第1章　容器技术

1.1 容器技术简介

1.1.1 容器与虚拟化

虚拟化（Virtualization）和容器（Container）都是系统虚拟化的实现技术，可实现系统资源的“一虚多”共享。容器技术是一种“轻量”的虚拟化方式，此处的“轻量”主要是相比于虚拟化技术而言的。例如，虚拟化通常在 Hypervisor 层实现对硬件资源的虚拟化，Hypervisor 为虚拟机提供了虚拟的运行平台，管理虚拟机的操作系统运行，每个虚拟机都有自己的操作系统、系统库以及应用。而容器并没有 Hypervisor 层，每个容器是和主机共享硬件资源及操作系统。

容器技术在操作系统层面实现了对计算机系统资源的虚拟化，在操作系统中，通过对CPU、内存和文件系统等资源的隔离、划分和控制，实现进程之间透明的资源使用。图 1.1 展示了虚拟机和容器在实现架构上的区别。

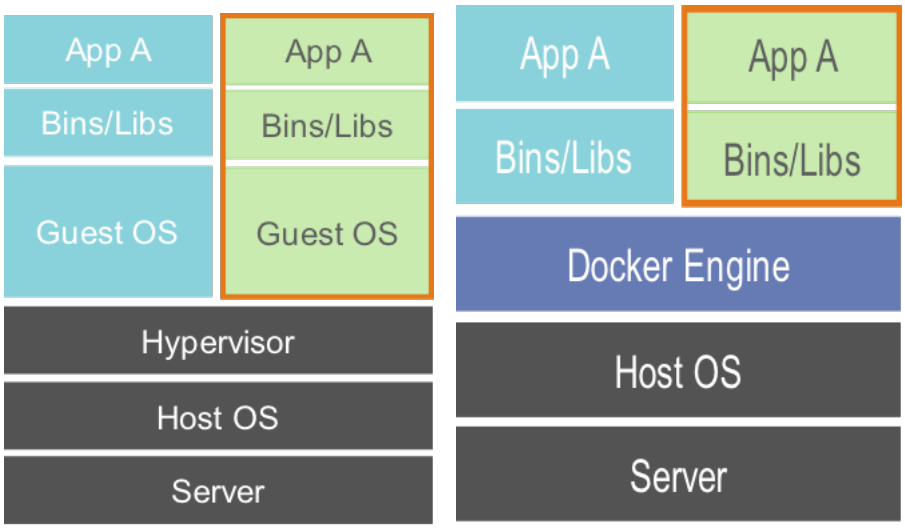

图 1.1 虚拟机和容器架构对比

1.1.2 容器发展历程

容器的概念最早可以追溯到 1979 年的 Unix 工具 Chroot，2000 年左右，FreeBSD 引入的 Jails 算是早期的容器技术之一，2004 年 Solaris 提出 Container，引入了容器资源管理的概念。

2008 年出现的 LXC（Linux Containers）可以说是第一个完整的 Linux 容器管理实现方案，它通过 Linux CGroups（Control Groups）以及 Linux Namespace 技术实现，LXC 存在于 liblxc 库中，提供各种编程语言的 API 实现，无需任何额外的补丁就能够运行在原版 Linux 内核上。

2013 年，DotCloud 开源了其内部的容器项目 Docker[1]。Docker 在开始阶段是基于 LXC 技术，之后则采用自己开发的 libcontainer 进行了替换。Docker 除了基础的容器服务之外，还引入了一整套管理容器的生态系统，包括容器镜像模型、镜像仓库、REST API、命令行等。

2014 年，CoreOS 发布了容器引擎 Rocket[2]（简称 rkt），它是在一个更加开放的标准 App Container 上实现的。除 Rocket 之外，CoreOS 也开发了其它几个容器相关的产品，如 CoreOS 操作系统、分布式键值存储组件 Etcd 和网络组件 Flannel。

2015 年，微软也在其 Windows Server 上为基于 Windows 的应用添加了容器支持，称之为 Windows Containers[3]。该实现可以在 Windows 上原生地运行 Docker 容器。

2017 年 11 月，阿里巴巴开源了其基于 Apache 2.0 许可协议的轻量级容器技术 Pouch[4]，Pouch 在其容器技术 t4 的基础上，逐渐吸收了社区中的 Docker 镜像技术，具有快速高效、可移植性高、资源占用少等特性，目前在阿里的使用场景中已经扮演了重要角色。

---

1 Docker, https://github.com/docker
2 Rocket, https://github.com/rkt/rkt
3 Windows Containers, https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/
4 Alibaba Pouch, https://github.com/alibaba/pouch

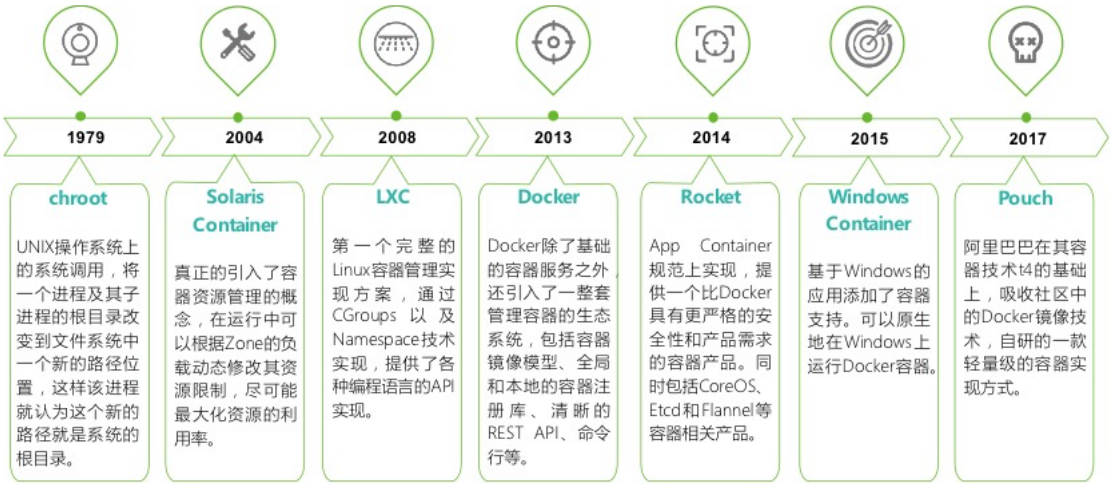

图 1.2 容器技术发展历程

1.2 容器镜像

镜像是容器运行的基础，容器引擎服务可使用不同的镜像启动相应的容器。在容器出现错误后，能迅速通过删除容器、启动新的容器来恢复服务，这都需要以容器镜像作为支撑技术。

1.2.1 镜像简介

镜像是由按层封装好的文件系统和描述镜像的元数据构成的文件系统包，包含应用所需要的系统、环境、配置和应用本身等。镜像由开发者构建好之后上传至镜像仓库，使用者获取镜像之后就可以使用镜像直接构建自己的应用。

由 Linux 基金会主导开发的开放容器标准规范（Open Container Initiative，OCI）于 2017 年发布 v1.0 版本，该标准将致力于统一容器运行时和镜像格式的规范。Docker 积极为 OCI 做出重要贡献，开发并捐赠了大部分的 OCI 代码，并作为项目维护者在定义运行时和镜像规范时做了建设性工作。

与虚拟机所用的系统镜像不同，容器镜像不仅没有 Linux 系统内核，同时在格式上也有很大的区别。虚拟机镜像是将一个完整系统封装成一个镜像文件，而容器镜像不是一个文件，而是分层存储的文件系统。

1.2.2 镜像原理

1.2.2.1 分层存储

分层存储是容器镜像的主要特点之一，从图 1.3 容器镜像结构图可以看出，每个镜像都是由一系列的“镜像层”组成。当需要修改镜像内的某个文件时，只会对最上方的读写层进行改动，不会覆盖下层已有文件系统的内容。当提交这个修改生成新的镜像时，保存的内容仅为最上层可读写文件系统中被更新过的文件，这样就实现了在不同的容器镜像间共享镜像层的效果。下图是一个简单的容器镜像示例，最上层是容器的读写层，剩余的是只读层。

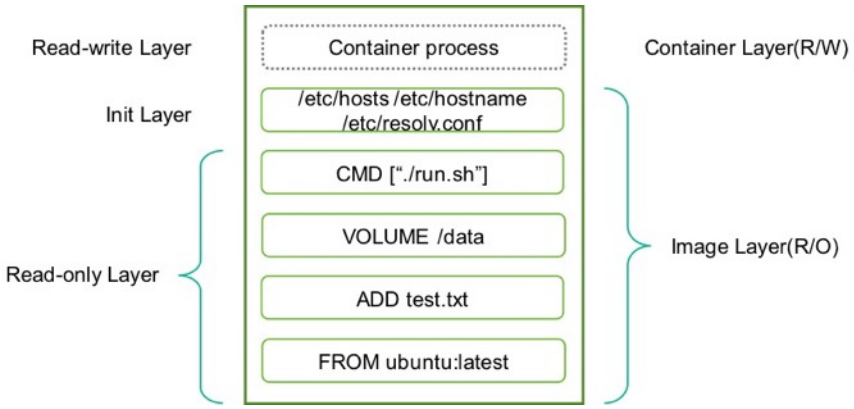

图 1.3 容器镜像结构图

1.2.2.2 写时复制

容器镜像使用了写时复制(Copy-on-Write)的策略，在多个容器之间共享镜像，每个容器在启动的时候并不需要单独复制一份镜像文件，而是将所有镜像层以只读的方式挂载到一个挂载点，再在上面覆盖一个可读写的容器层。在未更改文件内容时，所有容器共享同一份数据，只有在容器运行过程中文件系统发生变化时，才会把变化的文件内容写到可读写层，并隐藏只读层中的老版本文件。写时复制配合分层机制减少了镜像对磁盘空间的占用和容器启动时间。

