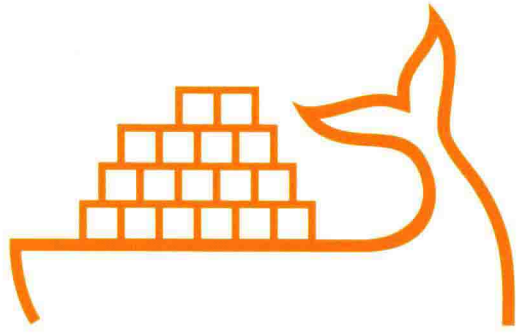
HZ BOOKS
华章IT

容器技术系列



循序渐进学Docker

LEARNING DOCKER STEP BY STEP

李金榜 尹烨 刘天斯 陈纯 著

腾讯4位资深专家大规模应用Docker的
技术心得与经验总结

机械工业出版社
China Machine Press

腾讯是国内较早研究和大规模使用Docker等容器技术的企业之一，为了将腾讯等大企业的Docker实践经验挖掘出来并与业内分享，我们有幸邀请到了腾讯最早开始从事Docker研究和使用的4位核心技术专家，协助他们将这几年在腾讯内部的Docker实践以循序渐进的方式分享了出来。

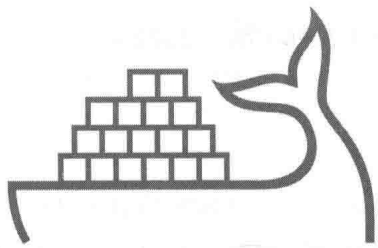
在内容上，本书与已出版的同类书有诸多不同之处，主要体现在以下三个方面：

- 本书来自于腾讯等超大型互联网公司的实践经验总结，尤为难得。
- 内容并不追求面面俱到，而是以企业实际应用中会遇到的常见问题为选材标准，只讲读者最关心的问题，针对性强，同时对这些问题给出了在腾讯被验证的解决方案。
- 没有一味地追求深度，而是以循序渐进的方式组织内容，从Docker的基础知识到高级知识，再到来自于实际生产环境的案例，最后是Docker的源码探索，让不同水平层次的读者都能有效阅读并学为己用。

上架指导：计算机/云计算
ISBN 978-7-111-54854-6
9 787111 548546 >
定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn



循序渐进学 Docker

LEARNING DOCKER STEP BY STEP

李金榜 尹烨 刘天斯 陈纯 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

循序渐进学 Docker / 李金榜等著 . —北京：机械工业出版社，2016.9
（容器技术系列）

ISBN 978-7-111-54854-6

I. 循… II. 李… III. Linux 操作系统 – 程序设计 IV. TP316.85

中国版本图书馆 CIP 数据核字（2016）第 252572 号

循序渐进学 Docker

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：李 艺 责任校对：殷 虹
印 刷：中国电影出版社印刷厂 版 次：2016 年 11 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：14.75
书 号：ISBN 978-7-111-54854-6 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）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

感谢家人一直以来对我工作的理解与支持，还有宝宝刘思彤今年马上要上一年级了，相信你已经准备好了，加油！没有你们背后默默的支持与鼓励，绝对没有今天的自己，谢谢你们！

刘天斯

感谢 SNG 社交网络运营部平台技术运营中心同事们的支持和帮助。

李金榜

感谢家人和同事对我的支持和帮助，感谢一起编写本书的同事，没有你们的鞭策，我无法坚持下来。

尹烨

很荣幸能有机会参与本书的创作。感谢在编写本书过程中家人和同事对我生活的照顾和工作的支持，同时感谢其他三位作者对我参与写书的鼓励和支持。

陈纯

前　言 Preface

为什么要写这本书

Docker 自 2013 年诞生以来，在短短几年就迅速引爆 IT 技术圈，全球各大知名 IT 企业也纷纷加入。Docker 社区的火爆程度也是前所未有，周边的技术案例、平台工具也是层出不穷，其中也不乏一线 IT 公司的身影，比如 Google、微软、Red Hat、VMware 等，放眼国内，基于 Docker 技术的创业公司也如雨后春笋，国内互联网公司的代表 BAT 也开始尝试在企业内部运用落地。在这样的大背景下，大家对掌握及运用 Docker 技术的欲望也越来越强烈。因此，四位笔者走到了一起，开始谋划这本书籍。

笔者都来自腾讯不同事业群及中心，都有针对各自不同应用场景做 Docker 技术研究及应用的实践经验，在研究的过程中，大家也将自己的研究历程、成果做了聚合，最终形成了本书的初稿，包括读者比较关心的 Docker 网络及存储、日常运营到源码探索，循序渐进的内容组织结构，可以让不同水平层次的读者均能有效地阅读和吸收。

本书的初衷是将研究、使用 Docker 过程中可能碰到的问题，以及解决的方法与思路做个自我梳理与总结，同时与大家分享。最终目的是让每位关注 Docker 技术的人受益。

读者对象

- ❑ 系统架构师、运维人员
- ❑ 运营开发、DevOps 人员
- ❑ 云计算工程师
- ❑ 系统管理员或企业网管
- ❑ 高等院校计算机专业的学生与教师

如何阅读本书

本书分为四部分：

第一部分为基础篇，包括第 1 至第 4 章，介绍 Docker 的基础知识及原理，介绍 Docker 是什么，可以做什么，以及如何使用 Docker 技术，包括了安装、创建容器与镜像、运行等。

第二部分为高级篇，包括第 5 至 11 章，着重讲解如何实现容器管理、镜像管理、仓库管理、网络和存储管理及项目日常维护，又补充了最新版本 Docker Swarm 容器集群和 Docker 插件开发等内容。

第三部分为案例篇，包括第 12 至第 15 章，通过对 3 个不同编排技术实现的 Docker 服务案例讲解，让读者了解一个完整的平台的搭建。

第四部分为源码探索篇，为第 16 章，介绍了 Docker 的源码结构和如何修改和编译 Docker，为读者更深入学习研究 Docker 提供一种新思路。

其中第三部分以接近实战的实例来讲解，相比于前两部分更独立。如果你是一名经验丰富的 Linux 管理员且具有 Docker 基础，可以直接切入高级篇；但如果你是一名初学者，请一定从 Docker 的基础理论知识开始学习；如果你对 Docker 的源码分解比较感兴趣，可以直接阅读第 16 章。

勘误和支持

由于水平有限，且编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，特意创建一个在线支持与应急方案问答站点 http://qa.liuts.com。你可以将书中的错误发布到“错误反馈”分类中，同时如果你遇到任何问题或有任何建议，也可以访问问答站点进行发表，我将尽量在线上为读者提供最满意的解答。我也会将相应的功能更新及时更正出来。如果你有更多的宝贵意见，欢迎加入“循序渐进学 Docker”读者 QQ 群（QQ 群账号 559435845 或者扫描以下二维码），期待能够得到你们的真挚反馈。

致谢

首先要感谢 dotCloud 公司，是他们创立了 Docker 这个容器引擎，同时也要感谢为 Docker 整个生态圈贡献大量周边组件的所有作者，是你们让 Docker 技术发展得越来越好，开源的精神与力量在你们身上体现得淋漓尽致。

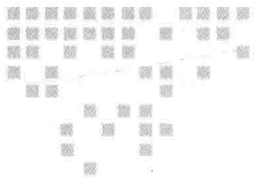
感谢王冬生兄贡献他在工作中的案例（Docker 离线系统应用案例），内容具有非常高的实用价值，感谢公司各位领导及同事，感谢本书的所有作者，在大家的努力下终于促成了这本书的合作与出版。

感谢机械工业出版社华章公司的编辑杨福川、姜影老师，在这一年多的时间中始终支持我的写作，你的鼓励和帮助引导我能顺利完成全部书稿。

Contents 目　录

第一部分 Part 1

基 础 篇



Chapter 1 第 1 章

全面认识 Docker

欢迎来到 Docker 的世界。

Docker，Golang 社区杀手级的应用，是 Github 上最活跃的项目之一，也是开源社区最受欢迎的项目。

Docker，号称要成为所有云应用的基石，并把互联网升级到下一代。

开发、测试、运维人员看到 Docker，都激动地说："太好了，这正是我所需要的！"

Docker 是什么，能解决什么问题，为什么这么火？本章将一一道来。

1.1 Docker 是什么

首先，我们了解下 Docker 产生的历史背景和当前发展情况，通过和一些熟悉的事物做类比，让大家对 Docker 有一个初步认识和了解。

1.1.1 Docker 的由来

Docker 是 dotCloud 公司开源的一款产品。dotCloud 公司是 2010 年新成立的一家公司，主要基于 PaaS（Platform as a Service，平台即服务）平台为开发者提供服务。在 PaaS 平台下，所有的服务环境已经预先配置好了，开发者只需要选择服务类型、上传代码就可对外服务，不需要花费大量的时间搭建服务和配置环境。dotCloud 的 PaaS 平

台已经做得足够好了，它支持几乎所有主流的 Web 编程语言和数据库，可以让开发者随心所欲地选择自己需要的编程语言、数据库和编程框架，而且它的设置非常简单，每次编码后只需要运行一条命令就能把整个网站部署上去；并且利用多层次平台的概念，理论上，它的应用可以运行在各种类型的云服务上。两三年下来，虽然 DotCloud 也在业界获得不错的口碑，但由于整个 PaaS 市场还处于培育阶段，dotCloud 公司表现得不温不火，没有出现爆发性的增长。

2013 年，dotCloud 的 CEO Solomon Hykes 决定把 dotCloud 内部使用的 Container 容器技术单独拿出来开源。2013 年 3 月发布 Docker 的 V0.1 版本，并且基本保持每月一个版本的迭代速度，到了 8 月，Docker 已经足够火爆，并广受好评，各种各样的技术论坛和技术峰会都开始热烈讨论与推荐 Docker，这时 Docker 才只发布到 V0.6 版本。

随着 Docker 的流行，越来越多的优秀开发者加入 Docker 社区参加开发。这里值得一提的是，Docker 是基于 Linux3.8 以上内核，在 aufs 分层文件系统下构建的，主要运行在 Ubuntu 的系统下。REHL/Centos 当时最新版 6 系列还是基于 Liunux2.6.32 内核，无法运行 Docker。为了让 REHL/Centos 尽快支持 Docker，RedHat 公司的工程师亲自出马，加班加点为 Docker 贡献代码，新增对 devicemapper 的支持来实现文件系统分层，终于顺利地让 Docker 在 REHL/Centos 运行起来。

随着 Docker 在业界的知名度越来越高，到了 2013 年 10 月，dotCloud 公司索性更名为 Docker 股份有限公司，工作的重心也从 PaaS 平台业务转向全面围绕 Docker 来开发。到了 2014 年 1 月，Docker 公司宣布完成 15 000 万美元的融资，雅虎联合创始人杨致远也参与跟投。

虽然 Docker 迟迟没有发布 1.0 版，但好多公司已纷纷把 Docker 应用到生产环境。其中，美国奢饰品电商 Gilt 的 CTO 说：“使用 Docker 以后，突然之间，传统方式中的各种问题都消失了，我们接下来要考虑如何进一步提高软件生产效率，让软件开发更加安全和创新。这种转变太不可思议了！”

千呼万唤，到了 2014 年 6 月 9 日，Docker 终于发布了 V1.0 版，并举办了 DockerCon 2014 大会，大会上来自 Google、IBM、RedHat、Rackspace 等公司的核心人物均发表了主题演讲，纷纷表示支持并加入 Docker 的阵营。Docker 的 CTO Solomon Hykes 充满雄心壮志地说：“我们能把互联网升级到下一代！”Google 的基础架构部副总裁 Eric Brewer 也附和道：“容器技术曾是 Google 的基础，我们和 Docker 联手，把容器技术打造为所有云应用的基石。”

Google 自 2004 年就开始使用容器技术，目前他们每周要启动超过 20 亿个容器，每秒钟新启动的容器就超过 3000 个，在容器技术方面有大量的积累。曾相继开源了 Cgroup 和 Imctfy 这两个重量级项目。Google 对 Docker 的支持力度非常大，不仅把 Imctfy 先进之处融入 Docker 中，还把自己的容器管理系统（kubernetes）也开源出来。

2014 年 8 月，不缺钱的 Docker 再次融资，融资超 4 千万美元，估值达到 4 亿美元。

所有的云计算大公司，如 Azure、Google 和亚马逊等都在支持 Docker 技术，这实际上也让 Docker 成为云计算领域的一大重要组成部分。

2014 年 10 月 15 日，Azure 副总裁 Jason Zander 宣布了微软与 Docker 的合作伙伴关系；2014 年 11 月 5 日，Google 发布支持 Docker 的产品 DockerGoogle Container Engine；2014 年 11 月 13 日，Amazon 发布支持 Docker 的产品 AWS Container Service。至此，几个重要的云计算大公司都已经支持 Docker 技术，这不仅让 Docker 成为云计算领域的一个重要级成员，也让 Docker 成为云应用部署的事实上的标准。

2014 年 12 月，Docker 发布了 Docker 集群管理工具 Machine 和 Swarm，标志着 Docker 开始突破一个标准的容器框架，打造属于 Docker 自己的集群平台和生态圈。

2015 年 4 月，Docker 公司宣布完成了 9500 万美元的 D 轮融资。

2015 年 10 月，Docker 收购 Tutum，Tutum 本身已经实现对亚马逊网络服务（AWS）、Digital Ocean、微软的 Azure 等主流云服务商的良好支持。

2016 年 1 月，Docker 官方计划全面支持自身的 Alpine Linux，使用它构建的基础镜像最小只有 5M。

截至 2016 年 3 月，Docker 在 Github 上收获 29 962 个关注（star）、8437 个拷贝（Fork），在 Github 所有项目中排第 7 位，在云平台管理领域排名第一，远远超 Openstack 项目的 1316 个关注、768 个拷贝。

1.1.2 Docker 为什么这么火

Docker 从诞生到现在，短短两年时间，已经成为开源社区最火爆的项目，风头已经远远盖过了近年来很流行的 Puppet 和 OpenStack。那么 Docker 的火爆到底是一种炒作、一种跟风，还是它确实名副其实、众望所归呢？

要回答这个问题，首先看看当前我们所处的环境和面临的问题。

随着计算机近几十年的蓬勃发展，产生了大量优秀系统和软件。比如：

- 操作系统，如 REHL/Centos、Debian/Unbuntu、FreeBSD、OpenSuse 等。
- 编程语言，如 Java、C/C++、Python、Ruby、Golang 等。
- Web 服务器，如 Apache、Nginx、Lighttpd 等。
- 数据库，如 Mysqld、Redis、Mongodb 等。

现在的软件开发人员真是幸运，可以在这么多种类中自由选择。自由选择的结果是，维护一个非常庞大的开发、测试和生产环境，开发、测试和运维人员都被种类繁多的环境折腾得筋疲力尽，不得不收缩战线，每种类型的软件只选择一两种来支持。许多优秀的开发框架和软件尽管有不少优秀特性，但因为维护麻烦，便没有了用武之地。

即便每种类型的软件只选择一两种来支持，随着操作系统和软件版本的更新迭代，维护工作还是变得越来越庞大。

面对这种情况，业界大牛群策群力，给出了很多解决方案，比较有代表的是 Puppet 和 OpenStack。

- Puppet 是集中的配置管理系统，它把文件、用户、cron 任务、软件包、系统服务等抽象为资源，并通过自有的语言描述资源间的依赖关系，集中管理各类资源的安装配置。Puppet 主要适用于需要大批量部署相同服务的应用场景。
- OpenStack 是开源的云计算管理平台项目，可以帮助企业内部实现类似于 Amazon EC2 的云基础架构服务。虽然灵活，但组件繁多、构建复杂，比较适合中大型企业使用。

Puppet 和 OpenStack 虽然比较流行，但适应的场景有限，不具备通用性。正当大家在众多方案中左右为难时，Docker 出现了，它作为一个开源的应用容器引擎，让开发者可以打包他们的应用及依赖环境到一个可移植的容器中，然后发布到任何运行有 Docker 引擎的机器上。它集版本控制、克隆继承、环境隔离等特性于一身，提出一整套软件构建、部署和维护的解决方案，可以非常方便地帮助开发人员，让大家可以随心所欲地使用软件而又不会深陷到环境配置中。

这只是 Docker 的一个应用场景而已，Docker 还能干更多的事情。

作为计算机的从业人员，下面场景你或许碰到过。

- 小 A 是一名资深码农，作为新招聘实习生的导师，小 A 要给实习生的开发机装一套和自己开发机一样的运行环境，不仅要安装 Nginx、Java、Mysqld 和一些依赖库等，还要修改相关的配置文件。结果花了一天时间，小 A 也没把实习生的开发环境搞定，在徒弟面前颜面尽失，尴尬不已。

- 小 B 是一名 QA 测试工程师，他按开发给的文档、部署的服务，测试出一大堆问题，通过和开发的沟通，发现是开发和测试环境不一致引起的。
- 小 C 作为一名业务运维工程师，同时维护开发、测试、生产三套环境，经常在不同环境下装相同的包，做大量重复工作。
- 小 D 同时在为三个项目开发功能模块，他要不停地修改他的开发环境为适应在三个项目间开发、联调测试。
- 小 E 发现服务器被入侵过，他想知道什么文件被篡改过。
- 小 F 从离职同事那里接手一个系统，文档不全，突然一台机器硬件故障，他不知道该如何重新部署这个应用。
- 小 G 新上线一个游戏，游戏火爆超预期，需要紧急扩容，花了一两个小时才完成扩容，期间用户体验很卡，流失不少潜在用户。
- 小 H 和小 I 共同维护一套系统，分工轮流值夜班，但一出现突发故障，排查问题时，即便半夜，还需要把对方叫醒，确认下对方在前一天有没有变更过什么配置。
- 小 M 的一个机房要裁撤了，该机房的数千个应用都要迁移到其他机房，小 M 觉得这项工作非常庞大，半年时间都未必能完成。

但是如果使用 Docker，这些根本不算事儿，分分钟就能搞定。

Docker 的解决方案简单、灵活、高效，还很直观，甚至不需要过多地改变现有的使用习惯，就可以和已有的工具，如 Puppet、OpenStack 等配合使用。各种优势让 Docker 脱颖而出，有鹤立鸡群的感觉，Docker 的火爆也就不难理解了。

1.1.3 Docker 究竟是什么

按照官方的说法，Docker 是一个开源的应用容器引擎。很多人觉得这个说法太抽象，不容易理解。

那我们就从最熟悉的事物说起吧，但凡从事过计算机相关行业的人，对 Java、Android 和 Github 都很熟悉。

先说 Java，在 Java 之前的编程语言，像 C/C++，是严重依赖平台的，在不同平台下，需要重新编译才能运行。Java 的一个非常重要的特性就是与平台无关性，而使用 Java 虚拟机是实现这一特性的关键。Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成可以在 Java 虚拟机上运行的目标代码（字节码），就可以在

多种平台上不加修改地运行。Java 虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。

软件部署也依赖平台，Ubuntu 的软件包在 Centos 下可能就运行不起来。和 Java 虚拟机类似，Docker 使用容器引擎解决平台依赖问题，它在每台宿主机上都启动一个 Docker 的守护进程，守护进程屏蔽了与具体平台相关的信息，对上层应用提供统一的接口。这样，Docker 化的应用，就可以在多个平台下运行，Docker 会针对不同的平台，解析给不同平台下的执行驱动、存储驱动和网络驱动去执行。

Java 曾提出“Write Once，Run Anywhere”，而 Docker 则提出了“Build once，Run anywhere，Configure once，Run anything”。虽然，Java 和 Docker 是为了解决不同领域的问题，但在平台移植方面却面临相同的问题，使用的解决方式也相似。

提起 Android，大家想到什么？它是一个开源的手机操作系统，也是一个生态圈，它的 App 应用以 apk 形式打包、发布，可以运行在任何厂商的 Android 手机上。它还有一个官方的安卓市场，提供各种各样的 App，我们需要某个 App 时，就从安卓市场上搜索下载，手机开发者也可以编写一些 App，发布到安卓市场，提供给别人使用，Android 也允许在第三方的安卓市场上下载或上传应用。

如果把软件部署的应用看作 Android 的 App，Docker 简直和 Android 一模一样，Docker 是一个开源的容器引擎，也有自己的生态圈，它的应用以镜像（image）的形式发布，可以运行在任何装有 Docker 引擎的操作系统上。它有一个官方的镜像仓库，提供各种各样的应用，当需要某个应用时，就从官方的仓库搜索并下载，个人开发者也可以提交镜像到官方仓库，分享给别人使用。Docker 也允许使用第三方的镜像仓库。

最后，再谈谈 Github。它主要用来做版本控制，不仅可以比较两个版本的差异，还可以基于某些历史版本创建新的分支。

使用 Docker 后，软件部署的应用也可以具备类似 Github 的版本控制功能，对应用做一些修改，提交新版本，运行环境可以在多个版本间快速切换，自由选择使用哪个版本对外提供服务。

通过和 Java、Android、Github 的对比，大家对 Docker 应该有了比较直观的认识，Docker 用来管理软件部署的应用，Docker 把应用打包成一个镜像，镜像带有版本控制功能，应用的每次修改迭代就对应镜像的一个版本，制作好的镜像可以发布到镜像仓库，分享给别人；也可以直接从镜像仓库下载别人制作好的应用，不做任何修改，即可运行起来。

1.2 Docker 的结构与特性

通过上一小节的介绍，大家对 Docker 有一个初步的了解。这一节，再来聊一下 Docker 的组织结构。

1.2.1 Docker 构成

如果把 Docker 当作一个独立的软件来看，它就是用 Golang 写的开源程序，采用 C/S 架构，包含 Docker Server 和 Docker Client，源代码托管在 https://github.com/docker/docker 上。

如果把 Docker 看作一个生态的话，它主要由两部分组成：Docker 仓库和 Docker 自身程序。拿 iPhone 做类比的话，Docker 仓库相当于 iPhone 的 Appstore（应用商店），Docker 相当于 iPhone 的 iOS 手机操作系统。

1. Docker 仓库

官方 Docker 仓库地址为 https://hub.docker.com，上面的应用非常丰富，既有各大公司打包的应用，也有大量个人开发者提供的应用，如图 1-1 所示。

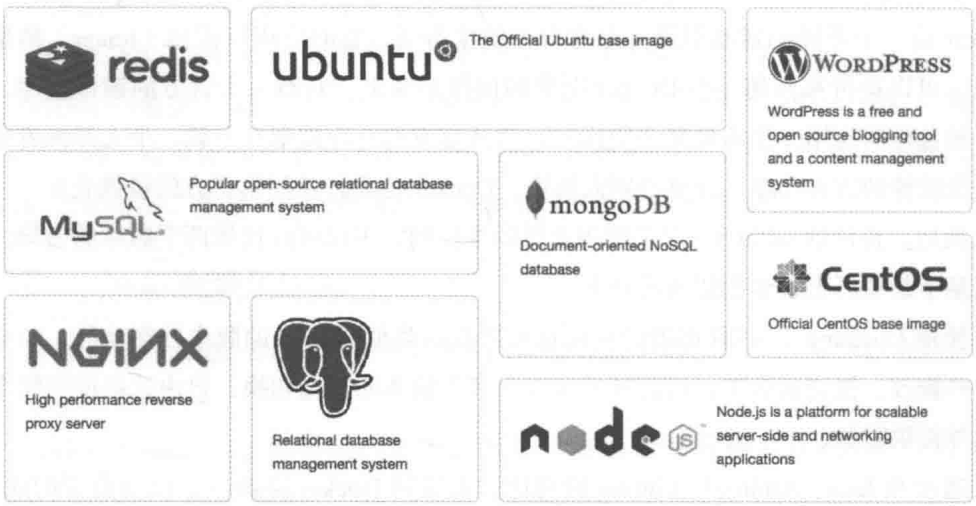


图 1-1 Docker 官方仓库截图

2. Docker 自身程序

Docker 本身是一个单机版的程序，它运行在 Linux 操作系统之上，属于用户态程

序，通过一些接口和内核交互。它在机器上的位置如图 1-2 所示。

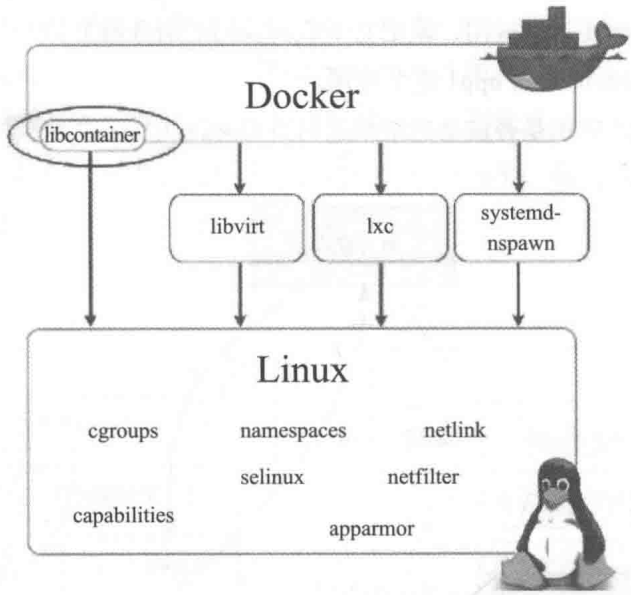


图 1-2　Docker 在 Linux 系统的位置

由于 Docker 需要用到 Linux 的 cgroups、namespaces 等特性，所以目前只能运行在 Linux 环境下，当然，通过虚拟机，也可以在 Windows 和 Mac 上使用 Docker。

Docker 是一个 C/S 的架构，它的 Docker Daemon 作为 Server 端，在宿主机上以后台守护进程的形式运行。Docker Client 使用比较灵活，既可以在本机上以 bin 命令的形式（如 Docker info、Docker start）发送指令，也可以在远端通过 RESTful API 的形式发送指令；Docker 的 Server 端接收指令并把指令分解为一系列任务去执行。

3. 工作流程

我们知道了 Docker 的构成，那么该如何使用 Docker 呢?

首先，要在 Linux 服务器上安装 Docker 软件包，并启动 Docker Daemon 守护进程。然后，就可以通过 Docker Client 端发送各种指令，Docker Daemon 守护进程执行完指令，向 Client 端返回结果。

假如要启动一个新的 Docker 应用 app1（名字是随便起的），它的工作流程大致如图 1-3 所示。

1）Docker Client 向 Daemon 发送启动 app1 指令。

2）因为我们的 Linux 服务器只装有 Docker 软件包，根本没有 app1 相关软件或服务，Docker Daemon 就发请求给 Docker 的官方仓库，在仓库中搜索 app1。

3）如果找到 app1 这个应用，就把它下载到我们的服务器上。

4）Docker Daemon 启动 app1 这个应用。

5）把启动 app1 应用是否成功的结果返回给 Docker Client。

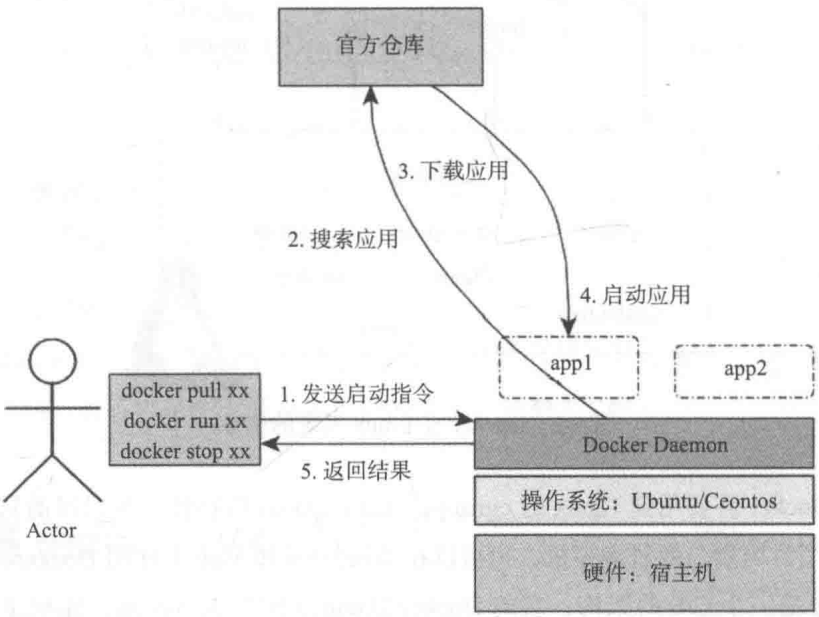


图 1-3 Docker 工作流程

Docker 的其他操作，比如停止或删除 Docker 应用和启动的流程差不多，这里就不再一一介绍了。

1.2.2 Docker 化应用的存在形式

我们知道，经过 20 多年的发展，Linux 下应用软件已经不计其数，不但种类繁多，而且安装部署方式也千奇百怪、不一而足，如有些软件依赖特定操作系统、有些依赖特定内核版本、有些依赖一些第三方软件和共享库等。另外，不同操作系统，不同的系统版本软件的配置和启动方式也存在很大差异。

既然软件安装部署方式没有一个统一的标准，那么 Docker 的官方仓库该如何做呢？总不能针对每个软件，写一个安装说明书吧。

换个角度想一下，用户的需求是什么——把软件运行起来，至于怎么安装软件、软件运行在什么操作系统下用户不太关心。那么，就把软件和它依赖的环境（包括操作系统和共享库等）、依赖的配置文件打包在一起，以虚拟机的形式放到官方仓库，供大家使用。只要有虚拟机的运行环境，就可以不做任何修改把软件轻松地运行起来。这种方式甚至不需要大家重复安装和配置软件，只要有一个人把软件安装和配置好，提交到官方仓库，其他人下载后就可以直接以虚拟机的形式运行起来。我们以这种方式解决了软件安装部署方式没有一个统一标准的问题，如图 1-4 所示。

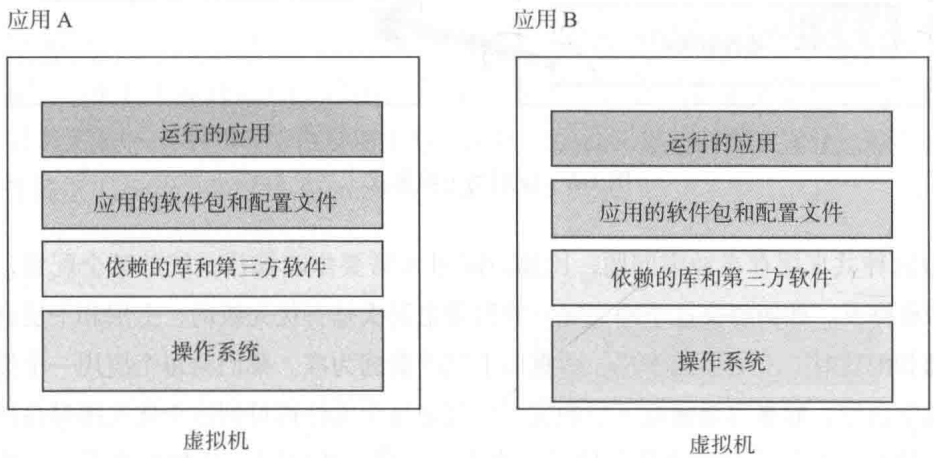


图 1-4　应用的组织形式（一）

但这种软件部署方式却存在很多问题，一般一个软件包大小也就几兆到几十兆不等，但一个操作系统却有好几个 G。如果每个软件都带上它依赖的操作系统，那么每个软件都有几个 G，不要说运行，仅仅下载 1 个软件都要数小时，是不是有“捡了芝麻丢了西瓜”的感觉？

Docker 为了解决这个问题，引入分层的概念。把一个应用分为任意多个层，比如操作系统是第一层，依赖的库和第三方软件是第二层，应用的软件包和配置文件是第三层。如果两个应用有相同的底层，就可以共享这些层。

以图 1-4 为例，假如应用 A 和应用 B 操作系统版本是一样的，它们就可以使用共享这一层，安装应用 A 时需要下载操作系统层，安装 B 应用就不用下载操作系统层，只需要下载它的依赖包和自身的软件包。因为主流的操作系统也就那么几个，最差情况下，也就把常用的操作系统都安装一遍，然后，包含操作系统的软件包就和传统的软件

包一样大小了，如图 1-5 所示。

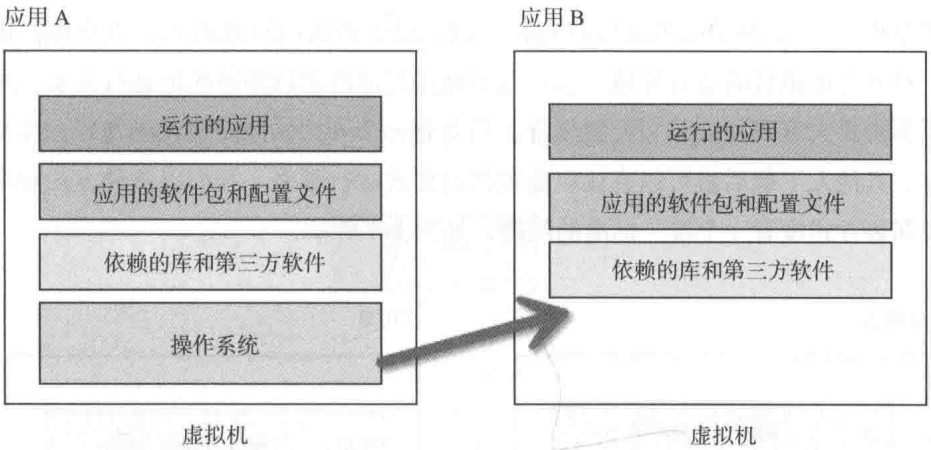


图 1-5 应用的组织形式（二）

但这种共享层存在冲突问题，比如，应用 A 需要修改操作系统的某个配置，应用 B 不需要修改。如何解决这个冲突呢？我们规定层次是有优先级的，上层和下层有相同的文件和配置时，上层覆盖下层，数据以上层的数据为准。我们给每个应用一个优先级最高的空白层，如果需要修改下层的文件，就把这个文件拷贝到这个优先级最高的空白层进行修改，保证下层的文件不做任何改变。这样，从应用 A 的角度来看，文件已经修改成功了，而从应用 B 的角度来看，文件没发生任何改变，如图 1-6 所示。

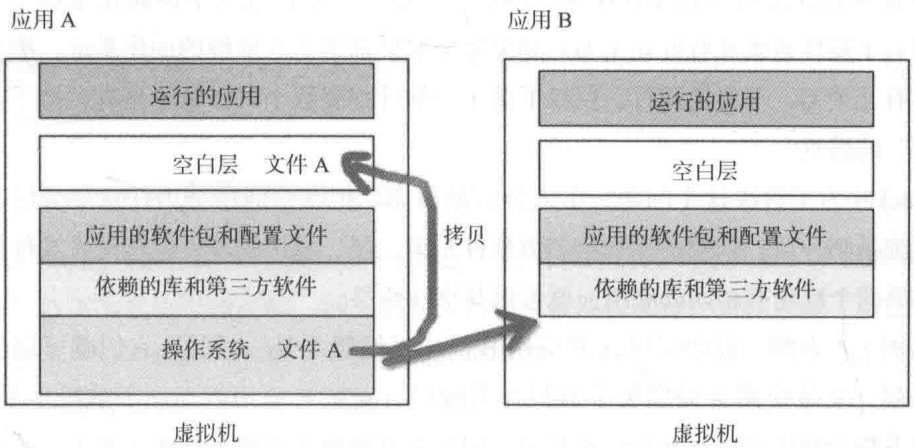


图 1-6 应用的组织形式（三）

Docker 的分层和写时拷贝策略，解决了包含操作系统的应用程序比较大的问题。但我们知道，主流的虚拟机（KVM、Xen、VMWare、VirtualBox 等）一般比较笨重，除了虚拟机本身运行要消耗大量的系统资源（CPU、内存等）外，启动一个虚拟机也需要花费数分钟，如何把虚拟机做到轻量化呢？

以 OpenVZ、VServer、LXC 为代表的容器类虚拟机，是一种内核虚拟化技术，与宿主机运行在相同 Linux 内核，不需要指令级模拟，性能消耗非常小，是非常轻量级的虚拟化容器，虚拟容器的系统资源消耗和一个普通的进程差不多。Docker 就是使用 LXC（后来又推出 libcontainer）让虚拟机变得轻量化。

在 Docker 的官方仓库里，只需它有完整的文件系统和程序包，没有动态生成新文件的需求；当把它下载到宿主机上运行对外提供服务时，有可能修改文件（比如输出新日志到日志文件中），需要有空白层用于写时拷贝。Docker 把这两种不同状态做了区分，分别叫作镜像（image）和容器（container），如图 1-7 所示。

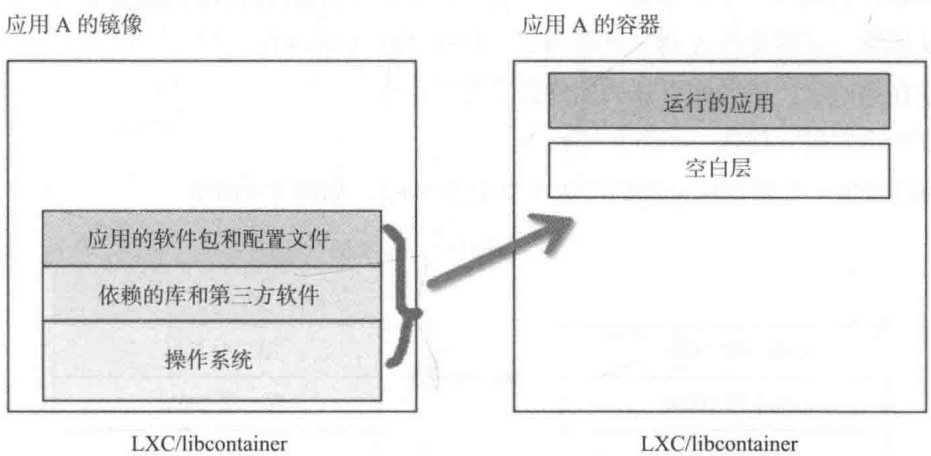


图 1-7　应用的组织形式（四）

在仓库中的应用都是以镜像的形式存在的，把镜像从 Docker 仓库中下拉到本机，以这个镜像为模板启动应用，就叫容器。

综上所述，镜像指的是以分层的、可以被 LXC/libcontainer 理解的文件存储格式。Docker 的应用都是以这种格式发布到 Docker 仓库中，供大家使用。把应用镜像从 Docker 仓库下载到本地机器上，以镜像为模板，在一个容器类虚拟机中把这个应用启动，这个虚拟机叫作容器。

在 Docker 的世界里，镜像和容器是它的两大核心概念，几乎所有的指令和文档都是围绕这两个概念展开的。

1.2.3 Docker 对变更的管理

对于软件开发来说，版本迭代、版本回退是常态，Docker 对变更管理又有什么特别之处呢？

假若有一个应用的 Docker 镜像，它的 V1.0 版本有三层，每层文件的大小如图 1-8 所示。

图 1-8 应用的分层结构和大小

接下来，我们需要对它做如下修改：

- 修改位于第一层的文件 A。
- 删除位于第二层的文件 B。
- 添加一个新文件 C。

Docker 会新增一个第四层，针对上面的修改需求，它处理方法如下：

- 把第一层的文件 A 拷贝到第四层，修改文件 A 的内容。
- 在第四层，把名称为 B 的文件设置为不存在。
- 在第四层，创建一个新文件 C。

通过增加一个第四层，我们的版本变更为 V1.1，如图 1-9 所示。

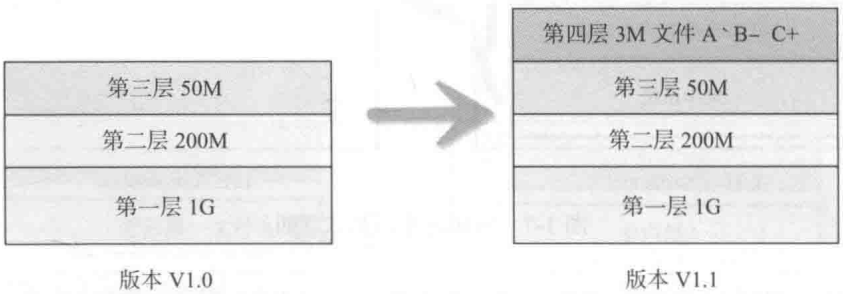


图 1-9 应用两个版本变化

我们想把应用的 V1.1 版本发布到 Docker 仓库，供其他宿主机使用。Docker 的仓库已经存在这个应用镜像的 V1.0 版本，也就存储有这个应用的第一层、第二层和第三层，我们上传 V1.1 版本时，不需要重复上传前三层，只需要把第四层（只有 3M 大小）上传到 Docker 仓库就可以了。

有一台远程服务器，正在运行这个应用的 V1.0 版，它想升级到 V1.1 版。因为它本机已经有这个应用的前三层，所以只需要从 Docker 仓库把第四层下载下来，就可以运行 V1.1 版，如图 1-10 所示。

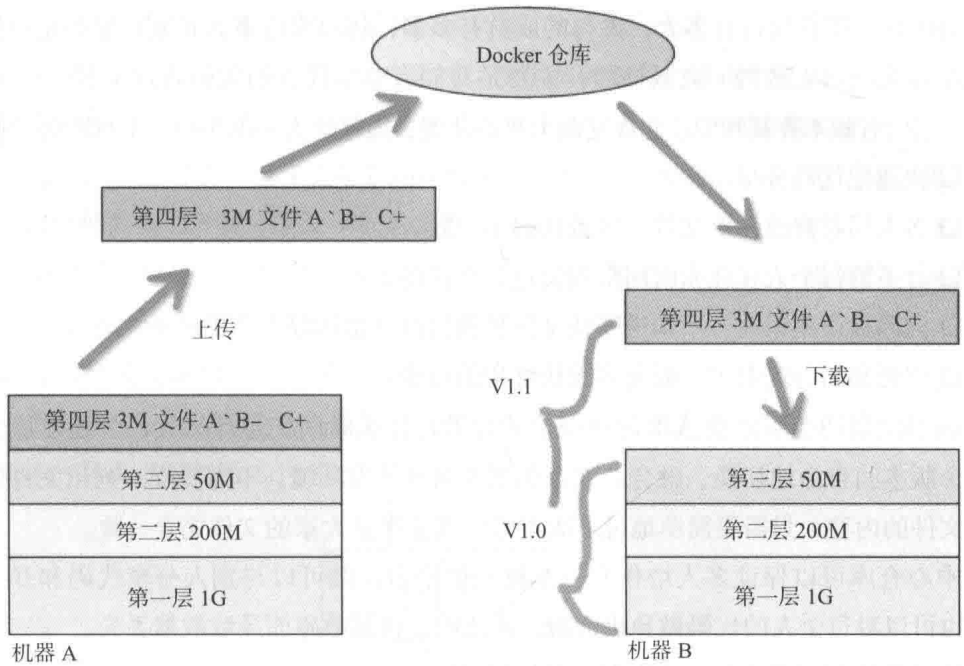


图 1-10　应用版本变更流程

综上所述，Docker 不仅具有版本控制功能，并且还能够利用分层特性做到增量更新。

1.3　为什么使用 Docker

当深入了解 Docker 后，你想在公司或部门推广 Docker，就需要给大家讲明白为什么要使用 Docker。

当讲 Docker 是什么时，你的听众是一批 Docker 爱好者，Docker 的原理和实现细节讲得越深入、越具体，就越受大家的欢迎；当讲为什么要使用 Docker 时，你的听众有领导，有开发、测试和运维人员，而他们可能从来没听过 Docker，或者对 Docker 知之甚少，他们更关注 Docker 能带来什么价值，引入 Docker 需要对现有的系统或程序做

多大改造，Docker 是不是足够的稳定，学习和使用 Docker 的成本有多高等问题。

1.3.1 从代码管理说起

现在的软件项目，失败的原因主要是结构设计不合理、编程人员水平不高、代码 bug 太多等因素。不管项目有多大，参与的成员有多多，代码修改多么频繁，很少是由管理上混乱导致项目失败的。究其原因，主要是我们有非常优秀的代码管理工具——Git 和 SVN。它们有版本控制和中心仓库这两大核心功能，能保证大家不用担心下列问题：

- 快速把代码分享给别人。
- 多人同时修改一个文件，导致代码不一致。
- 分不清每个人在什么时间都提交过什么代码。
- 代码被误删误改，不知道哪些文件被删被改，也恢复不到以前的状态。
- 变更和新分支太多，混乱到无法维护的地步。

版本控制功能不仅能清晰记录每个开发者在什么时间交过什么代码，还能让大家在各个版本间自由地切换、融合。大家如果要对齐开发环境，不需要逐一列出文件的名字、文件的内容，只需要简单地说下版本号，就能保证大家的文件完全一致。

中心仓库可以保证多人协作有一个统一的平台，既可以与别人分享代码和开发进度，也可以对每个人的代码做异机备份，防止因为机器故障而导致数据丢失。

软件部署和代码管理面临很多相似的问题：

- 如何快速把一台机器的应用环境分享给其他机器使用，用于扩容和故障时服务转移。
- 同一个功能模块的多台机器，软件版本和配置文件镜像出现不一致，很难被发现。
- 多人维护一套系统，很难清晰地记录下每个人都做过什么操作、每次变更都有哪些内容，以便在故障时快速回退。
- 有些配置文件或数据被误删了，恢复不回来，甚至很长一段时间都察觉不到。
- 随着操作系统版本、软件、硬件的更新迭代，系统维护的复杂度直线上升，混乱不堪。

既然代码管理做得这么好，那软件部署为什么不借鉴代码管理的方法呢？

代码管理的对象是纯文件（代码），体量小，有统一规范的文件编码方式，对平台无依赖。而软件部署管理的对象是一个环境，除了文件，还有二进制的软件和它依赖的运行环境（包括操作系统和依赖库）。由于操作系统和依赖库的体量很大，难以完全照

搬代码管理的方式，用版本控制和中心仓库来解决软件部署的问题。

1.3.2　当前的优化策略

下面让我们看看，对于软件部署，目前主流的解决方案是什么样子的。

把环境分为两个部分：基础环境和应用环境。

- 基础环境：包含机器硬件、操作系统、提供基础服务的应用（如 ssh、syslog 等）。
- 应用环境：包含应用需要的各种软件包和配置文件（虽然软件包中也可以包含配置文件，但对于一些变更频繁和需要个性化配置的文件，最好还是独立出来）。

对这两种类型的环境，采取不同的策略：

- 基础环境统一化，尽量保持完全一致。比如，使用相同的硬件服务器，运行相同版本的操作系统和基础软件。
- 应用环境分解出一个个独立服务，每个服务再分解为包和配置文件，使用版本控制和中心仓库来对包与配置进行管理。

整体的解决方案如图 1-11 所示。

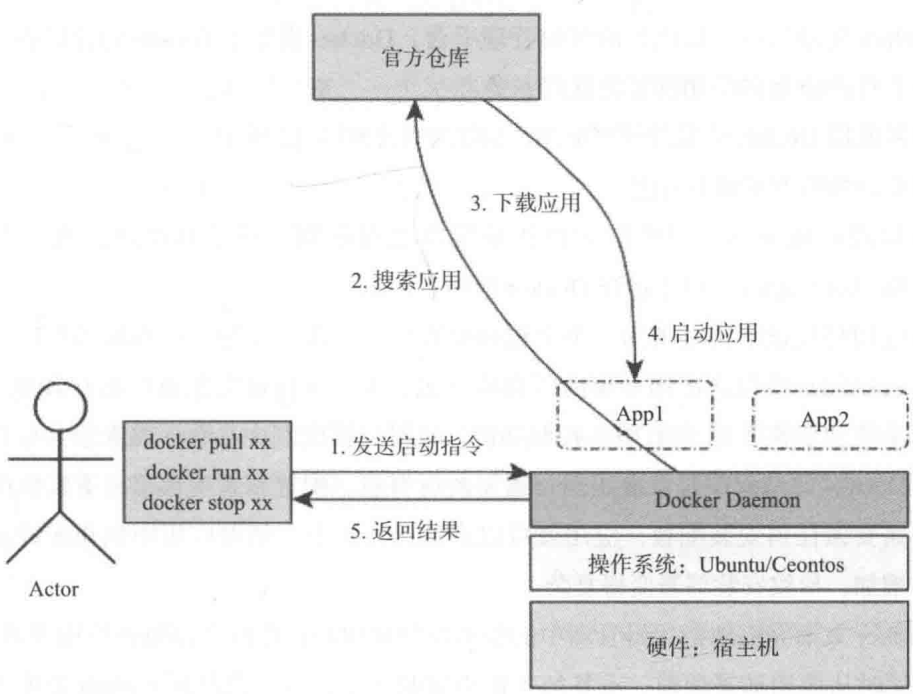


图 1-11　Docker 整体的解决方案

在这个解决方案中，我们把软件包和配置都版本化，每次变更都提交一个版本到中心仓库，然后再下发到各台机器上。由于每台机器的基础环境都保持一致，所以软件部署时可以不关注底层操作系统的适配问题，在一台机器打包编译的应用可以快速迁移到其他机器。

这个解决方案，在基础环境一致的前提下，实现了分解版的版本控制和中心仓库，针对分解出来的软件安装包和配置文件，可以通过版本来管理，并且可以把包和配置提交到中心仓库，让所有其他机器分享。

这个方案，存在以下两个问题：

- 基础环境难以改动：因为上层应用环境的软件和配置文件都是基于基础环境编译与配置的，一旦基础环境发生改变，可能导致上层的软件包和配置不能正常工作。
- 应用环境的维护成本取决于包和配置的数量：由于应用环境不是一个统一整体，而是分解出一个个包和配置，随着包和配置的增多，维护的复杂性也随之增加。

1.3.3 Github 版的应用部署解决方案

Github 是最流行、最优秀的代码管理平台，Docker 借鉴了 Github 的管理思路，打造了一个 Github 版的应用部署的管理方案。

如果使用 Docker 来管理部署应用，解决方案如图 1-12 所示。

和上一节的方案做下对比：

- 基础环境灵活，对硬件和操作系统都没有限制，只需要在每台机器上安装 Docker Engine，用于运行 Docker 应用。
- 应用环境也不再分解为一个个包和配置文件，而是作为一个有机的整体，这个有机的整体包含应用需要的所有软件包、配置文件和它依赖的运行环境（操作系统和依赖库），带有版本控制功能，也可以提交到中心仓库供大家共享。由于 Docker 的镜像中包含应用运行需要的所有包、配置和系统环境，下发镜像后不需要做任何安装配置，应用就可以直接运行，不会随着应用中包和配置数量的增加，导致安装部署变得复杂。

Docker 方案完美地把代码管理中的版本控制和中心仓库概念移植到应用部署领域，让大家顿时从应用部署烦琐、重复的工作中解脱出来，可以像使用 Github 管理代码那样优雅地管理应用的部署工作。

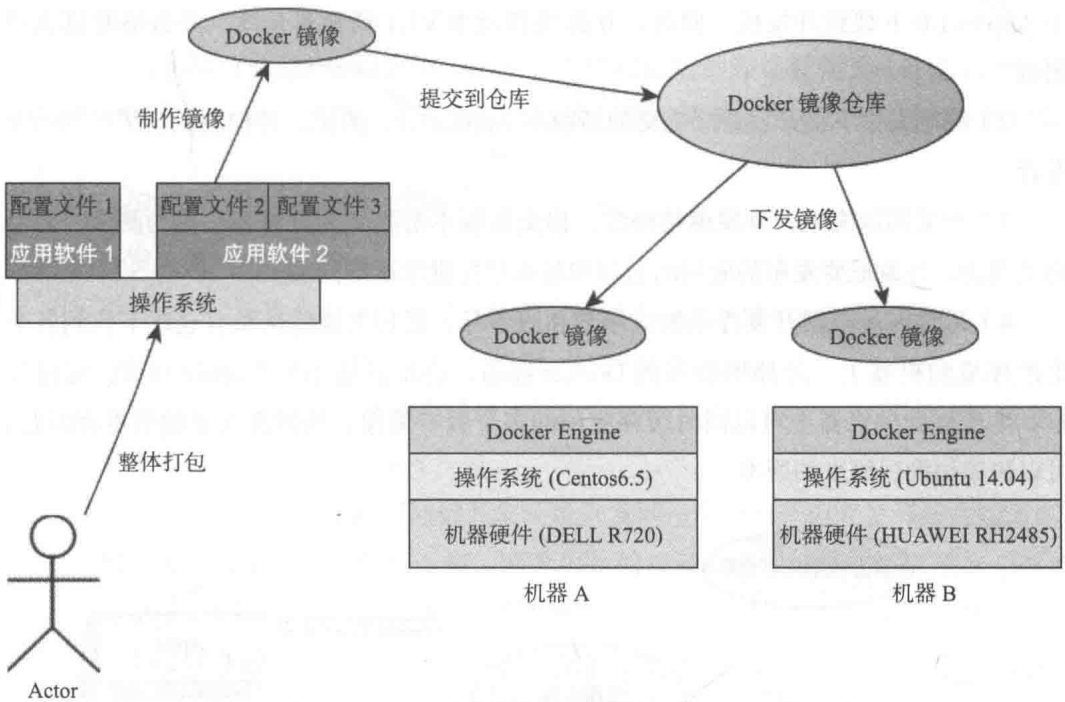


图 1-12　使用 Docker 来制作和下发镜像流程图

Docker 这个方案能顺利实施，一个关键点是通过分层共享和增量变更技术把应用的运行环境（包括操作系统在内）这么一个庞大的体量顺利瘦身，让应用运行环境的安装和修改在大多数情况下与只装软件包一样轻量、简单。分层共享和增量变更技术在 1.2.2 节有过详细讲解，这里就不再复述了。

1.3.4　Docker 应用场景

下面描述两个典型的 Docker 应用场景，如图 1-13 所示。

在这两个应用场景中，有开发、测试（QA）和运维人员，分别维护开发、测试和生产环境的服务，他们有一个私有的 Docker 仓库，存储着各种各样的 Docker 化的应用镜像。

场景一

现在有一个需求，需要对应用 App1 做修改、测试和发布新变更到生产环境，工作步骤如下：

1）开发者先从私有仓库找到 App1 这个应用最新稳定版本，假设为 V1.0 版，把这

个 App1:v1.0 下载到开发机，修改，并提交新版本 V1.1 到私有仓库，并告诉测试人员测试。

2）测试人员下载开发者刚提交的新版本 App1:v1.1，测试，并把测试结果反馈给开发者。

3）如果测试失败，开发继续修改，提交新版本给测试人员做新一轮的测试；如果测试成功，开发把要发布的应用的名称和版本号提供给运维同事。

4）运维人员根据开发提供的应用名和版本号，把相关镜像从私有仓库下拉到各个生产环境的机器上，停掉旧版本的 Docker 容器，启动新版本的 Docker 容器，完成发布。生产环境的机器上可以同时缓存应用的多个版本镜像，如果新发布的版本有问题，可以快速切换回原来的版本。

