

Docker in Production

Docker生产环境

实践指南

[美] Joe Johnston [西] Antoni Batchelli
[英] Justin Cormack [美] John Fiedler 著
[英] Milos Gajdos



DockOne.io
Community of Container

吴健兴 梁晓勇 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

目 录

版权信息	
版权声明	
内容提要	
对本书的赞誉	
译者介绍	
前言	
本书面向的读者	
谁真的在生产环境中使用Docker	
为什么使用Docker	
开发环境与生产环境	
我们所说的“生产环境”	
功能内置与组合工具	
哪些东西不要Docker化	
技术审稿人	
第1章 入门	
1.1 术语	
1.1.1 镜像与容器	
1.1.2 容器与虚拟机	
1.1.3 持续集成/持续交付	
1.1.4 宿主机管理	
1.1.5 编排	
1.1.6 调度	
1.1.7 发现	
1.1.8 配置管理	
1.2 从开发环境到生产环境	
1.3 使用Docker的多种方式	
1.4 可预期的情况	
为什么Docker在生产环境如此困难	
第2章 技术栈	
2.1 构建系统	
2.2 镜像仓库	
2.3 宿主机管理	
2.4 配置管理	

[2.5 部署](#)

[2.6 编排](#)

[第3章 示例：极简环境](#)

[3.1 保持各部分的简单](#)

[3.2 保持流程的简单](#)

[3.3 系统细节](#)

[利用systemd](#)

[3.4 集群范围的配置、通用配置及本地配置](#)

[3.5 部署服务](#)

[3.6 支撑服务](#)

[3.7 讨论](#)

[3.8 未来](#)

[3.9 小结](#)

[第4章 示例：Web环境](#)

[4.1 编排](#)

[4.1.1 让服务器上的Docker进入准备运行容器的状态](#)

[4.1.2 让容器运行](#)

[4.2 连网](#)

[4.3 数据存储](#)

[4.4 日志](#)

[4.5 监控](#)

[4.6 无须担心新依赖](#)

[4.7 零停机时间](#)

[4.8 服务回滚](#)

[4.9 小结](#)

[第5章 示例：Beanstalk环境](#)

[5.1 构建容器的过程](#)

[部署/更新容器的过程](#)

[5.2 日志](#)

[5.3 监控](#)

[5.4 安全](#)

[5.5 小结](#)

[第6章 安全](#)

[6.1 威胁模型](#)

[6.2 容器与安全性](#)

[6.3 内核更新](#)

[6.4 容器更新](#)

[6.5 suid及guid二进制文件](#)

[6.6 容器内的root](#)

[6.7 权能](#)

[6.8 seccomp](#)

[6.9 内核安全框架](#)

[6.10 资源限制及cgroup](#)

[6.11 ulimit](#)

[6.12 用户命名空间](#)

[6.13 镜像验证](#)

[6.14 安全地运行Docker守护进程](#)

[6.15 监控](#)

[6.16 设备](#)

[6.17 挂载点](#)

[6.18 ssh](#)

[6.19 私钥分发](#)

[6.20 位置](#)

[第7章 构建镜像](#)

[7.1 此镜像非彼镜像](#)

[7.1.1 写时复制与高效的镜像存储与分发](#)

[7.1.2 Docker对写时复制的使用](#)

[7.2 镜像构建基本原理](#)

[7.2.1 分层的文件系统和空间控管](#)

[7.2.2 保持镜像小巧](#)

[7.2.3 让镜像可重用](#)

[7.2.4 在进程无法被配置时，通过环境变量让镜像可配置](#)

[7.2.5 让镜像在Docker变化时对自身进行重新配置](#)

[7.2.6 信任与镜像](#)

[7.2.7 让镜像不可变](#)

[7.3 小结](#)

[第8章 存储Docker镜像](#)

[8.1 启动并运行存储的Docker镜像](#)

[8.2 自动化构建](#)

[8.3 私有仓库](#)

[8.4 私有registry的扩展](#)

[8.4.1 S3](#)

[8.4.2 本地存储](#)

[8.4.3 对registry进行负载均衡](#)

[8.5 维护](#)

[8.6 对私有仓库进行加固](#)

[8.6.1 SSL](#)

[8.6.2 认证](#)

[8.7 保存/载入](#)

[8.8 最大限度地减小镜像体积](#)

[8.9 其他镜像仓库方案](#)

[第9章 CI/CD](#)

[9.1 让所有人都进行镜像构建与推送](#)

[9.2 在一个构建系统中构建所有镜像](#)

[9.3 不要使用或禁止使用非标准做法](#)

[9.4 使用标准基础镜像](#)

[9.5 使用Docker进行集成测试](#)

[9.6 小结](#)

[第10章 配置管理](#)

[10.1 配置管理与容器](#)

[10.2 面向容器的配置管理](#)

[10.2.1 Chef](#)

[10.2.2 Ansible](#)

[10.2.3 Salt Stack](#)

[10.2.4 Puppet](#)

[10.3 小结](#)

[第11章 Docker存储引擎](#)

[11.1 AUFS](#)

[11.2 DeviceMapper](#)

[11.3 BTRFS](#)

[11.4 OverlayFS](#)

[11.5 VFS](#)

[11.6 小结](#)

[第12章 Docker 网络实现](#)

[12.1 网络基础知识](#)

[12.2 IP地址的分配](#)

[端口的分配](#)

[12.3 域名解析](#)

[12.4 服务发现](#)

[12.5 Docker高级网络](#)

[12.6 IPv6](#)

[12.7 小结](#)

[第13章 调度](#)

[13.1 什么是调度](#)

[13.2 调度策略](#)

[13.3 Mesos](#)

[13.4 Kubernetes](#)

[13.5 OpenShift](#)

[Red Hat公司首席工程师Clayton Coleman的想法](#)

[第14章 服务发现](#)

[14.1 DNS服务发现](#)

[DNS服务器的重新发明](#)

[14.2 Zookeeper](#)

[14.3 基于Zookeeper的服务发现](#)

[14.4 etcd](#)

[基于etcd的服务发现](#)

[14.5 consul](#)

[14.5.1 基于consul的服务发现](#)

[14.5.2 registrator](#)

[14.6 Eureka](#)

[基于Eureka的服务发现](#)

[14.7 Smartstack](#)

[14.7.1 基于Smartstack的服务发现](#)

[14.7.2 Nerve](#)

[14.7.3 Synapse](#)

[14.8 nsqlookupd](#)

[14.9 小结](#)

[第15章 日志和监控](#)

[15.1 日志](#)

[15.1.1 Docker原生的日志支持](#)

[15.1.2 连接到Docker容器](#)

[15.1.3 将日志导出到宿主机](#)

[15.1.4 发送日志到集中式的日志平台](#)

[15.1.5 在其他容器一侧收集日志](#)

[15.2 监控](#)

[15.2.1 基于宿主机的监控](#)

[15.2.2 基于Docker守护进程的监控](#)

[15.2.3 基于容器的监控](#)

[15.3 小结](#)
[DockOne社区简介](#)
[看完了](#)

版权信息

书名：Docker生产环境实践指南

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] Joe Johnston [西] Antoni Batchelli
 [英] Justin Cormack [美] John Fiedler
 [英] Milos Gajdos

译 吴佳兴 梁晓勇

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

Copyright © 2015 Bleeding Edge Press. All rights reserved. First published in the English language under the title *Docker in Production* by Joe Johnston, Antoni Batchelli, Justin Cormack, John Fiedler, and Milos Gajdos by Bleeding Edge Press, an imprint of Backstop Media.

本书中文简体版由Backstop Media LLC授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

内容提要

本书围绕“**Docker**该如何应用到生产环境”这一核心问题展开。在本书中，读者将接触到多个IT企业应用**Docker**到生产环境的成功案例，了解**Docker**实际投产时将会面临的问题，以及它与现有基础设施存在的矛盾与冲突，了解构建**Docker**生态系统所需的配套设施，包括安全、构建镜像、持续集成/持续交付、镜像存储、配置管理、网络实现、服务发现、持久化存储以及日志监控等模块的具体选型方案及利弊所在。本书编写时一些案例参考的**Docker**版本是**Docker 1.6**或**Docker 1.7**。

本书要求读者具备一定的容器管理和运维的基础知识，适合在生产环境中使用**Docker**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。

对本书的赞誉

经过2015年Docker项目的飞速发展，国内的互联网企业开始关注容器技术在生产环境下的使用案例。但是，由于与Docker相关的技术资料还没有得到系统性地整理，国内企业一直在各种实际场景中进行实践，所以本书的出现正好填补了当前生产环境实践的实际需要，让国内容器技术的推广迈出坚实的一步。

——肖德时，数人科技CTO

2015年，不管是传统IT企业、互联网巨头，还是初创公司，大大小小的公司都在实践如何在生产环境中应用Docker来解决其实际问题。在这个过程中会遇到各种各样的问题，如网络和存储驱动如何选型，资源限制不够彻底怎么办，镜像仓库如何做权限控制，容器内健康状况如何监测，容器有哪些安全隐患，以及在容器集群化过程中会遇到哪些问题，如服务发现、弹性伸缩等。这些问题都没有标准答案，这也激发了工程师们的探索乐趣，于是就有了各种各样的实战方案。

本书恰好就是对一些实战方案的归纳和总结。相对于市面上其他Docker图书，我认为这本书价值更大：对工程师们来说，将技术应用到“生产环境”中才是最大的价值和乐趣所在！

——陈轶飞，原百度PaaS平台负责人，国内最早大规模应用Docker的实践者

我在Google从事基于容器的基础设施和集群管理研发多年，许多关于容器使用最佳实践的知识都是通过“代代相传、口口相传”的方式获得的。在Docker迅速流行的同时，在开源社区里却缺少如Google（或其他公司）内部“老工程师对新人倾囊相授实践+真理”的这种奢侈。

在众多讲述Docker自身原理、使用方法的书中，这本书从生产角度出发，将作者在实战中积累的一线经验系统地汇总成了基于Docker搭建生

产系统的经验，值得我们借鉴。

——张鑫，才云科技CEO

在深入学习Docker时将面对这样的问题：Docker的流行催生出大量与之相关的新技术，这些新技术有哪些，它们是什么关系，又该如何正确选择？Docker也有适用场景，到底哪些场景适合用？把Docker用在生产环境，是否有成功案例或最佳实践，又该如何操作？如果你正面对这些问题，本书就非常适合你。另外，本书不仅以可以直接应用的真实生产环境示例贯穿始终，还讲解了技术的权衡之道，是Docker进阶的难得好书。

——刘凡，好雨云创始人

Docker进入大家视野已经有段时间了，也历经了多个重要的版本升级。有报告表明，在一线互联网公司中，已经有27%的企业在生产环节中使用到了Docker，先行者已开始从中获益。但目前Docker资料仍停留在概念或者实验级别，关于真正在生产环境中使用Docker的内容则少之又少，最佳实践方面的信息与资料也十分稀缺。

Docker技术对于IT架构乃至软件架构的改变是巨大的，对于想在生产环境中使用Docker的企业和团队来讲，只掌握概念和基本原理是不够的，如何能使用Docker解决自身问题，获得由此带来的收益，需要更多的生产实践方面的内容。

本书以生产部署为背景，讲述Docker在真实环境中的使用，能够给读者一个很好的参考，进而达到让读者“举一反三”的效果，使其能让自身的IT架构提升到一个新的技术高度。

——周东波，首都在线总工程师

Docker的出现不仅是单一技术的诞生，而是整个开发、测试、运维体系的一次变革，是从传统的底层主机化真正向资源服务化转变。Docker也同时改变了广大技术人员的思维模式，让原先枯燥重复的工作变得有趣。

本书非常系统地讲解了Docker在生产环境的搭建和应用，同时详细地讲解了其镜像、网络、存储、安全等方面的知识，是有志于研究Docker的

读者必备的书籍。

——孙斌，去哪儿网运维总监

Docker是互联网领域最近两年最火热的技术词语，没有之一。以Docker为代表的容器技术，通过务实的工程创新，正在影响无数企业的IT基础设施以及主流的公有云服务。国内越来越多的互联网企业开始将Docker应用到生产环境中。可以预见，掌握Docker相关知识即将成为软件工程师或运维工程师的必备技能。DockOne.io社区组织翻译的这本书恰逢其时，值得每名IT从业者阅读。

