Red Hat Enterprise Linux 9

构建、运行和管理容器

在 Red Hat Enterprise Linux 9 上使用 Podman、Buildah 和 Skopeo

Last Updated: 2024-06-26

Red Hat Enterprise Linux 9 构建、运行和管理容器

在 Red Hat Enterprise Linux 9 上使用 Podman、Buildah 和 Skopeo

法律通告

摘要

Red Hat Enterprise Linux 9 提供多个命令行工具来使用容器镜像。您可以使用 Podman 管理 pod 和容器镜像。要构建、更新和管理容器镜像，您可以使用 Buildah。要复制并检查远程存储库中的镜像，您可以使用 Skopeo。

目录

对红帽文档提供反馈

我们感谢您对我们文档的反馈。让我们了解如何改进它。

通过 Jira 提交反馈（需要帐户）

1. 登录到 Jira 网站。
2. 在顶部导航栏中点 **Create**
3. 在 **Summary** 字段中输入描述性标题。
4. 在 **Description** 字段中输入您的改进建议。包括文档相关部分的链接。
5. 点对话框底部的 **Create**。

第 1 章 从容器开始

Linux 容器已逐渐成为一种关键的开源应用程序打包和交付技术，将轻量级应用程序隔离与基于镜像的部署方法的灵活性相结合。Red Hat Enterprise Linux 使用以下核心技术实施 Linux 容器，例如：

- 控制组（cgroups）用于资源管理
- 命名空间（namespace）用于进程隔离
- SELinux 用于安全性
- 安全多租户

这些技术降低了安全漏洞的可能性，并为您提供了生成和运行企业级容器的环境。

Red Hat OpenShift 提供了强大的命令行和 Web UI 工具，用于以称为 pod 的单元形式构建、管理和运行容器。红帽允许您在 OpenShift 之外构建和管理单个容器和容器镜像。本指南描述了为执行在 RHEL 系统上直接运行这些任务所提供的工具。

与其他容器工具实现不同，这里描述的工具不以单一的 Docker 容器引擎和 **docker** 命令为中心。相反，红帽提供了一组命令行工具，无需容器引擎即可操作。它们是：

- **podman** - 用于直接管理 pod 和容器镜像（**run**、**stop**、**start**、**ps**、**attach** 和**exec**，等等）
- **buildah** - 用于构建、推送和签名容器镜像
- **skopeo** - 用于复制、检查、删除和签名镜像
- **runc** - 为 podman 和 buildah 提供容器运行和构建功能
- **crun** - 可选运行时，可以配置，并为 rootless 容器提供更大的灵活性、控制和安全性

由于这些工具与开放容器项目(OCI)兼容，因此它们可用于管理由 Docker 和其他兼容 OCI 的容器引擎生成和管理的相同的 Linux 容器。然而，它们特别适用于直接在 Red Hat Enterprise Linux 中运行在单节点用例。

如需多节点容器平台，请参阅 OpenShift 和 使用 CRI-O 容器引擎 以了解详细信息。

1.1. PODMAN、BUILDAH 和 SKOPEO 的特点

Podman、Skopeo 和 Buildah 工具被开发来取代 Docker 命令功能。这种场景中的每个工具都是非常轻量级的，并专注于功能的子集。

Podman、Skopeo 和 Buildah 工具的主要优点包括：

- 以无根模式运行 - rootless 容器更安全，因为它们在运行时不需要添加任何特权
- 不需要守护进程 - 这些工具在空闲时对资源的要求要低得多，因为如果您没有运行容器，Podman 就不会运行。相反，Docker 有一个守护进程一直在运行
- 原生 **systemd** 集成 - Podman 允许您创建 **systemd** 单元文件，并将容器作为系统服务运行

Podman、Skopeo 和 Buildah 的特点包括：

- Podman、Buildah 和 CRI-O 容器引擎都使用相同的后端存储目录，**/var/lib/containers**，而不是默认使用 Docker 存储位置 **/var/lib/docker**。

- 虽然 Podman、Buildah 和 CRI-O 共享相同的存储目录，但它们不能相互交互。这些工具可以共享镜像。
- 要以编程方式与 Podman 进行交互，您可以使用 Podman v2.0 RESTful API，它可以在有根和无根的环境中工作。如需更多信息，请参阅 使用 container-tools API 章节。

其他资源

- 向 Buildah、Podman 和 Skopeo "问好"
- Docker 用户的 podman 和 Buildah
- Buildah ：构建 OCI 容器镜像的工具
- Podman：管理 OCI 容器和 pod 的工具
- Skopeo:复制和检查容器镜像的工具

1.2. 常见 PODMAN 命令

您可以使用以下基本命令，使用 **podman** 工具管理镜像、容器和容器资源。要显示所有 Podman 命令的完整列表，请使用 **podman -h**。

attach

附加到正在运行的容器。

commit

从更改的容器创建新镜像。

容器检查点

检查一个或多个正在运行的容器。

容器恢复

从检查点恢复一个或多个容器。

build

使用 Containerfile 指令构建镜像。

create

创建容器但不启动容器。

diff

检查容器文件系统的更改。

exec

在正在运行的容器中运行一个进程。

export

将容器的文件系统内容导出为一个 tar 存档。

help, h

显示命令的列表或某个命令的帮助。

healthcheck

运行容器健康检查。

history

显示指定镜像的历史记录。

images

列出本地存储中的镜像。

import

导入一个 tar 包以创建文件系统镜像。

info

显示系统信息。

inspect

显示容器或镜像的配置。

kill

向一个或多个正在运行的容器发送一个特定的信号。

kube generate

根据容器、Pod 或卷生成 Kubernetes YAML。

kube play

根据 Kubernetes YAML 创建容器、pod 和卷。

load

从存档加载一个镜像。

login

登录到容器注册中心。

logout

从容器注册中心注销。

logs

获取容器的日志。

mount

挂载一个工作容器的根文件系统。

pause

暂停一个或多个容器中的所有进程。

ps

列出容器。

port

列出容器的端口映射或特定映射。

pull

从注册中心拉取镜像。

push

将镜像推送到指定的目的地。

restart

重启一个或多个容器。

rm

从主机中删除一个或多个容器。如果要运行，添加 **-f**。

rmi

从本地存储中删除一个或多个镜像。

run

在新容器中运行命令。

save

将镜像保存到存档。

search

在注册中心中搜索镜像。

start

启动一个或多个容器。

stats

显示一个或多个容器的 CPU 的百分比、内存、网络 I/O、块 I/O 和 PID 。

stop

停止一个或多个容器。

tag

向本地镜像添加额外的名称。

top

显示容器的运行进程。

umount, unmount

卸载工作容器的根文件系统。

unpause

取消一个或多个容器中进程的暂停。

version

显示 podman 版本信息。

wait

阻止一个或多个容器。

其他资源

- Podman 基础知识备忘单
- 现在试用 5 podman 功能

1.3. 不使用 DOCKER 运行容器

红帽从 RHEL 9 中删除了 Docker 容器引擎和 docker 命令。

如果您仍然希望在 RHEL 中使用 Docker，可以从不同的上游项目获取 Docker，但其在 RHEL 9 中不支持。

- 您可以安装 **podman-docker** 软件包，每次运行 **docker** 命令时，它实际上是运行 **podman** 命令。
- Podman 还支持 Docker Socket API，因此 **podman-docker** 软件包还在 **/var/run/docker.sock** 和 **/var/run/podman/podman.sock** 之间建立了一个链接。因此，您可以继续使用 **docker-py** 和 **docker-compose** 工具运行 Docker API 命令，而无需 Docker 守护进程。Podman 将为请求提供服务。
- **podman** 命令和 **docker** 命令一样，可以 **Containerfile** 或 **Dockerfile** 构建容器镜像。可以在 **Containerfile** 和 **Dockerfile** 中使用的命令一样。

- **podman** 不支持的 **docker** 命令选项包括 network、node、plugin（**podman** 不支持插件）、rename（**podman** 使用 rm 和 create 来重命名容器）、secret、service、stack 和 swarm（**podman** 不支持 Docker Swarm）。容器和镜像选项用于运行直接在 **podman** 中使用的子命令。

其他资源

- Docker 用户的 podman 和 Buildah

1.4. 为容器选择 RHEL 架构

红帽为以下计算机架构提供容器镜像和容器相关的软件：

- AMD64 和 Intel 64（基础镜像和分层镜像；不支持 32 位构架）
- PowerPC 8 和 9 64 位（基本镜像和大部分层次镜像）
- 64 位 IBM Z（基本镜像和大多数层次镜像）
- ARM 64-bit（仅用于基础镜像）

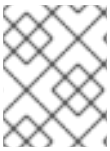
虽然还没有在所有构架中都首先支持所有红帽镜像，但几乎所有镜像现在都包含在所有列出的构架中。

其他资源

- 通用基础镜像(UBI)：镜像、存储库和软件包

1.5. 获取容器工具

此流程演示了如何安装 **container-tools** meta-package，其中包含 Podman、Buildah、Skopeo、CRIU、Udica 以及所有必需的库。



注意

RHEL 9 不提供 stable 流。要获得对 Podman 的稳定访问、Buildah、Skopeo 等，请使用 RHEL EUS 订阅。

流程

1. 安装 RHEL。
2. 注册 RHEL：输入您的用户名和密码。用户名和密码与您红帽客户门户网站的登录凭证相同：

```
# subscription-manager register
Registering to: subscription.rhsm.redhat.com:443/subscription
Username: <username>
Password: <password>
```

3. 订阅 RHEL。
   - 要自动订阅 RHEL：

```
# subscription-manager attach --auto
```

- 要按池 ID 订阅 RHEL：

```
# subscription-manager attach --pool <PoolID>
```

4. 安装 **container-tools** meta-package：

```
# dnf install container-tools
```

5. 可选：安装 **podman-docker** 软件包：

```
# dnf install podman-docker
```

podman-docker 软件包使用匹配的 Podman 命令替换 Docker 命令行界面和 **docker-api**。

1.6. 设置 ROOTLESS 容器

以超级用户特权（root 用户）的用户身份运行容器工具（如 Podman、Skopeo 或 Buildah）是确保容器对系统上任何可用功能具有全部权限的最佳方法。但是，从 Red Hat Enterprise Linux 8.1开始，提供名为"Rootless Containers"的功能，您可以以普通用户身份使用容器。

虽然容器引擎（如 Docker）可让您以普通（非 root）用户身份运行 Docker 命令，但执行这些请求的 Docker 守护进程还是以 root 用户身份运行。因此，普通用户可以通过其容器发出可能会损害系统的请求。通过设置 rootless 容器用户，系统管理员可以防止常规用户所做的潜在的损坏容器的活动，同时仍然允许这些用户在其自己的帐户下安全地运行大多数容器功能。

这个流程描述了如何设置您的系统，以非 root 用户（rootless）身份使用 Podman、Skopeo 和 Buildah 工具来与容器打交道。它还描述了您将遇到的一些限制，因为普通用户帐户对容器运行可能所需的所有操作系统功能没有全部的权限。

先决条件

- 您需要成为 root 用户来设置 RHEL 系统，以允许非 root 用户帐户使用容器工具。

流程

1. 安装 RHEL。
2. 安装 **podman** 软件包：

```
# dnf install podman -y
```

3. 创建新的用户帐户：

```
# useradd -c "Joe Jones" joe
# passwd joe
```

- 用户会自动配置为能够使用 rootless Podman。
- **useradd** 命令会在 **/etc/subuid** 和 **/etc/subgid** 文件中自动设置可访问用户和组 ID 的范围。
- 如果您手动更改 **/etc/subuid** 或 **/etc/subgid**，则必须运行 **podman system migrate** 命令，以允许应用新的更改。

4. 连接到用户：

```
$ ssh joe@server.example.com
```

注意

不要使用 **su** 或 **su -** 命令，因为这些命令不会设置正确的环境变量。

5. 拉取 **registry.access.redhat.com/ubi9/ubi** 容器镜像：

```
$ podman pull registry.access.redhat.com/ubi9/ubi
```

6. 运行名为 **myubi** 的容器,并显示 OS 版本：

```
$ podman run --rm --name=myubi registry.access.redhat.com/ubi9/ubi \
  cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="9 (Plow)"
```

其他资源

- 使用 Podman 的无根容器：基础知识
- **podman-system-migrate** man page

1.7. 升级到 ROOTLESS 容器

要从 Red Hat Enterprise Linux 7 升级到无根容器，您必须手动配置用户和组 ID。

从 Red Hat Enterprise Linux 7 升级到无根容器时需要考虑以下几点：

- 如果设置了多个 rootless 容器用户，请为每个用户使用唯一的范围。
- 使用 65536 UID 和 GID 来最大限度地与现有容器镜像兼容，但这个数字可以减小。
- 切勿使用 1000 以下的 UID 或 GID，或重新使用来自现有用户帐户的 UID 或 GID（默认情况下，从 1000 开始）。

先决条件

- 已创建用户帐户。

流程

- 运行 **usermod** 命令，为用户分配 UID 和 GID：

```
# usermod --add-subuids 200000-201000 --add-subgids 200000-201000 <username>
```

  - **usermod --add-subuid** 命令手动向用户帐户添加一组可访问的用户 ID。
  - **usermod --add-subgids** 命令手动向用户帐户添加一组可访问的用户 GID 和组 ID。

验证步骤

- 检查 UID 和 GID 是否已正确设置：

```
# grep <username> /etc/subuid /etc/subgid
 /etc/subuid:<username>:200000:1001
 /etc/subgid:<username>:200000:1001
```

1.8. 对 ROOTLESS 容器的特别考虑

以非 root 用户身份运行容器时，需要考虑以下事项：

- 对于 root 用户(**/var/lib/containers/storage**)和非 root 用户(**$HOME/.local/share/containers/storage**)，主机容器存储的路径是不同的。
- 运行无根容器的用户被授予在主机系统上作为用户和组群 ID 运行的特殊权限。但是，它们对主机上的操作系统没有 root 特权。
- 如果您手动更改 **/etc/subuid** 或 **/etc/subgid**，则必须运行 **podman system migrate** 命令，以允许应用新的更改。
- 如果您需要配置 rootless 容器环境，请在主目录(**$HOME/.config/containers**)中创建配置文件。配置文件包括 **storage.conf** （用于配置存储）和 **containers.conf** （用于各种容器设置）。您还可以创建 **registry.conf** 文件，以标识使用 Podman 进行拉取、搜索或运行镜像时可用的容器注册表。
- 有些系统功能在没有 root 特权的情况下无法更改。例如，您无法通过在容器内设置 **SYS_TIME** 功能并运行网络时间服务(**ntpd**)来更改系统时钟。您必须以 root 用户身份运行该容器，绕过 rootless 容器环境，并使用 root 用户的环境。例如：

```
# podman run -d --cap-add SYS_TIME ntpd
```

  请注意，这个示例允许 **ntpd** 为整个系统调整时间，而不只是在容器内调整。

- rootless 容器无法访问端口号小于 1024 的端口。在 rootless 容器命名空间中，它可以启动一个服务，该服务从容器中公开 httpd 服务的端口 80，但它不能在命名空间外访问：

```
$ podman run -d httpd
```

  但是，容器需要 root 权限，使用 root 用户的容器环境向主机系统公开该端口：

```
# podman run -d -p 80:80 httpd
```

- 工作站的管理员可以允许用户在编号低于 1024 的端口上公开服务，但他们应了解安全隐患。例如，普通用户可以在官方端口 80 上运行 Web 服务器，并让外部用户认为它是由管理员配置的。在工作站上进行测试是可以接受的，但在网络可访问的开发服务器上可能不是一个好主意，绝对不应该在生产服务器上这样做。要允许用户将端口绑定到 80 以下的端口，请运行以下命令：

```
# echo 80 > /proc/sys/net/ipv4/ip_unprivileged_port_start
```

其他资源

- Rootless Podman 的缺点

1.9. 使用模块进行高级 PODMAN 配置

您可以使用 Podman 模块来加载预定的一组配置。Podman 模块是 Tom 的 Obvious Minimal Language (TOML)格式的 **containers.conf** 文件。

这些模块位于以下目录或其子目录中：

- 对于无根用户： **$HOME/.config/containers/containers.conf.modules**
- 对于 root 用户： **/etc/containers/containers.conf.modules**，或 **/usr/share/containers/containers.conf.modules**

您可以使用 **podman --module <your_module_name>** 命令按需加载模块，以覆盖系统和用户配置文件。使用模块涉及以下事实：

- 您可以使用 **--module** 选项多次指定模块。
- 如果 **<your_module_name>** 是绝对路径，则配置文件将直接加载。
- 相对路径相对于前面提到的三个模块目录解析。
- **$HOME** 中的模块覆盖 **/etc/** 和 **/usr/share/** 目录中的模块。

其他资源

- 上游文档

1.10. 其他资源

- 容器术语实用简介

第 2 章 容器镜像的类型

容器镜像是一个二进制文件，其中包含运行单个容器的所有需求，以及描述其需求和功能的元数据。

容器镜像有两种类型：

- Red Hat Enterprise Linux Base Images (RHEL 基础镜像)
- Red Hat Universal Base Images (UBI 镜像)

两种类型的容器镜像都是从 Red Hat Enterprise Linux 的一部分构建的。通过使用这些容器，用户可以从出色的可靠性、安全性、性能和生命周期中受益。

两种容器镜像的主要区别在于 UBI 镜像允许您与其他人共享容器镜像。您可以使用 UBI 构建容器化的应用程序，将其推送到您选择的注册服务器，与他人轻松共享，甚至将其部署在非红帽平台上。UBI 镜像被设计成在容器中开发的云原生和 Web 应用程序用例的基础。

2.1. RHEL 容器镜像的一般特征

以下特征适用于 RHEL 基础镜像和 UBI 镜像。

通常，RHEL 容器镜像是：

- **Supported**:受红帽支持以用于容器化应用程序。它们包含与 Red Hat Enterprise Linux 中具有相同的安全性、经过测试并认证的软件包。
- **Cataloged** ：在 Red Hat Container Catalog 中列出，其中包含每个镜像的描述、技术详情和健康指数。
- **Updated**:提供明确的更新计划，以获取最新的软件，请参阅 Red Hat Container Image Updates 文章。
- **Tracked**:由红帽产品勘误表跟踪，以帮助了解每次更新中添加的更改。
- **Reusable**:在生产环境中，需要下载并缓存容器镜像。每个容器镜像都可以被将其作为基础包含的所有容器重复使用。

2.2. UBI 镜像的特点

UBI 镜像允许您与他人共享容器镜像。提供四个 UBI 镜像：micro、min、standard 和 init。预构建语言运行时镜像和 DNF 存储库可用于构建应用程序。

以下特点适用于 UBI 镜像：

- **从 RHEL 内容的子集构建**：红帽通用基础镜像由普通 Red Hat Enterprise Linux 内容的子集构建。
- **Redistributable**:UBI 镜像允许对红帽客户、合作伙伴、ISV 和其他人进行标准化。使用 UBI 镜像，您可以在可自由共享和部署的官方红帽软件的基础上构建容器镜像。
- **提供一组四个基础镜像** ：micro、min、standard 和 init。
- **提供一组预构建语言运行时容器镜像** ：基于 Application Streams 的运行时镜像为应用程序提供基础，这些应用程序可以受益于标准的、受支持的运行时，如 python、perl、php、dotnet、nodejs 和 ruby。

- **提供一组关联的 DNF 软件仓库**：DNF 软件仓库包括 RPM 软件包和更新，允许您添加应用程序依赖项并重建 UBI 容器镜像。
  - **ubi-9-baseos** 存储库包含容器中您可以包含的 RHEL 软件包的可重新分发的子集。
  - **ubi-9-appstream** 存储库包含应用程序流软件包，您可以将其添加到 UBI 镜像中，以帮助您对需要特定运行时的应用程序所使用的环境进行标准化。
  - **添加 UBI RPM**：您可以从预配置的 UBI 软件仓库将 RPM 软件包添加到 UBI 镜像中。如果您恰好处于断开连接的环境中，您必须将 UBI Content Delivery Network (**https://cdn-ubi.redhat.com**)放入 allowlist 来使用该功能。详情请查看 Connect to https://cdn-ubi.redhat.com。
- **Licensing**:您可以自由使用和重新分发 UBI 镜像，并遵循 Red Hat Universal Base Image End User Licensing Agreement。

注意

所有层次的镜像都基于 UBI 镜像。要根据哪个 UBI 镜像检查您的镜像，在 Red Hat Container Catalog 中显示 Containerfile，并确保 UBI 镜像包含所有需要的内容。

其他资源

- 红帽通用基础镜像简介
- 通用基础镜像(UBI)：镜像、存储库和软件包
- 您只需要了解 Red Hat Universal Base Image
- 常见问题 - 通用基础镜像

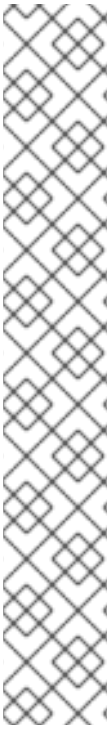
2.3. 了解 UBI 标准镜像

标准镜像(名为 **ubi**)专为在 RHEL 上运行的任何应用程序而设计。UBI 标准镜像的主要功能包括：

- **init 系统**：您需要管理 **systemd** 服务的 **systemd** 初始化系统的所有功能在标准基础镜像中提供。这些 init 系统可让您安装预配置的 RPM 软件包来自动启动服务，如 Web 服务器(**httpd**)或 FTP 服务器(**vsftpd**)。
- **dnf**：您可以访问免费 dnf 软件仓库来添加和更新软件。您可以使用 **dnf** 命令的标准集合（**dnf**、**dnf-config-manager**、**dnfdownloader** 等等）。
- **utilities**:工具包括 **tar**、**dmidecode**、**gzip**、**getfacl** 和其他 acl 命令、**dmsetup** 和其他设备映射器命令，以及此处未提及的其他工具。

2.4. 了解 UBI INIT 镜像

UBI init 镜像（名为 **ubi-init**，包含 **systemd** 初始化系统），有助于构建您要在其中运行 **systemd** 服务的镜像，如 Web 服务器或文件服务器。init 镜像内容小于您使用标准镜像获得的内容，但要比最小镜像中的内容要多。



注意

由于 **ubi9-init** 镜像构建在 **ubi9** 镜像基础之上，因此它们的内容基本相同。但是，有几个关键的区别：

- **ubi9-init**:
  - CMD 被设为 **/sbin/init**，以默认启动 **systemd** Init 服务
  - 包括 **ps** 和进程相关的命令（**procps-ng** 软件包）
  - 将 **SIGRTMIN+3** 设为 **StopSignal**，因为 **ubi9-init** 中的 **systemd** 会忽略正常信号而退出(**SIGTERM** 和 **SIGKILL**)，但如果它收到 **SIGRTMIN+3**，则会终止
- **ubi9** ：
  - CMD 设为 **/bin/bash**
  - 不包含 **ps** 和进程相关的命令（**procps-ng** 软件包）
  - 不要忽略正常信号退出（**SIGTERM** 和 **SIGKILL**）

2.5. 了解 UBI 最小镜像

UBI 最小镜像(名为 **ubi-minimal**) 提供最小的预安装的内容集和软件包管理器 (**microdnf**`)。因此，您可以在最小化镜像中包含的依赖项时使用 **Containerfile**。

UBI 最小镜像的主要功能包括：

- **Small size**:最小镜像在磁盘和 32M 上进行压缩时，最少为 92M。这比标准镜像小了多一半。
- **软件安装 (microdnf)**:不包含为使用软件存储库和 RPM 软件包而完全开发的 **dnf** 工具，最小镜像包括 **microdnf** 工具。**microdnf** 是 **dnf** 的缩小版，允许您启用和禁用存储库，删除和更新软件包，并在安装软件包后清除缓存。
- **Based on RHEL packaging**:最小镜像包含常规的 RHEL 软件 RPM 软件包，但删除了一些功能。最小镜像不包括初始化和服务管理系统，如 **systemd** 或 System V init、Python 运行时环境和一些 shell 工具。您可以依赖 RHEL 存储库来构建镜像，同时承担尽可能少的开销。
- **支持 microdnf 的模块**:与 **microdnf** 命令一起使用的模块可让您安装同一软件的多个版本。您可以使用 **microdnf module enable**、**microdnf module disable** 和 **microdnf module reset** 来分别启用、禁用和重置模块流。
  - 例如，要在 UBI 最小容器中启用 **nodejs:14** 模块流，请输入：

```
# microdnf module enable nodejs:14
Downloading metadata...
...
Enabling module streams:
   nodejs:14

Running transaction test...
```

红帽只支持最新版本的 UBI，且不支持点版本的停滞。如果您需要在特定的点版上进行 park，请参阅 延长更新支持。

2.6. 了解 UBI 微镜像

ubi-micro 可能是最小的 UBI 镜像，通过去掉软件包管理器及通常包含在容器镜像中的所有依赖项而得到。这可最小化基于 **ubi-micro** 镜像的容器镜像的攻击面，并适用于最小的应用程序，即使您对其他应用程序使用 UBI Standard、Minimal 或 Init。没有 Linux 发行包的容器镜像称为 Distroless 容器镜像。

第 3 章 使用容器注册中心

容器镜像注册中心是一个用于存储容器镜像和基于容器的应用程序工件的存储库或存储库集合。**/etc/containers/registries.conf** 文件是一个系统范围的配置文件，其包含容器镜像注册中心，可供各种容器工具（如 Podman、Buildah 和 Skopeo）使用。

如果提供给容器工具的容器镜像不是完全合格的，则容器工具会引用 **registry.conf** 文件。在 **registry.conf** 文件中，您可以为短名称指定别名，授予管理员完全控制从不完全合格的地方拉取的镜像。例如，**podman pull example.com/example_image** 命令将容器镜像从 **example.com** 注册表拉取到本地系统，如 **registry.conf** 文件中所指定的那样。

3.1. 容器注册中心

容器注册中心是用来存储容器镜像和基于容器的应用工件的存储库或存储库的集合。红帽提供的 registry 是：

- registry.redhat.io (需要身份验证)
- registry.access.redhat.com（不需要身份验证）
- registry.connect.redhat.com （保留 Red Hat Partner Connect 程序镜像）

要从远程注册中心（如红帽自己的容器注册中心）获取容器镜像，并将其添加到本地系统中，请使用 **podman pull** 命令：

```
# podman pull <registry>[:<port>]/[<namespace>/]<name>:<tag>
```

其中 **<registry>[:<port>]/[<namespace>/]<name>:<tag>** 是容器镜像的名称。

例如，**registry.redhat.io/ubi9/ubi** 容器镜像由以下标识：

- 注册中心服务器(**registry.redhat.io**)
- 命名空间(**ubi9**)
- 镜像名称(**ubi**)

如果同一镜像有多个版本，则请添加一个标签来明确指定镜像名称。默认情况下，Podman 使用 **:latest** 标签，如 **ubi9/ubi:latest**。

有些注册表也使用 *<namespace>* 来区分不同用户或机构所拥有的具有相同 *<name>* 的镜像。例如：

命名空间	示例（*<namespace>/<name>*）
机构	**redhat/kubernetes**，**google/kubernetes**
login（用户名）	**alice/application**，**bob/application**
role	**devel/database**、**test/database**、**prod/database**

有关转换到 registry.redhat.io 的详情，请参阅 Red Hat Container Registry Authentication 。在从 registry.redhat.io 拉取容器前，您需要使用 RHEL 订阅凭证进行身份验证。

3.2. 配置容器注册表

您可以使用 **podman info --format** 命令显示容器注册中心：

```
$ podman info -f json | jq '.registries["search"]'
[
  "registry.access.redhat.com",
  "registry.redhat.io",
  "docker.io"
]
```

注意

podman info 命令在 Podman 4.0.0 或更高版本中提供。

您可以编辑 **registries.conf** 配置文件中容器注册中心的列表。以 root 用户身份，编辑 **/etc/containers/registries.conf** 文件，来更改默认的系统范围的搜索设置。

以用户身份，创建 **$HOME/.config/containers/registries.conf** 文件来覆盖系统范围的设置。

```
unqualified-search-registries = ["registry.access.redhat.com", "registry.redhat.io", "docker.io"]
short-name-mode = "enforcing"
```

默认情况下，**podman pull** 和 **podman search** 命令以指定顺序在 **unqualified-search-registries** 列表中列出的注册表中搜索容器镜像。

配置本地容器注册中心

您可以配置没有 TLS 验证的本地容器注册表。关于如何禁用 TLS 验证，您有两个选项。首先，您可以在 Podman 中使用 **--tls-verify=false** 选项。其次，您可以在 **registry.conf** 文件中设置 **insecure=true** ：

```
[[registry]]
location="localhost:5000"
insecure=true
```

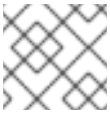
阻止注册表、命名空间或镜像

您可以定义不允许本地系统访问的注册表。您可以通过设置 **blocked=true** 来阻止特定的注册中心。

```
[[registry]]
location = "registry.example.org"
blocked = true
```

您也可以通过设置前缀 **prefix="registry.example.org/namespace"** 来阻止命名空间。例如，使用 **podman pull registry. example.org/example/image:latest** 命令拉取镜像会被阻止，因为指定的前缀匹配了。

```
[[registry]]
location = "registry.example.org"
prefix="registry.example.org/namespace"
blocked = true
```



注意

prefix 是可选的，默认值与 **location** 值相同。

您可以通过设置 **prefix="registry.example.org/namespace/image"** 来阻止特定的镜像。

```
[[registry]]
location = "registry.example.org"
prefix="registry.example.org/namespace/image"
blocked = true
```

镜像注册表

您可以设置一个注册中心镜像，以便在无法访问原始注册中心时访问。例如，您无法连接到互联网，因为您在高度敏感的环境中工作。您可以指定按照指定顺序联系的多个镜像。例如，当运行 **podman pull registry.example.com/myimage:latest** 命令时，首先会尝试 **mirror-1.com**，然后是 **mirror-2.com**。

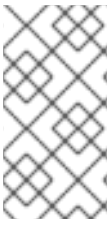
```
[[registry]]
location="registry.example.com"
[[registry.mirror]]
location="mirror-1.com"
[[registry.mirror]]
location="mirror-2.com"
```

其他资源

- 如何管理 Linux 容器注册表
- **podman-pull** 手册页
- **podman-info** 手册页

3.3. 搜索容器镜像

使用 **podman search** 命令，您可以在所选的容器注册表中搜索镜像。您还可以在 Red Hat Container Catalog 中搜索镜像。Red Hat Container Registry 包含镜像描述、内容、健康索引和其他信息。



注意

podman search 命令不是确定镜像是否存在的可靠方法。v1 和 v2 Docker 分发 API 的 **podman search** 行为是特定于每个注册中心的实现的。某些注册表可能根本就不支持搜索。在没有搜索词的情况下的搜索仅适用于实现 v2 API 的 注册表。**docker search** 命令也是如此。

要在 quay.io 注册中心搜索 **postgresql-10** 镜像，请按照以下步骤操作：

先决条件

- **container-tools** 元数据包已安装。
- 注册中心已配置。

流程

1. 向注册中心进行身份验证：

```
# podman login quay.io
```

2. 搜索镜像：

- 要搜索特定注册中心中的特定镜像，请输入：

```
# podman search quay.io/postgresql-10
INDEX       NAME                                      DESCRIPTION          STARS   OFFICIAL
AUTOMATED
redhat.io   registry.redhat.io/rhel8/postgresql-10         This container image ...  0
redhat.io   registry.redhat.io/rhscl/postgresql-10-rhel7   PostgreSQL is an  ...     0
```

- 另外，要显示特定注册中心提供的所有镜像，请输入：

```
# podman search quay.io/
```

- 要在所有注册表中搜索镜像名称，请输入：

```
# podman search postgresql-10
```

要显示完整的描述，请将 **--no-trunc** 选项传给命令。

其他资源

- **podman-search** 手册页

3.4. 从 REGISTRY 中拉取镜像

使用 **podman pull** 命令来将镜像提取到您的本地系统。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 登录到 registry.redhat.io registry:

```
$ podman login registry.redhat.io
Username: <username>
Password: <password>
Login Succeeded!
```

2. 拉取 registry.redhat.io/ubi9/ubi 容器镜像：

```
$ podman pull registry.redhat.io/ubi9/ubi
```

验证步骤

- 列出拉取到本地系统的所有镜像：

```
$ podman images
REPOSITORY                  TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi9/ubi latest  3269c37eae33  7 weeks ago  208 MB
```

其他资源

- **podman-pull** 手册页

3.5. 配置短名称别名

红帽建议始终使用完全限定名称拉取镜像。但是，通常按短名称拉取镜像。例如，您可以使用 **ubi9** 而不是 **registry.access.redhat.com/ubi9:latest**。

registries.conf 文件允许为短名称指定别名，使管理员能够完全控制从何处拉取镜像。别名在 **[aliases]** 表中指定，格式为 **"name" = "value"** 。您可以在 **/etc/containers/registries.conf.d** 目录中看到别名列表。红帽在此目录中提供了一组别名。例如，**podman pull ubi9** 直接解析为正确的镜像，即 **registry.access.redhat.com/ubi9:latest**。

例如：

```
unqualified-search-registries=["registry.fedoraproject.org", “quay.io"]

[aliases]
"fedora"="registry.fedoraproject.org/fedora"
```

简短名称模式为：

- **enforcing**:如果在镜像拉取过程中找不到匹配的别名，则 Podman 会提示用户选择一个非限定 registry。如果所选镜像拉取成功，Podman 将自动在 **$HOME/.cache/containers/short-name-aliases.conf** 文件（非 root 用户）或在 **/var/cache/containers/short-name-aliases.conf**（root 用户）中记录一个新的短名称别名。如果无法提示用户（例如，stdin 或 stdout 而不是 TTY），则 Podman 会失败。请注意，如果都指定了同一别名，**short-name-aliases.conf** 文件优先于 **registries.conf** 文件。

- **permissive**:与 enforcing 模式类似，但如果用户无法提示，Podman 不会失败。相反，Podman 会按照指定顺序搜索所有非限定 registry。请注意，没有记录别名。

- **disabled**:所有非限定 registry 都以给定顺序尝试，不记录别名。

注意

红帽建议使用完全限定镜像名称，包括注册中心、命名空间、镜像名称和标签。在使用短名称时，通常会存在欺骗风险。添加受信任的注册表，也就是不允许未知或匿名用户创建任意名称帐户的注册表。例如，用户希望从 **example.registry.com registry** 中拉取示例容器镜像。如果 **example.registry.com** 不是搜索列表中的第一个，则攻击者可以将不同的示例镜像放在搜索列表较前的注册中心中。用户会意外拉取并运行攻击者的镜像，而不是预期的内容。

其他资源

- Podman 中的容器镜像短名称

第 4 章 使用容器镜像

Podman 工具被设计来处理容器镜像。您可以使用此工具来拉取镜像、检查、打标签、保存、加载、重新分发和定义镜像签名。

4.1. 使用短名称别名拉取容器镜像

您可以使用安全的短名称将镜像提取到本地系统。以下流程描述了如何拉取一个 **fedora** 或 **nginx** 容器镜像。

先决条件

- **container-tools** 元数据包已安装。

流程

- 拉取容器镜像：
  - 拉取 **fedora** 镜像：

```
$ podman pull fedora
Resolved "fedora" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull registry.fedoraproject.org/fedora:latest…
...
Storing signatures
...
```

    找到别名，并且安全地拉取 **registry.fedoraproject.org/fedora** 镜像。**非限定搜索注册表** 列表不用于解析 **fedora** 镜像名称。

  - 拉取 **nginx** 镜像：

```
$ podman pull nginx
? Please select an image:
registry.access.redhat.com/nginx:latest
registry.redhat.io/nginx:latest
  ▸ docker.io/library/nginx:latest
✔ docker.io/library/nginx:latest
Trying to pull docker.io/library/nginx:latest…
...
Storing signatures
...
```

    如果没有找到匹配的别名，系统会提示您选择一个 **非限定搜索注册表**。如果成功拉取所选镜像，则会在本地记录新的短名称别名，否则会发生错误。

验证

- 列出拉取到本地系统的所有镜像：

```
$ podman images
REPOSITORY                                TAG     IMAGE ID      CREATED       SIZE
registry.fedoraproject.org/fedora         latest  28317703decd  12 days ago    184 MB
docker.io/library/nginx                   latest  08b152afcfae  13 days ago    137 MB
```