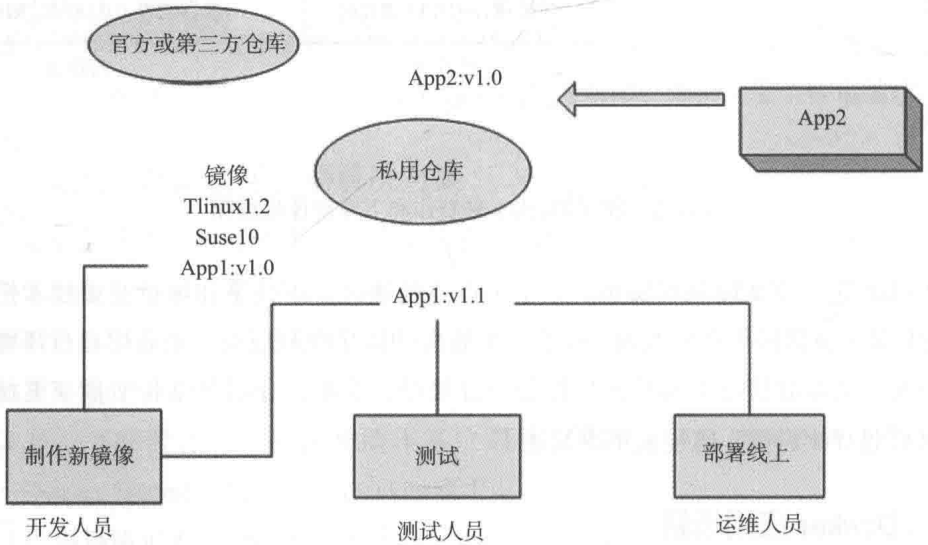


图 1-13 Docker 两个应用场景

因为 Docker 的应用镜像包含应用运行需要的所有软件包、配置和操作系统，所以开发者打包好 Docker 镜像，测试和运维人员从私有仓库下拉，不需要做任何修改，就可以运行起来，并保证和开发者运行的环境完全一致。“一处编译，到处运行”，真的很方便。

场景二

有一个应用 App2，目前在生产环境正常运行，但由于系统比较老、缺乏维护、开发和运维经历过更替且交接文档不全，现在谁都不知道该如何部署 App2 这个应用到新

服务器上。

如果 App2 是 Docker 化的应用，它可以直接把运行的环境转换为一个带版本号的 Docker 镜像（比如 App2:v1.0），提交到私有仓库，供开发者修改或运维人员发布新机器。

1.3.5　Docker 可以解决哪些痛点

1. 开发人员

不管在大公司还是小公司，开发人员经常被如下问题困扰：

- 为了节约成本，一台开发机多人使用，管理混乱，相互干扰。
- 一个开发往往只用一套开发环境，同时有多个开发任务时，不得不反复修改开发环境，以适应不同的开发任务。
- 多个开发人员希望保持相同的开发环境开发同一个项目，但开发环境难以复制，即便大家起初的开发环境一样，随着项目的滚动、开发环境的不停更新，很难保证每个人的开发环境都同步更新。
- 开发机硬件故障，需要更换新机器，重新搭建开发环境是件头疼的事。如果硬件故障，重要数据没备份，那就更让人崩溃。
- 打算调研下新软件，安装配置文档复杂，仅仅把软件安装、运行起来就要花费大半天时间。

上面问题带来的工作量不能体现开发人员的核心价值，但是是不得不面对的。

如果使用 Docker，可以轻松解决上述问题：

- Docker 化的应用使用容器虚拟化技术，每个应用都运行在独立的虚拟化环境中，天然具有隔离性，不用担心一机多用造成的管理混乱。
- 开发人员在多任务开发时，可以并行启动这些应用的 Docker 容器，每一个 Docker 应用有一个独立的运行环境，互不干扰。
- 开发机硬件故障，在新开发机上，重新从 Docker 仓库下拉开发环境的镜像，一两分钟内就可以重新搭建一套开发环境，并且即便新旧开发机的硬件和操作系统不一致，重新搭建的开发环境仍能保持和原来的环境一模一样。另外，还可以通过 Docker 仓库，把重要变更及时备份到远端。
- Docker 的每个复杂软件都可以制作成 Docker 镜像，分享给大家使用。随着 Docker 的流行，几乎所有主流的软件都提供 Docker 化的部署方式。软件部署将成为再简单不过的事情。

2. 测试人员

测试人员经常费了九牛二虎之力测出一些 bug，和开发逐一核对，发现大多数 bug 都是开发和测试环境不一致造成的。

测试人员经常为配置不同的测试环境浪费大量的时间，还是不能保证和开发环境完全保持一致，开发人员虽然很认真负责地告诉测试人员如何配置测试环境，但还是经常性地遗漏一些配置。

使用 Docker，不需要做任何配置，就能保证开发和测试环境完全一致，测试人员只需要关注测试本身就可以了。

3. 运维人员

运维人员大部分时间都浪费在装软件、修改配置上，重复单调，经常半夜还要起来做紧急扩容、故障机服务迁移。如果使用了 Docker，好处显而易见：

- 服务具备快速部署能力，扩缩容、版本回退在几秒钟内就可以完成。
- 基于同一个 Docker 镜像部署服务，可以保证每台机器应用完全一致。
- 由于 Docker 化应用是虚拟化，多个应用可以混合部署在一台机器上，互不干扰，可以提高机器使用率。
- Docker 化的应用可以运行在不同的硬件和操作系统平台下，在不同的环境自由迁移。
- 通过 Dockerfiel 管理 Docker 镜像，即使系统多次易手、交接文档不全，运维人员也可以快速了解系统是如何搭建的。
- Docker 倡导“Build once，Run anywhere”，再烦琐的活儿，只需要做一次，制作成 Docker 镜像，在任何环境下都可以运行；还可以基于这个 Docker 镜像做修改，制作新的镜像。

上面只罗列几条好处，运维在使用 Docker 的过程中，还会发现很多意想不到的好处。一句话，Docker 可以让运维工作变得简单和易于维护。

1.3.6 Docker 的使用成本

坦白说，Docker 是有学习和使用成本的。

Docker 虽然已经做得足够简单易用，但由于它的定位是虚拟化容器，是一个单机版的应用，如果要基于 Docker 构建集群或 PaaS 管理系统，如 Web 管理界面、任务调度策略、监控报警等，则还需要自己开发或从开源社区寻求支持。

另外，传统的运维是以机器为中心，而 Docker 是以应用为中心，它会颠覆我们一些固有的运维习惯和运维方式。

但好在 Docker 有非常活跃的社区，不但有大量的开发者为 Docker 贡献代码、修复 bug，让 Docker 越来越好用、越来越稳定；还有大量的学习资料和问题解答。另外，很多公司和个人也为 Docker 提供大量优秀的第三方开源软件，如 kubernetes、fig、etcd 和 cadvisor 等。这些都有助于进一步降低 Docker 学习和使用的门槛。

如果你被上面的理由说服了，那就赶快跟着本书，学习和使用 Docker 吧！

1.4　本章小结

本章概括性地介绍了 Docker 是什么，它有哪些独特的物理特性，以及它适用于哪些场合和能带来哪些好处。它灵活便利，但也有一定的学习成本。下面章节中我们将循序渐进，讲解 Docker 的使用方法。

Chapter 2 第 2 章

初步体验 Docker

上一章概括性地介绍了 Docker 的发展历史、组织结构、功能特性和使用场景等方面的内容。本章主要从实践的角度，介绍如何在本地搭建一个 Docker 运行环境。

由于大多数用户的个人电脑用的都是 Windows 系统，所以我们就先来讲讲在 Windows 环境下如何安装和运行 Docker。

2.1 Windows 下安装 Docker

为了运行 Docker，你的电脑必须安装 64 位 Windows 7 及以上版本的系统（包含 Windows 8/8.1 和 Windows 10）。另外，你要确保 CPU 是支持虚拟化的，并且系统的虚拟化是打开的。

我们也可以先跳过系统是否支持虚拟化的检查直接安装 Docker。安装运行过程中如果出现错误再回头检查。

安装步骤如下：

1）到官网 https://www.docker.com/toolbox 下载 Docker Toolbox。

2）双击 Docker Toolbox，按照指引进行安装。

3）如果安装成功，在桌面上会有如图 2-1 所示

图 2-1 安装成功后的界面

的两个快捷图标。

其中 Kitematic 是 Docker 图形化管理方式，Docker Quickstart 是命令行管理方式。

4）双击运行 Docker Quickstart 快捷图标。如果出现如图 2-2 所示的运行结果，则表明安装正常。

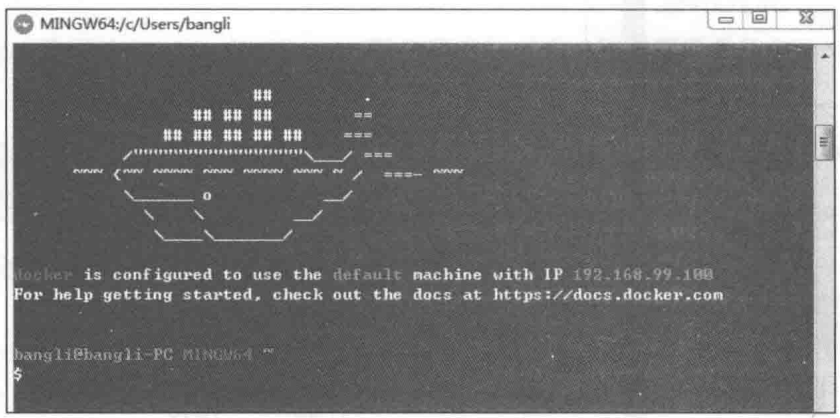


图 2-2　运行 Docker Quickstart 的成功界面

如果出现如图 2-3 所示的运行结果，表明系统的虚拟化是被禁止的。

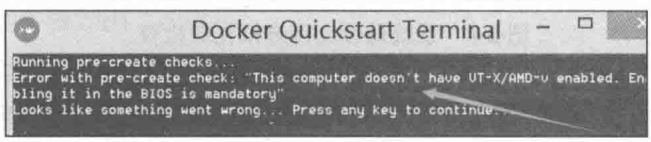


图 2-3　运行 Docker Quickstart 的报错界面

5）如果系统的虚拟化是被禁止的，可以通过如下方式检查。

在 Windows7 下，通过下载 Microsoft® Hardware-Assisted Virtualization Detection Tool（https://www.microsoft.com/en-us/download/details.aspx?id=592）工具，按照屏幕提示检查。

在 Windows 8 或 8.1 下，右击屏幕左下角的“Start”，选择“任务管理器（T)”，在弹出的界面上，单击左下角的“详细信息（D)”选择“性能”，找到右边的虚拟化，查看是否支持，如图 2-4 所示。

6）经过确认，如果由于系统禁用虚拟化导致 Docker 运行失败，需要在开机的 BIOS 中激活虚拟化，电脑型号不同，BIOS 的设置方式略有不同。

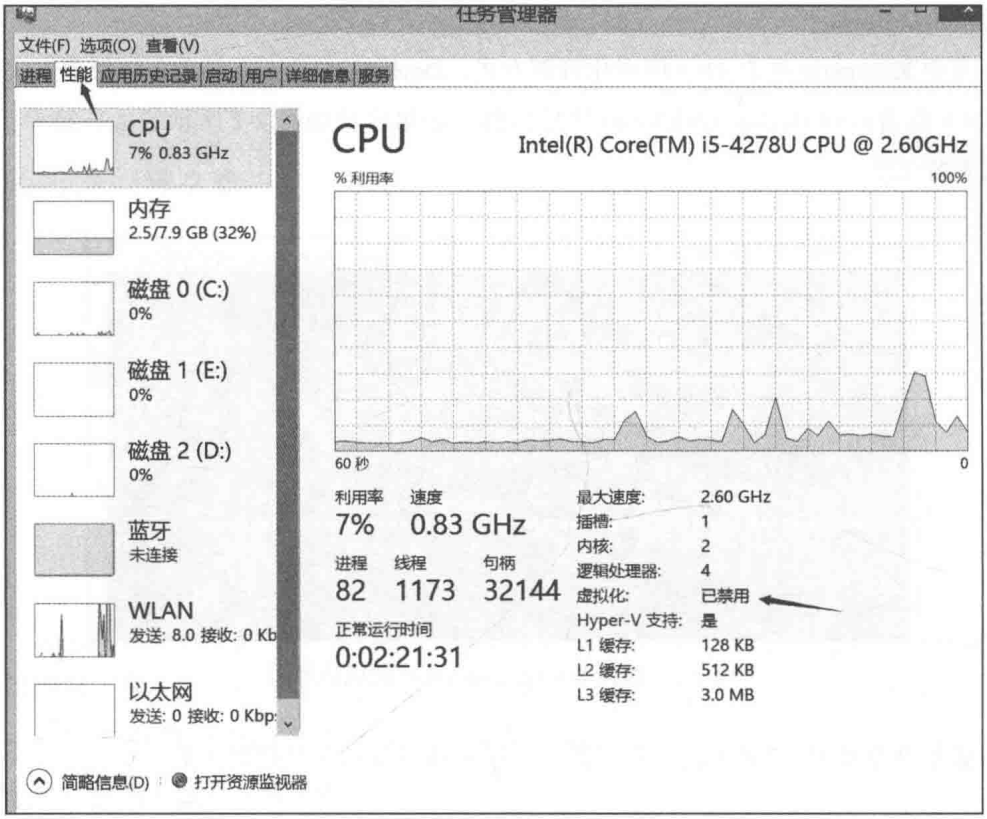


图 2-4 系统是否支持虚拟化的检查

7）如果设置成功，可以通过运行下面的命令确认 Docker 工作是否正常。

运行如下命令：

```
$ docker run hello-world
```

得到如下结果：

```
Unable to find image 'hello-world:latest' locally
511136ea3c5a: Pull complete
31cbccb51277: Pull complete
e45a5af57b00: Pull complete
hello-world:latest: The image you are pulling has been verified.
Important: image verification is a tech preview feature and should not be
relied on to provide security.
Status: Downloaded newer image for hello-world:latest
Hello from Docker.
This message shows that your installation appears to be working correctly.
```

```
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (Assuming it was not already locally available.)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.
```

至此，我们的 Docker 在 Windows 下已经安装成功。如果需要升级，下载最新版本的 Docker Toolbox 重新安装即可。

Docker 已经安装成功，下面，就让我们通过一个例子来讲解如何使用 Docker。

2.2　利用 Docker 搭建个人博客

WordPress 是一款功能强大的个人博客系统。使用者众多，社区非常活跃，有丰富的插件模板资源。使用 WordPress 可以快速搭建独立的博客网站。

2.2.1　传统的安装方法

按照传统的安装方法，参考官方的安装文档（http://codex.wordpress.org/zh-cn: 安装 _ WordPress），安装步骤如图 2-5 所示。

WordPress 运行环境需要如下软件的支持：

- ❑ PHP 5.6 或更新版本。
- ❑ MySQL5.6 或更新版本。
- ❑ Apache 和 mod_rewrite 模块。

虽然有“著名的 5 分钟安装”，但由于需要安装 PHP、MySQL 和 Apache 等软件，对于一个经验丰富的老手，安装 WordPress 也需要一个小时的时间。如果用户对 PHP、MySQL 和 Apache 不熟悉，花费一天甚至一周时间估计也不能把 WordPress 安装成功。

2.2.2　使用 Docker 进行安装

如果使用 Docker 来安装 WordPress 呢？一个完全不知道 PHP、MySQL 和 Apache 的小白用户，只通过两条命令就可以把 WordPress 安装成功，所花费的时间也只有几分钟（主要是从网上下载 Docker 版的 WordPress）。

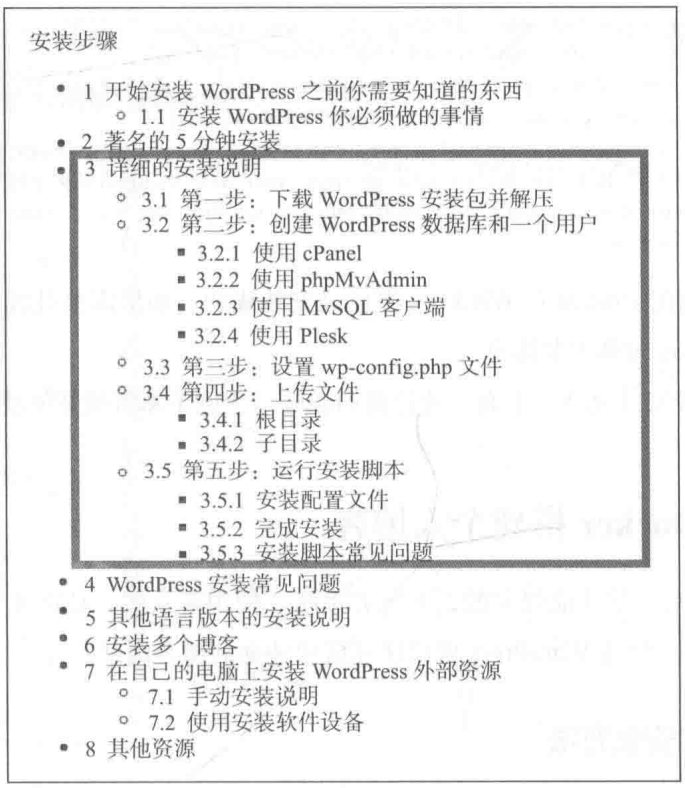


图 2-5 传统 WordPress 安装步骤

下面让我们来见识一下这两条神奇的 Docker 指令吧。

双击桌面的“Docker Quickstart”快捷图标，出现命令行界面，输入如下两条指令：

```
$ docker run --name db --env MYSQL_ROOT_PASSWORD=example -d mariadb
$ docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

等待下载完成，WordPress 就已经安装成功了。

注意 由于要下载的 mariadb 和 WordPress 文件比较大，建议尽量使用有线网络替换 Wi-Fi 无线网络。

安装完成后，如何访问 WordPress 呢？

在“Docker Quickstart”启动的命令行界面通过输入如下指令获取 IP：

```
$ docker-machine.exe ip
```

```
192.168.99.100
```

然后在浏览器中输入 http://192.168.99.100:8080，会出现如图 2-6 所示的界面。

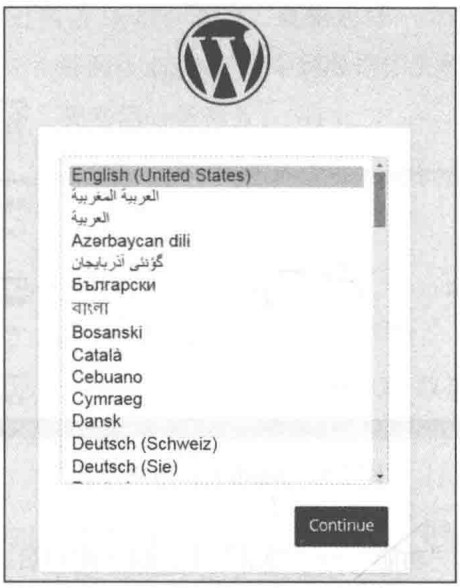


图 2-6　安装成功的引导界面

按照界面的指引，选择网站支持的语言、输入网站标题和用户名密码等信息，配置就完成了，如图 2-7 所示。

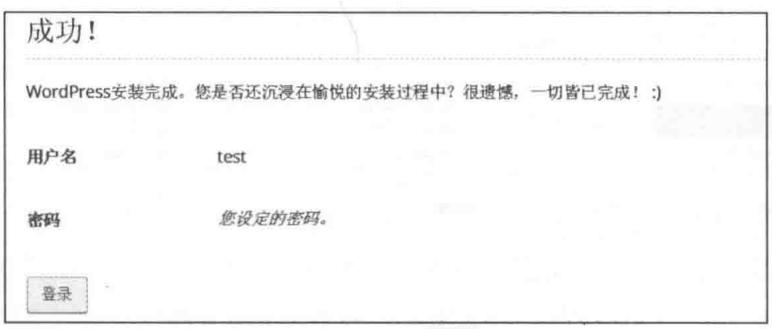


图 2-7　配置成功界面

在浏览器中重新输入 http://192.168.99.100:8080，一个高端、大气、上档次的个人博客就呈现在我们面前了，如图 2-8 所示。

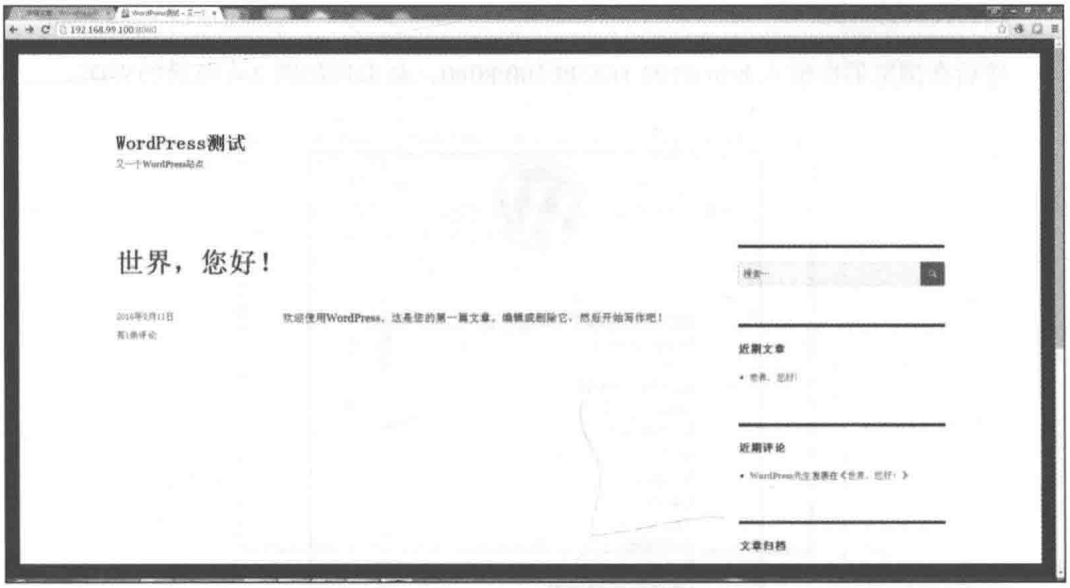


图 2-8 WordPress 用户界面

在页面的右下角，在“功能”→“登录”中，输入用户名、密码即可进入 WordPress 的管理界面，对博客进行修改和配置，如图 2-9 所示。

图 2-9 WordPress 管理界面

至此，一个完整的博客就搭建完成了。

2.2.3　解惑

在上一节，我们通过两条 Docker 指令，就搭建好一个个人博客网站。大家在惊讶的同时，是不是也很疑惑：那两条 Docker 指令到底是什么意思？

下面我们就解释一下。先看第一条指令：

```
docker run --name db --env MYSQL_ROOT_PASSWORD=example -d mariadb
```

其中：

docker run 是一条 Docker 指令，后面的所有内容“--name db --env MYSQL_ROOT_PASSWORD=example –d mariadb”是 Docker 指令的参数。

这条指令含义是启动一个 mariadb 数据库（MySQL 数据库的一个分支），数据库的管理员 root 的密码设置为 example，让这个数据库运行在后台，给它起了一个唯一的名字 db 并进行标示。

这些都是通过参数的指定来实现的。

通过参数最后一部分内容“mariadb”来告诉 docker run 启动的是一个 mariadb 数据库。

通过“--env MYSQL_ROOT_PASSWORD=example”参数，设置传入环境变量 MYSQL_ROOT_PASSWORD 为 example，就会在初始化 mariadb 数据库时 root 把密码设置为 example。

通过“-d”参数，把启动的 mariadb 数据库设置到后台运行，如果没有该参数，该进程就会在前台运行。

通过“--name db”参数，给这个运行的 mariadb 数据库起一个名字。假如我们在一台机器上要启动多个 mariadb 数据库，就可以通过这个名字定位到不同的数据库。

另外一个问题是，我们使用 docker run 来运行 mariadb，但 mariadb 从哪里来呢？docker run 指令首先会从本机检查有没有 mariadb 程序，如果没有，就会从 Docker Hub 搜索并下载该程序，Docker Hub 就像 iPhone 的 App 应用商店。

现在，我们理解了第一条指令是启动一个 mariadb 数据库。这是 WordPress 运行环境的三个必需条件之一。接下来看看第二条指令：

```
docker run -name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

和第一条指令非常类似，通过“ docker run”在后台运行 WordPress 程序。但它多出两个参数“--link”和“-p”。

WordPress 是把博客和个性化信息存储到数据库，所以需要和数据库建立连接。在第一条指令中我们已经启动了 mariadb 数据库，并把它命名为 db。在第二条指令中，我们通过“--link db:mysql”参数，把 WordPress 和数据库建立起了连接。

WordPress 是通过监听 Apache 的 80 端口对外提供服务。但每台机器的 80 端口只有一个，假如 80 端口被其他应用占用了怎么办呢？我们通过“ -p 8080:80”参数，把原服务的 80 端口映射到 8080，这样就可以通过访问 8080 端口来访问服务。上一节我们访问 WordPress 的 URL（ http://192.168.99.100:8080 ）端口就是 8080，原因就在于这里。我们可以通过“-p”把 80 端口映射到任意端口上。

2.2.4 其他注意事项

360 杀毒软件会把 Docker 识别为病毒而删掉，所以出现类似情况需要把 360 杀毒软件停掉再重新安装 DockerToolBox。

2.3 本章小结

这一章，我们介绍了在 Windows 环境下如何安装 Docker，并且通过搭建一个个人博客 WordPress，让大家了解 Docker 指令的运行方式。大家有没有被 Docker 的神奇特性深深吸引住呢？如果有，那么我们接下来就“折腾”得大一些。

第 3 章 Chapter 3

Ubuntu 下使用 Docker

第 2 章我们介绍了在 Windows 下如何搭建一个 Docker 运行环境。这一章我们要切换环境，在 Ubuntu 系统下使用 Docker。为什么要切换到 Ubuntu 下呢，还要从 Docker 的运行平台说起。

3.1 Docker 的运行平台

首先，我们需要知道 Docker 可以在哪些操作系统下运行。截止到 2016 年 3 月底，几乎所有的 Linux 系统（如 Red Hat Enterprise Linux(RHEL)/Centos、Debian/Ubuntu、gentoo、arch linux 等）和主流的云平台服务（如 Amazon EC2、Google Cloud Platform、Rackspace Cloud、阿里云等）都支持 Docker，非 Linux 平台的 Mac OS X 和 Microsoft Windows 通过 Docker Toolbox 来支持与运行 Docker。

需要注意的是，虽然几乎所有的系统和平台都支持 Docker，但并不是说每种系统的所有版本都支持。因为 Docker 是 2013 年 3 月才诞生，用到 Linux 内核 3.8 以上的系统才具有的一些新特性，刚开始时只是在 Ubunut 下运行，各大厂商看到 Docker 的优势，才纷纷拥抱 Docker，推出支持 Docker 的系统版本。所以只有相对比较新的系统版本才开始支持 Docker。

那么，是不是只有运行 Linux 内核 3.8 以上的系统才能支持 Docker？这个说法基本

正确，但 RHEL/Centos 系列是个例外，因为它没有用原生的 Linux 内核，它的内核是修剪过的，根据需要，它会在 Linux 的低版本的内核加入高版本的特性，看到的版本号却还是低版本的内核编号。正是这个原因，内核版本为 2.6.32-431 的 RHEL/Centos6.5 就已经开始支持 Docker 了，因为它把 Linux 高版本内核中支持 Docker 的特性迁移到 2.6.32-431。

由于 Docker 跨平台的特性，不同的系统平台有不同的优势，用户可以根据自己的需求进行选择。

Docker 是在 Ubuntu 下诞生和发展的，Docker 的最新特性都是在 Ubuntu 下开发和测试的，所以 Ubuntu 是支持 Docker 的最好的操作系统。

REHL/Centos 有强大的研发实力，在保证系统稳定的前提下，可以快速把 Docker 的新特性移植到该系统下，所以对系统稳定性要求比较高的生产环境，推荐使用 REHL/Centos。

CoreOS 是为 Docker 而生的操作系统，除了对 Docker 支持良好外，还集成 etcd、fleet 等，方便对 Docker 的集中管理。最近比较流行的 PaaS 开源软件 Flynn 和 Deis 都是基于 CoreOS 来做的。CoreOS 是对 Docker 支持最深入的操作系统，但是该系统比较新，稳定性有待时间的检验。另外，CoreOS 还推出了自家的类 Docker 的容器——Rocket，后续对 Docker 的支持有待观察。

在 Docker 自身工具包 Docker Toolbox 的帮助下，Docker 在 Windows 和 OS X 系统也有良好的表现，对非 Linux 用户（大部分的开发者）是一个福音。但是 Windows 和 OS X 系统本身并不支持 Docker，工具包 Docker Toolbox 通过集成一个 Linux 的虚拟机，让 Docker 运行起来，所以对于一些复杂的应用，Windows 环境并不能胜任。我们上一章介绍了 Windows 下的 Docker，主要是为了让大家快速体验 Docker，如果大家想深入学习，还是建议大家安装 Linux 环境（尤其推荐 Ubuntu）。

Docker 对操作系统的另外一个要求是必须是 64bit 的系统。

如果大家只有一台 Windows 计算机，建议大家再安装一个 Ubuntu 系统，形成双系统。不建议在 Windows 系统下通过虚拟机安装 Ubuntu，这样有些功能体验不好。

3.2 安装 Windows 和 Ubuntu 双系统

安装 Ubuntu 有很多方法，现在我们只介绍如何通过 U 盘安装，其他安装方式大家

可以自己去尝试。

准备工作：

- 一个存储空间不小于 4G 的 U 盘
- 下载 Ubuntu 安装 ISO 镜像

对于初学者，建议使用 64 位 Ubuntu Kylin 14.04.2 版本，因为它是 Ubuntu 最新长期维护的版本，并且对中文的支持很好。下载地址：http://www.ubuntu.org.cn/download/ubuntu-kylin-zh-CN。

3.2.1　制作 Ubuntu 安装 U 盘

我们使用 Win32 Disk Imager 工具制作 Ubuntu 的安装 U 盘，请到官方网站下载，下载地址是 http://sourceforge.net/projects/win32diskimager/files/latest/download。或者搜索国内的大型下载网站（如华军软件园）下载。

下载后双击进行安装，安装过程中，出现如下界面，勾选“Create a desktop icon”，这样安装成功后，在桌面会出现 Win32 Disk Imager 的图标。

通过如下步骤制作安装 U 盘：

1）先插入 U 盘，以管理员的身份运行 Win32 Disk Imager。

2）选择接入 U 盘的盘符（计算机最好只接入一个 U 盘，以免选错）。

3）在 Image File 中，选择系统 ISO 软件包（注意：ISO 包需要放在英文或数字目录下，即不能放在中文目录下）。

在“保存类型”中选择 *.*，这样才能发现 ISO 系统包。

4）单击“Write”按钮。

如图 3-1 所示。

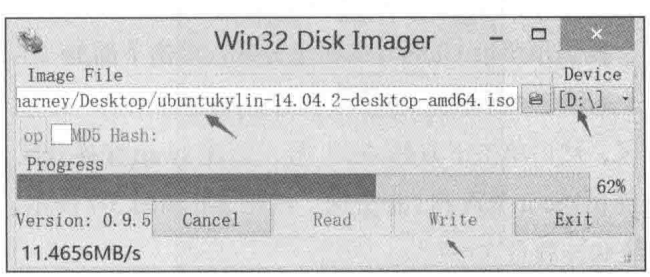


图 3-1　Win32 Disk Imager 使用界面

等待 10 分钟左右，会有弹框提示制作成功。

3.2.2 通过 U 盘安装 Ubuntu

安装 Ubuntu 之前，务必把计算机上的重要数据备份，并且预留一个空白的磁盘分区（不小于 30G）给 Ubuntu 使用。然后在 BIOS 中设置 USB 启动优先，接着插入安装 U 盘，重启计算机，就进入 Ubuntu 安装界面，如图 3-2 所示。

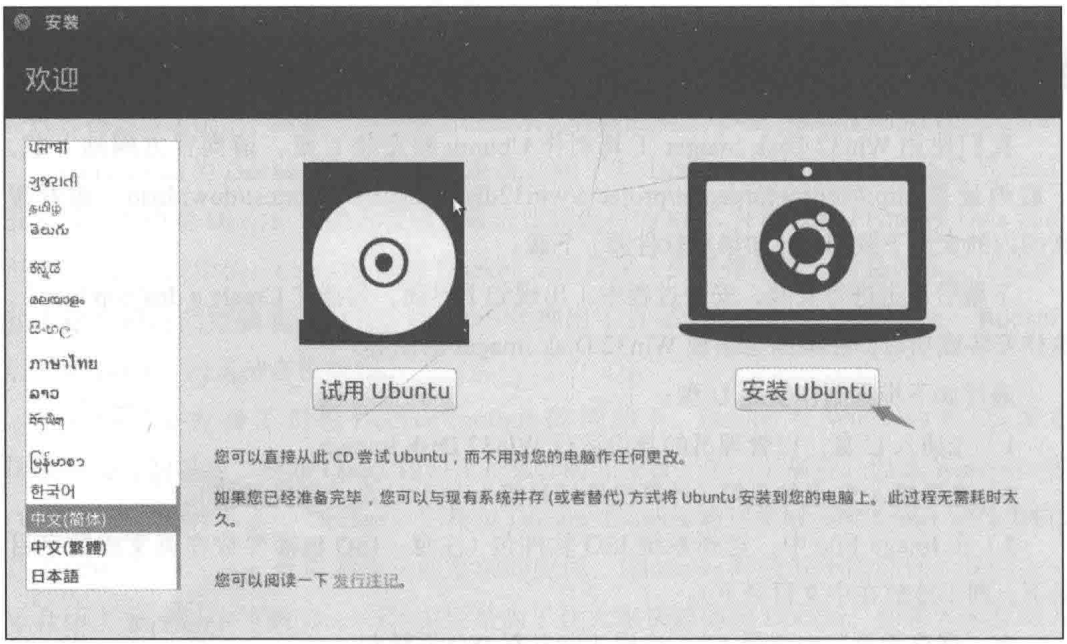


图 3-2 安装方式选择界面

接下来单击“继续”按钮，直到出现如图 3-3 所示的界面，选择最下面的“其他选项”。

进入分区界面，找到预留给 Ubuntu 的磁盘分区，单击下面的“-”号删除该分区，形成一个空闲分区，如图 3-4 所示。

选中该空闲分区，然后单击下方的“+”号，创建 swap 交换分区，如图 3-5 所示。

按相同的方法创建 /boot 分区和 / 根分区。如图 3-6 和图 3-7 所示，注意各分区的大小、用于的文件系统和挂载点。

创建完分区如图 3-8 所示。

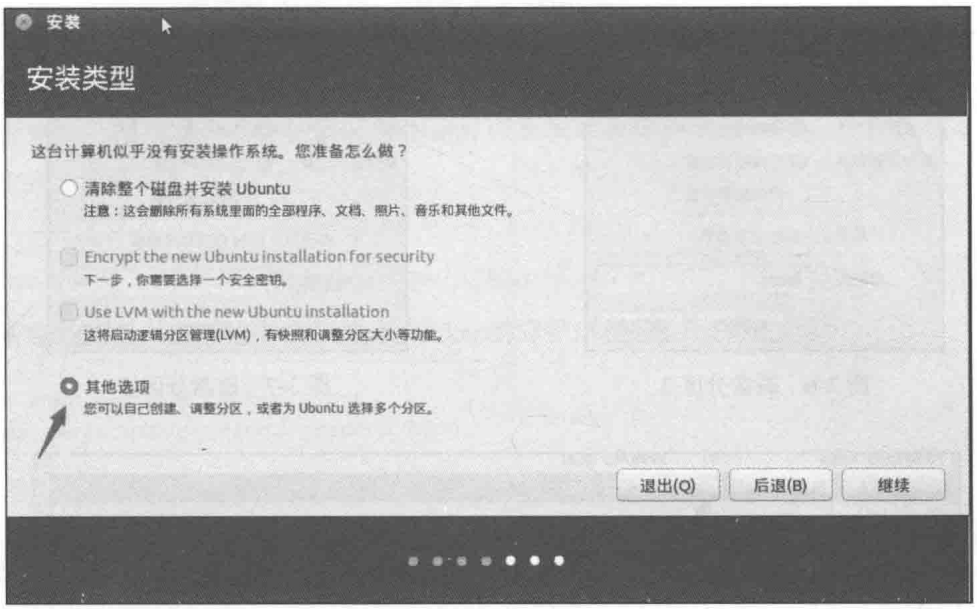


图 3-3　安装类型选择

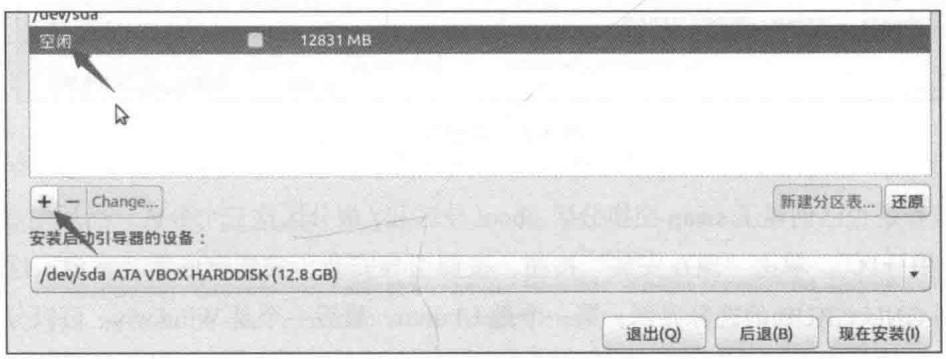


图 3-4　磁盘分区 1

图 3-5　磁盘分区 2

图 3-6 磁盘分区 3

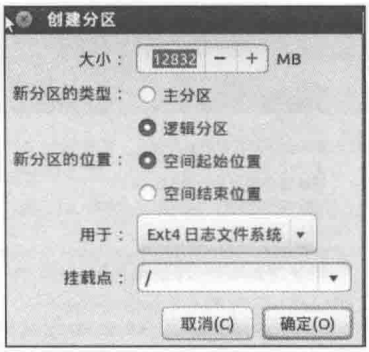


图 3-7 磁盘分区 4

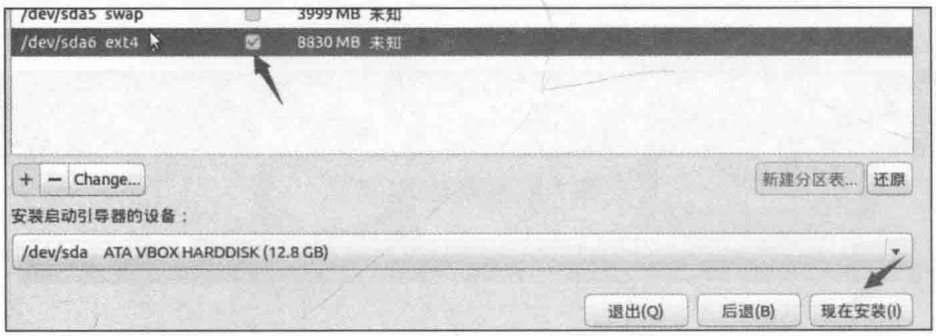


图 3-8 磁盘分区 5

检查是否已创建了 swap 交换分区、/boot 分区和 / 根分区这三个分区。然后勾选“格式化 / 根分区”，单击“现在安装”按钮，按照引导安装，安装完成后，重启。这时你会看到 GNU GRUB 的选择界面，第一个是 Ubuntu，最后一个是 Windows。可以分别进到两个系统，看看系统是否正常。如果正常，那么我们的 Ubuntu 已经安装成功了，接下来就可以安装 Docker 了。

3.3 在 Ubuntu 下安装 Docker

通过 GNU GRUB 选择进入 Ubuntu 系统，配置好网络。

先通过下面命令更新一下 apt 软件源。

```
sudo apt-get update
```

安装 Docker 有两种方式。

方法一：从 apt 源安装 docker.io，但版本比较旧。

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

方法二：使用官方提供的安装脚本，可以安装最新版本的 Docker，推荐使用这种安装方式，安装命令如下：

```
sudo apt-get install curl
curl -sSL https://get.docker.com/ | sh
```

安装完成后，通过如下命令启动 Docker 的守护进程。

```
$ sudo service docker start
docker start/running. process 3050
```

然后，可以通过如下脚本检查 Docker 安装是否成功。

```
$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
03f4658f8b78: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/userguide/
```

如果不想每次运行 Docker 都使用 sudo 权限，可以把用户加到 Docker 组中。例如，

我的用户名为 harney，则添加命令如下：

```
$ sudo usermod -aG docker harney
```

重启后生效，再次执行 Docker 的指令，直接输入“docker ××”，不需要加“sudo”了。

现在在 Ubuntu 下的 Docker 已经安装成功了。

3.4 再次体验 Docker

我们介绍了如何在 Ubuntu 系统下安装 Docker，并且指出 Ubuntu 是对 Docker 支持最好的系统。这一节我们就再次介绍几个例子，让大家更深入地体验 Docker。

3.4.1 再看个人博客 WordPress 的搭建

还记得第 2 章在 Windows 环境下通过两条 Docker 指令搭建 WordPress 吗？现在切换到 Ubuntu 系统下，再来看看这两条指令是否有效。

打开 Ubuntu 的命令行终端，依次执行这两条 Docker 指令。

```
$ docker run --name db --env MYSQL_ROOT_PASSWORD=example -d mariadb
$ docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

由于需要从网上下载几百兆的文件，请耐心等待指令执行完成。

然后通过 ifconfig 命令查看本机的 IP 地址。比如，我的地址是 192.168.10.103，在浏览器中输入 http://192.168.10.103:8080，出现如图 3-9 所示的界面。

它和 Windows 下 WordPress 配置界面完全一样。

在 Windows 和 Ubuntu 不同系统环境下，我们使用相同的 Docker 指令，就可以把 WordPress 安装成功。这体现了 Docker 非常优良的跨平台的特性。

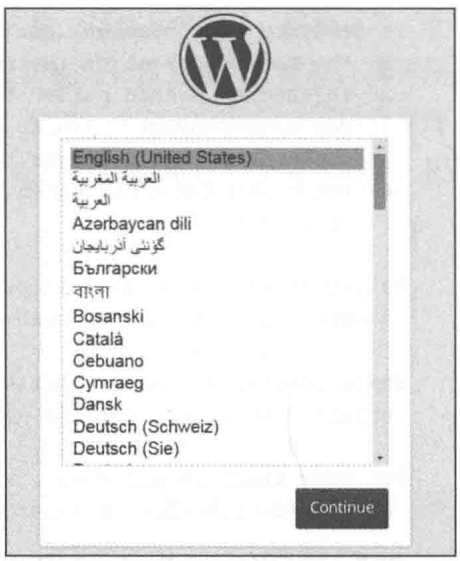


图 3-9 WordPress 安装成功界面

3.4.2 开源的版本控制利器——GitLab

作为一名程序员，都应该知道“程序员的

维基百科全书”——GitHub。它提供Web化的界面，很方便地对大型项目的代码进行协作开发和版本控制。但它也存在一些缺点，如托管的项目必须公开代码，如果建立私有仓库（代码不公开），需要收费；在国内访问GitHub有时会出现访问不了的情况；等等。

GitLab是一个类GitHub的开源的代码管理工具，它实现了GitHub大部分功能。它的优势是可以实现本地部署，搭建公司内部的版本控制系统。

下面，我们还是利用Docker，看看如何搭建GitLab服务。

1. 搭建GitLab服务

我们使用sameersbn/docker-gitlab来搭建GitLab服务，项目地址为https://github.com/sameersbn/docker-gitlab。它的运行环境由如下三部分组成：

- postgresql数据库
- redis缓存服务
- gitlab服务

我们使用Docker命令依次启动这三个服务：

启动postgresql：

```
docker run --name gitlab-postgresql -d \
  --env 'DB_NAME=gitlabhq_production' \
  --env 'DB_USER=gitlab' --env 'DB_PASS=password' \
  sameersbn/postgresql:9.4-12
```

启动redis：

```
docker run --name gitlab-redis -d sameersbn/redis:latest
```

启动gitlab：

```
docker run --name gitlab -d \
  --link gitlab-postgresql:postgresql --link gitlab-redis:redisio \
  --publish 10022:22 --publish 10080:80 \
  --env 'GITLAB_PORT=10080' --env 'GITLAB_SSH_PORT=10022' \
  --env 'GITLAB_SECRETS_DB_KEY_BASE=long-and-random-alpha-numeric-string' \
    sameersbn/gitlab:8.4.4
```

这三条Docker指令与安装WordPress的Docker指令和参数基本一样，唯一不同的是，传递的环境变量和映射的端口更多。从这里我们发现了一个特点：**Docker指令中的参数标示符可以重复使用，比如，如果传递多个环境变量，就连续使用多个“--env”。**

2. 测试 GitLab

上一节，我们已经搭建好了 GitLab 服务，接下来看看如何使用它。

首先通过 ifconfig 命令查看本机 IP，然后通过 http://192.168.10.103:10080 就可以访问到如图 3-10 所示的界面。

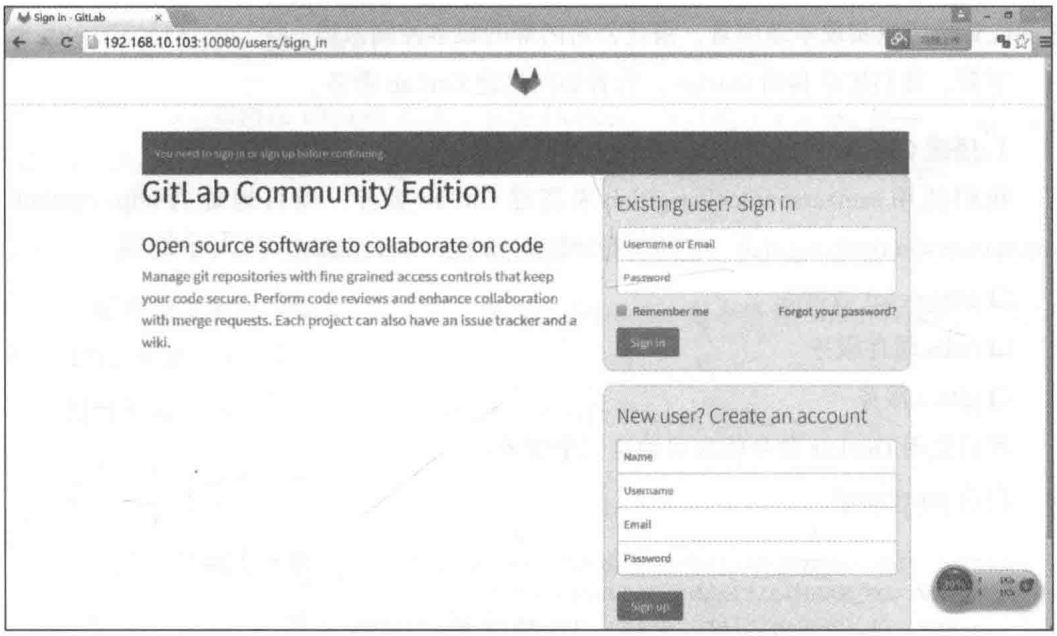


图 3-10 GitLab 的登录界面

系统默认的用户名：root，密码：5iveL!fe，在界面的右上侧，输入后就可以体验 gitlab 了。

我们创建了一个项目，就可以像 GitHub 那样使用了，界面如图 3-11 所示。

3.4.3 项目管理系统——Redmine

Redmine 是一套跨平台的项目管理系统，它通过“项目（Project）”的形式把成员、任务（问题）、文档、讨论及各种形式的资源组织在一起，大家参与更新任务、文档等内容来推动项目的进度，同时系统利用时间线索和各种动态的报表形式来自动给成员汇报项目进度。另外，它还集成了 Wiki 文档、版本控制、bug 跟踪等功能。Redmine 是项目管理不可或缺的好工具。

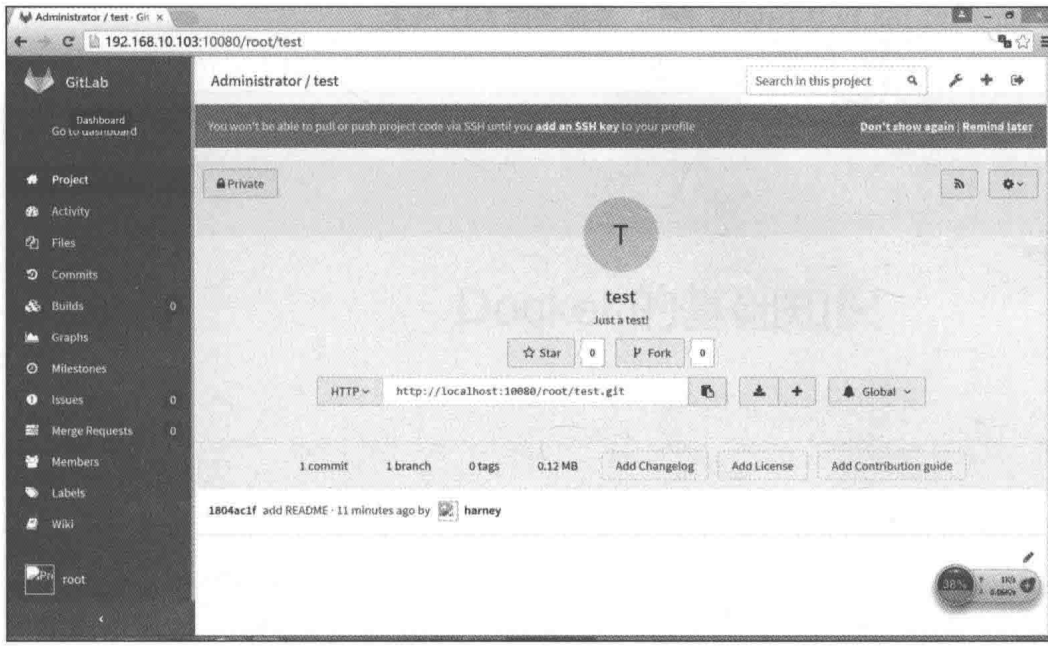


图 3-11　GitLab 项目界面

1. 搭建 Redmine 服务

搭建 Redmine 服务，我们使用 sameersbn/redmine 镜像，项目地址为 https://github.com/sameersbn/docker-redmine。

两条 Docker 指令就可以搞定。

第一条指令：

```
docker run --name=postgresql-redmine -d \
  --env='DB_NAME=redmine_production' \
  --env='DB_USER=redmine' --env='DB_PASS=password' \
  sameersbn/postgresql:9.4-12
```

第二条指令：

```
docker run --name=redmine -d \
  --link=postgresql-redmine:postgresql --publish=10083:80 \
  --env='REDMINE_PORT=10083' \
  sameersbn/redmine:3.2.0-4
```

2. 测试 Redmine

Docker 指令中，我们把 Redmine 的对外服务端口映射到 10083，所以我们可以通

过 http://192.168.10.103:10083 访问，界面如图 3-12 所示。

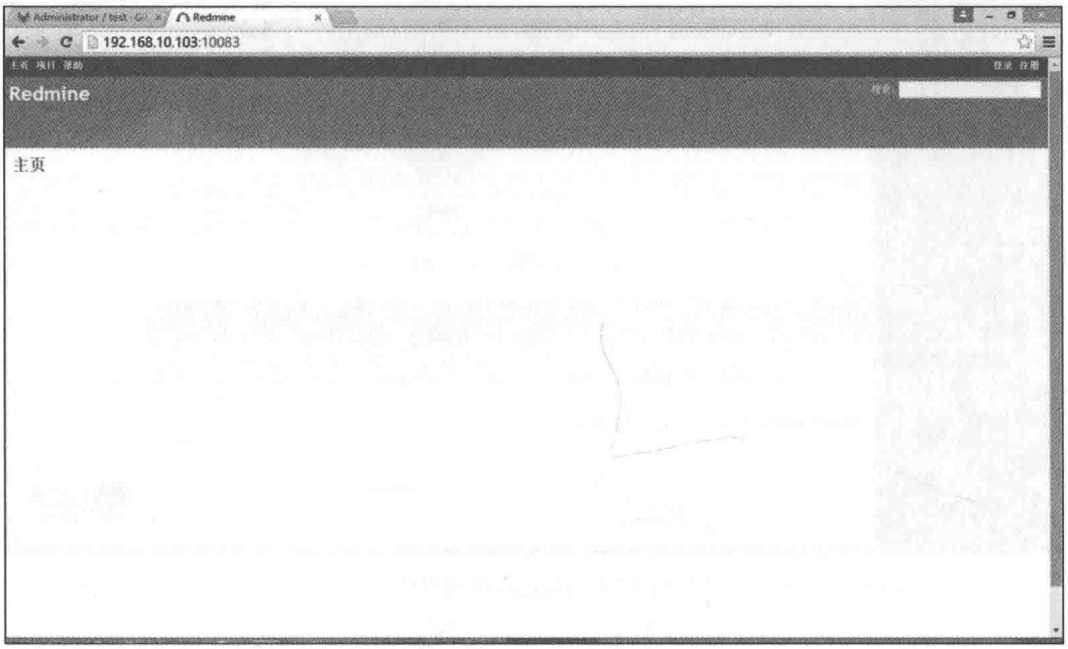


图 3-12 Redmine 登录界面

可以输入系统默认用户（用户名：admin，密码：admin）进行深入体验。

3.5 本章小结

本章我们先介绍了主流操作系统对 Docker 的支持情况，以及如何安装 Ubuntu 系统和在 Ubuntu 环境下安装 Docker。最后我们举了三个应用例子，介绍 Docker 如何部署服务，后续的章节，我们会继续结合这三个应用例子，深入探讨 Docker 的使用方法。

第 4 章 Chapter 4

Docker 的基础知识

上一章我们安装了 Ubuntu 系统并且举了三个例子讲解了如何使用 Docker 来安装应用，非常简单方便。那么它是如何做到的呢？这一章我们就深入 Docker 的内部，来了解它的运行原理。

4.1 Docker 的基本概念和常用操作指令

上一章，我们通过 docker run 指令创建并启动了三个 Docker 应用。Docker 提供了 docker ps 命令来查看相关的进程。

```
$ docker ps |awk '{print $2, $NF}'
IMAGE NAMES
sameersbn/redmine:3.2.0-4 redmine
sameersbn/postgresql:9.4-12 postgresql-redmine
sameersbn/gitlab:8.4.4 gitlab
wordpress MyWordPress
sameersbn/redis:latest gitlab-redis
sameersbn/postgresql:9.4-12 gitlab-postgresql
mariadb db
```

docker ps 指令输出多项信息，我们只列出 IMAGE 和 NAMES 两列。在搭建 WordPress 时使用的指令：

