



Docker

实践

Docker

IN PRACTICE

伊恩·米尔 (Ian Miell) 著
[美] 艾丹·霍布森·塞耶斯 (Aidan Hobson Sayers) 著
吴佳兴 梁晓勇 黄博文 杨锐 译

目 录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[译者简介](#)

[序](#)

[前言](#)

[致谢](#)

[关于本书](#)

[关于封面插画](#)

[第一部分 Docker基础](#)

[第1章 Docker初探](#)

[1.1 Docker是什么以及为什么用Docker](#)

[1.1.1 Docker是什么](#)

[1.1.2 Docker有什么好处](#)

[1.1.3 关键的概念](#)

[1.2 构建一个Docker应用程序](#)

[1.2.1 创建新的Docker镜像的方式](#)

[1.2.2 编写一个Dockerfile](#)

[1.2.3 构建一个Docker镜像](#)

[1.2.4 运行一个Docker容器](#)

[1.2.5 Docker分层](#)

[1.3 小结](#)

[第2章 理解Docker——深入引擎室](#)

[2.1 Docker的架构](#)

[2.2 Docker守护进程](#)

[技巧1 向世界开放Docker守护进程](#)

[技巧2 以守护进程方式运行容器](#)

[技巧3 将Docker移动到不同分区](#)

[2.3 Docker客户端](#)

[技巧4 使用socat监控Docker API流量](#)

[技巧5 使用端口连接容器](#)

[技巧6 链接容器实现端口隔离](#)

[技巧7 在浏览器中使用Docker](#)

[2.4 Docker注册中心](#)

[技巧8 建立一个本地Docker注册中心](#)

[2.5 Docker Hub](#)

[技巧9 查找并运行一个Docker镜像](#)

[2.6 小结](#)

[第二部分 Docker与开发](#)

[第3章 将Docker用作轻量级虚拟机](#)

[3.1 从虚拟机到容器](#)

[技巧10 将虚拟机转换为容器](#)

[技巧11 类宿主机容器](#)

[技巧12 将一个系统拆成微服务容器](#)

[3.2 管理容器的服务](#)

[技巧13 管理容器内服务的启动](#)

[3.3 保存和还原工作成果](#)

[技巧14 在开发中“保存游戏”的方式](#)

[技巧15 给Docker打标签](#)

[技巧16 在Docker Hub上分享镜像](#)

[技巧17 在构建时指向特定的镜像](#)

[3.4 进程即环境](#)

[技巧18 在开发中“保存游戏”的方式](#)

[3.5 小结](#)

[第4章 Docker日常](#)

[4.1 卷——持久化问题](#)

[技巧19 Docker卷——持久化的问题](#)

[技巧20 通过BitTorrent Sync的分布式卷](#)

[技巧21 保留容器的bash历史](#)

[技巧22 数据容器](#)

[技巧23 使用SSHFS挂载远程卷](#)

[技巧24 通过NFS共享数据](#)

[技巧25 开发工具容器](#)

[4.2 运行容器](#)

[技巧26 在Docker里运行GUI](#)

[技巧27 检查容器](#)

[技巧28 干净地杀掉容器](#)

[技巧29 使用Docker Machine来置备Docker宿主机](#)

[4.3 构建镜像](#)

[技巧30 使用ADD将文件注入到镜像](#)

[技巧31 重新构建时不使用缓存](#)

[技巧32 拆分缓存](#)

[4.4 保持阵型](#)

[技巧33 运行Docker时不加sudo](#)

[技巧34 清理容器](#)

[技巧35 清理卷](#)

[技巧36 解绑容器的同时不停掉它](#)

[技巧37 使用DockerUI来管理Docker守护进程](#)

[技巧38 为Docker镜像生成一个依赖图](#)

[技巧39 直接操作——对容器执行命令](#)

[4.5 小结](#)

[第5章 配置管理——让一切井然有序](#)

[5.1 配置管理和Dockerfile](#)

[技巧40 使用ENTRYPOINT创建可靠的定制工具](#)

[技巧41 在构建中指定版本来避免软件包的漂移](#)

[技巧42 用perl -p -i -e替换文本](#)

[技巧43 镜像的扁平化](#)

[技巧44 用alien管理外来软件包](#)

[技巧45 把镜像逆向工程得到Dockerfile](#)

[5.2 传统配置管理工具与Docker](#)

[技巧46 传统方式：搭配make和Docker](#)

[技巧47 借助Chef Solo构建镜像](#)

[技巧48 从源到镜像的构建](#)

[5.3 小即是美](#)

[技巧49 保持构建镜像更小的Dockerfile技巧](#)

[技巧50 让镜像变得更小的技巧](#)

[技巧51 通过BusyBox和Alpine来精简Docker镜像](#)

[技巧52 Go模型的最小容器](#)

[技巧53 使用inotifywait给容器瘦身](#)

[技巧54 大也可以美](#)

[5.4 小结](#)

第三部分 Docker与DevOps

第6章 持续集成：加快开发流水线

6.1 Docker Hub自动化构建

[技巧55 使用Docker Hub workflow](#)

6.2 更有效的构建

[技巧56 使用eatmydata为I/O密集型构建提速](#)

[技巧57 设置一个软件包缓存用于加快构建速度](#)

[技巧58 在Docker内部运行Selenium测试](#)

6.3 容器化CI过程

[技巧59 包含一个复杂的开发环境](#)

[技巧60 在一个Docker容器里运行Jenkins主服务器](#)

[技巧61 使用Jenkins的Swarm插件扩展CI](#)

6.4 小结

第7章 持续交付：与Docker原则完美契合

7.1 在CD流水线上与其他团队互动

[技巧62 Docker契约——减少摩擦](#)

7.2 推动Docker镜像的部署

[技巧63 手动同步注册中心镜像](#)

[技巧64 通过受限连接交付镜像](#)

[技巧65 以TAR文件方式共享Docker对象](#)

7.3 为不同环境配置镜像

[技巧66 使用etcd通知容器](#)

7.4 升级运行中的容器

[技巧67 使用confd启用零停机时间切换](#)

7.5 小结

第8章 网络模拟：无痛的现实环境测试

8.1 容器通信——超越手工链接

[技巧68 一个简单的Docker Compose集群](#)

[技巧69 一个使用Docker Compose的SQLite服务器](#)

[技巧70 使用Resolvable通过DNS查找容器](#)

8.2 使用Docker来模拟真实世界的网络

[技巧71 使用Comcast模拟有问题的网络](#)

[技巧72 使用Blockade模拟有问题的网络](#)

8.3 Docker和虚拟网络

[技巧73 使用Weave建立一个基底网络](#)

[技巧74 Docker的网络与服务功能](#)

8.4 小结

[第四部分 生产环境中的Docker](#)

[第9章 容器编排：管理多个Docker容器](#)

[9.1 简单的单台宿主机](#)

[技巧75 使用systemd管理宿主机上的容器](#)

[技巧76 使用systemd编排宿主机上的容器](#)

[9.2 多宿主机Docker](#)

[技巧77 使用Helios手动管理多宿主机Docker](#)

[技巧78 基于Swarm的无缝Docker集群](#)

[技巧79 使用Kubernetes集群](#)

[技巧80 在Mesos上构建框架](#)

[技巧81 使用Marathon细粒度管理Mesos](#)

[9.3 服务发现：我们有什么](#)

[技巧82 使用Consul来发现服务](#)

[技巧83 使用Registrator进行自动化服务注册](#)

[9.4 小结](#)

[第10章 Docker与安全](#)

[10.1 Docker访问权限及其意味着什么](#)

[你在乎吗](#)

[10.2 Docker中的安全手段](#)

[技巧84 限制能力](#)

[技巧85 Docker实例上的HTTP认证](#)

[技巧86 保护Docker API](#)

[10.3 来自Docker以外的安全](#)

[技巧87 OpenShift——一个应用程序平台即服务](#)

[技巧88 使用安全选项](#)

[10.4 小结](#)

[第11章 一帆风顺——生产环境中的Docker以及运维上的考量](#)

[11.1 监控](#)

[技巧89 记录容器的日志到宿主机的syslog](#)

[技巧90 把Docker日志发送到宿主机的输出系统](#)

[技巧91 使用cAdvisor监控容器](#)

[11.2 资源控制](#)

[技巧92 限制容器可以运行的内核](#)

[技巧93 给重要的容器更多CPU](#)

[技巧94 限制容器的内存使用](#)

[11.3 Docker的系统管理员用例](#)

[技巧95 使用Docker来运行cron作业](#)

[技巧96 通过“保存游戏”的方法来备份](#)
[11.4 小结](#)

[第12章 Docker生产环境实践——应对各项挑战](#)

[12.1 性能——不能忽略宿主机](#)

[技巧97 从容器访问宿主机资源](#)

[技巧98 Device Mapper存储驱动和默认的容器大小](#)

[12.2 在容器出问题时——调试Docker](#)

[技巧99 使用nsenter调试容器的网络](#)

[技巧100 无须重新配置，使用tcpflow进行实时调试](#)

[技巧101 调试在特定宿主机上出问题的容器](#)

[12.3 小结](#)

[附录A 安装并使用Docker](#)

[附录B Docker配置](#)

[附录C Vagrant](#)

[欢迎来到异步社区！](#)

版权信息

书名： Docker实践

ISBN： 978-7-115-47458-2

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

• 著 [美] 伊恩·米尔 (Ian Miell)

[美] 艾丹·霍布森·塞耶斯 (Aidan Hobson Sayers)

译 吴佳兴 梁晓勇 黄博文 杨 锐

责任编辑 杨海玲

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

- 读者服务热线: (010)81055410

反盗版热线: (010)81055315

版权声明

Original English language edition, entitled *Docker in Practice* by Ian Miell and Aidan Hobson Sayers published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830. Copyright © 2016 by Manning Publications Co.

Simplified Chinese-language edition copyright © 2018 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由**Manning Publications Co.**授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

内容提要

本书由浅入深地讲解了Docker的相关内容，涵盖从开发环境到DevOps流水线，再一路到生产环境的整个落地过程以及相关的实用技巧。书中介绍Docker的核心概念和架构，以及将Docker和开发环境有机、高效地结合起来的方法，包括用作轻量级的虚拟机以及构建和宿主机编排、配置管理、精简镜像等。不仅如此，本书还通过“问题/解决方案/讨论”的形式，将“Docker如何融入DevOps流水线”“如何在生产环境落地”等一系列难题拆解成101个相关的实用技巧，为读者提供解决方案以及一些细节和技巧方面的实践经验。阅读本书，读者将学到的不只是Docker，还包括持续集成、持续交付、构建和镜像管理、容器编排等相关领域的一线生产经验。本书编写时一些案例参考的Docker版本是Docker 1.9。

本书要求读者具备一定的容器管理和运维的基础知识，适合想要将Docker投入实践的相关技术人员阅读，尤其适合具有中高级DevOps和运维背景的读者阅读。

译者简介

吴佳兴，毕业于华东理工大学计算机系，目前是bilibili基础平台研发团队的一员，主要研究方向有CI/CD、监控和运维自动化、基于容器的PaaS平台建设、微服务架构等。2014年年底有幸加入DockOne社区，作为译者，利用闲暇时间为社区贡献一些微薄的力量。个人博客devopstarter.info。欢迎邮件联系（wjsx_colstu@hotmail.com）。

黄博文，ThoughtWorks资深软件工程师/咨询师，担任过开发、测试、运维、技术经理等角色，在国内外多家企业做过技术教练以及技术咨询，拥有丰富的敏捷团队工作经验。目前专注于DevOps技术及云端架构，在搭建持续集成及部署平台、自动化构建基础设施、虚拟化环境、云端运维等方面有着丰富的经验。拥有AWS解决方案架构师以及开发者证书。个人博客为www.huangbowen.net，个人邮箱为huangbowen521@gmail.com。译作有《Effective JavaScript》《Html5和CSS3响应式设计指南》《C#多线程编程实战》《面向对象的思考过程》《基础设施即代码》等。

杨锐，前ThoughtWorks咨询师，DevOps领域持续关注者，任职期间曾任某海外大型项目DevOps工程师，对其持续交付、基础设施即代码、流水线即代码等方面进行了持续推动，对云计算、容器化、持续交付等有一定经验。现供职美团点评。

梁晓勇，毕业于厦门大学，现任某互联网金融公司架构师，DockOne社区编外人员。长期奋战在技术研发第一线，在网络管理、技术开发、架构设计等方面略有心得。热爱互联网技术，积极投身开

源社区，对Docker等容器技术具有浓厚兴趣。欢迎邮件联系
(sunlxy@yahoo.com)。

序

可能是鄙人的拙见，不过Docker真的相当了不起。

在不久以前应用程序还是一些庞大的单体软件，独自运行在一堆钢铁和硅块上。这样的情况持续了很多年，它们拒绝改变，不思进取。对于那些想要快速前行的组织而言，这的确是一个难题，因此虚拟机的崛起也就不足为奇了。应用程序不必再和这些硬件捆绑在一起，这使得一切可以更快更替，也更加灵活。

但是，虚拟机本身也很复杂。它们在其他机器上模拟出整台计算机，而这台虚拟计算机仍然是异常复杂并且需要管理的。再者，因为虚拟机更小而且更容易被创建，所以围绕着它们也就产生了更多需要管理的东西。

我们到底该如何管理所有的复杂性呢？通过配置管理可以办到，当然，这又是另外一个极其复杂的系统。

Docker则采取了一种截然不同的做法。如果用户将软件放到一个容器里，它就会将应用程序本身的复杂度与基础设施相隔离开来，这使得基础设施本身可以变得更加简单，而应用程序也更加易于装配。基于这样的组织效率，与虚拟机相比，技术速度和执行效率都有了巨大的飞跃。容器的启动是毫秒级的，而不是分钟级的。内存是共享的，而不是预先分配的。这使得用户的应用程序能够以更低的成本运行，同样也意味着他可以按照想要的方式来架设应用，而不用再受缓慢的、不太灵活的基础设施的制约。

在我第一次见到Solomon Hykes（即“Docker之父”）谈论Docker并将其与装载集装箱做类比的时候，我便意识到他正在做的是一件了不起的大事。全球航运业建立标准之前的混乱状态可以很好地用来类比容器出现以前管理软件的复杂状态。Solomon的观点非常有说服力，我甚至为此开了一家公司，专门围绕Docker构建工具，这家公司最终被Docker公司收购了，并且最终变成了我们现在所熟知的Docker Compose。

初次见到Ian是在伦敦我们组织的某个Docker聚会（meetup）上。当时，我们坚称：“Docker还没有为生产环境做好准备，请不要在生产环境里使用它！”然而，Ian是何许人也，他无视这个看似明智的建议，一意孤行，非要让它运行在生产环境里。那时，他和Aidan一起为OpenBet公司工作，当我看到他们用我们当时的代码处理过的货币数额时，真的让我感到有点儿懵。

Ian和Aidan发现，他们在使用Docker的过程中收获的价值超越了其测试阶段使用它时伴随而来的不便之处。他们在很早的时候便使用上了这一技术，因此，对怎样更好地运用它也有自己独到的见解。他们在OpenBet内部构建的工具指出了Docker本身缺失的一些东西，而我们之间私下的一些闲聊也实际影响了我们所采用的设计和产品方向。

Docker自Ian和Aidan第一次使用以来，已然有了一些长足的进步，到如今已有成千上万的组织在用它来解决遇到的真实问题，如更快地装配软件、管理惊人的复杂性、提高基础设施的效率、解决“这个在我机器上运行得好好的”问题等。这促使我们在构建、部署和管理软件方式上的巨大转变，并且围绕着它正在形成一套全新的工具及理念。容器化的美好前景的确是让人兴奋的，但是它同我们之前所习惯的方式也迥然不同。

对读者而言，恐怕很难从这本书中看到怎样由此及彼，但是这本书中涵盖了大量如何应用Docker去解决自己当下遇到的种种问题的实用建议。遵循这些建议，用户的组织将能够快速前行。同时或许更重要的是，构建和部署应用的过程将会变得更有意思。

Ben Firshman

Docker公司产品管理总监、Docker Compose联合创始人

前言

2013年9月，浏览黑客志（Hacker News）的时候，我无意中在《连线》上看到一篇介绍一项叫作“Docker”的新技术的文章^[1]。在读到这篇文章时，我便意识到Docker所拥有的革命性的潜力，并为此兴奋不已。

我为之工作了十余年的这家公司一直饱受软件交付速度不够快的困扰。准备环境是一件高成本、费时、手工且十分不优雅的事情。持续集成几乎没有，而且配置开发环境也是一件很考验耐心的事情。由于我的职位含有“DevOps经理”的字样，所以我有特别的动力来解决这些问题！

我通过公司的邮件列表招募了一批积极进取的同事（他们中有一位如今是我的合著者），接着我们的创新团队一起努力，将一个尚处于测试阶段的工具变为商业优势，为公司省去了高昂的虚拟机成本，并且开启了构建和部署软件的新思路。我们甚至构建并开源了一款自动化工具（ShutIt），以满足我们的组织的交付需求。

Docker 为我们提供了一个打包和维护的工具，它解决了很多如果仅靠我们自己根本很难逾越的难题。这是开源技术最棒的地方，它为我们提供了利用业余时间接受挑战的机会，帮助克服技术债务，并且每天都能有收获。我们可以从中学到的不只是Docker，还包括持续集成、持续交付、打包、自动化以及人们该如何应对日新月异的技术革新。

对我们来说，Docker是一个用途异常广泛的工具。只要使用

Linux系统来运行软件，Docker便有用武之地。这也使编写这一主题的图书充满了挑战，毕竟我们的视角是落在广袤的软件本身。为了迎合软件生产方式这样一个本质上的变化，Docker生态系统也在飞速地产出新的解决方案，这也使写书的任务变得更加艰巨。随着时间的推移，我们开始逐渐了解这些问题和解决方案的本质，而在本书里，我们将竭尽所能地传达这些经验。这可以帮助读者找出满足自己的特定技术和业务约束场景的解决方案。

在聚会上面发表演讲时，我们也为Docker在愿意接纳它的组织内部迅速高效地投入使用的方法而感到震撼。本书如实讲述了我们是如何使用Docker的，涵盖了从桌面到DevOps流水线，再一路到生产环境的整个过程。因此，这本书可能会显得不那么正统，但是作为工程师来说，我们相信纯粹性有时候必须让步于实用性，尤其是涉及节约成本方面的话题！本书的所有内容均来源于一线生产的实际经验，我们衷心希望读者可以从我们来之不易的经验中获益。

Ian Miell

[1] <http://www.wired.com/2013/09/docker/>

致谢

如果没有我们最亲近的人的支持、牺牲和耐心，这本书是绝对无法完成的。特别要提到的是Stephen Hazleton，本书的不少内容正是他为了使Docker能够造福我们的客户，同我们一起不懈努力的成果。

几位Docker的贡献者和Docker的员工还非常热情地在不同阶段帮忙审阅了本书的内容，并且提供了许多有价值的反馈，下面几位阅读了本书的初稿：Benoit Benedetti、Burkhard Nestmann、Chad Davis、David Moravec、Ernesto Cárdenas Cangahuala、Fernando Rodrigues、JoséSan Leandro、Kirk Brattkus、Pethuru Raj、Scott Bates、Steven Lembark、Stuart Woodward、Ticean Bennett、Valmiky Arquissandas和Wil Moore III。

最后，本书很大程度上还归功于Manning出版社的编辑团队，他们用自己的方式推动我们改进，使这本书虽不至于完美，但已然做到尽可能好。我们希望他们会为在我们身上付出的努力而感到自豪。

Ian Miell感谢Sarah、Isaac和Rachel，感谢你们忍受我深夜编程，包容一位紧盯着笔记本屏幕的父亲，并且没完没了地念叨“Docker这Docker那，Docker.....”，还要感谢我的父母从小就鼓励我去质疑现状，还给我买Spectrum（腕表）。

Aidan Hobson Sayers感谢Mona的支持和鼓励，感谢我的父母睿智和鼓励的话，还有我的合著者决定命运的那封“有谁试过Docker这种东西？”的电子邮件。

关于本书

Docker可以说是目前增长速度最快的软件项目。它于2013年3月开源，到2016年它已经获得了近30 000个GitHub star以及超过7500次fork。它还接受了大量像Red Hat、IBM、微软、谷歌、思科和Vmware这些厂商的pull request。

Docker在这个关键时刻的出现正是为了迎合许多软件组织的一个迫切需求：以一种开放和灵活的方式来构建软件，然后在不同环境下能够可靠和一致地部署它。用户不需要学习新的编程语言，购买昂贵的硬件，也无须再为了构建、部署和运行可移植的应用程序而在安装或者配置过程上花费过多工夫。

本书将会通过我们在不同场景下用到的一些技术，带读者领略真实世界里的Docker实践案例。我们已经竭力尝试阐明这些技术，尽可能做到无须在阅读前事先具备其他相关技术的知识背景。我们假定读者了解一些基本的开发技术和概念，如开发一些结构化代码的能力，以及对软件开发和部署流程的一些了解。此外，我们认为读者还应了解一些核心的源代码管理理念并且对像TCP/IP、HTTP和端口这样的网络基础知识有一个基本的了解。其他不怎么主流的技术会在我们介绍到的时候予以说明。

我们将从第一部分介绍Docker的基础知识开始，而到了第二部分，我们将把重点放在介绍如何将Docker用到单台机器的开发环境。在第三部分里，我们将介绍Docker在DevOps流水线中的用法，介绍持续集成、持续交付和测试等内容。本书的最后一部分则覆盖了

Docker生产实践的内容，重点关注与编排相关的一些备选方案。

Docker是一个用途广泛、灵活和动态的工具，以至于没有一点儿魄力的话很难追上它快速发展的脚步。我们会尽力通过真实世界的应用和例子让读者更好地理解其中的一些关键概念，目的是希望读者能够有实力、有信心在Docker的生态系统里审慎评估未来采用的工具和技术。我们一直在努力让本书变得更像是一次愉快的旅行，即介绍我们在很多方面见证的Docker是怎样使我们的生活变得更加轻松甚至于更加有趣的。我们正沉浸在Docker以一个别致的方式为我们呈现的覆盖整个软件生命周期的许多有意思的软件技术里，而我们希望本书的读者同样也能分享这样的体验。

路线图

本书包括12章，分为4个部分。

第一部分奠定了本书其余部分的基础，介绍Docker的概念并且教导读者运行一些基本的Docker命令。第2章的一些内容旨在让读者熟悉Docker的客户-服务器架构以及如何调试它，这对在非常规的Docker配置中定位问题是非常有用的。

第二部分关注熟悉Docker以及在自己的机器上如何充分利用Docker。我们将用到一个读者可能比较熟悉的相关概念——虚拟机，作为第3章的基础，提供Docker使用的一些介绍。然后第4章会详细介绍几个我们发现自己每天都在使用的Docker技巧。这一部分的最后一章则探索了更为深入的镜像构建方面的主题。

第三部分从关注Docker在DevOps上下文中的使用开始，从用它完成软件构建和测试的自动化到将它迁移至不同的环境。这一部分还

会花一章的篇幅来总结Docker的虚拟网络，介绍Docker Compose，并且覆盖一些更为高级的网络主题，如网络模拟以及Docker网络插件等。

第四部分会介绍几个针对在生产环境中如何有效地利用Docker的主题。这一部分从第9章开始，在这一章里我们调研了一些最主流的容器编排工具，然后指出了它们往往倾向用在哪些场景。第10章讨论的是安全性的重要话题，阐明了如何锁定在容器里运行的进程，以及如何限制访问对外暴露的Docker守护进程。最后两章则会细讲一些在生产环境中运行Docker的重要实用信息。第11章会展示如何将经典的系统管理知识应用到容器上下文中，从登录到资源限制，而第12章着眼于一些读者可能遇到的问题并且给出对应的调试和解决步骤。

附录里则是一些以不同方式安装、使用和配置Docker的具体细节，包括在虚拟机里以及在Windows上。

代码

本书中用到的所有由作者创建的工具、应用以及Docker镜像的源代码都可以在Manning出版社网站下载，地址是 www.manning.com/books/docker-in-practice，读者也可以在GitHub上的docker-in-practise组织[https://github.com/docker-in-practice/](https://github.com/docker-in-practice)找到这些源代码。Docker Hub上dockerinpractise用户下的镜像（<https://hub.docker.com/u/dockerinpractice/>）均是从其中一个GitHub仓库自动构建生成的。在这里，我们已经意识到读者可能会有兴趣对技术背后的一些源代码做进一步的研究，因此在技术讨论里也嵌入了相关仓库的链接。

为了方便读者跟进，本书中列出的大量代码均以终端会话的形式，与相应的命令输出一起展示。这些会话里有几件事情要注意一下。

很长的终端命令可能会使用shell的续行字符（\`\`）将一条命令分割成多行。虽然读者直接把它贴到自己的shell下面运行也能工作，但是读者也可以略去这个续行字符，在一行里键入整条命令。

当输出的部分对于读者来说没有提供额外有用的信息时，它可能会被省略并在相应的位置用省略号（`[...]`）替代。

作者在线

购买本书的读者还可免费得到由Manning出版社运营的一个私有网站论坛的访问权限，在这里读者可以对本书作出评论，询问技术问题，并获得来自主要作者以及其他读者的帮助。要访问和订阅该论坛的话，请用Web浏览器打开www.manning.com/books/docker-in-practice。该页面将会提供一些信息，包括读者注册后该如何登录到论坛，能获得怎样的帮助，以及论坛里的一些行为准则。

Manning对读者的承诺是会为其提供一个读者之间以及读者和作者之间进行有价值讨论的场所。但并不承诺作者的参与程度，作者对论坛的贡献目前仍然还是停留在志愿性质（并且是无报酬的）。我们建议读者试着问一些有挑战的问题，以激发作者回答问题的兴趣！作者在线论坛和之前讨论的档案只要书还在发行就一直可以在Manning出版社网站上访问。

关于封面插画

本书的封面图片的标题是“一个来自克罗地亚赛尔切的男人”（Man from selc, croatia）。这张图片取自19世纪中期Nikola Arsenovic的一本克罗地亚传统服饰图集的复刻版，由克罗地亚斯普利特人种学博物馆在2003年出版。该图由人种学博物馆的一位热心的图书管理员提供。该博物馆位于中世纪时罗马帝国的核心城镇，从约公元304年起，帝国国王戴克里先退位后居住的皇宫的遗迹就在这里。这本书中涵盖了来自克罗地亚各个地区的华丽的彩色图片，并介绍了他们的服饰和日常生活。

在这过去的200年里，服饰和生活方式都发生了巨大的变化，各地当时的特色也已随时间消逝。如今，来自不同大陆的人都已经变得难以区分，更不用说相隔仅数千米的村镇居民了。或许，文化多样性也是我们为获得丰富多彩的个人生活而付出的代价——现在生活无疑是更多姿多彩的、快节奏的高科技生活。

Manning出版社用两个世纪前各地独具特色的生活方式来赞美计算机行业的诞生和发展，并用古老的书籍和图册中的图片让我们领略那个时代的风土人情。

第一部分 Docker基础

本书的第一部分由第1章和第2章构成，将带领读者开始使用Docker，并讲解其基础知识。第1章阐述Docker的起源及其核心概念，如镜像、容器和分层。在第1章的最后，读者将动手使用Dockerfile创建自己的第一个镜像。第2章介绍一些有用的技巧，让读者深入理解Docker的架构。我们通过依次讲解每个主要组件，阐述Docker守护进程与其客户端、Docker注册中心和Docker Hub之间的关系。在第一部分结束时，读者将对Docker的核心概念有所了解，并能够演示一些有用的技巧，为理解本书的后续内容打下坚实的基础。

第1章 Docker初探

本章主要内容

- Docker是什么
- Docker的使用以及它如何能节省时间和金钱
- 容器与镜像之间的区别
- Docker的分层特性
- 使用Docker构建并运行一个to-do应用程序

Docker是一个允许用户“在任何地方构建、分发及运行任何应用”的平台。它在极短的时间内发展壮大，目前已经被视为解决软件中最昂贵的方面之一——部署的一个标准方法。

在Docker出现之前，开发流水线通常由用于管理软件活动的不同技术组合而成，如虚拟机、配置管理工具、不同的包管理系统以及各类依赖库复杂的网站。所有这些工具需要由专业的工程师管理和维护，并且多数工具都具有自己独特的配置方式。

Docker改变了这一切，允许不同的工程师参与到这个过程中，有效地使用同一门语言，这让协作变得轻而易举。所有东西通过一个共同的流水线转变成可以在任何目标平台上使用的单一的产出——无须继续维护一堆让人眼花缭乱的工具配置，如图1-1所示。

与此同时，只要现存的软件技术栈依然有效，用户就无须抛弃它——用户可以将其原样打包到一个Docker容器内供其他人使用。由此获得的额外好处是，用户能看到这些容器是如何构建的，因此如果需

要深入其细节，完全没问题。

本书针对的是具有一定Docker知识的中级开发人员。如果读者对本书的基础部分较熟悉，可随意跳到后续章节。本书的目标是揭示Docker所带来的现实世界的挑战，并展示其解决之道。不过，首先我们将提供一个Docker自身的快速回顾。如果读者想了解更全面的Docker基础，请查阅Jeff Nickoloff编写的《Docker in Action》一书（Manning Publications, 2016）。

第2章将更深入地介绍Docker的架构，并通过一些技巧来演示其威力。在本章中，读者将了解到Docker是什么、为什么它很重要，并开始使用它。

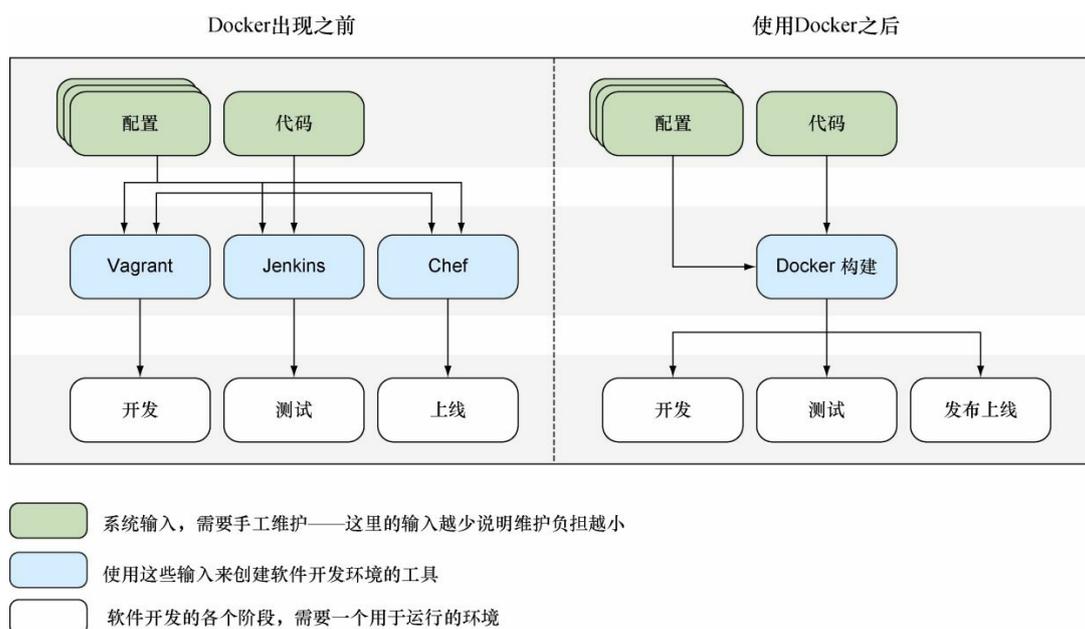


图1-1 Docker如何消除了工具维护的负担

1.1 Docker是什么以及为什么用Docker

在动手实践之前，我们将对Docker稍做讨论，以便读者了解它的背景、“Docker”名字的来历以及为什么使用它！

1.1.1 Docker是什么

要理解Docker是什么，从一个比喻开始会比技术性解释来得简单，而且这个Docker的比喻非常有说服力。Docker原本是指在船只停靠港口之后将商品移进或移出的工人。箱子和物品的大小和形状各异，而有经验的码头工人能以合算的方式手工将商品装入船只，因而他们倍受青睐（见图1-2）。雇人搬东西并不便宜，但除此之外别无选择。

对在软件行业工作的人来说，这听起来应该很熟悉。大量时间和精力被花在将奇形怪状的软件放置到装满了其他奇形怪状软件、大小各异的船只上，以便将其卖给其他地方的用户或商业机构。

图1-3展示了使用Docker概念时如何能节省时间和金钱。

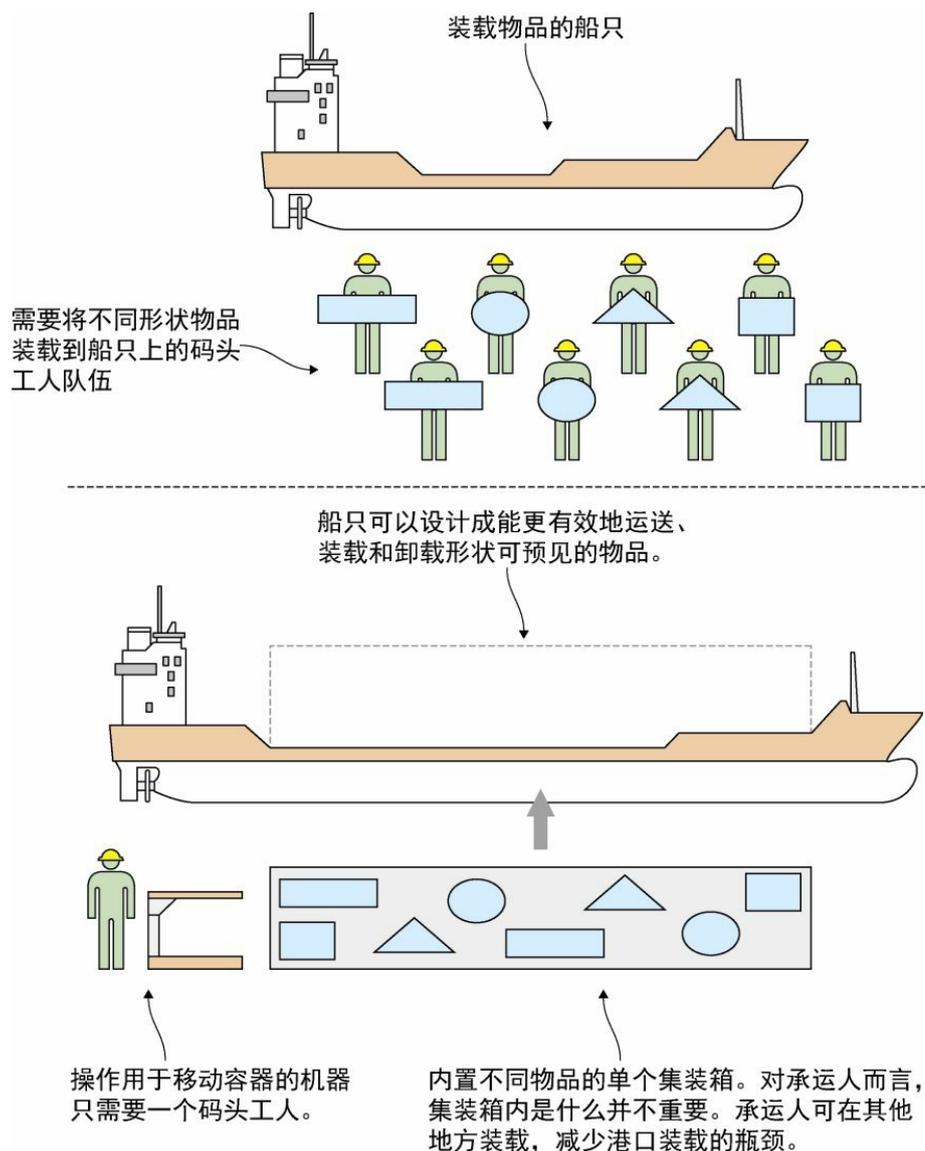


图1-2 标准化集装箱前后的航运对比

在Docker出现之前，部署软件到不同环境所需的工作量巨大。即使不是采用手工运行脚本的方式在不同机器上进行软件配备（还是有很多人这么做），用户也不得不全力应付那些配置管理工具，它们掌管着渴求资源且快速变化的环境的状态。即便将这些工作封装到虚拟机中，还是需要花费大量时间来部署这些虚拟机、等待它们启动并管理它们所产生的额外的资源开销。

使用Docker，配置工作从资源管理中分离了出来，而部署工作则

是微不足道的：运行docker run，环境的镜像会被拉取下来并准备运行，所消耗的资源更少并且是内含的，因此不会干扰其他环境。

读者无须担心容器是将被分发到Red Hat机器、Ubuntu机器还是CentOS虚拟机镜像中，只要上面有Docker，就没有问题。

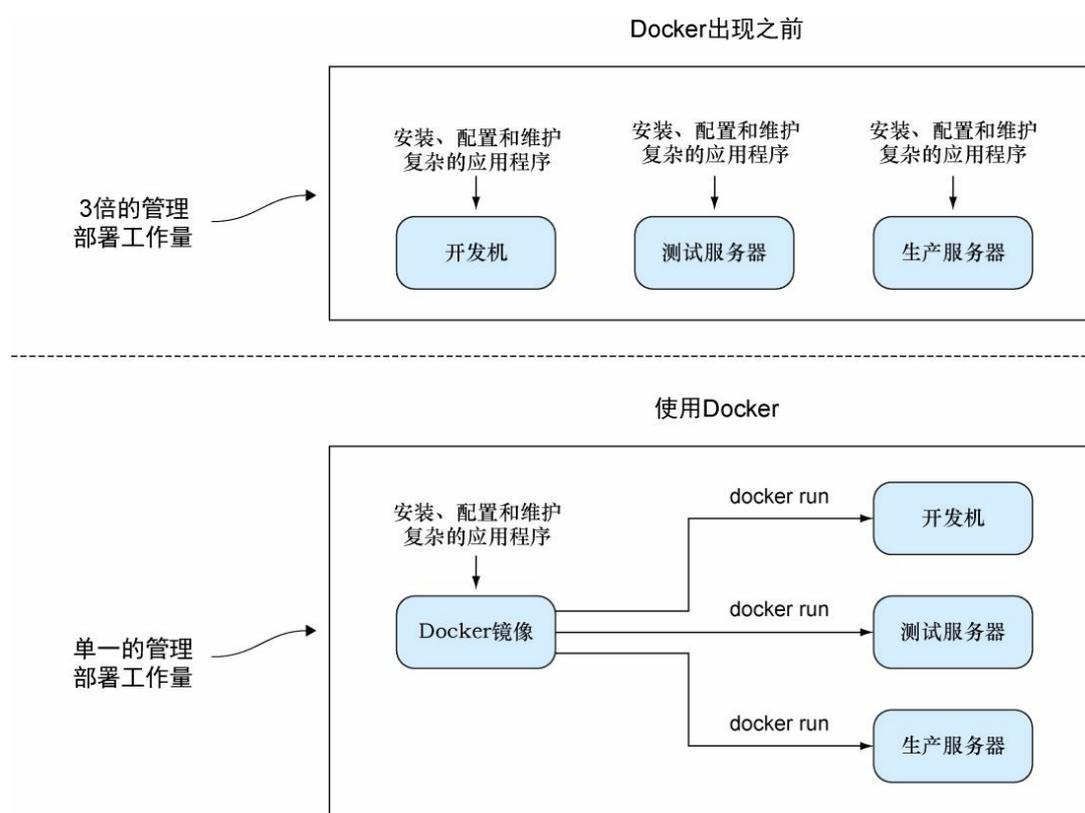


图1-3 使用Docker前后软件交付的对比

1.1.2 Docker有什么好处

几个重要的实际问题出现了：为什么要使用Docker，Docker用在什么地方？针对“为什么”的简要答案是：只需要一点点付出，Docker就能快速为企业节省大量金钱。部分方法（肯定不是所有的）将在随后的几节中讨论。我们已经在实际工作环境中切身体会到所有这些益处。

1. 替代虚拟机（VM）

Docker可以在很多情况下替代虚拟机。如果用户只关心应用程序，而不是操作系统，可以用Docker替代虚拟机，并将操作系统交给其他人去考虑。Docker不仅启动速度比虚拟机快，迁移时也更为轻量，同时得益于它的分层文件系统，与其他人分享变更时更简单、更快捷。而且，它牢牢地扎根在命令行中，非常适合脚本化。

2. 软件原型

如果想快速体验软件，同时避免干扰目前的设置或配备一个虚拟机的麻烦，Docker可以在几毫秒内提供一个沙箱环境。在亲身体验之前，很难感受到这种解放的效果。

3. 打包软件

因为对Linux用户而言，Docker镜像实际上没有依赖，所以非常适合用于打包软件。用户可以构建镜像，并确保它可以运行在任何现代Linux机器上——就像Java一样，但不需要JVM。

4. 让微服务架构成为可能

Docker有助于将一个复杂系统分解成一系列可组合的部分，这让用户可以用更离散的方式来思考其服务。用户可以在不影响全局的前提下重组软件使其各部分更易于管理和可插拔。

5. 网络建模

由于可以在一台机器上启动数百个（甚至数千个）隔离的容器，因此对网络进行建模轻而易举。这对于现实世界场景的测试非常有用，而且所费无几。

6. 离线时启用全栈生产力

因为可以将系统的所有部分捆绑在Docker容器中，所以用户可以将其编排运行在笔记本电脑中移动办公，即便在离线时也没问题。

7. 降低调试支出

不同团队之间关于软件交付的复杂谈判在业内司空见惯。我们亲身经历过不计其数的这类讨论：失效的库、有问题的依赖、更新被错误实施或是执行顺序有误，甚至可能根本没执行以及无法重现的错误等。估计读者也遇到过。Docker让用户可以清晰地说明（即便是脚本的形式）在一个属性已知的系统上调试问题的步骤，错误和环境重现变得更简单，而且通常与所提供的宿主机环境是分离的。

8. 文档化软件依赖及接触点

通过使用结构化方式构建镜像，为迁移到不同环境做好准备，Docker强制用户从一个基本出发点开始明确地记录软件依赖。即使用户不打算在所有地方都使用Docker，这种对文档记录的需要也有助于在其他地方安装软件。

9. 启用持续交付

持续交付（continuous delivery, CD）是一个基于流水线的软件交付范型，该流水线通过一个自动化（或半自动化）流程在每次变动时重新构建系统然后交付到生产环境中。

因为用户可以更准确地控制构建环境的状态，Docker 构建比传统软件构建方法更具有可重现性和可复制性。使持续交付的实现变得更容易。通过实现一个以Docker为中心的可重现的构建过程，标准的持续交付技术，如蓝/绿部署（blue/green deployment，在生产环境中维护“生产”和“最新”部署）和凤凰部署（phoenix deployment，每次发布时都重新构建整个系统）变得很简单。

现在，读者对Docker如何能够提供帮助有了一定了解。在进入一个真实示例之前，让我们来了解一下几个核心概念。

1.1.3 关键的概念

在本节中，我们将介绍一些关键的Docker概念，如图1-4所示。

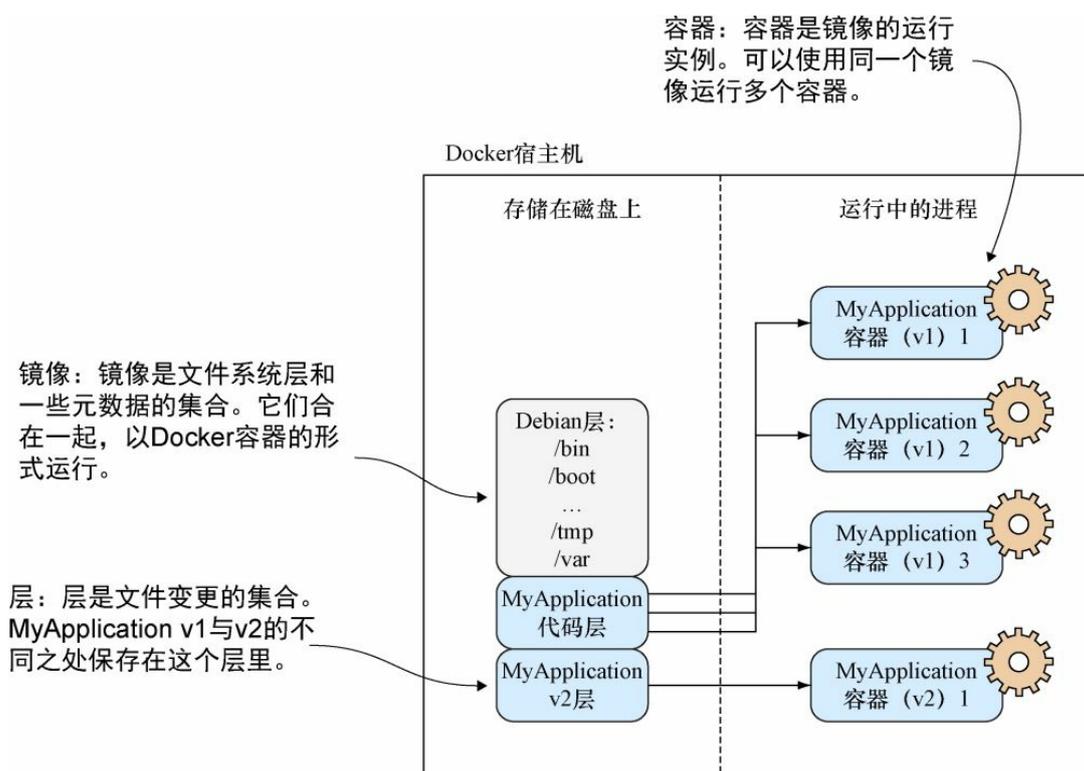


图1-4 Docker的核心概念

在开始运行Docker命令之前，将镜像、容器及层的概念牢记于心是极其有用的。简而言之，容器运行着由镜像定义的系统。这些镜像由一个或多个层（或差异集）加上一些Docker的元数据组成。

让我们来看一些核心的Docker命令。我们将把镜像转变成容器，修改它们，并添加层到我们即将提交的新镜像中。如果这一切听上去有点儿混乱，不用太担心。在本章结束时，一切都将更加清晰。

1. 关键的Docker命令

Docker的中心功能是构建、分发及在任何具有Docker的地方运行软件。

对终端用户而言，Docker是一个用于运行的命令行程序。就像git（或任何源代码控制工具）一样，这个程序具有用于执行不同操作的子命令。

表1-1中列出了将在宿主机上使用的主要的Docker子命令。

表1-1 Docker子命令

命 令	目 的
<code>docker build</code>	构建一个Docker镜像
<code>docker run</code>	以容器形式运行一个Docker镜像
<code>docker commit</code>	将一个Docker容器作为一个镜像提交

docker tag	给一个Docker镜像打标签
------------	----------------

2. 镜像与容器

如果读者不熟悉Docker，这可能是第一次听说本文所说的“容器”和“镜像”这两个词。它们很可能是Docker中最重要的概念，因此有必要花点儿时间明确其差异。

在图1-5中，读者将看到这些概念的展示，里面有从一个基础镜像启动的3个容器。

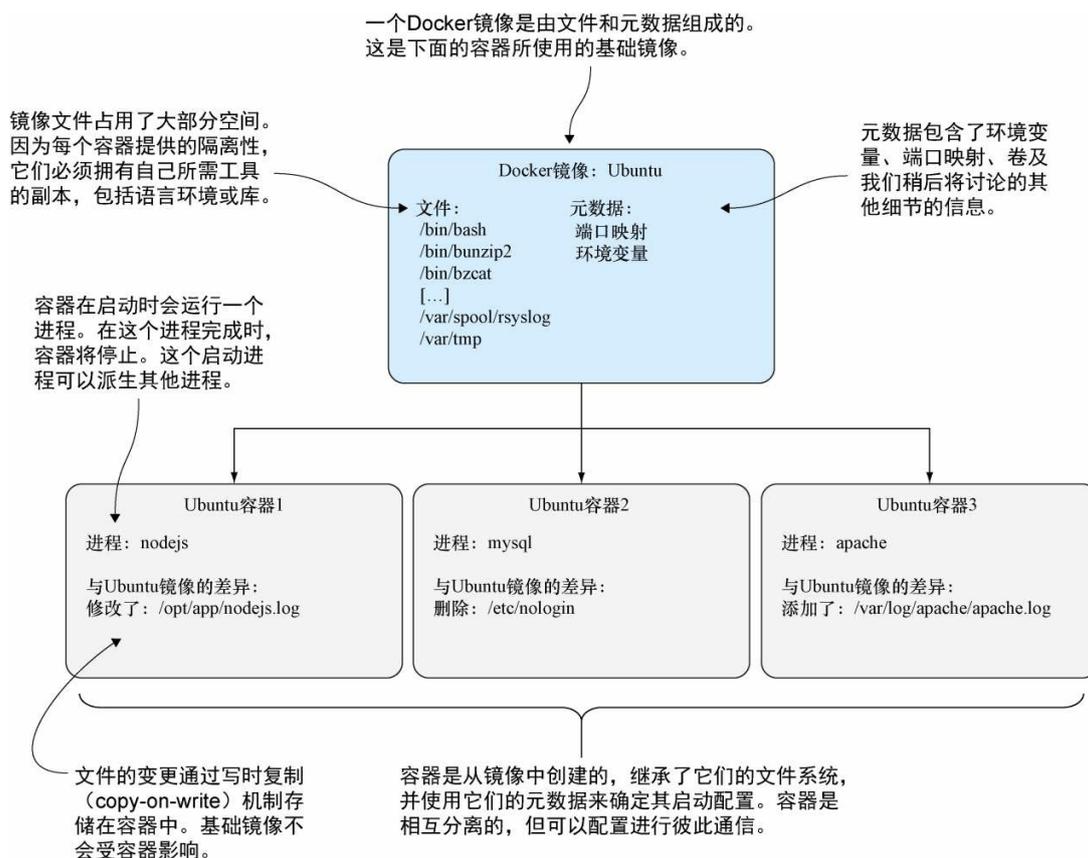


图1-5 Docker镜像与容器

看待镜像和容器的一种方式是将它们类比成程序与进程。一个进

程可以视为一个被执行的应用程序，同样，一个Docker容器可以视为一个运行中的Docker镜像。

如果读者熟悉面向对象原理，看待镜像和容器的另一种方法是将镜像看作类而将容器看作对象。对象是类的具体实例，同样，容器是镜像的实例。用户可以从单个镜像创建多个容器，就像对象一样，它们之间全都是相互隔离的。不论用户在对象内修改了什么，都不会影响类的定义——它们从根本上就是不同的东西。

1.2 构建一个Docker应用程序

现在，我们要动手使用Docker来构建一个简单的“to-do”应用程序（todoapp）镜像了。在这个过程中，读者会看到一些关键的Docker功能，如Dockerfile、镜像复用、端口暴露及构建自动化。这是接下来10分钟读者将学到的东西：

- 如何使用Dockerfile来创建Docker镜像；
- 如何为Docker镜像打标签以便引用；
- 如何运行新建的Docker镜像。

to-do应用是协助用户跟踪待完成事项的一个应用程序。我们所构建的应用将存储并显示可被标记为已完成的信息的简短字符串，它以一个简单的网页界面呈现。

图1-6展示了如此操作将得到的结果。

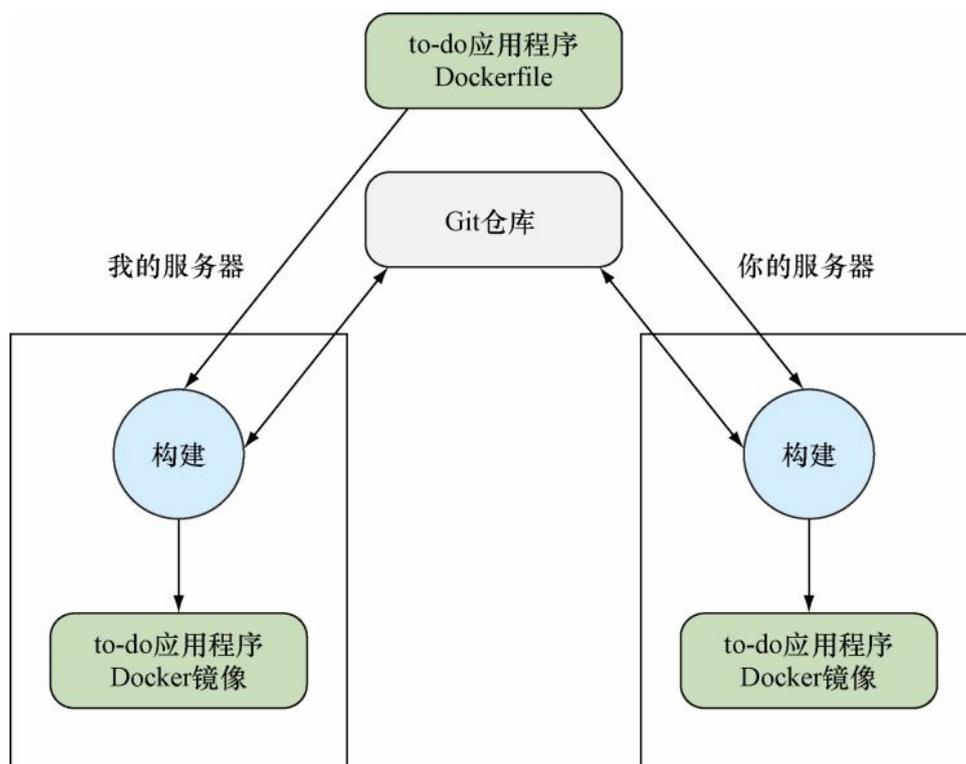


图1-6 构建一个Docker应用程序

应用程序的细节不是重点。我们将演示从我们所提供的一个简短的Dockerfile开始，读者可以放心地在自己的宿主机上使用与我们相同的方法构建、运行、停止和启动一个应用程序，而无须考虑应用程序的安装或依赖。这正是 Docker 赋予我们的关键部分——可靠地重现并简便地管理和分享开发环境。这意味着用户无须再遵循并迷失在那些复杂的或含糊的安装说明中。

to-do应用程序 这个to-do应用程序将贯穿本书，多次使用，它非常适合用于实践和演示，因此值得读者熟悉一下。

1.2.1 创建新的Docker镜像的方式

创建Docker镜像有4种标准的方式。表1-2逐一列出了这些方法。

表1-2 创建Docker镜像的方式

方 法	描 述	详见技巧
Docker命令/“手工”	使用docker run启动一个容器，并在命令行输入命令来创建镜像。使用docker commit来创建一个新镜像	详见技巧14
Dockerfile	从一个已知基础镜像开始构建，并指定一组有限的简单命令来构建	稍后讨论
Dockerfile及配置管理（configuration management, CM）工具	与Dockerfile相同，不过将构建的控制权交给了更为复杂的CM工具	详见技巧47
从头创建镜像并导入一组文件	从一个空白镜像开始，导入一个含有所需文件的TAR文件	详见技巧10

如果用户所做的是概念验证以确认安装过程是否正常，那么第一种“手工”方式是没问题的。在这个过程中，用户应对所采取的步骤做记录，以便在需要时回到同一点上。

到某个时间点，用户会想要定义创建镜像的步骤。这就是第二种方式（也就是我们这里所用的方式）。

对于更复杂的构建，用户需要使用第三种方式，特别是在Dockerfile功能还不足以满足镜像要求的时候。

最后一种方式从一个空镜像开始，通过叠加一组运行镜像所需要

的文件进行构建。如果用户想导入一组在其他地方创建好的自包含的文件，这将非常有用，不过这在主流应用中非常罕见。

现在，我们来看一下Dockerfile方法，其他方法将在本书后面再说明。

1.2.2 编写一个Dockerfile

Dockerfile是一个包含一系列命令的文本文件。本示例中我们将使用的Dockerfile如下：

```
FROM node (1)
MAINTAINER ian.miell@gmail.com (2)
RUN git clone -q https://github.com/docker-in-practice/todo.git (3)
WORKDIR todo (4)
RUN npm install > /dev/null (5)
EXPOSE 8000 (6)
CMD ["npm","start"] (7)
```

- (1) ❶ 定义基础镜像
- (2) ❷ 声明维护人员
- (3) ❸ 克隆todoapp代码
- (4) ❹ 移动到新的克隆目录
- (5) ❺ 运行node包管理器的安装命令（npm）
- (6) ❻ 指定从所构建的镜像启动的容器需要监听这个端口
- (7) ❼ 指定在启动时需要运行的命令

Dockerfile的开始部分是使用FROM命令定义基础镜像❶。本示例使用了一个Node.js镜像以便访问Node.js程序。官方的Node.js镜像名

为node。

接下来，使用**MAINTAINER**命令声明维护人员**②**。在这里，我们使用的是其中一个人的电子邮件地址，读者也可以替换成自己的，因为现在它是你的**Dockerfile**了。这一行不是创建可工作的**Docker**镜像所必需的，不过将其包含进来是一个很好的做法。到这个时候，构建已经继承了**node**容器的状态，读者可以在它上面做操作了。

接下来，使用**RUN**命令克隆**todoapp**代码**③**。这里使用指定的命令获取应用程序的代码：在容器内运行**git**。在这个示例中，**Git**是安装在基础**node**镜像里的，不过读者不能对这类事情做假定。

现在移动到使用**WORKDIR**命令新克隆的目录中**④**。这不仅会改变构建环境中的目录，最后一条**WORKDIR**命令还决定了从所构建镜像启动容器时用户所处的默认目录。

接下来，运行**node**包管理器的安装命令（**npm**）**⑤**。这将为应用程序设置依赖。我们对输出的信息不感兴趣，所以将其重定向到**/dev/null**上。

由于应用程序使用了**8000**端口，使用**EXPOSE**命令告诉**Docker**从所构建镜像启动的容器应该监听这个端口**⑥**。

最后，使用**CMD**命令告诉**Docker**在容器启动时将运行哪条命令**⑦**。

这个简单的示例演示了**Docker**及**Dockerfile**的几个核心功能。**Dockerfile**是一组严格按顺序运行的有限的命令集的简单序列。它们

影响了最终镜像的文件和元数据。这里的RUN命令通过签出并安装应用程序影响了文件系统，而EXPOSE、CMD和WORKDIR命令影响了镜像的元数据。

1.2.3 构建一个Docker镜像

读者已经定义了自己的Dockerfile的构建步骤。现在可以键入图1-7所示的命令从而构建Docker镜像了。

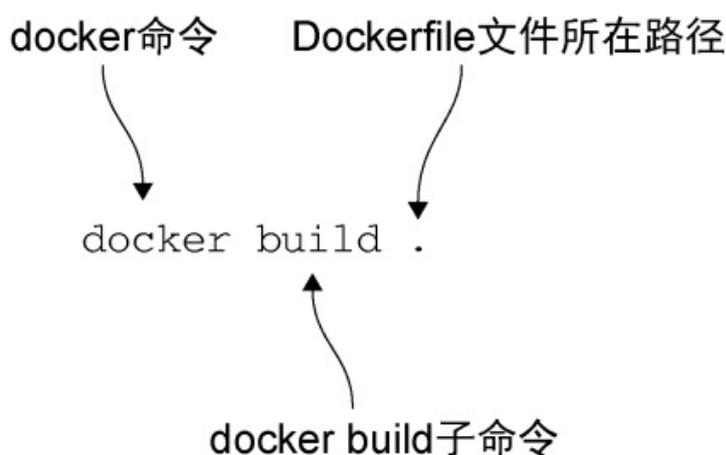


图1-7 Docker build命令

读者将看到类似下面这样的输出：

```
Sending build context to Docker daemon 178.7 kB (1)
Sending build context to Docker daemon
Step 0 : FROM node (2)
---> fc81e574af43 (3)
Step 1 : MAINTAINER ian.miell@gmail.com
---> Running in 21af1aad6950
---> 8f32669fe435
Removing intermediate container 21af1aad6950 (4)
Step 2 : RUN git clone https://github.com/ianmiell/todo.git
---> Running in 0a030ee746ea
Cloning into 'todo'...
---> 783c68b2e3fc
Removing intermediate container 0a030ee746ea
Step 3 : WORKDIR todo
---> Running in 2e59f5df7152
---> 8686b344b124
Removing intermediate container 2e59f5df7152
```

```
Step 4 : RUN npm install
---> Running in bdf07a308fca
npm info it worked if it ends with ok (5)
[...]
npm info ok
---> 6cf8f3633306
Removing intermediate container bdf07a308fca
Step 5 : RUN chmod -R 777 /todo
---> Running in c03f27789768
---> 2c0ededd3a5e
Removing intermediate container c03f27789768
Step 6 : EXPOSE 8000
---> Running in 46685ea97b8f
---> f1c29fecae036
Removing intermediate container 46685ea97b8f
Step 7 : CMD npm start
---> Running in 7b4c1a9ed6af
---> 66c76cea05bb
Removing intermediate container 7b4c1a9ed6af
Successfully built 66c76cea05bb (6)
```

- (1) Docker会上传docker build指定目录下的文件和目录
- (2) 每个构建步骤从 0 开始按顺序编号，并与命令一起输出
- (3) 每个命令会导致一个新镜像被创建出来，其镜像ID在此输出
- (4) 为节省空间，在继续前每个中间容器会被移除
- (5) 构建的调试信息在此输出（限于篇幅，本代码清单做了删减）
- (6) 此次构建的最终镜像ID，可用于打标签

现在，拥有了一个具有镜像ID（前面示例中的“66c76cea05bb”，不过读者的ID会不一样）的Docker镜像。总是引用这个ID会很麻烦，可以为其打标签以方便引用。

输入图1-8所示的命令，将66c76cea05bb替换成读者生成的镜像ID。



图1-8 Docker tag命令

现在就能从一个Dockerfile构建自己的Docker镜像副本，并重现别人定义的环境了！

1.2.4 运行一个Docker容器

读者已经构建出Docker镜像并为其打上了标签。现在可以以容器的形式来运行它了：

```
docker run -p 8000:8000 --name example1 todoapp (1)
npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm info prestart todomvc-swarm@0.0.1

> todomvc-swarm@0.0.1 prestart /todo (2)
> make all
npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
npm WARN package.json todomvc-swarm@0.0.1 license should be a
➤ valid SPDX license expression
npm info preinstall todomvc-swarm@0.0.1
npm info package.json statics@0.1.0 license should be a valid
➤ SPDX license expression
```

```

npm info package.json react-tools@0.11.2 No license field.
npm info package.json react@0.11.2 No license field.
npm info package.json node-jsx@0.11.0 license should be a valid
  ➤ SPDX license expression
npm info package.json ws@0.4.32 No license field.
npm info build /todo
npm info linkStuff todomvc-swarm@0.0.1
npm info install todomvc-swarm@0.0.1
npm info postinstall todomvc-swarm@0.0.1
npm info prepublish todomvc-swarm@0.0.1
npm info ok
if [ ! -e dist/ ]; then mkdir dist; fi
cp node_modules/react/dist/react.min.js dist/react.min.js

```

```

LocalTodoApp.js:9:    // TODO: default english version
LocalTodoApp.js:84:    fwdList =
  ➤ this.host.get('/TodoList#+listId); // TODO fn+id sig
TodoApp.js:117:    // TODO scroll into view
TodoApp.js:176:    if (i>=list.length()) { i=list.length()-1;}
  ➤ // TODO .length
local.html:30:    <!-- TODO 2-split, 3-split -->
model/TodoList.js:29:
  ➤ // TODO one op - repeated spec? long spec?
view/Footer.jsx:61:    // TODO: show the entry's metadata
view/Footer.jsx:80:    todoList.addObject(new TodoItem());
  ➤ // TODO create default
view/Header.jsx:25:
  ➤ // TODO list some meaningful header (apart from the id)

```

```
npm info start todomvc-swarm@0.0.1
```

```
> todomvc-swarm@0.0.1 start /todo
> node TodoAppServer.js
```

```
Swarm server started port 8000
```

```
^C (3)
```

```
$ docker ps -a (4)
```

CONTAINER ID	IMAGE	COMMAND	CREATED
b9db5ada0461	todoapp:latest	"npm start"	2 minutes ago

➤ STATUS PORTS NAMES
 ➤ Exited (130) 2 minutes ago example1

```
$ docker start example1 (5)
```

```
example1
```

```
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
b9db5ada0461	todoapp:latest	"npm start"	8 minutes ago

➤ STATUS PORTS NAMES
 ➤ Up 10 seconds 0.0.0.0:8000->8000/tcp example1 (6)

```
$ docker diff example1 (7)
```

```
C /todo (8)
```

```
A /todo/.swarm
```

```
A /todo/.swarm/ToDoItem
A /todo/.swarm/ToDoItem/1t10c02+A~4Uzcz (9)
A /todo/.swarm/_log
A /todo/dist
A /todo/dist/LocalToDoApp.app.js
A /todo/dist/ToDoApp.app.js
A /todo/dist/react.min.js
```

(1) ❶ `docker run`子命令启动容器，`-p`将容器的8000端口映射到宿主机的8000端口上，`--name`给容器赋予一个唯一的名字，最后一个参数是镜像

(2) 容器的启动进程的输出被发送到终端中

(3) ❷ 在此按`Ctrl+C`终止进程和容器

(4) ❸ 运行这个命令查看已经启动和移除的容器，以及其ID和状态（就像进程一样）

(5) ❹ 重新启动容器，这次是在后台运行

(6) ❺ 再次运行`ps`命令查看发生变化的状态

(7) ❻ `docker diff`子命令显示了自镜像被实例化成一个容器以来哪些文件受到了影响

(8) ❼ 修改了`/todo`目录

(9) ❽ 增加了`/todo/.swarm`目录

`docker run`子命令启动容器❶。`-p`标志将容器的8000端口映射到宿主机的8000端口上，读者现在应该可以使用浏览器访问`http://localhost:8000`来查看这个应用程序了。`--name`标志赋予了容器一个唯一的名称，以便后面引用。最后的参数是镜像名称。

一旦容器启动，我们按下CTRL-C终止进程和容器②。读者可以运行ps命令查看被启动且未被移除的容器③。注意，每个容器都具有自己的容器ID和状态，与进程类似。它的状态是Exited（已退出），不过读者可以重新启动它④。这么做之后，注意状态已经改变为Up（运行中），且容器到宿主机的端口映射现在也显示出来了⑤。

docker diff子命令显示了自镜像被实例化成一个容器以来哪些文件受到了影响⑥。在这个示例中，修改了todo目录⑦，并增加了其他列出的文件⑧。没有文件被删除，这是另一种可能性。

如读者所见，Docker“包含”环境的事实意味着用户可以将其视为一个实体，在其上执行的动作是可预见的。这赋予了Docker宽广的能力——用户可以影响从开发到生产再到维护的整个软件生命周期。这种改变正是本书所要描述的，在实践中展示Docker所能完成的东西。

接下来读者将了解Docker的另一个核心概念——分层。

1.2.5 Docker分层

Docker分层协助用户管理在大规模使用容器时会遇到的一个大问题。想象一下，如果启动了数百甚至数千个to-do应用，并且每个应用都需要将文件的一份副本存储在某个地方。

可想而知，磁盘空间会迅速消耗光！默认情况下，Docker在内部使用写时复制（copy-on-write）机制来减少所需的硬盘空间量（见图1-9）。每当一个运行中的容器需要写入一个文件时，它会通过将该项目复制到磁盘的一个新区域来记录这一修改。在执行Docker提交时，这块磁盘新区域将被冻结并记录为具有自身标识符的一个层。

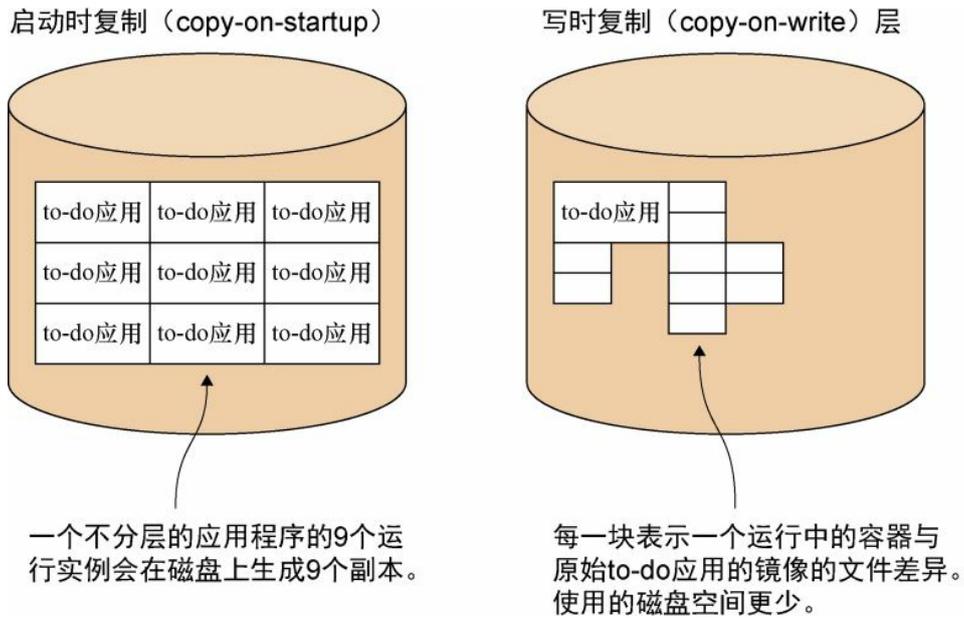


图1-9 启动时复制与写时复制对比

这部分解释了Docker容器为何能如此迅速地启动——它们不需要复制任何东西，因为所有的数据已经存储为镜像。

写时复制 写时复制是计算技术中使用的一种标准的优化策略。在从模板创建一个新的（任意类型）对象时，只在数据发生变化时才能将其复制进来，而不是复制整个所需的数据集。依据用例的不同，这能省下相当可观的资源。

图1-10展示了构建的to-do应用，它具有我们所感兴趣的3层。

因为层是静态的，所以用户只需要在想引用的镜像之上进行构建，变更的内容都在上一层中。在这个to-do应用中，我们从公开可用的node镜像构建，并将变更叠加在最上层。

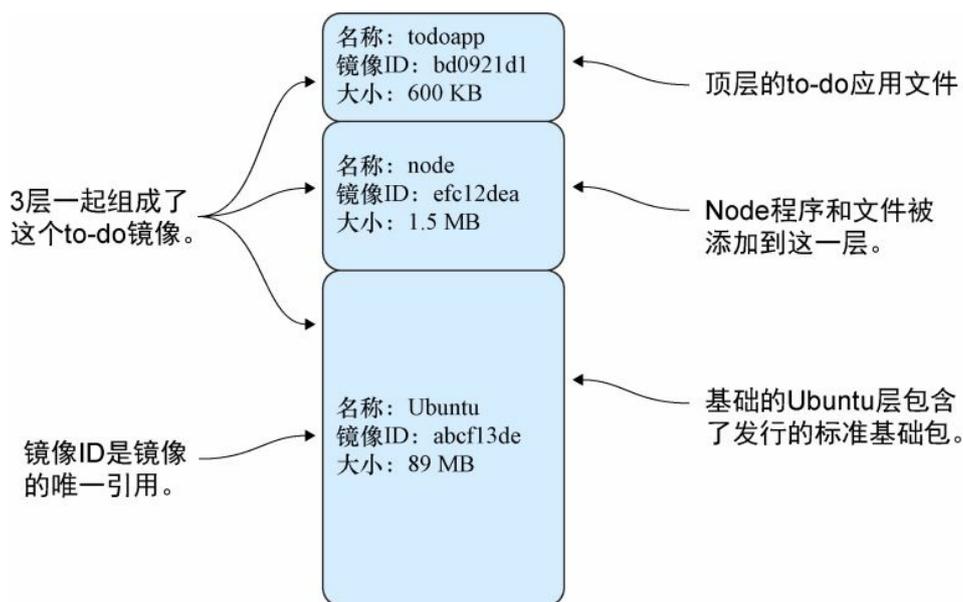


图1-10 Docker中to-do应用的文件系统分层

所有这3层都可以被多个运行中的容器共享，就像一个共享库可以在内存中被多个运行中的进程共享一样。对于运维人员来说，这是一项至关重要的功能，可以在宿主机上运行大量基于不同镜像的容器，而不至于耗尽磁盘空间。

想象一下，你将所运行的to-do应用作为在线服务提供给付费用户，可以将服务扩散给大量用户。如果你正在开发中，可以一下在本地机器上启动多个不同的环境。如果你正在进行测试，可以比之前同时运行更多测试，速度也更快。有了分层，所有这些东西都成为了可能。

通过使用Docker构建和运行一个应用程序，读者开始见识到Docker能给 workflow 带来的威力。重现并分享特定的环境，并能在不同的地方落地，让用户在开发过程中兼具灵活性和可控性。

1.3 小结

依据读者以往的Docker经验不同，本章可能存在一个陡峭的学习曲线。我们在短时间内讨论了大量的基础内容。

读者现在应该：

- 理解Docker镜像是什么；
- 知道Docker分层是什么，以及为什么它很有用；
- 能够从一个基础镜像提交一个新的Docker镜像；
- 知道Dockerfile是什么。

我们已经使用这项知识：

- 创建了一个有用的应用程序；
- 毫不费力地重现了一个应用程序中的状态。

接下来，我们将介绍几个有助于读者理解Docker如何工作的技巧，然后讨论有关Docker用法的一些更广泛的技术争论。前两个介绍性的章节构成了本书其余部分的基础，后者涵盖了从开发到生产的全过程，展示如何使用Docker来提升 workflow。

第2章 理解Docker——深入引擎室

本章主要内容

- Docker的架构
- 在用户的宿主机上追溯Docker的内部结构
- 使用Docker Hub查找和下载镜像
- 设置自己的Docker注册中心（registry）
- 实现容器间的相互通信

掌握Docker的架构是更全面地理解Docker的关键。在本章中，读者将在自己的主机和网络上对Docker的主要组件进行大致了解，并学习一些有助于增进这种理解的技巧。

在这个过程中，将学习一些有助于更有效地使用Docker（及Linux）的小窍门。后续的更高级的很多技巧都是基于这里所见的部分，因此请特别留意以下内容。

2.1 Docker的架构

图2-1展示了Docker的架构，这将是本章的核心内容。我们将从高层次视角入手，然后聚焦到每个部分，使用设计好的技巧来巩固理解。

宿主机上的Docker（在编写本书时）分成两个部分：一个具有REST风格API的守护进程，以及一个与守护进程通信的客户端。图2-1展示的是运行着Docker客户端和守护进程的宿主机。

REST风格 一个REST风格API是指使用标准HTTP请求类型，如GET、POST、DELETE等，来执行通常符合HTTP设计者预想的功能的API。

调用Docker客户端可以从守护进程获取信息或给它发送指令。守护进程是一个服务器，它使用HTTP协议接收来自客户端的请求并返回响应。相应地，它会向其他服务发起请求来发送和接收镜像，使用的同样是HTTP协议。该服务器将接收来自命令行客户端或被授权连接的任何人的请求。守护进程还负责在幕后处理用户的镜像和容器，而客户端充当的是用户与REST风格API之间的媒介。

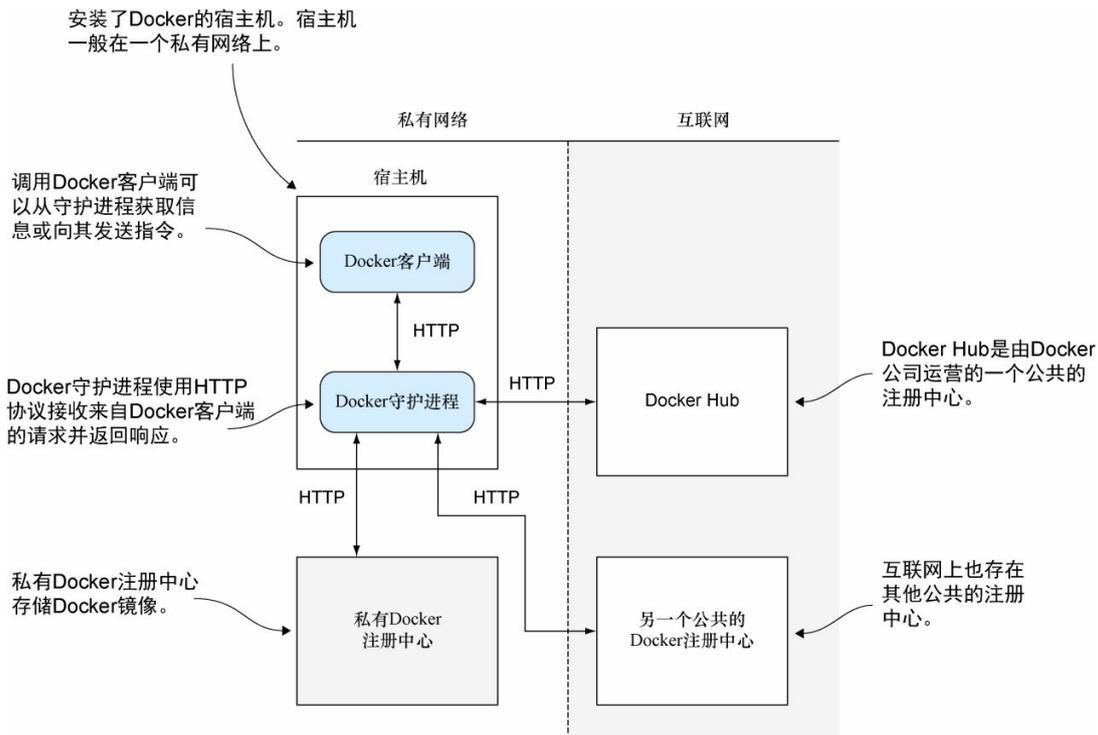


图2-1 Docker架构概览

私有Docker注册中心是存储Docker镜像的一项服务，可以从任何有相应权限的Docker守护进程向其发送请求。这个注册中心处于内部网络中，不能公开访问，因此被视为是私有的。

宿主机一般坐落在一个私有网络上。在收到请求时，Docker守护进程将连接互联网来获取镜像。

Docker Hub是由Docker公司运营的一个公共的注册中心。互联网上也存在其他公共的注册中心，且Docker守护进程可与之进行交互。

在第1章中我们说可以将Docker容器分发到任何能运行Docker的地方——这并不完全正确。实际上，只有当守护进程可以被安装到机器上时，容器才能在这台机器上运行。最明显的事实是，Docker客户端可以运行在Windows上，但守护进程（还）不行。

理解这张图的关键在于，当用户在自己的机器上运行Docker时，与其进行交互的可能是自己机器上的另一个进程，或者甚至是运行在内部网络或互联网上的服务。

现在，对Docker的结构有了大致的印象，我们来介绍几个与图中不同部分有关的技巧。

2.2 Docker守护进程

Docker守护进程（见图2-2）是用户与Docker交互的枢纽，因而它是理解所有相关部分的最佳切入点。它控制着用户机器上的Docker访问权限，管理着容器与镜像的状态，同时代理着与外界的交互。

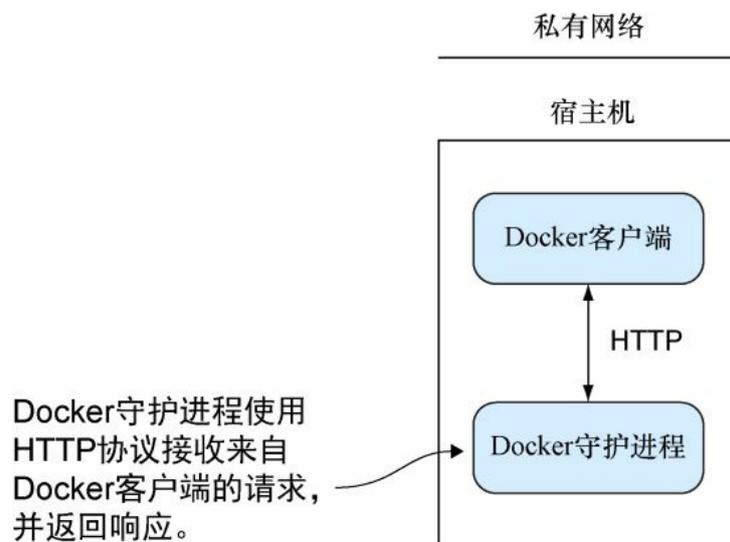


图2-2 Docker守护进程

守护进程与服务器 守护进程是运行在后台的一个进程，不在用户的直接控制之下。服务器是负责接受客户端请求，并执行用于满足该请求所需的操作的一个进程。守护进程通常也是服务器，接收来自客户端的请求，为其执行操作。`docker`命令是一个客户端，而Docker守护进程则作为服务器对Docker容器和镜像进行操作。

我们来看几个技巧，这些技巧用于展示Docker作为守护进程高效运行，同时使用`docker`命令与其进行的交互被限制为执行操作的简单请求，就像与Web服务器进行交互一样。第一个技巧允许其他人连接你的Docker守护进程，并执行与你在宿主机上所能执行的相同操作，第二个技巧说明的是Docker容器是由守护进程管理的，而不是你的shell会话。

技巧1 向世界开放Docker守护进程

虽然默认情况下Docker的守护进程只能在宿主机上访问，但是有些情况下还是需要允许其他人访问它。读者可能遇到了一个问题，需要其他人来远程调试，或者可能想让DevOps workflow中的某一部分在

宿主机上启动一个进程。

不安全！ 尽管这个技巧很强大也很有用，但它被认为是不安全的。开放的Docker守护进程可能被别有用心的人所利用，并获得提升的权限。

问题

想要将Docker服务器开放给其他人访问。

解决方案

使用开放的TCP地址启动Docker守护进程。

讨论

图2-3给出了这个技巧的工作概览。

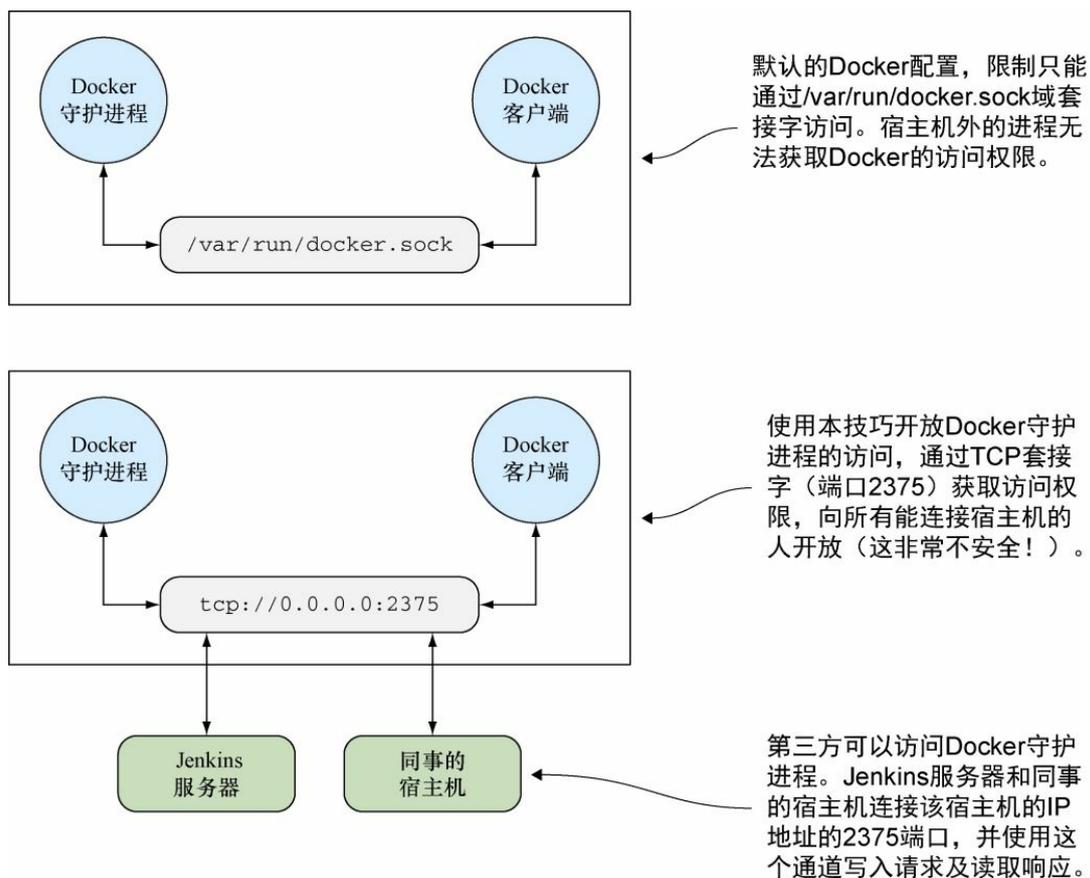


图2-3 Docker可访问性：正常情况与公开情况

在开放Docker守护进程之前，必须先停止正在运行的实例。操作的方式因操作系统而异。如果不清楚怎么做，可以首先试试这个命令：

```
$ sudo service docker stop
```

如果得到一个类似下面这样的消息，说明这是一个基于systemctl的启动系统：

```
The service command supports only basic LSB actions (start, stop, restart, try-restart, reload, force-reload, status). For other actions, please try to use systemctl.
```

可以试试这个命令：

```
$ systemctl stop docker
```

如果这个方法有效，以下命令将看不到任何输出：

```
ps -ef | grep -E 'docker (-d|daemon)\b' | grep -v grep
```

一旦Docker守护进程停止，就可以使用以下命令手工重启并向外界用户开放它：

```
docker daemon -H tcp://0.0.0.0:2375
```

这个命令以守护进程方式启动Docker（`docker daemon`），使用`-H`标志定义宿主机服务器，使用TCP协议，绑定到所有IP地址上（使用`0.0.0.0`），并以标准的Docker服务器端口（`2375`）开放。如果Docker提示`daemon`不是一个有效的子命令，请尝试使用旧版的`-d`参数。

可以从外部使用如下命令进行连接：

```
$ docker -H tcp://<宿主机IP>:2375
```

需要注意的是，在本地机器内部也需要这么做，因为Docker已经不再在其默认位置进行监听了。

如果想在宿主机上让这项变更永久生效，需要对启动系统进行配置。其操作方式参见附录B。

使用**IP限制** 如果开放了守护进程，请务必只向特定IP范围开放，同时不要绑定到`0.0.0.0`上，这样非常不安全！

技巧2 以守护进程方式运行容器

在熟悉了Docker之后，（与我们一样）读者会开始思考Docker的其他使用场景，首先想到的使用场景之一是以运行服务的方式来运行Docker容器。

以服务方式运行Docker容器，通过软件隔离实现可预测行为，是Docker的主要使用场景之一。本技巧将让读者使用适合自己的操作的方式来管理服务。

问题

想要以服务方式在后台运行一个Docker容器。

解决方案

在`docker run`命令中使用`-d`标志，并使用相关的容器管理标志定义此服务特性。

讨论

Docker容器与多数进程一样，默认在前台运行。在后台运行Docker容器最常见的方式是使用标准的`&`控制操作。虽然这行得通，但如果用户注销终端会话就可能出现问题，用户被迫使用`nohup`标志，而这将在本地目录中创建一个不得管理的输出文件.....是的，使用Docker守护进程的功能完成这一点将简洁得多。

要做到这一点，可使用`-d`标志。

```
$ docker run -d -i -p 1234:1234 --name daemon ubuntu nc -l 1234
```

与`docker run`一起使用的`-d`标志将以守护进程方式运行容器。`-i`标志则赋予容器与Telnet会话交互的能力。使用`-p`将容器的1234端口公布到宿主主机上。通过`--name`标志赋予容器一个名称，以

便后期用来对它进行引用。最后，使用netcat（nc）在1234端口上运行一个简单的监听应答（echo）服务器。

如果现在使用Telnet连接它并发送消息，就可以使用docker logs命令看到容器已经接收到该消息，如代码清单2-1所示。

代码清单2-1 使用Telnet连接容器netcat服务器

```
$ telnet localhost 1234 (1)
Trying ::1...
Connected to localhost.
Escape character is '^]'.
hello daemon (2)
^] (3)

telnet> q (4)
Connection closed.
$ docker logs daemon (5)
hello daemon
$ docker rm daemon (6)
daemon
$
```

- (1) 使用telnet命令连接到容器的netcat服务器
- (2) 输入发送给netcat服务器的一行文本
- (3) 按Ctrl+]然后按回车键退出Telnet会话
- (4) 输入q然后按回车键退出Telnet程序
- (5) 运行docker logs命令查看容器的输出
- (6) 使用rm命令清除容器

由此可见，以守护进程方式运行一个容器是相当简单的，但在实际操作中，还有一些问题有待解答：

- 如果服务失败了会发生什么？
- 在服务结束时会发生什么？
- 如果服务不断失败会发生什么？

幸运的是，**Docker**为每个问题都提供了相应标志！

非必需的标志 尽管重启标志经常会与守护进程标志（**-d**）一起使用，但从技术角度来说，与**-d**一起运行这些标志并不是必需的。

重启标志

`docker run --restart`标志允许用户应用一组容器终止时需要遵循的规则（就是所谓的“重启策略”，见表2-1）。

表2-1 重启策略

策 略	描 述
no	容器退出时不重启
always	容器退出时总是自动重启
on-failure[:max-retry]	只在失败时重启

no策略很简单：当容器退出时，它不会被重启。这是默认值。

always策略也很简单，不过还是值得简要讨论一下：

```
$ docker run -d --restart=always ubuntu echo done
```

这个命令以守护进程方式（`-d`）运行容器，并总是在容器终止时自动重启（`--restart=always`）。它发送了一个简单的快速完成的`echo`命令，然后退出容器。

如果运行了上述命令，然后运行`docker ps`命令，就会看到类似下面这样的输出：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
➤ STATUS          PORTS              NAMES
69828b118ec3      ubuntu:14.04      "echo done"        4 seconds ago
➤ Restarting (0)   Less than a second ago    sick_brattain
```

`docker ps`命令列出了所有运行中的容器及其信息，包括以下内容：

- 容器什么时候被创建的（`CREATED`）；
- 容器的当前状态——通常将是`Restarting`，因为它只运行了很短的时间（`STATUS`）；
- 容器上一次运行的退出码（也在`STATUS`下面）。0代表运行成功；
- 容器名称。默认情况下，`Docker`会通过连接两个随机单词为容器命名。有时这会造成一些奇怪的结果！

注意，`STATUS`一栏还告诉我们，容器在不到一秒前退出并正在重启。这是因为`echo done`命令会立即退出，而`Docker`必须不断地重启这个容器。

需要特别说明的是，`Docker`复用了容器ID。这个ID在重启时不会改变，并且对于这个`Docker`调用来说，`ps`表里永远只会有一条。

最后，`on-failure`策略只在容器从它的主进程返回一个非0（一般表示失败）退出码时重启：

```
$ docker run -d --restart=on-failure:10 ubuntu /bin/false
```

这条命令以守护进程（`-d`）形式运行容器，并设置了重启的尝试次数限制（`--restart=on-failure:10`），如果超出限制则退出。它运行了一个快速完成并肯定失败的简单的命令（`/bin/false`）。

如果运行上述命令并等待一分钟，然后运行`docker ps -a`，就会看到类似下面这样的输出：

```
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED
➤ STATUS      PORTS         NAMES
b0f40c410fe3  ubuntu:14.04  "/bin/false"           2 minutes ago
➤ Exited (1) 25 seconds ago              loving_rosalind
```

技巧3 将Docker移动到不同分区

Docker把所有与容器和镜像有关的数据都存储在一个目录下。由于它可能会存储大量不同的镜像，这个目录可能会迅速变大！

如果宿主机具有不同分区（这在企业Linux工作站上很常见），用户可能会更快遭遇空间限制。在这种情况下，用户会想移动Docker所操作的目录。

问题

想要移动Docker存储数据的位置。

解决方案

停止Docker守护进程，并使用`-g`标志指定新的位置来启动。

讨论

首先必须将Docker守护进程停止（有关这个问题的讨论参见附录B）。

假设想在/home/dockeruser/mydocker运行Docker。运行下列命令将在这个目录中创建一组新的目录和文件：

```
docker daemon -g /home/dockeruser/mydocker
```

这些目录是Docker内部使用的，对其进行操作风险自担（因为我们已经尝过滋味了！）。

请注意，这看起来像是把容器和镜像从之前的Docker守护进程清除了。不过不用担心。如果杀掉刚才运行的Docker进程，并重启Docker服务，Docker客户端就会指回它原来的位置，容器和镜像也将回归。

如果想让这个移动永久有效，需要对宿主机系统的启动进程进行相应配置。

2.3 Docker客户端

Docker客户端（见图2-4）是Docker架构中最简单的部件。在主机上输入docker run或docker pull这类命令时运行的便是它。它的任务是通过HTTP请求与Docker守护进程进行通信。

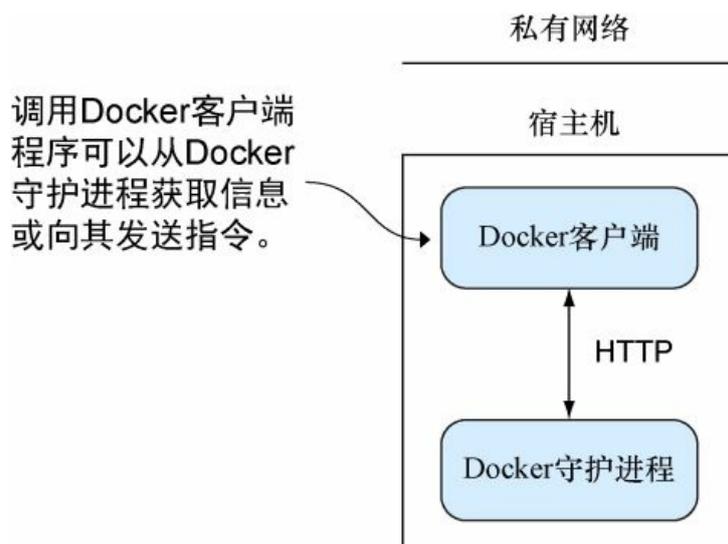


图2-4 Docker客户端

在本节中，读者将看到如何监听Docker客户端与服务器之间的信息，还将看到一些与端口映射有关的基本技巧，这是向本书后续的编排章节迈进的一小步，也是使用浏览器作为Docker客户端的一种方式。

技巧4 使用socat监控Docker API流量

有时docker命令可能会不按预期工作。多数时候是因为没有理解命令行参数的某些部分，不过偶尔也存在更严重的安装问题，如Docker的二进制文件过时了。为了诊断问题，查看与之通信的Docker守护进程来往的数据流是十分有用的。

Docker不是不稳定的 不用惊慌！本技巧的存在不表示Docker需要经常调试，或者有任何的不稳定！这条技巧在此是为了理解Docker架构的一个工具，同时也是为了介绍socat这个强大的工具。如果读者像我们一样，在众多不同的地方使用Docker，所使用的Docker版本将会有差异。与任何软件一样，不同的版本将具有不同的功能和标志，这可能会让读者无所适从。

问题

想要调试一个Docker命令的问题。

解决方案

使用流量监听器（traffic snooper）来检查API调用，并自行解决。

讨论

在本技巧中，用户将在自己的请求与服务器套接字之间插入一个代理Unix域套接字，并查看通过它的内容（如图2-5所示）。注意，要完成这一步需要root或sudo权限。

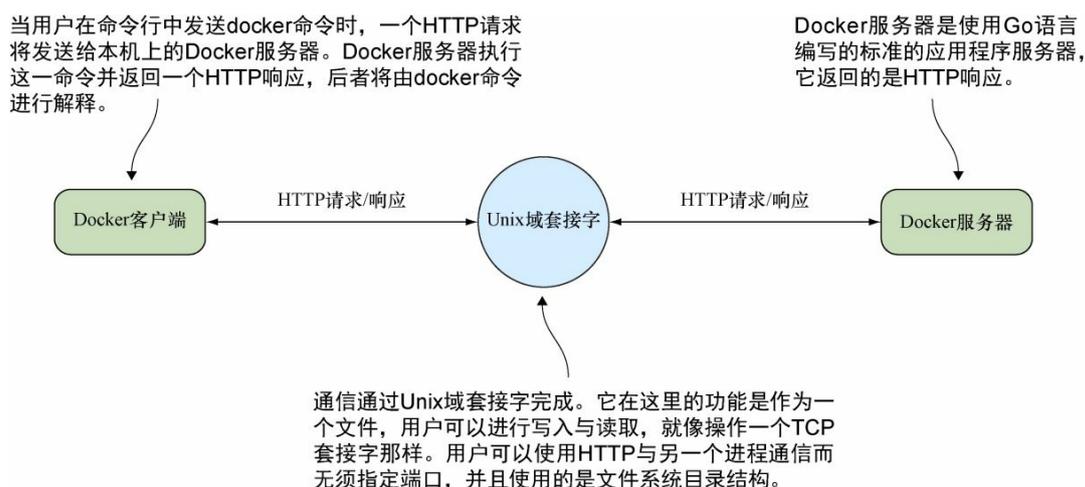


图2-5 宿主机上的Docker客户/服务器架构

要创建这个代理，会用到socat。

```
$ sudo socat -v UNIX-LISTEN:/tmp/dockerapi.sock \  
UNIX-CONNECT:/var/run/docker.sock &
```

socat socat是一个强大的命令，能让用户在两个几乎任意类型的数据通道之间中继数据。如果熟悉netcat，可以将其看作是加强版的netcat。

在这条命令中，`-v`用于提高输出的可读性，带有数据流的指示。`UNIX-LISTEN`部分是让`socat`在一个Unix套接字上进行监听，而`UNIX-CONNECT`是让`socat`连接到Docker的Unix套接字。“&”符号指定在后台运行该命令。

发往守护进程的请求所经过的新路由如图2-6所示。所有双向流量都会被`socat`看到，并与Docker客户端所提供的任何输出一起记录到终端日志中。

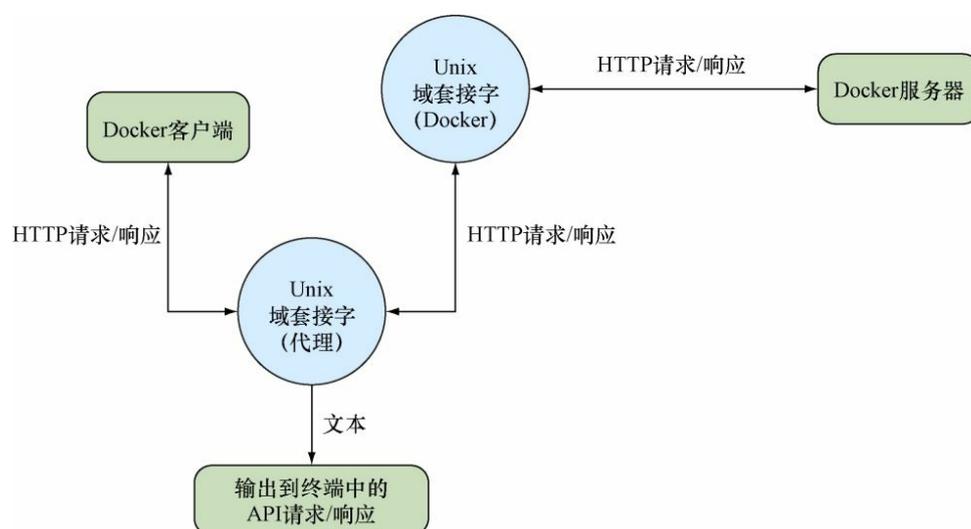


图2-6 插入`socat`作为代理的Docker客户端与服务器

现在一个简单的`docker`命令的输出看起来将类似下面这样：

```
$ docker -H unix:///tmp/dockerapi.sock ps -a (1)
> 2015/01/12 04:34:38.790706 length=105 from=0 to=104 (2)
GET /v1.16/containers/json?all=1 HTTP/1.1\r
Host: /tmp/dockerapi.sock\r
User-Agent: Docker-Client/1.4.1\r
\r
< 2015/01/12 04:34:38.792516 length=544 from=0 to=543 (3)
HTTP/1.1 200 OK\r
Content-Type: application/json\r
Date: Mon, 12 Jan 2015 09:34:38 GMT\r
Content-Length: 435\r
```

```

\r
[{"Command":"/bin/bash","Created":1420731043,"Id":
➤ "4eec1b50dc6db7901d3b3c5a8d607f2576829fd6902c7f658735c3bc0a09a39c",
➤ "Image":"debian:jessie","Names":["/lonely_mclean"],"Ports":[],
➤ "Status":"Exited (0) 3 days ago"} (4)
,{"Command":"/bin/bash","Created":1420729129,"Id":
➤ "029851aeccc887ecf9152de97f524d30659b3fa4b0dcc3c3fe09467cd0164da5",
➤ "Image":"debian:jessie","Names":["/suspicious_torvalds"],"Ports":[],
➤ "Status":"Exited (130) 3 days ago"}
]CONTAINER ID          IMAGE          COMMAND          CREATED
➤ STATUS              PORTS          NAMES (5)
4eec1b50dc6d          debian:jessie  "/bin/bash"      3 days ago
➤ Exited (0) 3 days ago          lonely_mclean
029851aeccc8          debian:jessie  "/bin/bash"      3 days ago
➤ Exited (130) 3 days ago          suspicious_torvalds

```

(1) 用于查看请求与响应所发送的命令

(2) HTTP请求从此处开始，左侧带有右尖括号

(3) HTTP响应从此处开始，左侧带有左尖括号

(4) 来自Docker服务器的响应的JSON内容

(5) 用户正常看到的输出，由Docker客户端从前面的JSON解释而来

小心 如果在前面的示例中以root身份运行socat，需要使用sudo来运行docker -H命令。这是因为dockerapi.sock文件的所有者是root。

使用socat不仅对Docker来说是一种强大的调试方式，对工作过程中可能碰到的任何其他网络服务也是如此。

技巧5 使用端口连接容器

Docker容器从一开始就被设计用于运行服务。在大多数情况下，都是这样或那样的 HTTP服务。其中很大一部分是可以使用浏览器访问的Web服务。

这会造成一个问题。如果有多个Docker容器运行在其内部环境中的80端口上，它们将无法全部通过宿主机的80端口进行访问。接下来的技巧将展示如何通过暴露和映射容器的端口来管理这一常见场景。

问题

想要将多个运行在同一个端口的Docker容器服务暴露到宿主机上。

解决方案

使用Docker的-p标志将容器的端口映射到宿主机上。

讨论

在这个示例中，我们将使用tutum-wordpress镜像。假设想在宿主机上运行两个实例来服务不同的博客。

由于此前有很多人想这么做，已经有人准备了任何人都可以获取并启动的镜像。要从外部地址获取镜像，可以使用docker pull命令。在默认情况下，镜像将从Docker Hub下载：

```
$ docker pull tutum/wordpress
```

要运行第一个博客，可使用如下命令：

```
$ docker run -d -p 10001:80 --name blog1 tutum/wordpress
```

这里的docker run命令以守护进程方式（-d）及发布标志（-p）运

行容器。它指定将宿主机端口（**10001**）映射到容器端口（**80**）上，并赋予该容器一个名称用于识别它（`--name blog1 tutum/wordpress`）。

可以对第二个博客做相同操作：

```
$ docker run -d -p 10002:80 --name blog2 tutum/wordpress
```

如果现在运行这个命令：

```
$ docker ps -a | grep blog
```

将看到列出的两个博客容器及其端口映射，看起来像下面这样：

```
9afb95ad3617 tutum/wordpress:latest "/run.sh" 9 seconds ago Up 9 seconds
➤ 3306/tcp, 0.0.0.0:10001->80/tcp blog1
31ddc8a7a2fd tutum/wordpress:latest "/run.sh" 17 seconds ago Up 16 seconds
➤ 3306/tcp, 0.0.0.0:10002->80/tcp blog2
```