——刘海锋，京东云平台总架构师

容器技术已经是2015年最热的技术。容器技术已经存在数年，为什么Docker作为容器技术的布道者能够有翻天覆地的影响？为什么容器技术能够给传统应用带来好处？如何把传统应用容器化并通过DevOps方式简洁化？如何通过具体参数和方法解决现实问题？如何使用Docker以及Docker实践？

其实还有很多问题需要解决，如应用的复杂性和难维护性、开发与运维的脱离性，以及成本不断攀升的现实性。本书从简入繁，通过具体案例解释具体问题，通过实践帮助读者理解具体问题，对正在或者即将使用基于容器的DevOps的读者有很多益处。

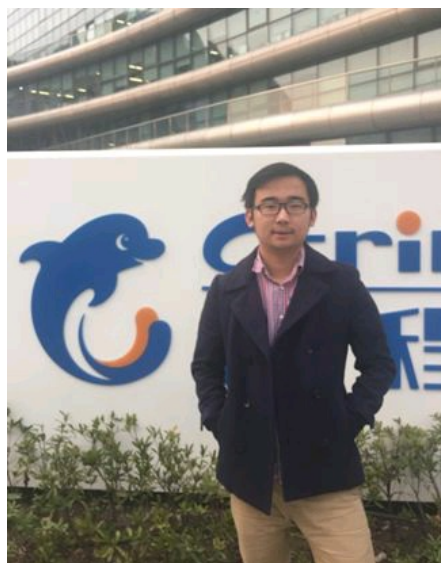
——陈冉，Linker Networks CTO & VP

Docker的出现使得IaaS和PaaS的界限进一步模糊化，引领了云计算技术变革。时下诸多公司正如火如荼地探索在生产环境中如何玩转Docker。本书的特点在于不仅介绍“是什么”，更进一步探讨“如何用”和“为什么用”；不限于某种特定的框架，而是对比各种不同的方案。书中涵盖容器监控、配置管理、安全、调度、持续交付等生产实践经验，想在生产环境中玩转Docker的技术人员定能从中获益。

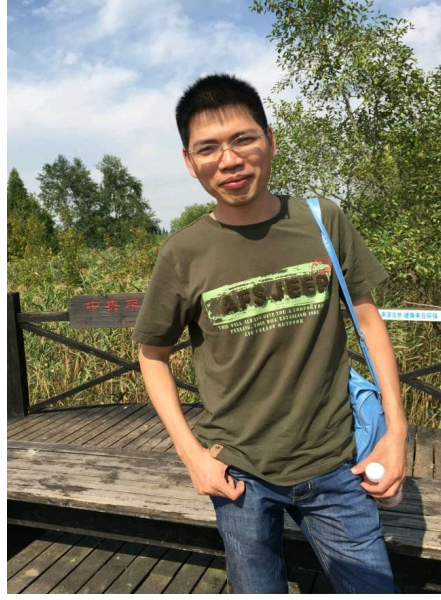
——吴毅挺，携程CIS-系统研发部总监

译者介绍

吴佳兴，毕业于华东理工大学计算机系，目前是携程网系统研发团队的一名DevOps工程师，主要研究方向有Python开发、运维自动化、配置管理及PaaS平台的构建等。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（wjx_colstu@hotmail.com）。



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，非著名互联网从业者。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），交流指教。



前言

Docker是基础设施的新成员。很少有新兴技术能像它这样，在DevOps和基础设施领域中快速风靡起来。在不到两年的时间内，Google、亚马逊、微软、IBM以及几乎所有云供应商都宣布支持运行Docker容器。大量与Docker相关的创业公司在2014年和2015年年初都获得了风险资本的投资。Docker开源技术背后的同名公司——Docker公司，在2015年第一季度的D轮融资中估值为10亿美元左右。

大大小小的公司都在转换其应用，使之运行于容器内，以此实现面向服务架构（SOA）和微服务。不论是参加从旧金山到柏林的任何DevOps聚会，还是阅读最热门的公司工程博客，都可以看出全世界的运维领导者们如今都在云上运行Docker。

毫无疑问，容器已经成为应用程序打包和基础设施自动化的重要组成部分。但有一个棘手的问题，促使本书作者和同僚们创作了另一本Docker图书。

本书面向的读者

具有中高级DevOps和运维背景的读者将从本书获益最多。因而，强烈建议读者应具备在生产环境中运行服务器以及创建和管理容器这两方面的基本经验。

很多图书和博客文章已经涵盖了与Docker安装及运行相关的话题，但能把在生产环境中运行Docker时产生的大量甚至是令人挠头的关注点结合在一起来的材料则少之又少。不用担心，如果你很喜欢《盗梦空间》（Inception）这部电影，在云服务器的虚拟机中运行容器会让你感觉很自然。

本书将带读者深入理解生产环境中架构的组成部分、关注点，以及如何运行基于Docker的基础设施。

谁真的在生产环境中使用Docker

换个更深刻的说法，对于在真实生产环境中使用Docker遇到的问题，如何找到解决之道？本书综合了访谈、真实公司端到端的生产环境实例，以及来自DevOps杰出专家的参考文献，以此来解答这些问题。虽然本书包含了一些有用的示例，但它并不是一本复制粘贴的“教程式”参考书。相反，本书侧重于生产环境中对前沿技术进行评估、风险抵御及运维所需的实践理论和经验。

作为作者，我们希望这本书所包含的内容能够为那些正在评估如何及何时将Docker相关技术引入其DevOps栈的团队提供一个可靠的决策指南，这远比代码片段要来得长久。

生产环境中运行的Docker为企业提供了多个新的运行和管理服务器端软件的方式。很多现成的用例讲解了如何使用Docker，但很少有公司公开分享过他们的全栈生产环境经验。本书汇集了作者在生产环境中运行Docker的多个实例和一组选定的友好公司分享的使用经验。

为什么使用Docker

Docker所使用的底层容器技术已经存在了很多年，甚至早于dotCloud这家平台即服务（PaaS）创业公司，即后来我们所熟知的Docker。在dotCloud之前，许多知名的公司（如Heroku和Iron.io）已经在生产环境中运行大型容器集群，以获取额外的超越虚拟机的性能优势。与虚拟机相比，在容器中运行软件赋予了这些公司秒级而非分钟级的实例启动与停止的能力，同时能使用更少的机器运行更多实例。

既然这项技术并不新鲜，为什么Docker能获得如此巨大的成功呢？主要是因为它的易用性。Docker创造了一种统一的方式，通过简便的命令行及HTTP API工具来打包、运行和维护容器。这种简化降低了将应用程

序及其运行时环境打包成一个自包含镜像的入门门槛，使之变得可行且有趣，而不需要类似Chef、Puppet及Capistrano之类的配置管理和发布系统。

Docker提供了一种统一手段，将应用程序及其运行时环境打包到一个简单的Dockerfile里，这从根本上改变了开发人员与DevOps团队之间的交互界面。从而极大简化了开发团队与DevOps之间的沟通需求与责任边界。

在Docker出现之前，各个公司的开发与运维团队之间经常会爆发史诗般的战争。开发团队想要快速前进，整合最新版的软件及依赖，以及持续部署。运维团队则以保证稳定为己任，他们负责把关可以运行于生产环境中的内容。如果运维团队对新的依赖或需求感到不适，他们通常会站在保守的立场上，要求开发人员使用旧版软件以确保糟糕的代码不会搞垮整台服务器。

Docker一下子改变了DevOps的决策思维，从“基本上说不”变成了“好的，只要运行在Docker中就可以”，因为糟糕的代码只会让容器崩溃，而不会影响到同一服务器上的其他服务。在这种泛型中，DevOps有效地负责为开发人员提供PaaS，而开发人员负责保证其代码能正常运行。如今，很多团队将开发人员加入到PagerDuty中，以监控他们在生产环境中的代码，让DevOps和运维人员专注于平台的稳定运行及安全。

开发环境与生产环境

对大多数团队而言，采用Docker是受开发人员更快的迭代和发布周期需求推动的。这对于开发环境是非常有益的，但对于生产环境，在单台宿主机上运行多个Docker容器可能会导致安全漏洞，这一点我们将在第6章“安全”中讲述。事实上，几乎所有关于在生产环境中运行Docker的话题都是围绕着将开发环境与生产环境区分开的两个关注点进行的：一是编排，二是安全。

有些团队试图让开发环境和生产环境尽可能保持一致。这种方法看起来很好，但是限于开发环境这样做所需定制工具的数量又或者说模拟云服务（如AWS）的复杂度，这种方法并不实际。

为了简化这本书的范畴，我们将介绍一些部署代码的用例，但判定最佳

开发环境设置的实践机会将留给读者。作为基本原则之一，尽量保持生产环境和开发环境的相似性，并使用一个持续集成/持续交付（CI/CD）系统以获取最佳结果。

我们所说的“生产环境”

对于不同的团队，生产环境意味着不同的东西。在本书中，我们所说的生产环境是指真实客户用于运行代码的环境。这是相对于开发环境、预演环境及测试环境而言的，后者的停机时间不会被客户感知到。

在生产环境中，**Docker**有时是用于接收公共网络流量的容器，有时则是用于处理来自队列负荷的异步的后台作业。不管哪种用途，在生产环境中运行**Docker**与在其他环境中运行相比，最主要的差异都是需要在其安全性与稳定性上投入较多的注意力。

编写本书的动力之一是，与**Docker**相关的文档和博客文章中缺乏对实际生产环境与其他环境的明确区分。我们认为，80%的**Docker**博客文章中的建议在尝试在生产环境中运行6个月之后会被放弃（或至少修改）。为什么？因为大多数博客文章中举的都是理想化的例子，使用了最新、最好用的工具，一旦某个极端的情况变成了致命缺陷，这些工具将被遗弃（或延期），被更简单的方法所取代。这是**Docker**技术生态系统现状的一个反映，而非技术博客的缺陷。

总的来说，生产环境很难管理。**Docker**简化了从开发到生产的工作流程，但同时增加了安全和编排的复杂度（更多关于编排的内容参见第4章）。

为了节省时间，下面给出本书的重点综述。

所有在生产环境中运行**Docker**的团队，都会在传统的安全最佳实践上做出一项或多项妥协。如果无法完全信任容器内运行的代码，那么就只得选用容器与虚拟机一对一的拓扑方式。对于很多团队而言，在生产环境中运行**Docker**的优势远远大于其带来的安全与编排问题。如果遇到工具方面的问题，请等待一到两个月，以便**Docker**社区对其进行修复，不要浪费时间去修补其他人的工具。保持**Docker**设置最小化。让一切自动

化。最后，对成熟的编排工具（如Mesos、Kubernetes等）的需求远比想象的要少得多。

功能内置与组合工具

Docker社区一个常见的口头禅是“电池内置但可移除”，指的是将很多功能捆绑在一起的单体二进制文件，这有别于传统Unix哲学下相对较小、功能单一、管道化的二进制文件。

这种单体式的做法是由两个主要因素决定的：（1）使Docker易于开箱即用；（2）Golang缺少动态链接。Docker及多数相关工具都是用Google的Go编程语言编写的，该语言可以简化高并发代码的编写与部署。虽然Go是一门出色的编程语言，但用它来构建的Docker生态系统中也因此迟迟无法实现一个可插拔的架构，在这种架构中可以很容易用替代品对工具进行更换。

如果读者有Unix系统背景，最好是编译自己的精简版Docker守护进程，以符合生产环境的需求。如果读者有开发背景，预计到2015年下半年，Docker插件将成为现实^[1]。在此期间，估计Docker生态系统中的工具将会出现明显的重叠现象，某些情况下甚至是相互排斥的。

换句话说，要让Docker运行于生产环境中，用户的一半工作将是决定哪些工具对自己的技术栈最有意义。与DevOps所有事情一样，先从最简单的解决方案入手，然后在必要时增加其复杂性。

2015年5月，Docker公司发布了Compose、Machine及Swarm，与Docker生态系统内的同类工具进行竞争。所有这些工具都是可选的，请根据实际情况对其进行评估，而不要认为Docker公司提供的工具就一定是最佳解决方案。

探索Docker生态系统时的另一项关键建议是：评估每个开源工具的资金来源及其商业目标。目前，Docker公司和CoreOS经常发布工具，以争夺关注度和市场份额。一个新工具发布后，最好等上几个月，看看社区的反应，不要因为它看起来很酷就切换到最新、最好用的工具上。