```
$ docker run --name db --env MYSQL_ROOT_PASSWORD=example -d mariadb
$ docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

可以看出，docker run 命令的最后一列是 mariadb 和 wordpress，--name 的参数是 db 和 MyWordPress，分别对应 docker ps 的 IMAGE 和 NAMES。

那么 IMAGE 和 NAMES 代表什么含义呢？其实它们分别对应 Docker 的两个重要概念：镜像和容器。

下面我们来了解一下 Docker 的几个基本概念。

4.1.1 Docker 三大基础组件

Docker 有三个重要的概念：仓库（Repository）、镜像（Image）和容器（Container），它们是 Docker 的三大基础组件，如图 4-1 所示。

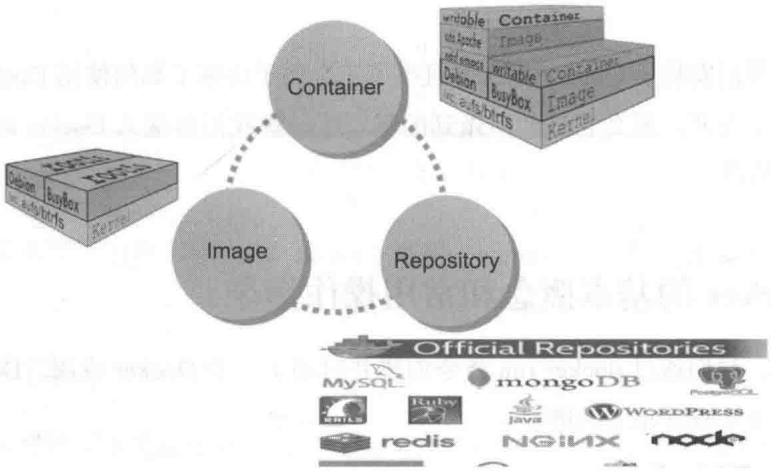


图 4-1 三大基础组件

首先，Docker 官方给用户提供一个官方的 Docker 仓库，它就像 iPhone 手机的 App 应用商店，里面存放着各种各样已经打包好的 Docker 应用——Docker 镜像。

其次，用户就可以搜索自己需要的镜像，下载到本地。Docker 镜像是为了满足特殊用途而按照 Docker 的规则制作的应用，有点儿类似于 Windows 里面的安装软件包。

最后，用户就可以利用 Docker 镜像创建 Docker 容器，容器会启动预先定义好的进程与用户交互，对外提供服务。容器都是基于镜像而创建的，基于一个镜像可以创建若干个名字不同但功能相同的容器。

了解了 Docker 的三大基础组件，再来看看这条指令：

```
docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

这条指令做了如下事情：

先在本机查找有没有 wordpress 镜像，如果没有，就到 Docker 的仓库查找该镜像，然后下载到本机。

基于 wordpress 镜像创建容器 MyWordPress，提供个人博客服务。

所以，我们通过 docker ps 可以查到名字（Name）是 MyWordPress，所使用的镜像是 WordPress 的容器。

4.1.2　常用的 Docker 指令

前面我们已经接触过两个 Docker 指令，docker run 和 docker ps。这一节，我们就从整体上系统地介绍一下 Docker 指令。

我们在命令行终端输入 docker，就可以看到 Docker 的指令用法及支持的指令。

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

A self-sufficient runtime for containers.

Options:

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

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
```

```
  ......
    wait        Block until a container stops, then print its exit code

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

Docker 指令的基本用法：

```
docker + 命令关键字 (COMMAND) + 一系列的参数 ([arg...])
```

比如，对于下面的指令来说，run 是命令关键字，后面的内容都是参数。

```
docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

如果我们不了解某个命令关键字支持哪些参数，可以通过下面指令获取帮助。

```
docker COMMAND --help
```

比如，如果想了解 docker run 的用法，使用如下命令：

```
$ docker run --help

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

  -a, --attach=[]              Attach to STDIN, STDOUT or STDERR
  --add-host=[]                Add a custom host-to-IP mapping (host:ip)
  --blkio-weight               Block IO (relative weight), between 10 and 1000
  --blkio-weight-device=[]     Block IO weight (relative device weight)
  --cpu-shares                 CPU shares (relative weight)
  --cap-add=[]                 Add Linux capabilities
  --cap-drop=[]                Drop Linux capabilities
  --cgroup-parent              Optional parent cgroup for the container
  --cidfile                    Write the container ID to the file
......
```

截止到 Docker 的 V1.10.1 版本，Docker 一共支持 51 条指令，操作对象主要针对四个方面：

- 针对守护进程的系统资源设置和全局信息的获取。比如：docker info、docker deamon 等。
- 针对 Docker 仓库的查询、下载操作。比如：docker search、docker pull。
- 针对 Docker 镜像的查询、创建、删除操作。比如：docker images、docker build。
- 针对 Docker 容器的查询、创建、开启、停止操作。比如：docker ps、docker run。

Docker 指令除了单条使用外，还支持赋值、解析变量、嵌套等使用。

比如：

例 1：获取容器的 ID，并根据 ID 提交到仓库。用到了赋值、解析变量功能。

```
$ ID=$(docker run -d ubuntu echo hello world)
hello world
$ docker commit $ID helloworld
fd08a884dc79
```

例 2：删除所有停止运行的容器（使用需谨慎！），用到了 Docker 指令嵌套功能。

```
docker rm $(docker ps -a -q)
```

4.1.3　Docker 的组织结构

从计算机整个软件层面来看，从操作系统层到应用软件层，Docker 到底处于什么位置呢？

通过图 4-2，我们可以看出，Docker 位于操作系统和虚拟容器（lxc 或 libcontainer）之上。它会通过调用 cgropu、namesapces 和 libcontainer 等系统层面的接口来完成资源分配与相互隔离。

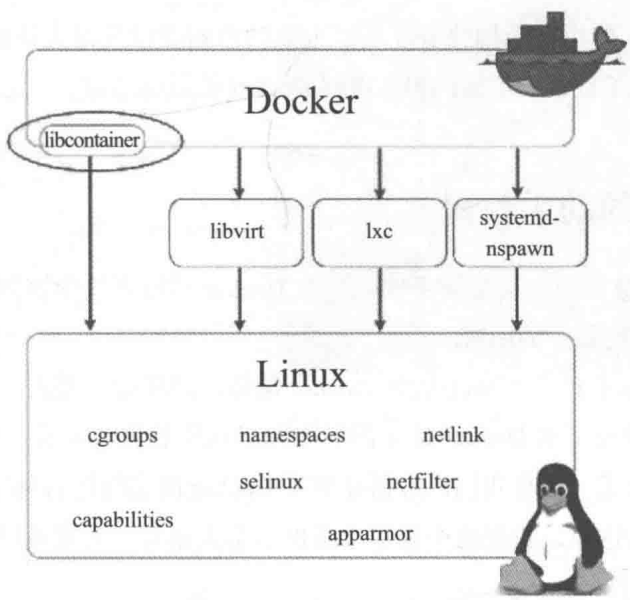


图 4-2　Docker 在 Linux 系统中的位置

我们再通过图 4-3 看看 Docker 内部的组织结构。

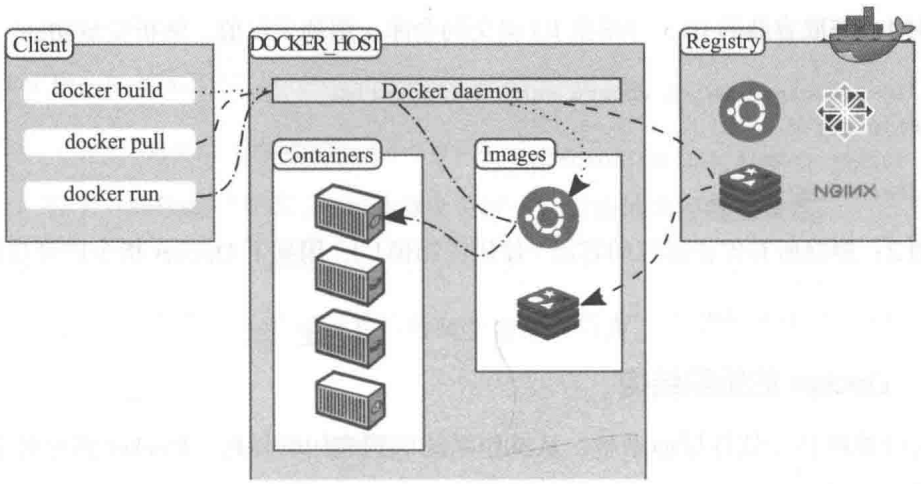


图 4-3 Docker 内部的组织结构

在一台主机上，首先要启动一个守护进程（Docker Daemon），所有的容器都被守护进程控制，同时守护进程监听并接收 Docker 客户端（Docker Client）指令，并把执行结果返回给 Docker 客户端。

其实，Docker 的组织结构比较复杂，上面的两幅图仅仅是大体的描述，省略了很多细节，这里是为了更有助于我们初步理解 Docker 的组织结构。

4.2 10 分钟的动手教程

理解 Docker 最好的方法是动手实践，在 Docker 运行环境中直观地体验 Docker 的各类基本操作和常用指令的用法。

Docker 官方原来有一个 Web 版的 Docker 模拟运行环境，提供一个 10 分钟的动手教程，让初学者快速了解 Docker 是如何工作的。这个教程非常棒，但不知什么原因，这个教程后来撤掉了。好在我们已经搭建好了 Docker 的实际运行环境，不需要官方的 Web 模拟环境。在这里，我把这个教程重新展示给大家看，大家可以在 Docker 的实际运行环境下跟着操作。

这个教程的主要内容为：

- 列出 Docker 的版本号。
- 在 Docker 的官方镜像仓库，搜索别人已经制作好的 Docker 镜像。
- 下载镜像，并以这个镜像为模板，在 Docker 容器中运行一个 shell 命令，输出“hello world”。
- 在 Docker 容器中安装 ping 软件包，把它提交为新镜像。
- 基于安装有 ping 软件的新镜像为模板，在 Docker 容器中测试 ping 命令工作是否正常。
- 如果测试 ping 命令工作正常，说明安装有 ping 软件的镜像制作正确，然后，我们就把这个新镜像提交到 Docker 官方镜像仓库，分享给大家使用。

首先进入教程的引导页面，如图 4-4 所示。

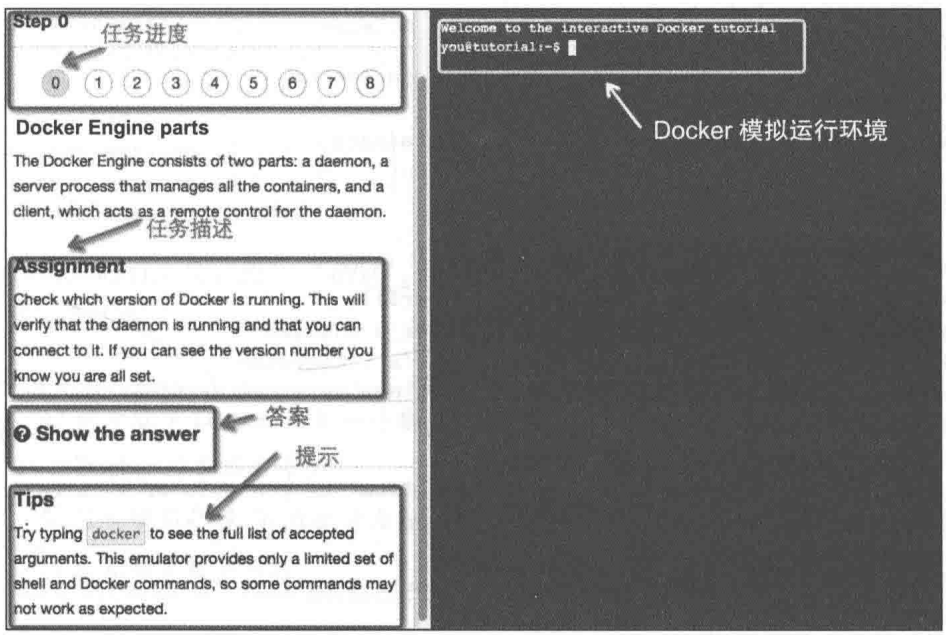


图 4-4　教程的页面布局

这个教程做得很棒，在网页的左侧，看完任务描述（Assignment），如果不知道如何操作的话，可以根据下面的提示（Tips）来做，如果看完提示还是不知道如何操作，还可以单击“show the answer”前的“?”，显示答案。在网页右侧是 Docker 的模拟运行环境，用来验证我们的操作是否可以完成网页左侧指定的任务。如果完成任务，在网页

的右上角会有“NEXT”按钮，单击进入下一个环节。

好的，接下来，我们就跟着网站的引导，一步一步地学习这个教程吧。

步骤0：查询Docker的版本号

Docker的引擎有两部分组成：Daemon和Client。Daemon是Service端的守护进程，接收Client端指令，管理本机上所有的镜像和容器；Client是通过Docker命令和Daemon交互，对Docker的镜像和容器进行查询、添加、修改、启动、停止等操作。

任务：找出Client是通过哪个Docker命令查询Docker的版本号。

提示：在命令行输入docker列出Docker支持的所有命令关键字。然后再找出哪个命令可以输出版本号。

在右侧Docker的模拟环境中，我们执行Docker命令，显示如下内容。

```
you@tutorial:~$ docker
Usage: Docker [OPTIONS] COMMAND [arg...]
-H="127.0.0.1:4243": Host:port to bind/connect to

A self-sufficient runtime for linux containers.

Commands:

attach    Attach to a running container
build     Build a container from a Dockerfile
commitCreate a new image from a container's changes
diff      Inspect changes on a container's filesystem
export    Stream the contents of a container as a tar archive
history   Show the history of an image
images    List images
import    Create a new filesystem image from the contents of a tarball
info      Display system-wide information
insertInsert a file in an image
inspect   Return low-level information on a container
killKill a running container
login     Register or Login to the Docker registry server
logsFetch the logsof a container
port      Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
ps        List containers
pull      Pull an image or a repository from the Docker registry server
push      Push an image or a repository to the Docker registry server
restart   Restart a running container
rm        Remove a container
rmi       Remove an image
run       Run a command in a new container
```

```
search    Search for an image in the Docker index
startStart a stopped container
stopStop a running container
tag       Tag an image into a repository
versionShow the Docker version information
wait      Block until a container stops, then print its exit code
```

我们可以看到 Docker 命令的基本用法为：docker+ 选项 + 命令关键字 + 参数，其中，选项和参数是可选的。格式如下：

```
Docker[OPTIONS]COMMAND[arg...]
```

在 Docker 支持的命令关键字中，有 version 这个关键字，用来显示 Docker 的版本信息。这正是我们需要的命令。在网页右侧 Docker 的模拟运行环境输入 docker version，显示如下内容。

```
you@tutorial:~$ docker version
Docker Emulator version0.1.3

Emulating:
Client version: 0.5.3
Server version: 0.5.3
Go version: go1.1
```

在网页的右上角，弹出“next”，说明我们已经成功完成这个任务，单击“next”进入下一个任务。

- ❑ 这个版本 Docker 只是一个模拟环境，仅仅为了配合本教程使用，并没有实现 Docker 的全部命令。
- ❑ 这个模拟环境 Docker 与命令关键字之间只允许有一个空格，两个或两个以上的空格识别不出来。

步骤 1：查询镜像

Docker 官方镜像仓库（Docker Hub Registry）储存着大量的 Docker 化的应用镜像，我们可以基于 Docker 官方仓库的镜像来创建我们的应用。Docker 支持通过 Client 端的命令来查询 Docker 官方仓库中镜像。

任务：用 Docker 的 Client 端命令查询一个名字叫“tutorial”的镜像。

提示：Docker 查询镜像的命令格式为 docker search <string>。

我们要从 Docker 官方镜像仓库查询一个名字叫“ tutorial”的镜像，在右侧 Docker 的模拟环境中，我们执行 docker search tutorial 命令，显示如下内容。

```
you@tutorial:~$ docker search tutorial
Found 1 results matching your query ("tutorial")
NAME DESCRIPTION
learn/tutorial          An image for the interactive tutorial
```

我们查到一个全名为“ learn/turorial”的 Docker 镜像。在 Docker 官方镜像仓库，镜像的全名都是如下格式：

```
<username>/<repository>
```

这是因为，每个用户都可以在 Docker 官方镜像仓库注册自己的账户，发布自己的 Docker 镜像，使用“用户名 + 镜像名”的命名方式，可以让不同用户拥有相同的镜像名而不相互干扰。

这时，在网页的右上角，弹出“next”，单击进入下一个任务。

步骤 2：下载镜像

在上一步，我们已经查询到“ learn/turorial”镜像。接下来我们就需要从 Docker 官方镜像仓库中下载这个镜像。Docker 提供 docker pull 命令来下载镜像。

任务：下载 tutorial 镜像。

提示：下载时不要忘记使用镜像的全名，如“ learn/turorial”。

在右侧 Docker 的模拟环境中，我们执行 docker pull learn/turorial 命令，显示如下内容。

```
you@tutorial:~$ docker pull learn/tutorial
Pulling repository learn/tutorial from https://index.docker.io/v1
Pulling image 8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c (prec
ise) from ubuntu
Pulling image b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc(12.1
0)from ubuntu
Pulling image 27cf784147099545()from tutorial
```

可以看到，Docker 的镜像正在被下载。这时，在网页的右上角，弹出“ next”，单击进入下一个任务。

步骤 3：创建并启动容器

我们下载了 Docker 镜像，就可以以 Docker 镜像为模板，启动容器。可以把容器理

解为在一个相对独立环境中运行一个（组）进程，这个独立环境拥有这个（组）进程运行所需要的一切，包括文件系统、库文件、shell 脚本等。

任务：运行下载的“learn/turorial”镜像，输出“hello world”。为了做到这点，需要在容器中运行 shell 命令“echo”，echo 的内容是“hello world”.

提示：docker run 命令用来创建和运行 Docker 容器。它至少需要两个参数，一个是镜像名，一个是在容器中需要运行的命令。

根据提示，我们明确了 docker run 的两个参数，镜像名为“learn/turorial”，在容器中需要运行的命令为 echo "hello world"。在右侧 Docker 的模拟环境中，我们执行 docker run learn/tutorial echo "hello world" 命令，显示如下内容。

```
you@tutorial:~$ docker run learn/tutorial echo "hello world"
hello world
```

可以看到，“hello world”正确地输出了。在网页的右上角，弹出“next”，单击进入下一个任务。

步骤 4：修改容器

接下来，我们要在容器中安装一个实用工具 ping，由于镜像是基于 Ubuntu 操作系统构建的，所以可以通过在容器中运行 apt-get install -y ping 来安装 ping 工具。一旦 ping 软件包安装完毕，容器会立刻停止运行，但容器中安装的软件包会一直保留。

任务：在基于“learn/tutorial”镜像的容器中安装 ping 软件包。

提示：在非交互模式下安装软件包，不要忘了使用“-y”。

在步骤 3 中，我们知道使用 docker run 可以创建容器，并在容器中运行指定的命令。在右侧的模拟环境，我们输入“docker run learn/tutorial apt-get install -y ping”，得到如下输出。

```
you@tutorial:~$ docker run learn/tutorial apt-get install -y ping
Reading package lists...
Building dependency tree...
The following NEW packages will be installed:
  iputils-ping
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 56.1 kB of archives.
After this operation, 143 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu/ precise/main iputils-ping amd64 3:20101006
-1ubuntu1 [56.1 kB]
debconf: delaying package configuration, since apt-utils is not installed
```

```
Fetched 56.1 kB in 1s (50.3 kB/s)
Selecting previously unselected package iputils-ping.
(Reading database ...7545 files and directories currently installed.)
Unpacking iputils-ping (from .../iputils-ping_3%3a20101006-1ubuntu1_amd64.deb) ...
Setting up iputils-ping (3:20101006-1ubuntu1) ...
```

在网页的右上角，有一个弹窗，告诉我们在 BaseImage 上已经安装了 ping 程序包，已经改变了文件系统，但还未保存。在弹窗最下方有“next”按钮，单击进入下一个任务。

步骤 5：创建新镜像

在上一步已经安装了 ping 程序包，你可能想保存这个变更，以便于以后启动容器时不需要重复安装 ping 程序包。Docker 支持在原有镜像基础上，只提交增量修改部分，形成一个新镜像。以后使用这个新镜像为模板启动容器，容器中就会存在 ping 软件包，不需要重复安装。

任务：首先用 dockerps -1 找到安装过 ping 包的容器的 ID 号，然后把这个容器提交为新镜像，镜像名为“learn/ping”。

提示：使用 docker commit 把容器提交为新镜像。

在右侧的模拟环境，我们首先输入 dockerps -1，显示本机上的所有容器，得到如下内容。

```
you@tutorial:~$ docker ps -l
ID IMAGE COMMAND CREATED
STATUS PORTS
6982a9948422  ubuntu:12.04  apt-get install ping  1 minute ago
Exit 0
```

可以看到，这个容器的 COMMAND 为“apt-get install ping”，这正是我们刚才安装过 ping 的容器，容器 ID 为 6982a9948422。有了容器 ID，我们就可以通过这条命令“docker commit 6982a9948422 learn/ping”，把容器提交为新镜像，在右侧的模拟环境，执行结果如下所示。

```
you@tutorial:~$ docker commit 6982a9948422 learn/ping
effb66b31edb
```

可以看到，执行结果是一个新 ID，这个 ID 就是新生成镜像的 ID。我们单击“next”进入下一个任务。

步骤 6：使用新镜像

我们基于容器生成了新的镜像，这个镜像包含 ping 软件包。然后这个新镜像就可

以运行在任何装有 Docker 引擎的机器上。

任务：在基于新镜像的容器中执行 ping www.docker.com 这条指令。

提示：新镜像要使用全名 learn/ping。

在右侧的模拟环境，执行 docker run learn/ping ping www.docker.com，执行结果如下所示。

```
you@tutorial:~$ docker run learn/ping ping www.google.com
PING www.google.com (74.125.239.129) 56(84) bytesof data.
64bytesfrom nuq05s02-in-f20.1e100.net (74.125.239.148): icmp_req=1 ttl=55time=2
.23 ms
64bytesfrom nuq05s02-in-f20.1e100.net (74.125.239.148): icmp_req=2 ttl=55time=2
.30 ms
64bytesfrom nuq05s02-in-f20.1e100.net (74.125.239.148): icmp_req=3 ttl=55time=2
.27 ms
64bytesfrom nuq05s02-in-f20.1e100.net (74.125.239.148): icmp_req=4 ttl=55time=2
.30 ms
```

可以看到，容器中已经存在 ping 命令，执行 ping www.google.com 可以得到预期的结果。使用 Ctrl-C 终止 ping 命令。单击“next”进入下一个任务。

步骤 7：查询容器信息

使用 docker ps 可以看到本机上所有正在运行的容器，使用 docker inspect 可以看到单个容器详细信息。

任务：找出正在运行容器的 ID 号，然后使用 docker inspect 查看容器的信息。

提示：可以使用容器 ID 来指定容器，也可以只使用容器 ID 的前 3 ~ 4 个字符来指定。

在右侧的模拟环境下，首先执行 docker ps 查看正在运行的容器，可以查到容器的 ID，内容如下：

```
you@tutorial:~$ docker ps
ID                  IMAGE               COMMAND                  CREATED
STATUS              PORTS
efefdc74a1d5        learn/ping:latest   ping www.google.com37 seconds ago
Up 36 seconds
```

根据提示，我们可以使用容器 ID 的前 3 ~ 4 个字符来指定容器。在模拟环境中，执行 docker inspect efe，可以得到如下结果。

```
you@tutorial:~$ docker inspect efe
```

```
[2013/07/3001:52:26 GET /v1.3/containers/efef/json
{
"ID": "efefdc74a1d5900d7d7a74740e5261c09f5f42b6dae58ded6a1fde1cde7f4ac5",
"Created": "2013-07-30T00:54:12.417119736Z",
"Path": "ping",
"Args": [
"www.google.com"
  ],
"Config": {
"Hostname": "efefdc74a1d5",
"User": "",
"Memory": 0,
"MemorySwap": 0,
"CpuShares": 0,
"AttachStdin": false,
"AttachStdout": true,
"AttachStderr": true,
"PortSpecs": null,
"Tty": false,
"OpenStdin": false,
"StdinOnce": false,
"Env": null,
"Cmd": [
"ping",
"www.google.com"
    ],
"Dns": null,
"Image": "learn/ping",
"Volumes": null,
"VolumesFrom": "",
"Entrypoint": null
  },
"State": {
"Running": true,
"Pid": 22249,
"ExitCode": 0,
"StartedAt": "2013-07-30T00:54:12.424817715Z",
"Ghost": false
  },
"Image": "a1dbb48ce764c6651f5af98b46ed052a5f751233d731b645a6c57f91a4cb7158",
"NetworkSettings": {
"IPAddress": "172.16.42.6",
"IPPrefixLen": 24,
"Gateway": "172.16.42.1",
"Bridge": "docker0",
"PortMapping": {
```

```
    "Tcp": {},
    "Udp": {}
        }
      },
    "SysInitPath": "/usr/bin/docker",
    "ResolvConfPath": "/etc/resolv.conf",
    "Volumes": {},
    "VolumesRW": {}
```

至此，可以看到容器的完整 ID、运行状态、网络设置、镜像等信息。单击“next”进入下一个任务。

步骤 8：把新镜像上传仓库

在步骤 6 和步骤 7 中，我们已经验证了新构建的镜像 learn/ping 可以正常工作，现在我们想把这个新镜像分享给别人使用，该如何做呢？还记得我们最初是从 Docker 的官方镜像仓库下载了 learn/tutorial 就可以直接使用了吗？如果把新镜像 learn/ping 上传到 Docker 官方仓库，就可以像 learn/tutorial 一样供别人下载后直接使用。

任务：把新镜像 learn/ping 推送到 Docker 官方镜像仓库。

提示：docker images 可以显示当前主机上所有的镜像。docker push 可以推送本机的镜像到 Docker 官方仓库。这个模拟器已经以 learn 这个用户登录了，所以我们只能把镜像推送到 learn 这个名字空间下。

在右侧的模拟环境中，先执行 docker images 查看本机的镜像列表，可以得到如下内容。

```
you@tutorial:~$ docker images
ubuntu                  latest              8dbd9e392a96        4 months ago
131.5 MB (virtual131.5 MB)
learn/tutorial          latest              8dbd9e392a96        2 months ago
131.5 MB (virtual131.5 MB)
learn/ping              latest              effb66b31edb        10 minutes
ago
11.57 MB (virtual143.1 MB)
```

其中，learn/ping 是我们新构建的镜像，执行 docker push learn/ping 把镜像推送到 Docker 官方仓库。

这就是整个教程。通过这个教程，我们可以快速掌握 Docker 的基本命令。

步骤 8，在模拟环境中已经以 learn 这个用户登录了，可以直接进行上传操作。

但我们自己搭建的Docker运行环境并没有登录，所以上传会失败。在https://hub.docker.com/account/signup/上注册一个Docker Hub的账号，使用dokcer login登录就可以做上传操作了。不过后续还会详细讲Docker Hub，步骤8可以先跳过。

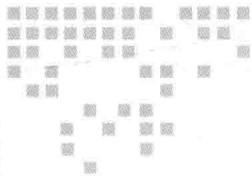
4.3 本章小结

本章主要介绍了Docker的基本概念和组织结构，以及Docker指令的格式和用法，最后，通过一个动手练习的教程，加深了大家对Docker指令用法的理解。

第二部分 Part 2

高 级 篇

- 第 5 章　Docker 容器管理
- 第 6 章　Docker 镜像管理
- 第 7 章　Docker 仓库管理
- 第 8 章　Docker 网络和存储管理
- 第 9 章　Docker 项目日常维护
- 第 10 章　Docker Swarm 容器集群
- 第 11 章　Docker 插件开发



Chapter 5 第 5 章

Docker 容器管理

上一章我们学习了 Docker 命令的基本用法，这一章，我们就结合第 3 章的例子来讲一下 Docker 容器的管理。

5.1 单一容器管理

我们在第 3 章启动了三个服务项目：WordPress（个人博客）、GitLab（版本控制）和 Redmine（项目流程管理）。通过 docker ps 可以看到已经启动 Docker 容器，可以看到每个容器的 ID 号、所使用的 Docker 镜像、创建时间、当前状态、监听的端口和容器的名字等，如图 5-1 所示。

```
harney@UShenzhou:~$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS                                                 NAMES
e8d028d58093        sameersbn/redmine:3.2.0-4      "/sbin/entrypoint.sh "   2 days ago          Up 2 days           443/tcp, 0.0.0.0:10083->80/tcp                        redmine
daedaff5a171        sameersbn/postgresql:9.4-12    "/sbin/entrypoint.sh"    2 days ago          Up 2 days           5432/tcp                                              postgresql-redmine
65c159dcc692        sameersbn/gitlab:8.4.4         "/sbin/entrypoint.sh "   2 days ago          Up 2 days           443/tcp, 0.0.0.0:10022->22/tcp, 0.0.0.0:10080->80/tcp gitlab
0ee24103a5cf        wordpress                      "/entrypoint.sh apach"   2 days ago          Up 4 hours          0.0.0.0:8080->80/tcp                                  MyWordPress
a281ac0deae4        sameersbn/redis:latest         "/sbin/entrypoint.sh"    2 days ago          Up 2 days           6379/tcp                                              gitlab-redis
85e9f3b71b70        sameersbn/postgresql:9.4-12    "/sbin/entrypoint.sh"    2 days ago          Up 2 days           5432/tcp                                              gitlab-postgresql
cc183cfecbc5        mariadb                        "/docker-entrypoint.s"   2 days ago          Up 2 days           3306/tcp                                              db
```

图 5-1　容器列表

还记得 WordPress 启动的两条指令吗？

```
$ docker run --name db --env MYSQL_ROOT_PASSWORD=example -d mariadb
$ docker run --name MyWordPress --link db:mysql -p 8080:80 -d wordpress
```

通过 --name 参数创建了两个 Docker 容器：db 和 MyWordPress，在图 5-1 中可

以很快找到这两个容器和查到它们的运行状态。另外，可以看到每个容器都有一个 CONTAINER ID。

5.1.1　容器的标示符

每个容器被创建后，都会分配一个 CONTAINER ID 作为容器的唯一标示，后续对容器的启动、停止、修改和删除等所有操作，都是通过 CONTAINER ID 来完成的，CONTAINER ID 有点儿像数据库的主键。CONTAINER ID 默认是 128 位，但对于大多数主机来说，ID 的前 16 位就足以保证其在本机的唯一性。所以，默认情况下我们使用 CONTAINER ID 简略形式即可（ID 的前 16 位）。使用 docker ps 可以查到 CONTAINER ID 简略形式，如果需要查询完整的 CONTAINER ID，使用 docker ps --no-trunc。

CONTAINER ID 简略形式如下：

```
0ee24103a5cf
```

CONTAINER ID 完整形式如下：

```
0ee24103a5cf7d4a703edfc148c10a7db84156ca6008b53c2c22d4901b28bf61
```

有了 CONTAINER ID，我们就可以通过 Docker 的相关指令启动和停止容器了。

比如：对于 CONTAINER ID 为 0ee24103a5cf 的容器，通过下面命令查到容器的状态是“Up 2 days”，说明容器正处于运行阶段：

```
docker ps -a |grep 0ee24103a5cf
```

通过下面命令来停止容器运行，再次查看容器状态，变为“Exited (0) 2 seconds ago”，说明容器已停止运行。

```
docker stop 0ee24103a5cf
```

如果再次把容器启动，容器状态就变更为“Up 22 seconds”，说明容器又启动起来了。

```
docker start 0ee24103a5cf
```

CONTAINER ID 虽然能保证唯一性，但很难记忆。在创建容器时，可以同 --name 参数给容器起一个别名，如 MyWordPress，然后通过别名来代替 CONTAINER ID 对容器进行操作。比如

```
docker start MyWordPress
```

5.1.2 查询容器信息

通过 docker inspcet 命令可以查询容器的所有基本信息，包括运行情况、存储位置、配置参数、网络设置等。

```
Usage: docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]
$ docker inspect MyWordPress
[
  {
"Id": "0ee24103a5cf7d4a703edfc148c10a7db84156ca6008b53c2c22d4901b28bf61",
"Created": "2016-02-14T03:00:34.079375628Z",
"Path": "/entrypoint.sh",
"Args": [
"apache2-foreground"
    ],
"State": {
"Status": "running",
"Running": true,
"Paused": false,
"Restarting": false,
"OOMKilled": false,
"Dead": false,
"Pid": 29964,
"ExitCode": 0,
"Error": "",
"StartedAt": "2016-02-16T03:23:00.1351266Z",
"FinishedAt": "2016-02-16T03:07:04.359397299Z"
    },
"Image": "sha256:313affca71d71838d0bc0fa32d3f582728ae510865a797441a2bc95b001622f2",
"ResolvConfPath": "/var/lib/docker/containers/0ee24103a5cf7d4a703edfc148c1
0a7db84156ca6008b53c2c22d4901b28bf61/resolv.conf",
"HostnamePath": "/var/lib/docker/containers/0ee24103a5cf7d4a703edfc148c10a
7db84156ca6008b53c2c22d4901b28bf61/hostname",
"HostsPath": "/var/lib/docker/containers/0ee24103a5cf7d4a703edfc148c10a7db
84156ca6008b53c2c22d4901b28bf61/hosts",
...
```

docker inspcet 以 JSON 的格式展示非常丰富的信息，通过“-f”可以使用 Golang 的模板来提取指定部分的信息。

比如提取容器的运行状态：

```
$ docker inspect -f {{.State.Status}} MyWordPress
running
```

提取容器的 IP 地址：

```
$ docker inspect -f {{.NetworkSettings.IPAddress}} MyWordPress
172.17.0.5
```

除了容器的基本信息外，容器的日志也是我们经常需要查看的。使用docker logs来查询日志。

```
$ docker logs MyWordPress
WordPress not found in /var/www/html - copying now...
Complete! WordPress has been successfully copied to /var/www/html
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.5. Set the 'ServerName' directive globally to suppress this message
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.5. Set the 'ServerName' directive globally to suppress this message
[Sun Feb 14 03:00:37.453842 2016] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.10 (Debian) PHP/5.6.18 configured -- resuming normal operations
[Sun Feb 14 03:00:37.453863 2016] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'
192.168.10.102 - - [14/Feb/2016:03:00:58 +0000] "GET / HTTP/1.1" 302 413 "-""Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.3; WOW64; Trident/7.0; .NET4.0E; .NET4.0C; InfoPath.2)"
192.168.10.102 - - [14/Feb/2016:03:00:58 +0000] "GET /wp-admin/install.php HTTP/1.1" 200 3669 "-""Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.3; WOW64; Trident/7.0; .NET4.0E; .NET4.0C; InfoPath.2)"
```

如果需要实时打印最新的日志，可以加上“-f”参数。

另外，我们还可以通过docker stats命令实时查看容器所占用的系统资源，如CPU使用率、内存、网络和磁盘开销。

```
$ docker stats  MyWordPress
CONTAINER     CPU %   MEM USAGE / LIMIT    MEM %    NET I/O              BLOCK I/O
MyWordPress 0.00%  17.94 MB / 4.061 GB  0.44%   4.966 kB / 1.426 kB  0 B / 16.38 kB
```

5.1.3 容器内部命令

经常有登入Docker容器内部执行命令的需求，可以在容器中启动sshd服务来响应用户登录。但sshd方式存在进程开销和增加被攻击的风险，同时也违反Docker所倡导的“一个容器一个进程”的原则。

Docker提供了原生的方式支持登入容器docker exec，使用形式如下：

```
docker exec+ 容器名 + 容器内执行的命令
```

比如要查看MyWordPress容器内启动了哪些进程，执行的命令和结果如下：

```
$ docker exec MyWordPress ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
root         1  0.0  0.7 313684 31208 ?     Ss   Feb16   0:04 apache2 -DFOREGROUND
www-data    59  0.0  0.1 313716  7824 ?     S    Feb16   0:00 apache2 -DFOREGROUND
www-data    60  0.0  0.1 313716  7824 ?     S    Feb16   0:00 apache2 -DFOREGROUND
www-data    61  0.0  0.1 313716  7824 ?     S    Feb16   0:00 apache2 -DFOREGROUND
www-data    62  0.0  0.1 313716  7824 ?     S    Feb16   0:00 apache2 -DFOREGROUND
www-data    63  0.0  0.1 313716  7824 ?     S    Feb16   0:00 apache2 -DFOREGROUND
root       164  0.0  0.0  17500  2064 ?     Rs   07:18   0:00 ps aux
```

如果希望在容器内连续执行多条命令，可以加上“-it”参数，就相当于以 root 身份登入容器内，可以连续执行命令，执行完成后通过“exit”退出。

```
harney@UShenzhou:~$ docker exec -it MyWordPress /bin/bash
root@0ee24103a5cf:/var/www/html# pwd
/var/www/html
root@0ee24103a5cf:/var/www/html# ls
index.php    readme.html      wp-admin            wp-comments-post.php
wp-config.php  wp-cron.php  wp-links-opml.php  wp-login.php  wp-settings.php
wp-trackback.php
license.txt  wp-activate.php  wp-blog-header.php  wp-config-sample.php
wp-content      wp-includes  wp-load.php          wp-mail.php    wp-signup.php
xmlrpc.php
root@0ee24103a5cf:/var/www/html# exit
exit
harney@UShenzhou:~$
```

5.2 多容器管理

Docker 倡导的理念是“一个容器一个进程”，假如一个服务由多个进程组成，就需要创建多个容器组成一个系统，相互分工和配合来对外提供完整的服务。

比如，我们的博客系统由两部分组成：

- Apache Web 服务器，用于提供 Web 站点和与用户交互。
- Mariadb 数据库，用于存储用户注册信息、个性化配置和博客等数据。

我们通过两条 docker run 指令创建并启动了数据库容器（db）和 Apache 容器（MyWrodpress）。这两个容器之间需要有数据交互，在同一台主机下，docker run 命令提供“--link”选项建立容器间的互联。但有一个前提条件，使用“--link containerA”创建容器 B 时，容器 A 必须已经创建并且启动运行。所以容器启动是按顺序的，容器 A 先于容器 B 启动。

对于博客系统 WordPress，数据库容器（db）要先于 Apache 容器（MyWordpress）启动。所以，启动 WordPress 的方式应该是：

```
docker start db
docker start MyWordPress
```

如果停止 WordPress 服务，则需要先停止 Apache 容器（MyWordpress），再停止数据库容器（db），或同时停止这两个容器。

```
docker stop db MyWordPress
```

对于 GitLab 系统，它有三个容器，就要同时考虑三个容器的优先顺序，并按这个顺序启动。假如有更多容器，维护就会变得比较烦琐，有没有简洁的方式呢？下一节给出答案。

5.2.1　Docker Compose

Docker 提供一个容器编排工具——Docker Compose，它允许用户在一个模板（YAML 格式）中定义一组相关联的应用容器，这组容器会根据配置模板中的“--link”等参数，对启动的优先级自动排序，简单执行一条“docker-compose up”，就可以把同一个服务中的多个容器依次创建和启动。

Docker Compose 的安装方式如下：

```
sudo curl -L https://github.com/docker/compose/releases/download/1.6.0/
docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

我们看看如何使用 Docker Compose 来管理 WordPress 项目。

首先，我们把 WordPress 项目原有的两个容器停掉。

```
docker stop db MyWordPress
```

接着，创建一个项目文件夹 ~/wordpress，在文件夹下创建一个名字叫 docker-compose.yml 的文件，内容如下：

```
wordpress:
  image: wordpress
  links:
    - db:mysql
  ports:
    - 8080:80
```

```
db:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: example
```

这个配置文件中创建了两个容器 wordpress 和 db，使用 image 选项来指定两个容器分别使用 wordpress 和 mariadb 镜像，另外有选项 links、ports、environment 分别对应 docker run 中“--links”(容器互联)、“-p”(端口映射) 和“-e”(环境变量设置)。

然后就可以通过 docker-compose up 命令来创建和启动 WordPress 服务。

```
$ cd ~/wordpress &&docker-compose up
Creating wordpress_db_1
Creating wordpress_WordPress_1
Attaching to wordpress_db_1, wordpress_WordPress_1
WordPress_1 | WordPress not found in /var/www/html - copying now...
db_1        | Initializing database
db_1        | 2016-02-18 12:41:46 140467523413952 [Note] /usr/sbin/mysqld
(mysqld 10.1.11-MariaDB-1~jessie) starting as process 56 ...
db_1        | 2016-02-18 12:41:46 140467523413952 [Note] InnoDB: Using mutexes
to ref count buffer pool pages
```

通过另外一个命令行终端，输出 docker ps 查看容器是否启动。

```
CONTAINER ID        IMAGE                                    NAMES
0e2ca95e8672        wordpress       ...          wordpress_WordPress_1
d324e92ae9a5        mariadb         ...                    wordpress_db_1
```

可以看出，容器的 CONTAINER ID 和 NAME 都与原来的不同，说明是新创建了一组容器。容器已经启动起来，通过 http://ip:8080 可以正常访问页面。

后续对个人博客项目的启动和停止就变得非常简单了。

启动命令：

```
$ docker-compose start
Starting wordpress_db_1
Starting wordpress_WordPress_1
```

停止命令：

```
$ docker-compose stop
Stopping wordpress_WordPress_1 ... done
Stopping wordpress_db_1 ... done
```

可以看出，启动和停止的顺序已经被 docker-compose 智能管理起来了。启动时先

启动数据库容器（wordpress_db_1），再启动Apache容器（wordpress_WordPress_1）。停止时先停止Apache容器（wordpress_WordPress_1），再停止数据库容器（wordpress_db_1）。

虽然Docker Compose可以判断容器间的依赖并生成正确的启动顺序，但这种顺序仅仅是容器的顺序，假如容器A的进程a依赖容器B的进程b，但进程b启动需要耗费很长时间的话，这时虽然容器B先于容器A创建和启动，但进程a仍然可能和进程不能正常交互而启动失败，因为虽然容器B已启动但进程b还没完全启动完成。在这种情况下，Docker Compose无能为力，需要进程a自行增加一些判断等待和重试机制。

上面我们使用docker-compose up命令创建和启动一组新的容器来为WordPress服务。原来由docker run创建的容器如何处理呢？建议删除。

通过docker ps命令只能列出已经启动的容器，如果想查到已停止运行的容器，需要在docker ps命令后加上“-a”选项。

```
$ docker ps -a
CONTAINER ID      IMAGE           STATUS                    NAMES
0ee24103a5cf      wordpress     Exited (0) 12 hours ago       MyWordPress
cc183cfecbc5      mariadb     Exited (0) 12 hours ago          db
```

我们可以看到这两个容器的状态是Exited（退出），通过docker rm命令可以删除它们。删除前确保容器内没有重要数据。

```
docker rm MyWordPressdb
```

5.2.2 配置文件

使用Docker Compose管理多个容器，首先需要把多容器写到它的配置文件中，默认配置文件名为docker-compose.yml，我们可以通过“-f”选项指定配置文件。

下面我们再看看GitLab和Redmine项目的多容器写成Docker Compose配置文件的形式。

GitLab项目需要三个容器：postgresql、redis和gitlab。

postgresql容器创建和启动的命令为：

```
docker run --name gitlab-postgresql -d \
  --env 'DB_NAME=gitlabhq_production' \
  --env 'DB_USER=gitlab' --env 'DB_PASS=password' \
  sameersbn/postgresql:9.4-12
```

它使用 sameersbn/postgresql:9.4-12 镜像创建了一个名字为 gitlab-postgresql 的容器，并且设置了三个环境变量。转换为 Docker Compose 配置文件内容如下：

```
postgresql:
  image: sameersbn/postgresql:9.4-12
  environment:
    - DB_USER=gitlab
    - DB_PASS=password
    - DB_NAME=gitlabhq_production
```

redis 容器创建和启动的命令为：

```
docker run --name gitlab-redis -d sameersbn/redis:latest
```

使用 sameersbn/redis:latest 镜像创建一个名字为 gitlab-redis 的容器，转换为 Docker Compose 配置文件内容如下：

```
redis:
  image: sameersbn/redis:latest
```

gitlab 容器的创建和运行的命令为：

```
docker run --name gitlab -d \
  --link gitlab-postgresql:postgresql --link gitlab-redis:redisio \
  --publish 10022:22 --publish 10080:80 \
  --env 'GITLAB_PORT=10080' --env 'GITLAB_SSH_PORT=10022' \
  --env 'GITLAB_SECRETS_DB_KEY_BASE=long-and-random-alpha-numeric-string' \
    sameersbn/gitlab:8.4.4
```

当有多个环境变量需要设置时，用 docker run 命令需要多次重复“--env”选项，很烦琐。由于 Docker Compose 的配置文件使用 YAML 格式的语法，支持数组格式，所以这条命令转化为 Docker Compose 的配置文件就会很简洁，转换后内容如下：

```
gitlab:
  image: sameersbn/gitlab:8.4.4
  links:
    - redis:redisio
    - postgresql:postgresql
  ports:
    - "10080:80"
```

```
    - "10022:22"
  environment:
    - GITLAB_PORT=10080
    - GITLAB_SSH_PORT=10022
    - GITLAB_SECRETS_DB_KEY_BASE=long-and-random-alphanumeric-string
```

创建一个项目 ~/gitlab，把上面三部分的 Docker Compose 配置文件合并在一起，放到 ~/gitlab/docker-compose.yml。然后，通过下面的命令创建和启动 GitLab 服务。在创建新容器之前，先把原来的旧容器删除。

删除旧容器，使用“ -f”参数可以强制把正在运行的容器删除。删除前确保容器内没有重要数据，如果有重要数据，通过 docker exec -it ContainerName 命令登入容器内部处理，容器的数据备份在后续章节会详细讲解。

```
docker rm -f gitlab gitlab-redis gitlab-postgresql
```

启动新容器组。

```
$ cd ~/gitlab/ && docker-compose up -d
Creating gitlab_redis_1
Creating gitlab_postgresql_1
Creating gitlab_gitlab_1
```

通过 docker ps 可以查到 gitlab_gitlab_1、gitlab_postgresql_1 和 gitlab_redis_1 这三个容器，通过 http://ip:10080 可以访问。

对于 Redmine 项目的改造步骤如下：

首先，删除旧的容器，确保容器内没有重要数据。

```
docker rm -f redmine  postgresql-redmine
```

接着，把 docker run 创建容器的指令改造为 Docker Compose 的配置文件。

```
docker run --name=postgresql-redmine -d \
  --env='DB_NAME=redmine_production' \
  --env='DB_USER=redmine' --env='DB_PASS=password' \
  sameersbn/postgresql:9.4-12

docker run --name=redmine -d \
  --link=postgresql-redmine:postgresql --publish=10083:80 \
  --env='REDMINE_PORT=10083' \
  sameersbn/redmine:3.2.0-4
```

创建配置文件 ~/redmine/docker-compose.yml，内容如下：

```
postgresql:
  image: sameersbn/postgresql:9.4-12
  environment:
    - DB_NAME=redmine_production
    - DB_USER=redmine
    - DB_PASS=password

redmine:
  image: sameersbn/redmine:3.2.0-4
  links:
    - postgresql:postgresql
  ports:
    - "10083:80"
  environment:
    - REDMINE_PORT=10083
```

执行新容器组的创建和启动。

```
$ docker-compose up -d
Creating redmine_postgresql_1
Creating redmine_redmine_1
```

最后，通过 http://ip:10083 就可以访问网站。

好了，我们已经把个人博客（WordPress）、版本控制管理（GitLab）、项目流程管理（Redmine）转换为由 Docker Compose 来管理，可以很方便地把多个容器划为一个项目统一管理。一个项目一个配置文件，通过配置文件（用“-f”选项指定），就可以对该项目中的容器进行查询、启动、停止等操作。

查询 GitLab 项目的所有容器状态：

```
$ docker-compose  -f gitlab/docker-compose.yml ps
       Name                     Command               State                    Ports
gitlab_gitlab_1       /sbin/entrypoint.sh app:start   Up      0.0.0.0:10022->22/tcp,
443/tcp, 0.0.0.0:10080->80/tcp
gitlab_postgresql_1   /sbin/entrypoint.sh             Up      5432/tcp
gitlab_redis_1        /sbin/entrypoint.sh             Up      6379/tcp
```

停止 GitLab 项目：

```
$ docker-compose  -f gitlab/docker-compose.yml stop
Stopping gitlab_gitlab_1 ... done
Stopping gitlab_postgresql_1 ... done
Stopping gitlab_redis_1 ... done
```

启动 GitLab 项目：

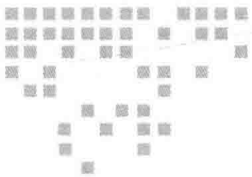
```
$ docker-compose  -f gitlab/docker-compose.yml start
Starting gitlab_redis_1
Starting gitlab_postgresql_1
Starting gitlab_gitlab_1
```

删除项目：

```
$ docker-compose  -f gitlab/docker-compose.yml down
Stopping gitlab_gitlab_1 ... done
Stopping gitlab_postgresql_1 ... done
Stopping gitlab_redis_1 ... done
Removing gitlab_gitlab_1 ... done
Removing gitlab_postgresql_1 ... done
Removing gitlab_redis_1 ... done
```

5.3　本章小结

本章主要介绍了容器管理的常用命令，主要包括容器状态的查询、创建、启动、停止、销毁和执行内部命令等。然后扩展到通过 Docker Compose 来管理多个容器。



Chapter 6 第 6 章

Docker 镜像管理

上一章我们介绍了如何对单个或多个容器进行管理，主要针对容器的状态查询、启动、停止等操作，没有涉及容器内部配置文件的修改和存储及容器的跨节点部署或迁移。主要是因为 Docker 有一个重要概念还没细讲，这就是 Docker 镜像。镜像是 Docker 的精髓，只有了解 Docker 镜像，才算真正理解 Docker 的内涵。

6.1 认识 Docker 镜像

我们创建容器时需要指定使用哪个镜像。比如，下面的命令就是使用镜像 sameersbn/redis:latest 创建容器，它先从本机查找有没有 sameersbn/redis:latest 镜像，如果不存在，就去官方的 Docker Hub 仓库查找并下载到本机，然后基于该镜像创建容器。

```
docker run --name gitlab-redis -d sameersbn/redis:latest
```

新容器创建后，不依赖镜像就可以运行。如果为了节省存储空间，可以手工删掉，默认不会自动删除，因为该镜像还有可能用于创建其他新镜像。通过 docker images 命令可以查到本机已有的所有镜像。

```
$ docker images -a
REPOSITORY              TAG         IMAGE ID        CREATED         SIZE
```

```
wordpress                latest     313affca71d7    8 days ago      517.3 MB
sameersbn/gitlab         8.4.4      9d1069e2b30c    8 days ago      720.5 MB
mariadb                  latest     1b6ea3e0ff8e    3 weeks ago     346.4 MB
sameersbn/redmine        3.2.0-4    7eb43870e9c7    4 weeks ago     636 MB
sameersbn/postgresql     9.4-12     a100f2a18ec3    4 weeks ago     231.3 MB
sameersbn/redis          latest     ad448e848573    4 weeks ago     196.5 MB
hello-world              latest     690ed74de00f    4 months ago    960 B
```

每个镜像都有唯一的标示 Image ID，这个和容器的 Container ID 一样，默认 128 位，可以使用前 16 位缩略形式，也可以使用镜像的名字（REPOSITORY）和版本号（TAG）两部分组合唯一标示。如果省略版本号，默认使用最新版本（latest）。

镜像的分层

在上一节，通过 docker images 我们看到每个镜像的大小（SIZE）都很大（几百兆），那么这些镜像所占磁盘的存储空间是否就是所有镜像大小之和吗？实际上，镜像所占的磁盘空间远远小于所有镜像之和，原因是 Docker 镜像采用分层机制，相同部分独立成层，只需要存储一份就可以了，大大节省了镜像空间。比如，wordpress 和 mariadb 都是基于 Ubuntu14.04 系统构建的，那么只需要一个 Ubuntu14.04 的镜像分层，在此基础上再根据 wordpress 和 mariadb 各自不同部分构建各自的独立分层，如图 6-1 所示。

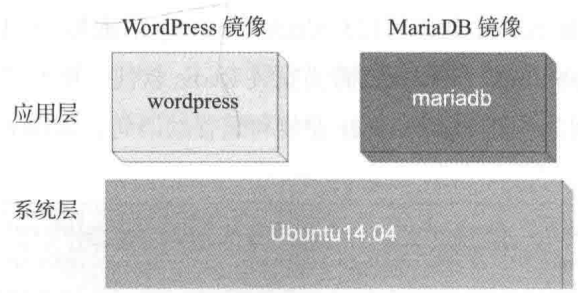


图 6-1　镜像的分层图

Docker 的镜像通过联合文件系统（union filesystem）将各层文件系统叠加在一起，在用户看来就像一个完整的文件系统。假如，某个镜像有两层，第一层有三个文件夹，第二层有两个文件夹，使用联合文件系统叠加后，用户可以看到五个文件夹，感觉不到分层的存在，如图 6-2 所示。

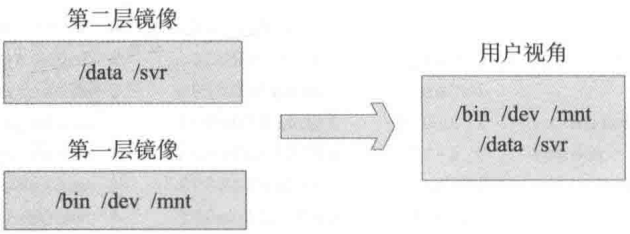


图 6-2 用户的视角看分层文件系统

通过 docker history 命令可以查询镜像分了多少层，每一层具体做了什么操作，如图 6-3 所示。

```
harney@UShenzhou:~$ docker history sameersbn/redis
IMAGE               CREATED             CREATED BY                                      SIZE
ad448e848573        4 weeks ago         /bin/sh -c #(nop) ENTRYPOINT &{["/sbin/entryp   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop) VOLUME [/var/lib/redis]       0 B
<missing>           4 weeks ago         /bin/sh -c #(nop) EXPOSE 6379/tcp               0 B
<missing>           4 weeks ago         /bin/sh -c chmod 755 /sbin/entrypoint.sh        1.48 kB
<missing>           4 weeks ago         /bin/sh -c #(nop) COPY file:fbcf0f32514d052d3   1.48 kB
<missing>           4 weeks ago         /bin/sh -c apt-get update  && DEBIAN_FRONTEND   2.118 MB
<missing>           4 weeks ago         /bin/sh -c #(nop) ENV REDIS_USER=redis REDIS_   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop) MAINTAINER sameer@damagehea   0 B
<missing>           4 weeks ago         /bin/sh -c echo 'APT::Install-Recommends 0;'    6.443 MB
<missing>           4 weeks ago         /bin/sh -c #(nop) MAINTAINER sameer@damagehea   0 B
<missing>           4 weeks ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>           4 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
<missing>           4 weeks ago         /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
<missing>           4 weeks ago         /bin/sh -c #(nop) ADD file:7ce20ce3daa6af21db   187.7 MB
```

图 6-3 镜像各分层具体操作

可以看到 sameersbn/redis 镜像有 14 层，每一层做的操作可以从 CREATED BY 看到。如果操作的内容显示不完全，可以在 docker history 后面加“ --tree”选项打印出完整的内容。像 sameersbn/redis 镜像肯定需要安装 Redis 软件，加上“--tree”选项后我们从 CREATED BY 列可以查到 redis-server 安装和配置的语句，如图 6-4 所示。

