

- 容器技术是继大数据和云计算之后又一炙手可热的技术，而且未来相当一段时间内都会非常流行
- 对 IT 从业者来说，掌握容器技术是市场的需要，也是提升自我价值的重要途径
- 每一轮新技术的兴起，无论对公司还是个人既是机会也是挑战



Docker is the leading Containers as a Service platform

每天5分钟 玩转Docker容器技术

CloudMan 著

清华大学出版社



是这种架构的基石。市场将需要更多能够开发出基于容器的应用程序的软件开发人员。

2. IT 实施和运维工程师

容器为应用提供了更好的打包和部署方式，越来越多的应用将以容器的方式在开发、测试和生产环境中运行。掌握容器相关技术将成为实施和运维工程师的核心竞争力。

3. 我自己

我坚信最好的学习方法是分享。编写这个教程同时也是对自己学习和实践容器技术的总结。对于知识，只有把它写出来并能够让其他人理解，才能说明真正掌握。

包含哪些内容

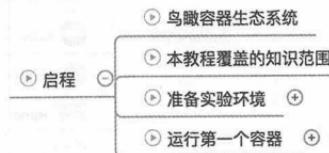
本系列教程分为《每天 5 分钟玩转 Docker 容器技术》和《每天 5 分钟玩转 Docker 容器平台》两本，包括以下三大块内容：



下面分别介绍各部分包含的内容。

1. 启程

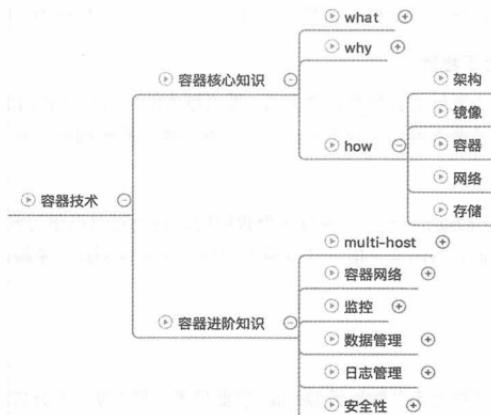
如下图所示，“启程”会介绍容器的生态系统，让大家先从整体上了解容器包含哪些技术，各种技术之间的相互关系是什么，然后再来看我们的教程都会涉及生态中的哪些部分。



为了让大家尽快对容器有个感性认识，我们会搭建实验环境并运行第一个容器，为之后的学习热身。

2. 容器技术

“容器技术”主要内容如下图所示，包含“容器核心知识”和“容器进阶知识”两部分。



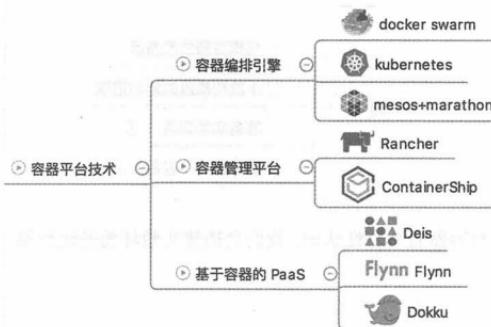
核心知识主要回答有关容器 What、Why 和 How 三方面的问题，其中以 How 为重，将展开讨论架构、镜像、容器、网络和存储。

进阶知识包括将容器真正用于生成所必需的技术，包括多主机管理、跨主机网络、监控、数据管理、日志管理和安全管理。

这部分内容将在本书《每天 5 分钟玩转 Docker 容器技术》中详细讨论。

3. 容器平台技术

如下图所示，“容器平台技术”包括容器编排引擎、容器管理平台和基于容器的 PaaS。容器平台技术在生态环境中占据着举足轻重的位置，对于容器是否能够落地，是否能应用于生产至关重要。



我们将在本系列教程的另一本书《每天 5 分钟玩转 Docker 容器平台》中详细讨论容器编排引擎、容器管理平台和基于容器的 PaaS，学习和实践业界最具代表性的开源产品。

怎样的编写方式

我会继续采用《每天 5 分钟玩转 OpenStack》（本书已在清华出版）的方式，通过大量的实验由浅入深地探讨和实践容器技术，力求达到如下目标：

- (1) 快速上手：以最直接、最有效的方式让大家把容器用起来。
- (2) 循序渐进：由易到难、从浅入深，详细分析容器的各种功能和配置使用方法。
- (3) 理解架构：从设计原理和架构分析入手，深入探讨容器的架构和运行机理。
- (4) 注重实践：以大量实际操作案例为基础，让大家能够掌握真正的实施技能。

在内容的发布上还是通过微信公众号（cloudman6）每周一、三、五定期分享。欢迎大家通过公众号提出问题和建议，进行技术交流。

为什么叫《每天 5 分钟玩转 Docker 容器技术》

为了降低学习的难度并且考虑到移动端碎片化阅读的特点，每次推送的内容大家只需要花 5 分钟就能看完（注意这里说的是看完，有时候完全理解可能需要更多时间哈），每篇内容包含 1~3 个知识点，这就是我把本书命名为《每天 5 分钟玩转 Docker 容器技术》的原因。虽然是碎片化推送，但整个教程是系统、连贯和完整的，只是化整为零了。

好了，今天这 5 分钟算是开了个头，下面我们正式开始玩转 Docker 容器技术。

编 者
2017 年 7 月

目 录

第一篇 启 程

第 1 章 鸟瞰容器生态系统	3
1.1 容器生态系统	3
1.2 本教程覆盖的知识范围	10
1.3 准备实验环境	10
1.3.1 环境选择	10
1.3.2 安装 Docker	10
1.4 运行第一个容器	11
1.5 小结	12

第二篇 容器技术

第 2 章 容器核心知识概述	15
2.1 What —— 什么是容器	15
2.2 Why —— 为什么需要容器	16
2.2.1 容器解决的问题	16
2.2.2 Docker 的特性	20
2.2.3 容器的优势	20
2.3 How —— 容器是如何工作的	21
2.4 小结	24
第 3 章 Docker 镜像	26
3.1 镜像的内部结构	26
3.1.1 hello-world —— 最小的镜像	26
3.1.2 base 镜像	27
3.1.3 镜像的分层结构	30
3.2 构建镜像	32
3.2.1 docker commit	32
3.2.2 Dockerfile	34
3.3 RUN vs CMD vs ENTRYPOINT	42
3.3.1 Shell 和 Exec 格式	42
3.3.2 RUN	44
3.3.3 CMD	44
3.3.4 ENTRYPOINT	45
3.3.5 最佳实践	46

3.4 分发镜像	46
3.4.1 为镜像命名	46
3.4.2 使用公共 Registry	49
3.4.3 搭建本地 Registry	51
3.5 小结	52
第 4 章 Docker 容器	55
4.1 运行容器	55
4.1.1 让容器长期运行	56
4.1.2 两种进入容器的方法	57
4.1.3 运行容器的最佳实践	59
4.1.4 容器运行小结	59
4.2 stop/start/restart 容器	60
4.3 pause / unpause 容器	61
4.4 删除容器	61
4.5 State Machine	62
4.6 资源限制	65
4.6.1 内存限额	65
4.6.2 CPU 限额	66
4.6.3 Block IO 带宽限额	68
4.7 实现容器的底层技术	69
4.7.1 cgroup	70
4.7.2 namespace	70
4.8 小结	72
第 5 章 Docker 网络	74
5.1 none 网络	74
5.2 host 网络	75
5.3 bridge 网络	76
5.4 user-defined 网络	78
5.5 容器间通信	84
5.5.1 IP 通信	84
5.5.2 Docker DNS Server	85
5.5.3 joined 容器	85
5.6 将容器与外部世界连接	87
5.6.1 容器访问外部世界	87
5.6.2 外部世界访问容器	90
5.7 小结	91
第 6 章 Docker 存储	92
6.1 storage driver	92
6.2 Data Volume	94

6.2.1 bind mount.....	94
6.2.2 docker managed volume.....	96
6.3 数据共享	99
6.3.1 容器与 host 共享数据	99
6.3.2 容器之间共享数据	99
6.4 volume container	100
6.5 data-packed volume container	102
6.6 Data Volume 生命周期管理	103
6.6.1 备份	104
6.6.2 恢复	104
6.6.3 迁移	104
6.6.4 销毁	104
6.7 小结	105

第三篇 容器进阶知识

第 7 章 多主机管理	109
7.1 实验环境描述	110
7.2 安装 Docker Machine	111
7.3 创建 Machine	112
7.4 管理 Machine	114
第 8 章 容器网络	117
8.1 libnetwork & CNM	117
8.2 overlay	119
8.2.1 实验环境描述	120
8.2.2 创建 overlay 网络	121
8.2.3 在 overlay 中运行容器	122
8.2.4 overlay 网络连通性	124
8.2.5 overlay 网络隔离	126
8.2.6 overlay IPAM	127
8.3 macvlan	127
8.3.1 准备实验环境	127
8.3.2 创建 macvlan 网络	128
8.3.3 macvlan 网络结构分析	130
8.3.4 用 sub-interface 实现多 macvlan 网络	131
8.3.5 macvlan 网络间的隔离和连通	132
8.4 flannel	136
8.4.1 实验环境描述	137
8.4.2 安装配置 etcd	137
8.4.3 build flannel	138

8.4.4 将 flannel 网络的配置信息保存到 etcd	139
8.4.5 启动 flannel	139
8.4.6 配置 Docker 连接 flannel	141
8.4.7 将容器连接到 flannel 网络	143
8.4.8 flannel 网络连通性	144
8.4.9 flannel 网络隔离	146
8.4.10 flannel 与外网连通性	146
8.4.11 host-gw backend	146
8.5 weave	148
8.5.1 实验环境描述	148
8.5.2 安装部署 weave	149
8.5.3 在 host1 中启动 weave	149
8.5.4 在 host1 中启动容器	150
8.5.5 在 host2 中启动 weave 并运行容器	153
8.5.6 weave 网络连通性	154
8.5.7 weave 网络隔离	155
8.5.8 weave 与外网的连通性	156
8.5.9 IPAM	158
8.6 calico	158
8.6.1 实验环境描述	159
8.6.2 启动 etcd	159
8.6.3 部署 calico	160
8.6.4 创建 calico 网络	161
8.6.5 在 calico 中运行容器	161
8.6.6 calico 默认连通性	164
8.6.7 calico policy	167
8.6.8 calico IPAM	169
8.7 比较各种网络方案	170
8.7.1 网络模型	171
8.7.2 Distributed Store	171
8.7.3 IPAM	171
8.7.4 连通与隔离	172
8.7.5 性能	172
第 9 章 容器监控	173
9.1 Docker 自带的监控子命令	173
9.1.1 ps	173
9.1.2 top	174
9.1.3 stats	175
9.2 sysdig	175
9.3 Weave Scope	179

9.3.1 安装	179
9.3.2 容器监控	181
9.3.3 监控 host	184
9.3.4 多主机监控	186
9.4 cAdvisor	189
9.4.1 监控 Docker Host	189
9.4.2 监控容器	191
9.5 Prometheus	194
9.5.1 架构	194
9.5.2 多维数据模型	195
9.5.3 实践	196
9.6 比较不同的监控工具	204
9.7 几点建议	205
第 10 章 日志管理	207
10.1 Docker logs	207
10.2 Docker logging driver	209
10.3 ELK	211
10.3.1 日志处理流程	211
10.3.2 安装 ELK 套件	212
10.3.3 Filebeat	214
10.3.4 管理日志	216
10.4 Fluentd	220
10.4.1 安装 Fluentd	221
10.4.2 重新配置 Filebeat	221
10.4.3 监控容器日志	221
10.5 Graylog	222
10.5.1 Graylog 架构	222
10.5.2 部署 Graylog	223
10.5.3 配置 Graylog	225
10.5.4 监控容器日志	227
10.6 小结	229
第 11 章 数据管理	230
11.1 从一个例子开始	230
11.2 实践 Rex-Ray driver	232
11.2.1 安装 Rex-Ray	232
11.2.2 配置 VirtualBox	234
11.2.3 创建 Rex-Ray volume	236
11.2.4 使用 Rex-Ray volume	237
写在最后	243

第一篇 启 程

对于像容器这类平台级别的技术，通常涉及的知识范围会很广，相关的软件，解决方案也会很多，初学者往往容易迷失。

那怎么办呢？

我们可以从生活经验中寻找答案。

当我们去陌生城市旅游想了解一下这个城市，一般我们会怎么做？

我想大部分人应该会打开手机看一下这个城市的地图：

- (1) 城市大概的位置和地理形状是什么？
- (2) 都由哪几个区或县组成？
- (3) 主要的交通干道是哪几条？

同样的道理，学习容器技术我们可以先从天上鸟瞰一下：

- (1) 容器生态系统包含哪些不同层次的技术？
- (2) 不同技术之间是什么关系？
- (3) 哪些是核心技术？哪些是辅助技术？

首先得对容器技术有个整体认识，之后我们的学习才能够有的放矢，才能够分清轻重缓急，做到心中有数，这样就不容易迷失了。

接下来我会根据自己的经验帮大家规划一条学习路线，一起探索容器生态系统。

学习新技术得到及时反馈是非常重要的，所以我们马上会搭建实验环境，并运行第一个容器，感受什么是容器。

千里之行始于足下，让我们从了解生态系统开始吧。

第 1 章

◀ 鸟瞰容器生态系统 ▶

1.1 容器生态系统

一谈到容器，大家都会想到 Docker。

Docker 现在几乎是容器的代名词。确实，是 Docker 将容器技术发扬光大。同时，大家也需要知道围绕 Docker 还有一个生态系统。Docker 是这个生态系统的基石，但完善的生态系统才是保障 Docker 以及容器技术能够真正健康发展的决定因素。

大致来看，容器生态系统包含核心技术、平台技术和支持技术，如图 1-1 所示。下面分别介绍。

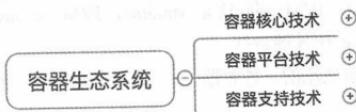


图 1-1

1. 容器核心技术

容器核心技术是指能够让 Container 在 host 上运行起来的那些技术，如图 1-2 所示。

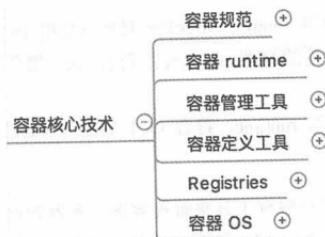


图 1-2

从上图可以看出，这些技术包括容器规范、容器 runtime、容器管理工具、容器定义工具、Registry 以及容器 OS，下面分别介绍。

(1) 容器规范

容器不光是 Docker，还有其他容器，比如 CoreOS 的 rkt。为了保证容器生态的健康发展，保证不同容器之间能够兼容，包含 Docker、CoreOS、Google 在内的若干公司共同成立了一个叫 Open Container Initiative (OCI) 的组织，其目的是制定开放的容器规范。

目前 OCI 发布了两个规范：runtime spec 和 image format spec。

有了这两个规范，不同组织和厂商开发的容器能够在不同的 runtime 上运行。这样就保证了容器的可移植性和互操作性，如图 1-3 所示。

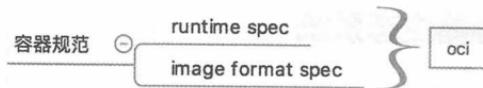


图 1-3

(2) 容器 runtime

runtime 是容器真正运行的地方。runtime 需要跟操作系统 kernel 紧密协作，为容器提供运行环境。

如果大家用过 Java，可以这样来理解 runtime 与容器的关系：

Java 程序就好比是容器，JVM 则好比是 runtime，JVM 为 Java 程序提供运行环境。同样的道理，容器只有在 runtime 中才能运行。

lxc、runc 和 rkt 是目前主流的三种容器 runtime，如图 1-4 所示。

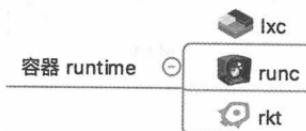


图 1-4

lxc 是 Linux 上老牌的容器 runtime。Docker 最初也是用 lxc 作为 runtime。

runc 是 Docker 自己开发的容器 runtime，符合 oci 规范，也是现在 Docker 的默认 runtime。

rkt 是 CoreOS 开发的容器 runtime，符合 OCI 规范，因而能够运行 Docker 的容器。

(3) 容器管理工具

光有 runtime 还不够，用户得有工具来管理容器。容器管理工具对内与 runtime 交互，对外为用户提供 interface，比如 CLI。这就好比除了 JVM，还得提供 Java 命令让用户能够

启停应用。容器管理工具如图 1-5 所示。

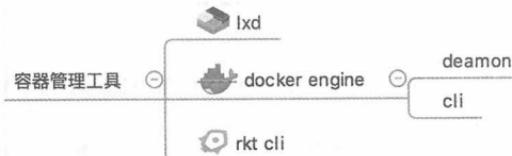


图 1-5

`lxd` 是 `lxc` 对应的管理工具。

`runc` 的管理工具是 `docker engine`。`docker engine` 包含后台 `deamon` 和 `cli` 两个部分。我们通常提到 Docker，一般就是指的 `docker engine`。

`rkt` 的管理工具是 `rkt cli`。

(4) 容器定义工具

容器定义工具允许用户定义容器的内容和属性，这样容器就能够被保存、共享和重建，如图 1-6 所示。

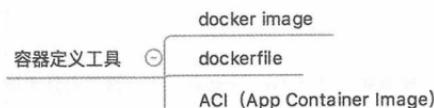


图 1-6

`docker image` 是 Docker 容器的模板，`runtime` 依据 `docker image` 创建容器。

`dockerfile` 是包含若干命令的文本文件，可以通过这些命令创建出 `docker image`。

`ACI (App Container Image)` 与 `docker image` 类似，只不过它是由 CoreOS 开发的 `rkt` 容器的 `image` 格式。

(5) Registry

容器是通过 `image` 创建的，需要有一个仓库来统一存放 `image`，这个仓库就叫做 Registry。

企业可以用 Docker Registry 构建私有的 Registry，如图 1-7 所示。

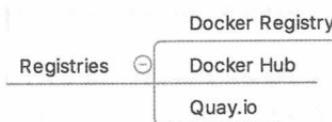


图 1-7

Docker Hub (<https://hub.docker.com>) 是 Docker 为公众提供的托管 Registry，上面有很多

现成的 image，为 Docker 用户提供了极大的便利。

Quay.io (<https://quay.io/>) 是另一个公共托管 Registry，提供与 Docker Hub 类似的服务。

(6) 容器 OS

由于有容器 runtime，几乎所有的 Linux、MAC OS 和 Windows 都可以运行容器，但这并没有妨碍容器 OS 的问世。

容器 OS 是专门运行容器的操作系统。与常规 OS 相比，容器 OS 通常体积更小，启动更快。因为是为容器定制的 OS，通常它们运行容器的效率会更高。

目前已经存在不少容器 OS，CoreOS、Atomic 和 Ubuntu Core 是其中的杰出代表，如图 1-8 所示。

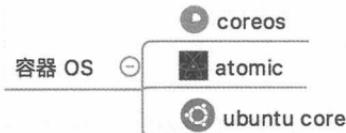


图 1-8

2. 容器平台技术

容器核心技术使得容器能够在单个 host 上运行，而容器平台技术能够让容器作为集群在分布式环境中运行。

容器平台技术包括容器编排引擎、容器管理平台和基于容器的 PaaS，如图 1-9 所示。

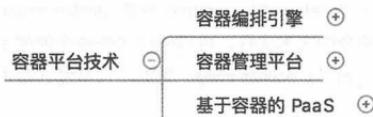


图 1-9

(1) 容器编排引擎

基于容器的应用一般会采用微服务架构。在这种架构下，应用被划分为不同的组件，并以服务的形式运行在各自的容器中，通过 API 对外提供服务。为了保证应用的高可用，每个组件都可能会运行多个相同的容器。这些容器会组成集群，集群中的容器会根据业务需要被动态地创建、迁移和销毁。

大家可以看到，这样一个基于微服务架构的应用系统实际上是一个动态的可伸缩的系统。这对我们的部署环境提出了新的要求，我们需要有一种高效的方法来管理容器集群。而这，就是容器编排引擎要干的工作。

所谓编排（orchestration），通常包括容器管理、调度、集群定义和服务发现等。通过容器编排引擎，容器被有机地组合成微服务应用，实现业务需求。

docker swarm 是 Docker 开发的容器编排引擎。

kubernetes 是 Google 领导开发的开源容器编排引擎，同时支持 Docker 和 CoreOS 容器。

mesos 是一个通用的集群资源调度平台，mesos 与 marathon 一起提供容器编排引擎功能。

以上三者是当前主流的容器编排引擎，如图 1-10 所示。

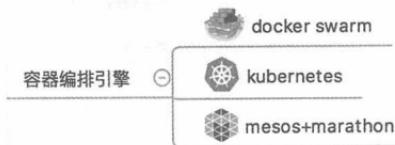


图 1-10

(2) 容器管理平台

容器管理平台是架构在容器编排引擎之上的一一个更为通用的平台。通常容器管理平台能够支持多种编排引擎，抽象了编排引擎的底层实现细节，为用户提供更方便的功能，比如 application catalog 和一键应用部署等。

Rancher 和 ContainerShip 是容器管理平台的典型代表，如图 1-11 所示。

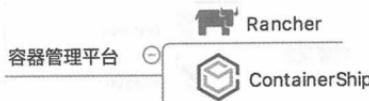


图 1-11

(3) 基于容器的 PaaS

基于容器的 PaaS 为微服务应用开发人员和公司提供了开发、部署和管理应用的平台，使用户不必关心底层基础设施而专注于应用的开发。

Deis、Flynn 和 Dokku 都是开源容器 PaaS 的代表，如图 1-12 所示。

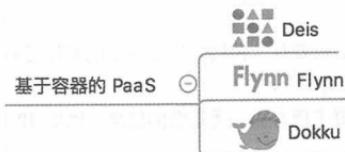


图 1-12

3. 容器支持技术

下面这些技术被用于支持基于容器的基础设施，如图 1-13 所示。

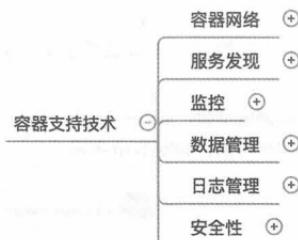


图 1-13

(1) 容器网络

容器的出现使网络拓扑变得更加动态和复杂。用户需要专门的解决方案来管理容器与容器、容器与其他实体之间的连通性和隔离性。

`docker network` 是 Docker 原生的网络解决方案。除此之外，我们还可以采用第三方开源解决方案，例如 flannel、weave 和 calico。不同方案的设计和实现方式不同，各有优势和特点，应根据实际需要来选型，如图 1-14 所示。

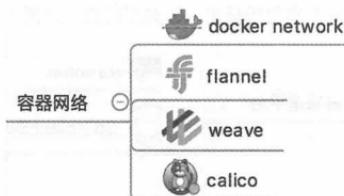


图 1-14

(2) 服务发现

动态变化是微服务应用的一大特点。当负载增加时，集群会自动创建新的容器；负载减小，多余的容器会被销毁。容器也会根据 host 的资源使用情况在不同 host 中迁移，容器的 IP 和端口也会随之发生变化。

在这种动态的环境下，必须要有一种机制让 client 能够知道如何访问容器提供的服务。这就是服务发现技术要完成的工作。

服务发现会保存容器集群中所有微服务最新的信息，比如 IP 和端口，并对外提供 API，提供服务查询功能。

`etcd`、`consul` 和 `zookeeper` 是服务发现的典型解决方案，如图 1-15 所示。

(3) 监控

监控对于基础架构非常重要，而容器的动态特征对监控提出更多挑战。针对容器环境，已经涌现出很多监控工具和方案，如图 1-16 所示。

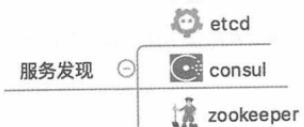


图 1-15

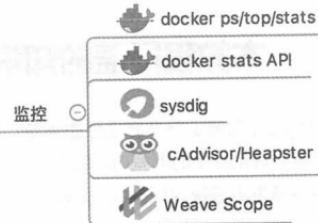


图 1-16

docker ps/top/stats 是 Docker 原生的命令行监控工具。除了命令行，Docker 也提供了 stats API，用户可以通过 HTTP 请求获取容器的状态信息。

sysdig、cAdvisor/Heapster 和 Weave Scope 是其他开源的容器监控方案。

(4) 数据管理

容器经常会在不同的 host 之间迁移，如何保证持久化数据也能够动态迁移，是 Rex-Ray 这类数据管理工具提供的能力，如图 1-17 所示。



图 1-17

(5) 日志管理

日志为问题排查和事件管理提供了重要依据。日志工具有两类，如图 1-18 所示。

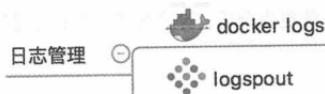


图 1-18

docker logs 是 Docker 原生的日志工具。而 logspout 对日志提供了路由功能，它可以收集不同容器的日志并转发给其他工具进行后处理。

(6) 安全性

对于年轻的容器，安全性一直是业界争论的焦点。OpenSCAP 是一种容器安全工具，如图 1-19 所示。



图 1-19

OpenSCAP 能够对容器镜像进行扫描，发现潜在的漏洞。

1.2 本教程覆盖的知识范围

前面我们已经鸟瞰了整个容器生态系统，对容器所涉及的技术体系有了全面的认识。那我们的系列教程会讨论其中的哪些内容呢？

会覆盖容器生态系统 91.6% 的技术！

本书《每天 5 分钟玩转 Docker 容器技术》详细讨论生态系统中核心技术和支持技术这两部分。

《每天 5 分钟玩转 Docker 容器平台》将会详细讨论容器平台技术。

1.3 准备实验环境

为了让大家对容器有个感性认识，我们将尽快让一个容器运行起来。首先我们需要搭建实验环境。

1.3.1 环境选择

容器需要管理工具、runtime 和操作系统，我们的选择如下：

(1) 管理工具：Docker Engine。因为 Docker 最流行使用最广泛。

(2) runtime : runc。Docker 的默认 runtime。

(3) 操作系统：Ubuntu。虽然存在诸如 CoreOS 的容器 OS，因考虑到我们目前处于初学阶段，选择大家熟悉的操作系统更为合适。等具备了扎实的容器基础知识后，再使用容器 OS 会更有利。

1.3.2 安装 Docker

本小节我们将在 Ubuntu 16.04 虚拟机中安装 Docker。因为安装过程需要访问 Internet，所以虚拟机必须能够上网。

Docker 支持几乎所有的 Linux 发行版，也支持 Mac 和 Windows。各操作系统的安装方法可以访问：<https://docs.docker.com/engine/installation/>。

Docker 分为开源免费的 CE (Community Edition) 版本和收费的 EE (Enterprise Edition) 版本。下面我们将按照文档，通过以下步骤在 Ubuntu 16.04 上安装 Docker CE 版本。

1. 配置 Docker 的 apt 源

(1) 安装包，允许 apt 命令 HTTPS 访问 Docker 源。

```
$ sudo apt-get install \ apt-transport-https \ ca-certificates \ curl
\ software-properties-common
```

(2) 添加 Docker 官方的 GPG key。

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

(3) 将 Docker 的源添加到 /etc/apt/sources.list。

```
$ sudo add-apt-repository \ "deb [arch=amd64] https://download.docker.com/linux/ubuntu \ $(lsb_release -cs) \ stable"
```

2. 安装 Docker

```
$ sudo apt-get update $ sudo apt-get install docker-ce
```

1.4 运行第一个容器

环境就绪，马上运行第一个容器，执行命令：

```
docker run -d -p 80:80 httpd
```

结果如图 1-20 所示。

```
root@ubuntu:~# docker run -d -p 80:80 httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
386e066cd84a: Pull complete
a11d6b8e2fac: Pull complete
c1fdc7bee37: Pull complete
bd14a67deca2: Pull complete
92b34ad02810: Pull complete
Digest: sha256:5b4a3c85b4b874ee84174ee7e78a59920818aa39903f6a28a47b9278f576b4a4d
Status: Downloaded newer image for httpd:latest
170edfc2065c563fbea7e00247bf644e1e449e7a632ae528f77aac48fc5c4324
root@ubuntu:~#
```

图 1-20

其过程可以简单地描述为：

- (1) 从 Docker Hub 下载 httpd 镜像。镜像中已经安装好了 Apache HTTP Server。
- (2) 启动 httpd 容器，并将容器的 80 端口映射到 host 的 80 端口。

下面可以通过浏览器验证容器是否正常工作。在浏览器中输入 `http://[your ubuntu host IP]`，如图 1-21 所示。

可以访问容器的 HTTP 服务了，看到上图所示的页面，表示第一个容器运行成功！我们轻轻松松就拥有了一个 Web 服务器。随着学习的深入，会看到容器技术带给我们更多的价值。

镜像下载加速

由于 Docker Hub 的服务器在国外，下载镜像会比较慢。幸好 DaoCloud 为我们提供了免费的国内镜像服务。

下面介绍如何使用镜像服务。

(1) 在 daocloud.io 免费注册一个用户。

(2) 登录后，单击顶部菜单“加速器”，如图 1-22 所示。

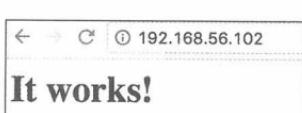


图 1-21



图 1-22

(3) copy “加速器”命令并在 host 中执行（你的命令可能跟我的会稍有不同），如图 1-23 所示。

配置 Docker 加速器

[Linux](#) [MacOS](#) [Windows](#)

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://85d5449d.m.daocloud.io
```

[Copy](#)

该脚本可以将 `--registry-mirror` 加入到你的 Docker 配置文件 `/etc/default/docker` 中。适用于 Ubuntu14.04、Debian、CentOS6、CentOS7、Fedora、Arch Linux、openSUSE Leap 42.1，其他版本可能有细微不同。更多详情请[访问文档](#)。

图 1-23

(4) 重启 Docker deamon，即可体验飞一般的感觉。

```
systemctl restart docker.service
```

1.5 小结

我们已经完成了教程的第一部分。

我们认识了容器生态系统，后面会陆续学习生态系统中的大部分技术。我们在 Ubuntu 16.04 上配置好了实验环境，并成功运行了第一个容器 `httpd`。

容器大门已经打开，让我们去探秘吧。

第二篇

容器技术

本篇通过 Docker 讨论容器技术的核心知识。



第 2 章

◀ 容器核心知识概述 ▶

容器核心知识主要回答有关容器 What、Why 和 How 三个问题。其中 How 是重点，将从架构、镜像、容器、网络和存储几个方面进行讲解。

2.1 What —— 什么是容器

容器是一种轻量级、可移植、自包含的软件打包技术，使应用程序可以在几乎任何地方以相同的方式运行。开发人员在自己笔记本上创建并测试好的容器，无须任何修改就能够在生产系统的虚拟机、物理服务器或公有云主机上运行。

容器与虚拟机

谈到容器，就不得不将它与虚拟机进行对比，因为两者都是为应用提供封装和隔离。

容器由两部分组成：（1）应用程序本身；（2）依赖：比如应用程序需要的库或其他软件。容器在 Host 操作系统的用户空间中运行，与操作系统的其他进程隔离。这一点显著区别于虚拟机。

传统的虚拟化技术，比如 VMWare、KVM、Xen，目标是创建完整的虚拟机。为了运行应用，除了部署应用本身及其依赖（通常几十 MB），还得安装整个操作系统（几十 GB）。

图 2-1 所示展示了二者的区别。

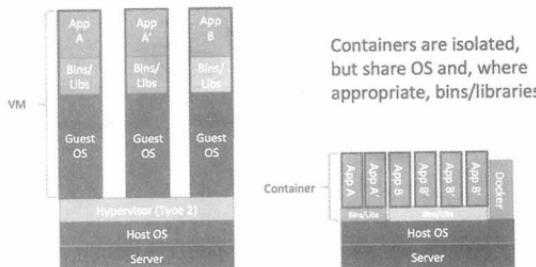


图 2-1

如图 2-1 所示,由于所有的容器共享同一个 Host OS,这使得容器在体积上要比虚拟机小很多。另外,启动容器不需要启动整个操作系统,所以容器部署和启动速度更快、开销更小,也更容易迁移。

2.2 Why —— 为什么需要容器

为什么需要容器?容器到底解决的是什么问题?简要的答案是:容器使软件具备了超强的可移植能力。

2.2.1 容器解决的问题

我们来看看今天的软件开发面临着怎样的挑战?

如今的系统在架构上较十年前已经变得非常复杂了。以前几乎所有的应用都采用三层架构(Presentation/Application/Data),系统部署到有限的几台物理服务器上(Web Server/Application Server/Database Server)。

而今天,开发人员通常使用多种服务(比如 MQ、Cache、DB)构建和组装应用,而且应用很可能会部署到不同的环境,比如虚拟服务器、私有云和公有云,如图 2-2 所示。

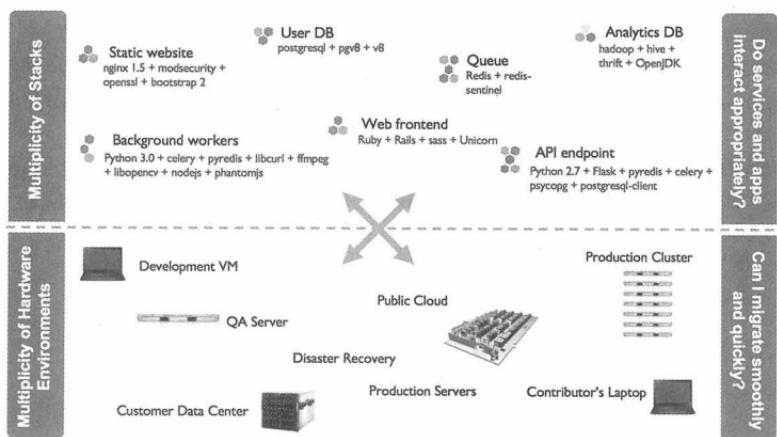


图 2-2

一方面应用包含多种服务,这些服务都有自己所依赖的库和软件包;另一方面存在多种部署环境,服务在运行时可能需要动态迁移到不同的环境中。这就产生了一个问题:如何让每种服务能够在所有的部署环境中顺利运行?

于是得到了如图 2-3 所示的这个矩阵。

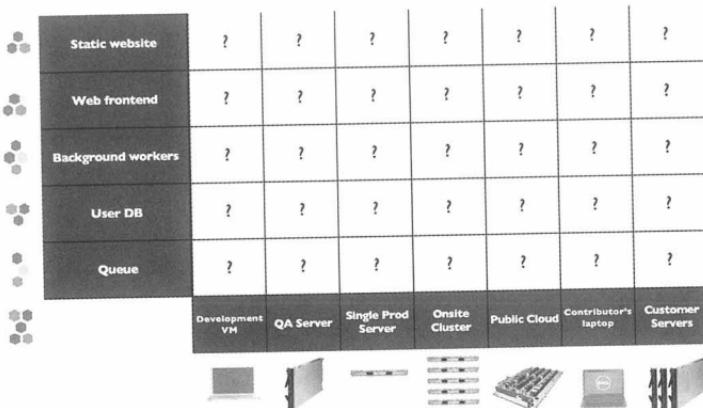


图 2-3

各种服务和环境通过排列组合产生了一个大矩阵。开发人员在编写代码时需要考虑不同的运行环境，运维人员则需要为不同的服务和平台配置环境。对他们双方来说，这都是一项困难而艰巨的任务。

如何解决这个问题呢？

聪明的技术人员从传统的运输行业找到了答案。

几十年前，运输业面临着类似的问题，如图 2-4 所示。

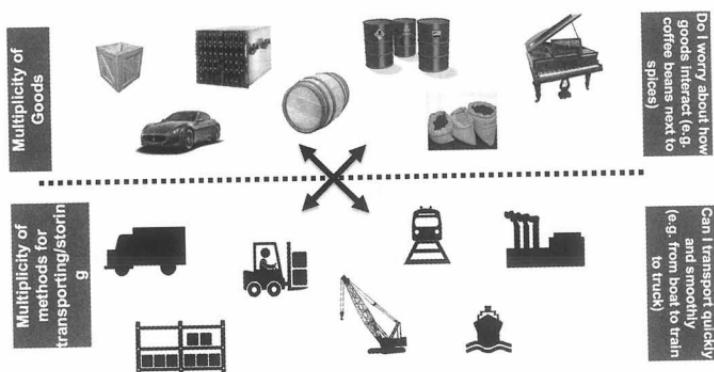


图 2-4

每一次运输，货主与承运方都会担心因货物类型的不同而导致损失，比如几个铁桶错误地压在了一堆香蕉上。另一方面，运输过程中需要使用不同的交通工具也让整个过程痛苦不堪：

货物先装上车运到码头，卸货，然后装上船，到岸后又卸下船，再装上火车，到达目的地，最后卸货。一半以上的时间花费在装货、卸货上，而且搬上搬下还容易损坏货物。

这同样也是一个 NxM 的矩阵，如图 2-5 所示。

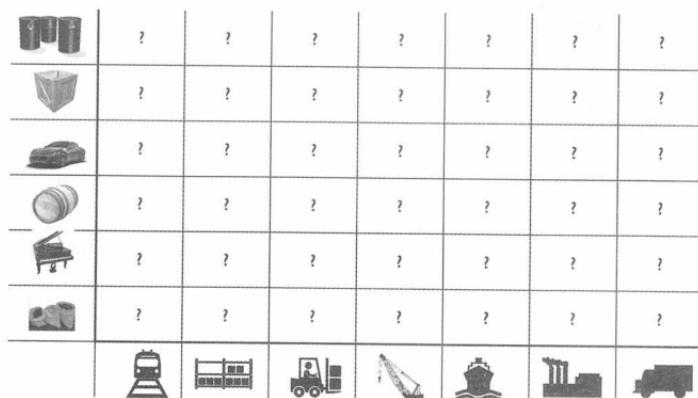


图 2-5

幸运的是，集装箱的发明解决了这个难题，如图 2-6 所示。

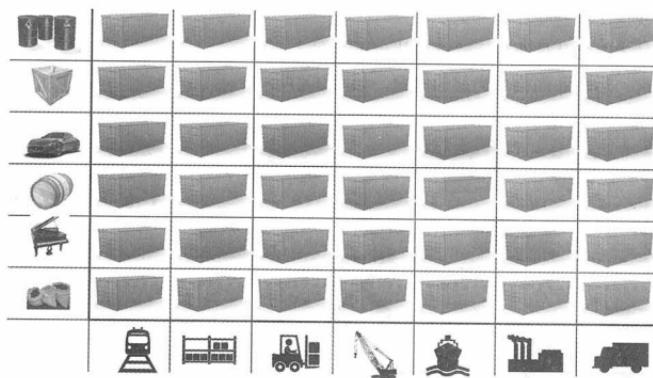


图 2-6

任何货物，无论钢琴还是保时捷，都被放到各自的集装箱中。集装箱在整个运输过程中都是密封的，只有到达最终目的地才被打开。标准集装箱可以被高效地装卸、重叠和长途运输。现代化的起重机可以自动在卡车、轮船和火车之间移动集装箱。集装箱被誉为运输业与世界贸易最重要的发明，如图 2-7 所示。

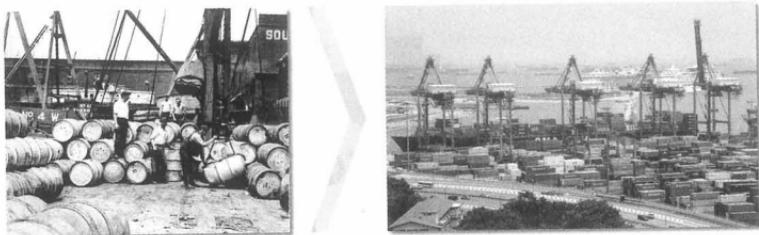


图 2-7

Docker 将集装箱思想运用到软件打包上，为代码提供了一个基于容器的标准化运输系统。Docker 可以将任何应用及其依赖打包成一个轻量级、可移植、自包含的容器。容器可以运行在几乎所有的操作系统上，如图 2-8 所示。

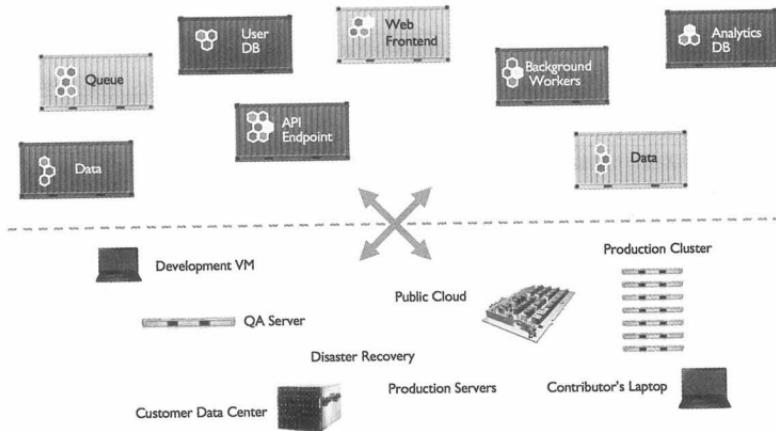


图 2-8

其实，“集装箱”和“容器”对应的英文单词都是“Container”。

“容器”是国内约定俗成的叫法，可能是因为容器比集装箱更抽象，更适合软件领域的缘故吧。

我个人认为：在外国人的思维中，“Container”只用到了集装箱这一个意思，Docker 的 Logo（如图 2-9 所示）不就是一堆集装箱吗？



图 2-9

2.2.2 Docker 的特性

可以看看集装箱思想是如何与 Docker 各种特性相对应的，如表 2-1 所示。

表 2-1 集装箱与 Docker 的特性对比

特性	集装箱	Docker
打包对象	几乎任何货物	任何软件及其依赖
硬件依赖	标准形状和接口允许集装箱被装卸到各种交通工具，整个运输过程无须打开	容器无须修改便可运行在几乎所有的平台上——虚拟机、物理机、公有云、私有云
隔离性	集装箱可以重叠起来一起运输，香蕉再也不会被铁桶压烂了	资源、网络、库都是隔离的，不会出现依赖问题
自动化	标准接口使集装箱很容易自动装卸和移动	提供 run、start、stop 等标准化操作，非常适合自动化
高效性	无须开箱，可在各种交通工具间快速搬运	轻量级，能够快速启动和迁移
职责分工	货主只需考虑把什么放到集装箱里；承运方只需关心怎样运输集装箱	开发人员只需考虑怎么写代码；运维人员只需关心如何配置基础环境

2.2.3 容器的优势

对于开发人员：Build Once、Run Anywhere。

容器意味着环境隔离和可重复性。开发人员只需为应用创建一次运行环境，然后打包成容器便可在其他机器上运行。另外，容器环境与所在的 Host 环境是隔离的，就像虚拟机一样，但更快更简单。

对于运维人员：Configure Once、Run Anything。

只需要配置好标准的 runtime 环境，服务器就可以运行任何容器。这使得运维人员的工作变得更高效、一致和可重复。容器消除了开发、测试、生产环境的不一致性。

2.3 How —— 容器是如何工作的

接下来学习容器核心知识的最主要部分。

我们首先会介绍 Docker 的架构，然后分章节详细讨论 Docker 的镜像、容器、网络和存储。

1. Docker 架构

Docker 的核心组件包括：

- Docker 客户端：Client
- Docker 服务器：Docker daemon
- Docker 镜像：Image
- Registry
- Docker 容器：Container

Docker 架构如图 2-10 所示。

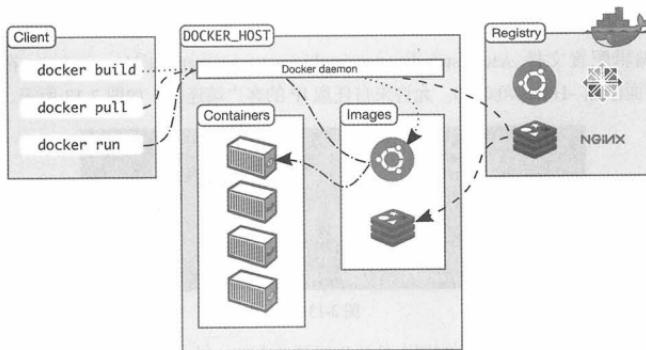


图 2-10

Docker 采用的是 Client/Server 架构。客户端向服务器发送请求，服务器负责构建、运行和分发容器。客户端和服务器可以运行在同一个 Host 上，客户端也可以通过 socket 或 REST API 与远程的服务器通信。

2. Docker 客户端

最常用的 Docker 客户端是 docker 命令。通过 docker 我们可以方便地在 Host 上构建和运行容器。

docker 支持很多操作（子命令），后面会逐步用到，如图 2-11 所示。

```
root@ubuntu:~# docker
attach  daemon  help   inspect  logs    ps      rm      service  tag     volume
build   diff    history kill     network pull    rmi    start   top     wait
commit  events  images  load    node    push   run    stats  unpause
cp      exec   import login  pause   rename save  stop   update
create  export  info   logout port   restart search swarm version
```

图 2-11

除了 docker 命令行工具，用户也可以通过 REST API 与服务器通信。

3. Docker 服务器

Docker daemon 是服务器组件，以 Linux 后台服务的方式运行，如图 2-12 所示。

```
root@ubuntu:~# systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled)
   Active: active (running) since Sat 2016-11-12 16:17:04 CST;
             Docs: https://docs.docker.com
             Main PID: 19133 (dockerd)
```

图 2-12

Docker daemon 运行在 Docker host 上，负责创建、运行、监控容器，构建、存储镜像。

默认配置下，Docker daemon 只能响应来自本地 Host 的客户端请求。如果要允许远程客户端请求，需要在配置文件中打开 TCP 监听，步骤如下：

(1) 编辑配置文件 `/etc/systemd/system/multi-user.target.wants/docker.service`，在环境变量 `ExecStart` 后面添加 `-H tcp://0.0.0.0`，允许来自任意 IP 的客户端连接，如图 2-13 所示。

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because
# /etc/init.d/docker currently does not support the cgroup
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0
ExecReload=/bin/kill -s HUP $MAINPID
```

图 2-13

如果使用的是其他操作系统，配置文件的位置可能会不一样。

(2) 重启 Docker daemon，如图 2-14 所示。

```
root@ubuntu:~# systemctl daemon-reload
root@ubuntu:~# systemctl restart docker.service
```

图 2-14

(3) 服务器 IP 为 192.168.56.102，客户端在命令行里加上 `-H` 参数，即可与远程服务器通信，如图 2-15 所示。

```
root@ubuntu:~# docker -H 192.168.56.102 info
Containers: 3
Running: 1
Paused: 0
Stopped: 2
Images: 1
```

图 2-15

info 子命令用于查看 Docker 服务器的信息。

4. Docker 镜像

可将 Docker 镜像看成只读模板，通过它可以创建 Docker 容器。

例如某个镜像可能包含一个 Ubuntu 操作系统、一个 Apache HTTP Server 以及用户开发的 Web 应用。

镜像有多种生成方法：（1）从无到有开始创建镜像；（2）下载并使用别人创建好的现成的镜像；（3）在现有镜像上创建新的镜像。

可以将镜像的内容和创建步骤描述在一个文本文件中，这个文件被称作 Dockerfile，通过执行 docker build <docker-file> 命令可以构建出 Docker 镜像，后面我们会讨论。

5. Docker 容器

Docker 容器就是 Docker 镜像的运行实例。

用户可以通过 CLI（Docker）或是 API 启动、停止、移动或删除容器。可以这么认为，对于应用软件，镜像是软件生命周期的构建和打包阶段，而容器则是启动和运行阶段。

6. Registry

Registry 是存放 Docker 镜像的仓库，Registry 分私有和公有两种。

Docker Hub (<https://hub.docker.com/>) 是默认的 Registry，由 Docker 公司维护，上面有数以万计的镜像，用户可以自由下载和使用。

出于对速度或安全的考虑，用户也可以创建自己的私有 Registry。后面我们会学习如何搭建私有 Registry。

docker pull 命令可以从 Registry 下载镜像。

docker run 命令则是先下载镜像（如果本地没有），然后再启动容器。

7. 一个完整的例子

还记得我们运行的第一个容器吗？现在通过它来体会一下 Docker 各个组件是如何协作的。

容器启动（如图 2-16 所示）过程如下：

```
root@ubuntu:~# docker run -d -p 80:80 httpd ①
Unable to find image 'httpd:latest' locally ②
latest: Pulling from library/httpd
386a066cd84a: Pull complete ③
a11d6b8e2fac: Pull complete
c1fdc7beec37: Pull complete
bd14a67deca2: Pull complete
92b34ad02810: Pull complete
Digest: sha256:5b4a3c85b4b874e84174ee7e78a59920818aa39903f6a28a47b9278f576b4a4a
Status: Downloaded newer image for httpd:latest ④
170edfc2065c5fbea7e00247bf644e1e449e7a632ae528f77aac48fc5c4324 ⑤
root@ubuntu:~#
```

图 2-16

- (1) Docker 客户端执行 docker run 命令。
- (2) Docker daemon 发现本地没有 httpd 镜像。
- (3) daemon 从 Docker Hub 下载镜像。
- (4) 下载完成, 镜像 httpd 被保存到本地。
- (5) Docker daemon 启动容器。

docker images 可以查看到 httpd 已经下载到本地, 如图 2-17 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
httpd	latest	50f10ef90911	5 days ago	193.3 MB

图 2-17

docker ps 或者 docker container ls 显示容器正在运行, 如图 2-18 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
170edfc2065c	httpd	"httpd-foreground"	46 hours ago	Up 23 seconds	0.0.0.0:80->80/tcp	tender_fermi

图 2-18

2.4 小结

Docker 借鉴了集装箱的概念。标准集装箱将货物运往世界各地, Docker 将这个模型运用到自己的设计哲学中, 唯一不同的是: 集装箱运输货物, 而 Docker 运输软件。

每个容器都有一个软件镜像, 相当于集装箱中的货物。容器可以被创建、启动、关闭和销毁。和集装箱一样, Docker 在执行这些操作时, 并不关心容器里到底装的什么, 它不管里面是 Web Server, 还是 Database。

用户不需要关心容器最终会在哪里运行, 因为哪里都可以运行。

开发人员可以在笔记本上构建镜像并上传到 Registry, 然后 QA 人员将镜像下载到物理或

虚拟机做测试，最终容器会部署到生产环境。

使用 Docker 以及容器技术，我们可以快速构建一个应用服务器、一个消息中间件、一个数据库、一个持续集成环境。因为 Docker Hub 上有我们能想到的几乎所有的镜像。

不知大家是否意识到，潘多拉盒子已经被打开。容器不但降低了我们学习新技术的门槛，更提高了效率。

如果你是一个运维人员，想研究负载均衡软件 HAProxy，只需要执行 `docker run haproxy`，无须烦琐的手工安装和配置即可以进入实战。

如果你是一个开发人员，想学习怎么用 Django 开发 Python Web 应用，执行 `docker run django`，在容器里随便折腾吧，不用担心会搞乱 Host 的环境。

不夸张地说：容器大大提升了 IT 人员的幸福指数。

第 3 章

◀ Docker 镜像 ▶

镜像是 Docker 容器的基石，容器是镜像的运行实例，有了镜像才能启动容器。

本章内容安排如下：首先通过研究几个典型的镜像，分析镜像的内部结构；然后学习如何构建自己的镜像；最后介绍怎样管理和分发镜像。

3.1 镜像的内部结构

为什么我们要讨论镜像的内部结构？

如果只是使用镜像，当然不需要了解，直接通过 docker 命令下载和运行就可以了。

但如果我们要创建自己的镜像，或者想理解 Docker 为什么是轻量级的，就非常有必要学习这部分知识了。

我们从一个最小的镜像开始吧。

3.1.1 hello-world —— 最小的镜像

hello-world 是 Docker 官方提供的一个镜像，通常用来验证 Docker 是否安装成功。我们先通过 docker pull 从 Docker Hub 下载它，如图 3-1 所示。

```
root@ubuntu:~# docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd6779851
Status: Downloaded newer image for hello-world:latest
```

图 3-1

用 docker images 命令查看镜像的信息，如图 3-2 所示。

root@ubuntu:~# docker images hello-world				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	c54a2cc56ccb	4 months ago	1.848 kB

图 3-2

hello-world 镜像竟然还不到 2KB! 通过 docker run 运行, 如图 3-3 所示。

其实我们更关心 hello-world 镜像包含哪些内容。

Dockerfile 是镜像的描述文件, 定义了如何构建 Docker 镜像。Dockerfile 的语法简洁且可读性强, 后面我们会专门讨论如何编写 Dockerfile。

hello-world 的 Dockerfile 内容如图 3-4 所示。

```
root@ubuntu:~# docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

图 3-3

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

图 3-4

只有短短三条指令。

- (1) FROM scratch 镜像是从白手起家, 从 0 开始构建。
- (2) COPY hello / 将文件 “hello” 复制到镜像的根目录。
- (3) CMD ["/hello"] 容器启动时, 执行 /hello。

镜像 hello-world 中就只有一个可执行文件 “hello”, 其功能就是打印出 “Hello from Docker” 等信息。

/hello 就是文件系统的全部内容, 连最基本的 /bin、/usr、/lib、/dev 都没有。

hello-world 虽然是一个完整的镜像, 但它并没有什么实际用途。通常来说, 我们希望镜像能提供一个基本的操作系统环境, 用户可以根据需要安装和配置软件。

这样的镜像我们称作 base 镜像。

3.1.2 base 镜像

base 镜像有两层含义: (1) 不依赖其他镜像, 从 scratch 构建; (2) 其他镜像可以以之为基础进行扩展。

所以, 能称作 base 镜像的通常都是各种 Linux 发行版的 Docker 镜像, 比如 Ubuntu、Debian、CentOS 等。

我们以 CentOS 为例考察 base 镜像包含哪些内容。

下载镜像:

```
docker pull centos
```

查看镜像信息, 如图 3-5 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
centos	latest	0584b3d2cf6d	2 weeks ago	196.5 MB

图 3-5

镜像大小不到 200MB。

等一下！

一个 CentOS 才 200MB ？

平时我们安装一个 CentOS 至少都有几个 GB，怎么可能才 200MB ！

相信这是几乎所有 Docker 初学者都会有的疑问，包括我自己。下面我们来解释这个问题。

Linux 操作系统由内核空间和用户空间组成，如图 3-6 所示。

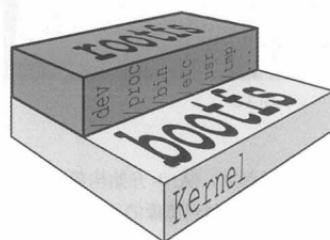


图 3-6

1. rootfs

内核空间是 kernel，Linux 刚启动时会加载 bootfs 文件系统，之后 bootfs 会被卸载掉。

用户空间的文件系统是 rootfs，包含我们熟悉的 /dev、/proc、/bin 等目录。

对于 base 镜像来说，底层直接用 Host 的 kernel，自己只需要提供 rootfs 就行了。

而对于一个精简的 OS，rootfs 可以很小，只需要包括最基本的命令、工具和程序库就可以了。相比其他 Linux 发行版，CentOS 的 rootfs 已经算臃肿的了，alpine 还不到 10MB。

我们平时安装的 CentOS 除了 rootfs 还会选装很多软件、服务、图形桌面等，需要好几个 GB 就不足为奇了。

2. base 镜像提供的是最小安装的 Linux 发行版

CentOS 镜像的 Dockerfile 的内容如图 3-7 所示。

```
FROM scratch
ADD centos-7-docker.tar.xz /
CMD ["/bin/bash"]
```

图 3-7

第二行 ADD 指令添加到镜像的 tar 包就是 CentOS 7 的 rootfs。在制作镜像时，这个 tar 包会自动解压到 / 目录下，生成 /dev、/proc、/bin 等目录。

注：可在 Docker Hub 的镜像描述页面中查看 Dockerfile 。

3. 支持运行多种 Linux OS

不同 Linux 发行版的区别主要就是 rootfs。

比如 Ubuntu 14.04 使用 upstart 管理服务，apt 管理软件包；而 CentOS 7 使用 systemd 和 yum。这些都是用户空间上的区别，Linux kernel 差别不大。

所以 Docker 可以同时支持多种 Linux 镜像，模拟出多种操作系统环境，如图 3-8 所示。

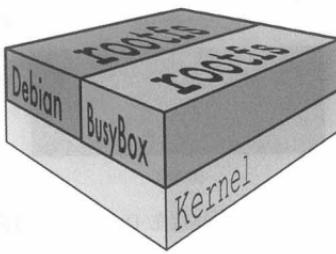


图 3-8

上图 Debian 和 BusyBox（一种嵌入式 Linux）上层提供各自的 rootfs，底层共用 Docker Host 的 kernel。

这里需要说明的是：

(1) base 镜像只是在用户空间与发行版一致，kernel 版本与发行版是不同的。

例如 CentOS 7 使用 3.x.x 的 kernel，如果 Docker Host 是 Ubuntu 16.04（比如我们的实验环境），那么在 CentOS 容器中使用的实际上是 Host 4.x.x 的 kernel，如图 3-9 所示。

```
root@ubuntu:~# uname -r          ①
4.4.0-31-generic
root@ubuntu:~#
root@ubuntu:~# docker run -it centos    ②
[root@727d35db6801 /]# cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core) ③
[root@727d35db6801 /]# uname -r
4.4.0-31-generic      ④
```

图 3-9

- ① Host kernel 为 4.4.0-31。
- ② 启动并进入 CentOS 容器。
- ③ 验证容器是 CentOS 7。
- ④ 容器的 kernel 版本与 Host 一致。

(2) 容器只能使用 Host 的 kernel，并且不能修改。

所有容器都共用 host 的 kernel，在容器中没办法对 kernel 升级。如果容器对 kernel 版本有要求（比如应用只能在某个 kernel 版本下运行），则不建议用容器，这种场景虚拟机可能更合适。

3.1.3 镜像的分层结构

Docker 支持通过扩展现有镜像，创建新的镜像。

实际上，Docker Hub 中 99% 的镜像都是通过在 base 镜像中安装和配置需要的软件构建出来的。比如我们现在构建一个新的镜像，Dockerfile 如图 3-10 所示。

