Для решения той же задачи можно использовать `podman login` и `podman push` или даже `skopeo login` и `skopeo copy`.

Поздравляю! Вы успешно создали OCI-совместимый образ контейнера вручную с помощью простых команд оболочки, без Containerfile. Если необходимо создать образ на основе существующего Containerfile или Dockerfile, в вашем распоряжении есть также команда `buildah build`.

A.2.8. Создание образа из Containerfile

Для создания OCI-совместимого образа из Containerfile или Dockerfile используйте команду `buildah build`. В состав Buildah входит парсер, понимающий формат Containerfile и выполняющий все задачи с помощью описанных ранее команд автоматически. В следующем примере используется файл Containerfile из раздела 2.3.2:

```
$ cat myapp/Containerfile
FROM ubi8/httpd-24
COPY index.html /var/www/html/index.html
```

Containerfile дает возможность создать образ контейнера, выполнив следующую команду:

```
$ buildah build ./myapp
STEP 1/2: FROM ubi8/httpd-24
Resolved "ubi8/httpd-24" as an alias (/home/dwalsh/.cache/containers/
➥ short-name-aliases.conf)
Trying to pull registry.access.redhat.com/ubi8/httpd-24:latest
...
Getting image source signatures
Checking if image destination supports signatures
Copying blob adffa6963146 skipped: already exists
```

```
...
STEP 2/2: COPY html/index.html /var/www/html/index.html
COMMIT
Getting image source signatures
Copying blob 352ba846236b skipped: already exists
...
bbfcf76c994c738f8496c1f274bd009ddbc960334b59a74953691fff00442417
```

Вы, вероятно, заметили, что этот вывод полностью совпадает с выводом команды `podman build`. Это объясняется тем, что команда `podman build` использует Buildah.

A.2.9. Buildah как библиотека

Buildah разрабатывался не только как инструмент командной строки, но и как библиотека на базе языка Golang. Buildah используется в различных инструментах, таких, например, как Podman и OpenShift image builder. Buildah позволяет этим инструментам осуществлять внутреннюю сборку образов OCI. Каждый раз, когда вы выполняете команду `podman build`, вы выполняете код библиотеки Buildah.

Вы научились создавать образы контейнеров с помощью Buildah, копировать образы в контейнерные хранилища с помощью Skopeo, управлять контейнерами и запускать их на хосте с помощью Podman, теперь поговорим о том, как все эти инструменты используются в экосистеме Kubernetes.

A.3. CRI-O: CONTAINER RUNTIME INTERFACE ДЛЯ КОНТЕЙНЕРОВ OCI

Когда Kubernetes только разрабатывался, для запуска контейнеров в нем использовался Docker API. Kubernetes опирался на возможности Docker, менявшиеся от релиза к релизу, что иногда приводило к сбоям в работе Kubernetes. В то же время разработчики CoreOS хотели, чтобы их альтернативный контейнерный движок, получивший название RKT (https://github.com/rkt/rkt), работал с Kubernetes. Тогда разработчики Kubernetes решили разделить функциональность Docker и использовать новый API под названием Container Runtime Interface (CRI; https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/). Этот интерфейс позволяет Kubernetes использовать не только Docker, но и другие контейнерные движки.

Когда Kubernetes собирается загрузить образ контейнера, он обращается к удаленному сокету через CRI и просит загрузить для него образ OCI. При необходимости запустить под или контейнер он обращается к этому сокету и просит его запустить контейнер.

ПРИМЕЧАНИЕ CoreOS в конечном итоге была приобретена компанией Red Hat, а проект RKT прекратил свое существование. Kubernetes отказался от использования Docker в качестве среды выполнения контейнеров.

Компания Red Hat рассматривала CRI как возможность разработки нового контейнерного движка, который в итоге был назван Container Runtime Interface for OCI containers (CRI-O; https://cri-o.io/). CRI-O основан на тех же библиотеках containers/storage и containers/image, что и Skopeo, Buildah и Podman, и может использоваться совместно с этими инструментами. Основной задачей CRI-O является замена Docker в качестве контейнерного движка для Kubernetes.

CRI-O привязан к релизам Kubernetes. Когда выходит новая версия Kubernetes, номера версий синхронизируются. CRI-O оптимизирован для рабочих нагрузок Kubernetes. Инженеры, работающие над ним, понимают задачи Kubernetes и стараются, чтобы CRI-O решал их максимально эффективно. Поскольку CRI-O не имеет других пользователей, Kubernetes не нужно беспокоиться о критических изменениях в CRI-O.

ПРИМЕЧАНИЕ CRI-O является базовой технологией, применяемой в основанной на Kubernetes системе OpenShift компании Red Hat. OpenShift использует Podman для установки и настройки CRI-O перед запуском Kubernetes. В конструктор образов OpenShift встроена функциональность Buildah, позволяющая пользователям создавать образы в кластерах OpenShift.

B Среды выполнения OCI

В этом приложении описаны основные среды выполнения OCI, используемые контейнерными движками типа Podman. Как уже упоминалось в главе 1, среда выполнения OCI (https://opencontainers.org) — это исполняемый файл, запускаемый контейнерными движками, он используется для конфигурирования ядра Linux и подсистем для работы с ядром. Последним его действием является запуск контейнера. Среда выполнения OCI считывает JSON-файл спецификации, затем настраивает пространства имен, элементы управления безопасностью и cgroups и в итоге запускает процесс контейнера (рис. B.1).

В этом приложении вы познакомитесь с четырьмя основными средами выполнения OCI. Параметр `--runtime` позволяет переключаться между различными средами выполнения. В следующем примере одна и та же команда контейнера будет запущена дважды, причем каждый раз с разными средами. В первой команде контейнер запускается со средой выполнения `crun`, определенной в файле containers.conf, поэтому путь к ней указывать не нужно.

Листинг B.1. Запуск Podman с альтернативной средой выполнения OCI crun

```
$ podman --runtime crun run --rm ubi8 echo hi
hi
```

Параметр --runtime дает указание Podman использовать среду выполнения crun OCI, а не стандартную

Параметр `--runtime` дает указание Podman использовать среду выполнения `crun`, а не ту, которая используется по умолчанию. Среда выполнения по умолчанию определяется в таблице `[containers]` в файле containers.conf.

Листинг B.2. Изменение среды выполнения OCI по умолчанию

```
$ grep -iA 3 "Default OCI Runtime" /usr/share/containers/containers.conf
# Default OCI runtime
#
#runtime = "crun"
```

В большинстве систем Podman по умолчанию использует crun. В некоторых старых дистрибутивах, например Red Hat Enterprise Linux, Podman по умолчанию использует runc

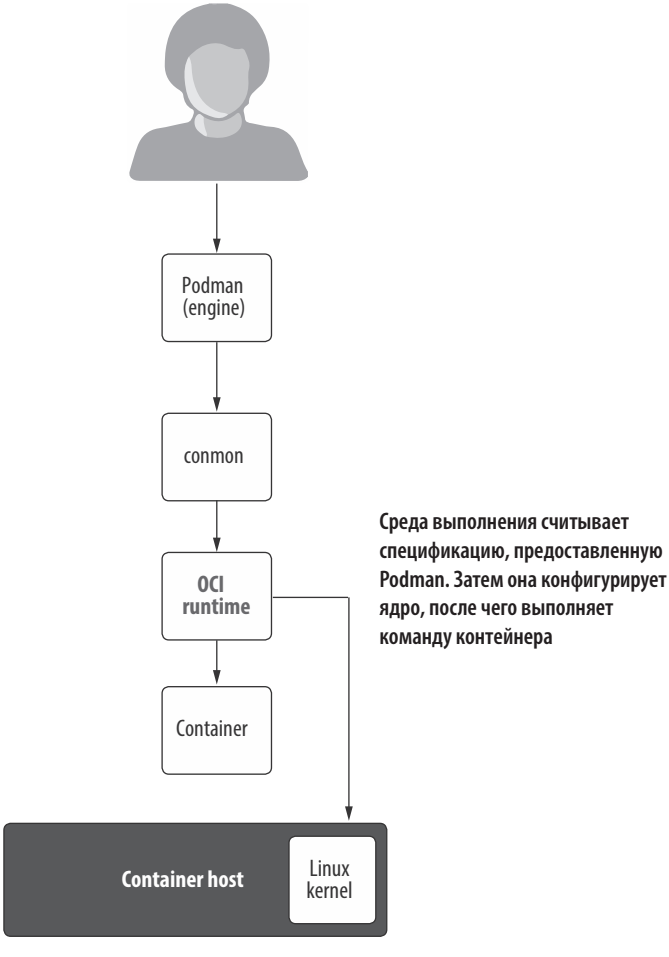


Рис. B.1. Podman вызывает среду выполнения OCI для запуска контейнера

Во втором примере указан полный путь к среде выполнения OCI /usr/bin/runc:

```
$ podman --runtime /usr/bin/runc run –rm ubi8 echo hi
hi
```

Если нужно изменить используемую по умолчанию среду выполнения OCI, задайте параметр `runtime` в таблице `[engine]` в файле containers.conf в вашем домашнем каталоге:

```
$ cat > ~/.config/containers/containers.conf << EOF
[engine]
runtime="runc"
EOF
$ podman --help | grep -- runc
    --runtime stringPath to the OCI-compatible binary used to run containers.
      (default "runc")'
```

> **ПРИМЕЧАНИЕ** Параметр `--runtime` доступен только в Linux. `podman --remote` и, соответственно, Podman на платформах Mac и Windows не поддерживают параметр `--runtime`, поэтому его необходимо задавать в файле containers.conf на стороне сервера.

Более подробную информацию можно найти на странице руководства `podman(1)`: `man podman`.

Среды выполнения OCI постоянно разрабатываются и тестируются. Можно ожидать, что и в дальнейшем в этой области будут появляться нововведения. Первой разработанной контейнерной средой выполнения, ставшей стандартом де-факто, оказалась `runc`.

B.1. RUNC

`runc` — первая среда выполнения OCI (https://github.com/opencontainers/runc). Когда OCI только формировался, компания Docker передала `runc` в OCI в качестве стандартной реализации среды выполнения OCI. OCI продолжает поддерживать и развивать `runc`. `runc` написана на языке Golang и включает в себя библиотеку libcontainer, которая используется во многих контейнерных движках и Kubernetes.

На сайте `runc` указано, что, как и все остальные среды выполнения OCI, `runc` является низкоуровневым инструментом, не предназначенным непосредственно для конечного пользователя. Ее рекомендуется запускать с помощью контейнерных движков, таких как Podman или Docker.

Напомню, что задача контейнерного движка — загрузить образы контейнеров на хост, настроить и смонтировать корневую файловую систему (rootfs), которая будет использоваться внутри контейнера, и наконец, подготовить JSON-файл среды выполнения OCI перед запуском самой среды выполнения.