其他资源

- Podman 中的容器镜像短名称

4.2. 列出镜像

使用 **podman images** 命令列出本地存储中的镜像。

先决条件

- **container-tools** 元数据包已安装。
- 本地系统上提供了拉取的镜像。

流程

- 列出本地存储中的所有镜像：

```
$ podman images
REPOSITORY                           TAG     IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi9/ubi  latest  3269c37eae33  6 weeks ago  208 MB
```

其他资源

- **podman-images**手册页

4.3. 检查本地镜像

将镜像拉取到本地系统并运行后，您可以使用 **podman inspect** 命令来调查镜像。例如，使用它来了解镜像是做什么的，并检查镜像内有什么软件。**podman inspect** 命令显示有关按名称或 ID 标识的容器和镜像的信息。

先决条件

- **container-tools** 元数据包已安装。
- 本地系统上提供了拉取的镜像。

流程

- 检查 **registry.redhat.io/ubi9/ubi** 镜像：

```
$ podman inspect registry.redhat.io/ubi9/ubi
…
 "Cmd": [
        "/bin/bash"
    ],
    "Labels": {
        "architecture": "x86_64",
        "build-date": "2020-12-10T01:59:40.343735",
        "com.redhat.build-host": "cpt-1002.osbs.prod.upshift.rdu2.redhat.com",
        "com.redhat.component": "ubi9-container",
        "com.redhat.license_terms": "https://www.redhat.com/...,
```

```
    "description": "The Universal Base Image is ...
    }
...
```

"Cmd" 键指定要在容器内运行的默认命令。您可以通过指定一个命令作为 **podman run** 命令的参数来覆盖此命令。在使用 **podman run** 启动它时，如果没有其它参数，则此 ubi9/ubi 容器将执行 bash shell。如果设置了 **"Entrypoint"** 键，则使用其值而不是 **"Cmd"** 的值，**"Cmd"** 的值被用作 Entrypoint 命令的参数。

其他资源

- **podman-inspect** 手册页

4.4. 检查远程镜像

在将镜像拉取到系统之前，请使用 **skopeo inspect** 命令显示远程容器注册中心中的镜像信息。

先决条件

- **container-tools** 元数据包已安装。

流程

- **container-tools** 元数据包已安装。
- 检查 **registry.redhat.io/ubi9/ubi-init** 镜像：

```
# skopeo inspect docker://registry.redhat.io/ubi9/ubi-init
{
    "Name": "registry.redhat.io/ubi9/ubi9-init",
    "Digest": "sha256:c6d1e50ab...",
    "RepoTags": [
        ...
        "latest"
    ],
   "Created": "2020-12-10T07:16:37.250312Z",
    "DockerVersion": "1.13.1",
    "Labels": {
        "architecture": "x86_64",
        "build-date": "2020-12-10T07:16:11.378348",
        "com.redhat.build-host": "cpt-1007.osbs.prod.upshift.rdu2.redhat.com",
        "com.redhat.component": "ubi9-init-container",
        "com.redhat.license_terms": "https://www.redhat.com/en/about/red-hat-end-user-
license-agreements#UBI",
        "description": "The Universal Base Image Init is designed to run an init system as PID 1
for running multi-services inside a container
        ...
    }
}
```

其他资源

- **skopeo-inspect** 手册页

4.5. 复制容器镜像

您可以使用 **skopeo copy** 命令将容器镜像从一个注册中心复制到另一个注册中心。例如，您可以使用外部注册表的镜像填充内部存储库，或者在两个不同的地方同步镜像注册表。

先决条件

- **container-tools** 元数据包已安装。

流程

- 将 **skopeo** 容器镜像从 **docker://quay.io** 复制到 **docker://registry.example.com** ：

```
$ skopeo copy docker://quay.io/skopeo/stable:latest
docker://registry.example.com/skopeo:latest
```

其他资源

- **skopeo-copy** 手册页

4.6. 将镜像层复制到本地目录中

您可以使用 **skopeo copy** 命令将容器镜像的层复制到本地目录中。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建 **/var/lib/images/nginx** 目录：

```
$ mkdir -p /var/lib/images/nginx
```

2. 将 **docker://docker.io/nginx:latest 镜像** 的层复制到新创建的目录中：

```
$ skopeo copy docker://docker.io/nginx:latest dir:/var/lib/images/nginx
```

验证

- 显示 **/var/lib/images/nginx** 目录的内容：

```
$ ls /var/lib/images/nginx
08b11a3d692c1a2e15ae840f2c15c18308dcb079aa5320e15d46b62015c0f6f3
...
4fcb23e29ba19bf305d0d4b35412625fea51e82292ec7312f9be724cb6e31ffd  manifest.json
version
```

其他资源

- **skopeo-copy** 手册页

4.7. 标记镜像

使用 **podman tag** 命令给本地镜像添加额外的名称。此额外名称可由几个部分组成：*<registryhost>/<username>/<name>:<tag>*。

先决条件

- **container-tools** 元数据包已安装。
- 本地系统上提供了拉取的镜像。

流程

1. 列出所有镜像：

```
$ podman images
REPOSITORY                  TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi9/ubi latest  3269c37eae33  7 weeks ago  208 MB
```

2. 使用以下选项之一将 **myubi** 名称分配给 **registry.redhat.io/ubi9/ubi** 镜像：

   - 镜像名称：

```
$ podman tag registry.redhat.io/ubi9/ubi myubi
```

   - 镜像 ID：

```
$ podman tag 3269c37eae33 myubi
```

   两个命令会给出同样的结果。

3. 列出所有镜像：

```
$ podman images
REPOSITORY                  TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi9/ubi latest  3269c37eae33  2 months ago  208 MB
localhost/myubi             latest  3269c37eae33  2 months ago  208 MB
```

   请注意，两个镜像的默认标签都是 **latest**。您可以看到所有镜像名称都被分配给单个镜像 ID 3269c37eae33。

4. 使用以下方法将 **9** 标签添加到 **registry.redhat.io/ubi9/ubi** 镜像：

   - 镜像名称：

```
$ podman tag registry.redhat.io/ubi9/ubi myubi:9
```

   - 镜像 ID：

```
$ podman tag 3269c37eae33 myubi:9
```

   两个命令会给出同样的结果。

5. 列出所有镜像：

```
$ podman images
REPOSITORY                    TAG     IMAGE ID      CREATED       SIZE
registry.redhat.io/ubi9/ubi   latest  3269c37eae33  2 months ago  208 MB
localhost/myubi               latest  3269c37eae33  2 months ago  208 MB
localhost/myubi               9       3269c37eae33  2 months ago  208 MB
```

请注意，两个镜像的默认标签都是 **latest**。您可以看到所有镜像名称都被分配给单个镜像 ID 3269c37eae33。

在标记 **registry.redhat.io/ubi9/ubi** 镜像后，您有三个选项来运行容器：

- 按 ID (**3269c37eae33**)
- 按名称(**localhost/myubi:latest**)
- 按名称 (**localhost/myubi:9**)

其他资源

- **podman-tag** man page

4.8. 保存并加载镜像

使用 **podman save** 命令将镜像保存到容器存档中。稍后您可以将其恢复到其他容器环境，或将其发送给其他人。您可以使用 **--format** 选项来指定归档格式。支持的格式有：

- **docker-archive**
- **oci-archive**
- **oci-dir** （带有 oci 清单类型的目录）
- **docker-dir** （带有 v2s2 清单类型的目录）

默认格式为 **docker-dir** 格式。

使用 **podman load** 命令将容器镜像存档中的镜像加载到容器存储中。

先决条件

- **container-tools** 元数据包已安装。
- 本地系统上提供了拉取的镜像。

流程

1. 将 **registry.redhat.io/rhel9/rsyslog** 镜像保存为 tarball：
   - 在默认的 **docker-dir** 格式下：

     ```
     $ podman save -o myrsyslog.tar registry.redhat.io/rhel9/rsyslog:latest
     ```

   - 在 **oci-archive** 格式下，使用 **--format** 选项：

```
$ podman save -o myrsyslog-oci.tar --format=oci-archive
registry.redhat.io/rhel9/rsyslog
```

myrsyslog.tar 和 **myrsyslog-oci.tar** 存档存储在您的当前目录中。接下来的步骤使用 **myrsyslog.tar** tar 包来执行。

2. 检查 **myrsyslog.tar** 的文件类型：

```
$ file myrsyslog.tar
myrsyslog.tar: POSIX tar archive
```

3. 从 **myrsyslog.tar** 中加载 **registry.redhat.io/rhel9/rsyslog:latest** 镜像：

```
$ podman load -i myrsyslog.tar
...
Loaded image(s): registry.redhat.io/rhel9/rsyslog:latest
```

其他资源

- **podman-save** man page

4.9. 重新分发 UBI 镜像

使用 **podman push** 命令将 UBI 镜像推送到您自己的或第三方注册中心，并将其与他人共享。您可以根据情况，从 UBI dnf 软件仓库升级或添加到该镜像。

先决条件

- **container-tools** 元数据包已安装。
- 本地系统上提供了拉取的镜像。

流程

1. 可选：在 **ubi** 镜像中添加额外名称：

```
# podman tag registry.redhat.io/ubi9/ubi registry.example.com:5000/ubi9/ubi
```

2. 将 **registry.example.com:5000/ubi9/ubi** 镜像从本地存储推送到注册中心:

```
# podman push registry.example.com:5000/ubi9/ubi
```

重要

虽然对这些镜像的使用方式的限制不多，但对于如何引用这些镜像会有一些限制。例如，除非通过 Red Hat Partner Connect Program （使用 Red Hat Container Certification 或 Red Hat OpenShift Operator 认证）获取这些镜像，否则它们不能被称为红帽认证的镜像或获得红帽的支持。

4.10. 删除镜像

使用 **podman rmi** 命令来移除本地存储的容器镜像。您可以通过其 ID 或名称来删除镜像。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 列出本地系统上的所有镜像：

```
$ podman images
REPOSITORY                          TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/rhel8/rsyslog    latest  4b32d14201de  7 weeks ago  228 MB
registry.redhat.io/ubi8/ubi         latest  3269c37eae33  7 weeks ago  208 MB
localhost/myubi                     X.Y     3269c37eae33  7 weeks ago  208 MB
```

2. 列出所有容器：

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND         CREATED        STATUS
PORTS   NAMES
7ccd6001166e  registry.redhat.io/rhel8/rsyslog:latest  /bin/rsyslog.sh  6 seconds ago  Up 5
seconds ago          mysyslog
```

要删除 **registry.redhat.io/rhel8/rsyslog** 镜像，您必须使用 **podman stop** 命令停止运行此镜像的所有容器。您可以通过其 ID 或名称来停止容器。

3. 停止 **mysyslog** 容器：

```
$ podman stop mysyslog
7ccd6001166e9720c47fbeb077e0afd0bb635e74a1b0ede3fd34d09eaf5a52e9
```

4. 删除 **registry.redhat.io/rhel8/rsyslog** 镜像：

```
$ podman rmi registry.redhat.io/rhel8/rsyslog
```

- 要删除多个镜像：

```
$ podman rmi registry.redhat.io/rhel8/rsyslog registry.redhat.io/ubi8/ubi
```

- 要从您的系统中删除所有镜像：

```
$ podman rmi -a
```

- 要删除与其有多个名称（标记）关联的镜像，请添加 **-f** 选项来删除它们：

```
$ podman rmi -f 1de7d7b3f531
1de7d7b3f531...
```

其他资源

- **podman-rmi** man page

第 5 章 操作容器

容器代表从解压缩的容器镜像中的文件创建的一个正在运行中或已停止的进程。您可以使用 Podman 工具来操作容器。

5.1. PODMAN RUN 命令

podman run 命令在基于容器镜像的新容器中运行一个进程。如果容器镜像尚未加载，则在从该镜像启动容器前，**podman run** 会以与 **podman pull *image*** 相同的方式从存储库中拉取镜像和所有镜像的依赖项。容器进程具有自己的文件系统、自己的网络，以及它自己的隔离进程树。

podman run 命令的格式如下：

```
podman run [options] image [command [arg ...]]
```

基本选项为：

- **--detach(-d)** ：在后台运行容器，并打印新容器 ID。
- **--attach(-a)**:在前台模式运行容器。
- **--name(-n)** ：为容器分配一个名称。如果没有使用 **--name** 为容器分配名称，则它会生成一个随机字符串名称。这适用于后台和前台容器。
- **--rm**:在容器退出时自动移除容器。请注意，当容器无法成功创建或启动时，不能删除容器。
- **--tty(-t)** ：将伪终端分配给容器的标准输入信息。
- **--interactive(-i)** ：对于交互式进程，请使用 **-i** 和 **-t** 为容器进程分配终端。**-i -t** 通常写为 **-it**。

5.2. 在主机的容器中运行命令

使用 **podman run** 命令显示容器的操作系统的类型。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 使用 **cat /etc/os-release** 命令显示基于 **registry.access.redhat.com/ubi9/ubi** 容器镜像的容器类型：

```
$ podman run --rm registry.access.redhat.com/ubi9/ubi cat /etc/os-release
NAME="Red Hat Enterprise Linux"
...
ID="rhel"
...
HOME_URL="https://www.redhat.com/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"

REDHAT_BUGZILLA_PRODUCT=" Red Hat Enterprise Linux 9"
...
```

2. 可选：列出所有容器。

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

由于 **--rm** 选项，您应该看不到任何容器。容器已被删除。

其他资源

- **podman-run** 手册页

5.3. 在容器内运行命令

使用 **podman run** 命令以交互方式运行容器。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 根据 **registry.redhat.io/ubi9/ubi** 镜像，运行名为 **myubi** 的容器：

```
$ podman run --name=myubi -it registry.access.redhat.com/ubi9/ubi /bin/bash
[root@6ccffd0f6421 /]#
```

   - **-i** 选项创建一个交互式会话。如果不使用 **-t** 选项，shell 将保持打开状态，但您无法对 shell 输入任何东西。
   - **-t** 选项打开一个终端会话。如果不使用 **-i** 选项，shell 会打开，然后退出。

2. 安装 **procps-ng** 软件包，其包含一组系统工具（如 **ps**、**top** 和 **uptime**，等等）：

```
[root@6ccffd0f6421 /]# dnf install procps-ng
```

3. 使用 **ps -ef** 命令列出当前的进程：

```
# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 12:55 pts/0    00:00:00 /bin/bash
root          31       1  0 13:07 pts/0    00:00:00 ps -ef
```

4. 输入 **exit** 退出容器并返回到主机：

```
# exit
```

5. 可选：列出所有容器：

```
$ podman ps
CONTAINER ID  IMAGE                            COMMAND    CREATED         STATUS
PORTS   NAMES
1984555a2c27  registry.redhat.io/ubi9/ubi:latest  /bin/bash  21 minutes ago  Exited (0) 21
minutes ago          myubi
```

您可以看到容器处于 Exited 状态。

其他资源

- **podman-run** 手册页

5.4. 列出容器

使用 **podman ps** 命令列出系统上正在运行的容器。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 根据 **registry.redhat.io/rhel9/rsyslog** 镜像运行容器：

```
$ podman run -d registry.redhat.io/rhel8/rsyslog
```

2. 列出所有容器：
   - 要列出所有正在运行的容器：

```
$ podman ps
CONTAINER ID IMAGE              COMMAND        CREATED       STATUS
PORTS NAMES
74b1da000a11 rhel9/rsyslog /bin/rsyslog.sh 2 minutes ago Up About a minute
musing_brown
```

   - 要列出所有运行或停止的容器：

```
$ podman ps -a
CONTAINER ID IMAGE         COMMAND    CREATED    STATUS              PORTS
NAMES     IS INFRA
d65aecc325a4 ubi9/ubi      /bin/bash  3 secs ago Exited (0) 5 secs ago peaceful_hopper
false
74b1da000a11 rhel9/rsyslog rsyslog.sh 2 mins ago Up About a minute     musing_brown
false
```

如果有容器没有运行，但没有被删除（**--rm** 选项），则容器存在并且可以重新启动。

其他资源

- **podman-ps** man page

5.5. 启动容器

如果您运行容器，然后停止它，且未将其删除，则容器会存储在本地系统上，准备再次运行。您可以使用 **podman start** 命令来重新运行容器。您可以根据其容器 ID 或名称来指定容器。

先决条件

- **container-tools** 元数据包已安装。
- 至少一个容器已经停止。

流程

1. 启动 **myubi** 容器：
   - 在非互动模式中：

     ```
     $ podman start myubi
     ```

     另外，您可以使用 **podman start 1984555a2c27**。
   - 在交互模式中，使用 **-a** (**--attach**)和 **-i** (**--interactive**)选项来使用容器 bash shell：

     ```
     $ podman start -a -i myubi
     ```

     或者，您可以使用 **podman start -a -i 1984555a2c27**。
2. 输入 **exit** 退出容器并返回到主机：

   ```
   [root@6ccffd0f6421 /]# exit
   ```

其他资源

- **podman-start** man page

5.6. 检查主机的容器

使用 **podman inspect** 命令，来检查 JSON 格式的现有容器的元数据。您可以根据其容器 ID 或名称来指定容器。

先决条件

- **container-tools** 元数据包已安装。

流程

- 检查 ID 64ad95327c74 定义的容器：
  - 要获取所有元数据：

    ```
    $ podman inspect 64ad95327c74
    [
        {
            "Id":
    "64ad95327c740ad9de468d551c50b6d906344027a0e645927256cd061049f681",
            "Created": "2021-03-02T11:23:54.591685515+01:00",
            "Path": "/bin/rsyslog.sh",
            "Args": [
                "/bin/rsyslog.sh"
            ],
            "State": {
    ```

```
    "OciVersion": "1.0.2-dev",
    "Status": "running",
    ...
```

- 要从 JSON 文件中获取特定的内容，如 **StartedAt** 时间戳：

```
$ podman inspect --format='{{.State.StartedAt}}' 64ad95327c74
2021-03-02 11:23:54.945071961 +0100 CET
```

这些信息保存在层次结构中。要查看容器 **StartedAt** 时间戳（**StartedAt** 在 **State**下），请使用 **--format** 选项以及容器 ID 或名称。

您想要检查的其他内容示例包括：

- **.Path** 来查看容器运行的命令
- **.Args** 命令的参数
- **.Config.ExposedPorts** 容器公开的 TCP 或 UDP 端口
- **.State.Pid** 来查看容器的进程 ID
- **.HostConfig.PortBindings** 从容器到主机的端口映射

其他资源

- **podman-inspect** 手册页

5.7. 将 LOCALHOST 上的目录挂载到容器

您可以通过在容器内挂载主机 **/dev/log** 设备，使容器中的日志消息对主机系统可用。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 运行名为 **log_test** 的容器，并在容器内挂载主机 **/dev/log** 设备：

```
# podman run --name="log_test" -v /dev/log:/dev/log --rm \
  registry.redhat.io/ubi9/ubi logger "Testing logging to the host"
```

2. 使用 **journalctl** 工具来显示日志：

```
# journalctl -b | grep Testing
Dec 09 16:55:00 localhost.localdomain root[14634]: Testing logging to the host
```

--rm 选项会在容器退出时删除容器。

其他资源

- **podman-run** 手册页

5.8. 挂载容器文件系统

使用 **podman mount** 命令将正常工作的容器根文件系统挂载到主机可访问的位置。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 运行名为 **mysyslog** 的容器：

```
# podman run -d --name=mysyslog registry.redhat.io/rhel9/rsyslog
```

2. 可选：列出所有容器：

```
# podman ps -a
CONTAINER ID  IMAGE                                COMMAND         CREATED        STATUS
PORTS   NAMES
c56ef6a256f8  registry.redhat.io/rhel9/rsyslog:latest  /bin/rsyslog.sh  20 minutes ago  Up 20
minutes ago                mysyslog
```

3. 挂载 **mysyslog** 容器：

```
# podman mount mysyslog
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797
d7be46750894719/merged
```

4. 使用 **ls** 命令显示挂载点的内容：

```
# ls
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/merged
bin  boot  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv  sys
tmp  usr  var
```

5. 显示 OS 版本：

```
# cat
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/merged/etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="9 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
...
```

其他资源

- **podman-mount** man page

5.9. 将服务作为使用静态 IP 的守护进程运行

以下示例将 **rsyslog** 服务作为守护进程在后台运行。**--ip** 选项将容器网络接口设为特定的 IP 地址（如 10.88.0.44）。之后，您可以运行 **podman inspect** 命令来检查是否正确设置了 IP 地址。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 将容器网络接口设为 IP 地址 10.88.0.44：

```
# podman run -d --ip=10.88.0.44 registry.access.redhat.com/rhel9/rsyslog
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

2. 检查是否正确设置了 IP 地址：

```
# podman inspect efde5f0a8c723 | grep 10.88.0.44
"IPAddress": "10.88.0.44",
```

其他资源

- **podman-inspect** 手册页
- **podman-run** 手册页

5.10. 在运行中的容器中执行命令

使用 **podman exec** 命令，来在正在运行的容器中执行命令并调查该容器。使用 **podman exec** 命令而不是 **podman run** 命令的原因是，您可以在不中断容器活动的情况下调查正在运行的容器。

先决条件

- **container-tools** 元数据包已安装。
- 容器正在运行。

流程

1. 在 **myrsyslog** 容器中执行 **rpm -qa** 命令来列出所有已安装的软件包：

```
$ podman exec -it myrsyslog rpm -qa
tzdata-2020d-1.el8.noarch
python3-pip-wheel-9.0.3-18.el8.noarch
redhat-release-8.3-1.0.el8.x86_64
filesystem-3.8-3.el8.x86_64
...
```

2. 在 **myrsyslog** 容器中执行 **/bin/bash** 命令：

```
$ podman exec -it myrsyslog /bin/bash
```

3. 安装 **procps-ng** 软件包，其包含一组系统工具（如 **ps**、**top** 和 **uptime**，等等）：

```
# dnf install procps-ng
```

4. 检查容器：

- 要列出系统上的每个进程：

```
# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 10:23 ?        00:00:01 /usr/sbin/rsyslogd -n
root           8       0  0 11:07 pts/0    00:00:00 /bin/bash
root          47       8  0 11:13 pts/0    00:00:00 ps -ef
```

- 要显示文件系统磁盘空间使用情况：

```
# df -h
Filesystem      Size  Used Avail Use% Mounted on
fuse-overlayfs   27G  7.1G   20G  27% /
tmpfs            64M     0   64M   0% /dev
tmpfs           269M  936K  268M   1% /etc/hosts
shm              63M     0   63M   0% /dev/shm
...
```

- 要显示系统信息：

```
# uname -r
4.18.0-240.10.1.el8_3.x86_64
```

- 要以 MB 为单位显示空闲和使用的内存量：

```
# free --mega
total        used        free      shared  buff/cache   available
Mem:         2818         615        1183          12        1020        1957
Swap:        3124           0        3124
```

其他资源

- **podman-exec** man page

5.11. 在两个容器间共享文件

使用卷可以持久保留容器中的数据，即使容器被删除。卷可用于在多个容器间共享数据。卷是保存在主机上的文件夹。卷可以在容器和主机间共享。

主要优点是：

- 卷可以在容器间共享。
- 卷可以更容易备份或迁移。
- 卷不会增加容器的大小。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建卷：

```
$ podman volume create hostvolume
```

2. 显示卷的信息：

```
$ podman volume inspect hostvolume
[
    {
        "name": "hostvolume",
        "labels": {},
        "mountpoint":
"/home/username/.local/share/containers/storage/volumes/hostvolume/_data",
        "driver": "local",
        "options": {},
        "scope": "local"
    }
]
```

请注意，它会在卷目录中创建一个卷。您可以将挂载点路径保存到变量，以便更轻松地操作：**$ mntPoint=$(podman volume inspect hostvolume --format {{.Mountpoint}})**。

请注意，如果您运行 **sudo podman volume create hostvolume**，则挂载点会更改为 **/var/lib/containers/storage/volumes/hostvolume/_data**。

3. 使用存储在 **mntPoint** 变量中的路径来在目录中创建一个文本文件：

```
$ echo "Hello from host" >> $mntPoint/host.txt
```

4. 列出 **mntPoint** 变量所定义的目录中的所有文件：

```
$ ls $mntPoint/
host.txt
```

5. 运行名为 **myubi1** 的容器，并将主机上 **hostvolume** 卷名定义的目录映射到容器上的 **/containervolume1** 目录：

```
$ podman run -it --name myubi1 -v hostvolume:/containervolume1
registry.access.redhat.com/ubi9/ubi /bin/bash
```

请注意，如果您使用 **mntPoint** 变量定义的卷路径(**-v $mntPoint:/containervolume1**)，则在运行 **podman volume prune** 命令时数据可能会丢失，因为这会删除未使用的卷。始终使用 **-v *hostvolume_name:/containervolume_name***。

6. 列出容器上共享卷中的文件：

```
# ls /containervolume1
host.txt
```

您可以查看在主机上创建的 **host.txt** 文件。

7. 在 **/containervolume1** 目录中创建一个文本文件：

```
# echo "Hello from container 1" >> /containervolume1/container1.txt
```

8. 使用 **CTRL+p** 和 **CTRL+q** 从容器分离。

9. 列出主机上共享卷中的文件，您应该看到两个文件：

```
$ ls $mntPoint
container1.rxt  host.txt
```

此时，您要在容器和主机间共享文件。要在两个容器之间共享文件，请运行另一个名为 **myubi2** 的容器。

10. 运行名为 **myubi2** 的容器，并将主机上 **hostvolume** 卷名定义的目录映射到容器上的 **/containervolume2** 目录：

```
$ podman run -it --name myubi2 -v hostvolume:/containervolume2
registry.access.redhat.com/ubi9/ubi /bin/bash
```

11. 列出容器上共享卷中的文件：

```
# ls /containervolume2
container1.txt host.txt
```

您可以看到您在主机上创建的 **host.txt** 文件，以及您在 **myubi1** 容器中创建的 **container1.txt** 文件。

12. 在 **/containervolume2** 目录中创建一个文本文件：

```
# echo "Hello from container 2" >> /containervolume2/container2.txt
```

13. 使用 **CTRL+p** 和 **CTRL+q** 从容器分离。

14. 列出主机上共享卷中的文件，您应该看到三个文件：

```
$ ls $mntPoint
container1.rxt  container2.txt host.txt
```

其他资源

- **podman-volume** man page

5.12. 导出和导入容器

您可以使用 **podman export** 命令将正在运行的容器的文件系统导出到本地机器上的 tar 包中。例如，如果您有一个大容器，不常使用，或者您想要保存快照以便以后恢复该容器，则您可以使用 **podman export** 命令将正在运行的容器的当前快照导出到 tar 包中。

您可以使用 **podman import** 命令导入 tar 包，并将其保存为文件系统镜像。然后您可以运行此文件系统映像，或者将其用作其他镜像的层。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 根据 **registry.access.redhat.com/ubi9/ubi** 镜像运行 **myubi** 容器：

```
$ podman run -dt --name=myubi registry.access.redhat.com/9/ubi
```

2. 可选：列出所有容器：

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND       CREATED     STATUS
PORTS   NAMES
a6a6d4896142  registry.access.redhat.com/9:latest   /bin/bash     7 seconds ago  Up 7
seconds ago          myubi
```

3. 附加到 **myubi** 容器：

```
$ podman attach myubi
```

4. 创建名为 **testfile** 的文件：

```
[root@a6a6d4896142 /]# echo "hello" > testfile
```

5. 使用 **CTRL+p** 和 **CTRL+q** 从容器分离。

6. 将 **myubi** 的文件系统导出为本地机器上的 **myubi-container.tar** ：

```
$ podman export -o myubi.tar a6a6d4896142
```

7. 可选：列出当前目录内容：

```
$ ls -l
-rw-r--r--. 1 user user 210885120 Apr  6 10:50 myubi-container.tar
...
```

8. 可选：创建一个 **myubi-container** 目录，提取 **myubi-container.tar** 存档中的所有文件。以树形格式列出 **myubi-directory** 的内容：

```
$ mkdir myubi-container
$ tar -xf myubi-container.tar -C myubi-container
$ tree -L 1 myubi-container
├── bin -> usr/bin
├── boot
├── dev
├── etc
├── home
├── lib -> usr/lib
├── lib64 -> usr/lib64
├── lost+found
├── media
├── mnt
├── opt
```

```
├── proc
├── root
├── run
├── sbin -> usr/sbin
├── srv
├── sys
├── testfile
├── tmp
├── usr
└── var

20 directories, 1 file
```

您可以看到 **myubi-container.tar** 包含容器文件系统。

9. 导入 **myubi.tar**，并将其保存为文件系统镜像：

```
$ podman import myubi.tar myubi-imported
Getting image source signatures
Copying blob 277cab30fe96 done
Copying config c296689a17 done
Writing manifest to image destination
Storing signatures
c296689a17da2f33bf9d16071911636d7ce4d63f329741db679c3f41537e7cbf
```

10. 列出所有镜像：

```
$ podman images
REPOSITORY                          TAG     IMAGE ID      CREATED         SIZE
docker.io/library/myubi-imported    latest  c296689a17da  51 seconds ago  211 MB
```

11. 显示 **testfile** 文件的内容：

```
$ podman run -it --name=myubi-imported docker.io/library/myubi-imported cat testfile
hello
```

其他资源

- **podman-export** 手册页
- **podman-import** 手册页

5.13. 停止容器

使用 **podman stop** 命令来停止正在运行的容器。您可以根据其容器 ID 或名称来指定容器。

先决条件

- **container-tools** 元数据包已安装。
- 至少有一个容器正在运行。

流程

- 停止 **myubi** 容器：
  - 使用容器名称：

```
$ podman stop myubi
```

  - 使用容器 ID:

```
$ podman stop 1984555a2c27
```

要停止附加到终端会话的正在运行的容器，您可以在容器中输入 **exit** 命令。

podman stop 命令发送 SIGTERM 信号来终止正在运行的容器。如果容器在定义的时间段后没有停止（默认为 10 秒），Podman 会发送一个 SIGKILL 信号。

您还可以使用 **podman kill** 命令来终止一个容器(SIGKILL)，或向容器发送不同的信号。以下是向容器发送 SIGHUP 信号的示例（如果应用程序支持，SIGHUP 会使应用程序重新读取其配置文件）：

```
# *podman kill --signal="SIGHUP" 74b1da000a11*
74b1da000a114015886c557deec8bed9dfb80c888097aa83f30ca4074ff55fb2
```

其他资源

- **podman-stop** man page
- **podman-kill** man page

5.14. 删除容器

使用 **podman rm** 命令来删除容器。您可以使用容器 ID 或名称指定容器。

先决条件

- **container-tools** 元数据包已安装。
- 至少一个容器已经停止。

流程

1. 列出所有运行或停止的容器：

```
$ podman ps -a
CONTAINER ID IMAGE          COMMAND    CREATED    STATUS                PORTS NAMES
IS INFRA
d65aecc325a4 ubi9/ubi       /bin/bash  3 secs ago Exited (0) 5 secs ago peaceful_hopper false
74b1da000a11 rhel9/rsyslog rsyslog.sh 2 mins ago Up About a minute      musing_brown
false
```

2. 删除容器：
   - 要删除 **peaceful_hopper** 容器：

```
$ podman rm peaceful_hopper
```

请注意，**peaceful_hopper** 容器处于 Exited 状态，这表示它已停止，可以立即删除。

- 要删除 **musing_brown** 容器，首先停止容器，然后将其删除：

```
$ podman stop musing_brown
$ podman rm musing_brown
```

注意

- 要删除多个容器：

```
$ podman rm clever_yonath furious_shockley
```

- 要从本地系统中删除所有容器：

```
$ podman rm -a
```

其他资源

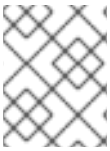
- **podman-rm** man page

5.15. 为容器创建 SELINUX 策略

要为容器生成 SELinux 策略，使用 UDICA 工具。如需更多信息，请参阅 udica SELinux 策略生成器简介。

5.16. 在 PODMAN 中配置预执行钩子

您可以创建插件脚本来定义对容器操作的精细控制，特别是阻止未经授权的操作，如拉取、运行或列出容器镜像。



注意

文件 **/etc/containers/podman_preexec_hooks.txt** 必须由管理员创建，可为空。如果 **/etc/containers/podman_preexec_hooks.txt** 不存在，插件脚本将不能执行。

以下规则适用于插件脚本：

- 必须是 root 所有且不可写。
- 必须位于 **/usr/libexec/podman/pre-exec-hooks** 和 **/etc/containers/pre-exec-hooks** 目录中。
- 按顺序和字母顺序执行。
- 如果所有插件脚本都返回零值，则 **podman** 命令执行了。
- 如果任何插件脚本返回一个非零值，则表示失败。**podman** 命令退出，并返回第一个失败的脚本的非零值。
- 红帽建议使用以下命名约定，按正确的顺序执行脚本：***DDD_name.lang***，其中：
  - ***DDD*** 是表示脚本执行顺序的十进制数。如有必要，前面使用一个或多个零。

- ***name*** 是插件脚本的名称。
- ***lang*** （可选）是给定编程语言的文件扩展名。例如，插件脚本的名称可以是：**001-check-groups.sh**.

注意

插件脚本在创建时是有效的。在插件脚本之前创建的容器不会受到影响。

先决条件

- **container-tools** 元数据包已安装。

流程

- 创建名为 **001-check-groups.sh** 的脚本插件。例如：

```
#!/bin/bash
if id -nG "$USER" 2> /dev/null | grep -qw "$GROUP" 2> /dev/null ; then
   exit 0
else
   exit 1
fi
```

  - 该脚本检查用户是否在指定的组中。
  - **USER** 和 **GROUP** 是 Podman 设置的环境变量。
  - **001-check-groups.sh** 脚本提供的退出码将被提供给 **podman** 二进制文件。
  - **podman** 命令退出，并返回第一个失败的脚本的非零值。

验证

- 检查 **001-check-groups.sh** 脚本是否正常工作：

```
$ podman run image
...
```

如果用户不在正确的组中，则会出现以下错误：

```
external preexec hook /etc/containers/pre-exec-hooks/001-check-groups.sh failed
```

5.17. 在容器中调试应用程序

您可以使用为故障排除的不同方面量身定制的各种命令行工具。如需更多信息，请参阅 在容器中调试应用程序。

第 6 章 选择容器运行时

runc 和 crun 是容器运行时，可以互换使用，因为二者都实现 OCI 运行时规范。与 runc 相比，crun 容器运行时有一些优点，因为它速度更快，且需要较少的内存。因此，crun 容器运行时是推荐使用的容器运行时。

6.1. RUNC 容器运行时

runc 容器运行时是开放容器项目(OCI)容器运行时规范的一个轻量级的、可移植的实现。runc 运行时与 Docker 共享大量低级代码，但不依赖于 Docker 平台的任何组件。runc 支持 Linux 命名空间、实时迁移，并有可移植的性能配置文件。

它还完全支持 Linux 安全特性，比如 SELinux、控制组群(cgroups)、seccomp 等等。您可以使用 runc 构建并运行镜像，或者您可以使用 runc 运行 OCI 兼容的镜像。

6.2. CRUN 容器运行时

crun 是一个快速、占用内存少的 OCI 容器运行时，是用 C 语言编写的。crun 二进制文件比 runc 二进制文件小多达 50 倍，快两倍。使用 crun，也可以在运行容器时设置最少的进程数。crun 运行时也支持 OCI hook。

crun 的其他功能包括：

- 对 rootless 容器按组共享文件
- 控制 OCI hook 的 stdout 和 stderr
- 在 cgroup v2 上运行旧版本的 **systemd**
- 其他程序使用的 C 库
- 可扩展性
- 可移植性

其他资源

- crun 简介，一个快速且占用内存少的容器运行时

6.3. 运行带有 RUNC 和 CRUN 的容器

有了 runc 或 crun，容器可以使用捆绑包进行配置。容器的捆绑包是一个目录，其中包含一个名为 **config.json** 的规范文件和根文件系统。根文件系统包含容器的内容。

注意

<runtime> 可以是 crun 或 runc。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 拉取 **registry.access.redhat.com/ubi9/ubi** 容器镜像：