1.2.2.3 内容寻址

在 Docker 1.10 版本后，引入了内容寻址存储的机制，根据文件的内容来索引镜像和镜像层。对镜像层的内容计算校验和，生成一个内容哈希值，并以此哈希值作为镜像层的唯一标识。该机制提高了镜像的安全性，并在 pull、push、load 和 save 操作后检测数据的完整性。

1.2.2.4 联合挂载

联合挂载技术可以在一个挂载点上同时挂载多个文件系统，将挂载点的原目录与被挂载内容进行整合，使得最终可见的文件系统包含整合之后各层的文件和目录。实现这种联合挂载技术的文件系统通常被称为联合文件系统(Union Filesystem)。

联合挂载是用于将多个镜像层的文件系统挂载到一个挂载点来实现一个统一文件系统视图的途径，是下层存储驱动实现分层合并的方式。

1.2.3 镜像制作

镜像作为容器运行的基础，有多种获取途径。其中一种是在镜像仓库中获取现成的镜像，包括公共仓库和私有仓库；另一种是开发人员自己打包制作镜像。本节主要介绍两种如何制作并生成镜像的方法，具体上手操作可参考 1.6.2 小节。

1. docker build

这种构建方法基于 Dockerfile 来自动构建镜像，Dockerfile 的可读性和可理解性都较高。其机制为：每一行都会基于上一层的中间容器来执行对应的修改命令，然后通过 docker build 提交，经过一次次循环，最终提交成为目标镜像。构建镜像的 Dockerfile 示例如下：

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Dockerfile 中的每一行指令（COPY、RUN、CMD）都会生成新的一层，叠加在上一个指令生成的文件系统之上，最后所有镜像层叠加就构成了镜像的文件系统。

2. docker commit

这种构建镜像的方式，首先使用某一镜像启动容器，进入容器中完成需要的操作，最后在宿主机上执行 docker commit 命令，该命令会将此时的容器打包成一个新的镜像。这种制作新镜像的优点是修改便捷、构建的镜像出现问题时容易排查解决。缺点在于不够透明、可维护性差。

1.2.4 镜像存储

镜像仓库是镜像存储的位置，也是用来获得镜像的重要渠道之一。镜像仓库根据用途不同分为公共仓库和私有仓库。

1.2.4.1 公共仓库

公共仓库是面向整个互联网用户的仓库，典型的代表为 Docker Hub，目前已经包括了超过 5300000 个镜像。大部分常用应用的镜像，都可以通过在 Docker Hub 中直接下载使用。

另外，用户也可以将自己创建好的镜像上传至公共仓库供其它的用户使用，软件厂商也可以以镜像形式发布自己的软件。

1.2.4.2 私有仓库

不是所有的镜像都适合在互联网进行发布共享，有时候有些含有敏感信息的镜像只能在团队内部共享使用。

因此，考虑到镜像的敏感性与网络的可用性和稳定性等因素，出现了私有仓库的使用场景，或者可以称之为本地仓库。私有仓库是在一定范围内可访问的镜像仓库，典型的私有仓库实现代表为 Harbor。

Harbor 由 VMware 中国研发团队负责开发，旨在帮助用户迅速搭建一个企业级的 Registry 服务。它以 Docker Registry 为基础，提供了管理 UI、基于角色的访问控制、AD/LDAP 集成、审计日志以及镜像漏洞扫描等功能。

1.2.5 镜像使用

除了 docker pull/push 等常用的命令外，Docker 中操作镜像的一些其它常用操作命令如下，更多镜像操作，可参考 Docker 官方文档[5]。

5 https://docs.docker.com/engine/reference/commandline/images/

1.3 容器存储

1.3.1 镜像元数据

在 Linux 系统中 Docker 的数据默认存放在/var/lib/docker 中，基于不同的系统又有不同的存储驱动、不同的目录结构。本文以 OCI 标准格式来了解镜像存储的内容。

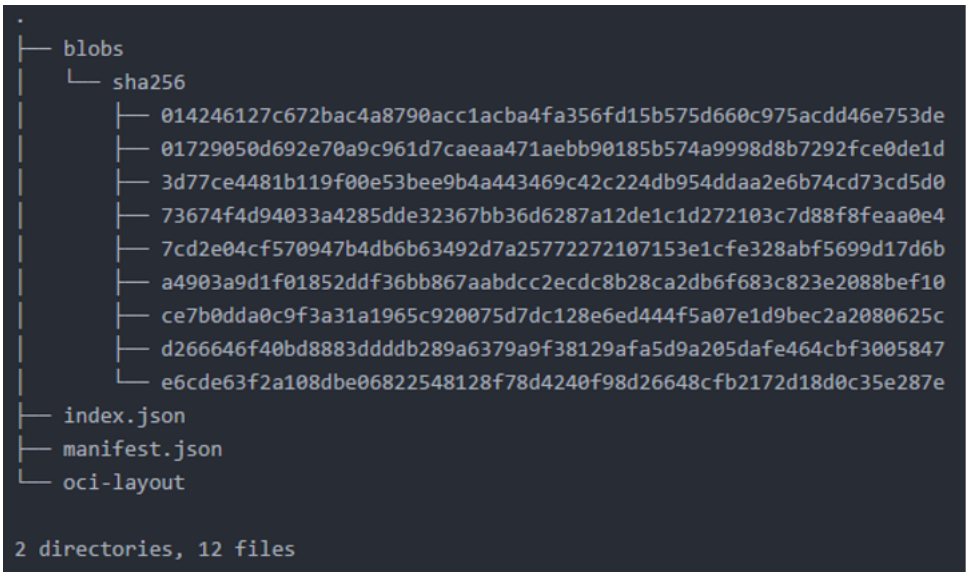

图 1.4 镜像存储目录

镜像每一层的 ID 是该文件内容的哈希校验值，作为该层的唯一标识。获取镜像后，会使用以下方式索引镜像：首先读取镜像的 manifests，根据 manifests 文件中 config 的 sha256 码，得到镜像 config 文件，遍历 manifests 里面的所有 layer，根据其 sha256 码在本地查找，拼出完整的镜像。

1.3.2 存储驱动

理想情况下，使用挂载卷来存储高读写的目录，很少将数据直接写入容器的可写层。但是，总有些特殊需求需要直接写入容器的可写层。这时候就需要存储驱动来作为容器和宿主机之间的媒介。Docker 依靠驱动技术来管理镜像和运行它们的容器间的存储和交互。

目前，Docker 主要支持 AUFS、BtrFS、Device Mapper、OverlayFS、ZFS 五种存储驱动[i]。没有单一的存储驱动适合所有的应用场景，要根据不同的场景选择合适的存储驱动，才能有效提高 Docker 的性能。

1.3.3数据卷

通常，有状态的容器都有数据持久化存储的需求。前一节提到过，文件系统的改动都是发生在最上面的可读写层。在容器的生命周期内，它是持续的，包括容器被停止后。但是，当容器被删除后，该数据层也随之被删除了。

因此，Docker 采用数据卷（Volume）的形式向容器提供持久化存储。数据卷是 Docker 容器数据持久化存储的首选机制。绑定挂载（Bind Mounts）依赖于主机的目录结构，但数据卷是由 Docker 管理。与绑定挂载相比，数据卷有以下几个优点：

- 与绑定挂载相比，数据卷更容易备份或迁移；
- 可以使用 Docker CLI 命令或 Docker API 管理数据卷；
- 数据卷在 Linux 和 Windows 上均可使用；
- 数据卷可以在多个容器之间更安全地共享；
- 数据卷驱动程序允许在远程主机或云上存储数据卷、加密卷的内容或添加其它功能；
- 新数据卷的内容可以由容器预填充。