```
FROM debian ①
RUN apt-get install emacs ②
RUN apt-get install apache2 ③
CMD ["/bin/bash"] ④
```

图 3-10

- ① 新镜像不再是从 scratch 开始，而是直接在 Debian base 镜像上构建。
- ② 安装 emacs 编辑器。
- ③ 安装 apache2。
- ④ 容器启动时运行 bash。

构建过程如图 3-11 所示。

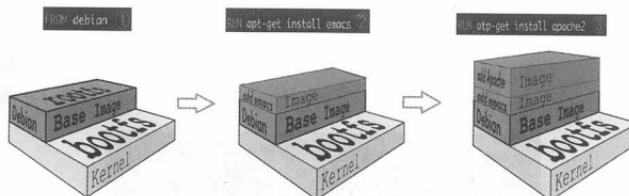


图 3-11

可以看到，新镜像是从 base 镜像一层一层叠加生成的。每安装一个软件，就在现有镜像的基础上增加一层。

为什么 Docker 镜像要采用这种分层结构呢？

最大的一个好处就是：共享资源。

比如：有多个镜像都从相同的 base 镜像构建而来，那么 Docker Host 只需在磁盘上保存一份 base 镜像；同时内存中也只需加载一份 base 镜像，就可以为所有容器服务了，而且镜像的每一层都可以被共享，我们将在后面更深入地讨论这个特性。

这时可能就有人会问了：如果多个容器共享一份基础镜像，当某个容器修改了基础镜像的内容，比如 /etc 下的文件，这时其他容器的 /etc 是否也会被修改？

答案是不会！

修改会被限制在单个容器内。

这就是我们接下来要学习的容器 Copy-on-Write 特性。

可写的容器层

当容器启动时，一个新的可写层被加载到镜像的顶部。

这一层通常被称作“容器层”，“容器层”之下的都叫“镜像层”，如图 3-12 所示。

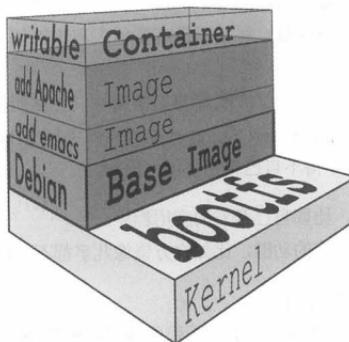


图 3-12

所有对容器的改动，无论添加、删除，还是修改文件都只会发生在容器层中。只有容器层是可写的，容器层下面的所有镜像层都是只读的。

下面我们深入讨论容器层的细节。

镜像层数量可能会很多，所有镜像层会联合在一起组成一个统一的文件系统。如果不同层中有一个相同路径的文件，比如 /a，上层的 /a 会覆盖下层的 /a，也就是说用户只能访问到上层中的文件 /a。在容器层中，用户看到的是一个叠加之后的文件系统。

- (1) 添加文件。在容器中创建文件时，新文件被添加到容器层中。
- (2) 读取文件。在容器中读取某个文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，打开并读入内存。
- (3) 修改文件。在容器中修改已存在的文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后修改之。
- (4) 删除文件。在容器中删除文件时，Docker 也是从上往下依次在镜像层中查找此文件。找到后，会在容器层中记录下此删除操作。

只有当需要修改时才复制一份数据，这种特性被称作 Copy-on-Write。可见，容器层保存的是镜像变化的部分，不会对镜像本身进行任何修改。

这样就解释了我们前面提出的问题：容器层记录对镜像的修改，所有镜像层都是只读的，不会被容器修改，所以镜像可以被多个容器共享。

3.2 构建镜像

对于 Docker 用户来说，最好的情况是不需要自己创建镜像。几乎所有常用的数据仓库、中间件、应用软件等都有现成的 Docker 官方镜像或其他人和组织创建的镜像，我们只需要稍作配置就可以直接使用。

使用现成镜像的好处除了省去自己做镜像的工作量外，更重要的是可以利用前人的经验。特别是使用那些官方镜像，因为 Docker 的工程师知道如何更好地在容器中运行软件。

当然，某些情况下我们也不得不自己构建镜像，比如：

- (1) 找不到现成的镜像，比如自己开发的应用程序。
- (2) 需要在镜像中加入特定的功能，比如官方镜像几乎都不提供 ssh。

所以本节我们将介绍构建镜像的方法。

同时分析构建的过程也能够加深我们对前面镜像分层结构的理解。

Docker 提供了两种构建镜像的方法： docker commit 命令与 Dockerfile 构建文件。

3.2.1 docker commit

docker commit 命令是创建新镜像最直观的方法，其过程包含三个步骤：

- 运行容器。
- 修改容器。
- 将容器保存为新的镜像。

举个例子：在 Ubuntu base 镜像中安装 vi 并保存为新镜像。

(1) 运行容器

如图 3-13 所示。

```
root@ubuntu:~# docker run -it ubuntu
root@412b30588f4a:/#
```

图 3-13

-it 参数的作用是以交互模式进入容器，并打开终端。412b30588f4a 是容器的内部 ID。

(2) 安装 vi

确认 vi 没有安装，如图 3-14 所示。

```
root@412b30588f4a:/# vim
bash: vim: command not found
```

图 3-14

安装 vi，如图 3-15 所示。

```
root@412b30588f4a:/# apt-get install -y vim
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  file libexpat1 libgpm2 libmagic1 libmpdec2 libpython
  libsqlite3-0 libssl1.0.0 mime-support vim-common vim

```

图 3-15

(3) 保存为新镜像

在新窗口中查看当前运行的容器，如图 3-16 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
412b30588f4a	ubuntu	"/bin/bash"	15 minutes ago	Up 15 minutes		silly_goldberg

图 3-16

silly_goldberg 是 Docker 为我们的容器随机分配的名字。

执行 docker commit 命令将容器保存为镜像，如图 3-17 所示。

```
root@ubuntu:~# docker commit silly_goldberg ubuntu-with-vi
sha256:3eb16ee8594bc653aff10181011747df99192a6966ece1f224e15536c06cadd6
root@ubuntu:~#
```

图 3-17

新镜像命名为 ubuntu-with-vi。

查看新镜像的属性，如图 3-18 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu-with-vi	latest	3eb16ee8594b	7 minutes ago	205.1 MB
ubuntu	latest	f753707788c5	5 weeks ago	127.2 MB

图 3-18

从 size 上看到镜像因为安装了软件而变大了。

从新镜像启动容器，验证 vi 已经可以使用，如图 3-19 所示。

```
root@ubuntu:~# docker run -it ubuntu-with-vi
root@f78bd38f9220:#
root@f78bd38f9220:/# which vi
/usr/bin/vi
```

图 3-19

以上演示了如何用 docker commit 创建新镜像。然而，Docker 并不建议用户通过这种方式构建镜像。原因如下：

(1) 这是一种手工创建镜像的方式，容易出错，效率低且可重复性弱。比如要在 debian

base 镜像中也加入 vi，还得重复前面的所有步骤。

(2) 更重要的：使用者并不知道镜像是如何创建出来的，里面是否有恶意程序。也就是说无法对镜像进行审计，存在安全隐患。

既然 docker commit 不是推荐的方法，我们为什么还要花时间学习呢？

原因是：即便是用 Dockerfile（推荐方法）构建镜像，底层也是 docker commit 一层一层构建新镜像的。学习 docker commit 能够帮助我们更加深入地理解构建过程和镜像的分层结构。

3.2.2 Dockerfile

Dockerfile 是一个文本文件，记录了镜像构建的所有步骤。

1. 第一个 Dockerfile

用 Dockerfile 创建上节的 ubuntu-with-vi，其内容如图 3-20 所示。

```
FROM ubuntu
RUN apt-get update && apt-get install -y vim
```

图 3-20

下面我们运行 docker build 命令构建镜像并详细分析每个细节。

```
root@ubuntu:~# pwd ①
/root
root@ubuntu:~# ls ②
Dockerfile
root@ubuntu:~# docker build -t ubuntu-with-vi-dockerfile . ③
Sending build context to Docker daemon 32.26 kB ④
Step 1 : FROM ubuntu ⑤
--> f753707788c5
Step 2 : RUN apt-get update && apt-get install -y vim ⑥
--> Running in 9f4d4166f7e3 ⑦
.....
Setting up vim (2:7.4.1.1689-3ubuntu1.1) ...
--> 35ca89798937 ⑧
Removing intermediate container 9f4d4166f7e3 ⑨
Successfully built 35ca89798937 ⑩
root@ubuntu:~#
```

① 当前目录为 /root。

② Dockerfile 准备就绪。

③ 运行 docker build 命令，-t 将新镜像命名为 ubuntu-with-vi-dockerfile，命令末尾的 . 指

明 build context 为当前目录。Docker 默认会从 build context 中查找 Dockerfile 文件，我们也可以通过 -f 参数指定 Dockerfile 的位置。

④ 从这步开始就是镜像真正的构建过程。首先 Docker 将 build context 中的所有文件发送给 Docker daemon。build context 为镜像构建提供所需要的文件或目录。

Dockerfile 中的 ADD、COPY 等命令可以将 build context 中的文件添加到镜像。此例中，build context 为当前目录 /root，该目录下的所有文件和子目录都会被发送给 Docker daemon。

所以，使用 build context 就得小心了，不要将多余文件放到 build context，特别不要把 /、/usr 作为 build context，否则构建过程会相当缓慢甚至失败。

⑤ Step 1：执行 FROM，将 Ubuntu 作为 base 镜像。

Ubuntu 镜像 ID 为 f753707788c5。

⑥ Step 2：执行 RUN，安装 vim，具体步骤为 ⑦ ⑧ ⑨。

⑦ 启动 ID 为 9f4d4166f7e3 的临时容器，在容器中通过 apt-get 安装 vim。

⑧ 安装成功后，将容器保存为镜像，其 ID 为 35ca89798937。

这一步底层使用的是类似 docker commit 的命令。

⑨ 删除临时容器 9f4d4166f7e3。

⑩ 镜像构建成功。

通过 docker images 查看镜像信息，如图 3-21 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu-with-vi-dockerfile	latest	35ca89798937	24 minutes ago	224.2 MB
ubuntu-with-vi	latest	3eb16ee8594b	5 hours ago	205.1 MB
ubuntu	latest	f753707788c5	5 weeks ago	127.2 MB

图 3-21

镜像 ID 为 35ca89798937，与构建时的输出一致。

在上面的构建过程中，我们要特别注意指令 RUN 的执行过程 ⑦ ⑧ ⑨。Docker 会在启动的临时容器中执行操作，并通过 commit 保存为新的镜像。

2. 查看镜像分层结构

ubuntu-with-vi-dockerfile 是通过在 base 镜像的顶部添加一个新的镜像层而得到的，如图 3-22 所示。



图 3-22

这个新镜像层的内容由 `RUN apt-get update && apt-get install -y vim` 生成。这一点我们可以通过 `docker history` 命令验证，如图 3-23 所示。

IMAGE	CREATED	CREATED BY	SIZE
f753707788c5	5 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0 B
<missing>	5 weeks ago	/bin/sh -c mkdir -p /run/systemd && echo 'doc'	7 B
<missing>	5 weeks ago	/bin/sh -c sed -i 's/\#\!/\#>/g' \$(deb-*universe*)\$/*	1.895 kB
<missing>	5 weeks ago	/bin/sh -c rm -rf /var/lib/apt/lists/*	0 B
<missing>	5 weeks ago	/bin/sh -c set -xe && echo '#!/bin/sh' > /u	745 B
<missing>	5 weeks ago	/bin/sh -c #(nop) ADD file:b1cd0e54ba28cb1d6d	127.2 MB
root@ubuntu:~#			
root@ubuntu:~# docker history ubuntu-with-vi-dockerfile			
35ca89798937	48 minutes ago	/bin/sh -c apt-get update && apt-get install	97.07 MB
f753707788c5	5 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0 B
<missing>	5 weeks ago	/bin/sh -c mkdir -p /run/systemd && echo 'doc'	7 B
<missing>	5 weeks ago	/bin/sh -c sed -i 's/\#\!/\#>/g' \$(deb-*universe*)\$/*	1.895 kB
<missing>	5 weeks ago	/bin/sh -c rm -rf /var/lib/apt/lists/*	0 B
<missing>	5 weeks ago	/bin/sh -c set -xe && echo '#!/bin/sh' > /u	745 B
<missing>	5 weeks ago	/bin/sh -c #(nop) ADD file:b1cd0e54ba28cb1d6d	127.2 MB
root@ubuntu:~#			

图 3-23

`docker history` 会显示镜像的构建历史，也就是 Dockerfile 的执行过程。

`ubuntu-with-vi-dockerfile` 与 Ubuntu 镜像相比，确实只是多了顶部的一层 `35ca89798937`，由 `apt-get` 命令创建，大小为 97.07MB。`docker history` 也向我们展示了镜像的分层结构，每一层由上至下排列。

注：missing 表示无法获取 IMAGE ID，通常从 Docker Hub 下载的镜像会有这个问题。

3. 镜像的缓存特性

我们接下来看 Docker 镜像的缓存特性。

Docker 会缓存已有镜像的镜像层，构建新镜像时，如果某镜像层已经存在，就直接使用，无须重新创建。

下面举例说明。

在前面的 Dockerfile 中添加一点新内容，往镜像中复制一个文件，如图 3-24 所示。

```
FROM ubuntu
RUN apt-get update && apt-get install -y vim
COPY testfile /
```

图 3-24

```

root@ubuntu:~# ls ①
Dockerfile testfile
root@ubuntu:~#
root@ubuntu:~# docker build -t ubuntu-with-vi-dockerfile-2 .
Sending build context to Docker daemon 32.77 kB
Step 1 : FROM ubuntu
--> f753707788c5
Step 2 : RUN apt-get update && apt-get install -y vim
--> Using cache ②
--> 35ca89798937
Step 3 : COPY testfile / ③
--> 8d02784a78f4 Removing intermediate container bf2b4040f4e9
Successfully built 8d02784a78f4

```

- ① 确保 testfile 已存在。
- ② 重点在这里：之前已经运行过相同的 RUN 指令，这次直接使用缓存中的镜像层 35ca89798937。
- ③ 执行 COPY 指令。

其过程是启动临时容器，复制 testfile，提交新的镜像层 8d02784a78f4，删除临时容器。

在 ubuntu-with-vi-dockerfile 镜像上直接添加一层就得到了新的镜像 ubuntu-with-vi-dockerfile-2，如图 3-25 所示。



图 3-25

如果我们希望在构建镜像时不使用缓存，可以在 docker build 命令中加上--no-cache 参数。

Dockerfile 中每一个指令都会创建一个镜像层，上层是依赖于下层的。无论什么时候，只要某一层发生变化，其上面所有层的缓存都会失效。

也就是说，如果我们改变 Dockerfile 指令的执行顺序，或者修改或添加指令，都会使缓存失效。举例说明，比如交换前面 RUN 和 COPY 的顺序，如图 3-26 所示。

```

FROM ubuntu
COPY testfile /
RUN apt-get update && apt-get install -y vim

```

图 3-26

虽然在逻辑上这种改动对镜像的内容没有影响，但由于分层的结构特性，Docker 必须重建受影响的镜像层。

```
root@ubuntu:~# docker build -t ubuntu-with-vi-dockerfile-3 . Sending
build context to Docker daemon 37.89 kB Step 1 : FROM ubuntu -->
f753707788c5 Step 2 : COPY testfile / --> bc87c9710f40 Removing
intermediate container 04ff324d6af5 Step 3 : RUN apt-get update && apt-
get install -y vim --> Running in 7f0fcbb5ee373 Get:1
http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB] .....
```

从上面的输出可以看到生成了新的镜像层 bc87c9710f40，缓存已经失效。

除了构建时使用缓存，Docker 在下载镜像时也会使用。例如我们下载 httpd 镜像，如图 3-27 所示。

```
root@ubuntu:~# docker pull httpd
Using default tag: latest
latest: Pulling from library/httpd
386a0066cd84a: Already exists
a1d6b8e2fac: Pull complete
c1fdc7bee37: Pull complete
bd1a67deca2: Pull complete
92b34ad02810: Pull complete
Digest: sha256:5b4a3c85b4b874e84174ee7e78a59920818aa39903f6a28a47b9278f576b4a4d
Status: Downloaded newer image for httpd:latest
```

图 3-27

docker pull 命令输出显示第一层（base 镜像）已经存在，不需要下载。

由 Dockerfile 可知 httpd 的 base 镜像为 debian，正好之前已经下载过 debian 镜像，所以有缓存可用。通过 docker history 可以进一步验证，如图 3-28 所示。

IMAGE	CREATED	CREATED BY	SIZE
50f10ef90911	12 days ago	/bin/sh -c #(nop) CMD ["httpd-foreground"]	0 B
<missing>	12 days ago	/bin/sh -c #(nop) EXPOSE 80/tcp	0 B
<missing>	12 days ago	/bin/sh -c #(nop) COPY file:761e313354b918b6c	133 B
<missing>	12 days ago	/bin/sh -c set -x && buildDeps -' bzip2 c	29.17 MB
<missing>	12 days ago	/bin/sh -c #(nop) ENV HTTPD_ASC_URL=https://	0 B
<missing>	12 days ago	/bin/sh -c #(nop) ENV HTTPD_BZ2_URL=https://	0 B
<missing>	12 days ago	/bin/sh -c #(nop) ENV HTTPD_SHA1=5101be340c4	0 B
<missing>	12 days ago	/bin/sh -c #(nop) ENV HTTPD_VERSION=2.4.23	0 B
<missing>	12 days ago	/bin/sh -c apt-get update && apt-get install	41.15 MB
<missing>	12 days ago	/bin/sh -c #(nop) WORKDIR /usr/local/apache2	0 B
<missing>	12 days ago	/bin/sh -c mkdir -p "SHUTDOWN_SCRIPTS" && chown	0 B
<missing>	12 days ago	/bin/sh -c #(nop) ENV PATH=/usr/local/apache	0 B
<missing>	12 days ago	/bin/sh -c #(nop) ENV HTTPD_PREFIX=/usr/loca	0 B
<missing>	13 days ago	/bin/sh -c #(nop) CMD ["./bin/bash"]	0 B
<missing>	13 days ago	/bin/sh -c #(nop) ADD file:41ea5187c50116884c	123 MB

IMAGE	CREATED	CREATED BY	SIZE
73e72bf822ca	13 days ago	/bin/sh -c #(nop) CMD ["./bin/bash"]	0 B
<missing>	13 days ago	/bin/sh -c #(nop) ADD file:41ea5187c50116884c	123 MB

图 3-28

4. 调试 Dockerfile

总结一下通过 Dockerfile 构建镜像的过程：

- (1) 从 base 镜像运行一个容器。
- (2) 执行一条指令，对容器做修改。
- (3) 执行类似 docker commit 的操作，生成一个新的镜像层。
- (4) Docker 再基于刚刚提交的镜像运行一个新容器。
- (5) 重复 2~4 步，直到 Dockerfile 中的所有指令执行完毕。

从这个过程可以看出，如果 Dockerfile 由于某种原因执行到某个指令失败了，我们也将能够得到前一个指令成功执行构建出的镜像，这对调试 Dockerfile 非常有帮助。我们可以运行最新的这个镜像定位指令失败的原因。

我们来看一个调试的例子。Dockerfile 内容如图 3-29 所示。

```
FROM busybox
RUN touch tempfile
RUN /bin/bash -c echo "continue to build..."
COPY testfile /
```

图 3-29

执行 docker build，如图 3-30 所示。

```
root@ubuntu:~# docker build -t image-debug .
Sending build context to Docker daemon 32.77 kB
Step 1 : FROM busybox
latest: Pulling from library/busybox
56bec22e3559: Pull complete
Digest: sha256:29f5d56d12684887bdfa50cd29fc31eea4aaaf4ad3bec43daf19026a7ce69912
Status: Downloaded newer image for busybox:latest
--> e02e811dd08f
Step 2 : RUN touch tempfile
--> Running in af387795fb6f
--> [22d31cc52b3e]
Removing intermediate container af387795fb6f
Step 3 : RUN /bin/bash -c echo "continue to build..."
--> Running in 66af52fb0773
/bin/sh: /bin/bash: not found
The command '/bin/sh -c /bin/bash -c echo "continue to build..."' returned a non-zero code: 127
root@ubuntu:~#
```

图 3-30

Dockerfile 在执行第三步 RUN 指令时失败。我们可以利用第二步创建的镜像 22d31cc52b3e 进行调试，方法是通过 docker run -it 启动镜像的一个容器，如图 3-31 所示。

```
root@ubuntu:~# docker run -it 22d31cc52b3e
/ #
/ # /bin/bash -c echo "continue to build..."
sh: /bin/bash: not found
/ #
```

图 3-31

手工执行 RUN 指令很容易定位失败的原因是 busybox 镜像中没有 bash。虽然这是个极其简单的例子，但它很好地展示了调试 Dockerfile 的方法。

5. Dockerfile 常用指令

是时候系统学习 Dockerfile 了。

下面列出了 Dockerfile 中最常用的指令，完整列表和说明可参看官方文档。

- FROM

指定 base 镜像。

- MAINTAINER

设置镜像的作者，可以是任意字符串。

- COPY

将文件从 build context 复制到镜像。

COPY 支持两种形式：COPY src dest 与 COPY ["src", "dest"]。

注意：src 只能指定 build context 中的文件或目录。

- ADD

与 COPY 类似，从 build context 复制文件到镜像。不同的是，如果 src 是归档文件（tar、zip、tgz、xz 等），文件会被自动解压到 dest。

- ENV

设置环境变量，环境变量可被后面的指令使用。例如：

```
ENV MY_VERSION 1.3 RUN apt-get install -y mypackage=$MY_VERSION
```

- EXPOSE

指定容器中的进程会监听某个端口，Docker 可以将该端口暴露出来。我们会在容器网络部分详细讨论。

- VOLUME

将文件或目录声明为 volume。我们会在容器存储部分详细讨论。

- WORKDIR

为后面的 RUN、CMD、ENTRYPOINT、ADD 或 COPY 指令设置镜像中的当前工作目录。

- RUN

在容器中运行指定的命令。

- CMD

容器启动时运行指定的命令。

Dockerfile 中可以有多个 CMD 指令，但只有最后一个生效。CMD 可以被 docker run 之后的参数替换。

● ENTRYPOINT

设置容器启动时运行的命令。

Dockerfile 中可以有多个 ENTRYPOINT 指令，但只有最后一个生效。CMD 或 docker run 之后的参数会被当作参数传递给 ENTRYPOINT。

下面我们来看一个较为全面的 Dockerfile，如图 3-32 所示。

```
# my dockerfile
FROM busybox
MAINTAINER cloudman@example.net
WORKDIR /testdir
RUN touch tmpfile1
COPY ["tmpfile2", "-"]
ADD ["bunch.tar.gz", "-"]
ENV WELCOME "You are in my container, welcome!"
```

图 3-32

注：Dockerfile 支持以“#”开头的注释。

构建镜像，如图 3-33 所示。

```
root@ubuntu:~# ls
Dockerfile bunch.tar.gz tmpfile2      ①
root@ubuntu:#
root@ubuntu:# docker build -t my-image .
Sending build context to Docker daemon 36.86 kB      ②
Step 1 : FROM busybox
--> e02e811dd08f
Step 2 : MAINTAINER cloudman@example.net
--> Running in 5a854581183f
--> 84dc36fc869f
Removing intermediate container 5a854581183f
Step 3 : WORKDIR /testdir
--> Running in 0e3d9567e16f
--> f1d56e8a580e
Removing intermediate container 0e3d9567e16f
Step 4 : RUN touch tmpfile1
--> Running in 333a00ba5d76
--> b3e80e7bc801
Removing intermediate container 333a00ba5d76
Step 5 : COPY tmpfile2 .
--> 11e1cd6d7722
Removing intermediate container 951384ae86e1
Step 6 : ADD bunch.tar.gz .
--> 4c6ec62b57d3
Removing intermediate container 1f4b5a3bb368
Step 7 : ENV WELCOME "You are in my container, welcome!"
--> Running in 859af1ac1e34
--> 23a56f08f5e7
Removing intermediate container 859af1ac1e34
Successfully built 23a56f08f5e7
root@ubuntu:~#
```

图 3-33

- ① 构建前确保 build context 中存在需要的文件。
- ② 依次执行 Dockerfile 指令，完成构建。

运行容器，验证镜像内容，如图 3-34 所示。

```
root@ubuntu:~# docker run -it my-image
/testdir # ①
/testdir # ls
bunch  tmpfile1  tmpfile2  ②
/testdir #
/testdir # echo $WELCOME  ③
You are in my container, welcome!
/testdir #
```

图 3-34

① 进入容器，当前目录即为 WORKDIR。

如果 WORKDIR 不存在，Docker 会自动为我们创建。

② WORKDIR 中保存了我们希望的文件和目录：

目录 bunch：由 ADD 指令从 build context 复制的归档文件 bunch.tar.gz，已经自动解压。

文件 tmpfile1：由 RUN 指令创建。

文件 tmpfile2：由 COPY 指令从 build context 复制。

③ ENV 指令定义的环境变量已经生效。

3.3 RUN vs CMD vs ENTRYPOINT

RUN、CMD 和 ENTRYPOINT 这三个 Dockerfile 指令看上去很类似，很容易混淆。本节将通过实践详细讨论它们的区别。

简单地说：

- (1) RUN：执行命令并创建新的镜像层，RUN 经常用于安装软件包。
- (2) CMD：设置容器启动后默认执行的命令及其参数，但 CMD 能够被 docker run 后面跟的命令行参数替换。
- (3) ENTRYPOINT：配置容器启动时运行的命令。

下面我们详细分析。

3.3.1 Shell 和 Exec 格式

我们可用两种方式指定 RUN、CMD 和 ENTRYPOINT 要运行的命令：Shell 格式和 Exec 格式，二者在使用上有细微的区别。

Shell 格式:

```
<instruction> <command>
```

例如:

```
RUN apt-get install python3
CMD echo "Hello world"
ENTRYPOINT echo "Hello world"
```

当指令执行时, shell 格式底层会调用 /bin/sh -c [command]。例如下面的 Dockerfile 片段:

```
ENV name Cloud Man ENTRYPOINT echo "Hello, $name"
```

执行 docker run [image] 将输出:

```
Hello, Cloud Man
```

注意环境变量 name 已经被值 Cloud Man 替换。

下面来看 Exec 格式。

```
<instruction> ["executable", "param1", "param2", ...]
```

例如:

```
RUN ["apt-get", "install", "python3"]
CMD ["/bin/echo", "Hello world"]
ENTRYPOINT ["/bin/echo", "Hello world"]
```

当指令执行时, 会直接调用 [command], 不会被 shell 解析。

例如下面的 Dockerfile 片段:

```
ENV name Cloud Man ENTRYPOINT ["/bin/echo", "Hello, $name"]
```

运行容器将输出:

```
Hello, $name
```

注意: 环境变量 name 没有被替换。

如果希望使用环境变量, 做如下修改 Dockerfile:

```
ENV name Cloud Man ENTRYPOINT ["/bin/sh", "-c", "echo Hello, $name"]
```

运行容器将输出:

```
Hello, Cloud Man
```

CMD 和 ENTRYPOINT 推荐使用 Exec 格式，因为指令可读性更强，更容易理解。RUN 则两种格式都可以。

3.3.2 RUN

RUN 指令通常用于安装应用和软件包。

RUN 在当前镜像的顶部执行命令，并创建新的镜像层。Dockerfile 中常常包含多个 RUN 指令。

RUN 有两种格式：

- (1) Shell 格式：RUN
- (2) Exec 格式：RUN ["executable", "param1", "param2"]

下面是使用 RUN 安装多个包的例子：

```
RUN apt-get update && apt-get install -y bzr\cvs\git\mercurial\
subversion
```

注意：apt-get update 和 apt-get install 被放在一个 RUN 指令中执行，这样能够保证每次安装的是最新的包。如果 apt-get install 在单独的 RUN 中执行，则会使用 apt-get update 创建镜像层，而这一层可能是很久以前缓存的。

3.3.3 CMD

CMD 指令允许用户指定容器的默认执行的命令。

此命令会在容器启动且 docker run 没有指定其他命令时运行。

- 如果 docker run 指定了其他命令，CMD 指定的默认命令将被忽略。
- 如果 Dockerfile 中有多个 CMD 指令，只有最后一个 CMD 有效。

CMD 有三种格式：

- (1) Exec 格式：CMD ["executable", "param1", "param2"]

这是 CMD 的推荐格式。

(2) CMD ["param1", "param2"] 为 ENTRYPOINT 提供额外的参数，此时 ENTRYPOINT 必须使用 Exec 格式。

- (3) Shell 格式：CMD command param1 param2

Exec 和 Shell 格式前面已经介绍过了。

第二种格式 CMD ["param1", "param2"] 要与 Exec 格式的 ENTRYPOINT 指令配合使用，其用途是为 ENTRYPOINT 设置默认的参数。我们将在后面讨论 ENTRYPOINT 时举例说明。

下面看看 CMD 是如何工作的。Dockerfile 片段如下：

```
CMD echo "Hello world"
```

运行容器 `docker run -it [image]` 将输出：

```
Hello world
```

但当后面加上一个命令，比如 `docker run -it [image] /bin/bash`，CMD 会被忽略掉，命令 `bash` 将被执行：

```
root@10a32dc7d3d3:/#
```

3.3.4 ENTRYPPOINT

`ENTRYPOINT` 指令可让容器以应用程序或者服务的形式运行。

`ENTRYPOINT` 看上去与 `CMD` 很像，它们都可以指定要执行的命令及其参数。不同的地方在于 `ENTRYPOINT` 不会被忽略，一定会被执行，即使运行 `docker run` 时指定了其他命令。

`ENTRYPOINT` 有两种格式：

(1) Exec 格式：`ENTRYPOINT ["executable", "param1", "param2"]` 这是 `ENTRYPOINT` 的推荐格式。

(2) Shell 格式：`ENTRYPOINT command param1 param2`。

在为 `ENTRYPOINT` 选择格式时必须小心，因为这两种格式的效果差别很大。

1. Exec 格式

`ENTRYPOINT` 的 Exec 格式用于设置要执行的命令及其参数，同时可通过 `CMD` 提供额外的参数。

`ENTRYPOINT` 中的参数始终会被使用，而 `CMD` 的额外参数可以在容器启动时动态替换掉。

比如下面的 Dockerfile 片段：

```
ENTRYPOINT ["/bin/echo", "Hello"] CMD ["world"]
```

当容器通过 `docker run -it [image]` 启动时，输出为：

```
Hello world
```

而如果通过 `docker run -it [image] CloudMan` 启动，则输出为：

```
Hello CloudMan
```

2. Shell 格式

ENTRYPOINT 的 Shell 格式会忽略任何 CMD 或 docker run 提供的参数。

3.3.5 最佳实践

- (1) 使用 RUN 指令安装应用和软件包，构建镜像。
- (2) 如果 Docker 镜像的用途是运行应用程序或服务，比如运行一个 MySQL，应该优先使用 Exec 格式的 ENTRYPOINT 指令。CMD 可为 ENTRYPOINT 提供额外的默认参数，同时可利用 docker run 命令行替换默认参数。
- (3) 如果想为容器设置默认的启动命令，可使用 CMD 指令。用户可在 docker run 命令行中替换此默认命令。

3.4 分发镜像

我们已经学会构建自己的镜像了。接下来的问题是如何在多个 Docker Host 上使用镜像。这里有几种可用的方法：

- (1) 用相同的 Dockerfile 在其他 host 构建镜像。
- (2) 将镜像上传到公共 Registry（比如 Docker Hub），Host 直接下载使用。
- (3) 搭建私有的 Registry 供本地 Host 使用。

第一种方法没什么特别的，前面已经讨论很多了。本节重点讨论如何使用公共和私有 Registry 分发镜像。

3.4.1 为镜像命名

无论采用何种方式保存和分发镜像，首先都得给镜像命名。

当我们执行 docker build 命令时已经为镜像取了个名字，例如前面：

```
docker build -t ubuntu-with-vi
```

这里的 ubuntu-with-vi 就是镜像的名字。通过 dock images 可以查看镜像的信息，如图 3-35 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu-with-vi	latest	3eb16ee8594b	47 hours ago	205.1 MB

图 3-35

这里注意到 ubuntu-with-vi 对应的是 REPOSITORY，而且还有一个叫 latest 的 TAG。

实际上一个特定镜像的名字由两部分组成：repository 和 tag。

```
[image name] = [repository]:[tag]
```

如果执行 docker build 时没有指定 tag，会使用默认值 latest。其效果相当于：

```
docker build -t ubuntu-with-vi:latest
```

tag 常用于描述镜像的版本信息，比如 httpd 镜像，如图 3-36 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
httpd	2.2	62ef267d1069	2 weeks ago	183.7 MB
httpd	2.4	50f10ef90911	2 weeks ago	193.3 MB
httpd	latest	50f10ef90911	2 weeks ago	193.3 MB

图 3-36

当然 tag 可以是任意字符串，比如 ubuntu 镜像如图 3-37 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	xenial	e4415b714b62	6 days ago	128.1 MB
ubuntu	trusty	4d44acee901c	6 days ago	187.9 MB
ubuntu	latest	f753707788c5	5 weeks ago	127.2 MB

图 3-37

1. 小心 latest tag

千万别被 latest tag 给误导了。latest 其实并没有什么特殊的含义。当没指明镜像 tag 时，Docker 会使用默认值 latest，仅此而已。

虽然 Docker Hub 上很多 repository 将 latest 作为最新稳定版本的别名，但这只是一种约定，而不是强制规定。

所以我们在使用镜像时最好还是避免使用 latest，明确指定某个 tag，比如 httpd:2.3，ubuntu:xenial。

2. tag 使用最佳实践

借鉴软件版本命名方式能够让用户很好地使用镜像。

一个高效的版本命名方案可以让用户清楚地知道当前使用的是哪个镜像，同时还可以保持足够的灵活性。

每个 repository 可以有多个 tag，而多个 tag 可能对应的是同一个镜像。下面通过例子为大家介绍 Docker 社区普遍使用的 tag 方案。

假设我们现在发布了一个镜像 myimage，版本为 v1.9.1，那么我们可以给镜像打上 4 个 tag：1.9.1、1.9、1 和 latest，如图 3-38 所示。

我们可以通过 docker tag 命令方便地给镜像打 tag。

```
docker tag myimage-v1.9.1 myimage:1
myimage:1.9
docker tag myimage-v1.9.1 myimage:1.9.1
docker tag myimage-v1.9.1 myimage:latest
```

过了一段时间，我们发布了 v1.9.2。这时可以打上 1.9.2 的 tag，并将 1.9、1 和 latest 从 v1.9.1 移到 v1.9.2，如图 3-39 所示。



图 3-38



图 3-39

命令为：

```
docker tag myimage-v1.9.2 myimage:1
myimage:1.9
docker tag myimage-v1.9.2 myimage:1.9.2
docker tag myimage-v1.9.2 myimage:latest
```

之后，v2.0.0 发布了。这时可以打上 2.0.0、2.0 和 2 的 tag，并将 latest 移到 v2.0.0，如图 3-40 所示。

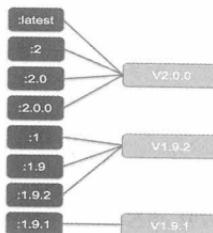


图 3-40

命令为：

```
docker tag myimage-v2.0.0 myimage:2
myimage:2.0
docker tag myimage-v2.0.0 myimage:2.0.0
docker tag myimage-v2.0.0 myimage:latest
```

这种 tag 方案使镜像的版本很直观，用户在选择时非常灵活：

- (1) myimage:1 始终指向 1 这个分支中最新的镜像。
- (2) myimage:1.9 始终指向 1.9.x 中最新的镜像。
- (3) myimage:latest 始终指向所有版本中最新的镜像。
- (4) 如果想使用特定版本，可以选择 myimage:1.9.1、myimage:1.9.2 或 myimage:2.0.0。

Docker Hub 上很多 repository 都采用这种方案，所以大家一定要熟悉。

3.4.2 使用公共 Registry

保存和分发镜像的最直接方法就是使用 Docker Hub。

Docker Hub 是 Docker 公司维护的公共 Registry。用户可以将自己的镜像保存到 Docker Hub 免费的 repository 中。如果不希望别人访问自己的镜像，也可以购买私有 repository。

除了 Docker Hub，quay.io 是另一个公共 Registry，提供与 Docker Hub 类似的服务。

下面介绍如何用 Docker Hub 存取我们的镜像。

- (1) 首先得在 Docker Hub 上注册一个账号。
- (2) 在 Docker Host 上登录，如图 3-41 所示。

```
root@ubuntu:~# docker login -u cloudman6
Password:
Login Succeeded
root@ubuntu:~#
```

图 3-41

这里用的是我自己的账号，用户名为 cloudman6，输入密码后登录成功。

- (3) 修改镜像的 repository，使之与 Docker Hub 账号匹配。

Docker Hub 为了区分不同用户的同名镜像，镜像的 registry 中要包含用户名，完整格式为：[username]/xxx:tag。我们通过 docker tag 命令重命名镜像，如图 3-42 所示。

```
root@ubuntu:~# docker tag httpd cloudman6/httpd:v1
root@ubuntu:~#
root@ubuntu:~# docker images cloudman6/httpd
REPOSITORY          TAG           IMAGE ID            CREATED             SIZE
cloudman6/httpd     v1            50f10ef90911      2 weeks ago        193.3 MB
```

图 3-42

注：Docker 官方自己维护的镜像没有用户名，比如 httpd。

通过 docker push 将镜像上传到 Docker Hub，如图 3-43 所示。

```
root@ubuntu:~# docker push cloudman6/httpd:v1
The push refers to a repository [docker.io/cloudman6/httpd]
b0eadb5d635d: Layer already exists
c856fa00775de: Layer already exists
2b9fc9190df9: Layer already exists
6b50c55a105a: Layer already exists
fe4c16cbf7a4: Layer already exists
v1: digest: sha256:5b4a3c85b4b874e84174ee7e78a59920818aa39903f6a28a47b9278f576b4a4d size: 1366
root@ubuntu:~#
```

图 3-43

Docker 会上传镜像的每一层。因为 cloudman6/httpd:v1 这个镜像实际上跟官方的 httpd 镜像一模一样，Docker Hub 上已经有了全部的镜像层，所以真正上传的数据很少。同样的，如果我们的镜像是基于 base 镜像的，也只有新增加的镜像层会被上传。如果想上传同一 repository 中所有镜像，省略 tag 部分就可以了，例如：

```
docker push cloudman6/httpd
```

(1) 登录 <https://hub.docker.com>，在 Public Repository 中就可以看到上传的镜像，如图 3-44 所示。

Tag Name	Compressed Size	Last Updated
v1	71 MB	9 minutes ago

图 3-44

如果要删除上传的镜像，只能在 Docker Hub 界面上操作。

(2) 这个镜像可被其他 Docker host 下载使用了，如图 3-45 所示。

```
root@ubuntu:~# docker pull cloudman6/httpd:v1
v1: Pulling from cloudman6/httpd
386a066cd84a: Already exists
a11d6b8e2fac: Pull complete
c1fdc7beec37: Pull complete
bd14a67deca2: Pull complete
92b34ad02810: Pull complete
Digest: sha256:5b4a3c85b4b874e84174ee7e78a59920818aa399
Status: Downloaded newer image for cloudman6/httpd:v1
root@ubuntu:~#
```

图 3-45

3.4.3 搭建本地 Registry

Docker Hub 虽然非常方便，但还是有些限制，比如：

- (1) 需要 Internet 连接，而且下载和上传速度慢。
- (2) 上传到 Docker Hub 的镜像任何人都能够访问，虽然可以用私有 repository，但不是免费的。
- (3) 因安全原因很多组织不允许将镜像放到外网。

解决方案就是搭建本地的 Registry。

Docker 已经将 Registry 开源了，同时在 Docker Hub 上也有官方的镜像 registry。下面我们在 Docker 中运行自己的 registry。

1. 启动 registry 容器

我们使用的镜像是 registry:2，如图 3-46 所示。

```
root@ubuntu:~# docker run -d -p 5000:5000 -v /myregistry:/var/lib/registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
3690ec4760f9: Pull complete
930045f1e8fb: Pull complete
feedad0cbdbc: Pull complete
61f85310d350: Pull complete
b6082c239858: Pull complete
Digest: sha256:1152291c7f93a4ea2ddc95e46d142c31e743b6dd70e194af9e6ebe530f782c17
Status: Downloaded newer image for registry:2
e1c289488c5c7928ef732b349df6bc51b00890f9fe696e6553e0751f90d62437
root@ubuntu:~#
```

图 3-46

- -d：后台启动容器。
- -p：将容器的 5000 端口映射到 Host 的 5000 端口。5000 是 registry 服务端口。端口映射我们会在容器网络章节详细讨论。
- -v：将容器 /var/lib/registry 目录映射到 Host 的 /myregistry，用于存放镜像数据。-v 的使用我们会在容器存储章节详细讨论。

通过 docker tag 重命名镜像，使之与 registry 匹配，如图 3-47 所示。

```
root@ubuntu:~# docker tag cloudman6/httpd:v1 registry.example.net:5000/cloudman6/httpd:v1
root@ubuntu:~#
```

图 3-47

我们在镜像的前面加上了运行 registry 的主机名称和端口。

前面已经讨论了镜像名称由 repository 和 tag 两部分组成。而 repository 的完整格式为：

[registry-host]:[port]/[username]/xxx

只有 Docker Hub 上的镜像可以省略 registry-host:[port]。

2. 通过 docker push 上传镜像

通过 docker push 上传镜像如图 3-48 所示。

```
root@ubuntu:~# docker push registry.example.net:5000/cloudman6/httpd:v1
The push refers to a repository [registry.example.net:5000/cloudman6/httpd]
b0eab5d635d7: Pushed
c856fd075de: Pushed
2b9fc9190df9: Pushed
6b50c55a105a: Pushed
fe4c16cbf7a4: Pushed
v1: digest: sha256:5b4a3c85b4b874e84174ee7e78a59920818aa39903f6a28a47b9278f576b4a4d size: 1366
root@ubuntu:~#
```

图 3-48

现在已经可通过 docker pull 从本地 registry 下载镜像了，如图 3-49 所示。

```
root@ubuntu:~# docker pull registry.example.net:5000/cloudman6/httpd:v1
v1: Pulling from cloudman6/httpd
386a066cd84a: Already exists
a11d6b8e2fac: Pull complete
c1fdc7beec37: Pull complete
bd14a67deca2: Pull complete
92b34ad02810: Pull complete
Digest: sha256:5b4a3c85b4b874e84174ee7e78a59920818aa39903f6a28a47b9278f576b4a4d
Status: Downloaded newer image for registry.example.net:5000/cloudman6/httpd:v1
root@ubuntu:~#
root@ubuntu:~# docker images registry.example.net:5000/cloudman6/httpd
REPOSITORY          TAG           IMAGE ID
registry.example.net:5000/cloudman6/httpd   v1            50f10ef90911
root@ubuntu:~#
```

图 3-49

除了镜像的名称长一些（包含 registry host 和 port），使用方式完全一样。

以上是搭建本地 registry 的简要步骤。当然 registry 也支持认证，https 安全传输等特性，具体可以参考官方文档

<https://docs.docker.com/registry/configuration/>

3.5 小结

本章我们学习了 Docker 镜像。首先讨论了镜像的分层结构，然后学习了如何构建镜像，最后实践使用 Docker Hub 和本地 registry。

下面是镜像的常用操作子命令：

- images：显示镜像列表。
- history：显示镜像构建历史。

- commit: 从容器创建新镜像。
- build: 从 Dockerfile 构建镜像。
- tag: 给镜像打 tag。
- pull: 从 registry 下载镜像。
- push: 将镜像上传到 registry。
- rmi: 删除 Docker host 中的镜像。
- search: 搜索 Docker Hub 中的镜像。

除了 rmi 和 search, 其他命令都已经用过了。

1. rmi

rmi 只能删除 host 上的镜像, 不会删除 registry 的镜像。

如果一个镜像对应了多个 tag, 只有当最后一个 tag 被删除时, 镜像才被真正删除。例如 host 中 debian 镜像有两个 tag, 如图 3-50 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
debian	jessie	73e72bf822ca	2 weeks ago	123 MB
debian	latest	73e72bf822ca	2 weeks ago	123 MB

图 3-50

删除其中 debian:latest 只是删除了 latest tag, 镜像本身没有删除, 如图 3-51 所示。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
debian	jessie	73e72bf822ca	2 weeks ago	123 MB
debian	latest			

图 3-51

只有当 debian:jessie 也被删除时, 整个镜像才会被删除, 如图 3-52 所示。

root@ubuntu:~# docker rmi debian:jessie
Untagged: debian:jessie
Untagged: debian@sha256:c1ce85a0f7126a3b5cbf7c57676b01b37c755b9ff9e2f39ca88181c02b985724
Deleted: sha256:73e72bf822ca801578cd888d87da21811687e6669c80232e96105598c3c4902d
Deleted: sha256:fe4c16cbf7a4c70a5462654cf2c8f9f69778db280f235229bd98cf8784e878e4
root@ubuntu:~#

图 3-52

2. search

search 让我们无须打开浏览器, 在命令行中就可以搜索 Docker Hub 中的镜像, 如图 3-53 所示。当然, 如果想知道镜像都有哪些 tag, 还是得访问 Docker Hub。

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
httpd	The Apache HTTP Server Project	790	[OK]	[OK]
centos/httpd		9		
lolhens/httpd	Apache httpd 2 Server	1	[OK]	[OK]
microwebapps/httpd-frontend	Httpd frontend allowing simple deployment ...	1	[OK]	[OK]
rgielen/httpd-image-php5	Docker image for Apache httpd with PHP 5 b...	1	[OK]	[OK]
interferex/httpd	Simple HTTPD Server + TC Rate Limiting	0	[OK]	[OK]
publicisworldwide/httpd	The Apache httpd webserver.	0	[OK]	[OK]
dionefc/httpd	httpd docker	0	[OK]	[OK]
efrecon/httpd	A micro-sized httpd. Start serving files i...	0	[OK]	[OK]
pubcli/httpd	httpd:latest	0	[OK]	[OK]
rgielen/httpd-image-simple	Docker image for simple Apache httpd based...	0	[OK]	[OK]
objectstyle/httpd	ObjectStyle HTTPD Image	0	[OK]	[OK]
steelorbis/httpd	local httpd	0	[OK]	[OK]
ibtech/httpd	Apache HTTPD padrão da IBTech	0	[OK]	[OK]
superkul/httpd	Centos httpd server	0	[OK]	[OK]

图 3-53

第 4 章

◀Docker 容器▶

上一章我们学习了如何构建 Docker 镜像，并通过镜像运行容器。本章将深入讨论容器：学习容器的各种操作、容器各种状态之间如何转换，以及实现容器的底层技术。

4.1 运行容器

docker run 是启动容器的方法。在讨论 Dockerfile 时我们已经学习到，可用三种方式指定容器启动时执行的命令：

- (1) CMD 指令。
- (2) ENTRYPPOINT 指令。
- (3) 在 docker run 命令行中指定。

例如下面的例子，如图 4-1 所示。

```
root@ubuntu:~# docker run ubuntu pwd  
/  
root@ubuntu:~#
```

图 4-1

容器启动时执行 pwd，返回的 / 是容器中的当前目录。执行 docker ps 或 docker container ls 可以查看 Docker host 中当前运行的容器，如图 4-2 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
--------------	-------	---------	---------	--------

图 4-2

咦，怎么没有容器？用 docker ps -a 或 docker container ls -a 看看，如图 4-3 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
dccacacd8f19	ubuntu	"pwd"	12 seconds ago	Exited (0) 11 seconds ago

图 4-3

-a 会显示所有状态的容器，可以看到，之前的容器已经退出了，状态为 Exited。

这种“一闪而过”的容器通常不是我们想要的结果，我们希望容器能够保持 running 状态，这样才能被我们使用。

4.1.1 让容器长期运行

如何让容器保存运行呢？

因为容器的生命周期依赖于启动时执行的命令，只要该命令不结束，容器也就不会退出。

理解了这个原理，我们就可以通过执行一个长期运行的命令来保持容器的运行状态。例如执行下面的命令，如图 4-4 所示。

```
root@ubuntu:~# docker run ubuntu /bin/bash -c "while true ; do sleep 1; done"
```

图 4-4

while 语句让 bash 不会退出。可以打开另一个终端查看容器的状态，如图 4-5 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
fe39cc22cc8b	ubuntu	/bin/bash -c 'while '	22 minutes ago	Up 22 minutes		agitated_bassl

图 4-5

可见容器仍处于运行状态。不过这种方法有个缺点：它占用了一个终端。

可以加上参数 -d 以后台方式启动容器，如图 4-6 所示。

```
root@ubuntu:~# docker run -d ubuntu /bin/bash -c "while true ; do sleep 1; done"
18cc3a46a6a9d50e05e835b77ab5ab5a003e472753a0c031505e7a89be3aa241
root@ubuntu:~#
```

图 4-6

容器启动后回到了 docker host 的终端。这里看到 docker 返回了一串字符，这是容器的 ID。通过 docker ps 查看容器，如图 4-7 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
18cc3a46a6a9d50e05e835b77ab5ab5a003e472753a0c031505e7a89be3aa241	ubuntu	/bin/bash -c 'while '	3 minutes ago	Up 3 minutes		hopeful_carson
fe39cc22cc8b	ubuntu	/bin/bash -c 'while '	26 minutes ago	Up 26 minutes		agitated_bassl

图 4-7

现在我们有了两个正在运行的容器。这里注意一下容器的 CONTAINER ID 和 NAMES 这两个字段。

CONTAINER ID 是容器的“短 ID”，前面启动容器时返回的是“长 ID”。短 ID 是长 ID 的前 12 个字符。

NAMES 字段显示容器的名字，在启动容器时可以通过 --name 参数显式地为容器命名，如果不指定，docker 会自动为容器分配名字。

对于容器的后续操作，我们需要通过“长 ID”“短 ID”或者“名称”来指定要操作的

容器。比如下面停止一个容器，如图 4-8 所示。

```
root@ubuntu:~# docker stop fe39cc2ccc8b
fe39cc2ccc8b
root@ubuntu:~#
```

图 4-8

这里我们就是通过“短 ID”指定了要停止的容器。

通过 while 启动的容器虽然能够保持运行，但实际上没有干什么有意义的事情。容器常见的用途是运行后台服务，例如前面我们已经看到的 http server，如图 4-9 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -name "my_http_server" -d httpd
3d6d292d1a0c1c04468d23d27e583bfcc05cad6309759f260e983059e73d
root@ubuntu:~#
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
3d6d292d1a0c        httpd              "httpd-foreground"
18cc304660a9        ubuntu              "/bin/bash -c 'while "
root@ubuntu:~#
```

图 4-9

这一次我们用 --name 指定了容器的名字。我们还看到容器运行的命令是 httpd-foreground，通过 docker history 可知这个命令是通过 CMD 指定的，如图 4-10 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker history httpd
IMAGE          CREATED          CREATED BY
50f10ef90911   4 weeks ago      /bin/sh -c #(nop)  CMD ["httpd-foreground"]
<missing>       4 weeks ago      /bin/sh -c #(nop) EXPOSE 80/tcp
<missing>       4 weeks ago      /bin/sh -c #(nop) COPY file:761e313354b918b6c  133 B
```

图 4-10

4.1.2 两种进入容器的方法

我们经常需要进到容器里去做一些工作，比如查看日志、调试、启动其他进程等。有两种方法进入容器：attach 和 exec。

1. docker attach

通过 docker attach 可以 attach 到容器启动命令的终端，例如如图 4-11 所示的例子。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d ubuntu /bin/bash -c "while true ; do sleep 1; echo I_am_in_container; done"
969fac2f0e41f05a904504e552274d04ad09c0ee5d9f8c3b2ab673e07e0d757f
root@ubuntu:~#
root@ubuntu:~# docker attach 969fac2f0e41f05a904504e552274d04ad09c0ee5d9f8c3b2ab673e07e0d757f
I_am_in_container
I_am_in_container
I_am_in_container
I_am_in_container
I_am_in_container
```

图 4-11

这次我们通过“长 ID”attach 到了容器的启动命令终端，之后看到的是 echo 每隔一秒

打印的信息。

注：可通过 Ctrl+p，然后 Ctrl+q 组合键退出 attach 终端。

2. docker exec

通过 docker exec 进入相同的容器，如图 4-12 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker exec -it 969fac2f0e41 bash ①
root@969fac2f0e41:/# ②
root@969fac2f0e41:/# ps -elf ③
F S UID PID PPID C PRI NI ADDR SZ WCHAN STIME TTY          TIME CMD
4 S root      1  0  80  0 - 4507 wait   08:10 ?          00:00:02 /bin/bash -c while true ; do sleep 1; echo I
4 S root      8705  0  80  0 - 4552 wait   10:35 ?          00:00:00 bash
0 S root      8729  1  0  80  0 - 1095 hrtime 10:35 ?          00:00:00 sleep 1
0 R root      8730  8705  0  80  0 - 8606 -          10:35 ?          00:00:00 ps -elf
root@969fac2f0e41:/#
root@969fac2f0e41:/#
root@969fac2f0e41:/# exit ④
exit
```

图 4-12

说明如下：

- ① -it 以交互模式打开 pseudo-TTY，执行 bash，其结果就是打开了一个 bash 终端。
- ② 进入到容器中，容器的 hostname 就是其“短 ID”。
- ③ 可以像在普通 Linux 中一样执行命令。ps -elf 显示了容器启动进程 while 以及当前的 bash 进程。
- ④ 执行 exit 退出容器，回到 docker host。

```
docker exec -it <container> bash|sh
```

这是执行 exec 最常用的方式。

3. attach VS exec

attach 与 exec 主要区别如下：

- (1) attach 直接进入容器启动命令的终端，不会启动新的进程。
- (2) exec 则是在容器中打开新的终端，并且可以启动新的进程。
- (3) 如果想直接在终端中查看启动命令的输出，用 attach；其他情况使用 exec。

当然，如果只是为了查看启动命令的输出，可以使用 docker logs 命令，如图 4-13 所示。

```
root@ubuntu:~# docker logs -f 969fac2f0e41
I_am_in_container
I_am_in_container
I_am_in_container
I_am_in_container
I_am_in_container
I_am_in_container
```

图 4-13

`-f` 的作用与 `tail -f` 类似，能够持续打印输出。

4.1.3 运行容器的最佳实践

按用途容器大致可分为两类：服务类容器和工具类的容器。

服务容器以 daemon 的形式运行，对外提供服务，比如 Web Server、数据库等。通过 -d 以后台方式启动这类容器是非常合适的。如果要排查问题，可以通过 exec -it 进入容器。

工具类容器通常能给我们提供一个临时的工作环境，通常以 run -it 方式运行，比如如图 4-14 所示的例子。

```
root@ubuntu:~# docker run -it busybox
/ #
/ # wget www.baidu.com
Connecting to www.baidu.com (58.217.200.37:80)
index.html      100% |*****
```

图 4-14

运行 busybox, run -it 的作用是在容器启动后就直接进入。我们这里通过 wget 验证了在容器中访问 internet 的能力。执行 exit 退出终端，同时容器停止。

工具类容器多使用基础镜像，例如 busybox、debian、ubuntu 等。

4.1.4 容器运行小结

容器运行相关的知识点：

- (1) 当 CMD、Entrypoint 和 docker run 命令行指定的命令运行结束时，容器停止。
 - (2) 通过 -d 参数在后台启动容器。
 - (3) 通过 exec -it 可进入容器并执行命令。

指定容器的三种方法：

- (1) 短 ID。
 - (2) 长 ID。
 - (3) 容器名称。 可通过 `--name` 为容器命名。重命名容器可执行 `docker rename`。

容器按用途可分为两类：

- (1) 服务类的容器。
 - (2) 工具类的容器。

4.2 stop/start/restart 容器

通过 docker stop 可以停止运行的容器，如图 4-15 所示。

```
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
969fac2f0e41        ubuntu              "/bin/bash -c 'while "
3d6d92d51a0          httpd
root@ubuntu:~#
root@ubuntu:~# docker stop suspicious_euler
suspicious_euler
root@ubuntu:~#
root@ubuntu:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
969fac2f0e41        ubuntu              "/bin/bash -c 'while "
3d6d92d51a0          httpd
root@ubuntu:~#
```

图 4-15

容器在 docker host 中实际上是一个进程，docker stop 命令本质上是向该进程发送一个 SIGTERM 信号。如果想快速停止容器，可使用 docker kill 命令，其作用是向容器进程发送 SIGKILL 信号，如图 4-16 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker kill my_http_server
my_http_server
root@ubuntu:~#
```

图 4-16

对于处于停止状态的容器，可以通过 docker start 重新启动，如图 4-17 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker start suspicious_euler
suspicious_euler
root@ubuntu:~#
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
969fac2f0e41        ubuntu              "/bin/bash -c 'while "
root@ubuntu:~#
```

图 4-17

docker start 会保留容器的第一次启动时的所有参数。

docker restart 可以重启容器，其作用就是依次执行 docker stop 和 docker start。

容器可能会因某种错误而停止运行。对于服务类容器，我们通常希望在这种情况下容器能够自动重启。启动容器时设置 --restart 就可以达到这个效果，如图 4-18 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d --restart=always httpd
0590d8168212da786ce6d084e5f16f3de73cb8aaf1bec1615c40ffb9b0511d6e
root@ubuntu:~#
```

图 4-18

--restart=always 意味着无论容器因何种原因退出（包括正常退出），都立即重启；该参数

的形式还可以是 `--restart=on-failure:3`, 意思是如果启动进程退出代码非 0, 则重启容器, 最多重启 3 次。

4.3 pause / unpause 容器

有时我们只是希望让容器暂停工作一段时间, 比如要对容器的文件系统打个快照, 或者 docker host 需要使用 CPU, 这时可以执行 docker pause, 如图 4-19 所示。

```
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
0590d8168212        httpd              "httpd-foreground"   27 minutes ago    Up 21 minutes      80/tcp
root@ubuntu:~#
root@ubuntu:~# docker pause tiny_kare
tiny_kare
root@ubuntu:~#
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
0590d8168212        httpd              "httpd-foreground"   27 minutes ago    Up 21 minutes (Paused) 80/tcp
root@ubuntu:~#
```

图 4-19

处于暂停状态的容器不会占用 CPU 资源, 直到通过 docker unpause 恢复运行, 如图 4-20 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker unpause tiny_kare
tiny_kare
root@ubuntu:~#
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
0590d8168212        httpd              "httpd-foreground"   31 minutes ago    Up 24 minutes      80/tcp
root@ubuntu:~#
```

图 4-20

4.4 删除容器

使用 docker 一段时间后, host 上可能会有大量已经退出了的容器, 如图 4-21 所示。