```
# podman pull registry.access.redhat.com/ubi9/ubi
```

2. 将 **registry.access.redhat.com/ubi9/ubi** 镜像导出到 **rhel.tar** 归档：

```
# podman export $(podman create registry.access.redhat.com/ubi9/ubi) > rhel.tar
```

3. 创建 **bundle/rootfs** 目录：

```
# mkdir -p bundle/rootfs
```

4. 将 **rhel.tar** 归档解压到 **bundle/rootfs** 目录中：

```
# tar -C bundle/rootfs -xf rhel.tar
```

5. 为捆绑包创建一个名为 **config.json** 的新规范文件：

```
# <runtime> spec -b bundle
```

- **-b** 选项指定捆绑包目录。默认值为当前目录。

6. 可选：更改设置：

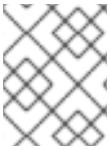
```
# vi bundle/config.json
```

7. 为捆绑包创建一个名为 **myubi** 的容器实例：

```
# <runtime> create -b bundle/ myubi
```

8. 启动 **myubi** 容器：

```
# <runtime> start myubi
```



注意

容器实例的名称对于主机来说必须是唯一的。要启动容器的一个新实例：**# <runtime> start <container_name>**

验证

- 列出由 ***<runtime>*** 启动的容器：

```
# <runtime> list
ID          PID         STATUS      BUNDLE         CREATED                          OWNER
myubi       0           stopped     /root/bundle   2021-09-14T09:52:26.659714605Z   root
```

其他资源

- **crun** 手册页
- **runc** 手册页

- crun 简介，一个快速且占用内存少的容器运行时

6.4. 临时更改容器运行时

您可以使用 **podman run** 命令和 **--runtime** 选项来更改容器运行时。

注意

<runtime> 可以是 crun 或 runc。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 拉取 **registry.access.redhat.com/ubi9/ubi** 容器镜像：

```
$ podman pull registry.access.redhat.com/ubi9/ubi
```

2. 使用 **--runtime** 选项更改容器运行时：

```
$ podman run --name=myubi -dt --runtime=<runtime> ubi9
e4654eb4df12ac031f1d0f2657dc4ae6ff8eb0085bf114623b66cc664072e69b
```

3. 可选。列出所有镜像：

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND  CREATED        STATUS
PORTS   NAMES
e4654eb4df12  registry.access.redhat.com/ubi9:latest  bash     4 seconds ago  Up 4 seconds
ago          myubi
```

验证

- 确保 myubi 容器中的 OCI 运行时设为 ***<runtime>*** ：

```
$ podman inspect myubi --format "{{.OCIRuntime}}"
<runtime>
```

其他资源

- crun 简介，一个快速且占用内存少的容器运行时

6.5. 永久更改容器运行时

您可以以 root 用户身份在 **/etc/containers/containers.conf** 配置文件中设置容器运行时及其选项，或者以非 root 用户身份在 **$HOME/.config/containers/containers.conf** 配置文件中设置容器运行时及其选项。

注意

<runtime> 可以是 crun 或 runc 运行时。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 更改 **/etc/containers/containers.conf** 文件中的运行时：

```
# vim /etc/containers/containers.conf
[engine]
runtime = "<runtime>"
```

2. 运行名为 myubi 的容器：

```
# podman run --name=myubi -dt ubi9 bash
Resolved "ubi9" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi9:latest…
...
Storing signatures
```

验证

- 确保 **myubi** 容器中的 OCI 运行时设为 ***<runtime>*** ：

```
# podman inspect myubi --format "{{.OCIRuntime}}"
<runtime>
```

其他资源

- crun 简介，一个快速且占用内存少的容器运行时
- **containers.conf** 手册页

第 7 章 将软件添加到 UBI 容器中

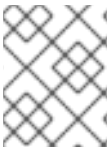
红帽通用基础镜像 (UBI) 是从 RHEL 内容的子集构建的。UBI 还提供了 RHEL 软件包的一个子集，其可以免费安装并与 UBI 一起使用。要在正在运行的容器中添加或更新软件，您可以使用包含 RPM 软件包和更新的 dnf 软件仓库。UBI 提供了一组预先构建的语言运行时容器镜像，如 Python、Perl、Node.js、Ruby，等等。

要将 UBI 存储库中的软件包添加到正在运行的 UBI 容器中：

- 在 UBI init 和 UBI 标准镜像中，使用 **dnf** 命令
- 在 UBI 最小镜像上，请使用 **microdnf** 命令

注意

直接在运行的容器中安装和使用软件包会临时添加一些软件包。更改不会保存在容器镜像中。要使软件包更改持久，请参阅 使用 Buildah 从 Containerfile 构建镜像 部分。



注意

当您将软件添加到 UBI 容器时，其流程与在订阅的 RHEL 主机上或未订阅（或非 RHEL）系统上更新 UBI 的流程有所不同。

7.1. 使用 UBI INIT 镜像

您可以使用安装并配置 Web 服务器(**httpd**)的 **Containerfile** 构建一个容器，来在容器在主机系统上运行时由 **systemd** 服务(**/sbin/init**)自动启动。**podman build** 命令使用一个或多个 **Containerfiles** 中指令和一个指定的构建上下文目录构建镜像。上下文目录可以指定为存档的 URL、Git 存储库或 **Containerfile**。如果没有指定上下文目录，则当前工作目录被视为构建上下文，且必须包含 **Containerfile**。您还可以使用 **--file** 选项指定 **Containerfile**。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 在新目录中创建一个包含以下内容的 **Containerfile** ：

```
FROM registry.access.redhat.com/ubi9/ubi-init
RUN dnf -y install httpd; dnf clean all; systemctl enable httpd;
RUN echo "Successful Web Server Test" > /var/www/html/index.html
RUN mkdir /etc/systemd/system/httpd.service.d/; echo -e '[Service]\nRestart=always' >
/etc/systemd/system/httpd.service.d/httpd.conf
EXPOSE 80
CMD [ "/sbin/init" ]
```

Containerfile 安装 **httpd** 软件包，使 **httpd** 服务在引导时启动，创建了一个测试文件 (**index.html**)，将 Web 服务器公开给主机（端口 80），并在容器启动时启动 **systemd** init 服务 (**/sbin/init**)。

2. 构建容器：

```
# podman build --format=docker -t mysysd .
```

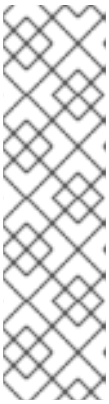
3. 可选：如果要使用 **systemd** 运行容器，且在您的系统上启用了 SELinux，您必须设置 **container_manage_cgroup** 布尔值变量：

```
# setsebool -P container_manage_cgroup 1
```

4. 运行名为 **mysysd_run** 的容器：

```
# podman run -d --name=mysysd_run -p 80:80 mysysd
```

mysysd 镜像作为 **mysysd_run** 容器运行，并作为守护进程，容器的端口 80 暴露在主机系统上的端口 80 上。



注意

在 rootless 模式中，您必须选择主机端口号 >= 1024。例如：

```
$ podman run -d --name=mysysd -p 8081:80 mysysd
```

要使用 < 1024 的端口号，您必须修改 **net.ipv4.ip_unprivileged_port_start** 变量：

```
# sysctl net.ipv4.ip_unprivileged_port_start=80
```

5. 检查容器是否正在运行：

```
# podman ps
a282b0c2ad3d  localhost/mysysd:latest  /sbin/init  15 seconds ago  Up 14 seconds ago
0.0.0.0:80->80/tcp  mysysd_run
```

6. 测试 Web 服务器：

```
# curl localhost/index.html
Successful Web Server Test
```

其他资源

- Rootless Podman 的缺点

7.2. 使用 UBI 微镜像

您可以使用 Buildah 工具构建一个 **ubi-micro** 容器镜像。

先决条件

- **container-tools** 元数据包已安装。

先决条件

- **containers-tool** 元软件包提供的 **podman** 工具已安装。

流程

1. 拉取并构建 **registry.access.redhat.com/ubi8/ubi-micro** 镜像：

```
# microcontainer=$(buildah from registry.access.redhat.com/ubi9/ubi-micro)
```

2. 挂载可正常工作的容器根文件系统：

```
# micromount=$(buildah mount $microcontainer)
```

3. 将 **httpd** 服务安装到 **micromount** 目录中：

```
# dnf install \
   --installroot $micromount \
   --releasever=/ \
   --setopt install_weak_deps=false \
   --setopt=reposdir=/etc/yum.repos.d/ \
   --nodocs -y \
   httpd
# dnf clean all \
   --installroot $micromount
```

4. 在工作容器中卸载根文件系统：

```
# buildah umount $microcontainer
```

5. 从工作容器创建 **ubi-micro-httpd** 镜像：

```
# buildah commit $microcontainer ubi-micro-httpd
```

验证步骤

1. 显示有关 **ubi-micro-httpd** 镜像的详情：

```
# podman images ubi-micro-httpd
localhost/ubi-micro-httpd latest 7c557e7fbe9f  22 minutes ago  151 MB
```

7.3. 在订阅的主机上将软件添加到 UBI 容器中

如果您在注册和订阅的 RHEL 主机上运行 UBI 容器，会在标准 UBI 容器以及所有 UBI 软件仓库中启用 RHEL Base 和 AppStream 存储库。

其他资源

- 通用基础镜像(UBI)：镜像、存储库、软件包和源代码

7.4. 在标准 UBI 容器中添加软件

要在标准 UBI 容器中添加软件，请禁用非 UBI dnf 存储库，以确保您构建的容器可以被重新分发。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 拉取并运行 **registry.access.redhat.com/ubi9/ubi** 镜像：

```
$ podman run -it --name myubi registry.access.redhat.com/ubi9/ubi
```

2. 将软件包添加到 **myubi** 容器。

- 要添加在 UBI 存储库中的软件包，请禁用除 UBI 软件仓库以外的所有 dnf 软件仓库。例如，要添加 **bzip2** 软件包：

```
# dnf install --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-baseos-rpms bzip2
```

- 要添加不在 UBI 存储库中的软件包，请不要禁用任何存储库。例如，要添加 **zsh** 软件包：

```
# dnf install zsh
```

- 要添加位于不同主机存储库中的软件包，请明确启用您需要的存储库。例如，要安装 **codeready-builder-for-rhel-8-x86_64-rpms** 存储库中的 **python38-devel** 软件包：

```
# dnf install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms python38-devel
```

验证步骤

1. 列出容器内所有启用的存储库：

```
# dnf repolist
```

2. 确保列出了所需的存储库。

3. 列出所有安装的软件包：

```
# rpm -qa
```

4. 确保列出了所需的软件包。

注意

安装不在 Red Hat UBI 存储库中的红帽软件包可能会限制在订阅的 RHEL 系统外分发容器的功能。

7.5. 在最小的 UBI 容器中添加软件

UBI dnf 软件仓库默认在 UBI Minimal 镜像中启用。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 拉取并运行 **registry.access.redhat.com/ubi9/ubi-minimal** 镜像：

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi9/ubi-minimal
```

2. 将软件包添加到 **myubimin** 容器：

- 要添加 UBI 存储库中的软件包，请不要禁用任何存储库。例如，要添加 **bzip2** 软件包：

```
# microdnf install bzip2
```

- 要添加位于不同主机存储库中的软件包，请明确启用您需要的存储库。例如，要安装 **codeready-builder-for-rhel-8-x86_64-rpms** 存储库中的 **python38-devel** 软件包：

```
# microdnf install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms
python38-devel
```

验证步骤

1. 列出容器内所有启用的存储库：

```
# microdnf repolist
```

2. 确保列出了所需的存储库。
3. 列出所有安装的软件包：

```
# rpm -qa
```

4. 确保列出了所需的软件包。

注意

安装不在 Red Hat UBI 存储库中的红帽软件包可能会限制在订阅的 RHEL 系统外分发容器的功能。

7.6. 将软件添加到未订阅的主机上的 UBI 容器中

在未订阅的 RHEL 系统上添加软件包时，您不必禁用任何存储库。

先决条件

- **container-tools** 元数据包已安装。

流程

- 将软件包添加到基于 UBI 标准或 UBI init 镜像的正在运行的容器中。不要禁用任何存储库。使用 **podman run** 命令运行容器，然后使用 **dnf install** 命令在容器内使用。
  - 例如，要将 **bzip2** 软件包添加到基于 UBI 标准的容器中：

```
$ podman run -it --name myubi registry.access.redhat.com/ubi9/ubi
# dnf install bzip2
```

- 例如，要将 **bzip2** 软件包添加到基于 UBI init 的容器中：

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi9/ubi-minimal
# microdnf install bzip2
```

验证步骤

1. 列出所有启用的存储库：
   - 要列出基于 UBI 标准或 UBI init 镜像的容器中所有启用的存储库：

```
#  dnf repolist
```

   - 要列出基于 UBI 最小容器的容器中所有启用的存储库：

```
# microdnf repolist
```

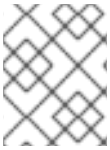
2. 确保列出了所需的存储库。
3. 列出所有安装的软件包：

```
# rpm -qa
```

4. 确保列出了所需的软件包。

7.7. 构建基于 UBI 的镜像

您可以使用 Buildah 工具从 **Containerfile** 创建基于 UBI 的 web 服务器容器。您必须禁用所有非UBI dnf 软件仓库，以确保您的镜像只包含可重新分发的红帽软件。



注意

对于 UBI 最小镜像，使用 **microdnf** 而不是使用 **dnf** ：**RUN microdnf update -y && rm -rf /var/cache/yum** 和 **RUN microdnf install httpd -y && microdnf clean all** 命令。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建 **Containerfile** ：

```
FROM registry.access.redhat.com/ubi9/ubi
USER root
LABEL maintainer="John Doe"
# Update image
RUN dnf update --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-baseos-rpms -y && rm -rf /var/cache/yum
RUN dnf install --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-baseos-rpms httpd -y && rm -rf /var/cache/yum
# Add default Web page and expose port
RUN echo "The Web Server is Running" > /var/www/html/index.html
```

```
EXPOSE 80
# Start the service
CMD ["-D", "FOREGROUND"]
ENTRYPOINT ["/usr/sbin/httpd"]
```

2. 构建容器镜像：

```
# buildah bud -t johndoe/webserver .
STEP 1: FROM registry.access.redhat.com/ubi9/ubi:latest
STEP 2: USER root
STEP 3: LABEL maintainer="John Doe"
STEP 4: RUN dnf update --disablerepo=* --enablerepo=ubi-8-appstream-rpms --
enablerepo=ubi-8-baseos-rpms -y
...
Writing manifest to image destination
Storing signatures
--> f9874f27050
f9874f270500c255b950e751e53d37c6f8f6dba13425d42f30c2a8ef26b769f2
```

验证步骤

1. 运行 web 服务器：

```
# podman run -d --name=myweb -p 80:80 johndoe/webserver
bbe98c71d18720d966e4567949888dc4fb86eec7d304e785d5177168a5965f64
```

2. 测试 Web 服务器：

```
# curl http://localhost/index.html
The Web Server is Running
```

7.8. 使用 APPLICATION STREAM 运行时镜像

基于 Application Streams 的运行时镜像提供了一组容器镜像，您可以将它们用作容器构建的基础。

支持的运行时镜像有 Python、Ruby、s2-core、s2i-base、.NET Core、PHP。红帽容器目录 中提供运行时镜像。

注意

由于这些 UBI 镜像包含与旧镜像相同基本软件，因此您可以从 使用 Red Hat Software Collections Container Images 指南 中了解这些镜像。

其他资源

- Red Hat Container Catalog
- Red Hat Container Image Updates

7.9. 获取 UBI 容器镜像源代码

源代码以可下载容器镜像的形式提供给所有基于 UBI 的红帽镜像。虽然源容器镜像被打包为容器，但无法运行。要在您的系统上安装红帽源容器镜像，请使用 **skopeo** 命令，而不是 **podman pull** 命令。

源容器镜像根据其代表的二进制容器命名。例如，对于特定标准 RHEL UBI 9 容器 **registry.access.redhat.com/ubi9:8.1-397** 附加 **-source** 来获取源容器镜像 (**registry.access.redhat.com/ubi9:8.1-397-source**)。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 使用 **skopeo copy** 命令将源容器镜像复制到本地目录中：

```
$ skopeo copy \
docker://registry.access.redhat.com/ubi9:8.1-397-source \
dir:$HOME/TEST
...
Copying blob 477bc8106765 done
Copying blob c438818481d3 done
...
Writing manifest to image destination
Storing signatures
```

2. 使用 **skopeo inspect** 命令检查源容器镜像：

```
$ skopeo inspect dir:$HOME/TEST
{
    "Digest":
"sha256:7ab721ef3305271bbb629a6db065c59bbeb87bc53e7cbf88e2953a1217ba7322",
    "RepoTags": [],
    "Created": "2020-02-11T12:14:18.612461174Z",
    "DockerVersion": "",
    "Labels": null,
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
        "sha256:1ae73d938ab9f11718d0f6a4148eb07d38ac1c0a70b1d03e751de8bf3c2c87fa",
        "sha256:9fe966885cb8712c47efe5ecc2eaa0797a0d5ffb8b119c4bd4b400cc9e255421",
        "sha256:61b2527a4b836a4efbb82dfd449c0556c0f769570a6c02e112f88f8bbcd90166",
        ...
        "sha256:cc56c782b513e2bdd2cc2af77b69e13df4ab624ddb856c4d086206b46b9b9e5f",
        "sha256:dcf9396fdada4e6c1ce667b306b7f08a83c9e6b39d0955c481b8ea5b2a465b32",

"sha256:feb6d2ae252402ea6a6fca8a158a7d32c7e4572db0e6e5a5eab15d4e0777951e"
    ],
    "Env": null
}
```

3. 解包所有内容：

```
$ cd $HOME/TEST
$ for f in $(ls); do tar xvf $f; done
```

4. 检查结果：

```
$ find blobs/ rpm_dir/
blobs/
blobs/sha256
blobs/sha256/10914f1fff060ce31388f5ab963871870535aaaa551629f5ad182384d60fdf82
rpm_dir/
rpm_dir/gzip-1.9-4.el8.src.rpm
```

如果结果正确，则镜像可以使用。

注意

发布容器镜像后可能需要几小时时间才能使用其相关源容器。

其他资源

- **skopeo-copy** 手册页
- **skopeo-inspect** 手册页

第 8 章 签名容器镜像

您可以使用 GNU Privacy Guard (GPG) 签名或 sigstore 签名来签署容器镜像。两种签名技术均与任何 OCI 兼容容器 registry 兼容。在将镜像推送至远程注册中心并配置消费者之前，您可以使用 Podman 来签署镜像，以便拒绝任何未签名镜像。签名容器镜像有助于防止供应链攻击。

使用 GPG 密钥进行签名需要部署单独的查找服务器来分发签名。外观服务器可以是任何 HTTP 服务器。从 Podman 版本 4.2 开始，您可以使用容器签名的 sigstore 格式。与 GPG 密钥相比，不需要单独的查找服务器，因为 sigstore 签名存储在容器注册中心中。

8.1. 使用 GPG 签名对容器镜像进行签名

您可以使用 GNU Privacy Guard (GPG) 密钥为镜像签名。

先决条件

- **container-tools** 元数据包已安装。
- 已安装 GPG 工具。
- 设置查找 Web 服务器，您可以发布它上的文件。
  - 您可以在 **/etc/containers/registries.d/default.yaml** 文件中检查系统范围的 registry 配置。**lookaside-staging** 选项引用签名写入的文件路径，通常在主机发布签名中设置。

```
# cat /etc/containers/registries.d/default.yaml
docker:
   <registry>:
      lookaside: https://registry-lookaside.example.com
      lookaside-staging: file:///var/lib/containers/sigstore
...
```

流程

1. 生成 GPG 密钥：

```
# gpg --full-gen-key
```

2. 导出公钥：

```
# gpg --output <path>/key.gpg --armor --export <username@domain.com>
```

3. 使用 **Containerfile** 在当前目录中构建容器镜像：

```
$ podman build -t <registry>/<namespace>/<image>
```

将 ***<registry>***, ***<namespace>***, and ***<image>*** 替换为容器进行的标识符。如需了解更多详细信息，请参阅 容器注册中心。

4. 为镜像签名并将其推送到注册中心：

```
$ podman push \
  --sign-by <username@domain.com> \
  <registry>/<namespace>/<image>
```

注意

如果您需要在容器 registry 间移动现有镜像时为现有镜像签名，您可以使用 **skopeo copy** 命令。

5. 可选：显示新镜像签名：

```
# (cd /var/lib/containers/sigstore/; find . -type f)
./<image>@sha256=<digest>/signature-1
```

6. 将本地签名复制到 lookaside web 服务器中：

```
# rsync -a /var/lib/containers/sigstore <user@registry-
lookaside.example.com>:/registry-lookaside/webroot/sigstore
```

签名存储在 **lookaside-staging** 选项决定的位置，本例中为 **/var/lib/containers/sigstore** 目录。

验证

- 如需了解更多详细信息，请参阅 验证 GPG 镜像签名。

其他资源

- **podman-image-trust** 手册页
- **podman-push** man page
- **podman-build** man page
- 如何生成 GPG 密钥对

8.2. 验证 GPG 镜像签名

您可以按照以下流程验证是否使用 GPG 密钥正确签名容器镜像。

先决条件

- **container-tools** 元数据包已安装。
- 设置签名读取的 Web 服务器，您可以在其上发布文件。
  - 您可以在 **/etc/containers/registries.d/default.yaml** 文件中检查系统范围的 registry 配置。**lookaside** 选项引用用于签名读取的 Web 服务器。必须设置 **lookaside** 选项以验证签名。

```
# cat /etc/containers/registries.d/default.yaml
docker:
    <registry>:
```

```
        lookaside: https://registry-lookaside.example.com
        lookaside-staging: file:///var/lib/containers/sigstore
    ...
```

流程

1. 为 ***<registry>*** 更新信任范围：

```
$ podman image trust set -f <path>/key.gpg <registry>/<namespace>
```

2. 可选：显示 **/etc/containers/policy.json** 文件来验证信任策略配置：

```
$ cat /etc/containers/policy.json
{
  ...
  "transports": {
    "docker": {
      "<registry>/<namespace>": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "<path>/key.gpg"
        }
      ]
    }
  }
}
```

注意

通常，**/etc/containers.policy.json** 文件在使用相同的键的机构级别进行配置。例如，***<registry>/<namespace>*** 用于公共注册中心，***<registry>*** 用于只针对单一公司的专用注册中心。

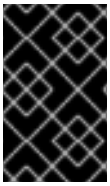
3. 拉取镜像：

```
# podman pull <registry>/<namespace>/<image>
...
Storing signatures
e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a99776a
```

podman pull 命令会根据配置强制实施签名存在，不需要额外的选项。

注意

您可以在 **/etc/containers/registries.d/default.yaml** 文件中编辑系统范围的注册中心配置。您还可以编辑 **/etc/containers/registries.d** 目录中任何 YAML 文件中的注册中心或存储库配置部分。所有 YAML 文件都是读取的，文件名可以是任意的。单个范围 (default-docker、注册中心或命名空间) 只能存在于 **/etc/containers/registries.d** 目录中的一个文件中。



重要

/etc/containers/registries.d/default.yaml 文件中的系统范围 registry 配置可以访问已发布的签名。**sigstore** 和 **sigstore-staging** 选项现已弃用。这些选项引用签名存储，它们没有连接到 sigstore 签名格式。使用新的、等同的 **lookaside** 和 **lookaside-staging** 选项。

其他资源

- **podman-image-trust** 手册页
- **podman-pull** 手册页

8.3. 使用私钥使用 SIGSTORE 签名签名容器镜像

从 Podman 版本 4.2 开始，您可以使用容器签名的 sigstore 格式。

先决条件

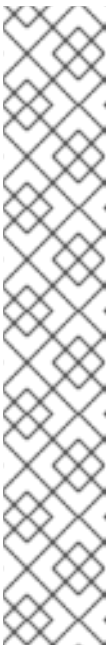
- **container-tools** 元数据包已安装。

流程

1. 生成 sigstore 公钥/私钥对：

```
$ skopeo generate-sigstore-key --output-prefix myKey
```

   - 公钥和私钥 ***myKey.pub*** 和 ***myKey.private*** 生成了。



注意

skopeo generate-sigstore-key 命令可在 RHEL 9.2 中提供。否则，您必须使用上游 Cosign 项目来生成公钥/私钥对：

   - 安装 cosign 工具：

```
$ git clone -b v2.0.0 https://github.com/sigstore/cosign
$ cd cosign
$ make ./cosign
```

   - 生成公钥/私钥对：

```
$ ./cosign generate-key-pair
...
Private key written to cosign.key
Public key written to cosign.pub
```

2. 在 **/etc/containers/registries.d/default.yaml** 文件中添加以下内容：

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

通过设置 **use-sigstore-attachments** 选项，Podman 和 Skopeo 可以借助镜像读取和写入容器 sigstore 签名，并将它们保存在与签名镜像相同的存储库中。

注意

您可以在 **/etc/containers/registries.d/default.yaml** 文件中编辑系统范围的注册中心配置。您还可以编辑 **/etc/containers/registries.d** 目录中任何 YAML 文件中的注册中心或存储库配置部分。所有 YAML 文件都是读取的，文件名可以是任意的。单个范围 (default-docker、注册中心或命名空间) 只能存在于 **/etc/containers/registries.d** 目录中的一个文件中。

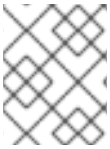
3. 使用 **Containerfile** 在当前目录中构建容器镜像：

```
$ podman build -t <registry>/<namespace>/<image>
```

4. 为镜像签名并将其推送到注册中心：

```
$ podman push --sign-by-sigstore-private-key ./myKey.private
<registry>/<namespace>/image>
```

podman push 命令将 ***<registry>/<namespace>/<image>*** 本地镜像推送到远程的注册中心作为 ***<registry>/<namespace>/<image>***。**--sign-by-sigstore-private-key** 选项使用 **myKey.private** 私钥将 sigstore 签名添加到 ***<registry>/<namespace>/<image>*** 镜像。镜像和 sigstore 签名将上传到远程注册中心。



注意

如果您需要在容器 registry 间移动现有镜像时为现有镜像签名，您可以使用 **skopeo copy** 命令。

验证

- 如需了解更多详细信息，请参阅 使用公钥验证 sigstore 镜像签名。

其他资源

- **podman-push** 手册页
- **podman-build** 手册页
- Sigstore：对软件供应链信任和安全的公开回答

8.4. 使用公钥验证 SIGSTORE 镜像签名

您可以使用以下步骤验证容器镜像是否已正确签名。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 在 **/etc/containers/registries.d/default.yaml** 文件中添加以下内容：

```
docker:
   <registry>:
      use-sigstore-attachments: true
```

通过设置 **use-sigstore-attachments** 选项，Podman 和 Skopeo 可以借助镜像读取和写入容器 sigstore 签名，并将它们保存在与签名镜像相同的存储库中。

注意

您可以在 **/etc/containers/registries.d/default.yaml** 文件中编辑系统范围的注册中心配置。您还可以编辑 **/etc/containers/registries.d** 目录中任何 YAML 文件中的注册中心或存储库配置部分。所有 YAML 文件都是读取的，文件名可以是任意的。单个范围 (default-docker、注册中心或命名空间) 只能存在于 **/etc/containers/registries.d** 目录中的一个文件中。

2. 编辑 **/etc/containers/policy.json** 文件以强制 sigstore 签名存在：

```
...
"transports": {
        "docker": {
            "<registry>/<namespace>": [
                {
                    "type": "sigstoreSigned",
                    "keyPath": "/some/path/to/cosign.pub"
                }
            ]
        }
    }
...
```

通过修改 **/etc/containers/policy.json** 配置文件，您可以更改信任策略配置。Podman、Buildah 和 Skopeo 强制执行容器镜像签名的存在。

3. 拉取镜像：

```
$ podman pull <registry>/<namespace>/<image>
```

podman pull 命令会根据配置强制实施签名存在，不需要额外的选项。

其他资源

- Sigstore：对软件供应链信任和安全的公开回答

8.5. 使用 FULCIO 和 REKOR 使用 SIGSTORE 签名签名容器镜像

有了 Fulcio 和 Rekor 服务器，您现在可以根据 OpenID Connect (OIDC)服务器身份验证使用短期证书来创建签名，而不是手动管理私钥。

先决条件

- **container-tools** 元数据包已安装。

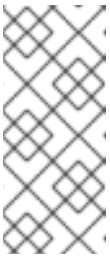
- 您有 Fulcio (https:// *<your-fulcio-server>*)和 Rekor (https:// *<your-rekor-server>*)服务器正在运行，且配置了。
- 您已安装了 Podman v4.4 或更高版本。

流程

1. 将以下内容添加到 **/etc/containers/registries.conf.d/default.yaml** 文件中：

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

- 通过设置 **use-sigstore-attachments** 选项，Podman 和 Skopeo 可以借助镜像读取和写入容器 sigstore 签名，并将它们保存在与签名镜像相同的存储库中。



注意

您可以编辑 **/etc/containers/registries.d** 目录中的任何 YAML 文件中的注册中心或存储库配置部分。单个范围 (default-docker、注册中心或命名空间) 只能存在于 **/etc/containers/registries.d** 目录中的一个文件中。您还可以在 **/etc/containers/registries.d/default.yaml** 文件中编辑系统范围的注册中心配置。请注意，所有 YAML 文件都是可读的，文件名是任意的。

2. 创建 ***file.yml*** 文件：

```
fulcio:
  fulcioURL: "https://<your-fulcio-server>"
  oidcMode: "interactive"
  oidcIssuerURL: "https://<your-OIDC-provider>"
  oidcClientID: "sigstore"
rekorURL: "https://<your-rekor-server>"
```

- ***file.yml*** 是sigstore 签名参数 YAML 文件，用于存储创建 sigstore 签名所需的选项。

3. 为镜像签名并将其推送到注册中心：

```
$ podman push --sign-by-sigstore=file.yml <registry>/<namespace>/<image>
```

- 您还可以使用带有类似 **--sign-by-sigstore** 选项的 **skopeo copy** 命令来为现有镜像签名，同时将它们在多个容器注册中心移动。

警告

请注意，您对公共服务器的提交包含有关公钥和证书的数据，以及有关签名的元数据。

验证

- 使用 Fulcio 和 Rekor 验证带有 sigstore 签名的容器镜像

其他资源

- **containers-sigstore-signing-params.yaml** man page
- **podman-push** 手册页
- **container-registries.d** 手册页

8.6. 使用 FULCIO 和 REKOR 验证带有 SIGSTORE 签名的容器镜像

您可以通过向 **policy.json** 文件中添加 Fulcio 和 Rekor-related 信息来验证镜像签名。验证容器镜像签名可确保镜像来自可信的源，且没有被篡改或修改。

先决条件

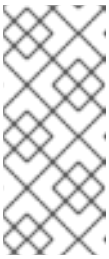
- **container-tools** 元数据包已安装。

流程

1. 将以下内容添加到 **/etc/containers/registries.conf.d/default.yaml** 文件中：

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

- 通过设置 **use-sigstore-attachments** 选项，Podman 和 Skopeo 可以借助镜像读取和写入容器 sigstore 签名，并将它们保存在与签名镜像相同的存储库中。



注意

您可以编辑 **/etc/containers/registries.d** 目录中的任何 YAML 文件中的注册中心或存储库配置部分。单个范围 (default-docker、注册中心或命名空间) 只能存在于 **/etc/containers/registries.d** 目录中的一个文件中。您还可以在 **/etc/containers/registries.d/default.yaml** 文件中编辑系统范围的注册中心配置。请注意，所有 YAML 文件都是可读的，文件名是任意的。

2. 在 **/etc/containers/policy.json** 文件中添加 **fulcio** 部分和 **rekorPublicKeyPath** 或 **rekorPublicKeyData** 字段：

```
{
    ...
    "transports": {
 "docker": {
        "<registry>/<namespace>": [
            {
            "type": "sigstoreSigned",
            "fulcio": {
                "caPath": "/path/to/local/CA/file",
                "oidcIssuer": "https://expected.OIDC.issuer/",
                "subjectEmail", "expected-signing-user@example.com",
            },
            "rekorPublicKeyPath": "/path/to/local/public/key/file",
```

```
        }
      ]
    ...
  }
  }
  ...
}
```

- **fulcio** 部分提供基于 Fulcio 发布的证书的签名。
- 您必须指定 **caPath** 和 **caData** 字段中的一个，其中包含 Fulcio 实例的 CA 证书。
- **oidcIssuer** 和 **subjectEmail** 是必需的，准确地指定预期的身份提供程序，以及获取 Fulcio 证书的用户的身份。
- 您必须指定 **rekorPublicKeyPath** 和 **rekorPublicKeyData** 字段之一。

3. 拉取镜像：

```
$ podman pull <registry>/<namespace>/<image>
```

podman pull 命令会根据配置强制实施签名存在，不需要额外的选项。

其他资源

- **policy.json** 手册页
- **container-registries.d** 手册页

8.7. 使用私钥和 REKOR 签名带有 SIGSTORE 签名的容器镜像

从 Podman 版本 4.4 开始，您可以将容器签名的 sigstore 格式与 Rekor 服务器一起使用。您还可以将公共签名上传到公共 rekor.sigstore.dev 服务器，这会增加 Cosign 的互操作性。然后，您可以使用 **cosign verify** 命令验证您的签名，而无需明确禁用 Rekor。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 生成 sigstore 公钥/私钥对：

```
$ skopeo generate-sigstore-key --output-prefix myKey
```

- 公钥和私钥 ***myKey.pub*** 和 ***myKey.private*** 生成了。

2. 将以下内容添加到 **/etc/containers/registries.conf.d/default.yaml** 文件中：

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

- 通过设置 **use-sigstore-attachments** 选项，Podman 和 Skopeo 可以借助镜像读取和写入容器 sigstore 签名，并将它们保存在与签名镜像相同的存储库中。

注意

您可以编辑 **/etc/containers/registries.d** 目录中的任何 YAML 文件中的注册中心或存储库配置部分。单个范围 (default-docker、注册中心或命名空间) 只能存在于 **/etc/containers/registries.d** 目录中的一个文件中。您还可以在 **/etc/containers/registries.d/default.yaml** 文件中编辑系统范围的注册中心配置。请注意，所有 YAML 文件都是可读的，文件名是任意的。

3. 使用 **Containerfile** 在当前目录中构建容器镜像：

```
$ podman build -t <registry>/<namespace>/<image>
```

4. 创建 ***file.yml*** 文件：

```
privateKeyFile: "/home/user/sigstore/myKey.private"
privateKeyPassphraseFile: "/mnt/user/sigstore-myKey-passphrase"
rekorURL: "https://<your-rekor-server>"
```

- ***file.yml*** 是sigstore 签名参数 YAML 文件，用于存储创建 sigstore 签名所需的选项。

5. 为镜像签名并将其推送到注册中心：

```
$ podman push --sign-by-sigstore=file.yml <registry>/<namespace>/<image>
```

- 您还可以使用带有类似 **--sign-by-sigstore** 选项的 **skopeo copy** 命令来为现有镜像签名，同时将它们在多个容器注册中心移动。

警告

请注意，您的公共服务器的提交包含有关公钥的数据和有关签名的元数据。

验证

- 使用以下方法之一验证容器镜像是否已正确签名：
  - 使用 **cosign verify** 命令：

```
$ cosign verify <registry>/<namespace>/<image> --key myKey.pub
```

- 使用 **podman pull** 命令：
  - 在 **/etc/containers/policy.json** 文件中添加 **rekorPublicKeyPath** 或 **rekorPublicKeyData** 字段：

```
{
  ...
```

```
 "transports": {
"docker": {
      "<registry>/<namespace>": [
         {
            "type": "sigstoreSigned",
            "rekorPublicKeyPath": "/path/to/local/public/key/file",
         }
      ]
...
}
 }
 ...
}
```