Спецификация среды выполнения OCI описывает только содержимое JSON-файла, используемого этой средой. Поскольку каждый OCI-движок поддерживает командную строку `runc`, другие среды выполнения OCI применяют те же команды и параметры CLI. Это облегчает замену одной среды выполнения на другую при запуске контейнерного движка. В табл. B.1 приведены команды, поддерживаемые `runc` и, соответственно, всеми остальными средами выполнения OCI.

Таблица B.1. Команды runc

Команда	Описание
`checkpoint`	Создание контрольных точек для запущенного контейнера
`create`	Создание контейнера
`delete`	Удаление всех ресурсов, принадлежащих контейнеру; часто используется для отсоединенных (detached) контейнеров
`events`	Отображение событий контейнера, таких как OOM, статистика использования CPU, памяти и IO
`init`	Инициализация пространств имен и запуск процесса
`kill`	Отправка заданного сигнала (по умолчанию `SIGTERM`) init-процессу контейнера
`list`	Вывод списка контейнеров, запущенных программой `runc` с заданным `--root`
`pause`	Приостановка всех процессов внутри контейнера
`ps`	Отображение процессов, запущенных внутри контейнера
`restore`	Восстановление контейнера из предыдущей контрольной точки
`resume`	Возобновление всех процессов, которые были ранее приостановлены
`run`	Создание и запуск контейнера
`spec`	Создание нового файла спецификации
`start`	Выполнение определенного пользователем процесса в созданном контейнере
`state`	Вывод состояния контейнера
`update`	Обновление ограничений на ресурсы контейнера

`runc` продолжает развиваться и имеет очень активное сообщество. Недостаток `runc` заключается в том, что она написана на языке Golang. Golang не предназначен для небольших, часто выполняемых приложений, которым нужно быстро

запуститься, выполнить команду fork/exec и выйти. Fork/exec — ресурсоемкая операция в Golang, и хотя `runc` пытается обойти эту проблему, со временем она все больше и больше жертвует производительностью. Гораздо лучше с этой точки зрения работает `crun`.

B.2. CRUN

`runc`, написанная на языке Golang, представляет собой очень тяжелый исполняемый файл размером 12 мегабайт. Golang — отличный язык, но он не использует преимущества общих библиотек. Поэтому исполняемые файлы Golang занимают достаточно много памяти. Из-за размера `runc` несколько медленнее загружается при старте контейнера. Еще одна проблема состоит в том, что Golang не очень хорошо поддерживает модель fork/exec — она работает медленнее, чем fork/exec в других языках (например, в C). Этот недостаток быстродействия приобретает критически важное значение при запуске и остановке сотен или тысяч контейнеров, как это происходит в кластере Kubernetes. Контейнерные движки типа Podman, также написанные на Golang, обычно работают намного медленнее, поэтому затраты времени на запуск не так важны. Среды выполнения OCI, такие как `runc`, запускаются в течение очень короткого времени и быстро завершаются.

Джузеппе Скривано (Giuseppe Scrivano), контрибьютор `runc` и Podman, осознал эти недостатки `runc` и решил написать OCI-совместимую среду выполнения на языке C. Он создал легковесную среду выполнения под названием `crun`.

`crun` описывает себя как «*быструю и легковесную среду выполнения OCI*» (https://github.com/containers/crun). Она поддерживает все те же команды и параметры, что и `runc`, а исполняемый файл `crun` во много раз меньше соответствующего файла `runc`. Для сравнения размеров выполните команду `du -s`:

```
$ du -s /usr/bin/runc /usr/bin/crun
14640   /usr/bin/runc
392   /usr/bin/crun
```

Поскольку `crun` написана на языке C, она гораздо эффективнее поддерживает fork и exec, чем Golang, и, следовательно, гораздо быстрее запускает контейнер.

Это также позволяет ей легко подключаться к другим библиотекам в системе, и сейчас ведутся эксперименты по использованию `crun` в качестве библиотеки для обработки JSON-файла среды выполнения OCI и запуска различных типов контейнеров (например, контейнеров WASM и Windows в Linux). У `crun` есть и потенциал для запуска KVM-разделенных контейнеров, основанных на libkrun.

В настоящее время `crun` является средой выполнения OCI по умолчанию, используемой Podman в Fedora и в Red Hat Enterprise Linux 9. `runc` продолжает

поддерживаться как среда выполнения по умолчанию в Red Hat Enterprise Linux 8.

`crun` и `runc` являются двумя основными средами выполнения для управления традиционными контейнерами, использующими разделение пространств имен. Оба проекта работают в тесном сотрудничестве. Обнаруженные в одном из средств ошибки или проблемы быстро исправляются в обоих. Более подробную информацию можно найти на man-странице `crun(1)`: `man crun`.

B.3. KATA

Среды выполнения OCI также написаны для использования VM-разделения, основным примером которого являются контейнеры Kata. Проект Kata Containers (https://katacontainers.io) позиционирует себя следующим образом: «*Скорость контейнеров, безопасность виртуальных машин. Kata Containers — это среда выполнения контейнеров с открытым исходным кодом, создающая легковесные виртуальные машины, которые легко подключаются к экосистеме контейнеров*».

Контейнеры Kata используют технологию виртуальных машин для запуска каждого контейнера, что существенно отличается от запуска виртуальной машины и работы Podman внутри нее. Стандартная VM имеет init-систему, которая запускает всевозможные сервисы, такие как системы логирования, cron и т. д. С другой стороны, Kata-контейнер запускает микро ОС, в которой работает только сам контейнер и его вспомогательные сервисы (рис. B.2). Поскольку ее единственной целью является запуск контейнера, при завершении работы контейнера эта VM исчезает.

Я считаю, что запуск контейнеров с разделением с помощью VM и гипервизора обеспечивает лучшую безопасность, чем традиционное разделение контейнеров, при котором они напрямую взаимодействуют с ядром хоста. Контейнер, отделенный от VM, должен сначала выйти из-под контроля внутри VM, затем найти способ обойти контроль гипервизора и только после этого совершить атаку на ядро хоста.

Хотя контейнеры, разделенные на виртуальные машины, более безопасны, этот способ имеет и свои недостатки. Запуск Kata-контейнера, настройка гипервизора, запуск ядра и других процессов внутри VM, а затем и самого контейнера

требуют значительных накладных расходов. Память, процессор и другие ресурсы должны быть предварительно выделены для VM, и их трудно изменить. Запуск Kata внутри VM в облаке зачастую невозможен или, по крайней мере, более затратен, поскольку большинство облачных провайдеров не одобряют вложенную виртуализацию.

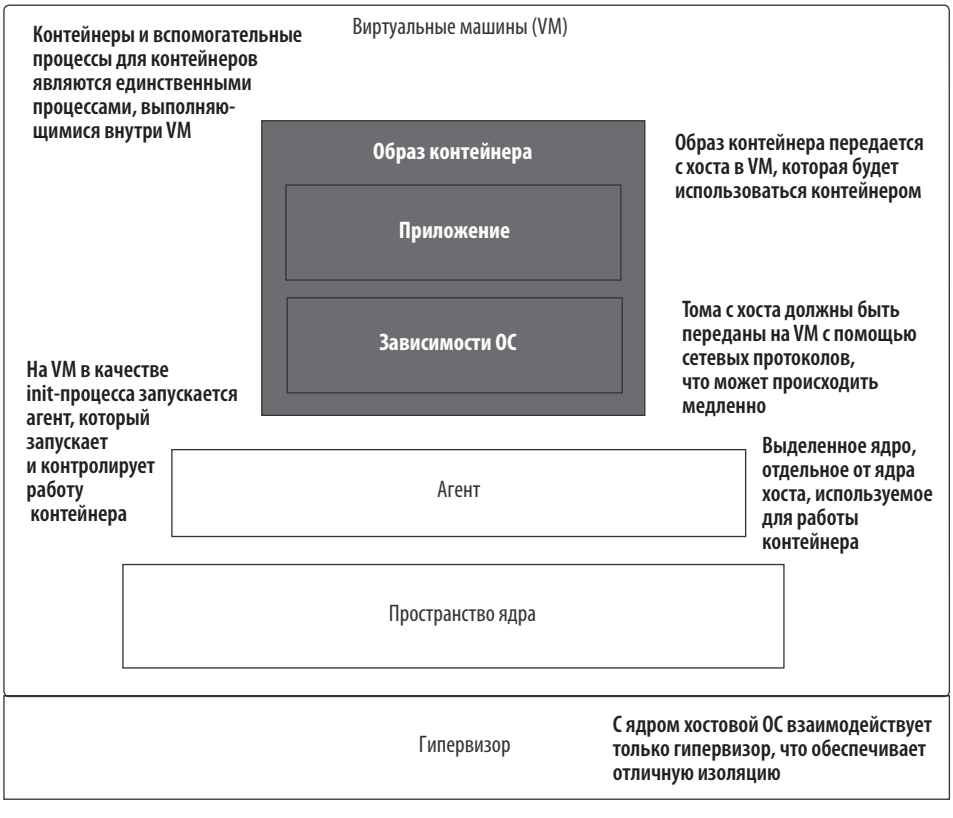


Рис. В.2. Kata-контейнеры запускают легковесную VM, на которой работает только контейнер

И наконец, самое главное: контейнеры, разделенные VM, по своей природе испытывают трудности с обменом содержимым с другими контейнерами и операционной системой хоста. Самая большая проблема связана с томами.

Если в обычных контейнерах для обмена содержимым с хост-машиной достаточно просто монтировать тома, то в контейнерах с разделением по виртуальным машинам такое монтирование не работает. Поскольку процессы на хосте

и в контейнере работают на двух разных ядрах, для обмена содержимым необходим сетевой протокол. Изначально контейнеры Kata использовали сетевые файловые системы NFS и Plan 9. Чтение и запись данных через эти сетевые файловые системы происходят значительно медленнее, чем чтение и запись через собственную файловую систему.

Virtiofs — это новая файловая система, обладающая свойствами сетевой файловой системы, но позволяющая виртуальным машинам получать доступ к файлам на хосте. Она уже способна продемонстрировать значительный прирост скорости по сравнению с сетевыми файловыми системами, хотя все еще находится в стадии активной разработки.

Kata-контейнеры запускаются двумя способами. Традиционно для Kata используется командная строка `kata-runtime`, основанная на командах `runc`, поддерживаемых Podman. Пути, заданные в файле containers.conf, можно увидеть на Linux-машине, выполнив поиск `#kata`:

```
$ grep -A 9 '^#kata' /usr/share/containers/containers.conf
#kata = [
# "/usr/bin/kata-runtime",
# "/usr/sbin/kata-runtime",
# "/usr/local/bin/kata-runtime",
# "/usr/local/sbin/kata-runtime",
# "/sbin/kata-runtime",
# "/bin/kata-runtime",
# "/usr/bin/kata-qemu",
# "/usr/bin/kata-fc",
#]
```

В итоге Kata-контейнеры обеспечивают более высокий уровень безопасности при снижении производительности. Выбор одного из вариантов среды выполнения OCI нужно осуществлять, ориентируясь на вашу рабочую нагрузку.