```
root@ubuntu:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
690ab06673f        busybox             "sh"               About a minute ago   Exited (0) About a minute ago
2z8251524fe5       busybox             "sh"               About a minute ago   Exited (0) About a minute ago
c43cf1f692b0       ubuntu              "/bin/bash"         About a minute ago   Exited (0) About a minute ago
177e7bd1b3b2       ubuntu              "/bin/bash"         About a minute ago   Exited (0) About a minute ago
038a2a14dd         ubuntu              "/bin/bash"         About a minute ago   Exited (0) About a minute ago
03bfc91f060        httpd              "httpd-foreground"  23 minutes ago     Exited (137) 8 minutes ago
0590d8168212        httpd              "httpd-foreground"  36 minutes ago     Up 29 minutes      80/tcp
069foc2f0e41       ubuntu              "/bin/bash -c 'while "  17 hours ago     Exited (137) 30 minutes ago
3dd6d292d1a0        httpd              "httpd-foreground"  18 hours ago      Exited (137) 57 minutes ago
root@ubuntu:~#
```

图 4-21

这些容器依然会占用 host 的文件系统资源，如果确认不会再重启此类容器，可以通过 docker rm 删 除，如图 4-22 所示。

```
root@ubuntu:~# docker rm 228251524fe5 177e7bd183b2
228251524fe5
177e7bd183b2
root@ubuntu:~#
```

图 4-22

docker rm 一次可以指定多个容器，如果希望批量删除所有已经退出的容器，可以执行如下命令，结果如图 4-23 所示。

```
docker rm -v $(docker ps -aq -f status=exited)
```

```
root@ubuntu:~#
root@ubuntu:~# docker rm -v $(docker ps -aq -f status=exited)
690dd006673f
cc43c4f6b29a
a38c610afc4d
d3bfe691f060
969fac2f0a41
3dd292db1a0
root@ubuntu:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS             NAMES
0590d8168212        httpd              "httpd-foreground"   43 minutes ago    Up 36 minutes     80/tcp            tiny_kare
root@ubuntu:~#
```

图 4-23

顺便说一句： docker rm 是删除容器，而 docker rmi 是删除镜像。

4.5 State Machine

前面我们已经讨论了容器的各种操作，对容器的生命周期有了大致的理解，下面这张状态机很好地总结了容器各种状态之间是如何转换的，如图 4-24 所示。

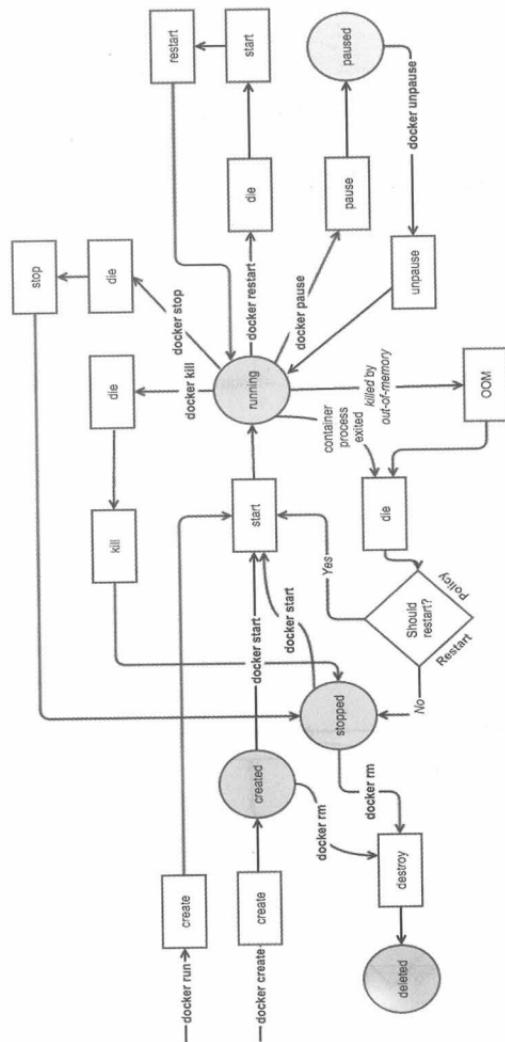


图 4-24

如果掌握了前面的知识，要看懂这张图应该不难。不过有两点还是需要补充一下：

- (1) 可以先创建容器，稍后再启动，如图 4-25 所示。

```

root@ubuntu:~# docker create httpd ①
989e12e4d8ea0833ffed05f56a7dd3e76a14c875c194191a05060026b5636c4b
root@ubuntu:#
root@ubuntu:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
989e12e4d8ea        httpd               "httpd-foreground"   4 seconds ago    Created
root@ubuntu:#
root@ubuntu:~# docker start 989e12e4d8ea ②
989e12e4d8ea
root@ubuntu:#
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
989e12e4d8ea        httpd               "httpd-foreground"   22 seconds ago   Up 5 seconds      80/tcp
root@ubuntu:#
root@ubuntu:~#

```

图 4-25

- ① docker create 创建的容器处于 Created 状态。
- ② docker start 将以后台方式启动容器。 docker run 命令实际上是 docker create 和 docker start 的组合。
- (2) 只有当容器的启动进程退出时，--restart 才生效，如图 4-26 所示。

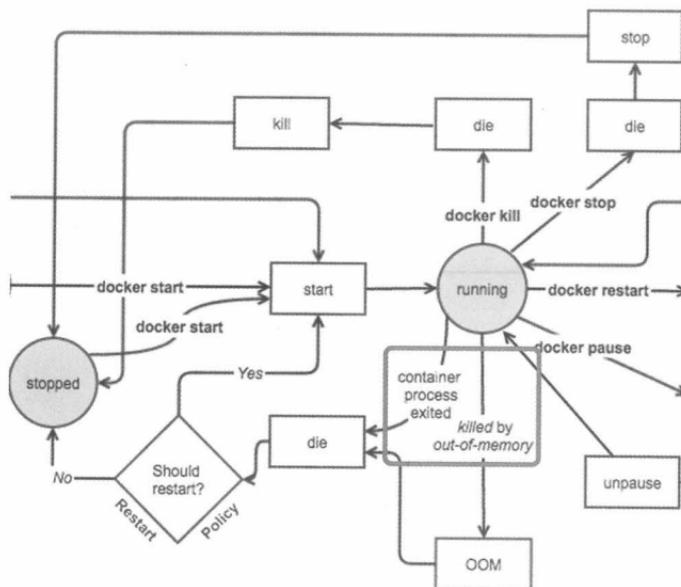


图 4-26

退出包括正常退出或者非正常退出。这里举了两个例子：启动进程正常退出或发生 OOM，此时 Docker 会根据 --restart 的策略判断是否需要重启容器。但如果容器是因为执行 docker stop 或 docker kill 退出，则不会自动重启。

4.6 资源限制

一个 docker host 上会运行若干容器，每个容器都需要 CPU、内存和 IO 资源。对于 KVM、VMware 等虚拟化技术，用户可以控制分配多少 CPU、内存资源给每个虚拟机。对于容器，Docker 也提供了类似的机制避免某个容器因占用太多资源而影响其他容器乃至整个 host 的性能。

4.6.1 内存限额

与操作系统类似，容器可使用的内存包括两部分：物理内存和 swap。Docker 通过下面两组参数来控制容器内存的使用量。

(1) `-m` 或 `--memory`: 设置内存的使用限额，例如 100MB, 2GB。

(2) `--memory-swap`: 设置内存+swap 的使用限额。

当我们执行如下命令：

```
docker run -m 200M --memory-swap=300M ubuntu
```

其含义是允许该容器最多使用 200MB 的内存和 100MB 的 swap。默认情况下，上面两组参数为 `-1`，即对容器内存和 swap 的使用没有限制。

下面我们将使用 `program/stress` 镜像来学习如何为容器分配内存。该镜像可用于对容器执行压力测试。执行如下命令：

```
docker run -it -m 200M --memory-swap=300M program/stress --vm 1 --vm-bytes 280M
```

- `--vm 1`: 启动 1 个内存工作线程。
- `--vm-bytes 280M`: 每个线程分配 280MB 内存。

运行结果如图 4-27 所示。

```
root@ubuntu:~# docker run -it -m 200M --memory-swap=300M program/stress --vm 1 --vm-bytes 280M
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: debug: [1] using backoff sleep of 3000us
stress: debug: [1] --> hogvm worker 1 [5] forked
stress: debug: [5] allocating 293601280 bytes ...
stress: debug: [5] touching bytes in strides of 4096 bytes ...
stress: debug: [5] freed 293601280 bytes
stress: debug: [5] allocating 293601280 bytes ...
stress: debug: [5] touching bytes in strides of 4096 bytes ...
stress: debug: [5] freed 293601280 bytes
stress: debug: [5] allocating 293601280 bytes ...
```

图 4-27

因为 280MB 在可分配的范围（300MB）内，所以工作线程能够正常工作，其过程是：

- (1) 分配 280MB 内存。
- (2) 释放 280MB 内存。
- (3) 再分配 280MB 内存。
- (4) 再释放 280MB 内存。
- (5) 一直循环……

如果让工作线程分配的内存超过 300MB，结果如图 4-28 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it -m 200M --memory-swap=300M program/stress --vm 1 --vm-bytes 310M
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: dbug: [1] using backoff sleep of 3000us
stress: dbug: [1] -> hogvm worker 1 [5] forked
stress: dbug: [5] allocating 325058560 bytes ...
stress: dbug: [5] touching bytes in strides of 4096 bytes ...
stress: FAIL: [1] (416) <-- worker 5 got signal 9
stress: WARN: [1] (418) now reaping child worker processes
stress: FAIL: [1] (422) kill error: No such process
stress: FAIL: [1] (452) failed run completed in 0s
root@ubuntu:~#
```

图 4-28

分配的内存超过限额，stress 线程报错，容器退出。

如果在启动容器时只指定 -m 而不指定 --memory-swap，那么 --memory-swap 默认为 -m 的两倍，比如：

```
docker run -it -m 200M ubuntu
```

容器最多使用 200MB 物理内存和 200MB swap。

4.6.2 CPU 限额

默认设置下，所有容器可以平等地使用 host CPU 资源并且没有限制。

Docker 可以通过 -c 或 --cpu-shares 设置容器使用 CPU 的权重。如果不指定，默认值为 1024。

与内存限额不同，通过 -c 设置的 cpu share 并不是 CPU 资源的绝对数量，而是一个相对的权重值。某个容器最终能分配到的 CPU 资源取决于它的 cpu share 占所有容器 cpu share 总和的比例。

换句话说：通过 cpu share 可以设置容器使用 CPU 的优先级。

比如在 host 中启动了两个容器：

```
docker run --name "container_A" -c 1024 ubuntu docker run --name
"container_B" -c 512 ubuntu
```

containerA 的 cpu share 1024，是 containerB 的两倍。当两个容器都需要 CPU 资源时，containerA 可以得到的 CPU 是 containerB 的两倍。

需要特别注意的是，这种按权重分配 CPU 只会发生在 CPU 资源紧张的情况下。如果 containerA 处于空闲状态，这时，为了充分利用 CPU 资源，containerB 也可以分配到全部可用的 CPU。

下面我们继续用 program/stress 做实验。

(1) 启动 container_A，cpu share 为 1024，如图 4-29 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run --name container_A -it -c 1024 program/stress --cpu 1
stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd
stress: dbug: [1] using backoff sleep of 3000us
stress: dbug: [1] --> hogcpu worker 1 [5] forked
```

图 4-29

--cpu 用来设置工作线程的数量。因为当前 host 只有 1 颗 CPU，所以一个工作线程就能将 CPU 压满。如果 host 有多颗 CPU，则需要相应增加 --cpu 的数量。

(2) 启动 container_B，cpu share 为 512，如图 4-30 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run --name container_B -it -c 512 program/stress --cpu 1
stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd
stress: dbug: [1] using backoff sleep of 3000us
stress: dbug: [1] --> hogcpu worker 1 [5] forked
```

图 4-30

(3) 在 host 中执行 top，查看容器对 CPU 的使用情况，如图 4-31 所示。

```
top - 16:22:19 up 15:43, 4 users,  load average: 1.83, 1.28, 0.68
Tasks: 135 total,  3 running, 132 sleeping,  0 stopped,  0 zombie
%CPU(s): 100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 2048464 total, 1280840 free, 116632 used, 650992 buff/cache
KiB Swap: 2097148 total, 2097148 free,      0 used. 1739292 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	VMEM	TIME+	COMMAND
4039	root	20	0	7316	100	0 R	66.2	0.0	1:25.93	stress	container_A
4197	root	20	0	7316	96	0 R	33.1	0.0	0:38.91	stress	container_B
2711	root	10	-10	5724	3520	2428 S	0.3	0.2	0:08.27	iscsid	
3132	root	20	0	41824	3776	3160 R	0.3	0.2	0:00.29	top	
1	root	20	0	37688	5792	4028 S	0.0	0.3	0:32.37	systemd	
2	root	20	0	0	0	0 S	0.0	0.0	0:00.00	kthreadd	

图 4-31

containerA 消耗的 CPU 是 containerB 的两倍。

(4) 现在暂停 container_A，如图 4-32 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker pause container_A
container_A
root@ubuntu:~#
```

图 4-32

(5) top 显示 containerB 在 containerA 空闲的情况下能够用满整颗 CPU, 如图 4-33 所示。

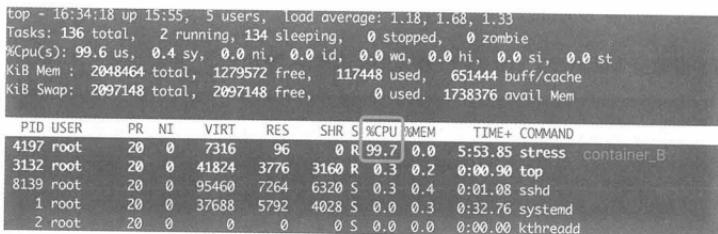


图 4-33

4.6.3 Block IO 带宽限额

Block IO 是另一种可以限制容器使用的资源。Block IO 指的是磁盘的读写, docker 可通过设置权重、限制 bps 和 iops 的方式控制容器读写磁盘的带宽, 下面分别讨论。

注: 目前 Block IO 限额只对 direct IO (不使用文件缓存) 有效。

1. block IO 权重

默认情况下, 所有容器能平等地读写磁盘, 可以通过设置 `--blkio-weight` 参数来改变容器 block IO 的优先级。

`--blkio-weight` 与 `--cpu-shares` 类似, 设置的是相对权重值, 默认为 500。在下面的例子中, containerA 读写磁盘的带宽是 containerB 的两倍。

```
docker run -it --name container_A --blkio-weight 600 ubuntu docker
run -it --name container_B --blkio-weight 300 ubuntu
```

2. 限制 bps 和 iops

bps 是 byte per second, 每秒读写的数据量。

iops 是 io per second, 每秒 IO 的次数。

可通过以下参数控制容器的 bps 和 iops:

- `--device-read-bps`: 限制读某个设备的 bps。
- `--device-write-bps`: 限制写某个设备的 bps。
- `--device-read-iops`: 限制读某个设备的 iops。

- `--device-write-iops`: 限制写某个设备的 iops。

下面这个例子限制容器写 `/dev/sda` 的速率为 30 MB/s:

```
docker run -it --device-write-bps /dev/sda:30MB ubuntu
```

我们来看看实验结果, 如图 4-34 所示。

```
root@ubuntu:~# docker run -it --device-write-bps /dev/sda:30MB ubuntu
root@fdfff202033d:/#
root@fdfff202033d:/# time dd if=/dev/zero of=test.out bs=1M count=800 oflag=direct
800+0 records in
800+0 records out
838860800 bytes (839 MB, 800 MiB) copied, 32.8017 s, 25.6 MB/s

real    0m32.804s
user    0m0.000s
sys     0m0.176s
root@fdfff202033d:/#
```

图 4-34

通过 `dd` 测试在容器中写磁盘的速度。因为容器的文件系统是在 host `/dev/sda` 上的, 在容器中写文件相当于对 host `/dev/sda` 进行写操作。另外, `oflag=direct` 指定用 direct IO 方式写文件, 这样 `--device-write-bps` 才能生效。

结果表明, bps 25.6 MB/s 没有超过 30 MB/s 的限速。

作为对比测试, 如果不限速, 结果如图 4-35 所示。

```
root@ubuntu:~# docker run -it ubuntu
root@3822684457f8:/#
root@3822684457f8:/# time dd if=/dev/zero of=test.out bs=1M count=800 oflag=direct
800+0 records in
800+0 records out
838860800 bytes (839 MB, 800 MiB) copied, 0.529198 s, 1.6 GB/s

real    0m0.531s
user    0m0.004s
sys     0m0.108s
root@3822684457f8:/#
```

图 4-35

其他参数的使用方法类似, 留给大家自己练习。

4.7 实现容器的底层技术

为了更好地理解容器的特性, 本节我们将讨论容器的底层实现技术。

`cgroup` 和 `namespace` 是最重要的两种技术。`cgroup` 实现资源限额, `namespace` 实现资源隔离。

4.7.1 cgroup

cgroup 全称 Control Group。Linux 操作系统通过 cgroup 可以设置进程使用 CPU、内存和 IO 资源的限额。相信你已经猜到了：前面我们看到的 `--cpu-shares`、`-m`、`--device-write-bps` 实际上就是在配置 cgroup。

cgroup 到底长什么样子呢？我们可以在 `/sys/fs/cgroup` 中找到它。还是用例子来说明，启动一个容器，设置 `--cpu-shares=512`，如图 4-36 所示。

```
root@ubuntu:~# docker run -it --cpu-shares 512 program/stress -c 1
stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd
stress: dbug: [1] using backoff sleep of 3000us
stress: dbug: [1] --> hogcpu worker 1 [5] forked
```

图 4-36

查看容器的 ID，如图 4-37 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
c35651ce130e	program/stress	"/usr/bin/stress --ve"	24 seconds ago	Up 24 seconds

图 4-37

在 `/sys/fs/cgroup/cpu/docker` 目录中，Linux 会为每个容器创建一个 cgroup 目录，以容器长 ID 命名，如图 4-38 所示。

```
root@ubuntu:~# ls /sys/fs/cgroup/cpu/docker/c35651ce130e983e4bf50a12d7e7b06c2f566574083f652d5912c69bdd97a24/
cgroup.clone_children  cpacct.stat          cpacct.usage_percpu  cpu.cfs_quota_us   cpu.stat           tasks
cgroup.procs          cpacct.usage        cpu.cfs_period_us  cpu.shares        notify_on_release
root@ubuntu:~# cat /sys/fs/cgroup/cpu/docker/c35651ce130e983e4bf50a12d7e7b06c2f566574083f652d5912c69bdd97a24/cpu.shares
512
root@ubuntu:~#
```

图 4-38

目录中包含所有与 cpu 相关的 cgroup 配置，文件 `cpu.shares` 保存的就是 `--cpu-shares` 的配置，值为 512。

同样的，`/sys/fs/cgroup/memory/docker` 和 `/sys/fs/cgroup/blkio/docker` 中保存的是内存以及 Block IO 的 cgroup 配置。

4.7.2 namespace

在每个容器中，我们都可以看到文件系统、网卡等资源，这些资源看上去是容器自己的。拿网卡来说，每个容器都会认为自己有一块独立的网卡，即使 host 上只有一块物理网卡。这种方式非常好，它使得容器更像一个独立的计算机。

Linux 实现这种方式的技术是 namespace。namespace 管理着 host 中全局唯一的资源，并可以让每个容器都觉得只有自己在使用它。换句话说，namespace 实现了容器间资源的隔离。

Linux 使用了 6 种 namespace，分别对应 6 种资源：Mount、UTS、IPC、PID、Network 和 User，下面我们分别讨论。

1. Mount namespace

Mount namespace 让容器看上去拥有整个文件系统。

容器有自己的 / 目录，可以执行 mount 和 umount 命令。当然我们知道这些操作只在当前容器中生效，不会影响到 host 和其他容器。

2. UTS namespace

简单地说，UTS namespace 让容器有自己的 hostname。默认情况下，容器的 hostname 是它的短 ID，可以通过 -h 或 --hostname 参数设置，如图 4-39 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -h myhost -it ubuntu
root@myhost:#
root@myhost:~# hostname
myhost
root@myhost:~#
```

图 4-39

3. IPC namespace

IPC namespace 让容器拥有自己的共享内存和信号量（semaphore）来实现进程间通信，而不会与 host 和其他容器的 IPC 混在一起。

4. PID namespace

前面提到过，容器在 host 中以进程的形式运行。例如当前 host 中运行了两个容器，如图 4-40 所示。

	CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
34e61d26316		httpd	"httpd-foreground"	2 minutes ago	Up 2 minutes	80/tcp	stupified_liskov
5b946f262fcf		ubuntu	"/bin/bash"	11 minutes ago	Up 11 minutes		sick_hodkin

图 4-40

通过 ps auxf 可以查看容器进程，如图 4-41 所示。

2259 ?	Ssl	0:24	/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0				
2378 ?	Ssl	0:03	_ docker-containernerd -l unix:///var/run/docker/libcontainernd/docker-containernerd.sock				
7634 ?	Sl	0:00	①_ docker-containernerd-shim 5b946f262fcf32ba6d779158ec7e308e1c621eaee6ada2e215eb				
7664 pts/0	Ss+	0:00	_ /bin/bash				
8268 ?	Sl	0:00	②_ docker-containernerd-shim 34e61d12a8169e21e1b844dd4fe74a1ca6d8b049f2a8061111503				
8294 ?	Ss	0:00	_ httpd -DFOREGROUND				
8336 ?	Sl	0:00	_ httpd -DFOREGROUND				
8337 ?	Sl	0:00	_ httpd -DFOREGROUND				
8338 ?	Sl	0:00	_ httpd -DFOREGROUND				

图 4-41

所有容器的进程都挂在 dockerd 进程下，同时也可以看到容器自己的子进程。如果我们进

进入到某个容器，ps 就只能看到自己的进程了，如图 4-42 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker exec -it 34e61d12a816 bash
root@34e61d12a816:/usr/local/apache2#
root@34e61d12a816:/usr/local/apache2# ps axf
  PID TTY      STAT   TIME COMMAND
 90 ?        Ss     0:00 bash
 94 ?        R+     0:00  \_ ps axf
 1 ?        Ss     0:00 httpd -DFOREGROUND
 6 ?        Sl     0:00 httpd -DFOREGROUND
 7 ?        Sl     0:00 httpd -DFOREGROUND
 8 ?        Sl     0:00 httpd -DFOREGROUND
root@34e61d12a816:/usr/local/apache2#
```

图 4-42

而且进程的 PID 不同于 host 中对应进程的 PID，容器中 PID=1 的进程当然也不是 host 的 init 进程。也就是说：容器拥有自己独立的一套 PID，这就是 PID namespace 提供的功能。

5. Network namespace

Network namespace 让容器拥有自己独立的网卡、IP、路由等资源，我们会在后面网络章节详细讨论。

6. User namespace

User namespace 让容器能够管理自己的用户，host 不能看到容器中创建的用户，如图 4-43 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker exec -it 34e61d12a816 bash
root@34e61d12a816:/usr/local/apache2#
root@34e61d12a816:/usr/local/apache2# useradd cloudman
root@34e61d12a816:/usr/local/apache2#
root@34e61d12a816:/usr/local/apache2# exit
exit
root@ubuntu:~# su - cloudman
No passwd entry for user 'cloudman'
root@ubuntu:~#
```

图 4-43

在容器中创建了用户 cloudman，但 host 中并不会创建相应的用户。

4.8 小结

本章首先通过大量实验学习了容器的各种操作以及容器状态之间如何转换，然后讨论了限制容器使用 CPU、内存和 Block IO 的方法，最后学习了实现容器的底层技术：cgroup 和 namespace。

下面是容器的常用操作命令：

create: 创建容器;

run: 运行容器;

pause: 暂停容器;

unpause: 取消暂停继续运行容器;

stop: 发送 SIGTERM 停止容器;

kill: 发送 SIGKILL 快速停止容器;

start: 启动容器;

restart: 重启容器;

attach: attach 到容器启动进程的终端;

exec: 在容器中启动新进程, 通常使用 "-it" 参数;

logs: 显示容器启动进程的控制台输出, 用 "-f" 持续打印;

rm: 从磁盘中删除容器。

第 5 章

◀Docker 网络▶

本章讨论 Docker 网络。

我们会首先学习 Docker 提供的几种原生网络，以及如何创建自定义网络；然后探讨容器之间如何通信，以及容器与外界如何交互。

Docker 网络从覆盖范围可分为单个 host 上的容器网络和跨多个 host 的网络，本章重点讨论前一种。对于更为复杂的多 host 容器网络，我们会在后面进阶技术章节单独讨论。

Docker 安装时会自动在 host 上创建三个网络，我们可用 docker network ls 命令查看，如图 5-1 所示。

```
root@ubuntu:~# docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
cb325ec4bbe5    bridge    bridge      local
f48f4d42ceec    host      host       local
c252509338fd    none     null       local
root@ubuntu:~#
```

图 5-1

下面我们分别讨论它们。

5.1 none 网络

顾名思义，none 网络就是什么都没有的网络。挂在这个网络下的容器除了 lo，没有其他任何网卡。容器创建时，可以通过 --network=none 指定使用 none 网络，如图 5-2 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --network=none busybox
/ #
/ # ifconfig
lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
/ #
```

图 5-2

我们不禁会问，这样一个封闭的网络有什么用呢？

其实还真有应用场景。封闭意味着隔离，一些对安全性要求高并且不需要联网的应用可以使用 none 网络。

比如某个容器的唯一用途是生成随机密码，就可以放到 none 网络中避免密码被窃取。

当然大部分容器是需要网络的，我们接着看 host 网络。

5.2 host 网络

连接到 host 网络的容器共享 Docker host 的网络栈，容器的网络配置与 host 完全一样。可以通过 `--network=host` 指定使用 host 网络，如图 5-3 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --network=host busybox
/ #
/ # ip l
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 08:00:27:5f:79:3f brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 08:00:27:9c:21:5e brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:14:26:0f:c4 brd ff:ff:ff:ff:ff:ff
7: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue qlen 1000
    link/ether 52:54:00:96:f4:fa brd ff:ff:ff:ff:ff:ff
8: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast master virbr0 qlen 1000
    link/ether 52:54:00:96:f4:fa brd ff:ff:ff:ff:ff:ff
/ #
/ # hostname
ubuntu
/ #
```

图 5-3

在容器中可以看到 host 的所有网卡，并且连 hostname 也是 host 的。host 网络的使用场景又是什么呢？

直接使用 Docker host 的网络最大的好处就是性能，如果容器对网络传输效率有较高要求，则可以选择 host 网络。当然不便之处就是牺牲一些灵活性，比如要考虑端口冲突问题，Docker host 上已经使用的端口就不能再用了。

Docker host 的另一个用途是让容器可以直接配置 host 网路，比如某些跨 host 的网络解决方案，其本身也是以容器方式运行的，这些方案需要对网络进行配置，比如管理 iptables，大家将会在后面进阶技术章节看到。

下面讨论应用更广的 bridge 网络。

5.3 bridge 网络

Docker 安装时会创建一个命名为 docker0 的 Linux bridge。如果不指定--network，创建的容器默认都会挂到 docker0 上，如图 5-4 所示。

```
8c166c98adbff
root@ubuntu:~# brctl show
bridge name      bridge id          STP enabled     interfaces
docker0          8000.02421426fc4   no
virbr0           8000.52540096f4fa  yes            virbr0-nic
root@ubuntu:~#
```

图 5-4

当前 docker0 上没有任何其他网络设备，我们创建一个容器看看有什么变化，如图 5-5 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d httpd
2b668e52480e493beff35e8bef8ca6dac0241afc8a332217511995f0c383f38
root@ubuntu:~#
root@ubuntu:~# brctl show
bridge name      bridge id          STP enabled     interfaces
docker0          8000.02421426fc4   no
virbr0           8000.52540096f4fa  yes            veth28c57df
                                         virbr0-nic
root@ubuntu:~#
```

图 5-5

一个新的网络接口 veth28c57df 被挂到了 docker0 上，veth28c57df 就是新创建容器的虚拟网卡。

下面看一下容器的网络配置，如图 5-6 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker exec -it 2b668e52480e bash
root@2b668e52480e:/usr/local/apache2#
root@2b668e52480e:/usr/local/apache2# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inette ::1/128 scope host
        valid_lft forever preferred_lft forever
33: eth0@if34: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.0.255 scope global eth0
        valid_lft forever preferred_lft forever
    inette fe80::42:acff:fe11:2/64 scope link
        valid_lft forever preferred_lft forever
root@2b668e52480e:/usr/local/apache2#
```

图 5-6

容器有一个网卡 eth0@if34。大家可能会问了，为什么不是 veth28c57df 呢？

实际上 eth0@if34 和 veth28c57df 是一对 veth pair。veth pair 是一种成对出现的特殊网络设备，可以把它们想象成由一根虚拟网线连接起来的一对网卡，网卡的一头（eth0@if34）在容器中，另一头（veth28c57df）挂在网桥 docker0 上，其效果就是将 eth0@if34 也挂在了 docker0 上。

我们还看到 eth0@if34 已经配置了 IP 172.17.0.2，为什么是这个网段呢？让我们通过 docker network inspect bridge 看一下 bridge 网络的配置信息，如图 5-7 所示。

```
root@ubuntu:~# docker network inspect bridge
[{"Name": "bridge",
 "Id": "cb325ec4bbe514a15005a089f47ae1af345ea1cff7ef9db7446d85ff426cf6d",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
     "Driver": "default",
     "Options": null,
     "Config": [
         {
             "Subnet": "172.17.0.0/16",
             "Gateway": "172.17.0.1"
         }
     ]
 },
 "Containers": {}}
```

图 5-7

原来 bridge 网络配置的 subnet 就是 172.17.0.0/16，并且网关是 172.17.0.1。这个网关在哪儿呢？大概你已经猜出来了，就是 docker0，如图 5-8 所示。

当前容器网络拓扑结构如图 5-9 所示。

```
root@ubuntu:~#
root@ubuntu:~# ifconfig docker0
docker0: Link encap:Ethernet HWaddr 02:42:14:26:0f:c4
      inet addr:172.17.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
        inet6 addr: fe80::42:14ff:fe26:fc4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:97 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:6452 (6.4 KB) TX bytes:648 (648.0 B)
```

图 5-8

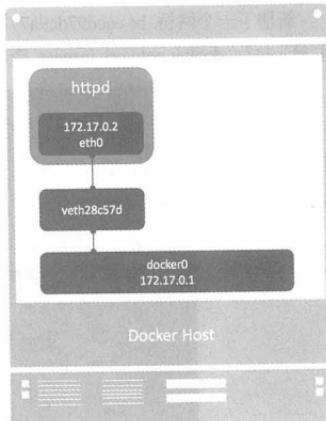


图 5-9

容器创建时，docker 会自动从 172.17.0.0/16 中分配一个 IP，这里 16 位的掩码保证有足够的 IP 可以供容器使用。

5.4 user-defined 网络

除了 none、host、bridge 这三个自动创建的网络，用户也可以根据业务需要创建 user-defined 网络。

Docker 提供三种 user-defined 网络驱动：bridge、overlay 和 macvlan。overlay 和 macvlan 用于创建跨主机的网络，我们后面有章节单独讨论。

我们可通过 bridge 驱动创建类似前面默认的 bridge 网络，如图 5-10 所示的例子。

```
root@ubuntu:~#
root@ubuntu:~# docker network create --driver bridge my_net
eaed97dc9a773d8894d96fe0a88058707feb1206406bf5209c7efb86ea037d
root@ubuntu:~#
```

图 5-10

查看一下当前 host 的网络结构变化，如图 5-11 所示。

```
root@ubuntu:~#
root@ubuntu:~# brctl show
bridge name      bridge id          STP enabled     interfaces
br-eaed97dc9a77    8000.02429372fe2f    no
docker0           8000.024214260fc4    no
virbr0            8000.52540096f4fa    yes
virbr0-nic        -                -             -
root@ubuntu:~#
```

图 5-11

新增了一个网桥 br-eaed97dc9a77，这里 eaed97dc9a77 正好是新建 bridge 网络 my_net 的短 id。执行 docker network inspect 查看一下 my_net 的配置信息，如图 5-12 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker network inspect my_net
[{"Name": "my_net",
 "Id": "eaed97dc9a773d8894d96fe0a88058707feb1206406bf5209c7efb86ea037d",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
 "Driver": "default",
 "Options": {},
 "Config": [
 {
 "Subnet": "172.18.0.0/16",
 "Gateway": "172.18.0.1/16"
 }
 ],
 "Internal": false,
 "Containers": {}
 },
 "Options": {},
 "Labels": {}
}]
```

图 5-12

这里 172.18.0.0/16 是 Docker 自动分配的 IP 网段。

我们可以自己指定 IP 网段吗？

答案是：可以。

只需在创建网段时指定 `--subnet` 和 `--gateway` 参数，如图 5-13 所示。

```
root@ubuntu:~# docker network create --driver bridge --subnet 172.22.16.0/24 --gateway 172.22.16.1 my_net2
5d863e9f78b6b0458b0cb2d9b751d1a93095c63bf1ab883b329b138286c17d66
root@ubuntu:~# docker network inspect my_net2
[{"Name": "my_net2", "Id": "5d863e9f78b6b0458b0cb2d9b751d1a93095c63bf1ab883b329b138286c17d66", "Scope": "local", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "default", "Options": {}, "Config": [{"Subnet": "172.22.16.0/24", "Gateway": "172.22.16.1"}]}, "Internal": false, "Containers": {}, "Options": {}, "Labels": {}}]
```

图 5-13

这里我们创建了新的 bridge 网络 `my_net2`，网段为 `172.22.16.0/24`，网关为 `172.22.16.1`。与前面一样，网关在 `my_net2` 对应的网桥 `br-5d863e9f78b6` 上，如图 5-14 所示。

```
root@ubuntu:~# ifconfig br-5d863e9f78b6
br-5d863e9f78b6 Link encap:Ethernet HWaddr 02:42:c7:6b:ee:e2
          inet addr 172.22.16.1 Bcast:0.0.0.0 Mask:255.255.255.0
              UP BROADCAST MULTICAST MTU:1500 Metric:1
              RX packets:0 errors:0 dropped:0 overruns:0 frame:0
              TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:0
              RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

图 5-14

容器要使用新的网络，需要在启动时通过 `--network` 指定，如图 5-15 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --network=my_net2 busybox
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inetc6 ::1/128 scope host
        valid_lft forever preferred_lft forever
38: eth0@if39: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:16:10:02 brd ff:ff:ff:ff:ff:ff
    inet 172.22.16.2/24 scope global eth0
        valid_lft forever preferred_lft forever
    inetc6 fe80::42:acff:fe16:1002/64 scope link
        valid_lft forever preferred_lft forever
/ #
```

图 5-15

容器分配到的 IP 为 172.22.16.2。

到目前为止，容器的 IP 都是 docker 自动从 subnet 中分配，我们能否指定一个静态 IP 呢？答案是：可以，通过--ip 指定，如图 5-16 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --network=my_net2 --ip 172.22.16.8 busybox
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inetc6 ::1/128 scope host
        valid_lft forever preferred_lft forever
40: eth0@if41: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:16:10:08 brd ff:ff:ff:ff:ff:ff
    inet [172.22.16.8/24] scope global eth0
        valid_lft forever preferred_lft forever
    inetc6 fe80::42:acff:fe16:1008/64 scope link
        valid_lft forever preferred_lft forever
/ #
```

图 5-16

注：只有使用 --subnet 创建的网络才能指定静态 IP。

my_net 创建时没有指定 --subnet，如果指定静态 IP 报错如下，如图 5-17 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --network=my_net --ip 172.18.0.8 busybox
docker: Error response from daemon: User specified IP address is supported only when connecting to net
works with user configured subnets.
root@ubuntu:~#
```

图 5-17

好了，我们来看看当前 docker host 的网络拓扑结构，如图 5-18 所示。

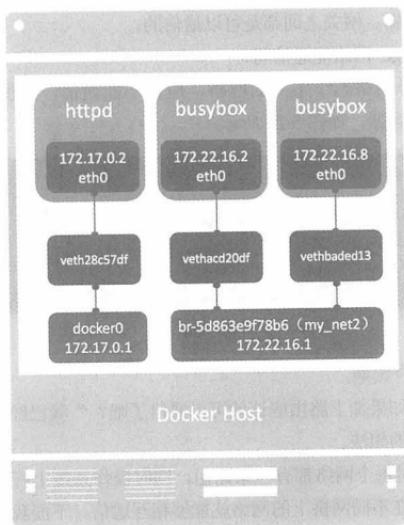


图 5-18

两个 busybox 容器都挂在 my_net2 上，应该能够互通，我们验证一下，如图 5-19 所示。

```

/ #
/ # ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 02:42:AC:16:10:02
          inet addr 172.22.16.2  Bcast 0.0.0.0  Mask:255.255.255.0
              inet6 addr: fe80::42:acff:fe16:1002/64 Scope:Link
                  UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                  RX packets:47 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:31 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:0
                  RX bytes:3974 (3.8 Kib)  TX bytes:2678 (2.6 Kib)

/ # ping -c 3 172.22.16.8
PING 172.22.16.8 (172.22.16.8): 56 data bytes
64 bytes from 172.22.16.8: seq=0 ttl=64 time=0.075 ms
64 bytes from 172.22.16.8: seq=1 ttl=64 time=0.083 ms
64 bytes from 172.22.16.8: seq=2 ttl=64 time=0.085 ms

--- 172.22.16.8 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.075/0.081/0.085 ms
/ #

/ # ping -c 3 172.22.16.1
PING 172.22.16.1 (172.22.16.1): 56 data bytes
64 bytes from 172.22.16.1: seq=0 ttl=64 time=0.089 ms
64 bytes from 172.22.16.1: seq=1 ttl=64 time=0.093 ms
64 bytes from 172.22.16.1: seq=2 ttl=64 time=0.096 ms

--- 172.22.16.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.089/0.092/0.096 ms
/ #

```

图 5-19

可见同一网络中的容器、网关之间都是可以通信的。

`my_net2` 与默认 `bridge` 网络能通信吗？

从拓扑图可知，两个网络属于不同的网桥，应该不能通信，我们通过实验验证一下，让 `busybox` 容器 `ping` `httpd` 容器，如图 5-20 所示。

```
/ #
/ # ping -c 3 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
^C
--- 172.17.0.2 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
/ #
```

图 5-20

确实 `ping` 不通，符合预期。

“等等！不同的网络如果加上路由应该就可以通信了吧？”我已经听到有读者在建议了。这是一个非常非常好的想法。

确实，如果 `host` 上对每个网络都有一条路由，同时操作系统上打开了 `ip forwarding`，`host` 就成了一个路由器，挂接在不同网桥上的网络就能够相互通信。下面我们来看看 `docker host` 是否满足这些条件呢？

`ip r` 查看 `host` 上的路由表：

```
ip r
.....
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
172.22.16.0/24 dev br-5d863e9f78b6 proto kernel scope link src
172.22.16.1
.....
```

`172.17.0.0/16` 和 `172.22.16.0/24` 两个网络的路由都定义好了。再看看 `ip forwarding`：

```
sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

`ip forwarding` 也已经启用了。条件都满足，为什么不能通行呢？

我们还得看看 `iptables`：

```
iptables-save
.....
-A DOCKER-ISOLATION -i br-5d863e9f78b6 -o docker0 -j DROP -A DOCKER-
ISOLATION -i docker0 -o br-5d863e9f78b6 -j DROP
.....
```

原因就在这里了：`iptables` `DROP` 掉了网桥 `docker0` 与 `br-5d863e9f78b6` 之间双向的流量。

从规则的命名 DOCKER-ISOLATION 可知 docker 在设计上就是要隔离不同的 network。那么接下来的问题是：怎样才能让 busybox 与 httpd 通信呢？

答案是：为 httpd 容器添加一块 net_my2 的网卡。这个可以通过 docker network connect 命令实现，如图 5-21 所示。

```
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
223849fdf8ce      busybox             "sh"               43 minutes ago   Up 43 minutes
fe532b88e0be      busybox             "sh"               5 hours ago      Up 5 hours
2b668e52480e       httpd              "httpd-foreground" 9 hours ago      Up 9 hours
root@ubuntu:~# docker network connect my_net2 2b668e52480e
root@ubuntu:~#
```

图 5-21

我们在 httpd 容器中查看一下网络配置，如图 5-22 所示。

```
root@2b668e52480e:/usr/local/apache2# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 brd :: scope host
        valid_lft forever preferred_lft forever
3: eth0@if34: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.0.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:2/64 brd ff:ff:ff:ff:ff:ff scope link
        valid_lft forever preferred_lft forever
42: eth1@if43: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:16:10:03 brd ff:ff:ff:ff:ff:ff
    inet 172.22.16.3/24 brd 172.22.16.255 scope global eth1
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe16:1003/64 brd ff:ff:ff:ff:ff:ff scope link
        valid_lft forever preferred_lft forever
root@2b668e52480e:/usr/local/apache2#
```

图 5-22

容器中增加了一个网卡 eth1，分配了 my_net2 的 IP 172.22.16.3。现在 busybox 应该能够访问 httpd 了，验证一下，如图 5-23 所示。

```
/ #
/ # ping -c 3 172.22.16.3
PING 172.22.16.3 (172.22.16.3): 56 data bytes
64 bytes from 172.22.16.3: seq=0 ttl=64 time=0.111 ms
64 bytes from 172.22.16.3: seq=1 ttl=64 time=0.088 ms
64 bytes from 172.22.16.3: seq=2 ttl=64 time=0.083 ms
...
--- 172.22.16.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.083/0.094/0.111 ms
/ #
/ # wget 172.22.16.3
Connecting to 172.22.16.3 (172.22.16.3:80)
index.html          100% [*****]
/ #
/ # cat index.html
<html><body><h1>It works!</h1></body></html>
/ #
```

图 5-23

busybox 能够 ping 到 httpd，并且可以访问 httpd 的 Web 服务。当前网络结构如图 5-24 所示。

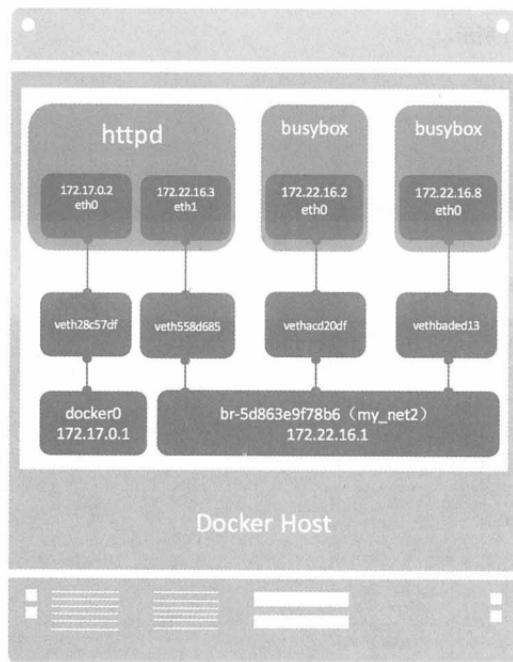


图 5-24

学习了 Docker 各种类型网络之后，接下来我们讨论容器与容器、容器与外界的连通问题。

5.5 容器间通信

容器之间可通过 IP, Docker DNS Server 或 joined 容器三种方式通信。

5.5.1 IP 通信

从前面的例子可以得出这样一个结论：两个容器要能通信，必须要有属于同一个网络的网卡。满足这个条件后，容器就可以通过 IP 交互了。具体做法是在容器创建时通过 `--network` 指定相应的网络，或者通过 `docker network connect` 将现有容器加入到指定网络。可参考上一节

httpd 和 busybox 的例子，这里不再赘述。

5.5.2 Docker DNS Server

通过 IP 访问容器虽然满足了通信的需求，但还是不够灵活。因为在部署应用之前可能无法确定 IP，部署之后再指定要访问的 IP 会比较麻烦。对于这个问题，可以通过 docker 自带的 DNS 服务解决。

从 Docker 1.10 版本开始，docker daemon 实现了一个内嵌的 DNS server，使容器可以直接通过“容器名”通信。方法很简单，只要在启动时用 `--name` 为容器命名就可以了。

下面启动两个容器 `bbox1` 和 `bbox2`：

```
docker run -it --network=my_net2 --name=bbox1 busybox
docker run -it --network=my_net2 --name=bbox2 busybox
```

然后，`bbox2` 就可以直接 ping 到 `bbox1` 了，如图 5-25 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --network=my_net2 --name=bbox2 busybox
/
/ # ping -c 3 bbox1
PING bbox1 (172.22.16.2): 56 data bytes
64 bytes from 172.22.16.2: seq=0 ttl=64 time=0.079 ms
64 bytes from 172.22.16.2: seq=1 ttl=64 time=0.076 ms
64 bytes from 172.22.16.2: seq=2 ttl=64 time=0.088 ms
```

图 5-25

使用 docker DNS 有个限制：只能在 user-defined 网络中使用。也就是说，默认的 bridge 网络是无法使用 DNS 的。下面验证一下：

创建 `bbox3` 和 `bbox4`，均连接到 bridge 网络。

```
docker run -it --name=bbox3 busybox
docker run -it --name=bbox4 busybox
```

`bbox4` 无法 ping 到 `bbox3`，如图 5-26 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --name=bbox4 busybox
/
/ # ping -c 3 bbox3
ping:(bad address 'bbox3')
/ #
```

图 5-26

5.5.3 joined 容器

`joined` 容器是另一种实现容器间通信的方式。

joined 容器非常特别，它可以使两个或多个容器共享一个网络栈，共享网卡和配置信息，joined 容器之间可以通过 127.0.0.1 直接通信。请看下面的例子：

先创建一个 httpd 容器，名字为 web1。

```
docker run -d -it --name=web1 httpd
```

然后创建 busybox 容器并通过 --network=container:web1 指定 joined 容器为 web1，如图 5-27 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -it --network=container:web1 busybox
/ #
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
60: eth0@if61: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
        inet6 fe80::42:acff:fe11:2/64 scope link
            valid_lft forever preferred_lft forever
/ #
```

图 5-27

请注意 busybox 容器中的网络配置信息，下面我们查看一下 web1 的网络，如图 5-28 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker exec -it web1 bash
root@ffeb49562149:/usr/local/apache2#
root@ffeb49562149:/usr/local/apache2# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
60: eth0@if61: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
        inet6 fe80::42:acff:fe11:2/64 scope link
            valid_lft forever preferred_lft forever
root@ffeb49562149:/usr/local/apache2#
```

图 5-28

看！busybox 和 web1 的网卡 mac 地址与 IP 完全一样，它们共享了相同的网络栈。busybox 可以直接用 127.0.0.1 访问 web1 的 http 服务，如图 5-29 所示。

```

/ #
/ # wget 127.0.0.1
Connecting to 127.0.0.1 (127.0.0.1:80)
index.html      100% |*****| 0.00B/s
/ #
/ # cat index.html
<html><body><h1>It works!</h1></body></html>
/ #

```

图 5-29

joined 容器非常适合以下场景：

- (1) 不同容器中的程序希望通过 loopback 高效快速地通信，比如 Web Server 与 App Server。
- (2) 希望监控其他容器的网络流量，比如运行在独立容器中的网络监控程序。

5.6 将容器与外部世界连接

前面我们已经解决了容器间通信的问题，接下来讨论容器如何与外部世界通信。这里涉及两个方向：

- (1) 容器访问外部世界。
- (2) 外部世界访问容器。

5.6.1 容器访问外部世界

在我们当前的实验环境下，docker host 是可以访问外网的，如图 5-30 所示。

```

root@ubuntu:~# ping -c 3 www.bing.com
PING cn.a-0001.a-msedge.net (202.89.233.104) 56(84) bytes of data.
64 bytes from 202.89.233.104: icmp_seq=1 ttl=63 time=50.1 ms
64 bytes from 202.89.233.104: icmp_seq=2 ttl=63 time=49.4 ms
64 bytes from 202.89.233.104: icmp_seq=3 ttl=63 time=49.7 ms

```

图 5-30

我们看一下容器是否也能访问外网呢？如图 5-31 所示。

```

root@ubuntu:~#
root@ubuntu:~# docker run -it busybox
/ #
/ # ping -c 3 www.bing.com
PING www.bing.com (202.89.233.104): 56 data bytes
64 bytes from 202.89.233.104: seq=0 ttl=61 time=49.211 ms
64 bytes from 202.89.233.104: seq=1 ttl=61 time=50.986 ms
64 bytes from 202.89.233.104: seq=2 ttl=61 time=49.298 ms

```

图 5-31

可见，容器默认就能访问外网。

请注意：这里外网指的是容器网络以外的网络环境，并非特指 Internet。

现象很简单，但更重要的：我们应该理解现象下的本质。

在上面的例子中，busybox 位于 docker0 这个私有 bridge 网络中（172.17.0.0/16），当 busybox 从容器向外 ping 时，数据包是怎样到达 bing.com 的呢？

这里的关键就是 NAT。我们查看一下 docker host 上的 iptables 规则，如图 5-32 所示。

```
root@ubuntu:~# iptables -t nat -S
root@ubuntu:~# [REDACTED]
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A POSTROUTING -s 192.168.122.0/24 -d 224.0.0.0/24 -j RETURN
-A POSTROUTING -s 192.168.122.0/24 -d 255.255.255.255/32 -j RETURN
-A POSTROUTING -s 192.168.122.0/24 ! -d 192.168.122.0/24 -p tcp -j MASQUERADE
-A POSTROUTING -s 192.168.122.0/24 ! -d 192.168.122.0/24 -p udp -j MASQUERADE
-A POSTROUTING -s 192.168.122.0/24 ! -d 192.168.122.0/24 -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
root@ubuntu:~#
```

图 5-32

在 NAT 表中，有这么一条规则：

```
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
```

其含义是：如果网桥 docker0 收到来自 172.17.0.0/16 网段的外出包，把它交给 MASQUERADE 处理。而 MASQUERADE 的处理方式是将包的源地址替换成 host 的地址发出去，即做了一次网络地址转换（NAT）。

下面我们通过 tcpdump 查看地址是如何转换的。先查看 docker host 的路由表，如图 5-33 所示。

```
root@ubuntu:~#
root@ubuntu:~# ip r
default via 10.0.2.2 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.15
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
```

图 5-33

默认路由通过 enp0s3 发出去，所以我们要同时监控 enp0s3 和 docker0 上的 icmp（ping）数据包。

当 busybox ping bing.com 时，tcpdump 输出如下，如图 5-34 所示。

```
root@ubuntu:~# tcpdump -i docker0 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on docker0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:35:09.497340 IP [172.17.0.2]> 202.89.233.104: ICMP echo request, id 2816, seq 0, length 64
16:35:10.498604 IP [172.17.0.2]> 202.89.233.104: ICMP echo reply, id 2816, seq 0, length 64
16:35:10.549093 IP 202.89.233.104 > 172.17.0.2: ICMP echo request, id 2816, seq 1, length 64
16:35:11.500401 IP [172.17.0.2]> 202.89.233.104: ICMP echo request, id 2816, seq 2, length 64
16:35:11.551701 IP 202.89.233.104 > 172.17.0.2: ICMP echo reply, id 2816, seq 2, length 64
```

图 5-34

docker0 收到 busybox 的 ping 包，源地址为容器 IP 172.17.0.2，这没问题，交给 MASQUERADE 处理。这时，在 enp0s3 上我们看到了变化，如图 5-35 所示。

```
root@ubuntu:~# tcpdump -i enp0s3 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
16:35:09.497353 IP [10.0.2.15]> 202.89.233.104: ICMP echo request, id 2816, seq 0, length 64
16:35:09.546873 IP 202.89.233.104 > 10.0.2.15: ICMP echo reply, id 2816, seq 0, length 64
16:35:10.498625 IP [10.0.2.15]> 202.89.233.104: ICMP echo request, id 2816, seq 1, length 64
16:35:10.549061 IP 202.89.233.104 > 10.0.2.15: ICMP echo reply, id 2816, seq 1, length 64
16:35:11.500423 IP [10.0.2.15]> 202.89.233.104: ICMP echo request, id 2816, seq 2, length 64
16:35:11.551579 IP 202.89.233.104 > 10.0.2.15: ICMP echo reply, id 2816, seq 2, length 64
```

图 5-35

ping 包的源地址变成了 enp0s3 的 IP 10.0.2.15

这就是 iptable NAT 规则处理的结果，从而保证数据包能够到达外网。下面用一张图来说明这个过程，如图 5-36 所示。

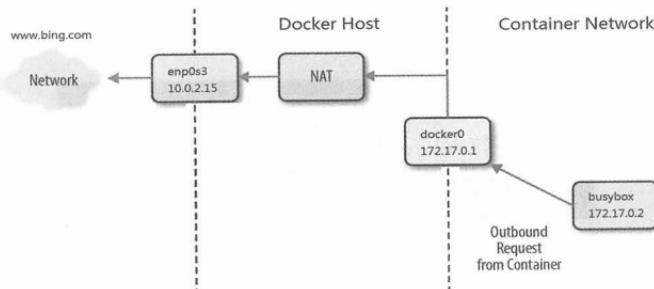


图 5-36

- (1) busybox 发送 ping 包: 172.17.0.2 > www.bing.com。
- (2) docker0 收到包，发现是发送到外网的，交给 NAT 处理。
- (3) NAT 将源地址换成 enp0s3 的 IP: 10.0.2.15 > www.bing.com。
- (4) ping 包从 enp0s3 发送出去，到达 www.bing.com。

通过 NAT，Docker 实现了容器对外网的访问。

5.6.2 外部世界访问容器

下面我们来讨论另一个方向：外网如何访问到容器？

答案是：端口映射。

docker 可将容器对外提供服务的端口映射到 host 的某个端口，外网通过该端口访问容器。容器启动时通过-p 参数映射端口，如图 5-37 所示。

```
root@ubuntu:~# docker run -d -p 80 httpd
add8464d4165a1169a972e5475e43a69eae5265e461bd96f455fc3518d4cf64d
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
add8464d4165        httpd              "httpd-foreground"   23 seconds ago    Up 23 seconds
root@ubuntu:~# docker port add8464d4165
80/tcp -> 0.0.0.0:32773
root@ubuntu:~#
```

图 5-37

容器启动后，可通过 docker ps 或者 docker port 查看到 host 映射的端口。在上面的例子中，httpd 容器的 80 端口被映射到 host 32773 上，这样就可以通过 <host ip>:<32773> 访问容器的 Web 服务了，如图 5-38 所示。

```
root@ubuntu:~#
root@ubuntu:~# curl 10.0.2.15:32773
<html><body><h1>It works!</h1></body></html>
root@ubuntu:~#
```

图 5-38

除了映射动态端口，也可在 -p 中指定映射到 host 某个特定端口，例如可将 80 端口映射到 host 的 8080 端口，如图 5-39 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d -p 8080:80 httpd
5844c401dca2d0395cc43f2a1c8dd282514136072c80b1ce579cdcf06b5760
root@ubuntu:~#
root@ubuntu:~# curl 10.0.2.15:8080
<html><body><h1>It works!</h1></body></html>
root@ubuntu:~#
```

图 5-39

每一个映射的端口，host 都会启动一个 docker-proxy 进程来处理访问容器的流量，如图 5-40 所示。

```
root@ubuntu:~#
root@ubuntu:~# ps -ef | grep docker-proxy
root  21417 2288  0 21:57 ?        00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 32773 -container-ip 172.17.0.2 -container-port 80
root  23314 2288  0 21:14 ?        00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 8080 -container-ip 172.17.0.3 -container-port 80
root  24447 21218  0 21:24 pts/1    00:00:00 grep --color=auto docker-proxy
root@ubuntu:~#
```

图 5-40

以 0.0.0.0:32773->80/tcp 为例分析整个过程，如图 5-41 所示。

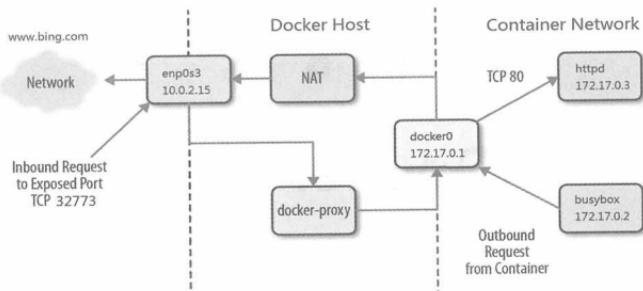


图 5-41

- (1) `docker-proxy` 监听 host 的 32773 端口。
- (2) 当 curl 访问 10.0.2.15:32773 时, `docker-proxy` 转发给容器 172.17.0.2:80。
- (3) `httpd` 容器响应请求并返回结果。

5.7 小结

在这一章我们首先学习了 Docker 的三种网络: `none`、`host` 和 `bridge`, 并讨论了它们的不同使用场景; 然后我们实践了创建自定义网络; 最后详细讨论了如何实现容器与容器之间、容器与外部网络之间的通信。

本章重点关注的是单个主机内的容器网络, 对于跨主机网络通信将在后面章节详细讨论。

第 6 章

◀Docker 存储▶

本章讨论 Docker 存储。

Docker 为容器提供了两种存放数据的资源：

- (1) 由 storage driver 管理的镜像层和容器层。
- (2) Data Volume。

我们会详细讨论它们的原理和特性。

6.1 storage driver

在前面镜像章节我们学习到 Docker 镜像的分层结构，简单回顾一下，如图 6-1 所示。

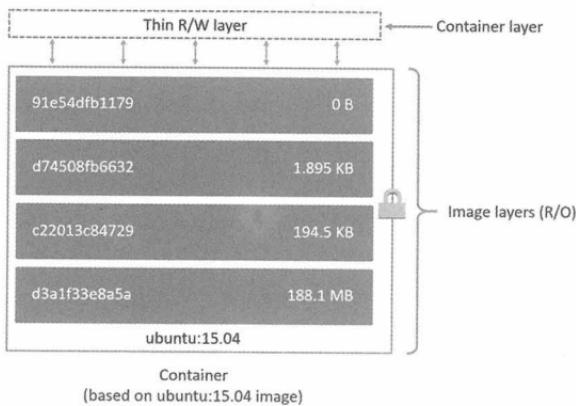


图 6-1

容器由最上面一个可写的容器层，以及若干只读的镜像层组成，容器的数据就存放在这些层中。这样的分层结构最大的特性是 Copy-on-Write：

- (1) 新数据会直接存放在最上面的容器层。
- (2) 修改现有数据会先从镜像层将数据复制到容器层，修改后的数据直接保存在容器层中，镜像层保持不变。
- (3) 如果多个层中有命名相同的文件，用户只能看到最上面那层中的文件。

