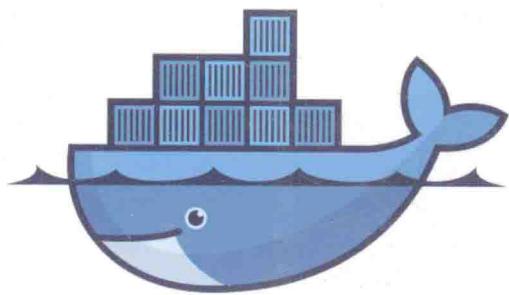




华章科技

中国首部 Docker 著作，一线 Docker 先驱实战经验结晶，来自IBM和新浪等多位技术专家联袂推荐！

结合企业生产环境，深入浅出地剖析 Docker 的核心概念、应用技巧、实现原理以及生态环境，为解决各类问题提供了有价值的参考。



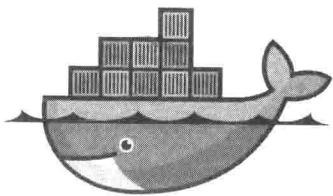
杨保华 戴王剑 曹亚仑 编著

*Docker Primer*

# Docker 技术入门与实战



机械工业出版社  
China Machine Press



# Docker

## 技术入门与实战

杨保华 戴王剑 曹亚仑 编著



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

Docker 技术入门与实战 / 杨保华, 戴王剑, 曹亚伦编著. —北京: 机械工业出版社,  
2014.12  
(实战)

ISBN 978-7-111-48852-1

I. D… II. ①杨… ②戴… ③曹… III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字 (2014) 第 287910 号

在云计算时代, 开发者将应用转移到云上已经解决了硬件管理的问题, 然而软件配置和管理相关的问题依然存在。Docker 的出现正好能帮助软件开发者开阔思路, 尝试新的软件管理方法来解决这个问题。通过掌握 Docker, 开发人员便可享受先进的自动化运维理念和工具, 无需运维人员介入即可顺利运行于各种运行环境。

本书分为三大部分: Docker 入门、实战案例和高级话题。第一部分 (第 1 ~ 8 章) 介绍 Docker 与虚拟化技术的基本概念, 包括安装、镜像、容器、仓库、数据管理等; 第二部分 (第 9 ~ 17 章) 通过案例介绍 Docker 的应用方法, 包括与各种操作系统平台、SSH 服务的镜像、Web 服务器与应用、数据库的应用、各类编程语言的接口、私有仓库等; 第三部分 (第 18 ~ 21 章) 是一些高级话题, 如 Docker 核心技术、安全、高级网络配置、相关项目等。

本书从基本原理开始入手, 深入浅出地讲解 Docker 的构建与操作, 内容系统全面, 可帮助开发人员、运维人员快速部署应用。

# Docker 技术入门与实战

---

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: 吴 怡

责任校对: 董纪丽

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2015 年 1 月第 1 版第 1 次印刷

开 本: 186mm × 240mm 1/16

印 张: 19.5

书 号: ISBN 978-7-111-48852-1

定 价: 59.00 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) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@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

## *Foreword 序一*

信息技术领域一直试图解决的核心问题之一是提供强大的计算能力。在过去很长一段时间里，我们可以依靠硬件性能的提升来提高物理计算资源的能力，提升处理器的主频或者增加每个处理器里面的处理核心的数量。然而这个时代随着摩尔定律无法胜过物理定律而不得不终结。

在云计算时代，信息技术所面临的难题则截然不同。分布式、虚拟化、大数据……每一项挑战都不是仅仅依靠硬件或软件的局部优化就能得到解决的，更需要通过高效的资源利用来“压榨”计算平台的每一点运算能力。

作为充分发掘计算平台能力的 Linux 容器虚拟化技术，在近些年得到广泛的关注和发展。从早期 IBM 发起的 LXC 项目，到今天如火如荼的 Docker 项目，这些不断涌现的创新项目给计算模式本身带来了巨大的变革。市面上关于传统虚拟机相关技术的书籍有不少，但是探讨容器虚拟化技术的著作寥寥无几。尽管互联网上已经出现了很多关于容器虚拟化、Docker 的文章，但这些文章或过于简略，或仅关注某个技术方面，总体上缺少系统化的从概念、到实现、到如何使用的介绍。这为广大信息产业从业人员了解最新的技术潮流带来了不小的障碍。

值得庆幸的是，能够第一时间拜读这本《Docker 技术入门与实战》。作为国内开发者撰写的首本探讨 Docker 容器虚拟化技术的书籍，一方面它深入浅出地讲解了 Docker 应用的诸多话题，包括围绕镜像、容器、仓库等核心概念如何来实现“Build、Ship、Run”的高效流程；另一方面，难能可贵的是书中提供了大量翔实的实战案例，涵盖 DevOps 领域的典型场景。无论是 Docker 技术的初学者还是业内的一线研发人员或资深专家，本书都值得一阅。

作者之一的杨保华博士在加入 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，并积极参与开源社区的讨论、积极提交代码。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。

如果你只是 Docker 的初学者，阅读本书，或许并不能让你立刻成为行业内的专业人士，但一定能让你马上体会 Docker 技术所带来的众多优势。如果你已经开始使用 Docker，阅读本书也可以帮助你解答实践中的一些问题，帮助你更恰当地使用 Docker 技术。本书深入浅出，讲解到位，是一本值得常置案头的好书。

刘天成

IBM 中国研究院，云计算运维技术研究组经理

2014 年 11 月

## *Forward* 序二

最近的几年，云计算是计算机与互联网界的焦点，它的广泛应用离不开虚拟化技术的支持。作为 Linux 下的容器虚拟化工具，Docker 以其轻便易用而受人关注。

这本书向读者清晰地介绍了 Docker 这个虚拟化工具；详细比较了 Docker 和传统虚拟机在组织架构、实现技术和性能上的差异。在此基础上，本书围绕着镜像、容器、仓库三个部分，从实践的角度出发，讲解了 Docker 的安装、配置、使用的方式。在本书的后面几个章节，也介绍了许多 Docker 的实现细节和工作原理。总体而言，本书从实际的案例入手，由浅至深，循序渐进，内容相当丰富。

对于正在寻找虚拟化工具的用户来说，年轻而有活力的 Docker 项目绝对是首选。而如果你正在使用或打算使用 Docker，或者想学习一些新的技术以丰富自己，那就一定不要错过这本书。书中有大量的实践案例、完备的细节讲解，将这本书常备于手边，比起查阅复杂繁琐的文档，能为工作或学习节省更多的时间。

王灿

浙江大学计算机学院副教授

2014 年 11 月

## 前言 *Preface*

在一台服务器上同时运行一百个虚拟机，肯定会被认为是痴人说梦。而在一台服务器上同时运行一千个 Docker 容器，这已经成为现实。在计算机技术高速发展的今天，昔日的天方夜谭正在一个个变成现实。

多年研发和运维（DevOps）经历中，笔者时常会碰到这样一个困境：用户的需求越来越多样，系统的规模越来越庞大，运行的软件越来越复杂，环境配置问题所造成的麻烦层出不穷……为了解决这些问题，开源社区推出过不少优秀的工具。这些方案虽然在某些程度上确能解决部分“燃眉之急”，但是始终没有一种方案能带来“一劳永逸”的效果。

让作为企业最核心资源的工程师们花费大量的时间，去解决各种环境和配置引发的 Bug，这真的正常吗？

回顾计算机的发展历程，最初，程序设计人员需要直接操作各种枯燥的机器指令，编程效率之低可想而知。高级语言的诞生，将机器指令的具体实现成功抽象出来，从此揭开了计算机编程效率突飞猛进的大时代。那么，为什么不能把类似的理念（抽象与分层）也引入到现代的研发和运维领域呢？

Docker 无疑在这一方向上迈出了具有革新意义的一步。笔者在刚接触 Docker 时，就为它所能带来的敏捷工作流程而深深吸引，也为它能充分挖掘云计算资源的效能而兴奋不已。我们深信，Docker 的出现，必将给 DevOps 技术，甚至整个信息技术产业的发展带来深远的影响。

笔者曾尝试编写了介绍 Docker 技术的中文开源文档。短短一个月的时间，竟收到了来自全球各个地区超过 20 万次的阅读量和全五星的好评。这让我们看到国内技术界对于新兴开源技术的敏锐嗅觉和迫切需求，同时也倍感压力，生怕其中有不妥之处，影响了大家学习和推广 Docker 技术的热情。在开源文档撰写过程中，我们一直在不断思考，在生产实践中到底怎么用 Docker 才是合理的？在“华章图书”的帮助下，终于有了现在读者手中的这本书。

与很多技术类书籍不同，本书中避免一上来就讲述冗长的故事，而是试图深入浅出、直奔主题，在最短时间内让读者理解和掌握最关键的技术点，并且配合实际操作案例和精炼的

点评，给读者提供真正可以上手的实战指南。

本书在结构上分为三大部分。第一部分是 Docker 技术的基础知识介绍，这部分将让读者对 Docker 技本能做什么有个全局的认识；第二部分将具体讲解各种典型场景的应用案例，供读者体会 Docker 在实际应用中的高效秘诀；第三部分将讨论一些偏技术环节的高级话题，试图让读者理解 Docker 在设计上的工程美学。最后的附录归纳了应用 Docker 的常见问题和一些常用的参考资料。读者可根据自身需求选择阅读重点。全书主要由杨保华和戴王剑主笔，曹亚伦写作了编程开发和实践之道章节。

本书在写作过程中参考了官方网站上的部分文档，并得到了 DockerPool 技术社区网友们的积极反馈和支持，在此一并感谢！

成稿之际，Docker 已经发布了增强安全特性的 1.32 版本。衷心祝愿 Docker 及相关技术能够快速成长和成熟，让众多 IT 从业人员的工作和生活都更加健康、更加美好！

作者于 2014 年 11 月

## 作者简介 *About the Authors*

**杨保华** 清华大学博士毕业，现为 IBM 中国研究院研究员。主要从事数据中心网络解决方案的研发与部署，技术方向包括云计算、软件定义网络（SDN）、网络安全等，是国内较早从事 SDN 和网络虚拟化相关技术的推广者，同时也是 DockerPool 开源社区的发起人之一。他的个人主页为 [yeasy.github.io](http://yeasy.github.io)。

---

在本书的写作期间，得到了我的父母亲和妻子吴渝萱女士的关怀与支持，得到了公司领导和同事们的信任与鼓励，特别是刘天成帮忙审阅了部分内容。在此表示最深厚的感谢！

---

**戴王剑** 资深架构师，从事计算机网络、服务器架构设计多年，负责过多个省级项目的架构设计。热衷开源事业，是 DockerPool 开源社区的发起人之一。

---

写作期间，我的女儿戴子萱刚刚出生，感谢我的父母亲以及我的妻子刘乃华对我的大力支持，没有你们的辛勤付出，我不可能安心写完这本书，我爱你们！在这里一并感谢公司领导和同事对我的信任，感谢郭捷给予的硬件支持，在经过半年左右的测试之后，Docker 在公司的项目中正式上线，并给我们的开发、测试、生产带来了实实在在的效率。特别感谢我的导师：浙江师范大学杨传斌教授，没有杨老师，我可能在毕业前就放弃从事 IT 行业了。本书的出版能得到杨老师的肯定，是最让我开心的事。

---

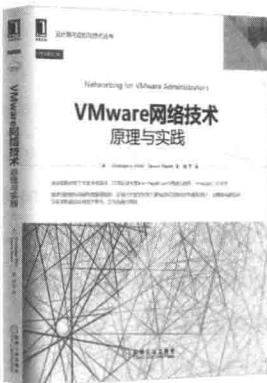
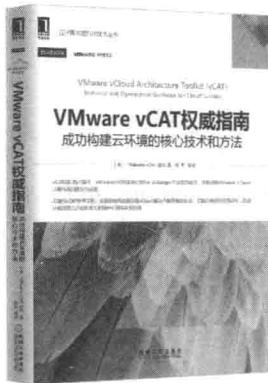
**曹亚仑** 85 后，全栈 Web 开发者，擅长并专注于 SaaS 系统架构设计与研发，兴趣方向为 PaaS 和智能可穿戴设备。译著有《Arduino 无线传感器网络实践指南》，开源图书有《程序员禅修指南》。个人主页为 [allengaller.com](http://allengaller.com)。

---

我要感谢我的父母和妻子丁小芬，感谢你们在我写书过程中所给予的帮助和支持，我爱你们。

---

# 推荐阅读



## 云计算：概念、技术与架构

作者：Thomas Erl 等 ISBN：978-7-111-46134-0 定价：69.00元

## 深入理解大数据：大数据处理与编程实践

作者：黄宜华 ISBN：978-7-111-47325-1 定价：79.00元

## VMware网络技术：原理与实践

作者：Christopher Wahl 等 ISBN：978-7-111-47987-1 定价：59.00元

## 云计算与分布式系统：从并行处理到物联网

作者：Kai Hwang 等 ISBN：978-7-111-41065-2 定价：85.00元

## VMware vCAT权威指南：成功构建云环境的核心技术和方法

作者：VMware vCAT 团队 ISBN：978-7-111-48228-4 定价：119.00元

## VMware Virtual SAN权威指南

作者：Cormac Hogan 等 ISBN：978-7-111-48023-5 定价：59.00元

## *Contents* 目录

序一  
序二  
前言  
作者简介

### 第一部分 Docker 入门

#### 第 1 章 初识 Docker ..... 3

- 1.1 什么是 Docker ..... 3
- 1.2 为什么要使用 Docker ..... 5
- 1.3 虚拟化与 Docker ..... 7
- 1.4 本章小结 ..... 8

#### 第 2 章 Docker 的核心概念和安装 ..... 9

- 2.1 核心概念 ..... 9
- 2.2 安装 Docker ..... 11
- 2.3 本书环境介绍 ..... 14
- 2.4 本章小结 ..... 15

#### 第 3 章 镜像 ..... 16

- 3.1 获取镜像 ..... 16
- 3.2 查看镜像信息 ..... 17
- 3.3 搜寻镜像 ..... 20

- 3.4 删除镜像 ..... 21
- 3.5 创建镜像 ..... 23
- 3.6 存出和载入镜像 ..... 24
- 3.7 上传镜像 ..... 25
- 3.8 本章小结 ..... 25

#### 第 4 章 容器 ..... 26

- 4.1 创建容器 ..... 26
- 4.2 终止容器 ..... 28
- 4.3 进入容器 ..... 29
- 4.4 删除容器 ..... 31
- 4.5 导入和导出容器 ..... 31
- 4.6 本章小结 ..... 32

#### 第 5 章 仓库 ..... 33

- 5.1 Docker Hub ..... 33
- 5.2 Docker Pool 简介 ..... 35
- 5.3 创建和使用私有仓库 ..... 36
- 5.4 本章小结 ..... 38

#### 第 6 章 数据管理 ..... 39

- 6.1 数据卷 ..... 39
- 6.2 数据卷容器 ..... 40

6.3 利用数据卷容器迁移数据	42	11.2 Nginx	86
6.4 本章小结	42	11.3 Tomcat	95
<b>第 7 章 网络基础配置</b>	<b>43</b>	11.4 Weblogic	102
7.1 端口映射实现访问容器	43	11.5 LAMP	119
7.2 容器互联实现容器间通信	45	11.5.1 下载 LAMP 镜像	119
7.3 本章小结	47	11.5.2 使用默认方式启动 LAMP 容器	119
<b>第 8 章 使用 Dockerfile 创建镜像</b>	<b>48</b>	11.5.3 部署自己的 PHP 应用	120
8.1 基本结构	48	11.5.4 在 PHP 程序中连接数据库	120
8.2 指令	49	11.6 CMS	121
8.3 创建镜像	53	11.7 本章小结	123
8.4 本章小结	53	<b>第 12 章 数据库应用</b>	<b>124</b>
<b>第二部分 实战案例</b>			
<b>第 9 章 操作系统</b>	<b>57</b>	12.1 MySQL	124
9.1 Busybox	57	12.2 Oracle XE	129
9.2 Debian/Ubuntu	60	12.3 MongoDB	130
9.3 CentOS/Fedora	62	12.4 本章小结	134
9.4 CoreOS	64	<b>第 13 章 编程语言</b>	<b>136</b>
9.5 本章小结	69	13.1 PHP	136
<b>第 10 章 创建支持 SSH 服务的镜像</b>	<b>70</b>	13.1.1 PHP 技术栈	136
10.1 基于 commit 命令创建	70	13.1.2 PHP 常用框架	142
10.2 使用 Dockerfile 创建	74	13.1.3 相关资源	147
10.3 本章小结	79	13.2 C/C++	147
<b>第 11 章 Web 服务器与应用</b>	<b>80</b>	13.2.1 GCC	147
11.1 Apache	80	13.2.2 LLVM	150
		13.2.3 Clang	150
		13.3 Java	151
		13.4 Python	153
		13.4.1 Python 技术栈	153
		13.4.2 Flask	155

13.4.3 Django	157	14.5 本章小结	196
13.4.4 相关资源	159		
13.5 Perl	160	<b>第 15 章 构建 Docker 容器集群</b>	197
13.5.1 Perl 技术栈	160	15.1 使用自定义网桥连接跨主机容器	197
13.5.2 Catalyst	161	15.2 使用 Ambassador 容器	199
13.5.3 相关资源	161	15.3 本章小结	200
13.6 Ruby	162		
13.6.1 Ruby 技术栈	162	<b>第 16 章 在公有云上使用 Docker</b>	202
13.6.2 JRuby	163	16.1 公有云上安装 Docker	202
13.6.3 Ruby on Rails	164	16.1.1 CentOS 6.5 系统	202
13.6.4 Sinatra	165	16.1.2 Ubuntu 14.04 系统	207
13.6.5 相关资源	166	16.2 阿里云 Docker 的特色服务	207
13.7 JavaScript	166	16.3 本章小结	213
13.7.1 JavaScript 技术栈	166		
13.7.2 Node.js	167	<b>第 17 章 Docker 实践之道</b>	214
13.7.3 Express	168	17.1 个人学习之道	214
13.7.4 AngularJS	170	17.1.1 温故而知新	215
13.7.5 相关资源	171	17.1.2 众人拾柴火焰高	216
13.8 Go	172	17.2 技术创业之道	217
13.8.1 Go 技术栈	172	17.3 中小型企业实践之道	218
13.8.2 Beego	174	17.3.1 开发、测试和发布中应用 Docker	218
13.8.3 Revel	175	17.3.2 应用 Docker 到生产环境	220
13.8.4 Martini	177	17.4 本章小结	220
13.8.5 相关资源	179		
13.9 本章小结	180		
<b>第 14 章 使用私有仓库</b>	181		
14.1 使用 docker-registry	181	<b>第三部分 高级话题</b>	
14.2 用户认证	183		
14.3 使用私有仓库批量上传镜像	186	<b>第 18 章 Docker 核心技术</b>	223
14.4 仓库配置文件	189	18.1 基本架构	223
		18.2 命名空间	225

18.3 控制组 .....	227	20.7 创建一个点到点连接 .....	246
18.4 联合文件系统 .....	229	20.8 工具和项目 .....	247
18.5 Docker 网络实现 .....	230	20.9 本章小结 .....	251
18.6 本章小结 .....	232		
<b>第 19 章 Docker 安全 .....</b>	<b>233</b>	<b>第 21 章 Docker 相关项目 .....</b>	<b>252</b>
19.1 命名空间隔离的安全 .....	233	21.1 平台即服务方案 .....	252
19.2 控制组资源控制的安全 .....	234	21.2 持续集成 .....	253
19.3 内核能力机制 .....	234	21.3 管理工具 .....	256
19.4 Docker 服务端的防护 .....	235	21.4 编程开发 .....	261
19.5 其他安全特性 .....	236	21.5 其他项目 .....	262
19.6 本章小结 .....	237	21.6 本章小结 .....	267
<b>第 20 章 高级网络配置 .....</b>	<b>238</b>		
20.1 网络启动与配置参数 .....	238		
20.2 配置容器 DNS 和主机名 .....	240		
20.3 容器访问控制 .....	241		
20.4 映射容器端口到宿主主机 的实现 .....	243		
20.5 配置 docker0 网桥 .....	244		
20.6 自定义网桥 .....	245		
		<b>附录</b>	
		<b>A 常见问题汇总 .....</b>	<b>270</b>
		<b>B 常见仓库 .....</b>	<b>276</b>
		<b>C Docker 命令查询 .....</b>	<b>294</b>
		<b>D Docker 资源链接 .....</b>	<b>299</b>

## 第一部分 *Part 1*

# Docker 入门

- 第 1 章 初识 Docker
- 第 2 章 Docker 的核心概念和安装
- 第 3 章 镜像
- 第 4 章 容器
- 第 5 章 仓库
- 第 6 章 数据管理
- 第 7 章 网络基础配置
- 第 8 章 使用 Dockerfile 创建镜像

欢迎来到 Docker 的世界！

在这一部分里，笔者将介绍 Docker 的基础知识，本部分分为 8 章。

第 1 章介绍 Docker 开源项目以及它与现有的虚拟化技术，特别是 Linux 容器技术的关系；第 2 章介绍 Docker 的三大核心概念，以及如何在常见的操作系统环境中安装 Docker；第 3 章第 5 章通过具体的示例操作讲解 Docker 的常用命令；第 6 章剖析如何在 Docker 中使用数据卷来保存数据；第 7 章介绍如何使用容器网络，特别是使容器访问外网和其他容器；第 8 章介绍如何编写 Dockerfile，以及使用 Dockerfile 配置文件来创建镜像的具体方法和注意事项。

## 初识 Docker

如果说个人主机时代大家比拼的关键是 CPU 主频的高低和内存的大小，那么在云计算时代，虚拟化技术无疑是整座信息技术大厦最核心的一块基石。

伴随着信息技术产业的发展，虚拟化技术已经应用到各种关键场景中。从最早上世纪 60 年代 IBM 推出的大型主机虚拟化到后来 X86 平台上的虚拟化，虚拟化技术自身也在不断丰富和创新。

虚拟化既可以通过硬件模拟来实现，也可以通过操作系统来实现。而近些年出现的容器虚拟化方案，更是充分利用了操作系统本身已有的机制和特性，可以实现轻量级的虚拟化，甚至有人把它称为新一代的虚拟化技术。Docker 毫无疑问就是其中的佼佼者。

那么，什么是 Docker？它会带来什么好处？它跟现有虚拟化技术又有何关系呢？

本章在介绍 Docker 项目的起源和发展之后，会剖析 Docker 和 Linux 容器技术的密切联系，以及在开发和运维中使用 Docker 的突出优势。最后，还将阐述 Docker 在整个虚拟化领域中的定位。

### 1.1 什么是 Docker

#### Docker 开源项目

Docker 是基于 Go 语言实现的云开源项目，诞生于 2013 年初，最初发起者是 dotCloud 公司。Docker 自开源后受到广泛的关注和讨论，目前已有多个相关项目，逐渐形成了围绕 Docker 的生态体系。dotCloud 公司后来也改名为 Docker Inc，专注于 Docker 相关技术和产

品的开发。

Docker项目目前已加入了Linux基金会，遵循Apache 2.0协议，全部开源代码均在<https://github.com/docker/docker>上进行维护。在最近一次Linux基金会的调查中，Docker是仅次于OpenStack的最受欢迎的云计算开源项目。

现在主流的Linux操作系统都已经支持Docker。例如，Redhat RHEL 6.5/CentOS 6.5往上的操作系统、Ubuntu 14.04操作系统，都已经默认带有Docker软件包。Google公司宣称在其PaaS(Platform as a Service)平台及服务产品中广泛应用了Docker。微软公司宣布和Docker公司合作，以加强其云平台Azure对Docker的支持。公有云提供商亚马逊近期也推出了AWS EC2 Container，提供对Docker的支持。

Docker的主要目标是“Build, Ship and Run Any App, Anywhere”，即通过对应用组件的封装(Packaging)、分发(Distribution)、部署(Deployment)、运行(Runtime)等生命周期的管理，达到应用组件级别的“一次封装，到处运行”。这里的应用组件，既可以是一个Web应用，也可以是一套数据库服务，甚至是一个操作系统或编译器。

Docker基于Linux的多项开源技术提供了高效、敏捷和轻量级的容器方案，并且支持在多种主流云平台(PaaS)和本地系统上部署。可以说Docker为应用的开发和部署提供了“一站式”的解决方案。

## Linux 容器技术

Docker引擎的基础是Linux容器(Linux Containers, LXC)技术。IBM DeveloperWorks上给出了关于容器技术的准确描述：

容器有效地将由单个操作系统管理的资源划分到孤立的组中，以便更好地在孤立的组之间平衡有冲突的资源使用需求。与虚拟化相比，这样既不需要指令级模拟，也不需要即时编译。容器可以在核心CPU本地运行指令，而不需要任何专门的解释机制。此外，也避免了准虚拟化(paravirtualization)和系统调用替换中的复杂性。

Linux容器其实不是一个全新的概念。最早容器技术可以追溯到1982年Unix系列操作系统上的chroot工具(直到今天，主流的Unix、Linux操作系统仍然支持和带有该工具)。早期的容器实现技术包括Sun Solaris操作系统上的Solaris Containers(2004年发布)，FreeBSD操作系统上的FreeBSD jail(2000年左右出现)，以及GNU/Linux上的Linux-VServer(<http://linux-vserver.org/>) (2001年10月)和OpenVZ(<http://openvz.org>) (2005年)。

虽然这些技术经过多年的演化已经十分成熟，但是由于种种原因，这些容器技术并没有被集成到主流的Linux内核中，使用起来并不方便。例如，如果用户要使用OpenVZ技术，就需要先给操作系统打上特定的内核补丁方可使用。

后来LXC项目借鉴了前人成熟的容器设计理念，并基于一系列新的内核特性实现了更具扩展性的虚拟化容器方案。更加关键的是，LXC被集成到了主流Linux内核中，进而成为Linux系统轻量级容器技术的事实标准。

## 从 Linux 容器到 Docker

在 LXC 的基础上，Docker 进一步优化了容器的使用体验。Docker 提供了各种容器管理工具（如分发、版本、移植等）让用户无需关注底层的操作，可以简单明了地管理和使用容器。用户操作 Docker 容器就像操作一个轻量级的虚拟机那样简单。

读者可以简单地将 Docker 容器理解为一种沙盒（Sandbox）。每个容器内运行一个应用，不同的容器相互隔离，容器之间也可以建立通信机制。容器的创建和停止都十分快速，容器自身对资源的需求也十分有限，远远低于虚拟机。很多时候，甚至直接把容器当作应用本身也没有任何问题。

有理由相信，随着 Docker 技术的进一步成熟，它将成为更受欢迎的容器虚拟化技术实现，得到更广泛的应用。

## 1.2 为什么要使用 Docker

### Docker 容器虚拟化的好处

Docker 项目的发起人和 Docker Inc. 的 CTO Solomon Hykes 认为，Docker 在正确的地点、正确的时间顺应了正确的趋势——即高效地构建应用。现在开发者需要能方便地创建运行在云平台上的应用，也就是说应用必须能够脱离底层机器，而且同时必须是“任何时间任何地点”可获取的。因此，开发者们需要一种创建分布式应用程序的方式，这也是 Docker 所能够提供的。

举个简单的应用场景的例子。假设用户试图基于最常见的 LAMP（Linux + Apache + MySQL + PHP）组合来运维一个网站。按照传统的做法，首先，需要安装 Apache、MySQL 和 PHP 以及它们各自运行所依赖的环境；之后分别对它们进行配置（包括创建合适的用户、配置参数等）；经过大量的操作后，还需要进行功能测试，看是否工作正常；如果不正常，则意味着更多的时间代价和不可控的风险。可以想象，如果再加上更多的应用，事情会变得更加难以处理。

更为可怕的是，一旦需要服务器迁移（例如从阿里云迁移到腾讯云），往往需要重新部署和调试。这些琐碎而无趣的“体力活”，极大地降低了工作效率。

而 Docker 提供了一种更为聪明的方式，通过容器来打包应用，意味着迁移只需要在新的服务器上启动需要的容器就可以了。这无疑将节约大量的宝贵时间，并降低部署过程出现问题的风险。

### Docker 在开发和运维中的优势

对开发和运维（DevOps）人员来说，可能最梦寐以求的就是一次性地创建或配置，可以

在任意环境、任意时间让应用正常地运行。而 Docker 恰恰是可以实现这一终极目标的瑞士军刀。

具体说来，Docker 在开发和运维过程中，具有以下几个方面的优势。

- 更快速的交付和部署。使用 Docker，开发人员可以使用镜像来快速构建一套标准的开发环境；开发完成之后，测试和运维人员可以直接使用相同环境来部署代码。Docker 可以快速创建和删除容器，实现快速迭代，大量节约开发、测试、部署的时间。并且，各个步骤都有明确的配置和操作，整个过程全程可见，使团队更容易理解应用的创建和工作过程。
- 更高效的资源利用。Docker 容器的运行不需要额外的虚拟化管理程序（Virtual Machine Manager, VMM，以及 Hypervisor）支持，它是内核级的虚拟化，可以实现更高的性能，同时对资源的额外需求很低。
- 更轻松的迁移和扩展。Docker 容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等。这种兼容性让用户可以在不同平台之间轻松地迁移应用。
- 更简单的更新管理。使用 Dockerfile，只需要小小的配置修改，就可以替代以往大量的更新工作。并且所有修改都以增量的方式进行分发和更新，从而实现自动化并且高效的容器管理。

## Docker 与虚拟机比较

作为一种轻量级的虚拟化方式，Docker 在运行应用上跟传统的虚拟机方式相比具有显著优势：

- Docker 容器很快，启动和停止可以在秒级实现，这相比传统的虚拟机方式要快得多。
- Docker 容器对系统资源需求很少，一台主机上可以同时运行数千个 Docker 容器。
- Docker 通过类似 Git 的操作来方便用户获取、分发和更新应用镜像，指令简明，学习成本较低。
- Docker 通过 Dockerfile 配置文件来支持灵活的自动化创建和部署机制，提高工作效率。

Docker 容器除了运行其中的应用之外，基本不消耗额外的系统资源，保证应用性能的同时，尽量减小系统开销。传统虚拟机方式运行 N 个不同的应用就要启动 N 个虚拟机（每个虚拟机需要单独分配独占的内存、磁盘等资源），而 Docker 只需要启动 N 个隔离的容器，并将应用放到容器内即可。

当然，在隔离性方面，传统的虚拟机方式多了一层额外的隔离。但这并不意味着 Docker 就不安全。Docker 利用 Linux 系统上的多种防护机制实现了严格可靠的隔离。从 1.3 版本开始，Docker 引入了安全选项和镜像签名机制，极大地提高了使用 Docker 的安全性。

下表总结了使用 Docker 容器技术与传统虚拟机技术的特性比较。

特性	容器	虚拟机
启动速度	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个
隔离性	安全隔离	完全隔离

### 1.3 虚拟化与 Docker

虚拟化技术是一个通用的概念，在不同领域有不同的理解。在计算领域，一般指的是计算虚拟化（Computing Virtualization），或通常说的服务器虚拟化。维基百科上的定义如下：

在计算机技术中，虚拟化（Virtualization）是一种资源管理技术，是将计算机的各种实体资源，如服务器、网络、内存及存储等，予以抽象、转换后呈现出来，打破实体结构间的不可切割的障碍，使用户可以用比原本的组态更好的方式来应用这些资源。

可见，虚拟化的核心是对资源进行抽象，目标往往是为了在同一个主机上运行多个系统或应用，从而提高系统资源的利用率，同时带来降低成本、方便管理和容错容灾等好处。

从大类上分，虚拟化技术可分为基于硬件的虚拟化和基于软件的虚拟化。其中，真正意义上的基于硬件的虚拟化技术不多见，少数如网卡中的单根多 IO 虚拟化（Single Root I/O Virtualization and Sharing Specification，SR-IOV）等技术，也超出了本书的讨论范畴。

基于软件的虚拟化从对象所在的层次，又可以分为应用虚拟化和平台虚拟化（通常说的虚拟机技术即属于这个范畴）。其中，前者一般指的是一些模拟设备或 Wine 这样的软件。后者又可以细分为如下几个子类：

- 完全虚拟化。虚拟机模拟完整的底层硬件环境和特权指令的执行过程，客户操作系统无需进行修改。例如 VMware Workstation、VirtualBox、QEMU 等。
- 硬件辅助虚拟化。利用硬件（主要是 CPU）辅助支持（目前 x86 体系结构上可用的硬件辅助虚拟化技术包括 Intel-VT 和 AMD-V）处理敏感指令来实现完全虚拟化的功能，客户操作系统无需修改，例如 VMware Workstation、Xen、KVM。
- 部分虚拟化。只针对部分硬件资源进行虚拟化，客户操作系统需要进行修改。现在有些虚拟化技术的早期版本仅支持部分虚拟化。
- 超虚拟化（Paravirtualization）。部分硬件接口以软件的形式提供给客户机操作系统，客户操作系统需要进行修改，例如早期的 Xen。
- 操作系统级虚拟化。内核通过创建多个虚拟的操作系统实例（内核和库）来隔离不同的进程。容器相关技术即在这个范畴。

可见，Docker 以及其他容器技术都属于操作系统的虚拟化这个范畴。

Docker 虚拟化方式之所以拥有众多优势，这跟操作系统的虚拟化自身的特点是分不开

的。下面图 1-1 比较了 Docker 和常见的虚拟机方式的不同之处。

传统方式是在硬件层面实现虚拟化，需要有额外的虚拟机管理应用和虚拟机操作系统层。

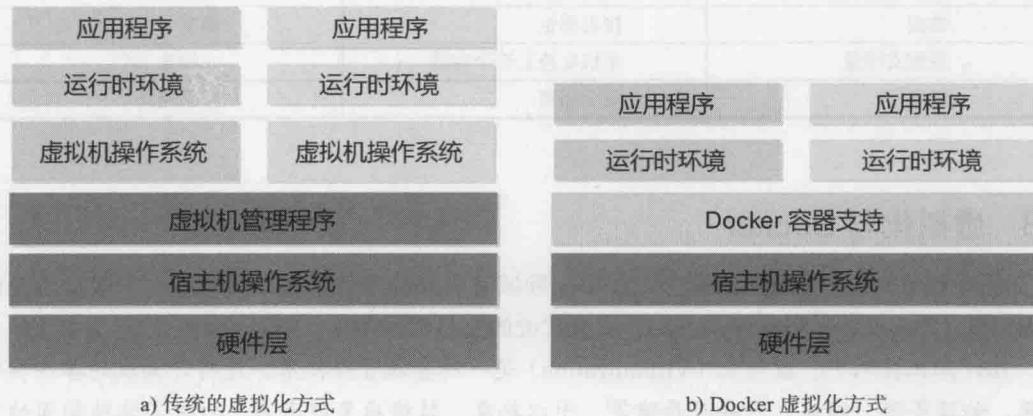


图 1-1 Docker 和传统的虚拟机方式的不同之处

Docker 容器是在操作系统层面上实现虚拟化，直接复用本地主机的操作系统，因此更加轻量级。

## 1.4 本章小结

通过本章内容的叙述，相信读者已经对于 Docker 技术不再陌生，并为它带来的众多优势所深深吸引。通过为 Linux 容器技术提供更简便的使用和管理方案、更高效的版本控制机制，Docker 让容器技术一下子变得前所未有的方便易用。

笔者相信，随着云计算技术的进一步发展，以 Docker 技术为代表的容器技术必将在整个虚拟化领域占有越来越重要的地位。

## Docker 的核心概念和安装

本章首先介绍 Docker 的三大核心概念：

- 镜像 (Image)
- 容器 (Container)
- 仓库 (Repository)

读者理解了这三个核心概念，就能顺利地理解 Docker 的整个生命周期。社区讨论很激烈的一个话题，就是 Docker 和 Linux 容器技术到底有何区别？相信读者在阅读完本章后，会得到更清晰的答案。

随后，笔者将介绍如何在常见的操作系统上安装 Docker，包括 Ubuntu、CentOS、Windows 和 MacOS 等。

### 2.1 核心概念

#### Docker 镜像

Docker 镜像 (Image) 类似于虚拟机镜像，可以将它理解为一个面向 Docker 引擎的只读模板，包含了文件系统。

例如：一个镜像可以只包含一个完整的 Ubuntu 操作系统环境，可以把它称为一个 Ubuntu 镜像。镜像也可以安装了 Apache 应用程序（或用户需要的其他软件），可以把它称为一个 Apache 镜像。

镜像是创建 Docker 容器的基础。通过版本管理和增量的文件系统，Docker 提供了一套十分简单的机制来创建和更新现有的镜像，用户甚至可以从网上下载一个已经做好的应用镜

像，并通过简单的命令就可以直接使用。

## Docker 容器

Docker 容器（Container）类似于一个轻量级的沙箱，Docker 利用容器来运行和隔离应用。

容器是从镜像创建的应用运行实例，可以将其启动、开始、停止、删除，而这些容器都是相互隔离、互不可见的。

读者可以把容器看做一个简易版的 Linux 系统环境（这包括 root 用户权限、进程空间、用户空间和网络空间等），以及运行在其中的应用程序打包而成的应用盒子。

镜像自身是只读的。容器从镜像启动的时候，Docker 会在镜像的最上层创建一个可写层，镜像本身将保持不变。

## Docker 仓库

Docker 仓库（Repository）类似于代码仓库，是 Docker 集中存放镜像文件的场所。

有时候会看到有资料将 Docker 仓库和注册服务器（Registry）混为一谈，并不严格区分。实际上，注册服务器是存放仓库的地方，其上往往存放着多个仓库。每个仓库集中存放某一类镜像，往往包括多个镜像文件，通过不同的标签（tag）来进行区分。例如存放 Ubuntu 操作系统镜像的仓库，称为 Ubuntu 仓库，其中可能包括 14.04、12.04 等不同版本的镜像。仓库注册服务器的示例如图 2-1 所示。

根据所存储的镜像公开分享与否，Docker 仓库可以分为公开仓库（Public）和私有仓库（Private）两种形式。

目前，最大的公开仓库是 Docker Hub，存放了数量庞大的镜像供用户下载。国内的公开仓库包括 Docker Pool 等，可以提供稳定的国内访问。

当然，用户如果不希望公开分享自己的镜像文件，Docker 也支持用户在本地网络内创建一个只能自己访问的私有仓库。

当用户创建了自己的镜像之后就可以使用 push 命令将它上传到指定的公有或者私有仓库。这样用户下次在另外一台机器上使用该镜像时，只需将其从仓库上 pull 下来就可以了。

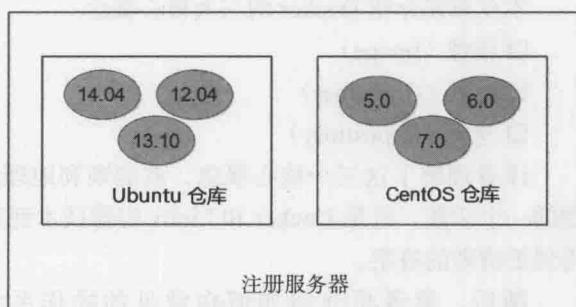


图 2-1 仓库和注册服务器

---

 **注意** 可以看出，Docker 利用仓库管理镜像的设计理念与 Git 非常相似。

## 2.2 安装 Docker

Docker 支持在主流的操作系统平台上使用，包括 Ubuntu、CentOS、Windows 以及 MacOS 系统等。当然，在 Linux 系列平台上是原生支持，使用体验也最好。

### Ubuntu

#### 1. Ubuntu 14.04 及以上版本

Ubuntu 14.04 版本官方软件源中已经自带了 Docker 包，可以直接安装：

```
$ sudo apt-get update
$ sudo apt-get install -y docker.io
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
$ sudo sed -i '$acomplete -F _docker docker' /etc/bash_completion.d/docker.io
```

以上流程使用 Ubuntu 14.04 系统默认自带 docker.io 安装包安装 Docker，这样安装的 Docker 版本相对较旧。

读者也可通过下面的方法从 Docker 官方源安装最新版本。首先需要安装 apt-transport-https，并添加 Docker 官方源：

```
$ sudo apt-get install apt-transport-https
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 36A1D7
869245C8950F966E92D8576A8BA88D21E9
$ sudo bash -c "echo deb https://get.docker.io/ubuntu docker main > /etc/apt/
sources.list.d/docker.list"
$ sudo apt-get update
```

之后，可以通过下面的命令来安装最新版本的 Docker：

```
$ sudo apt-get install -y lxc-docker
```

在安装了 Docker 官方软件源后，若需要更新 Docker 软件版本，只需要执行以下命令即可升级：

```
$ sudo apt-get update -y lxc-docker
```



**注意** 后文中使用 \$ 作为终端引导符时，表示非 root 权限用户；# 代表是 root 用户。

#### 2. ubuntu 14.04 以下的版本

如果使用的是较低版本的 Ubuntu 系统，则需要先进行内核更新并重启系统后再进行安装：

```
$ sudo apt-get update
```

```
$ sudo apt-get install -y linux-image-generic-lts-raring linux-headers-generic-lts-raring
$ sudo reboot
```

重启后，重复在 Ubuntu 14.04 系统的安装步骤即可。

## CentOS

Docker 支持 CentOS 6 及以后的版本。

对于 CentOS 6 系统可使用 EPEL 库安装 Docker，命令如下：

```
$ sudo yum install -y http://mirrors.yun-idc.com/epel/6/i386/epel-release-6-8.noarch.rpm
$ sudo yum install -y docker-io
```

对于 CentOS 7 系统，由于 CentOS-Extras 源中已内置 Docker，读者可以直接使用 yum 命令进行安装：

```
$ sudo yum install -y docker
```

目前在 Centos 系统中更新 Docker 软件有两种方法，一是自行通过源码编译安装，二是下载二进制文件进行更新。

## Windows

目前 Docker 官方已经宣布 Docker 通过虚拟机方式支持 Windows 7.1 和 8，前提是主机的 CPU 支持硬件虚拟化。由于近几年发布的 Intel 和 AMD CPU 基本上都已支持了硬件虚拟化特性，因此在 Windows 中使用 Docker 通常不会有硬件支持的问题。

由于 Docker 引擎使用了 Linux 内核特性，所以在 Windows 上运行的话，需要额外使用一个虚拟机来提供 Linux 支持。这里推荐使用 Boot2Docker 工具，它会首先安装一个经过加工与配置的 VirtualBox 轻量级虚拟机，然后在其中运行 Docker。主要步骤如下：

- 1) 从 <https://docs.docker.com/installation/windows/> 下载最新官方 Docker for Windows Installer。

- 2) 运行 Installer。这个过程将安装 VirtualBox, MSYS-git, boot2docker Linux ISO 镜像，以及 Boot2Docker 管理工具。如图 2-2 所示。

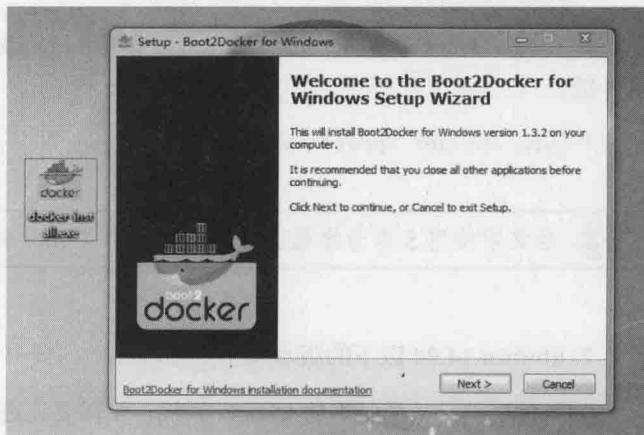


图 2-2 Windows 下安装 Docker

3) 打开桌面的 Boot2Docker Start 程序，或者用以下命令：Program Files > Boot2Docker for Windows。此初始化脚本在第一次运行时需要输入一个 SSH Key Passphrase (用于 SSH 密钥生成的口令)。读者可以自行设定，也可以直接按回车键，跳过此设定，如图 2-3 所示。

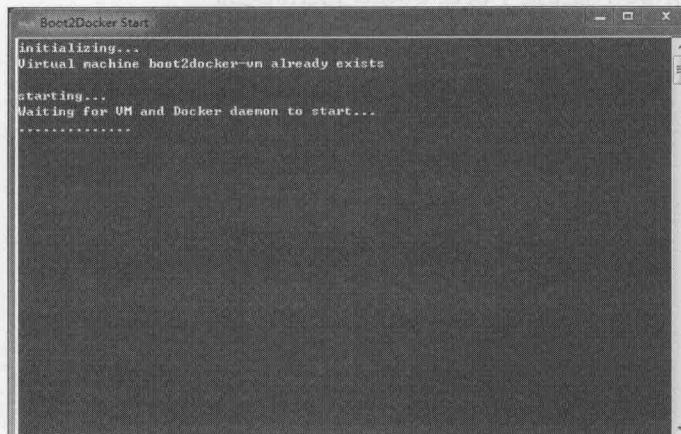


图 2-3 Boot2Docker 启动后界面

此时 Boot2Docker Start 程序将连接至虚拟机中的 Shell 会话，Docker 已经运行起来了！

## Mac OS

目前 Docker 已经支持 Mac OS X 10.6 Snow Leopard 及以上版本的 Mac OS。

在 Mac OS 上使用 Docker，同样需要 Boot2Docker 工具的支持。主要步骤如下：

1) 下载最新官方 Docker for OS X Installer。读者可以从 <https://docs.docker.com/installation/mac/> 下载。

2) 双击运行安装包。这个过程将安装一个 VirtualBox 虚拟机、Docker 本身以及 Boot2Docker 管理工具，如图 2-4 所示。

3) 安装成功后，找到 Boot2Docker (Mac 系统的 Application 或“应用”文件夹中) 并运行它。现在进行 Boot2Docker 的初始化：

```
$ boot2docker init
$ boot2docker start
$ $(boot2docker shellinit)
```

读者将看到虚拟机在命令行窗口中启动运行，并显示 Docker 的启动信息，则说明 Docker 安装成功。当虚拟机初始化完毕后，可以使用 `boot2docker stop` 和 `boot2docker start` 来控制它。

注意：如果在命令行中看到如下提示信息：

To connect the Docker client to the Docker daemon, please set: export DOCKER\_HOST=tcp://192.168.59.103:2375

可以执行提示信息中的语句: `export DOCKER_HOST=tcp://192.168.59.103:2375`。此语句的作用是在系统环境变量中设置 Docker 的主机地址。

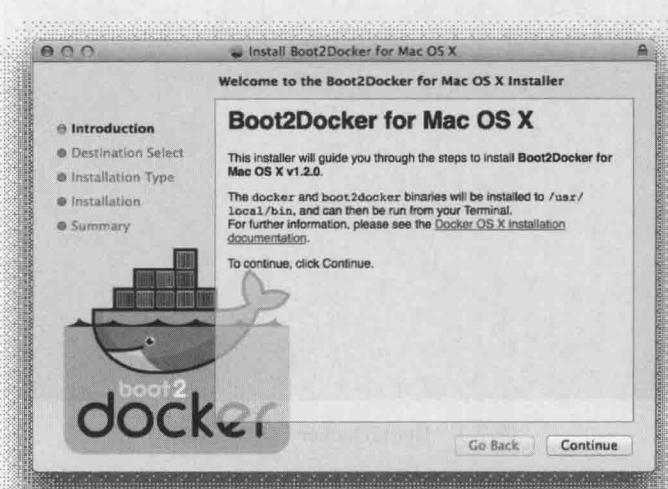


图 2-4 Mac OS 上安装 Boot2Docker

## 2.3 本书环境介绍

本书的实践环境是一台装有 Linux Mint 17 的笔记本电脑，并使用虚拟机软件 VirturBox 虚拟了一套 Ubuntu 14.04 系统，两套系统上都安装了 Docker 的 1.3 版本，虚拟机通过 VirturBox 网络的 NAT 方式连接到外部，如图 2-5 所示。

其中，Ubuntu 14.04 虚拟机将是主要的操作环境（自动获取的 IP 地址为 10.0.2.15/24），而笔记本上装的 Linux Mint 环境（内网地址为 10.0.2.2/24，外网地址为 192.168.1.0/24 段地址）将作

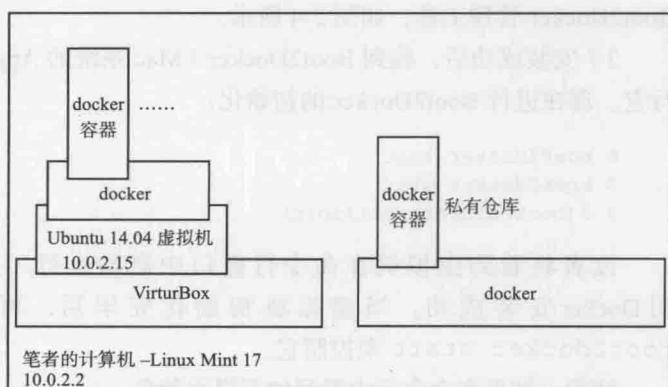


图 2-5 本书环境



为本地私有仓库的服务器，演示跟仓库相关的操作。

读者可根据自己本地环境，选择搭建类似环境。

## 2.4 本章小结

本章介绍了 Docker 的三大核心概念：镜像、容器和仓库。

通过这三大核心概念所构建的高效工作流程，毫无疑问，正是 Docker 得以从众多容器虚拟化方案中脱颖而出的重要原因。

熟悉 Git 和 GitHub 的读者，会理解这一工作流程为文件分发和合作所带来的众多优势。在后续章节，笔者将进一步地介绍围绕这三大核心概念的 Docker 常见操作命令。

## 第3章 镜像

镜像是 Docker 的三大核心概念之一。

Docker 运行容器前需要本地存在对应的镜像，如果镜像不存在本地，Docker 会尝试先从默认镜像仓库下载（默认使用 Docker Hub 公共注册服务器中的仓库），用户也可以通过配置，使用自定义的镜像仓库。

本章将介绍围绕镜像这一核心概念的具体操作，包括如何使用 pull 命令从 Docker Hub 仓库中下载镜像到本地；如何查看本地已有的镜像信息；如何在远端仓库使用 search 命令进行搜索和过滤；如何删除镜像标签和镜像文件；如何创建用户定制的镜像并且保存为外部文件。最后，还将介绍如何向 Docker Hub 仓库中推送自己的镜像。

### 3.1 获取镜像

镜像是 Docker 运行容器的前提。

读者可以使用 docker pull 命令从网络上下载镜像。该命令的格式为 docker pull NAME [:TAG]。对于 Docker 镜像来说，如果不显式地指定 TAG，则默认会选择 latest 标签，即下载仓库中最新版本的镜像。

下面，笔者将演示如何从 Docker Hub 的 Ubuntu 仓库下载一个最新的 Ubuntu 操作系统的镜像。

```
$ sudo docker pull ubuntu
ubuntu:latest: The image you are pulling has been verified
```

```
d497ad3926c8: Downloading [=====>
```

```
] 25.41 MB/201.6 MB 51m14s
```

```
ccb62158e970: Download complete
e791be0477f2: Download complete
3680052c0f5c: Download complete
22093c35d77b: Download complete
5506de2b643b: Download complete
511136ea3c5a: Download complete
```

该命令实际上下载的就是 `ubuntu:latest` 镜像，目前最新的 14.04 版本的镜像。

下载过程中可以看出，镜像文件一般由若干层组成，行首的 `2185fd50e2ca` 这样的字符串代表了各层的 ID。下载过程中会获取并输出镜像的各层信息。层（Layer）其实是 AUFS（Advanced Union File System，一种联合文件系统）中的重要概念，是实现增量保存与更新的基础。

读者还可以通过指定标签来下载特定版本的某一个镜像，例如 `14.04` 标签的镜像。

```
$ sudo docker pull ubuntu: 14.04
```

上面两条命令实际上都相当于 `$ sudo docker pull registry.hub.docker.com/ubuntu:latest` 命令，即从默认的注册服务器 `registry.hub.docker.com` 中的 `ubuntu` 仓库来下载标记为 `latest` 的镜像。

用户也可以选择从其他注册服务器的仓库下载。此时，需要在仓库名称前指定完整的仓库注册服务器地址。例如从 DockerPool 社区的镜像源 `dl.dockerpool.com` 下载最新的 Ubuntu 镜像。

```
$ sudo docker pull dl.dockerpool.com:5000/ubuntu
```

下载镜像到本地后，即可随时使用该镜像了，例如利用该镜像创建一个容器，在其中运行 `bash` 应用。

```
$ sudo docker run -t -i ubuntu /bin/bash
root@fe7fc4bd8fc9: #
```

## 3.2 查看镜像信息

使用 `docker images` 命令可以列出本地主机上已有的镜像。

例如，下面的命令列出了本地刚从官方下载的 `ubuntu:14.04` 镜像，以及从 DockerPool 镜像源下载的 `ubuntu:latest` 镜像。

```
$ sudo docker images
REPOSITORY          TAG        IMAGE ID      CREATED       VIRTUAL SIZE
ubuntu              14.04     5506de2b643b   1 weeks ago   197.8 MB
dl.dockerpool.com:5000/ubuntu  latest    5506de2b643b   1 weeks ago   197.8 MB
```

在列出信息中，可以看到几个字段信息：

- 来自于哪个仓库，比如 ubuntu 仓库。
- 镜像的标签信息，比如 14.04。
- 镜像的 ID 号（唯一）。
- 创建时间。
- 镜像大小。

其中镜像的 ID 信息十分重要，它唯一标识了镜像。

TAG 信息用于标记来自同一个仓库的不同镜像。例如 ubuntu 仓库中有多个镜像，通过 TAG 信息来区分发行版本，包括 10.04、12.04、12.10、13.04、14.04 等标签。

为了方便在后续工作中使用这个镜像，还可以使用 docker tag 命令为本地镜像添加新的标签。例如添加一个新的 ubuntu:latest 镜像标签如下：

```
$ sudo docker tag dl.dockerpool.com:5000/ubuntu:latest ubuntu:latest
```

再次使用 docker images 列出本地主机上镜像信息，可以看到多了一个 ubuntu:latest 标签的镜像。

```
$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED       VIRTUAL SIZE
ubuntu              14.04   5506de2b643b   1 weeks ago   197.8 MB
dl.dockerpool.com:5000/ubuntu latest   5506de2b643b   1 weeks ago   192.8 MB
ubuntu              latest   5506de2b643b   1 weeks ago   192.8 MB
```

细心的读者可能会注意到，这些不同标签的镜像的 ID 是完全一致的，说明它们实际上指向了同一个镜像文件，只是别名不同而已。标签在这里起到了引用或快捷方式的作用。

使用 docker inspect 命令可以获取该镜像的详细信息。

```
$ sudo docker inspect 5506de2b643b
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "/bin/bash"
        ],
        "CpuShares": 0,
        "Cpuset": "",
        "Domainname": "",
        "Entrypoint": null,
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "ExposedPorts": null,
        "Labels": {}
    },
    "ContainerId": "5506de2b643b",
    "Container": "5506de2b643b",
    "Created": "2015-07-20T11:12:40.000000000Z",
    "Docker": "5506de2b643b",
    "Id": "5506de2b643b",
    "Image": "5506de2b643b",
    "Labels": {},
    "LastUsed": "2015-07-20T11:12:40.000000000Z",
    "Name": "/ubuntu_14.04_14.04_1",
    "Os": "linux",
    "Platform": "linux/amd64",
    "Ports": null,
    "Size": 197840,
    "Status": "running"
}]
```

```
"Hostname": "065262ce3c91",
"Image": "964692831e07f7362f5c3fedf0c4b81a622f2c6e3ec5f19d0edff21af64c12",
"Memory": 0,
"MemorySwap": 0,
"NetworkDisabled": false,
"OnBuild": [],
"OpenStdin": false,
"PortSpecs": null,
"StdinOnce": false,
"Tty": false,
"User": "",
"Volumes": null,
"WorkingDir": ""

},
"Container": "f26bc14cc07412402bdab911b8a935fead0322649cf042cee8515c02ebdfa53a",
"ContainerConfig": {
    "AttachStderr": false,
    "AttachStdin": false,
    "AttachStdout": false,
    "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) CMD [/bin/bash]"
    ],
    "CpuShares": 0,
    "Cpuset": "",
    "Domainname": "",
    "Entrypoint": null,
    "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "ExposedPorts": null,
    "Hostname": "065262ce3c91",
    "Image": "964692831e07f7362f5c3fedf0c4b81a622f2c6e3ec5f19d0edff21af64c12",
    "Memory": 0,
    "MemorySwap": 0,
    "NetworkDisabled": false,
    "OnBuild": [],
    "OpenStdin": false,
    "PortSpecs": null,
    "StdinOnce": false,
    "Tty": false,
    "User": "",
    "Volumes": null,
    "WorkingDir": ""

},
"Created": "2014-09-23T22:37:05.812213629Z",
"DockerVersion": "1.2.0",
"Id": "53bf7a53e8903fce40d24663901aac6211373a8d8b4effe08bc884e63e181805",
"Os": "linux",
"Parent": "964692831e07f7362f5c3fedf0c4b81a622f2c6e3ec5f19d0edff21af64c12",
```

```

        "Size": 0
    }
]

```

docker inspect 命令返回的是一个 JSON 格式的消息，如果我们只要其中一项内容时，可以使用 -f 参数来指定，例如，获取镜像的 Architecture 信息：

```
$ sudo docker inspect -f {{".Architecture"}} 550
amd64
```

在指定镜像 ID 的时候，通常使用该 ID 的前若干个字符组成的可区分字符串来替代完整的 ID。

### 3.3 搜索镜像

使用 docker search 命令可以搜索远端仓库中共享的镜像，默认搜索 Docker Hub 官方仓库中的镜像。用法为 docker search TERM，支持的参数包括：

- --automated=false 仅显示自动创建的镜像。
- --no-trunc=false 输出信息不截断显示。
- -s, --stars=0 指定仅显示评价为指定星级以上的镜像。

例如，搜索带 mysql 关键字的镜像如下所示：

```
$ sudo docker search mysql
NAME          DESCRIPTION          STARS      OFFICIAL      AUTOMATED
NAME          DESCRIPTION          STARS      OFFICIAL      AUTOMATED
mysql          MySQL is a widely used, open-source relati...  213      [OK]
tutum/mysql    MySQL Server image - listens in port 3306....  74       [OK]
orchardup/mysql
tutum/lamp     LAMP image - Apache listens in port 80, an...  32       [OK]
tutum/wordpress Wordpress Docker image - listens in port 8...  26       [OK]
paintedfox/mariadb A docker image for running MariaDB 5.5, a ...  21      [OK]
dockerfile/mysql Trusted automated MySQL (http://dev.mysql....  14      [OK]
google/mysql   MySQL server for Google Compute Engine        13      [OK]
anapsix/gitlab-ci GitLab-CI Continuous Integration in Docker...  12      [OK]
centurylink/drupal Drupal docker image without a DB included ...  11      [OK]
stenote/docker-lemp MySQL 5.6, PHP 5.5, Nginx, Memcache        10      [OK]
...

```

可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、星级（表示该镜像的受欢迎程度）、是否官方创建、是否自动创建等。

默认的输出结果将按照星级评价进行排序。官方的镜像说明是官方项目组创建和维护的，automated 资源则允许用户验证镜像的来源和内容。

## 3.4 删除镜像

### 使用镜像的标签删除镜像

使用 docker rmi 命令可以删除镜像，命令格式为 docker rmi IMAGE [IMAGE... ]，其中 IMAGE 可以为标签或 ID。

例如，要删除掉 dl.dockerpool.com:5000/ubuntu:latest 镜像，可以使用如下命令：

```
$ sudo docker rmi dl.dockerpool.com:5000/ubuntu
Untagged: dl.dockerpool.com:5000/ubuntu:latest
```

读者可能会担心，本地的 ubuntu:latest 镜像是否会受到此命令的影响。无需担心，当同一个镜像拥有多个标签的时候，docker rmi 命令只是删除了该镜像多个标签中的指定标签而已，并不影响镜像文件。因此上述操作相当于只是删除了镜像 5506de2b643b 的一个标签而已。

为保险起见，再次查看本地的镜像，发现 ubuntu:latest 镜像（准确地说，是 5506de2b643b 镜像）仍然存在：

```
$ sudo docker images
REPOSITORY      TAG          IMAGE ID       CREATED        VIRTUAL SIZE
ubuntu          14.04        5506de2b643b   1 weeks ago   197.8 MB
ubuntu          latest        5506de2b643b   1 weeks ago   192.8 MB
```

但当镜像只剩下一个标签的时候就要小心了，此时再使用 docker rmi 命令会彻底删除该镜像。

假设本地存在一个标签为 mysql:latest 的镜像，且没有额外的标签指向它，执行 docker rmi 命令，可以看出它会删除这个镜像文件的所有 AUFS 层：

```
$ sudo docker rmi mysql:latest
Untagged: mysql:latest
Deleted: 9a09222edf600a03ea48bd23cfa363841e45a8715237e3a58cb0167f0e8bad54
Deleted: 4daeda4ad839a152a3b649672bd5135977d7f81866d3bc0e16d0af3f65cc8af6
Deleted: cf07a411bf0883bd632940e8108dac49c64456a47f7390507de5685bb6daf85
Deleted: 4f513746df18b222a07bb8d76d4b6d29752ce5dcbb69bfad0ce92e6c1449a3821
Deleted: 228ecd435c8a29d25b7799036701a27f2d67874c915bb8eb9fb175b1f98aa60
Deleted: 37e4b3932afa186924a09eb332bc8ebec3aac8bac074314ed9a2d1e94547f50
Deleted: 898883ccfcce705e440547e30e240cb025c12410d7c9e4d2bcb11973ba075975
Deleted: 0a09ddcf99b7fd8fcbb3525c41b54696038ecf13677f4459f1c98c742ffa60ab2
Deleted: 35bc8591e39be5089265a093e234d13a4b155a01d2ab9e8904eafa81664fb597
Deleted: 857e856e4481d59ee88a4cdedd9aaef855666bd494fa38506e6788361c0af4cda
```

### 使用镜像 ID 删除镜像

当使用 docker rmi 命令后面跟上镜像的 ID（也可以是 ID 能进行区分的部分前缀串）

时，会先尝试删除所有指向该镜像的标签，然后删除该镜像文件本身。

注意，当有该镜像创建的容器存在时，镜像文件默认是无法被删除的，例如：

先利用 ubuntu 镜像创建一个简单的容器，输出一句话“hello! I am here!”：

```
$ sudo docker run ubuntu echo 'hello! I am here!'
hello! I am here!
```

使用 docker ps -a 命令可以看到本机上存在的所有容器：

```
$ sudo docker ps -a
CONTAINER ID        IMAGE       COMMAND       CREATED          STATUS          PORTS      NAMES
e812617b41f6        ubuntu:latest "echo 'hello! I am h"   13 seconds ago
                           Exited (0) 12 seconds ago   silly_leakey
```

可以看到，后台存在一个退出状态的容器，是刚基于 ubuntu:latest 镜像创建的。

试图删除该镜像，Docker 会提示有容器正在运行，无法删除：

```
$ sudo docker rmi ubuntu
Error response from daemon: Conflict, cannot delete 5506de2b643b because the
container e812617b41f6 is using it, use -f to force
2014/10/16 18:10:31 Error: failed to remove one or more images
```

如果要想强行删除镜像，可以使用 -f 参数：

```
$ sudo docker rmi -f ubuntu
```

笔者不推荐使用 -f 参数来强制删除一个存在容器依赖的镜像，因为这样往往会造成一些遗留问题。

再次使用 docker images 查看本地的镜像列表，读者会发现一个标签为 <none> 的临时镜像，原来被强制删除的镜像换了新的 ID 继续存在系统中。

```
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED          VIRTUAL SIZE
<none>          <none>   2318d26665ef    3 months ago    198.7 MB
```

因此，正确的做法是，先删除依赖该镜像的所有容器，再来删除镜像。首先删除容器 e812617b41f6：

```
$ sudo docker rm e81
e81
```

此时再使用临时的 ID 来删除镜像，此时会正常打印出删除的各层信息：

```
core@localhost ~ $ docker rmi -f 2318d26665ef
Deleted: 2318d26665eff33e9f91c4c99036751afb40eb58f944a585372bec1407828ad3
Deleted: ebc34468f71dca9cb9937bf4c33062540bcacae148df8a70053bfd1acbcaa20
Deleted: 25f11f5fb0cb9e41531d1da8dc56351286427e070c536f7015fe76e4dae0a4bc
Deleted: 9bad880da3d219b10423804147d6982da1a7bb1e285777a4d746afca6215bebb
Deleted: 511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
```

此时查看本地镜像，读者会发现临时镜像已经被删除：

```
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
```

## 3.5 创建镜像

创建镜像的方法有三种：基于已有镜像的容器创建、基于本地模板导入、基于 Dockerfile 创建。

本节将重点介绍前两种方法。最后一种基于 Dockerfile 创建的方法将在后续章节专门予以详细介绍。

### 基于已有镜像的容器创建

该方法主要是使用 docker commit 命令，其命令格式为 docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]，主要选项包括：

- a, --author="" 作者信息。
- m, --message="" 提交消息。
- p, --pause=true 提交时暂停容器运行。

下面将演示如何使用该命令创建一个新镜像。首先，启动一个镜像，并在其中进行修改操作，例如创建一个 test 文件，之后退出：

```
$ sudo docker run -ti ubuntu:14.04 /bin/bash
root@a925cb40b3f0:/# touch test
root@a925cb40b3f0:/# exit
```

记住容器的 ID 为 a925cb40b3f0。

此时该容器跟原 ubuntu:14.04 镜像相比，已经发生了改变，可以使用 docker commit 命令来提交为一个新的镜像。提交时可以使用 ID 或名称来指定容器：

```
$ sudo docker commit -m "Added a new file" -a "Docker Newbee" a925cb40b3f0 test
9e9c814023bcfffc3e67e892a235afe61b02f66a947d2747f724bd317dda02f27
```

顺利的话，命令会返回新创建的镜像的 ID 信息，例如：

```
9e9c814023bcfffc3e67e892a235afe61b02f66a947d2747f724bd317dda02f27
```

此时查看本地镜像列表，即可看到新创建的镜像：

```
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
test           latest   9e9c814023bc      4 seconds ago    225.4 MB
```

## 基于本地模板导入

也可以直接从一个操作系统模板文件导入一个镜像。在这里，推荐使用 OpenVZ 提供的模板来创建。OPENVZ 模板的下载地址为 <http://openvz.org/Download/templates/precreated>。

比如，笔者下载了一个 ubuntu-14.04 的模板压缩包后，可以使用以下命令导入：

```
$ sudo cat ubuntu-14.04-x86_64-minimal.tar.gz | docker import - ubuntu:14.04
```

然后查看新导入的镜像，已经在本地存在了：

```
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
ubuntu          14.04   05ac7c0b9383   17 seconds ago  215.5 MB
```

## 3.6 存出和载入镜像

可以使用 `docker save` 和 `docker load` 命令来存出和载入镜像。

### 存出镜像

如果要存出镜像到本地文件，可以使用 `docker save` 命令。例如，存出本地的 `ubuntu:14.04` 镜像为文件 `ubuntu_14.04.tar`：

```
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
ubuntu          14.04   c4ff7513909d   5 weeks ago  225.4 MB
...
$ sudo docker save -o ubuntu_14.04.tar ubuntu:14.04
```

### 载入镜像

可以使用 `docker load` 从存出的本地文件中再导入到本地镜像库，例如从文件 `ubuntu_14.04.tar` 导入镜像到本地镜像列表，如下所示：

```
$ sudo docker load --input ubuntu_14.04.tar
```

或

```
$ sudo docker load < ubuntu_14.04.tar
```

这将导入镜像以及其相关的元数据信息（包括标签等），可以使用 `docker images` 命令进行查看。

## 3.7 上传镜像

可以使用 `docker push` 命令上传镜像到仓库，默认上传到 DockerHub 官方仓库（需要登录），命令格式为 `docker push NAME[:TAG]`。

用户在 DockerHub 网站注册后，即可上传自制的镜像。例如用户 `user` 上传本地的 `test:latest` 镜像，可以先添加新的标签 `user/test:latest`，然后用 `docker push` 命令上传镜像：

```
$ sudo docker tag test:latest user/test:latest
$ sudo docker push user/test:latest
The push refers to a repository [base/163] (len: 1)
Sending image list

Please login prior to push:
Username:
Password:
Email: xxx@xxx.com
```

第一次使用时，会提示输入登录信息或进行注册。

## 3.8 本章小结

本章具体介绍了围绕 Docker 镜像的一系列重要命令操作，包括获取、查看、搜索、删除、创建等。

读者可能已经发现，镜像是使用 Docker 的前提，也是最重要的资源。所以，在平时的 Docker 使用中，推荐大家注意积累定制的镜像文件，并将自己创建的高质量镜像分享到社区中。

在后续章节，笔者将会介绍更多对镜像进行操作的场景。

要将 Docker 容器与宿主机隔离，可以使用 Docker 容器的网络功能。Docker 容器的网络功能非常强大，可以实现容器与宿主机、容器与容器之间的通信。通过 Docker 容器的网络功能，可以轻松地实现容器间的通信。

*Chapter 4*

## 第4章 容器

容器是 Docker 的另一个核心概念。

简单地说，容器是镜像的一个运行实例，所不同的是，它带有额外的可写文件层。

如果认为虚拟机是模拟运行的一整套操作系统（提供了运行态环境和其他系统环境）和跑在上面的应用。那么 Docker 容器就是独立运行的一个或一组应用，以及它们的必需运行环境。

本章将具体介绍围绕容器的重要操作，包括创建一个容器、启动容器、终止一个容器、进入容器内执行操作、删除容器和通过导入导出容器来实现容器迁移等。

### 4.1 创建容器

Docker 的容器十分轻量级，用户可以随时创建或删除容器。

#### 新建容器

可以使用 `docker create` 命令新建一个容器，例如：

```
$ sudo docker create -it ubuntu:latest
7a0c26f96889de46b6276608501b7e8f99e4e31e42ec4a288a1f8e7644316637
$ sudo docker ps -a
CONTAINER ID        IMAGE       COMMAND       CREATED      STATUS      PORTS      NAMES
7a0c26f96889        ubuntu:latest   "/bin/bash"   6 seconds ago   stoic_albattani
```

使用 `docker create` 命令新建的容器处于停止状态，可以使用 `docker start` 命令来启动它。

## 新建并启动容器

启动容器有两种方式，一种是基于镜像新建一个容器并启动，另外一个是将在终止状态 (stopped) 的容器重新启动。所需要的命令主要为 docker run，等价于先执行 docker create 命令，再执行 docker start 命令。

例如，下面的命令输出一个“Hello World”，之后容器自动终止：

```
$ sudo docker run ubuntu /bin/echo 'Hello world'
Hello world
```

这跟在本地直接执行 /bin/echo 'hello world' 几乎感觉不出任何区别。

当利用 docker run 来创建并启动容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载。
- 利用镜像创建并启动一个容器。
- 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层。
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去。
- 从地址池配置一个 IP 地址给容器。
- 执行用户指定的应用程序。
- 执行完毕后容器被终止。

下面的命令则启动一个 bash 终端，允许用户进行交互：

```
$ sudo docker run -t -i ubuntu:14.04 /bin/bash
root@af8bae53bdd3:/#
```

其中，-t 选项让 Docker 分配一个伪终端 (pseudo-tty) 并绑定到容器的标准输入上，-i 则让容器的标准输入保持打开。

在交互模式下，用户可以通过所创建的终端来输入命令，例如：

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@af8bae53bdd3:/# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   11 ?        00:00:00 ps
```

在容器内用 ps 命令查看进程，可以看到，只运行了 bash 应用，并没有运行其他不需要的进程。

用户可以按 Ctrl+d 或输入 exit 命令来退出容器：

```
root@af8bae53bdd3:/# exit
exit
```

对于所创建的 bash 容器，当使用 exit 命令退出之后，该容器就自动处于终止状态

了。这是因为对于 Docker 容器来说，当运行的应用（此处例子中为 bash）退出后，容器也就没有继续运行的必要了。

## 守护态运行

更多的时候，需要让 Docker 容器在后台以守护态（Daemonized）形式运行。用户可以通过添加 -d 参数来实现。

例如下面的命令会在后台运行容器：

```
$ sudo docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"
ce554267d7a4c34eefc92c5517051dc37b918b588736d0823e4c846596b04d83
```

容器启动后会返回一个唯一的 ID，也可以通过 docker ps 命令来查看容器信息：

```
$ sudo docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED          STATUS          PORTS     NAMES
ce554267d7a4        ubuntu:latest   "/bin/sh -c 'while t
Up About a minute                           About a minute ago
                                            determined_pik"
```

要获取容器的输出信息，可以通过 docker logs 命令：