现在可以通过浏览<http://localhost:10001>和<http://localhost:10002>来访问自己的容器。

要在完成后删除这些容器（假设不想保留它们），可运行下面这个命令：

```
$ docker rm -f blog1 blog2
```

如果需要的话，现在就可以通过管理端口分配在宿主机上运行多个相同的镜像和服务了。

牢记 **-p** 标志的参数顺序 在使用 **-p** 标志时，很容易忘记哪个端口属于宿主机，哪个端口属于容器。我们可以将它看作是在从左向右读

一个句子。用户连接到宿主机（-p），并从宿主机的端口传递到容器的端口（宿主机端口:容器端口）。如果熟悉SSH的端口转发命令的话，会发现它们的格式是一样的。

技巧6 链接容器实现端口隔离

技巧5展示的是如何通过暴露端口将容器开放给宿主机网络。用户不会总想将服务暴露给宿主机或外界，但是会希望容器彼此相连。

本技巧展示的是如何使用Docker的链接标志来实现这一点，并确保外人无法访问内部服务。

问题

出于内部目的，想要让容器间实现通信。

解决方案

使用Docker的链接功能可以让容器彼此通信。

讨论

继续安装WordPress的任务，我们将把mysql数据库层从wordpress容器中分离出来，并将它们链接在一起，且不需要进行端口配置。图2-7展示了最终状态的概览。

为什么这一点很有用 既然已经可以将端口暴露给宿主机来使用，为什么还要用链接？链接可以让用户封装并定义容器间的关系，而无须将服务暴露给宿主机网络（即可能暴露给外界）。用户可能会因为安全因素而这么做。

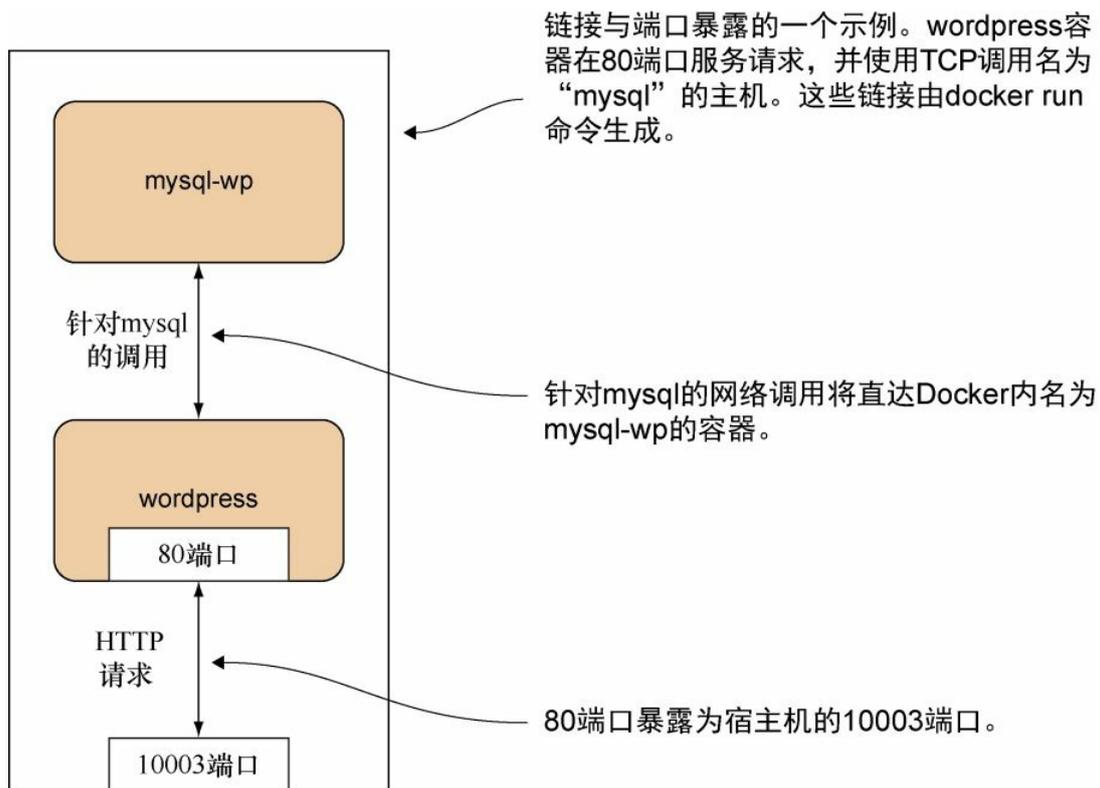


图2-7 使用链接容器设置WordPress

要像这样运行容器，可按照以下顺序执行，并在第一条和第二条命令之间暂停大约一分钟：

```
$ docker run --name wp-mysql \
  -e MYSQL_ROOT_PASSWORD=yoursecretpassword -d mysql (1)
$ docker run --name wordpress \
  --link wp-mysql:mysql -p 10003:80 -d wordpress (2)
```

(1) ❶

(2) ❷

首先将mysql容器命名为wp-mysql，用于在后面引用它❶。还需要提供一个环境变量以便mysql容器可以初始化数据库（-e MYSQL_ROOT_PASSWORD=yoursecretpassword）。两个容器都以守护进程方式运行（-d），同时使用了Docker Hub上官方mysql镜像的

引用。

在第二个命令②中，将wordpress容器命名为wordpress，以备后面要引用它。同时将wp-mysql容器链接到wordpress容器中（`--link wp-mysql:mysql`）。在wordpress容器内对mysql服务器的引用将被发送到名为wp-mysql的容器中。有如技巧5所述，使用了一个本地端口映射（`-p 10003:80`），并添加了Docker Hub上官方wordpress镜像（wordpress）的引用。请注意，链接不会等待被链接容器启动，因此才有在命令之间暂停的指示。完成这一步更精确的方法是，在运行wordpress容器之前，在`docker logs wp-mysql`的输出中查找mysqlid: ready for connections。

如果现在浏览<http://localhost:10003>，将会看到wordpress介绍画面，并可设置这个wordpress实例。

这个示例的关键在于第二条命令里的`--link`标志。这个标志会设置容器的host文件以便wordpress容器能够引用mysql服务器，这将被路由到具有“wp-mysql”名称的容器。这有很大的好处，即无须对wordpress容器做任何改动，就可以将不同的mysql容器交换进来，使不同服务的配置管理变得更简单。

启动顺序至关重要 容器必须以正确的顺序启动，以便能对已经存在的容器名称做映射。截至编写本书时，Docker不具备动态解析链接的功能。

为了使用这种方式链接容器，在构建镜像时必须指定暴露容器的端口。这可以通过在镜像构建的Dockerfile中使用EXPOSE命令来达成。

现在，已经见识了Docker编排的一个简单示例，并朝着微服务构架前进了一步。在这个例子中，可以在不影响wordpress容器的同时对mysql容器进行操作，反之亦然。这种对运行中服务的细粒度控制是微服务构架的关键的运维优势之一。

技巧7 在浏览器中使用Docker

销售新技术可能很艰难，因此简单而有效的演示是非常有价值的。让演示可操作则效果更佳，这也是为什么我们发现，为了以易于达成的方式给新手带来Docker的初体验，创建一个能在浏览器中与容器进行交互的网页是一个非常棒的技巧。这种让人眼前一亮的体验没有坏处！

问题

想要演示Docker的强大威力，用户无须自己安装Docker或运行自己不理解命令。

解决方案

使用一个开放端口启动Docker守护进程，并启用CORS^[1]。然后使用所选择的Web服务器为Docker终端仓库提供服务。

讨论

REST API最常见的用法是在一台服务器上暴露它，并在一个网页上使用JavaScript来调用。由于Docker正巧是通过REST API来执行所有交互的，因此可以使用相同方式来控制Docker。尽管一开始看起来有点儿令人惊讶，但这种控制一直延伸到能通过浏览器里的终端与容器进行交互。

我们在技巧1中已经讨论过如何在2375端口上启动守护进程，因

而不再赘述。此外，CORS太庞大，这里无法深入讲述（可以参考Monsur Hossain所著的*CORS in Action*[Manning Publications, 2014]）——简言之，它是小心地绕过限制JavaScript只能访问当前域这一常规限制的一种机制。在这个例子中，它将允许守护进程监听一个与提供Docker终端页面不同的端口上。要启用它，需要使用**--api-enable-cors**选项和用于监听端口的选项一起来启动Docker守护进程。

现在，先决条件已经梳理好，我们将它运行起来。首先，需要获取代码：

```
git clone https://github.com/aidanhs/Docker-Terminal.git
cd Docker-Terminal
```

然后需要提供文件服务：

```
python2 -m SimpleHTTPServer 8000
```

上述命令使用Python内置的一个模块为目录中的静态文件服务。用户可以使用任何自己喜欢的等效服务。

现在可以在浏览器中访问<http://localhost:8000>并启动一个容器。

图2-8展示了Docker终端是如何连接起来的。页面托管在本地计算机中，并连接到本地计算机上的Docker守护进程，以执行所有操作。

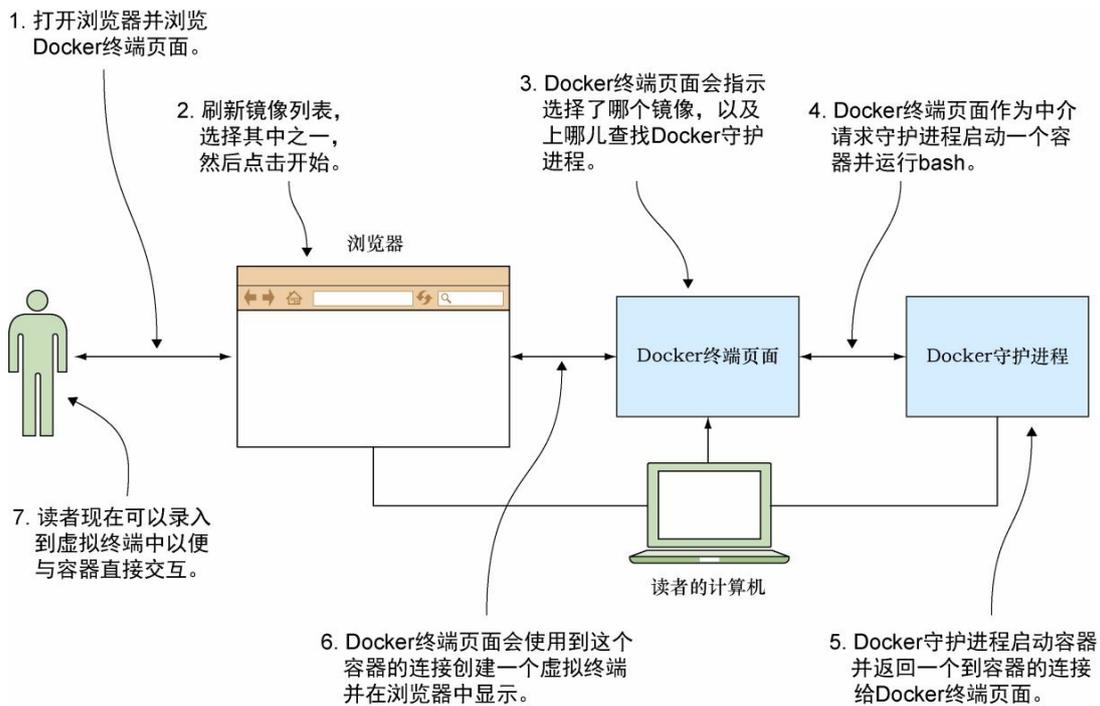


图2-8 Docker终端是如何工作的

如果想把链接发送给其他人，以下几点值得注意。

- 其他人不能使用任何类型的代理。这是我们见到的最常见的错误根源——Docker终端使用Websocket，后者目前无法通过代理工作。
- 给出指向localhost的链接明显无法工作——需要给出外部IP地址。
- Docker终端需要知道上哪儿找到Docker API——它应该可以根据浏览器访问的地址自动完成这一点，不过这一点需要留意。

这里为什么不使用**Docker** 如果读者的Docker经验更丰富，会奇怪为什么我们没在这个技巧中使用Docker。原因是，我们还在介绍Docker，不想给刚接触Docker的读者增加复杂度。“Docker化”这个技巧将作为一个练习留给读者。

2.4 Docker注册中心

一旦创建了镜像，读者可能就想与其他用户分享它。这是Docker注册中心概念的所在。

图2-9中的3个注册中心差别在于它们的可达性。一个处于私有网络上，一个开放在公共网络中，而另一个是公共的但只有注册用户才能使用Docker访问。它们全部使用相同的API完成相同的功能，这就是Docker守护进程如何知道怎样与它们进行相互通信。

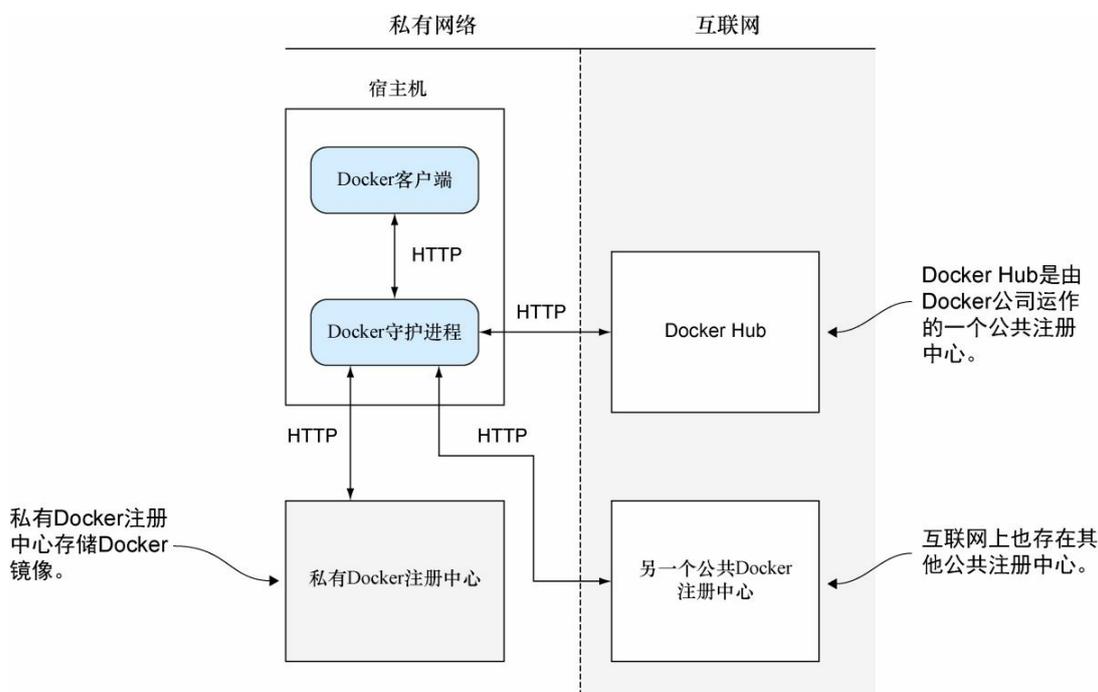


图2-9 一个Docker注册中心

Docker注册中心允许多个用户使用REST风格API将镜像推送到一个中央存储中，也可以从中拉取镜像。

与Docker自身一样，注册中心代码也是开源的。很多公司（如我们公司）建立了私有注册中心在内部存储和共享专有的镜像。这是在进一步说明Docker公司的注册中心之前，我们这里将要讨论的东西。

技巧8 建立一个本地Docker注册中心

读者已经看到Docker公司具有一项服务，人们可以在其上公开地共享他们的镜像（如果想私下进行，可以付费实现）。不过存在一些不想通过Hub来共享镜像的原因——有些商业组织想尽可能把东西保留在内部，或者镜像可能很大，通过互联网传输太慢，或者也许想在试验时保持镜像私有化，同时又不想付费。不管出于什么原因，幸运的是有一个简单的解决方案。

问题

想要一个在本地托管镜像的方法。

解决方案

在本地网络上建立一个注册中心服务器。

讨论

要让注册中心运行起来，可在一台具有大量磁盘空间的机器上发起以下命令：

```
$ docker run -d -p 5000:5000 -v $HOME/registry:/var/lib/registry registry:2
```

这条命令让注册中心运行于Docker宿主机的5000端口上（`-p 5000:5000`），并使用主目录下的`registry`作为容器的`/var/lib/registry`目录，后者是容器里的`registry`默认存储文件的位置。它同时指定了容器内的`registry`将文件存储在`/registry`目录下（`STORAGE_PATH=/registry`）。

在所有想访问这个注册中心的机器上，将下列内容添加到守护进程选项中（`HOSTNAME`是新的注册中心服务器的主机名或IP地址）： -

`-insecure-registry HOSTNAME`。

现在可以执行`docker push HOSTNAME:5000/image:tag`。

正如所见，一个本地注册中心最基础层次的配置很简单，所有数据都存储在`$HOME/registry`目录中。如果要扩容或让它变得更健壮，GitHub上的仓库（<https://github.com/docker/distribution/blob/v2.2.1/docs/storagedrivers.md>）罗列了一些可选项，例如，在Amazon S3里存储数据。

读者可能会对`--insecure-registry`选项感到好奇。为了帮助用户保持安全，Docker只允许使用签名HTTPS证书从注册中心上拉取。因为对本地网络相当信任，我们覆盖了这个选项。不过，毫无疑问的是，在互联网上这么做必须慎之又慎！

注册中心路线图 与Docker生态系统里的其他事物一样，注册中心也在发生变化。尽管注册中心镜像将保持可用及稳定，但它最终将被一个名为`distribution`（见<https://github.com/docker/distribution>）的新工具取代。

2.5 Docker Hub

Docker Hub（见图2-10）是由Docker公司维护的一个注册中心。它拥有成千上万个镜像可供下载和运行。任何Docker用户都可以在上面创建免费账号及公共Docker镜像。除了用户提供的镜像，上面还维护着一些作为参考的官方镜像。

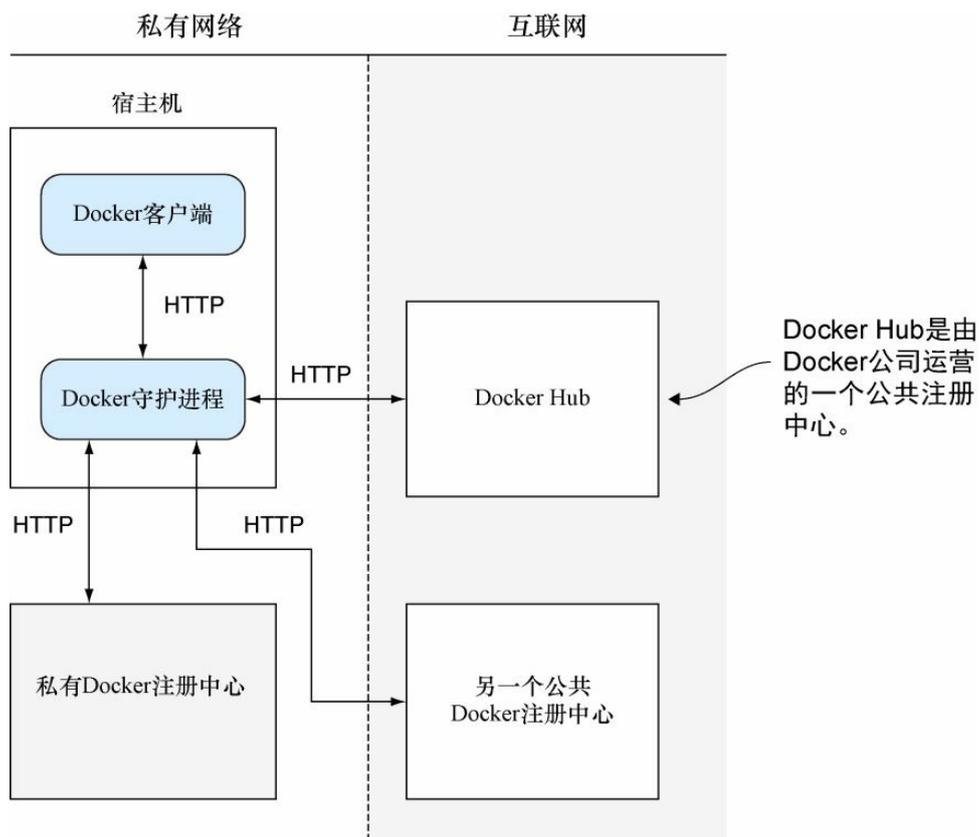


图2-10 Docker Hub

镜像受用户认证的保护，同时具有一个与GitHub类似的支持率打星系统。

这些官方镜像的表现形式可能是Linux发行版，如Ubuntu或Cent OS，或是预装软件包，如Node.js，或是完整的软件栈，如WordPress。

技巧9 查找并运行一个Docker镜像

Docker注册中心造就的是与GitHub相似的社交编码文化。如果读者有兴趣尝试一个新的软件应用程序，或正在找寻服务于某个特定用途的新的应用程序，那么Docker镜像将是一个简单的实验手段，它不会对宿主机造成干扰，不需要配备一个虚拟机，也不必担心安装步骤。

问题

想要查找一个Docker镜像形式的应用程序或工具，并进行尝试。

解决方案

使用`docker search`命令来查找要拉取的镜像，然后运行它。

讨论

假设读者对Node.js有兴趣。在下面的代码中，我们使用`docker search`命令搜索出匹配“node”的镜像：

```
$ docker search node
NAME                                DESCRIPTION
└─ STARS          OFFICIAL  AUTOMATED
node              Node.js is a JavaScript-based platform fo
r...
└─ 432            [OK]    (1)

dockerfile/nodejs              Trusted automated Node.js (http://nodejs.o
...
└─ 57              [OK]    (2)
dockerfile/nodejs-bower-grunt Trusted automated Node.js (http://nodejs.
O...
└─ 17              [OK]    (3)
nodesource/node               [OK]    (4)
selenium/node-firefox         [OK]
selenium/node-chrome          [OK]
selenium/node-base           [OK]
strongloop/node              StrongLoop, Node.js, and tools.
└─ 3                [OK]
selenium/node-chrome-debug   [OK]
└─ 3                [OK]
dockerfile/nodejs-runtime    Trusted automated Node.js runtime Build .
.
└─ 3                [OK]
jprjr/stackbrew-node         A stackbrew/ubuntu-based image for Docker
, ...
└─ 2                [OK]
selenium/node-firefox-debug  [OK]
└─ 2                [OK]
```

maccam912/tahoe-node	Follow "The Easy Way" in the description
t...	
➤ 1 [OK]	
homme/node-mapserv	The latest checkouts of Mapserver and its
...	
➤ 1 [OK]	
maxexcloo/nodejs	Docker framework container with Node.js an
...	
➤ 1 [OK]	
brownman/node-0.10	
➤ 0 [OK]	
kivra/node	Image with build dependencies for frontend
...	
➤ 0 [OK]	
thenativeweb/node	
➤ 0 [OK]	
thomaswelton/node	
➤ 0 [OK]	
siomiz/node-opencv	_/node + node-opencv
➤ 0 [OK]	
bradegler/node	
➤ 0 [OK]	
tcnksm/centos-node	Dockerfile for CentOS packaging node
➤ 0 [OK]	
azukiapp/node	
➤ 0 [OK]	
onesysadmin/node-imagetools	
➤ 0 [OK]	
fishhead/node	
➤ 0 [OK]	

- (1) `docker search`的输出是按评星数量排序的
- (2) 描述是上传者对镜像用途的解释
- (3) 官方镜像是指受Docker Hub信任的镜像
- (4) 自动化镜像是指使用Docker Hub自动化构建功能构建的镜像

一旦选择了一个镜像，就可以通过对其名称执行`docker pull`命令来下载它：

```
$ docker pull node (1)
node:latest: The image you are pulling has been verified
81c86d8c1e0c: Downloading
81c86d8c1e0c: Pull complete
3a20d8faf171: Pull complete
c7a7a01d634e: Pull complete
2a13c2a76de1: Pull complete
4cc808131c54: Pull complete
bf2afba3f5e4: Pull complete
0cba665db8d0: Pull complete
322af6f234b2: Pull complete
9787c55efe92: Pull complete
511136ea3c5a: Already exists
bce696e097dc: Already exists
58052b122b60: Already exists
Status: Downloaded newer image for node:latest (2)
```

(1) 从Docker Hub拉取名为node的镜像

(2) 如果Docker拉取了一个新的镜像（与之相对的是说明没有比已有镜像更新的版本），会显示这条信息。读者看到的输出可能会有所不同

接着，可以使用-t和-i标志以交互方式运行它。-t标志指明创建一个tty设备（一个终端），而-i标志指明该Docker会话是交互式的：

```
$ docker run -t -i node /bin/bash
root@c267ae999646:/# node
> process.version
'v0.12.0'
>
```

-ti标志惯用法 可以在上述docker run调用中用-ti取代-t -i来减少输入。从这里开始，本书将使用这种用法。

常常会有来自镜像维护人员的有关如何运行镜像的建议。在

<http://hub.docker.com>网站上搜索镜像将引导到该镜像的页面。其描述标签页可提供更多信息。

这个镜像可信吗 如果用户下载并运行了一个镜像，运行的将是自己无法充分验证的代码。虽然使用受信任的镜像具有相对的安全性，但是通过互联网下载和运行软件时，没有什么是能保证100%安全的。

有了这方面的知识和经验，现在可以对Docker Hub提供的大量资源进行挖掘了。毫不夸张地说，要试用这成千上万的镜像，有很多东西要学。请慢慢享受！

2.6 小结

在本章中，我们学习了Docker是如何结合在一起的，并且使用这一认知对不同组件进行了操作。

下面是涉及的主要领域：

- 通过TCP或Web浏览器向外界开放Docker守护进程；
- 以服务守护进程方式运行容器；
- 通过Docker守护进程将容器链接在一起；
- 监听Docker守护进程API；
- 设置自己的注册中心；
- 使用Docker Hub来查找和下载镜像。

前两章已经涵盖了基础知识（但仍希望读者学到了一些新东西，即使对Docker已经比较熟悉）。现在我们继续学习第二部分，看一看Docker在软件开发世界中扮演的角色。

[1] Cross-Origin Resource Sharing, 跨域资源共享。——译者注

第二部分 Docker与开发

在第一部分里，我们通过示例学习了Docker的核心概念和架构。第二部分会以此作为基础来演示Docker在开发环境里的应用。

第3章将讲述如何将Docker用作轻量级的虚拟机。这是一个有争议的领域。由于虚拟机和Docker容器之间存在本质的差别，使用Docker在很多情况下可以大大加快开发速度。在转向更高级的Docker使用场景前，这也是上手Docker的有效手段。第4章里会介绍20余个技巧，使结合Docker的日常开发变得更加有用和高效。除了构建和运行容器外，读者还将了解到如何使用卷来持久化数据以及如何编排Docker宿主机。第5章覆盖了重要的配置管理领域。我们将会使用Dockerfile以及传统的配置管理工具来管理Docker的构建。我们还介绍了最小化Docker镜像的创建和数据管理等内容，以期减少镜像的膨胀。在本部分结尾，读者将能够收获许多针对Docker单机使用场景的各种有用的技巧，并且准备好将Docker运用到DevOps场景。

第3章 将Docker用作轻量级虚拟机

本章主要内容

- 将虚拟机转换成Docker镜像
- 管理容器的服务启动
- 在工作的时候随时保存成果
- 在机器上管理Docker镜像
- 在Docker Hub上分享镜像
- 即玩——即用——2048 Docker版

自本世纪以来，虚拟机（virtual machine，VM）已经在软件开发和部署等领域广泛普及。机器到软件的抽象使互联网时代下的软件和服务的更替及管控变得更加轻松和廉价。

虚拟机 一台虚拟机即是一个模拟真实计算机的应用程序，它通常会运行一个操作系统和一些应用。它可以被放到任何（兼容的）可用的物理资源上。最终用户对该软件的体验是，尽管它好像是在物理机上，但那些管理硬件的人员可以专注于大规模的资源分配。

Docker并不是一项虚拟机技术。它不会模拟一台机器的硬件，也不会包括一个操作系统。一个Docker容器在默认情况下没有被约束于指定的硬件限制。如果说Docker抽象了什么的话，那便是它将服务运行的环境给虚拟化了，而不是机器本身。此外，Docker很难运行Windows软件（甚至为其他Unix衍生的操作系统编写的软件）。

虽说从某些方面来看，可以把Docker当成是一台虚拟机使用，但事实上，对互联网时代的开发人员和测试人员而言，有没有初始化进程或者是否直接和硬件交互并没有什么重大意义。而Docker和虚拟机有一些显著的共性，如它对周围硬件具有较好的隔离性以及顺应更加细粒度的软件交付方式。

本章将会带读者领略一些之前使用虚拟机但如今可以用Docker实现的场景。相比于虚拟机，使用Docker不会给用户带来任何明显的功能优势，但是Docker在更替和环境跟踪方面带来的速度及便利性也许可以改写开发流水线的游戏规则。

3.1 从虚拟机到容器

在理想的情况下，从虚拟机迁移到容器也许就是在一个与虚拟机类似的发行版的Docker镜像上运行配置管理脚本这么简单。本节会为读者展示针对非理想情况的场景，该如何从虚拟机转换到容器。

技巧10 将虚拟机转换为容器

Docker Hub上并不是拥有所有的基础镜像，因此，针对一些小众的Linux发行版和用例，人们可能需要创建自己的镜像。同样的原则也适用于现有一台虚拟机想放到Docker里从而在其上迭代或者利用Docker生态系统优势的情况。

在理想情况下，用户会想使用标准的Docker技术，如一些结合了标准配置管理工具（见第5章）的Dockerfile，从头开始构建一个等同于虚拟机的容器。然而，现实情况是，很多虚拟机都没有被仔细地做过配置管理。这一点的确可能发生，因为一台虚拟机从人们开始用它的时候起会不断地演进，而以一个更加结构化的方式重新创建它的

话，从成本上来说是不值得的。

问题

有一台虚拟机，想要将其转换成一个Docker镜像。

解决方案

为自己的虚拟机文件系统创建一个TAR文件，可以使用`qemu-nbd`、通过`ssh`执行`tar`命令或者其他方法，然后在Dockerfile里使用`ADD`命令加入TAR文件以创建自己的镜像。

讨论

首先我们将虚拟机划分为两大类：本地（虚拟机磁盘镜像放在本地，虚拟机的执行操作发生在用户的计算机上）以及远程（虚拟机磁盘镜像存储在远程，虚拟机的执行操作发生在其他地方）。

这两类虚拟机（以及其他任何用户想为之创建Docker镜像的虚拟机）在原则上是一致的——需要拿到整个文件系统的TAR，然后用`ADD`命令将TAR文件加到`scratch`镜像的`/`。

ADD命令 在镜像中拥有`ADD`命令时，Dockerfile的`ADD`命令（不像它的兄弟命令`COPY`）会自动将TAR文件（`gzip`压缩过的文件以及其他一些类似的文件类型也是如此）解压出来。

scratch镜像 `scratch`镜像是一个零字节虚拟镜像，可以基于此构建其他镜像。一般来说，它适用于想使用一个Dockerfile复制（或者添加）一个完整文件系统的情况。

现在，让我们先来看看用户有一个本地Virtualbox虚拟机的情况。

在开始之前，首先需要完成下列任务：

- 安装qemu-nbd工具（在Ubuntu上是作为qemu-utils包的一部分提供出来的）；
- 定义虚拟机的磁盘镜像的路径；
- 关闭虚拟机。

如果用户的虚拟机磁盘镜像是.vdi或者.vmdk格式的话，这一技巧应该会很奏效。其他格式则可能成败参半。

下列代码展示了该如何将用户的虚拟机文件转成一个虚拟磁盘，这样一来他也就可以从这里复制出所有文件：

```
$ VMDISK="$HOME/VirtualBox VMs/myvm/myvm.vdi" (1)
$ sudo modprobe nbd (2) (3)
$ sudo qemu-nbd -c /dev/nbd0 -r $VMDISK (4)
$ ls /dev/nbd0p*
/dev/nbd0p1 /dev/nbd0p2
$ sudo mount /dev/nbd0p2 /mnt (5)
$ sudo tar cf img.tar -C /mnt . (6)
$ sudo umount /mnt && sudo qemu-nbd -d /dev/nbd0 (7)
```

(1) 设置一个环境变量指向用户的虚拟机磁盘镜像

(2) 初始化一个qemu-nbd需要的内核模块

(3) 列出这块磁盘上可以挂载的分区号

(4) 将虚拟机的磁盘连接到一个虚拟设备节点

(5) 通过qemu-nbd把选择的分区挂载到/mnt

(6) 从/mnt创建出一个名为img.tar的TAR文件

(7) 卸载分区然后用qemu-nbd清理

选择一个分区 要选择挂载的分区的话，可以运行`sudo cfdisk /dev/nbd0`来查看可选项。注意，如果看到了LVM选项，也就意味着磁盘采用的不是普通的分区方案，关于如何挂载LVM分区，用户将需要做一些额外的调研。

如果是远程虚拟机，用户需要作出一个选择：要么关掉虚拟机然后让运维团队介入，转储分区文件，要么在虚拟机是运行的状态下为它创建一个TAR。

如果用户拿到的是分区转储文件，就可以很轻松地进行挂载，然后按照如下命令把它转换成一个TAR文件：

```
$ sudo mount -o loop partition.dump /mnt
$ sudo tar cf $(pwd)/img.tar -C /mnt .
$ sudo umount /mnt
```

另外，也可以选择从一个正在运行的系统创建出TAR文件。在登录到系统后可以轻松实现这一点：

```
$ cd /
$ sudo tar cf /img.tar --exclude=/img.tar --one-file-system /
```

至此，用户拿到了文件系统镜像的TAR，紧接着可以通过scp把它发送到其他机器。

状态中断的风险 从一个正在运行的系统创建出TAR看上去可能是最简单的方案（没有关机，不需要安装软件或者请求别的团队），

但是它也存在一些弊端——用户复制出来的文件可能存在状态不一致的情况，并且可能会在试用制作出来的新的Docker镜像时遇到一些莫名其妙的问题。如果只能这样做的话，那就尽可能多地停掉一些应用和服务。

一旦拿到了文件系统的TAR，便可以将其加到镜像里。这一过程再简单不过了，它是由一个两行代码的Dockerfile组成的：

```
FROM scratch
ADD img.tar /
```

现在可以执行`docker build .`，然后便能得到自己的镜像了！

从TAR导入 除了ADD，Docker还提供了一个替代的`docker import`形式的命令，可以使用`cat img.tar | docker import - new_image_name`来导入文件。然而，即便选择在构建镜像时附带一些额外的指令，用户仍然始终需要创建一个Dockerfile。因此，使用ADD命令可能会更简单一些，也可以轻松地看到镜像的历史。

现在，因为在Docker里已经有了一个镜像，就可以开始用它来做一些实验了。在本例中，用户可能会从基于新镜像创建出一个新的Dockerfile开始，通过剥离文件和软件包来实验。

一旦完成了这一点并且得到了满意的结果，紧接着便可以在运行中的容器上用`docker export`导出一个新的、更小巧的TAR，用户可以把它用作新一层镜像的基础，然后重复这一过程直到对得到的镜像满意为止。

图3-1中的流程图展示了这一过程。

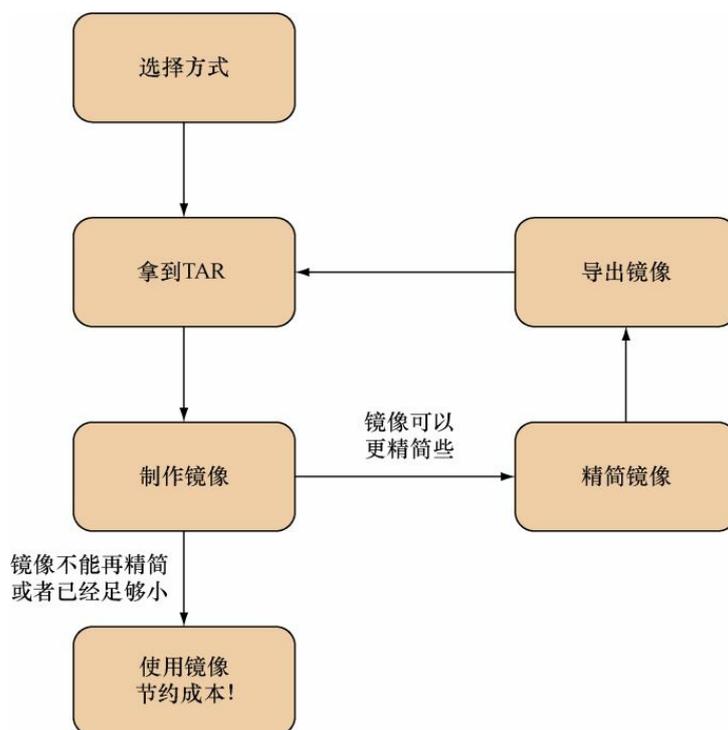


图3-1 容器的“瘦身”过程

技巧11 类宿主机容器

现在，我们将继续讨论一个在Docker社区里颇具争议的领域——运行一个从一开始就运行着多个进程的类宿主机镜像。

在Docker社区里，一部分人认为这是一个糟糕的做法。容器并不是虚拟机（有着显著差异），而且完全没办法假装说对此没有困惑或者问题。

好也罢，坏也罢，这一技巧将会展示该如何运行一个类宿主机镜像，然后讨论围绕着这种做法所带来的一些问题。

逐步引入 运行类宿主机镜像会是一个说服Docker反对派的好方法，告诉他们Docker是有用的。他们用得越多，对范式的理解就会越透彻，微服务方案对他们来说也就越有意义。在我们将Docker引入企业内部后，我们发现这种单体方式是一个很棒的切入点，它可

以推动人们从以前的在开发服务器及笔记本上开发转换到一个更包容和可管理的环境。由此，将Docker推广到测试、持续集成、托管环境以及DevOps工作流也就水到渠成了。

虚拟机和Docker容器之间的差异!

虚拟机和Docker容器之间存在以下一些不同之处。

- Docker 是面向应用的，而虚拟机是面向操作系统的。
- Docker容器和其他容器共享同一个操作系统。相反，每个虚拟机都有一个由hypervisor管理的它们自己的操作系统。
- Docker容器被设计成只运行一个主要进程，而不是管理多组进程。

问题

想要自己的容器就像一个正常的宿主机环境，可以运行多个进程和服务。

解决方案

使用一个被设计成模拟一台宿主机的镜像，然后置备它时包含需要的应用程序。

讨论

这里我们打算使用phusion/baseimage Docker镜像，一个被设计成运行多个进程的镜像。

第一步就是启动该镜像然后使用docker exec连到它里面：

```
user@docker-host$ docker run -d phusion/baseimage (1)
3c3f8e3fb05d795edf9d791969b21f7f73e99eb1926a6e3d5ed9e1e52d0b446e (2)
user@docker-host$ docker exec -i -t 3c3f8e3fb05d795 /bin/bash (3)
```

```
root@3c3f8e3fb05d:/# (4)
```

(1) ❶

(2) ❷

(3) ❸

(4) ❹

在上述代码中，`docker run`将会在后台启动该镜像❶，执行该镜像默认的启动命令，然后返回新创建的容器的ID❷。

然后将这个容器的ID传给`docker exec`❸，该命令会在这个已经运行的容器内部启动一个新的进程。`-i`标志代表着可以和新进程交互，而`-t`则意味着想配置一个TTY，它允许在容器内部开启一个终端（`/bin/bash`）❹。

如果等待1 min，然后查看进程表，输出内容如代码清单3-1所示。

代码清单3-1 一个类宿主机容器里正在运行的进程

```
root@aba74d81c088:/# ps -ef (1)
UID    PID  PPID  C  STIME TTY      TIME CMD
root    1    0    0  13:33 ?        00:00:00 /usr/bin/python3 -u /sbin/my_init
(2)

root    7    0    0  13:33 ?        00:00:00 /bin/bash (3)
root   111    1    0  13:33 ?        00:00:00 /usr/bin/runsvdir -P /etc/service
(4)
root   112   111    0  13:33 ?        00:00:00 runsv cron (5)
root   113   111    0  13:33 ?        00:00:00 runsv sshd
root   114   111    0  13:33 ?        00:00:00 runsv syslog-ng
root   115   112    0  13:33 ?        00:00:00 /usr/sbin/cron -f
root   116   114    0  13:33 ?        00:00:00 syslog-ng -F -p
      /var/run/syslog-ng.pid --no-caps
root   117   113    0  13:33 ?        00:00:00 /usr/sbin/sshd -D
```

```
root 125 7 0 13:38 ? 00:00:00 ps -ef (6)
```

- (1) 执行ps命令列出所有正在运行的进程
- (2) 一个简单的init进程，设计用来运行所有其他服务
- (3) bash进程由docker exec启动，并且当作shell使用
- (4) runsvdir运行所有在 /etc/service目录里定义的服务
- (5) 通过runsv命令在这里启动3个标准服务（cron、sshd和syslog）
- (6) 当前执行的ps命令

可以看到，容器的启动过程很像一台宿主机，初始化一些像cron和sshd这样的服务使它看上去和一台标准的Linux主机没什么两样。这作为对那些新入门的Docker工程师的最初演示时非常有用。

至于这样做是否违反了微服务的“一个容器一个服务”的原则，在Docker社区里是一个见仁见智的问题。类宿主机镜像方案的支持者认为这样做并没有违背该原则，因为容器仍然可以满足为里面运行的系统提供单一的离散功能的需求。

技巧12 将一个系统拆成微服务容器

我们已经探讨了该如何把一个容器用作一个单体实体（像传统的服务器那样），并且阐明这会是一个快速将一个系统架构迁移到Docker上的好方法。然而，在Docker的世界里，公认的最佳实践是尽可能多地把系统拆分开，直到在每个容器上都只运行一个“服务”，并且所有容器都通过链接互相连通。由于这是Docker官方推荐的做法，

因此读者会发现从Docker Hub上获取的绝大多数容器都遵循这个方案。而理解该如何以这种方式构建镜像，对与Docker生态系统的其他组件交互也尤为重要。

使用一个容器一个服务的主要原因在于可以更容易通过单一职责原则（single responsibility principle）实现关注点分离（separation of concerns）^[1]。如果用户的容器执行的是单一任务，那么可以很方便地把该容器应用到从开发、测试到生产的整个软件开发生命周期里，而无须太担心它与其他组件的交互问题。这就使软件项目可以更敏捷地交付并且具备更好的扩展性。但是，它的确带来了一些管理上的负担，因此，最好思量一下在自己的用例场景下这样做是否真的值得。

暂且不论哪种方案更适合，最佳实践方法至少拥有一个明显的优势——正如所见，在使用Dockerfile时实验和重新构建都比前一套方案快上不少。

问题

想将应用程序拆分为各个单独的且更易于管理的服务。

解决方案

使用Docker将应用程序拆分并堆砌出多个基于容器的服务。

讨论

在Docker社区里，关于应当怎样严格遵守“一个服务一个容器”的规则方面还存在着一些争议，这其中部分源自在定义方面的一些异议——它是一个单个的进程，还是说可以是结合在一起共同满足一个需求的一组进程？最终，它往往会被归结为这样一种说法，即赋予从头开始重新设计系统的能力，微服务也许会是大多数人的选择。但是有

时候实用主义可能会战胜理想主义——当为组织评估Docker时，为了能够让Docker尽可能更快、更容易地用起来，我们发现在当时的处境下只能选择单体这条路。

让我们一起来看看在Docker内部运行单体应用的其中一个具体弊端。正如代码清单3-2中所列的那样，我们需要先展示的是如何构建一个拥有数据库、应用程序以及Web服务器的单体应用。

简化版**Dockerfile** 这些例子只是用于教学目的，并且已经被相应简化。尝试直接运行它们不一定能正常工作。

代码清单3-2 配置一个简单的PostgreSQL、NodeJS和Nginx应用

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql nodejs npm nginx
WORKDIR /opt
COPY . /opt/
RUN service postgresql start && \
    cat db/schema.sql | psql && \
    service postgresql stop
RUN cd app && npm install
RUN cp conf/mysite /etc/nginx/sites-available/ && \
    cd /etc/nginx/sites-enabled && \
    ln -s ../sites-available/mysite
```

何时在**RUN**语句里使用命令链 在**RUN**语句里使用**&&**能够有效确保这些命令作为一条命令执行。这一点非常有用，因为它可以保持镜像不至于太大。每条Dockerfile命令都会在之前的基础上创建一个单一的新镜像层。如果以这种方式执行一个软件包更新的命令，如**apt-get update**这样的安装命令，用户便能够确保无论软件包是何时安装的，它们都将是来源于一个已经更新过的包缓存。

前面的例子是一个简单的概念版的Dockerfile，它会在容器里安

装一切需要的软件，并随后配置好数据库、应用程序和Web服务器。但是，如果想快速地重新构建容器的话就有问题了——仓库下的任何文件的任意改动均会造成一切事物从{*}开始重新构建，因为这种情况下无法复用之前的缓存。如果存在一些执行较慢的步骤（数据库的创建或者npminstall）的话可能就得在容器重新构建的时候等待一段时间。

针对这个问题的解决方案便是将COPY . /opt/指令拆到应用（数据库、应用程序和Web配置）的各个部分：

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql nodejs npm nginx
WORKDIR /opt
COPY db /opt/db                                +-
RUN service postgresql start && \             |- (1)
    cat db/schema.sql | psql && \             |
    service postgresql stop                  +-
COPY app /opt/app                               +-
RUN cd app && npm install                       |- (2)
RUN cd app && ./minify_static.sh                +-
COPY conf /opt/conf                             +-
RUN cp conf/mysite /etc/nginx/sites-available/ && \ +
    cd /etc/nginx/sites-enabled && \         |- (3)
    ln -s ../sites-available/mysite         +-

```

(1) 设置db

(2) 设置app

(3) 设置Web

在前面的代码里，COPY命令被分成两个单独的指令。由于可以利用缓存复用在修改代码之前未经变更的交付文件，这就意味着数据库不会在每次代码更改的时候都重新构建。

但是，由于缓存功能是相当简单粗糙的，容器仍然不得不在每次

对schema脚本做出更改时完全地重新构建。解决这一问题的唯一途径便是抛弃原有顺序配置的步骤，创建多份Dockerfile，内容如代码清单3-3、代码清单3-4和代码清单3-5所示。

代码清单3-3 数据库 Dockerfile

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql
WORKDIR /opt
COPY db /opt/db
RUN service postgresql start && \
    cat db/schema.sql | psql && \
    service postgresql stop
```

代码清单3-4 应用程序 Dockerfile

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nodejs npm
WORKDIR /opt
COPY app /opt/app
RUN cd app && npm install
RUN cd app && ./minify_static.sh
```

代码清单3-5 Web服务器 Dockerfile

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nginx
WORKDIR /opt
COPY conf /opt/conf
RUN cp conf/mysite /etc/nginx/sites-available/ && \
    cd /etc/nginx/sites-enabled && \
    ln -s ../sites-available/mysite
```

每当db、app或者conf文件夹下的内容有一个发生变化，将会只有一个容器需要被重新构建。当有超过3个以上的容器又或者有一些对时间敏感的配置步骤时，这样做会特别有用——只要花上一些心思，在每个步骤中添加最低限度的所需文件，结果便是可以最大程度地利用Dockerfile的缓存机制。在应用的Dockerfile（代码清单3-4）里，`npm install`操作定义在了一个单独的文件package.json里，因

此我们可以通过修改我们的Dockerfile以利用dockefile本身的镜像层缓存机制，并且只需要在必要的时候才去重新构建缓慢的npm install，如代码清单3-6所示。

代码清单3-6 重新排序的Dockerfile，更早执行npm install

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nodejs npm
WORKDIR /opt
COPY app/package.json /opt/app/package.json
RUN cd app && npm install
COPY app /opt/app
RUN cd app && ./minify_static.sh
```

但是，天下没有免费的午餐——我们必须将一个简单的Dockerfile转换为多个重复的Dockerfile。我们可以通过添加另外一个Dockerfile当作自己的基础镜像来解决部分问题，但是其他这样重复的情况并不少见。此外，现在启动镜像时又会冒出一些新的复杂性——除了在EXPOSE步骤让一些合适的端口可以用于链接以及修改Postgres配置外，我们还需要确保在自己每次启动时链接到各个容器。幸运的是，已经有这样的一款工具，它叫作docker-compose（前身是fig），我们将会技巧68里介绍它。

3.2 管理容器的服务

Docker官方文档中清楚地表达了Docker容器并不是虚拟机。Docker容器和虚拟机之间一个关键的区别就是，容器是被设计成运行单个进程的。当该进程结束时，容器便会退出。这就是它和一台Linux虚拟机（或者任意一个Linux操作系统）的不同之处，它没有init进程。

init进程在Linux操作系统上是以进程ID为1并且父进程ID为0的形

式运行的。这个init进程可能会被叫作“init”或者“systemd”。无论它叫什么，它的职责都是承担运行在该操作系统上的所有其他进程的维护工作。

如果开始实验Docker的话，用户可能会发现自己仍然有启动多个进程的需求。例如，用户可能会想要运行一些cron作业来收拾本地应用的日志文件，又或者在容器里配置一个内部的memcached服务器。如果选择走这条路的话，那么可能最终需要编写一个shell脚本来管理这些子进程的启动。实际上，用户将会效仿init进程的做法。别这么干！进程管理中的许多问题之前都已经被其他人遇到过了，并且已经在预打包系统里解决了。

技巧13 管理容器内服务的启动

当尝试将Docker用作虚拟机的替代品时，选择在容器里运行多个服务可能会是一个方便之举，或者说这可能是在最初将虚拟机转换成容器后要运行重要服务时的必要之举。

无论原因是什么，重要的是要避免尝试在容器里管理进程时重复造轮子。

问题

想要在一个容器里管理多个进程。

解决方案

使用Supervisor应用（<http://supervisord.org/>）来管理进程的启动。

讨论

我们打算展示一下如何置备带有Tomcat和一个Apache Web服务器的容器，并且以Supervisor托管的方式启动并运行它。

首先，如代码清单3-7所示，在一个新的空目录里创建Dockerfile。

代码清单3-7 Supervisor示例Dockerfile

```
FROM ubuntu:14.04 (1)
RUN apt-get update && apt-get install -y python-pip apache2 tomcat7 (2)
ENV DEBIAN_FRONTEND noninteractive (3)
RUN pip install supervisor (4)
RUN mkdir -p /var/lock/apache2 (5)
RUN mkdir -p /var/run/apache2
RUN mkdir -p /var/log/tomcat
RUN echo_supervisord_conf > /etc/supervisord.conf (6)
ADD ./supervisord_add.conf /tmp/supervisord_add.conf (7)
RUN cat /tmp/supervisord_add.conf >> /etc/ (8)
  └─> supervisord.conf
RUN rm /tmp/supervisord_add.conf (9)
CMD ["supervisord", "-c", "/etc/supervisord.conf"] (10)
```

- (1) 从ubuntu:14.04开始
- (2) 安装python- pip（用来安装Supervisor）、apache2和tomcat7
- (3) 设置一个环境变量，表明此会话是非交互式的
- (4) 通过pip安装Supervisor
- (5) 创建一些运行应用所需的维护目录
- (6) 利用echo_sup- ervisord_conf工具创建一个默认的supervisord配置文件
- (7) 将Apache和Tomcat的supervisord配置设定复制到镜像里，

做好加到默认配置的准备

(8) 将Apache和Tomcat的supervisord配置设定追加到supervisord的配置文件里

(9) 由于不再有用处了，删除之前上传的文件

(10) 现在，只需要在容器启动时运行Supervisor即可

还将需要配置Supervisor，指示它需要启动哪些应用，如代码清单3-8所示。

代码清单3-8 supervisord_add.conf

```
[supervisord] (1)
nodaemon=true (2)

# apache
[program:apache2] (3)
command=/bin/bash -c "source /etc/apache2/envvars && (4)
  exec /usr/sbin/apache2 -DFOREGROUND"

# tomcat
[program:tomcat]
command=service start tomcat
redirect_stderr=true
stdout_logfile=/var/log/tomcat/supervisor.log (5)
stderr_logfile=/var/log/tomcat/supervisor.error_log
```

(1) 为supervisord声明全局配置块

(2) 设置成不要后台运行Supervisor进程，因为对容器来说它是一个前台进程

(3) 声明新程序的代码块

(4) 用于启动在该代码块中声明的程序的命令

(5) 配置相关日志

由于用的是Dockerfile，因此可以借助标准的单个Docker命令来构建镜像：

```
docker build -t supervised .
```

现在可以运行构建好的镜像了！

```
$ docker run -p 9000:80 --name supervised supervised (1)
2015-02-06 10:42:20,336 CRIT Supervisor running as root (no user in con
fig file) (2)
2015-02-06 10:42:20,344 INFO RPC interface 'supervisor' initialized
2015-02-06 10:42:20,344 CRIT Server 'unix_http_server' running
  ➤ without any HTTP authentication checking
2015-02-06 10:42:20,344 INFO supervisord started with pid 1 (3)

2015-02-06 10:42:21,346 INFO spawned: 'tomcat' with pid 12
2015-02-06 10:42:21,348 INFO spawned: 'apache2' with pid 13
2015-02-06 10:42:21,368 INFO reaped unknown pid 29
2015-02-06 10:42:21,403 INFO reaped unknown pid 30
2015-02-06 10:42:22,404 INFO success: tomcat entered RUNNING state, (4
)
  ➤ process has stayed up for > than 1 seconds (startsecs)
2015-02-06 10:42:22,404 INFO success: apache2 entered RUNNING state,
  ➤ process has stayed up for > than 1 seconds (startsecs)
```

(1) 将容器的80端口映射到宿主机上的9000端口，给容器分配一个名字，然后指定要运行的镜像名称，即之前构建命令标记的那个

(2) 启动Supervisor进程

(3) 启动被托管的进程

(4) 被托管的进程被Supervisor识别为已经成功启动

如果访问<http://localhost:9000>，应该就能看到启动的Apache服务器的默认页面。

要清理容器，可以执行如下命令：

```
docker rm -f supervised
```

如果对Supervisor的其他替代品有兴趣，还有一个runit，技巧11里介绍过的Phusion基础镜像用到了它。

3.3 保存和还原工作成果

有些人说代码直到提交到了源代码管理中才算编写完成，对容器来说又何尝不是这样。如果使用虚拟机可以借助快照来保存现有状态，但是Docker采取的是一个更为积极的方案，鼓励保存和复用已有的工作成果。

我们将介绍在开发中“保存游戏”的方式、打标签（tagging）技术的一些具体细节、Docker Hub的使用，以及如何在构建时指向特定镜像。由于这些操作被认为是非常基础的，因而Docker将它们打造得相对简单和快捷。

尽管如此，对于Docker新手来说这仍然是一个令人困惑的主题，因此在这一节中，我们将带领读者一步步更加全面地了解与这个主题相关的内容。

技巧14 在开发中“保存游戏”的方式

如果曾经开发过任意类型的软件，那么你应该不止一次抱怨过：“我敢肯定在此之前它运行得好好的！”也许原话还没有这么淡定。由于无法将系统还原到一个已知的良好（或者也许只有“更好”）状态，就只能赶紧强行修改（hack）代码以赶上最后期限或者修复漏洞，这也是许多人敲碎键盘累死累活的原因。

源代码管理极大地改善了这一点，然而在特殊情况下还是存在以下两个问题：

- 源代码仓库可能无法反映出“工作”环境下文件系统的状态；
- 暂时还不想把代码提交上去。

第一个问题比第二个更明显一些。尽管像Git这样的现代化源代码管理工具可以轻松地创建出一次性的本地分支，但是抓取整个开发环境文件系统的状态并不是源代码管理的初衷。

Docker通过它的提交（commit）功能提供了一个廉价、快捷的方式来保存容器的开发环境文件系统状态，而这正是接下来要探讨的内容。

问题

想要保存开发环境的状态。

解决方案

使用docker commit来保存状态。

讨论

我们不妨假设用户想要修改第1章里的to-do应用。ToDo公司的CEO对这个应用不是很满意，并且想把浏览器上显示的标题从“Swarm+React - TodoMVC”改为“ToDoCorp’s ToDo App”。

用户拿不准要怎么实现这一点，也许会想要先把应用运行起来，然后通过修改文件来进行实验，看看到底会发生什么：

```
$ docker run -d -p 8000:8000 --name todobug1 dockerinpractice/todoapp  
(1)  
3c3d5d3ffd70d17e7e47e90801af7d12d6fc0b8b14a8b33131fc708423ee4372
```

```
$ docker exec -i -t todobug1 /bin/bash (2)
```

(1) ❶

(2) ❷

上述`docker run`命令 ❶ 在一个容器里以后台模式 (`-d`) 启动了 `to-do` 应用，将容器的8000端口映射到了宿主机上的8000端口 (`-p 8000:8000`)，为方便引用起见，将其命名为 `todobug1` (`--name todobug1`)，然后返回了该容器的ID。该容器启动时执行的命令默认会是我们构建的 `dockerinpractice/todoapp` 镜像在构建时指定的命令，该镜像也可以在 Docker Hub 上找到。

第二条命令 ❷ 将会在正在运行的容器里启动 `/bin/bash`。这里用到的是名为 `todobug1` 的容器，不过用户也可以使用其原本的容器ID。`-i` 意味着这条 `exec` 命令以交互模式运行，而 `-t` 确保 `exec` 将会按照一个终端预期的那样工作。

如今我们已经在容器里了，那么，实验的第一步便是安装一个编辑器。我们更喜欢 `vim`，所以采用了如下命令：

```
apt-get update  
apt-get install vim
```

小经波折之后我们意识到需要修改的文件是 `local.html`。因此，我们将该文件的第5行改成了如下内容：

```
<title>ToDoCorp's ToDo App</title>
```

但CEO的意思可能是希望标题都是小写的，因为她听说这样看上去会更时尚些。这两种方式我们都想准备好，所以我们选择先提交现

有成果。在另外一个终端下运行如下命令：

```
$ docker commit todobug1 (1)
ca76b45144f2cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970 (2)
```

(1) 把之前创建出来的容器转成镜像

(2) 刚提交的容器的新镜像ID

如今已经将容器提交成镜像，并且可以在之后运行它。

状态是无法捕获的！提交一个容器只会保存在提交那个时刻容器的文件系统的状态，而不是进程。记住，Docker容器不是虚拟机。如果环境的状态依赖于一些正在运行的进程的状态，而这些进程不能通过标准文件恢复的话，这个技巧将无法帮助用户保存所需的那个状态。在这种情况下，用户也许得想办法看怎样才能恢复开发环境里进程的状态。

紧接着，将local.html的内容修改成另外一个可能要求的值：

```
<title>todocorp's todo app</title>
```

然后再次提交：

```
$ docker commit todobug1
071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036
```

现在拥有两个镜像的ID（在这个演示里是ca76b45144f2cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970和071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036但是读者的可能会不一样），代表两种选择。当CEO来评估其想要的

方案时，可以把两个镜像都运行起来，然后让她决定要提交哪一个。

可以通过打开新的终端然后执行如下命令来实现这一点：

```
$ docker run -p 8001:8000 \  
ca76b45144f2cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970 (1)  
$ docker run -p 8002:8000 \  
071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036 (2)
```

(1) 将容器的8000端口映射到宿主机的8001端口，然后指定小写的那个镜像的ID

(2) 将容器的8000端口映射到宿主机的8002端口，然后指定大写的那个镜像的ID

这样一来，便可以在<http://local-host:8001>上展示大写的方案，在<http://localhost:8002>上呈现小写的方案。

毋庸置疑，比起一长串随机的字符串，肯定还有更好的办法来引用镜像。下面一个技巧将会关注如何给这些镜像指定一个名称，以便能够更加轻松地引用它们。

我们发现，当我们需要通过一系列棘手的命令来配置应用时这会是一个很有用的技巧。提交该容器时，一旦成功的话，它会记录我们的bash会话的历史，这也意味着通过一系列步骤重新恢复系统的状态成为可能。这可以节省大量的时间！而且，这在正实验一个新功能而又不太确定是否完工，或者当重现出一个漏洞然后想尽量确保能够回到那个挂掉的状态时，也很有用处。

外部依赖是无法捕获的！对容器的任何外部的依赖（如数据库、Docker卷或者其他被调用的服务）在提交时均不会被保存。当前介绍的这个技巧没有任何外部的依赖，因此我们也无须担心这一点。

技巧15 给Docker打标签

现在，用户已经通过提交容器保存了容器的状态，并且还得到了一个代表镜像ID的随机字符串。很显然，记住和管理这些包含大量数字的镜像ID是非常困难的。如果能够利用Docker的打标签功能给这些镜像赋予一些可读的名称（和标签）的话那就太给力了，它还能提醒用户为什么创建这些镜像。

掌握这一技巧可以对镜像的用途一目了然，使机器上的镜像管理变得简单很多。

问题

想要方便地引用并且保存一次Docker提交。

解决方案

使用docker tag给这次提交命名。

讨论

打标签功能的基本用法是非常简单的：

```
$ docker tag \ (1)
071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036 \ (2)
)
imagename (3)
```

(1) docker tag命令

(2) 想要命名的镜像ID

(3) 想给镜像起的名字

上述操作就是给镜像命名，可以用这个名字来引用该镜像，例如：

```
docker run imagename
```

这可比记住大量包含字母和数字的随机字符串轻松多了！

如果想和别人分享镜像，除了设置标签还有其他事情要做。遗憾的是，与标签相关的术语可以说相当混乱。像镜像（`image`）、名称（`name`）和仓库（`repository`）这些术语在使用时很容易混淆。表3-1给出了这些术语的一些定义。

表3-1 Docker标签的术语

术语	含义
镜像	一个只读层
名称	镜像的名字，如“ <code>todoapp</code> ”
标签	作为动词的话，它指的是给一个镜像起名字。作为名词的话，它是镜像名的一个修饰词
仓库	一组托管的打好标签的镜像，它们一起为容器创建相应的文件系统

在这个表里面最容易混淆的术语恐怕还是镜像和仓库。一直以来我们使用的术语镜像其实可以简单地理解成是一组我们从一个容器划分出来的多个分层，但是从技术上来说，一个镜像即是一个递归地指向它的父镜像层的分层。而仓库则是受托管的，这意味着它会被存放

在某个地方（可以是在Docker守护进程，也可以是在一个注册中心）。此外，仓库是一组打好标签的镜像，它们共同组成了容器的文件系统。

这里用Git来进行类比可能有助于理解。当克隆一个Git仓库时，可以签出（check out）所要求的那个点的文件的状态。这一点可以与镜像类比。该仓库保存了每一次提交时文件的整个历史，可以据此追溯到最初的那次提交。因此，在最上面一层签出仓库时，其他的分层（或者提交）就都在克隆的那个仓库里了。

在实践中，镜像和仓库这两个术语或多或少有些混用，因此不必太担心这一点。但要注意的是，这些术语都是存在的，并且用起来也的确很相似。

到目前为止所看到的内容都是如何给一个镜像ID命名的。令人不解的是，这个名字并不是该镜像的标签，虽说人们经常这样提到它。我们可以根据打标签（动词）和给镜像起名字的那个标签（名词）的行为来区分二者。这个标签（名词）允许用户给镜像的一个特定版本命名。用户可以通过添加标签来管理同一镜像不同版本的引用。例如，可以把一个版本名称或者提交日期当作标签。

带有多个标签的仓库的一个很好的例子便是Ubuntu镜像。如果拉取Ubuntu镜像，然后运行docker images，就会得到与代码清单清单3-9所示类似的输出结果。

代码清单3-9 带有多个标签的镜像

```
$ docker images
REPOSITORY TAG          IMAGE ID      CREATED      VIRTUAL SIZE
ubuntu     trusty         8eaa4ff06b53 4 weeks ago 192.7 MB
ubuntu     14.04         8eaa4ff06b53 4 weeks ago 192.7 MB
ubuntu     14.04.1       8eaa4ff06b53 4 weeks ago 192.7 MB
```

上面REPOSITORY一列列出了一组托管的叫作“ubuntu”的分层。通常这指的便是镜像。这里的TAG一列列出了4个不同的名称（`trusty`、`14.04`、`14.04.1`和`latest`）。IMAGE ID一列列出了几个完全一致的镜像ID。这是因为这些打上不同标签的镜像都是同一个。

这表明用户可以拥有一个带有多个标签的相同镜像ID的仓库。尽管从理论上讲这些标签以后也可以指向不同的镜像ID。例如，如果“`trusty`”有一个安全更新的话，维护人员一次新的提交可能就会改变镜像ID，然后打上新的“`trusty`”“`14.04.2`”“`latest`”标签。

如果没有指定标签的话默认会给镜像打上“`latest`”标签。

“latest”标签的含义 “latest”标签在Docker里并没有什么特殊的含义——它只是在打标签和拉取镜像时的一个默认值。这并不意味着它是这个镜像设定的最后一个标签。镜像的“latest”标签也可能指向的是该镜像的一个老版本，因为这之后构建的版本可能会被打上一个特定的标签，如“`v1.2.3`”。

技巧16 在Docker Hub上分享镜像

如果能和其他人分享这些名字（和镜像）的话，给镜像打标签的时候用一些带描述的名字可能会更有帮助。为了满足这一需求，Docker提供了轻松将镜像迁移到其他地方的能力，且Docker公司也创建了Docker Hub这一免费服务以鼓励这样的分享。

Docker Hub所需的账号 要利用本技巧的话，用户将需要一个Docker Hub账号，并且它已经在宿主机上通过运行`docker login`

登录过。如果还没有配置这样的一个账号，可以到 <http://hub.docker.com> 上面创建一个。只需要照着注册说明的指示去做即可。

问题

想要公开分享一个Docker镜像。

解决方案

使用Docker Hub注册中心（registry）来分享镜像。

讨论

当讨论标签时，注册中心相关的各种术语可能容易让人混淆。表3-2应该有助于理解这些术语该如何使用。

表3-2 Docker注册中心的术语

术 语	含 义
用户名（user name）	Docker注册中心上的用户名
注册中心（registry）	注册中心持有镜像。一个注册中心就是一个可以上传镜像或者从这里下载镜像的存储。注册中心可以是公开的，也可以是私有的
注册中心宿主机（registry host）	运行Docker注册中心的宿主机
Docker Hub	托管在 https://hub.docker.com 上公开默认的注册中心

索引 (index)	与注册中心宿主机含义相同，这似乎是一个过时的术语
------------	--------------------------

正如之前所见，只要喜欢，用户可以给一个镜像打上多个标签。这对复制过来的镜像很有用，如此一来用户便拥有了它的控制权。

例如，假设用户在Docker Hub上的用户名是adev。代码清单3-10中的3条命令展示了怎样从Docker Hub上将debian:wheezy镜像复制到自己的账号下。

代码清单3-10 将一个公用镜像复制和推送到adev的Docker Hub账号

```
docker pull debian:wheezy (1)
docker tag debian:wheezy adev/debian:mywheezy1 (2)
docker push adev/debian:mywheezy1 (3)
```

(1) 从Docker Hub上拉取debian镜像

(2) 给wheezy镜像打上自己的用户名 (adev) 并且指定一个标签 (mywheezy1)

(3) 推送新创建的标签

至此，用户已经得到了一个下载好的Debian wheezy镜像的引用，可以维护、关联或者以它为基础构建其他镜像。

如果有可以推送的私有仓库，除了必须在标签的前面指定注册中心的地址外，其他流程是完全一致的。假设有一个仓库放在 <http://mycorp.private.dockerregistry>。代码清单3-11中列出的命令将会为镜像打上标签然后推送到该注册中心。

代码清单3-11 将一个公用镜像复制并推送到adev的私有注册中心

```
docker pull debian (1)
```

```
docker tag debian:wheezy \
mycorp.private.dockerregistry/adev/debian:mywheezy1 (2)
docker push mycorp.private.dockerregistry/adev/debian:mywheezy1 (3)
```

(1) 从Docker Hub上拉取Debian镜像

(2) 用注册中心（mycorp.private.dockerregistry）、用户名（adev）和标签（mywheezy1）给wheezy镜像打上标签

(3) 将新创建的标签推送到私有注册中心。需要注意的是，在打标签和推送时都必须指定私有注册中心服务器的地址，这样一来Docker才能确保把它推送到正确的位置

上述命令将不会把镜像推送到公有的Docker Hub上，而是会把它推送到私有的仓库里，因此任何人只要可以访问该服务上的资源就能够拉取它。

至此，用户已经具备了和其他人分享镜像的能力。这是一个同其他工程师分享工作成果、想法甚至于遇到的一些问题的绝佳办法。

技巧17 在构建时指向特定的镜像

在构建过程中绝大部分时间里所引用的将是一些通用的镜像名，如“node”或者“ubuntu”，而且这些用起来可能不会有问题。

如果要引用一个镜像名的话，该镜像有可能会在标签保持不变的情况下发生变化。尽管听起来很荒谬，但是的确是这样的！仓库的名字只是一个引用，而它所指向的底层镜像可能会变成不同的。用冒号指定一个标签（如ubuntu:trusty）也没办法消除这一风险，因为像一些安全更新就可以用相同的标签自动地重新构建易受攻击的镜像。

绝大多数时候用户可能是希望这样的——镜像的维护人员可能找

到了一个改进以及修补安全漏洞的方法，这通常是一件好事。不过有时候这也会是一个痛点。而这不只是一个理论上的风险：在一些场合下已经发生了这样的事情，它以一种难以调试的方式破坏了持续交付的构建。在使用Docker的初期，那些最受欢迎的镜像会定期地添加和删除软件包（这里面还包括一个令人难忘的回忆，`passwd`命令居然消失了！），造成之前还正常工作的构建突然崩掉。

问题

想要确保是从一个特定的未做更改的镜像构建。

解决方案

从一个特定的镜像ID构建。

讨论

当想要绝对地确定构建时使用的是给定的文件时，可以在Dockerfile里指定一个特定的镜像ID。

下面是一个例子（可能在读者的环境里无法正常工作）：

```
FROM 8eaa4ff06b53 (1)
RUN echo "Built from image id:" > /etc/buildinfo (2)
RUN echo "8eaa4ff06b53" >> /etc/buildinfo
RUN echo "an ubuntu 14.4.01 image" >> /etc/buildinfo
CMD ["echo", "/etc/buildinfo"] (3)
```

(1) 从一个指定的镜像（或者分层）ID构建

(2) 在这个镜像里运行一个命令，把构建时引用的镜像记录到新镜像的一个文件里

(3) 构建的镜像默认会输出记录到/etc/buildinfo文件里的信息

要像这样从一个特定的镜像（或者分层）ID构建的话，镜像ID就必须存储到Docker守护进程本地。Docker注册中心将不会执行任何类型的查找操作来找出Docker Hub上可用镜像的各个分层的镜像ID，也不会可能在可能配置使用的任何其他注册中心上这样做。

值得一提的是，用户指向的那个镜像是不需要打过标签的——它可以是本地的任意一个镜像分层。用户可以从希望的任何分层开始构建。这在做某些调查或者一些实验性步骤而想要完成Dockerfile的构建分析时也许有用。

如果想远程持久化镜像，那么最好是给该镜像打上标签，然后将它推送到远程注册中心里一个受控制的仓库下。

Docker镜像可能会停止工作 值得指出的是，绝大部分要面对的问题都发生在一个之前还工作的Docker镜像突然间就不工作了。通常这是因为一些东西在网络层面有所变动。这其中一个记忆犹新的例子便是，某个早上我们的构建因为`apt-get update`失败。我们假定这是本地deb缓存的问题，然后尝试调试但是没有成功，直到一位可爱的系统管理员指出我们正在构建的这个特定版本的Ubuntu已经不再被支持了。这意味着`apt-get update`的网络调用都在返回HTTP错误。

3.4 进程即环境

看待Docker的视角之一是把它看作将环境变成各个进程的工具。此外，虽说虚拟机同样也可以这样对待，但Docker使这件事情变得更加便捷、高效。

为了说明这一点，我们将展示应该怎样加速启动、存储和重建容

器状态，这可以做成一些以其他方式（几乎）很难办到的事情——在2048上夺冠！

技巧18 在开发中“保存游戏”的方式

本技巧旨在提供一点儿轻松的调味剂，展示Docker可以怎样用来轻松地恢复状态。如果对2048不是很熟悉的话，不妨把它看作是一个容易上瘾的在板上推数字的游戏。如果想先熟悉一下它，在<http://gabrielecirulli.github.io/2048>有一个在线的最初版本。

问题

为了能够在需要的时候恢复到一个已知的状态，想要定期保存容器的状态。

解决方案

当不确定是否可以活下来时使用`docker commit`来“保存游戏”。

讨论

我们在此之前已经创建了一个单体镜像，用户可以在一个拥有VNC服务器以及Firefox的Docker容器里玩2048。

要使用这个镜像的话，用户需要安装一个VNC客户端。热门的实现方案有TigerVNC和VNC Viewer等。如果一个都没有的话，那么在宿主机上的包管理器里快速搜索关键字“vnc client”应该也能得到有用的结果。

要启动容器，可以执行代码清单3-12中列出的命令。

代码清单3-12 启动2048容器

```
$ docker run -d -p 5901:5901 -p 6080:6080 --name win2048 \
  imiell/win2048 (1)
$ vncviewer localhost:1 (2)
```

(1) ❶

(2) ❷

先从*imiell/win2048*镜像运行一个容器，这一步我们已经准备好了❶。我们在后台启动这个容器，然后指定它应该给宿主机开放两个端口（5901和6080）。在容器内部自动启动的VNC服务器将会使用这些端口，还给容器起了一个以后易于使用的名字——*win2048*。

现在可以运行VNC Viewer了（根据安装的情况可执行文件可能会不同），然后指示它连接到本地计算机❷。因为相应的端口已经从容器里暴露出来，连接到本地主机实际上也就是连接到容器。如果宿主机上除了一个标准的桌面外没有X显示，那么*localhost*后面的*:1*便是合理的——要是有的话，用户可能就得选择一个不同的数字，然后查阅下VNC Viewer的文档，将VNC端口手动指定为5901。

一旦连上了VNC服务器，它就会提示输入密码。这个镜像的VNC密码是*vncpass*。

我们会看到一个带有Firefox标签页的窗口和一个预先加载的2048的表格。点击它以获取焦点，然后玩到准备好保存游戏为止。

要保存游戏的话，得在提交它之后给这个命名好的容器打上一个小标签：

```
$ docker commit win2048 (1)
4ba15c8d337a0a4648884c691919b29891cbbe26cb709c0fde74db832a942083 (2)
$ docker tag 4ba15c8d337 my2048tag:$(date +%s)
```

(1) ❶

(2) ❷

在提交win2048容器❶后可以生成一个镜像ID❷，现在想给它赋予一个唯一的名字（因为可能创建了一堆这样的镜像）。为了做到这一点，我们将利用date +%s命令的输出作为镜像名称的一部分，该命令会输出一串从1970年的第一天开始算起的总秒数，提供一个唯一的（我们的目的）、不断增长的值。\$(command)语法只是在该位置将内容替换为整个命令的输出。如果愿意的话，也可以手动执行date +%s，取而代之的是，粘贴输出作为镜像名称的一部分。

然后可以继续玩下去，直到输了为止。现在该表演魔术了！我们可以通过代码清单3-13所示的命令返回到存档点。

代码清单3-13 返回到之前的游戏存档

```
$ docker rm -f win2048
$ docker run -d -p 5901:5901 -p 6080:6080 --name win2048 my2048tag:$mytag
```

\$mytag是在docker images命令里选出的一个标签。重复打标签、删除、运行这几个步骤，直到完成2048为止。

3.5 小结

在本章中，读者已经看到了在许多使用场景中Docker是怎样取代虚拟机的。虽然Docker并不是一个虚拟机技术，而且虚拟机和容器之间也有重大差别，但是Docker带来的便利的确可以加快开发过程。

本章在前面两章介绍的Docker架构的基础上，展示了它能够为我

们的工作带来多大的便利。

我们主要讨论了以下几点：

- 最开始迁移到Docker时选择将虚拟机转换成Docker镜像会相对轻松些；
- 可以在容器里管理服务，像以前虚拟机一样操作；
- 提交是一种正确的随手保存工作的方式；
- 可以给镜像命名，然后在Docker Hub上免费向世界分享它们。

采用新的模式开发常常会新增一些日常琐事和考验，在下一章里我们将探讨这其中一些较为重要的部分。

[1] 这是两个计算机领域的术语，具体见

https://en.wikipedia.org/wiki/Separation_of_concerns和

https://en.wikipedia.org/wiki/Single_responsibility_principle。——译者注

第4章 Docker日常

本章主要内容

- 使用和管理Docker卷以实现共享数据的持久化
- 学习第一个Docker模式：数据及开发工具容器
- 在Docker里使用GUI应用
- 操纵Docker构建缓存以实现快速可靠的构建
- 让Docker镜像系谱图可视化
- 从宿主机上直接让容器执行命令

当使用Docker开发软件时，读者会发现不断冒出来各种各样的需求。用户可能想法设法地要在一个容器里运行GUI应用，陷入对Dockerfile构建缓存的困惑，想要在使用的时候直接操纵容器，好奇镜像之间的系谱关系，想要从一个外部源引用数据，等等。

本章将带读者领略一些技巧，展示该怎样处理上述这些问题以及可能出现的其他问题。把它看成是Docker工具箱吧！

4.1 卷——持久化问题

容器是一个强大的概念，但是有时候并不是所有想访问的事物都能立马被封装成容器。用户可能有一个存储在大的集群上的相关Oracle数据库，想要连接它做一些测试。又或者，用户可能有一台遗留的大型服务器，而它上面现有配置好的二进制程序很难重现。

刚开始使用Docker时，用户想要访问的大部分事物可能会是容器

外部的数据和程序。我们将和读者一起从直接在宿主机上挂载文件转到更为复杂的容器模式：数据容器和开发工具容器。我们还将展示一些实用的技巧，例如，只需要一个SSH连接便能跨网络进行远程挂载，以及通过BitTorrent协议与其他用户分享数据。

卷是Docker的一个核心部分，有关外部数据引用的问题也是Docker生态系统中另一个快速变化的领域。

技巧19 Docker卷——持久化的问题

容器的大部分力量源自它们能够尽可能多地封装环境的文件系统的状态，这一点的确很有用处。

然而有时候用户并不想把文件放到容器里，而是想要在容器之间共享或者单独管理一些大文件。一个经典的例子便是想要容器访问一个大型的中央式数据库，但是又希望其他的（也许更传统些的）客户端也能和新容器一样访问它。

解决方案便是卷，一种Docker用来管理容器生命周期外的文件的机制。虽然这有悖于容器“部署在任何地方”的原则（例如，用户将无法在没有兼容数据库用来挂载的地方部署有数据库依赖的容器），但在实际的Docker使用中这仍然是一个很有用的功能。

问题

想要在容器里访问宿主机上的文件。

解决方案

使用Docker的卷标志，在容器里访问宿主机上的文件。

讨论

图4-1演示了使用卷标志和宿主机上的文件系统交互。

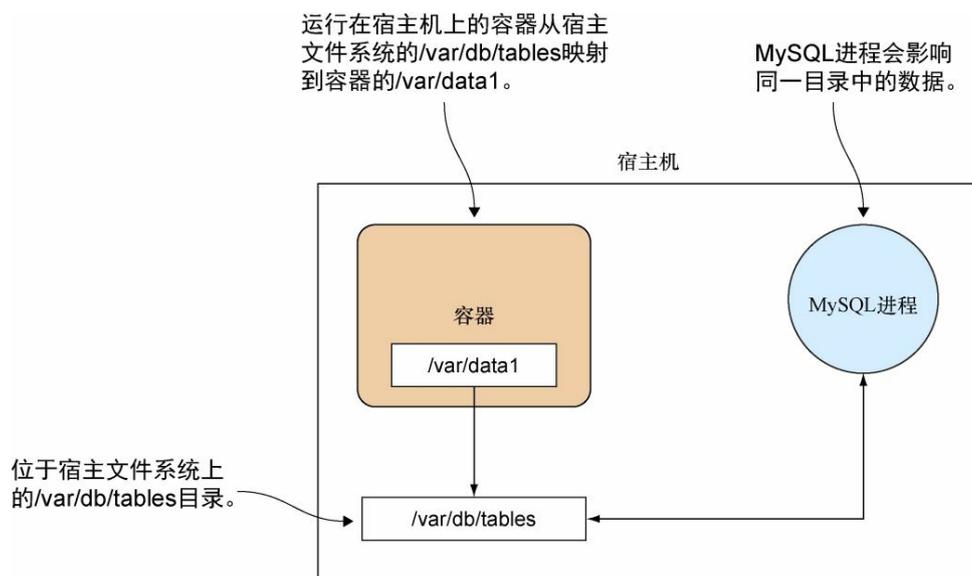


图4-1 容器里的卷

下面的命令展示宿主机上的/var/db/tables目录被挂载到了/var/data1，并且在图4-1里启动容器时会被执行：

```
$ docker run -v /var/db/tables:/var/data1 -it debian bash
```

-v标志（--volume的简写）表示为容器指定一个外部的卷。随后的参数以冒号分隔两个目录的形式给出了卷的格式，告知Docker将外部的/var/db/tables目录映射到容器里的/var/data1目录。如果外部目录和容器目录不存在均会被创建。

对已经存在的目录建立映射要小心。即使镜像中已经存在需要映射的目录，容器的目录也会被建立映射。这意味着在容器里映射的目录的原内容将会消失。如果试图映射一个关键目录，就会发生有趣的事情！例如，尝试挂载一个空目录到/bin。

还要注意的，卷在Dockerfile里被设定为不是持久化的。如果添加了一个卷，然后在一个Dockerfile里对该目录做了一些更改，这些

变动将不会被持久化到生成的目标镜像。

SELinux问题？ 如果宿主机运行SELinux，可能会遇到一些问题。如果SELinux的策略是enforced，容器可能无法写入/var/db/tables目录。用户将会看到一个“permission denied”错误。如果想解决这个问题，就需要联系系统管理员（如果有的话）或者关掉SELinux（仅用于开发目的）。可以查看技巧101来了解关于SELinux的更多内容。

技巧20 通过BitTorrent Sync的分布式卷

在一个团队里试用Docker时，开发人员可能希望能够在团队成员之间共享大量的数据，但是他可能又没有办法为共享服务器申请到足够容量的资源。对此最懒的办法便是在需要它们的时候从其他团队成员那里复制最新的文件——这对规模更大的团队而言很快便会失去控制。

解决办法便是用一个去中心化的工具来共享文件——无须专门的资源。

问题

想要在互联网上跨宿主机共享卷。

解决方案

使用BitTorrent Sync镜像来共享卷。

讨论

图4-2展示了达到这一目标所需的设置。

最终结果便是无须任何复杂配置便可以在互联网上方便地同步卷 (/data)。

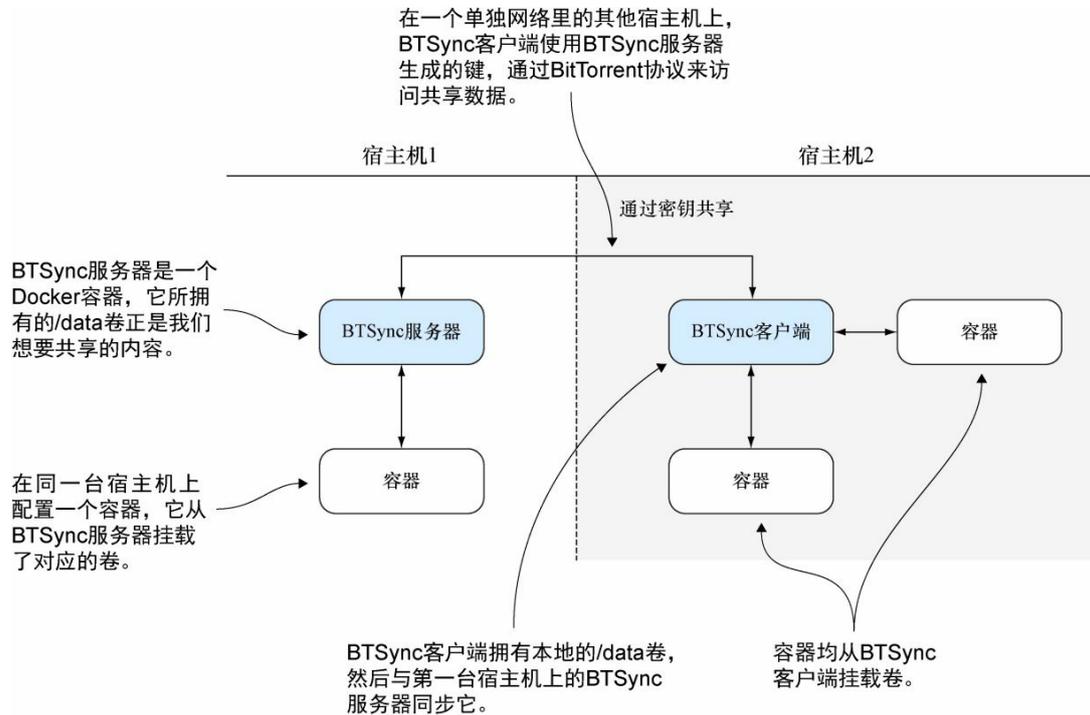


图4-2 使用BitTorrent Sync

在主服务器上，运行如下命令来配置第一台宿主机的容器：

```
[host1]$ docker run -d -p 8888:8888 -p 55555:55555 \ (1)
--name btsync ctlc/btsync
$ docker logs btsync (2)
Starting btsync with secret: \
ALSVEUABQQ5ILRS20QJKAOKCU5SIIP6A3 (3)
By using this application, you agree to our Privacy Policy and Terms.
http://www.bittorrent.com/legal/privacy
http://www.bittorrent.com/legal/terms-of-use
total physical memory 536870912 max disk cache 2097152
Using IP address 172.17.4.121

[host1]$ docker run -i -t --volumes-from btsync \ (4)
ubuntu /bin/bash
$ touch /data/shared_from_server_one (5)
$ ls /data
shared_from_server_one
```

(1) 运行已发布的ctlc/btsync镜像作为一个守护容器，称为

btsync，然后开放必要的端口

- (2) 获取btsync容器的输出,以便记录键的内容
- (3) 记下这个键——运行该容器的时候它可能会有所不同
- (4) 启动一个交互容器，然后挂载上btsync服务器的卷
- (5) 添加一个文件到/data卷

在次服务器上，开启一个终端并运行这些命令来同步卷：

```
[host2]$ docker run -d --name btsync-client -p 8888:8888 \
-p 55555:55555 \
ctlc/btsync ALSVEUABQQ5ILRS20QJKAOKCU5SIIP6A3 (1)
[host2]$ docker run -i -t --volumes-from btsync-client \
ubuntu bash (2)
$ ls /data
shared_from_server_one (3)
$ touch /data/shared_from_server_two (4)
$ ls /data
shared_from_server_one shared_from_server_two
```

(1) 启动一个btsync客户端容器作为守护进程，并且传递运行在host1上的守护进程生成的键

- (2) 启动一个交互容器，通过客户端守护进程挂载卷
- (3) 在host1上创建的该文件已经被传送到host2
- (4) 在host2上创建第二个文件

回到host1上运行的容器，应该能看到该文件正如第一个文件那样在宿主机之间被同步成完全一样的内容：

```
[host1]$ ls /data
shared_from_server_one shared_from_server_two
```

这些文件的同步没有时序保证，因此可能不得不等待数据同步完成，尤其是在文件较大的情况下。

不保证百分百完美 由于数据是在互联网上传输的，而且是由一个没法掌控的协议来处理的，因此如果有任何安全性、扩展性或者性能要求的话，最好不要依靠它。

技巧21 保留容器的bash历史

在一个容器里实验时用户会认识到完成的时候可以擦掉所有痕迹，这是一个很开放的体验。但是这样做也会失去一些便利性。一个我们已经经历过很多次的痛点便是不能再重新调用我们之前在容器里运行的一系列命令。

问题

想要将容器的bash历史与宿主机上的命令历史共享。

解决方案

给docker run命令起一个别名来共享容器与宿主机的bash历史。

讨论

为了理解这个问题，我们打算展示一个简单的场景，在该场景中没有历史记录的话会很不方便。

试想一下用户在Docker容器里正在做一些实验，并且工作内容是一些有趣而又可以重复的事情。这里我们将使用一个简单的echo命令，但是它可能会是一长串复杂拼接的程序，最终生成一个有用的输

出：

```
$ docker run -ti --rm ubuntu /bin/bash
$ echo my amazing command
$ exit
```

在一段时间后，用户想要重新调用早先运行过的那个难以置信的 `echo` 命令。但是，他不能再重新调用它了，而且也失去了在屏幕上可以切换过去的终端会话。出于习惯，他尝试在宿主机上翻看 `bash` 历史：

```
$ history | grep amazing
```

什么都没返回，因为 `bash` 历史被保存在了如今已经删除的容器里，而不是正在使用的宿主机上。为了在宿主机上共享 `bash` 历史，可以在运行 Docker 镜像时挂载一个卷。下面是一个例子：

```
$ docker run -e HIST_FILE=/root/.bash_history \ (1)
-v=$HOME/.bash_history:/root/.bash_history \ (2)
-ti ubuntu /bin/bash
```

(1) 设置 `bash` 拾取的环境变量。这可以确保所用的 `bash` 历史文件就是挂载的那个文件

(2) 将容器里 `root` 目录下的 `bash` 历史文件映射到宿主机上

分离容器的 `bash` 历史文件 用户可能想要将容器的 `bash` 历史和宿主机分离开。要达成这一点的话，一个办法便是改变前面 `-v` 参数的第一部分的值。

每次都要敲键盘还是挺烦人的，因此为了让它对用户更加友好，可以将这个别名放到 `~/.bashrc` 文件里：

```
$ alias dockbash='docker run -e HIST_FILE=/root/.bash_history \  
-v=$HOME/.bash_history:/root/.bash_history
```

这仍然不是很平滑，如果想执行docker run命令，必须得记得敲dockbash。为了追求更完美的体验，可以将这些写到~/.bashrc文件里：

```
function basher() { (1)  
  if [[ $1 = 'run' ]] (2)  
  then  
    shift (3)  
  
    /usr/bin/docker run \  
    -e HIST_FILE=/root/.bash_history \  
    -v $HOME/.bash_history:/root/.bash_history "$@" (5)  
  else  
    /usr/bin/docker "$@" (6)  
  fi  
}  
alias docker=basher (7)
```

(1) 创建一个叫basher的bash函数来处理docker命令

(2) 确定basher/ docker的第一个参数是否是'run'。如果是

(3)删除传入的一系列参数中的一个

(4) 运行之前运行的docker run命令，通过指定Docker命令的绝对路径来避免与接下来的docker别名混淆。在实施这一方案前需要先通过运行which docker命令找出宿主机上实际正在运行的Docker的运行位置

(5) 在run的后面传入实际参数给Docker运行时

(6) 以完整原始传入的参数来运行docker命令

(7) 给在命令行上调用的docker命令起一个别名映射到创建的

basher函数。这可以确保调用docker的操作在bash从path里找到docker二进制之前被捕获

如今在下一次打开bash shell，运行任何docker run命令时，在该容器内运行的命令都将会被添加到宿主机的bash历史。一定要确保Docker的路径是正确的。例如，它的路径可能是/bin/docker。

别忘了注销宿主机的 **bash** 会话。为了让历史文件得到更新，将需要注销宿主机的原始bash会话。这得归咎于bash的微妙机制以及它更新保存在内存里的bash历史的方式。如有疑问，不妨先退出所有已知的bash会话，然后再启动一个新的以尽量确保bash历史是最新的。

技巧22 数据容器

如果在一台宿主机上大量用到卷，容器启动的管理可能会变得很麻烦。用户可能也希望用Docker专门管理数据，而不是通过宿主机进行访问。更干净地管理这些东西的一种办法便是使用纯数据容器（data-only container）的设计模式。

问题

想要在容器里使用一个外部卷，但是只想让Docker访问这些文件。

解决方案

启动一个数据容器，然后在运行其他容器时使用--volumes-from标志。

讨论

图4-3展示了数据容器模式的结构，并且解读了它的工作原理。要注意的关键一点是，在第二台宿主机里，容器并不需要知道数据位于磁盘的哪个位置，它们只要了解数据容器的名字，一切便准备就绪。这样做可以使容器的操作更加具有可移植性。

与直接映射宿主机目录的方式相比，这种方法的另一个好处是这些文件的访问是由Docker管理的，这也就意味着不太可能出现非Docker进程影响其内容的情况。

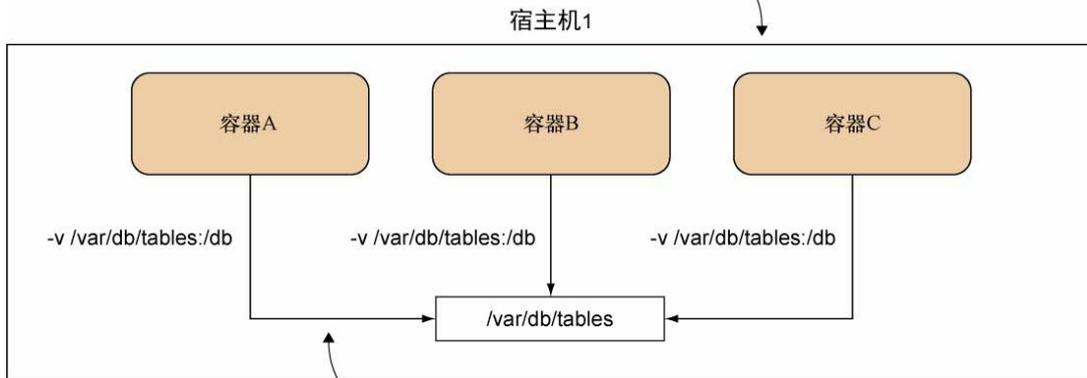
纯数据容器不一定需要运行 一个常常让人困惑的问题便是纯数据容器是否需要运行。不需要！它只需要存在，在宿主机上运行过并且没有被删除。

让我们通过一个简单的例子直观地展示一下该如何使用这一技巧。首先，运行一个数据容器：

```
$ docker run -v /shared-data --name dc busybox \  
touch /shared-data/somefile
```

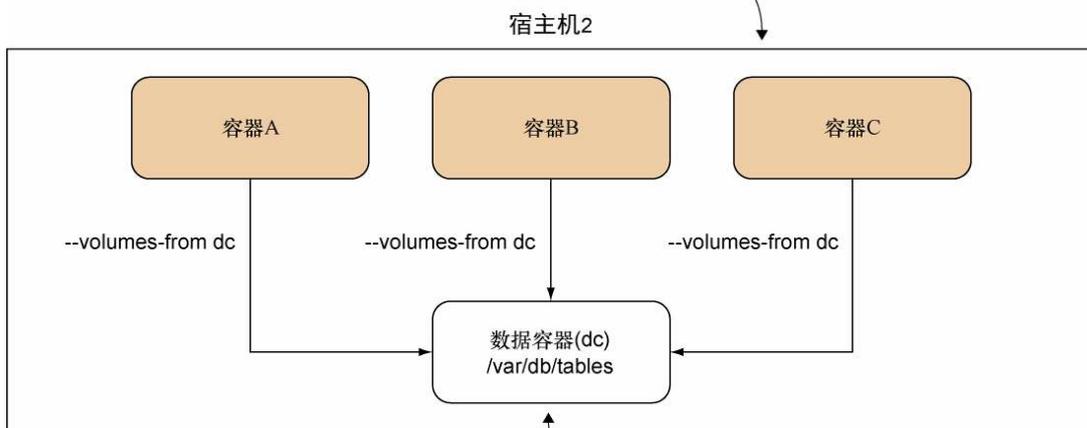
`-v`参数并没有将卷映射到一个宿主机目录，因此它将会在这个容器的管辖范围内创建一个目录。这个目录通过`touch`填充了一个文件，然后容器立刻退出了——一个数据容器被使用的时候并不需要处于运行状态。我们使用了一个小而实用的`busybox`镜像来减少我们的数据容器所需的额外成本。

这个宿主机上运行了3个容器，每一个都通过--volume/-v标志指向了宿主上的/var/db/tables目录。这里没有数据容器。



每个容器单独挂载该目录，因此如果文件夹的位置有变动或者挂载需要挪动位置的话，每个容器都需要重新配置。

这个宿主上现在运行了4个容器。前一个宿主上运行的3个容器通过--volumes-from标志均指向数据容器。



这个单个数据容器挂载宿主机的卷，为挂载到主机上的数据创建单个响应点。

图4-3 数据容器模式

然后便可以运行其他容器来访问刚创建的文件了：

```
docker run -t -i --volumes-from dc busybox /bin/sh
/ # ls /shared-data
somefile
```

`--volumes-from`标志允许通过挂载它们到当前容器的形式来引用数据容器里的文件——只需要给传入卷定义的ID即可。`busybox`镜像里没有`bash`，因此必须启动一个简化版`shell`来确认`dc`容器里的`/shared-data`目录对用户而言是的确可用的。

用户可以启动任意数量的容器，都从指定的数据容器的卷里读和写。

卷会持久化！ 需要明白的重要一点是，使用这种模式会造成大量的磁盘消耗，这可能会导致调试变得相当困难。由于Docker在纯数据容器里管理卷，而且不会在最后一个引用它的容器已经退出的前提下删除该卷，因此，任何在该卷上的数据均会被保留下来。这是为了防止意外的数据丢失。有关这方面该如何管理的建议，见技巧35。

使用这种模式的话就无须使用卷——读者可能会发现这个方案比直接挂载一个宿主机目录执行起来要更困难些。但是，如果想要将管理数据的职责完全委派给Docker进行单点管理而不受其他宿主机进程干扰的话，那么数据容器会满足这一需求。

文件路径之争 如果应用程序是从多个容器写日志到同一数据容器的话，很重要的一点便是得确保每个容器日志文件写入的是一个唯一的文件路径。如果无法确保这一点，不同的容器便有可能覆盖或者截断该文件，从而造成数据的丢失，或者可能写入的数据是交错混杂的，这就很难解析文件中的内容。类似地，如果对数据容器调用`--volumes-from`，就是允许该容器潜在地覆盖自己的目录，因此也要小心这里的命名冲突。

技巧23 使用SSHFS挂载远程卷

前面我们讨论了该怎样挂载本地文件，但是很快又出现如何挂载远程文件系统的问题。例如，用户可能会打算在一台远程服务器上共享一个引用的数据库，然后将它当成是本地文件系统来使用。

虽然从理论上来说可以在宿主机系统及服务器上配置NFS，然后通过挂载该目录来访问远程的文件系统，但是对于大多数用户而言这里有一种更为快捷和简单的方式，无须在服务器上做任何配置（只要有SSH访问权限）。

需要root权限 用户需要有root权限才能使用这一技巧，而且需要安装FUSE（Linux的用户空间级的文件系统内核模块）。可以通过在一个终端里运行`ls /dev/fuse`查看文件是否存在来确认当前系统是否支持。

问题

想要挂载一个远程文件系统而无须任何服务器端的配置。

解决方案

使用SSHFS来挂载远程文件系统。

讨论

本技巧使用FUSE内核模块通过SSH提供一个标准的文件系统接口，后台的所有通信都是通过SSH完成的。SSHFS后台还提供了各种功能（如预读取远程文件），从而使用户产生一种文件就在本地的错觉。结果便是用户一旦登录到远程服务器，就可以看到上面的文件，就像这些文件在本地一样。图4-4帮助诠释了这一点。

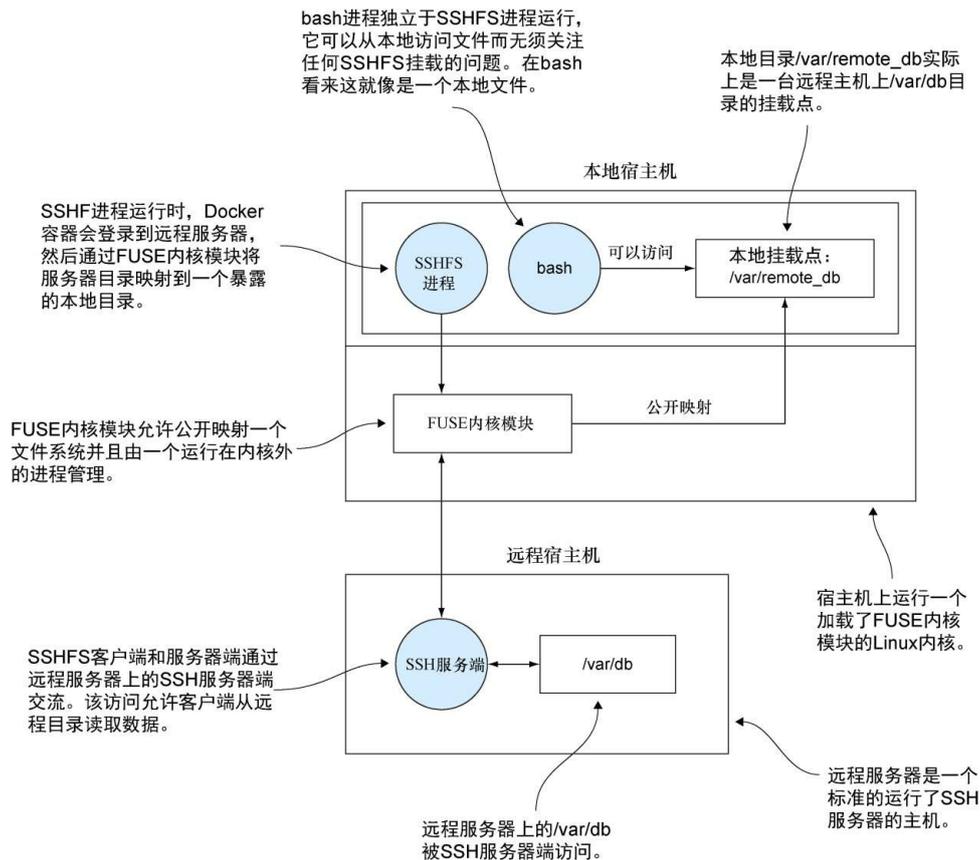


图4-4 通过SSHFS挂载一个远程文件系统

变更不会持久化到容器 虽然这一技巧没有用到Docker卷功能，并且这些文件在文件系统上也是可见的，但是这一技巧并不会提供任何容器级别的持久化。任何变更都将只作用到远程服务器的文件系统。

用户可以从运行如下命令开始，命令内容可以根据环境做相应调整。

第一步便是在宿主机上启动一个容器并附加`-privileged`:

```
$ docker run -t -i --privileged debian /bin/bash
```

然后在它启动了之后，在容器里运行`apt-get update && apt-get`

install sshfs来安装SSHFS。

在SSHFS安装成功后，按照如下步骤登录到远程宿主机：

```
$ LOCALPATH=/path/to/local/directory (1)
$ mkdir $LOCALPATH (2)
$ sshfs user@host:/path/to/remote/directory $LOCALPATH (3)
```

(1) 选择一个远程位置对应的挂载目录

(2) 创建本地挂载目录

(3) 将这里的对应值替换成远程宿主机用户名，远程宿主机的地址以及远程路径

现在就可以在刚刚创建的那个文件夹里看到远程服务器对应路径下的内容了。

通过**nonempty**选项挂载已经存在内容的目录 挂载到一个新创建的目录是最简单的，但是如果使用**-o nonempty**选项的话也可以挂载一个已经存在一些文件的目录。可以查阅SSHFS帮助手册来了解更多细节。

要干净地卸载这些文件的话，可以按照如下方式使用**fusermount**，根据需要将其替换成对应的路径：

```
fusermount -u /path/to/local/directory
```

这是一种很不错的方法，以最小的成本在容器（和标准的Linux机器）里完成远程挂载。

技巧24 通过NFS共享数据

在一个大型企业里，它很有可能有一些已经在使用的NFS共享目录——NFS是一种经过验证的方案，用于从一个中央位置提取文件。对Docker而言，能够访问这些共享文件是一件非常重要的事情。

Docker并没有原生支持NFS，并且在每个容器上安装一个NFS客户端来挂载远程文件夹也不见得是一个最佳实践。相反，推荐的方案是设置一个容器充当从NFS到一个更为Docker友好的概念——卷的中转站。

问题

想要通过NFS无缝访问远程文件系统。

解决方案

使用一个基础设施数据容器来中转访问。

讨论

这一技巧是在技巧22的基础上构建的。图4-5从概念层面展示了其理念。

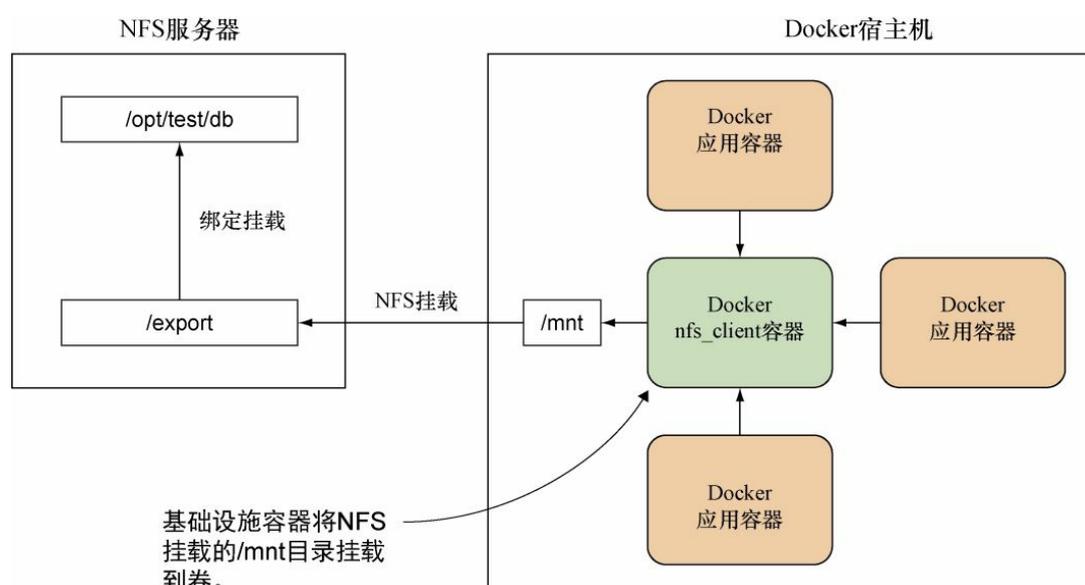


图4-5 一个用作NFS访问中转的基础设施容器

NFS服务器将一个内部目录公开为/export文件夹，它会被绑定挂载到NFS服务器宿主主机上。Docker宿主主机随后会使用NFS协议将该文件夹挂载到它的/mnt文件夹，然后再创建一个所谓的基础设施容器来绑定挂载的文件夹。

乍看上去这样做好像有一点儿过度设计，但是这样做的好处是，对Docker容器而言它提供了一个中间层：它们需要做的就是从一个预先约定好的基础设施容器挂载卷，然后由基础设施容器来处理内部设备的管道、可用性、网络等。

如何深入了解NFS超出了本书的讨论范畴。在这一技巧中，我们将通过一系列步骤在单台宿主主机上配置这样的一个共享，NFS服务器的组件作为Docker容器运行在同一台宿主主机上。该项实验已经在Ubuntu 14.04上测试通过。

假设用户想要共享宿主主机上的/opt/test/db文件夹的内容，它里面有一个mybigdb.db的文件。

以root身份安装一个NFS服务器，然后创建一个开放权限的export目录：

```
# apt-get install nfs-kernel-server
# mkdir /export
# chmod 777 /export
```

绑定挂载该db目录到export目录：

```
# mount --bind /opt/test/db /export
```

现在应该可以在/export里看到/opt/test/db目录下的内容了。

持久化该绑定挂载 如果想要在下次重启的时候持久化这一操作，需要将`opt/test/db/export none bind 0 0`这一行加到`/etc/fstab`文件中。

现在添加这一行内容到`/etc/exports`文件中：

```
/export 127.0.0.1(ro,fsid=0,insecure,no_subtree_check,async)
```

针对这个概念性验证示例，我们在`127.0.0.1`上做了本地挂载，这可能和目标有点儿差距。在一个真实场景里，用户可能会把这个锁定到一类 IP 地址，如`192.168.1.0/24`。不要将`127.0.0.1`替换为`*`从而对全世界开放，这是在玩儿火！

为了安全起见，我们这里做了只读（`ro`）挂载，但是用户可以通过将`ro`替换成`rw`做可读写挂载。别忘了如果要这么做的话，需要在`async`标志后面加上一个`no_root_squash`标志，但是在实施之前请先考虑一下安全性。

将NFS上共享的目录挂载到`/mnt`文件夹下，导出之前在`/etc/exports`里指定的文件系统，然后重启NFS服务以生效：

```
# mount -t nfs 127.0.0.1:/export /mnt
# exportfs -a
# service nfs-kernel-server restart
```

现在准备好运行基础设施容器：

```
# docker run -ti --name nfs_client --privileged -v /mnt:/mnt
└─ busybox /bin/true
```

而现在可以运行容器——不需要权限也无须拥有底层实现的知识

就可以访问目录：

```
# docker run -ti --volumes-from nfs_client debian /bin/bash
root@079d70f79d84:/# ls /mnt
myb
root@079d70f79d84:/# cd /mnt
root@079d70f79d84:/mnt# touch asd
touch: cannot touch `asd': Read-only file system
```

采用一个命名规范以提高运维效率。如果需要管理大量的这种容器，可以通过设定一个命名规范，例如，针对一个暴露了/opt/database/livepath的容器起名为--name nfs_client_opt_database_live来简化管理。

这种挂载一个中央授权访问的共享资源供其他人在多个容器中使用模式真的很强大，它可以使开发工作流变得更加简单。

安全性是无可替代的。请记住这一技巧只提供隐晦的（几乎没有）安全性保证。正如最后看到的那样，任何人只要能够运行Docker便拥有宿主主机上的root权限。

技巧25 开发工具容器

作为一名工程师，如果发现自己常常苦恼的问题是，在其他机器上没有自己雪花般美丽独特的个人开发环境里的程序或配置，那么这一技巧也许正好适用。类似地，如果想和其他人分享自己精心定制的开发环境，Docker可以让这件事情变得更简单。

问题

想在其他人的机器上访问自己定制的开发环境。

解决方案

用自己的配置创建一个容器，然后把它放到注册中心上。

讨论

作为演示，这里将使用一个我们的开发工具镜像。读者可以通过运行`docker pull dockerinpractice/docker-dev-tools-image`来下载它。如果想查看Dockerfile，仓库地址是<https://github.com/docker-in-practice/docker-dev-tools-image>。

运行该容器非常简单——直接执行`docker run -t -i dockerinpractice/docker-dev-tools-image`就能给用户一个开发环境的shell。这里读者可以使用我们默认的配置，也可以就配置方面给我们一些建议。

本技巧和其他技巧配合起来更能体现其威力。这里读者可以看到一个开发工具容器，用来在宿主机网络和IPC栈上展示一个用户图形界面（GUI），然后挂载宿主机上的代码：

```
docker run -t -i \  
-v /var/run/docker.sock:/var/run/docker.sock \ (1)  
-v /tmp/.X11-unix:/tmp/.X11-unix \ (2)  
  
-e DISPLAY=$DISPLAY \ (3)  
--net=host --ipc=host \ (4)  
-v /opt/workspace:/home/dockerinpractice \ (5)  
dockerinpractice/docker-dev-tools-image
```

(1) 挂载Docker套接字以访问宿主机上的Docker守护进程

(2) 挂载X服务器Unix域套接字可以启动一个基于用户图形界面的应用（见技巧26）

(3) 设置构建容器时的环境变量以使用宿主机的显示器

(4) 这些参数绕过容器的网桥并且通过这些参数可以访问宿主机上的进程间通信文件（见技巧97）

(5) 挂载工作区到该容器的home目录

上述命令提供了一个环境，它能够访问宿主机的下列资源：

- 网络；
- Docker守护进程（运行普通Docker命令，就像在宿主机上一样）；
- 进程间通信（IPC）文件；
- 如果需要的话，用X服务器启动基于用户图形界面的应用。

宿主机安全性 挂载宿主机目录时，小心不要挂载任何重要的目录，因为可能会破坏文件。通常要避免挂载宿主机上root用户的任何目录。

4.2 运行容器

虽然本书的大部分内容都是关于正在运行的容器的，但是也有一些与在宿主机上运行容器相关的实战技巧不容易被注意到。我们将看看如何让GUI应用程序工作，包括顺畅地退出一个启动的容器而不是杀掉它，检查容器的状态和源镜像，以及关闭这些容器。

技巧26 在Docker里运行GUI

在技巧14中，我们已经看到一个在容器里由VNC服务器提供的GUI。这是一种在Docker容器中查看应用程序的方法，它是内置的，只需要一个VNC客户端便可以使用。

幸好有更轻量、集成度更高的方法在桌面上运行GUI，不过这需要更多的设置。它会在宿主机上挂载一个目录以管理和X服务器的通信，以便容器可以访问到它。

问题

想要在一个容器里运行GUI，就像是正常的桌面应用一样。

解决方案

使用自己的用户凭证和程序创建镜像，然后将用户的X服务器绑定挂载到它上面。

讨论

图4-6展示了最终设置的工作原理。

容器会通过挂载宿主机的/tmp/.X11目录链接到宿主机，而这正是容器可以在宿主机的桌面上完成操作的原因所在。

首先在一个觉得合适的地方创建一个新目录，然后通过id命令确定用户ID和组ID，如代码清单4-1所示。

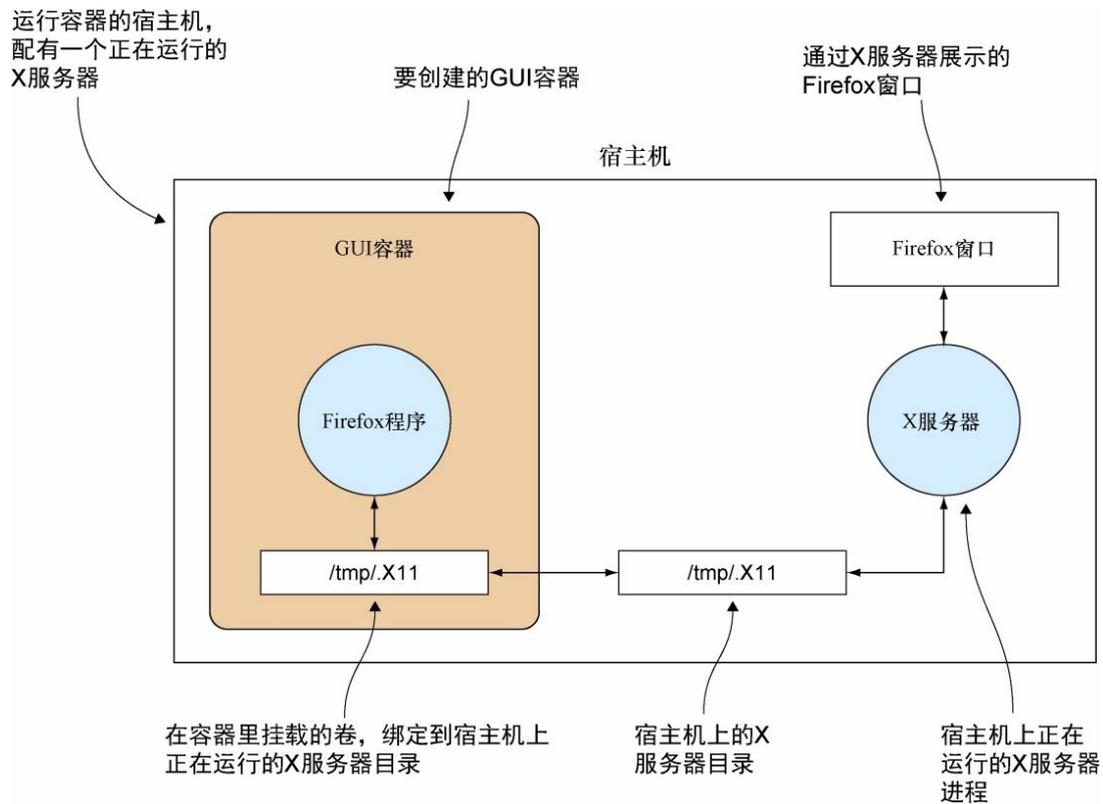


图4-6 和宿主机的X服务器通信

代码清单4-1 设置目录然后找出具体的用户信息

```

$ mkdir dockergui
$ cd dockergui
$ id (1)
uid=1000(dockerinpractice) \ (2)
gid=1000(dockerinpractice) \ (3)
groups=1000(dockerinpractice),10(wheel),989(vboxusers),990(docker)

```

- (1) 收集Dockerfile所需的用户信息
- (2) 记住用户ID (uid)。在这个例子里，它是1000
- (3) 记住组ID (gid)。在这个例子里，它是1000

现在创建一个叫Dockerfile的文件，如下所示：

```

FROM ubuntu:14.04
RUN apt-get update

```

```
RUN apt-get install -y firefox (1)

RUN groupadd -g GID USERNAME (2)
RUN useradd -d /home/USERNAME -s /bin/bash \
-m USERNAME -u UID -g GID (3)

USER USERNAME (4)
ENV HOME /home/USERNAME (5)
CMD /usr/bin/firefox (6)
```

(1) 安装Firefox作为GUI应用。用户可以把这个换成任何想要安装的应用

(2) 把宿主机上的组添加到镜像里，把GID替换为组ID，把USERNAME替换为用户名

(3) 把用户账号加到镜像里，把USERNAME替换为用户名，把UID替换为用户ID，把GID替换为组ID

(4) 镜像应当以刚创建的用户身份运行。把USERNAME替换为用户名

(5) 正确地设置HOME变量。把USERNAME替换为用户名

(6) 默认在启动时运行Firefox

现在就可以基于该Dockerfile来构建镜像，然后把结果标记为“gui”了，如代码清单4-2所示。

代码清单4-2 构建gui镜像

```
$ docker build -t gui .
```

以代码清单4-3所示的方式运行它。

代码清单4-3 运行gui镜像

```
docker run -v /tmp/.X11-unix:/tmp/.X11-unix \ (1)
-e DISPLAY=$DISPLAY gui (2)
-h $HOSTNAME -v $HOME/.Xauthority:/home/$USER/.Xauthority (3)
```

(1) 绑定挂载X服务器目录到容器.....