哪些东西不要Docker化

最后一个关键点是，不要期望能在Docker容器中运行所有东西。Heroku风格的“十二要素”（12 factor）应用是最容易Docker化的，因为它们不维护状态。在理想的微服务环境中，容器能在几毫秒内启动、停止而不影响集群的健康或应用程序的状态。

类似ClusterHQ这样的创业公司正着手实现Docker化数据库和有状态的应用程序，但眼下，由于编排和性能方面的原因，可能需要继续直接在虚拟机或裸机上运行数据库。

Docker还不适用于任何需要动态调整CPU和内存要求的应用^[2]。允许动态调整的代码已经完成，但尚不清楚何时才能在一般的生产环境中投入使用。目前，若对容器的CPU和内存的限制进行调整，需要停止并重新启动容器。

另外，对网络吞吐量有高要求的应用进行最佳优化时不要使用Docker，因为Docker使用iptables来完成宿主机IP到容器IP的NAT转换。通过禁用Docker的NAT来提升网络性能是可行的，但这是一个高级的使用场景，很少有团队会在生产环境中这么做。

技术审稿人

衷心感谢以下技术审稿人提供的早期反馈及细致的评论：Mika Turunen、Xavier Bruhiere和Felix Rabe。

[1] Docker 1.7版中正式引入了插件系统。——译者注

[2] Docker 1.10版中新增的docker update命令可实现CPU和内存的动态调整。——译者注

第1章 入门

建立Docker生产环境系统的首要任务，是以一个有助于想象各组件如何相互配合的方式来理解其术语。与其他快速发展的技术生态系统一样，我们可以预见，Docker野心勃勃的市场营销、不完善的文档以及过时的博客文章将造成使用者对各个工具职责理解上的混乱。

我们将在本章中定义贯穿全书的术语和概念，而非提供一份统一的Docker百科全书。通常情况下，我们的定义与生态系统中的大体一致，但如果你所阅读的博客文章中使用了不同的术语也不用太过惊讶。

在本章中，我们将介绍在生产环境中运行Docker的核心概念以及不涉及具体技术的容器常识。在随后的章节中，我们将讨论真实世界的生产环境用例，并详细说明其组件和供应商信息。

1.1 术语

下面让我们来看一下本书所采用的Docker术语。

1.1.1 镜像与容器

- 镜像是指文件系统快照或tar包。
- 容器是指镜像的运行态。

1.1.2 容器与虚拟机

- 虚拟机持有整个操作系统和应用程序的快照。
- 虚拟机运行着自己的内核。

- 虚拟机可以运行Linux之外的其他操作系统。
- 容器只持有应用程序，不过应用程序的概念可以延伸到整个Linux发行版。
- 容器共享宿主机的内核。
- 容器只能运行Linux，不过在同一宿主机上运行的每个容器都可包含不同的发行版。

1.1.3 持续集成/持续交付

在应用程序新代码提交或触发其他条件时，系统自动构建新镜像并进行部署。

1.1.4 宿主机管理

设置/配备一台物理服务器或虚拟机以便用于运行Docker容器的过程。

1.1.5 编排

编排（orchestration，也称编配）这个术语在Docker生态系统中有多种含义。通常情况下，它包括调度和集群管理，不过有时也包括了宿主机管理。

在本书中，我们将编排作为一个松散的总称，包括容器调度的过程、集群的管理、容器的链接（发现），以及网络流量路由。或者换句话说，编排是个控制器进程，用于决定在哪里运行容器，以及如何让集群知道可用的服务。

1.1.6 调度

用于决定哪些容器可以以给定的资源约束（如CPU、内存和IO）运行在哪些宿主机上。

1.1.7 发现

容器如何公开服务给集群，以及发现如何查找其他服务并与之通信的过

程。举个简单的用例：一个网站应用容器发现如何连接到数据库服务。

Docker文档中的发现是指将容器链接在一起，不过在生产级系统中，通常使用的是更复杂的发现机制。

1.1.8 配置管理

配置管理过去常常指的是Docker出现之前的自动化工具，如Chef和Puppet。大多数的DevOps团队正在转移到Docker上，以消除这类配置管理系统的复杂度。

在本书的示例中，配置管理工具只用于配备具有Docker和少量其他东西的宿主机。

1.2 从开发环境到生产环境

本书着重于生产环境或非开发环境中的Docker，这意味着我们不会花太多的篇幅在开发环境中Docker的配置和运行上。但由于所有服务器都在运行代码，如何看待在Docker和非Docker系统中的应用程序代码还是值得简单讨论一下的。

与Chef、Puppet和Ansible这类传统配置系统不同，Docker最好的使用方式是将应用程序代码预先打包成一个Docker镜像。镜像通常包含所有的应用程序代码、运行时的依赖以及系统的需求。而包含数据库凭证和其他敏感信息的配置文件通常在运行时添加，而非内建到镜像中。

有些团队会在开发机上手工构建Docker镜像，然后推送到镜像仓库，之后再从仓库中拉取镜像到生产环境宿主机中。这是个很简单的用例。虽然行得通，但从工作流和安全角度考虑并不理想。

一个更常见的生产环境示例是，使用持续集成/持续交付系统在应用程序代码或Dockerfile文件发生变更时自动构建新镜像。

1.3 使用Docker的多种方式

过去的几年时间，科技发生了巨大变化，从物理服务器到虚拟服务器，再到拥有PaaS环境的云计算。不论是否采用了全新架构，Docker镜像都可以在当前环境中很容易地被使用。要使用Docker，并不需要立即从单体应用程序迁移到面向服务架构。有很多用例允许在不同层次上集成Docker。

Docker常用于以下场景。

- 使用以镜像为基础的部署方式取代类似Capistrano的代码部署系统。
- 安全地在同一台服务器中运行遗留应用和新应用。
- 使用一个工具链循序渐进地迁移到面向服务架构。
- 管理云端或裸机上的水平扩展性和弹性。
- 确保从开发环境到预演环境到生产环境跨环境的一致性。
- 简化开发人员的机器设置和一致性。

将应用的后台程序迁移到Docker集群中，同时保持网页服务器和数据库服务器不变是开始使用Docker的常见示例。另一示例是将应用的部分REST API迁移到Docker中运行，前端使用Nginx代理在遗留服务和Docker集群之间路由通信。通过使用此类技术，团队可以渐进式地从单体应用无缝地迁移到面向服务架构。

如今的应用程序往往需要几十个第三方库，用于加速功能开发或连接第三方SaaS和数据库服务。每个库都可能产生bug，或是让用户陷入版本依赖的泥沼。再加上库的频繁更改，要在基础设施上完成工作代码的持续部署而不引起失败，压力巨大。

Docker可贵的镜像思想使得技术团队在部署工作代码时，不论是单体架构、面向服务或是二者的混合，由于代码及其依赖项捆绑在同一个镜像中，所使用的方式对每次部署都是可测试、可重复、文档化且一致的。一旦一个镜像构建完毕，就可以部署到任意多个运行着Docker守护进程的服务器上。

另外一个常见的Docker用例是跨环境部署一个单一容器，其典型的代码路径是从开发环境到预演环境再到生产环境。容器为整个代码路径提供了一个一致的、可测试的环境。

作为一个开发人员，Docker模型允许在其个人电脑上调试与生产环境完全一致的代码。开发人员可以很容易地下载、运行和调试有问题的生产环境镜像，且无需事先对本地开发环境进行修改。

1.4 可预期的情况

在生产环境中运行Docker容器困难不小，但还是能实现的。每天都有越来越多公司开始在生产环境中运行Docker。如同所有的基础设施一样，我们建议以小规模入手，然后渐进式地完成迁移。

为什么Docker在生产环境如此困难

对生产环境有很多要求：安全可靠的部署、健康检查、最小或零停机时间、从失败中恢复的能力（回滚）、一个集中存储日志的方式、一种分析或调试应用的方式，以及一种聚合监控参数的方式。类似Docker这样的新技术虽然使用起来非常有趣，但还需要时间来完善。

Docker在可移植性、一致性以及打包具有众多依赖的服务这些方面非常有优势。多数团队会因为以下一个或多个痛点而坚持使用Docker。

- 一个应用的不同部分使用大量不同的依赖。
- 支持使用旧依赖的遗留应用程序。
- 开发团队与DevOps之间的工作流问题。

本书中我们所采访的团队，有一个共同的警示：切勿尝试在一个组织内让采用Docker这事一蹴而就。即便运维团队已经为采用Docker做好了充分的准备，也请记住，过渡到Docker通常意味着将管理依赖的重任推给了开发人员。虽然很多开发人员都渴求这种自主权，以便加快迭代，但并非每位开发人员都有能力或兴趣将其列入自己的责任范围。为了能有一个良好的Docker工作流，还是需要花些时间来转变企业文化。

在第2章中，我们将阐述Docker的技术栈。

第2章 技术栈

生产环境的Docker设置包括了一些基本的架构组件，这些组件对运行容器化的及传统的服务器集群来说是通用的。在很多方面，可以简单地认为构建和运行容器的方式与当前构建和运行虚拟机的方式是一样的，只是使用了一套新的工具和技术。

- (1) 构建并保存镜像快照。
- (2) 将镜像上传到仓库中。
- (3) 下载镜像到某台宿主机中。
- (4) 以容器方式运行镜像。
- (5) 将容器连接到其他服务上。
- (6) 路由流量到容器中。
- (7) 将容器日志发送到指定位置。
- (8) 监控容器。

与虚拟机不同的是，容器通过将宿主机（裸机或虚拟机）与应用程序服务隔离，从而提供了更高的灵活性。这为构建和配备流程带来了直接的改善，但由于额外的容器嵌入层，会增加一些开销。

典型的Docker技术栈将包括用于解决以下关注点的组件：

- 构建系统；
- 镜像仓库；
- 宿主机管理；
- 配置管理；
- 部署；
- 编排；
- 日志；
- 监控。

2.1 构建系统

- 如何构建镜像，并将其推送到镜像仓库中？
- Dockerfile位于何处？

构建Docker镜像通常有以下两种方式。

- (1) 在开发人员电脑上手工构建，然后推送到仓库中。
- (2) 使用CI/CD系统在代码提交时自动构建。

理想的Docker生产环境将使用类似Jenkins或Codeship这样的CI/CD（配置集成/持续部署）系统，在代码提交时自动构建镜像。一旦容器构建完毕，它将被发送到镜像仓库中，自动化测试系统就可以从中下载并运行该镜像。

2.2 镜像仓库

- Docker镜像保存在哪里？

当前的Docker镜像仓库可靠性比较差，但是每个月都在改善。Docker官方的镜像仓库中心是众所周知的不可靠，需要额外的重试和故障保护措施。多数团队一般会在自己的基础设施上运行私有的镜像仓库，以减少网络传输成本和延迟。

2.3 宿主机管理

- 如何配备宿主机？
- 如何升级宿主机？

由于Docker镜像包含了应用及其依赖，宿主机管理系统通常只需要添加

新服务器，配置访问权限和防火墙，并安装Docker守护进程即可。

类似亚马逊的[EC2 Container Service](#)这类服务将消除对传统宿主机管理的依赖。

2.4 配置管理

- 如何定义容器的集群？
- 如何处理宿主机和容器运行时的配置？
- 如何管理密钥和机密信息？

一个基本规则是：尽量避免使用传统的配置管理系统。其增加的复杂性往往会造成故障。[Ansible](#)、[SaltStack](#)、[Chef](#)或[Puppet](#)这类工具仅用于配备带有Docker守护进程的宿主机。尽可能试着摆脱对旧的配置管理系统的依赖，并使用本书所述的发现和集群技术转移到自我配置的容器上。

2.5 部署

- 如何将容器放置在宿主机上？

镜像部署有以下两种基本方法。

- (1) 推送 —— 部署或编排系统将镜像推送给相关宿主机。
- (2) 拉取 —— 事先或按需从镜像仓库拉取镜像。

2.6 编排

- 如何将容器组织成集群？
- 在哪些服务器上运行容器？
- 如何调度服务器资源？

- 如何运行容器？
- 如何将流量路由给容器？
- 如何让容器公开和发现服务？

“编排 = 强力胶带”。至少多数情况下可以这么认为。

市面上有很多处于早期阶段的全功能容器编排系统，如[Docker Swarm](#)、[Kubernetes](#)、[Mesos](#)和[Flynn](#)。但对大多数团队而言，这些系统通常过于强大，增加了在生产环境中出现问题时调试的复杂度。决定使用哪个工具来完成编排常常是设置和运行Docker中最艰难的部分。