```
<missing>                                                   4 weeks ago         /bin/sh -c apt-get update  && DEBIAN_F
RONTEND=noninteractive apt-get install -y redis-server  && sed 's/^daemonize yes/daemonize no/' -i /etc/redis/redis.conf  && sed 's/
^bind 127.0.0.1/bind 0.0.0.0/' -i /etc/redis/redis.conf  && sed 's/^# unixsocket /unixsocket /' -i /etc/redis/redis.conf  && sed 's/
^# unixsocketperm 755/unixsocketperm 777/' -i /etc/redis/redis.conf  && sed '/^logfile/d' -i /etc/redis/redis.conf  && rm -rf /var/l
ib/apt/lists/*
```

图 6-4 镜像中软件安装和配置的命令

对于分层的 Docker 镜像有两个特性：一个是已有的分层只能读不能修改，另外一个是上层镜像的优先级高于底层镜像。

这里举一个例子来解释下原因，镜像 B 和镜像 C 都是在镜像 A 的基础上搭建起来的，镜像 A 有一个文件 a.txt，内容为“ hello world”。从用户的视角，镜像 B 和镜像 C

都可以看到文件 a.txt，内容都是“ hello world”。这时，镜像 B 想修改文件 a.txt 内容为“ hello docker”，如果我们允许直接对镜像 A 中的文件 a.txt 进行修改，那么镜像 C 看到 a.txt 内容也随之变为“ hello docker”，对于镜像 C 来说，这是一个不可接受的错误。所以，已有的分层都不能修改，如果要修改，只能通过在镜像 B 的基础上新增加一个分层 B'，存储修改后的 a.txt，利用“上层镜像的优先级高于底层镜像”的原则，新增分层 B' 的 a.txt 会覆盖原有镜像 A 的 a.txt。从用户的视角，就会看到修改后的 a.txt 的内容“ hello docker”，而镜像 C 还是看到原有的 a.txt，内容为“ hello world”，如图 6-5 所示。

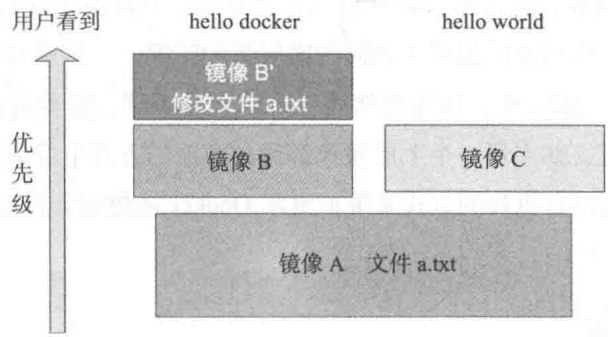


图 6-5　从用户的视角看分层文件的修改

从镜像 B 如何修改文件 a.txt，生成镜像 B' 呢?

在回答这个问题之前，先回头看一下如何用分层的概念描述 Docker 容器。我们知道，容器是在镜像的基础上创建的，从文件系统的角度来讲，它是在分层镜像的基础上增加一个新的空白分层，这个新分层是可读写的，如图 6-6 所示。

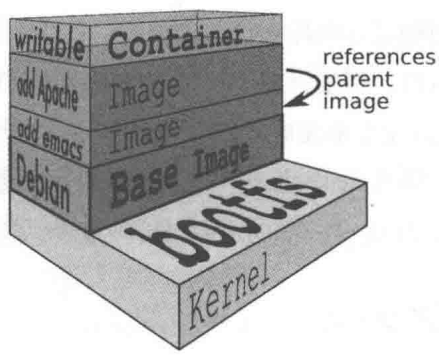


图 6-6　容器的分层示意图

新创建的容器启动后是可写的。所有的写操作都会存储在最上面的可读写层。Docker 容器可以通过 docker commit 命令提交生成新镜像。

对于上面提到的问题：从镜像 B 如何修改文件 a.txt，生成镜像 B'。

步骤如下：

首先，基于镜像 B 创建一个新容器 M1。

其次，在容器 M1 中修改文件 a.txt 的内容。

最后，通过 docker commit 命令提交生成新的镜像 B'。

上面我们讲述了通过对容器的可写层修改，来生成新镜像。但这种方式会让镜像的层数越来越多，达到联合文件系统所允许的最多层数（aufs 最多支持 128 层）；另外一种情况，许多上层的应用镜像都基于相同的底层基础镜像，一旦基础镜像需要修改，比如，底层镜像安装有 glibc 库，该库突然爆出一个安全漏洞，需要升级，而基于它的上层应用镜像成千上万，如果每一个上层镜像都通过容器的方式生成新镜像，那么维护工作量太大了。那还有没有更好的方式来维护更新 Docker 镜像呢？答案是 Dockerfile。

6.2 Dockerfile

Linux 环境下的程序员都应该使用过 GNU make 来构建和管理自己的工程。使用 GNU 的 make 工具能够比较方便地构建一个属于你自己的工程，整个工程的编译只需要一个命令就可以完成编译、连接以至于最后的执行。不过这需要我们投入一些时间去完成一个称为 Makefile 文件的编写。

Makefile 文件中描述了整个工程所有文件的编译顺序、编译规则。Makefile 有自己的书写格式、关键字、函数。其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建哪些库文件以及如何创建这些库文件、如何最后产生我们想要的可执行文件。尽管看起来可能是很复杂的事情，但是，为工程编写 Makefile 的好处是一旦提供一个正确的 Makefile，就能够使用一行命令来完成“自动化编译”。编译整个工程所要做的唯一的一件事就是在 shell 提示符下输入 make 命令。整个工程完全自动编译，极大提高了效率。

Docker 提供了和 Makefile 完全一样的机制来管理镜像，这就是 Dockerfile。

Dockerfile 语法

在讲解 Dockerfile 语法之前，先来看看我们前面使用的一个镜像（sameersbn/redis）

的 Dockerfile 文件的内容。

```
FROM sameersbn/ubuntu:14.04.20160121
MAINTAINER sameer@damagehead.com

ENV REDIS_USER=redis \
    REDIS_DATA_DIR=/var/lib/redis \
    REDIS_LOG_DIR=/var/log/redis

RUN apt-get update \
&& DEBIAN_FRONTEND=noninteractive apt-get install -y redis-server \
&& sed 's/^daemonize yes/daemonize no/' -i /etc/redis/redis.conf \
&& sed 's/^bind 127.0.0.1/bind 0.0.0.0/' -i /etc/redis/redis.conf \
&& sed 's/^# unixsocket /unixsocket /' -i /etc/redis/redis.conf \
&& sed 's/^# unixsocketperm 755/unixsocketperm 777/' -i /etc/redis/redis.conf \
&& sed '/^logfile/d' -i /etc/redis/redis.conf \
&& rm -rf /var/lib/apt/lists/*

COPY entrypoint.sh /sbin/entrypoint.sh
RUN chmod 755 /sbin/entrypoint.sh

EXPOSE 6379/tcp
VOLUME ["${REDIS_DATA_DIR}"]
ENTRYPOINT ["/sbin/entrypoint.sh"]
```

从这个例子我们看到 Dockerfile 的语法规则：每行都以一个关键字为行首，如果一行内容过长，它使用“\”把多行连接到一起。

第一行使用关键字 FROM，它表示新的镜像是从 sameersbn/ubuntu:14.04.20160121 这个基础镜像开始构建的，sameersbn/ubuntu:14.04.20160121 是它的最底层镜像。

MAINTAINER：指定该镜像创建者。

ENV：设置环境变量。

RUN：运行 shell 命令，如果有多条命令可以用“&&”连接。

COPY：将编译机本地文件拷贝到镜像文件系统中。

EXPOSE：指定监听的端口。

ENTRYPOINT：这个关键字和以上所有的关键字是有区别的，上面的关键字都是在构建镜像时执行，但这一个关键字是欲执行命令，在创建镜像时不执行，要等到使用该镜像创建容器，容器启动后才执行的命令。

简单来说，对于 sameersbn/redis 镜像来说，它从基础镜像 sameersbn/ubuntu 开始创建，通过 RUN 关键字安装 redis-server，命令如下：

```
RUN apt-get update \
&& DEBIAN_FRONTEND=noninteractive apt-get install -y redis-server
```

通过 ENTRYPOINT 关键字指定将来创建的新容器使用 /sbin/entrypoint.sh 来启动 redis 服务。脚本文件 entrypoint.sh 的完整内容参考 https://github.com/sameersbn/docker-redis/blob/master/entrypoint.sh。我们只需找出 entrypoint.sh 中启动 redis 服务的那条语句。

```
if [[ -z ${1} ]]; then
  echo "Starting redis-server..."
  exec start-stop-daemon --start --chuid ${REDIS_USER}:${REDIS_USER} --exec
$(which redis-server) -- \
    /etc/redis/redis.conf ${REDIS_PASSWORD:+--requirepass $REDIS_PASSWORD} ${EXTRA_ARGS}
else
  exec "$@"
fi
```

以 sameersbn/redis 为例，我们分析了 Dockerfile 基本语法。下面我们看看如何通过 Dockerfile 编译生成镜像。

先创建一个镜像文件 image_redis，把这一小节开头的内容放入 image_redis/Dockerfile 文件下，另外创建一个 image_redis/entrypoint.sh 文件，内容如下：

```
#!/bin/bash
set -e

REDIS_PASSWORD=${REDIS_PASSWORD:-}

map_redis_uid() {
  USERMAP_ORIG_UID=$(id -u redis)
  USERMAP_ORIG_GID=$(id -g redis)
  USERMAP_GID=${USERMAP_GID:-${USERMAP_UID:-$USERMAP_ORIG_GID}}
  USERMAP_UID=${USERMAP_UID:-$USERMAP_ORIG_UID}
  if [ "${USERMAP_UID}" != "${USERMAP_ORIG_UID}" ] || [ "${USERMAP_GID}"
!= "${USERMAP_ORIG_GID}" ]; then
    echo "Adapting uid and gid for redis:redis to $USERMAP_UID:$USERMAP_GID"
    groupmod -g "${USERMAP_GID}" redis
    sed -i -e "s/:${USERMAP_ORIG_UID}:${USERMAP_GID}:/:${USERMAP_UID}:${USERMAP_
GID}:/" /etc/passwd
  fi
}

create_socket_dir() {
  mkdir -p /run/redis
  chmod -R 0755 /run/redis
```

```
    chown -R ${REDIS_USER}:${REDIS_USER} /run/redis
  }

  create_data_dir() {
    mkdir -p ${REDIS_DATA_DIR}
    chmod -R 0755 ${REDIS_DATA_DIR}
    chown -R ${REDIS_USER}:${REDIS_USER} ${REDIS_DATA_DIR}
  }

  create_log_dir() {
    mkdir -p ${REDIS_LOG_DIR}
    chmod -R 0755 ${REDIS_LOG_DIR}
    chown -R ${REDIS_USER}:${REDIS_USER} ${REDIS_LOG_DIR}
  }

  map_redis_uid
  create_socket_dir
  create_data_dir
  create_socket_dir

  # allow arguments to be passed to redis-server
  if [[ ${1:0:1} = '-' ]]; then
    EXTRA_ARGS="$@"
    set --
  fi

  # default behaviour is to launch redis-server
  if [[ -z ${1} ]]; then
    echo "Starting redis-server..."
    exec start-stop-daemon --start --chuid ${REDIS_USER}:${REDIS_USER} --exec
$(which redis-server) -- \
      /etc/redis/redis.conf ${REDIS_PASSWORD:+--requirepass $REDIS_PASSWORD}
${EXTRA_ARGS}
  else
    exec "$@"
  fi
```

然后，用 docker build 命令编译 Dockerfile，通过“ -t”选项给镜像起一个名字（带版本号）。

```
  $ docker build -t image_redis:v1.0 .
  Sending build context to Docker daemon 4.608 kB
  Step 1 : FROM sameersbn/ubuntu:14.04.20160121
   ---> 4dc780eb0d90
  Step 2 : MAINTAINER sameer@damagehead.com
```

```
 ---> Using cache
 ---> 5641bc665c06
Step 3 : ENV REDIS_USER redis REDIS_DATA_DIR /var/lib/redis REDIS_LOG_DIR
/var/log/redis
 ---> Using cache
 ---> 62058c3b1963
Step 4 : RUN apt-get update  && DEBIAN_FRONTEND=noninteractive apt-get install
-y redis-server  && sed 's/^daemonize yes/daemonize no/' -i /etc/redis/redis.
conf  && sed 's/^bind 127.0.0.1/bind 0.0.0.0/' -i /etc/redis/redis.conf  &&
sed 's/^# unixsocket /unixsocket /' -i /etc/redis/redis.conf  && sed 's/^#
unixsocketperm 755/unixsocketperm 777/' -i /etc/redis/redis.conf  && sed
'/^logfile/d' -i /etc/redis/redis.conf  && rm -rf /var/lib/apt/lists/*
 ---> Using cache
 ---> f08e7917d224
Step 5 : COPY entrypoint.sh /sbin/entrypoint.sh
 ---> 74bd9a76c27b
Removing intermediate container 452ba6152a61
Step 6 : RUN chmod 755 /sbin/entrypoint.sh
 ---> Running in 04d853e4384c
 ---> c2778f2bf1a9
Removing intermediate container 04d853e4384c
Step 7 : EXPOSE 6379/tcp
 ---> Running in 5f99c269417d
 ---> 358281b67a60
Removing intermediate container 5f99c269417d
Step 8 : VOLUME ${REDIS_DATA_DIR}
 ---> Running in dd0cb7dc7469
 ---> dd8fda457738
Removing intermediate container dd0cb7dc7469
Step 9 : ENTRYPOINT /sbin/entrypoint.sh
 ---> Running in 4a30e014dc4f
 ---> f00930ed158b
Removing intermediate container 4a30e014dc4f
Successfully built f00930ed158b
```

编译过程有九步（Step1~Step9），每一步对应 Dockerfiel 的一个关键字，每执行完一步，都会生成一个临时镜像，如 Step 5 生成的临时镜像的 ID 为 74bd9a76c27b。

构建完毕，通过 docker images 就可以查到名字是 image_redis:v1.0 的新镜像。

```
$ docker images
REPOSITORY          TAG         IMAGE ID          CREATED             SIZE
image_redis         v1.0        f00930ed158b      12 minutes ago      196.5 MB
```

有了新镜像，就可以通过 docker run 命令创建和使用新容器了。但该镜像只存在于编译主机，如何把编译好的镜像分发给其他机器使用呢？这需要用到 Docker 仓库中转，

在后续的章节会介绍。

有了 Dockerfile 文件，维护镜像就很简单了。只需要修改 Dockerfile 的某条语句，通过 docker build 重新构建即可，另外还可以通过“-t”选项指定一个新版本，可以很方便地在新旧两个版本间快速切换。

6.3　项目中的镜像分层

我们前面讲过三个项目：WordPress、GitLab 和 Redmine。由于 WordPress 没有公开 Dockerfile，我们跳过不谈。我们把 GitLab 和 Redmine 项目所有镜像分层放在一起来看，就会发现一些有意思的东西，需要说明一下，这两个项目都是由同一个人维护的。GitLab 和 Redmine 的镜像分层的结构如图 6-7 所示。

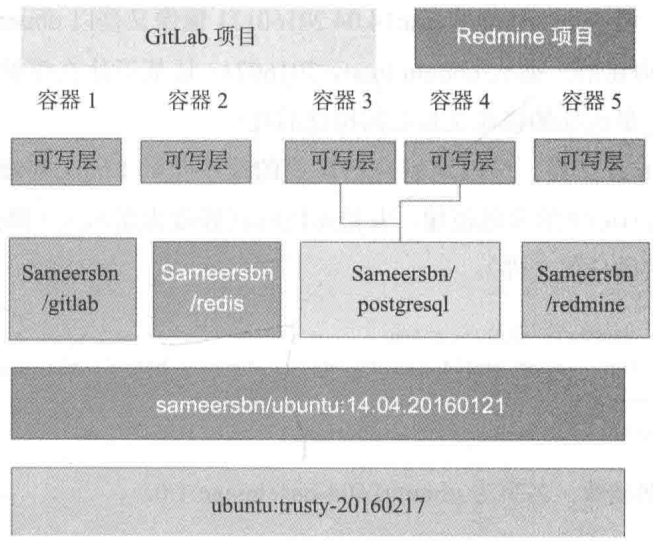


图 6-7　GitLab 和 Redmine 的镜像分层的结构

从图 6-7 我们可以总结如下几点：

- 这两个项目使用四个镜像创建五个容器，这四个镜像都是基于相同的基础镜像（sameersbn/ubuntu），而 sameersbn/ubuntu 又是在更通用的 Ubuntu 系统镜像基础上制作的。
- 每个镜像加一个可写层形成容器，多个容器组合在一起，对外提供某些特殊功能的服务。

- 基于同一个镜像只需要增加一个可写层，就可以为不同项目创建各自需要的容器（容器 3 和容器 4）。

对我们的启发是：当我们制作自己的应用镜像时，也尽量考虑使用相同的底层镜像，这样可以极大地降低后续维护的成本。我们根据自己的实际应用场景选择适合自己的基础镜像，也可以在已有的基础镜像上改造提交新镜像作为自己项目的基础镜像。但有些时候，我们在 Docker Hub 上实在找不到适合自己用的基础镜像，这时就可以从头打造一个完全属于自己的基础镜像。

6.4 定制私有的基础镜像

从 6.2 节我们知道 sameersbn/redis 镜像是以 sameersbn/ubuntu:14.04.20160121 为基础镜像进行构建的，而 sameersbn/ubuntu:14.04.20160121 镜像又是以 ubuntu:trusty-20160217 为基础镜像进行构建的。那么 ubuntu:trusty-20160217 是基于什么镜像构建的呢？我们想知道，最基础、最底层的镜像又是如何构建的呢？

使用 debootstrap 工具，可以定制自己需要的最小化的 Linux 基础镜像，这里我们制作了一个 Ubuntu14.04 的基础镜像，并把系统时区修改为东八区（修改时区只是举例，其实可以修改系统的任何文件）。

```
sudo apt-get install debootstrap
sudo debootstrap --arch amd64 trusty ubuntu-trusty http://mirrors.163.com/ubuntu/
cd ubuntu-trusty
sudo cp usr/share/zoneinfo/Asia/Shanghai  etc/localtime
```

提交生成基础镜像，名字为 ubuntu1404-baseimage:1.0。

```
cd ubuntu-trusty
sudo tar -c .|docker import - ubuntu1404-baseimage:1.0
```

通过 docker images 可以查到新创建的镜像。

```
$ docker images
REPOSITORY              TAG     IMAGE ID        CREATED             SIZE
ubuntu1404-baseimage    1.0     47f52695a5c6    43 seconds ago      228.3 MB
```

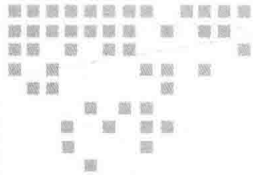
我们新创建一个容器，查看 Ubuntu 的系统版本和时区修改是否成功（通过 date 命令核对系统时间是否正确）。

```
    harney@UShenzhou:~/image_redis/ubuntu-trusty$ docker run -t -i ubuntu1404-
baseimage:1.0 /bin/bash
    root@c09ba1a54004:/# cat /etc/issue
    Ubuntu 14.04 LTS \n \l
    root@1b7b128888e9:/# date
    Sun Feb 21 17:29:12 CST 2016
```

这样我们就制作好了属于自己的私有镜像。我们的应用层的镜像就可以在该镜像的基础上继续扩展了。

6.5　本章小结

本章主要分析了 Docker 镜像的分层的组织结构和背后的原理，然后通过举例讲解了 DockerFile 在镜像创建和修改时的操作，最后还简单介绍了如何定制一个私有的基础镜像。



Chapter 7 第 7 章

Docker 仓库管理

前面已经介绍了 Docker 的容器和镜像，本章就详细介绍最后一个组件——Docker 仓库，仓库主要用于镜像的存储，它是 Docker 镜像分发、部署的关键。在实际应用中，由开发者或者运维制作好应用程序镜像，然后上传到镜像仓库。Docker 守护进程再从仓库拉取镜像，然后运行相应的镜像。

我们可以使用官方的公有仓库 Docker Hub，也可以搭建自己的私有仓库来存储我们的镜像。本章将详细讨论这两种方式。

7.1 镜像的公有仓库

目前 Docker 官方维护了一个公有仓库 Docker Hub，其中已经包括了超过 125 000 个公共镜像。如果我们仅仅需要搜索和使用 Docker Hub 的公共镜像，不需要 Docker Hub 账户就可以直接操作。但如果需要上传和分享我们创建的镜像，就需要 Docker Hub 账户。另外，Docker Hub 还支持用户创建私有的镜像仓库，用于私有镜像的存储和跨主机部署。

7.1.1 创建 Docker Hub 账户

在使用 Docker Hub 之前，我们需要先创建自己的账号。通过 Web 界面 https://hub.

docker.com/，输入用户名、邮箱和密码就可以完成注册，然后通过邮箱激活。使用用户名、密码登录后界面如图 7-1 所示。

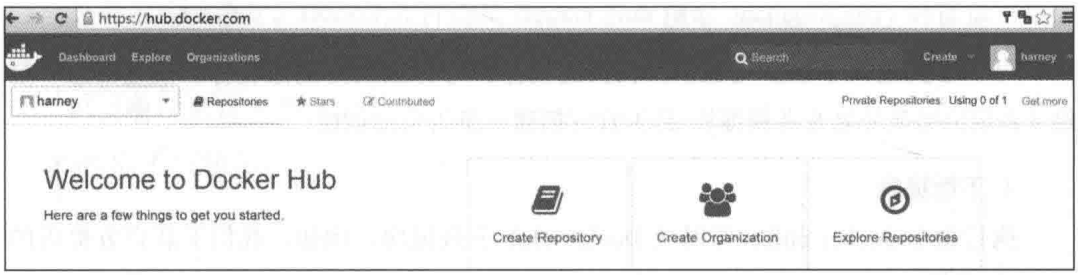


图 7-1　Docker Hub 登录界面

在命令行终端也可以通过 docker login 登录 Docker Hub 账户。

```
$ docker login
Username: harney
Password:
Email: harney@hahabot.com
WARNING: login credentials saved in /home/harney/.docker/config.json
Login Succeeded
```

登录 Docker Hub 后，就可以进行搜索、下载和上传镜像等基本操作了。

7.1.2　基本操作

在第 6 章，我们制作了一个定制化的基础镜像 ubuntu1404-baseimage:1.0，现在我们讨论一下如何通过 Docker Hub 来上传、搜索、下载该镜像。

1. 上传镜像

首先通过 docker login 登录 Docker Hub，然后才能上传镜像。上传镜像通过 docker push 命令实现。

```
docker push ubuntu1404-baseimage:1.0
```

2. 搜索镜像

我们可以执行 docker search 来查找 Docker Hub 中的镜像，可以通过镜像名称、用户名称及描述信息等搜索镜像。例如，我们搜索与 centos 相关的镜像。

```
# docker search centos
```

```
NAME              DESCRIPTION                STARS     OFFICIAL   AUTOMATED
centosThe official build of CentOS.          558       [OK]
tianon/centos    CentOS 5 and 6, created using rinse instea...   28
ansible/centos7-ansible    Ansible on Centos7                    14   [OK]
...
```

可以看到返回了很多与 centos 相关的镜像，每行依次为镜像名称、描述信息、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。

3. 下载镜像

执行命令 docker pull，可以从 Docker Hub 下载镜像。例如，我们下载官方提供的 CentOS 镜像。

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

Status: Downloaded newer image for centos
```

7.2 私有仓库

对于个人来说，如果只是学习 Docker，Docker Hub 就足够了。但是，如果想构建一个基于 Docker 的 PaaS 平台，使用 Docker Hub 这样的公共仓库可能不方便，原因如下：

- Docker 公有仓库和私有仓库基本使用。
- 很多公司的 IDC 环境是无法访问外网的。
- 很多应用，考虑到安全因素，将程序直接放到公共仓库是不合适的。
- 网络速度、带宽都会成为“瓶颈”。

所以，我们有必要构建自己的私有仓库。Docker 官方已经提供了 docker-registry 组件，我们可以用它来构建我们自己的私有镜像仓库。

7.2.1 安装 docker-registry

1. 使用镜像方式

Docker 官方提供了 docker-registry 的镜像，我们可以直接使用该镜像。这也是最简

单的方式。

```
#docker run -p 5000:5000 registry
```

执行上面的命令，Docker 会自动从 Docker Hub 拉取 docker-registry 的镜像，然后启动 docker-registry 服务，docker-registry 默认监听 5000 端口。

可以通过环境变量方式“-e”设置配置参数。例如，如果 docker-registry 想使用 Amazon S3 存储镜像，可以执行下面的命令：

```
docker run \
        -e SETTINGS_FLAVOR=s3 \
        -e AWS_BUCKET=mybucket \
        -e STORAGE_PATH=/registry \
        -e AWS_KEY=myawskey \
        -e AWS_SECRET=myawssecret \
        -e SEARCH_BACKEND=sqlalchemy \
        -p 5000:5000 \
registry
```

详细配置参数见 7.2.2 节。

2. 使用 rpm 包方式

目前，EPEL（Fedora Extra Packages for Enterprise Linux）中已经包含 docker-registry 的包，我们可以直接使用。

安装 docker-registry：

```
#yum install docker-registry-y
```

启动 docker-registry：

```
# servicedocker-registry start
Starting docker-registry:                                  [  OK  ]
# servicedocker-registry status
docker-registry (pid  31079) is running...
```

默认，docker-registry 会监听 5000 端口，启动八个工作进程。

```
# netstat -ltnp|grep 5000
tcp            0          0  0.0.0.0:5000                          0.0.0.0:*
LISTEN      31079/python
# ps -ef|grep 31079
root      31079      1   0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
```

```
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31085 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31087 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31091 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31096 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31101 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31106 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31111 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
    root      31112 31079  0 10:39 pts/2    00:00:00 /usr/bin/python /usr/bin/
gunicorn --access-logfile - --debug --max-requests 100 --graceful-timeout 3600
-t 3600 -k gevent -b 0.0.0.0:5000 -w 8 docker_registry.wsgi:application
```

这些参数是在配置文件 /etc/sysconfig/docker-registr 时设置的。

```
# The Docker registry configuration file
# DOCKER_REGISTRY_CONFIG=/etc/docker-registry.yml

# The configuration to use from DOCKER_REGISTRY_CONFIG file
SETTINGS_FLAVOR=local# 控制 /etc/docker-registry.yml 使用的 flavor

# Address to bind the registry to
REGISTRY_ADDRESS=0.0.0.0# 监听地址

# Port to bind the registry to
REGISTRY_PORT=5000 # 监听端口

# Number of workers to handle the connections
GUNICORN_WORKERS=8# 工作进程数量
```

注意 这个配置并不是 docker-registry 提供的，而是在 EPEL 中加上去的，使用 /etc/init.d/docker-registry。

7.2.2　配置文件

在默认情况下，docker-registry 使用 config_sample.yml 进行各种配置，rpm 方式则使用 /etc/docker-registry.yml。

配置文件使用 yml 格式，并提供各种不同的模板，docker-registry 可以针对不同的环境选择不同的模板。

在 config_sample.yml 文件中，可以看到一些示例模板：

```
common: 公共基础配置，其他模板可以引用该模板
local: 存储数据到本地文件系统
s3: 存储数据到 AWS S3
ceph-s3: 通过 Ceph 对象网关将数据存储到 Ceph 集群
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
```

官方提供了一个针对生产环境、开发环境和测试环境的配置，我们只需要稍做修改就可以用于自己的环境中。

```
common: &common
standalone: true
loglevel: info
search_backend: "_env:SEARCH_BACKEND:"
sqlalchemy_index_database:
"_env:SQLALCHEMY_INDEX_DATABASE:sqlite:////tmp/docker-registry.db"

prod:
<<: *common
loglevel: warn
storage: s3
s3_access_key: _env:AWS_S3_ACCESS_KEY
s3_secret_key: _env:AWS_S3_SECRET_KEY
s3_bucket: _env:AWS_S3_BUCKET
boto_bucket: _env:AWS_S3_BUCKET
storage_path: /srv/docker
smtp_host: localhost
from_addr: docker@myself.com
```

```
to_addr: my@myself.com

dev:
<<: *common
loglevel: debug
storage: local
storage_path: /home/myself/docker

test:
<<: *common
storage: local
storage_path: /tmp/tmpdockertmp
```

7.3 构建安全的私有仓库

目前 docker-registry 没有提供安全认证，所以，所有知道 URL 的人都可以上传镜像，这在实际生产环境中是非常危险的。我们需要认证功能，可以使用 Nginx 构建一个带认证功能的私有仓库。

7.3.1 Nginx 安装与配置

1. 安装 Nginx

安装 Nginx 的命令如下：

```
#yum install nginx -y
```

推荐使用 1.3.9 以上的 Nginux。

2. 配置

创建 /etc/nginx/conf.d/registry.conf 文件，内容如下：

```
# For versions of nginx> 1.3.9 that include chunked transfer encoding support
# Replace with appropriate values where necessary

upstreamdocker-registry {
  server localhost:5000; # 这里修改为你的 docker-registry 的地址
}

# uncomment if you want a 301 redirect for users attempting to connect
# on port 80
# NOTE: docker client will still fail. This is just for convenience
```

```
    # server {
    #   listen *:80;
    #   server_name my.docker.registry.com;
    #   return 301 https://$server_name$request_uri;
    # }

    server {
    listen 443;
    server_name dev.registry.com;

    ssl on;# 打开 SSL
    ssl_certificate /etc/ssl/certs/docker-registry.crt; # 公钥证书
    ssl_certificate_key /etc/ssl/private/docker-registry.key; # 私钥

    client_max_body_size 0; # disable any limits to avoid HTTP 413 for large
image uploads

      # required to avoid HTTP 411: see Issue #1486 (https://github.com/docker/
docker/issues/1486)
    chunked_transfer_encoding on;

    location / {
    auth_basic            "Restricted";
    auth_basic_user_filedocker-registry.htpasswd; # 用户名、密码文件
    includedocker-registry.conf;
      }

    location /_ping {
    auth_basic off;
    includedocker-registry.conf;
      }

    location /v1/_ping {
    auth_basic off;
    includedocker-registry.conf;
      }
    }
```

因为后面涉及密码传输，这里打开了 SSL 的支持。

创建 /etc/nginx/docker-registry.conf 文件，内容如下：

```
    proxy_pass                    http://docker-registry;
    proxy_set_header  Host        $http_host;   # required for docker client's sake
    proxy_set_header  X-Real-IP   $remote_addr; # pass on real client's IP
    proxy_set_header  Authorization  ""; # see https://github.com/dotcloud/
docker-registry/issues/170
```

```
proxy_read_timeout              900;
```

用 htpasswd 创建认证的用户和密码，命令如下：

```
htpasswd -bc /etc/nginx/docker-registry.htpasswd USERNAME PASSWORD
```

例如：

```
# htpasswd -bc /etc/nginx/docker-registry.htpasswd  docker  docker
Adding password for user docker
# cat /etc/nginx/docker-registry.htpasswd
docker:FRF5oER6LpDCc
```

到这里，Nginx 的基本配置完成，但别忙启动 Nginx，我们还需要给 Nginx 配置 SSL 证书。

7.3.2 SSL 证书

一般来说，我们应该使用权威 CA（Certification Authority）机构签名的证书（Certificates）。为了简单，我们这里使用自己签名（Self-Signed）的证书。

1. 创建 CA

在给 Nginx 创建签名的证书之前，我们先要创建一个我们自己的 CA，CA 包含公钥和私钥，私钥用于给其他证书签名，公钥用于别人验证证书的有效性

```
# echo 01 >ca.srl
# openssl genrsa -des3 -out ca-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.............................+++
.....................................................................
.....................................................................+++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:

# openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
```

```
Country Name (2 letter code) [XX]:CN
State or Province Name (full name) []:Guangdong
Locality Name (eg, city) [Default City]:Shenzhen
Organization Name (eg, company) [Default Company Ltd]:Tencent
Organizational Unit Name (eg, section) []:IEG
Common Name (eg, your name or your server's hostname) []:dev.registry.com
Email Address []:dbyin@tencent.com
```

现在，我们有了一个自己的 CA，就可以为 Nginx 创建证书了。

2. 为 Ngnix 创建证书

使用 openssslgenrsa 创建证书的命令如下：

```
# openssslgenrsa -des3 -out server-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.................+++
e is 65537 (0x10001)
Enter pass phrase for server-key.pem:
Verifying - Enter pass phrase for server-key.pem:

# openssslreq -subj '/CN=dev.registry.com' -new -key server-key.pem -out
server.csr
Enter pass phrase for server-key.pem:

# openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkeyca-key.pem
-out server-cert.pem
Signature ok
subject=/CN=dev.registry.com
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

接着，删除 server key 中的 passphrase，

```
# openssslrsa -in server-key.pem -out server-key.pem
Enter pass phrase for server-key.pem:
writing RSA key
```

然后，安装 server-key 和 server-crt。

```
# cp server-cert.pem /etc/ssl/certs/docker-registry.crt
# cp server-key.pem /etc/ssl/private/docker-registry.key
```

到这里，Nginx 的 SSL 证书就算配置好了。启动 Nginx 即可。

```
#service  nginx start
```

7.3.3 客户端配置

为了 Docker 能够正常地访问 Nginx，需要安装我们自己的 CA，用于验证 Nginx 的证书的有效性。

注意 目前，Docker 如果使用 HTTPS 链接，会验证证书的有效性，不允许 curl -k 类似的非安全链接。但是，已经有一些议题（Issues）在讨论非安全的链接，具体可参考：

https://github.com/docker/docker/pull/2687

https://github.com/docker/docker/pull/5817

https://github.com/docker/docker/pull/8467

1. 安装 CA

通过以下命令完成安装 CA：

```
# update-ca-trust enable
# cpca.pem /etc/pki/ca-trust/source/anchors/ca.crt
# update-ca-trust extract
```

然后重启 Docker。

2. 登录 Nginx

执行以下命令登录 Nginx：

```
# docker login -u docker -p docker -e dbyin@tencent.com https://dev.
registry.com
Login Succeeded
```

这会在 $home/ 目录下生成一个 .dockercfg 文件，保存认证信息。

```
# cat /root/.dockercfg
{"https://dev.registry.com":{"auth":"ZG9ja2VyOmRvY2tlcg==","email":"dbyin@
tencent.com"}}
```

然后我们就可以上传自己的镜像了。

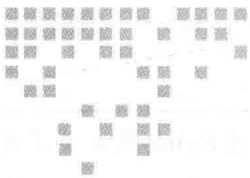
```
# docker images
REPOSITORY          TAG         IMAGE ID        CREATED         VIRTUAL SIZE
dbyin/httpd         latest      93e711fab1c1    5 weeks ago     399.6 MB
```

```
# docker tag 93e711fab1c1 dev.registry.com/dbyin/httpd
# docker push dev.registry.com/dbyin/httpd
```

总的来说，相对于官方的 Docker Hub，这种通过 Nginx 完成验证的做法是比较粗糙的。但是，社区已经在讨论给 docker-registry 增加验证的功能了，可以参考：https://github.com/docker/docker-registry/issues/541

7.4　本章小结

本章介绍了公有仓库的使用方法，以及在什么场景下需要使用私有仓库，如何创建和管理安全的私有仓库。



Chapter 8 第 8 章

Docker 网络和存储管理

本章我们着重讨论一下 Docker 各容器之间如何相互通信，以及在 Docker 整个生命周期中数据管理方式。

8.1 Docker 网络

网络是虚拟化技术中最复杂的部分，也是 Docker 应用中的一个重要环节。Docker 中的网络主要解决容器与容器、容器与外部网络、外部网络与容器之间的互相通信的问题。本节我们主要讨论一下 Docker 中网络的一些基本原理和应用。

8.1.1 Docker 的通信方式

在默认情况下，Docker 使用网桥（bridge）+NAT 的通信模型，大致如图 8-1 所示。Docker 在启动时默认会自动创建网桥设备 Docker0，并配置 IP172.17.42.1/16：

```
# ifconfig docker0
docker0   Link encap:EthernetHWaddr 46:2E:39:8C:D9:57
inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
inet6addr: fe80::442e:39ff:fe8c:d957/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
```

```
collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:468 (468.0 b)
```

当 Docker 启动容器时，会创建一对 veth 虚拟网络设备，并将其中一个 veth 网络设备附加到网桥 docker0，另一个加入容器的网络名字空间（network namespace），并改名为 eth0。这样，同一个 Host 的容器与容器之间就可以通过 docker0 通信了。

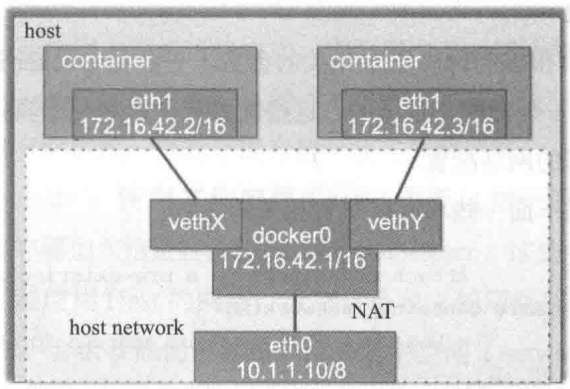


图 8-1　网桥 +NAT 的通信模型

仅仅解决 Host 内部的容器之间的通信是不够的，还需要解决容器与外部网络之间的通信，为此，Docker 引入 NAT。

（1）容器访问外部网络

为了解决容器访问外部网络，Docker 创建如下 MASQUERADE 规则：

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

这条规则将所有从容器（172.17.0.0/16）发出的、目的地址为 Host 外部网络的包的 IP 都修改成 Host 的 IP，并由 Host 发送出去。

（2）外部网络访问容器

如果容器提供的服务需要暴露给外部网络，Docker 在启动容器时，就会创建 SNAT 规则。比如，我们启动一个 apache 容器：

```
#docker run -d -P 80:80 apache
```

这会创建下面的 SNAT 规则：

```
iptables -t nat-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
iptables -t nat-A DOCKER ! -i docker0 -p tcp -m tcp --dport80 -j DNAT
```

```
--to-destination 172.17.0.2:80
```

实际上，第一条规则是在 Docker 进程启动时默认创建的，第二条规则是在启动容器时创建的。

8.1.2 网络配置

1. 网络配置参数

在 Docker 中，有很多与网络配置相关的参数，一些是 Docker 进程本身的，这会影响所有的容器；另外一些是配置容器的，这些配置只会影响某个具体的容器。

（1）Docker 进程的网络配置

Docker 进程提供下面一些与网络配置相关的参数：

```
    -b, --bridge=""        Attach containers to a pre-existing network bridge
    use 'none' to disable container networking
      --bip=""               Use this CIDR notation address for the network bridge's
IP, not compatible with -b
      --dns=[]               Force docker to use specific DNS servers
      --dns-search=[]        Force Docker to use specific DNS search domains
    --icc=true             Enable inter-container communication
      --ip="0.0.0.0"         Default IP address to use when binding container ports
      --ip-forward=true      Enable net.ipv4.ip_forward
      --iptables=true        Enable Docker's addition of iptables rules
      --mtu=0                Set the containers network MTU
    if no value is provided: default to the default route MTU or 1500 if no default
route is available
```

我们来逐个看这些参数：

- -b/--bridge：指定 Docker 使用的网桥设备。默认情况下，Docker 会创建（使用）docker0 网桥设备，通过该参数可以指定 Docker 使用已经存在的网桥设备。
- --bip：指定网桥设备 docker0 的 IP 和掩码，使用标准的 CIDR 形式，如 192.168.1.5/24。
- --dns/--dns-search：配置容器的 DNS，该参数既可以在启动 Docker 进程时指定（成为所有容器的默认值），也可以在启动容器（docker run）时指定（覆盖默认值）。我们会在下面介绍“配置 DNS”时详细讨论。

（2）容器的网络配置

下面一些参数是在执行 docker run 时，提供给具体容器的：

```
    --net="bridge"              Set the Network mode for the container
                                    'bridge': creates a new network stack for
the container on the docker bridge
                                    'none': no networking for this container
                                    'container:<name|id>': reuses another
container network stack
                                    'host': use the host network stack inside
the container.  Note: the host mode gives the container full access to local
system services such as D-bus and is therefore considered insecure.
```

--net 用于指定容器使用的网络通信方式，它可以取下面四个值：

- bridge：这个 Docker 中的容器默认的方式，在 8.1.1 节中已经详细讨论过。
- none：容器没有网络栈，也就是说容器无法与外部通信。
- container:<name|id>：使用其他容器（name 或者 id 指定）的网络栈。实际上，Docker 会将该容器加入指定容器的 network namespace，这是一种非常有用的方式。
- host：表示容器使用 Host 的网络，没有自己独立的网络栈。实际上，在这种情况下，Docker 不会给容器创建单独的网络名字空间（network namespace）。由于容器可以完全访问 Host 的网络，所以此方式也是不安全的。

2. 配置 DNS

一般来说，每个容器的 hostname 和 DNS 配置信息是不同的，我们不可能为每个容器都构建一个镜像，并在镜像中指定这些信息。那么如何解决这个问题呢?

实际上，Docker 在启动容器时，会使用 bind mount 动态挂载 /etc/hostname、/etc/hosts、/etc/resolv.conf 几个文件，覆盖镜像中原来的文件，我们可以在容器内部看到这个信息：

```
$$ mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
tmpfs on /etc/resolv.conf type tmpfs ...
```

所以，我们可以通过命令行参数在启动 Docker 时指定 DNS。

8.2　Docker 数据管理

8.2.1　基本介绍

Docker 中的容器一旦删除，容器本身对应的 rootfs 文件系统就会被删除，容器中

的所有数据也将随之删除。但有的时候，我们想要数据如日志或者其他需要持久化的数据，不随容器的删除而删除。还有的时候，我们希望在同一台 Host 的容器之间可以共享数据。

为此，Docker 提供了数据卷（data volume），数据卷除了可以持久化数据，还可以用于容器之间共享数据。

8.2.2 数据卷

Docker 中有两个与数据卷相关的参数：

```
-v=[]: Create a bind mount with: [host-dir]:[container-dir]:[rw|ro].
     If "container-dir" is missing, then docker creates a new volume.
--volumes-from="": Mount all volumes from the given container(s)
```

我们先看参数“-v”，通过该参数可以给容器创建数据卷，它有三个变量：

- host-dir：表示 Host 上的目录，如果不存在，Docker 会自动在 Host 上创建该目录。
- container-dir：表示容器内部对应的目录，如果该目录不存在，Docker 也会在容器内部创建该目录。
- rw|ro：用于控制卷的读写权限。

1. 创建数据卷

我们可以不指定 host-dir，从而在容器内部创建一个数据卷：

```
[root@yinye ~]# docker run -it --rm -v /volume1 --name test1 ubuntu:14.04
/bin/bash
root@5cb86de3eee4:/# df -lh
FilesystemSize  Used Avail Use% Mounted on
tmpfs                          935M     0  935M   0% /dev
shm                             64M     0   64M   0% /dev/shm
/dev/mapper/vg_yinye-lv_root    18G   14G  3.0G  83% /volume1
tmpfs                          935M     0  935M   0% /proc/kcore
root@5cb86de3eee4:/# ls /volume1/
root@5cb86de3eee4:/# echo  "volume1"> /volume1/test.txt
root@5cb86de3eee4:/# ls /volume1/
test.txt
```

执行 df 可以看到 Host 的根分区被挂载到了容器的 /volume1。实际上，Docker 会在 Host 的 /var/lib/docker/vfs/dir/ 目录生成一个随机的目录，然后挂载容器的 /volume1。

```
# docker inspect test1
```

```
[{
...
"Volumes": {
"/volume1": "/var/lib/docker/vfs/dir/144d7df40fc569b25221b4d2dd14b5b21ba84
0accbbac627958080bbb04db109"
    },
"Vo
"VolumesRW": {
"/volume1": true
    }
}
]
[root@yinye ~]#ls /var/lib/docker/vfs/dir/144d7df40fc569b25221b4d2dd14b5b2
1ba840accbbac627958080bbb04db109
test.txt
[root@yinye ~]# cat /var/lib/docker/vfs/dir/144d7df40fc569b25221b4d2dd14b5
b21ba840accbbac627958080bbb04db109/test.txt
volume1
```

对于这种方式创建的数据卷，当容器被删除后，如果没有其他容器引用该数据卷，对应的 Host 目录也会被删除。所以，如果不想 Host 的目录被删除，必须指定 Host 的目录。

2. 挂载 Host 的目录作为数据卷

除了创建数据卷外，我们还可以挂载 Host 的目录到容器，作为容器的数据卷。

```
[root@yinye ~]# docker run -it --rm -v /data/volume1:/volume1 ubuntu:14.04
/bin/bash
root@97556a0d5b21:/# df -lh
FilesystemSize  Used Avail Use% Mounted on
tmpfs                          935M     0  935M   0% /dev
shm                             64M     0   64M   0% /dev/shm
/dev/mapper/vg_yinye-lv_root    18G   14G  3.0G  83% /volume1
tmpfs                          935M     0  935M   0% /proc/kcore
root@97556a0d5b21:/# ls /volume1/
root@97556a0d5b21:/# echo "hello"> /volume1/hello.txt
root@97556a0d5b21:/# exit
exit
[root@yinye ~]# ls /data/volume1/
hello.txt
[root@yinye ~]# cat /data/volume1/hello.txt
hello
```

我们将 Host 上的 /data/volume1 挂载容器中的 /volume1。通过这种方式我们可以在

Host 与容器之间进行数据交换。比如，容器内的应用程序可以将日志、重要数据写到 /volume1 上，这样，即使容器被删除，数据仍然会保留在 Host 上。实际上，Docker 内部是通过 mount --bind 来实现的。

Host 目录必须是绝对路径，如果该目录不存在，Docker 会自动创建该目录。

在默认情况下，容器对挂载的数据具有读写权限。我们也可以挂载为只读权限：

```
[root@yinye ~]# docker run -it --rm -v /data/volume1:/volume1:ro
ubuntu:14.04 /bin/bash
root@762ec88b090b:/# df -lh
FilesystemSize  Used Avail Use% Mounted on
tmpfs                          935M     0  935M   0% /dev
shm                             64M     0   64M   0% /dev/shm
/dev/mapper/vg_yinye-lv_root    18G   14G  3.0G  83% /volume1
tmpfs                          935M     0  935M   0% /proc/kcore
root@762ec88b090b:/# touch /volume1/hello.txt
touch: cannot touch '/volume1/hello.txt': Read-only file system
root@762ec88b090b:/#
```

可以看到，当我们以只读权限挂载时，容器对目录写会失败。

3. 挂载 Host 的文件作为数据卷

我们除了可以挂载 Host 的目录作为容器的数据卷外，还可以挂载 Host 的文件作为容器的数据卷。例如：

```
#docker run --rm -it -v ~/.bash_history:/root/.bash_historyubuntu /bin/bash
```

这样就能在容器中查看 Host 的 bash 的历史命令了，在我们退出容器后，在 Host 中也能看到容器执行的命令历史。

这种挂载 Host 的文件方式主要用于在 Host 与容器之间共享配置文件。一般来说，应用程序不会变，而配置文件可能会经常变，如果对每个配置文件都做一个镜像，会造成镜像版本过多、管理不便，而且不够灵活。实际上，我们可以将配置文件放在 Host 上面，然后挂载到容器，这样，我们就可以随时更改 Host 文件，容器内部看到的内部也会随之改变，这种方式会更加简单灵活。来看个实际例子吧。

在一般情况下，我们运行一个容器，容器内部看到的时区可能会与 Host 不一致：

```
[root@yinye ~]# date +%z
+0800
[root@yinye ~]# docker run -it --rm ubuntu:14.04 /bin/bash
root@e659855775b5:/# date +%z
```

```
+0000
```

我们可以将 Host 的 /etc/localtime 挂载到容器内部：

```
[root@yinye ~]# docker run -it --rm -v /etc/localtime:/etc/localtime
ubuntu:14.04 /bin/bash
root@77c80ec1f9e2:/# date +%z
+0800
```

可以看到，容器内部看到的时区与 Host 已经一致了。

注意　很多编辑工具，包括 vi 和 sed --in-place 可能会造成文件的 inode 改变。从 Docker 1.1.0 开始，这会产生错误“sed: cannot rename ./sedKdJ9Dy: Device or resource busy”。在这种情况下，如果想编辑文件，最好挂载文件的父目录。

8.2.3　数据卷容器

在前面的内容中，我们提到 Docker 有两个与数据卷相关的参数，本节我们来讨论另外一个参数“--volumes-from”，该参数主要用于数据卷容器（Data Volume Container）的场景。

1. 创建和挂载数据卷容器

很多时候，我们会将一些相关的容器部署在同一个 Host 上，并且希望这些容器之间可以共享数据。这时，我们可以创建一个命名的数据卷容器，然后供其他容器挂载。例如，我们创建一个 dbdata 的容器，它包含一个 /dbdata 的数据卷：

```
#docker run -d -v /dbdata --name dbdata training/postgres echo Data-only
container for postgres
```

然后我们就可以通过 --volumes-from 在其他容器挂载 /dbdata 数据卷。

我们创建容器 db1：

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

我们还可以再创建容器挂载该数据卷：

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

这样，db1 和 db2 也能看到容器 dbdata 所有的数据卷（/dbdata）的内容。

我们可以同时使用多个 --volumes-from 参数，从多个容器挂载多个数据卷。我们还可以从其他已经挂载容器卷的容器（如 db1）挂载数据卷：

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

如果我们删除挂载了数据卷的容器（包括初始的 dbdata 容器和其他的容器 db1、db2），数据卷并不会被删除。如果想删除该数据卷，必须在删除最后一个引用该数据卷的时候调用 dockerrm -v 显示删除数据卷。

2. 数据卷容器的应用

Docker 的应用哲学是一个容器一个程序，当然，我们也可以在一个容器中运行多个程序（推荐使用 supervisor），但这并不是 Docker 推荐的。

而实际上，很多应用程序通常会通过系统的 syslog 记录日志。我们可以将应用程序和 rsyslog 同时安装到镜像中，然后在同一个容器中同时运行应用程序和 rsyslog。如果你只需要一个容器，这样做可能没有太大问题，如果你需要多个容器，每个容器都跑一个 rsyslog，难免造成资源的浪费。更好的方式是，我们创建一个容器专门运行 rsyslog 收集日志，其他应用程序将日志发送到该日志容器。这样，我们不仅可以更好地集中管理日志，也避免了资源的浪费。

（1）构建 rsyslog 镜像

我们先创建一个只运行 rsyslog 的镜像，Dockerfile 如下：

```
#forrsyslog
FROM centos6
MAINTAINER hustcat
RUN  yum -y install rsyslog&& yum clean all

CMD rsyslogd -n
VOLUME /dev
VOLUME /var/log
```

执行下面的命令生成镜像：

```
#docker build -t hustcat/rsyslog .
```

（2）运行 rsyslog 容器

我们启动 rsyslog 容器：

```
# docker run --name rsyslog -d -v /tmp/syslogdev:/devhustcat/rsyslog
```

成功启动后，我们在 Host 上可以看到 /tmp/syslogdev/log 的 Unix socket 文件：

```
# ls /tmp/syslogdev/log -lh
srw-rw-rw- 1 root root 0 Sep 19 14:24 /tmp/syslogdev/log
```

（3）在其他容器写 log 到日志容器

接下来，我们就可以在另一个容器写 log 了：

```
# docker run --rm -v /tmp/syslogdev/log:/dev/log centos6 logger -p info
"hello rsyslog"
```

我们还可以在日志容器中做更多的事情，如将日志发到远端服务集中存储管理。总之，通过这种方式管理容器日志更加方便灵活。目前，业界已经有一些专门处理容器日志的工具，如 loggly（https://www.loggly.com/blog/centralize-logs-docker-containers/）。

8.2.4　备份、恢复和迁移数据卷

我们使用数据卷共享数据，难免面临数据的备份、恢复和迁移的问题。

1. 备份数据卷

我们可以通过参数“--volumes-from”从数据卷挂载数据卷，然后备份数据卷中的数据，例如：

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

这里我们创建一个新的容器，将 Host 本地目录挂载到 /backup，然后将数据卷容器 dbdata 的数据卷（/dbdata）打包到 /backup/backup.tar。然后在 Host 的当前目录下就可以得到 backup.tar。

2. 恢复数据卷

我们可以将备份的数据恢复到原有容器或者其他任何容器。假设我们想把 backup.tar 的数据恢复到一个新的容器 dbdata2：

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

然后执行下面的命令即可。

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

8.3 Docker 存储驱动

Docker 存储驱动（storage driver）是 Docker 的核心组件，它是 Docker 实现分层镜像的基础，本节将介绍一下 Docker storage driver 的历史及一些最新的进展。

8.3.1 Docker 存储驱动历史

Docker 目前支持很多 graph driver，最开始使用 AUFS，但 AUFS 一直没有进入内核主线。但 RHEL/Fedora 等发行版本并不支持 AUFS，所以，Redhat 的 Alexander Larsson 实现了 device-mapper 的 driver，现在 dm driver 由 Vincent Batts 在维护。Alexander 当时选择了 device-mapper，主要是由于 btrfs 不成熟，overlayfs 也没有进入内核主线，所以，它选择了 device-mapper 作为 RHEL/Fedora 下的解决方案。

1. device mapper

在相当长一段时间内，DM（device mapper）几乎成为生产环境的使用 Docker 的唯一选择，但在实际中，经常会遇到很多问题。比如，你一定经常遇到下面的问题：