(2)在容器里将DISPLAY变量设置成宿主机上的相同值，这样程序便能知道与哪个X服务器通信.....

(3)并为容器提供合适的授权凭证

会看到弹出一个Firefox窗口！

可以使用这一技巧来避免桌面工作与开发工作混在一起。以Firefox为例，出于测试目的，开发人员可能想要看到应用程序在没有Web缓存、书签或者搜索历史的情况下的行为，并且这种查看可以重复。如果在尝试启动镜像和运行Firefox的时候出现无法打开显示器这样的出错信息的话，不妨查看技巧58以获取关于使容器启动的图形应用在宿主机上得以展示的更多方法。

技巧27 检查容器

虽然Docker命令提供了查看镜像和容器信息的能力，有时候用户也许也想了解这些Docker对象的内部元数据的更多信息。

问题

想要找出一个容器的IP地址。

解决方案

使用docker inspect查询和过滤容器的元数据。

讨论

`docker inspect`命令提供了一个以JSON格式查看Docker容器内部元数据的能力。该命令会产生大量的输出，因此在代码清单4-4中仅列出了一个镜像元数据的简明摘要。

代码清单4-4 检查镜像的原始输出

```
$ docker inspect ubuntu | head
[{"Architecture": "amd64",
  "Author": "",
  "Comment": "",
  "Config": {
    "AttachStderr": false,
    "AttachStdin": false,
    "AttachStdout": false,
    "Cmd": [
      "/bin/bash"
    ]
  }
}]
$
```

用户可以通过名字或ID来查看镜像和容器，不过它们的元数据会不太一样。例如，一个容器会有像“State”之类的运行时字段，但镜像是没有的（镜像是没有状态的）。

在这个例子里，用户想要在宿主机上找出一个容器的IP地址。为了做到这一点，可以使用`docker inspect`命令并附带`format`标志（如代码清单4-5所示）。

代码清单4-5 确定一个容器的IP地址

```
docker inspect \ (1)
--format '{{.NetworkSettings.IPAddress}}' \ (2)
0808ef13d450 (3)
```

(1) 使用`docker inspect`命令

(2) 附加`format`标志。该标志会使用Go模板（不属于本书的内容范畴）来格式化输出。这里，会从`inspect`输出中的`NetworkSettings`

字段里提取出IPAddress

(3) 想要查看的DockerID

本技巧对自动化来说非常有用，因为这个接口比其他Docker命令可能会更稳定些。代码清单4-6给出的命令列出了所有正在运行的容器的IP地址，然后对它们执行ping命令。

代码清单4-6 拿到正在运行中的容器的IP然后逐个ping

```
$ docker ps -q | \ (1)
xargs docker inspect --format='{{.NetworkSettings.IPAddress}}' | \ (2)
xargs -l1 ping -c1 (3)
PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
64 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.095 ms

--- 172.17.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.095/0.095/0.095/0.000 ms
```

(1) 拿到所有正在运行中的容器的ID

(2) 针对所有容器ID执行inspect命令以获取它们的IP地址

(3) 逐个取出IP地址，然后依次运行ping

注意，因为ping只接受单个IP地址，所以必须给xargs传入一个额外的参数，告诉它针对每个单独的行执行该命令。

设置一个正在运行的容器来测试 如果没有正在运行的容器的话，运行下面这条命令来获取一个：`docker run -d ubuntu sleep 1000`。

技巧28 干净地杀掉容器

如果容器终止时的状态对用户而言很重要，用户可能会想要了解 `docker kill` 和 `docker stop` 之间的区别。这一差异在需要应用程序正常关闭以保存数据时显得尤为重要。

问题

想要干净地终止一个容器。

解决方案

使用 `docker stop` 而不是 `docker kill`。

讨论

要理解的关键点在于 `docker kill` 的行为和标准的 `kill` 命令行程序并不相同。

除非另有说明，`kill` 程序的默认工作方式是向指定的进程发送 `TERM` 信号（即信号值为15）。这个信号表示程序应该终止，但是不要强迫程序终止。当这个信号被处理时，大多数程序将执行某种清理工作，但是该程序也可以执行其他操作，包括忽略该信号。

相反，`KILL` 信号（即信号值为9）会强迫指定的程序终止。

令人困惑的是，`docker kill` 对正在运行的进程使用的是 `KILL` 信号，这使得该进程没办法处理终止过程。这就意味着一些诸如包含运行进程ID之类的文件可能会残留在文件系统中。根据应用程序管理状态的能力，如果再次启动容器，这可能会也可能不会造成问题。

更令人不解的是，`docker stop` 命令则像 `kill` 命令那样工作，发送的是一个 `TERM` 信号（见表4-1）。

表4-1 停止和杀死容器

命 令	默认信号量	默认信号量的值
kill	TERM	15
docker kill	KILL	9
docker stop	TERM	15

总而言之，如果想使用kill就不要使用docker kill，而且最好养成使用docker stop的习惯。

技巧29 使用Docker Machine来置备Docker宿主机

在本机上安装Docker可能不是一件太困难的事情——有一个脚本可以很方便地用来做这件事，或者也就是几条命令添加几个适当的源到包管理器的事情。但是，当试图在其他宿主机上管理Docker安装时会很乏味。

问题

想要在与自己机器独立的Docker宿主机上启动容器。

解决方案

使用Docker Machine。

讨论

如果需要在多个外部宿主机上运行Docker容器，这一技巧就有用

武之地了。需要它可能有下面几个原因：

- 通过在自己物理机里置备一台虚拟机来测试Docker容器之间的网络；
- 借助VPS供应商到一台更强力的机器上置备容器；
- 冒着损毁宿主机的风险去进行一些疯狂实验；
- 保留运行在多个云厂商的选择权。

无论什么原因，Docker Machine也许就是答案。它也是更复杂的编排工具（如Docker Swarm）的门户。

1. Docker Machine是什么

Docker Machine主要是一个便利程序。它将大量配置外部宿主机的繁琐的指令包装起来，变成一些易于上手的命令。如果对Vagrant很熟悉，这有着类似的体验：通过一个一致的接口使置备和管理其他机器环境变得更简单。如果把注意力放回到第2章里的架构概览图，查看该图的方式之一便是把它想象成一个客户端，它可以方便地管理不同的Docker守护进程（见图4-7）。

图4-7里列出的Docker宿主机提供商并不全，而且这个清单可能还会不断增长。在编写本书的时候有下列驱动可用，它允许使用给定的宿主机供应商来置备：

- Amazon Web Services（AWS）；
- Digital Ocean；
- Google Compute Engine；
- IBM Softlayer；
- Microsoft Azure；

- Microsoft Hyper-V;
- OpenStack;
- Rackspace;
- Oracle VirtualBox;
- VMware Fusion;
- VMware vCloud Air;
- VMware vSphere。

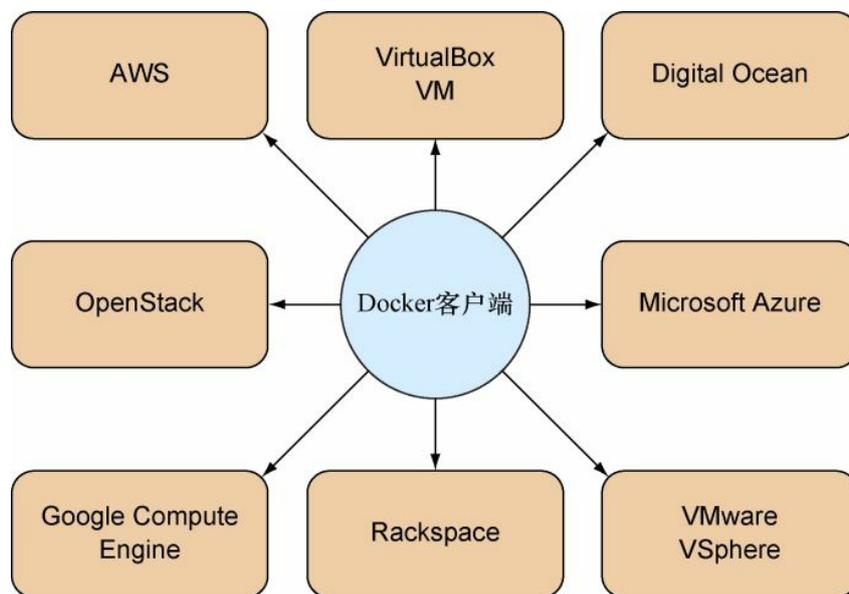


图4-7 Docker Machine作为外部宿主机的一个客户端

根据驱动提供的功能，置备机器时必须指定的参数也会有很大的不同。与OpenStack的17个参数相比，用户的机器置备一台Oracle Virtualbox虚拟机在创建时只有3个可用的参数。

2. Docker Machine不是什么

需要指出的是，Docker Machine不是一种Docker的集群解决方案。其他工具（像Docker Swarm）填补了这块功能，我们将在后面探讨这些。

3. 安装

安装程序是一个简单的二进制文件。针对不同架构的下载链接和安装指令可以在<https://github.com/docker/machine/releases>找到。

迁移二进制文件？ 用户可能想要把二进制文件放置到一个标准的位置，如/usr/bin，然后在继续之前先确保它被重命名或者符号链接到了docker-machine，因为下载的文件可能会有一个更长的带有运行架构的后缀名。

4. 使用Docker Machine

为了演示Docker Machine的用法，可以从创建一个上面正常运行着Docker守护进程的虚拟机开始。

假定选用 **VirtualBox** 虚拟机管理器 为了让它能够正常运行，用户需要安装Oracle公司的VirtualBox。在大多数包管理器里均可以找到它。

```
$ docker-machine create --driver virtualbox host1 (1)
INFO[0000] Creating CA: /home/imiell/.docker/machine/certs/ca.pem
INFO[0000] Creating client certificate: /home/imiell/.docker/machine/certs/
└─> cert.pem

INFO[0002] Downloading boot2docker.iso to /home/imiell/.docker/machine/cache/
└─> boot2docker.iso...
INFO[0011] Creating VirtualBox VM...
INFO[0023] Starting VirtualBox VM...
INFO[0025] Waiting for VM to start...
INFO[0043] "host1" has been created and is now the active machine. (2)
INFO[0043] To point your Docker client at it, run this in your shell:
└─> $(docker-machine env host1) (3)
```


(1) `$()`可以拿到`docker-machine env`命令的输出，然后将它应用到用户的环境。`docker-machine env`会输出一组命令，用户可以用它来设置默认宿主机上的Docker命令

(2) 环境变量名均以`DOCKER_`开头

(3) `DOCKER_HOST`变量即虚拟机上Docker守护进程的端口

(4) 这些变量是用来处理客户端和新宿主机之间连接的安全性

(5) `docker`命令现在被指向了创建好的虚拟机上，而不再是之前所使用的宿主机。在新的虚拟机上并没有创建任何的容器，所以输出结果是空

(6) `ssh`子命令将会直接连接到新的虚拟机上

5. 管理宿主机

通过一个客户端机器管理多台Docker宿主机可能使追踪背后的运作很困难。为了简化这一点，Docker Machine提供了众多的管理命令，如表4-2所示。

表4-2 一系列`docker machine`命令

子 命 令	行 为
<code>create</code>	创建一台新的机器
<code>ls</code>	列出Docker宿主机

stop	停止机器
start	启动机器
restart	停止并随后启动机器
rm	销毁一台机器
kill	杀掉一台机器
inspect	以JSON的格式返回机器的元数据
config	返回连接机器所需的配置信息
ip	返回一台机器的IP地址
url	返回一个机器上Docker守护进程的URL
upgrade	将宿主机上的Docker升级到最新版本

下面这个例子列出了两台机器。当前活跃的机器旁边会标有一个星号，并且拥有一个相关的状态，类似于容器或进程的状态标识：

```

$ docker-machine ls
NAME    ACTIVE DRIVER      STATE    URL                                SWARM
host1           virtualbox Running  tcp://192.168.99.102:2376
host2  *        virtualbox Running  tcp://192.168.99.103:2376

```

事实上，这里可以看作是将机器转变为进程，就像Docker本身也可以看作是将环境转变为进程。

切换回来 读者可能会疑惑该怎么切回到最开始宿主机上的那个 Docker实例。在编写本书时，我们仍然没有找到一种简单的办法来做到这一点。用户可以通过`docker-machine rm`删除所有机器，如果实在没有办法，也可以通过`unset DOCKER_HOST DOCKER_TLS_VERIFY DOCKER_CERT_PATH`手动重置之前设置的环境变量来实现这一点。

4.3 构建镜像

尽管Dockerfile的简单性使其成为一款强大的省时工具，但是这里仍然有一些细节之处可能会让人感到困惑。我们将学习一些节省时间的功能及其具体细节，从ADD指令开始，然后会覆盖到Docker构建的缓存，如何让它失效，以及怎么操作它来获得收益。

要了解Dockerfile指令相关的完整内容，请参考Docker的官方文档。

技巧30 使用ADD将文件注入到镜像

尽管可以在Dockerfile里使用RUN命令和基本的shell来添加文件，但是这很快就会变得无法维护。ADD命令被加入Dockerfile命令清单里正是为了满足优雅地将大量文件放入镜像的需求。

问题

想要以一种简便的方式下载和解压一个压缩包到镜像里。

解决方案

打包并压缩文件，随后在Dockerfile里使用ADD指令。

讨论

通过`mkdir add_example && cd add_example`为这次的Docker构建创造一个全新的环境，然后获取一个压缩包并给它赋予一个名称，以便稍后引用：

```
$ curl \
https://www.flamingspork.com/projects/libeatmydata/libeatmydata-105.tar
.gz > \
my.tar.gz
```

在这个案例里，我们使用的是其他技术创建的tar文件，但是它也可以是任意你喜欢的压缩包：

```
FROM debian
RUN mkdir -p /opt/libeatmydata
ADD my.tar.gz /opt/libeatmydata/
RUN ls -lRt /opt/libeatmydata
```

通过`docker build --no-cache .`构建这个Dockerfile，并且输出应该看似是下面这样：

```
$ docker build --no-cache .
Sending build context to Docker daemon 422.9 kB
Sending build context to Docker daemon
Step 0 : FROM debian
---> c90d655b99b2
Step 1 : RUN mkdir -p /opt/libeatmydata
---> Running in fe04bac7df74
---> c0ab8c88bb46
Removing intermediate container fe04bac7df74
Step 2 : ADD my.tar.gz /opt/libeatmydata/
---> 06dcd7a88eb7
Removing intermediate container 3f093a1f9e33
Step 3 : RUN ls -lRt /opt/libeatmydata
---> Running in e3283848ad65
/opt/libeatmydata:
total 4
drwxr-xr-x 7 1000 1000 4096 Oct 29 23:02 libeatmydata-105

/opt/libeatmydata/libeatmydata-105:
total 880
```

```

drwxr-xr-x 2 1000 1000    4096 Oct 29 23:02 config
drwxr-xr-x 3 1000 1000    4096 Oct 29 23:02 debian
drwxr-xr-x 2 1000 1000    4096 Oct 29 23:02 docs
drwxr-xr-x 3 1000 1000    4096 Oct 29 23:02 libeatmydata
drwxr-xr-x 2 1000 1000    4096 Oct 29 23:02 m4
-rw-r--r-- 1 1000 1000    4096 Oct 29 23:02 config.h.in
[...edited...]
-rw-r--r-- 1 1000 1000   1824 Jun 18 2012 pandora_have_better_malloc.m4
-rw-r--r-- 1 1000 1000    742 Jun 18 2012 pandora_header_assert.m4
-rw-r--r-- 1 1000 1000    431 Jun 18 2012 pandora_version.m4
---> 2ee9b4c8059f
Removing intermediate container e3283848ad65
Successfully built 2ee9b4c8059f

```

可以看到，Docker守护进程在这里输出的内容（所有文件的扩展输出已被修改），tar包被解压到了目标目录。Docker能够解压绝大多数标准类型（.gz、.bz2、.xz、.tar）的压缩文件。

值得一提的是，尽管可以指定一个URL来下载压缩包，但是只有存储在本地文件系统中的包才会被自动解压。这可能容易导致混淆。

如果使用下面的Dockerfile重复前面的过程，我们会发现文件被下载下来，但是并没有被解压：

```

FROM debian (1)
RUN mkdir -p /opt/libeatmydata
ADD \
https://www.flamingspork.com/projects/libeatmydata/libeatmydata-105.tar
.gz \
/opt/libeatmydata/ (2)
RUN ls -lRt /opt/libeatmydata

```

(1) 通过一个URL从互联网上获取文件

(2) 通过目录名和末尾的斜杠指明目标目录。没有末尾的斜杠的话，参数将被视作是下载好的文件的文件名

下面是最终的构建输出：

```
Sending build context to Docker daemon 422.9 kB
Sending build context to Docker daemon
Step 0 : FROM debian
---> c90d655b99b2
Step 1 : RUN mkdir -p /opt/libeatmydata
---> Running in 6ac454c52962
---> bdd948e413c1
Removing intermediate container 6ac454c52962
Step 2 : ADD \
https://www.flamingpork.com/projects/libeatmydata/libeatmydata-105.tar
.gz \
/opt/libeatmydata/
Downloading [=====] \
419.4 kB/419.4 kB
---> 9d8758e90b64
Removing intermediate container 02545663f13f
Step 3 : RUN ls -lRt /opt/libeatmydata
---> Running in a947eaa04b8e
/opt/libeatmydata:
total 412
-rw----- 1 root root 419427 Jan  1 1970 \
libeatmydata-105.tar.gz (1)
---> f18886c2418a
Removing intermediate container a947eaa04b8e
Successfully built f18886c2418a
```

(1) libeatmydata-105.tar.gz文件被下载下来并放到
了/opt/libeatmydata目录下，但是没有自动解压

注意，如果在之前的Dockerfile中的ADD那行没有带末尾的斜杠符的话，文件会被下载下来然后以该文件名保存。末尾的斜杠符指明该文件应该被下载下来，并放置到指定的目录里。

所有的新文件和新目录的所有者为root（或是那个在容器里组或者用户ID为0的用户）。

不想解压？ 如果想要从本地文件系统添加一个压缩文件但不想解压它，可以使用COPY命令，它和ADD命令看上去的确很像，但它不会自动解压任何文件。

文件名里有空格？

如果文件名里带有空格，需要在ADD（或COPY）指令里用双引号的形式标明：

```
ADD "space file.txt" "/tmp/space file.txt"
```

技巧31 重新构建时不使用缓存

借助Dockerfile进行构建时可以利用一个很有用的缓存功能：已经运行过的构建步骤只有在命令内容发生变化时才会被重新执行。代码清单4-7展示了第1章中的to-do应用程序重新构建的输出。

代码清单4-7 重新构建时使用缓存

```
$ docker build .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM node
---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
---> Using cache (1)
---> 8f5a8a3d9240 (2)
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
---> Using cache
---> 48db97331aa2
Step 3 : WORKDIR todo
---> Using cache
---> c5c85db751d6
Step 4 : RUN npm install > /dev/null
---> Using cache
---> be943c45c55b
Step 5 : EXPOSE 8000
---> Using cache
---> 805b18d28a65
Step 6 : CMD npm start
---> Using cache
---> 19525d4ec794
Successfully built 19525d4ec794 (3)
```

- (1) 表明用户在构建时使用了缓存
- (2) 指定已缓存的镜像/分层ID
- (3) 最后镜像被“重新构建”，但是实际上并没有什么改动

这样做的确很有用而且省时，但是它往往并不是用户期望的行为。

以前面的Dockerfile为例，假设改动了源代码并推送到了Git仓库。新代码将不会被检出，因为`git clone`命令本身并没有发生变化。就Docker构建而言，它是相同的，因此缓存的镜像会在这次构建过程中被复用。

在这类情况下，用户可能想要的是不使用缓存进行重新构建。

问题

想要不使用缓存重新构建Dockerfile。

解决方案

构建镜像时加上`--no-cache`标志。

讨论

为了强制重新构建时不使用镜像缓存，需要在运行`docker build`时加上`--no-cache`标志。代码清单4-8中加上`--no-cache`运行之前的构建。

代码清单4-8 强制重新构建时不使用缓存

```
$ docker build --no-cache . (1)
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
```

```
Step 0 : FROM node
---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
---> Running in ca243b77f6a1 (2)
---> 602f1294d7f1 (3)
Removing intermediate container ca243b77f6a1
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
---> Running in f2c0ac021247
---> 04ee24faaf18
Removing intermediate container f2c0ac021247
Step 3 : WORKDIR todo
---> Running in c2d9cd32c182
---> 4e0029de9074
Removing intermediate container c2d9cd32c182
Step 4 : RUN npm install > /dev/null
---> Running in 79122dbf9e52
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
---> 9b6531f2036a
Removing intermediate container 79122dbf9e52
Step 5 : EXPOSE 8000
---> Running in d1d58e1c4b15
---> f7c1b9151108
Removing intermediate container d1d58e1c4b15
Step 6 : CMD npm start
---> Running in 697713ebb185
---> 74f9ad384859
Removing intermediate container 697713ebb185
Successfully built 74f9ad384859 (4)
```

(1) 通过--no-cache标志在重新构建Docker镜像时无视缓存的分层

(2) 这次没有提到在使用缓存

(3) 中间镜像的ID与代码清单4-7中列出的不同

(4) 一个新的镜像构建完成

展示的输出没有提到缓存，而且每一个中间层的ID都和代码清单4-7中的输出不同。

相同的问题同样可能发生在其他场景中。我们早期在使用

Dockerfile时曾经诧异于一个网络波动导致一条命令无法正确地从网络获取到某个东西，但是该命令却没有报错。我们继续调用`docker build`，但是结果里的bug始终存在！这是因为一个“错误”的镜像已经被加入缓存中，而我们之前并没有理解Docker构建缓存的工作机制。最终，我们解决了这个问题。

技巧32 拆分缓存

使用`--no-cache`标志通常足以解决在缓存方面遇到的任何问题。但是有时候往往需要的是一个更细粒度的解决方案。如果有一个构建需要花费很长时间，例如，想使用缓存到一个指定层，然后重新运行一条命令使其上的缓存失效并创建一个新镜像。

问题

想要在Dockerfile构建中从一个指定的点开始失效Docker构建缓存。

解决方案

在命令后面增加一条无害的注释，从而让缓存失效。

讨论

不妨从<https://github.com/docker-in-practice/todo>的Dockerfile开始上手，我们已经完成了一次构建，并随后在CMD这一行上添加了一条注释。我们可以在这里看到再次执行`docker build`的输出：

```
$ docker build . (1)
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM node
----> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
----> Using cache
```

```
---> 8f5a8a3d9240
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.gi
t
---> Using cache
---> 48db97331aa2
Step 3 : WORKDIR todo
---> Using cache
---> c5c85db751d6
Step 4 : RUN npm install
---> Using cache
---> be943c45c55b
Step 5 : EXPOSE 8000
---> Using cache (2)
---> 805b18d28a65
Step 6 : CMD ["npm","start"] #bust the cache (3)
---> Running in fc6c4cd487ce
---> d66d9572115e (4)
Removing intermediate container fc6c4cd487ce
Successfully built d66d9572115e
```

- (1) 一次“正常” docker build
- (2) 缓存用到这里结束
- (3) 缓存已经失效但是命令仍然有效地维持不变
- (4) 一个新的镜像被创建出来

这一技巧的原理在于Docker将非空的更改均当作一行新的命令来对待，因此缓存的镜像层便不会再被复用。

读者可能会感到疑惑（我们第一次看到Docker时也有同感），Docker的分层是否可以在镜像之间迁移并且合并它们，就像它们是Git中的更改集一样。在Docker里，至少当前是不可能实现的。一个分层被定义成仅仅是一个指定镜像的更改集。因此，一旦缓存被破坏了，构建里的命令就不能再复用它。

正因为如此，如果可以的话建议最好将不太可能变动的命令放到更靠近Dockerfile开头的位置。

4.4 保持阵型

如果你跟我们一样（并且仔细阅读了本书），**Docker**知识日益增长，使用**Docker**成瘾，从而启动更多数量的容器，而且会在宿主机上下载更多的镜像。

随着时间的流逝，**Docker**将会消耗越来越多的资源，而一些容器和卷的清理将变得很有必要——我们将展示如何做以及这样做的原因。我们也会介绍一些用来保持**Docker**环境干净整洁的可视化工具，以便不喜欢命令行的用户可以从中解脱出来。

技巧33 运行**Docker**时不加**sudo**

Docker守护进程在机器上以root用户运行在后台，在暴露给用户的同时，也给了它很大的权利。需要使用**sudo**只是一个结果，但是这样做会变得不太方便，而且也会造成一些第三方**Docker**工具无法使用。

问题

想要无须**sudo**便可以运行**docker**命令。

解决方案

将自身加到**docker**组。

讨论

Docker使用一个用户组来围绕着**Docker Unix**域套接字来管理权限。出于安全的原因，发行版默认不会将用户加到该用户组中。

通过将用户加到该组，用户便能以自己的身份使用**docker**命

令：

```
$ sudo addgroup -a username docker
```

重新启动Docker然后完全注销并再次登录，或者如果更简单一些的话，直接重启机器。现在以自己的用户身份运行Docker时无须再记住敲sudo或者设置一个别名了。

技巧34 清理容器

Docker新用户经常抱怨的是，在短时间内，用户可能因为各种情况在系统上残留许多容器，而命令行没有用于管理这些的标准工具^[1]。

问题

想要整理系统上的容器。

解决方案

设置一个别名来运行清理旧容器的命令。

讨论

这里最简单的方式自然是删除所有容器。显然，这是一个有风险的方案，只应该在确定这就是预期的行为时使用。下列命令将会删除在宿主机上的所有容器：

```
$ docker ps -a -q | \ (1)  
xargs --no-run-if-empty docker rm -f (2)
```

(1) 获取所有容器ID的列表，包括正在运行的以及已停止的，然后将它们传给.....

(2)`docker rm -f`命令，被传入的任意容器将会被删除，即使它们还处于运行状态

简要解释一下`xargs`命令，它会获取输入的每一行内容，并将它们全部作为参数传递给后续命令。为了防止报错，我们这里传入了一个额外的参数`--no-run-if-empty`，这可以避免在前面的命令完全没有输出的情况下执行该命令。

如果有正在运行的容器想要保留，却又想删除所有已经退出的容器的话，不妨过滤一下`docker ps`命令的返回的条目：

```
docker ps -a -q --filter status=exited | \ (1)
xargs --no-run-if-empty docker rm (2)
```

(1) `--filter`标志会告知`docker ps`命令想要返回的容器。在这种情况下限制成状态为已经退出的那些容器。也可以选择处于正在运行中或者正在重启状态的容器

(2) 这次不用再强行删除容器，因为根据给定的过滤参数，它们本身就不应该处于运行状态

作为更高级用例的示范，下列命令将列出所有返回非零错误码的容器。如果系统上有许多容器而用户想要自动检查和删除那些异常退出的容器，就可能需要这样做：

```
comm -3 \ (1)
<(docker ps -a -q --filter=status=exited | sort) \ (2)
<(docker ps -a -q --filter=exited=0 | sort) | \ (3)
xargs --no-run-if-empty docker inspect > error_containers (4)
```

(1) 运行`comm`命令来比较两个文件内容的差异。加上`-3`参数将不会显示同时出现在两个文件里的行内容（这些容器的退出码都是

0)，然后输出其他不同的部分

(2) 找出退出的容器 ID，给它们排序，然后以文件形式传给 `comm`

(3) 找出退出码为0的容器，给它们排序，然后以文件形式传给 `comm`

(4) 对非0退出码（`comm`命令管道的输出）的容器执行 `docker inspect`，并将输出结果保存到 `error_containers` 文件中

bash中的进程替换 你也许还没看到过这种用法，`bash`里的“<(命令)”语法被称为进程替换。它允许把一个命令的输出结果作为一个文件，传给其他命令，这在无法使用管道输出的时候非常有用。

上述示例相对比较复杂，但是它展示了将不同的工具命令组合在一起的威力。它会输出所有已停止的容器的ID，然后挑出那些非0退出码的容器（即那些异常方式退出的容器）。如果读者还在努力理解这个用法的话，不妨先单独运行每条命令，然后理解它们的含义，这样有助于更快地了解整个过程。

像这样的命令可以用来在生产环境里采集容器信息。可能需要调整一下它，改为运行一个从容作业来清除正常退出的容器。

将单行代码设置为命令

可以给命令设置别名，以便在登录到宿主机上后更容易执行。为了达成这一点，需要在 `~/.bashrc` 文件里加上如下代码行：

```
alias dockernuke='docker ps -a -q | \
```

```
xargs --no-run-if-empty docker rm -f'
```

然后，在下次登录时，从命令行运行 `dockernuke`，将删除在系统上找到的任何 Docker 容器。

我们发现这样做节省的时间是相当可观的。但是要小心！这种方式同样也非常容易误删生产环境的容器，我们可以证明。即使足够小心，不去删除正在运行的容器，仍然可能会误删那些没有运行但却仍然有用的纯数据容器。

技巧35 清理卷

尽管卷是 Docker 的一个强大的功能，伴随而来的也有一些显著的运维缺陷。

因为卷可以在不同的容器之间共享，所以在挂载它们的容器被删除时无法清空掉这些卷。试想一下图4-8中描述的场景。

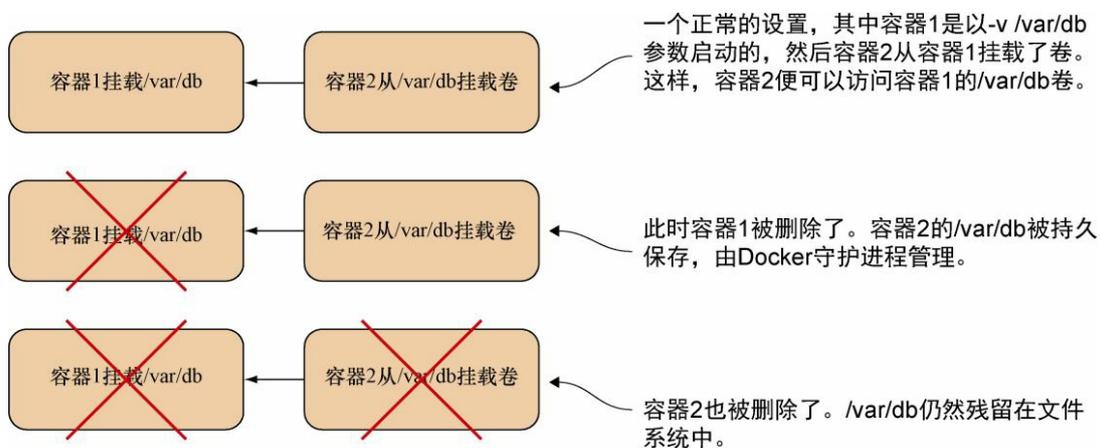


图4-8 当容器被删除时 `/var/db` 下会发生什么

“简单！”你可能会这样想，“在最后一个引用的容器被删除时把卷删掉不就行了！”事实上，Docker 可以采取这种手段，这也是垃圾回收式编程语言从内存中删除对象时所采用的方法：当没有其他对象引用它时，它便可以删除。

但是Docker认为这可能会让人们不小心丢失重要的数据，并且最好把是否在删除容器的时候删除卷的决定权交给用户。这样做带来的一个不幸的副作用便是，默认情况下，卷会一直保留在Docker守护进程所在的宿主机磁盘上，直到它们被手动删除。

如果这些卷填满了数据的话，磁盘可能就会没有存储空间，因此最好关注一下管理这些孤立卷的方法。

问题

孤立的Docker卷挂载到本地用掉了大量的磁盘空间。

解决方案

在调用`docker rm`命令时加上`-v`标志，或者如果忘记的话用一个脚本来销毁它们。

讨论

在图4-8所描述的场景中，如果在调用`docker rm`时总是加上`-v`标志的话便可以确保`/var/db`最后被删除掉。`-v`标志会将那些没有被其他容器挂载的关联卷一一删除。幸好，Docker很聪明，它知道是否有其他容器挂载该卷，因此不会出现什么意外尴尬的情形。

最简单的方式莫过于养成在删除容器时加上`-v`标志这样的好习惯。这样的话可以保留对容器是否删除卷的控制权。

而这种方式的问题在于用户可能会不想每次都删除卷。如果用户正在写大量数据到这些卷的话，极有可能不希望丢失这些数据。此外，如果养成了这样的习惯，很有可能就会变成自动的了，而用户将会在删除某些重要东西之后才反应过来，但已经为时已晚。

在这类情况下，用户将需要使用一个脚本来做这件事情，而为了方便，它也是放到容器里的——Docker化。注意需要用root权限来运行：

```
$ docker run \  
-v /var/run/docker.sock:/var/run/docker.sock \  
-v /var/lib/docker:/var/lib/docker \  
--privileged dockerinpractice/docker-cleanup-volumes
```

(1) 挂载Docker服务器的套接字，这样就可以在容器里调用Docker

(2) 挂载Docker目录这样便可以删除孤立的卷

(3) 升级权限，这样才能有权限执行对孤立卷的删除操作

上述命令将删除任何现有容器不再访问的卷。输出内容看上去会是下面这个样子：

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock \  
-v /var/lib/docker:/var/lib/docker --privileged 951acdb777bf  
  
Delete unused volume directories from /var/lib/docker/volumes  
Deleting 659cfdc5d394ec7ad5942862ba5feb1d24c9f67ca314462207835ef5bf6571  
31  
In use 6ae01c5524267c8f01f1d1e83933b494fdb5c709d9468122b470bfcdd5a5b03d  
  
Deleting 73260d192a0a4d0ebc3606d9daf7137ab220e41cbbfe919ef1dded01a2f37b  
29  
  
Delete unused volume directories from /var/lib/docker/vfs/dir  
Deleting 659cfdc5d394ec7ad5942862ba5feb1d24c9f67ca314462207835ef5bf6571  
31  
In use 6ae01c5524267c8f01f1d1e83933b494fdb5c709d9468122b470bfcdd5a5b03d  
Deleting 73260d192a0a4d0ebc3606d9daf7137ab220e41cbbfe919ef1dded01a2f37b  
29
```

如果怕执行这条命令可能删掉不想删除的东西，可以在执行命令的末尾加上--dry-run以阻止它实际删除任何东西。

恢复数据 如果想要从一个不再被任何容器引用的未删除卷中恢复数据，需要以root用户身份查看/var/lib/docker/volumes文件夹里的内容。

技巧36 解绑容器的同时不停掉它

在使用Docker时，常常会发现打开的是一个交互式shell，然后因为它是容器的主进程，所以一旦退出会话容器便会被终止。幸运的是，有办法可以做到从一个容器解绑（而且，如果愿意的话，可以用`docker attach`命令再连到容器）。

问题

想要解绑一个容器的交互会话，同时不停掉它。

解决方案

按下Ctrl+P然后再按Ctrl+Q来解绑。

讨论

Docker很有建设性地实现了一个不太可能被其他应用使用也不太可能被意外按到的键序列。

假设通过执行`docker run -t -i -p 9005:80 ubuntu /bin/bash`启动了一个容器，然后用`apt-get`安装了一个nginx Web服务器。用户想在宿主机上使用一个快捷的`curl`命令来测试该Web服务器能否被访问到`localhost:9005`。

先按Ctrl+P然后再按Ctrl+Q。注意，不是所有3个键一起按！

运行的容器具有 `-rm` 标志。如果运行时加上 `-rm` 标志，需要按 `Ctrl+C`，一旦按下键序列就返回到终端了。容器会持续在后台运行。

技巧37 使用 DockerUI 来管理 Docker 守护进程

在演示 Docker 时，可能很难展示容器与镜像之间的不同——终端上显示的行并非是可视化的。此外，Docker 的命令行工具对于想要杀掉和删除许多特定容器的需求可能会不太友好。通过创建一个即点即用的在宿主机上管理镜像和容器的工具可以解决这个问题。

问题

想要在宿主机上不通过命令行形式管理容器和镜像。

解决方案

使用 DockerUI。

讨论

DockerUI 是一个由 Docker 的核心开发人员创建的工具，可以在 <https://github.com/crosbymichael/dockerui> 找到和阅读相关的源代码。因为使用它无须任何前置要求，可以跳过这一步，直接去运行它：

```
$ docker run -d -p 9000:9000 --privileged \
-v /var/run/docker.sock:/var/run/docker.sock dockerui/dockerui
```

它会在后台启动一个 `dockerui` 容器。如果现在访问 `http://localhost:9000`，将可以看到一个控制面板，里面展示了电脑上 Docker 的概览信息。

容器管理功能可能是这里面最有一个功能版块了——访问容器页面将会列出正在运行的容器（包括dockerui容器本身），并且它还提供了一个展示全部容器的选项。在这里，用户可以对容器完成一些批量的操作（如杀掉它们）或者点击单个容器名跳转到容器具体的页面，然后完成对该容器的一些单独操作。举个例子，用户可以看到删除一个正在运行的容器的选项。

镜像页面看起来和容器页面差不多，并且它也允许用户选中多个镜像然后完成一些批量操作。点击镜像ID将会出现一些有意思的选项，如基于该镜像创建一个容器和给镜像打个标签。

记住DockerUI可能会落后于Docker官方所提供的功能——如果想要体验到最新和最好的功能，那可能还得被迫用命令行。

技巧38 为Docker镜像生成一个依赖图

Docker的文件分层系统是一个非常强大的理念，它可以节省空间，而且可以让软件的构建变得更快。但是一旦启用了大量的镜像，便很难搞清楚镜像之间是如何关联的。`docker images -a`命令会返回系统上所有镜像层的列表，但是对理解它们的关联关系而言这并不是对用户友好的方式——使用Graphviz，可以很方便地通过创建一个镜像树并做成镜像的形式来可视化镜像之间的关系。

这也展示了Docker在把复杂的任务变得简单方面的强大实力。在宿主机上安装所有的组件来生产镜像时，老的方式可能会包含一长串容易出错的步骤，但是对Docker而言，这就变成了一个不太可能失败的可移植命令。

问题

想要以树的形式将存放在宿主机上的镜像可视化。

解决方案

使用一个我们之前为此任务创建的镜像作为一条单行命令来输出一个PNG或者获取一个Web视图。

讨论

依赖图的生成涉及使用一个我们之前提供的镜像，它里面包含了调用Graphviz来生成PNG图片文件的脚本。运行的命令里需要做的只是挂载Docker服务器套接字然后一切便准备就绪，如代码清单4-9所示。

代码清单4-9 生成一个镜像的分层树

```
$ docker run --rm \ (1)
-v /var/run/docker.sock:/var/run/docker.sock \ (2)
dockerinpractice/docker-image-graph > docker_images.png (3)
```

(1) 在生成镜像之后删除容器

(2) 挂载 Docker 服务器的Unix 域套接字，以便可以在容器里访问Docker服务器。如果已经改了Docker守护进程的默认配置，这不会奏效

(3) 指定一个镜像然后生成一个PNG作为产品

图4-9以PNG形式展示了一台机器的镜像树。从这张图中可以看出，node和golang:1.3镜像拥有一个共同的根节点，然后golang:runtime只和golang:1.3共享全局的根节点。类似地，mesosphere镜像和ubuntu-upstart镜像也是基于同一个根节点构建的。

读者可能会好奇这棵树上的全局根节点是什么。它是一个最小镜像（scratch image），实际上大小为0字节。

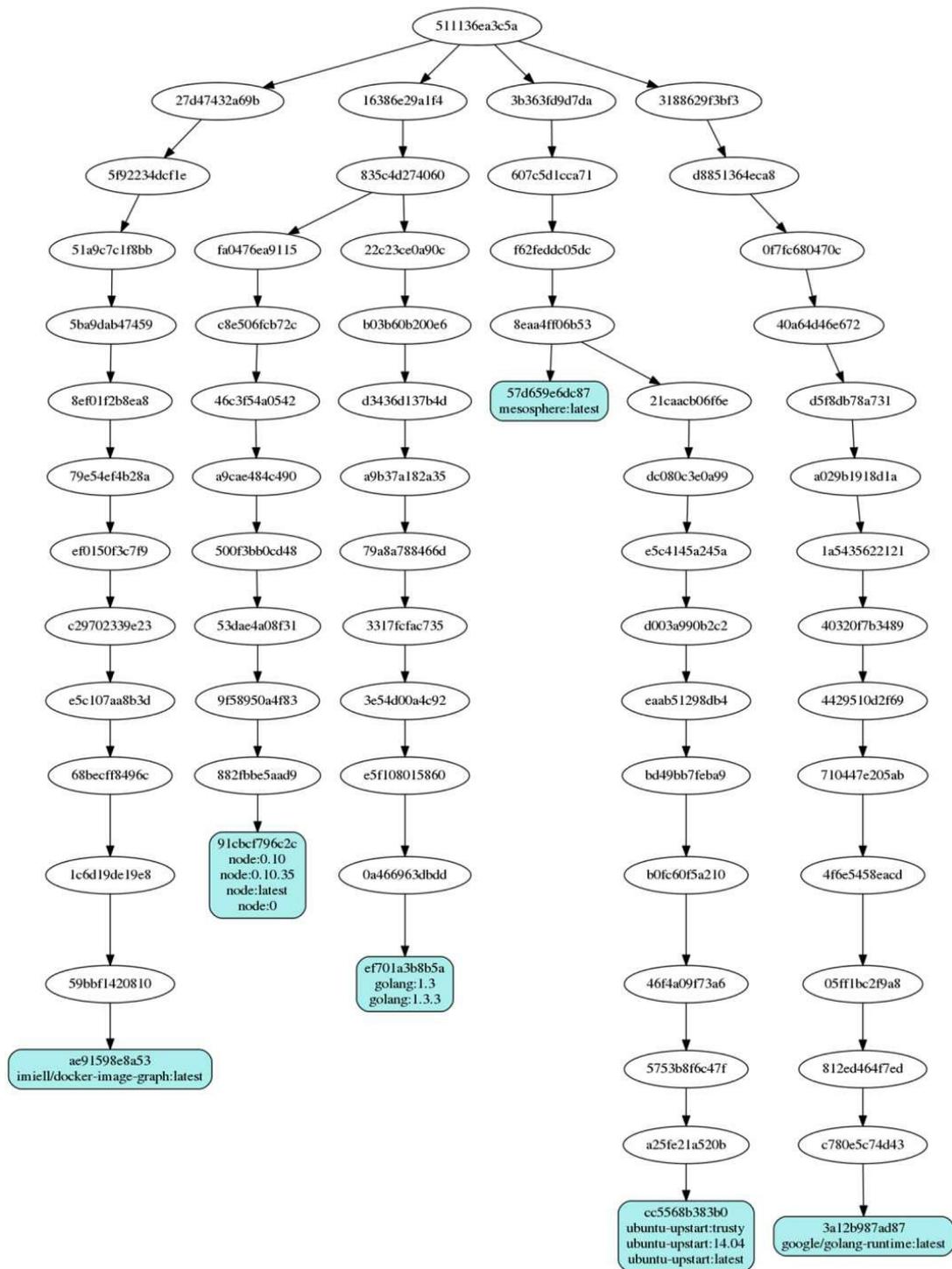


图4-9 一棵镜像树

技巧39 直接操作——对容器执行命令

在Docker早期，许多用户在他们的镜像里添加SSH服务，这样一来便可以从外部通过一个shell来访问它们。Docker不主张这样做，它认为这相当于把容器当成是一台虚拟机（我们知道，容器不是虚拟机），并且为一个本不需要它的系统增加了额外的进程开销。很多人对此持反对意见的原因在于，一旦容器启动了，就没有一个简便的办法进到容器里面。结果，Docker引入了exec命令，它是一个更好地解决干涉和查看启动后容器内部问题的方案。这也是我们这里即将讨论的命令。

问题

想要在一个正在运行的容器里执行一些命令。

解决方案

使用docker exec命令。

讨论

在代码清单4-10中，我们将在后台（附加-d标志）启动一个容器然后告诉它一直休眠（不做任何事情）。我们把这条命令命名为sleeper。

代码清单4-10 运行一个容器，然后在上面执行docker exec命令

```
docker run -d --name sleeper debian sleep infinity
```

现在已经启动了一个容器，可以用Docker的exec命令对它做一些操作。该命令可以看成是有3种基本模式，如表4-3所示。

表4-3 Docker exec模式

模 式	描 述
基本	在命令行上对容器同步地执行命令
守护进程	在容器的后台运行命令
交互	运行命令并允许用户与其交互

我们先介绍基本模式。代码清单4-11给出的命令会在sleeper容器内部运行一个echo命令。

代码清单4-11 从容器里运行echo命令

```
$ docker exec sleeper echo "hello host from container"
hello host from container
```

注意，该命令的结构和docker run命令非常相似，但是代替镜像ID的是我们给定的一个正在运行的容器的ID。echo命令指代的是容器里面的echo可执行文件，而非容器外部的。

守护进程模式会在后台运行该命令，用户将无法在终端看到输出结果。这可能适用于一些常规的清理任务，那些运行后就结束的任务，如代码清单4-12中列出的清理日志文件的任务。

代码清单4-12 在一个容器里删除一周前的日志文件

```
$ docker exec -d sleeper \ (1)
find / -ctime 7 -name '*log' -exec rm {} \; (2)
$ (3)
```

(1) 运行命令时加上-d标志即可在后台以守护进程的形式运行

(2) 删除所有在最近7天没有做过更改并且以log结尾的文件

(3) 无论需要多长时间完成这一操作，该命令都会立即返回

最后，我们来试试交互模式。这种模式允许用户在容器里运行任何喜欢的命令。要启用这一功能，通常需要指定用来在运行时交互的shell，在下面的代码里便是bash：

```
$ docker exec -i -t sleeper /bin/bash
root@d46dc042480f:/#
```

`-i`和`-t`参数和所熟悉的`docker run`做着相同的事情——它们会让命令成为可交互的，然后设置一个TTY设备，以便shell可以正常工作。在运行这一命令后，用户便拿到了一个在容器里面运行的命令提示符。

4.5 小结

在本章中，我们从理论讲到实践。Docker能为日常工作流带来的诸多可能性已经初见端倪。现在，读者不仅了解了Docker的架构，也涉及了日常使用遇到的一些问题及其解决方法。这是坚实的一小步，由此我们打下了从这里到更高阶Docker应用的基础。

在本章中，读者已经学到了以下几项内容：

- 如果需要从容器中获取外部数据，则应该访问卷；
- SSHFS是一种无须额外配置即可访问其他计算机上数据的简单方法；
- 在Docker里运行GUI应用只需要镜像做少量准备即可；
- 构建过程中的缓存是一把双刃剑；

- 可以使用数据容器来抽象数据的存放位置；
- `docker exec`命令才是进入正在运行的容器内部的正确方式——抵制安装SSH。

现在，我们将从每天都可能做的随意的实验性工作转到一个严谨的话题，即Docker的配置管理。

[1] Docker 1.13中已经引入了`system`子命令来完成清理残留的容器、镜像等工作。——译者注

第5章 配置管理——让一切井然有序

本章主要内容

- 使用Dockerfile管理镜像的构建
- 使用传统的配置管理工具来构建镜像
- 管理构建镜像时所需的私密信息
- 缩减镜像的大小，以便更快、更轻量、更安全地交付

配置管理是一门管理运行时环境的艺术，从而使其稳定且可以预测。像Chef和Puppet这样的工具致力于减轻系统管理员维护多台机器的负担。就一定程度而言，Docker带来的软件环境的隔离性和可移植性同样也减轻了这一负担。即便如此，依然需要用配置管理技术来产出Docker镜像，而这也是一个值得关注的重要领域。

随着对Docker的不断了解，这些技巧将为读者提供更多工具来构建镜像以实现想要满足的任何配置需求。

5.1 配置管理和Dockerfile

Dockerfile被认为是构建Docker镜像的标准方式。人们常常会疑惑Dockerfile对于配置管理意味着什么。读者也可能会有许多疑问（尤其是在对一些其他配置管理工具有些经验的时候），例如：

- 如果基础镜像更改会发生什么？
- 如果更改要安装的包然后重新构建会发生什么？
- 它会取代Chef/Puppet/Ansible吗？

事实上，Dockerfile很简单：从给定的镜像开始，Dockerfile为Docker指定一系列的shell命令和元指令，从而产出最终所需的镜像。

Dockerfile提供了一个普通、简单和通用的语言来配置Docker镜像。使用Dockerfile，用户可以使用任何偏好的方式来达到所期望的最终状态。用户可以调用Puppet，可以从其他脚本里复制，甚至可以从一个完整的文件系统复制！

让我们先一起考虑一下如何处理Dockerfile带来的一些小挑战，然后再来讨论我们刚提到的更棘手的那些问题。

技巧40 使用ENTRYPOINT创建可靠的定制工具

Docker允许在任何地方执行命令的潜质意味着在命令行上运行的一些复杂的定制指令或者脚本可以预先配置然后包装到一个打包好的工具。

容易被曲解的ENTRYPOINT指令便是这其中的一个重要部分。读者将会看到它是怎样创建出封装良好、清晰定义以及足够灵活和有价值的Docker镜像来作为工具的。

问题

想要定义容器将会执行的命令，但是将命令的具体参数留给用户。

解决方案

使用Dockerfile的ENTRYPOINT指令。

讨论

作为演示，不妨试想一下企业里有一个这样的简单场景，有一个常规的管理任务是要清理旧的日志文件。通常这很容易出错，人们可能会意外删错东西，因此我们打算使用一个Docker镜像来降低出现问题的风险。

下列脚本（用户应该在保存的时候命名为clean_log）会删除超过特定天数的日志，其中具体天数作为一个命令行选项传入：

```
#!/bin/bash
echo "Cleaning logs over $1 days old"
find /log_dir -ctime "$1" -name '*log' -exec rm {} \;
```

注意，该脚本清理的是/log_dir文件夹下的日志。此文件夹只有在运行时挂载了它才会存在。读者可能还会注意到，这里没有检查是否有参数传给该脚本。这样做的原因我们会在后面介绍此技巧时披露。

现在，让我们在同一目录下创建一个Dockerfile来创建一个镜像，这个Dockerfile包含下面的脚本作为定义好的命令来运行，或者叫入口点（entrypoint）：

```
FROM ubuntu:14.04
ADD clean_log /usr/bin/clean_log (1)
RUN chmod +x /usr/bin/clean_log (2)
ENTRYPOINT ["/usr/bin/clean_log"]
CMD ["7"] (3)
```

- (1) 将之前组织好的clean_log脚本添加到镜像里
- (2) 将此镜像的入口点定义为clean_log脚本
- (3) 设置ENTRYPOINT命令的默认参数（7天）

最佳实践——总是使用数组模式 读者可能会发现，相比于shell形式（`CMD /usr/bin/ command`），我们更喜欢用CMD和ENTRYPOINT的数组形式（如`CMD ["/usr/bin/command"]`）。这是因为，如果是shell形式，它会自动在用户提供的命令前面加上一个`/bin/bash -c`的命令，这可能会导致难以预料的行为。然而，在某些时候shell形式反而会更有用些（见技巧47）。

人们常常费解ENTRYPOINT和CMD之间到底有什么区别。要理解的关键点是，入口点总会在镜像启动之后运行，即使命令被提供给`dockerrun`调用。如果用户尝试传入一条命令，它将会作为参数被传给入口点，然后取代在CMD指令部分定义的默认值。用户只能通过显式地传入一个`-entrypoint`标志给`docker run`命令来重载入口点。

这就意味着，通过`/bin/bash`命令运行镜像将不会提供一个shell，而会将`/bin/bash`作为`clean_log`脚本的参数。

事实上，通过CMD指令定义好默认参数，就意味着不需要再检查是否有传入参数。

下列命令展示了该如何构建和调用此工具：

```
docker build -t log-cleaner .
docker run -v /var/log/myapplogs:/log_dir log-cleaner 365
```

在构建完该镜像后，通过将`/var/log/myapplogs`挂载到脚本将用到的目录，并传入`365`以删除过去一年（而不是一周）的日志文件。

如果有人尝试以不指定天数这样的不正确方式使用镜像的话，他将会得到一条出错消息：

```
$ docker run -ti log-cleaner /bin/bash
```

```
Cleaning logs over /bin/bash days old
find: invalid argument '-name' to '-ctime'
```

这个例子的确相当简单，不过试想一下，一家企业便可以借此跨资源集中管理脚本，以便可以通过一个私有的注册中心维护和安全地分发脚本。

镜像可用 可以在Docker Hub上的dockerinpractice/log-cleaner查看并使用该镜像。

技巧41 在构建中指定版本来避免软件包的漂移

Dockerfile语法简单，功能也有限，它们可以极大地帮助用户理清构建的需求，并且它们可以促进生成镜像的稳定性，但是它们无法确保可重复的构建结果。我们将会探索众多方案中的一个，借以解决这一问题并减少在底层包管理依赖关系改变时带来的意外风险。

这一技巧有助于避免那些“它昨天还运行得好好的”的尴尬处境，如果读者用过经典的配置管理工具就会感同身受。构建Docker镜像与维护一台服务器本质上有很大的不同，但是是一些艰辛的教训依然适用。

仅支持基于**Debian**的镜像 本技巧只适用于基于Debian的镜像，如Ubuntu。

问题

想要确保deb包是自己期望的版本。

解决方案

在一个已验证安装的系统上运行一个脚本来抓取所有依赖软件包的版本，并且获取依赖包的版本。在Dockerfile里安装指定的版本。

讨论

针对版本方面的基本检查可以通过在一个已经验证过的系统上调用apt-cache来完成：

```
$ apt-cache show nginx | grep ^Version:  
Version: 1.4.6-1ubuntu3
```

然后可以像下面这样在Dockerfile里指定版本：

```
RUN apt-get -y install nginx=1.4.6-1ubuntu3
```

这可能已经足以满足需求。但是这无法保证这一版本的nginx所安装的所有依赖都和之前验证的版本一致。可以通过在参数里添加一个--recurse标志来获取所有依赖的信息：

```
apt-cache --recurse depends nginx
```

这一命令的输出内容相当多，因此要获取版本需求的清单也是一件棘手的事情。幸好，我们维护了一个Docker镜像（还有其他的吗？）为用户提供方便。它会输出需要放到Dockerfile里RUN行的内容，以确保所有依赖包的版本都是正确的：

```
$ docker run -ti dockerinpractice/get-versions vim  
RUN apt-get install -y \  
vim=2:7.4.052-1ubuntu3 vim-common=2:7.4.052-1ubuntu3 \  
vim-runtime=2:7.4.052-1ubuntu3 libacl1:amd64=2.2.52-1 \  
libc6:amd64=2.19-0ubuntu6.5 libc6:amd64=2.19-0ubuntu6.5 \  
libgpm2:amd64=1.20.4-6.1 libpython2.7:amd64=2.7.6-8 \  
libselinux1:amd64=2.2.2-1ubuntu0.1 libselinux1:amd64=2.2.2-1ubuntu0.1 \  
libtinfo5:amd64=5.9+20140118-1ubuntu1 libattr1:amd64=1:2.4.47-1ubuntu1 \  
\  
libgcc1:amd64=1:4.9.1-0ubuntu1 libgcc1:amd64=1:4.9.1-0ubuntu1 \  
libpython2.7-stdlib:amd64=2.7.6-8 zlib1g:amd64=1:1.2.8.dfsg-1ubuntu1 \  
\  
libpython2.7:amd64=2.7.6-8
```

```
libpcre3:amd64=1:8.31-2ubuntu2 gcc-4.9-base:amd64=4.9.1-0ubuntu1 \  
gcc-4.9-base:amd64=4.9.1-0ubuntu1 libpython2.7-minimal:amd64=2.7.6-8 \  
mime-support=3.54ubuntu1.1 mime-support=3.54ubuntu1.1 \  
libbz2-1.0:amd64=1.0.6-5 libdb5.3:amd64=5.3.28-3ubuntu3 \  
libexpat1:amd64=2.1.0-4ubuntu1 libffi6:amd64=3.1~rc1+r3.0.13-12 \  
libncursesw5:amd64=5.9+20140118-1ubuntu1 libreadline6:amd64=6.3-4ubuntu  
2 \  
libsqlite3-0:amd64=3.8.2-1ubuntu2 libssl1.0.0:amd64=1.0.1f-1ubuntu2.8 \  
  
libssl1.0.0:amd64=1.0.1f-1ubuntu2.8 readline-common=6.3-4ubuntu2 \  
debconf=1.5.51ubuntu2 dpkg=1.17.5ubuntu5.3 dpkg=1.17.5ubuntu5.3 \  
libnewt0.52:amd64=0.52.15-2ubuntu5 libslang2:amd64=2.2.4-15ubuntu1 \  
vim=2:7.4.052-1ubuntu3
```

在某些时候，用户的构建将会因为版本不再可用而失败。遇到这种情况时可以看看有哪些包做了改动，然后重新检查一下这些改动，确认是否满足特定镜像的需求。

这个例子假定用的镜像是ubuntu:14.04。如果用的是Debian的不同发行版，那么可以fork该仓库并修改Dockerfile里的FROM指令，并构建它。该仓库可以在<https://github.com/docker-in-practice/get-versions.git>找到。

尽管本技巧有助于提升构建的稳定性，但它在安全性方面没有任何建树，因为用户仍然需要从一个无法直接管控的仓库下载软件包。

技巧42 用perl -p -i -e替换文本

使用Dockerfile构建镜像时，在多个文件间替换文本的特定内容的需求并不罕见。有多种方案可以解决这一问题，但是我们这里要介绍的是一个有点儿不太常见的方式，用在Dockerfile里特别方便。

问题

想要在构建期间修改多个文件里的特定行。

解决方案

使用`perl -p -i -e`。

讨论

我们偏好这个命令是有些原因的。

- 与`sed -i`不同（该命令具有类似的语法和效果），这一命令天然支持处理多个文件，即便遇到的问题是修改其中一个文件。这意味着可以在一个目录下加上`'*'`通配符来执行该命令，便不用再担心在包的后续版本里添加目录时它会突然被破坏。
- 和`sed`一样，搜索和替换命令中的正斜杠可以使用其他字符替换。
- 它还很容易记忆（我们不妨称之为“perl pie”命令）。

假定有正则表达式的知识 本技巧假定读者对正则表达式有所了解。如果对正则表达式不熟悉的话，有很多网站可以帮到你。

下面是该命令的一个典型示例：

```
perl -p -i -e 's/127\.0\.0\.1/0.0.0.0/g' *
```

在这条命令里，`-p`标志要求perl循环处理看到的所有行，`-i`标志要求perl即时更新匹配的行内容，而`-e`标志则要求perl把传入的字符串当作一个perl程序处理。`s`是perl的一个指令，用来搜索和替换输入里匹配的字符串。这里的`127.0.0.1`会被替换成`0.0.0.0`。`g`修饰符则确保所有匹配都被更新，而不只是任何给定行里的第一个匹配。最后，星号（*）会使这一文件夹下的所有文件都被更新。

上述命令对Docker容器而言完成的不过是一个很平常的操作。它

会在使用一个地址来监听时，将标准的本地主机IP地址

(**127.0.0.1**) 替换成一个指示“任意”IPv4地址 (**0.0.0.0**)。许多应用会通过仅监听本地主机地址限制成只有该IP地址才能访问，而通常用户会想要在它们的配置文件里将这一配置修改成“任意”地址，因为从宿主机上访问这些应用时，容器就变成了一台外部主机。

不能访问容器里的应用？ 如果一个Docker容器里的应用，即使端口是打开的，用户仍然无法从宿主机上访问的话，不妨尝试一下在应用的配置文件里把监听的地址修改为**0.0.0.0**，并重启应用。这可能是应用拒绝访问所致，因为用户不是从本地主机访问它。在运行镜像时加上**--net=host**（后面技巧97会介绍到）可以帮助验证这一猜测。

perl -p -i -e（和**sed**）还有另外一个很不错的功能，那便是：如果有麻烦的字符转义问题，用户可以使用其他字符替换正斜杠符。下面是一个来自本书作者编写的脚本的真实示例，它在默认的Apache站点文件里添加了一些指令。

这条尴尬的命令

```
perl -p -i -e 's/\\usr\\share\\www\\/\\var\\www\\/html/g' /etc/apache2/ *
```

就变成了

```
perl -p -i -e 's@/usr/share/www@/var/www/html/@g' /etc/apache2/ *
```

在极少数情况下，用户要匹配或替换/和@字符的话可以尝试其他字符，如|或者#。

技巧43 镜像的扁平化

Dockerfile的设计以及它们产出Docker镜像的结果便是，最终镜像里包含了Dockerfile里每一步的数据状态。在构建镜像的过程中，可能需要复制私密信息来确保构建工作可以顺利进行。这些所谓的私密信息可能是SSH密钥、证书或者密码文件等。在提交镜像前删除这些私密信息的话可能不会提供任何实质性的保护，因为它们将出现在最终镜像的更高分层里，而恶意用户则可以轻松地从镜像中提取它们。解决这一问题的其中一个办法便是将得到的镜像扁平化。

问题

想要从镜像的分层历史中移除私密信息。

解决方案

基于该镜像创建一个容器，将它导出再导入，然后给它打上最初镜像ID的标签。

讨论

为了演示这种做法的可用场景，让我们在一个新目录里创建一个简单的Dockerfile，该目录下藏着一个大秘密。运行`mkdir secrets && cd secrets`，然后在该目录里创建一个包含如下内容的Dockerfile:

```
FROM debian
RUN echo "My Big Secret" >> /tmp/secret_key (1)
RUN cat /tmp/secret_key (2)
RUN rm /tmp/secret_key (3)
```

(1) 在构建里放置一个带有一些私密信息的文件

(2) 对该私密文件做一些事情。当前这个Dockerfile只会列出该文件的内容，但是你的Dockerfile可能会SSH到其他服务器或者在镜

像里加密该私密信息

(3) 删除该私密文件

现在运行 `docker build -t mysecret .` 以构建该 Dockerfile 并给它打标签。

一旦它完成构建，可以通过 `docker history` 命令检查得到的 Docker 镜像的分层：

```
$ docker history mysecret (1)
IMAGE          CREATED          CREATED BY
└─ SIZE
55f3c131a35d  25 seconds ago  /bin/sh -c rm /tmp/secret.key
└─ 0 B (2)
5b376ff3d7cd  26 seconds ago  /bin/sh -c cat /tmp/secret_key
└─ 0 B
5e39caf7560f  27 seconds ago  /bin/sh -c echo "My Big Secret" >> /tmp/s
ecre
└─ 14 B (3)
c90d655b99b2  2 weeks ago     /bin/sh -c #(nop) CMD [/bin/bash]
└─ 0 B
30d39e59ffe2  2 weeks ago     /bin/sh -c #(nop) ADD file:3f1a40df75bc56
73ce
└─ 85.01 MB
511136ea3c5a  20 months ago  (4)
└─ 0 B (5)
```

(1) 使用刚创建的镜像的名称运行 `docker history` 命令

(2) 这一分层是删除密钥的地方

(3) 这一分层是添加密钥的地方

(4) 在这一分层添加了 Debian 文件系统。注意，该分层是历史记录里最大的一个

(5) 最开始（空的）分层

现在试想一下用户从一个公开的注册中心下载了这一镜像。他可以检索镜像分层的历史然后运行如下命令列出私密信息：

```
$ docker run 5b376ff3d7cd cat /tmp/secret_key
My Big Secret
```

这里我们运行了一个特定的分层然后将它构造出来，`cat`我们在更高分层里已经删除的那个密钥的内容。正如所见，文件是可以访问的。

至此，用户有一个里面藏有秘密的“危险”容器，我们已经见证了的确可以“黑”到里面的私密信息。要让这个镜像变得安全，需要将该镜像扁平化处理。这意味着用户可以在该镜像里保留相同的数据但是会删除中间分层的信息。为了达成这一目的，需要将该镜像导出为一个简单运行的容器然后再重新导入并给得到的镜像打上标签：

```
$ docker run -d mysecret /bin/true (1)
28cde380f0195b24b33e19e132e81a4f58d2f055a42fa8406e755b2ef283630f
$ docker export 28cde380f | docker import - mysecret (2)
$ docker history mysecret
IMAGE          CREATED          CREATED BY SIZE
fdbeae08751b  13 seconds ago          85.01 MB (3)
```

(1) 运行一个简单的命令让容器可以快速退出，因为并不需要它处于运行状态

(2) 运行`docker export`，把容器ID当作参数并输出文件系统的内容的一个TAR文件。这一操作被管道到`docker import`，它会以TAR文件的内容作为输入，基于这些内容创建一个镜像

(3) `docker history`的输出现在只显示最后一组文件的分层

传给`docker import`命令的`-`参数指明用户想要从命令的标准输

入读取TAR文件内容。`docker import`的最后一个参数指明该如何给导入的镜像打标签。在这个例子里它会覆盖之前的标签。

由于现在镜像里只有一个分层，因此就没有藏着秘密的分层记录。现在再也无法从镜像提取任何秘密了。

技巧44 用alien管理外来软件包

本书里（还有互联网上）绝大部分Dockerfile示例使用的都是基于Debian的镜像，而软件开发的现实决定了许多人不会专门做这些打包的事情。

好在有现成的工具可以帮助用户实现这一点。

问题

想要安装一个外来的发行版的软件包。

解决方案

使用一个基于alien的Docker镜像转换软件包。

讨论

alien是一款命令行工具，是专为转换不同格式的软件包文件设计的，如表5-1所示。我们不止一次遇到过需要让外来软件包管理系统下的软件包正常工作，例如，.deb用在centos中，.rpm文件用在非Red Hat系的系统。

表5-1 alien支持的包格式

扩展名	描述

.deb	Debian包
.rpm	Red Hat包管理
.tgz	Slackware Gzip压缩的TAR文件
.pkg	Solaris pkg包
.slp	Stampede包

不涉及**Solaris**包和**Stampede**包 出于本技巧的初衷，没有完全覆盖到Solaris包和Stampede包。Solaris要求安装Solaris特有的软件，而Stampede则是一个已经废弃的项目。

在本书的研究过程中我们发现，在非Debian系的发行版上安装alien可能有些费劲儿。这是一本Docker书，我们自然决定以Docker镜像的形式提供一个转换工具。作为一个小福利，这一工具用到了技巧40中介绍的ENTRYPOINT命令，让用户可以更加便利地使用它。

举个例子，让我们来看看eatmydata这个软件包，在技巧56里会用到它：

```
$ mkdir tmp && cd tmp (1)
$ wget \
http://mirrors.kernel.org/ubuntu/pool/main/libe/libeatmydata/
➤ eatmydata_26-2_i386.deb (2)
$ docker run -v $(pwd):/io dockerinpractice/alienate (3)
Examining eatmydata_26-2_i386.deb from /io
eatmydata_26-2_i386.deb appears to be a Debian package (4)
eatmydata-26-3.i386.rpm generated
eatmydata-26.slp generated
eatmydata-26.tgz generated
```

```

=====
/io now contains:
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
eatmydata_26-2_i386.deb
=====
$ ls -l (5)
eatmydata_26-2_i386.deb
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz

```

(1) 创建一个空的工作目录

(2) 获取想要转换的包文件

(3) 运行dockerinpractice/alienate镜像，将当前目录挂载到容器的/io路径下。容器会检查该目录，尝试转换找到的任意有效文件

(4) 容器通知用户它运行Alien包装脚本时的行为

(5) 文件已经被转换为RPM、Slackware tgz和Stampede文件

或者，也可以直接将URL传给docker run命令：

```

$ mkdir tmp && cd tmp
$ docker run -v $(pwd):/io dockerinpractice/alienate \
http://mirrors.kernel.org/ubuntu/pool/main/libe/
➤ libeatmydata/eatmydata_26-2_i386.deb
wgetting http://mirrors.kernel.org/ubuntu/pool/main/libe/
➤ libeatmydata/eatmydata_26-2_i386.deb
--2015-02-26 10:57:28-- http://mirrors.kernel.org/ubuntu/pool/main/libe/
/
➤ libeatmydata/eatmydata_26-2_i386.deb
Resolving mirrors.kernel.org (mirrors.kernel.org)...
➤ 198.145.20.143, 149.20.37.36, 2001:4f8:4:6f:0:1994:3:14, ...
Connecting to mirrors.kernel.org (mirrors.kernel.org)
➤ |198.145.20.143|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7782 (7.6K) [application/octet-stream]
Saving to: 'eatmydata_26-2_i386.deb'

0K .....                               100% 2.58M=0.003s

```

```
2015-02-26 10:57:28 (2.58 MB/s) - 'eatmydata_26-2_i386.deb' saved [7782 /7782]
```

```
Examining eatmydata_26-2_i386.deb from /io
eatmydata_26-2_i386.deb appears to be a Debian package
eatmydata-26-3.i386.rpm generated
eatmydata-26.slp generated
eatmydata-26.tgz generated =====
=====
/io now contains:
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
eatmydata_26-2_i386.deb =====
=====
$ ls -l
eatmydata_26-2_i386.deb
eatmydata-26-3.i386.rpm
eatmydata-26.slp
atmydata-26.tgz
```

如果想在自己的容器里运行alien，可以通过这个命令启动容器：

```
docker run -ti --entrypoint /bin/bash dockerinpractice/alienate
```

不确保一定奏效 alien这款工具的定位是“尽力而为”，它并不保证一定能够将提供的包转换成功。

技巧45 把镜像逆向工程得到Dockerfile

用户可能会发现自己遇到这样的场景：有人用一个Dockerfile创建了一个可以访问的镜像，但最初的Dockerfile却遗失了。我们发现自己不止一次遇到过这种情况。能恢复构建步骤相关信息可以避免冗长的自己重新探索的过程。

问题

有一个镜像，想要逆向工程得到最初的Dockerfile。

解决方案

使用`docker history`命令并通过查看分层的方式尝试确定更改过的地方。

讨论

尽管一个Docker镜像不是每次都能完全逆向工程，但是倘若它本质上是通过一个Dockerfile创建出来的，这会是一个很好的弄清它是怎样装配到一起的机会（这是一个机会，很有希望借此按需重建镜像）。

在本技巧中我们打算采用如下Dockerfile作为示例。我们尽量用最少的步骤覆盖尽可能多的不同类型的指令。读者会去构建这个Dockerfile，然后运行一个简单的shell命令来了解这一技巧是如何工作的，最终我们会看一个更接近容器化的解决方案。

```
FROM busybox
MAINTAINER ian.miell@gmail.com
ENV myenvname myenvvalue
LABEL mylabelname mylabelvalue
WORKDIR /opt
RUN mkdir -p copied
COPY Dockerfile copied/Dockerfile
RUN mkdir -p added
ADD Dockerfile added/Dockerfile
RUN touch /tmp/afile
ADD Dockerfile /
EXPOSE 80
VOLUME /data
ONBUILD touch /tmp/built
ENTRYPOINT /bin/bash
CMD -r
```

LABEL指令可能无法工作 在编写本书期间LABEL还是Docker相对较新的一条指令，因此在用户的安装版本里可能还没有这一指令。

首先，用户需要构建这个示例镜像，将得到的镜像命名为reverseme:

```
$ docker build -t reverseme .
```

1. shell解决方案

基于shell的实现方案所需的大部分指令都在这里，与接下来的容器化解决方案相比不那么完备。举个例子，这一方案使用docker inspect命令来提取元数据。

jq是必需的 要运行这一解决方案，用户需要安装jq程序。它允许用户查询和操作JSON数据。

```
docker history reverseme | \ (1)
awk '{print $1}' | \ (2)
grep -v IMAGE | \ (3)
tac | \ (4)
sed "s/\(.*\) /docker inspect \1 | \ (5)
jq -r \'.[0].ContainerConfig.Cmd[2] | tostring\''/" | \
sh | \ (6)
sed 's/^#(nop) //' (7)
```

- (1) 检索组成指定镜像的分层
- (2) 从docker history的输出里检索出每一分层的镜像ID
- (3) 排除标题那行（带有“IMAGE”字样的这一行），因为它是不相关的
- (4) 将列出的镜像翻转回Dockerfile的顺序（tac是cat的反向）
- (5) 拿到镜像ID然后构造出一条命令，运行它的话可以检索

Docker镜像分层元数据里记录的执行命令。将docker inspect的输出内容管道到一个jq命令，它会提取出用来创建镜像分层的具体命令，像Docker元数据里已经记录的那样

(6) 执行由sed命令输出的docker inspect命令

(7) 输出那些不会更改文件系统的指令——它们都带有#(nop)前缀

输出内容看上去将会是下面这样的：

```
MAINTAINER Jérôme Petazzoni <jerome@docker.com>
ADD file:8cf517d90fe79547c474641cc1e6425850e04abbd8856718f7e4a184ea8785
38 in /
CMD ["/bin/sh"]
MAINTAINER ian.miell@gmail.com
ENV myenvname=myenvvalue
WORKDIR /opt
mkdir -p copied
COPY file:d0fb99565b15f8dfec37ea1cf3f9c4440b95b1766d179c11458e31b5d08a2
ced in
➤ copied/Dockerfile
mkdir -p added
ADD file:d0fb99565b15f8dfec37ea1cf3f9c4440b95b1766d179c11458e31b5d08a2c
ed in
➤ added/Dockerfile
touch /tmp/afile
ADD file:d0fb99565b15f8dfec37ea1cf3f9c4440b95b1766d179c11458e31b5d08a2c
ed in /
COPY dir:9cc240dcc0e31ce1b68951d230ee03fc6d3b834e2ae459f4ad7b7d023845e8
34 in /
COPY file:97bc58d5eaefdf65278cf82674906836613be10af02e4c02c81f6c8c7eb44
868 in /
EXPOSE 80/tcp
VOLUME ["/data"]
ONBUILD touch /tmp/built
ENTRYPOINT ["/bin/sh -c /bin/bash"]
CMD ["/bin/sh -c -r]
```

上述输出看上去和我们最初的Dockerfile大同小异。FROM命令被3条用来创建BusyBox镜像分层的命令取代。ADD和COPY命令指向了一

个校验和以及文件被解压的位置。这是因为最开始的构建上下文不会在元数据里保存。最后，**CMD**和**ENTRYPOINT**指令被修改成规范的方括号形式。

什么是构建上下文 Docker构建上下文是指docker build命令里传入的目录的位置及目录里的一组文件。该上下文也是Docker在Dockerfile里用来查找ADD或者COPY对应文件的地方。

因为缺少构建上下文会导致**ADD**和**COPY**指令无效，所以这个Dockerfile不能按照原来的效果运行。但是，如果用户在其逆向工程得到的Dockerfile里也有这些内容的话，他就必须尝试去找出从上下文信息里添加的那一个或一组文件。举个例子，在上面的输出中，如果用户以reverseme镜像运行起一个容器，然后再查看复制的/Dockerfile，应该就可以提取所需的文件并将它添加到新的构建上下文中。

2. 容器化的解决方案

尽管前面的方案是一个很有用而且很有指导意义的获取用户感兴趣的镜像相关信息的方式（并且比较容易根据自己需求进行修改），但是还有一种更为优雅的方式可以实现相同的效果——这可能也是一种更容易维护的方式。作为一个小福利，这一解决方案也为你提供了（如果可以的话）一条和用户最初那个Dockerfile中的**FROM**类似的**FROM**命令：

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock \
  dockerinpractice/dockerfile-from-image reverseme
FROM busybox:buildroot-2014.02
MAINTAINER ian.miell@gmail.com
```

```
ENV myenvname=myenvvalue
WORKDIR /opt
RUN mkdir -p copied
COPY file:43a582585c738bb8fd3f03f29b18caaf3b0829d3ceb13956b3071c5f0befc
bfc \
in copied/Dockerfile
RUN mkdir -p added
ADD file:43a582585c738bb8fd3f03f29b18caaf3b0829d3ceb13956b3071c5f0befc
bfc \
in added/Dockerfile
RUN touch /tmp/afile
ADD \
file:43a582585c738bb8fd3f03f29b18caaf3b0829d3ceb13956b3071c5f0befc
bfc in /
EXPOSE 80/tcp
VOLUME [/data]
ONBUILD touch /tmp/built
ENTRYPOINT [/bin/sh -c /bin/bash]
CMD [/bin/sh -c -r]
```

这一技巧真的是解决了我们工作中的好几个痛点！

仅适用于**Dockerfile**创建的镜像。如果镜像是用Dockerfile正确创建出来，这一技巧应该能够按描述的方式工作。倘若镜像是手工制作然后再提交的，镜像之间的差异将不会出现在镜像的元数据里。

5.2 传统配置管理工具与Docker

现在我们将继续介绍Dockerfile如何与更传统的配置管理工具一起工作。

我们在这里一起看看使用make的传统配置管理，展示如何使用现有的Chef脚本通过Chef Solo来置备镜像，以及使用一个shell脚本框架来帮助不精通Docker的用户构建镜像。

技巧46 传统方式：搭配make和Docker

有时候，用户可能会发现有些Dockerfile限制了自己的构建流程。举个例子，如果限制自己运行docker build命令，就无法产生任何输出文件，并且无法在Dockerfile中定义变量^[1]。

这种附加工具化的需求可以通过一些工具（包括纯shell脚本）来实现。在这一技巧里，我们将一起来看看可以怎样结合老牌的make工具与Docker一起工作。

问题

想要在docker build执行过程中增加额外的任务。

解决方案

在make里封装镜像的创建。

讨论

为防用户之前没有make使用经验，我们在这里对它进行一些简单的介绍，make是一款工具，它需要一个或多个输入文件并会产出一个输出文件，但是它也可以用作一个任务运行器。下面是一个简单的示例（注意所有缩进都必须是制表符）：

```
.PHONY: default createfile catfile (1)
default: createfile (2)
createfile: x.y.z (3)
catfile: (4)
    cat x.y.z
x.y.z: (5)
    echo "About to create the file x.y.z"
    echo abc > x.y.z
```

(1) 默认情况下，make会假定所有目标均是被任务创建出来

的文件名，使用.PHONY表明这不是任务的真正名称

(2) 按照惯例，Makefile中的第一个目标是default。如果在运行的时候没有指定一个明确的目标，make将会选取文件中的第一个目标。可以看到，因为createfile是default的唯一依赖，default将会执行它

(3) createfile是一个伪任务，它依赖x.y.z任务

(4) catfile是一个伪任务，它运行单条命令

(5) x.y.z是一个文件任务，会运行两条命令并创建目标x.y.z文件

空距很重要 一个Makefile里的所有缩进都必须是制表符，并且目标里的每条命令都是在不同的shell里运行的（所以环境变量不会被传递过去）。

一旦在名为Makefile的文件中定义了上述内容，便可以使用像make createfile这样的命令去调用任意目标。

现在我们可以查看一些有用的模式——接下来要讨论的目标都将是伪任务，因为它很难（尽管可以）通过追踪文件的变动来自动触发Docker构建。Dockerfile会对镜像分层进行缓存，因此构建往往会很快。

第一步就是运行一个Dockerfile。因为Makefile是由shell命令组成，所以这一点很容易办到：

```
base:
```

```
docker build -t corp/base .
```

上述命令做的工作带来的一些正常变动正是用户所期许的结果（例如，将文件通过管道传递给`docker build`以去掉上下文，或是用`-f`指定采用不同命名的Dockerfile），而且用户可以使用`make`的依赖功能，在必要时自动构建基础镜像（在FROM中使用的那个）。例如，如果用户在一个叫`repos`的子目录下迁出几个仓库（这样也容易做`make`），用户可以像下面这样添加一个目标：

```
app1: base
    cd repos/app1 && docker build -t corp/app1 .
```

这样做的缺点是，每当基础镜像需要重新构建，Docker就需要上传一个包含所有依赖仓库的构建上下文。可以通过显式地传入一个作为构建上下文的TAR文件给Docker来解决这一问题：

```
base:
    tar -cvf - file1 file2 Dockerfile | docker build -t corp/base .
```

如果用户目录内包含大量与构建无关的文件，那么这种依赖的显式声明语句将会带来一个显著的速度方面的提升。如果用户想要将所有构建依赖保留在不同的目录里，可以稍微修改一下这个目标：

```
base:
    tar --transform 's/^deps\\/' -cf - deps/ * Dockerfile | \
    docker build -t corp/base .
```

在这里，用户可以将`deps`目录下的所有内容添加到构建上下文中，然后使用`--transform`选项压缩tar包（Linux上的最新tar版本支持），这样便可以从文件名中除去任何前导“`deps/`”。在这个例子里，更好的办法是将`deps`和Dockerfile放在各自的目录中以允许正常的`docker build`，但是了解这种高级用法很有价值，因为它可以在

一些最不可能的地方派上用场。在使用这一方案之前往往要考虑清楚，毕竟它会增加构建流程的复杂度。

简单的变量替换是一件相对简单的事情，但是（如之前的`--transform`）在使用它之前还是得考虑清楚——`Dockerfile`之所以故意不支持变量，就是为了保持构建是易于重现的。这里我们将用到传给`make`的一些变量，然后使用`sed`替换，不过用户也可以按照自己的喜好来传参和替换：

```
VAR1 ?= defaultvalue
base:
  cp Dockerfile.in Dockerfile
  sed -i 's/{VAR1}/${VAR1}/' Dockerfile
  docker build -t corp/base .
```

`Dockerfile`将在每次基础目标运行时被重新生成，而且用户可以通过添加更多的`sed -i`条目添加更多的变量替换。要覆盖`VAR1`的默认值，可以执行`make VAR1 = newvalue base`。倘若变量里面包含斜杠的话，用户可能需要另外指定一个`sed`分隔符，如`sed -i 's#VAR1}#$(VAR1) #'Dockerfile`。

最后，如果用户一直使用`Docker`作为构建工具，那便需要知道怎样才能从`Docker`中获取文件。我们将介绍几种不同的可选方案，具体取决于实际用例场景：

```
singlefile: base
  docker run --rm corp/base cat /path/to/myfile > outfile
multifile: base
  docker run --rm -v $(pwd)/outdir:/out corp/base sh \
  -c "cp -r /path/to/dir/ * /out/"
```

在这里，`singlefile`对一个文件执行`cat`然后管道输出到一个新文件。这个方案具有自动设置正确的文件拥有者的优点，但是对于

多个文件的处理就会变得很麻烦。推荐的多文件方案则是在容器里挂载一个卷并将所有文件从一个目录复制到该卷。用户可以使用`chown`命令来设置文件的真正拥有者，但是别忘了在调用时可能需要带上`sudo`。

Docker项目本身从源代码构建Docker时便是用的挂载卷的方案。

技巧47 借助Chef Solo构建镜像

Docker新手们常常疑惑的一件事情便是，Dockerfile是否是唯一被支持的配置管理工具，以及现有配置管理工具是否应该移植到Dockerfile。这些观点都是不对的。

尽管Dockerfile被设计成是一种简单、可移植的镜像置备手段，但是它也足够灵活，允许任何其他配置管理工具接管。简而言之，如果可以在终端里运行它，便可以在Dockerfile中运行它。

这里作为演示，我们将展示如何在Dockerfile中启动并运行Chef（可能是最成熟的配置管理工具）。使用像Chef这样的工具可以减少配置镜像的工作量。

问题

想要通过使用Chef来减少配置工作。

解决方案

在容器里安装Chef然后运行recipe来置备它。

讨论

在这个示例里，我们将使用Chef置备一个简单的“Hello

World!”Apache网站。通过这个例子可以给读者直观的感受，即Chef在配置管理方面能做些什么。

针对这个例子，我们将使用Chef Solo，它不需要配置外部的Chef服务器。如果读者对Chef很熟悉的话，这个例子可以很容易适配现有脚本。

我们将演示这一Chef示例的创建过程，但如果想要得到可执行代码，这里有一个Git仓库可供下载。要下载它的话，运行下面这条命令：

```
git clone https://github.com/docker-in-practice/docker-chef-solo-example.git
```

我们将从一个小目标做起，利用Apache设置一个一访问它便输出“Hello World!”的Web服务器。该站点工作在mysite.com下，而且在镜像上会设置一个mysiteuser用户。

首先，创建一个目录并设定好Chef配置所需的文件：

```
$ mkdir chef_example
$ cd chef_example
$ touch attributes.json (1)
$ touch config.rb (2)

$ touch Dockerfile (3)
$ mkdir -p cookbooks/mysite/recipes (4)
$ touch cookbooks/mysite/recipes/default.rb
$ mkdir -p cookbooks/mysite/templates/default (5)
$ touch cookbooks/mysite/templates/default/message.erb
```

(1) Chef的属性文件，它定义了这个镜像（或者用Chef的说法是节点）的一些变量，包含这个镜像的执行列表里的recipe，以及其他的一些信息

- (2) Chef的配置文件，它设置一些Chef配置相关的基础变量
- (3) 将用来构建镜像的Dockerfile
- (4) 创建默认的recipe文件夹，在这里面保存构建该镜像的Chef指令
- (5) 针对动态配置的内容创建一些模板

attributes.json的内容如代码清单5-1所示。

代码清单5-1 attributes.json

```
{
  "run_list": [
    "recipe[apache2::default]",
    "recipe[mysite::default]"
  ]
}
```

上述文件列出了要运行的recipe。apache2 recipe将会从一个公共仓库获取，而mysite recipe则在本地编写。

接下来，config.rb里放了一些基础信息，如代码清单5-2所示。

代码清单5-2 config.rb

```
base_dir      "/chef/"
file_cache_path base_dir + "cache/"
cookbook_path base_dir + "cookbooks/"
verify_api_cert true
```

上述文件设置了相关位置的一些基础信息，并添加了配置参数verify_api_cert来去掉不相干的错误。

至此，我们终于收获了劳动成果——该镜像的Chef recipe。代码

块里每一个由end结尾的小节定义了一个Chef资源（见代码清单5-3）。

代码清单5-3 cookbooks/mysite/recipes/default.rb

```
user "mysiteuser" do (1)
  comment "mysite user"
  home "/home/mysiteuser"
  shell "/bin/bash"
  supports :manage_home => true
end

directory "/var/www/html/mysite" do (2)
  owner "mysiteuser"
  group "mysiteuser"
  mode 0755
  action :create
end

template "/var/www/html/mysite/index.html" do (3)
  source "message.erb"
  variables(
    :message => "Hello World!"
  )
  user "mysiteuser"
  group "mysiteuser"
  mode 0755
end

web_app "mysite" do (4)
  server_name "mysite.com"
  server_aliases ["www.mysite.com", "mysite.com"] (5)
  docroot "/var/www/html/mysite"
  cookbook 'apache2'
end
```

(1) 创建一个用户

(2) 创建一个目录来放置Web内容

(3) 定义了一个将会放到 Web 文件夹下的文件。根据source属性中定义的模板创建该文件

(4) 为apache2定义一个Web应用

(5) 在真实场景里，用户必须将这些引用从mysite改成自己的站点名称。如果用户是在自己的宿主机上访问或测试的话那就没问题了

网站的内容包含在模板文件里。Chef会去读取其中一行，如代码清单5-4所示，将其替换成来自config.rb的“Hello World!”消息，然后再将替换后的文件写到模板目标（/var/www/html/mysite/index.html）。这个例子中用到的模板语言在这里不做详细介绍。

代码清单5-4 cookbooks/mysite/templates/default/message.erb

```
<%= @message %>
```

最后，将所有内容和Dockerfile放到一起，它会设置Chef的前置要求然后运行Chef来配置镜像，如代码清单5-5所示。

代码清单5-5 Dockerfile

```
FROM ubuntu:14.04

RUN apt-get update && apt-get install -y git curl

RUN curl -L \
https://opscode-omnibus-packages.s3.amazonaws.com/
➤ ubuntu/12.04/x86_64/chefdk_0.3.5-1_amd64.deb \
-o chef.deb
RUN dpkg -i chef.deb && rm chef.deb (1)
COPY . /chef (2)

WORKDIR /chef/cookbooks
RUN knife cookbook site download apache2 (3)
RUN knife cookbook site download iptables
RUN knife cookbook site download logrotate

RUN /bin/bash -c 'for f in $(ls *gz); do tar -zxf $f; rm $f; done' (4)

RUN chef-solo -c /chef/config.rb -j /chef/attributes.json (5)
```

```
CMD /usr/sbin/service apache2 start && sleep infinity (6)
```

(1) 下载并安装Chef。如果这一下载方式不起作用，可以检查之前在这一技巧的讨论中提到的docker-chef- solo-example的最新代码，因为这里可能需要更新的版本

(2) 将所在文件夹下的内容复制到镜像上的/chef 文件夹

(3) 切换到cookbooks文件夹然后使用Chef的knife工具下载apache2 cookbook及相关依赖的压缩包

(4) 解压下载完的压缩包然后删除它们

(5) 运行chef命令配置镜像。把事先创建好的属性和配置文件传给它

(6) 定义镜像默认的启动命令。sleep infinity命令可以确保容器不会在service命令完成任务后立刻退出

现在可以构建并运行镜像：

```
docker build -t chef-example .  
docker run -ti -p 8080:80 chef-example
```

如果读者现在浏览<http://localhost:8080>，应该能看到“Hello World!”的字样。

Docker Hub访问超时 如果Chef构建需要很长时间，而且用的是Docker Hub工作流的话，构建可能会出现超时。如果发生这种情况，用户可以在一台受自己管控的机器上完成构建，为支持的服务买单，或者是把构建步骤拆分成更小的单元，这样一来Dockerfile里每个单独步骤返回的时间便会更短。



这个例子虽然很简单，但是使用这一方案的好处是显而易见的。通过一些相对明了的配置文件，将镜像转换成所需状态的具体细节交由配置管理工具处理。这并不意味着可以忘记配置的细节，更改变量值的话还是要求理解其语义的，这样可以确保不会把事情搞砸。但是，这种方法的确可以节省很多的时间和精力，特别是那些不需要了解太多细节的项目。

技巧48 从源到镜像的构建

我们已经介绍了构建Docker镜像的几种方式，但是唯一一个从头设计来利用Docker的功能的便是Dockerfile。然而，这里还有一些可供选择的替代方案，它们可以让那些对Docker不感兴趣的开发者从中解脱出来，或是可以给构建过程提供更强大的能力。