分层结构使镜像和容器的创建、共享以及分发变得非常高效，而这些都要归功于 Docker storage driver。正是 storage driver 实现了多层数据的堆叠并为用户提供一个单一的合并之后的统一视图。

Docker 支持多种 storage driver，有 AUFS、Device Mapper、Btrfs、OverlayFS、VFS 和 ZFS。它们都能实现分层的架构，同时又有各自的特性。对于 Docker 用户来说，具体选择使用哪个 storage driver 是一个难题，因为：

- (1) 没有哪个 driver 能够适应所有的场景。
- (2) driver 本身在快速发展和迭代。

不过 Docker 官方给出了一个简单的答案：优先使用 Linux 发行版默认的 storage driver。

Docker 安装时会根据当前系统的配置选择默认的 driver。默认 driver 具有最好的稳定性，因为默认 driver 在发行版上经过了严格的测试。

运行 docker info 查看 Ubuntu 的默认 driver，如图 6-2 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker info
Containers: 5
Running: 3
Paused: 0
Stopped: 2
Images: 33
Server Version: 1.12.3
Storage Driver: aufs
  Root Dir: /var/lib/docker/aufs
  Backing Filesystem: extfs
  Dirs: 66
  Dirperm1 Supported: true
```

图 6-2

Ubuntu 默认 driver 用的是 AUFS，底层文件系统是 extfs，各层数据存放在 /var/lib/docker/aufs。

Redhat/CentOS 的默认 driver 是 Device Mapper，SUSE 则是 Btrfs。

对于某些容器，直接将数据放在由 storage driver 维护的层中是很好的选择，比如那些无状态的应用。无状态意味着容器没有需要持久化的数据，随时可以从镜像直接创建。

比如 busybox，它是一个工具箱，启动 busybox 是为了执行诸如 wget、ping 之类的命令，不需要保存数据供以后使用，使用完直接退出，容器删除时存放在容器层中的工作数据也一起

被删除，这没问题，下次再启动新容器即可。

但对于另一类应用这种方式就不合适了，它们有持久化数据的需求，容器启动时需要加载已有的数据，容器销毁时希望保留产生的新数据，也就是说，这类容器是有状态的。

这就要用到 Docker 的另一种存储机制：Data Volume。

6.2 Data Volume

Data Volume 本质上是 Docker Host 文件系统中的目录或文件，能够直接被 mount 到容器的文件系统中。Data Volume 有以下特点：

- (1) Data Volume 是目录或文件，而非没有格式化的磁盘（块设备）。
- (2) 容器可以读写 volume 中的数据。
- (3) volume 数据可以被永久地保存，即使使用它的容器已经销毁。

好，现在我们有数据层（镜像层和容器层）和 volume 都可以用来存放数据，具体使用的时候要怎样选择呢？考虑下面几个场景：

- (1) Database 软件 vs Database 数据。
- (2) Web 应用 vs 应用产生的日志。
- (3) 数据分析软件 vs input/output 数据。
- (4) Apache Server vs 静态 HTML 文件。

相信大家会做出这样的选择：

- (1) 前者放在数据层中。因为这部分内容是无状态的，应该作为镜像的一部分。
- (2) 后者放在 Data Volume 中。这是需要持久化的数据，并且应该与镜像分开存放。

还有个大家可能会关心的问题：如何设置 volume 的容量？

因为 volume 实际上是 docker host 文件系统的一部分，所以 volume 的容量取决于文件系统当前未使用的空间，目前还没有方法设置 volume 的容量。

在具体的使用上，docker 提供了两种类型的 volume：bind mount 和 docker managed volume。

6.2.1 bind mount

bind mount 是将 host 上已存在的目录或文件 mount 到容器。

例如 docker host 上有目录 \$HOME/htdocs，如图 6-3 所示。

```
root@ubuntu:~#
root@ubuntu:~# cat htdocs/index.html
<html><body><h1>This is a file in host file system !</h1></body></html>
root@ubuntu:~#
```

图 6-3

通过 -v 将其 mount 到 httpd 容器，如图 6-4 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d -p 80:80 -v ~/htdocs:/usr/local/apache2/htdocs httpd
011911ef5f1bdd4378a10cc260f6f36ed87bf824def50c1df7c3f854e67ddb28
root@ubuntu:~#
```

图 6-4

-v 的格式为 <host path>:<container path>。/usr/local/apache2/htdocs 就是 Apache Server 存放静态文件的地方。由于/usr/local/apache2/htdocs 已经存在，原有数据会被隐藏起来，取而代之的是 host \$HOME/htdocs/中的数据，这与 Linux mount 命令的行为是一致的，如图 6-5 所示。

```
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:80
<html><body><h1>This is a file in host file system !</h1></body></html>
root@ubuntu:~#
```

图 6-5

curl 显示当前主页确实是 \$HOME/htdocs/index.html 中的内容。更新一下，看是否能生效，如图 6-6 所示。

```
root@ubuntu:~#
root@ubuntu:~# echo "updated index page!" > ~/htdocs/index.html
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:80
updated index page!
root@ubuntu:~#
```

图 6-6

host 中的修改确实生效了，bind mount 可以让 host 与容器共享数据。这在管理上是非常方便的。

下面我们将容器销毁，看看对 bind mount 有什么影响，如图 6-7 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker stop 011911ef5f1b
011911ef5f1b
root@ubuntu:~# docker rm 011911ef5f1b
011911ef5f1b
root@ubuntu:~# cat ~/htdocs/index.html
updated index page!
root@ubuntu:~#
```

图 6-7

可见，即使容器没有了，bind mount 也还在。这也合理，bind mount 是 host 文件系统中的数据，只是借给容器用用，哪能随便就删了啊。

另外，bind mount 时还可以指定数据的读写权限，默认是可读可写，可指定为只读，如图 6-8 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d -p 80:80 -v ~/htdocs:/usr/local/apache2/htdocs:ro httpd
80dd8cd9a1fd917a0b7de43cd45c7a61ab689720a079d307ef82fdd5812cbe1f
root@ubuntu:~#
root@ubuntu:~# docker exec -it 80dd8cd9a1fd bash
root@80dd8cd9a1fd:/usr/local/apache2#
root@80dd8cd9a1fd:/usr/local/apache2# echo "do some changes" > htdocs/index.html
bash: [htdocs/index.html: Read-only file system]
root@80dd8cd9a1fd:/usr/local/apache2#
```

图 6-8

ro 设置了只读权限，在容器中是无法对 bind mount 数据进行修改的。只有 host 有权修改数据，提高了安全性。

除了 bind mount 目录，还可以单独指定一个文件，如图 6-9 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d -p 80:80 -v ~/htdocs/index.html:/usr/local/apache2/htdocs/new_index.html httpd
b01dfcad74df06e24036e731dfc179eca20a4127aec71fc6ca289d51bce6b
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:80
<html><body><h1>It works!</h1></body></html>
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:80/new_index.html
updated index page!
root@ubuntu:~#
```

图 6-9

使用 bind mount 单个文件的场景是：只需要向容器添加文件，不希望覆盖整个目录。在上面的例子中，我们将 html 文件加到 apache 中，同时也保留了容器原有的数据。

使用单一文件有一点要注意：host 中的源文件必须要存在，不然会当作一个新目录 bind mount 给容器。

mount point 有很多应用场景，比如我们可以将源代码目录 mount 到容器中，在 host 中修改代码就能看到应用的实时效果。再比如将 MySQL 容器的数据放在 bind mount 里，这样 host 可以方便地备份和迁移数据。

bind mount 的使用直观高效，易于理解，但它也有不足的地方：bind mount 需要指定 host 文件系统的特定路径，这就限制了容器的可移植性，当需要将容器迁移到其他 host，而该 host 没有要 mount 的数据或者数据不在相同的路径时，操作会失败。

移植性更好的方式是 docker managed volume。

6.2.2 docker managed volume

docker managed volume 与 bind mount 在使用上的最大区别是不需要指定 mount 源，指明

mount point 就行了。还是以 httpd 容器为例，如图 6-10 所示。

```
root@ubuntu:~# docker run -d -p 80:80 -v /usr/local/apache2/htdocs httpd
21acc2ca0729f092aca82b729cdda425a80b1f2806a3d08c13641e13030ed75
root@ubuntu:~#
```

图 6-10

我们通过-v 告诉 docker 需要一个 data volume，并将其 mount 到 /usr/local/apache2/htdocs。那么这个 data volume 具体在哪儿呢？

这个答案可以在容器的配置信息中找到，执行 docker inspect 命令：

```
docker inspect 21acc2ca072 ... . . .
```

```
"Mounts": [ { "Name": "f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340",
  "Source": "/var/lib/docker/volumes/f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340/_data",
  "Destination": "/usr/local/apache2/htdocs",
  "Driver": "local",
  "Mode": "",
  "RW": true,
  "Propagation": "" } ], ..... docker inspect 的输出很多，我们感兴趣的是 Mounts 这部分，这里会显示容器当前使用的所有 data volume，包括 bind mount 和 docker managed volume。
```

Source 就是该 volume 在 host 上的目录。

原来，每当容器申请 mount docker manged volume 时，docker 都会在 /var/lib/docker/volumes 下生成一个目录（例子中是 /var/lib/docker/volumes/f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340/_data），这个目录就是 mount 源。

下面继续研究这个 volume，看看里面有些什么东西，如图 6-11 所示。

```
root@ubuntu:~# ls -l /var/lib/docker/volumes/f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340/_data
total 4
-rw-r--r-- 1 501 staff 45 Jun 12 2007 index.html
root@ubuntu:~# curl 127.0.0.1:80
<html><body><h1>It works!</h1></body></html>
root@ubuntu:~#
```

图 6-11

volume 的内容跟容器原有 /usr/local/apache2/htdocs 完全一样，这是怎么回事呢？

这是因为：如果 mount point 指向的是已有目录，原有数据会被复制到 volume 中。

但要明确一点：此时的 /usr/local/apache2/htdocs 已经不再是由 storage driver 管理的层数据了，它已经是一个 data volume。我们可以像 bind mount 一样对数据进行操作，例如更新数据，如图 6-12 所示。

```
root@ubuntu:~# echo "update volume from host !" > /var/lib/docker/volumes/f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340/_data/index.html
root@ubuntu:~# curl 127.0.0.1:80
update volume from host !
root@ubuntu:~#
```

图 6-12

简单回顾一下 docker managed volume 的创建过程：

- (1) 容器启动时，简单地告诉 docker“我需要一个 volume 存放数据，帮我 mount 到目录 /abc”。
- (2) docker 在/var/lib/docker/volumes 中生成一个随机目录作为 mount 源。
- (3) 如果/abc 已经存在，则将数据复制到 mount 源。
- (4) 将 volume mount 到/abc。

除了通过 docker inspect 查看 volume，我们也可以用 docker volume 命令，如图 6-13 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker volume ls
DRIVER      VOLUME NAME
local        f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340
root@ubuntu:~#
root@ubuntu:~# docker volume inspect f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340
[
    {
        "Name": "f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/f4a0a1018968f47960efe760829e3c5738c702533d29911b01df9f18babf3340/_data",
        "Labels": null,
        "Scope": "local"
    }
]
root@ubuntu:~#
```

图 6-13

目前，docker volume 只能查看 docker managed volume，还看不到 bind mount；同时也无法知道 volume 对应的容器，这些信息还得靠 docker inspect。

我们已经学习了两种 data volume 的原理和基本使用方法，下面做个对比：

- (1) 相同点：两者都是 host 文件系统中的某个路径。
- (2) 不同点：如表 6-1 所示。

表 6-1 bind mount 和 docker managed volume 的不同点

不同点	bind mount	docker managed volume
volume 位置	可任意指定	/var/lib/docker/volumes/...
对已有 mount point 影响	隐藏并替换为 volume	原有数据复制到 volume
是否支持单个文件	支持	不支持，只能是目录
权限控制	可设置为只读，默认为读写权限	无控制，均为读写权限
移植性	移植性弱，与 host path 绑定	移植性强，无须指定 host 目录

6.3 数据共享

数据共享是 volume 的关键特性，本节我们详细讨论通过 volume 如何在容器与 host 之间、容器与容器之间共享数据。

6.3.1 容器与 host 共享数据

有两种类型的 data volume，它们均可实现在容器与 host 之间共享数据，但方式有所区别。

对于 bind mount 是非常明确的：直接将要共享的目录 mount 到容器。具体请参考前面 httpd 的例子，不再赘述。

docker managed volume 就要麻烦点。由于 volume 位于 host 中的目录，是在容器启动时才生成，所以需要将共享数据复制到 volume 中。请看下面的例子，如图 6-14 所示。

```
root@ubuntu:~#
root@ubuntu:# docker run -d -p 80:80 -v /usr/local/apache2/htdocs httpd
2f7a9edffd15e938bd5fc9778e05916c6e579506d6bc4b581f3de9b9a6aa4651
root@ubuntu:#
root@ubuntu:# docker cp ~/htdocs/index.html 2f7a9edffd15:/usr/local/apache2/htdocs
root@ubuntu:#
root@ubuntu:# curl 127.0.0.1:80
updated index page!
root@ubuntu:#
```

图 6-14

docker cp 可以在容器和 host 之间复制数据，当然我们也可以直接通过 Linux 的 cp 命令复制到 /var/lib/docker/volumes/xxx。

6.3.2 容器之间共享数据

第一种方法是将共享数据放在 bind mount 中，然后将其 mount 到多个容器。还是以 httpd 为例，不过这次的场景复杂些，我们要创建由三个 httpd 容器组成的 Web server 集群，它们使用相同的 html 文件，操作如下：

- (1) 将 \$HOME/htdocs mount 到三个 httpd 容器，如图 6-15 所示。

```
root@ubuntu:~#
root@ubuntu:# docker run --name web1 -d -p 80 -v ~/htdocs:/usr/local/apache2/htdocs httpd
78cd027964b70d62bf6740241f3bcd1bffd93a70250ddb7b4a90dd618a503070
root@ubuntu:#
root@ubuntu:# docker run --name web2 -d -p 80 -v ~/htdocs:/usr/local/apache2/htdocs httpd
51048ffff8a03c4325f4599e4b1f8c742d4f679b8dbe465d551a7355a8a1a877
root@ubuntu:#
root@ubuntu:# docker run --name web3 -d -p 80 -v ~/htdocs:/usr/local/apache2/htdocs httpd
0f326806c4e17455c8261bb2e6e3247dabc5e7040085d53741412c379cc87f0d
root@ubuntu:#
```

图 6-15

(2) 查看当前主页内容, 如图 6-16 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE       COMMAND      CREATED          STATUS          PORTS
0f32680c4e1        httpd      "httpd-foreground"   About a minute ago   Up About a minute
51048ffff8a0        httpd      "httpd-foreground"   2 minutes ago     Up 2 minutes
78c0027964b7        httpd      "httpd-foreground"   2 minutes ago     Up 2 minutes
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:32770
updated index page!
root@ubuntu:~# curl 127.0.0.1:32769
updated index page!
root@ubuntu:~# curl 127.0.0.1:32768
updated index page!
root@ubuntu:~#
```

图 6-16

(3) 修改 volume 中的主页文件, 再次查看并确认所有容器都使用了新的主页, 如图 6-17 所示。

```
root@ubuntu:~#
root@ubuntu:~# echo "This is a new index page for web cluster!" > ~/htdocs/index.html
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:32770
This is a new index page for web cluster!
root@ubuntu:~# curl 127.0.0.1:32769
This is a new index page for web cluster!
root@ubuntu:~# curl 127.0.0.1:32768
This is a new index page for web cluster!
root@ubuntu:~#
```

图 6-17

另一种在容器之间共享数据的方式是使用 volume container。

6.4 volume container

volume container 是专门为其他容器提供 volume 的容器。它提供的卷可以是 bind mount, 也可以是 docker managed volume。下面我们创建一个 volume container, 如图 6-18 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker create --name vc_data \
>           -v ~/htdocs:/usr/local/apache2/htdocs \
>           -v /other/useful/tools \
>           busybox
2f459897d6dbd12d78fb41e6cccbc43e038f551ebdf3e47ed1cdaec83e7af59
root@ubuntu:~#
```

图 6-18

我们将容器命名为 vc_data (vc 是 volume container 的缩写)。注意这里执行的是 docker create 命令, 这是因为 volume container 的作用只是提供数据, 它本身不需要处于运行状态。容

器 mount 了两个 volume:

- (1) bind mount, 存放 Web Server 的静态文件。
- (2) docker managed volume, 存放一些实用工具 (当然现在是空的, 这里只是做个示例)。

通过 docker inspect 可以查看到这两个 volume。

```
docker inspect vc_data
.....
"Mounts": [ { "Source": "/root/htdocs", "Destination": "/usr/local/apache2/htdocs", "Mode": "", "RW": true, "Propagation": "rprivate" }, { "Name": "1b603669398d117e499449862636a56c4f4c804d447c680e7b3ba7c7f5e52205", "Source": "/var/lib/docker/volumes/1b603669398d117e499449862636a56c4f4c804d447c680e7b3ba7c7f5e52205/_data", "Destination": "/other/useful/tools", "Driver": "local", "Mode": "", "RW": true, "Propagation": "" } ], .....
```

其他容器可以通过--volumes-from 使用 vc_data 这个 volume container, 如图 6-19 所示。

```
root@ubuntu:~# docker run --name web1 -d -p 80 --volumes-from vc_data httpd
7c585378abd731150d39c6ce85e346199c9bc3fc3bda8373d38d01287514176b
root@ubuntu:~#
root@ubuntu:~# docker run --name web2 -d -p 80 --volumes-from vc_data httpd
8a40f8b522f20f49e7fa6baa08f920746a242e36d0ec333d7b30bfc746bee563
root@ubuntu:~#
root@ubuntu:~# docker run --name web3 -d -p 80 --volumes-from vc_data httpd
accc23a3b7f22c47093aec79de69db57c23539257554d51df4aea68b3992d162
root@ubuntu:~#
```

图 6-19

三个 httpd 容器都使用了 vc_data, 看看它们现在都有哪些 volume, 以 web1 为例:

```
docker inspect web1.....
"Mounts": [ { "Source": "/root/htdocs", "Destination": "/usr/local/apache2/htdocs", "Mode": "", "RW": true, "Propagation": "rprivate" }, { "Name": "1b603669398d117e499449862636a56c4f4c804d447c680e7b3ba7c7f5e52205", "Source": "/var/lib/docker/volumes/1b603669398d117e499449862636a56c4f4c804d447c680e7b3ba7c7f5e52205/_data", "Destination": "/other/useful/tools", "Driver": "local", "Mode": "", "RW": true, "Propagation": "" } ], .....
```

web1 容器使用的就是 `vc_data` 的 volume，而且连 mount point 都是一样的。验证一下数据共享的效果，如图 6-20 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE       COMMAND     CREATED      STATUS      PORTS     NAMES
cdf2f2c3cf21        httpd      "httpd-foreground"   6 minutes ago   Up 6 minutes   0.0.0.0:32776->80/tcp   web1
391724ee964a        httpd      "httpd-foreground"   6 minutes ago   Up 6 minutes   0.0.0.0:32775->80/tcp   web2
ef2665d36c87        httpd      "httpd-foreground"   6 minutes ago   Up 6 minutes   0.0.0.0:32774->80/tcp   web3
root@ubuntu:~#
root@ubuntu:~# echo "This content is from a volume container!" > ~/htdocs/index.html
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:32776
This content is from a volume container!
root@ubuntu:~# curl 127.0.0.1:32775
This content is from a volume container!
root@ubuntu:~# curl 127.0.0.1:32774
This content is from a volume container!
root@ubuntu:~#
```

图 6-20

可见，三个容器已经成功共享了 volume container 中的 volume。

下面我们讨论一下 volume container 的特点：

(1) 与 bind mount 相比，不必为每一个容器指定 host path，所有 path 都在 volume container 中定义好了，容器只需与 volume container 关联，实现了容器与 host 的解耦。

(2) 使用 volume container 的容器，其 mount point 是一致的，有利于配置的规范和标准化，但也带来一定的局限，使用时需要综合考虑。

6.5 data-packed volume container

上面的例子中 volume container 的数据归根到底还是在 host 里，有没有办法将数据完全放到 volume container 中，同时又能与其他容器共享呢？

当然可以，通常我们称这种容器为 data-packed volume container。其原理是将数据打包到镜像中，然后通过 docker managed volume 共享。

我们用下面的 Dockerfile 构建镜像，如图 6-21 所示。

```
FROM busybox:latest
ADD htdocs /usr/local/apache2/htdocs
VOLUME /usr/local/apache2/htdocs
```

图 6-21

`ADD` 将静态文件添加到容器目录 `/usr/local/apache2/htdocs`。

`VOLUME` 的作用与 `-v` 等效，用来创建 docker managed volume，mount point 为 `/usr/local/apache2/htdocs`，因为这个目录就是 `ADD` 添加的目录，所以会将已有数据复制到 volume 中。

build 新镜像 datapacked，如图 6-22 所示。

```
root@ubuntu:~/data-packed#
root@ubuntu:~/data-packed# docker build -t datapacked .
Sending build context to Docker daemon 4.608 kB
Step 1 : FROM busybox:latest
--> e02e811dd08f
Step 2 : ADD htdocs /usr/local/apache2/htdocs
--> 09f00891de10
Removing intermediate container 896978185e74
Step 3 : VOLUME /usr/local/apache2/htdocs
--> Running in 36fbecca262f
--> dda20a3040c2
Removing intermediate container 36fbecca262f
Successfully built dda20a3040c2
root@ubuntu:~/data-packed#
```

图 6-22

用新镜像创建 data-packed volume container，如图 6-23 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker create --name vc_data datapacked
c399ea731ac3790f45a12f59e52c564449264a8451309367d94050bab2f4cf29
root@ubuntu:~#
```

图 6-23

因为在 Dockerfile 中已经使用了 VOLUME 指令，这里就不需要指定 volume 的 mount point 了。启动 httpd 容器并使用 data-packed volume container，如图 6-24 所示。

```
root@ubuntu:~#
root@ubuntu:~# docker run -d -p 80:80 --volumes-from vc_data httpd
aac24ee880f127148b4bb59685337246022a1c6f65d0537728bb142e21f5da40
root@ubuntu:~#
root@ubuntu:~# curl 127.0.0.1:80
This content is from a data packed volume container!
root@ubuntu:~#
```

图 6-24

容器能够正确读取 volume 中的数据。data-packed volume container 是自包含的，不依赖 host 提供数据，具有很强的移植性，非常适合只使用静态数据的场景，比如应用的配置信息、Web server 的静态文件等。

6.6 Data Volume 生命周期管理

Data Volume 中存放的是重要的应用数据，如何管理 volume 对应用至关重要。前面我们主

要关注的是 volume 的创建、共享和使用，本节将讨论如何备份、恢复、迁移和销毁 volume。

6.6.1 备份

因为 volume 实际上是 host 文件系统中的目录和文件，所以 volume 的备份实际上是对文件系统的备份。

还记得前面我们是如何搭建本地 Registry 的吗？如图 6-25 所示。

```
root@ubuntu:~# docker run -d -p 5000:5000 -v /myregistry:/var/lib/registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
3690ec4760f9: Pull complete
930045f1e8fb: Pull complete
feeda90cbdbc: Pull complete
61f85310d350: Pull complete
b6082c239858: Pull complete
Digest: sha256:1152291c7f93a4ea2ddc95e46d142c31e743b6dd70e194af9e6ebe530f782c17
Status: Downloaded newer image for registry:2
e1c289488c5c7928ef732b349df6bc51b00890f9fe696e6553e0751f90d62437
root@ubuntu:~#
```

图 6-25

所有的本地镜像都保存在 host 的/myregistry 目录中，我们要做的就是定期备份这个目录。

6.6.2 恢复

volume 的恢复也很简单，如果数据损坏了，直接用之前备份的数据复制到/myregistry 就可以了。

6.6.3 迁移

如果我们想使用更新版本的 Registry，这就涉及数据迁移，方法是：

- (1) docker stop 当前 Registry 容器。
- (2) 启动新版本容器并 mount 原有 volume。

```
docker run -d -p 5000:5000 -v /myregistry:/var/lib/registry
registry:latest
```

当然，在启用新容器前要确保新版本的默认数据路径是否发生变化。

6.6.4 销毁

可以删除不再需要的 volume，但一定要确保知道自己正在做什么，volume 删除后数据是找不回来的。

docker 不会销毁 bind mount，删除数据的工作只能由 host 负责。对于 docker managed volume，在执行 docker rm 删除容器时可以带上 -v 参数，docker 会将容器使用到的 volume

一并删除，但前提是没有任何其他容器 mount 该 volume，目的是保护数据，非常合理。

如果删除容器时没有带 -v 呢？这样就会产生孤儿 volume，好在 docker 提供了 volume 子命令可以对 docker managed volume 进行维护。请看下面的例子，如图 6-26 所示。

```
root@ubuntu:~# 
root@ubuntu:~# docker volume ls
DRIVER          VOLUME NAME
root@ubuntu:~#
root@ubuntu:~# docker run --name bbox -v /test/data busybox
root@ubuntu:~#
root@ubuntu:~# docker volume ls
DRIVER          VOLUME NAME
local           8540af1f553d0e71c2de02fbclae436519eee1500f578d1b707a02851d986f0a
root@ubuntu:~#
```

图 6-26

容器 bbox 使用的 docker managed volume 可以通过 docker volume ls 查看到。

删除 bbox，如图 6-27 所示。

```
root@ubuntu:~# 
root@ubuntu:~# docker rm bbox
bbox
root@ubuntu:~#
root@ubuntu:~# docker volume ls
DRIVER          VOLUME NAME
local           8540af1f553d0e71c2de02fbclae436519eee1500f578d1b707a02851d986f0a
root@ubuntu:~#
```

图 6-27

因为没有使用 -v，volume 遗留了下来。对于这样的孤儿 volume，可以用 docker volume rm 删除，如图 6-28 所示。

```
root@ubuntu:~# 
root@ubuntu:~# docker volume rm 8540af1f553d0e71c2de02fbclae436519eee1500f578d1b707a02851d986f0a
8540af1f553d0e71c2de02fbclae436519eee1500f578d1b707a02851d986f0a
root@ubuntu:~#
root@ubuntu:~# docker volume ls
DRIVER          VOLUME NAME
root@ubuntu:~#
```

图 6-28

如果想批量删除孤儿 volume，可以执行：

```
docker volume rm $(docker volume ls -q)
```

6.7 小结

本章我们学习了以下内容：

- (1) Docker 为容器提供了两种存储资源：数据层和 Data Volume。
- (2) 数据层包括镜像层和容器层，由 storage driver 管理。
- (3) Data Volume 有两种类型：bind mount 和 docker managed volume。
- (4) bind mount 可实现容器与 host 之间、容器与容器之间共享数据。
- (5) volume container 是一种具有更好移植性的容器间数据共享方案，特别是 data-packed volume container。
- (6) 最后我们学习了如何备份、恢复、迁移和销毁 Data Volume。

不知大家发现没有，这章我们学习的只是单个 docker host 中的存储方案；而跨主机存储也是一个重要的主题，当然也更复杂，我们会在容器进阶技术章节详细讨论。

第三篇

容器进阶知识

前面我们已经学习了容器的架构、镜像、容器、网络和存储，这些是容器必不可少的组件，属于容器技术的核心内容。

在此基础上，本章将开始讨论容器的进阶知识，包括管理多个 docker host、跨主机的容器网络、监控、日志管理、数据管理等高级主题，如下图所示。



这些是将容器技术真正应用到生产环境必须要考虑的主题，我们将分章节详细讨论。

第 7 章

◀ 多主机管理 ▶

前面我们的实验环境中只有一个 docker host，所有的容器都是运行在这么一个 host 上的。但在真正的环境中会有多个 host，容器在这些 host 中启动、运行、停止和销毁，相关容器会通过网络互相通信，无论它们是否位于相同的 host。

对于这样一个 multi-host 环境，我们将如何高效地进行管理呢？

我们面临的第一个问题是：为所有的 host 安装和配置 Docker。

在前面我们手工安装了第一个 docker host，步骤包括：

- (1) 安装 https CA 证书。
- (2) 添加 GPG key。
- (3) 添加 docker apt 源。
- (4) 安装 docker。

可见步骤还是挺多的，对于多主机环境手工方式效率低且不容易保证一致性，针对这个问题，Docker 给出的解决方案是 Docker Machine。

用 Docker Machine 可以批量安装和配置 docker host，这个 host 可以是本地的虚拟机、物理机，也可以是公有云中的云主机。

Docker Machine 支持在不同的环境下安装配置 docker host，包括：

- (1) 常规 Linux 操作系统。
- (2) 虚拟化平台——VirtualBox、VMWare、Hyper-V 3. OpenStack。
- (3) 公有云——Amazon Web Services、Microsoft Azure、Google Compute Engine、Digital Ocean 等。

Docker Machine 为这些环境起了一个统一的名字：provider。对于某个特定的 provider，Docker Machine 使用相应的 driver 安装和配置 docker host，如图 7-1 所示。

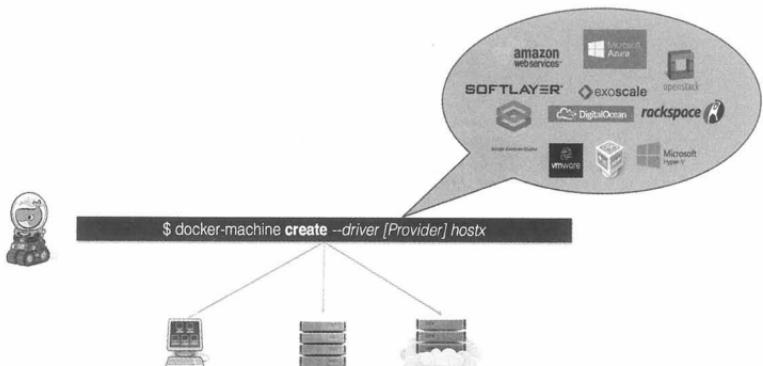


图 7-1

下面我们通过实验来学习 Docker Machine。

7.1 实验环境描述

实验环境中由三个运行 Ubuntu 的 host，如图 7-2 所示。



图 7-2

我们将在 192.168.56.101 上安装 Docker Machine，然后通过 docker-machine 命令在其他两个 host 上部署 docker。

7.2 安装 Docker Machine

官方安装文档在 [https://docs.docker.com/machine/install-machine/。](https://docs.docker.com/machine/install-machine/)

安装方法很简单，执行如下命令：

```
curl -L
https://github.com/docker/machine/releases/download/v0.9.0/docker-
machine-`uname -s`-`uname -m` >/tmp/docker-machine && chmod +x
/tmp/docker-machine && sudo cp /tmp/docker-machine
/usr/local/bin/docker-machine
```

下载的执行文件被放到 /usr/local/bin 中，执行 docker-machine version 验证命令是否可用，如图 7-3 所示。

```
[root@ubuntu ~]#
[root@ubuntu ~]# docker-machine version
docker-machine version 0.9.0-rc2, build 7b19591
[root@ubuntu ~]#
```

图 7-3

注：当你看到这篇文章的时候，Docker Machine 应该有了更新的版本，可参考官方文档进行安装。

为了得到更好的体验，我们可以安装 bash completion script，这样在 bash 中能够通过 Tab 键补全 docker-machine 的子命令和参数。安装方法是从 <https://github.com/docker/machine/tree/master/contrib/completion/bash> 下载 completion script，如图 7-4 所示。

图 7-4

将其放置到 /etc/bash_completion.d 目录下，然后将如下代码添加到 \$HOME/.bashrc：

```
PS1='[\u@\h \w$(_docker_machine_ps1)]\$ '
```

其作用是设置 docker-machine 的命令行提示符，不过要等到部署完其他两个 host 才能看出效果。

Docker Machine 已经就绪，当前环境如图 7-5 所示。



图 7-5

7.3 创建 Machine

对于 Docker Machine 来说，术语 Machine 就是运行 docker daemon 的主机。“创建 Machine”指的就是在 host 上安装和部署 docker。先执行 docker-machine ls 查看一下当前的 machine，如图 7-6 所示。

```
[root@ubuntu ~]#
[root@ubuntu ~]# docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
[root@ubuntu ~]#
```

图 7-6

如我们所料，当前还没有 machine，接下来我们创建第一个 machine：host1 - 192.168.56.104。

创建 machine 要求能够无密码登录远程主机，所以需要先通过如下命令将 ssh key 复制到 192.168.56.104：

```
ssh-copy-id 192.168.56.104
```

一切准备就绪，执行 docker-machine create 命令创建 host1：

```
docker-machine create --driver generic --generic-ip-
address=192.168.56.104 host1
```

因为我们是往普通的 Linux 中部署 docker，所以使用 generic driver，其他 driver 可以参考文档 <https://docs.docker.com/machine/drivers/>。

--generic-ip-address 指定目标系统的 IP，并命名为 host1。命令执行过程如图 7-7 所示。

```
[root@Ubuntu ~]#
[root@Ubuntu ~]# docker-machine create --driver generic --generic-ip-address=192.168.56.104 host1
Running pre-create checks...
Creating machine...
(machine) No SSH key specified. Assuming an existing key at the default location. ①
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)... ②
Installing Docker...
Copying certs to the local machine directory... ③
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon... ④
Checking connection to Docker... ⑤
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-machine env host1
[root@Ubuntu ~]#
```

图 7-7

- ① 通过 ssh 登录到远程主机。
- ② 安装 docker。
- ③ 复制证书。
- ④ 配置 docker daemon。
- ⑤ 启动 docker。

再次执行 docker-machine ls，如图 7-8 所示。

```
[root@Ubuntu ~]#
[root@Ubuntu ~]# docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
host1 - generic Running tcp://192.168.56.104:2376 v1.13.0
[root@Ubuntu ~]#
```

图 7-8

已经能看到 host1 了。我们可以登录到 host1 查看 docker daemon 的具体配置 /etc/systemd/system/docker.service，如图 7-9 所示。

```
[Service]
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H unix:///var/run/docker.sock --storage-driver aufs
--tlsverify --tlscacert /etc/docker/ca.pem --tlscert /etc/docker/server.pem --tlskey /etc/docker/serve
r-key.pem --label provider=generic
MountFlags=slave
LimitNOFILE=1048576
LimitNPROC=1048576
LimitCORE=infinity
Environment=
[Install]
WantedBy=multi-user.target
```

图 7-9

- -H tcp://0.0.0.0:2376：使 docker daemon 接受远程连接。
- --tls*: 对远程连接启用安全认证和加密。

同时我们也看到 hostname 已经设置为 host1，如图 7-10 所示。

```
root@host1:~#
root@host1:~# hostname
host1
```

图 7-10

使用同样的方法创建 host2:

```
docker-machine create --driver generic --generic-ip-address=192.168.56.105 host2
```

创建成功后 docker-machine ls 可以看到 host1 和 host2 都已经就绪, 如图 7-11 所示。

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
host1	-	generic	Running	tcp://192.168.56.104:2376		v1.13.0	
host2	-	generic	Running	tcp://192.168.56.105:2376		v1.13.0	

图 7-11

当前环境如图 7-12 所示。



图 7-12

7.4 管理 Machine

用 docker-machine 创建 machine 的过程很简洁, 非常适合多主机环境。除此之外, Docker Machine 也提供了一些子命令方便对 machine 进行管理。其中最常用的就是无须登录到 machine 就能执行 docker 相关操作。

我们前面学过, 要执行远程 docker 命令需要通过-H 指定目标主机的连接字符串, 比如。

```
docker -H tcp://192.168.56.105:2376 ps
```

Docker Machine 则让这个过程更简单。 docker-machine env host1 显示访问 host1 需要的所有环境变量如图 7-13 所示。

```
[root@ubuntu ~]#
[root@ubuntu ~]# docker-machine env host1
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.56.104:2376"
export DOCKER_CERT_PATH="/root/.docker/machine/machines/host1"
export DOCKER_MACHINE_NAME="host1"
export DOCKER_API_VERSION="1.25"
# Run this command to configure your shell:
# eval $(docker-machine env host1)
[root@ubuntu ~]#
```

图 7-13

根据提示，执行 eval \$(docker-machine env host1)，如图 7-14 所示。

```
[root@ubuntu ~]#
[root@ubuntu ~]# eval $(docker-machine env host1)
[root@ubuntu ~ [host1]]#
[root@ubuntu ~ [host1]]#
```

图 7-14

然后，就可以看到命令行提示符已经变了，其原因是我们在 \$HOME/.bashrc 中配置了 PS1='[\u@\h \W\$(_ docker_machine_ps1)]\\$'，用于显示当前 docker host。

在此状态下执行的所有 docker 命令其效果都相当于在 host1 上执行，例如启动一个 busybox 容器，如图 7-15 所示。

```
[root@ubuntu ~ [host1]]#
[root@ubuntu ~ [host1]]# docker run -itd busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
4b00c1c4050b: Pull complete
Digest: sha256:817a12c32a39bbe394944ba49de563e085f1d3c5266eb8e9723256bc4448680e
Status: Downloaded newer image for busybox:latest
f9feb7ca067d9a70bf6c01eb97c6172534cf411809fc37e1862722fd88a7272
[root@ubuntu ~ [host1]]#
[root@ubuntu ~ [host1]]# docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           PORTS     NAMES
f9feb7ca067d        busybox             "sh"          9 seconds ago   Up 8 seconds   zealous_nightingale
[root@ubuntu ~ [host1]]#
```

图 7-15

执行 eval \$(docker-machine env host2) 切换到 host2，如图 7-16 所示。

```
[root@ubuntu ~ [host1]]#
[root@ubuntu ~ [host1]]# eval $(docker-machine env host2)
[root@ubuntu ~ [host2]]#
[root@ubuntu ~ [host2]]# docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           PORTS     NAMES
[root@ubuntu ~ [host2]]#
```

图 7-16

下面再介绍几个有用的 docker-machine 子命令。

- docker-machine upgrade: 更新 machine 的 docker 到最新版本，可以批量执行，如图 7-17 所示。

```
[root@ubuntu ~]# [root@ubuntu ~]# docker-machine upgrade host1 host2
Waiting for SSH to be available...
Waiting for SSH to be available...
Detecting the provisioner...
Detecting the provisioner...
Upgrading docker...
Upgrading docker...
Restarting docker...
Restarting docker...
[root@ubuntu ~]#
```

图 7-17

- docker-machine config: 查看 machine 的 docker daemon 配置，如图 7-18 所示。

```
[root@ubuntu ~]# [root@ubuntu ~]# docker-machine config host1
--tlsverify
--tlscacert="/root/.docker/machine/certs/ca.pem"
--tlscert="/root/.docker/machine/certs/cert.pem"
--tlskey="/root/.docker/machine/certs/key.pem"
-H=tcp://192.168.56.104:2376
[root@ubuntu ~]#
```

图 7-18

- stop/start/restart: 是对 machine 的操作系统操作，而不是 stop/start/restart docker daemon。
- docker-machine scp: 可以在不同 machine 之间复制文件，比如：

```
docker-machine scp host1:/tmp/a host2:/tmp/b
```

可见，在多主机环境下 Docker Machine 可以大大提高效率，而且操作也很简单，希望大家都能掌握。

第 8 章

容器网络

前面已经学习了 Docker 的几种网络方案：none、host、bridge 和 joined 容器，它们解决了单个 Docker Host 内容器通信的问题。本章的重点则是讨论跨主机容器间通信的方案，如图 8-1 所示。

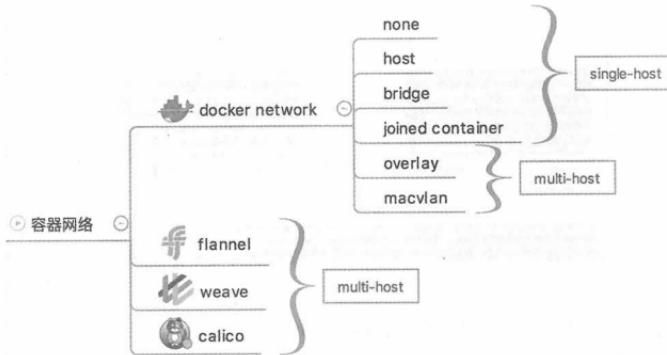


图 8-1

跨主机网络方案包括：（1）docker 原生的 overlay 和 macvlan；（2）第三方方案：常用的包括 flannel、weave 和 calico。

docker 网络是一个非常活跃的技术领域，不断有新的方案开发出来，那么要问个非常重要的问题：如此众多的方案是如何与 docker 集成在一起的？

答案：libnetwork 以及 CNM。

8.1 libnetwork & CNM

libnetwork 是 docker 容器网络库，最核心的内容是其定义的 Container Network Model

(CNM)，这个模型对容器网络进行了抽象，由以下三类组件组成：

1. Sandbox

Sandbox 是容器的网络栈，包含容器的 interface、路由表和 DNS 设置。Linux Network Namespace 是 Sandbox 的标准实现。Sandbox 可以包含来自不同 Network 的 Endpoint。

2. Endpoint

Endpoint 的作用是将 Sandbox 接入 Network。Endpoint 的典型实现是 veth pair，后面我们会举例。一个 Endpoint 只能属于一个网络，也只能属于一个 Sandbox。

3. Network

Network 包含一组 Endpoint，同一 Network 的 Endpoint 可以直接通信。Network 的实现可以是 Linux Bridge、VLAN 等。

下面是 CNM 的示例，如图 8-2 所示。

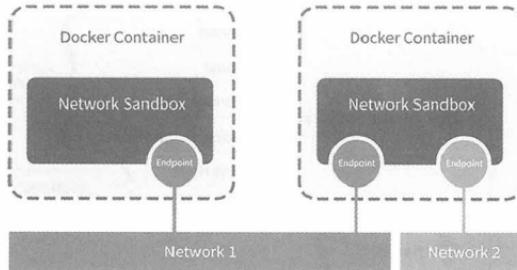


图 8-2

如图所示两个容器，一个容器一个 Sandbox，每个 Sandbox 都有一个 Endpoint 连接到 Network 1，第二个 Sandbox 还有一个 Endpoint 将其接入 Network 2。

libnetwork CNM 定义了 docker 容器的网络模型，按照该模型开发出的 driver 就能与 docker daemon 协同工作，实现容器网络。docker 原生的 driver 包括 none、bridge、overlay 和 macvlan，第三方 driver 包括 flannel、weave、calico 等，如图 8-3 所示。

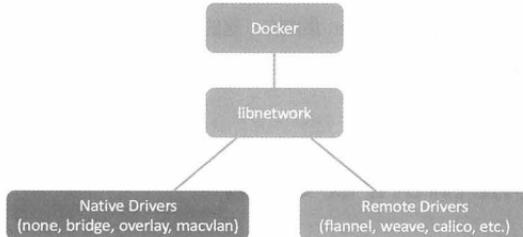


图 8-3

下面我们以 docker bridge driver 为例讨论 libnetwork CNM 是如何被实现的，如图 8-4 所示。

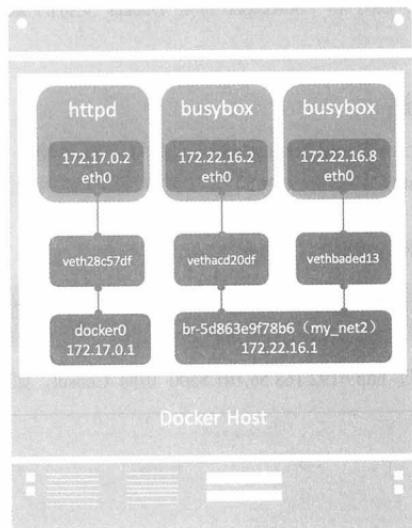


图 8-4

这是前面我们讨论过的一个容器环境：

- (1) 两个 Network：默认网络 bridge 和自定义网络 my_net2。实现方式是 Linux Bridge：docker0 和 br-5d863e9f78b6。
- (2) 三个 Endpoint，由 veth pair 实现，一端（vethxxx）挂在 Linux Bridge 上，另一端（eth0）挂在容器内。
- (3) 三个 Sandbox，由 Network Namespace 实现，每个容器有自己的 Sandbox。

接下来我们将详细讨论各种跨主机网络方案。

8.2 overlay

为支持容器跨主机通信，Docker 提供了 overlay driver，使用户可以创建基于 VxLAN 的 overlay 网络。VxLAN 可将二层数据封装到 UDP 进行传输，VxLAN 提供与 VLAN 相同的以太网二层服务，但是拥有更强的扩展性和灵活性。有关 VxLAN 更详细的内容可参考 CloudMan 写的图书《每天 5 分钟玩转 OpenStack》中的相关章节。

Docker overlay 网络需要一个 key-value 数据库用于保存网络状态信息，包括 Network、Endpoint、IP 等。Consul、Etcd 和 ZooKeeper 都是 Docker 支持的 key-value 软件，我们这里使用 Consul。

8.2.1 实验环境描述

我们会直接使用上一章 docker-machine 创建的实验环境。在 docker 主机 host1（192.168.56.104）和 host2（192.168.56.105）上实践各种跨主机网络方案，在 192.168.56.101 上部署支持的组件，比如 Consul。

最简单的方式是以容器方式运行 Consul：

```
docker run -d -p 8500:8500 -h consul --name consul program/consul -server -bootstrap
```

容器启动后，可以通过 <http://192.168.56.101:8500> 访问 Consul，如图 8-5 所示。



图 8-5

接下来修改 host1 和 host2 的 docker daemon 的配置文件 /etc/systemd/system/docker.service，如图 8-6 所示。

```
[Service]
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H unix:///var/run/docker.sock --storage-driver aufs --tlsverify - -tlscacert /etc/docker/ca.pem --tlscert /etc/docker/server.pem --tlskey /etc/docker/server-key.pem --label provider=generic --cluster-store=consul://192.168.56.101:8500 --cluster-advertise=enp0s8:2376
MountFlags=slave
LimitNOFILE=1048576
```

图 8-6

- `--cluster-store`: 指定 consul 的地址。
- `--cluster-advertise`: 告知 consul 自己的链接地址。

重启 docker daemon。

```
sudo systemctl daemon-reload
sudo systemctl restart docker.service
```

host1 和 host2 将自动注册到 Consul 数据库中，如图 8-7 所示。



图 8-7

准备就绪，实验环境如图 8-8 所示。

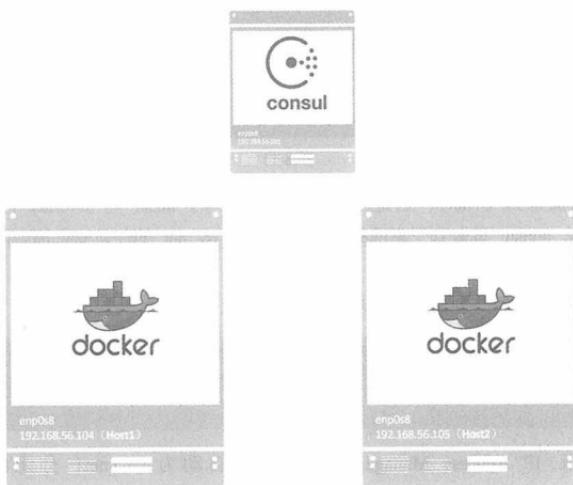


图 8-8

8.2.2 创建 overlay 网络

在 host1 中创建 overlay 网络 ov_net1，如图 8-9 所示。

```
root@host1:~# docker network create -d overlay ov_net1
632a99292e703c7e1197530b298bd200f6a40788342690ccb30ed23d4b135322
root@host1:~#
```

图 8-9

-d overlay 指定 driver 为 overlay。

docker network ls 查看当前网络，如图 8-10 所示。

NETWORK ID	NAME	DRIVER	SCOPE
090699c68ba9	bridge	bridge	local
e5b3c2e071e3	host	host	local
c16af7d078b1	none	null	local
632a99292e70	ov_net1	overlay	global

图 8-10

注意到 ov_net1 的 SCOPE 为 global，而其他网络为 local。在 host2 上查看存在的网络，如图 8-11 所示。

NETWORK ID	NAME	DRIVER	SCOPE
6d917f87a70	bridge	bridge	local
e5b3c2e071e3	host	host	local
c16af7d078b1	none	null	local
632a99292e70	ov_net1	overlay	global

图 8-11

host2 上也能看到 ovnet1。这是因为创建 ovnet1 时 host1 将 overlay 网络信息存入了 consul，host2 从 consul 读取到了新网络的数据。之后 ov_net 的任何变化都会同步到 host1 和 host2。

docker network inspect 查看 ov_net1 的详细信息：

```
root@host1:~# docker network inspect ov_net1 ..... "IPAM": { "Driver": "default", "Options": {}, "Config": [ { "Subnet": "10.0.0.0/24", "Gateway": "10.0.0.1" } ] }, .....
```

IPAM 是指 IP Address Management，docker 自动为 ov_net1 分配的 IP 空间为 10.0.0.0/24。

8.2.3 在 overlay 中运行容器

运行一个 busybox 容器并连接到 ov_net1，如图 8-12 所示。

```
root@host1:~# docker run -itd --name bbox1 -network ov_net1 busybox
100dd1207eb8412a81c980dcda191d00eb4377a7a175a806a26038e8c12259766
root@host1:~#
```

图 8-12

查看容器的网络配置，如图 8-13 所示。

```
root@host1:~# docker exec bbox1 ip r
root@host1:~# default via 172.17.0.1 dev eth1
10.0.0.0/24 dev eth0 src 10.0.0.2
172.17.0.0/16 dev eth1 src 172.17.0.2
root@host1:~#
```

图 8-13

bbox1 有两个网络接口 eth0 和 eth1。eth0 IP 为 10.0.0.2，连接的是 overlay 网络 ov_net1。eth1 IP 172.17.0.2，容器的默认路由是走 eth1，eth1 是哪儿来的呢？

其实，docker 会创建一个 bridge 网络“docker_gwbridge”，为所有连接到 overlay 网络的容器提供访问外网的能力，如图 8-14 所示。

```
root@host1:~# docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
0ae0a1887f87    bridge    bridge      local
d127a32e011    docker_gwbridge bridge      local
e5b3c2e071e3    host      host       local
c16af7d078b1    none      null       local
632a9929e70    ov_net1   overlay    global
root@host1:~#
```

图 8-14

从 docker network inspect docker_gwbridge 输出可确认 docker_gwbridge 的 IP 地址范围是 172.17.0.0/16，当前连接的容器就是 bbox1 (172.17.0.2)。

```
docker network inspect docker_gwbridge [ ..... "IPAM": { "Driver": "bridge", "Default", "Options": null, "Config": [ { "Subnet": "172.17.0.0/16", "Gateway": "172.17.0.1" } ] }, "Internal": false, "Attachable": false, "Containers": [ { "Name": "gateway_100dd1207eb8", "EndpointID": "5077a2bfef80c695661f555412c3679b1a309cbb8a2f1a3247d6b414d35b819", "MacAddress": "02:42:ac:11:00:02", "IPv4Address": "172.17.0.2/16", "IPv6Address": "" } ], .....
```

而且此网络的网关就是网桥 docker_gwbridge 的 IP 172.17.0.1，如图 8-15 所示。

```
root@host1:~# ifconfig docker_gwbridge
docker_gwbridge Link encap:Ethernet HWaddr 02:42:ab:78:c7:3c
          inet addr:172.17.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
                  inet6 addr: fe80::42:abff:fe78:c73c/64 Scope:Link
                        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
```

图 8-15

这样容器 bbox1 就可以通过 docker_gwbridge 访问外网，如图 8-16 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 www.bing.com
PING www.bing.com (202.89.233.103): 56 data bytes
64 bytes from 202.89.233.103: seq=0 ttl=108 time=67.005 ms
64 bytes from 202.89.233.103: seq=1 ttl=108 time=90.828 ms
```

图 8-16

如果外网要访问容器，可通过主机端口映射，比如：

```
docker run -p 80:80 -d --net ov_net1 --name web1 httpd
```

验证完外网的连通性，接下来验证一下 overlay 网络跨主机通信。

8.2.4 overlay 网络连通性

在 host2 中运行容器 bbox2，如图 8-17 所示。

```
root@host2:~#
root@host2:~# docker run -itd --name bbox2 --network ov_net1 busybox
9d346eb33aaa904e1698175f7755ab0c63de17f1fe8d41215d5daa33df31f51
root@host2:~#
root@host2:~# docker exec bbox2 ip r
default via 172.17.0.1 dev eth1
10.0.0.0/24 dev eth0 src [10.0.0.3]
172.17.0.0/16 dev eth1 src 172.17.0.2
root@host2:~#
```

图 8-17

bbox2 IP 为 10.0.0.3，可以直接 ping bbox1，如图 8-18 所示。

```
root@host2:~#
root@host2:~# docker exec bbox2 ping -c 2 bbox1
PING bbox1 (10.0.0.2): 56 data bytes
64 bytes from 10.0.0.2: seq=0 ttl=64 time=0.576 ms
64 bytes from 10.0.0.2: seq=1 ttl=64 time=0.418 ms
```

图 8-18

可见 overlay 网络中的容器可以直接通信，同时 docker 也实现了 DNS 服务。

下面我们讨论一下 overlay 网络的具体实现：docker 会为每个 overlay 网络创建一个独立的 network namespace，其中会有一个 linux bridge br0，endpoint 还是由 veth pair 实现，一端连接到容器中（即 eth0），另一端连接到 namespace 的 br0 上。br0 除了连接所有的 endpoint，还会连接一个 vxlan 设备，用于与其他 host 建立 vxlan tunnel。容器之间的数据就是通过这个 tunnel 通信的。逻辑网络拓扑结构如图 8-19 所示。

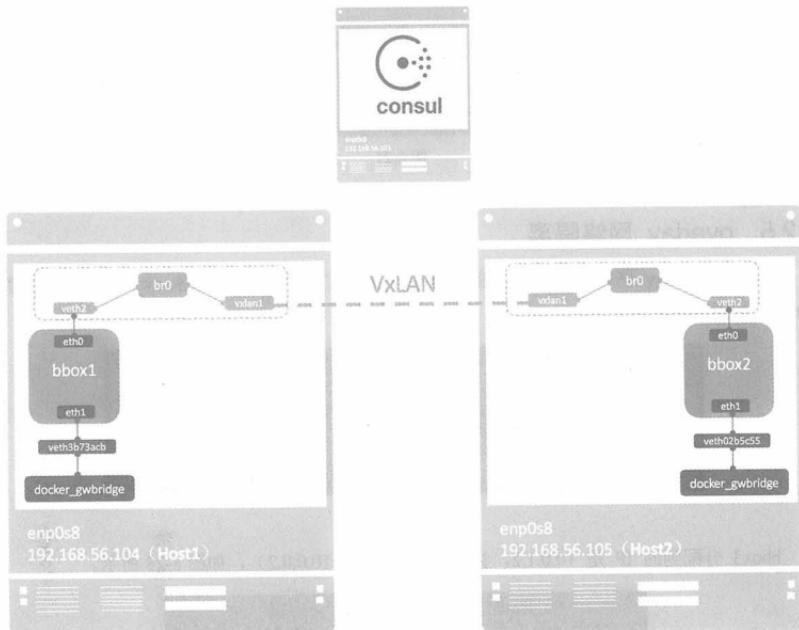


图 8-19

要查看 overlay 网络的 namespace，可以在 host1 和 host2 上执行 ip netns（请确保在此之前执行过 ln -s /var/run/docker/netns /var/run/netns），可以看到两个 host 上有一个相同的 namespace “1-f4af9b33c0”：

ip netns 1-f4af9b33c0 这就是 ov_net1 的 namespace，查看 namespace 中的 br0 上的设备，如图 8-20 所示。

```
root@host1:~#
root@host1:~# ip netns exec 1-f4af9b33c0 brctl show
bridge name      bridge id          STP enabled     interfaces
br0              8000.361b1043527c    no            veth2
                                         vxlan1
root@host1:~#
```

图 8-20

查看 vxlan1 设备的具体配置信息可知此 overlay 使用的 VNI (VxLAN ID) 为 256，如图 8-21 所示。

```
root@ubuntu:~#
root@ubuntu:~# ip netns exec 1-f4af9b33c0 ip -d l show vxlan1
19: vxlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0
    link/ether 36:1b:10:43:52:7c brd ff:ff:ff:ff:ff:ff promiscuity 1
        vxlan id 256 port 0 0 proxy l2miss l3miss ageing 300
root@ubuntu:~#
```

图 8-21

8.2.5 overlay 网络隔离

不同的 overlay 网络是相互隔离的。我们创建第二个 overlay 网络 ov_net2 并运行容器 bbox3，如图 8-22 所示。

```
root@host1:~#
root@host1:~# docker network create -d overlay ov_net2
38e1f217d35f892b5ede21e1b2ef651e5970194be0ff369e15bd6c7b2f7393
root@host1:~#
root@host1:~# docker run -itd --name bbox3 --network ov_net2 busybox
a562de4a7407e90a425a53ad3e79314db0fec80dd428374da8dfbb7935b6b475
root@host1:~#
```

图 8-22

bbox3 分配到的 IP 是 10.0.1.2，尝试 ping bbox1（10.0.0.2），如图 8-23 所示。

```
root@host1:~#
root@host1:~# docker exec -it bbox3 ip r
default via 172.17.0.1 dev eth1
10.0.1.0/24 dev eth0 [src 10.0.1.2]
172.17.0.0/16 dev eth1 src 172.17.0.3
root@host1:~#
root@host1:~# docker exec -it bbox3 ping -c 2 10.0.0.2
PING 10.0.0.2 (10.0.0.2): 56 data bytes

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#
```

图 8-23

ping 失败，可见不同 overlay 网络之间是隔离的。即便是通过 docker_gwbridge 也不能通信，如图 8-24 所示。