```
$ sudo docker logs ce5
hello world
hello world
hello world
...
.
```

## 4.2 终止容器

可以使用 docker stop 来终止一个运行中的容器，命令的格式为 docker stop [-t | --time [=10]]。它会首先向容器发送 SIGTERM 信号，等待一段时间后（默认为 10 秒），再发送 SIGKILL 信号终止容器。

此外，当 Docker 容器中指定的应用终结时，容器也自动终止。例如对于上一节中只启动了一个终端的容器，用户通过 exit 命令或 Ctrl+d 来退出终端时，所创建的容器立刻终止。

另外，可以使用 docker stop 来终止一个运行中的容器：

```
$ sudo docker stop ce5
ce5
```



注意

docker kill 命令会直接发送 SIGKILL 信号来强行终止容器。

可以使用 docker ps -a -q 命令看到处于终止状态的容器的 ID 信息。例如：

```
$ sudo docker ps -a -q
ce554267d7a4
d58050081fe3
e812617b41f6
```

处于终止状态的容器，可以通过 `docker start` 命令来重新启动：

```
$ sudo docker start ce5
ce5
$ sudo docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
ce554267d7a4      ubuntu:latest      "/bin/sh -c 'while t
Up 5 seconds                                determined_pike"
```

此外，`docker restart` 命令会将一个运行态的容器终止，然后再重新启动它：

```
$ sudo docker restart ce5
ce5
$ sudo docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
ce554267d7a4      ubuntu:latest      "/bin/sh -c 'while t
Up 14 seconds                                determined_pike"
```

## 4.3 进入容器

在使用 `-d` 参数时，容器启动后会进入后台，用户无法看到容器中的信息。某些时候如果需要进入容器进行操作，有多种方法，包括使用 `docker attach` 命令、`docker exec` 命令，以及 `nsenter` 工具等。

### attach 命令

`docker attach` 是 Docker 自带的命令。下面示例如何使用该命令：

```
$ sudo docker run -dit ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ sudo docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
243c32535da7      ubuntu:latest      "/bin/bash"
Up 17 seconds                                nostalgic_hypatia
$ sudo docker attach nostalgic_hypatia
root@243c32535da7:/#
```

但是使用 `attach` 命令有时候并不方便。当多个窗口同时 `attach` 到同一个容器的时候，所有窗口都会同步显示。当某个窗口因命令阻塞时，其他窗口也无法执行操作了。

## exec 命令

Docker 自 1.3 版本起，提供了一个更加方便的工具 exec，可以直接在容器内运行命令。例如进入到刚创建的容器中，并启动一个 bash：

```
$ sudo docker exec -ti 243c32535da7 /bin/bash
root@243c32535da7:/#
```

## nsenter 工具

nsenter 工具在 util-linux 包 2.23 版本后包含。如果系统中 util-linux 包没有该命令，可以按照下面的方法从源码安装：

```
$ cd /tmp; curl https://www.kernel.org/pub/linux/utils/util-linux/v2.24/util-linux-2.24.tar.gz | tar -zxf-; cd util-linux-2.24;
$ ./configure --without-ncurses
$ make nsenter && sudo cp nsenter /usr/local/bin
```

为了使用 nsenter 连接到容器，还需要找到容器的进程的 PID，可以通过下面的命令获取：

```
PID=$(docker inspect --format "{{ .State.Pid }}" <container>)
```

通过这个 PID，就可以连接到这个容器：

```
$ nsenter --target $PID --mount --uts --ipc --net --pid
```

下面给出一个完整的例子：

```
$ sudo docker run -idt ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ sudo docker ps
CONTAINER ID        IMAGE        COMMAND      CREATED       STATUS        PORTS     NAMES
243c32535da7        ubuntu:latest   "/bin/bash"   18 seconds ago
Up 17 seconds          nostalgic_hypatia
$ PID=$(docker-pid 243c32535da7)
10981
$ sudo nsenter --target 10981 --mount --uts --ipc --net --pid
root@243c32535da7:/#
root@ce554267d7a4:/# w
 11:07:36 up  3:14,  0 users,  load average: 0.00, 0.02, 0.05
USER        TTY        FROM                  LOGIN@        IDLE        JCPU      PCPU WHAT
root@ce554267d7a4:/# ps -ef
# 通过 ps -ef 命令可以看到容器中运行的进程
UID        PID    PPID  C STIME TTY          TIME CMD
root         1      0  0 10:56 ?
root         1      0  0 11:07 ?
root        699      0  0 11:07 ?
root        716      1  0 11:07 ?
root        717     699  0 11:07 ?
root         1      0  0 00:00:00 /bin/sh -c while true; do echo
hello world; sleep 1; done
root        699      0  0 00:00:00 /bin/bash
root        716      1  0 00:00:00 sleep 1
root        717     699  0 00:00:00 ps -ef
```

## 4.4 删除容器

可以使用 docker rm 命令删除处于终止状态的容器，命令格式为 docker rm [OPTIONS] CONTAINER [CONTAINER...]。支持的选项包括：

- ❑ -f, --force=false 强行终止并删除一个运行中的容器。
- ❑ -l, --link=false 删除容器的连接，但保留容器。
- ❑ -v, --volumes=false 删除容器挂载的数据卷。

例如，查看处于终止状态的容器并删除如下所示：

```
$ sudo docker ps -a
CONTAINER ID        IMAGE      COMMAND       CREATED          STATUS          PORTS      NAMES
ce554267d7a4        ubuntu:latest    "/bin/sh -c 'while t"   3 minutes ago
Exited (-1) 13 seconds ago           determined_pike
d58050081fe3        ubuntu:latest    "/bin/bash"     About an hour ago   Exited (0)
About an hour ago           berserk_brattain
e812617b41f6        ubuntu:latest    "echo 'hello! I am h"   2 hours ago
Exited (0) 3 minutes ago

$ sudo docker rm ce554267d7a4
$ ce554267d7a4
```

如果要删除一个运行中的容器，可以添加 -f 参数。Docker 会发送 SIGKILL 信号给容器，终止其中的应用：

```
$ sudo docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep
1; done"
2aeed76caf8292c7da6d24c3c7f3a81a135af942ed1707a79f85955217d4dd594
$ sudo docker rm 2ae
Error response from daemon: You cannot remove a running container. Stop the
container before attempting removal or use -f
2014/11/03 04:05:24 Error: failed to remove one or more containers
$ sudo docker rm -f 2ae
2ae
```

## 4.5 导入和导出容器

### 导出容器

导出容器是指导出一个已经创建的容器到一个文件，不管此时这个容器是否处于运行状态，可以使用 docker export 命令，该命令格式为 docker export CONTAINER。

查看所有的容器如下所示：

```
$ sudo docker ps -a
```

```

CONTAINER ID        IMAGE       COMMAND      CREATED     STATUS      PORTS      NAMES
ce554267d7a4        ubuntu:latest " /bin/sh -c 'while t'"   3 minutes ago
Exited (-1) 13 seconds ago      determined_pike
d58050081fe3        ubuntu:latest " /bin/bash"
ago    Exited (0) About an hour ago berserk_brattain
e812617b41f6        ubuntu:latest "echo 'hello! I am h'"
Exited (0) 3 minutes ago      silly_leakey

```

分别导出 ce554267d7a4 容器和 e812617b41f6 容器到 test\_for\_run.tar 文件和 test\_for\_stop.tar 文件：

```

$ sudo docker export ce5 >test_for_run.tar
$ ls
test_for_run.tar
$ sudo docker export e81 >test_for_stop.tar
$ ls
test_for_run.tar test_for_stop.tar

```

可将这些文件传输到其他机器上，在其他机器上通过导入命令实现容器的迁移。

## 导入容器

导出的文件又可以使用 docker import 命令导入，成为镜像，例如：

```

$ cat test_for_run.tar | sudo docker import - test/ubuntu:v1.0
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
test/ubuntu      v1.0    9d37a6082e97  About a minute ago  171.3 MB

```

读者可能会记得，笔者在之前章节曾介绍过使用 docker load 命令来导入一个镜像文件。实际上，既可以使用 docker load 命令来导入镜像存储文件到本地的镜像库，又可以使用 docker import 命令来导入一个容器快照到本地镜像库。这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息（即仅保存容器当时的快照状态），而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

## 4.6 本章小结

容器是直接提供应用服务的组件，也是 Docker 实现快速的启停和高效服务性能的基础。

通过本章内容的介绍和示例，相信已经掌握了对容器整个生命周期进行管理的各项操作命令。

在生产环境中，因为容器自身的轻量级特性，笔者推荐使用容器时在容器前段引入 HA（高可靠性）机制，例如使用 HAProxy 工具来代理容器访问，这样在容器出现故障时候，可以快速切换到其他容器，还可以自动重启故障容器。

仓库（Repository）是集中存放镜像的地方。

一个容易与之混淆的概念是注册服务器（Registry）。实际上注册服务器是存放仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像。从这方面来说，仓库可以被认为是一个具体的项目或目录。例如对于仓库地址 `dl.dockerpool.com/ubuntu` 来说，`dl.dockerpool.com` 是注册服务器地址，`ubuntu` 是仓库名。

仓库又分公共仓库和私有仓库，在本章笔者将分别展示如何使用 DockerHub 官方仓库进行登录、下载等基本操作，以及使用 DockerPool 社区提供的仓库下载镜像；最后还将介绍创建和使用私有仓库的基本操作。

## 5.1 Docker Hub

目前 Docker 官方维护了一个公共仓库 <https://hub.docker.com>，其中已经包括 15 000 多个的镜像。大部分需求都可以通过在 Docker Hub 中直接下载镜像来实现，如图 5-1 所示。

### 登录

可以通过执行 `docker login` 命令来输入用户名、密码和邮箱来完成注册和登录。注册成功后，本地用户目录的 `.dockercfg` 中将保存用户的认证信息。

### 基本操作

用户无需登录即可通过 `docker search` 命令来查找官方仓库中的镜像，并利用

`docker pull` 命令来将它下载到本地。



图 5-1 Docker Hub

在搜寻镜像的章节，已经具体介绍了如何使用 `docker pull` 命令。例如以 `centos` 为关键词进行搜索：

```
$ sudo docker search centos
NAME          DESCRIPTION          STARS      OFFICIAL      AUTOMATED
centos        The official build of CentOS.          465      [OK]
tianon/centos  CentOS 5 and 6, created using rinse instead...  28
blalor/centos Bare-bones base CentOS 6.5 image          6      [OK]
saltstack/centos-6-minimal          6      [OK]
tutum/centos-6.4   DEPRECATED. Use tutum/centos:6.4 instead....  5      [OK]
...
...
```

根据是否为官方提供，可将这些镜像资源分为两类。一种是类似 `centos` 这样的基础镜像，称为基础或根镜像。这些镜像是由 Docker 公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。

还有一种类型，比如 `tianon/centos` 镜像，它是由 DockerHub 的用户 `tianon` 创建并维护的，带有用户名称为前缀，表明是某用户的某仓库。可以通过用户名前缀 `username/` 来指定使用某个用户提供的镜像，比如 `tianon` 用户的镜像前缀为 `tianon/`。

另外，在查找的时候通过 `-s N` 参数可以指定仅显示评价为 N 星以上的镜像。

下载官方 `centos` 镜像到本地如下所示：

```
$ sudo docker pull centos
Pulling repository centos
0b443ba03958: Download complete
539c0211cd76: Download complete
511136ea3c5a: Download complete
7064731afe90: Download complete
```

用户也可以在登录后通过 `docker push` 命令来将本地镜像推送到 Docker Hub。

## 自动创建

自动创建 (Automated Builds) 功能对于需要经常升级镜像内程序来说十分方便。有时候，用户创建了镜像，安装了某个软件，如果软件发布新版本则需要手动更新镜像。

而自动创建功能使得用户通过 Docker Hub 指定跟踪一个目标网站（目前支持 GitHub 或 BitBucket）上的项目，一旦项目发现新的提交，则自动执行创建。

要配置自动创建，包括如下的步骤：

- 1) 创建并登录 Docker Hub，以及目标网站；\* 在目标网站中连接帐户到 Docker Hub。
- 2) 在 Docker Hub 中配置一个自动创建。
- 3) 选取一个目标网站中的项目（需要含 Dockerfile）和分支。
- 4) 指定 Dockerfile 的位置，并提交创建。

之后，可以在 Docker Hub 的“自动创建”页面中跟踪每次创建的状态。

## 5.2 Docker Pool 简介

Docker Pool (<http://dockerpool.com>) 是国内专业的 Docker 技术社区，目前也提供了官方镜像的下载管理服务，如图 5-2 所示。



图 5-2 Docker Pool

## 查看镜像

访问 <http://www.dockerpool.com/downloads>，即可看到已有的仓库和存储的镜像，包括 CentOS、Ubuntu、Java、Mongo、MySQL、Nginx 等热门仓库和镜像。

以 CentOS 仓库为例，其中包括 Centos 5、CentOS 6 和 CentOS 7 等镜像。

## 下载镜像

下载镜像也是使用 docker pull 命令，但是要在镜像名称前添加注册服务器的具体地址 dl.dockerpool.com:5000。

例如，要下载 ubuntu 仓库的 12.04 镜像，可以使用：

```
$ sudo docker pull dl.dockerpool.com:5000/ubuntu:12.04
```

通过 docker images 命令来查看下载到本地的镜像：

```
$ sudo docker images
dl.dockerpool.com:5000/ubuntu      12.04      ae7818fad1bc      1 min ago      116.2 MB
```

下载后，可以更新镜像的标签，与官方标签保持一致：

```
$ sudo docker tag dl.dockerpool.com:5000/ubuntu:12.04 ubuntu:12.04
```

需要注意的是，从 Docker Pool 下载的镜像文件，与官方镜像文件是完全一致的。

另外，阿里云等服务商也已经提供了 Docker 镜像的下载服务。当然，最便捷的方式还是搭建本地的仓库服务器，将在后续章节予以介绍。

## 5.3 创建和使用私有仓库

### 使用 registry 镜像创建私有仓库

安装 Docker 后，可以通过官方提供的 registry 镜像来简单搭建一套本地私有仓库环境：

```
$ sudo docker run -d -p 5000:5000 registry
```

这将自动下载并启动一个 registry 容器，创建本地的私有仓库服务。

默认情况下，会将仓库创建在容器的 /tmp/registry 目录下。可以通过 -v 参数来将镜像文件存放在本地的指定路径上。

例如下面的例子将上传的镜像放到 /opt/data/registry 目录：

```
$ sudo docker run -d -p 5000:5000 -v /opt/data/registry:/tmp/registry registry
```

此时，在本地将启动一个私有仓库服务，监听端口为 5000。

## 管理私有仓库镜像

首先在本书环境的笔记本上（Linux Mint）搭建私有仓库，查看其地址为 10.0.2.2:5000。然后在虚拟机系统（Ubuntu 14.04）里测试上传和下载镜像。

在 Ubuntu 14.04 系统查看已有的镜像：

```
$ sudo docker images
REPOSITORY      TAG          IMAGE ID      CREATED        VIRTUAL SIZE
ubuntu          14.04       ba5877dc9bec   6 days ago    199.3 MB
```

使用 docker tag 命令将这个镜像标记为 10.0.2.2:5000/test（格式为 docker tag IMAGE[:TAG] [REGISTRYHOST/] [USERNAME/]NAME[:TAG]）：

```
$ sudo docker tag ubuntu:14.04 10.0.2.2:5000/test
$ sudo docker images
REPOSITORY      TAG          IMAGE ID      CREATED        VIRTUAL SIZE
ubuntu          14.04       ba5877dc9bec   6 days ago    199.3 MB
10.0.2.2:5000/test      latest       ba5877dc9bec   6 days ago    199.3 MB
```

使用 docker push 上传标记的镜像：

```
$ sudo docker push 10.0.2.2:5000/test
The push refers to a repository [10.0.2.2:5000/test] (len: 1)
Sending image list
Pushing repository 10.0.2.2:5000/test (1 tags)
Image 511136ea3c5a already pushed, skipping
Image 9bad880da3d2 already pushed, skipping
Image 25f11f5fb0cb already pushed, skipping
Image ebc34468f71d already pushed, skipping
Image 2318d26665ef already pushed, skipping
Image ba5877dc9bec already pushed, skipping
Pushing tag for rev [ba5877dc9bec] on {http://10.0.2.2:5000/v1/repositories/test/tags/latest}
```

用 curl 查看仓库 10.0.2.2:5000 中的镜像：

```
$ curl http://10.0.2.2:5000/v1/search
{"num_results": 1, "query": "", "results": [{"description": "", "name": "library/test"}]}
```

在结果中可以看到 {"description": "", "name": "library/test"}，表明镜像已经成功上传了。

现在可以到任意一台能访问到 10.0.2.2 地址的机器去下载这个镜像了：

```
$ sudo docker pull 10.0.2.2:5000/test
Pulling repository 10.0.2.2:5000/test
ba5877dc9bec: Download complete
511136ea3c5a: Download complete
9bad880da3d2: Download complete
```

```
25f11f5fb0cb: Download complete  
ebc34468f71d: Download complete  
2318d26665ef: Download complete  
$ sudo docker images  
REPOSITORY          TAG      IMAGE ID      CREATED        VIRTUAL SIZE  
10.0.2.2:5000/test  latest   ba5877dc9bec  6 days ago   199.3 MB
```

下载后，还可以添加一个更通用的标签 `ubuntu:14.04`：

```
$ sudo docker tag 10.0.2.2:5000/test ubuntu:14.04
```

## 5.4 本章小结

仓库概念的引入，为 Docker 镜像文件的分发和管理提供了便捷的途径。

本章介绍的 DockerHub 和 DockerPool 两个公共仓库服务，可以方便个人用户进行镜像的下载和使用等操作。

在企业的生产环境中，则往往需要使用私有仓库来维护内部镜像。在本书第二部分的实战案例中，将介绍私有仓库的更多配置选项。

## 数据管理

用户在使用 Docker 的过程中，往往需要能查看容器内应用产生的数据，或者需要把容器内的数据进行备份，甚至多个容器之间进行数据的共享，这必然涉及容器的数据管理操作。

容器中管理数据主要有两种方式：

- 数据卷 (Data Volumes)
- 数据卷容器 (Data Volume Containers)

本章将首先介绍如何在容器内创建数据卷，并且把本地的目录或文件挂载到容器内的数据卷中。接下来，会介绍如何使用数据卷容器在容器和主机、容器和容器之间共享数据，并实现数据的备份和恢复。

### 6.1 数据卷

数据卷是一个可供容器使用的特殊目录，它绕过文件系统，可以提供很多有用的特性：

- 数据卷可以在容器之间共享和重用。
- 对数据卷的修改会立马生效。
- 对数据卷的更新，不会影响镜像。
- 卷会一直存在，直到没有容器使用。

数据卷的使用，类似于 Linux 下对目录或文件进行 mount 操作。

#### 在容器内创建一个数据卷

在用 docker run 命令的时候，使用 -v 标记可以在容器内创建一个数据卷。多次使用 -v 标记可以创建多个数据卷。

下面使用 `training/webapp` 镜像创建一个 Web 容器，并创建一个数据卷挂载到容器的 `/webapp` 目录：

```
$ sudo docker run -d -P --name web -v /webapp training/webapp python app.py
```



**注意** `-P` 是允许外部访问容器需要暴露的端口。

## 挂载一个主机目录作为数据卷

使用 `-v` 标记也可以指定挂载一个本地的已有目录到容器中去作为数据卷：

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python app.py
```

上面的命令加载主机的 `/src/webapp` 目录到容器的 `/opt/webapp` 目录：

这个功能在进行测试的时候十分方便，比如用户可以放置一些程序或数据到本地目录中，然后在容器内运行和使用。另外，本地目录的路径必须是绝对路径，如果目录不存在，Docker 会自动创建。

Docker 挂载数据卷的默认权限是读写 (`rw`)，用户也可以通过，`ro` 指定为只读：

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp:ro training/webapp python app.py
```

加入了 `:ro` 之后，容器内挂载的数据卷的数据就无法修改了。

## 挂载一个本地主机文件作为数据卷

`-v` 标记也可以从主机挂载单个文件到容器中作为数据卷：

```
$ sudo docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu /bin/bash
```

这样就可以记录在容器输入过的命令历史了。



**注意** 如果直接挂载一个文件到容器，使用文件编辑工具，包括 `vi` 或者 `sed --in-place` 的时候，可能会造成文件 `inode` 的改变，从 Docker 1.1.0 起，这会导致报错信息。所以推荐的方式是直接挂载文件所在的目录。

## 6.2 数据卷容器

如果用户需要在容器之间共享一些持续更新的数据，最简单的方式是使用数据卷容器。数据卷容器其实就是一个普通的容器，专门用它提供数据卷供其他容器挂载使用方法如下。

首先，创建一个数据卷容器 `dbdata`，并在其中创建一个数据卷挂载到 `/dbdata`：

```
$ sudo docker run -it -v /dbdata --name dbdata ubuntu
root@3ed94f279b6f:/#
```

查看 /dbdata 目录：

```
root@3ed94f279b6f:/# ls
bin boot dbdata dev etc home lib lib64 media mnt opt proc root run
sbin srv sys tmp usr var
```

然后，可以在其他容器中使用 `--volumes-from` 来挂载 dbdata 容器中的数据卷，例如创建 db1 和 db2 两个容器，并从 dbdata 容器挂载数据卷：

```
$ sudo docker run -it --volumes-from dbdata --name db1 ubuntu
$ sudo docker run -it --volumes-from dbdata --name db2 ubuntu
```

此时，容器 db1 和 db2 都挂载同一个数据卷到相同的 /dbdata 目录。三个容器任何一方在该目录下的写入，其他容器都可以看到。

例如，在 dbdata 容器中创建一个 test 文件：

```
root@3ed94f279b6f:/# cd /dbdata
root@3ed94f279b6f:/dbdata# touch test
root@3ed94f279b6f:/dbdata# ls
test
```

在 db1 容器内查看它：

```
$ sudo docker run -it --volumes-from dbdata --name db1 ubuntu
root@4128d2d804b4:/# ls
bin boot dbdata dev etc home lib lib64 media mnt opt proc root run
sbin srv sys tmp usr var
root@4128d2d804b4:/# ls dbdata/
test
```

可以多次使用 `--volumes-from` 参数来从多个容器挂载多个数据卷。还可以从其他已经挂载了容器卷的容器来挂载数据卷：

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

---

 **注意** 使用 `--volumes-from` 参数所挂载数据卷的容器自身并不需要保持在运行状态。

如果删除了挂载的容器（包括 dbdata、db1 和 db2），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时显式使用 `docker rm -v` 命令来指定同时删除关联的容器。

使用数据卷容器可以让用户在容器之间自由地升级和移动数据卷。具体的操作将在下一节中进行讲解。

## 6.3 利用数据卷容器迁移数据

可以利用数据卷容器对其中的数据卷进行备份、恢复，以实现数据的迁移。

### 备份

使用下面的命令来备份 dbdata 数据卷容器内的数据卷：

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

这个命令稍微有点复杂，具体分析下。

首先利用 ubuntu 镜像创建了一个容器 worker。使用 `--volumes-from dbdata` 参数来让 worker 容器挂载 dbdata 容器的数据卷（即 dbdata 数据卷）；使用 `-v $(pwd):/backup` 参数来挂载本地的当前目录到 worker 容器的 `/backup` 目录。

worker 容器启动后，使用了 `tar cvf /backup/backup.tar /dbdata` 命令来将 `/dbdata` 下内容备份为容器内的 `/backup/backup.tar`，即宿主主机当前目录下的 `backup.tar`。

### 恢复

如果要恢复数据到一个容器，可以按照下面的操作。首先创建一个带有数据卷的容器 dbdata2：

```
$ sudo docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

然后创建另一个新的容器，挂载 dbdata2 的容器，并使用 `untar` 解压备份文件到所挂载的容器卷中即可：

```
$ sudo docker run --volumes-from dbdata2 -v $(pwd):/backup busybox tar xvf
/backup/backup.tar
```

## 6.4 本章小结

数据是最宝贵的资源，Docker 无疑为数据管理提供了充分的支持。

本章介绍了通过数据卷和数据卷容器对容器内数据进行共享、备份和恢复等操作，通过这些机制，即使容器在运行中出现故障，用户也不必担心数据发生丢失，只需要快速地重新创建容器即可。

在生产环境中，笔者推荐在使用数据卷或数据卷容器之外，定期将主机的本地数据进行备份，或者使用支持容错的存储系统，包括 RAID 或分布式文件系统，如 Ceph、GPFS、HDFS 等。

## 第7章

# 网络基础配置

大量的互联网应用服务包括多个服务组件，这往往需要多个容器之间通过网络通信进行相互配合。

Docker 目前提供了映射容器端口到宿主主机和容器互联机制来为容器提供网络服务。

本章将讲解如何使用 Docker 的网络功能。包括使用端口映射机制来将容器内应用服务提供给外部网络，以及通过容器互联系统让多个容器之间进行快捷的网络通信。

## 7.1 端口映射实现访问容器

### 从外部访问容器应用

在启动容器的时候，如果不指定对应参数，在容器外部是无法通过网络来访问容器内的网络应用和服务的。

当容器中运行一些网络应用，要让外部访问这些应用时，可以通过 -P 或 -p 参数来指定端口映射。当使用 -P 标记时，Docker 会随机映射一个 49000 ~ 49900 的端口至容器内部开放的网络端口：

```
$ sudo docker run -d -P training/webapp python app.py
$ sudo docker ps -l
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
bc533791f3f5 training/webapp:latest python app.py 5 seconds ago Up 2 seconds
0.0.0.0:49155->5000/tcp nostalgic_morse
```

此时，可以使用 docker ps 看到，本地主机的 49155 被映射到了容器的 5000 端口。

访问宿主主机的 49115 端口即可访问容器内 Web 应用提供的界面。

同样，可以通过 `docker logs` 命令来查看应用的信息：

```
$ sudo docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 -- [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 -- [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -
```

`-p`（小写的）则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。支持的格式有 `ip:hostPort:containerPort` | `ip::containerPort` | `hostPort:containerPort`。

## 映射所有接口地址

使用 `hostPort:containerPort` 格式将本地的 5000 端口映射到容器的 5000 端口，可以执行如下命令：

```
$ sudo docker run -d -p 5000:5000 training/webapp python app.py
```

此时默认会绑定本地所有接口上的所有地址。多次使用 `-p` 标记可以绑定多个端口。例如：

```
$ sudo docker run -d -p 5000:5000 -p 3000:80 training/webapp python app.py
```

## 映射到指定地址的指定端口

可以使用 `ip:hostPort:containerPort` 格式指定映射使用一个特定地址，比如 `localhost` 地址 `127.0.0.1`：

```
$ sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py
```

## 映射到指定地址的任意端口

使用 `ip::containerPort` 绑定 `localhost` 的任意端口到容器的 5000 端口，本地主机会自动分配一个端口：

```
$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python app.py
```

还可以使用 `udp` 标记来指定 `udp` 端口：

```
$ sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

## 查看映射端口配置

使用 `docker port` 来查看当前映射的端口配置，也可以查看到绑定的地址：

```
$ sudo docker port nostalgic_morse 5000
127.0.0.1:49155.
```



**注意** 容器有自己的内部网络和 IP 地址（使用 `docker inspect`+ 容器 ID 可以获取所有的变量值）。

## 7.2 容器互联实现容器间通信

容器的连接（linking）系统是除了端口映射外另一种可以与容器中应用进行交互的方式。它会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

### 自定义容器命名

连接系统依据容器的名称来执行。因此，首先需要自定义一个好记的容器命名。

虽然当创建容器的时候，系统默认会分配一个名字，但自定义命名容器有两个好处：

- 自定义的命名，比较好记，比如一个 Web 应用容器，我们可以给它起名叫 `web`。
  - 当要连接其他容器时候，可以作为一个有用的参考点，比如连接 `Web` 容器到 `db` 容器。
- 使用 `--name` 标记可以为容器自定义命名：

```
$ sudo docker run -d -P --name web training/webapp python app.py
```

使用 `docker ps` 来验证设定的命名：

```
$ sudo docker ps -l
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
aed84ee21bde training/webapp:latest python app.py 12 hours ago Up 2 seconds
0.0.0.0:49154->5000/tcp web
```

也可以使用 `docker inspect` 来查看容器的名字：

```
$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde
/web
```



**注意** 容器的名称是唯一的。如果已经命名了一个叫 `web` 的容器，当你要再次使用 `web` 这个名称的时候，需要先用 `docker rm` 来删除之前创建的同名容器。

在执行 `docker run` 的时候如果添加 `-- rm` 标记，则容器在终止后会立刻删除。注意，`-- rm` 和 `- d` 参数不能同时使用。

## 容器互联

使用 `--link` 参数可以让容器之间安全的进行交互。

下面先创建一个新的数据库容器：

```
$ sudo docker run -d --name db training/postgres
```

删除之前创建的 web 容器：

```
$ sudo docker rm -f web
```

然后创建一个新的 web 容器，并将它连接到 db 容器：

```
$ sudo docker run -d -P --name web --link db:db training/webapp python app.py
```

此时，db 容器和 web 容器建立互联关系。

`--link` 参数的格式为 `--link name:alias`，其中 `name` 是要链接的容器的名称，`alias` 是这个连接的别名。

使用 `docker ps` 来查看容器的连接：

```
$ sudo docker ps
CONTAINER ID IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
349169744e49  training/postgres:latest  su postgres -c '/usr About a minute
ago  Up About a minute  5432/tcp      db, web/db
aed84ee21bde  training/webapp:latest    python app.py      16 hours ago
Up 2 minutes   0.0.0.0:49154->5000/tcp  web
```

可以看到自定义命名的容器：db 和 web，db 容器的 names 列有 db 也有 web/db。这表示 web 容器链接到 db 容器，这允许 web 容器访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主主机上。在启动 db 容器的时候并没有使用 `-p` 和 `-P` 标记，从而避免了暴露数据库端口到外部网络上。

Docker 通过两种方式为容器公开连接信息：

□ 环境变量。

□ 更新 `/etc/hosts` 文件。

使用 `env` 命令来查看 web 容器的环境变量：

```
$ sudo docker run --rm --name web2 --link db:db training/webapp env
.
.
.
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5000_TCP=tcp://172.17.0.5:5432
DB_PORT_5000_TCP_PROTO=tcp
DB_PORT_5000_TCP_PORT=5432
DB_PORT_5000_TCP_ADDR=172.17.0.5
.
.
```

其中 DB\_ 开头的环境变量是供 web 容器连接 db 容器使用，前缀采用大写的连接别名。

除了环境变量，Docker 还添加 host 信息到父容器的 /etc/hosts 的文件。下面是父容器 web 的 hosts 文件：

```
$ sudo docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7 aed84ee21bde
...
172.17.0.5 db
```

这里有两个 hosts 信息，第一个是 web 容器，web 容器用自己的 id 作为默认主机名，第二个是 db 容器的 ip 和主机名。可以在 web 容器中安装 ping 命令来测试跟 db 容器的连通：

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db
PING db (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

用 ping 来测试 db 容器，它会解析成 172.17.0.5。注意，官方的 ubuntu 12.04 镜像默认没有安装 ping，需要自行安装。

用户可以链接多个子容器到父容器，比如可以链接多个 web 到 db 容器上。

### 7.3 本章小结

网络是云时代最核心也是最复杂的系统之一。本章通过具体案例讲解了 Docker 涉及网络的基本操作，包括基础的容器端口映射机制和容器互联机制。Docker 目前采用了 Linux 系统自带的网络系统来实现对网络服务的支持，这既可以利用现有成熟的技术提供稳定支持，又可以实现快速的高性能转发。

在生产环境中，网络方面的需求更加复杂和多变，这时候往往就需要引入额外的机制，例如 SDN（软件定义网络）或 NFV（网络功能虚拟化）的相关技术。本书的第三部分将进一步探讨 Docker 网络的高级配置和实现细节。

## 第8章 使用 Dockerfile 创建镜像

Dockerfile 是一个文本格式的配置文件，用户可以使用 Dockerfile 快速创建自定义的镜像。

本章首先将介绍 Dockerfile 典型的基本结构及其支持的众多指令，并具体讲解通过这些指令来编写定制镜像的 Dockerfile。

最后，会介绍使用 Dockerfile 创建镜像的过程。

### 8.1 基本结构

Dockerfile 由一行行命令语句组成，并且支持以 # 开头的注释行。

一般而言，Dockerfile 分为四部分：基础镜像信息、维护者信息、镜像操作指令和容器启动时执行指令。例如：

```
# This dockerfile uses the ubuntu image
# VERSION 2 - EDITION 1
# Author: docker_user
# Command format: Instruction [arguments / command] ..

# 第一行必须指定基于的基础镜像
FROM ubuntu

# 维护者信息
MAINTAINER docker_user docker_user@email.com

# 镜像的操作指令
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

# 容器启动时执行指令
CMD /usr/sbin/nginx
```

其中，一开始必须指明所基于的镜像名称，接下来一般会说明维护者信息。

后面则是镜像操作指令，例如 RUN 指令，RUN 指令将对镜像执行跟随的命令。每运行一条 RUN 指令，镜像添加新的一层，并提交。最后是 CMD 指令，来指定运行容器时的操作命令。

下面是两个 Dockerhub 上的 Dockerfile 的例子，读者可以对 Dockerfile 结构有个基本的感知。

第一个是在 ubuntu 父镜像基础上安装 inotify-tools、nginx、apache2、openssh-server 等软件，从而创建一个新的 Nginx 镜像：

```
# Nginx
#
# VERSION 0.0.1

FROM      ubuntu
MAINTAINER Victor Vieux <victor@docker.com>

RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server
```

第二个也是基于 ubuntu 父镜像，安装 **firefox** 和 vnc 软件，启动后，用户可以通过 5900 端口通过 vnc 方式使用 **firefox**：

```
# Firefox over VNC
#
# VERSION 0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir /.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD   ["x11vnc", "-forever", "-usepw", "-create"]
```

## 8.2 指令

指令的一般格式为 INSTRUCTION arguments，指令包括 FROM、MAINTAINER、

RUN 等。下面分别介绍。

### 1. FROM

格式为 FROM <image> 或 FROM<image>:<tag>。

第一条指令必须为 FROM 指令。并且，如果在同一个 Dockerfile 中创建多个镜像时，可以使用多个 FROM 指令（每个镜像一次）。

### 2. MAINTAINER

格式为 MAINTAINER <name>，指定维护者信息。

### 3. RUN

格式为 RUN <command> 或 RUN ["executable", "param1", "param2"]。

前者将在 shell 终端中运行命令，即 /bin/sh -c；后者则使用 exec 执行。指定使用其他终端可以通过第二种方式实现，例如 RUN ["/bin/bash", "-c", "echo hello"]。

每条 RUN 指令将在当前镜像基础上执行指定命令，并提交为新的镜像。当命令较长时可以使用 \ 来换行。

### 4. CMD

支持三种格式：

- 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 命令。如果指定了多条命令，只有最后一条会被执行。

如果用户启动容器时候指定了运行的命令，则会覆盖掉 CMD 指定的命令。

### 5. EXPOSE

格式为 EXPOSE <port> [<port>...]。

例如：

```
EXPOSE 22 80 8443
```

告诉 Docker 服务端容器暴露的端口号，供互联系统使用。在启动容器时需要通过 -P，Docker 主机会自动分配一个端口转发到指定的端口；使用 -p，则可以具体指定哪个本地端口映射过来。

## 6. ENV

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。例如：

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJc /usr/
src/postgres && ...
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

## 7. ADD

格式为 `ADD <src> <dest>`。

该命令将复制指定的 `<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（文件或目录）；也可以是一个 URL；还可以是一个 tar 文件（自动解压为目录）。

## 8. COPY

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，文件或目录）为容器中的 `<dest>`。目标路径不存在时，会自动创建。

当使用本地目录为源目录时，推荐使用 `COPY`。

## 9. ENTRYPOINT

有两种格式：

```
ENTRYPOINT ["executable", "param1", "param2"]
```

```
ENTRYPOINT command param1 param2 (shell 中执行)。
```

配置容器启动后执行的命令，并且不可被 `docker run` 提供的参数覆盖。

每个 Dockerfile 中只能有一个 `ENTRYPOINT`，当指定多个 `ENTRYPOINT` 时，只有最后一个生效。

## 10. VOLUME

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和需要保持的数据等。

## 11. USER

格式为 `USER daemon`。

指定运行容器时的用户名或 UID，后续的 RUN 也会使用指定用户。

当服务不需要管理员权限时，可以通过该命令指定运行用户。并且可以在之前创建所需要的用户，例如：`RUN groupadd -r postgres && useradd -r -g postgres postgres`。要临时获取管理员权限可以使用 `gosu`，而不推荐 `sudo`。

## 12. WORKDIR

格式为 `WORKDIR /path/to/workdir`。

为后续的 RUN、CMD、ENTRYPOINT 指令配置工作目录。

可以使用多个 WORKDIR 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。例如：

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

则最终路径为 `/a/b/c`。

## 13. ONBUILD

格式为 `ONBUILD [INSTRUCTION]`。

配置当所创建的镜像作为其他新创建镜像的基础镜像时，所执行的操作指令。例如，Dockerfile 使用如下的内容创建了镜像 `image-A`。

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

如果基于 `image-A` 创建新的镜像时，新的 Dockerfile 中使用 `FROM image-A` 指定基础镜像时，会自动执行 `ONBUILD` 指令内容，等价于在后面添加了两条指令。

```
FROM image-A

#Automatically run the following
ADD . /app/src
RUN /usr/local/bin/python-build --dir /app/src
```

使用 `ONBUILD` 指令的镜像，推荐在标签中注明，例如 `ruby:1.9-onbuild`。

### 8.3 创建镜像

编写完成 Dockerfile 之后，可以通过 docker build 命令来创建镜像。

基本的格式为 docker build[ 选项 ] 路径，该命令将读取指定路径下（包括子目录）的 Dockerfile，并将该路径下所有内容发送给 Docker 服务端，由服务端来创建镜像。因此一般建议放置 Dockerfile 的目录为空目录。

另外，可以通过 .dockerignore 文件（每一行添加一条匹配模式）来让 Docker 忽略路径下的目录和文件。

要指定镜像的标签信息，可以通过 -t 选项。

例如，指定 Dockerfile 所在路径为 /tmp/docker\_builder/，并且希望生成镜像标签为 **build\_repo/first\_image**，可以使用下面的命令：

```
$ sudo docker build -t build_repo/first_image /tmp/docker_builder/
```

### 8.4 本章小结

本章主要介绍围绕 Dockerfile 配置文件的重要概念，包括基本结构、所支持的内部指令，以及使用它创建镜像的基本过程。在使用 Dockerfile 配置文件的过程中，读者能体会到 Docker “一点修改代替大量更新”的灵活之处。

当然，如何编写一个高质量的 Dockerfile 并不是一件容易的事情，DockerHub 和 DockerPool 社区都提供了大量的 Dockerfile 范例供大家参考。在本书的第二部分中，笔者也给出了大量热门镜像的 Dockerfile 的介绍和使用方法。



## 第二部分 Part 2

# 实战案例

- 第 9 章 操作系统
- 第 10 章 创建支持 SSH 服务的镜像
- 第 11 章 Web 服务器与应用
- 第 12 章 数据库应用
- 第 13 章 编程语言
- 第 14 章 使用私有仓库
- 第 15 章 构建 Docker 容器集群
- 第 16 章 在公有云上使用 Docker
- 第 17 章 Docker 实践之道

实战是检验技术的唯一标准。

通过第一部分的学习，读者应该掌握了 Docker 的核心概念和常用操作。

在第二部分中，笔者将展示大量的 Docker 应用案例，更加深入地展示 Docker 技术如何在生产环境中进行应用。

第 9 章介绍通过 Docker 来运行典型的操作系统环境，包括 BusyBox、Debian/Ubuntu、CentOS、Fedora，以及基于 Docker 的特色操作系统 CoreOS。

第 10 章介绍如何为一个镜像添加 SSH 服务的支持，以及探讨访问容器内部的合理方案。

第 11 章介绍利用 Docker 来提供典型的 Web 服务，包括 Apache、Nginx、Tomcat、Weblogic、LAMP、CMS 等流行的 Web 解决方案。

第 12 章通过 MySQL、Oracle XE 和 MongoDB 等典型例子，展示在 Docker 中运行常见的 SQL 和 NoSQL 数据库软件。

编程开发也可以从 Docker 中获得方便。第 13 章将介绍流行的编程语言，包括 C/C++、Java、PHP、Python、Perl、Ruby、JavaScript、Go 等，以及如何用 Docker 来快速构建相应的编程环境。

第 14 章具体介绍使用 docker-registry 创建和使用私有仓库的更多技术细节。

第 15 章探讨利用 Docker 创建容器集群要解决的核心问题和可行方案。

接下来，会以国内的阿里云为例，讲解在公有云平台部署 Docker 的过程和一些特色服务。

最后，结合 Docker，对个人学习、技术创业以及企业的生产实践中的常见需求和问题进行探讨。

通过第二部分的实战案例学习，读者可以在实际的工作、生产环境中更加高效地使用 Docker。

# 操作 系统

目前最常用的 Linux 发行版包括 Ubuntu 系列和 CentOS 系列。

前者以自带软件包版本较新而出名；后者则宣称运行更稳定一些。选择哪个操作系统取决于读者的具体需求。同时，社区还推出了完全基于 Docker 的 Linux 发行版 CoreOS。

使用 Docker，读者只需要一个命令就能得到 Linux 发行版的 Docker 镜像，这是以往包括各种虚拟化技术都难以实现的。虽然 Docker 的镜像一般都很精简，但是它几乎可以实现所有的 Linux 服务器系统能实现的功能。

本章将介绍如何使用 Docker 安装和使用 Busybox、Debian/Ubuntu、CentOS/Fedora、CoreOS 等操作系统。

## 9.1 Busybox

BusyBox 是一个集成了一百多个最常用 Linux 命令和工具的软件工具箱，它在单一的可执行文件中提供了精简的 Unix 工具集。BusyBox 可运行于多款 POSIX 环境的操作系统中，如 Linux（包括 Android）、Hurd、FreeBSD 等。

BusyBox 既包含了一些简单实用的工具，如 cat 和 echo，也包含了一些更大、更复杂的工具，如 grep、find、mount 以及 telnet。可以说，BusyBox 是 Linux 系统的瑞士军刀。

### 使用官方镜像

在 DockerHub 中搜索 Busybox 相关的镜像。

```
$ sudo docker search busybox
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
busybox	Busybox base image.	83	[OK]	
programm/busybox		12	[OK]	
lgsd/diamond	Diamond is the smallest lightweight Docker...	6	[OK]	
maxexcloo/data	Docker base image with /data defined as a ...	4	[OK]	
centurylink/ngrok	A Docker image for ngrok, introspected tun...	2	[OK]	
virtuald/nsq	Automated build for NSQ using busybox + of...	1	[OK]	
skomma/busybox-data	Docker image suitable for data volume cont...	1	[OK]	
peer60/newrelic-sysmond	NewRelic monitoring in a docker image. Bas...	1	[OK]	
onsi/grace-busybox		1		
radial/busyboxplus	Full-chain, Internet enabled, busybox made...	1	[OK]	
marcuslinke/busybox		0		
yungsang/busybox	<a href="https://github.com/jpetazzo/docker-busybox">https://github.com/jpetazzo/docker-busybox</a>	0		
abashok/busybox	Me privat busybox	0		
flynny/busybox	Busybox from Ubuntu 13.10 with libc	0		
microcloud/busybox-fs		0	[OK]	
alars/busybox-go-webapp			[OK]	
dlacewell/busybox		0		
opensvc/busybox	Derived from busybox, with an infinitive loo...	0		
jbeda/busybox	A version of busybox with netcat installed.	0		
jpetazzo/busybox		0		
papa99do/busybox		0		
...				

读者可以看到最受欢迎的镜像同时带有 official 标记，说明它是官方镜像。

下面使用 docker pull 命令下载这个镜像：

```
$ sudo docker pull busybox
busybox:latest: The image you are pulling has been verified
e433a6c5b276: Pull complete
e72ac664f4f0: Pull complete
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
Status: Downloaded newer image for busybox:latest
```

如果不指定标签信息，则 Docker 会下载最新版本的 Busybox 镜像，可以看到 Busybox 镜像十分精巧，只有 2.433 MB。

```
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
busybox          latest   e72ac664f4f0   6 weeks ago   2.433 MB
...
```

## 运行 Busybox

启动一个 Busybox 容器，并在容器中执行 grep 命令。

```
$ sudo docker run -it busybox
/ # grep
```

```

BusyBox v1.22.1 (2014-05-22 23:22:11 UTC) multi-call binary.

Usage: grep [-HhnLoqvsvFE] [-m N] [-A/B/C N] PATTERN/-e PATTERN.../-f FILE [FILE]...

Search for PATTERN in FILES (or stdin)

-H      Add 'filename:' prefix
-h      Do not add 'filename:' prefix
-n      Add 'line_no:' prefix
-l      Show only names of files that match
-L      Show only names of files that don't match
-c      Show only count of matching lines
-o      Show only the matching part of line
-q      Quiet. Return 0 if PATTERN is found, 1 otherwise
-v      Select non-matching lines
-s      Suppress open and read errors
-r      Recurse
-i      Ignore case
-w      Match whole words only
-x      Match whole lines only
-F      PATTERN is a literal (not regexp)
-E      PATTERN is an extended regexp
-m N   Match up to N times per file
-A N   Print N lines of trailing context
-B N   Print N lines of leading context
-C N   Same as '-A N -B N'
-e PTRN Pattern to match
-f FILE Read pattern from file

```

查看容器内的挂载信息。

```

/ # mount
rootfs on / type rootfs (rw)
none on / type aufs (rw, relatime, si=b455817946f8505c)
proc on /proc type proc (rw, nosuid, nodev, noexec, relatime)
tmpfs on /dev type tmpfs (rw, nosuid, mode=755)
shm on /dev/shm type tmpfs (rw, nosuid, nodev, noexec, relatime, size=65536k)
devpts on /dev/pts type devpts (rw, nosuid, noexec, relatime, gid=5, mode=620,
ptmxmode=666)
sysfs on /sys type sysfs (ro, nosuid, nodev, noexec, relatime)
/dev/disk/by-uuid/b1f2dba7-d91b-4165-a377-bf1a8bed3f61 on /etc/resolv.conf type
ext4 (rw, relatime, errors=remount-ro, data=ordered)
/dev/disk/by-uuid/b1f2dba7-d91b-4165-a377-bf1a8bed3f61 on /etc/hostname type
ext4 (rw, relatime, errors=remount-ro, data=ordered)
/dev/disk/by-uuid/b1f2dba7-d91b-4165-a377-bf1a8bed3f61 on /etc/hosts type ext4
(rw, relatime, errors=remount-ro, data=ordered)
devpts on /dev/console type devpts (rw, nosuid, noexec, relatime, gid=5,
mode=620, ptmxmode=000)
proc on /proc/sys type proc (ro, nosuid, nodev, noexec, relatime)
proc on /proc/sysrq-trigger type proc (ro, nosuid, nodev, noexec, relatime)

```

```
proc on /proc/irq type proc (ro, nosuid, nodev, noexec, relatime)
proc on /proc/bus type proc (ro, nosuid, nodev, noexec, relatime)
tmpfs on /proc/kcore type tmpfs (rw, nosuid, mode=755)
```

Busybox 镜像虽然小巧，但提供了常见的 Linux 命令，读者可以用它快速熟悉 Linux 命令。

## 9.2 Debian/Ubuntu

Debian 和 Ubuntu 都是目前较为流行的 Debian 系的服务器操作系统，在 Docker Hub 上都可以直接搜索到官方版本。

### 搜索 Debian

```
$ sudo docker search debian
NAME          DESCRIPTION      STARS      OFFICIAL      AUTOMATED
debian        (Semi) Official Debian base image.          199      [OK]
google/debian
tianon/debian  use "debian" instead - https://index.docke... 14      [OK]
eboraas/apache-php PHP5 on Apache (with SSL support), built o... 6      [OK]
toke/owncloud7 Basic image with Owncloud 7 from debian pa... 5      [OK]
yesnault/docker-phabricator Debian Jessie / Apache 2 / Mysql / Phabric... 5 [OK]
hanswesterbeek/google-debian-oracle-jdk Oracle's JDK installed on top of
    Google's ...                                         4      [OK]
eddelbuettel/docker-debian-r                           4      [OK]
maxexcloo/java   Docker framework container with the Oracle... 4      [OK]
eboraas/apache   Apache (with SSL support), built on Debian 4      [OK]
```

### 搜索 Ubuntu

Ubuntu 相关的镜像有很多，因此这里使用 “-s” 参数，只搜索那些被收藏 10 次以上的镜像。

```
$ sudo docker search -s 10 ubuntu
```

```
NAME          DESCRIPTION      STARS      OFFICIAL      AUTOMATED
ubuntu        Official Ubuntu base image                  840      [OK]
dockerfile/ubuntu Trusted automated Ubuntu (http://www.ubunt... 30      [OK]
crashsystems/gitlab-docker A trusted, regularly updated build of GitL... 20      [OK]
sylvainlasnier/memcached This is a Memcached 1.4.14 docker images b... 16      [OK]
ubuntu-upstart Upstart is an event-based replacement for ... 16      [OK]
mbentley/ubuntu-django-uwsgi-nginx                         16      [OK]
ansible/ubuntu14.04-ansible Ubuntu 14.04 LTS with ansible 15      [OK]
clue/ttrss       The Tiny Tiny RSS feed reader allows you t... 14      [OK]
dockerfile/ubuntu-desktop Trusted automated Ubuntu Desktop (LXDE) (h... 14      [OK]
tutum/ubuntu    Ubuntu image with SSH access. For the root... 12      [OK]
```

根据搜索出来的结果，用户可以自行选择下载镜像并使用。

下面就以 Ubuntu 14.04 为例，让我们进去一个 Docker 版的 Ubuntu:14.04 操作系统来体验一番吧！

使用 -ti 参数进入，查看 Ubuntu 的版本号：

```
$ sudo docker run -ti ubuntu:14.04 /bin/bash
root@7d93de07bf76:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 14.04.1 LTS
Release:        14.04
Codename:       trusty
```

当我们试图安装一个 curl 软件的时候，会提示 E: Unable to locate package curl。因为，Docker 镜像为了精简镜像容量，默认删除了这些信息，需要我们使用 apt-get update 命令来更新一次，读者也可以自己编辑 /etc/apt/sources.list 文件，将默认的软件源改为国内的源：

```
root@7d93de07bf76:/# apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package curl
```

更新完之后就可以安装软件了：

```
root@7d93de07bf76:/# apt-get update
Ign http://archive.ubuntu.com trusty InRelease
Ign http://archive.ubuntu.com trusty-updates InRelease
Ign http://archive.ubuntu.com trusty-security InRelease
Ign http://archive.ubuntu.com trusty-proposed InRelease
Get:1 http://archive.ubuntu.com trusty Release.gpg [933 B]
...
root@7d93de07bf76:/# apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
ca-certificates krb5-locales libasn1-8-heimdal libcurl3 libgssapi-krb5-2
libgssapi3-heimdal libhcrypto4-heimdal libheimbase1-heimdal
libheimntlm0-heimdal libhx509-5-heimdal libidn11 libk5crypto3 libkeyutils1
libkrb5-26-heimdal libkrb5-3 libkrb5support0 libldap-2.4-2
libroken18-heimdal librtmp0 libsasl2-2 libsasl2-modules libsasl2-modules-db
libwind0-heimdal openssl
...
root@7d93de07bf76:/# curl
curl: try 'curl --help' or 'curl --manual' for more information
```

接下来，让我们在这个镜像里面再安装一个 Apache 服务：

```
root@7d93de07bf76:/# apt-get install apache2
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  apache2-bin apache2-data libapr1 libaprutil1 libaprutil1-dbd-sqlite3
  libaprutil1-ldap libxml2 sgml-base ssl-cert xml-core
...
...
```

启动这个 Apache 服务，然后使用 curl 来测试本地访问：

```
root@7d93de07bf76:/# service apache2 start
 * Starting web server apache2 AH00558: apache2: Could not reliably determine
the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName'
directive globally to suppress this message
 *
root@7d93de07bf76:/# curl 127.0.0.1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<!--
  Modified from the Debian original for Ubuntu
  Last updated: 2014-03-19
  See: https://launchpad.net/bugs/1288690
-->
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Apache2 Ubuntu Default Page: It works</title>
<style type="text/css" media="screen">
...
...
```

那么，我如何才能从容器以外的设备来访问的 Apache 服务呢？

答案是：如果以 `-ti` 参数启动容器，是无法让外部的设备来访问的，如果要允许外部设备来访问容器的话，就需要使用 `-p` 参数对外映射端口。它通常搭配 `-d-v` 等参数一起使用，具体的 Apache 容器的创建，我们在稍后第 11 章“Web 服务器与应用”有详细介绍。

使用 `-ti` 参数启动的容器，更适合作为测试、学习使用，实际应用中应用的反而较少。在本书的下一章将会介绍如何创建一个带 SSH 服务的容器帮助我们以更加熟悉的方式进入容器。

### 9.3 CentOS/Fedora

CentOS 和 Fedora 都是基于 Redhat 的 Linux 发行版。前者以兼容 Redhat 软件而出名，CentOS 是企业级服务器的常用选型；后者则主要面向个人用户。

## 搜索 CentOS

```
$ sudo docker search -s 2 centos
NAME          DESCRIPTION      STARS      OFFICIAL      AUTOMATED
centos        The official build of CentOS.           542 [OK]
tianon/centos  CentOS 5 and 6, created using rinse instea... 28
ansible/centos7-ansible  Ansible on Centos7            13 [OK]
blalor/centos   Bare-bones base CentOS 6.5 image       7 [OK]
saltstack/centos-6-minimal
steeef/graphite-centos  CentOS 6.x with Graphite and Carbon via ng... 6 [OK]
ariya/centos6-teamcity-server  TeamCity Server 8.1 on CentOS 6       6 [OK]
tutum/centos-6.4    DEPRECATED. Use tutum/centos:6.4 instead. ... 5 [OK]
tutum/centos     Centos image with SSH access. For the root... 5 [OK]
```

对最新的 CentOS7 感兴趣的读者，只需一个命令就可以在 Docker 中来体验 CentOS7 的魅力了。

## 搜索 Fedora

首先改用 docker search 命令来搜索标星至少为 2 的 fedora 相关镜像，结果如下：

```
$ sudo docker search -s 2 fedora
NAME          DESCRIPTION      STARS      OFFICIAL      AUTOMATED
fedora        (Semi) Official Fedora base image.      84 [OK]
mattdm/fedora  A basic Fedora image corresponding roughly... 51
fedora/apache
fedora/couchdb
fedora/mariadb
fedora/redis
fedora/nginx
fedora/python
fedora/qpidd
fedora/rabbitmq
fedora/registry
fedora/memcached
fedora/earthquake
fedora/systemd-systemd
fedora/firefox
fedora/ssh
fedora/nodejs
mattdm/fedora-small  A small Fedora image on which to build. Co... 9
fedora/systemd-apache
helber/fedora-small
tutum/fedora-20   DEPRECATED. Use tutum/fedora:20 or tutum/f... 2 [OK]
```

根据搜索出来的结果，读者可以自行选择下载镜像并使用。

## 9.4 CoreOS

CoreOS 是一个基于 Docker 的 Linux 发行版，官方介绍了若干安装方法，笔者推荐初学者使用 VMware Workstation 来运行 CoreOS。

### 使用官方镜像

第一步，从官方网站下载 CoreOS 镜像，地址为 [http://alpha.release.core-os.net/amd64-usr/current/coreos\\_production\\_vmware\\_insecure.zip](http://alpha.release.core-os.net/amd64-usr/current/coreos_production_vmware_insecure.zip)。如果读者遇到无法下载此镜像包的情况，则可以前往 <http://dockerpool.com> 下载。

如果读者已经安装 VMware Workstation，则解压镜像包后双击 vmx 文件，如图 9-1 所示。

双击 vmx 文件后，即可启动 CoreOS 虚拟机，如图 9-2 所示。

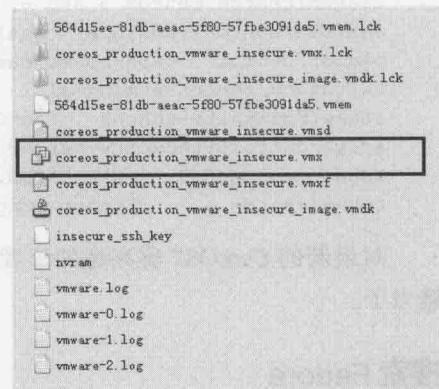


图 9-1 虚拟机文件

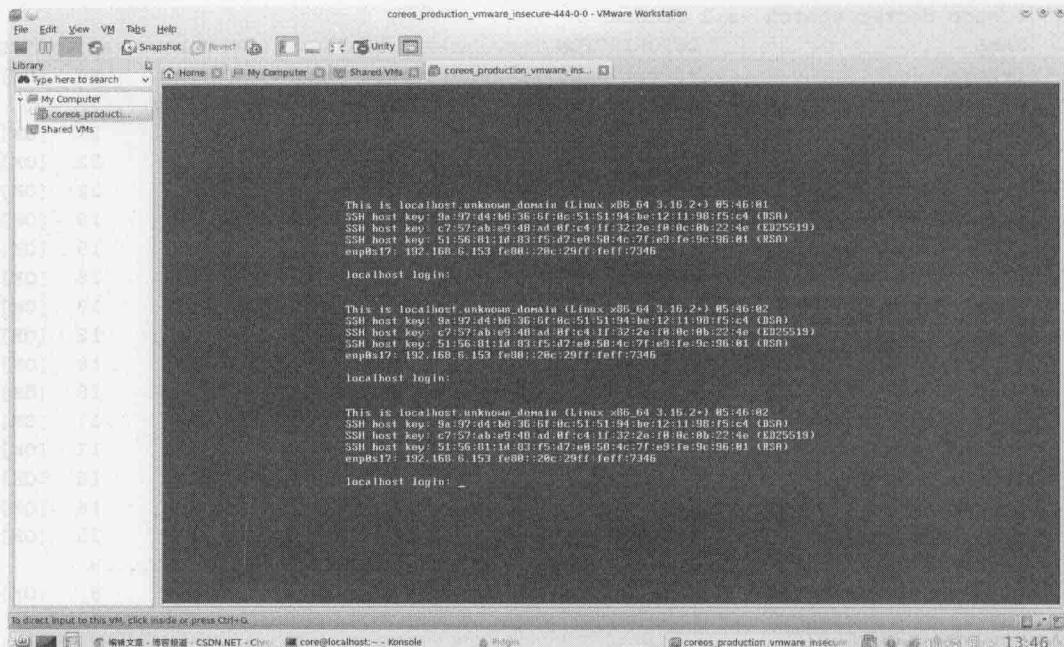


图 9-2 启动 CoreOS 虚拟机

如果读者未安装 VMware Workstation，则可以前往其官网 <http://www.vmware.com/products/workstation/> 下载，如图 9-3 所示。

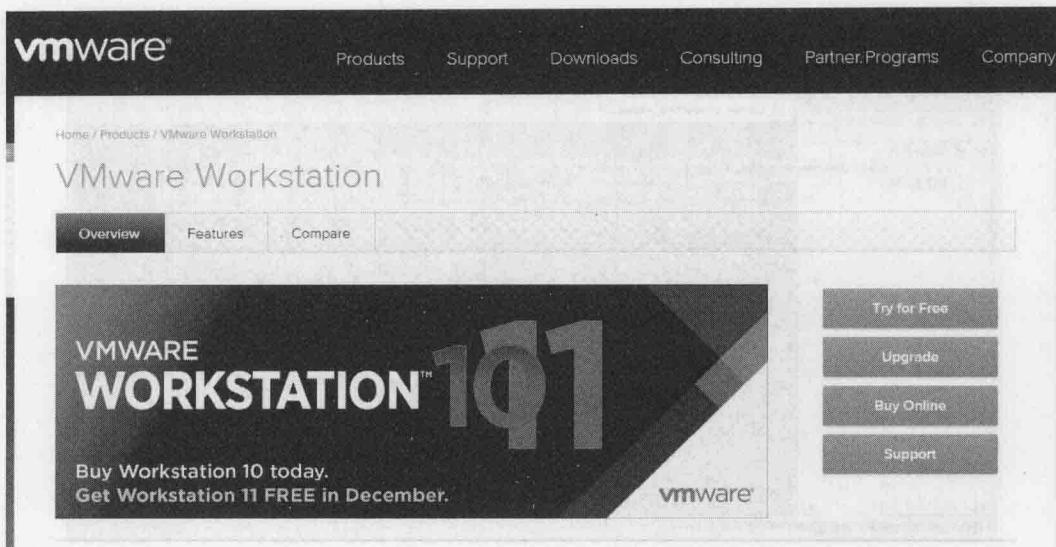


图 9-3 VMware Workstation 下载



经过笔者测试，CoreOS 官方镜像在 VMware 推出的免费版虚拟机 VMware Player 中无法通过 DHCP 自动获取 IP 地址，这会造成读者无法正常使用 SSH 客户端登录 CoreOS 虚拟机。故笔者推荐使用 VMware Workstation 打开 vmx 文件。如超过试用期，请通过正规渠道购买正版授权。

另外，如果在安装 VMware Workstation 过程中出现 HTTPS 端口被占用的情况，可以选择 443 之外的端口，或者禁用占用端口的服务，如图 9-4 所示。

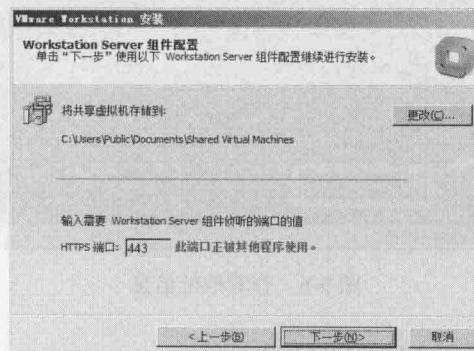


图 9-4 VMware Workstation 安装

第二步，此时 CoreOS 系统已经在 VMware Workstation 中启动，显示登录提示，如图 9-5 所示。

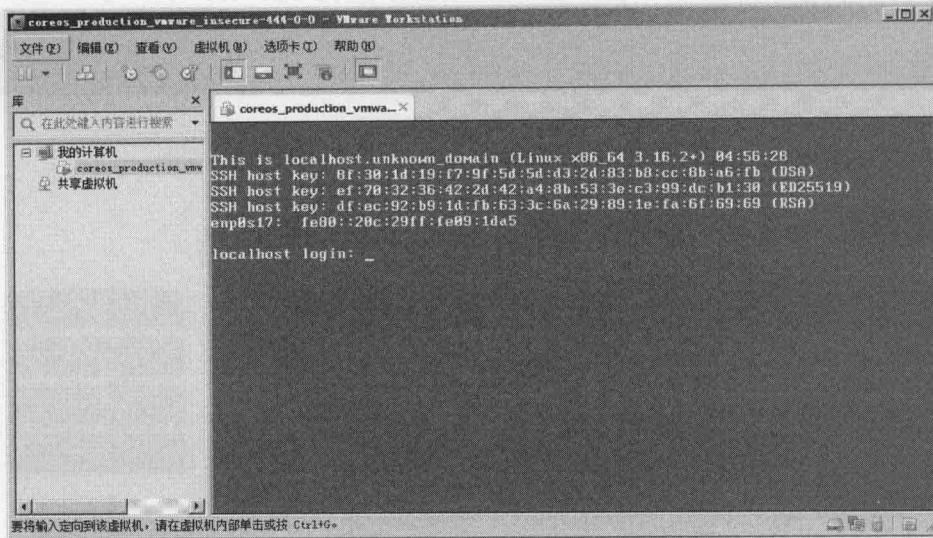


图 9-5 CoreOS 在 VMware Workstation 中启动

直接按回车键，获取当前系统的 IP 地址，如图 9-6 所示。

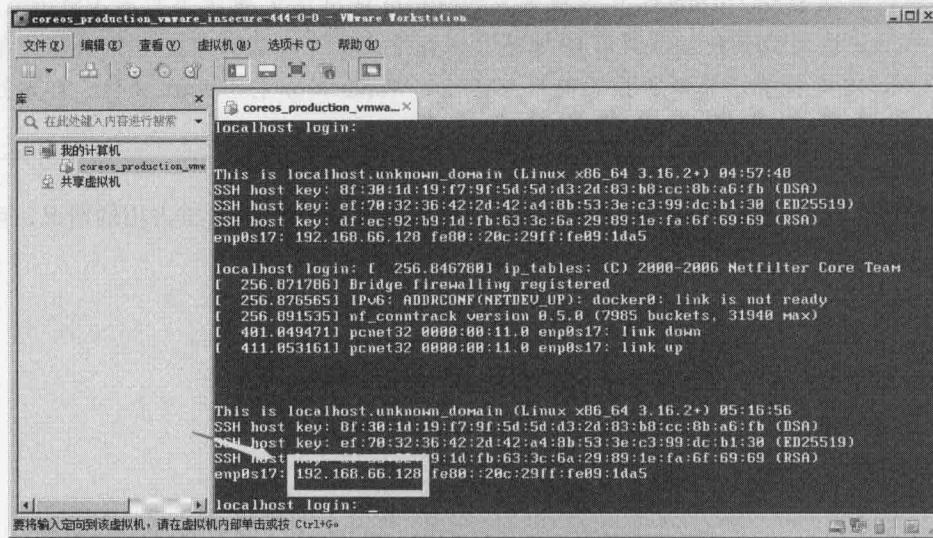


图 9-6 查看地址信息

此时 CoreOS 的 IP 地址是 192.168.66.128。

第三步，使用 SSH 客户端连接此镜像。

笔者以 Windows 环境为例，使用 SecureCRT 工具进行连接。此处读者需要确定如下信息：

- CoreOS 虚拟机的 IP 地址。
  - CoreOS 虚拟机的文件目录下含有 `insecure_ssh_key` 公钥文件。
- 打开 SecureCRT，建立新的 SSH 连接，如图 9-7 所示。

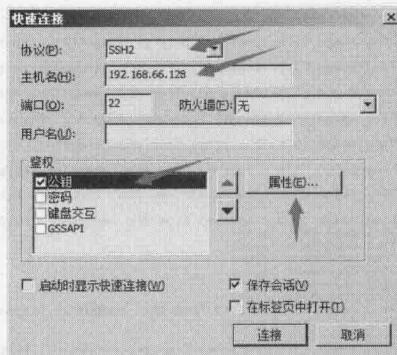


图 9-7 使用 SecureCRT 建立连接

点击“属性”按钮添加 `insecure_ssh_key` 公钥文件后，即可点击“连接”。如果连接成功，则读者可以看到命令行页面，在命令行中可以输入如下命令：

```
$ docker version
```

之后应该可以看到 Docker 的版本信息，如图 9-8 所示。

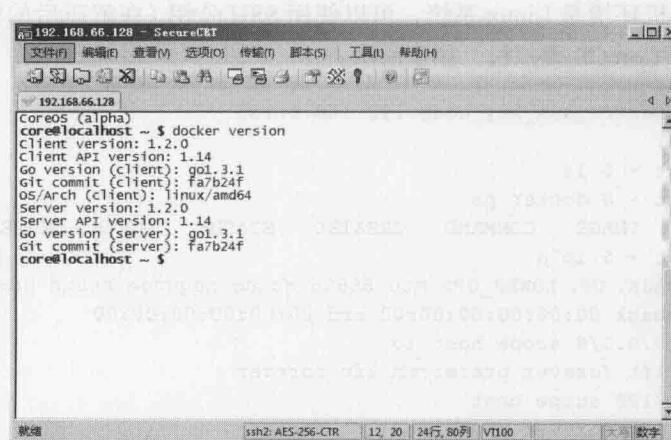


图 9-8 通过 SecureCRT 访问 CoreOS 虚拟机

此时，CoreOS 虚拟机已经成功运行，并且可以使用 SSH 客户端方便地操作 CoreOS 虚拟机。Docker 已经内置于 CoreOS 中，读者可以进行各种 Docker 操作，如图 9-9 所示。

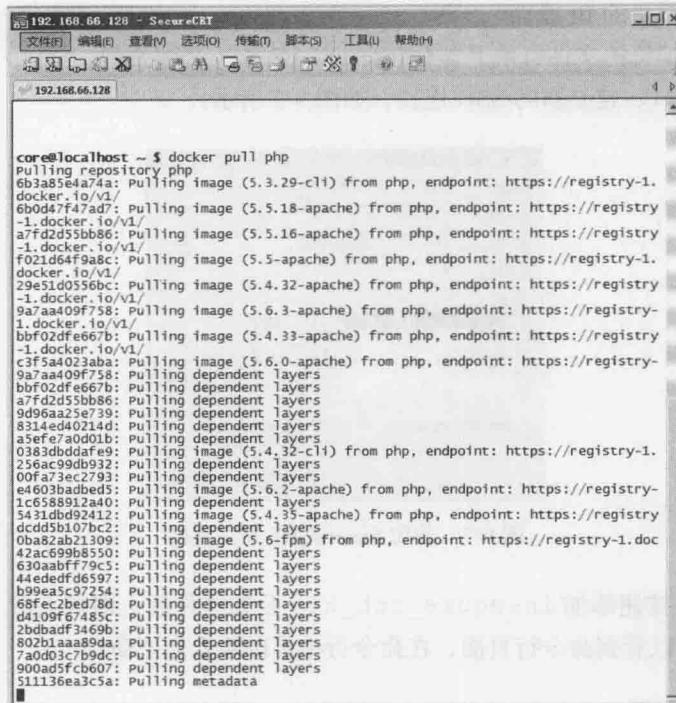


图 9-9 通过 SecureCRT 在 CoreOS 虚拟机中执行操作

如果读者的本机环境是 Linux 系统，可以使用 SSH 公钥（在解压后的根目录下），直接使用 ssh 命令连接 CoreOS 虚拟机。如下所示：

```

$ ssh -i ~/insecure_ssh_key core@192.168.6.153
CoreOS (alpha)
core@localhost ~ $ ls
core@localhost ~ $ docker ps
CONTAINER ID        IMAGE          COMMAND       CREATED      STATUS      PORTS      NAMES
core@localhost ~ $ ip a
1: lo: <LOOPBACK, UP, LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    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: enp0s17: <BROADCAST, MULTICAST, UP, LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0c:29:ff:73:46 brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.153/20 brd 192.168.15.255 scope global dynamic enp0s17
        valid_lft 604500sec preferred_lft 604500sec
    inet6 fe80::20c:29ff:feff:7346/64 scope link
        valid_lft forever preferred_lft forever
  
```

```
3: docker0: <NO-CARRIER, BROADCAST, MULTICAST, UP> mtu 1500 qdisc noqueue state DOWN
   link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
   inet 172.17.42.1/16 scope global docker0
      valid_lft forever preferred_lft forever
core@localhost ~ $
```

读者可见上文中出现 ip a 命令，此命令可以查看 CoreOS 虚拟机的网口信息。

## 9.5 本章小结

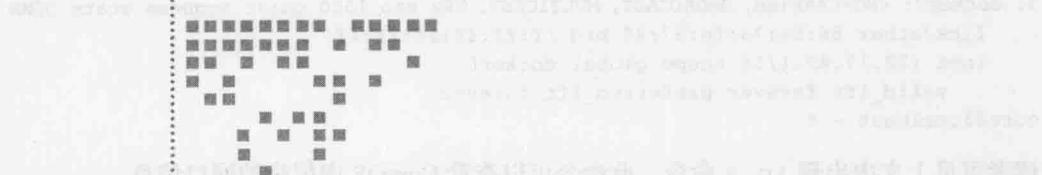
除了官方的镜像外，在 DockerHub 上还有许多第三方组织或个人上传的镜像。一般来说，它们各有特点，读者可以根据具体情况来选择。在选择过程中，有以下几点需要注意：

- 官方的镜像体积都比较小，只安装了一些基本的组件。一个精简的系统有利于安全、稳定和高效地运行，也更加适合用来定制一些服务。
- 个别组织和个人上传的镜像质量也非常高。他们通常针对某个具体应用做了最精准的定位，比如：当需要下载一个包含 LAMP 组件的 Ubuntu 镜像时，DockerHub 上可能已经有集成好的镜像了，比如 tutum、dockerpool 等。

另外，想查看下载镜像的详细信息，可以通过 docker inspect + 镜像 ID 的方法来获取更多信息。

最后，关于镜像登录的用户名和密码。出于安全考虑，几乎所有带官方标志的操作系统都无法直接使用用户名和密码直接登录，而且一般都没有安装 SSH 服务。

下一章，笔者将带领读者一起创建一个带 SSH 服务的基础镜像。



## 第 10 章

## 创建支持 SSH 服务的镜像

查看本章

第 10 章将向读者介绍如何通过 Docker 容器来管理 Linux 系统。在前几章中，我们已经学习了如何使用 Docker 容器来运行各种应用。本章将介绍如何在容器内安装和配置 SSH 服务，从而实现对容器的远程管理。通过本章的学习，读者将能够掌握如何在 Docker 容器内安装 SSH 服务，并通过 SSH 连接来管理容器内的应用。