B.4. GVISOR

Последняя среда выполнения OCI, которую я рассматриваю в этом приложении, — gVisor (https://gvisor.dev/). На сайте gVisor позиционируется как «*ядро приложений для контейнеров, обеспечивающее эффективную глубокую защиту везде*».

gVisor включает в себя среду выполнения OCI под названием `runsc` и работает с Podman и другими контейнерными движками. Проект gVisor называет себя ядром приложений, написанным на языке Golang и реализующим значительную часть интерфейса системных вызовов Linux. Он обеспечивает дополнительный уровень изоляции между запущенными приложениями и операционной системой хоста. Инженеры Google разработали первоначальные версии gVisor и утверждают, что большая часть контейнеров, запускаемых в Google Cloud, использует среду выполнения gVisor.

gVisor похож на контейнеры, изолированные средствами VM, в том смысле, что gVisor перехватывает почти все системные вызовы внутри контейнера и затем обрабатывает их. В то же время у него нет проблем с вложенной виртуализацией, как у Kata.

Однако gVisor ведет к снижению производительности за счет дополнительных циклов CPU и использования большего объема памяти. Это может привести к увеличению задержек, снижению пропускной способности или к тому и другому сразу. В gVisor также представлена независимая реализация системных вызовов, а значит, многие подсистемы или специфические вызовы не так оптимизированы, как в более развитых реализациях.

C Установка Podman

Podman — отличный инструмент для работы с контейнерами, но как установить его в свою систему? Какие пакеты необходимы для его работы? В этом приложении рассказывается об установке или сборке Podman в вашей системе.

C.1. УСТАНОВКА PODMAN

Podman доступен практически для всех дистрибутивов Linux через их пакетные менеджеры. Он также доступен для платформ Mac, Windows и FreeBSD. Официальный сайт podman.io, https://podman.io/getting-started/installation, регулярно пополняется новыми инструкциями по установке для различных дистрибутивов. Как видно из рис. C.1, большая часть материалов данного приложения взята с сайта podman.io.

C.1.1. macOS

Поскольку Podman — это инструмент для запуска Linux-контейнеров, использовать его на компьютере с macOS можно только при наличии локального или удаленного доступа к Linux-машине. Чтобы несколько упростить этот процесс, в Podman применяется команда `podman machine` для автоматического управления виртуальными машинами.

HOMEBREW

Клиент для Mac доступен через Homebrew (https://brew.sh/):

```
$ brew install podman
```

В Podman предусмотрена возможность установки виртуальной машины и запуска экземпляра Linux на вашей машине с помощью команды `podman machine`. На компьютере Mac для успешного локального запуска контейнеров необходимо выполнить следующие команды для установки и запуска виртуальной машины Linux:

```
$ podman machine init
$ podman machine start
```

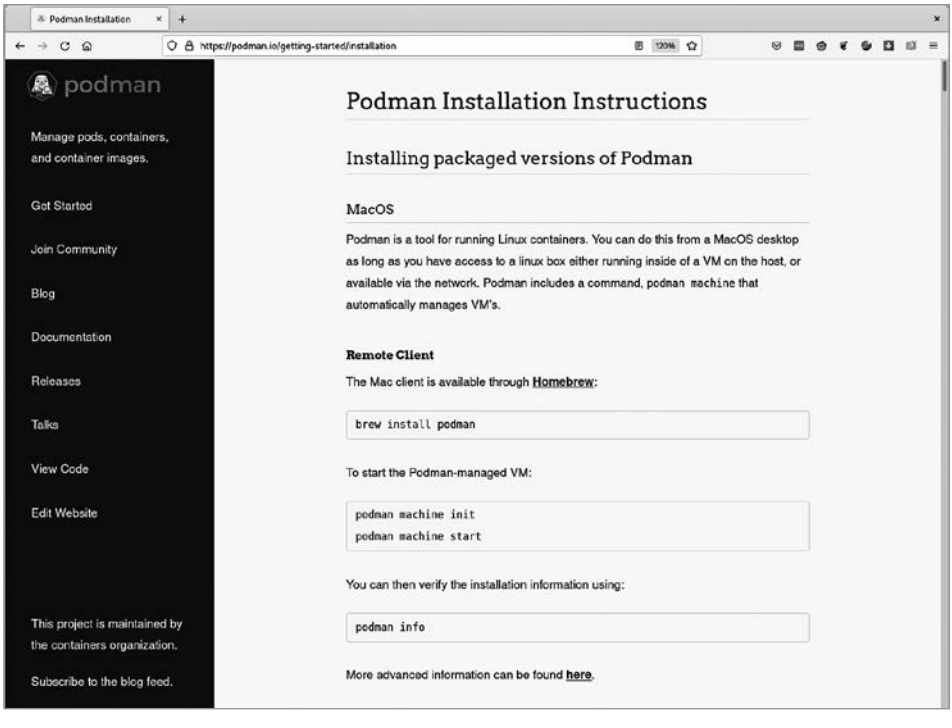


Рис. С.1. Веб-сайт с инструкциями по установке Podman

Для установки SSH-соединений с удаленными Linux-машинами, на которых работает служба Podman, дополнительно используется команда `podman system connection`.

Информация об установленной системе проверяется с помощью следующей команды:

```
$ podman info
```

Команда Podman выполняется непосредственно на Mac, но взаимодействует с экземпляром Podman, запущенным на виртуальной машине.

С.1.2. Windows

Поскольку Podman — это инструмент для запуска Linux-контейнеров, использовать его в Windows удастся только в том случае, если у вас есть доступ к Linux-системе, работающей локально или удаленно. В Windows Podman может также использовать подсистему WSL (Windows Subsystem for Linux).

WINDOWS REMOTE CLIENT

Последняя версия удаленного клиента для Windows находится здесь: //github.com/containers/podman/releases.

После установки следует настроить клиент Windows Remote на подключение к серверу Linux с помощью команды podman system connection. Подробная информация об этом процессе представлена на сайте https://docs.podman.io/en/latest/markdown/podman-system-connection.1.html.

Подсистема WINDOWS для LINUX (WSL) 2.0

Посмотрите документацию Windows по установке WSL 2.0, а затем выберите включающий Podman дистрибутив из описанных ниже. В качестве альтернативы воспользуйтесь командой `podman machine init`, которая автоматически установит и настроит WSL, загрузит и установит на нее Fedora Core VM, а также создаст соответствующие SSH-соединения для удаленного клиента Podman.

ПРИМЕЧАНИЕ WSL 1.0 не поддерживается.

С.1.3. Arch Linux и Manjaro Linux

В Arch Linux и Manjaro Linux для установки приложений используется инструмент `pacman`:

```
$ sudo pacman -S podman
```

С.1.4. CentOS

Podman доступен в стандартном репозитории Extras для CentOS 7 и в репозитории AppStream для CentOS 8 и Stream:

```
$ sudo yum -y install podman
```

C.1.5. Debian

Пакет podman доступен в репозиториях Debian 11 (bullseye) и более поздних версий:

```
$ sudo apt-get -y install podman
```

C.1.6. Fedora

```
$ sudo dnf -y install podman
```

C.1.7. Fedora-CoreOS, Fedora Silverblue

Podman предустановлен в этих дистрибутивах, в его установке необходимости нет.

C.1.8. Gentoo

```
$ sudo emerge app-emulation/podman
```

C.1.9. OpenEmbedded

Рецепты BitBake для Podman и его зависимостей доступны в слое метавиртуализации (https://git.yoctoproject.org/meta-virtualization/). Добавьте этот слой в среду сборки OpenEmbedded и собирайте Podman, используя:

```
$ bitbake podman
```

C.1.10. openSUSE

```
sudo zypper install podman
```

C.1.11. openSUSE Kubic

В дистрибутив openSUSE Kubic уже встроен Podman. Необходимость в его установке отсутствует.

C.1.12. Raspberry Pi OS arm64

Raspberry Pi OS использует стандартные репозитории Debian, поэтому она полностью совместима с репозиторием Debian arm64:

```
$ sudo apt-get -y install podman
```

C.1.13. Red Hat Enterprise Linux

RHEL7

Убедитесь, что у вас есть подписка на RHEL7, затем включите репозиторий extras и установите Podman:

```
$ sudo subscription-manager repos --enable=rhel-7-server-extras-rpms
$ sudo yum -y install podman
```

> **ПРИМЕЧАНИЕ** RHEL7 больше не получает обновлений пакета Podman, за исключением исправлений безопасности.

RHEL8

Podman входит в состав модуля `container-tools` вместе с Buildah и Skopeo:

```
$ sudo yum module enable -y container-tools:rhel8
$ sudo yum module install -y container-tools:rhel8
```

RHEL9 (AND BEYOND)

```
$ sudo yum install podman
```

C.1.14. Ubuntu

Пакет podman доступен в официальных репозиториях для Ubuntu 20.10 и новее:

```
$ sudo apt-get -y update
$ sudo apt-get -y install podman
```

C.2. СБОРКА PODMAN ИЗ ИСХОДНОГО КОДА

Обычно я рекомендую использовать пакетные версии Podman, поскольку для успешной работы Podman в Linux требуется установка дополнительных инструментов, таких как `conmon` (мониторинг контейнеров), `containernetworking-plugins` (настройка сети) и `containers-common` (общая конфигурация). Хотя процесс сборки Podman из исходных текстов не очень сложен, список зависимостей для разных дистрибутивов Linux различается. С последними инструкциями вы всегда можете ознакомиться на странице https://podman.io/docs/installation.

C.3. PODMAN DESKTOP

Существует также графический интерфейс Podman Desktop для просмотра и исследования контейнеров, а также образов различных контейнерных движков,

и управления ими. Он доступен по адресу https://github.com/containers/podman-desktop. Podman Desktop предоставляет возможность одновременного подключения к нескольким движкам и обеспечивает унифицированный интерфейс. Это относительно новый проект, находящийся в стадии активной разработки, поэтому в нем вполне воможны некоторые шероховатости.

В качестве справки приведу информацию, что в сентябре 2021 года компания Docker Inc. объявила о начале взимания платы за ранее бесплатную версию Docker Desktop на macOS. Объявление Docker заставило многих людей искать ему замену.

РЕЗЮМЕ

- Podman — это инструмент для запуска Linux-контейнеров, поэтому он работает только в Linux.
- Podman доступен в стандартных репозиториях большинства основных дистрибутивов Linux.
- Podman доступен в виде удаленного клиента на платформах Mac и Windows, который подключается к локальной или удаленной системе Linux.
- Podman предоставляет специальную команду для управления виртуальными машинами Linux в операционных системах macOS и Windows.
- Podman может быть собран из исходного кода, но для его успешной работы требуется множество дополнительных инструментов.
- Podman Desktop — это альтернатива популярному Docker Desktop.