```
root@host1:~#
root@host1:~# docker exec -it bbox3 ping -c 2 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#
```

图 8-24

如果要实现 bbox3 与 bbox1 通信，可以将 bbox3 也连接到 ov_net1，如图 8-25 所示。

```
root@host1:~#
root@host1:~# docker network connect ov_net1 bbox3
root@host1:~#
root@host1:~# docker exec -it bbox3 ping bbox1
PING bbox1 (10.0.0.2): 56 data bytes
64 bytes from 10.0.0.2: seq=0 ttl=64 time=0.119 ms
64 bytes from 10.0.0.2: seq=1 ttl=64 time=0.084 ms
```

图 8-25

8.2.6 overlay IPAM

docker 默认为 overlay 网络分配 24 位掩码的子网（10.0.X.0/24），所有主机共享这个 subnet，容器启动时会顺序从此空间分配 IP。当然我们也可以通过 --subnet 指定 IP 空间。

```
docker network create -d overlay --subnet 10.22.1.0/24 ov_net3
```

8.3 macvlan

除了 overlay，docker 还开发了另一个支持跨主机容器网络的 driver：macvlan。

macvlan 本身是 linux kernel 模块，其功能是允许同一个物理网卡配置多个 MAC 地址，即多个 interface，每个 interface 可以配置自己的 IP。macvlan 本质上是一种网卡虚拟化技术，Docker 用 macvlan 实现容器网络就不奇怪了。

macvlan 的最大优点是性能极好，相比其他实现，macvlan 不需要创建 Linux bridge，而是直接通过以太 interface 连接到物理网络。下面我们就来创建一个 macvlan 网络。

8.3.1 准备实验环境

我们会使用 host1 host2 上单独的网卡 enp0s9 创建 macvlan。为保证多个 MAC 地址的网络包都可以从 enp0s9 通过，我们需要打开网卡的模式。

```
ip link set enp0s9 promisc on
```

确保 enp0s9 状态 UP 并且 promisc 模式已经生效，如图 8-26 所示。

```
root@host1:~#
root@host1:~# ip link show enp0s9
4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    link/ether 08:00:27:58:6d:41 brd ff:ff:ff:ff:ff:ff
root@host1:~#
```

图 8-26

因为 host1 和 host2 是 VirtualBox 虚拟机，还需要在网卡配置选项页中设置混杂模式，如图 8-27 所示。

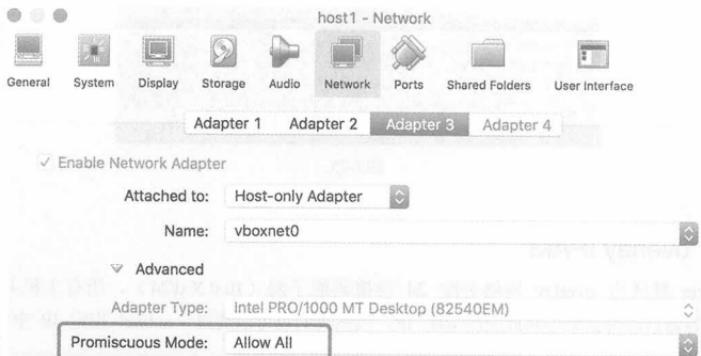


图 8-27

当前实验环境如图 8-28 所示。

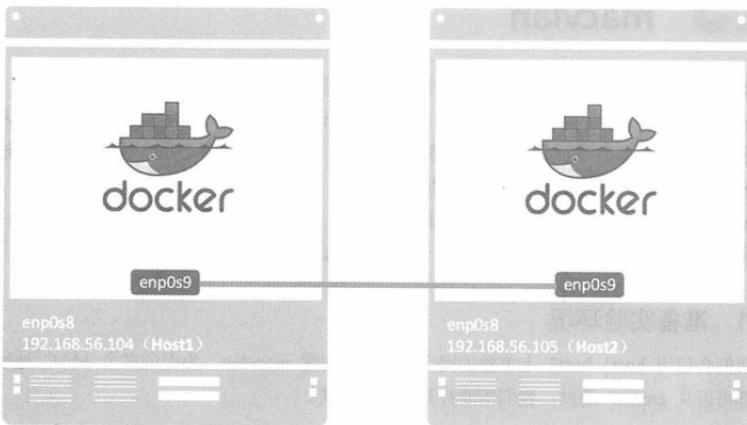


图 8-28

8.3.2 创建 macvlan 网络

在 host1 和 host2 中创建 macvlan 网络 mac_net1，如图 8-29 所示。

```
root@host1:~#
root@host1:~# docker network create -d macvlan \
>           --subnet=172.16.86.0/24 \
>           --gateway=172.16.86.1 \
>           -o parent=enp0s9 mac_net1
977cd5a39cd062ce722e658a32b7d615db581dafd84ecb8c6513adf4ea74c720
root@host1:~#
```

图 8-29

注意：在 host2 中也要执行相同的命令。

① -d macvlan 指定 driver 为 macvlan。

② macvlan 网络是 local 网络，为了保证跨主机能够通信，用户一般需要自己管理 IP subnet。

③ 与其他网络不同，docker 不会为 macvlan 创建网关，这里的网关应该是真实存在的，否则容器无法路由。

④ -o parent 指定使用的网络 interface。

在 host1 中运行容器 bbox1 并连接到 mac_net1，如图 8-30 所示。

```
root@host1:~#
root@host1:~# docker run -itd --name bbox1 --ip=172.16.86.10 --network mac_net1 busybox
ba8475a22005fe687e91a0a13587e01d472b5530319f11cd1b61813dea4908d9
root@host1:~#
```

图 8-30

由于 host1 中的 mac_net1 与 host2 中的 mac_net1 本质上是独立的，为了避免自动分配造成 IP 冲突，我们通过 -ip 指定 bbox1 地址为 172.16.86.10。

在 host2 中运行容器 bbox2，指定 IP 172.16.86.11，如图 8-31 所示。

```
root@host2:~#
root@host2:~# docker run -itd --name bbox2 --ip=172.16.86.11 --network mac_net1 busybox
2388b2777722e8095e0de6135ff5d60254f51a5a5f861610d766ba174d2d5e73
root@host2:~#
```

图 8-31

验证 bbox2 和 bbox1 的连通性，如图 8-32 所示。

```
root@host2:~#
root@host2:~# docker exec bbox2 ping -c 2 172.16.86.10
PING 172.16.86.10 (172.16.86.10): 56 data bytes
64 bytes from 172.16.86.10: seq=0 ttl=64 time=0.746 ms
64 bytes from 172.16.86.10: seq=1 ttl=64 time=0.466 ms
```

图 8-32

bbox2 能够 ping 到 bbox1 的 IP 172.16.86.10，但无法解析 “bbox1” 主机名，如图 8-33 所示。

```
root@host2:~#
root@host2:~# docker exec bbox2 ping -c 2 bbox1
ping: bad address 'bbox1'
root@host2:~#
```

图 8-33

可见 docker 没有为 macvlan 提供 DNS 服务，这点与 overlay 网络是不同的。

8.3.3 macvlan 网络结构分析

macvlan 不依赖 Linux bridge，brctl show 可以确认没有创建新的 bridge，如图 8-34 所示。

```
root@host1:~#
root@host1:~# brctl show
bridge name      bridge id          STP enabled     interfaces
docker0          8000.0242c07ad3ac   no             -
virbr0           8000.52540096f4fa   yes            virbr0-nic
root@host1:~#
```

图 8-34

查看一下容器 bbox1 的网络设备，如图 8-35 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
43: [eth0@if4] <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:10:56:0a brd ff:ff:ff:ff:ff:ff
root@host1:~#
```

图 8-35

除了 lo，容器只有一个 eth0，请注意 eth0 后面的 @if4，这表明该 interface 有一个对应的 interface，其全局的编号为 4。根据 macvlan 的原理，我们有理由猜测这个 interface 就是主机的 enp0s9，确认如图 8-36 所示。

```
root@host1:~#
root@host1:~# ip link show enp0s9
4: enp0s9: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500
    qlen 1000
    link/ether 08:00:27:58:6d:41 brd ff:ff:ff:ff:ff:ff
root@host1:~#
```

图 8-36

可见，容器的 eth0 就是 enp0s9 通过 macvlan 虚拟出来的 interface。容器的 interface 直接与主机的网卡连接，这种方案使得容器无须通过 NAT 和端口映射就能与外网直接通信（只要有网关），在网络上与其他独立主机没有区别。当前网络结构如图 8-37 所示。

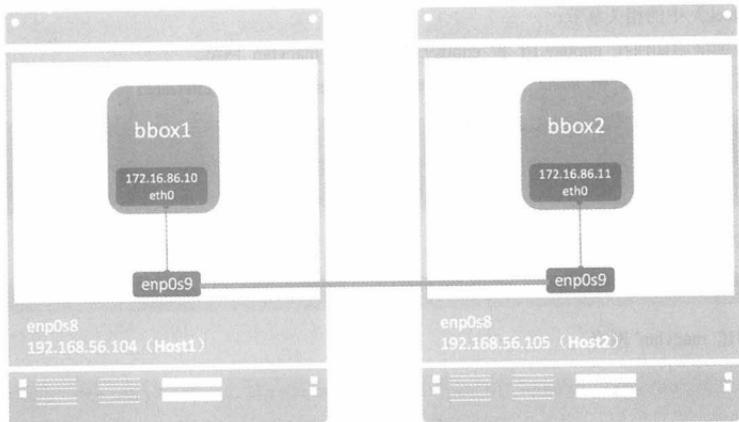


图 8-37

8.3.4 用 sub-interface 实现多 macvlan 网络

macvlan 会独占主机的网卡，也就是说一个网卡只能创建一个 macvlan 网络，否则会报错，如图 8-38 所示。

```
root@host1:~# docker network create -d macvlan -o parent=enp0s9 mac_net2
Error response from daemon: network dm-977cd5a39cd0 is already using parent interface enp0s9
root@host1:~#
```

图 8-38

但主机的网卡数量是有限的，如何支持更多的 macvlan 网络呢？

好在 macvlan 不仅可以连接到 interface（如 enp0s9），也可以连接到 sub-interface（如 enp0s9.xxx）。

VLAN 是现代网络常用的网络虚拟化技术，它可以将物理的二层网络划分成最多 4094 个逻辑网络，这些逻辑网络在二层上是隔离的，每个逻辑网络（即 VLAN）由 VLAN ID 区分，VLAN ID 的取值为 1~4094。

Linux 的网卡也能支持 VLAN（apt-get install vlan），同一个 interface 可以收发多个 VLAN 的数据包，不过前提是需要创建 VLAN 的 sub-interface。

比如希望 enp0s9 同时支持 VLAN10 和 VLAN20，则需创建 sub-interface enp0s9.10 和 enp0s9.20。在交换机上，如果某个 port 只能收发单个 VLAN 的数据，该 port 为 Access 模式，如果支持多 VLAN，则为 Trunk 模式，所以接下来实验的前提是：enp0s9 要接在交换机的 trunk 口上。不过因为我们用的是 VirtualBox 虚拟机，则不需要额外配置了。

如果大家想了解更多 Linux VLAN 实践，可参看 CloudMan 写的图书《每天 5 分钟玩转

OpenStack》中的相关章节。

下面演示如何在 enp0s9.10 和 enp0s9.20 上创建 macvlan 网络。

首先编辑 host1 和 host2 的 /etc/network/interfaces，配置 sub-interface：

```
auto enp0s9 iface enp0s9 inet manual
auto enp0s9.10 iface enp0s9.10 inet manual vlan-raw-device enp0s9
auto enp0s9.20 iface enp0s9.20 inet manual vlan-raw-device enp0s9
```

然后启用 sub-interface：

```
ifup enp0s9.10 ifup enp0s9.20
```

创建 macvlan 网络：

```
docker network create -d macvlan --subnet=172.16.10.0/24 --
gateway=172.16.10.1 -o parent=enp0s9.10 mac_net10 docker network create
-d macvlan --subnet=172.16.20.0/24 --gateway=172.16.20.1 -o
parent=enp0s9.20 mac_net20
```

在 host1 中运行容器：

```
docker run -itd --name bbox1 --ip=172.16.10.10 --network mac_net10
busybox docker run -itd --name bbox2 --ip=172.16.20.10 --network
mac_net20 busybox
```

在 host2 中运行容器：

```
docker run -itd --name bbox3 --ip=172.16.10.11 --network mac_net10
busybox docker run -itd --name bbox4 --ip=172.16.20.11 --network
mac_net20 busybox
```

8.3.5 macvlan 网络间的隔离和连通

下面验证 macvlan 之间的连通性，如图 8-39 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 172.16.10.11
PING 172.16.10.11 (172.16.10.11): 56 data bytes
64 bytes from 172.16.10.11: seq=0 ttl=64 time=0.384 ms
64 bytes from 172.16.10.11: seq=1 ttl=64 time=0.635 ms

root@host1:~#
root@host1:~# docker exec bbox2 ping -c 2 172.16.20.11
PING 172.16.20.11 (172.16.20.11): 56 data bytes
64 bytes from 172.16.20.11: seq=0 ttl=64 time=0.678 ms
64 bytes from 172.16.20.11: seq=1 ttl=64 time=0.477 ms
```

图 8-39

bbox1 能 ping 通 bbox3, bbox2 能 ping 通 bbox4, 同一 macvlan 网络内能通信, 如图 8-40 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 172.16.20.10
PING 172.16.20.10 (172.16.20.10): 56 data bytes

--- 172.16.20.10 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#

root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 172.16.20.11
PING 172.16.20.11 (172.16.20.11): 56 data bytes

--- 172.16.20.11 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#
```

图 8-40

bbox1 无法 ping 通 bbox2 和 bbox4, 不同 macvlan 网络之间不能通信。但准确的说法应该是: 不同 macvlan 网络不能在二层上通信。在三层上可以通过网关将 macvlan 连通, 下面我们就启用网关。

我们会将 Host 192.168.56.101 配置成一个虚拟路由器, 设置网关并转发 VLAN10 和 VLAN20 的流量。当然也可以使用物理路由器达到同样的效果。首先确保操作系统 IP Forwarding 已经启用, 如图 8-41 所示。

```
[root@ubuntu ~]#
[root@ubuntu ~]# sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
[root@ubuntu ~]#
```

图 8-41

输出为 1 则表示启用, 如果为 0 可通过如下命令启用:

```
sysctl -w net.ipv4.ip_forward=1
```

在 /etc/network/interfaces 中配置 vlan sub-interface:

```
auto eth2 iface eth2 inet manual
auto eth2.10 iface eth2.10 inet manual vlan-raw-device eth2
auto eth2.20 iface eth2.20 inet manual vlan-raw-device eth2
```

启用 sub-interface:

```
ifup eth2.10 ifup eth2.20
```

将网关 IP 配置到 sub-interface:

```
ifconfig eth2.10 172.16.10.1 netmask 255.255.255.0 up ifconfig
eth2.20 172.16.20.1 netmask 255.255.255.0 up
```

添加 iptables 规则, 转发不同 VLAN 的数据包。

```

iptables -t nat -A POSTROUTING -o eth2.10 -j MASQUERADE
iptables -t nat -A POSTROUTING -o eth2.20 -j MASQUERADE
iptables -A FORWARD -i eth2.10 -o eth2.20 -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -A FORWARD -i eth2.20 -o eth2.10 -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -A FORWARD -i eth2.10 -o eth2.20 -j ACCEPT
iptables -A FORWARD -i eth2.20 -o eth2.10 -j ACCEPT

```

当前网络拓扑如图 8-42 所示。

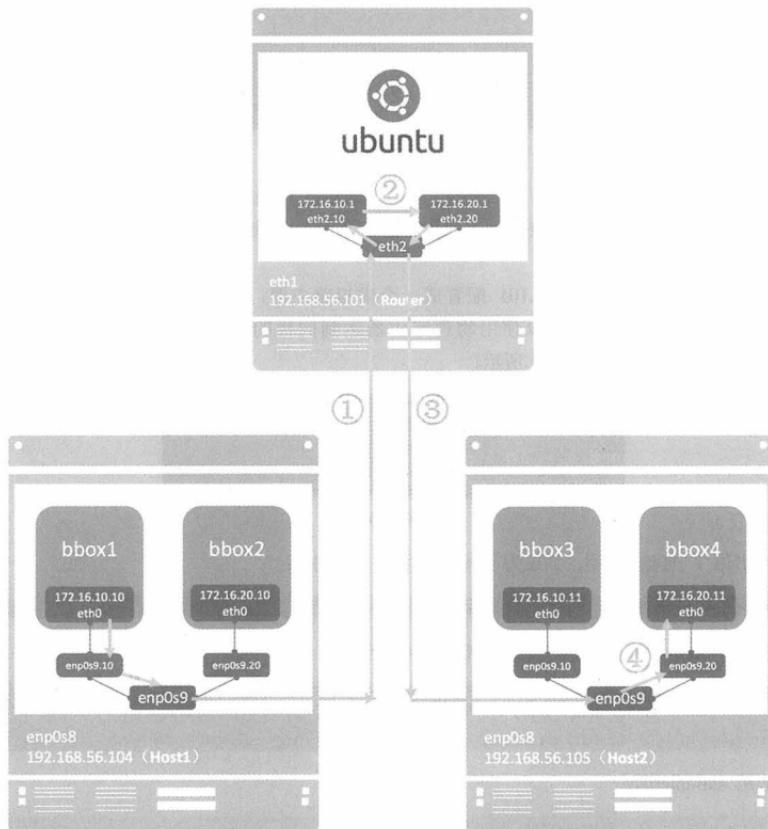


图 8-42

现在 host1 上位于 macnet10 的 bbox1 已经可以与 host2 上位于 macnet20 的 bbox4 通信了，如图 8-43 所示。

```
root@host1:~# docker exec bbox1 ping -c 2 172.16.20.11
root@host1:~# docker exec bbox1 ping -c 2 172.16.20.11
PING 172.16.20.11 (172.16.20.11): 56 data bytes
64 bytes from 172.16.20.11: seq=0 ttl=63 time=0.583 ms
64 bytes from 172.16.20.11: seq=1 ttl=63 time=0.759 ms
```

图 8-43

下面我们分析数据包是如何从 bbox1 (172.16.10.10) 到达 bbox4 (172.16.20.11) 的。整个过程如图 8-44 所示。

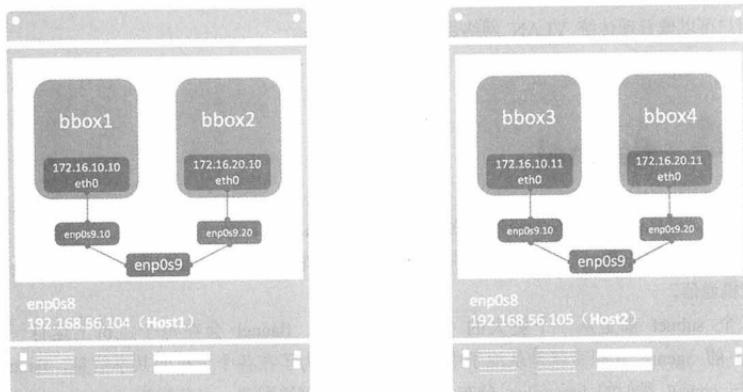
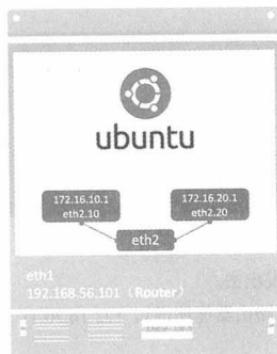


图 8-44

- ① 因为 bbox1 与 bbox4 在不同的 IP 网段，根据 bbox1 的路由表，如图 8-45 所示。

```
root@host1:~# docker exec bbox1 ip route
default via 172.16.10.1 dev eth0
172.16.10.0/24 dev eth0 src 172.16.10.10
root@host1:~#
```

图 8-45

数据包将发送到网关 172.16.10.1。

- ② 路由器从 eth2.10 收到数据包，发现目的地址是 172.16.20.11，查看自己的路由表，如图 8-46 所示。

```
[root@ubuntu ~]# ip route
default via 10.0.2.1 dev eth0
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.4
10.1.24.0/24 dev docker0 proto kernel scope link src 10.1.24.1
172.16.10.0/24 dev eth2.10 proto kernel scope link src 172.16.10.1
172.16.20.0/24 dev eth2.20 proto kernel scope link src 172.16.20.1
192.168.56.0/24 dev eth1 proto kernel scope link src 192.168.56.101
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1
[root@ubuntu ~]#
```

图 8-46

于是将数据包从 eth2.20 转发出去。

- ③ 通过 ARP 记录的信息，路由器能够得知 172.16.20.11 在 host2 上，于是将数据包发送给 host2。

④ host2 根据目的地址和 VLAN 信息将数据包发送给 bbox4。

macvlan 网络的连通和隔离完全依赖 VLAN、IP subnet 和路由，docker 本身不做任何限制，用户可以像管理传统 VLAN 网络那样管理 macvlan。

8.4 flannel

flannel 是 CoreOS 开发的容器网络解决方案。flannel 为每个 host 分配一个 subnet，容器从此 subnet 中分配 IP，这些 IP 可以在 host 间路由，容器间无须 NAT 和 port mapping 就可以跨主机通信。

每个 subnet 都是从一个更大的 IP 池中划分的，flannel 会在每个主机上运行一个叫 flanneld 的 agent，其职责就是从池子中分配 subnet。为了在各个主机间共享信息，flannel 用 etcd（与 consul 类似的 key-value 分布式数据库）存放网络配置、已分配的 subnet、host 的 IP 等信息。

数据包如何在主机间转发是由 backend 实现的。flannel 提供了多种 backend，最常用的有 vxlan 和 host-gw，我们将在本章讨论这两种 backend。其他 backend 请参考 <https://github.com/coreos/flannel>。

接下来我们就开始实践 flannel。

8.4.1 实验环境描述

本章实验环境如图 8-47 所示。

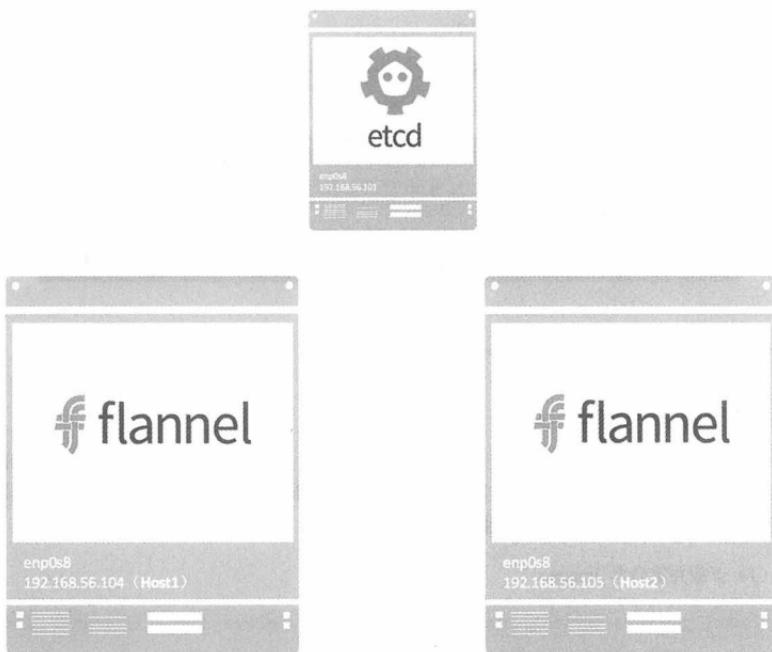


图 8-47

etcd 部署在 192.168.56.101，host1 和 host2 上运行 flanneld，首先安装配置 etcd。

8.4.2 安装配置 etcd

在 192.168.56.101 上运行如下脚本：

```
ETCD_VER=v2.3.7
DOWNLOAD_URL=https://github.com/coreos/etcd/releases/download/curl -L
${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.tar.gz -o
/tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz mkdir -p /tmp/test-etcd && tar
xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz -C /tmp/test-etcd --strip-
components=1 cp /tmp/test-etcd/etcd* /usr/local/bin/
```

该脚本从 github 上下载 etcd 的可执行文件并保存到 /usr/local/bin/，启动 etcd 并打开 2379 监听端口。

```
etcd -listen-client-urls http://192.168.56.101:2379 --advertise-client-urls http://192.168.56.101:2379
```

测试 etcd 是否可用，如图 8-48 所示。

```
etcdctl --endpoints=192.168.56.101:2379 set foo "bar" etcdctl --endpoints=192.168.56.101:2379 get foo
```

```
[root@ubuntu ~]# [root@ubuntu ~]# etcdctl --endpoints=192.168.56.101:2379 set foo "bar" bar [root@ubuntu ~]# etcdctl --endpoints=192.168.56.101:2379 get foo bar [root@ubuntu ~]#
```

图 8-48

可以正常在 etcd 中存取数据了。

8.4.3 build flannel

flannel 没有现成的执行文件可用，必须自己 build，最可靠的方法是在 Docker 容器中 build。不过用于做 build 的 docker 镜像托管在 gcr.io，国内可能无法直接访问，为方便大家，我把它 mirror 到了 docker hub，构建步骤如下：

(1) 下载并重命名 image。

```
docker pull cloudman6/kube-cross:v1.6.2-2 docker tag cloudman6/kube-cross:v1.6.2-2 gcr.io/google_containers/kube-cross:v1.6.2-2
```

(2) 下载 flannel 源码。

```
git clone https://github.com/coreos/flannel.git
```

(3) 开始构建。

```
cd flannel make dist/flanneld-amd64
```

(4) 将 flanneld 执行文件拷贝到 host1 和 host2。

```
scp dist/flanneld-amd64 192.168.56.104:/usr/local/bin/flanneld scp dist/flanneld-amd64 192.168.56.105:/usr/local/bin/flanneld
```

8.4.4 将 flannel 网络的配置信息保存到 etcd

先将配置信息写到文件 flannel-config.json 中，内容为：

```
{ "Network": "10.2.0.0/16", "SubnetLen": 24, "Backend": { "Type": "vxlan" } }
```

- (1) Network 定义该网络的 IP 池为 10.2.0.0/16。
- (2) SubnetLen 指定每个主机分配到的 subnet 大小为 24 位，即 10.2.X.0/24。
- (3) Backend 为 vxlan，即主机间通过 vxlan 通信，后面我们还会讨论 host-gw。

将配置存入 etcd：

```
etcdctl --endpoints=192.168.56.101:2379 set /docker-test/network/config < flannel-config.json
```

/docker-test/network/config 是此 etcd 数据项的 key，其 value 为 flannel-config.json 的内容。key 可以任意指定，这个 key 后面会作为 flanneld 的一个启动参数。执行 etcdctl get 确保设置成功，如图 8-49 所示。

```
[root@ubuntu ~]# [root@ubuntu ~]# etcdctl --endpoints=192.168.56.101:2379 get /docker-test/network/config
{
  "Network": "10.2.0.0/16",
  "SubnetLen": 24,
  "Backend": {
    "Type": "vxlan"
  }
}
[root@ubuntu ~]#
```

图 8-49

8.4.5 启动 flannel

在 host1 和 host2 上执行如下命令：

```
flanneld -etcd-endpoints=http://192.168.56.101:2379 -iface=enp0s8 -etcd-prefix=/docker-test/network
```

- -etcd-endpoints：指定 etcd url。
- -iface：指定主机间数据传输使用的 interface。
- -etcd-prefix：指定 etcd 存放 flannel 网络配置信息的 key。

host1 上输出如图 8-50 所示。

```
root@host1:~# flanneld -etcd-endpoints=http://192.168.56.101:2379 -iface=enp0s8 -etcd-prefix=/docker-test/network
I0203 23:45:17.295427 22950 main.go:133] Installing signal handlers
I0203 23:45:17.296280 22950 manager.go:149] Using interface with name enp0s8 and address 192.168.56.104 ①
I0203 23:45:17.296555 22950 manager.go:166] Defaulting external address to interface address (192.168.56.104)
I0203 23:45:17.317026 22950 local_manager.go:179] Picking subnet in range 10.2.1.0 ... 10.2.255.0 ②
I0203 23:45:17.323441 22950 manager.go:250] Lease acquired: 10.2.40.0/24 ③
I0203 23:45:17.325754 22950 network.go:58] Watching for L3 misses
I0203 23:45:17.326083 22950 network.go:66] Watching for new subnet leases
```

图 8-50

① enp0s8 被选作与外部主机通信的 interface。

② 识别 flannel 网络池 10.2.0.0/16。

③ 分配的 subnet 为 10.2.40.0/24。

flanneld 启动后，host1 内部网络会发生一些变化：

(1) 一个新的 interface flannel.1 被创建，而且配置上 subnet 的第一个 IP 10.2.40.0，如图 8-51 所示。

```
root@host1:~#
root@host1:~# ip addr show flannel.1
51: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
    link/ether 82:25:ef:c1:5c:e0 brd ff:ff:ff:ff:ff:ff
        [inet 10.2.40.0/32] scope global flannel.1
            valid_lft forever preferred_lft forever
        inet6 fe80::8025:efff:fece:5ce0/64 scope link
            valid_lft forever preferred_lft forever
root@host1:~#
```

图 8-51

(2) host1 添加了一条路由：目的地址为 flannel 网络 10.2.0.0/16 的数据包都由 flannel.1 转发，如图 8-52 所示。

```
root@host1:~#
root@host1:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.5
10.1.60.0/24 dev docker0 proto kernel scope link src 10.1.60.1 linkdown
10.2.0.0/16 dev flannel.1
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.104
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
root@host1:~#
```

图 8-52

host2 输出类似，主要区别是 host2 的 subnet 为 10.2.17.0/24，如图 8-53 所示。

```
root@host2:~#
root@host2:~# flanneld -etcd-endpoints=http://192.168.56.101:2379 -iface=enp0s8 -etcd-prefix=/docker-test/network
10204 02:17:22.276391 14648 main.go:133] Installing signal handlers
10204 02:17:22.277278 14648 manager.go:149] Using interface with name enp0s8 and address 192.168.56.105
10204 02:17:22.277652 14648 manager.go:166] Defaulting external address to interface address (192.168.56.105)
10204 02:17:22.302992 14648 local_manager.go:179] Picking subnet in range 10.2.1.0 ... 10.2.255.0
10204 02:17:22.307578 14648 manager.go:250] Lease acquired: [10.2.17.0/24]
10204 02:17:22.308277 14648 network.go:58] Watching for L3 misses
10204 02:17:22.308689 14648 network.go:66] Watching for new subnet leases
```

图 8-53

当前环境网络拓扑如图 8-54 所示。

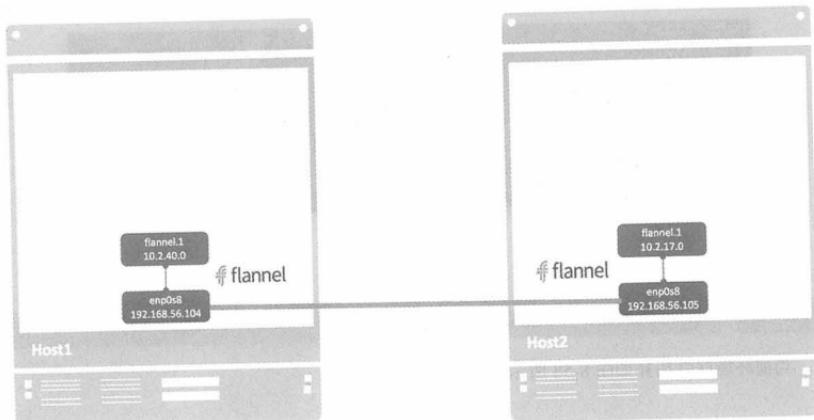


图 8-54

8.4.6 配置 Docker 连接 flannel

编辑 host1 的 Docker 配置文件 /etc/systemd/system/docker.service，设置 --bip 和 --mtu，如图 8-55 所示。

```
[Service]
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H unix:///var/run/docker.sock --storage-driver aufs --tlsverify
--tlscacert /etc/docker/ca.pem --tlscert /etc/docker/server.pem --tlskey /etc/docker/server-key.pem --label provider=generic
--bip=10.2.40.1/24 --mtu=1450
MountFlags=slave
```

图 8-55

这两个参数的值必须与/run/flannel/subnet.env 中 FLANNEL_SUBNET 和 FLANNEL_MTU 一致, 如图 8-56 所示。

```
FLANNEL_NETWORK=10.2.0.0/16
FLANNEL_SUBNET=10.2.40.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

图 8-56

重启 Docker daemon。

```
systemctl daemon-reload systemctl restart docker.service
```

Docker 会将 10.2.40.1 配置到 Linux bridge docker0 上, 并添加 10.2.40.0/24 的路由, 如图 8-57 所示。

```
root@host1:~#
root@host1:# ip r
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.5
10.2.0.0/16 dev flannel.1
10.2.40.0/24 dev docker0 proto kernel scope link src 10.2.40.1
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.104
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
root@host1:~#
```

图 8-57

host2 配置类似:

```
--bip=10.2.17.1/24 --mtu=1450
```

当前环境网络拓扑如图 8-58 所示。

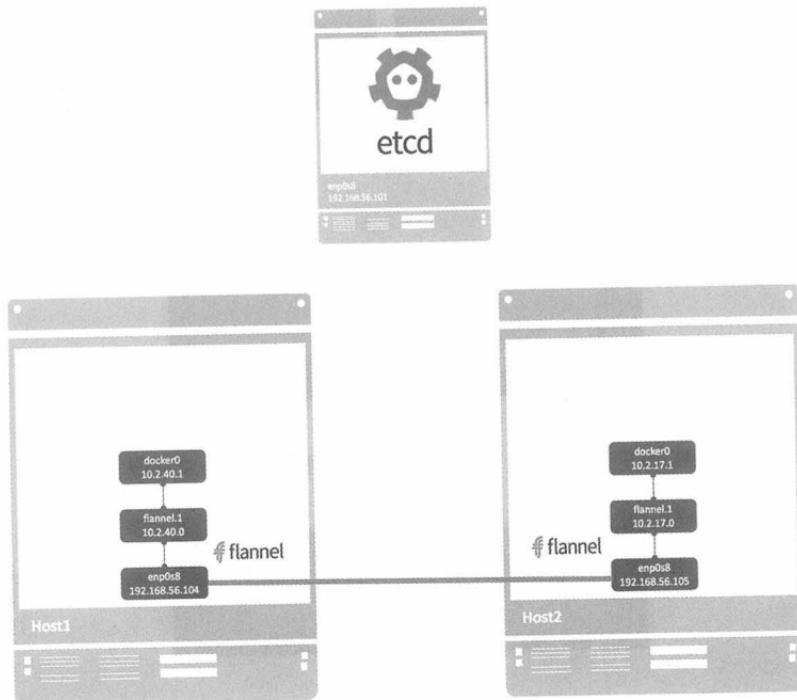


图 8-58

可见：flannel 没有创建新的 docker 网络，而是直接使用默认的 bridge 网络。同一主机的容器通过 docker0 连接，跨主机流量通过 flannel.1 转发。

8.4.7 将容器连接到 flannel 网络

在 host1 中运行容器 bbox1：

```
docker run -itd --name bbox1 busybox
```

在 host2 中运行容器 bbox2：

```
docker run -itd --name bbox2 busybox
```

bbox1 和 bbox2 的 IP 分别为 10.2.40.2 和 10.2.17.2，如图 8-59 所示。

```
root@host1:~# docker exec bbox1 ip r
default via 10.2.40.1 dev eth0
10.2.40.0/24 dev eth0 src 10.2.40.2
root@host1:~#
```

```
root@host2:~# docker exec bbox2 ip r
default via 10.2.17.1 dev eth0
10.2.17.0/24 dev eth0 src 10.2.17.2
root@host2:~#
```

图 8-59

8.4.8 flannel 网络连通性

测试 bbox1 和 bbox2 的连通性，如图 8-60 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 10.2.17.2
PING 10.2.17.2 (10.2.17.2): 56 data bytes
64 bytes from 10.2.17.2: seq=0 ttl=62 time=0.866 ms
64 bytes from 10.2.17.2: seq=1 ttl=62 time=0.665 ms
```

图 8-60

bbox1 能够 ping 到位于不同 subnet 的 bbox2，通过 traceroute 分析一下 bbox1 到 bbox2 的路径，如图 8-61 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 traceroute 10.2.17.2
traceroute to 10.2.17.2 (10.2.17.2), 30 hops max, 46 byte packets
1 10.2.40.1 (10.2.40.1) 0.003 ms 0.004 ms 0.002 ms
2 10.2.17.0 (10.2.17.0) 0.410 ms 0.284 ms 0.434 ms
3 10.2.17.2 (10.2.17.2) 0.466 ms 0.224 ms 0.313 ms
root@host1:~#
```

图 8-61

- (1) bbox1 与 bbox2 不是一个 subnet，数据包发给默认网关 10.2.40.1（docker0）。
- (2) 根据 host1 的路由表，数据包会发给 flannel.1，如图 8-62 所示。

```
root@host1:~#
root@host1:~# ip r
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.5
[10.2.0.0/16 dev flannel.1]
10.2.40.0/24 dev docker0 proto kernel scope link src 10.2.40.1
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.104
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
root@host1:~#
```

图 8-62

- (3) flannel.1 将数据包封装成 VxLAN，通过 enp0s8 发送给 host2。
- (4) host2 收到包解封装，发现数据包目的地址为 10.2.17.2，根据路由表将数据包发送给 flannel.1，并通过 docker0 到达 bbox2。路由表如图 8-63 所示。

```
root@host2:~# ip r
root@host2:~# ip r
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.6
10.2.0.0/16 dev flannel.1
10.2.17.0/24 dev docker0 proto kernel scope link src 10.2.17.1
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.105
192.168.56.0/24 dev enp0s9 proto kernel scope link src 192.168.56.106
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
root@host2:~#
```

图 8-63

数据流向如图 8-64 所示。

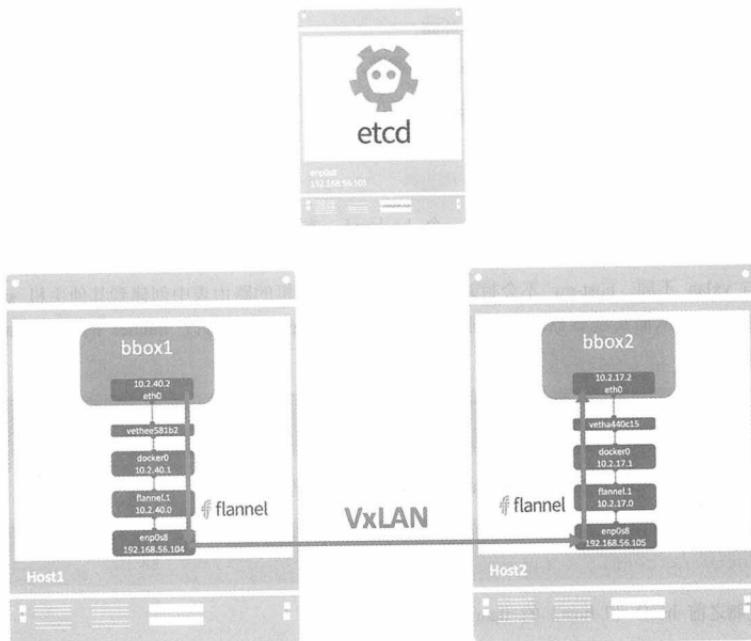


图 8-64

另外，flannel 是没有 DNS 服务的，容器无法通过 hostname 通信，如图 8-65 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ping bbox2
ping: bad address 'bbox2'
root@host1:~#
```

图 8-65

8.4.9 flannel 网络隔离

flannel 为每个主机分配了独立的 subnet，但 flannel.1 将这些 subnet 连接起来了，相互之间可以路由。本质上，flannel 将各主机上相互独立的 docker0 容器网络组成了一个互通的大网络，实现了容器跨主机通信。flannel 没有提供隔离。

8.4.10 flannel 与外网连通性

因为 flannel 网络利用的是默认的 bridge 网络，所以容器与外网的连通方式与 bridge 网络一样，即：

- (1) 容器通过 docker0 NAT 访问外网。
- (2) 通过主机端口映射，外网可以访问容器。

详细讨论可参考前面 bridge 网络相关章节。

8.4.11 host-gw backend

host-gw backend 是 flannel 的另一个 backend，本节会将前面的 vxlan backend 切换成 host-gw。

与 vxlan 不同，host-gw 不会封装数据包，而是在主机的路由表中创建到其他主机 subnet 的路由条目，从而实现容器跨主机通信。要使用 host-gw 首先修改 flannel 的配置 flannel-config.json：

```
{ "Network": "10.2.0.0/16", "SubnetLen": 24, "Backend": { "Type": "host-gw" } }
```

Type 用 host-gw 替换原先的 vxlan。更新 etcd 数据库：

```
etcdctl --endpoints=192.168.56.101:2379 set /docker-test/network/config < flannel-config.json
```

复制之前 host1 和 host2 的 flanneld 进程并重启。

```
flanneld -etcd-endpoints=http://192.168.56.101:2379 -iface=enp0s8 -etcd-prefix=/docker-test/network
```

host1 上 flanneld 启动输出如图 8-66 所示。

```

root@host1:~# flanneld -etcd-endpoints=http://192.168.56.101:2379 -iface=enp0s8 -etcd-prefix=/docker-test/network
10204 22:07:47.524467 3860 main.go:133] Installing signal handlers
10204 22:07:47.525679 3860 manager.go:149] Using interface with name enp0s8 and address 192.168.56.104
10204 22:07:47.526013 3860 manager.go:166] Defaulting external address to interface address (192.168.56.104)
10204 22:07:47.531964 3860 local_manager.go:134] Found lease (10.2.40.0/24) for current IP (192.168.56.104), reusing ①
10204 22:07:47.533991 3860 manager.go:250] Lease acquired: 10.2.40.0/24
10204 22:07:47.535213 3860 network.go:51] Watching for new subnet leases
10204 22:07:47.546024 3860 network.go:83] Subnet added: 10.2.17.0/24 via 192.168.56.105 ②
10204 22:07:47.546339 3860 network.go:86] Ignoring non-host-gw subnet: type=vxlan
10204 22:09:34.834501 3860 network.go:83] Subnet added: 10.2.17.0/24 via 192.168.56.105 ③

```

图 8-66

与之前 vxlan backend 启动时有几点不同：

- ① flanneld 检查到原先已分配的 subnet 10.2.40.0/24，重用之。
- ② flanneld 从 etcd 数据库中检索到 host2 的 subnet 10.2.17.0/24，但因为其 type=vxlan，立即忽略。
- ③ 两分钟后，再次发现 subnet 10.2.17.0/24，将其加到路由表中。这次没有忽略 subnet 的原因是此时我们在 host2 上重启了 flanneld，根据当前 etcd 的配置使用 host-gw backend。

查看 host1 的路由表，增加了一条到 10.2.17.0/24 的路由，网关为 host2 的 IP 192.168.56.105，如图 8-67 所示。

```

root@host1:~#
root@host1:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.5
10.0.2.0/16 dev flannel.1
10.2.17.0/24 via 192.168.56.105 dev enp0s8
10.2.40.0/24 dev docker0 proto kernel scope link src 10.2.40.1
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.104
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
root@host1:#

```

图 8-67

类似的，host2 启动 flanneld 时会重用 subnet 10.2.17.0/24，并将 host1 的 subnet 10.2.40.0/24 添加到路由表中，网关为 host1 IP 192.168.56.104，如图 8-68、图 8-69 所示。

```

root@host2:~#
root@host2:~# flanneld -etcd-endpoints=http://192.168.56.101:2379 -iface=enp0s8 -etcd-prefix=/docker-test/network
10204 22:09:34.900468 19157 main.go:133] Installing signal handlers
10204 22:09:34.901327 19157 manager.go:149] Using interface with name enp0s8 and address 192.168.56.105
10204 22:09:34.901932 19157 manager.go:166] Defaulting external address to interface address (192.168.56.105)
10204 22:09:34.908495 19157 local_manager.go:134] Found lease (10.2.17.0/24) for current IP (192.168.56.105), reusing ①
10204 22:09:34.910173 19157 manager.go:250] Lease acquired: 10.2.17.0/24
10204 22:09:34.911183 19157 network.go:51] Watching for new subnet leases
10204 22:09:34.915920 19157 network.go:83] Subnet added: 10.2.40.0/24 via 192.168.56.104

```

图 8-68

```
root@host2:~# ip r
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.6
10.2.0.0/16 dev flannel.1
10.2.17.0/24 dev docker0 proto kernel scope link src 10.2.17.1
10.2.40.0/24 via 192.168.56.104 dev enp0s8
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.105
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
root@host2:~#
```

图 8-69

从 /run/flannel/subnet.env 可以看到 host-gw 使用的 MTU 为 1500，如图 8-70 所示。

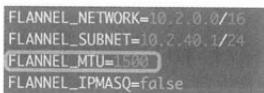


图 8-70

这与 vxlan MTU=1450 不同，所以应该修改 docker 启动参数 --mtu=1500 并重启 docker daemon。

下面对 host-gw 和 vxlan 这两种 backend 做个简单比较。

- (1) host-gw 把每个主机都配置成网关，主机知道其他主机的 subnet 和转发地址。vxlan 则在主机间建立隧道，不同主机的容器都在一个大的网段内（比如 10.2.0.0/16）。
- (2) 虽然 vxlan 与 host-gw 使用不同的机制建立主机之间连接，但对于容器则无须任何改变，bbox1 仍然可以与 bbox2 通信。
- (3) 由于 vxlan 需要对数据进行额外打包和拆包，性能会稍逊于 host-gw。

8.5 weave

weave 是 Weaveworks 开发的容器网络解决方案。weave 创建的虚拟网络可以将部署在多个主机上的容器连接起来。对容器来说，weave 就像一个巨大的以太网交换机，所有容器都被接入这个交换机，容器可以直接通信，无须 NAT 和端口映射。除此之外，weave 的 DNS 模块使容器可以通过 hostname 访问。

8.5.1 实验环境描述

weave 不依赖分布式数据库（例如 etcd 和 consul）交换网络信息，每个主机上只需运行 weave 组件就能建立起跨主机容器网络。我们会在 host1 和 host2 上部署 weave 并实践 weave 的各项特性，如图 8-71 所示。



图 8-71

8.5.2 安装部署 weave

weave 安装非常简单，在 host1 和 host2 上执行如下命令：

```
curl -L git.io/weave -o /usr/local/bin/weave chmod a+x
/usr/local/bin/weave
```

8.5.3 在 host1 中启动 weave

在 host1 中执行 weave launch 命令，启动 weave 相关服务。weave 的所有组件都是以容器方式运行的，weave 会从 docker hub 下载最新的 image 并启动容器，如图 8-72 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
fdecfcf9ab71	weaveworks/plugin:1.8.2	"./home/weave/plugin"	5 minutes ago	Up 5 minutes		weaveplugin
5019f45od13f	weaveworks/weaveexec:1.8.2	"./home/weave/weave..."	5 minutes ago	Up 5 minutes		weaveproxy
705447ce19ce	weaveworks/weave:1.8.2	"./home/weave/weave..."	5 minutes ago	Up 5 minutes		weave

图 8-72

weave 运行了三个容器：

- weave：是主程序，负责建立 weave 网络，收发数据，提供 DNS 服务等。
- Weaveplugin：是 libnetwork CNM driver，实现 Docker 网络。
- Weaveproxy：提供 Docker 命令的代理服务，当用户运行 Docker CLI 创建容器时，它会自动将容器添加到 weave 网络。

weave 会创建一个新的 Docker 网络 weave，如图 8-73 所示。

NETWORK ID	NAME	DRIVER	SCOPE
54336c206f43	bridge	bridge	local
e5b3c2e071e3	host	host	local
c16af7d078b1	none	null	local
f82d9c4a465c	weave	weavemesh	local

图 8-73

driver 为 weavemesh，IP 范围为 10.32.0.0/12。

```
docker network inspect weave ..... "Config": [ { "Subnet": "10.32.0.0/12" } ] .....
```

8.5.4 在 host1 中启动容器

在 host1 中运行容器 bbox1：

```
eval $(weave env) docker run --name bbox1 -itd busybox
```

首先执行 eval \$(weave env) 很重要，其作用是将后续的 docker 命令发给 weave proxy 处理。如果要恢复之前的环境，可执行 eval \$(weave env --restore)。

查看一下当前容器 bbox1 的网络配置，如图 8-74 所示。

```
root@host1:~#
root@host1:~# docker exec -it bbox1 ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
74: eth0@if75: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:0a:02:28:02 brd ff:ff:ff:ff:ff:ff
    inet 10.2.40.2/24 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe02:2802/64 scope link
        valid_lft forever preferred_lft forever
76: ethwe@if77: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1410 qdisc noqueue
    link/ether 5a:05:d4:08:97:2f brd ff:ff:ff:ff:ff:ff
    inet 10.32.0.1/12 scope global ethwe
        valid_lft forever preferred_lft forever
    inet6 fe80::5805:d4ff:fe08:972f/64 scope link
        valid_lft forever preferred_lft forever
root@host1:~#
```

图 8-74

bbox1 有两个网络接口 eth0 和 ethwe，其中 eth0 连接的是默认 bridge 网络，即网桥 docker0。

现在我们重点分析 ethwe。从命名和分配的 IP 10.32.0.1/12 可以猜测 ethwe 与 weave 相关，ethwe@if77 告诉我们与 ethwe 对应的是编号 77 的 interface。从 host1 的 ip link 命令输

出中找到该 interface，如图 8-75 所示。

```
77: vethwep122809@if76: <POINTWISE,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue master weave state UP
    link/ether 0e:26:ca:19:11:40 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

图 8-75

vethwep122809 与 ethwe 是一对 veth pair，而且 vethwep122809 挂在 host1 的 Linux bridge weave 上，如图 8-76 所示。

bridge name	bridge id	STP enabled	interfaces
docker0	8000.0242c07ad3ac	no	veth597ee52
virbr0	8000.52540096f4fa	yes	virbr0-nic
weave	8000.7e3776a4e87	no	vethwe-bridge vethwep122809

图 8-76

除了 vethwep122809，weave 上还挂了一个 vethwe-bridge，这是什么？让我们更深入地分析一下，查看 ip -d link 输出，如图 8-77 所示。

```
62: datapath: <POINTWISE,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1
    link/ether 5a:e7:82:60:4a:7e brd ff:ff:ff:ff:ff:ff promiscuity 1
    openvswitch addrgenmode eui64
64: weave: <POINTWISE,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether 7e:37:76:04:ae:87 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge forward_delay 1500 hello_time 200 max_age 2000 ageing_time 30000 stp_state 0 priority 32768 vlan_filtering 0
65: dummy0: <POINTWISE,NOARP> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 66:bc:ca:27:0d:2f brd ff:ff:ff:ff:ff:ff promiscuity 0
    dummy addrgenmode eui64
67: vethwe-datapath:vethwe-bridge: <POINTWISE,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue master datapath state UP mode DEFAULT
    link/ether 0a:f4:26:f1:1c:71 brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    openvswitch_slave addrgenmode eui64
68: vethwe-bridge@vethwe-datapath: <POINTWISE,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue master weave state UP mode DEFAULT
    link/ether 66:95:bc:9a:85:fe brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    bridge_slave state forwarding priority 32 cost 2 hairpin off guard off root_block off fastleave off learning on flood
69: vxlan-6784: <POINTWISE,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65485 qdisc noqueue master datapath state UNKNOWN mode DEFAULT group default
    link/ether f2:a0:1a:06:00:15 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 0 srport 0 dstport 6784 nolearning ageing 300 udp6zerochecksum
    openvswitch_slave addrgenmode eui64
```

图 8-77

这里出现了多个新 interface：

- ① vethwe-bridge 与 vethwe-datapath 是 veth pair。
- ② vethwe-datapath 的父设备（master）是 datapath。
- ③ datapath 是一个 openvswitch。
- ④ vxlan-6784 是 vxlan interface，其 master 也是 datapath，weave 主机间是通过 VxLAN 通信的。

host1 的网络结构如图 8-78 所示。

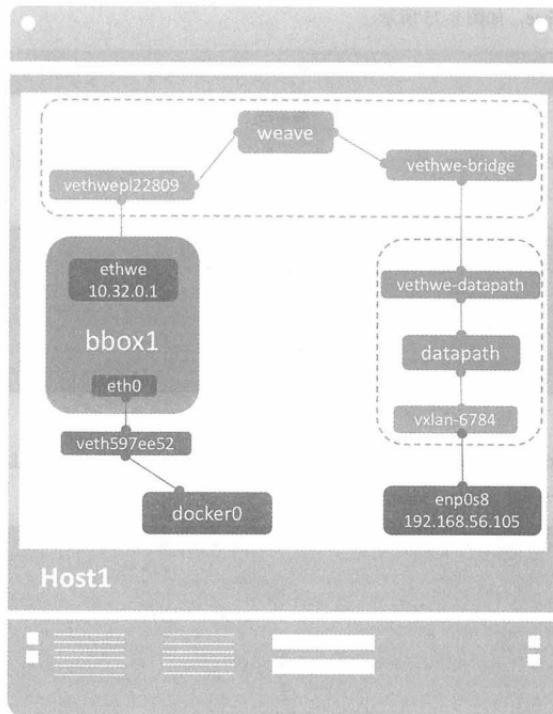


图 8-78

Weave 网络包含两个虚拟交换机：Linux bridge weave 和 Open vSwitch datapath，veth pair vethwe-bridge 和 vethwe-d datapath 将二者连接在一起。weave 和 datapath 分工不同，weave 负责将容器接入 weave 网络，datapath 负责在主机间 VxLAN 隧道中并收发数据。

再运行一个容器 bbox2。

```
docker run --name bbox2 -itd busybox
```

Weave DNS 为容器创建了默认域名 weave.local，bbox1 能够直接通过 hostname 与 bbox2 通信，如图 8-79 所示。

```

root@host1:~#
root@host1:~# docker exec bbox1 hostname
bbox1.weave.local
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 bbox2
PING bbox2 (10.32.0.2): 56 data bytes
64 bytes from 10.32.0.2: seq=0 ttl=64 time=0.041 ms
64 bytes from 10.32.0.2: seq=1 ttl=64 time=0.091 ms

```

图 8-79

当前 host1 网络结构如图 8-80 所示。

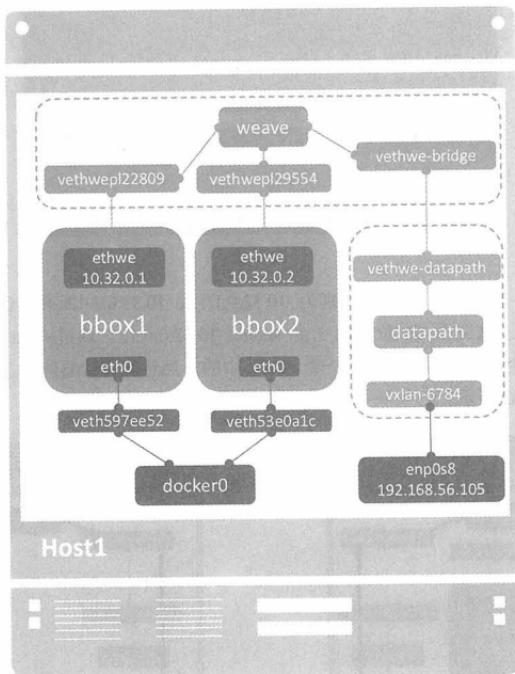


图 8-80

host1 已准备就绪，下面部署 host2。

8.5.5 在 host2 中启动 weave 并运行容器

host2 执行如下命令：

```
weave launch 192.168.56.104
```

这里必须指定 host1 的 IP 192.168.56.104，这样 host1 和 host2 才能加入到同一个 weave 网络。

运行容器 bbox3：

```
eval $(weave env) docker run --name bbox3 -itd busybox
```

8.5.6 weave 网络连通性

bbox3 能够直接 ping bbox1 和 bbox2，如图 8-81 所示。

```
root@host2:~#  
root@host2:~# docker exec bbox3 ping -c 2 bbox1  
PING bbox1 (10.32.0.1): 56 data bytes  
64 bytes from 10.32.0.1: seq=0 ttl=64 time=0.520 ms  
64 bytes from 10.32.0.1: seq=1 ttl=64 time=0.727 ms  
  
root@host2:~#  
root@host2:~# docker exec bbox3 ping -c 2 bbox2  
PING bbox2 (10.32.0.2): 56 data bytes  
64 bytes from 10.32.0.2: seq=0 ttl=64 time=18.828 ms  
64 bytes from 10.32.0.2: seq=1 ttl=64 time=0.614 ms
```

图 8-81

bbox1、bbox2 和 bbox3 的 IP 分别为 10.32.0.1/12、10.32.0.2/12 和 10.44.0.0/12，注意掩码为 12 位，实际上这三个 IP 位于同一个 subnet 10.32.0.0/12。通过 host1 和 host2 之间的 VxLAN 隧道，三个容器逻辑上是在同一个 LAN 中的，当然能直接通信了。bbox3 ping bbox1 的数据流向如图 8-82 所示。

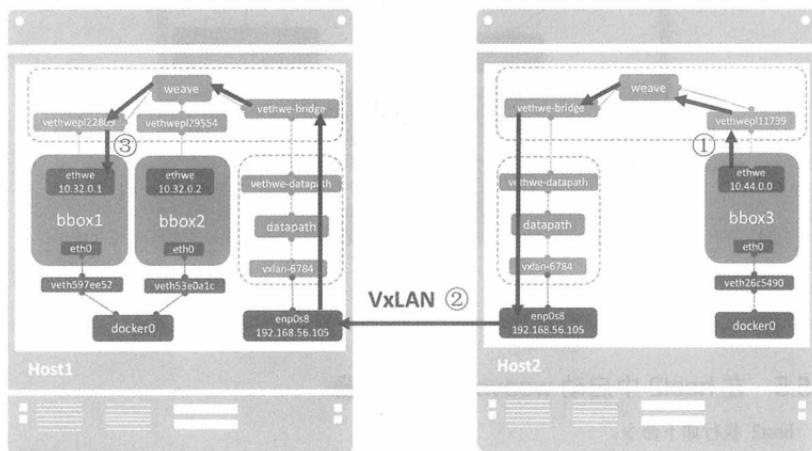


图 8-82

① 数据包目的地址为 10.32.0.1，根据 bbox3 的路由表，数据从 ethwe 发送出去，如图 8-83 所示。