一般情况下，Linux 系统管理员通过 SSH 服务来管理操作系统，但 Docker 的很多镜像是不带 SSH 服务的，那么我们怎样才能管理操作系统呢？

在第一部分中我们介绍了一些进入容器的办法，比如用 `attach`、`exec` 等命令，但是这些命令都无法解决远程管理容器的问题。因此，当读者需要远程登录到容器内进行一些操作的时候，就需要 SSH 的支持了。

本章将具体介绍如何自行创建一个带有 SSH 服务的镜像，并详细介绍两种创建容器的方法：基于 `docker commit` 命令创建和基于 `Dockerfile` 创建。

## 10.1 基于 `commit` 命令创建

Docker 提供了 `docker commit` 命令，支持用户提交自己对容器的修改，并生成新的镜像。命令格式为 `docker commit CONTAINER [REPOSITORY[:TAG]]`。

这里将介绍如何使用 `docker commit` 命令，为 `ubuntu:14.04` 镜像添加 SSH 服务。

### 准备工作

首先，使用 `ubuntu:14.04` 镜像来创建一个容器：

```
$ sudo docker run -it ubuntu:14.04 /bin/bash
```

尝试使用 `SSHD` 命令，读者会发现容器中并没有安装该服务：

```
root@fc1936ea8ceb:/# sshd
bash: sshd: command not found
```

同时，笔者从 **apt** 包管理器的软件源信息中亦找不到启动 SSH 服务需要的 **openssh-server**，这是因为 Ubuntu 官方镜像中并没有包含软件包的缓存文件：

```
root@fc1936ea8ceb:/# apt-get install openssh-server
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package openssh-server
```

下面，笔者将演示如何更新软件包缓存，并安装 SSHD 服务。

## 配置软件源

检查软件源，并使用 **apt-get update** 来更新软件源信息：

```
root@fc1936ea8ceb:/# apt-get update

Ign http://archive.ubuntu.com trusty InRelease
Ign http://archive.ubuntu.com trusty-updates InRelease
Ign http://archive.ubuntu.com trusty-security InRelease
Ign http://archive.ubuntu.com trusty-proposed InRelease
...
Fetched 20.4 MB in 2min 55s (116 kB/s)
Reading package lists... Done
```

如果默认的官方源速度慢的话，也可以替换为国内 163、Sohu 等镜像的源。以 163 源为例，在容器内创建 **/etc/apt/sources.list.d/163.list** 文件：

```
root@fc1936ea8ceb:/# vi /etc/apt/sources.list.d/163.list
```

添加如下内容到文件中：

```
deb http://mirrors.163.com/ubuntu/ trusty main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-security main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-updates main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-proposed main restricted universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-backports main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-security main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-updates main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-proposed main restricted universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-backports main restricted universe multiverse
```

之后重新执行 **apt-get update** 命令即可。

## 安装和配置 SSH 服务

更新软件包缓存后，已经可以安装 SSH 服务了，选择主流的 `openssh-server` 作为服务端。可以看到需要下载安装众多的依赖软件包：

```
root@fc1936ea8ceb:/# apt-get install openssh-server
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  ca-certificates krb5-locales libck-connector0 libedit2 libgssapi-krb5-2
  libidn11 libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib libwrap0 libx11-6
  libx11-data libxauf libxcb1 libxdmcp6 libxext6 libxmuul ncurses-term
  openssh-client openssh-sftp-server openssl python python-chardet
  python-minimal python-requests python-six python-urllib3 python2.7
  python2.7-minimal ssh-import-id tcpd wget xauth
Suggested packages:
  krb5-doc krb5-user ssh-askpass libpam-ssh keychain monkeysphere rssh
  molly-guard ufw python-doc python-tk python2.7-doc binutils binfmt-support
The following NEW packages will be installed:
  ca-certificates krb5-locales libck-connector0 libedit2 libgssapi-krb5-2
  libidn11 libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib libwrap0 libx11-6
  libx11-data libxauf libxcb1 libxdmcp6 libxext6 libxmuul ncurses-term
  openssh-client openssh-server openssh-sftp-server openssl python
  python-chardet python-minimal python-requests python-six python-urllib3
  python2.7 python2.7-minimal ssh-import-id tcpd wget xauth
0 upgraded, 38 newly installed, 0 to remove and 29 not upgraded.
Need to get 7599 kB of archives.
After this operation, 35.3 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://archive.ubuntu.com/ubuntu/ trusty/main libedit2 amd64 3.1-20130712-2 [86.7 kB]
Get:2 http://archive.ubuntu.com/ubuntu/ trusty-proposed/main libkrb5support0 amd64 1.12+dfsg-2ubuntu5 [30.0 kB]
Get:3 http://archive.ubuntu.com/ubuntu/ trusty-proposed/main libk5crypto3 amd64 1.12+dfsg-2ubuntu5 [79.9 kB]
Get:4 http://archive.ubuntu.com/ubuntu/ trusty/main libkeyutils1 amd64 1.5.6-1 [7318 B]
Get:5 http://archive.ubuntu.com/ubuntu/
...
Updating certificates in /etc/ssl/certs... 164 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....done.
Processing triggers for ureadahead (0.100.0-16) ...
```

要正常启动 SSH 服务，需要目录 `/var/run/sshd` 存在，手动创建它，并启动服务：

```
root@fc1936ea8ceb:/# mkdir -p /var/run/sshd
root@fc1936ea8ceb:/# /usr/sbin/sshd -D &
[1] 3254
```

此时查看容器的 22 端口 (SSH 服务默认监听的端口), 已经处于监听状态:

```
root@fc1936ea8ceb:/# netstat -tunlp
Active Internet connections (only servers)
Proto Recv-Q Local Address      Foreign Address      State      PID/Program name
tcp        0      0.0.0.0:22          0.0.0.0:*      LISTEN
tcp        0      0 :::22            ::::*           LISTEN
```

修改 SSH 服务的安全登录配置, 取消 pam 登录限制:

```
root@fc1936ea8ceb:/# sed -ri 's/session required pam_loginuid.so/#session required pam_loginuid.so/g' /etc/pam.d/sshd
```

在 root 用户目录下创建 .ssh 目录, 并复制需要登录的公钥信息 (一般为本地主机用户目录下的 .ssh/id\_rsa.pub 文件, 可由 ssh-keygen -t rsa 命令生成) 到 authorized\_keys 文件中。

```
root@fc1936ea8ceb:/# mkdir root/.ssh
root@fc1936ea8ceb:/# vi /root/.ssh/authorized_keys
```

创建自动启动 SSH 服务的可执行文件 run.sh, 并添加可执行权限:

```
root@fc1936ea8ceb:/# vi /run.sh
root@fc1936ea8ceb:/# chmod +x run.sh
```

run.sh 脚本内容如下:

```
#!/bin/bash
/usr/sbin/sshd -D
```

最后, 退出容器:

```
root@fc1936ea8ceb:/# exit
exit
```

## 保存镜像

将所退出的容器用 docker commit 命令保存为一个新的 sshd:ubuntu 镜像:

```
$ sudo docker commit fc1 sshd:ubuntu
7aef2cd95fd0c712f022bcff6a4ddefccf20fd693da2b24b04ee1cd3ed3eb6fc
```

使用 docker images 查看本地生成的新镜像 sshd:ubuntu, 目前拥有的镜像如下:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
sshd	ubuntu	7aef2cd95fd0	10 seconds ago	255.2 MB
busybox	latest	e72ac664f4f0	3 weeks ago	2.433 MB
ubuntu	latest	ba5877dc9bec	3 months ago	192.7 MB

## 使用镜像

启动容器，并添加端口映射 10022 → 22。其中 10022 是宿主主机的端口，22 是容器的 SSH 服务监听端口：

```
$ sudo docker run -p 10022:22 -d sshd:ubuntu /run.sh
3ad7182aa47f9ce670d933f943fdec946ab69742393ab2116bace72db82b4895
```

启动成功后，可以在宿主主机上看到容器运行的详细信息：

```
$ sudo docker ps
CONTAINER ID        IMAGE          COMMAND       CREATED          STATUS          PORTS          NAMES
3ad7182aa47f        sshd:ubuntu   "/run.sh"    2 seconds ago   Up 2 seconds
0.0.0.0:10022->22/tcp      focused_ptolemy
```

在宿主主机（192.168.1.200）或其他主机上，可以通过 SSH 访问 10022 端口来登录容器：

```
$ ssh 192.168.1.200 -p 10022
The authenticity of host '[192.168.1.200]:10022 ([192.168.1.200]:10022)' can't
be established.
ECDSA key fingerprint is 5f:6e:4c:54:8f:c7:7f:32:c2:38:45:bb:16:03:c9:e8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.1.200]:10022' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.2.0-37-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@3ad7182aa47f:~#
```

## 10.2 使用 Dockerfile 创建

在第一部分中笔者曾介绍过 Dockerfile 的基础知识，下面将介绍如何使用 Dockerfile 来创建一个支持 SSH 服务的镜像。

### 1. 创建工作目录

首先应创建一个 `sshd_ubuntu` 工作目录：

```
$ mkdir sshd_ubuntu
$ ls
sshd_ubuntu
```

在其中，创建 **Dockerfile** 和 **run.sh** 文件：

```
$ cd sshd_ubuntu/
$ touch Dockerfile run.sh
$ ls
Dockerfile  run.sh
```

## 2. 编写 run.sh 脚本和 authorized\_keys 文件

脚本文件 **run.sh** 的内容与上一小节中一致：

```
#!/bin/bash
/usr/sbin/sshd -D
```

在宿主主机上生成 SSH 密钥对，并创建 **authorized\_keys** 文件：

```
$ ssh-keygen -t rsa
...
$ cat ~/.ssh/id_rsa.pub >authorized_keys
```

## 3. 编写 Dockerfile

下面是 **Dockerfile** 的内容及各部分的注释，可以发现，对比上一节中利用 **docker commit** 命令创建镜像过程，所进行的操作基本一致。

```
# 设置继承镜像
FROM ubuntu:14.04

# 提供一些作者的信息
MAINTAINER from www.dockerpool.com by waitfish (dwj_zz@163.com)

# 下面开始运行命令，此处更改 ubuntu 的源为国内 163 的源
RUN echo "deb http://mirrors.163.com/ubuntu/ trusty main restricted universe
multiverse" > /etc/apt/sources.list
RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-security main restricted
universe multiverse" >> /etc/apt/sources.list
RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-updates main restricted
universe multiverse" >> /etc/apt/sources.list
RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-proposed main restricted
universe multiverse" >> /etc/apt/sources.list
RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-backports main restricted
universe multiverse" >> /etc/apt/sources.list
RUN apt-get update

# 安装 ssh 服务
RUN apt-get install -y openssh-server
RUN mkdir -p /var/run/sshd
RUN mkdir -p /root/.ssh
```

```

# 取消 pam 限制
RUN sed -ri 's/session required pam_loginuid.so/#session required
pam_loginuid.so/g' /etc/pam.d/sshd

# 复制配置文件到相应位置，并赋予脚本可执行权限
ADD authorized_keys /root/.ssh/authorized_keys
ADD run.sh /run.sh
RUN chmod 755 /run.sh

# 开放端口
EXPOSE 22

# 设置自启动命令
CMD ["/run.sh"]

```

#### 4. 创建镜像

在 sshd\_ubuntu 目录下，使用 docker build 命令来创建镜像。注意一下，在最后还有一个“.”，表示使用当前目录中的 Dockerfile。

```
$ cd sshd_ubuntu
$ sudo docker build -t sshd:dockerfile .
```

如果读者使用 Dockerfile 创建自定义镜像，那么需要注意的是 Docker 会自动删除中间临时创建的层，还需要注意每一步的操作和编写的 Dockerfile 中命令的对应关系。

执行 docker build 命令的输出参考结果如下：

```

Sending build context to Docker daemon 5.632 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
--> ba5877dc9bec
Step 1 : MAINTAINER dwj_zz@163.com
--> Running in 188d74d02d35
--> 473eb019b331
Removing intermediate container 188d74d02d35
Step 2 : RUN echo "deb http://mirrors.163.com/ubuntu/ trusty main restricted
universe multiverse" > /etc/apt/sources.list
--> Running in f52e2a583db5
--> bd4ceef2ee19
Removing intermediate container f52e2a583db5
Step 3 : RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-security main
restricted universe multiverse" >> /etc/apt/sources.list
--> Running in 897d65dfe9be
--> 9cd736f11928
Removing intermediate container 897d65dfe9be
Step 4 : RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-updates main
restricted universe multiverse" >> /etc/apt/sources.list
--> Running in ec3433db813e
--> 3fcfa0b605de4

```

```

Removing intermediate container ec3433db813e
Step 5 : RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-proposed main
restricted universe multiverse" >> /etc/apt/sources.list
    --> Running in 4a0fe165598f
    --> f6d1c7af36c8
Removing intermediate container 4a0fe165598f
Step 6 : RUN echo "deb http://mirrors.163.com/ubuntu/ trusty-backports main
restricted universe multiverse" >> /etc/apt/sources.list
    --> Running in 209179c21053
    --> 0cda758c9f3c
Removing intermediate container 209179c21053
Step 7 : RUN apt-get update
    --> Running in 1fd40eb66f7b
Ign http://archive.ubuntu.com trusty-proposed InRelease
Get:1 http://archive.ubuntu.com trusty-proposed Release.gpg [933 B]
Get:2 http://archive.ubuntu.com trusty-proposed Release [110 kB]
Get:3 http://archive.ubuntu.com trusty-proposed/main amd64 Packages [160 kB]
...
Fetched 11.3 MB in 1min 37s (116 kB/s)
Reading package lists...
--> 0f132591eddc
Removing intermediate container 1fd40eb66f7b
Step 8 : RUN apt-get install -y openssh-server
    --> Running in 399e4ea726d2
Reading package lists...
Building dependency tree...
Reading state information...
The following extra packages will be installed:
  ca-certificates krb5-locales libck-connector0 libedit2 libgssapi-krb5-2
  libidn11 libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib libwrap0 libx11-6
  libx11-data libxau6 libxcb1 libxdmcp6 libxext6 libxmuu1 ncurses-term
  openssh-client openssh-sftp-server openssl python python-chardet
  python-minimal python-requests python-six python-urllib3 python2.7
  python2.7-minimal ssh-import-id tcpd wget xauth
Suggested packages:
  krb5-doc krb5-user ssh-askpass libpam-ssh keychain monkeysphere rssh
  molly-guard ufw python-doc python-tk python2.7-doc binutils binfmt-support
The following NEW packages will be installed:
  ca-certificates krb5-locales libck-connector0 libedit2 libgssapi-krb5-2
  libidn11 libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib libwrap0 libx11-6
  libx11-data libxau6 libxcb1 libxdmcp6 libxext6 libxmuu1 ncurses-term
  openssh-client openssh-server openssl python
  python-chardet python-minimal python-requests python-six python-urllib3
  python2.7 python2.7-minimal ssh-import-id tcpd wget xauth
0 upgraded, 38 newly installed, 0 to remove and 29 not upgraded.
Need to get 7599 kB of archives.
After this operation, 35.3 MB of additional disk space will be used.
Get:1 http://mirrors.163.com/ubuntu/ trusty/main libedit2 amd64 3.1-20130712-2 [86.7 kB]
...
Running hooks in /etc/ca-certificates/update.d....done.
Processing triggers for ureadahead (0.100.0-16) ...
--> 62f952643e33

```

```

Removing intermediate container 399e4ea726d2
Step 9 : RUN mkdir -p /var/run/sshd
--> Running in aa1c4d469284
--> e81557dd4887
Removing intermediate container aa1c4d469284
Step 10 : RUN mkdir -p /root/.ssh
--> Running in 0626987081d0
--> 23882ee06756
Removing intermediate container 0626987081d0
Step 11 : RUN sed -ri 's/session required pam_loginuid.so/#session
required pam_loginuid.so/g' /etc/pam.d/sshd
--> Running in 3808c650bf85
--> 6c0cald20d7f
Removing intermediate container 3808c650bf85
Step 12 : ADD authorized_keys /root/.ssh/authorized_keys
--> a64bbd8ae617
Removing intermediate container 16e2d93d6ef0
Step 13 : ADD run.sh /run.sh
--> 230711022f7d
Removing intermediate container 137e56188d7b
Step 14 : RUN chmod 755 /run.sh
--> Running in a876e4ea378e
--> 32d74bbb7406
Removing intermediate container a876e4ea378e
Step 15 : EXPOSE 22
--> Running in eeaf9352ca11
--> 901e3fa9f596
Removing intermediate container eeaf9352ca11
Step 16 : CMD /run.sh
--> Running in 48c37db83ffb
--> 570c26a9de68
Removing intermediate container 48c37db83ffb
Successfully built 570c26a9de68

```

命令执行完毕后，如果可见“Successfully built XXX”字样，则说明镜像创建成功。可以看到，以上命令生成的镜像 ID 是 570c26a9de68。

在本地查看 **sshd:dockerfile** 镜像已存在：

```

$ sudo docker images
REPOSITORY      TAG          IMAGE ID      CREATED        VIRTUAL SIZE
sshd            dockerfile    570c26a9de68  4 minutes ago  246.5 MB
sshd            ubuntu        7aef2cd95fd0  12 hours ago   255.2 MB
busybox         latest       e72ac664f4f0  3 weeks ago   2.433 MB
ubuntu          14.04       ba5877dc9bec  3 months ago   192.7 MB
ubuntu          latest       ba5877dc9bec  3 months ago   192.7 MB

```

## 5. 测试镜像，运行容器

使用刚才创建的 **sshd:dockerfile** 镜像来运行一个容器。直接启动镜像，映射容器的 22 端口到本地的 10122 端口：

```
$ sudo docker run -d -p 10122:22 sshd:dockerfile
890c04ff8d769b604386ba4475253ae8c21fc92d60083759afa77573bf4e8af1
$ sudo docker ps
CONTAINER ID        IMAGE          COMMAND       CREATED          STATUS          PORTS          NAMES
890c04ff8d76        sshd:dockerfile    "/run.sh"     4 seconds ago   Up 3 seconds   0.0.0.0:10122->22/tcp   high_albattani
```

在宿主主机新打开一个终端，连接到新建的容器：

```
$ ssh 192.168.1.200 -p 10122
The authenticity of host '[192.168.1.200]:10122 ([192.168.1.200]:10122)' can't
be established.
ECDSA key fingerprint is d1:59:f1:09:3b:09:79:6d:19:16:f4:fd:39:1b:be:27.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.1.200]:10122' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.2.0-37-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
```

```
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.
```

```
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

```
root@890c04ff8d76:~#
```

效果与上一小节一致，镜像创建成功。

## 10.3 本章小结

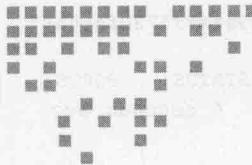
在 Docker 社区中，对于是否需要为 Docker 容器启用 SSH 服务一直有争论。

一方的观点是：Docker 的理念是一个容器只运行一个服务。因此，如果每个容器都运行一个额外的 SSH 服务，就违背了这个理念。另外认为根本没有从远程主机进入容器进行维护的必要。

另一方的观点是：在 Docker 1.3 版本之前，如果要用 attach 进入容器，经常容易出现卡死的情况。1.3 之后，虽然官方推出了 docker exec 命令，在从宿主主机进入是没有障碍了，但是如果要从其他远程主机进入容器依然没有更好的解决方案。

笔者认为，通过一些目前看来较为复杂的方式，确实能够不需要进入容器进行维护，但是使用 SSH 进行服务的维护，是目前 Linux 用户较为熟悉的方式。

因此，在 Docker 推出更加高效、安全的方式对容器进行维护之前，容器的 SSH 服务还是比较重要的，而且它对资源的需求不高，比较适合于生产环境。



Chapter 11

## 第 11 章

## Web 服务器与应用

Web 服务（Web Service）和 Web 应用（Web App）是目前互联网领域的热门技术。

本章将重点介绍如何使用 Docker 来运行常见的 Web 服务器（包括 Apache、Nginx、Tomcat、Weblogic），以及一些常用应用（包括 LAMP 和 CMS）。包括 Docker 镜像的构建方法与使用。

本章会继续展示使用前一章介绍的两种方法（通过 `docker commit` 命令，以及通过 `Dockerfile`）来创建镜像的过程。其中一些操作比较简单的镜像使用 `Dockerfile` 来创建，而像 Weblogic 这样复杂的应用，则使用 `commit` 方式来创建，读者也可以根据自己的需求选择其他方式。

通过本章的介绍，用户将可以根据自己的需求轻松定制 Web 服务或 Web 应用镜像。

## 11.1 Apache

Apache 是目前世界使用排名第一的 Web 服务器软件。由于其良好的跨平台和安全性，Apache 被广泛应用在多种平台和操作系统上。Apache（阿帕奇）的名字源自美国的西南部一个印第安人部落：阿帕奇族。

这里将展示笔者使用 `Dockerfile` 来创建带 Apache 服务的 Docker 镜像的具体过程。

### 准备工作

首先，创建一个 `apache_ubuntu` 工作目录，在其中创建 `Dockerfile` 文件、`run.sh` 文件和 `sample` 目录。

```
$ mkdir apache_ubuntu && cd apache_ubuntu
$ touch Dockerfile run.sh
$ mkdir sample
```

下面是 Dockerfile 的内容和各个部分的说明：

```
FROM sshd:dockerfile
# 设置继承自我们创建的 sshd 镜像

MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)
# 创建者的基本信息

# 设置环境变量，所有操作都是非交互式的
ENV DEBIAN_FRONTEND noninteractive

# 安装
RUN apt-get -yq install apache2&& \
    rm -rf /var/lib/apt/lists/*

RUN echo "Asia/Shanghai" > /etc/timezone && \
    dpkg-reconfigure -f noninteractive tzdata
# 注意这里要更改系统的时区设置，因为在 Web 应用中经常会用到时区这个系统变量，默认的 ubuntu 会让你的应用程序发生不可思议的效果哦

# 添加我们的脚本，并设置权限，这会覆盖之前放在这个位置的脚本
ADD run.sh /run.sh
RUN chmod 755 /*.sh

# 添加一个示例的 Web 站点，删掉默认安装在 apache 文件夹下面的文件，并将我们添加的示例用软链接链接到 /var/www/html 目录下面
RUN mkdir -p /var/lock/apache2 &&mkdir -p /app && rm -fr /var/www/html && ln -s \
/app /var/www/html
COPY sample/ /app

# 设置 apache 相关的一些变量，在容器启动的时候可以使用 -e 参数替代
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_SERVERADMIN admin@localhost
ENV APACHE_SERVERNAME localhost
ENV APACHE_SERVERALIAS docker.localhost
ENV APACHE_DOCUMENTROOT /var/www

EXPOSE 80
WORKDIR /app
CMD ["/run.sh"]
```

这个 sample 站点的内容很简单，就输出一句话 Hello Docker!。在 sample 目录

下创建 index.html 文件，内容为：

```
<!DOCTYPE html>
<html>
<body>
<p>Hello, Docker!</p>
</body>
</html>
```

run.sh 脚本内容也很简单，只是启动 Apache 服务：

```
$ cat run.sh
#!/bin/bash
exec apache2 -D FOREGROUND
```

此时，apache\_ubuntu 目录下面的文件结构为：

```
$ tree .
.
|-- Dockerfile
|-- run.sh
`-- sample
    '-- index.html

1 directory, 3 files
```

## 创建 apache:ubuntu 镜像

使用 docker build 命令创建 apache:ubuntu 镜像，注意命令最后的“.”：

```
$ sudo docker build -t apache:ubuntu .
Sending build context to Docker daemon 6.144 kB
Sending build context to Docker daemon
Step 0 : FROM sshd:dockerfile
--> 570c26a9de68
Step 1 : MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)
--> Using cache
--> 5c6b90057a1d
Step 2 : ENV DEBIAN_FRONTEND noninteractive
--> Using cache
--> e06feb0790d7
Step 3 : RUN apt-get -yq install apache2&& rm -rf /var/lib/apt/lists/*
--> Using cache
--> 54e5665500b9
Step 4 : RUN echo "Asia/Shanghai" > /etc/timezone && dpkg-reconfigure -f
noninteractive tzdata
--> Running in 8c1ad26742bc

Current default time zone: 'Asia/Shanghai'
```

```
Local time is now:      Mon Oct 27 20:19:19 CST 2014.
Universal Time is now:  Mon Oct 27 12:19:19 UTC 2014.

--> 04d64839c7b3
Removing intermediate container 8c1ad26742bc
Step 5 : ADD run.sh /run.sh
--> f995bd0d6f89
Removing intermediate container f372648d02d9
Step 6 : RUN chmod 755 /*.sh
--> Running in ae60847251c8
--> 2e0a58be0f9c
Removing intermediate container ae60847251c8
Step 7 : RUN mkdir -p /var/lock/apache2 &&mkdir -p /app && rm -fr /var/www/html
&& ln -s /app /var/www/html
--> Running in d5355f5e2992
--> 6e96f581ee0d
Removing intermediate container d5355f5e2992
Step 8 : COPY sample/ /app
--> c048a78c17fd
Removing intermediate container 2461604081e3
Step 9 : ENV APACHE_RUN_USER www-data
--> Running in c9c6c12d1982
--> 5cd772b48df2
Removing intermediate container c9c6c12d1982
Step 10 : ENV APACHE_RUN_GROUP www-data
--> Running in b179ff1391c8
--> 47d9b17eaec3
Removing intermediate container b179ff1391c8
Step 11 : ENV APACHE_LOG_DIR /var/log/apache2
--> Running in a3daf191e6b2
--> 8d56f68dbfc6
Removing intermediate container a3daf191e6b2
Step 12 : ENV APACHE_PID_FILE /var/run/apache2.pid
--> Running in 92401ab1dbe
--> 27f2ddba296e
Removing intermediate container 92401ab1dbe
Step 13 : ENV APACHE_RUN_DIR /var/run/apache2
--> Running in 76c6e7401eca
--> 6d0e7acaf398
Removing intermediate container 76c6e7401eca
Step 14 : ENV APACHE_LOCK_DIR /var/lock/apache2
--> Running in a9c46477ebe9
--> 1f352c6d2635
Removing intermediate container a9c46477ebe9
Step 15 : ENV APACHE_SERVERADMIN admin@localhost
--> Running in 3d0d8506856c
--> 679d9964bb34
Removing intermediate container 3d0d8506856c
Step 16 : ENV APACHE_SERVERNAME localhost
--> Running in 405884f45d23
--> 88f25f8e2609
```

```

Removing intermediate container 405884f45d23
Step 17 : ENV APACHE_SERVERALIAS docker.localhost
    --> Running in 0528963fade4
    --> 693cc00c91fe
Removing intermediate container 0528963fade4
Step 18 : ENV APACHE_DOCUMENTROOT /var/www
    --> Running in 2c38a2ca6d14
    --> d8649b79e085
Removing intermediate container 2c38a2ca6d14
Step 19 : EXPOSE 80
    --> Running in cab9677d0756
    --> 682d055d6f15
Removing intermediate container cab9677d0756
Step 20 : WORKDIR /app
    --> Running in 641c59ded9da
    --> 5b08d6572b75
Removing intermediate container 641c59ded9da
Step 21 : CMD /run.sh
    --> Running in ee18331b3e69
    --> 1d865e3032d7
Removing intermediate container ee18331b3e69
Successfully built 1d865e3032d7

```

此时镜像已经创建成功了。下面，查看本地已有的镜像列表，读者可见新增的 apache: ubuntu 镜像：

```

$ sudo docker images
REPOSITORY      TAG          IMAGE ID       CREATED        VIRTUAL SIZE
apache          ubuntu        1d865e3032d7  46 seconds ago  263.8 MB
sshd            dockerfile   570c26a9de68  9 hours ago   246.5 MB
sshd            ubuntu        7aef2cd95fd0  21 hours ago  255.2 MB
debian          latest       61f7f4f722fb  6 days ago   85.1 MB
busybox         latest       e72ac664f4f0  3 weeks ago  2.433 MB
ubuntu          14.04       ba5877dc9bec  3 months ago  192.7 MB
ubuntu          latest       ba5877dc9bec  3 months ago  192.7 MB

```

## 测试镜像

运行镜像，并使用 -P 参数映射需要开放的端口（22 和 80 端口）：

```

$ sudo docker run -d -P apache:ubuntu
64681e2ae943f18eae9f599dbc43b5f44d9090bdca3d8af641d7b371c124acfd
$ sudo docker ps -a
CONTAINER ID   IMAGE      COMMAND     CREATED      STATUS      PORTS      NAMES
64681e2ae943   apache:ubuntu  "/run.sh"   2 seconds ago  Up 1 seconds
0.0.0.0:49171->22/tcp, 0.0.0.0:49172->80/tcp   naughty_poincare
890c04ff8d76   sshd:dockerfile  "/run.sh"   9 hours ago   Exited (0)
3 hours ago   0.0.0.0:101->22/tcp   high_albattani
3ad7182aa47f   sshd:ubuntu    "/run.sh"   21 hours ago  Exited (0) 3
hours ago   0.0.0.0:100->22/tcp   focused_ptolemy

```

在本地主机上用 curl 抓取网页来验证刚才创建的 sample 站点：

```
$ curl 127.0.0.1:49172
Hello Docker!
```

读者也可以在其他设备上通过访问宿主主机 ip:49172 来访问 sample 站点。

## Dockerfile 创建的镜像拥有继承的特性

不知道有没有细心的读者发现，在 apache 镜像的 Dockerfile 中只用 EXPOSE 定义了对外开放的 80 端口，而在 sudo docker ps -a 命令的返回中，却看到新启动的容器映射了两个端口：22 和 80。

但是实际上，当尝试使用 SSH 登录到容器时，会发现无法登录。

这是因为在 run.sh 脚本中并未启动 SSH 服务。

这说明在使用 Dockerfile 创建镜像时，会继承父镜像的开放端口，但却不会继承启动命令。因此，需要在 run.sh 脚本中添加启动 sshd 的服务的命令：

```
$ cat run.sh
#!/bin/bash
/usr/sbin/sshd &
exec apache2 -D FOREGROUND
```

再次创建镜像：

```
$ sudo docker build -t apache:ubuntu .
```

这次创建的镜像将默认会同时启动 SSH 和 Apache 服务。

## 映射本地目录

可以通过映射本地目录的方式来指定容器内 Apache 服务响应的内容，例如映射本地主机上当前目录下的 www 目录到容器内的 /var/www 目录：

```
$ sudo docker run -i -d -p 80:80 -p 103:22 -e APACHE_SERVERNAME=test -v `pwd`/www:/var/www:ro apache:ubuntu
```

在当前目录内创建 www 目录，并放上自定义的页面 index.html，内容为：

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>Hi Docker</title>
</head><body>
<h1>Hi Docker</h1>
<p>This is the first day I meet the new world.</p>
<p>How are you?</p>
<hr>
<address>Apache/2.4.7 (Ubuntu) Server at 127.0.0.1 Port 80</address>
</body></html>
```

在本地主机上可访问测试容器提供的 Web 服务，查看获取内容为新配置的 index.html 页面信息：

## 11.2 Nginx

Nginx 是一个高性能的 Web 和反向代理服务器，它具有很多非常优越的特性：

- 作为 Web 服务器：相比 Apache，Nginx 使用更少的资源，支持更多的并发连接，体现更高的效率，这点使 Nginx 尤其受到虚拟主机提供商的欢迎。一个 Nginx 实例能够轻松支持高达 50 000 个并发连接数的响应。
- 作为负载均衡服务器：Nginx 既可以在内部直接支持 Rails 和 PHP，也可以支持作为 HTTP 代理服务器对外进行服务。Nginx 用 C 编写，不论是系统资源开销还是 CPU 使用效率都比 Perlbal 要好得多。
- 作为邮件代理服务器：Nginx 同时也是一个非常优秀的邮件代理服务器（最早开发这个产品的目的之一也是作为邮件代理服务器），Last.fm 描述了成功并且美妙的使用经验。
- Nginx 安装非常简单，配置文件非常简洁（还能够支持 Perl 语法），Bug 非常少。Nginx 启动特别容易，并且几乎可以做到 7×24 不间断运行，即使运行数个月也不需要重新启动。你還能够在不间断服务的情况下进行软件版本的升级。

本节将首先介绍 Nginx 官方发行版本的镜像生成，然后介绍在国内应用量众多的 Nginx 淘宝增强版——Tengine 镜像的生成。

### Nginx 官方版本

由于使用 Dockerfile 生成镜像的步骤大多类似。为了节约篇幅，这里直接介绍使用的 Dockerfile 文件和需要的脚本文件，如果读者对使用 Dockerfile 创建镜像的步骤还有不清楚的地方，可以查看第一部分中关于 Dockerfile 的介绍章节和上一小节 Apache 镜像的创建过程。

#### 1. Nginx Dockerfile

```
# 设置继承自创建的 sshd 镜像
FROM sshd:dockerfile

# 下面是一些创建者的基本信息
MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)

# 安装 nginx，设置 nginx 以非 daemon 启动。
RUN \
    apt-get install -y nginx && \
    rm -rf /var/lib/apt/lists/* && \
    echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
    chown -R www-data:www-data /var/lib/nginx
```

```

RUN echo "Asia/Shanghai" > /etc/timezone && \
    dpkg-reconfigure -f noninteractive tzdata
# 注意这里要更改系统的时区设置，因为在Web应用中经常会用到时区这个系统变量，默认的ubuntu会让你的应用程序发生不可思议的效果哦

# 添加我们的脚本，并设置权限，这会覆盖之前放在这个位置的脚本
ADD run.sh /run.sh
RUN chmod 755 /*.sh

# 定义可以被挂载的目录，分别是虚拟主机的挂载目录、证书目录、配置目录和日志目录
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/conf.d", "/var/log/nginx"]

# 定义工作目录
WORKDIR /etc/nginx

# 定义输出命令
CMD ["/run.sh"]

# 定义输出端口
EXPOSE 80
EXPOSE 443

```

## 2. 查看 run.sh 脚本文件内容

```
$ cat run.sh
#!/bin/bash
/usr/sbin/sshd &
/usr/sbin/nginx
```

## 3. 创建镜像

使用 docker build 命令，创建镜像 nginx:stable：

```
$ sudo docker build -t nginx:stable .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM sshd:dockerfile
--> 570c26a9de68
Step 1 : MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)
--> Using cache
--> 5c6b90057a1d
Step 2 : RUN apt-get install -y nginx && rm -rf /var/lib/apt/lists/* &&
echo "\ndaemon off;" >> /etc/nginx/nginx.conf && chown -R www-data:www-data /
var/lib/nginx
--> Using cache
--> 79149a429c7a
Step 3 : ADD run.sh /run.sh
```

```

--> ab72d235e438
Removing intermediate container 5c16bf1046aa
Step 4 : RUN chmod 755 /*.sh
--> Running in a0c0b0ec6bcc
--> 8d3262eaf1b8
Removing intermediate container a0c0b0ec6bcc
Step 5 : VOLUME /etc/nginx/sites-enabled /etc/nginx/certs /etc/nginx/conf.d /var/log/nginx
--> Running in 6966fb517eed
--> 3b64e8cb7119
Removing intermediate container 6966fb517eed
Step 6 : WORKDIR /etc/nginx
--> Running in 391a0b606082
--> 666fb7e351fe
Removing intermediate container 391a0b606082
Step 7 : CMD /run.sh
--> Running in 9f1e239daf52
--> e1c0b7bde8cf
Removing intermediate container 9f1e239daf52
Step 8 : EXPOSE 80
--> Running in 36d7b8f0e7cf
--> fcccd40af3367
Removing intermediate container 36d7b8f0e7cf
Step 9 : EXPOSE 443
--> Running in 84095d6a71c2
--> 4e3936e36e31
Removing intermediate container 84095d6a71c2
Successfully built 4e3936e36e3

```

#### 4. 测试

启动容器，查看内部的 80 端口被映射到本地的 49193 端口：

```

$ sudo docker run -d -P nginx:stable
08c456536e69c8e36670f3bc6b496020e76d28fc9d33a8bcd01ff6d61bc72c4a
$ sudo docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED          STATUS          PORTS          NAMES
08c456536e69        nginx:stable   "/run.sh"     8 seconds ago   Up 8 seconds
0.0.0.0:49191->22/tcp, 0.0.0.0:49192->443/tcp, 0.0.0.0:49193->80/tcp

```

访问本地的 49193 端口：

```
$ curl 127.0.0.1:49193
```

返回 Nginx 的欢迎页面，说明 Nginx 已经正常启动了：

```

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>

```

```
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

最后，为了能充分发挥 Nginx 的性能，可对系统内核参数做一些调整。

下面是一份常见的 Nginx 内核的优化参数：

```
net.ipv4.ip_forward = 0
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.default.accept_source_route = 0
kernel.sysrq = 0
kernel.core_uses_pid = 1
net.ipv4.tcp_syncookies = 1
kernel.msgmnb = 65536
kernel.msgmax = 65536
kernel.shmmmax = 68719476736
kernel.shmall = 4294967296
net.ipv4.tcp_max_tw_buckets = 6000
net.ipv4.tcp_sack = 1
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_rmem = 4096 87380 4194304
net.ipv4.tcp_wmem = 4096 16384 4194304
net.core.wmem_default = 8388608
net.core.rmem_default = 8388608
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.core.netdev_max_backlog = 262144
net.core.somaxconn = 262144
net.ipv4.tcp_max_orphans = 3276800
net.ipv4.tcp_max_syn_backlog = 262144
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_synack_retries = 1
```

```

net.ipv4.tcp_syn_retries = 1
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_mem = 94500000 915000000 927000000
net.ipv4.tcp_fin_timeout = 1
net.ipv4.tcp_keepalive_time = 30
net.ipv4.ip_local_port_range = 1024 65000

```

## Tengine 镜像

### 1. Tengine Dockerfile

```

# 设置继承自我们创建的 sshd 镜像
FROM sshd:dockerfile

# 下面是一些创建者的基本信息
MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)

# Let the container know that there is no tty

# 安装编译环境
RUN apt-get install -y build-essential debhelper make autoconf automake patch
RUN apt-get install -y dpkg-dev fakeroot pbuilder gnupg dh-make libssl-dev
libpcre3-dev git-core

RUN echo "Asia/Shanghai" > /etc/timezone && \
    dpkg-reconfigure -f noninteractive tzdata
# 注意这里要更改系统的时区设置，因为在 Web 应用中经常会用到时区这个系统变量，默认的 ubuntu 会让你的应用程序发生不可思议的效果哦

# 创建 Nginx 用户
RUN adduser --disabled-login --gecos 'Tengine' nginx

# tengine 安装的 shell 脚本
WORKDIR /home/nginx
RUN su nginx -c 'git clone https://github.com/alibaba/tengine.git'

WORKDIR /home/nginx/tengine
RUN su nginx -c 'mv packages/debian .'

ENV DEB_BUILD_OPTIONS nocheck

RUN su nginx -c 'dpkg-buildpackage -rfakeroot -uc -b'

WORKDIR /home/nginx
RUN dpkg -i tengine_2.0.2-1_amd64.deb

# 定义挂载的目录

```

```
VOLUME ["/data", "/etc/nginx/sites-enabled", "/var/log/nginx"]

# 让 Nginx 运行在排 Daemon 模式
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf

# 定义工作目录
WORKDIR /etc/nginx

# 添加我们的脚本，并设置权限，这会覆盖之前放在这个位置的脚本
ADD run.sh /run.sh
RUN chmod 755 /*.sh

# 定义输出命令
CMD ["/run.sh"]

# 定义输出端口
EXPOSE 80
EXPOSE 443
```

## 2. 查看 run.sh 脚本文件内容

```
$ cat run.sh
#!/bin/bash
/usr/sbin/sshd &
/usr/sbin/nginx
```

## 3. 创建过程

```
$ sudo docker build -t nginx:albb .
```

## 4. 测试

启动一个容器，并查看端口映射信息：

```
$ sudo docker run -d -P nginx:albb
ff4650e77c53b174a10b4cd29533deffad889458f88d98c4443ac3654b01552a
$ sudo docker ps
CONTAINER ID        IMAGE          COMMAND       CREATED      STATUS      PORTS      NAMES
ff4650e77c53        nginx:albb    "/run.sh"     3 seconds ago   Up 2 seconds
0.0.0.0:49194->443/tcp, 0.0.0.0:49195->80/tcp, 0.0.0.0:49196->22/tcp   furious_wright
08c456536e69        nginx:stable   "/run.sh"     13 minutes ago  Up 13
minutes            0.0.0.0:49191->22/tcp, 0.0.0.0:49192->443/tcp, 0.0.0.0:49193->80/
tcp    romantic_curie
ffd58545b787        apache:ubuntu  "/run.sh"     About an hour ago  Up About
an hour            0.0.0.0:49177->22/tcp, 0.0.0.0:49178->80/tcp   jovial_galileo
```

访问本地的 49195 端口进行测试：

```
$ curl 127.0.0.1:49195
```

返回的内容是淘宝版本的 Nginx 特有的页面：

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to tengine!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to tengine!</h1>
<p>If you see this page, the tengine web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://tengine.taobao.org/">tengine.taobao.org</a>.</p>

<p><em>Thank you for using tengine.</em></p>
</body>
</html>
```

## 5. 进入容器查看创建的容器信息

可以使用 docker exec 命令进入刚启动的 Tengine 容器，查看建立容器后默认运行的进程和默认映射的端口：

```
$ sudo docker exec -ti ff4 /bin/bash
root@ff4650e77c53:/etc/nginx# ps -ef
UID      PID  PPID  C STIME TTY TIME CMD
root        1      0  0 15:09 ?  00:00:00 /bin/bash /run.sh
root       11      1  0 15:09 ?  00:00:00 nginx: master process /usr/sbin/nginx
nginx      12     11  0 15:09 ?  00:00:00 nginx: worker process
root       13      1  0 15:09 ?  00:00:00 /usr/sbin/sshd
root       14      0  1 15:09 ?  00:00:00 /bin/bash
root      23     14  0 15:09 ?  00:00:00 ps -ef
root@ff4650e77c53:/etc/nginx# netstat -tunlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State      PID/Program name
tcp        0      0 0.0.0.0:80           0.0.0.0:*        LISTEN     -
tcp        0      0 0.0.0.0:22           0.0.0.0:*        LISTEN     -
tcp6       0      0 ::1:22              ::*:*             LISTEN     -
```

查看 Tengine 的编译参数和模块特性：

```
root@ff4650e77c53:/etc/nginx# nginx -V
Tengine version: Tengine/2.0.3 (nginx/1.6.1)
built by gcc 4.8.2 (Ubuntu 4.8.2-19ubuntu1)
TLS SNI support enabled
configure arguments: --prefix=/etc/nginx --sbin-path=/usr/sbin/nginx --conf-path=/etc/nginx/nginx.conf --error-log-path=/var/log/nginx/error.log --http-log-path=/var/log/nginx/access.log --pid-path=/var/run/nginx.pid --lock-path=/var/run/nginx.lock --http-client-body-temp-path=/var/cache/nginx/client_temp --http-proxy-temp-path=/var/cache/nginx/proxy_temp --http-fastcgi-temp-path=/var/cache/nginx/fastcgi_temp --http-uwsgi-temp-path=/var/cache/nginx/uwsgi_temp --http-scgi-temp-path=/var/cache/nginx/scgi_temp --user=nginx --group=nginx --with-http_ssl_module --with-http_realip_module --with-http_addition_module --with-http_sub_module --with-http_dav_module --with-http_flv_module --with-http_mp4_module --with-http_gunzip_module --with-http_gzip_static_module --with-http_random_index_module --with-http_secure_link_module --with-http_stub_status_module --with-mail --with-mail_ssl_module --with-file-aio --with-http_spdy_module --with-cc-opt='-g -O2 -fstack-protector --param=ssp-buffer-size=4 -Wformat -Werror=format-security' --with-ld-opt='-Wl,-Bsymbolic-functions -Wl, -z, relro' --with-ipv6
loaded modules:
    ngx_core_module (static)
    ngx_errlog_module (static)
    ngx_conf_module (static)
    ngx_dso_module (static)
    ngx_syslog_module (static)
    ngx_events_module (static)
    ngx_event_core_module (static)
    ngx_epoll_module (static)
    ngx_procs_module (static)
    ngx_proc_core_module (static)
    ngx_openssl_module (static)
    ngx_regex_module (static)
    ngx_http_module (static)
    ngx_http_core_module (static)
    ngx_http_log_module (static)
    ngx_http_upstream_module (static)
    ngx_http_spdy_module (static)
    ngx_http_static_module (static)
    ngx_http_gzip_static_module (static)
    ngx_http_dav_module (static)
    ngx_http_autoindex_module (static)
    ngx_http_index_module (static)
    ngx_http_random_index_module (static)
    ngx_http_auth_basic_module (static)
    ngx_http_access_module (static)
    ngx_http_limit_conn_module (static)
    ngx_http_limit_req_module (static)
    ngx_http_realip_module (static)
    ngx_http_geo_module (static)
```

```

ngx_http_map_module (static)
ngx_http_split_clients_module (static)
ngx_http_referer_module (static)
ngx_http_rewrite_module (static)
ngx_http_ssl_module (static)
ngx_http_proxy_module (static)
ngx_http_fastcgi_module (static)
ngx_http_uwsgi_module (static)
ngx_http_scgi_module (static)
ngx_http_memcached_module (static)
ngx_http_empty_gif_module (static)
ngx_http_browser_module (static)
ngx_http_user_agent_module (static)
ngx_http_secure_link_module (static)
ngx_http_flv_module (static)
ngx_http_mp4_module (static)
ngx_http_upstream_ip_hash_module (static)
ngx_http_upstream_consistent_hash_module (static)
ngx_http_upstream_check_module (static)
ngx_http_upstream_least_conn_module (static)
ngx_http_reqstat_module (static)
ngx_http_upstream_keepalive_module (static)
ngx_http_upstream_dynamic_module (static)
ngx_http_stub_status_module (static)
ngx_http_write_filter_module (static)
ngx_http_header_filter_module (static)
ngx_http_chunked_filter_module (static)
ngx_http_spdy_filter_module (static)
ngx_http_range_header_filter_module (static)
ngx_http_gzip_filter_module (static)
ngx_http_postpone_filter_module (static)
ngx_http_ssi_filter_module (static)
ngx_http_charset_filter_module (static)
ngx_http_sub_filter_module (static)
ngx_http_addition_filter_module (static)
ngx_http_gunzip_filter_module (static)
ngx_http_userid_filter_module (static)
ngx_http_footer_filter_module (static)
ngx_http_trim_filter_module (static)
ngx_http_headers_filter_module (static)
ngx_http_upstream_session_sticky_module (static)
ngx_http_copy_filter_module (static)
ngx_http_range_body_filter_module (static)
ngx_http_not_modified_filter_module (static)
ngx_mail_module (static)
ngx_mail_core_module (static)
ngx_mail_ssl_module (static)
ngx_mail_pop3_module (static)
ngx_mail_imap_module (static)
ngx_mail_smtp_module (static)

```

```
nginx_mail_auth_http_module (static)
nginx_mail_proxy_module (static)
root@ff4650e77c53:/etc/nginx#
```

## 11.3 Tomcat

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下普遍使用，是开发和调试 JSP 程序的首选。

Tomcat 最初是由 Sun 的软件构架师詹姆斯·邓肯·戴维森开发的。后来在他的帮助下 Tomcat 成为开源项目，并由 Sun 贡献给 Apache 软件基金会。

当配置正确时，Apache 为 HTML 页面服务，而 Tomcat 实际上运行 JSP 页面和 Servlet。另外，Tomcat 和 IIS 等 Web 服务器一样，具有处理 HTML 页面的功能（但处理静态 HTML 的能力不如 Apache），另外它还是一个 Servlet 和 JSP 容器，独立的 Servlet 容器是 Tomcat 的默认模式。

### 设计 Tomcat 的 Dockerfile

首先，尝试在 Dockerhub 上搜索 Tomcat 相关镜像的个数：

```
$ sudo docker search tomcat |wc -l
285
```

可以看到，已经有 285 个相关镜像。如是个人开发或测试，可以随意选择一个镜像，按照提示启动应用即可。若准备在生产环境中使用，这些镜像都不是那么合适了。原因有三个方面：

- 项目需要的 Tomcat 版本可能不同。
- 项目需要的 Tomcat 变量不一致。
- 项目需要的 JDK 可能不一致。

因此，比较好的方式应该是由架构师通过 Dockerfile 或者其他方式构建好统一的镜像，然后分发给项目组所有成员来进行。

下面以 sun\_jdk 1.6、tomcat 7.0、ubuntu 14.04 为环境介绍如何定制自己的 tomcat 镜像。

### 准备工作

创建 tomcat7.0\_jdk1.6 文件夹，从 [www.oracle.com](http://www.oracle.com) 上下载 sun\_jdk 1.6 压缩包，解压为 jdk 目录。

创建 Dockerfile 和 run.sh 文件：

```
$ mkdir tomcat7.0_jdk1.6
$ cd tomcat7.0_jdk1.6/
$ touch Dockerfile run.sh
```

下载 Tomcat，可以到官方网站下载最新的版本，也可以直接使用下面链接中给出的版本：

```
$ wget http://mirror.bit.edu.cn/apache/tomcat/tomcat-7/v7.0.56/bin/apache-
tomcat-7.0.56.zip
--2014-10-27 22:25:23--  http://mirror.bit.edu.cn/apache/tomcat/tomcat-7/
v7.0.56/bin/apache-tomcat-7.0.56.zip
Resolving mirror.bit.edu.cn (mirror.bit.edu.cn)... 219.143.204.117, 2001:da8:20
4:2001:250:56ff:feal:22
Connecting to mirror.bit.edu.cn (mirror.bit.edu.cn)|219.143.204.117|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 9466255 (9.0M) [application/zip]
Saving to: 'apache-tomcat-7.0.56.zip'

100%[=====>] 9,466,255      152KB/s   in 70s

2014-10-27 22:26:34 (131 KB/s) - 'apache-tomcat-7.0.56.zip' saved
[9466255/9466255]
```

解压后，tomcat7.0\_jdk1.6 目录底下应如下所示（多余的压缩包文件已经被删除）：

```
$ ls
Dockerfile  apache-tomcat-7.0.56  jdk  run.sh
```

## Dockerfile 文件和其他脚本文件

Dockerfile 文件内容如下：

```
FROM sshd:dockerfile
# 设置继承自我们创建的 sshd 镜像
MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)
# 下面是一些创建者的基本信息

# 设置环境变量，所有操作都是非交互式的
ENV DEBIAN_FRONTEND noninteractive

RUN echo "Asia/Shanghai" > /etc/timezone && \
    dpkg-reconfigure -f noninteractive tzdata

# 注意这里要更改系统的时区设置，因为在 Web 应用中经常会用到时区这个系统变量，默认的 ubuntu 会让你的应用程序发生不可思议的效果哦

# 安装跟 tomcat 用户认证相关的软件
RUN apt-get install -yq --no-install-recommends wget pwgen ca-certificates && \
    apt-get clean && \
```

```

rm -rf /var/lib/apt/lists/*
# 设置 tomcat 的环境变量，若读者有其他的环境变量需要设置，也可以在这里添加。
ENV CATALINA_HOME /tomcat
ENV JAVA_HOME /jdk

# 复制 tomcat 和 jdk 文件到镜像中
ADD apache-tomcat-7.0.56 /tomcat
ADD jdk /jdk

ADD create_tomcat_admin_user.sh /create_tomcat_admin_user.sh
ADD run.sh /run.sh
RUN chmod +x /*.sh
RUN chmod +x /tomcat/bin/*.sh

EXPOSE 8080
CMD ["/run.sh"]

```

创建 tomcat 用户和密码脚本文件 `create_tomcat_admin_user.sh` 文件，内容为：

```

#!/bin/bash

if [ -f /.tomcat_admin_created ]; then
    echo "Tomcat 'admin' user already created"
    exit 0
fi

#generate password
PASS=$(TOMCAT_PASS:-$(pwgen -s 12 1))
_word=$( [ ${TOMCAT_PASS} ] && echo "preset" || echo "random" )

echo "=> Creating and admin user with a ${_word} password in Tomcat"
sed -i -r 's/<\!tomcat-users>//' ${CATALINA_HOME}/conf/tomcat-users.xml
echo '<role rolename="manager-gui"/>' >> ${CATALINA_HOME}/conf/tomcat-users.xml
echo '<role rolename="manager-script"/>' >> ${CATALINA_HOME}/conf/tomcat-users.xml
echo '<role rolename="manager-jmx"/>' >> ${CATALINA_HOME}/conf/tomcat-users.xml
echo '<role rolename="admin-gui"/>' >> ${CATALINA_HOME}/conf/tomcat-users.xml
echo '<role rolename="admin-script"/>' >> ${CATALINA_HOME}/conf/tomcat-users.xml
echo "<user username=\"admin\" password=\"$${PASS}\" roles=\"manager-gui, manager-script, manager-jmx, admin-gui, admin-script\"/>" >> ${CATALINA_HOME}/conf/tomcat-users.xml
echo '</tomcat-users>' >> ${CATALINA_HOME}/conf/tomcat-users.xml
echo "=> Done!"

touch /.tomcat_admin_created

echo =====
echo "You can now configure to this Tomcat server using:"
echo ""
echo "    admin:${PASS}"
echo ""
echo ====="

```

编写 run.sh 脚本文件，内容为：

```
#!/bin/bash

if [ ! -f /.tomcat_admin_created ]; then
    /create_tomcat_admin_user.sh
fi
/usr/sbin/sshd -D &
exec ${CATALINA_HOME}/bin/catalina.sh run
```

## 创建和测试镜像

通过下面的命令创建镜像 tomcat7.0:jk1.6：

```
$ sudo docker build -t tomcat7.0:jk1.6 .
Sending build context to Docker daemon 234.8 MB
Sending build context to Docker daemon
Step 0 : FROM sshd:dockerfile
--> 570c26a9de68
Step 1 : MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)
--> Using cache
--> 5c6b90057a1d
Step 2 : ENV DEBIAN_FRONTEND noninteractive
--> Using cache
--> e06feb0790d7
Step 3 : RUN echo "Asia/Shanghai" > /etc/timezone && dpkg-reconfigure -f
noninteractive tzdata
--> Running in 6dba2d312627
Current default time zone: 'Asia/Shanghai'
Local time is now: Tue Oct 28 13:47:08 CST 2014. UTC offset: +8
Universal Time is now: Tue Oct 28 05:47:08 UTC 2014.
--> a1dccb384edb
Removing intermediate container 6dba2d312627
...
Setting up pwgen (2.06-1ubuntu4) ...
--> e0e4ab118cda
Removing intermediate container aee38d8ab936
Step 5 : ENV CATALINA_HOME /tomcat
--> Running in 8d0d7176fb7e
--> e4d8891f4e86
Removing intermediate container 8d0d7176fb7e
Step 6 : ENV JAVA_HOME /jdk
--> Running in 53ce1fa9b8a0
--> f17a13a87981
Removing intermediate container 53ce1fa9b8a0
Step 7 : ADD apache-tomcat-7.0.56 /tomcat
--> calfa71b4130
Removing intermediate container 27e2d96bcb78
```

```
Step 8 : ADD jdk /jdk
--> d7a595c4c4f9
Removing intermediate container 00d980ad2cad
Step 9 : ADD create_tomcat_admin_user.sh /create_tomcat_admin_user.sh
--> 5055ca84decc
Removing intermediate container 220922d934ce
Step 10 : ADD run.sh /run.sh
--> da469edb1022
Removing intermediate container f0dde8563174
Step 11 : RUN chmod +x /*.sh
--> Running in 71564c350a2e
--> 5f566293e37c
Removing intermediate container 71564c350a2e
Step 12 : EXPOSE 8080
--> Running in 055c41de3bd8
--> b1213c1bc920
Removing intermediate container 055c41de3bd8
Step 13 : CMD /run.sh
--> Running in 5dbe1220a559
--> ce78537c247d
Removing intermediate container 5dbe1220a559
Successfully built ce78537c247d
```

查看下目前本地拥有的镜像：

```
$ sudo docker images
REPOSITORY      TAG          IMAGE ID       CREATED        VIRTUAL SIZE
tomcat7.0        jdk1.6      ce78537c247d   9 minutes ago  473.3 MB
nginx           albb         8e333a6f1d10   14 hours ago   567.6 MB
nginx           stable       4e3936e36e31   15 hours ago   262.3 MB
apache          ubuntu       06d84c79e905   16 hours ago   263.8 MB
sshd            dockerfile   570c26a9de68   26 hours ago   246.5 MB
sshd            ubuntu       7aef2cd95fd0   39 hours ago   255.2 MB
debian          latest       61f7f4f722fb   7 days ago    85.1 MB
busybox          latest       e72ac664f4f0   3 weeks ago   2.433 MB
ubuntu           14.04       ba5877dc9bec   3 months ago   192.7 MB
ubuntu           latest       ba5877dc9bec   3 months ago   192.7 MB
```

启动一个 tomcat 容器进行测试：

```
$ sudo docker run -d -P tomcat7.0:jdk1.6
3cd4238cb32a713a3alc29d93fbfc80cba150653b5eb8bd7629bee957e7378ed
```

通过 docker logs 得到 tomcat 的密码 aBwN0CNCPckw：

```
$ sudo docker logs 3cd
=> Creating and admin user with a random password in Tomcat
=> Done!
=====
You can now configure to this Tomcat server using:
```

```
admin:aBwN0CNCpckw
```

```
=====
Oct 28, 2014 2:02:24 PM org.apache.catalina.core.AprLifecycleListener init
INFO: The APR based Apache Tomcat Native library which allows optimal
performance in production environments was not found on the java.library.path:
/jdk/jre/lib/amd64/server:/jdk/jre/lib/amd64:/jdk/jre/../lib/amd64:/usr/java/
packages/lib/amd64:/usr/lib64:/lib:/usr/lib
Oct 28, 2014 2:02:24 PM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-bio-8080"]
Oct 28, 2014 2:02:24 PM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["ajp-bio-8009"]
Oct 28, 2014 2:02:24 PM org.apache.catalina.startup.Catalina load
```

查看映射的端口信息：

```
$ sudo docker ps
CONTAINER ID        IMAGE          COMMAND       CREATED      STATUS      PORTS          NAMES
3cd4238cb32a        tomcat7.0:jdk1.6    "/run.sh"     4 seconds ago   Up 3
seconds            0.0.0.0:49157->22/tcp, 0.0.0.0:49158->8080/tcp   cranky_wright
```

在本地使用浏览器登录 Tomcat 管理界面，请访问 <http://127.0.0.1:49158>：如图 11-1 所示。

图 11-1 Tomcat 管理界面

输入从 docker logs 中得到的密码，如图 11-2 所示。

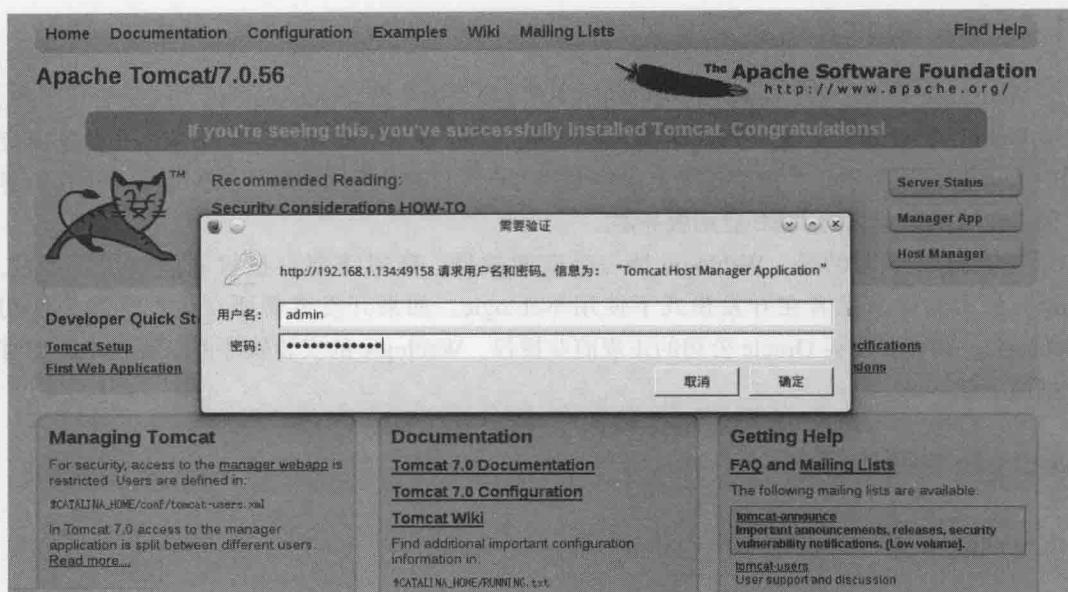


图 11-2 输入管理员密码

成功进入管理界面如图 11-3 所示。

The screenshot shows the Tomcat Virtual Host Manager interface. At the top, it features the Apache Software Foundation logo and the URL <http://www.apache.org/>. To the right is a cartoon cat icon. The main title is "Tomcat Virtual Host Manager". Below the title, there's a "Message" field containing "OK". A "Host Manager" section has tabs for "List Virtual Hosts", "HTML Host Manager Help (TODO)", "Host Manager Help (TODO)", and "Server Status". Under "Host name", there's a table with one row for "localhost" where the "Commands" column says "Host Manager installed - commands disabled". A "Add Virtual Host" section allows adding a host with fields for "Name", "Aliases", and "App base".

图 11-3 Apache 管理界面



**注意** 在实际环境中，可以通过使用 `-v` 参数来挂载 Tomcat 的日志文件、程序所在目录、以及与 Tomcat 相关的配置。

## 11.4 Weblogic

WebLogic 是一个基于 Java EE 架构的中间件（应用服务器），WebLogic 由 Oracle 公司维护。

WebLogic 是用于开发、集成、部署和管理大型分布式 Web 应用、网络应用和数据库应用的 Java 应用服务器。是商业市场上最主要的 Java (J2EE) 应用服务器软件之一，也是世界上第一个成功商业化的 J2EE 应用服务器。

与 Tomcat 不同的是，Weblogic 是一个商业软件，所以需要有授权才能使用。不过，Oracle 公司允许开发者在开发模式下使用 Weblogic。如果开发者需要在生产环境中使用 Weblogic，则需要购买 Oracle 公司的正规商业授权。Weblogic 的安装软件可以到 Oracle 的官方网站下载。

### Weblogic 的基本概念

#### Weblogic 域

Weblogic 域是作为单元进行管理的一组相关的 WebLogic 服务器资源。一个域包含一个或多个 WebLogic 服务器实例，这些实例可以是群集实例、非群集实例，或者群集与非群集实例的组合。一个域可以包含多个群集。域还包含部署在域中的应用程序组件、此域中的这些应用程序组件和服务器实例所需的资源和服务。应用程序和服务器实例使用的资源和服务示例包括计算机定义、可选网络通道、连接器和启动类。

#### Administration 服务器

域中包含一个特殊的 WebLogic 服务器实例，叫做 Administration 服务器，这是用户配置、管理域中所有资源的核心。

#### Managed 服务器

通常，称加入 Domain 的其他实例为 Managed 服务器，所有的 Web 应用、EJB、Web 服务和其他资源都部署在这些服务器上。

一个典型的 Weblogic 部署应该如图 11-4 所示。

如果要使用常规的 administrator +node 的方式部署，就需要在 run.sh 脚本中分别写出 administrator 服务器和 node 服务器的启动脚本。这样做的优点是：可以使用 Weblogic 的集群、同步等概念。部署一个集群应用程序，只需要安装一次应用到集群上即可。

缺点是：

- Docker 配置复杂了。
- 没办法自动扩展集群的计算容量，如需添加节点，需要在 administrator 上先创建节点，然后再配置新的容器 run.sh 启动脚本，然后再启动容器。

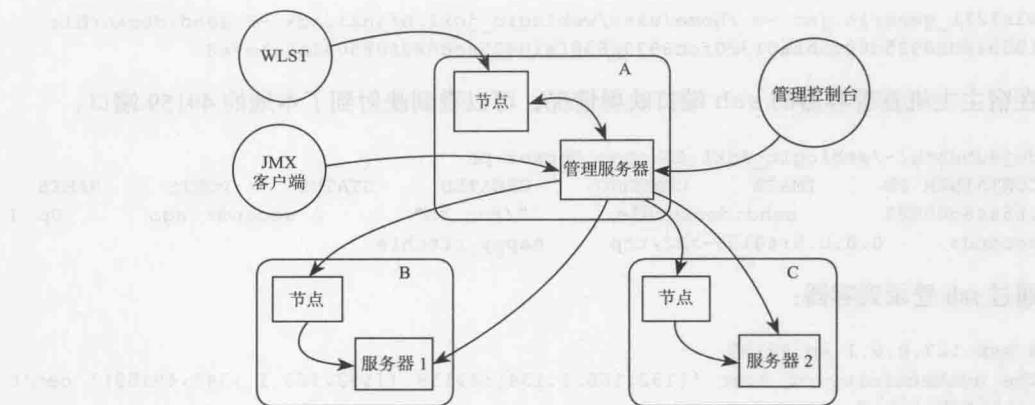


图 11-4 典型的 Weblogic 部署

笔者推荐将应用程序安装在 adminiserver 上面，需要扩展的时候，启动多个 adminiserver 节点即可，将 adminiserver 当作 Manged server 使用。这样做的优点和缺点和传统的部署方法恰恰相反。

## 使用 docker commit + Dockerfile 方式创建镜像

下面笔者将以 Weblogic 12.11、jdk 1.6、Ubuntu14.04 为例子，创建一个带有 Weblogic 服务的镜像。

### 1. 准备工作

由于 Weblogic 的安装、部署方式较为复杂，笔者将先通过 `docker run -ti` 进入容器完成大部分操作，然后通过 `docker commit` 将这个容器提交为一个镜像，最后再进一步使用 Dockerfile 来完成最终的 Weblogic 镜像创建，对于一些复杂镜像的创建，读者也可以参考这种方法。

在本地主机上创建 weblogic 目录，从其他主机上传 jdk 和 weblogic 安装文件到该目录下，并创建 Dockerfile 和 run.sh 脚本文件：

```
$ mkdir weblogic_jdk1.6
$ cd weblogic_jdk1.6/
$ touch Dockerfile run.sh
$ ls
Dockerfile jdk run.sh wls1211_generic.jar
```

### 2. 安装 Weblogic 到容器

使用 `-v-d-P` 参数运行之前创建的 `sshd_dockerfile` 镜像：

```
$ sudo docker run -d -v /home/user/weblogic_jdk1.6/wls1211_generic.jar:/
```

```
wls1211_generic.jar -v /home/user/weblogic_jdk1.6/jdk:/jdk -P sshd:dockerfile
185546d00925d80c5bfa01320feb8939c838fa10429acc648f8850efb8ale968
```

在宿主主机查看容器的 ssh 端口映射情况，可以看到映射到了本地的 49159 端口：

```
dwj@ubuntu:~/weblogic_jdk1.6$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
185546d00925        sshd:dockerfile   "/run.sh"          2 seconds ago    Up 1 second        0.0.0.0:49159->22/tcp   happy_ritchie
```

通过 ssh 登录到容器：

```
$ ssh 127.0.0.1 -p 49159
The authenticity of host '[192.168.1.134]:49159 ([192.168.1.134]:49159)' can't
be established.
ECDSA key fingerprint is d1:59:f1:09:3b:09:79:6d:19:16:f4:fd:39:1b:be:27.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.1.134]:49159' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.2.0-37-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

```

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/\*/\*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

找到映射到容器内的 weblogic 安装文件：

```
root@185546d00925:/# ls -lh
total 997M
drwxr-xr-x  2 root root 4.0K Jul 17 03:38 bin
drwxr-xr-x  2 root root 4.0K Apr 10 2014 boot
drwxr-xr-x  4 root root  340 Oct 28 07:53 dev
drwxr-xr-x  83 root root 4.0K Oct 28 07:53 etc
drwxr-xr-x  2 root root 4.0K Apr 10 2014 home
drwxr-xr-x  8 1000 1000 4.0K Oct 28 07:43 jdk
drwxr-xr-x 14 root root 4.0K Oct 27 03:01 lib
drwxr-xr-x  2 root root 4.0K Jul 17 03:34 lib64
drwxr-xr-x  2 root root 4.0K Jul 17 03:34 media
drwxr-xr-x  2 root root 4.0K Apr 10 2014 mnt
drwxr-xr-x  2 root root 4.0K Jul 17 03:34 opt
dr-xr-xr-x 253 root root    0 Oct 28 07:53 proc
drwx-----  5 root root 4.0K Oct 28 07:55 root
drwxr-xr-x  8 root root 4.0K Oct 28 07:55 run
-rwxr-xr-x  1 root root  30 Oct 27 02:51 run.sh
drwxr-xr-x  2 root root 4.0K Jul 21 21:47 sbin
drwxr-xr-x  2 root root 4.0K Jul 17 03:34 srv
dr-xr-xr-x 13 root root    0 Oct 28 07:53 sys
```

```
drwxrwxrwt  3 root root 4.0K Oct 28 07:56 tmp
drwxr-xr-x 16 root root 4.0K Oct 27 03:01 usr
drwxr-xr-x 19 root root 4.0K Oct 28 07:55 var
-rw-----  1 1000 1000 997M Oct 28 07:39 wls1211_generic.jar
```

笔者使用命令行模式，在容器内进行 weblogic 的安装。

以 console 模式启动 weblogic 安装，默认使用图形界面安装：

```
root@185546d00925:/# ./jdk/bin/java -jar wls1211_generic.jar -mode=console

Extracting 0%.....100%
----- Oracle Installer - WebLogic 12.1.1.0 -----
Welcome:
-----
This installer will guide you through the installation of WebLogic 12.1.1.0.
Type "Next" or enter to proceed to the next prompt. If you want to change data
entered previously, type "Previous". You may quit the installer at any time by
typing "Exit".
```

Enter [Exit] [Next]>

不使用默认的安装位置，我们将 weblogic 安装在 opt 目录下面：

```
----- Oracle Installer - WebLogic 12.1.1.0 -----
Choose Middleware Home Directory:
-----
"Middleware Home" = [Enter new value or use default "/root/Oracle/Middleware"]
Enter new Middleware Home OR [Exit] [Previous] [Next]> /opt/Middleware
----- Oracle Installer - WebLogic 12.1.1.0 -----
Choose Middleware Home Directory:
-----
"Middleware Home" = [/opt/Middleware]
Use above value or select another option:
1 - Enter new Middleware Home
2 - Change to default [/root/Oracle/Middleware]
```

Enter option number to select OR [Exit] [Previous] [Next]>

因为我们这里使用的授权时开发模式的，所以选择不接收安全更新。

```

<----- Oracle Installer - WebLogic 12.1.1.0 ----->
Register for Security Updates:
----->

Provide your email address for security updates and to initiate configuration manager.

1|Email:[]
2|Support Password:[]
3|Receive Security Update:[Yes]

Enter index number to select OR [Exit][Previous][Next]> 3
<----- Oracle Installer - WebLogic 12.1.1.0 ----->
Register for Security Updates:
----->

Provide your email address for security updates and to initiate configuration manager.

"Receive Security Update:" = [Enter new value or use default "yes"]
Enter [Yes][No]? no

<----- Oracle Installer - WebLogic 12.1.1.0 ----->

Register for Security Updates:
----->

Provide your email address for security updates and to initiate configuration manager.

"Receive Security Update:" = [Enter new value or use default "yes"]

** Do you wish to bypass initiation of the configuration manager and
** remain uninformed of critical security issues in your configuration?

Enter [Yes][No]? yes

<----- Oracle Installer - WebLogic 12.1.1.0 ----->

Register for Security Updates:
----->

Provide your email address for security updates and to initiate configuration manager.

1|Email:[]
2|Support Password:[]
3|Receive Security Update:[No]

Enter index number to select OR [Exit][Previous][Next]>
```

在这里，选择默认的组件。

```
<----- Oracle Installer - WebLogic 12.1.1.0 ----->

Register for Security Updates:
----- [Next] [Exit] [Previous] [Help] [Cancel]

Provide your email address for security updates and to initiate configuration manager.

1|Email: []
2|Support Password: []
3|Receive Security Update:[No]

Enter index number to select OR [Exit][Previous][Next]>
<----- Oracle Installer - WebLogic 12.1.1.0 ----->

Choose Install Type:
----- [Next] [Exit] [Previous] [Help] [Cancel]

Select the type of installation you wish to perform.

->1|Typical
| Install the following product(s) and component(s):
| - WebLogic Server
| - Oracle Coherence

2|Custom
| Choose software products and components to install and perform optional
| configuration.

Enter index number to select OR [Exit][Previous][Next]>
```

因为我们是在 jdk 目录下面使用 Java，所以 weblogic 记住了我们的 jdk 位置，选择使用默认的 jdk 位置，后面还需要选择产品安装目录，之后即开始安装过程。

```
<----- Oracle Installer - WebLogic 12.1.1.0 ----->

JDK Selection (Any * indicates Oracle Supplied VM):
----- [Next] [Exit] [Previous] [Help] [Cancel]

JDK(s) chosen will be installed. Defaults will be used in script string-
substitution if installed.

1|Add Local Jdk
2|/jdk[x]

*Estimated size of installation: 589.7 MB

Enter 1 to add or >= 2 to toggle selection OR [Exit][Previous][Next]>
<----- Oracle Installer - WebLogic 12.1.1.0 ----->
```

```
Choose Product Installation Directories:
-----
Middleware Home Directory: [/opt/Middleware]

Product Installation Directories:
1|WebLogic Server: [/opt/Middleware/wlserver_12.1]
2|Oracle Coherence: [/opt/Middleware/coherence_3.7]

Enter index number to select OR [Exit][Previous][Next]>
<----- Oracle Installer - WebLogic 12.1.1.0 ----->

The following Products and JDKs will be installed:
-----
WebLogic Platform 12.1.1.0
| ____ WebLogic Server
| | ____ Core Application Server
| | ____ Administration Console
| | ____ Configuration Wizard and Upgrade Framework
| | ____ Web 2.0 HTTP Pub-Sub Server
| | ____ WebLogic SCA
| | ____ WebLogic JDBC Drivers
| | ____ Third Party JDBC Drivers
| | ____ WebLogic Server Clients
| | ____ Xquery Support
| | ____ Evaluation Database
| ____ Oracle Coherence
| | ____ Coherence Product Files

*Estimated size of installation: 589.9 MB

Enter [Exit][Previous][Next]>
Oct 28, 2014 7:59:44 AM java.util.prefs.FileSystemPreferences$2 run
INFO: Created user preferences directory.

<----- Oracle Installer - WebLogic 12.1.1.0 ----->

Installing files..
0%          25%          50%          75%          100%
[-----|-----|-----|-----]                                     [*****]
[*****]                                     *****]

Performing String Substitutions...

<----- Oracle Installer - WebLogic 12.1.1.0 ----->

Configuring OCM...
```

```

----- Oracle Installer - WebLogic 12.1.1.0 -----
0% 25% 50% 75% 100%
[-----|-----|-----|-----] remaining, not executable
[*****]
----- Oracle Installer - WebLogic 12.1.1.0 -----
Installing Patches...
0% 25% 50% 75% 100%
[-----|-----|-----|-----] remaining, not executable
[*****]
----- Oracle Installer - WebLogic 12.1.1.0 -----
Creating Domains...
----- Oracle Installer - WebLogic 12.1.1.0 -----
Installation Complete
Congratulations! Installation is complete.
Press [Enter] to continue or type [Exit]>
----- Oracle Installer - WebLogic 12.1.1.0 -----
Clean up process in progress ...
完成以上步骤后，容器内已成功安装 Weblogic 服务。

```

### 3. 创建节点

接下来，笔者将创建默认的 weblogic 域和 Adminserver 节点：

同样使用 console 模式启动安装，并选择新建 weblogic 域。

```

root@185546d00925:/# cd /opt/Middleware/wlserver_12.1/common/bin/
root@185546d00925:/opt/Middleware/wlserver_12.1/common/bin# ./config.sh
-mode=console
----- Fusion Middleware Configuration Wizard -----
Welcome:
-----
Choose between creating and extending a domain. Based on your selection,
the Configuration Wizard guides you through the steps to generate a new or
extend an existing domain.
->1|Create a new WebLogic domain
    | Create a WebLogic domain in your projects directory.

