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

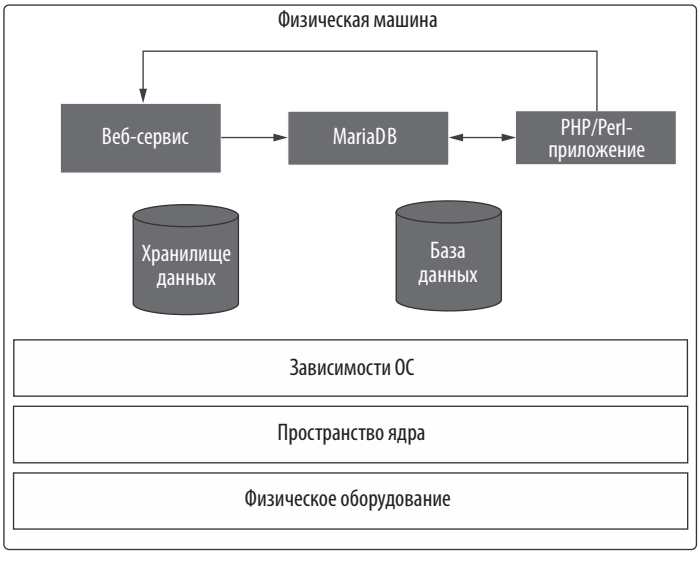


Рис. 1.6. Классический LAMP стек (Linux, Apache, MariaDB и приложение PHP/PERL), работающий на сервере

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.

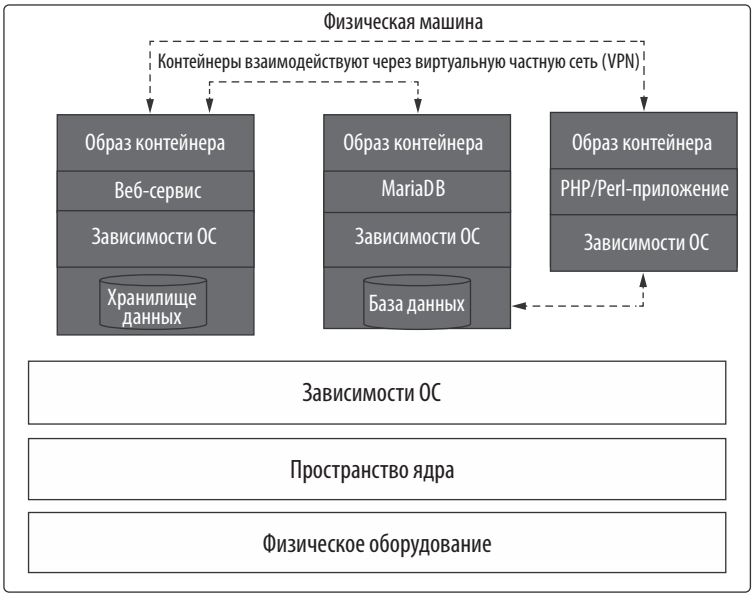


Рис. 1.7. Стек LAMP, упакованный в контейнеры микросервисов. Поскольку контейнеры взаимодействуют по сети, их можно легко перемещать на другие виртуальные машины, что значительно упрощает повторное использование

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

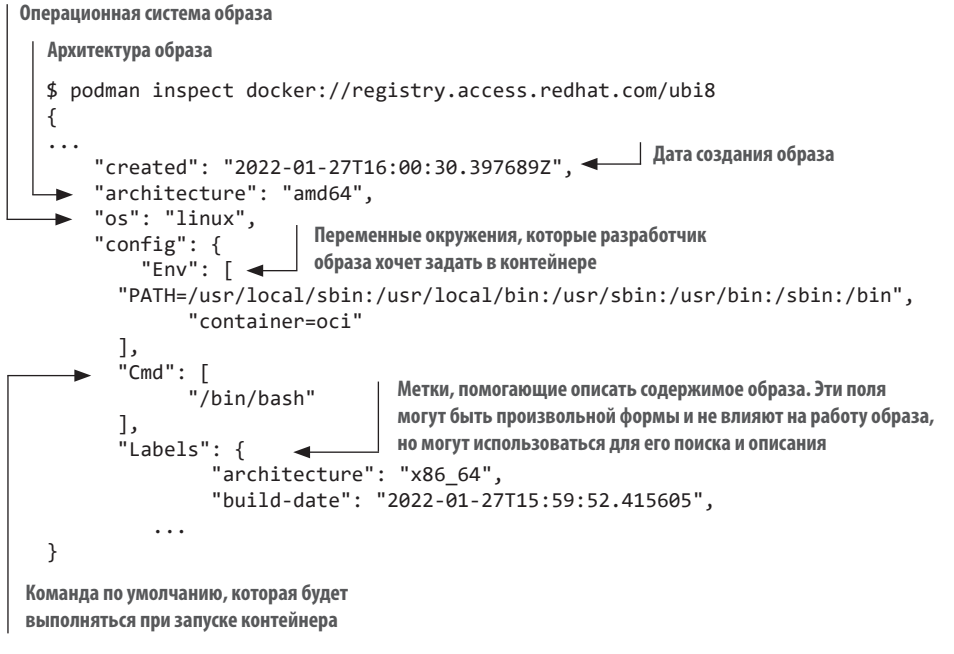
Образ контейнера состоит из трех компонентов:

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

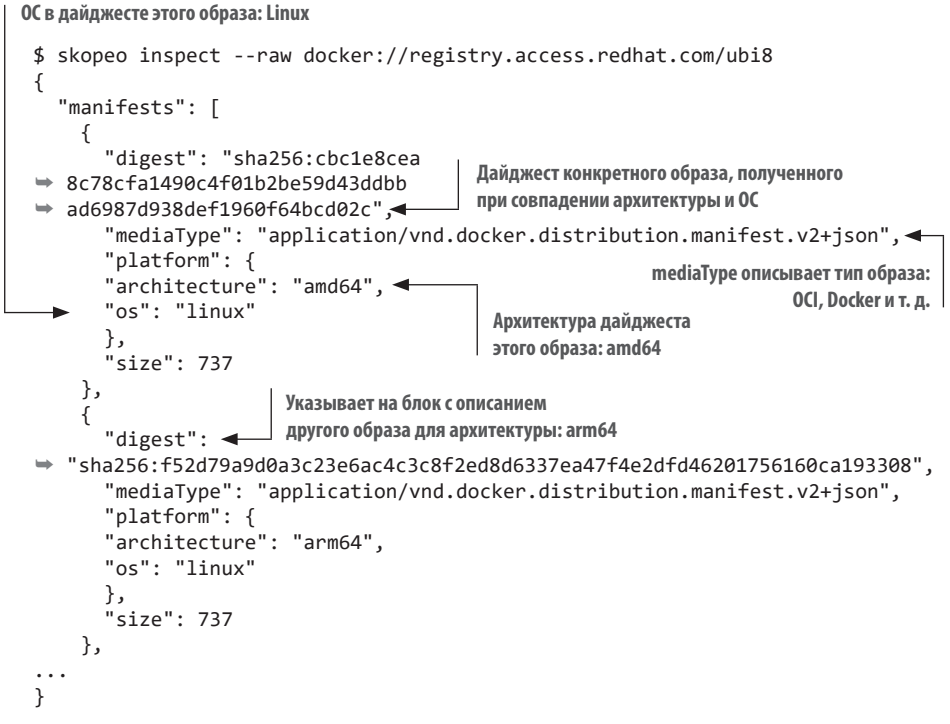
- JSON-файл, описывающий содержимое rootfs;
- еще один файл JSON, называемый manifest list, который связывает несколько образов вместе для поддержки различных архитектур.

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.

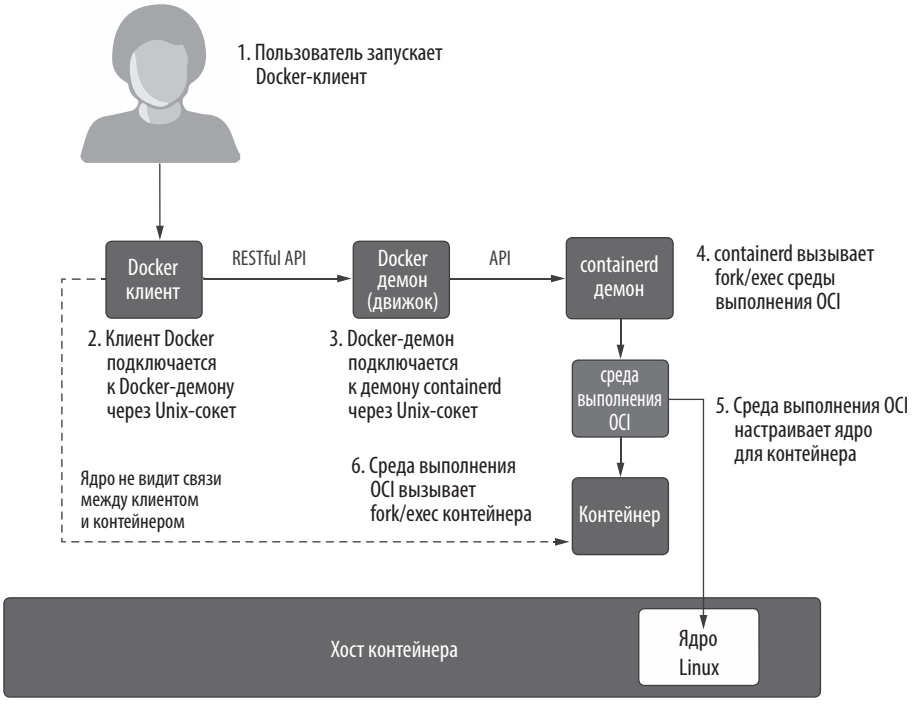


Рис. 1.8. Клиент-серверная архитектура Docker. Контейнер является прямым потомком containerd, а не Docker-клиента. Ядро не видит связи между клиентской программой и контейнером

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.