```
root@host2:~#
root@host2:~# docker exec bbox3 ip route
default via 10.2.17.1 dev eth0
10.2.17.0/24 dev eth0 src 10.2.17.2
10.32.0.0/12 dev ethwe src 10.44.0.0
224.0.0.0/4 dev ethwe
root@host2:~#
```

图 8-83

② host2 weave 查询到目的地主机，将数据通过 VxLAN 发送给 host1。

③ host1 weave 接收到数据，根据目的 IP 将数据转发给 bbox1。

8.5.7 weave 网络隔离

默认配置下，weave 使用一个大 subnet（例如 10.32.0.0/12），所有主机的容器都从这个地址空间中分配 IP，因为同属一个 subnet，容器可以直接通信。如果要实现网络隔离，可以通过环境变量 WEAVE_CIDR 为容器分配不同 subnet 的 IP，举例如图 8-84 所示。

```
root@host1:~#
root@host1:~# docker run -e WEAVE_CIDR=net:10.32.2.0/24 -ti busybox
/ #
/ # ip r
default via 10.2.40.1 dev eth0
10.2.40.0/24 dev eth0 src 10.2.40.4
10.32.2.0/24 dev ethwe src 10.32.2.2
224.0.0.0/4 dev ethwe
/ #
/ # ping -c 2 bbox1
PING bbox1 (10.32.0.1): 56 data bytes

--- bbox1 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
/ #
```

图 8-84

这里 WEAVE_CIDR=net:10.32.2.0/24 的作用是使容器分配到 IP 10.32.2.2。由于 10.32.0.0/12 与 10.32.2.0/24 位于不同的 subnet，所以无法 ping 到 bbox1。除了 subnet，我们还可以直接为容器分配特定的 IP，如图 8-85 所示。

```

root@host1:~#
root@host1:~# docker run -e WEAVE_CIDR=ip:10.32.6.6/24 -ti busybox
/ #
/ # ip r
default via 10.2.40.1 dev eth0
10.2.40.0/24 dev eth0 src 10.2.40.4
10.32.6.0/24 dev ethwe src 10.32.6.6
224.0.0.0/4 dev ethwe
/ #
/ # ping -c 2 bbox1
PING bbox1 (10.32.0.1): 56 data bytes

--- bbox1 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
/ #

```

图 8-85

8.5.8 weave 与外网的连通性

weave 是一个私有的 VxLAN 网络，默认与外部网络隔离。外部网络如何才能访问到 weave 中的容器呢？

答案是：

- (1) 首先将主机加入到 weave 网络。
- (2) 然后把主机当作访问 weave 网络的网关。

要将主机加入到 weave，执行 weave expose，如图 8-86 所示。

```

root@host1:~#
root@host1:~# weave expose
10.32.0.3
root@host1:~#

```

图 8-86

这个 IP 为 10.32.0.3，会被配置到 host1 的 weave 网桥上，如图 8-87 所示。

```

root@host1:~#
root@host1:~# ip addr show weave
64: [weave] <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue state UP
    link/ether 7e:37:76:a4:ae:87 brd ff:ff:ff:ff:ff:ff
    inet [10.32.0.3/12] scope global weave
        valid_lft forever preferred_lft forever
    inet6 fe80::7e37:76ff:fea4:ae87/64 scope link
        valid_lft forever preferred_lft forever
root@host1:~#

```

图 8-87

这是个精妙的设计，让我们再看看下面 host1 的网络结构，如图 8-88 所示。

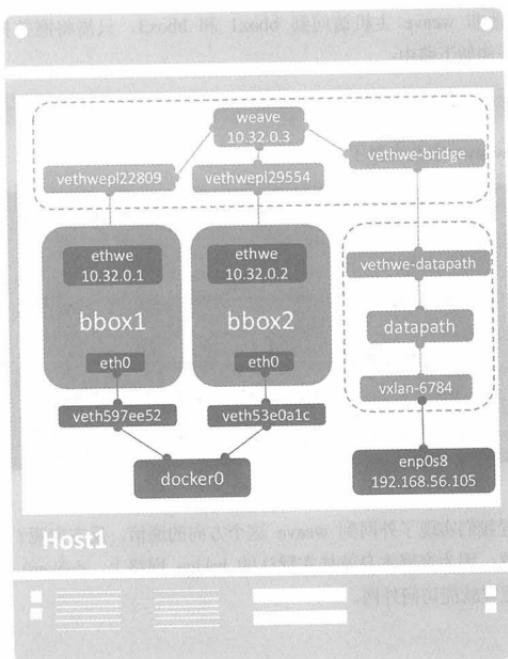


图 8-88

weave 网桥位于 root namespace，它负责将容器接入 weave 网络。给 weave 配置同一 subnet 的 IP，其本质就是将 host1 接入 weave 网络。host1 现在已经可以直接与同一 weave 网络中的容器通信了，无论容器是否位于 host1。

ping 同一主机的 bbox1，如图 8-89 所示。

```
root@host1:~#
root@host1:~# ping -c 2 10.32.0.1
PING 10.32.0.1 (10.32.0.1) 56(84) bytes of data.
64 bytes from 10.32.0.1: icmp_seq=1 ttl=64 time=0.068 ms
64 bytes from 10.32.0.1: icmp_seq=2 ttl=64 time=0.084 ms
```

图 8-89

ping host2 上的 bbox3，如图 8-90 所示。

```
root@host1:~#
root@host1:~# ping -c 2 10.44.0.0
PING 10.44.0.0 (10.44.0.0) 56(84) bytes of data.
64 bytes from 10.44.0.0: icmp_seq=1 ttl=64 time=0.579 ms
64 bytes from 10.44.0.0: icmp_seq=2 ttl=64 time=0.562 ms
```

图 8-90

接下来要让其他非 weave 主机访问到 bbox1 和 bbox3，只需将网关指向 host1。例如在 192.168.56.101 上添加如下路由：

```
ip route add 10.32.0.0/12 via 192.168.56.104
```

能够 ping 到 weave 中的容器了，如图 8-91 所示。

```
[root@ubuntu ~]# ip route
default via 10.0.2.1 dev eth0
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.4
10.1.24.0/24 dev docker0 proto kernel scope link src 10.1.24.1
10.32.0.0/12 via 192.168.56.104 dev eth1
192.168.56.0/24 dev eth1 proto kernel scope link src 192.168.56.101
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1
[root@ubuntu ~]#
[root@ubuntu ~]# ping 10.44.0.0
PING 10.44.0.0 (10.44.0.0) 56(84) bytes of data.
64 bytes from 10.44.0.0: icmp_seq=1 ttl=63 time=0.774 ms
64 bytes from 10.44.0.0: icmp_seq=2 ttl=63 time=0.774 ms
```

图 8-91

通过上面的配置我们实现了外网到 weave 这个方向的通信，反方向呢？

其实答案很简单：因为容器本身就挂在默认的 bridge 网络上，docker0 已经实现了 NAT，所以容器无须额外配置就能访问外网。

8.5.9 IPAM

10.32.0.0/12 是 weave 网络使用的默认 subnet，如果此地址空间与现有 IP 冲突，可以通过 `--ipalloc-range` 分配特定的 subnet。

```
weave launch --ipalloc-range 10.2.0.0/16
```

不过请确保所有 host 都使用相同的 subnet。

8.6 calico

Calico 是一个纯三层的虚拟网络方案，Calico 为每个容器分配一个 IP，每个 host 都是 router，把不同 host 的容器连接起来。与 VxLAN 不同的是，Calico 不对数据包做额外封装，不需要 NAT 和端口映射，扩展性和性能都很好。

与其他容器网络方案相比，Calico 还有一大优势：network policy。用户可以动态定义 ACL 规则，控制进出容器的数据包，实现业务需求。

8.6.1 实验环境描述

Calico 依赖 etcd 在不同主机间共享和交换信息，存储 Calico 网络状态。host 192.168.56.101 负责运行 etcd。

Calico 网络中的每个主机都需要运行 Calico 组件，实现容器 interface 管理、动态路由、动态 ACL、报告状态等。

实验环境如图 8-92 所示。



图 8-92

首先启动 etcd。

8.6.2 启动 etcd

在 host 192.168.56.101 上运行如下命令启动 etcd：

```
etcd -listen-client-urls http://192.168.56.101:2379 --advertise-
client-urls http://192.168.56.101:2379
```

etcd 安装配置详细方法请参考 flannel 章节。

修改 host1 和 host2 Docker daemon 配置文件 /etc/systemd/system/docker.service，连接 etcd，如图 8-93 所示。

```
--cluster-store=etcd://192.168.56.101:2379
```

```
[Service]
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H unix:///var/run/docker.sock --storage-driver aufs
--tlsverify --tlscacert /etc/docker/ca.pem --tlscert /etc/docker/server.pem --tlskey /etc/docker/ser
ver-key.pem --label provider=generic --cluster-store=etcd://192.168.56.101:2379
MountFlags=slave
```

图 8-93

重启 Docker daemon。

```
systemctl daemon-reload systemctl restart docker.service
```

8.6.3 部署 calico

下载 calicoctl:

```
wget -O /usr/local/bin/calicoctl
https://github.com/projectcalico/calicoctl/releases/download/v1.0.2/calicoctl
chmod +x calicoctl
```

在 host1 和 host2 上启动 calico:

```
calicoctl node run
```

启动过程如图 8-94 所示。

```
root@host1:~#
root@host1:~# calicoctl node run
Running command to load modules: modprobe -a xt_set_ip6_tables
Enabling IPv4 forwarding ①
Enabling IPv6 forwarding
Increasing conntrack limit
Removing old calico-node container (if running).
Running the following command to start calico-node: ②
docker run --net=host --privileged --name=calico-node -d --restart=always -e NODENAME=host1 -e CALICO_NETWORKING_BACKEND=bird -e NO_DEFAULT_POOLS= -e CALICO_LIBNETWORK_ENABLED=true -e CALICO_LIBNETWORK_IFPREFIX=cali -e ETCD_ENDPOINTS=http://192.168.56.101:2379 -e ETCD_AUTHORITY= -e ETCD_SCHEME= -v /var/run/calico:/var/run/calico -v /lib/modules:/lib/modules -v /var/log/calico:/var/log/calico -v /run/docker/plugins:/run/docker/plugins -v /var/run/docker.sock:/var/run/docker.sock calico/node:v1.0.1

Image may take a short time to download if it is not available locally.
Container started, checking progress logs.
Waiting for etcd connection... ③
Using configured IPv4 address: 192.168.56.104
No IPv6 address configured
Using global AS number
Calico node name: host1
CALICO_LIBNETWORK_ENABLED is true - start libnetwork service
Calico node started successfully ④
root@host1:~#
```

图 8-94

- ① 设置主机网络，例如 enable IP forwarding。
- ② 下载并启动 calico-node 容器，calico 会以容器的形式运行（与 weave 类似），如图 8-95 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
9e315d3cdaf5	calico/node:v1.0.1	"start_runit"	7 seconds ago	Up 7 seconds		calico-node

图 8-95

- ③ 连接 etcd。
- ④ calico 启动成功。

8.6.4 创建 calico 网络

在 host1 或 host2 上执行如下命令创建 calico 网络 cal_ent1：

```
root@host1:~# docker network create --driver calico --ipam-driver calico-ipam
cal_net1
```

- --driver calico：指定使用 calico 的 libnetwork CNM driver。
- --ipam-driver calico-ipam：指定使用 calico 的 IPAM driver 管理 IP。

calico 为 global 网络，etcd 会将 cal_net 同步到所有主机，如图 8-96 所示。

NETWORK ID	NAME	DRIVER	SCOPE
c609e2f471b8	bridge	bridge	local
f54f08f7eb1a	cal_net1	calico	global
e5b3c2e071e3	host	host	local
c16af7d078b1	none	null	local

图 8-96

8.6.5 在 calico 中运行容器

在 host1 中运行容器 bbox1 并连接到 cal_net1：

```
root@host1:~# docker container run --net cal_net1 --name bbox1 -tid busybox
```

查看 bbox1 的网络配置，如图 8-97 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
10: cali0@if11 <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.119.2/32 scope global cali0
        valid_lft forever preferred_lft forever
    inet6 fe80::ecce:efff%cali0/64 scope link
        valid_lft forever preferred_lft forever
root@host1:~#
```

图 8-97

cali0 是 calico interface，分配的 IP 为 192.168.119.2。cali0 对应 host1 编号 11 的 interface cali5f744ac07f0，如图 8-98 所示。

```
11: cali5f744ac07f0@if10 <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 46:a6:82:93:74:1d brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

图 8-98

host1 将作为 router 负责转发目的地址为 bbox1 的数据包，如图 8-99 所示。

```
root@host1:~#
root@host1:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.5
10.2.79.0/24 dev docker0 proto kernel scope link src 10.2.79.1 linkdown
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.104
blackhole 192.168.119.0/26 proto bird
192.168.119.2 dev cali5f744ac07f0 scope link
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
```

图 8-99

所有发送到 bbox1 的数据都会发给 cali5f744ac07f0，因为 cali5f744ac07f0 与 cali0 是一对 veth pair，bbox1 能够接收到数据。

host1 网络结构如图 8-100 所示。

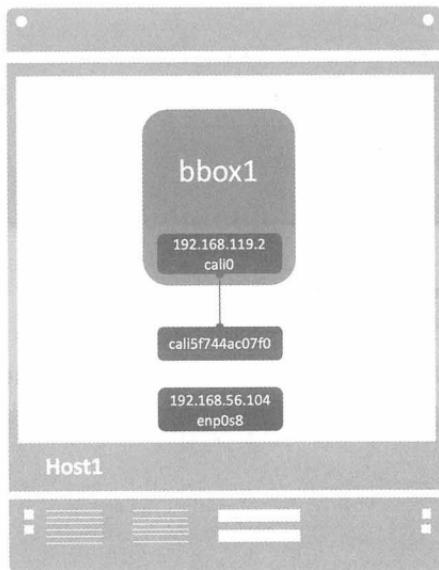


图 8-100

接下来我们在 host2 中运行容器 bbox2，也连接到 cal_net1：

```
docker container run --net cal_net1 --name bbox2 -tid busybox
```

IP 为 192.168.183.65，如图 8-101 所示。

```
root@host2:~#
root@host2:~# docker exec bbox2 ip address show cali0
10: cali0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
        inet 192.168.183.65/32 brd 192.168.183.65 scope global cali0
            valid_lft forever preferred_lft forever
        inet6 fe80::ecee:effff:feee:64/64 brd fe80::ff:feee:64 scope link
            valid_lft forever preferred_lft forever
root@host2:~#
```

图 8-101

host2 添加了两条路由，如图 8-102 所示。

```
root@host2:~# ip route
root@host2:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.6
10.2.17.0/24 dev docker0 proto kernel scope link src 10.2.17.1 linkdown
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.105
192.168.119.0/26 via 192.168.56.104 dev enp0s8 proto bird
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
blackhole 192.168.183.64/26 proto bird
192.168.183.65 dev cali666a18df71 scope link
root@host2:~#
```

图 8-102

(1) 目的地址为 host1 容器 subnet 192.168.119.0/26 的路由。

(2) 目的地址为本地 bbox2 容器 192.168.183.65 的路由。

同样的, host1 也自动添加到了 192.168.183.64/26 的路由, 如图 8-103 所示。

```
root@host1:~#
root@host1:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.5
10.2.79.0/24 dev docker0 proto kernel scope link src 10.2.79.1 linkdown
192.168.56.0/24 dev enp0s8 proto kernel scope link src 192.168.56.104
blackhole 192.168.119.0/26 proto bird
192.168.119.2 dev cali5f744ac07f0 scope link
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
192.168.183.64/26 via 192.168.56.105 dev enp0s8 proto bird
root@host1:~#
```

图 8-103

8.6.6 calico 默认连通性

测试一下 bbox1 与 bbox2 的连通性, 如图 8-104 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 bbox2
PING bbox2 (192.168.183.65): 56 data bytes
64 bytes from 192.168.183.65: seq=0 ttl=62 time=0.367 ms
64 bytes from 192.168.183.65: seq=1 ttl=62 time=0.446 ms
```

图 8-104

ping 成功, 数据包流向如图 8-105 所示。

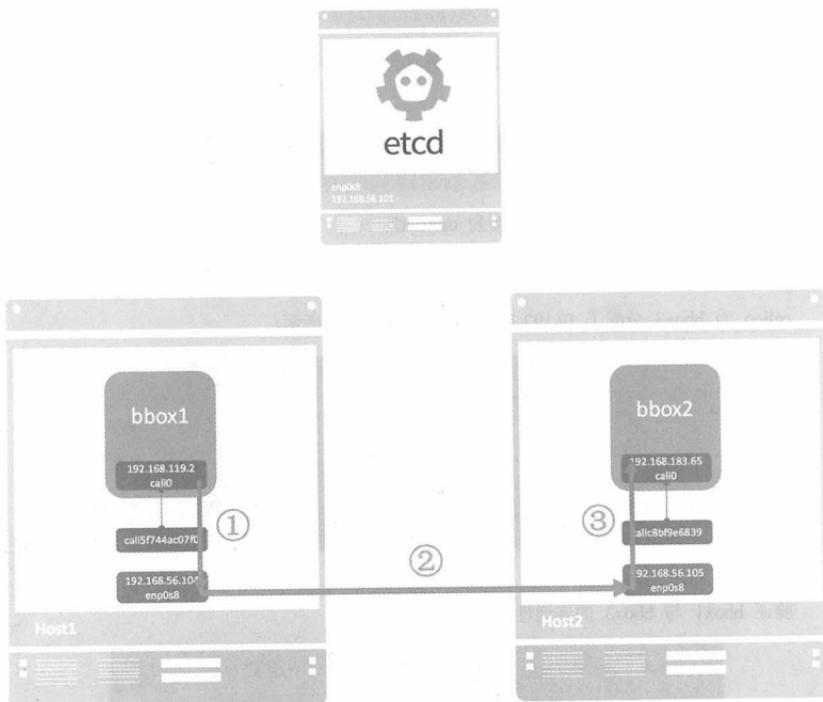


图 8-105

① 根据 bbox1 的路由表, 将数据包从 cali0 发出, 如图 8-106 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ip route
default via 169.254.1.1 dev cali0
169.254.1.1 dev cali0
root@host1:~#
```

图 8-106

② 数据经过 veth pair 到达 host1, 查看路由表, 数据由 enp0s8 发给 host2 (192.168.56.105)。

```
192.168.183.64/26 via 192.168.56.105 dev enp0s8 proto bird
```

③ host2 收到数据包, 根据路由表发送给 calic8bf9e68397, 进而通过 veth pair cali0 到达 bbox2。

```
192.168.183.65 dev calic8bf9e68397 scope link
```

接下来我们看看不同 calico 网络之间的连通性。

创建 cal_net2。

```
docker network create --driver calico --ipam-driver calico-ipam
cal_net2
```

在 host1 中运行容器 bbox3，连接到 cal_net2：

```
docker container run --net cal_net2 --name bbox3 -tid busybox
```

calico 为 bbox3 分配了 IP 192.168.119.5，如图 8-107 所示。

```
root@host1:~#
root@host1:~# docker exec bbox3 ip address show cali0
18: cali0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
   inet 192.168.119.5/32 brd ff:ff:ff:ff:ff:ff scope global cali0
        valid_lft forever preferred_lft forever
    inet6 fe80::ecee:eff:feee:eeee/64 scope link
        valid_lft forever preferred_lft forever
root@host1:~#
```

图 8-107

验证 bbox1 与 bbox3 的连通性，如图 8-108 所示。

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 192.168.119.5
PING 192.168.119.5 (192.168.119.5): 56 data bytes

--- 192.168.119.5 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#
```

图 8-108

虽然 bbox1 和 bbox3 都位于 host1，而且都在一个 subnet 192.168.119.0/26，但它们属于不同的 calico 网络，默认不能通行。calico 默认的 policy 规则是：容器只能与同一个 calico 网络中的容器通信。

calico 的每个网络都有一个同名的 profile，profile 中定义了该网络的 policy。我们具体看一下 cal_net1 的 profile，如图 8-109 所示。

```
calicoctl get profile cal_net1 -o yaml
```

```

root@host1:~#
root@host1:~# calicoctl get profile cal_net1 -o yaml
- apiVersion: v1
  kind: profile
  metadata:
    name: cal_net1 ①
    tags:
      - cal_net1 ②
  spec:
    egress: ③
      - action: allow
        destination: {}
        source: {}
    ingress:
      - action: allow
        destination: {}
        source:
          tag: cal_net1 ④
root@host1:~#

```

图 8-109

① 命名为 cal_net1，这就是 calico 网络 cal_net1 的 profile。

② 为 profile 添加一个 tag cal_net1。注意，这个 tag 虽然也叫 cal_net1，其实可以随便设置，这跟上面的 name: cal_net1 没有任何关系。此 tag 后面会用到。

③ egress 对从容器发出的数据包进行控制，当前没有任何限制。

④ ingress 对进入容器的数据包进行限制，当前设置是接收来自 tag cal_net1 的容器，根据第①步设置我们知道，实际上就是只接收本网络的数据包，这也进一步解释了前面的实验结果。

既然这是默认 policy，那就有方法定制 policy，这也是 calico 较其他网络方案最大的特性。

8.6.7 calico policy

calico 能够让用户定义灵活的 policy 规则，精细化控制进出容器的流量，下面我们就来实践一个场景：

- (1) 创建一个新的 calico 网络 cal_web 并部署一个 httpd 容器 web1。
- (2) 定义 policy 允许 cal_net2 中的容器访问 web1 的 80 端口。

首先创建 cal_web。

```

docker network create --driver calico --ipam-driver calico-ipam
cal_web

```

在 host1 中运行容器 web1，连接到 cal_web：

```

docker container run --net cal_web --name web1 -d httpd

```

web1 的 IP 为 192.168.119.7，如图 8-110 所示。

```
root@host1:~#
root@host1:~# docker container exec web1 ip address show calio
22: calio@if23: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
        [inet 192.168.119.7/32] scope global calio
            valid_lft forever preferred_lft forever
            inet6 fe80::ecce:eff:ffff:119.7/64 scope link
                valid_lft forever preferred_lft forever
root@host1:~#
```

图 8-110

目前 bbox3 还无法访问 web1 的 80 端口，如图 8-111 所示。

```
root@host1:~# docker container exec bbox3 wget 192.168.119.7
Connecting to 192.168.119.7 (192.168.119.7:80)
wget: can't connect to remote host (192.168.119.7): Connection timed out
root@host1:~#
```

图 8-111

创建 policy 文件 web.yml，内容如图 8-112 所示。

```
- apiVersion: v1
  kind: profile
  metadata:
    name: cal_web ①
  spec:
    ingress:
      - action: allow
        protocol: tcp
        source:
          tag: cal_net2 ②
        destination:
          ports:
            - 80 ③
```

图 8-112

① profile 与 cal_web 网络同名，cal_web 的所有容器（web1）都会应用此 profile 中的 policy。

② ingress 允许 cal_net2 中的容器（bbox3）访问。

③ 只开放 80 端口。

应用该 policy。

```
calicoctl apply -f web.yml
```

现在 bbox3 已经能够访问 web1 的 http 服务了，如图 8-113 所示。

```
root@host1:~#
root@host1:~# docker container exec bbox3 wget 192.168.119.7
Connecting to 192.168.119.7 (192.168.119.7:80)
index.html      100% |*****| 45  0:00:00 ETA
```

图 8-113

不过 ping 还是不行，因为只放开了 80 端口，如图 8-114 所示。

```
root@host1:~#
root@host1:~# docker container exec bbox3 ping -c 2 192.168.119.7
PING 192.168.119.7 (192.168.119.7): 56 data bytes

--- 192.168.119.7 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#
```

图 8-114

上面这个例子比较简单，不过已经向我们展示了 calico 强大的 policy 功能。通过 policy，可以动态实现非常复杂的容器访问控制。有关 calico policy 更多的配置，可参看官网文档 <http://docs.projectcalico.org/v2.0/reference/calicoctl/resources/policy>。

8.6.8 calico IPAM

如前所示，如果不特别配置，calico 会自动为网络分配 subnet，当然我们也可以定制。首先定义一个 IP Pool，比如：

```
cat << EOF | calicoctl create -f -- apiVersion: v1 kind: ipPool
metadata: cidr: 17.2.0.0/16 EOF
```

用此 IP Pool 创建 calico 网络。

```
docker network create --driver calico --ipam-driver calico-ipam --
subnet=17.2.0.0/16 my_net
```

此时运行容器将分配到指定 subnet 中的 IP，如图 8-115 所示。

```
root@host1:~#
root@host1:~# docker container run --net my_net -ti busybox
/ #
/ # ip address show cali0
26: cali0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
        inet 17.2.119.1/32 scope global cali0
            valid_lft forever preferred_lft forever
        inet6 fe80::ecce:eeff:feee:eeee/64 scope link
            valid_lft forever preferred_lft forever
/ #
```

图 8-115

当然也可以通过 `--ip` 为容器指定 IP，但必须在 subnet 范围之内，如图 8-116 所示。

```
root@host1:~#
root@host1:~# docker container run --net my_net --ip 17.2.3.11 -ti busybox
/ #
/ # ip address show cali0
30: cali0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
        inet 17.2.3.11/32 brd 17.2.3.11 scope global cali0
            valid_lft forever preferred_lft forever
        inet6 fe80::ecee:efff:feee:eeee/64 scope link
            valid_lft forever preferred_lft forever
/ #
```

图 8-116

8.7 比较各种网络方案

Docker 起初只提供了简单的 single-host 网络，显然这不利于 Docker 构建容器集群并通过 `scale-out` 方式横向扩展到多个主机上。

在市场需求的推动下，跨主机容器网络技术开始发展。这是一个非常活跃的技术领域，在较短的时间里已经涌现了很多优秀方案。本章我们详细讨论了几种主流的方案：Docker Overlay、Macvlan、Flannel、Weave 和 Calico。现在是时候做个比较了，让大家对各种方案的特点和优势有更深入的理解。

我们将从以下几个方面比较，大家可以根据不同场景选择最合适的方案。

1. 网络模型

采用何种网络模型支持 multi-host 网络？

Distributed Store 是否需要 etcd 或 consul 这类分布式 key-value 数据库存储网络信息？

2. IPMA

如何管理容器网络的 IP？

3. 连通与隔离

提供怎样的网络连通性？支持容器间哪个级别和哪个类型的隔离？

4. 性能

性能比较。

8.7.1 网络模型

跨主机网络意味着将不同主机上的容器用同一个虚拟网络连接起来。这个虚拟网络的拓扑结构和实现技术就是网络模型。

Docker overlay 如名称所示，是 overlay 网络，建立主机间 VxLAN 隧道，原始数据包在发送端被封装成 VxLAN 数据包，到达目的后在接收端解包。

Macvlan 网络在二层上通过 VLAN 连接容器，在三层上依赖外部网关连接不同 macvlan。数据包直接发送，不需要封装，属于 underlay 网络。

Flannel 讨论了两种 backend：vxlan 和 host-gw。vxlan 与 Docker overlay 类似，属于 overlay 网络。host-gw 将主机作为网关，依赖三层 IP 转发，不需要像 vxlan 那样对包进行封装，属于 underlay 网络。

Weave 是 VxLAN 实现，属于 overlay 网络。

各方案的网络模型描述如表 8-1 所示。

表 8-1 各方案的网络模型

	Docker Overlay	Macvlan	Flannel vxlan	Flannel host-gw	Weave	Calico
网络模型	Overlay: VxLAN	Underlay	Overlay: VxLAN	Underlay: 纯三层	Overlay: VxLAN	Underlay: 纯三层

8.7.2 Distributed Store

Docker Overlay、Flannel 和 Calico 都需要 etcd 或 consul。Macvlan 是简单的 local 网络，不需要保存和共享网络信息。Weave 自己负责在主机间交换网络配置信息，也不需要 Distributed Store，如表 8-2 所示。

表 8-2 各个网络 Distributed Store 要求

	Docker Overlay	Macvlan	Flannel vxlan	Flannel host-gw	Weave	Calico
Distributed Store	Yes	No	Yes	Yes	No	Yes

8.7.3 IPAM

Docker Overlay 网络中所有主机共享同一个 subnet，容器启动时会顺序分配 IP，可以通过 --subnet 定制此 IP 空间。

Macvlan 需要用户自己管理 subnet，为容器分配 IP，不同 subnet 通信依赖外部网关。

Flannel 为每个主机自动分配独立的 subnet，用户只需要指定一个大的 IP 池。不同 subnet 之间的路由信息也由 Flannel 自动生成和配置。

Weave 的默认配置下所有容器使用 10.32.0.0/12 subnet，如果此地址空间与现有 IP 冲突，可以通过 --ipalloc-range 分配特定的 subnet。

Calico 从 IP Pool (可定制) 中为每个主机分配自己的 subnet, 如表 8-3 所示。

表 8-3 各个网络 IPAM 要求

	Docker Overlay	Macvlan	Flannel vxlan	Flannel host-gw	Weave	Calico
IPAM	单一 subnet	自定义	每个 host 一个 subnet	每个 host 一个 subnet	单一 subnet	每个 host 一个 subnet

8.7.4 连通与隔离

同一 Docker Overlay 网络中的容器可以通信, 但不同网络之间无法通信, 要实现跨网络访问, 只有将容器加入多个网络。与外网通信可以通过 docker_gwbridge 网络。

Macvlan 网络的连通或隔离完全取决于二层 VLAN 和三层路由。

不同 Flannel 网络中的容器直接就可以通信, 没有提供隔离。与外网通信可以通过 bridge 网络。

Weave 网络默认配置下所有容器在一个大的 subnet 中, 可以自由通信, 如果要实现隔离, 需要为容器指定不同的 subnet 或 IP。与外网通信的方案是将主机加入到 weave 网络, 并把主机当作网关。

Calico 默认配置下只允许位于同一网络中的容器之间通信, 但通过其强大的 Policy 能够实现几乎任意场景的访问控制。

8.7.5 性能

性能测试是一个非常严谨和复杂的工程, 这里我们只尝试从技术方案的原理上比较各方案的性能。

最朴素的判断是: Underlay 网络性能优于 Overlay 网络。

Overlay 网络利用隧道技术, 将数据包封装到 UDP 中进行传输。因为涉及数据包的封装和解封, 存在额外的 CPU 和网络开销。虽然几乎所有 Overlay 网络方案底层都采用 Linux kernel 的 vxlan 模块, 这样可以尽量减少开销, 但这个开销与 Underlay 网络相比还是存在的。所以 Macvlan、Flannel host-gw、Calico 的性能会优于 Docker overlay、Flannel vxlan 和 Weave。

Overlay 较 Underlay 可以支持更多的二层网段, 能更好地利用已有网络, 以及有避免物理交换机 MAC 表耗尽等优势, 所以在方案选型的时候需要综合考虑。

第 9 章

◀ 容器监控 ▶

随着 Docker 部署规模逐步变大后，可视化监控容器环境的性能和健康状态将会变得越来越重要。

在本章中，我们将讨论几个目前比较常用的容器监控工具和方案，为大家构建自己的监控系统提供参考，如图 9-1 所示。

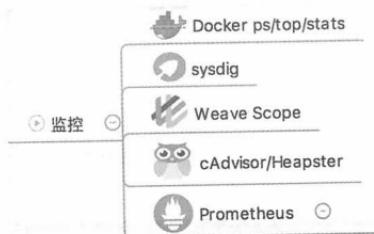


图 9-1

首先我们会讨论 Docker 自带的几个监控子命令：ps、top 和 stats，然后是几个功能更强的开源监控工具 sysdig、Weave Scope、cAdvisor 和 Prometheus，最后我们会对这些不同的工具和方案做一个比较。

9.1 Docker 自带的监控子命令

9.1.1 ps

`docker container ps` 是我们早已熟悉的命令了，方便我们查看当前运行的容器，如图 9-2 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2fa379bcf0e	prox/node-exporter	"/bin/node_exporter..."	16 hours ago	Up 16 hours		node-exporter
32cd721050b2	google/cadvisor:latest	"/usr/bin/cadvisor..."	16 hours ago	Up 16 hours		cadvisor
0d92b2b28b384	weaveworks/weavescope:1.2.1	"/home/weave/entry..."	4 days ago	Up 4 days		weavescope
61e2d9067694	sysdig/sysdig	"/docker-entrypoint..."	4 days ago	Up 4 days		sysdig
fef1f331c1ff	weaveworks/weaveplugin:1.8.2	"/home/weave/plugin..."	7 weeks ago	Up 4 days		weaveplugin
7e10fe03f336	weaveworks/weaveexec:1.8.2	"/home/weave/weave..."	7 weeks ago	Up 4 days		weaveexec
54950b00132d	weaveworks/weave:1.8.2	"/home/weave/weave..."	7 weeks ago	Up 4 days		weave

图 9-2

前面已经有大量示例，这里就不赘述了。值得注意的是，新版的 Docker 提供了一个新命令 docker container ls，其作用和用法与 docker container ps 完全一样。不过 ls 含义可能比 ps 更准确，所以更推荐使用，如图 9-3 所示。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2fa379bcf0e	prox/node-exporter	"/bin/node_exporter..."	16 hours ago	Up 16 hours		node-exporter
32cd721050b2	google/cadvisor:latest	"/usr/bin/cadvisor..."	16 hours ago	Up 16 hours		cadvisor
0d92b2b28b384	weaveworks/weavescope:1.2.1	"/home/weave/entry..."	4 days ago	Up 4 days		weavescope
61e2d9067694	sysdig/sysdig	"/docker-entrypoint..."	4 days ago	Up 4 days		sysdig
fef1f331c1ff	weaveworks/weaveplugin:1.8.2	"/home/weave/plugin..."	7 weeks ago	Up 4 days		weaveplugin
7e10fe03f336	weaveworks/weaveexec:1.8.2	"/home/weave/weave..."	7 weeks ago	Up 4 days		weaveexec
54950b00132d	weaveworks/weave:1.8.2	"/home/weave/weave..."	7 weeks ago	Up 4 days		weave

图 9-3

9.1.2 top

如果想知道某个容器中运行了哪些进程，可以执行 docker container top [container] 命令，如图 9-4 所示。

[root@ubuntu ~]# docker container top sysdig							
UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	8281	8251	0	Mar17	pts/1	00:00:00	bash
root	9567	8281	4	Mar17	pts/1	04:32:24	csysdig

图 9-4

上面显示了 sysdig 这个容器中的进程。命令后面还可以跟上 Linux 操作系统 ps 命令的参数显示特定的信息，比如 -au，这样命令 docker container top sysdig -au 执行结果如图 9-5 所示。

USER	PID	%CPU	%MEM	VSS	RSS	TTY	STAT	START
root	8281	0.0	0.0	19057	1680	pts/1	Ss	Mar17
root	9567	4.3	3.5	128192	71816	pts/1	S+	Mar17
dig								

图 9-5

9.1.3 stats

docker container stats 用于显示每个容器各种资源的使用情况，如图 9-6 所示。

CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
2fa379cef0e	0.00%	11.63 MiB / 1.954 GiB	0.55%	0 B / 0 B	0 B / 0 B	4
32cd721050b2	2.13%	74.07 MiB / 1.954 GiB	3.70%	0 B / 0 B	15.8 MB / 0 B	14
0d92b2b28b384	5.56%	184.6 MiB / 1.954 GiB	9.23%	0 B / 0 B	2.36 MB / 180 kB	37
61e2d9067b94	4.42%	71.82 MiB / 1.954 GiB	3.59%	16.4 kB / 648 B	397 MB / 1.68 MB	2
fef1f331c11f	0.09%	12.37 MiB / 1.954 GiB	0.62%	0 B / 0 B	12.4 MB / 0 B	7
7efafaa3f336	0.09%	12.2 MiB / 1.954 GiB	0.61%	0 B / 0 B	18.6 MB / 0 B	9
54950b90132d	0.25%	63.38 MiB / 1.954 GiB	3.17%	0 B / 0 B	34.9 MB / 188 kB	15

图 9-6

默认会显示一个实时变化的列表，展示每个容器的 CPU 使用率、内存使用量和可用量。

注意：容器启动时如果没有特别指定内存 limit，stats 命令这里会显示 host 的内存总量，但这并不意味着每个 container 都能使用到这么多的内存。

除此之外 docker container stats 命令还会显示容器网络和磁盘的 IO 数据。

默认的输出有关缺点，显示的是容器 ID 而非名字。我们可以在 stats 命令后面指定容器的名称只显示某些容器的数据。比如 docker container stats sysdig weave，如图 9-7 所示。

CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
sysdig	3.32%	71.7 MiB / 1.954 GiB	3.58%	16.4 kB / 648 B	397 MB / 1.68 MB	2
weave	0.16%	47.27 MiB / 1.954 GiB	2.36%	0 B / 0 B	34.9 MB / 188 kB	15

图 9-7

Ps、top、stats 这几个命令是 Docker 自带的，优点是运行方便，很适合想快速了解容器的运行状态的场景。其缺点是输出的数据有限，而且都是实时数据，无法反映历史变化和趋势。接下来介绍的几个监控工具会提供更丰富的功能。

9.2 sysdig

sysdig 是一个轻量级的系统监控工具，同时它还原支持容器。通过 sysdig 我们可以近距离观察 Linux 操作系统和容器的行为。

Linux 上有很多常用的监控工具，比如 strace、tcpdump、htop、iftop、lsof

而 sysdig 则是将这些工具的功能集成到一个工具中，并且提供一个友好统一的操作界面。

下面我们将演示 sysdig 强大的监控能力。

安装和运行 sysdig 的最简单方法是运行 Docker 容器，命令行为：

```
docker container run -it --rm --name=sysdig --privileged=true \
--volume=/var/run/docker.sock:/host/var/run/docker.sock \
--volume=/dev:/host/dev \
--volume=/proc:/host/proc:ro \
--volume=/boot:/host/boot:ro \
--volume=/lib/modules:/host/lib/modules:ro
```

```
\ --volume=/usr:/host/usr:ro \ sysdig/sysdig
```

可以看到，sysdig 容器是以 privileged 方式运行，而且会读取操作系统 /dev、/proc 等数据，这是为了获取足够的系统信息。

启动后，通过 docker container exec -it sysdig bash 进入容器，执行 csysdig 命令，将以交互方式启动 sysdig，如图 9-8 所示。

Viewing: Processes For: whole machine						
Source: Live System	Filter: evt.type==switch	TH	VIRT	RES	FILE	NET Command
2587	5.50 root	14	465M	86M	76K	9.46K /usr/bin/cadvisor -logtostderr
7529	5.00 root	30	801M	154M	533K	41.37K /usr/bin/dockerd -H tcp://0.0.0.0:23
18333	3.50 root	1	94M	45M	0	0.00 sysdig
15818	2.50 root	15	585M	113M	474K	44.14K scope-probe --mode probe --probe.doc
7536	1.00 root	22	461M	29M	2K	9.50K docker-containerd -l unix:///var/run
15816	0.50 root	11	479M	104M	0	10.75K scope-app --mode app --probe.docker=
2502	0.00 root	3	271M	5M	0	0.00 /usr/lib/policykit-1/polkitd --no-de
7731	0.00 root	15	708M	44M	0	3.15K /home/weave/weaver --port 6783 --nam
20351	0.00 root	1	0	0	0	0.00 docker-runc events --stats 32cd72105
20526	0.00 root	1	0	0	0	0.00 docker-runc events --stats fef1331c
20594	0.00 root	1	0	0	0	0.00 docker-runc events --stats 0d92bb28b
20507	0.00 root	1	0	0	0	0.00 docker-runc events --stats 32cd72105
20332	0.00 root	1	0	0	0	0.00 docker-runc events --stats 54950b901
20065	0.00 root	1	0	0	0	0.00 docker-runc events --stats 2fa0379bce
20870	0.00 root	1	0	0	0	0.00 docker-runc events --stats 7efafa03f
2420	0.00 root	3	157M	1M	0	0.00 /usr/bin/lxcfs /var/lib/lxcfs/
27749	0.00 root	1	24M	7M	0	0.00 -bash
20045	0.00 root	1	0	0	0	0.00 docker-runc events --stats 0d92bb28b
19911	0.00 root	1	0	0	0	0.00 docker-runc events --stats 54950b901
7751	0.00 root	9	202M	4M	0	0.00 docker-containerd-shim feff1331c1f1
20086	0.00 root	1	0	0	0	0.00 docker-runc events --stats 7efafa03f

图 9-8

这是一个类似 Linux top 命令的界面，但要强大太多。sysdig 按不同的 View 来监控不同类型的资源，单击底部 Views 菜单（或者按 F2 键），显示 View 选择列表，如图 9-9 所示。

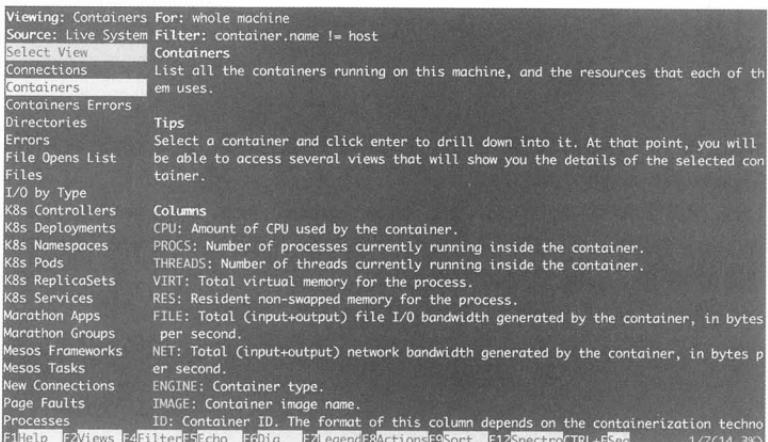


图 9-9

界面左边列出了 sysdig 支持的 View，一共 30 多项，涵盖了操作系统的各个方面，因为这里主要是讨论容器监控，所以我们将光标移到 Containers 这一项，界面右边立即显示出此 View 的功能介绍。

回车或者双击 Containers，进入容器监控界面，如图 9-10 所示。

The screenshot shows the sysdig interface with the 'Containers' view selected. The table lists 29 containers, each with columns for CPU, PROCS, THREADS, VIRT, RES, FILE, NET, ENGINE, IMAGE, ID, and NAME. The containers listed include various weaveworks applications, a sysdig container, and several node-exporter and advisor containers. The interface includes a toolbar at the top with buttons for Help, Views, Filter, Echo, Finfo, Legend, Actions, Sort, Spectra, CTRL+F, Search, and Pause.

Viewing: Containers For: whole machine										
Source: Live System Filter: container.name != host										
CPU	PROCS	THREADS	VIRT	RES	FILE	NET	ENGINE	IMAGE	ID	NAME
3.50	3	3	153M	75M	0	0.00	docker	weaveworks/scope:1.2.1	d538e7c95021	sysdig
3.00	6	28	1G	215M	465K	34.15K	docker	weaveworks/scope:1.2.1	0d92bb28b384	weavescope
2.50	1	12	465M	84M	80K	0.00	docker	google/advisor:latest	32cd721b050b2	advisor
0.50	1	4	48M	19M	28K	6.95K	docker	prom/node-exporter	2fa379bcff0e	node-exporter
0.50	0	5	272M	8M	0	0.00	docker	weaveworks/weaveexec:1.8.2	7e1fa0af3f36	weaveproxy
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	43b667ed758	compassionate_bell
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	77c5e8bd0d0	naughty_bhaskara
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	ee4967098262	quizzical_rentgen
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	54950b090132d	wave
0.00	1	13	708M	43M	0	3.15K	docker	weaveworks/weave:1.8.2	23ca4684e43b	compassionate_rosalind
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	fe1f1331c11f	weaveplugin
0.00	0	5	256M	11M	0	0.00	docker	weaveworks/plugin:1.8.2	c80057339383	cranky_euler
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	d50538103f6d	hardcore_poitras
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	a10e3c67160e	stoic_almeida
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	cb408402f9	elastic_roman
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	d1a0d7dc40e9	brave_poincare
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	591470a0145	inspiring_leakey
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	e6f959872c3	clever_wescoff
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	eac25f745555	happy_brottain
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	dfc54ccfdb61	thirsty_poitras

图 9-10

sysdig 会显示该 Host 所有容器的实时数据，每两秒刷新一次。各列数据的含义也是自解释的，如果不清楚，可以点一下底部 Legend（或者按 F7 键）。如果想按某一列排序，比如按使用的内存量，很简单，点一下列头 VIRT，如图 9-11 所示。

The screenshot shows the sysdig interface with the 'Containers' view selected. The table lists 29 containers, similar to Figure 9-10. The 'VIRT' column is highlighted with a yellow background. The interface includes a toolbar at the top with buttons for Help, Views, Filter, Echo, Finfo, Legend, Actions, Sort, Spectra, CTRL+F, Search, and Pause.

Viewing: Containers For: whole machine										
Source: Live System Filter: container.name != host										
CPU	PROCS	THREADS	VIRT	RES	FILE	NET	ENGINE	IMAGE	ID	NAME
5.50	6	28	1G	215M	465K	29.70K	docker	weaveworks/scope:1.2.1	0d92bb28b384	weavescope
0.00	1	13	708M	43M	0	0.00	docker	weaveworks/weave:1.8.2	54950b090132d	wave
9.00	1	12	465M	84M	102K	9.47K	docker	google/advisor:latest	32cd721b050b2	advisor
0.00	0	5	272M	8M	0	0.00	docker	weaveworks/weaveexec:1.8.2	7e1fa0af3f36	weaveproxy
0.00	0	5	256M	11M	0	0.00	docker	weaveworks/plugin:1.8.2	fe1f1331c11f	weaveplugin
5.00	3	13	153M	75M	0	0.00	docker	sysdig/sysdig	d538e7c95021	sysdig
0.00	1	4	48M	19M	0	0.00	docker	prom/node-exporter	2fa379bcff0e	node-exporter
0.00	0	0	25M	8M	0	0.00	docker	weaveworks/weaveexec:1.9.0	c7c50dd6046d	clever_poitras
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	d1a0d7dc40e9	dazzling_nobel
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	11c5135992d	upbeat_einstein
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	43b667ed7580	compassionate_bell
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	c80057339383	cranky_euler
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	231d78759996	nostalgic_almeida
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	5c64108f9e5e	hopeful_wiles
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	8fe0a554a022	laughing_nightingale
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	5bd1869d6051	tender_hobit
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	77c5e8bd0d0	naughty_bhaskara
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	b0d9b1f6d268	musing_goldstone
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	658721e7f60	relaxed_morse
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	e26758fe8e6	elastic_shirley
0.00	0	0	0	0	0	0.00	docker	weaveworks/weaveexec:1.9.0	c669698fcf24	practical_turing

图 9-11

如果想看某个容器运行的进程，比如 weavescope，将光标移到目标容器，然后回车或者双击，如图 9-12 所示。

Viewing: Processes For: container.id="0d92bb28b384"							
Source: Live System Filter: (((container.name != host) and container.id="0d92bb28b384")) and (evt.type!=switch)							
PID	CPU	USER	TH	VIRT	RES	FILE	NET Command
15818	5.50	root	15	585M	113M	508K	32.63K scope-probe --mode probe --probe.docker=true
15816	1.00	root	11	479M	104M	0	11.31K scope-app --mode app --probe.docker=true
15814	0.00	root	1	748K	0	0	0.00 runsv probe
15970	0.00	root	1	7M	536K	0	0.00 conntrack --buffer-size 212992 -E -o id -p tcp --any-nat
15813	0.00	root	1	768K	0	0	0.00 /sbin/runsvdir /etc/service
15815	0.00	root	1	748K	0	0	0.00 runsv app
15949	0.00	root	1	7M	444K	0	0.00 conntrack --buffer-size 212992 -E -o id -p tcp
15798	0.00	root	6	3M	0	0	0.00 /home/weave/runsvinit

图 9-12

还可以继续双击查看进程中的线程，如图 9-13 所示。

Viewing: Threads For: container.id="0d92bb28b384" and proc.pid=15818							
Source: Live System Filter: (((container.name != host) and container.id="0d92bb28b384") and ((evt.type!=switch)) and (proc.pid==15818))							
PID	TID	CPU	FILE	NET	Command		
15818	15841	2.50	117.01K	19.65K	scope-probe --mode probe --probe.docker=true		
15818	31379	1.00	0.00	1.85K	scope-probe --mode probe --probe.docker=true		
15818	15839	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15939	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15977	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15934	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15856	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	23049	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15976	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15847	0.00	0.00	1.44K	scope-probe --mode probe --probe.docker=true		
15818	15974	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15818	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	16030	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15838	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		
15818	15972	0.00	0.00	0.00	scope-probe --mode probe --probe.docker=true		

图 9-13

返回上一级，按退格键即可。

sysdig 的交互功能很强，如果界面显示的条目很多，可以单击底部 Search 菜单，然后输入关键字进行查找，如图 9-14 所示，关键字为 service。

Viewing: Processes For: container.id="0d92bb28b384"							
Source: Live System Filter: (((container.name != host) and container.id="0d92bb28b384")) and (evt.type!=switch)							
PID	CPU	USER	TH	VIRT	RES	FILE	NET Command
15818	2.00	root	15	585M	113M	465K	22.06K scope-probe --mode probe --probe.docker=true
15816	0.00	root	11	479M	104M	0	5.13K scope-app --mode app --probe.docker=true
15949	0.00	root	1	7M	444K	0	0.00 conntrack --buffer-size 212992 -E -o id -p tcp --any-nat
15970	0.00	root	1	7M	536K	0	0.00 conntrack --buffer-size 212992 -E -o id -p tcp --any-nat
15798	0.00	root	6	3M	0	0	0.00 /home/weave/runsvinit
15813	0.00	root	1	768K	0	0	0.00 /sbin/runsvdir /etc/service
15814	0.00	root	1	748K	0	0	0.00 runsv probe
15815	0.00	root	1	748K	0	0	0.00 runsv app
28162	0.00	root	1	0	0	0	0.00 docker ps -q
25007	0.00	root	1	0	0	0	0.00 docker -v
24940	0.00	root	1	0	0	0	0.00 docker ps -q
25025	0.00	root	1	0	0	0	0.00 docker run -rm --privileged --net-host --pid=host -v /v
29763	0.00	root	1	0	0	0	0.00 docker ps -q
25454	0.00	root	1	0	0	0	0.00 docker -v
28706	0.00	root	1	0	0	0	0.00 docker -v
29193	0.00	root	1	0	0	0	0.00 docker -v
24493	0.00	root	1	0	0	0	0.00 docker run -rm --privileged --net-host --pid=host -v /v
24929	0.00	root	1	0	0	0	0.00 docker -v
29204	0.00	root	1	0	0	0	0.00 docker ps -q
25467	0.00	root	1	0	0	0	0.00 docker ps -q
26490	0.00	root	1	0	0	0	0.00 docker -v

图 9-14

如果觉得界面刷新太快，看不清楚关注的信息，可以单击底部 Pause 菜单。

sysdig 的特点如下：

- (1) 监控信息全，包括 Linux 操作系统和容器。
- (2) 界面交互性强。

不过 sysdig 显示的是实时数据，看不到变化和趋势，而且是命令行操作方式，需要 ssh 到 Host 上执行，会带来一些不便。下一节介绍的 Weave Scope 在这方面似乎提供了更好的解决方案。

9.3 Weave Scope

Weave Scope 的最大特点是会自动生成一张 Docker Host 地图，让我们能够直观地理解、监控和控制容器。千言万语不及一张图，先感受一下，如图 9-15 所示。

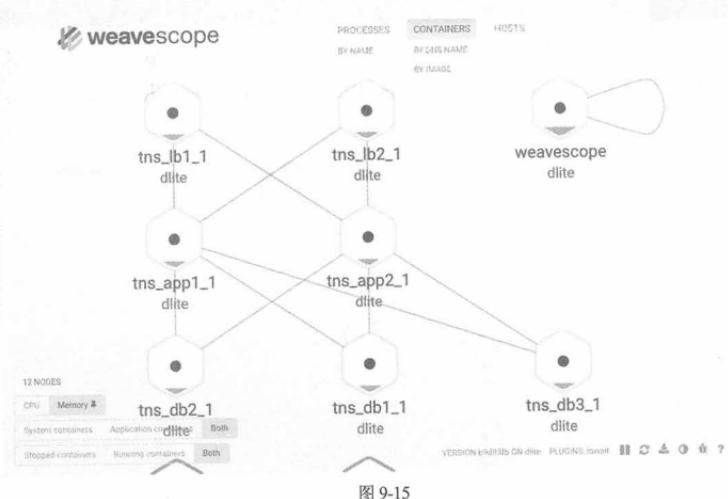


图 9-15

下面开始实践 Weave Scope。

9.3.1 安装

执行如下脚本安装运行 Weave Scope。

```
curl -L git.io/scope -o /usr/local/bin/scope chmod a+x
```

```
/usr/local/bin/scope scope launch
```

scope launch 将以容器方式启动 Weave Scope，如图 9-16、图 9-17 所示。

```
[root@ubuntu ~]# [root@ubuntu ~]# scope launch
98744dd0fe34962975e613caaf975cd99e2f6bab0e7438b1f71736becfd50c
Scope probe started
Weave Scope is reachable at the following URL(s):
  * http://192.168.122.1:4040/
  * http://10.0.2.15:4040/
  * http://192.168.56.102:4040/
  * http://10.32.0.3:4040/
[root@ubuntu ~]#
```

图 9-16

```
[root@ubuntu ~]# [root@ubuntu ~]# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
98744dd0fe3        weaveworks/scope:1.2.1   "/home/weave/entry..."   6 minutes ago      Up 6 minutes          sysdig
2cced07c77b0        sysdig/sysdig           "/docker-entrypoint..."   28 hours ago       Up 28 hours          node-exporter
2fc379bcfe0e        prom/node-exporter        "/bin/node_exporter..."  28 hours ago       Up 28 hours          cadvisor
32cd721050b2        google/cadvisor:latest    "/usr/bin/cadvisor..."  28 hours ago       Up 28 hours          weaveproxy
felf1331c11f        weaveworks/plugin:1.8.2     "/home/weave/plugin"   7 weeks ago        Up 4 days           weave
7efafa03f36         weaveworks/weaveexec:1.8.2   "/home/weave/weave..."  7 weeks ago        Up 4 days           weaveplugin
54950b09132d        weaveworks/weave:1.8.2     "/home/weave/weave..."  7 weeks ago        Up 4 days           weave
[root@ubuntu ~]#
```

图 9-17

根据提示，Weave Scope 的访问地址为 [http://\[Host IP\]:4040/](http://[Host IP]:4040/)，如图 9-18 所示。

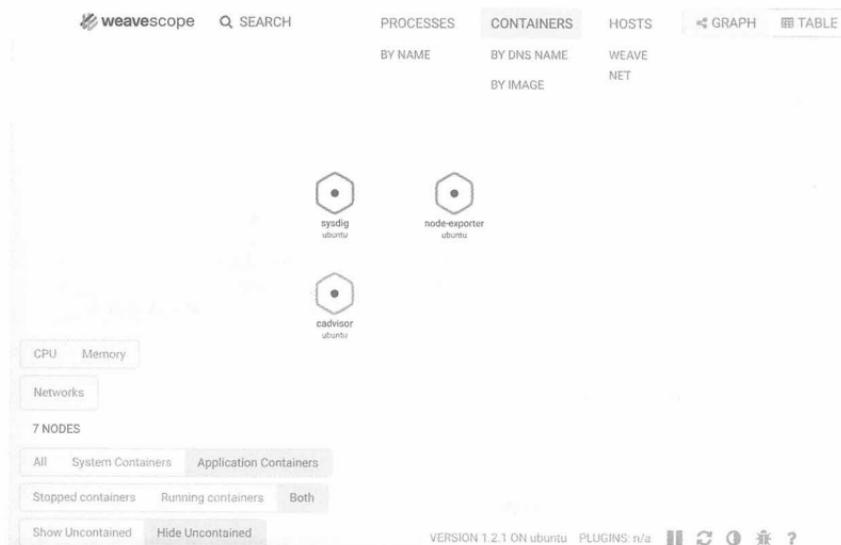


图 9-18

9.3.2 容器监控

Weave Scope 地图中间显示了 Host 当前运行的容器，不过少了几个 weave 相关的容器。

Weave Scope 将容器分为两类：Weave 自己的容器 System Container 和其他容器 Application Container，默认只显示后者。

Weave Scope 界面是一个可交互的地图，使用起来很方便。比如单击地图左下角选择开关 All，如图 9-19 所示。



图 9-19

地图上会立刻显示出所有的容器，如图 9-20 所示。

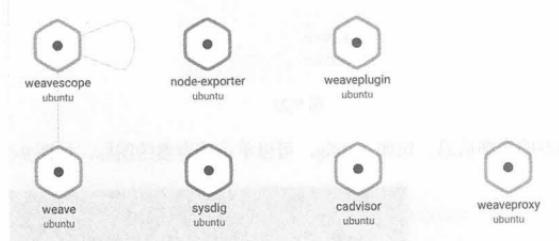


图 9-20

单击 CPU 选择器，如图 9-21 所示。

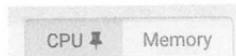


图 9-21

Weave Scope 将以高低水位方式显示容器 CPU 使用量，如图 9-22 所示。



图 9-22

如果此时我们将鼠标放到容器图标上，则会显示具体的 CPU 使用量百分比（%），如图 9-23 所示。

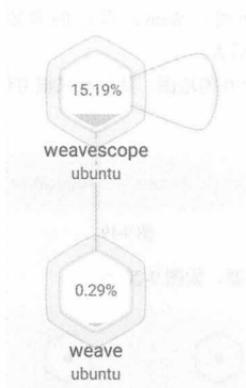


图 9-23

如果要查看容器的详细信息，比如 sysdig，可以单击该容器的图标，如图 9-24 所示。



图 9-24

详细信息包括这么几部分：

- (1) Status: CPU、内存的实时使用情况以及历史曲线。
 - (2) INFO: 容器 image、启动命令、状态、网络等信息。
- 以下几项需拉动滚动条查看，如图 9-25 所示。