```

2|Extend an existing WebLogic domain  
 | Use this option to add new components to an existing domain and modify  
 | configuration settings.

Enter index number to select OR [Exit][Next]>

<----- Fusion Middleware Configuration Wizard ----->

Select Domain Source:

Select the source from which the domain will be created. You can create the domain by selecting from the required components or by selecting from a list of existing domain templates.

->1|Choose Weblogic Platform components  
 | You can choose the Weblogic component(s) that you want supported in  
 | your domain.

2|Choose custom template  
 | Choose this option if you want to use an existing template. This could be a custom created template using the Template Builder.

Enter index number to select OR [Exit][Previous][Next]>

选择默认的 weblogic 组件。

<----- Fusion Middleware Configuration Wizard ----->

Application Template Selection:

Available Templates  
 | \_\_\_\_ Basic WebLogic Server Domain - 12.1.1.0 [wlserver\_12.1]x  
 | \_\_\_\_ Basic WebLogic SIP Server Domain - 12.1.1.0 [wlserver\_12.1] [2]  
 | \_\_\_\_ WebLogic Advanced Web Services for JAX-RPC Extension - 12.1.1.0  
 [wlserver\_12.1] [3]  
 | \_\_\_\_ WebLogic Advanced Web Services for JAX-WS Extension - 12.1.1.0  
 [wlserver\_12.1] [4]

Enter number exactly as it appears in brackets to toggle selection OR [Exit]  
 [Previous][Next]>

设置新建的域名：

<----- Fusion Middleware Configuration Wizard ----->

Edit Domain Information:

Name	Value
------	-------

```

_ | _____ | _____ |
1| *Name: | base_domain | [Exit] [Previous] [Next]>
Enter value for "Name" OR [Exit][Previous][Next]>

```

使用默认的安装位置：

```

----- Fusion Middleware Configuration Wizard -----
Select the target domain directory for this domain:
-----
"Target Location" = [Enter new value or use default
"/opt/Middleware/user_projects/domains"]
Enter new Target Location OR [Exit][Previous][Next]>

```

设置用户名和密码：

```

----- Fusion Middleware Configuration Wizard -----
Configure Administrator User Name and Password:
-----
Create a user to be assigned to the Administrator role. This user is the
default administrator used to start development mode servers.

|      Name          |      Value       |
| _____ | _____ |
1| *Name:           | weblogic      |
2| *User password: | [Exit] [Previous] [Next]>
3| *Confirm user password: | [Exit] [Previous] [Next]>
4| Description:    | This user is the default administrator. |
```

Use above value or select another option:

- 1 - Modify "Name"
- 2 - Modify "User password"
- 3 - Modify "Confirm user password"
- 4 - Modify "Description"

Enter option number to select OR [Exit][Previous][Next]> 3

```

----- Fusion Middleware Configuration Wizard -----
Configure Administrator User Name and Password:
-----
Create a user to be assigned to the Administrator role. This user is the
default administrator used to start development mode servers.

"Confirm user password:" = []

```

```

Enter new *Confirm user password: OR [Exit] [Reset] [Accept]> weblogic_test
<----- Fusion Middleware Configuration Wizard ----->

Configure Administrator User Name and Password:
-----
Create a user to be assigned to the Administrator role. This user is the
default administrator used to start development mode servers.



|   | Name                    | Value                                   |
|---|-------------------------|-----------------------------------------|
| 1 | *Name:                  | weblogic                                |
| 2 | *User password:         |                                         |
| 3 | *Confirm user password: | *****                                   |
| 4 | Description:            | This user is the default administrator. |



Use above value or select another option:
1 - Modify "Name"
2 - Modify "User password"
3 - Modify "Confirm user password"
4 - Modify "Description"
5 - Discard Changes

Enter option number to select OR [Exit] [Previous] [Next]> 2
<----- Fusion Middleware Configuration Wizard ----->

Configure Administrator User Name and Password:
-----
Create a user to be assigned to the Administrator role. This user is the
default administrator used to start development mode servers.

"User password:" = []

Enter new *User password: OR [Exit] [Reset] [Accept]> weblogic_test
<----- Fusion Middleware Configuration Wizard ----->

Configure Administrator User Name and Password:
-----
Create a user to be assigned to the Administrator role. This user is the
default administrator used to start development mode servers.



|   | Name   | Value    |
|---|--------|----------|
| 1 | *Name: | weblogic |


```

```

2| *User password: | ***** | ****
3| *Confirm user password: | * | ****
4| Description: | This user is the default administrator. | ****

```

Use above value or select another option:

- 1 - Modify "Name"
- 2 - Modify "User password"
- 3 - Modify "Confirm user password"
- 4 - Modify "Description"
- 5 - Discard Changes

Enter option number to select OR [Exit] [Previous] [Next]>

根据授权，使用开发模式或生产模式：

<----- Fusion Middleware Configuration Wizard ----->

Domain Mode Configuration:

Enable Development or Production Mode for this domain.

- >1|Development Mode
- 2|Production Mode

Enter index number to select OR [Exit] [Previous] [Next]> 2

<----- Fusion Middleware Configuration Wizard ----->

Java SDK Selection:

- >1|Sun SDK 1.6.0\_43 @ /jdk
- 2|Other Java SDK

Enter index number to select OR [Exit] [Previous] [Next]>

选择只安装一个 administration server，其他服务的安装跟这个一样：

<----- Fusion Middleware Configuration Wizard ----->

Select Optional Configuration:

- 1|Administration Server [ ]
- 2|Managed Servers, Clusters and Machines [ ]
- 3|RDBMS Security Store [ ]

Enter index number to select OR [Exit] [Previous] [Next]> 1

```
<----- Fusion Middleware Configuration Wizard ----->
----- Select Optional Configuration: [1] Administration [2] Managed Servers, Clusters and Machines [3] RDBMS Security Store [ ] -----  

1|Administration Server [x] 2|Managed Servers, Clusters and Machines [ ] 3|RDBMS Security Store [ ]  

-----  

Enter index number to select OR [Exit][Previous][Next]>  

<----- Fusion Middleware Configuration Wizard ----->  

Configure the Administration Server:  

-----  

Each WebLogic Server domain must have one Administration Server. The Administration Server is used to perform administrative tasks.  


|   | Name             | Value               |
|---|------------------|---------------------|
| 1 | *Name:           | AdminServer         |
| 2 | *Listen address: | All Local Addresses |
| 3 | Listen port:     | 7001                |
| 4 | SSL listen port: | N/A                 |
| 5 | SSL enabled:     | false               |

Use above value or select another option:  

1 - Modify "Name"  

2 - Modify "Listen address"  

3 - Modify "Listen port"  

4 - Modify "SSL enabled"  

Enter option number to select OR [Exit][Previous][Next]>  

<----- Fusion Middleware Configuration Wizard ----->  

Creating Domain...  

0% 25% 50% 75% 100%  

[-----|-----|-----|-----|-----]  

[*****|*****|*****|*****|*****]  

**** Domain Created Successfully! ****
```

#### 4. 配置 Weblogic

首先，修改 Weblogic 的一些环境变量：

```
root@185546d00925:/opt/Middleware/user_projects/domains/base_domain# vi bin/
setDomainEnv.sh
```

使用用户名和密码启动一次Weblogic之后，会在 /opt/Middleware/user\_projects/domains/base\_domain 下面生成一个 server 的文件夹的 AdminServer 文件夹。

创建 security/boot.properties 文件，内容如下：

```
username=weblogic
password=password
```

这样，再次启动 Weblogic 时，就不需要输入密码了。通过 console 的输出可以看到 Weblogic 在容器内启动成功，并监听到 7001 端口：

```
root@185546d00925:/opt/Middleware/user_projects/domains/base_domain# ./startWebLogic.sh
.
.
.
JAVA Memory arguments: -Xms256m -Xmx512m -XX:MaxPermSize=256m
.
.
WLS Start Mode=Production
# 此处省去了一些启动过程
.
.
<Oct 28, 2014 8:13:52 AM UTC> <Notice> <Server> <BEA-002613> <Channel "Default" is now listening on 172.17.0.13:7001 for protocols iiop, t3, ldap, snmp, http.>
<Oct 28, 2014 8:13:52 AM UTC> <Notice> <Server> <BEA-002613> <Channel "Default[2]" is now listening on 127.0.0.1:7001 for protocols iiop, t3, ldap, snmp, http.>
<Oct 28, 2014 8:13:52 AM UTC> <Warning> <Server> <BEA-002611> <The hostname "localhost", maps to multiple IP addresses: 127.0.0.1, 0:0:0:0:0:0:1:>
<Oct 28, 2014 8:13:52 AM UTC> <Notice> <Server> <BEA-002613> <Channel "Default[3]" is now listening on 0:0:0:0:0:0:1:7001 for protocols iiop, t3, ldap, snmp, http.>
<Oct 28, 2014 8:13:52 AM UTC> <Notice> <WebLogicServer> <BEA-000329> <Started the WebLogic Server Administration Server "AdminServer" for domain "base_domain" running in production mode.>
<Oct 28, 2014 8:13:52 AM UTC> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to RUNNING.>
<Oct 28, 2014 8:13:52 AM UTC> <Notice> <WebLogicServer> <BEA-000360> <The server started in RUNNING mode.>
```

## 5. Dockerfile

编写 Dockerfile，内容为：

```
FROM weblogic_1
# 设置继承自我们创建的 weblogic_1 镜像
MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)
```

```
# 下面是一些创建者的基本信息
# 设置环境变量，所有操作都是非交互式的
ENV DEBIAN_FRONTEND noninteractive

RUN echo "Asia/Shanghai" > /etc/timezone && \
    dpkg-reconfigure -f noninteractive tzdata

# 注意这里要更改系统的时区设置，因为在 Web 应用中经常会用到时区这个系统变量，默认的 ubuntu 会让你的应用程序发生不可思议的效果哦

COPY jdk /jdk
COPY run.sh /run.sh

RUN chmod +x /run.sh

EXPOSE 7001

CMD ["/run.sh"]
```

## 6. run.sh

编写 run.sh 脚本，内容为：

```
#!/bin/bash
/usr/sbin/sshd -D &
/opt/Middleware/user_projects/domains/base_domain/startWebLogic.sh
```

## 7. 创建镜像及测试

使用 docker build 命令创建镜像，命名为 weblogic:jk1.6：

```
$ sudo docker build -t weblogic:jk1.6 .
Sending build context to Docker daemon 1.256 GB
Sending build context to Docker daemon
Step 0 : FROM weblogic_1
--> 6b73c466305f
Step 1 : MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)
--> Using cache
--> c93c4f458baa
Step 2 : ENV DEBIAN_FRONTEND noninteractive
--> Using cache
--> 18fef76dc41b
Step 3 : RUN echo "Asia/Shanghai" > /etc/timezone && dpkg-reconfigure -f
noninteractive tzdata
--> Using cache
--> d307ba8bd052
Step 4 : COPY jdk /jdk
--> Using cache
```

```

--> a54ffa93184
Step 5 : COPY run.sh /run.sh
--> a21b0ccb0b07
Removing intermediate container eabf04e467dd
Step 6 : RUN chmod +x /run.sh
--> Running in 4231d8062f5b
--> f46358aac2e3
Removing intermediate container 4231d8062f5b
Step 7 : EXPOSE 7001
--> Running in d17c80a9ea5b
--> 8bc8dd5c8caa
Removing intermediate container d17c80a9ea5b
Step 8 : CMD /run.sh
--> Running in 45de5ac8883a
--> d904fe4f91f9
Removing intermediate container 45de5ac8883a
Successfully built d904fe4f91f9

```

启动一个容器，并查看它的映射端口：

```

$ sudo docker run -d -P weblogic:jdk1.6
6c08f5b110affaec256e48b925a1914991c931a8c581f4817fbc5d538e7af2e6
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
6c08f5b110af        weblogic:jdk1.6      "/run.sh"          15 seconds ago   Up 14 seconds        0.0.0.0:49163->22/tcp, 0.0.0.0:49164->7001/tcp

```

使用浏览器登录 Weblogic 控制台，如图 11-5 所示。

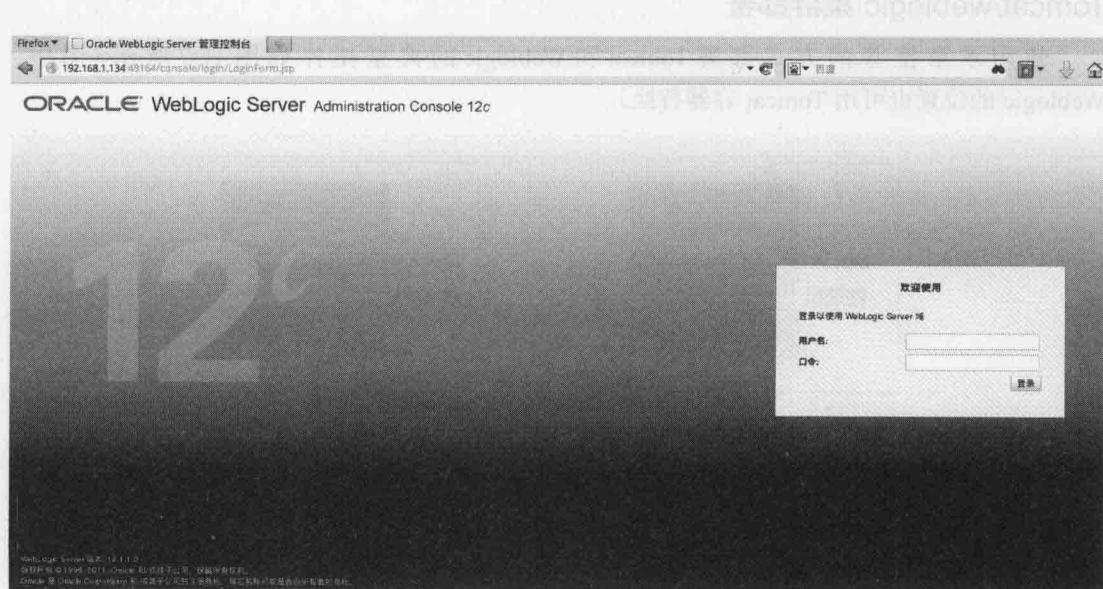


图 11-5 登录 Weblogic 控制台

架构师在控制台设置完启动程序、数据源等参数（如图 11-6 所示）之后，可以重复使用以上步骤创建适合项目运行的 Weblogic 镜像。

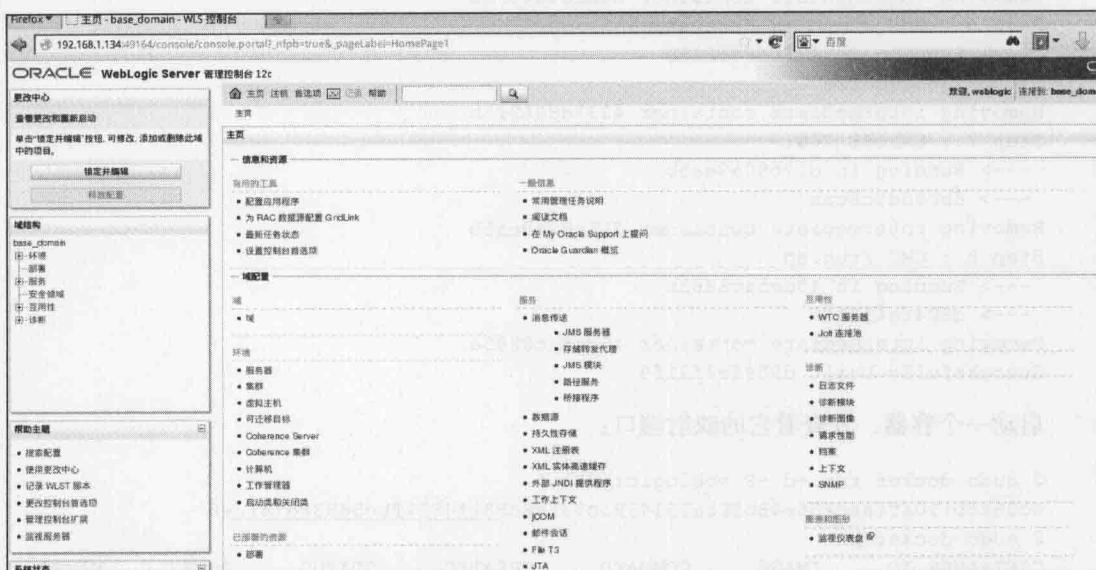


图 11-6 Weblogic 控制台配置界面

## Tomcat/weblogic 集群部署

使用本章推荐的方式部署 Tomcat 和 Weblogic 的典型拓扑如图 11-7 所示（图中，Weblogic 的位置也可用 Tomcat 容器替换）。

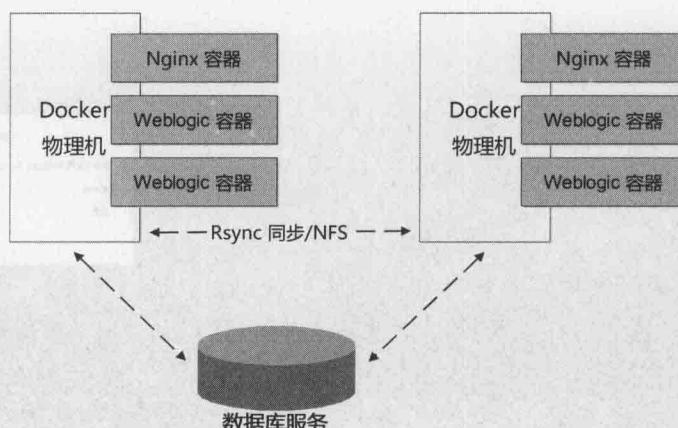


图 11-7 Tomcat/Weblogic 集群部署

## 11.5 LAMP

LAMP 指的 Linux (操作系统)、ApacheHTTP 服务器、MySQL (有时也指 MariaDB, 数据库软件) 和 PHP (有时也是指 Perl 或 Python) 的组合方案, 一般很适合用来建立 Web 服务器环境。

下面介绍如何使用 Docker 来搭建一个包含 LAMP 组件的容器。

### 11.5.1 下载 LAMP 镜像

搜索 Docker Hub 上被收藏或使用较多的 LAMP 镜像, 笔者推荐选择 tutum/lamp 镜像:

```
$ sudo docker search -s 10 lamp
NAME      DESCRIPTION      STARS      OFFICIAL      AUTOMATED
tutum/lamp  LAMP image - Apache listens in port 80, an...  31          [OK]
```

执行 docker pull 命令, 下载镜像:

```
$ sudo docker pull tutum/lamp
Pulling repository tutum/lamp
4b32789c7d66: Download complete
...
```

### 11.5.2 使用默认方式启动 LAMP 容器

利用下载的镜像启动一个容器, 并映射容器的 8080 端口和 3306 端口:

```
$ sudo docker run -d -p 8080:80 -p 3306:3306 tutum/lamp
0ee00c97a5cdefb985baf826c16723f333aa5edddee4072a5107c724ad09f10d
$ docker ps
CONTAINER ID        IMAGE           COMMAND       CREATED          STATUS          PORTS          NAMES
0ee00c97a5cd        tutum/lamp:latest     "/run.sh"    3 seconds ago   Up 2
seconds            0.0.0.0:3306->3306/tcp, 0.0.0.0:8080->80/tcp   lonely_davinci
```

使用 curl 命令测试, 可以查看到默认的应用已经启动:

```
$ curl http://127.0.0.1:8080
```

返回的内容如下:

```
<html>
<head>
  <title>Hello world!</title>
  <style>
    body {
      background-color: white;
      text-align: center;
      padding: 50px;
      font-family: "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
    }
  </style>
</head>
<body>
  Hello world!
</body>
</html>
```

```

        #logo {
            margin-bottom: 40px;
        }
    </style>
</head>
<body>
    
    <h1>Hello world!</h1>
    <h2>MySQL Server version: 5.5.38-0ubuntu0.14.04.1</h2>
</body>
</html>

```

### 11.5.3 部署自己的 PHP 应用

默认的容器启动了一个 helloworld 应用。读者可以基于此镜像，编辑 Dockerfile 来创建自定义 LAMP 应用镜像。

在宿主主机上创建新的工作目录 lamp：

```

$ mkdir lamp
$ cd lamp
$ touch Dockerfile

```

在 php 目录下里面创建 **Dockerfile** 文件，内容为：

```

FROM tutum/lamp:latest
RUN rm -fr /app && git clone https://github.com/username/customapp.git /app
# 这里将 https://github.com/username/customapp.git 地址替换为你自己的项目地址
EXPOSE 80 3306
CMD ["/run.sh"]

```

创建镜像，命名为 dockerpool/my-lamp-app：

```
$ docker build -t dockerpool/my-lamp-app .
```

利用新创建镜像启动容器，注意启动时候指定 -d 参数，让容器后台运行：

```
$ docker run -d -p 8080:80 -p 3306:3306 dockerpool/my-lamp-app
```

在本地主机上使用 curl 看一下自己的应用程序是不是已经正确启动：

```
$ curl http://127.0.0.1:8080/
```

### 11.5.4 在 PHP 程序中连接数据库

#### 1. 在容器中访问 MySQL 数据库

下载的 tutum/lamp 镜像中的 MySQL 数据库已带有默认的 root 用户，本地连接可以不使

用密码，所以在代码中访问数据库的实现非常简单：

```
<?php
$mysql = new mysqli("localhost", "root");
echo "MySQL Server info: ".$mysql->host_info;
?>
```

## 2. 在容器外访问 MySQL 数据库

默认的 MySQL 数据库不支持 root 用户远程登录，因此在容器外无法直接通过 root 用户访问 MySQL 数据库。

当第一次使用 tutum/lamp 镜像启动容器的时候，它会自动创建一个叫 admin 的 MySQL 用户，并生成一个随机密码，使用 docker logs 命令可以获取到这个密码：

```
$ sudo docker logs 9cb
=> An empty or uninitialized MySQL volume is detected in /var/lib/mysql
=> Installing MySQL ...
=> Done!
=> Waiting for confirmation of MySQL service startup
=> Creating MySQL admin user with random password
=> Done!
=====
You can now connect to this MySQL Server using:

mysql -uadmin -p2Ijg6gvmM0N3 -h<host> -P<port>

Please remember to change the above password as soon as possible!
MySQL user 'root' has no password but only allows local connections
=====
```



**注意** admin 用户具有 root 相同的权限。

## 11.6 CMS

内容管理系统（Content Management System, CMS）指的是提供内容编辑服务的平台程序。CMS 可以让不懂编程的普通人方便又轻松地发布、更改和管理各类数字内容（主要以文本和图像为主）。

下面，笔者将以 WordPress 为例介绍如何使用 Docker 运行 CMS。

## WordPress 简介

WordPress 是风靡全球的免费开源的内容管理系统。WordPress 是博客、企业官网、产品首页等内容相关平台的主流实现方案之一，除 WordPress 之外还有 Drupal、Joomla、Typo3 等 CMS 系统。

WordPress 基于 PHP 和 MySQL，架构设计简单明了，可以方便地制作主题、插件和各种功能模块。更重要的是，WordPress 的社区非常庞大，在线资源非常丰富，并且在各大网络空间商和云平台中受到广泛的支持。根据 2013 年 8 月的统计数据，流量排名前一千万的网站中有 22% 使用了 WordPress 系统。



## 使用官方镜像

首先，通过 Docker Hub 下载官方 WordPress 镜像：

```
$ sudo docker pull wordpress
```

然后，就可以创建并运行一个 WordPress 容器，并连接到 mysql 容器：

```
$ sudo docker run --name some-wordpress --link some-mysql:mysql -d wordpress
```

启动容器时可以带以下环境参数：

- ❑ -e WORDPRESS\_DB\_USER=... 设置 WordPress 的数据库用户名，默认是“root”。
- ❑ -e WORDPRESS\_DB\_PASSWORD=... 设置 WordPress 的数据库密码，默认值是连接至此 WordPress 容器的 MySQL 容器的 MYSQL\_ROOT\_PASSWORD 环境变量的值。
- ❑ -e WORDPRESS\_DB\_NAME=... 设置 WordPress 所使用的数据库的名称，默认是“wordpress”。
- ❑ -e WORDPRESS\_AUTH\_KEY=..., -e WORDPRESS\_SECURE\_AUTH\_KEY=...,  
     -e WORDPRESS\_LOGGED\_IN\_KEY=..., -e WORDPRESS\_NONCE\_KEY=..., -e  
     WORDPRESS\_AUTH\_SALT=..., -e WORDPRESS\_SECURE\_AUTH\_SALT=..., -e  
     WORDPRESS\_LOGGED\_IN\_SALT=..., -e WORDPRESS\_NONCE\_SALT=... 加密盐  
     和随机串，默认值是随机的 SHA1 值。

如果 WORDPRESS\_DB\_NAME 指定的数据库在 MySQL 容器中不存在，那么此镜像会使用 WORDPRESS\_DB\_USER 用户自动创建一个同名数据库。

同样，用户可以使用 -p 参数来进行端口映射：

```
$ sudo docker run --name some-wordpress --link some-mysql:mysql -p 8080:80 -d wordpress
```

此时，可在浏览器中访问 <http://localhost:8080> 来打开 WordPress 页面。

WordPress 官方镜像的更多信息可以参考 [https://registry.hub.docker.com/\\_/wordpress/](https://registry.hub.docker.com/_/wordpress/)。

## 11.7 本章小结

本章首先介绍了 Apache 和 Nginx 两种比较流行的 Web 服务器镜像的创建，其中 Nginx 还介绍了淘宝的衍生版本。

但是，我们介绍的安装、配置、编译的方法都是依据一些比较常见的需求，如果读者有其他需求（比如 Apache、Nginx 需要编译新的功能模块），应该根据自己需求来重新定制镜像，特别是在生产环境中，一些细微的参数配置可能带来性能上巨大的变化。

本章的 2.3.3 和 2.3.4 小节介绍了目前比较流行的 Java 中间件服务器 Tomcat 和 Weblogic 的镜像创建。

笔者一直认为，中间件服务器是 Docker 容器应用的最佳实践，原因如下：

- 中间件服务器是除数据库服务器外的主要计算节点，很容易成为性能瓶颈，所以通常需要大批量部署，而 Docker 对于批量部署有着许多先天的优势（详见本书第一部分内容）。
- 中间件服务器结构清晰，在剥离了配置文件、日志、代码目录之后，容器几乎可以处于零增长状态，这使得容器的迁移和批量部署更加方便。
- 中间件服务器很容易实现集群，在使用硬件的 F5，软件的 Nginx 等负载均衡后，中间件服务器集群变得非常容易。

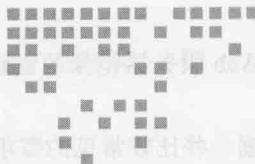
值得注意的是，由于目前 Docker 暂不支持修改运行中的容器的一些配置，比如无法为运行中的容器映射更多的宿主主机目录，无法为运行中的主机映射更多宿主主机的网络，等等，读者在使用中间件容器的时候，需要事先规划好容器的用途和可能开放的网络端口等资源。

本章的最后两个小节介绍了比较常见的 LAMP 套件和 WordPress 镜像的创建。其在开发环境中使用非常方便，但是生产环境中因为性能和其他方面的一些考虑通常会有专门的 Web 和数据库服务器。

需要特别注意的是，对于程序代码、程序的资源目录、日志、数据库文件等需要实时更新的数据一定要通过 -v 参数映射到宿主主机的目录中来，使用 Docker 的 AUFS 文件格式，会产生较大的性能问题（原理部分请参考本书的第一部分和第三部分）。

IBM 研究院也针对 Docker 的各项性能做了比较详细的测试，可以从这里下载报告：

[http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/\\$File/rc25482.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf)。



Chapter 12

## 第 12 章 数据库应用

在本章中，将通过 MySQL、Oracle 和 MongoDB 三个主流的开源数据库，来介绍如何使用 Docker 容器化它们。首先，将通过 MySQL 的安装和配置，来学习如何使用 Docker 容器化 MySQL。接着，将通过 Oracle 和 MongoDB 的安装和配置，来学习如何使用 Docker 容器化它们。

主流数据库方案包括关系数据库（SQL）和非关系数据库（NoSQL）方案。

关系数据库是建立在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据，支持复杂的事物处理和结构化查询。目前流行的关系型数据库有 MySQL、Oracle、PostgreSQL、MariaDB、SQLServer 等等。

非关系数据库是新兴的数据库技术，它放弃了传统关系型数据库的部分强一致性限制，使其更适用于需要大规模的并行处理的生产环境，并能在这些场景下发挥出优异的性能。NoSQL 是关系型数据库的良好补充，代表产品有 MongoDB、Redis、CouchDB 等。

本章选取了最具代表性的 MySQL、Oracle、MongoDB 三款数据库，来展示基于 Docker 创建相关镜像并进行应用的过程。

### 12.1 MySQL

MySQL 是流行的开源关系数据库实现，因为其高性能、可靠性和适应性而得到广泛应用和关注。

#### 1. 下载文件

从 GitHub Dockerpool 社区下载 MySQL 镜像项目：

```
$ git clone https://github.com/DockerPool/mysql.git
Cloning into 'mysql'...
remote: Counting objects: 13, done.
remote: Compressing objects: 100% (13/13), done.
```

```
remote: Total 13 (delta 1), reused 8 (delta 0)
Unpacking objects: 100% (13/13), done.
Checking connectivity... done.
```

查看内容，包括已经写好的 Dockerfile 和若干脚本：

```
$ cd mysql
$ ls
create_db.sh  Dockerfile  import_sql.sh  LICENSE  my.cnf  mysqld_charset.cnf
README.md    run.sh
```

其中 Dockerfile 内容为：

```
# 本文件参考了 tutum 的 Dockerfile
FROM sshd
MAINTAINER Waitfish <dwj_zz@163.com>

# 安装软件
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && \
    apt-get -yq install mysql-server-5.6 pwgen && \
    rm -rf /var/lib/apt/lists/*

# 删除预安装的数据库文件
RUN rm -rf /var/lib/mysql/*

# 添加文件夹下的 MYSQL 配置文件
ADD my.cnf /etc/mysql/conf.d/my.cnf
ADD mysqld_charset.cnf /etc/mysql/conf.d/mysqld_charset.cnf

# 添加 MYSQL 的脚本
ADD import_sql.sh /import_sql.sh
ADD run.sh /run.sh
RUN chmod 755 /*.sh

# 设置环境变量，用户名以及秘密
ENV MYSQL_USER admin
ENV MYSQL_PASS **Random**

# 设置主从复制模式下的环境变量
ENV REPLICATION_MASTER **False**
ENV REPLICATION_SLAVE **False**
ENV REPLICATION_USER replica
ENV REPLICATION_PASS replica

# 设置可以允许挂载的卷，可以用来备份数据库和配置文件
VOLUME  ["/etc/mysql", "/var/lib/mysql"]

# 设置可以映射的端口，如果是从我们的 sshd 镜像继承的话，默认还会开启 22 端口
EXPOSE 3306
CMD ["/run.sh"]
```

## 2. 创建镜像

使用 docker build 命令来创建镜像 mysql:latest:

```
$ sudo docker build -t mysql:latest .
Sending build context to Docker daemon 95.23 kB
Sending build context to Docker daemon
Step 0 : FROM sshd
--> 312c93647dc3
Step 1 : MAINTAINER Waitfish <dwj_zz@163.com>
--> Running in a149f8a7933f
--> edbbfe8b4895
Removing intermediate container a149f8a7933f
Step 2 : ENV DEBIAN_FRONTEND noninteractive
--> Running in e80cbb29cadb
--> 81fc6101a236
Removing intermediate container e80cbb29cadb
Step 3 : RUN apt-get update && apt-get -yq install mysql-server-5.6 pwgen &&
rm -rf /var/lib/apt/lists/*
--> Running in 5d220fe833c2
...
Removing intermediate container 3c3254e8cc1e
Successfully built f008f97bdc14
dwj@iZ23pzn1je4Z:~/mysql$ sudo docker images
REPOSITORY      TAG          IMAGE ID      CREATED        VIRTUAL SIZE
mysql           latest       f008f97bdc14    About a minute ago   539.1 MB
```

## 3. 使用镜像

使用默认方式启动后台容器，不添加环境变量，并使用 -P 参数自动映射容器的 22 和 3306 端口。

```
$ sudo docker run -d -P mysql
```

检查容器进程启动情况和端口映射情况，可见容器的 22 端口被映射到本地的 49153 端口。

```
$ sudo docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
eef1632ccd4e    mysql:latest "/run.sh"   8 seconds ago  Up 8 seconds
0.0.0.0:49153->22/tcp, 0.0.0.0:49154->3306/tcp    angry_einstein
```

通过映射的本地 49153 端口 SSH 登录容器，并查看运行的进程。

```
$ ssh 127.0.0.1 -p 49153
The authenticity of host '[127.0.0.1]:49153 ([127.0.0.1]:49153)' can't be
established.
ECDSA key fingerprint is db:35:7a:60:2d:11:d5:97:5a:e6:84:a6:95:f0:4f:32.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[127.0.0.1]:49153' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.2.0-54-generic x86_64)
```

\* Documentation: <https://help.ubuntu.com/>

The programs included with the Ubuntu system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/\*/\*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by  
applicable law.

```
root@eef1632ccd4e:~# ps -ef |grep mysql
root      1      0 20:14 ?        00:00:00 /bin/sh /usr/bin/mysqld_safe
mysql    1974      1 20:14 ?        00:00:00 /usr/sbin/mysqld --basedir=/usr
--datadir=/var/lib/mysql --plugin-dir=/usr/lib/mysql/plugin --user=mysql --log-
error=/var/log/mysql/error.log --pid-file=/var/run/mysqld/mysqld.pid --socket=/
var/run/mysqld/mysqld.sock --port=3306
root     2022  2010  0 20:15 pts/0    00:00:00 grep --color=auto mysql
```

默认情况下，容器内的 MySQL 提供了 root 账号和 admin 账号，其中 root 账号无需密码，但只允许本地访问。

```
mysql> select host, user, password from mysql.user;
+-----+-----+
| host      | user   | password          |
+-----+-----+
| localhost | root   |                    |
| eef1632ccd4e | root   |                    |
| 127.0.0.1 | root   |                    |
| ::1       | root   |                    |
| localhost |       |                    |
| eef1632ccd4e |       |                    |
| %         | admin  | *ADDD6793DD97A040C9B039F72682E5AA31A92C35 |
+-----+-----+
7 rows in set (0.00 sec)
```

admin 账号拥有远程访问权限。其密码可以使用 docker logs 命令来查看获取：

```
$ sudo docker logs eef
=> An empty or uninitialized MySQL volume is detected in /var/lib/mysql
=> Installing MySQL ...
=> Done!
=> Creating admin user ...
=> Waiting for confirmation of MySQL service startup, trying 0/13 ...
=> Creating MySQL user admin with random password
=> Done!
=====
You can now connect to this MySQL Server using:
```

```
mysql -uadmin -pt1FWuDCgQicT -h<host> -P<port>
```

```
Please remember to change the above password as soon as possible!
MySQL user 'root' has no password but only allows local connections
=====
141106 20:14:21 mysqld_safe Can't log to error log and syslog at the same time.
Remove all --log-error configuration options for --syslog to take effect.
141106 20:14:21 mysqld_safe Logging to '/var/log/mysql/error.log'.
141106 20:14:21 mysqld_safe Starting mysqld daemon with databases from /var/
lib/mysql
```

上面的 t1FWuDCgQicT 就是 admin 的密码。

#### 4. 指定 admin 账号用户名和密码

用户也可以在启动容器时指定 admin 账号的用户名和密码，例如：

```
$ sudo docker run -d -P -e MYSQL_PASS="mypass" mysql
1b32444ebb7232f885961faa15fb1a052ca93b81c308cc41d16bd3d276c77d75
```

#### 5. 挂载目录到容器

默认情况下数据库的数据库文件和日志文件都会存在容器的 AUFS 文件层，这不仅会使得容器变得越来越臃肿，不便于迁移、备份等管理，而且数据库的性能也会受到影响。因此，建议挂载本地主机的目录到容器内，例如：

```
$ docker run -d -P -v /opt/mysqldb:/var/lib/mysql mysql
```

这样，容器就会将数据文件和日志文件都放到指定的本地主机目录下面：

```
$ tree /opt/mysqldb/
/opt/mysqldb/
|-- auto.cnf
|-- ib_logfile0
|-- ib_logfile1
|-- ibdata1
|-- mysql
|   |-- columns_priv.MYD
|   |-- columns_priv.MYI
|   |-- columns_priv.frm
|   |-- db.MYD
|   |-- db.MYI
|   |-- db.frm
|   |-- event.MYD
|   |-- event.MYI
|   |-- event.frm
|   |-- func.MYD
|   |-- func.MYI
|   |-- func.frm
|   |-- general_log.CSM
...
...
```

## 6. 启用主从模式

利用主从模式，可以为数据库提供更好的可靠性。

首先，创建一个名称为 mysql 的主容器：

```
$ sudo docker run -d -e REPLICATION_MASTER=true -P --name mysql mysql
```

创建从容器，并连接到刚刚创建的主容器：

```
$ sudo docker run -d -e REPLICATION_SLAVE=true -P --link mysql:mysql mysql
```

注意，这里的主 mysql 服务器的名字必须为 mysql，否则会收到错误提示：

'Cannotconfigure slave, please link it to another MySQL container with alias as 'mysql'.

查看容器互联信息：

```
# sudo docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED      STATUS      PORTS      NAMES
a781d1c74024        mysql:latest "/run.sh"   About a minute ago   Up About a minute   0.0.0.0:49167->22/tcp, 0.0.0.0:49168->3306/tcp   romantic_fermi
38c73b5555aa        mysql:latest "/run.sh"   About a minute ago   Up About a minute   0.0.0.0:49165->22/tcp, 0.0.0.0:49166->3306/tcp   mysql
```

现在，就可以通过相应的端口来直接连接主或者从 MySQL 服务器了。

## 12.2 Oracle XE

Oracle 快捷版（Oracle XE）是一款基于 Oracle 11g 第 2 版代码库的小型入门级数据库，它具备以下优点：

- 免费开发、部署和分发。
- 下载速度快。
- 管理简单。

作为一款优秀的入门级数据库，它适合以下用户使用：

- 致力于 PHP、Java、.NET、XML 和开源应用程序的开发人员。
- 需要免费的入门级数据库进行培训和部署的 DBA。
- 需要入门级数据库进行免费分发的独立软件供应商（ISV）和硬件供应商。
- 需要在课程中使用免费数据库的教育机构和学生。

Oracle Database XE 对安装主机的规模和 CPU 数量不作限制（每台计算机一个数据库），但 XE 将最多存储 11 GB 的用户数据，同时最多使用 1 GB 内存和主机上的一个 CPU。

### 1. 搜索 Oracle 镜像

直接在 DockerHub 上搜索镜像，并下载 wnameless/oracle-xe-11g 镜像：

```
$ sudo docker search -s 10 oracle
NAME      DESCRIPTION      STARS      OFFICIAL      AUTOMATED
wnameless/oracle-xe-11g  SYS & SYSTEM password: oracle https://inde...    24      [OK]
alexeiled/docker-oracle-xe-11g This is a spin off from wnameless/docker-o...    21      [OK]
$ sudo docker pull wnameless/oracle-xe-11g
```

## 2. 启动和使用容器

启动容器，并分别映射 22 和 1521 端口到本地的 49160 和 49161 端口。

```
$ sudo docker run -d -p 49160:22 -p 49161:1521 wnameless/oracle-xe-11g
```

使用下列参数可以连接 oracle 数据库：

```
hostname: localhost
port: 49161
sid: xe
username: system
password: oracle
Password for SYS
```

使用 SSH 登录容器，默认的用户名为 root，密码为 admin。

```
$ ssh root@localhost -p 49160
password: admin
```

## 12.3 MongoDB

MongoDB 是一款可扩展、高性能的开源文档（Document-Oriented）数据库。它采用 C++ 开发，支持复杂的数据类型和强大的查询语言，提供了关系数据库的绝大部分功能。MongoDB 由于其高性能、易部署、易使用等特点，已经在各种领域都得到了广泛的应用。

### 1. 下载文件

从 GitHub Dockerpool 社区帐户下载 Mongodbs 镜像项目：

```
$ git clone https://github.com/DockerPool/Mongodb.git
```

查看内容，包括写好的 Dockerfile 和若干脚本等：

```
$ cd Mongodb
$ ls
Dockerfile  LICENSE  README.md  run.sh  set_mongodb_password.sh
```

其中 Dockerfile 内容为：

```
# 设置从我们之前创建的 sshd 镜像继承。
FROM sshd
```

```

MAINTAINER waitfish from dockerpool.com(dwj_zz@163.com)

RUN apt-get update && \
    apt-get install -y mongodb pwgen && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# 创建 mongodb 存放数据文件的文件夹
RUN mkdir -p /data/db
VOLUME /data/db

ENV AUTH yes

# 添加脚本
ADD run.sh /run.sh
ADD set_mongodb_password.sh /set_mongodb_password.sh
RUN chmod 755 ./*.sh

EXPOSE 27017
EXPOSE 28017

CMD ["/run.sh"]

```

set\_mongodb\_password.sh 脚本主要负责配置数据库的用户名和密码，内容为：

```

#!/bin/bash
# 这个脚本主要是用来设置数据库的用户名和密码。

# 判断是否已经设置过密码。
if [ -f /.mongodb_password_set ]; then
    echo "MongoDB password already set!"
    exit 0
fi

/usr/bin/mongod --smallfiles --nojournal &

PASS=${MONGODB_PASS:-$(pwgen -s 12 1)}
_word=$( [ ${MONGODB_PASS} ] && echo "preset" || echo "random" )

RET=1
while [[ RET -ne 0 ]]; do
    echo "=> Waiting for confirmation of MongoDB service startup"
    sleep 5
    mongo admin --eval "help" >/dev/null 2>&1
    RET=$?
done

# 通过 docker logs + id 可以看到下面的输出。
echo "=> Creating an admin user with a ${_word} password in MongoDB"
mongo admin --eval "db.addUser({user: 'admin', pwd: '$PASS', roles: [ 'userAdminAnyDatabase', 'dbAdminAnyDatabase' ]});"
mongo admin --eval "db.shutdownServer();"

```

```

echo "=> Done!"
touch /.mongodb_password_set

echo "=====
echo "You can now connect to this MongoDB server using:"
echo ""
echo "    mongo admin -u admin -p $PASS --host <host> --port <port>"
echo ""
echo "Please remember to change the above password as soon as possible!"
echo
"=====

```

run.sh 脚本是主要的启动脚本，内容为：

```

#!/bin/bash
if [ ! -f /.mongodb_password_set ]; then
    /set_mongodb_password.sh
fi

if [ "$AUTH" == "yes" ]; then
    # 这里读者可以自己设定 Mongodb 的启动参数。
    export mongodb='/usr/bin/mongod --nojournal --auth --httpinterface --rest'
else
    export mongodb='/usr/bin/mongod --nojournal --httpinterface --rest'
fi

if [ ! -f /data/db/mongod.lock ]; then
    eval $mongodb
else
    export mongodb=$mongodb' --dbpath /data/db'
    rm /data/db/mongod.lock
    mongod --dbpath /data/db --repair && eval $mongodb
fi

```

## 2. 创建镜像

根据 Dockerfile 创建镜像 `mongodb:latest`:

```

$ sudo docker build -t mongodb .
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
mongodb        latest    e3200a24cf28    3 hours ago   256 MB

```

## 3. 使用示范

启动后台容器，并分别映射 27017 和 28017 端口到本地：

```
$ sudo docker run -d -p 27017:27017 -p 28017:28017 mongodb
```

通过 `docker logs` 来查看默认的 `admin` 帐户密码：

```
$ sudo docker logs sa9
=====
You can now connect to this MongoDB server using:

    mongo admin -u admin -p 5elsT6KtjrqV --host <host> --port <port>

Please remember to change the above password as soon as possible!
=====
```

输出中的 5elsT6KtjrqV 就是 admin 用户的密码：

还可以利用环境变量在容器启动时指定密码：

```
$ sudo docker run -d -p 27017:27017 -p 28017:28017 -e MONGODB_PASS="mypass" mongodb
```

甚至，设定不需要密码：

```
$ sudo docker run -d -p 27017:27017 -p 28017:28017 -e AUTH=no mongodb
```

同样，读者也可以使用 -v 参数来映射本地目录到容器。

#### 4. 详细启动参数

Mongodb 的启动参数有很多，包括：

```
--quiet      # 安静输出
--port arg    # 指定服务端口号，默认端口 27017
--bind_ip arg  # 绑定服务 IP，若绑定 127.0.0.1，则只能本机访问，不指定默认本地所有 IP
--logpath arg   # 指定 MongoDB 日志文件，注意是指定文件不是目录
--logappend     # 使用追加的方式写日志
--pidfilepath arg  # PID File 的完整路径，如果没有设置，则没有 PID 文件
--keyFile arg    # 集群的私钥的完整路径，只对于 Replica Set 架构有效
--unixSocketPrefix arg  # UNIX 域套接字替代目录，(默认为 /tmp)
--fork        # 以守护进程的方式运行 MongoDB，创建服务器进程
--auth        # 启用验证
--cpu         # 定期显示 CPU 的 CPU 利用率和 iowait
--dbpath arg    # 指定数据库路径
--diaglog arg   # diaglog 选项 0=off 1=W 2=R 3=both 7=W+some reads
--directoryperdb # 设置每个数据库将被保存在一个单独的目录
--journal       # 启用日志选项，MongoDB 的数据操作将会写入到 journal 文件夹的文件里
--journalOptions arg  # 启用日志诊断选项
--ipv6        # 启用 IPv6 选项
--jsonp        # 允许 JSONP 形式通过 HTTP 访问（有安全影响）
--maxConns arg   # 最大同时连接数默认 2000
--noauth       # 不启用验证
--nohttpinterface # 关闭 http 接口，默认关闭 27018 端口访问
--noprealloc    # 禁用数据文件预分配（往往影响性能）
--noscripting   # 禁用脚本引擎
--notablescan   # 不允许表扫描
--nounixsocket # 禁用 Unix 套接字监听
--nssize arg (=16) # 设置信数据库 .ns 文件大小 (MB)
--objcheck     # 在收到客户数据，检查的有效性，
```

```
--profile arg      # 档案参数 0=off 1=slow, 2=all
--quota          # 限制每个数据库的文件数，设置默认为 8
--quotaFiles arg    # number of files allower per db, requires --quota
--rest            # 开启简单的 rest API
--repair          # 修复所有数据库 run repair on all dbs
--repairpath arg    # 修复库生成的文件的目录，默认为目录名称 dbpath
--slowms arg (=100)   # value of slow for profile and console log
--smallfiles       # 使用较小的默认文件
--syncdelay arg (=60)  # 数据写入磁盘的时间秒数 (0=never, 不推荐)
--sysinfo          # 打印一些诊断系统信息
--upgrade          # 如果需要升级数据库 * Replicaton 参数
-----
--fastsync # 从一个 dbpath 里启用从库复制服务，该 dbpath 的数据库是主库的快照，可用于快速启用同步
--autoresync      # 如果从库与主库同步数据差得多，自动重新同步
--oplogSize arg    # 设置 oplog 的大小 (MB) * 主 / 从参数
-----
--master          # 主库模式
--slave           # 从库模式
--source arg      # 从库 端口号
--only arg        # 指定单一的数据库复制
--slavedelay arg   # 设置从库同步主库的延迟时间 * Replica set (副本集) 选项:
-----
--replSet arg      # 设置副本集名称 * Sharding(分片) 选项
-----
--configsvr      # 声明这是一个集群的 config 服务，默认端口 27019，默认目录 /data/configdb
--shardsvr        # 声明这是一个集群的分片，默认端口 27018
--noMoveParanoia   # 关闭偏执为 moveChunk 数据保存
```

上述参数也可以直接在 mongod.conf 配置文件中配置，例如：

```
dbpath = /data/mongodb
logpath = /data/mongodb/mongodb.log
logappend = true
port = 27017
fork = true
auth = true
```

## 12.4 本章小结

本章 MySQL 小节介绍了标准镜像的创建过程，以及如何将数据库文件映射到宿主主机来减少 AUFS 系统的性能损耗，还介绍了 MySQL 数据库的主从复制模式，读者通过阅读该

小节，应该能够进一步了解如何在生产环境中部署和使用 MySQL 的 Docker 容器。

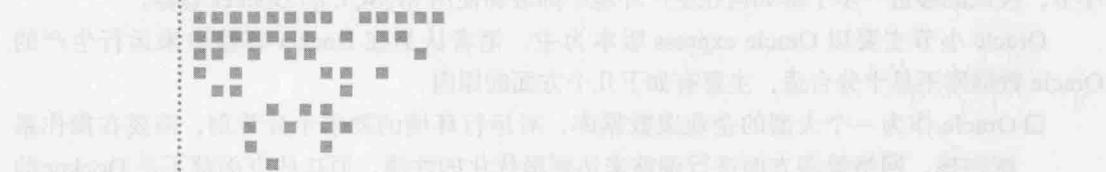
Oracle 小节主要以 Oracle express 版本为主。笔者认为在 Docker 容器中来运行生产的 Oracle 数据库不是十分合适，主要有如下几个方面的原因：

- Oracle 作为一个大型的企业级数据库，对运行环境的要求十分苛刻，需要在操作系统内核、网络等多方面进行调整来达到最优化的性能，而这些方面都不是 Docker 的强项。
- Oracle 数据库拥有自己非常完善的集群软件，包括 Clusterware、ASM、Dataguard 等组件，来保证性能和可用性，这方面目前还没有比较好的 Docker 支持。
- 购买 Oracle 软件需要昂贵的授权许可，而且 Docker 技术目前不在 Oracle 公司支持的操作系统列表里面，无法得到 Oracle 公司的完善的技术支持。

虽然如此，经过笔者的验证，Oracle Express 版本可以在 Docker 中正常运行，读者可以用它来快速搭建个人开发、学习 Oracle 数据库的环境。

本章最后还介绍了非关系型数据库 MongoDB 镜像的创建，跟 MySQL 镜像一样，根据该小节介绍创建的镜像，拥有许多可以供读者自定义的选项，来创建符合自己需求的 MongoDB 应用。

阅读本章需要对特定数据库的配置和结构有一定的基础知识，由于篇幅所限，无法一一介绍。读者可以通过各大数据库的官方网站查阅相关资料。



## Chapter 13

# 第 13 章 编程语言

本章主要介绍如何使用 Docker 快速部署主流编程语言的开发环境及其常用框架，包括 C、C++、Java、PHP、Python、Perl、Ruby、JavaScript、Ruby 等。其中，笔者将重点介绍常用 Web 编程语言 PHP 的 Docker 使用。

## 13.1 PHP

### 13.1.1 PHP 技术栈

PHP 是一种广泛使用的动态脚本语言，尤其适用于各种 Web 方案。由于 PHP 易于入门，易于维护的特性，它在国内外被大量的用于快速 Web 开发。即使随着网站 PV/UV 的增长，需要支持更大的并发的时候，基于原有 PHP 系统进行分层优化和业务整合也是相对容易的。PHP 的哲学是 quickand dirty（快速有效为先），任何对交付速度、灵活性甚至招聘成本有要求的创业团队，可以大胆地选择 PHP。



下面，笔者将重点讲解 PHP 语言的 Docker 环境，并简述 PHP 主流 MVC 框架的 Docker 环境。

#### 1. 使用官方镜像

首先，下载 PHP 官方基础镜像。

```
$ sudo docker pull php
```

下载成功后，读者已经可以使用一个 PHP 容器去运行 PHP 程序 / 站点了。

**第一步** 如果读者需要以 CLI ( command line interface 命令行) 方式运行 PHP 脚本，可以按照以下步骤操作：

1) 在 PHP 程序 / 站点的根目录中新建一个 Dockerfile，内容为：

```
FROM php:5.6-cli
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
CMD [ "php", "./your-script.php" ]
```

2) 然后运行以下命令去构建 Docker 镜像：

```
$ sudo docker build -t my-php-app .
```

3) 最后执行以下命令去运行 Docker 镜像：

```
$ sudo docker run -it --rm --name my-running-app my-php-app
```

**第二步** 如果读者需要运行简单的，甚至单文件的 PHP 项目，那么每次都写 Dockerfile 会很麻烦。这种情况下，你可以用以下命令直接运行 PHP 脚本：

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp php:5.6-cli php your-script.php
```

**第三步** 通常情况下，PHP 项目需要和 Apache httpd/Nginx 一起运行。这样就需要 PHP 容器中内含 Apache Web Server。读者可以使用带有 apache 标签的镜像，如 php:5.6-apache。

1) 在读者的 PHP 项目的根目录中新建一个 Dockerfile，并使用 Docker Hub 官方的基础镜像：

```
FROM php:5.6-apache
COPY src/ /var/www/html/
```

src/ 是当前包含所有 PHP 代码的目录。

2) 使用此 Dockerfile 构建自定义镜像：

```
$ sudo docker build -t my-php-app .
```

3) 创建并运行此镜像：

```
$ sudo docker run -it --rm --name my-running-app my-php-app
```

笔者建议加入一个自定义的 php.ini 配置文件，将其拷贝到 /usr/local/lib。这样读者可以对 PHP 项目做更多的定制化，如开启某些 PHP 插件，或者对 PHP 解释器进行一些安全 / 性能相关的配置。

添加方法很简单：

```
FROM php:5.6-apache
COPY config/php.ini /usr/local/lib/
COPY src/ /var/www/html/
```



**注意** src/ 是当前存放 PHP 代码的文件夹, config/ 文件夹包含 php.ini 文件。

如果读者希望直接使用 Docker Hub 官方镜像运行 PHP 项目, 可以执行:

```
$ sudo docker run -it --rm --name my-apache-php-app -v "$(pwd)":/var/www/html
php:5.6-apache
```

Docker Hub 中的优质 PHP 镜像很多, 笔者特别推荐 tutum 团队发布的系列镜像 (包括 apache-php), 具体地址可以在 Docker Hub 中搜索 php 或者 tutum php, 也可以参见资源章节

## 2. 定制镜像

笔者首先推荐读者基于本书第 10 章提供的 SSHD 镜像进行自定义 PHP 镜像的制作, 如此一来, 读者可以方便地使用 SSH 服务连接 PHP 容器, 即方便地运行容器中的 PHP 站点。

下面, 笔者将带领读者从头操作一次。

第一步, 下载 PHP 官方基础镜像:

```
$ sudo docker pull php
Pulling repository docker/php
7a0d03c7b9dc: Download complete
511136ea3c5a: Download complete
36fd425d7d8a: Download complete
aaabd2b41e22: Download complete
35a6381b9f4d: Download complete
afcdff084bd7: Download complete
31d7cb82d7f6: Download complete
8859f0d2ad74: Download complete
a50bbc401f05: Download complete
01a6ed55fbf9: Download complete
1a74af35a74: Download complete
da565d6c25d3: Download complete
Status: Downloaded newer image for docker/php:latest
```

下载完成后, 可以使用 docker images 查看 PHP 基础镜像是否安装完成:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
docker/php	cli	7a0d03c7b9dc	6 days ago	394.7 MB
docker/php	latest	7a0d03c7b9dc	6 days ago	394.7 MB
docker/php	5-cli	7a0d03c7b9dc	6 days ago	394.7 MB
docker/php	5	7a0d03c7b9dc	6 days ago	394.7 MB
docker/php	5.6	7a0d03c7b9dc	6 days ago	394.7 MB

```
docker/php    5.6-cli          7a0d03c7b9dc    6 days ago  394.7 MB
docker/php    5.6.2           7a0d03c7b9dc    6 days ago  394.7 MB
docker/php    5.6.2-cli       7a0d03c7b9dc    6 days ago  394.7 MB
```

命令结果显示，PHP 5.6.2 已经安装完成（5.6.2 即为目前最新的 PHP 官方镜像标签）。

第二步，笔者将在 Docker 中运行一条 PHP 命令（CLI）：

```
$ sudo docker run -it docker/php
Interactive shell

php >
```

可见 CLI 可以直接输出。

```
$ sudo docker run -it php echo 'hello docker!'
hello docker!
```

第三步，笔者将在 Docker 中运行一段 PHP 代码：

首先，读者需要确定当前目录位置，笔者使用 `pwd` 命令，创建一个 `sample` 目录：

```
$ pwd
/home/core
$ mkdir sample
$ cd sample
```

此时笔者已经确定在此目录是有读写权限的，下面使用 Vim 文本编辑器新建一个 PHP 文件：

```
$ vim demo.php
```

此时会打开一个空文件，然后输入 `i` 进入 Vim 的编辑模式，输入一下代码：

```
<?php
class demo {
    function __construct() {
        echo 'Building Object.';
        echo "\n";
    }

    function hello_world() {
        print 'Hello World!';
    }
}

$demo_object = new demo();
$demo_object->hello_world();
?>
```

下面，我们基于本书创建的 `sshd` 镜像，构建一个能够方便地运行 PHP 业务代码（非单个 PHP CLI 命令）的镜像：

```

FROM sshd:dockerfile

# 安装基础镜像
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && \
    apt-get -yq install \
        curl \
        apache2 \
        libapache2-mod-php5 \
        php5-mysql \
        php5-gd \
        php5-curl \
        php-pear \
        php-apc && \
    rm -rf /var/lib/apt/lists/*
RUN sed -i "s/variables_order.*/variables_order = \"EGPC$\"/g" /etc/php5/
apache2/php.ini
RUN curl -ss https://getcomposer.org/installer | php -- --install-dir=/usr/
local/bin --filename=composer

RUN echo "Asia/Shanghai" > /etc/timezone && \
    dpkg-reconfigure -f noninteractive tzdata
# 注意这里要更改系统的时区设置，因为在 web 应用中经常会用到时区这个系统变量，默认的 ubuntu 会让
你的应用程序发生不可思议的效果哦

# 添加我们的脚本，并设置权限，这会覆盖之前放在这个位置的脚本
ADD run.sh /run.sh
RUN chmod 755 /*.sh

# 添加一个示例的 php 站点，删掉默认安装在 apache 文件夹下面的文件，并将我们添加的示例用软链接链接
到 /var/www/html 目录下面
RUN mkdir -p /var/lock/apache2 && mkdir -p /app && rm -fr /var/www/html && ln -s
/app /var/www/html
COPY sample/ /app

# 设置 apache 相关的一些变量，在容器启动的时候可以使用 -e 参数替代
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_SERVERADMIN admin@localhost
ENV APACHE_SERVERNAME localhost
ENV APACHE_SERVERALIAS docker.localhost
ENV APACHE_DOCUMENTROOT /var/www

# 使用 80 端口
EXPOSE 80
WORKDIR /app
CMD ["/run.sh"]

```

run.sh 文件内容如下：

```
#!/bin/bash
/usr/sbin/sshd -D &
chown www-data:www-data /app -R
source /etc/apache2/envvars
exec apache2 -D FOREGROUND
```

然后，构建此镜像：

```
$ sudo docker build -t my-php:dockerfile .
sudo docker build -t php .
Sending build context to Docker daemon 7.68 kB
Sending build context to Docker daemon
Step 0 : FROM sshd
--> 2e89cae5f1b6
Step 1 : ENV DEBIAN_FRONTEND noninteractive
--> Using cache
--> cfdf02ec333c
Step 2 : RUN apt-get update && apt-get -yq install curl apache2
libapache2-mod-php5 php5-mysql php5-gd php5-curl php-pear
php-apc && rm -rf /var/lib/apt/lists/*
...
Removing intermediate container 6273e14f6a91
Successfully built bdb6460f17d7
```

构建镜像成功后，运行此镜像：

```
$ sudo docker run -d -P php
55745292b34cb085fbf2425988d00a51c451c79071f0929f2ca4943d93e33dad
```

查看是否启动成功：

```
$ sudo docker ps
CONTAINER ID        IMAGE          COMMAND       CREATED      STATUS        PORTS     NAMES
55745292b34c        php:latest    "/run.sh"     3 seconds ago   Up 2 seconds
0.0.0.0:49159->22/tcp, 0.0.0.0:49160->80/tcp   happy_blackwell
```

使用 49160 端口可以打开 php 页面，使用 49159 端口可以打开 ssh 服务。

```
$ curl 127.0.0.1:49160/demo.php
Building Object.
Hello World!
```

我们还可以通过浏览器访问 <http://宿主主机 ip:49160> 来访问我们的示例程序，如图 13-1 所示。

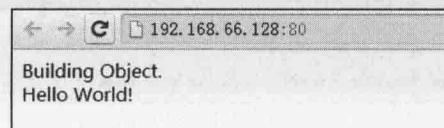


图 13-1 PHP 页面

如果读者遇到命令行报错: ssh: connect to host coreos-ip port 10123: Connection refused, 则请检查 SSH 连接所需的 RSA 密钥是否生成正确。

### 13.1.2 PHP 常用框架

PHP MVC 框架资源非常丰富。常用的 Web 解决方案中涉及的有 Zend、Yii、CakePHP、ThinkPHP、CodeIgniter、Laravel 等。笔者将以 CakePHP 框架为主, 其他框架为辅, 阐述 Docker 的各种使用方法。

#### 1. CakePHP

CakePHP 是 Rails 风格的全栈开源 MVC 框架 (此处的全栈 Full-stack Framework, 是相对于 Yaf 这类微框架 micro-framework 而言的), CakePHP 提供了完整的高级框架所需提供的所有组件, 如脚手架 (代码生成器)、模板引擎、功能全面的 ORM, 完整的面向对象的封装 (包括 CakeRequest、AppModel、AppController 等)。CakePHP 社区完善, 企业级使用者众多, 是优秀的快速开发框架。



使用官方镜像的方法如下。

第一步, 下载 Docker Hub 镜像:

```
$ sudo docker pull vcarl/cakephp
```

第二步, 使用 docker run 命令运行 CakePHP 容器:

```
$ sudo docker run -d -p 80:80 -p 443:443 -v /path/to/project:/var/www/html vcarl/apache
```

注意, /path/to/project 不能指向 app/ 文件夹内部, 而应该指向项目根目录。

第三步, 打开浏览器, 打开 localhost: 80 查看运行结果, 如图 13-2 所示。

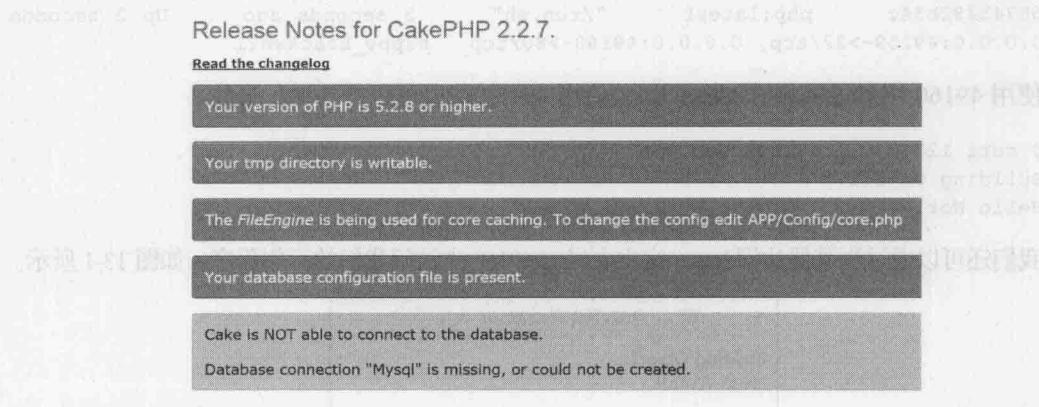


图 13-2 CakePHP 页面

如果读者需要运行现有 CakePHP 项目，则可以选择使用上文的实战演练中的方法——基于第 10 章的 SSHD 镜像，自定义 Dockerfile 后，构建 CakePHP 镜像。这样可以方便地使用 SSH 服务连接 CakePHP 镜像。当然，读者也可以到 Docker Hub 自行搜索 CakePHP 第三方镜像。

## 2. Zend

Zend Framework (ZF) 是用 PHP 5 来开发 Web 程序和服务的开源框架。ZF 用 100% 面向对象编码实现。ZF 的组件结构独一无二，每个组件几乎不依靠其他组件。这样的松耦合结构 (use-at-will) 可以让开发者独立使用各种组件。正确使用 Zend Framework 可以保持良好的代码结构和高可维护性，即使是规模庞大的 Web 项目。Zend 框架是 PHP Web 方案的常用选型之一。它比较适合业务逻辑复杂，并且对拓展性、稳定性要求较高的中大型 Web 项目。



使用 Zend Framework 最简单的方式是安装 Zend Server。下面笔者将介绍 Zend Server 及其 Docker 化部署。

Zend Server 是针对于关键 Web 业务的 Web 应用服务器，适合企业级解决方案。笔者在此简略介绍一下 Zend Server 的特性：

- 企业级 PHP：一款最新的，经受测试以及支持 PHP 堆栈的服务器，它可以确保应用程序的高可靠性、提高应用程序的安全性以及效率。
- 使部署更有信心：在开发中使用一个完整的和一致的环境，这样在部署过程中读者可以消除许多测试和运行所遇到的问题。
- 快速问题响应：利用先进的应用程序监测和诊断，能及早发现问题并能快速分析问题的根源。
- 最好的应用性能：内置的优化和加速功能，确保了高性能和低资源利用率。
- 代码加速：PHP 的字节码缓存提高性能而没有使应用程序发生变化。
- 完全页面缓存：以 URL 为基础的 HTML 输出缓存，它不需要任何应用程序的变。
- 部分页面缓存：它允许开发人员能够在共享内存或磁盘中缓存数据。

### 使用官方镜像

Zend Server 在 Docker Hub 上以 `php-zendserver` 的名称标示。首先，我们下载 Zend Server 的 Docker Hub 官方镜像：

```
$ sudo docker pull php-zendserver
```

如果读者对 Zend Server 的版本有要求，可以在以上命令中加入 Tag 标签，以便于在下一步的 Dockerfile 的 FROM 指令中明确 Zend Server 版本号。官方镜像都有明确的标签列表，具体地址可以参见后面“相关资源”部分。

创建并运行单个 Zend Server 容器：

```
$ sudo docker run php-zendserver
```

注意以下几点：

1) 如果你对 PHP 和 Zend Server 的版本都有要求，可以在 docker run 命令中加入：:<php-version> 或者 :<ZS-version>-php<version>。目前可供使用的 PHP 版本有 5.4 和 5.5（默认），Zend Server 版本为 7（如：php-zendserver:7.0-php5.4）。

2) 如果需要使用特定版本的 Zend Server，也可以选择直接下载此版本的镜像：

Zend Server 5.4 Docker 官方镜像：docker pull zend/php-5.4-zend-server

Zend Server 5.5 Docker 官方镜像：docker pull zend/php-5.5-zend-server

3) 如果读者需要搭建一个 Zend Server 集群，并在每个 Zend 集群节点（即 Cluster Node）都使用来自官方镜像的 Zend Server 容器，那么可以在每个节点执行：

```
$ sudo docker run -e MYSQL_HOSTNAME=<db-ip> -e MYSQL_PORT=3306 -e MYSQL_USERNAME=<username> -e MYSQL_PASSWORD=<password> -e MYSQL_DBNAME=zend php-zendserver
```

如果读者需要执行自定义操作，可以自定义 Dockerfile，以生成自定义的镜像，步骤如下：

1) 修改 Dockerfile，加入定制内容。在含有 Dockerfile 的根目录执行以下语句构建镜像：

```
$ sudo docker build .
```

执行完毕后，命令行会输出镜像的 ID (image-id)。

2) 通过镜像 ID，启动单个 Zend Server 容器：

```
$ sudo docker run <image-id>
```

如果读者需要搭建一个 Zend Server 集群，并在每个 Zend 集群节点（即 Cluster Node）都使用来自自定义镜像的 Zend Server 容器，那么读者可以在每个节点执行：

```
$ sudo docker run -e MYSQL_HOSTNAME=<db-ip> -e MYSQL_PORT=3306 -e MYSQL_USERNAME=<username> -e MYSQL_PASSWORD=<password> -e MYSQL_DBNAME=zend <image-id>
```

注意以下几点：

1) 当运行多个容器（即 Instance 实例）时，只有一个实例只可以绑定至一个端口（命令中的 port）。如果需要运行一个集群，可以有两个做法：a) 将不同的若干端口分别定向至各个节点。b) 将不同的若干端口分别定向至各个容器。

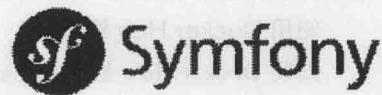
2) 使用环境变量。使用预定义的管理员用户名密码启动 Zend Server：- ZS\_ADMIN\_PASSWORD。集群操作中的 MySQL 变量（以下指令都是节点正常加入集群所必须的）：

□ MYSQL\_HOSTNAME：MySQL 数据库的 ip 或 hostname。

- MySQL\_PORT: MySQL 监听的端口。
- MySQL\_USERNAME - MySQL\_PASSWORD - MySQL\_DBNAME: 集群操作时 Zend Server 使用的数据库用户名密码 (如果不存在将会新建)。
- 3) ZEND\_LICENSE\_KEY - ZEND\_LICENSE\_ORDER: 使用已购买的许可证。
- 4) 每个 Zend Server 的 Docker 容器至少需要 1GB 的可用内存。

### 3. Symfony

Symfony 是一个优秀的以依赖注入为核心的全栈式 PHP MVC 开发框架。它以设计完善和性能优越著称。它提供模板系统，数据持久层，代码生成器，以及大量可复用的功能集合 (Bundle)。同时，Symfony 以依赖注入的方式给开发者提供了近乎无限的扩展性。



#### 使用 Docker Hub 镜像

运行 Symfony 最简便的方法是去 Docker Hub 直接搜索相关关键字并直接下载 Symfony 镜像：