另外，与使用容器的读写层保存数据相比，数据卷通常是更好的选择。因为使用数据卷存储不会增加容器的大小，并且数据卷是持久化的，不会依赖于容器的生命周期。

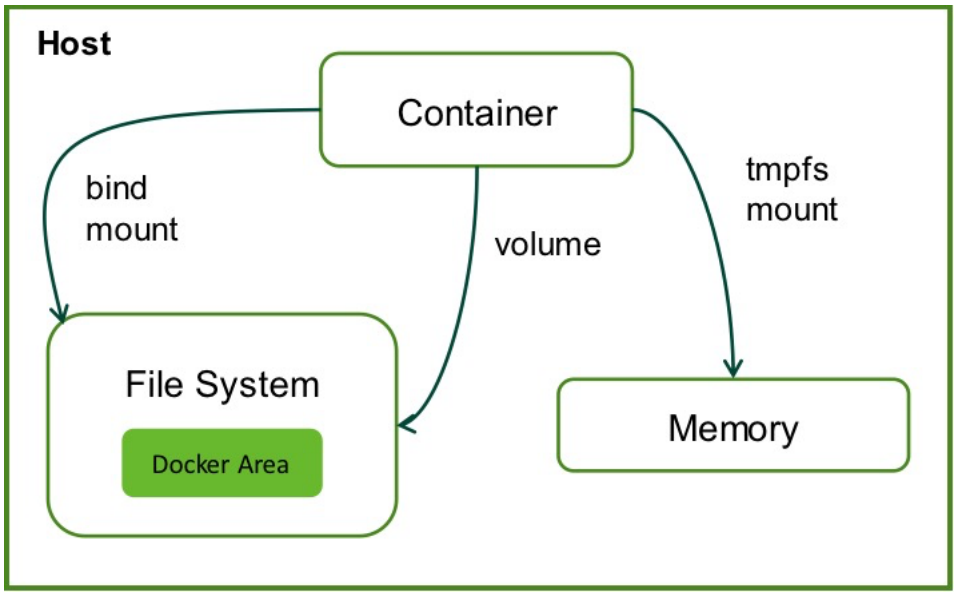

图 1.5 Docker 主机上数据卷的挂载方式

1.4 容器网络

从云计算系统的发展来看，业界普遍的共识是，计算虚拟化和存储虚拟化已经不断突破和成熟，但网络虚拟化的发展仍相对滞后，成为制约云计算发展的一大瓶颈。网络虚拟化、多租户、混合云等特性均不同程度地给云网络的安全建设提出全新的挑战。

容器技术提供了轻量级虚拟化的能力，使实例资源占用大幅降低，提升了分布式计算系统的性能，但分布式容器系统的网络仍是较为复杂的部分。

本节将针对容器主机网络以及容器集群网络分别进行介绍。

1.4.1 容器网络支撑技术

容器网络虽然有多种形态，但多数使用了若干种支撑技术，例如网络命名空间（Network Namespace）、Linux 网桥（Linux Bridge）以及虚拟网络接口对（Veth Pair）。

1.4.1.1 网络命名空间

网络命名空间是一种实现网络隔离的技术，创建一个网络命名空间后就有一个包括网络接口、路由、访问控制规则（Iptables）等网络资源的独立网络环境，该命名空间的网络与其它网络隔离。

1.4.1.2 Linux 网桥

Linux 网桥是 Linux 系统中的虚拟网桥，它可以将不同主机的网络接口连接，从而实现主机间的通信。

Docker 启动后，会默认创建名为 docker0 的 Linux 网桥。在未创建任何容器时，可以看到 docker0 网桥没有接口连接。

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242d1837f60       no
```

当创建容器时（d458f9bd528），Docker 会为容器创建虚拟网络接口，并将其连接到 docker0 网桥上。

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242f22b2de4       no              veth4d91464
                                                        veth6ed0a8c
```

1.4.1.3 虚拟网络接口对

为了实现容器与宿主机网络、外部网络之间的通信，需要通过虚拟网络接口对将容器与 Linux 网桥连接。

当 Docker 启动一个容器时（d458f9bd528），会创建一个虚拟网络接口对，即两个相连的虚拟网络接口。其中一个连接容器，成为容器 d458f9bd528 的网卡 eth0；另一个被连接到 docker0 网桥上，从而容器内部的数据包先后经过 eth0 和 veth6ed0a8c 对到达 docker0 网桥，实现在同一子网内不同容器之间的通信。

通过下面命令可以得到容器的网卡 eth0 在宿主机上的编号（peer-ifindex）。

```
# docker exec d458f9bd528 ethtool -S eth0
NIC statistics:
peer_ifindex: 37
```

继续执行下面命令找到宿主机上 peer_ifindex 为 37 的接口名称。

```
# ip link | grep 37
37: vethfaa2a17@if36: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master docker0 state UP mode DEFAULT
```

运行 ethtool 可查找其对端接口编号。

```
# ethtool  -S vethfaa2a17
NIC statistics:
    peer_ifindex: 36
```

1.4.2 主机网络

以 Docker 为例，目前 Docker 容器主机网络（Host Network）主要分为以下 4 种模式。

1.4.2.1 None 网络模式

None 网络模式下，容器拥有自己的网络命名空间，但是并不为容器进行任何的网络配置。创建的容器只有 loopback 接口，需要用户为容器添加网卡、配置 IP 等。

rkt 同样支持 None 网络模式。该网络模式的用途主要是测试容器、稍后为容器分配网络、以及一些对安全性要求高且不需要联网的场景。

1.4.2.2 Bridge 网络模式

Bridge（网桥）网络模式主要是利用 Iptables 进行 NAT 和端口映射，从而提供单主机网络。与虚拟机下的 NAT 网络类似，这种网络模式下同一主机上的容器之间是可以互相通信的，但是分配给每个容器的 IP 地址从主机外部不能访问。

Bridge 网络是 Docker 默认的网络类型，在安装完 Docker 后会默认创建 docker0 网桥，并通过虚拟网络接口对连接容器和 docker0 网桥，这样主机上的所有容器就处在了一个二层网络中。

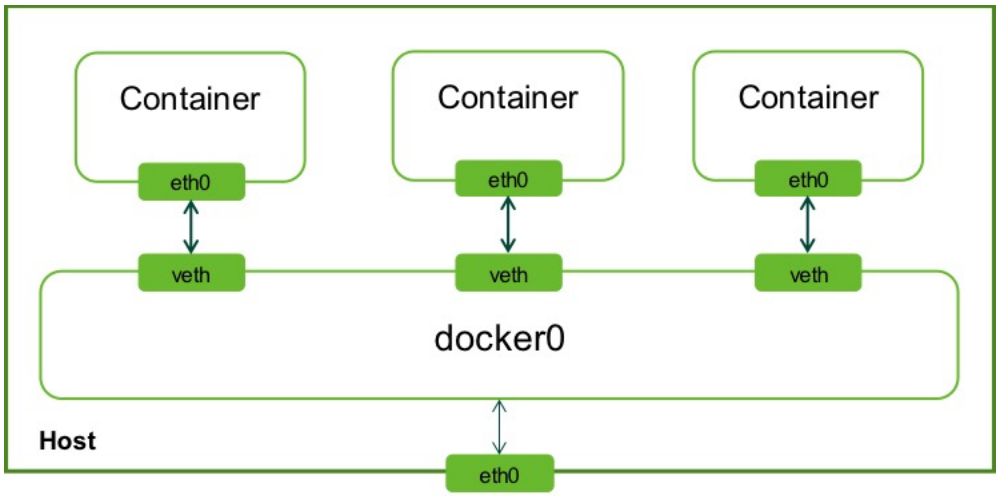

图 1.6 Bridge 网络模式

1.4.2.3 Host 网络模式

Host（主机）网络模式下，Docker 服务启动容器时并不会为容器创建一个隔离的网络环境，容器将会被加入主机所在网络，共享主机的网络命名空间（/var/run/docker/netns/default）。其网络配置（网络地址、路由表和 Iptables 等）和主机保持一致，容器通过主机的网卡和 IP，实现与外部的通信。

由于容器并没有独立的网络命名空间，容器服务的端口没有经过端口映射就直接暴露在主机上，因此容器中服务的端口号不能与主机上已经使用的端口号冲突。

以这种网络模式创建的容器虽然可以访问主机的所有网络接口，但除非在特权模式下部署，否则容器可能不会重新配置主机的网络堆栈。Host 网络模式是 Apache Mesos 中默认使用的网络模式，如果没有指定网络类型，新的网络命名空间将不会与容器相关联，而是与主机网络相关联。

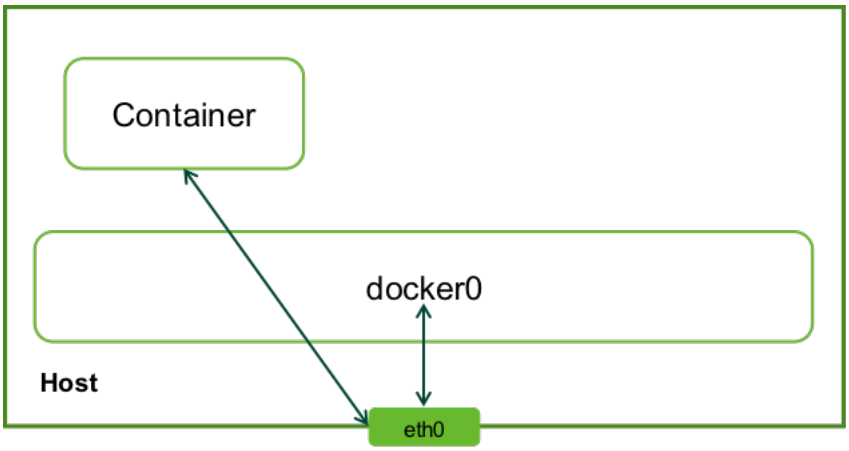

图 1.7 Host 网络模式

1.4.2.4 Container 网络模式

Container（容器）网络模式比较特殊，新创建的容器和已经存在的某个容器共享同一个命名空间。该新容器不会创建自己的网卡或配置自己的 IP，而是和一个指定容器共享 IP、端口范围等。