- 拉取镜像：

```
$ podman pull <registry>/<namespace>/<image>
```

  - **podman pull** 命令会根据配置强制实施签名存在，不需要额外的选项。

其他资源

- **podman-push** 手册页
- **podman-build** 手册页
- **container-registries.d** 手册页
- Sigstore：对软件供应链信任和安全的公开回答

第 9 章 管理容器网络

本章提供了有关如何在容器间通信的信息。

9.1. 列出容器网络

在 Podman 中，有两个网络行为 - rootless 和 rootful：

- rootless 网络 - 网络会自动设置，容器没有 IP 地址。
- Rootful 网络 - 容器有一个 IP 地址。

先决条件

- **container-tools** 元数据包已安装。

流程

- 以 root 用户身份列出所有网络：

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
```

  - 默认情况下，Podman 提供了一个桥接网络。
  - Rootful 用户的网络列表与无根用户相同。

其他资源

- **podman-network-ls** man page

9.2. 检查网络

对于 **podman network ls** 命令列出的指定网络，显示启用的插件、启用的插件、网络类型等。

先决条件

- **container-tools** 元数据包已安装。

流程

- 检查默认 **podman** 网络：

```
$ podman network inspect podman
[
   {
      "cniVersion": "0.4.0",
      "name": "podman",
      "plugins": [
         {
            "bridge": "cni-podman0",
            "hairpinMode": true,
```

```
            "ipMasq": true,
            "ipam": {
               "ranges": [
                  [
                     {
                        "gateway": "10.88.0.1",
                        "subnet": "10.88.0.0/16"
                     }
                  ]
               ],
               "routes": [
                  {
                     "dst": "0.0.0.0/0"
                  }
               ],
               "type": "host-local"
            },
            "isGateway": true,
            "type": "bridge"
         },
         {
            "capabilities": {
               "portMappings": true
            },
            "type": "portmap"
         },
         {
            "type": "firewall"
         },
         {
            "type": "tuning"
         }
      ]
   }
]
```

您可以看到 IP 范围、已启用插件、网络类型和其他网络设置。

其他资源

- **podman-network-inspect** man page

9.3. 创建网络

使用 **podman network create** 命令创建新网络。

注意

默认情况下，Podman 创建一个外部网络。您可以使用 **podman network create --internal** 命令创建内部网络。内部网络中的容器可以和主机上的其他容器通信，但无法连接主机之外的网络，也无法从主机访问。

先决条件

- **container-tools** 元数据包已安装。

流程

- 创建名为 **mynet** 的外部网络：

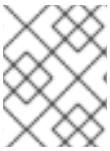
```
# podman network create mynet
/etc/cni/net.d/mynet.conflist
```

验证

- 列出所有网络：

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
11c844f95e28  mynet       0.4.0       bridge,portmap,firewall,tuning,dnsname
```

您可以看到创建的 **mynet** 网络和默认 **podman** 网络。



注意

从 Podman 4.0 开始，如果使用 **podman network create** 命令创建一个新的外部网络，则默认启用 DNS 插件。

其他资源

- **podman-network-create** man page

9.4. 将容器连接到网络

使用 **podman network connect** 命令，将容器连接到网络。

先决条件

- **container-tools** 元数据包已安装。
- 使用 **podman network create** 命令创建了一个网络。
- 容器已创建。

流程

- 将名为 **mycontainer** 的容器连接到名为 **mynet** 的网络：

```
# podman network connect mynet mycontainer
```

验证

- 验证 **mycontainer** 是否已连接到 **mynet** 网络：

```
# podman inspect --format='{{.NetworkSettings.Networks}}' mycontainer
map[podman:0xc00042ab40 mynet:0xc00042ac60]
```

您可以看到 **mycontainer** 已连接到 **mynet** 和 **podman** 网络。

其他资源

- **podman-network-connect** man page

9.5. 断开容器与网络的连接

使用 **podman network disconnect** 命令，断开容器与网络的连接。

先决条件

- **container-tools** 元数据包已安装。
- 使用 **podman network create** 命令创建了一个网络。
- 容器连接到网络。

流程

- 将名为 **mycontainer** 的容器与名为 **mynet** 的网络的连接：

```
# podman network disconnect mynet mycontainer
```

验证

- 验证 **mycontainer** 是否从 **mynet** 网络断开连接：

```
# podman inspect --format='{{.NetworkSettings.Networks}}' mycontainer
map[podman:0xc000537440]
```

  您可以看到 **mycontainer** 从 **mynet** 网络断开连接，**mycontainer** 仅连接到默认的 **podman** 网络。

其他资源

- **podman-network-disconnect** man page

9.6. 删除网络

使用 **podman network rm** 命令删除指定的网络。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 列出所有网络：

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
11c844f95e28  mynet       0.4.0       bridge,portmap,firewall,tuning,dnsname
```

2. 删除 **mynet** 网络：

```
# podman network rm mynet
mynet
```

注意

如果移除的网络关联有关联的容器，则必须使用 **podman network rm -f** 命令删除容器和 pod。

验证

- 检查 **mynet** 网络是否已删除：

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
```

其他资源

- **podman-network-rm** man page

9.7. 删除所有未使用的网络

使用 **podman network prune** 删除所有未使用的网络。未使用的网络是一个网络，没有连接它的容器。**podman network prune** 命令不会删除默认的 **podman** 网络。

先决条件

- **container-tools** 元数据包已安装。

流程

- 删除所有未使用的网络：

```
# podman network prune
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N] y
```

验证

- 验证所有网络已被删除：

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
```

其他资源

- **podman-network-prune** man page

第 10 章 使用 POD

容器是您可以使用 Podman、Skopeo 和 Buildah 容器工具管理的最小单元。Podman pod 是一个或多个容器的组。Pod 概念是由 Kubernetes 引入的。podman pod 与 Kubernetes 定义类似。Pod 是您可以在 OpenShift 或 Kubernetes 环境中创建、部署和管理的最小计算单元。每个 Podman pod 都包括一个 infra 容器。此容器包含与 pod 关联的命名空间，允许 Podman 将其他容器连接到 pod。它允许您在 pod 中启动和停止容器，pod 将保持运行。**registry.access.redhat.com/ubi9/pause** 镜像上的默认 infra 容器。

10.1. 创建 POD

您可以创建具有一个容器的 pod。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建空 pod:

```
$ podman pod create --name mypod
223df6b390b4ea87a090a4b5207f7b9b003187a6960bd37631ae9bc12c433aff
The pod is in the initial state Created.
```

   pod 处于 Created 的初始状态。

2. 可选：列出所有 pod:

```
$ podman pod ps
POD ID        NAME    STATUS    CREATED               # OF CONTAINERS   INFRA ID
223df6b390b4  mypod   Created   Less than a second ago  1                3afdcd93de3e
```

   请注意，pod 有一个容器。

3. 可选：列出与其关联的所有 pod 和容器：

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                 COMMAND  CREATED              STATUS   PORTS
NAMES              POD
3afdcd93de3e  registry.access.redhat.com/ubi9/pause          Less than a second ago
Created        223df6b390b4-infra  223df6b390b4
```

   您可以看到 **podman ps** 命令中的 pod ID 与 **podman pod ps** 命令中的 pod ID 匹配。默认 infra 容器基于 **registry.access.redhat.com/ubi9/pause** 镜像。

4. 在名为 **mypod** 的现有 pod 中运行名为 **myubi** 的容器：

```
$ podman run -dt --name myubi --pod mypod registry.access.redhat.com/ubi9/ubi
/bin/bash
5df5c48fea87860cf75822ceab8370548b04c78be9fc156570949013863ccf71
```

5. 可选：列出所有 pod:

```
$ podman pod ps
POD ID        NAME    STATUS    CREATED               # OF CONTAINERS   INFRA ID
223df6b390b4  mypod   Running   Less than a second ago  2               3afdcd93de3e
```

您可以看到 pod 有两个容器。

6. 可选：列出与其关联的所有 pod 和容器：

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                 COMMAND    CREATED
STATUS                  PORTS  NAMES             POD
5df5c48fea87  registry.access.redhat.com/ubi9/ubi:latest  /bin/bash  Less than a second ago
Up Less than a second ago         myubi             223df6b390b4
3afdcd93de3e  registry.access.redhat.com/ubi9/pause                    Less than a
second ago  Up Less than a second ago        223df6b390b4-infra  223df6b390b4
```

其他资源

- **podman-pod-create** 手册页
- Podman：在本地容器运行时中管理 pod 和容器

10.2. 显示 POD 信息

了解如何显示 pod 信息。

先决条件

- **container-tools** 元数据包已安装。
- pod 已创建。详情请参阅 创建 pod 一节。

流程

- 显示在 pod 中运行的活跃进程：
  - 要显示 pod 中容器的运行进程，请输入：

```
$ podman pod top mypod
USER   PID   PPID   %CPU    ELAPSED         TTY     TIME   COMMAND
0      1     0      0.000   24.077433518s   ?       0s     /pause
root   1     0      0.000   24.078146025s   pts/0   0s     /bin/bash
```

  - 要显示一个或多个 pod 中容器的资源使用情况统计的实时流，请输入：

```
$ podman pod stats -a --no-stream
ID            NAME             CPU %   MEM USAGE / LIMIT   MEM %   NET IO    BLOCK IO
PIDS
a9f807ffaacd  frosty_hodgkin   --      3.092MB / 16.7GB    0.02%   -- / --   -- / --   2
3b33001239ee  sleepy_stallman  --      -- / --             --      -- / --   -- / --   --
```

  - 要显示描述 pod 的信息，请输入：

```
$ podman pod inspect mypod
{
   "Id": "db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5b19a",
   "Name": "mypod",
   "Created": "2020-09-08T10:35:07.536541534+02:00",
   "CreateCommand": [
      "podman",
      "pod",
      "create",
      "--name",
      "mypod"
   ],
   "State": "Running",
   "Hostname": "mypod",
   "CreateCgroup": false,
   "CgroupParent": "/libpod_parent",
   "CgroupPath":
"/libpod_parent/db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5
b19a",
   "CreateInfra": false,
   "InfraContainerID":
"891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
   "SharedNamespaces": [
      "uts",
      "ipc",
      "net"
   ],
   "NumContainers": 2,
   "Containers": [
      {
         "Id":
"891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
         "Name": "db99446fa9c6-infra",
         "State": "running"
      },
      {
         "Id":
"effc5bbcfe505b522e3bf8fbb5705a39f94a455a66fd81e542bcc27d39727d2d",
         "Name": "myubi",
         "State": "running"
      }
   ]
}
```

您可以查看 pod 中容器的信息。

其他资源

- **podman pod 顶部** 手册页
- **podman-pod-stats** 手册页
- **podman-pod-inspect** 手册页

10.3. 停止 POD

您可以使用 **podman pod stop** 命令来停止一个或多个 pod。

先决条件

- **container-tools** 元数据包已安装。
- pod 已创建。详情请参阅 创建 pod 一节。

流程

1. 停止 pod **mypod**:

```
$ podman pod stop mypod
```

2. 可选：列出与其关联的所有 pod 和容器：

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                COMMAND    CREATED            STATUS
PORTS   NAMES              POD ID        PODNAME
5df5c48fea87  registry.redhat.io/ubi9/ubi:latest  /bin/bash  About a minute ago  Exited (0) 7
seconds ago          myubi              223df6b390b4  mypod

3afdcd93de3e  registry.access.redhat.com/9/pause                      About a minute ago
Exited (0) 7 seconds ago          8a4e6527ac9d-infra  223df6b390b4  mypod
```

您可以看到 pod **mypod** 和容器 **myubi** 都处于"Exited"状态。

其他资源

- **podman-pod-stop** 手册页

10.4. 删除 POD

您可以使用 **podman pod rm** 命令删除一个或多个已停止的 pod 和容器。

先决条件

- **container-tools** 元数据包已安装。
- pod 已创建。详情请参阅 创建 pod 一节。
- pod 已停止。详情请参阅停止 pod 一节。

流程

1. 删除 pod **mypod**，请输入：

```
$ podman pod rm mypod
223df6b390b4ea87a090a4b5207f7b9b003187a6960bd37631ae9bc12c433aff
```

请注意，删除 pod 会自动移除其中的所有容器。

2. 可选：检查所有容器和 pod 是否已移除：

```
$ podman ps
$ podman pod ps
```

其他资源

- **podman-pod-rm** 手册页

第 11 章 容器间的通信

了解利用端口映射、DNS 解析或编排 pod 内通信的信息，在容器、应用程序和主机系统之间建立通信。

11.1. 网络模式和层

Podman 中有几个不同的网络模式：

- **bridge** - 在默认网桥网络上创建另一个网络
- **container:<id>** - 使用与容器相同的网络 **<id>** id
- **host** - 使用主机网络堆栈
- **network-id** - 使用一个用户定义的、由 **podman** network create 命令创建的网络
- **private** - 为容器创建一个新网络
- **slirp4nets** - 创建一个用户网络堆栈 slirp4netns，即 rootless 容器的默认选项
- **pasta** - 对 iwl4netns 的高性能替换。您可以使用从 Podman v4.4.1 开始的 **pasta**。
- **None** - 为容器创建网络命名空间，但不为其配置网络接口。容器没有网络连接。
- **ns:<path** > - 要加入的网络命名空间的路径

注意

主机模式使容器可以完全访问本地系统服务，如 D-bus、一个用于进程间通信的系统 (IPC)，因此被视为不安全。

11.2. IWL4NETNS 和 PASTA 之间的区别

与 iwl **4netns** 相比，**pasta** 网络模式的显著区别包括：

- **pasta** 支持 IPv6 端口转发。
- **pasta** 比 **netns4netns** 更高效。
- **pasta** 从主机复制 IP 地址，而 iwl4netns 使用预定义的 IPv4 地址。
- **pasta** 使用主机中的接口名称，而 iwl4netns 使用 tap0 作为接口名称。
- **pasta** 使用主机中的网关地址，而 iwl **4netns** 定义自己的网关地址并使用 NAT。

注意

rootless 容器的默认网络模式是 iwl **4netns**。

11.3. 设置网络模式

其他资源

您可以使用 **podman run** 命令和 **--network** 选项来选择网络模式。

先决条件

- **container-tools** 模块已安装。

流程

1. 可选：如果要使用 **pasta** 网络模式，请安装 **passt** 软件包：

```
$ {PackageManager} install passt
```

2. 运行基于 **registry.access.redhat.com/ubi9/ubi** 镜像的容器：

```
$ podman run --network=<netwok_mode> -d --name=myubi
registry.access.redhat.com/ubi9/ubi
```

& ***lt;netwok_mode&*** gt; 是所需的网络模式。另外，您可以使用 **containers.conf** 文件中的 **default_rootless_network_cmd** 选项切换默认的网络模式。

注意

rootless 容器的默认网络模式是 iwl **4netns**。

验证

- 验证网络模式的设置：

```
$ podman inspect --format {{.HostConfig.NetworkMode}} myubi
<netwok_mode>
```

11.4. 检查容器的网络设置

其他资源

使用 **podman inspect** 命令和 **--format** 选项来显示 **podman inspect** 输出中的各个项目。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 显示容器的 IP 地址：

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' <containerName>
```

2. 显示容器连接到的所有网络：

```
# podman inspect --format='{{.NetworkSettings.Networks}}' <containerName>
```

3. 显示端口映射：

```
# podman inspect --format='{{.NetworkSettings.Ports}}' <containerName>
```

其他资源

- **podman-inspect** 手册页

11.5. 容器和应用程序间的通信

您可以在容器和应用程序之间进行通信。应用程序端口处于 listening 或 open 状态。这些端口会自动公开给容器网络，因此您可以使用这些网络来访问这些容器。默认情况下，Web 服务器侦听端口 80。使用这个流程，**myubi** 容器与 **web-container** 应用通信。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 启动名为 **web-container** 的容器：

```
# podman run -dt --name=web-container docker.io/library/httpd
```

2. 列出所有容器：

```
# podman ps -a

CONTAINER ID  IMAGE                           COMMAND         CREATED        STATUS
PORTS       NAMES
b8c057333513  docker.io/library/httpd:latest  httpd-foreground  4 seconds ago  Up 5 seconds
ago             web-container
```

3. 检查容器并显示 IP 地址：

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web-container

10.88.0.2
```

4. 运行 **myubi** 容器并验证 web 服务器是否正在运行：

```
# podman run -it --name=myubi ubi9/ubi curl 10.88.0.2:80

<html><body><h1>It works!</h1></body></html>
```

11.6. 容器和主机间的通信

默认情况下，**podman** 网络是一个桥接网络。这意味着网络设备将容器网络桥接到主机网络。

先决条件

- **container-tools** 元数据包已安装。
- **web-container** 正在运行。如需更多信息，请参阅容器和应用程序之间的沟通部分。

流程

1. 验证网桥是否已配置：

```
# podman network inspect podman | grep bridge

    "bridge": "cni-podman0",
    "type": "bridge"
```

2. 显示主机网络配置：

```
# ip addr show cni-podman0

6: cni-podman0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default qlen 1000
    link/ether 62:af:a1:0a:ca:2e brd ff:ff:ff:ff:ff:ff
    inet 10.88.0.1/16 brd 10.88.255.255 scope global cni-podman0
       valid_lft forever preferred_lft forever
    inet6 fe80::60af:a1ff:fe0a:ca2e/64 scope link
       valid_lft forever preferred_lft forever
```

   您可以看到 **web-container** 的 IP 地址为 **cni-podman0** 网络，网络则桥接到主机。

3. 检查 **web-container** 并显示其 IP 地址：

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web-container

10.88.0.2
```

4. 直接从主机访问 **web-container**:

```
$ curl 10.88.0.2:80

<html><body><h1>It works!</h1></body></html>
```

其他资源

- **podman-network** man page

11.7. 使用端口映射在容器间通信

在两个容器之间进行通信的最方便方法是使用公布的端口。可使用两种方式发布端口：自动或手动发布。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 运行未发布的容器：

```
# podman run -dt --name=web1 ubi9/httpd-24
```

2. 运行自动发布的容器：

```
# podman run -dt --name=web2 -P ubi9/httpd-24
```

3. 运行手动发布的容器并发布容器端口 80：

```
# podman run -dt --name=web3 -p 9090:80 ubi9/httpd-24
```

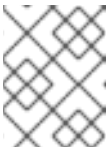
4. 列出所有容器：

```
# podman ps

CONTAINER ID  IMAGE                                    COMMAND           CREATED
STATUS            PORTS                                NAMES
f12fa79b8b39  registry.access.redhat.com/ubi9/httpd-24:latest  /usr/bin/run-http...  23
seconds ago  Up 24 seconds ago                                    web1
9024d9e815e2  registry.access.redhat.com/ubi9/httpd-24:latest  /usr/bin/run-http...  13
seconds ago  Up 13 seconds ago  0.0.0.0:43595->8080/tcp, 0.0.0.0:42423->8443/tcp  web2
03bc2a019f1b  registry.access.redhat.com/ubi9/httpd-24:latest  /usr/bin/run-http...  2 seconds
ago   Up 2 seconds ago   0.0.0.0:9090->80/tcp                    web3
```

您可以看到：

- 容器 **web1** 没有公布的端口，只能通过容器网络或网桥访问。
- 容器 **web2** 已自动映射端口 43595 和 42423，以分别发布应用端口 8080 和 8443。



注意

可能会出现自动端口映射，因为 **registry.access.redhat.com/9/httpd-24** 镜像在 Containerfile 中具有 **EXPOSE 8080** 和 **EXPOSE 8443** 命令。

- 容器 **web3** 有一个手动发布的端口。主机端口 9090 映射到容器端口 80。

5. 显示 **web1** 和 **web3** 容器的 IP 地址：

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web1
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web3
```

6. 使用 <IP>:<port> 表示法访问 **web1** 容器：

```
# curl 10.88.0.14:8080
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

7. 使用 localhost:<port> 表示法访问 **web2** 容器：

```
# curl localhost:43595
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

8. 使用 <IP>:<port> 表示法访问 **web3** 容器：

```
# curl 10.88.0.14:9090
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

11.8. 使用 DNS 在容器间通信

启用 DNS 插件时，请使用容器名称来处理容器。

先决条件

- **container-tools** 元数据包已安装。
- 使用 **podman network create** 命令创建带有启用 DNS 插件的网络。

流程

1. 运行附加到 **mynet** 网络的 **接收器** 容器：

```
# podman run -d --net mynet --name receiver ubi9 sleep 3000
```

2. 运行 发**送者** 容器并通过其名称访问 **接收器（receiver）**容器：

```
# podman run -it --rm --net mynet --name sender alpine ping receiver

PING rcv01 (10.89.0.2): 56 data bytes
64 bytes from 10.89.0.2: seq=0 ttl=42 time=0.041 ms
64 bytes from 10.89.0.2: seq=1 ttl=42 time=0.125 ms
64 bytes from 10.89.0.2: seq=2 ttl=42 time=0.109 ms
```

使用 **CTRL+C** 退出。

您可以看到，发**送者（sender）** 容器可以使用其名称 ping **接收器** 容器。

11.9. 在 POD 中的两个容器间通信

同一 pod 中的所有容器共享 IP 地址、MAC 地址和端口映射。您可以使用 localhost:port 表示法在同一 pod 中的容器之间进行通信。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建名为 **web-pod** 的 pod：

```
$ podman pod create --name=web-pod
```

2. 在 pod 中运行名为 **web-container** 的 web 容器：

```
$ podman container run -d --pod web-pod --name=web-container
docker.io/library/httpd
```

3. 列出与其关联的所有 pod 和容器：

```
$ podman ps --pod

CONTAINER ID  IMAGE                           COMMAND           CREATED         STATUS
PORTS       NAMES               POD ID        PODNAME
58653cf0cf09  k8s.gcr.io/pause:3.5                              4 minutes ago   Up 3 minutes ago
4e61a300c194-infra  4e61a300c194  web-pod
b3f4255afdb3  docker.io/library/httpd:latest  httpd-foreground  3 minutes ago   Up 3 minutes
ago               web-container  4e61a300c194  web-pod
```

4. 根据 docker.io/library/fedora 镜像在 **web-pod** 中运行容器：

```
$ podman container run -it --rm --pod web-pod docker.io/library/fedora curl localhost

<html><body><h1>It works!</h1></body></html>
```

您可以看到容器可以访问 **web-container**。

11.10. POD 的通信

在创建 pod 时，您必须在 pod 中发布容器的端口。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建名为 **web-pod** 的 pod：

```
# podman pod create --name=web-pod-publish -p 80:80
```

2. 列出所有 pod:

```
# podman pod ls

POD ID        NAME         STATUS   CREATED         INFRA ID      # OF CONTAINERS
26fe5de43ab3  publish-pod  Created  5 seconds ago  7de09076d2b3  1
```

3. 在 **web-pod** 中运行名为 **web-container** 的 web 容器：

```
# podman container run -d --pod web-pod-publish --name=web-container
docker.io/library/httpd
```

4. 列出容器

```
# podman ps
```

```
CONTAINER ID  IMAGE                 COMMAND          CREATED            STATUS
PORTS               NAMES
7de09076d2b3  k8s.gcr.io/pause:3.5                    About a minute ago  Up 23 seconds ago
0.0.0.0:80->80/tcp  26fe5de43ab3-infra
088befb90e59  docker.io/library/httpd  httpd-foreground  23 seconds ago      Up 23 seconds
ago  0.0.0.0:80->80/tcp  web-container
```

5. 验证可以访问 **web-container**:

```
$ curl localhost:80

<html><body><h1>It works!</h1></body></html>
```

11.11. 将 POD 附加到容器网络

在创建 pod 期间，将 pod 中的容器附加到网络。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建名为 **pod-net** 的网络：

```
# podman network create pod-net

/etc/cni/net.d/pod-net.conflist
```

2. 创建 pod **web-pod** ：

```
# podman pod create --net pod-net --name web-pod
```

3. 在 **web-pod** 中运行一个名为 **web-container** 的容器：

```
# podman run -d --pod webt-pod --name=web-container docker.io/library/httpd
```

4. 可选：显示容器与之关联的 pod：

```
# podman ps -p

CONTAINER ID  IMAGE                                  COMMAND          CREATED        STATUS
PORTS       NAMES              POD ID        PODNAME
b7d6871d018c  registry.access.redhat.com/ubi9/pause:latest                          9 minutes
ago  Up 6 minutes ago             a8e7360326ba-infra  a8e7360326ba  web-pod
645835585e24  docker.io/library/httpd:latest  httpd-foreground  6 minutes ago  Up 6 minutes
ago               web-container    a8e7360326ba  web-pod
```

验证

- 显示连接到容器的所有网络：

```
# podman ps --format="{{.Networks}}"
pod-net
```

第 12 章 设置容器网络模式

本章介绍了如何设置不同的网络模式的信息。

12.1. 使用静态 IP 运行容器

使用 **--ip** 选项的 **podman run** 命令将容器网络接口设置为特定的 IP 地址（如 10.88.0.44）。要验证您是否正确设置了 IP 地址，请运行 **podman inspect** 命令。

先决条件

- **container-tools** 元数据包已安装。

流程

- 将容器网络接口设为 IP 地址 10.88.0.44：

```
# podman run -d --name=myubi --ip=10.88.0.44 registry.access.redhat.com/ubi9/ubi
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

验证

- 检查是否正确设置了 IP 地址：

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' myubi
10.88.0.44
```

12.2. 在没有 SYSTEMD 的情况下运行 DHCP 插件

使用 **podman run --network** 命令连接到用户定义的网络。虽然大多数容器镜像都没有 DHCP 客户端，但 **dhcp** 插件充当容器的代理 DHCP 客户端，以便容器与 DHCP 服务器交互。

注意

此流程只适用于 rootfull 容器。Rootless 容器不使用 **dhcp** 插件。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 手动运行 **dhcp** 插件：

```
# /usr/libexec/cni/dhcp daemon &
[1] 4966
```

2. 检查 **dhcp** 插件是否正在运行：

```
# ps -a | grep dhcp
4966 pts/1    00:00:00 dhcp
```

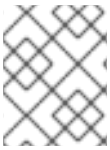
3. 运行 **alpine** 容器：

```
# podman run -it --rm --network=example alpine ip addr show enp1s0
Resolved "alpine" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/alpine:latest...
...
Storing signatures

2: eth0@eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP
    link/ether f6:dd:1b:a7:9b:92 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.22/24 brd 192.168.1.255 scope global eth0
    ...
```

在本例中：

- **--network=example** 选项指定要连接的名为 example 的网络。
- **alpine** 容器内的 **ip addr show enp1s0** 命令检查网络接口 **enp1s0** 的 IP 地址。
- 主机网络是 192.168.1.0/24
- **eth0** 接口为 alpine 容器租期 IP 地址 192.168.1.122。



注意

如果您有大量短期容器和一个具有长租用的 DHCP 服务器，此配置可能会耗尽可用的 DHCP 地址。

其他资源

- 使用 Podman 容器读取可路由 IP 地址

12.3. 使用 SYSTEMD 运行 DHCP 插件

您可以使用 **systemd** 单元文件来运行 **dhcp** 插件。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建套接字单元文件：

```
# cat /usr/lib/systemd/system/io.podman.dhcp.socket
[Unit]
Description=DHCP Client for CNI

[Socket]
ListenStream=%t/cni/dhcp.sock
SocketMode=0600
```

```
[Install]
WantedBy=sockets.target
```

2. 创建服务单元文件：

```
# cat /usr/lib/systemd/system/io.podman.dhcp.service
[Unit]
Description=DHCP Client CNI Service
Requires=io.podman.dhcp.socket
After=io.podman.dhcp.socket

[Service]
Type=simple
ExecStart=/usr/libexec/cni/dhcp daemon
TimeoutStopSec=30
KillMode=process


[Install]
WantedBy=multi-user.target
Also=io.podman.dhcp.socket
```

3. 立即启动该服务：

```
# systemctl --now enable io.podman.dhcp.socket
```

验证

- 检查套接字的状态：

```
# systemctl status io.podman.dhcp.socket
io.podman.dhcp.socket - DHCP Client for CNI
Loaded: loaded (/usr/lib/systemd/system/io.podman.dhcp.socket; enabled; vendor preset:
disabled)
Active: active (listening) since Mon 2022-01-03 18:08:10 CET; 39s ago
Listen: /run/cni/dhcp.sock (Stream)
CGroup: /system.slice/io.podman.dhcp.socket
```

其他资源

- 使用 Podman 容器读取可路由 IP 地址

12.4. MACVLAN 插件

大多数容器镜像都没有 DHCP 客户端，但 **dhcp** 插件充当容器的代理 DHCP 客户端，以便容器与 DHCP 服务器交互。

主机系统没有容器的网络访问权限。要允许主机外部到容器的网络连接，容器必须与主机在同一网络上具有 IP。**macvlan** 插件允许您将容器连接到与主机相同的网络。

注意

此流程只适用于 rootfull 容器。Rootless 容器无法使用 **macvlan** 和 **dhcp** 插件。

注意

您可以使用 **podman network create --macvlan** 命令创建 macvlan 网络。

其他资源

- 使用 Podman 容器读取可路由 IP 地址
- **podman-network-create** man page

12.5. 将网络堆栈从 CNI 切换到 NETAVARK

在以前的版本中，只有连接到单个 Container Network Interface(CNI)插件时，容器才可以使用 DNS。Netavark 是容器的网络堆栈。您可以将 Netavark 与 Podman 和其他开放容器项目(OCI)容器管理应用程序一起使用。Podman 的高级网络堆栈与高级 Docker 功能兼容。现在，多个网络中的容器访问这些网络上的容器。

Netavark 能够执行以下操作：

- 创建、管理和移除网络接口，包括网桥和 MACVLAN 接口。
- 配置防火墙设置，如网络地址转换(NAT)和端口映射规则。
- 支持 IPv4 和 IPv6。
- 改进对多个网络中容器的支持。

警告

CNI 网络堆栈已弃用，并将在 Podman v5.0 中删除。使用 Netavark 网络堆栈替代。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 如果 **/etc/containers/containers.conf** 文件不存在，请将 **/usr/share/containers/containers.conf** 文件复制到 **/etc/containers/** 目录中：

```
# cp /usr/share/containers/containers.conf /etc/containers/
```

2. 编辑 **/etc/containers/containers.conf** 文件，并在 **[network]** 部分中添加以下内容：

```
network_backend="netavark"
```

3. 如果您有任何容器或 pod，请将存储重置回初始状态：

```
# podman system reset
```

4. 重启系统：

```
# reboot
```

验证

- 验证网络堆栈是否已更改为 Netavark：

```
# cat /etc/containers/containers.conf
...
[network]
network_backend="netavark"
...
```

注意

如果使用 Podman 4.0.0 或更高版本，请使用 **podman info** 命令检查网络堆栈设置。

其他资源

- podman 4.0 的新网络堆栈：您需要了解的内容
- **podman-system-reset** man page

12.6. 将网络堆栈从 NETAVARK 切换到 CNI

您可以将网络堆栈从 Netavark 切换到 CNI。

警告

CNI 网络堆栈已弃用，并将在 Podman v5.0 中删除。使用 Netavark 网络堆栈替代。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 如果 **/etc/containers/containers.conf** 文件不存在，请将 **/usr/share/containers/containers.conf** 文件复制到 **/etc/containers/** 目录中：

```
# cp /usr/share/containers/containers.conf /etc/containers/
```

2. 编辑 **/etc/containers/containers.conf** 文件，并在 **[network]** 部分中添加以下内容：

```
network_backend="cni"
```

3. 如果您有任何容器或 pod，请将存储重置回初始状态：

```
# podman system reset
```

4. 重启系统：

```
# reboot
```

验证

- 验证网络堆栈是否已更改为 CNI：

```
# cat /etc/containers/containers.conf
...
[network]
network_backend="cni"
...
```

注意

如果使用 Podman 4.0.0 或更高版本，请使用 **podman info** 命令检查网络堆栈设置。

其他资源

- podman 4.0 的新网络堆栈：您需要了解的内容
- **podman-system-reset** man page

第 13 章 使用 PODMAN 将容器传送到 OPENSHIFT

您可以使用 YAML ("YAML Ain't Markup Language")格式生成容器和 pod 的可移植性描述。YAML 是一种用于描述配置数据的文本格式。

YAML 文件为：

- 可读。
- 易于生成。
- 可在环境间移植（例如，RHEL 和 OpenShift 之间）。
- 可移植编程语言。
- 方便使用（不需要在命令行中添加所有参数）。

使用 YAML 文件的原因：

1. 您可以使用最小输入来重新运行本地编配的容器和 pod，这对于迭代开发非常有用。
2. 您可以在另一台机器上运行相同的容器和 pod。例如，要在 OpenShift 环境中运行应用程序并确保应用程序正常工作。您可以使用 **podman generate kube** 命令来生成 Kubernetes YAML 文件。然后，在将生成的 YAML 文件传送到 Kubernetes 或 OpenShift 环境之前，您可以使用 **podman play** 命令来测试本地系统上 pod 和容器的创建情况。使用 **podman play** 命令，您也可以重新创建最初在 OpenShift 或 Kubernetes 环境中创建的 pod 和容器。

注意

podman kube play 命令支持 Kubernetes YAML 功能的子集。如需更多信息，请参阅 支持的 YAML 字段的支持矩阵。

13.1. 使用 PODMAN 生成 KUBERNETES YAML 文件

您可以创建具有一个容器的 pod，并使用 **podman generate kube** 命令生成 Kubernetes YAML 文件。

先决条件

- **container-tools** 元数据包已安装。
- pod 已创建。详情请参阅 创建 pod 一节。

流程

1. 列出与其关联的所有 pod 和容器：

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                  COMMAND    CREATED
STATUS                  PORTS  NAMES              POD
5df5c48fea87  registry.access.redhat.com/ubi9/ubi:latest  /bin/bash  Less than a second ago
Up Less than a second ago         myubi              223df6b390b4
3afdcd93de3e  k8s.gcr.io/pause:3.1                        Less than a second ago  Up Less
than a second ago         223df6b390b4-infra  223df6b390b4
```

2. 使用 pod 名称或 ID 来生成 Kubernetes YAML 文件：

```
$ podman generate kube mypod > mypod.yaml
```

请注意，**podman generate** 命令不反映任何可能附加到容器的逻辑卷管理器(LVM)逻辑卷或物理卷。

3. 显示 **mypod.yaml** 文件：

```
$ cat mypod.yaml
# Generation of Kubernetes YAML is still under development!
#
# Save the output of this file and use kubectl create -f to import
# it into Kubernetes.
#
# Created with podman-1.6.4
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-06-09T10:31:56Z"
  labels:
app: mypod
  name: mypod
spec:
  containers:
  - command:
        - /bin/bash
        env:
        - name: PATH
            value: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
        - name: TERM
            value: xterm
        - name: HOSTNAME
        - name: container
            value: oci
        image: registry.access.redhat.com/ubi9/ubi:latest
        name: myubi
        resources: {}
        securityContext:
            allowPrivilegeEscalation: true
            capabilities: {}
            privileged: false
            readOnlyRootFilesystem: false
        tty: true
        workingDir: /
status: {}
```

其他资源

- **podman-generate-kube** man page
- Podman：在本地容器运行时中管理 pod 和容器

13.2. 在 OPENSHIFT 环境中生成 KUBERNETES YAML 文件

在 OpenShift 环境中，使用 **oc create** 命令来生成描述应用程序的 YAML 文件。

流程

- 为您的 **myapp** 应用程序生成 YAML 文件：

```
$ oc create myapp --image=me/myapp:v1 -o yaml --dry-run > myapp.yaml
```

oc create 命令创建并运行 **myapp** 镜像。使用 **--dry-run** 选项打印对象，并重定向到 **myapp.yaml** 输出文件。

注意

在 Kubernetes 环境中，您可以使用带有同样标志的 **kubectl create** 命令。

13.3. 使用 PODMAN 启动容器和 POD

使用生成的 YAML 文件，您可以在任何环境中自动启动容器和 pod。可以使用 Podman 以外的工具（如 Kubernetes 或 Openshift）生成 YAML 文件。**podman play kube** 命令允许您根据 YAML 输入文件重新创建 pod 和容器。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 从 **mypod.yaml** 文件创建 pod 和容器：

```
$ podman play kube mypod.yaml
Pod:
b8c5b99ba846ccff76c3ef257e5761c2d8a5ca4d7ffa3880531aec79c0dacb22
Container:
848179395ebd33dd91d14ffbde7ae273158d9695a081468f487af4e356888ece
```