```
Driver devicemapper failed to remove root filesystem ... : Device is Busy
```

另外，想让 DM 工作稳定，需要 udev 的支持，而 udev 没有静态库。最后，Docker 希望通过容器之间共享 pagecache，试想，如果一台机器上有几百个容器，如果每个容器都打开一份 glibc，这会浪费许多内存。由于 DM 工作在块层，很难实现 pagecache 的共享。

另外，在默认情况下，Docker 基于文件＋ loop 设备构建 DM 块设备，会导致 IO 路径过于冗长，性能和稳定性都是一个很大的问题。

因此，很多人都不建议在生产环境中使用 DM。个人在使用 DM 的过程中也遇到一些问题，包括导致内核 crash 的问题、性能问题等。

2. btrfs

再后来社区实现了 btrfs driver。但 btrfs 在稳定性、性能上都存在一些问题。

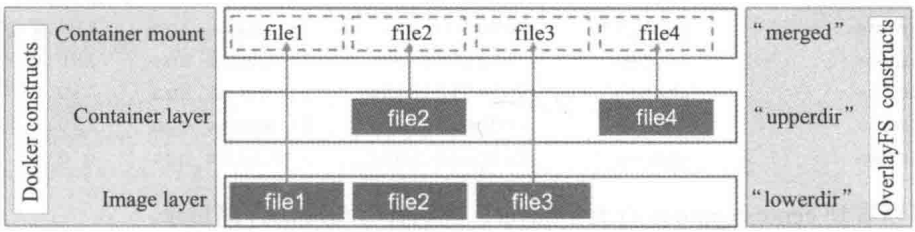
3. overlayfs

在内核 3.18 中，overlayfs 终于正式进入主线。相比 AUFS，overlayfs 设计简单，代码也很少，而且可以实现 pagecache 共享。这似乎是一个非常好的选择。于是，在这之后，Docker 社区开始转向将 overlayfs 作为第一选择。

8.3.2　Docker overlayfs driver

1. 介绍

Docker 使用 overlayfs 的 lowerdir 指向 image layer，使用 upperdir 指向 container layer，merged 将 lowerdir 与 upperdir 整合起来提供统一视图给容器，作为根文件系统。内容如下：



lowerdir 与 upperdir 可以包含相同的文件，upperdir 会隐藏 lowerdir 的文件。

（1）read file

在容器内读文件时，如果 upperdir(container layer) 存在，就从 container layer 读取；如果不存在，就从 lowerlay（image layer）读取。

（2）write file

写容器内文件时，如果 upperdir 不存在，overlay 则会发起 copy_up 操作，从 lowerdir 拷贝文件到 upperdir。由于拷贝发生在文件系统层面，而不是块层，会拷贝整个文件，即使只修改文件很小一部分，如果文件很大，也会导致效率低下。但好在拷贝只会在第一次打开时发生。另外，由于 overlay 只有两层，所以性能影响也很小。

（3）deleting files and directories

删除容器内文件时，upperdir 会创建一个 whiteout 文件，它会隐藏 lowerdir 的文件（不会删除）。同样，删除目录时，upperdir 会创建一个 opaque directory，隐藏 lowerdir 的目录。

2.overlayfs driver 实践

以 Docker 1.9.1 为例，指定 overlay driver 启动：

```
#docker daemon --storage-driver=overlay
# docker info
Containers: 0
Images: 0
```

```
Server Version: 1.9.1
Storage Driver: overlay
 Backing Filesystem: extfs
...
```

下载一个镜像：

```
# docker pull centos:centos6
# docker images  -a
REPOSITORY          TAG          IMAGE ID          CREATED          VIRTUAL SIZE
centos              centos6      1a895dd3954a      11 weeks ago     190.6 MB
<none>              <none>       366219586e86      11 weeks ago     190.6 MB
<none>              <none>       501f51238f9e      11 weeks ago     190.6 MB
<none>              <none>       ebdbe10e9b33      11 weeks ago     190.6 MB
<none>              <none>       fa5be2806d4c      3 months ago     0 B
```

可以看到 centos:centos6 有五个 layer，我们查看对应的存储目录：

```
# ls /var/lib/docker/overlay/
1a895dd3954aede5ea9e6bc23d23e8b1f6040df94647d83e71f96d60131d3235  ebdbe10e
9b3379125ce3c105cb711f80afdc22a5adac56f0045bc2c19f08887c
366219586e86f21918abb0571e668eb702b506d825702856539515ba2ac4be52  fa5be280
6d4c9aa0f75001687087876e47bb45dc8afb61f0c0e46315500ee144
501f51238f9ef52bcb6aecb6e2c1c04b3f8607c855d9b2cf7da780946ce02ec2

# ls /var/lib/docker/overlay/1a895dd3954aede5ea9e6bc23d23e8b1f6040df94647d
83e71f96d60131d3235/root/
bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root
sbin  selinux  srv  sys  tmp  usr  var
```

可以看到，每个 layer 对应一个目录。

我们创建一个容器，然后查看容器存储目录：

```
# docker run -it centos:centos6 /bin/bash
[root@b90c75273b11 /]#

# ls /var/lib/docker/overlay/b90c75273b116a4dac754f425380012bbdf90d098cdbc
829de3691f857137435
lower-id  merged  upper  work

# cat /var/lib/docker/overlay/b90c75273b116a4dac754f425380012bbdf90d098cdb
c829de3691f857137435/lower-id
1a895dd3954aede5ea9e6bc23d23e8b1f6040df94647d83e71f96d60131d3235

# cat /proc/mounts
overlay /var/lib/docker/overlay/b90c75273b116a4dac754f425380012bbdf90d09
```

```
8cdbc829de3691f857137435/merged overlay rw,relatime,lowerdir=/var/lib/docker/
overlay/1a895dd3954aede5ea9e6bc23d23e8b1f6040df94647d83e71f96d60131d3235/
root,upperdir=/var/lib/docker/overlay/b90c75273b116a4dac754f425380012bbdf90d09
8cdbc829de3691f857137435/upper,workdir=/var/lib/docker/overlay/b90c75273b116a4
dac754f425380012bbdf90d098cdbc829de3691f857137435/work 0 0
```

可以看到，容器对应的目录有三个（merged、upper、work），work 目录用于 overlayfs 实现 copy_up 操作，lower-id 保存 image ID。

（1）创建文件

在容器创建一个文件：

```
[root@b90c75273b11 ~]# echo "hello" > /root/f1.txt
[root@b90c75273b11 ~]# ls /root/
anaconda-ks.cfg  f1.txt  install.log  install.log.syslog
```

overlay 目录变化：

```
[root@yy1 ~]# ls /var/lib/docker/overlay/b90c75273b116a4dac754f425380012bb
df90d098cdbc829de3691f857137435/merged/root/
anaconda-ks.cfg  f1.txt  install.log  install.log.syslog
[root@yy1 ~]# ls /var/lib/docker/overlay/b90c75273b116a4dac754f425380012bb
df90d098cdbc829de3691f857137435/upper/root/
f1.txt
[root@yy1 ~]# ls /var/lib/docker/overlay/1a895dd3954aede5ea9e6bc23d23e8b1f
6040df94647d83e71f96d60131d3235/root/root/
anaconda-ks.cfg  install.log  install.log.syslog
```

可以看到文件出现在 upper 目录。

（2）删除文件

在容器删除一个文件：

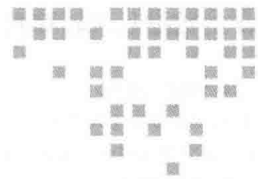
```
[root@b90c75273b11 ~]# rm /root/install.log
[root@b90c75273b11 ~]# ls /root/
anaconda-ks.cfg  f1.txt  install.log.syslog
[root@yy1 ~]# ls /var/lib/docker/overlay/b90c75273b116a4dac754f425380012bb
df90d098cdbc829de3691f857137435/merged/root/
anaconda-ks.cfg  f1.txt  install.log.syslog

[root@yy1 ~]# ls /var/lib/docker/overlay/b90c75273b116a4dac754f425380012bb
df90d098cdbc829de3691f857137435/upper/root/* -l
-rw-r--r-- 1 root root    6 12 月 31 17:55 /var/lib/docker/overlay/b90c7527
3b116a4dac754f425380012bbdf90d098cdbc829de3691f857137435/upper/root/f1.txt
c--------- 1 root root 0, 0 12 月 31 18:01 /var/lib/docker/overlay/b90c7527
3b116a4dac754f425380012bbdf90d098cdbc829de3691f857137435/upper/root/install.log
```

```
[root@yy1 ~]# ls /var/lib/docker/overlay/1a895dd3954aede5ea9e6bc23d23e8b1f
6040df94647d83e71f96d60131d3235/root/root/
anaconda-ks.cfg  install.log  install.log.syslog
```

可以看到 upper 目录多了一个“install.log”文件。

8.4 本章小结

数据卷是 Docker 应用中的重要的一环。通过数据卷，我们可以灵活地管理应用程序的数据，也可以通过数据卷在容器与容器、容器与 Host 之间共享数据。理解数据卷可以让我们更好地使用 Docker，让 Docker 为应用程序服务。



第 9 章 Chapter 9

Docker 项目日常维护

从第 4 章到第 8 章，我们陆续介绍了 Docker 的三大组件（容器、镜像和仓库）、网络和数据存储的基础知识。这一章，我们将介绍如何利用上述的基础知识，来维护 Docker 项目整个生命周期。

前面我们基于 Ubuntu 系统搭建 Docker 运行环境。从本章开始，我们再次切换环境到 REHL/Centos 系统下，因为接下来几章，我们将讨论在生产环境下如何运用 Docker，国内大部分的生产环境都是使用 REHL/Centos 系列的操作系统。

9.1 宿主机的管理

9.1.1 安装 Docker 并启动

安装：

```
yum install docker-io
```

启动：

```
service dokcer start
```

检查 docker 进程是否启动：

```
ps aux |grep docker
```

把 Dokcer 的数据目录转移到大的磁盘分区上。

```
service docker stop
mkdir /data/dockerData/
mv  /var/lib/docker /data/dockerData/
ln -s /data/dockerData/docker /var/lib/docker
service docker start
```

启动 Docker 时，*会自动分配* 172.17.42.1。

升级 Docker 到最新版本（如 docker-1.11.1）。

```
service docker stop
wget https://get.docker.com/builds/Linux/x86_64/docker-1.11.1.tgz
tar zxvf docker-1.11.1.tgz
cp docker/docker /usr/bin/docker
chmod +x  /usr/bin/docker
service docker start
```

Docker 监听内网的配置文件 /etc/sysconfig/docker 内容如下：

```
#!/bin/bash
NUM=$(ifconfig|grep 'inet addr'|awk -F':' '{print $2}'|awk '{print $1}'|grep
-v '127.0.0.1' |grep -v '172.17.42.1' |egrep '^10|^172'|wc -l)
if [ $NUM -eq 1 ];then
    HOSTIP=$(ifconfig|grep 'inet addr'|awk -F':' '{print $2}'|awk '{print
$1}'|grep -v '127.0.0.1' |grep -v '172.17.42.1' |egrep '^10|^172')
else
i   HOSTIP="127.0.0.1"
fi
other_args="-H tcp://${HOSTIP}:2375 -H unix:///var/run/docker.sock
--insecure-registry 10.100.10.2:5000  -dns 10.100.10.3"
```

其中，“--insecure-registry”指定内部 Docker 仓库的 IP 和端口，“-dns”指定内部 DNS 服务器的地址。

/etc/sysctl.conf 中添加如下一行内容：

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

使配置生效：

```
sysctl -p
```

9.1.2　网桥模式

默认网络配置的是 NAT 模式，如果要配置网桥模式，需要增加以下三步。

1. 宿主机配网桥

备份原来网卡配置：

```
cp  /etc/sysconfig/network-scripts/ifcfg-eth1 /root
```

修改 /etc/sysconfig/network-scripts/ifcfg-eth1：

```
DEVICE='eth1'
HWADDR=00:15:17:d8:cc:c6
ONBOOT=yes
BRIDGE=br1
```

修改 /etc/sysconfig/network-scripts/ifcfg-br1：

```
DEVICE='br1'
TYPE=Bridge
BOOTPROTO=static
ONBOOT=yes
IPADDR='10.100.10.xx'
NETMASK='255.255.255.192'
GATEWAY='10.100.10.129'
```

其中 IPADDR、NETMASK、GATEWAY 是宿主机的 IP、子网掩码和网关，根据实际情况进行配置。

然后重启网卡使配置生效。

```
/etc/init.d/network  restart
```

2. 安装 pipework 脚本

pipework 用于给 container 指定 IP。

```
wget  -O /usr/bin/pipework  https://github.com/jpetazzo/pipework/blob/master/pipework
Chmod +x /usr/bin/pipework
```

3. 更新 iproute

```
yum install iproute
```

9.2 GitLab 的日常维护

前面我们已经介绍过 GitLab 项目有三个容器，通过 Docker Compose 组件对这三个容器的启动顺序进行管理。但这还远远不够，下面我们针对 GitLab 这个例子，讲一下 Docker 项目维护所需要的常用操作和注意事项。

9.2.1 项目的创建

我们原来已经创建过 GitLab 项目，配置文件在 ~/gitlab/docker-compose.yml 下，先把它删除。

```
docker-compose  -f ~/gitlab/docker-compose.yml down
```

接着备份旧的 docker-compose.yml 文件。下载最新的 docker-compose.yml 文件。

```
cd ~/gitlab
mv docker-compose.yml docker-compose.yml_v1.0
wget -O docker-compose.yml \  https://raw.githubusercontent.com/sameersbn/docker-gitlab/master/docker-compose.yml
```

然后，通过 docker-compose up 就可以创建并启动容器组了。

下面看看最新的 docker-compose.yml 文件的内容。

```
postgresql:
  restart: always
  image: sameersbn/postgresql:9.4-13
  environment:
    - DB_USER=gitlab
    - DB_PASS=password
    - DB_NAME=gitlabhq_production
  volumes:
    - /srv/docker/gitlab/postgresql:/var/lib/postgresql
gitlab:
  restart: always
  image: sameersbn/gitlab:8.4.4-1
  links:
    - redis:redisio
    - postgresql:postgresql
  ports:
    - "10080:80"
    - "10022:22"
  environment:
    - DEBUG=false
```

```
    - TZ=Asia/Kolkata
    - GITLAB_TIMEZONE=Kolkata

    - GITLAB_SECRETS_DB_KEY_BASE=long-and-random-alphanumeric-string

    - GITLAB_HOST=localhost
    - GITLAB_PORT=10080
    - GITLAB_SSH_PORT=10022
    - GITLAB_RELATIVE_URL_ROOT=

    - GITLAB_NOTIFY_ON_BROKEN_BUILDS=true
    - GITLAB_NOTIFY_PUSHER=false

    - GITLAB_EMAIL=notifications@example.com
    - GITLAB_EMAIL_REPLY_TO=noreply@example.com
    - GITLAB_INCOMING_EMAIL_ADDRESS=reply@example.com

    - GITLAB_BACKUP_SCHEDULE=daily
    - GITLAB_BACKUP_TIME=01:00

    - SMTP_ENABLED=false
    - SMTP_DOMAIN=www.example.com
    - SMTP_HOST=smtp.gmail.com
    - SMTP_PORT=587
    - SMTP_USER=mailer@example.com
    - SMTP_PASS=password
    - SMTP_STARTTLS=true
    - SMTP_AUTHENTICATION=login

    - IMAP_ENABLED=false
    - IMAP_HOST=imap.gmail.com
    - IMAP_PORT=993
    - IMAP_USER=mailer@example.com
    - IMAP_PASS=password
    - IMAP_SSL=true
    - IMAP_STARTTLS=false
  volumes:
    - /srv/docker/gitlab/gitlab:/home/git/data
redis:
  restart: always
  image: sameersbn/redis:latest
  volumes:
    - /srv/docker/gitlab/redis:/var/lib/redis
```

和原来的 docker-compose.yml 文件相比，新增了如下几点：

- 该配置文件容器书写的顺序是 postgresql、gitlab 和 redis，我们知道 gitlab 依赖 postgresql 和 redis，postgresql 和 redis 启动优先级高于 gitlab，所以，配置文件的书写顺序和容器实际启动的顺序没有关系，Docker Compose 通过“links”等带有逻辑关系的选项确定容器启动的优先级。
- restart：always 容器如果异常退出后会自动重启，这主要应用于生成环境，保证不间断提供服务。
- 这个配置文件最大的改动是使用“volumes”选项，通过该选项，我们把重要的文件的存储位置从容器内转移到容器外部的宿主机上。这样，从容器的角度来看，它就像一个系统分区使用挂载的方式挂载到容器的文件系统上，使用上不受任何影响。对于宿主机来说，这个文件就是本机文件，可以很方便地查阅备份。使用“volumes”选项方式还有一个好处是：让容器和重要数据分离，容器的更新或删除，不会造成重要数据的丢失。

9.2.2 代码版本控制

GitLab 服务器端使用公私钥方式来认证客户端 git 的请求。

1. 添加 ssh 密钥

客户端使用 ssh 的方式和 GitLab 交互，在 http:ip:10080/profile/keys/new 添加公钥。公钥制作方法：

```
ssh-keygen -t rsa -C "$your_email"
cat ~/.ssh/id_rsa.pub
```

生成公钥时，可以设置密码为空。具体可参考：官方文档公钥设置 http://doc.gitlab.com/ce/ssh/README.html。

但有时会出现如下问题：添加公钥后，有时仍要求输入密码的情况。这是由于添加的公钥出现了问题。

解决方法：登录到 Docker 容器中（docker exec -it gitlab bash），把 /home/git/.ssh 下的 authorized_keys 内容清空，删除其余文件，然后重新添加公钥。

2. 创建新仓库

接着，创建 group，在该组下创建项目，然后在 git client 端设置 git 全局配置。

```
git config --global user.name "xxx"
```

```
git config --global user.email "xxx@163.com"
```

创建仓库并上传到 GitLab。

```
GITHOST=xxx.xxx.xxx.xxx
GITPROJECT=docker-book
mkdir $GITPROJECT
cd $GITPROJECT
git init
touch README.md
git add README.md
git commit -m "first commit"
git remote add  origin ssh://git@${GITHOST}:10022/mydocker/${GITPROJECT}.git
git push -u origin master
```

如果 git 项目已存在，可以通过下面的方式直接 push 到 gitlab 上。

```
GITHOST=xxx.xxx.xxx.xxx
GITPROJECT=docker-book
cd $GITPROJECT
git remote add origin ssh://git@{GITHOST}:10022/mydocker/${GITPROJECT}.git
git push -u origin master
```

如推送时报如下错误，原因是 git remote 的 url 有误。

```
$ git push -u origin master
fatal: protocol error: bad line length character: No s
```

通过 git remote -v 检查下 url，有误的话，先删“git remote rm xxx”再重建。

```
git remote add xxx ssh://git@IP:10022/xxx/xxxx.git
```

9.2.3　日常维护

1. 备份

执行下面命令备份：

```
docker run \
  --name='gitlab_backup' \
  -it \
  --rm \
  --link postgresql:postgresql \
    --link redis:redisio \
    -e 'GITLAB_PORT=10080' -e 'GITLAB_SSH_PORT=10022' \
    -v /data/gitlabServer/gitlab/data:/home/git/data \
    sameersbn/gitlab:7.6.1 app:rake gitlab:backup:create
```

备份的文件保存到 /data/gitlabServer/gitlab/data/backups 目录下。

2. 恢复

通过“BACKUP=xxx”指定恢复到哪个版本。

```
docker run \
  --name='gitlab_restore' \
  -it \
  --rm \
  --link postgresql:postgresql \
  --link redis:redisio \
  -e 'GITLAB_PORT=10080' -e 'GITLAB_SSH_PORT=10022' \
  -v /data/gitlabServer/gitlab/data:/home/git/data \
  sameersbn/gitlab:7.6.1  app:rake gitlab:backup:restore  BACKUP=1427203299
```

恢复时会将当前数据库中的所有表先删掉再导入备份 tar 包的里 sql 文件，因此要小心。

若 redis、mysql 是使用环境变量带入 gitlab 容器的，备份和恢复命令也类似，将启动 gitlab 的命令复制过来，修改 --name，添加一个 --rm，CMD 改为 gitlab:backup:create 或 gitlab:backup:restore 即可。

不同版本备份的文件不能相互使用。

如是在 gitlab7.6.1 下备份的文件，在 gitlab7.9.0 下使用，报如下错误。

```
GitLab version mismatch:
  Your current GitLab version (7.9.0) differs from the GitLab version in the backup!
  Please switch to the following version and try again:
  version: 7.6.1
```

在 linode 上备份 gitlab7.9.0 时报如下错误。

```
Running gitlab rake task...
** Invoke gitlab:backup:create (first_time)
** Invoke environment (first_time)
```

```
** Execute environment
rake aborted!
Errno::ENOMEM: Cannot allocate memory - git
/usr/lib/ruby/2.1.0/open3.rb:193:in `spawn'
/usr/lib/ruby/2.1.0/open3.rb:193:in `popen_run'
/usr/lib/ruby/2.1.0/open3.rb:93:in `popen3'
/home/git/gitlab/lib/gitlab/popen.rb:23:in `popen'
```

先停掉 gitlab 再执行备份，就正常了。

3. 升级

假如从 GitLab7.6.1 升级到 7.9.0，先删除原来的 GitLab。

```
docker rm -f gitlab
```

备份老版本：

```
docker run --name=gitlab_backup -it --rm --link postgresql:postgresql \
  --link redis:redisio \
  -e 'GITLAB_PORT=10080' -e 'GITLAB_SSH_PORT=10022' \
  -v /data/gitlabServer/gitlab/data:/home/git/data \
  sameersbn/gitlab:7.6.1  app:rake gitlab:backup:create
```

下载并启用新版本：

```
docker run --name=gitlab -d --link postgresql:postgresql \
--link redis:redisio \
-e 'GITLAB_PORT=10080' -e 'GITLAB_SSH_PORT=10022' \
-p 10022:22 -p 10080:80 \
-v /data/gitlabServer/gitlab/data:/home/git/data \
sameersbn/gitlab:7.9.0
```

备份新版本。

```
docker run --name=gitlab_backup -it --rm --link postgresql:postgresql \
  --link redis:redisio \
  -e 'GITLAB_PORT=10080' -e 'GITLAB_SSH_PORT=10022' \
  -v /data/gitlabServer/gitlab/data:/home/git/data \
  sameersbn/gitlab:7.9.0  app:rake gitlab:backup:create
```

4. 迁移

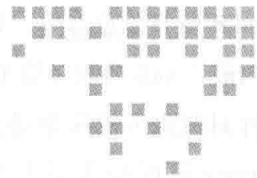
在机器 A 上执行备份操作后，在 /data/gitlabServer/gitlab/data/backups 目录下生成的 1427250996_gitlab_backup.tar 在机器 B 上，机器 A 和机器 B 的 GitLab 必须保持一致。

在机器 B 上启动 GitLab，然后把机器 A 的备份导入机器 B。方法如下：

```
docker run \
  --name='gitlab_restore' \
  -it \
  --rm \
  --link postgresql:postgresql \
  --link redis:redisio \
  -e 'GITLAB_PORT=10080' -e 'GITLAB_SSH_PORT=10022' \
  -v /data/gitlabServer/gitlab/data:/home/git/data \
  sameersbn/gitlab:7.9.0  app:rake gitlab:backup:restore  BACKUP=1427250996
```

9.3 本章小结

本章主要介绍了宿主机如何初始化 Docker 运行环境，并结合 GitLab 项目讲述如何维护 Docker 项目备份、升级和迁移等常用操作。



第 10 章 Chapter 10

Docker Swarm 容器集群

Docker Swarm 项目开始于 2014 年，是 Docker 公司推出的第一个容器集群项目。项目的核心设计是将几台安装 Docker 的机器组合成一个大的集群，该集群提供给用户管理集群所有容器的操作接口与使用一台 Docker 几乎相同。

Docker Swarmkit 项目开始于 2016 年，是 Docker 公司推出的第二个容器集群项目，于 Docker1.12 版本正式发布。虽然也叫 Swarm，但是与第一个项目完全不同。该项目直接在 Docker Engine 上内嵌了集群管理功能，并新增了集群管理的用户接口。

两个容器集群项目可能实现了相同的功能，但其上层接口还是有很大的不同，Docker 公司推荐用户使用更适合自己的项目，如果都没有使用过，推荐使用后者。另外，Docker Swarm 项目并没有被 Docker 公司列为不推荐的项目，仍然会继续支持新的 Docker Engine 的功能。本章重点介绍 Docker Swarmkit 项目。

10.1 Swarmkit 核心设计

Swarmkit 的架构图如图 10-1 所示，项目目前的核心设计包括：

- Docker Engine 内嵌 Swarmkit 提供集群管理，除了安装 Docker Engine 外无须安装其他任何软件。使用 Docker Engine 新增的 Docker Swarm 模式客户端接口管理集群。

- Swarmkit 所有节点对等，每个节点可选择转化为 Manager 或者 Worker。Manager 节点内嵌了 raft 协议（基于 etcd 的 raft 协议）实现高可用，并存储集群状态。
- 集群针对微服务的模型进行设计，Service 表示作业，Task 表示作业的副本。一个 Service 可以包含多个 Task，每个 Task 是一个容器，同一个 Service 的所有 Task 状态对等。
  - 声明式的 Service 状态定义，Service 的提交配置定义了 Task 希望维持的状态。
  - 支持 Task 的扩容和缩容。
  - 自动容错，一个 Worker 节点挂了，容器自动迁移到其他 Worker 节点。
  - 支持灰度升级。
- 支持跨主机的网络模型。
  - 依赖 Libnetwork 项目实现集群网络。
  - 基于 Vxlan 协议实现 SDN。
  - 使用 Docker NAT 访问外网。
  - 基于 DNS 服务与 LVS 技术实现服务发现和负载均衡。
- 安全方面，每个节点使用对等的 TLS 相互通信。
  - TLS 证书是周期滚动的，由 Manager 节点下发。

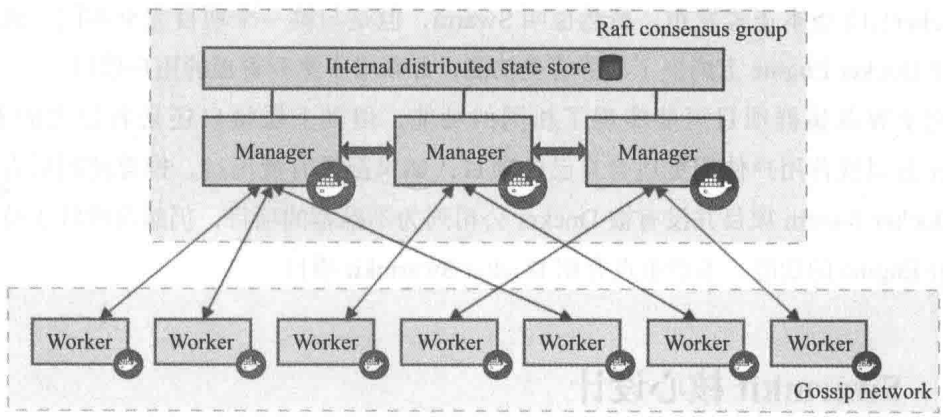


图 10-1 Swarmkit 架构

10.2 Swarmkit 集群搭建

以 1 个 Manager 节点和 2 个 Worker 节点组成的集群为例。首先准备 3 台机器或者

虚拟机，安装好 Docker Engine 1.12 版本并启动。这里使用 Docker Machine 创建的虚拟机为例进行介绍，读者可以根据实际情况进行调整。

使用 Docker Machine 创建三台虚拟机，Docker Machine 会自动下载最新的 boot2docker.iso，启动的虚拟机已经安装好 Docker 1.12 版本。

```
$ docker-machine create -d virtualbox manager1
$ docker-machine create -d virtualbox worker1
$ docker-machine create -d virtualbox worker2

$ docker-machine ls
NAME       ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER   ERRORS
manager1   *        virtualbox   Running   tcp://192.168.99.101:2376           v1.12.1
worker1    -        virtualbox   Running   tcp://192.168.99.102:2376           v1.12.1
worker2    -        virtualbox   Running   tcp://192.168.99.103:2376           v1.12.1
```

10.2.1　创建 Manager 节点

切换到 Manager1 节点，执行 docker swarm init --advertise-addr <MANAGER-IP> 命令创建新的 Swarm 集群，Manager1 节点的 Docker 成为 Manager 角色。

```
# 切换到 manager1 的 docker 环境
$ eval $(docker-machine env manager1)

$ docker swarm init --advertise-addr 192.168.99.101
Swarm initialized: current node (9p6hf9w8mnqkxzdby03si4b22) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-06fg2v27725iy8le0aj13jfaywta7b7ua8blltln77bwzoil6e-
bjvj2kyr88c8na67uz173kepc \
    192.168.99.101:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and
follow the instructions.
```

--advertise-addr 参数定义 Manager 节点使用 192.168.99.101 作为自己的 IP。docker swarm init 命令的输出非常友好，提示了用户如何将另外一个节点加入集群。

创建 Manager 节点后我们可以使用 docker info，docker node ls 命令查看 Manager 节点的状态。docker info 命令新增加了 Swarm 模式下的集群简要配置和状态信息。docker node ls 命令可以查看集群所有 Manager 和 Worker 节点的状态。我们可以从下面

的输出看到当前集群只有一个 Manager 节点。

```
$ docker info
...
Swarm: active
  NodeID: 9p6hf9w8mnqkxzdby03si4b22
  Is Manager: true
  ClusterID: 5umhbh08rzg11szvxd7eh9nba
  Managers: 1
  Nodes: 3
  Orchestration:
    Task History Retention Limit: 5
  Raft:
    Snapshot Interval: 10000
    Heartbeat Tick: 1
    Election Tick: 3
  Dispatcher:
    Heartbeat Period: 5 seconds
  CA Configuration:
    Expiry Duration: 3 months
  Node Address: 192.168.99.101
...

$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
9p6hf9w8mnqkxzdby03si4b22 *  manager1  Ready   Active        Leader
```

10.2.2 创建 Worker 节点

接下来切换到 Worker1 和 Worker2 虚拟机节点，执行 docker swarm join -token <TOKEN> <MANAGER-IP> 创建集群的 Worker 节点。

```
$ eval $(docker-machine env worker1)
$ docker swarm join --token SWMTKN-1-06fg2v27725iy81e0aj13jfaywta7b7ua8bll
tln77bwzoil6e-bjvj2kyr88c8na67uz173kepc 192.168.99.101:2377
This node joined a swarm as a worker.

$ eval $(docker-machine env worker2)
$ docker swarm join --token SWMTKN-1-06fg2v27725iy81e0aj13jfaywta7b7ua8bll
tln77bwzoil6e-bjvj2kyr88c8na67uz173kepc 192.168.99.101:2377
This node joined a swarm as a worker.
```

--token 参数的值是从上一步创建 Manager 节点的输出获取的，192.168.99.101:2377 是 Manager 节点的地址。如果没有保存上一步输出的 token，可以切换到 Manager 节点执行 docker swarm join-token worker 获取。

同样在 Manager 节点使用 docker node ls 打印集群的节点状态，可以看到现在集群有 1 个 Manager 节点和 2 个 Worker 节点。

```
$ eval $(docker-machine env manager1)
$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
2p07afdsfe8tabwpz27sqlrlf    worker2   Ready   Active
9p6hf9w8mnqkxzdby03si4b22 *  manager1  Ready   Active        Leader
cumt545vk2a10wagf026qiehm    worker1   Ready   Active
```

10.3 Swarmkit 基本功能

下面简单介绍 Swarmkit 集群 service 的基本功能。

10.3.1 service 创建与删除

切换到 manager 节点，使用 docker service create 命令创建 service。

```
$ docker service create --replicas 1 --name helloworld alpine ping docker.com
6kai8eak653bhuaomofkni7kt
```

--replicas 参数指定创建 1 个保持运行的 task。

我们可以使用 docker service ls 命令查看所有的 service 列表。

```
$ docker service ls
ID            NAME        REPLICAS  IMAGE   COMMAND
6kai8eak653b  helloworld  1/1       alpine  ping docker.com
```

使用 docker service ps 和 docker service inspect 命令可以查看 service 的简略和详细信息。

```
$ docker service inspect --pretty helloworld
ID:             6kai8eak653bhuaomofkni7kt
Name:           helloworld
Mode:           Replicated
  Replicas:     1
Placement:
UpdateConfig:
  Parallelism:  1
  On failure:   pause
ContainerSpec:
  Image:        alpine
  Args:         ping docker.com
```

```
    Resources:

    $ docker service ps helloworld
    ID                         NAME          IMAGE   NODE      DESIRED STATE
CURRENT STATE           ERROR
    en20bib4yn8jjkg61b8qnsbbd  helloworld.1  alpine  manager1  Running
Running 11 minutes ago
```

使用 docker service rm 命令可以删除 service。

```
    $ docker service rm helloworld
```

10.3.2 service 扩容与缩容

使用 docker service scale 命令对 service 进行扩容或者缩容。下面是将上一步创建的 service 扩容到 3 个 task 的代码示例。

```
    $ docker service scale helloworld=3
    helloworld scaled to 3
    $ docker service ps helloworld
    ID                         NAME          IMAGE   NODE      DESIRED STATE
CURRENT STATE           ERROR
    en20bib4yn8jjkg61b8qnsbbd  helloworld.1  alpine  manager1  Running
Running 14 minutes ago
    d9fpsza181dwuu3fx6weebki6  helloworld.2  alpine  worker2   Running
Preparing 5 seconds ago
    b91vkmv2oqoavfzp1zs7n8xfd  helloworld.3  alpine  worker1   Running
Preparing 5 seconds ago
```

10.3.3 service 灰度升级

首先使用 3.0.6 版本的 redis 镜像创建一个 3 个 task 的 redis service。

```
    $ docker service create --replicas 3 --name redis --update-delay 10s redis:3.0.6
    4tip9e8p9us14s634ncbsv6y0
```

--update-delay 参数配置 service 灰度升级的时间间隔，默认 scheduler 一次只升级一个 task，可以同时使用 --update-parallelism 参数配置并发升级的 task 数。查看 redis service 是否已经启动。

```
    $ docker service inspect --pretty redis
    ID:             4tip9e8p9us14s634ncbsv6y0
    Name:           redis
    Mode:           Replicated
      Replicas:     3
```

```
    Placement:
    UpdateConfig:
      Parallelism:    1
      Delay:          10s
      On failure:     pause
    ContainerSpec:
      Image:          redis:3.0.6
    Resources:
```

确定所有 task 已经启动后，使用 docker service update 命令将 redis service 的所有 task 升级到 3.0.7 版本，--image 参数指定升级的版本。

```
$ docker service update --image redis:3.0.7 redis
redis
```

使用 docker service ps 命令查看 redis service 升级前后 task 的状态变化。下面的操作输出中我们可以看到 redis.1 从 Manager1 节点迁移到了 Worker2 节点，并完成了 task 的镜像升级。

```
$ docker service ps redis
ID                         NAME          IMAGE         NODE      DESIRED
STATE  CURRENT STATE               ERROR
721wjyobwdv4temqryifdb216  redis.1       redis:3.0.7   worker2   Running
Running 3 seconds ago
54cbfx39xtttn0z5802gttc0j   \_ redis.1   redis:3.0.6   manager1  Shutdown
Shutdown about a minute ago
86rr3cnbqk7d5ib770r4ko3pd  redis.2       redis:3.0.6   worker2   Running
Running about a minute ago
cb9p34evyo51iya1p5hkc0ex6  redis.3       redis:3.0.6   worker1   Running
Running 3 seconds ago
```

10.3.4　service 网络配置、域名解析和负载均衡

创建 service 时使用 --publish 参数配置容器 NAT 网络的端口映射。

```
$ docker service create --name my_web --replicas 3 --publish 8080:80 nginx
1so3f1p7iphhj2ccxmvyin871

$ docker-machine ssh manager1
docker@manager1:~$ curl http://192.168.99.101:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

Swarmkit 也提供了 Overlay 的跨主机网络，使用 docker network create 创建 overlay 网络，操作命令与 docker engine 相同。使用 docker network ls 查看集群网络列表，docker network inspect 查看某个网络的详细信息。下面的命令创建了一个名为 my-network 的 Overlay 网络。

```
$ docker network create --driver overlay my-network
5yg4chi6b6xj8dsb0bi8zl0vw
```

创建了 Overlay 网络后，我们可以在创建 service 时使用 --network 参数配置容器加入我们创建的 Overlay 网络。

```
$ docker service create --replicas 3 --network my-network --name my-web nginx
```

默认情况下，将 service 接入 Overlay 网络时，Swarm 会给 service 分配一个 VIP，VIP 与一个包含 service 名称的 DNS 记录形成映射关系，这个 service 的所有 container 共享这条 DNS 记录，Swarm 也会创建一个 load balance 将访问 VIP 的流量均衡到所有的 task 上。我们再启动另一个 service 加入 my-network 网络体验 Swarm 提供的域名解析服务。

```
$ docker service create --name my-busybox --network my-network busybox sleep 3000
0urwetl5jfs9cphq7ggynja2y

$ docker service ps my-busybox
ID                         NAME              IMAGE    NODE      DESIRED
STATE  CURRENT STATE            ERROR
0urwetl5jfs9cphq7ggynja2y  my-busybox.1      busybox  manager1  Running
Running 12 minutes ago
```

使用 docker exec 进入容器查询以查询这个 DNS 记录。直接查询 service 名称的域名返回这个 service 的 VIP，查询 tasks.<service name>DNS 记录返回所有 task 的 IP。

```
$ eval $(docker-machine env manager1)
$ docker exec -it my-busybox.1.0urwetl5jfs9cphq7ggynja2y sh

/ # nslookup my-web
Server:    127.0.0.11
Address 1: 127.0.0.11

Name:      my-web
Address 1: 10.0.0.2

/ # nslookup tasks.my-web
```

```
Server:    127.0.0.11
Address 1: 127.0.0.11

Name:      tasks.my-web
Address 1: 10.0.0.4 my-web.2.14hggmn6m8rucruo2omt8wygt.my-network
Address 2: 10.0.0.5 my-web.3.ehfb5ue134nyasq2g539uaa6g.my-network
Address 3: 10.0.0.3 my-web.1.5yzoighbalqvio4djic464a9j.my-network

/ # wget -O- my-web
Connecting to my-web (10.0.0.2:80)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

10.3.5　Swarmkit 节点管理

我们使用 drain worker 以便对节点做一些运维操作，比如换机器。首先查看上一步灰度升级后的 redis service，有一个 task 在 worker2 节点上。

```
$ docker service ps redis
ID                         NAME          IMAGE         NODE      DESIRED
STATE  CURRENT STATE           ERROR
721wjyobwdv4temqryifdb216  redis.1       redis:3.0.7   worker2   Running
Running 5 minutes ago
54cbfx39xtttn0z5802gttc0j   \_ redis.1   redis:3.0.6   manager1  Shutdown
Shutdown 6 minutes ago
15vhgz93khpgt7aak2uwiikfq  redis.2       redis:3.0.7   manager1  Running
Running 4 minutes ago
86rr3cnbqk7d5ib770r4ko3pd   \_ redis.2   redis:3.0.6   worker2   Shutdown
Shutdown 5 minutes ago
a8pdozzumncwffbr7k69zzsdd  redis.3       redis:3.0.7   manager1  Running
Running 3 minutes ago
cb9p34evyo51iya1p5hkc0ex6   \_ redis.3   redis:3.0.6   worker1   Shutdown
Shutdown 3 minutes ago
```

使用 docker node update 命令对 worker2 节点进行下线操作。--availability drain 表示 worker 节点为不可用状态。

```
$ docker node update --availability drain worker2
worker2
```

使用 docker node inspect 命令查看 worker2 节点的可用性是否处于 drain 状态。

```
$ docker node inspect --pretty worker2
```

```
ID:          2p07afdsfe8tabwpz27sqlrlf
Hostname:        worker2
Joined at:       2016-09-19 07:13:53.507177094 +0000 utc
Status:
  State:          Ready
  Availability:       Drain
Platform:
  Operating System:  linux
  Architecture:       x86_64
Resources:
  CPUs:           1
  Memory:         995.9 MiB
Plugins:
    Network:       bridge, host, null, overlay
    Volume:        local
Engine Version:      1.12.1
Engine Labels:
  - provider = virtualbox
```

worker2 节点变为 drain 状态后，scheduler 会把 drain 状态节点上的 task 迁移到其他节点。片刻后，观察 task 的迁移状况，可以发现之前运行在 worker2 节点的 redis.1 task 已经迁移到 worker1 节点了。

```
$ docker service ps redis
ID                         NAME          IMAGE         NODE      DESIRED
STATE  CURRENT STATE              ERROR
77digxshezgp1v5k4pfxzl863  redis.1       redis:3.0.7   worker1   Running
Preparing 21 seconds ago
721wjyobwdv4temqryifdb216   \_ redis.1   redis:3.0.7   worker2   Shutdown
Shutdown 21 seconds ago
54cbfx39xtttn0z5802gttc0j   \_ redis.1   redis:3.0.6   manager1  Shutdown
Shutdown 7 minutes ago
15vhgz93khpgt7aak2uwiikfq  redis.2       redis:3.0.7   manager1  Running
Running 4 minutes ago
86rr3cnbqk7d5ib770r4ko3pd   \_ redis.2   redis:3.0.6   worker2   Shutdown
Shutdown 6 minutes ago
a8pdozzumncwffbr7k69zzsdd  redis.3       redis:3.0.7   manager1  Running
Running 4 minutes ago
cb9p34evyo51iya1p5hkc0ex6   \_ redis.3   redis:3.0.6   worker1   Shutdown
Shutdown 4 minutes ago
```

既然能将节点设置为不可用状态，那么我们也能将它重新设置为可用状态，docker node update --availability active worker 命令可以重新将 drain node 恢复为可用节点。恢复完成后，使用 docker node inspect 命令可以查看到节点状态切换为 Active 状态。

```
$ docker node update --availability active worker2
worker2

$ docker node inspect --pretty worker2
ID:          2p07afdsfe8tabwpz27sqlrlf
Hostname:        worker2
Joined at:       2016-09-19 07:13:53.507177094 +0000 utc
Status:
  State:         Ready
  Availability:      Active
Platform:
  Operating System:  linux
  Architecture:      x86_64
Resources:
  CPUs:          1
  Memory:        995.9 MiB
Plugins:
    Network:      bridge, host, null, overlay
    Volume:       local
Engine Version:     1.12.1
Engine Labels:
  - provider = virtualbox
```

10.3.6　Manager 节点和 Worker 节点角色切换

Docker Swarm 模式提供了 promote/demote 命令对节点的角色进行管理，方便对 Manager 节点进行容灾处理。docker node promote 命令将 Worker 节点升级为 Manager 节点。

```
$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
14e2n6kebft5hs8arc5njm7ti *  manager1  Ready   Active        Leader
aqbfvym02d85exu7i8th9yklo    worker1   Ready   Active
bjfb223324jjle3fprhvxg7of    worker2   Ready   Active

$ docker node promote worker1
Node worker1 promoted to a manager in the swarm.

$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
14e2n6kebft5hs8arc5njm7ti *  manager1  Ready   Active        Leader
aqbfvym02d85exu7i8th9yklo    worker1   Ready   Active        Reachable
bjfb223324jjle3fprhvxg7of    worker2   Ready   Active
```

相反的，使用 docker node demote 命令可以将 Manager 节点降级为 Worker 节点。

```
$ docker node demote worker1
Manager worker1 demoted in the swarm.

$ docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
14e2n6kebft5hs8arc5njm7ti *   manager1  Ready   Active        Leader
aqbfvym02d85exu7i8th9yklo     worker1   Ready   Active
bjfb223324jjle3fprhvxg7of     worker2   Ready   Active
```

使用 docker swarm leave 命令可以将节点的 docker engine 退出 swarm 状态。

```
$ docker swarm leave
Node left the swarm.
```

10.4 Swarmkit 负载均衡原理分析

按照上一节介绍的负载均衡的内容，我们首先创建一个 Overlay 网络，再创建一个包含 3 个 task 的 service 加入这个网络。

```
$ docker network create --driver overlay my-network
$ docker service create --replicas 3 --network my-network --name my-web nginx
$ docker service create --name my-busybox --network my-network busybox sleep 3000
```

4 个 task 的基本信息如表 10-1 所示。

表 10-1 4 个 task 的基本信息

service	container	ip vip	mac	node
my-web	5cca2a34de2b	10.0.0.5 10.0.0.2	02:42:0a:00:00:05	worker1
my-web	c00fffd5faba	10.0.0.3 10.0.0.2	02:42:0a:00:00:03	worker2
my-web	e6001f9a802e	10.0.0.4 10.0.0.2	02:42:0a:00:00:04	worker2
my-busybox	8b876998b1c2	10.0.0.7 10.0.0.6	02:42:0a:00:00:07	worker2

分析网络原理最直接的方法还是使用 Linux 提供的 ip/bridge/iptables/ipvs 等网络相关的命令去查看网络的配置。分别使用这些命令在各节点的 host/overlay/container network namespace 查看一遍。

```
ip link/address/route/neigh
ip netns exec overlay_namespace/container_namespace
bridge fdb
iptables filter/nat/mangle
ipvs
```

由结果可以发现，在原来的 Overlay 网络配置基础上，多了下面的一些配置。

- 四个容器的 network namespace 都增加了 ipvs 配置，并且四个容器 ipvs 配置都相同。

```
root@worker1:/home/docker# ip netns exec 5cca2a34de2b ipvsadm
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
FWM  257 rr
  -> 10.0.0.3:0                   Masq    1      0          0
  -> 10.0.0.4:0                   Masq    1      0          0
  -> 10.0.0.5:0                   Masq    1      0          0
FWM  260 rr
  -> 10.0.0.7:0                   Masq    1      0          0
```

- 容器 nat 表的 POSTROUTING 链增加了一条 ipvs 规则 -A POSTROUTING -d 10.0.0.0/24 -m ipvs --ipvs -j SNAT --to-source 10.0.0.5（127.0.0.11 的规则是 docker dns 服务增加的规则），每个容器这条规则的 --to-source 都是容器自己的 IP。

```
root@worker1:/home/docker# ip netns exec 5cca2a34de2b iptables -S -t nat
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER_OUTPUT
-N DOCKER_POSTROUTING
-A OUTPUT -d 127.0.0.11/32 -j DOCKER_OUTPUT
-A POSTROUTING -d 127.0.0.11/32 -j DOCKER_POSTROUTING
-A POSTROUTING -d 10.0.0.0/24 -m ipvs --ipvs -j SNAT --to-source 10.0.0.5
-A DOCKER_OUTPUT -d 127.0.0.11/32 -p tcp -m tcp --dport 53 -j DNAT --to-
destination 127.0.0.11:38534
-A DOCKER_OUTPUT -d 127.0.0.11/32 -p udp -m udp --dport 53 -j DNAT --to-
destination 127.0.0.11:39353
-A DOCKER_POSTROUTING -s 127.0.0.11/32 -p tcp -m tcp --sport 38534 -j SNAT
--to-source :53
-A DOCKER_POSTROUTING -s 127.0.0.11/32 -p udp -m udp --sport 39353 -j SNAT
--to-source :53
```

- 容器 mangle 表 OUTPUT 链增加了 mark 规则，四个容器规则相同。

```
root@worker1:/home/docker# ip netns exec 5cca2a34de2b iptables -S -t mangle
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
```

```
-A OUTPUT -d 10.0.0.2/32 -j MARK --set-xmark 0x101/0xffffffff
-A OUTPUT -d 10.0.0.6/32 -j MARK --set-xmark 0x106/0xffffffff
```

❑ 容器 Overlay veth 网卡多了一个 secondary IP，IP 地址是各个 service 的 VIP。

```
root@worker1:/home/docker# ip netns exec 5cca2a34de2b ip ad
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
      valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
      valid_lft forever preferred_lft forever
25: eth0@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
state UP group default
    link/ether 02:42:0a:00:00:05 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.5/24 scope global eth0
      valid_lft forever preferred_lft forever
    inet 10.0.0.2/32 scope global eth0
      valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe00:5/64 scope link
      valid_lft forever preferred_lft forever
27: eth1@if28: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.3/16 scope global eth1
      valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:3/64 scope link
      valid_lft forever preferred_lft forever
```

如图 10-2 所示，从以上增加的网络配置不难看出，这里使用了 LVS NAT 模式，在各容器的 network namespace 实现了负载均衡，访问 VIP 时报文 IP 的替换发生在发起访问的容器中。

我们可以类似地做个简单的实验，三个容器或者三台机器，假设被访问的两个容器 IP 分别为 10.250.1.2，10.250.1.3，给它们分配 10.250.1.4 的 VIP，使用 LVS 做 NAT 负载均衡。在发起访问的容器中配置下面的 iptables和 ipvs 规则之后，即可以实现通过轮询 VIP 10.250.1.4 访问 10.250.1.2 和 10.250.1.3 两个容器。

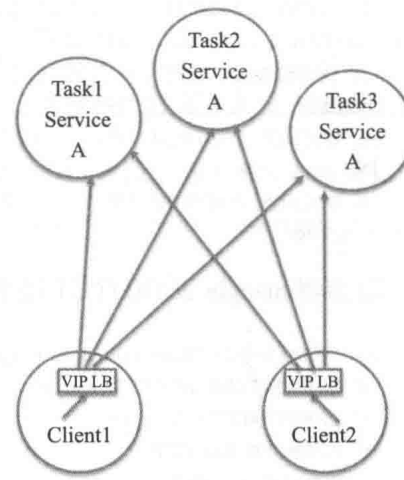


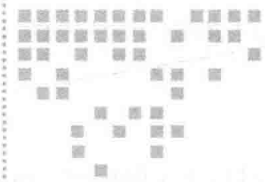
图 10-2 容器负载均衡原理

```
iptables -t mangle -A OUTPUT -d 10.250.1.4/32 -j MARK --set-mark 1
ipvsadm -A -f 1 -s rr
ipvsadm -a -f 1 -r 10.250.1.2:0 -m -w 1
ipvsadm -a -f 1 -r 10.250.1.3:0 -m -w 1
```

上面的规则中 -set-mark 1 使报文通过 OUTPUT 链时打上 1 的标记，ipvsadm -A -f 1 -s rr 创建了 FWMARK 的 virtual server，只要报文有 1 的标记，就会应用 ipvs 的规则，改变 dest IP。新加的 eth0 的 secondary IP 和 SNAT 规则是给容器通过 VIP 访问到自己时使用。

10.5　本章小节

本章主要介绍了 Docker Swarm 容器集群的核心设计和基本功能，然后对负载均衡等重要功能的原理进行了分析。虽然 Docker Swarm 项目相比于 Mesos/Kubernetes 项目还比较年轻和不成熟，但是项目当前的开发迭代速度非常快，项目的架构和设计也有其独特和创新之处，值得我们学习和思考。



Chapter 11 第 11 章

Docker 插件开发

用户可以使用第三方的插件扩展 Docker Engine 的功能。目前 Docker 1.12 正式版本支持 authorization，network，volume 插件，experimental 版本还支持 graphdriver 插件。

目前社区已经实现了很多开源的 Docker 插件，详细的列表读者可以参考 https://docs.docker.com/engine/extend/plugins/。如果这些插件不能满足读者的需求，我们可以实现自己的 Docker 插件，本章主要介绍如何开发一个 Docker 插件。

11.1 Docker 插件工作机制

图 11-1 表示 Docker 插件的工作流程。当用户使用 docker volume/network 命令使用第三方插件时，首先 Docker 客户端程序向 Docker 守护（docker daemon）进程发出 HTTP 请求，Docker 守护进程收到请求后如果发现操作的对象是第三方插件，便会从特定的目录文件中发现匹配的插件进程地址并向其发起 HTTP 请求。下面几个小节将对这个过程进行详细的介绍。

11.1.1 Docker 插件接口

Docker 插件是单独的一个进程，不是 docker daemon 进程的一个模块。插件进程与 docker daemon 进程运行于同一台机器或者不同的机器上，通过注册特定的文件到

docker daemon 的机器上，使得 docker daemon 进程发现插件进程。插件进程可以运行在容器中，或者容器外，目前推荐使用后者。

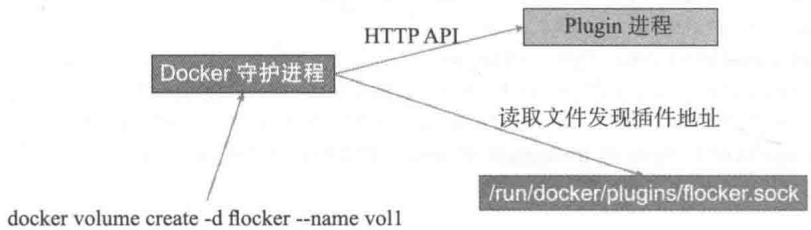


图 11-1　Docker 插件工作原理

11.1.2　插件发现机制

docker daemon 进程通过查询插件目录发现插件进程，目前支持三种格式的文件：

- .sock 文件，Unix domain socket。
- .spec 文件，普通文件，文件中指定了插件进程的 URL，例如，unix:///other.sock 或者 tcp://localhost:8080。
- .json 文件，包含了插件的 json 描述。

使用 .sock 文件的插件必须与 docker daemon 进程运行在同一台机器上，使用 .spec 文件或者 .json 文件的插件可以运行在另一台机器上。.sock 文件必须位于 /run/docker/plugins 目录下，.spec 文件和 .json 文件必须位于 /etc/docker/plugins 或者 /usr/lib/docker/plugins 目录下。

文件名称就是插件的名称，例如，使用 /run/docker/plugins/flocker.sock 文件的叫 flock 插件。也可以在 /run/docker/plugins 目录下创建子目录，把 /run/docker/plugins/flocker 目录挂载到 flock 插件的容器中，然后创建 /run/docker/plugins/flocker/flocker.sock 文件。

Docker 首先从 /run/docker/plugins 目录查找 .sock 文件，如果查找不到 .sock 文件，再从 /etc/docker/plugins 目录和 /usr/lib/docker/plugins 查找 .spec 和 .json 文件。如果查找到其中一个文件，就停止查找插件。

11.1.3　JSON 文件格式

JSON 文件格式如下所示。TLSConfig 字段是可选的，如果设置了这个字段，才会

进行 TLS 认证。

```
{
  "Name": "plugin-example",
  "Addr": "https://example.com/docker/plugin",
  "TLSConfig": {
    "InsecureSkipVerify": false,
    "CAFile": "/usr/shared/docker/certs/example-ca.pem",
    "CertFile": "/usr/shared/docker/certs/example-cert.pem",
    "KeyFile": "/usr/shared/docker/certs/example-key.pem",
  }
}
```

11.1.4 插件的生命周期

插件通常应该在 docker daemon 启动之前启动，在 docker daemon 停止后停止。如果使用 systemd 管理插件进程，可以使用 systemd 的 dependencies 功能管理插件和 docker 的启动和停止顺序。但是启动顺序并没有那么严格，因为 docker daemon 并不是启动的时候就去激活插件，而是使用按需加载的方式去激活插件，也就是说只有用户使用插件时（例如：docker run —volume-driver=foo，使用 foo volume 插件），docker daemon 才会激活插件进程。所以只需要保证插件进程在用户使用前启动就可以。

11.1.5 利用 systemd socket activation 功能管理插件

可以利用 systemd 的 socket activation 功能管理插件的启动顺序，Docker 提供的插件 sdk 已经支持 socket activation。使用这种方式需要编写两个文件，service 文件和 socket 文件。service 文件如下（例如 /lib/systemd/system/your-plugin.service）：