这两个容器只有网络方面共享数据，文件系统、进程列表等其它方面还是隔离的。两个容器的进程可以通过 loopback 网卡通信。

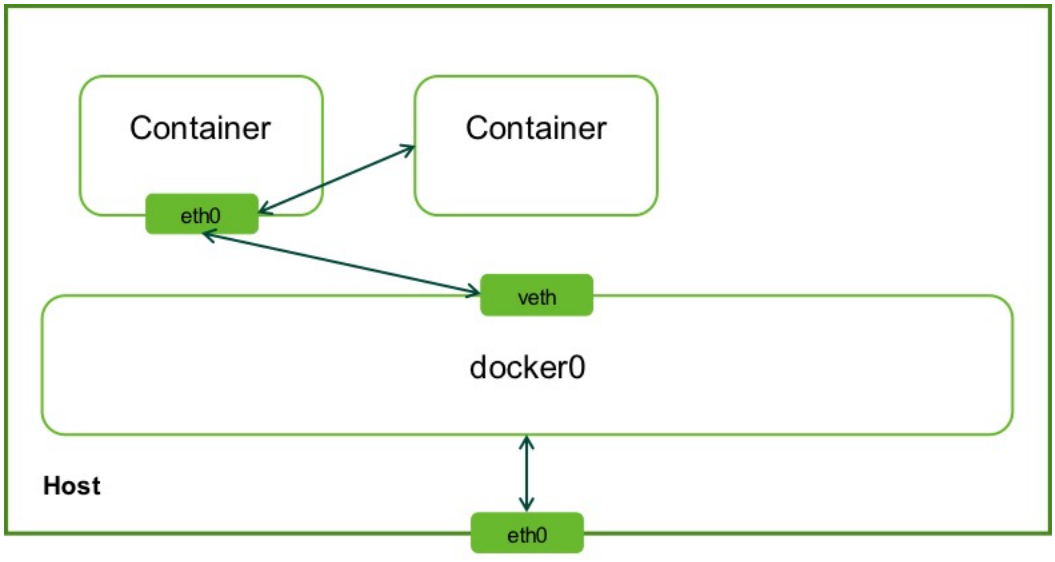

图 1.8 Container 网络模式

1.4.3 集群网络

本节以 Kubernetes 为例，具体阐述容器集群中的网络实现。

Kubernetes 设计了 Pod 对象，对应某特定应用中的逻辑主机（Logical Host），将每个服务按任务拆分，分别将相应进程包装到相应的 Pod 中。一个 Pod 中包含一个或多个相关的容器，这些容器都会运行在同一个主机中，并且共享相同的网络命名空间和相同的 Linux 协议栈。

一个 Kubernetes 集群通常会涉及到以下三种通信：

1. 同一个 Pod 内，容器和容器之间的通信；
2. 同一个主机内不同 Pod 之间的通信；
3. 跨主机 Pod 之间通信。

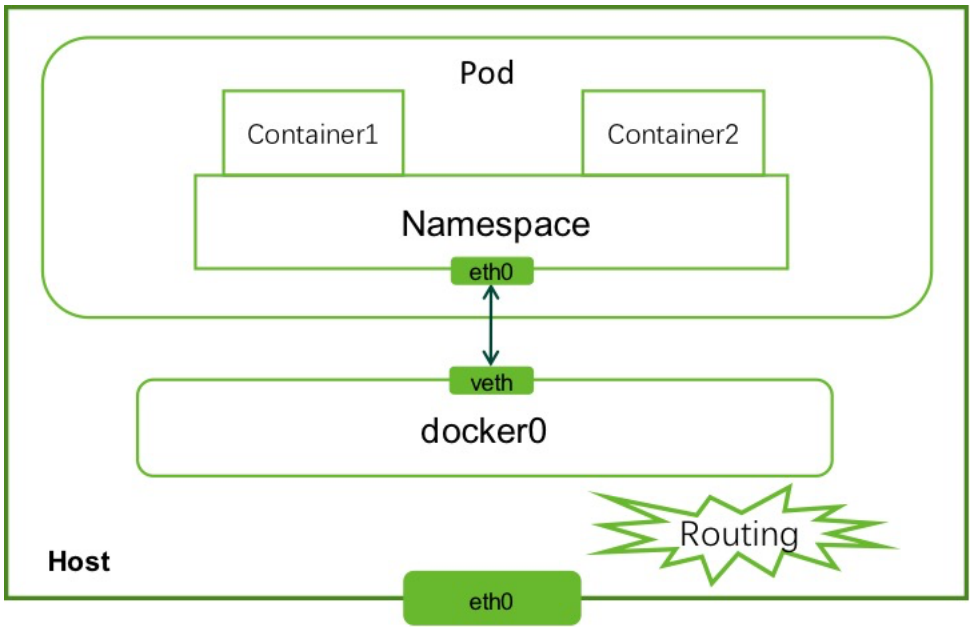

图 1.9 同一 Pod 内容器网络模型

同一个 Pod 内容器之间的通信，由于其共享网络命名空间、共享 Linux 协议栈，因此它们之间的通信是最简单的，这些容器好像是运行在同一台机器上，直接使用 Linux 本地的 IPC 进行通信，它们之间的互相访问只需要使用 localhost 加端口号就可以。

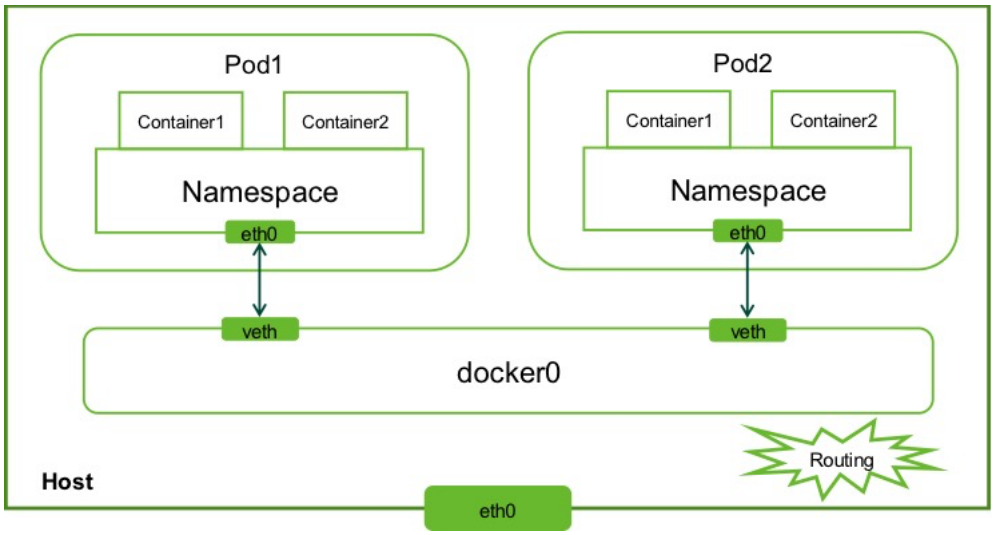

图 1.10 同一主机不同 Pod 间的容器网络模型

同一个主机上不同的 Pod 通过 veth 连接在同一个 docker0 网桥上，每个 Pod 从 docker0 动态获取 IP 地址，该 IP 地址和 docker0 的 IP 地址是处于同一网段的。这些 pod 的默认路由都是 docker0 的 IP 地址，所有非本地的网络数据都会默认送到 docker0 网桥上，由 docker0 网桥直接转发，相当于一个本地的二层网络。