2. 列出所有 pod:

```
$ podman pod ps
POD ID        NAME    STATUS    CREATED         # OF CONTAINERS   INFRA ID
b8c5b99ba846  mypod   Running   19 seconds ago  2                 aa4220eaf4bb
```

3. 列出与其关联的所有 pod 和容器：

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                  COMMAND    CREATED            STATUS
PORTS  NAMES              POD
848179395ebd  registry.access.redhat.com/ubi9/ubi:latest  /bin/bash  About a minute ago  Up
About a minute ago        myubi             b8c5b99ba846
aa4220eaf4bb  k8s.gcr.io/pause:3.1                        About a minute ago  Up About a
minute ago        b8c5b99ba846-infra  b8c5b99ba846
```

podman ps 命令中的 pod ID 与 **podman pod ps** 命令中的 pod ID 相匹配。

其他资源

- **podman-play-kube** man page
- Podman 现在可以简化到 Kubernetes 和 CRI-O 的转换

13.4. 在 OPENSHIFT 环境中启动容器和 POD

您可以使用 **oc create** 命令在 OpenShift 环境中创建 pod 和容器。

流程

- 从 OpenShift 环境中的 YAML 文件创建 pod:

```
$ oc create -f mypod.yaml
```

注意

在 Kubernetes 环境中，您可以使用带有同样标志的 **kubectl create** 命令。

13.5. 使用 PODMAN 手动运行容器和 POD

以下步骤演示了如何使用 Podman 手动创建与 MariaDB 数据库配对的 WordPress 内容管理系统。

假设以下目录布局：

```
├── mariadb-conf
│   ├── Containerfile
│   ├── my.cnf
```

先决条件

- **container-tools** 元数据包已安装。

流程

1. 显示 **mariadb-conf/Containerfile** 文件：

```
$ cat mariadb-conf/Containerfile
FROM docker.io/library/mariadb
COPY my.cnf /etc/mysql/my.cnf
```

2. 显示 **mariadb-conf/my.cnf** 文件：

```
[client-server]
# Port or socket location where to connect
port = 3306
socket = /run/mysqld/mysqld.sock

# Import all .cnf files from the configuration directory
[mariadbd]
skip-host-cache
skip-name-resolve
bind-address = 127.0.0.1
```

```
!includedir /etc/mysql/mariadb.conf.d/
!includedir /etc/mysql/conf.d/
```

3. 使用 **mariadb-conf/Containerfile** 来构建 **docker.io/library/mariadb** 镜像：

```
$ cd mariadb-conf
$ podman build -t mariadb-conf .
$ cd ..
STEP 1: FROM docker.io/library/mariadb
Trying to pull docker.io/library/mariadb:latest...
Getting image source signatures
Copying blob 7b1a6ab2e44d done
...
Storing signatures
STEP 2: COPY my.cnf /etc/mysql/my.cnf
STEP 3: COMMIT mariadb-conf
--> ffae584aa6e
Successfully tagged localhost/mariadb-conf:latest
ffae584aa6e733ee1cdf89c053337502e1089d1620ff05680b6818a96eec3c17
```

4. 可选：列出所有镜像：

```
$ podman images
LIST IMAGES
REPOSITORY                                  TAG         IMAGE ID      CREATED
SIZE
localhost/mariadb-conf                      latest      b66fa0fa0ef2  57 seconds ago
416 MB
```

5. 创建名为 **wordpresspod** 的 pod，并配置容器和主机系统之间的端口映射：

```
$ podman pod create --name wordpresspod -p 8080:80
```

6. 在 **wordpresspod** pod 中创建 **mydb** 容器：

```
$ podman run --detach --pod wordpresspod \
  -e MYSQL_ROOT_PASSWORD=1234 \
  -e MYSQL_DATABASE=mywpdb \
  -e MYSQL_USER=mywpuser \
  -e MYSQL_PASSWORD=1234 \
  --name mydb localhost/mariadb-conf
```

7. 在 **wordpresspod** pod 中创建 **myweb** 容器：

```
$ podman run --detach --pod wordpresspod \
  -e WORDPRESS_DB_HOST=127.0.0.1 \
  -e WORDPRESS_DB_NAME=mywpdb \
  -e WORDPRESS_DB_USER=mywpuser \
  -e WORDPRESS_DB_PASSWORD=1234 \
  --name myweb docker.io/wordpress
```

8. 可选。列出与其关联的所有 pod 和容器：

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                           COMMAND            CREATED
STATUS                   PORTS               NAMES              POD ID        PODNAME
9ea56f771915  k8s.gcr.io/pause:3.5                               Less than a second ago  Up Less
than a second ago  0.0.0.0:8080->80/tcp  4b7f054a6f01-infra  4b7f054a6f01  wordpresspod
60e8dbbabac5  localhost/mariadb-conf:latest       mariadbd           Less than a second ago
Up Less than a second ago  0.0.0.0:8080->80/tcp  mydb               4b7f054a6f01
wordpresspod
045d3d506e50  docker.io/library/wordpress:latest  apache2-foregroun...  Less than a second
ago  Up Less than a second ago  0.0.0.0:8080->80/tcp  myweb              4b7f054a6f01
wordpresspod
```

验证

- 验证 Pod 是否在运行：访问 http://localhost:8080/wp-admin/install.php 页面或使用 **curl** 命令：

```
$ curl http://localhost:8080/wp-admin/install.php
<!DOCTYPE html>
<html xml:lang="en-US">
<head>
...
</head>
<body class="wp-core-ui">
<p id="logo">WordPress</p>
    <h1>Welcome</h1>
...
```

其他资源

- 使用 Podman play kube 构建 Kubernetes pod
- **podman-play-kube** man page

13.6. 使用 PODMAN 生成 YAML 文件

您可以使用 **podman generate kube** 命令生成 Kubernetes YAML 文件。

先决条件

- **container-tools** 元数据包已安装。
- 名为 **wordpresspod** 的 pod 已创建。详情请参阅 创建 pod 一节。

流程

1. 列出与其关联的所有 pod 和容器：

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                           COMMAND            CREATED
STATUS                   PORTS               NAMES              POD ID        PODNAME
9ea56f771915  k8s.gcr.io/pause:3.5                               Less than a second ago  Up Less
than a second ago  0.0.0.0:8080->80/tcp  4b7f054a6f01-infra  4b7f054a6f01  wordpresspod
60e8dbbabac5  localhost/mariadb-conf:latest       mariadbd           Less than a second ago
```

```
Up Less than a second ago  0.0.0.0:8080->80/tcp  mydb              4b7f054a6f01
wordpresspod
045d3d506e50  docker.io/library/wordpress:latest  apache2-foregroun...  Less than a second
ago  Up Less than a second ago  0.0.0.0:8080->80/tcp  myweb             4b7f054a6f01
wordpresspod
```

2. 使用 pod 名称或 ID 来生成 Kubernetes YAML 文件：

```
$ podman generate kube wordpresspod >> wordpresspod.yaml
```

验证

- 显示 **wordpresspod.yaml** 文件：

```
$ cat wordpresspod.yaml
...
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2021-12-09T15:09:30Z"
  labels:
    app: wordpresspod
  name: wordpresspod
spec:
  containers:
  - args:
        value: podman
      - name: MYSQL_PASSWORD
        value: "1234"
      - name: MYSQL_MAJOR
        value: "8.0"
      - name: MYSQL_VERSION
        value: 8.0.27-1debian10
      - name: MYSQL_ROOT_PASSWORD
        value: "1234"
      - name: MYSQL_DATABASE
        value: mywpdb
      - name: MYSQL_USER
        value: mywpuser
        image: mariadb
            name: mydb
            ports:
            - containerPort: 80
              hostPort: 8080
              protocol: TCP
  - args:
    - name: WORDPRESS_DB_NAME
      value: mywpdb
    - name: WORDPRESS_DB_PASSWORD
      value: "1234"
    - name: WORDPRESS_DB_HOST
      value: 127.0.0.1
    - name: WORDPRESS_DB_USER
```

```
    value: mywpuser
  image: docker.io/library/wordpress:latest
  name: myweb
```

其他资源

- 使用 Podman play kube 构建 Kubernetes pod
- **podman-play-kube** man page

13.7. 使用 PODMAN 自动运行容器和 POD

在将生成的 YAML 文件传送到 Kubernetes 或 OpenShift 环境前，您可以使用 **podman play kube** 命令测试本地系统中的 pod 和容器的创建。

podman play kube 命令还可使用类似 docker compose 命令的 YAML 文件自动构建并运行带有 pod 中多个容器的多个 pod。如果满足以下条件，会自动构建镜像：

1. 存在一个名称与 YAML 文件中所用镜像相同的目录
2. 该目录包含一个 Containerfile

先决条件

- **container-tools** 元数据包已安装。
- 名为 **wordpresspod** 的 pod 已创建。详情请参阅 使用 Podman 手动运行容器和 pod 一节。
- 已生成 YAML 文件。详情请参阅使用 Podman 生成 YAML 文件一节。
- 要从头重复整个场景，请删除本地存储的镜像：

```
$ podman rmi localhost/mariadb-conf
$ podman rmi docker.io/library/wordpress
$ podman rmi docker.io/library/mysql
```

流程

1. 使用 **wordpress.yaml** 文件创建 wordpress pod：

```
$ podman play kube wordpress.yaml
STEP 1/2: FROM docker.io/library/mariadb
STEP 2/2: COPY my.cnf /etc/mysql/my.cnf
COMMIT localhost/mariadb-conf:latest
--> 428832c45d0
Successfully tagged localhost/mariadb-conf:latest
428832c45d07d78bb9cb34e0296a7dc205026c2fe4d636c54912c3d6bab7f399
Trying to pull docker.io/library/wordpress:latest...
Getting image source signatures
Copying blob 99c3c1c4d556 done
...
Storing signatures
Pod:
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
```

```
Containers:
6c59ebe968467d7fdb961c74a175c88cb5257fed7fb3d375c002899ea855ae1f
29717878452ff56299531f79832723d3a620a403f4a996090ea987233df0bc3d
```

podman play kube 命令：

- 根据 **docker.io/library/mariadb** 镜像，自动构建 **localhost/mariadb-conf:latest** 镜像。
- 拉取 **docker.io/library/wordpress:latest** 镜像。
- 创建名为 **wordpresspod** 的 pod，它有两个容器，名为 **wordpresspod-mydb** 和 **wordpresspod-myweb**。

2. 列出所有容器和 pod：

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                                COMMAND               CREATED        STATUS
PORTS                   NAMES               POD ID        PODNAME
a1dbf7b5606c  k8s.gcr.io/pause:3.5                                      3 minutes ago  Up 2 minutes ago
0.0.0.0:8080->80/tcp  3e391d091d19-infra  3e391d091d19  wordpresspod
6c59ebe96846  localhost/mariadb-conf:latest        mariadbd              2 minutes ago  Exited (1)
2 minutes ago  0.0.0.0:8080->80/tcp  wordpresspod-mydb   3e391d091d19  wordpresspod
29717878452f  docker.io/library/wordpress:latest  apache2-foregroun...  2 minutes ago  Up 2
minutes ago          0.0.0.0:8080->80/tcp  wordpresspod-myweb  3e391d091d19
wordpresspod
```

验证

- 验证 Pod 是否在运行：访问 http://localhost:8080/wp-admin/install.php 页面或使用 **curl** 命令：

```
$ curl http://localhost:8080/wp-admin/install.php
<!DOCTYPE html>
<html xml:lang="en-US">
<head>
...
</head>
<body class="wp-core-ui">
<p id="logo">WordPress</p>
    <h1>Welcome</h1>
...
```

其他资源

- 使用 Podman play kube 构建 Kubernetes pod
- **podman-play-kube** man page

13.8. 使用 PODMAN 自动停止和删除 POD

podman play kube --down 命令停止并删除所有 pod 及其容器。

注意

如果使用卷，它不会被删除。

先决条件

- **container-tools** 元数据包已安装。
- 名为 **wordpresspod** 的 pod 已创建。详情请参阅 使用 Podman 手动运行容器和 pod 一节。
- 已生成 YAML 文件。详情请参阅使用 Podman 生成 YAML 文件一节。
- Pod 正在运行。详情请参阅使用 Podman 自动运行容器和 pod 一节。

流程

- 删除 **wordpresspod.yaml** 文件创建的所有 pod 和容器：

```
$ podman play kube --down wordpresspod.yaml
Pods stopped:
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
Pods removed:
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
```

验证

- 验证 **wordpresspod.yaml** 文件创建的所有 pod 和容器已被删除：

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                       COMMAND           CREATED
STATUS                 PORTS             NAMES             POD ID       PODNAME
```

其他资源

- 使用 Podman play kube 构建 Kubernetes pod
- **podman-play-kube** man page

第 14 章 使用 PODMAN 将容器移植到 SYSTEMD

Podman (Pod Manager)是一个简单的无守护进程工具，功能齐全的容器引擎。Podman 提供了一个与 Docker-CLI 类似的命令行，可以轻松地从其他容器引擎进行转换，并能够管理 pod、容器和镜像。

最初，Podman 没有设计为提供整个 Linux 系统或管理服务，如启动顺序、依赖项检查和失败的服务恢复。**systemd** 负责整个系统初始化。由于红帽将容器与 **systemd** 集成，您可以像在 Linux 系统中管理其他服务和功能一样管理 Podman 构建的 OCI 和 Docker 格式的容器。您可以使用 **systemd** 初始化服务来处理 pod 和容器。

使用 **systemd** 单元文件，您可以：

- 设置容器或 pod 以作为 **systemd** 服务启动。
- 定义容器化服务运行的顺序，并检查依赖项（例如，确保另一个服务正在运行、文件可用或已挂载资源）。
- 使用 **systemctl** 命令控制 **systemd** 系统的状态。

您可以使用 **systemd** 单元文件生成容器和 pod 的可移植描述。

14.1. 使用 QUADLETS 自动生成 SYSTEMD 单元文件

使用 Quadlet 时，您描述了如何以与常规 **systemd** 单元文件非常相似的格式运行容器。容器描述侧重于相关的容器详情，隐藏在 **systemd** 下运行容器的技术详情。在以下一个目录中创建 ***<CTRNAME>*.container** 单元文件：

- 对于 root 用户：**/usr/share/containers/systemd/** 或 **/etc/containers/systemd/**
- 对于无根用户：**$HOME/.config/containers/systemd/**, **$XDG_CONFIG_HOME/containers/systemd/**, **/etc/containers/systemd/users/$(UID)** 或 **/etc/containers/systemd/users/**

注意

Quadlet 从 Podman v4.6 开始提供。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建 **mysleep.container** 单元文件：

```
$ cat $HOME/.config/containers/systemd/mysleep.container
[Unit]
Description=The sleep container
After=local-fs.target

[Container]
Image=registry.access.redhat.com/ubi9-minimal:latest
Exec=sleep 1000
```

```
[Install]
# Start by default on boot
WantedBy=multi-user.target default.target
```

在 **[Container]** 部分中，您必须指定：

- **Image** - 您要运行的容器镜像
- **exec** - 您要在容器内运行的命令
  这可让您能够使用 **systemd** 单元文件中指定的所有其他字段。

2. 根据 **mysleep.container** 文件创建 **mysleep.service** ：

```
$ systemctl --user daemon-reload
```

3. 可选：检查 **mysleep.service** 的状态：

```
$ systemctl --user status mysleep.service
○ mysleep.service - The sleep container
  Loaded: loaded (/home/username/.config/containers/systemd/mysleep.container;
generated)
  Active: inactive (dead)
```

4. 启动 **mysleep.service** ：

```
$ systemctl --user start mysleep.service
```

验证

1. 检查 **mysleep.service** 的状态：

```
$ systemctl --user status mysleep.service
● mysleep.service - The sleep container
  Loaded: loaded (/home/username/.config/containers/systemd/mysleep.container;
generated)
  Active: active (running) since Thu 2023-02-09 18:07:23 EST; 2s ago
   Main PID: 265651 (conmon)
      Tasks: 3 (limit: 76815)
  Memory: 1.6M
     CPU: 94ms
  CGroup: ...
```

2. 列出所有容器：

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND           CREATED         STATUS
PORTS   NAMES
421c8293fc1b  registry.access.redhat.com/ubi9-minimal:latest             sleep 1000  30
seconds ago   Up 10 seconds ago systemd-mysleep
```

请注意，创建的容器的名称由以下元素组成：

- **systemd-** 前缀

- **systemd** 单元的名称，即 **systemd-mysleep**
  这种命名有助于将常见容器与在 **systemd** 单元中运行的容器区分开来。它还有助于确定容器运行在哪个单元中。如果要更改容器的名称，请使用 **[Container]** 部分中的 **ContainerName** 字段。

其他资源

- 使用 Quadlet 使 systemd 更适合 Podman
- Quadlet 上游文档

14.2. 启用 SYSTEMD 服务

启用服务时，您可以有不同的选项。

流程

- 启用服务：
  - 要在系统启动时启用服务，无论用户是否登录，输入：

    ```
    # systemctl enable <service>
    ```

    您必须将 **systemd** 单元文件复制到 **/etc/systemd/system** 目录中。
  - 要在用户登录时启动服务并在用户注销时停止该服务，输入：

    ```
    $ systemctl --user enable <service>
    ```

    您必须将 **systemd** 单元文件复制到 **$HOME/.config/systemd/user** 目录中。
  - 要允许用户在系统启动并保留日志时启动服务，请输入：

    ```
    # loginctl enable-linger <username>
    ```

其他资源

- **systemctl** man page
- **loginctl** man page
- 启用一个系统服务，以便在引导时启动

14.3. 使用 SYSTEMD 自动启动容器

您可以使用 **systemctl** 命令控制 **systemd** 系统和服务管理器的状态。您可以以非 root 用户身份启用、启动、停止服务。若要以 root 用户身份安装服务，请省略 **--user** 选项。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 重新载入 **systemd** Manager 配置：

```
# systemctl --user daemon-reload
```

2. 启用服务 **container.service**，并在引导时启动它：

```
# systemctl --user enable container.service
```

3. 立即启动该服务：

```
# systemctl --user start container.service
```

4. 检查服务的状态：

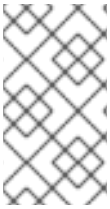
```
$ systemctl --user status container.service
● container.service - Podman container.service
   Loaded: loaded (/home/user/.config/systemd/user/container.service; enabled; vendor
preset: enabled)
   Active: active (running) since Wed 2020-09-16 11:56:57 CEST; 8s ago
     Docs: man:podman-generate-systemd(1)
  Process: 80602 ExecStart=/usr/bin/podman run --conmon-pidfile
//run/user/1000/container.service-pid --cidfile //run/user/1000/container.service-cid -d ubi9-
minimal:>
  Process: 80601 ExecStartPre=/usr/bin/rm -f //run/user/1000/container.service-pid
//run/user/1000/container.service-cid (code=exited, status=0/SUCCESS)
 Main PID: 80617 (conmon)
   CGroup: /user.slice/user-1000.slice/user@1000.service/container.service
           ├─ 2870 /usr/bin/podman
           ├─80612 /usr/bin/slirp4netns --disable-host-loopback --mtu 65520 --enable-sandbox -
-enable-seccomp -c -e 3 -r 4 --netns-type=path /run/user/1000/netns/cni->
           ├─80614 /usr/bin/fuse-overlayfs -o
lowerdir=/home/user/.local/share/containers/storage/overlay/l/YJSPGXM2OCDZPLMLXJOW3N
RF6Q:/home/user/.local/share/contain>
           ├─80617 /usr/bin/conmon --api-version 1 -c
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa -u
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72>
           └─cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa
             └─80626 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1d
```

您可以使用 **systemctl is-enabled container.service** 命令检查服务是否已启用。

验证步骤

- 列出正在运行或已退出的容器：

```
# podman ps
CONTAINER ID  IMAGE                                  COMMAND  CREATED         STATUS
PORTS  NAMES
f20988d59920  registry.access.redhat.com/ubi9-minimal:latest  top      12 seconds ago  Up 11
seconds ago         funny_zhukovsky
```



注意

要停止 **container.service**，请输入：

```
# systemctl --user stop container.service
```

其他资源

- **systemctl** man page
- 使用 Podman 和共享 systemd 服务运行容器
- 启用一个系统服务，以便在引导时启动

14.4. 与 PODMAN GENERATE SYSTEMD 命令相比，使用 QUADLETS 的优点

您可以使用 Quadlets 工具，它描述了如何以类似于常规 **systemd** 单元文件的格式运行容器。

注意

Quadlet 从 Podman v4.6 开始提供。

与使用 **podman generate systemd** 命令生成的单元文件相比，Quadlets 有很多优点，例如：

- **易于维护** ：容器描述侧重于相关的容器详情，隐藏在 **systemd** 下运行容器的技术详情。
- **自动更新** ：在更新后，Quadlet 不需要手动重新生成单元文件。如果发布了新版本的 Podman，则在 **systemclt daemon-reload** 命令执行时会自动更新您的服务，例如在引导时。
- **简化的工作流** ：由于简化的语法，您可以从头开始创建 Quadlet 文件，并在任何位置部署它们。
- **支持标准 systemd 选项** ：Quadlet 使用新表扩展了现有的 systemd-unit 语法，例如用于配置容器的表。

注意

Quadlet 支持 Kubernetes YAML 功能的子集。如需更多信息，请参阅 支持的 YAML 字段的支持矩阵。您可以使用以下工具之一生成 YAML 文件：

- podman: **podman generate kube** 命令
- OpenShift: 带有 **--dry-run** 选项的 **oc generate** 命令
- Kubernetes: 带有 **--dry-run** 选项的 **kubectl create** 命令

Quadlet 支持以下单元文件类型：

- **容器单元** ：用于通过运行 **podman run** 命令管理容器。
  - 文件扩展名：**.container**
  - 部分名称：**[Container]**

- 必填字段：**Image** 描述服务运行的容器映像

- **Kube units**:用于通过运行 **podman kube play** 命令管理 Kubernetes YAML 文件中定义的容器。
  - 文件扩展名：**.kube**
  - 部分名称：**[Kube]**
  - 必填字段：**YAML** 定义 Kubernetes YAML 文件的路径
- **Network units** ：用于创建可能在 **.container** 或 **.kube** 文件中引用的 Podman 网络。
  - 文件扩展名：**.network**
  - 部分名称：**[Network]**
  - 必填字段：None
- **Volume units** ：用于创建可能在 **.container** 文件中引用的 Podman 卷。
  - 文件扩展名：**.volume**
  - 部分名称：**[Volume]**
  - 必填字段：None

其他资源

- Quadlet 上游文档

14.5. 使用 PODMAN 生成 SYSTEMD 单元文件

Podman 允许 **systemd** 控制和管理容器进程。您可以使用 **podman generate systemd** 命令为现有的容器和 pod 生成一个 **systemd** 单元文件。建议使用 **podman generate systemd**，因为生成的单元文件会频繁变化（通过对 Podman 的更新），并使用 **podman generate systemd** 确保获取最新版本的单元文件。

注意

从 Podman v4.6 开始，您可以使用描述如何以类似于常规 **systemd** 单元文件的格式运行容器的 Quadlets，并隐藏了在 **systemd** 下运行容器的复杂性。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建容器（如 **myubi**）：

```
$ podman create --name myubi registry.access.redhat.com/ubi9:latest sleep infinity
0280afe98bb75a5c5e713b28de4b7c5cb49f156f1cce4a208f13fee2f75cb453
```

2. 使用容器名称或 ID 生成 **systemd** 单元文件，并将其定向到 **~/.config/systemd/user/container-myubi.service** 文件中：

```
$ podman generate systemd --name myubi > ~/.config/systemd/user/container-myubi.service
```

验证步骤

- 显示生成的 **systemd** 单元文件的内容：

```
$ cat ~/.config/systemd/user/container-myubi.service
# container-myubi.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:34:46 CEST 2021

[Unit]
Description=Podman container-myubi.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/1000/containers

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start myubi
ExecStop=/usr/bin/podman stop -t 10 myubi
ExecStopPost=/usr/bin/podman stop -t 10 myubi
PIDFile=/run/user/1000/containers/overlay-containers/9683103f58a32192c84801f0be93446cb33c1ee7d9cdda225b78049d7c5deea4/userdata/conmon.pid
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

  - **Restart=on-failure** 行设置重启策略，并指示 **systemd** 在服务无法启动或无法完全停止服务时重启，或者当进程非零退出时重启。
  - **ExecStart** 行描述了如何启动容器。
  - **ExecStop** 行描述了如何停止和移除容器。

其他资源

- 使用 Podman 和共享 systemd 服务运行容器

14.6. 使用 PODMAN 自动生成 SYSTEMD 单元文件

默认情况下，Podman 为现有容器或 pod 生成一个单元文件。您可以使用 **podman generate systemd --new** 生成更多可移植的 **systemd** 单元文件。**--new** 标志指示 Podman 生成创建、启动和删除容器的单元文件。

注意

从 Podman v4.6 开始，您可以使用描述如何以类似于常规 **systemd** 单元文件的格式运行容器的 Quadlets，并隐藏了在 **systemd** 下运行容器的复杂性。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 拉取您要在系统中使用的镜像。例如，要拉取 **httpd-24** 镜像：

```
# podman pull registry.access.redhat.com/ubi9/httpd-24
```

2. 可选：列出系统中所有可用镜像：

```
# podman images
REPOSITORY                                TAG         IMAGE ID      CREATED       SIZE
registry.access.redhat.com/ubi9/httpd-24  latest      8594be0a0b57  2 weeks ago   462
MB
```

3. 创建 **httpd** 容器：

```
# podman create --name httpd -p 8080:8080 registry.access.redhat.com/ubi9/httpd-24
cdb9f981cf143021b1679599d860026b13a77187f75e46cc0eac85293710a4b1
```

4. 可选：验证容器是否已创建：

```
# podman ps -a
CONTAINER ID  IMAGE                                     COMMAND               CREATED
STATUS      PORTS                   NAMES
cdb9f981cf14  registry.access.redhat.com/ubi9/httpd-24:latest  /usr/bin/run-http...  5 minutes
ago  Created     0.0.0.0:8080->8080/tcp  httpd
```

5. 为 **httpd** 容器生成 **systemd** 单元文件：

```
# podman generate systemd --new --files --name httpd
/root/container-httpd.service
```

6. 显示生成的 **container-httpd.service systemd** 单元文件的内容：

```
# cat /root/container-httpd.service
# container-httpd.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:41:44 CEST 2021

[Unit]
Description=Podman container-httpd.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 --sdnotify=conmon --cgroups=no-conmon --rm -d --replace --name httpd -p 8080:8080 registry.access.redhat.com/ubi9/httpd-24
ExecStop=/usr/bin/podman stop --ignore --cidfile=%t/%n.ctr-id
ExecStopPost=/usr/bin/podman rm -f --ignore --cidfile=%t/%n.ctr-id
Type=notify
NotifyAccess=all

[Install]
WantedBy=multi-user.target default.target
```

注意

使用 **--new** 选项生成的单元文件不会期望容器和 pod 存在。因此，它们会在启动服务时执行 **podman run** 命令（请参阅 **ExecStart** 行），而不是 **podman start** 命令。例如，请参阅使用 Podman 生成 systemd 单元文件一节。

- **podman run** 命令使用以下命令行选项：
  - **--conmon-pidfile** 选项指向存储主机上运行的 **conmon** 进程的进程 ID 的路径。**conmon** 进程以与容器相同的退出状态终止，允许 **systemd** 报告正确的服务状态并在需要时重启容器。
  - **--cidfile** 选项指向存储容器 ID 的路径。
  - **%t** 是运行时间目录根目录的路径，例如 **/run/user/$UserID**。
  - **%n** 是该服务的全名。

1. 将单元文件复制到 **/etc/systemd/system** 中，以便以 root 用户身份安装它们：

```
# cp -Z container-httpd.service /etc/systemd/system
```

2. 启用并启动 **container-httpd.service** ：

```
# systemctl daemon-reload
# systemctl enable --now container-httpd.service
Created symlink /etc/systemd/system/multi-user.target.wants/container-httpd.service → /etc/systemd/system/container-httpd.service.
Created symlink /etc/systemd/system/default.target.wants/container-httpd.service → /etc/systemd/system/container-httpd.service.
```

验证步骤

- 检查 **container-httpd.service** 的状态：

```
# systemctl status container-httpd.service
  ● container-httpd.service - Podman container-httpd.service
    Loaded: loaded (/etc/systemd/system/container-httpd.service; enabled; vendor preset: disabled)
```

```
     Active: active (running) since Tue 2021-08-24 09:53:40 EDT; 1min 5s ago
       Docs: man:podman-generate-systemd(1)
    Process: 493317 ExecStart=/usr/bin/podman run --conmon-pidfile /run/container-
httpd.pid --cidfile /run/container-httpd.ctr-id --cgroups=no-conmon -d --repla>
    Process: 493315 ExecStartPre=/bin/rm -f /run/container-httpd.pid /run/container-httpd.ctr-
id (code=exited, status=0/SUCCESS)
   Main PID: 493435 (conmon)
      ...
```

其他资源

- 改进了 Systemd 与 Podman 2.0 的集成 的文章
- 启用一个系统服务，以便在引导时启动

14.7. 使用 SYSTEMD 自动启动 POD

您可以将多个容器作为 **systemd** 服务启动。请注意，**system** ctl 命令应该只用于 pod，不应通过 **systemctl** 单独启动或停止容器，因为它们由 pod 服务与内部 infra-container 一起管理。

注意

从 Podman v4.6 开始，您可以使用描述如何以类似于常规 **systemd** 单元文件的格式运行容器的 Quadlets，并隐藏了在 **systemd** 下运行容器的复杂性。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建一个空 pod，如 **systemd-pod**:

```
$ podman pod create --name systemd-pod
11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

2. 可选：列出所有 pod:

```
$ podman pod ps
POD ID        NAME         STATUS   CREATED         # OF CONTAINERS  INFRA ID
11d4646ba41b  systemd-pod  Created  40 seconds ago  1                8a428b257111
11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

3. 在空 pod 中创建两个容器。例如，要在 **systemd-pod** 中创建 **container0** 和 **container1** ：

```
$ podman create --pod systemd-pod --name container0
registry.access.redhat.com/ubi9 top
$ podman create --pod systemd-pod --name container1
registry.access.redhat.com/ubi9 top
```

4. 可选：列出与其关联的所有 pod 和容器：

```
$ podman ps -a --pod
```

```
CONTAINER ID  IMAGE                                COMMAND  CREATED         STATUS
PORTS   NAMES               POD ID        PODNAME
24666f47d9b2  registry.access.redhat.com/ubi9:latest  top      3 minutes ago  Created
container0          3130f724e229  systemd-pod
56eb1bf0cdfe  k8s.gcr.io/pause:3.2                          4 minutes ago  Created
3130f724e229-infra  3130f724e229  systemd-pod
62118d170e43  registry.access.redhat.com/ubi9:latest  top      3 seconds ago  Created
container1          3130f724e229  systemd-pod
```

5. 为新的 pod 生成 **systemd** 单元文件：

```
$ podman generate systemd --files --name systemd-pod
/home/user1/pod-systemd-pod.service
/home/user1/container-container0.service
/home/user1/container-container1.service
```

请注意，生成了三个 **systemd** 单元文件，一个用于 **systemd-pod** pod，两个用于容器 **container0** 和 **container1**。

6. 显示 **pod-systemd-pod.service** 单元文件：

```
$ cat pod-systemd-pod.service
# pod-systemd-pod.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:49:17 CEST 2021

[Unit]
Description=Podman pod-systemd-pod.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=
Requires=container-container0.service container-container1.service
Before=container-container0.service container-container1.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start bcb128965b8e-infra
ExecStop=/usr/bin/podman stop -t 10 bcb128965b8e-infra
ExecStopPost=/usr/bin/podman stop -t 10 bcb128965b8e-infra
PIDFile=/run/user/1000/containers/overlay-
containers/1dfdcf20e35043939ea3f80f002c65c00d560e47223685dbc3230e26fe001b29/userda
ta/conmon.pid
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

- **[Unit]** 部分中的 **Requires** 行定义 **container-container0.service** 和 **container-container1.service** 单元文件的依赖项。两个单元文件都会被激活。
- **[Service]** 部分中的 **ExecStart** 和 **ExecStop** 行分别启动和停止 infra-container。

7. 显示 **container-container0.service** 单元文件：

```
$ cat container-container0.service
# container-container0.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:49:17 CEST 2021

[Unit]
Description=Podman container-container0.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/1000/containers
BindsTo=pod-systemd-pod.service
After=pod-systemd-pod.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start container0
ExecStop=/usr/bin/podman stop -t 10 container0
ExecStopPost=/usr/bin/podman stop -t 10 container0
PIDFile=/run/user/1000/containers/overlay-
containers/4bccd7c8616ae5909b05317df4066fa90a64a067375af5996fdef9152f6d51f5/userdat
a/conmon.pid
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

- **[Unit]** 部分中的 **BindsTo** 行定义 **pod-systemd-pod.service** 单元文件的依赖项
- **[Service]** 部分中的 **ExecStart** 和 **ExecStop** 行分别启动和停止 **container0**。

8. 显示 **container-container1.service** 单元文件：

```
$ cat container-container1.service
```

9. 将所有生成的文件复制到 **$HOME/.config/systemd/user** 中，以便以非 root 用户身份安装：

```
$ cp pod-systemd-pod.service container-container0.service container-
container1.service $HOME/.config/systemd/user
```

10. 启用该服务并在用户登录时启动：

```
$ systemctl enable --user pod-systemd-pod.service
Created symlink /home/user1/.config/systemd/user/multi-user.target.wants/pod-systemd-
pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
Created symlink /home/user1/.config/systemd/user/default.target.wants/pod-systemd-
pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
```

请注意，服务会在用户注销时停止。

验证步骤

- 检查是否启用该服务：

```
$ systemctl is-enabled pod-systemd-pod.service
enabled
```

其他资源

- **podman-create** man page
- **podman-generate-systemd** man page
- **systemctl** man page
- 使用 Podman 和共享 systemd 服务运行容器
- 启用一个系统服务，以便在引导时启动

14.8. 使用 PODMAN 自动更新容器

podman auto-update 命令允许您根据自动更新策略自动更新容器。当容器镜像在注册中心中被更新时，**podman auto-update** 命令会更新服务。要使用自动更新，必须使用 **--label "io.containers.autoupdate=image"** 标签创建容器，并在 **podman generate systemd --new** 命令生成的 **systemd** 单元中运行。

Podman 会搜索 **"io.containers.autoupdate"** 标签设为 **"image"** 的正在运行的容器，并与容器注册表进行通信。如果镜像已更改，Podman 会重启相应的 **systemd** 单元来停止旧容器，并使用新镜像创建一个新容器。因此，容器、其环境和所有依赖项都会被重启。

注意

从 Podman v4.6 开始，您可以使用描述如何以类似于常规 **systemd** 单元文件的格式运行容器的 Quadlets，并隐藏了在 **systemd** 下运行容器的复杂性。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 启动基于 **registry.access.redhat.com/ubi9/ubi-init** 镜像的 **myubi** 容器：

```
# podman run --label "io.containers.autoupdate=image" \
--name myubi -dt registry.access.redhat.com/ubi9/ubi-init top
bc219740a210455fa27deacc96d50a9e20516492f1417507c13ce1533dbdcd9d
```

2. 可选：列出正在运行或已退出的容器：

```
# podman ps -a
CONTAINER ID  IMAGE                                  COMMAND  CREATED        STATUS
PORTS   NAMES
76465a5e2933  registry.access.redhat.com/9/ubi-init:latest  top      24 seconds ago  Up 23
seconds ago          myubi
```

3. 为 **myubi** 容器生成一个 **systemd** 单元文件：

```
# podman generate systemd --new --files --name myubi /root/container-myubi.service
```

4. 将单元文件复制到 **/usr/lib/systemd/system** 中，以便以 root 用户身份安装它：

```
# cp -Z ~/container-myubi.service /usr/lib/systemd/system
```

5. 重新载入 **systemd** Manager 配置：