```
$ sudo docker pull gregory90/php-symfony
```

或者

```
$ sudo docker pull teamrock/symfony2
```

由于这种方式有一定局限性，对于需要执行已有 Symfony 项目的用户并不实用，所以读者也可以如下文所示定制镜像。

#### 定制镜像

读者可以基于 SSHD 镜像定制，也可以使用以下 Dockerfile 来定制镜像：

```
# 获取 Apache 基础镜像
FROM teamrock/apache2:production

RUN DEBIAN_FRONTEND=noninteractive apt-get update -y
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y php5-imagick php5-gd
php5-intl php5-mcrypt php5-apcu php5-curl php5-mysql

# 添加 vhost 配置
ADD ./virtual-host.conf /etc/apache2/sites-enabled/0-virtual-host.conf

# 添加安装脚本
ADD ./run.sh /tmp/run.sh

# 设置启动脚本
ENTRYPOINT [ "/bin/bash", "/tmp/run.sh" ]
```

#### 4. Phalcon

PhalconPHP 是一个使用 C 扩展开发的 PHP Web 框架，提供高性能和低资源占用。

Phalcon 是一个开源的、全堆栈的 PHP 5 框架，使用 C 扩展编写，专门为高性能优化。无需学习和使用 C 语言，所有函数都以 PHP 类的方式出现。Phalcon 是一个松耦合的框架。

Phalcon 是目前 PHP MVC 框架中性能最高的全栈框架，Yaf 也具有极高的性能，但是非全栈框架。

##### 使用 Docker Hub 镜像

安装 Docker Hub 上的 Phalcon 镜像：

```
$ sudo docker pull szeist/phalcon-apache2
```

创建并运行 Phalcon 镜像：

```
$ sudo docker run -v /document/root/on/your/mahine:/var/www -p 8080:80 szeist/phalcon-apache2
```

此时 Apache 服务器会在 TCP 端口 8080 运行 PhalconPHP 框架。`-v` 命令后的文件夹就是 Phalcon 项目的根目录。

##### 定制镜像

笔者推荐读者基于本书第 10 章使用的 SSHD 镜像来定制自定义镜像，这样可以方便地使用 SSH 服务访问 Phalcon 容器。当然，读者也可以使用第三方 Dockerfile 来定制镜像：

```
FROM szeist/phalcon

ENV DEBIAN_FRONTEND noninteractive

RUN apt-get install -y -q apache2 libapache2-mod-php5 ;\ 
    a2enmod rewrite

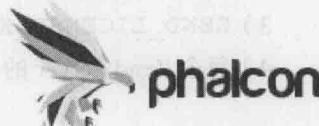
RUN apt-get clean

ADD etc/apache2/sites-available/000-default.conf /etc/apache2/sites-available/000-default.conf

ENV DEBIAN_FRONTEND dialog

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_PID_FILE /var/run/apache2/apache2.pid

EXPOSE 80
```



```
ENTRYPOINT ["/usr/sbin/apache2"]
CMD ["-D", "FOREGROUND"]
```

### 13.1.3 相关资源

PHP 官网: <http://php.net/>

PHP Docker 官方镜像: [https://registry.hub.docker.com/\\_/php/](https://registry.hub.docker.com/_/php/)

PHP tutum 镜像: <https://registry.hub.docker.com/u/tutum/apache-php/>

PHP tutum Dockerfile: <https://registry.hub.docker.com/u/tutum/apache-php/dockerfile/>

CakePHP 框架官网: <http://cakephp.org/>

CakePHP Docker Hub 镜像: <https://registry.hub.docker.com/u/vcarl/cakephp/>

Zend 框架官网: <http://framework.zend.com/>

Zend Server 官网: <http://www zend com/en/products/server>

Zend Server Docker 官方镜像: [https://registry.hub.docker.com/\\_/php-zendserver/](https://registry.hub.docker.com/_/php-zendserver/)

Zend Server Docker 官方镜像标签: [https://registry.hub.docker.com/\\_/php-zendserver/tags/manage/](https://registry.hub.docker.com/_/php-zendserver/tags/manage/)

Zend Server 5.4 Docker 官方镜像: <https://registry.hub.docker.com/u/zend/php-5.4-zend-server/>

Zend Server 5.5 Docker 官方镜像: <https://registry.hub.docker.com/u/zend/php-5.5-zend-server/>

Symfony 2 框架官网: <http://symfony.com/>

Symfony 2 Dockerfile : <https://registry.hub.docker.com/u/gregory90/php-symfony/dockerfile/>

Phalcon 框架官网: <http://www.phalconphp.com/en/>

Phalcon Docker Hub 镜像: <https://registry.hub.docker.com/u/szeist/phalcon-apache2/>

Phalcon Dockerfile: <https://registry.hub.docker.com/u/szeist/phalcon-apache2/dockerfile/>

## 13.2 C/C++

本节将介绍三款流行的 C/C++ 开发环境: GCC、LLVM 和 Clang。

### 13.2.1 GCC

GCC (GNU Compiler Collection) 是一个开源的 C/C++ 语言的编译器系统, 它由 GNU 项目主持。GCC 支持多种编程语言, 并支持交叉编译至多种指令集的处理器 (Target Processor)。GCC 是 GNU 工具链的关键组件, 遵循 GNU GPL 协议。



#### 1. 使用官方镜像

将 C/C++ 代码运行在 GCC 容器内的最简方法, 就是将 GCC 编译指令写入 Dockerfile

中，然后使用此 Dockerfile 构建自定义镜像，最后直接运行此镜像，即可启动程序。

假定在当前目录创建一个 C 语言源文件 main.c，内容可能为：

```
/* Hello World program */

#include<stdio.h>

int main()
{
    printf("Hello World\n");
    return 0
}
```

首先，从官方仓库获取 GCC 基础镜像：

```
$ sudo docker pull gcc
```

如果对 GCC 的版本有要求，可以在以上命令中加入镜像标签，并在下一步的 Dockerfile 的 FROM 指令中明确 GCC 版本号。

之后，在 Dockerfile 中，加入需要执行的 GCC 编译命令：

```
FROM gcc:4.9
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN gcc -o myapp main.c
CMD ["./myapp"]
```

现在，就可以使用 Dockerfile 来构建镜像 my-gcc-app：

```
$ sudo docker build -t my-gcc-app .
```

创建并运行此容器，会编译并运行程序，输出 Hello World：

```
$ sudo docker run -it --rm --name my-running-app my-gcc-app
Hello World
```

如果只需要容器编译程序，而不需要运行它，可以使用如下命令：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp gcc:4.9 gcc
-o myapp myapp.c
```

以上命令会将当前目录 ("\$(pwd)") 挂载到容器的 /usr/src/myapp 目录，并执行 gcc -o myapp myapp.c。GCC 将会编译 myapp.c 代码，并将生成的可执行文件输出至 /usr/src/myapp 文件夹。

如果项目已经编写好了 Makefile，也可以在容器中直接执行 make 命令：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp gcc:4.9 make
```

## 2. 定制镜像

读者可以基于本书第10章中使用的SSHD基础镜像来定制GCC镜像，构建后直接运行。读者也可以使用Docker Hub中提供的第三方Dockerfile，定制或修改后构建镜像，然后运行容器即可。下面，笔者给出了基于buildpack-deps:wheezy镜像创建GCC镜像的Dockerfile：

```
# https://registry.hub.docker.com/u/snormore/llvm/dockerfile/
FROM buildpack-deps:wheezy

# https://gcc.gnu.org/mirrors.html
RUN gpg --keyserver pgp.mit.edu --recv-key \
    B215C1633BCA0477615F1B35A5B3A004745C015A \
    B3C42148A44E6983B3E4CC0793FA9B1AB75C61B8 \
    90AA470469D3965A87A5DCB494D03953902C9419 \
    80F98B2E0DAB6C8281BDF541A7C8C3B2F71EDF1C \
    7F74F97C103468EE5D750B583AB00996FC26A641 \
    33C235A34C46AA3FFB293709A328C3A2C3C45C06

ENV GCC_VERSION 4.9.1

# 下载需要的tar格式源码并解压安装
RUN apt-get update \
    && apt-get install -y curl flex wget \
    && rm -r /var/lib/apt/lists/* \
    && curl -SL "http://ftpmirror.gnu.org/gcc/gcc-$GCC_VERSION/gcc-$GCC_ \
VERSION.tar.bz2" -o gcc.tar.bz2 \
    && curl -SL "http://ftpmirror.gnu.org/gcc/gcc-$GCC_VERSION/gcc-$GCC_ \
VERSION.tar.bz2.sig" -o gcc.tar.bz2.sig \
    && gpg --verify gcc.tar.bz2.sig \
    && mkdir -p /usr/src/gcc \
    && tar -xvf gcc.tar.bz2 -C /usr/src/gcc --strip-components=1 \
    && rm gcc.tar.bz2* \
    && cd /usr/src/gcc \
    && ./contrib/download_prerequisites \
    && { rm *.tar.* || true; } \
    && dir=$(mktemp -d) \
    && cd "$dir" \
    && /usr/src/gcc/configure \
        --disable-multilib \
        --enable-languages=c, c++ \
    && make -j"$(nproc)" \
    && make install-strip \
    && cd .. \
    && rm -rf "$dir" \
    && apt-get purge -y --auto-remove curl gcc g++ wget
```

### 13.2.2 LLVM

LLVM (Low Level Virtual Machine) 是一个起源于 2000 年的编译器基础建设项目，以 C++ 实现，包含 LLVM 中介码 (LLVM IR)、LLVM 除错工具、LLVM C++ 标准库等。该项目是为了对任意编程语言实现的程序，利用虚拟技术，创造出编译时期、链接时期、运行时期以及“闲置时期”的优化。它最早是以 C/C++ 为对象，目前已经支持了包括 ActionScript、Ada、D、Fortran、GLSL、Haskell、Java bytecode、Objective-C、Swift、Python、Ruby、Rust、Scala 以及 C# 等众多语言。

#### 1. 使用官方镜像

在 Docker Hub 中已经有用户提供了 LLVM 的镜像，读者可以直接下载使用。

```
$ sudo docker pull imiell/llvm
```

#### 2. 定制镜像

读者可以基于本书第 10 章中使用的 SSHD 基础镜像来定制 GCC 镜像，构建后直接运行。读者也可以使用 Docker Hub 中提供的第三方 Dockerfile，定制或修改后构建镜像，然后运行容器即可。

### 13.2.3 Clang

Clang 是一个用 C++ 实现、基于 LLVM 的 C/C++/Objective C/Objective C++ 编译器，其目标（之一）就是超越 GCC 成为标准的 C/C++ 编译器，它遵循 LLVM BSD 许可。Clang 最初是 Apple 公司为了解决使用 GCC 编译 Objective-C 的问题，从头实现的一套编译器系统。

Clang 有如下特性：

- 快：通过编译 OS X 上几乎包含了所有 C 头文件的 carbon.h 的测试，包括预处理 (Preprocess)、语法 (lex)、解析 (parse)、语义分析 (Semantic Analysis)、抽象语法树生成 (Abstract Syntax Tree) 的时间，Clang 比 GCC 4.0 快 2.5 倍。
- 内存占用小：Clang 内存占用约是源码的 1.3 倍，而 Apple GCC 则超过 10 倍的内存使用。
- 诊断信息可读性强：Clang 对于错误的语法不但有源码提示，还会在错误的调用和相关上下文上有更好提示。
- 良好的 GCC 兼容性。
- 设计清晰简单，容易理解，易于扩展增强。与代码基础古老的 GCC 相比，学习曲线平缓。
- 基于库的模块化设计，易于 IDE 集成及其他用途的重用。由于历史原因，GCC 是一个单一的可执行程序编译器，其内部完成了从预处理到最后代码生成的全部过程，

中间诸多信息都无法被其他程序重用。Clang 将编译过程分成彼此分离的几个阶段，AST 信息可序列化。通过库的支持，程序能够获取到 AST 级别的信息，将大大增强对于代码的操控能力。对于 IDE 而言，代码补全、重构是重要的功能，然而如果没有底层的支持，只使用 tags 分析或正则表达式匹配是很难达成的。

## 1. 使用官方镜像

在 Docker Hub 中已经有用户提供了 Clang 的镜像，读者可以直接下载使用。

```
$ sudo docker pull bowery/clang
```

## 2. 定制镜像

如前文所说，读者可以基于 SSHD 镜像自定义 Dockerfile。也可以使用 Docker Hub 中的第三方镜像构建 Clang 容器。这里笔者以 ubuntu:trusty 系统为例，下面给出了一个示例 Dockerfile 文件：

```
# https://registry.hub.docker.com/u/rsmmr/clang/dockerfile
FROM      ubuntu:trusty

# 设置环境变量
ENV PATH /opt/llvm/bin:$PATH

# 确定默认的启动命令
CMD bash

# 安装依赖包 Setup packages.
RUN apt-get update && apt-get -y install cmake git build-essential vim python

# 将 install-clang 拷贝至本目录
ADD . /opt/install-clang

# 编译和安装 LLVM/clang
RUN /opt/install-clang/install-clang -j 4 -C /opt/llvm
```

## 13.3 Java

Java 是一种拥有跨平台、面向对象、泛型编程特点的编译型语言，广泛应用于企业级 Web 应用开发和移动应用开发。它是并发的，基于类的面向对象的高级语言。Java 的设计理念是尽可能地减少部署依赖，致力于允许 Java 应用的开发者“开发一次，到处运行”。这就意味着 Java 的二进制编码不需要再次编辑，即可运行在异构的 JVM 上。Java 在大型互联网项目，特别是互联网金融和电子商务项目中非常受欢迎。



## 1. 使用官方镜像

将 Java 代码运行在 Docker 容器中的最简单方法，就是将 Java 编译指令直接写入 Dockerfile，然后使用此 Dockerfile 构建镜像，最后直接运行此镜像，即可启动程序。具体步骤如下：

首先，从官方仓库获取 Java 基础镜像：

```
$ sudo docker pull java
```

然后，在本地新建一个空目录，在其中创建 Dockerfile 文件。在 Dockerfile 中，加入需要执行的 Java 编译命令，例如：

```
FROM java:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac Main.java
CMD ["java", "Main"]
```

使用此 Dockerfile 构建镜像 my-java-app：

```
$ sudo docker build -t my-java-app .
```

然后，运行此镜像即自动编译程序并执行：

```
$ sudo docker run -it --rm --name my-running-app my-java-app
```

如果只需要容器中编译 Java 程序，而不需要运行，则可以使用如下命令：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp java:7
javac Main.java
```

以上命令会将当前目录（“\$(pwd)”）挂载为容器的工作目录，并执行 **javac Main.java** 命令编译代码。生成的可执行文件将输出至当前目录下。

## 2. 定制镜像

笔者建议用户基于第 10 章的 SSHD 镜像来自定义 Dockerfile。当然，读者也可以基于各种系统基础镜像（如 Ubuntu）来定制 Java 镜像。此处笔者给出了基于 **dockerfile/ubuntu** 镜像创建 Java 镜像的 Dockerfile 文件，供读者参考：

```
#
# OpenJDK Java 7 JRE Dockerfile
#
# https://github.com/dockerfile/java
# https://github.com/dockerfile/java/tree/master/openjdk-7-jre
#
# 使用 Ubuntu 基础镜像
```

```

FROM dockerfile/ubuntu

# 安装 Java 环境
RUN \
    apt-get update && \
    apt-get install -y openjdk-7-jre && \
    rm -rf /var/lib/apt/lists/*

# 定义工作目录
WORKDIR /data

# 定义环境变量
ENV JAVA_HOME /usr/lib/jvm/java-7-openjdk-amd64

# 定义默认启动命令
CMD [ bash ]

```

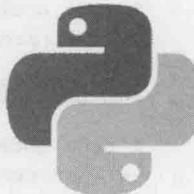
### 3. 相关资源

- Java Docker 官方镜像: [https://registry.hub.docker.com/\\_/java/](https://registry.hub.docker.com/_/java/)
- Java Docker 官方镜像标签: [https://registry.hub.docker.com/\\_/java/tags/manage/](https://registry.hub.docker.com/_/java/tags/manage/)

## 13.4 Python

### 13.4.1 Python 技术栈

Python 是一种解释型的、带 CLI 交互接口的、面向对象的、开源的动态脚本语言。它集成了模块 (modules)、异常处理 (exceptions)、动态类型 (dynamic typing)、高级数据结构 (元组、列表、序列)，以及类 (classes) 等高级特性。Python 设计精良，语法简约，表达能力很强。Python 代码与 C/C++ 代码之间可以方便地相互调用。Python 还可以作为各种需要可编程接口的应用：无论是各种系统管理接口，还是各种需要交互式 CLI 的应用。Python 的可移植性非常强，所有的当前主流操作系统（Windows 2000+，所有 Mac，类 Unix 系统）都支持 Python。Python 非常适合敏捷 Web 开发的各种应用场景。在中小型团队中，可以选择 Python 作为 Restful API 后端或者移动 App 后端的实现语言。



下面，笔者将带领大家使用 Docker 部署 Python 环境，以及部署 Python 技术栈中的主流框架。

#### 1. 使用官方镜像

我们可以使用 Docker 官方的 Python 镜像作为基础，在此基础上进行必要的定制：

第一步，下载 Docker 官方的 Python 镜像：

```
$ sudo docker pull python
```

如果读者对 Python 的版本有要求，可以在以上命令中加入 Tag 标签，以便于在下一步的 Dockerfile 的 FROM 指令中明确 Python 版本号。官方镜像都有明确的标签列表，具体地址可以参见本章的“相关资源”部分。

第二步，在读者的 Python 项目中新建一个 Dockerfile：

```
FROM python:3-onbuild
CMD [ "python", "./your-daemon-or-script.py" ]
```

如果读者需要使用 Python 2 的话，可用下列命令：

```
FROM python:2-onbuild
CMD [ "python", "./your-daemon-or-script.py" ]
```

之后读者就可以在一个 Python 容器中运行自己的程序了。

第三步，通过此 Dockerfile，构建自定义的镜像：

```
$ sudo docker build -t my-python-app .
```

第四步，创建容器并运行：

```
$ sudo docker run -it --rm --name my-running-app my-python-app
```

如果读者只需要运行单个 Python 脚本，那么无需使用 Dockerfile 构建自定义镜像，而是通过以下命令直接使用官方 Python 镜像，带参数运行容器：

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp python:3 python your-daemon-or-script.py
```

如果读者需要使用 Python 2 的话，可用下列命令：

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp python:2 python your-daemon-or-script.py
```

## 2. 定制镜像

笔者首先推荐读者基于本书第 10 章的 SSHD 镜像进行 Python 镜像的定制，这样可以使用户 SSH 服务方便地访问 Python 容器。当然，也可以使用以下 Dockerfile 来定制镜像：

```
FROM buildpack-deps
# 系统更新
RUN apt-get update && apt-get install -y curl procps
# 清理 Debian 系统已有的 Python 环境
RUN apt-get purge -y python python-minimal python2.7-minimal
```

```

RUN mkdir /usr/src/python
WORKDIR /usr/src/python

# 设置系统变量
ENV LANG C.UTF-8

ENV PYTHON_VERSION 2.7.8

RUN curl -SL "https://www.python.org/ftp/python/$PYTHON_VERSION/Python-$PYTHON_
VERSION.tar.xz" \
    | tar -xJ --strip-components=1
"test_listfolders" with "AssertionError: Lists differ: [] != ['deep', 'deep/
f1', 'deep/f2', ..."
RUN ./configure \
    && make -j$(nproc) \
    && make EXTRATESTOPTS='--exclude test_file2k test_mhlib' test \
    && make install \
    && make clean

# 安装 pip 和 virtualenv
RUN curl -SL 'https://bootstrap.pypa.io/get-pip.py' | python2
RUN pip install virtualenv

CMD ["python2"]

```

### 13.4.2 Flask

Flask 是一个使用 Python 编写的轻量级 Web 应用框架。基于 Werkzeug WSGI 工具箱和 Jinja2 模板引擎。Flask 使用 BSD 授权。Flask 也称为“microframework”，因为它仅仅使用简单的核心，使用 extension 来增加其他功能。

笔者在此简述一下 Flask 的特色：

- 内置开发用服务器和调试器 (debugger)
- 集成单元测试 (unit testing)
- RESTful request dispatching
- 使用 Jinja2 模板引擎
- 支持 secure cookies (client side sessions)
- 100% WSGI 1.0 兼容
- Unicode based
- 详细的文件、教学
- Google App Engine 兼容



web development,  
one drop at a time

### □ 可用 Extensions 增加其他功能

Flask 是目前广受欢迎的 Python Web 技术选型之一。

## 1. 使用官方镜像

第一步，项目准备工作：构建 Flask App 目录：

```
src/
  run.py
  app/
    __init__.py
    views.py
    static/
    templates/

run.py:
#!/usr/bin/python
from app import app
app.run(host='0.0.0.0', port=5000, debug=True)
__init__.py:
from flask import Flask

app = Flask(__name__)
from app import views
```

第二步，下载 Docker Hub 的 Flask 镜像：

```
$ sudo docker pull verdverm/flask
```

第三步，创建并运行 Flask 容器（Flask 的 App 代码作为 Docker 数据卷）：

```
$ sudo docker run -d --name flask-app \
-v /path/to/app/src:/src \
-p 5000:5000 \
verdverm/flask
```

## 2. 定制镜像

笔者首先推荐读者基于本书第 10 章的 SSHD 镜像进行 Flask 镜像的定制，这样可以使用 SSH 服务方便的访问 Flask 容器。当然，读者也可以使用以下 Dockerfile 来定制镜像：

```
FROM google/debian:wheezy
# 系统更新
RUN apt-get update
# 安装 Python 环境
RUN apt-get --no-install-recommends install -y python-setuptools build-
```

```

essential python-dev libpq-dev ca-certificates

# 安装 pip
RUN easy_install pip

ADD requirements.txt /tmp/requirements.txt

# 安装 requirements.txt 中的依赖包，此处可至
RUN pip install -r /tmp/requirements.txt

EXPOSE 5000

VOLUME ["/src"]
WORKDIR /src

ENTRYPOINT ["python", "/src/run.py"]
CMD ["runserver"]

```

### 13.4.3 Django

Django 是一个开放源代码的 Web 应用框架，由 Python 写成。采用了 MVC 的软件设计模式，即模型 M、视图 V 和控制器 C。它最初是用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站的。并于 2005 年 7 月在 BSD 许可证下发布。这套框架是以比利时的吉普赛爵士吉他手 Django Reinhardt 来命名的。Django 的主要目标是使得开发复杂的、数据库驱动的网站变得简单。Django 注重组件的重用性和“可插拔性”，敏捷开发和 DRY 法则（Don't Repeat Yourself）。在 Django 中 Python 被普遍使用，甚至包括配置文件和数据模型。Django 于 2008 年 6 月 17 日正式成立基金会。Django 是 Python MVC 框架中开源组织最完善、社区人气最旺的。



Django 框架的核心包括：

- 一个轻量级的、独立的 Web 服务器，用于开发和测试。
- 一个表单序列化及验证系统，用于 HTML 表单和适于数据库存储的数据之间的转换。
- 一个缓存框架，并有几种缓存方式可供选择。
- 中间件支持，允许对请求处理的各个阶段进行干涉。
- 内置的分发系统允许应用程序中的组件采用预定义的信号进行相互间的通信。
- 一个序列化系统，能够生成或读取采用 XML 或 JSON 表示的 Django 模型实例。
- 一个用于扩展模板引擎的能力的系统。

Django 包含了很多应用在它的“contrib”包中，这些包括：

- 一个可扩展的认证系统。
- 动态站点管理页面。
- 一组产生 RSS 和 Atom 的工具。
- 一个灵活的评论系统。

- 产生 Google 站点地图 (Google Sitemaps) 的工具。
- 防止跨站请求伪造 (cross-site request forgery) 的工具。
- 一套支持轻量级标记语言 (Textile 和 Markdown) 的模板库。
- 一套协助创建地理信息系统 (GIS) 的基础框架。

Django 虽然相比 Tornado、Flask、Web.py 等框架要“重”。但是它“重”在提供一站式框架，提供一些方便快速开发的特性，如脚手架，完整的面向对象的 ORM，还有方便的模板系统。如果创业过程中的中小型项目，选择 Python 技术栈，又对并发支撑能力要求不苛刻的话，可以放心地选择 Django。

Django 是目前广受欢迎的 Python Web 技术选型之一。

## 1. 使用 Docker Hub 镜像

测试数据显示：Django，uWSGI and Nginx 的技术组合拥有优异的性能。我们选用的 Docker Hub 镜像即使用此技术栈：

```
$ sudo docker pull dockerfiles/django-uwsgi-nginx
```

读者也可以利用 Dockerfile 进行定制，自行构建：

```
$ sudo docker build -t webapp .
```

运行：

```
$ sudo docker run -d webapp
```

## 2. 定制镜像

读者可以自建 SSHD 镜像后制作自定义的 Django 镜像。当然，也可以使用以下 Dockerfile 来定制镜像：

```
# 本 Dockerfile 由 Thatcher Peskens 贡献，遵循 Apache 许可
from ubuntu:precise
maintainer Dockerfiles

run echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/
apt/sources.list
run apt-get update
run apt-get install -y build-essential git
run apt-get install -y python python-dev python-setuptools
run apt-get install -y nginx supervisor
run easy_install pip

# 安装 uwsgi
```

```

run pip install uwsgi

# 安装 nginx
run apt-get install -y python-software-properties
run apt-get update
RUN add-apt-repository -y ppa:nginx/stable
run apt-get install -y sqlite3

# 添加项目代码
add . /home/docker/code/

# 修改配置
run echo "daemon off" >> /etc/nginx/nginx.conf
run rm /etc/nginx/sites-enabled/default
run ln -s /home/docker/code/nginx-app.conf /etc/nginx/sites-enabled/
run ln -s /home/docker/code/supervisor-app.conf /etc/supervisor/conf.d

# 运行 pip install
run pip install -r /home/docker/code/app/requirements.txt

# 安装 django
run django-admin.py startproject website /home/docker/code/app/

expose 80
cmd ["supervisord", "-n"]

```

#### 13.4.4 相关资源

Python 官网: <https://www.python.org/>

关于 Python 1: <http://wiki.woodpecker.org.cn/moin/WhyPython>

关于 Python 2: <http://www.linuxjournal.com/article/3882>

Python Docker 官方 Dockerfile: <https://github.com/docker-library/python/>

Python Docker 官方镜像: [https://registry.hub.docker.com/\\_/python/](https://registry.hub.docker.com/_/python/)

Python Docker 官方镜像 Tag: [https://registry.hub.docker.com/\\_/python/tags/manage/](https://registry.hub.docker.com/_/python/tags/manage/)

Flask 官网: <http://flask.pocoo.org/>

Flask Docker 镜像: <https://registry.hub.docker.com/u/verdverm/flask/>

Flask Dockerfile: <https://registry.hub.docker.com/u/verdverm/flask/dockerfile/>

Django 官网: <https://www.djangoproject.com/>

Django Docker 镜像: <https://registry.hub.docker.com/u/dockerfiles/django-uwsgi-nginx/>

Django Dockerfile: <https://registry.hub.docker.com/u/dockerfiles/django-uwsgi-nginx/dockerfile/>

## 13.5 Perl

### 13.5.1 Perl 技术栈

Perl 是一个高级的动态的解释型脚本语言。Perl 的设计借鉴了 C、Shell、awk 和 sed。Perl 最重要的特性是它内部集成了正则表达式的功能，以及巨大的第三方代码库 CPAN。Perl 像 C 一样强大，同时像 awk、sed 等脚本语言一样富有表达性，被称为“Unix 系统王牌工具”。目前 Perl 常见于系统管理、文件处理等程序，笔者认为 Perl 多数情况下属于 Web 方案中的胶水语言。

下面，笔者将主要介绍 Perl5 的 Docker 环境。

#### 使用官方镜像

我们可以使用 Docker 官方的 Perl 镜像作为基础，在此之上进行必要的定制。

第一步，下载 Docker 官方的 Perl 镜像：

```
$ sudo docker pull perl
```

如果读者对 Perl 的版本有要求，可以在以上命令中加入 Tag 标签，以便于在下一步 Dockerfile 的 FROM 指令中明确 Perl 版本号。官方镜像都有明确的标签列表，具体地址可以参见后面的“相关资源”部分。

第二步，在读者的 Perl 项目中新建一个 Dockerfile：

```
FROM perl:5.20
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
CMD [ "perl", "./your-daemon-or-script.pl" ]
```

第三步，通过此 Dockerfile，构建自定义的镜像：

```
$ sudo docker build -t my-perl-app .
```

第四步，创建容器并运行：

```
$ sudo docker run -it --rm --name my-running-app my-perl-app
```

如果读者只需要运行单个的 Perl 脚本，那么无需使用 Dockerfile 构建自定义镜像，而是通过以下命令直接使用官方 Perl 镜像，带参数运行容器：

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp perl:5.20 perl your-daemon-or-script.pl
```

如果读者需要运行 Perl 的 Web 项目，则最好先自建内置 SSH 服务的镜像，然后以此为基础定制 Perl 容器，这样可以方便地通过 SSH 服务访问 Perl 容器。

### 13.5.2 Catalyst

Catalyst 是一个用 Perl 语言开发的 MVC 框架。

#### 1. 使用 Docker Hub 镜像

使用 Docker Hub 镜像如下所示：



```
$ sudo docker pull rsrchboy/perl-catalyst-latest
```

使用 Perl 容器进行一下操作：

```
$ sudo cpan
cpan>install Catalyst::Devel
$catalyst.pl myApp
```

#### 2. 定制镜像

读者可以使用以下 Dockerfile 来定制镜像：

```
# 本 dockerfile 由 Chris Weyl 贡献
FROM rsrchboy/perl-stable:latest

RUN apt-get update && apt-get install -y libmysqlclient-dev libssl-dev
libxml2-dev

# 运行 cpan 管理器
RUN cpanm -q --notest Capture::Tiny && rm -rf ~/.cpanm

RUN cpanm -q --installdeps LWP::UserAgent && rm -rf ~/.cpanm
RUN cpanm -q --notest LWP::UserAgent && rm -rf ~/.cpanm

RUN cpanm -q DBI DBD::mysql && rm -rf ~/.cpanm
RUN cpanm -q DBIx::Class && rm -rf ~/.cpanm
RUN cpanm -q DBIx::Class::Schema::Loader && rm -rf ~/.cpanm
RUN cpanm -q Reindeer && rm -rf ~/.cpanm
RUN cpanm -q Dist::Zilla && rm -rf ~/.cpanm
RUN cpanm -q Task::Plack && rm -rf ~/.cpanm
RUN cpanm -q Task::Catalyst && rm -rf ~/.cpanm
```

### 13.5.3 相关资源

Perl 官网：<http://www.perl.org/>

Perl 中国官网：<http://www.perlchina.org/>

Perl6 官网：<http://rakudo.org/how-to-get-rakudo/>

Perl Docker 镜像：[https://registry.hub.docker.com/\\_/perl/](https://registry.hub.docker.com/_/perl/)

Perl Docker 镜像标签: [https://registry.hub.docker.com/\\_/perl/tags/manage/](https://registry.hub.docker.com/_/perl/tags/manage/)

Catalyst 官网: <http://www.catalystframework.org/>

Catalyst Docker 镜像: <https://registry.hub.docker.com/u/rsrchboy/perl-catalyst-latest/>

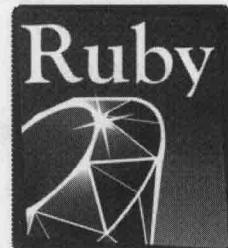
Catalyst Dockerfile: <https://registry.hub.docker.com/u/rsrchboy/perl-catalyst-latest/dockerfile/>

Catalyst 安装: [http://blog.sina.com.cn/s/blog\\_4aea5d890100ija9.html](http://blog.sina.com.cn/s/blog_4aea5d890100ija9.html)

## 13.6 Ruby

### 13.6.1 Ruby 技术栈

Ruby 是一种跨平台的、面向对象的、通用的开源动态脚本语言。Ruby 的设计受到 Perl、Smalltalk、Eiffel、Ada 和 Lisp 的影响。Ruby 支持多种编程范式，如函数编程、面向对象编程、CLI 交互式编程。Ruby 还有动态的数据类型系统（dynamic type system）和自动的内存管理。由于 Ruby 设计精良，语法简约，表达能力强，社区火爆的特点，所以在创业型和敏捷型项目中被大量使用，并有持续增长的趋势。



#### 使用官方镜像

我们可以使用 Docker 官方的 Ruby 镜像作为基础，在此之上进行必要的定制：

第一步，下载 Docker 官方的 Ruby 镜像：

```
$ sudo docker pull ruby
```

如果读者对 Ruby 的版本有要求，可以在以上命令中加入 Tag 标签，以便于在下一步的 Dockerfile 的 FROM 指令中明确 Ruby 版本号。官方镜像都有明确的标签列表，具体地址可以参见后面的“相关资源”部分。

第二步，在 Ruby 项目中创建一个 Dockerfile：

```
FROM ruby:2.1.2-onbuild
CMD ["/your-daemon-or-script.rb"]
```

将此文件放在 App 的根目录（与 Gemfile 同级）。注意，我们使用的官方镜像带有 onbuild 标签，也就意味着它包含了启动大部分 Ruby 项目所需的基本指令。在构建镜像的时候，Docker 会执行 COPY . /usr/src/app 以及 RUN bundle install。

第三步，构建自定义镜像：

```
$ sudo docker build -t my-ruby-app .
```

第四步，创建并运行此镜像：

```
$ sudo docker run -it --name my-running-script my-ruby-app
```

由于我们在构建时使用了带有 `onbuild` 标签的镜像，所以在 App 目录下需要有 `Gemfile.lock` 文件。这时可以在 App 的根目录运行以下命令：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/app -w /usr/src/app ruby:2.1.2
bundle install --system
```

如果读者只需要运行单个 Ruby 脚本，那么无需使用 Dockerfile 构建自定义镜像，而是通过以下命令直接使用官方 Ruby 镜像，带参数运行容器：

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp ruby:2.1.2 ruby your-daemon-or-script.rb
```

### 13.6.2 JRuby

JRuby 类似于 Python 的 Jython，一个可用于 Java 上运行 Ruby 的语言，支持 Java 的接口和类。



#### 使用官方镜像

我们可以使用 Docker 官方的 Ruby 镜像作为基础，在此基础上进行必要的定制。

第一步，下载 Docker 官方的 JRuby 镜像：

```
$ sudo docker pull jruby
```

如果读者对 JRuby 的版本有要求，可以在以上命令中加入 Tag 标签，以便于在下一步的 Dockerfile 的 `FROM` 指令中明确 JRuby 版本号。官方镜像都有明确的标签列表，具体地址可以参见后面的“相关资源”部分。

第二步，在 JRuby 项目中创建一个 Dockerfile：

```
FROM jruby:1.7.15-onbuild
CMD ["./your-daemon-or-script.rb"]
```

将此文件放在 App 的根目录（与 `Gemfile` 同级）。注意，我们使用的官方镜像带有 `onbuild` 标签，也就意味着它包含了启动大部分 JRuby 项目所需的基本指令。在构建镜像的时候，Docker 会执行 `COPY . /usr/src/app` 以及 `RUN bundle install`。

第三步，构建自定义镜像：

```
$ sudo docker build -t my-ruby-app .
```

第四步，创建并运行此镜像：

```
$ sudo docker run -it --name my-running-script my-ruby-app
```

由于我们在构建时使用了带有 `onbuild` 标签的镜像，所以在 App 目录下需要有 `Gemfile`。

lock 文件。这时可以在 App 的根目录运行以下命令：

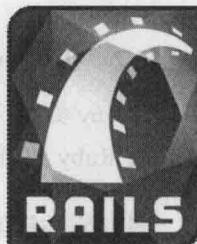
```
$ sudo docker run --rm -v "$(pwd)":/usr/src/app -w /usr/src/app jruby:1.7.15
bundle install --system
```

如果读者只需要运行单个 JRuby 脚本，那么无需使用 Dockerfile 构建自定义镜像，而是通过以下命令直接使用官方 JRuby 镜像，带参数运行容器：

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp jruby:1.7.15 jruby your-daemon-or-script.rb
```

### 13.6.3 Ruby on Rails

Rails 是使用 Ruby 语言编写的网页程序开发框架。Rails 的一些设计理念和机制比较创新和优雅，比较彻底地实现了面向对象编程，也比较满足敏捷开发的需要。很多其他语言的 Web MVC 框架都有所借鉴。



#### 使用官方镜像

我们可以使用 Docker 官方的 Rails 镜像作为基础，在此基础上进行必要的定制：

第一步，下载 Docker 官方的 Rails 镜像：

```
$ sudo docker pull rails
```

如果读者对 Rails 的版本有要求，可以在以上命令中加入 Tag 标签，以便于在下一步的 Dockerfile 的 FROM 指令中明确 Rails 版本号。官方镜像都有明确的标签列表，具体地址可以参见后面的“相关资源”部分。

第二步，在 Rails 项目中创建一个 Dockerfile：

```
FROM rails:onbuild
```

将此文件放在 App 的根目录（与 Gemfile 同级）。注意，我们使用的官方镜像带有 onbuild 标签，也就意味着它包含了启动大部分 JRuby 项目所需的基本指令。在构建镜像的时候，Docker 会执行 COPY ./usr/src/app，RUN bundle install，以及 EXPOSE 3000，并将默认的运行指令设为 rails server。

第三步，构建自定义镜像：

```
$ sudo docker build -t my-rails-app .
```

第四步，创建并运行此镜像：

```
$ sudo docker run --name some-rails-app -d my-rails-app
```

此时读者可以使用浏览器访问 <http://container-ip:3000>。

如果你需要在局域网的另一台机器上访问此容器中的 App，可以使用以下命令：

```
$ sudo docker run --name some-rails-app -p 8080:3000 -d my-rails-app
```

现在读者可以使用浏览器访问 `http://localhost:8080` 或者 `http://host-ip:8080`。

由于我们在构建时使用了带有 `onbuild` 标签的镜像，所以在 App 目录下需要有 `Gemfile.lock` 文件。这时可以在 App 的根目录运行以下命令：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/app -w /usr/src/app ruby:2.1.2
bundle install --system
```

### 13.6.4 Sinatra

Sinatra 是一个优雅地包装了 Web 开发的 DSL（领域特定语言）。用 Sinatra 只需 5 行代码即可实现一个简单的 hello world：

```
require 'rubygems'
require 'sinatra'

get '/' do
  'Hello World'
end
```



如果读者自定义镜像时基于系统标准镜像（如 Ubuntu），则可以参考以下安装命令：

```
$ gem install sinatra
$ ruby -rubygems hi.rb
```

#### 1. 使用 Docker Hub 镜像

读者可以基于自建的 SSHD 镜像，定制 Sinatra 镜像。如此一来，即可方便地使用 SSH 服务来访问此 Sinatra 镜像。当然，也可以直接到 Docker Hub 上搜索 Sinatra 镜像，直接下载：

```
$ sudo docker pull yoheimuta/docker-sinatra
```

#### 2. 定制镜像

如上文所述，笔者推荐使用 SSHD 镜像为基础来定制。当然，读者也可以使用以下 Dockerfile 来定制镜像。

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get -y install ruby ruby-dev build-essential redis-tools
RUN gem install --no-rdoc --no-ri sinatra json redis
```

```
RUN mkdir -p /opt/webapp
# 使用 4567 端口
EXPOSE 4567
# 设置默认命令
CMD ["/opt/webapp/bin/webapp"]
```

RubyGems 使用提示：目前搭建 RoR 环境过程中会遇到一些网络原因导致 rubygems.org 存放在 Amazon S3 > 上面的资源文件间歇性连接失败。有时候 gem install rack 或 bundle install 命令无法有效执行（可以用 gem install rails -V 来查看执行过程）。笔者推荐使用淘宝 RubyGems 镜像或者其他国内镜像站。

### 13.6.5 相关资源

Ruby 官网：<https://www.ruby-lang.org/>

Ruby Docker 官方源：[https://registry.hub.docker.com/\\_/ruby/](https://registry.hub.docker.com/_/ruby/)

Ruby Docker 官方源标签：[https://registry.hub.docker.com/\\_/ruby/tags/manage/](https://registry.hub.docker.com/_/ruby/tags/manage/)

淘宝 Ruby Gems：<https://ruby.taobao.org/>

JRuby 官网：<http://www.jruby.org/>

JRuby Docker 官方镜像：[https://registry.hub.docker.com/\\_/jruby/](https://registry.hub.docker.com/_/jruby/)

JRuby Docker 官方镜像标签：[https://registry.hub.docker.com/\\_/jruby/tags/manage/](https://registry.hub.docker.com/_/jruby/tags/manage/)

Rails 官网：<http://rubyonrails.org/>

Rails Docker 官方镜像：[https://registry.hub.docker.com/\\_/rails/](https://registry.hub.docker.com/_/rails/)

Rails Docker 官方镜像标签：[https://registry.hub.docker.com/\\_/rails/tags/manage/](https://registry.hub.docker.com/_/rails/tags/manage/)

Sinatra 官网：<http://www.sinatrarb.com/>

Sinatra Docker 镜像 1：<https://registry.hub.docker.com/u/yoheimuta/docker-sinatra/>

Sinatra Docker 镜像 2：<https://registry.hub.docker.com/u/gwjjeff/sinatra/>

## 13.7 JavaScript

### 13.7.1 JavaScript 技术栈

JavaScript 是一种弱类型的解释型动态脚本语言，内置支持类（面向对象编程）。它的解释器（JavaScript 引擎）是浏览器的一部分。JavaScript 广泛用于浏览器的各种前台业务逻辑，动态显示和 Ajax 请求。目前被许多创业团队喜爱的 SPA（Single-page Application）方案，就是充分利用了 JavaScript MVC 框架的优势，将敏捷开发发挥到极致。例如通过双向绑定之类的特性，可以在客户端完成大部分的业务逻辑，后台服务器只需要解析 Restful API 请求即可完成一个功能强大、体验丰



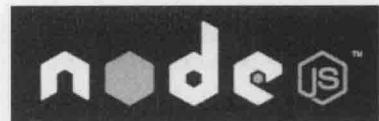
富的 Web App。

JavaScript 也可以通过其解释器运行在服务器端, Node.js 就是服务端 JavaScript 的主流技术方案, 并拥有优秀的性能和大量实践案例。

下面, 笔者将简述如何使用 Docker 搭建 Node.js 环境。

### 13.7.2 Node.js

在 Node.js 环境中, JavaScript 代码通过 V8 引擎运行于网络服务器。由于它优秀的高并发处理能力, Node 服务端在高并发技术方案的技术选型中与 PHP、Python、Perl、Ruby 平起平坐。



#### 1. 使用官方镜像

第一步, 安装 Node.js 的 Docker Hub 官方镜像:

```
$ sudo docker pull node
```

第二步, 读者可以使用上面安装的基础镜像, 构建自定义镜像。下面, 在 Node.js 项目中新建一个 Dockerfile:

```
FROM node:0.10-onbuild
# 使用 8888 端口替换原应用端口
EXPOSE 8888
```

读者可以通过修改 Expose 命令后面的数字来修改默认端口。

第三步, 使用此 Dockerfile 构建镜像:

```
$ sudo docker build -t my-nodejs-app .
```

第四步, 创建并运行 Node.js 容器:

```
$ sudo docker run -it --rm --name my-running-app my-nodejs-app
```

注意, 本镜像预设使用读者的项目中文件名为 package.json[注释] 的文件, 此文件包含项目依赖信息以及启动脚本 [注释]。

如果读者需要一个运行单个 Node.js 脚本的 Node 容器, 则无需通过书写 Dockerfile 构建镜像的方式。读者可以使用以下指令:

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp node:0.10 node your-daemon-or-script.js
```

#### 2. 定制镜像

如果读者需要定制 Node 镜像, 推荐读者基于一个带有 SSH 服务的 Docker 镜像进行改

造，这样可以方便使用 SSH 服务连接 Node 容器。当然，也可以参考以下 Dockerfile 来定制镜像：

```
# Node.js Dockerfile
#
# https://github.com/dockerfile/nodejs
#
# 下载基础镜像
FROM dockerfile/python

# 安装 Node.js 环境
RUN \
    cd /tmp && \
    wget http://nodejs.org/dist/node-latest.tar.gz && \
    tar xvzf node-latest.tar.gz && \
    rm -f node-latest.tar.gz && \
    cd node-v* && \
    ./configure && \
    CXX="g++ -Wno-unused-local-typedefs" make && \
    CXX="g++ -Wno-unused-local-typedefs" make install && \
    cd /tmp && \
    rm -rf /tmp/node-v* && \
    echo '\n# Node.js\nexport PATH="node_modules/.bin:$PATH"' >> /root/.bashrc

# 定义工作目录
WORKDIR /data

# 定义默认命令
CMD ["bash"]
```

### 13.7.3 Express

Express 作为 Node.js 的开发框架是目前最稳定、使用最广泛且官方推荐的 Web 开发框架。Express 简洁、灵活，它提供一系列强大的特性，帮助开发者创建各种 Web 和移动设备应用。Express 不对 Node.js 已有的特性进行二次抽象，而只是在其之上扩展了 Web 应用所需的功能。它拥有丰富的 HTTP 工具以及来自 Connect 框架的中间件。读者可以使用 Express 快速又简单地创建强健和友好的 API 服务。

#### 1. 使用官方镜像

第一步，安装 Node.js 的官方镜像，作为基础镜像：

```
$ sudo docker pull node
```

第二步，安装 Express 在 Docker Hub 上的镜像：

```
$ sudo docker pull otium360/express
```

第三步，此镜像预设情况是：1) 读者的 Express 项目有名为 `server.js` 的启动脚本。  
2) 读者的 Express 项目监听 8080 端口。

此镜像将会给 Express 项目添加 `/express/package.json` 文件（内含 NPM 的依赖关系，如下所示），然后将这些依赖包本地安装至 `/express` 文件夹：

- 1) connect
- 2) express
- 3) serve-static

此镜像的默认启动指令是：

```
CMD ['npm', 'start']
```

当然，读者也可以通过自定义 Dockerfile 的方式，进一步拓展镜像功能：

1) 由于此镜像会使用 `/express` 文件夹作为 Node/Express 应用的根目录，所以需要拷贝 `server.js` 文件以及所有的应用代码至此目录下。

2) 在应用的根目录新建一个 Dockerfile 文件：

```
# 下载基础镜像
FROM otium360/express

# 添加 Node 命令
ADD server.js /express/server.js
ADD www /express/www
```

`server.js` 文件内容如下：

```
'use strict';

var connect = require('connect');
var serveStatic = require('serve-static');

var app = connect();
app.use(serveStatic('www', {'index': ['index.html']}));
app.listen(8080);

console.log('MyApp is ready at http://localhost:8080');
```

3) 使用自定义 Dockerfile 构建镜像：

```
$ sudo docker build -t my-app-express /path/to/your/Dockerfile
```

4) 创建并运行此镜像：

```
$ sudo docker run -d -p 8080:8080 --name my-app-express my-app-express
```

## 2. 定制镜像

除了以上方法，读者也可以在 Docker Hub 自行搜索含 node 关键字的镜像，然后改造其 Dockerfile，也可以运行 Node 容器。笔者给出以下 Dockerfile，供读者参考：

```
# 下载基础镜像
FROM dockerfile/nodejs

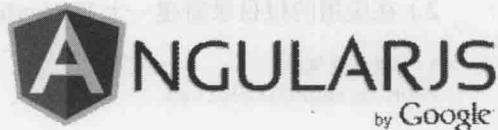
# 安装 NPM 管理器
ADD package.json /express/package.json
WORKDIR /express
RUN npm install

# 使用 8080 端口
EXPOSE 8080

# 设置默认运行命令
CMD ["npm", "start"]
```

### 13.7.4 AngularJS

AngularJS 是一个 JavaScript MVC 框架。AngularJS 很小，只有 60K，兼容主流浏览器，与 jQuery 配合良好。AngularJS 项目由 Google 维护，它是一款优秀的前端 JavaScript 框架，已经用于 Google 的多款产品当中。



笔者在此简述一下 AngularJS 的特性：

- 数据绑定：这可能是 AngularJS 最酷最实用的特性。它能够帮助你避免书写大量的初始代码从而节约开发时间。一个典型的 web 应用可能包含了 80% 的代码用来处理，查询和监听 DOM。数据绑定使得代码更少，你可以专注于你的应用。
- 模板：在 AngularJS 中，一个模板就是一个 HTML 文件。但是 HTML 的内容扩展了，包含了很多帮助你映射 model 到 view 的内容。
- MVC：针对客户端应用开发 AngularJS 吸收了传统的 MVC 基本原则。MVC 或者 Model-View-Controller 设计模式针对不同的人可能意味不同的东西。AngularJS 并不执行传统意义上的 MVC，更接近于 MVVM (Model-View-ViewModel)。
- 依赖注入 (Dependency Injection，即 DI)：AngularJS 拥有内建的依赖注入子系统，可以帮助开发人员更容易的开发，理解和测试应用。
- Directives (指令)：指令是我个人最喜欢的特性。你是不是也希望浏览器可以做点儿有意思的事情？那么 AngularJS 可以做到。指令可以用来创建自定义的标签。它们可以用来装饰元素或者操作 DOM 属性。

## 使用 Docker Hub 镜像

第一步，新建 Angular 项目文件夹，并通过 CLI 进入此目录：

```
mkdir -p ~/Projects/Personal/nameOfProject
cd ~/Projects/Personal/nameOfProject
```

第二步，直接拷贝或者通过 Git 克隆一份项目代码：

```
$ git clone https://github.com/username/nameOfProject.git
```

第三步，安装 Docker Hub 上的 Angular 镜像：

```
$ sudo docker pull sesteva/grunt-angular
```

第四步，创建并运行 Angular 容器：

```
$ sudo docker run --name nameOfProject -p 9000:9000 -v ~/Projects/Personal/
nameOfProject:/home/project -i -t sesteva/grunt-angular
```

当然，读者也可以通过修改 Dockerfile 来拓展和自定义镜像。笔者在此提供以下 Dockerfile，供读者参考：

```
# 下载 Yeoman 基础镜像
FROM sesteva/yeoman

# 使用 NPM 安装 Angular 和 CoffeeScript
RUN npm install -g generator-angular coffee-script

# 创建目录
RUN mkdir -p /home/project

CMD cd /home/project && \
  npm install && \
  bower install --allow-root && \
  grunt serve

# 设定工作目录
WORKDIR /home/project

# 使用 9000 端口
EXPOSE 9000
```

### 13.7.5 相关资源

JavaScript 入门：<http://www.w3schools.com/js/>

Node.js 官网：<http://www.nodejs.org/>

Node.js Docker Hub 官方镜像：[https://registry.hub.docker.com/\\_/node/](https://registry.hub.docker.com/_/node/)

Node.js Docker Hub 官方镜像标签：[https://registry.hub.docker.com/\\_/node/tags/manage/](https://registry.hub.docker.com/_/node/tags/manage/)

Express 官网: <http://expressjs.com/>

Express Docker Hub 镜像: <https://registry.hub.docker.com/u/otium360/express/>

Express Dockerfile: <https://registry.hub.docker.com/u/otium360organization/express/dockerfile/>

package.json: <https://www.npmjs.org/doc/files/package.json.html>

启动脚本: <https://www.npmjs.org/doc/misc/npm-scripts.html#default-values>

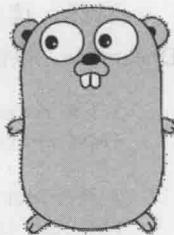
Angular 官网: <http://angularjs.org/>

Angular Docker 镜像: <https://registry.hub.docker.com/u/sesteva/grunt-angular/>Angular Dockerfile: <https://registry.hub.docker.com/u/sesteva/grunt-angular/dockerfile/>

## 13.8 Go

### 13.8.1 Go 技术栈

Go 语言（也称 Golang）是一个由 Google 主导研发的编程语言。它的语法清晰明了，设计精良，拥有一些先进的特性，还有一个庞大的标准库。Go 的基本设计理念是：编译效率、运行效率和开发效率要三者兼顾。使用 Go 开发，要让开发人员感觉到 Python 的便利，C/C++ 的运行效率，以及小到可以被忽略的编译时间。笔者在此简述一下 Go 语言的特性：



- 编译，静态类型语言。由此可以提供满足对运行效率敏感的系统级应用。
- 垃圾回收，去除复杂的内存释放工作。
- 简洁的符号和语法，极力减少开发人员输入的字符数。
- 平坦的类型系统，去除了复杂的继承关系。使用结构化类型系统（Structural type system），既简化了事前设计工作，也为未来增加抽象层提供了非侵入式的解决方法。
- 基于 CSP 模型的并行，简化了并发结构之间的通信和数据共享。为多核时代的程序开发打好基础。比线程更轻量的 goroutine，让一个线程可以执行多个并发结构。不必使用异步通信，就足以达到线程池与 select/poll/epoll 的效果。极大简化了多连接的开发。
- 使用一套简单的规范，开发人员不必再单独编写脚本指定依赖关系和编译流程。仅仅使用代码本身和 go 工具链，就可以处理各种依赖关系。写完代码，一条命令，自动下来各种依赖，直接编译 / 安装。无需 make、autoconf、automake、setup.py 等工具支持。

#### 使用官方镜像

将 Go 代码运行在 Docker 容器中的最简方法，就是将 Go 编译指令写入 Dockerfile 中，然后使用此 Dockerfile 构建自定义镜像，最后直接运行此镜像，即可启动 Go 程序。

具体步骤如下。

第一步，首先安装 Go 的 Docker Hub 官方镜像作为基础镜像：

```
$ sudo docker pull golang
```

如果读者对 Go 的版本有要求，可以在以上命令中加入 Tag 标签，以便于在下一步的 Dockerfile 的 FROM 指令中明确 Go 版本号。官方镜像都有明确的标签列表，具体地址可以参见后面的“相关资源”部分。

第二步，在 Dockerfile 中，加入读者需要执行的 Go 编译命令：

```
FROM golang:1.3.1-onbuild
```

注意，我们使用的官方镜像带有 onbuild 标签，也就意味着它包含了启动大部分 Go 项目所需的基本指令。在构建镜像的时候，Docker 会执行 COPY ./usr/src/app, RUN go get -d -v，以及 go install -v。

在使用此镜像，不带参数运行 Go 容器时，此会执行 CMD ["app"] 指令。

第三步，使用此 Dockerfile 构建镜像：

```
$ sudo docker build -t my-golang-app .
```

第四步，创建并运行 Go 容器：

```
$ sudo docker run -it --rm --name my-running-app my-golang-app
```

在 Docker 容器中编译 Go 项目如果读者需要在容器中编译 Go 代码，但是不需要在容器中运行它，可以执行：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp golang:1.3.1 go build -v
```

此指令会将读者的 Go 项目文件夹作为 Docker 数据卷挂载至 Docker，并作为运行目录。然后 Docker 会执行 go build，在工作目录中编译代码，输出可执行文件至 myapp。

如果此项目有 Makefile，那么可以在容器中执行：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp golang:1.3.1 make
```

在 Docker 容器中交叉编译 Go 项目如果读者需要在常用的 linux\amd64 架构之外的其他架构的平台编译 Go 应用，如 windows\386。可以在指令中加入 cross 标签：

```
$ sudo docker run --rm -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp -e GOOS=windows -e GOARCH=386 golang:1.3.1-cross go build -v
```

读者也可以使用以下命令将 Go 程序一次性编译至多个平台：

```
$ sudo docker run --rm -it -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp golang:1.3.1-cross bash
$ for GOOS in darwin linux; do
```

```
> for GOARCH in 386 amd64; do
>   go build -v -o myapp-$GOOS-$GOARCH
> done
> done
```

如果读者需要方便地运行已有 Go 项目的代码，特别是 Web 项目，那么笔者推荐读者使用一个内含 SSH 服务的镜像，以此为基础定制 Go 镜像。这样可以方便地使用 SSH 服务访问 Go 容器中的站点。

### 13.8.2 Beego

Beego 是一个使用 Go 的思维来帮助开发者构建并开发 Go 应用程序的开源框架。Beego 使用 Go 开发，思路来自于 Tornado，路由设计来源于 Sinatra。



笔者在此简述一下 Beego 框架的特性：

- 简单化：RESTful 支持、MVC 模型，可以使用 bee 工具快速地开发应用，包括监控代码修改进行热编译、自动化测试代码以及自动化打包部署。
- 智能化：支持智能路由、智能监控，可以监控 QPS、内存消耗、CPU 使用，以及 goroutine 的运行状况，让线上应用尽在掌握。
- 模块化：Beego 内置了强大的模块，包括 Session、缓存操作、日志记录、配置解析、性能监控、上下文操作、ORM 模块、请求模拟等强大的模块，足以支撑你任何的应用。
- 高性能：Beego 采用了 Go 原生的 http 包来处理请求，goroutine 的并发效率足以应付大流量的 Web 应用和 API 应用，目前已经应用于大量高并发的产品中。

#### 1. 准备工作

第一步，下载安装：

```
$ go get github.com/astaxie/beego
```

第二步，创建文件 hello.go：

```
package main

import "github.com/astaxie/beego"

func main() {
    beego.Run()
}
```

第三步，编译运行：

```
$ go build -o hello hello.go
$ ./hello
```

第四步，打开浏览器并访问 <http://localhost:8080>。恭喜！第一个 Beego 项目已经成功的构建了。

读者可以查阅开发文档以进行深入学习。

## 2. 使用 Docker Hub 镜像

读者可以使用 Docker Hub 提供的第三方 Beego 镜像，下载后直接运行即可：

```
$ sudo docker pull cloudcube/beego
```

## 3. 定制镜像

如果需要定制 Beego 镜像，则如前文所述，建议读者基于内含 SSH 的镜像进行定制。当然，读者也可以参考以下 Dockerfile 构建自定义镜像：

```
FROM cloudcube/golang

# system update
RUN apt-get update

# install beego
RUN go get github.com/astaxie/beego

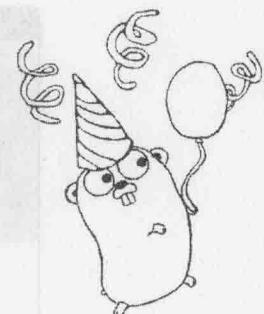
# install bee
RUN go get github.com/cloudcube/bee
```

### 13.8.3 Revel

Revel 是一个高生产力的 Go 语言 Web 框架。Revel 从 Rails 和 Play! 中吸收了许多成熟的设计思想。Revel 支持 MVC 设计模式，它是一个轻量级的高效 Web 开发框架。Revel+Xorm (ORM) 的技术栈可以方便的构建各种基于 Go 语言的 Web 服务。

笔者在此简述一下 Revel 框架的特性：

- 热编译：编辑、保存和刷新时，Revel 自动编译代码和模板，如果代码编译错误，会给出一个错误提示，同时捕捉运行期错误。
- 全栈功能：Revel 支持路由、参数解析、验证、session/flash、模板、缓存、计划任务、测试、国际化等功能。
- 高性能：Revel 基于 Go HTTP server 构建。这是 techempower 发布的最新评测结果。在各种不同的场景下进行了多达三到十次的请求负载测试。



### 1. 准备工作

第一步，安装 Revel 框架：

```
$ go get github.com/revel/revel
```

第二步，安装 Revel 命令行工具：

```
$ go get github.com/revel/cmd/revel
```

第三步，创建 Revel 应用：

```
$ revel new myapp
```

生成的目录结构如下：

myapp		项目根目录
	app	MVC 框架目录
	controllers	控制器目录
	init.go	
	models	模型目录
	routes	
	tmp	
	views	视图目录
	conf	
	app.conf	配置文件
	routes	路由文件
	messages	国际化目录
	public	静态文件目录
	tests	

第四步，打开浏览器访问 <http://localhost:9000>，如图 13-3 所示。

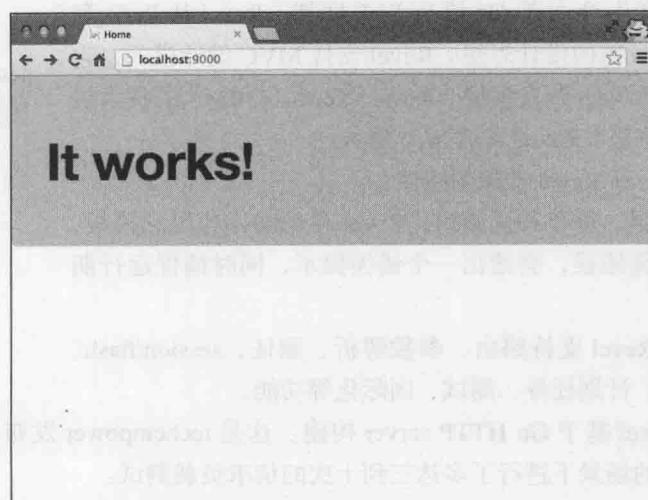


图 13-3 Revel 界面

## 2. 使用 Docker Hub 镜像

读者可以使用 Docker Hub 中提供的第三方 Revel 镜像，下载并直接运行。笔者列出以下镜像供读者参考：

```
$ sudo docker pull taddev/revel-base
```

如果读者需要运行已有的 Revel 站点，或者需要自定义启动流程，则笔者推荐使用内含 SSH 服务的镜像，以此为基础定制 Revel 镜像。当然，读者也可以参考以下 Dockerfile 构建自定义镜像：

```
# 安装基础镜像
FROM google/golang

# 获取 Revel
RUN go get github.com/revel/cmd/revel
```

### 13.8.4 Martini

Martini 是一个优雅的 Go 语言 Web 框架（Classy web framework for Go），它是专门为使用 Go 语言编写模块化 Web 应用而生的。



笔者在此简述以下 Martini 框架的特性：

- 使用极其简单。
- 无侵入式的设计。
- 很好地与其他的 Go 语言包协同使用。
- 超赞的路径匹配和路由。
- 模块化的设计——容易插入功能件，也容易将其拔出来。
- 已有很多的中间件可以直接使用。
- 框架内已拥有很好的开箱即用的功能支持。
- 完全兼容 http.HandlerFunc 接口。
- 更多中间件和功能组件，可参考代码仓库 :<http://github.com/martini-contrib>。

## 1. 准备工作

在读者安装了 GO 语言并设置了自己的 GOPATH 之后，创建自己的 .go 文件，这里我们假设它的名字叫做 server.go：

```
package main

import "github.com/go-martini/martini"

func main() {
    m := martini.Classic()
```

```
m.Get("/", func() string {
    return "Hello world!"
})
m.Run()
}
```

然后，安装 Martini 的包（注意 Martini 需要 Go 语言 1.1 或者以上的版本支持）：

```
$ go get github.com/go-martini/martini
```

最后，运行 server.go：

```
$ go run server.go
```

此时已经启动了一个 Martini 的 Web 服务，地址是 :localhost:3000。

## 2. 使用 Docker Hub 镜像

读者可以直接使用 Docker Hub 中提供的第三方 Martini 镜像，下载镜像并直接运行。笔者给出以下命令供读者参考：

```
$ sudo docker pull lgsd/docker-martini
```

## 3. 定制镜像

如果读者需要加载已有的 Martini 站点，或者需要定制启动流程，则笔者推荐使用内含 SSH 服务的镜像，以此为基础进行定制，这样可以方便地使用 SSH 服务访问 Martini 容器中的站点。当然，读者也可以参考以下 Dockerfile 构建自定义镜像：

```
# Dockerfile for Martini/GOLANG 1.2

# 下载基础镜像
FROM lgsd/saucy

# 更新系统
RUN sed 's/main$/main universe/' -i /etc/apt/sources.list && \
    apt-get -qq update && \
    apt-get -y upgrade && \
    apt-get install -y wget tar ca-certificates git

# 下载 Go 语言源码安装包
# into /usr/local, creating a Go tree in /usr/local/go
RUN wget https://go.googlecode.com/files/go1.2.linux-amd64.tar.gz && \
    tar -C /usr/local -xzf go1.2.linux-amd64.tar.gz && \
    rm go1.2.linux-amd64.tar.gz

# 安装完成后使用 apt 进行清理
RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
```

```

# 设置环境变量
ENV GOPATH /go
ENV PATH $PATH:/usr/local/go/bin:$GOPATH/bin

# 安装 Martini 安装包
RUN go get github.com/codegangsta/martini

# 根据官方启动流程，创建 server.go 文件，并验证安装结果
ENV FILE $HOME/server.go
RUN echo 'package main' > $FILE && \
    echo 'import "github.com/codegangsta/martini"' >> $FILE && \
    echo 'func main() {' >> $FILE && \
    echo '    m := martini.Classic()' >> $FILE && \
    echo '    m.Get("/", func() string {' >> $FILE && \
    echo '        return "Hello world!"' >> $FILE && \
    echo '    })' >> $FILE && \
    echo '    m.Run()' >> $FILE && \
    echo '}' >> $FILE

# 使用 3000 端口运行 Martini
localhost:3000
EXPOSE 3000

# 设置默认启动命令
CMD ["go", "run", "server.go"]

```

### 13.8.5 相关资源

Go 语言官网：<https://golang.org/>

Go Docker 官方镜像：[https://registry.hub.docker.com/\\_/golang/](https://registry.hub.docker.com/_/golang/)

Go Docker 官方镜像标签：[https://registry.hub.docker.com/\\_/golang/tags/manage/](https://registry.hub.docker.com/_/golang/tags/manage/)

Docker Hub 中 Google 提供的 Go 镜像：

<https://registry.hub.docker.com/u/google/golang-hello/>

<https://registry.hub.docker.com/u/google/golang-runtime/>

<https://registry.hub.docker.com/u/google/golang/>

Beego 官网：<http://beego.me/>

Beego Docker Hub 镜像：<https://registry.hub.docker.com/u/cloudcube/beego/>

Boogo Dockerfile：<https://registry.hub.docker.com/u/cloudcube/beego/dockerfile/>

Revel 官网：<http://www.gorevel.cn/>

Revel 中国官网：<http://gorevel.cn/>

Revel Docker Hub 镜像：<https://registry.hub.docker.com/u/taddev/revel-base/>

Revel Dockerfile：<https://registry.hub.docker.com/u/taddev/revel-base/dockerfile/>

Martini 官网：<http://martini.codegangsta.io/>

Martini Docker Hub 镜像: <https://registry.hub.docker.com/u/lgsd/docker-martini/>

Martini Dockerfile: <https://registry.hub.docker.com/u/lgsd/docker-martini/dockerfile/>

### 13.9 本章小结

在本章中，笔者主要介绍了如何使用 Docker 搭建主流编程语言及其常用开发框架的 Docker 环境。由于时间仓促且水平有限，所以笔者重点阐述了 PHP 语言的 Docker 环境。其他编程语言的自定义 Docker 环境与 PHP 环境下的操作流程大同小异：均是基于内含 SSH 服务的镜像，定制 Dockerfile，随后构建并运行镜像即可。这样读者可以方便地使用 SSH 服务访问自定义容器中的站点或程序。当然，读者也可以直接使用官网的编程语言镜像，通过适当的配置也可以构建自定义镜像并正常使用容器。如果读者对本章内容有任何疑问，可以前往 DockerPool 社区 (<http://dockerpool.com>)。由于编程语言的更新迭代比较频繁，所以本章内容会在 DockerPool 社区持续更新，敬请关注。

## 第 14 章 使用私有仓库

在使用 Docker 一段时间后，往往你会发现手头积累了大量的自定义镜像文件，这些文件通过公有仓库进行管理并不十分方便；另外有时候只是希望内部用户之间进行分享。

在这种情况下，就有必要搭建一个本地的私有仓库服务器。在第一部分中，笔者讲解了快速使用 docker-registry 镜像搭建一个私有仓库的方法。本章将具体介绍 docker-registry 开源项目的使用细节，并通过具体案例来展示如何搭建一个本地的仓库服务器。

在搭建完本地的私有仓库服务器后，接下来会介绍如何编写脚本来批量上传本地镜像到私有仓库中。最后会对 docker-registry 项目的配置文件以及各种选项进行剖析。

### 14.1 使用 docker-registry

docker-registry 是一个基于 Python 的开源项目，可以用于构建私有的镜像注册服务器。官方仓库中也提供了 docker-registry 的镜像，因此用户可以通过容器运行和源码安装两种方式来使用 docker-registry。

#### 基于容器运行

在第一部分中，笔者简单介绍了基于容器运行 docker-registry 的过程。首先，获取并运行官方 registry 镜像：

```
$ sudo docker run -d -p 5000:5000 registry
```

启动后比较关键的参数是指定配置文件和仓库存储路径。

通过如下命令，可以指定本地路径（如 /home/user/registry-conf）下的配置文件：

```
$ sudo docker run -d -p 5000:5000 -v /home/user/registry-conf:/registry-conf -e DOCKER_REGISTRY_CONFIG=/registry-conf/config.yml registry
```

通过 -v 参数来配置仓库路径。例如下面的例子将镜像存储到本地 /opt/data/registry 目录：

```
$ sudo docker run -d -p 5000:5000 -v /opt/data/registry:/tmp/registry registry
```

## 本地安装运行

有时候需要本地运行仓库服务，可以通过安装包或源码方式进行。

对于 Ubuntu 或 CentOS 行发行版，可以直接通过源安装。

### □ Ubuntu 版的安装：

```
$ sudo apt-get install -y build-essential python-dev libevent-dev python-pip liblzma-dev
$ sudo pip install gunicorn pyyaml flask flask-cors rsa
$ sudo pip install docker-registry
```

### □ CentOS 版的安装：

```
$ sudo yum install -y python-devel libevent-devel python-pip gcc xz-devel
$ sudo pip install gunicorn pyyaml flask flask-cors rsa gevent
$ sudo python-pip install docker-registry
```

也可以从 docker-registry(<https://github.com/docker/docker-registry>) 项目下载源码进行安装：

```
$ sudo apt-get install build-essential python-dev libevent-dev python-pip
libssl-dev liblzma-dev libffi-dev
$ git clone https://github.com/docker/docker-registry.git
$ cd docker-registry
```

然后基于样例配置创建配置文件：

```
$ cp config/config_sample.yml config/config.yml
```

修改 local 模板段的 storage\_path 到本地的存储仓库的路径，例如：opt/data/registry。

```
local: &local
<<: *common
storage: local
storage_path: _env:STORAGE_PATH:/opt/data/registry
```

然后执行安装操作：

```
$ sudo python setup.py install
```

对于通过软件包方式安装的，配置文件一般需要放在 /usr/local/lib/python2.7/dist-packages/docker\_registry/config/config.yml。

此时，可以通过下面的命令来启动：

```
$ sudo gunicorn --access-logfile /var/log/docker-registry/access.log --error-logfile
/var/log/docker-registry/server.log -k gevent --max-requests 100 --graceful-
timeout 3600 -t 3600 -b 127.0.0.1:5000 -w 1 docker_registry.wsgi:application
```

此时使用访问本地的 5000 端口，看到输出 docker-registry 的版本信息说明运行成功：

```
$ sudo curl 127.0.0.1:5000
"docker-registry server (dev) (v0.8.1)"
```

## 配置服务脚本

一般通过服务脚本来管理 registry 服务会更加方便，以 Ubuntu 14.04 系统为例。

首先，创建 /etc/init/docker-registry.conf 文件，内容为：

```
description "Docker Registry"

start on runlevel [2345]
stop on runlevel [016]

respawn
respawn limit 10 5

script
exec gunicorn --access-logfile /var/log/docker-registry/access.log --error-logfile
/var/log/docker-registry/server.log -k gevent --max-requests 100 --graceful-
timeout 3600 -t 3600 -b localhost:15000 -w 8 docker_registry.wsgi:application
end script
```

然后，执行 service docker-registry start，将在本地的 15000 端口启动 registry 服务。

## 14.2 用户认证

通常在生产场景中，对私有仓库还需要进行访问代理和提供认证和用户管理。

### 配置 Nginx 代理

使用 Nginx 来代理 registry 服务的原理十分简单，在上一节中，我们让 registry 服务监听

在 127.0.0.1:15000，这意味着只允许本机才能通过 15000 端口访问到，其他主机是无法访问到的。

为了让其他主机访问到，可以通过 Nginx 监听在对外地址的 5000 端口，当外部访问请求到达 5000 端口时，内部再将请求转发到本地的 15000 端口。

首先，安装 Nginx。