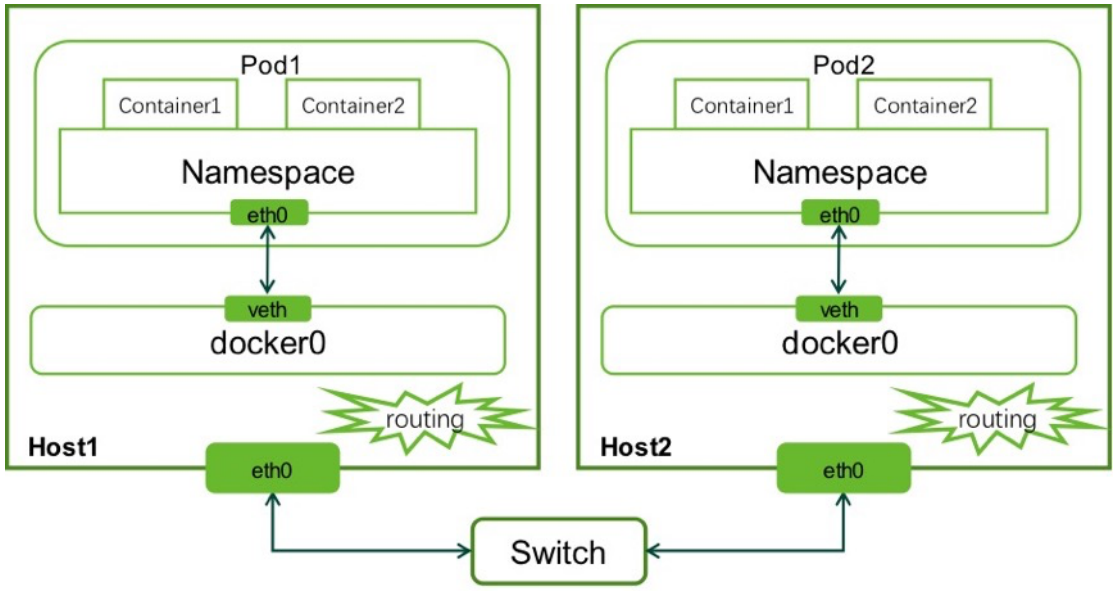

图 1.11 跨主机不同 Pod 间的网络模型

跨主机的 Pod 之间通信较复杂，每个 Pod 的地址和其所在主机的 docker0 在同一个网段，而 docker0 和主机的物理网络是不同的网段，如何保证 Pod 内的网络数据能够通过物理网络，找到对端 Pod 所在物理主机地址，并且完成数据传输是通信成功的关键。

因此在 Kubernetes 的网络中，需要有一个全局网络地址规划的模块，当 Pod1 向 Pod2 发送数据时，能够知道目的 Pod2 所在主机的 IP 地址。这样数据从 Pod1 发出，经 Host1 的 docker0 网桥路由到 Host1 的物理网卡 eth0，然后通过物理网络到达 Host2 的物理网卡 eth0、docker0 网桥，进而送达 Pod2。

在多数的私有容器云环境中还可以借助第三方开源的网络插件来实现集群网络，比如 Flannel、Calico 等。例如，Flannel 是 CoreOS 团队针对 Kubernetes 设计的一个 Overlay 网络工具，其目的在于帮助每一个使用 Kubernetes 的主机拥有一个完整的子网。

1.5 容器运行时

1.5.1 背景

容器运行时负责管理容器运行的整个生命周期，包括但不限于指定容器镜像格式、构建镜像、上传和拉取镜像、管理镜像、管理容器实例、运行容器等。在容器技术发展早期，Docker 作为容器运行时的标准，被广为使用，而后为避免 Docker 公司一家独大，由 Google、CoreOS、Docker 等公司在 2015 年联合创建了开放容器标准（Open Container Initiative OCI）[6]，用于推进容器标准化，其主要包含两个标准，分别为容器运行时标准[7]和容器镜像标准[8]，OCI 标准的容器运行时主要包括 runC、Rocket、Kata 容器、gVisor 等。再后来随着容器编排技术的不断发展，处于行业翘楚的 Kubernetes 推出了容器运行时接口（Container Runtime Interface CRI）用于与容器运行时进行通信，进而操作容器化应用程序，目前 Kubernetes 支持的 CRI 运行时包括 Docker、Containerd、CRI-O。

2020 年 12 月，Kubernetes 1.20 版本的 ChangeLog 中写到其 Kubelet 组件对 Docker 的支持将进入淘汰阶段，并且会在未来被移除，原因是 Docker 并不符合 Kubernetes 的 CRI 标准，官方之前一直通过维护中间件的形式来调用 Docker，从 Kubernetes 的长远发展来

6 https://github.com/opencontainers
7 https://github.com/opencontainers/runtime-spec
8 https://github.com/opencontainers/image-spec

看，这并非是一项明智之举，因此建议用户评估并迁移至 CRI 支持更完善的运行时上，例如 Containerd、CRI-O 等。

从容器运行时的发展历程来看，容器和 Docker 这两个经常被混淆使用的词，其边界将会愈发清晰，未来容器的构建、管理将会更倾向于使用各自领域的工具实现，各司其职。

1.5.2 容器运行时结构

针对云原生容器运行时结构，笔者看来可以简单汇总为三层抽象，由左至右分别为编排 API 层、容器 API 层、内核 API 层，每一层均有着不同的实现方式，其中 Kubernetes 为实现编排 API 的标准，在 Kubernetes 中，CRI 标准运行时实现了容器 API 层，而 OCI 标准运行时实现了内核 API 层，因而若要在 Kubernetes 中启动一个容器，需要使用 CRI 标准运行时启动 OCI 标准运行时实现。图 1.12 展示了这三层间的逻辑关系：

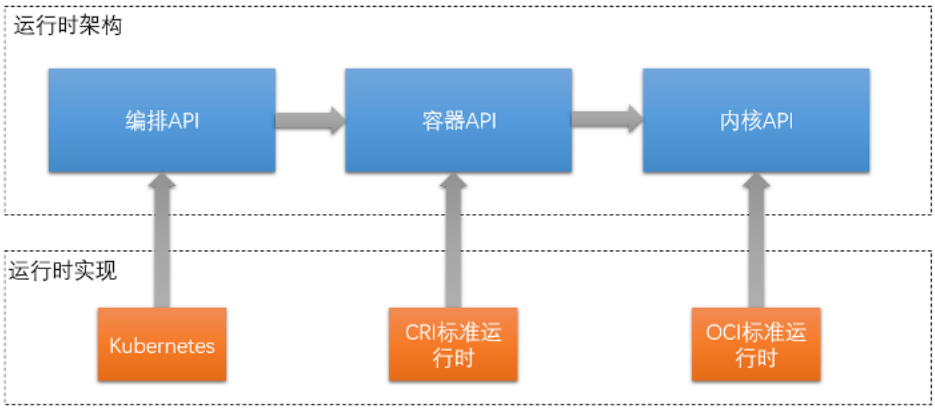

图 1.12 容器运行时架构

1.5.3 容器运行时分类

1.5.3.1 OCI 层面的容器运行时

Rocket(rkt)

2014 年底，CoreOS 正式发布了其开源容器引擎 rkt9作为 docker/runC 的竞争对手。自发行的几年内，rkt 已经走向成熟并得到了广泛的应用，尤其与 Kubernetes 有着良好的

9 https://github.com/rkt/rkt

兼容性。此外，尽管在兼容 OCI 标准的路途中与 docker/runC 仍有差距，但 rkt 也做出了巨大的努力，后来随着各类容器运行时的兴起，rkt 开源社区渐渐销声匿迹，如今该项目已停止维护。

技术实现上，与 runC 不同点在于 rkt 无守护进程，而是直接通过客户端命令启动容器。

runC

2015 年，docker 发布 runC，runC 前身是由 docker 内部的 libcontainer 进化而来，后来被 docker 单独提取出来作为开源项目[10]捐赠给了 OCI，也成为了 OCI 事实上的第一个标准容器运行时。runC 非常轻量级，由 golang 实现，可以脱离 docker 引擎直接运行容器。

Kata Containers

虽然 runC 通过 Cgroup 和 Namespace 实现了资源的隔离及管理，但其启动容器是共享内核的，从而导致容器逃逸现象频繁出现。为实现资源的强隔离，在不共享系统内核的前提下还能保证容器的秒级部署特性，Kata 容器应运而生，Kata 容器[11]整合了 Hyper.sh 的 runV 项目和 Intel 的 Clear Container 项目，支持不同平台的硬件（X86_64,ARM,IBM）等，并于 2017 年开源，其符合 OCI 运行时规范，并且可以替换 runC 或 rkt，此外还支持 Kubernetes。

技术实现上，Kata 容器实际上通过虚拟机的形式来运行容器，从而实现内核层面的硬隔离，图 1.13 直观的展示了 kata 容器与传统容器之间的区别：

10 https://github.com/opencontainers/runc
11 https://katacontainers.io

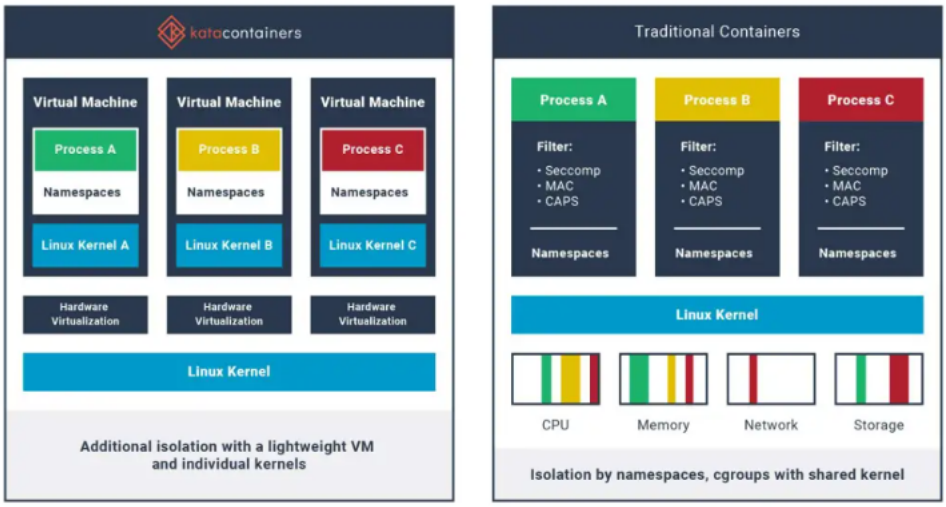

图 1.13 Kata 容器与传统容器对比图

欲了解更多关于 Kata Containers 的内容，可以参考官方文档12。

gVisor

2018 年 5 月，Google 开源了一款安全容器 gVisor13，它在实现上与 Kata 容器有明显不同。Kata 容器虽然同样避免了容器与宿主机共享内核，但它的思路是提供一个虚拟机，容器与虚拟机共享内核；而 gVisor 则直接在用户层实现了内核，用来拦截容器内程序对系统 API 的调用，处理并响应。与 Kata 容器相同的是，gVisor 也遵循 OCI 标准，并且支持 Kubernetes。