```
# systemctl daemon-reload
```

6. 启动并检查容器的状态：

```
# systemctl start container-myubi.service
# systemctl status container-myubi.service
```

7. 自动更新容器：

```
# podman auto-update
```

其他资源

- 改进了 Systemd 与 Podman 2.0 的集成 的文章
- 使用 Podman 和共享 systemd 服务运行容器
- 启用一个系统服务，以便在引导时启动

14.9. 使用 SYSTEMD 自动更新容器

如 一节中所述，使用 Podman 自动更新容器，

您可以使用 **podman auto-update** 命令更新容器。它整合成自定义脚本，并在需要时调用。自动更新容器的另一种方法是使用预安装的 **podman-auto-update.timer** 和 **podman-auto-update.service systemd** 服务。**podman-auto-update.timer** 可以配置成在特定日期或时间触发自动更新。**podman-auto-update.service** 可以进一步由 **systemctl** 命令启动，或者用作其他 **systemd** 服务的依赖项。因此，可以基于时间和事件的自动更新可以以各种方法触发，以满足单个需求和用例。

注意

从 Podman v4.6 开始，您可以使用描述如何以类似于常规 **systemd** 单元文件的格式运行容器的 Quadlets，并隐藏了在 **systemd** 下运行容器的复杂性。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 显示 **podman-auto-update.service** 单元文件：

```
# cat /usr/lib/systemd/system/podman-auto-update.service
```

```
[Unit]
Description=Podman auto-update service
Documentation=man:podman-auto-update(1)
Wants=network.target
After=network-online.target

[Service]
Type=oneshot
ExecStart=/usr/bin/podman auto-update

[Install]
WantedBy=multi-user.target default.target
```

2. 显示 **podman-auto-update.timer** 单元文件：

```
# cat /usr/lib/systemd/system/podman-auto-update.timer

[Unit]
Description=Podman auto-update timer

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
```

在本例中，**podman auto-update** 命令在每天午夜启动。

3. 在系统启动时启用 **podman-auto-update.timer** 服务：

```
# systemctl enable podman-auto-update.timer
```

4. 启动 **systemd** 服务：

```
# systemctl start podman-auto-update.timer
```

5. 可选：列出所有计时器：

```
# systemctl list-timers --all
NEXT                        LEFT      LAST                        PASSED       UNIT
ACTIVATES
Wed 2020-12-09 00:00:00 CET  9h left   n/a                         n/a          podman-auto-
update.timer     podman-auto-update.service
```

您可以看到 **podman-auto-update.timer** 激活了 **podman-auto-update.service**。

其他资源

- 改进了 Systemd 与 Podman 2.0 的集成 的文章
- 使用 Podman 和共享 systemd 服务运行容器
- 启用一个系统服务，以便在引导时启动

第 15 章 使用 ANSIBLE PLAYBOOK 管理容器

使用 Podman 4.2 时，您可以使用 Podman RHEL 系统角色来管理 Podman 配置、容器以及运行 Podman 容器的 systemd 服务。

RHEL 系统角色提供了一个配置界面，用于远程管理多个 RHEL 系统。您可以使用界面跨多个 RHEL 版本管理系统配置，以及采用新的主版本。如需更多信息，请参阅使用 RHEL 系统角色自动化系统管理。

15.1. 使用绑定挂载创建无根容器

您可以通过运行 Ansible playbook 并使用它来管理应用程序配置，使用 **podman** RHEL 系统角色创建带有绑定挂载的无根容器。

先决条件

- 您已准备好控制节点和受管节点
- 以可在受管主机上运行 playbook 的用户登录到控制节点。
- 用于连接到受管节点的帐户具有 **sudo** 权限。

流程

1. 创建一个包含以下内容的 playbook 文件，如 ~/**playbook.yml** ：

```
- hosts: managed-node-01.example.com
  vars:
    podman_create_host_directories: true
    podman_firewall:
      - port: 8080-8081/tcp
        state: enabled
      - port: 12340/tcp
        state: enabled
    podman_selinux_ports:
      - ports: 8080-8081
        setype: http_port_t
    podman_kube_specs:
      - state: started
        run_as_user: dbuser
        run_as_group: dbgroup
        kube_file_content:
          apiVersion: v1
          kind: Pod
          metadata:
            name: db
          spec:
            containers:
              - name: db
                image: quay.io/db/db:stable
                ports:
                  - containerPort: 1234
                    hostPort: 12340
                volumeMounts:
                  - mountPath: /var/lib/db:Z
                    name: db
```

```
        volumes:
          - name: db
            hostPath:
              path: /var/lib/db
    - state: started
      run_as_user: webapp
      run_as_group: webapp
      kube_file_src: /path/to/webapp.yml
roles:
  - linux-system-roles.podma
```

此流程创建一个有两个容器的 pod。**podman_kube_specs** 角色变量描述了 pod。

- **run_as_user** 和 **run_as_group** 字段指定容器是无根的。
- 包含 Kubernetes YAML 文件的 **kube_file_content** 字段定义了名为 **db** 的第一个容器。您可以使用 **podman kube generate** 命令生成 Kubernetes YAML 文件。
  - **db** 容器是基于 **quay.io/db/db:stable** 容器镜像。
  - **db** 绑定挂载将主机上的 **/var/lib/db** 目录映射到容器中的 **/var/lib/db** 目录。**Z** 标志使用私有的非共享标签标记内容，因此只有 **db** 容器才能访问其内容。
- **kube_file_src** 字段定义第二个容器。控制器节点上的 **/path/to/webapp.yml** 文件的内容将复制到受管节点上的 **kube_file** 字段中。
- 设置 **podman_create_host_directories: true** 以在主机上创建目录。这指示角色检查 **hostPath** 卷的 kube 规范，并在主机上创建这些目录。如果您需要对所有权和权限进行更多的控制，请使用 **podman_host_directories**。

2. 验证 playbook 语法：

```
$ ansible-playbook --syntax-check --ask-vault-pass ~/playbook.yml
```

请注意，这个命令只验证语法，不会防止错误但有效的配置。

3. 运行 playbook：

```
$ ansible-playbook --ask-vault-pass ~/playbook.yml
```

其他资源

- **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** 文件
- **/usr/share/doc/rhel-system-roles/podman/** directory

15.2. 使用 PODMAN 卷创建有根容器

您可以通过运行 Ansible playbook 并使用它来管理应用程序配置，使用 **podman** RHEL 系统角色创建带有 Podman 卷的 rootful 容器。

先决条件

- 您已准备好控制节点和受管节点

- 以可在受管主机上运行 playbook 的用户登录到控制节点。
- 用于连接到受管节点的帐户具有 **sudo** 权限。

流程

1. 创建一个包含以下内容的 playbook 文件，如 **~/playbook.yml** ：

```
- hosts: managed-node-01.example.com
  vars:
    podman_firewall:
      - port: 8080/tcp
        state: enabled
    podman_kube_specs:
      - state: started
        kube_file_content:
          apiVersion: v1
          kind: Pod
          metadata:
            name: ubi8-httpd
          spec:
            containers:
              - name: ubi8-httpd
                image: registry.access.redhat.com/ubi8/httpd-24
                ports:
                  - containerPort: 8080
                    hostPort: 8080
                volumeMounts:
                  - mountPath: /var/www/html:Z
                    name: ubi8-html
            volumes:
              - name: ubi8-html
                persistentVolumeClaim:
                  claimName: ubi8-html-volume
  roles:
    - linux-system-roles.podman
```

此流程创建一个带有一个容器的 pod。**podman_kube_specs** 角色变量描述了 pod。

- 默认情况下，**podman** 角色创建根容器。
- 包含 Kubernetes YAML 文件的 **kube_file_content** 字段定义了名为 **ubi8-httpd** 的容器。
  - **ubi8-httpd** 容器是基于 **registry.access.redhat.com/ubi8/httpd-24** 容器镜像。
    - **ubi8-html-volume** 将主机上的 **/var/www/html** 目录映射到容器。**Z** 标志使用私有的非共享标签标记内容，因此只有 **ubi8-httpd** 容器可以访问其内容。
    - pod 使用挂载路径 **/var/www/html** 挂载名为 **ubi8-html-volume** 的现有持久性卷。

2. 验证 playbook 语法：

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

请注意，这个命令只验证语法，不会防止错误但有效的配置。

3. 运行 playbook：

```
$ ansible-playbook ~/playbook.yml
```

其他资源

- **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** 文件
- **/usr/share/doc/rhel-system-roles/podman/** directory

15.3. 创建带有 SECRET 的 QUADLET 应用程序

您可以通过运行 Ansible playbook，使用 **podman** RHEL 系统角色创建带有 secret 的 Quadlet 应用程序。

先决条件

- 您已准备好控制节点和受管节点
- 以可在受管主机上运行 playbook 的用户登录到控制节点。
- 用于连接到受管节点的帐户具有 **sudo** 权限。
- 容器中的 web 服务器应使用的证书和对应的私钥存储在 **~/certificate.pem** 和 **~/key.pem** 文件中。

流程

1. 显示证书和私钥文件的内容：

```
$ cat ~/certificate.pem
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----

$ cat ~/key.pem
-----BEGIN PRIVATE KEY-----
...
-----END PRIVATE KEY-----
```

   后续步骤中需要此信息。

2. 将您的敏感变量存储在加密文件中：

   a. 创建密码库：

```
$ ansible-vault create vault.yml
New Vault password: <vault_password>
Confirm New Vault password: <vault_password>
```

   b. 在 **ansible-vault create** 命令打开编辑器后，以 ***< key> : < value> 格式输入*** 敏感数据：

```
root_password: <root_password>
certificate: |-
```

```
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
  key: |-
    -----BEGIN PRIVATE KEY-----
    ...
    -----END PRIVATE KEY-----
```

确保 证书**和密**钥 变量中的所有行都以两个空格开头。

c. 保存更改，并关闭编辑器。Ansible 加密 vault 中的数据。

3. 创建一个包含以下内容的 playbook 文件，如 ~/**playbook.yml** ：

```
- name: Deploy a wordpress CMS with MySQL database
  hosts: managed-node-01.example.com
  vars_files:
    - vault.yml
  tasks:
  - name: Create and run the container
    ansible.builtin.include_role:
      name: rhel-system-roles.podman
    vars:
      podman_create_host_directories: true
      podman_activate_systemd_unit: false
      podman_quadlet_specs:
        - name: quadlet-demo
          type: network
          file_content: |
            [Network]
            Subnet=192.168.30.0/24
            Gateway=192.168.30.1
            Label=app=wordpress
        - file_src: quadlet-demo-mysql.volume
        - template_src: quadlet-demo-mysql.container.j2
        - file_src: envoy-proxy-configmap.yml
        - file_src: quadlet-demo.yml
        - file_src: quadlet-demo.kube
          activate_systemd_unit: true
      podman_firewall:
        - port: 8000/tcp
          state: enabled
        - port: 9000/tcp
          state: enabled
      podman_secrets:
        - name: mysql-root-password-container
          state: present
          skip_existing: true
          data: "{{ root_password }}"
        - name: mysql-root-password-kube
          state: present
          skip_existing: true
          data: |
            apiVersion: v1
            data:
              password: "{{ root_password | b64encode }}"
```

```
      kind: Secret
      metadata:
        name: mysql-root-password-kube
    - name: envoy-certificates
      state: present
      skip_existing: true
      data: |
        apiVersion: v1
        data:
          certificate.key: {{ key | b64encode }}
          certificate.pem: {{ certificate | b64encode }}
        kind: Secret
        metadata:
          name: envoy-certificates
```

该流程创建一个与 MySQL 数据库配对的 WordPress 内容管理系统。**podman_quadlet_specs role** 为 Quadlet 定义一组配置，它指向以某种方式一起工作的一组容器或服务。它包括以下规范：

- Wordpress 网络由 **quadlet-demo** 网络单元定义。
- MySQL 容器的卷配置由 **file_src: quadlet-demo-mysql.volume** 字段定义。
- **template_src: quadlet-demo-mysql.container.j2** 字段用于生成 MySQL 容器的配置。
- 两个 YAML 文件如下：**file_src: envoy-proxy-configmap.yml** 和 **file_src: quadlet-demo.yml**.请注意，.yml 不是一个有效的 Quadlet 单元类型，因此这些文件将只会被复制，不会作为 Quadlet 规范处理。
- Wordpress 和 envoy 代理容器和配置由 **file_src: quadlet-demo.kube** 字段定义。kube 单元将之前 **[Kube]** 部分中的 YAML 文件称为 **Yaml=quadlet-demo.yml** 和 **ConfigMap=envoy-proxy-configmap.yml**。

4. 验证 playbook 语法：

```
$ ansible-playbook --syntax-check --ask-vault-pass ~/playbook.yml
```

请注意，这个命令只验证语法，不会防止错误但有效的配置。

5. 运行 playbook：

```
$ ansible-playbook --ask-vault-pass ~/playbook.yml
```

其他资源

- **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** 文件
- **/usr/share/doc/rhel-system-roles/podman/** directory

第 16 章 使用 RHEL WEB 控制台管理容器镜像

您可以使用基于 Web 界面的 RHEL web 控制台来拉取、修剪或删除您的容器镜像。

16.1. 在 WEB 控制台中拉取容器镜像

您可以将容器镜像下载到本地系统，并使用它们创建容器。

先决条件

- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

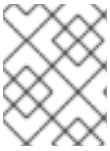
```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 在 **Images** 表中，单击右上角的溢出菜单，然后选择 **Download new image**。
3. 此时会出现 **Search for an image**对话框。
4. 在 **Search for** 字段中，输入镜像的名称或指定其描述。
5. 在 **in** 下拉列表中，选择要从中拉取镜像的注册中心。
6. 可选：在 **Tag** 字段中，输入镜像的标签。
7. 点 **Download**。

验证

- 点击主菜单中的 **Podman containers**。您可以在 **Images** 表中看到新下载的镜像。



注意

您可以通过点 **Images** 表中的 **Create container**，从下载的镜像创建容器。要创建容器，请按照 在 web 控制台中创建容器 中的步骤 3-8。

16.2. 在 WEB 控制台中修剪容器镜像

您可以删除所有未使用的、没有任何基于它的容器的镜像。

先决条件

- 至少一个容器镜像会被拉取。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 在 **Images** 表中，单击右上角的溢出菜单，然后选择 **Prune unused images**。
3. 此时会出现带有镜像列表的弹出窗口。点 **Prune** 来确认您的选择。

验证

- 点击主菜单中的 **Podman containers**。删除的镜像不应列在 **Images** 表中。

16.3. 在 WEB 控制台中删除容器镜像

您可以使用 Web 控制台删除之前拉取的容器镜像。

先决条件

- 至少一个容器镜像会被拉取。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点主菜单中的 **Podman containers**。
2. 在 **Images** 表中，选择您要删除的镜像，然后点溢出菜单，并选择 **Delete**。
3. 此时会出现一个窗口。点 **Delete tagged images** 以确认您的选择。

验证

- 点 主菜单中的 **Podman containers**。删除的容器不应列在 **Images** 表中。

第 17 章 使用 RHEL WEB 控制台管理容器

您可以使用 Red Hat Enterprise Linux Web 控制台管理容器和 pod。使用 Web 控制台，您可以以非 root 或 root 用户身份创建容器。

- 作为 *root* 用户，您可以使用额外的特权和选项创建系统容器。
- 作为 *非 root* 用户，您有两个选项：
  - 要仅创建用户容器，您可以在其默认模式 - **Limited access** 中使用 Web 控制台。
  - 要创建用户和系统容器，请单击 Web 控制台页面顶部面板中的 **Administrative access**。

有关根和无根容器之间的区别的详情，请参阅 无根容器的特殊注意事项。

17.1. 在 WEB 控制台中创建容器

您可以创建容器并添加端口映射、卷、环境变量、健康检查等。

先决条件

- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 点 **Create container**。
3. 在 **Name** 字段中输入容器的名称。
4. 在 **Details** 选项卡中提供所需信息。
   - *仅适用于具有管理权限* ：选择容器的所有者：系统或用户。
   - 在 **Image** 下拉列表中，选择或搜索所选注册中心中的容器镜像。
     - 可选：选中 **Pull latest image** 复选框，以拉取最新的容器镜像。
   - **Command** 字段指定命令。如果需要，您可以更改默认命令。
     - 可选：选中 **With terminal**，以使用终端运行容器。
   - **Memory limit** 字段指定容器的内存限制。要更改默认的内存限制，请选中复选框并指定限制。
   - *仅适用于系统容器* ：在 **CPU 共享字段** 中，指定 CPU 时间的相对量。默认值为 1024。选中复选框以修改默认值。
   - *仅适用于系统容器* ：在 **Restart policy** 下拉菜单中，选择以下选项之一：

- No（默认值）：无操作。
- On Failure ：在失败时重启容器。
- Always ：在退出或重启系统后重启容器。

5. 在 Integration 选项卡中提供所需的信息。
   - 点 **Add port mapping**，来在容器和主机系统之间添加端口映射。
     - 输入 *IP 地址*、*Host port*、*Container port* 和 *Protocol*。
   - 点 **Add volume** 添加卷。
     - 输入 *host path*、*Container path*。您可以选择 **Writable** 选项复选框来创建一个可写卷。在 SELinux 下拉列表中选择以下选项之一：**No Label**、**Shared** 或 **Private**。
   - 单击 **Add variable** 添加环境变量。
     - 输入 *Key* 和 *Value*。
6. 在 **Health check** 选项卡中提供所需的信息。
   - 在 **Command** 字段中，输入 'healthcheck' 命令。
   - 指定 healthcheck 选项：
     - **Interval**（默认为 30 秒）
     - **Timeout**（默认为 30 秒）
     - **Start period**
     - **Retries** （默认为 3）
     - 当不健康时：选择以下选项之一：
       - **No action**（默认）：不执行任何操作。
       - **Restart** ：重启容器。
       - **Stop** ：停止容器。
       - **Force stop** ：强制停止容器，它不等待容器退出。
7. 点 **Create and run** 创建并运行容器。

注意

您可以点击 **Create** 来只创建容器。

验证

- 点击主菜单中的 **Podman containers**。您可以在 **Containers** 表中看到新创建的容器。

17.2. 在 WEB 控制台中检查容器

您可以在 web 控制台中显示容器的详细信息。

先决条件

- 容器已创建。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 点 **>** 箭头图标查看容器的详情。
   - 在 **Details** 选项卡中，您可以看到容器 ID、镜像、命令、何时创建的（创建容器时的时间戳）及其状态。
     - *仅适用于系统容器*：您还可以查看 IP 地址、MAC 地址和网关地址。
   - 在 **Integration** 选项卡中，您可以查看环境变量、端口映射和卷。
   - 在 **Log** 选项卡中，您可以查看容器日志。
   - 在 **Console** 选项卡中，您可以使用命令行与容器进行交互。

17.3. 在 WEB 控制台中更改容器的状态

在 Red Hat Enterprise Linux web 控制台中，您可以启动、停止、重启、暂停和重命名系统上的容器。

先决条件

- 容器已创建。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 在 **Containers** 表中，选择您要修改的容器，点击 overflow 菜单，选择您要执行的操作：
   - **Start**
   - **stop**
   - **Force stop**
   - **Restart**

- Force restart
- Pause
- Rename

17.4. 在 WEB 控制台中提交容器

您可以根据容器的当前状态创建一个新镜像。

先决条件

- 容器已创建。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

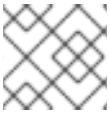
1. 点击主菜单中的 **Podman containers**。
2. 在 **Containers** 表中，选择您要修改的容器，点击 overflow 菜单并选择 **Commit**。
3. 在 **Commit container** 表单中，添加以下详情：
   - 在 **New image name** 字段中，输入镜像名称。
   - 可选：在 **Tag** 字段中，输入标签。
   - 可选：在 **Author** 字段中输入您的名称。
   - 可选：在 **Command** 字段中，更改命令（如果需要的话）。
   - 可选：检查您需要的 **Options** ：
     - 在创建镜像时暂停容器：容器及其进程在提交镜像时暂停。
     - 使用传统的 Docker 格式：如果您不使用 Docker 镜像格式，请使用 OCI 格式。
4. 点 **Commit**。

验证

- 点 主菜单中的 **Podman containers**。您可以在 **Images** 表中看到新创建的镜像。

17.5. 在 WEB 控制台中创建一个容器检查点

使用 Web 控制台，您可以在正在运行的容器或单个应用程序上设置一个检查点，并将其状态存储在磁盘上。



注意

创建检查点仅在系统容器上提供。

先决条件

- 容器正在运行。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

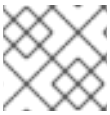
1. 点击主菜单中的 **Podman containers**。
2. 在 **Containers** 表中，选择您要修改的容器，点击 overflow 图标菜单并选择 **Checkpoint**。
3. 可选：在 **Checkpoint container** 表单中，检查您需要的选项：
   - 保留所有临时检查点文件：保留检查点过程中 CRIU 创建的所有临时日志和统计数据文件。如果检查点失败，这些文件不会被删除，以便做进一步调试。
   - 将检查点写入磁盘后让其继续运行：在检查点后让容器继续运行，而不是停止它。
   - 支持保留建立的 TCP 连接
4. 点 **Checkpoint**。

验证

- 点 主菜单中的 **Podman containers**。选择您做检查点的容器，点 overflow 菜单图标，并验证是否有 **Restore** 选项。

17.6. 在 WEB 控制台中恢复容器检查点

您可以在重启时同时使用保存的数据来恢复容器。



注意

创建检查点仅在系统容器上提供。

先决条件

- 已对容器执行了检查点。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

-

流程

1. 点击主菜单中的 **Podman containers**。
2. 在 **Containers** 表中，选择您要修改的容器，点击 overflow 菜单并选择 **Restore**。
3. 可选：在 **Restore container** 表单中，检查您需要的选项：
   - **保留所有临时检查点文件** ：在检查点过程中保留 CRIU 创建的所有临时日志和统计数据文件。如果检查点失败，这些文件不会被删除，以便做进一步调试。
   - **恢复建立的 TCP 连接**
   - **如果设置为静态，则忽略 IP 地址** ：如果容器使用 IP 地址启动，恢复的容器也会尝试使用该 IP 地址，如果该 IP 地址已在使用，则恢复会失败。如果您在创建容器时在 Integration 选项卡中添加了端口映射，则此选项适用。
   - **如果设置为静态，则忽略 MAC 地址** ：如果容器使用 MAC 地址启动，恢复的容器也会尝试使用该 MAC 地址，如果该 MAC 地址已在使用，则恢复会失败。
4. 单击 **Restore**。

验证

- 点 主菜单中的 **Podman containers**。您可以看到 **Containers** 表中恢复的容器正在运行。

17.7. 在 WEB 控制台中删除容器

您可以使用 Web 控制台删除现有容器。

先决条件

- 容器在系统上存在。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 在 **Containers** 表中，选择您要删除的容器，点击 overflow 菜单并选择 **Delete**。
3. 此时会出现弹出窗口。点 **Delete** 确认您的选择。

验证

- 点 主菜单中的 **Podman containers**。删除的容器不应列在 **Containers** 表中。

17.8. 在 WEB 控制台中创建 POD

您可以在 RHEL web 控制台界面中创建 pod。

先决条件

- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 点 **Create pod**。
3. 在 **Create pod** 表单中提供所需信息：
   - *仅适用于具有管理权限*：选择容器的所有者：系统或用户。
   - 在 **Name** 字段中输入容器的名称。
   - 点 **Add port mapping** 添加容器和主机系统之间的端口映射。
     - 输入 IP 地址、主机端口、容器端口和协议。
   - 点 **Add volume** 添加卷。
     - 输入主机路径，容器路径。您可以选中 Writable 复选框来创建可写卷。在 SELinux 下拉列表中选择以下选项之一：无标签、共享或私有。
4. 点 **Create**。

验证

- 点击主菜单中的 **Podman containers**。您可以在 **Containers** 表中看到新创建的 pod。

17.9. 在 WEB 控制台中在 POD 中创建容器

您可以在 pod 中创建容器。

先决条件

- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

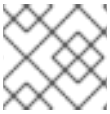
流程

1. 点击主菜单中的 **Podman containers**。
2. 点 **Create container in pod**。
3. 在 **Name** 字段中输入容器的名称。
4. 在 **Details** 选项卡中提供所需的信息。
   - *仅适用于具有管理权限* ：选择容器的所有者：系统或用户。
   - 在 **Image** 下拉列表中，选择或搜索所选注册中心中的容器镜像。
     - 可选：选中 **Pull latest image**复选框，以拉取最新的容器镜像。
   - **Command** 字段指定命令。如果需要，您可以更改默认命令。
     - 可选：选中 **With terminal**，以使用终端运行容器。
   - **Memory limit**字段指定容器的内存限制。要更改默认的内存限制，请选中复选框并指定限制。
   - *仅适用于系统容器* ：在 **CPU 共享字段** 中，指定 CPU 时间的相对量。默认值为 1024。选中复选框以修改默认值。
   - *仅适用于系统容器* ：在 **Restart policy** 下拉菜单中，选择以下选项之一：
     - **No**（默认值）：无操作。
     - **On Failure** ：在失败时重启容器。
     - **Always** ：在退出或系统引导时重启容器。
5. 在 **Integration** 选项卡中提供所需的信息。
   - 点 **Add port mapping**，来在容器和主机系统之间添加端口映射。
     - 输入 *IP 地址*、*Host port*、*Container port* 和 *Protocol*。
   - 点 **Add volume** 添加卷。
     - 输入 *host path*、*Container path*。您可以选择 **Writable** 选项复选框来创建一个可写卷。在 SELinux 下拉列表中选择以下选项之一：**No Label**、**Shared** 或 **Private**。
   - 单击 **Add variable** 添加环境变量。
     - 输入 *Key* 和 *Value*。
6. 在 **Health check** 选项卡中提供所需的信息。
   - 在 **Command** 字段中，输入 healthcheck 命令。
   - 指定 healthcheck 选项：
     - **Interval**（默认为 30 秒）
     - **Timeout**（默认为 30 秒）
     - **Start period**
     - **Retries** （默认为 3）

- 当不健康时：选择以下选项之一：
  - **No action**（默认）：不执行任何操作。
  - **Restart**：重启容器。
  - **Stop**：停止容器。
  - **Force stop**：强制停止容器，它不等待容器退出。

注意

容器的所有者与 pod 的所有者相同。



注意

在 pod 中，您可以检查容器，更改容器的状态、提交容器或删除容器。

验证

- 点击主菜单中的 **Podman containers**。您可以在 **Containers** 表下的 pod 中看到新创建的容器。

17.10. 在 WEB 控制台中更改 POD 的状态

您可以更改 pod 的状态。

先决条件

- pod 已创建。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 在 **Containers** 表中，选择要修改的 pod，点击 overflow 菜单并选择您要执行的操作：
   - **Start**
   - **stop**
   - **Force stop**
   - **Restart**
   - **Force restart**
   - **Pause**

17.11. 在 WEB 控制台中删除 POD

您可以使用 Web 控制台删除现有 pod。

先决条件

- pod 在系统上存在。
- Web 控制台已安装并可以访问。如需更多信息，请参阅 安装 Web 控制台 和 登录到 Web 控制台。
- **cockpit-podman** 附加组件已安装：

```
# dnf install cockpit-podman
```

流程

1. 点击主菜单中的 **Podman containers**。
2. 在 **Containers** 表中，选择您要删除的 pod，点击 overflow 菜单并选择 **Delete**。
3. 在以下弹出窗口中点击 **Delete** 以确认您的选择。

警告

您可以删除 pod 中的所有容器。

验证

- 点 主菜单中的 **Podman containers**。已删除的 pod 不应列在 **Containers** 表中。

第 18 章 在容器中运行 SKOPEO、BUILDAH 和 PODMAN

您可以在容器中运行 Skopeo、Buildah 和 Podman。

使用 Skopeo，您可以检查远程注册中心中的镜像，而无需下载整个镜像及其所有层。您还可以使用 Skopeo 来复制镜像、签名镜像、同步镜像以及在不同格式和层压缩间转换镜像。

Buildah 处理对 OCI 容器镜像的构建。使用 Buildah，您可以创建一个可运行的容器，可以从头开始，也可以使用一个镜像作为起点。您可以从正常工作的容器或使用 **Containerfile** 中的指令来创建镜像。您可以挂载和卸载正常工作容器的根文件系统。

借助 Podman，您可以管理容器和镜像、挂载到这些容器的卷，以及容器组构成的 pod。Podman 是基于容器生命周期管理的 **libpod** 库。**libpod** 库提供用于管理容器、Pod、容器镜像和卷的 API。

在容器中运行 Buildah、Skopeo 和 Podman 的原因：

- **CI/CD 系统** ：
  - **Podman 和 Skopeo** ：您可以在 Kubernetes 中运行 CI/CD 系统，或者使用 OpenShift 来构建容器镜像，并在不同的容器 registry 之间分发这些镜像。要将 Skopeo 集成到 Kubernetes 工作流中，您需要在容器中运行它。
  - **Buildah** ：您希望在 Kubernetes 或 OpenShift CI/CD 系统中构建持续构建镜像的 OCI/container 镜像。在以前的版本中，人们使用 Docker 套接字连接到容器引擎，并执行 **docker build** 命令。这等同于为系统提供 root 访问权限而无需一个不安全的密码。因此，Red Hatrecommends 在容器中使用 Buildah。
- **不同的版本** ：
  - **All**:您在主机上运行旧的操作系统，但您想要运行最新版本的 Skopeo、Buildah 或 Podman。解决方法是在容器中运行容器工具。例如，这可用于在 Red Hat Enterprise Linux 7 容器主机上运行 Red Hat Enterprise Linux 8 中提供的容器工具的最新版本，而无需原生访问最新版本。
- **HPC 环境** ：
  - **All**:HPC 环境中的一个常见的限制是不允许非 root 用户在主机上安装软件包。当您在容器中运行 Skopeo、Buildah 或 Podman 时，您可以以非 root 用户身份执行这些特定的任务。

18.1. 在容器中运行 SKOPEO

您可以使用 Skopeo 检查远程容器镜像。在容器中运行 Skopeo 意味着容器根文件系统与主机 root 文件系统分离。要在主机和容器间共享或复制文件，您必须挂载文件和目录。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 登录到 registry.redhat.io 注册中心:

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: <password>
Login Succeeded!
```

2. 获取 **registry.redhat.io/rhel9/skopeo** 容器镜像：

```
$ podman pull registry.redhat.io/rhel9/skopeo
```

3. 使用 Skopeo 检查远程容器镜像 **registry.access.redhat.com/ubi9/ubi**:

```
$ podman run --rm registry.redhat.io/rhel9/skopeo \
  skopeo inspect docker://registry.access.redhat.com/ubi9/ubi
{
    "Name": "registry.access.redhat.com/ubi9/ubi",
    ...
    "Labels": {
        "architecture": "x86_64",
        ...
        "name": "ubi9",
        ...
        "summary": "Provides the latest release of Red Hat Universal Base Image 9.",
        "url":
"https://access.redhat.com/containers/#/registry.access.redhat.com/ubi9/images/8.2-347",
        ...
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
    ...
    ],
    "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "container=oci"
    ]
}
```

--rm 选项会在容器退出后删除 **registry.redhat.io/rhel9/skopeo** 镜像。

其他资源

- 如何在容器中运行 skopeo

18.2. 使用凭证在容器中运行 SKOPEO

使用容器 registry 需要进行身份验证才能访问和更改数据。Skopeo 支持各种指定凭证的方法。

通过这种方法，您可以使用 **--cred USERNAME[:PASSWORD]** 选项在命令行中指定凭据。

先决条件

- **container-tools** 元数据包已安装。

流程

- 使用 Skopeo 根据锁定的注册中心检查远程容器镜像：

```
$ podman run --rm registry.redhat.io/rhel9/skopeo inspect --creds
$USER:$PASSWORD docker://$IMAGE
```

其他资源

- 如何在容器中运行 skopeo

18.3. 使用 AUTHFILE 在容器中运行 SKOPEO

您可以使用身份验证文件（authfile）来指定凭证。**skopeo login** 命令登录到特定的注册中心，并将身份验证令牌存储在 authfile 中。使用 authfile 的优点是，无需重复输入凭证。

当在同一主机上运行时，所有容器工具，如 Skopeo、Buildah 和 Podman 共享相同的 authfile。在容器中运行 Skopeo 时，您必须通过在容器中卷挂载 authfile 来共享主机上的 authfile，或者您必须在容器内重新验证。

先决条件

- **container-tools** 元数据包已安装。

流程

- 使用 Skopeo 根据锁定的注册中心检查远程容器镜像：

```
$ podman run --rm -v $AUTHFILE:/auth.json registry.redhat.io/rhel9/skopeo inspect docker://$IMAGE
```

-v $AUTHFILE:/auth.json 选项会将 authfile 卷挂载在容器中的 /auth.json 处。Skopeo 现在可以访问主机上的 authfile 中的身份验证令牌，并获得对注册中心的安全访问。

其他 Skopeo 命令也可以工作，例如：

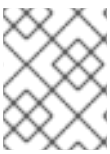
- 使用 **skopeo-copy** 命令，通过 **--source-creds** 和 **--dest-creds** 选项在命令行中为源和目标镜像指定凭据。它还读取 **/auth.json** authfile。
- 如果要为源和目标镜像指定单独的 authfile，请使用 **--source-authfile** 和 **--dest-authfile** 选项，并将这些 authfiles 从主机卷挂载到容器中。

其他资源

- 如何在容器中运行 skopeo

18.4. 将容器镜像复制到主机或从主机复制

Skopeo、Buildah 和 Podman 共享相同的本地容器镜像存储。如果要将容器复制到主机容器存储或从主机容器存储中复制，则需要将其挂载到 Skopeo 容器中。



注意

到主机容器存储的路径在 root 用户(**/var/lib/containers/storage**)和非 root 用户(**$HOME/.local/share/containers/storage**)之间有所不同。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 将 **registry.access.redhat.com/ubi9/ubi** 镜像复制到本地容器存储中：

```
$ podman run --privileged --rm -v
$HOME/.local/share/containers/storage:/var/lib/containers/storage \
registry.redhat.io/rhel9/skopeo skopeo copy \
docker://registry.access.redhat.com/ubi9/ubi containers-
storage:registry.access.redhat.com/ubi9/ubi
```

   - **privileged** 选项禁用所有安全机制。红帽建议仅在可信环境中使用这个选项。
   - 为了避免禁用安全机制，请将镜像导出到 tarball 或其它基于路径的镜像传输,并将其挂载到 Skopeo 容器中：
     - **$ podman save --format oci-archive -o oci.tar $IMAGE**
     - **$ podman run --rm -v oci.tar:/oci.tar registry.redhat.io/rhel9/skopeo copy oci-archive:/oci.tar $DESTINATION**

2. 可选：列出本地存储中的镜像：

```
$ podman images
REPOSITORY                              TAG     IMAGE ID      CREATED       SIZE
registry.access.redhat.com/ubi9/ubi     latest  ecbc6f53bba0  8 weeks ago   211 MB
```

其他资源

- 如何在容器中运行 skopeo

18.5. 在容器中运行 BUILDAH

流程演示了如何在容器中运行 Buildah，并根据镜像创建可正常工作的容器。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 登录到 registry.redhat.io 注册中心:

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: <password>
Login Succeeded!
```

2. 拉取并运行 **registry.redhat.io/rhel9/buildah** 镜像：

```
# podman run --rm --device /dev/fuse -it \
  registry.redhat.io/rhel9/buildah /bin/bash
```

   - **--rm** 选项会在容器退出后删除 **registry.redhat.io/rhel9/buildah** 镜像。
   - **device** 选项将主机设备添加到容器中。

- **sys_chroot** - 能够更改为不同的根目录。它不包含在容器的默认功能中。

3. 使用 **registry.access.redhat.com/ubi9** 镜像创建一个新容器：

```
# buildah from registry.access.redhat.com/ubi9
...
ubi9-working-container
```

4. 在 **ubi9-working-container** 容器中运行 **ls /** 命令：

```
# buildah run --isolation=chroot ubi9-working-container ls /
bin  boot  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv
```