在第3章中，我们将讲述Peerspace所采取的一种构建Docker系统的简约方法。

第3章 示例：极简环境

一说起生产环境中容器的使用，大家的第一反应是那些在同样量级的宿主主机上部署成千上万容器的大型公司。但实际上恰恰相反，要发挥容器的作用，并不需要构建如此庞大的系统。小规模团队反而能从容器中获得最大收益，因为容器使构建和部署服务不仅变得简单，而且可重复、可扩展。

本章描述的就是一家名为PeerSpace的小规模公司构建系统时采取的一种极简方式。这种极简方式使他们能在短时间内使用有限的资源开辟一个新市场，并自始至终保持着极高的开发速度。

PeerSpace构建系统时的目标是既要易于开发，又要在生产环境中足够稳定。这两个目标通常是相互矛盾的，因为高速开发引起的大量变化反过来会对系统的构建和配置产生很大影响。任何一个有经验的系统管理员都知道，这样的变化率必然导致不稳定性。

Docker看起来非常适合用在刚起步的时候，因为它既对开发人员友好，又支持以敏捷的方式构建和运维系统。Docker简化了开发和系统配置的某些方面，但有时却过于简单化了。在易于开发和稳健运维之间取得平衡不是件容易的事。

3.1 保持各部分的简单

PeerSpace实现开发速度和稳定的生产环境这两个目标的方法之一是拥抱简单。这里所说的简单是指系统的每个部分——容器——有且只有一个目标。这个目标就是：相同的过程，如日志收集，在任何地方都以相同的方式完成，而各部分连接的方法也是明确、静态地定义在配置文件中的。

在这种简单的系统中，开发人员可以同步地、独立地构建系统的不同部分，并确信构建的容器可组装在一起。另外，在生产环境出现问题时，简单性也让问题的排查与解决变得非常简单。

要长期保持系统的简单，需要大量的思考、折中和坚持，但最终这种简单将物有所值。

PeerSpace的系统由20个零散的微服务组成，其中有部分使用了MongoDB数据库和/或ElasticSearch搜索引擎。该系统设计遵循下列指导原则。

（1）倾向无状态服务。这可能是简化PeerSpace生产环境时最大的决策：大部分服务都是无状态的。除了用于处理当前进行中的请求的临时信息，无状态服务不需要保持任何需要持久化的数据。无状态服务的优势在于可以非常容易地对他们进行销毁、重启、复制及伸缩，所有这一切都无需考虑任何数据处理方面的逻辑。并且，无状态服务更易于编写。

（2）倾向静态配置。所有宿主机和服务的配置都是静态的：一旦给服务器推送一项配置，该配置就会一直生效，直至显式地推送来新配置。与之相对的是那些动态配置的系统，其系统的实际配置是实时生成的，并会根据不同因素（如可用宿主机和即将到达的负载）进行自主修改。尽管动态系统的伸缩性更好，并且具有一些有趣的属性，如在出现某些故障时自动恢复等，但静态配置更易于理解和排错。

（3）倾向静态的网络布局。如果在一台宿主机中找到一项服务，除非新配置被确定并提交，否则总能在该台宿主机中找到该服务。

（4）区别对待无状态和有状态服务。尽管PeerSpace的多数服务是无状态的，他们还是使用MongoDB和ElasticSearch来持久化数据。这两种类型的服务在本质上是不同的，应该区别处理。例如，将一个无状态服务从一台宿主机移动到另一台上非常简单，只需要启动新服务，然后停止旧服务即可。但要对一个数据库进行移动，数据也要跟着移动。移动数据可能会花费很长时间，要求在迁移过程中停止服务，或通过设备方法进行在线迁移。在开发领域，通常将无状态服务比做“牲口”，它们没有名字，很容易被代替和伸缩，而将有状态服务比做“宠物”，它们唯一的、具名的，需要维护，并且难以伸缩。幸运的是，PeerSpace正像一个农场一样，其“牲口”数量要远远多于“宠物”。

以上这些设计原则是简化PeerSpace系统的基础。将有状态服务与无状态服务分离，可以对本质上完全不同的服务进行区别处理（如图3-1所

示），因此可以对每一种情况的处理方式进行优化和尽可能地简化。使用静态配置运行无状态服务使得操作系统的流程变得非常简单：多数情况下流程被简化成文件复制和容器重启，完全不需要考虑其他因素，如对第三方系统的依赖。

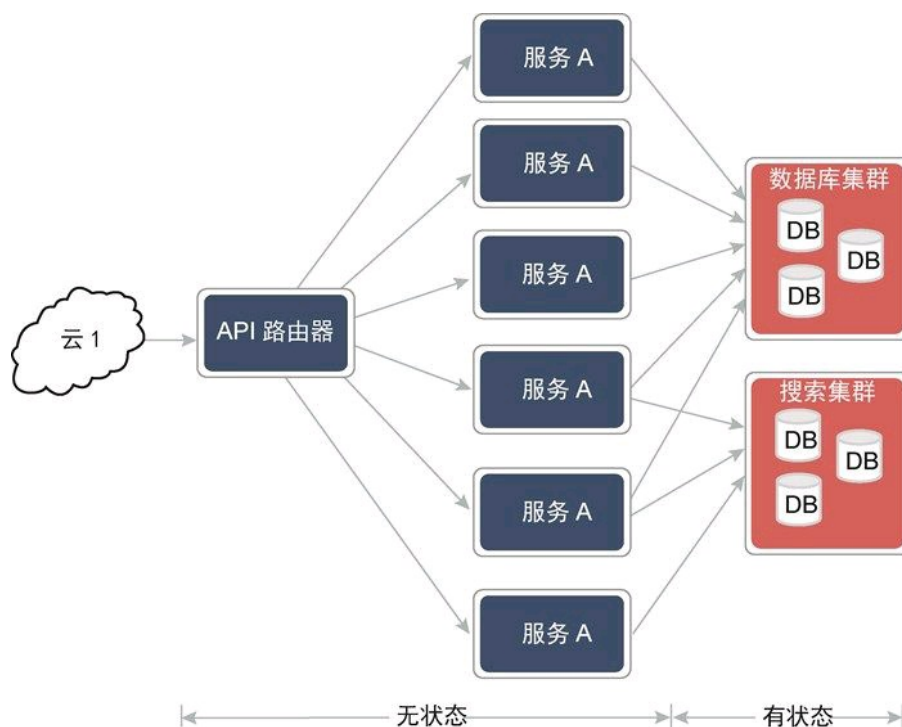


图3-1

上述设计准则能否产生一个简单的系统，完全取决于系统操作是否同样简单。

3.2 保持流程的简单

在设计业务流程时，PeerSpace基于观察做出了如下假定：在他们的基础设施中离硬件越近(layer)变更越少，而越接近终端用户的层变更越频繁（如图3-2所示）。

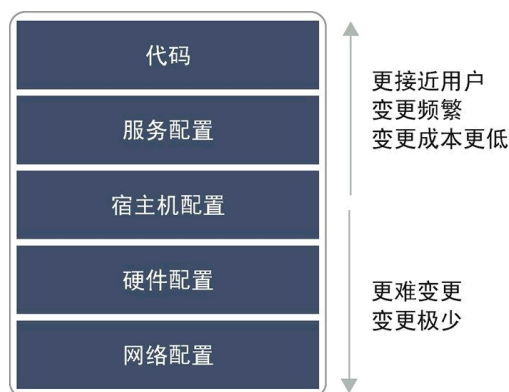


图3-2

根据这一观察，生产环境中的服务器数量很少变更，通常是由于缩放问题或硬件故障。而这些服务器的配置变更频次可能更高一些，通常是由于性能补丁、系统错误修复或安全问题等原因。

在这些服务器上运行的服务数量和类别变更更为频繁。通常是指移动服务、添加新类型服务或对数据进行操作。这个层级上的其他修改可能与要求重新配置或变更第三方服务的新版本部署有关。不过，这类变更仍然不是很常见。

在这样的基础设施中，多数的变更与多个服务的新版本推送有关。每天，PeerSpace都会执行很多次新版服务的部署。多数情况下，新版本的推送只是简单地将现有版本替换成运行新镜像的新版本。有时也会使用相同镜像，但对配置参数进行变更。

PeerSpace的流程建立是为了让最频繁的变更最容易也最简单进行，即便这样会造成基础设施更难以变更（实际上并未发生）。

3.3 系统细节

PeerSpace运行着3个类生产环境集群：集成环境、预演环境与生产环境。每个集群包含了相同数量的服务，并使用相同的方式进行配置，唯一不同的是它们的原始性能（CPU、内存等）。开发人员同样会在自己的电脑上运行全部或部分集群。

每个集群由以下几个部分组成：

- 几台运行着CentOS 7的Docker宿主机，使用systemd作为系统管理程序；
- 一台MongoDB服务器或一个复制集合；
- 一台ElasticSearch服务器或一个集群。

MongoDB和/或ElasticSearch服务器可能在某些环境中是Docker化的，而在其他环境中不是Docker化的（如图3-3所示）。它们也会在多个环境中共享。在生产环境中，出于运维和性能的原因，这些数据服务是不做Docker化的。

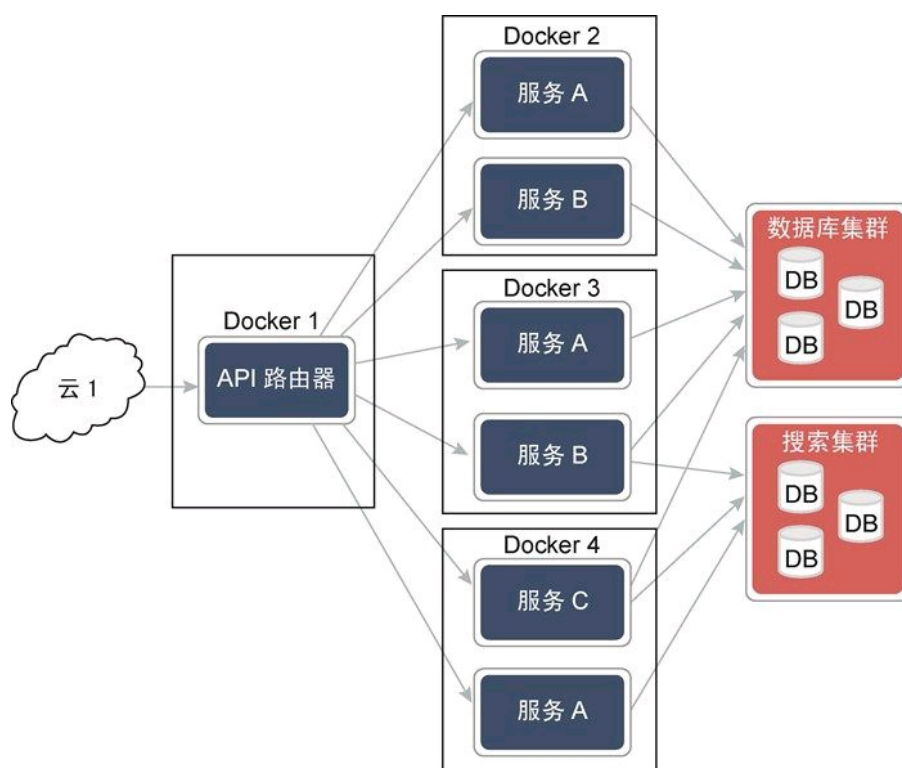


图3-3

每个Docker宿主机运行着一个服务的静态集合，所有这些服务都会遵循如下模式进行构建：

- 所有配置都通过环境变量进行设置，包括其他服务的地址（和端口）；
- 不将数据写入磁盘；
- 将日志发送到标准输出（stdout）中；

- 生命周期由systemd管理，并定义在一个systemd单元文件中。

利用systemd

所有服务都由systemd管理。systemd是一个借鉴了OSX launchd的服务管理程序，此外，systemd使用普通数据文件命名单元来定义每个服务的生命周期（如图3-4所示），这与其他使用shell脚本完成这类事务的传统管理程序完全不同。