问题

想要给自己的用户提供一种手段，使他们无须了解Docker便可以创建出一个Docker镜像。

解决方案

使用Red Hat的Source to Image（S2I或者STI）框架来构建Docker镜像。

讨论

Source to Image是通过将源代码存放到一个单独定义的负责构建镜像的Docker镜像来创建所需Docker镜像的一种手段。

读者可能想知道为什么会想出这么一种构建方法。主要原因便是

它允许应用开发人员对自己的代码进行修改而无须关心Dockerfile甚至Docker镜像的细节。如果镜像交付到了一个aPaaS（应用程序平台即服务）平台，个别工程师不需要懂Docker也可以为项目贡献代码。这在企业环境里相当有用，在那里有大群人专注于特定领域，而且不用直接关注他们项目的构建过程。

S2I也称为**STI** 从源到镜像的构建方法在源代码和文档里有两个众所周知的名字：最开始是STI，新的名字是S2I。它们是一回事。

图5-1在核心轮廓里展示了S2I的工作流。

一旦流程建立起来，工程师们只需要专注他们希望对源代码做出的变更，致力于将它推广到不同的环境。其他一切事情则交由启动该流程的**sti**工具驱动。

1. 额外收益

这一方案的优点体现在以下几个方面。

- 灵活性——这个过程可以很容易地嵌到任何现有的软件交付流程中，而且它几乎可以使用任意Docker镜像作为它的基础镜像层。
- 速度——这种构建方法可以比Dockerfile构建更快，因为任意数量的复杂操作都可以添加到构建过程中而无须在每个步骤创建出一个新的分层。S2I还使用户能够在构建之间复用组件以节省时间。
- 业务解耦——由于源代码和Docker镜像均是清晰且明确分离的，开发人员可以专注于代码而基础架构团队则专注于Docker镜像及

其交付。因为基础的底层镜像和代码是分离的，一些升级和补丁也更容易交付。

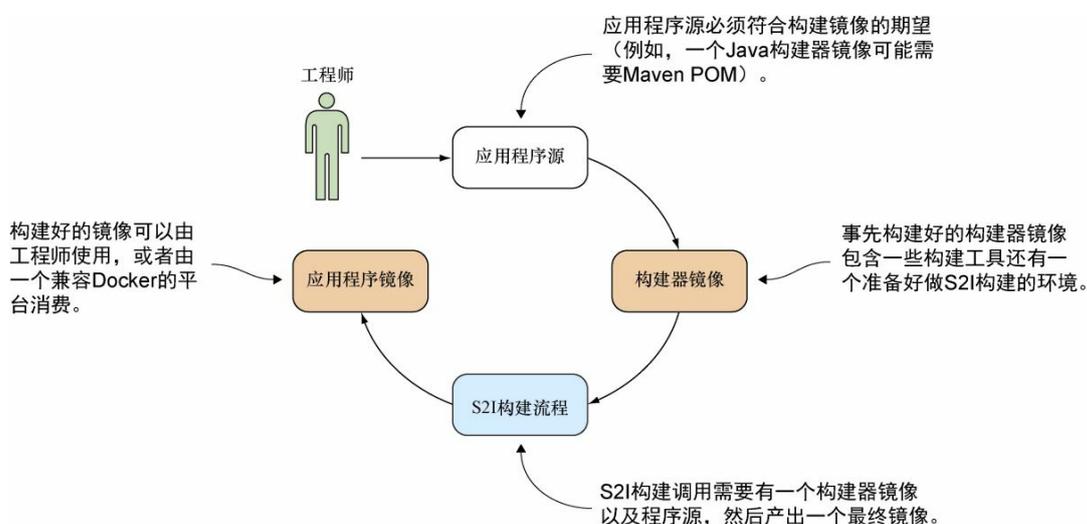


图5-1 从源到镜像的工作流

- 安全性——这个流程可以将构建中执行的操作限制为特定用户，而Dockerfile则允许以root身份运行任意命令。
- 生态系统——此框架的结构允许用户建立一个镜像和代码分离模式的共享生态系统，这更易于大型环境的运维。

本技巧将展示如何完成镜像构建这样的模式，尽管它很简单而且在某些方面有所限制。我们的应用模式将包含：

- 包含一个shell脚本的源代码；
- 一个用来创建镜像的构建器，该镜像会获取shell脚本，使它可运行，然后运行它。

2. 创建自己的S2I镜像

创建自己的S2I镜像需要以下几个步骤：

- (1) 启动一个S2I开发环境；
- (2) 创建自己的Git项目；
- (3) 创建构建器镜像；
- (4) 构建应用程序镜像。

一旦镜像被创建出来，对其做修改和重新构建就很简单了。

3. 启动一个S2I开发环境

为了帮助确保建立一种一致的体验，用户可以使用一个受维护的环境来开发自己的S2I构建镜像和项目：

```
$ docker run-ti \  
-v /var/run/docker.sock:/var/run/docker.sock \  
dockerinpractice/shutit-s2i
```

- (1) 确保宿主机的Docker守护进程在容器里可用（见技巧25）
- (2) 使用一个受维护的S2I构建环境

有问题？启用了**SELinux**？ 如果读者使用的是一个开启了SELinux的环境，那么在容器里执行Docker可能会遇到问题。详情见技巧88。

4. 创建自己的Git项目

用户可以使用一个在其他地方构建然后放到GitHub上的Git项

目，但是为了保证这个例子的简单性和独立性，我们会在S2I开发环境本地创建一个项目。正如前面所讲，源代码里包含一个向终端输出“Hello World”的shell脚本：

```
mkdir /root/myproject
cd /root/myproject
git init
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
cat > app.sh <<< "echo 'Hello World'"
git add .
git commit -am 'Initial commit'
```

5. 创建构建器镜像

为了创建构建器镜像，我们打算用sti创建一个脚手架然后在它上面做修改：

```
sti create sti-simple-shell /opt/sti-simple-shell
cd /opt/sti-simple-shell
```

这条S2I命令会创建出若干文件。为了让我们的 workflow 能够运行起来，我们要重点编辑如下文件：

- Makefile;
- Dockerfile;
- .sti/bin/assemble;
- .sti/bin/run。

先拿Dockerfile来说，将其内容修改成下面这些：

```
FROM openshift/base-centos7 (1)
RUN chown -R default:default /opt/openshift (2)
COPY ./sti/bin /usr/local/sti (3)
RUN chmod +x /usr/local/sti/* (4)
USER default (5)
```

(1) 使用标准的OpenShift基础centos7镜像。它里面事先创建好了一个默认用户

(2) 将默认的Openshift代码位置的拥有者修改成默认用户

(3) 将S2I脚本复制到一个S2I构建的默认位置

(4) 确保S2I脚本是可执行的

(5) 让构建器镜像默认情况下使用事先创建好的默认用户

接下来需要创建汇编脚本，它负责获取源代码并对其进行编译以便可以运行。下面这个bash脚本是一个可供使用的版本，尽管有所简化但是功能俱全：

```
#!/bin/bash -e (1)
if [ "$1" = "-h" ]; then (2)
    exec /usr/local/sti/usage
fi
if [ "$(ls /tmp/artifacts/ 2>/dev/null)" ]; then (3)
    echo "---> Restoring build artifacts"
    mv /tmp/artifacts/ * ./
fi
echo "---> Installing application source" (4)
cp -Rf /tmp/src/. ./
echo "---> Building application from source" (5)
chmod +x /opt/openshift/src/app.sh
```

(1) 作为一个bash脚本运行并且一旦失败就退出

(2) 如果传入了usage标志就输出使用帮助

(3) 可以的话，从先前的构建中恢复任何已经保存的组件

(4) 将应用程序源安装到默认目录中

(5) 从源代码构建应用程序。在这个例子里，构建即是简单的给app.sh赋予可执行权限这一步

S2I构建的运行脚本负责运行应用程序。它是镜像默认会去执行的脚本：

```
#!/bin/bash -e
exec /opt/openshift/src/app.sh
```

至此，构建器已经准备就绪，用户可以执行make来构建自己的S2I构建器镜像了。它会创建出一个叫sti-simple-shell的Docker镜像，里面会提供应用程序镜像构建所需的环境，包括之前建出来的软件项目。make的输出结果看上去应该会类似下面这样：

```
$ make
imiell@osboxes:/space/git/sti-simple-shell$ make
docker build --no-cache -t sti-simple-shell .
Sending build context to Docker daemon 153.1 kB
Sending build context to Docker daemon
Step 0 : FROM openshift/base-centos7
---> f20de2f94385
Step 1 : RUN chown -R default:default /opt/openshift
---> Running in f25904e8f204
---> 3fb9a927c2f1
Removing intermediate container f25904e8f204
Step 2 : COPY ./sti/bin /usr/local/sti
---> c8a73262914e
Removing intermediate container 93ab040d323e
Step 3 : RUN chmod +x /usr/local/sti/ *
---> Running in d71fab9bbae8
---> 39e81901d87c
Removing intermediate container d71fab9bbae8
Step 4 : USER default
---> Running in 5d305966309f
---> ca3f5e3edc32
Removing intermediate container 5d305966309f
Successfully built ca3f5e3edc32
```

如果执行docker images，现在应该可以看到一个叫sti-simple-shell的镜像存放在宿主机本地。

6. 构建应用程序镜像

回想一下图5-1中列出的流程，如今我们已经得到了一次S2I构建所需的3样东西：

- 源代码；
- 一个构建器镜像，提供构建和运行源代码的环境；
- **sti**程序。

在这次演练里3样东西都放在同一地方，但是在运行时唯一需要在本地的只是**sti**程序。构建器镜像可以从注册中心获取，而源代码可以从Git仓库（如GitHub）中提取。

由于已经准备好了所有3个部分，现在就可以通过**sti**程序跟踪构建流程了：

```
$ sti build --force-pull=false --loglevel=1 \ (1)
file:///root/myproject sti-simple-shell final-image-1 (2)
I0608 13:02:00.727125 00119 sti.go:112] Building final-image-1

I0608 13:02:00.843933 00119 sti.go:182] Using assemble from image:///usr/local/sti (3)
I0608 13:02:00.843961 00119 sti.go:182] Using run from image:///usr/local/sti
I0608 13:02:00.843976 00119 sti.go:182] Using save-artifacts from image:///
➤ usr/local/sti
I0608 13:02:00.843989 00119 sti.go:120] Clean build will be performed
I0608 13:02:00.844003 00119 sti.go:130] Building final-image-1
I0608 13:02:00.844026 00119 sti.go:330] No .sti/environment provided
➤ (no environment file found in application sources)
I0608 13:02:01.178553 00119 sti.go:388]---> Installing application source
I0608 13:02:01.179582 00119 sti.go:388]---> Building application from source (4)
I0608 13:02:01.294598 00119 sti.go:216] No .sti/environment provided
➤ (no environment file found in application sources)
I0608 13:02:01.353449 00119 sti.go:246] Successfully built final-image-1 ) (5)
```

(1) 使用S2I的build子命令来运行构建，禁用镜像默认的强制拉取功能（镜像只存在本地）然后把日志记录提高到一个有用的级别（可以增加数值来获得更多信息）

(2) 将构建指向源代码的Git仓库，然后传入此源代码的S2I构建器镜像引用以及预期产出的应用镜像的标签

(3) 相关构建详情的常规调试信息

(4) 相关构建镜像里应用程序源的通用调试信息

(5) 应用程序镜像构建的细节

在这个例子里，Git仓库是本地存放的（因此前缀是file://），但是也可以通过指向一个在线的仓库的URL（如https://gitserver.example.com/yourusername/yourproject或git@gitserver.example.com:yourusername/yourproject）来引用。

现在可以运行构建好的镜像，源代码已经被应用到了里面：

```
$ docker run final-image-1
Hello World
```

7. 修改及重新构建

现在既然已经有了一个可工作的案例，就不难发现这一构建方法的设计初衷。试想你是一个新来的开发人员，准备给项目贡献代码。你可以在Git仓库上做修改，然后运行一条简单的命令来重新构建镜像而无须知道任何Docker的细节：

```
$ cd /root/myproject
$ cat > app.sh <<< "echo 'Hello S2I!'"
```

```
$ git commit -am 'new message'
$ sti build --force-pull=false file:///root/myproject sti-simple-shell
\
final-image-2
```

运行这个镜像会展示在上述代码里设置的新消息：

```
$ docker run final-image-2
Hello S21!
```

本技巧演示的例子虽然简单，但是不难想象这个框架可以怎样适配用户的特定需求。用户最终得到的是一种可以让开发人员推送变更给他们的软件的其他消费者而无须关心Docker镜像产出的细节的手段。

本技巧可以结合其他技术来改进DevOps流程。举个例子，通过使用Git的post-commit钩子，可以在签入代码的时候自动调用S2I的构建。

5.3 小即是美

如果用户创建了大量的镜像并且到处传播它们，那么镜像大小的问题很可能会被提上议程。尽管Docker使用的镜像分层技术可以帮助改善这一点，但开发人员在自己的一亩三分地上仍然可能会有这样一堆不易于实际管理的镜像。

遇到这些情况时，在组织内部推行一些最佳实践相对而言有助于把镜像尽可能往小缩减。在本节中，我们将展示其中的一些技巧，甚至标准的工具镜像可以从96 MB减到只有6.5 MB——在自己内部网络上传播的对象要小得多。

技巧49 保持构建镜像更小的Dockerfile技巧

由于Dockerfile是构建镜像的推荐方式，当提出缩减镜像大小的想法时社区自然会想到它们。这样的结果是形成了一些利用Dockerfile的功能还有解决一些局限性的建议。

问题

想要缩减用Dockerfile产出的镜像的大小。

解决方案

减少Docker构建时镜像分层的开销。

讨论

我们打算从一个相当经典的用来构建OSQuery的Dockerfile开始，它是一款使用SQL接口来报告系统性能的工具，其Dockerfile的内容如代码清单5-6所示。

代码清单5-6 OSQuery Dockerfile

```
FROM ubuntu:14.04
RUN apt-get update && apt-get upgrade -y (1)
RUN apt-get install -y git (2)
RUN apt-get install -y wget
RUN git clone https://github.com/facebook/osquery.git (3)
WORKDIR /osquery (4)
RUN git checkout 1.0.3
RUN ./tools/provision.sh
RUN make
RUN make package
RUN dpkg -i \
./build/linux/osquery-0.0.1-trusty.amd64.deb (5)
CMD ["/usr/bin/osqueryi"] (6)
```

(1) 初始化Ubuntu容器并升级到最新的包

(2) 安装必需的软件包

- (3) 签出OSQuery Git仓库
- (4) 将OSQuery 1.0.3版本构建成一个deb包
- (5) 安装创建的deb包
- (6) 将容器设置成默认启动OSQuery工具

这个镜像的构建给我们提供了一个大小为2.377 GB的镜像——一个相当大的镜像！

1. 使用一个更小的基础镜像

缩减最终镜像大小的最简单的方法便是从一个更小的基础镜像构建。对于这个例子，我们将把**FROM**这一行改为基于 `stackbrew:ubuntu:14.04` 镜像而非官方的 `ubuntu:14.04`。

请记住使用一个更小的基础镜像便意味着可能会缺失一些先前安装到大一点儿的那个镜像的软件。这些软件包可能包括 `sudo`、`wget` 等。本技巧中不考虑这一点。

完成这一步的话可以把镜像的大小缩减大约10%到2.186 GB。

2. 自己事后清理

可以通过从镜像里删除一些软件包和信息来进一步缩减镜像的大小。如代码清单5-7所示，在**CMD**指令前面添加这几行到代码清单5-6里将会极大地减少容器里的文件数量。

代码清单5-7 OSQuery Dockerfile的碎片清理

```
RUN SUDO_FORCE_REMOVE=yes apt-get purge -y git wget sudo
RUN rm -rf /var/lib/apt/lists/ *
RUN apt-get autoremove -y
RUN apt-get clean
RUN cd
RUN rm -rf /osquery
```

我们的镜像大小现在比最近的一个版本还大一点儿，大约是2.19 GB！由于Docker是分层的，每一条RUN命令都会在最终镜像里创建一个新的写时复制（copy-on-write）分层，即使我们把文件删掉还是会导致镜像大小的增长。这触发了我们下面的改进。

什么是写时复制 写时复制是一项在处理文件时最小化资源占用的技术。只希望读取文件的进程看到的是来自最上层展示的文件——不同容器里的进程可以看到相同的底层文件，因为它们的镜像共享同一个分层。在大多数系统上这样做可以显著降低所需磁盘空间的使用量。当一个容器想要修改一个文件时，该文件会在它被修改前（否则其他容器也能感知到这一变化）复制到该容器所在的分层，这便是写时复制。

3. 将一系列命令设置为一行

尽管可以手动将镜像扁平化（见技巧43），但是也可以在Dockerfile里通过将所有命令放到一条RUN指令中以实现相同的效果，如代码清单5-8所示。

代码清单5-8 带有单条RUN指令的OSQuery Dockerfile

```
FROM stackbrew/ubuntu:14.04
RUN apt-get update && apt-get upgrade -y && \ (1)
    apt-get install -y git wget sudo && \
    git clone https://github.com/facebook/osquery.git && \
    cd /osquery && \
    git checkout 1.0.3 && \
```

```
./tools/provision.sh && \  
make && \  
make package && \  
dpkg -i ./build/linux/osquery-0.0.1-trusty.amd64.deb && \  
SUDO_FORCE_REMOVE=yes apt-get purge -y git wget sudo && \  
rm -rf /var/lib/apt/lists/ * && \  
apt-get autoremove -y && \  
apt-get clean && \  
cd / && \  
rm -rf /osquery  
CMD ["/usr/bin/osqueryi"]
```

(1) 整个安装过程可以缩减到一条RUN指令

成功了！构建好的镜像报告显示的大小现在变成了1.05 GB。我们已经把镜像大小缩减到了原本大小的一半了。

4. 编写一个脚本来完成安装

完成所有这一切之后，用户可能会觉得生成的Dockerfile可读性有点儿差。实现相同效果的一种更加友好的方式是，只需要一点小小的开销，将RUN命令放到一个脚本里然后自行复制进去并且执行即可，如代码清单5-9和代码清单5-10所示。

代码清单5-9 基于一个shell脚本安装的OSQuery Dockerfile

```
FROM stackbrew/ubuntu:14.04  
COPY install.sh /install.sh  
RUN /bin/bash /install.sh && rm /install.sh  
CMD ["/usr/bin/osqueryi"]
```

代码清单5-10 install.sh

```
#!/bin/bash  
set -o errexit (1)  
apt-get update  
apt-get upgrade -y  
apt-get install -y git wget sudo  
git clone https://github.com/facebook/osquery.git
```

```
cd /osquery
git checkout 1.0.3
./tools/provision.sh
make
make package
dpkg -i ./build/linux/osquery-0.0.1-trusty.amd64.deb
SUDO_FORCE_REMOVE=yes apt-get purge -y git wget sudo
rm -rf /var/lib/apt/lists/ *
apt-get autoremove -y
apt-get clean
cd /
rm -rf /osquery
```

(1) 将bash脚本配置为只要其中任何命令返回一个非零退出码就抛出错误

已构建的镜像报告显示的大小没有改变，仍然是1.05 GB。

解决一个又来一个 由于做出了这些更改，用户也因此失去了许多Dockerfile显而易见的有用功能。举个例子，因为工作变成了一个单条指令，所以不会有构建缓存，也就丧失了构建缓存所带来的省时的的好处。同以往一样，这是一个在镜像大小、构建灵活性和构建时间三者之间的权衡。

利用这样几个简单的技巧，用户可以显著地缩减生成的镜像的大小。尽管如此，故事还没有结束，利用下面更加激进的技巧，用户还可以对镜像进行更加严格的瘦身。

技巧50 让镜像变得更小的技巧

我们不妨假设用户拿到一个第三方提供的镜像，而他希望让镜像变得更小。最简单的办法便是启动一个可以工作的镜像，然后删除一些不必要的文件。

经典的配置管理工具往往不会删除东西，除非显式地声明这样做

——取而代之的是它们会从一个不工作的状态开始然后往里面添加新的配置和文件。这导致出于一个特定目的制作出的系统千奇百怪，而且可能会与在一台全新的服务器上运行配置管理工具的结果看上去不太一样，尤其是在配置已经演变了一段时间。通过Docker友好的分层及轻量的镜像技术，可以完成这一流程的反向操作并尝试删除一些东西。

问题

想要让镜像变得更小。

解决方案

删除不必要的软件包和文档文件。

讨论

本技巧将会遵循下列步骤来缩减一个镜像的大小：

- (1) 运行该镜像；
- (2) 进入容器内部；
- (3) 删除不必要的文件；
- (4) 提交容器作为一个新镜像（见技巧14）；
- (5) 扁平化该镜像（见技巧43）。

最后两步在本书前面已经讲过，所以这里仅介绍前面3个步骤。

为了讲解如何利用这一技巧，我们打算使用在技巧40里创建的镜像，并尝试使这个镜像变得更小。

先将该镜像作为一个容器运行起来；

```
docker run -ti --name smaller --entrypoint /bin/bash \  
dockerinpractice/log-cleaner
```

因为这是一个基于Debian的镜像，所以用户可以先看看有哪些可能不需要的软件包然后删掉它们。执行`dpkg -l | awk '{print $2}'`用户便能拿到系统上已经安装的软件包的清单。

之后，用户可以通过执行`apt-get purge -y package_name`来清理这些软件包。如果跳出一条吓人的消息警告用户“你要做的操作可能有害”，不妨点击“返回”以继续。

一旦删掉所有能够安全删除的软件包，紧接着就可以运行如下这些命令来清理apt的缓存：

```
apt-get autoremove  
apt-get clean
```

这是一个相对安全地减少镜像里空间占用的办法。

通过删除文档可以进一步节省大量空间。举个例子，执行`rm -rf /usr/share/doc/ * /usr/share/man/ * /usr/share/info/ *`往往可以删除大概永远不会用到的一些大文件。用户还可以通过手动运行`rm`来删除一些不需要的二进制和类库以进一步缩减镜像的大小。

多数还会选择的另一个地方便是`/var`目录，这里面应该会包含一些临时数据，或者一些对正在运行的程序并非必要的的数据。

下面这条命令将会排除所有后缀为`.log`的文件：

```
find /var | grep '\.log$' | xargs rm -v
```

借助这个有点儿手工的流程，用户可以把最初的 `dockerinpractice/log-cleaner` 镜像轻松减少几十MB，而且如果有动力的话甚至还可以让它变得更小。记住，由于Docker是分层的，用户将需要按照技巧43里介绍的那样把镜像导出并导入。

相关技巧 技巧53将会展示一种更为有效（但是也颇有风险）的方法，它可以显著地缩减镜像的大小。

由于Docker的分层技术，其镜像的大小只会在对它进行操作后才会变大。要让缩减的镜像最终定型，就得用技巧43里重点提到的镜像扁平化技术。

受维护的例子 这个例子中介绍的一系列命令在 <https://github.com/docker-in-practice/log-cleaner-purged> 维护，而且可以从 `dockerinpractice/log-cleaner-purged` 拉取。

技巧51 通过BusyBox和Alpine来精简Docker镜像

自Linux诞生起就出现了一些小而可用的操作系统，它们可以嵌到低功耗或者廉价的计算机上。幸运的是，这些项目的努力成果已经被重新用于生产小型Docker镜像，它们可以用于镜像大小非常重要的场合。

问题

想要得到一个小而功能俱全的镜像。

解决方案

使用一个最小的Linux构建版本，如BusyBox或者Alpine。

讨论

这是另外一个领域，其中现有技术的更迭非常迅速。两种流行的选择是BusyBox和Alpine，而每种都有不同的特点。

如果用户的目标是小而精，那么BusyBox可能会是首选。倘若用户以如下命令启动一个BusyBox镜像，那么可能会发生一些意外情况：

```
$ docker run -ti busybox /bin/bash
exec: "/bin/bash": stat /bin/bash: no such file or directory2015/02/23
➤ 09:55:38 Error response from daemon: Cannot start container
➤ 73f45e34145647cd1996ae29d8028e7b06d514d0d32dec9a68ce9428446faa19: ex
ec:
➤ "/bin/bash": stat /bin/bash: no such file or directory
```

BusyBox甚至已经精简到了没有bash的程度！取而代之的是它使用ash，这是一个兼容posix的shell——实际上它是像bash和ksh这样更高级shell的一个受限版本：

```
$ docker run -ti busybox /bin/ash
/#
```

而许多类似这样的决策的结果便是，BusyBox镜像的大小竟然精简到了小于2.5 MB！

标准实用程序的非GNU版本 BusyBox还有一些其他出人意料的行为。举个例子，该镜像下tar命令的版本将很难从GNU标准的tar包中解压出TAR文件。

如果用户想要编写一个小脚本而只依赖一些简单工具的话，这样做会很赞，但是如果想要在其他任何必须自行安装它的地方运行，这就不太好了。**BusyBox**没有自带的包管理。

其他维护人员已经给**BusyBox**加上了包管理的功能。举个例子，`progrium/busybox`可能不是最小的**BusyBox**容器（它现在小于5 MB），但是它有`opkg`，这意味着用户可以轻松地安装其他常用软件包，同时将镜像的大小保持为绝对最小。举个例子，如果缺少**bash**的话，可以像下面这样安装它：

```
$ docker run -ti progrium/busybox /bin/ash
/ # opkg-install bash > /dev/null
/ # bash
bash-4.3#
```

在提交时，这会生成一个6 MB的镜像。

有一个不太完善但是很有意思的**Docker**镜像（可能会取代`progrium/busybox`）便是`gliderlabs/alpine`。它和**BusyBox**很像，但是有更广泛的软件包，读者可以浏览<http://forum.alpinelinux.org/packages>了解更多细节。

这些软件包均被设计为精简安装。作为一个具体示例，代码清单5-11展示了一个**Dockerfile**，它产生的镜像大小为三分之一吉字节。

代码清单5-11 Ubuntu加mysql-client

```
FROM ubuntu:14.04
RUN apt-get update -q \
&& DEBIAN_FRONTEND=noninteractive apt-get install -qy mysql-client \
&& apt-get clean && rm -rf /var/lib/apt
ENTRYPOINT ["mysql"]
```

使用`DEBIAN_FRONTEND=noninteractive`避免交互 在`apt-get install`之前加上`DEBIAN_FRONTEND = noninteractive`可以确保在安装时不会在安装过程中提示任何输入。由于用户不能在运行命令时轻松地响应问题，因此这一点往往在Dockerfile里非常有用。

对比之下，这次会产出略大于16 MB的镜像：

```
FROM gliderlabs/alpine:3.1
RUN apk-install mysql-client
ENTRYPOINT ["mysql"]
```

技巧52 Go模型的最小容器

尽管可以通过删除冗余文件的方式把工作中的容器精简下来，但这里还有另外一个选择——编译没有依赖的最小二进制。

这样做从根本上简化了配置管理的任务——如果只有一个文件要部署，而且没有依赖包的话，大量的配置管理工具就会变得多余。

问题

想要构建一个没有外部依赖的二进制Docker镜像。

解决方案

构建一个静态链接的二进制。

讨论

为了演示如何使用它，我们首先创建一个小的带有一个C语言小程序的“Hello World”镜像，然后我们进一步展示，针对一个更有价值的应用程序如何完成等价的事情。

1. 一个最小的Hello World二进制

先创建一个新目录，然后创建一个Dockerfile，如代码清单5-12所示。

代码清单5-12 Hello Dockerfile

```
FROM gcc (1)
RUN echo 'int main() { puts("Hello world!"); }' > hi.c (2)
RUN gcc -static hi.c -w -o hi (3)
```

- (1) 这个gcc镜像是一个专为编译而设计的镜像
- (2) 创建一个简单的单行C程序
- (3) 使用-static标志编译这一程序，然后使用-w禁止警告

上述Dockerfile编译了一个简单的没有任何依赖的“Hello world”程序。用户现在可以构建它，并从容器中提取该二进制文件，如代码清单5-13所示。

代码清单5-13 从镜像里提取二进制文件

```
$ docker build -t hello_build . (1)
$ docker run --name hello hello_build /bin/true (2)
$ docker cp hello:/hi new_folder (3)
$ docker rm hello (4)
hello
$ docker rmi hello_build
Deleted: 6afcbf3a650d9d3a67c8d67c05a383e7602baecc9986854ef3e5b9c0069ae9f2
```

- (1) 构建包含静态链接的“hi”二进制程序的镜像
- (2) 使用一条小命令运行镜像以便复制出二进制文件

(3) 使用docker cp命令将“hi”二进制程序复制到new_folder里

(4) 清理：不再需要这些东西

至此，用户在一个全新的目录里得到了一个静态地构建好的二进制程序。通过下面这条命令切换到该目录：

```
$ cd new_folder
```

现在创建其他的Dockerfile，如代码清单5-14所示。

代码清单5-14 最小Hello Dockerfile

```
FROM scratch (1)
ADD hi /hi (2)
CMD ["/hi"] (3)
```

(1) 使用零字节的空白镜像

(2) 把“hi”二进制程序添加到镜像

(3) 将镜像设置为默认执行“hi”二进制程序

如代码清单5-15中展示的那样构建并运行它。

代码清单5-15 创建最小容器

```
$ docker build -t hello_world .
Sending build context to Docker daemon 931.3 kB
Sending build context to Docker daemon
Step 0 : FROM scratch
--->
Step 1 : ADD hi /hi
---> 2fe834f724f8
Removing intermediate container 01f73ea277fb
Step 2 : ENTRYPOINT /hi
---> Running in 045e32673c7f
---> 5f8802ae5443
Removing intermediate container 045e32673c7f
```

```
Successfully built 5f8802ae5443
$ docker run hello_world
Hello world!
$ docker images | grep hello_world
hello_world      latest      5f8802ae5443   24 seconds ago  928.3 kB
```

该镜像构建出来，运行，而大小不超过1 MB！

2. 最小的Go Web服务器镜像

这是一个相对简单的例子，但是相同的原理可以推广到用Go语言来构建的程序。Go语言的一个很吸引人的特性是，构建这种静态二进制文件相对比较简单。

为了演示这种能力，我们创建了一个用Go语言实现的简单Web服务器，它的全部代码都放在<https://github.com/docker-in-practice/go-web-server>。

构建这一简单Web服务器的Dockerfile，如代码清单5-16所示。

代码清单5-16 静态编译一个Go Web服务器的Dockerfile

```
FROM golang:1.4.2 (1)
RUN CGO_ENABLED=0 go get \ (2)
-a -ldflags '-s' -installsuffix cgo \ (3)
github.com/docker-in-practice/go-web-server (4)
CMD ["cat", "/go/bin/go-web-server"] (5)
```

(1) 这一次构建已经验证过是可以工作在这一版本的golang镜像；如果构建失败，则可能是这一版本已经不再可用

(2) go get命令会从提供的URL获取源代码，然后在本地编译它。将CGO_ENABLED环境变量设置为0是为了防止交叉编译

(3) 为Go编译器设置的许多杂七杂八的标志是为了保证静态编

译并缩减编译后的文件大小

(4) Go Web服务器的源代码仓库

(5) 设置生成镜像的默认命令为输出该可执行文件

如果将此Dockerfile保存到一个空目录然后用它做构建的话，将可以得到一个包含该程序的镜像。因为已经将该镜像的默认命令指定为输出可执行文件的内容，所以现在只需要运行该镜像，然后把输出发送到宿主机上的一个文件，如代码清单5-17所示。

代码清单5-17 从镜像里获取Go Web服务器

```
$ docker build -t go-web-server . (1)
$ mkdir -p go-web-server && cd go-web-server (2)
$ docker run go-web-server > go-web-server (3)
$ chmod +x go-web-server (4)
```

(1) 构建并给镜像打标签

(2) 创建并切换到一个全新的目录来存放二进制文件

(3) 运行该镜像然后将二进制文件的输出重定向到一个文件

(4) 给二进制文件赋予可执行权限

现在，和“hi”程序一样，用户得到一个没有类库依赖也不需要访问文件系统的二进制文件。如此一来，正如前面所讲，我们就可以从零字节的空白镜像开始创建一个Dockerfile，然后把二进制文件添加到里面：

```
FROM scratch
ADD go-web-server /go-web-server (1)
ENTRYPOINT ["/go-web-server"] (2)
```

(1) 将静态二进制文件添加到镜像

(2) 将镜像设置为默认运行此程序

现在可以构建它并且运行这一镜像。生成的镜像的大小略大于4 MB:

```
$ docker build -t go-web-server .
$ docker images | grep go-web-server
go-web-server   latest      de1187ee87f3  3 seconds ago  4.156 MB
$ docker run -p 8080:8080 go-web-server -port 8080
```

可以打开<http://localhost:8080>访问它。如果端口事先已经被占用，不妨自己选一个端口替换上述代码里的两个8080。

3. Docker是多余的吗

如果可以将应用程序捆绑到一个二进制文件的话，为何还要用Docker呢？用户可以把二进制文件移走，运行多个副本，等等。

用户愿意的话，当然可以这么做，但是这样会失去下面这些特性：

- Docker生态系统里所有的容器管理工具；
- Docker镜像里的元数据，它记录了重要的应用信息，如端口、卷、标签等；
- Docker的隔离性所带来的可运维能力。

技巧53 使用inotifywait给容器瘦身

现在我们将使用一个小巧的工具来进一步给容器瘦身，它会告诉我们当运行一个容器时有哪些文件会被引用。

这可以称作是一项“核武器”，因为在生产中实施的话可能是相当危险的。但是，它算是一个有助于了解系统的指导手段，即使不遵循下面介绍的去实际使用它也没关系——要知道配置管理的一个关键部分便是理解应用程序正常运转所需的条件。

问题

想要将容器里的文件和权限集尽可能缩减到最小。

解决方案

使用inotify-tools来确定应用需要哪些文件，然后删除所有其他文件。

讨论

从整体上来说，用户需要知道当其在在一个容器里执行一条命令时它会访问哪些文件。如果用户将容器文件系统上所有其他文件都删掉的话，理论上来说依旧可以拥有一切运行时所需的东西。

在这次演示中，将会用到技巧50里介绍过的log-cleaner-purged镜像。用户需要安装好inotify-tools，然后再执行inotifywait得到一个访问了哪些文件的报告。随后运行模拟该镜像的入口点的程序

（log_clean脚本）。紧接着，用户可以依据生成的文件报告，删除任何没有访问到的文件：

```
[host]$ docker run -ti --entrypoint /bin/bash \ (1)
--name reduce dockerinpractice/log-cleaner-purged (2)
$ apt-get update && apt-get install -y inotify-tools (3)
$ inotifywait -r -d -o /tmp/inotifywaitout.txt \ (4)
/bin /etc /lib /sbin /var (5)
inotifywait[115]: Setting up watches. Beware: since -r was given, this
  ► may take a while!
inotifywait[115]: Watches established.
$ inotifywait -r -d -o /tmp/inotifywaitout.txt /usr/bin /usr/games \ (
6)
```

```

/usr/include /usr/lib /usr/local /usr/sbin /usr/share /usr/src
inotifywait[118]: Setting up watches. Beware: since -r was given, this
➡ may take a while!
inotifywait[118]: Watches established.
$ sleep 5 (7)
$ cp /usr/bin/clean_log /tmp/clean_log (8)
$ rm /tmp/clean_log
$ bash
$ echo "Cleaning logs over 0 days old"

$ find /log_dir -ctime "0" -name '*log' -exec rm {} \; (9)
$ awk '{print $1$3}' /tmp/inotifywaitout.txt | sort -u > \
/tmp/inotify.txt (10)
$ comm -2 -3 \ (11)
<(find /bin /etc /lib /sbin /var /usr -type f | sort) \
<(cat /tmp/inotify.txt) > /tmp/candidates.txt
$ cat /tmp/candidates.txt | xargs rm (12)
$ exit (13)
$ exit

```

(1) 安装inotify-tools包

(2) 覆盖此镜像默认的入口程序

(3) 给容器起一个名字，后面可以用它来引用该容器

(4) 以递归 (-r) 和守护进程 (-d) 模式运行inotifywait，获取一个已访问文件的清单并写到outfile（以-o标志指定的）里

(5) 指定感兴趣的需要关注的文件夹。注意不要监听/tmp，因为/tmp/inotifywaitout.txt文件如果自己监听自己的话可能会造成一个死循环

(6) 对/usr文件夹上的子文件夹再次执行inotifywait。由于/usr文件夹里有太多文件需要inotifywait来处理，因此用户需要单独一个地去指定

(7) Sleep 会给予inotifywait 一个合理的等待启动的时间

(8) 记得访问一个要用到的脚本文件。还有，要确保有执行rm命令的权限

(9) 像脚本里做的那样，启动一个bash shell，然后运行脚本里本来要执行的一些命令

(10) 利用awk工具从inotifywait的日志输出里生成一个文件名清单，然后将它去重并排序

(11) 使用comm工具输出一个文件系统中未访问的文件清单

(12) 删除所有未访问的文件

(13) 退出之前打开的bash shell，随后再退出容器本身

现在已经完成：

- 给一些文件设置监听以查看哪些文件是被访问的；
- 执行所有的命令来模拟脚本的运行；
- 执行一些命令确保用户有权限访问后面肯定要用到的脚本和rm实用工具；
- 获取一个运行期间未被访问的所有文件的清单；
- 删除所有未被访问的文件。

现在，可以将此容器扁平化（见技巧43），创建出一个新镜像，然后测试它是否仍然能够正常工作：

```
$ ID=$(docker export reduce | docker import -) (1)
$ docker tag $ID smaller (2)
$ docker images | grep smaller
smaller latest 2af3bde3836a 18 minutes ago 6.378 MB (3)
$ mkdir -p /tmp/tmp (4)
$ touch /tmp/tmp/a.log (5)
$ docker run -v /tmp/tmp:/log_dir smaller \
```

```
/usr/bin/clean_log 0
Cleaning logs over 0 days old
$ ls /tmp/tmp/a.log
ls: cannot access /tmp/tmp/a.log: No such file or directory
```

(1) 将镜像扁平化然后把镜像ID放到环境变量ID里

(2) 给新的扁平镜像打上smaller的标签

(3) 现在该镜像甚至比之前大小的10%还小

(4) 在测试目录上运行新创建的镜像，并检查创建的文件是否已经被删除

(5) 为了测试创建一个新的文件夹和文件，模拟一个日志目录

我们将此镜像的大小从96 MB缩减到了大约6.5 MB，而它似乎仍然可以正常工作。相当节俭！

有风险！ 本技巧就像CPU超频一样，并不是一个无关紧要的优化。这个特定案例能够很正常地运转，是因为它是一个运行范围相当有限的应用程序，但是用户的一些核心关键业务应用程序可能是更复杂的，而且在如何访问文件方面可能是更加动态的。用户可以轻易删除一个在运行时未访问的文件，但是该文件可能会在某些其他场景下需要用到。

如果有点儿担心删掉的这些文件后面可能会用到而导致镜像损坏的话，可以用/tmp/candidates.txt文件收录未触及的最大文件的清单，如下所示：

```
cat /tmp/candidates.txt | xargs wc -c | sort -n | tail
```

然后可以删掉那些确定应用程序将来不会用到的大一点儿的文

件。这也是一场大的胜利！

技巧54 大也可以美

尽管本节是关于如何保持镜像小的，但值得铭记的是小也不一定就是更好的。正如我们接下来将讨论的那样，一个相对较大的单体镜像可以比一个小镜像更加高效。

问题

想要降低由于Docker镜像导致的磁盘空间占用和网络带宽。

解决方案

为组织内部创建一个统一的、较大的、单体的基础镜像。

讨论

这是一个两难的取舍，但是使用一个大的单体镜像可以帮用户节省磁盘空间和网络带宽。

回想一下，Docker在容器正在运行时使用的是写时复制机制。这意味着用户可以运行数百个Ubuntu容器，而在每个容器启动后只需要占用少量额外的磁盘空间。

如图5-2所示，如果用户在Docker服务器上运行了大量不同的较小的镜像的话，那么使用的磁盘空间甚至可能会比运行一个大一些的一切所需都包揽到其中的单体镜像还要多。

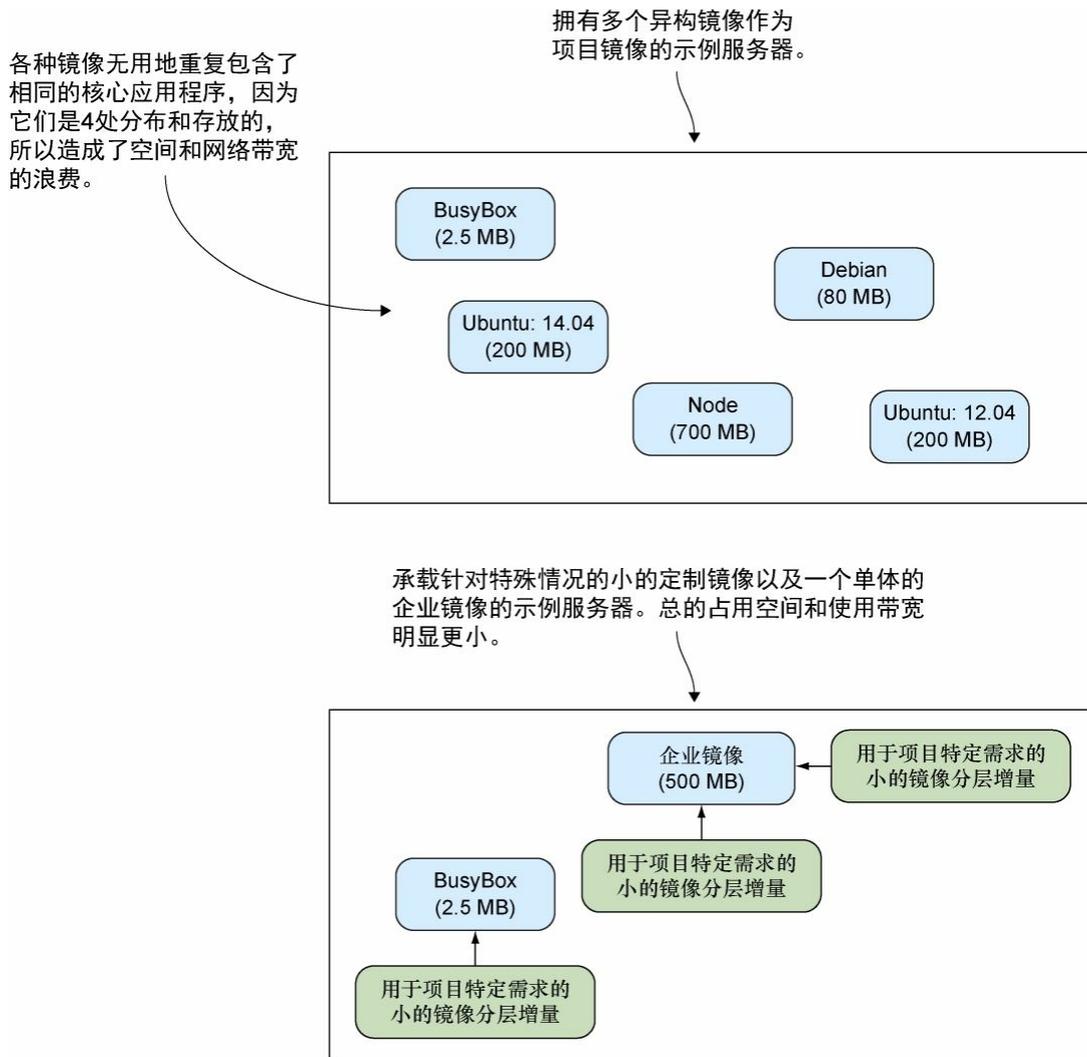


图5-2 许多小的基础镜像与较少的大的基础镜像的对比

读者可能会想起共享库的原理。一个共享库可以一次性被多个应用程序加载，这减少了要运行这些程序所需的磁盘空间和内存的使用量。同理，一个组织内共享的基础镜像可以节省空间，因为它只需要被下载一次，而且应该囊括了所需的一切。之前多个镜像里需要用到的程序和库现在只需要引用一次。

此外，这样做的另外一个好处是可以有一个跨团队共享的单一的、集中管理的镜像。该镜像的维护可以是集中式的，一些改进也是共享的，并且构建中遇到的问题只需要解决一次。

要采用本技巧，需要注意下列事项。

- 基础镜像首先应该是可靠的。如果它的行为不一致，应当避免使用。
- 对基础镜像的变更应该在某处可以可视化地跟踪到，以便用户可以自行调试。
- 在更新香草（vanilla）镜像时一些回归测试是至关重要的，这样可以减少麻烦。
- 在添加内容到基础镜像时要谨慎——一旦添加到了基础镜像，它便很难删除，而且镜像会膨胀得很快。

我们在自己600强的开发公司里使用了本技巧，产生的效果非常好。每月构建的核心应用会被打包到一个大的镜像并发布到内部的Docker注册中心。默认情况下，团队将会在所谓的“香草”企业镜像上构建应用，有必要的话再在上面创建定制镜像分层。

5.4 小结

本章展示了如何管理Docker镜像的配置，这是任何Docker工作流都必须关注的一个重点。我们一起了解了用户可以如何在Docker官方推荐的路线中结合现有流程，并且介绍了一些构建镜像时的最佳实践。

我们就下面几个问题讨论了“如何做”：

- 如果觉得受限的话，可以扩展Dockerfile的灵活性；
- 可以通过扁平化镜像的方式删除底层镜像分层里的私密文件；
- Dockerfile和像Chef这样更加传统的配置管理工具并不是互斥的；

- 用户应该努力让自己的镜像尽可能小。

至此，读者已经通过一些手段得到了自己的Docker镜像，是时候考虑该如何和其他人分享这些镜像，为促进持续交付和持续集成打下基础。

[1] Ddocker从1.9版本开始已经引入ARG指令支持传入变量，见<https://docs.docker.com/engine/reference/builder/#arg>。——译者注

第三部分 Docker与DevOps

现在可以将Docker带出开发环境，并开始 in 软件交付的其他阶段使用了。构建和测试自动化是DevOps运动的基石。本章将通过软件交付生命周期、部署及现实环境测试的自动化来演示Docker的威力。第6章将展示几个用于交付和完善持续集成的技巧，让软件交付变得更加可靠和可扩展。持续交付是第7章的重点。这一章解释了什么是持续交付，并说明用Docker来完善开发流程中这个方面的方法。第8章讲述如何充分发挥Docker网络模型的效率、创建多容器服务、模拟现实网络以及按需创建网络。这一部分将引领读者从开发走向在生产环境中运行Docker所能想到的方方面面。

第6章 持续集成：加快开发流水线

本章主要内容

- 将Docker Hub工作流作为CI工具使用
- 提升IO密集型构建的速度
- 使用Selenium进行自动化测试
- 在Docker里运行Jenkins
- 把Docker作为Jenkins从节点使用
- 在开发团队内扩展可用的运算能力

本章中将说明几个使用Docker来启用并提升持续集成（continuous integration, CI）效率的技巧。

到目前为止，读者应该清楚Docker是非常适合用于自动化的。它的轻量级特性以及它具有的在不同场所进行环境移植的能力，让它成为了持续集成的关键推动者。实践表明，本章中的技巧在实现业务持续集成流程中的价值不可估量。

保证构建环境的稳定性和可重现性、使用测试工具要求大量设置、对构建容量进行扩展，这些都是读者可能遇到的问题，而Docker可以提供相应的支持。

持续集成 持续集成是指用于加快开发流水线的的一个软件生命周期策略。在每次代码库发生重大修改时，通过自动重新运行测试，可以获得更快且更稳定的交付，因为被交付的软件具有一个基础层次的稳定性。

6.1 Docker Hub自动化构建

Docker Hub自动化构建功能已经在技巧9中提到过，只是未深入其细节。简而言之，如果（将项目）指向一个包含Dockerfile的Git仓库，Docker Hub将会负责处理镜像构建及提供下载的过程。Git仓库中发生任何变更都将触发一次镜像重新构建，这对于持续集成流程来说相当有用。

技巧55 使用Docker Hub workflow

本技巧将介绍Docker Hub workflow，通过它可触发镜像的重新构建。

需要docker.com账号 在本节中，需要一个docker.com账号，并链接到GitHub或Bitbucket账号。如果读者还未设置及建立链接，可在github.com和bitbucket.org的首页找到说明。

问题

想要在代码发生变更时自动测试并将变更推送到镜像中。

解决方案

建立一个Docker Hub仓库并将其链接到代码上。

讨论

尽管Docker Hub构建并不复杂，还是有一些必要的步骤，因此将其分解成表6-1中的小块，以此作为该流程的概述。

表6-1 建立一个链接的**Docker Hub**仓库

序 号	步 骤
1	在GitHub或Bitbucket上创建仓库
2	克隆新的Git仓库
3	将代码添加到Git仓库中
4	提交源文件
5	推送Git仓库
6	在Docker Hub上创建一个新仓库
7	将Docker Hub仓库链接到Git仓库上
8	等待Docker Hub构建完成
9	提交并推送一项变更到源文件中
10	等待第二次Docker Hub构建完成

Git仓库与Docker仓库 Git和Docker都使用仓库这个术语来指向一个项目。这可能会对用户造成困扰。即便此处将Git仓库和Docker仓库链接在一起，这两个类型的仓库也并不是一回事。

1. 在GitHub或Bitbucket上创建仓库

在GitHub或Bitbucket上创建一个新仓库。可以给它起任何一个想要的名字。

2. 克隆新的Git仓库

将这个新的Git仓库克隆到宿主机上。可以在Git项目首页找到执行这一步的命令。

将目录切换到这个仓库里。

3. 将代码添加到Git仓库中

现在需要将代码添加到该项目中。

此处可以添加任何所需的Dockerfile，不过代码清单6-1展示的是一个可以工作的示例。它包含两个文件，展示的是一个简单的开发工具环境。它会安装一些首选工具，并打印出当前的bash版本。

代码清单6-1 Dockerfile——简单的开发工具容器

```
FROM ubuntu:14.04
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update
RUN apt-get install -y curl (1)
RUN apt-get install -y nmap
RUN apt-get install -y socat
RUN apt-get install -y openssh-client
RUN apt-get install -y openssl
RUN apt-get install -y iotop
RUN apt-get install -y strace
RUN apt-get install -y tcpdump
RUN apt-get install -y lsof
RUN apt-get install -y inotify-tools
```

```
RUN apt-get install -y sysstat
RUN apt-get install -y build-essential
RUN echo "source /root/bash_extra" >> /root/.bashrc (2)
ADD bash_extra /root/bash_extra (3)
CMD ["/bin/bash"]
```

- (1) 安装有用的软件包
- (2) 在root的bashrc中添加一行用以加载bash_extra
- (3) 将源文件中的bash_extra添加到容器中

现在需要创建上面引用的bash_extra文件，其内容如代码清单6-2所示。

代码清单6-2 bash_extra——额外的bash命令

```
bash --version
```

4. 提交源文件

要提交这些源文件，可使用以下命令：

```
git commit -am "Initial commit"
```

5. 推送Git仓库

现在可以使用以下命令将源文件推送到Git服务器上：

```
git push origin master
```

6. 在Docker Hub上创建一个新仓库

接下来需要在Docker Hub上为这个项目创建一个仓库。打开 <https://hub.docker.com>并确保已经登录，然后点击“Add Repository”（添加仓库）并选择“Automated Build”（自动化构建）^[1]。

7. 将Docker Hub仓库链接到Git仓库上

此时将看到一个选择Git服务的界面。选取所使用的源代码服务（GitHub或Bitbucket），然后从所提供的清单中选择新仓库。（如果这一步不成功，可能需要先建立Docker Hub账号与Git服务之间的链接。）

接着将看到一个构建配置选项页面。可以保留默认值并点击下面的“Create Repository”（创建仓库）。

8. 等待Docker Hub构建完成

这时将看到一个说明链接工作正常的页面。点击“Build Details”（构建详情）链接。

接下来，将看到一个展示构建细节的页面。在“Builds History”（构建历史）下面会有第一次构建的条目。如果什么也没看到，可能需要点击按钮^[2]来手工触发构建。构建ID后面的“Status”（状态）字段将显示“Pending”（挂起）^[3]、“Finished”（完成）^[4]、“Building”（正在构建）或“Error”（错误）。如果一切顺利，将看到前3个状态之一。如果看到了“Error”，就说明存在问题，需要点击构建ID查看其错误信息。

构建可能很花时间 构建启动可能需要花费一段时间，因此有时在等待时看到“Pending”是非常正常的。

可以时不时点击“Refresh”（刷新），直到看到构建完成。一旦构建完成，就可以通过页面顶部列出的`docker pull`命令拉取这个镜像。

9. 提交并推送一项变更到源文件中

假设现在想要在登录时获取更多的环境信息，如输出正在运行的发行版详情。要实现这一点，可在`bash_extra`文件中添加这几行，而此时它看起来是这样的：

```
bash --version
cat /etc/issue
```

然后按第4步和第5步所示进行提交和推送。

10. 等待（第二次）DockerHub构建完成

如果返回构建页面，新的一行将出现在“Builds History”（构建历史）一节的下面，可以按步骤8所述对此次构建进行跟踪。

只在出错时收到电子邮件 如果构建出现错误将会收到相关电子邮件（如果一切正常则不会有电子邮件），因此一旦适应了这个 workflow，只需要在收到电子邮件时进行检查。

现在，可以使用Docker Hub工作流了！读者将很快适应这个框

架，并发现它在保持构建更新和减少手工重新构建Dockerfile的认知负荷这两方面非常有价值。

6.2 更有效的构建

CI意味着软件和测试更频繁的重新构建。尽管Docker使交付CI更加容易，但接下来可能会遇到运算资源负载上升的问题。

这里将介绍几种用于缓解磁盘IO、网络带宽及自动化测试方面的压力的方法。

技巧56 使用eatmydata为I/O密集型构建提速

因为Docker非常适合用于自动化构建，随着时间推移，可能会用它来执行大量的I/O密集型构建。Jenkins作业、数据库重建脚本以及大量的代码签出都将对磁盘造成冲击。在这种情况下，任何能获得的速度提升对用户来说都大有裨益，这不仅能节省时间，还能极大减少资源竞争造成的大量额外开销。

本技巧已经被证实可以提升高达1/3的速度，而实际经验也支持这一点。其作用不容小觑。

问题

想要加快I/O密集型构建的速度。

解决方案

在镜像上安装eatmydata。

讨论

eatmydata是一个使用系统调用来写入数据，并通过绕开持久化变更所需工作从而大大提升速度的程序。这会造成部分安全性的缺失，因此不建议作为常规使用，不过对于那些不需要持久化的环境，如测试环境，这就非常有用。

1. 安装

要安装eatmydata，有很多种选择。

如果运行的是基于deb的发行版，可以运行`apt-get install`来安装。

如果运行的是基于rpm的发行版，可以通过网站上搜索并下载它，然后运行`rpm --install`来安装。类似rpmfind.net这样的网站是一个不错的入口。

在不得已的情况下，如果安装了一个编译器，可以按代码清单6-3所示直接下载并编译它。

代码清单6-3 编译并安装eatmydata

```
$ url=https://www.flamingspork.com/projects/libeatmydata/  
➤ libeatmydata-105.tar.gz (1)  
$ wget -qO- $url | tar -zxf - (2)  
$ ./configure --prefix=/usr (3)  
$ make (4)  
$ sudo make install (5)
```

(1) flamingspork.com是其维护人员的网站

(2) 如果无法下载这个版本，请到其网站上检查它是否已经更新到105之后的版本

(3) 如果想把eatmydata可执行文件安装在/usr/bin之外的其他地方，可修改此前缀目录

(4) 构建eatmydata可执行文件

(5) 安装该软件，这一步要求使用root权限

2. 使用eatmydata

一旦libeatmydata被安装到镜像上（不论是使用软件包或是源文件），就可以在任何命令之前运行eatmydata包装脚本来使用它：

```
docker run -d mybuildautomation eatmydata /run_tests.sh
```

图6-1从高层次展示了eatmydata是如何节省处理时间的。

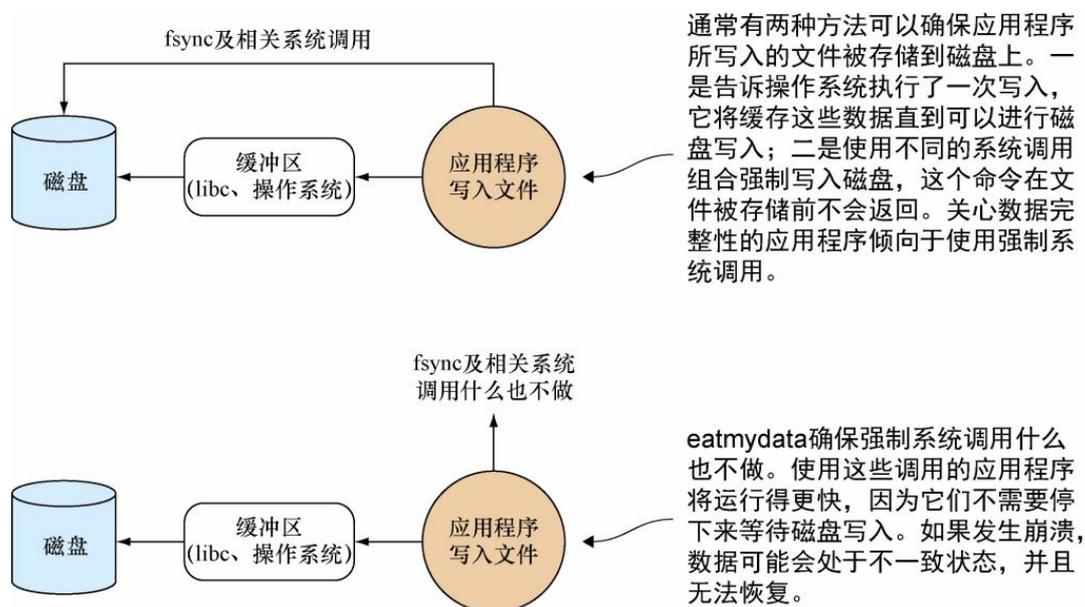


图6-1 应用程序写入磁盘时不使用和使用eatmydata的对比

谨慎使用！ eatmydata跳过了用于保证数据被安全地写入磁盘的步骤，因此存在一定的风险，即在程序认为数据已经写入磁盘时可能

这一步还没完成。对于测试环节而言，这通常无关紧要，因为其数据可任意处理，但不要在任何数据很重要的环境中使用eatmydata来提升速度！

技巧57 设置一个软件包缓存用于加快构建速度

由于Docker非常适合开发环境、测试环境和生产环境服务的频繁重新构建，读者很快会发现这会对网络反复造成冲击。其中的主要原因之一是从互联网下载软件包文件。即使是在一台机器上，这也可能是一个缓慢（且昂贵）的开销。本技巧展示了如何为软件包下载设置一个本地缓存，同时涵盖apt与yum。

问题

想要通过减少网络I/O来加快构建速度。

解决方案

为包管理器安装一个Squid代理。

讨论

图6-2演示了本技巧在理论上是如何工作的。

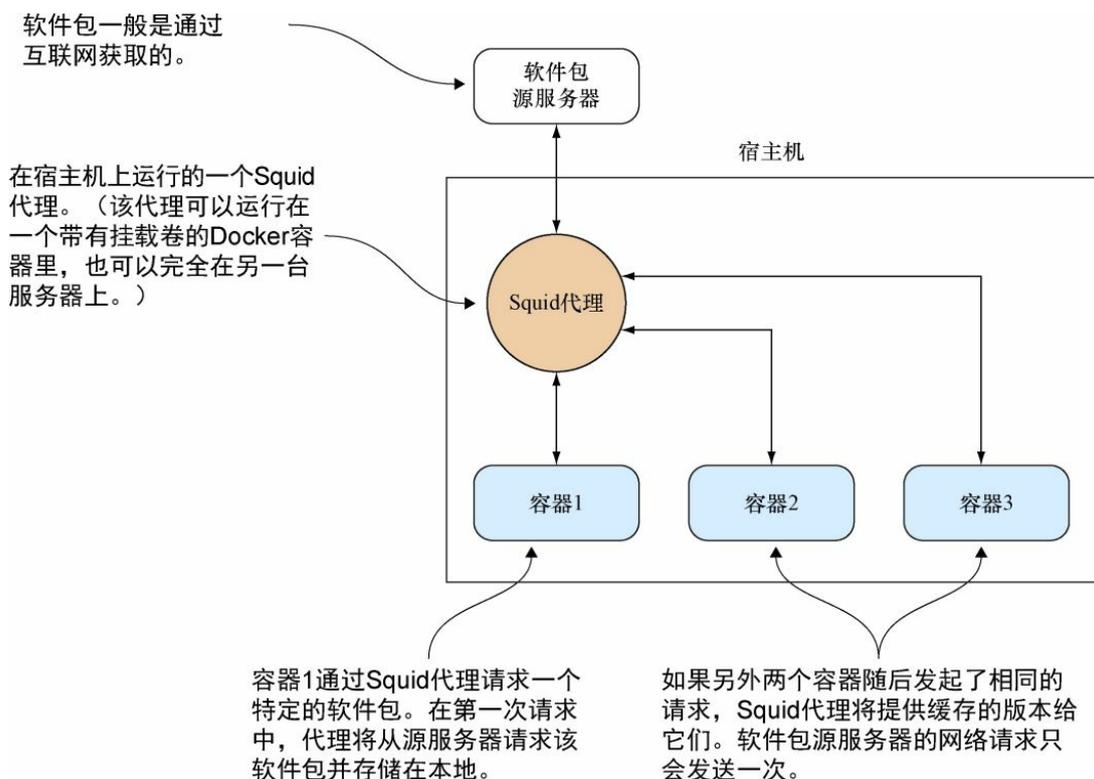


图6-2 使用一个Squid代理来缓存软件包

因为软件包的调用会首先到达本地Squid代理，而只有第一次会通过互联网进行请求，对每一个软件包而言，只会有一次互联网请求。如果有数百个容器全部都在从互联网拉取相同的大型软件包，这将节省大量的时间和金钱。

网络设置！ 在宿主机上进行安装时可能会碰到网络配置问题。后续几节会给出用以判定这种情况的建议，不过如果读者不确定如何处理，可能需要寻求来自友好的网络管理员的帮助。

1. Debian

对于Debian（或通常所说的apt或.deb）软件包，安装要简单得多，因为存在一个打包好的版本。

在基于Debian的宿主机上运行下面这条命令：

```
sudo apt-get install squid-deb-proxy
```

通过telnet连接到8000端口来确认该服务已经启动：

```
$ telnet localhost 8000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

如果看到了上述输出，可按下Ctrl+]再按Ctrl+d退出。如果没看到这个输出，则要么Squid未被正确安装，要么它被安装在了一个非标准端口上。

为了设置容器使用这个代理，这里提供了如下示例Dockerfile。需要注意的是，从容器的角度看，宿主机的IP地址每次运行都可能发生改变。在安装新软件前，可能需要将这个Dockerfile转换成一个在容器里运行的脚本。

```
FROM debian
RUN apt-get update -y && apt-get install net-tools (1)
RUN echo "Acquire::http::Proxy \"http://$( \ (2)
route -n | awk '/^0.0.0.0/ {print $2}' \ (3)
):8000\";" \
> /etc/apt/apt.conf.d/30proxy (4)
RUN echo "Acquire::http::Proxy::ppa.launchpad.net DIRECT;" >> \
/etc/apt/apt.conf.d/30proxy
CMD ["/bin/bash"]
```

(1) 确保route工具已安装

(2) 8000端口用于连接宿主机上的Squid代理

(3) 为了确定从容器角度看到的宿主机IP地址，运行route命令，并使用awk从输出中提取相关IP地址（见技巧59）

(4) 带有正确的 IP 地址和配置的输出行被添加到apt的代理配置文件中

2. yum

在宿主机上，使用软件包管理器安装“squid”软件包，确保Squid安装就位。

然后需要修改Squid配置文件来创建一个更大的缓存空间。打开/etc/squid/squid.conf文件并用cache_dir ufs /var/spool/squid 10000 16 256替换以#cache_dir ufs /var/spool/squid开头的那行注释。这将创建一个10 000 MB的空间，应该够用了。

通过telnet连接到3128端口来确保服务已经启动：

```
$ telnet localhost 3128
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

如果看到了上述输出，可按下Ctrl+]再按Ctrl+d退出。如果没看到这个输出，则要么Squid未被正确安装，要么它被安装在了一个非标准端口上。

为了设置容器使用这个代理，这里提供了如下示例Dockerfile。需要注意的是，从容器的角度看，宿主机的IP地址每次运行都可能发生改变。在安装新软件前，可能需要将这个Dockerfile转换成一个在容器里运行的脚本。

```
FROM centos:centos7
RUN yum update -y && yum install -y net-tools (1)
RUN echo "proxy=http://$(route -n | \ (2)
awk '/^0.0.0.0/ {print $2}'):3128" >> /etc/yum.conf (3)
```

```
RUN sed -i 's/^mirrorlist/#mirrorlist/' \
/etc/yum.repos.d/CentOS-Base.repo
RUN sed -i 's/^#baseurl/baseurl/' \ (4)
/etc/yum.repos.d/CentOS-Base.repo
RUN rm -f /etc/yum/pluginconf.d/fastestmirror.conf (5)
RUN yum update -y (6)
CMD ["/bin/bash"]
```

(1) 确保route工具已安装

(2) 为了确定从容器角度看到的宿主机IP地址，运行route命令，并使用awk从输出中提取相关IP地址（见技巧59）

(3) 3128端口用于连接宿主机上的Squid代理

(4) 为了避免可能出现的缓存未命中，移除镜像清单，只使用基础的URL。这将确保只会命中一组URL用于获取软件包，从而更有可能命中缓存的文件

(5) 移除fastest-mirror插件，因为已经不再需要它了

(6) 确保检查所有镜像。在运行yum update时，配置文件里列出的镜像可能包含了过期信息，因此第一次更新可能会很慢

如果按这种方式设置两个容器，并依次在上面安装相同的大型软件包，就会发现第二次安装在下载它的必要软件时比第一次快得多。

Docker化Squid代理 读者可能已经注意到可以在一个容器里而不是在宿主机上运行Squid代理。为了保持说明简单，这里没有展示这一做法（在某些情况下，要让Squid工作在一个容器里需要更多步骤）。读者可以在<https://github.com/jpetazzo/squid-in-a-can>阅读更多这方面的内容，以及如何让容器自动使用这个代理。

技巧58 在Docker内部运行Selenium测试

本书尚未深入讲解的Docker用例之一是运行图形化应用程序。在第3章中，在开发环境的“保存游戏”中VNC被用来连接容器（技巧18），但这可能过于笨重了——窗口被包含在VNC Viewer窗口里面，并且桌面互动性可能比较有限。这里介绍一种替代方法，将演示如何使用Selenium来编写图形化测试，同时展示如何作为CI工作流程的一部分使用这个镜像来运行测试。

问题

想要在CI过程中运行图形化程序，同时能将这些图形化程序显示到自己的屏幕上。

解决方案

共享X11服务器套接字以便在自己的屏幕上查看程序，同时在CI过程中使用xvfb。

讨论

不管启动容器需要做什么其他事情，都必须把X11用来显示窗口的Unix套接字作为一个卷挂载到容器里，同时需要指定窗口要显示到哪个显示器上。可以通过运行以下命令确认这两样东西是否被设置为其默认值：

```
~ $ ls /tmp/.X11-unix/  
X0  
~ $ echo $DISPLAY  
:0
```

第一个命令检查的是X11服务器Unix套接字正运行在本技巧后续内容所假定的位置上。第二个命令检查的是应用程序用于查找X11套

接字的环境变量。如果运行这些命令的输出与这里的输出不一致，可能需要修改本技巧中的某些命令参数。

检查好机器设置，现在要把运行在一个容器内的应用程序无缝地显示在容器外。需要解决的主要问题是：计算机为了防止其他人连接该机器、接管显示器以及悄悄记录按键动作所实施的安全性。在技巧26中我们已经大致看到如何完成这一步，不过当时并未说明它的工作原理以及其替代方案。

X11具有多种对使用X套接字的容器进行认证的方式。先来看一下.Xauthority文件——它应该存在于用户的主目录中。它包含了主机名以及每台主机连接时必须使用的“私密cookie”。通过赋予Docker容器与机器相同的主机名（hostname），可以使用现有的X认证文件：

```
$ ls $HOME/.Xauthority
/home/myuser/.Xauthority
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
  --hostname=$HOSTNAME -v $HOME/.Xauthority:/root/.Xauthority \
  -it ubuntu:14.04 bash
```

第二种允许Docker访问该套接字的方法是一个较为低级的工具，但它具有安全问题，因为它会禁用X带来的所有防护措施。如果无人能访问该电脑，那么这是一个可以接受的解决方案，不过应该优先尝试使用X认证文件。在尝试以下步骤之后，可以通过运行xhost -来恢复安全性（不过这会把Docker容器阻挡在外）：

```
$ xhost +
access control disabled, clients can connect from any host
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
  -it ubuntu:14.04 bash
```

第一行禁用了对X的所有访问控制，而第二行将运行容器。值得注意的是无须设置主机名或挂载X套接字的任何部分。

一旦容器启动，接下来就要检查一下它是否工作正常。可以通过运行如下命令来进行这一步：

```
root@ef351febcee4:/# apt-get update && apt-get install -y x11-apps
[...]
root@ef351febcee4:/# xeyes
```

这将启动检测X是否正常工作的一个经典的应用程序——xeyes。我们可以看到跟随鼠标在屏幕上移动的一双眼睛。需要注意的是，（与VNC不同）该应用程序是整合到桌面里的——如果多次启动xeyes，将看到多个窗口。

现在可以开始使用 Selenium 了。假如读者之前从未使用过它，它是一个能够实现浏览器动作自动化的工具，常常用于测试网站代码——它需要一个用于运行浏览器的图形显示器。尽管它最经常与Java一起使用，但为了获取更多互动性，这里将使用Python：

```
root@ef351febcee4:/# apt-get install -y python2.7 python-pip firefox
[...]
root@ef351febcee4:/# pip install selenium
Downloading/unpacking selenium==2.47.3
[...]
Successfully installed selenium==2.47.3
Cleaning up...
root@ef351febcee4:/# python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from selenium import webdriver
>>> b = webdriver.Firefox()
```

正如所看到的，Firefox浏览器已经启动并出现在屏幕上！上述代码所做的是安装Python、Firefox以及Python包管理器。然后它使用Python包管理器来安装Selenium的Python软件包。

现在可以对Selenium进行尝试。下面是一个针对GitHub运行的示

例会话——要理解这里的内容需要对CSS选择器有一个基本的了解。值得注意的是，网站经常会变，因此要让这段特定的代码正确地工作可能需要做一些修改：

```
>>> b.get('http://github.com')
>>> searchselector = '.js-site-search-form input[type="text"]'
>>> searchbox = b.find_element_by_css_selector(searchselector)
>>> searchbox.send_keys('docker-in-practice\n')
>>> usersxpath = '//nav//a[contains(text(), "Users")]']'
>>> userslink = b.find_element_by_xpath(usersxpath)
>>> userslink.click()
>>> dlinkselector = '.user-list-info a'
>>> dlink = b.find_elements_by_css_selector(dlinkselector)[0]
>>> dlink.click()
>>> mlinkselector = '.org-header a.meta-link'
>>> mlink = b.find_element_by_css_selector(mlinkselector)
>>> mlink.click()
```

这里的细节并不重要。只需要注意，我们在容器里使用Python编写命令，并看到它们在运行于容器内部的Firefox窗口中生效，却显示在桌面上。

这对于调试用户编写的测试非常有用，但要如何使用同一个Docker镜像将它们整合到一个CI流水线里呢？一个CI服务器通常不需要图形显示器，因此无须挂载自己的X服务器套接字即可工作，但Firefox仍然需要运行在一个X服务器里。有一个很有用的工具应运而生，它的名字叫xvfb，它会伪装运行一个可供应用程序使用的X服务器，但它不需要显示器。

为了看一下这是如何工作的，现在来安装xvfb，提交这个容器，给它打上selenium标签，并创建一个测试脚本：

```
>>> exit()
root@ef351febcee4:/# apt-get install -y xvfb
[...]
root@ef351febcee4:/# exit
$ docker commit ef351febcee4 selenium
```

```
d1cbfbc76790cae5f4ae95805a8ca4fc4cd1353c72d7a90b90ccfb79de4f2f9b
$ cat > myscript.py << EOF
from selenium import webdriver
b = webdriver.Firefox()
print 'Visiting github'
b.get('http://github.com')
print 'Performing search'
searchselector = '.js-site-search-form input[type="text"]'
searchbox = b.find_element_by_css_selector(searchselector)
searchbox.send_keys('docker-in-practice\n')
print 'Switching to user search'
usersxpath = '//nav//a[contains(text(), "Users")]'
userslink = b.find_element_by_xpath(usersxpath)
userslink.click()
print 'Opening docker in practice user page'
dlinkselector = '.user-list-info a'
dlink = b.find_elements_by_css_selector(dlinkselector)[99]
dlink.click()
print 'Visiting docker in practice site'
mlinkselector = '.org-header a.meta-link'
mlink = b.find_element_by_css_selector(mlinkselector)
mlink.click()
print 'Done!'
EOF
```

注意dlink变量赋值语句的细微差异。通过尝试获取包含文本“Docker in Practice”的第100个结果，将触发一个错误，这将导致Docker容器以非零状态退出，然后在CI流水线中触发故障。

马上来试试：

```
$ docker run --rm -v $(pwd):/mnt selenium sh -c \
"xvfb-run -s '-screen 0 1024x768x24 -extension RANDR'\
python /mnt/myscript.py"
Visiting github
Performing search
Switching to user search
Opening docker in practice user page
Traceback (most recent call last):
  File "myscript.py", line 15, in <module>
    dlink = b.find_elements_by_css_selector(dlinkselector)[99]
  IndexError: list index out of range
$ echo $?
1
```

上面运行了一个自我删除的容器，它将执行这个运行在虚拟X服务器之下的Python测试脚本。和预期一样，它失败了并返回一个非零的退出码。

CMD与入口点 `sh -c "命令字符串"`是Docker对**CMD**值的默认处理方式的一个不良后果。如果将它放到Dockerfile中，就可以去除**sh -c**而将**xvfb-run**作为入口点，这样就可以运行任何所需的测试脚本了。

正如上面所演示的，Docker是一个灵活的工具，可以实现一些乍看起来很神奇的用途（如这里的图形化应用）。有人在Docker内部运行所有的图形化应用，包括游戏！我们不会这么疯狂，不过我们发现重新审视对于Docker的假设可能会带来令人意想不到的使用场景。

6.3 容器化CI过程

一旦团队之间具有一个一致的开发过程，具有一个一致的构建过程也同样重要。随机失败的构建将破坏Docker的优势。

因此，将整个CI过程容器化是合情合理的。这不仅能确保构建是可重复的，还可以将CI过程迁移到任何地方而不用担心遗落配置的某些关键部分（可能在经历种种挫折后才发现问题所在）。

在本节的技巧中，我们将使用Jenkins（因为这是使用最广泛的CI工具），不过同样的技巧对其他CI工具应该也适用。这里不会假定读者对Jenkins非常熟悉，不过也不打算对标准的测试和构建进行说明。对这里的技巧而言这类信息不是重点。

技巧59 包含一个复杂的开发环境

Docker的可移植性和轻量性，使其成为CI从节点（一台供CI主服务器连接以便执行构建的机器）的理想选择。与虚拟机从节点相比，Docker CI从节点向前迈了一大步（相对构建裸机更是一个飞跃）。它可以使用一台宿主机在多种环境上执行构建、快速销毁并创建整洁的环境来确保不受污染的构建，来使用所有熟悉的Docker工具来管理构建环境。

能把CI从节点当作另一个Docker容器这一点特别有趣。在其中一台Docker CI从节点上出现了神秘的构建失败？把镜像拉取下来，并自己尝试这个构建。

问题

想要扩展并修改Jenkins从节点。

解决方案

将从节点配置封装在一个Docker镜像中，然后部署。

讨论

很多组织会建立一个重量级的Jenkins从节点（通常与主服务器在一台宿主机上），由一个集中的IT职能部门维护，这在一段时间内会起到很有益的作用。随着时间推移，团队在不断壮大他们的代码库和分支，为了保证作业运行，需要安装、更新或变更越来越多的软件。

图6-3展示的是这种情景的一个简化版本。想象一下，数百个软件包及多重的新请求将让早已疲惫不堪的基础设施团队头痛不已。

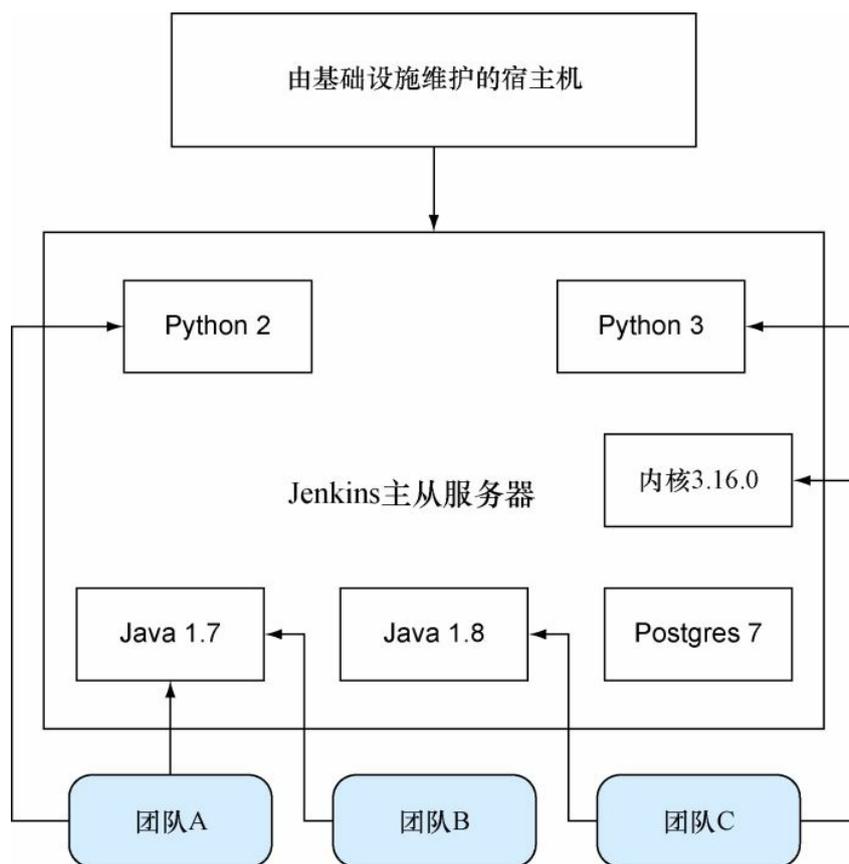


图6-3 一台超负荷的Jenkins服务器

说明性的、不可移植的示例 制定这个技巧是为了展示在一个容器内运行Jenkins从节点的基本要素。其结果的可移植性差一些，但是课程更易于掌握。一旦读者理解了本章的所有技巧，就能够创建一个更具可移植性的版本。

僵持局面随之而来，因为系统管理员担心打乱其他人的构建，可能不愿意为一群人更新他们的配置管理脚本，而变更的迟缓将使各个团队变得越来越沮丧。

Docker（天然地）提供了一个解决方案，多个团队可以使用一个基础镜像作为自己的个人Jenkins从节点，与此同时使用与之前相同的硬件。可以创建一个具有必要的共享工具的镜像，并且允许团队对

其进行变更以满足他们自己的需要。

有些贡献人员已经在Docker Hub上传了他们自用的从节点，可以在Docker Hub上通过搜索“jenkins slave”查找。代码清单6-4展示了一个最小的Jenkins从节点Dockerfile。

代码清单6-4 基础的Jenkins从节点Dockerfile^[5]

```
FROM ubuntu:14.04
ENV DEBIAN_FRONTEND noninteractive
RUN groupadd -g 1000 jenkins_slave (1)
RUN useradd -d /home/jenkins_slave -s /bin/bash \
-m jenkins_slave -u 1000 -g jenkins_slave
RUN echo jenkins_slave:jpass | chpasswd (2)
RUN apt-get update && \
apt-get install -y openssh-server openjdk-7-jre wget iproute (3)
RUN mkdir -p /var/run/sshd (4)
CMD ip route | grep "default via" \
| awk '{print $3}' && /usr/sbin/sshd -D
```

(1) 创建Jenkins从节点用户与用户组

(2) 设置Jenkins用户密码为jpass。在更复杂的设置中，最好使用其他认证方式

(3) 安装Jenkins从节点工作所需的软件

(4) 在启动时，输出以容器的角度看到的宿主机的IP地址，并启动SSH服务器

构建该从节点镜像，并给它打上jenkins_slave标签：

```
$ docker build -t jenkins_slave .
```

使用如下命令来运行它：

```
$ docker run --name jenkins_slave -ti -p 2222:22 jenkins_slave
```

Jenkins服务器必须处于运行状态

如果在宿主机上还未运行Jenkins服务器，请确保 Jenkins 服务器如前面技巧所述那样运行。如果读者比较心急，可运行下面这个命令：

```
$ docker run --name jenkins_server -p 8080:8080 -p 50000:50000 \  
dockerinpractice/jenkins:server
```

如果是在本地机器上运行这个命令，可通过<http://localhost:8080>访问Jenkins服务器。

如果浏览Jenkins服务器，将看到图6-4所示的欢迎页。

要添加一个从节点，可以点击“Build Executor Status”（构建执行器状态）>“New Node”（新节点），并添加节点的名字作为“Dump Slave”（哑节点），如图6-5所示。将其命名为mydockerslave。

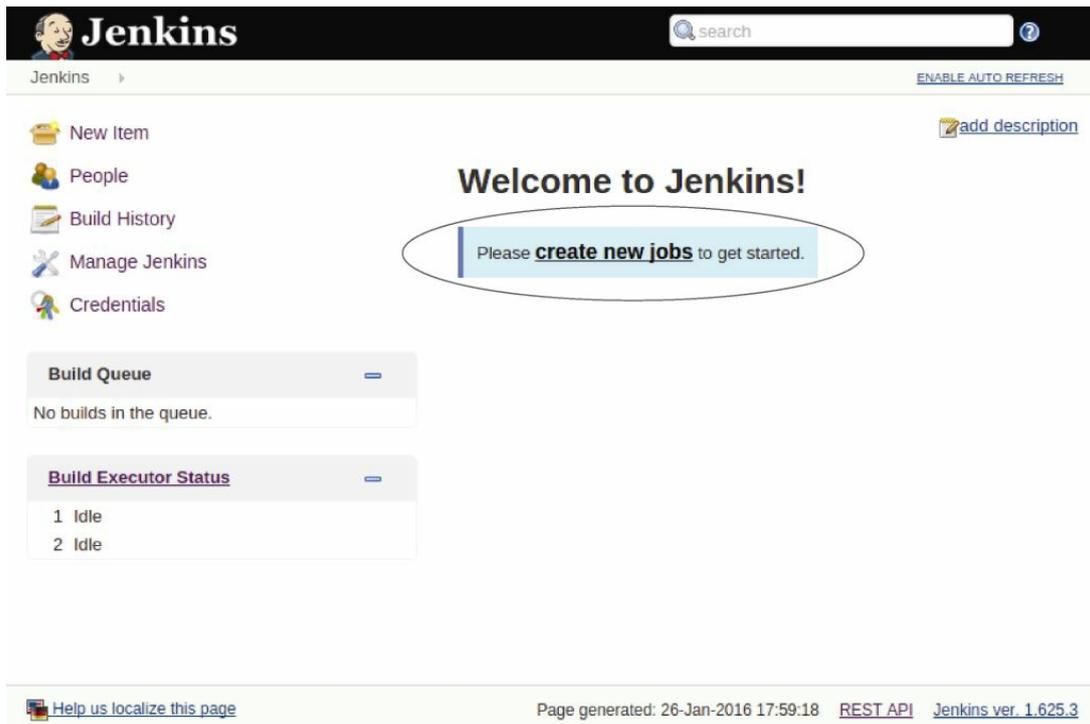


图6-4 Jenkins的欢迎页

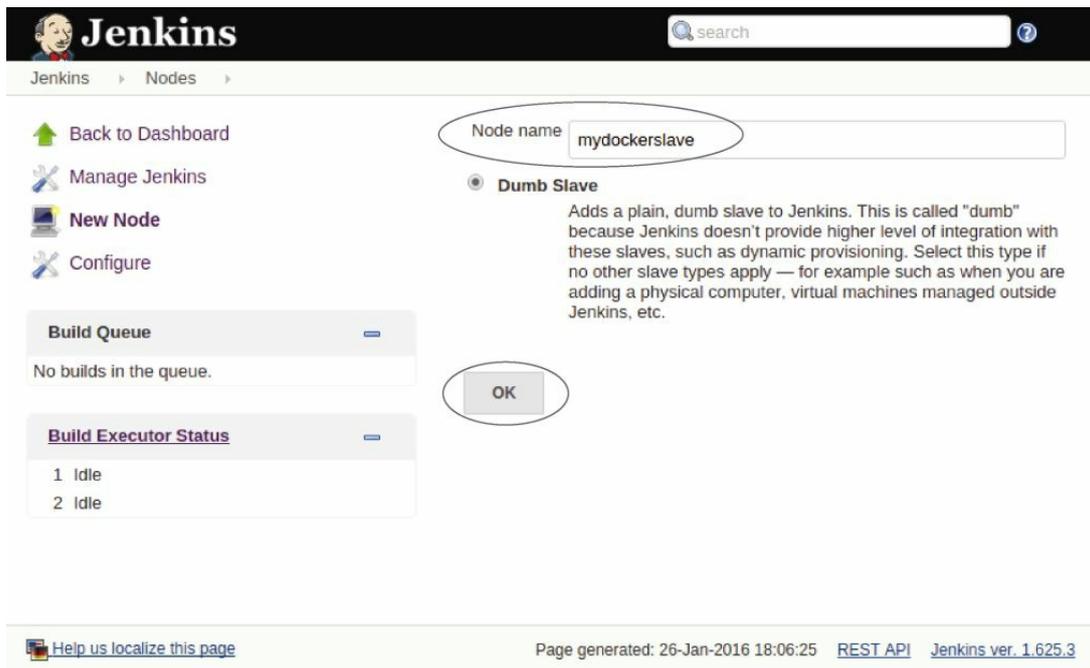


图6-5 命名一个新节点

点击“OK”并使用如下设置来配置它，如图6-6所示：

- 设置“Remote Root Directory”（远程根目录）

为/home/jenkins_slave;

- 点击“Advanced”（高级）以显示端口字段，并将其设置为2222;
- 点击“Add”（添加）来添加凭证，并将用户名设置为jenkins_slave，密码设置为jpass;
- 确保选择的是“Launch Slave Agents on Unix Machines Via SSH”（通过SSH启动Unix机器上的从节点代理）选项;
- 设置宿主机为容器内所看到的路由IP（此前docker run的输出）;
- 设置“Label”（标签）为dockerslave;
- 点击“Save”（保存）。

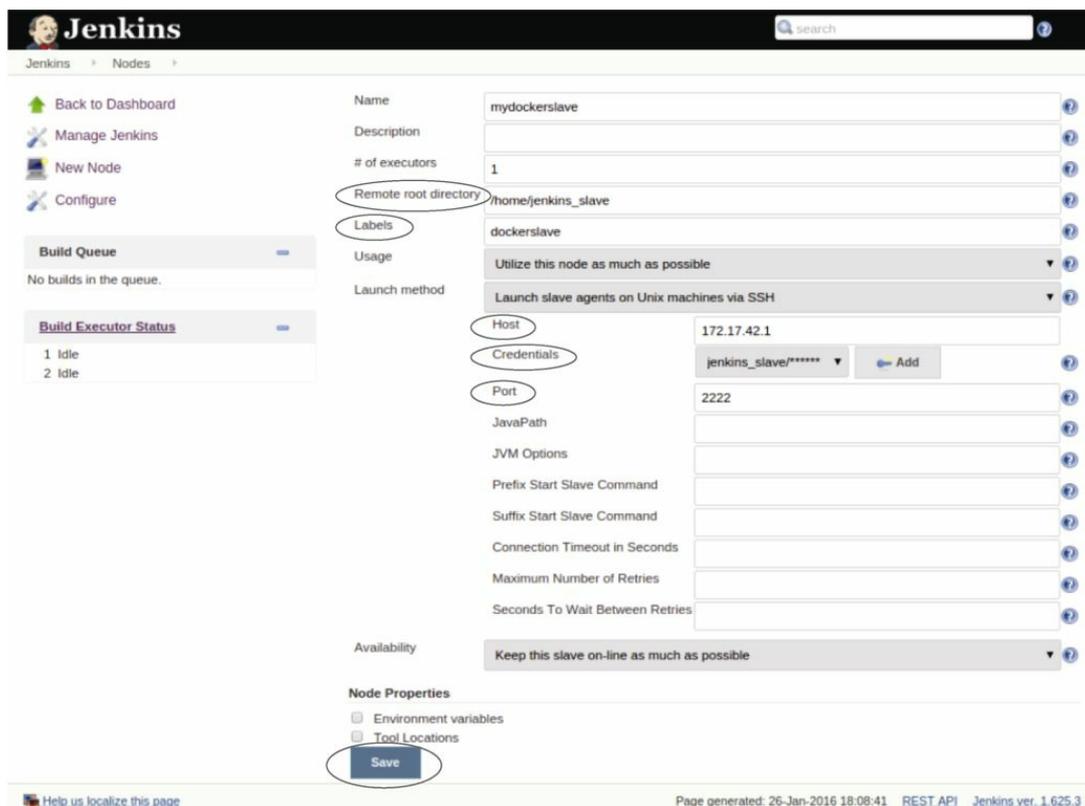


图6-6 配置新的节点

现在，点击“Launch Slave Agent”（启动从节点代理）（假设其未自动启动），将看到该从节点代理现在被标记为在线状态。

点击左上角的“Jenkins”返回到首页，并点击“New Item”（新项目）。创建一个名为test的“Freestyle Project”（自由式项目），点击“Build”（构建）区域下方的“Add Build Step”（添加构建步骤）>“Execute shell”（执行shell），并填入命令echo done。滚动到上方并选中“Restrict Where Project Can Be Run”（限制项目可运行的位置），并在“Label Expression”（标签表达式）中输入dockerslave。将看到“Slaves In Label”（标签中的从节点）被设置为1。

该作业现在被链接到Docker从节点上了。点击“Build Now”（马上构建），然后点击左侧下方出现的构建条目，然后点击“Console Output”（控制台输出），将在主窗口中看到类似这样的输出：

```
Started by user anonymous
Building remotely on testslave (dockerslave) in workspace
/home/jenkins_slave/workspace/ls
[ls] $ /bin/sh -xe /tmp/hudson4490746748063684780.sh
+ echo done
done
Finished: SUCCESS
```

干得漂亮！你已经成功创建了自己的Jenkins从节点。

现在如果读者想创建个人定制的从节点，只需要根据自己的需要修改从节点镜像的Dockerfile，并代替示例中的版本进行运行即可。

已在 **GitHub** 上可用 本技巧及其相关内容的代码可从 GitHub 获取，地址是 <https://github.com/docker-in-practice/jenkins>。

技巧60 在一个Docker容器里运行Jenkins主服务器

将 Jenkins 主服务器放到一个容器里不像从节点那样有很多好处，不过确实可以带来Docker的不可变镜像的常规优势。

我们发现，能对有效的主服务器配置和插件进行提交，可以极大地减轻实验负担。

问题

想要一个可移植的Jenkins服务器。

解决方案

使用一个Jenkins Docker镜像。

讨论

相比直接在宿主机上安装，在一个Docker容器里运行Jenkins具有一定的优势。办公室时常出现这样的叫喊：“不要动我的Jenkins服务器配置！”甚至是更糟的：“谁动了我的Jenkins服务器？”对正在运行的容器执行`docker export`可以克隆出Jenkins服务器的状态，以此进行升级和修改尝试将有助于消除这些抱怨。同样，备份和移植也变得更容易。

在本技巧中，将采用官方的Jenkins Docker镜像并做一些修改，以满足后续一些技巧对访问Docker套接字的需要，例如，在Jenkins里进行Docker构建。

直奔源代码 本书中与Jenkins相关的示例都可以在GitHub中找到，地址是<https://github.com/docker-in-practice/jenkins.git>。

共同的基础 该Jenkins镜像及其`run`命令在本书与Jenkins相关的技巧中都将作为服务器来使用。

1. 构建服务器

首先准备一个所需的服务器插件清单，并将其放置在一个名为jenkins_plugins.txt的文件中：

```
swarm:1.22
```

这个简短的清单包括了Jenkins Swarm插件（与Docker Swarm无关），这个插件在后续技巧中将会用到。

代码清单6-5展示的是构建Jenkins服务器的Dockerfile。

代码清单6-5 Jenkins服务器构建

```
FROM Jenkins (1)
COPY jenkins_plugins.txt /tmp/jenkins_plugins.txt (2)
RUN /usr/local/bin/plugins.sh /tmp/jenkins_plugins.txt (3)
USER root (4)
RUN rm /tmp/jenkins_plugins.txt
RUN groupadd -g 142 docker (5)
RUN addgroup -a jenkins docker
USER Jenkins (6)
```

- (1) 使用官方的 Jenkins 镜像作为基础
- (2) 复制要安装的插件清单
- (3) 将插件安装到服务器中
- (4) 切换到 root 用户并删除插件文件
- (5) 使用与宿主机相同的用户组 ID 将 Docker 组添加到容器中（此数字可能与读者的有所不同）
- (6) 切换回容器里的 Jenkins 用户

这里没有CMD或ENTRYPOINT指令，因为要继承官方Jenkins镜像中定义的启动命令。

读者的宿主机上的Docker的用户组ID可能会不一样。要想查看这个ID，可运行下面这条命令来查看本地用户组ID：

```
$ grep -w ^docker /etc/group
docker:x:142:imiell
```

如果这个值不是142，则使用相应值来替换它。

跨环境匹配用户组 ID 如果打算在 Jenkins Docker 容器中运行 Docker，Jenkins 服务器环境及Jenkins从节点环境中的用户组ID必须一致。在这种情况下，在迁移服务器时将可能出现移植问题（读者应该已经在原生服务器安装中遇到过同样的问题）。环境变量本身在这里无法起作用，因为用户组是在构建期设置的，无法进行动态配置。

运行下面这条命令来构建这个场景中的镜像：

```
docker build -t jenkins_server .
```

2. 运行服务器

现在可以使用这个命令在Docker下运行服务器：

```
docker run --name jenkins_server -p 8080:8080 \ (1)
-p 50000:50000 \ (2)
-v /var/run/docker.sock:/var/run/docker.sock \ (3)
-v /tmp:/var/jenkins_home \ (4)
-d \ (5)
jenkins_server
```

- (1) 将Jenkins服务器端口映射到宿主机的8080端口上
- (2) 如果想附加构建从服务器，容器的50000端口需要打开
- (3) 挂载Docker套接字，以便能在容器里与Docker守护进程互动
- (4) 将 Jenkins 应用程序数据挂载到宿主机的/tmp上，这样就不会出现文件权限错误。如果要投入实际使用，可将其挂载到一个任何人都可写的目录上
- (5) 以守护进程来运行该服务器

如果访问<http://localhost:8080>，将看到Jenkins服务器已准备就绪，并安装了相应的插件。要确认这一点，可进入“Manage Jenkins”（管理Jenkins）>“Manage Plugins”（管理插件）>“Installed”（已安装）并查找Swarm来确认它已被安装。

已在 **GitHub** 上可用 本技巧及其相关内容的代码可从GitHub获取，地址是 <https://github.com/docker-in-practice/jenkins>。

技巧61 使用Jenkins的Swarm插件扩展CI

能再现环境是一个巨大的胜利，但构建能力还是受限于所拥有的专用于构建的机器的数量。如果想借用新发现的Docker从节点的灵活性在不同的环境上做实验，其结果可能会让人沮丧。构建能力可能也会因为更现实的原因——团队的成长——而变成一个问题！

问题

想要CI运算能力与开发工作效率一起提高。

解决方案

使用Jenkins的Swarm插件及一个Docker Swarm从节点来动态地配备Jenkins从节点。

讨论

很多中小型企业具有这样的CI模型：一台或多台Jenkins服务器致力于提供运行Jenkins作业所需的资源。图6-7展示了这一点。

这在一段时间内可以运行良好，但随着CI过程变得越来越内嵌，经常会达到其容量限制。多数的Jenkins工作负载是受代码控制的签入动作触发的，因此当更多的开发人员进行签入时，其工作负载将上升。由于忙碌的开发人员对构建结果的等待忍耐有限，对运维团队的投诉数量将会激增。

一个巧妙的解决方案是运行与签入代码人数相当的Jenkins从节点，如图6-8所示。

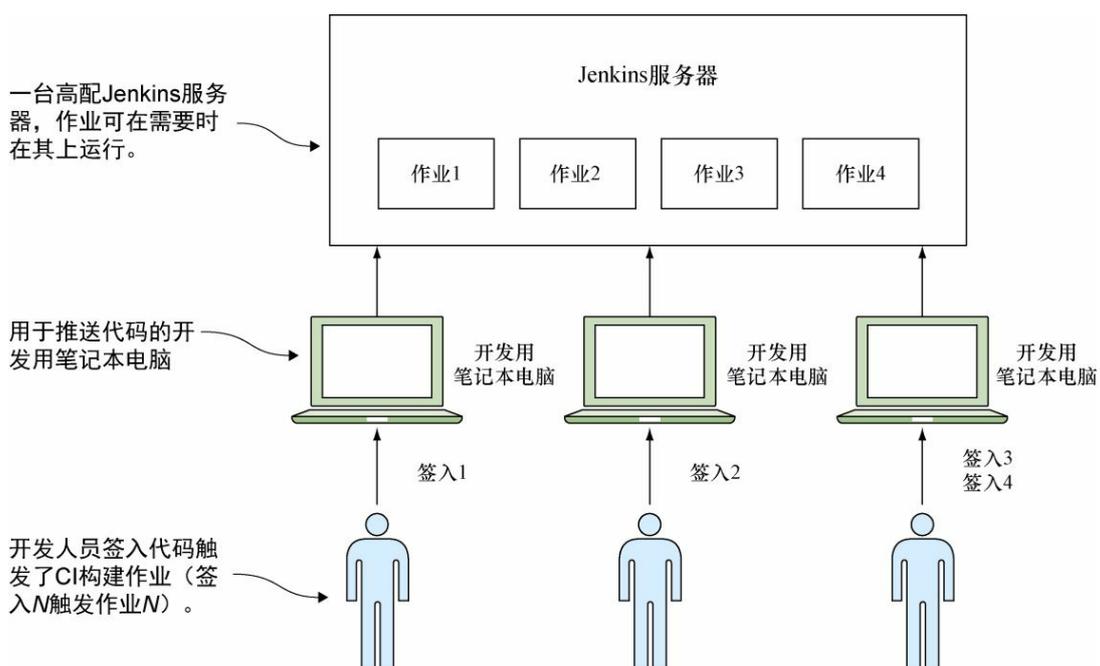


图6-7 之前：Jenkins服务器——只有一个开发人员时没问题，但无法扩展

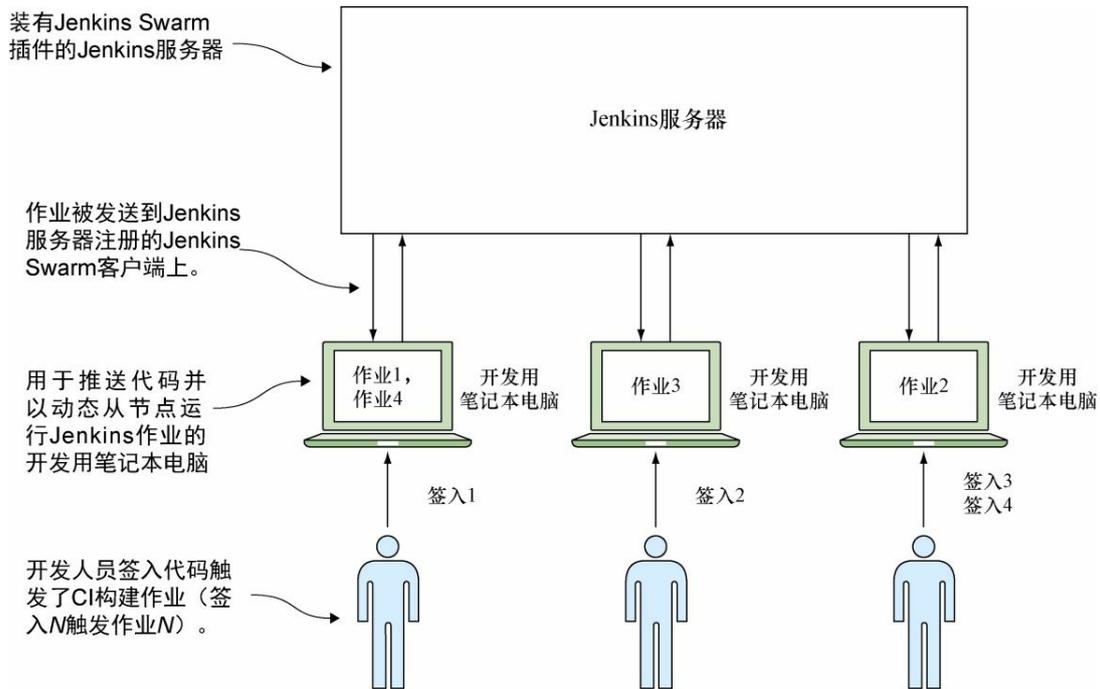


图6-8 之后：运算能力随着团队提升

代码清单6-6中所示的Dockerfile创建的是一个安装了Jenkins Swarm客户端插件的镜像，允许具有恰当的Jenkins Swarm服务器插件的Jenkins主服务器进行连接并运行作业。它与上一个技巧中正常的Jenkins从节点Dockerfile的启动方式相同。

代码清单6-6 Dockerfile^[6]

```
FROM ubuntu:14.04
ENV DEBIAN_FRONTEND noninteractive
RUN groupadd -g 1000 jenkins_slave
RUN useradd -d /home/jenkins_slave -s /bin/bash \
-m jenkins_slave -u 1000 -g jenkins_slave
RUN echo jenkins_slave:jpass | chpasswd
RUN apt-get update && apt-get install -y openjdk-7-jre wget unzip
RUN wget -O /home/jenkins_slave/swarm-client-1.22-jar-with-dependencies
.jar \
http://maven.jenkins-ci.org/content/repositories/releases/org/jenkins-ci/
plugins/swarm-client/1.22/swarm-client-1.22-jar-with-dependencies.jar (1)
COPY startup.sh /usr/bin/startup.sh (2)
```

```
RUN chmod +x /usr/bin/startup.sh (3)
ENTRYPOINT ["/usr/bin/startup.sh"] (4)
```

- (1) 获取Jenkins Swarm插件
- (2) 将启动脚本复制到容器中
- (3) 将启动脚本标记为可执行
- (4) 将启动脚本设置为默认的运行命令

代码清单6-7给出的是复制到上述Dockerfile的启动脚本。

代码清单6-7 startup.sh

```
#!/bin/bash
HOST_IP=$(ip route | grep ^default | awk '{print $3}') (1)
DOCKER_IP=${DOCKER_IP:-$HOST_IP} (2)
JENKINS_PORT=${JENKINS_PORT:-8080} (3)
JENKINS_LABELS=${JENKINS_LABELS:-swarm} (4)
JENKINS_HOME=${JENKINS_HOME:-$HOME} (5)
echo "Starting up swarm client with args:"
echo "$@"
echo "and env:"
echo "$(env)"
set -x (6)
java -jar \ (7)
/home/jenkins_slave/swarm-client-1.22-jar-with-dependencies.jar \
  -fsroot "$JENKINS_HOME" \ (8)
  -labels "$JENKINS_LABELS" \ (9)
  -master http://$DOCKER_IP:$JENKINS_PORT $@ (10)
sleep infinity (11)
```

- (1) 确定宿主机的IP地址
- (2) 除非DOCKER_IP在调用该脚本的环境变量中做了设置，否则使用宿主机ID作为Docker IP
- (3) 设置Jenkins端口为默认的8080