```
[Unit]
Description=Your plugin
Before=docker.service
After=network.target your-plugin.socket
Requires=your-plugin.socket docker.service

[Service]
ExecStart=/usr/lib/docker/your-plugin

[Install]
WantedBy=multi-user.target
```

socket 文件如下（例如 /lib/systemd/system/your-plugin.socket）：

```
[Unit]
Description=Your plugin

[Socket]
ListenStream=/run/docker/plugins/your-plugin.sock

[Install]
WantedBy=sockets.target
```

这种方式插件将在 docker daemon 真正去连接 socket 时才启动。

11.1.6　API 格式

插件 API 使用 HTTP JSON 格式。docker daemon 主动向插件发起 HTTP 请求，插件需要实现一个 HTTP server，去监听上文的 Unix socket。所有的 HTTP 请求都是 POST 请求。API 版本通过 Accept 请求头发送，目前这个头被设置为“application/vnd.docker.plugins.v1+json”。

插件通过 handeshake api 激活。

```
/Plugin.Activate
request: empty body

response:
{
  "Implements": ["VolumeDriver"]
}
```

HTTP response 是这个插件实现的 docker 插件类型的集合。激活插件后，docker 会向插件发起相关具体的请求。“implements”支持的值有 authz、NetworkDriver、VolumeDriver，即权限插件、网络插件和数据卷插件。

Docker 对插件接口的 HTTP 请求采取指数增长重试间隔的重试策略，最长重试间隔 30 秒。Docker 官方提供了一个插件的 SDK 工具 https://github.com/docker/go-plugins-helpers，这个 SDK 定义好了各插件的 API 接口和参数，读者可以使用它快速开发插件。

11.2　Docker volume 插件开发

由于 linux 内核没有对 /proc/meminfo、/proc/stat 等文件中的资源数据实现按容器分别统计，在 docker 容器中使用 free、top 等命令显示的资源使用情况依然是主机的整体

情况。LXC 技术实现了 lxcfs（https://github.com/lxc/lxcfs），利用 fuse 实现用户态的文件系统。lxcfs 从容器的 cgroup 状态文件中读取数据来反应容器的内存、CPU 等资源的实际使用情况，为容器提供仿真文件替换实际的 /proc/meminfo、/proc/stat。

下面我们借助同样的技术来实现容器中 /proc/meminfo 的隔离，并且将这个功能通过 docker 提供的 volume 插件机制来提供给容器使用。cgroupfs https://github.com/chenchun/cgroupfs 是作者用 go 语言实现的类似 lxcfs 的用户态文件系统。我们借助这个 go 语言工具包来实现这个 volume 插件。

11.2.1 cgroupfs 使用方法和工作原理

首先介绍下 cgroupfs 的使用方法，操作流程如下面的代码框所示。先创建一个容器，设置内存上限为 15M，保存容器 id。切换到第二个控制台窗口，创建 /tmp/cgroupfs 目录，并启动 cgroupfs 进程。再切换回第一个窗口，启动容器，在容器中查看总内存上限和已经使用的内存，可以看到内存上限为我们设置的 15M，已使用为容器实际使用的 2M 内存。

```
    container_id=`docker create -v /tmp/cgroupfs/meminfo:/proc/meminfo -m=15m
ubuntu sleep 2000`
    ## In the second console tab
    mkdir /tmp/cgroupfs
    ./cgroupfs /tmp/cgroupfs /docker/$container_id

    ## Go to the first tab
    docker start $container_id

    ## Take a look at /tmp/cgroupfs/meminfo now
    ## cgroupfs file system should be able to show the memory usage of the container
    cat /tmp/cgroupfs/meminfo
    MemTotal:       15360 kB
    MemFree:        13432 kB
    MemAvailable:   13432 kB
    Buffers:        0 kB
    Cached:         1804 kB
    SwapCached:     0 kB

    ## Enter docker container, you should see free is showing the real usage
    docker exec -it $container_id bash
    root@251d4d18bca6:/# free -m
                 total       used       free     shared    buffers     cached
    Mem:            15          2         12          0          0          1
```

```
-/+ buffers/cache:          0         14
Swap:             0         0          0
```

cgroupfs 的工作原理如图 11-2 所示。cgroupfs 利用内核的 fuse 功能实现了用户态的文件系统。启动 Cgroupfs 进程时，第一个参数 `/tmp/cgroupfs` 表示 cgroupfs 使用哪个目录作为其文件系统的目录，第二个参数 `/docker/$container_id` 表示 cgroupfs 从哪个 cgroup 中读取 cgroup 状态数据。Cgroupfs 进程在主机 /tmp/cgroupfs/ 目录下创建 meminfo/stat/cpuinfo 等文件。我们将 /tmp/cgroupfs/meminfo 挂载到容器中，读取 meminfo 文件内容时，会调用内核中的 vfs 接口。vfs 发现这些文件是 fuse 文件系统的文件时，向 fuse 模块发送读请求。之后内核的 fuse 模块向用户态的 cgroupfs 进程发送读请求，最后 cgroupfs 通过读取 cgroup 的相关文件，获取该容器的 cgroup 内存信息并返回。

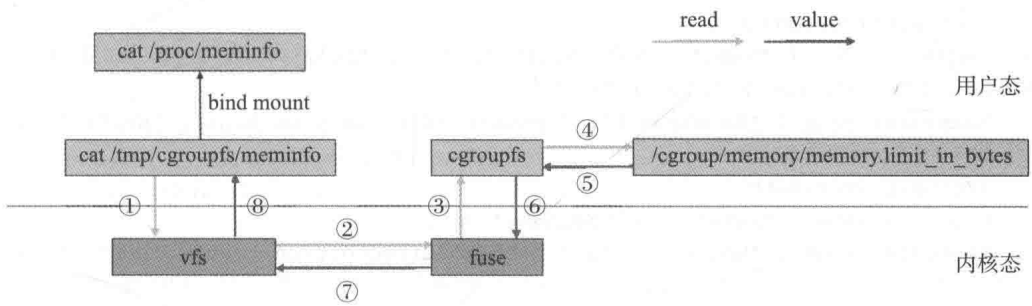


图 11-2　cgroupfs 原理

以上面的方式已经可以给容器提供仿真的 /proc/meminfo 等文件了，但是这种方式需要去管理 cgroupfs 进程的启动和停止，能不能用更加 Docker 的方式来实现这个功能呢？答案是肯定的，我们可以实现 cgroupfs 的 docker volume 插件，将 cgroupfs 进程的启动和停止替换为对 docker volume 的操作。

11.2.2　docker volume 接口

Docker 提供了 docker volume 子命令对 volume 进行操作。下面的命令表示使用 local volume 插件创建一个名称是 foo 的 volume，并传入了一个 type=brtfs 的参数。

```
docker volume create --driver local --opt type=btrfs --name foo
```

同样的，还提供了 docker volume rm 命令删除 volume，docker volume ls、docker

volume inspect 命令查看 volume 列表和 volume 状态。

介绍完 docker volume 的用户操作接口，我们再来看看 Docker volume 插件的接口。Volume 接口主要包含了下面 8 个 HTTP 接口。Capabilities 接口是在 docker1.12 版本引入的，scope 只有 global 或者 local 两个值，表示 volume 插件是否支持跨主机的 volume。Create/remove 接口在使用创建和删除 volume 时被调用，mount/unmount 接口在使用该 volume 的容器启动和停止时 volume 时被调用。

```
/VolumeDriver.Create
Request: json { "Name": "volume_name", "Opts": {} }
Response: json { "Err": "" }

/VolumeDriver.Remove
Request: json { "Name": "volume_name" }
Response: json { "Err": "" }

/VolumeDriver.Mount
Request: json { "Name": "volume_name", "ID": "b87d7442095999a92b65b3d9691e
697b61713829cc0ffd1bb72e4ccd51aa4d6c" }
Response: json { "Mountpoint": "/path/to/directory/on/host", "Err": "" }

/VolumeDriver.Path
Request: json { "Name": "volume_name" }
Response: json { "Mountpoint": "/path/to/directory/on/host", "Err": "" }

/VolumeDriver.Unmount
Request: json { "Name": "volume_name", "ID": "b87d7442095999a92b65b3d9691e
697b61713829cc0ffd1bb72e4ccd51aa4d6c" }
Response: json { "Err": "" }

/VolumeDriver.Get
Request: json { "Name": "volume_name" }
Response: json { "Volume": { "Name": "volume_name", "Mountpoint": "/path/
to/directory/on/host", "Status": {} }, "Err": "" }

/VolumeDriver.List
Request: json {}
Response: json { "Volumes": [ { "Name": "volume_name", "Mountpoint": "/
path/to/directory/on/host" } ], "Err": "" }

/VolumeDriver.Capabilities
Request: json {}
Response: json { "Capabilities": { "Scope": "global" } }
```

11.2.3　实现 cgroupfs-volume volume 插件

按照上文的介绍，我们使用 sdk 包和 cgroupfs 包来实现 cgroupfs-volume volume 插件。使用 sdk 只需要两步：

- 定义 volumedriver 实现下面代码中的 Volume.Driver 接口，通过这个 driver 创建 handler 对象。
- 选择使用 TCP 或者 unix socket 提供服务。

```
d := MyVolumeDriver{}
h := volume.NewHandler(d)
h.ServeTCP("test_volume", ":8080")
h.ServeUnix("root", "test_volume")
```

```
type Driver interface {
  Create(Request) Response
  List(Request) Response
  Get(Request) Response
  Remove(Request) Response
  Path(Request) Response
  Mount(MountRequest) Response
  Unmount(UnmountRequest) Response
  Capabilities(Request) Response
}
```

要将 cgroupfs 的功能包装成 volume driver 主要需要考虑它的两个参数如何传递的问题。第一个参数是 cgroupfs 创建的用户态文件系统存放在哪个目录，第二个参数是从哪个 cgroup 获取数据。第一个问题比较好解决，docker sdk 定义了 volume.DefaultDockerRootDirectory 参数，它的值是 /var/lib/docker-volumes，我们可以在这个目录下加两级目录 /cgroupfs/$volume_name 以区分不同的 volume 插件和不同的 volume。

第二个问题主要是 containerid 如何传递给 cgroupfs 插件进程。Docker 给每个容器分配的 cgroup 目录是类似 /docker/$containerid 目录或者其他形式，不过无论哪种目录格式，只有 $containerid 这个目录字符串在创建容器前是无法获知的，只有创建后才能得到。而如果容器要使用 volume，那么在创建容器前需要创建 volume。所以，容器要使用 volume，就需要先创建 volume，但是创建 volume 时又需要将 containerid 作为参数传给 cgroupfs 模块，那么我们如何在创建容器前得知其 containerid？从上一节我们知道 mount 接口并不是创建容器时就调用的，而是在启动容器时才会调用，所以可以利用这点，约定使用一个文件，创建 volume 时告诉 volume 插件使用这个文件读

取 containerid，创建容器后将 containerid 写入这个文件。当启动容器调用 volume 的 mount 接口时，cgroupfs volume 插件正好可以从这个文件中读取到 containerid，从而调用 cgroupfs 包创建 fuse 的仿真文件系统。docker create 和 docker run 命令恰好提供了 cidfile 这个参数，如果创建容器时增加 cidfile，docker 会在创建容器后将容器的 cid 写入参数指定的文件中。由于使用了 cidfile 参数，我们可以将 docker create 和 docker start 合成一步，那么按照设计，最后的操作过程会是这样：

```
    docker volume create -d cgroupfs_volume --name myvolume -o cidfile=/tmp/
containerid
    docker run -d --cidfile /tmp/containerid --volume-driver=cgroupfs_volume
-v myvolume:/proc/meminfo -m=15m chenchun/hello /hello
```

做好了设计，编码就很简单了。定义 volumeServer struct 实现 volume.Driver 接口。这个结构体只有一个 volumes 字段存储创建的 volume。在 create 方法中仅记录创建了这个 volume，在 mount 方法中调用 cgroupfs 包创建 fuse 文件系统。

```
type volumeServer struct {
  volumes map[string]*volume.Request
}

func (s *volumeServer) Create(req volume.Request) volume.Response {
  s.volumes[req.Name] = &req
  return volume.Response{}
}

func (s *volumeServer) Mount(req volume.MountRequest) volume.Response {
  resp := volume.Response{}
  if r, ok := s.volumes[req.Name]; ok {
    if err := mount(req.Name, memoryCgroupPath(r)); err != nil {
      resp.Err = err.Error()
    } else {
      resp.Mountpoint = mountPoint(req.Name)
    }
  } else {
    resp.Err = "volume does not exist"
  }
  return resp
}

func (s *volumeServer) List(req volume.Request) volume.Response {
  var volumes []*volume.Volume
  for v, _ := range s.volumes {
```

```
        volumes = append(volumes, &volume.Volume{Name: v, Mountpoint: mountPoint(v)})
    }
    return volume.Response{Volumes: volumes}
```

注：本节的代码可以在 https://github.com/chenchun/cgroupfs-volume/ 下载。

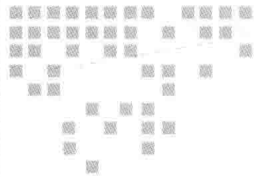
11.3　本章小节

本章主要介绍了 docker 插件的工作机制和实现方法，然后通过举例详细讲解了如何实现一个 docker volume 插件。

第三部分 Part 3

案 例 篇

- 第 12 章　Docker 离线系统应用案例
- 第 13 章　Etcd、Cadvisor 和 Kubernetes 实践
- 第 14 章　构建 Docker 高可用及自动发现架构实践
- 第 15 章　Docker Overlay Network 实践



Chapter 12 第 12 章

Docker 离线系统应用案例

“古巴比伦王颁布了汉谟拉比法典，刻在黑色的玄武岩，距今已经三千七百多年，你在橱窗前，凝视碑文的字眼，我却在旁静静欣赏你那张我深爱的脸……”，想必大家都听过这首周杰伦演唱的《爱在西元前》，由方文山填词，歌词生动美妙、不落俗套。作为周杰伦的御用作词人，方文山其实只有高中学历，但他才华横溢，创作了不少经典之作，有些作品甚至被收录进了学校的课本中。那么他是如何写出这么美妙的歌词的呢？直到看了他的访谈，笔者才了解到其实他的歌词大多来源于生活中的灵感，这首《爱在西元前》就是他在某日逛完博物馆后有感而发，他从现实的场景获得灵感再进行文字加工，从而文思泉涌，妙笔生花。其实运维工作也亦然，需要我们时时从工作中寻找灵感。笔者从事运维工作将近 8 年，多年的工作经验总结出一个道理：我们要善于从运维工作中发现问题，并根据问题及过往经验发明创造改进工具，接着再从发明创造中提炼技术，最后根据技术提炼原理。

近来以 Docker 为代表的容器很火，很多互联网公司都在使用。那么 Docker 是如何获得互联网公司的青睐的？我们又该如何来应用 Docker？本章将逐一介绍：

12.1 为什么使用 Docker

首先来看一下我们在运维中发现了哪些问题。以笔者就职的腾讯公司案例为例，腾

讯产品战线很长，笔者所在的团队负责运维腾讯的 QQ、Qzone 等核心产品，而这些产品也算是中国互联网骨灰级的业务了，整体架构都有着复杂的历史背景。不仅团队的老员工发现了架构逐渐产生的问题，团队新入职的同事也会指出与原任职公司比较架构存在的一些问题，等等。如何将很多好的技术融入我们的业务架构中，以解决产品的问题，是摆在每一个运维架构师面前的难题。架构的复杂性与历史沉重的包袱决定着牵一发而动全身，并且架构支持的业务为近半数的中国互联网网民所使用，还有着不少的优缺点，在对其进行改进优化之前，我们主要先分析了它的优缺点。

优点：目前还是单机或单集群对应一个业务模块的形式，彼此间没有交集与复用，如图 12-1 所示，其架构优势应对爆发式增长扩容方便，成本结算简单，架构形成的历史原因更多的是前者流量的爆发增长。

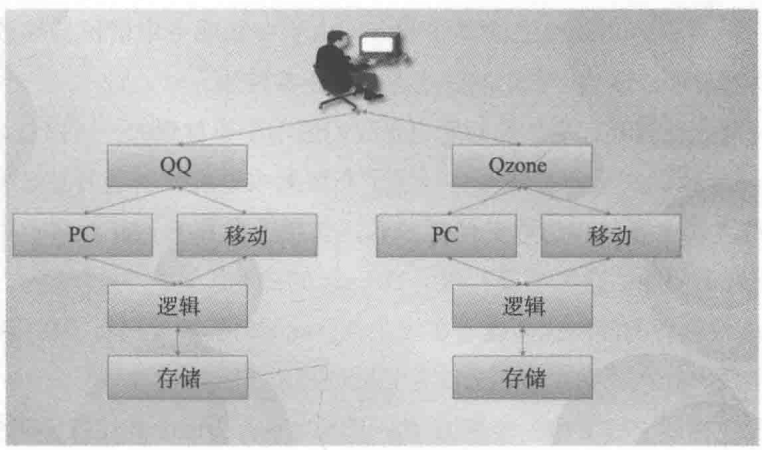


图 12-1　常见的业务模块架构

缺点：随着网民的饱和，相同业务模块下移动流量上涨，PC 流量直线下降，很多业务出现了低负载情况。海量服务器的运维场景中即使只有 1% 的低负载规模也是惊人的，在实际运营中低负载 + 长尾模块下的机器数量可能达到了 25% ~ 30% 甚至更高。当然，这里不排除业务对流量做了冗余与灾备，但是多个业务模块的灾备就会出现浪费资源的情况。除低负载与长尾业务外，为了应对业务的流量徒增，团队为产品留出了足够富裕的 Buffer 资源，而这些资源的实际利用率并不高。

注意　低负载：对服务器利用率衡量的标准，其中标准指标包括 CPU、内存、磁盘与

网络，并经过算法来确定某服务器的利用率是否处于合理区间内，如果不符合标示为低负载。

长尾业务：某业务在线使用，但用户不再增长并有逐渐下降的趋势界定为长尾业务。

分析了优缺点及运营中发现的问题后，从2014年年中我们就开始着手离线业务混合部署项目。项目的思路是通过高负载业务譬如音频转码、视频转码、图片人脸计算和图片特征提取业务（注：后统称“离线业务”）与低负载、长尾业务和部门Buffer业务进行混合部署，来提升机器利用率并同时为部门节约机器成本。不过经过半年多的运营，我们也发现了很多问题：

问题一：在低负载与长尾业务上部署离线业务，如果离线业务抖动出现大的CPU毛刺（特别在晚高峰时段）就会影响用户的使用体验，从而导致部分投诉产生的情况。

问题二：部署在Buffer池上的离线业务，由于在线业务申请机器导致机器上的离线环境没有及时清理，继续运营影响在线业务的正常使用。

这两个问题是比较有代表性的问题也是我们最不希望看到的，所以总结经验教训后在2015年上半年我们又重新打磨项目开发了新版离线系统，新项目与老项目相比通过Docker技术将离线业务与在线业务的低负载、长尾业务和Buffer池进行了部署，从而顺利解决了上述的问题。这里主要利用了Docker的三个优势：

- Cgroup对资源的隔离让离线与在线业务彼此间对资源使用有了保障。
- 命名空间，让相同框架多业务跑在相同物理机上成为可能。
- 容器快速回收上线下线，使用Buffer资源时不再为回收离线资源而头痛。

12.2 离线系统业务架构

上文中笔者提到QQ、Qzone的整体架构有着很复杂的历史背景，而仅仅通过Docker或一两种工具来解决现实存在的所有问题是不现实的，所以我们在保持现有架构与内部用户习惯的基础上另辟蹊径逐渐解决产品运营上的一些问题，具体部署如图12-2所示。

从图12-2中可以看到在部门Cmdb基础之上，我们建立了数据分析系统，通过它可以分析出哪些是低负载业务，哪些是时段低负载业务。在此基础之上我们使用Clip名字服务系统重新建立了业务之间的IP关系，并根据重新划定关系的IP进行了业务流

量与资源快速的调度。其中 Etcd 为服务发现工具，根据 Clip 名字服务系统传入的信息通过 Confd 重新生成 HAproxy 的配置文件并热重启它，离线业务流量通过 HAproxy 的虚拟 IP 屏蔽底层资源信息，建立离线与资源的生态供应链，同时上层监控与调度均为 Clip 名字服务系统。这里读者可能会有疑问，虽然 HAproxy 的虚拟 IP 屏蔽底层资源信息（其为动态的），但如果程序有问题，能快速到机器上去调试自己的程序吗？这里我们使用的是 Clip 名字服务。

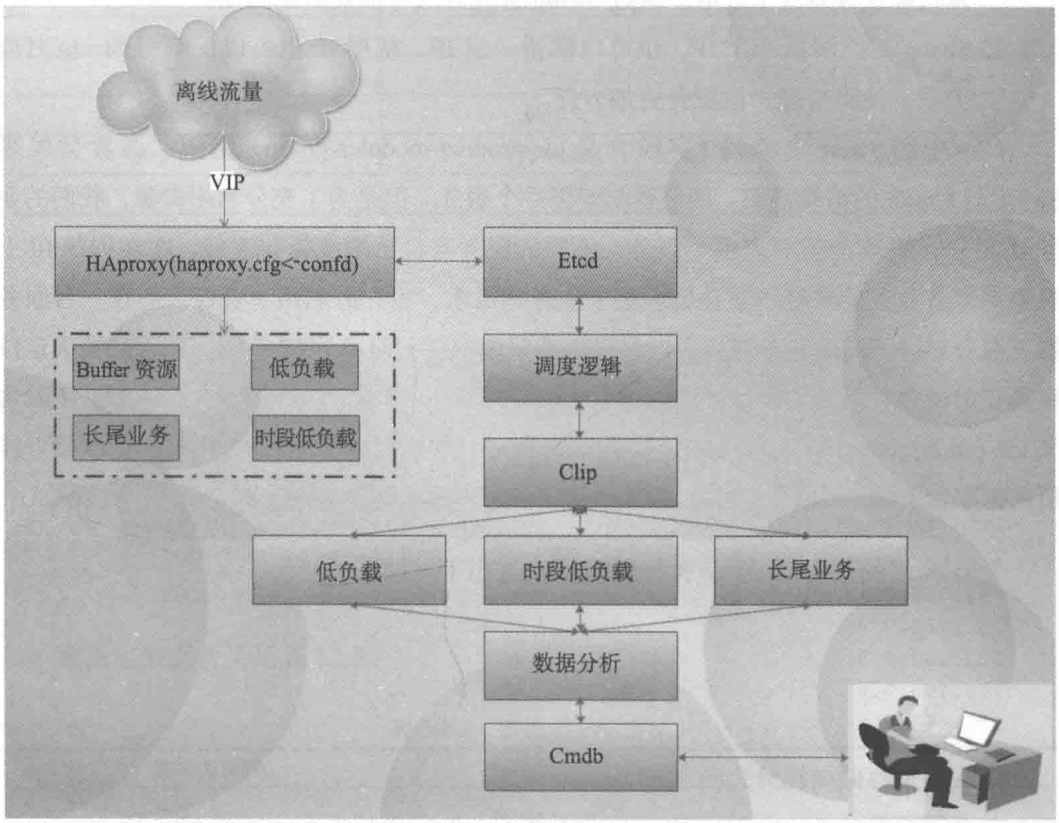


图 12-2　离线系统业务架构具体部署

12.3　Clip 名字服务

Clip（http://blog.puppeter.com/read.php?7）是一个名字服务 C/S 架构，它将传统的 IP 管理维度替换为名字服务，即有意义可记忆的 String。Clip 将 IP 对应的 String 关系保存在 Server 端。Client 端可以下载 SDK，通过 SDK 遍历 Server 端的 String 对应 IP

的关系，并在本地对获取的 IP 模块关系进行重新的组织与编排。传统服务器 IP 方式与 String 方式相比，String 方式有三点优势：

- 传统 IP 管理方式，IP 由四组无意义的数字组成，比较难记忆。String 更加方便记忆。
- 管理海量服务时，IP 相似经常会导致运营故障，譬如 A 模块（10.131.24.37）和 B 模块（10.117.24.37），后两组数字一致，系统惯性地认为 B 模块就是 A 模块，发送配置导致线上故障。通过 String 管理方式可以规避此问题。
- String 可以解析一个 IP，也可以解析一组 IP，根据 IP 也可以反解析 String 对应关系，使得管理一组服务更加方便。

Clip 中的 String 由四段（字段含义 idc-product-modules-group）组成，读者会发现 String 与 Cmdb 的结构很像：四级模块定位一个服务。但是为了充分利用资源，我们的业务要求一个 IP 可能要对应多个服务，不同的服务有自己的波峰与波谷，在相同的 IP 上只要保证各业务的波峰不重叠就满足了业务的需求，也充分利用了资源。而在一台服务器上混合部署不同的业务模块，四级模块就只能定位到服务的 IP 级别，而无法精确定位到真正的服务，所以 Clip 名字服务在 Cmdb 的基础上增加了 port 端口，即五段（字段含义 idc-product-modules-group-port）定位一个服务。我们可以将动态的 IP 通过 Clip 接口注册到指定含义的 String 上，通过 Clip 自带工具来解析实时的 String 对应 IP 关系，譬如：

```
# clip cstring -q idc-product-modules-group-port
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4
192.168.0.5
```

Clip 数据存储结构分为两层：

关系层保存了 String 对应 IP 或内部系统 Cmdb 的模块关系，如图 12-3 所示。

Field	Type	Null	Key	Default	Extra
idc	varchar(20)	NO	MUL	NULL	
product	varchar(50)	NO		NULL	
modules	varchar(50)	NO		NULL	
group	varchar(50)	NO		NULL	
ext	varchar(20)	YES		0	
s_k	varchar(20)	NO		NULL	
s_v	varchar(200)	NO		NULL	
operator	varchar(20)	NO		NULL	
flag	int(11)	YES		1	
timestamp	timestamp	NO		CURRENT_TIMESTAMP	on update CURRENT_TIMESTAMP

图 12-3 Clip 名字服务关系表

关系层数据含义见表 12-1。

表 12-1　关系层数据含义

数据库字段	含　义
idc	机房
product	产品
modules	模块
group	组
ext	扩展字段（端口）
s_k	String 与 IP 或内部系统模块对应关系键
s_v	String 与 IP 或内部系统模块对应关系值
operator	操作人
flag	字段状态（1 在线、2 离线、3 故障）
timestamp	创建变更时间戳

数据层保存了 String 与 IP 的具体关系，如图 12-4 所示。

Field	Type	Null	Key	Default	Extra
idc	varchar(20)	NO	MUL	NULL	
product	varchar(20)	NO		NULL	
modules	varchar(20)	NO		NULL	
group	varchar(20)	NO		NULL	
ext	varchar(20)	YES		0	
ipaddress	varchar(15)	YES		NULL	
flag	int(11)	YES		1	
timestamp	timestamp	NO		CURRENT_TIMESTAMP	on update CURRENT_TIMESTAMP

图 12-4　Clip 名字服务数据表

数据层数据含义见表 12-2。

表 12-2　数据层数据含义

数据库字段	含　义
idc	机房
product	产品
modules	模块
group	组
ext	扩展字段（端口）
ipaddress	String 对应 IP 关系
operator	操作人
flag	字段状态（1 在线、2 离线、3 故障）
timestamp	创建变更时间戳

Clip 名字服务存储在 String 对应 IP 关系基础上，在 SDK 上还提供了远程端口扫描、远程 ssh、远程文件拷贝和查找 String 关系结构的工具子命令等。

12.4 Clip 名字服务与 Docker 应用

如笔者上文所提，离线系统的建设思路就是将离线业务通过 Docker 快速地部署在资源空闲的机器上，而空闲的机器是通过数据分析系统长期分析沉淀的结果，Clip 名字服务就是这两种资源建立联系的桥梁，但只有 Clip 绑定关系还是不够的，还需要建立绑定关系 String 对应 Docker 环境的关系，所以在 Clip 名字服务的基础上我们又扩充了 Docker 的资源关系表，如图 12-5 所示。

Field	Type	Null	Key	Default	Extra
id	int(10) unsigned	NO	PRI	NULL	auto_increment
cstring	varchar(50)	NO	MUL	NULL	
idc	varchar(20)	NO		NULL	
product	varchar(20)	NO		NULL	
modules	varchar(20)	NO		NULL	
group	varchar(20)	NO		NULL	
ext	varchar(20)	NO		NULL	
admin_port	int(11)	YES		18000	
app_port	varchar(50)	YES		18000	
mcount	int(11)	NO		NULL	
docker_id	varchar(50)	NO		NULL	
docker_version	varchar(50)	NO		NULL	
mload	int(11)	NO		0	
status	int(5)	YES		0	
timestamp	timestamp	NO		CURRENT_TIMESTAMP	on update CURRENT_TIMESTAMP

图 12-5 Docker 资源关系表

其中 Docker 资源使用的关系表的字段含义见表 12-3。

表 12-3 Docker 资源使用关系表

id	自增 id
cstring	名字服务 idc-product-modules-group-port
idc	机房
product	产品
modules	模块
group	组
ext	扩充信息（端口）
admin_port	ssh 管理端口
app_port	业务端口，多端口用“;”分割。用于业务多端口监控

（续）

id	自增 id
mcount	某业务需要最大的核心数
Docker_id	Docker 镜像环境 ID
Docker_version	Docker 的 tag
mload	自动缩容标识，系统计算 String 对应的 IP 负载是否在合理区间范围内，如果不在，则自动缩容资源
status	当前 String 状态是在线、离线还是停止

我们来看一个需求案例：某视频转码业务在上海需要使用资源约 1000 核心，对应的关系表见表 12-4。

表 12-4　Docker 资源使用关系表

id	自增 id
cstring	名字服务 String（sh-buffer-qq-video-2877）
idc	上海机房（sh）
product	使用空闲资源在部门 Buffer 中
modules	qq（某业务）
group	Video 组
ext	业务端口
admin_port	22（ssh 管理端口）
app_port	2877;80（业务暴露的端口）
mcount	1000
Docker_id	dockerimages.docker.com:5000/images/imagesName
Docker_version	latest
mload	1 ~ 7（以天为单位，累计 +1，当业务连续 7 低负载时，将 mcount 核量降低 20%）
status	1 在线

由于 Buffer 资源不时在变动，我们需监控 String（sh-buffer-qq-video-2877）整体核数是否低于 mcount 值，如果低于，则出发自动扩容策略。同时也需针对 Docker 资源使用关系中的 app_port 字段来监控整体 String（sh-buffer-qq-video-2877）业务是否处于健康状态。在这里我们沉淀了 String（sh-buffer-qq-video-2877）的一些基础数据如负载、内存、网络和磁盘 IO 等为资源的调度奠定了基石，如图 12-6 所示。

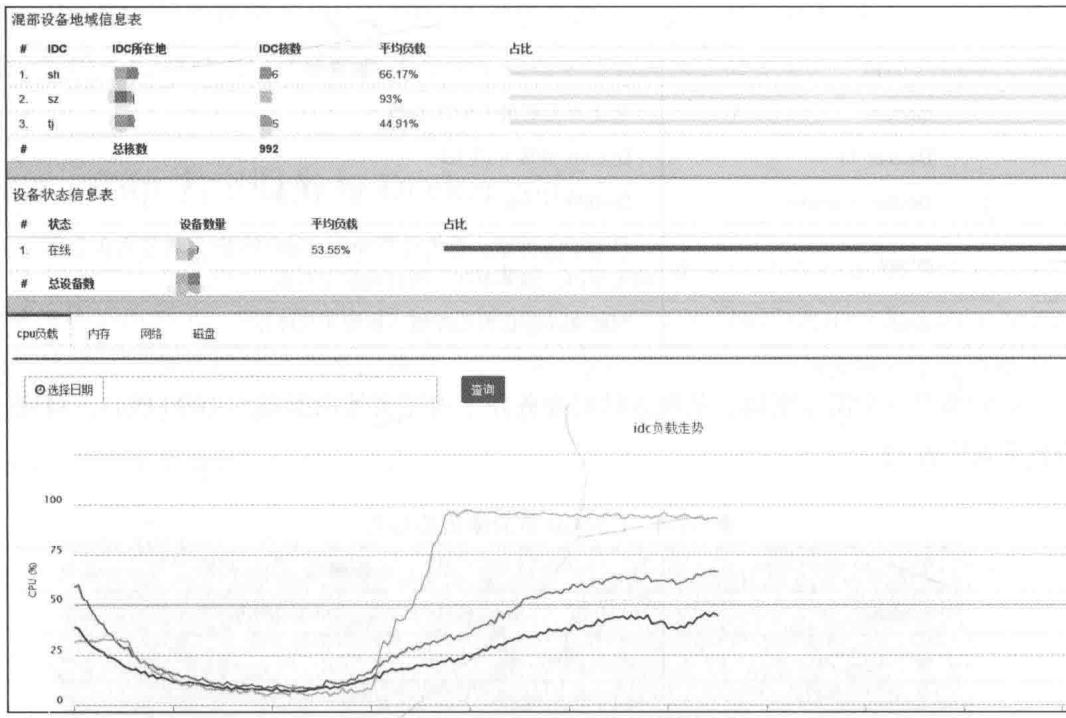
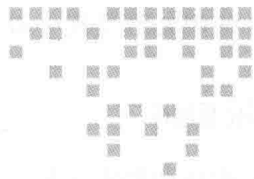


图 12-6 离线运营系统负载

12.5 本章小结

本章借助于 Clip 名字服务工具，介绍了一个 Docker 的离线系统应用案例，它可以非常方便地管理海量服务器，并且可以通过服务混合部署充分利用机器的各项资源，有效地降低机房的运营成本。



第 13 章 Chapter 13

Etcd、Cadvisor 和 Kubernetes 实践

本章节将介绍 Docker 的容器集群管理平台 Kubernetes、服务发现的键值存储系统 Etcd，以及容器监控平台 Cadvisor，详细介绍三个平台的功能特点、部署及使用方法。其中 Etcd 与 Cadvisor 也是 Kubernetes 平台的核心组件，可以了解到组件之间是如何协作的。

13.1 Etcd 实践

Etcd（https://github.com/coreos/etcd）是一个高可用的键值存储系统，主要用于共享配置和服务发现。Etcd 是由 CoreOS 开发并维护的，灵感来自 ZooKeeper 和 Doozer，它使用 Go 语言编写，并通过 Raft 一致性算法处理日志复制以保证强一致性。Raft 是一个来自 Stanford 的新的一致性算法，适用于分布式系统的日志复制，Raft 通过选举的方式来实现一致性。在 Raft 中，任何一个节点都可能成为 Leader。Google 的容器集群管理系统 Kubernetes、开源 PaaS 平台 Cloud Foundry 和 CoreOS 的 Fleet 都广泛使用了 Etcd，主要用于分享配置和服务发现。Etcd 具备如下特点：

- 简单：支持 curl 方式的用户 API (HTTP+JSON)。
- 安全：可选 SSL 客户端证书认证。
- 快速：单实例可达每秒 1000 次写操作。

❑ 可靠：使用 Raft 实现分布式。

13.1.1 安装 Etcd

Etcd 官方提供两种安装模式，一种为源码编译模式，要求具备 golang 语言环境，另一种为二进制程序包下载，解压后即可使用，简单快捷，笔者比较推荐此方式，详细安装方法如下：

```
# mkdir -p /home/install && cd /home/install
# wget https://github.com/coreos/etcd/releases/download/v0.4.6/etcd-v0.4.6-linux-amd64.tar.gz
# tar -zxvf etcd-v0.4.6-linux-amd64.tar.gz
# cd etcd-v0.4.6-linux-amd64
# cp etcd* /bin/
# /bin/etcd -version
etcd version 0.4.6  # 显示此信息说明部署成功
```

启动服务 Etcd 服务，如有提供第三方管理需求，另需在启动参数中添加“-cors='*'”参数，比较好的管理工具有 etcd-browser（http://henszey.github.io/etcd-browser/）。

```
# mkdir /data/etcd  #etcd 数据目录
# /bin/etcd -name etcdserver -peer-addr 192.168.1.10:7001 -addr 192.168.1.10:4001 -data-dir /data/etcd -peer-bind-addr 0.0.0.0:7001 -bind-addr 0.0.0.0:4001 &
```

由于 Etcd 具备多机容灾支持，参数“-peer-addr”指定与其他节点通信的地址；参数“-addr”指定服务监听地址；参数“-data-dir”为指定数据存储目录；IP 地址 192.168.1.10 为安装 Etcd 的主机。以下为配置 Etcd 服务防火墙，其中 4001 为服务端口，7001 为集群数据交互端口，策略如下：

```
# iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 4001 -j ACCEPT
# iptables -I INPUT -s 192.168.1.0/24 -p tcp --dport 7001 -j ACCEPT
```

13.1.2 使用方法

Etcd 提供了两种操作方法，一种是基于 HTTP 的 RESTful API，是一个使用 HTTP 并遵循 REST 原则的 Web 服务，通过不同 URI 来封装业务逻辑，由于基于 HTTP 协议，因此我们可以使用 curl 命令来操作，比较适合程序的调用。另一种是通过命令 etcdctl 操作，适合运维及系统管理人员使用。下面将详细介绍两种方式操作 Etcd 的常用方法。

（1）Set key（设置键值）

给“/message”key 设置“Hello world”值，操作命令如下：

```
# curl http://192.168.1.10:4001/v2/keys/message -XPUT -d value="Hello world"
{
  "action": "set",
  "node": {
    "key": "/message",
    "value": "Hello world",
    "modifiedIndex": 231006,
    "createdIndex": 231006
  }
}
```

命令行操作如下：

```
# etcdctl set /message "Hello world"
Hello world
```

（2）Get key（获取键值）

获取“/message”key 值，操作命令如下：

```
# curl http://192.168.1.10:4001/v2/keys/message
{
  "action": "get",
  "node": {
    "key": "/message",
    "value": "Hello world",
    "modifiedIndex": 231006,
    "createdIndex": 231006
  }
}
```

命令行操作如下：

```
# etcdctl get /message
Hello world
```

（3）Changing value（更新键值）

更新“/message”key 的值为“Hello etcd”，操作命令如下：

```
# curl http://192.168.1.10:4001/v2/keys/message -XPUT -d value="Hello etcd"
{
  "action": "set",
```

```
  "node": {
    "key": "/message",
    "value": "Hello etcd",
    "modifiedIndex": 231225,
    "createdIndex": 231225
  },
  "prevNode": {
    "key": "/message",
    "value": "Hello world",
    "modifiedIndex": 231006,
    "createdIndex": 231006
  }
}
```

命令行操作如下：

```
# etcdctl update /message "Hello etcd"
Hello etcd
```

（4）Deleting a key（删除键）

删除“/message”key，操作命令如下：

```
# curl http://192.168.1.10:4001/v2/keys/message -XDELETE
{
  "action": "delete",
  "node": {
    "key": "/message",
    "modifiedIndex": 231395,
    "createdIndex": 231225
  },
  "prevNode": {
    "key": "/message",
    "value": "Hello etcd",
    "modifiedIndex": 231225,
    "createdIndex": 231225
  }
}
```

命令行操作如下：

```
# etcdctl rm /message
```

（5）Creating Directories（创建目录）

创建一个“/dir”目录，操作命令如下：

```
# curl http://192.168.1.10:4001/v2/keys/dir -XPUT -d dir=true
{
  "action": "set",
  "node": {
    "key": "/dir",
    "dir": true,
    "modifiedIndex": 232067,
    "createdIndex": 232067
  }
}
```

命令行操作如下：

```
# etcdctl mkdir /dir
```

（6）Deleting a Directory（删除目录）

删除“/dir”目录，操作命令如下：

```
# curl 'http://192.168.1.10:4001/v2/keys/dir?dir=true' -XDELETE
{
  "action": "delete",
  "node": {
    "key": "/dir",
    "dir": true,
    "modifiedIndex": 232213,
    "createdIndex": 232067
  },
  "prevNode": {
    "key": "/dir",
    "dir": true,
    "modifiedIndex": 232067,
    "createdIndex": 232067
  }
}
```

命令行操作如下：

```
# etcdctl rmdir /dir
```

（7）watch value change（捕捉 key 的 value 更新事件）

用于捕捉 key 的 value 的更新事件，从而触发某个动作。实现捕捉“/message”key 的变化，命令执行后会处于等待状态，直到 key 的 value 发生改变才退到系统提示符，操作命令如下：

```
# curl 'http://192.168.1.10:4001/v2/keys/message?wait=true
```

命令行操作如下：

```
# etcdctl watch /message
```

更多 Ectd 操作参考官方 API 文档：https://github.com/coreos/etcd/blob/master/Documentation/api.md。

13.2 Cadvisor 实践

Cadvisor（https://github.com/google/cadvisor）是谷歌公司用来分析运行中的 Docker 容器的资源占用及性能特性的工具。Cadvisor 是一个运行中的守护进程，用来收集、聚合、处理和导出运行容器相关的信息，每个容器保持独立的参数、历史资源使用情况和完整的资源使用数据。当前支持 lmctfy 容器和 Docker 容器。Cadvisor 提供前端 UI 及 API 接口，以供第三方程序进行调用。

13.2.1 安装 Cadvisor

部署 Cadvisor 容器监控平台，官网提供了两种方式，一种为源码编译方式，需要本地具有 goland 环境；另一种为镜像方式，目前托管在 registry.hub.docker.com 下，用户在 Docker 主宿机环境运行“ docker pull google/cadvisor”即可，最后再启动该镜像的一个实例，运行：

```
# docker run \
  --volume=/var/run:/var/run:rw \
  --volume=/sys/fs/cgroup/:/sys/fs/cgroup:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  google/cadvisor
```

访问 http:// 主宿机 IP:8080，出现如图 13-1 所示界面，说明安装成功。

Cadvisor 获取该主宿机所有容器的 CPU、内存、网卡等信息，且以秒为单位进行刷新，及时性非常高，缺点是不能集中式管理，要求每台主宿机都要部署 Cadvisor 环境，有没有办法实现集中式的数据采集？答案是肯定的，Cadvisor 也提供了 Remote REST API，可以轻松与采集程序进行对接，下一章节将详细进行说明。

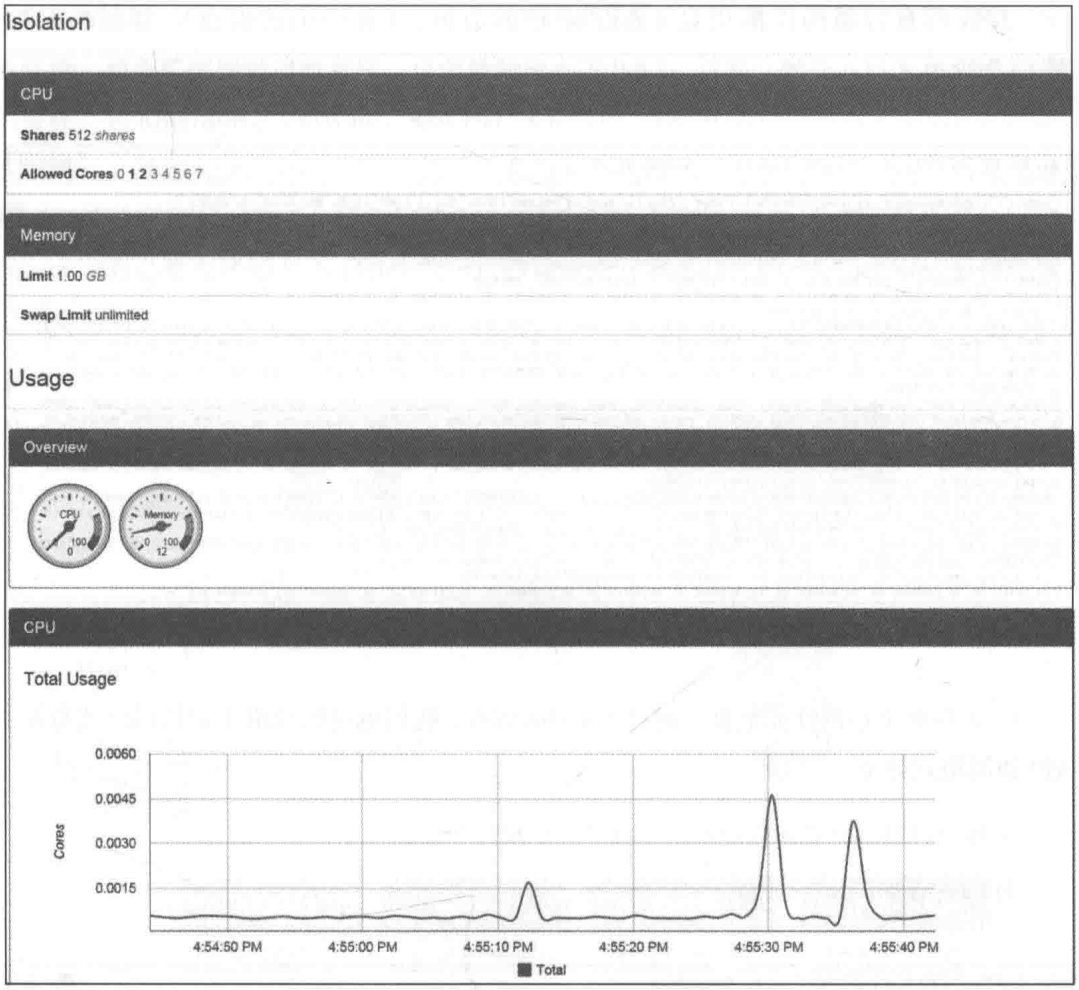


图 13-1　Cadvisor 前端界面

13.2.2　Cadvisor API

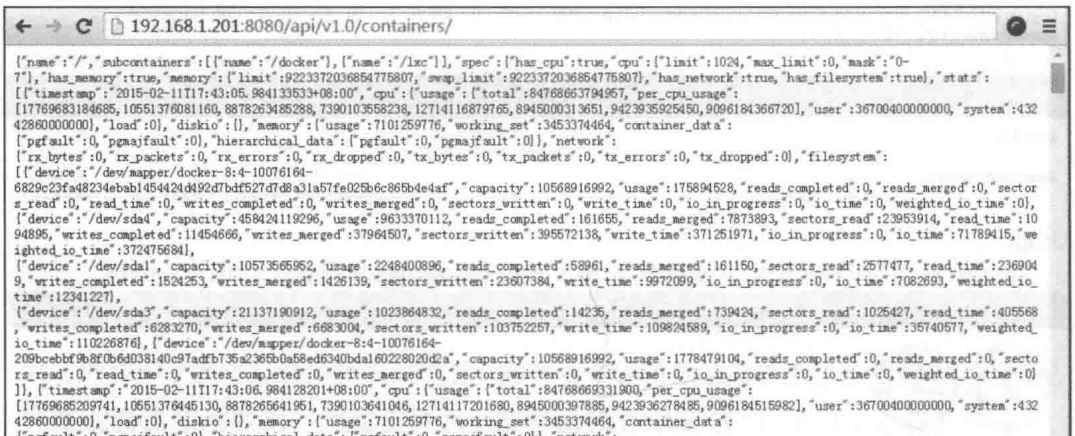
Cadvisor 提供了远程 REST API 的接口，通过接口我们可以获取前端 UI 看到的所有原始数据，调用地址格式如下（限 V1.0 版本）：

```
http://<hostname>:<port>/api/<version>/<request>
```

（1）获取容器性能数据。API 访问格式如下：

```
http://192.168.1.201:8080/api/v1.0/containers/  # 获取所有容器信息
http://192.168.1.201:8080/api/v1.0/containers/docker/666a95d88aa11418f37d4
d64319b2ada20bebd2e074e9899505492b307dcba4c          # 获取指定容器 ID 的信息
```

获取所有容器的性能信息（最近 60 秒的数据，1 秒一个数据点），详细内容如图 13-2 所示。



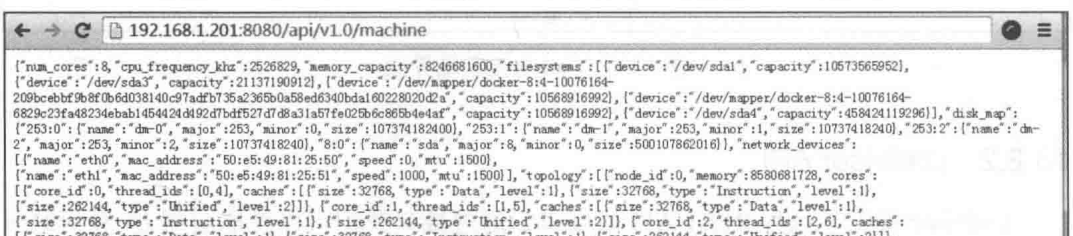
{"name":"/","subcontainers":[{"name":"/docker"},{"name":"/lxc"}],"spec":{"has_cpu":true,"cpu":{"limit":1024,"max_limit":0,"mask":"0-7"},"has_memory":true,"memory":{"limit":9223372036854775807,"swap_limit":9223372036854775807},"has_network":true,"has_filesystem":true},"stats":[{"timestamp":"2015-02-11T17:43:05.984133533+08:00","cpu":{"usage":{"total":84768663794957,"per_cpu_usage":[17769683184685,10551376081160,8878263485288,7390103558238,12714116879765,8945000313651,9423935925450,9096184366720],"user":36700400000000,"system":43242860000000},"load":0},"diskio":{},"memory":{"usage":7101259776,"working_set":3453374464,"container_data":{"pgfault":0,"pgmajfault":0},"hierarchical_data":{"pgfault":0,"pgmajfault":0}},"network":{"rx_bytes":0,"rx_packets":0,"rx_errors":0,"rx_dropped":0,"tx_bytes":0,"tx_packets":0,"tx_errors":0,"tx_dropped":0},"filesystem":[{"device":"/dev/mapper/docker-8:4-10076164-6829c23fa48234ebab1454424d92d7bdf527d7d8a31a57fe025b6c865b4e4af","capacity":10568916992,"usage":175894528,"reads_completed":0,"reads_merged":0,"sectors_read":0,"read_time":0,"writes_completed":0,"writes_merged":0,"sectors_written":0,"write_time":0,"io_in_progress":0,"io_time":0,"weighted_io_time":0},{"device":"/dev/sda4","capacity":458424119296,"usage":9633370112,"reads_completed":161655,"reads_merged":7873893,"sectors_read":23953914,"read_time":1094895,"writes_completed":11454666,"writes_merged":37964507,"sectors_written":395572138,"write_time":371251971,"io_in_progress":0,"io_time":71789415,"weighted_io_time":372475684},{"device":"/dev/sda1","capacity":10573565952,"usage":2248400896,"reads_completed":58961,"reads_merged":161150,"sectors_read":2577477,"read_time":2369049,"writes_completed":1524253,"writes_merged":1426139,"sectors_written":23607384,"write_time":9972099,"io_in_progress":0,"io_time":7082693,"weighted_io_time":12341227},{"device":"/dev/sda3","capacity":21137190912,"usage":1023864832,"reads_completed":14235,"reads_merged":739424,"sectors_read":1025427,"read_time":405568,"writes_completed":6283270,"writes_merged":6683004,"sectors_written":103752257,"write_time":109824589,"io_in_progress":0,"io_time":35740577,"weighted_io_time":110226876},{"device":"/dev/mapper/docker-8:4-10076164-209bcebbf9b8f0b6d038140c97adfb735a2365b0a58ed6340bda160228020d2a","capacity":10568916992,"usage":1778479104,"reads_completed":0,"reads_merged":0,"sectors_read":0,"read_time":0,"writes_completed":0,"writes_merged":0,"sectors_written":0,"write_time":0,"io_in_progress":0,"io_time":0,"weighted_io_time":0}]},{"timestamp":"2015-02-11T17:43:06.984128201+08:00","cpu":{"usage":{"total":84768669331900,"per_cpu_usage":[17769685209741,10551376445130,8878265641951,7390103641046,12714117201680,8945000397885,9423936278485,9096184515982],"user":36700400000000,"system":43242860000000},"load":0},"diskio":{},"memory":{"usage":7101259776,"working_set":3453374464,"container_data":{"pgfault":0,"pgmajfault":0},"hierarchical_data":{"pgfault":0,"pgmajfault":0}},"network":{"rx_bytes":0,"rx_packets":0,"rx_errors":0,"rx_dropped":0,"tx_bytes":0,"tx_packets":0,"tx_errors":0,"tx_dropped":0},"filesystem":

图 13-2 返回所有容器性能信息（JSON 格式）

（2）获取主宿机性能数据。通过 CadvisorAPI，我们也可以获取主宿机的性能数据，API 访问格式如下：