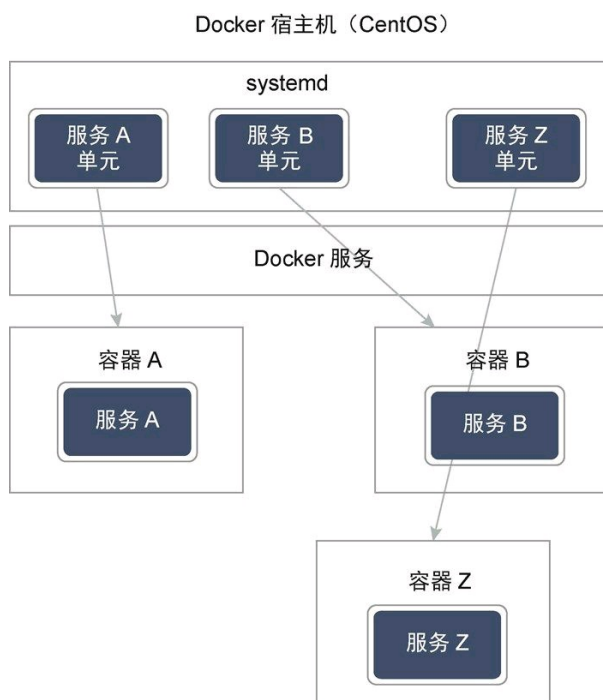


图3-4

PeerSpace的服务只将Docker进程当作唯一的运行时的依赖。systemd的依赖管理只用来确保Docker处于运行状态，但不确保其拥有的服务以正确顺序启动。服务构建时要求它们可以以任何顺序启动。

所有服务都由以下部分组成（如图3-5所示）：

- 一个容器镜像；
- 一个systemd单元文件；
- 一个该容器专用的环境变量文件；
- 一组用于全局配置参数的共享环境变量文件。

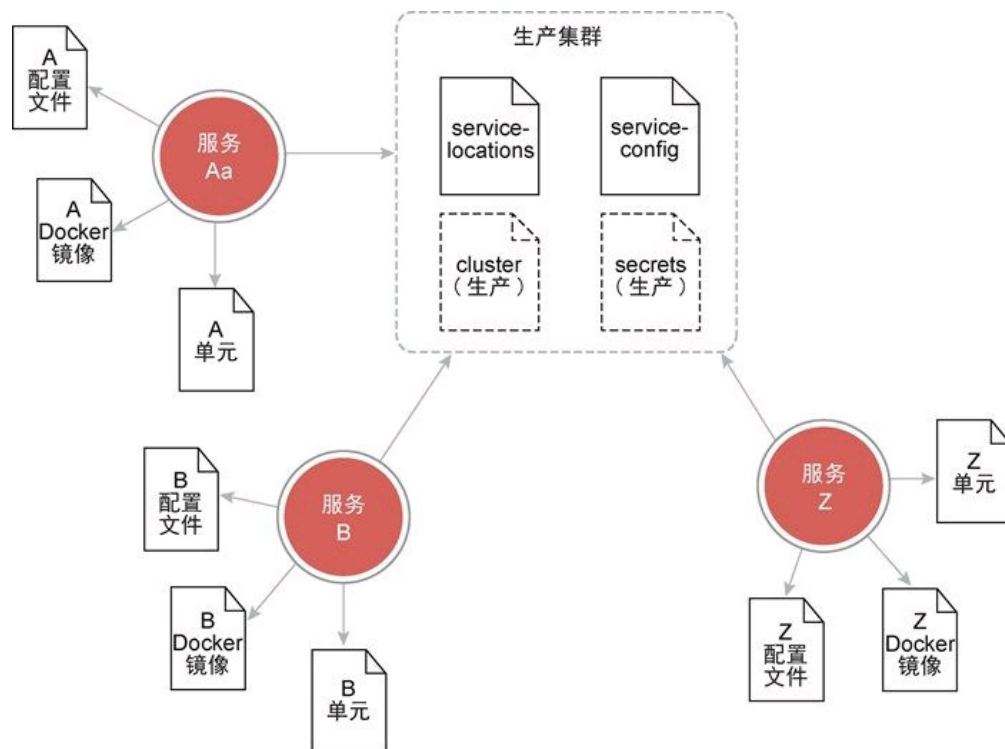


图3-5

所有单元都遵循相同的结构。在服务启动之前，一系列包含环境变量的文件将被加载：

```
EnvironmentFile=/usr/etc/service-locations.env
EnvironmentFile=/usr/etc/service-config.env
EnvironmentFile=/usr/etc/cluster.env
EnvironmentFile=/usr/etc/secrets.env
EnvironmentFile=/usr/etc/%n.env
```

这确保了每个服务会加载一系列通用环境文件（**service-locations.env**、**service-config.env**、**cluster.env**及**secrets.env**），外加一个专用于该服务的文件：**%n.env**，此处的**%n**在运行时将被替换成该单元的全称。例如，一个名为**docker-search**的服务单元将被替换成**docker-search.service**。

接下来的条目是确保在启动新容器前旧容器被正确删除的：

```
ExecStartPre=--/bin/docker kill %n
ExecStartPre=--/bin/docker rm -f %n
```

通过使用`%n`，将容器命名为单元的全称。使用变量进行容器命名能让单元文件更通用并且可移植。在`docker`程序路径之前使用“-”可防止单元在命令失败时中止启动。这里需要忽略潜在的错误，因为如果此前不存在该容器，这些命令将执行失败，而这种情况又是合法的。

单元中主要的条目是`ExecStart`，它将告之`systemd`如何启动该容器。这里内容较多，但我们只关注一下其最重要的部分：

```
ExecStart=/bin/docker \
    run \
    -p "${APP_PORT}:${APP_PORT}" \
    -e "APP_PORT=${APP_PORT}" \
    -e "SERVICE_C_HOST=${SERVICE_C_HOST}" \
    -e "SERVICE_D_HOST=${SERIVCE_D_HOST}" \
    -e "SERVICE_M_HOST=${SERVICE_M_HOST}" \
    --add-host docker01:${DOCKER01_IP} \
    --add-host docker02:${DOCKER02_IP} \
    --volume /usr/local/docker-data/%n/db:/data/data \
    --volume /usr/local/docker-data/%n/logs:/data/logs \
    --name %n \
    ${IMAGE_NAME}:${IMAGE_TAG}
```

(1) 使用`EnvironmentFile`加载的环境变量来配置容器（如通过`-p`公开的端口）。

(2) 将集群中的其他宿主机地址添加到容器的`/etc/hosts`文件中（`--add-host`）。

(3) 映射用于日志和数据的数据卷。这主要是作为一个“蜜罐”（honey pot^[1]），以便检查这些目录并确保无人对其进行写入。

(4) 镜像自身（名称和版本）来自于从`/usr/etc/%n.env`中加载的环境变量，在本示例中它将映射到`/usr/etc/docker-search.service.env`中。

最后，是一些定义如何停止容器及其他生命周期要素的条目：

```
ExecStop=-/bin/docker stop %n
Restart=on-failure
RestartSec=1s
TimeoutStartSec=120
TimeoutStopSec=30
```

3.4 集群范围的配置、通用配置及本地配置

PeerSpace将集群配置分成两种类型文件：环境变量文件和systemd单元文件。上面已经讲述了单元文件及其加载环境变量文件的方式，接下来看一下环境文件。

将环境变量分解到不同文件中的主要原因在于，这些文件在跨集群时是否需要修改以及如何修改，不过也有其他操作层面的原因。

- **service-locations.env**: 集群中所有服务的宿主机名。这个文件在不同集群里通常是一样，不过也有例外。
- **service-config.env**: 与服务自身相关的配置。如果不同集群运行的是服务的兼容性版本，这个文件应该是一样的。
- **secrets.env**: 密钥信息。因其内容关系，这个文件被处理的方法与其他文件不同，而且在不同集群上也有差异。
- **cluster.env**: 包括了集群间的所有不同之处，如所使用的数据库前缀、是测试还是生产环境、外部地址等。这个文件中最重要的信息是属于该集群的所有宿主机的IP地址。

下面是某些示例集群中的文件。这是**cluster.env**文件：

```
CLUSTER_ID=alpha
CLUSTER_TYPE="test"
DOCKER01_IP=x.x.x.226
DOCKER02_IP=x.x.x.144
EXTERNAL_ADDRESS=https://somethingorother.com
LOG_STORE_HOST=x.x.x.201
LOG_STORE_PORT=9200
MONGODB_PREFIX=alpha
MONGODB_HOST_01=x.x.x.177
MONGODB_HOST_02=x.x.x.299
MONGODB_REPLICA_SET_ID=rs001
```

这是**service-locations.env**文件：

```
SERVICE_A_HOST=docker01
SERVICE_B_HOST=docker03
CLIENTLOG_HOST=docker02
SERIVCE_D_HOST=docker01
...
SERVICE_Y_HOST=docker03
SERVICE_Z_HOST=docker01
```

每个systemd单元都包含集群中其他宿主机的引用，而这些引用来自于环境变量。包含服务宿主机名的变量会被装配到Docker命令中，以便容器进程使用。这是通过-e参数实现的，如-e "SERVICE_D_HOST=\${SERIVCE_D_HOST}"。

Docker宿主机的IP地址也同样通过--add-host docker01:\${DOCKER01_IP}注入到容器中。这样，只需要修改这两个文件并且保持单元文件的完好无损，就可以将容器扩散到不同数量的宿主机中。

3.5 部署服务

容器级别或配置级别的修改通过3个步骤完成：第1步，在配置仓库（Git）上做修改；第2步，将配置文件复制到宿主机的预演区域（ssh）；第3步，运行宿主机上的一个脚本来逐一部署每个服务，使得配置修改生效。这种方法提供了版本化配置，一次只推送一项相关配置，以及让推送配置生效的一种灵活方式。

如果需要针对一组服务进行修改，首先在Git上做修改并提交。然后运行脚本，将这个配置推送到所有宿主机的预演区域。一旦配置被推送过去，在每台宿主机上运行一个脚本来部署或重部署该宿主机的所有容器集合。这个脚本会对在列的所有服务执行如下命令。

（1）将配置文件从预演区域复制到其最终位置：

- systemd单元文件；
- 共享的配置文件；
- 当前服务的配置文件；

- 密钥文件（解密后的）。

- (2) 需要的话下载镜像文件（镜像定义在服务自身的配置文件中）。
- (3) 重载systemd的配置，以便读取新的单元文件。
- (4) 重启容器对应的systemd单元。

PeerSpace具有两个部署 workflow，理解这一点有助于阐述其部署流程：一个用于开发环境，另一个用于生产环境，而后者是前者的一个超集。

在开发过程中，他们会通过以下步骤将临时构建部署到集成服务器中。

- (1) 使用最新代码库创建一个新的容器镜像。
- (2) 将镜像推送到镜像仓库中。
- (3) 在运行该镜像的容器宿主机上运行部署脚本。

开发环境的systemd单元会追踪镜像的最新版本，所以只要配置不做修改，那我们只需推送镜像并重新部署即可。

类生产环境的服务器（生产环境和预演环境）与开发环境配置方式大体相同，主要区别在于生产环境中的容器镜像都打上了版本标签，而非**latest**。部署发布镜像到类生产环境容器的流程如下。

- (1) 在仓库中为容器镜像运行发布脚本。该脚本将为Git仓库打上新版本标签，然后使用这个版本号构建并推送镜像。
- (2) 更新每个服务环境变量文件以引用新镜像标签。
- (3) 将新的配置推送到各宿主机中。
- (4) 在运行该镜像的容器宿主机上运行部署脚本。

他们通常会批次地将服务从开发环境转移到生产环境（一般是两周一次）。在推送发行版到生产环境时，开发环境中用于该发行版的配置文件会被复制到生产目录中。多数文件可以完全照搬，因为它们是从集群的具体细节（IP地址、宿主机数量等）抽象出来的，不过**cluster.env**和**secrets.env**文件在各个集群中是不一样的，在发行时也对其进行更新。一般情况下，会一次性推送所有新版本服务。