1.5.3.2 CRI 层面的容器运行时

Containerd

2017 年，Docker 公司将 Containerd 单独提取出来作为开源项目14捐赠给了云原生计算基金会（Cloud Native Computing Foundation CNCF），目前该项目已毕业。Containerd 是一个工业标准的容器运行时，其秉承简单、健壮、可移植的原则，可作为 Linux 和 Windows 的守护进程，从而实现容器完整的生命周期管理。Containerd 符合 OCI 和 CRI 标准，天然支持 Kubernetes。

12 https://github.com/kata-containers/documentation/tree/master/design
13 https://github.com/google/gvisor
14 https://github.com/containerd/containerd

从 Containerd 1.1 版本后，其对 Kubernetes CRI 的适配逻辑是以 Plugin 的方式插入 Containerd 主进程实现，非常轻量便捷，如图 1.14 所示：

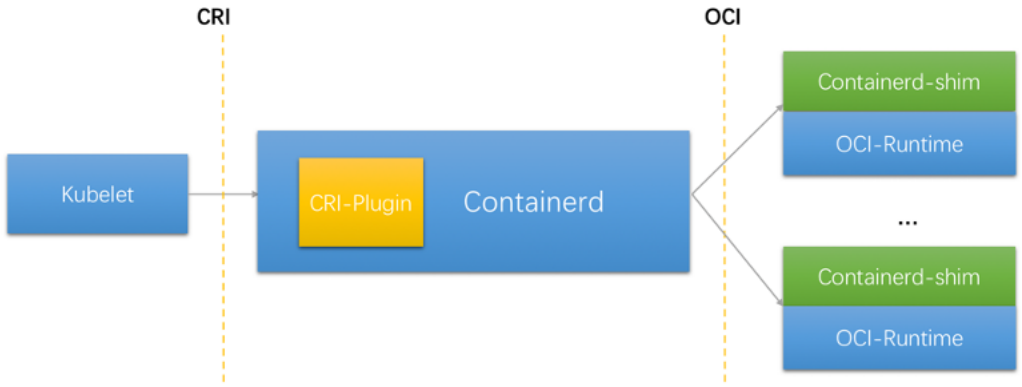

图 1.14 Containerd CRI 适配图

CRI-O

2016 年，红帽公司开发了一款 CRI 运行时 CRI-O，CRI-O 相比 Containerd 更为轻量级和专一，且只支持 Kubernetes，不支持 Docker 和 Docker Swarm，并且也兼容 OCI 及 CRI 标准。Containerd 可直接调用 runC 实现对容器的生命周期管理。

如图 1.15 展示了 CRI-O 运行时与 Kubernetes CRI 的适配，其中 Conmon 对应图 1.14 中的 Containerd-shim。

图 1.15 CRI-O CRI 适配图

Docker

Kubernetes 也使用 Docker 作为容器运行时和 Kubelet 进行集成，进而实现 CRI 适配，如图 1.16 所示：

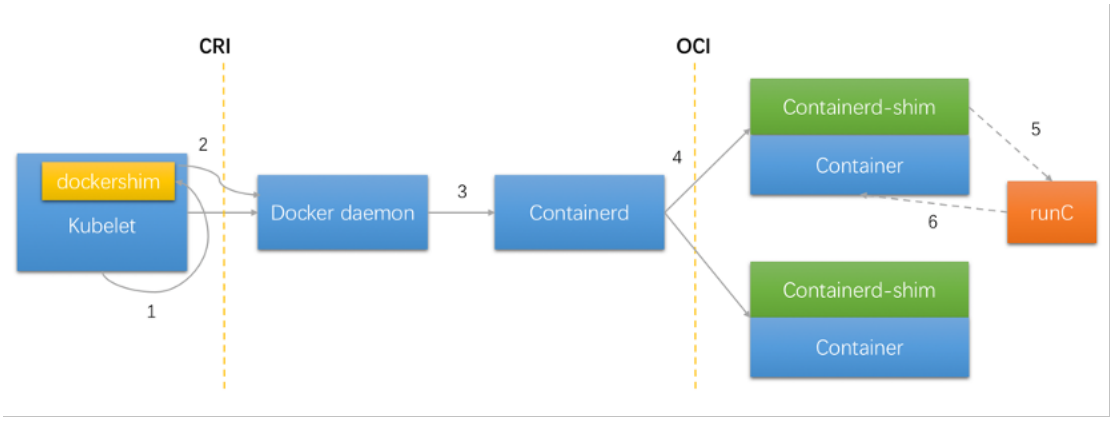

图 1.16 Docker CRI 适配图

图 1.16 可以看出 Docker 适配 CRI 接口流程，主要包含以下步骤：

1. Kubelet 组件通过 CRI 接口调用 dockershim，dockershim 用于将客户端请求转化为 Docker daemon 能理解的语言，
2. Docker Daemon 向 Containerd 发起创建容器的请求；
3. Containerd 收到请求后不会自己操作容器，而是创建一个名为 Container-shim 的进程去操作容器，这么做是因为容器进程需要有一个父进程来收集容器的状态，若使用 Containerd 作为父进程，当 Containerd 挂掉或升级时，整个宿主机上的容器均会退出，而 Container-shim 可以很好的解决这个问题；
4. runC 做为 OCI 的标准实现在 Docker 中被频繁调用，此例中，Containerd-shim 会调用 runC 来启动容器；
5. runC 启动容器完成后自动退出，此时，Containerd-shim 成为容器进程的父进程，负责收集容器的状态，并上报给 Containerd，在容器销毁时，Containerd-shim 将容器中的子进程进行清理，以避免出现僵尸进程；

1.5.4 容器运行时底层实现

通过前面对各类容器运行时的介绍我们可以看出，创建容器的过程最终会调用 runC，而 runC 是在 libcontainer 的基础上演化而来。libcontainer 通过克隆内核调用直接创

建容器，此处用到的内核调用主要包括 Cgroups 和 Namespace，前者用于限制和隔离一组进程对系统资源的使用，后者用于实现操作系统进程间的有效隔离。

1.5.4.1 控制组（Cgroups）

Cgroups 全称为 Linux Control Groups，其在 2016 年由 Google 公司团队启动研发，并在 Linux 2.6.24 内核版本中发布，为 Linux 内核的一重要特性，它可以限制、记录和隔离操作系统中进程组所使用的物理资源，例如 CPU、内存、IO 等，其主要功能包括：

资源限制：限制进程使用的资源上限，例如最大内存、文件系统缓存使用限制等；

优先级控制：不同控制组有不同的优先级，例如 CPU 使用和磁盘 IO 吞吐；

审计：计算 group 的资源使用情况；

控制：重启一组进程或挂起一组进程；

1.5.4.2 命名空间(Namespace)

Linux namespace 是一种内核级别的资源隔离机制，它的出现让运行在同一个操作系统上的进程互不干扰，Linux 内核主要实现了以下几种资源的 Namespace:

表 1.1 Linux 内核实现的 Namespace

名称	宏定义	隔离的内容
IPC	CLONE_NEWIPC	信号量、消息队列、共享内存 (Linux 内核 2.6.19 版本实现)
Network	CLONE_NEWNET	网络设备、网络栈、端口等 (Linux 内核 2.6.24 版本实现)
Mount	CLONE_NEWNS	文件系统挂载点(Linux 内核 2.4.19 版本实现)
PID	CLONE_NEWPID	进程编号(Linux 内核 2.6.24 版本实现)
User	CLONE_NEWUSER	用户和用户组(从 Linux 内核 2.6.23 版本开始实现， Linux 内核 3.8 版本宣布完成)
UTS	CLONE_NEWUTS	主机名和 NIS 域名(Linux 内核 2.6.19 版本实现)
Cgroup	CLONE_NEWCGROUP	Cgroup 根目录 (Linux 内核 4.6 版本实现)