```
$ sudo apt-get -y install nginx
```

在 /etc/nginx/sites-available/ 目录下，创建新的站点配置文件 /etc/nginx/sites-available/docker-registry.conf，代理本地的 5000 端口转发到 15000 端口。

配置文件内容为：

```
# 本地的 registry 服务监听在 15000 端口
upstream docker-registry {
    server localhost:15000;
}

# 代理服务器监听在 5000 端口
server {
    listen 5000;
    server_name private-registry-server.com;

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

    proxy_set_header Host      $http_host;    # required for Docker client sake
    proxy_set_header X-Real-IP $remote_addr; # pass on real client IP

    client_max_body_size 0; # disable any limits to avoid HTTP 413 for large image
    uploads

    chunked_transfer_encoding on;

    location / {
        # 配置转发对于 / 的访问请求到 registry 服务
        proxy_pass http://docker-registry;
    }
    location /_ping {
        # 配置转发对于 /ping 的访问请求到 registry 服务
        auth_basic off;
        proxy_pass http://docker-registry;
    }
    location /v1/_ping {
        # 配置转发对于 /v1/ping 的访问请求到 registry 服务
        auth_basic off;
        proxy_pass http://docker-registry;
    }
}
```

}

建立配置文件软连接，放到 /etc/nginx/sites-enabled/ 下面，让 Nginx 启用它，最后重启 Nginx 服务。

```
$ sudo ln -s /etc/nginx/sites-available/docker-registry.conf /etc/nginx/sites-enabled/docker-registry.conf
$ service nginx restart
```

之后，可以通过上传镜像来测试服务是否正常。测试上传本地的 ubuntu:latest 镜像：

```
$ sudo docker tag ubuntu:14.04 127.0.0.1:5000/ubuntu:latest
$ sudo docker push 127.0.0.1:5000/ubuntu:latest
```

## 添加用户认证

公共仓库 DockerHub 是通过注册索引（index）服务来实现的。由于 index 服务并没有完善的开源实现，在这里介绍基于 Nginx 代理的用户访问管理方案。

Nginx 支持基于用户名和密码的访问管理。

首先，在配置文件的 location/ 字段中添加两行。

```
...
location / {
    # let Nginx know about our auth file
    auth_basic "Please Input username/password";
    auth_basic_user_file docker-registry-htpasswd;

    proxy_pass http://docker-registry;
}
...
```

auth\_basic "Please Input username/password"; 行说明启用认证服务，不通过的请求将无法转发。

auth\_basic\_user\_file docker-registry-htpasswd; 指定了验证的用户名密码存储文件为本地（/etc/nginx/ 下）的 docker-registry-htpasswd 文件。

docker-registry-htpasswd 文件中存储用户名密码的格式为每行放一个用户名、密码对：

```
...
user1:password1
user2:password2
...
```

需要注意的是，密码字段存储的并不是明文，而是使用 crypt 函数加密过的字符串。

要生成加密后的字符串，可以通过 htpasswd 工具，首先安装 apache2-utils：

```
$ sudo aptitude install apache2-utils -y
```

创建用户 user1，并添加密码。

例如，如下的操作会创建 /etc/nginx/docker-registry-htpasswd 文件来保存用户名和加密后的密码信息，并创建 user1 和对应密码。

```
$ sudo htpasswd -c /etc/nginx/docker-registry-htpasswd user1
$ New password:
$ Re-type new password:
$ Adding password for user user1
```

添加更多用户，可以重复上面的命令（密码文件存在后，不需使用 -c 选项重新创建）。

最后，重新启动 Nginx 服务。

```
$ sudo service nginx restart
```

此时，通过浏览器访问本地的服务 `http://127.0.0.1:5000/v1/search`，会弹出对话框，提示需要输入用户名和密码。

通过命令行访问，需要在地址前面带上用户名和密码才能正常返回：

```
$ curl USERNAME:PASSWORD@localhost:5000/v1/search
```

### 14.3 使用私有仓库批量上传镜像

在第一部分对 Docker 私有仓库的讲解中，我们介绍了如何使用本地私有仓库进行上传、下载等操作。有时候，本地镜像很多，逐个打标记上传将十分浪费时间。这里我们给出两个自动化脚本，来快速完成对大量镜像的上传操作。

#### 批量上传指定镜像

可以使用下面的 `push_images.sh` 脚本，批量上传本地的镜像到注册服务器中，默认是本地注册服务器 `127.0.0.1:5000`，用户可以通过修改 `registry=127.0.0.1:5000` 这行来指定目标注册服务器：

```
#!/bin/sh

# This script will upload the given local images to a registry server ($registry
# is the default value).
# See: https://github.com/yeasy/docker_practice/blob/master/_local/push_images.sh
# Usage: push_images image1 [image2...]
# Author: yeasy@gmail.com
# Create: 2014-09-23

#The registry server address where you want push the images into
registry=127.0.0.1:5000
```

```
### DO NOT MODIFY THE FOLLOWING PART, UNLESS YOU KNOW WHAT IT MEANS ###
echo_r () {
    [ $# -ne 1 ] && return 0
    echo -e "\033[31m\$1\033[0m"
}
echo_g () {
    [ $# -ne 1 ] && return 0
    echo -e "\033[32m\$1\033[0m"
}
echo_y () {
    [ $# -ne 1 ] && return 0
    echo -e "\033[33m\$1\033[0m"
}
echo_b () {
    [ $# -ne 1 ] && return 0
    echo -e "\033[34m\$1\033[0m"
}
usage () {
    sudo docker images
    echo "Usage: $0 registry1:tag1 [registry2:tag2...]"
}

[ $# -lt 1 ] && usage && exit
echo_b "The registry server is $registry"
```

```
for image in "$@"
do
    echo_b "Uploading $image..."
    sudo docker tag $image $registry/$image
    sudo docker push $registry/$image
    sudo docker rmi $registry/$image
    echo_g "Done"
done
```

建议把脚本放到本地的可执行路径下，例如 /usr/local/bin/ 下面。

然后添加可执行权限，就可以使用该脚本了：

```
$ sudo chmod a+x /usr/local/bin/push_images.sh
```

例如，推送本地的 ubuntu:latest 和 centos:centos7 两个镜像到本地仓库：

```
$ ./push_images.sh ubuntu:latest centos:centos7
The registry server is 127.0.0.1
Uploading ubuntu:latest...
The push refers to a repository [127.0.0.1:5000/ubuntu] (len: 1)
Sending image list
Pushing repository 127.0.0.1:5000/ubuntu (1 tags)
```

```

Image 511136ea3c5a already pushed, skipping
Image bfb8b5a2ad34 already pushed, skipping
Image c1f3bdbd8355 already pushed, skipping
Image 897578f527ae already pushed, skipping
Image 9387bcc9826e already pushed, skipping
Image 809ed259f845 already pushed, skipping
Image 96864a7d2df3 already pushed, skipping
Pushing tag for rev [96864a7d2df3] on {http://127.0.0.1:5000/v1/repositories/ubuntu/tags/latest}
Untagged: 127.0.0.1:5000/ubuntu:latest
Done
Uploading centos:centos7...
The push refers to a repository [127.0.0.1:5000/centos] (len: 1)
Sending image list
Pushing repository 127.0.0.1:5000/centos (1 tags)
Image 511136ea3c5a already pushed, skipping
34e94e67e63a: Image successfully pushed
70214e5d0a90: Image successfully pushed
Pushing tag for rev [70214e5d0a90] on {http://127.0.0.1:5000/v1/repositories/centos/tags/centos7}
Untagged: 127.0.0.1:5000/centos:centos7
Done

```

上传后，查看本地镜像，会发现上传中创建的临时标签也同时被清理了。

## 上传本地所有镜像

在 push\_images 工具的基础上，还可以进一步的创建 push\_all 工具，来上传本地所有镜像：

```

#!/bin/sh
# This script will upload all local images to a registry server ($registry is
the default value).
# This script requires the push_images, which can be found at https://github.
com/yeasy/docker_practice/blob/master/_local/push_images.sh
# Usage: push_all
# Author: yeasy@gmail.com
# Create: 2014-09-23

for image in `sudo docker images|grep -v "REPOSITORY"|grep -v "<none>"|awk
'{print $1":"$2}'`'
do
    push_images.sh $image
done

```

同样，读者把它放在 /usr/local/bin/ 下面，并添加可执行权限，之后就可以通过 push\_all 命令来上传本地所有镜像到本地私有仓库了。



## 14.4 仓库配置文件

Docker 的 Registry 利用配置文件提供了一些仓库的模板 (flavor), 用户可以直接使用它们来进行开发或生产部署。

我们将以下面的示例配置为例, 来介绍如何使用仓库配置文件来管理私有仓库。

### 示例配置

```
#All other flavors inherit the 'common' config snippet
common: &common
  issue: '"docker-registry server"'
  # 默认记录 info 和以上级别的日志
  loglevel: _env:LOGLEVEL:info
  # 是否启用 debug 模式
  debug: _env:DEBUG:false
  # 默认是 standalone 模式, 不查询 index 服务
  standalone: _env:STANDALONE:true
  # 非 standalone 模式下的 index 服务默认是 index.docker.io
  index_endpoint: _env:INDEX_ENDPOINT:https://index.docker.io
  # 默认不启用存储转向
  storage_redirect: _env:STORAGE_REDIRECT
  # 非 standalone 模式下启用基于口令串的认证
  disable_token_auth: _env:DISABLE_TOKEN_AUTH
  # 默认不使用特权
  privileged_key: _env:PRIVILEGED_KEY
  # 搜索后端支持
  search_backend: _env:SEARCH_BACKEND
  # SQLite 搜索后端数据库地址
  sqlalchemy_index_database: _env:SQLALCHEMY_INDEX_DATABASE:sqlite:///tmp/
docker-registry.db

  # 默认不启用镜像服务
  mirroring:
    source: _env:MIRROR_SOURCE # https://registry-1.docker.io
    source_index: _env:MIRROR_SOURCE_INDEX # https://index.docker.io
    tags_cache_ttl: _env:MIRROR_TAGS_CACHE_TTL:172800 # seconds

  cache:
    host: _env:CACHE_REDIS_HOST
    port: _env:CACHE_REDIS_PORT
    db: _env:CACHE_REDIS_DB:0
    password: _env:CACHE_REDIS_PASSWORD

  # 访问远端存储后端时, 配置 LRU 缓存
  cache_lru:
    host: _env:CACHE_LRU_REDIS_HOST
    port: _env:CACHE_LRU_REDIS_PORT
    db: _env:CACHE_LRU_REDIS_DB:0
```

```

password: _env:CACHE_LRU_REDIS_PASSWORD

# 发生异常时发送邮件通知
email_exceptions:
  smtp_host: _env:SMTP_HOST
  smtp_port: _env:SMTP_PORT:25
  smtp_login: _env:SMTP_LOGIN
  smtp_password: _env:SMTP_PASSWORD
  smtp_secure: _env:SMTP_SECURE:false
  from_addr: _env:SMTP_FROM_ADDR:docker-registry@localdomain.local
  to_addr: _env:SMTP_TO_ADDR:noise+dockerregistry@localdomain.local

# 启用 bugsnag
bugsnag: _env:BUGSNAG

# CORS 支持, 默认未启用
cors:
  origins: _env:CORS_ORIGINS
  methods: _env:CORS_METHODS
  headers: _env:CORS_HEADERS:[Content-Type]
  expose_headers: _env:CORS_EXPOSE_HEADERS
  supports_credentials: _env:CORS_SUPPORTS_CREDENTIALS
  max_age: _env:CORS_MAX_AGE
  send_wildcard: _env:CORS_SEND_WILDCARD
  always_send: _env:CORS_ALWAYS_SEND
  automatic_options: _env:CORS_AUTOMATIC_OPTIONS
  vary_header: _env:CORS_VARY_HEADER
  resources: _env:CORS_RESOURCES

local: &local
<<: *common
storage: local
storage_path: _env:STORAGE_PATH:/tmp/registry

s3: &s3
<<: *common
storage: s3
s3_region: _env:AWS_REGION
s3_bucket: _env:AWS_BUCKET
boto_bucket: _env:AWS_BUCKET
storage_path: _env:STORAGE_PATH:/registry
s3_encrypt: _env:AWS_ENCRYPT:true
s3_secure: _env:AWS_SECURE:true
s3_access_key: _env:AWS_KEY
s3_secret_key: _env:AWS_SECRET
boto_host: _env:AWS_HOST
boto_port: _env:AWS_PORT
boto_calling_format: _env:AWS_CALLING_FORMAT

# Ceph 对象网关配置

```

```

ceph-s3: &ceph-s3
<<: *common
storage: s3
s3_region: ~
s3_bucket: _env:AWS_BUCKET
s3_encrypt: _env:AWS_ENCRYPT:false
s3_secure: _env:AWS_SECURE:false
storage_path: _env:STORAGE_PATH:/registry
s3_access_key: _env:AWS_KEY
s3_secret_key: _env:AWS_SECRET
boto_bucket: _env:AWS_BUCKET
boto_host: _env:AWS_HOST
boto_port: _env:AWS_PORT
boto_debug: _env:AWS_DEBUG:0
boto_calling_format: _env:AWS_CALLING_FORMAT

# Google 云存储配置
gcs:
<<: *common
storage: gcs
boto_bucket: _env:GCS_BUCKET
storage_path: _env:STORAGE_PATH:/registry
gs_secure: _env:GCS_SECURE:true
gs_access_key: _env:GCS_KEY
gs_secret_key: _env:GCS_SECRET
# 存储服务的 OAuth 2.0 认证
oauth2: _env:GCS_OAUTH2:false

# 存储镜像文件到 Openstack Swift 服务
swift: &swift
<<: *common
storage: swift
storage_path: _env:STORAGE_PATH:/registry
# keystone authorization
swift_authurl: _env:OS_AUTH_URL
swift_container: _env:OS_CONTAINER
swift_user: _env:OS_USERNAME
swift_password: _env:OS_PASSWORD
swift_tenant_name: _env:OS_TENANT_NAME
swift_region_name: _env:OS_REGION_NAME

# 存储镜像文件到 Open Stack Glance 服务
# 参见: https://github.com/docker/openstack-docker
glance: &glance
<<: *common
storage: glance
storage_alternate: _env:GLANCE_STORAGE_ALTERNATE:file
storage_path: _env:STORAGE_PATH:/tmp/registry

openstack:
<<: *glance

```

```

# 存储镜像文件到 Glance，标签信息到 Swift
glance-swift: &glance-swift
  <<: *swift
  storage: glance
  storage_alternate: swift

openstack-swift:
  <<: *glance-swift

elliptics:
  <<: *common
  storage: elliptics
  elliptics_nodes: _env:ELLIPTICS_NODES
  elliptics_wait_timeout: _env:ELLIPTICS_WAIT_TIMEOUT:60
  elliptics_check_timeout: _env:ELLIPTICS_CHECK_TIMEOUT:60
  elliptics_io_thread_num: _env:ELLIPTICS_IO_THREAD_NUM:2
  elliptics_net_thread_num: _env:ELLIPTICS_NET_THREAD_NUM:2
  elliptics_nonblocking_io_thread_num: _env:ELLIPTICS_NONBLOCKING_IO_THREAD_NUM:2
  elliptics_groups: _env:ELLIPTICS_GROUPS
  elliptics_verbosity: _env:ELLIPTICS_VERBOSITY:4
  elliptics_logfile: _env:ELLIPTICS_LOGFILE:/dev/stderr
  elliptics_addr_family: _env:ELLIPTICS_ADDR_FAMILY:2

# 默认启用的配置选项
dev: &dev
  <<: *local
  loglevel: _env:LOGLEVEL:debug
  debug: _env:DEBUG:true
  search_backend: _env:SEARCH_BACKEND:sqlalchemy

# 用于测试
test:
  <<: *dev
  index_endpoint: https://registry-stage.hub.docker.com
  standalone: true
  storage_path: _env:STORAGE_PATH:./tmp/test

# 在环境变量 SETTINGS_FLAVOR 中指定启用哪个配置，例如 $ export SETTINGS_FLAVOR=prod
prod:
  <<: *s3
  storage_path: _env:STORAGE_PATH:/prod

```

## 模板

在 config\_sample.yml 文件中，可以看到一些现成的模板段：

- ❑ common：基础配置。
- ❑ local：存储数据到本地文件系统。
- ❑ s3：存储数据到 AWS S3 中。

- dev：使用 local 模板的基本配置。
- test：单元测试使用。
- prod：生产环境配置（基本上跟 s3 配置类似）。
- gcs：存储数据到 Google 的云存储。
- swift：存储数据到 OpenStack Swift 服务。
- glance：存储数据到 OpenStack Glance 服务，本地文件系统为后备。
- glance-swift：存储数据到 OpenStack Glance 服务，Swift 为后备。
- elliptics：存储数据到 Elliptics key/value 存储。

用户也可以添加自定义的模板段。

默认情况下使用的模板是 dev，要使用某个模板作为默认值，可以添加 SETTINGS\_FLAVOR 到环境变量中，例如：

```
export SETTINGS_FLAVOR=dev
```

另外，配置文件中支持从环境变量中加载值，语法格式为：

```
_env:VARIABLENAME[:DEFAULT]
```

## 选项

基本选项如下：

- loglevel：字符串类型，标注输出调试信息的级别，包括 debug、info、warn、error 和 critical。
- debug：布尔类型，开启后会在访问 /\_ping 时候输出更多的信息，包括库版本和主机信息等。
- storage\_redirect：重定向存储请求。
- boto\_host/boto\_port：使用 s3 模板时，标准 boto 配置文件的位置。

认证选项如下：

- standalone：布尔类型，运行在独立模式下，不进行用户验证等，同时会配置 disable\_token\_auth。
- index\_endpoint：字符串，配置 index 服务位置，用来验证用户，默认是 https://index.docker.io。
- disable\_token\_auth：布尔类型，禁止通过 token 进行验证，此时用户需要提供自己的验证机制。

搜索选项如下：

Docker 注册服务器可以将仓库的索引信息放到数据库中，供通过 GET 方法访问 /v1/search 时使用。

- search\_backend：选择搜索后端类型，目前仅支持 sqlalchemy。用户也可以将它指定

为自定义的模块。例如：

```
common:
  search_backend: foo.registry.index.xapian
```

- sqlalchemy\_index\_database：当使用 sqlalchemy 作为索引后端引擎时，可以通过 sqlalchemy\_index\_database 来指定创建数据库的位置。例如

```
common:
  search_backend: sqlalchemy
  sqlalchemy_index_database: sqlite:///tmp/docker-registry.db
```

镜像选项都放在 mirroring 字段下面，例如：

```
common:
  mirroring:
    source: https://registry-1.docker.io
    source_index: https://index.docker.io
    tags_cache_ttl: 172800 # 2 days
```

默认并未启用。

缓存选项包括 cache 字段和 cache\_lru 字段，例如：

```
cache:
  host: _env:CACHE_REDIS_HOST
  port: _env:CACHE_REDIS_PORT
  db: _env:CACHE_REDIS_DB:0
  password: _env:CACHE_REDIS_PASSWORD

  # Enabling LRU cache for small files
  # This speeds up read/write on small files
  # when using a remote storage backend (like S3).
  cache_lru:
    host: _env:CACHE_LRU_REDIS_HOST
    port: _env:CACHE_LRU_REDIS_PORT
    db: _env:CACHE_LRU_REDIS_DB:0
    password: _env:CACHE_LRU_REDIS_PASSWORD
```

通过配置缓存（事先本地要启动一个 LRU 模式下的 redis 服务器），可以将小文件缓存在本地，加速仓库的查询性能。

Email 选项为 email\_exceptions 字段，通过配置该选项，当仓库发生异常时可自动发送 Email。例如

```
email_exceptions:
  smtp_host: _env:SMTP_HOST
  smtp_port: _env:SMTP_PORT:25
  smtp_login: _env:SMTP_LOGIN
  smtp_password: _env:SMTP_PASSWORD
  smtp_secure: _env:SMTP_SECURE:false
```

```
from_addr: _env:SMTP_FROM_ADDR:docker-registry@localdomain.local
to_addr: _env:SMTP_TO_ADDR:noise+dockerregistry@localdomain.local
```

**存储选项为 storage**, 该选项将选择事先存储的引擎, 仓库默认自带两种类型的引擎: file 和 s3。用户如果需要其他引擎支持, 可以通过下面的命令来进行搜索可用引擎并安装。

```
$ pip search docker-registry-driver
$ pip install docker-registry-driver-NAME
```

安装后, 可能需要对引擎进行配置。目前支持的引擎包括:

- elliptics** (一种分布式键值数据存储)
- swift** (OpenStack 的子项目, 提供对象存储服务)
- gcs** (Goolge 的子存储)
- glance** (OpenStack 的子项目, 提供文件存储服务)

#### file 引擎

file 引擎意味着存储到本地文件。当使用 file 引擎的时候, 可以通过 storage\_path 来指定存储的具体位置, 以 local 模板为例, 默认为 /tmp/registry。

```
local: &local
<<: *common
storage: local
storage_path: _env:STORAGE_PATH:/tmp/registry
```

因此, 我们在运行 registry 镜像时, 可以挂载本地目录到这个位置, 来保存仓库中的数据到本地, 即

```
$ docker run -p 5000 -v /tmp/registry:/tmp/registry registry
```

#### s3 引擎

s3 引擎意味着存储到亚马逊的云服务。亚马逊 s3 引擎支持的选项包括:

- s3\_access\_key**: 字符串类型, s3 的访问口令。
- s3\_secret\_key**: 字符串类型, s3 密钥。
- s3\_bucket**: 字符串类型, s3 桶名称。
- s3\_region**: s3 桶所在的存放域。
- s3\_encrypt**: 布尔类型, 是否加密存储。
- s3\_secure**: 布尔类型, 进行访问时是否启用 HTTPS。
- boto\_bucket**: 字符串类型, 对 s3 不兼容对象存储的桶名。
- boto\_host**: 字符串类型, 对 s3 不兼容对象存储的主机。
- boto\_port**: 对 s3 不兼容对象存储的端口。
- boto\_debug**: 对 s3 不兼容对象存储的调试输出。
- boto\_calling\_format**: 字符串类型, boto 调用所使用格式的类名。

- storage\_path: 字符串类型，镜像文件存储的子路径。

例如：

```
prod:
  storage: s3
  s3_region: us-west-1
  s3_bucket: acme-docker
  storage_path: /registry
  s3_access_key: AKIAHSHB43HS3J92MXZ
  s3_secret_key: xdDowwlK7TJaJv1Y7EoOZrmuPEJlHYcNP2k4j49T
```

## 14.5 本章小结

本章详细介绍了使用 docker-registry 的两种主要方式：通过容器方式运行和通过本地安装运行并注册为系统服务，以及添加 Nginx 反向代理和添加基于 HTTP 的用户认证功能。接下来编写了批量上传镜像到仓库的脚本实现。最后还详细介绍了 docker-registry 配置文件中各个选项的含义和使用。

通过本章的学习，读者将能轻松搭建一套私有的仓库服务环境，并对其进行管理操作。私有仓库服务是集中存储镜像的场所，它的稳定性将影响整个 Docker 使用环节。

在生产环境中，笔者推荐使用负载均衡来提高仓库服务的性能；还可以利用 HAProxy 等方式对仓库服务进行容错。同时，为了安全考虑，可以为仓库访问启用 HTTPS 等加密协议来确保通信的安全。

## 第 15 章 构建 Docker 容器集群

对 Docker 不熟悉的读者在生产环境中使用 Docker 的过程中，往往回碰到构建集群的需求。这里最核心的问题就是让不同主机中的 Docker 容器可以互相访问。

本章将介绍几种解决方案，包括利用端口映射实现容器之间的快速互联，使用 Ambassador 容器解决跨主机的容器互联等。最后，对现有方案的问题进行探讨。

在实际应用中，读者可根据自身情况灵活选择或组合几种方案来满足需求。

### 15.1 使用自定义网桥连接跨主机容器

Docker 默认的网桥是 docker0。它只会在本机连接所有的容器。

举例来说容器的虚拟网卡在主机上看一般叫做 veth\* 而 docker0 网桥把所有这些网卡桥接在一起，如下所示：

```
[root@opnvz ~]# brctl show
bridge name      bridge id               STP enabled     interfaces
docker0          8000.56847afe9799       no              veth0889
                           veth3c7b
                           veth4061
```

在容器中看到的地址一般是像下面这样的地址：

```
root@ac6474aeb31d:~# ip a
1: lo: <LOOPBACK, UP, LOWER_UP> mtu 1500 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
11: eth0: <BROADCAST, UP, LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    link/ether 4a:7d:68:da:09:cf brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::487d:68ff:fed:9cf/64 scope link
        valid_lft forever preferred_lft forever

```

这样就可以把这个网络看成是一个私有的网络，如果要让外网连接到容器中，就需要做端口映射，即 -p 参数。

例如，主机 A 和主机 B 的网卡一都连着物理交换机的同一个 vlan 101，这样网桥一和网桥三就相当于在同一个物理网络中了，而容器一、容器三、容器四也在同一物理网络中了，它们之间可以相互通信，而且可以跟同一 vlan 中的其他物理机器互联，如图 15-1 所示。

下面以 ubuntu 系统为例，创建跨多个主机主机的容器联网。

首先，创建自己的网桥 br0，编辑 /etc/network/interface 文件：

```

auto br0
iface br0 inet static
address 192.168.7.31
netmask 255.255.240.0
gateway 192.168.7.254
bridge_ports em1
bridge_stp off
dns-nameservers 8.8.8.8 192.168.6.1

```

重启后，默认将本地物理网卡 em1 连接到了 br0 上。

在本地修改 /etc/default/docker 文件，添加最后一行内容：

```

# Docker Upstart and SysVinit configuration file
# Customize location of Docker binary (especially for development testing).
#DOCKER="/usr/local/bin/docker"
# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"

# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"

DOCKER_OPTS="-b=br0"

```

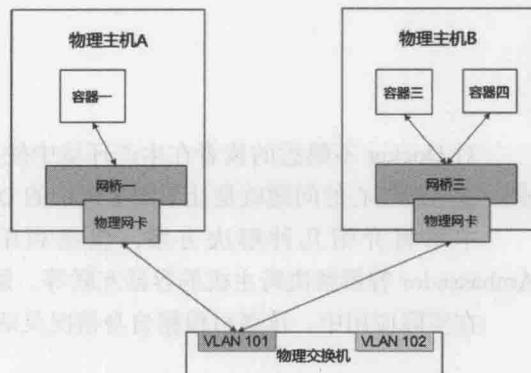


图 15-1 跨主机的 Docker 容器互联

在启动 Docker 的时候使用 -b 参数可以将容器绑定到指定网桥 br0 上。

重启 Docker 服务后，再进入容器可以看到它已经连接到物理网络上了：

```
root@ubuntudocker:~# docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED          STATUS          PORTS          NAMES
58b043aa05eb        desk_hz:v1    "/startup.sh"   5 days ago      Up
2 seconds           5900/tcp, 6080/tcp, 22/tcp   yanlx
root@ubuntudocker:~# brctl show
bridge name            bridge id      STP enabled     interfaces
br0                  8000.7e6e617c8d53    no             em1
                                         veth6e5
```

这样的情况下，容器端口通过映射直接暴露到物理网络上，多台物理主机的容器通过访问外部映射端口即可相互联网了。这样实现的主要问题是需要知道容器所在物理主机的 IP 地址。

## 15.2 使用 Ambassador 容器

当两个 Docker 容器在同一主机（或虚拟机）时，可以通过 --link 命令让两者直接互相访问。如果要跨主机实现容器互联，则往往需要容器知道其他物理主机的 IP 地址。利用 Ambassador 容器机制，可以让互联的容器无需知道所在物理主机的 IP 地址即可互联。

### 基本场景

Ambassador 容器也是一种 Docker 容器，它在内部提供了转发服务。

如图 15-2 所示。当客户端容器要访问服务端容器的时候，直接访问客户端 Ambassador 容器；这个请求会被客户端 Ambassador 转发出去，到达服务端主机。服务端 Ambassador 容器监听在对应端口上，收到请求后再转发请求给服务端容器。

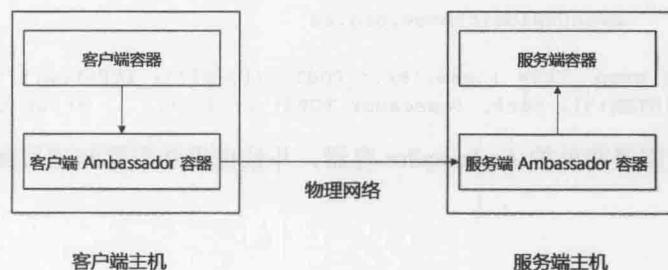


图 15-2 通过 Ambassador 容器实现容器互联

### 使用 Ambassador 容器

以 redis 镜像为例。

首先在服务端主机上创建一个服务端容器 redis-server:

```
$ sudo docker run -d -name redis-server crosbymichael/redis
```

创建一个服务端 Ambassador 容器 redis\_ambassador, 连接到服务端容器 redis-server, 并监听本地的 6379 端口:

```
$ sudo docker run -d -link redis-server:redis -name redis_ambassador -p 6379:6379 svendowideit/ambassador
```

在客户端主机上创建客户端 Ambassador 容器, 告诉它服务端物理主机的监听地址是 `tcp://x.x.x.x:6379`, 将本地收集到 6379 端口的流量转发到服务端物理主机:

```
$ sudo docker run -d -name redis_ambassador -expose 6379 -e REDIS_PORT_6379_TCP=tcp://x.x.x.x:6379 svendowideit/ambassador
```

最后, 创建一个客户端容器, 进行测试, 默认访问 6379 端口实际上是访问的服务端容器内的 redis 应用:

```
$ sudo docker run -i -t -rm -link redis_ambassador:redis relateiq/redis-cli
redis 172.17.0.160:6379> ping
PONG
```

## Ambassador 镜像的 Dockerfile

Ambassador 镜像的 Dockerfile 如下所示。其实现十分简单, 主要是一行正则表达式, 从环境变量中找到包含 "TCP" 字符串的变量, 然后使用正则表达式 `.*_PORT([0-9])_TCP=tcp://(.):(.*)` 从中提取 IP 和端口号, 最后利用 `socat` (一个 socket 转发程序) 将流量转发到指定的地址上:

```
FROM      docker-ut
MAINTAINER SvenDowideit@home.org.au
CMD      env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp://\/\/\(\.\*\):\(\.\*\)/
socat TCP4-LISTEN:\1, fork, reuseaddr TCP4:\2:\3 \&/' | sh && top
```

这种情况下, 需要额外的 Ambassador 容器, 并且也仍然需要知道目标容器所在的物理主机的地址。

## 15.3 本章小结

本章介绍了实现 Docker 容器集群的两种基本方式, 通过端口映射方法实现利用外部物理网络的构建, 以及通过 Ambassador 容器来解决容器跨主机情况下通过内网地址访问的问题。

实际上，要实现容器集群的管理，关键要实现两方面的需求，一是容器名称的动态管理，即容器利用固定的名称可以互相访问，即使有容器发生重启；另外一方面是需要底层网络提供灵活的跨主机的支持租户隔离的连接。

现有的方案，在这两方面的解决并没有做到十分完美。一旦容器发生重启，容器内分配到的IP发生变化，原先的连接就无法使用了。同时，底层直接暴露在物理网络上，依赖于指定物理地址的访问，无法实现灵活的租户隔离等需求。

要解决容器名称的动态管理，有不同的思路，包括添加反向代理的方式、社区内正在讨论的几个补丁以及 SkyDNS+SkyDock、etcd、consul 等工具，本质上就是实现一套容器名到地址的访问，或者说实现容器的 DNS 系统。

而解决底层网络的灵活连接，则需要使用 Overlay 技术。这方面可以基于 VXLAN 等网络虚拟化协议实现跨主机甚至跨物理网络的大二层连通。事实上，管理虚拟机的 OpenStack 等项目提供了相对成熟的基于 SDN 的网络管理方案，已经可以很好地支持容器之间的网络管理。

现在，已经有包括 Shipyard、Kubernetes 等项目实现一整套的容器集群管理方案，笔者将在第三部分中进行介绍。

首先安装 Docker。Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖项，在不同的环境中运行，而不需要考虑底层主机环境的差异。

Docker 提供了一个完整、开源、免费的平台，可以用来构建和部署应用。Docker 容器是完全隔离的，有自己的文件系统，有自己的网络接口，有自己的端口映射，有自己的操作系统层。Docker 容器可以在任何支持 Docker 的平台上运行，从而简化了应用的部署和管理。

Docker 容器的优点在于它们的轻量级、快速启动、易于部署和维护。Docker 容器可以在任何支持 Docker 的平台上运行，从而简化了应用的部署和管理。

## Chapter 16

### 第 16 章

## 在公有云上使用 Docker

本章将介绍如何在公有云上使用 Docker。我们将探讨如何在阿里云、Amazon AWS、Google Cloud Platform 和其他公有云平台上安装 Docker，以及如何在这些平台上运行 Docker 容器。

首先，我们将介绍如何在阿里云上安装 Docker。我们将使用 Aliyun 官方提供的 Docker 镜像，并将其部署到一个名为“my-docker”的新容器中。然后，我们将通过 SSH 连接并运行一个简单的命令来验证 Docker 容器是否正常运行。

在安装 Docker 后，我们将创建一个名为“my-app”的新容器，并将其与“my-docker”容器连接起来。这样，我们就可以在两个容器之间共享数据和资源了。

最后，我们将介绍如何在公有云上部署 Docker 应用。我们将使用 Docker Compose 和 Docker Swarm 来实现这一点。

Docker 目前已经得到了众多的公有云平台的良好支持，包括 Aliyun、Amazon、Rackspace、Softlayer、腾讯云等。

在国内的公有云厂商中，阿里云率先对其 ECS 服务器上安装 Docker 提供了更友好的支持。本章将以国内的阿里云为例，介绍在公有云平台上安装、使用 Docker 的过程和注意事项。其中第一节介绍了在阿里云的 ECS 服务器上安装 Docker 的详细步骤和用法，第二节介绍了在阿里云上使用 Docker 的一些特色服务。

### 16.1 公有云上安装 Docker

以阿里云提供的 CentOS6.5 系统和 Ubuntu 14.04 系统为例，介绍安装和使用 Docker 的过程。

#### 16.1.1 CentOS 6.5 系统

首先，在阿里云网站上申请机器，选择 CentOS 6.5 系统。

通过 ssh 登录阿里云的服务器，查看系统版本号以及内核版本：

```
$ ssh user@your_aliyun_vm
Welcome to aliyun Elastic Compute Service!
# lsb_release -a
LSB Version:    :base-4.0-amd64:base-4.0-noarch:core-4.0-amd64:core-4.0-noarch
Distributor ID: CentOS
Description:    CentOS release 6.5 (Final)
Release:        6.5
```

```
Codename: Final  
# uname -a  
Linux xxxxxxxx 2.6.32-431.23.3.el6.x86_64 #1 SMP Thu Jul 31 17:20:51 UTC 2014  
x86_64 x86_64 x86_64 GNU/Linux
```

可以看到，内核默认认为比较旧的 2.6 系列版本。

## 1. 升级内核

Docker 推荐使用 3.8 以上内核，所以推荐首先升级内核。

导入 KEY，安装软件源。在 YUM 的 ELRepo 源中，有 mainline (3.13.1)、long-term (3.10.28) 这 2 个内核版本，考虑到 long-term 会长期保持支持和更新，所以选择这个版本：

```
# rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org
# yum --enablerepo=elrepo-kernel install kernel-lt -y
Loaded plugins: security
base                               | 3.7 kB     00:00
base/primary_db                     | 4.6 MB     00:00
elrepo                             | 2.9 kB     00:00
elrepo/primary_db                   | 709 kB    00:46
elrepo-kernel                       | 2.9 kB     00:00
elrepo-kernel/primary_db            | 20 kB      00:01
epel                               | 4.4 kB     00:00
epel/primary_db                     | 6.3 MB     00:01
extras                             | 3.4 kB     00:00
extras/primary_db                   | 29 kB      00:00
updates                            | 3.4 kB     00:00
updates/primary_db                  | 181 kB     00:00
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package kernel-lt.x86_64 0:3.10.59-1.el6.elrepo will be installed
--> Finished Dependency Resolution
Dependencies Resolved

=====
=====
=====
Package      Arch      Version       Repository      Size
=====
=====
=====
Installing:
kernel-lt    x86_64    3.10.59-1.el6.elrepo  elrepo-kernel  33 M

Transaction Summary
=====
=====
=====
Install      1 Package(s)
```

```
Total download size: 33 M
Installed size: 153 M
Downloading Packages:
kernel-lt-3.10.59-1.el6.elrepo.x86_64.rpm | 33 MB 28:58
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
Warning: RPMDB altered outside of yum.
Installing : kernel-lt-3.10.59-1.el6.elrepo.x86_64 1/1
Verifying : kernel-lt-3.10.59-1.el6.elrepo.x86_64 1/1

Installed:
kernel-lt.x86_64 0:3.10.59-1.el6.elrepo

Complete!
```

安装后，检查 /etc/grub.conf 文件，查看默认的启动内核。新安装的内核一般在第一个，这里把 default=1 改为 default=0 就好了：

```
default=1
timeout=5
splashimage=(hd0, 0)/boot/grub/splash.xpm.gz
hiddenmenu
title CentOS (3.10.59-1.el6.elrepo.x86_64)
    root (hd0, 0)
    kernel /boot/vmlinuz-3.10.59-1.el6.elrepo.x86_64 ro root=UUID=94e4e384-0ace-437f-bc96-057dd64f42ee rd_NO_LUKS rd_NO_LVM LANG=en_US.UTF-8 rd_NO_MD
SYSFONT=latarcyrheb-sun16 crashkernel=auto KEYBOARDTYPE=pc KEYTABLE=us rd_NO_DM rhgb quiet
    initrd /boot/initramfs-3.10.59-1.el6.elrepo.x86_64.img
title CentOS (2.6.32-431.23.3.el6.x86_64)
    root (hd0, 0)
    kernel /boot/vmlinuz-2.6.32-431.23.3.el6.x86_64 ro root=UUID=94e4e384-0ace-437f-bc96-057dd64f42ee rd_NO_LUKS rd_NO_LVM LANG=en_US.UTF-8 rd_NO_MD
SYSFONT=latarcyrheb-sun16 crashkernel=auto KEYBOARDTYPE=pc KEYTABLE=us rd_NO_DM rhgb quiet
    initrd /boot/initramfs-2.6.32-431.23.3.el6.x86_64.img
~
```

重启云主机，查看内核是否升级成功：

```
# uname -a
Linux xxxxxxxx 3.10.59-1.el6.elrepo.x86_64 #1 SMP Thu Oct 30 23:46:31 EDT 2014
x86_64 x86_64 x86_64 GNU/Linux
```

## 2. 安装 Docker

添加软件源，并安装 Docker 软件：

```
# yum install http://mirrors.yun-idc.com/epel/6/i386/epel-release-6-8.noarch.rpm
# yum install docker-io
```

### 3. 启动 Docker

使用 service 命令启动 Docker，发现会提示有问题：

```
# service docker start
Starting cgconfig service: [ OK ]
Starting docker: [ OK ]
# docker version
Client version: 1.2.0
Client API version: 1.14
Go version (client): go1.3.3
Git commit (client): fa7b24f/1.2.0
OS/Arch (client): linux/amd64
2014/11/05 21:03:08 Cannot connect to the Docker daemon. Is 'docker -d' running
on this host?
```

使用 docker-d 启动方式来查看详细的启动过程：

```
# docker -d
2014/11/05 21:10:56 docker daemon: 1.2.0 fa7b24f/1.2.0; execdriver: native;
graphdriver:
[40a2dcc2] +job serveapi(unix:///var/run/docker.sock)
[info] Listening for HTTP on unix (/var/run/docker.sock)
[40a2dcc2] +job init_networkdriver()
[40a2dcc2.init_networkdriver()] creating new bridge for docker0
Could not find a free IP address range for interface 'docker0'. Please configure
its address manually and run 'docker -b docker0'
[40a2dcc2] -job init_networkdriver() = ERR (1)
2014/11/05 21:10:56 Could not find a free IP address range for interface
'docker0'. Please configure its address manually and run 'docker -b docker0'
```

提示没有空余 ip 分配给 docker0 了。

可以使用 --bip 参数来手工分配给 ip 地址。比如：

```
# docker --bip=192.168.100.1/24 -d &
[2] 2388
[root@iZ23pznIje4Z ~]# 2014/11/05 21:16:55 docker daemon: 1.2.0 fa7b24f/1.2.0;
execdriver: native; graphdriver:
[dc6906e7] +job serveapi(unix:///var/run/docker.sock)
[info] Listening for HTTP on unix (/var/run/docker.sock)
[dc6906e7] +job init_networkdriver()
[dc6906e7] -job init_networkdriver() = OK (0)
2014/11/05 21:16:55 WARNING: Your kernel does not support cgroup swap limit.
[info] Loading containers:
[info] : done.
[dc6906e7] +job acceptconnections()
[dc6906e7] -job acceptconnections() = OK (0)
```

仍然有警告 Your kernel does not support cgroup swap limit。

使用 **lxc-checkconfig** 进行检查：

```
# lxc-checkconfig
Kernel configuration not found at /proc/config.gz; searching...
Kernel configuration found at /boot/config-3.10.59-1.el6.elrepo.x86_64
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled

Note : Before booting a new kernel, you can check its configuration
usage : CONFIG=/path/to/config /usr/bin/lxc-checkconfig
```

可以看到 Control groups 项目在内核是支持的，所以可以暂时忽略这个告警。

#### 4. 测试使用

首先，下载 ubuntu 镜像。这里使用 Dockerpool 官方网站的标准 ubuntu 镜像来进行测试。

下载镜像，重新标记镜像：

```
# docker pull dl.dockerpool.com:5000/ubuntu:14.04
# docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL      SIZE
dl.dockerpool.com:5000/ubuntu      latest      5506de2b643b      12 days ago
197.8 MB
# docker tag 550 ubuntu
# docker rmi dl.dockerpool.com:5000/ubuntu
[dc6906e7] +job image_delete(dl.dockerpool.com:5000/ubuntu)
```

```
[dc6906e7] +job log(untag, 5506de2b643be1e6febfb3b8a240760c6843244c41e12aa2f60c
ccb7153d17f5, )
[dc6906e7] -job log(untag, 5506de2b643be1e6febfb3b8a240760c6843244c41e12aa2f60c
ccb7153d17f5, ) = OK (0)
[dc6906e7] -job image_delete(dl.dockerpool.com:5000/ubuntu) = OK (0)
Untagged: dl.dockerpool.com:5000/ubuntu:latest
```

利用刚下载的镜像启动一个容器，并测试网络：

```
# docker run -ti ubuntu
root@66ff9a55a4f5:/# ping www.dockerpool.com
PING www.dockerpool.com (xxx.xxx.xxx.xxx) 56(84) bytes of data.
^C64 bytes from 203.195.193.251: icmp_seq=1 ttl=47 time=31.4 ms
```

### 16.1.2 Ubuntu 14.04 系统

Ubuntu 14.04 的内核是比较新的 3.13 版本，可以较好地支持 Docker，所以不需要进行升级。

利用 apt-get 安装 Docker 步骤如下：

```
# apt-get update
# apt-get install apt-transport-https
# apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 36A1D786924
5C8950F966E92D8576A8BA88D21E9
# bash -c "echo deb https://get.docker.io/ubuntu docker main > /etc/apt/
sources.list.d/docker.list"
# apt-get update
# apt-get install lxc-docker
```

启动的时候也会遇到无法分配空闲 IP 的提示，可以使用如下命令：

```
# docker --bip 192.168.100.1/24 -d &
```

启动即可。

后面的下载和测试步骤与 CentOS 类似，在此不再赘述。

## 16.2 阿里云 Docker 的特色服务

图 16-2（摘自阿里云官方网站）描述了使用阿里云 ECS Docker 将使开发、测试和运维之间合作更加紧密，各司其职，实现更高的效率。

图 16-2 是阿里云 ECS Docker 的完整生态图（摘自阿里云官方网站）。

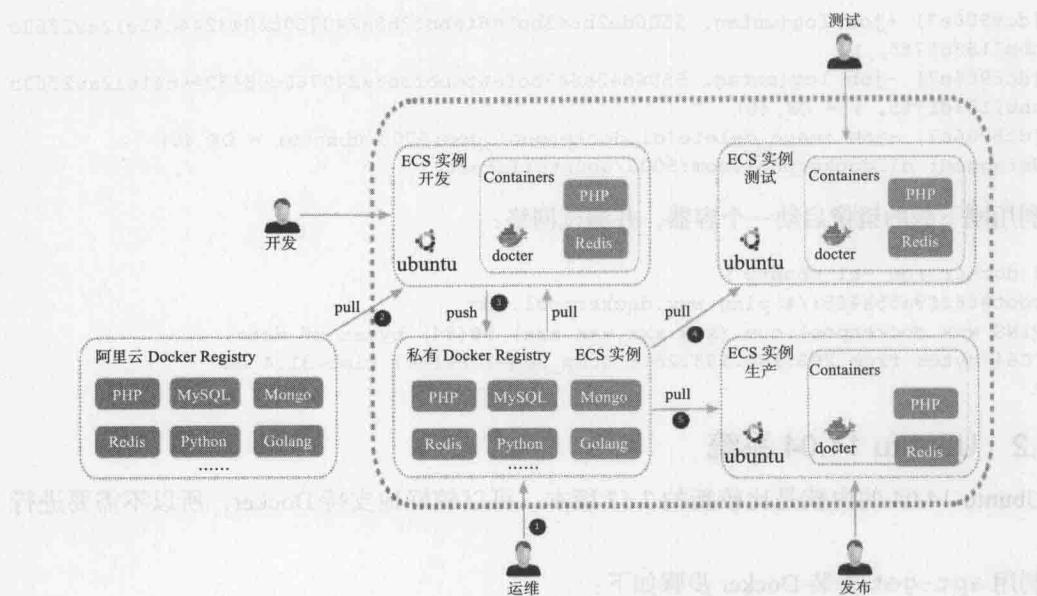


图 16-1 阿里云 Docker 的配置

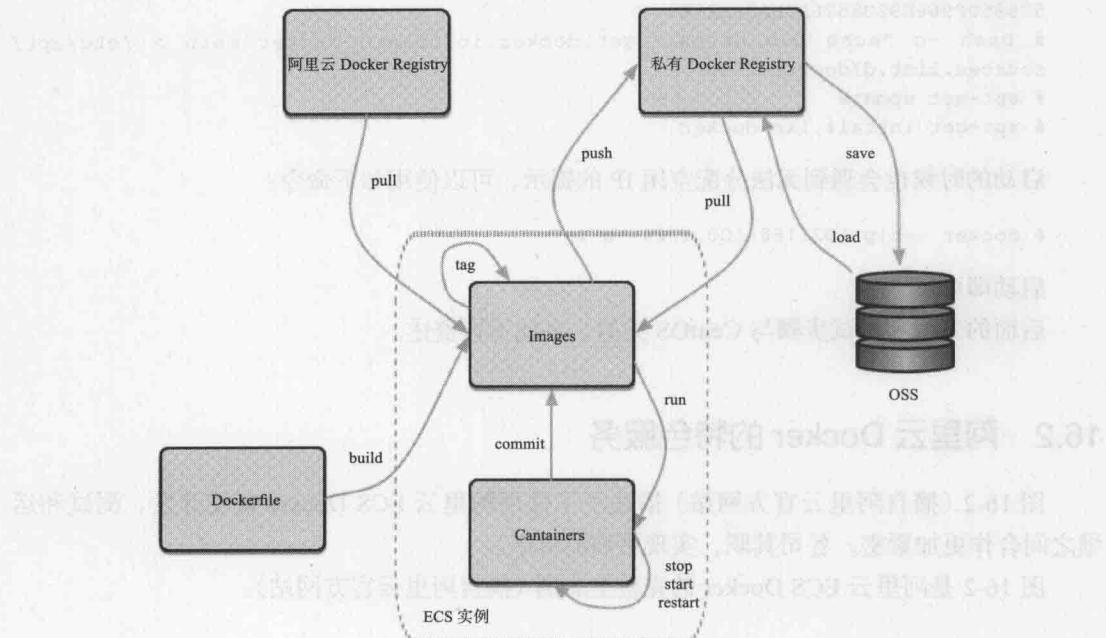


图 16-2 阿里云 ECS Docker 的生态图

## 阿里云镜像市场中的第三方 Docker 镜像

在阿里云的镜像市场有一款镜像“Docker 运行环境”，如图 16-3 所示，它的操作系统使用 Ubuntu 14.04 64 位并预装了 Docker 1.2 版本，一旦 ECS 实例运行，读者就能在其上构建和运行 Docker 容器了。镜像地址为：<http://market.aliyun.com/imageproduct/16-122106003-jxsc000057.html>

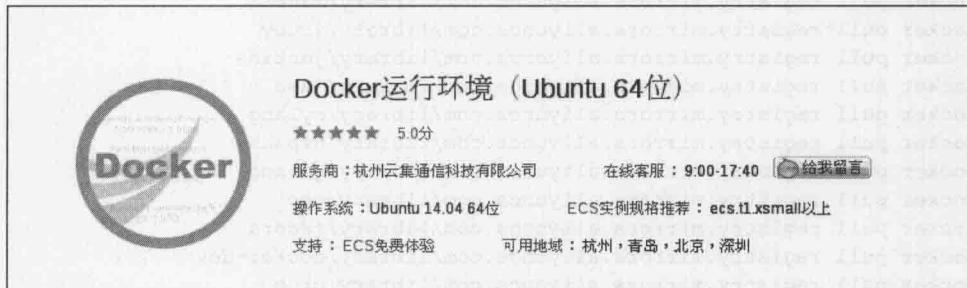


图 16-3 阿里云市场上的 Docker 运行环境

## 专为阿里云 ECS 用户提供下载的 Docker 私有仓库

为方便 ECS 用户使用 Docker 官方镜像，阿里云同步 Docker 官方镜像库的最新版本到国内服务器，使得 ECS 用户可以通过内网连接该服务器。这些镜像来自 Docker Hub 的 stackbrew 用户下的所有镜像仓库，一部分镜像由 Docker 官方维护，一部分由软件官方社区维护。目前只支持镜像下载。

笔者下面展示一下如何使用阿里云的源来下载镜像。

阿里云的私有仓库不支持使用标准方式来查询，下面的结果返回为空：

```
# curl registry.mirrors.aliyuncs.com/v1/search
{"num_results": 0, "query": "", "results": []}
```

不过，官方公布了两种下载镜像的方法。

### 第一种方法

可以通过下面的命令下载各种镜像：

```
docker pull registry.mirrors.aliyuncs.com/library/debian
docker pull registry.mirrors.aliyuncs.com/library/hello-world
docker pull registry.mirrors.aliyuncs.com/library/zend-php
docker pull registry.mirrors.aliyuncs.com/library/wordpress
docker pull registry.mirrors.aliyuncs.com/library/ubuntu-upstart
docker pull registry.mirrors.aliyuncs.com/library/ubuntu-debootstrap
docker pull registry.mirrors.aliyuncs.com/library/ubuntu
docker pull registry.mirrors.aliyuncs.com/library/ruby
```

```

docker pull registry.mirrors.aliyuncs.com/library/registry
docker pull registry.mirrors.aliyuncs.com/library/redis
docker pull registry.mirrors.aliyuncs.com/library/rails
docker pull registry.mirrors.aliyuncs.com/library/python
docker pull registry.mirrors.aliyuncs.com/library/postgres
docker pull registry.mirrors.aliyuncs.com/library/php
docker pull registry.mirrors.aliyuncs.com/library/perl
docker pull registry.mirrors.aliyuncs.com/library/opensuse
docker pull registry.mirrors.aliyuncs.com/library/node
docker pull registry.mirrors.aliyuncs.com/library/mageia
docker pull registry.mirrors.aliyuncs.com/library/jruby
docker pull registry.mirrors.aliyuncs.com/library/jenkins
docker pull registry.mirrors.aliyuncs.com/library/java
docker pull registry.mirrors.aliyuncs.com/library/hylang
docker pull registry.mirrors.aliyuncs.com/library/hipache
docker pull registry.mirrors.aliyuncs.com/library/golang
docker pull registry.mirrors.aliyuncs.com/library/gcc
docker pull registry.mirrors.aliyuncs.com/library/fedora
docker pull registry.mirrors.aliyuncs.com/library/docker-dev
docker pull registry.mirrors.aliyuncs.com/library/crux
docker pull registry.mirrors.aliyuncs.com/library/crate
docker pull registry.mirrors.aliyuncs.com/library/clojure
docker pull registry.mirrors.aliyuncs.com/library/cirros
docker pull registry.mirrors.aliyuncs.com/library/centos
docker pull registry.mirrors.aliyuncs.com/library/busybox
docker pull registry.mirrors.aliyuncs.com/library/buildpack-deps
docker pull registry.mirrors.aliyuncs.com/library/nginx
docker pull registry.mirrors.aliyuncs.com/library/mongo
docker pull registry.mirrors.aliyuncs.com/library/neurodebian
docker pull registry.mirrors.aliyuncs.com/library/mysql

```

我们来测试下这个内网的阿里云源下载速度如何，让我们来下载一个 ubuntu:14.04 的镜像：

```

# time docker pull registry.mirrors.aliyuncs.com/library/ubuntu:14.04
2014/11/19 21:22:22 Error: Invalid registry endpoint https://registry.mirrors.aliyuncs.com/v1/: Get https://registry.mirrors.aliyuncs.com/v1/_ping: dial tcp 10.157.230.35:443: i/o timeout. If this private registry supports only HTTP or HTTPS with an unknown CA certificate, please add '--insecure-registry registry.mirrors.aliyuncs.com' 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/registry.mirrors.aliyuncs.com/ca.crt

real    0m5.020s
user    0m0.009s
sys     0m0.006s

```

如果读者跟我一样已经将 Docker 的版本升级到 1.3，那么就会出现这个提示，当我们使用 1.3 版本的 Docker 来下载非官方镜像时，都会让我们手工确认该源的安全性，如果确认没问题，则需要手工添加 `--insecure-registry` 到启动参数中。具体的添加方法在本书最

后的FAQ中有详细介绍。

添加完之后，我们再次来下载ubuntu:14.04镜像：

```
$ sudo time docker pull registry.mirrors.aliyuncs.com/library/ubuntu
Pulling repository registry.mirrors.aliyuncs.com/library/ubuntu
5506de2b643b: Download complete
511136ea3c5a: Download complete
d497ad3926c8: Download complete
ccb62158e970: Download complete
e791be0477f2: Download complete
3680052c0f5c: Download complete
22093c35d77b: Download complete
Status: Image is up to date for registry.mirrors.aliyuncs.com/library/
ubuntu:latest

real    0m12.681s
user    0m0.010s
sys     0m0.007s
```

12秒就下载了ubuntu:latest镜像，还可以手工指定需要下载的版本：

```
$ sudo docker pull registry.mirrors.aliyuncs.com/library/ubuntu:12.04
Pulling repository registry.mirrors.aliyuncs.com/library/ubuntu
0b310e6bf058: Download complete
511136ea3c5a: Download complete
5f18d94c3eca: Download complete
53db23c604fd: Download complete
9f045ea36057: Download complete
d03a1a9d7555: Download complete
30868777f275: Download complete
Status: Downloaded newer image for registry.mirrors.aliyuncs.com/library/
ubuntu:12.04
```

## 第二种方法

创建from\_aly空目录，并在其中创建Dockerfile文件：

```
$ mkdir from_aly
$ cd from_aly/
$ vi Dockerfile
```

Dockerfile内容为：

```
FROM registry.mirrors.aliyuncs.com/library/ubuntu:14.04
```

根据这一句话的Dockerfile创建镜像：

```
$ sudo docker build -t ubuntu_aly:14.04 .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM registry.mirrors.aliyuncs.com/library/ubuntu:14.04
```

```
Pulling repository registry.mirrors.aliyuncs.com/library/ubuntu
5506de2b643b: Download complete
511136ea3c5a: Download complete
d497ad3926c8: Download complete
ccb62158e970: Download complete
e791be0477f2: Download complete
3680052c0f5c: Download complete
22093c35d77b: Download complete
Status: Image is up to date for registry.mirrors.aliyuncs.com/library/ubuntu:14.04
--> 5506de2b643b
Successfully built 5506de2b643b
```

## 支持阿里云 OSS 的私有仓库

官方在 Github 上有一个项目 docker-registry，专门用于自建 Docker 的私有镜像库。镜像管理是 Docker 的核心，为了保证镜像数据的可靠、可用和安全，docker-registry 现在支持镜像数据存储在 S3、GCS 等云存储上。

已经有人给 docker-registry 开发了针对阿里云 OSS 的驱动，并把它和 docker-registry 一起做成了 Docker 镜像。以下是快速启动支持 OSS 的 docker-registry 的方式：

```
$ sudo docker run -e OSS_BUCKET=<your_ali_oss_bucket> -e STORAGE_PATH=/docker/
-e OSS_KEY=<your_ali_oss_key> -e OSS_SECRET=<your_ali_oss_secret> -p 5000:5000
-d chrisjin/registry:ali_oss
```

读者也可以从 <https://github.com/docker/docker-registry> 下载安装 docker-registry，并通过 pip 安装 OSS driver。

```
$ sudo pip install docker-registry-driver-alioss
```

接下来配置 config.yml。

```
local: &local
  <<: *common
  storage: alioss
  storage_path: _env:STORAGE_PATH:/devregistry/
  oss_bucket: _env:OSS_BUCKET[:default_value]
  oss_accessid: _env:OSS_KEY[:your_access_id]
  oss_accesskey: _env:OSS_SECRET[:your_access_key]
```

最后启动 docker-registry。

```
DOCKER_REGISTRY_CONFIG= [ your_config_path ] gunicorn -k gevent -b 0.0.0.0:5000 -w
1 docker_registry.wi:application
```

### 16.3 本章小结

公有云已经提供了诸多虚拟化带来的便利，那么在上面使用 Docker 还有意义吗？

其实，通过整合公有云的虚拟机和 Docker 方式，可能获得更多的好处，包括：

- 快速交付和部署。

- 利用内核级虚拟化，对公有云中服务器资源进行更加高效地利用。

- 利用公有云和 Docker 的特性更加方便的迁移和扩展应用。

以一个简单的应用开发、测试和发布过程来说明 Docker 在公有云上的应用过程。

首先，运维人员在公有云上搭建私有 Docker 注册服务器，以存储项目组镜像。开发人员在开发过程中从搭建的私有 Docker Registry 获取应用需要的基础镜像。

之后，可以开发并构造应用容器，测试通过后提交容器为新的镜像并推送到私有的 Docker Registry。QA 在测试云服务器上测试容器，通过后提交到私有的 Docker Registry。

最后，发布人员下载最新版本镜像并在生产云服务器上部署容器。

*Chapter 17*

## 第 17 章

# Docker 实践之道

笔者在刚接触 Docker 的时候，曾回想到起自己第一次安装和使用 Linux 的情景。

坐了两个小时的公交车专门到电脑市场，花 20 块钱买了 2 张 Redhat 9 的安装盘。回来后兴冲冲删除了电脑上原本的 Windows 2000 系统，安装完之后，却发现无法访问网络。从笔者决定要安装 Redhat 到最后能正常使用，并完成一些基本的日常操作，花了将近半个月的时间。

而现在，只需要一个命令就可以下载一个操作系统，比如 `docker pull centos7` 就可以完成当年半个月才能完成的工作。当然，我们也可以通过 VMware、Virtrbox 等工具来实现同样的需求。但是 Docker 正以一种前所未有的方式让我们可以快速在各种 Linux 发行版中切换。这对 Linux 爱好者来说真是一种福音。此外，读者还可以使用 Docker 在个人电脑 (PC 或笔记本) 上来搭建各种开发环境，且不必担心破坏个人电脑中的系统环境 (如环境变量) 和应用程序。系统架构师们也可以使用 Docker 来快速搭建各种网络架构的系统，且可以方便的管理这些系统之间的数据连接和共享。

目前 Docker 发展迅速，基于 Docker 的 PaaS 平台也层出不穷。这让技术创业者无需折腾服务器部署，只需专注业务代码的实现即可。

本章主要介绍了 Docker 在个人的学习和技术创业方面的典型应用场景，以及如何使用 Docker 为企业构建标准化的开发、测试、生产环境。

## 17.1 个人学习之道

本章节，笔者希望和大家一起探讨 Docker 在个人技术学习和修炼的过程中起到的积极作用，并与大家探讨 Docker 作为一款‘全栈神器’和‘修炼利器’的内涵与外延。

笔者是在技术圈沉浸多年的开源技术爱好者，平时对自己的技能提高有着习惯性地追求，对新技术有着持续的好奇心。如果读者经常逛各大 IT 论坛和社区的话，那么读者会发现技术圈内的这类 Geek 不算少数。而恰恰是这帮热爱技术并乐于实践的 Geek，给整个中国 IT 界的技术更新和创新注入了新活力。比如润物细无声的 Python 社区，火热的 Ruby 社区和快速发展的 Golang 社区。这帮走在技术前沿的 Geek，也给个人的职业生涯创造了更多的机遇，往往也可以带来更好的待遇。可见，修炼与不修炼，这是一个伪命题。因为如果彻底放弃技术学习，那么被淘汰只是时间问题。所以，如何使用 Docker 将技术修炼的效能最大化，是笔者想和大家分享的一个有趣的话题。

笔者将从以下两点出发讨论工程师的修炼之道：

- 温故而知新：快速入门并建立自己的 Docker 化的代码库。
- 众人拾柴火焰高：使用 Docker Hub 发布开源项目。

### 17.1.1 温故而知新

孔子曰：“温故而知新，可以为师矣”。现代软件 / 运维工程师，与古代的铁匠木匠一样，都是利用工具进行创造性工作的匠人。工作若干年后，笔者发现身边的优秀工程师，无一不似匠人般追求精进技能和拓展视野。作为即需要“温故”又需要“知新”的“匠人”工程师，如果将 Docker 的作用最大化呢？下面，笔者将从三个方面去阐述：

- 快速上手 Hello World：去除学习过程中的“噪音”，快速上手开发环境。
- Docker 化的代码仓库：从收集代码，到收集 Docker 容器。
- 面向业务编程：快速使用新技术支撑新业务。

#### 1. 快速上手 Hello World

众所周知，IT 新技术的学习往往从 Hello World 开始。这是学习新知识的标准思路：最小系统原则，即从变量最少的最小系统开始，循序渐进地学习。

现实生活中，简单的事物背后往往蕴含着复杂的机制。我们在构建最小系统的时候，首先面对的就是其母环境（或者说前置条件）的搭建。虽然随着程序语言和系统程序的发展，语言和工具都越来越好学好用。但学习成本仍然居高不下。各大编程语言论坛中关于环境安装的问题总是层出不穷。

通过 Docker 的使用，我们可以将精力和注意力都尽快地放在语言本身的学习上，而无需折腾系统环境的各种配置。Docker 官网的口号就包含了以上含义：Build, Ship and Run Any App, Anywhere，即任何应用，都可以构建，发布，运行于任何环境，将环境的影响因素降至最低，统一地掌控整个应用的生命周期。

目前 Docker 官方支持的编程语言镜像就有十几种，涵盖所有的主流编程语言的开发环境。除此之外，常用数据库，缓存系统，主流 Web 框架等都有官方的镜像。除此之外，Docker Hub 还提供丰富的第三方镜像和 Dockerfile。

## 2. Docker 化的代码仓库

常用代码库往往是软件工程师实现高效交付的“绝活”。在技术团队中，为何行业新人和资深工程师之间的生产力可以有几倍甚至几十倍的差距呢？暂且不论基础技能的差距，同样是做一件任务，新人接手后首先面对的就是思路和工具的抉择，然后需要面对实践中的各种‘坑’。而资深工程师接手后，如果拥有 Docker 化的代码仓库，那么就可以快速完成任务。停止研发的历史任务，也可以以 Docker 容器的方式保存。以后遇到类似的需求，可以直接运行，调试并复用代码。

## 3. 面向业务编程

国人编著了一本非常干货的程序员基本素养的书籍叫《程序员的自我修养 - 链接，构建与库》，书中提到一点：MOP Market/Money Oriented Programming（面向市场或利润编程）是唯一不变的编程范式<sup>①</sup>。现在的 IT 业界，每年都创造出各种语言和工具来解决各种问题，如高并发或用户体验问题等。俗话说：站得高，望得远。IT 工程师如何高效的提高素养并跟上技术发展的脚步，是业界关心的问题。

笔者根据 Docker 的特性，给出一个可行方案：使用 Docker 快速掌握新技术要点并完成适当的技术储备。下面，笔者举一些简单的例子说明：若读者是 Python 技术栈的后端工程师，熟悉常规网站的后台建设，那么如何快速实现移动应用的 Restful API Sever 呢？笔者在此建议读者去 Docker Hub 搜索适合做 API 服务器的 Python 快速开发框架。如 docker pull python，以此为基础，加入 SSH 服务，数据库和开发框架，即可构建一个 Python 容器。然后，通过 docker run 运行容器。此时即可使用浏览器访问自定义容器中的 API 接口。可见，Docker 可以帮助软件工程师面向业务需求，快速了解新技术，或进行各种技术研究。

### 17.1.2 众人拾柴火焰高

随着各类开源组织和社区的风靡，软件 / 运维工程师都可以使用拿手的语言和工具，参与到各种开源项目中。这不仅提高了自己，也造福了技术圈中的使用者。笔者在此建议：读者如果参与开源项目的建设，那么可以通过 Docker 完成程序的打包，测试，发布和部署。这样可以统一又清晰的管理整个开源项目。

笔者在此举例说明：读者加入开源贡献者的队伍后，首先需要搭建统一的开发环境。如果完成编码工作后不进行全面的测试，则无法提供完整的 Test Case，一般而言是无法快速提交至主版本库的。那么，使用 Docker 发布项目后，任何参与者可以一键运行各种运行环境（容器），轻松的运行单元测试，最终可以快速提交代码至主干（Trunk）。

## 17.2 技术创业之道

目前的互联网创业团队往往需要在人力紧缺的情况下快速发布 Demo 或初版产品，并快速迭代，以发布切合市场需求的产品。

本章笔者将从以下两点出发，阐述 Docker 如何帮助创业团队在以上条件下，做得更快更好：

- 使用 Docker 助力 DevOps：搭建 Docker 化的 DevOps 流程。
- 产品的 Docker 化发布：助力构建各种 SaaS、PaaS 平台。

### 使用 Docker 助力 DevOps

敏捷开发是一套软件工程方法论。其中有一个重要的概念是 DevOps。

DevOps（英文 Development 和 Operations 的组合）是一组过程、方法与系统的统称，用于促进开发（应用程序 / 软件工程）、技术运营和质量保障（QA）部门之间的沟通、协作与整合。它的出现是由于软件行业日益清晰地认识到：为了按时交付软件产品和服务，开发和运营工作必须紧密合作。

DevOps 从一开始的敏捷开发的常用实践方法论，逐渐落地至大量的生产环境。时至今日已经经常出现在很多技术团队的日常工作中。

从上文引用的维基百科释义可以看出，DevOps 的核心理念在于生产团队（研发，运维，QA）之间的高效沟通协作，以解决以下常见问题：

- 更小、更频繁的需求变更。
- 生产环境不受开发人员控制。
- 业务应用程序成为中心，而不是基础设施。
- 定义简洁明了的研发部署流程需要更多成本与时间。
- 研发部署流程无法彻底自动化。
- 现有 PaaS 虚拟机难以促成开发与运营的协作。

在这种情况下，使用 Docker 可以满足以下几点：

- 完整地封装系统：包括 OS 系统，Lib 环境，App 应用，完整的三层封装。
- 自由地定制系统：包括以上三层的灵活又彻底的自定义。
- 方便地发布系统：包括部署管理，自动化部署。

### 产品的 Docker 化发布

由于 Docker 具有易于部署，跨平台能力强的特性，所以 Docker 非常适合做自动化的持续集成，和快速部署。这种情况下，直接使用 Docker 建设产品的发布流程，不仅可以方便地管理迭代与集成，还可以直接以 Docker 化的方式（Dockerize）发布产品。

Docker 化发布特别适用于 BS 架构的产品。由于 BS 产品的服务器端的部署往往受到各种不定因素（系统版本，依赖关系）的影响，所以其部署流程的用户体验往往非常繁琐而且

容易产生错误，需要使用者去解决各种部署环境中的问题。

例如，WordPress 这种主流的 CMS 系统，尽管 WordPress 的安装流程已经极尽简化，并以图形化界面的形式进行安装。但是笔者在使用过程中，仍然会遇到一些部署上的问题需要处理。如果使用 Docker 发布 WordPress（即使用官方镜像），确实可以做到一句命令即可安装到位，而且安装过程是绝对统一的，环境也是绝对纯净的。

笔者相信 Docker 这种无痛部署的方式，会随着 Docker 逐渐成为云计算平台（或者说 PaaS）的标配，而进入各种 BS 架构产品的发布流程中。

## 17.3 中小型企业实践之道

### 17.3.1 开发、测试和发布中应用 Docker

在传统模式中，开发团队在开发环境中完成软件开发，自己做了一遍单元测试，测试通过，提交到代码版本管理库，打包给开发团队进行测试。运维把应用部署到测试环境，开发团队进行测试，没问题后通知部署人员发布到生产环境。

在上述过程中涉及到至少三个环境：开发、测试和生产。现实情况是，开发自测没问题，但到了测试或者生产环境程序无法运行，让开发团队排查，经过长时间排查最后发现是测试环境的一个第三方库过时了。这样的现象在软件开发中很普遍，已经不适用如今的快速开发和部署。

在 Docker 模式中，应用是以容器的形式存在，所有和该应用相关的依赖都会在容器中，因此移植非常方便，不会存在像传统模式那样的环境不一致。图 17-1 比较了两种模式下的不同流程。

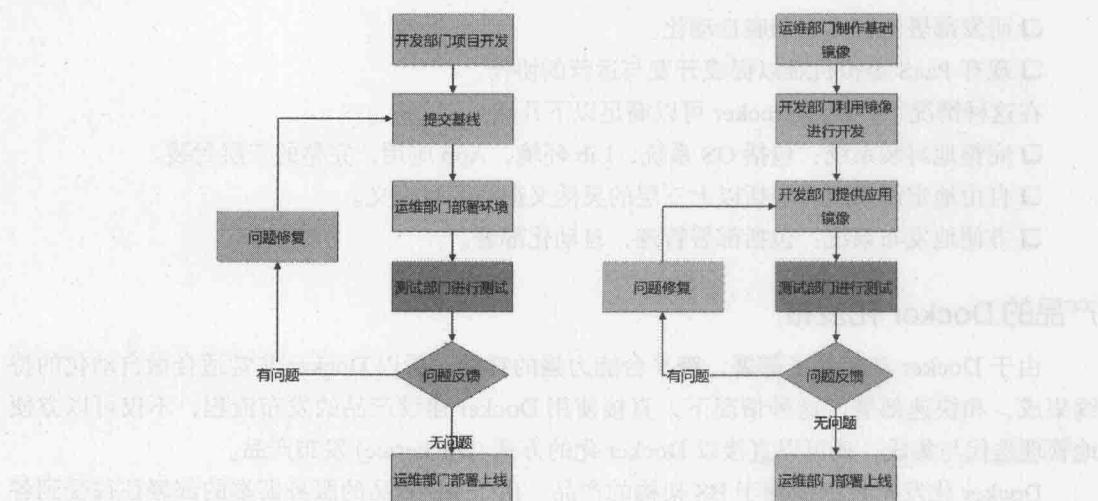


图 17-1 两种开发模式下的不同流程的比较

## 1. 场景示例

我们假定一个场景，一个 200 人左右的软件企业，主要使用 Java 作为开发语言，使用 Tomcat、Weblogic 作为中间件服务器，后台数据库使用 Oracle、MySQL。在应用 Docker 之前，开发到测试的流程如图 17-2 所示。

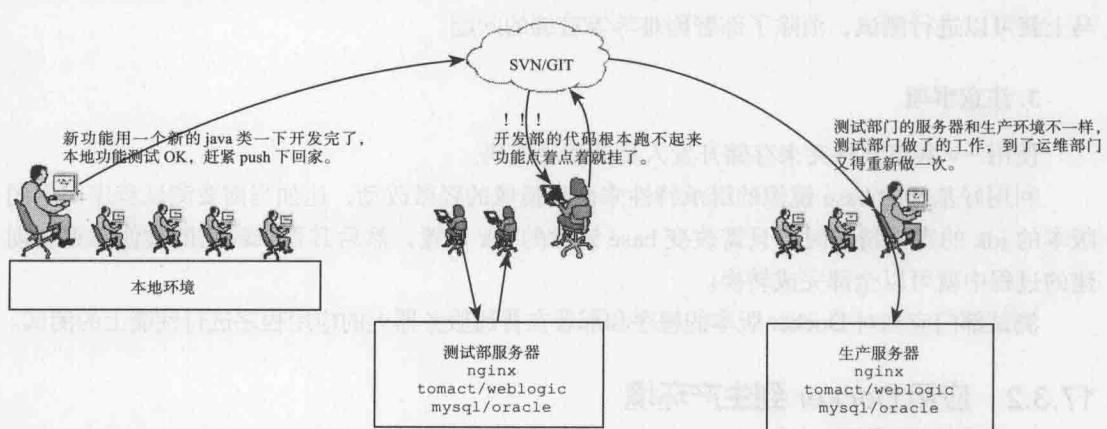


图 17-2 传统的开发流程

可见因为环境的不一样，开发、测试、运维三个部门做了很多重复的工作。

而 Docker 正好可以解决这个问题，如图 17-3 所示。



图 17-3 利用 Docker 环境开发的流程

## 2. 操作流程

在 Docker 应用中，项目架构师的作用贯穿整个开发、测试、生产三个环节。

项目伊始，架构师根据项目预期创建好需要的基础 Base 镜像、Nginx、Tomcat、MySQL 镜像或者将 Dockerfile 分发给所有开发人员，所有开发人员根据 Dockerfile 创建的容器或者从内部仓库下载的镜像来进行开发，达到开发环境的充分一致。若开发过程中需要添加新的软件，只需要向架构师申请修改基础的 base 镜像的 Dockerfile 即可。

开发任务结束后，架构师调整 Dockerfile 或者 image，然后分发给测试部门，测试部门马上就可以进行测试，消除了部署困难等等难缠的问题。

### 3. 注意事项

使用 -v 共享文件夹来存储开发人员的程序代码。

利用好基础的 base 镜像的继承特性来调整镜像的轻微改动，比如当需要测试程序对不同版本的 jdk 的支持情况时，只需改变 base 镜像的 jdk 设置，然后其他依赖它的镜像在重新创建的过程中就可以全部完成转换。

测试部门应当对 Docker 版本的程序和部署在普通服务器上的应用程序进行性能上的测试。

#### 17.3.2 应用 Docker 到生产环境

目前在生产环境上使用 Docker 的企业已经不在少数，比较有名的就是百度 BAE。

对于是否要在生产环境中使用 Docker，笔者也是刚刚在生产环境中使用 Docker 不到半年，在这里仅提供一些建议供大家参考：

- 1) 如果 Docker 出现不可控的风险，是否考虑了其他的解决方案。
- 2) 是否需要对 Docker 容器做资源限制，以及如何限制，如 CPU、内存、网络、磁盘等等。
- 3) 目前，Docker 对容器的安全管理做得不够完善，在应用到生产环境之前可以使用第三方工具来加强容器的安全管理。如使用 apparmor 对容器的能力进行限制、使用更加严格的 iptables 规则、禁止 root 用户登录、限制普通用户权限以及做好系统日志的记录。
- 4) 公司内部私有仓库的管理，镜像的管理问题是否解决。目前官方提供的私有仓库管理工具功能并不十分完善，若在生产环境中使用还需要更多的完善，由于国内用户连接官方源的网络不是十分稳定，所以这个问题需要在上线前想好解决办法。

本书的第三部分也会针对这些问题介绍一些目前比较流行的解决方案。

#### 17.4 本章小结

本章主要介绍 Docker 在个人学习、技术创业和中小型企业生产环境中的实践之道。在个人学习和技术创业部分，笔者从实践出发，同时基于 Docker 的特性和设计理念，给读者提供了一些 Docker 的创新用法。在中小型企业实践部分，笔者从企业生产环境出发，通过实际案例讲解 Docker 在标准化开发、测试、生产环境中的使用。

### 第三部分 *Part 3*

## 高级话题

- 第 18 章 Docker 核心技术
- 第 19 章 Docker 安全
- 第 20 章 高级网络配置
- 第 21 章 Docker 相关项目

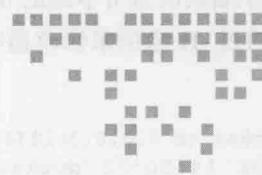
经过前两部分的讲解和实践，相信读者已经比较深入地了解了 Docker 的设计理念与应用操作。那么，Docker 是如何实现的？它目前有何问题？它的技术生态环境是否已经成长起来了？接下来，笔者将在第三部分介绍 Docker 的相关高级话题。

第 18 章将介绍 Docker 的核心实现技术，包括架构、命名空间、控制组、联合文件系统、虚拟网络等技术话题。

第 19 章将从命名空间、控制组、内核能力、服务端等角度来剖析 Docker 目前保障安全的相关手段。

第 20 章将具体讲解 Docker 使用网络的一些高级配置等，并分析底层实现的技术过程。

最后，在第 21 章笔者还将介绍 Docker 周边相关项目的进展情况，包括平台及服务应用、持续集成、容器管理和编程开发等。



## 第 18 章

# Docker 核心技术

*Chapter 18*

Docker 归根到底是一种容器虚拟化技术。

早期版本 Docker 的底层是基于成熟的 Linux Container(LXC) 技术实现的。自 Docker 0.9 版本起，Docker 除了继续支持 LXC 格式之外，还开始引入自家的 libcontainer (<https://github.com/docker/libcontainer>)，试图打造更通用的底层容器虚拟化库。

从操作系统功能上看，Docker 底层依赖的核心技术主要包括 Linux 操作系统的命名空间 (Namespaces)、控制组 (Control Groups)、联合文件系统 (Union File Systems) 和 Linux 虚拟网络支持。本章将介绍这些基本概念。

## 18.1 基本架构

Docker 采用了标准的 C/S 架构，包括客户端和服务端两大部分。

客户端和服务端既可以运行在一个机器上，也可通过 socket 或者 RESTful API 来进行通信，如图 18-1 所示。

### 1. 服务端

Docker daemon 一般在宿主主机后台运行，作为服务端接受来自客户的请求，并处理这些请求（创建、运行、分发容器）。在设计上，Docker daemon 是一个非常松耦合的架构，通过专门的 Engine 模块来分发管理各个来自客户端的任务。

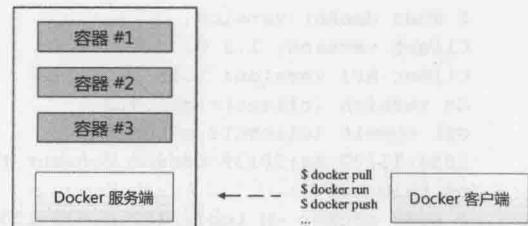


图 18-1 Docker 基本架构

Docker 服务端默认监听本地的 unix:///var/run/docker.sock 套接字，只允许本地的 root 用户访问。可以通过 -H 选项来修改监听的方式。例如，让服务端监听本地的 TCP 连接 1234 端口：