3.6 支撑服务

PeerSpace使用了一组服务来支撑自己的服务。这些服务包括以下两个。

- 日志聚合：fluentd+kibana以及docker-gen的组合。docker-gen可根据宿主机中运行的容器创建和重创建一个配置文件。docker-gen为每个运行中的容器生成一个fluentd条目，用于发送日志给kibana。这个服务运行良好，且易于调试。
- 监控：Datadog——一个SaaS监控服务。Datadog代理在容器中运行，用于监控各项性能指标、API使用情况和业务事件。Datadog为标签提供了丰富的支持，通过fluentd可以使用多种方式对单一事件进行标记。数据收集起来后（如跨集群的相同服务、所有Docker服务、使用某个发行版的所有API端点等），可以利用丰富的标签对数据进行多种方式的切割。

3.7 讨论

在系统中，所有宿主机和服务的配置都非常明确，开发人员很容易理解系统的配置，并能不受干扰地工作于系统的不同部分上。每位开发人员都可以在任何时候对集成集群进行推送，并且推送到生产环境所需的协调也很少。

由于每个集群的配置都保存在Git上，很容易追踪配置的变化，并在出现配置问题时对集群进行排错。

因为配置推送的方式，一旦新配置设置妥当，该配置将保持不变。静态配置带来的是极大的稳定性。

另外，服务编写的方式，如通过环境变量进行配置、日志写入控制台、无状态等，使得它们之后可原封不动地被Mesos或Kubernetes这类集群管理工具使用。

当然，要得到这些好处是有代价的。一个最明显的缺点是配置有些繁琐、重复并且易出错。我们可以通过大量的自动化的工具来生成这些配置文件。

修改全局配置要求重启多个容器。目前是由开发人员来重启正确的容

器。在生产环境中，如果推送的修改很多，通常会执行滚动重启，但这并不是一个很好的解决方法。这绝对是一个薄弱环节，但到目前为止，还是可控的。

3.8 未来

PeerSpace正在考虑几个系统扩展的方式。其中之一是通过反向代理实现零停机时间部署。这将使得PeerSpace有能力对每个服务进行水平扩展。

另外一个方向是从集群的更高层级描述中生成所有的配置文件。这种方法能在配置发生改变后计算哪些容器需要重启。

在考虑这些未来的方向时，PeerSpace也在权衡使用Mesos或Kubernetes的可能性，因为他们认为，增加部署脚本的任何复杂度势必造成对简单模式的过度拉伸。

3.9 小结

尽管本章讲解了一个极其简单的Docker使用方式，但我们仍希望它能成为“Docker思想”的基石。不论是使用极简方式还是集群管理系统，读者都能利用这种方式在阅读本书其他部分时获益。

当然，使用Docker还有很多其他方式，第4章将讲述RelateIQ使用Docker运行了一年多的一个真实的Web服务器生产环境。

[1] 用于隐藏宿主机的真实路径。——译者注

第4章 示例：Web环境

我们所知的大多数公司都曾以一个很低的容器和宿主机比例（1~2个容器对应1台宿主机）成功地使用过Docker。也就是说，要在生产环境中成功运行Docker，并不是必须要运行Apache Mesos或Kubernetes。在本示例中，将对RelateIQ公司^[1]使用Docker运行了一年多的一个真实Web服务器生产环境做详细的说明。这个环境在运行Ubuntu的标准亚马逊云服务（AWS）实例上，使用Docker支撑其CRM Web应用。当初使用Docker的原因有三：一是Docker能快速生成和销毁容器，从而为客户提供零停机时间部署；二是因为Docker为不同Web版本提供依赖隔离；三是Docker支持即时回滚。图4-1所示为该环境的高层次示图。

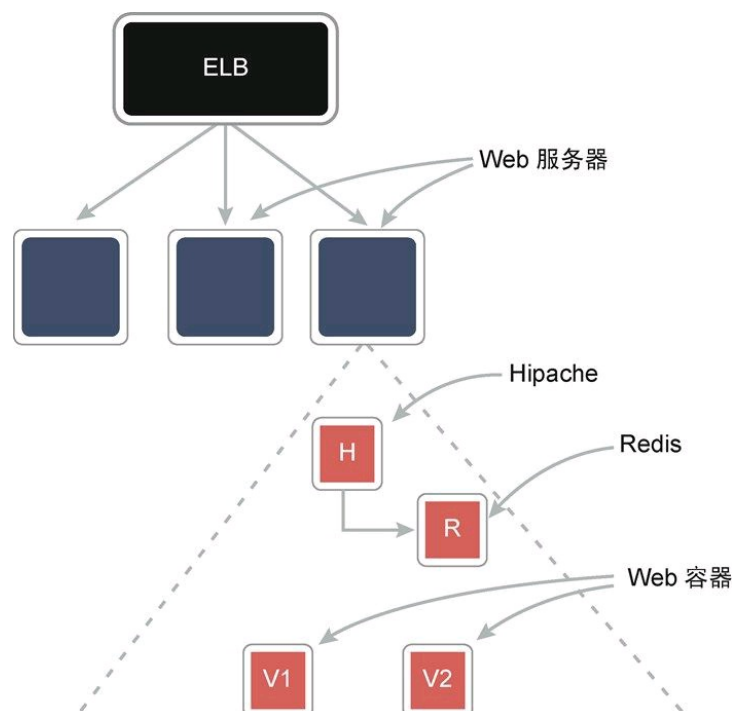


图4-1

相信吗？这个Web环境提供了如下功能：稳定的零停机时间部署、回

滚、集中式日志、监控及分析JVM的一种方式。所有这些都是通过bash脚本编排Docker镜像获得的。图4-2所示为主机的详细情况。

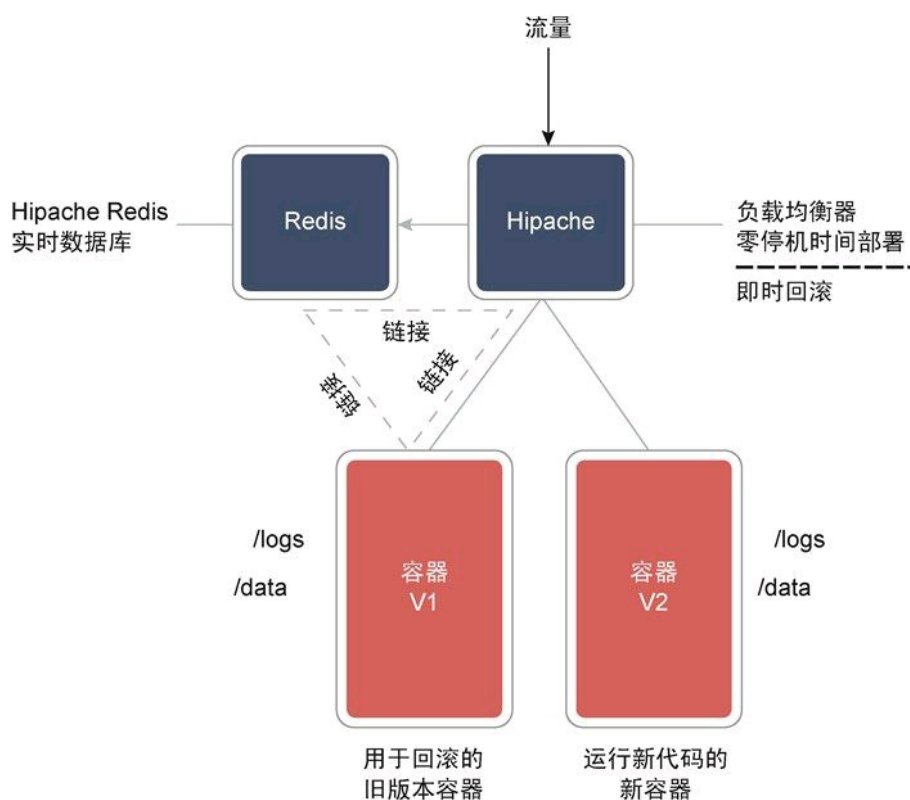


图4-2

这台Web服务器运行于单台AWS服务器上，并通过Docker运行着4个容器。部分容器被链接在一起，以便与Docker网桥上的其他容器进行通信。它给宿主机公开了多个端口，用于为性能分析提供HTTP服务和JVM监控。它使用了亚马逊ELB负载均衡器（健康检查在其上进行）。所有容器都将它们的日志保存在宿主机上，这样现有的日志方案（SumoLogic）依旧适用，同时有一个简单的bash编排脚本用于部署和设置新版本Web服务。

为了便于理解很多公司在生产环境中运行Docker时会遇到的问题，我们来看一些具体细节。

4.1 编排

编排归根到底就是做两件事：一是获取已安装Docker的服务器，并且使之准备好运行容器的服务器；二是在服务器上启动并运行容器。

4.1.1 让服务器上的Docker进入准备运行容器的状态

该服务器使用标准的基本Ubuntu AMI（亚马逊机器镜像）在AWS上部署，并通过Chef的标准配置管理系统对宿主机进行设置。其设置过程与当下的多数环境完全相同。服务器启动之后，Chef就会运行并设置ssh用户、ssh密钥，然后通过其包安装器安装基础包（如iostat），安装并配置监控代理（本例中是Datadog），集合一些临时磁盘空间用于数据或日志存储，安装并配置日志代理（SumoLogic），安装最新版Docker，最后创建bash设置脚本，并配置一个cron任务来运行它。

Chef在服务器上运行之后，宿主机就准备好在其上运行机器所需的任何容器了。Chef还配置了监控和日志软件，用于未来的调试。这个环境可以运行任何类型的容器服务，与当下运行的大多数服务器环境，甚至是物理环境也一般无二。现在，Docker已经安装完毕，宿主机也准备好核心操作工具，下面就可以让宿主机上的容器开始运行Web应用了。

4.1.2 让容器运行

早期运行Docker的大多数公司一般都是使用bash脚本来设置容器的，这个环境也不例外。这个环境使用一个cron任务，每5分钟运行一个bash脚本来进行容器的所有编排工作。脚本的核心功能是正确地设置容器并拉取最新版的Web服务器镜像。我们来深入看一下所使用的脚本片段。

这个脚本完成以下操作。

- （1）检查容器是否正在运行（通常是的，这主要用于新机器的情况）。
- （2）如果容器未运行，则部署Hipache和Redis容器并将它们链接在一起。
- （3）拉取最新版的Web服务器镜像并运行。
- （4）等待Web服务器健康检查通过，然后再将其添加到负载均衡器中。
- （5）一旦上述操作成功，给服务器上的迷你负载均衡器hipache发送一条消息（本例中是使用netcat运行一个redis-cli命令），告知