5. 可选：列出本地存储中的所有镜像：

```
# buildah images
REPOSITORY                       TAG      IMAGE ID       CREATED       SIZE
registry.access.redhat.com/ubi9  latest   ecbc6f53bba0   5 weeks ago   211 MB
```

6. 可选：列出工作容器及其基础镜像：

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
0aaba7192762     *     ecbc6f53bba0 registry.access.redhat.com/ub... ubi9-working-container
```

7. 可选：将 **registry.access.redhat.com/ubi9** 镜像推送到 **registry.example.com** 上的本地注册中心：

```
# buildah push ecbc6f53bba0 registry.example.com:5000/ubi9/ubi
```

其他资源

- 在容器中运行 Buildah 的最佳实践

18.6. 特权和非特权 PODMAN 容器

默认情况下，Podman 容器是非特权的，例如无法修改主机上操作系统的某些部分。这是因为默认情况下，仅允许容器对设备进行有限的访问。

下表强调了特权容器的重要属性：您可以使用 **podman run --privileged *<image_name>*** 命令运行特权容器。

- 特权容器被授予与启动容器的用户相同的设备访问权限。
- 特权容器禁用将容器与主机隔离的安全功能。丢弃的功能、有限的设备、只读挂载点、Apparmor/SELinux 隔离和 Seccomp 过滤器都被禁用。
- 特权容器不能比启动它们的帐户具有更多的特权。

其他资源

- 如何将 --privileged 标志用于容器引擎

- **podman-run** 手册页

18.7. 运行带扩展权限的 PODMAN

如果您无法在 rootless 环境中运行工作负载，则需要以 root 用户身份运行这些工作负载。运行具有扩展特权的容器应谨慎，因为它禁用了所有安全功能。

先决条件

- **container-tools** 元数据包已安装。

流程

- 在 Podman 容器中运行 Podman 容器：

```
$ podman run --privileged --name=privileged_podman \
  registry.access.redhat.com//podman podman run ubi9 echo hello
Resolved "ubi9" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi9:latest...
...
Storing signatures
hello
```

- 根据 **registry.access.redhat.com/ubi9/podman** 镜像，运行名为 **privileged_podman** 的外部容器。
- **--privileged** 选项禁用将容器与主机隔离的安全功能。
- 运行 **podman run ubi9 echo hello** 命令，以基于 **ubi9** 镜像创建内容器。
- 请注意，**ubi9** 短镜像名称被解析为别名。因此，会拉取 **registry.access.redhat.com/ubi9:latest** 镜像。

验证

- 列出所有容器：

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND               CREATED         STATUS
PORTS   NAMES
52537876caf4  registry.access.redhat.com/ubi9/podman              podman run ubi9 e...  30
seconds ago     Exited (0) 13 seconds ago               privileged_podman
```

其他资源

- 如何在容器中使用 Podman
- **podman-run** 手册页

18.8. 运行具有较少特权的 PODMAN

您可以运行两个嵌套的 Podman 容器，而无需使用 **--privileged** 选项。运行不带 **--privileged** 选项的容器是一个更安全的选择。

如果您要以尽可能安全的方式尝试不同版本的 Podman，这非常有用。

先决条件

- **container-tools** 元数据包已安装。

流程

- 运行两个嵌套的容器：

```
$ podman run --name=unprivileged_podman --security-opt label=disable \
  --user podman --device /dev/fuse \
  registry.access.redhat.com/ubi9/podman \
  podman run ubi9 echo hello
```

- 根据 **registry.access.redhat.com/ubi9/podman** 镜像，运行名为 **unprivileged_podman** 的外部容器。
- **--security-opt label=disable** 选项禁用主机 Podman 上的 SELinux 隔离。SELinux 不允许容器化进程挂载在容器内运行所需的所有文件系统。
- **--user podman** 选项会自动使外部容器内的 Podman 在用户命名空间内运行。
- **--device /dev/fuse** 选项使用容器中的 **fuse-overlayfs** 软件包。此选项将 **/dev/fuse** 添加到外部容器中，因此容器中的 Podman 可以使用它。
- 运行 **podman run ubi9 echo hello** 命令，以基于 **ubi9** 镜像创建内容器。
- 请注意，ubi9 短镜像名称被解析为别名。因此，会拉取 **registry.access.redhat.com/ubi9:latest** 镜像。

验证

- 列出所有容器：

```
$ podman ps -a
CONTAINER ID  IMAGE                         COMMAND               CREATED         STATUS
PORTS   NAMES
a47b26290f43  podman run ubi9 e...  30 seconds ago  Exited (0) 13 seconds ago
unprivileged_podman
```

18.9. 在 PODMAN 容器内构建容器

您可以使用 Podman 在容器中运行容器。本例演示了如何使用 Podman 在此容器内构建并运行另一个容器。容器将运行"Moon-buggy"，这是一个基于文本的简单游戏。

先决条件

- **container-tools** 元数据包已安装。
- 登陆到 registry.redhat.io 注册表:

```
# podman login registry.redhat.io
```

流程

1. 根据 **registry.redhat.io/rhel9/podman** 镜像运行容器：

```
# podman run --privileged --name podman_container -it \
 registry.redhat.io/rhel9/podman /bin/bash
```

   - 根据 **registry.redhat.io/rhel9/podman** 镜像，运行名为 **podman_container** 的 outer 容器。
   - **--it** 选项指定您要在容器内运行交互式 bash shell。
   - **--privileged** 选项禁用将容器与主机隔离的安全功能。

2. 在 **podman_container** 容器中创建一个 **Containerfile**：

```
# vi Containerfile
FROM registry.access.redhat.com/ubi9/ubi
RUN dnf install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
RUN dnf -y install moon-buggy && dnf clean all
CMD ["/usr/bin/moon-buggy"]
```

   Containerfile 中的命令会导致以下构建命令：

   - 从 **registry.access.redhat.com/ubi9/ubi** 镜像构建容器。
   - 安装 **epel-release-latest-8.noarch.rpm** 软件包。
   - 安装 **moon-buggy** 软件包。
   - 设置容器命令。

3. 使用 **Containerfile** 构建名为 **moon-buggy** 的新容器镜像：

```
# podman build -t moon-buggy .
```

4. 可选：列出所有镜像：

```
# podman images
REPOSITORY                TAG      IMAGE ID      CREATED        SIZE
localhost/moon-buggy  latest  c97c58abb564  13 seconds ago  1.67 GB
registry.access.redhat.com/ubi9/ubi latest 4199acc83c6a  132seconds ago 213 MB
```

5. 运行一个基于 **moon-buggy** 容器的新容器：

```
# podman run -it --name moon moon-buggy
```

6. 可选：标记 **moon-buggy** 镜像：

```
# podman tag moon-buggy registry.example.com/moon-buggy
```

7. 可选：将 **moon-buggy** 镜像推送到 registry：

```
# podman push registry.example.com/moon-buggy
```

其他资源

- 技术预览：在容器内运行容器

第 19 章 使用 BUILDAH 构建容器镜像

Buildah 有助于构建符合 OCI 运行时规范 的 OCI 容器镜像。使用 Buildah，您可以创建一个可运行的容器，可以从头开始，也可以使用一个镜像作为起点。您可以使用 **Containerfile** 中的说明从工作容器创建镜像，或使用一系列 Buildah 命令来模拟 **Containerfile** 中的命令。

19.1. BUILDAH 工具

使用 Buildah 与使用 docker 命令构建镜像的不同之处在于：

无守护进程

Buildah 不需要容器运行时。

基础镜像或空镜像

您可以基于另一个容器构建镜像，或者从一个空镜像(scratch)开始。

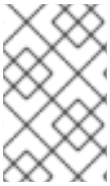
构建工具是外部的

Buildah 不包含镜像本身内的构建工具。因此，Buildah:

- 减少构建的镜像的大小。
- 通过从生成的镜像中排除软件（如 gcc、make 和 dnf）来提高镜像的安全性。
- 由于镜像大小减少了，所以允许使用较少的资源传输镜像。

兼容性

Buildah 支持使用 Dockerfile 构建容器镜像，从而可以轻松地从 Docker 迁移到 Buildah。



注意

Buildah 用于容器存储的默认位置与 CRI-O 容器引擎用于存储镜像本地副本的位置相同。因此，通过 CRI-O 或 Buildah 从注册表拉取的镜像或由 buildah 命令提交的镜像以同样的目录结构存储。但是，即使 CRI-O 和 Buildah 当前能够共享镜像，它们也无法共享容器。

其他资源

- Buildah - 有助于构建开放容器项目(OCI)容器镜像的工具
- Buildah 教程 1：构建 OCI 容器镜像
- Buildah 教程 2：使用带有容器 registry 的 Buildah
- 使用 Buildah 构建：Dockerfile、命令行或脚本
- 无根 Buildah 的工作原理：在非特权环境中构建容器

19.2. 安装 BUILDAH

使用 **dnf** 命令安装 Buildah 工具。

流程

- 安装 Buildah 工具：

```
# dnf -y install buildah
```

验证

- 显示帮助信息：

```
# buildah -h
```

19.3. 使用 BUILDAH 获取镜像

使用 **buildah from** 命令从头开始创建新的工作容器，或者以指定的镜像作为起点。

先决条件

- **container-tools** 元数据包已安装。

流程

- 根据 **registry.redhat.io/ubi9/ubi** 镜像，创建一个新的可正常工作的容器：

```
# buildah from registry.access.redhat.com/ubi9/ubi
Getting image source signatures
Copying blob…
Writing manifest to image destination
Storing signatures
ubi-working-container
```

验证

1. 列出本地存储中的所有镜像：

```
# buildah images
REPOSITORY                                TAG      IMAGE ID       CREATED       SIZE
registry.access.redhat.com/ubi9/ubi       latest   272209ff0ae5   2 weeks ago   234 MB
```

2. 列出工作容器及其基础镜像：

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
01eab9588ae1     *     272209ff0ae5 registry.access.redhat.com/ub... ubi-working-container
```

其他资源

- **buildah-from** 手册页
- **buildah-images** 手册页
- **buildah-containers** man page

19.4. 使用 BUILDAH 从 CONTAINERFILE 构建镜像

使用 **buildah bud** 命令按照说明从 **Containerfile** 构建镜像。

注意

如果在上下文目录中找到, **buildah bud** 命令会使用 **Containerfile** ，如果未找到，**buildah bud** 命令会使用 **Dockerfile** ；否则，可以使用 **--file** 来指定任何文件。可以在 **Containerfile** 和 **Dockerfile** 中使用的命令一样。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建 **Containerfile** ：

```
# cat Containerfile
FROM registry.access.redhat.com/ubi9/ubi
ADD myecho /usr/local/bin
ENTRYPOINT "/usr/local/bin/myecho"
```

2. 创建 **myecho** 脚本：

```
# cat myecho
echo "This container works!"
```

3. 更改 **myecho** 脚本的访问权限：

```
# chmod 755 myecho
```

4. 使用当前目录中的 **Containerfile** 构建 **myecho** 镜像：

```
# buildah bud -t myecho .
STEP 1: FROM registry.access.redhat.com/ubi9/ubi
STEP 2: ADD myecho /usr/local/bin
STEP 3: ENTRYPOINT "/usr/local/bin/myecho"
STEP 4: COMMIT myecho
...
Storing signatures
```

验证

1. 列出所有镜像：

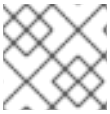
```
# buildah images
REPOSITORY                  TAG      IMAGE ID       CREATED              SIZE
localhost/myecho            latest   b28cd00741b3   About a minute ago   234 MB
```

2. 运行基于 **localhost/myecho** 镜像的 **myecho** 容器：

```
# podman run --name=myecho localhost/myecho
This container works!
```

3. 列出所有容器：

```
# podman ps -a
0d97517428d  localhost/myecho                          12 seconds ago  Exited (0) 13
seconds ago          myecho
```



注意

您可以使用 **podman history** 命令显示镜像中所使用的每个层的信息。

其他资源

- **buildah-bud** 手册页

19.5. 使用 BUILDAH 从头开始创建镜像

您可以创建一个仅包含最小容器元数据的新容器，而不是从基础镜像开始。

从全新容器中创建镜像时，请考虑：

- 您可以将不带依赖项的可执行文件复制到涂销空间镜像中，再进行一些配置设置才能使容器正常工作。
- 您必须初始化 RPM 数据库并在容器中添加发行版本软件包以使用 **dnf** 或 **rpm** 等工具。
- 如果您添加很多软件包，请考虑使用标准 UBI 或最小 UBI 镜像而不是 scratch 镜像。

先决条件

- **container-tools** 元数据包已安装。

流程

您可以向容器中添加 Web 服务 httpd，并将其配置为运行。

1. 创建一个空容器：

```
# buildah from scratch
working-container
```

2. 挂载 **working-container** 容器，并将挂载点路径保存到 **scratchmnt** 变量中：

```
# scratchmnt=$(buildah mount working-container)


# echo $scratchmnt
/var/lib/containers/storage/overlay/be2eaecf9f74b6acfe4d0017dd5534fde06b2fa8de9ed875691
f6ccc791c1836/merged
```

3. 在全新镜像中初始化 RPM 数据库，并添加 **redhat-release** 软件包：

```
# dnf install -y --releasever=8 --installroot=$scratchmnt redhat-release
```

4. 将 **httpd** 服务安装到 **scratch** 目录中：

```
# dnf install -y --setopt=reposdir=/etc/yum.repos.d \
    --installroot=$scratchmnt \
    --setopt=cachedir=/var/cache/dnf httpd
```

5. 创建 **$scratchmnt/var/www/html/index.html** 文件：

```
# mkdir -p $scratchmnt/var/www/html
# echo "Your httpd container from scratch works!" >
$scratchmnt/var/www/html/index.html
```

6. 配置 **working-container** 以直接从容器运行 **httpd** 守护进程：

```
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/myhttpd:latest
```

验证

1. 列出本地存储中的所有镜像：

```
# podman images
REPOSITORY                  TAG     IMAGE ID      CREATED         SIZE
localhost/myhttpd           latest  08da72792f60  2 minutes ago   121 MB
```

2. 运行 **localhost/myhttpd** 镜像，并配置容器和主机系统之间的端口映射：

```
# podman run -p 8080:80 -d --name myhttpd 08da72792f60
```

3. 测试 Web 服务器：

```
# curl localhost:8080
Your httpd container from scratch works!
```

其他资源

- **buildah-config** 手册页
- **buildah-commit** 手册页

19.6. 使用 BUILDAH 删除镜像

使用 **buildah rmi** 命令移除本地存储的容器镜像。您可以通过其 ID 或名称来删除镜像。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 列出本地系统上的所有镜像：

```
# buildah images
```

```
REPOSITORY                                TAG      IMAGE ID       CREATED          SIZE
localhost/johndoe/webserver               latest   dc5fcc610313   46 minutes ago   263 MB
docker.io/library/mynewecho               latest   fa2091a7d8b6   17 hours ago     234 MB
docker.io/library/myecho2                 latest   4547d2c3e436   6 days ago       234 MB
localhost/myecho                          latest   b28cd00741b3   6 days ago       234 MB
localhost/ubi-micro-httpd                 latest   c6a7678c4139   12 days ago      152 MB
registry.access.redhat.com/ubi9/ubi       latest   272209ff0ae5   3 weeks ago      234 MB
```

2. 删除 **localhost/myecho** 镜像：

```
# buildah rmi localhost/myecho
```

- 要删除多个镜像：

```
# buildah rmi docker.io/library/mynewecho docker.io/library/myecho2
```

- 要从您的系统中删除所有镜像：

```
# buildah rmi -a
```

- 要删除与其有多个名称（标记）关联的镜像，请添加 **-f** 选项来删除它们：

```
# buildah rmi -f localhost/ubi-micro-httpd
```

验证

- 确保镜像已被删除：

```
# buildah images
```

其他资源

- **buildah-rmi** 手册页

第 20 章 使用 BUILDAH 操作容器

使用 Buildah，您可以从命令行对容器镜像或容器进行几个操作。操作示例：从头开始或从容器镜像创建一个工作容器作为起点，从工作容器或使用 **Containerfile** 创建一个镜像，配置容器的入口点、标签、端口、shell 和工作目录。您可以挂载工作容器目录以进行文件系统操作，删除工作容器或容器镜像等。

然后，您可以从正常工作的容器创建镜像，并将镜像推送到注册中心。

20.1. 在容器内运行命令

使用 **buildah run** 命令从容器执行命令。

先决条件

- **container-tools** 元数据包已安装。
- 本地系统上提供了拉取的镜像。

流程

- 显示操作系统版本：

```
# buildah run ubi-working-container cat /etc/redhat-release
Red Hat Enterprise Linux release 8.4 (Ootpa)
```

其他资源

- **buildah-run** 手册页

20.2. 使用 BUILDAH 检查容器和镜像

使用 **buildah inspect** 命令显示有关容器或镜像的信息。

先决条件

- **container-tools** 元数据包已安装。
- 镜像是使用 Containerfile 中的指令构建的。详情请参阅 使用 Buildah 从 Containerfile 构建镜像一节。

流程

- 检查镜像：
  - 要检查 myecho 镜像，请输入：

```
# buildah inspect localhost/myecho
 {
    "Type": "buildah 0.0.1",
    "FromImage": "localhost/myecho:latest",
    "FromImageID":
"b28cd00741b38c92382ee806e1653eae0a56402bcd2c8d31bdcd36521bc267a4",
    "FromImageDigest":
```

```
"sha256:0f5b06cbd51b464fabe93ce4fe852a9038cdd7c7b7661cd7efef8f9ae8a59585",
    "Config":
    ...
     "Entrypoint": [
                "/bin/sh",
                "-c",
                "\"/usr/local/bin/myecho\""
           ],
    ...
}
```

- 要检查来自于 **myecho** 镜像的工作容器：

  i. 创建一个基于 **localhost/myecho** 镜像的可工作的容器：

  ```
  # buildah from localhost/myecho
  ```

  ii. 检查 **myecho-working-container** 容器：

  ```
  # buildah inspect ubi-working-container
  {
      "Type": "buildah 0.0.1",
      "FromImage": "registry.access.redhat.com/ubi8/ubi:latest",
      "FromImageID":
  "272209ff0ae5fe54c119b9c32a25887e13625c9035a1599feba654aa7638262d",
      "FromImageDigest":
  "sha256:77623387101abefbf83161c7d5a0378379d0424b2244009282acb39d42f1fe13
  ",
      "Config":
      ...
  "Container": "ubi-working-container",
  "ContainerID":
  "01eab9588ae1523746bb706479063ba103f6281ebaeeccb5dc42b70e450d5ad0",
  "ProcessLabel": "system_u:system_r:container_t:s0:c162,c1000",
  "MountLabel": "system_u:object_r:container_file_t:s0:c162,c1000",
  ...
  }
  ```

其他资源

- **buildah-inspect** 手册页

20.3. 使用 BUILDAH MOUNT 修改容器

使用 **buildah mount** 命令显示容器或镜像的信息。

先决条件

- **container-tools** 元数据包已安装。
- 使用 Containerfile 中的指令构建的镜像。详情请参阅 使用 Buildah 从 Containerfile 构建镜像 一节。

流程

1. 创建一个基于 **registry.access.redhat.com/ubi8/ubi** 镜像的可工作容器，并将容器名称保存到 **mycontainer** 变量中：

```
# mycontainer=$(buildah from localhost/myecho)

# echo $mycontainer
myecho-working-container
```

2. 挂载 **myecho-working-container** 容器，并将挂载点路径保存到 **mymount** 变量中：

```
# mymount=$(buildah mount $mycontainer)

# echo $mymount
/var/lib/containers/storage/overlay/c1709df40031dda7c49e93575d9c8eebcaa5d8129033a58e5
b6a95019684cc25/merged
```

3. 修改 **myecho** 脚本并使其可执行：

```
# echo 'echo "We modified this container."' >> $mymount/usr/local/bin/myecho
# chmod +x $mymount/usr/local/bin/myecho
```

4. 从 **myecho-working-container** 容器创建 **myecho2** 镜像：

```
# buildah commit $mycontainer containers-storage:myecho2
```

验证

1. 列出本地存储中的所有镜像：

```
# buildah images
REPOSITORY                    TAG      IMAGE ID       CREATED          SIZE
docker.io/library/myecho2     latest   4547d2c3e436   4 minutes ago    234 MB
localhost/myecho              latest   b28cd00741b3   56 minutes ago   234 MB
```

2. 运行基于 **docker.io/library/myecho2** 镜像的 **myecho2** 容器：

```
# podman run --name=myecho2 docker.io/library/myecho2
This container works!
We even modified it.
```

其他资源

- **buildah-mount** 手册页
- **buildah-commit** 手册页

20.4. 使用 BUILDAH COPY 和 BUILDAH CONFIG 修改容器

使用 **buildah copy** 命令在不挂载的情况下将文件复制到容器。然后，您可以使用 **buildah config** 命令配置容器，以运行默认创建的脚本。

先决条件

- **container-tools** 元数据包已安装。
- 使用 Containerfile 中的指令构建的镜像。详情请参阅 使用 Buildah 从 Containerfile 构建镜像 一节。

流程

1. 创建一个名为 **newecho** 的脚本，并使其可执行：

```
# cat newecho
echo "I changed this container"
# chmod 755 newecho
```

2. 创建新的可正常工作的容器：

```
# buildah from myecho:latest
myecho-working-container-2
```

3. 将 newecho 脚本复制到容器中的 **/usr/local/bin** 目录中：

```
# buildah copy myecho-working-container-2 newecho /usr/local/bin
```

4. 更改配置以使用 **newecho** 脚本作为新入口点：

```
# buildah config --entrypoint "/bin/sh -c /usr/local/bin/newecho" myecho-working-container-2
```

5. 可选：运行 **myecho-working-container-2** 容器， 其会触发 **newecho** 脚本执行：

```
# buildah run myecho-working-container-2 -- sh -c '/usr/local/bin/newecho'
I changed this container
```

6. 将 **myecho-working-container-2** 容器提交成名为 **mynewecho** 的新镜像：

```
# buildah commit myecho-working-container-2 containers-storage:mynewecho
```

验证

- 列出本地存储中的所有镜像：

```
# buildah images
REPOSITORY                      TAG      IMAGE ID       CREATED          SIZE
docker.io/library/mynewecho     latest   fa2091a7d8b6   8 seconds ago    234 MB
```

其他资源

- **buildah-copy** 手册页
- **buildah-config** 手册页
- **buildah-commit** 手册页
- **buildah-run** 手册页

20.5. 将容器推送到私有 REGISTRY

使用 **buildah push** 命令将镜像从本地存储推送到公共或私有存储库。

先决条件

- **container-tools** 元数据包已安装。
- 镜像是使用 Containerfile 中的指令构建的。详情请参阅 使用 Buildah 从 Containerfile 构建镜像 一节。

流程

1. 在机器上创建本地注册表：

```
# podman run -d -p 5000:5000 registry:2
```

2. 将 **myecho:latest** 镜像推送到 **localhost** 注册表:

```
#  buildah push --tls-verify=false myecho:latest localhost:5000/myecho:latest
Getting image source signatures
Copying blob sha256:e4efd0...
...
Writing manifest to image destination
Storing signatures
```

验证

1. 列出 **localhost** 存储库中的所有镜像：

```
# curl http://localhost:5000/v2/_catalog
{"repositories":["myecho2]}


# curl http://localhost:5000/v2/myecho2/tags/list
{"name":"myecho","tags":["latest"]}
```

2. 检查 **docker://localhost:5000/myecho:latest** 镜像：

```
# skopeo inspect --tls-verify=false docker://localhost:5000/myecho:latest | less
{
    "Name": "localhost:5000/myecho",
    "Digest": "sha256:8999ff6050...",
    "RepoTags": [
        "latest"
    ],
    "Created": "2021-06-28T14:44:05.919583964Z",
    "DockerVersion": "",
    "Labels": {
        "architecture": "x86_64",
        "authoritative-source-url": "registry.redhat.io",
    ...
}
```

3. 拉取 **localhost:5000/myecho** 镜像：

```
# podman pull --tls-verify=false localhost:5000/myecho2
# podman run localhost:5000/myecho2
This container works!
```

其他资源

- **buildah-push** 手册页

20.6. 将容器推送到 DOCKER HUB

使用 Docker Hub 凭证通过 **buildah** 命令从 Docker Hub 推送和拉取镜像。

先决条件

- **container-tools** 元数据包已安装。
- 使用 Containerfile 中的指令构建的镜像。详情请参阅 使用 Buildah 从 Containerfile 构建镜像 一节。

流程

1. 将 **docker.io/library/myecho:latest** 推送到 Docker Hub。将 ***username*** 和 ***password*** 替换为您的 Docker Hub 凭证：

```
# buildah push --creds username:password \
  docker.io/library/myecho:latest docker://testaccountXX/myecho:latest
```

验证

- 获取并运行 **docker.io/testaccountXX/myecho:latest** 镜像：
  - 使用 Podman 工具：

```
# podman run docker.io/testaccountXX/myecho:latest
This container works!
```

  - 使用 Buildah 和 Podman 工具：

```
# buildah from docker.io/testaccountXX/myecho:latest
myecho2-working-container-2
# podman run myecho-working-container-2
```

其他资源

- **buildah-push** 手册页

20.7. 使用 BUILDAH 删除容器

使用 **buildah rm** 命令删除容器。您可以使用容器 ID 或名称来指定要移除的容器。

先决条件

先决条件

- **container-tools** 元数据包已安装。
- 至少一个容器已经停止。

流程

1. 列出所有容器：

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
05387e29ab93     *     c37e14066ac7 docker.io/library/myecho:latest  myecho-working-
container
```

2. 删除 myecho-working-container 容器：

```
# buildah rm myecho-working-container
05387e29ab93151cf52e9c85c573f3e8ab64af1592b1ff9315db8a10a77d7c22
```

验证

- 确保容器已被删除：

```
# buildah containers
```

其他资源

- **buildah-rm** 手册页

第 21 章 监控容器

使用 Podman 命令管理 Podman 环境。通过这种方式，您可以通过显示系统和 pod 信息，以及监控 Podman 事件来确定容器的健康状况。

21.1. 在容器上使用健康检查

您可以使用健康检查来确定容器中运行的进程的健康状态或就绪状态。

如果健康检查成功，容器将标记为 "healthy"；否则，其状态为"unhealthy"。您可以将健康检查与运行 **podman exec** 命令进行比较，并检查退出代码。零退出值表示容器为 "healthy"。

在使用 **Containerfile** 中的 **HEALTHCHECK** 指令或在命令行中创建容器时，可以设置健康检查。您可以使用 **podman inspect** 或 **podman ps** 命令显示容器的 health-check 状态。

健康检查由六个基本组件组成：

- Command（命令）
- Retries（重试）
- Interval（间隔）
- Start-period（开始期间）
- Timeout（超时）
- 容器恢复

健康检查组件的描述如下：

命令 (--health-cmd 选项)

Podman 在目标容器内执行命令并等待退出代码。

其他五个组件与健康检查的调度相关，它们是可选的。

重试 (--health-retries 选项)

定义容器被标记为"unhealthy"前需要连续失败的健康检查数量。成功健康检查会重置重试计数器。

间隔 (--health-interval 选项)

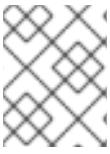
描述运行 healthcheck 命令之间的时间。请注意：小的间隔会导致系统花费很多时间运行健康检查。间隔时间太长会导致难以捕捉超时。

start-period (--health-start-period 选项)

描述容器启动和要忽略健康检查失败的时间间隔。

超时 (--health-timeout 选项)

描述健康检查在被视为不成功前必须完成的时间。



注意

Retries、Interval 和 Start-period 组件的值是持续时间，如 "30s" 或 "1h15m"。有效的时间单位是"ns"、"us"或"ãs"，"ms"，"s"、"m"和"h"。

容器恢复 (--health-on-failure 选项)

决定在容器状态不健康时要执行的操作。当应用程序失败时，Podman 会自动重启它以提供稳健性。**--health-on-failure** 选项支持四个操作：

- **none**:不执行任何操作，这是默认操作。
- **kill**:终止容器。
- **restart**:重启容器。
- **stop**:停止容器。

注意

Podman 版本 4.2 及更高版本中提供了 **--health-on-failure** 选项。

警告

不要将 **restart** 操作与 **--restart** 选项一起使用。在 **systemd** 单元内运行时，请考虑使用 **kill** 或 **stop** 操作来使用 **systemd** 重启策略。

健康检查在容器内运行。只有在您知道该服务的健康状态，并可以区分成功和不成功健康检查时，健康检查才有意义。

其他资源

- **podman-healthcheck** man page
- **podman-run** 手册页
- 位于边缘处的 podman ：使用自定义的健康检查操作来保持服务正常运行
- 使用 Podman 监控容器严重性和可用性

21.2. 使用命令行执行健康检查

在命令行中创建容器时，您可以设置健康检查。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 定义健康检查：

```
$ podman run -dt --name=hc-container -p 8080:8080 --health-cmd='curl
http://localhost:8080 || exit 1' --health-interval=0
registry.access.redhat.com/ubi8/httpd-24
```

- **--health-cmd** 选项为容器设置健康检查命令。
- 带有 0 值的 **--health-interval=0** 选项表示您要手动运行健康检查。

2. 检查 **hc-container** 容器的健康状态：

- 使用 **podman inspect** 命令：

```
$ podman inspect --format='{{json .State.Health.Status}}' hc-container
healthy
```

- 使用 **podman ps** 命令：

```
$ podman ps
CONTAINER ID  IMAGE                            COMMAND               CREATED      STATUS
PORTS       NAMES
a680c6919fe  localhost/hc-container:latest  /usr/bin/run-http...  2 minutes ago  Up 2
minutes (healthy) hc-container
```

- 使用 **podman healthcheck run** 命令：

```
$ podman healthcheck run hc-container
healthy
```

其他资源

- **podman-healthcheck** man page
- **podman-run** 手册页
- 位于边缘处的 podman ：使用自定义的健康检查操作来保持服务正常运行
- 使用 Podman 监控容器严重性和可用性

21.3. 使用 CONTAINERFILE 执行健康检查

您可以使用 **Containerfile** 中的 **HEALTHCHECK** 指令来设置健康检查。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建 **Containerfile** ：

```
$ cat Containerfile
FROM registry.access.redhat.com/ubi8/httpd-24
EXPOSE 8080
HEALTHCHECK CMD curl http://localhost:8080 || exit 1
```

注意

HEALTHCHECK 指令仅支持 **docker** 镜像格式。对于 **oci** 镜像格式，该指令会被忽略。

2. 构建容器镜像并添加镜像名称：

```
$ podman build --format=docker -t hc-container .
STEP 1/3: FROM registry.access.redhat.com/ubi8/httpd-24
STEP 2/3: EXPOSE 8080
--> 5aea97430fd
STEP 3/3: HEALTHCHECK CMD curl http://localhost:8080 || exit 1
COMMIT health-check
Successfully tagged localhost/health-check:latest
a680c6919fe6bf1a79219a1b3d6216550d5a8f83570c36d0dadfee1bb74b924e
```

3. 运行容器：

```
$ podman run -dt --name=hc-container localhost/hc-container
```

4. 检查 **hc-container** 容器的健康状态：

- 使用 **podman inspect** 命令：

```
$ podman inspect --format='{{json .State.Health.Status}}' hc-container
healthy
```

- 使用 **podman ps** 命令：

```
$ podman ps
CONTAINER ID  IMAGE                            COMMAND               CREATED      STATUS
PORTS       NAMES
a680c6919fe  localhost/hc-container:latest  /usr/bin/run-http...  2 minutes ago  Up 2
minutes (healthy) hc-container
```

- 使用 **podman healthcheck run** 命令：

```
$ podman healthcheck run hc-container
healthy
```

其他资源

- **podman-healthcheck** man page
- **podman-run** 手册页
- 位于边缘处的 podman ：使用自定义的健康检查操作来保持服务正常运行
- 使用 Podman 监控容器严重性和可用性

21.4. 显示 PODMAN 系统信息

podman system 命令允许您通过显示系统信息来管理 Podman 系统。

先决条件

- **container-tools** 元数据包已安装。

流程

- 显示 Podman 系统信息：
  - 要显示 Podman 磁盘用量，请输入：

```
$ podman system df
TYPE           TOTAL       ACTIVE      SIZE        RECLAIMABLE
Images         3           2           1.085GB     233.4MB (0%)
Containers     2           0           28.17kB     28.17kB (100%)
Local Volumes  3           0           0B          0B (0%)
```

  - 要显示空间使用情况的详细信息，请输入：

```
$ podman system df -v
Images space usage:

REPOSITORY                            TAG         IMAGE ID      CREATED     SIZE
SHARED SIZE  UNIQUE SIZE  CONTAINERS
registry.access.redhat.com/ubi9       latest      b1e63aaae5cf  13 days     233.4MB
233.4MB      0B           0
registry.access.redhat.com/ubi9/httpd-24  latest      0d04740850e8  13 days     461.5MB
0B           461.5MB      1
registry.redhat.io/rhel8/podman       latest      dce10f591a2d  13 days     390.6MB
233.4MB      157.2MB      1

Containers space usage:

CONTAINER ID  IMAGE         COMMAND                 LOCAL VOLUMES  SIZE
CREATED     STATUS      NAMES
311180ab99fb  0d04740850e8  /usr/bin/run-httpd      0              28.17kB     16 hours
exited      hc1
bedb6c287ed6  dce10f591a2d  podman run ubi9 echo hello  0          0B          11 hours
configured  dazzling_tu

Local Volumes space usage:

VOLUME NAME                                                 LINKS       SIZE
76de0efa83a3dae1a388b9e9e67161d28187e093955df185ea228ad0b3e435d0  0
0B
8a1b4658aecc9ff38711a2c7f2da6de192c5b1e753bb7e3b25e9bf3bb7da8b13  0
0B
d9cab4f6ccbcf2ac3cd750d2efff9d2b0f29411d430a119210dd242e8be20e26  0           0B
```

  - 要显示主机、当前存储统计和 Podman 构建的信息，请输入：

```
$ podman system info
host:
  arch: amd64
  buildahVersion: 1.22.3
  cgroupControllers: []
```

```
  cgroupManager: cgroupfs
  cgroupVersion: v1
  conmon:
    package: conmon-2.0.29-1.module+el8.5.0+12381+e822eb26.x86_64
    path: /usr/bin/conmon
    version: 'conmon version 2.0.29, commit:
7d0fa63455025991c2fc641da85922fde889c91b'
  cpus: 2
  distribution:
    distribution: '"rhel"'
    version: "8.5"
  eventLogger: file
  hostname: localhost.localdomain
  idMappings:
    gidmap:
    - container_id: 0
      host_id: 1000
      size: 1
    - container_id: 1
      host_id: 100000
      size: 65536
    uidmap:
    - container_id: 0
      host_id: 1000
      size: 1
    - container_id: 1
      host_id: 100000
      size: 65536
  kernel: 4.18.0-323.el8.x86_64
  linkmode: dynamic
  memFree: 352288768
  memTotal: 2819129344
  ociRuntime:
    name: runc
    package: runc-1.0.2-1.module+el8.5.0+12381+e822eb26.x86_64
    path: /usr/bin/runc
    version: |-
      runc version 1.0.2
      spec: 1.0.2-dev
      go: go1.16.7
      libseccomp: 2.5.1
  os: linux
  remoteSocket:
    path: /run/user/1000/podman/podman.sock
  security:
    apparmorEnabled: false
    capabilities:
CAP_NET_RAW,CAP_CHOWN,CAP_DAC_OVERRIDE,CAP_FOWNER,CAP_FSETID,C
AP_KILL,CAP_NET_BIND_SERVICE,CAP_SETFCAP,CAP_SETGID,CAP_SETPCAP,CA
P_SETUID,CAP_SYS_CHROOT
    rootless: true
    seccompEnabled: true
    seccompProfilePath: /usr/share/containers/seccomp.json
    selinuxEnabled: true
  serviceIsRemote: false
  slirp4netns:
```

```
    executable: /usr/bin/slirp4netns
    package: slirp4netns-1.1.8-1.module+el8.5.0+12381+e822eb26.x86_64
    version: |-
      slirp4netns version 1.1.8
      commit: d361001f495417b880f20329121e3aa431a8f90f
      libslirp: 4.4.0
      SLIRP_CONFIG_VERSION_MAX: 3
      libseccomp: 2.5.1
  swapFree: 3113668608
  swapTotal: 3124752384
  uptime: 11h 24m 12.52s (Approximately 0.46 days)
registries:
  search:
  - registry.fedoraproject.org
  - registry.access.redhat.com
  - registry.centos.org
  - docker.io
store:
  configFile: /home/user/.config/containers/storage.conf
  containerStore:
    number: 2
    paused: 0
    running: 0
    stopped: 2
  graphDriverName: overlay
  graphOptions:
    overlay.mount_program:
      Executable: /usr/bin/fuse-overlayfs
      Package: fuse-overlayfs-1.7.1-1.module+el8.5.0+12381+e822eb26.x86_64
      Version: |-
        fusermount3 version: 3.2.1
        fuse-overlayfs: version 1.7.1
        FUSE library version 3.2.1
        using FUSE kernel interface version 7.26
  graphRoot: /home/user/.local/share/containers/storage
  graphStatus:
    Backing Filesystem: xfs
    Native Overlay Diff: "false"
    Supports d_type: "true"
    Using metacopy: "false"
  imageStore:
    number: 3
  runRoot: /run/user/1000/containers
  volumePath: /home/user/.local/share/containers/storage/volumes
version:
  APIVersion: 3.3.1
  Built: 1630360721
  BuiltTime: Mon Aug 30 23:58:41 2021
  GitCommit: ""
  GoVersion: go1.16.7
  OsArch: linux/amd64
  Version: 3.3.1
```