D Участие в проекте Podman

Больше всего в open source (проектах с открытым исходном кодом) мне нравится участие сообщества. Здорово, когда ты можешь внести свой вклад в проект, а еще лучше — привлечь людей к участию в твоем проекте. Мне нравится проводить аналогию со сказкой братьев Гримм «Эльфы» (https://sites.pitt.edu/~dash/grimm039.html):

> *Один сапожник, не по своей вине, стал настолько беден, что кожи у него хватило только на одну пару обуви. Однажды вечером он выкроил ее и отправился спать, намереваясь закончить на следующее утро. С чистой совестью он спокойно лег в постель, вверил себя Богу и заснул. На следующее утро, помолившись, он уже собирался вернуться к работе, как вдруг обнаружил на верстаке полностью готовые сапоги.*

Далее в рассказе говорится о паре эльфов, которые приходят каждую ночь и доделывают обувь. Я считаю, что именно так и работает open source. По сути, все люди, которые вносят свой небольшой вклад — сообщают об ошибках, исправляют ошибки, исправляют документацию, запрашивают новые функции, рекламируют проект, — это всё эльфы. Иногда я ложусь спать, а когда просыпаюсь, обнаруживаю, что кто-то исправил проблему, с которой я бился накануне вечером! А иногда эльфы вырастают в мейнтейнеров. Небольшие вклады некоторых со временем увеличиваются, и эти разработчики становятся основными членами команды Podman. Кое-кого мы даже наняли на работу.

D.1. ВСТУПЛЕНИЕ В СООБЩЕСТВО

Каждое небольшое изменение помогает сделать проект лучше. Когда я разговариваю со студентами колледжей об открытом исходном коде, я рассказываю

им об уникальных возможностях, которых не было, когда я сам был студентом. Они могут внести свой вклад в программный проект или продукт, а затем указать это в своем резюме. При собеседовании со студентом, претендующим на стажировку или работу, наличие в резюме нескольких работ на github.com производит сильное впечатление.

Podman и лежащие в его основе технологии постоянно нуждаются в новых разработках (рис. D.1). Никакой вклад не является слишком маленьким — от исправления орфографической ошибки на man-странице до разработки полноценной функциональности. Чтобы внести свой вклад, не обязательно быть разработчиком. Нам всегда нужна помощь в работе над документацией, веб-дизайном для podman.io, а также в разработке программного обеспечения. Многие отличные идеи исходят от пользователей продукта. Простое сообщение об ошибке или о том, что вам не нравится, может привести к улучшению проекта. Я часто прошу людей, создавших сложные окружения с помощью Podman, поделиться этим в блоге с другими.

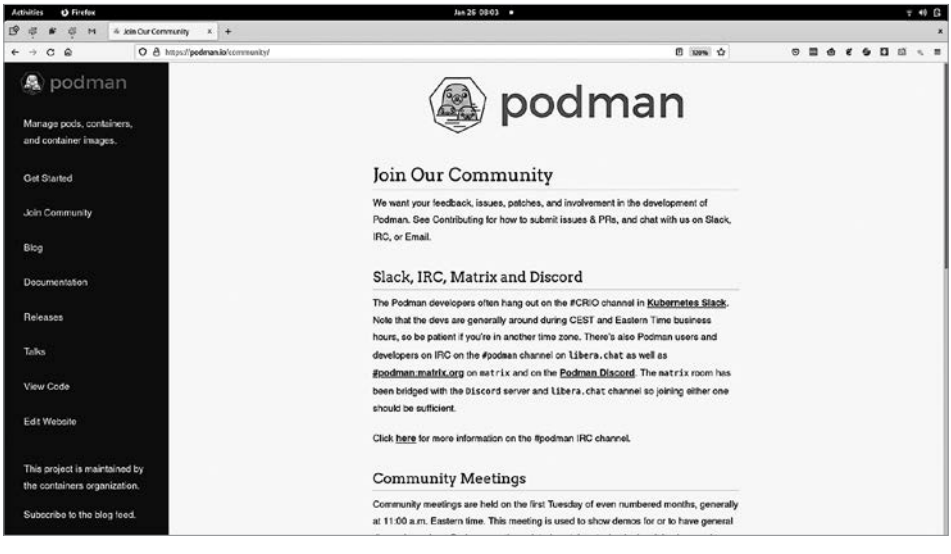


Рис. D.1. Страница сообщества Podman (https://podman.io/community)

Podman — сообщество инклюзивное, как и все проекты github.com/containers. Положение о кодексе поведения для проекта containers на сайте гласит (см. https://github.com/containers/common/blob/main/CODE-OF-CONDUCT.md):

Будучи участниками и мейнтейнерами проектов в репозитории https://github.com/containers, в интересах развития открытого и гостеприимного сообщества, мы обязуемся уважать всех людей, которые вносят свой вклад, сообщая о проблемах, публикуя запросы на функциональность, обновляя документацию, подавая запросы на исправления или патчи, а также выполняя другие действия в рамках любого из проектов под эгидой проекта containers.

D.2. PODMAN НА GITHUB.COM

Вопросы, обсуждения и пул-реквесты находятся в репозитории https://github.com/containers/podman (рис. D.2). На момент написания материала проект имеет более 1200 ответвлений и 12 000 звезд. Очевидно, это очень активный проект.

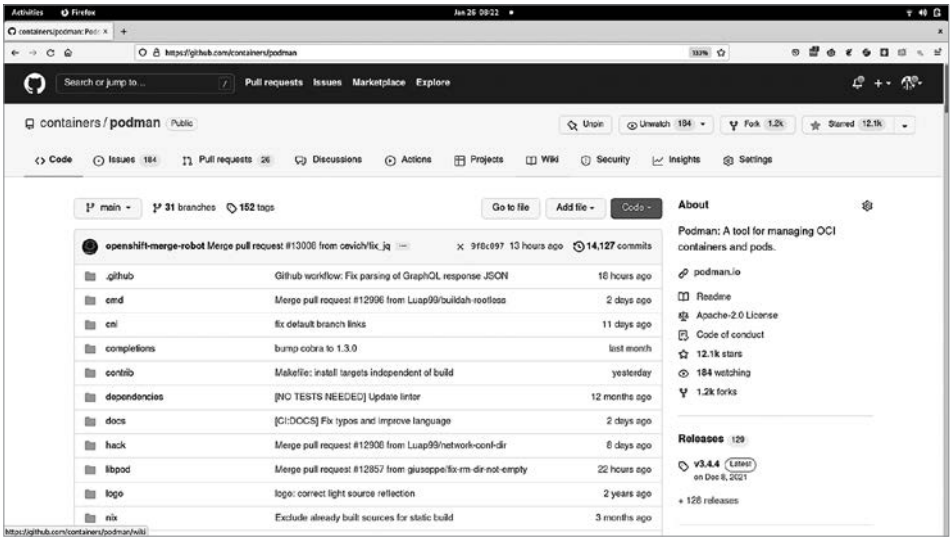


Рис. D.2. Страница Podman на github (github.com/containers/podman)

Вы можете напрямую обращаться к основной группе мейнтейнеров в IRC на канале #podman на сайте libera.chat. IRC-канал также связан с #podman:matrix.org (https://matrix.to/#/#podman:matrix.org) на Matrix и Podman Discord (https://discord.com/invite/x5GzFF6QH4).

Существует и почтовая рассылка, для присоединения к которой нужно отправить письмо по адресу podman: join@lists.podman.io. Наконец, вы можете следить в Twitter за @podman_io или за мной — @rhatdan.

E Podman на macOS

В ЭТОМ ПРИЛОЖЕНИИ

- ✓ Установка Podman на macOS
- ✓ Использование команды `podman machine init` для загрузки VM с установленной службой Podman
- ✓ Использование команды `podman` для связи со службой Podman, запущенной на VM
- ✓ Запуск или остановка VM с помощью команд `podman machine start/stop`

Podman — это инструмент для запуска Linux-контейнеров. Контейнеры Linux требуют наличия ядра Linux. Как бы мне ни хотелось убедить весь мир перейти на Linux Desktop, которым пользуюсь я, большинство пользователей работают в операционных системах macOS и Windows, — возможно, и вы тоже. Если вы используете Linux Desktop, это здорово!

Если вы не работаете на macOS, пропустите это приложение.

Раз вы не пропустили это приложение, предположу, что вы хотите создавать Linux-контейнеры без необходимости подключаться по ssh к Linux-машине.

Скорее всего, вы будете использовать штатные средства разработки программного обеспечения и вести разработку локально.

Одним из способов достижения этой цели является запуск Podman в качестве службы на Linux-компьютере и использование команды `podman --remote` для связи с этой службой. В Podman предусмотрена команда `podman system connection` для настройки взаимодействия Podman с Linux-компьютером. Однако это трудоемкий процесс, требующий выполнения ряда действий вручную. Обновленное руководство по этому процессу можно найти на веб-странице https://github.com/containers/podman/blob/main/docs/tutorials/remote_client.md.

Более эффективным способом является использование специальной команды `podman machine`, которая объединяет все эти шаги и расширяет возможности управления Linux-машиной. В этом приложении вы узнаете, как установить Podman на macOS, а затем применять команды `podman machine` для установки, настройки виртуальной машины и управления ею, что позволит вам использовать собственный клиент Podman для запуска контейнеров. Первым шагом к запуску Podman в macOS является его установка. Клиент для macOS доступен через Homebrew (https://brew.sh/).

> **ПРИМЕЧАНИЕ** Homebrew описывает себя как «...самый простой и гибкий способ установки UNIX-инструментов, которые Apple не включила в macOS» (https://docs.brew.sh/Manpage).