Docker为之分配的随机端口和IP地址。

(6) 保持旧容器运行，以便在需要进行回滚。

(7) 清除旧镜像。

下面是脚本的一些片断（为适合阅读，删除了部分代码）：

```
#!/bin/bash

# 检查Hipache容器
STATE=$(docker inspect hipache | jq ".[0].State.Running")
if [[ "$STATE" != "true" ]]; then
    set +e
    docker rm hipache >/dev/null 2>&1
    set -e
    mkdir -p /logs/hipache/
    docker run -p 80:80 -p 6379:6379 --name hipache -v /logs/hipache:/logs
    -d repo.com/hipache
    echo "$(date +"%Y-%m-%d %H:%M:%S %Z") lpush frontend:* default"
    sleep 5
    (echo -en "lpush frontend:* default\r\n"; sleep 1) | nc localhost 6379
fi

# 拉取最新镜像
IMAGE_ID=$(docker images | grep ${IMAGE_NAME} | grep $REMOTE_VERSION |
head -n 1 | awk '{print $3}')
if [ -z $IMAGE_ID ]; then
    docker pull $DOCKER_IMAGE_NAME
fi

echo $REMOTE_VERSION >$VERSION_FILE

# 启动新容器
echo "$(date +"%Y-%m-%d %H:%M:%S %Z") launching $DOCKER_IMAGE_NAME,
logging to $LOG_DIR"
mkdir -p $LOG_DIR
NEW_WEBAPP_ID="abcdefghijklmnopqrstuvwxyz"
MAX_TIMEOUT=5
set +e
until [ $MAX_TIMEOUT -le 0 ] || NEW_WEBAPP_ID=$(docker run -P -h
$(hostname) --link hipache:hipache $(dockerParameters $BRANCH) -d -v
$LOG_DIR:/logs $DOCKER_IMAGE_NAME); do
    echo -n "."
    sleep 1
    let MAX_TIMEOUT-=1
done
set -e
```

```

# 检查Web应用容器是否已启动
NEW_WEBAPP_IP_ADDR=$(docker inspect $NEW_WEBAPP_ID | jq '[0].NetworkSettings.IPAddress' -r)
if [ -z "$NEW_WEBAPP_IP_ADDR" -o "$NEW_WEBAPP_IP_ADDR" = "null" ]; then
    echo "$(date +%Y-%m-%d %H:%M:%S %Z)" no new webapp ip, failed to start"
    # send_deploy_message $HOSTNAME $BRANCH $IMAGE_NAME "error"
    send_webhook $HOSTNAME $BRANCH $BUILD_ID $BUILD_NUMBER "failure"
    exit 1
fi

echo -n "$(date +%Y-%m-%d %H:%M:%S %Z)" new instance $NEW_WEBAPP_ID starting, on ip $NEW_WEBAPP_IP_ADDR"
# 5分钟
MAX_TIMEOUT=300
HEALTH_RC=1
set +e
until [ $HEALTH_RC == 0 ]; do
    if [ $MAX_TIMEOUT -le 0 ]; then
        echo "$(date +%Y-%m-%d %H:%M:%S %Z)" failed to be healthy within 5 minutes, killing and exiting..."
        docker kill $NEW_WEBAPP_ID
        docker rm $NEW_WEBAPP_ID
        # send_deploy_message $HOSTNAME $BRANCH $IMAGE_NAME "error"
        send_webhook $HOSTNAME $BRANCH $BUILD_ID $BUILD_NUMBER "failure"
        exit 1
    fi

    ${SCRIPT_HOME}/health.sh $NEW_WEBAPP_IP_ADDR
    HEALTH_RC=$?
    echo -n "."
    sleep 5
    let MAX_TIMEOUT-=5
done
set -e
echo

# 将自身作为后端添加到Redis中
(echo -en "rpush frontend:* http://${NEW_WEBAPP_IP_ADDR}:${WEBAPP_PORT}\r\n"; sleep 1) | nc localhost 6379
# 确保自己是Redis的第一个后端
(echo -en "lset frontend:* 1 http://${NEW_WEBAPP_IP_ADDR}:${WEBAPP_PORT}\r\n"; sleep 1) | nc localhost 6379
# 将Redis中所有其他后端移除
(echo -en "ltrim frontend:* 0 1\r\n"; sleep 1) | nc localhost 6379

```

如我们所见，这段脚本大部分都是一些很基础的**bash**指令。只要有一些**bash**脚本的经验，任何系统管理员或运维工程师都能完成此类编排。容器的编排可以很简单，但必须经过几次迭代，过一段时间脚本就会变得更强壮。即便是在出现失败的情况下，这个脚本也能正确工作，不会将未通过健康检查的新容器上线。随着与**Docker**相关的新技术不断出现，类似**Apache Mesos**和**Kubernetes**这样的系统将取代**bash**脚本来完成编排。下面来看这个环境在其他方面是如何工作的。

4.2 连网

只要掌握窍门，运行**Docker**和单一容器的宿主机网络就很容易理解。**Docker**通过**docker run**命令将容器的端口公开给宿主机。服务器公开的端口包括负载均衡器监听的80端口（**ssl**只传递到负载均衡器）、用于**Java**优化的优化端口、**Redis**用于切换负载均衡器后端的端口，以及**Web**服务器自身的一个端口（后续章节详述）。服务器之外的负载均衡器只监控80端口。宿主机中的**Web**服务器会启动一个随机端口，来自80端口的请求会被**Hipache**代理转发到这个端口上。

4.3 数据存储

由于这是一个**Web**服务，存储的需求不会太多。有时需要存储日志、文件的缓存或加载静态内容。本例中，使用的是宿主机的而非容器的存储。将数据保存在宿主机上的理由很简单。如果容器宕机了，我们仍然需要排查出现的问题。服务一般是将日志写入到某个文件路径中。本例中我们将**Docker**容器映射到宿主机文件系统中，并将持久化的日志文件从容器里重定向到宿主机上，以便未来进行日志分析。这通过**docker run**的**-v**卷参数很容易实现。

4.4 日志

容器日志根据服务进行分类。例如，使用 `/logs/Redis`、`/logs/hipache` 和 `/logs/webserver/`（如图4-3所示）。这里需要特别注意，Web服务器会根据请求的日期时间戳来记录错误和请求日志。容器记录日志时，其文件名类似这样：`/logs/webserver/2015-03-01.request.log`。如果文件存在，日志会自动追加到同一文件中。如果有另外一个或多个容器启动，日志同样会被追加到同一个文件中。通过Chef安装logrotate，可防止日志无限制地增长。

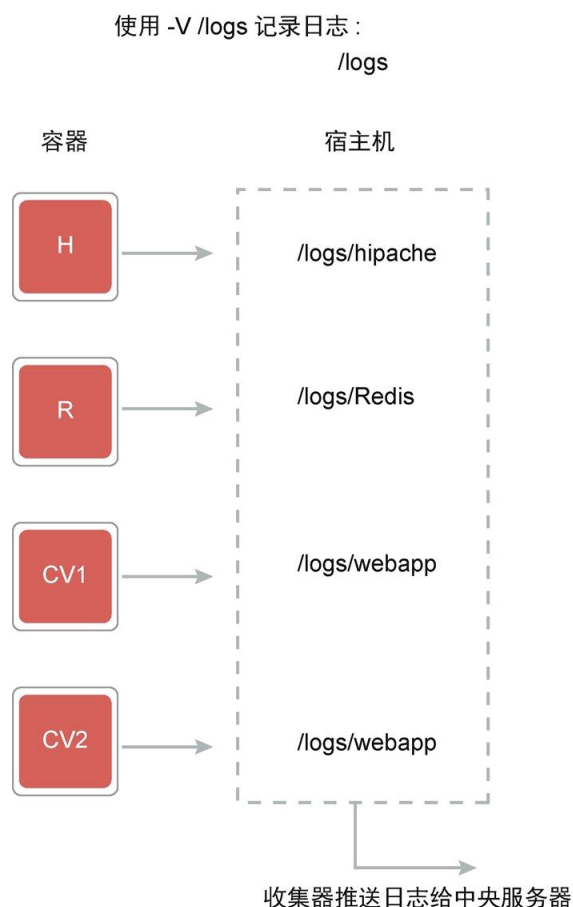


图4-3

在生产环境中，通常会有一个集中式日志服务器，因此服务器上的日志只是在被收集器取走前做临时保存。由于所有的容器都将日志写入宿主机中，所以无须为Docker采用一项全新的日志技术。非运行Docker的环境极可能也会采用相同的日志操作方式，以保持现存监控框架的不变。在这个环境中，很容易将创建或追加的日志文件发送到中央日志服务器（Splunk、Sumologic和Loggly）上以便分析。

4.5 监控

这里需要特别注意的是，负载均衡器监控服务器的负荷，并根据需要自动将下一个请求发送给其他可用Web服务。宿主机是通过带有Docker插件（这里是Datadog）的宿主机上的监控代理来监控的。本示例中的监控是一个全栈监视器。这个代理监控着宿主机的使用情况，如CPU、内存、磁盘IO、JVM监控以及运行的容器数量。这个环境里的应用程序指标通过StatsD发送给中央收集器。这里的指标包括了：网页点击量、应用程序查询速度以及特定功能的延迟指标。

在这个环境中，使用了一个名为Yourkit的JVM优化工具来监控堆的运行情况。这样，运维团队或开发人员可以将自己的优化工具连接到宿主机上，从而通过调用栈发现应用程序的深层问题。其缺点之一是，每个容器都需要有单独的端口，如果宿主机上同时运行着两个容器，它们的端口也不能一样。所以需要通过一个快速的SSH或工具来检查这个端口。类似New Relic和Sysdig（在生态系统中提到过）的这类新技术可以对其进行监控。

4.6 无须担心新依赖

由于所有的应用程序依赖都存储在容器镜像中，运维团队只需要管理服务器管理方面的依赖即可。这简化了Chef配置管理框架以及用于保持环境更新的脚本数量。

4.7 零停机时间

这个Web服务环境可以提供零停机时间部署。零停机时间部署通过Hipache以及一个由Redis支撑的实时Web查询引擎实现，由于Redis是单线程的，在此作为数据库非常完美。Hipache会将HTTP会话重定向给数据库列表中最顶部的服务器。新容器上线时，将发送一条更新列表的命令。

令给Redis，然后这个新容器就能接收所有新的点击。会话状态保存于后台数据库中，因此容器可以短暂存活，并且不会造成客户端状态的丢失。

4.8 服务回滚

因为Docker镜像存储于服务器上且容器启动速度非常快，这个环境可以非常容易地在需要时回滚旧代码。由于宿主机上保存着多个容器，很容易使用类似脚本或其他编排工具来启动旧容器并取代新（错误部署的）容器。

4.9 小结

RelateIQ已经在生产环境中运行本章所描述的设置一年多了，取得了巨大的成功。他们的团队将标准的运维工具应用到Docker中，创造出一个功能完整的Web编排层。使得他们无须进行重大的基础性变更即可尝试新技术。他们也能将Docker和当前的基础设施监控及日志方案相结合，使其易于在生产环境中运行。对这个环境有兴趣的读者，可以阅读有关该环境的[博客文章](#)和[访谈](#)。

在第5章中，我们将讲述RelateIQ如何使用AWS Beanstalk通过Docker为每个分支完整编排一个Web环境。

[1] 现已被Salesforce收购并更名为SalesforceIQ。——译者注

第5章 示例：Beanstalk环境

目前，大多数软件公司内部都有多套基础设施环境。这些环境具有典型的3层结构：测试、预演和生产环境。部分公司在此之外还有预生产甚至是公测（canary）环境，不过这些都是特例。不同的环境为新代码甚至是基础设施组件的生命周期提供了隔离。这些环境通常由至少一个用于应用程序逻辑和展示的Web服务器层和一个数据库层组成。过去的10多年里，这些环境在不同公司内部已经相当稳定。我们在RelateIQ发现了另外一个极佳的Docker环境示例。他们正在做的事情很有趣，有可能打破这种标准的环境模式。

RelateIQ使用AWS Beanstalk为每个分支完整地编排出一个Web环境。这是通过其CI/CD基础设施使用Docker技术创造出来的一个全新的基础设施环境。他们从根本上把Web层与数据层分开。将典型的3层模型转换成仅剩数据存储。这一点一开始可能有点儿难以想象，我们来直观地看一下（如图5-1所示）。

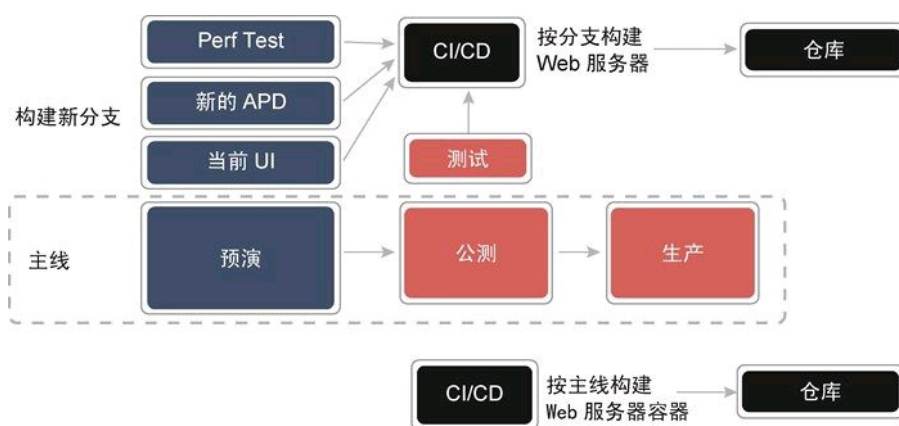


图5-1

图5-1展示了3个不同的分支，即PerfTest、新的APD及新的UI分支。每个分支通过CI/CD系统来构建自己的Web容器。容器被构建和测试后，将被推送到仓库中。任何开发人员或团队，都能根据需要构建各个分支

的Web环境。这还允许用户以不同的方式考虑如何使用容器，同时重新考虑环境的各个部分。SaaS公司采用这种模式的巨大优势之一是适应快速变化的能力。试想一下，如果将这类环境用于持续交付模型中，产品经理和设计师会如何利用这些环境。