```
$ sudo docker -H 0.0.0.0:1234 -d &
2014/11/02 14:30:32 docker daemon: 1.3.0 c78088f; execdriver: native;
graphdriver:
[9f3e8962] +job serveapi(tcp://0.0.0.0:1234)
[9f3e8962] +job initserver()
[9f3e8962.initserver()] Creating server
2014/11/02 14:30:32 Listening for HTTP on tcp (0.0.0.0:1234)
2014/11/02 14:30:32 /!\ DON'T BIND ON ANOTHER IP ADDRESS THAN 127.0.0.1 IF YOU
DON'T KNOW WHAT YOU'RE DOING /!\
[9f3e8962] +job init_networkdriver()
[9f3e8962] -job init_networkdriver() = OK (0)
...
...
```

此外，Docker 还支持通过 HTTPS 认证方式来验证访问。

注：Ubuntu 系统中，Docker 服务端的默认启动配置文件在 /etc/default/docker。

## 2. 客户端

Docker 客户端则为用户提供一系列可执行命令，用户用这些命令实现与 Docker daemon 的交互。

用户使用的 Docker 可执行命令即为客户端程序。与 Docker daemon 不同的是，客户端发送命令后，等待服务端返回，一旦收到返回后，客户端立刻执行结束并退出。用户执行新的命令，需要再次调用客户端命令。

同样，客户端默认通过本地的 unix:///var/run/docker.sock 套接字向服务端发送命令。如果服务端没有监听到默认套接字，则需要客户端在执行命令的时候显式指定。例如，假定服务端监听在本地的 TCP 连接 1234 端口，只有通过 -H 参数指定了正确的信息才能连接到服务端：

```
$ sudo docker version
Client version: 1.3.0
Client API version: 1.15
Go version (client): go1.3.3
Git commit (client): c78088f
2014/11/02 14:30:39 Cannot connect to the Docker daemon. Is 'docker -d' running
on this host?
$ sudo docker -H tcp://127.0.0.1:1234 version
Client version: 1.3.0
Client API version: 1.15
Go version (client): go1.3.3
Git commit (client): c78088f
OS/Arch (client): linux/amd64
```

```
Server version: 1.3.0
Server API version: 1.15
Go version (server): go1.3.3
Git commit (server): c78088f
```

## 18.2 命名空间

命名空间（Namespace）是 Linux 内核针对实现容器虚拟化而引入的一个强大特性。

每个容器都可以拥有自己单独的命名空间，运行在其中的应用都像是在独立的操作系统中运行一样。命名空间保证了容器之间彼此互不影响。

我们知道，在操作系统中，包括内核、文件系统、网络、PID、UID、IPC、内存、硬盘、CPU 等资源，所有的资源都是应用进程直接共享的。要想实现虚拟化，除了要实现对内存、CPU、网络 IO、硬盘 IO、存储空间等的限制外，还要实现文件系统、网络、PID、UID、IPC 等等的相互隔离。前者相对容易实现一些，后者则需要宿主主机系统的深入支持。

随着 Linux 系统对于命名空间功能的逐步完善，Linux 软件工程师已经可以实现上文所述的所有需求，让某些进程在彼此隔离的命名空间中运行。虽然，这些进程都共用一个内核和某些运行时环境（runtime，例如一些系统命令和系统库），但是彼此是不可见的——它们都认为自己是独占系统的。

### 1. 进程命名空间

Linux 通过命名空间管理进程号，对于同一进程（同一个 task\_struct），在不同的命名空间中，看到的进程号不相同，每个进程命名空间有一套自己的进程号管理方法。进程命名空间是一个父子关系的结构，子空间中的进程对于父空间是可见的。新 fork 出的进程在父命名空间和子命名空间将分别有一个进程号来对应。

例如，查看 Docker 主进程的 pid 进程号是 5989：

```
$ ps -ef |grep docker
root      5989  5988  0 14:38 pts/6    00:00:00 docker -d
```

新建一个 Ubuntu 的“hello world”容器：

```
$ sudo docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"
2014/11/02 14:42:15 POST /v1.13/containers/create
[c5fe8ac9] +job create()
[error] mount.go:11 [warning]: couldn't run auplink before unmount: exec:
"apulink": executable file not found in $PATH
[c5fe8ac9] -job create() = OK (0)
ec559327572b5bf99d0f80b98ed3a3b62023844c7fdbea3f8caed4ffa5c62e86
...
```

查看新建容器进程的父进程，正是 Docker 主进程 5989：

```
$ ps -ef |grep while
root 6126 5989 0 14:41 ? 00:00:00 /bin/sh -c while true; do echo hello world;
sleep 1; done
```

## 2. 网络命名空间

如果有了 PID 命名空间，那么每个名字空间中的进程就可以相互隔离，但是网络端口还是共享本地系统的端口。

通过网络命名空间，可以实现网络隔离。一个网络命名空间为进程提供了一个完全独立的网络协议栈的视图。包括网络设备接口、IPv4 和 IPv6 协议栈、IP 路由表、防火墙规则，sockets 等等。这样每个容器的网络就能隔离开来。Docker 采用虚拟网络设备（Virtual Network Device）的方式，将不同命名空间的网络设备连接到一起。默认情况下，容器中的虚拟网卡将同本地主机上的 docker0 网桥连接在一起。如图 18-2 所示。

使用 brctl 工具，则可以看到桥接到宿主主机 docker0 网桥上的虚拟网口：

```
$ brctl show
bridge name      bridge id      STP enabled      interfaces
docker0          8000.56847afe9799    no
                                veth4148
                                vethd166
                                vethd533
```

## 3. IPC 命名空间

容器中进程交互还是采用了 Linux 常见的进程间交互方法（Interprocess Communication - IPC），包括信号量、消息队列和共享内存等。PID 命名空间和 IPC 命名空间可以组合起来一起使用，同一个 IPC 名字空间内的进程可以彼此可见，允许进行交互；不同空间的进程则无法交互。

## 4. 挂载命名空间

类似 chroot，将一个进程放到一个特定的目录执行。挂载命名空间允许不同命名空间的进程看到的文件结构不同，这样每个命名空间中的进程所看到的文件目录彼此被隔离。

## 5. UTS 命名空间

UTS（UNIX Time-sharing System）命名空间允许每个容器拥有独立的主机名和域名，从

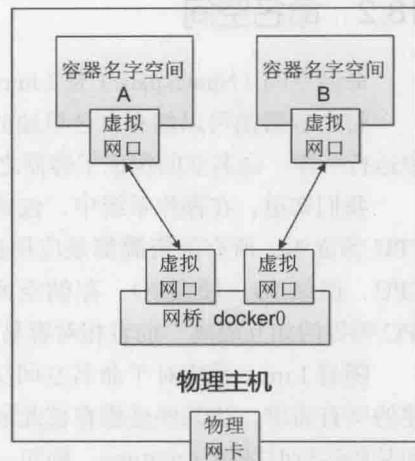


图 18-2 Docker 使用了网络命名空间

而可以虚拟出一个有独立主机名和网络空间的环境，就跟网络上一台独立的主机一样。

默认情况下，Docker 容器的主机名就是返回的容器 ID：

```
$ sudo docker ps
2014/11/02 15:00:29 GET /v1.13/containers/json
[c5fe8ac9] +job containers()
[c5fe8ac9] -job containers() = OK (0)
CONTAINER ID        IMAGE          COMMAND       CREATED      STATUS      PORTS      NAMES
ec559327572b        127.0.0.1:5000/ubuntu:latest   "/bin/sh -c 'while tr      18
minutes ago         Up 18 minutes    furious_goodall
$ docker inspect -f {{".Config.Hostname"}} ec5
2014/11/02 15:30:56 GET /v1.13/containers/ec5/json
[c5fe8ac9] +job container_inspect(ec5)
[c5fe8ac9] -job container_inspect(ec5) = OK (0)
ec559327572b
```

## 6. 用户命名空间

每个容器可以有不同的用户和组 id，也就是说可以在容器内使用特定的内部用户执行程序，而非本地系统上存在的用户。

每个容器内部都可以有 root 帐号，跟宿主主机不在一个命名空间。

## 18.3 控制组

控制组（CGroups）是 Linux 内核的一个特性，主要用来对共享资源进行隔离、限制、审计等。只有能控制分配到容器的资源，Docker 才能避免多个容器同时运行时的系统资源竞争。

控制组技术最早是由 Google 的程序员 2006 年提出，Linux 内核自 2.6.24 开始支持。

控制组可以提供对容器的内存、CPU、磁盘 IO 等资源进行限制和计费管理。

控制组的设计目标是为不同的应用情况提供统一的接口，从控制单一进程（比如 nice 工具）到系统级虚拟化（包括 OpenVZ、Linux-VServer、LXC 等）。

具体来看，控制组提供如下功能：

- 资源限制（Resource Limiting） 组可以设置为不超过设定的内存限制。比如：内存子系统可以为进程组设定一个内存使用上限，一旦进程组使用的内存达到限额再申请内存，就会发出 Out of Memory 警告。
- 优先级（Prioritization） 通过优先级让一些组优先得到更多的 CPU 等资源。
- 资源审计（Accounting） 用来统计系统实际上把多少资源用到适合的目的上，可以使 useracct 子系统记录某个进程组使用的 CPU 时间。
- 隔离（Isolation） 为组隔离名字空间，这样一个组不会看到另一个组的进程、网络连接和文件系统。
- 控制（Control） 挂起、恢复和重启动等操作。

安装 Docker 后，用户可以在 /sys/fs/cgroup/memory/docker/ 目录下看到对 Docker 组应用的各种限制项，包括：

```
$ cd /sys/fs/cgroup/memory/docker
$ ls
42352bb6c1d1c5c411be8fa04e97842da87d14623495189c4d865dfc444d12ae  memory.numa_stat
89a5c51c6d94a28dad4bb742662e15c99a0b4232ade7261df37aff3899b94490  memory.oom_control
cgroup.clone_children
cgroup.event_control
cgroup.procs
memory.failcnt
memory.force_empty
memory.limit_in_bytes
memory.max_usage_in_bytes
memory.move_charge_at_immigrate
tasks
memory.pressure_level
memory.soft_limit_in_bytes
memory.stat
memory.swappiness
memory.usage_in_bytes
memory.use_hierarchy
notify_on_release
```

用户可以通过修改这些文件值来控制组限制 Docker 应用资源。例如，通过下面的命令可限制 Docker 组中的所有进程使用的物理内存总量不超过 100 MB：

```
$ sudo echo 104857600 >/sys/fs/cgroup/memory/docker/memory.limit_in_bytes
```

进入对应的容器文件夹，可以看到对应容器的一些状态：

```
$ cd 42352bb6c1d1c5c411be8fa04e97842da87d14623495189c4d865dfc444d12ae/
$ ls
cgroup.clone_children  memory.max_usage_in_bytes      memory.stat
cgroup.event_control   memory.move_charge_at_immigrate  memory.swappiness
cgroup.procs           memory.numa_stat            memory.usage_in_bytes
memory.failcnt         memory.oom_control          memory.use_hierarchy
memory.force_empty     memory.pressure_level       notify_on_release
memory.limit_in_bytes  memory.soft_limit_in_bytes    tasks
$ cat memory.stat
cache 110592
rss 107286528
rss_huge 16777216
mapped_file 0
writeback 0
pgpgin 74766
pgpgout 52634
pgfault 115722
pgmajfault 0
inactive_anon 12288
active_anon 107384832
inactive_file 0
active_file 0
unevictable 0
hierarchical_memory_limit 18446744073709551615
total_cache 110592
total_rss 107286528
total_rss_huge 16777216
```

```

total_mapped_file 0
total_writeback 0
total_pgpgin 74766
total_pgpgout 52634
total_pgfault 115722
total_pgmajfault 0
total_inactive_anon 12288
total_active_anon 107384832
total_inactive_file 0
total_active_file 0
total_unevictable 0

```

在开发容器工具时，往往需要一些容器运行状态数据，这时就可以从这里得到更多的信息。

 **注意** 可以在创建或启动容器时候为每个容器指定资源的限制，例如使用 `-c|--cpu-shares[=0]` 参数来调整容器使用 CPU 的权重；使用 `-m|--memory[=MEMORY]` 参数来调整容器使用内存的大小。

## 18.4 联合文件系统

联合文件系统（UnionFS）是一种轻量级的高性能分层文件系统，它支持将文件系统中的修改信息作为一次提交，并层层叠加，同时可以将不同目录挂载到同一个虚拟文件系统下。

联合文件系统是实现 Docker 镜像的技术基础。镜像可以通过分层来进行继承。例如，用户基于基础镜像（没有父镜像的镜像被称为基础镜像）来制作各种不同的应用镜像。这些镜像共享同一个基础镜像层，提高了存储效率。此外，当用户改变了一个 Docker 镜像（比如升级程序到新的版本），则一个新的层（layer）会被创建。因此，用户不用替换整个原镜像或者重新建立，只需要添加新层即可。用户分发镜像的时候，也需要分发被改动的新层内容（增量部分）。这让 Docker 的镜像管理变得十分轻量级和快速。

Docker 中使用的 AUFS（Another Union File System，或 v2 版本往后的 Advanced Multi-layered Unification File System）就是一种联合文件系统。AUFS 支持为每一个成员目录（类似 Git 的分支）设定只读（readonly）、读写（readwrite）和写出（whiteout-able）权限，同时 AUFS 里有一个类似分层的概念，对只读权限的分支可以逻辑上进行增量地修改（不影响只读部分）。

当 Docker 利用镜像启动一个容器时，将利用镜像分配文件系统并且挂载一个新的可读写的层给容器，容器会在这个文件系统中创建，并且这个可读写的层被添加到镜像中。如图 18-3 所示。

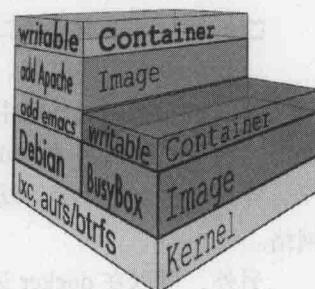


图 18-3 联合文件系统

Docker 目前支持的联合文件系统种类包括 AUFS、btrfs、vfs 和 DeviceMapper 等。

## 18.5 Docker 网络实现

Docker 的网络实现其实就是利用了 Linux 上的网络命名空间和虚拟网络设备（特别是 veth pair）。熟悉这两部分的基本概念，可以有助于理解 Docker 网络的实现过程。

### 1. 基本原理

直观上看，要实现网络通信，机器需要至少一个网络接口（物理接口或虚拟接口）与外界相通，并可以收发数据包；此外，如果不同子网之间要进行通信，需要额外的路由机制。

Docker 中的网络接口默认都是虚拟的接口。虚拟接口的最大优势就是转发效率极高。这是因为 Linux 通过在内核中进行数据复制来实现虚拟接口之间的数据转发，即发送接口的发送缓存中的数据包将被直接复制到接收接口的接收缓存中，而无需通过外部物理网络设备进行交换。对于本地系统和容器内系统来看，虚拟接口跟一个正常的以太网卡相比并无区别，只是它速度要快得多。

Docker 容器网络就很好地利用了 Linux 虚拟网络技术。它在本地主机和容器内分别创建一个虚拟接口，并让它们彼此连通（这样的一对接口叫做 veth pair）。如图 18-4 所示。

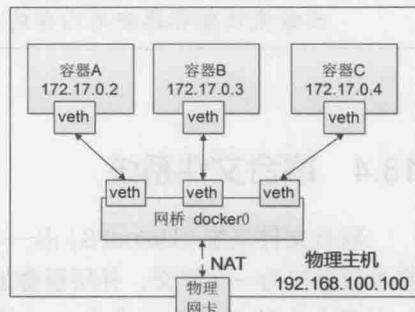


图 18-4 Docker 的网络实现

### 2. 网络创建过程

Docker 创建一个容器的时候，会具体执行如下操作：

- 创建一对虚拟接口，分别放到本地主机和新容器的命名空间中。
- 本地主机一端的虚拟接口连接到默认的 docker0 网桥或指定网桥上，并具有一个以 veth 开头的唯一名字，如 veth1234。
- 容器一端的虚拟接口将放到新创建的容器中，并修改名字作为 eth0。这个接口只在容器的命名空间可见。
- 从网桥可用地址段中获取一个空闲地址分配给容器的 eth0（例如 172.17.0.2/16），并配置默认路由网关为 docker0 网卡的内部接口 docker0 的 IP 地址（例如 172.17.42.1/16）。

完成这些之后，容器就可以使用它所能看到的 eth0 虚拟网卡来连接其他容器和访问外部网络。

另外，可以在 docker 运行的时候通过 `--net` 参数来指定容器的网络配置，有 4 个可选值 `bridge`、`host`、`container` 和 `none`：

- `--net=bridge`: 默认值，在 Docker 网桥上为容器创建新的网络栈。
- `--net=host`：告诉 Docker 不要将容器网络放到隔离的命名空间中，即不要容器化容器内的网络。此时容器使用本地主机的网络，它拥有完全的本地主机接口访问权限。容器进程可以跟主机其他 root 进程一样打开低范围的端口，可以访问本地网络服务比如 D-bus，还可以让容器做一些影响整个主机系统的事情，比如重启主机。因此使用这个选项的时候要非常小心。如果进一步的使用 `--privileged=true` 参数，容器甚至会被允许直接配置主机的网络堆栈。
- `--net=container`：让 Docker 将新建容器的进程放到一个已存在容器的网络栈中，新容器进程有自己的文件系统、进程列表和资源限制，但会和已存在的容器共享 IP 地址和端口等网络资源，两者进程可以直接通过 lo 环回接口通信。
- `--net=none`：让 Docker 将新容器放到隔离的网络栈中，但是不进行网络配置。之后，用户可以自己进行配置。

### 3. 网络配置细节

用户使用 `--net=none` 后，Docker 将不对容器网络进行配置。下面，将手动完成配置网络的整个过程。通过这个过程，可以了解到 Docker 配置网络的更多细节。

首先，启动一个 `/bin/bash` 容器，指定 `--net=none` 参数：

```
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@63f36fc01b5f:/#
```

在本地主机查找容器的进程 id，并为它创建网络命名空间：

```
$ sudo docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid
```

检查桥接网卡的 IP 和子网掩码信息：

```
$ ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...
```

创建一对“veth pair”接口 A 和 B，绑定 A 接口到网桥 docker0，并启用它：

```
$ sudo ip link add A type veth peer name B
$ sudo brctl addif docker0 A
$ sudo ip link set A up
```

将 B 接口放到容器的网络命名空间，命名为 eth0，启动它并配置一个可用 IP(桥接网段)和默认网关：

```
$ sudo ip link set B netns $pid
$ sudo ip netns exec $pid ip link set dev B name eth0
$ sudo ip netns exec $pid ip link set eth0 up
$ sudo ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0
$ sudo ip netns exec $pid ip route add default via 172.17.42.1
```

以上就是 Docker 配置网络的具体过程。

当容器终止后，Docker 会清空容器，容器内的网络接口会随网络命名空间一起被清除，A 接口也被自动从 docker0 卸载并清除。

此外，在删除 /var/run/netns/ 下的内容之前，用户可以使用 ip netns exec 命令在指定网络命名空间中进行配置，从而影响容器内的网络。

## 18.6 本章小结

本章具体剖析了 Docker 实现的一些核心技术，包括它的基本架构，以及所依赖的 Linux 操作系统中的命名空间、控制组、联合文件系统、虚拟网络支持等。

从本章的讲解中读者可以看到，Docker 的优秀特性与 Linux 操作系统的强大、特别是与 Linux 上成熟的容器技术支持是分不开的。在实际使用 Docker 容器的过程中，还将涉及如何调整系统配置来优化容器性能，这些都需要有丰富的 Linux 系统运维知识和实践经验。

## 第 19 章 Docker 安全

Docker 是在 Linux 操作系统层面上的虚拟化实现，运行在容器内的进程，跟运行在本地系统中的进程，本质上并无区别，不合适的安全策略将可能给本地系统带来风险。因此，Docker 的安全性在生产环境中是十分关键的衡量因素。

Docker 容器的安全性，很大程度上其实依赖于 Linux 系统自身，因此在评估 Docker 的安全性时，主要考虑下面几个方面：

- Linux 内核的命名空间机制提供的容器隔离安全。
- Linux 控制组机制对容器资源的控制能力安全。
- Linux 内核的能力机制所带来的操作权限安全。
- Docker 程序（特别是服务端）本身的抗攻击性。
- 其他安全增强机制（包括 AppArmor、SELinux 等）对容器安全性的影响。

### 19.1 命名空间隔离的安全

Docker 容器和 LXC 容器在实现上很相似，所提供的安全特性也基本一致。当用 `docker run` 启动一个容器时，Docker 将在后台为容器创建一个独立的命名空间。

命名空间提供了最基础也是最直接的隔离，在容器中运行的进程不会被运行在本地主机上的进程和其他容器通过正常渠道发现和影响。

例如，通过命名空间机制，每个容器都有自己独有的网络栈，意味着它们不能访问其他容器的套接字（sockets）或接口。当然，容器默认可以与本地主机网络连通，如果主机系统上做了相应的交换设置，容器可以像跟主机交互一样的和其他容器交互。启动容器时，指定公共端口或使用连接系统，容器可以相互通信了（用户可以根据配置来限制通信的策略）。

从网络架构的角度来看，所有的容器实际上是通过本地主机的网桥接口（Docker0）进行相互通信，就像物理机器通过物理交换机通信一样。

那么，Linux 内核中实现命名空间（特别是网络命名空间）的机制是否足够成熟呢？

Linux 内核从 2.6.15 版本（2008 年 7 月发布）开始引入命名空间，至今经历了数年的演化和改进，并应用于诸多大型生产系统中。

实际上，命名空间的想法和设计提出的时间要更早，最初是 OpenVZ 项目的重要特性。OpenVZ 项目早在 2005 年就已经正式发布，其设计和实现更加成熟。

当然，与虚拟机方式相比，通过命名空间来实现的隔离并不是那么绝对。运行在容器中的应用可以直接访问系统内核和部分系统文件。因此，用户必须保证容器中应用是安全可信的（这跟保证运行在系统中的软件是可信的一个道理），否则本地系统将可能受到威胁。实际上，Docker 自 1.30 版本起对镜像管理引入了签名系统，用户可以通过签名来验证镜像的完整性和正确性。

## 19.2 控制组资源控制的安全

控制组是 Linux 容器机制中的另外一个关键组件，它负责实现资源的审计和限制。当用 docker run 启动一个容器时，Docker 将在后台为容器创建一个独立的控制组策略集合。

控制组机制始于 2006 年，Linux 内核从 2.6.24 版本开始被引入。

它提供了很多有用的特性；以及确保各个容器可以公平地分享主机的内存、CPU、磁盘 IO 等资源；当然，更重要的是，控制组确保了当发生在容器内的资源压力不会影响到本地主机系统和其他容器。

尽管控制组不负责隔离容器之间相互访问、处理数据和进程，但是它在防止拒绝服务攻击（DDoS）方面是必不可少的。尤其是在多用户的平台（比如公有或私有的 PaaS）上，控制组十分重要。例如，当某些应用容器出现异常的时候，可以保证本地系统和其他容器正常运行而不受影响。

## 19.3 内核能力机制

能力机制（Capability）是 Linux 内核一个强大的特性，可以提供细粒度的权限访问控制。传统的 Unix 系统对进程权限只有根权限（用户 id 为 0，即为 root 用户）和非根权限（用户非 root 用户）两种。

Linux 内核自 2.2 版本起支持能力机制，它将权限划分为更加细粒度的操作能力，既可以作用在进程上，也可以作用在文件上。

例如，一个 Web 服务进程只需要绑定一个低于 1024 的端口的权限，并不需要完整的 root 权限。那么它只需要被授权 net\_bind\_service 能力即可。此外，还有很多其他的类似能力

来避免进程获取 root 权限。

默认情况下，Docker 启动的容器被严格限制只允许使用内核的一部分能力，包括 chown、dac\_override、fowner、kill、setgid、setuid、setpcap、net\_bind\_service、net\_raw、sys\_chroot、mknod、setfcap、audit\_write 等。

使用能力机制对加强 Docker 容器的安全性有很多好处。通常，在服务器上会运行一堆需要特权权限的进程，包括有 ssh、cron、syslogd、硬件管理工具模块（例如负载模块）、网络配置工具等等。容器跟这些进程是不同的，因为几乎所有的特权进程都由容器以外的支持系统来进行管理。

- ssh 访问被宿主主机上的 ssh 服务来管理。
- cron 通常应该作为用户进程执行，权限交给使用它服务的应用来处理。
- 日志系统可由 Docker 或第三方服务管理。
- 硬件管理无关紧要，容器中也就无需执行 udevd 以及类似服务。
- 网络管理也都在主机上设置，除非特殊需求，容器不需要对网络进行配置。

从上面的例子可以看出，大部分情况下，容器并不需要“真正的”root 权限，容器只需要少数的能力即可。为了加强安全，容器可以禁用一些没必要的权限。

- 完全禁止任何文件挂载操作。
- 禁止直接访问本地主机的套接字。
- 禁止访问一些文件系统的操作，比如创建新的设备、修改文件属性等。
- 禁止模块加载。

这样，就算攻击者在容器中取得了 root 权限，也不能获得本地主机的较高权限，能进行的破坏也有限。

不恰当地分配了内核能力，会导致容器内应用获取破坏本地系统的权限。例如，早期的 Docker 版本曾经不恰当的继承 CAP\_DAC\_READ\_SEARCH 能力，导致容器内进程可以通过系统调用访问到本地系统的任意文件目录。

默认情况下，Docker 采用“白名单”机制，禁用“必需功能”之外的其他权限。当然，用户也可以根据自身需求来为 Docker 容器启用额外的权限。

## 19.4 Docker 服务端的防护

使用 Docker 容器的核心是 Docker 服务端。Docker 服务的运行目前还需要 root 权限的支持，因此服务端安全性十分关键。

首先，必须确保只有可信的用户才可以访问到 Docker 服务。Docker 允许用户在主机和容器间共享文件夹，同时不需要限制容器的访问权限，这就容易让容器突破资源限制。例如，恶意用户启动容器的时候将主机的根目录 / 映射到容器的 /host 目录中，那么容器理论上就可以对主机的文件系统进行任意修改了。事实上，几乎所有虚拟化系统都允许类似的资源

共享，而没法阻止恶意用户共享主机根文件系统到虚拟机系统。

这将会造成很严重的安全后果。因此，当提供容器创建服务时（例如通过一个 web 服务器），要更加注意进行参数的安全检查，防止恶意的用户用特定参数来创建一些破坏性的容器。

为了加强对服务端的保护，Docker 的 RESTAPI（客户端用来跟服务端通信的接口）在 0.5.2 之后使用本地的 Unix 套接字机制替代了原先绑定在 127.0.0.1 上的 TCP 套接字，因为后者容易遭受跨站脚本攻击。现在用户使用 Unix 权限检查来加强套接字的访问安全。

用户仍可以利用 HTTP 提供 REST API 访问。建议使用安全机制，确保只有可信的网络或 VPN 网络，或证书保护机制（例如受保护的 stunnel 和 ssl 认证）下的访问可以进行。此外，还可以使用 HTTPS 和证书来加强保护。

最近改进的 Linux 命名空间机制将可以实现使用非 root 用户来运行全功能的容器。这将从根本上解决了容器和主机之间共享文件系统而引起的安全问题。

目前，Docker 自身改进安全防护的目标是实现以下两个重要安全特性：

- 将容器的 root 用户映射到本地主机上的非 root 用户，减轻容器和主机之间因权限提升而引起的安全问题。
- 允许 Docker 服务端在非 root 权限下运行，利用安全可靠的子进程来代理执行需要特权权限的操作。这些子进程将只允许在限定范围内进行操作，例如仅仅负责虚拟网络设定或文件系统管理、配置操作等。

## 19.5 其他安全特性

除了能力机制之外，还可以利用一些现有的安全软件或机制来增强使用 Docker 的安全性，例如 TOMOYO，AppArmor，SELinux，GRSEC 等。

Docker 当前默认只启用了能力机制。用户可以选择启用更多的安全方案来加强 Docker 主机的安全，例如：

- 在内核中启用 GRSEC 和 PAX，这将增加更多的编译和运行时的安全检查；并且通过地址随机化机制来避免恶意探测等。启用该特性不需要 Docker 进行任何配置。
- 使用一些有增强安全特性的容器模板，比如带 AppArmor 的模板和 Redhat 带 SELinux 策略的模板。这些模板提供了额外的安全特性。
- 用户可以自定义更加严格的访问控制机制来定制安全策略。

此外，在将文件系统挂载到容器内部时候，可以通过配置只读（read-only）模式来避免容器内的应用通过文件系统破坏外部环境，特别是一些系统运行状态相关的目录，包括但不限于 /proc/sys、/proc/irq、/proc/bus 等等。这样，容器内应用进程可以获取所需要的系统信息，但无法对它们进行修改。

## 19.6 本章小结

总体来看，基于Linux上成熟的安全机制以及Apparmor, SELinux, GRSEC等安全机制，Docker容器还是比较安全的，特别是在容器内不使用root权限来运行进程的话。

但是任何技术层面实现的安全都需要用户合理的使用才能得到巩固，在使用Docker过程中，需要注意如下几方面：

- 在使用Docker容器运行应用的时候，一定要牢记容器自身所提供的隔离性其实并没有那么完善，需要加强对容器内应用的安全审查。容器即应用，保障应用安全的各种手段，都应该进行合理地利用。
- 采用专用的服务器来运行Docker服务端和相关的管理服务（例如管理服务比如ssh监控和进程监控、管理工具nrpe、collectd等），并对该服务器启用最高级别的安全机制。而把其他的业务服务都放到容器中去运行。
- 将运行Docker容器的机器划分为不同的组，互相信任的机器放到同一个组内；组之间进行安全隔离；同时进行定期的安全检查。
- 随着容器大规模地使用和集成，甚至组成容器集群。需要考虑在容器网络上进行必备的安全防护，避免诸如DDoS、ARP攻击、规则表攻击等网络安全威胁，这也是生产环境需要关注的重要问题。

*Chapter 20*

## 第 20 章 高级网络配置

本章将介绍 Docker 的一些关于网络的高级知识，包括网络的启动和配置参数、DNS 的使用配置、容器访问和端口映射的相关实现。

接下来，笔者将介绍在一些场景中，Docker 所有的网络定制配置。以及通过 Linux 命令来调整、补充、甚至替换 Docker 默认的网络配置。

最后，将介绍关于 Docker 网络的一些工具和项目。

### 20.1 网络启动与配置参数

#### 1. 基本过程

Docker 启动时会在主机上自动创建一个 docker0 虚拟网桥，实际上是一个 Linux 网桥，可以理解为一个软件交换机。它会在挂载其上的接口之间进行转发。

同时，Docker 随机分配一个本地未占用的私有网段（在 RFC1918 中定义）中的一个地址给 docker0 接口。比如典型的 172.17.42.1，掩码为 255.255.0.0。此后启动的容器内的网口也会自动分配一个同一网段（172.17.0.0/16）的地址。

当创建一个 Docker 容器的时候，同时会创建了一对 veth pair 接口（当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包）。这对接口一端在容器内，即 eth0；另一端在本地并被挂载到 docker0 网桥，名称以 veth 开头（例如 vethAQI2QT）。通过这种方式，主机可以跟容器通信，容器之间也可以相互通信。如此一来，Docker 就创建了在主机和所有容器之间一个虚拟共享网络。如图 20-1 所示。

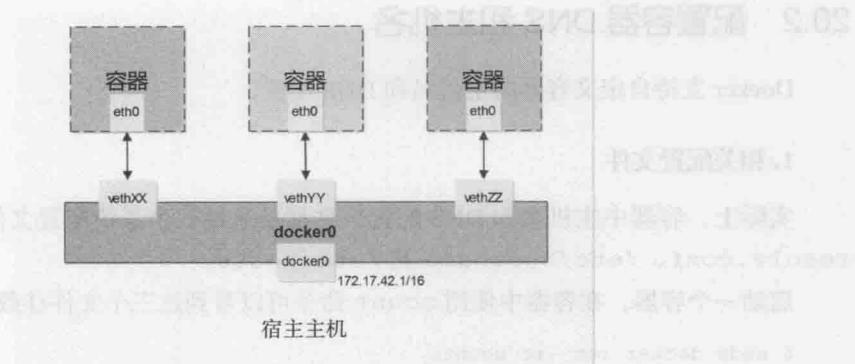


图 20-1 Docker 中的网络

## 2. 网络相关参数

下面是跟 Docker 网络相关的命令参数。其中有些命令选项只有在 Docker 服务启动的时候才能配置，而且不能马上生效：

- `-b BRIDGE or --bridge=BRIDGE`——指定容器挂载的网桥。
- `--bip=CIDR`——定制 docker0 的掩码。
- `-H SOCKET... or --host=SOCKET...`——Docker 服务端接收命令的通道。
- `--icc=true|false`——是否支持容器之间进行通信。
- `--ip-forward=true|false`——启用 net.ipv4.ip forward，即打开转发功能。
- `--iptables=true|false`——禁止 Docker 添加 iptables 规则。
- `--mtu=BYTES`——容器网络中的 MTU。

下面两个命令选项既可以在启动服务时指定，也可以 Docker 容器启动（`docker run`）时候指定。在 Docker 服务启动的时候指定则会成为默认值，后续执行 `docker run` 时可以覆盖设置的默认值：

- `--dns=IP_ADDRESS...`——使用指定的 DNS 服务器。
- `--dns-search=DOMAIN...`——指定 DNS 搜索域。

最后这些选项只能在 `docker run` 执行时使用，因为它是针对容器的特性内容：

- `-h HOSTNAME or --hostname=HOSTNAME`——配置容器主机名。
- `--link=CONTAINER_NAME:ALIAS`——添加到另一个容器的连接。
- `--net=bridge|none|container:NAME_or_ID|host`——配置容器的桥接模式。
- `-p SPEC or --publish=SPEC`——映射容器端口到宿主主机。
- `-P or --publish-all=true|false`——映射容器所有端口到宿主主机。

## 20.2 配置容器 DNS 和主机名

Docker 支持自定义容器的主机名和 DNS 配置。

### 1. 相关配置文件

实际上，容器中主机名和 DNS 配置信息都是通过三个系统配置文件来维护的：/etc/resolv.conf、/etc/hostname 和 /etc/hosts。

启动一个容器，在容器中使用 mount 命令可以看到这三个文件挂载信息：

```
$ sudo docker run -it ubuntu
root@75dbd6685305:/# mount
...
/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)
...
```

其中，/etc/resolv.conf 文件在创建容器时候，默认会与宿主机 /etc/resolv.conf 文件内容保持一致：

```
root@75dbd6685305:/# cat /etc/resolv.conf
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#      DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
nameserver 8.8.8.8
search dockerpool.com
```

/etc/hosts 文件中默认只记录了容器自身的一些地址和名称：

```
root@75dbd6685305:/# cat /etc/hosts
172.17.0.2    75dbd6685305
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
127.0.0.1    localhost
```

/etc/hostname 文件则记录了容器的主机名：

```
root@75dbd6685305:/# cat /etc/hostname
75dbd6685305
```

### 2. 容器内修改配置文件

Docker 1.2.0 开始支持在运行中的容器里直接编辑 /etc/hosts、/etc/hostname 和

/etc/resolve.conf 文件。

但是这些修改是临时的，只在运行的容器中保留，容器终止或重启后并不会被保存下来。也不会被 docker commit 提交。

### 3. 通过参数指定

如果用户想要自定义容器的配置，可以在创建或启动容器时利用下面的参数指定：

- 指定主机名 -h HOSTNAME or --hostname=HOSTNAME。设定容器的主机名，它会被写到容器内的 /etc/hostname 和 /etc/hosts。但这个主机名只有容器内能看到，在容器外部则看不到，既不会在 docker ps 中显示，也不会在其他的容器的 /etc/hosts 看到。
- 记录其他容器主机名 --link=CONTAINER\_NAME:ALIAS。选项会在创建容器的时候，添加一个所连接容器的主机名到容器内 /etc/hosts 文件中。这样，新创建容器可以直接使用主机名来与所连接容器通信。
- 指定 DNS 服务器 --dns=IP\_ADDRESS。添加 DNS 服务器到容器的 /etc/resolv.conf 中，容器会用指定的服务器来解析所有不在 /etc/hosts 中的主机名。
- 指定 DNS 搜索域 --dns-search=DOMAIN。设定容器的搜索域，当设定搜索域为 .example.com 时，在搜索一个名为 host 的主机时，DNS 不仅搜索 host，还会搜索 host.example.com。

## 20.3 容器访问控制

容器的访问控制，主要通过 Linux 上的 iptables 防火墙软件来进行管理和实现。iptables 是 Linux 系统流行的防火墙软件，在大部分发行版中都自带。

### 1. 容器访问外部网络

从前面的描述中，我们知道容器默认指定了网关为 docker0 网桥上的 docker0 内部接口。docker0 内部接口同时也是宿主机的一个本地接口。因此，容器默认情况下是可以访问到宿主机本地的。

更进一步地，容器要想通过宿主机访问到外部网络，需要宿主机进行转发。

在宿主机 Linux 系统中，检查转发是否打开：

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

如果为 0，说明没有开启转发，则需要手动打开：

```
$ sudo sysctl -w net.ipv4.ip_forward=1
```

更简单的，在启动 Docker 服务的时候设定 `--ip-forward=true`，Docker 服务会自动打开宿主机系统的转发服务。

## 2. 容器之间访问

容器之间相互访问需要以下两方面的支持。

- 网络拓扑是否已经连通。默认情况下，所有容器都会连接到 `docker0` 网桥上，这意味着默认情况下拓扑是互通的。
- 本地系统的防火墙软件 `iptables` 是否允许访问通过。这取决于防火墙的默认规则是允许（大部分情况）还是禁止。

### 访问所有端口

当启动 Docker 服务时候，默认会添加一条“允许”转发策略到 `iptables` 的 FORWARD 链上。通过配置 `--icc=true|false`（默认值为 `true`）参数可以控制默认的策略。

为了安全考虑，可以在 `/etc/default/docker` 文件中配置 `DOCKER_OPTS=--icc=false` 来默认禁止容器之间的相互访问。

同时，如果启动 Docker 服务时手动指定 `--iptables=false` 参数则不会修改宿主机系统上的 `iptables` 规则。

### 访问指定端口

在通过 `--icc=false` 禁止容器间相互访问后，仍可以通过 `--link=CONTAINER_NAME:ALIAS` 选项来允许访问指定容器的开放端口。

例如，在启动 Docker 服务时，可以同时使用 `icc=false--iptables=true` 参数来配置容器间禁止访问，并允许 Docker 自动修改系统中的 `iptables` 规则。

此时，系统中的 `iptables` 规则可能是类似如下规则，禁止所有转发流量：

```
$ sudo iptables -nL
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  0.0.0.0/0            0.0.0.0/0
...

```

之后，启动容器（`docker run`）时使用 `--link=CONTAINER_NAME:ALIAS` 选项。Docker 会在 `iptables` 中为两个互联容器分别添加一条 `ACCEPT` 规则，允许相互访问开放的端口（取决于 Dockerfile 中的 `EXPOSE` 行）。

此时，`iptables` 的规则可能是类似如下规则：

```
$ sudo iptables -nL
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT    tcp  --  172.17.0.2            172.17.0.3        tcp spt:80

```

```
ACCEPT  tcp  --  172.17.0.3      172.17.0.2      tcp dpt:80
DROP   all  --  0.0.0.0/0      0.0.0.0/0
```



--link=CONTAINER\_NAME:ALIAS 中的 CONTAINER\_NAME 目前必须是 Docker 自动分配的容器名，或使用 --name 参数指定的名字。不能为容器 -h 参数配置的主机名。

## 20.4 映射容器端口到宿主主机的实现

默认情况下，容器可以主动访问到外部网络的连接，但是外部网络无法访问到容器。

### 1. 容器访问外部实现

假设容器内部的网络地址为 172.17.0.2，本地网络地址为 10.0.2.2。容器要能访问外部网络，源地址不能为 172.17.0.2，需要进行源地址映射（SourceNAT，SNAT），修改为本地系统的 IP 地址 10.0.2.2。映射是通过 iptables 的源地址伪装操作实现的。

查看主机 nat 表上 POSTROUTING 链的规则。该链负责网包要离开主机前，改写其源地址：

```
$ sudo iptables -t nat -nvL POSTROUTING
Chain POSTROUTING (policy ACCEPT 12 packets, 738 bytes)
pkts bytes target     prot opt in     out     source               destination
...
0      0 MASQUERADE  all  --  *      !docker0  172.17.0.0/16  0.0.0.0/0
...
```

其中，上述规则将所有源地址在 172.17.0.0/16 网段，且不是从 docker0 接口发出的流量（即从容器中出来的流量），动态伪装为从系统网卡发出。MASQUERADE 行动跟传统 SNAT 行动的好处是它能动态从网卡获取地址。

### 2. 外部访问容器实现

容器允许外部访问，可以在 dockerrun 时候通过 -p 或 -P 参数来启用。

不管用那种办法，其实也是在本地的 iptable 的 nat 表中添加相应的规则，将访问外部 IP 地址的网包进行目标地址 DNAT，将目标地址修改为容器的 IP 地址。

以一个开放 80 端口的 web 容器为例，使用 -P 时，会自动映射本地 49000~49900 范文内的端口随机端口到容器的 80 端口：

```
$ sudo iptables -t nat -nvL
...
Chain PREROUTING (policy ACCEPT 236 packets, 33317 bytes)
```

```

pkts bytes target      prot opt in      out      source      destination
 567 30236 DOCKER     all   --  *      *      0.0.0.0/0      0.0.0.0/0
ADDRTYPE match dst-type LOCAL

Chain DOCKER (2 references)
pkts bytes target      prot opt in      out      source      destination
  0    0 DNAT          tcp   -- !docker0 *      0.0.0.0/0      0.0.0.0/0
tcp dpt:49153 to:172.17.0.2:80
...

```

可以看到，nat 表中涉及两条链，PREROUTING 链负责包到达网络接口时，改写其目的地址。其中规则将所有流量都扔到 DOCKER 链。而 DOCKER 链中将所有不是从 docker0 进来的网包（意味着不是本地主机产生），将目标端口为 49153 的，修改目标地址为 172.17.0.2，目标端口修改为 80。

使用 -p 80:80 时，与上面类似，只是本地端口也为 80：

```

$ sudo iptables -t nat -nvL
...
Chain PREROUTING (policy ACCEPT 236 packets, 33317 bytes)
pkts bytes target      prot opt in      out      source      destination
 567 30236 DOCKER     all   --  *      *      0.0.0.0/0      0.0.0.0/0
ADDRTYPE match dst-type LOCAL

Chain DOCKER (2 references)
pkts bytes target      prot opt in      out      source      destination
  0    0 DNAT          tcp   -- !docker0 *      0.0.0.0/0      0.0.0.0/0
tcp dpt:80 to:172.17.0.2:80
...

```

注意以下问题：

- 这里的规则映射了 0.0.0.0，意味着将接受主机来自所有网络接口上的流量。用户可以通过 -p IP:host\_port:container\_port 或 -p IP::port 来指定绑定的外部网络接口，以制定更严格的访问规则。
- 如果希望映射永久绑定到某个固定的 IP 地址，可以在 Docker 配置文件 /etc/default/docker 中指定 DOCKER\_OPTS="--ip=IP\_ADDRESS"，之后重启 Docker 服务即可生效。

## 20.5 配置 docker0 网桥

Docker 服务默认会创建一个名称为 docker0 的 Linux 网桥（其上有一个 docker0 内部接口），它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和本地主机都放到同一个物理网络。

Docker 默认指定了 docker0 接口的 IP 地址和子网掩码，让主机和容器之间可以通过

网桥相互通信，它还给出了 MTU（接口允许接收的最大传输单元），通常是 1500 Bytes，或宿主主机网络路由上支持的默认值。这些值都可以在服务启动的时候进行配置。

□ --bip=CIDR——IP 地址加掩码格式，例如 192.168.1.5/24。

□ --mtu=BYTES——覆盖默认的 Docker mtu 配置。

也可以在配置文件中配置 DOCKER\_OPTS，然后重启服务。由于目前 Docker 网桥是 Linux 网桥，用户可以使用 brctl show 来查看网桥和端口连接信息：

```
$ sudo brctl show
bridge name      bridge id          STP enabled     interfaces
docker0          8000.3a1d7362b4ee    no             veth65f9
                                         vethddaa6
```

 **注意** brctl 命令在 Debian、Ubuntu 中可以使用 sudo apt-get install bridge-utils 来安装。

每次创建一个新容器的时候，Docker 从可用的地址段中选择一个空闲的 IP 地址分配给容器的 eth0 端口。并且使用本地主机上 docker0 接口的 IP 作为容器的默认网关：

```
$ sudo docker run -i -t --rm base /bin/bash
$ ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
        valid_lft forever preferred_lft forever
$ ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0 proto kernel scope link src 172.17.0.3
$ exit
```

目前，Docker 不支持在启动容器时候指定 IP 地址。

 **注意** 实际上，Linux 网桥可以很容易替换为 OpenvSwitch 等功能更强大的网桥实现，可以支持 VLAN 等属性。

## 20.6 自定义网桥

除了默认的 docker0 网桥，用户也可以指定网桥来连接各个容器。

在启动 Docker 服务的时候，使用 -b BRIDGE 或 --bridge=BRIDGE 来指定使用的网桥。

如果服务已经运行，那需要先停止服务，并删除旧的网桥：

```
$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
```

然后创建一个网桥 bridge0：

```
$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.5.1/24 dev bridge0
$ sudo ip link set dev bridge0 up
```

查看确认网桥创建并启动：

```
$ ip addr show bridge0
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
        inet 192.168.5.1/24 scope global bridge0
            valid_lft forever preferred_lft forever
```

配置 Docker 服务，将默认桥接到创建的网桥上：

```
$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker start
```

启动 Docker 服务。新建一个容器，可以看到它已经桥接到了 bridge0 上。

可以继续用 brctl show 命令查看桥接的信息。另外，在容器中可以使用 ip addr 和 ip route 命令来查看 IP 地址配置和路由信息。

## 20.7 创建一个点到点连接

默认情况下，Docker 会将所有容器连接到由 docker0 提供的虚拟子网中。

用户有时候需要两个容器之间可以直连通信，而不用通过主机网桥进行桥接。

解决办法很简单：创建一对 peer 接口，分别放到两个容器中，配置成点到点链路类型即可。

首先启动两个容器：

```
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@1f1f4c1f931a:/
$ sudo docker run -i -t --rm --net=none base /bin/bash
root@12e343489d2f:/#
```

找到进程号，然后创建网络命名空间的跟踪文件：

```
$ sudo docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a
2989
$ sudo docker inspect -f '{{.State.Pid}}' 12e343489d2f
```

```

3004
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/2989/ns/net /var/run/netns/2989
$ sudo ln -s /proc/3004/ns/net /var/run/netns/3004

创建一对 peer 接口，然后配置路由：

$ sudo ip link add A type veth peer name B
$ sudo ip link set A netns 2989
$ sudo ip netns exec 2989 ip addr add 10.1.1.1/32 dev A
$ sudo ip netns exec 2989 ip link set A up
$ sudo ip netns exec 2989 ip route add 10.1.1.2/32 dev A

$ sudo ip link set B netns 3004
$ sudo ip netns exec 3004 ip addr add 10.1.1.2/32 dev B
$ sudo ip netns exec 3004 ip link set B up
$ sudo ip netns exec 3004 ip route add 10.1.1.1/32 dev B

```

现在这两个容器就可以相互 ping 通，并成功建立连接。点到点链路不需要子网和子网掩码。此外，也可以不指定 --net=none 来创建点到点链路。这样容器还可以通过原先的网络来通信。

利用类似的办法，可以创建一个只跟主机通信的容器。但是一般情况下，更推荐使用 --icc=false 来关闭容器之间的通信。

## 20.8 工具和项目

围绕 Docker 网络的管理和使用，现在已经诞生了一些方便用户操作的工具和项目，包括 pipework、playground，以及 Docker 社区正在推进的 libswarm 项目。

### 1. pipework

Jérôme Petazzoni 编写了一个叫 pipework 的 shell 脚本封装了一些操作，可以简化在比较复杂的场景对容器连接的操作命令。

例如，分别启动两个终端，在其中创建两个测试容器 c1 和 c2，并查看网卡配置。

容器 c1：

```

$ sudo docker run --name c1 -it ubuntu
root@e1c70b140f1f:/# ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:05
          inet addr:172.17.0.5 Bcast:0.0.0.0 Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:5/64 Scope:Link
            UP BROADCAST RUNNING MTU:1500 Metric:1
            RX packets:8 errors:0 dropped:0 overruns:0 frame:0
            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0

```

```

        collisions:0 txqueuelen:1000
        RX bytes:648 (648.0 B)   TX bytes:648 (648.0 B)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)   TX bytes:0 (0.0 B)

```

**容器 c2:**

```

$ sudo docker run --name c2 -it ubuntu
root@2c47b6af0c3f:/# ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:06
        inet addr:172.17.0.6 Bcast:0.0.0.0 Mask:255.255.0.0
        inet6 addr: fe80::42:acff:fe11:6/64 Scope:Link
          UP BROADCAST RUNNING MTU:1500 Metric:1
          RX packets:5 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:418 (418.0 B)   TX bytes:418 (418.0 B)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)   TX bytes:0 (0.0 B)

```

利用 pipework 为容器 c1 和 c2 添加新的网卡，并将它们连接到新创建的 br1 网桥上：

```

$ sudo pipework br1 c1 192.168.1.1/24
$ sudo pipework br1 c2 192.168.1.2/24

```

此时在主机系统中查看网桥信息，会发现新创建的网桥 br1，并且有两个 veth 端口连接上去。

```

$ sudo brctl show
bridge name      bridge id      STP enabled      interfaces
br1              8000.868b605fc7a4    no            veth1pl17805
                                         veth1pl17880
docker0          8000.56847afe9799    no            veth89934d8

```

**容器 c1:**

```
root@10ef0bd4cb77:/# ifconfig
```

```

eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:05
          inet addr:172.17.0.5 Bcast:0.0.0.0 Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:5/64 Scope:Link
            UP BROADCAST RUNNING MTU:1500 Metric:1
            RX packets:17 errors:0 dropped:0 overruns:0 frame:0
            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:1338 (1.3 KB) TX bytes:648 (648.0 B)

eth1      Link encap:Ethernet HWaddr 0a:95:e0:8b:7c:d3
          inet addr:192.168.1.1 Bcast:0.0.0.0 Mask:255.255.255.0
          inet6 addr: fe80::895:e0ff:fe8b:7cd3/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:15 errors:0 dropped:0 overruns:0 frame:0
            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:1206 (1.2 KB) TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING MTU:65536 Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

### 容器 c2:

```

root@2c47b6af0c3f:/# ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:06
          inet addr:172.17.0.6 Bcast:0.0.0.0 Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:6/64 Scope:Link
            UP BROADCAST RUNNING MTU:1500 Metric:1
            RX packets:13 errors:0 dropped:0 overruns:0 frame:0
            TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:1048 (1.0 KB) TX bytes:856 (856.0 B)

eth1      Link encap:Ethernet HWaddr a6:9e:6e:a0:4a:44
          inet addr:192.168.1.2 Bcast:0.0.0.0 Mask:255.255.255.0
          inet6 addr: fe80::a49e:6eff:fea0:4a44/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:8 errors:0 dropped:0 overruns:0 frame:0
            TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:648 (648.0 B) TX bytes:690 (690.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host

```

```
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

此时，容器 c1 和 c2 可以通过子网 192.168.1.0/16 相互连通。

另外，pipework 还支持指定容器内的网卡名称、MAC 地址、网络掩码和网关等配置，甚至通过 macvlan 连接容器到本地物理网卡，实现跨主机通信。

pipework 代码只有 200 多行，建议进行阅读，有助于理解如何利用 Linux 系统上的 iproute 等工具实现容器连接的配置。实际上，Docker 在实现上也是采用了相同的底层机制。目前，Docker 社区在推动具有更多功能的 libswarm 项目，以实现更标准的接口。

## 2. playground

Brandon Rhodes 创建了一个提供完整的 Docker 容器网络拓扑管理的 Python 库，包括路由、NAT 防火墙；以及一些提供 HTTP、SMTP、POP、IMAP、Telnet、SSH、FTP 的服务器的实现。

基于 playground，用户可以提前配置好容器的拓扑，然后一条命令，启动多个容器并构成互联关系。该项目跟下一章要介绍的 fig 项目功能有重复，被替代的可能性较大。

## 3. libswarm 项目

该项目诞生于 2014 年 3 月，遵循 Apache 2.0 许可，目前项目仍在进行中。libswarm 项目源码在 <https://github.com/docker/libswarm> 上进行维护，目标是打造管理 Docker 网络的最小化工具集。

libswarm 的主要目标为定义分布式系统各个组件之间进行通信的统一接口，以实现多种平台上 Docker 容器操作的统一性，隐藏下层不同的实现接口。它包括许多的子服务项目，包括 Docker server、Docker client、SSH 隧道、Etcd 配置管理、SkyDNS 等。

使用 libswarm 可以实现，在同一个控制端同时查看、管理运行在多个不同主机和平台上的 Docker 镜像、容器；引入更强大的服务发现和集群系统等。

安装 libswarm 可以通过如下步骤。首先，安装 go 语言环境。访问 <https://golang.org/dl/>，下载源码包，解压后安装，并更新系统环境中的路径变量。

```
$ export GOPATH=/usr/local/go
$ export PATH=$PATH:$GOPATH/bin
```

之后，使用 Go 下载源码：

```
$ go get github.com/docker/libswarm
```

安装 bzr：

```
$ sudo apt-get install bzip2 -y
```

编辑并安装：

```
$ cd $GOPATH/src/github.com/docker/libswarm
$ make deps
$ go install github.com/docker/libswarm/swarmd
```

安装成功后，可以查看 swarmd 的用法：

```
$ swarmd -h
NAME:
    swarmd - Compose distributed systems from lightweight services

USAGE:
    swarmd [global options] command [command options] [arguments...]

VERSION:
    0.0.1

COMMANDS:
    help, h  Shows a list of commands or help for one command

GLOBAL OPTIONS:
    --version, -v      print the version
    --help, -h         show help
```

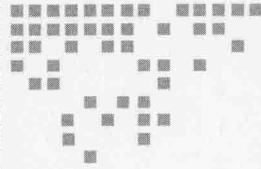
## 20.9 本章小结

本章具体讲解了使用 Docker 网络的一些高级部署和操作配置，包括配置启动参数、DNS、容器的访问控制管理等。并介绍了 Docker 网络相关的一些工具和项目。

网络是一个复杂的领域，特别在云计算领域，因为网络配置造成的管理成本，以及因为网络原因造成的业务损失，都占到十分可观的比例。这是因为网络领域所涉及的学科和技术门类众多，包括软件、硬件、系统，等等。而且往往要求用户对于各种技术的细节把握得十分精确。

从目前来看，Docker 网络所能提供的功能还十分简单，并且基本上都是依赖于 Linux 操作系统上的现有技术。这在初期可以让 Docker 不必考虑太多的网络问题，可以关注自身的特点得以快速发展。但随着 Docker 应用在各种分布式环境、特别是云平台上，网络方面的需求和瓶颈将会大量出现，而且不少都是新的问题。

如何结合已有的各种网络虚拟化技术来解决 Docker 网络的问题，将是未来一段时间内云计算方向中值得持续探讨的重点技术话题。



## 第 21 章

## Docker 相关项目

Docker 虽然属于新兴技术，但围绕它已经出现了不少优秀的技术项目，包括利用 Docker 进行云计算平台搭建，特别是实现平台即服务，利用 Docker 来实现高效的持续集成服务，以及对大规模 Docker 容器的管理和进行编程开发等。

本章将介绍这方面的一些典型项目，包括 Deis、Flynn、Drone、Citadel、Shipyard、Kubernetes、Panamax、Fig 等。

## 21.1 平台即服务方案

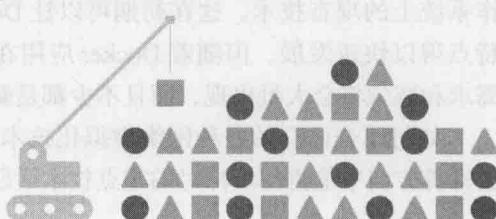
PaaS (Platform as a Service，平台即服务) 是希望提供一个统一的操作系统平台环境，让所有软件直接运行在它上面，而无需复杂配置。Docker 天生的应用封装，为实现 PaaS 提供了某种便利。这里介绍几个基于 Docker 相关技术的 PaaS 项目。

### 1. Deis

项目官方网站为 <http://deis.io>，代码在 <https://github.com/deis/deis> 维护。

Deis 是开源的 PaaS 项目，基于 Go 语言实现，遵循 Apache 2.0 协议。由 OpDemand 公司在 2013 年 7 月发起，目前还处于开发阶段。OpDemand 公司提供对 Deis 的商业服务支持。

Deis 试图提供轻量级的 PaaS 实现，为用户提供简单的应用管理和部署。



Deis 基于 Docker 项目和 CoreOS 项目，并遵循了 SaaS ( Software-as-a-Service，软件即应用) 应用的“十二因素”风格。它通过简单的 git push 命令来部署应用，加速集成和部署过程；它还支持对应用容器通过单条命令进行扩展。



**注意** 十二因素是指一套 SaaS 应用所遵循的风格，包括对代码、依赖、配置、后端服务、生命周期、进程、端口、并发、可丢弃性、开发与生产差异性、日志、管理等十二个方面的规定。

在架构设计上，Deis 整合了一系列 Docker 容器，可以被部署到公有云、私有云，以及本地环境中，并提供了完整的测试、诊断的工具。

## 2. Flynn

项目官方网站为 <http://flynn.io>，代码在 <https://github.com/flynn/flynn> 维护。

Flynn 项目由一个创业团队在 2013 年 7 月发起，基于 Go 语言，目前还处于 beta 阶段。Flynn 已经在 Selfstarter 创业募集网站上筹集到了近十万美元的赞助。

Flynn 项目的发起也是由于一些部署实践问题：在部署 SOA ( Service Oriented Architecture，是大规模分布式系统常采用的架构风格，需要功能组件之间的松耦合) 产品至公有云的过程中，往往需要人工部署和维护大量不同功能部件。

Flynn 基于 Heroku 项目。它受到 Omege 概念（来自剑桥大学、加州伯克利大学和 Google 公司合作的《Omega: flexible, scalable schedulers for large compute clusters》论文）的启发，Flynn 不仅能完成简单可控的部署，还能进行自由的扩展，并提供数据库管理等功能。Flynn 可以方便地实现一套比较理想的 PaaS 方案。



**注意** Heroku 是一个支持多种编程语言（包括 Ruby、Java、Node.js、Scala、Clojure、Python 以及 PHP 和 Perl 等）的云平台即服务实现。在 2010 年被 Salesforce.com 收购。

在设计上，Flynn 项目尽量保持 API 驱动和模块化，以便模块支持不同的实现方案。底层 (layer 0) 实现一套支持服务发现的资源管理框架，上层 (layer 1) 实现适合部署和维护的应用组件。

目前，Flynn 项目已经获得了 Shopify 等公司的支持。

## 21.2 持续集成

目前，Drone 项目利用 Docker 技术，实现持续集成 (Continuous Integration) 平台服务。

### Drone

项目官方网站为 <http://drone.io>, 代码在 <https://github.com/drone/drone> 维护。

Drone 是开源的开源持续集成平台项目，基于 Go 语言实现，遵循 Apache 2.0 协议。该项目最初由 Drone 公司在 2014 年 2 月发起，目前还处于开发阶段。Drone 公司基于它，提供支持 Github、Bitbucket 和 Google Code 等第三方代码托管平台的持续集成服务。

Drone 基于 Docker 和 AUFS 实现，为用户提供基于网站的操作。

用户登录网站后，可以选择源码的存放服务，如图 21-1 所示。



图 21-1 选择服务器

此处选择 Github 服务，然后从仓库列表中选择项目，如图 21-2 所示。



图 21-2 选择项目

并配置项目的语言种类，如图 21-3 所示。



图 21-3 选择语言

最后是检查创建命令是否正确，并根据需要进行调整，如图 21-4 所示。

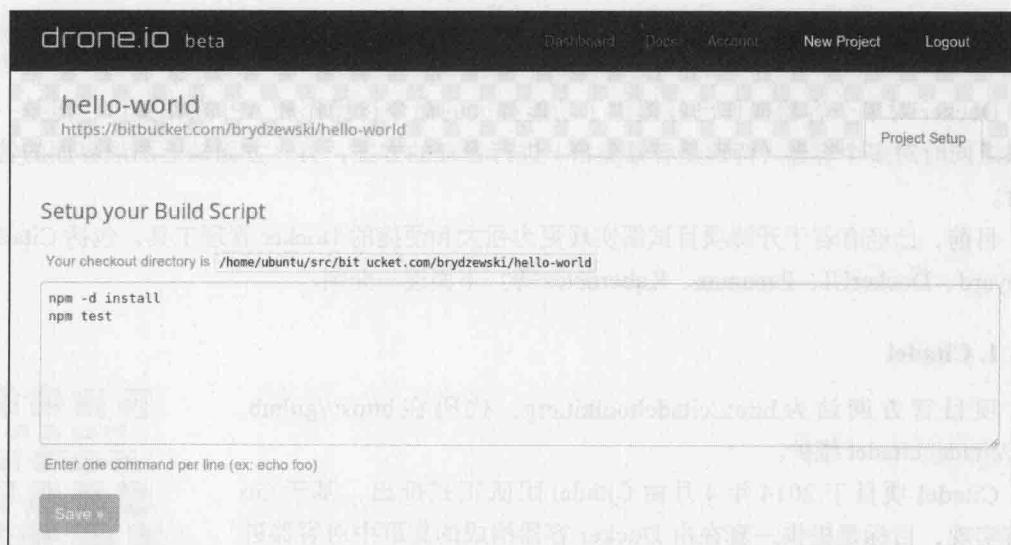


图 21-4 检查创建命令

最后，项目就可以在 Drone 平台上进行持续集成管理了，如图 21-5 所示。

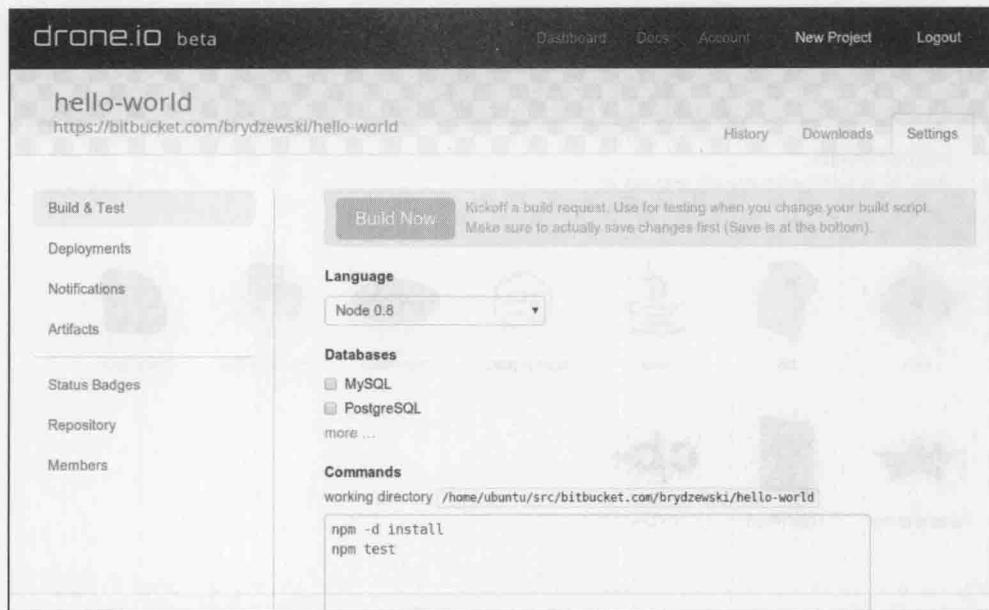


图 21-5 Drone 平台

### 21.3 管理工具

Docker 对单个容器操作已经提供了功能强大的命令行操作和 API 操作接口，但是一方面缺乏同时对多个容器（特别是容器集群）进行管理的方案，另一方面缺乏图形界面的管理平台。

目前，已经有若干开源项目试图实现更为强大和便捷的 Docker 管理工具，包括 Citadel、Shipyard、DockerUI、Panamax、Kubernetes 等。下面逐一介绍。

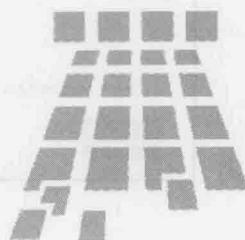
#### 1. Citadel

项目官方网站为 <http://citadeltoolkit.org>，代码在 <https://github.com/citadel/citadel> 维护。

Citadel 项目于 2014 年 4 月由 Citadel 团队正式推出，基于 Go 语言实现，目标是提供一套在由 Docker 容器构成的集群中对容器进行调度的工具，主要包括集群管理组件和调度组件。

集群管理组件（Cluster Manager）负责管理集群的状态，通过调用 Docker 提供的 API 来连接到主机，管理容器。

调度组件（Scheduler）决策如何进行调度，支持多套调度方法，包括基于标签、基于策略等。



否同一镜像、基于主机、组合方法等多种调度机制。

用户使用 Citadel 首先要为调度组件提供容器类型，并指定调度所关心的资源限制；此后，调度器会根据容器类型、服务将容器启动到合适的主机上去。

## 2. Shipyard

项目官方网站为 <http://shipyard-project.com/>，代码在 <https://github.com/shipyard/shipyard> 维护。

Shipyard 项目于 2013 年 11 月发起，它目前基于 Citadel 项目（部分开发者来自同一团队），希望提供一套对 Docker 集群中资源进行管理的工具，包括对 Docker 容器、主机等资源的管理。它最大的特点是在核心部件之外还支持扩展镜像，可以根据需求灵活实现应用负载均衡、集中日志管理和自动化部署等功能。

此外，Shipyard 还提供了方便用户的 Web 界面，功能更加强大的命令行操作接口，以及统一的 API，如图 21-6 所示。

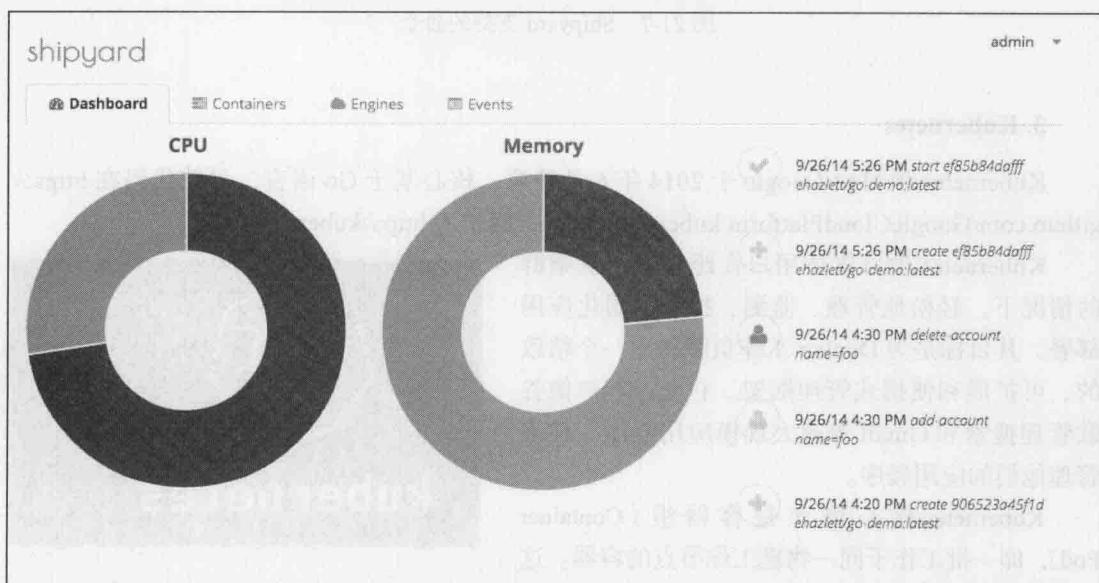


图 21-6 Shipyard 工具

Shipyard 支持的命令包括 login、run、stop、rstart、info 等，如图 21-7 所示。

```

shipyard cli> shipyard
NAME:
  shipyard - manage a shipyard cluster

USAGE:
  shipyard [global options] command [command options] [arguments...]