1.6 Docker 简明实践

本实践以 Docker 为基础，内容重点包括镜像部分、容器部分，其中镜像部分涵盖镜像的构建、管理及仓库操作等内容，容器部分涵盖容器从创建到销毁的整个过程。

1.6.1 实验环境

操作系统：Ubuntu 18.04.3

Docker 版本：19.03.3

Linux 内核版本：4.15.0-66-generic

私有镜像仓库： test.cloudnative.intra.nsfocus.com

1.6.2 镜像部分

1.6.2.1 构建镜像

1. 通过 Dockerfile 构建

首先介绍常见的 Dockerfile 指令，如下所示：

FROM — 基于哪个基础镜像构建

MAINTAINER — 镜像作者信息

RUN — 容器镜像构建的时候需运行的命令

ADD — 为容器镜像添加主机或远程 URL 的资源

WORKDIR — 镜像的工作目录

VOLUME — 给予镜像指定的挂在卷

EXPOSE — 镜像对外暴露的端口

CMD — 容器启动后执行的命令，可被 docker run 后跟的命令行参数替换

ENTRYPOINT — 容器启动后执行的命令，不可被替换

COPY — 和 ADD 类似，区别为 COPY 仅支持本地资源复制

ENV — 设置环境变量，如设置数据库用户名密码

以下是个简单的 Dockerfile：

```
FROM ubuntu:18.04
MAINTAINER cloudnative@nsfocus.com
RUN apt-get update && apt-get install -y nginx
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

该 Dockerfile 基于 ubuntu:latest 基础镜像去构建 Nginx 镜像，在本地我们已经有了 ubuntu:18.04 基础镜像，如果你的本地环境没有该镜像可通过如下命令进行获取：

```
docker pull ubuntu:18.04
```

上述命令会自动从 Docker 的官方镜像仓库 dockerhub 中下载此镜像。结合上述 Dockerfile 我们可通过 docker build 命令进行镜像构建，如下所示：

```
root@docker: ~ # docker build -t test:v1 .
Sending build context to Docker daemon  4.096kB
Step 1/5 : FROM ubuntu:18.04
 ---> 15ab9f7b0886
```

```
Step 2/5 : MAINTAINER cloudnative@nsfocus.com
 ---> .....
 ---> 11279a4b4741
Step 3/5 : RUN apt-get update && apt-get install -y nginx
 ---> .....
 ---> 650a9bfc081b
Step 4/5 : EXPOSE 80
 ---> .....
 ---> 4f2d83226afe
Step 5/5 : CMD ["nginx", "-g", "daemon off;"]
 ---> .....
 ---> 4056262454c2
Successfully built 4056262454c2
Successfully tagged test:v1
```

从该 docker build 命令中我们可以看出构建的镜像名为 test:v1，当然开发者也可以自己命名。从上述输出内容的先后顺序我们可以看出，docker 构建镜像是按照 Dockerfile 中开发者定义的命令由上至下依次执行的，且均会生成对应的层。我们可通过如下命令查看该镜像的层：

```
root@docker: ~ # docker history test:v1
IMAGE        CREATED      CREATED BY                  SIZE  COMMENT
4056262454c2 14 minutes ago /bin/sh -c #(nop)  CMD ·····     0B
4f2d83226afe  14 minutes ago /bin/sh -c #(nop)  EXPOSE 80    0B
650a9bfc081b  14 minutes ago /bin/sh -c apt-get update ···     91.5MB
11279a4b4741  15 minutes ago /bin/sh -c #(nop)   ···          0B
```

2. 通过 docker commit 构建

除了通过 Dockerfile 构建镜像，我们还可以通过 docker commit 命令构建镜像，两者主要区别为生成镜像的大小差异，通过 Dockerfile 构建的镜像只有在 docker build 时才会增添新的资源，而 docker commit 基本上是获取运行中的容器快照保存为镜像，如果运行的容器正在生成大量日志或更新包文件，那么当 docker commit 后这些数据会被保存至镜像中，因而通常 docker commit 构建的镜像相对 Dockerfile 构建的要大一些，下面我们通过 docker commit 命令构建上述 test:v1 镜像，在此之前，我们需要先将镜像启动为容器，如下所示：

```
root@docker: ~ # docker run -dt --name test-nginx -p 8080:80 test:v1
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2
```

通过 docker exec 命令进入容器并安装网络配置工具 net-tools，如下所示：

```
root@docker: ~ # docker exec -it
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2 bash
```

```
root@77854d02b3b5:/# apt-get install net-tools
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  net-tools
0 upgraded, 1 newly installed, 0 to remove and 63 not upgraded.
Need to get 194 kB of archives.
After this operation, 803 kB of additional disk space will be used.
Get:1 http://mirrors.163.com/ubuntu bionic/main amd64 net-tools amd64
1.60+git20161116.90da8a0-1ubuntu1 [194 kB]
Fetched 194 kB in 0s (887 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package net-tools.
(Reading database ... 7484 files and directories currently installed.)
Preparing to unpack .../net-tools_1.60+git20161116.90da8a0-1ubuntu1_amd64.deb ...
Unpacking net-tools (1.60+git20161116.90da8a0-1ubuntu1) ...
Setting up net-tools (1.60+git20161116.90da8a0-1ubuntu1) ...
```

然后退出容器，在宿主机上通过 docker commit 构建一个名为 test:v2 的新镜像，如下所示：

```
root@77854d02b3b5:/# exit
root@docker:/home/nsfocus/ #
root@docker:/home/nsfocus/# docker commit 77854d02b3b5 test:v2
sha256:0350e1129afb6a5c77e38059a0cc067cd6ce11da2258ab4504dcb6b2c0feb893
```

1.6.2.2 镜像查看

我们还可通过 docker 命令对自己构建的镜像或官方的镜像进行管理。

例如通过 docker images 命令显示宿主机下所有的镜像：

```
root@docker: ~ # docker images
REPOSITORY  TAG     IMAGE ID      CREATED            SIZE
ubuntu      latest    f63181f19b2f  5 weeks ago        72.9MB
quay.io/kiali/kiali v1.26  8d5c44c165c4  2 months ago       165MB
istio/proxyv2  1.8.1    e4172523c1de   2 months ago       314MB
istio/pilot    1.8.1    ca7541d949ef    2 months ago       263MB
grafana/grafana  7.2.1  2f17bd84b75d   4 months ago       180MB
nginx       1.9.1     94ec7e53edfc    5 years ago        133MB
```

例如通过 docker rmi <container_id>/<container_name>命令删除其中某一镜像：

```
root@docker: ~ # docker rmi nginx:1.9.1
Untagged: nginx:1.9.1
Untagged:
```

```
nginx@sha256:2f68b99bc0d6d25d0c56876b924ec20418544ff28e1fb89a4c27679a40da811b
Deleted: sha256:94ec7e53edfc793d6d8412b4748cd84270da290ce9256730eb428574f98f7c95
Deleted: sha256:77cdb54f785ed2ff2cc1cff0acf7a6579d6b87a61bf43d2a2500db33678d1d01
……
Deleted: sha256:d55f823e63e3bd76ed8a77582f6701fe86bbceb674e953fc218df06d10f99da5
```

1.6.2.3 镜像上传

我们也可以通过 docker push 命令上传某个镜像至镜像仓库，本例中我们搭建了私有 harbor 仓库，关于 harbor 的搭建可以参考官方文档[15]，下面我们将上述构建的 test:v1 镜像上传至 harbor 仓库，如下所示：

首先为 test:v1 镜像打标签：

```
root@docker: ~ # docker tag test:v1 test.cloudnative.intra.nsfocus.com/cloudnativetest/test:v1
```

通过 docker push 命令上传 test:v1 镜像至 harbor 仓库的 cloudnativetest 项目中：

```
root@docker: ~ # docker push test.cloudnative.intra.nsfocus.com/cloudnativetest/test:v1
The push refers to repository [test.cloudnative.intra.nsfocus.com/cloudnativetest/test]
ebf0173257e3: Pushed
66072aeccae0: Pushed
2c77720cf318: Pushed
1f6b6c7dc482: Pushed
c8dbbe73b68c: Pushed
2fb7bfc6145d: Pushed
v1: digest: sha256:6bce2bc60d49e3f520969f777516e8435a71f9e26e01ea7bd91617e862b1a7c6
size: 1574
```

1.6.2.4 镜像拉取

我们也可以通过 **docker pull** 命令拉取私有仓库镜像或官方仓库镜像，如下所示：

拉取私有镜像仓库中的镜像：

```
root@docker: ~ # docker pull test.cloudnative.intra.nsfocus.com/cloudnativetest/test:v1
v1: Pulling from cloudnativetest/test
Digest: sha256:6bce2bc60d49e3f520969f777516e8435a71f9e26e01ea7bd91617e862b1a7c6
Status: Image is up to date for test.cloudnative.intra.nsfocus.com/cloudnativetest/test:v1
```

拉取官方镜像仓库中的镜像：

```
root@docker: ~ # docker pull nginx:latest
latest: Pulling from library/nginx
```

[15] https://goharbor.io/docs/2.1.0/install-config/quick-install-script/