- (4) 设置该从节点的Jenkins标签为swarm
- (5) 设置Jenkins主目录默认为jenkins_slave用户的主目录
- (6) 从此处开始将运行的命令作为脚本的输出部分进行记录
- (7) 运行Jenkins Swarm客户端
- (8) 将Jenkins的主目录设置为根目录
- (9) 设置标签用于识别执行作业的客户端
- (10) 设置从节点所要指向的Jenkins服务器
- (11) 确保脚本（也就是容器）永远运行下去

上述脚本的大部分是在为最后的Java调用设置和输出环境变量。此Java调用将运行Swarm客户端，把运行它的机器转换成一个动态Jenkins从节点，其根目录在`-fsroot`标志中指定，运行由`-labels`标志标记的作业并指向由`-master`标志指定的Jenkins服务器。具有`echo`的几行只是提供一些参数和环境设置的调试信息。

构建和运行该容器很简单，按众所周知的样板运行即可：

```
$ docker build -t jenkins_swarm_slave .
$ docker run -d --name \
jenkins_swarm_slave jenkins_swarm_slave
```

现在已经在这台机器上设置了一个从节点，可以在上面运行Jenkins作业了。可像平常那样设置一个Jenkins作业，只不过在Restrict Where This Project Can Be Run（限制项目可运行的位置）中添加`swarm`作为一个标签表达式（见技巧59）。

建立一个系统服务来传播 可以在所有的PC上将其设置为 supervised 系统服务来实现这个过程的自动化（见技巧75）。

对从节点机器的性能冲击 Jenkins作业可能是一些繁重的进程，其运行完全有可能对笔记本电脑造成负面影响。如果该作业很繁重，可以在作业和相应的Swarm客户端上设置标签。例如，可以设置一个作业的标签为4CPU8G，并将它匹配到运行在具有8 GB内存、4个CPU的机器的Swarm容器。

本技巧对Docker概念做了一些展示。一个可预测、可移植的环境可以放置在多台宿主机上，从而降低昂贵的服务器的负载，并将所需配置降到最低。

尽管本技巧实施时不能不考虑性能，我们还是认为这里面有很大的发挥空间，可以将开发人员的计算机资源转换成某种形式的游戏，从而提升开发组织的工作效率，而无须昂贵的新硬件。

已在 **GitHub** 上可用 本技巧及其相关内容的代码可从GitHub获取，地址是<https://github.com/docker-in-practice/jenkins>。

6.4 小结

本章中展示了如何在组织内部使用Docker来启用和推动CI。读者可以看到CI的很多障碍在Docker的协助下可以被扫清，如原始运算能力的可用性、与他人共享资源等。

在本章中，读者了解到：

- 通过使用eatmydata及软件包缓存，可以极大提升构建速度；
- 可以在Docker内运行GUI测试（如Selenium）；
- 一个Docker CI 从节点可以保持对环境的完全控制；
- 使用Docker及Jenkins的Swarm插件，可以将构建进程分包到整个团队中。

在下一章中，我们将从CI转移到部署，并讨论与持续交付相关的技巧，这是DevOps图景的另一个关键组成部分。

[1] 读者看到这本书的时候，DockerHub的界面可能已经更新，操作的按钮可能有所不同。翻译本书时，界面中创建仓库需要点击右上角的“Create”（创建）菜单并选择“CreateAutomatedBuild”（创建自动化构建）。——译者注

[2] “BuildSettings”（构建设置）页面中的“Trigger”（触发）按钮。——译者注

[3] 翻译本书时，界面中“Pending”已经更换成“Queued”。——译者注

[4] 翻译本书时，界面中“Finished”已经更换成“Success”。——译者注

[5] 原文未指定版本，翻译本书时的最新版的镜像会出现jre无法安装以及ip命令不存在的问题。——译者注

[6] 原文未指定版本，最新版的镜像可能会出现jre无法安装以及ip命令不存在的问题。——译者注

第7章 持续交付：与Docker原则完美契合

本章主要内容

- 开发与运维之间的Docker契约
- 掌控跨环境的构建可用性
- 通过低带宽连接在不同环境间迁移构建
- 集中配置一个环境中的所有容器
- 使用Docker实现零停机时间部署

一旦确信使用一致的持续集成（CI）过程对所有的构建都进行了质量检验，下一步自然是开始着手将每个良好的构建部署给用户。这个目标称为持续交付（continuous delivery, CD）。

本章涉及的就是CD流水线——构建从CI流水线出来后所经历的过程。这两者的分界点有时会比较模糊，不过可以这么认为，在构建过程中通过初始测试获得最终镜像的那一刻即是CD流水线开始的时刻。图7-1演示了镜像在到达生产环境（但愿如此）之前是如何通过CD流水线的。

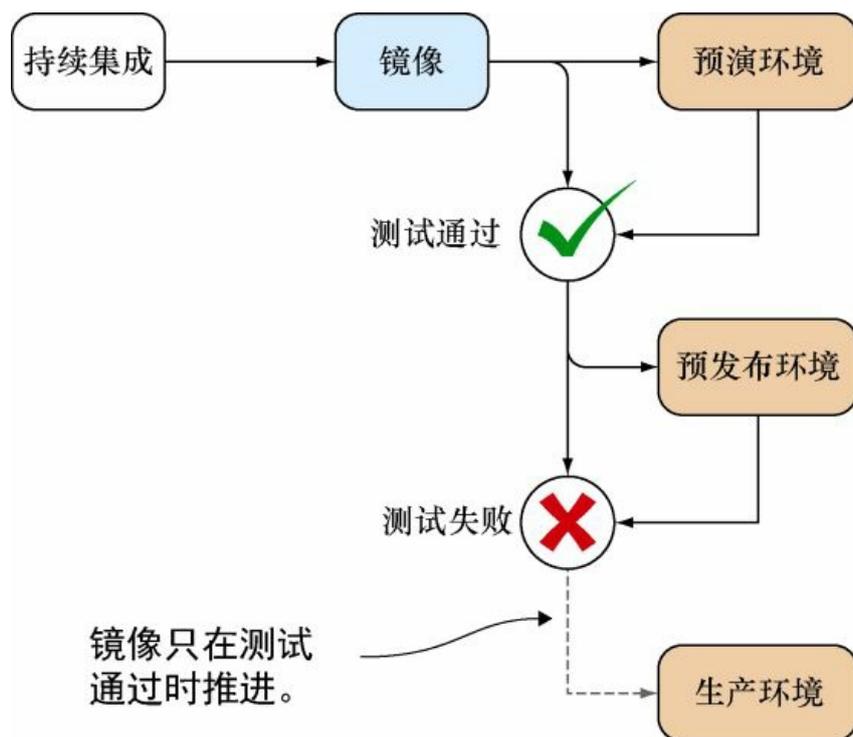


图7-1 一个典型的CD流水线

最后一点值得再提一下，在CD全过程中，从CI产出的镜像必须是最终的、不可修改的！ Docker通过不可变镜像及状态封装很容易实现这一点，因此使用Docker已经让你在CD的路上前进了一步。

7.1 在CD流水线上与其他团队互动

先回头看一下Docker是如何改变开发团队与运维团队之间的关系。

软件开发的一些最大的挑战不是技术性的——按照角色和技能将人员划分成团队是一个普遍做法，但是这会造成沟通障碍和封闭性。一个成功的CD流水线要求在这个过程的所有阶段，从开发环境到测试环境再到生产环境，所有场景里的团队都参与进来，此时对所有团队而言，一个单一的参考点有助于通过提供结构来促进互动。

技巧62 Docker契约——减少摩擦

Docker 的目标之一是让与包含单个应用程序的容器相关联的输入与输出易于表达。在与其他人一起工作时这可以增加透明度——沟通是合作的重要环节之一，而理解Docker如何通过提供一个参考点来简化事务将有助于赢得不信任Docker的人的支持。

问题

想要合作团队的可交付成果是整洁的、明确的，从而减少交付流水线里的摩擦。

解决方案

使用Docker契约来推动团队间整洁的可交付成果。

讨论

随着公司规模扩大，经常可以看到其曾经拥有的扁平化的、精益化的组织架构——几个关键的个人“了解整个系统”，让位给了一个更加结构化的组织架构——不同的团队具有不同的职责和能力。我们在效力过的组织中都对此有过切身体会。

如果没有进行技术投入，随着团队之间相互交付的增多，摩擦也会不断升级。对日益增长的复杂度的抱怨——“把这个版本扔出去！”以及问题一堆的升级将变得稀松平常。“呃，在我们的机器上是正常的！”这样的叫喊声不绝于耳，所有各方都感到失望。图7-2展示了这个场景的一个简化了但具有代表性的情形。

图7-2中的工作流有几个大家熟知的问题。这些问题最终都归结于状态管理的困难。测试团队可能在一台不是运维团队所设置的机器上进行测试。理论上，对所有环境的修改都应仔细地记录下来，并在

出现问题时进行回滚以保持一致性。但是，商业压力与人类行为的存在总是破坏这个目标，造成环境性漂移。

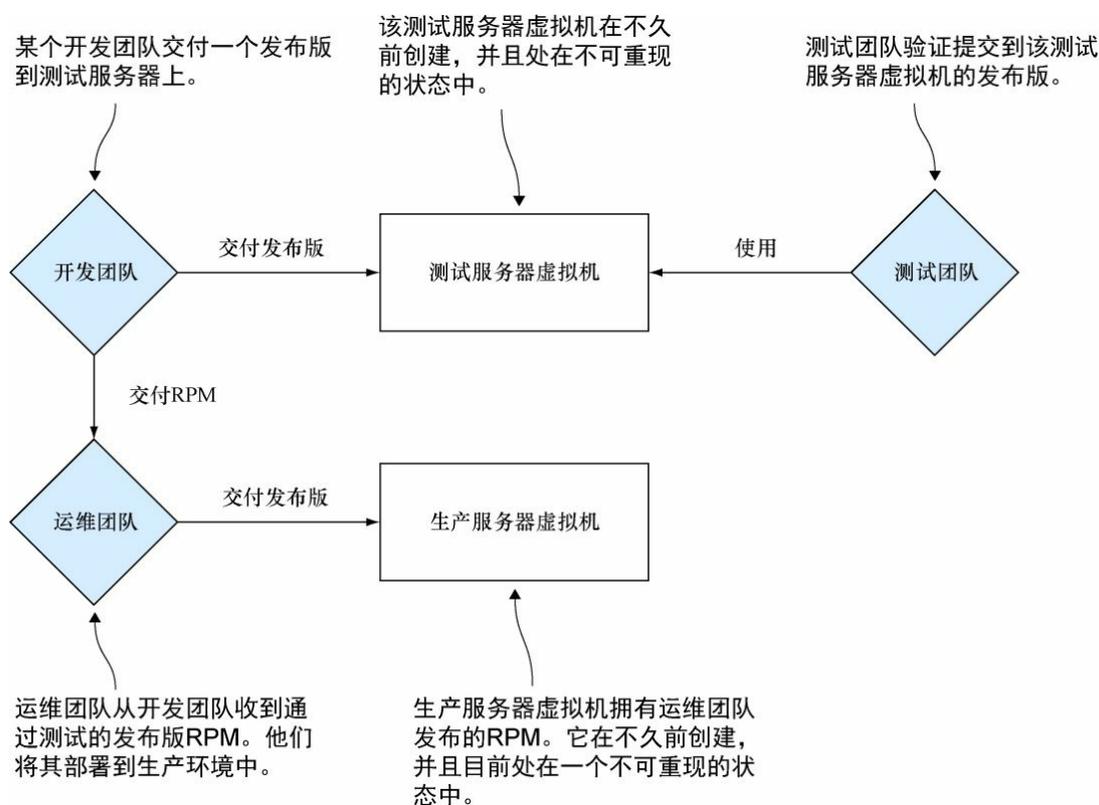


图7-2 之前：一个典型的软件 workflow

对这一问题的现有解决方案包括虚拟机及RPM。虚拟机通过交付完整的机器表示来减少环境引发风险的可能性。其缺点在于虚拟机是相对单一的实体，对各个团队来说很难有效地进行操作。RPM提供了一种打包应用程序的标准方法，可以在交付软件时定义其依赖。这并未消除配置管理的问题，交付兄弟团队创建的RPM要远比使用互联网上久经考验的RPM问题多得多。

Docker 契约

Docker所能做的是在团队之间划出清晰的分界线，Docker镜像既是分界线，又是交换的单位。我们称其为Docker契约，如图7-3所

示。

使用Docker，所有团队的参考点变成更加清晰了。与处理处于不可重现状态的庞大的单体虚拟机（或真机）不同，所有团队面对的是相同的代码，而不论是在测试环境、生产环境还是开发环境。此外，代码与数据有了一个清晰的分离，更易于推断问题是由数据的变化造成的还是由代码的变化造成的。

因为Docker使用非常稳定的Linux API作为环境，交付软件的团队具有更大的自由度使用他们喜欢的风格来构建软件和服务，并确信它可在不同环境中按预期运行。这不代表可以忽略它运行所在的环境，但它确实减少了环境差异造成问题的风险。

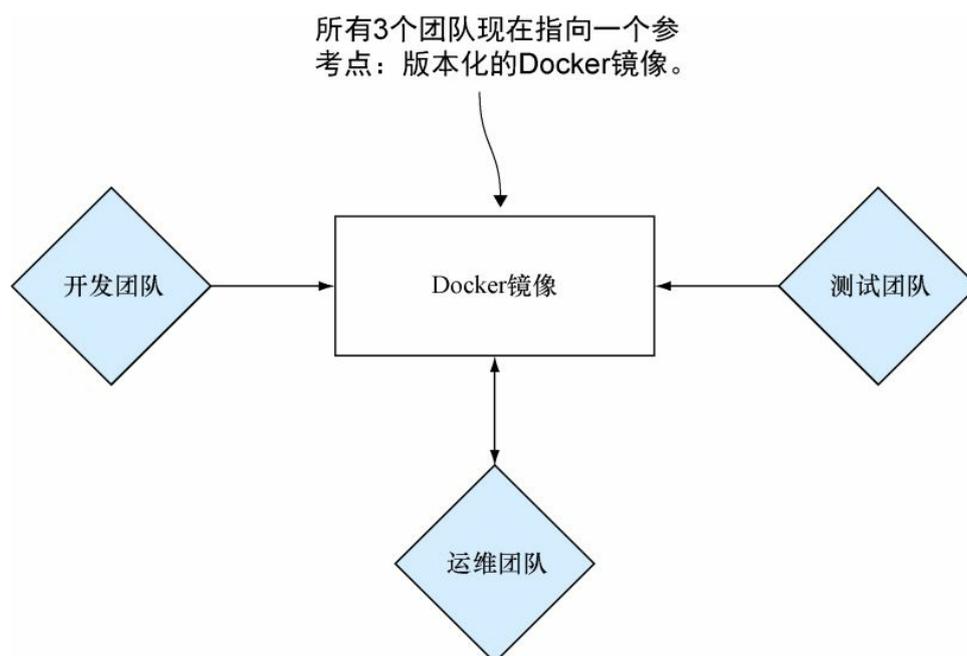


图7-3 之后：Docker契约

这样单一参考接触点带来的是各种运维效率。问题重现变得更加简单，所有团队都能从一个已知的起点描述并重现问题。升级变成了交付变更的团队的责任。简而言之，状态由那些做变更的人管理。所有这些优点极大地降低了沟通开销，让各个团队能继续他们的工作。

沟通开销的降低还有助于向微服务构架进行迁移。

这不只是理论上的好处：我们在一家拥有超过500名开发人员的公司亲身体会了这种提升，它也是各类Docker技术聚会中一个频繁讨论的话题。

7.2 推动Docker镜像的部署

在尝试实现CD时，首要的问题是将构建过程的产出移动到合适的位置。如果对CD流水线中所有场景使用同一个注册中心，看起来问题好像是解决了。但这并未涵盖CD的一个关键方面。

CD背后的关键思想之一是构建提升。构建提升是指流水线的每个场景（用户验收测试、集成测试以及性能测试）只有在前一个场景成功时才能触发下一个场景。使用多个注册中心时，只在某个构建场景通过时才将其提交到下一个注册中心中，就可确保使用的是提升后的构建。

下面讲述在注册中心间迁移镜像的几种方法，甚至有一个无须注册中心即可共享Docker对象的方法。

技巧63 手动同步注册中心镜像

最简单的镜像同步情景是有一台与两个registry高速连接的机器。使用正常的Docker功能即可完成镜像复制。

问题

想要在两个注册中心之间复制一个镜像。

解决方案

拉取镜像，重新打标签，然后推送。

讨论

假设在test-registry.company.com有一个镜像，想要移动到stage-registry.company.com，这个过程很简单：

```
$ IMAGE=mygroup/myimage:mytag
$ OLDREG=test-registry.company.com
$ NEWREG=stage-registry.company.com
$ docker pull $OLDREG/$MYIMAGE
[...]
$ docker tag -f $OLDREG/$MYIMAGE $NEWREG/$MYIMAGE
$ docker push $NEWREG/$MYIMAGE
$ docker rmi $OLDREG/$MYIMAGE
$ docker rmi $(docker images -q --filter dangling=true)
```

这个过程有以下3个重要的点需要注意。

(1) 新镜像被强制打了标签。这意味着这台机器上任何具有相同名称的旧镜像（用于层缓存而留在那儿）将丢失镜像名称，因而新的镜像可以使用所需的名称打上标签。

(2) 所有没有标签的镜像已经被删除了。虽然层缓存对加速部署非常有用，但是留着无用的镜像层会很快耗尽磁盘空间。一般而言，随着时间过去，旧的层很少会被使用，变得越来越过时。

(3) 可能需要使用docker login登录到新的注册中心中。

现在，该镜像就在新的注册中心里可用了，可供CD流水线的后续场景使用。

技巧64 通过受限连接交付镜像

即使是使用了分层，推送和拉取Docker镜像仍然是一个耗费带宽

的过程。在具有免费大带宽的世界，这不成问题，但现实有时会迫使我们去处理两个数据中心之间低宽带连接或昂贵的宽带计费的问题。在这种情况下，需要找到一种更加高效的传输差异部分的方法，否则一天运行多次流水线的CD憧憬将遥不可及。

理想的解决方案是一个可以降低镜像的平均尺寸的工具，甚至比典型的压缩方法可达到的更小。

问题

想要在两台使用低带宽连接的机器之间复制镜像。

解决方案

导出镜像，使用bup进行拆分，传输bup块，并在另一端导入重新组合的镜像。

讨论

首先需要介绍一个新的工具bup。它是作为一个备份工具创造出来的，具有极其高效的去重能力——去重能力是指识别重复使用的数据，只存储一份副本。去重在其他情形中也非常有用，例如，传输多个具有非常相似内容的镜像。我们为这个技巧创建了一个名为dbup（“docker bup”的简称）的镜像，可更容易地使用bup对镜像进行去重。可在<https://github.com/docker-in-practice/dbup>找到其背后的代码。

作为一个示范，来看看从ubuntu:14.04.1镜像升级到ubuntu:14.04.2时可以节省多少宽带。需要牢记的一点是，在现实中这些镜像上面都有多个层，Docker会在下面的层变更时进行完全的重新传输。相比之下，本技巧将识别出高度的相似性，节省大量的带宽。

第一步是把两个镜像都拉取下来以便查看通过网络传输了多少流量：

```
$ docker pull ubuntu:14.04.1 && docker pull ubuntu:14.04.2
[...]
$ docker history ubuntu:14.04.1
IMAGE          CREATED          CREATED BY
SIZE
5ba9dab47459   3 months ago    /bin/sh -c #(nop) CMD ["/bin/bash]
0 B
51a9c7c1f8bb   3 months ago    /bin/sh -c sed -i 's/^#\s*\s*(deb.*universe\)$
/ 1.895 kB
5f92234dcf1e   3 months ago    /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic
c 194.5 kB
27d47432a69b   3 months ago    /bin/sh -c #(nop) ADD file:62400a49cced0d752
1 188.1 MB
511136ea3c5a   23 months ago
0 B
$ docker history ubuntu:14.04.2
IMAGE          CREATED          CREATED BY
SIZE
07f8e8c5e660   2 weeks ago     /bin/sh -c #(nop) CMD ["/bin/bash"]
0 B
37bea4ee0c81   2 weeks ago     /bin/sh -c sed -i 's/^#\s*\s*(deb.*universe\)$
/ 1.895 kB
a82efea989f9   2 weeks ago     /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic
194.5 kB
e9e06b06e14c   2 weeks ago     /bin/sh -c #(nop) ADD file:f4d7b4b3402b5c53f2
188.1 MB
$ docker save ubuntu:14.04.1 | gzip | wc -c
65970990
$ docker save ubuntu:14.04.2 | gzip | wc -c
65978132
```

这里演示的是Ubuntu镜像间不共享层，因此可将整个镜像尺寸作为推送新镜像时将传输的数据量。同样需要注意的是，Docker注册中心使用gzip压缩来传输层，因此在测量时也将其加入（而不是从docker history获得尺寸）。在初始部署及后续部署中都大概传输了65 MB。

在开始前，需要准备两样东西——一个bup作为内部存储器用于存储数据“池”的目录，以及dockerinpractice/dbup镜像。然后就可以将

镜像添加到bup数据池中了：

```
$ mkdir bup_pool
$ alias dbup="docker run --rm \
  -v $(pwd)/bup_pool:/pool -v /var/run/docker.sock:/var/run/docker.sock \
  dockerinpractice/dbup"
$ dbup save ubuntu:14.04.1
Saving image!
Done!
$ du -sh bup_pool
74M    bup_pool
$ dbup save ubuntu:14.04.2
Saving image!
Done!
$ du -sh bup_pool
90M    bup_pool
```

将第二个镜像添加到bup数据池中只增加了大概15 MB的大小。假设在添加ubuntu:14.04.1后已经将这个目录同步到另一台机器上（可能使用的是rsync），再次同步这个目录将只传输15 MB（而不是之前的65 MB）。

在另一端需要这样加载镜像：

```
$ dbup load ubuntu:14.04.1
Loading image!
Done!
```

在两个注册中心之间传输的过程将类似下面这样：

- (1) 在主机1上docker pull;
- (2) 在主机1上dbup save;
- (3) 从主机1 rsync到主机2;
- (4) 在主机2上dbup load;

(5) 在主机2上docker push。

本技巧将之前的很多不可能变成了可能。例如，现在可以重新安排和合并层，而无须考虑通过低带宽连接传输所有的新层需要花费多长时间。

即使是遵从最佳实践将应用程序代码作为最后一个环节进行添加，bup也能帮上忙——它将识别出大部分未修改的代码，而只将差异部分添加到数据池中。

虽然读者对这个方法可能没有迫切的需要，但请在宽带账单开始上升时记起这一点！

技巧65 以TAR文件方式共享Docker对象

TAR文件是Linux上移动文件的一种传统方法。在没有注册中心或者没必要设立注册中心的情况下，可以使用Docker手工创建TAR文件然后进行移动。下面展示这些命令的详细内容。

问题

想要在没有注册中心的情况下与其他人共享镜像和容器。

解决方案

使用docker export或者docker save创建 TAR 文件，然后经由 SSH 使用docker import或docker load来使用它们。

讨论

如果只是随意地使用这些命令，它们之间的区别将难于掌握，因此下面花点儿时间快速看一下它们都做了什么。表7-1概述了这些命

令的输入与输出。

表7-1 **export**和**import**与**save**和**load**的对比

命令	创建了	目标类型	来源
export	TAR文件	容器文件系统	容器
import	Docker镜像	平面文件系统	TAR包
save	TAR文件	Docker镜像（带历史记录）	镜像
load	Docker镜像	Docker镜像（带历史记录）	TAR包

前两个命令使用平面文件系统。**docker export**命令输出一个TAR文件，这个TAR文件包含了组成容器状态的文件。在Docker中，正在运行进程的状态不会被保存——只有文件。**docker import**命令从TAR包创建Docker镜像——没有历史也没有元数据。

这些命令不是对称的——无法仅使用导入和导出从现有容器创建一个容器。这种不对称非常有用，因为可以使用**docker export**将一个镜像导出成一个TAR文件，然后使用**docker import**导入从而“丢弃”所有的层历史及元数据。这就是技巧43中描述的镜像扁平化方法。

在导出或保存成TAR文件时，文件会被默认发送到标准输出（**stdout**）中，因此需要像下面这样确保将其保存到文件中：

```
docker pull debian:7:3
```

```
[...]
docker save debian:7.3 > debian7_3.tar
```

然后就可以放心地在网络上传送这个TAR文件了，而其他人可以使用它们完好地导入镜像。可以通过电子邮件或scp来传输：

```
$ scp debian7_3.tar example.com:/tmp/debian7_3.tar
```

如果拥有相应权限，还可以更进一步，直接将镜像发送给其他用户的Docker守护进程：

```
docker save debian:7.3 | \ (1)
ssh example.com \ (2)
docker import - (3)
```

(1) `docker save`命令将7.3版本的Debian提取出来，并通过管道发送给ssh命令

(2) `ssh`命令在远程机器example.com上运行命令

(3) `docker import`命令从赋予它的TAR文件创建一个没有历史的镜像。-表示TAR文件是通过标准输入获取的

如果要保留镜像的历史，可以使用`load`而不是`import`，这样其历史将在另一边的Docker守护进程上得到保留：

```
docker save debian:7.3 | ssh example.com docker load
```

docker load与**docker import**不一致性 与**docker import**不同，**docker load**不需要在最后使用一个-来表示TAR文件是通过标准输入获取的。

7.3 为不同环境配置镜像

如本章介绍中提到的，CD的一个重点是“在所有地方完成同样的事情”的概念。在没有Docker的情况下，这意味着要进行一次部署工件的构建，并在所有地方使用同一个工件。在一个Docker化的世界，这意味着在所有地方使用同一个镜像。

不过环境并不完全相同，例如，外部服务的URL可能是不同的。对于“一般”的应用程序，可以使用环境变量来解决这个问题（需要说明的是它们不易于应用到大量机器上）。相同的解决方案对Docker也适用（明确地传入变量），不过对Docker而言还有更好的方法可以实现这一点，并且可以带来额外的好处。

技巧66 使用etcd通知容器

Docker镜像被设计成可以在任何地方部署，不过用户经常希望能在部署后添加额外信息以便影响运行中的应用程序的行为。此外，运行Docker的机器可能需要保持不变，因此需要一个外部的信息来源（这使得环境变量变得不太适合）。

问题

在运行容器时，需要一个外部的配置源。

解决方案

创建一个etcd集群来保存配置，并使用etcd代理来访问它。

讨论

etcd 是一个分布式键值存储系统——它保存着信息片段，可作为实现弹性的多节点集群的一部分。

保持简洁的配置 etcd保存的每个值都应该保持小巧——小于512 KB是一个较好的经验规则，超过这个点则应考虑进行基准测试，以确认etcd是否仍然按预期运行。这种限制不是etcd特有的，其他键值存储系统（如Zookeeper和Consul）也应注意这个问题。

因为etcd集群节点需要互相通信，所以第一步是识别外部IP地址。如果节点运行在不同机器上，则需要每一台机器的外部IP：

```
$ ip addr | grep 'inet ' | grep -v 'lo$|docker0$'  
inet 10.194.12.221/20 brd 10.194.15.255 scope global eth0
```

这里查找了所有的IPv4网卡，并将环回及Docker排除在外。这将得到所需的IP（此行的第一个）。

现在可以开始设置集群了。请谨慎修改每一行的下列参数：暴露的及公布的端口，以及集群节点与容器的名称：

```
$ IMG=quay.io/coreos/etcd:v2.0.10  
$ docker pull $IMG  
[...]  
$ HTTPIP=http://10.194.12.221 (1)  
$ CLUSTER="etcd0=$HTTPIP:2380,etcd1=$HTTPIP:2480,etcd2=$HTTPIP:2580" (2)  
$ ARGS=  
$ ARGS="$ARGS -listen-client-urls http://0.0.0.0:2379" (3)  
$ ARGS="$ARGS -listen-peer-urls http://0.0.0.0:2380" (4)  
$ ARGS="$ARGS -initial-cluster-state new"  
  
$ ARGS="$ARGS -initial-cluster $CLUSTER"  
$ docker run -d -p 2379:2379 -p 2380:2380 --name etcd0 $IMG \  
  $ARGS -name etcd0 -advertise-client-urls $HTTPIP:2379 \  
  -initial-advertise-peer-urls $HTTPIP:2380  
912390c041f8e9e71cf4cc1e51fba2a02d3cd4857d9ccd90149e21d9a5d3685b  
$ docker run -d -p 2479:2379 -p 2480:2380 --name etcd1 $IMG \  
  $ARGS -name etcd1 -advertise-client-urls $HTTPIP:2479 \  
  -initial-advertise-peer-urls $HTTPIP:2480  
446b7584a4ec747e960fe2555a9aaa2b3e2c7870097b5babe65d65cffa175dec  
$ docker run -d -p 2579:2379 -p 2580:2380 --name etcd2 $IMG \  
  $ARGS -name etcd2 -advertise-client-urls $HTTPIP:2579 \  
  -initial-advertise-peer-urls $HTTPIP:2580
```

```
-initial-advertise-peer-urls $HTTPIP:2580
3089063b6b2ba0868e0f903a3d5b22e617a240cec22ad080dd1b497ddf4736be
$ curl -L $HTTPIP:2579/version
etcd 2.0.10
```

(1) 机器的外部IP地址

(2) 在集群定义中使用机器的外部IP地址，让节点可以相互通信。因为所有的节点都在同一台宿主机上，集群的端口（用于连接其他节点）必须不同

(3) 用于处理客户端请求的端口

(4) 用于与集群其他节点通信的监听端口，与\$CLUSTER中指定的端口一致

现在集群就已经启动了，并且每个节点都有响应。在上述命令中，指向对等点（peer）的内容是在控制etcd节点如何查找其他节点并与之通信，而指向客户端的内容则定义了其他应用程序如何连接到etcd。

下面用实例来说明一下etcd的分布式特性：

```
$ curl -L $HTTPIP:2579/v2/keys/mykey -XPUT -d value="test key"
{"action":"set","node":{"key":"/mykey","value":"test key",
➤ "modifiedIndex":7,"createdIndex":7}}
$ sleep 5
$ docker kill etcd2
etcd2
$ curl -L $HTTPIP:2579/v2/keys/mykey
curl: (7) couldn't connect to host
$ curl -L $HTTPIP:2379/v2/keys/mykey
{"action":"get","node":{"key":"/mykey","value":"test key",
➤ "modifiedIndex":7,"createdIndex":7}}
```

上述代码添加了一个键到etcd2中，然后杀掉它。不过etcd已经将信息自动复制到其他节点上，因此还是能够提供该信息。尽管上述代

码暂停了5秒，etcd通常会在1秒内进行复制（即便是在跨不同机器时）。可随时运行`docker start etcd2`让其再次投入使用。

可以看出，数据还是可用的，不过必须手工选择另外一个节点进行连接显然有点儿不友好。幸好etcd为此提供了一个解决方案——可以以“代理”模式启动节点，这意味着它不复制任何数据，而只是将请求转发给其他节点：

```
$ docker run -d -p 8080:8080 --restart always --name etcd-proxy $IMG \
  -proxy on -listen-client-urls http://0.0.0.0:8080 \
  -initial-cluster $CLUSTER
037c3c3dba04826a76c1d4506c922267885edbfa690e3de6188ac6b6380717ef
$ curl -L $HTTPIP:8080/v2/keys/mykey2 -XPUT -d value="t"
{"action":"set","node":{"key":"/mykey2","value":"t",
➤ "modifiedIndex":12,"createdIndex":12}}
$ docker kill etcd1 etcd2
$ curl -L $HTTPIP:8080/v2/keys/mykey2
{"action":"get","node":{"key":"/mykey2","value":"t",
➤ "modifiedIndex":12,"createdIndex":12}}
```

这样就可以自由地体验当一半的节点离线时etcd是如何工作的了：

```
$ curl -L $HTTPIP:8080/v2/keys/mykey3 -XPUT -d value="t"
{"message":"proxy: unable to get response from 3 endpoint(s)"}
$ docker start etcd2
etcd2
$ curl -L $HTTPIP:8080/v2/keys/mykey3 -XPUT -d value="t"
{"action":"set","node":{"key":"/mykey3","value":"t",
➤ "modifiedIndex":16,"createdIndex":16}}
```

当一半或更多节点不可用时，etcd将允许读取，而禁止写入。

由此可见，可以在集群的每个节点上启动一个etcd代理作为“大使容器”（ambassador container）用于获取集中式配置：

```
$ docker run -it --rm --link etcd-proxy:etcd ubuntu:14.04.2 bash
root@8df11eaae71e:/# apt-get install -y wget
root@8df11eaae71e:/# wget -q -O- http://etcd:8080/v2/keys/mykey3
```

```
{"action": "get", "node": {"key": "/mykey3", "value": "t",  
  └─ "modifiedIndex": 16, "createdIndex": 16}}
```

什么是大使 大使是Docker用户中流行的一种所谓“Docker模式”。大使容器位于应用程序容器与外部服务之间，负责处理相应请求。它与代理类似，但融入了一些智能用于处理具体情况的需求——就像现实中的大使一样。

一旦在所有环境中运行了etcd，在某个环境中创建机器只需要将其链接到etcd-proxy容器并启动即可——所有该机器的CD构建都将使用该环境的正确配置。下一个技巧将展示如何使用etcd-provided配置来实现零停机时间升级。

7.4 升级运行中的容器

为了实现每天多次部署到生产环境的理想目标，减少部署流程最后一步——停止旧应用程序并启动新应用程序——的停机时间就非常重要了。如果每次切换都需要一小时，那么一天部署4次就没有意义！

因为容器提供了一个隔离的环境，所以很多问题已经得到缓解。例如，无须再担心一个应用程序的两个版本会使用同一工作目录并互相冲突，也无须使用新代码重启来重新读取配置文件以获取新值。

但是，这同样存在一些弊端——就地修改文件不再是一件简单的事，“软重启”（用于获取配置文件变更）因而变得更难实现。因此，一个好的做法是，不论修改的是一些配置文件还是上千行代码，永远执行相同的升级过程。

下面来看一个升级过程，它将实现面向Web的应用程序零停机时间部署的黄金标准。

技巧67 使用confd启用零停机时间切换

由于容器在一台宿主机上可以共存，删除一个容器并启动一个新容器这样的简单切换方式可以在几秒内完成（同样可以实现快速回滚）。

对大多数应用程序来说，这可能已经够快了，但那些具有较长启动时间或高可用需求的应用程序则需要另外一种方法。有时，这是一个要求应用程序自身做特殊处理的不可避免的复杂过程，不过面向Web的应用程序有一个方案可以优先考虑。

问题

想要在升级面向Web的应用程序时做到零停机时间。

解决方案

在宿主机上使用nginx和confd来执行两阶段切换。

讨论

nginx是一个非常流行的Web服务器，它具有一项重要的内置功能——在不断开客户端与服务器连接的情况下重新加载配置文件。通过将其与confd——一个可从中央数据仓库（如etcd）获取信息并对配置文件进行相应修改的工具——组合，即可使用最新的设置对etcd进行更新，然后由其完成后续处理。

Apache/HAProxy方案 Apache HTTP服务器与HAProxy二者也提供了零停机时间重载功能，对于有相应配置经验的用户也可以用其替

代nginx。

第一步是启动一个应用程序作为最终将更新的旧应用程序。

Ubuntu附带的Python具有一个内置的Web服务器，可以使用它作为示例：

```
$ ip addr | grep 'inet ' | grep -v 'lo$|docker0$'
    inet 10.194.12.221/20 brd 10.194.15.255 scope global eth0
$ HTTPIP=http://10.194.12.221
$ docker run -d --name py1 -p 80 ubuntu:14.04.2 \
  sh -c 'cd / && python3 -m http.server 80'
e6b769ec3efa563a959ce771164de8337140d910de67e1df54d4960fdff74544
$ docker inspect -f '{{.NetworkSettings.Ports}}' py1
map[80/tcp:[map[HostIp:0.0.0.0 HostPort:49156]]]
$ curl -s localhost:49156 | tail
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
<li><a href="var/">var/</a></li>
</ul>
<hr>
</body>
</html>
```

HTTP服务器已经成功启动，这里使用了`inspect`命令的过滤选项将宿主机端口与内部容器映射信息提取出来。

现在确保`etcd`正在运行——本技巧假定工作环境与上一技巧相同。为简单起见，这一次将使用`etcdctl`（“`etcd controller`”的简称）与`etcd`（而不是直接对`etcd`进行`curl`）进行交互：

```
$ IMG=dockerinpractice/etcdctl
$ docker pull dockerinpractice/etcdctl
[...]
$ alias etcdctl="docker run --rm $IMG -C \"${HTTPIP}:8080\""
$ etcdctl set /test value
value
$ etcdctl ls
/test
```

这将下载我们准备好的etcdctl Docker镜像，同时设置一个别名用于连接前面设置的etcd集群。现在启动nginx：

```
$ IMG=dockerinpractice/confd-nginx
$ docker pull $IMG
[...]
$ docker run -d --name nginx -p 8000:80 $IMG $HTTPIP:8080
5a0b176586ef9e3514c5826f17d7f78ba8090537794cef06160ea7310728f7dc
```

这是一个我们提前准备好的镜像，它使用confd从etcd获取信息，并自动更新配置文件。传递的参数将告知容器它可以连接的etcd集群。不过这里尚未告诉它到哪去查找应用程序，因此日志中充满了错误！

下面将添加适当的信息到etcd中：

```
$ docker logs nginx
Using http://10.194.12.221:8080 as backend
2015-05-18T13:09:56Z fc6082e55a77 confd[14]:
➤ ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:06Z fc6082e55a77 confd[14]:
➤ ERROR 100: Key not found (/app) [14]
$ echo $HTTPIP
http://10.194.12.221
$ etcdctl set /app/upstream/py1 10.194.12.221:49156
10.194.12.221:49156
$ sleep 10
$ docker logs nginx
Using http://10.194.12.221:8080 as backend
2015-05-18T13:09:56Z fc6082e55a77 confd[14]:
➤ ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:06Z fc6082e55a77 confd[14]:
➤ ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:16Z fc6082e55a77 confd[14]:
➤ ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:26Z fc6082e55a77 confd[14]:
➤ INFO Target config /etc/nginx/conf.d/app.conf out of sync
2015-05-18T13:10:26Z fc6082e55a77 confd[14]:
➤ INFO Target config /etc/nginx/conf.d/app.conf has been updated
$ curl -s localhost:8000 | tail
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
```

```
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
<li><a href="var/">var/</a></li>
</ul>
<hr>
</body>
</html>
```

对etcd的更新已经被confd读取并应用到nginx配置文件中，让用户可以访问这个简单的文件服务器。这里包含sleep命令是因为配置了confd每10秒检查更新。在这背后，confd-nginx容器中运行着一个confd守护进程来拉取etcd集群中的变更，并只在检测到变更时使用容器内的模板重新生成nginx配置。

假设我们决定对外提供/etc目录而不是/目录。现在启动第二个应用程序并将其添加到etcd中。因为此时有两个后端，最终将得到其各自的响应：

```
$ docker run -d --name py2 -p 80 ubuntu:14.04.2 \
  sh -c 'cd /etc && python3 -m http.server 80'
9b5355b9b188427abaf367a51a88c1afa2186e6179ab46830715a20eacc33660
$ docker inspect -f '{{.NetworkSettings.Ports}}' py2
map[80/tcp:[map[HostIp:0.0.0.0 HostPort:49161]]]
$ curl $HTTPIP:49161 | tail | head -n 5
<li><a href="udev/">udev/</a></li>
<li><a href="update-motd.d/">update-motd.d/</a></li>
<li><a href="upstart-xsessions">upstart-xsessions</a></li>
<li><a href="vim/">vim/</a></li>
<li><a href="vtrgb">vtrgb@</a></li>
$ etcdctl set /app/upstream/py2 $HTTPIP:49161
10.194.12.221:49161
$ etcdctl ls /app/upstream
/app/upstream/py1
/app/upstream/py2
$ curl -s localhost:8000 | tail | head -n 5
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
$ curl -s localhost:8000 | tail | head -n 5
<li><a href="udev/">udev/</a></li>
```

```
<li><a href="update-motd.d/">update-motd.d</a></li>
<li><a href="upstart-xsessions">upstart-xsessions</a></li>
<li><a href="vim/">vim</a></li>
<li><a href="vtrgb">vtrgb@</a></li>
```

在上述过程中，在将新容器添加到etcd之前，会先确认它是否正确启动（见图7-4）。因此，可以通过覆盖etcd中的/app/upstream/py1键一步完成该过程。如果要求一次只能有一个后台可供访问，这个做法就很有用。

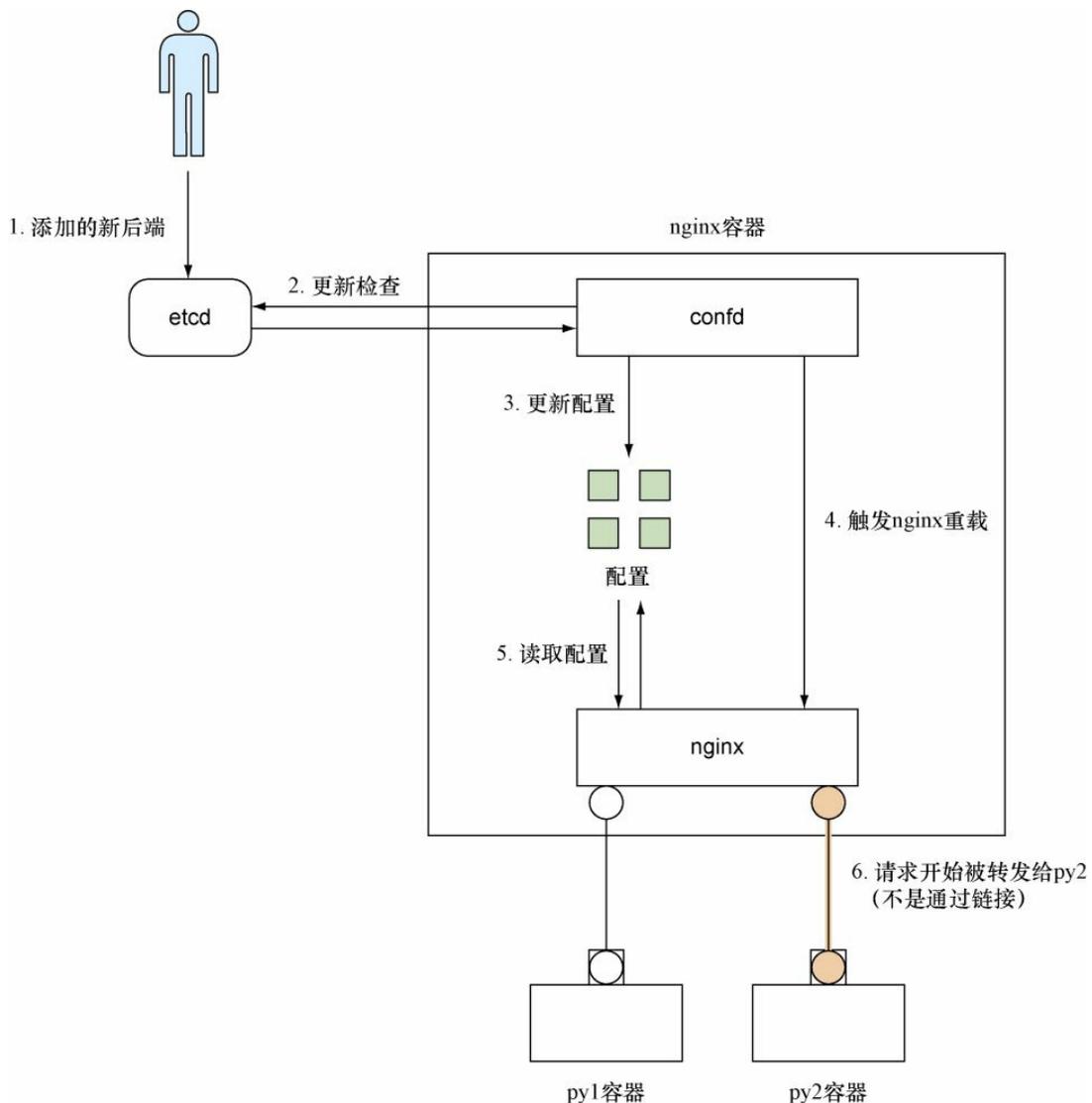


图7-4 添加py2容器到etcd中

使用两阶段切换，最后的阶段是删除旧的后端和容器：

```
$ etcdctl rm /app/upstream/py1

$ etcdctl ls /app/upstream
/app/upstream/py2
$ docker rm -f py1
py1
```

这样，新的应用程序就启动运行了！对用户而言，此应用程序没有不可访问的情况，并且不需要手工连接到Web服务器机器上去重载nginx。

confd的使用不仅限于配置Web服务器：只要某个文件包含需要根据外部值来更新的文本，confd便可派上用场。上一个技巧中提到了etcd不能用于存储长度很大的值。因此，并不一定要将confd与etcd搭配使用。多数流行的键值存储系统都有可用的集成方案，因此如果已经有可以正常运转的系统，则无须再添加另外的部分。

在生产环境中使用Docker时，如果需要为某个服务更新后端服务器，完全可以避免手工修改etcd，后面的技巧83就介绍了这样一种方法。

7.5 小结

使用Docker将引导用户走上流线型CD流水线的道路。不过，本章也探索了一些方法，使用额外的工具对Docker进行补充，解决Docker的一些局限性。

本章主要讲述：

- 在registry之间移动镜像可作为控制构建能在CD流水线中走多远的一个好办法；
- bup在压榨镜像传输方面非常在行，甚至比分层效果更佳；

- etcd可作为环境的中央配置存储；
- 零停机时间部署可通过etcd、confd以及nginx的组合来实现。

下一章将讲述在测试时通过模拟网络进一步加快CD流水线，而不需要一个完整的环境。

第8章 网络模拟：无痛的现实环境测试

本章主要内容

- Docker Compose初探
- 运行一个DNS服务器来完成基本的容器服务发现
- 在有问题的网络上测试应用程序
- 创建一个基底网络实现跨Docker宿主机的无缝通信

作为DevOps workflow的一部分，读者总会涉及某种形式的网络使用。不论是查找本地memcache容器的所在、连接到外部世界，还是将运行在不同宿主机上的Docker容器整合在一起，迟早都需要接触到广阔的网络。

本章将展示如何使用Docker的虚拟网络工具来模拟和管理网络，还将向编排（orchestration）和服务发现（service discovery）迈进一小步。编排和服务发现是第9章将深入探讨的话题。

8.1 容器通信——超越手工链接

技巧6中讲述过如何使用链接来连接容器，同时提到了明确的容器依赖声明所带来的优势。但是，链接具有一些不足之处。链接在启动每个容器时都要手工指定、容器必须以正确的顺序启动（以免容器链接循环）、链接无法进行替换（如果某个容器宕了，那么所有依赖于它的容器都需要重启以便重新创建链接）。

幸好有些工具可直击这些痛点。

技巧68 一个简单的Docker Compose集群

Docker Compose的前身名为fig，fig是一个现已弃用的独立项目，旨在减轻使用正确的链接、卷及端口参数来启动多个容器的痛苦。Docker 公司对其情有独钟，直接将其收购、重制，并使用新的名字进行了发布。

本技巧使用一个简单的Docker容器编排示例来介绍Docker Compose。

问题

想要让宿主机上链接的容器协同工作。

解决方案

使用Docker Compose。

讨论

Docker Compose是一个用于定义和运行复杂的Docker应用程序的工具。其核心思想是声明应用程序的启动配置，然后使用一条简单的命令来启动该应用程序，而无须使用复杂的shell脚本或Makefile来组装复杂的容器启动命令。

在编写本书时，还不建议在生产环境中使用Docker Compose。

安装 本书假定读者已安装好Docker Compose。在编写本书时其安装说明还在快速变化，因此请访问Docker的说明文档（<http://docs.docker.com/compose/install>）以获取最新信息。读者可能需要使用sudo来运行docker-compose。

为保持尽可能简单，本技巧将使用一个回响（echo）服务器和客户端。客户端每5 s发送一个常见的“Hello world!”消息给服务器，然后接收返回的信息。

源代码可用 本技巧的源代码可从<https://github.com/docker-in-practice/docker-compose-echo>获取。

下面的命令将创建一个工作目录用于创建服务器镜像：

```
$ mkdir server
$ cd server
```

使用代码清单8-1 所示代码创建服务器Dockerfile。

代码清单8-1 Dockerfile——简单的回响服务器

```
FROM debian
RUN apt-get update && apt-get install -y nmap (1)
CMD ncat -l 2000 -k --exec /bin/cat (2)
```

(1) 安装nmap包，它提供了这里所使用的ncat程序

(2) 在启动该镜像时默认运行ncat程序

参数-l 2000指示ncat监听端口2000，参数-k让它同时接受多个客户端连接，并在客户端关闭连接后继续运行，以便更多客户端可以接入。最后一个参数--exec /bin/cat是让ncat为所有接入的连接运行/bin/cat，把来自该连接的数据转发给该运行中的程序。

接下来，使用以下命令构建这个Dockerfile：

```
$ docker build -t server .
```

现在可以创建客户端镜像，用于发送消息给服务器。创建一个新目录，并将 `client.py` 文件及 `Dockerfile` 放置于此：

```
$ cd ..  
$ mkdir client  
$ cd client
```

代码清单8-2给出的是一个简单的Python程序，作为回响服务器的客户端。

代码清单8-2 `client.py`——一个简单的回响客户端

```
import socket, time, sys (1)  
while True:  
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) (2)  
    s.connect(('talkto', 2000)) (3)  
    s.send('Hello, world\n') (4)  
    data = s.recv(1024) (5)  
    print 'Received:', data (6)  
    sys.stdout.flush() (7)  
    s.close() (8)  
    time.sleep(5) (9)
```

- (1) 导入所需的Python包
- (2) 创建一个套接字对象
- (3) 使用该套接字连接'talkto'服务器的2000 端口
- (4) 发送一个带换行符的字符串到该套接字上
- (5) 创建一个1024字节的缓冲区用于接收数据，并在收到消息时将数据放置到data变量中
- (6) 将接收到的数据打印到标准输出中
- (7) 刷新标准输出缓冲区以便在消息进入时将其显示出来

(8) 关闭该套接字对象

(9) 等待5秒然后重复上述步骤

客户端的Dockerfile很简单。它安装Python，添加client.py文件，然后将其指定为启动时的默认运行项，如代码清单8-3所示。

代码清单8-3 Dockerfile——一个简单的回响客户端

```
FROM debian
RUN apt-get update && apt-get install -y python
ADD client.py /client.py
CMD ["/usr/bin/python", "/client.py"]
```

使用以下命令构建客户端：

```
docker build -t client .
```

为了展示docker-compose的价值，首先手动地运行这些容器：

```
docker run --name echo-server -d server
docker run --rm --name client --link echo-server:talkto client
```

命令执行完成后，按Ctrl+C退出客户端，并删除这些容器：

```
docker rm -f client echo-server
```

即便在这个简单的示例中，还是会有很多东西出错：先启动客户端将造成应用程序启动失败，忘记删除容器将在重启时造成问题，错误命名容器也将造成失败。当容器及其架构变得越来越复杂时，这类编排问题将随之增多。

Compose对此提供了解决之道，它把容器的启动和配置的编排封装在一个简单的文本文件中，并为用户管理启动与关闭命令的细节。

Compose需要一个YAML文件。请在一个新目录中创建该文件：

```
cd ..
mkdir docker-compose
cd docker-compose
```

YAML文件的内容如代码清单8-4所示。

代码清单8-4 docker-compose.yml——Docker Compose回响服务器与客户端
YAML文件

```
echo-server: (1)
  image: server (2)
  expose: (3)
  - "2000"

client:
  image: client
  links: (4)
  - echo-server:talkto
```

(1) 运行中的服务的引用名称是它们的标识：本示例中是echo-server和client

(2) 每个小节都必须定义所使用的镜像：本示例中是客户端与服务器镜像

(3) 将echo-server的2000端口暴露给其他服务

(4) 定义一个指向echo-server的链接。客户端内对talkto的引用将被发送给回响服务器。其映射是通过在运行的容器中动态设置/etc/hosts文件完成的

docker-compose.yml的语法非常容易理解：最左侧是命名的服务，其配置声明在下方缩进的小节中。每个配置项名称后面都有一个冒号，这些项目的属性要么声明在同一行，要么声明在后续以相同缩

进层次破折号开始的几行中。

这里需要理解的关键配置项是客户端定义中的**links**。这些链接的创建方式与**docker run**命令创建链接的方式一致。实际上，大部分Docker命令行参数在docker-compose.yml语法中都有直接的对应关系。

这个示例中使用**image:**语句来定义每个服务所使用的镜像，不过也可以在**build:**语句中定义Dockerfile的路径，让docker-compose动态地重新构建所需镜像。Docker Compose会自动进行构建。

什么是**YAML**文件？YAML文件是使用简单语法的一个文本配置文件。更多信息可从其官网<http://yaml.org>获得。

现在所有的基础设施都创建好了，运行该应用程序很简单：

```
$ docker-compose up
Creating dockercompose_server_1...
Creating dockercompose_client_1...
Attaching to dockercompose_server_1, dockercompose_client_1
client_1 | Received: Hello, world
client_1 |
client_1 | Received: Hello, world
client_1 |
```

出现无法连接错误？如果在启动docker-compose时出现类似“Couldn't connect to Docker daemon at http+unix://var/run/docker.sock--is it running?”这样的错误，其问题可能是需要使用**sudo**运行。

确认结果无误后，按多次**Ctrl+C**退出应用程序。使用相同的命令

即可重新运行该应用程序，而无须考虑删除容器的问题。需要注意的是，在重新运行时输出的是“Recreating”（重新创建）而不是“Creating”（创建）。

我们已经对Docker Compose有所了解，接下来讨论一个更复杂的docker-compose现实场景：使用socat、卷及链接为运行在宿主机上的一个SQLite实例添加类似服务器的功能。

技巧69 一个使用Docker Compose的SQLite服务器

默认情况下，SQLite没有任何TCP服务器的概念。本技巧在上一个技巧的基础上，提供了一种使用Docker Compose来实现TCP服务器功能的方法。

具体说来，它是使用此前介绍过的工具和概念来构建的：

- 卷；
- 使用socat代理；
- 容器链接；
- Docker Compose。

要求使用**SQLite 3**版本 本技巧要求在宿主机上安装SQLite 3版本。同时建议安装rlwrap，以便在与SQLite服务器交互时让行编辑变得更友好一些（虽然这不是必需的）。这些软件包在标准的包管理器中都能免费获得。

本技巧对应的代码可在<https://github.com/docker-in-practice/docker-compose-sqlite>下载。

如果在使用本技巧时出现问题，则说明Docker版本可能需要升

级。1.7.0之后的版本应该都可以正常工作。

问题

想要使用Docker高效地开发一个复杂的引用宿主机外部数据的应用程序。

解决方案

使用Docker Compose。

讨论

图8-1给出了本技巧架构的一个概述。从高层次上看，有两个运行中的Docker容器，一个负责执行SQLite客户端，另一个用于将不同的TCP连接代理到这些客户端上。需要注意的是，执行SQLite的容器并未暴露给宿主机，代理容器则实现了这一点。将职责分离成离散单元是微服务构架的一个共同特点。

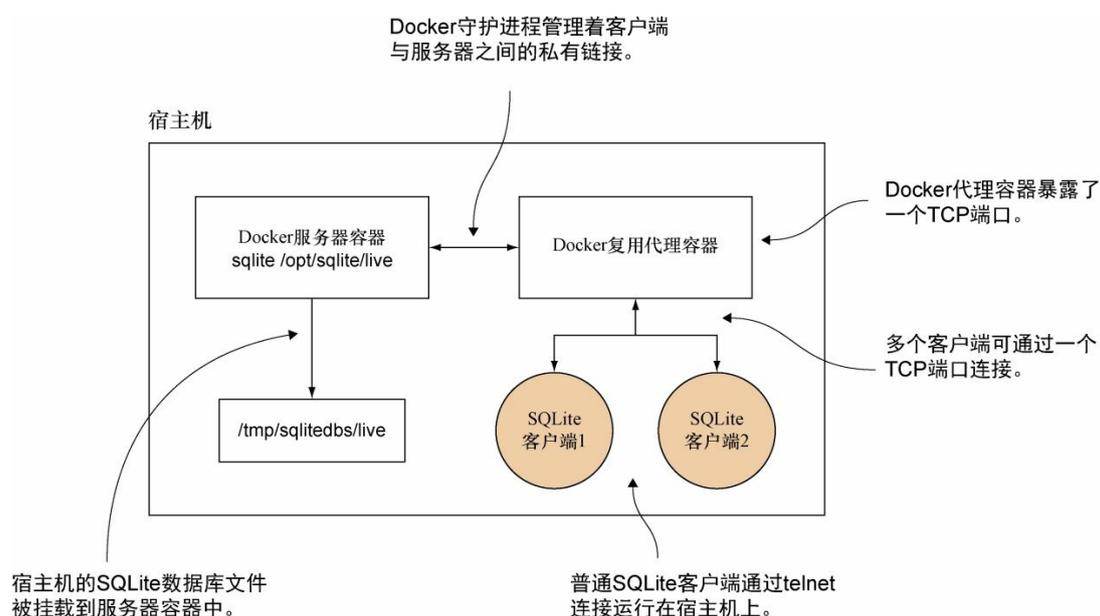


图8-1 SQLite服务器工作原理

所有节点都将使用同一个镜像。设置代码清单8-5所示的

Dockerfile。

代码清单8-5 SQLite服务器、客户端及代理合一的Dockerfile

```
FROM ubuntu:14.04
RUN apt-get update && apt-get -y install rlwrap sqlite3 socat (1)
EXPOSE 12345 (2)
```

(1) 安装必要的应用程序

(2) 暴露12345端口以便节点可通过Docker守护进程进行通信

代码清单8-6展示的是docker-compose.yml的内容，它定义了容器将如何启动。

代码清单8-6 SQLite服务器与代理的docker-compose.yml

```
server: (1)
  command: socat TCP-L:12345,fork,reuseaddr
  ➤ EXEC:'sqlite3 /opt/sqlite/db',pty (2)
  build: . (3)
  volumes: (4)
  - /tmp/sqlitedbs/test:/opt/sqlite/db
proxy:
  command: socat TCP-L:12346,fork,reuseaddr TCP:sqliteserver:12345 (5)
  build: .
  links: (6)
  - server:sqliteserver
  ports: (7)
  - 12346:12346
```

(1) 服务器与代理容器定义在这一节中

(2) 创建一个socat代理，用于将SQLite调用的输出链接到一个TCP端口上

(3) 在启动时从当前目录的Dockerfile构建镜像

(4) 将SQLite测试数据库文件挂载到容器内的/opt/ sqlite/db上

(5) 创建一个socat代理，用于把12346端口的数据传递到服务器容器的12345端口上

(6) 定义一个代理与服务器之间的链接，将此容器内对sqliteserver的引用映射到服务器容器中

(7) 向宿主机发布12346端口

参数TCP-L:12345, fork, reuseaddr指定服务器容器中的socat进程监听12345端口，并允许接入多个连接。后面的EXEC:部分告诉socat针对每个连接在/opt/sqlite/db文件上运行SQLite，并为此进程分配一个伪终端。客户端容器中的socat进程与服务器容器的监听行为一样（除了端口不同），不过它将建立一个与SQLite服务器的TCP连接，而不是为呼入的连接运行某个程序。

尽管这项功能可以在一个容器内实现，但服务器/代理容器的设置可以让这个系统的架构更易于扩展，因为每个容器只对一项任务负责：服务器负责打开SQLite连接，而代理负责将服务暴露给宿主机。

代码清单8-7（从仓库的原始版本简化而来，见<https://github.com/docker-in-practice/docker-compose-sqlite>）将在宿主机上创建两个最小化的SQLite数据库，即test和live。

代码清单8-7 setup_dbs.sh

```
#!/bin/bash
echo "Creating directory"
SQLITEDIR=/tmp/sqlitedbs
rm -rf $SQLITEDIR (1)
if [ -a $SQLITEDIR ] (2)
then
    echo "Failed to remove $SQLITEDIR"
    exit 1
fi
```

```
mkdir -p $SQLITEDIR
cd $SQLITEDIR
echo "Creating DBs"
echo 'create table t1(c1 text);' | sqlite3 test (3)
echo 'create table t1(c1 text);' | sqlite3 live (4)
echo "Inserting data"
echo 'insert into t1 values ("test");' | sqlite3 test (5)
echo 'insert into t1 values ("live");' | sqlite3 live (6)
cd - > /dev/null 2>&1 (7)
echo "All done OK"
```

- (1) 删除上一次运行的所有目录
- (2) 如果目录依然存在则抛出一个错误
- (3) 创建test数据库以及一张表
- (4) 创建live数据库以及一张表
- (5) 插入一行"test"字符串到表中
- (6) 插入一行"live"字符串到表中
- (7) 返回到此前的目录

要运行这个示例，可如代码清单8-8所示设置数据库并用 `docker-compose up`。

代码清单8-8 启动Docker Compose集群

```
$ chmod +x setup_dbs.sh
$ ./setup_dbs.sh
$ sudo docker-compose up
Creating dockercomposesqlite_server_1...
Creating dockercomposesqlite_proxy_1...
Attaching to dockercomposesqlite_server_1, dockercomposesqlite_proxy_1
```

接着，在一个或多个其他终端中，可以针对一个SQLite数据库运行Telnet来创建多个会话，如代码清单8-9所示。

```

$ rlwrap telnet localhost 12346 (1)
Trying 127.0.0.1... (2)
Connected to localhost.
Escape character is '^]'.
SQLite version 3.7.17 (3)
Enter ".help" for instructions
sqlite> select * from t1; (4)
select * from t1;
test
sqlite>

```

(1) 使用Telnet连接代理，将其包装在rlwrap里，实现命令行的编辑和历史功能

(2) Telnet连接的输出

(3) SQLite连接在此时建立

(4) 在sqlite提示符下运行SQL命令

现在如果要切换到live服务器，通过修改docker-compose.yml的volumes这一行来修改配置，从

```
- /tmp/sqlitedbs/test:/opt/sqlite/db
```

变为

```
- /tmp/sqlitedbs/live:/opt/sqlite/db
```

然后运行这个命令：

```
$ sudo docker-compose up
```

不适用于生产环境 尽管我们对这个SQLite客户端的多路复用做了

一些基本的测试，但我们不对任何类型负载下该服务器的数据完整性及性能做保证。

本技巧演示了 Docker Compose 如何能把相对棘手和复杂的事务变得健壮且简单。这里以SQLite为例，通过连接容器，将SQLite调用代理到宿主机的数据上，从而赋予它类似服务器的功能。使用Docker Compose的YAML配置，使容器复杂度的管理变得十分简单，它把编排容器的复杂事务从手工且易出错的过程变成了可通过源代码控制的更安全和自动化的过程。这是编排之路的开端，第9章将对此做更多介绍。

技巧70 使用Resolvable通过DNS查找容器

容器启动时默认会被分配到自己的IP地址，并且只有在知道对方的IP时才能与之通信——链接（不论是手工建立还是由Docker Compose提供）是将IP分布到/etc/hosts及环境变量中的一种方法。

不过，这种分布方式存在限制——运行中的容器的环境变量无法更新，并且试图动态更新/etc/hosts可能会带来问题。为避免出现这两个问题，Docker不允许为运行中的容器添加链接。

解决分布IP地址的问题有一个很好的方案就是使用，每天都会用到的DNS服务器！

问题

想要无须使用链接即可让容器相互发现。

解决方案

使用Resolvable作为DNS服务器。

讨论

Resolvable (<https://github.com/gliderlabs/resolvable>) 是一个工具，它会读取宿主机上当前运行的容器的信息，并以标准方式提供名称到IP地址的映射服务——它就是一个DNS服务器。

内置到**Docker**中 Docker的某些版本（1.7.1之后，1.9.0之前）提供了一项功能，无须任何外部工具，容器即可使用名称与其他容器通信。不过如果想从宿主机上查找容器，Resolvable可能还是有价值的，有如本技巧末尾所述。

在开始之前，需要先确定一些设置——docker0网络接口的地址以及启动容器时当下使用的DNS服务器：

```
$ ip addr | grep 'inet.*docker0'
    inet 172.17.42.1/16 scope global docker0
$ docker run --rm ubuntu:14.04.2 cat /etc/resolv.conf | grep nameserver
nameserver 8.8.8.8
nameserver 8.8.4.4
```

上述两项是Docker的默认值。如果调整过Docker守护进程的设置，或电脑上有些不同寻常的配置，则结果可能会略有不同。

现在即可使用刚查找出的恰当值来启动Resolvable容器：

```
$ DNSARGS="--dns 8.8.8.8 --dns 8.8.4.4"
$ PORTARGS="-p 172.17.42.1:53:53/udp"
$ VOLARGS="-v /var/run/docker.sock:/tmp/docker.sock"
$ docker run -d --name res -h resolvable $DNSARGS $PORTARGS $VOLARGS \
gliderlabs/resolvable:master
5ebbe218b297da6390b8f05c0217613e47f46fe46c04be919e415a5a1763fb11
```

容器的启动过程中提供了以下3个关键信息。

- Resolvable需要知道在请求的地址无法映射到容器时向谁查询。

答案是向上游DNS服务器查询。Resolvable会从容器内的/etc/resolv.conf获取这些值，--dns参数用于填充/etc/resolv.conf。虽然此处并不是严格必需的（指定的值是默认值），但在后面就能看到这还是相当有用的。

- 监听DNS请求的接口。使用Docker网桥的IP地址的好处在于无须将服务器暴露给外界（用0.0.0.0则会这样），且使用的是不变的IP地址（与使用容器IP地址不同）。
- Resolvable需要Docker套接字以便能查找容器名称。

对于每个运行的容器，Resolvable会创建两个名称，即<容器名称>.docker和<容器主机名>。可使用dig命令通过查找Resolvable容器自身对此进行测试（dig命令存在于Ubuntu的dnsutils包或CentOS的bind-utils中）：

```
$ dig +short @172.17.42.1 res.docker
172.17.0.22
$ dig +short @172.17.42.1 resolvable
172.17.0.22
```

这很有趣，但并不是十分有用——类似的信息使用docker inspect也可获得。不过，在启动配置使用这个新的DNS服务器的容器时，它的价值就体现出来了：

```
$ docker run -it --dns 172.17.42.1 ubuntu:14.04.2 bash
root@216a71584c9c:/# ping -q -c1 res.docker
PING res.docker (172.17.0.22) 56(84) bytes of data.

--- res.docker ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.065/0.065/0.065/0.000 ms
root@216a71584c9c:/# ping -q -c1 www.google.com
PING www.google.com (216.58.210.36) 56(84) bytes of data.

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.991/7.991/7.991/0.000 ms
```

此处证实了使用由Resolvable提供的DNS服务器可以同时访问其他容器和外界。但是，每次启动容器时都要指定--dns参数有些麻烦。好在，可以通过给Docker守护进程传递相关参数来节省时间。编辑守护进程参数（使用适合当前操作系统的方法），添加以下内容：

```
--bip=172.17.42.1/16 --dns=172.17.42.1
```

设置Docker守护进程参数 Docker守护进程的具体参数帮助详见附录B。

这些值来自本技巧开始时运行的用于查找Docker网桥细节的命令。读者需要根据自己的结果进行适当修改。--dns参数对守护进程的作用相当直观——它修改了容器使用的默认DNS服务器。与此同时，--bip将Docker网桥的配置固定了下来，这样在守护进程重启时它就不会发生潜在的变化（这会让所有容器DNS失效）。

在启动Resolvable时传递的DNS参数在这里很关键——如果未对其进行指定，Resolvable在向上游查询时将使用默认的DNS服务器，也就是指向它自身！如果这种情况发生，将产生巨量的日志，而客户端的查询将会超时。

一旦Docker守护进程完成重启，且Resolvable启动后，就可尝试启动一个容器：

```
$ docker run --rm ubuntu:14.04.2 ping -q -c1 resolvable
PING resolvable (172.17.0.1) 56(84) bytes of data.

--- resolvable ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.095/0.095/0.095/0.000 ms
```

`Resolvable`具有一些决定性功能，虽然这里不对其进行详述，但是为了保持完整性稍作提及。如果将`/etc/resolv.conf`挂载到`/tmp/resolv.conf`上，该DNS服务器地址将被添加到宿主机的DNS服务器中，这样就可以在容器之外通过容器名称对其进行访问。使用`systemd`的用户可以通过将`/run/systemd`挂载到`/tmp/systemd`，并将`/var/run/dbus/system_bus_socket`挂载到容器中相同路径来实现类似的整合。

在使用多宿主机时，如何能方便地查找和连接到其他容器是一个热门话题。本书后续章节将对服务发现做更多详细论述。

8.2 使用Docker来模拟真实世界的网络

多数用户会将互联网当作一个黑盒看待，即通过某些方式从世界其他角落获取信息并显示在屏幕上。时不时会碰到网速慢或连接中断的情况，对ISP的抱怨屡见不鲜。

在构建的镜像包含需要进行连接的应用程序时，用户可能对哪些组件需要连接到哪里以及整体的设置情况有一个清晰的了解。但有一件事是不变的：还是会遭遇网速慢和连接中断的情况。即使是拥有并运营着自有数据中心的大型公司，也能观察到不可靠的网络及其引起的应用程序问题。

接下来介绍几种方法，对不可靠网络进行体验，以确定现实世界中可能面对的问题。

技巧71 使用Comcast模拟有问题的网络

尽管用户在进行跨多主机分发应用程序时希望网络状况尽可能

好，但现实却是残酷的——分组（packet，也称数据包）丢失（也称丢包）、连接中断、网络分区比比皆是，尤其是在商用云服务供应商上。

在技术栈遭遇现实世界的这些情况之前对其进行测试以确认其行为是非常明智的——一个为高可用设计的应用程序不应在外部服务开始出现显著的额外延迟时陷入停顿。

问题

想要为单个容器应用不同的网络状况。

解决方案

使用Comcast（指的是网络工具，而非ISP）。

讨论

Comcast（<https://github.com/tylertreat/Comcast>）是一个娱乐化命名的工具，用于修改Linux机器的网络接口，以便对其应用某些不同寻常的（或者，对不走运的人而言是典型的）状况。

在Docker创建容器时，它会同时创建几个虚拟的网络接口——这也是容器具有不同IP并且可以相互通信的原因。因为这些都是标准网络接口，只要能查找出其网络接口名称，就可以在其上使用Comcast。这说起来容易做起来难。

以下Docker镜像包含了Comcast及其前置要求，以及一些优化：

```
$ IMG=dockerinpractice/comcast
$ docker pull $IMG
latest: Pulling from dockerinpractice/comcast
[...]
Status: Downloaded newer image for dockerinpractice/comcast:latest
$ alias comcast="docker run --rm --pid=host --privileged \
```

```

-v /var/run/docker.sock:/var/run/docker.sock $IMG"
$ comcast -help
Usage of comcast:
  -cont="": Container ID or name to get virtual interface of
  -default-bw=-1: Default bandwidth limit in kbit/s (fast-lane)
  -device="": Interface (device) to use (defaults to eth0 where applica
ble)
  -dry-run=false: Specifies whether or not to commit the rule changes
  -latency=-1: Latency to add in ms
  -mode="start": Start or stop packet controls
  -packet-loss="0": Packet loss percentage (eg: 0.1%)
  -target-addr="": Target addresses, \
(eg: 10.0.0.1 or 10.0.0.0/24 or 10.0.0.1,192.168.0.0/24)
  -target-bw=-1: Target bandwidth limit in kbit/s (slow-lane)
  -target-port="": Target port(s) (eg: 80 or 1:65535 or 22,80,443,1000:
1010)
  -target-proto="tcp,udp,icmp": \
Target protocol TCP/UDP (eg: tcp or tcp,udp or icmp)

```

新增的优化提供了 `-cont` 选项，可以指向一个容器而无须查找虚拟网络接口的名称。请注意，为了赋予容器更多权限，`docker run` 命令中增加了一些特殊的标志，这样Comcast就可以自由地对网络接口进行检查并应用变更。

为了展示Comcast可以带来的变化，先来看一下一个正常的网络连接是什么样的。打开一个新的终端，并运行以下命令来设置基准网络性能的预期：

```

$ docker run -it --name c1 ubuntu:14.04.2 bash
root@0749a2e74a68:/# apt-get update && apt-get install -y wget
[...]
root@0749a2e74a68:/# ping -q -c 5 www.docker.com
PING www.docker.com (104.239.220.248) 56(84) bytes of data.

--- www.docker.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms (1)

rtt min/avg/max/mdev = 98.546/101.272/106.424/2.880 ms (2)

root@0749a2e74a68:/# time wget -o /dev/null https://www.docker.com

real    0m0.680s (3)
user    0m0.012s
sys     0m0.006s

```

```
root@0749a2e74a68: /#
```

(1) 这台机器与www.docker.com的连接看起来是可靠的，没有分组丢失

(2) 到www.docker.com的往返时间在100 ms左右

(3) 下载www.docker.com的HTML首页总共花费时间大概是0.7

s

完成上述步骤后，保持该容器处于运行状态，然后对其应用一些网络状况：

```
$ comcast -cont c1 -default-bw 50 -latency 100 -packet-loss 20%
2015/07/29 02:28:13 Found interface vetha7b90a7 for container 'c1'
sudo tc qdisc show | grep "netem"
sudo tc qdisc add dev vetha7b90a7 handle 10: root htb
sudo tc class add dev vetha7b90a7 parent 10: classid 10:1 htb rate 50kb
it
sudo tc class add dev vetha7b90a7 parent 10:1 classid 10:10 htb rate 50
kbit
sudo tc qdisc add dev vetha7b90a7 parent 10:10 handle 100:
└─ netem delay 100ms loss 20.00%
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p
tcp
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p
udp
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p
icmp
2015/07/29 02:28:13 Packet rules setup...
2015/07/29 02:28:13 Run `sudo tc -s qdisc` to double check
2015/07/29 02:28:13 Run `comcast --mode stop` to reset
```

上述命令应用了3种不同的状况：针对所有目标设置50 KB/s的带宽上限（唤起了对拨号的回忆），添加100 ms的延迟，以及20%的分组丢失率。

Comcast首先确定容器正确的虚拟网络接口，然后调用一些标准的Linux命令行网络工具来应用流量规则，并在执行过程中列出其所

做的动作。来看一下容器是如何对此进行回应的：

```
root@0749a2e74a68:/# ping -q -c 5 www.docker.com
PING www.docker.com (104.239.220.248) 56(84) bytes of data.

--- www.docker.com ping statistics ---
5 packets transmitted, 2 received, 60% packet loss, time 4001ms
rtt min/avg/max/mdev = 200.453/200.581/200.709/0.128 ms
root@0749a2e74a68:/# time wget -o /dev/null https://www.docker.com

real    0m9.673s
user    0m0.011s
sys     0m0.011s
```

成功了！ping报告的延迟增加了100 ms，而对wget的计时展示了10倍左右的降速，与预期相当（带宽上限、额外的延迟及分组丢失同时产生了影响）。但是分组丢失有点儿奇怪——似乎比预期大了3倍。需要注意的很重要的一点是，ping只发送了少量的分组，而分组丢失不是精确的“五分之一”计数器——如果将ping次数提高到50，将会发现分组丢失结果与预期要接近得多。

注意，上面应用的规则对通过该网络接口的所有网络连接都有效，包括与宿主机及其他容器的连接。

现在告诉Comcast删除这些规则。Comcast还无法对单个状况进行添加或删除，因此修改某张网络接口上的任何东西意味着要完全删除或重新添加该网络接口上的规则。如果要恢复正常的容器网络操作，也必须删除这些规则。不过，在退出容器时无须考虑这些规则的删除——它们会在Docker删除虚拟网络接口时被自动删除：

```
$ comcast -cont c1 -mode stop
2015/07/29 02:31:34 Found interface vetha7b90a7 for container 'c1'
[...]
2015/07/29 02:31:34 Packet rules stopped...
2015/07/29 02:31:34 Run `sudo tc -s qdisc` to double check
2015/07/29 02:31:34 Run `comcast --mode start` to start
```

如果读者有兴趣动手实践，可以深入挖掘Linux的流量控制工具，使用Comcast来生成要使用的命令示例集合。其可能性的完整性论述已经超出本技巧的范围，不过请记住，只要能将其放到容器内并连接到网络，就能使用它来做试验。

技巧72 使用Blockade模拟有问题的网络

对很多应用程序而言，Comcast是一个优秀的工具，不过有一个重要的使用场景它无法解决——如何将网络状况应用到全体容器中？手工对几十个容器运行Comcast将是非常痛苦的，上百个更是无法想象！这个问题对容器而言尤为相关，因为它们启动的代价非常低——如果在单台机器上运行上百个虚拟机而不是容器来模拟大型网络，将会遇到更大的问题，如内存不足！

在使用多台机器模拟网络时，有一种特殊类型的网络故障会在这种规模下变得有趣起来——网络分区。当一组网络化的机器被分成两个或更多部分时，这种情况就会出现，同一部分里的所有机器可以相互通信，但不同部分则无法通信。研究表明这种情况的发生比想象中要多得多，尤其是在消费级的云服务上！

遵循经典的Docker微服务线路可急剧缓解此类问题，而要理解服务如何对其进行处理，拥有用于做实验的工具就至关重要了。

问题

想要对大量容器进行网络状况编排设置，包括创建网络分区。

解决方案

使用Blockade。

讨论

Blockade (<https://github.com/dcm-oss/blockade.git>) 是出自戴尔公司一个部门的开源软件，为“测试网络故障及分区”而生。看起来正是这里所需要的。

注意 在编写本书时，如果不做优化，官方的Blockade仓库无法与Docker 1.6.2之后的任何版本兼容。为了本技巧的需要，我们对其做了一些修改，希望可以贡献回去。如果读者想深入Blockade，请记住这一点。

Blockade通过读取当前目录中的配置文件（`blockade.yml`）来工作，该配置文件定义了容器启动的方式以及需要对其应用的状况。完整的配置细节可从Blockade文档中获得，因此这里只对核心部分进行说明。

首先需要创建一个`blockade.yml`：

```
containers:
  server:
    image: ubuntu:14.04.2
    command: /bin/sleep infinity
    expose: [10000]

  client1:
    image: ubuntu:14.04.2
    command: sh -c "ping $SERVER_PORT_10000_TCP_ADDR"
    links: ["server"]

  client2:
    image: ubuntu:14.04.2
    command: sh -c "ping $SERVER_PORT_10000_TCP_ADDR"
    links: ["server"]

network:
  flaky: 50%
  slow: 100ms
```

上述配置中设置的容器代表由两个客户端连接的一个服务器。在

实践中，可能是一个数据库服务器及其客户端应用程序，并且对要建模的组件数量没有任何限制。只要它能在一个`compose.yml`文件（见技巧68）中表示，就可以在Blockade中对其进行建模。

在`server`的配置中指定了要暴露的一个端口，但并未在其上提供服务也不会对其进行连接——这只是启用Docker的链接功能，并在客户端容器中暴露相关的环境变量以便让它们知道去ping哪个IP。如果使用了其他IP发现技巧，如本章讲述的DNS技巧（见技巧70），则这些链接就不是必需的。

这里暂时不用考虑`network`小节，稍后会对其进行说明。

与往常一样，使用Blockade的第一步是拉取镜像：

```
$ IMG=dockerinpractice/blockade
$ docker pull $IMG
latest: Pulling from dockerinpractice/blockade
[...]
Status: Downloaded newer image for dockerinpractice/blockade:latest
$ alias blockade="docker run --rm --pid=host --privileged \
-v \ $PWD:/blockade -v /var/run/docker.sock:/var/run/docker.sock $IMG"
```

可以看到，这里传递给`docker run`的参数与上一个技巧中的参数一致，只有一个例外——Blockade将当前目录挂载到容器中以便访问`blockade.yml`，并将状态存储在一个隐藏目录中。

网络化文件系统之痛 如果是运行在网络文件系统之上，第一次启动Blockade可能会遇到奇怪的权限问题，这可能是由于Docker正在尝试以`root`身份创建该隐藏的状态目录，而网络文件系统不予配合。解决方案是使用本地磁盘。

最后到了关键时刻——运行Blockade。要确保目前位于保存

blockade.yml的目录中：

```
$ blockade up
NODE      CONTAINER ID  STATUS  IP           NETWORK  PARTITION
client1   8c4d956cf9cf  UP      172.17.0.53  NORMAL
client2   fcd9af2b0eef  UP      172.17.0.54  NORMAL
server    b8f9f179a10d  UP      172.17.0.52  NORMAL
```

调试提示 在启动时，Blockade有时可能会报“/proc”中文件不存在的晦涩错误。首先检查容器是否在启动时马上退出，阻止了Blockade检查其网络状态。此外，请尽量不要使用Blockade的-c选项来指定自定义配置文件目录——容器内只有当前目录的子目录可用。

所有配置文件中定义的容器已经启动，并显示了已启动容器的一些有用信息。现在来应用一些基本的网络状况。在一个新的终端中持续打印client1的日志（使用docker logs -f 8c4d956cf9cf），以便在做修改时可以查看所发生的情况：

```
$ blockade flaky -all (1)
$ sleep 5 (2)
$ blockade slow client1 (3)
$ blockade status (4)
NODE      CONTAINER ID  STATUS  IP           NETWORK  PARTITION
client1   8c4d956cf9cf  UP      172.17.0.53  SLOW
client2   fcd9af2b0eef  UP      172.17.0.54  FLAKY
server    b8f9f179a10d  UP      172.17.0.52  FLAKY
$ blockade fast -all (5)
```

(1) 让所有容器的网络变得不稳定（分组丢失）

(2) 延后下一条命令，让前一条命令有时间生效并输出一些日志

(3) 让容器client1的网络变慢（为分组增加了延迟）

(4) 检查容器所处的状态

(5) 将所有容器恢复为正常操作

`flaky`和`slow`命令使用了之前配置文件中`network`一节定义的值——限定值无法在命令行中指定。如果有需要，可以在容器运行时编辑`blockade.yml`，然后有选择性地将新的限定值应用到容器上。需要注意的是，一个容器只能处在慢速网络或不稳定网络中，不能二者皆有。撇开这些限制，对成百上千个容器运行这一命令的便捷性还是相当可观的。

如果回头查看`client1`的日志，可以看到不同命令生效的时间：

```
64 bytes from 172.17.0.52: icmp_seq=638 ttl=64 time=0.054 ms (1)
64 bytes from 172.17.0.52: icmp_seq=639 ttl=64 time=0.098 ms
64 bytes from 172.17.0.52: icmp_seq=640 ttl=64 time=0.112 ms
64 bytes from 172.17.0.52: icmp_seq=645 ttl=64 time=0.112 ms (2)
64 bytes from 172.17.0.52: icmp_seq=652 ttl=64 time=0.113 ms
64 bytes from 172.17.0.52: icmp_seq=654 ttl=64 time=0.115 ms
64 bytes from 172.17.0.52: icmp_seq=660 ttl=64 time=100 ms (3)
64 bytes from 172.17.0.52: icmp_seq=661 ttl=64 time=100 ms
64 bytes from 172.17.0.52: icmp_seq=662 ttl=64 time=100 ms
64 bytes from 172.17.0.52: icmp_seq=663 ttl=64 time=100 ms
```

(1) `icmp_seq`是连续的（没有分组丢失），`time`也比较低（延迟小）

(2) `icmp_seq`出现了一个大跳跃——`flaky`命令生效了

(3) `time`出现了一个大跳跃——`slow`命令生效了

虽然这很有用，不过在Comcast之上使用一些（可能是比较费力的）脚本也能实现，那么来看看Blockade的杀手锏功能——网络分区：

```

$ blockade partition server client1,client2
$ blockade status
NODE          CONTAINER ID  STATUS  IP           NETWORK  PARTITION
client1       8c4d956cf9cf  UP      172.17.0.53  NORMAL   2
client2       fcd9af2b0eef  UP      172.17.0.54  NORMAL   2
server        b8f9f179a10d  UP      172.17.0.52  NORMAL   1

```

这会将3个节点划分成2个区域——服务器在其中一个区域，而客户端在另一个区域，它们之间无法进行通信。可以看到**client1**的日志停止了，因为所有的**ping**分组都丢失了！不过，两个客户端依然可以相互通信，这一点可以通过在二者之间发送一些**ping**分组来验证：

```

$ docker exec 8c4d956cf9cf ping -qc 3 172.17.0.54
PING 172.17.0.54 (172.17.0.54) 56(84) bytes of data.

--- 172.17.0.54 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.065/0.084/0.095/0.015 ms

```

没有分组丢失，延迟也很低……看起来是个不错的连接。分区及其他网络状况是独立操作的，因此可以在应用分区的同时试验分组丢失。可以定义的分区数量没有限制，因此可以尽情地复杂场景进行试验。

最后一个建议是，将Blockade和Comcast组合起来能获得比它们单独提供的能力更强大。Blockade擅长创建分区以及完成启动容器的繁杂事务，添加Comcast则能实现每一个容器网络连接的细粒度控制！

8.3 Docker和虚拟网络

Docker的核心功能都与隔离性有关。前面几章已经展示了进程和文件系统隔离的好处，而本章呈现的是网络隔离。

可以认为网络隔离具有以下两个方面：

- 个体沙箱——每个容器具有各自用于监听的IP及端口集合，不会与其他容器（或宿主机）发生重叠；
- 组沙箱——这是个体沙箱的逻辑扩展，所有隔离的容器都被分组在同一个私有网络中，可以在不干扰主机网络（惹恼公司网络管理员）的情况下进行试验。

前面的两个技巧提供了这两个方面的一些实例——Comcast操纵单独的沙箱来为每个容器应用规则，而Blockade中的分区依赖对私有容器网络的全面监管能力来将其拆分成几部分。

这些场景的背后看起来有点儿像图8-2。

网桥工作的具体细节并不重要。简单来说，网桥在容器之间创建了一个扁平化网络（无须中间步骤即可直接通信），并将对外界的请求转发到外部连接上。

虚拟网络产生的灵活性极大地激发了第三方通过多种方式扩展网络系统以实现更复杂的用例。Docker 公司使用这些成果来引导当下（编写本书时）的工作，允许直接将网络扩展插入到Docker中而不是绕过它。

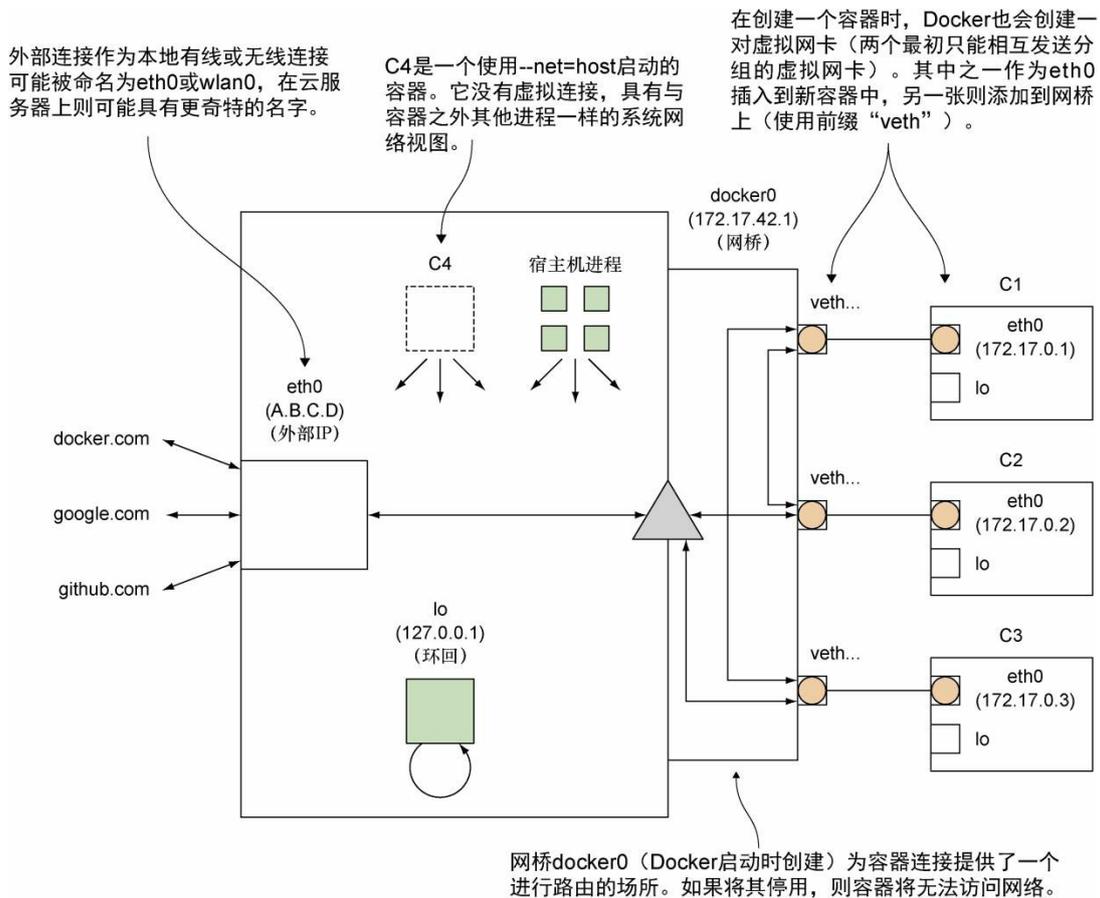


图8-2 宿主机上的Docker内部网络

技巧73 使用Weave建立一个基底网络

基底网络是构建于其他网络之上的软件级网络。从效果上看，它像是一个本地网络，但其背后还是通过其他网络进行通信。这代表着性能代价，这个网络相比本地网络稳定性要差一些，不过从可用性角度来看还是大有好处的：位置完全不同的节点可以像在同一个空间里一样进行通信。

实现这一点对Docker容器来说尤为有趣——容器可以像通过网络连接宿主机一样跨宿主机无缝地进行连接。这么做消除了规划单个宿主主机上能容纳多少容器的迫切需要。

问题

想要跨宿主机进行容器间无缝通信。

解决方案

使用基底网络。

讨论

接下来将使用Weave (<http://weave.works>) 来演示基底网络的原理，Weave是为这个目的设计的工具。图8-3展示了一个典型Weave网络的梗概。

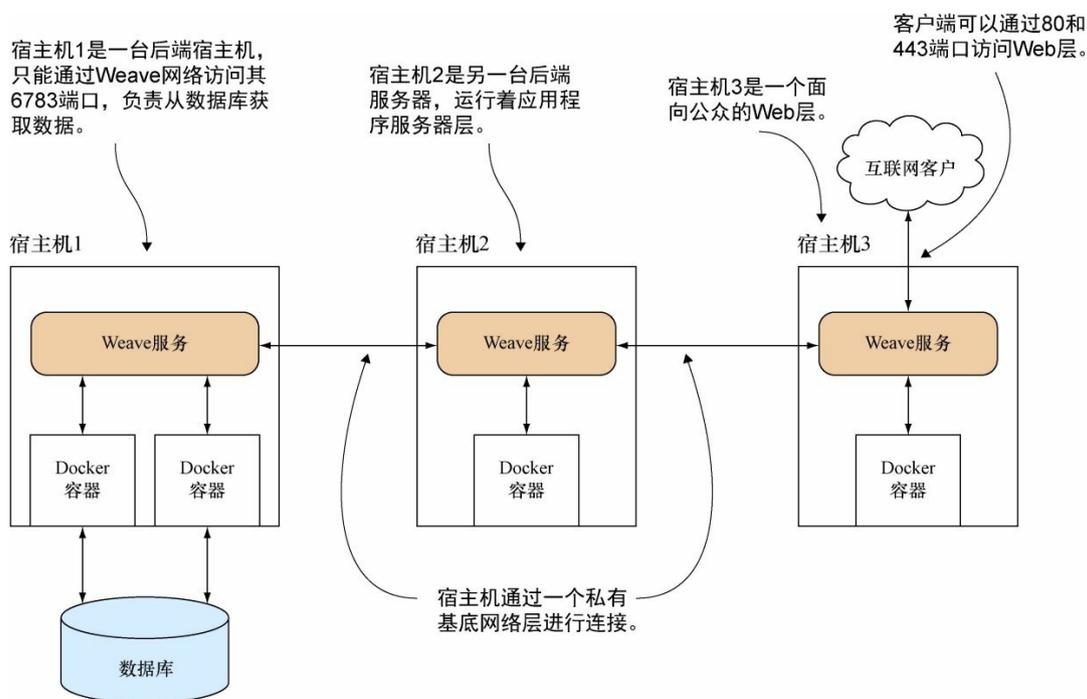


图8-3 一个典型的Weave网络

在图8-3中，宿主机1无法访问宿主机3，不过它们可以像本地连接一样通过Weave网络相互通信。Weave网络不对公众开放——只对Weave下启动的容器开放。这使得跨不同环境的代码开发、测试与部署变得相对简单，因为可以将所有情景中的网络拓扑都做成一样的。

1. 安装Weave

Weave是一个二进制程序，可以在<https://github.com/zettio/weave>找到其安装说明。

Weave需要安装在要作为Weave网络一部分的所有宿主机上。可执行以下指令：

```
$ sudo wget -O /usr/local/bin/weave \
https://github.com/zettio/weave/releases/download/latest_release/weave
$ sudo chmod +x /usr/local/bin/weave
```

Weave二进制文件冲突 如果在此处遇到问题，机器上可能已经存在一个作为其他软件包一部分的Weave二进制文件。

2. 设置Weave

要操作本示例，需要使用两台宿主机。此处分别将其命名为host1和host2。使用ping确保它们可以相互通信。接着获取两台宿主机的IP地址。

如何获取IP地址 一种获取宿主机IP地址的快速方法是使用浏览器访问<http://ip-addr.es>，或者运行 `curl http://ip-addr.es`。

网络防火墙 如果在此处遇到问题，网络中可能存在某种形式的防火墙。如果不清楚，请与网络管理员确认。特别需要注意的是，TCP与UDP的6783端口需要同时打开。

在第一台宿主主机上运行第一个Weave路由器：

```
host1$ curl http://ip-addr.es (1)
1.2.3.4
host1$ sudo weave launch (2)
host1$ C=$(sudo weave run 10.0.1.1/24 -t -i ubuntu) (3)
```

(1) 确认host1的IP地址

(2) 以root身份在host1上启动Weave服务。这一步需要在每台宿主主机上进行一次

(3) 在容器内启动Weave路由器。以CIDR标记法为它指定一个IP地址和网络。此处给定的IP地址和网络定义了私有Weave网络的样子

什么是**CIDR** CIDR是Classless Inter-Domain Routing（无类别域间路由）的缩写。它是分配IP地址和路由IP网络分组的一种方法。CIDR标记法由一个IP地址、一个斜杠及一个数字组成。最后的数字表示路由前缀中前导位的位数。其他位则形成了路由的地址空间。最后的数字越小，地址空间就越大。例如，一个192.168.2.0/24的网络将具有一个包含256个地址的8位地址空间，而一个16.0.0.0/8的网络将具有一个包含16 777 216个地址的地址空间。

在host2上可以执行类似的步骤，不过需要将host1的位置通知给Weave，并为它分配一个不同的IP地址：

```
host2$ curl http://ip-addr.es (1)
1.2.3.5
host2# sudo weave launch 1.2.3.4 (2)
host2# C=$(sudo weave run 10.0.1.2/24 -t -i ubuntu) (3)
```

(1) 确定host2的IP地址

(2) 以root身份在host2上启动Weave服务。这一次添加了第一台宿主机的公共IP地址以便它可以连接到另一台宿主机上

(3) 在容器内启动 Weave 路由器。为其指定了相同网络上与 host1不同的IP地址

除了应用程序容器的IP地址的选择之外，host2上唯一的不同是要通知Weave，让它与host1上的Weave对等（通过host2可以访问到的IP地址或主机名及可选的“:端口”进行指定）。

3. 测试连接

现在万事俱备，可以进行测试，看看容器是否能相互通信：

```
host1# docker attach $C (1)
root@28841bd02eff:/# ping -c 1 -q 10.0.1.2 (2)
PING 10.0.1.2 (10.0.1.2): 48 data bytes
--- 10.0.1.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss (3)
round-trip min/avg/max/stddev = 1.048/1.048/1.048/0.000 ms
```

(1) 附加到本次交互会话早前返回的容器ID上

(2) ping 另一台服务器分配到的IP地址

(3) 一个成功的ping响应

如果ping成功了，即可证实自行分配的私有网络内部的可连接性。现在可以将10.0.1.1分配给应用程序服务器，将10.0.1.2分配给Web服务器。

ICMP被阻塞？ 如果（ping使用的）ICMP协议信息被防火墙阻

塞，这一步就无法工作。如果出现这种情况，可以尝试telnet到另一台宿主机的6783端口来测试是否可以建立连接。

技巧74 Docker的网络与服务功能

Weave是一个优秀的工具，不过它依赖于Docker外部的一个工具，因此可能无法很好地与生态系统里的其他工具整合。

由于Weave这类工具的流行，Docker公司收集了对Docker内网络解决方案有兴趣的多家公司的反馈，并形成一个方案来尝试解决这些最迫切的需求，同时又不将人们限制在一个大而全的解决方案之中。这项工作还在进行中，不过已经接近发布了！

问题

想要一种由Docker公司支持的用于创建虚拟网络的解决方案。

解决方案

使用实验性的网络与服务功能。

讨论

在阅读本书时，Docker可能已经在稳定版本中发布了（目前是）实验性的网络功能。这一点可以通过运行以下命令来确认：

```
$ docker network --help

Usage: docker network [OPTIONS] COMMAND [OPTIONS] [arg...]

Commands:
  create          Create a network
  rm              Remove a network
  ls              List all networks
  info            Display information of a network

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

```
--help=false      Print usage
```

如果提示类似'`network`' is not a docker command的信息，则表示需要安装Docker的实验性版本。要安装实验性功能，可查阅GitHub上的Docker Experimental Features（Docker实验性功能）页面：<https://github.com/docker/docker/tree/master/experimental>。

实验性功能发生根本变更 本书尽可能地保持本技巧的时效性，不过实验性功能往往会发生改变。在试验本技巧时，相应的指令可能需要做轻微改动。

这项功能的高级别目标是允许在Docker中使用网络插件把虚拟网络的创建抽象掉。这些插件要么是内置的，要么由第三方提供，都将提供一个虚拟网络。在这些场景的背后，插件将完成创建网络所需的所有必要工作，让用户可以接着使用它。在理论上，类似Weave的工具应该可以变成网络插件，带来更奇特的用例。在实践中，这个功能的设计很可能是一个迭代的过程。

运行以下命令可查看Docker拥有的网络清单：

```
$ docker network ls
NETWORK ID          NAME                TYPE
04365ecf2eaa       none                null
c82bde52597d       host                host
7e8c8a0eab7d       bridge              bridge
```

可以看出，这就是在启动容器时传递给`docker run`的`--net`选项的可选值。下面添加一个新的`bridge`网络（一个让容器可以在其中自由通信的扁平化网络）：

```
$ docker network create --driver=bridge mynet
```

```

3265097deff3847cb1f7b8e8bc924bae1c439d8bf6247458400e620b35447292
$ docker network ls | grep mynet
3265097deff3          mynet          bridge
$ ip addr | grep mynet
34: mynet: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    state DOWN
    inet 172.18.42.1/16 scope global mynet
$ ip addr | grep docker
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    state DOWN
    inet 172.17.42.1/16 scope global docker0

```

这创建了一个新的网络接口，它将使用与普通Docker网桥不同的一个IP地址范围。

接下来启动两个容器，将服务绑定到该网络中。目前，服务的概念是紧密结合在网络功能里的，容器要参与特定的网络必须具有一个服务名：

```

$ docker run -it -d --name c1 ubuntu:14.04.2 bash (1)
87c67f4fb376f559976e4a975e3661148d622ae635fae4695747170c00513165
$ docker service publish c1service.mynet (2)
ed190f2cc0887ac87e1024ebb425f653989582942ab25a341e3d3e2a980475f5
$ docker service attach c1 c1service.mynet (3)
$ docker run -it -d --name c2 \
--publish-service=c2service.mynet ubuntu:14.04.2 bash (4)
0ee74a3e3444f27df9c2aa973a156f2827bcdd0852c6fd4ecfd5b152846dea5b
$ docker service ls --network mynet (5)
SERVICE ID      NAME           NETWORK      CONTAINER
ed190f2cc088     c1service     mynet        87c67f4fb376 (6)
21aef543af70     c2service     mynet        0ee74a3e3444

```

(1) 启动一个名为c1的容器

(2) 在mynet网络内部创建一个名为c1service的未关联的服务名

(3) 将容器c1 与服务c1service关联起来

(4) 在mynet网络内部创建一个名为c2 的容器以及一个c2service服务名

(5) 展示两个容器现在已经将服务绑定到mynet里

(6) 如果CONTAINER字段为空，记住要运行“docker service attach c1 c1service.mynet”来附加该服务

上述命令演示了注册服务的两种不同方法，即创建一个容器然后附加服务，以及一步完成创建和附加。

这二者之间存在一个差异。第一个容器将在启动时加入默认网络（通常是 Docker 网桥，不过这可以使用 Docker 守护进程的参数进行定制），然后添加一个新的网络接口以便可以同时访问mynet。第二个容器将只加入mynet——正常 Docker 网桥上的任何容器都无法访问它。

下面做一些连接性检查：

```
$ docker exec c1 ip addr | grep 'inet.*eth' (1)
  inet 172.17.0.6/16 scope global eth0
  inet 172.18.0.5/16 scope global eth1
$ docker exec c2 ip addr | grep 'inet.*eth' (2)
  inet 172.18.0.6/16 scope global eth0
$ docker exec c1 ping -qc1 c2service (3)
ping: unknown host c2service
$ docker exec c1 ping -qc1 172.18.0.6 (4)
PING 172.18.0.6 (172.18.0.6) 56(84) bytes of data.

--- 172.18.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.069/0.069/0.069/0.000 ms
$ docker exec c2 ping -qc1 c1service (5)
PING c1service (172.18.0.5) 56(84) bytes of data.