Утилита Homebrew — это лучший способ установки открытого программного обеспечения в macOS. Если в настоящее время в вашей macOS нет Homebrew, откройте терминал и установите ее с помощью следующей команды:

```
$ /bin/bash -c "$(curl -fsSL
➥ https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Теперь выполните команду `brew` для установки урезанной версии Podman с поддержкой только `--remote` в каталог /opt/homebrew/bin:

```
$ brew install podman
```

Если у вас нет доступа к виртуальной машине Linux или удаленному серверу Linux, Podman позволяет создать локально работающую виртуальную машину с помощью команды `podman machine`. Это упрощает задачу, создавая и конфигурируя VM со включенной службой Podman.

> **ПРИМЕЧАНИЕ** Если у вас есть Linux-машина, вы можете использовать команды подключения Podman для установки соединений.

E.1. ИСПОЛЬЗОВАНИЕ КОМАНД PODMAN MACHINE

Команды `podman machine` позволяют загрузить виртуальную машину из интернета, запустить ее, остановить или удалить. Эта VM содержит предустановленную службу Podman. Кроме того, команда создает SSH-соединение и добавляет информацию об этом в хранилище `podman system connection`, что значительно упрощает процесс настройки среды `podman-remote`. В табл. E.1 перечислены все команды (или, точнее, подкоманды) `podman machine`, используемые для управления жизненным циклом виртуальной машины Podman. Первым шагом является инициализация новой виртуальной машины в системе с помощью команды `podman machine init`, которая описана в следующем разделе.

Таблица E.1. Команды podman machine

Команда	Описание
`init`	Инициализация новой виртуальной машины
`list`	Вывод списка виртуальных машин
`rm`	Удаление виртуальной машины
`ssh`	Подключение по ssh к виртуальной машине. Это удобно для входа в виртуальную машину и выполнения штатных команд Podman. Некоторые команды Podman не поддерживаются удаленно, кроме того, вам может потребоваться изменить некоторые конфигурации внутри виртуальной машины
`start`	Запуск виртуальной машины
`stop`	Остановка виртуальной машины. Если вы не используете контейнеры, вам может потребоваться завершить работу виртуальной машины для экономии системных ресурсов

E.1.1. podman machine init

Команда `podman machine init` загружает и настраивает VM на системе macOS (рис. E.1). По умолчанию загружается последний выпущенный образ `fedora-coreos` (https://getfedora.org/en/coreos), если он не был загружен ранее. Fedora CoreOS — это минимизированная операционная система, предназначенная для запуска контейнеров.

> **ПРИМЕЧАНИЕ** Эта виртуальная машина имеет относительно большой размер, и ее загрузка занимает несколько минут.

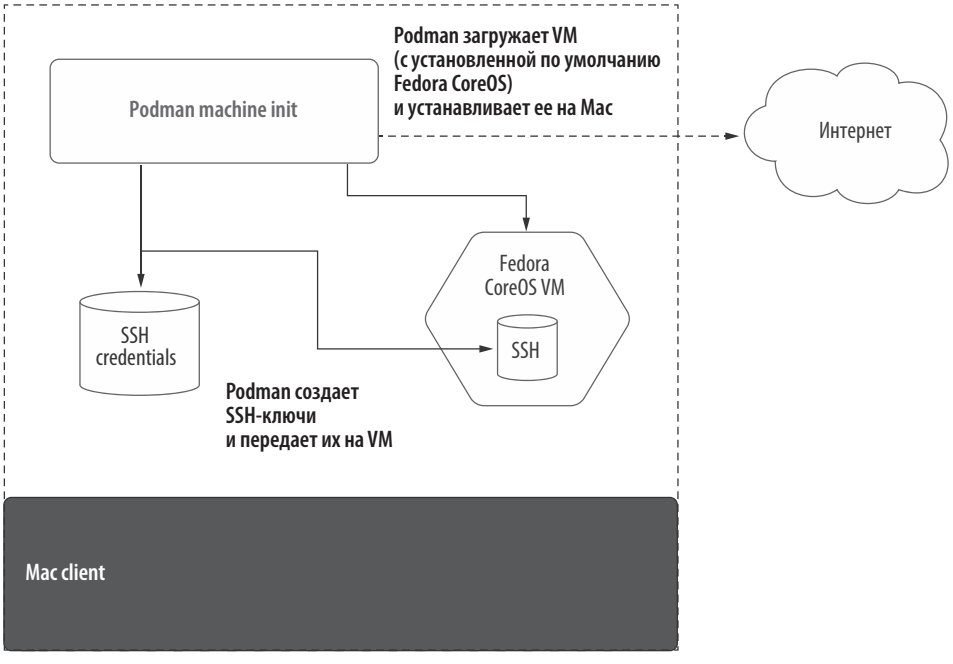


Рис. E.1. Команда podman machine init загружает VM и настраивает SSH-соединения

Листинг E.1. Podman загружает виртуальную машину на Mac и подготавливает ее к запуску

```
$ podman machine init
Downloading VM image: fedora-coreos-35.20211215.2.
➥ 0-qemu.x86_64.qcow2.xz
[=========>----------------------------------------------] 111.0MiB /
➥ 620.7MiB
Downloading VM image: fedora-coreos-35.20211215.2.0-qemu.x86_64.qcow2.xz: done
Extracting compressed file
```

Podman находит и загружает в систему последний образ fedora-coreos qcow

После загрузки образа Podman распаковывает его и настраивает qemu для его запуска. Также настраивается SSH-соединение в хранилище системных соединений Podman

Podman предварительно конфигурирует VM, определяя объем памяти, размер диска и количество процессоров, которые она будет использовать. Эти значения настраиваются с помощью параметров команды `init`. Таблица E.2 описывает эти параметры.

Таблица E.2. Параметры команды podman machine init

Параметр	Описание
`--cpus uint`	Количество CPU (по умолчанию 1)
`--disk-size uint`	Размер диска в гигабайтах (по умолчанию 10). Это очень важный параметр, поскольку он ограничивает количество контейнеров и образов, которые можно использовать в VM. Если у вас есть свободное место, я рекомендую увеличить это поле
`--image-path string`	Путь к `qcow`-образу (по умолчанию — `testing`). Podman имеет два встроенных образа Fedora CoreOS, которые он может загрузить: `testing` и `stable`. Вы также можете выбрать другие ОС и VM для загрузки, но эти VM должны поддерживать файлы CoreOS/Ignition (https://coreos.github.io/ignition/)
`--memory integer`	Память в мегабайтах (по умолчанию — 2048). Для работы VM требуется определенный объем оперативной памяти, и в зависимости от контейнеров, которые вы хотите запустить в VM, вам может понадобиться больше выделяемого по умолчанию

После того как `podman machine init` завершит загрузку и установку VM, вы можете увидеть ее с помощью команды `podman machine list`. Обратите внимание, что символ * указывает на используемую по умолчанию VM. В настоящее время команда `podman machine` поддерживает одновременную работу только одной VM:

```
$ podman machine list
NAME                     VM TYPE    CREATED      LAST UP     CPUS
➥ MEMORY DISK SIZE
podman-machine-default*  qemu       2 minutes ago  2 minutes ago 1
➥ 2.147GB 10.74GB
```

В следующем разделе мы рассмотрим автоматически созданное SSH-соединение.

E.1.2. Настройка SSH для машины Podman

Команда `podman machine init` предоставляет ОС с Ignition конфигурацией, которая включает SSH-ключ для пользователя `core`. Затем Podman добавляет SSH-соединения на клиентской машине для режимов rootless и rootful, настраивает учетную запись пользователя и добавляет необходимые пакеты и конфигурации в VM. Конфигурация SSH позволяет выполнять SSH-команды без пароля для учетных записей `core` и `root` с клиента. Команда `podman machine init` также настраивает информацию о системных соединениях Podman (раздел 9.5.4). База данных системных подключений конфигурируется как для пользователя rootful,

так и для пользователя rootless внутри VM. Если предыдущие соединения отсутствуют, то команда `podman machine init` сделает вновь созданное соединение соединением по умолчанию.

Получить список всех соединений можно с помощью команды `podman system connection list`. Соединение по умолчанию, `podman-machine-default`, является rootless:

```
$ podman system connection list
Name URI
Identity Default
podman-machine-default
➥ ssh://core@localhost:50107/run/user/501/podman/podman.sock
➥ /Users/danwalsh/.ssh/podman-machine-default true
podman-machine-default-root
➥ ssh://root@localhost:50107/run/podman/podman.sock
➥ /Users/danwalsh/.ssh/podman-machine-default false
```

Иногда для запуска контейнеров требуются привилегии root, а они не могут работать в режиме rootless. Необходимо изменить системное соединение так, чтобы оно по умолчанию подключалось к rootful-службе, используя команду `podman system connection default`:

```
$ podman system connection default podman-machine-default-root
```

Проверьте соединения еще раз, чтобы убедиться, что теперь соединением по умолчанию является `podmanmachine-default-root`:

```
$ podman system connection list
Name                    URI
➥ Identity                                Default
podman-machine-default
➥ ssh://core@localhost:50107/run/user/501/podman/podman.sock
➥ /Users/danwalsh/.ssh/podman-machine-default false
podman-machine-default-root
➥ ssh://root@localhost:50107/run/podman/podman.sock
➥ /Users/danwalsh/.ssh/podman-machine-default true
n-machine-default ssh://root@localhost:38243/run/podman/podman.sock
```

Теперь все команды Podman подключаются непосредственно к службе Podman, запущенной под учетной записью root. Измените соединение по умолчанию на соединение с rootless-пользователем, снова применив команду `podman system connection default`:

```
$ podman system connection default podman-machine-default
```

Если вы попытаетесь запустить контейнер Podman на этом этапе, это не получится, поскольку VM на самом деле не запущена. Необходимо ее запустить.

E.1.3. Запуск виртуальной машины

После добавления VM и установки определенного соединения в качестве соединения по умолчанию попробуйте выполнить команду:

```
$ podman version
Cannot connect to Podman. Please verify your connection to the Linux system
using 'podman system connection list', or try 'podman machine init' and
'podman machine start' to manage a new Linux VM
Error: unable to connect to Podman. failed to create sshClient: Connection
to bastion host (ssh://root@localhost:38243/run/podman/podman.sock)
failed.: dial tcp [::1]:38243: connect: connection refused
```

Как указывает сообщение об ошибке, виртуальная машина не запущена.

Запуск одной VM осуществляется с помощью команды `podman machine start`. Podman поддерживает одновременный запуск только одной VM. Команда `start` запускает VM по умолчанию. Если у вас несколько VM и вы хотите запустить другую, укажите имя этой машины:

```
$ podman machine start
INFO[0000] waiting for clients...
INFO[0000] listening tcp://127.0.0.1:7777
INFO[0000] new connection from @ to /run/user/3267/podman/
➥ qemu_podman-machine-default.sock
Waiting for VM ...
macOShine "podman-machine-default" started successfully
```

Теперь вы готовы приступить к выполнению команд Podman на Linux-машине, на которой запущена служба Podman. Выполните команду `podman version`, чтобы убедиться, что клиент и сервер настроены правильно. Если это не так, то команды Podman должны подсказать вам, как настроить систему:

```
$ podman version
Client:
Version: 4.1.0
API Version: 4.1.0
Go Version: go1.18.1
Built: Thu May 5 16:07:47 2022
OS/Arch: darwin/arm64
Server:
Version: 4.1.0
API Version: 4.1.0
Go Version: go1.18
Built: Fri May 6 12:16:38 2022
OS/Arch: linux/arm64
```

Вы можете использовать изученные в предыдущих главах команды Podman непосредственно в macOS. После завершения работы с контейнерами в виртуальной машине ее, вероятно, следует выключить для экономии ресурсов.

ПРИМЕЧАНИЕ Podman поддерживается на машинах M1 с архитектурой arm64, а также на платформах x86. Команда `podman machine init` загружает VM соответствующей архитектуры, что позволяет вам создавать образы для этой архитектуры. Поддержка создания образов для других архитектур на момент написания книги находилась в стадии разработки.

E.1.4. Остановка виртуальной машины

Команда `podman machine stop` позволяет выключить все контейнеры внутри VM, а также саму VM:

```
$ podman machine stop
```

Когда вам понадобится снова использовать контейнеры, запустите VM с помощью команды `podman machine start`.

ПРИМЕЧАНИЕ Все команды `podman machine` работают и в Linux и позволяют одновременно тестировать различные версии Podman. Podman в Linux — это полноценная команда. Поэтому для связи со службой Podman, запущенной в VM с помощью `podman machine`, необходимо использовать параметр `--remote`. На платформах, отличных от Linux, параметр `--remote` не требуется, поскольку клиент уже сконфигурирован в режиме `--remote`.

РЕЗЮМЕ

- Для Linux-контейнеров требуется ядро Linux, а значит, для запуска контейнеров в macOS необходима виртуальная машина под управлением Linux.
- Podman в macOS не запускает контейнеры локально в этой macOS. Команда Podman на самом деле взаимодействует со службой Podman, запущенной на Linux-машине.
- Команда `podman machine init` загружает и устанавливает на вашу платформу виртуальную машину Fedora CoreOS, на которой запущена служба Podman.
- Команда `podman machine init` также настраивает SSH-подключение, необходимое для связи удаленного клиента Podman с сервером Podman внутри VM.

F Podman в Windows

В ЭТОМ ПРИЛОЖЕНИИ

- ✓ Установка Podman в Windows
- ✓ Использование команды `podman machine init` для установки WSL 2 на базе Fedora для запуска Podman
- ✓ Использование командной строки podman в Windows для связи со службой Podman, запущенной в экземпляре WSL 2
- ✓ Запуск или остановка экземпляра WSL 2 с помощью команд `podman machine start/stop`

Podman — это инструмент для запуска Linux-контейнеров. Контейнеры Linux требуют наличия ядра Linux. Как бы мне ни хотелось убедить весь мир перейти на Linux Desktop, которым пользуюсь я, большинство пользователей работают в операционных системах macOS и Windows, — возможно, и вы тоже. Если вы используете Linux Desktop, это здорово!

Если вы не работаете в Windows, пропустите это приложение.

Раз вы не пропустили это приложение, предположу, что вы хотите создавать Linux-контейнеры без необходимости подключаться по SSH к Linux-машине.

Скорее всего, вы будете использовать штатные средства разработки программного обеспечения и вести разработку локально.

В Linux Podman может быть запущен как служба, позволяющая устанавливать удаленные соединения для запуска контейнеров. Тогда из другой системы команда podman `--remote` может быть использована для связи с этой удаленной службой для запуска контейнеров.

Кроме того, с помощью команды `podman system connection` можно настроить `podman --remote` для связи с удаленной Linux-машиной, на которой запущена служба Podman, по SSH, не указывая URL для каждой команды. Проблема только в том, что кто-то должен настроить на удаленной машине правильную версию службы Podman, а затем настроить SSH.

Понимая, что такой подход не является оптимальным для новых пользователей Podman в Windows, компания Podman добавила команду `podman machine`. Команда `podman machine` позволяет легко создавать среду Linux на базе WSL 2 с предустановленным и настроенным Podman и управлять ею. На самом деле команда Podman под Windows — это урезанная команда с поддержкой только `podman --remote`. В этом приложении вы узнаете, как установить Podman на Windows-машине, а затем использовать команды `podman machine` для установки, настройки экземпляра WSL и управления им.

F.1. ПЕРВЫЕ ШАГИ

Команда `podman machine` в Windows принимает все те же команды, что и в Linux и macOS, и ведет себя похоже. И все же несколько отличий есть, поскольку базовый бэкенд в Windows основан на Windows Subsystem for Linux (https://docs.microsoft.com/en-us/windows/wsl/), а не на виртуальной машине, как в других операционных системах.

WSL 2 предполагает использование гипервизора Windows Hyper-V, однако, в отличие от стандартного подхода в виде виртуальной машины, WSL 2 использует одну и ту же виртуальную машину и экземпляр ядра Linux для всех дистрибутивов Linux, установленных пользователем. Например, если создать два дистрибутива WSL 2 и запустить `dmesg` в каждом из них, можно получить один и тот же результат, поскольку в обоих случаях используется одно и то же ядро.

> **ПРИМЕЧАНИЕ** WSL 1 не работает с Podman. Необходимо обновить машину с Windows до версии ОС, поддерживающей WSL 2. Для систем x64 требуется версия Windows 1903 и выше, сборка 18362 и выше. Для систем arm64 требуется Windows версии 2004 и выше, сборка 19041 и выше.

Запуск Podman с WSL 2 обеспечивает эффективное разделение ресурсов между хостом и всеми запущенными экземплярами в обмен на меньшую изоляцию. Следует помнить, что команда `podman machine` использует одно и то же ядро с другими запущенными дистрибутивами, поэтому будьте осторожны при манипулировании любыми настройками уровня ядра (например, сетевыми интерфейсами и политикой netfilter) в любом дистрибутиве, поскольку это может повлиять на контейнеры, выполняемые Podman.

F.1.1. Предварительные условия

Для работы Podman под Windows требуется Windows 10 (сборка 19041 или более поздняя) или Windows 11. Поскольку WSL 2 использует гипервизор, на машине должны быть включены функции виртуализации (например, Intel VT-x или AMD-V). Кроме того, гипервизор должен поддерживать трансляцию адресов второго уровня (SLAT). И наконец, система должна иметь либо подключение к интернету, либо автономную копию всего программного обеспечения, которое будет загружаться командой `podman machine`.

> **ПРИМЕЧАНИЕ** Если в какой-то момент времени возникают ошибки 0x80070003 или 0x80370102 (или любая ошибка, указывающая на невозможность запуска VM), то скорее всего, виртуализация отключена. Проверьте настройки BIOS (или экземпляра WSL 2), чтобы убедиться, что VT-x/AMD-V/WSL 2 и SLAT включены.

Установка Windows Terminal (в отличие от стандартного CMD или PowerShell) необязательна, но настоятельно рекомендуется (в будущих версиях Windows 11 он будет включен по умолчанию). Помимо современных возможностей терминала, он предлагает прямую интеграцию с WSL и PowerShell, что упрощает переключение между средами. Установить его можно через магазин Windows или `winget`:

```
PS C:\Users\User> winget install Microsoft.WindowsTerminal
```

F.1.2. Установка Podman

Установка Podman не представляет собой ничего сложного. Перейдите на сайт Podman или в репозиторий Podman на GitHub и загрузите последнюю версию Podman MSI Windows Installer в разделе Releases (рис. F.1; https://github.com/containers/podman/releases).

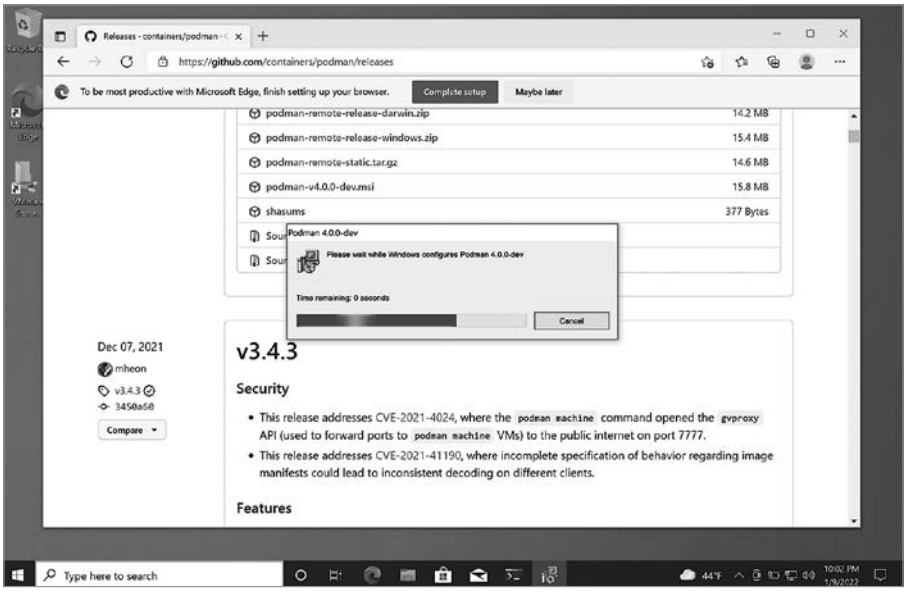


Рис. F.1. Загрузка и запуск программы установки Podman

После запуска программы установки откройте терминал (используйте команду `wt`, если вы установили Windows Terminal, как было рекомендовано) и выполните первую podman-команду (рис. F.2).

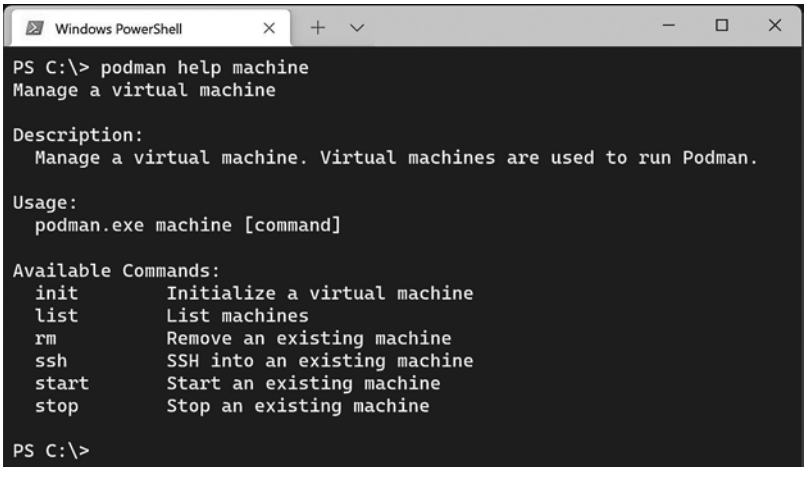


Рис. F.2. Команды Podman, выполняемые в терминале Windows

F.1.3. Автоматическая установка WSL

Если WSL не установлена в вашей Windows-системе, Podman сделает это за вас. Просто выполните команду `podman machine init` (как показано на рис. F.3) для создания первого экземпляра машины, после чего Podman запросит разрешение на установку WSL. Процесс установки WSL требует перезагрузки, но при этом возобновляется выполнение процесса создания машины (не забудьте подождать несколько минут, пока терминал перезапустится и выполнит установку). Если вы предпочитаете ручную установку, обратитесь к руководству по установке WSL: https://docs.microsoft.com/en-us/windows/wsl/install.

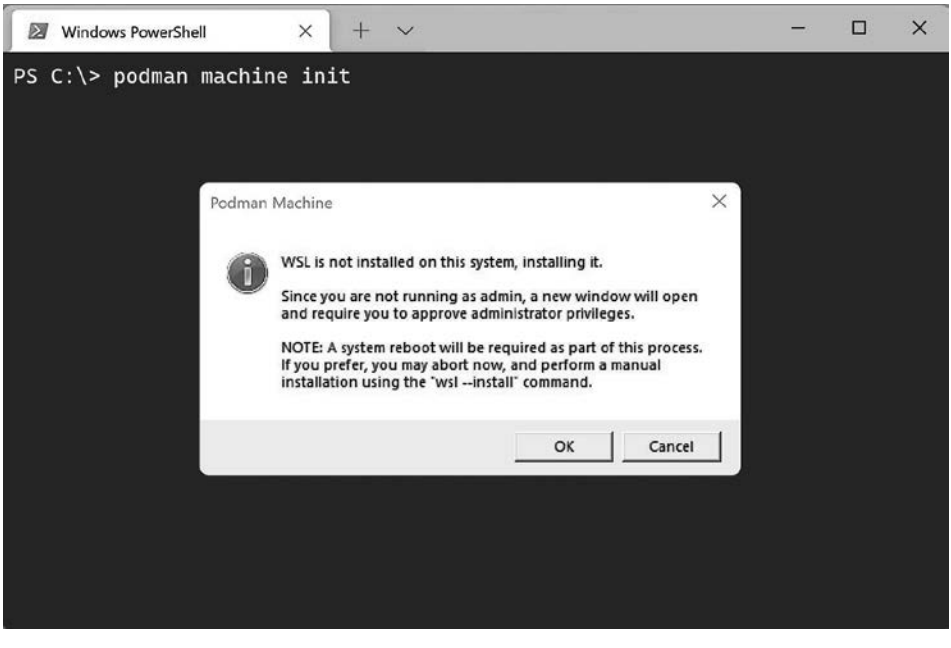


Рис. F.3. Команда podman machine init запускает установку WSL

F.2. ИСПОЛЬЗОВАНИЕ КОМАНДЫ PODMAN MACHINE

Настройка и использование окружения Linux упрощается за счет применения команд `podman machine`. Под Windows эти команды создают дистрибутив WSL 2 и управляют им, включая загрузку базового образа Linux с пакетами из

интернета и настройку всего необходимого. Дистрибутив WSL 2 предварительно настраивается с помощью службы Podman, а конфигурация SSH-соединения автоматически добавляется в хранилище `podman system connection`. В итоге вы получаете возможность легко запускать команды Podman на вашем компьютере с Windows, как если бы это была система Linux. В табл. F.1 приведены все команды `podman machine`, используемые для управления жизненным циклом окружения Linux на базе WSL 2.

После установки Podman (раздел F.1.2) первым шагом будет создание экземпляра машины WSL 2 в вашей системе. Для этого используется команда `podman machine init`, описанная в следующем разделе.

Таблица F.1. Команды podman machine

Команда	Описание
`init`	Инициализация нового экземпляра машины на базе WSL 2
`list`	Вывод списка машин WSL 2
`rm`	Удаление экземпляра машины WSL 2
`set`	Задание изменяемых настроек машины WSL
`ssh`	Подключение по SSH к экземпляру WSL 2. Это удобно для входа в машину WSL и выполнения штатных команд Podman. Некоторые команды Podman не поддерживаются удаленно, кроме того, вам может потребоваться изменить некоторые конфигурации внутри экземпляра WSL 2
`start`	Запуск экземпляра машины WSL 2
`stop`	Остановка экземпляра машины WSL 2. Если вы не используете контейнеры, вам может потребоваться остановка для экономии системных ресурсов

F.2.1. podman machine init

Как показано на рис. F.4, с помощью команды `podman machine init` можно автоматизировать установку Linux-окружения на базе WSL 2, в котором размещена служба Podman для запуска контейнеров. По умолчанию `podman machine init` загружает совместимый релиз Fedora для создания экземпляра WSL 2 (https://getfedora.org). Fedora хорошо интегрирована с Podman и является операционной системой, используемой большинством основных разработчиков Podman.

ПРИМЕЧАНИЕ В дополнение к базовому образу необходимо загрузить и установить ряд пакетов, что может занять несколько минут.

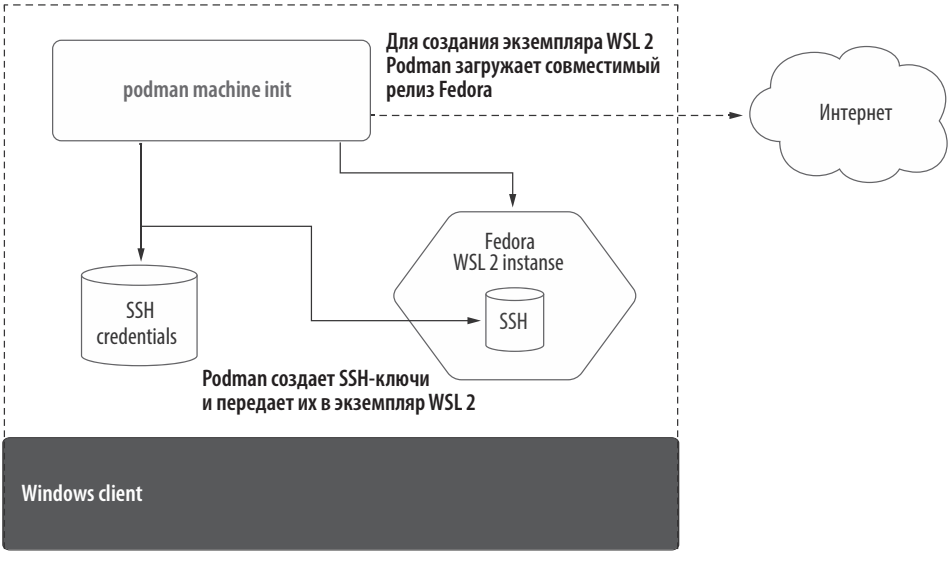


Рис. F.4. Команда podman machine init создает дистрибутив WSL 2 и настраивает SSH-соединения

Ниже показан сокращенный результат выполнения команды `podman machine init`:

```
PS C:\Users\User> podman machine init
Downloading VM image: fedora-35.20211125-x86_64.tar.xz: done
Extracting compressed file
Importing operating system into WSL (this may take 5+ minutes on a new WSL
➥ install)...
Installing packages (this will take a while)...
Fedora 35 - x86_64         5.5 MB/s       | 79 MB       00:14
Complete!
Configuring system...
Generating public/private ed25519 key pair.
Machine init complete
To start your machine run:
        podman machine start
```

В табл. F.2 описаны параметры команды init, позволяющие изменить настройки по умолчанию.

Таблица F.2. Параметры команды podman machine init

Параметр	Описание
`--cpus uint`	Не используется
`--disk-size uint`	Не используется

Параметр	Описание
`--image-path string`	В Windows этот параметр указывает на номер дистрибутива Fedora (например, 35). Как и в случае с Linux и Mac, можно указать произвольный URL или место с образом в файловой системе, но Podman ожидает, что это будет образ, полученный из дистрибутива Fedora
`--memory integer`	Не используется
`--rootful`	Определяет, должен данный экземпляр машины быть rootful или rootless

ПРИМЕЧАНИЕ Физические лимиты, указанные в табл. F.2 (например, процессор, память и диск), в настоящее время игнорируются в Windows, поскольку бэкенд Windows Subsystem for Linux (WSL) динамически изменяет размер и распределяет ресурсы между дистрибутивами. Если необходимо ограничить ресурсы, можно настроить эти ограничения в пользовательском файле .wslconfig. Однако они применяются глобально ко всем дистрибутивам WSL 2, поскольку используется одна и та же базовая VM.

F.2.2. Настройка SSH в podman machine

Команда `podman machine init` создает учетную запись в экземпляре WSL 2. По умолчанию пользователем в Fedora является user@localhost. Podman настраивает SSH на клиентской машине, новую учетную запись `user` и `root` в экземпляре WSL 2. Данная конфигурация SSH позволяет выполнять SSH-команды без пароля для учетных записей `user` и `root` с клиента. Команда `podman machine init` также настраивает информацию о системных соединениях Podman (раздел 9.5.4). База данных системных подключений настраивается как для пользователя rootful, так и для пользователя rootless в рамках экземпляра WSL 2. Если у вас нет существующих подключений, команда `podman machine init` создает и устанавливает по умолчанию подключение одного из rootless-пользователей к вашему экземпляру WSL 2.

Вы можете просмотреть все соединения с помощью команды `podman system connection list`. Соединение по умолчанию `podman-machine-default` является rootless:

```
PS C:\Users\User> podman system connection ls
Name URI Identity
➥ Default
podman-machine-default ssh://user@localhost:57051.. podman-machine-
➥ default true
podman-machine-default-root ssh://root@localhost:57051.. podman-machine-
➥ default false
```

Иногда для запуска контейнеров требуются привилегии root, и их нельзя запустить в rootless-режиме. Вы можете изменить подключение по умолчанию на rootful, переключив режим по умолчанию для созданного экземпляра машины. Измените режим по умолчанию на rootful с помощью команды `podman machine set`:

```
PS C:\Users\User> podman machine set --rootful
```

Снова посмотрите на подключения, чтобы убедиться, что по умолчанию теперь стоит `podman-machine-defaultroot`:

```
PS C:\Users\User> podman system connection ls
Name                         URI                       Identity
➥ Default
podman-machine-default       ssh://user@localhost:57051..
➥ podman-machine-default false
podman-machine-default-root  ssh://root@localhost:57051..
➥ podman-machine-default true
```

Теперь все команды Podman подключаются непосредственно к службе Podman, запущенной под учетной записью root. Снова измените подключение по умолчанию на соединение с пользователем без root-прав, используя команду `podman machine set`:

```
PS C:\Users\User> podman machine set --rootful=false
```

Если в этот момент попытаться запустить контейнер Podman, это не удастся, поскольку экземпляр машины фактически не запущен. Необходимо запустить машину.

F.2.3. Запуск экземпляра WSL 2

Команда `podman version` не срабатывает, так как экземпляр WSL 2 не запущен:

```
PS C:\Users\User> podman version
Cannot connect to Podman. Please verify your connection to the Linux system
using 'podman system connection list', or try 'podman machine init' and
'podman machine start' to manage a new Linux Linux VM
Error: unable to connect to Podman. failed to create sshClient: Connection
to bastion host (ssh://root@localhost:38243/run/podman/podman.sock)
failed.: dial tcp [::1]:38243: connect: connection refused
```

Как следует из сообщения об ошибке, виртуализированное Linux-окружение (экземпляр машины WSL 2) не запущено.

Для запуска одного экземпляра WSL 2 используется команда `podman machine start`. По умолчанию запускается стандартный экземпляр WSL 2:

podman-machine-default. Если у вас есть несколько экземпляров WSL 2 и вы хотите запустить другой, укажите имя машины дополнительно в команде podman machine start:

```
PS C:\Users\User> podman machine start
Starting machine "podman-machine-default"
This machine is currently configured in rootless mode. If your containers
require root permissions (e.g. ports < 1024), or if you run into compatibility
issues with non-podman clients, you can switch using the following command:
         podman machine set –rootful
API forwarding listening on: npipe:////./pipe/docker_engine
Docker API clients default to this address. You do not need to set
DOCKER_HOST.
Machine "podman-machine-default" started successfully
```

Теперь вы готовы приступить к выполнению команд Podman на хосте, который взаимодействует со службой Podman, которая, в свою очередь, запущена в экземпляре WSL 2. Выполните команду podman version, чтобы убедиться, что клиент и сервер настроены правильно. Если это не так, команды Podman подскажут, как настроить систему:

```
PS C:\Users\User> podman version
Client:      Podman Engine
Version:     4.0.0-dev
API Version: 4.0.0-dev
Go Version:  go1.17.1
Git Commit:  bac389043f268e632c45fed7b4e88bdefd2d95e6-dirty
Built:       Wed Feb 16 00:33:20 2022
OS/Arch:     windows/amd64
Server:      Podman Engine
Version:     4.0.1
API Version: 4.0.1
Go Version:  go1.16.14
Built:       Fri Feb 25 13:22:13 2022
OS/Arch:     linux/amd64
```

Теперь вы можете использовать изученные в предыдущих главах команды Podman непосредственно в Windows. Важно понимать, что использование Podman в Windows означает использование команды podman --remote для удаленного обращения к службе Podman в экземпляре WSL 2.

F.2.4. Использование команд podman machine

После запуска экземпляра машины вы можете выполнять команды Podman в командной строке PowerShell.

```
PS C:\Users\User> podman run ubi8-micro date
Thu Jan 6 05:09:59 UTC 2022
```

Остановка экземпляра WSL 2

После завершения использования контейнеров в вашей системе может потребоваться выключить экземпляр WSL 2 для экономии ресурсов системы. Используйте команду `podman machine stop`, чтобы выключить все контейнеры в экземпляре WSL 2, а также сам экземпляр WSL 2:

```
PS C:\Users\User> podman machine stop
```

При необходимости снова работать с контейнерами запустите экземпляр WSL 2 с помощью команды `podman machine start`.

> **ПРИМЕЧАНИЕ** Все команды `podman machine` работают и в Linux и позволяют одновременно тестировать различные версии Podman. Podman в Linux — это полноценная команда, поэтому для связи со службой Podman, запущенной в WSL 2 с помощью команды `podman machine`, необходимо использовать параметр `--remote`. На платформах, отличных от Linux, параметр `--remote` не требуется, поскольку клиент уже сконфигурирован в режиме `--remote`.

Вывод списка машин

Список доступных экземпляров машин выдает команда `podman machine ls`. Значения, возвращаемые этой командой в Windows, отражают текущее состояние, а не фиксированные ограничения ресурсов, как в macOS и Linux. Объем дискового пространства отражает объем, выделенный в данный момент для каждого экземпляра машины. Значения CPU показывают количество процессоров на Windows-хосте (если оно не ограничено WSL), которое повторяется для каждого экземпляра машины. Значения памяти также повторяются (с небольшими отклонениями) и отражают общий объем памяти, потребляемый ядром Linux для всех используемых дистрибутивов (поскольку память является общей). Другими словами, для определения общих ресурсов необходимо суммировать размеры дисков, но не памяти и процессора.

```
PS C:\Users\User> podman machine ls
NAME                    VM TYPE   CREATED       LAST UP   CPUS
➥ MEMORY   DISK SIZE
podman-machine-default wsl       3 days ago    Running   4
➥ 528.4MB  845.2MB
other                   wsl       4 minutes ago Running   4
➥ 524.5MB  778MB
```

Использование PODMAN в командной строке WSL

В дополнение к команде `podman machine ssh` можно получить доступ к `podman machine quest` машине с помощью командной строки WSL. Если вы используете Windows Terminal, то `podman machine quests` (имена с префиксом Podman) находятся в раскрывающемся списке со стрелками вниз. Другой способ — перейти

в командную строку WSL из любого окна PowerShell, используя команду wsl и указав имя дистрибутива. Например, экземпляр по умолчанию, созданный командой `podman machine init`, имеет имя `podman-machine-default`. Применяйте любой из этих подходов для управления гостевой машиной и выполнения команд Podman в полноценной оболочке Linux:

```
PS C:\Users\User> wsl -d podman-machine-default
[root@WIN10PRO /]# podman version
Client:        Podman Engine
Version:       4.0.1
API Version:   4.0.1
Go Version:    go1.16.14

Built:      Fri Feb 25 13:22:13 2022
OS/Arch:    linux/amd64
```

Обновление FEDORA

Поскольку реализация машины для Windows основана на Fedora, а не на Fedora CoreOS, исправления и улучшения не производятся автоматически. Они должны быть явно инициированы на гостевой машине с помощью команды управления пакетами Fedora `dnf`. Кроме того, переход на новую версию Fedora требует экспорта всех данных, которые необходимо сохранить, и использования команды `podman machine init` для создания второго экземпляра машины (или замены существующего экземпляра после выполнения команды `podman machine rm`).

> **ПРИМЕЧАНИЕ** В настоящее время с запуском Fedora CoreOS внутри WSL существуют сложности, поэтому было принято решение использовать по умолчанию Fedora. Если в будущем ситуация с поддержкой CoreOS в Windows изменится, `podman machine` перейдет на Fedora CoreOS.

Например, чтобы загрузить последние пакеты для текущей версии Fedora, запущенной на `podman machine quest`, выполните следующую команду:

```
PS C:\Users\User> podman machine ssh dnf upgrade -y
Warning: Permanently added '[localhost]:52581' (ED25519) to the list of
known hosts.
Last metadata expiration check: 1:18:35 ago on Wed Jan 5 21:13:15 2022.
Dependencies resolved.
...
Complete!
```

Расширенная остановка и перезапуск

Обычно для остановки и перезапуска Podman используются соответствующие команды `podman machine stop` и `podman machine start`. Остановка машины предпочтительна, так как при этом системные службы будут остановлены

корректно. Однако в некоторых случаях требуется принудительный перезапуск WSL, включая общее ядро Linux, остающееся активным даже после остановки машины. Для уничтожения всех процессов, связанных с дистрибутивом WSL, используйте команду `wsl --terminate <имя машины>`. Для остановки ядра Linux с завершением работы всех запущенных дистрибутивов используйте команду `wsl --shutdown`. После выполнения этих команд для возобновления работы экземпляра используется стандартная команда `podman machine start`:

```
PS C:\Users\User> wsl --shutdown
PS C:\Users\User> podman machine start
Starting machine...
Machine "podman-machine-default" started successfully
```

РЕЗЮМЕ

- Linux-контейнеры требуют наличия ядра Linux, поэтому для запуска контейнеров на платформе Mac или Windows нужна виртуальная машина под управлением Linux.
- Podman под Windows не запускает контейнеры локально на Windows. На самом деле команда Podman в этом случае — это `podman --remote`, взаимодействующая со службой, запущенной на Linux-машине, которая работает под управлением WSL 2.
- Команда `podman machine init` загружает и устанавливает на вашу платформу виртуальное Linux-окружение, в котором работает служба Podman.
- Команда `podman machine init` также настраивает SSH, позволяя удаленному клиенту Podman взаимодействовать с сервером Podman внутри экземпляра WSL 2.
- Podman в Windows с использованием WSL — это полноценная команда Podman. WSL выполняет команды Podman под управлением ядра Linux, хотя кажется, что он работает непосредственно на машине Windows.

Дэниэл Уолш

Podman в действии

Перевел с английского Д. Иванов

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*Е. Ваулина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 13.02.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 500. Заказ 00[redacted].

Никита Скабцов

KALI LINUX В ДЕЙСТВИИ. АУДИТ БЕЗОПАСНОСТИ ИНФОРМАЦИОННЫХ СИСТЕМ

2-е издание

В этой книге рассматриваются методы обхода систем безопасности сетевых сервисов и проникновения в открытые информационные системы. Информационная безопасность, как и многое в нашем мире, представляет собой медаль с двумя сторонами. С одной стороны, мы проводим аудит, ищем способы проникновения и даже применяем их на практике, а с другой — работаем над защитой. Тесты на проникновение являются частью нормального жизненного цикла любой ИТ-инфраструктуры, позволяя по-настоящему оценить возможные риски и выявить скрытые проблемы.

Может ли взлом быть законным? Конечно, может! Но только в двух случаях — когда вы взламываете принадлежащие вам ИС или когда вы взламываете сеть организации, с которой у вас заключено письменное соглашение о проведении аудита или тестов на проникновение. Мы надеемся, что вы будете использовать информацию из данной книги только в целях законного взлома ИС. Пожалуйста, помните о неотвратимости наказания — любые незаконные действия влекут за собой административную или уголовную ответственность.

Крис Риккомини, Дмитрий Рябой

README. СУРОВЫЕ РЕАЛИИ РАЗРАБОТЧИКОВ

Начинающим программистам требуется нечто большее, чем навыки программирования. Столкнувшись с реальной работой, вы моментально понимаете, что самым нужным вещам, имеющим критическое значение для карьеры, не обучают ни в университетах, ни на курсах. Книга «README. Суровые реалии разработчиков» призвана восполнить этот пробел.

Познакомьтесь с важнейшими практиками инжиниринга, которым обучают разработчиков в ведущих компаниях.

Вы узнаете о том, что вас ждет при устройстве на работу, затем познакомитесь с особенностями кода промышленного уровня, эффективным тестированием, рецензированием кода, непрерывной интеграцией и развертыванием, созданием проектной документации и лучшими практиками архитектуры ПО. В последних главах описываются навыки гибкого планирования и даются советы по построению карьеры.

Ключевые концепции и лучшие практики для начинающих разработчиков — то, чему вас не учили в университете!

КРОК

СОЗДАЕМ НАСТОЯЩЕЕ,
ИНТЕГРИРУЕМ БУДУЩЕЕ

croc.ru

КРОК — технологический партнер с комплексной экспертизой в области построения и развития инфраструктуры, внедрения информационных систем, разработки программных решений и сервисной поддержки.

Центры компетенций КРОК фокусируются на ключевых отраслевых кластерах — промышленность, финансовый сектор, розничные продажи, муниципальное управление, спорт и культура.

Ежегодно сотни проектов КРОК становятся системообразующими для экономики и социально-культурной сферы.