```
http://192.168.1.201:8080/api/v1.0/machine
```

详细内容如图 13-3 所示。



{"num_cores":8,"cpu_frequency_khz":2526829,"memory_capacity":8246681600,"filesystems":[{"device":"/dev/sda1","capacity":10573565952},{"device":"/dev/sda3","capacity":21137190912},{"device":"/dev/mapper/docker-8:4-10076164-209bcebbf9b8f0b6d038140c97adfb735a2365b0a58ed6340bda160228020d2a","capacity":10568916992},{"device":"/dev/mapper/docker-8:4-10076164-6829c23fa48234ebab1454424d92d7bdf527d7d8a31a57fe025b6c865b4e4af","capacity":10568916992},{"device":"/dev/sda4","capacity":458424119296}],"disk_map":{"253:0":{"name":"dm-0","major":253,"minor":0,"size":10737418240},"253:1":{"name":"dm-1","major":253,"minor":1,"size":10737418240},"253:2":{"name":"dm-2","major":253,"minor":2,"size":10737418240},"8:0":{"name":"sda","major":8,"minor":0,"size":500107862016}},"network_devices":[{"name":"eth0","mac_address":"50:e5:49:81:25:50","speed":0,"mtu":1500},{"name":"eth1","mac_address":"50:e5:49:81:25:51","speed":1000,"mtu":1500}],"topology":[{"node_id":0,"memory":8580681728,"cores":[{"core_id":0,"thread_ids":[0,4],"caches":[{"size":32768,"type":"Data","level":1},{"size":32768,"type":"Instruction","level":1},{"size":262144,"type":"Unified","level":2}]},{"core_id":1,"thread_ids":[1,5],"caches":[{"size":32768,"type":"Data","level":1},{"size":32768,"type":"Instruction","level":1},{"size":262144,"type":"Unified","level":2}]},{"core_id":2,"thread_ids":[2,6],"caches":[{"size":32768,"type":"Data","level":1},{"size":32768,"type":"Instruction","level":1},{"size":262144,"type":"Unified","level":2}]},{"core_id":3,"thread_ids":[3,7],"caches":[{"size":32768,"type":"Data","level":1},{"size":32768,"type":"Instruction","level":1},{"size":262144,"type":"Unified","level":2}]}],"caches":null}]}

图 13-3 返回所有主宿机性能信息（JSON 格式）

13.3 Kubernetes 实践

Kubernetes(https://github.com/GoogleCloudPlatform/kubernetes) 是 Google 开源的容器

集群管理系统，基于 Docker 构建一个容器的调度服务，提供资源调度、均衡容灾、服务注册、动态扩缩容等功能套件，目前最新版本为 0.6.2。目前受到各大巨头及初创公司的青睐，如 Microsoft、VMWare、Red Hat、CoreOS、Mesos 等纷纷加入，给 Kubernetes 贡献代码。随着 Kubernetes 社区及各大厂商的不断改进、发展，Kuberentes 将成为容器管理领域的领导者。本文介绍如何基于 Centos7.0 构建 Kubernetes 平台，在正式介绍之前，大家有必要先理解 Kubernetes 几个核心概念及其承担的功能。Kubernetes 架构图如图 13-4 所示。

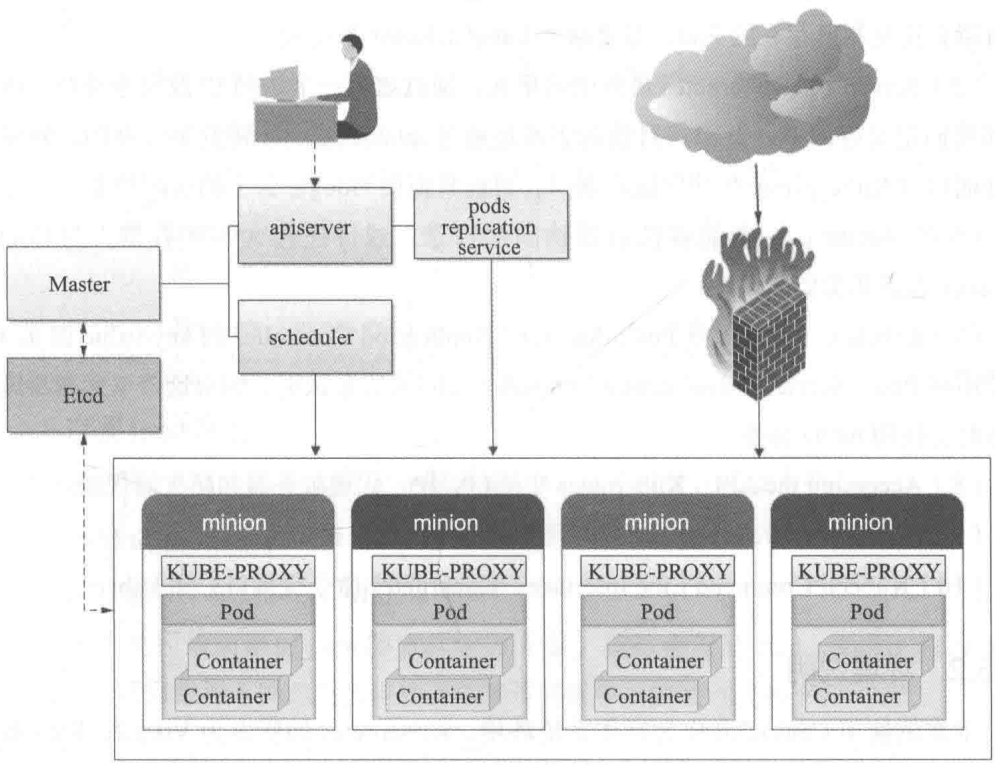


图 13-4　Kubernetes 架构图

13.3.1　基本概念

在开启 Kubernetes 之旅前，大家先了解 Kubernetes 几个基本概念，对理解其工作原理将起到至关重要的作用。

（1）Nodes：代表 Kubernetes 平台中的工作节点，如一台主宿机。

（2）Pods：在 Kubernetes 系统中，调度的最小颗粒不是单纯的容器，而是抽象成一个 Pod，Pod 是一个可以被创建、销毁、调度、管理的最小的部署单元，如一个或一组容器。

（3）The Life of a Pod：包括 Pod 的状态、事件及重启生命周期策略、复制控制器等。

（4）Replication Controllers：Kubernetes 系统中最有用的功能，实现复制多个 Pod 副本，往往一个应用需要多个 Pod 来支撑，并且可以保证其复制的副本数。即使副本所调度分配的主宿机出现异常，通过 Replication Controller 可以保证在其他主宿机启用同等数量的 Pod。Replication Controller 可以通过 repcon 模板来创建多个 Pod 副本，同样也可以直接复制已存在的 Pod，需要通过 Label selector 来关联。

（5）Services：Kubernetes 最外围的单元，通过虚拟一个访问 IP 及服务端口，可以访问我们定义好的 Pod 资源，目前的版本是通过 iptables 的 nat 转发来实现的，转发的目标端口为 Kube_proxy 生成的随机端口，目前只提供 Google 云上的访问调度。

（6）Volumes：一个能够被容器访问的目录，或许还包含一些数据，与 Docker Volumes 有点儿类似。

（7）Labels：用于区分 Pod、Service、Replication Controller 的 key/value 键值对，仅使用在 Pod、Service、Replication Controller 之间的关系识别，但对这些单元本身进行操作时得使用 name 标签。

（8）Accessing the API：Kubernetes 中端口、IP、代理服务器和防火墙规则。

（9）Kubernetes Web Interface：访问 Kubernetes Web 接口。

（10）Kubectl Command Line Interface：Kubernetes 命令行接口，如 kubectl。

13.3.2 环境说明

本案例使用 Centos7.0 作为操作系统环境，Kubernetes 的版本为 V0.6.2，Etcd 版本为 0.4.6，Docker 的版本为 1.3.2，服务器角色环境见表 13-1。

表 13-1 服务器角色环境

角色	主机名	IP	环境说明
Master	SN2014-12-200	192.168.1.200	Kubernetes
Etcd	SN2014-12-010	192.168.1.10	Etcd
Minion	SN2014-12-201	192.168.1.201	Kubernetes+docker
Minion	SN2014-12-202	192.168.1.202	Kubernetes+docker

13.3.3　环境部署

下面为系统初始化工作，操作对象为所有主机，本节涉及系统安装都选择“最小化安装”，保证系统的精简，节省无谓的资源损耗，以下为安装系统基础包及 epel 源：

```
    # yum -y install wget ntpdate bind-utils
    # wget http://mirror.centos.org/centos/7/extras/x86_64/Packages/epel-release-7-2.
noarch.rpm
    # yum update
```

CentOS 7.0 默认使用的是 firewall 作为防火墙管理，这里改为 iptables 防火墙（熟悉度更高，非必需）。

（1）关闭 firewall

```
    # systemctl stop firewalld.service #停止 firewall
    # systemctl disable firewalld.service #禁止 firewall 开机启动
```

（2）安装 iptables 防火墙

```
    # yum install iptables-services #安装
    # systemctl start iptables.service #最后重启防火墙使配置生效
    # systemctl enable iptables.service #设置防火墙开机启动
```

1. 部署 Etcd 环境

本案例为 192.168.1.10 主机，详细参考 9.1.1 章节内容，此处省略。

2. 安装 Kubernetes

操作对象为所有 Master、Minion 主机。

本案例使用最简单的 yum 源方式进行安装，默认会安装 etcd、docker 和 cadvisor 等相关包，可以根据不同角色去启动及配置不同服务，安装方式如下：

```
    # curl https://copr.fedoraproject.org/coprs/eparis/kubernetes-epel-7/
repo/epel-7/eparis-kubernetes-epel-7-epel-7.repo -o /etc/yum.repos.d/eparis-
kubernetes-epel-7-epel-7.repo
    #yum -y install kubernetes
```

由于 yum 源方式安装的版本相对比较老旧，需要升级至最新版本 V0.6.2（二进制包），直接覆盖 bin 文件即可，方法如下：

```
    # mkdir -p /home/install && cd /home/install
    # wget https://github.com/GoogleCloudPlatform/kubernetes/releases/download/
v0.6.2/kubernetes.tar.gz
```

```
# tar -zxvf kubernetes.tar.gz
# tar -zxvf kubernetes/server/kubernetes-server-linux-amd64.tar.gz
# cp kubernetes/server/bin/kube* /usr/bin
```

接下来我们校验安装结果，出现图 13-5 所示的提示信息则说明安装正确。

```
[root@SN2014-12-200 ~]# /usr/bin/kubectl version
Client Version: version.Info{Major:"0", Minor:"6+", GitVersion:"v0.6.2", GitCommit:"729fde276613eedcd99ecf5b93f095b8deb64eb4", GitTreeState:"clean"}
Server Version: &version.Info{Major:"0", Minor:"6+", GitVersion:"v0.6.2", GitCommit:"729fde276613eedcd99ecf5b93f095b8deb64eb4", GitTreeState:"clean"}
```

图 13-5 命令行接口版本信息

3. Kubernetes 配置（仅 Master 主机）

在角色 Master 运行三个组件，包括 apiserver、scheduler、controller-manager，相关配置项也只涉及这三块。其中 scheduler 为调度器，负责收集和分析当前 Kubernetes 集群中所有 Minion 节点的资源的负载情况，根据这些信息合理地分发新建的 Pod 到 Kubernetes 集群中可用的节点当中。详细的配置信息如下：

【 /etc/kubernetes/config 】

```
# Comma seperated list of nodes in the etcd cluster
# 指定 ECTD 服务器 IP 及服务端口
KUBE_ETCD_SERVERS="--etcd_servers=http://192.168.1.10:4001"

# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privleged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"
```

【 /etc/kubernetes/apiserver 】

```
# The address on the local server to listen to.
# API 监听主机地址
KUBE_API_ADDRESS="--address=0.0.0.0"

# The port on the local server to listen on.
# API 监听端口
KUBE_API_PORT="--port=8080"

# How the replication controller and scheduler find the kube-apiserver
# 复制与调度使用的 API 主机与端口
KUBE_MASTER="--master=192.168.1.200:8080"
```

```
# Port minions listen on
# Minion 监听端口
KUBELET_PORT="--kubelet_port=10250"

# Address range to use for services
# 定义 SERVICE 随机网段
KUBE_SERVICE_ADDRESSES="--portal_net=10.254.0.0/16"

# Add you own!
KUBE_API_ARGS=""
```

【/etc/kubernetes/controller-manager】

```
# Comma seperated list of minions
# 定义 Minion 主机列表
KUBELET_ADDRESSES="--machines= 192.168.1.201,192.168.1.202"

# Add you own!
KUBE_CONTROLLER_MANAGER_ARGS=""
```

【/etc/kubernetes/scheduler】

```
# Add your own!
KUBE_SCHEDULER_ARGS=""
```

最后一步就是启动 master 端的相关服务了。在 Centos7 中，服务管理使用的是 systemctl，有关 systemctl 的用法可参考：http://www.linuxbrigade.com/centos-7-rhel-7-systemd-commands/，具体的操作步骤如下：

```
# systemctl daemon-reload
# systemctl start kube-apiserver.service kube-controller-manager.service
kube-scheduler.service
# systemctl enable kube-apiserver.service kube-controller-manager.service
kube-scheduler.service
```

4. Kubernetes 配置（仅 Minion 主机）

Minion（部署 Docker 环境的主宿机）运行两个组件，包括 kubelet、proxy，相关配置项也只针对这两部分，开始之前需要对 Docker 的启动参数进行修改，以便后面提供远程 API 操作支持，具体操作如下：

【/etc/sysconfig/docker】

```
# Modify these options if you want to change the way the docker daemon runs
OPTIONS=--selinux-enabled -H tcp://0.0.0.0:2375 -H fd://
# Location used for temporary files, such as those created by
```

```
# docker load and build operations. Default is /var/lib/docker/tmp
# Can be overriden by setting the following environment variable.
# DOCKER_TMPDIR=/var/tmp
```

修改 Minion 防火墙配置，通常 master 找不到 Minion 主机多半是由于 10250 端口没有连通，插入以下 iptables 规则：

```
iptables -I INPUT -s 192.168.1.200 -p tcp --dport 10250 -j ACCEPT
```

下面为修改 Kubernetes Minion 端的相关配置。以 192.168.1.201 主机为例，其他 Minion 主机配置同理，也可以通过 scp 命令远程复制至其他 Minion 主机。

【/etc/kubernetes/config】

```
# Comma seperated list of nodes in the etcd cluster
# 指定 ECTD 服务器 IP 及服务端口
KUBE_ETCD_SERVERS="--etcd_servers=http://192.168.1.10:4001"

# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privleged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"
```

【/etc/kubernetes/kubelet】

```
###
# kubernetes kubelet (minion) config

# The address for the info server to serve on (set to 0.0.0.0 or "" for
all interfaces)
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
# 定义监听的服务端口
KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
# 定义 Minion 主机名
KUBELET_HOSTNAME="--hostname_override=192.168.1.201"

# Add your own!
KUBELET_ARGS=""
```

【 /etc/kubernetes/proxy 】

```
KUBE_PROXY_ARGS=""
```

启动 Minion 端 Kubernetes 服务，命令如下：

```
# systemctl daemon-reload
# systemctl enable docker.service kubelet.service kube-proxy.service
# systemctl start docker.service kubelet.service kube-proxy.service
```

13.3.4　API 常用操作

Kubernetes 提供了两种 API 操作方式，一种为 kubectl 命令行，另一种为 HTTP REST 方式，笔者推荐第二种方式，优势是可以在非 master 主机上通过 HTTP 方式调用操作，且及时性更高。

（1）命令行方式。

```
# kubectl get minions     # 查查看 Minion 主机
# kubectl get pods        # 查看 pods 清单
# kubectl get services 或 kubectl get services -o json     # 查看 service 清单
# kubectl get replicationControllers     # 查看 replicationControllers 清单
# for i in `kubectl get pod|tail -n +2|awk '{print $1}'`; do kubectl delete
pod $i; done                    # 删除所有 pods
```

（2）Server api for REST 方式。

```
# curl -s -L http://192.168.1.200:8080/api/v1beta1/version | python -mjson.tool
                                # 查看 kubernetes 版本
# curl -s -L http://192.168.1.200:8080/api/v1beta1/pods | python -mjson.tool
                                # 查看 pods 清单
# curl -s -L http://192.168.1.200:8080/api/v1beta1/replicationControllers |
python -mjson.tool              # 查看 replicationControllers 清单
# curl -s -L http://192.168.1.200:8080/api/v1beta1/minions | python -m json.tool
                                # 查查看 minion 主机
# curl -s -L http://192.168.1.200:8080/api/v1beta1/services | python -m json.tool
                                # 查看 service 清单
```


注意　在 Kubernetes V0.6 版本后，所有的操作命令都整合至 kubectl，包括 kubecfg、kubectl.sh、kubecfg.sh 等。

13.3.5　创建 pod 单元

在 Kubernetes 中支持使用 json 格式来描述资源或对象，比如 pod、replication、

service 等，下面创建一个 pod 的 json 描述。

```
# /home/kubermange/pods && cd /home/kubermange/pods
```

【 /home/kubermange/pods/apache-pod.json 】

```
{
  "id": "fedoraapache",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "fedoraapache",
      "containers": [{
        "name": "fedoraapache",
        "image": "fedora/apache",
        "ports": [{
          "containerPort": 80,
          "hostPort": 8080
        }]
      }]
    }
  },
  "labels": {
    "name": "fedoraapache"
  }
}
```

apache-pod.json 有两个较关键的配置项，其中“ containers”标签为定义一个完整的容器描述，包括指定镜像（image）、名称（name）、端口映射（ports）等，另一个为“ labels”标签，定义该 pod 的引用标志，通过一个 key:value 来定义，本例为 "name": "fedoraapache"，名称 "fedoraapache" 代表了这个 pod。

下一步，执行 kubectl 命令创建此 pod，使用 create 参数，如下：

```
# kubectl create -f apache-pod.json
```

使用 get 参数获取此 pod 信息，执行以下命令：

```
# kubectl get pod
```

返回 pod 的信息如图 13-6 所示。

NAME	IMAGE(S)	HOST	LABELS	STATUS
fedoraapache	fedora/apache	192.168.1.202/	name=fedoraapache	Running

图 13-6 返回 pod 的信息

从返回的 pod 信息可以看到，该 pod 被分配至 192.168.1.202 主机上，状态为 Runing。启动浏览器访问 http://192.168.1.202:8080/，对应的服务端口（如 8080）切记在 iptables 中已添加，出现图 13-7 所示的结果，说明我们已经成功创建了一个 pod 单元。

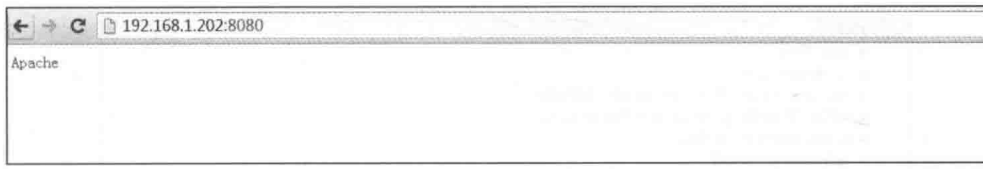


图 13-7　访问容器服务截图

最后，我们观察下 Etcd 键值存储平台发生了什么，可以通过 etcd-browser 工具来查看，结果如图 13-8 所示，nodes 节点下面为集群所有 minion 主机列表，pods 节点多了一个 default 子节点，单击可以看到创建好的“fedoraapache”pod 信息。

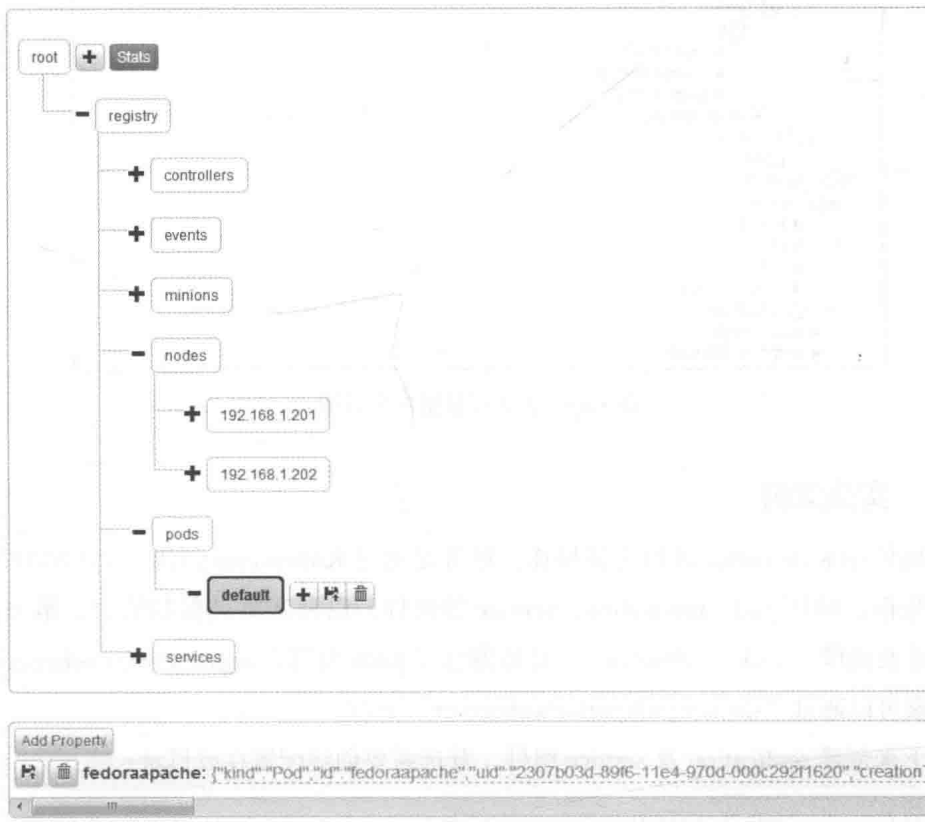


图 13-8　etcd-browser 页面显示信息

为了方便读者更清楚看到完整的pod信息，笔者将key“fedoraapache”对应的value json信息进行格式，可以使用在线json工具格式化平台，如http://www.bejson.com/jsonview2/，效果如图13-9所示。

图13-9 pod信息格式化截图

13.3.6 实战案例

下面使用Kubernetes进行实战操作，任务是通过Kubernetes创建一个LNMP架构的服务集群，使用pod、replication、service等组件，以及观察其复制能力、服务接入特点，涉及镜像“yorko/webserver”，该镜像已经push至官方registry.hub.docker.com仓库，大家可以通过“docker pull yorko/webserver”下载。

接下来创建replication及service组件，首选需要创建配置存放目录：

```
# mkdir -p /home/kubermange/replication && mkdir -p /home/kubermange/service
# cd /home/kubermange/replication
```

第一步，创建一个 replication，本例直接在 replication 中创建 pod 并复制。当然，也可以单独创建 pod 再通过 replication 来复制，下面为完成的 replication 描述信息。

【 replication/lnmp-replication.json 】

```
{
  "id": "webserverController",
  "kind": "ReplicationController",
  "apiVersion": "v1beta1",
  "labels": {"name": "webserver"},
  "desiredState": {
    "replicas": 6,
    "replicaSelector": {"name": "webserver_pod"},
    "podTemplate": {
      "desiredState": {
         "manifest": {
           "version": "v1beta1",
           "id": "webserver",
           "volumes": [
             {"name":"httpconf", "source":{"hostDir":{"path":"/etc/httpd/
conf"}}},
             {"name":"httpconfd", "source":{"hostDir":{"path":"/etc/httpd/
conf.d"}}},
             {"name":"httproot", "source":{"hostDir":{"path":"/data"}}}
           ],
           "containers": [{
             "name": "webserver",
             "image": "yorko/webserver",
             "command": ["/bin/sh", "-c", "/usr/bin/supervisord -c /etc/
supervisord.conf"],
             "volumeMounts": [
               {"name":"httpconf", "mountPath":"/etc/httpd/conf"},
               {"name":"httpconfd", "mountPath":"/etc/httpd/conf.d"},
               {"name":"httproot", "mountPath":"/data"}
             ],
             "cpu": 100,
             "memory": 50000000,
             "ports": [{
               "containerPort": 80,
             },{
               "containerPort": 22,
           }]
         }]
       }
```

```
        },
        "labels": {"name": "webserver_pod"},
      },
  }
}
```

在 lnmp-replication.json 描述配置中，实现了一个 LNMP 架构的容器的 pod，并且复制了六份，其中“replicas”指定复制的份数，“replicaSelector”为复制选择器，即复制的 pod 对象，与 podTemplate（pod 模板）的 labels 标签一致。完整的容器的描述信息在 desiredState.desiredState 节点中定义，包括 volumes、cpu、memory、ports 等信息，理论上都可以与命令行 docker run 一一对应上。

执行创建命令，同样使用 kubectl 命令，如下：

```
# kubectl create -f lnmp-replication.json
```

执行 kubectl get pod 命令观察生成的 pod 副本清单，发现已经生成了六个 pod 副本，且平均分配至不同主宿机上，都处于运行状态。

```
[root@SN2014-12-200 replication]# kubectl get pod
NAME                                    IMAGE(S)            HOST
LABELS                STATUS
84150ab7-89f8-11e4-970d-000c292f1620    yorko/webserver     192.168.1.202/
name=webserver_pod    Running
84154ed5-89f8-11e4-970d-000c292f1620    yorko/webserver     192.168.1.201/
name=webserver_pod    Running
840beb1b-89f8-11e4-970d-000c292f1620    yorko/webserver     192.168.1.202/
name=webserver_pod    Running
84152d93-89f8-11e4-970d-000c292f1620    yorko/webserver     192.168.1.202/
name=webserver_pod    Running
840db120-89f8-11e4-970d-000c292f1620    yorko/webserver     192.168.1.201/
name=webserver_pod    Running
8413b4f3-89f8-11e4-970d-000c292f1620    yorko/webserver     192.168.1.201/
name=webserver_pod    Running
```

第二步，创建一个 service 来对外提供服务，通过指定 selector 的“"name": "webserver_pod"”参数与 pods 进行关联。

【service/lnmp-service.json】

```
{
  "id": "webserver",
  "kind": "Service",
```

```
    "apiVersion": "v1beta1",
    "selector": {
      "name": "webserver_pod",
    },
    "protocol": "TCP",
    "containerPort": 80,
    "port": 8080
  }
```

通过“protocol”指定服务的协议，如 TCP，“containerPort”为指定容器的服务端口，“port”为映射的服务端口，最后执行创建 service 命令，如下：

```
# kubectl create -f lnmp-service.json
```

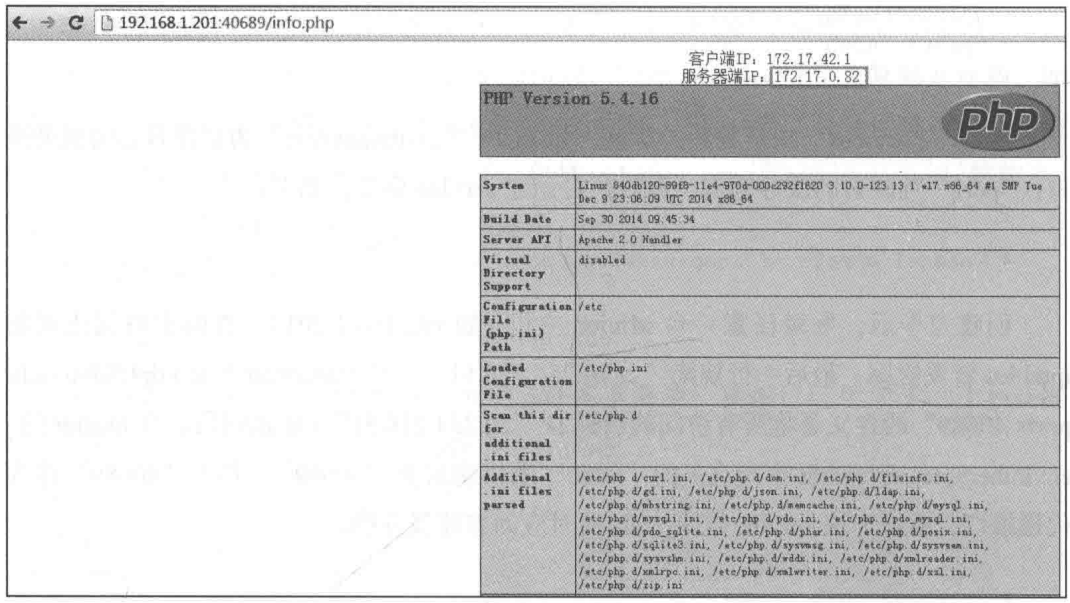
创建完毕后，登录任意一台 Minion 主机（如 192.168.1.201），查询主宿机生成的 iptables 转发规则，最后一行规则“... 10.254.216.51　　/* webserver */ tcp dpt:8080 redir ports 40689”的含义是将所有访问的目标 IP“10.254.216.51”（虚拟网段，在 master 的 /etc/kubernetes/apiserver 中定义）的“8080”端口映射至“40689”。其中“40689”作为代理端口，后端为该 service 定义 pods 所对应的容器服务端口。

```
# iptables -nvL -t nat
Chain KUBE-PROXY (2 references)
pkts bytes target     prot opt in     out     source               destination
    2   120 REDIRECT   tcp  --  *      *       0.0.0.0/0
10.254.102.162       /* kubernetes */ tcp dpt:443 redir ports 47700
    1    60 REDIRECT   tcp  --  *      *       0.0.0.0/0
10.254.28.74         /* kubernetes-ro */ tcp dpt:80 redir ports 60099
    0     0 REDIRECT   tcp  --  *      *       0.0.0.0/0
10.254.216.51        /* webserver */ tcp dpt:8080 redir ports 40689
```

最后，我们可以访问测试页来观察代理端口均衡后端容器的效果，访问 http://192.168.1.201:40689/info.php，刷新浏览器后发现 proxy 后端的变化，默认为随机轮循算法，详细如图 13-10 所示。

当前版本接入层官方侧重点还放在 GCE（Google Compute Engine）的对接优化，如本案例中的 10.254.216.0/24 虚拟网段。针对个人私有云还未推出一套可行的接入解决方案。在 V0.5 版本中才引用 service 代理转发的机制，且是通过 iptables 来实现的，在高并发下性能令人担忧。但笔者依然看好 Kubernetes 未来

的发展，至少目前还未看到另外一个成体系、具备良好生态圈的平台，相信在V1.0时就会具备生产环境的服务支撑能力。



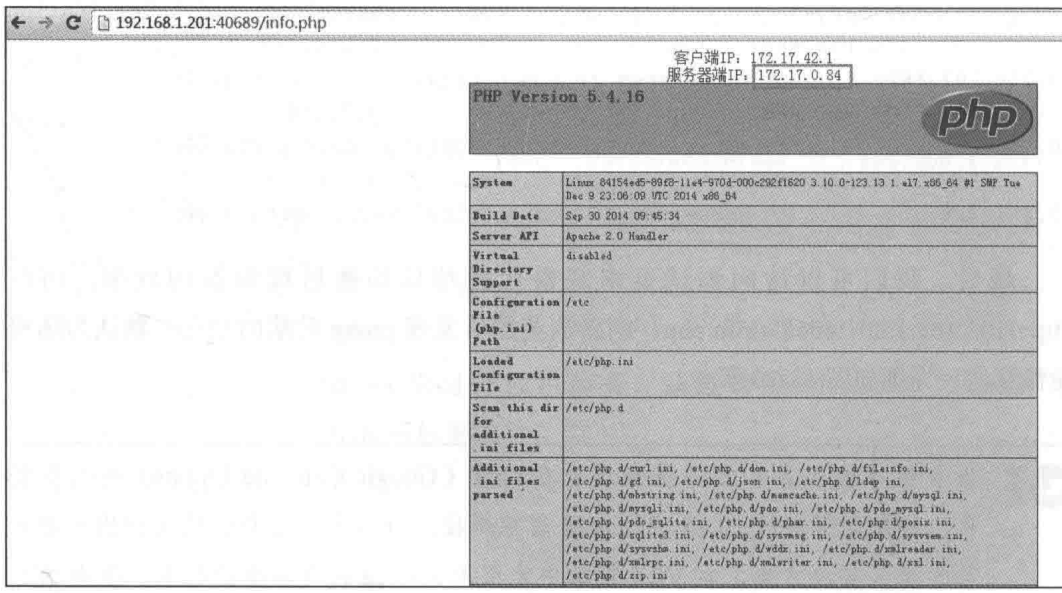
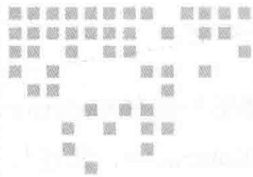


图 13-10 访问 Kubernetes 代理端口地址效果

13.4　本章小结

本章节介绍了 Etcd 的基本用法，以及 Docker 容器监控组件 Cadvisor 的部署与使用，最后介绍了业界最为流行的 Docker 编排工具——Kubernetes，通过与 Etcd 组件的结合，实现了 Kubernetes 相关配置信息的存储，同时介绍了 Kubernetes 最小调度单元 pod 及常见 API 的操作，最后通过一个实战案例，帮助读者对 Kubernetes 这个工具有一个更加清晰的认识。



Chapter 14 第 14 章

构建 Docker 高可用及自动发现架构实践

Docker 的生态日趋成熟，开源社区也不断孵化出优秀的周边项目，如覆盖网络、监控、维护、部署、开发等方面。帮助开发、运维人员快速构建、运营 Docker 服务环境，其中也不乏大公司的身影，如 Google、IBM、Red Hat，甚至微软也宣称后续将提供 Docker 在 Windows 平台的支持。Docker 的发展前景一片大好。但在企业当中，如何选择适合自己的 Docker 构建方案？可选的方案有 Kubernetes 与 CoreOS（都已整合各类组件），另外一种方案为 Haproxy+Etcd+Confd，采用松散式的组织结构，但各个组件之间的通信是非常严密的，且扩展性更强，定制也更加灵活。下面将详细介绍如何使用 Haproxy+Etcd+Confd 构建一个高可用及自动发现的 Docker 基础架构。

14.1 架构优势

笔者约定由 Haproxy+Etcd+Confd+Docker 构建的基础服务平台简称“ HECD”架构，整合了多种开源组件，看似松散的结构，事实上已经是一个有机的整体，它们互相联系、互相作用，是 Docker 生态圈中最理想的组合之一，具有以下优势：

- ❑ 自动、实时发现及无感知服务刷新。
- ❑ 支持任意多台 Docker 主宿机。
- ❑ 支持多种 App 接入且打散至不分主宿机。

- 采用 Etcd 存储信息，集群支持可靠性高。
- 采用 Confd 配置引擎，支持各类接入层，如 Nginx。
- 支持负载均衡、故障迁移。
- 具备资源弹性，伸缩自如（通过生成、销毁容器实现）。

14.2　架构介绍

在 HECD 架构中，首先管理员操作 Docker Client，除了提交容器（Container）启动与停止指令外，还通过 REST-API 方式向 Etcd（K/V）存储组件注册容器信息，包括容器名称、主宿机 IP、映射端口等。Confd 配置组件会定时查询 Etcd 组件获取最新的容器信息，根据定义好的配置模板生成 Haproxy 配置文件 Haproxy.cfg，并且自动 reload haproxy 服务。用户在访问业务服务时，完全没有感知后端 App 的上线、下线、切换及迁移，达到了自动发现、高可用的目的。详细架构图如图 14-1 所示。

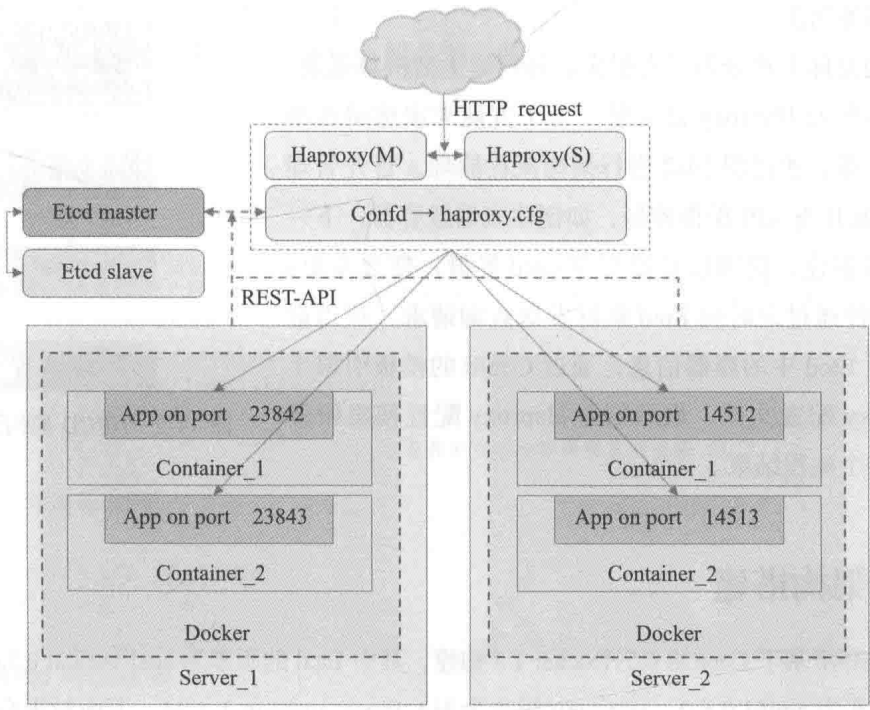


图 14-1　HECD 架构图

在开始对架构做详细介绍之前，我们先逐一对架构中涉及的各个组件及发挥的作用进行简单介绍。

1. Etcd 介绍

Etcd 是一个高可用的 Key/Value 存储系统，主要用于分享配置和服务发现，更多 Etcd 介绍参见 13.1 节。在本架构中负责存储容器的注册信息。

2. Confd 介绍

Confd 是一个轻量级的配置管理工具。通过查询 Etcd，结合配置模板引擎，保持本地配置最新，同时具备定期探测机制，配置变更自动 reload。在本架构中负责读取 Etcd 集群中容器的注册信息并刷新接入层 Haproxy 的配置。

3. Haproxy 介绍

Haproxy 是提供高可用性、负载均衡及基于 TCP 和 HTTP 应用的代理，支持虚拟主机，它是免费、快速并且可靠的一种解决方案。在本架构中作为业务的接入层，包括容器服务的负载均衡、故障迁移等功能。

架构总体上拆分为三大层次，分别为主宿机容器层、Etcd 集群层及 Haproxy 接入层。为了方便大家理解各组件间的关系，通过图 14-2 进行架构流程梳理，首先管理员通过 Shell 或 API 操作容器，如创建或销毁容器；下一步将容器创建、销毁信息提交至 Etcd 集群，使之变更；Confd 组件通过定时向 Etcd 集群发送查询请求，获得最新提交至 Etcd 中的容器信息，通过 Confd 的模板引擎生成 Haproxy 配置文件，最后刷新 Haproxy 配置使之服务生效，整个流程结束。

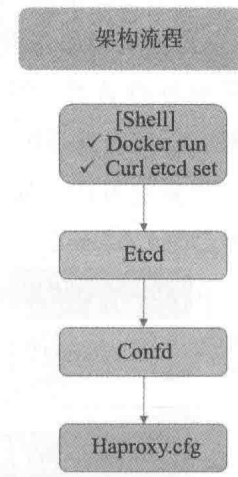


图 14-2 HECD 架构流程图

14.3 架构搭建

平台环境基于 CentOS 6.5+Docker 1.3 构建，其中 Etcd 的版本为 etcd version 0.5.0-alpha，Confd 版本为 confd 0.6.2，Haproxy 版本为 HA-Proxy version 1.4.24。下面对平台的运行环境、安装部署、组件说明等进行详细说明，环境设备角色表如下：

角色	主机名	IP	环境说明
接入层	SN2013-08-020	192.168.1.20	Haproxy+confd
存储层	SN2012-07-021	192.168.1.21	Etcd
主宿机	SN2012-07-022	192.168.1.22	Docker
主宿机 N	…	…	Docker

14.3.1　组件环境部署

下面针对三种不同角色组件进行安装部署，本案例采用了 yum 及二进制安装方式，读者可以根据实际情况进行调整。

1. Docker 环境部署

使用 SSH 终端登录主宿机 192.168.1.22 服务器，执行以下命令：

```
# yum -y install docker-io
 # service docker start
 # chkconfig docker on
```

2. Haproxy、Confd 环境部署

SSH 终端登录接入层 192.168.1.20 服务器，执行以下命令：

（1）安装 haproxy

```
# yum -y install haproxy          # 采用 yum 安装模式；
```

（2）安装 confd

```
    # wget https://github.com/kelseyhightower/confd/releases/download/v0.6.3/
confd-0.6.3-linux-amd64
    # mv confd /usr/local/bin/confd
    # chmod +x /usr/local/bin/confd
    # /usr/local/bin/confd -version
    confd 0.6.2                     # 显示版本号则说明成功安装；
```

命令行操作如下：

```
    # etcdctl get /message
    Hello world
```

3. Etcd 环境部署

SSH 终端登录存储层 192.168.1.21 服务器，本案例使用单机模式，生产环境建议使用集群模式来提供服务，可解决单点问题，详细可参考官网文档（https://github.com/

coreos/etcd/blob/master/Documentation/clustering.md）部署，本文此处省略。执行以下命令进行安装：

```
# mkdir -p /home/install && cd /home/install
# wget https://github.com/coreos/etcd/releases/download/v0.4.6/etcd-v0.4.6-linux-amd64.tar.gz
# tar -zxvf etcd-v0.4.6-linux-amd64.tar.gz
# cd etcd-v0.4.6-linux-amd64
# cp etcd* /bin/
# /bin/etcd -version
etcd version 0.4.6     # 显示版本号则说明成功安装；
```

14.3.2 Etcd 配置

当各个组件安装完毕后，我们就可以对组件的启动参数、配置文件进行操作，由于Etcd是一个轻量级的K/V存储平台，启动时指定相关参数即可，无须配置。

```
# mkdir /data/etcd     # 创建数据存储目录；
# /bin/etcd -name etcdserver -peer-addr 192.168.1.21:7001 -addr 192.168.1.21:4001 -data-dir /data/etcd -peer-bind-addr 0.0.0.0:7001 -bind-addr 0.0.0.0:4001 &
```

Etcd是具备集群服务能力的，参数“-peer-addr”指定与其他节点通信的地址；参数“-addr”指定服务监听地址；参数“-data-dir”指定数据存储目录。由于Etcd是通过REST-API方式进行交互的，常见操作参见11.1.2节。

此启动参数在V0.4.6版本中测试通过，其他不同版本可能存在细微差异，可从官网https://github.com/coreos/etcd了解最新参数说明。

14.3.3 Confd 配置

由于Haproxy的配置文件是由Confd组件生成的，要求Confd务必要与Haproxy安装在同一台主机上，Confd的配置有两类：一类为Confd资源配置文件，即定义外部应用程序的基本信息，默认路径为“/etc/confd/conf.d”目录；另一类为配置模板文件，默认路径为“/etc/confd/templates”。具体配置如下。

首先创建两类配置文件存储目录。

```
# mkdir -p /etc/confd/{conf.d,templates}
```

1. 配置资源文件

详细参见以下配置文件，其中“ src”为指定模板文件名称（默认到路径 /etc/confd/templates 中查找）；“ dest”指定生成的 Haproxy 配置文件路径；“ keys”指定关联 Etcd 中 key 的 URI 列表；“ reload_cmd”指定服务重载的命令，本例中配置成 Haproxy 的 reload 命令。

【 /etc/confd/conf.d/ haproxy.toml 】

```
[template]
src = "haproxy.cfg.tmpl"
dest = "/etc/haproxy/haproxy.cfg"
keys = [
  "/app/servers",
]
reload_cmd = "/etc/init.d/haproxy reload"
```

2. 应用配置模板文件

Confd 模板采用了 Go 语言的文本模板引擎，可实现通过自身的语法对应用程序（Haproxy）配置文件进行灵活处理。具备简单的逻辑语法，包括循环体、处理函数等，本示例的模板文件如下所示，通过 range 循环输出 Key 及 Value 信息。

【 /etc/confd/templates/haproxy.cfg.tmpl 】

```
global
        log 127.0.0.1 local3
        maxconn 5000
        uid 99
        gid 99
        daemon

defaults
        log 127.0.0.1 local3
        mode http
        option dontlognull
        retries 3
        option redispatch
        maxconn 2000
        contimeout  5000
        clitimeout  50000
        srvtimeout  50000

listen frontend 0.0.0.0:80
        mode http
        balance roundrobin
        maxconn 2000
```

```
    option forwardfor
    {{range gets "/app/servers/*"}}
    server {{base .Key}} {{.Value}} check inter 5000 fall 1 rise 2
    {{end}}

    stats enable
    stats uri /admin-status
    stats auth admin:123456
    stats admin if TRUE
```

3. Confd 模板引擎介绍

本小节详细介绍了 Confd 模板引擎基础语法与示例，方便大家操作不限于 haproxy.cfg 配置文件，同样也可以用于 nginx.conf 或其他，首先提交示例用到的 Key 信息到 Etcd 主机，其中 "/app/servers" 为应用键信息，与“ /etc/confd/conf.d/ haproxy.toml”配置文件中的“keys”参数值保持一致。详细操作如下：

```
# curl -XPUT http://192.168.1.21:4001/v2/keys/app/servers/backstabbing_
  rosalind -d value="192.168.1.22:49156"
# curl -XPUT http://192.168.1.21:4001/v2/keys/app/servers/cocky_morse -d
  value="192.168.1.22:49158"
# curl -XPUT http://192.168.1.21:4001/v2/keys/app/servers/goofy_goldstine
  -d value="192.168.1.22:49160"
# curl -XPUT http://192.168.1.21:4001/v2/keys/app/servers/prickly_
  blackwell -d value="192.168.1.22:49162"
```

下面介绍 Confd 模板引擎常用语法及结果输出。

（1）base 函数

作为 path.Base 函数的别名，获取 URI 路径最后一段。

```
{{ with get "/app/servers/prickly_blackwell"}}
  server {{base .Key}} {{.Value}} check
{{end}}
```

结果输出：

```
prickly_blackwell 192.168.1.22:49162
```

（2）get 函数

返回一对匹配的 KV，找不到则返回错误。

```
{{with get "/app/servers/prickly_blackwell"}}
  key: {{.Key}}
  value: {{.Value}}
{{end}}
```

结果输出：

```
/app/servers/prickly_blackwell 192.168.1.22:49162
```

（3）gets 函数

返回所有匹配的 KV，找不到则返回错误。

```
{{range gets "/app/servers/*"}}
  {{.Key}} {{.Value}}
{{end}}
```

结果输出：

```
/app/servers/backstabbing_rosalind 192.168.1.22:49156
/app/servers/cocky_morse 192.168.1.22:49158
/app/servers/goofy_goldstine 192.168.1.22:49160
app/servers/prickly_blackwell 192.168.1.22:49162
```

（4）getv 函数

返回一个匹配 key 的字符串型 Value，找不到则返回错误。

```
{{getv "/app/servers/cocky_morse"}}
```

结果输出：

```
192.168.1.22:49158
```

（5）getvs 函数

返回所有匹配 key 的字符串型 Value，找不到则返回错误。

```
{{range getvs "/app/servers/*"}}
  value: {{.}}
{{end}}
```

结果输出：

```
value: 192.168.1.22:49156
value: 192.168.1.22:49158
value: 192.168.1.22:49160
value: 192.168.1.22:49162
```

（6）split 函数

对输入的字符串做 split 处理，即将字符串按指定分隔符拆分成数组。

```
{{ $url := split (getv "/app/servers/cocky_morse") ":" }}
  host: {{index $url 0}}
  port: {{index $url 1}}
```

结果输出：

```
host: 192.168.1.22
port: 49158
```

（7）ls 函数

返回所有的字符串型 subkey，找不到则返回错误。

```
{{range ls "/app/servers/"}}
  subkey: {{.}}
{{end}}
```

结果输出：

```
subkey: backstabbing_rosalind
subkey: cocky_morse
subkey: goofy_goldstine
subkey: prickly_blackwell
```

（8）lsdir 函数

返回所有的字符串型子目录，找不到则返回一个空列表。

```
{{range lsdir "/app/"}}
  subdir: {{.}}
{{end}}
```

结果输出：

```
subdir: servers
```

更多语法介绍见 http://golang.org/pkg/text/template/。

4. 启动 Confd 及 Haproxy 服务

下面为启动 Confd 服务命令行，参数“interval”为指定探测 Etcd 的频率，单位为秒，参数“-node”为指定 Etcd 监听服务地址，以便获取容器注册的信息。

```
# /usr/local/bin/confd -verbose -interval 10 -node '192.168.1.21:4001'
  -confdir /etc/confd > /var/log/confd.log &
# /etc/init.d/haproxy start
```

14.3.4 容器提交注册

前面 HECD 架构介绍，有提到容器的操作会即时注册到 Etcd 组件中，是通过 curl 命令以 REST-API 方式提交的，下面详细介绍通过 Shell 及 Python-API 两种方式的实现方法，支持容器启动、停止的联动。

1. Shell 脚本实现方法

实现的原理是通过获取“Docker run ***”命令输出的 Container ID，通过“docker inspect Container ID”得到详细的容器信息，分析出容器服务映射的外部端口及容器名称，将以“/app/servers/ 容器名称”作为 Key，“主宿机 : 映射端口”作为 Value 注册到 Etcd 中。其中 Key 信息前缀（/app/servers）与“/etc/confd/conf.d/haproxy.toml”中的“keys”参数是保持一致的，完整的脚本如下：

【docker.sh】

```
#!/bin/bash
if [ -z $1 ]; then
        echo "Usage: c run <image name>:<version>"
        echo "       c stop <container name>"
        exit 1
fi
if [ -z $ETCD_HOST ]; then
  ETCD_HOST="192.168.1.21:4001"          # 指定默认 Etcd 主机地址;
fi
if [ -z $ETCD_PREFIX ]; then
  ETCD_PREFIX="app/servers"              # 指定业务 Etcd key 信息;
fi
if [ -z $CPORT ]; then
  CPORT="80"                             # 指定默认服务端口;
fi
if [ -z $FORREST_IP ]; then
  # 指定主宿机服务 IP 地址，如 eth0;
  FORREST_IP=`ifconfig eth0| grep "inet addr" | head -1 | cut -d : -f2 | awk
'{print $1}'`
Fi
function launch_container {              # 启动容器处理函数;
    echo "Launching $1 on $FORREST_IP ..."
    CONTAINER_ID=`docker run -d --dns 172.17.42.1 -P -v /data:/data -v /
etc/httpd/conf:/etc/httpd/conf -v /etc/httpd/conf.d:/etc/httpd/conf.d -v /
etc/localtime:/etc/localtime:ro $1 /bin/sh -c "/usr/bin/supervisord -c /etc/
supervisord.conf"`
                                         # 启动容器并获得容器 ID 号;
    PORT=`docker inspect $CONTAINER_ID|grep "\"Ports\"" -A 50|grep "\"$CPORT/
tcp\"" -A 3| grep HostPort|cut -d '"' -f4|head -1` # 根据容器 ID 获得容器映射的随机端口;
    NAME=`docker inspect $CONTAINER_ID | grep Name | cut -d '"' -f4 | sed
"s/\///g"|sed -n 2p`                      # 根据容器 ID 获得容器名称;
    echo "Announcing to $ETCD_HOST..."
    # 向 Etcd 主机提交容器注册信息，内容包括容器服务 IP、端口及名称;
    curl -XPUT "http://$ETCD_HOST/v2/keys/$ETCD_PREFIX/$NAME" -d value="
$FORREST_IP:$PORT"
    echo "$1 running on Port $PORT with name $NAME"
}
```

```
function stop_container {                    # 停止容器处理函数;
    echo "Stopping $1..."
    CONTAINER_ID=`docker ps -a| grep $1 | awk '{print $1}'`
                                             # 根据传入的容器名获得容器 ID
    echo "Found container $CONTAINER_ID"
    docker stop $CONTAINER_ID
    # 向 Etcd 主机提交待删除的 key 信息 ( 容器名称 );
    curl -XDELETE http://$ETCD_HOST/v2/keys/$ETCD_PREFIX/$1 &> /dev/null
    echo "Stopped."
}
if [ $1 = "run" ]; then
  launch_container $2
else
  stop_container $2
fi
```

docker.sh 脚本使用方法非常简单，只需要传入相对的镜像或容器名称即可，如下：

```
启动一个容器
# ./docker.sh run yorko/webserver:v3( 镜像名 )
停止一个容器
# ./docker.sh stop berserk_hopper( 容器名 )
```

2. Docker-py API 实现方法

通过调用 Docker 的 Python API 可实现远程对容器进行操作，包括容器的创建、运行、停止、镜像管理、获取 Docker 对象相关信息等，同时结合 Etcd 的 Python API 模块对 Etcd 进行操作，包括 set 及 delete 等，达到与 Shell 方式一样的效果。很明显，Docker-py 方式更容易扩展，可以无缝与现有运营平台对接。

为兼顾到远程 API 操作支持，需对 Docker 启动文件“ exec”处进行修改，详细内容如下：

【 /etc/init.d/docker 】(Ubuntu 系统默认路径为 /etc/default/docker)

```
$exec -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock -d &>> $logfile &
```

为便于功能模块的引用，拆分成两个 Python 文件，其中一个为启动容器程序，另一个为停止容器程序，下面为完整启动容器程序源码，流程分连接 Docker 主宿机，创建容器并启动，最后获取该容器相关信息注册至 Etcd 主机。

【 docker_run.py 】

```
#!/usr/local/Python/bin/python
import docker
import etcd
import sys
```

```
    Etcd_ip="192.168.1.21"                    # 定义 Etcd 主机地址;
    Server_ip="192.168.1.22"                  # 定义当前连接的主宿机 IP;
    App_port="80"                             # 定义默认服务端口;
    App_protocol="tcp"                        # 定义默认服务协议;
    Image="yorko/webserver:v3"                # 定义默认镜像名;
    Port=""
    Name=""

    idict={}
    rinfo={}
    try:
      # 创建主宿机连接对象;
      c = docker.Client(base_url='tcp://'+Server_ip+':2375',version='1.14',timeout=15)
    except Exception,e:
      print "Connection docker server error:"+str(e)
      sys.exit()

    try:
      rinfo=c.create_container(image=Image,stdin_open=True,tty=True,command="/
usr/bin/supervisord -c /etc/supervisord.conf",volumes=['/data','/etc/httpd/
conf','/etc/httpd/conf.d
    ','/etc/localtime'],ports=[80,22],name=None) # 调用 create_container() 方法创建容器;
      containerId=rinfo['Id']                   # 获取容器 ID;
    except Exception,e:
      print "Create docker container error:"+str(e)
      sys.exit()

    try:
      c.start(container=containerId, binds={'/data':{'bind': '/data','ro': False},'/
etc/httpd/conf':{'bind': '/etc/httpd/conf','ro': True},'/etc/httpd/conf.
d':{'bind': '/etc/htt
    pd/conf.d','ro': True},'/etc/localtime':{'bind': '/etc/localtime','ro':
True}}, port_bindings={80:None,22:None}, lxc_conf=None,publish_all_ports=True,
links=None, privileged=F
    alse,dns='172.17.42.1', dns_search=None, volumes_from=None, network_
mode=None,restart_policy=None, cap_add=None, cap_drop=None)    # 启动容器服务;
    except Exception,e:
      print "Start docker container error:"+str(e)
      sys.exit()
    try:
      idict=c.inspect_container(containerId) # 调用 inspect_container() 方法获取容器信息;
      Name=idict["Name"][1:]
      skey=App_port+'/'+App_protocol
      for _key in idict["NetworkSettings"]["Ports"].keys():
        if _key==skey:
            Port=idict["NetworkSettings"]["Ports"][skey][0]["HostPort"]
    except Exception,e:
```

```
    print "Get docker container inspect error:"+str(e)
    sys.exit()

if Name!="" and Port!="":
    try:
        #连接 Etcd 主机，提交容器注册信息;
        client = etcd.Client(host=Etcd_ip, port=4001)
        client.write('/app/servers/'+Name, Server_ip+":"+str(Port))
        print Name+" container run success!"
    except Exception,e:
        print "set etcd key error:"+str(e)
else:
    print "Get container name or port error."
```

停止容器服务完整程序如下：

【 docker_stop.py 】