--- c1service ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.084/0.084/0.084/0.000 ms
```

(1) 列出c1的网络接口和IP地址

- (2) 列出c2的网络接口和IP地址
- (3) 尝试从容器1 ping容器2的服务
- (4) 尝试从容器1 ping容器2的IP地址
- (5) 尝试从容器2 ping容器1的服务

这里发生了很多事情！我们发现，尽管容器的实际IP地址对彼此都是绝对可用的，容器c1无法找到c2的服务。这是由服务的工作方式造成的——在添加和删除服务时，容器内的/etc/hosts将被更新，不过c1不会发生这样的情况，因为它初始启动在默认了docker0网桥上。这个意外的行为在未来可能会改变，也可能不会。

简单起见，建议尽可能坚持使用`--publish-service`，不过，由一个容器连接两个网络的场景对模拟现实世界堡垒主机的场景还是相当有用的。

这里未说明的一件事（因为项目还在进行中）是内置的叠加（overlay）网络插件——根据用例不同，将其作为Weave可能的替代品还是值得做些研究的。

8.4 小结

在评估 Docker 其他方面时，它提供的网络可能性最初可能会被忽略，因此，本章通过使用网络功能对其他功能进行补充，同时展现了其自身的价值。

本章主要讲述了：

- 如何使用Docker Compose;

- 使用链接的一种替代方案；
- 在恶劣的网络环境中对容器进行测试；
- 把不同宿主机的容器串联在一起。

在开发流水线中使用Docker的话题到此为止，是时候看看如何在生产环境中实际使用它了，第一步是探索如何实际地管理所有容器。

第四部分 生产环境中的Docker

终于到了该考虑如何在生产环境中运行Docker的时候了。第四部分我们将重点阐述Docker在生产环境中运行时应该注意的几个关键点。第9章涵盖了当下发展非常活跃的编排领域。只要在同一个环境中运行任意多个不同的容器，就需要考虑如何以一种一致且可靠的方式来管理它们，因此这一章我们将考察一些该领域目前最流行的工具。第10章将重点关注安全。通过一些实用的技巧，读者将对Docker带来的一些安全挑战和相应的解决方案有更真实的理解。第11章将重点关注备份、日志和资源管理等主题，我们将向读者展示在Docker上下文中如何管理传统系统管理任务。最后，在第12章，我们将话题转向问题排查，这一章涵盖了几个Docker有可能出问题的常见领域，以及如何在生产环境中调试容器。

第9章 容器编排：管理多个Docker容器

本章主要内容

- 在单台宿主机上管理Docker容器
- 将容器部署到多台宿主机
- 获取容器部署的位置信息

Docker依赖的技术实际上已经以不同形式存在一段时间了，但Docker是那个成功抓住技术行业兴趣点的解决方案。这把Docker推到了一个令人羡慕不已的位置——社区先驱们完成了这一系列工具的开创工作，这些工具又吸引使用者加入社区并不断地回馈社区，形成了一个自行运转的生态系统。

这片繁荣的景象在编排领域尤为明显。在这一领域提供服务的公司可以列出一大堆，看过这些公司名字的清单之后读者会发现，关于如何实现编排它们都有自己的看法，也都开发了自己的工具。

虽然该生态系统是Docker的一个巨大优势（这也是我们在本书中用如此大的篇幅介绍的原因），然而编排工具数量众多，无论对新手还是老手都有点儿难以抉择。本章将带读者浏览一些最受瞩目的工具，感受一下这些高端工具，以便在需要选用适合自己的框架时能更加了解情况。

有多种不同的方式来组合编排工具的家族树。图9-1展示了我们熟悉的一些工具。

树的根节点是`docker run`命令，这是启动容器最常用的方式。Docker家族的几乎所有工具都衍生于这一命令。树的左侧分支上的工具将一组容器视为单个实体，中间分支的工具借助`systemd`或服务文件管理容器，右侧分支上的工具将单个容器视为单个实体。沿着这些分支往下，这些工具做的事情越来越多，例如，它可以跨多台宿主机工作，或者让用户远离手动部署容器的繁琐操作。

读者可能会注意到图 9-1 中看似孤立的两个区域——Mesos和Consul/etcd/Zookeeper组。Mesos是一个有趣的东西，它在Docker之前就已经存在，并且它对Docker的支持是一个附加功能，而不是核心功能。虽然它做得不错，但是也需要仔细评估，如果仅仅是从功能特性上来看，用户可能在其他工具中也想要有这些。相比之下，Consul、etcd和Zookeeper根本不是编排工具。相反，它们为编排提供了重要的补充功能——服务发现。

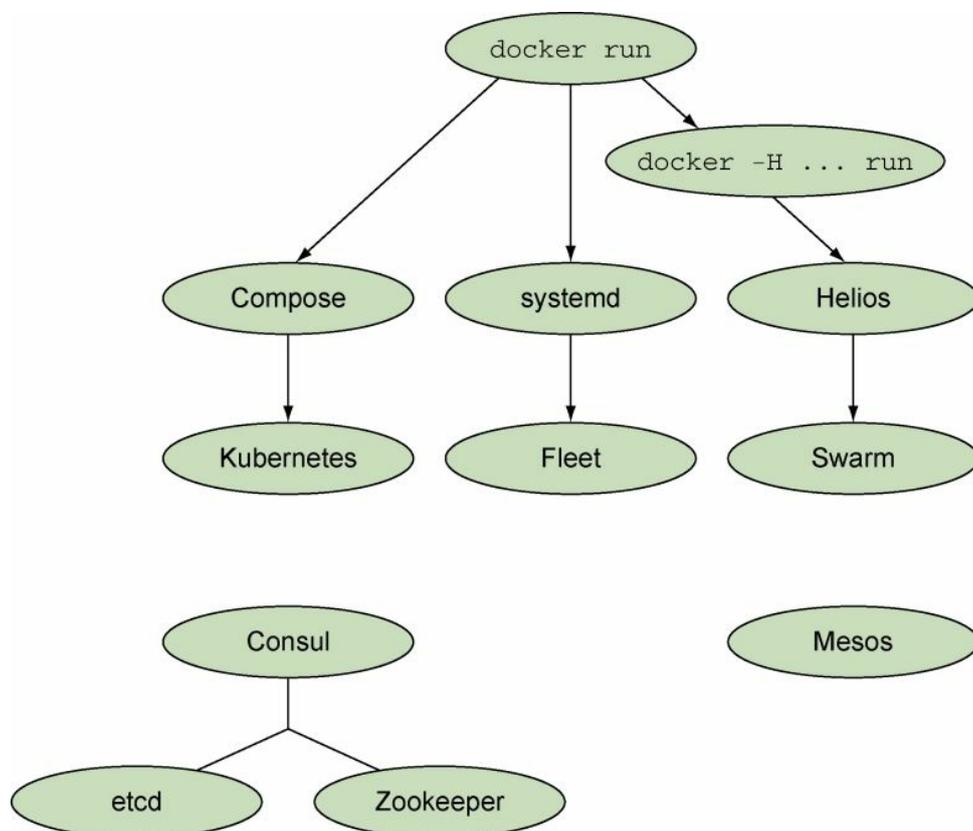


图9-1 Docker生态系统中的编排工具

阅读本章时，时不时地回顾每个编排工具，并尝试和提出一个适用场景，可能会有助于确定哪款特定的工具适合用户的需求。我们还会提供一些示例，供读者开始学习。

接下来先从单台宿主机开始。

9.1 简单的单台宿主机

在本地机器上管理容器可能是一种痛苦的体验。Docker为长期运行的容器提供的管理功能比较原始，而启动带有链接和共享卷的容器更是一个令人沮丧的手动过程。

在第8章里我们讲过如何使用Docker Compose来更方便地管理链接，因此现在我们把重心转向另一个痛点，来看看如何使单机下长期运行容器的管理变得更加健壮。

技巧75 使用systemd管理宿主机上的容器

在这一技巧里，我们将使用systemd配置一个简单的Docker服务。如果读者熟悉systemd，跟进本章内容将会相对容易些，但我们假设读者之前对此工具并不了解。

对于一个拥有运维团队的成熟公司来说，使用systemd控制Docker是很有用的，因为他们更喜欢沿用自己已经了解并且已经工具化的经过生产验证的技术。

问题

想要管理宿主机上Docker容器服务的运行。

解决方案

使用systemd管理容器服务。

讨论

systemd是一个系统管理的守护进程，它在前段时间取代了Fedora的SysV init脚本。它可以通过独立单元的形式管理系统上的所有服务——从挂载点到进程，甚至到一次性脚本。它在被推广到其他发行版和操作系统后变得愈加受欢迎，虽然在一些系统上安装和启用它可能还有问题（编写本书时Gentoo就是一个例子）。设置systemd时，别人使用systemd过程中遇到类似问题的处理经验值得借鉴。

在本技巧里，我们将通过运行第1章中的to-do应用程序来演示如何使用systemd管理容器的启动。

安装systemd

如果用户的宿主机系统上还没有安装systemd（可以运行systemctl status命令来检查，查看是否能得到正确的响应），可以使用标准包管理工具将其直接安装到宿主机的操作系统上。如果不太习惯以这种方式与宿主机系统交互，推荐使用Vagrant来部署一个已经安装好systemd的虚拟机，如代码清单9-1所示。这里我们只做简要介绍，有关安装Vagrant的更多建议参见附录C。

代码清单9-1 设置Vagrant

```
$ mkdir centos7_docker (1)
$ cd centos7_docker
$ vagrant init jdiprizio/centos-docker-io (2)
$ vagrant (3)
$ vagrant ssh (4)
```

- (1) 创建并进入一个新的目录
- (2) 将目录初始化成一个Vagrant环境，指定Vagrant镜像
- (3) 启动虚拟机
- (4) 采用SSH的方式登入虚拟机

jdiprizio/centos-docker-io镜像库不再可用？ 在编写本书时，**jdiprizio/centos-docker-io**是一个合适并可用的虚拟机镜像。如果读者阅读本书时发现它已经失效，可以使用另一个镜像名称来替换代码清单9-1中的这一字符串。读者可以在Atlas的“Discover Vagrant Boxes”页面上搜索一个镜像：
<https://atlas.hashicorp.com/boxes/search>（**box**是一个Vagrant用来指代虚拟机镜像的术语）。要查找该镜像，我们可以搜索“**docker centos**”。在启动新的虚拟机之前，读者可能需要查看**vagrant box add**命令行的帮助文档，了解如何下载该虚拟机。

用systemd设置一个简单的Docker应用程序

现在机器上安装好了systemd和Docker，接下来使用该机器运行第1章中讲到的to-do应用程序。

systemd通过读取INI格式的配置文件来工作。

INI文件 INI文件是一种简单的文本文件，其基本结构由节、属性和值组成。

首先以root身份创建一个服务文

件/etc/systemd/system/todo.service，如代码清单9-2所示。在这个文件里告诉systemd在宿主机的8000端口上运行一个名为todo的Docker容器。

代码清单9-2 /etc/systemd/system/todo.service

```
[Unit] (1)
Description=Simple ToDo Application
After=docker.service (2)
Requires=docker.service (3)

[Service] (4)
Restart=always (5)
ExecStartPre=/bin/bash \
-c '/usr/bin/docker rm -f todo || /bin/true' (6)
ExecStartPre=/usr/bin/docker pull dockerinpractice/todo (7)

ExecStart=/usr/bin/docker run --name todo \
-p 8000:8000 dockerinpractice/todo (8)
ExecStop=/usr/bin/docker rm -f todo (9)

[Install] (10)
WantedBy=multi-user.target (11)
```

- (1) Unit部分定义了systemd对象的通用信息
- (2) Docker服务启动之后立即启动这个单元
- (3) 该单元成功运行的前提是运行Docker服务
- (4) Service 部分定义了与systemd 服务单元类型相关的配置信息
- (5) 如果服务终止了，总是重启它
- (6) ExecStartPre 定义了一个命令。该命令会在该单元启动前运行。要确保启动该单元前容器已经删掉，可以在这里删除它

- (7) 确保运行容器之前已经下载了该镜像
- (8) ExecStart 定义了服务启动时要运行的命令
- (9) ExecStop 定义了服务停止时要运行的命令
- (10) Install 部分包含了启用该单元时systemd所需的信息
- (11) 告知systemd当进入多用户目标环境的时候希望启动该服务单元

从该配置文件可以非常清楚地看出，systemd为进程的管理提供了一种简单的声明式模式，将依赖管理的细节交给systemd服务去处理。但这并不意味着用户可以忽视这些细节，只是它确实为用户提供了很多方便的工具来管理Docker（和其他）进程。

Docker重启策略和进程管理器 默认情况下Docker不会设置任何重启策略，但值得注意的是，一旦有所设置，它都会和大多数的进程管理器冲突。因此，如果使用了进程管理器就不要设置重启策略。

启动一个新的服务单元即是调用systemctl enable命令。如果希望系统启动的时候该服务单元能够自动启动，也可以在systemd的multi-user.target.wants目录下创建一个符号链接。一旦完成，就可以使用systemctl start来启动该单元了：

```
$ systemctl enable /etc/systemd/system/todo.service
$ ln -s '/etc/systemd/system/todo.service' \
'/etc/systemd/system/multi-user.target.wants/todo.service'
$ systemctl start todo.service
```

然后只要等它启动。如果出现问题会有相应的提示。

可以使用`systemctl status`命令来检查是否一切正常。它会打印一些关于该服务单元的通用信息，如进程运行的时间以及相应的进程 ID，紧随其后的是该进程的日志信息。通过以下例子可以看出 Swarm 服务端在 8000 端口下正常启动：

```
[root@centos system]# systemctl status todo.service
todo.service - Simple ToDo Application
  Loaded: loaded (/etc/systemd/system/todo.service; enabled)
  Active: active (running) since Wed 2015-03-04 19:57:19 UTC; 2min 13s ago
 Process: 21266 ExecStartPre=/usr/bin/docker pull dockerinpractice/todo
  └─ (code=exited, status=0/SUCCESS)
 Process: 21255 ExecStartPre=/bin/bash -c /usr/bin/docker rm -f todo ||
  └─ /bin/true (code=exited, status=0/SUCCESS)
 Process: 21246 ExecStartPre=/bin/bash -c /usr/bin/docker kill todo ||
  └─ /bin/true (code=exited, status=0/SUCCESS)
 Main PID: 21275 (docker)
  CGroup: /system.slice/todo.service
          └─?21275 /usr/bin/docker run --name todo
              └─ -p 8000:8000 dockerinpractice/todo
Mar 04 19:57:24 centos docker[21275]: TodoApp.js:117:
  └─ // TODO scroll into view
Mar 04 19:57:24 centos docker[21275]: TodoApp.js:176:
  └─ if (i>=list.length()) { i=list.length()-1;} // TODO .length
Mar 04 19:57:24 centos docker[21275]: local.html:30:
  └─ <!-- TODO 2-split, 3-split -->
Mar 04 19:57:24 centos docker[21275]: model/ToDoList.js:29:
  └─ // TODO one op - repeated spec? long spec?
Mar 04 19:57:24 centos docker[21275]: view/Footer.jsx:61:
  └─ // TODO: show the entry's metadata
Mar 04 19:57:24 centos docker[21275]: view/Footer.jsx:80:
  └─ todoList.addObject(new TodoItem()); // TODO create default
Mar 04 19:57:24 centos docker[21275]: view/Header.jsx:25:
  └─ // TODO list some meaningful header (apart from the id)
Mar 04 19:57:24 centos docker[21275]: > todomvc-swarm@0.0.1 start /todo

Mar 04 19:57:24 centos docker[21275]: > node TodoAppServer.js
Mar 04 19:57:25 centos docker[21275]: Swarm server started port 8000
```

本技巧中介绍的一些原理不只适用于 `systemd`，大部分进程管理器，包括其他的 `init` 系统，都可以采用类似的方式来配置。

在下一个技巧里，我们会更进一步，使用 `systemd` 来实现在技巧

69中创建的SQLite服务器。

技巧76 使用systemd编排宿主机上的容器

不同于docker-compose（编写本书时），systemd已经是一个用于生产的成熟技术。在本技巧中，我们将展示如何使用systemd来实现和docker-compose类似的本地编排功能。

如果读者在学习本技巧的过程中遇到问题，可能需要升级一下Docker版本，1.7.0及以上版本应该会工作正常。

问题

想要在生产环境的宿主机上管理更复杂的容器编排。

解决方案

使用systemd和相应的依赖服务来管理容器。

讨论

为了展示systemd在更复杂的场景中的应用，我们会在systemd中重新实现一遍技巧69中提到的SQLite TCP服务器。

图9-2展示了我们计划实现的systemd服务单元配置中的依赖。

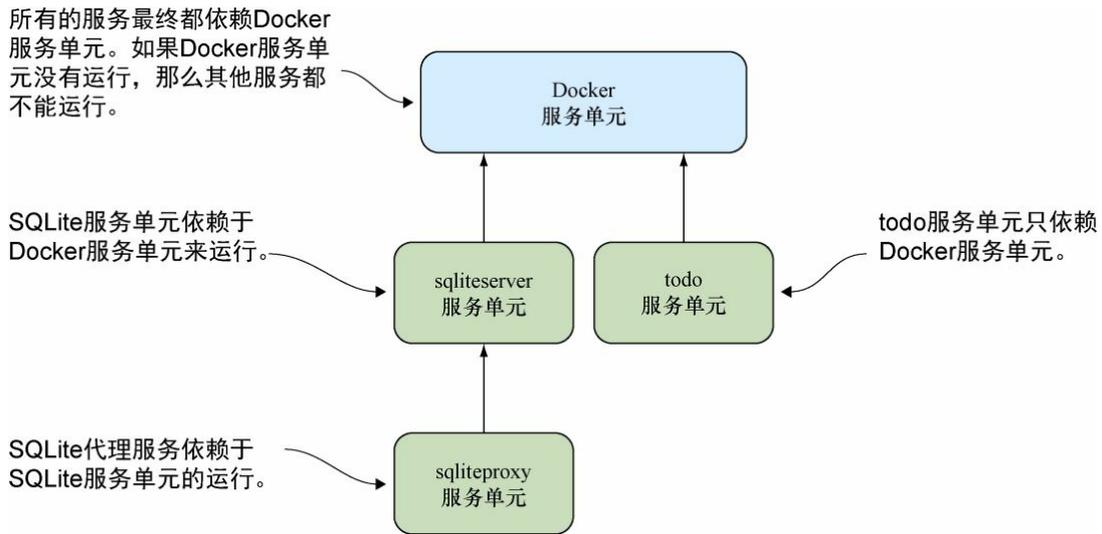


图9-2 systemd单元依赖图

这里的模式和技巧69中Docker Compose例子中的模式是类似的。其中有一个关键的差别，SQLite服务在这里不再被当成一个单体对象，这里的每个容器都是一个离散的实体。在这个场景下，SQLite代理可以独立于SQLite服务器，被单独停止。

代码清单9-3展示了sqliteserver服务的代码。像以前一样，它依赖Docker服务，但和前面介绍的to-do实例有一些不同之处。

代码清单9-3 /etc/systemd/system/sqliteserver.service

```
[Unit] (1)
Description=SQLite Docker Server
After=docker.service (2)
Requires=docker.service (3)

[Service]
Restart=always
ExecStartPre=-/bin/touch /tmp/sqlitedbs/test (4)
ExecStartPre=-/bin/touch /tmp/sqlitedbs/live
ExecStartPre=/bin/bash \
-c '/usr/bin/docker kill sqliteserver || bin/true'
ExecStartPre=/bin/bash \ (5)
-c '/usr/bin/docker rm -f sqliteserver || /bin/true'
ExecStartPre=/usr/bin/docker \
pull dockerinpractice/docker-compose-sqlite (6)
ExecStart=/usr/bin/docker run --name sqliteserver \ (7)
```

```
-v /tmp/sqlitedbs/test:/opt/sqlite/db \  
dockerinpractice/docker-compose-sqlite /bin/bash -c \  
'socat TCP-L:12345,fork,reuseaddr \  
EXEC:"sqlite3 /opt/sqlite/db",pty'  
ExecStop=/usr/bin/docker rm -f sqliteserver (8)  
  
[Install]  
WantedBy=multi-user.target
```

- (1) Unit小节定义了该systemd对象的通用信息
- (2) Docker服务启动之后启动该单元
- (3) 为了让该服务正常运行，Docker服务必须处于正常运行状态
- (4) 这几行代码确保服务启动之前SQLite的数据库文件是存在的，touch命令行之前的-告诉systemd：如果该命令返回错误代码则表明启动失败
- (5) ExecStartPre定义了服务单元被启动之前运行的命令。为了确保容器在用户启动之前已被删除，这里使用了一个前置命令将其删除
- (6) 确保启动容器之前镜像已下载完成了
- (7) ExecStart 定义了服务被启动之后运行的命令。这里值得注意的是，我们在另一个/bin/bash-c调用中包含了socat 命令，因为在ExecStart 这一行定义的命令是由systemd来运行的
- (8) ExecStop定义了服务停止之后运行的命令

要求绝对路径 systemd中使用的路径必须是绝对路径。

代码清单9-4列出的是sqliteproxy服务。这里最大的区别在于，代理服务依赖于刚刚定义的服务器进程，而服务端进程又依赖于Docker服务。

代码清单9-4 /etc/systemd/system/sqliteproxy.service

```
[Unit]
Description=SQLite Docker Proxy
After=sqliteserver.service (1)
Requires=sqliteserver.service (2)

[Service]
Restart=always
ExecStartPre=/bin/bash -c '/usr/bin/docker kill sqliteproxy || /bin/true'
ExecStartPre=/bin/bash -c '/usr/bin/docker rm -f sqliteproxy || /bin/true'
ExecStartPre=/usr/bin/docker pull dockerinpractice/docker-compose-sqlite
ExecStart=/usr/bin/docker run --name sqliteproxy \
-p 12346:12346 --link sqliteserver:sqliteserver \
dockerinpractice/docker-compose-sqlite /bin/bash \
-c 'socat TCP-L:12346,fork,reuseaddr TCP:sqliteserver:12345' (3)
ExecStop=/usr/bin/docker rm -f sqliteproxy

[Install]
WantedBy=multi-user.target
```

- (1) 该代理单元必须在前面定义的sqliteserver服务之后运行
- (2) 启动该代理之前要求服务器实例在运行
- (3) 该命令用于运行容器

通过这两个配置文件，我们为在systemd控制下安装和运行SQLite服务奠定了基础。现在我们可以启用这些服务了：

```
$ sudo systemctl enable /etc/systemd/system/sqliteserver.service
ln -s '/etc/systemd/system/sqliteserver.service' \
'/etc/systemd/system/multi-user.target.wants/sqliteserver.service'
$ sudo systemctl enable /etc/systemd/system/sqliteproxy.service
ln -s '/etc/systemd/system/sqliteproxy.service' \
```

```
'/etc/systemd/system/multi-user.target.wants/sqliteproxy.service'
```

然后启动它们：

```
$ sudo systemctl start sqlit\eproxy
$ telnet localhost 12346
[vagrant@centos ~]$ telnet localhost 12346
Trying ::1...
Connected to localhost.
Escape character is '^]'.
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from t1;
select * from t1;
test
```

值得注意的是，`sqliteproxy`服务依赖于`sqliteserver`服务的运行。只需要启动`sqliteproxy`服务即可，其他依赖的服务会自动启动。

9.2 多宿主主机Docker

目前读者应该对单台机器上相对复杂的Docker部署和编排已经有了一定的信心，现在该考虑一些更复杂的事情了——让我们进入多宿主主机世界，这样我们便可以更大规模地使用Docker。

将Docker容器迁移到目标机器并启动它们的最佳流程在Docker世界里一直颇具争议。一些知名公司创造了自己的方式并发布给全世界使用。如果用户可以自行决定使用什么工具的话，他会从中获益不少。

这是一个快速变动的话题——我们已经看到了Docker的多个编排工具的诞生和消亡，而在考虑是否迁移到全新的工具时建议保持谨慎。因此，我们应该试图去选用那些显著稳定或是势头强劲（或两者兼有）的工具。

技巧77 使用Helios手动管理多宿主机Docker

将一组机器的初始化和部署工作都交给一个应用程序确实容易让人恐慌，如果能有一种手动的方式来操作可以缓解这种恐慌。

对主要采用静态基础设施并且希望将 Docker 用于关键业务时这一过程能够有人力监督（可以理解）的公司来说，Helios是理想的选择。

问题

想要初始化多台Docker宿主机环境来运行容器，同时又保留对整个过程的手动控制。

解决方案

使用Spotify公司的Helios工具。

讨论

Helios是Spotify公司目前在生产环境中用来管理其服务器的工具，它具有易于上手和稳定的友好特性（如你所望）。Helios允许用户管理Docker容器在多台宿主机上的部署。它提供了一个简单的命令行接口，用户可以用它来指定运行的内容以及运行的位置，也可以查看当前运行的状态。

因为这里只是介绍Helios，简单起见，我们将在Docker内的单个节点上运行所有内容——不用担心，与多宿主机运行场景有关的一切都会被清楚地着重标示出来。Helios的整体架构如图9-3所示。

如图9-3所示，运行Helios时只需要运行一个额外的服务Zookeeper即可。Helios使用Zookeeper跟踪所有宿主机的状态，同时

它也作为主机和代理节点之间的通信通道。

什么是**Zookeeper** Zookeeper是一款轻量级的分布式数据库，经过优化用于存储Java编写的配置信息。它是Apache开源软件产品套件之一，功能类似于etcd（第7章介绍过该工具，本章也会再次出现）。

在本技巧中读者只需要知道Zookeeper是用来存放数据的，以便通过运行多个Zookeeper实例的方式将数据分布在多个节点上，从而实现可扩展性和可靠性。这可能听起来和第7章中对etcd的描述有些类似——这两个工具在功能上有很大的重叠。

执行如下命令来运行我们将在本技巧中使用的单个Zookeeper实例：

```
$ docker run --name zookeeper -d jplock/zookeeper:3.4.6
cd0964d2ba18baac58b29081b227f15e05f11644adfa785c6e9fc5dd15b85910
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' zookeeper
172.17.0.9
```

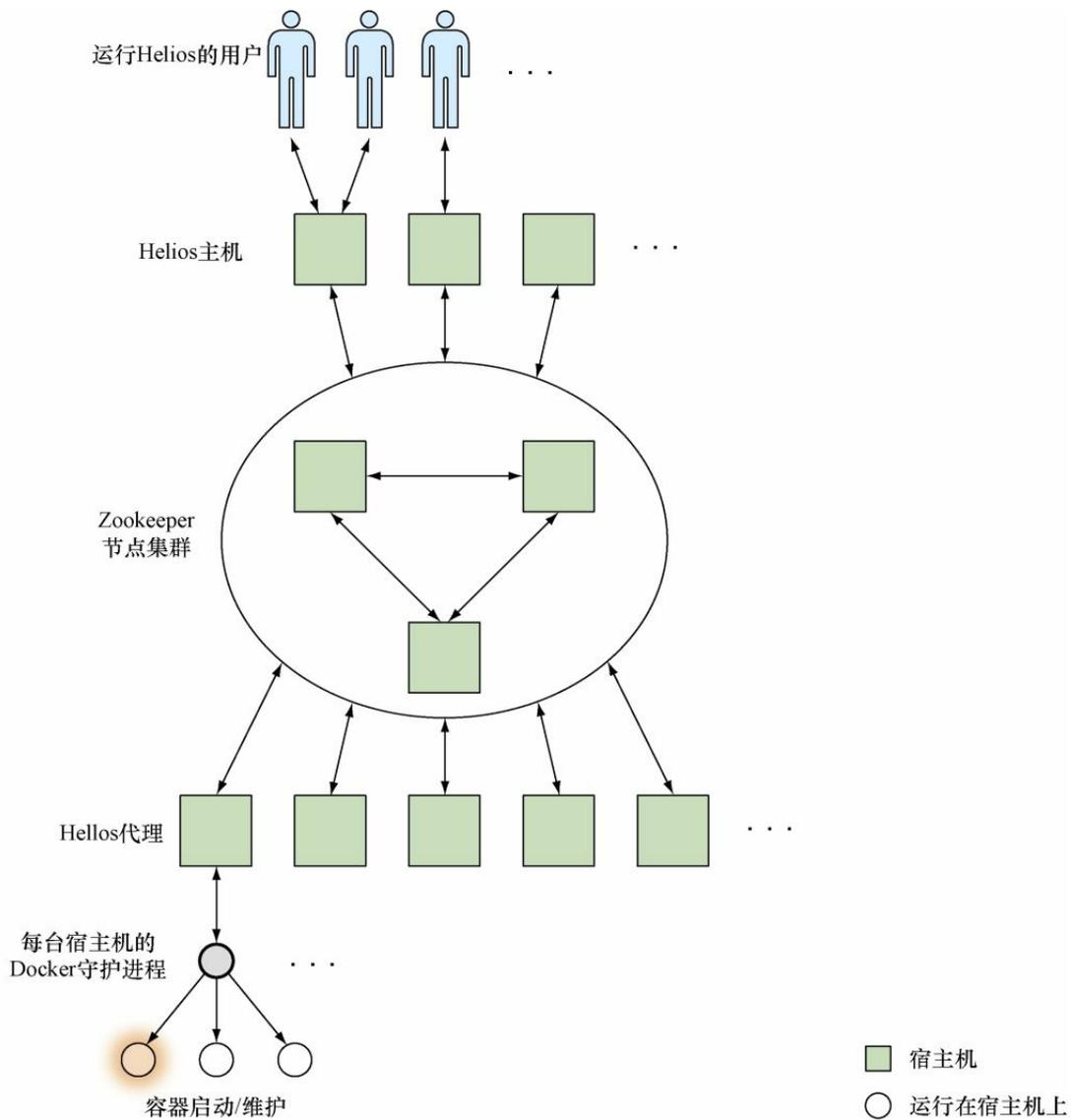


图9-3 Helios安装的鸟瞰图

宿主机和其他节点上的端口 在自身节点启动Zookeeper实例时，用户需要暴露一些服务端口供其他宿主机访问，并且使用卷来持久化数据。查看Docker Hub上的Dockerfile（<https://hub.docker.com/r/jplock/zookeeper/~/dockerfile/>）来了解应该使用哪些端口和文件夹等细节。读者也可能想要在多个节点上运行Zookeeper，但是配置一个Zookeeper集群超出了本技巧的范畴。

可以使用zkCli.sh工具检查Zookeeper存储的数据，既可以通过交互的方式也可以通过管道输入。该工具的启动过程提示信息非常丰富，但它会立马进入一个交互式命令行，用户可以针对存储Zookeeper数据的文件树状结构执行一些命令：

```
$ docker exec -it zookeeper bin/zkCli.sh
Connecting to localhost:2181
2015-03-07 02:56:05,076 [myid:] - INFO [main:Environment@100] - Client
  ➤ environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:
09 GMT
2015-03-07 02:56:05,079 [myid:] - INFO [main:Environment@100] - Client
  ➤ environment:host.name=917d0f8ac077
2015-03-07 02:56:05,079 [myid:] - INFO [main:Environment@100] - Client
  ➤ environment:java.version=1.7.0_65
2015-03-07 02:56:05,081 [myid:] - INFO [main:Environment@100] - Client
  ➤ environment:java.vendor=Oracle Corporation
[...]
2015-03-07 03:00:59,043 [myid:] - INFO [main-SendThread(localhost:2181)
:
  ➤ ClientCnxn$SendThread@1235] - Session establishment complete on serv
er
  ➤ localhost/0:0:0:0:0:0:1:2181, sessionId = 0x14bf223e159000d, negot
iated
  ➤ timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
```

目前没有对Zookeeper进行任何操作，因此目前仅存储了一些Zookeep内部信息。先保留这个命令行提示符开放，稍后我们会继续用到它。

Helios本身由以下3部分组成：

- 主机（master）——它通常是用作对Zookeeper中数据进行修改的接口；
- 代理（agent）——运行在每台Docker宿主机上，启动和停止基

- 于Zookeeper的容器，然后报告状态；
- 命令行工具——用于向主机发起请求。

图9-4展示了完成对系统发起的操作时最终系统是如何处理的（箭头表示数据流）。

现在Zookeeper已经运行起来了，是时候去启动Helios了。启动主机时需要指定前面启动的Zookeeper节点的IP地址：

```

$ IMG=dockerinpractice/docker-helios
$ docker run -d --name hmaster $IMG helios-master --zk 172.17.0.9
896bc963d899154436938e260b1d4e6fdb0a81e4a082df50043290569e5921ff
$ docker logs --tail=3 hmaster
03:20:14.460 helios[1]: INFO [MasterService STARTING] ContextHandler:
➤ Started i.d.j.MutableServletContextHandler@7b48d370{/,null,AVAILABLE
}
03:20:14.465 helios[1]: INFO [MasterService STARTING] ServerConnector:
➤ Started application@2192bcac{HTTP/1.1}{0.0.0.0:5801}
03:20:14.466 helios[1]: INFO [MasterService STARTING] ServerConnector:
➤ Started admin@28a0d16c{HTTP/1.1}{0.0.0.0:5802}
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' hmaster
172.17.0.11

```

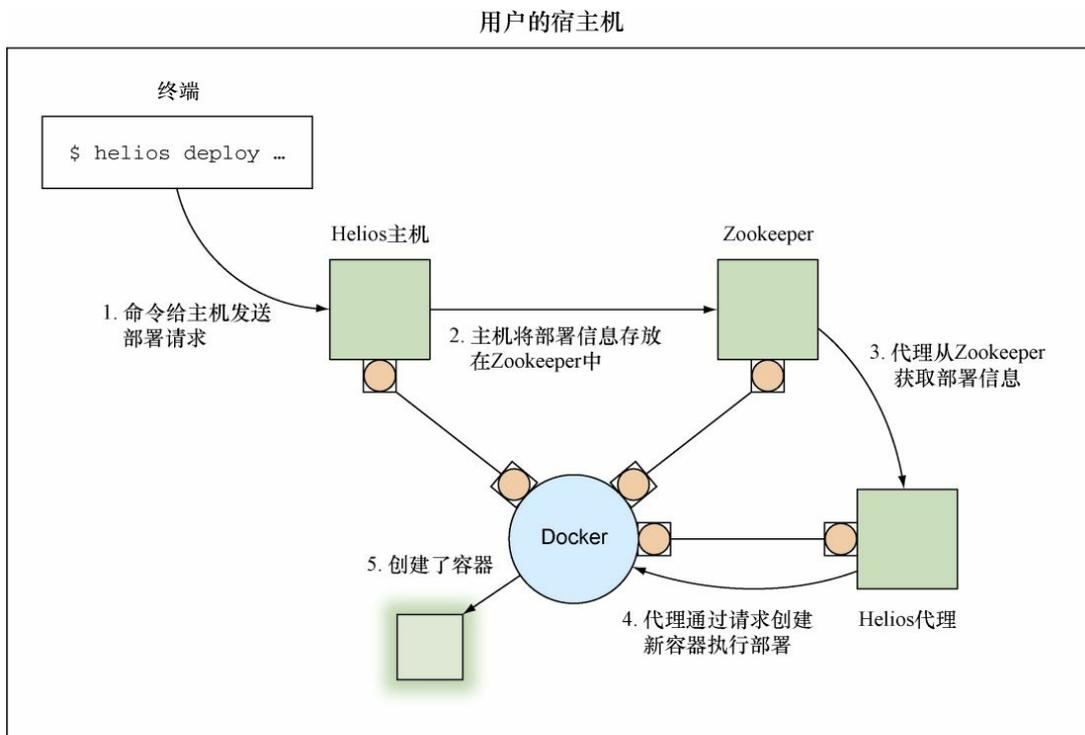


图9-4 在安装了Helios的单台宿主机上启动容器

现在看看Zookeeper里面新增了些什么：

```
[zk: localhost:2181(CONNECTED) 1] ls /
[history, config, status, zookeeper]
[zk: localhost:2181(CONNECTED) 2] ls /status/masters
[896bc963d899]
[zk: localhost:2181(CONNECTED) 3] ls /status/hosts
[]
```

看起来Helios主机已经创建了一堆新的配置，包括将自身注册为主机。遗憾的是，现在还没有任何宿主机。让我们通过启动一个代理来解决这个问题。该代理将使用当前宿主机的Docker套接字来启动容器：

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock -d --name hagent \
dockerinpractice/docker-helios helios-agent --zk 172.17.0.9
5a4abcb27107d0171ca809ff2beafac5798e86131b72aeb201fe27df64b2698
$ docker logs --tail=3 hagent
03:30:53.344 helios[1]: INFO [AgentService STARTING] ContextHandler:
➤ Started i.d.j.MutableServletContextHandler@774c71b1{/,null,AVAILABLE
}
03:30:53.375 helios[1]: INFO [AgentService STARTING] ServerConnector:
➤ Started application@7d9e6c27{HTTP/1.1}{0.0.0.0:5803}
03:30:53.376 helios[1]: INFO [AgentService STARTING] ServerConnector:
➤ Started admin@2bceb4df{HTTP/1.1}{0.0.0.0:5804}
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' hagent
172.17.0.12
```

再次检查Zookeeper：

```
[zk: localhost:2181(CONNECTED) 4] ls /status/hosts
[5a4abcb27107]
[zk: localhost:2181(CONNECTED) 5] ls /status/hosts/5a4abcb27107
[agentinfo, jobs, environment, hostinfo, up]
[zk: localhost:2181(CONNECTED) 6] get /status/hosts/5a4abcb27107/agenti
nfo
{"inputArguments":["-Dcom.sun.management.jmxremote.port=9203", [...]
[...]
```

可以看到/status/hosts现在有了一条数据。深入该宿主机的Zookeeper目录可以看到这里面有Helios针对该宿主机存储的内部信息。

多宿主机设置时需要提供宿主机名 如果运行在多宿主机环境下，将需要把--name `\$(hostname -f)`作为Helios主机和代理使用的参数传递进去。用户还需要为宿主暴露5801和5802端口，为代理暴露5803和5804端口。

让我们一起来简化和Helios的交互：

```
$ alias helios="docker run -i --rm dockerinpractice/docker-helios \
helios -z http://172.17.0.11:5801"
```

上面的别名意味着调用helios的话将会启动一个一次性容器来执行所需的操作，并且在开头指向正确的helios集群。注意，cli需要指向Helios主机而不是Zookeeper。

现在一切准备就绪。我们可以轻松地与Helios集群进行交互了，不妨尝试下面这个示例：

```
$ helios create -p nc=8080:8080 netcat:v1 ubuntu:14.04.2 -- \
sh -c 'echo hello | nc -l 8080'
Creating job: {"command":["sh","-c","echo hello | nc -l 8080"],
➤ "creatingUser":null,"env":{},"expires":null,"gracePeriod":null,
➤ "healthCheck":null,"id":
➤ "netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac",
➤ "image":"ubuntu:14.04.2","ports":{"nc":{"externalPort":8080,
➤ "internalPort":8080,"protocol":"tcp"}},"registration":{}},
➤ "registrationDomain":"","resources":null,"token":"","volumes":{}}
Done.
netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac
$ helios jobs
```

JOB ID	NAME	VERSION	HOSTS	COMMAND	ENVIRONME
NT					
netcat:v1:2067d43	netcat	v1	0	sh -c "echo hello nc -l 8080"	

Helios是围绕作业（job）的概念建立的——所有被执行的对象都必须被表示为一个作业，然后才能被发送到要执行的主机。至少，用户需要一个镜像，带上Helios需要知道的如何启动容器的基本信息：一个要执行的命令以及任意端口、卷或者环境选项。用户可能还需要一些其他高级选项，包括健康检查、过期时间和服务注册等。

上面的第一条命令创建了一个监听8080端口的作业，访问该端口时会先输出hello，然后终止运行。

可以使用helios hosts列出可用于作业部署的主机，然后使用helios deploy命令完成部署。然后helios status命令将显示作业已成功启动：

```
$ helios hosts
HOST      STATUS      DEPLOYED RUNNING CPUS MEM LOAD AVG MEM USAGE
└─ OS      HELIOS DOCKER
5a4abcb27107.Up 19 minutes 0      0      4      7 gb 0.61      0.84
└─ Linux 3.13.0-46-generic 0.8.213 1.3.1 (1.15)
$ helios deploy netcat:v1 5a4abcb27107
Deploying Deployment{jobId=netcat:v1:
└─ 2067d43fc2c6f004ea27d7bb7412aff502e3cdac, goal=START, deployerUser=null}
└─ on [5a4abcb27107]
5a4abcb27107: done
$ helios status
JOB ID      HOST      GOAL STATE      CONTAINER ID PORTS
netcat:v1:2067d43 5a4abcb27107.START RUNNING b1225bc      nc=8080:8080
```

当然，我们现在要验证服务是否正常工作：

```
$ curl localhost:8080
hello
$ helios status
JOB ID      HOST      GOAL STATE      CONTAINER ID PORTS
netcat:v1:2067d43 5a4abcb27107.START PULLING_IMAGE b1225bc      nc=8080:8080
```

curl命令的结果清楚地表明服务正在正常工作，但是helios

`status`当前展示了一些有趣的信息。在定义作业时我们注意到，在输出完`hello`之后作业会终止，但上面的输出结果中显示的是`PULLING_IMAGE`状态。这涉及Helios是如何管理作业的——一旦将其部署到一台宿主机上，Helios会尽可能地确保作业处于运行状态。这个状态说明Helios正在进行完整的作业启动过程，包括确保镜像被成功拉取。

最后，我们需要自己手动做一些清理：

```
$ helios undeploy -a --yes netcat:v1
Undeploying netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac from
➔ [5a4abcb27107]
5a4abcb27107: done
$ helios remove --yes netcat:v1
Removing job netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac
netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac: done
```

我们要求从所有节点中删除该作业（如有必要就终止它，并且停止所有自动重启的设置），然后删除作业本身，这意味着它不能再被部署到任何节点。

Helios是一个将容器部署到多台宿主机的简单可靠的方案。和后续将讲到的一些技巧不同的是，它的背后没有“魔法”来确定合适的部署位置——它会严格地部署到给定的位置，不会出现意外。

技巧78 基于Swarm的无缝Docker集群

能够完全控制集群当然很好，但是有时候集群管理太细是没有必要的。实际上，如果用户的这些应用程序没有很复杂的需求，完全可以充分利用Docker可以在任何地方运行的承诺——实在没有任何理由不把容器丢到集群里，让集群决定在哪里运行它们。

Swarm在研究型的实验室环境中很有用处，如果实验室能够将计

算密集型的问题分解为一系列的小块，将使它们可以在机器集群里非常轻松地运行它们的问题。

问题

有一组安装了Docker的宿主机，想要启动容器并且不需要很细地管理它们的运行位置。

解决方案

使用Docker Swarm将宿主机集群当成一个Docker守护程序，并像平常一样运行Docker命令。

讨论

Docker Swarm由3个部分组成：代理（多个）、发现服务和主机。图9-5展示了这3个部分是如何在具备3个节点的Docker Swarm上交互的——每个节点上都安装了代理。

代理是一个运行在作为集群一部分的宿主机上的应用程序，它们将连接信息汇报到发现服务，并将宿主机转换为Docker Swarm中的一个节点。具有代理的宿主机都需要让Docker守护程序对外暴露一个端口——我们在技巧1中讲述了如何通过给Docker守护程序带上-H选项来实现这一点，并且我们假设使用默认的2375端口。

只有一个主机 默认情况下Swarm集群中只有一个主机。如果想要主机在出错情况下有容错机制，可以查看Docker官方提供的关于高可用的文档：<https://docs.docker.com/swarm/multi-manager-setup/>。

主节点启动时会联系发现服务，寻找集群中的节点。之后就可以直接连接到主机运行命令，主机会自动将请求转发给代理。

Docker 客户端最小要求 集群中的代理和客户端使用的所有 Docker 版本都必须至少是 1.4.0。应该尽量让所有版本都保持一致，但是如果客户端中的版本不比代理中的版本新，集群应该也能正常工作。

设置 Swarm 集群的第一步是选择一个想用的发现服务。这里有很多选择，可以在一个文件中列出一组 IP 地址，也可以使用 Zookeeper（还记得 Helios 吗？）。在本技巧中，我们将使用 Docker Hub 内置的带有令牌机制的发现服务。

发现服务后端 本技巧中的所有内容都可以在用户部署的服务中完成——借助 Docker Hub 提供的发现服务来注册用户的节点可以方便用户快速实现发现服务的功能。但如果用户不想把自己节点的 IP 地址放到潜在的公共区域（有些人可能会猜出用户的集群 ID），可以阅读相关的可选后端系统的文档：

<http://docs.docker.com/swarm/discovery/>。

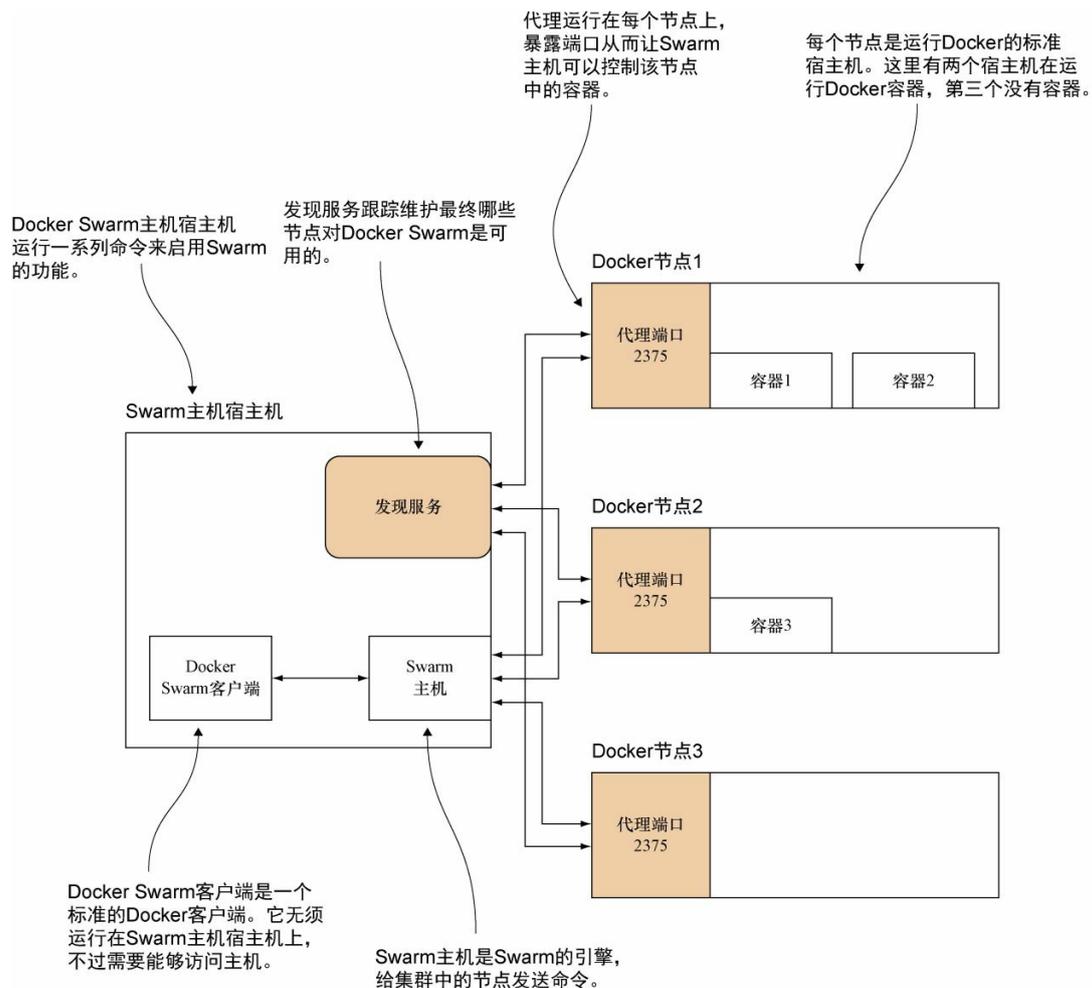


图9-5 包含3个节点的Docker Swarm集群

Docker Hub发现服务要求用户取得一个令牌（token）来识别用户的集群。因为它是Docker公司提供的服务，所以Swarm拥有内置的功能来简化这个过程。Swarm二进制文件（自然）可以作为Docker镜像使用，因此可以使用如下命令进行操作：

```
h1 $ docker pull swarm
h1 $ docker run swarm create
126400c309dbd1405cd7218ed3f1a25e
h1 $ CLUSTER_ID=126400c309dbd1405cd7218ed3f1a25e
```

`swarm create`命令后面的一长串字符串是用来标识用户集群的令牌。这个令牌非常重要——记下它！本技巧的后续内容中我们将使用`CLUSTER_ID`变量来引用它。

现在可以检查新创建的Swarm:

```
h1 $ docker run swarm list token://$CLUSTER_ID
h1 $ curl https://discovery-stage.hub.docker.com/v1/clusters/$CLUSTER_ID
D
[]
```

正如所见，当前还没什么东西。`swarm list`命令返回了空内容，同时（深入一点）直接查询Docker Hub发现服务里面集群的主机返回一个空列表。

启用了TLS的Docker守护程序 某些云服务提供了对启用了TLS的Docker守护程序的访问，读者也可以自己启用TLS。读者应该查阅Swarm文档，了解相关的生成证书并将其用于Swarm连接的最新信息：<https://docs.docker.com/v1.5/swarm/#tls>。

可以通过如下命令在当前机器上启动第一个代理：

```
h1 $ ip addr show eth0 | grep 'inet '
    inet 10.194.12.221/20 brd 10.194.15.255 scope global eth0
h1 $ docker run -d swarm join --addr=10.194.12.221:2375 token://$CLUSTER_ID
9bf2db849bac7b33201d6d258187bd14132b74909c72912e5f135b3a4a7f4e51
h1 $ docker run swarm list token://$CLUSTER_ID
10.194.12.221:2375
h1 $ curl https://discovery-stage.hub.docker.com/v1/clusters/$CLUSTER_ID
D
["10.194.12.221:2375"]
```

第一步是找出主机用于连接代理的IP地址，这种连接可以通过任何用户最熟悉的方式。启动代理时会使用该IP地址，并会将其信息发送给发现服务，这意味着`swarm list`命令输出会被更新，打印出新代理的信息。

现在还无法对节点执行任何操作——我们需要运行一个主机。因

为我们要在同一台机器上作为代理运行主机，而且标准的Docker端口也被暴露出来，所以需要为主机使用其他任意一个不同的端口：

```
h1 $ docker run -d -p 4000:2375 swarm manage token://$CLUSTER_ID
04227ba0c472000bafac8499e2b67b5f0629a80615bb8c2691c6ceda242a1dd0
h1 $ docker -H tcp://localhost:4000 info
Containers: 10
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 1
  h1: 10.194.12.221:2375
    ? Containers: 2
    ? Reserved CPUs: 0 / 4
    ? Reserved Memory: 0 B / 7.907 GiB
```

现在已经启动了主机，并运行docker info命令获取了该集群的一些详细信息。列出运行的两个容器分别是主机和代理。

现在，让我们在一个完全不同的节点上启动一个代理：

```
h2 $ docker run -d swarm join --addr=10.194.8.7:2375 token://$CLUSTER_I
D
h2 $ docker -H tcp://10.194.12.221:4000 info
Containers: 3
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
  h2: 10.194.8.7:2375
    ? Containers: 1
    ? Reserved CPUs: 0 / 4
    ? Reserved Memory: 0 B / 3.93 GiB
  h1: 10.194.12.221:2375
    ? Containers: 2
    ? Reserved CPUs: 0 / 4
    ? Reserved Memory: 0 B / 7.907 GiB
```

另一个节点已经添加到了集群里。注意，这里利用了从另一台机器访问主机的能力。

Swarm策略和过滤器 读者可能已经注意到docker info的输出结果中以Strategy和Filters开头的几行。这里暗示了几个Swarm提

现在读者已经了解了容器编排的两种极端方式——比较保守的Helios方式以及更自由的Docker Swarm方式。但有些用户或者公司可能期望他们使用的工具更复杂一些。这种可定制编排的需求有很多工具可以满足，不过有些工具使用率和活跃度比其他工具要高一些。从某种角度讲，这部分原因无疑在于其背后的品牌，而用户寄希望于Google知道如何构建编排软件。

Kubernetes适合那种希望对编排应用程序和应用程序之间的关系具有清晰的指引和最佳实践的公司。它允许用户使用一些专门设计的工具来管理基于特定结构的动态基础设施。

问题

想要跨宿主机管理Docker服务。

解决方案

使用Kubernetes。

讨论

在正式介绍Kubernetes的细节之前，让我们快速浏览一下图9-6所示的Kubernetes宏观架构图。

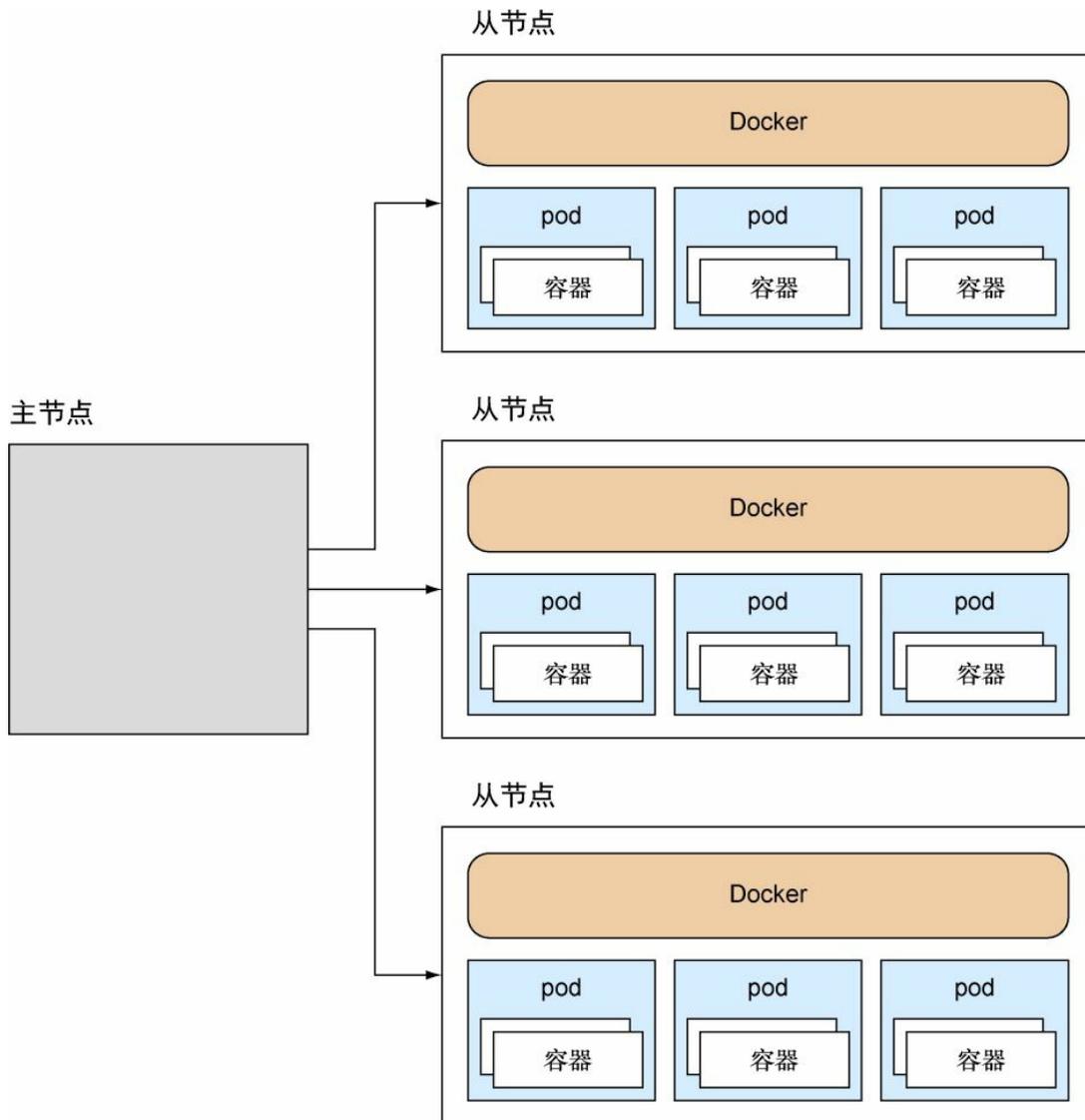


图9-6 Kubernetes高层视图

Kubernetes有一个主从架构（master-minion architecture）。主节点的职责是接收需要在集群上执行的命令，以及编排自身的资源。每个从节点上都安装了Docker，以及一个kubelet服务，kubelet用于管理每个节点上的pod（一组容器）。集群的信息交由etcd维护，etcd是一个分布式的键值数据存储（见技巧66），它是集群中信息的真实来源。

什么是pod 我们将在本技巧的稍后部分再探讨这一块，所以现在

不必有过多疑问，只需要知道pod是一组相关的容器即可。这个概念是为了便于Docker容器的管理和维护。

Kubernetes的最终目标是让系统以简单可扩展的方式来运行用户容器，用户只需要声明他需要什么，让Kubernetes确保集群满足用户的需求。在本技巧中，读者将可以看到如何通过执行一条命令将一个简单的服务配置到指定的规模。

Kubernetes的起源 Kubernetes起初由Google公司开发，作为一种大规模管理容器的手段。Google公司大规模运行容器已达10年之久，在Docker开始流行时即决定开发这个容器编排系统。Kubernetes建立在Google大量的使用经验之上。Kubernetes也被简称为k8s。

关于Kubernetes安装、设置以及功能的详细介绍是一个很大且快速变化的话题，已超出本书的讨论范围（毫无疑问，不久之后将会有一本专门的书来介绍它）。这里我们将专注于Kubernetes的核心概念，并设置一个简单的服务以便读者能够对它有一个简单认知。

1. 安装Kubernetes

可以直接在宿主机上安装Kubernetes，从而得到一个单从节点的集群，也可以使用Vagrant来安装一个由虚拟机管理的多从节点集群。

要在主机上安装一个单从节点集群，可以运行以下命令：

```
export KUBERNETES_PROVIDER=vagrant  
curl -sS https://get.k8s.io | bash
```

获得最新的安装指南 在本书印刷时这种安装方式是正确的。访问GitHub上的文档链接可以获取Kubernetes的最新安装方式：
<http://mng.bz/62ZH>。

如果想安装多从节点集群，还有另一个选择。可以按照Kubernetes GitHub仓库（如前面注释所述）上对Vagrant的说明，或者尝试我们维护的一个自动化脚本来设置两个从节点的集群（<https://github.com/docker-in-practice/shutit-kubernetes-vagrant>）。

如果已经安装了Kubernetes，那么不妨跟着继续。接下来的内容将是基于一个多节点的集群。我们将创建一个容器并使用Kubernetes对它进行扩展。

2. 扩展单个容器

用于管理Kubernetes的命令行工具叫kubect1。在下面例子中会使用它的run-container子命令，用一个给定的镜像作为容器运行在一个pod中：

```
$ kubect1 run-container todo --image=dockerinpractice/todo (1)
$ kubect1 get pods | egrep "(POD|todo)" (2)
POD          IP    CONTAINER(S)  IMAGE(S)  HOST
└─ LABELS                STATUS    CREATED      MESSAGE
todo-hmj8e                10.245.1.3/ (3)
└─ run-container=todo Pending About a minute (4)
```

(1) todo是产出的pod的名称，可以通过--image标志来指定想要的镜像；这里我们用的todo镜像就是第1章里的那个

(2) kuberctl的get pods子命令列出所有的pod。我们只对todo

pod感兴趣，所以使用grep过滤出这个pod

(3) todo-hmj8e是pod名

(4) LABELS是与pod有关的键值对，如这里显示的run-container标签。pod的状态是Pending（挂起），这说明Kubernetes正在准备运行这个pod，很可能是正在从Docker Hub下载镜像

Kubernetes生成pod名称是根据run-container命令中的名称（上面的例子是todo）加上破折号，然后再加上一个随机的字符串。这确保了不会和其他的pod重名。

等待几分钟下载todo镜像后，最终会看到状态变为Running（运行中）：

```
$ kubectl get pods | egrep "(POD|todo)"
POD          IP           CONTAINER(S)  IMAGE(S)
└─ HOST      LABELS        STATUS        CREATED        MESSAG
E
todo-hmj8e   10.246.1.3
└─ 10.245.1.3/10.245.1.3  run-container=todo  Running        4 minutes
                                todo              dockerinpractice/todo
                                                Running        About a minute
```

这次IP、CONTAINER(S)和IMAGE(S)列都有值。IP列是pod的地址（这个例子中是10.246.1.3），CONTAINER(S)列中每一行包含了pod中的一个容器（这个例子中只有一个容器，即todo）。可以直接访问该IP地址和端口来测试todo容器确实已经在运行并且提供服务处理请求：

```
$ wget -qO- 10.246.1.3:8000
<html manifest="/todo.appcache">
[...]
```

现在我们还没看到这与直接运行一个Docker容器有什么区别。为

了首次尝试Kubernetes，可以运行Kubernetes的**resize**命令来扩展该服务：

```
$ kubectl resize --replicas=3 replicationController todo
resized
```

这一命令告诉Kubernetes我们想要todo的复制控制器（**replication controller**），以确保有3个todo应用程序实例运行在集群中。

什么是复制控制器 复制控制器是一个Kubernetes服务，用来确保存在正确数量的pod节点运行在集群中。

可以使用**kubectl get pods**命令来检查todo应用程序的已经启动的额外实例：

```
$ kubectl get pods | egrep "(POD|todo)"
POD          IP           CONTAINER(S)  IMAGE(S)
➤ HOST      LABELS       STATUS        CREATED      MESSA
GE
todo-2ip3n   10.246.2.2
➤ 10.245.1.4/10.245.1.4  run-container=todo  Running  10 minutes
  todo          dockerinpractice/todo
➤                                     Running  8 minutes
todo-4os5b   10.246.1.3
➤ 10.245.1.3/10.245.1.3  run-container=todo  Running  2 minutes
  todo          dockerinpractice/todo
➤                                     Running  48 seconds
todo-cuggp   10.246.2.3
➤ 10.245.1.4/10.245.1.4  run-container=todo  Running  2 minutes
  todo          dockerinpractice/todo
➤                                     Running  2 minutes
```

Kubernetes已经获得了**resize**指令和todo复制控制器，并确保启动了正确数目的pod。注意，有两个pod在同一个宿主机上（10.245.1.4），有一个pod在另一个宿主机上（10.245.1.3）。这是因为Kubernetes的默认调度程序有一个算法会默认跨节点散布pod。

什么是调度程序 调度程序是一个软件，它决定一些工作负载应该在哪里以及什么时候运行。例如，Linux内核便有一个调度程序，它会决定下一步应该运行什么任务。调度程序可以有非常简单的，也有超级复杂的。

读者已经看到了 Kubernetes使跨宿主机管理容器更加容易的方法。接下来我们将深入了解Kubernetes的pod概念。

3. 使用pod

pod是一组容器，它们被设计成以某种方式在一起工作并共享资源。

每个pod拥有自己的IP地址并共享相同的卷和网络端口段。因为一个pod的所有容器共享一台本地主机，所以只要它们被部署了，依赖的不同服务都是可用和相互可见的。

图9-7用两个容器共享一个卷来演示了这一点。在该图中，容器1是一个Web服务器，从共享卷中读取数据，而容器2则会更新数据。因此两个容器都是无状态的。状态存储在在共享卷中。

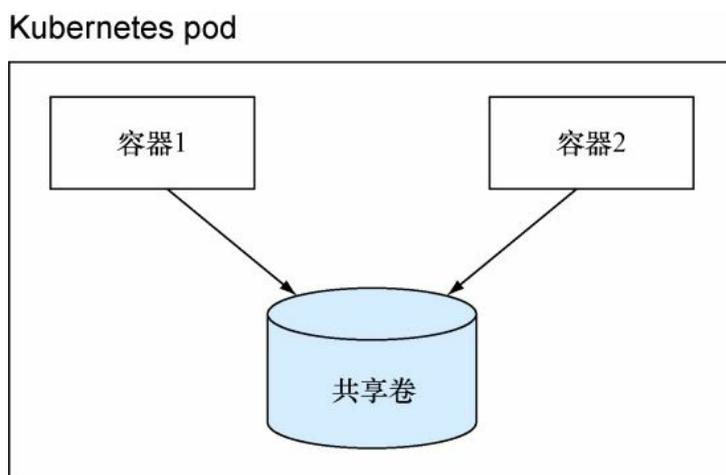


图9-7 拥有两个容器的pod

这种责任分离的设计通过单独管理服务的每个部分实现了微服务的方式。升级一个镜像不必担心会影响其他镜像。

代码清单9-5中的pod规范定义了一个复杂的pod，其拥有两个容器，一个容器会每5秒在文件中写入随机数据（`simple-writer`），另一个容器会从同一个文件中读取数据。文件通过卷（`pod-disk`）共享。

代码清单9-5 complexpod.json

```
{
  "id": "complexpod", (1)
  "kind": "Pod", (2)
  "apiVersion": "v1beta1", (3)
  "desiredState": {
    "manifest": { (4)
      "version": "v1beta1",
      "id": "complexpod",
      "containers": [{ (5)
        "name": "simplereader",
        "image": "dockerinpractice/simplereader", (6)
        "volumeMounts": [{ (7)
          "mountPath": "/data", (8)
          "name": "pod-disk" (9)
        }]
      }]
    }, {
      "name": "simplewriter",
      "image": "dockerinpractice/simplewriter",
      "volumeMounts": [{
        "mountPath": "/data",
        "name": "pod-disk"
      }]
    }
  ],
  "volumes": [{ (10)
    "name": "pod-disk", (11)
    "emptydir": {} (12)
  }]
}
```

- (1) `id`属性给该实体一个名称
- (2) `kind`属性指定了这是什么类型的对象
- (3) `apiVersion`属性指定了Kubernetes使用的JSON目标版本
- (4) `pod`规范的内容位于`desired-State`和`manifest`属性中
- (5) `pod`中容器的细节存储在JSON数组中
- (6) 每个容器都有一个名称用于引用，`Docker`镜像被定义在`image`属性中
- (7) 为每个容器都指定了卷的挂载点
- (8) `mountPath`是卷挂载到容器文件系统的路径。每个容器可以使用不同的位置
- (9) 卷挂载名称引用了`pod manifest`的卷定义中的名称
- (10) `volumes`属性定义了为这个`pod`创建的卷名称
- (11) 卷名称在前面`volumeMounts`项中被引用
- (12) 一个临时目录，用来共享`pod`的生命周期

创建一个包含上述配置的文件，运行如下命令来加载`pod`规范：

```
$ kubectl create -f complexpod.json
pods/complexpod
```

等待一会儿下载完镜像后，通过运行`kubectl log`命令并指定第一个`pod`和感兴趣的容器来查看容器的日志输出：

```
$ kubectl log complexpod simplereader
2015-08-04T21:03:36.535014550Z '? U
[2015-08-04T21:03:41.537370907Z] h(^3eSk4y
[2015-08-04T21:03:41.537370907Z] CM(@
[2015-08-04T21:03:46.542871125Z] qm>5
[2015-08-04T21:03:46.542871125Z] {Vv_
[2015-08-04T21:03:51.552111956Z] KH+74    f
[2015-08-04T21:03:56.556372427Z] j?p+!\
```

4. 接下来做什么

这里我们只是接触了Kubernetes的冰山一角，但这已经让读者清楚它能干什么以及它是怎样让编排容器变得更加简单的。在后面的OpenShift部分读者会再次看到Kubernetes，OpenShift是一个使用Kubernetes作为其编排引擎的应用程序平台即服务（见技巧87）。

技巧80 在Mesos上构建框架

当讨论众多的编排可能性时，读者可能会发现一个被特别提及作为Kubernetes的替代品的工具——Mesos。人们通常会这样描述Mesos，如“Mesos是框架的框架”，以及“Kubernetes可以运行在Mesos之上”！

我们遇到的最恰当的类比是将Mesos看作数据中心的内核。如果单独使用它办不成什么有价值的事情，将它和一个init系统还有应用程序组合在一起时就价值了！

一个通俗的解释是，想象有一只猴子坐在面板前面控制所有的机器，并有权随意启动和停止应用程序。当然，你需要给猴子一个非常清楚的指示，在特定情况下要做什么，何时启动应用程序等。你可以自己做，但是很花时间，而猴子的劳动力比较廉价。

Mesos就是这只猴子！

Mesos是具有高度动态化和复杂基础设施的企业的理想选择，这些企业拥有自己的生产环境编排方案的经验。如果不满足这些条件的话，比起花时间量身定制Mesos，现成的解决方案可能会服务得更好一些。

问题

有很多控制应用程序和作业启动的规则，想要无须手动就能在远程机器上启动它们并跟踪其状态。

解决方案

使用Apache Mesos和一个量身定制的框架。

讨论

Mesos是一款成熟的软件，用于在多台机器上提供资源管理的抽象。一些有所耳闻的公司已经将它们部署到了生产环境并且历经考验，结果证明，它是稳定和可靠的。

要求**Docker 1.6.2+** 本技巧要求Docker 1.6.2或更高版本，这样Mesos才能使用正确的Docker API版本。

图9-8展示了一个通用的生产环境Mesos配置。

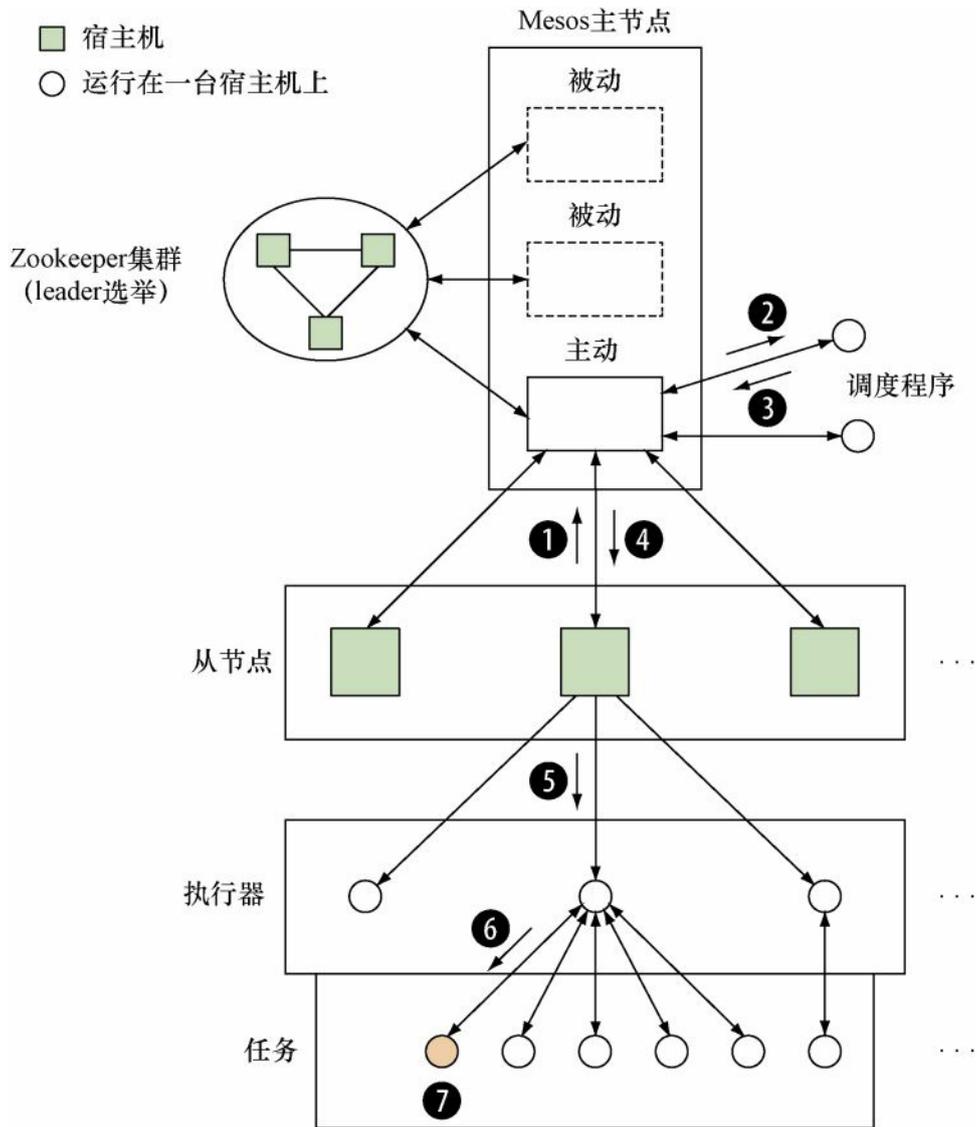


图9-8 通用的生产环境Mesos配置

参考图9-8可以看到Mesos启动一个任务的基本生命周期是怎样的。

(1) 一个从节点运行在节点上，追踪资源利用率并持续接收主节点的通知。

(2) 主节点从一个或多个从节点上收集可用的资源，并向调度程序提供资源。

(3) 调度程序接收主节点提供的资源，决定在哪里运行任务，

并将消息通知回主节点。

(4) 主节点将任务信息传递给适当的从节点。

(5) 每个从节点将任务信息传递给节点上现有的执行器，或启动一个新的执行器。

(6) 执行器读取任务信息并在节点上启动任务。

(7) 任务运行。

Mesos项目提供了主节点（**master**）和从节点（**slave**），还有内置的**shell**执行器。你的工作是提供一个框架（或应用程序），它是由一个调度程序（猴子例子中的指令列表）和可选的自定义执行器组成。

许多第三方项目提供了使用户可以运行在Mesos上的框架（我们将在下一个技巧中详细介绍），但为了更好地了解如何充分利用Mesos和Docker的力量，我们将构建自己的框架，该框架只包含一个调度程序。如果启动应用程序有非常复杂的逻辑，这可能也会是最终的途径。

与**Mesos**结合使用**Docker**的本质 与**Mesos**结合使用**Docker**并不是必须的，但由于这是本书的内容，我们会这样做。因为**Mesos**非常灵活，所以我们不会深入细节。我们也会在单台机器上运行**Mesos**，但我们会尝试尽可能地保持其真实性，并指出需要怎么做才能上线。

我们还没有解释 **Docker** 是如何适配 **Mesos** 的生命周期的——该谜题的最后部分便是**Mesos**提供了对容器化支持，允许用户隔离执

行器或任务（或两者同时隔离）。在这里，Docker并不是唯一可用的工具，但是它非常受欢迎，所以Mesos有一些特定于Docker的功能支持，可以让用户快速上手。

我们的示例只会容器化我们运行的任务，因为我们使用的是默认执行器。如果用户有自定义的执行器只能运行在一个特定的语言环境，每个任务需要动态加载和执行一些代码，那么可能需要考虑容器化执行器。作为一个示例用例，用户可能会把JVM用作执行器来实时加载和执行代码，从而避免当执行潜在的非常小的任务时JVM的启动开销。

图9-9展示了在创建新的Docker化任务时，我们当前示例里它的后台会发生什么。

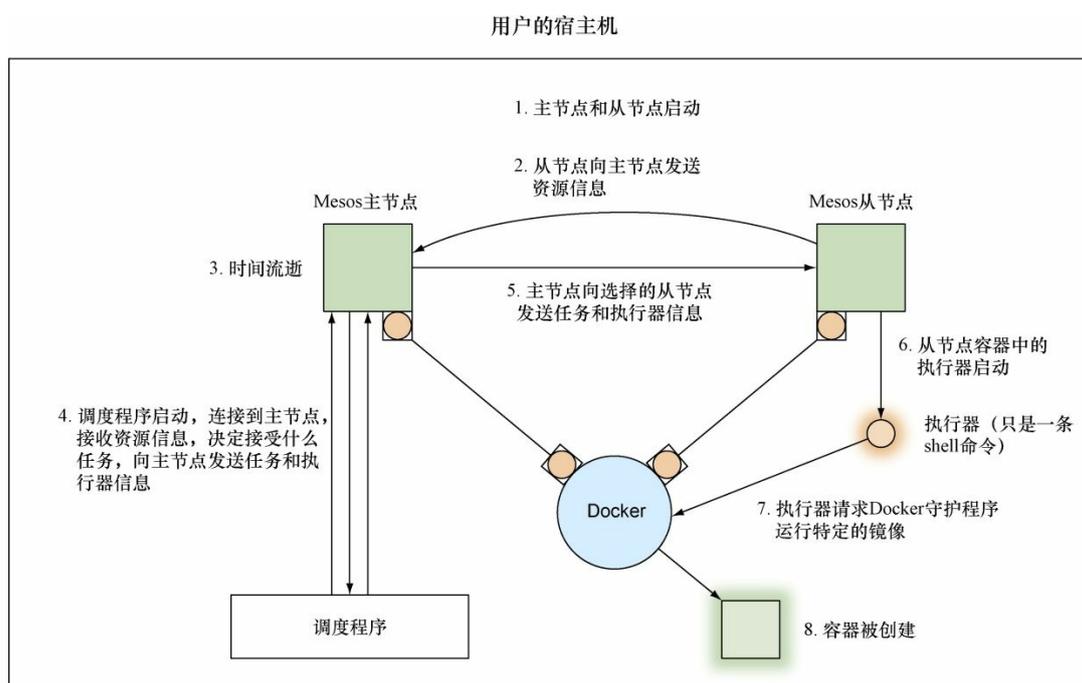


图9-9 单宿主机Mesos设置启动一个容器

话不多说，让我们开始吧！接下来首先要做的是通过代码清单9-

6启动一个主节点。

代码清单9-6 启动主节点

```
$ docker run -d --name mesmaster redjack/mesos:0.21.0 mesos-master \
--work_dir=/opt
24e277601260dcc6df35dc20a32a81f0336ae49531c46c2c8db84fe99ac1da35
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' mesmaster
172.17.0.2
$ docker logs -f mesmaster
I0312 01:43:59.182916    1 main.cpp:167] Build: 2014-11-22 05:29:57 by r
oot
I0312 01:43:59.183073    1 main.cpp:169] Version: 0.21.0
I0312 01:43:59.183084    1 main.cpp:172] Git tag: 0.21.0
[...]
```

主节点的启动有点儿冗长，但是用户应该能发现它很快就停止了日志记录。保持该终端开启，以便在启动其他容器时可以看到主节点发生了什么。

多主节点的**Mesos**设置 通常**Mesos**将会配置成具有多个**Mesos**主节点（一个归档和多个备份），以及一个**Zookeeper**集群。**Mesos**站点的“**Mesos High-Availability Mode**”（**Mesos**高可用模式）页面（<http://mesos.apache.org/documentation/latest/high-availability>）记录了如何配置多主节点的**Mesos**。用户还需要暴露端口5050用于外部通信，并且使用**work_dir**文件夹作为卷来保存持久化信息。

我们也需要从节点。但是这需要一些技巧。**Mesos**的定义特征之一就是能对执行任务有资源限制的能力，这要求从节点拥有自由检查和管理进程的能力。因此，运行从节点的命令需要将一些外部系统的细节暴露到容器内部，如代码清单9-7所示。

代码清单9-7 启动从节点

```
$ docker run -d --name messlave --pid=host \
```

```

-v /var/run/docker.sock:/var/run/docker.sock -v /sys:/sys \
redjack/mesos:0.21.0 mesos-slave \
--master=172.17.0.2:5050 --executor_registration_timeout=5mins \
--isolation=cgroups/cpu,cgroups/mem --containerizers=docker,mesos \
--resources="ports(*):[8000-8100]"
1b88c414527f63e24241691a96e3e3251fbb24996f3bfba3ebba91d7a541a9f5
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' messlave
172.17.0.3
$ docker logs -f messlave
I0312 01:46:43.341621 32398 main.cpp:142] Build: 2014-11-22 05:29:57 by
root
I0312 01:46:43.341789 32398 main.cpp:144] Version: 0.21.0
I0312 01:46:43.341795 32398 main.cpp:147] Git tag: 0.21.0
[...]
I0312 01:46:43.554498 32429 slave.cpp:627] No credentials provided.
➤ Attempting to register without authentication
I0312 01:46:43.554633 32429 slave.cpp:638] Detecting new master
I0312 01:46:44.419646 32424 slave.cpp:756] Registered with master
➤ master@172.17.0.2:5050; given slave ID 20150312-014359-33558956-5050
-1-S0
[...]

```

此刻，用户应该可以看到Mesos主节点终端的一些活动，开始几行如下所示：

```

I0312 01:46:44.332494 9 master.cpp:3068] Registering slave at
➤ slave(1)@172.17.0.3:5051 (8c6c63023050) with id
➤ 20150312-014359-33558956-5050-1-S0
I0312 01:46:44.333772 8 registrar.cpp:445] Applied 1 operations in
➤ 134310ns; attempting to update the 'registry'

```

这两行日志展示了用户已经启动了从节点并连接到了主节点。如果没有看到日志的话，不妨停下来再次检查配置的主节点的IP地址。当没有可连接的从节点时，尝试和调试为什么框架没有启动任何任务是非常令人沮丧的。

不管怎样，代码清单9-7中的命令做了很多事情。`run`之后和`redjack/mesos:0.21.0`之前的部分传入的参数都是Docker的参数，它们主要包含了给从节点容器传入的很多关于外部世界的信息。`mesos-slave`之后的参数更有意思。`master`告诉从节点在哪里

能找到主节点（或者Zookeeper集群）。接下来的3个参数`executor_registration_timeout`、`isolation`和`containerizers`都是对Mesos设置的调整，使用Docker时要始终设置这3个参数。最后，也是相当重要的一点是，需要让Mesos从节点知道哪些端口可以作为资源交出来。默认情况下，Mesos提供31000~32000，但我们想使用更小的更容易记忆的端口。

现在简单的步骤都已经完成，我们进行到了设置Mesos的最后阶段——创建调度程序。

幸好我们已经有了一个示例框架可以使用。我们来试试它能做什么，然后探索它的工作原理。用户不妨在主节点容器和从节点容器中保持`docker logs -f`命令窗口是打开的，以便可以看到通信。

代码清单9-8中给出的命令将从GitHub获取示例框架的源代码库并启动它。

代码清单9-8 下载和启动示例框架

```
$ git clone https://github.com/docker-in-practice/mesos-nc.git
$ docker run -it --rm -v $(pwd)/mesos-nc:/opt redjack/mesos:0.21.0 bash
# apt-get update && apt-get install -y python
# cd /opt
# export PYTHONUSERBASE=/usr/local
# python myframework.py 172.17.0.2:5050
I0312 02:11:07.642227 182 sched.cpp:137] Version: 0.21.0
I0312 02:11:07.645598 176 sched.cpp:234] New master detected at
➤ master@172.17.0.2:5050
I0312 02:11:07.645800 176 sched.cpp:242] No credentials provided.
➤ Attempting to register without authentication
I0312 02:11:07.648449 176 sched.cpp:408] Framework registered with
➤ 20150312-014359-33558956-5050-1-0000
Registered with framework ID 20150312-014359-33558956-5050-1-0000
Received offer 20150312-014359-33558956-5050-1-00. cpus: 4.0, mem: 6686
.0,
➤ ports: 8000-8100
Creating task 0
Task 0 is in state TASK_RUNNING
```

```
[...]
Received offer 20150312-014359-33558956-5050-1-05. cpus: 3.5, mem: 6586
.0,
➡ ports: 8005-8100
Creating task 5
Task 5 is in state TASK_RUNNING
Received offer 20150312-014359-33558956-5050-1-06. cpus: 3.4, mem: 6566
.0,
➡ ports: 8006-8100
Declining offer
```

读者可能会注意到，我们已经把Git仓库挂载到了Mesos镜像中。这是因为它包含了我们需要的所有Mesos库。不然，安装它们会是一个很痛苦的过程。

`mesos-nc`框架被设计用来在所有可用的宿主机上的8000和8005之间的所有可用端口之间运行`echo 'hello <task id>' | nc -l <port>`命令。由于netcat的工作原理，这些“服务器”在用户访问它们时就会终止，无论是通过curl、Telnet、nc还是浏览器来访问。可以在新的终端中运行`curl localhost:8003`来进行验证。它将会返回预期的响应，并且Mesos的日志会显示新产生的任务替代了被终止的那个。用户还可以使用`docker ps`来跟踪哪些任务正在运行。

值得一提的是，这里有Mesos保持跟踪已分配的资源并在任务终止时将它标记为可用的证据。特别是，当访问`localhost:8003`（不妨再次尝试下）时，请仔细查看收到的日志行，它展示了两个端口范围（因为它们没有被连接），包括刚刚释放的端口范围：

```
Received offer 20150312-014359-33558956-5050-1-045. cpus: 3.5, mem: 6586.0,
➡ ports: 8006-8100,8003-8003
```

Mesos从节点命名冲突 Mesos从节点给所有容器的命名均是以mesos-开头的，并且它假定这样任意的名称可以安全地被从节点管

理。留心容器的命名，否则Mesos从节点可能会杀掉它。

框架代码（myframework.py）加上了很好的注释，以使用户能够读懂。我们将介绍一些宏观的设计：

```
class TestScheduler(mesos.interface.Scheduler):
[...]
```

```
    def registered(self, driver, frameworkId, masterInfo):
[...]
```

```
    def statusUpdate(self, driver, update):
[...]
```

```
    def resourceOffers(self, driver, offers):
[...]
```

所有的Mesos调度程序都是Mesos调度程序类的子类，并且实现了一些方法，Mesos将会在适当的时间点通知用户运行的框架，使其对相应的事件做出响应。尽管上面的代码段中已经实现了3个方法，但其中的两个是可选的，它们的实现是用来添加额外的日志来进行演示的。唯一一个用户必须实现的方法是resourceOffers——框架在启动任务时没有太多不清楚的点。用户可以为了自己的目的添加任意额外的方法，如init和_makeTask，只要它们不和Mesos期望使用的任何方法冲突就好，因此，请读者先确保阅读过相关文档（<http://mesos.apache.org/documentation/latest/app-framework-development-guide/>）。

构建自己的框架？ 如果用户最终选择编写自己的框架，那么需要查看一些方法和结构的文档。但是在编写本书时，官方唯一产出的文档是针对Java方法的。想要找到深入结构的起点的读者，可以从查看Mesos源代码中的include/mesos/mesos.proto文件开始。祝你好运！

我们来看一下main方法中一个有意思的部分——resourceOffers的更多细节。它会决定启动任务还是拒绝任务。图9-10展示了Mesos调用了框架的resourceOffers方法后的执行流程（通常是因为一些资源已经可供框架使用）。

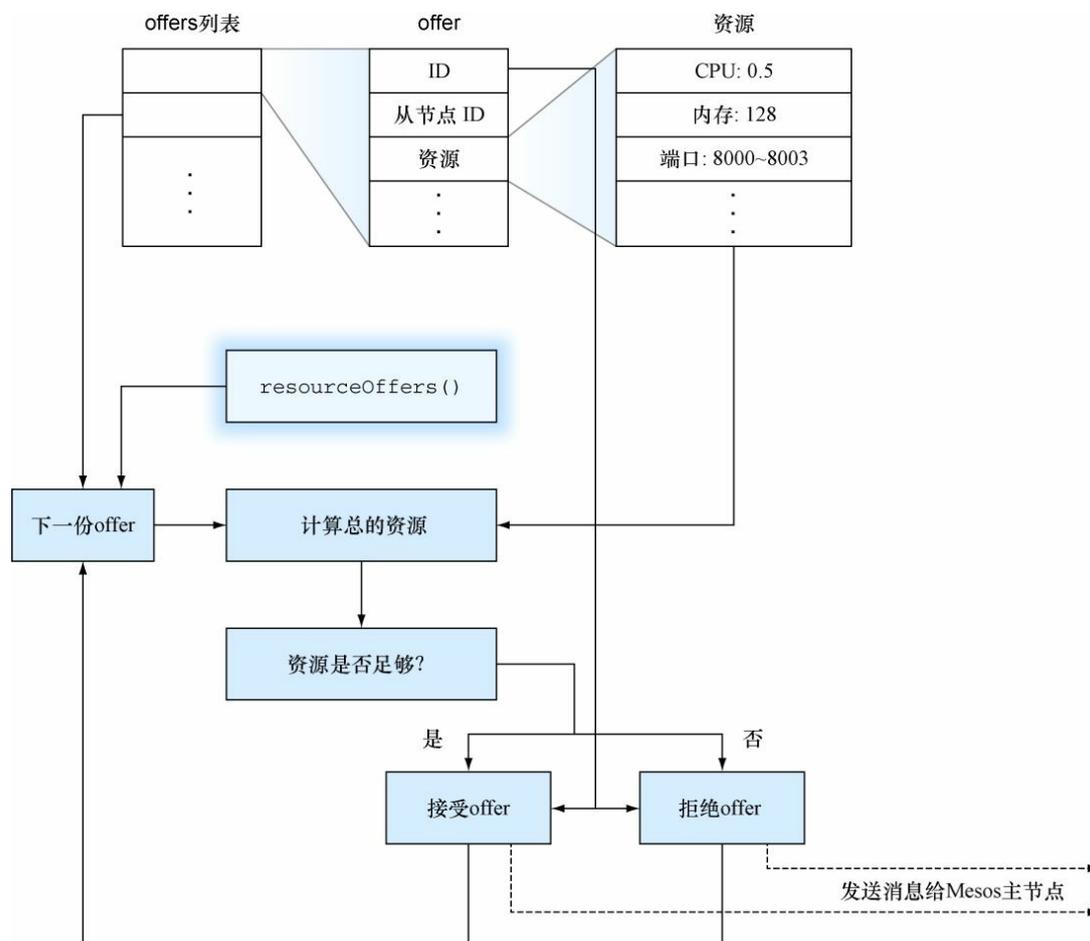


图9-10 调用resourceOffers时框架的执行流程

`resourceOffers`会接收一个offer列表，每个offer对应单个Mesos从节点。该offer包含了在从节点上运行的任务的可供使用的资源的细节，而该方法的一个典型实现将会使用该信息来识别最适合的位置来启动想要运行的任务。启动任务会给Mesos主节点发送消息，主节点会继续图9-9中列出的生命周期。

重要的是注意这里的灵活性——任务的启动取决于选定的任意标

准，从外部服务的健康检查乃至月相都行！这种灵活性可能是一种负担，所以业内现有的已发布的框架会隐藏这一底层的细节并简化对Mesos的使用。接下来的技巧会讲述其中一个框架。

读者可以阅读Roger Ignazio的《Mesos in Action》来了解Mesos的功能的更多细节——这里只进行了一些简单的介绍，而我们已经看到了它和Docker的配合是多么轻松。

技巧81 使用Marathon细粒度管理Mesos

现在读者应该已经意识到了，使用Mesos需要考虑很多细节，即便是对一个极其简单的框架也是如此。能够信赖被正确部署的应用程序这一点非常重要——框架中的bug造成的影响可能会导致部署新应用程序失败，也可能导致整个服务中断。

随着集群规模的扩展，风险也在上升，除非团队擅长于编写可靠的动态部署代码，否则可能需要考虑更多经过验证的方法——Mesos自身是很稳定的，但内部定制的框架可能不是人们想象的那么可靠。

Marathon适用于那些没有内部部署工具开发经验但需要一个良好支持而且易于使用的方案，以便在有些动态的环境中部署容器的公司。

问题

需要一种可靠的方式来利用Mesos的力量，而不会陷入编写自己的框架的困扰。

解决方案

使用Marathon。

讨论

Marathon是一款Apache Mesos框架，它是由Mesosphere构建的、用于管理长期运行的应用程序。市场资料将其描述为数据中心（Mesos是其核心）的init或upstart守护进程。这个比喻并非没有道理。

Marathon可以让用户启动一个包含了Mesos主节点、Mesos从节点和Marathon自身的容器来作为简单的快速上手。这对于演示很有用，但不适用于生产环境下的Marathon部署。要配置一个真实环境的Marathon，用户需要一个Mesos主节点和从节点（来自之前的技巧）以及一个Zookeeper实例（出自技巧77）。确保这些都在运行之后，我们将开始运行Marathon容器：

```
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' mesmaster
172.17.0.2
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' messlave
172.17.0.3
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' zookeeper
172.17.0.4
$ docker pull mesosphere/marathon:v0.8.2
[...]
$ docker run -d -h $(hostname) --name marathon -p 8080:8080 \
mesosphere/marathon:v0.8.2 --master 172.17.0.2:5050 --local_port_min 80
00 \
--local_port_max 8100 --zk zk://172.17.0.4:2181/marathon
accd6de46cfab65572539ccffa5c2303009be7ec7dbfb49e3ab8f447453f2b93
$ docker logs -f marathon
MESOS_NATIVE_JAVA_LIBRARY is not set. Searching in /usr/lib /usr/local/
lib.
MESOS_NATIVE_LIBRARY, MESOS_NATIVE_JAVA_LIBRARY set to
➤ '/usr/lib/libmesos.so'
[2015-06-23 19:42:14,836] INFO Starting Marathon 0.8.2
➤ (mesosphere.marathon.Main$:87)
[2015-06-23 19:42:16,270] INFO Connecting to Zookeeper...
➤ (mesosphere.marathon.Main$:37)
[...]
[2015-06-30 18:20:07,971] INFO started processing 1 offers,
➤ launching at most 1 tasks per offer and 1000 tasks in total
➤ (mesosphere.marathon.tasks.IterativeOfferMatcher$:124)
[2015-06-30 18:20:07,972] INFO Launched 0 tasks on 0 offers,
```

↳ declining 1 (mesosphere.marathon.tasks.IterativeOfferMatcher\$:216)

就像Mesos一样，Marathon非常啰嗦，不过（也像Mesos）它也会很快停下来。此刻，我们将从编写自己框架那部分内容进入一个很熟悉的环节——考虑资源供应并决定如何用这些资源做些什么。因为还没有启动任何东西，所以自从前面的日志的**declining 1**后我们便看不到任何活动。

Marathon有一个漂亮的Web界面，这也是要在宿主机上暴露8080端口——在浏览器中访问<http://localhost:8080>端口来打开页面的原因。

我们直接切换到Marathon的具体操作部分，先创建一个新的应用程序。这里有一些术语要澄清一下——在Marathon的世界里“应用程序”（app）是拥有完全相同定义的一个或多个任务的集合。

点击右上角的App New（新建应用程序）按钮，会弹出一个对话框，可以用它来定义要启动的应用程序。我们将继续沿用自己创建的框架，设置ID为marathon-nc，设置CPU、内存和磁盘空间为默认值（以符合mesos-nc框架的资源限制），并且设置启动命令为echo "hello \$MESOS_TASK_ID" | nc -l \$PORT0（使用该任务可用的环境变量，注意就是数字0）。将端口字段值设置为8000，指定我们想要监听的位置。随即，跳过其他的字段设置。点击Create（创建）。

用户新定义的应用程序现在将显示在Web界面上。状态会先简要显示为Deploying，然后变为Running。应用程序现在已经启动了！

如果点击应用程序列表中的/marathon-nc条目，将会看到该应用程序的唯一ID。通过REST API可以得到完整的配置，如下面的代码

所示，也可以通过Mesos从节点容器对应的端口运行curl命令来验证它在运行。用户需要确保保存了REST API返回的完整的配置，因为稍后会派上用场——它被保存在下面例子中的app.json中：

```
$ curl http://localhost:8080/v2/apps/marathon-nc/versions
{"versions":["2015-06-30T19:52:44.649Z"]}
$ curl -s \
http://localhost:8080/v2/apps/marathon-nc/versions/2015-06-30T19:52:44.
649Z \
> app.json
$ cat app.json
{"id":"/marathon-nc",
  "cmd":"echo \"hello $MESOS_TASK_ID\" | nc -l $PORT0",[...]
$ curl http://172.17.0.3:8000
hello marathon-nc.f56f140e-19e9-11e5-a44d-0242ac110012
```

留意一下对应用程序执行curl命令的输出结果中hello后面的文本——它应该和界面中的唯一ID是匹配的。检查要快速，因为运行curl命令会终止该应用程序，Marathon会重新启动它，界面中的唯一ID会改变。一旦验证了这些，继续点击Destroy App（销毁应用程序）按钮来删除marathon-nc。

一切工作正常，但是读者可能已经注意到，我们没有达成使用Marathon的目的——编排Docker容器。尽管应用程序在容器中，但它在Mesos从节点容器中启动，而不是在自己的容器中启动。阅读Marathon文档说明，在Docker容器中创建任务还需要做更多的配置（就像编写自己的框架时一样）。

幸好，之前启动的Mesos从节点都有所需的设置，所以只需修改一些Marathon选项——特别是应用程序方面的选项。通过获取之前Marathon API的响应信息（存放在app.json中），我们可以专注于添加Marathon的设置信息从而启用Docker。我们将使用jq工具执行操作，尽管通过文本编辑器来做也同样简单：

```

$ JQ=https://github.com/stedolan/jq/releases/download/jq-1.3/jq-linux-x86_64
$ curl -Os $JQ && mv jq-linux-x86_64 jq && chmod +x jq
$ cat >container.json <<EOF
{
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "ubuntu:14.04.2",
      "network": "BRIDGE",
      "portMappings": [{"hostPort": 8000, "containerPort": 8000}]
    }
  }
}
$ # merge the app and container details
$ cat app.json container.json | ./jq -s add > newapp.json

```

现在，我们可以将新的应用程序定义发送给API然后见证 Marathon启动它：

```

$ curl -X POST -H 'Content-Type: application/json; charset=utf-8' \
--data-binary @newapp.json http://localhost:8080/v2/apps
{"id":"/marathon-nc",
  ▶ "cmd":"echo \"hello $MESOS_TASK_ID\" | nc -l $PORT0",[...]
$ sleep 10
$ docker ps --since=marathon
CONTAINER ID   IMAGE          COMMAND                  CREATED
  ▶ STATUS      PORTS          NAMES
284ced88246c  ubuntu:14.04  "\"/bin/sh -c 'echo About a minute ago
  ▶ Up About a minute 0.0.0.0:8000->8000/tcp mesos-
  ▶ 1da85151-59c0-4469-9c50-2bfc34f1a987
$ curl localhost:8000
hello mesos-nc.675b2dc9-1f88-11e5-bc4d-0242ac11000e
$ docker ps --since=marathon
CONTAINER ID   IMAGE          COMMAND                  CREATED
  ▶ STATUS      PORTS          NAMES
851279a9292f  ubuntu:14.04  "\"/bin/sh -c 'echo 44 seconds ago
  ▶ Up 43 seconds    0.0.0.0:8000->8000/tcp mesos-
  ▶ 37d84e5e-3908-405b-aa04-9524b59ba4f6
284ced88246c  ubuntu:14.04  "\"/bin/sh -c 'echo 24 minutes ago
  ▶ Exited (0) 45 seconds ago
  ▶ mesos-1da85151-59c0-4469-9c50-2bfc34f1a987

```

和我们的自定义框架一样，Mesos 已经启动了一个Docker容器，应用程序就运行在里面。运行curl命令会终止应用程序和容器，然后自动启动一个新的。

这些框架之间有一些显著的差异。例如，在一个自定义框架里，我们可以针对资源供给的接受进行非常细粒度的控制，我们可以选择和征用单个要监听的端口。为了在Marathon中也能做类似的事情，则需要给每一个从节点强加一些额外的设置。

相比之下，Marathon拥有很多内置的功能，包括健康检查、事件通知系统和REST API。这并不是微不足道的实现细节，使用Marathon可以确保的一点是在操作它时你并不是第一个吃螃蟹的人。如果没别的需求，获取Marathon的支持比定制框架要容易得多。我们发现Marathon的文档比Mesos的更加通俗易懂。

我们已经介绍了设置和使用Marathon的一些基础知识，但是这里面还有更多事情要做。我们看到的更有趣的一个建议便是使用Marathon启动其他的Mesos框架，可能包括你自己的定制框架！我们鼓励读者去积极探索——Mesos是一个专注于编排领域的高品质工具，而Marathon在其上提供了一个可用的应用层。

9.3 服务发现：我们有什么

本节将介绍的服务发现即是编排的另一面。能够将应用程序部署到数百台不同的机器当然很棒，但如果不知道哪些应用程序位于哪里，就无法实际使用它们。

虽然服务发现领域不如编排领域那么饱和，但是这一领域也有一些竞争对手。不过，他们的功能集只是略有不同。

服务发现通常需要两个功能：一个通用的键值存储和通过一些方便的接口（就像DNS）检索服务终端的方法。etcd和Zookeeper是前者的例子，而SkyDNS（这一工具我们将不会有所涉及）就是后者的

例子。事实上，SkyDNS正是使用etcd来存储所需信息的。

技巧82 使用Consul来发现服务

etcd是一款非常流行的工具，但它有一个特别的竞争者，谈论它时总会被提及，那就是Consul。这有点儿奇怪，因为业内有其他更像etcd的工具（Zookeeper具有与etcd相似的功能，但是用不同的语言实现的），而Consul通过一些有意思的附加功能与其区别开来，如服务发现和健康检查。

事实上如果仔细看，Consul有点儿像是etcd、SkyDNS、Nagios的整个打包。

问题

想要能分发消息给一组容器、在一组容器中发现服务以及监控一组容器。

解决方案

在每台Docker宿主机上启动带有Consul的容器，从而提供服务目录和配置通信系统。

讨论

当用户需要协同一些独立的服务时，Consul试图成为用于完成一些重要任务的通用工具。其他工具当然也可以完成这些任务，但Consul提供了统一的配置界面，这对用户而言非常方便。Consul从宏观来说提供了以下功能：

- 服务配置——用于存储和共享小值的键值存储，类似于etcd和Zookeeper；

- 服务发现——用于注册服务的API和用于发现服务的DNS端点，就像SkyDNS；
- 服务监控——用于注册健康检查的API，就像Nagios。

用户可以使用这里面的全部或者部分功能，因为它们之间没有任何联系。如果用户已经有了监控应用服务的基础设施，则无须替换成Consul。

本技巧会覆盖Consul的服务发现和服务监控部分，但是不包括键值存储。在熟读了Consul的文档之后，etcd和Consul之间的强相似性也使第7章的最后两个技巧是可以相互转换的。

图9-11展示了典型的Consul设置。

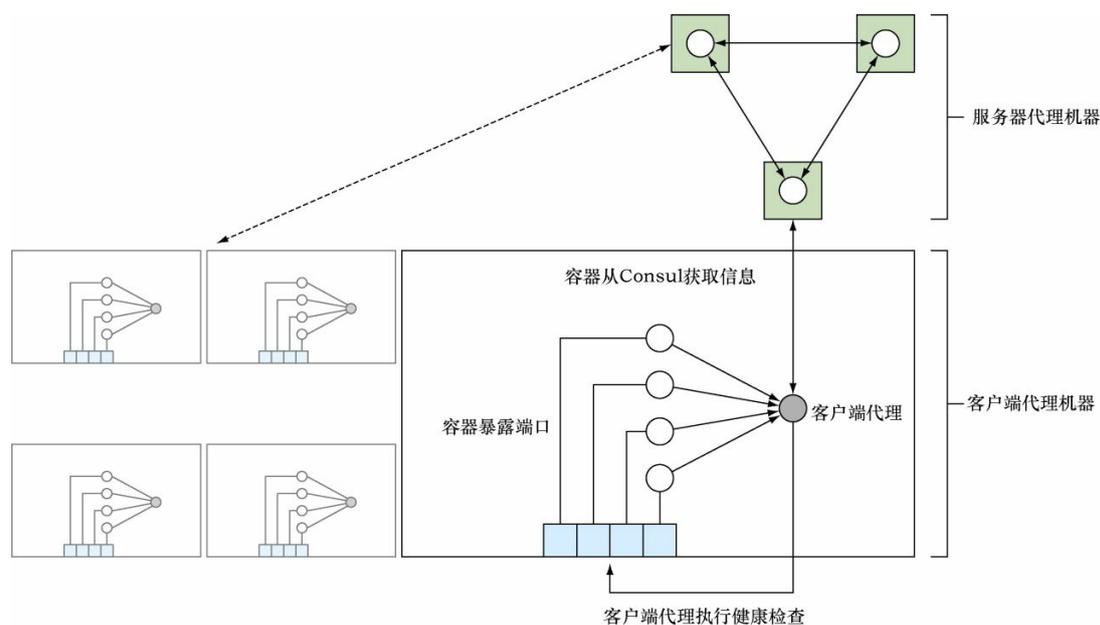


图9-11 一个典型的Consul设置

存储在 Consul 中的数据由服务器代理负责。它们负责对存储的信息达成共识——这个概念存在于大多数分布式数据存储系统中。简而言之，如果丢失了少于一半的服务器代理，该分布式系统将仍然能够确保数据是可恢复的（见技巧66中与etcd相关的示例）。因为这些

服务器代理很重要，而且有更高的资源要求，所以将其部署到专用的机器上是典型的选择。

保留数据 虽然本技巧中的命令会将Consul数据目录（/data）保存在容器中，但通常至少针对服务器而言，最好将此目录指定为卷，从而可以保留备份。

这里建议用户控制的所有可能要与Consul交互的机器都应该运行客户端代理。这些代理会将请求转发给服务器并运行健康检查。

让Consul运行的第一步是启动服务器代理：

```
c1 $ IMG=dockerinpractice/consul-server
c1 $ docker pull $IMG
[...]
c1 $ ip addr | grep 'inet ' | grep -v 'lo$|docker0$|vbox.*$'
    inet 192.168.1.87/24 brd 192.168.1.255 scope global wlan0
c1 $ EXTIP1=192.168.1.87
c1 $ echo '{"ports": {"dns": 53}}' > dns.json
c1 $ docker run -d --name consul --net host \
-v $(pwd)/dns.json:/config/dns.json $IMG -bind $EXTIP1 -client $EXTIP1 \
\
-recursor 8.8.8.8 -recursor 8.8.4.4 -bootstrap-expect 1
88d5cb48b8b1ef9ada754f97f024a9ba691279e1a863fa95fa196539555310c1
c1 $ docker logs consul
[...]
    Client Addr: 192.168.1.87 (HTTP: 8500, HTTPS: -1, DNS: 53, RPC: 8
400)
    Cluster Addr: 192.168.1.87 (LAN: 8301, WAN: 8302)
[...]
==> Log data will now stream in as it occurs:

    2015/08/14 12:35:41 [INFO] serf: EventMemberJoin: mylaptop 192.168.1
.87
[...]
    2015/08/14 12:35:43 [INFO] consul: member 'mylaptop' joined, marking
➡ health alive
    2015/08/14 12:35:43 [INFO] agent: Synced service 'consul'
```