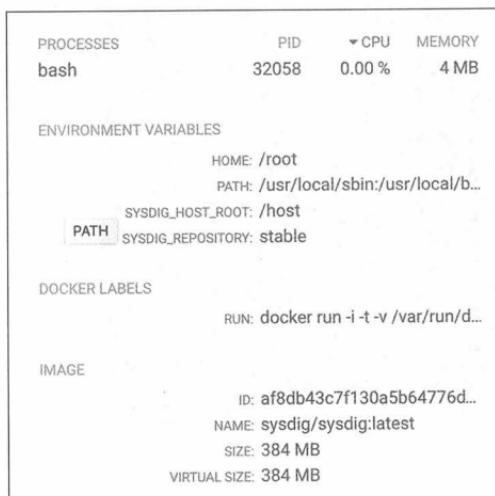


图 9-25

- (3) PROCESSES: 容器中运行的进程。
- (4) ENVIRONMENT VARIABLES: 环境变量。
- (5) DOCKER LABELS: 容器启动命令。
- (6) IMAGE: 镜像详细信息。

在容器信息的上面还有一排操作按钮，如图 9-26 所示。



图 9-26

分别是：

- █**: attach 到容器启动进程，相当于执行 docker container attach。
- █**: 打开 shell，相当于执行 docker container exec。
- █**: 重启容器，相当于执行 docker container restart。
- █**: 暂停容器，相当于执行 docker container pause。

■：关闭容器，相当于执行 docker container stop。

这排按钮使我们能够远程控制容器，相当方便。最常用的可能就是了。比如可以直接跳进 sysdig 容器，启动 csydidg 监控工具，如图 9-27 所示。



图 9-27

9.3.3 监控 host

除了监控容器，Weave Scope 还可以监控 Docker Host。单击顶部 HOSTS 菜单项，地图将显示当前 host，如图 9-28 所示。

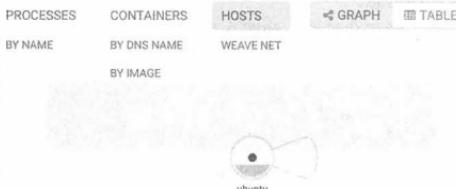


图 9-28

与容器类似，单击该 host 图标将显示详细信息，如图 9-29 所示。

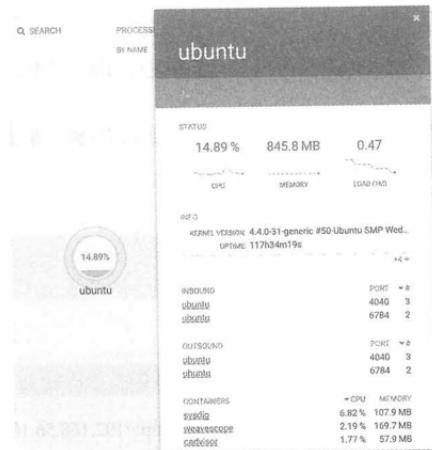


图 9-29

host 当前的资源使用情况和历史曲线一览无余。除此之外也能很方便地查看 host 上运行的进程和容器列表，单击容器名字还可以打开此容器的信息页面。

host 页面上部有一个按钮，单击可直接打开 host 的 shell 窗口，这个远程管理功能真的很贴心，如图 9-30 所示。

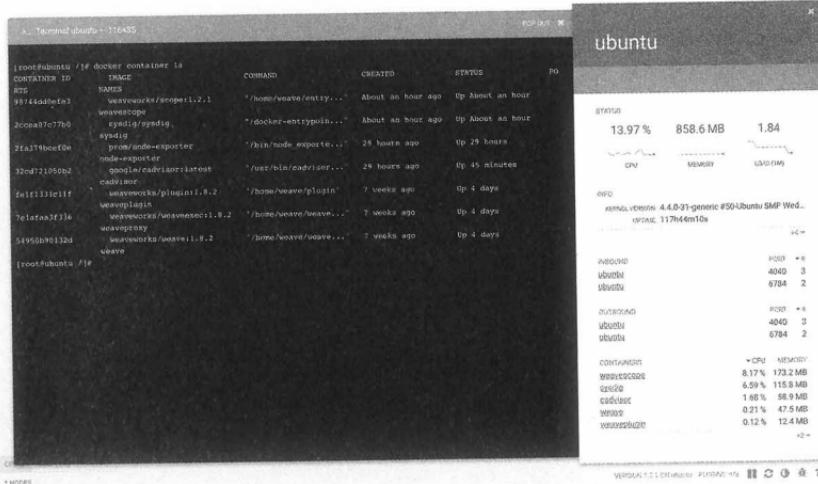


图 9-30

9.3.4 多主机监控

前面我们已经领略了 Weave Scope 的丰富功能和友好的操作界面，不过它还有一个重要功能：多主机监控。

真正的部署环境都不可能只有一个 host 在一个界面上监控整个容器环境，那绝对是非常有效率的事情。下面我们就来实践这个功能。

两个 Docker Host：

```
ubuntu: 192.168.56.102  
ubuntu2: 192.168.56.103
```

在两个 host 上都执行如下命令：

```
scope launch 192.168.56.102 192.168.56.103
```

这样，无论访问 <http://192.168.56.102:4040> 还是 <http://192.168.56.103:4040>，都能监控两个 host，如图 9-31 所示。

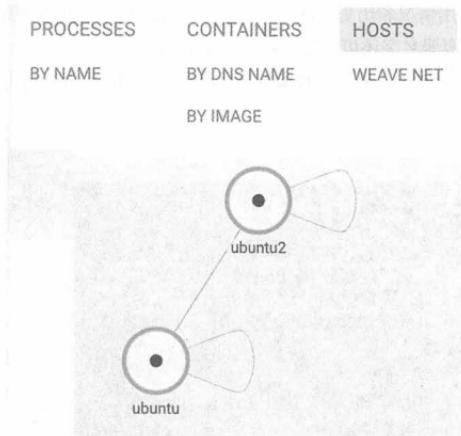


图 9-31

单击 CONTAINERS 菜单项，将显示部署环境中所有的容器，如图 9-32 所示。

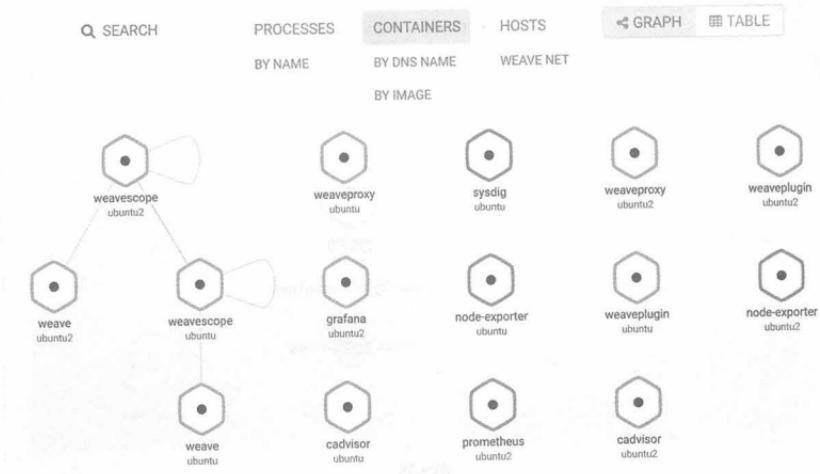


图 9-32

容器图标下面标明了所在的 host，如图 9-33 所示。



图 9-33

如果部署的容器数量太多（很常见），Weave Scope 还提供了强悍的搜索功能，如图 9-34 所示。



图 9-34

输入关键词 sysd，立刻会在地图中定位到容器 sysdig，如图 9-35 所示。

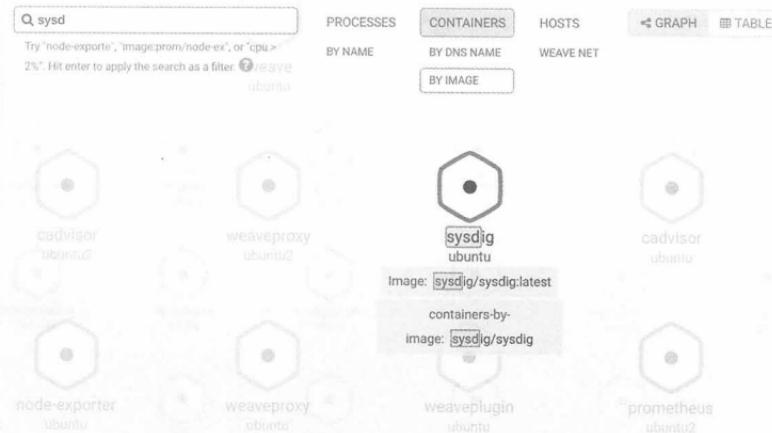


图 9-35

Weave Scope 还支持逻辑条件查询，比如输入 `cpu > 2`，立刻会找出 CPU 利用率高于 2% 的容器，如图 9-36 所示。

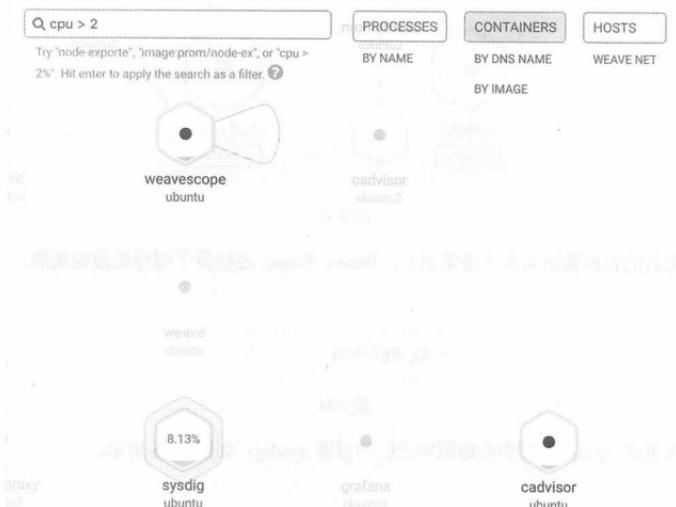


图 9-36

更多过滤方法可单击搜索框下面的 ? ，如图 9-37 所示。

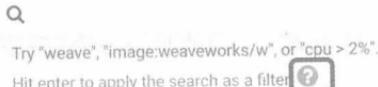


图 9-37

参考帮助和示例，如图 9-38 所示。

HELP	SHORTCUTS	SEARCH		FIELDS AND METRICS	
		BASIC	REGULAR EXPRESSIONS	FIELDS	METRICS
	GENERAL	Close active panel Activate search field Toggle shortcut menu Toggle Table mode Toggle Graph mode	Q: foo Q: pid: 12345 Q: foobar Q: command: foo(barbaz)	All fields for foo Any field matching pid for the value 12345 All fields for foo or bar command field for foobar or foobaz	Searchable fields and metrics in the currently selected CONTAINERS topology: command containersbyimage created execimage gfssecurityadminpassword gfservicerooturl glibversion home hosts id image ips
	CANVAS METRICS	Select and pin previous metric Select and pin next metric Unpin current metric			

图 9-38

Weave Scope 就讨论到这里，更多的功能，期待大家自己去发现。

9.4 cAdvisor

cAdvisor 是 google 开发的容器监控工具，我们来看看 cAdvisor 有什么能耐。

在 host 中运行 cAdvisor 容器。

```
docker run \ --volume=/:/rootfs:ro \ --volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \ --volume=/var/lib/docker/:/var/lib/docker:ro \
-p 8080:8080 \ --detach=true \ --name=cadvisor \
google/cadvisor:latest
```

通过 [http://\[Host IP\]:8080](http://[Host IP]:8080) 访问 cAdvisor。

9.4.1 监控 Docker Host

cAdvisor 会显示当前 host 的资源使用情况，包括 CPU、内存、网络、文件系统等，如图 9-39~图 9-42 所示。

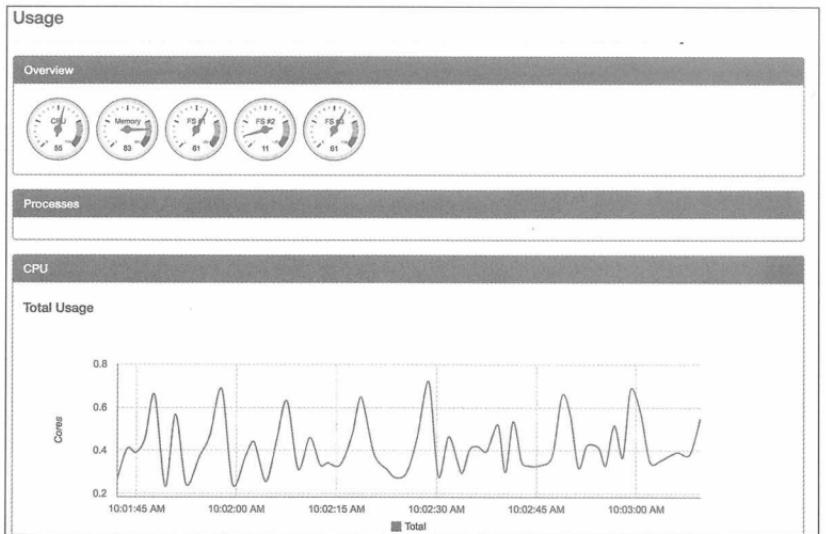


图 9-39

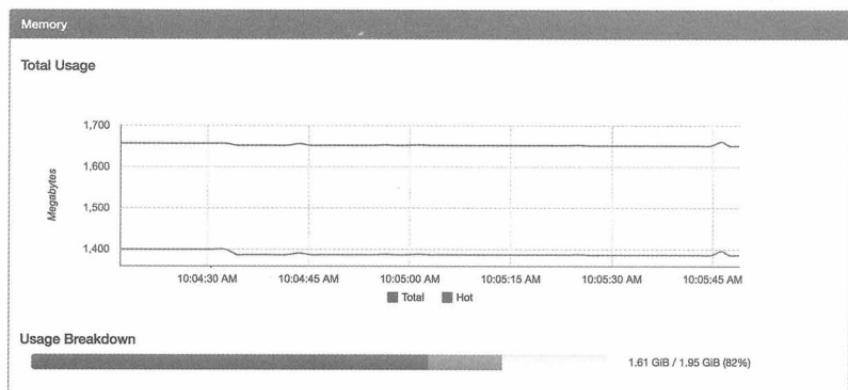


图 9-40

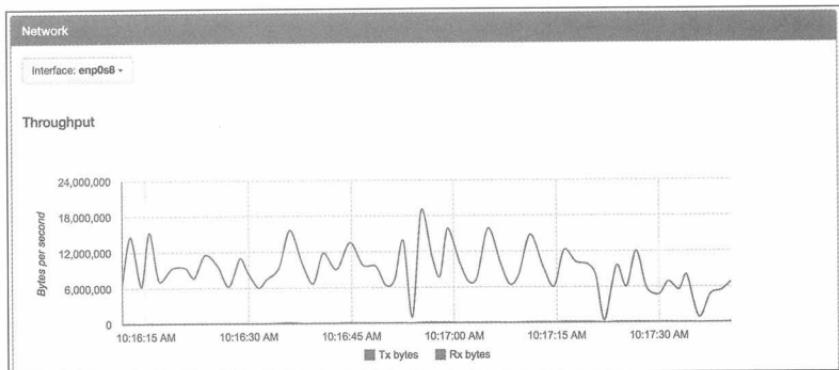


图 9-41

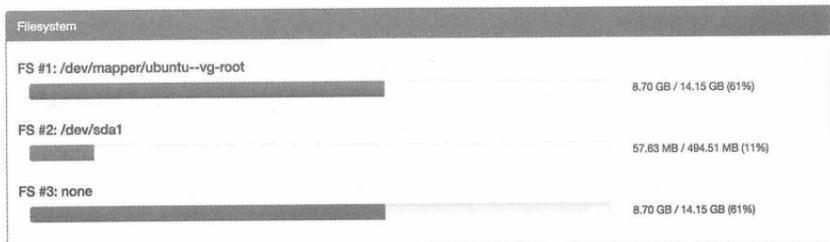


图 9-42

9.4.2 监控容器

单击 Docker Containers 链接，如图 9-43 所示。



图 9-43

显示容器列表，如图 9-44 所示。

```
Subcontainers

weaveproxy (/docker/7efafaa3f336a958da3788aff09d44404bd12a4d739ed836cb865032bacd4230)
weavescope (/docker/a0fcdea3fa5a5a275f17037fde180467c77fb03ea0281e59b284cc0a27814)
sysdig (/docker/2cce07c77b03936aa5b44d3d2a16012c720ca3e4d7aa734b5bb804b323106)
cadvisor (/docker/32cd7210502b31ned034b25e7f76cf7be661bc007453c3716a4c0f55bd14c)
weaveplugin (/docker/fef11331c11ffab539372955da487877eb0927bf4cca830754b9ff4acde7c715c)
weave (/docker/54950b90132d3145a26164cf0f1cf18b8ab6e0eacd0d4345fe1f7e039500e)
node-exporter (/docker/2fa5790cef0e1577d80c8b5e3f47e92d5fe189968ffeb936e97303e61a528c69)
```

图 9-44

单击某个容器，比如 sysdig，进入该容器的监控页面，如图 9-45~图 9-48 所示。

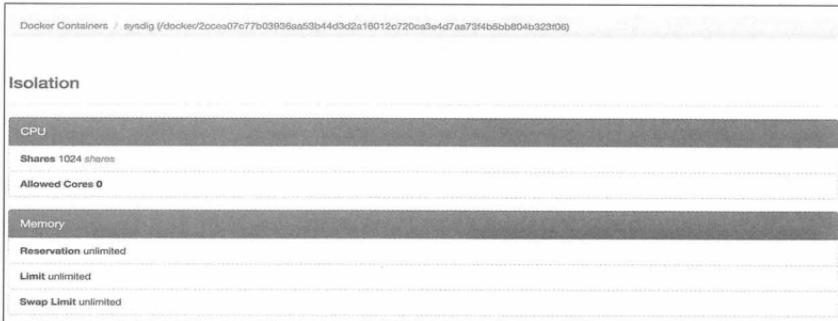


图 9-45

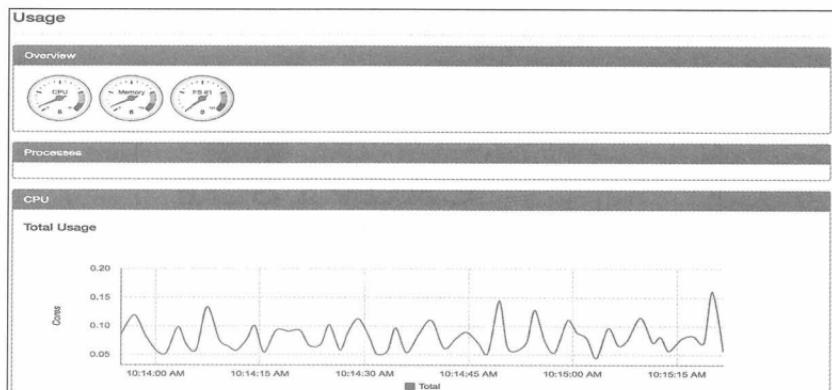


图 9-46

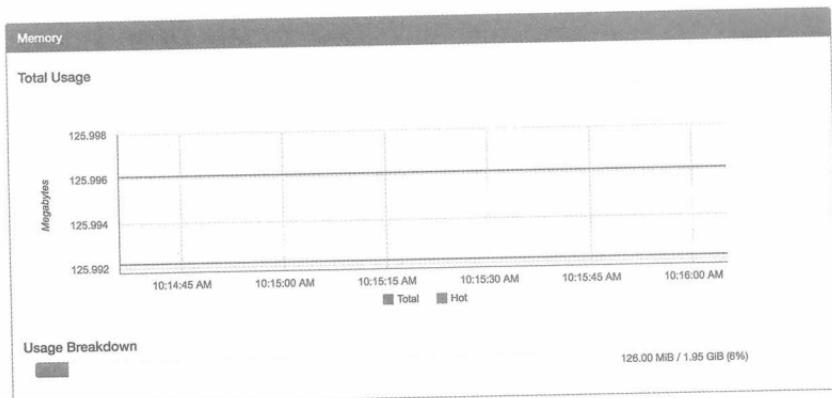


图 9-47

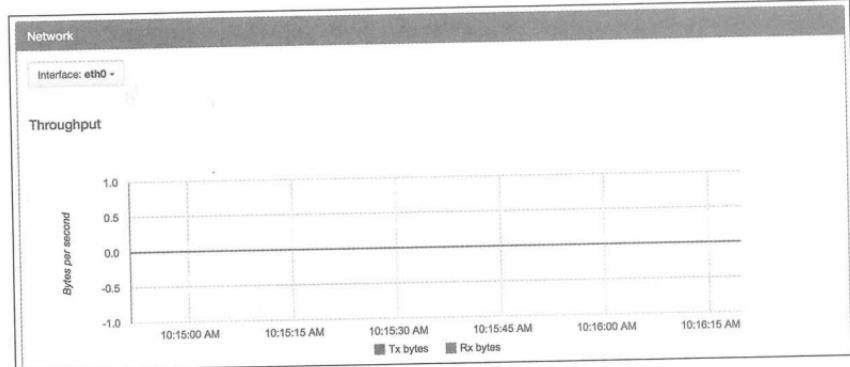


图 9-48

以上就是 cAdvisor 的主要功能，总结起来主要两点：

- (1) 展示 Host 和容器两个层次的监控数据。
- (2) 展示历史变化数据。

由于 cAdvisor 提供的操作界面略显简陋，而且需要在不同页面之间跳转，并且只能监控一个 host，这不免会让人质疑它的实用性。但 cAdvisor 的一个亮点是它可以将监控到的数据导出给第三方工具，由这些工具进一步加工处理。

我们可以把 cAdvisor 定位为一个监控数据收集器，收集和导出数据是它的强项，而非展示数据。

cAdvisor 支持很多的第三方工具，其中就包括我们接下来要重点介绍的 Prometheus。

9.5 Prometheus

Prometheus 是一个非常优秀的监控工具。准确地说，应该是监控方案。Prometheus 提供了监控数据搜集、存储、处理、可视化和告警一套完整的解决方案。

让我们先来看看 Prometheus 的架构。

9.5.1 架构

Prometheus 架构如图 9-49 所示。

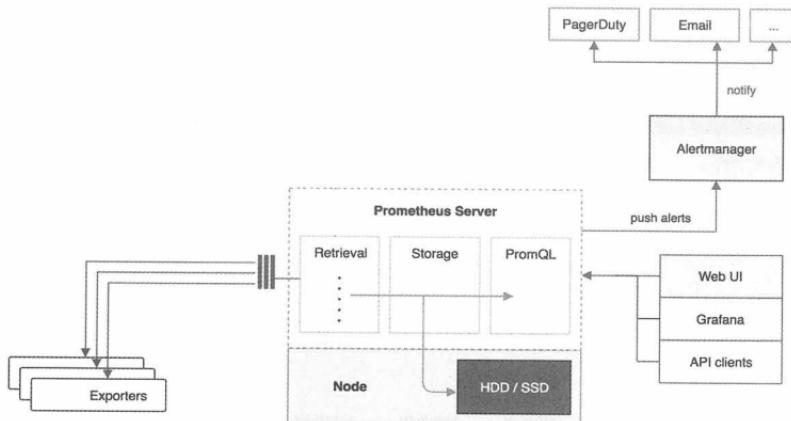


图 9-49

官网上的原始架构图比上面这张要复杂一些，为了避免注意力分散，我只保留了最重要的组件。

1. Prometheus Server

Prometheus Server 负责从 Exporter 拉取和存储监控数据，并提供一套灵活的查询语言（PromQL）供用户使用。

2. Exporter

Exporter 负责收集目标对象（host、container 等）的性能数据，并通过 HTTP 接口供 Prometheus Server 获取。

3. 可视化组件

监控数据的可视化展现对于一个监控方案至关重要，以前 Prometheus 自己开发了一套工

具，不过后来废弃了，因为开源社区出现了更为优秀的产品 Grafana。Grafana 能够与 Prometheus 无缝集成，提供完美的数据展示能力。

4. Alertmanager

用户可以定义基于监控数据的告警规则，规则会触发告警。一旦 Alertmanager 收到告警，会通过预定义的方式发出告警通知。支持的方式包括 Email、PagerDuty、Webhook 等。

可能对于熟悉其他监控方案的同学看了 Prometheus 的架构图会不以为然，“这些功能 Zabbix、Graphite、Nagios 这类监控系统也都有，没什么特别的啊！”。

Prometheus 最大的亮点和先进性来自它的多维数据模型。

9.5.2 多维数据模型

我们先来看一个例子。

比如要监控容器的内存使用情况，现在我们有一个容器 webapp1，最传统和典型的方法是定义一个指标 container_memory_usage_bytes_webapp1 来记录 webapp1 的内存使用数据。假如每 1 分钟取一次样，那么在数据库里就会有类似表 9-1 所示的记录。

表 9-1 内存使用情况

time	container.memory.usage.bytes.webapp1
00:01:00	37738736
00:02:00	37736822
00:03:00	37723425
.....

好，现在需求发生变化，我们需要知道所有 webapp 容器的内存使用情况。如果还是采用前面的方法，就需要增加新的指标 container_memory_usage_bytes_webapp2、container_memory_usage_bytes_webapp3 等。

像 Graphite 这类更高级的监控方案采用了更优雅的层次化数据模型。为了满足上面的需求，Graphite 会定义指标 container.memory_usage_bytes.webapp1、container.memory_usage_bytes.webapp2、container.memory_usage_bytes.webapp3 等。

然后就可以用 container.memory_usage_bytes.webapp* 获取所有的 webapp 的内存使用数据。

此外，Graphite 还支持 sum() 等函数对指标进行计算和处理，比如 sum(container.memory_usage_bytes.webapp*) 可以得到所有 webapp 容器占用的总内存量。

目前为止问题处理得都很好。但客户总是会提出更多的需求：现在不仅要按容器名字统计内存使用量，还要按镜像来统计；或者想对比一下某一组容器在生产环境和测试环境中对内存的不同使用情况。

当然你可以说：只要定义更多的指标就能满足这些需求。比如 container.memory_usage_bytes.image1.webapp1、container.memory_usage_bytes.webapp1.prod 等。

但问题在于，我们没办法提前预知客户要用这些数据回答怎样的问题，所以我们没办法提前定义好所有的指标。

下面来看看 Prometheus 的解决方案。

Prometheus 只需要定义一个全局的指标 `container_memory_usage_bytes`，然后通过添加不同的维度数据来满足不同的业务需求。

比如对于前面 `webapp1` 的三条取样数据，转换成 Prometheus 多维数据将变成如表 9-2 所示的数据。

表 9-2 Prometheus 多维数据

time	containermemoryusagebytes	containername	image	env
00:01:00	37738736	webapp1	mycom/webapp:1.2	prod
00:02:00	37736822	webapp1	mycom/webapp:1.2	prod
00:03:00	37723425	webapp1	mycom/webapp:1.2	prod
.....

这里 `container_name`、`image`、`env` 就是数据的三个维度。想象一下，如果不同 `env` (`prod`、`test`、`dev`)、不同 `image` (`mycom/webapp:1.2`、`mycom/webapp:1.3`) 的容器，它们的内存使用数据中标注了这三个维度信息，那么将能满足很多业务需求，比如：

- (1) 计算 `webapp2` 的平均内存使用情况: `avg(containermemoryusagebytes{containername="webapp2"})`。
- (2) 计算运行 `mycom/webapp:1.3` 镜像的所有容器内存使用总量: `sum(containermemoryusage_bytes{image="mycom/webapp:1.3"})`。
- (3) 统计不同运行环境中 `webapp` 容器内存使用总量: `sum(containermemoryusagebytes{containername=~"webapp"}) by (env)`。

这里只列了几个例子，不过已经能够说明 Prometheus 数据模型的优势了：

- (1) 通过维度对数据进行说明，附加更多的业务信息，进而满足不同业务的需求。同时维度是可以动态添加的，比如再给数据加上一个 `user` 维度，就可以按用户来统计容器内存使用量了。
- (2) Prometheus 丰富的查询语言能够灵活、充分地挖掘数据的价值。前面示例中的 `avg`、`sum`、`by` 只是查询语言中很小的一部分功能，但已经为我们展现了 Prometheus 对多维数据进行分片、聚合的强大能力。

9.5.3 实践

好了，说了这么多 Prometheus 厉害的地方，但技术最终还是需要落地的。接下来我们将演示如何快速搭建 Prometheus 监控系统。

1. 环境说明

我们将通过 Prometheus 监控两台 Docker Host: 192.168.56.102 和 192.168.56.103，监控

host 和容器两个层次的数据。

按照架构图，我们需要运行如下组件：

(1) Prometheus Server，Prometheus Server 本身也将以容器的方式运行在 host 192.168.56.103 上。

(2) Exporter，Prometheus 有很多现成的 Exporter，完整列表请参考 <https://prometheus.io/docs/instrumenting/exporters/>。

我们将使用：

- Node Exporter：负责收集 host 硬件和操作系统数据，它将以容器方式运行在所有 host 上。
- cAdvisor：负责收集容器数据，它将以容器方式运行在所有 host 上。

(3) Grafana，显示多维数据，Grafana 本身也将以容器方式运行在 host 192.168.56.103 上。

2. 运行 Node Exporter

在两个 host 上执行如下命令：

```
docker run -d -p 9100:9100 \ -v "/proc:/host/proc" \ -v
"/sys:/host/sys" \ -v "/:/rootfs" \ --net=host \ prom/node-exporter \
collector.procfs /host/proc \ -collector.sysfs /host/sys \ -
collector.filesystem.ignored-mount-points
"^{/(sys|proc|dev|host|etc)($|/)}"
```

注意，这里我们使用了 `--net=host`，这样 Prometheus Server 可以直接与 Node Exporter 通信。

Node Exporter 启动后，将通过 9100 提供 host 的监控数据。在浏览器中通过 <http://192.168.56.102:9100/metrics> 测试一下，如图 9-50 所示。

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 4.3606e-05
go_gc_duration_seconds{quantile="0.25"} 0.0311e-05
go_gc_duration_seconds{quantile="0.5"} 5.312500000000004e-05
go_gc_duration_seconds{quantile="0.75"} 5.597500000000005e-05
go_gc_duration_seconds{quantile="1"} 0.001130708
go_gc_duration_seconds_sum 0.7822576750000001
go_gc_duration_seconds_count 10799
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 10
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 3.384256e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 3.0642037648e+10
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.647338e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 3.03129253e+08
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 440320
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 3.384256e+06
```

图 9-50

3. 运行 cAdvisor

在两个 host 上执行如下命令：

```
docker run \ --volume=/:/rootfs:ro \ --volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \ --volume=/var/lib/docker/:/var/lib/docker:ro \
-p 8080:8080 \ --detach=true \ --name=cadvisor \ --net=host \
google/cadvisor:latest
```

注意，这里我们使用了 `--net=host`，这样 Prometheus Server 可以直接与 cAdvisor 通信。

Node Exporter 启动后，将通过 8080 提供 host 的监控数据。在浏览器中通过 `http://192.168.56.102:8080/metrics` 测试一下，如图 9-51 所示。



```
← ⌂ ① 192.168.56.102:8080/metrics

# HELP cadvisor_version_info A metric with a constant '1' value labeled by kernel version, OS version, docker version, os type and cAdvisor version.
# TYPE cadvisor_version_info gauge
cadvisor_version_info{advisorVersion="ae6934c",cadvisorVersion="v0.24.1",dockerVersion="1.13.0",kernelVersion="4.4.0-31-generic",osType="Linux"} 1
# HELP container_cpu_system_seconds_total Cumulative system cpu time consumed in seconds.
# TYPE container_cpu_system_seconds_total counter
container_cpu_system_seconds_total{id="/" } 23202.86
container_cpu_system_seconds_total{id="/docker" } 10007.38
container_cpu_system_seconds_total{id="/init.scope" } 377.25
container_cpu_system_seconds_total{id="/system.slice" } 12206.4
container_cpu_system_seconds_total{id="/system.slice/accounts-daemon.service" } 3.24
container_cpu_system_seconds_total{id="/system.slice/apparmor.service" } 0
container_cpu_system_seconds_total{id="/system.slice/apparmor.service" } 0
container_cpu_system_seconds_total{id="/system.slice/apport.service" } 0
container_cpu_system_seconds_total{id="/system.slice/std.service" } 0
container_cpu_system_seconds_total{id="/system.slice/cmanager.service" } 0
container_cpu_system_seconds_total{id="/system.slice/cgroups-mount.service" } 0
container_cpu_system_seconds_total{id="/system.slice/console-setup.service" } 0
container_cpu_system_seconds_total{id="/system.slice/cron.service" } 6.13
container_cpu_system_seconds_total{id="/system.slice/dbus.service" } 2.54
container_cpu_system_seconds_total{id="/system.slice/dev-disk-by\x2did-dm\x2dname\x2dubuntu\x2d\x2dvg\x2dswap_1.swap" } 0
container_cpu_system_seconds_total{id="/system.slice/dev-disk-by\x2did-dm\x2duuid\x2dLVM\x2dM10hT0550VgT69Fru0UL4ow" } 0
container_cpu_system_seconds_total{id="/system.slice/dev-disk-by\x2duuid-d5594b06\x2debaa\x2d4de9\x2dbb58\x2dbc4fa1" } 0
container_cpu_system_seconds_total{id="/system.slice/dev-dm\x2d1.swap" } 0
container_cpu_system_seconds_total{id="/system.slice/dev-mapper-ubuntu\x2d\x2dvg\x2dswap_1.swap" } 0
container_cpu_system_seconds_total{id="/system.slice/dev-ubuntu\x2dvg\x2dswap_1.swap" } 0
container_cpu_system_seconds_total{id="/system.slice/docker.service" } 11678.76
container_cpu_system_seconds_total{id="/system.slice/ebtables.service" } 0
```

图 9-51

4. 运行 Prometheus Server

在 host 192.168.56.103 上执行如下命令：

```
docker run -d -p 9090:9090 \ -v
/root/prometheus.yml:/etc/prometheus/prometheus.yml \ --name prometheus
\ --net=host \ prom/prometheus
```

注意，这里我们使用了 `--net=host`，这样 Prometheus Server 可以直接与 Exporter 和 Grafana 通信。

`prometheus.yml` 是 Prometheus Server 的配置文件，如图 9-52 所示。

```

global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

  # Attach these labels to any time series on clients when communicating with
  # external systems (federation, remote storage, alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

  # Told rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  # - 'first.rules'
  # - 'second.rules'

  # A scrape configuration containing exactly one endpoint to scrape.
  # Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label job_name to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'

    static_configs:
      - targets: ['localhost:9090', 'localhost:8080', 'localhost:9100', '192.168.56.102:8080', '192.168.56.102:9100']

```

图 9-52

最重要的配置是：

```

static_configs: - targets:
['localhost:9090','localhost:8080','localhost:9100','192.168.56.102:8080
','192.168.56.102:9100']

```

指定从哪些 exporter 抓取数据。这里指定了两台 host 上的 Node Exporter 和 cAdvisor。

另外 localhost:9090 就是 Prometheus Server 自己，可见 Prometheus 本身也会收集自己的监控数据。同样地，我们也可以通过 <http://192.168.56.103:9090/metrics> 测试一下，如图 9-53 所示。

```

< -> C ⌂ ① 192.168.56.103:9090/metrics
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc.duration_seconds{quantile="0"} 1.3466000000000001e-05
go_gc.duration_seconds{quantile="0.25"} 4.9418e-05
go_gc.duration_seconds{quantile="0.5"} 9.112600000000001e-05
go_gc.duration_seconds{quantile="0.75"} 0.00010904600000000002
go_gc.duration_seconds{quantile="1"} 0.00018475700000000001
go_gc.duration.seconds.sum 0.003973357
go_gc.duration.seconds.count 47
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 82
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats.alloc.bytes 9.3808640007
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats.alloc.bytes.total 2.526375744e+09
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats.buck.hash.sys.bytes 1.589178e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter

```

图 9-53

在浏览器中打开 `http://192.168.56.103:9090`，单击菜单 Status→Targets，如图 9-54、图 9-55 所示。



图 9-54

Targets				
prometheus				
Endpoint	State	Labels	Last Scrape	
http://localhost:2000/metrics	UP	instances="localhost:2000"	10.96s ago	
http://localhost:2000/metrics	UP	instances="localhost:2000"	1.725s ago	
http://localhost:3100/metrics	UP	instances="localhost:3100"	7.582s ago	
http://192.168.56.102:8080/metrics	UP	instances="192.168.56.102:8080"	1.027s ago	
http://192.168.56.102:5100/metrics	UP	instances="192.168.56.102:5100"	6.381s ago	

图 9-55

所有 Target 的 State 都是 UP，说明 Prometheus Server 能够正常获取监控数据。

5. 运行 Grafana

在 host 192.168.56.103 上执行如下命令：

```
docker run -d -i -p 3000:3000 \ -e
"GF_SERVER_ROOT_URL=http://grafana.server.name" \ -e
"GF_SECURITY_ADMIN_PASSWORD=secret" \ --net=host \ grafana/grafana
```

注意，这里我们使用了 `--net=host`，这样 Grafana 可以直接与 Prometheus Server 通信。
`-e "GF_SECURITY_ADMIN_PASSWORD=secret"` 指定了 Grafana admin 用户密码 secret。
Grafana 启动后。在浏览器中打开 `http://192.168.56.103:3000/`，如图 9-56 所示。



图 9-56

登录后，Grafana 将引导我们配置 Data Source，如图 9-57 所示。

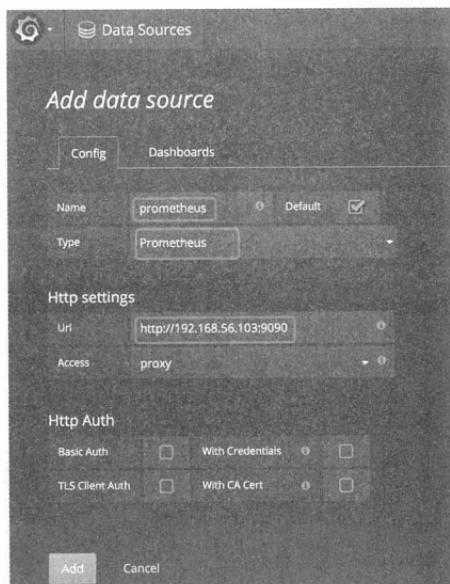


图 9-57

Name 为 Data Source 命名，例如 prometheus。

Type 选择 Prometheus，如图 9-58 所示。

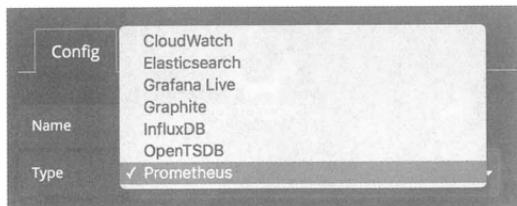


图 9-58

Url 输入 Prometheus Server 的地址 <http://192.168.56.103:9090>。

其他保持默认值，单击 Add。

如果一切顺利，Grafana 应该已经能够访问 Prometheus 中存放的监控数据了，那么如何展示呢？

Grafana 是通过 Dashboard 展示数据的，在 Dashboard 中需要定义：

- (1) 展示 Prometheus 的哪些多维数据？需要给出具体的查询语言表达式。
- (2) 用什么形式展示？比如二维线性图、仪表图、各种坐标的含义等。

可见，要做出一个 Dashboard 也不是件容易的事情。幸运的是，我们可以借助开源社区的力量，直接使用现成的 Dashboard。

访问 <https://grafana.com/dashboards?dataSource=prometheus&search=docker>，将会看到很多用于监控 Docker 的 Dashboard，如图 9-59 所示。

Dashboard Name	Author	Description	Downloads
Docker and system monitoring	mojofort	A simple overview of the most important Docker host and container metrics. (cAdvisor/Prometheus)	1211
Docker Dashboard	Brian Christner	Docker Monitoring Template	3172
Docker Engine Metrics	basi	Draw some docker metrics	189
Docker Host & Container Overview	uschiwill	A simple overview of the most important Docker host and container metrics. (cAdvisor/Prometheus)	1252
Docker Hub Stats	Brian Christner	Docker Hub image metrics	74
Docker monitoring	philicous	Docker monitoring with Prometheus and cAdvisor	1432

图 9-59

我们可以下载这些现成的 Dashboard，然后 import 到我们的 Grafana 中就可以直接使用了。

比如下载 Docker and system monitoring，得到一个 json 文件，然后单击 Grafana 左上角菜单 Dashboards → Import，如图 9-60 所示。

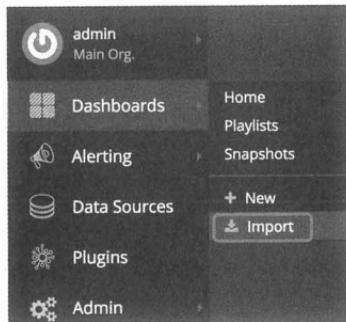


图 9-60

导入我们下载的 json 文件，如图 9-61 所示。

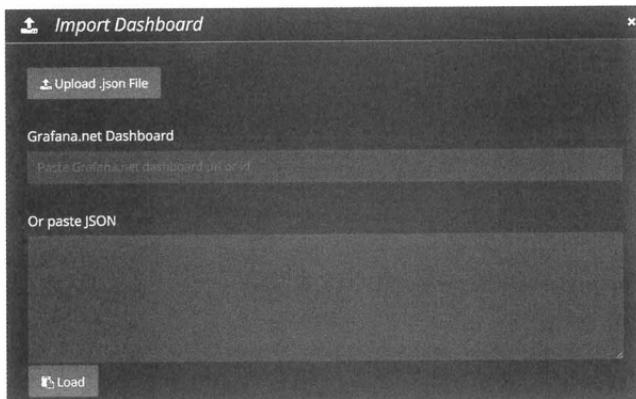


图 9-61

Dashboard 将立刻展示出漂亮的图表，如图 9-62 所示。



图 9-62

在这个 Dashboard 中，上部分是 host 的数据，我们可以通过 Node 切换不同的 host，如图 9-63 所示。



图 9-63

Dashboard 的下半部分展示的是所有的容器监控数据。Grafana 的 Dashboard 是可交互的，我们可以在图表上只显示指定的容器、选取指定的时间区间、重新组织和排列图表、调整刷新频率，功能非常强大。

9.6 比较不同的监控工具

前面我们已经介绍了容器监控的多种工具和方案，是时候做一个比较了。下面将从 4 个方面来考察这些工具之间的优劣。

1. 部署容易度

`ps/top/stats` 无疑是最容易使用的，它们是 Docker 自带的子命令，随时随地都可以用来快速了解容器的状态。其余几种也都可以容器的方式运行，总的来说都不算复杂。相对而言，Prometheus 涉及的组件比较多，搭建整个方案需要运行的容器数量也要多些，部署和管理的难道稍大。

2. 数据详细度

`ps/top/stats` 和 `cAdvisor` 能够监控容器基本的资源使用情况，`Sysdig`、`Weave Scope` 和 `Prometheus` 则能提供更丰富的数据。

3. 多 Host 监控

`Weave Scope` 和 `Prometheus` 可以监控整个集群，而其余的工具只提供单个 Host 的监控能力。

4. 告警功能

只有 `Prometheus` 具备原生的告警功能。

5. 监控非容器资源

`Sysdig`、`Weave Scope` 和 `cAdvisor` 可以监控到 Host 操作系统的状态，而 `Prometheus` 则可以通过 Exporter 支持应用级别的监控，比如监控 `ceph`、`haproxy` 等。

4 个方面的比较结果如表 9-3 所示。

表 9-3 4 个方面的比较结果

	Docker <code>ps/top/stats</code>	Sysdig	Weave Scope	<code>cAdvisor</code>	<code>Prometheus</code>
部署容易度	*****	****	***	*****	***
数据详细度	***	*****	*****	***	*****
多 Host 监控	none	none	****	none	****
告警功能	none	none	none	none	****
监控非容器资源	none	***	***	**	*****

9.7 几点建议

(1) Docker `ps/top/stats` 最适合快速了解容器运行状态，从而判断是否需要进一步分析和排查。

(2) Sysdig 提供了丰富的分析和挖掘功能，是 Troubleshooting 的神器。

(3) cAdvisor 一般不会单独使用，通常作为其他监控工具的数据收集器，比如 Prometheus。

(4) Weave Scope 流畅简洁的操控界面是其最大亮点，而且支持直接在 Web 界面上执行命令。

(5) Prometheus 的数据模型和架构决定了它几乎具有无限的可能性。Prometheus 和 Weave Scope 都是优秀的容器监控方案。除此之外，Prometheus 还可以监控其他应用和系统，更为综合和全面。

(6) 监控系统的选择，并不是一道单选题，应该根据需求和实际情况搭配组合，优势互补。除了这里介绍的 5 种工具和方案，监控领域还有很多选项，也都可以考虑。

第 10 章

◀ 日志管理 ▶

高效的监控和日志管理对保持生产系统持续稳定的运行以及排查问题至关重要。

在微服务架构中，由于容器的数量众多以及快速变化的特性使得记录日志和监控变得越来越重要。考虑到容器短暂和不固定的生命周期，当我们需要 debug 问题时有些容器可能已经不存在了。因此，一套集中式的日志管理系统是生产环境中不可或缺的组成部分。

本章我们将讨论监控容器的各种可用技术和方案，首先会介绍 Docker 自带的 logs 子命令，然后讨论 Docker 的 logging driver，接下来通过实践学习几个已经广泛应用的日志管理方案：ELK、Fluentd 和 Graylog，如图 10-1 所示。

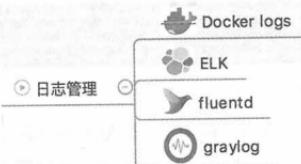


图 10-1

10.1 Docker logs

我们首先来看一看默认配置下 Docker 的日志功能。

对于一个运行的容器，Docker 会将日志发送到容器的标准输出设备（STDOUT）和标准错误设备（STDERR），STDOUT 和 STDERR 实际上就是容器的控制台终端。

举个例子，用下面的命令运行 httpd 容器，结果如图 10-2 所示。

```
root@ubuntu:~# docker run -p 80:80 httpd
```

```
root@ubuntu:~# docker run -p 80:80 httpd
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set the 'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set the 'ServerName' directive globally to suppress this message
[Thu May 04 06:52:35 2017] [mpm_event:notice] [pid 1:tid 139845529798528] AH00489: Apache/2.4.25 (Unix) configured -- resuming normal operations
[Thu May 04 06:52:35 2017] [core:notice] [pid 1:tid 139845529798528] AH00054: Command line: "httpd -D FOREGROUND"
```

图 10-2

因为我们在启动日志的时候没有用 -d 参数，httpd 容器以前台方式启动，日志会直接打印在当前的终端窗口。

如果加上 -d 参数以后台方式运行容器，我们就看不到输出的日志了，如图 10-3 所示。

```
root@ubuntu:~# docker run -p 80:80 -d httpd
21ea55e7b0042058603b90896edd2c28f6487780dfdb092acf054c6b50d87f52
root@ubuntu:~#
```

图 10-3

这种情况下如果要查看容器的日志，有两种方法：

- (1) attach 到该容器。
- (2) 用 docker logs 命令查看日志。

先来看 attach 的方法。运行 docker attach 命令，如图 10-4 所示。

```
root@ubuntu:~# docker attach 21ea55e7b004
```

图 10-4

attach 到了 httpd 容器，但并没有任何输出，这是因为当前没有新的日志信息。

为了产生一条新的日志，可以在 host 的另一个命令行终端执行 curl localhost，如图 10-5 所示。

```
root@ubuntu:~# curl localhost
<html><body><h1>It works!</h1></body></html>
root@ubuntu:~#
```

图 10-5

这时，attach 的终端就会打印出新的日志，如图 10-6 所示。

```
root@ubuntu:~# docker attach 21ea55e7b004
172.17.0.1 - - [04/May/2017:07:07:13 +0000] "GET / HTTP/1.1" 200 45
```

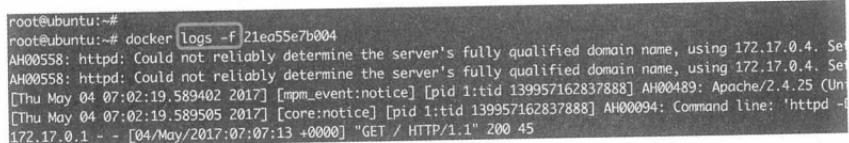
图 10-6

attach 的方法在实际使用中不太方便，因为：

- (1) 只能看到 attach 之后的日志，以前的日志不可见。

(2) 退出 attach 状态比较麻烦（按 Ctrl+p 键，然后再按 Ctrl+q 键），一不小心很容易将容器杀掉（比如按 Ctrl+C）。

查看容器日志推荐的方法是用 docker logs 命令，如图 10-7 所示。



```
root@ubuntu:~# docker logs -f 21ea55e7b004
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set
[Thu May 04 07:02:19.589402 2017] [mpm_event:notice] [pid 1:tid 139957162837888] AH000489: Apache/2.4.25 (Ubuntu)
[Thu May 04 07:02:19.589505 2017] [core:notice] [pid 1:tid 139957162837888] AH00094: Command line: 'httpd -f 172.17.0.1 - - [04/May/2017:07:07:13 +0000] "GET / HTTP/1.1" 200 45
```

图 10-7

docker logs 能够打印出自容器启动以来完整的日志，并且 -f 参数可以继续打印出新产生的日志，效果与 Linux 命令 tail -f 一样。

10.2 Docker logging driver

将容器日志发送到 STDOUT 和 STDERR 是 Docker 的默认日志行为。实际上，Docker 提供了多种日志机制帮助用户从运行的容器中提取日志信息，这些机制被称作 logging driver。

Docker 的默认 logging driver 是 json-file。

```
docker info |grep 'Logging Driver'
Logging Driver: json-file
```

如果容器在启动时没有特别指明，就会使用这个默认的 logging driver。

json-file 会将容器的日志保存在 json 文件中，Docker 负责格式化其内容并输出到 STDOUT 和 STDERR。

我们可以在 Host 的容器目录中找到这个文件，容器路径为 /var/lib/docker/containers/<container ID>/<container ID>.json.log。

比如我们可以查看前面 httpd 容器 json 格式的日志文件，如图 10-8 所示。

```
root@ubuntu:~#
root@ubuntu:~# cat /var/lib/docker/containers/21ea55e7b0042058603b90896edd2c28f6487780dfdb092acf054c6b50d87f52/21ea55e7b0042058603b90896edd2c28f6487780dfdb092acf054c6b50d87f52-json.log
{"log":"AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set the 'ServerName' directive globally to suppress this message\n","stream":"stderr","time":"2017-05-04T07:02:19.586904895Z"}
{"log":"AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set the 'ServerName' directive globally to suppress this message\n","stream":"stderr","time":"2017-05-04T07:02:19.591750443Z"}
{"log": "[Thu May 04 07:02:19.589402 2017] [mpm_event:notice] [pid 1:tid 139957162837888] AH00489: Apache/2.4.25 (Unix) configured -- resuming normal operations\n","stream":"stderr","time":"2017-05-04T07:02:19.591772001Z"}
{"log": "[Thu May 04 07:02:19.589505 2017] [core:notice] [pid 1:tid 139957162837888] AH00094: Command line: \"httpd -D FOREGROUND\"\n","stream":"stdout","time":"2017-05-04T07:02:19.591774447Z"}
{"log": "172.17.0.1 - - [04/May/2017:07:07:13 +0000] \"GET / HTTP/1.1\" 200 45\n","stream":"stdout","time":"2017-05-04T07:07:13.911477837Z"}
root@ubuntu:~#
```

图 10-8

可以看到 5 条日志记录。

除了 json-file，Docker 还支持多种 logging driver。完整列表可访问官方文档 <https://docs.docker.com/engine/admin/logging/overview/#supported-logging-drivers>，如图 10-9 所示。

Driver	Description
none	No logs will be available for the container and <code>docker logs</code> will not return any output.
json-file	The logs are formatted as JSON. The default logging driver for Docker.
syslog	Writes logging messages to the <code>syslog</code> facility. The <code>syslog</code> daemon must be running on the host machine.
journald	Writes log messages to <code>journald</code> . The <code>journald</code> daemon must be running on the host machine.
gelf	Writes log messages to a Graylog Extended Log Format (GELF) endpoint such as Graylog or Logstash.
fluentd	Writes log messages to <code>fluentd</code> (forward input). The <code>fluentd</code> daemon must be running on the host machine.
awslogs	Writes log messages to Amazon CloudWatch Logs.
splunk	Writes log messages to <code>splunk</code> using the HTTP Event Collector.
etwlogs	Writes log messages as Event Tracing for Windows (ETW) events. Only available on Windows platforms.
gcplogs	Writes log messages to Google Cloud Platform (GCP) Logging.

图 10-9

none 是 disable 容器日志功能。

syslog 和 journald 是 Linux 上的两种日志管理服务。

awslogs、splunk 和 gcplogs 是第三方日志托管服务。

gelf 和 fluentd 是两种开源的日志管理方案，我们会在后面分别讨论。

容器启动时可以通过 `--log-driver` 指定使用的 logging driver。如果要设置 Docker 默认的 logging driver，需要修改 Docker daemon 的启动脚本，指定 `--log-driver` 参数，比如：

```
ExecStart=/usr/bin/dockerd -H fd:// --log-driver=syslog --log-opt .....
```

每种 logging driver 都有自己的 `--log-opt`，使用时请参考官方文档。

10.3 ELK

在开源的日志管理方案中，最出名的莫过于 ELK 了。ELK 是三个软件的合称：Elasticsearch、Logstash、Kibana。

1. Elasticsearch

一个近乎实时查询的全文搜索引擎。Elasticsearch 的设计目标就是要能够处理和搜索巨量的日志数据。

2. Logstash

读取原始日志，并对其进行分析和过滤，然后将其转发给其他组件（比如 Elasticsearch）进行索引或存储。Logstash 支持丰富的 Input 和 Output 类型，能够处理各种应用的日志。

3. Kibana

一个基于 JavaScript 的 Web 图形界面程序，专门用于可视化 Elasticsearch 的数据。Kibana 能够查询 Elasticsearch 并通过丰富的图表展示结果。用户可以创建 Dashboard 来监控系统的日志。

本节将讨论如何用 ELK 这组黄金搭档来监控 Docker 容器的日志。

10.3.1 日志处理流程

图 10-10 展示了 Docker 部署环境下典型的 ELK 日志处理流程：

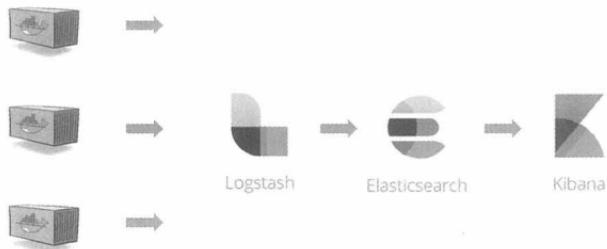


图 10-10

Logstash 负责从各个 Docker 容器中提取日志，Logstash 将日志转发到 Elasticsearch 进行索引和保存，Kibana 分析和可视化数据。

下面开始实践这套流程。

10.3.2 安装 ELK 套件

ELK 的部署方案可以非常灵活，在规模较大的生产系统中，ELK 有自己的集群，实现了高可用和负载均衡。我们的目标是在最短的时间内学习并实践 ELK，因此将采用最小部署方案：在容器中搭建 ELK。

```
docker run -p 5601:5601 -p 9200:9200 -p 5044:5044 -it --name elk
sebp/elk
```

我们使用的是 sebp/elk 这个现成的 image，里面已经包含了整个 ELK stack。容器启动后 ELK 各组件将分别监听如下端口：

- 5601: Kibana web 接口。
- 9200: Elasticsearch JSON 接口。
- 5044: Logstash 日志接收接口。

先访问一下 Kibana [http://\[Host IP\]:5601/](http://[Host IP]:5601/) 看看效果，如图 10-11 所示。

图 10-11

当前 Kibana 没有可显示的数据，因为当前 Elasticsearch 还没有任何日志数据。

访问一下 Elasticsearch 的 JSON 接口 `http://[Host IP]:9200/_search?pretty`，如图 10-12 所示。

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1.0,
    "hits": [
      {
        "_index": ".kibana",
        "_type": "config",
        "_id": "5.3.0",
        "_score": 1.0,
        "_source": {
          "buildNum": 14823
        }
      }
    ]
  }
}
```

图 10-12

确实，目前 Elasticsearch 没有与日志相关的 index。

接下来我们的工作就是将 Docker 的日志导入 ELK。

10.3.3 Filebeat

几乎所有的软件和应用都有自己的日志文件，容器也不例外。前面我们已经知道 Docker 会将容器日志记录到 /var/lib/docker/containers/<container ID>/<container ID>-json.log，那么只要我们能够将此文件发送给 ELK 就可以实现日志管理。

要实现这一步其实不难，因为 ELK 提供了一个配套小工具 Filebeat，它能将指定路径下的日志文件转发给 ELK。同时 Filebeat 很聪明，它会监控日志文件，当日志更新时，Filebeat 会将新的内容发送给 ELK。

1. 安装 Filebeat

下面在 Docker Host 中安装和配置 Filebeat。

```
curl -L -O
https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-5.4.0-
amd64.deb
sudo dpkg -i filebeat-5.4.0-amd64.deb
```

当你看到这篇文章时，Filebeat 可能已经有了更新的版本，请参考最新的安装文档 <https://www.elastic.co/guide/en/beats/filebeat/current/filebeat-installation.html>。

2. 配置 Filebeat

Filebeat 的配置文件为 /etc/filebeat/filebeat.yml，我们需要告诉 Filebeat 两件事：

- 监控哪些日志文件？
- 将日志发送到哪里？

首先回答第一个问题，如图 10-13 所示。

```
- input_type: log
  # Paths that should be crawled and fetched. Glob-based paths.
  paths:
    - /var/lib/docker/containers/*/*.log
    - /var/log/syslog
```

图 10-13

在 paths 中我们配置了两条路径：

- (1) /var/lib/docker/containers/*/*.log 是所有容器的日志文件。
- (2) /var/log/syslog 是 Host 操作系统的 syslog。

接下来告诉 Filebeat 将这些日志发送给 ELK。

Filebeat 可以将日志发送给 Elasticsearch 进行索引和保存；也可以先发送给 Logstash 进行分析和过滤，然后由 Logstash 转发给 Elasticsearch。