```
45b42c59be33: Pull complete
8acc495f1d91: Pull complete
ec3bd7de90d7: Pull complete
19e2441aeeab: Pull complete
f5a38c5f8d4e: Pull complete
83500d851118: Pull complete
Digest: sha256:f3693fe50d5b1df1ecd315d54813a77afd56b0245a404055a946574deb6b34fc
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

1.6.3 容器部分

Docker 提供了相对完善的容器操作命令，下面主要为各位读者介绍容器的创建、删除、执行和管理。

1.6.3.1 容器创建

在上述镜像部分中，我们构建了 test:v1 镜像，之后可以通过 docker run 命令将构建的镜像运行为容器，具体命令如下所示：

```
docker run –dt --name test-nginx –p 8080:80 test:v1
```

上述命令中“-d”参数代表 docker 将容器在后台运行并输出该容器 ID；“-t” 参数代表 Docker 为该容器分配一个虚拟终端；” --name”参数代表为容器取一个名字；“-p”参数代表设置该容器的端口映射，其中冒号左边代表宿主机映射端口，右边代表容器端口，容器端口也是 Dockerfile 中设置的暴露端口，最后 test:v1 代表通过此镜像启动容器。终端上的操作如下所示：

```
root@docker: ~ # docker run -dt --name test-nginx -p 8080:80 test:v1
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2
```

“77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2” 代表该容器的 ID。

完整的容器运行参数可通过 docker run --help 进行查看，此处不再赘述。

1.6.3.2 容器执行

在应用开发过程中，我们不可避免要进入容器并进行相应调试，Docker 也提供了相应的命令，如下述所示：

```
root@docker: ~ # docker exec -it
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2 bash
```

其中 “-it “参数中” -t “代表分配一个伪终端，” -i “代表保持交互模式;” bash “代表以 bash 方式与容器进行交互。

完整的容器执行参数可通过 docker exec --help 进行查看，此处不再赘述。

1.6.3.3 容器删除

当我们决定删除创建的容器时，可通过 docker rm 命令实现，例如我们删除上述运行的 ID 为 “77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2 “的容器，如下所示：

```
root@docker: ~ # docker rm
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2
Error response from daemon: You cannot remove a running container
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2. Stop the container
before attempting removal or force remove
```

可以看出删除容器时出现了错误，原因是该容器正在运行中，我们可以通过 docker stop 先停止此容器，再通过 docker rm 删除或直接通过 docker rm –f 强制删除，本例中我们使用第二种方法删除容器，如下所示：

```
root@docker: ~ # docker rm -f
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2
77854d02b3b575a51acfd1a863fe2f75c4f5c6d0d884cac1b19c033d9c67b6e2
```

1.6.3.4 容器查看

Docker 支持查看所有容器或单一容器

例如通过 docker ps 查看所有正在运行的容器：

```
root@docker: ~ # docker ps
CONTAINER ID   IMAGE                                                    COMMAND
CREATED             STATUS              PORTS                                                NAMES

d83950cc651b        k8s.gcr.io/pause:3.1                                                     "/pause"
4 weeks ago         Up 4 weeks
k8s_POD_nginx-797b47466d-gl9qk_default_921f6758-a887-4b1a-a47f-b300d0fc33f8_0
a723d7659216        84581e99d807

"nginx -g 'daemon of…"   4 weeks ago         Up 4 weeks
k8s_nginx_nginx-b8dcdd49d-mnrtw_default_38b6f5d4-0914-45f7-8436-5b2707a13b36_0
```

例如通过 docker ps –a 查看所有容器（包含正在运行及停止运行的所有容器）

```
root@docker: ~ # docker ps -a
```

```
CONTAINER ID        IMAGE                                                    COMMAND
CREATED             STATUS                      PORTS                        NAMES
5649365ced6f        d66af524a87f                                             "/bin/sh -c
'apt-get…"   2 hours ago         Exited (100) 2 hours ago
musing_bhabha
4d05c40fb056        dafe84f7cc3a                                             "/bin/sh -c
'apt-get…"   2 hours ago         Exited (100) 2 hours ago
quirky_poincare
d83950cc651b        k8s.gcr.io/pause:3.1                                     "/pause"
4 weeks ago         Up 4 weeks
k8s_POD_nginx-797b47466d-gl9qk_default_921f6758-a887-4b1a-a47f-b300d0fc33f8_0
a723d7659216        84581e99d807                                             "nginx -g
'daemon of…"   4 weeks ago         Up 4 weeks
k8s_nginx_nginx-b8dcdd49d-mnrtw_default_38b6f5d4-0914-45f7-8436-5b2707a13b36_0
5a2d83027f9d        84581e99d807                                             "nginx -g
'daemon of…"   4 weeks ago         Up 4 weeks
k8s_nginx_nginx-b8dcdd49d-hhdcj_default_c9695b08-1a00-4ee2-b1c6-8bba6ad9d3ee_0
```

例如通过 docker inspect <container_id>查看单一容器:

```
root@docker: ~ # docker inspect d83950cc651b
[
    {
        "Id": "d83950cc651b2d2320c3c3977ea13622289ae9398d521172dba7ebf0a11ac2d2",
        "Created": "2021-01-25T05:06:36.102783954Z",
        "Path": "/pause",
        "Args": [],
        "State": {
         ......
        },
        "Image":
"sha256:da86e6ba6ca197bf6bc5e9d900febd906b133eaa4750e6bed647b0fbe50ed43e",
         ......
        "HostsPath":
"/var/lib/docker/containers/d83950cc651b2d2320c3c3977ea13622289ae9398d521172dba7ebf0a1
1ac2d2/hosts",
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID":
```

```
"d273164cd48e89103bc99358f04c8974806b91b592155dd06c581f06d6533805",
        ……
          }
        }
]
```

1.6.4 其它部分

1.6.4.1 日志查看

我们可通过 docker log <container_id>对容器日志进行查看，例如我们对 Kubernetes 的 API Server 组件日志进行查看：

```
root@docker: ~ # docker logs b5f9b1a0f1b8
I1209 03:33:15.493877       1 serving.go:319] Generated self-signed cert in-memory
W1209 03:33:16.688635       1 authentication.go:199] Error looking up in-cluster authentication configuration: Get https://10.65.152.107:6443/api/v1/namespaces/kube-system/configmaps/extension-apiserver-authentication: dial tcp 10.65.152.107:6443: connect: connection refused
W1209 03:33:16.688676       1 authentication.go:200] Continuing without authentication configuration. This may treat all requests as anonymous.
W1209 03:33:16.688687       1 authentication.go:201] To require authentication configuration lookup to succeed, set --authentication-tolerate-lookup-failure=false
I1209 03:33:16.708335       1 server.go:143] Version: v1.16.2
I1209 03:33:16.708426       1 defaults.go:91] TaintNodesByCondition is enabled, PodToleratesNodeTaints predicate is mandatory
W1209 03:33:16.752097       1 authorization.go:47] Authorization is disabled
W1209 03:33:16.752128       1 authentication.go:79] Authentication is disabled
I1209 03:33:16.752148       1 deprecated_insecure_serving.go:51] Serving healthz insecurely on [::]:10251
I1209 03:33:16.752906       1 secure_serving.go:123] Serving securely on 127.0.0.1:10259
E1209 03:33:17.158556       1 reflector.go:123] k8s.io/kubernetes/cmd/kube-scheduler/app/server.go:236: Failed to list *v1.Pod: Get https://10.65.152.107:6443/api/v1/pods?fieldSelector=status.phase%21%3DFailed%2Cstatus.phase%21%3DSucceeded&limit=500&resourceVersion=0: dial tcp 10.65.152.107:6443: connect: connection refused
……
```

1.6.4.2 容器环境下的复制

在容器中开发应用程序时，我们可以使用 docker cp 用于容器与宿主机之间的数据拷贝，如下所示：

```
root@docker: ~ # docker cp source.list
b9f88b2c51df95bd5bc044dd92a5a5d14b79fe23d5202859731f9e5aa0c42dbc:/etc/apt/source.lis
```

可以看出上述命令实则替换了容器内 Ubuntu 操作系统的镜像源。

1.6.4.3 镜像导入和导出

对于没有私有镜像仓库的场景，若想将本地镜像放在其它服务器上执行，我们可通过 docker save 首先对镜像进行压缩导入，再通过 docker load 进行导出，具体操作如下所示：

```
root@docker: ~ # docker save test:v1 > /tmp/test.tar
root@docker: ~ # docker load < /tmp/test.tar
Loaded image: test:v1
```

若想了解其它 docker 命令的具体用法，可参考官方文档[16]

1.7 小结

本章为各位读者较为全面的介绍了容器的基础实现技术，这一章是云原生技术的基础同时也非常重要，深度理解容器技术有助于在云原生安全领域带来更多的思考。

[i] Docker storage drivers, https://docs.docker.com/storage/storagedriver/select-storage-driver/

16 https://docs.docker.com/engine/reference/commandline/cli/