由于我们想要把 Consul 当作一台 DNS 服务器使用，因此，我们将一个文件放到了 Consul 读取配置的文件夹里，使 Consul 监听 53 端口（DNS 协议的注册端口）。然后，我们使用在之前的技巧中学到的一个命令序列来尝试查找机器的面向外部的 IP 地址，以便与其他代理进行通信并监听客户端请求。

DNS 端口冲突 IP 地址 `0.0.0.0` 通常用于指示应用程序应该监听机器上的所有可用接口。我们故意没有这样做，因为一些 Linux 发行版有一个 DNS 缓存守护进程，监听 `127.0.0.1`，它不允许监听 `0.0.0.0:53`。

在前面的 `docker run` 命令里有以下 3 个注意事项。

- 使用了 `--net host`。虽然这可以被视为 Docker 世界的一个人造天地，但是另外一种选择是在命令行上暴露 8 个端口——这是个人偏好的问题，但是我们认为这是合理的。它还有助于绕过 UDP 通信的潜在问题。如果是手动设置的路由，则不需要设置 DNS 端口——可以将默认的 Consul DNS 端口（8600）作为端口 53 暴露在宿主机。
- 两个 `recursor` 参数告诉 Consul，如果 consul 本身不知道请求的地址，可以通过哪些 DNS 服务器查看。
- `-bootstrap-expect 1` 参数表明 Consul 集群启动运转之初只需要运行一个代理即可，这不是很健壮。一个典型的设置是将数量设置为 3（或更多），以确保直到所需数量的服务器已加入后，集群才会启动。要启动其他服务器代理，可以添加一个 `-join` 参数，我们将在启动客户端时讨论。

现在让我们配置第二台机器，启动客户端代理，并将其添加到集

群中。

这里有坑 由于Consul与其他代理通信时期望能够监听到一组特别的端口，这样一来在单个机器上配置多个代理用来演示现实世界中Consul的工作方式，就会有点儿困难。现在，我们将使用不同的宿主机——如果决定使用IP别名，一定要确保带上了 `-node newAgent` 参数，因为Consul默认会使用主机名，而这会引起冲突。

```
c2 $ IMG=dockerinpractice/consul-agent
c2 $ docker pull $IMG
[...]
c2 $ EXTIP1=192.168.1.87
c2 $ ip addr | grep docker0 | grep inet
    inet 172.17.42.1/16 scope global docker0
c2 $ BRIDGEIP=172.17.42.1
c2 $ ip addr | grep 'inet ' | grep -v 'lo$|docker0$'
    inet 192.168.1.80/24 brd 192.168.1.255 scope global wlan0
c2 $ EXTIP2=192.168.1.80
c2 $ echo '{"ports": {"dns": 53}}' > dns.json
c2 $ docker run -d --name consul-client --net host \
-v $(pwd)/dns.json:/config/dns.json $IMG -client $BRIDGEIP -bind $EXTIP
2 \
-join $EXTIP1 -recursor 8.8.8.8 -recursor 8.8.4.4
5454029b139cd28e8500922d1167286f7e4fb4b7220985ac932f8fd5b1cdef25
c2 $ docker logs consul-client
[...]
    2015/08/14 19:40:20 [INFO] serf: EventMemberJoin: mylaptop2 192.168.
1.80
[...]
    2015/08/14 13:24:37 [INFO] consul: adding server mylaptop
➡ (Addr: 192.168.1.87:8300) (DC: dc1)
```

反馈信息 我们使用的镜像是基于gliderlabs/consul-server:0.5和gliderlabs/consul-agent:0.5的，并且它们附带一个较新版本的Consul，以避免UDP通信的可能的的问题，这可以通过不断记录的日志行（如“Refuting a suspect message.”）获得提示。在镜像的0.6版本发布后，用户可以切换回gliderlabs中的镜像。

所有的客户端服务（HTTP、DNS等）都已配置为监听Docker网桥IP地址。这为容器提供了一个周知的位置来获取Consul的信息，而它只能在机器的内部公开Consul，这迫使其他机器直接访问服务器代理，而不是由客户端代理再到服务器代理这样一条更慢的路径。为了确保所有宿主机的网桥IP地址一致，可以查看Docker守护进程的**--bip**参数——在技巧70中配置Resolvable时可能会熟悉该指令。

跟之前一样，我们已经找到了外部的IP地址并且将集群的通信绑定到了上面。**-join**参数告诉Consul以哪里为起点寻找集群。用户无须担心集群信息的管理细节——当两个代理最初相遇时，它们会协同交流（**gossip**），传递在集群中发现的其他代理信息。最后的**-recursor**参数告诉Consul，在处理本意并非是查找已注册服务的DNS请求时应该使用上游的哪些DNS服务器。

我们来验证代理是否已经通过客户端机器上的HTTP API连接到服务器。我们将用到的API调用会返回客户端代理程序当前认为在集群中的成员列表（在很大的、快速变化的集群中，这可能并不总是与集群的成员相匹配——针对这一点，这里还有一个更慢的API调用可供使用）：

```
c2 $ curl -sSL $BRIDGEIP:8500/v1/agent/members | tr ',' '\n' | grep Name
e
[{"Name":"mylaptop2"}
{"Name":"mylaptop"}
```

现在 Consul 的基础设施已经建立起来了，是时候查看如何注册和发现服务。典型的注册过程是让应用程序初始化后对本地客户端代理进行API的调用，提醒客户端代理将信息分发给服务器代理。出于演示的目的，我们手动执行这一注册步骤：

```
c2 $ docker run -d --name files -p 8000:80 ubuntu:14.04.2 \
```

```
python3 -m http.server 80
96ee81148154a75bc5c8a83e3b3d11b73d738417974eed4e019b26027787e9d1
c2 $ docker inspect -f '{{.NetworkSettings.IPAddress}}' files
172.17.0.16
c2 $ /bin/echo -e 'GET / HTTP/1.0\r\n\r\n' | nc -i1 172.17.0.16 80 \
| head -n 1
HTTP/1.0 200 OK
c2 $ curl -X PUT --data-binary '{"Name": "files", "Port": 8000}' \
$BRIDGEIP:8500/v1/agent/service/register
c2 $ docker logs consul-client | tail -n 1
2015/08/15 03:44:30 [INFO] agent: Synced service 'files'
```

这里我们在容器中设置了一个简单的HTTP服务器，将其暴露在宿主机的8000端口，并检查其是否工作。然后使用curl和Consul的HTTP API来注册服务定义。这里唯一一件必不可少的事情便是给服务配一个名字——端口和Consul文档中列出的其他字段都是可选的。ID字段值得一提，它默认便是服务的名称，但在所有服务中必须是唯一的。如果想要同一个服务的多个实例，就需要指定它。

Consul的日志行告诉我们服务已经同步，所以我们应该可以通过服务的DNS接口来检索服务的信息。该信息来自服务器代理，所以它可以作为验证该服务已经被Consul目录接受的依据。可以使用dig命令查询服务DNS信息并检查它是否出现：

```
c2 $ EXTIP1=192.168.1.87
c2 $ dig @$EXTIP1 files.service.consul +short (1)
192.168.1.80
c2 $ BRIDGEIP=172.17.42.1
c2 $ dig @$BRIDGEIP files.service.consul +short (2)
192.168.1.80
c2 $ dig @$BRIDGEIP files.service.consul srv +short (3)
1 1 8000 mylaptop2.node.dc1.consul.
c2 $ docker run -it --dns $BRIDGEIP ubuntu:14.04.2 bash (4)
root@934e9c26bc7e:/# ping -c1 -q www.google.com (5)
PING www.google.com (216.58.210.4) 56(84) bytes of data.

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 25.358/25.358/25.358/0.000 ms
root@934e9c26bc7e:/# ping -c1 -q files.service.consul (6)
PING files.service.consul (192.168.1.80) 56(84) bytes of data.
```

```
--- files.service.consul ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.062/0.062/0.062/0.000 ms
```

(1) 通过服务端代理DNS查找files服务的IP地址。该DNS服务对于任何不在Consul集群的机器都是可用的，允许它们享用服务发现的好处

(2) 从客户端代理DNS查找files服务的IP地址。如果使用`\$BRIDGEIP`失败，可能希望尝试使用`\$EXTIP1`

(3) 从客户端代理DNS请求files服务的SRV记录。SRV记录是通过DNS交流服务信息的方式，包括协议，端口和其他条目。值得注意的两条是，用户可以在响应中看到端口号，并且用户已经被授予提供服务的机器的规范主机名而非IP地址

(4) 启动一个使用本地客户端代理作为唯一DNS服务器的容器。如果用户自己对之前提到的关于Resolvable的技巧（技巧70）还熟悉的话，不妨重新试下，将其设置为所有容器的默认值。别忘了要覆盖Consul代理的默认值，否则可能会导致意外的行为

(5) 验证外部地址的查找是否依然有效

(6) 验证服务查找是否在容器内仍然自动工作

Resolvable和Consul DNS服务的类似之处实在令人震惊。它们之间的关键区别在于Consul可以跨多个节点找到容器。然而，正如本技巧开始时提到的那样，Consul还有一个我们将会了解到的有趣特性，即健康检查。

健康检查是一个大的课题，如果想要了解细节请查看Consul的综

合文档，这里我们关注监控的其中一个选项，即脚本检查。配置该选项将会运行一个命令，并根据返回值设置健康状态，0代表成功，1代表警告，其他值代表致命。可以在初始化服务时就注册健康检查，也可以通过单独的API调用中进行注册，就像这里做的一样：

```
c2 $ cat >check <<'EOF' (1)
#!/bin/sh
set -o errexit
set -o pipefail
SVC_ID="$1"
SVC_PORT=\
"${wget -q0 - 172.17.42.1:8500/v1/agent/services | jq ".$SVC_ID.Port")"
wget -qs0 - "localhost:$SVC_PORT"
echo "Success!"
EOF
c2 $ cat check | docker exec -i consul-client sh -c \
'cat > /check && chmod +x /check' (2)
c2 $ cat >health.json <<'EOF' (3)
{
  "Name": "filescheck",
  "ServiceID": "files",
  "Script": "/check files",
  "Interval": "10s"
}
EOF
c2 $ curl -X PUT --data-binary @health.json \
172.17.42.1:8500/v1/agent/check/register (4)
c2 $ sleep 300 (5)
c2 $ curl -sSL 172.17.42.1:8500/v1/health/service/files | \
python -m json.tool | head -n 13 (6)
[
  {
    "Checks": [
      {
        "CheckID": "filescheck",
        "Name": "filescheck",
        "Node": "mylaptop2",
        "Notes": "",
        "Output": "/check: line 6: jq: not \
found\nConnecting to 172.17.42.1:8500 (172.17.42.1:8500)\n",
        "ServiceID": "files",
        "ServiceName": "files",
        "Status": "critical"
      }
    ]
  }
]
c2 $ dig @$BRIDGEIP files.service.consul srv +short (7)
c2 $
```

- (1) 创建检查脚本，验证服务中的HTTP状态代码是否为200 OK。将服务ID作为参数传递给脚本用来查找服务端口
- (2) 复制检查脚本到Consul代理容器
- (3) 创建一个健康检查的定义，发送给Consul HTTP API。在ServiceID字段和脚本命令行中指定服务ID
- (4) 提交健康检查JSON给Consul代理
- (5) 等待与服务器代理通信后的检查输出
- (6) 从注册的检查中获取健康检查信息
- (7) 试图查找files服务，结果为空

避免检查状态的分流 因为健康检查的输出在每次执行时会改变（例如，它包括时间戳），Consul只会在服务器状态更改或每5分钟（尽管这个时间间隔是可配置的）同步检查输出。由于状态开始时是critical，因此在这种情况下没有初始状态的更改，也就需要等待一个时间间隔才能获得输出。

我们为files服务添加了每10秒运行一次的健康检查，但是检查显示服务为critical状态。因此，Consul自动将失败的端点从DNS返回的条目中删除，我们也就没有服务器可用。这对于在生产环境中自动从多后端服务中删除服务器特别有用。

我们曾经遇到的一个错误的根本原因，也就是在容器内运行Consul时需要注意的一个很重要的点。所有检查也都在容器中运行，因此必须将检查脚本复制到容器中，还需要确保需要的任何命令都已

安装在容器中。在当前特定场景下，我们缺少jq命令（用于从JSON中提取信息的实用程序）。我们可以手动安装，但是在生产环境里正确的方法是在镜像中添加层：

```
c2 $ docker exec consul-client sh -c 'apk update && apk add jq'
fetch http://dl-4.alpinelinux.org/alpine/v3.2/main/x86_64/APKINDEX.tar.
gz
v3.2.3 [http://dl-4.alpinelinux.org/alpine/v3.2/main]
OK: 5289 distinct packages available
(1/1) Installing jq (1.4-r0)
Executing busybox-1.23.2-r0.trigger
OK: 14 MiB in 28 packages
c2 $ docker exec consul-client sh -c \
'wget -qO - 172.17.42.1:8500/v1/agent/services | jq ".files.Port"
8000
c2 $ sleep 15
c2 $ curl -sSL 172.17.42.1:8500/v1/health/service/files | \
python -m json.tool | head -n 13
[
  {
    "Checks": [
      {
        "CheckID": "filescheck",
        "Name": "filescheck",
        "Node": "mylaptop2",
        "Notes": "",
        "Output": "Success!\n",
        "ServiceID": "files",
        "ServiceName": "files",
        "Status": "passing"
      }
    ]
  }
],
```

我们现在已经借助Alpine Linux软件包管理器（见技巧51）将jq安装到了镜像，可以通过手动执行以前在脚本中失败的行来验证这一点，然后等待检查重新运行。现在成功了！

借助在前面的例子中用到的脚本来实施健康检查，现在用户拥有了一个重要的构建块来组建应用程序的监控——如果能够将健康检查表示为在终端中运行的一组命令，那么便可以让Consul自动运行它。如果你发现自己想要检查的状态代码可以由一个HTTP端口返回，那

么你是幸运的——这是一个普通任务的常见做法，Consul的3种类型的健康检查的其中一种就是专为它设计的。最后一种健康检查——“存活时间”，需要和应用程序进行更深入的集成。状态必须定期设置为健康的，否则检查将自动设置为失败。通过结合这3种类型的健康检查，用户可以在自己的系统之上构建一套全面的应用层健康状况的监控。

在本技巧最后，我们将看看服务器代理镜像附带的可选的Consul Web界面，它会帮助用户了解集群的当前状态。可以访问服务器代理外部IP地址的8500端口来打开界面。这里用户可以访问 `$EXTIP1:8500`。请记住，即使用户在服务器代理的宿主机上，`localhost`或`127.0.0.1`也将是无法正常工作的。

我们在本技巧中已经介绍了很多内容——Consul 是一个很大的话题！幸运的是，正如使用etcd一样，使用键值存储的知识可以转换到其他键值存储（如Consul），服务发现的相关知识也可以类比到其他提供DNS接口的工具（SkyDNS是可能遇到的一款）。我们所涉及的使用宿主机网络栈和使用外部IP地址的一些细节之处也是能够以此类推的。大多数容器化的分布式工具需要跨多个节点进行服务发现时都面临类似的问题。

技巧83 使用Registrar进行自动化服务注册

到目前为止，Consul（以及任何服务发现工具）最明显的缺点是必须管理服务条目的创建和删除。如果将它集成到应用程序中，将会存在多种实现方案以及多个可能出错的地方。

对于没有完全控制权的应用程序，集成也不起作用，因此在启动数据库之类的应用时最终还得编写包装脚本。

问题

不想在Consul手动管理服务条目和健康检查。

解决方案

使用Registrar。

讨论

本技巧是构建在之前的技巧之上的，并假设有一个两部分的Consul集群可用，如前文所描述的那样。我们还假设集群中没有服务，所以可能需要从头开始重新创建容器。

Registrar (<http://gliderlabs.com/registrator/latest/>) 消除了管理Consul服务的复杂性，它监视容器的启动和停止，根据暴露的端口和容器环境变量注册服务。了解该行为的最简单方法便是亲自去试试。

我们所做的一切都将在客户端代理机器上。如前所述，除服务器代理之外，我们不应该在其他机器上运行任何容器。

需要根据以下命令启动Registrar:

```
$ IMG=gliderlabs/registrator:v6
$ docker pull $IMG
[...]
$ ip addr | grep 'inet ' | grep -v 'lo$|docker0$'
    inet 192.168.1.80/24 brd 192.168.1.255 scope global wlan0
$ EXTIP=192.168.1.80
$ ip addr | grep docker0 | grep inet
    inet 172.17.42.1/16 scope global docker0
$ BRIDGEIP=172.17.42.1
$ docker run -d --name registrator -h $(hostname) -reg \
-v /var/run/docker.sock:/tmp/docker.sock $IMG -ip $EXTIP -resync \
60 consul://$BRIDGEIP:8500 # if this fails, $EXTIP is an alternative
b3c8a04b9dfaf588e46a255ddf4e35f14a9d51199fc6f39d47340df31b019b90
$ docker logs registrator
2015/08/14 20:05:57 Starting registrator v6 ...
2015/08/14 20:05:57 Forcing host IP to 192.168.1.80
```

```
2015/08/14 20:05:58 consul: current leader 192.168.1.87:8300
2015/08/14 20:05:58 Using consul adapter: consul://172.17.42.1:8500
2015/08/14 20:05:58 Listening for Docker events ...
2015/08/14 20:05:58 Syncing services on 2 containers
2015/08/14 20:05:58 ignored: b3c8a04b9dfa no published ports
2015/08/14 20:05:58 ignored: a633e58c66b3 no published ports
```

这里的第一个命令（用于拉动镜像和查找外部IP地址）应该看上去很熟悉。该IP地址会传给Registrar，这样一来它便知道使用哪个IP地址来广播服务。Docker套接字已挂载，以便容器启动和停止时随时通知Registrar。我们也告诉了Registrar该如何连接到 Consul 代理，我们希望所有的容器每 60 秒刷新一次。容器的变化会自动通知Registrar，因此最终的配置有助于减轻Registrar可能错过变更时带来的影响。

现在Registrar正在运行，注册第一个服务非常简单：

```
$ curl -sSL 172.17.42.1:8500/v1/catalog/services | python -m json.tool
{
  "consul": []
}
$ docker run -d -e "SERVICE_NAME=files" -p 8000:80 ubuntu:14.04.2 python3 \
-m http.server 80
3126a8668d7a058333d613f7995954f1919b314705589a9cd8b4e367d4092c9b
$ docker inspect 3126a8668d7a | grep 'Name.*/'
  "Name": "/evil_hopper",
$ curl -sSL 172.17.42.1:8500/v1/catalog/services | python -m json.tool
{
  "consul": [],
  "files": [] }
$ curl -sSL 172.17.42.1:8500/v1/catalog/service/files | python -m json.
tool
[
  {
    "Address": "192.168.1.80",
    "Node": "mylaptop2",
    "ServiceAddress": "192.168.1.80",
    "ServiceID": "mylaptop2-reg:evil_hopper:80",
    "ServiceName": "files",
    "ServicePort": 8000,
    "ServiceTags": null
  }
]
```

在注册服务时，要做的唯一一件事情便是传递一个环境变量给 **Registrator**，告诉它要使用的服务的名称。在默认情况下，**Registrator** 使用的名称基于斜杠之后和标签之前的容器名称组件，`mycorp.com/myteam/myimage:0.5` 的名称为 `myimage`。用户可以使用该命名约定，也可以根据自己的命名约定手动指定名称。

其余的值也正如预期的那样。**Registrator** 已经发现正在侦听的端口，将其添加到 **Consul**，并为之配置了一个服务 ID，用于指示哪里可以找到容器（这就是为什么主机名配置在了 **Registrator** 容器里）。

如果环境里有其他的详细信息，**Registrator** 也会获取，包括标签、每个端口的服务名称（如果有多个）以及所使用的健康检查（如果使用 **Consul** 作为数据存储）。可以在 JSON 中指定环境中检查的细节来启用所有 3 种类型的 **Consul** 健康检查。读者可以在“**Registrator Backends**”这一节文档的 **Consul** 部分阅读更多信息，网址为 <http://gliderlabs.com/registrator/latest/user/backends/#consul>。

如果在不断变化的环境中拥有一大波容器，那么 **Registrator** 会表现得很优秀，它可以确保用户不必再担心创建服务时是否建好健康检查的问题。

9.4 小结

本章可能是本书中最开放的部分。我们试图让读者了解 **Docker** 编排的世界，以便可以自行做出决定——在这里绝对没有一刀切的解决方案。

即使我们在这里对这些可用的工具做了这一调查，实际要具体选

择某一款时可能还是会令人很纠结的。我们的建议是保持尽可能简单，如果在本章开头考虑了图9-1中的分支，请权衡利弊。例如，如果流量增长表明两台服务器在下一年将足够，那么现在可能不需要动态置备以及服务发现。

本章包括的主题有：

- 使用Docker世界以外的成熟解决方案来控制单个机器上的容器执行；
- 简单的多宿主机编排解决方案；
- Docker编排领域的两个重量级工具；
- 将应用程序自动插入所选的服务发现后端。

下一章将转向更严肃的话题，即保证Docker安全性。

第10章 Docker与安全

本章主要内容

- Docker提供的开箱即用的安全方案到了何种水平
- Docker为更加安全做了哪些努力
- 其他开发商为了安全正在付出怎样的努力
- 为了改善安全问题，还可以采取哪些步骤
- 在多租户环境下如何管理用户

正如Docker在其文档中明确指出的，对于Docker API的调用需要root权限，这也就是为什么Docker通常需要用sudo命令来运行，或者必须把用户加入一个允许使用Docker API的用户组（该组可能叫作docker或者dockerroot）。

本章中我们将看一下Docker的安全问题。

10.1 Docker访问权限及其意味着什么

读者可能想知道一个用户如果可以运行Docker会造成多大的破坏。一个简单的例子是，这个命令（不要运行！）可能会删除宿主机/sbin目录下的所有二进制文件（如果拿掉了那个伪造的--donotrunme标志）：

```
docker run --donotrunme -v /sbin:/sbin busybox rm -rf /sbin
```

值得指出的是，即使你不是root用户这段代码仍然生效。下面这

条命令将会展示宿主机上安全shadow密码文件的内容：

```
docker run -v /etc/shadow:/etc/shadow busybox cat /etc/shadow
```

Docker的不安全性经常遭到误解，一部分是由于对内核中的命名空间的好处的误解。Linux命名空间提供了对系统中其他部分的隔离，但是对Docker的隔离级别是由用户自行决定的（正如前述的几个docker run例子所展示的）。而且，Linux操作系统不是所有部分都能够使用命名空间。设备和内核模块就是两个没有使用命名空间的Linux核心功能的例子。

Linux命名空间 Linux命名空间是设计来为进程提供独立于其他进程的系统视角。例如，进程命名空间意味着容器只能够看到与容器有关的进程——在同一宿主机上运行的其他进程对其而言是不可见的。网络空间命名意味着容器似乎有自己独有的网络栈可用。命名空间成为Linux内核的一部分已有多多年。

而且，由于用户在容器内部可以通过系统调用来与内核进行root级别的交互，所以任何的内核缺陷都可能在Docker容器内被利用。当然，虚拟机通过连接到hypervisor可以达到同样的攻击级别，只是危害小一些。hypervisor本身也被爆出一些安全缺陷。

另一种理解的方式是认为运行容器与能通过包管理工具安装程序没什么区别（从安全的角度看）。

换句话说，用户对运行Docker容器的安全需求应该和安装软件包是一样的。如果用户有Docker，可以作为root用户安装软件。这也有人说Docker最好被理解为一个软件打包系统的部分原因。

用户命名空间 有些工作正在进行以期望通过用户命名空间来解除此风险，它可以把容器中的root用户映射到宿主机上的非特权用户。

你在乎吗

考虑到调用Docker API和root权限是等价的，接下来的问题就是：“你在乎吗？”尽管这话看上去有点儿奇怪，安全全然是关乎信任的，如果你信任用户在他们操作的环境里安装软件，那么他们在那儿运行Docker就应该没有障碍。安全的困难性主要体现在多租户环境下。由于容器内的root用户在关键的方面都和容器外的root用户一样，所以系统内拥有众多root用户可能会有隐患。

多租户 一个多租户环境就是许多用户共享同样的资源的环境。举例来说，两个团队可能通过两个不同的虚拟机来共享同一台服务器。多租户通过共享硬件而非向特定应用提供硬件来节省开支。但是它可能带来一些足以抵消开支节省的挑战，包括服务可靠性以及安全隔离。

一些组织采取了把每个用户的Docker运行在专用虚拟机上的做法。这台虚拟机可以用于安全隔离、运维隔离或者资源隔离。在虚拟机的信任界限内，用户运行Docker容器以获取其性能和运维上的好处。这就是Google计算引擎采取的方式，为了更进一步的安全性和其他一些运维上的好处，在用户的容器和其运行的基础设施间再加一个虚拟机。Google公司拥有大量可用的计算资源，所以他们不介意这样做的开销。

10.2 Docker中的安全手段

Docker维护者采取了多种手段来减少运行容器的安全风险。例如：

- 某些特定的核心挂载点（如/proc和/sys）现在是只读方式挂载的；
- 降低了默认的Linux能力；
- 现在已经存在对第三方安全系统（如SELinux和AppArmor）的支持。

本节中，我们会对这些有更深入的了解，可以采取其中的一些手段来降低在系统上运行容器的风险。

技巧84 限制能力

正如我们提到的，容器上的root用户和宿主机上的root用户是一样的。但是不是所有的root都生而平等。Linux允许为root用户分配进程级别的细粒度的权限。这些细粒度的权限被称为能力（capability），这样即使是root用户，也能限制他们所能做的破坏。本技巧展示了当运行Docker容器的时候如何操纵这些能力。

问题

想要降低容器在宿主机上进行破坏性活动的的能力。

解决方案

使用--drop-cap标志来减少容器拥有的访问权限。

讨论

如果并非完全信任在系统上运行的容器的内容，可以通过减少容器可以获得的能力来降低出问题的风险。

1. Unix信任模型

为了了解本技巧的含义和作用，需要一点儿背景知识。当Unix系统被设计出来的时候，其信任模型并不复杂。存在被信任的管理员（root用户）和不被信任的用户。root用户可以做任何事情，然而普通用户只能影响他们自己的文件。因为这个操作系统一般是在大学实验室里使用而且本身不大，所以这个模型还是合理的。

随着Unix模型的发展以及互联网的到来，这个模型越来越不合理了。类似网络服务器的程序需要root权限来在80端口提供内容，同时也作为在宿主机上运行命令的有效代理。针对这些情况有些标准的应对模式，例如，绑定到端口80并把有效用户ID赋予一个非root用户。扮演着不同角色的用户，从系统管理员到数据库管理员，直到应用支持工程师和开发者，可能都需要对不同的系统上的资源有细粒度的访问权限。Unix用户组从某种程度上减轻了这个问题，但是正如任何系统管理员都会说的那样，为这些权限需求建模并不是一个小问题。

2. Linux能力

为了尝试支持一个更加细粒度的对用户权限进行管理的方式，Linux内核工程师们开发了能力。它尝试把单块的root权限拆解成各个可以独立授予的功能片段。读者可以通过运行`man 7 capabilities`来查看更多细节（假设安装了帮助手册）。

有用的是，Docker默认关闭了一些特定的能力。也就是说，即使

在容器内有root权限，有些事也做不了。例如，允许影响网络栈的CAP_NET_ADMIN能力，默认就被禁用了。

表10-1列出了Linux的各项能力，给出了对它们允许做的事情的简要介绍，并且标明了它们是否在Docker容器内默认开启。记住，每项能力都和root用户改变其他用户在系统上的文件的能力有关。例如，容器内root用户仍然能chown宿主机上root的文件，只要该文件可以作为容器上的卷访问。

表10-1 Docker容器中的Linux能力

能力	描述	开启与否
CHOWN	对任意文件进行所有权改变	是
DAC_OVERRIDE	重载读、写和执行权限检查	是
FSETID	当修改文件时，不清除suid和guid位	是
FOWNER	存储文件时，无视所有权检查	是
KILL	对于信号，绕过权限检查	是
MKNOD	使用mknod来创建特殊文件	是
NET_RAW	使用原始套接字和分组套接字，并且绑定到端口以进行透明代理	是

SETGID	对进程的组所有权进行更改	是
SETUID	对进程的用户所有权进行更改	是
SETFCAP	设定文件能力	是
SETPCAP	如果不支持文件能力，那么对来自其他进程和发往其他进程的能力进行限制	是
NET_BIND_SERVICE	绑定套接字到小于1024的端口	是
SYS_CHROOT	使用chroot	是
AUDIT_WRITE	写入内核日志	是
AUDIT_CONTROL	启用/禁用内核日志记录	否
BLOCK_SUSPEND	使用能阻止系统中止的特性	否
DAC_READ_SEARCH	绕过读取文件和目录时的权限检查	否
IPC_LOCK	锁定内存	否
IPC_OWNER	绕过进程间通信对象权限	否
LEASE	在一般文件上建立租约（对试图打开或者删除的监控）	否
LINUX_IMMUTABLE	设立i-node标志FS_APPEND_FL和FS_IMMUTABLE_FL	否

MAC_ADMIN	重载强制访问控制（和SmackLinux安全模组（SLM）有关）	否
MAC_OVERRIDE	改变强制访问控制（和SLM有关）	否
NET_ADMIN	各种网络相关的操作，包括IP防火墙改变和网关配置	否
NET_BROADCAST	不再使用	否
SYS_ADMIN	一系列管理员功能。查看man capabilities获取更多信息	否
SYS_BOOT	重启	否
SYS_MODULE	载载/卸载内核模块	否
SYS_NICE	操纵进程的nice优先级	否
SYS_PACCT	开启或关闭进程记账	否
SYS_PTRACE	追踪进程的系统调用以及其他进程操纵能力	否
SYS_RAWIO	对系统很多核心部分进行输入/输出，如内存和SCSI设备命令	否
SYS_RESOURCE	控制和重载多种资源限制	否
SYS_TIME	设置系统时钟	否
SYS_TTY_CONFIG	在虚拟终端上的特权操作	否

基于libcontainer引擎 如果读者不是在使用Docker默认的引擎容器（libcontainer），这些能力可能在读者安装的软件上有所不同。如果有系统管理员并且想要确认这些，就去请教他们。

但是，内核维护者仅在系统内分配了32个能力，所以这些能力都拓展了自己的范围，同时越来越多的细粒度root权限在内核外被创造出来。最值得一提的是，命名模糊的CAP_SYS_ADMIN能力涵盖了从改变宿主机域名到超出系统范围内打开文件数量的上限等多种不同行为。

一种极端的做法是从容器内移除所有在Docker默认开启的权限，然后看一下什么不工作了。在此我们运行一个移除所有默认开启的能力的bash脚本：

```
$ docker run -ti --cap-drop=CHOWN --cap-drop=DAC_OVERRIDE \  
--cap-drop=FSETID --cap-drop=FOwner --cap-drop=KILL --cap-drop=MKNOD \  
--cap-drop=NET_RAW --cap-drop=SETGID --cap-drop=SETUID \  
--cap-drop=SETFCAP --cap-drop=SETPCAP --cap-drop=NET_BIND_SERVICE \  
--cap-drop=SYS_CHROOT --cap-drop=AUDIT_WRITE debian /bin/bash
```

如果通过这个脚本来运行程序，可以看到它是在什么地方如期失败，然后重新加上所需的能力。例如，用户可能需要改变文件所有权的能力，那么在前述的代码中就不能去掉FOWNER能力：

```
$ docker run -ti --cap-drop=CHOWN --cap-drop=DAC_OVERRIDE \  
--cap-drop=FSETID --cap-drop=KILL --cap-drop=MKNOD \  
--cap-drop=NET_RAW --cap-drop=SETGID --cap-drop=SETUID \  
--cap-drop=SETFCAP --cap-drop=SETPCAP --cap-drop=NET_BIND_SERVICE \  
--cap-drop=SYS_CHROOT --cap-drop=AUDIT_WRITE debian /bin/bash
```

移除/启用所有的能力 如果想要启用或者禁用所有的能力，可以使用all而不是某个特定的能力，如docker run -ti --cap-

```
drop=all ubuntu bash。
```

如果在bash脚本中运行了一些基本的命令，会发现这很有用。尽管在运行一些更复杂的程序的时候得到的好处可能会有所不同。

root还是root! 值得澄清的是，这些能力中的很大一部分是和影响其他用户的对象的root能力相关的，而不是root自己的对象。一个root用户仍然能够chown宿主机上root的文件。例如，假设他们是在容器内操作并且通过卷加载的方式能够访问宿主机的文件。因此，纵使所有这些能力都关闭了，仍然值得把程序降级为一个非root用户。

这种对容器的能力的调优能力意味着对docker run使用--privileged标志就不必要了。需要能力的进程会被审计并且处于宿主机管理员的控制之下。

技巧85 Docker实例上的HTTP认证

在技巧1中我们看到了如何给守护进程打开网络访问权限，在技巧4中我们了解了如何使用socat来嗅探Docker API。

本技巧把二者结合起来：我们将能远程访问自己的守护进程并查看其反应。访问仅限于那些拥有用户名/密码组合的人，所以稍微安全些。除此之外还有好处，不用通过重启Docker守护进程来达到这个目的——启动一个容器守护进程吧！

问题

想要一些对Docker守护进程上可用的网络访问的基本验证。

解决方案

设立HTTP验证。

讨论

在本技巧中，我们将展示如何使用一种临时的办法把自己的Docker守护进程分享给其他人。图10-1给出了整体架构的布局。

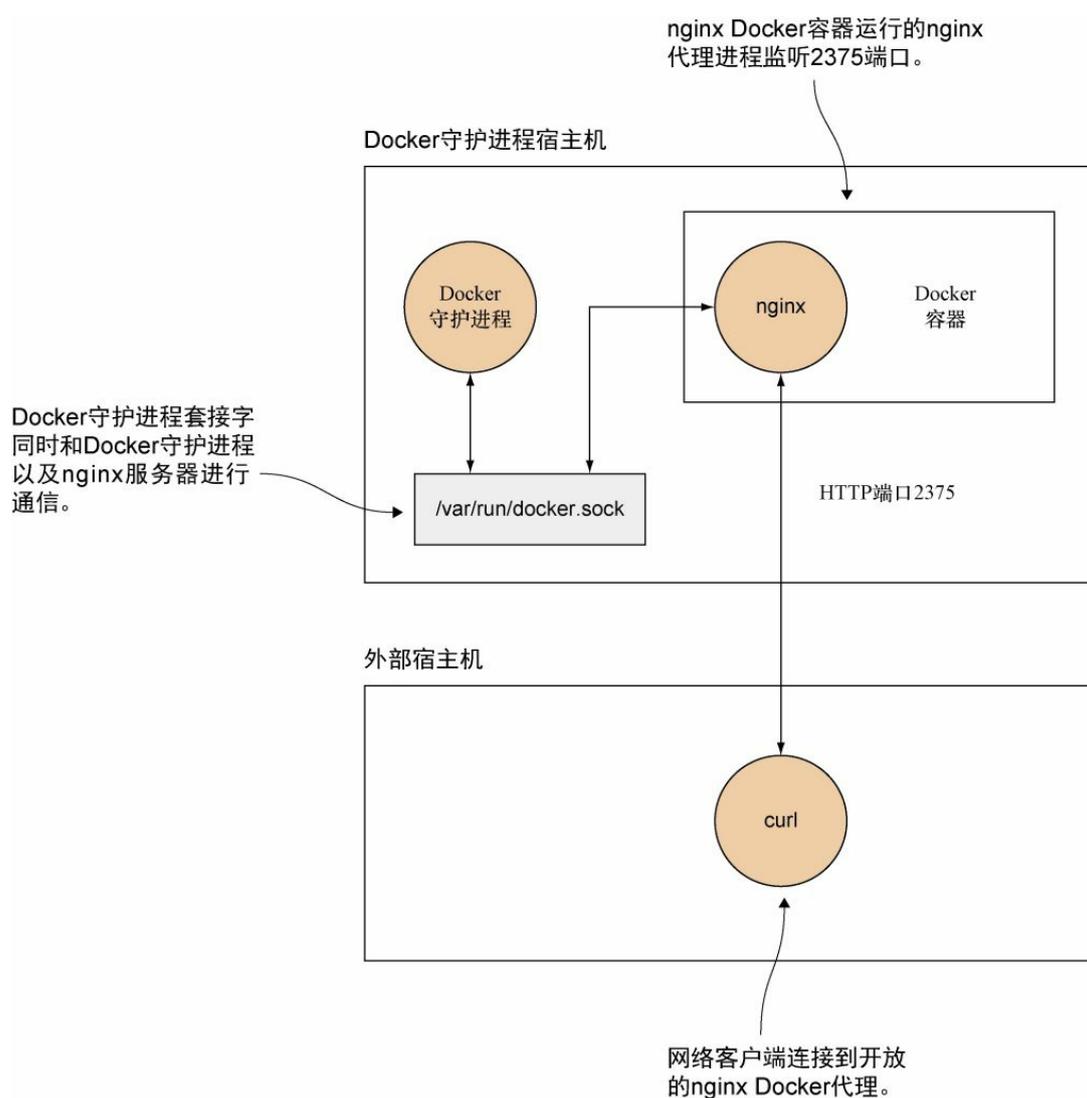


图10-1 一个带有基本认证的Docker守护进程架构

假定的 **Docker** 默认设置 本讨论假设用户的Docker守护进程使用的是位于/var/run/docker.sock中的Docker默认的Unix套接字接入方

法。

本技巧中的代码可以在<https://github.com/docker-in-practice/docker-authenticate>找到。代码清单10-1展示了这个仓库中的用来创建本技巧中的镜像的Dockerfile。

代码清单10-1 用来创建dockerinpractice/docker-authenticate镜像的Dockerfile

```
FROM debian
RUN apt-get update && apt-get install -y \
  nginx apache2-utils (1)
RUN htpasswd -c /etc/nginx/.htpasswd username (2)
RUN htpasswd -b /etc/nginx/.htpasswd username password (3)
RUN sed -i 's/user .*/user root;/' \
  /etc/nginx/nginx.conf (4)
ADD etc/nginx/sites-enabled/docker \
  /etc/nginx/sites-enabled/docker (5)
CMD service nginx start && sleep infinity (6)
```

(1) 确保需要的软件都安装并且更新好了

(2) ❶为名为username的用户创建密码文件

(3) ❷为名为username的用户创建密码为password

(4) nginx 需要以 root 权限运行以获取Docker Unix套接字，因此我们把用户那一行替换为了root用户的信息

(5) 在Docker的nginx site文件中复制（代码清单10-2）

(6) 默认开启nginx服务并且无限等待

在❶和❷中建立的密码文件包含在允许（或者禁止）连接到Docker套接字之前的凭证。如果用户在自己创建此镜像，可能想改变这两步中的username和password来定制对Docker套接字权限的凭

证。

保持此镜像私有 小心不要共享此镜像，因为它包含你设置的密码！

Docker中的nginx site文件如代码清单10-2所示。

代码清单10-2 /etc/nginx/sites-enabled/docker

```
upstream docker {
    server unix:/var/run/docker.sock; (1)
}

server {
    listen 2375 default_server; (2)
    location / {
        proxy_pass http://docker; (3)
        auth_basic_user_file /etc/nginx/.htpasswd; (4)
        auth_basic "Access restricted"; (5)
    }
}
```

- (1) 定义指向Docker的域套接字时nginx中“docker”的位置
- (2) 监听2375端口（标准的Docker端口）
- (3) 将发往和来自之前定义的“docker”地址的请求代理
- (4) 定义要用到的密码文件
- (5) 通过密码限制权限

现在把这个镜像作为守护进程容器运行起来，将需要的宿主机上的资源映射进来：

```
$ docker run -d --name docker-authenticate -p 2375:2375 \
-v /var/run:/var/run dockerinpractice/docker-authenticate
```

这条命令会在后台以docker-authenticate为名运行，之后可以引用。在宿主机上暴露了容器的2375端口，而且这个容器加载了默认的含有Docker套接字的目录，从而能够访问Docker守护进程。如果读者用的是自己定制的镜像，有自己的用户名和密码，在这里需要把镜像名替换成自己的。

网络服务现在应该就启动运行起来了。如果读者使用自己设立的用户名和密码来curl这个服务，应该能看到一个API响应：

```
$ curl http://username:password@localhost:2375/info (1)
{"Containers":115,"Debug":0, (2)
  ➤ "DockerRootDir":"/var/lib/docker","Driver":"aufs",
  ➤ "DriverStatus":[["Root Dir","/var/lib/docker/aufs"],
  ➤ ["Backing Filesystem","extfs"],["Dirs","1033"]],
  ➤ "ExecutionDriver":"native-0.2",
  ➤ "ID":"QSCJ:NLPA:CRS7:WCOI:K23J:6Y2V:G35M:BF55:0A2W:MV3E:RG47:DG23",
  ➤ "IPv4Forwarding":1,"Images":792,
  ➤ "IndexServerAddress":"https://index.docker.io/v1/",
  ➤ "InitPath":"/usr/bin/docker","InitSha1":"",
  ➤ "KernelVersion":"3.13.0-45-generic",
  ➤ "Labels":null,"MemTotal":5939630080,"MemoryLimit":1,
  ➤ "NCPU":4,"NEventsListener":0,"NFd":31,"NGoroutines":30,
  ➤ "Name":"rothko","OperatingSystem":"Ubuntu 14.04.2 LTS",
  ➤ "RegistryConfig":{"IndexConfigs":{"docker.io":
  ➤ {"Mirrors":null,"Name":"docker.io",
  ➤ "Official":true,"Secure":true}}},
  ➤ "InsecureRegistryCIDRs":["127.0.0.0/8"]},"SwapLimit":0}
```

(1) 在要curl的URL中写上username:password，地址放在@符号后。本请求是访问Docker守护进程API的/info端点

(2) 从Docker守护进程返回的JSON

完成之后，通过下面的命令移除容器：

```
$ docker rm -f docker-authenticate
```

权限现在被收回了！

使用docker命令？

读者可能想知道其他用户能否通过docker命令来链接。例如，类似于：

```
docker -H tcp://username:password@localhost:2375 ps
```

在编写本书时，验证功能尚未被内置到Docker本身。但是我们已经创建了可以处理验证信息并且允许Docker连接到守护进程的镜像。简单地使用此镜像如下：

```
$ docker run -d --name docker-authenticate-client \ (1)
  -p 127.0.0.1:12375:12375 \ (2)
  dockerinpractice/docker-authenticate-client \ (3)
  192.168.1.74:2375 username:password (4)
```

(1) 在该背景下运行客户端容器，并给它一个名字

(2) 暴露出一个端口用来让Docker连接到守护进程，但是仅允许来自本地机器的连接

(3) 使用我们创建的镜像以允许和Docker建立验证连接

(4) 镜像的两个参数（指定验证连接的另一端以及用户名和密码）应当替换为你自己设立的合适的值

注意，对于指定验证连接的另一端来说，localhost或者127.0.0.1是不起作用的，如果想要在一台宿主机上试用，必须使用ip addr来指定机器的外部IP地址。

现在可以用下列命令来使用验证连接：

```
docker -H localhost:12375 ps
```

注意，由于一些实现上的限制，交互式的Docker命令（`run`、`exec`带上`-i`参数）不能使用。

不要指望这样就安全了！ 这样提供了基本的验证，但是这并不能提供真正意义上的安全（尤其能够监听你网络通信的人可以拦截你的用户名和密码）。更加需要的是创建一个用TLS保护的服务器，接下来的技巧中会介绍。

技巧86 保护Docker API

在本技巧中我们会展示可以通过TCP端口向其他人开放自己的Docker服务器，同时确保只有受信任的用户才能连接。这是通过创建一个只有受信任的宿主机才能得到的密钥来实现的。只要受信任的密钥在服务器和客户端之间保持秘密，那么Docker服务器应当就是安全的。

问题

想要Docker API能通过端口安全地服务。

解决方案

创建一个自签名的证书并且带上`--tls-verify`标志来运行Docker守护进程。

讨论

本安全方法依赖于服务器上创建的所谓密钥文件（`key file`）。这

些文件是通过一些特殊工具来创建的，确保了如果没有服务器密钥（server key），就很难复制。图10-2大体介绍了这种方法是如何工作的。

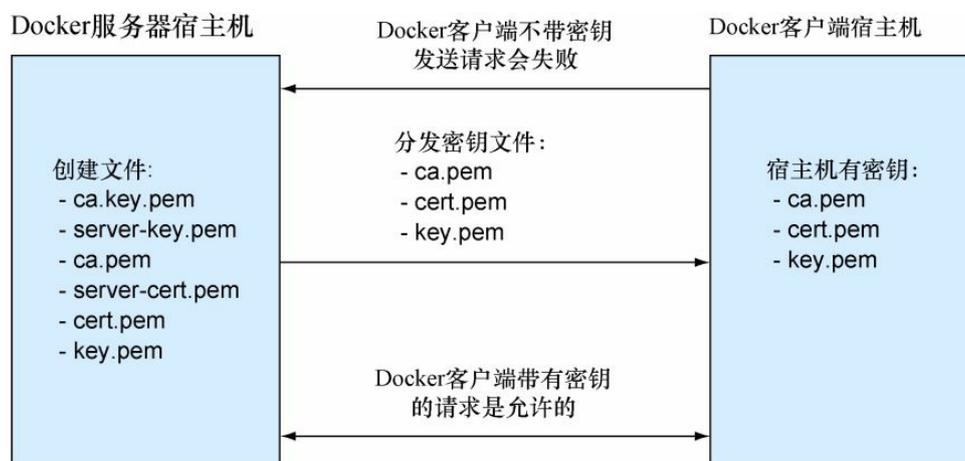


图10-2 创立密钥以及分发

什么是服务器密钥和客户端密钥 服务器密钥是一个保存有仅服务器知道的秘密数字的文件，读取被服务器的主人分发出去的密钥（所谓客户端密钥）加密过的信息，它是必不可少的。一旦这些密钥创立并分发完毕，就可以用它们来建立客户端和服务器的安全连接。

1. 创建Docker服务器证书

首先要创建证书和密钥。产生密钥需要使用OpenSSL包。在终端运行`openssl`命令来检查它是否已经安装好了。如果没有安装，在使用下面的代码来产生证书和密钥之前需要进行安装：

```
$ sudo su (1)
$ read -s PASSWORD (2)
$ read SERVER
$ mkdir -p /etc/docker (3)
$ cd /etc/docker
```

```

$ openssl genrsa -aes256 -passout pass:$PASSWORD \ (4)
-out ca-key.pem 2048
$ openssl req -new -x509 -days 365 -key ca-key.pem -passin pass:$PASSWO
RD \
-sha256 -out ca.pem -subj "/C=NL/ST=../L=../O=../CN=$SERVER" (5)

$ openssl genrsa -out server-key.pem 2048 (6)
$ openssl req -subj "/CN=$SERVER" -new -key server-key.pem \
-out server.csr (7)
$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.p
em
-passin "pass:$PASSWORD" -CAcreateserial \
-out server-cert.pem (8)
$ openssl genrsa -out key.pem 2048 (9)
$ openssl req -subj '/CN=client' -new -key key.pem\
-out client.csr (10)
$ sh -c 'echo "extendedKeyUsage = clientAuth" > extfile.cnf'
$ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.p
em \
-passin "pass:$PASSWORD" -CAcreateserial -out cert.pem \
-extfile extfile.cnf (11)
$ chmod 0400 ca-key.pem key.pem server-key.pem (12)
$ chmod 0444 ca.pem server-cert.pem cert.pem (13)
$ rm client.csr server.csr (14)

```

- (1) 确保你是root用户
- (2) 输入你证书的密码和你将用来连接Docker服务器的服务器名称
- (3) 如果docker配置目录不存在，创建它，并进入该目录
- (4) 使用2048位安全码产生证书授权（CA）.pem文件
- (5) 用你的密码和地址给CA密钥签名一年期
- (6) 用2048位安全码产生服务器密钥
- (7) 用你宿主机的名字处理服务器密钥
- (8) 使用你的密码给密钥签名一年期

- (9) 用2048位安全码产生一个客户端密钥
- (10) 把这个密钥处理成客户端密钥
- (11) 用你的密码给密钥签名一年期
- (12) 把服务器文件的权限改为对root只读
- (13) 把客户端文件的权限改为对所有人只读
- (14) 删除剩余文件

辅助函数 一个叫CA.pl的脚本在你的系统里可能已经安装好了，它可以简化这个过程。我们在这里通过原始的openssl命令的方式是因为这样更富有指导性。

2. 设置Docker服务器

接下来需要在Docker守护进程文件里设置Docker选项来指定使用哪个密钥来为通信加密（参考附录B来了解如何配置以及重启Docker守护进程）：

```
DOCKER_OPTS="$DOCKER_OPTS --tlsverify" (1)
DOCKER_OPTS="$DOCKER_OPTS \
--tlscacert=/etc/docker/ca.pem" (2)
DOCKER_OPTS="$DOCKER_OPTS \
--tlscert=/etc/docker/server-cert.pem" (3)
DOCKER_OPTS="$DOCKER_OPTS \
--tlskey=/etc/docker/server-key.pem" (4)
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2376" (5)
DOCKER_OPTS="$DOCKER_OPTS \
-H unix:///var/run/docker.sock" (6)
```

- (1) 告诉Docker守护进程你想要通过TLS加密的方式来保障连

接的安全

- (2) 为Docker服务端指定CA文件
- (3) 为服务器指定证书
- (4) 指定服务器使用的私钥
- (5) 在2376端口把Docker守护进程通过TCP开放给外界
- (6) 按照一般的做法，通过一个套接字来在本地打开Docker守护进程

3. 分发客户端密钥

接下来需要把密钥发送到客户端宿主机上以便其能连接到服务器并且交换信息。我们不希望向其他人展示自己的密钥，所以这个过程也需要安全地传送给客户端。一种相对安全的做法是通过SCP（安全复制）从服务器直接复制到客户端。SCP组件总体上是使用了我们在这里展示的技术来传输数据的，只是使用的是不同的密钥。

在客户端宿主机上，如同之前做的那样在/etc下创建Docker配置文件夹：

```
user@client:~$ sudo su
root@client:~$ mkdir -p /etc/docker
```

然后通过SCP把文件从服务器传输到客户端。确保在接下来的命令中把client替换为了你的客户端及其宿主机名。同时要保证对于要在客户端运行docker命令的用户来说，这些文件都是可读的。

```
user@server:~$ sudo su
```

```
root@server:~$ scp /etc/docker/ca.pem client:/etc/docker
root@server:~$ scp /etc/docker/cert.pem client:/etc/docker
root@server:~$ scp /etc/docker/key.pem client:/etc/docker
```

4. 测试

为了测试你的设置，首先不带任何凭证向Docker服务器发起请求，应该被拒绝才对：

```
root@client~: docker -H myserver.localdomain:2376 info
FATA[0000] Get http://myserver.localdomain:2376/v1.17/info: malformed HTTP
response "\x15\x03\x01\x00\x02\x02". Are you trying to connect to a
TLS-enabled daemon without TLS?
```

接下来使用凭证来链接，应该会返回一些有用的输出：

```
root@client~: docker --tlsverify --tlscacert=/etc/docker/ca.pem \
--tlscert=/etc/docker/cert.pem --tlskey=/etc/docker/key.pem \
-H myserver.localdomain:2376 info
243 info
Containers: 3
Images: 86
Storage Driver: aufs
  Root Dir: /var/lib/docker/aufs
  Backing Filesystem: extfs
  Dirs: 92
Execution Driver: native-0.2
Kernel Version: 3.16.0-34-generic
Operating System: Ubuntu 14.04.2 LTS
CPUs: 4
Total Memory: 11.44 GiB
Name: rothko
ID: 4YQA:KK65:FXON:YVLT:BVVH:Y3KC:UATJ:I4GK:S3E2:UTA6:R43U:DX5T
WARNING: No swap limit support
```

本技巧带来了两方面的好处——一个开放给其他人使用的Docker守护进程，以及只有受信任的用户可以访问。一定要保证密钥的安全！

10.3 来自Docker以外的安全

宿主机上的安全性不仅在于使用docker命令。在本节中读者会看到另外两种保障Docker容器安全的方式，这次是来自Docker以外的。

第一种方式展示了应用程序平台即服务（application platform as a service, aPaaS）的方式，它采用严加约束并受到管理员控制的方式运行Docker。作为例子，我们用Docker命令运行一个OpenShift Origin服务器（一个以可控的方式部署Docker的aPaas）。我们会看到终端用户的能力会受到管理员的限制和管理，对于Docker运行时的访问也可以移除了。

第二种方式不止是这种程度的安全性，它使用SELinux（一个可以限制谁可以做什么的细粒度的安全技术）来进一步限制了运行中的容器内部的自由。

什么是**SELinux** SELinux是一个由美国国家安全局（NSA）创建并开源的工具，它满足了他们对于强访问控制的需求。至今为止它成为安全标准已经有段时间了，并且十分强大。但是，很多人遇到问题的时候都直接把它关了，而不是花时间来理解它。我们希望在`这里展示的技巧能够让这种方式不那么让人望而却步。`

技巧87 OpenShift——一个应用程序平台即服务

OpenShift是一个由Red Hat管理的產品，它允许一个组织运行应用程序平台即服务（aPaas）并为开发团队提供了一个不需要关心硬件细节就可以运行代码的平台。这个产品的第三版用Go语言进行了

彻头彻尾的重写，使用Docker作为容器技术，并用Kubernetes和etcd进行编排。不仅如此，Red Hat还加入了一些企业级特性，使其能够简单地部署到企业及关注安全的环境中。

尽管我们可以讨论OpenShift的很多特性，但在这里我们只把它作为一种安全管理的方式，去除用户直接运行Docker的能力，同时又保持使用Docker的好处。

OpenShift 有企业支持的商业产品，也有开源项目名为Origin，后者在[https://github.com/ openshift/origin](https://github.com/openshift/origin)维护。

问题

想要管理不受信任的用户调用docker run的安全风险。

解决方案

使用OpenShift这样的应用程序平台即服务（aPaaS）工具。

讨论

aPaaS有很多优点，我们在这里关注的是它管理用户权限和代表用户运行Docker容器的能力，这为运行Docker容器的用户提供了安全审计点。

为什么这一点很重要？使用这一aPaaS的用户没有调用docker命令的直接权限，因此，除非他们颠覆OpenShift提供的安全性，否则他们不能做出任何破坏。例如，默认来说容器是由非root用户部署的，想要克服这一点需要由管理员授权。如果不能信任用户，那么使用aPaaS是给他们Docker访问权限的高效方式。

什么是**aPaaS** aPaaS为用户提供了为开发、测试乃至生产环境按需快速启动应用程序的能力。Docker对这些服务天然适用，因为它提供了一种可靠而隔离的应用交付格式，让运维团队去处理部署细节。

简而言之，OpenShift在Kubernetes基础上构建（见技巧79），但是增加了一些合格的aPaaS的特性。这些额外的特性包括：

- 用户管理；
- 权限管理；
- 限额；
- 安全上下文；
- 路由。

1. 安装OpenShift

对OpenShift的安装做一个完全介绍超出了本书的范围。

如果想要我们维护的使用Vagrant的自动安装，参见<https://github.com/docker-in-practice/shutit-openshift-origin>。如果在安装Vagrant时需要帮助，参考附录C。

其他选项，如一个仅限Docker的安装（仅限单节点）或者完全手工的构建，在OpenShift Origin的代码库<https://github.com/openshift/origin.git>上都可以找到而且还有文档。

什么是**OpenShift Origin** OpenShift Origin是OpenShift的“上游”版本。“上游”这里是指它是由Red Hat同步了代码，为OpenShift定制了

一些功能，它是OpenShift官方支持的作品。Origin是开源的，任何人都可以使用，也接受任何人的贡献。但它由Red Hat管理的版本是收费的，并且作为“OpenShift”项目受到支持。上游版本通常更加先进但是不那么稳定。

2. OpenShift应用程序

在本技巧中我们要使用OpenShift网络接口展示一个创建、构建、运行和访问应用程序的简单的例子。这个应用程序是一个提供简单Web页面的基本NodeJS应用程序。

这个应用程序会在底层用到Docker、Kubernetes和S2I。Docker用来封装构建和部署环境。来自技巧48的从源代码到镜像（S2I）的构建方法被用于构建 Docker 容器，Kubernetes被用于在OpenShift集群上运行应用程序。

3. 登录

要开始登录，先在shutit-openshift-origin文件夹运行`./run.sh`，然后导航到`https://localhost:8443`，忽视所有的安全警告。我们会看到图 10-3所示的登录页面。注意，如果是使用Vagrant安装的，那么需要在虚拟机里启动一个Web浏览器。（要了解如何给虚拟机添加图形用户界面，参见附录C。）

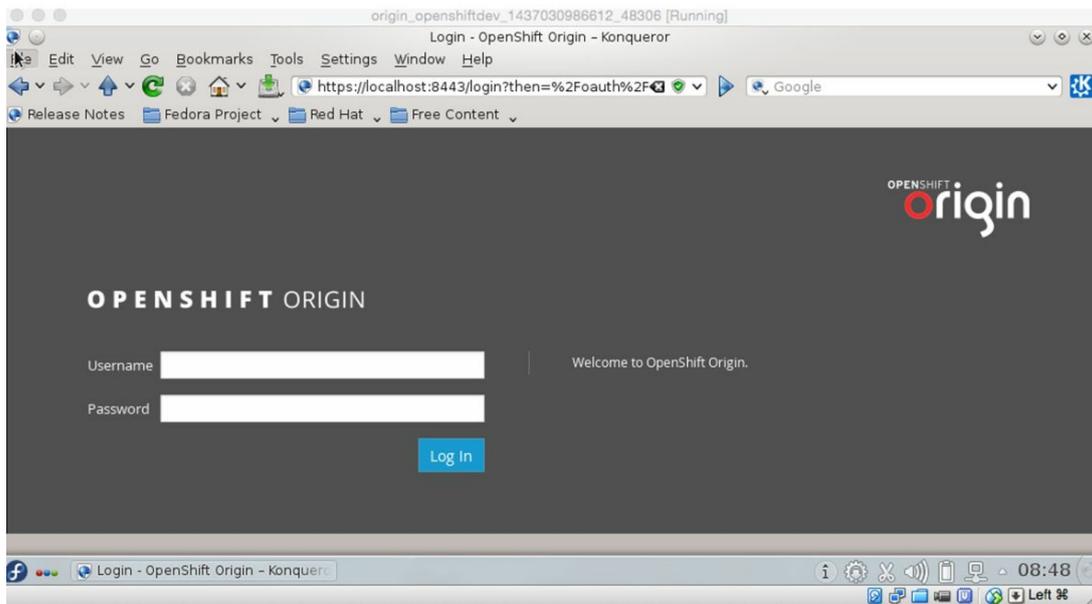


图10-3 OpenShift登录页面

使用任意密码以hal-1登录。

4. 构建一个NodeJS应用

现在以开发者的身份登录到了OpenShift（参见图10-4）。

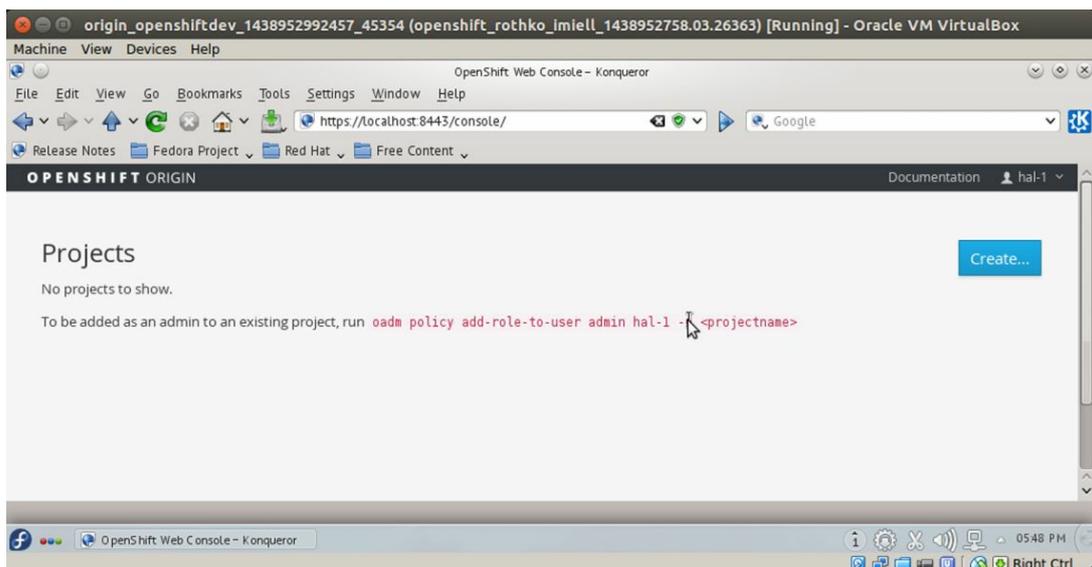


图10-4 OpenShift项目页面

点击Creat按钮来创建一个项目。如图10-5所示这样填写表单。然后再次点击Create按钮。

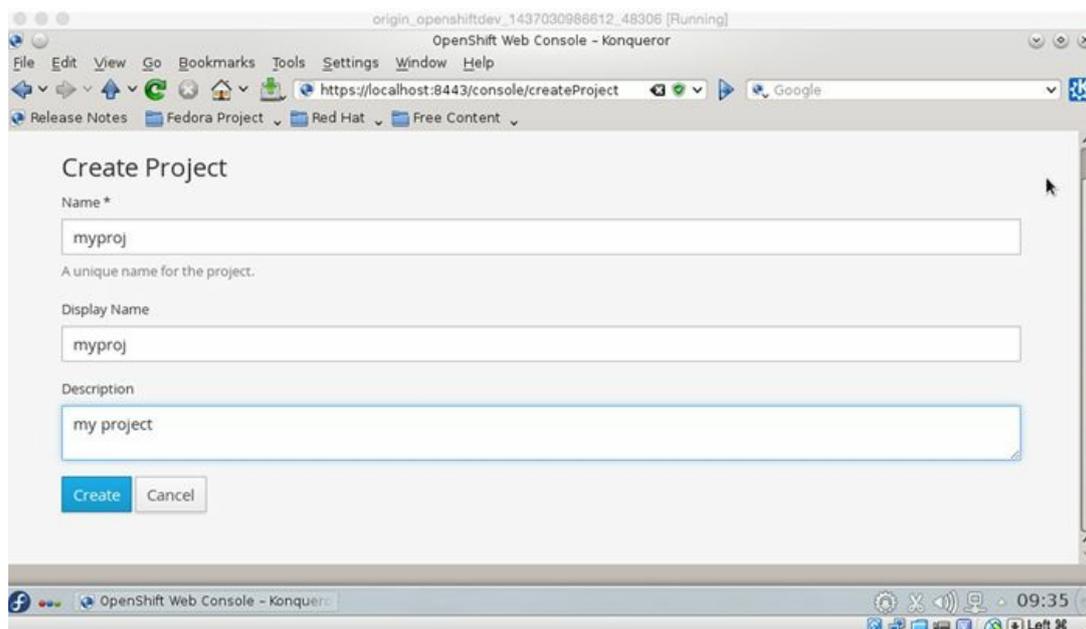


图10-5 OpenShift项目创建页面

一旦项目创建完成，再次点击Create按钮，输入推荐的 GitHub 仓库（<https://github.com/openshift/nodejs-ex>），如图10-6所示。

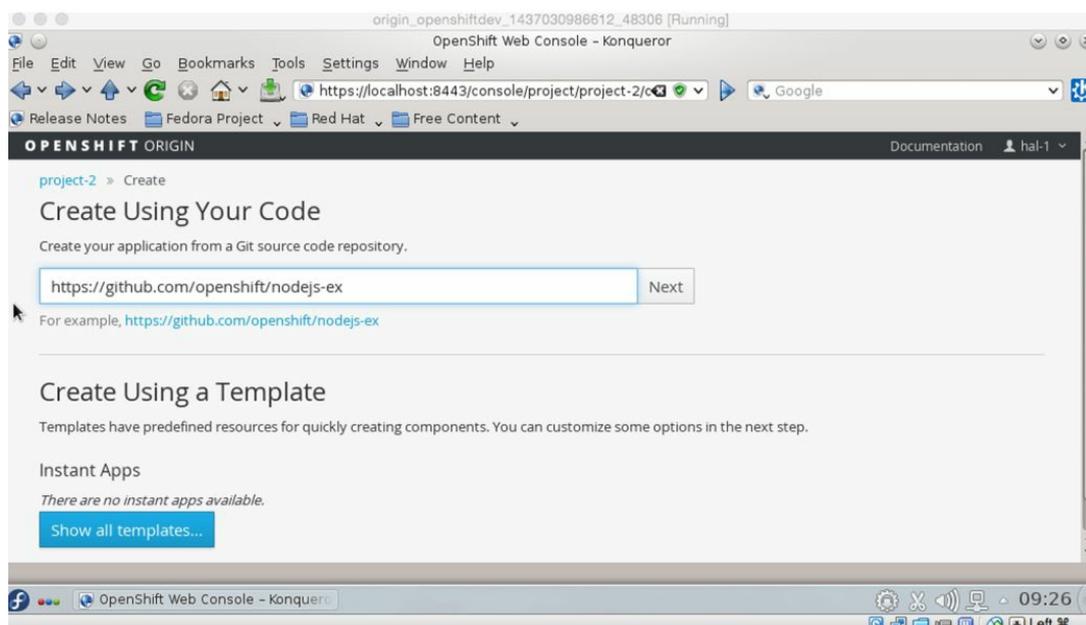


图10-6 OpenShift项目源页面

点击Next按钮，要在众多创建镜像中选择一个，如图10-7所示。
创建镜像定义了代码构建的上下文。选择NodeJS构建镜像。

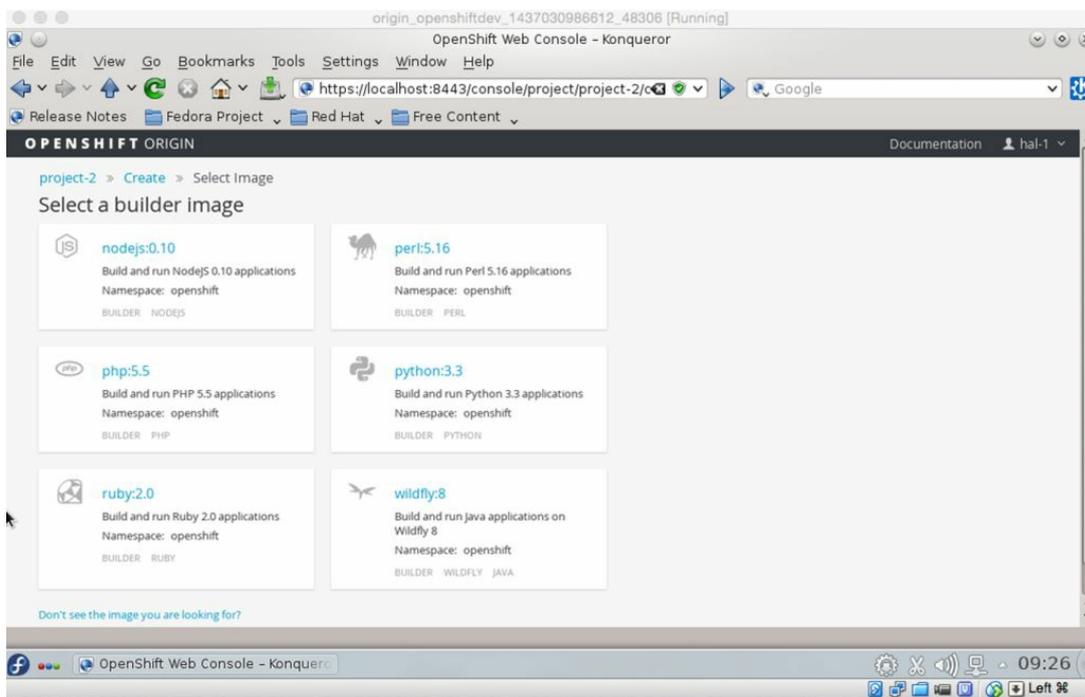


图10-7 OpenShift构建镜像选择页面

现在像图10-8所示这样填写表单，滚动到表单下方，选择页面底部的Create on NodeJS。

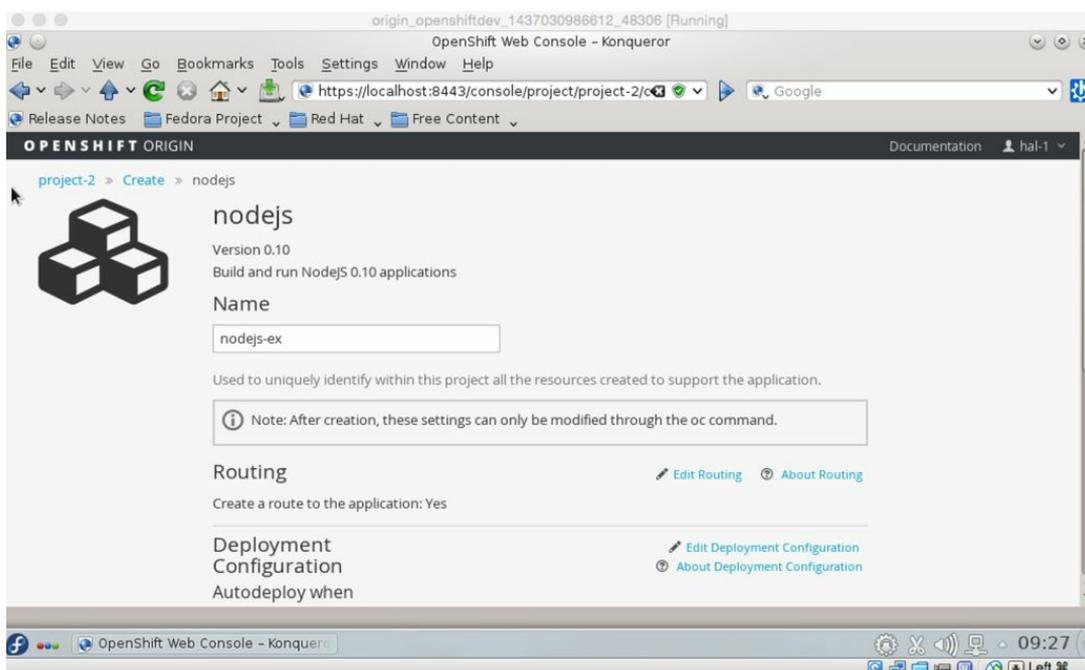


图10-8 OpenShift NodeJS模板表单

几分钟后，屏幕应该如图10-9所示。

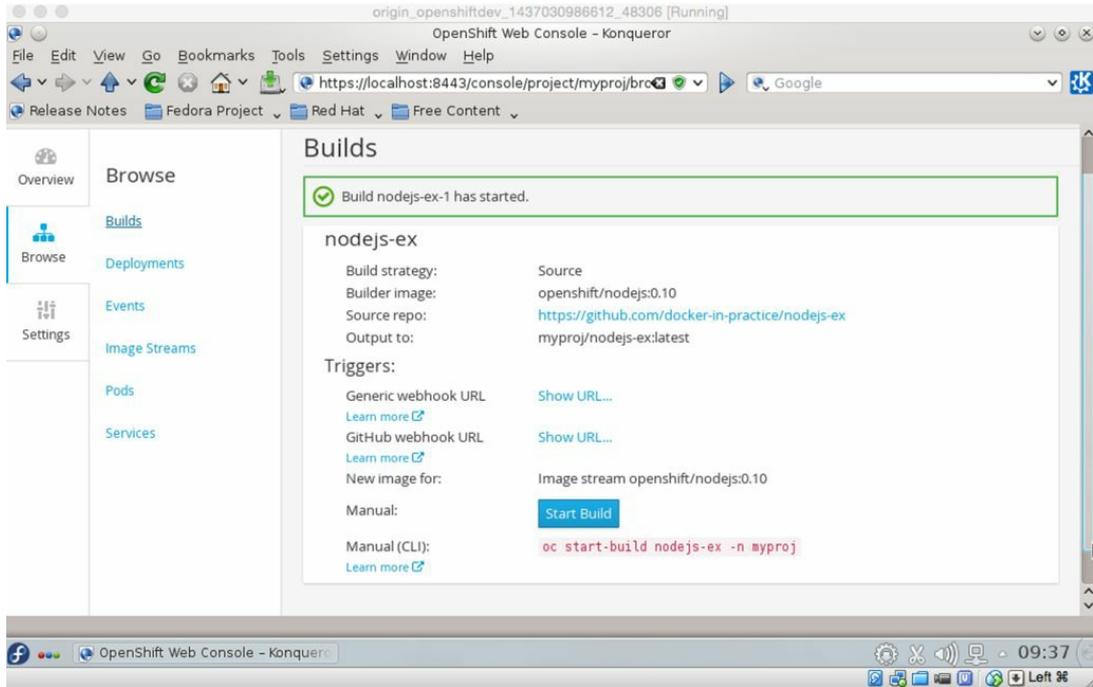


图10-9 OpenShift开始构建页面

过一会儿，如果向下滚动屏幕，就会看到构建已经开始，如图10-10所示。

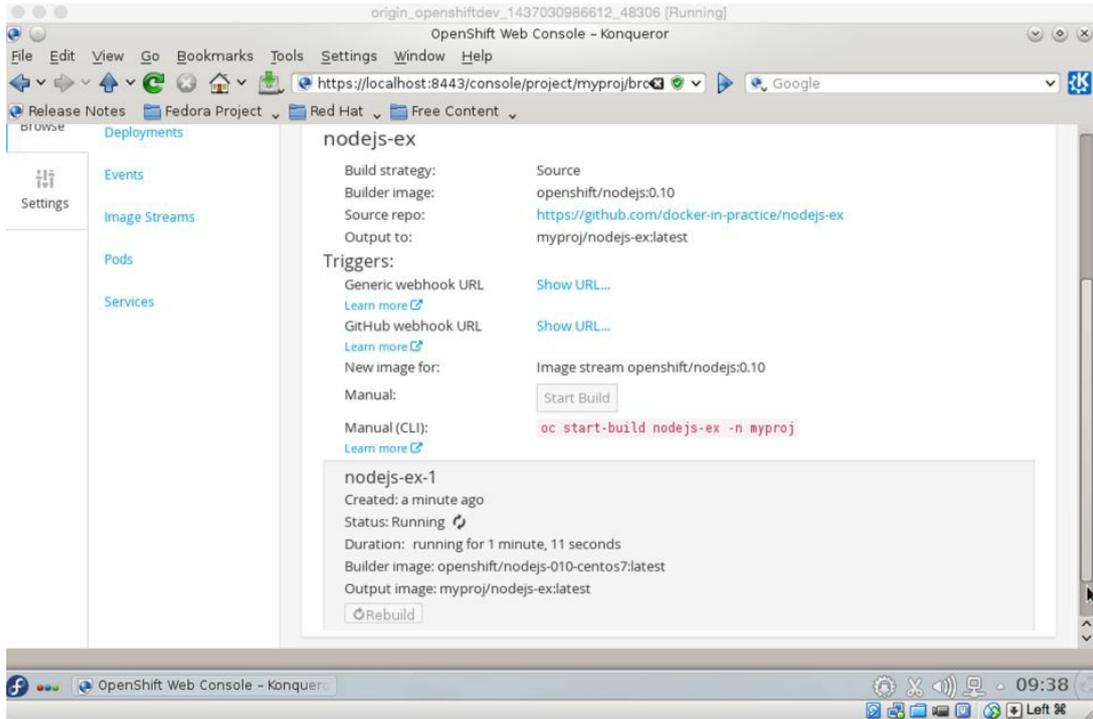


图10-10 OpenShift构建信息窗口

构建没有开始？ 在OpenShift早期版本中，构建有时不会自动开始。如果发生这种情况，几分钟后点击Start Build按钮即可。

过一会儿就能看到应用程序正在运行，如图10-11所示。

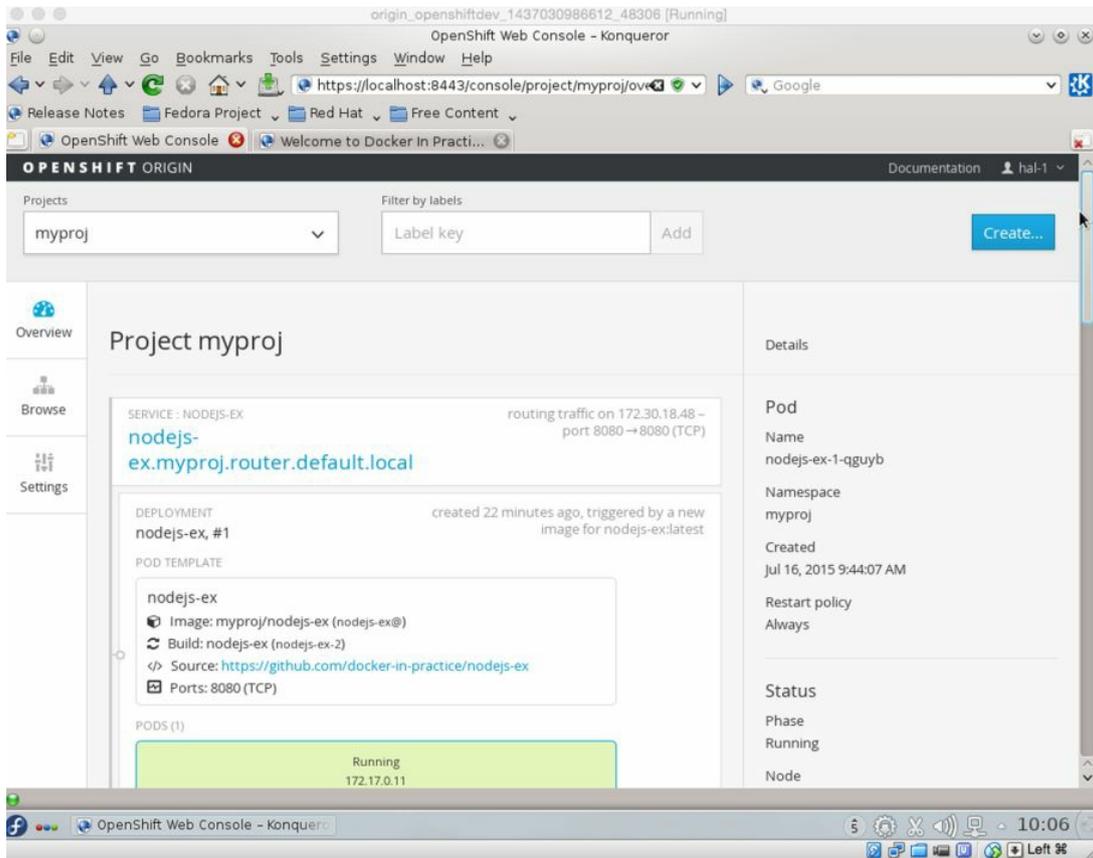


图10-11 程序运行页面

通过点击Browse和Pods，可以发现pod已经部署上去了，如图10-12所示。

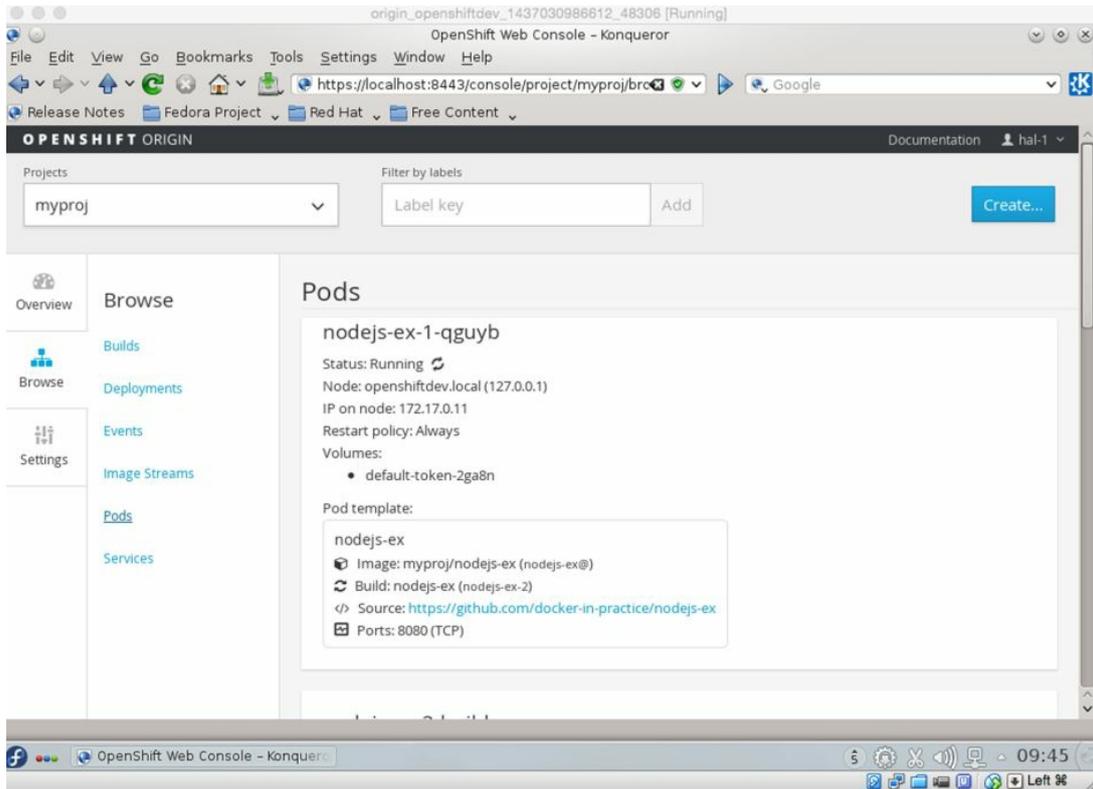


图10-12 OpenShift pod清单

什么是**pod** 关于什么是pod的解释见技巧79。

如何访问pod? 查看Services标签 (如图10-13所示), 可以看到用于访问的IP地址和端口号。

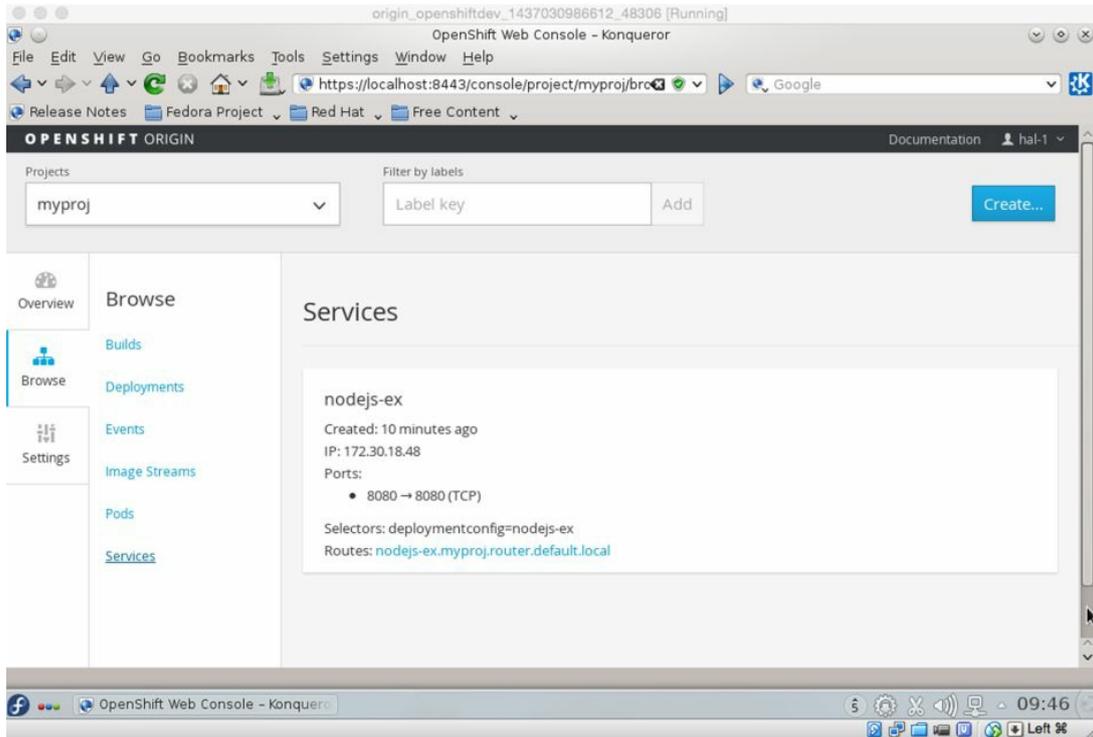


图10-13 OpenShift NodeJS应用程序服务细节

把浏览器指向这个地址，NodeJS应用程序就会运行起来，如图10-14所示。

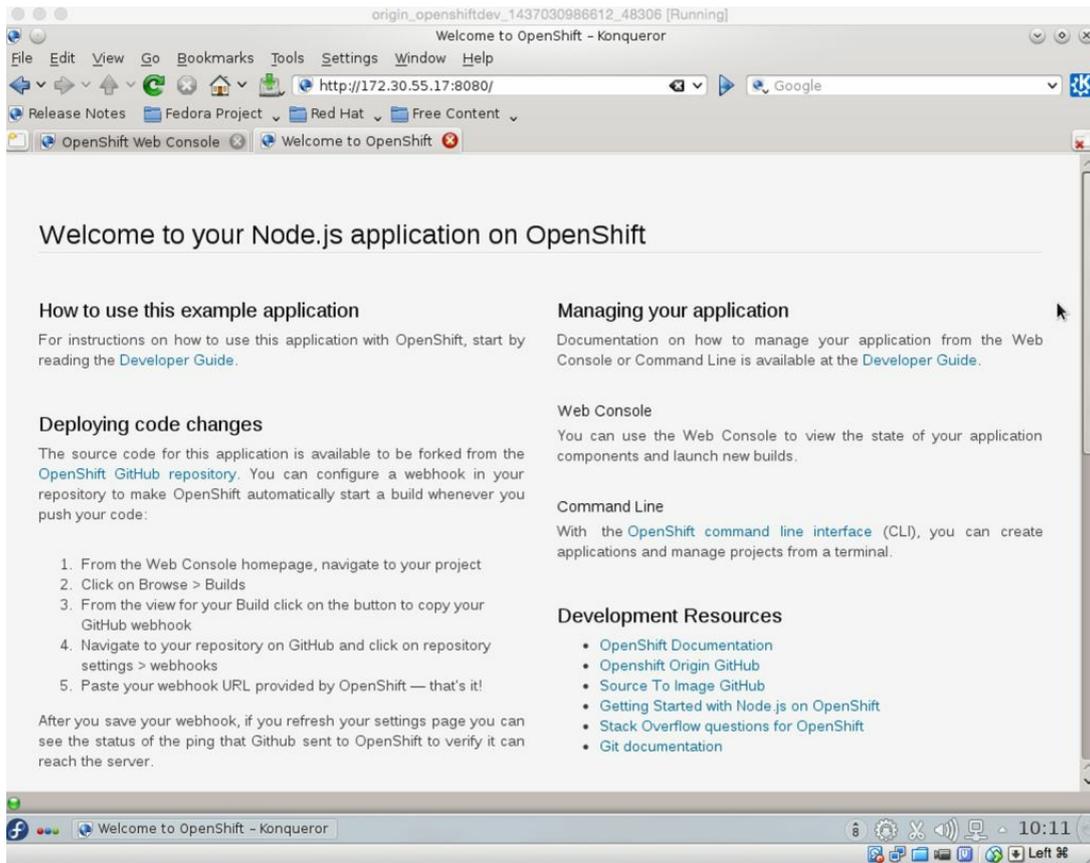


图10-14 NodeJS应用程序登录页

5. 总结

我们来总结一下到目前为止我们做到了什么以及它们对安全的重要性。

从用户的角度看，他们不用接触Dockerfile或者使用docker run命令，就登录了一个Web应用程序，并使用基于Docker的技术部署了一个应用程序。

OpenShift的管理员可以：

- 控制用户访问权限；
- 限制项目用到的资源；

- 集中供应资源；
- 确保代码默认不是以高权限运行的。

这比直接让用户运行`docker run`安全多了！

6. 接下来该怎么办

读者如果想在这个应用程序的基础上继续构建，了解 aPaaS 是如何促进迭代的，可以 fork 这个 Git 仓库，在 fork 的仓库里修改代码，然后创建一个新应用程序。我们已经完成了这些，参见 <https://github.com/docker-in-practice/nodejs-ex>。

要阅读更多关于 OpenShift 的资料，可以访问 <http://www.openshift.org>。

技巧88 使用安全选项

在之前的技巧里读者已经了解到了，默认情况下用户有 Docker 容器的 root 权限，这个用户和宿主机的 root 用户是一样的。为了改善这一点，我们展示了如何减少用户作为 root 用户的能力，以便即使他们脱离了容器，内核仍然不会允许有些操作执行。

然而，用户可以做更多事情。通过使用 Docker 的安全选项标志，用户可以防止宿主机上的资源受到容器内执行的操作的影响。这样就限制了容器仅能影响宿主机授权了的资源。

问题

想要保护宿主机不受容器操作的危害。

解决方案

使用内核支持的强制访问控制工具。

讨论

我们要使用SELinux作为强制访问控制（mandatory access control, MAC）工具。SELinux在某种程度上是工业标准，尤其受到关注安全的组织的青睐。它最初由NSA开发，用来保护他们的系统，随后被开源。它在基于Red Hat的系统上被当作标准使用。

SELinux是一个很大的话题，我们无法在本书中详尽讨论。接下来展示如何编写并实施一个简单的策略，以便读者能感受一下SELinux的工作方式。如果需要的话，读者可以更进一步或者做一些实验。

什么是**MAC**工具 除用户熟悉的标准安全规则之外，Linux中的强制访问控制（MAC）工具还执行很多。简而言之，它不仅确保文件和进程执行了常规读-写-执行规则，还可在内核级别对进程施加更细粒度的规则。例如，一个MySQL进程可能只允许在特定文件夹（如/var/lib/mysql）下写文件。基于Debian的系统上对应的标准是AppArmor。

本技巧假设用户有一个启用了SELinux的宿主机。也就是说，用户必须首先安装SELinux（如果没有安装好的话）。如果用户在运行Fedora或者其他基于Red Hat的系统，很可能已经装好了。

运行**sestatus**命令来确定是否启用了SELinux：

```
# sestatus
SELinux status:                enabled
SELinuxfs mount:              /sys/fs/selinux
SELinux root directory:      /etc/selinux
Loaded policy name:          targeted
```

```
Current mode:          permissive
Mode from config file: permissive
Policy MLS status:    enabled
Policy deny_unknown status: allowed
Max kernel policy version: 28
```

第一行输出会说明 SELinux 是否已经启用了。如果这个命令不可用，就说明宿主机上没有安装SELinux。

用户还需要一些相关的SELinux策略创建工具。例如，在一个能使用yum的机器上，需要运行`yum -y install selinux-policy-devel`。

1. Vagrant机器上的SELinux

如果没有SELinux又想要构建它的话，可以使用一个ShutIt脚本来在宿主机内构建提前安装好Docker和SELinuxd的虚拟机。图10-15大体介绍了它做了些什么。

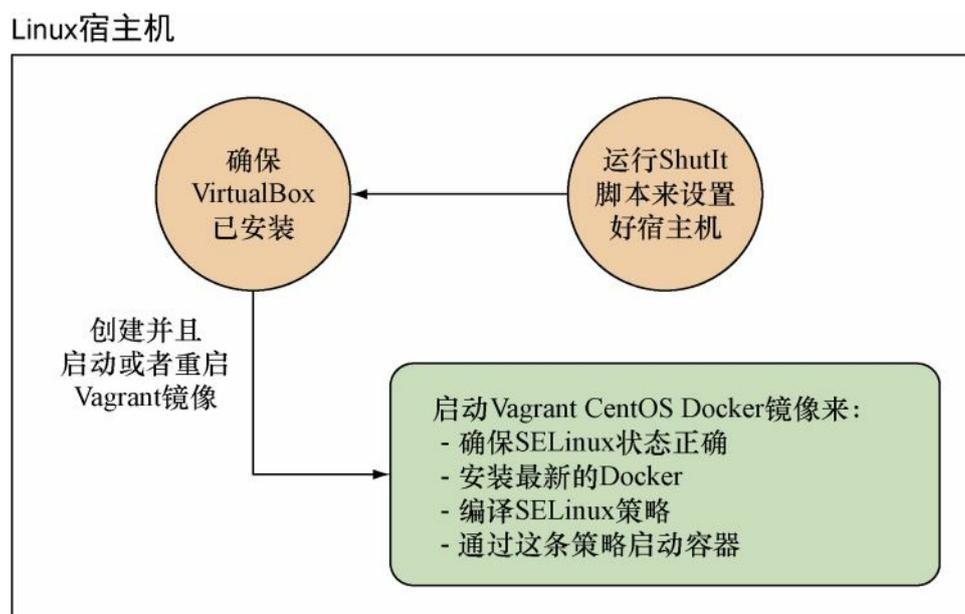


图10-15 提供SELinux虚拟机的脚本

什么是**ShutIt** ShutIt是一个我们原创的通用的shell自动化工具，可以突破Dockerfile的一些局限。如果想要了解更多，参见GitHub网页<http://ianmiell.github.io/shutit>。

图10-15中列出了建立策略所需的步骤。脚本会做以下几件事：

- (1) 创建虚拟机；
- (2) 启动一个合适的Vagrant镜像；
- (3) 登录这台虚拟机；
- (4) 确保SELinux的状态正确；
- (5) 安装最新版的Docker；
- (6) 安装SELinux策略开发工具；
- (7) 给你一个shell。

下面是用来设置和运行的它的命令（在Debian和基于Red Hat的发行版上测试过）：

```
sudo su - (1)
apt-get install -y git python-pip docker.io || \
yum install -y git python-pip docker.io (2)
pip install shutit (3)
git clone https://github.com/ianmiell/docker-selinux.git (4)
cd docker-selinux
shutit build --delivery bash \ (5)
-s io.dockerinpractice.docker_selinux.docker_selinux \
compile_policy no (6)
```

- (1) 在开始运行前确保你是root用户

- (2) 确保在宿主机上安装了所需要的包
- (3) 安装ShutIt
- (4) 复制SELinux ShutIt脚本并进入其目录
- (5) 运行ShutIt脚本。--delivery bash意味着命令是在bash中执行的而非通过SSH或者在Docker容器中执行
- (6) 设置脚本不要去编译SELinux策略，以为我们会手动做这一步

运行这个脚本之后，最后应该可以看到下面这样的输出：

```
Pause point:
Have a shell:
You can now type in commands and alter the state of the target.
Hit return to see the prompt
Hit CTRL and ] at the same time to continue with build
Hit CTRL and u to save the state
```

现在在虚拟机内有一个安装了SELinux的shell在运行了。如果输入sestatus，可以看到SELinux以宽容（permissive）模式开启（如代码清单10-3）。按Ctrl+]以返回宿主机的shell。

2. 编译SELinux策略

不管使用ShutIt脚本与否，我们都假设读者有一台启用了SELinux的宿主机。输入sestatus来获得一个状态汇总（如代码清单10-3所示）。

代码清单10-3 SELinux状态总结

```
# sestatus
```

```
SELinux status:          enabled
SELinuxfs mount:        /sys/fs/selinux
SELinux root directory: /etc/selinux
Loaded policy name:     targeted
Current mode:           permissive
Mode from config file:  permissive
Policy MLS status:     enabled
Policy deny_unknown status: allowed
Max kernel policy version: 28
```

我们处于宽容模式，也就是说SELinux会把违反安全的行为记录在日志里，但是不会强制实施。这样就可以安全地测试新策略而不会搞得系统没法用。用root身份录入**setenforce Permissive**来把SELinux的状态改为宽容状态。如果因为安全原因没法在自己的宿主机上这么做，不用担心，在代码清单10-4中有一个把策略设为宽容的选项。

在守护进程上设置**--selinux-enabled** 如果在宿主机上自行安装SELinux和Docker，一定要确保Docker守护进程设置了**--selinux-enabled**标志。读者可以使用**ps -ef | grep 'docker -d.*-- selinux-enabled**来检查，它应该会在输出中返回一些匹配结果。

给策略创建一个文件夹并进入这一文件夹，然后以root身份创建代码清单10-4所示的策略文件。这个策略文件包含我们将要采用的策略。

代码清单10-4 创建SELinux策略

```
mkdir -p /root/httpd_selinux_policy &&
➤ cd /root/httpd_selinux_policy (1)
cat > docker_apache.te << END (2)
policy_module(docker_apache,1.0) (3)
virt_sandbox_domain_template(docker_apache) (4)
allow docker_apache_t self: capability { chown dac_override kill setgid
```

```
➤ setuid net_bind_service sys_chroot sys_nice sys_tty_config} ; (5)
allow docker_apache_t self:tcp_socket (6)
➤ create_stream_socket_perms;
allow docker_apache_t self:udp_socket
➤ create_socket_perms;
corenet_tcp_bind_all_nodes(docker_apache_t)
corenet_tcp_bind_http_port(docker_apache_t)
corenet_udp_bind_all_nodes(docker_apache_t)
corenet_udp_bind_http_port(docker_apache_t)
sysnet_dns_name_resolve(docker_apache_t) (7)
#permissive docker_apache_t (8)
END (9)
```

(1) 创建一个保存策略文件的文件夹并且进入

(2) 使用“原地”文档来创建要编译的策略文件

(3) 使用policy_module指令创建SELinux策略模块docker_apache

(4) 使用提供的模板来创建docker_apache_t SELinux类型，它可以作为 Docker容器运行。这个模板给了docker_apache SELinux域运行起来的最小权限。我们会增加一些权限来让这个容器成为一个有用的环境

(5) Apache网络服务器要运行需要这些能力，所以用allow指令在这里添加这些能力

(6) 这些allow和corenet规则给了容器在网络上监听Apache端口的权限

(7) 使用sysnet指令允许DNS服务器解析

(8) 或者设置docker_apache_t类型为宽容模式，以便即使宿主机在强制施行SELinux这个策略也不会强制施行。无法设置宿主机的SELinux模式的时候使用这个

(9) 结束“原地”文档，将其写到磁盘

SELinux策略文档 为了获得关于前述授权的更多信息，了解其他授权，可以安装selinux-policy-doc包，然后用浏览器浏览位于file:///usr/share/doc/selinux-policy-doc/html/index.html的文档。这些文档在http://mcs.une.edu.au/doc/selinux-policy/html/templates.html上也提供。

现在编译策略，观察程序在强制模式下会启动失败。以宽容模式重启，检查违反情况并在之后改正：

```
$ make -f /usr/share/selinux/devel/Makefile \
docker_apache.te (1)
Compiling targeted docker_apache module
/usr/bin/checkmodule: loading policy configuration from
➤ tmp/docker_apache.tmp
/usr/bin/checkmodule: policy configuration loaded
/usr/bin/checkmodule: writing binary representation (version 17)
➤ to tmp/docker_apache.mod
Creating targeted docker_apache.pp policy package
rm tmp/docker_apache.mod tmp/docker_apache.mod.fc
$ semodule -i docker_apache.pp (2)
$ setenforce Enforcing (3)
$ docker run -ti --name selinuxdock
➤ --security-opt label:type:docker_apache_t httpd (4)
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
2a341c7141bd: Pull complete
[...]
Status: Downloaded newer image for httpd:latest
permission denied
Error response from daemon: Cannot start container
➤ 650c446b20da6867e6e13bdd6ab53f3ba3c3c565abb56c4490b487b9e8868985:
➤ [8] System error: permission denied
$ docker rm -f selinuxdock (5)
selinuxdock
$ setenforce Permissive (6)
$ docker run -d --name selinuxdock
➤ --security-opt label:type:docker_apache_t httpd (7)
```

(1) 把docker_apache.te文件编译为以.pp为后缀的二进制

SELinux模块

(2) 安装模块

(3) 将SELinux模式设置为“强制”

(4) 把httpd镜像作为守护进程运行，应用在模块里定义的docker_apache_t安全标签类型。这条命令会失败，因为它违反了SELinux安全配置

(5) 移除刚创建的容器

(6) 将SELinux模式设置为“宽容”，以允许程序启动

(7) 把httpd镜像作为守护进程运行，应用在模块里定义的docker_apache_t安全标签类型。这条命令会成功运行

3. 检查违反情况

到此为止，我们已经创建了一个SELinux模块并在宿主机上应用这一模块。因为在宿主机上SELinux的执行模式被设为了“宽容”，在“强制”模式里会被禁止的行为允许执行，同时会在审计日志里留下一条日志记录。可以通过运行以下命令来检查这些信息：

```
$ grep -w denied /var/log/audit/audit.log
type=AVC msg=audit(1433073250.049:392): avc: (1)
➤ denied { transition } for (2)
➤ pid=2379 comm="docker" (3)
➤ path="/usr/local/bin/httpd-foreground" dev="dm-1" ino=530204 (4)

➤ scontext=system_u:system_r:init_t:s0
➤ tcontext=system_u:system_r:docker_apache_t:s0:c740,c787 (5)
➤ tclass=process (6)
type=AVC msg=audit(1433073250.049:392): avc: denied { write } for
➤ pid=2379 comm="httpd-foregroun" path="pipe:[19550]" dev="pipefs"
```

```
➤ ino=19550 scontext=system_u:system_r:docker_apache_t:s0:c740,c787
➤ tcontext=system_u:system_r:init_t:s0 tclass=fifo_file
type=AVC msg=audit(1433073250.236:394): avc: denied { append} for
➤ pid=2379 comm="httpd" dev="pipefs" ino=19551
➤ scontext=system_u:system_r:docker_apache_t:s0:c740,c787
➤ tcontext=system_u:system_r:init_t:s0 tclass=fifo_file
type=AVC msg=audit(1433073250.236:394): avc: denied { open} for
➤ pid=2379 comm="httpd" path="pipe:[19551]" dev="pipefs" ino=19551
➤ scontext=system_u:system_r:docker_apache_t:s0:c740,c787
➤ tcontext=system_u:system_r:init_t:s0 tclass=fifo_file
[...]
```

(1) 在审计日志中的消息类型永远是AVC代表SELinux违反行为，时间戳表示为时代开始（定义为1970年1月1日）以来的秒数

(2) 花括号内展示了被拒绝的行为类型

(3) 处罚违反行为的进程ID和命令名

(4) 目标文件的路径、设备和i-node

(5) 目标的SELinux上下文

(6) 目标对象的类别

这里好多术语，我们没时间教读者关于SELinux的一切。如果读者想了解更多，可以从Red Hat的SELinux文档开始：

<http://mng.bz/QyFh>。

就现在来说，用户需要检查这些违反行为没有预见外的。什么是预见外的？例如，程序试图打开一个用户没打算让它打开的端口或者文件。接下来就要好好考虑我们要教的了：通过一个新的SELinux模块给这些违反行为打补丁。