为了不引入过多的复杂性，我们这里将日志直接发送给 Elasticsearch，如图 10-14 所示。

```
#----- Elasticsearch output -----
output.elasticsearch:
  # Array of hosts to connect to.
  hosts: ["localhost:9200"]

  # Optional protocol and basic auth credentials.
  #protocol: "https"
  #username: "elastic"
  #password: "changeme"

#----- Logstash output -----
output.logstash:
  # The Logstash hosts
  hosts: ["localhost:5044"]

  # Optional SSL. By default is off.
  # List of root certificates for HTTPS server verifications
  #ssl_certificateAuthorities: ["/etc/pki/root/ca.pem"]

  # Certificate for SSL client authentication
  #ssl_certificate: "/etc/pki/client/cert.pem"
```

图 10-14

如果要发送给 Logstash，可参考后半部分的注释。

当前的日志处理流程如图 10-15 所示。

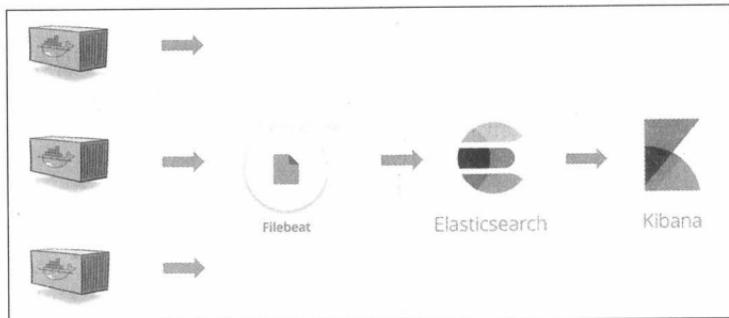


图 10-15

3. 启动 Filebeat

Filebeat 安装时已经注册为 systemd 的服务，可以直接启动服务。

```
systemctl start filebeat.service
```

10.3.4 管理日志

Filebeat 启动后，正常情况下会将监控的日志发送给 Elasticsearch。刷新 Elasticsearch 的 JSON 接口 `http://[Host IP]:9200/_search?pretty` 进行确认，如图 10-16 所示。

```

← → C | ① 192.168.56.101:9200/_search?pretty
{
  "index" : "filebeat-2017.05.05",
  "_type" : "log",
  "_id" : "AVvWiffnB6d9VZQCui4aw",
  "_score" : 1.0,
  "_source" : {
    "@timestamp" : "2017-05-05T02:43:51.656Z",
    "beat" : {
      "hostname" : "ubuntu",
      "name" : "ubuntu",
      "version" : "5.3.2"
    },
    "input_type" : "log",
    "message" : "{\"log\":{\"@version\":1,\"@type\":\"log\",\"@source\":\"/var/lib/docker/containers/1b6fd5647e2d7929ce8882455f38266281e61573caae365d460894d5ba4e664f/1b6fd505T01:55:48.703833097z\"},\"offset\":2064}",
    "offset" : 2064,
    "source" : "/var/lib/docker/containers/1b6fd5647e2d7929ce8882455f38266281e61573caae365d460894d5ba4e664f/1b6fd5json.log",
    "type" : "log"
  }
},
{
  "index" : "filebeat-2017.05.05",
  "_type" : "log",
  "_id" : "AVvWiffnB6d9VZQCui4a2",
  "_score" : 1.0,
  "_source" : {
    "@timestamp" : "2017-05-05T02:43:51.656Z",
    "beat" : {
      "hostname" : "ubuntu",
      "name" : "ubuntu",
      "version" : "5.3.2"
    },
    "input_type" : "log",
    "message" : "Apr 30 06:55:51 ubuntu dhclient[966]: DHCPACK of 192.168.56.101 from 192.168.56.100",
    "offset" : 2014,
    "source" : "/var/log/syslog",
    "type" : "log"
  }
},

```

图 10-16

这次我们能够看到 `filebeat-*` 的 index，以及 Filebeat 监控的那两个路径下的日志。好，Elasticsearch 已经创建了日志的索引并保存起来，接下来是在 Kibana 中展示日志。首先需要配置一个 index pattern，即告诉 Kibana 查询和分析 Elasticsearch 中的哪些日志，如图 10-17 所示。

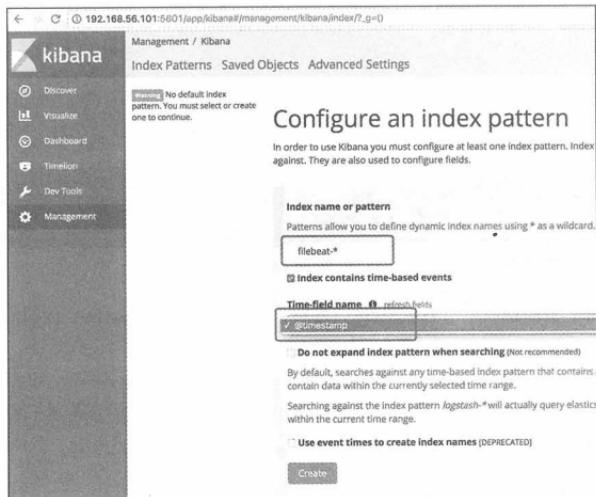


图 10-17

指定 index pattern 为 filebeat-*, 这与 Elasticsearch 中的 index 一致。

Time-field name 选择 @timestamp。

单击 Create 创建 index pattern。

单击 Kibana 左侧 Discover 菜单，便可看到容器和 syslog 日志信息，如图 10-18 所示。



图 10-18

下面我们启动一个新的容器，该容器将向控制台打印信息，模拟日志输出，如图 10-19 所示。

```
docker run busybox sh -c 'while true; do echo "This is a log message from container busybox!"; sleep 10; done;'
```

```
root@ubuntu:~#
root@ubuntu:~# docker run busybox sh -c 'while true; do echo "This is a log message from container busybox!"; sleep 10; done;'
This is a log message from container busybox!
This is a log message from container busybox!
This is a log message from container busybox!
```

图 10-19

刷新 Kibana 页面或者单击右上角的图标，马上就能看到 busybox 的日志，如图 10-20 所示。

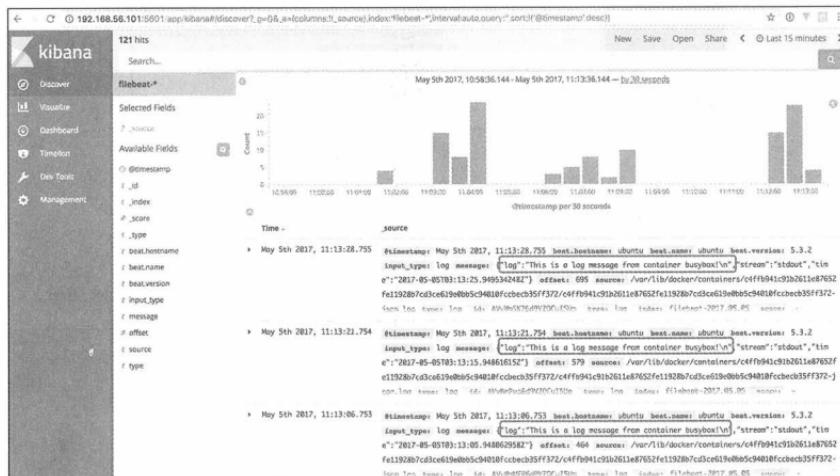


图 10-20

Kibana 也提供了强大的查询功能，比如输入关键字 busybox 能搜索出所有匹配的日志条目，如图 10-21 所示。

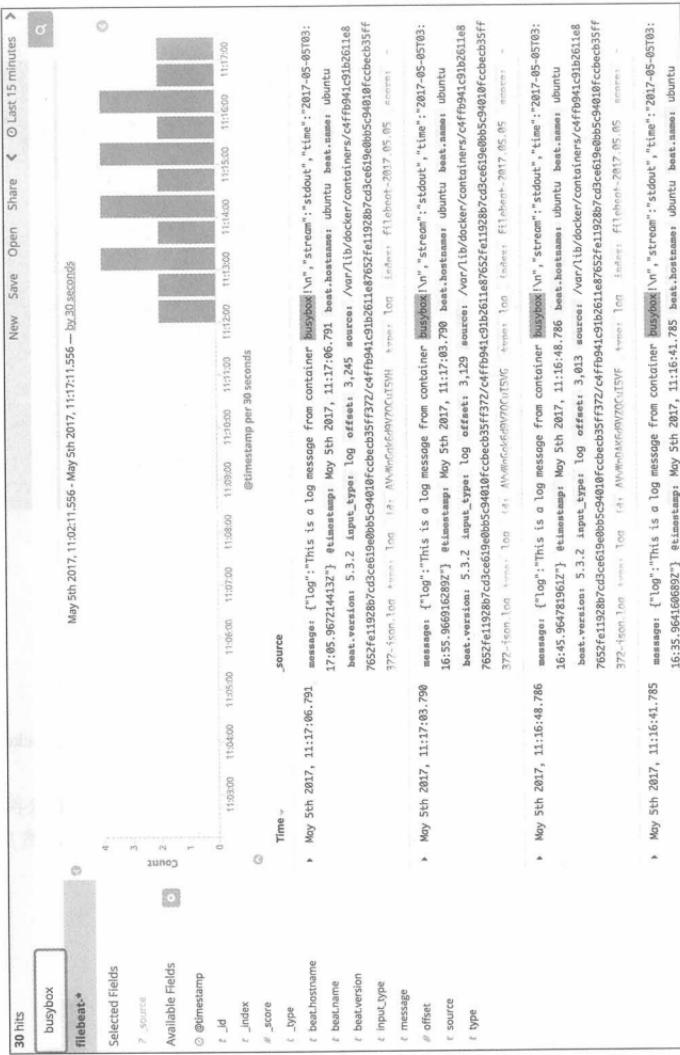


图 10-21

我们这里只是简单地将日志导入 ELK 并朴素地显示出来，实际上 ELK 还可以对日志进行归类汇总、分析聚合、创建炫酷的 Dashboard 等，可以挖掘的内容很多，玩法很丰富。由于这个教程的重点是容器，这里就不过多的展开。下面这张图可以感受一下 ELK 的能力，更多

的功能留给大家自己去探索，如图 10-22 所示。



图 10-22

10.4 Fluentd

前面的 ELK 中我们是用 Filebeat 收集 Docker 容器的日志，利用的是 Docker 默认的 logging driver json-file，本节我们将使用 fluentd 来收集容器的日志。

Fluentd 是一个开源的数据收集器，它目前拥有超过 500 种 plugin，可以连接各种数据源和数据输出组件。在接下来的实践中，Fluentd 会负责收集容器日志，然后发送给 Elasticsearch。日志处理流程如图 10-23 所示。

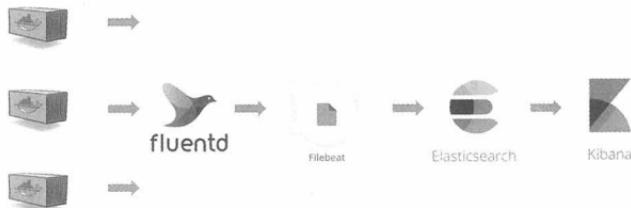


图 10-23

这里用 Filebeat 将 Fluentd 收集到的日志转发给 Elasticsearch。这当然不是唯一的方案，

Fluentd 有一个 plugin fluent-plugin-elasticsearch 可以直接将日志发送给 Elasticsearch。条条道路通罗马，开源世界给予了我们多种可能性，可以根据需要选择合适的方案。

10.4.1 安装 Fluentd

同样的，最高效的实践方式是运行一个 fluentd 容器。

```
docker run -d -p 24224:24224 -p 24224:24224/udp -v /data:/fluentd/log fluentd/fluentd
```

fluentd 会在 TCP/UDP 端口 24224 上接收日志数据，日志将保存在 Host 的 /data 目录中。

10.4.2 重新配置 Filebeat

编辑 Filebeat 的配置文件 /etc/filebeat/filebeat.yml，将 /data 添加到监控路径中，如图 10-24 所示。

```
- input_type: log
  # Paths that should be crawled and fetched. Glob based paths.
  paths:
    - /data/*.log
```

图 10-24

重启 Filebeat。

```
systemctl restart filebeat.service
```

10.4.3 监控容器日志

启动测试容器。

```
docker run -d \ --log-driver=fluentd \ --log-opt fluentd-address=localhost:24224 \ --log-opt tag="log-test-container-A" \ busybox sh -c 'while true; do echo "This is a log message from container A"; sleep 10; done;'
docker run -d \ --log-driver=fluentd \ --log-opt fluentd-address=localhost:24224 \ --log-opt tag="log-test-container-B" \ busybox sh -c 'while true; do echo "This is a log message from container B"; sleep 10; done;'
```

--log-driver=fluentd 告诉 Docker 使用 Fluentd 的 logging driver。

--log-opt fluentd-address=localhost:24224 将容器日志发送到 Fluentd 的数据接收端口。

--log-opt tag="log-test-container-A" 和 --log-opt tag="log-test-container-B" 在日志中添加一个

可选的 tag，用于区分不同的容器。

容器启动后，Kibana 很快就能够查询到容器的日志，如图 10-25 所示。

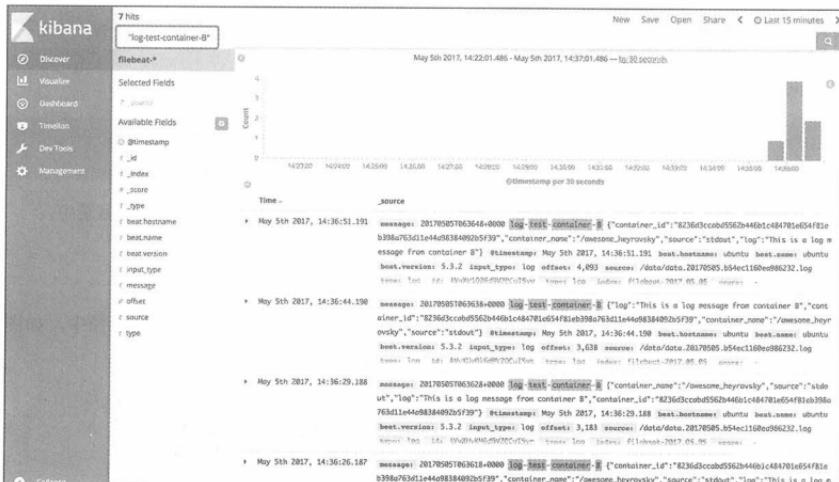


图 10-25

10.5 Graylog

Graylog 是与 ELK 可以相提并论的一款集中式日志管理方案，支持数据收集、检索、可视化 Dashboard。本节将实践用 Graylog 来管理 Docker 日志。

10.5.1 Graylog 架构

Graylog 架构如图 10-26 所示。

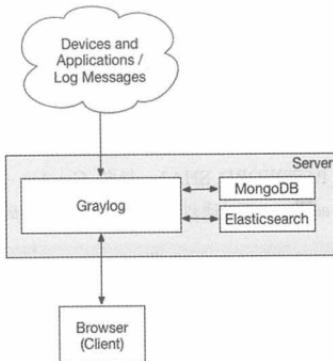


图 10-26

Graylog 负责接收来自各种设备和应用的日志，并为最终用户提供 Web 访问接口。

Elasticsearch 用于索引和保存 Graylog 接收到的日志。

MongoDB 负责保存 Graylog 自身的配置信息。

与 ELK 一样，Graylog 的部署方案很灵活，快速搭建一个 all-in-one 的环境对于学习很有益处；部署一个高可用高伸缩性的集群对于生成环境也是必要的。

接下来我们将在容器环境下搭建 Graylog。

10.5.2 部署 Graylog

Graylog 及其相关组件都将以容器的方式部署。

1. MongoDB

```
docker run --name graylog-mongo -d mongo:3
```

2. Elasticsearch

```
docker run --name graylog-elasticsearch -d elasticsearch:2
elasticsearch -Des.cluster.name="graylog"
```

3. Graylog

```
docker run --link graylog-mongo:mongo \ --link graylog-
elasticsearch:elasticsearch \ -p 9000:9000 \ -p 12201:12201/udp \ -e
GRAYLOG_WEB_ENDPOINT_URI="http://192.168.56.101:9000/api" \ -e
GRAYLOG_PASSWORD_SECRET=somepasswordpepper \ -e
GRAYLOG_ROOT_PASSWORD_SHA2=8c6976e5b5410415bd908bd4dee15dfb167a9c873fc4
bb8a81f6f2ab448a918 \ -d graylog2/server
```

- `--link`: 让 Graylog 容器能够用主机名 `mongo` 和 `elasticsearch` 访问 MongoDB 和

Elasticsearch 的服务。

- -p 9000:9000：映射 Graylog 的 Web 服务端口 9000。
- -p 12201:12201/udp：映射 Graylog 接收日志数据的 UDP 端口 12201。
- GRAYLOG_WEB_ENDPOINT_URI：指定 Graylog 的 Web 访问 URI，请注意这里需要使用 Docker Host 的外部 IP（在实验环境中为 192.168.56.101）。
- GRAYLOG_ROOT_PASSWORD_SHA2：指定 Graylog 管理员用户密码的哈希值，在这个例子中密码为 admin。可以通过如下命令生成自己的密码哈希，比如：

```
echo -n yourpassword | shasum -a 256
```

容器启动后，在 Web 浏览器中访问 [http://\[Docker Host IP\]:9000](http://[Docker Host IP]:9000)，如图 10-27 所示。

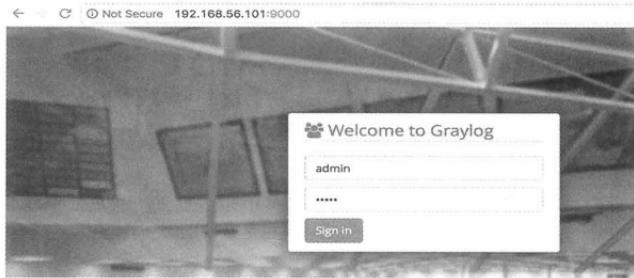


图 10-27

用户名/密码 = admin/admin。

登录后显示 Getting Started 页面，如图 10-28 所示。

Getting Started - Graylog v2.2.3+7adc951

No one is born a master. Use this page if you need assistance with your first steps. Make sure to ask the community if you should get stuck.

- 1 Send in first log messages**
Graylog is pretty useless without some log data in it. Let's start by sending in some messages.
- 2 Do something with your data**
Perform searches to solve some example use cases and get a feeling for the basic Graylog search functionalities.
- 3 Create a dashboard**
Dashboards are a great way to organize information that you look at often. Learn how to create them and how to add widgets with interesting information.
- 4 Be alerted**
Immediately receive alerts and trigger actions when something interesting or unusual happens.

Head over to the documentation and learn about Graylog in more depth.

图 10-28

10.5.3 配置 Graylog

目前 Graylog 还没法接收任何日志，我们需要配置一个 Input，单击顶部菜单 System → Inputs，如图 10-29 所示。

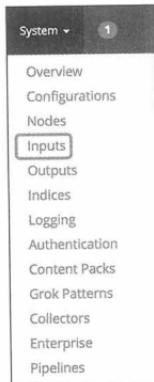


图 10-29

Graylog 支持多种 Input 类型，与 Graylog 对接的 Docker logging driver 是 gelf，因此这里我们需要运行一个 GELF UDP 类型的 Input，如图 10-30 所示。

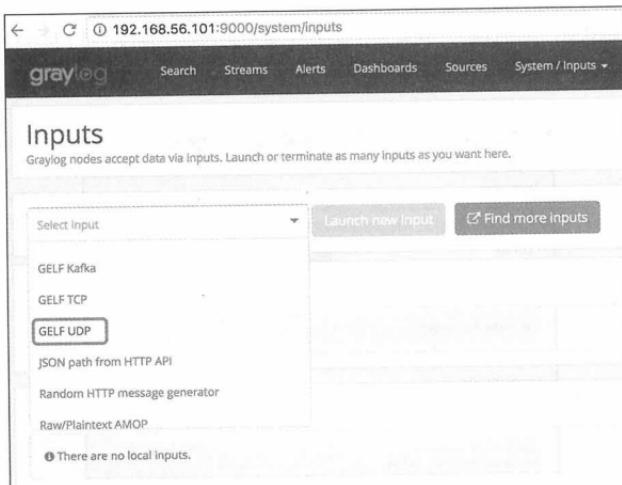


图 10-30

单击 **Launch new input**，出现如图 10-31 所示的界面。



图 10-31

在 Node 列表中选择 Graylog 容器。

Title 命名为 docker GELF input。

其他保持默认值，其中 port 12201 即为容器启动时映射到 Host 的端口，用于接收日志数据。

单击 **Save**，Input 成功运行，如图 10-32 所示。

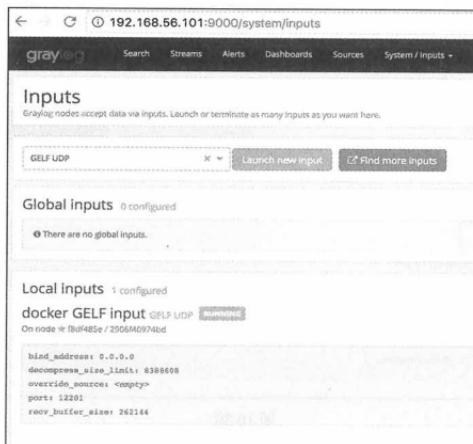


图 10-32

Graylog 已经准备就绪，接下来就可以将容器的日志发送给 Graylog 了。

10.5.4 监控容器日志

启动测试容器。

```
docker run -d \ --log-driver=gelf \ --log-opt gelf-
address=udp://localhost:12201 \ --log-opt tag="log-test-container-A" \
busybox sh -c 'while true; do echo "This is a log message from container
A"; sleep 10; done;'

docker run -d \ --log-driver=gelf \ --log-opt gelf-
address=udp://localhost:12201 \ --log-opt tag="log-test-container-B" \
busybox sh -c 'while true; do echo "This is a log message from container
B"; sleep 10; done;'
```

- `--log-driver=gelf`: 告诉 Docker 使用 GELF 的 logging driver。
- `--log-opt gelf-address=localhost:12201`: 将容器日志发送到 Graylog 的日志接收端口。
- `--log-opt tag="log-test-container-A"` 和 `--log-opt tag="log-test-container-B"`: 在日志中添加一个可选的 tag，用于区分不同的容器。

容器启动后，单击 Graylog 顶部菜单 Search，就能够查询到容器的日志了，如图 10-33 所示。

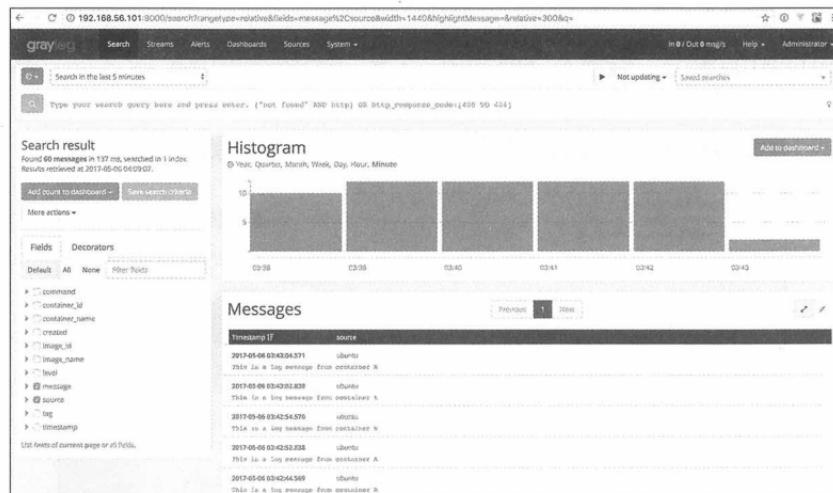


图 10-33

与 Kibana 一样，Graylog 也提供了强大的查询功能，比如输入关键字 `container B` 能搜索

出所有匹配的日志条目，如图 10-34 所示。

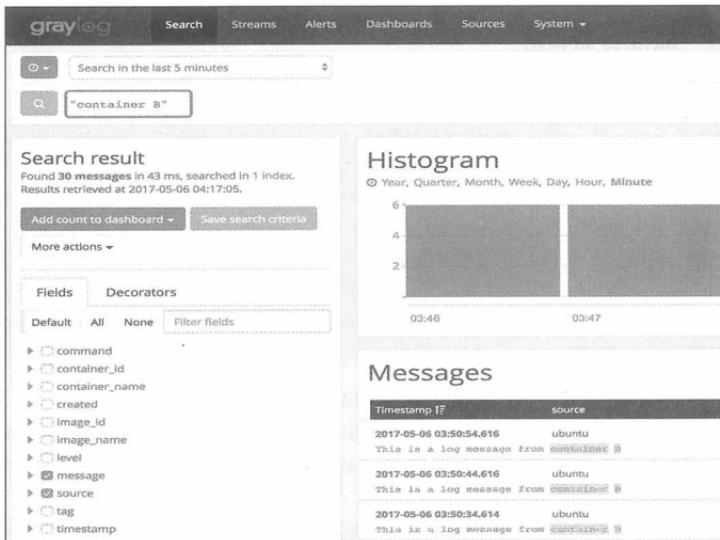


图 10-34

与前面 ELK 一样，这里我们只是简单地将日志导入到 Graylog。实际上 Graylog 也可以对日志进行归类汇总、分析聚合、创建 Dashboard 等。下面这张图可以感受一下 Graylog 的特性，更多的功能留给大家自己去探索，如图 10-35 所示。

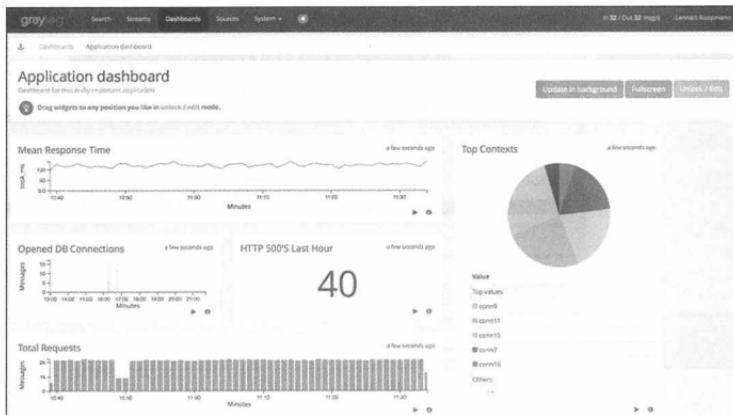


图 10-35

10.6 小结

本章介绍了 Docker 日志管理的方案，我们由 docker logs 引出了 Docker logging driver；进而学习了 ELK 日志处理 stack；通过 fluentd logging driver，我们很容易地将 fluentd 接入到日志管理方案中；最后我们还实践了与 ELK 同等量级的 Graylog。

与容器监控一样，容器日志管理也是一个百花齐放、高速迭代的技术领域。没有最好的，只有最适合的。

不同企业有不同的部署规模，有自己的管理流程，有各自的业务目标；运维团队有不同的技术背景、人员结构和工作方式；唯有保持开放的心态，多看、多学、多实践，才能构建出适合自己的系统。

第 11 章

◀ 数据管理 ▶

从业务数据的角度看，容器可以分为两类：无状态（stateless）容器和有状态（stateful）容器。

无状态是指容器在运行过程中不需要保存数据，每次访问的结果不依赖上一次访问，比如提供静态页面的 Web 服务器。

有状态是指容器需要保存数据，而且数据会发生变化，访问的结果依赖之前请求的处理结果，最典型的就是数据库服务器。

简单来讲，状态（state）就是数据，如果容器需要处理并存储数据，它就是有状态的，反之则无状态。

对于有状态的容器，如何保存数据呢？

前面在 Docker 存储章节我们学习到 data volume 可以存储容器的状态，不过当时讨论的 volume 其本质是 Docker 主机本地的目录。

本地目录就存在一个隐患：如果 Docker Host 宕机了，如何恢复容器？

一个办法就是定期备份数据，但这种方案还是会丢失上次备份到宕机这段时间的数据。更好的方案是由专门的 storage provider 提供 volume，Docker 从 provider 那里获取 volume 并挂载到容器。这样即使 Host 挂了，也可以立刻在其他可用 Host 上启动相同镜像的容器，同时直接挂载之前使用的 volume，这样不会有数据丢失。

本章将详细讨论如何实现跨 Docker 主机管理 data volume。

11.1 从一个例子开始

假设有两个 Docker 主机，Host1 运行了一个 MySQL 容器，为了保护数据，data volume 由 storage provider 提供，如图 11-1 所示。

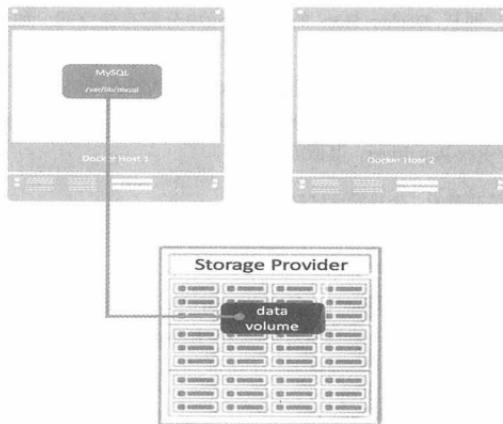


图 11-1

当 Host1 发生故障，我们会在 Host2 上启动相同的 MySQL 镜像，并挂载 data volume，如图 11-2 所示。

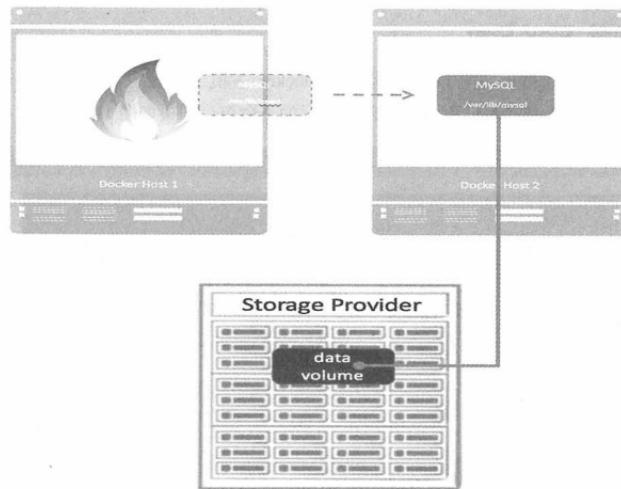


图 11-2

Docker 是如何实现这个跨主机管理 data volume 方案的呢？

答案是 volume driver。

任何一个 data volume 都是由 driver 管理的，创建 volume 时如果不特别指定，将使用

local 类型的 driver，即从 Docker Host 的本地目录中分配存储空间。如果要支持跨主机的 volume，则需要使用第三方 driver。

目前已经有很多可用的 driver，比如使用 Azure File Storage 的 driver，使用 GlusterFS 的 driver，完整的列表可参考 https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins。

我们这里将选择 Rex-Ray driver，其原因是：

- (1) Rex-Ray 是开源的，而且社区活跃。
- (2) 支持多种 backend，如 VirtualBox 的 Virtual Media、Amazon EBS、Ceph RBD、Openstack Cinder 等。
- (3) 支持多种操作系统，如 Ubuntu、CentOS、RHEL 和 CoreOS。
- (4) 支持多种容器编排引擎，如 Docker Swarm、Kubernetes 和 Mesos。
- (5) Rex-Ray 安装使用方法非常简单。

下面开始实践。

11.2 实践 Rex-Ray driver

11.2.1 安装 Rex-Ray

Rex-Ray 以 standalone 进程的方式运行在 Docker 主机上，安装方法很简单，在需要使用 Rex-Ray driver 的主机 docker1 和 docker2 上运行如下命令，结果如图 11-3 所示。

```
curl -sSL https://dl.bintray.com/emccode/rexray/install | sh -
```

```
root@docker1:~#
root@docker1:~# curl -sSL https://dl.bintray.com/emccode/rexray/install | sh
Selecting previously unselected package rexray.
(Reading database ... 159004 files and directories currently installed.)
Preparing to unpack rexray_0.9.1-1_amd64.deb ...
Unpacking rexray (0.9.1-1) ...
Setting up rexray (0.9.1-1) ...

rexray has been installed to /usr/bin/rexray

REX-Ray
-----
Binary: /usr/bin/rexray
Flavor: client+agent+controller
SemVer: 0.9.1
OsArch: Linux-x86_64
Branch: v0.9.1
Commit: 2373541479478b81768b143629e552f404f75226
Formed: Sat, 10 Jun 2017 03:23:38 HKT

libStorage
-----
SemVer: 0.6.1
OsArch: Linux-x86_64
Branch: v0.9.1
Commit: fd26f0ec72b077ffa7c82160fd12a276e12c2ad
Formed: Sat, 10 Jun 2017 03:23:05 HKT

root@docker1:~#
```

图 11-3

然后创建并编辑 Rex-Ray 的配置文件：/etc/rexray/config.yml。
可以使用图形化的在线 Rex-Ray 配置生成器 <http://rexrayconfig.codedellemc.com/>，如图 11-4 所示。

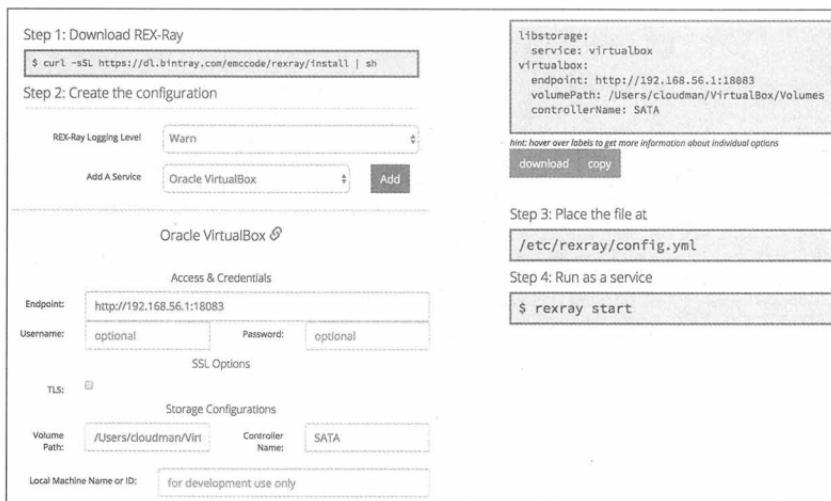


图 11-4

我们的配置文件内容如图 11-5 所示。

```

libstorage:
  service: virtualbox ①
  virtualbox:
    endpoint: http://192.168.56.1:18083 ②
    volumePath: /Users/cloudman/VirtualBox/Volumes ③
    controllerName: SATA ④
  
```

图 11-5

① 使用 VirtualBox 的 Virtual Media 作为 backend，提供 data volume。原因是我们的实验环境就是用的 VirtualBox，不需要额外部署存储系统，作为验证和实践，已经足够了。当然，如果是生产系统，肯定得选择更健壮的 backend，比如 Ceph RBD。

② http://192.168.56.1:18083 是 VirtualBox 宿主机（我的笔记本）的服务端口，后面会演示如何启动这个服务。

③ volumePath 是 VirtualBox 宿主机上存放 data volume 的目录。

④ SATA 是 controller 的名字，不用修改。

11.2.2 配置 VirtualBox

在 VirtualBox 宿主机，即我的笔记本上启动 vboxwebsrv 服务，结果如图 11-6 所示。

```
vboxwebsrv -H 0.0.0.0
```

```
→ ~
→ ~ vboxwebsrv -H 0.0.0.0
Oracle VM VirtualBox web service Version 5.1.4
(C) 2007-2016 Oracle Corporation
All rights reserved.

VirtualBox web service 5.1.4 r110228 darwin.amd64 (Aug 16 2016 20:07:28) release log
00:00:00.000158 main Log opened 2017-06-21T08:17:58.283372000Z
00:00:00.000169 main Build Type: release
00:00:00.000179 main OS Product: Darwin
00:00:00.000184 main OS Release: 15.5.0
00:00:00.000189 main OS Version: Darwin Kernel Version 15.5.0: Tue Apr 19 18:36:36 PDT 2016; root:xnu-3248.50.21~RELEASE_X86_64
00:00:00.000282 main DMI Product Name: MacBookPro11,5
00:00:00.000322 main DMI Product Version: 1.0
00:00:00.000333 main Host RAM: 16384MB total, 2885MB available
00:00:00.000336 main Executable: /Applications/VirtualBox.app/Contents/MacOS/vboxwebsrv
00:00:00.000337 main Process ID: 87306
00:00:00.000337 main Package type: DARWIN_64BITS_GENERIC
00:00:00.000373 main IPC socket path: /tmp/.vbox-wanglei-ipc/ipcd
00:00:00.007055 SQMp Socket connection successful: host = 0.0.0.0, port = 18083, master socket = 7
00:00:05.007932 Watchdog Statistics: 0 websessions, 0 references
```

图 11-6

执行如下命令关闭 VirtualBox 的登录认证：

```
VBoxManage setproperty websrvauthlibrary null
```

在关机状态下修改虚拟机 docker1 和 docker2 的 Storage 配置：

(1) 删除 IDE controller，如图 11-7 所示。

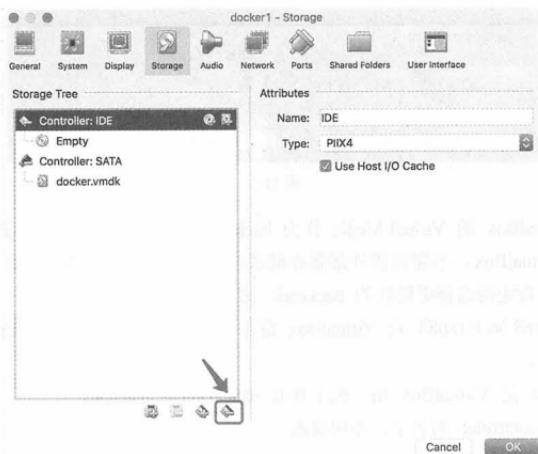


图 11-7

(2) 设置 SATA controller 的 port 数量为最大值 30, 如图 11-8 所示。

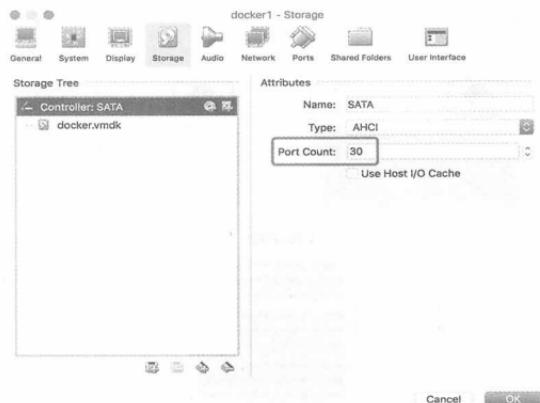


图 11-8

(3) 重启 Rex-Ray 服务:

```
systemctl restart rexray.service
```

运行 `rexray volume ls` 测试 Rex-Ray 是否能够正常工作, 如图 11-9 所示。

ID	Name	Status	Size
1e8bd5fd-aa26-41ba-8256-b77e66fc14d6	NewVirtualDisk1.vmdk	unavailable	8
5bc8eeb9-2b8b-4cfa-af7c-ac0802130658	NewVirtualDisk1.vmdk	unavailable	47
a3ae93e6-1172-4acf-9f01-20f766e0658a	devstack-compute.vdi	unavailable	39
889cad05-f93b-4875-9168-f42898d72139	devstack-controller.vdi	unavailable	79
58edb0a0-6300-4089-bdf0-5f78d4fbfb51	docker.vmdk	attached	8
27795e35-05a7-4212-8049-3f00e42cd2d3	docker2.vmdk	unavailable	8
2e182c60-540d-47c4-92be-66a482fce916	jenkins-disk1.vmdk	unavailable	8
0nb00993-ceb9-4d55-ba67-caff37dfb721	k8s-master.vmdk	unavailable	8
5bc3966e-8421-4808-8194-3633a8f98f72	k8s-node1.vmdk	unavailable	8
c0ea0672-f0fc-4a80-a7c7-1ae9cf409757	k8s-node2.vmdk	unavailable	8
67789d8a-4446-4842-b018-1ca8b14467	mesos-master.vmdk	unavailable	8
d0fc9116-5470-46fa-98e5-ed0cced8d8187	mesos-slave.vmdk	unavailable	8
6c15761c-4529-4cb6-a16d-fee4cb75f95f	mydata	unavailable	16
02633025-152d-4bf8-0002-205fffa92e6b	packer-virtualbox-iso-149158748-disk001.vmdk	unavailable	100
03bb0601-0633-4d72-ac40-324ff9a04e48	packer-virtualbox-iso-149158748-disk001.vmdk	unavailable	100
d83a0351-a7dc-49b9-b0b8-7e2101271914	packer-virtualbox-iso-149158748-disk001.vmdk	unavailable	100
697570a6-7032-4c34-bb4d-844e6c76c15d	packer-virtualbox-iso-149158748-disk001.vmdk	unavailable	100
73e68deaa-e287-436f-ba29-03044f729c92	rancher-kubernetes-disk1.vmdk	unavailable	8
c9092e65-38c0-4de3-a758-e0ad9a79351e	rancher.vmdk	unavailable	8
c8d98319-6283-4402-9ff4-aa59fe2727ae	swarm-manager.vmdk	unavailable	8
9d81692f-f1fe-443f-9f4f-1a024a60e6a	swarm-worker1.vmdk	unavailable	8
5d3ebc95-67c3-4db8-b7ff-e80772127c9d	swarm-worker2.vmdk	unavailable	8
076590c3-af4e-4fe-bb25-ca1f8b6fec36	ubuntu16.04_docker.vmdk	unavailable	8
0b0a5c42-c0b0-4884-a0ab-768770ab1c43	ubuntu16.04.vmdk	unavailable	8
32c2b5d4-2f02-46e2-a0c9-5b0e85db1e27	win7.vmdk	unavailable	24

图 11-9

(4) 列表中的 volume 是当前 VirtualBox 所有的虚拟磁盘。准备就绪, 当前实验环境如图 11-10 所示。

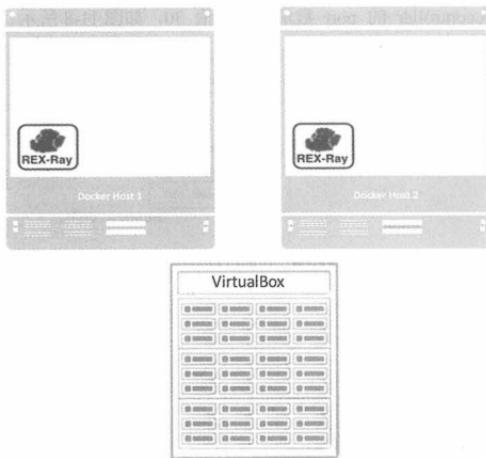


图 11-10

11.2.3 创建 Rex-Ray volume

在 docker1 或 docker2 上执行如下命令创建 volume，结果如图 11-11 所示。

```
docker volume create --driver rexray --name=mysqldata --opt=size=2
```

ID	Name	Status	Size
SbcBeeb9-2b8b-4cfa-af7c-acd802130658	NewVirtualDisk1.vmdk	unavailable	47
1e8bd5fd-aa26-41ba-8256-b77e66fc14d4	NewVirtualDisk1.vmdk	unavailable	8
a3ae93e6-1172-4acf-9f01-202f660e6058	devstack-compute.vdi	unavailable	39
889cad5f-f93b-487d-a000-42806723bb1	devstack-controller.vdi	unavailable	72
586e0300-4a0d-474c-92b4-66e482fce916	docker.vmdk	attached	8
277905e3-0507-4212-8049-30f0e42cd073	rancher2.vmdk	unavailable	8
2e182c60-540d-47c4-92b4-66e482fce916	jenkins-disk1.vmdk	unavailable	8
0ab00993-ceb9-4a55-b0b7-coff3dfb721	k8s-master.vmdk	unavailable	8
Sbc3966e-8421-480b-8194-3633a08f98f72	k8s-node1.vmdk	unavailable	8
cceaa072-f0fc-4a80-07c7-1ee9c9f409757	k8s-node2.vmdk	unavailable	8
67789d8a-4446-4842-b018-a1ca08b414467	mesos-master.vmdk	unavailable	8
d0fcfb916-5479-46fa-98e5-ed0cde8d8187	mesos-slave.vmdk	unavailable	8
6c15761c-4529-4cb6-01ed-fed4cb75f95f	mysqldata	unavailable	16
c7031480-d18e-4a08-9187-84b09d9e0c57	mysqldata	available	2
0dbbd061-0610-4007-0204-189a0d000000	pocket-virtualbox-iso-1401588748-disk001.vmdk	unavailable	100
6204022d-4bfb-4002-2a04-189a0d000000	pocket-virtualbox-iso-1401588748-disk001.vmdk	unavailable	100
6375206-7032-4c34-bb4d-844a6c26c15d	pocket-virtualbox-iso-1401588748-disk001.vmdk	unavailable	100
d3a5351-c2dc-49b9-b0b7-7e20121719f4	pocket-virtualbox-iso-1401588748-disk001.vmdk	unavailable	100
73ef6dea-e287-43f6-ha29-03044f729c92	rancher-kubernetes-disk1.vmdk	unavailable	8
c9092e65-38c0-4de3-0758-eed9a79351e	rancher.vmdk	unavailable	8
c8d98319-6283-4402-9fff-aa05c9feb27ae	swarm-manager.vmdk	unavailable	8
9d81602f-f7ea-443f-984f-bc024e60e60	swarm-worker1.vmdk	unavailable	8
5d3ebc95-67c3-4d08-b7ff-e080772127c9d	swarm-worker2.vmdk	unavailable	8
076590c3-af4e-4efc-bb25-c01f8bf6ec36	ubuntu16.04_docker.vmdk	unavailable	8
0b0d5c22-c0b0-4884-a0ab-76877aab1e43	ubuntu16.04.vmdk	unavailable	8
32c2b54a-2102-46e2-a0c9-5b0e85db1e27	win7.vmdk	unavailable	24

图 11-11

volume mysqldata 创建成功，大小为 2GB。在 VirtualBox 宿主机中也能看到 mysqldata，如图 11-12 所示。

```

→ ~
→ ~ ls -l VirtualBox/Volumes
total 640
-rw----- 1 cloudman  staff  327680 Jun 21 16:50 mysqldata
→ ~

```

图 11-12

因为 VirtualBox 使用的是 thin-provisioning, volume 初始分配的空间很小。

11.2.4 使用 Rex-Ray volume

接下来我们将:

- (1) 在 dokcer1 上启动 MySQL 容器 mydb_on_docker1, 并使用 mysqldata 作为数据卷。
- (2) 更新数据库, 然后销毁 mydb_on_docker1。
- (3) 在 dokcer2 上启动 MySQL 容器 mydb_on_docker2, 也使用 mysqldata 作为数据卷, 然后验证数据的有效性。

1. 创建容器并使用数据卷

在 dokcer1 上执行如下命令, 启动 MySQL 容器:

```

docker run --name mydb_on_docker1 -v mysqldata:/var/lib/mysql -e
MYSQL_ROOT_PASSWORD=passw0rd -d mysql

```

执行-v mysqldata:/var/lib/mysql 将之前创建的 volume mount 到 MySQL 的数据目录。我们接下来从更底层分析一下这个 mount 是如何实现的。

首先在 VirtualBox 中查看虚拟机 docker1 的 storage 配置, 如图 11-13 所示。

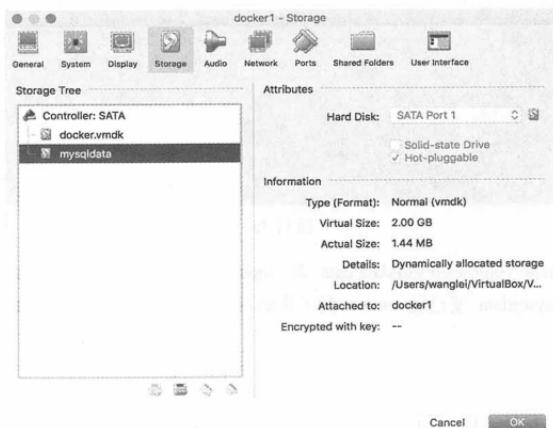


图 11-13

Rex-Ray volume mysqldata 已经挂载到 docker1。

执行 docker volume inspect mysqldata，结果如图 11-14 所示。

```
root@docker1:~#
root@docker1:~# docker volume inspect mysqldata
[
  {
    "Driver": "rexray",
    "Labels": {},
    "Mountpoint": "/var/lib/libstorage/volumes/mysqldata/data",
    "Name": "mysqldata",
    "Options": {
      "size": "2"
    },
    "Scope": "global",
    "Status": {
      "availabilityZone": "",
      "fields": null,
      "iops": 0,
      "name": "mysqldata",
      "server": "virtualbox",
      "service": "virtualbox",
      "size": 2,
      "type": "HardDisk"
    }
  }
]
root@docker1:~#
```

图 11-14

mysqldata 已被 mount 到 docker1 目录 /var/lib/libstorage/volumes/mysqldata/data，执行 docker inspect mydb_on_docker1 查看容器的 volume 信息，如图 11-15 所示。

```
"Mounts": [
  {
    "Type": "volume",
    "Name": "mysqldata",
    "Source": "/var/lib/libstorage/volumes/mysqldata/data",
    "Destination": "/var/lib/mysql",
    "Driver": "rexray",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
],
```

图 11-15

/var/lib/libstorage/volumes/mysqldata/data 被 mount 到了容器的目录 /var/lib/mysql，这样 Rex-Ray volume mysqldata 就已经 mount 到了容器 mydb_on_docker1，如图 11-16 所示。

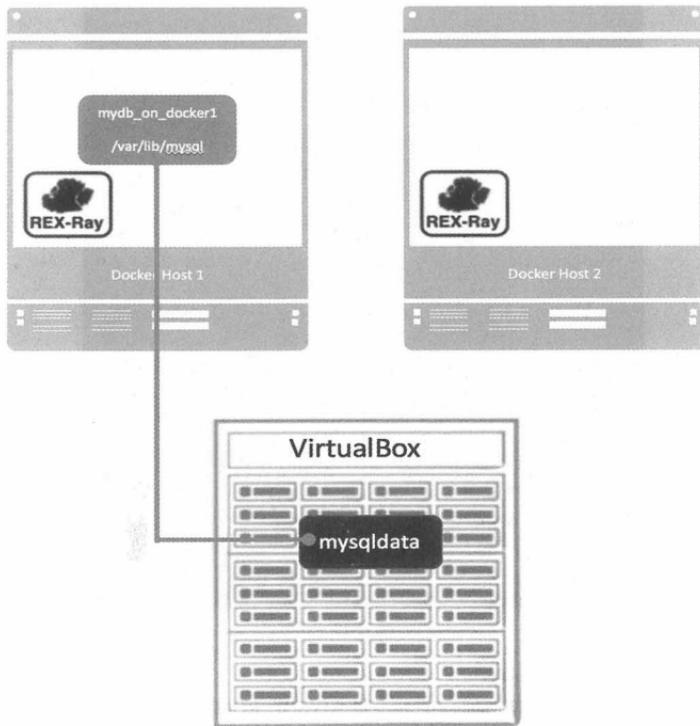


图 11-16

2. 更新数据库

按照如下步骤更新 MySQL 数据，如图 11-17 所示。

```

root@docker1:#
root@docker1: # docker exec -it mydb_on_docker1 bash ①
root@00dec9d55c6a:/#
root@00dec9d55c6a:/# mysql -p ②
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.18 MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use mysql ③
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> create table my_id( id int(4) ); ④
Query OK, 0 rows affected (0.01 sec)

mysql> insert my_id values( 111 ); ⑤
Query OK, 1 row affected (0.00 sec)

mysql> select * from my_id; ⑥
+----+
| id |
+----+
| 111 |
+----+
1 row in set (0.00 sec)

mysql>

```

图 11-17

- ① 进入容器 `mydb_on_docker1`。
- ② 登录数据库，输入容器启动时由环境变量 `MYSQL_ROOT_PASSWORD` 指定的密码。
- ③ 切换到数据库 `mysql`。
- ④ 创建数据库表 `my_id`。
- ⑤ 插入一条数据。
- ⑥ 确认数据已经写入。

执行 `docker rm -f mydb_on_docker1` 删除容器。

3. 创建新容器并使用数据卷

在 `doker2` 上执行如下命令，启动 MySQL 容器：

```

docker run --name mydb_on_docker2 -v mysqldata:/var/lib/mysql -d
mysql

```

新容器也使用相同的卷 `mysqldata`，不过这次不需要指定环境变量 `MYSQL_ROOT_PASSWORD`，因为密码已经保存到 `mysqldata` 里面了。

现在 Rex-Ray volume `mysqldata` 已经挂载到 `docker2`，如图 11-18 所示。

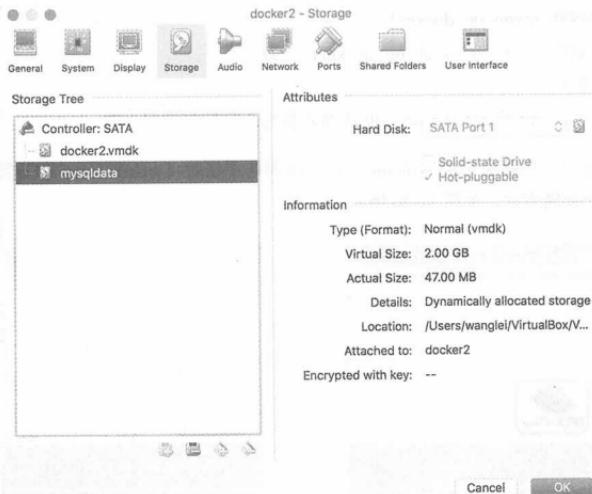


图 11-18

同样可以按照之前的方法用 docker volume inspect 和 docker inspect 查看具体的 mount 信息，这里不再赘述。

4. 验证 MySQL 的数据

按照如下步骤验证 MySQL 的数据，如图 11-19 所示。

```

root@docker2:~# docker exec -it mydb_on_docker2 bash ①
root@80e2dc5a83a0$:# mysql -p ②
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.18 MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use mysql ③
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from my_id; ④
+----+
| id |
+----+
| 111 |
+----+
1 row in set (0.00 sec)

mysql>

```

图 11-19

- ① 进入到容器 mydb_on_docker2。
- ② 登录数据库，密码与 mydb_on_docker1 一致。
- ③ 切换到数据库 mysql。
- ④ 确认之前由 mydb_on_docker1 创建的表和写入的数据完好无损。

Rex-Ray 可以提供跨主机的 volume，其生命周期不依赖 Docker Host 和容器，是 stateful 容器理想的数据存储方式，如图 11-20 所示。

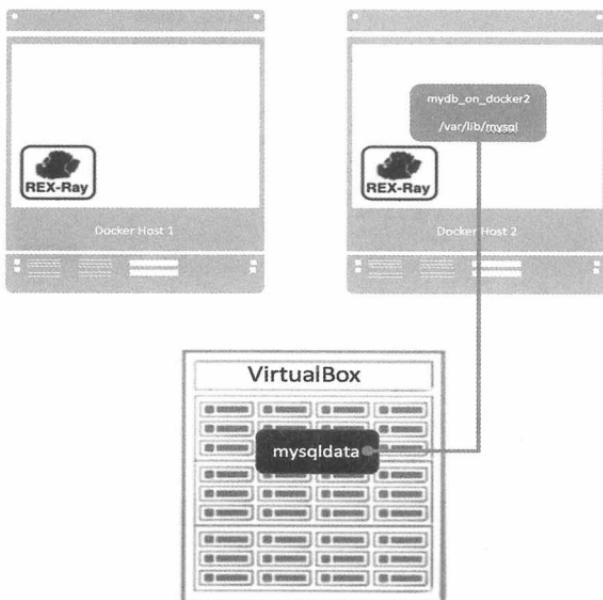
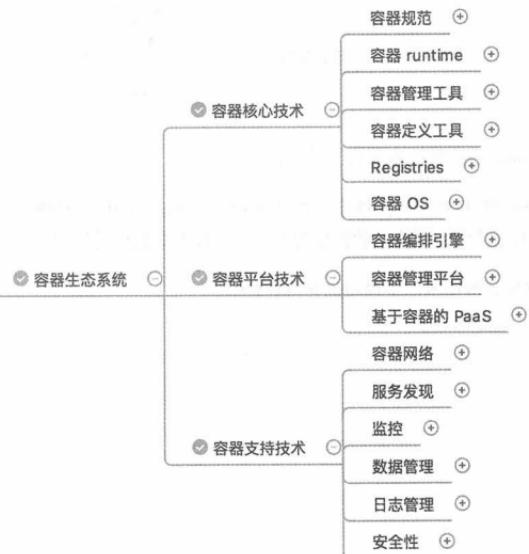


图 11-20

写在最后

作为本书的结尾，让我们再来看看一下容器生态系统。如下图所示。



本书包含三个部分：

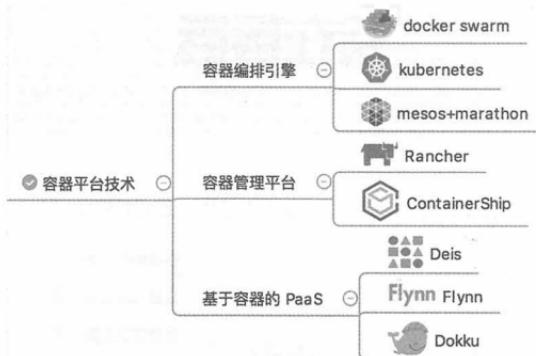
- (1) 第一部分介绍容器技术生态环境。
- (2) 第二部分是容器核心知识，包括架构、镜像、容器、网络和存储。
- (3) 第三部分是容器进阶知识，包括多主机管理、跨主机网络方案、监控和日志管理。

掌握了这三部分知识，就打下了在工作中应用容器的坚实基础。

另外，如果将这三部分内容与容器生态系统对照，本书实际上覆盖了容器核心技术和容器支持技术的大部分内容。

容器平台技术涉及容器编排引擎、容器管理平台和基于容器的 PaaS，在大规模生产部署中占有相当重要的位置。如下图所示。同时其本身是一个庞大的主题，我们会在同系列的下一本

书中专门讨论。



最后, CloudMan 再次推荐本书的使用方法:

- (1) 跟着教程进行操作, 在实践中掌握 Docker 容器技术的核心技能。
- (2) 在之后的工作中, 可将本教程作为参考书, 按需查找相关知识点。

祝大家早日掌握容器技术, 实现和提升自我价值。