VERSION:
  2.0.0

COMMANDS:
  login           login to a shipyard cluster
  change-password update your password
  accounts        show accounts
  add-account     add account
  delete-account  delete account
  containers      list containers
  inspect         inspect container
  run             run a container
  stop            stop a container
  restart         restart a container
  destroy         destroy a container
  engines          list engines
  add-engine      add shipyard engine
  remove-engine   removes an engine
  inspect-engine  inspect an engine
  service-keys    list service keys
  add-service-key adds a service key
  remove-service-key removes a service key
  extensions      show extensions
  add-extension   add extension
  remove-extension remove an extension
  webhook-keys    list webhook keys
  add-webhook-key adds a webhook key
  remove-webhook-key removes a webhook key
  info            show cluster info
  events          show cluster events
  help, h         Shows a list of commands or help for one command

```

图 21-7 Shipyard 支持的命令

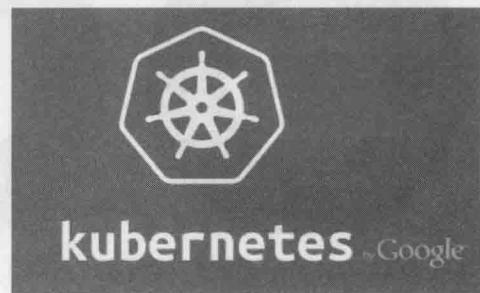
### 3. Kubernetes

Kubernetes 项目由 Google 于 2014 年 6 月开源，核心基于 Go 语言，目前代码在 <https://github.com/GoogleCloudPlatform/kubernetes> 维护，网站为 <http://kubernetes.io/>。

Kubernetes 项目支持用户在跨容器主机集群的情况下，轻松地管理、监测、控制容器化应用部署。其目标是为 Docker 工作负载构建一个精致的、可扩展和便携式管理框架。它允许客户像谷歌管理搜索和 Gmail 等超大规模应用程序一样来管理他们的应用程序。

Kubernetes 核心概念是容器组（Container Pod），即一批工作于同一物理工作节点的容器。这些容器拥有相同的网络命名空间、IP 地址和存储配额，可以根据实际情况对每一个容器组进行端口映射。此外，Kubernetes 有一个与软件定义网络（Software Defined Networking，SDN）非常相似的网络管理概念：通过一个服务代理创建一个可以分配给任意数目容器的 IP 地址，前端的应用程序或使用该服务的用户仅通过这一 IP 地址调用服务，不需要关心其他细节。

目前，已有 Microsoft、RedHat、IBM、Docker、Mesosphere、CoreOS 以及 SaltStack 等



公司加入了 Kubernetes 社区。还有 Flannel 等项目针对 Kubernetes 提供覆盖网络功能。

#### 4. DockerUI

DockerUI 项目目前在 <https://github.com/crosbymichael/dockerui> 维护。

该项目于 2013 年 12 月发起，主要基于 html/js 语言实现，遵循 MIT 许可。用户可以通过下面的命令简单测试该工具：

```
$ sudo docker build -t crosbymichael/dockerui
github.com/crosbymichael/dockerui
$ sudo docker run -d -p 9000:9000 -v /var/run/docker.sock:/docker.sock
crosbymichael/dockerui -e /docker.sock
```

运行成功后，打开浏览器，访问 <http://:9000>。

#### 5. Panamax

项目官方网站为 <http://panamax.io>，代码在 <https://github.com/CenturyLinkLabs/panamax-ui> 维护。

Panamax 项目诞生于 2014 年 3 月，由 CenturyLink 实验室发起（是该实验室的第一个孵化出的开源项目），希望通过一套优雅的界面来实现对复杂的 Docker 容器应用的管理，例如利用简单拖拽来完成操作。Panamax 项目基于 Docker、CoreOS 和 Fleet，可以提供对容器的自动化管理和任务调度，如图 21-8 所示。

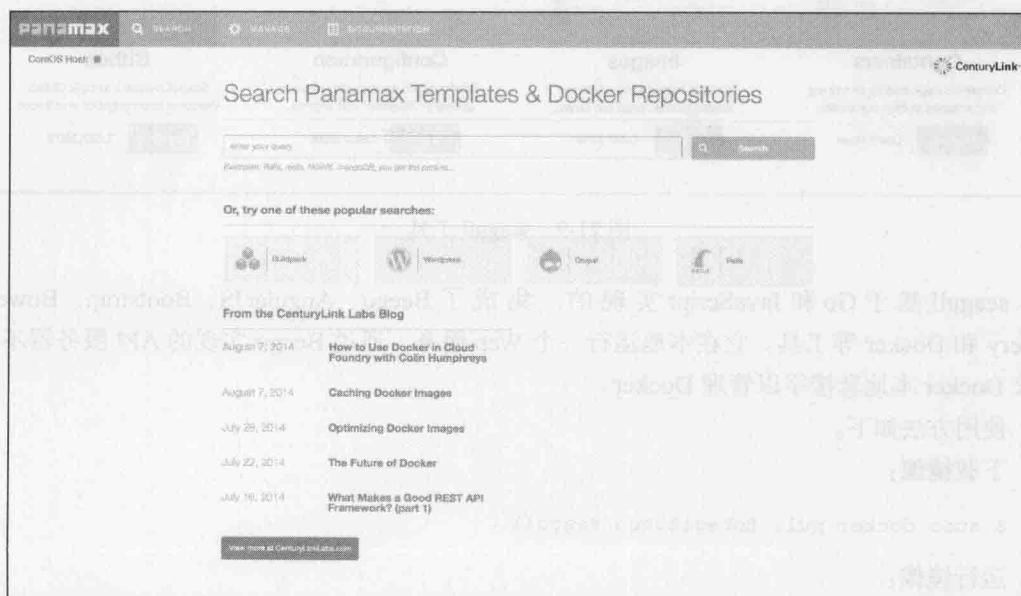


图 21-8 Panamax 工具

Panamax 项目基于 Ruby 语言，遵循 Apache 2 许可，可以部署在 Google、Amazon 等云平台甚至本地环境。

此外，Panamax 还提供了开源应用的模板库，来集中管理不同应用的配置和架构。

## 6. seagull

seagull 是由小米工程师陈迪豪发布的开源 Docker 容器和镜像的 Web 界面监控工具，目前在 <https://github.com/tobegin3hub/seagull> 维护，如图 21-9 所示。

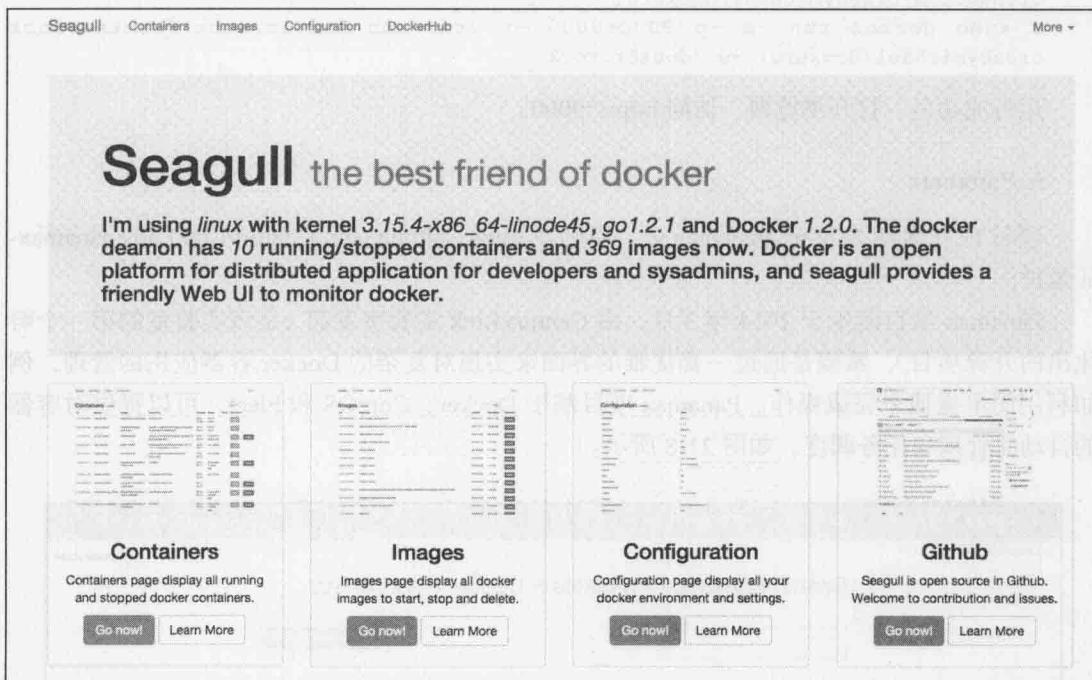


图 21-9 seagull 工具

seagull 基于 Go 和 JavaScript 实现的，集成了 Beego、AngularJS、Bootstrap、Bower、JQuery 和 Docker 等工具。它在本地运行一个 Web 服务，通过 Beego 实现的 API 服务器不断请求 Docker 本地套接字以管理 Docker。

使用方法如下。

下载镜像：

```
$ sudo docker pull tobegit3hub/seagull
```

运行镜像：

```
$ sudo docker run -d -p 10086:10086 -v /var/run/docker.sock:/var/run/docker.
```

```
socktobegin3hub/seagull
```

然后就可以通过浏览器访问地址 <http://127.0.0.1:10086> 登录管理界面。

官方 Dockerfile 如下所示。

```
# Base image is in https://registry.hub.docker.com/_/golang/
# Refer to https://blog.golang.org/docker for usage

FROM golang
MAINTAINER tobe tobeg3oole@gmail.com
ENV REFRESH_AT 20141023

# ENV GOPATH /go

# Install dependency
RUN go get github.com/astaxie/beego
RUN go get github.com/beego/bee
RUN go get github.com/tobegin3hub/seagull

# Go to the folder of seagull
WORKDIR /go/src/github.com/tobegin3hub/seagull/

# Build the project
RUN go build seagull.go

# This should be the same as httpport in conf/app.conf
EXPOSE 10086

# Run the server
CMD ["./seagull"]
```

## 21.4 编程开发

由于 Docker 服务端提供了 REST 风格的 API，通过对这些 API 进一步地封装，可以提供给各种开发语言作为 Docker 的使用库。

这里以 docker-py 项目为例，介绍在 Python 语言中对 Docker 相关资源进行操作。

### 安装 docker-py

```
$ sudo pip install docker-py
```

安装后，可以发现，代码结构十分清晰，主要提供了 Client 类，用来封装提供用户可以用 Docker 命令执行的各种操作，包括 build、run、commit、create\_container、info 等接口。

对 REST 接口的调用是使用了 request 库。对于这些 API，用户也可以通过 curl 来进行调用测试。

## 使用示例

打开 Python 的终端，首先创建一个 Docker 客户端连接：

```
$ sudo python
>>> import docker
>>> c = docker.Client(base_url='unix://var/run/docker.sock',version='1.15',timeout=10)
```

通过 info() 方法查看 Docker 系统信息：

```
>>> c.info()
{'KernelVersion': '3.13.0-24-generic', 'NFd': 19, 'MemoryLimit': 1,
 'InitShal': '', 'SwapLimit': 0, 'Driver': 'aufs', 'IndexServerAddress':
 'https://index.docker.io/v1/', 'NGoroutines': 27, 'Images': 200, 'InitPath':
 '/usr/bin/docker', 'Containers': 2, 'ExecutionDriver': 'native-0.2',
 'Debug': 0, 'NEventsListener': 0, 'DriverStatus': [[{"Root Dir": "/var/lib/
 docker/aufs"}, {"Dirs": "204"}]], 'OperatingSystem': 'Ubuntu 14.04.1 LTS',
 'IPv4Forwarding': 1}
```

通过 images() 和 containers() 方法可以查看本地的镜像和容器的列表：

```
>>> c.images()
[{"Created": 1414108439, "VirtualSize": 199257566, "ParentId": "22093
c35d77bb609b9257ffb2640845ec05018e3d96cb939f68d0e19127f1723", "RepoTags":
["ubuntu:latest"], "Id": "5506de2b643be1e6febbf3b8a240760c6843244c41e12aa2f6
0ccb7153d17f5", "Size": 0}]
>>> c.containers()
[{"Status": "Up 5 seconds", "Created": 1415086513, "Image": "ubuntu:latest",
 "Ports": [], "Command": "/bin/bash", "Names": ["romantic_blackwell"], "Id": "f51f2e4cc6df8829baa00018fe3a9e5112cc4d0786cc674d
621e51ab795c9fd6"}]
```

通过 create\_container() 方法来创建一个容器，之后启动它：

```
>>> container = c.create_container(image='ubuntu:latest', command='bash')
>>> print(container)
{'Id': 'a8439e4c8e64a94a287d408fdc3ff9a0b4a8577fe3b5e32975b790afb414af',
 'Warnings': None}
>>> c.start(container='a8439e4c8e64a94a287d408fdc3ff9a0b4a8577fe3b5e32975b790af
b414af')
```

可见，所提供的方法跟 Docker 提供的命令都十分类似。实际上，在执行 Docker 命令的时候，也是通过 Docker 提供的客户端进行了封装，并向服务端发起 API 请求。

## 21.5 其他项目

### 1. CoreOS

项目官方网站为 <https://coreos.com/>，代码在 <https://github.com/coreos> 维护。

CoreOS 项目基于 Python 语言，遵循 Apache 2.0 许可，由 CoreOS 团队在 2013 年 7 月发起，目前已经正式发布首个稳定版本，如图 21-10 所示。

CoreOS 项目目标是提供一个基于 Docker 的轻量级容器化 Linux 发行版，通过轻量的系统架构和灵活的应用部署能力来简化数据中心的维护成本和复杂度。

CoreOS 基于一套精简的 Linux 环境，不使用包管理工具，而将所有应用都进行容器化，彼此隔离，从而提高了系统的安全性。此外，运行期间，系统分区是只读状态，利用主从分区支持更稳定的无缝升级。配合 etcd（一套分布式高可用的键值数据库）、fleet（CoreOS 集群的管理工具）等工具，CoreOS 也将适用于在大规模集群环境中进行使用。

该项目目前得到了 KPCB 等多家基金的投资。

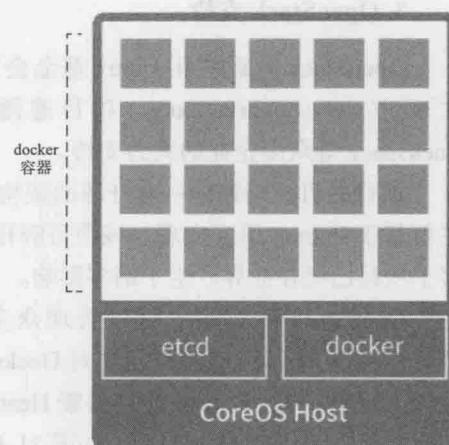


图 21-10 CoreOS 项目

## 2. Fig

项目官方网站为 <http://www.fig.sh>，代码在 <https://github.com/docker/fig> 维护。

Fig 项目基于 Python 语言，由 Docker 公司在 2013 年 12 月发起，目前已经正式发布了 1.0 版本。

用户在使用 Docker 的过程中往往会碰到同时部署多个容器并将它们联合的需求，比如部署一个 web 服务容器和 db 服务容器，同时需要将它们连接起来。当容器数量较多之后，通过手动写脚本来管理已经变得很不方便了。

Fig 项目正是要解决这个问题，它在同一个配置文件中可以定义依赖的镜像和容器之间的连接关系，通过一条简单的命令完成部署。

例如，下面的 Fig 配置文件 (YAML 格式) 给出了同时使用 Web 服务容器和 db 服务容器的一个例子，十分简单易懂。

```
web:
  build: .
  command: python app.py
  links:
    - db
  ports:
    - "8000:8000"
db:
  image: postgres
```



写好配置文件后，用户可以通过执行 figup 命令来自动启动相应容器，完成部署。

### 3. OpenStack 支持

OpenStack 是近些年 Linux 基金会发起的，最受欢迎的云开源项目。项目的官方网站在 <http://www.openstack.org>。项目遵循 Apache 许可，受到包括 IBM、Cisco、AT&T、HP、Rackspace 等众多企业的大力支持。

项目的目标是搭建一套开源的架构即服务（Infrastructure as a Service，IaaS）实现方案，主要基于 Python 语言实现。该项目孵化出来的众多子项目已经在业界产生了诸多影响。

OpenStack 目前除了可以管理众多虚机外，其计算服务（Nova）已经支持了对 Docker 的驱动，此外，还支持通过 Stack 管理引擎 Heat 子项目来使用模板来管理 Docker 容器，如图 21-11 所示。

例如，下面的 Heat 模板定义了使用 Docker 容器运行一个 cirros 镜像：

```
heat_template_version: 2013-05-23

description: Single compute instance running cirros in a Docker container.

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: ewindisch_key
      image: ubuntu-precise
      flavor: m1.large
      user_data: #include https://get.docker.io
  my_docker_container:
    type: DockerInc::Docker::Container
    docker_endpoint: { get_attr: [my_instance, first_address] }
    image: cirros
```

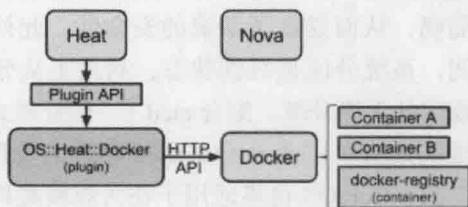


图 21-11 OpenStack 支持 Docker

### 4. dockerize

一般来说，要将一个应用放到容器里，需要考虑两方面的因素，一是应用依赖的配置信息；另外一个是应用运行时候的输出日志信息。dockerize 是一个 Go 程序，试图简化这两方面的管理成本，目前代码在 <https://github.com/jwilder/dockerize> 维护。

它主要可以提供两个功能，一是对于依赖于配置文件的应用，能自动提取环境变量并生成配置文件；另外一个是将应用输出的日志信息重定向到 STDOUT 和 STDERR。

下面给出一个简单的例子，比如要创建一个 Nginx 镜像，标准的 Dockerfile 内容为：

```

FROM ubuntu:14.04

# 安装 Nginx
RUN echo "deb http://ppa.launchpad.net/nginx/stable/ubuntu trusty main" > /etc/
apt/sources.list.d/nginx-stable-trusty.list
RUN echo "deb-src http://ppa.launchpad.net/nginx/stable/ubuntu trusty main" >>
/etc/apt/sources.list.d/nginx-stable-trusty.list
RUN apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C300EE8C
RUN apt-get update
RUN apt-get install -y nginx

RUN echo "daemon off;" >> /etc/nginx/nginx.conf

EXPOSE 80

CMD nginx

```

使用 dockerize，则需要在最后的 CMD 命令中利用 dockerize 进行封装，利用模板生成应用配置文件，并重定向日志文件输出到标准输出。

首先，创建配置模板文件为 default.tmpl，内容是：

```

server{
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    root /usr/share/nginx/html;
    index index.html index.htm;

    # Make site accessible from http://localhost/
    server_name localhost;

    location / {
        access_log off;
        proxy_pass {{ .Env.PROXY_URL }};
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}

```

该模板将接收来自环境变量 PROXY\_URL 的值。

编辑新的 Dockerfile 内容为：

```

FROM ubuntu:14.04

# 安装 Nginx
RUN echo "deb http://ppa.launchpad.net/nginx/stable/ubuntu trusty main" > /etc/
apt/sources.list.d/nginx-stable-trusty.list
RUN echo "deb-src http://ppa.launchpad.net/nginx/stable/ubuntu trusty main" >>
/etc/apt/sources.list.d/nginx-stable-trusty.list

```

```

RUN apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C300EE8C
RUN apt-get update
RUN apt-get install -y wget nginx

RUN echo "daemon off;" >> /etc/nginx/nginx.conf

RUN wget https://github.com/jwilder/dockerize/releases/download/v0.0.1/
dockerize-linux-amd64-v0.0.1.tar.gz
RUN tar -C /usr/local/bin -xvzf dockerize-linux-amd64-v0.0.1.tar.gz

ADD default.tmpl /etc/nginx/sites-available/default.tmpl

EXPOSE 80

CMD dockerize -template /etc/nginx/sites-available/default.tmpl:/etc/nginx/
sites-available/default -stdout /var/log/nginx/access.log -stderr /var/log/
nginx/error.log nginx

```

最后的 CMD 命令中利用 `-template` 参数指定了配置模板位置，以及生成的配置文件的位置。

创建镜像后，通过如下的方式启动一个容器，整个过程无需手动添加 Nginx 的配置文件，并且日志重定向到了标准输出：

```
$ sudo docker run -p 80:80 -e PROXY_URL="http://jasonwilder.com" --name nginx
-d nginx
```

## 5. libcontainer、libchan、libswarm

这三个项目都是 2014 年 6 月 Docker 团队在 DockerCon 大会上正式宣布的项目，目前还处于快速发展阶段。

`libcontainer` 项目的目标是实现容器技术的统一 API。2013 年 Linux 内核 3.12 的推出，引入了专门为容器技术考虑的一套 API，已有各种容器技术可以使用这套内核 API。但各种容器技术可以有自己的实现，这就造成其中的应用需要关心是运行在哪种容器和平台上。`libcontainer` 项目则试图为应用提供统一的一套 API，让它们无需关心具体的容器实现。`libcontainer` 现在已经整合了 Parallels 公司所支持的 `libct` 项目 (<https://github.com/xemul/libct>)，并得到了包括 Red Hat、Google、Canonical、Parallels 等多家公司的支持。Docker 也计划逐步将依赖 LXC 的底层实现迁移到 `libcontainer` 上。该项目将为容器技术的发展带来更大的可能性，比如实现多种平台（包括 Windows 平台）之间容器的迁移。

`libchan` 和 `libswarm` 项目都是为了 Docker 容器集群服务的。

`libchan` 项目是试图在不同的网络服务之间打造一套“channel”系统。`channel` 是 Go 语言里的一个概念，为实现并发情况下不同 goroutine 之间交互数据。`libchan` 所实现的功能类似于已有的各种消息系统，但它宣称将打造适合现代各种微型并发服务的极轻量级（ultra-lightweight）的网络服务库，并且支持 In-memory Go channel、Unix socket、Raw TCP、TLS、

HTTP2/SPDY、Websocket 等多种协议。

libswarm 项目基于 libchan 项目，目标是一套组件网络服务的最小工具集。它定义了在分布式系统中各种服务组件之间互相通信的标准接口，并且试图实现一整套的网络服务组件。其目标组件包括 Docker Server、Docker Client、SSH tunnel、Etcd、SkyDNS、Mesos、OpenStack Nova、Google Compute 等能想到的各种服务（大部分还没有实现）。使用它，用户可以在同一机器上同时管理运行在不同主机上的 Docker 容器，任意的替换集群中的各种服务，包括服务发现、DNS。libswarm 实际上是试图实现一套基础系统，其他各种集群服务作为一个功能模块与 libswarm 对接。这是一个很宏达的目标，但是可能会引发与已有自己基础系统实现的项目的冲突，例如 kubernetes 等。

libcontainer 项目网址为：<https://github.com/docker/libcontainer>。

libchan 项目网址为：<https://github.com/docker/libchan>。

libswarm 项目网址为：<https://github.com/docker/libswarm>。

除了这些项目外，还有一些项目专注于 Docker 周边的功能，例如 Weave (<https://github.com/zettio/weave>) 创建一个虚拟网络来连接部署在多台主机上的 Docker 容器；Flannel 为 Kubernetes 提供覆盖网络支持；SocketPlane 试图将软件定义网络技术引入容器管理等。由于篇幅所限，笔者不在此一一介绍，感兴趣的读者可以自行查阅相关资料。

## 21.6 本章小结

本章介绍了围绕 Docker 生态环境的一些热门技术项目，包括云平台构建、持续集成、容器管理和编程开发等方向。

一项新兴技术能否成功，技术自身的设计、实现固然重要，但围绕技术的生态环境和经济体系往往更为关键。

笔者很欣喜地看到，Docker 无疑已经得到了大量的认同和支持。

基于 Docker 的平台即服务和持续集成这两大方面，是笔者认为 Docker 技术的所谓“杀手级应用”(killing apps)。这些项目充分结合了 Docker 技术的特点，能够充分地发挥出使用 Docker 的技术优势。

在具体的生产环境中使用 Docker，则无法绕开容器管理和编程开发这两种需求。特别是大规模的容器管理，将是一个颇有挑战的难题。不断出现的各种方案，特别是有众多 IT 巨头支持的 Kubernetes 将在一定程度上缓解这个问题的难度，但仍不能说解决了这个挑战。此外，众多编程开发上的技术支持，也将加速 Docker 应用的大量产生。

最后，包括 CoreOS、Fig 等特色项目的出现，以及 OpenStack 这类项目对 Docker 快速支持，都证明了在某种意义上 Docker 在站稳脚跟之后，已经开始引导整个技术体系的变革，这毫无疑问是整个信息技术产业发展的大好事！



## 附录 Appendix

- 附录 A 常见问题汇总
- 附录 B 常见仓库
- 附录 C Docker 命令查询
- 附录 D Docker 资源链接

## 附录 A 常见问题汇总

### A.1 镜像相关

#### 1. 如何批量清理临时镜像文件？

答：可以使用 `sudo docker rmi $(sudo docker images -q -f dangling =true)` 命令。

#### 2. 如何查看镜像支持的环境变量？

答：可以使用 `sudo docker run IMAGE env` 命令。

#### 3. 本地的镜像文件都存放在哪里？

答：与 Docker 相关的本地资源都存放在 `/var/lib/docker/` 目录下，其中 `container` 目录存放容器信息，`graph` 目录存放镜像信息，`aufs` 目录下存放具体的镜像层文件。

#### 4. 构建 Docker 镜像应该遵循哪些原则？

答：整体原则上，尽量保持镜像功能的明确和内容的精简，要点包括：

- 尽量选取满足需求但较小的基础系统镜像，例如大部分时候可以选择 `debian:wheezy` 镜像，仅有 85 MB 大小。

- 清理编译生成文件、安装包的缓存等临时文件。
- 安装各个软件时候要指定准确的版本号，并避免引入不需要的依赖。
- 从安全角度考虑，应用要尽量使用系统的库和依赖。
- 如果安装应用时候需要配置一些特殊的环境变量，在安装后要还原不需要保持的变量值。
- 使用 Dockerfile 创建镜像时候要添加 .dockerignore 文件或使用干净的工作目录。

## A.2 容器相关

### 1. 容器退出后，通过 docker ps 命令查看不到，数据会丢失么？

答：容器退出后会处于终止（exited）状态，此时可以通过 docker ps -a 查看。其中的数据也不会丢失，还可以通过 docker start 命令来启动它。只有删除掉容器才会清除所有数据。

### 2. 如何停止所有正在运行的容器？

答：可以使用 sudo docker kill \$(sudo docker ps -q) 命令。

### 3. 如何清理批量后台停止的容器？

答：可以使用 sudo docker rm \$(sudo docker ps -a -q) 命令。

### 4. 如何给容器指定一个固定 IP 地址，而不是每次重启容器 IP 地址都会变？

答：参考本书第 21 章中介绍的 pipework 工具。

### 5. 如何临时退出一个正在交互的容器的终端，而不终止它？

答：按 Ctrl-p Ctrl-q。如果按 Ctrl-c 往往会让容器内应用进程终止，进而会终止容器。

### 6. 很多应用容器都是默认后台运行的，怎么查看它们的输出和日志信息？

答：使用 docker logs 命令，后面跟容器名称或 ID 信息。

### 7. 使用“docker port”命令映射容器的端口时，系统报错 Error: No public port '80' published for e7d817698b6f，是什么意思？

答：

- 创建镜像时 Dockerfile 要指定正确的 EXPOSE 的端口。

□ 容器启动时指定 `PublisAllPort=true`。

### 8. 可以在一个容器中同时运行多个应用进程吗？

答：一般并不推荐在同一个容器内运行多个应用进程。如果有类似需求，可以通过一些额外的进程管理机制，比如 `supervisord` 来管理所运行的进程。可以参考 [https://docs.docker.com/articles/using\\_supervisord/](https://docs.docker.com/articles/using_supervisord/)。

### 9. 如何控制容器占用系统资源（CPU、内存）的份额？

答：在使用 `docker create` 命令创建容器或使用 `docker run` 创建并启动容器的时候，可以使用 `-c|--cpu-shares[=0]` 参数来调整容器使用 CPU 的权重；使用 `-m|--memory[=MEMORY]` 参数来调整容器使用内存的大小。

## A.3 仓库相关

### 1. 仓库（Repository）、注册服务器（Registry）、注册索引（Index）有何关系？

答：首先，仓库是存放一组关联镜像的集合，比如同一个应用的不同版本的镜像。注册服务器是存放实际的镜像文件的地方。注册索引则负责维护用户的账号、权限、搜索、标签等的管理。因此，注册服务器利用注册索引来实现认证等管理。

### 2. 从非官方仓库（例如 `dl.dockerpool.com`）下载镜像时候，有时候会提示“`Error: Invalid registry endpoint https://dl.dockerpool.com:5000/v1/.....`”？

答：Docker 自 1.3.0 版本往后，加强了对镜像安全性的验证，需要手动添加对非官方仓库的信任。

编辑 Docker 配置文件，在其中添加：

```
DOCKER_OPTS="--insecure-registry dl.dockerpool.com:5000"
```

之后，重启 Docker 服务即可。

## A.4 配置相关

### 1. Docker 的配置文件放在哪里，如何修改配置？

答：Ubuntu 系统的配置文件是 `/etc/default/docker`，Centos 系统的配置文件放在 `/etc/sysconfig/docker`。Ubuntu 下面的配置文件内容如下，读者可以参考配。（如果出

现该文件不存在的情况，重启或者自己新建一个文件都可以解决。)

```
# Docker Upstart and SysVinit configuration file

# Customize location of Docker binary (especially for development testing).
DOCKER="/usr/local/bin/docker"

# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"

# If you need Docker to use an HTTP proxy, it can also be specified here.
export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary files go.
export TMPDIR="/mnt/bigdrive/docker-tmp"
```

## 2. 如何更改 Docker 的默认存储位置？

答：Docker 的默认存储位置是 /var/lib/docker，如果希望将 Docker 的本地文件存储到其他分区，可以使用 Linux 软连接的方式来完成。

例如，如下操作将默认存储位置迁移到 /storage/docker：

```
[root@s26 ~]# df -h
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup-lv_root   50G  5.3G  42G  12% /
tmpfs                 48G  228K  48G   1% /dev/shm
/dev/sdal              485M   40M  420M   9% /boot
/dev/mapper/VolGroup-lv_home  222G 188M  210G   1% /home
/dev/sdb2                2.7T 323G  2.3T  13% /storage

[root@s26 ~]# service docker stop
[root@s26 ~]# cd /var/lib/
[root@s26 lib]# mv docker /storage/
[root@s26 lib]# ln -s /storage/docker/ docker
[root@s26 lib]# ls -la docker
lrwxrwxrwx. 1 root root 15 11月 17 13:43 docker -> /storage/docker
[root@s26 lib]# service docker start
```

## A.5 Docker 与虚拟化

### 1. Docker 与 LXC (Linux Container) 有何不同？

答：LXC 利用 Linux 上相关技术实现了容器。Docker 则在如下的几个方面进行了改进：

- 移植性：通过抽象容器配置，容器可以实现从一个平台移植到另一个平台；
- 镜像系统：基于 AUFS 的镜像系统为容器的分发带来了很多的便利，同时共同的镜像

层只需要存储一份，实现高效率的存储；

- 版本管理：类似于 Git 的版本管理理念，用户可以更方便的创建、管理镜像文件；
- 仓库系统：仓库系统大大降低了镜像的分发和管理的成本；
- 周边工具：各种现有工具（配置管理、云平台）对 Docker 的支持，以及基于 Docker 的 PaaS、CI 等系统，让 Docker 的应用更加方便和多样化。

## 2. Docker 与 Vagrant 有何不同？

答：两者的定位完全不同。

- Vagrant 类似于 Boot2Docker（一款运行 Docker 的最小内核），是一套虚拟机的管理环境。Vagrant 可以在多种系统上和虚拟机软件中运行，可以在 Windows、Mac 等非 Linux 平台上为 Docker 提供支持，自身具有较好的包装性和移植性。
- 原生的 Docker 自身只能运行在 Linux 平台上，但启动和运行的性能都比虚拟机要快，往往更适合快速开发和部署应用的场景。

简单说：Vagrant 适合用来管理虚拟机，而 Docker 适合用来管理应用环境。

## 3. 开发环境中 Docker 和 Vagrant 该如何选择？

答：Docker 不是虚拟机，而是进程隔离，对于资源的消耗很少，但是目前需要 Linux 环境支持。Vagrant 是虚拟机上做的封装，虚拟机本身会消耗资源。

如果本地使用的 Linux 环境，推荐都使用 Docker。

如果本地使用的是 OSX 或者 Windows 环境，那就需要开虚拟机，单一开发环境下 vagrant 更简单；多环境开发下推荐在 vagrant 里面再使用 Docker 进行环境隔离。

## A.6 其他问题

### 1. Docker 能在非 Linux 平台（比如 Windows 或 MacOS）上运行吗？

答：可以，但需要使用 boot2docker 等软件创建一个轻量级的 Linux 虚拟机层。

### 2. 如何将一台宿主主机的 docker 环境迁移到另外一台宿主主机？

答：停止 Docker 服务。将整个 docker 存储文件夹复制到另外一台宿主主机，然后调整另外一台宿主主机的配置即可。

### 3. 创建 Docker 容器后，在宿主机用“ip netns show”为何看不到容器的网络名字空间？

答：Docker 在创建容器后，删除了 /var/run/netns 目录中的网络名字空间文件。因此，可

以手动恢复它。

首先，使用下面的命令查看容器进程信息，比如这里的 1234。

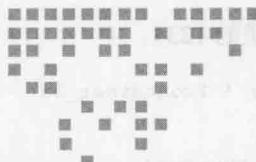
```
$ sudo docker inspect --format='{{. State.Pid}}' $container_id  
1234
```

接下来，在 /proc 目录下，把对应的网络名字空间文件链接到 /var/run/netns 目录。

```
$ sudo ln -s /proc/1234/ns/net /var/run/netns/
```

然后，就可以通过正常的系统命令来查看或操作容器的名字空间了。例如

```
$ sudo ip netns show  
1234
```



## Appendix B

# 常见仓库

本章将介绍常见的仓库和镜像的功能、使用方法和生成它们的 Dockerfile 等，包括 Ubuntu、CentOS、MySQL、MongoDB、Redis、Nginx、Wordpress、Node.js 等。

## B.1 Ubuntu

### 基本信息

Ubuntu 是流行的 Linux 发行版，其自带软件版本往往较新一些。该仓库提供了 Ubuntu 从 12.04~14.10 各个版本的镜像。

### 使用方法

默认会启动一个最小化的 Ubuntu 环境。

```
$ sudo docker run --name some-ubuntu -i -t ubuntu  
root@523c70904d54:/#
```

### Dockerfile

#### 12.04 版本

```
FROM scratch  
ADD precise-core-amd64.tar.gz /
```

```

# a few minor docker-specific tweaks
# see https://github.com/dotcloud/docker/blob/master/contrib/mkimage/
debootstrap
RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
    && echo 'exit 101' >> /usr/sbin/policy-rc.d \
    && chmod +x /usr/sbin/policy-rc.d \
    \
    && dpkg-divert --local --rename --add /sbin/initctl \
    && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
    && sed -i 's/^exit.*$/exit 0/' /sbin/initctl \
    \
    && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \
    \
    && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/
cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' > /etc/apt/
apt.conf.d/docker-clean \
    \
    && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/
cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' >> /etc/apt/
apt.conf.d/docker-clean \
    \
    && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgs cache ""';' >> /etc/apt/
apt.conf.d/docker-clean \
    \
    && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/docker-no-
languages \
    \
    && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTypes::Order::
"gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes

# delete all the apt list files since they're big and get stale quickly
RUN rm -rf /var/lib/apt/lists/*
# this forces "apt-get update" in dependent images, which is also good

# enable the universe
RUN sed -i 's/^#\s*\(\deb.*universe\)\$/\1/g' /etc/apt/sources.list

# upgrade packages for now, since the tarballs aren't updated frequently enough
RUN apt-get update && apt-get dist-upgrade -y && rm -rf /var/lib/apt/lists/*

# overwrite this with 'CMD []' in a dependent Dockerfile
CMD ["/bin/bash"]

```

## 14.04 版本

```

FROM scratch
ADD trusty-core-amd64.tar.gz /

# a few minor docker-specific tweaks
# see https://github.com/dotcloud/docker/blob/master/contrib/mkimage/
debootstrap
RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
    && echo 'exit 101' >> /usr/sbin/policy-rc.d \
    && chmod +x /usr/sbin/policy-rc.d \

```

```

  \
  && dpkg-divert --local --rename --add /sbin/initctl \
  && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
  && sed -i 's/^exit.*/exit 0/' /sbin/initctl \
  \
  && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \
  \
  && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/
cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' > /etc/apt/
apt.conf.d/docker-clean \
  && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /
var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' >> /
etc/apt/apt.conf.d/docker-clean \
  && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache "";' >> /etc/apt/
apt.conf.d/docker-clean \
  \
  && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/docker-no-
languages \
  \
  && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTypes::Order::
"gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes

# delete all the apt list files since they're big and get stale quickly
RUN rm -rf /var/lib/apt/lists/*
# this forces "apt-get update" in dependent images, which is also good

# enable the universe
RUN sed -i 's/^#\s*\(\deb.*universe\)$/\1/g' /etc/apt/sources.list

# upgrade packages for now, since the tarballs aren't updated frequently enough
RUN apt-get update && apt-get dist-upgrade -y && rm -rf /var/lib/apt/lists/*

# overwrite this with 'CMD []' in a dependent Dockerfile
CMD ["/bin/bash"]

```

## 14.10 版本

```

FROM scratch
ADD utopic-core-amd64.tar.gz /

# a few minor docker-specific tweaks
# see https://github.com/dotcloud/docker/blob/master/contrib/mkimage/
debootstrap
RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
  && echo 'exit 101' >> /usr/sbin/policy-rc.d \
  && chmod +x /usr/sbin/policy-rc.d \
  \
  && dpkg-divert --local --rename --add /sbin/initctl \
  && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
  && sed -i 's/^exit.*/exit 0/' /sbin/initctl \
  \
  && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \

```

```

  \\
  && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/
cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' > /etc/apt/
apt.conf.d/docker-clean \
  && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /
var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' >> /
etc/apt/apt.conf.d/docker-clean \
  && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgsocache "";' >> /etc/apt/
apt.conf.d/docker-clean \
  \\
  && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/docker-no-
languages \
  \\
  && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTypes::Order::
"gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes

# delete all the apt list files since they're big and get stale quickly
RUN rm -rf /var/lib/apt/lists/*
# this forces "apt-get update" in dependent images, which is also good

# enable the universe
RUN sed -i 's/^#\s*\(\deb.*universe\)\$/\1/g' /etc/apt/sources.list

# upgrade packages for now, since the tarballs aren't updated frequently enough
RUN apt-get update && apt-get dist-upgrade -y && rm -rf /var/lib/apt/lists/*

# overwrite this with 'CMD []' in a dependent Dockerfile
CMD ["/bin/bash"]

```

## B.2 CentOS

### 基本信息

CentOS 是流行的 Linux 发行版，其软件包大多跟 RedHat 系列保持一致。该仓库提供了 CentOS 从 5 ~ 7 各个版本的镜像。

### 使用方法

默认会启动一个最小化的 CentOS 环境：

```
$ sudo docker run --name some-centos -i -t centos bash
bash-4.2#
```

### Dockerfile

#### CentOS 5 版本

```
FROM scratch
```

```
MAINTAINER The CentOS Project <cloud-ops@centos.org> - ami_creator
ADD centOS-5-20140926_1219-docker.tar.xz /
```

### CentOS 6 版本

```
FROM scratch
MAINTAINER The CentOS Project <cloud-ops@centos.org> - ami_creator
ADD CentOS-6-20140926_1219-docker.tar.xz /
```

### CentOS 7 版本

```
FROM scratch
MAINTAINER The CentOS Project <cloud-ops@centos.org> - ami_creator
ADD CentOS-7-20140926_1219-docker.tar.xz /
```

## B.3 MySQL

### 基本信息

MySQL 是开源的关系数据库实现。该仓库提供了 MySQL 各个版本的镜像，包括 5.6 系列、5.7 系列等。

### 使用方法

默认会在 3306 端口启动数据库：

```
$ sudo docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
```

之后就可以使用其他应用来连接到该容器：

```
$ sudo docker run --name some-app --link some-mysql:mysql -d application-that-uses-mysql
```

或者通过 mysql：

```
$ sudo docker run -it --link some-mysql:mysql --rm mysql sh -c 'exec mysql -h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT" -uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
```

### Dockerfile

#### 5.6 版本

```
FROM debian:wheezy
```

```
# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
```

```

RUN groupadd -r mysql && useradd -r -g mysql mysql
# FATAL ERROR: please install the following Perl modules before executing /usr/local/mysql/scripts/mysql_install_db:
# File::Basename
# File::Copy
# Sys::Hostname
# Data::Dumper
RUN apt-get update && apt-get install -y perl --no-install-recommends && rm -rf /var/lib/apt/lists/*
# mysqld: error while loading shared libraries: libaio.so.1: cannot open shared object file: No such file or directory
RUN apt-get update && apt-get install -y libaio1 && rm -rf /var/lib/apt/lists/*
# gpg: key 5072E1F5: public key "MySQL Release Engineering <mysql-build@oss.oracle.com>" imported
RUN gpg --keyserver pgp.mit.edu --recv-keys A4A9406876FCBD3C456770C88C718D3B5072E1F5
ENV MYSQL_MAJOR 5.6
ENV MYSQL_VERSION 5.6.20

# note: we're pulling the *.asc file from mysql.he.net instead of dev.mysql.com because the official mirror 404s that file for whatever reason - maybe it's at a different path?
RUN apt-get update && apt-get install -y curl --no-install-recommends && rm -rf /var/lib/apt/lists/*
  && curl -SL "http://dev.mysql.com/get/Downloads/MySQL-$MYSQL_MAJOR/mysql-$MYSQL_VERSION-linux-glibc2.5-x86_64.tar.gz" -o mysql.tar.gz \
  && curl -SL "http://mysql.he.net/Downloads/MySQL-$MYSQL_MAJOR/mysql-$MYSQL_VERSION-linux-glibc2.5-x86_64.tar.gz.asc" -o mysql.tar.gz.asc \
  && apt-get purge -y --auto-remove curl \
  && gpg --verify mysql.tar.gz.asc \
  && mkdir /usr/local/mysql \
  && tar -xzf mysql.tar.gz -C /usr/local/mysql --strip-components=1 \
  && rm mysql.tar.gz* \
  && rm -rf /usr/local/mysql/mysql-test /usr/local/mysql/sql-bench \
  && rm -rf /usr/local/mysql/bin/*-debug /usr/local/mysql/bin/*_embedded \
  && find /usr/local/mysql -type f -name "*.a" -delete \
  && apt-get update && apt-get install -y binutils && rm -rf /var/lib/apt/lists/* \
  && { find /usr/local/mysql -type f -executable -exec strip --strip-all '{}' + || true; } \
  && apt-get purge -y --auto-remove binutils
ENV PATH $PATH:/usr/local/mysql/bin:/usr/local/mysql/scripts

WORKDIR /usr/local/mysql
VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh

```

```

ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 3306
CMD ["mysqld", "--datadir=/var/lib/mysql", "--user=mysql"]

```

## 5.7 版本

```

FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
RUN groupadd -r mysql && useradd -r -g mysql mysql

# FATAL ERROR: please install the following Perl modules before executing /usr/
# local/mysql/scripts/mysql_install_db:
# File::Basename
# File::Copy
# Sys::Hostname
# Data::Dumper
RUN apt-get update && apt-get install -y perl --no-install-recommends && rm -rf
/var/lib/apt/lists/*

# mysqld: error while loading shared libraries: libaio.so.1: cannot open shared
# object file: No such file or directory
RUN apt-get update && apt-get install -y libaio1 && rm -rf /var/lib/apt/lists/*

# gpg: key 5072E1F5: public key "MySQL Release Engineering <mysql-build@oss.
# oracle.com>" imported
RUN gpg --keyserver pgp.mit.edu --recv-keys A4A9406876FCBD3C456770C88C718D3B5072E1F5

ENV MYSQL_MAJOR 5.7
ENV MYSQL_VERSION 5.7.4-m14

# note: we're pulling the *.asc file from mysql.he.net instead of dev.mysql.com
# because the official mirror 404s that file for whatever reason - maybe it's at a
# different path?
RUN apt-get update && apt-get install -y curl --no-install-recommends && rm -rf
/var/lib/apt/lists/* \
  && curl -SL "http://dev.mysql.com/get/Downloads/MySQL-$MYSQL_MAJOR/mysql-
$MYSQL_VERSION-linux-glibc2.5-x86_64.tar.gz" -o mysql.tar.gz \
  && curl -SL "http://mysql.he.net/Downloads/MySQL-$MYSQL_MAJOR/mysql-$MYSQL_
VERSION-linux-glibc2.5-x86_64.tar.gz.asc" -o mysql.tar.gz.asc \
  && apt-get purge -y --auto-remove curl \
  && gpg --verify mysql.tar.gz.asc \
  && mkdir /usr/local/mysql \
  && tar -xzf mysql.tar.gz -C /usr/local/mysql --strip-components=1 \
  && rm mysql.tar.gz* \
  && rm -rf /usr/local/mysql/mysql-test /usr/local/mysql/sql-bench \
  && rm -rf /usr/local/mysql/bin/*-debug /usr/local/mysql/bin/*_embedded \
  && find /usr/local/mysql -type f -name "*.a" -delete \
  && apt-get update && apt-get install -y binutils && rm -rf /var/lib/apt/
lists/* \

```

```

    && { find /usr/local/mysql -type f -exec strip --strip-all '{}' \;
+ || true; } \
    && apt-get purge -y --auto-remove binutils
ENV PATH $PATH:/usr/local/mysql/bin:/usr/local/mysql/scripts
WORKDIR /usr/local/mysql
VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 3306
CMD ["mysqld", "--datadir=/var/lib/mysql", "--user=mysql"]

```

## B.4 MongoDB

### 基本信息

MongoDB 是开源的 NoSQL 数据库实现。该仓库提供了 MongoDB2.2~2.7 各个版本的镜像。

### 使用方法

默认会在 27017 端口启动数据库：

```
$ sudo docker run --name some-mongo -d mongo
```

使用其他应用连接到容器，可以用：

```
$ sudo docker run --name some-app --link some-mongo:mongo -d application-that-uses-mongo
```

或者通过 mongo：

```
$ sudo docker run -it --link some-mongo:mongo --rm mongo sh -c 'exec mongo "$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT/test"'
```

### Dockerfile

#### 2.2 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb
```

```

RUN apt-get update \
&& apt-get install -y curl \
&& rm -rf /var/lib/apt/lists/*
RUN curl -o /usr/local/bin/gosu -SL 'https://github.com/tianon/gosu/releases/
download/1.1/gosu' \
&& chmod +x /usr/local/bin/gosu

RUN gpg --keyserver pgp.mit.edu --recv-keys 3ADEF01FE92B6927CC1EEC80F56417
9A36496327

ENV MONGO_VERSION 2.2.7

RUN curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz" -o mongo.tgz \
&& curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz.sig" -o mongo.tgz.sig \
&& gpg --verify mongo.tgz.sig \
&& tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
&& rm mongo.tgz*

VOLUME /data/db

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]

```

## 2.4 版本

```

FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
&& apt-get install -y curl \
&& rm -rf /var/lib/apt/lists/*

RUN curl -o /usr/local/bin/gosu -SL 'https://github.com/tianon/gosu/releases/
download/1.1/gosu' \
&& chmod +x /usr/local/bin/gosu

RUN gpg --keyserver pgp.mit.edu --recv-keys CEA1E18DDA77EF4E67884FF2A6982D01604
56C5A

ENV MONGO_VERSION 2.4.11

RUN curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz" -o mongo.tgz \

```

```

    && curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz.sig" -o mongo.tgz.sig \
    && gpg --verify mongo.tgz.sig \
    && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
    && rm mongo.tgz*
VOLUME /data/db

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]

```

## 2.6 版本

```

FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*

RUN curl -o /usr/local/bin/gosu -SL 'https://github.com/tianon/gosu/releases/
download/1.1/gosu' \
    && chmod +x /usr/local/bin/gosu

RUN gpg --keyserver pgp.mit.edu --recv-keys DFFA3DCF326E302C4787673A01C4E7FAAAB
2461C

ENV MONGO_VERSION 2.6.4

RUN curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz" -o mongo.tgz \
    && curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz.sig" -o mongo.tgz.sig \
    && gpg --verify mongo.tgz.sig \
    && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
    && rm mongo.tgz*

VOLUME /data/db

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]

```

## 2.7 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*

RUN curl -o /usr/local/bin/gosu -SL 'https://github.com/tianon/gosu/releases/
download/1.1/gosu' \
    && chmod +x /usr/local/bin/gosu

RUN gpg --keyserver pgp.mit.edu --recv-keys BDC0DB28022D7DEA1490DC3E7085801C857
FD301

ENV MONGO_VERSION 2.7.6

RUN curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz" -o mongo.tgz \
    && curl -SL "https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-$MONGO_
VERSION.tgz.sig" -o mongo.tgz.sig \
    && gpg --verify mongo.tgz.sig \
    && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
    && rm mongo.tgz*

VOLUME /data/db

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]
```

## B.5 Redis

### 基本信息

Redis 是开源的内存键值数据库实现，该仓库提供了 Redis2.6~2.8.9 各个版本的镜像。

### 使用方法

默认会在 6379 端口启动数据库：

```
$ sudo docker run --name some-redis -d redis
```

另外还可以启用持久存储:

```
$ sudo docker run --name some-redis -d redis redis-server --appendonly yes
```

默认数据存储位置在 VOLUME/data。可以使用 --volumes-from some-volume-container 或 -v/docker/host/dir:/data 将数据存放到本地。

使用其他应用连接到容器，可以用:

```
$ sudo docker run --name some-app --link some-redis:redis -d application-that-uses-redis
```

或者通过 redis-cli:

```
$ sudo docker run -it --link some-redis:redis --rm redis sh -c 'exec redis-cli -h "$REDIS_PORT_6379_TCP_ADDR" -p "$REDIS_PORT_6379_TCP_PORT"'
```

## Dockerfile

### 2.6 版本

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
RUN groupadd -r redis && useradd -r -g redis redis

ENV REDIS_VERSION 2.6.17
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-2.6.17.tar.gz
ENV REDIS_DOWNLOAD_SHA1 b5423e1c423d502074cbd0b21bd4e820409d2003

RUN buildDeps='gcc libc6-dev make'; \
    set -x; \
    apt-get update && apt-get install -y $buildDeps curl --no-install-recommends \
    && rm -rf /var/lib/apt/lists/* \
    && mkdir -p /usr/src/redis \
    && curl -sSL "$REDIS_DOWNLOAD_URL" -o redis.tar.gz \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | shasum -c - \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apt-get purge -y $buildDeps curl \
    && apt-get autoremove -y

RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data

USER redis
```

```

EXPOSE 6379
CMD [ "redis-server" ]

最新 2.8 版本
FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned consistently,
# regardless of whatever dependencies get added
RUN groupadd -r redis && useradd -r -g redis redis

ENV REDIS_VERSION 2.8.13
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-2.8.13.tar.gz
ENV REDIS_DOWNLOAD_SHA1 a72925a35849eb2d38a1ea076a3db82072d4ee43

RUN buildDeps='gcc libc6-dev make'; \
    set -x; \
    apt-get update && apt-get install -y $buildDeps curl --no-install-recommends \
    && rm -rf /var/lib/apt/lists/* \
    && mkdir -p /usr/src/redis \
    && curl -ssl "$REDIS_DOWNLOAD_URL" -o redis.tar.gz \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | shasum -c - \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apt-get purge -y $buildDeps curl \
    && apt-get autoremove -y

RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data

USER redis
EXPOSE 6379
CMD [ "redis-server" ]

```

## B.6 Nginx

### 基本信息

Nginx 是开源的高效的 Web 服务器实现，支持 HTTP、HTTPS、SMTP、POP3、IMAP 等协议。该仓库提供了 Nginx 1.0~1.7 各个版本的镜像。

### 使用方法

下面的命令将作为一个静态页面服务器启动：

```
$ sudo docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro
-d nginx
```

用户也可以不使用这种映射方式，通过利用 Dockerfile 来直接将静态页面内容放到镜像中，内容为：

```
FROM nginx
COPY static-html-directory /usr/share/nginx/html
```

之后生成新的镜像，并启动一个容器：

```
$ sudo docker build -t some-content-nginx .
$ sudo docker run --name some-nginx -d some-content-nginx
```

开放端口，并映射到本地的 8080 端口：

```
sudo docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

Nginx 的默认配置文件路径为 /etc/nginx/nginx.conf，可以通过映射它来使用本地的配置文件，例如：

```
docker run --name some-nginx -v /some/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx
```

使用配置文件时，为了在容器中正常运行，需要保持 daemon off;。

## Dockerfile

### 1 ~ 1.7 版本

```
FROM debian:wheezy

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

RUN apt-key adv --keyserver pgp.mit.edu --recv-keys 573BFD6B3D8FBC641079A6ABAF
5BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/ wheezy nginx" >> /etc/
apt/sources.list

ENV NGINX_VERSION 1.7.5-1~wheezy

RUN apt-get update && apt-get install -y nginx=${NGINX_VERSION}

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

# be backwards compatible with pre-official images
RUN ln -sf ../share/nginx /usr/local/nginx

VOLUME ["/usr/share/nginx/html"]
VOLUME ["/etc/nginx"]
```

```
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

## B.7 WordPress

### 基本信息

WordPress 是开源的 Blog 和内容管理系统框架，它基于 PHP 和 MySQL。该仓库提供了 WordPress 4.0 版本的镜像。

### 使用方法

启动容器需要 MySQL 的支持，默认端口为 80：

```
$ sudo docker run --name some-wordpress --link some-mysql:mysql -d wordpress
```

启动 WordPress 容器时可以指定的一些环境参数包括：

- -e WORDPRESS\_DB\_USER=... 默认为“root”
- -e WORDPRESS\_DB\_PASSWORD=... 默认为连接 mysql 容器的环境变量 MYSQL\_ROOT\_PASSWORD 的值
- -e WORDPRESS\_DB\_NAME=... 默认为“wordpress”
- -e WORDPRESS\_AUTH\_KEY=..., -e WORDPRESS\_SECURE\_AUTH\_KEY=..., -e WORDPRESS\_LOGGED\_IN\_KEY=..., -e WORDPRESS\_NONCE\_KEY=..., -e WORDPRESS\_AUTH\_SALT=..., -e WORDPRESS\_SECURE\_AUTH\_SALT=..., -e WORDPRESS\_LOGGED\_IN\_SALT=..., -e WORDPRESS\_NONCE\_SALT=... 默认为随机 sha1 串

### Dockerfile

#### 4.0 版本

```
FROM debian:wheezy
RUN apt-get update && apt-get install -y \
    apache2 \
    curl \
    libapache2-mod-php5 \
    php5-curl \
    php5-gd \
    php5-mysql \
    rsync \
    wget \
```

```

    && rm -rf /var/lib/apt/lists/*
RUN a2enmod rewrite

# copy a few things from apache's init script that it requires to be setup
ENV APACHE_CONFDIR /etc/apache2
ENV APACHE_ENVVARS $APACHE_CONFDIR/envvars
# and then a few more from $APACHE_CONFDIR/envvars itself
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_PID_FILE $APACHE_RUN_DIR/apache2.pid
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_LOG_DIR /var/log/apache2
ENV LANG C
RUN mkdir -p $APACHE_RUN_DIR $APACHE_LOCK_DIR $APACHE_LOG_DIR

# make CustomLog (access log) go to stdout instead of files
# and ErrorLog to stderr
RUN find "$APACHE_CONFDIR" -type f -exec sed -ri ' \
  s|^(\s*CustomLog)\s+\$+!\`1 /proc/self/fd/1!g; \
  s|^(\s*ErrorLog)\s+\$+!\`1 /proc/self/fd/2!g; \
  ' '{}' ;'

RUN rm -rf /var/www/html && mkdir /var/www/html
VOLUME /var/www/html
WORKDIR /var/www/html

ENV WORDPRESS_VERSION 4.0.0
ENV WORDPRESS_UPSTREAM_VERSION 4.0

# upstream tarballs include ./wordpress/ so this gives us /usr/src/wordpress
RUN curl -SL http://wordpress.org/wordpress-${WORDPRESS_UPSTREAM_VERSION}.tar.gz | tar -xzC /usr/src/
COPY docker-apache.conf /etc/apache2/sites-available/wordpress
RUN a2dissite 000-default && a2ensite wordpress

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
EXPOSE 80
CMD ["apache2", "-DFOREGROUND"]

```

## B.8 Node.js

### 基本信息

Node.js 是基于 JavaScript 的可扩展服务端和网络软件开发平台。该仓库提供了 Node.

js0.8~0.11 各个版本的镜像。

## 使用方法

在项目中创建一个 Dockerfile:

```
FROM node:0.10-onbuild
# replace this with your application's default port
EXPOSE 8888
```

然后创建镜像，并启动容器：

```
$ sudo docker build -t my-nodejs-app
$ sudo docker run -it --rm --name my-running-app my-nodejs-app
```

也可以直接运行一个简单容器：

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp
-w /usr/src/myapp node:0.10 node your-daemon-or-script.js
```

## Dockerfile

### 0.8 版本

```
FROM buildpack-deps

RUN apt-get update && apt-get install -y \
    ca-certificates \
    curl

# verify gpg and sha256: http://nodejs.org/dist/v0.10.30/SHASUMS256.txt.asc
# gpg: aka "Timothy J Fontaine (Work) <tj.fontaine@joyent.com>"
RUN gpg --keyserver pgp.mit.edu --recv-keys 7937DFD2AB06298B2293C3187D33FF9D0246406D

ENV NODE_VERSION 0.8.28

RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-
linux-x64.tar.gz" \
    && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
    && gpg --verify SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256.txt.asc | \
sha256sum -c - \
    && tar -xzf "node-v$NODE_VERSION-linux-x64.tar.gz" -C /usr/local --strip-
components=1 \
    && rm "node-v$NODE_VERSION-linux-x64.tar.gz" SHASUMS256.txt.asc

CMD [ "node" ]
```

### 0.10 版本

```
FROM buildpack-deps
```

```

RUN apt-get update && apt-get install -y \
    ca-certificates \
    curl

# verify gpg and sha256: http://nodejs.org/dist/v0.10.31/SHASUMS256.txt.asc
# gpg: aka "Timothy J Fontaine (Work) <tj.fontaine@joyent.com>"
RUN gpg --keyserver pgp.mit.edu --recv-keys 7937DFD2AB06298B2293C3187D33FF9D0246406D

ENV NODE_VERSION 0.10.32

RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-
linux-x64.tar.gz" \
    && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
    && gpg --verify SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256.txt.asc | \
sha256sum -c - \
    && tar -xzf "node-v$NODE_VERSION-linux-x64.tar.gz" -C /usr/local --strip-
components=1 \
    && rm "node-v$NODE_VERSION-linux-x64.tar.gz" SHASUMS256.txt.asc

CMD [ "node" ]

```

## 0.11 版本

```

FROM buildpack-deps

RUN apt-get update && apt-get install -y \
    ca-certificates \
    curl

# verify gpg and sha256: http://nodejs.org/dist/v0.10.30/SHASUMS256.txt.asc
# gpg: aka "Timothy J Fontaine (Work) <tj.fontaine@joyent.com>"
RUN gpg --keyserver pgp.mit.edu --recv-keys 7937DFD2AB06298B2293C3187D33FF9D0246406D

ENV NODE_VERSION 0.11.13

RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-
linux-x64.tar.gz" \
    && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
    && gpg --verify SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256.txt.asc | \
sha256sum -c - \
    && tar -xzf "node-v$NODE_VERSION-linux-x64.tar.gz" -C /usr/local --strip-
components=1 \
    && rm "node-v$NODE_VERSION-linux-x64.tar.gz" SHASUMS256.txt.asc

CMD [ "node" ]

```

## 附录 C

# Docker 命令查询

以 Docker 1.30 版本为例。

## C.1 基本语法

```
docker [OPTIONS] COMMAND [arg...]
```

一般来说，Docker 命令可以用来管理 daemon，或者通过 CLI 命令管理镜像和容器。可以通过 `man docker` 来查看这些命令。

## C.2 选项

`-D=true|false`

使用 debug 模式。默认为 `false`。

`-H, --host=[unix:///var/run/docker.sock], tcp://[host:port]`

在 daemon 模式下绑定的 socket，通过一个或多个 `tcp://host:port`, `unix:///path/to/socket`, `fd:///*` 或 `fd://socketfd` 来指定。

`--api-enable-cors=true|false`

在远端 API 中启用 CORS 头。默认为 `false`。

`-b=""`

将容器挂载到一个已存在的网桥上。指定为 '`none`' 时则禁用容器的网络。

```
--bip=""
    让动态创建的 docker0 采用给定的 CIDR 地址；与 -b 选项互斥。          (1) 0.1.1.100-101.255.255.0

-d=true|false
    使用 daemon 模式。默认为 false。          (2) 0.1.1.100-101.255.255.0

--dns=""
    让 Docker 使用给定的 DNS 服务器。          (3) 0.1.1.100-101.255.255.0

-g=""
    指定 Docker 运行时的 root 路径。默认为 /var/lib/docker。          (4) 0.1.1.100-101.255.255.0

--icc=true|false
    启用容器间通信。默认为 true。          (5) 0.1.1.100-101.255.255.0

--ip=""
    绑定端口时候的默认 IP 地址。默认为 0.0.0.0。          (6) 0.1.1.100-101.255.255.0

--iptables=true|false
    禁止 Docker 添加 iptables 规则。默认为 true。          (7) 0.1.1.100-101.255.255.0

--mtu=VALUE
    指定容器网络的 mtu。默认为 1500。          (8) 0.1.1.100-101.255.255.0

-p=""
    指定 daemon 的 PID 文件路径。默认为 /var/run/docker.pid。          (9) 0.1.1.100-101.255.255.0

--registry-mirror=://
    指定一个注册服务器的镜像地址。          (10) 0.1.1.100-101.255.255.0

-s=""
    强制 Docker 运行时使用给定的存储驱动。          (11) 0.1.1.100-101.255.255.0

-v=true|false
    输出版本信息并退出。默认值为 false。          (12) 0.1.1.100-101.255.255.0

--selinux-enabled=true|false
    启用 SELinux 支持。默认值为 false。SELinux 目前不支持 BTRFS 存储驱动。          (13) 0.1.1.100-101.255.255.0
```

## C.3 命令

Docker 的命令可以采用 docker-CMD 或者 docker CMD 的方式执行，两者一致。

**docker-attach(1)**  
依附到一个正在运行的容器中。

**docker-build(1)**  
从一个 Dockerfile 创建一个镜像。

<b>docker-commit(1)</b>	从一个容器的修改中创建一个新的镜像。
<b>docker-create(1)</b>	创建一个新容器，但并不运行它。
<b>docker-cp(1)</b>	从容器中复制文件到宿主系统中。
<b>docker-diff(1)</b>	检查一个容器文件系统的修改。
<b>docker-events(1)</b>	从服务端获取实时的事件。
<b>docker-exec(1)</b>	在运行的容器内执行命令。
<b>docker-export(1)</b>	导出容器内容为一个 tar 包。
<b>docker-history(1)</b>	显示一个镜像的历史。
<b>docker-images(1)</b>	列出存在的镜像。
<b>docker-import(1)</b>	导入一个文件（典型为 tar 包）路径或目录来创建一个镜像。
<b>docker-info(1)</b>	显示一些相关的系统信息。
<b>docker-inspect(1)</b>	显示一个容器的底层具体信息。
<b>docker-kill(1)</b>	关闭一个运行中的容器（包括进程和所有相关资源）。
<b>docker-load(1)</b>	从一个 tar 包中加载一个镜像。
<b>docker-login(1)</b>	注册或登录到一个 Docker 的仓库服务器。
<b>docker-logout(1)</b>	从 Docker 的仓库服务器登出。
<b>docker-logs(1)</b>	显示一个正在运行的容器的日志。

获取容器的 log 信息。

**docker-pause(1)**

暂停一个容器中的所有进程。

**docker-port(1)**

查找一个 nat 到一个私有网口的公共口。

**docker-ps(1)**

列出容器。

**docker-pull(1)**

从一个 Docker 的仓库服务器下拉一个镜像或仓库。

**docker-push(1)**

将一个镜像或者仓库推送到一个 Docker 的注册服务器。

**docker-restart(1)**

重启一个运行中的容器。

**docker-rm(1)**

删除给定的若干个容器。

**docker-rmi(1)**

删除给定的若干个镜像。

**docker-run(1)**

创建一个新容器，并在其中运行给定命令。

**docker-save(1)**

保存一个镜像为 tar 包文件。

**docker-search(1)**

在 Docker index 中搜索一个镜像。

**docker-start(1)**

启动一个容器。

**docker-stop(1)**

终止一个运行中的容器。

**docker-tag(1)**

为一个镜像打标签。

**docker-top(1)**

查看一个容器中的正在运行的进程信息。

**docker-unpause(1)**

将一个容器内所有的进程从暂停状态中恢复。

`docker-version(1)`  
输出 Docker 的版本信息。

`docker-wait(1)`  
阻塞直到一个容器终止，然后输出它的退出符。

## C.4 一张图总结 Docker 的命令

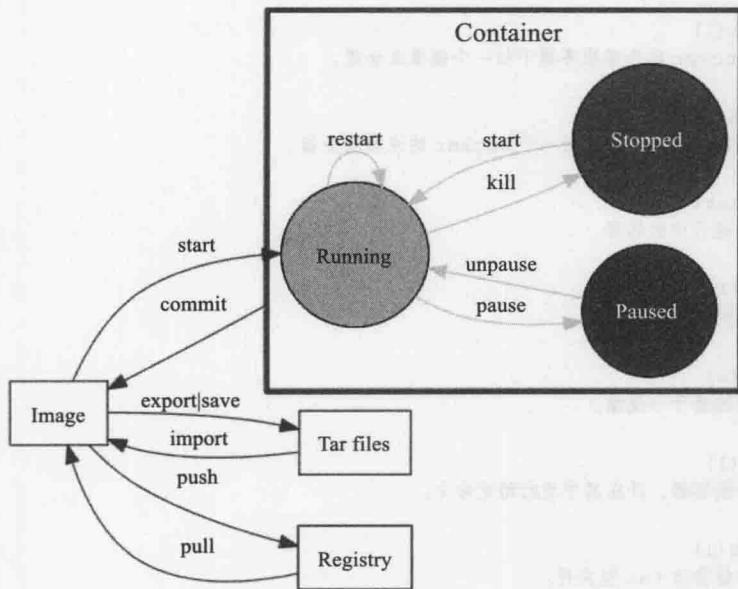


图 C-1 Docker 命令

## Docker 资源链接

Docker 官方主页: <https://www.docker.io>

Docker 注册中心 API: [http://docs.docker.com/reference/api/registry\\_api/](http://docs.docker.com/reference/api/registry_api/)

Docker Hub API: [http://docs.docker.com/reference/api/docker-io\\_api/](http://docs.docker.com/reference/api/docker-io_api/)

Docker 远端应用 API: [http://docs.docker.com/reference/api/docker\\_remote\\_api/](http://docs.docker.com/reference/api/docker_remote_api/)

Dockerfile 参考: <https://docs.docker.com/reference/builder/>

Dockerfile 最佳实践: [https://docs.docker.com/articles/dockerfile\\_best-practices/](https://docs.docker.com/articles/dockerfile_best-practices/)

Docker Hub: <http://hub.docker.com>

Docker 官方博客: <http://blog.docker.com>

Docker 官方文档: <http://docs.docker.com>

Docker 官方入门指南: <http://www.docker.com/tryit/>

Docker 的 Github 源代码: <https://github.com/docker/docker>

Docker Forge (收集了各种 Docker 工具、组件和服务): <https://github.com/dockerforge>

Docker 邮件列表: <https://groups.google.com/forum/#!forum/docker-user>

Docker 的 IRC 频道: <irc.freenode.net>

Docker 的 Twitter 主页: <http://twitter.com/docker>

Docker 的 StackOverflow 问答主页: <http://stackoverflow.com/search?q=docker>



虽然前几年在容器方面所做的工作不多，但是从2015年起，我们计划将工作重点放在 Docker 等容器技术上。

——杨卫华  
新浪微博技术总监

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

本书围绕着镜像、容器、仓库三个部分，从实践的角度出发，讲解了 Docker 的安装、配置、使用的方式。在本书的后面几个章节，也介绍了许多 Docker 的实现细节和工作原理。总体而言，本书从实际的案例入手，由浅至深，循序渐进，内容相当丰富。

——王灿  
浙江大学计算机学院副教授

Docker 在公司多个项目中正式上线，目前运行稳定，在系统的关键节点使用 Docker 容器集群来快速扩展计算能力效果显著。本书作者之一戴王剑前期的充分调研和测试功不可没。

——徐勋业  
浙江中正智能科技有限公司副总裁

本书详细介绍了 Docker 的发展历史、作用、部署方法和应用案例，文笔流畅，通俗易懂，对促进开源软件和虚拟化技术发展很有意义，对加强信息化在各行业的应用有较大的参考价值。

——杨传斌  
浙江师范大学计算机学院教授



上架指导：计算机\云计算

ISBN 978-7-111-48852-1



9 787111 488521 >

定价：59.00元

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：[www.hzbook.com](http://www.hzbook.com)

网上购书：[www.china-pub.com](http://www.china-pub.com)

数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)