在本例中，我们很高兴看到httpd可以写管道。我们已经弄明白了SELinux在拒绝做什么，因为提到的“拒绝”行为是向虚拟机的管道文

件执行append、write和open。

4. 给SELinux违反行为打补丁

一旦确定了看到的违反行文是可以接受的，有些工具就可以自动生成要应用的策略文件，因此就不用犯难又犯险地自己去写了。接下来的例子用了audit2allow工具来达成这一点：

```
mkdir -p /root/selinux_policy_httpd_auto (1)
cd /root/selinux_policy_httpd_auto (2)
audit2allow -a -w (3)
audit2allow -a -M newmodname create policy
semodule -i newmodname.pp (4)
```

(1) 创建一个用来存储新的SELinux模块的新目录

(2) 使用audit2allow工具来展示要通过读取审计日志生成的策略。检查一遍它的合理性

(3) 用-M标志和你为模块选的名字来创建模块

(4) 通过新创建的.pp文件来安装模块

重要的是要明白，我们新创建的这个SELinux模块，通过引用并且给docker_apache_t类型增加权限，“包含”（或者“需要”）并且改变了我们之前创建的那个模块。把二者结合到一个完整并独立的.te策略文件里就留给读者作为练习。

5. 测试新模块

安装好了新模块，就可以试一下重新启用SELinux并重启容器。

无法把强制模式设为宽容模式？ 如果之前无法把宿主机设为permissive模式（而且在原始的docker_apache.te文件里加入了讨论过的那一行），那么在继续之前要重新编译和重新安装原始的docker_apache.te文件（带上讨论过的那一行）。

```
docker rm -f selinuxdock
setenforce Enforcing
docker run -d --name selinuxdock \
--security-opt label:type:docker_apache_t httpd
docker logs selinuxdock
grep -w denied /var/log/audit/audit.log
```

审计日志中应该没有错误。应用程序在SELinux上下文的管控中启动了。

SELinux以复杂且难以管理闻名，有一句流传甚广的抱怨说人们经常关了它而不是调试它。这一点很不安全。虽然SELinux好的方面需要认真努力才能掌握，但是我们希望本技巧展示了在Docker不是开箱即用的情况下，如何创建一份安全专家可以审查乃至批准的东西。

10.4 小结

本章中我们从不同的角度解决了Docker中的安全问题。我们讨论了Docker中安全的基本问题，展示了解决这些问题的方式。你需要或者想要什么样的东西，取决于所在组织的性质以及对自己的用户的信任程度。

本章涉及以下内容：

- 使用SELinux降低容器以root身份运行的风险；
- 通过HTTP对Docker API的用户进行鉴权；
- 用证书为Docker API加密；

- 限制容器内root用户的能力；
- 使用应用程序平台即服务（aPaaS）来控制对Docker运行时的访问。

现在读者应该充分了解了Docker带来的安全问题，并了解了如何减轻它们。

接下来我们要把Docker带入生产环境，看一下在把Docker作为在线运维的一部分应该考虑的一些方面。

第11章 一帆风顺——生产环境中的 Docker以及运维上的考量

本章主要内容

- 记录容器日志输出的选项
- 如何监控运行中的容器
- 管理容器的资源使用
- 使用Docker的能力来帮助管理传统系统管理员的任务

本章中我们会讨论生产环境中会遇到的一些话题。在生产环境中运行Docker是一个很大的主题，而且Docker在生产中的用法还是一个发展中的领域。很多主要的工具还处于开发的早期阶段，就在我们写这本书的过程中都还在变化。例如，我们写本章的时候，Kubernetes更新到了1.0版本，Docker也发布了一个改版的注册中心（registry）。

本章中我们重点关注一些从动荡环境迁移到稳定环境时应该考虑的关键的事情。

11.1 监控

在生产环境中运行Docker首先要考虑的一点就是，如何追踪及测量容器所倚靠的事物。本节中我们会学到如何从运维的角度来考虑运行中的容器的日志活动及其性能。

这是Docker生态系统中仍在发展的一个方面，但是某些工具和技

术正在变得比其他的更为主流。我们将会谈及把应用程序的日志重定向到宿主机的syslog，把docker logs命令的输出重定向到一个集中的地方，以及Google公司的容器性能监控工具——cAdvisor。

技巧89 记录容器的日志到宿主机的syslog

Linux发行版通常会运行一个syslog守护进程。这个守护进程是系统日志记录功能的服务器端——应用程序给这个守护进程发送消息，以及一些类似消息重要性这样的元数据，这个守护进程就会决定存储消息的场所（如果要存储的话）。很多应用程序，从网络连接管理器到内核本身，都在使用这个功能来在遇到错误的时候转储信息。

正因为syslog是如此可靠且广泛使用，用户自己写的程序也应在它那里记录日志。但是，一旦用户对自己的程序进行了容器化，这种方法就行不通了（因为容器中默认没有syslog守护进程）。如果用户确实决定要在自己的所有容器中启动syslog守护进程，就要自己去每个容器里获取日志。

问题

想要在Docker宿主机上集中获取各个syslog。

解决方案

在宿主机上创建一个中心syslog守护进程容器，然后把syslog绑定挂载到一个集中的地方。

讨论

本技巧的基本思想就是要运行一个运行了syslog守护进程的服务容器，然后把日志接触点（/dev/log）通过宿主机的文件系统共享。日志本身可以通过查询syslog Docker容器来获得，并且存储在卷上。

图11-1对此进行了说明。

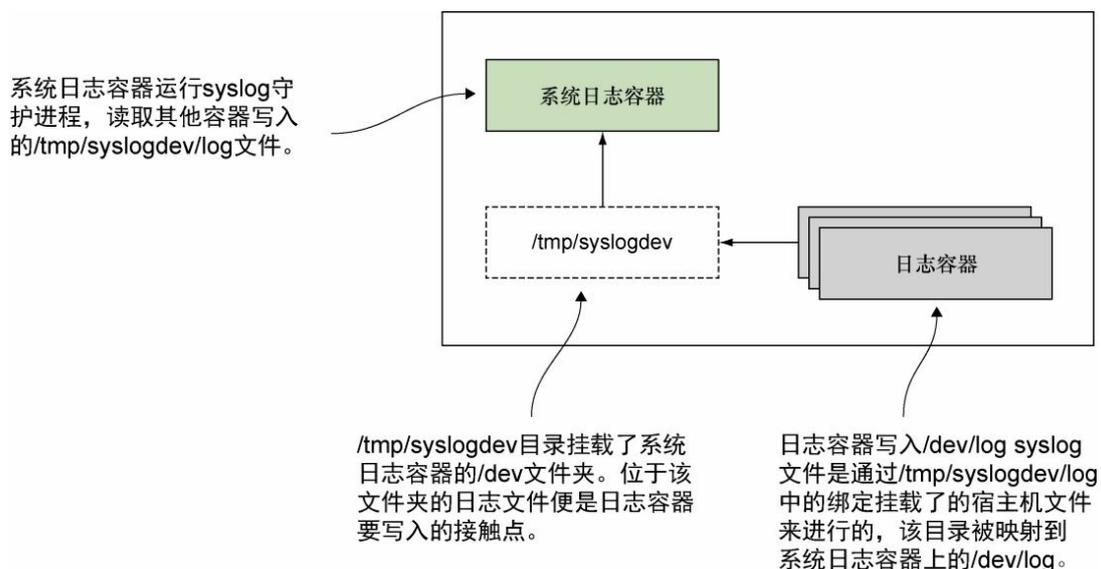


图11-1 Docker容器中心化系统日志概览

图11-1说明了宿主机文件系统上的/tmp/syslogdev是如何被运行其上的所有容器用作syslog活动的接触点的。日志容器挂载到该位置，并把syslog写入该位置。系统日志容器就收集全部的输入。

syslog 守护进程是什么 syslog守护进程是运行在服务器上的一种进程，它会收集并管理发送到一个中心文件（通常是一个Unix域套接字）的所有消息。它一般会使用/dev/log作为接收日志消息的文件，并且把日志记录到/var/log/syslog。

系统日志容器可以通过下面这个简单明了的Dockerfile来创建：

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install rsyslog (1)
VOLUME /dev (2)
VOLUME /var/log (3)
CMD rsyslogd -n (4)
```

(1) 安装rsyslog包，以使rsyslogd守护进程程序可用。“r”代表

reliable（可靠的）

- （2）创建/dev卷来与其他容器共享
- （3）创建/var/log卷来允许syslog文件持久保存
- （4）在启动时运行rsyslogd进程

接下来构建容器，为其打上syslogger标签，并且运行它：

```
docker build -t syslogger
docker run -name syslogger -d -v /tmp/syslogdev:/dev syslogger
```

把容器的/dev文件夹绑定挂载到宿主机的/tmp/syslogdev文件夹，然后就可以把/dev/log套接字作为一个卷挂载到每个容器上，马上就会看到效果了。这个容器会在后台继续运行，读取所有来自/dev/log文件的信息并且处理之。

在宿主机上现在可以看到系统日志容器上的/dev 文件夹已经挂载到了宿主机的/tmp/syslogdev文件夹：

```
$ ls -l /tmp/syslogdev/
fd
full
fuse
kcore
log
null
ptmx
random
stderr
stdin
stdout
tty
urandom
zero
```

在这个演示中，将要启动100个守护进程容器，它们会把自己的

启动顺序使用**logger**命令按从0到100记录到**syslog**。然后就能在宿主机上运行**docker exec**来检查系统日志容器的**syslog**文件以查看这些消息了。

首先，启动容器：

```
for d in {1..100}
do
  docker run -d -v /tmp/syslogdev/log:/dev/log ubuntu logger hello_$d
done
```

上述的卷挂载操作把容器的**syslog**端点（**/dev/log**）链接到了宿主机的**/tmp/syslogdev**文件，而该文件转而映射到了系统日志容器的**/dev/log**文件。通过这样的搭线，所有的**syslog**输出都会送到同一个文件。

该操作完成的时候，我们会看到类似于下面这样的输出（本输出经过编辑）：

```
$ docker exec -ti syslogger tail -f /var/log/syslog
May 25 11:51:25 f4fb5d829699 logger: hello
May 25 11:55:15 f4fb5d829699 logger: hello_1
May 25 11:55:15 f4fb5d829699 logger: hello_2
May 25 11:55:16 f4fb5d829699 logger: hello_3
[...]
May 25 11:57:38 f4fb5d829699 logger: hello_97
May 25 11:57:38 f4fb5d829699 logger: hello_98
May 25 11:57:39 f4fb5d829699 logger: hello_99
```

如果用户希望，可以通过修改**exec**命令来归档这些**syslog**。例如，可以运行下面的命令来获取所有5月25日11时的文档并归档到一个压缩文件里：

```
$ docker exec syslogger bash -c "cat /var/log/syslog | grep '^May 25 11'
'" | \
xz - > /var/log/archive/May25_11.log.xz
```

应用程序必须记录日志到**syslog**。为了让这些消息在中心系统日志容器中出现，程序需要记录日志到**syslog**。我们在这里通过运行**logger**命令来确保这一点，然而你的应用程序也要执行一样的操作来使之工作。大多数现代的记录日志方法都有写入本地可见的**syslog**的手段。

读者可能想知道，通过本技巧如何区分不同容器的日志消息。这里有两种选择：一是可以把应用程序的日志信息改为输出容器的宿主机名，二是可以了解下一个技巧让**Docker**来做这个粗重活。

syslog驱动程序是另一回事。本技巧看起来和下一个使用**Docker syslog**驱动程序的技巧类似，但是它们不是一码事。本技巧让容器运行进程的输出作为**docker logs**命令的输出，但下一个技巧接管了**logs**命令，使本技巧显得多余。

技巧90 把**Docker**日志发送到宿主机的输出系统

正如所见，**Docker**提供了基本的日志系统来获取用户的容器的启动命令的输出。如果你是一名在不止一个宿主机上运行着多个服务的系统管理员，那么手工轮流在每个容器上执行**docker logs**命令来跟踪并获取日志操作起来可能很是烦琐。

在本技巧中，我们会讲解一下**Docker**的日志驱动程序特性。这让我们能够使用标准的日志系统来追踪某个（乃至跨多个）宿主机上的很多服务。

问题

想要在**Docker**宿主机上集中获取**docker logs**的输出。

解决方案

启动一个日志驱动程序来在其他地方获取docker logs的输出。

讨论

在默认情况下，Docker日志是在Docker守护进程内部被捕获的，可以通过docker logs命令来查看。读者可能已经注意到了，这展示的是容器主进程的输出。

编写本书时，Docker为重定向该输出提供了若干选择，包括：

- syslog;
- journald;
- json-file。

默认设置是json-file，但是其他两项也可以通过--log-driver命令来选择。syslog和journald选项会把日志输出发送到各自同名的守护进程。在<https://docs.docker.com/engine/reference/logging/>可以找到所有可用的日志驱动程序的官方文档。

版本依赖 本技巧需要Docker 1.6.1或更高版本。

syslog守护进程是运行在服务器上的一个进程，它会收集并管理发送到一个中心文件（通常是一个Unix域套接字）的所有消息。它一般会使用/dev/log作为接收日志消息的文件，并且把日志记录到/var/log/syslog。

journald是一个收集并存储日志数据的系统服务。它为不同来源的日志创建并维护一个结构严密的索引。这些日志可以通过

journalctl命令来查询。

1. 记录日志到syslog

为了把输出定向到syslog，需要使用--log-driver标志：

```
docker run --log-driver=syslog ubuntu echo 'outputting to syslog'
outputting to syslog
```

这会将输出记录在syslog文件中。如果有访问该文件的权限，可以通过标准Unix工具来检查这些日志：

```
$ grep 'outputting to syslog' /var/log/syslog
Jun 23 20:37:50 myhost docker/6239418882b6[2559]: outputting to sysl
```

2. 记录日志到journald

输出到journal守护进程看起来类似这样：

```
$ docker run --log-driver=journald ubuntu echo 'outputting to journald'
outputting to syslog
$ journalctl | grep 'outputting to journald'
```

运行**journal**守护进程？ 确保在运行前面的命令之前宿主机上有一个journal守护进程在运行。

3. 对所有的容器应用这些命令

为宿主机上所有的容器应用这一参数可能会十分费力，因此设置Docker守护进程默认输出日志到这些受支持的机制。

更改守护进程的/etc/default/docker，或者/etc/sysconfig/docker，或者用户自己的发行版设置的Docker配置文件。激活

DOCKER_OPTS=""这一行，添加--log-driver标志。例如，如果这一行是

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
```

就变为

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 --log-driver syslog"
```

Docker中的配置变化 参见附录B以了解如何在宿主主机上改变Docker守护进程的配置。

如果重启了Docker守护进程，容器会记录日志到相关服务。

另一个在这种情境下值得一提的通常做法（在这里未做讲解）是可以使用容器实现一个ELK（Elasticsearch, Logstash, Kibana）日志基础设施。

破坏Docker日志命令 把守护进程的设置改成json-flie或者journald之外的任何东西都将意味着默认情况下标准的docker logs命令将不起作用了。此Docker守护进程的用户可能并不喜欢这个变化，尤其是/var/log/syslog文件（被syslog驱动程序使用）通常对非root用户来说是无法访问的。

技巧91 使用cAdvisor监控容器

一旦在生产环境中有一系列的容器运行，那么用户可能会想要像

多个进程运行在宿主机上一样能够监控它们的资源利用和性能。

监控领域中（包括总体上来讲，以及考虑Docker）有众多候选的热门地带。之所以选择cAdvisor是因为它受众很广。它是一个由Google公司开源的快速流行起来的项目。如果之前用过传统的宿主机监控工具，如Zabbix或者Sysdig，那么值得看一下它们是否已经提供了所需的功能——编写本书时很多工具都在添加面向容器的功能。

问题

想要监控容器的性能。

解决方案

使用cAdvisor。

讨论

cAdvisor 是一个由Google公司开发的用来监控容器的工具。它在GitHub上开源：[https:// github.com/google/cadvisor.](https://github.com/google/cadvisor)。

cAdvisor作为一个收集运行中的容器的性能数据的守护进程运行。在其他事情中，它会监测：

- 资源隔离参数；
- 历史资源使用；
- 网络统计数据。

cAdvisor可以在宿主机上原生安装或者作为Docker容器运行：

```
$ docker run \  
--volume /:/rootfs:ro \ (1)  
--volume /var/run:/var/run:rw \ (2)  
--volume /sys:/sys:ro \ (3)  
--volume /var/lib/docker/:/var/lib/docker:ro \ (4)
```

```
-p 8080:8080 -d --name cadvisor \ (5)
--restart on-failure:10 google/cadvisor (6)
```

(1) 给予cAdvisor对root文件系统的只读权限，以便它检测宿主机的信息

(2) 以读写权限挂载/var/run文件夹。大多数情况下，每个宿主机上应当运行一个cAdvisor实例

(3) 给予cAdvisor对宿主机的/sys文件夹的只读权限，其中包含内核子系统和挂载到宿主机的设备的信息

(4) 给予cAdvisor对Docker宿主机目录的只读权限

(5) cAdvisor的网络接口在端口8080提供服务，我们在宿主机上也在同一端口发布。用来在后台运行容器并给容器取名的标准的Docker参数也会被使用

(6) 在失败时重启容器最多十次。这个镜像是用Google的账户存储在Docker Hub上的

一旦启动了镜像，就可以通过浏览器访问<http://localhost:8080>以检查数据输出。这里有一些宿主机的信息，但是点击主页顶部的Docker Containers链接，可以通过点击Subcontainers标题下列出的容器检查CPU、内存和其他历史数据。

容器运行时数据被收集并在内存中保存。在GitHub页面上有一个在InfluxDB里保存数据的文档。那个GitHub仓库里还有REST API的细节以及一个用Go语言编写的样例客户端程序。

什么是**InfluxDB** InfluxDB是一个设计用来处理时间序列数据的轨

迹的开源数据库。因此它对记录和分析实时提供的监控信息来说很理想。

11.2 资源控制

在生产环境里运行服务的一个中心问题是公平有效的资源分配。在底层，Docker使用了核心操作系统的cgroup概念来管理容器的资源使用。容器进行资源竞争的时候，默认使用简单均分算法。但是有时候这还不够。出于运维上的考虑，用户可能想保留或者限制某个容器或者某类容器的资源。

本节中，我们会学习如何对容器的CPU和内存使用进行调优。

技巧92 限制容器可以运行的内核

默认情况下，Docker可以在机器的任意内核上运行。只有一个进程和线程的容器明显最多只能耗尽一个内核，但是容器中的多线程程序（或者多个单线程程序）可以使用CPU上所有的内核。如果有一个容器比其他容器都重要，用户可能想对这个行为进行修改——面向客户的程序每次都要在内部日常报告系统运行的时候争抢CPU可不太好。本技巧还可以用于防止失控的容器把用户挡在服务器的SSH之外。

问题

想要让容器有最小CPU分配额，对CPU消耗有硬性的限制，或者想要限制可以运行容器的内核。

解决方案

使用 `--cpuset-cpus` 选项来保留内核为容器所用。

讨论

在多核计算机上用户需要遵循本技巧以合理探索 `--cpuset-cpus` 选项。如果使用的是云机器可能遇不到这种情况。

重命名了的标志 `--cpuset` 老版本的Docker使用的是现在已经弃用的 `--cpuset` 标志。如果 `--cpuset-cpus` 不工作，可以试一下 `--cpuset`。

我们将使用 `htop` 命令来看一下 `--cpuset-cpus` 选项的作用，这条命令会给出计算机内核使用的有用图形。请在继续之前确保这个命令已经装好——通常从系统包管理器以 `htop` 包的形式提供。或者，可以在一个以 `--pid=host` 选项启动的Ubuntu容器里安装它，这样就可以把宿主机的信息暴露给容器。

如果现在运行 `htop`，大概不会看到任何内核处于忙碌状态。在两个不同的终端里运行以下命令，以模拟多个容器内部的负载：

```
docker run ubuntu:14.04 sh -c 'cat /dev/zero >/dev/null'
```

现在回头来看 `htop`，可以看到有两个内核显示出 100% 使用率。为了限制到一个内核上，`docker kill` 之前的容器，然后在两个终端里运行以下命令：

```
docker run --cpuset-cpus=0 ubuntu:14.04 sh -c 'cat /dev/zero >/dev/null'
```

现在 `htop` 会显示出这些容器只使用了第一个内核。

--cpuset-cpus选项允许通过逗号分隔的列表（0,1,2）、范围（0-2）或者两者结合的方式（0-1,3）来指定多个内核。因此为宿主机保留CPU就是在给容器选范围的时候排除一个内核的事儿了。

这个功能可以以多种方式使用。例如，可以通过不断分配剩余CPU给运行中的容器的方式，来为宿主机进程保留特定的CPU；也可以限制特定的容器运行在各自独立的CPU上，从而防止它们干扰其他容器所用的计算。

技巧93 给重要的容器更多CPU

宿主机上的容器一般在竞争的时候平分CPU使用。读者已经了解了如何做出绝对监管或者限制，但是那些有点儿太不灵活了。如果想让一个进程比其他进程使用更多的CPU，一直为它保留整个内核就太浪费了。如果只有几个内核，这样做难免受限制。

对于想把应用程序部署到共享服务器的用户，Docker创造了多租户的便利条件。这可能会导致在有经验的虚拟机用户中有名的“吵闹的邻居”问题，一个用户耗尽了所有的资源，影响了在同一硬件上工作的其他虚拟机。

一个具体的例子是，在写本书时，我们不得不采用这个功能来减少一个尤其贪得无厌的Postgres应用程序的资源占用，它耗尽了CPU周期，让网络服务器无法为终端用户服务。

问题

想要给重要的容器更多的CPU份额，或者把一些容器标记为不那么重要。

解决方案

为docker run命令添加-c/--cpu-shares参数以定义CPU相对使用份额。

讨论

当一个容器启动起来的时候，它会得到一个CPU份额的数值（默认是1024）。当只有一个进程在运行的时候，如果有必要它对CPU可以有100%的使用权，不管它有多少的CPU份额。只有当和其他容器竞争CPU的时候，这个数值才有用。

设想我们有3台容器（A、B和C）同时都在试图使用所有可用的CPU资源：

- 如果它们的CPU份额相等，那么每个都可以分配到1/3的CPU；
- 如果A和B拿到512，C拿到1024，那么C获得CPU的一半，A和B各得1/4；
- 如果A拿到10，B拿到100，C拿到1000，A拿到可用CPU资源的1%不到，并且只有在B和C空闲的时候才能做一些资源消耗较大的事。

以上所有都假设容器可以使用机器上的所有内核（或者只有一个内核）。Docker会尽量把来自容器的负载分配到所有的内核上。如果有两个容器在一个双核机器上运行着单线程应用程序，那么明显无法在应用相对权重的同时最大化使用可用资源。每个容器都会分配到一个在其上执行的内核，不管权重是多少。

如果想试一下，运行下面的命令：

```
docker run --cpuset-cpus=0 -c 10000 ubuntu:14.04 \
sh -c 'cat /dev/zero > /dev/null' &
docker run --cpuset-cpus=0 -c 1 -it ubuntu:14.04 bash
```

现在看看在bash里执行操作是多么缓慢。注意，这些数值是相对的，可以把它们全部乘10（举例来说），它们代表的意思仍旧完全相同。然而，默认得到的仍然是 1024，所以当修改这些数值的时候，应当考虑一下在同一套CPU上运行一个没有指定份额的进程会怎么样。

选择设置 为用例选择合适的CPU层级是一门艺术。值得看一下top和vmstat一类程序的输出，看看什么在用CPU时间。使用top的时候，按“1”键展示每个CPU各自在做什么尤其有用。

技巧94 限制容器的内存使用

运行容器的时候，Docker 会允许它从宿主机分配尽可能多的内存。通常这是我们想要的效果（也是与虚拟机相比的巨大优势，虚拟机分配内存的方式并不灵活）。但是有时候应用程序可能会脱离控制，分配了太多内存，当机器开始交换内存的时候就会死机。这很烦人，我们以前也发生了很多次。我们想要一种能限制容器内存消耗的方式来防止这件事。

问题

想要限制容器的内存消耗。

解决方案

对docker run使用-m/--memory参数。

讨论

如果正在运行Ubuntu，很可能内存限制能力默认并没有启用。运行docker info来检查。如果输出中有一行在警告No swap limit

support，就要先做些准备工作。注意，这些改变可能对机器上所有的应用程序都有性能影响，参见Ubuntu安装文档，获取更多信息（<http://docs.docker.com/engine/installation/ubuntu/linux/#adjust-memory-and-swap-accounting>）。

简单来说，需要在启动时就告诉内核，希望这些限制可用。为了达到这个目的，需要如下修改/etc/default/grub。如果GRUB_CMDLINE_LINUX已经有值了，在末尾加上新的：

```
-GRUB_CMDLINE_LINUX=""  
+GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

现在需要运行sudo update-grub并重启计算机。运行docker info应该不会再得到警告了，现在可以继续正题了。

首先，简单粗暴地展示一下使用4 MB的最低可能内存限制确实有用：

```
$ docker run -it -m 4m ubuntu:14.04 bash (1)  
root@cfffc126297e2:/# \  
python3 -c 'open("/dev/zero").read(10*1024*1024)' (2)  
Killed (3)  
root@e9f13cacd42f:/# \  
A=$(dd if=/dev/zero bs=1M count=10 | base64) (4)  
$ (5)  
$ echo $? (6)  
137 (7)
```

- (1) 以4 MB内存限制运行容器
- (2) 试着把10 MB加载到内存
- (3) 进程消耗了太多内存，因此被杀死
- (4) 试着把10 MB的内存直接加载bash

(5) `bash`也被杀死了，容器因而退出

(6) 检查退出码

(7) 退出码非零，表明容器因错退出

这种限制有一个缺陷。为了展示这一点，我们要使用 `jess/stress` 镜像，这一镜像里面包含有 `stress`，一个用来测试系统限制的工具。

轻松压力测试 `jess/stress` 是一个用来测试在容器上施加的资源限制的有用镜像。想要实验更多的话，用这个镜像试一下之前的技巧吧。

如果运行下面的命令，你会惊讶地发现它立刻就不存在了：

```
docker run -m 100m jess/stress --vm 1 --vm-bytes 150M --vm-hang 0
```

你已经让 Docker 限制容器到 100 MB 了，已经让 `stress` 占用 150 MB 了。可以通过运行下面的命令检查 `stress` 正在如期运行：

```
docker top <container_id> -eo pid,size,args
```

大小 (`size`) 那一栏是以 KB 为单位显示的，展示出容器确实占用了 150 MB 内存.....问题来了，为什么它还没有被杀死呢！原来是 Docker 双重保留了内存，一半给物理内存一半用于内存交换。如果使用下面的命令，容器会立刻终止：

```
docker run -m 100m jess/stress --vm 1 --vm-bytes 250M --vm-hang 0
```

这种双重保留机制是默认设置，可以通过 `--memory-swap` 参数来控制，该参数指定了总体虚拟内存的大小（内存+交换）。例如，

想要完全消除交换内存的使用，应当把`--memory`和`--memory-swap`设定为同一大小。在Docker运行参考中可以找到更多的例子：

<https://docs.docker.com/engine/reference/run/#user-memory-constraints>。

11.3 Docker的系统管理员用例

本节中我们要看一下Docker的一些令人惊奇的用法。尽管第一眼看上去可能会觉得奇怪，但Docker确实可以让cron作业管理更加轻松，还可以用作一种备份工具。

什么是**cron**作业 cron作业是一个定时定期运行的命令，它由一个几乎所有Linux系统都作为服务包含的守护进程运行。每个用户可以指定自己要运行的命令的周期。它通常被一些系统管理员重度使用来运行一些周期性任务，如清理日志文件或者运行备份。

我们在这里不可能穷尽它的可能用法，但是可以让读者对Docker的灵活性有一点儿感觉，并对其特性能够如何被用在意想不到的地方有所了解。

技巧95 使用Docker来运行cron作业

如果读者在多个宿主机上管理过cron作业，那么可能遇到过这样的难题，要在不同地方部署同一套软件，还要确保crontab本身正确调用想要运行的程序。

尽管这个问题有其他的解决方案（例如，使用Chef、Puppet、Ansible或者其他配置管理工具来管理跨宿主机的软件部署），但有一个选项是可以用Docker注册中心来存储正确的调用。

虽然对于这个问题这并不总是最好的解决方案，但是它极好地展示出了有一个对应用程序运行时配置隔离又便携的存储是多么好。如果已经在用Docker了这还是免费的。

问题

想要cron作业能被集中管理和自动更新。

解决方案

把cron作业脚本作为Docker容器来拉取和运行。

讨论

如果有很多机器要定期运行作业，一般会使用crontab并手动配置（没错，还有人这么干），或者用一个类似Puppet或者Chef这样的配置管理工具。更新其配置脚本可以确保当机器的配置管理控制程序下次运行的时候这些变化都会应用到crontab上，为下次运行做好准备。

什么是**crontab** crontab文件是一种由用户维护的特殊文件，它指定了脚本应该运行的次数。通常都是些维护任务，如压缩和归档日志文件，但是也可能是商业上很重要的应用程序，如信用卡付款结算程序。

本技巧中，我们会展示如何通过从Docker注册中心中拉取镜像的方式取代这些把戏。

如图11-2展示的一般情况，维护者更新配置管理工具，接下来在有代理（agent）运行的时候这些工具就被分发到服务器上。同时，系统更新时，cron作业运行着新旧代码。

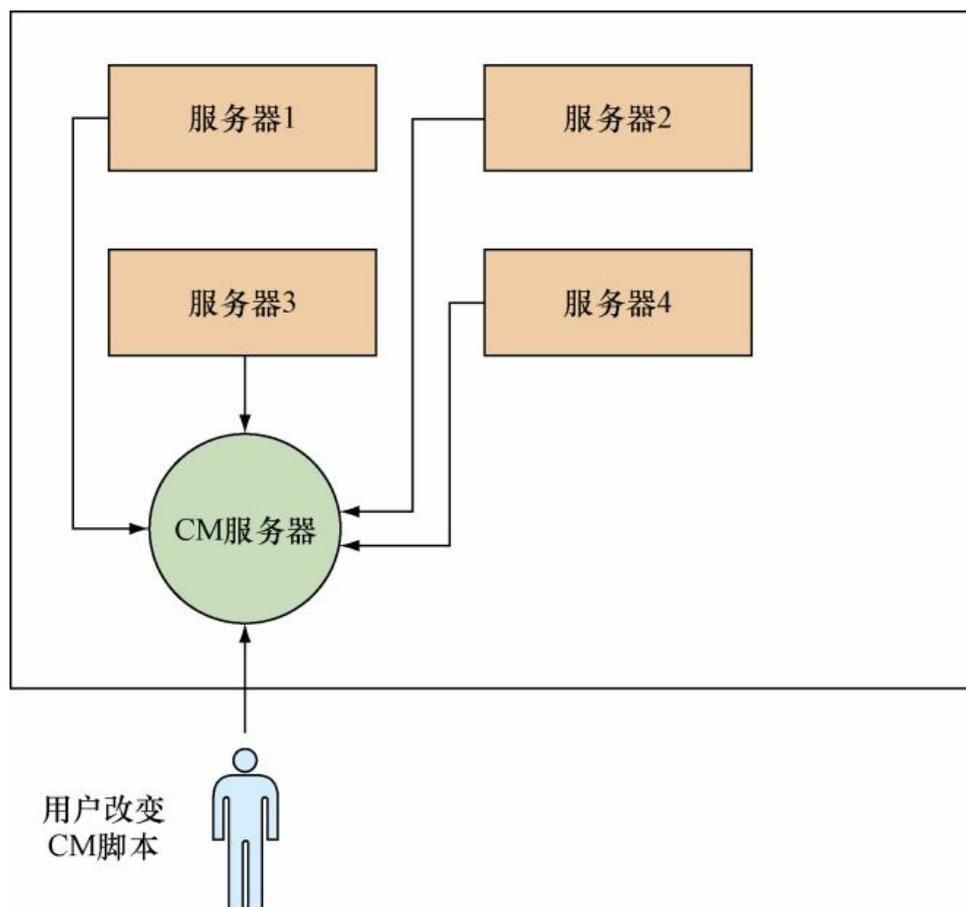


图11-2 在CM代理定期运行中每个服务器都更新cron脚本

在图11-3描绘的Docker场景下，在cron作业运行前每个服务器都去拉取了最新版的代码。

到此为止读者可能会纳闷，如果已经有了能工作的解决方案，为什么要这么麻烦呢。把Docker作为交付机制有以下一些好处：

- 当作业运行的时候，它会从中央地点把自己更新到最新版；
- crontab文件变得更加简单，因为脚本和代码都在Docker镜像里封装了；
- 对于更加庞大复杂的变化，只有Docker镜像变化中的差值需要拉取，加速了交付和更新的速度；
- 不用在机器本身上维护代码或者二进制文件；

- 可以把Docker和其他技术结合，如把输出定向到syslog，以简化并集中化对这些管理服务的管理。

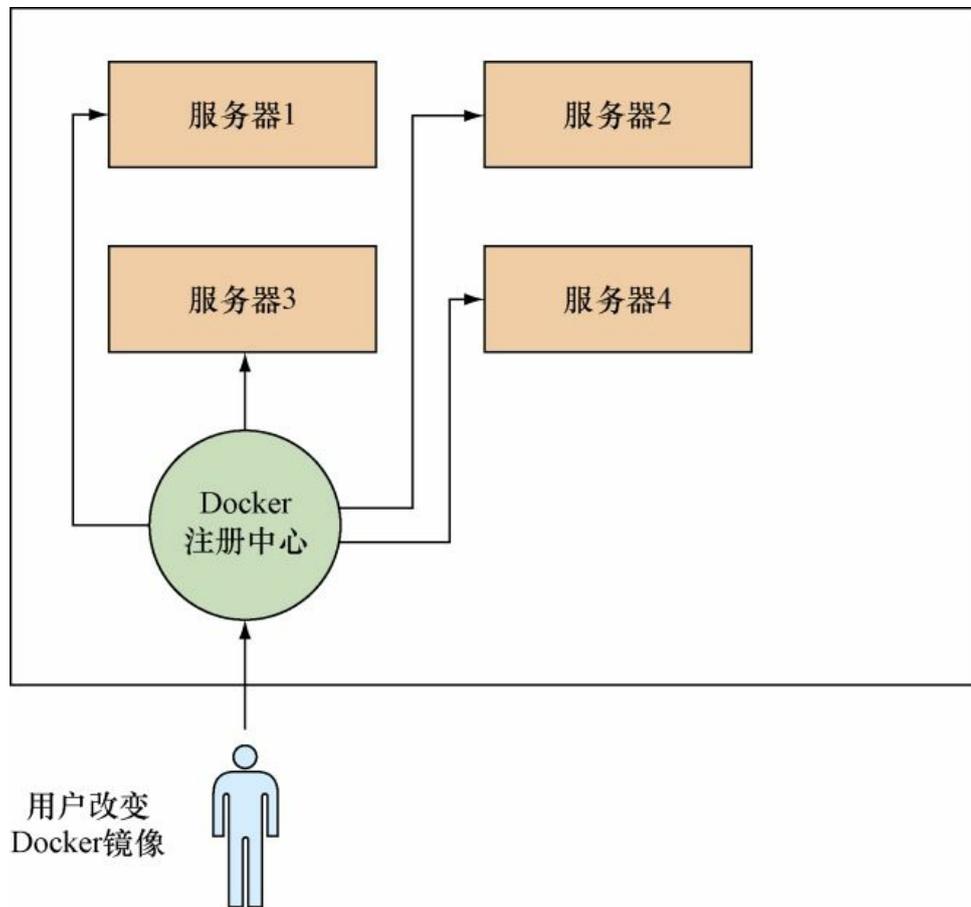


图11-3 在每个cron作业运行前，每个服务器拉取最新的镜像

在这个例子中，我们要使用在技巧40中创建的log_cleaner镜像。回忆一下，这个镜像封装了一个脚本，该脚本会清理服务器上的日志文件并且接收一个指定清理多少天的日志的参数。使用Docker作为交付机制的crontab大概看起来像下面这样：

```
00***\ (1)
docker pull dockerinpractice/log_cleaner && \ (2)
docker run \ (3)
-v /var/log/myapplogs:/log_dir dockerinpractice/log_cleaner 1
```

(1) 每天午夜运行

(2) 先拉取最新版本的镜像

(3) 运行日志清理程序清理一天的日志文件

不熟悉 **cron**? 如果对 **cron** 不是很熟悉, 想要了解如何编辑 **crontab** 的话, 可以运行 **crontab -e**。每行都让一条命令在行首的5个值指定的时间运行。在 **crontab** 的帮助手册可以找到更多信息。

如果有故障发生, 那么应当触发发送邮件的标准的 **cron** 机制。如果不信任这个, 就用 **or** 操作符添加一条命令。接下来的例子中, 我们假设读者定制的报警命令是 **my_alert_command**:

```
0 0 * * * (docker pull dockerinpractice/log_cleaner && \  
docker run -d -v /var/log/myapplogs:/log_dir \  
dockerinpractice/log_cleaner 1) \  
|| my_alert_command 'log_cleaner failed'
```

什么是 **or** 操作符 (本例中是双竖线 **||**) 确保了至少两边的命令有一条得到执行。如果第一个命令失败了 (本例中, 是在 **cron** 对时间的指定 **0 0 * * *** 之后括号中由与操作符 **&&** 连接起来的两条命令中的一条), 那么第二条就会执行。

|| 操作符确保了如果日志清理作业的任一部分失败了, 都会触发报警命令。

技巧96 通过“保存游戏”的方法来备份

如果读者运行过交易系统, 那么就会知道, 当有差错的时候, 在出问题的时候推断系统状态的能力对于源头分析至关重要。

通常它是通过一些方法的结合来达成的：

- 应用程序日志分析；
- 数据库法医学（确定在给定时间点的数据状态）；
- 构建历史分析（确定在给定的时间点，什么样的代码和配置在服务器上运行过）；
- 生产系统分析（例如，有没有人登录并改动过什么东西）。

对这样重要的系统，可以采取简单有效的备份Docker服务容器的办法。尽管可能数据库是和Docker基础设施分离的，配置、代码和日志的状态都可以通过几个简单的命令存储在注册中心里。

问题

想要保存Docker容器的备份。

解决

在容器运行的时候提交，然后作为一个单独的仓库推送。

讨论

遵循Docker最佳实践，利用一些Docker特性可以不用存储容器备份。例如，使用技巧90中的日志驱动程序而不是把日志记录到容器的文件系统里，意味着不用从容器备份里获取日志。

但是，有时候用户不得被迫做一些不喜欢做的事情，必须看一下容器是什么样的。接下来的命令展示了提交和推送备份容器的整个过程：

```
DATE=$(date +%Y%m%d_%H%M%S) (1)
TAG="your_log_registry:5000/live_pmt_svr_backup:$(hostname -s)_${DATE}"
(2)
docker commit -m="${DATE}" -a="Backup Admin" live_pmt_svr $TAG (3)
```

- (1) 产生一个记录到秒的时间戳
- (2) 通过带有宿主机名和日期的标签产生一个指向你的注册中心URL的标签
- (3) 以日期为消息，以Backup Admin为作者，提交容器
- (4) 把容器推送到注册中心

停止容器服务 本技巧会在容器运行时将它暂停，高效地停止它的服务。你的服务要么可以忍受中断，要么你应该以负载均衡的方式保持有其他可以处理请求的节点。

如果在各个宿主机上交错执行这些操作，就会有一个高效的备份系统，并且有一个为支持工程师复原尽可能清晰的状态的方法。

图11-4描述了这个过程的简化视图。

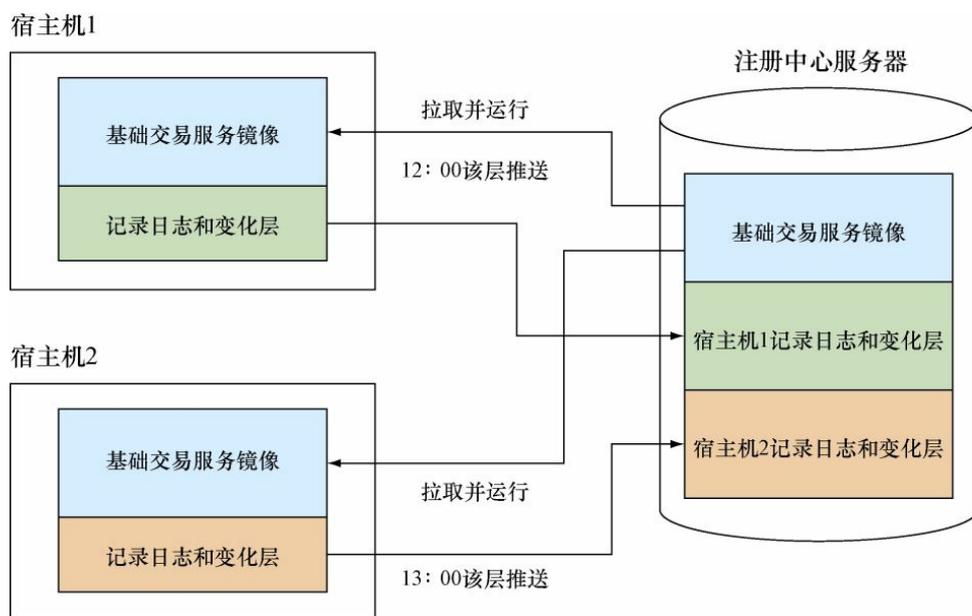


图11-4 双宿主机服务备份

备份只会推送基础镜像和备份时容器的状态之间的差异，备份是交错进行的以确保至少有一台宿主机上的服务是运行的。注册中心服务器只存储基础镜像和每次提交时的差异，节省磁盘空间。

和凤凰式部署结合

用户可以更进一步把这个技巧和凤凰式部署模型结合起来。凤凰式部署是一个强调尽可能替换系统而不是就地升级部署的部署模型。它是很多Docker工具的中心原则。

本例中，比起提交容器然后让它继续运行，可以按照如下做法：

- 从注册中心拉取最新的镜像的副本；
- 停止运行中的容器；
- 开启新容器；
- 提交，打标签，然后把旧的容器推送到registry。

和这种方法结合，让你更加确定生产系统没有偏离源镜像。作者中的一位用这种方法来管理家庭服务器上的一个生产系统。

11.4 小结

本章中展示了很多在生产环境运行Docker会遇到的问题。和Docker的很多部分一样，这是个变化很快的领域，尤其是因为各类组织迁移工作负担的时候，还在不断发现在生产环境使用Docker的用例和痛点。

我们讨论过的主要内容有：

- 把容器中的日志捕获到宿主机的syslog守护进程；
- 把Docker日志的输出捕获到宿主机级别的服务；
- 使用cAdvisor监控容器的性能；
- 限制容器对CPU、内核和内存资源的使用；
- 一些Docker的奇妙用法，如cron交付工具和备份系统。

现在我们讨论过了Docker在生产环境中可以做什么，应该做什么，接下来我们看一下当出错的时候如何调试Docker。

第12章 Docker生产环境实践——应对各项挑战

本章主要内容

- 绕过Docker的命名空间功能直接使用宿主机的资源
- 通过调整存储来获取更多空间
- 使用宿主机工具直接调试容器的网络
- 跟踪系统调用来确定宿主机上的容器无法正常工作的原因

在本章中，我们将讨论当Docker的抽象不工作时我们能做些什么。这些主题会涉及Docker的底层机制，以及理解为何要采用这样的解决方案。在此过程中我们旨在让读者更深入地了解使用Docker时的一些陷阱以及应对方法。

12.1 性能——不能忽略宿主机

虽然Docker试图将应用程序从其运行的宿主机中抽象出来，但是我们不能完全忽略宿主机。为了提供这样的抽象，Docker必须添加几个中间层。这些层可能会影响正在运行的系统。而且有时我们需要理解这些层的作用，以便解决或者变相绕过一些运维方面的难题。

本节我们将讨论如何绕过这些抽象层，最终得到一个Docker很少侵入的Docker容器。我们也将展示，尽管Docker似乎高度抽象了所用存储的细节，但是有时候这样做反而会遇到麻烦。

技巧97 从容器访问宿主机资源

我们在技巧19中讨论了卷，它是最常用的绕过Docker抽象的手段。通过使用卷，用户可以方便地与宿主机共享文件，可以在镜像层外保存大文件。而且应用程序访问这些卷的速度比访问容器文件系统要快得多，这是因为一些存储后端给某些工作负载带来了巨大的开销——这一点并不是对所有应用程序都那么重要，但是在某些情况下会很重要。

Docker 为了给每个容器提供独立的网络而设置的网络接口则会带来另一个性能问题。就像文件系统性能一样，网络性能肯定不是每个人都会遇到的瓶颈，但是读者可能要按需进行基准测试（尽管网络调优的细节已经超出了本书的范围）。或者，读者可能有其他原因需要直接绕过Docker网络，例如，一台开放了随机端口监听的服务器可能无法很好地与Docker指定映射端口范围的方式相契合，特别是如果要为此直接映射整段的端口的话，宿主机的这段端口就会一直被占用，不论服务是否真的在使用它们。

无论什么样的原因，有时候Docker这样的抽象会是一个阻碍，因此，Docker也为那些有需求的用户提供了绕开其限制的能力。

问题

想要从容器访问宿主机的资源。

解决方案

在运行`docker run`命令时带上宿主机选项和卷标志。

讨论

Docker提供了几种方法来绕过Docker使用的内核命名空间功能。

什么是命名空间 内核命名空间是内核提供给程序的一个服务，允许它们以某种方式获取全局资源的视图，使这些资源看起来像是提供给自己的单独实例。例如，一个程序可以请求一个网络命名空间，看上去就像是一个完整的网络栈。Docker使用和管理这些命名空间来创建其容器。

表12-1总结了Docker如何使用命名空间，以及如何有效地关闭它们。

表12-1 命名空间和Docker

内核命名空间	描述	是否在Docker中使用	“关闭”选项
Network	网络子系统	是	--net=host
IPC	进程间通信：共享内存、信号量等	是	--ipc=host
UTS	主机名和NIS域	是	--uts=host
PID	进程ID	是	--pid=host
Mount	挂载点	是	--volume,--device
User	用户和用户组ID	否	N/A

标志不可用？ 如果这些标志不可用，很可能是因为所使用的 Docker 版本过时了。

如果应用程序需要大量使用共享内存，例如，想让容器和主机共享内存空间，可以使用 `--ipc=host` 标志来实现这一点。这种用法比较高级，因此我们将主要关注其他一些更常见的用法。

Docker 目前还未使用 Linux 内核的用户命名空间功能，不过 Docker 正在努力实现^[1]。

1. 网络和主机名

要使用宿主机的网络，可以在运行容器时将 `--net` 标志设置为 `host`，如下所示：

```
user@yourhostname:/$ docker run -ti --net=host ubuntu /bin/bash
root@yourhostname:/#
```

不难发现，这与使用了网络命名空间的容器的不同之处在于，容器中的主机名与宿主机的主机名是相同的。从实践角度讲，这可能会导致混乱，因为它不能清楚地告诉用户，是在一个容器里。

在一个网络隔离的容器中，可以使用 `netstat` 来快速验证启动时没有网络连接：

```
host$ docker run -ti ubuntu
root@b1c4877a00cd:/# netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags               Type                   State                   I-Node  Path
root@b1c4877a00cd:/#
```

运行相同的命令但使用`host`网络指令会显示繁忙的宿主机网络:

```
$ docker run -ti --net=host ubuntu
root@host:/# netstat -nap | head
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State      PID/
Program name
tcp      0      0 127.0.0.1:47116 0.0.0.0:*    LISTEN   -
tcp      0      0 127.0.1.1:53    0.0.0.0:*    LISTEN   -
tcp      0      0 127.0.0.1:631   0.0.0.0:*    LISTEN   -
tcp      0      0 0.0.0.0:3000    0.0.0.0:*    LISTEN   -
tcp      0      0 127.0.0.1:54366 0.0.0.0:*    LISTEN   -
tcp      0      0 127.0.0.1:32888 127.0.0.1:47116 ESTABLISHED -
tcp      0      0 127.0.0.1:32889 127.0.0.1:47116 ESTABLISHED -
tcp      0      0 127.0.0.1:47116 127.0.0.1:32888 ESTABLISHED -
root@host:/#
```

netstat是什么 `netstat`是一个命令，允许用户查看本地网络栈上的相关网络信息。它常用于确定网络套接字的状态。

通常使用`net=host`标志有几个原因。首先，它可以让外界服务更容易连接到容器。不过这样也会失去为容器创建端口映射的好处。例如，如果有两个容器都监听80端口，使用这种方式的话将无法在同一个宿主机上同时运行它们。其次，使用这个标志时网络性能相比Docker网络会有显著提升。

图12-1展示了Docker中的网络分组（也称数据包）和本地网络中的分组必须经过的层数。尽管本地网络只需通过宿主机的TCP/IP栈到网卡（network interface card, NIC），但Docker必须额外维护一个虚拟以太网对（也称为veth对，它是对使用以太网电缆的物理连接的虚拟表示），veth对和宿主机网络之间有一个网桥，并且有一个网络地址转换（network address translation, NAT）层。在正常的使用场景里，这样的开销可以导致Docker网络的速度仅有本地宿主机网络的一半。

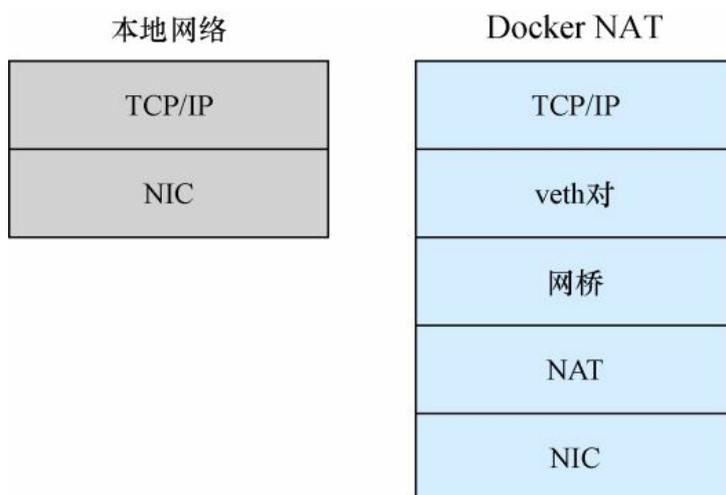


图12-1 Docker网络与本地网络对比

2. PID

PID命名空间标志与其他命名空间很相似：

```

imiell@host:/$ docker run ubuntu ps -p 1 (1)
  PID TTY          TIME CMD
   1 ?                00:00:00 ps (2)
imiell@host:/$ docker run --pid=host ubuntu ps -p 1 (3)
  PID TTY          TIME CMD
   1 ?                00:00:27 systemd (4)

```

(1) 在容器化环境中运行ps命令，显示只有一个PID为1的进程

(2) 运行的ps命令是该容器中唯一的进程，而且PID为1

(3) 运行同样的ps命令但不使用PID命名空间，展示了主机的进程的视图

(4) 这次PID为1的进程是systemd命令，它是宿主机的操作系统的启动进程。读者看到的展示可能有所不同，这取决于所使用的发行版

上面的示例演示了在具有宿主机PID视图的容器中，`systemd` 进程ID为1，而在没有该视图的情况下，能看到的唯一一个进程是`ps`命令本身。

3. 挂载

如果要访问宿主机的设备，可以使用`--device`标志来使用特定设备，或者使用`--volume`标志来挂载宿主机的整个文件系统：

```
docker run -ti --volume /:/host ubuntu /bin/bash
```

此命令将宿主机的`/`目录挂载到容器的`/host`目录。读者可能想知道为什么不能把宿主机的`/`目录挂载到容器的`/`目录，原因是这是`docker`命令明确禁止的行为。

读者可能还想知道是否可以使用这些标志创建一个与宿主机几乎无法区分的容器。这个问题我们将在下一节讨论。

4. 类宿主机容器

可以使用这些标志位来创建一个几乎拥有宿主机透明视图的容器：

```
host:/$ docker run -ti --net=host --pid=host --ipc=host \ (1)
--volume /:/host \ (2)
busybox chroot /host (3)
```

(1) 使用3个`host`参数(`net,pid,ipc`)运行容器

(2) 将宿主机的根文件系统挂载到容器的`/host`目录。Docker不允许将卷挂载到`/`目录，所以用户必须指定`/host`子目录卷

(3) 启动BusyBox容器。用户只需要chroot命令，而这是一个包含该命令的小镜像。执行chroot可以使被挂载的文件系统就像root一样展现给用户

具有讽刺意味的是，Docker被称为“吃了类固醇的chroot”，而这里我们使用某些特性作为框架，以一种破坏chroot主要设计目标之一（即保护宿主机系统）的方式运行chroot。在这一点上我们尽量不要想得太复杂。

在任何情况下，很难想象有人会在现实世界中使用这个命令（有指导性的）。如果读者想到的话，请联系我们。

也就是说，用户可能更想将它作为一个更有用的命令的基础，像这样：

```
$ docker run -ti --workdir /host \  
--volume /:/host:ro ubuntu /bin/bash
```

`--workdir /host`将容器启动时的工作目录设置为宿主机的文件系统的根目录，使用`--volume`参数进行挂载。卷规格说明的`:ro`部分意思是宿主机文件系统以只读模式挂载。

执行该命令可以给用户一个文件系统的只读视图，从而拥有一个环境安装工具（使用标准的Ubuntu包管理工具）并进行检查。例如，可以使用一个运行了nifty工具的镜像来向宿主机的文件系统报告安全问题，而不用将该工具安装到宿主机上。

不安全！正如前面的讨论中暗示的那样，带上这些标志的话会引入更多的安全风险。在安全性方面，使用它们应该被视为等同于运行时使用`--privileged`标志。

在本技巧中，读者学到了该如何绕过容器中Docker的一些抽象。我们将在接下来的技巧里一起看看如何绕过Docker底层磁盘存储的限制。

技巧98 Device Mapper存储驱动和默认的容器大小

Docker附带了一些默认支持的存储驱动（storage driver）。它们在处理层上提供了不同的方法，每种方法都有自己的优点和缺点。可以在<https://docs.docker.com/engine/userguide/storagedriver/>找到更多相关的Docker文档。

CentOS和Red Hat上默认的存储驱动是devicemapper，Red Hat为其加入了更多的支持，用以替代AUFS（Ubuntu使用的默认存储驱动），因为当时它具备bug更少、使用上更灵活的特点。

Device Mapper术语 Device Mapper是一项Linux技术，它通过提供以某些用户定义的方式将物理设备映射到虚拟设备，从而抽象对物理设备的访问。本技巧中我们谈论的devicemapper指的是在Device Mapper之上构建的Docker存储驱动的名称。

devicemapper驱动器的默认行为是分配一个文件，将其视为可读取和写入的“设备”。但是，这个文件有一个固定的最大容量，当它空间不足时不会自动增加。

问题

使用Device Mapper存储驱动时，Docker容器上的可用空间已耗尽。

解决方案

更改Docker容器的最大容量。

讨论

为了说明这个问题，不妨尝试运行下面的Dockerfile:

```
FROM ubuntu:14.04
RUN truncate --size 11G /root/file
```

如果没有更改Docker守护程序上关于存储驱动的任何默认配置，应该会看到下面这样的输出:

```
$ docker build .
Sending build context to Docker daemon 24.58 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
Pulling repository ubuntu
d2a0ecffe6fa: Download complete
83e4dde6b9cf: Download complete
b670fb0c7ecd: Download complete
29460ac93442: Download complete
Status: Downloaded newer image for ubuntu:14.04
---> d2a0ecffe6fa
Step 1 : RUN truncate --size 11G /root/file
---> Running in 77134fcbd040
INFO[0200] ApplyLayer exit status 1 stdout: stderr: write /root/file:
no space left on device
```

这次构建最终会失败，因为尝试创建11 GB文件时失败，并且抛出“no space left”的错误消息。注意，在执行这一命令时，我们的机器上仍然有超过200 GB的磁盘空间，因此该容器并没有达到机器范围的限制。

如何知道是否在使用**devicemapper** 如果运行**docker info**，它会在输出中告诉用户使用了哪个存储驱动。如果输出中有**devicemapper**，那么说明用户正在使用**devicemapper**，也就是说本技

巧可能与用户相关。

默认情况下，`devicemapper`容器的空间限制为 10 GB。要改变这个设定，需要清理Docker文件夹，重新配置Docker守护进程，然后重新启动。参见附录B了解有关如何在发行版上重新配置Docker守护程序的详细信息。

空间限制已被更改 大约在本书出版时，`devicemapper`限制已升至 100 GB，因此，读者的容器的空间限制可能会比这里提到的要高。

要更改该配置，需要在Docker选项中添加或替换`dm.basesize`项，使其大于试图创建的11 GB文件：

```
--storage-opt dm.basesize=20G
```

一个典型的文件可能如下所示：

```
DOCKER_OPTIONS="-s devicemapper --storage-opt dm.basesize=20G"
```

在重新启动Docker守护进程后，可以重新执行之前运行过的`docker build`命令：

```
\# ocker build --no-cache -t big .  
Sending build context to Docker daemon 24.58 kB  
Sending build context to Docker daemon  
Step 0 : FROM ubuntu:14.04  
----> d2a0ecffe6fa  
Step 1 : RUN truncate --size 11G /root/file  
----> Running in f947affe7900  
----> 39766546a1a5  
Removing intermediate container f947affe7900  
Successfully built 39766546a1a5
```

可以看到11 GB的文件被成功创建。

这是devicemapper存储驱动的运行约束，无论用户是使用Dockerfile构建镜像还是运行容器，都是如此。

12.2 在容器出问题——调试Docker

在本节中，我们将介绍几个技巧，帮助读者了解和解决在Docker容器中运行的应用程序遇到的一些问题。我们将介绍到在使用宿主机的工具来调试问题时如何“跳入”容器的网络，然后通过直接监控网络接口来了解一个更具有“实践性”的解决方案。

最后，我们将演示Docker抽象是如何被破坏的，从而导致容器在一个宿主机上工作，而在其他宿主机上不工作，以及如何在一个生产系统上对其进行调试。

技巧99 使用nsenter调试容器的网络

在理想世界中，用户可以使用socat（见技巧4）在大使容器（ambassador container）中诊断问题。用户可以启动一个额外的容器，并确保连接转到这个作为代理的新容器。该代理允许诊断和监控连接，然后将其转发到正确的地方。但是，现实世界里，往往不那么方便（或可能）设置这样一个只用于调试的容器。

大使容器模式 见技巧66对大使模式的描述。

读者已经在技巧14中了解了docker exec命令。本技巧讨论nsenter，这个工具和docker exec命令看起来很相似，但允许在容器

中使用自己机器上的工具，而不是局限于容器已经安装的东西。

问题

想要调试容器中的网络问题，但使用的工具却不在容器中。

解决方案

使用nsenter来跳到容器的网络，但是工具仍然在宿主机上。

讨论

如果Docker宿主主机上还未安装nsenter，可以通过以下命令来安装：

```
$ docker run -v /usr/local/bin:/target jpetazzo/nsenter
```

这将在/usr/local/bin中安装nsenter，然后便即刻可以使用。nsenter也可能包含在所使用的系统发行版中（在util-linux包）。

读者可能已经注意到，一般可用的BusyBox镜像默认不附带bash。作为一个初步的演示，我们将展示如何使用宿主机的bash程序进入容器：

```
$ docker run -ti busybox /bin/bash
FATA[0000] Error response from daemon: Cannot start container
➤ a81e7e6b2c030c29565ef7adb94de20ad516a6697deeeb617604e652e979fda6:
➤ exec: "/bin/bash": stat /bin/bash: no such file or directory
$ CID=$(docker run -d busybox sleep 9999) (1)
$ PID=$(docker inspect --format {{.State.Pid}} $CID) (2)
$ sudo nsenter --target $PID \ (3)
--uts --ipc --net /bin/bash (4)
root@781c1fed2b18:~#
```

(1) 启动BusyBox容器并保存容器ID（CID）

(2) 检查容器，提取进程ID（PID）（见技巧27）

(3) 运行 `nsenter`，指定 `--target` 标志位来进入容器。可能无须带上 `sudo`

(4) 通过余下的参数指定进入容器时的命名空间（见技巧97，了解更多关于命名空间的知识）。这里的关键点是不使用 `--mount` 标志，因为它会使用容器的文件系统，而该文件系统没有安装 `bash`。指定 `/bin/bash` 作为可执行命令来启动容器

需要指出的是虽然不能直接访问容器的文件系统，但是用户可以使用宿主机拥有的所有工具。

我们之前的某个需求是想有一种办法找出宿主机上哪个 `veth` 接口设备对应于哪个容器。例如，有时候我们需要快速将容器从网络上卸载，而不必使用第8章中的任何工具来模拟网络中断。但是，一个没有特权的容器无法关闭网络接口，所以我们需要找出 `veth` 接口的名称然后在宿主机上完成这项任务：

```
$ docker run -d --name offlinetest ubuntu:14.04.2 sleep infinity
fad037a77a2fc337b7b12bc484babb2145774fde7718d1b5b53fb7e9dc0ad7b3
$ docker exec offlinetest ping -q -c1 8.8.8.8 (1)
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.966/2.966/2.966/0.000 ms
$ docker exec offlinetest ifconfig eth0 down (2)
SIOCSIFFLAGS: Operation not permitted
$ PID=$(docker inspect --format {{.State.Pid}} offlinetest)
$ nsenter --target $PID --net ethtool -S eth0 (3)
NIC statistics:
  peer_ifindex: 53
$ ip addr | grep '^53' (4)
53: veth2e7d114: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noque
ue
➤ master docker0 state UP
$ sudo ifconfig veth2e7d114 down (5)
$ docker exec offlinetest ping -q -c1 8.8.8.8 (6)
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data. (7)
```

```
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

(1) 尝试从新容器内部执行ping命令验证连接成功与否

(2) 我们不能关闭容器中的接口。注意，用户的接口可能不是eth0, 所以如果该命令不生效，那么不妨试试通过ifconfig找出主接口的名称

(3) 进入该容器的网络空间，使用ethtool命令从宿主机查找对等接口的索引，即虚拟接口的另一端

(4) 查找宿主机上的接口列表，从而找出容器的对应veth接口

(5) 查找宿主机上的接口列表，从而找出容器的对应veth接口

(6) 关闭虚拟接口

(7) 从容器内部使用ping命令验证连接是失败的

作为最后一个例子，读者可能想要在容器里使用的程序应该是tcpdump，这是一种在网络接口上记录所有TCP分组的工具。要使用它，需要使用--net命令运行nsenter，这样可以在宿主机上“查看”容器的网络，这样一来便可以使用tcpdump监控分组。

例如，下面代码中的tcpdump命令会将所有分组记录到/tmp/google.tcpdump文件中（我们假设用户仍然在前面启动的nsenter会话中）。然后，我们可以通过访问网页触发一些网络流量：

```
root@781c1fed2b18:/# tcpdump -XXs 0 -w /tmp/google.tcpdump &
root@781c1fed2b18:/# wget google.com
--2015-08-07 15:12:04-- http://google.com/
Resolving google.com (google.com)... 216.58.208.46, 2a00:1450:4009:80d:
:200e
```

```

Connecting to google.com (google.com)|216.58.208.46|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://www.google.co.uk/?gfe_rd=cr&ei=tLzEVcCXN7Lj8wepgarQAQ
➤ [following]
--2015-08-07 15:12:04--
➤ http://www.google.co.uk/?gfe_rd=cr&ei=tLzEVcCXN7Lj8wepgarQAQ
Resolving www.google.co.uk (www.google.co.uk)... 216.58.208.67,
➤ 2a00:1450:4009:80a::2003
Connecting to www.google.co.uk (www.google.co.uk)|216.58.208.67|:80...
➤ connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: 'index.html'

index.html          [ <=>                ] 18.28K --.-KB/s in 0.008s

2015-08-07 15:12:05 (2.18 MB/s) - 'index.html' saved [18720]

root@781c1fed2b18:# 15:12:04.839152 IP 172.17.0.26.52092 >
➤ google-public-dns-a.google.com.domain: 7950+ A? google.com. (28)
15:12:04.844754 IP 172.17.0.26.52092 >
➤ google-public-dns-a.google.com.domain: 18121+ AAAA? google.com. (28)
15:12:04.860430 IP google-public-dns-a.google.com.domain >
➤ 172.17.0.26.52092: 7950 1/0/0 A 216.58.208.46 (44)
15:12:04.869571 IP google-public-dns-a.google.com.domain >
➤ 172.17.0.26.52092: 18121 1/0/0 AAAA 2a00:1450:4009:80d::200e (56)
15:12:04.870246 IP 172.17.0.26.47834 > lhr08s07-in-f14.1e100.net.http:
➤ Flags [S], seq 2242275586, win 29200, options [mss 1460,sackOK,TS va
l
➤ 49337583 ecr 0,nop,wscale 7], length 0

```

“**Temporary failure in name resolution**”错误 这取决于读者的网络配置，读者可能需要临时修改`resolv.conf`文件，使DNS查找可以正常工作。如果收到“Temporary failure in name resolution”错误，请尝试将 `nameserver 8.8.8.8` 这行添加到`/etc/resolv.conf`文件的顶部。别忘了在完成实验后还原它。

这也展示了Docker另外一个备受瞩目的使用场景——在Docker提供的隔离网络环境中调试网络问题会更加容易。试着记住`tcpdump`的一些正确的参数，从而妥善地过滤掉一些不相关的分组，这在半夜里维护系统时会是一个容易出错的环节。使用前面的方法，大可忘掉这

一点，使用tcpdump捕获容器内的所有内容，而无须在镜像里安装（或不安装）它。

技巧100 无须重新配置，使用tcpflow进行实时调试

tcpdump是网络诊断的事实标准，如果需要深入调试网络问题，它可能是大多数人首选的工具。

但 tcpdump 通常用于显示分组摘要以及检查分组首部和协议信息——它对于两个程序之间应用层数据流的展示并不是很完善。但是这些信息在调查两个应用程序间的通信问题时可能非常重要。

问题

需要监控容器化应用程序的通信数据。

解决方案

使用tcpflow捕获通过接口的流量。

讨论

tcpflow类似于tcpdump（接受相同的模式匹配表达式），但是它的目的是更好地了解应用程序的数据流。可以通过系统包管理工具安装tcpflow，但是，如果系统包中没有，我们也为此准备了一个可用的Docker镜像，它在功能上与通过包管理工具安装的效果一样：

```
$ IMG=dockerinpractice/tcpflow
$ docker pull $IMG
$ alias tcpflow="docker run --rm --net host $IMG"
```

这里有两种方式通过Docker使用tcpflow，一是将其指向docker0接口，并使用分组过滤表达式只检索想要的分组，或者使用上一个技巧的方法来找到感兴趣的容器的veth接口，并捕获该接口。

表达式过滤更加强大，因为它可以让用户深入了解感兴趣的流量，所以我们将展示一个简单的示例以便读者可以快速上手：

```
$ docker run -d --name tcpflowtest alpine:3.2 sleep 30d
fa95f9763ab56e24b3a8f0d9f86204704b770ffb0fd55d4fd37c59dc1601ed11
$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' tcpflowtest
172.17.0.1
$ tcpflow -c -J -i docker0 'host 172.17.0.1 and port 80'
tcpflow: listening on docker0
```

在上面的示例中，我们要求tcpflow以彩色的方式打印容器中通过80端口（通常用于HTTP流量）的流入或流出流量。现在，读者可以通过在容器的新终端中访问网页来体验上述命令的效果：

```
$ docker exec tcpflowtest wget -O /dev/null http://www.example.com/
Connecting to www.example.com (93.184.216.34:80)
null          100% |*****| 1270  0:00:00 ETA
```

读者将可以看到在tcpflow终端中的彩色输出。到目前为止，命令的累积输出看上去会是这样：

```
$ tcpflow -J -c -i docker0 'host 172.17.0.1 and (src or dst port 80)'
tcpflow: listening on docker0
172.017.000.001.36042-093.184.216.034.00080:
➡ GET / HTTP/1.1 (1)
Host: www.example.com
User-Agent: Wget
Connection: close

093.184.216.034.00080-172.017.000.001.36042:
➡ HTTP/1.0 200 OK (2)
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Mon, 17 Aug 2015 12:22:21 GMT
[...]

<!doctype html>
<html>
<head>
  <title>Example Domain</title>
[...]
```

(1) 蓝色开始

(2) 红色开始

tcpflow 是工具箱的一个很好补充，尽管它并不引人注目。用户可以对长时间运行的容器执行tcpflow，以便了解其现在正在传送的内容，或者与tcpdump一起使用，来获得更详细的应用程序发送的请求以及传输的信息。

技巧101 调试在特定宿主机上出问题的容器

前两个技巧已经展示了该如何对由容器和其他位置（无论是其他容器还是互联网上的第三方）之间的交互造成的问题开始调查。

如果用户已经将问题定位到某台宿主机，并且确定不是外部交互的原因，那么下一步应该是尝试减少变动部分（删除卷和端口）的数量，并检查宿主机自身的详细信息（可用磁盘空间、打开的文件描述符的数量等）。每个宿主机是否使用了最新版本的Docker可能也值得检查一下。

在某些情况下，上面的方法都行不通。例如，有一个镜像，运行时没有带任何参数（如docker run imagename），而它在不同的宿主主机上运行的行为不同，这完全有可能发生。

问题

想要确认为什么在特定宿主主机上的某个容器的特定行为无法正常工作。

解决方案

追踪整个过程来查看它的系统调用，同时将其与一个正常工作的

系统作对比。

讨论

虽然Docker的目标是允许应用程序“在任何地方运行任何应用程序”，但它试图实现这一点的手段并不总是万无一失。

Docker将Linux内核API视为其宿主机（即它可以运行的环境）。当第一次了解Docker如何工作时，许多人会问Docker如何处理Linux API的变更。据我们所知，目前它没有做任何处理。幸运的是，Linux API是向后兼容的，但是不难想象将来的某个场景，即创建一个新的Linux API并被Docker化的应用程序使用，然后将该应用程序部署到一个足够运行Docker但版本足够老的内核时，它可能并不支持该特定的API调用。

这是否会发生 读者可能认为Linux内核API的变更只是一个理论上存在的问题而已，但是实际上，我们在编写这本书时的确遇到了这种情况。我们工作的项目使用`memfd_create` Linux系统调用，它只存在于3.17及以上版本的内核。因为我们工作的一些宿主机有较老的内核，我们的容器只能在部分系统上正常工作，其他的系统则不行。

并不是只有这种情况会导致Docker抽象不工作。容器还有可能无法工作在特定的内核上，因为应用程序可能对宿主机上的文件做出一些假设。虽然这种情况发生的概率很小，但是它确实存在，重要的是要警惕这种风险。

1. SELinux

可以让Docker抽象无法工作的一个例子是与SELinux交互的任何东西。如第10章所讨论的，SELinux是在内核中实现的安全层，它游离于正常的用户权限之外。

Docker通过这层来增强容器的安全性，即管理可以在容器里执行什么操作。例如，如果你是容器的root用户，那么你也也是宿主机上的root用户。虽然很难，但是一旦容器被攻破，也就同时获得了宿主机的root权限。这并非不可能。之前已经有漏洞被曝光，而且可能还存在一些社区还未发觉的漏洞。SELinux等于提供了另一层保护，即使root用户从容器侵入宿主机，这里对其在宿主机上可以执行的操作也做了一些限制。

到目前为止还算不错，但是对Docker来说，问题在于SELinux是在宿主机上实现的，而不是容器内部。这就意味着在容器里运行的应用程序查询SELinux并发现它是开启的状态时，可能会促使它们对运行的环境做出某些假设。如果不满足这些预期，则会以意想不到的方式失败。

在下面的示例中，我们运行一个安装了Docker的CentOS 7 Vagrant机器，并且在里面安装了一个Ubuntu 12.04容器。如果我们执行一条相当简单的命令来添加一个用户的话，退出码会是12，这代表有错误，并且提示用户没有被成功创建：

```
[root@centos vagrant]# docker run -ti ubuntu:12.04
Unable to find image 'ubuntu:12.04' locally
Pulling repository ubuntu
78cef618c77e: Download complete
b5da78899d3a: Download complete
87183ecb6716: Download complete
82ed8e312318: Download complete
root@afade8b94d32:/# useradd -m -d /home/dockerinpractice dockerinpractice
root@afade8b94d32:/# echo $?
12
```

同样的命令在ubuntu: 14.04容器上却可以正常工作。如果想尝试重现这个结果，需要一台CentOS 7（或类似的）机器。但是出于学习的目的，这对于接下来技巧中使用的任何命令和容器已经足够了。

\$?是做什么的 在bash中，**\$?**给出最后一条执行命令的退出码。退出码的含义因命令而异，但通常退出码为0意味着调用成功，非零码指示某种错误或异常条件。

2. 调试Linux API调用

我们知道容器之间可能的不同是由于在宿主机上运行的内核API之间存在差异性，**strace**可以帮助确定调用内核API之间的差异。

什么是**strace** **strace**是一个工具，它允许嗅探一个进程对Linux API所做的调用（也称为系统调用）。这是一个非常有用的调试和教学工具。

首先，需要使用相应的包管理工具在容器上安装**strace**，然后使用**strace**命令运行不同的命令。下面是失败的**useradd**调用的一些输出示例：

```
# strace -f \ (1)
useradd -m -d /home/dockerinpractice dockerinpractice (2)
execve("/usr/sbin/useradd", ["useradd", "-m", "-d", (3)
➤ "/home/dockerinpractice", "dockerinpractice"], [/* 9 vars */]) = 0
[...]
open("/proc/self/task/39/attr/current", (4)
➤ O_RDONLY) = 9
read(9, "system_u:system_r:svirt_lxc_net"... (5)
4095) = 46
```

```

close(9)                                =0 (6)
[...]

open("/etc/selinux/config", O_RDONLY) =
➤ -1 ENOENT (No such file or directory)
open("/etc/selinux/targeted/contexts/files/ (7)
➤ file_contexts.subs_dist", O_RDONLY) = -1 ENOENT (No such file or dir
ectory)
open("/etc/selinux/targeted/contexts/files/
file_contexts.subs", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/etc/selinux/targeted/contexts/files/
➤ file_contexts", O_RDONLY) = -1 ENOENT (No such file or directory)

[...]
exit_group(12)                            =? (8)

```

(1) 带上-f标志执行strace命令，它可以确保命令的进程及其子进程都能被strace追踪

(2) 在strace调用中追加想要调试的命令

(3) strace输出的每一行都从Linux API调用开始。execve调用在这里执行传给strace的命令。结束处的0是调用的返回值（表示成功）

(4) open系统调用打开一个文件进行读取。返回值（9）是用于对文件进行后续调用的文件句柄号。在这种情况下，/proc文件系统会检索SELinux信息，该文件系统保存正在运行的进程的有关信息

(5) read系统调用工作于之前打开的文件（文件句柄号为9），返回了读取的字节数（46）

(6) close系统调用根据文件句柄关闭了引用的文件

(7) 应用程序试图打开它期望位置的SELinux文件，但是都失败了。strace可以告诉用户返回值的含义：没有该文件或目录

(8) 进程退出码为12，对于uesradd命令来说意味着不能创建目

录

上面的输出可能看起来很混乱，但重复阅读几次之后就不难理解了。每一行代表一个对Linux内核的调用，以便在所谓的内核空间（而不是用户空间，意味着应用程序不会将执行操作的责任交给内核）执行一些操作。

使用`man 2`查找有关系统调用的更多信息。如果读者想了解有关特定系统调用的更多信息，可以运行`man 2 <callname>`来了解更多信息。读者可能需要使用`apt-get install manpages-dev`或类似的命令安装帮助手册。或者，在Google搜索`man 2 <callname>`也可能会得到需要的信息。

这是一个Docker抽象无法正常工作的例子。在这种情况下，操作失败是因为程序期望SELinux文件存在，因为SELinux似乎在容器上是启用的，但是限制的细节保留在宿主机上。

尽管这种情况很少见，但是使用`strace`来调试和了解应用程序如何进行交互的能力是一种非常宝贵的技术，它不仅针对Docker，还可用在更通用的开发过程。

阅读帮助手册。如果你真的想成为一名开发人员，阅读所有系统调用的帮助手册是非常有用的。起初，这里面看似充斥着晦涩的术语，但是当阅读完各个主题之后，你会学到很多基本的Linux概念。在某些时候，你会开始看到大多数语言是如何起源于此，而且它们的一些轶事也相当有趣。耐心一点吧，毕竟你没办法一下子就全部理解。

12.3 小结

在本章中，我们介绍了当Docker不能像预期那样正常工作时我们可以做的一些事情。虽然在使用Docker过程中很少会发现抽象层的缺陷，但是这种可能性仍然是存在的，做好准备，了解哪些地方可能会出错是至关重要的。

本章展示了以下几点内容：

- 通过docker的标志直接使用宿主机的资源，以提高效率；
- 调整DeviceMapper磁盘大小，最大限度地使用磁盘空间；
- 使用nsenter跳转到容器的网络；
- 运行strace以确定为什么Docker容器在特定宿主机上不工作。

本书到此完结！我们希望读者已经了解了Docker的一些用途，并初步有了一些将Docker集成到自己的公司或个人项目中的想法。如果想联系我们或给我们一些反馈，请在Manning网站的本书论坛（<https://forums.manning.com/forums/docker-in-practice>）中创建一个主题，或者向“docker-in-practice”GitHub的任何一个仓库发起issue。

[1] Docker在1.10版本中已经引入了对用户命名空间的支持。——译者注

附录A 安装并使用Docker

本书中的技巧有时候需要创建文件，以及从GitHub克隆仓库。为了避免干扰，我们建议读者在需要工作空间的时候，为每个技巧单独创建一个新的空白文件夹。

对安装和使用Docker来说，Linux用户做起来相对容易，尽管在不同的Linux发行版之间可能具体的细节有很大不同。在这里我们就不列举各种不同的可能性了，建议读者查看最新的Docker文档：<https://docs.docker.com/installation/>。

尽管本书假设读者在使用一个Linux发行版（容器目前是基于Linux的，所以这样事情就简单了）。很多读者对于基于Windows或者OS X的机器的Docker工作非常感兴趣，对于这些读者值得一提的是，本书中的技巧仍然有效，只要是用下述方法中的一种来设置Docker守护进程的。

Windows Server即将迎来变化 微软公司承诺要支持Docker容器范型和管理接口。Windows Server将作出一些改变以允许创建基于Windows的容器。本书未来的版本可能会涉及这些内容，但是编写本书时还不能使用。

A.1 虚拟机的方式

在Windows或者Mac上使用Docker的一种途径就是安装一个完全的Linux虚拟机。一旦完成，就可以像使用任意原生Linux机器一样使

用这台虚拟机。

达到这个目标最常见的方式是安装VirtualBox。参见<http://virtualbox.org>以获得相关的信息和安装指导。

A.2 连接到外部Docker服务器的Docker客户端

如果读者已经有一个设置为服务器的Docker守护进程，可以在Windows或者Mac上安装一个和它通信的原生客户端。注意要暴露的端口是在外部Docker服务器上暴露的，不是在本地机器上——可能需要变换IP地址以访问暴露出来的服务。

这一高级方法的本质见技巧1，让它更安全的细节见技巧86。

A.3 原生Docker客户端和虚拟机

一个常用的方式是，安装一个运行着Linux和Docker的最小虚拟机，再安装一个和虚拟机上的Docker通信的Docker客户端。使用这个方法的标准做法是使用Boot2Docker虚拟机，它使用了一个“小内核Linux”发行版来提供Docker守护进程可以作为服务器来运行的环境。

Windows上的Docker

Windows与Mac和Linux是差异很大的操作系统，所以这里多讨论一些细节，强调一些常见的问题和解决方式。读者应该按照<https://docs.docker.com/engine/installation/windows/>上的Docker Toolbox官方安装文档来进行，在这里我们只强调一些重要的部分。

在安装过程中，应该确保你检查了Docker Compose for Windows（在技巧68以及其他技巧中使用）以及Git for Windows。后

者能让你可以使用`git clone`命令克隆整本书中提到的仓库，并且可以使用`bash shell`和一些Linux工具，如`grep`、`sed`和`curl`。它甚至附带了`ssh`和`perl`。本书全书的脚本都假设你在使用`bash`（或者类似的`shell`）并且这些工具可用，所以如果跟着做的话，从安装它们的过程中退出，使用Windows内置的`shell`或者Windows Powershell会造成一些问题。安装Kitematic Docker图形界面与否都可以——它可以让使用镜像变成简单的点击操作，你可能会觉得试一试也挺有趣的，但是本书不会涉及有关它的内容。

在安装过程中也应该选择Add Docker Binaries to PATH，这确保了你可以在终端中随时运行`docker`命令。

如果没有安装Oracle Virtualbox，Docker Toolbox会为你安装——你所有的容器都会在虚拟机内运行，因为Linux容器无法在Windows上原生运行。和A.1节讨论过的虚拟机的方式不同，由Docker Toolbox创建的虚拟机是非常轻量的，因为它只运行Docker，但是如果运行资源消耗大的程序，仍然要在VirtualBox接口设置中调整虚拟机的内存。

为了检查一切准备完毕，在程序列表里打开Docker Quickstart Terminal。如果运行着Docker的虚拟机没在运行的话，Docker Quickstart Terminal会把它启动并且设置好环境以便你可以立刻开始使用Docker。如果运行`docker run hello-world`，Docker会自动从Docker Hub拉下来`hello-world`镜像并且运行它。这个镜像的输出对刚才发生的和Docker客户端以及守护进程通信有关的步骤，给出了一个简单的说明。如果看不懂的话不用担心，在第2章中有后台运行原理的更多细节。

注意在Windows上有几件不可避免的怪事。

- 在卷的开头需要双斜杠，这在 <https://github.com/docker/docker/issues/12751>有讨论。
- 因为容器是在虚拟机内运行的，如果想要从宿主机上访问暴露的端口，就需要在shell中使用 `docker-machine ip default` 找到虚拟的IP来访问它（这在Windows的Docker安装指南中有涉及：<https://docs.docker.com/engine/installation/windows/>）。
- 一些没那么流行的工具可能不是简单地通过我们在技巧中的指示“使用包管理”就能获得的（例如，`socat`最好通过Cygwin来获得），其他工具（例如，与Linux强相关的工具，像 `strace` 和用于 `ip addr` 的 `ip` 命令）可能不能在Windows上直接使用。

Cygwin是什么 Cygwin，可在<https://www.cygwin.com/>获得，是一个在Windows上可用的Linux工具的合集。如果想要一个可以在Windows上实验的类Linux环境，Cygwin应该是首屈一指的。它自带一个包管理器，以便用户可以看一下可用软件里有哪些可用。

以下是对Window上一些命令和组件有用的替代品的简表，但是要铭记其中的一些并不是完美的替代品。本书着眼于使用Docker来运行Linux容器，所以更合理的是用一个完全的Linux发行版（不管是臃肿的虚拟机、云中的一个环境还是本地的安装）来试验Docker的完全潜力。

- `ip addr`——我们一般在本书中使用这条命令来找到在本地网络中我们的机器的IP。Windows的等价命令是 `ipconfig`。
- `strace`——见技巧97中讨论的“类宿主机的容器”以了解如何绕过Docker容器化来在运行Docker的虚拟机中获得类宿主机的权

限。你可能想启动shell而不是运行chroot，比起BusyBox你可能更想使用如同Ubuntu那样的带有包管理的Linux发行版。然后你就可以好像在宿主机上那样安装并运行命令。这条技巧适用于很多命令，并且可以让你的Docker虚拟机几乎和功能齐全的虚拟机一样。

- 向宿主机暴露端口——安装Cygwin并且从包列表中安装socat。需要从程序列表中启动Cygwin命令行以使用这些工具。要使用socat来转发端口，为宿主机外提供访问，可以使用socat `TCP-LISTEN:$PORT, reuseaddr, fork` `TCP:$DOCKERIP:$PORT`，其中\$PORT是想要转发的端口，\$DOCKERIP是在Docker终端中的`docker-machine ip default`的输出。

Windows上的图形应用程序

在Windows上运行Linux图形工具很有挑战性——不仅需要让所有这些代码都在Windows上工作，还需要决定如何展示。Linux上使用的窗口系统（称为X Window System或者X11）并不适用于Windows。幸运的是，X允许通过网络显示应用程序窗口，所以可以使用Windows上的X实现来显示Docker容器中运行的应用程序。

Windows上有数个不同的X实现，我们仅介绍可以用Cygwin获得的安装版。你应该遵循<http://x.cygwin.com/docs/ug/setup.html#setup-cygwin-x-installing>上的官方文档。当选择要安装的包时候，必须确保xorg-server、xinit和xhost被选中。

一旦安装完成，就可以打开Cygwin终端然后运行`XWin :0 -listen tcp -multiwindow`。它会在你的Windows机器上启动一个

X服务器，它有监听来自网络的连接（`-listen tcp`）以及在自己的窗口内显示每个应用程序（`-multiwindow`）的能力，而不是就一个显示应用程序的虚拟屏幕的单个窗口。一旦启动，读者就应该在系统的托盘区看到一个X图标。

授权问题 虽然这个X服务器可以监听网络，但目前它只信任本地机器。在目前我们见过的案例里，这允许来自Docker虚拟机的访问，但是如果有授权问题，你可以试着运行不安全的`xhost +`命令来允许来自所有机器的访问。如果这么做了，一定要确保防火墙设置了拒绝来自网络的任何连接企图。

是时候试用X服务器了！使用`ipconfig`找到本地机器的IP地址。你想要查找VirtualBox Host-Only的网络适配器的IP地址——Docker虚拟机收到的你的宿主机IP地址。如果你有很多这样的适配器，可能要挨个试试。启动第一个图形应用程序应该就像在Docker Quickstart Terminal运行`docker run -e DISPLAY=$MY_IP:0 --rm fr3nd/xeyes`一样简单，`$MY_IP`就是之前描述的适配器的IP地址。

A.4 获得帮助

如果读者运行在一个非Linux操作系统上并且想要获得进一步的帮助或者建议，Docker文档（<https://docs.docker.com/installation/>）中有官方对Windows和Mac用户的最新的官方建议。

附录B Docker配置

在本书的很多地方我们都建议读者更改自己的Docker配置文件，在启动Docker宿主机时做一些永久性的改变。本附录会为读者达成这条最佳实践提供一些建议。在这种情境下，所使用的操作系统发行版就很重要了。

大多数主流发行版的配置文件的位置列在表B-1中。

表B-1 Docker配置文件位置

发 行 版	配 置 文 件
Ubuntu/Debian/Gentoo	/etc/default/docker
OpenSuse/CentOS/Red Hat	/etc/sysconfig/docker

注意，有一些发行版把配置作为单个文件，而另外一些发行版使用一个目录或者多个文件。例如，在Red Hat企业版证书中，有一个叫/etc/sysconfig/docker/docker-storage的文件，习惯上它包含与Docker守护进程的存储选项有关的配置。

如果读者所用的发行版没有和表B-1中的名字匹配的任何文件，那么值得检查一下/etc/docker文件夹，因为那里可能有相关的文件。

在这些文件中，管理着传递给Docker守护进程启动命令的相关参数。例如，编辑的时候，下面这样的命令允许为宿主机上的Docker守

护进程设置启动参数:

```
DOCKER_OPTS=""
```

例如, 如果想把Docker根目录从默认的位置 (`/var/lib/docker`) 改到别的地方, 用户可能会把之前的那条改成:

```
DOCKER_OPTS="-g /mnt/bigdisk/docker"
```

如果所用的发行版使用systemd配置文件 (和/etc不同), 那么还可以查找systemd文件夹下的docker文件中的ExecStart这一行, 想要修改的话也可以修改。例如, 这个文件可能位于 `/usr/lib/systemd/system/service/docker`。下面是一个示例文件:

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.io
After=network.target

[Service]
Type=notify
EnvironmentFile=-/etc/sysconfig/docker
ExecStart=/usr/bin/docker -d --selinux-enabled
Restart=on-failure
LimitNOFILE=1048576
LimitNPROC=1048576

[Install]
WantedBy=multi-user.target
```

`EnvironmentFile`这一行把启动脚本指向了我们之前讨论过的有`DOCKER_OPTS`这项的文件。如果直接修改systemctl文件, 需要运行`systemctl daemon-reload`命令, 以便确保systemd守护进程采用了这个变化。

B.1 重启 Docker

只修改Docker守护进程的配置是不够的，为了采用这些变化，必须重启守护进程。注意，这会停止所有正在运行的容器，取消所有进行中的镜像下载。

B.1.1 使用systemctl重启

大多数现代Linux发行版都使用systemd来管理机器上的服务的启动。如果在命令行运行systemctl，获取好几页输出，那么宿主机就是在运行systemd。如果得到一个“command not found”消息，那么试一下B.1.2节中的方法。

如果想要更改配置，可以按如下方式停止和启动Docker:

```
$ systemctl stop docker
$ systemctl start docker
```

也可以简单地重启:

```
$ systemctl restart docker
```

通过下面这些命令来检查进度:

```
$ journalctl -u docker
$ journalctl -u docker -f
```

这里的第一行会输出Docker守护进程的可用的日志，第二行输出任何新条目的日志。

B.1.2 重启服务

如果系统运行一个基于System V的init脚本，试一下运行service --status-all。如果返回一系列的服务，就可以使用service来重启新配置的Docker:

```
$ service docker stop  
$ service docker start
```

附录C Vagrant

在本书的很多地方我们都使用虚拟机来演示需要整个机器表示或者多个虚拟机编排的技巧。

Vagrant为从命令行启动、供应以及管理虚拟机提供了一种简易的方式，它可在多个平台上获得。

C.1 设置

访问<https://www.vagrantup.com>，按照指示设置。

C.2 图形用户界面

当运行`vagrant up`来启动一台虚拟机的时候，Vagrant读取称为Vagrantfile的本地文件来确定设置。

可以在`provider`那一节创建或者修改的有用的设置是`gui`：

```
v.gui = true
```

例如，如果提供者（`provider`）是VirtualBox，一个典型的配置部分看起来可能会像下面这样：

```
config.vm.provider "virtualbox" do |v, override|
  override.vm.box      = vagrant_openshift_config['virtualbox']
  ➔ ['box_name'] unless dev_cluster
  override.vm.box_url  = vagrant_openshift_config['virtualbox']
  ➔ ['box_url'] unless dev_cluster
  override.ssh.insert_key = vagrant_openshift_config['insert_key']
  v.memory             = vagrant_openshift_config['memory'].to_i
```

```
v.cpus                = vagrant_openshift_config['cpus'].to_i
v.customize ["modifyvm", :id,
  ➡ "--cpus", vagrant_openshift_config['cpus'].to_s]
v.gui                 = false
end if vagrant_openshift_config['virtualbox']
```

在运行`vagrant up`之前，可以把`v.gui`这一行的`false`改成`true`（或者如果之前没有的话添加这一行）来获得运行虚拟机的图形用户界面（GUI）。

什么是提供者 在Vagrant里，提供者是提供虚拟机环境的程序的名字。对大多数用户来说，它是`virtualbox`，但它也可以是`libvirt`、`openstack`或者`vmware_fusion`（除此之外还有）。

C.3 内存

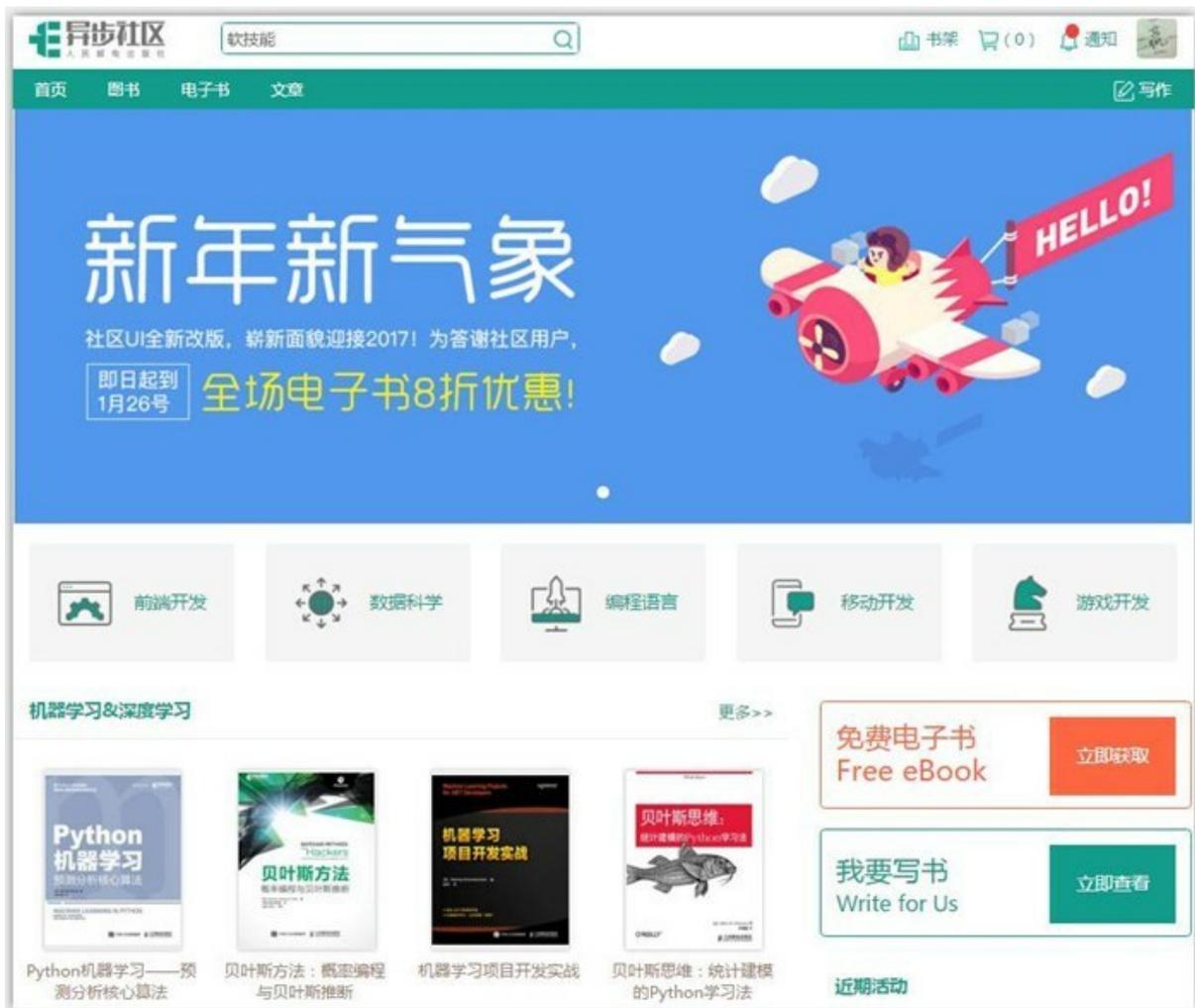
Vagrant使用虚拟机来创建其环境，这些虚拟机可能会消耗很多的内存。如果运行着一个3个节点的集群，每个虚拟机占用了2 GB的内存，用户的机器就需要6 GB的可用内存。如果机器运行得举步维艰，那么内存匮乏最有可能是元凶——唯一的解决方式就是停止所有不重要的虚拟机或者购买更大的内存。能够避免这个问题是Docker比虚拟机强大的众多原因之一。用户不用预先给容器分配资源，它们只会消耗自己需要的部分。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“57AWG”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在这一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：436746675

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询：contact@epubit.com.cn

目录

版权信息	8
版权声明	10
内容提要	11
译者简介	12
序	14
前言	17
致谢	19
关于本书	20
路线图	21
代码	22
作者在线	23
关于封面插画	24
第一部分 Docker基础	25
第1章 Docker初探	26
1.1 Docker是什么以及为什么用Docker	27
1.1.1 Docker是什么	28
1.1.2 Docker有什么好处	30
1.1.3 关键的概念	33
1.2 构建一个Docker应用程序	36
1.2.1 创建新的Docker镜像的方式	37
1.2.2 编写一个Dockerfile	39
1.2.3 构建一个Docker镜像	41
1.2.4 运行一个Docker容器	43
1.2.5 Docker分层	46
1.3 小结	48
第2章 理解Docker——深入引擎室	50
2.1 Docker的架构	50
2.2 Docker守护进程	52
技巧1 向世界开放Docker守护进程	53
技巧2 以守护进程方式运行容器	56

技巧3 将Docker移动到不同分区	61
2.3 Docker客户端	62
技巧4 使用socat监控Docker API流量	63
技巧5 使用端口连接容器	66
技巧6 链接容器实现端口隔离	69
技巧7 在浏览器中使用Docker	72
2.4 Docker注册中心	75
技巧8 建立一个本地Docker注册中心	76
2.5 Docker Hub	77
技巧9 查找并运行一个Docker镜像	78
2.6 小结	82
第二部分 Docker与开发	84
第3章 将Docker用作轻量级虚拟机	85
3.1 从虚拟机到容器	86
技巧10 将虚拟机转换为容器	86
技巧11 类宿主机容器	91
技巧12 将一个系统拆成微服务容器	94
3.2 管理容器的服务	99
技巧13 管理容器内服务的启动	100
3.3 保存和还原工作成果	104
技巧14 在开发中“保存游戏”的方式	104
技巧15 给Docker打标签	109
技巧16 在Docker Hub上分享镜像	112
技巧17 在构建时指向特定的镜像	115
3.4 进程即环境	117
技巧18 在开发中“保存游戏”的方式	118
3.5 小结	120
第4章 Docker日常	122
4.1 卷——持久化问题	122
技巧19 Docker卷——持久化的问题	123
技巧20 通过BitTorrent Sync的分布式卷	125
技巧21 保留容器的bash历史	128
技巧22 数据容器	131
技巧23 使用SSHFS挂载远程卷	134

技巧24 通过NFS共享数据	137
技巧25 开发工具容器	141
4.2 运行容器	143
技巧26 在Docker里运行GUI	143
技巧27 检查容器	147
技巧28 干净地杀掉容器	149
技巧29 使用Docker Machine来置备Docker宿主机	151
4.3 构建镜像	158
技巧30 使用ADD将文件注入到镜像	158
技巧31 重新构建时不使用缓存	162
技巧32 拆分缓存	165
4.4 保持阵型	167
技巧33 运行Docker时不加sudo	167
技巧34 清理容器	168
技巧35 清理卷	171
技巧36 解绑容器的同时不停掉它	174
技巧37 使用DockerUI来管理Docker守护进程	175
技巧38 为Docker镜像生成一个依赖图	176
技巧39 直接操作——对容器执行命令	179
4.5 小结	181
第5章 配置管理——让一切井然有序	183
5.1 配置管理和Dockerfile	183
技巧40 使用ENTRYPOINT创建可靠的定制工具	184
技巧41 在构建中指定版本来避免软件包的漂移	187
技巧42 用perl -p -i -e替换文本	189
技巧43 镜像的扁平化	191
技巧44 用alien管理外来软件包	195
技巧45 把镜像逆向工程得到Dockerfile	198
5.2 传统配置管理工具与Docker	203
技巧46 传统方式：搭配make和Docker	203
技巧47 借助Chef Solo构建镜像	208
技巧48 从源到镜像的构建	214
5.3 小即是美	223
技巧49 保持构建镜像更小的Dockerfile技巧	223

技巧50 让镜像变得更小的技巧	228
技巧51 通过BusyBox和Alpine来精简Docker镜像	231
技巧52 Go模型的最小容器	234
技巧53 使用inotifywait给容器瘦身	239
技巧54 大也可以美	244
5.4 小结	246
第三部分 Docker与DevOps	248
第6章 持续集成：加快开发流水线	249
6.1 Docker Hub自动化构建	250
技巧55 使用Docker Hub workflow	250
6.2 更有效的构建	256
技巧56 使用eatmydata为I/O密集型构建提速	256
技巧57 设置一个软件包缓存用于加快构建速度	259
技巧58 在Docker内部运行Selenium测试	264
6.3 容器化CI过程	269
技巧59 包含一个复杂的开发环境	269
技巧60 在一个Docker容器里运行Jenkins主服务器	276
技巧61 使用Jenkins的Swarm插件扩展CI	280
6.4 小结	285
第7章 持续交付：与Docker原则完美契合	287
7.1 在CD流水线上与其他团队互动	288
技巧62 Docker契约——减少摩擦	289
7.2 推动Docker镜像的部署	292
技巧63 手动同步注册中心镜像	292
技巧64 通过受限连接交付镜像	293
技巧65 以TAR文件方式共享Docker对象	297
7.3 为不同环境配置镜像	300
技巧66 使用etcd通知容器	300
7.4 升级运行中的容器	304
技巧67 使用confd启用零停机时间切换	305
7.5 小结	310
第8章 网络模拟：无痛的现实环境测试	312
8.1 容器通信——超越手工链接	312

技巧68 一个简单的Docker Compose集群	313
技巧69 一个使用Docker Compose的SQLite服务器	319
技巧70 使用Resolvable通过DNS查找容器	325
8.2 使用Docker来模拟真实世界的网络	329
技巧71 使用Comcast模拟有问题的网络	329
技巧72 使用Blockade模拟有问题的网络	334
8.3 Docker和虚拟网络	339
技巧73 使用Weave建立一个基底网络	341
技巧74 Docker的网络与服务功能	346
8.4 小结	350
第四部分 生产环境中的Docker	352
第9章 容器编排：管理多个Docker容器	353
9.1 简单的单台宿主机	355
技巧75 使用systemd管理宿主机上的容器	355
技巧76 使用systemd编排宿主机上的容器	361
9.2 多宿主机Docker	365
技巧77 使用Helios手动管理多宿主机Docker	366
技巧78 基于Swarm的无缝Docker集群	374
技巧79 使用Kubernetes集群	380
技巧80 在Mesos上构建框架	390
技巧81 使用Marathon细粒度管理Mesos	401
9.3 服务发现：我们有什么	406
技巧82 使用Consul来发现服务	407
技巧83 使用Registrator进行自动化服务注册	418
9.4 小结	421
第10章 Docker与安全	423
10.1 Docker访问权限及其意味着什么	423
你在乎吗	425
10.2 Docker中的安全手段	426
技巧84 限制能力	426
技巧85 Docker实例上的HTTP认证	432
技巧86 保护Docker API	438
10.3 来自Docker以外的安全	444
技巧87 OpenShift——一个应用程序平台即服务	444

技巧88 使用安全选项	457
10.4 小结	468
第11章 一帆风顺——生产环境中的Docker以及运维上的考量	470
11.1 监控	470
技巧89 记录容器的日志到宿主机的syslog	471
技巧90 把Docker日志发送到宿主机的输出系统	475
技巧91 使用cAdvisor监控容器	478
11.2 资源控制	481
技巧92 限制容器可以运行的内核	481
技巧93 给重要的容器更多CPU	483
技巧94 限制容器的内存使用	485
11.3 Docker的系统管理员用例	488
技巧95 使用Docker来运行cron作业	488
技巧96 通过“保存游戏”的方法来备份	492
11.4 小结	495
第12章 Docker生产环境实践——应对各项挑战	497
12.1 性能——不能忽略宿主机	497
技巧97 从容器访问宿主机资源	497
技巧98 Device Mapper存储驱动和默认的容器大小	505
12.2 在容器出问题时——调试Docker	508
技巧99 使用nsenter调试容器的网络	508
技巧100 无须重新配置，使用tcpflow进行实时调试	513
技巧101 调试在特定宿主主机上出问题的容器	515
12.3 小结	521
附录A 安装并使用Docker	522
A.1 虚拟机的方式	522
A.2 连接到外部Docker服务器的Docker客户端	523
A.3 原生Docker客户端和虚拟机	523
Windows上的Docker	523
A.4 获得帮助	527
附录B Docker配置	528
B.1 重启Docker	529

B.1.1 使用systemctl重启	530
B.1.2 重启服务	530
附录C Vagrant	532
C.1 设置	532
C.2 图形用户界面	532
C.3 内存	533
欢迎来到异步社区！	534
异步社区的来历	534
社区里都有什么？	535
购买图书	535
下载资源	535
与作译者互动	536
灵活优惠的购书	536
纸电图书组合购买	536
社区里还可以做什么？	537
提交勘误	537
写作	537
会议活动早知道	537
加入异步	538