```
#!/usr/local/Python/bin/python
import docker
import etcd
import sys

Etcd_ip="192.168.1.21"              #定义 Etcd 主机地址;
Server_ip="192.168.1.22"            #定义当前连接的主宿机 IP;
containerName="grave_franklin"      #指定需要停止容器的名称;

try:
    #创建主宿机连接对象;
    c = docker.Client(base_url='tcp://'+Server_ip+':2375',version='1.14',timeout=10)
    #调用 stop() 方法停止指定容器;
    c.stop('furious_heisenberg')
except Exception,e:
    print str(e)
    sys.exit()

try:
    #连接 Etcd 主机，提交容器删除请求;
    client = etcd.Client(host=Etcd_ip, port=4001)
    client.delete('/app/servers/'+containerName)
    print containerName+" container stop success!"
except Exception,e:
print str(e)
```

由于容器是无状态的，尽量让其以松散的形式存在，映射端口选项要求使用“-P”参数，即使用随机端口的模式，以减少人工干预。

14.4　业务上线

截至目前，HECD 架构已部署完毕，接下来就是让其为我们服务，案例中使用的镜像“ yorko/webserver:v3”是一个已经构建好的“ LAMP 环境”的镜像。在 Docker Hub 中的位置是：https://hub.docker.com/r/yorko/webserver/，在主宿机中执行以下命令来获取该镜像：

```
# docker pull yorko/webserver:v3
```

开始跑起，登录任一台主宿机（本示例为 192.168.1.22 主机），为便于测试，在主宿机容器挂载“/data”分区创建一个 index.php（主页测试）文件，源码如下：

【 /data/index.php 】

```
<!Doctype html><html xmlns=http://www.w3.org/1999/xhtml>
<head>
<meta http-equiv=Content-Type content="text/html;charset=utf-8"><title> 我的主页 </title>
</head>
<body>
        <h1>Hello world</h1>
        <h2> 当前容器：<?=gethostname();  // 输出容器主机名 ?></h2>
</body>
</html>
```

再执行 docker.sh 脚本，当然也可登录一台管理机执行 Docker-py 相关脚本达到同样效果，图 14-3 所示为创建的三个容器截图。

```
[root@SN2013-08-022 docker]# ./docker.sh run yorko/webserver:v3
Launching yorko/webserver:v3 on 192.168.1.22 ...
Announcing to 192.168.1.21:4001...
{"action":"set","node":{"key":"/app/servers/reverent_hypatia","value":"192.168.1.22:49160","modifiedIndex"
:9,"createdIndex":9}}
yorko/webserver:v3 running on Port 49160 with name reverent_hypatia
[root@SN2013-08-022 docker]#
[root@SN2013-08-022 docker]# ./docker.sh run yorko/webserver:v3
Launching yorko/webserver:v3 on 192.168.1.22 ...
Announcing to 192.168.1.21:4001...
{"action":"set","node":{"key":"/app/servers/desperate_bohr","value":"192.168.1.22:49162","modifiedIndex":1
0,"createdIndex":10}}
yorko/webserver:v3 running on Port 49162 with name desperate_bohr
[root@SN2013-08-022 docker]#
[root@SN2013-08-022 docker]# ./docker.sh run yorko/webserver:v3
Launching yorko/webserver:v3 on 192.168.1.22 ...
Announcing to 192.168.1.21:4001...
{"action":"set","node":{"key":"/app/servers/determined_shockley","value":"192.168.1.22:49164","modifiedInd
ex":11,"createdIndex":11}}
yorko/webserver:v3 running on Port 49164 with name determined_shockley
```

图 14-3　创建容器运行结果

如图 14-3 所示，说明已经成功启动了 desperate_bohr、determined_shockley、reverent_hypatia 三个容器，且成功注册至 Etcd 主机，下面我们看看 haproxy.cfg 发生了什么变化。如图 14-4 所示，增加了三条 server ACL 规则（截图框内部分），其中第二列与第三列的信息是从 Ectd 侧定时拉取，通过 Confd 模板引擎进行渲染，最终生成了图 14-4 所示的 haproxy.cfg 配置。

```
defaults
        log 127.0.0.1 local3
        mode http
        option dontlognull
        retries 3
        option redispatch
        maxconn 2000
        contimeout  5000
        clitimeout  50000
        srvtimeout  50000

listen frontend 0.0.0.0:80
        mode http
        balance roundrobin
        maxconn 2000
        option forwardfor

        server desperate_bohr 192.168.1.22:49162 check inter 5000 fall 1 rise 2

        server determined_shockley 192.168.1.22:49164 check inter 5000 fall 1 rise 2

        server reverent_hypatia 192.168.1.22:49160 check inter 5000 fall 1 rise 2


        stats enable
        stats uri /admin-status
        stats auth admin:123456
        stats admin if TRUE
```

图 14-4 haproxy.cfg 部分配置截图

接下来访问接入层 Haproxy 管理地址：http://192.168.1.20/admin-status，可以看到三个容器已处正常服务状态，且具备了负载均衡、故障迁移等功能，如图 14-5 所示。

下面访问接入层 Web 服务地址：http://192.168.1.20/index.php，根据负载均衡策略，会显示当前命令中的容器 ID（容器主机名），刷新页面时会不定期发生变化，如图 14-6 所示。

最后，我们再测试下删除容器的效果，具体操作如图 14-7 所示。我们删除名称为“determined_shockley”的容器，同时脚本会将信息提交至 Etcd 主机，删除此容器对应的 key，以便 Confd 同时更新 haproxy.cfg 配置。

刷新 Haproxy 管理页面，如图 14-8 所示，发现“determined_shockley”对应的主机成员已经被删除，测试删除功能成功。

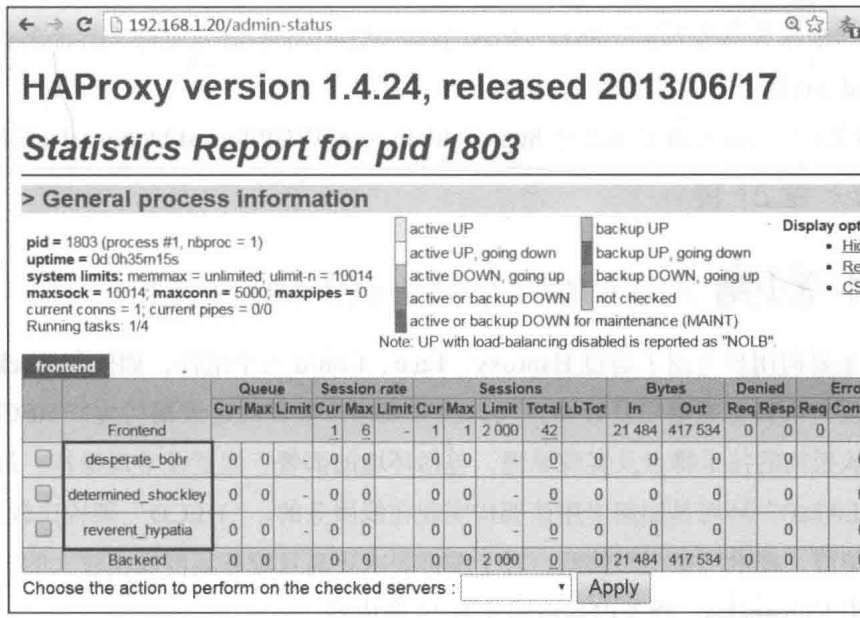


图 14-5　Haproxy 管理页面截图

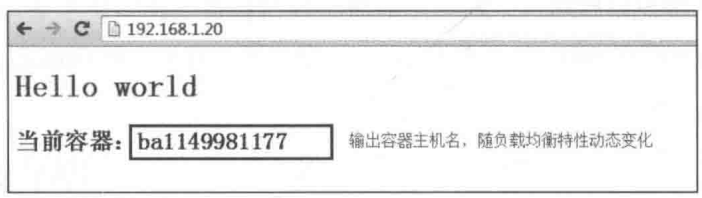


图 14-6　Web 服务首页截图

```
[root@SN2013-08-022 docker]# ./docker.sh stop determined_shockley
Stopping determined_shockley...
Found container 9194032c4aa1
9194032c4aa1
http://192.168.1.21:4001/v2/keys/app/servers/determined_shockley
Stopped.
[root@SN2013-08-022 docker]#
```

图 14-7　删除容器命令截图

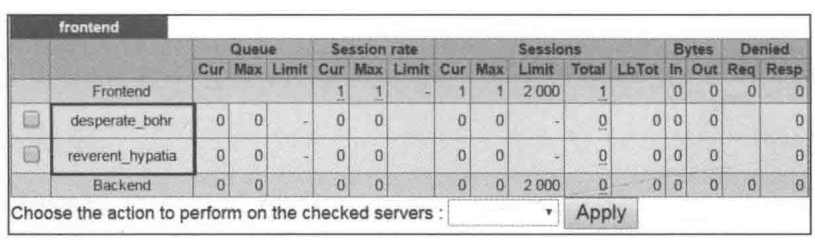


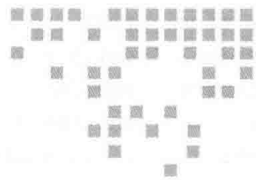
图 14-8　删除后 Haproxy 管理页截图

注意 14.2 节架构参考 http://ox86.tumblr.com/post/90554410668/easy-scaling-with-docker-haproxy-and-confd。

14.3.4 节 docker.sh 脚本参考 https://github.com/AVGP/forrest/blob/master/forrest.sh。

14.5 本章小结

本章主要向用户介绍了通过 Haproxy、Etcd、Confd 三个组件，如何与 Docker 进行结合，实现一个具备高可用及自动发现的服务架构，笔者约定该架构为“HECD”，同时介绍了该架构的技术特点及实现原理，包括环境的部署、配置及常规操作，最后介绍流行的“LAMP”环境是如何应用该架构实现在线服务的。“HECD”架构比较适合中、小型服务集群，技术门槛相对较低。对于编排要求更高且服务集群规模较大的，笔者强烈推荐使用 Kubernetes，相关内容请阅读第 13 章内容。



第 15 章 Chapter 15

Docker Overlay Network 实践

从 1.9 版本开始，Docker 开始支持 Overlay Network，解决了跨主机通信的问题。在这之前，Docker 本身没有一种好的跨主机通信方案，只能通过许多第三方工具来解决，如 weave、flannel 等。本章主要介绍 Docker 自身的 Overlay Network 的实现。

15.1 环境介绍

三台主机：

```
node1 172.17.42.40
node2 172.17.42.41
node3 172.17.42.42
```

其中 node1 和 node2 作为 Docker 容器的主机，在 node3 上运行 Etcd。

OS 内核版本：

```
[root@node1 ~]# uname -a
Linux node1 4.3.3-1.el6.elrepo.x86_64
```

Docker 版本：

```
[root@node1 ~]# docker version
Client:
  Version:      1.10.1
```

```
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   9e83765
 Built:        Thu Feb 11 20:39:58 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.10.1
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   9e83765
 Built:        Thu Feb 11 20:39:58 2016
 OS/Arch:      linux/amd64
```

15.2 容器与容器之间通信

15.2.1 启动 docker daemon

在 node1 和 node2 上启动 Docker：

```
#/usr/bin/docker daemon --cluster-store=etcd://172.17.42.43:2379 --cluster-advertise=eth0:2376
```

- --cluster-store= 参数指向 docker daemon 所使用 key value service 的地址。
- --cluster-advertise= 参数决定了所使用网卡及 docker daemon 端口信息。

当 docker daemon 启动后，会自动创建三个网络：bridge、host、none。

```
[root@node1 ~]# docker network ls
NETWORK ID          NAME                DRIVER
2b2961121f3c        bridge              bridge
adf6d1b8cda1        none                null
8263553e76e4        host                host
[root@node2 ~]# docker network ls
NETWORK ID          NAME                DRIVER
0fc8ed04cbee        bridge              bridge
b6f4e21451d2        none                null
5875c84936d8        host                host
```

15.2.2 创建网络

在 node1 上创建一个名为“overlay”的 Overlay 网络，网络使用的网段为 192.168.10.0/24：

```
[root@node1 ~]# docker network create --internal -d overlay
--subnet=192.168.10.0/24 overlay
1dc144293a1186332d485924fc951d27ed200dfdfd409fc31765093be0b928f0
```

网络信息会自动同步到 node2：

```
[root@node2 ~]# docker network ls
NETWORK ID          NAME                DRIVER
1dc144293a11        overlay             overlay
0fc8ed04cbee        bridge              bridge
b6f4e21451d2        none                null
5875c84936d8        host                host
```

查看网络信息：

```
[root@node1 ~]# docker network inspect overlay
[
  {
"Name": "overlay",
"Id": "1dc144293a1186332d485924fc951d27ed200dfdfd409fc31765093be0b928f0",
"Scope": "global",
"Driver": "overlay",
"IPAM": {
"Driver": "default",
"Options": null,
"Config": [
        {
"Subnet": "192.168.10.0/24"
        }
      ]
    },
"Containers": {},
"Options": {}
  }
]
```

15.2.3　启动容器

在 node1 和 node2 上各启动一个容器：

```
[root@node1 ~]# docker run --net=overlay -itd  --name='vm1' sshd:1.0
305ad9368b0933638899eeaa1cc480393ffd378d5591ae27bdc7f08e24241765
```

```
[root@node2 ~]# docker run --net=overlay -itd  --name='vm2' sshd:1.0
315d6bc52bdd1201d9b9bec4ea3a7542df0c8206f585434d60eaa7f483cf2558
```

容器 vm1 的网络信息：

```
[root@node1 ~]# docker exec vm1 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
31: eth0@if32: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc
noqueue state UP
link/ether 02:42:c0:a8:0a:02 brd ff:ff:ff:ff:ff:ff
inet 192.168.10.2/24 scope global eth0
valid_lft forever preferred_lft forever
  inet6 fe80::42:c0ff:fea8:a02/64 scope link
valid_lft forever preferred_lft forever

[root@node1 ~]# docker inspect vm1
[
...
"NetworkSettings": {
"Bridge": "",
"SandboxID": "3d53b1063c52fe5ceaaee204fc98641c8d8e95cb701589038f14213de3f75fbe",
"HairpinMode": false,
"LinkLocalIPv6Address": "",
"LinkLocalIPv6PrefixLen": 0,
"Ports": {},
"SandboxKey": "/var/run/docker/netns/3d53b1063c52",
"SecondaryIPAddresses": null,
"SecondaryIPv6Addresses": null,
"EndpointID": "",
"Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "",
"IPPrefixLen": 0,
"IPv6Gateway": "",
"MacAddress": "",
"Networks": {
"overlay": {
"IPAMConfig": null,
"Links": null,
"Aliases": null,
"NetworkID": "1dc144293a1186332d485924fc951d27ed200dfdfd409fc31765093be0b928f0",
"EndpointID": "a5a97a57db307af2d5166b75186148eb3df59a254c1bd4c0fc1b9b579009971b",
"Gateway": "",
"IPAddress": "192.168.10.2",
"IPPrefixLen": 24,
```

```
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:c0:a8:0a:02"
            }
        }
    }
```

容器 vm2 的网络信息：

```
[root@node2 ~]# docker exec vm2 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
31: eth0@if32: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
state UP
link/ether 02:42:c0:a8:0a:03 brd ff:ff:ff:ff:ff:ff
inet 192.168.10.3/24 scope global eth0
valid_lft forever preferred_lft forever
  inet6 fe80::42:c0ff:fea8:a03/64 scope link
valid_lft forever preferred_lft forever

[root@node2 ~]# docker inspect vm2
[
...
"NetworkSettings": {
"Bridge": "",
"SandboxID": "19722a2038754e329f5313bd8366c1a765a83d56af07e9915d58e1dad7dbcde8",
"HairpinMode": false,
"LinkLocalIPv6Address": "",
"LinkLocalIPv6PrefixLen": 0,
"Ports": {},
"SandboxKey": "/var/run/docker/netns/19722a203875",
"SecondaryIPAddresses": null,
"SecondaryIPv6Addresses": null,
"EndpointID": "",
"Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "",
"IPPrefixLen": 0,
"IPv6Gateway": "",
"MacAddress": "",
```

```
"Networks": {
"overlay": {
"IPAMConfig": null,
"Links": null,
"Aliases": null,
"NetworkID": "1dc144293a1186332d485924fc951d27ed200dfdfd409fc31765093be0b928f0",
"EndpointID": "7123c69e51ad43fc4d7a5ffc9167ff8c07b691190988b90eaa86128e1b005bb5",
"Gateway": "",
"IPAddress": "192.168.10.3",
"IPPrefixLen": 24,
"IPv6Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"MacAddress": "02:42:c0:a8:0a:03"
        }
      }
    }
  }
]
```

可以看到，容器 vm1 的 IP 为 192.168.10.2/24，vm2 的 IP 为 192.168.10.3/24。从 vm1 访问 vm2：

```
[root@node1 ~]# docker exec vm1 ping -c 3 192.168.10.3
PING 192.168.10.3 (192.168.10.3) 56(84) bytes of data.
64 bytes from 192.168.10.3: icmp_seq=1 ttl=64 time=0.355 ms
64 bytes from 192.168.10.3: icmp_seq=2 ttl=64 time=0.270 ms
64 bytes from 192.168.10.3: icmp_seq=3 ttl=64 time=0.240 ms

--- 192.168.10.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.240/0.288/0.355/0.050 ms
```

15.3 Docker 的 VXLAN 实现

Docker 自身的 Overlay Network 是基于 VXLAN 实现的。VXLAN 协议是一个隧道协议，设计出来是为了解决 VLAN ID（只有 4096 个）不够用的问题。VXLAN ID 有三个字节（24bit），最多可以支持 16 777 216 个隔离的 VXLAN 网络。

VXLAN 将以太网包封装在 UDP 中，并使用物理网络的 IP/MAC 作为 outer-header 进行封装，然后在物理网络上传输，到达目的地后由隧道端点解封并将数据发送给目标机器。这些对报文做封装和解封装的隧道端点被称为 VTEP（Vlan Transport End Point）。对于 Docker 容器，宿主机 Host 即为 VTEP。

15.3.1 VXLAN 帧结构

VXLAN 数据帧的格式如下：

Outer Ethernet header	Outer IP header	Outer UDP header	VXLAN header	Inner Ethernet header	Inner IP header	Inner TCP/UDP header	data

我们可以在 node2 上通过 tcpdump 抓取 vm1 发送到 vm2 的包：

```
No.     Time        Source                Destination           Protocol Length Info
     1 0.000000     192.168.10.2          192.168.10.3          ICMP      148 Echo (ping) request
     2 0.000146     192.168.10.3          192.168.10.2          ICMP      148 Echo (ping) reply
     3 0.999847     192.168.10.2          192.168.10.3          ICMP      148 Echo (ping) request
     4 0.999929     192.168.10.3          192.168.10.2          ICMP      148 Echo (ping) reply
     5 1.999843     192.168.10.2          192.168.10.3          ICMP      148 Echo (ping) request
     6 1.999931     192.168.10.3          192.168.10.2          ICMP      148 Echo (ping) reply
▶ Frame 1: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits)
▶ Ethernet II, Src: 52:54:60:11:02:01 (52:54:60:11:02:01), Dst: 52:54:60:11:02:02 (52:54:60:11:02:02)
▶ Internet Protocol Version 4, Src: 172.17.42.40, Dst: 172.17.42.41
▶ User Datagram Protocol, Src Port: 37390 (37390), Dst Port: 4789 (4789)
▼ Virtual eXtensible Local Area Network
  ▶ Flags: 0x0800, VXLAN Network ID (VNI)
    Group Policy ID: 0
    VXLAN Network Identifier (VNI): 256
    Reserved: 0
▶ Ethernet II, Src: 02:42:c0:a8:0a:02 (02:42:c0:a8:0a:02), Dst: 02:42:c0:a8:0a:03 (02:42:c0:a8:0a:03)
▶ Internet Protocol Version 4, Src: 192.168.10.2, Dst: 192.168.10.3
▶ Internet Control Message Protocol
```

15.3.2 Docker 内部实现

Docker 会为每个 Overlay Network 创建一个独立的 network namespace，名称为 1-$ID 的形式：

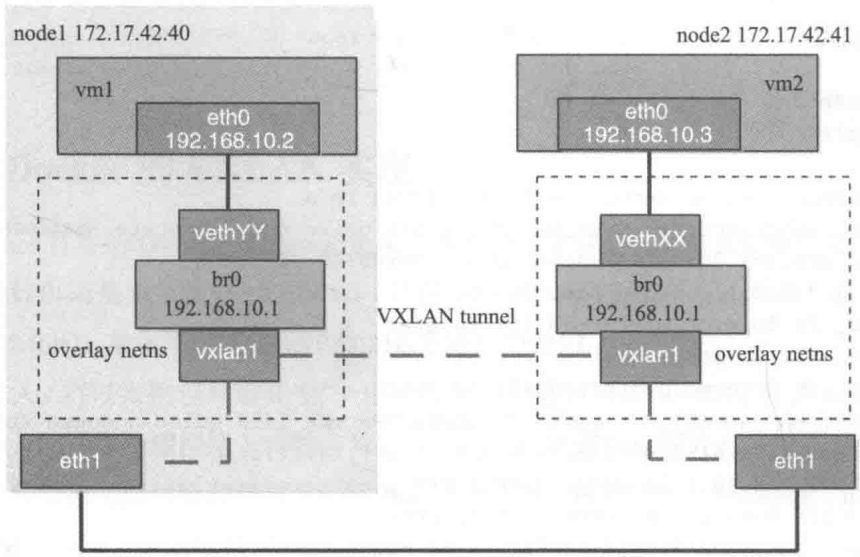
```
[root@node1 ~]# ip netns ls
1-1dc144293a

[root@node1 ~]#ip netns exe 1-1dc144293a 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: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP
link/ether 0a:6a:d9:51:9d:9a brd ff:ff:ff:ff:ff:ff
inet 192.168.10.1/24 scope global br0
valid_lft forever preferred_lft forever
  inet6 fe80::cc37:d6ff:feb8:c7c0/64 scope link
```

```
    valid_lft forever preferred_lft forever
    30: vxlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master br0 state UNKNOWN
      link/ether ea:84:00:ba:dd:52 brd ff:ff:ff:ff:ff:ff
      inet6 fe80::e884:ff:feba:dd52/64 scope link
    valid_lft forever preferred_lft forever
    32: veth2@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
master br0 state UP
    link/ether 0a:6a:d9:51:9d:9a brd ff:ff:ff:ff:ff:ff
      inet6 fe80::86a:d9ff:fe51:9d9a/64 scope link
    valid_lft forever preferred_lft forever

    [root@node1 ~]# ip netns exe 1-1dc144293a brctl show
    bridge name     bridge id               STP enabled     interfaces
    br0             8000.0a6ad9519d9a       no              veth2
                                                            vxlan1
```

其中，veth2 连接容器内部的 Veth 网络设备（eth0）。vxlan1 为 VXLAN 设备，负责 VXLAN 协议的封装和解封。

```
    [root@node1 ~]# ip netns exe 1-1dc144293a ip -d link show vxlan1
    30: vxlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master br0 state UNKNOWN mode DEFAULT
      link/ether ea:84:00:ba:dd:52 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 256 srcport 0 0 dstport 4789 proxy l2miss l3miss ageing 300
```

整体网络结构如下：



15.3.3　Linux VXLAN 设备

前面的章节已经介绍了 Docker 是通过 Linux 内核的 VXLAN 设备实现 VXLAN 协议的封装和解封。在这里，VXLAN 需要解决两个核心的问题。

（1）Overlay Network 的 ARP 广播问题

当容器 vm1（192.168.10.2）想访问容器 vm2（192.168.10.3）时，首先就是向二层网络发送 ARP 广播请求，获取 192.168.10.3 对应的 MAC 地址。

VXLAN 的思路是求助于三层组播。道理很简单，每个 VNI 对应于一个 IPv4 三层的组播地址，然后相关的 VTEP 必须加入到此组播地址中去。当 VTEP 发现某个报文的 DMAC 是广播地址时，目的 IP 被设成此 VNI 对应的三层组播地址。这样所有相关的 VTEP 节点都会收到此报文。但这要求下层物理网络支持 IP 组播。

实际上，我们可以给 VXLAN 设备配置 IP/MAC 映射关系，这样就不需要下层的 IP 组播网络了。Docker 采用的正是这种方式：

```
[root@node1 ~]# ip netns exe 1-1dc144293a ip neigh show dev vxlan1
192.168.10.3 lladdr 02:42:c0:a8:0a:03 PERMANENT
```

（2）确定目标 VTEP

解决了 ARP 问题，当 node1 上的 VXLAN 设备收到 vm1 发送的（MAC2，MAC1）数据帧时，它需要知道 MAC2 对应的目标 VTEP，即 node2。

实际上，VXLAN 内部维护了一张 <MAC,VTEP> 的转发表（FDB）。Docker 在创建容器 vm2 时，就会将容器对应的 MAC 地址和 Host IP 信息，即 <MAC2,node2> 加到转发表中。

```
[root@node1 ~]# ip netns exe 1-1dc144293a bridge fdb show dev vxlan1
ea:84:00:ba:dd:52 permanent
ea:84:00:ba:dd:52 vlan 1 permanent
02:42:c0:a8:0a:03 dst 172.17.42.41 self permanent
```

15.4　容器访问外部网络

前面介绍了容器与容器之间的通信，很多时候，容器还需要与外部网络通信。但是，Overlay 网络是无法直接与外部网络通信的。假设容器 vm1（192.168.10.2）需要访问外部主机，比如 node3（172.17.42.43），我们可以将 vm1 加入 bridge 网络，然后通过 node1 上的 NAT 实现与 node3 的通信。

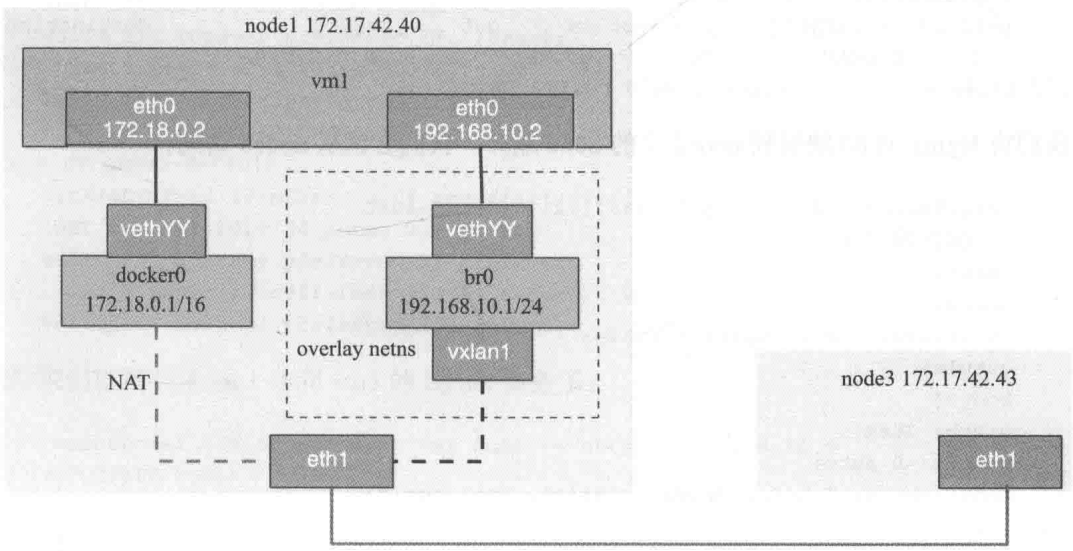
```
[root@node1 ~]# docker network inspect bridge
[
  {
"Name": "bridge",
"Id": "5b7e8f106bfd5d29b05a10583756ba3328a29c82436e139a6b2085a645eb8f28",
"Scope": "local",
"Driver": "bridge",
"IPAM": {
"Driver": "default",
"Options": null,
"Config": [
        {
"Subnet": "172.18.0.0/16",
"Gateway": "172.18.0.1"
        }
      ]
    },
"Containers": {},
"Options": {
"com.docker.network.bridge.default_bridge": "true",
"com.docker.network.bridge.enable_icc": "true",
"com.docker.network.bridge.enable_ip_masquerade": "true",
"com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
"com.docker.network.bridge.name": "docker0",
"com.docker.network.driver.mtu": "1500"
    }
  }
]

[root@node1 ~]# docker network connect bridge vm1
[root@node1 ~]# docker exec vm1 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
31: eth0@if32: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc
noqueue state UP
link/ether 02:42:c0:a8:0a:02 brd ff:ff:ff:ff:ff:ff
inet 192.168.10.2/24 scope global eth0
valid_lft forever preferred_lft forever
  inet6 fe80::42:c0ff:fea8:a02/64 scope link
valid_lft forever preferred_lft forever
33: eth1@if34: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue state UP
```

```
link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
inet 172.18.0.2/16 scope global eth1
valid_lft forever preferred_lft forever
  inet6 fe80::42:acff:fe12:2/64 scope link
valid_lft forever preferred_lft forever
```

可以看到，当我们将容器 vm1 加入网络 bridge 后，容器 vm1 内部多了一个 eth1（172.18.0.2），然后我们访问 node3：

```
[root@node1 ~]# docker exec vm1 ping -c 3 172.17.42.43
PING 172.17.42.43 (172.17.42.43) 56(84) bytes of data.
64 bytes from 172.17.42.43: icmp_seq=1 ttl=63 time=0.246 ms
64 bytes from 172.17.42.43: icmp_seq=2 ttl=63 time=0.177 ms
64 bytes from 172.17.42.43: icmp_seq=3 ttl=63 time=0.221 ms

--- 172.17.42.43 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.177/0.214/0.246/0.033 ms
```

网络结构大致如下：



15.5　外部网络访问容器

如果我们想创建一个运行 Nginx 的容器 vm3，而且希望可以同时从外部网络主机（如 node3）和内部 Overlay 网络的容器（如 vm1）访问，那么该如何实现呢？

我们可以先通过 bridge 网络实现外部网络访问 vm3：

```
[root@node2 ~]# docker run --net=bridge -itd -p 172.17.42.41:8080:80
--name='vm3' nginx:latest
1401d69084c7467c4b9308fb75c3ee674d081f45e5d42f22984b4e2944cb7a65
[root@node2 ~]# docker exec vm3 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
37: eth0@if38: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff
inet 172.18.0.3/16 scope global eth0
valid_lft forever preferred_lft forever
  inet6 fe80::42:acff:fe12:3/64 scope link
valid_lft forever preferred_lft forever
[root@node2 ~]# iptables -t nat -nvL
Chain DOCKER (2 references)
pkts bytes target     prot opt in     out     source               destination
  0     0 DNAT       tcp  --  !docker0 *       0.0.0.0/0
172.17.42.41        tcp dpt:8080 to:172.18.0.3:80
```

我们将 Nginx 的 80 映射到 node2 上的 8080 端口。我们尝试从 node3 访问：

```
[root@node3 ~]# curl -s http://172.17.42.41:8080
<!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>
```

然后我们将 vm3 加入 overlay 的网络：

```
[root@node2 ~]# docker network connect overlay vm3
[root@node2 ~]# docker exec vm3 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
37: eth0@if38: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff
inet 172.18.0.3/16 scope global eth0
valid_lft forever preferred_lft forever
  inet6 fe80::42:acff:fe12:3/64 scope link
valid_lft forever preferred_lft forever
39: eth1@if40: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
state UP group default
link/ether 02:42:c0:a8:0a:04 brd ff:ff:ff:ff:ff:ff
inet 192.168.10.4/24 scope global eth1
valid_lft forever preferred_lft forever
  inet6 fe80::42:c0ff:fea8:a04/64 scope link
valid_lft forever preferred_lft forever
```

然后我们就可以从 vm1 访问 vm3 的 Nginx 服务了：

```
[root@node1 ~]# docker exec vm1 curl -s http://192.168.10.4
<!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>
```

主要参考：https://docs.docker.com/engine/userguide/networking/dockernetworks/。

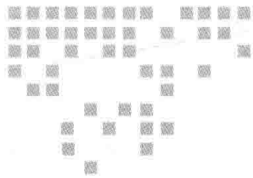
15.6 本章小结

本章主要介绍了 Docker 最新的一些网络特性 Overlay Network，以及 Overlay Network 如何与原有网络的互相通信等问题。Docker 的原有网络方案（bridge）一直广为诟病，Docker 自己对 Overlay Network 寄予厚望，希望通过 Overlay Network 解决 bridge 的一些问题。

第四部分 Part 4

源码探索篇

■ 第 16 章　Docker 源码探索



Chapter 16 第 16 章

Docker 源码探索

但凡比较活跃的开源项目，它的代码质量也一定很高。因为只有代码易于阅读，易于修改，才会有更多的开发者参与到项目中，修改和提交代码。像 Docker 这么火的开源项目，把它使用好是一方面；如果能深入代码，不仅可以让你深入透彻了解 Docker，还可以学习如何用 Golang 编写大型项目的经验。

这一章，我们从 Docker 的源代码入手，讲解如何修改源码，如何调试 Docker 源码。

16.1 Docker 源码目录结构

Docker 的源码可以在 https://github.com/docker/docker 上获取，选择 master 分支。

大家可以在 github 上直接阅读源码，也可以下载到本地，根据个人喜好，选择合适的代码阅读工具。这里推荐在 sourcegraph.com 上阅读，使用 sourcegraph 的优势：

- 简单易用：在线浏览，不需要做任何配置。
- 便捷：鼠标放到代码上，就可以查看文档、函数定义和使用用例。
- sourcegraph 和国产牛人王垠有很深的渊源，支持一下。

Docker 在 sourcegraph 上的阅读地址为 https://sourcegraph.com/github.com/docker/docker。

在 Docker 的根目录下，主要的目录和文件介绍如下：

- Dockerfile：构建 Docker 源码的编译环境。
- Makefile：源码编译指令。
- MAINTAINERS：Docker 主要维护人员，是 Docker 的权威并决定 Docker 的走向，他们的言论和观点需要引起大家足够的重视，尤其是 Solomon Hykes，他是 Docker 的发起人。
- AUTHORS：Docker 代码贡献者，居然还有一名国人贡献者——“尹吉峰”，希望以后会有更多的国人参与这个项目。
- CHANGELOG.md：列出每个版本修改的内容。
- LICENSE：支出 Docker 使用 Apache License 2.0 授权协议。

1. Docker 目录

在 Docker 文件夹下，主要的文件及函数如下：

- docker.go：整个项目的入口 func main()，但它不是最先被调用的。在 Golang 中，init 函数先于 main 函数自动被调度。main 函数：主要执行各个子模块 init 函数中定义的指令（reexec.Init()）和参数解析（flag.Parse()）。
- flags.go：包含 init 函数，定义 Docker 的 server 和 client 共有部分的参数解析，主要是日志级别和证书路径。
- daemon.go docker 的 server 端处理逻辑：包含 init 函数，调用 daemon/config.go 中 InstallFlags()，它定义 docker server 特有参数的解析。

在这里，我们解释一下容易混淆的两个函数 func init() 与 func Init() 的区别：

- init() 是 init 函数，先于 main 函数自动被调度。
- Init() 是模块定义的普通函数，可以被其他函数调用。

在这里，简单补充下 Golang 语言中 init 函数知识，它对找到 Docker 源码的入口函数很有帮助。init 函数用于包（package）的初始化，该函数是 Golang 语言的一个重要特性，它具有如下特征：

- init 函数是用于程序执行前做包的初始化的函数，如初始化包里的变量等。
- 每个包可以拥有多个 init 函数。
- 包的每个源文件也可以拥有多个 init 函数。
- 同一个包中多个 init 函数的执行顺序 go 语言没有明确的定义（说明）。
- 不同包的 init 函数按照包导入的依赖关系决定该初始化函数的执行顺序。

- init 函数不能被其他函数调用，而是在 main 函数执行之前自动被调用。

2. daemon 目录

在 daemon 文件夹下，主要的文件及函数如下：

- config.go：Docker 守护进程启动参数。
- daemon.go：守护进程。
- monitor.go：containerMonitor 监控容器主进程的执行情况。如果执行 restart，containerMonitor 要确保主进程 restart；如果是 stopped，要确保重设或清除容器相关资源，如释放分配的网络资源和 umount 容器 rootfs 文件系统。
- container.go：cleanup() 函数清除 networking 和 mounts；toDisk() 函数 dump 容器的状态到磁盘。

3. image 目录

在 image.go 文件中限定镜像的最多层数是 127。

```
MaxImageDepth = 127
```

4. api 目录

在 server/server.go 参数 api 入口，以 job 的形式运行。针对每一种 job，对它初始化，设置操作对象（如容器名）、参数、环境变量、标准输入、标准输出和错误输出。

获取参数方式：

```
image = r.Form.Get("fromImage")
```

设置 job 的环境变量：

```
job          = eng.Job("commit", r.Form.Get("container"))
job.Setenv("repo", r.Form.Get("repo"))
job.SetenvJson("metaHeaders", metaHeaders)
```

func getImagesSearch() 查询 image 的函数。修改代码，支持选择查询的仓库。

func createRouter：

```
"GET": {
"/_ping":                          ping,
"/events":                         getEvents,
"/info":                           getInfo,
"/version":                        getVersion,
"/images/json":                    getImagesJSON,
```

```
"/images/viz":                    getImagesViz,
"/images/search":                 getImagesSearch,
"/images/get":                    getImagesGet,
"/images/{name:.*}/get":          getImagesGet,
      "/images/{name:.*}/history":       getImagesHistory,
      "/images/{name:.*}/json":          getImagesByName,
      "/containers/ps":                  getContainersJSON,
      "/containers/json":                getContainersJSON,
      "/containers/{name:.*}/export":    getContainersExport,
      "/containers/{name:.*}/changes":   getContainersChanges,
      "/containers/{name:.*}/json":      getContainersByName,
      "/containers/{name:.*}/top":       getContainersTop,
      "/containers/{name:.*}/logs":      getContainersLogs,
      "/containers/{name:.*}/attach/ws": wsContainersAttach,
      "/exec/{id:.*}/json":              getExecByID,
    },
"POST": {
      "/auth":                         postAuth,
      "/commit":                       postCommit,
      "/build":                        postBuild,
      "/images/create":                postImagesCreate,
      "/images/load":                  postImagesLoad,
      "/images/{name:.*}/push":        postImagesPush,
      "/images/{name:.*}/tag":         postImagesTag,
      "/containers/create":            postContainersCreate,
      "/containers/{name:.*}/kill":    postContainersKill,
      "/containers/{name:.*}/pause":   postContainersPause,
      "/containers/{name:.*}/unpause": postContainersUnpause,
      "/containers/{name:.*}/restart": postContainersRestart,
      "/containers/{name:.*}/start":   postContainersStart,
      "/containers/{name:.*}/stop":    postContainersStop,
      "/containers/{name:.*}/wait":    postContainersWait,
      "/containers/{name:.*}/resize":  postContainersResize,
      "/containers/{name:.*}/attach":  postContainersAttach,
      "/containers/{name:.*}/copy":    postContainersCopy,
      "/containers/{name:.*}/exec":    postContainerExecCreate,
      "/exec/{name:.*}/start":         postContainerExecStart,
      "/exec/{name:.*}/resize":        postContainerExecResize,
    },

    "DELETE": {
      "/containers/{name:.*}": deleteContainers,
      "/images/{name:.*}":     deleteImages,
    },

    "OPTIONS": {
```

```
        "": optionsHandler,
    },
```

性能数据的使用：

```
    func AttachProfiler(router *mux.Router) {
    router.HandleFunc("/debug/vars", expvarHandler)
    router.HandleFunc("/debug/pprof/", pprof.Index)
    router.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
    router.HandleFunc("/debug/pprof/profile", pprof.Profile)
    router.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
    router.HandleFunc("/debug/pprof/block", pprof.Handler("block").ServeHTTP)
    router.HandleFunc("/debug/pprof/heap", pprof.Handler("heap").ServeHTTP)
    router.HandleFunc("/debug/pprof/goroutine", pprof.Handler("goroutine").
ServeHTTP)
    router.HandleFunc("/debug/pprof/threadcreate", pprof.Handler("threadcreate").
ServeHTTP)
}
```

5. 常用变量

摘自 daemon/daemon.go，container 起名规则。

```
validContainerNameChars   = `[a-zA-Z0-9][a-zA-Z0-9_.-]`
```

新创建的容器默认使用的 DNS 如下：

```
DefaultDns                = []string{"8.8.8.8", "8.8.4.4"}
```

6. 常用结构体

摘自 daemon/daemon.go。

```
type Daemon struct {
ID            string
repository    string
sysInitPath   string
containers    *contStore
execCommands  *execStore
graph         *graph.Graph
repositories  *graph.TagStore
idIndex       *truncindex.TruncIndex
sysInfo       *sysinfo.SysInfo
volumes       *volumes.Repository
eng           *engine.Engine
config        *Config
containerGraph *graphdb.Database
```

```
    driver          graphdriver.Driver
    execDriver      execdriver.Driver
    trustStore      *trust.TrustStore
}
```

7. 常用的对应关系

摘自 daemon/daemon.go，daemon 关键字对应的函数如下：

```
for name, method := range map[string]engine.Handler{
    "attach":            daemon.ContainerAttach,
    "commit":            daemon.ContainerCommit,
    "container_changes": daemon.ContainerChanges,
    "container_copy":    daemon.ContainerCopy,
    "container_inspect": daemon.ContainerInspect,
    "containers":        daemon.Containers,
    "create":            daemon.ContainerCreate,
    "rm":                daemon.ContainerRm,
    "export":            daemon.ContainerExport,
    "info":              daemon.CmdInfo,
    "kill":              daemon.ContainerKill,
    "logs":              daemon.ContainerLogs,
    "pause":             daemon.ContainerPause,
    "resize":            daemon.ContainerResize,
    "restart":           daemon.ContainerRestart,
    "start":             daemon.ContainerStart,
    "stop":              daemon.ContainerStop,
    "top":               daemon.ContainerTop,
    "unpause":           daemon.ContainerUnpause,
    "wait":              daemon.ContainerWait,
    "image_delete":      daemon.ImageDelete, // FIXME: see above
    "execCreate":        daemon.ContainerExecCreate,
    "execStart":         daemon.ContainerExecStart,
    "execResize":        daemon.ContainerExecResize,
    "execInspect":       daemon.ContainerExecInspect,
  }
```

8. 其他

Docker 守护进程退出，会给每个容器发 SIGTERM(15) 信号，等待退出。

16.2 源码编译 Docker

下载源码：

```
$ git clone https://git@github.com/docker/docker
```

编译：

```
$ cd docker
$ sudo make build
$ sudo make binary
```

编译成功后，在 `./bundles/<version>-dev/binary/` 目录下就会生成 Docker 可执行的二进制文件。

但是不幸的是，按照官方教程，在 `make build` 这一步就会编译失败。

查一下原因，执行 `make build` 对应的是 Makefile 文件中的下面语句：

```
build: bundles
  docker build -t "$(DOCKER_IMAGE)" .

bundles:
  mkdir bundles
```

我们知道，`docker build.` 会在当前目录下查找 Dockerfile 文件，生成镜像。

在 Dockerfile 中，可以看到如下内容：

```
RUN curl -sSL https://golang.org/dl/go1.4.src.tar.gz | tar -v -C /usr/local -xz
...
RUN go get golang.org/x/tools/cmd/cover
```

执行这些语句需要访问 golang.org 网站，而这个网站在国内是访问不了的。Golang 这个编程语言在国内一直温而不火，不知道是不是和这个原因有关。

解决办法参考马全一的“如何在‘特殊’的网络环境下编译 Docker”(https://docker.cn/p/how-to-build-docker-in-mainland)，主要思路是把国外访问不了或访问慢的站点更改为国内源。具体步骤如下。

16.2.1 修改 Dockerfile

修改内容如下：把默认的 apt 源由国外改为国内。具体操作在：

```
FROM    ubuntu:14.04
MAINTAINER Meaglith Ma <genedna@gmail.com> (@genedna)
```

之后添加如下内容：

```
RUN echo "deb http://mirrors.aliyun.com/ubuntu trusty main universe"> /
```

```
etc/apt/sources.list && \
    echo "deb-src http://mirrors.aliyun.com/ubuntu/ trusty main restricted">>
/etc/apt/sources.list && \
    echo "deb http://mirrors.aliyun.com/ubuntu/ trusty-updates main
restricted">> /etc/apt/sources.list && \
    echo "deb-src http://mirrors.aliyun.com/ubuntu/ trusty-updates main
restricted">> /etc/apt/sources.list && \
    echo "deb http://mirrors.aliyun.com/ubuntu/ trusty universe">> /etc/apt/
sources.list && \
    echo "deb-src http://mirrors.aliyun.com/ubuntu/ trusty universe">> /etc/
apt/sources.list && \
    echo "deb http://mirrors.aliyun.com/ubuntu/ trusty-updates universe">> /
etc/apt/sources.list && \
    echo "deb-src http://mirrors.aliyun.com/ubuntu/ trusty-updates universe">>
/etc/apt/sources.list && \
    echo "deb http://mirrors.aliyun.com/ubuntu/ trusty-security main
restricted">> /etc/apt/sources.list && \
    echo "deb-src http://mirrors.aliyun.com/ubuntu/ trusty-security main
restricted">> /etc/apt/sources.list && \
    echo "deb http://mirrors.aliyun.com/ubuntu/ trusty-security universe">> /
etc/apt/sources.list && \
    echo "deb-src http://mirrors.aliyun.com/ubuntu/ trusty-security universe">>
/etc/apt/sources.list
```

修改 lvm2 的获取源，把

```
    RUN git clone -b v2_02_103 https://git.fedorahosted.org/git/lvm2.git /usr/
local/lvm2
```

修改为

```
    RUN      git clone --no-checkout https://coding.net/genedna/lvm2.git /usr/
local/lvm2 && cd /usr/local/lvm2 && git checkout -q v2_02_103
```

修改 Golang 的获取源，把

```
    RUN curl -sSL https://golang.org/dl/go1.3.3.src.tar.gz | tar -v -C /usr/
local -xz
```

修改为

```
    RUN curl -sSL https://github.com/golang/go/archive/go1.3.3.tar.gz |tar -v
-C /usr/local -xz && mv /usr/local/go-go1.3.3 /usr/local/go
```

修改 Golang 的包文件 cover 获取方式，把

```
    RUN go get golang.org/x/tools/cmd/cover
```

修改为

```
RUN     apt-get install -y zip unzip \
&& mkdir -p /go/src/github.com/gpmgo \
&& cd /go/src/github.com/gpmgo \
&& curl -o gopm.zip http://gopm.io/api/v1/download?pkgname=github.com/
gpmgo/gopm\&revision=dev --location \
&& unzip gopm.zip \
&& mv $(ls | grep "gopm-") gopm \
&& rm gopm.zip \
&& cd gopm \
&& go install
RUN     gopm bin -v golang.org/x/tools/cmd/cover
```

dockerfile 经过上述修改，就可以在 `make build` 这一步成功。

16.2.2 其他

另外，在执行 `make binary` 之前，还需要修改 hack/make.sh，把

```
else
  echo >&2 'error: .git directory missing and DOCKER_GITCOMMIT not specified'
  echo >&2 '  Please either build with the .git directory accessible, or
specify the'
  echo >&2 '  exact (--short) commit hash you are building using DOCKER_
GITCOMMIT for'
  echo >&2 '  future accountability in diagnosing build issues.  Thanks!'
    exit 1
fi
```

中的 `exit 1` 这行删除。

编译成功后，二进制文件在 ./bundles/<version>-dev/binary/docker-<version> 下。

替换现有 Docker 二进制文件，方法如下：

```
service docker stop
cp /usr/bin/docker /usr/bin/docker_bak
cp ./bundles/<version>-dev/binary/docker-<version> /usr/bin/docker
chmod +x /usr/bin/docker
service docker start
```

16.2.3 编译源码的好处

当我们可以编译源码时，就可以自由修改 Docker，输出定制的调试信息，为 Docker 添加新功能等，甚至回馈我们的代码给开源社区。

例 1：在 Docker 中启动的容器，默认网卡名为 eth0，且源代码中是写死的，没有提供配置或参数来修改它。而我们这边的使用习惯是用 eth1 作为内网，如果改为 eth0，就会涉及大量脚本的修改。

我们的做法是把 Docker 源码中的 eth0 全部替换为 eth1：

```
find ./ -type f -exec grep eth1 {} \;
find ./ -type f -exec sed -i s/eth0/eth1/g {} \;
find ./ -type f -exec grep eth1 {} \;
```

重新编译源码，替换官方提供的 Docker 二进制文件，在新启动的容器中，可以看到，默认网卡名已经变为 eth1 了。

例 2：为了阅读和分析源代码，我们想知道，执行一个 Docker 操作，如 `docker images`，会有哪些函数被调用，调用顺序是怎样的。下一节，我们会详细讲解如何通过修改源码，让函数被调用时输出下函数名和它的上层调用者。

16.3　输出函数调用关系

当执行 Docker 操作时，通过 runtime.Caller 打印出哪些函数被调用，然后再针对函数做进一步的分析。

1）在 Docker 的源码中添加 [tdebug/getFuncName.go](./code/getFuncNmae.go) 文件，该文件定义了 GetFuncName() 函数。

2）使用 [addDebugFunc.sh](./code/addDebugFunc.sh) 调用 [change.pl](./code/change.pl)，遍历 Docker 源码根目录（除 utils 目录）下所有 xx.go（不包含 xx_test.go），在每个函数中添加 GetFuncName()，显示函数的调用关系。

用法：把 addDebugFunc.sh、change.pl 复制到 Docker 源码根目录下，执行

```
sh addDebugFunc.sh
```

然后按照上一节介绍的方法，重新编译源码。

3）使用方法如下

清空以前的日志：

```
>/var/log/docker
```

执行一个 Docker 操作：

```
docker pull busybox:latest
```

查看结果：

```
grep 'frame 1' /var/log/docker  |grep image
```

结果如下：

```
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/image.
NewImgJSON,file:/go/src/github.com/docker/docker/image/image.go,line:262]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.LoadImage,file:/go/src/github.com/docker/docker/image/image.go,line:42]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.jsonPath,file:/go/src/github.com/docker/docker/image/image.go,line:127]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SetGraph,file:/go/src/github.com/docker/docker/image/image.
go,line:113]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/image.
NewImgJSON,file:/go/src/github.com/docker/docker/image/image.go,line:262]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/image.
NewImgJSON,file:/go/src/github.com/docker/docker/image/image.go,line:262]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/image.
NewImgJSON,file:/go/src/github.com/docker/docker/image/image.go,line:262]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SetGraph,file:/go/src/github.com/docker/docker/image/image.
go,line:113]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.StoreImage,file:/go/src/github.com/docker/docker/image/image.go,line:76]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SaveSize,file:/go/src/github.com/docker/docker/image/image.
go,line:119]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.jsonPath,file:/go/src/github.com/docker/docker/image/image.go,line:127]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SetGraph,file:/go/src/github.com/docker/docker/image/image.
go,line:113]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.StoreImage,file:/go/src/github.com/docker/docker/image/image.go,line:76]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SaveSize,file:/go/src/github.com/docker/docker/image/image.
go,line:119]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.jsonPath,file:/go/src/github.com/docker/docker/image/image.go,line:127]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SetGraph,file:/go/src/github.com/docker/docker/image/image.
go,line:113]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.StoreImage,file:/go/src/github.com/docker/docker/image/image.go,line:76]
```

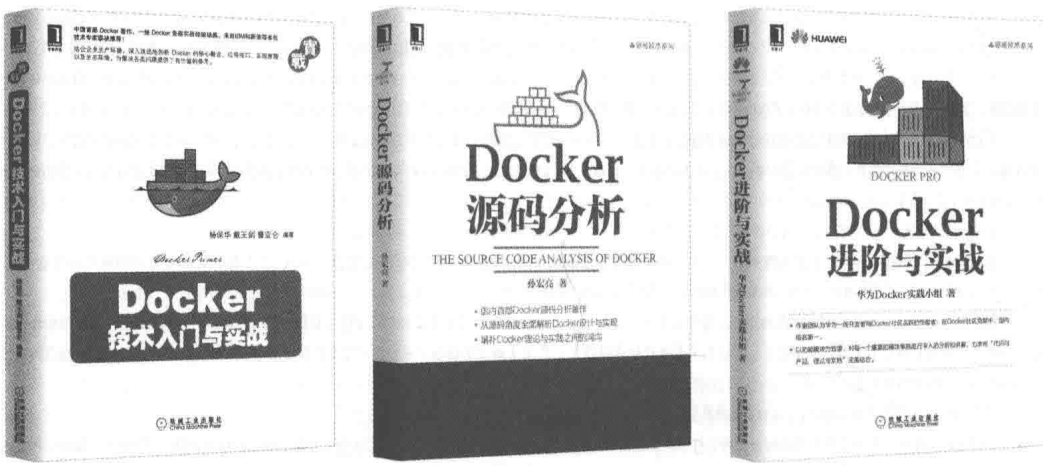
```
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SaveSize,file:/go/src/github.com/docker/docker/image/image.
go,line:119]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.jsonPath,file:/go/src/github.com/docker/docker/image/image.go,line:127]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.LoadImage,file:/go/src/github.com/docker/docker/image/image.go,line:42]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.jsonPath,file:/go/src/github.com/docker/docker/image/image.go,line:127]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
image.(*Image).SetGraph,file:/go/src/github.com/docker/docker/image/image.
go,line:113]

    grep -v  'frame 1' /var/log/docker |grep -v 'getFuncName.go:9
GetFuncName'|grep -v mux|sed  /^$/d |wc -l
    debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
daemon/networkdriver.GetIfaceAddr,file:/go/src/github.com/docker/docker/
daemon/networkd
    river/utils.go,line:88]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
daemon/networkdriver/bridge.setupIPTables,file:/go/src/github.com/docker/
docker/daemon/
    networkdriver/bridge/driver.go,line:191]

    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/pkg/
iptables.Exists,file:/go/src/github.com/docker/docker/pkg/iptables/iptables.
go,line
    :153]
    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
pkg/iptables.Raw,file:/go/src/github.com/docker/docker/pkg/iptables/iptables.
go,line:17
    8]

    [debug] getFuncName.go:13 ----frame 1:[func:github.com/docker/docker/
daemon/networkdriver/portmapper.SetIptablesChain,file:/go/src/github.com/
docker/docker/daemon/networkdriver/portmapper/mapper.go,line:39]
    DefaultNetworkBridge     = "docker0"
```

docker client 也可以使用 -D 来输出 debug 日志。这些信息，对我们梳理 Docker 的源码结构和调用关系非常有帮助。

16.4　本章小结

作为本书的最后一章，我们介绍了 Docker 的源码结构和如何修改与编译 Docker，为用户更深入学习研究 Docker 提供了一种新思路。

推荐阅读



Docker 技术入门与实战

作者：杨保华 戴王剑 曹亚仑 ISBN：978-7-111-48852-1 定价：59.00元

Docker源码分析

作者：孙宏亮 ISBN：978-7-111-51072-7 定价：59.00元

Docker进阶与实战

作者：华为Docker实践小组 ISBN：978-7-111-52339-0 定价：79.00元