- 要删除所有未使用的容器、镜像和卷数据，请输入：

```
$ podman system prune
```

```
WARNING! This will remove:
        - all stopped containers
        - all stopped pods
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
```

- **podman system prune** 命令删除所有未使用的容器（悬停和未引用的）、pod 以及可选的来自本地存储的卷。
- 使用 **--all** 选项删除所有未使用的镜像。未使用的镜像是悬停的镜像，以及没有任何容器基于该镜像的镜像。
- 使用 **--volume** 选项来修剪卷。默认情况下，如果没有容器使用该卷，也不会删除卷以防止重要数据被删除。

其他资源

- **podman-system-df** man page
- **podman-system-info** man page
- **podman-system-prune** man page

21.5. PODMAN 事件类型

您可以监控 Podman 中发生的事件。存在多个事件类型，每个事件类型都会报告不同的状态。

*容器*事件类型会报告以下状态：

- attach
- checkpoint
- cleanup
- commit
- create
- exec
- export
- import
- init
- kill
- mount
- pause
- prune
- remove

- restart
- restore
- start
- stop
- sync
- unmount
- unpause

pod 事件类型报告以下状态：

- create
- kill
- pause
- remove
- start
- stop
- unpause

镜像 事件类型会报告以下状态：

- prune
- push
- pull
- save
- remove
- tag
- untag

*系统*类型报告以下状态：

- refresh
- renumber

*卷*类型报告以下状态：

- create
- prune

- remove

其他资源

- **podman-events** man page

21.6. 监控 PODMAN 事件

您可以使用 **podman events** 命令监控和打印 Podman 中发生的事件。每个事件都将包括一个时间戳、类型、状态、名称、镜像（如果适用的话）。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 运行 **myubi** 容器：

```
$ podman run -q --rm --name=myubi registry.access.redhat.com/ubi8/ubi:latest
```

2. 显示 Podman 事件：
   - 要显示所有 Podman 事件，请输入：

```
$ now=$(date --iso-8601=seconds)
$ podman events --since=now --stream=false
2023-03-08 14:27:20.696167362 +0100 CET container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi,...)
2023-03-08 14:27:20.652325082 +0100 CET image pull
registry.access.redhat.com/ubi8/ubi:latest
2023-03-08 14:27:20.795695396 +0100 CET container init
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.809205161 +0100 CET container start
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.809903022 +0100 CET container attach
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.831710446 +0100 CET container died
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.913786892 +0100 CET container remove
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
```

--stream=false 选项可确保 **podman events** 命令在读取最后一个已知事件时退出。

您可以在输入 **podman run** 命令时看到几个发生的事件：

- 创建新容器时发生 **container create**。
- 如果本地存储中不存在容器镜像，则拉取镜像时会发生 **image pull**。

- 在运行时初始化容器并设置网络时，会发生 **container init**。
- 在启动容器时会发生 **container start**。
- 当附加到容器的终端时，会发生 **container attach**。这是因为容器在前台运行。
- 当容器退出时，会发出 **container died**。
- 因为 **--rm** 标志用于在容器退出后删除容器，所以会发生 **container remove**。

- 您还可以使用 **journalctl** 命令显示 Podman 事件：

```
$ journalctl --user -r SYSLOG_IDENTIFIER=podman
Mar 08 14:27:20 fedora podman[129324]: 2023-03-08 14:27:20.913786892 +0100 CET
m=+0.066920979 container remove
...
Mar 08 14:27:20 fedora podman[129289]: 2023-03-08 14:27:20.696167362 +0100 CET
m=+0.079089208 container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72f>
```

- 要只显示 Podman 创建事件，请输入：

```
$ podman events --filter event=create
2023-03-08 14:27:20.696167362 +0100 CET container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi,...)
```

- 您还可以使用 **journalctl** 命令显示 Podman 创建事件：

```
$ journalctl --user -r PODMAN_EVENT=create
Mar 08 14:27:20 fedora podman[129289]: 2023-03-08 14:27:20.696167362 +0100 CET
m=+0.079089208 container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72f>
```

其他资源

- **podman-events** 手册页
- 容器事件和审核

21.7. 使用 PODMAN 事件进行审核

在以前的版本中，必须连接到事件才能正确解释事件。例如，**container-create** 事件必须与 **image-pull** 事件相关联，以了解已使用了哪个镜像。**container-create** 事件也不包含所有数据，如安全设置、卷、挂载等。

从 Podman v4.4 开始，您可以直接从单个事件和 **journald** 条目中收集关于容器的所有相关信息。数据采用 JSON 格式，与 **podman container inspect** 命令相同，并包含容器的所有配置和安全设置。您可以配置 Podman 来附加容器检查数据以进行审核。

先决条件

- **container-tools** 元数据包已安装。

流程

流程

1. 修改 **~/.config/containers/containers.conf** 文件，并将 **events_container_create_inspect_data=true** 选项添加到 **[engine]** 部分：

```
$ cat ~/.config/containers/containers.conf
[engine]
events_container_create_inspect_data=true
```

对于系统范围的配置，修改 **/etc/containers/containers.conf** 或 **/usr/share/container/containers.conf** 文件。

2. 创建容器：

```
$ podman create registry.access.redhat.com/ubi8/ubi:latest
19524fe3c145df32d4f0c9af83e7964e4fb79fc4c397c514192d9d7620a36cd3
```

3. 显示 Podman 事件：

- 使用 **podman events** 命令：

```
$ now=$(date --iso-8601=seconds)
$ podman events --since $now --stream=false --format "{{.ContainerInspectData}}"
| jq “.Config.CreateCommand"
[
  "/usr/bin/podman",
  "create",
  "registry.access.redhat.com/ubi8"
]
```

  - **--format "{{.ContainerInspectData}}"** 选项显示检查数据。
  - **jq ".Config.CreateCommand"** 将 JSON 数据转换为更易读的格式，并显示 **podman create** 命令的参数。

- 使用 **journalctl** 命令：

```
$ journalctl --user -r PODMAN_EVENT=create --all -o json | jq
".PODMAN_CONTAINER_INSPECT_DATA | fromjson" | jq
".Config.CreateCommand"
[
  "/usr/bin/podman",
  "create",
  "registry.access.redhat.com/ubi8"
]
```

podman events 和 **journalctl** 命令的输出数据是一样的。

其他资源

- **podman-events** 手册页
- **containers.conf** 手册页
- 容器事件和审核

第 22 章 创建并恢复容器检查点

Checkpoint/Restore in Userspace（CRIU）是一个软件，可让您在正在运行的容器或独立应用程序中设置检查点，并将它的状态保存到磁盘中。您可以在重启时同时使用保存的数据来恢复容器。

警告

内核不支持 AArch64 上的预复制检查点。

22.1. 本地创建并恢复容器检查点

这个示例基于 Python 的 web 服务器，该服务器会返回一个整数，它会在每个请求后递增。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建基于 Python 服务器：

```
# cat counter.py
#!/usr/bin/python3

import http.server

counter = 0

class handler(http.server.BaseHTTPRequestHandler):
    def do_GET(s):
        global counter
        s.send_response(200)
        s.send_header('Content-type', 'text/html')
        s.end_headers()
        s.wfile.write(b'%d\n' % counter)
        counter += 1


server = http.server.HTTPServer(('', 8088), handler)
server.serve_forever()
```

2. 使用以下定义创建容器：

```
# cat Containerfile
FROM registry.access.redhat.com/ubi9/ubi

COPY counter.py /home/counter.py

RUN useradd -ms /bin/bash counter
```

```
RUN dnf -y install python3 && chmod 755 /home/counter.py

USER counter
ENTRYPOINT /home/counter.py
```

容器基于通用基础镜像（UBI 8）并使用基于 Python 的服务器。

3. 构建容器：

```
# podman build . --tag counter
```

文件 **counter.py** 和 **Containerfile** 是容器构建进程（**podman build**）的输入。构建的镜像存储在本地，并使用标签 **counter** 进行标记。

4. 以 root 用户身份启动容器：

```
# podman run --name criu-test --detach counter
```

5. 要列出所有正在运行的容器，请输入：

```
# podman ps
CONTAINER ID  IMAGE  COMMAND  CREATED   STATUS  PORTS NAMES
e4f82fd84d48  localhost/counter:latest  5 seconds ago  Up 4 seconds ago  criu-test
```

6. 显示容器的 IP 地址：

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

7. 将请求发送到容器：

```
# curl 10.88.0.247:8088
0
# curl 10.88.0.247:8088
1
```

8. 为容器创建一个检查点：

```
# podman container checkpoint criu-test
```

9. 重启系统：

10. 恢复容器：

```
# podman container restore --keep criu-test
```

11. 将请求发送到容器：

```
# curl 10.88.0.247:8080
2
# curl 10.88.0.247:8080
```

```
3
# curl 10.88.0.247:8080
4
```

现在，这个结果不会再次从 **0** 开始，而是从之前的值开始。

这样，您可以通过重新引导轻松保存完整的容器状态。

其他资源

- Podman 检查点

22.2. 使用容器恢复减少启动时间

您可以使用容器迁移来减少容器的启动时间，这需要一定时间进行初始化。通过使用检查点，您可以在同一个主机上或不同的主机上多次恢复容器。本例是基于 在本地创建和恢复容器检查点 的容器。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 创建容器的检查点，并将检查点镜像导出到 **tar.gz** 文件中：

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

2. 从 **tar.gz** 文件中恢复容器：

```
# podman container restore --import /tmp/chkpt.tar.gz --name counter1
# podman container restore --import /tmp/chkpt.tar.gz --name counter2
# podman container restore --import /tmp/chkpt.tar.gz --name counter3
```

--name(**-n**)选项为从导出的检查点恢复的容器指定一个新名称。

3. 显示每个容器的 ID 和名称：

```
# podman ps -a --format "{{.ID}} {{.Names}}"
a8b2e50d463c counter3
faabc5c27362 counter2
2ce648af11e5 counter1
```

4. 显示每个容器的 IP 地址：

```
#  podman inspect counter1 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.248

#  podman inspect counter2 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.249

#  podman inspect counter3 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.250
```

5. 将请求发送到每个容器：

```
# curl 10.88.0.248:8080
4
# curl 10.88.0.249:8080
4
# curl 10.88.0.250:8080
4
```

请注意，在所有情况下结果都是 **4**，因为您正在使用从同一检查点恢复的不同容器。

使用这种方法，您可以快速启动进行了初始检查点的容器的有状态副本。

其他资源

- 在 RHEL 上使用 Podman 进行容器迁移

22.3. 在系统间迁移容器

您可以将正在运行的容器从一个系统迁移到另一个系统，而不丢失容器中运行的应用程序的状态。本示例基于本地创建并恢复一个容器检测点 中的内容，使用 **counter** tag。

重要

只有在系统配置完全匹配时，才支持使用 **podman container checkpoint** 和 **podman container restore** 命令在系统间迁移容器，如下所示：

- Podman 版本
- OCI 运行时(runc/crun)
- 网络堆栈(CNI/Netavark)
- Cgroups 版本
- 内核版本
- CPU 功能

您可以迁移到具有更多功能的 CPU，但不能迁移到没有您使用的特定功能的 CPU。低级工具执行检查点(CRIU)可能检查 CPU 功能兼容性 ：https://criu.org/Cpuinfo。

先决条件

- **container-tools** 元数据包已安装。
- 如果容器被推送到注册表，则不需要执行下列步骤，因为 Podman 将自动从注册表下载容器（如果本地没有的话）。本例没有使用注册中心，您必须导出之前构建并标记的容器（请参阅 在本地创建和恢复容器检查点）。
  - 导出之前构建的容器：

```
# podman save --output counter.tar counter
```

  - 将导出的容器镜像复制到目标系统(***other_host)***：

```
# scp counter.tar other_host:
```

- 在目的系统中导入导出的容器：

```
# ssh other_host podman load --input counter.tar
```

现在，此容器迁移的目标系统在其本地容器存储中有一个同样的容器镜像。

流程

1. 以 root 用户身份启动容器：

```
# podman run --name criu-test --detach counter
```

2. 显示容器的 IP 地址：

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

3. 将请求发送到容器：

```
# curl 10.88.0.247:8080
0
# curl 10.88.0.247:8080
1
```

4. 创建容器的检查点，并将检查点镜像导出到 **tar.gz** 文件中：

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

5. 将 checkpoint 归档复制到目标主机：

```
# scp /tmp/chkpt.tar.gz other_host:/tmp/
```

6. 在目标主机上恢复检查点(***other_host***)：

```
# podman container restore --import /tmp/chkpt.tar.gz
```

7. 向目标主机上的容器(***other_host***)发送一个请求：

```
# *curl 10.88.0.247:8080*
2
```

因此，有状态的容器已从一个系统迁移到另一个系统，而不丢失其状态。

其他资源

- 在 RHEL 上使用 Podman 进行容器迁移

第 23 章 使用 TOOLBX 进行开发和故障排除

在系统中安装软件会带来某些风险：它可以改变系统的行为，并在不再需要后保留不需要的文件和目录。您可以通过将最喜欢的开发和调试工具、编辑器和软件开发工具包(SDK)安装到 Toolbx 完全可变的容器中来防止这些风险，而不影响基础操作系统。您可以使用命令（如 **less**、**lsof**、**rsync**、**ssh**、**sudo** 和 **unzip** ）在主机系统上执行更改。

Toolbx 工具执行以下操作：

1. 将 **registry.access.redhat.com/ubi9/toolbox:latest** 镜像拉取到本地系统
2. 从镜像启动容器
3. 在容器内运行 shell，从中可以访问主机系统

注意

toolbx 可以运行根容器或无根容器，具体取决于创建 Toolbx 容器的用户的权限。需要主机系统上 root 权利的实用程序也应在 root 容器中运行。

默认容器名称是 **rhel-toolbox**。

23.1. 启动 TOOLBX 容器

您可以使用 **toolbox create** 命令创建 Toolbx 容器。然后您可以使用 **toolbox enter** 命令输入容器。

流程

1. 创建 Toolbx 容器：
   - 作为无根用户：

     ```
     $ toolbox create <mytoolbox>
     ```

   - 作为 root 用户：

     ```
     $ sudo toolbox create <mytoolbox>
     Created container: <mytoolbox>
     Enter with: toolbox enter
     ```

   - 验证您是否拉取了正确的镜像：

     ```
     [user@toolbox ~]$ toolbox list
     IMAGE ID      IMAGE NAME    CREATED
     fe0ae375f149   registry.access.redhat.com/ubi{ProductVersion}/toolbox 5 weeks ago

     CONTAINER ID  CONTAINER NAME  CREATED        STATUS   IMAGE NAME
     5245b924c2cb  <mytoolbox>       7 minutes ago   created
     registry.access.redhat.com/ubi{ProductVersion}/toolbox:8.9-6
     ```

2. 输入 Toolbx 容器：

   ```
   [user@toolbox ~]$ toolbox enter <mytoolbox>
   ```

验证

- 在 < ***mytoolbox&*** gt; 容器中输入一个命令，并显示容器和镜像的名称：

```
⬢ [user@toolbox ~]$ cat /run/.containerenv
engine="podman-4.8.2"
name="<mytoolbox>"
id="5245b924c2cb..."
image="registry.access.redhat.com/ubi{ProductVersion}/toolbox"
imageid="fe0ae375f14919cbc0596142e3aff22a70973a36e5a165c75a86ea7ec5d8d65c"
```

23.2. 使用 TOOLBX 进行开发

您可以使用 Toolbx 容器作为无根用户来安装开发工具，如编辑器、编译器和软件开发工具包(SDK)。安装后，您可以继续使用这些工具作为无根用户。

先决条件

- Toolbx 容器已创建并运行。您输入了 Toolbx 容器。您不需要创建具有 root 权限的 Toolbx 容器。请参阅 启动工具箱容器。

流程

- 安装您选择的工具，例如 Emacs 文本编辑器、GCC 编译器和 GNU Debugger (GDB)：

```
⬢[user@toolbox ~]$ sudo dnf install emacs gcc gdb
```

验证

- 验证是否安装了工具：

```
⬢[user@toolbox ~]$  dnf repoquery --info --installed <package_name>
```

23.3. 使用 TOOLBX 对主机系统进行故障排除

您可以使用 **systemd**、**journalctl** 和 **nmap** 等工具使用具有 root 特权的 Toolbx 容器来查找主机系统的各种问题的根本原因，而无需在主机系统上安装它们。例如，在 Toolbx 容器中，您可以执行以下操作：

先决条件

- Toolbx 容器已创建并运行。您输入了 Toolbx 容器。您需要使用 root 特权创建 Toolbx 容器。请参阅 启动工具箱容器。

流程

1. 安装 **systemd** 套件以便能够运行 **journalctl** 命令：

```
⬢[root@toolbox ~]# dnf install systemd
```

2. 显示主机上运行的所有进程的日志消息：

```
⬢[root@toolbox ~]# j journalctl --boot -0
```

```
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
```

3. 显示内核的日志消息：

```
⬢[root@toolbox ~]# journalctl --boot -0 --dmesg
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-e820: [mem 0x0000>
```

4. 安装 **nmap** 网络扫描工具：

```
⬢[root@toolbox ~]# dnf install nmap
```

5. 扫描网络中的 IP 地址和端口：

```
⬢[root@toolbox ~]# nmap -sS scanme.nmap.org
Starting Nmap 7.93 ( https://nmap.org ) at 2024-01-02 10:39 CET
Stats: 0:01:01 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 29.79% done; ETC: 10:43 (0:02:24 remaining)
Nmap done: 256 IP addresses (0 hosts up) scanned in 206.45 seconds
```

- **-sS** 选项执行 TCP SYN 扫描。大多数 Nmap 的扫描类型仅适用于特权用户，因为它们发送和接收原始数据包，这需要 UNIX 系统的 root 访问权限。

23.4. 停止 TOOLBX 容器

使用 **exit** 命令来离开 Toolbox 容器和 **podman stop** 命令停止容器。

流程

1. 保留容器并返回到主机：

```
⬢ [user@toolbox ~]$ exit
```

2. 停止 toolbox 容器：

```
⬢ [user@toolbox ~]$ podman stop <mytoolbox>
```

3. 可选：删除 toolbox 容器：

```
⬢ [user@toolbox ~]$ toolbox rm <mytoolbox>
```

另外，您还可以使用 **podman rm** 命令删除容器。

第 24 章 在 HPC 环境中使用 PODMAN

您可以使用带有 Open MPI（消息传递接口）的 Podman，来在高性能计算(HPC)环境中运行容器。

24.1. 使用带有 MPI 的 PODMAN

这个例子基于 Open MPI 的 ring.c 程序。在这个例子中, 一个值被所有进程以类似环形的方式传递。每次消息通过 rank 0 时，其值就会减少。当每个进程收到 0 信息时，它会把它传递给下一个进程，然后退出。通过先传递 0，每一个进程都会得到 0 信息，并可以正常退出。

先决条件

- **container-tools** 元数据包已安装。

流程

1. 安装 Open MPI:

```
# dnf install openmpi
```

2. 要激活环境模块，请输入：

```
$ . /etc/profile.d/modules.sh
```

3. 加载 **mpi/openmpi-x86_64** 模块：

```
$ module load mpi/openmpi-x86_64
```

另外，要自动载入 **MPI/openmpi-x86_64** 模块，请将此行添加到 **.bashrc** 文件中：

```
$ echo "module load mpi/openmpi-x86_64" >> .bashrc
```

4. 要将 **mpirun** 与 **podman** 相结合，请使用以下定义创建一个容器：

```
$ cat Containerfile
FROM registry.access.redhat.com/ubi9/ubi

RUN dnf -y install openmpi-devel wget && \
    dnf clean all

RUN wget https://raw.githubusercontent.com/open-mpi/ompi/master/test/simple/ring.c && \
    /usr/lib64/openmpi/bin/mpicc ring.c -o /home/ring && \
    rm -f ring.c
```

5. 构建容器：

```
$ podman build --tag=mpi-ring .
```

6. 启动容器。在有 4 个 CPU 的系统上,这个命令会启动 4 个容器：

```
$ mpirun \
  --mca orte_tmpdir_base /tmp/podman-mpirun \
```

```
$ mpirun -np 4 --hostfile ./my-hosts \
  podman run --env-host \
  -v /tmp/podman-mpirun:/tmp/podman-mpirun \
  --userns=keep-id \
  --net=host --pid=host --ipc=host \
  mpi-ring /home/ring
Rank 2 has cleared MPI_Init
Rank 2 has completed ring
Rank 2 has completed MPI_Barrier
Rank 3 has cleared MPI_Init
Rank 3 has completed ring
Rank 3 has completed MPI_Barrier
Rank 1 has cleared MPI_Init
Rank 1 has completed ring
Rank 1 has completed MPI_Barrier
Rank 0 has cleared MPI_Init
Rank 0 has completed ring
Rank 0 has completed MPI_Barrier
```

mpirun 会启动 4 个 Podman 容器，每个容器都运行一个 **ring** 二进制的实例。所有 4 个进程都通过 MPI 进行沟通。

其他资源

- HPC 环境中的 podman

24.2. MPIRUN 选项

以下 **mpirun** 选项用于启动容器：

- **--mca orte_tmpdir_base /tmp/podman-mpirun line** 告诉 Open MPI 在 **/tmp/podman-mpirun** 中创建所有临时文件，而不是在 **/tmp** 中创建。如果使用多个节点，则在其他节点上这个目录的名称会不同。这需要将完整的 **/tmp** 目录挂载到容器中，而这更为复杂。

mpirun 命令指定要启动的命令（ **podman** 命令）。以下 **podman** 选项用于启动容器：

- **run** 命令运行容器。
- **--env-host** 选项将主机中的所有环境变量复制到容器中。
- **-v /tmp/podman-mpirun:/tmp/podman-mpirun** 行告诉 Podman 挂载目录，Open MPI 在该目录中创建容器中可用的临时目录和文件。
- **--userns=keep-id** 行确保容器内部和外部的用户 ID 映射。
- **--net=host --pid=host --ipc=host** 行设置同样的网络、PID 和 IPC 命名空间。
- **mpi-ring** 是容器的名称。
- **/home/ring** 是容器中的 MPI 程序。

其他资源

- HPC 环境中的 podman

第 25 章 运行特殊容器镜像

您可以运行一些特殊类型的容器镜像。有些容器镜像有名为 *runlabels* 的内置标签，可让您使用预设置的选项和参数运行这些容器。**podman container runlabel <label>** 命令，您可以为容器镜像执行定义在 **<label>** 中的命令。支持的标签包括 **install**、**run** 和 **uninstall**。

25.1. 打开到主机的权限

特权容器和无特权容器之间存在一些区别：例如，toolbox 容器是一个特权容器。以下是可以从容器对主机公开或可能不公开的特权示例：

- Privileges:特权容器禁用将容器与主机隔离的安全功能。您可以使用 **podman run --privileged *<image_name>*** 命令运行特权容器。例如，您可以删除主机上挂载的 root 用户所拥有的文件和目录。
- Process tables:您可以使用 **podman run --privileged --pid=host *<image_name>*** 命令将主机 PID 命名空间用于容器。然后，您可以在特权容器中使用 **ps -e** 命令列出所有在主机上运行的进程。您可以将来自主机的进程 ID 传给在特权容器中运行的命令（例如：**kill <PID>**）。
- **网络接口** ：默认情况下，容器只有一个外部网络接口和一个回环网络接口。您可以使用 **podman run --net=host *<image_name>*** 命令直接从容器内部访问主机网络接口。
- Inter-process communications:主机上的 IPC 工具可从特权容器内访问。您可以运行 **ipcs** 等命令，来查看主机上活动的消息队列、共享内存段和 semaphore 设置的信息。

25.2. 带有 RUNLABELS 的容器镜像

有些红帽镜像包括为使用这些镜像提供预设置命令行的标签。使用 **podman container runlabel <label>** 命令，您可以使用 **podman** 命令执行为镜像在 **<label>** 中定义的命令。

现有 runlabels 包括：

- **install**:执行镜像前设置主机系统。通常情况下，这会在主机上创建文件和目录，容器可在稍后运行时访问。
- **run**:标识在运行容器时要使用的 podman 命令行选项。通常，这些选项将在主机上放开特权，并挂载容器需要永久保留在主机上的主机内容。
- **uninstall**:完成运行容器后，清理主机系统。

25.3. 使用 RUNLABELS 运行 RSYSLOG

发出 **rhel9/rsyslog** 容器镜像以运行 **rsyslogd** 守护进程的容器化版本。**rsyslog** 镜像包含以下 runlabels：**install**、**run** 和 **uninstall**。以下流程介绍了安装、运行和卸载 **rsyslog** 镜像的步骤：

先决条件

- **container-tools** 元数据包已安装。

流程

1. 拉取 **rsyslog** 镜像：

```
# podman pull registry.redhat.io/rhel9/rsyslog
```

2. 为 **rsyslog** 显示 **install** runlabel：

```
# podman container runlabel install --display rhel9/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel9/rsyslog:latest /bin/install.sh
```

此时该命令会为主机打开权限，将主机 root 文件系统挂载到容器中的 **/host**，并运行 **install.sh** 脚本。

3. 为 **rsyslog** 运行 **install** runlabel:

```
# podman container runlabel install rhel9/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel9/rsyslog:latest /bin/install.sh
Creating directory at /host//etc/pki/rsyslog
Creating directory at /host//etc/rsyslog.d
Installing file at /host//etc/rsyslog.conf
Installing file at /host//etc/sysconfig/rsyslog
Installing file at /host//etc/logrotate.d/syslog
```

这会在 **rsyslog** 镜像稍后要使用的主机系统上创建文件。

4. 为 **rsyslog** 显示 **run** runlabel：

```
# podman container runlabel run --display rhel9/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e
NAME=rsyslog --restart=always registry.redhat.io/rhel9/rsyslog:latest /bin/rsyslog.sh
```

这表明，在启动 **rsyslog** 容器来运行 **rsyslogd** 守护进程时，该命令会向主机开放特权，并在容器内挂载来自主机的特定的文件和目录。

5. 为 **rsyslog** 执行 **run** runlabel：

```
# podman container runlabel run rhel9/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e
NAME=rsyslog --restart=always registry.redhat.io/rhel9/rsyslog:latest /bin/rsyslog.sh
28a0d719ff179adcea81eb63cc90fcd09f1755d5edb121399068a4ea59bd0f53
```

rsyslog 容器会开放特权，挂载其需要的来自主机的内容，并在后台(**-d**)运行 **rsyslogd** 守护进程。**rsyslogd** 守护进程开始收集日志消息，并将信息定向到 **/var/log** 目录中的文件。

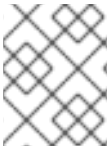
6. 显示 **rsyslog** 的 **uninstall** runlabel：

```
# podman container runlabel uninstall --display rhel9/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
```

```
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel9/rsyslog:latest /bin/uninstall.sh
```

7. 为 **rsyslog** 运行 **uninstall** runlabel：

```
# podman container runlabel uninstall rhel9/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel9/rsyslog:latest /bin/uninstall.sh
```



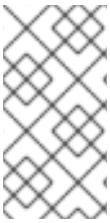
注意

在这种情况下, **uninstall.sh** 脚本只删除 **/etc/logrotate.d/syslog** 文件。它不会清理配置文件。

第 26 章 使用 CONTAINER-TOOLS API

新的基于 Podman 2.0 API 的 REST 替换了使用 varlink 库的 Podman 的旧远程 API。新的 API 可以在根和无根环境中工作。

Podman v2.0 RESTful API 由 为 Podman 和 Docker 兼容的 API 提供支持的 Libpod API 组成。借助这一新的 REST API，您可以从 cURL、Postman、Google 的高级 REST 客户端等平台调用 Podman。



注意

由于 podman 服务支持套接字激活，除非套接字上的连接处于活动状态，否则 podman 服务将不会运行。因此，要启用套接字激活功能，您需要手动启动 **podman.socket** 服务。当连接在套接字上处于活跃状态时，它会启动 podman 服务，并运行请求的 API 操作。操作完成后，podman 进程结束，podman 服务返回到非活动状态。

26.1. 在 ROOT 模式中使用 SYSTEMD 启用 PODMAN API

您可以执行以下操作：

1. 使用 **systemd** 激活 Podman API 套接字。
2. 使用 Podman 客户端执行基本命令。

先决条件

- **podman-remote** 软件包已安装。

```
# dnf install podman-remote
```

流程

1. 立即启动该服务：

```
# systemctl enable --now podman.socket
```

2. 使用 **docker-podman** 软件包启用到 **var/lib/docker.sock** 的链接：

```
# dnf install podman-docker
```

验证步骤

1. 显示 Podman 的系统信息：

```
# podman-remote info
```

2. 验证链接：

```
# ls -al /var/run/docker.sock
lrwxrwxrwx. 1 root root 23 Nov  4 10:19 /var/run/docker.sock -> /run/podman/podman.sock
```

其他资源

- Podman v2.0 RESTful API
- A First Look At Podman 2.0 API
- Sneak peek:podman 的新 REST API

26.2. 在无根模式下使用 SYSTEMD 启用 PODMAN API

您可以使用 **systemd** 激活 Podman API 套接字和 podman API 服务。

先决条件

- **podman-remote** 软件包已安装。

```
# dnf install podman-remote
```

流程

1. 立即启用并启动该服务：

```
$ systemctl --user enable --now podman.socket
```

2. 可选：使用 Docker 使程序与 rootless Podman 套接字交进行互：

```
$ export DOCKER_HOST=unix:///run/user/<uid>/podman//podman.sock
```

验证步骤

1. 检查套接字的状态：

```
$ systemctl --user status podman.socket
● podman.socket - Podman API Socket
 Loaded: loaded (/usr/lib/systemd/user/podman.socket; enabled; vendor preset: enabled)
Active: active (listening) since Mon 2021-08-23 10:37:25 CEST; 9min ago
Docs: man:podman-system-service(1)
Listen: /run/user/1000/podman/podman.sock (Stream)
CGroup: /user.slice/user-1000.slice/user@1000.service/podman.socket
```

podman.socket 处于活动状态，并侦听 **/run/user/*<uid>*/podman.podman.sock**，其中 ***<uid>*** *是* 用户的 ID。

2. 显示 Podman 的系统信息：

```
$ podman-remote info
```

其他资源

- Podman v2.0 RESTful API
- A First Look At Podman 2.0 API
- Sneak peek:podman 的新 REST API

- 使用 Python 和 Bash 探索 Podman RESTful API

26.3. 手动运行 PODMAN API

您可以运行 Podman API。这对于调试 API 调用，特别是在使用 Docker 兼容性层时很有用。

先决条件

- **podman-remote** 软件包已安装。

```
# dnf install podman-remote
```

流程

1. 为 REST API 运行服务：

```
# podman system service -t 0 --log-level=debug
```

   - 0 表示没有超时。rootful 服务的默认端点为 **unix:/run/podman/podman.sock**。
   - **--log-level <level>** 选项设定日志级别。标准日志记录级别为 **debug**、**info**、**warn**、**error**、**fatal** 和 **panic**。

2. 在另一个终端中，显示 Podman 的系统信息。**podman-remote** 命令与常规的 **podman** 命令不同，其可通过 Podman 套接字通信：

```
# podman-remote info
```

3. 若要对 Podman API 进行故障排除，并显示请求和响应，请使用 **curl** 命令。以 JSON 格式在 Linux 服务器上获取有关 Podman 安装的信息：

```
# curl -s --unix-socket /run/podman/podman.sock http://d/v1.0.0/libpod/info | jq
    {
  "host": {
    "arch": "amd64",
    "buildahVersion": "1.15.0",
    "cgroupVersion": "v1",
    "conmon": {
      "package": "conmon-2.0.18-1.module+el8.3.0+7084+c16098dd.x86_64",
      "path": "/usr/bin/conmon",
      "version": "conmon version 2.0.18, commit:
7fd3f71a218f8d3a7202e464252aeb1e942d17eb"
    },
    …
  "version": {
    "APIVersion": 1,
    "Version": "2.0.0",
    "GoVersion": "go1.14.2",
    "GitCommit": "",
    "BuiltTime": "Thu Jan  1 01:00:00 1970",
    "Built": 0,
    "OsArch": "linux/amd64"
  }
}
```

jq 工具是一个命令行 JSON 处理器。

4. 拉取 **registry.access.redhat.com/ubi8/ubi** 容器镜像：

```
# curl -XPOST --unix-socket /run/podman/podman.sock -v
'http://d/v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi'
*   Trying /run/podman/podman.sock...
* Connected to d (/run/podman/podman.sock) port 80 (#0)
> POST /v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi
HTTP/1.1
> Host: d
> User-Agent: curl/7.61.1
> Accept: /
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:58:37 GMT
< Content-Length: 231
<
{"status":"pulling image () from registry.access.redhat.com/ubi8/ubi:latest,
registry.redhat.io/ubi8/ubi:latest","error":"","progress":"","progressDetail":
{},"id":"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e"}
* Connection #0 to host d left intact
```

5. 显示拉取的镜像：

```
# curl --unix-socket /run/podman/podman.sock -v 'http://d/v1.0.0/libpod/images/json' |
jq
*   Trying /run/podman/podman.sock...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Connected to d
(/run/podman/podman.sock) port 80 (0) > GET /v1.0.0/libpod/images/json HTTP/1.1 > Host: d
> User-Agent: curl/7.61.1 > Accept: / > < HTTP/1.1 200 OK < Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:59:55 GMT < Transfer-Encoding: chunked < { [12498 bytes
data] 100 12485 0 12485 0 0 2032k 0 --:--:-- --:--:-- --:--:-- 2438k * Connection #0 to host d
left intact [ { "Id":
"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e",
"RepoTags": [ "registry.access.redhat.com/ubi8/ubi:latest", "registry.redhat.io/ubi8/ubi:latest"
], "Created": "2020-09-01T19:44:12.470032Z", "Size": 210838671, "Labels": { "architecture":
"x86_64", "build-date": "2020-09-01T19:43:46.041620", "com.redhat.build-host": "cpt-
1008.osbs.prod.upshift.rdu2.redhat.com", ... "maintainer": "Red Hat, Inc.", "name": "ubi8", ...
"summary": "Provides the latest release of Red Hat Universal Base Image 8.", "url":
"https://access.redhat.com/containers//registry.access.redhat.com/ubi8/images/8.2-347",
      ...
    },
    "Names": [
      "registry.access.redhat.com/ubi8/ubi:latest",
      "registry.redhat.io/ubi8/ubi:latest"
    ],
    ...
    ]
  }
]
```

其他资源

- Podman v2.0 RESTful API
- Sneak peek:podman 的新 REST API
- 使用 Python 和 Bash 探索 Podman RESTful API
- **podman-system-service** 手册页