举个例子，RelateIQ在2014年夏季通过Docker使用这种新模式，用一个全新的外观取代CSS，完全重新设计了整个Web应用程序。他们可以在并排的Web服务器中运行A/B测试，以对新旧版本进行比较。由于开发人员会在15~30分钟内提交代码，设计师和产品经理能够在一个隔离的环境中看到最近生效的变化。使用环境变量，还能够将Web服务器后端从预演数据服务器重定向到生产数据服务器上。Docker容器快速重启后，他们就能在几分钟内看到生产数据的最近生效的变化。当RelateIQ准备上线新设计的网站时，他们只需将容器从预演数据切换到生产数据。这样，开发人员就能确保数据与新UI正确匹配。

5.1 构建容器的过程

在这个环境中，RelateIQ使用Teamcity构建和部署应用程序。他们使用VCS触发器来监控GitHub仓库中特定的分支名称，以执行自动构建。例如，他们使用了“**docker-<分支>**”这样的名称。如果仓库中创建了一个以“**docker-**”开头的分支，那么这个分支自身的Docker容器将会自动被构建。为了快速启动，大多数开发人员会在预演时直接创建分支。分支一旦创建，Teamcity就会构建这个容器并推送到一个本地仓库中。他们使用亚马逊AWS的Beanstalk服务来部署和更新容器。

部署/更新容器的过程

“AWS Elastic Beanstalk 是一项易于使用的服务，用于在熟悉的服务器（如 Apache、Nginx、Passenger 和 IIS ）上部署和扩展使用 Java、.NET、PHP、Node.js、Python、Ruby、GO 和 Docker 开发的 Web 应用程序和服务。

您只需上传代码，Elastic Beanstalk 即可自动处理从容量预置、负载均衡、自动扩展到应用程序健康监控的部署。同时，您能够完全控制为应用程序提供支持的 AWS 资源，并可随时访问基础资源。”

Beanstalk将自动部署一台负载均衡器，设置自动扩展组，根据设置配备相应数量的实例/服务器，拉取并运行Docker容器，提供健康监控，并且使用安全组加固服务器。对刚刚起步的公司而言，这将使其在基础设施方面走得很远。如果将其与“一个Web服务一个容器”组合起来，对SaaS公司来说，这将变成一个非常有用的环境。

部署的执行有多种方式，从Elastic Beanstalk到使用S3存储桶（bucket）的JSON文件或对服务自身的API调用。在本示例中，使用的是在Teamcity配置中创建的S3存储桶的JSON文件。构建步骤之一将写新文件，并将变化推送给S3存储桶。文件包含了容器的位置、需要打开的端口，以及容器的名称。新文件被上传后，Elastic Beanstalk环境将自动启动一台新服务器，在这台服务器上拉取容器并进行设置，然后在健康检查通过后停止旧的容器和服务（如图5-2所示），基本上创造了新服务的零停机时间部署。

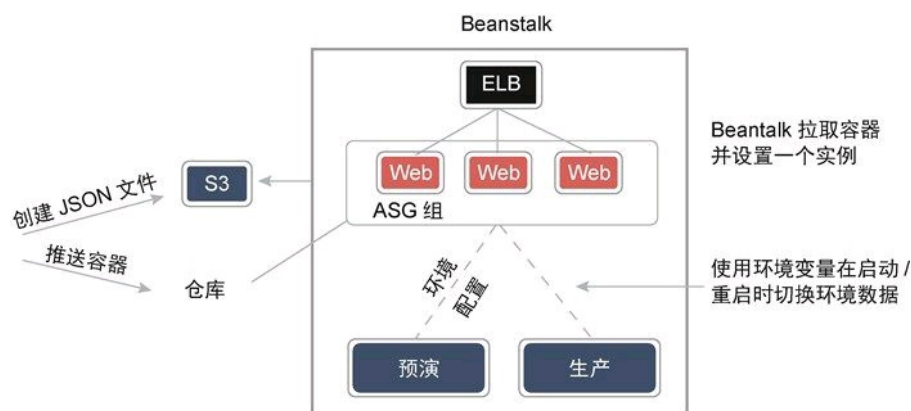


图5-2

Elastic Beanstalk的使用表明，基础设施供应商真正开始运行自有容器只是时间问题，就像他们之前运行虚拟化基础设施一样。

5.2 日志

Elastic Beanstalk容器的日志是全自动化的，就像该基础设施的其他部分一样。可以通过GUI工具拉取日志，并选择拉取的是服务器上的完整日

志还是标准输出的最后100行。这两个选项只在排错时才有用，因此他们提供了一个新选项用于将日志尾部发送到S3存储桶中。使用能读取S3服务日志的集中式日志服务让公司能将这些日志直接集成到现有日志方案中。围绕日志部分有几个注意事项。日志的名称与容器名称无关。其名称是根据一个作为服务唯一标识而随机生成的服务名称所创建的。要追踪哪个唯一标识属于哪个Elastic Beanstalk将非常麻烦。

5.3 监控

所有通过亚马逊AWS提供的服务都内置了各自的云监控方案，也支持Elastic Beanstalk。不过，监控非常基础。能获取ELB指标和服务器指标，但无法从容器本身获取任何东西。这是由于Elastic Beanstalk服务中容器与服务器比例为1:1。这意味着网络、CPU、磁盘以及内存指标通常来自于运行于宿主机上的容器指标。如果机器出现异常，只能SSH登录到服务中进行排错或部署新版本。

5.4 安全

Beanstalk通过安全组和IAM角色提供了自动化防火墙端口安全，以加固用户访问。由于Beanstalk使用一个很低的容器宿主机比例，就像普通的应用程序和服务器环境一样，要将容器与其他容器隔离开非常容易。Beanstalk模板保证了跨新部署时新环境的一致性，可以轻松进行跨多台宿主机的安全性修改。

5.5 小结

通过使用Docker以及亚马逊AWS的Elastic Beanstalk自动化基础设施，RelateIQ能够在一个可扩展的模板环境中为每一位前端工程师提供一个Web环境。这个环境非常容易创建，加上CI/CD系统的一点编排，它是完全自动化的。注意：如果想在自己的基础设施中做尝试，这个环境还

有很多部分处于构建状态。日志可以做得更好，因为Beanstalk环境提供的唯一标识非常难用，截至本书编写时，每个服务上已经运行一个容器（很快就能支持多容器）。请记住，使用Docker可以让环境变得异常灵活，并为推动开发速度提供新的创新方法。

在第6章中，我们将深入讲述Docker的安全性话题。

第6章 安全

安全一直都是个困难重重的领域。优秀的安全专家会在追求完美和生产上线之间寻求恰当的平衡，并促进开发团队考虑安全问题。

Docker在安全方面一直备受关注，因其“实用为先，安全在后”的模式总是会招致非议。此外，它并不具备单一的强安全模式及响应，而是依赖于多个层次，其中一些层次可能尚不存在或无法用于应用程序中。官方的[安全文档](#)也非常缺乏，因此用户需要更多的支持。

这反映出了Linux的安全现状，在市面上有大量选择。尽管这些选择不全是容器专用的，但是容器添加了更多工具，以至于比非容器化环境面临的安全问题更为复杂。

6.1 威胁模型

容器或一般性微服务构架的安全评估，离不开对威胁模型（threat model）的理解。不同情况下，这些模型的形式也不同，不过其中很多都具有共通之处。

如果打算运行不受信任的代码，可能是作为服务提供商，或运行一个PaaS平台，人们可能会故意上传恶意代码，这与小团队开发和托管一个应用程序的要求有所不同。大型企业的情况则不一样，其监管要求代表着某些形式的隔离是强制性的。

如果你是一家服务提供商，目前虚拟化是隔离恶意代码最成熟的技术。不是说它没有安全问题，而是问题的数量很少且攻击面更小。这并不意味着容器不能作为解决方案的一部分，特别是如果正在提供某种语言的运行时（如Ruby或Node.js），下面所讲的很多技术也是完全适用的。

容器是隔离的延伸形式，可以作为一个层次化防护，进一步减少攻击面。

Google在[Borg论文](#)中解释了他们如何为受信任的内部作业提供完全不同的架构：“我们使用Linux chroot牢笼作为同一机器上多个任务之间的主要安全隔离机制”，这种安全性的形式比容器更脆弱，与之相反：“Google应用引擎（GAE）和Google计算引擎（GCE）使用虚拟机和安全沙箱技术运行外部软件。我们在一个作为Borg任务运行的KVM进程中运行每个托管的虚拟机”。

那些关注围绕容器构建微服务架构的安全性的大型和小型公司会特别关心这一章。

安全是一个复杂的领域，需要评估运行中应用程序的内容。不存在什么灵丹妙药，深度防御才是关键，这就是本章将涵盖多种不同方法来加强应用程序安全性的原因。很多方法最终会变成用户使用的工具，但是理解它们保护或不保护的范围，以及能否使用更具体的方案取代通用方案仍然非常重要。

6.2 容器与安全性

Linux容器不是一个单一实体，与FreeBSD的[牢笼](#)（jail）不同，后者采用单一系统调用来创建和配置容器。相反，Linux容器是一组工具，可以进一步加强进程隔离，并远远超过传统的Unix用户ID机制和权限。

容器从根本上是构建于命名空间、cgroup及权能（capability）之上的。

Linux容器的核心是一系列的“命名空间”，效仿于Plan 9操作系统的思想。一个进程命名空间隐藏了所有命名空间之外的进程，给用户分配了一组新的进程ID，包括新的init（pid 1）。一个网络命名空间隐藏了系统的网络接口，并以一个新的集合取而代之。这里的安全方面是，如果一个项目没有可以引用的名字，就不能与之进行交互，也就形成了隔离。

系统中不是所有东西都命名空间化了，还存在大量全局状态。例如，时

钟就不具备命名空间，所以如果有容器设置了系统时间，将影响内核中运行的所有东西。这些程序绝大多数只能被以root运行的进程影响。

另外一个问题是，Linux内核接口非常庞大，有些错误会藏身其中。具有超过300个系统调用，以及数以千计的各类*ioctl*操作，只要有一个存在用户输入验证错误，就会导致内核漏洞。

不过，有很多方法可以用来降低这些风险，接下来我们会对此做一些介绍。

6.3 内核更新

最基本的建议是及时使用安全补丁更新内核。我们并不是很清楚哪些修改会造成安全问题，因为可能很多错误已经造成漏洞，只是还没被人发现。这意味着用户需要经常性地重启容器宿主机，因此也将重启所有的容器。

显然，用户不会想要同时重启整个集群，这会造成所有服务下线，造成分布式系统法定节点缺失，因此需要考虑如何进行管理。CoreOS机器运行着*etcd*，当它们检测自己处于一个集群中时，会从*etcd*中获取一个[重启锁](#)，因此每次只会有一台机器重启。其他系统也需要一个类似的交错重启机制。

6.4 容器更新

用户必须保持宿主机内核和宿主机操作系统被更新，并且保持对运行中的容器进行安全更新修补则是一个关键要求。

如果用户所运行的“胖”容器包含了整个宿主机操作系统，如RHEL或Ubuntu，要保持它们被更新就非常简单。只需要像对待虚拟机那样运行同样的工具软件即可。虽然这无法利用到基于容器的工作流，但至少是个很好理解的问题。

人们担心的情况是，**Docker**的使用是否会造成开发人员把内容不明的随机容器放置到生产环境中。显然，这不是用户希望发生的事。容器必须从头可重现地构建，如果组件中存在安全问题且无法在运行时更新的话，则必须重新构建。

最接近传统做法的方法是拿传统的发行版，使用类似**Puppet**这样的工具对其进行配置，然后将其作为一个基础**Docker**镜像。

微服务的路线是让容器只包含静态链接的二进制文件，如使用**Go**生产的，这样的构建过程就只是简单地利用更新后的依赖对应用程序进行重新构建。然后，升级问题就变成了构建时依赖管理问题。**Java**应用程序也与此类似。

在这些极端情况之间还存在着大量其他模型。重要的是要有一个模型，并且最理想的是有测试，可以测试构建产物。例如，在**bash shellshock bug**被发现之后，用户希望能检查生产环境中的容器，并测试它们是否包含**bash**且存在漏洞。

6.5 **suid**及**guid**二进制文件

Unix长期以来都用着一个设计糟糕的特权提升机制，文件可以被标记上**suid**或**guid**，在这种情况下，程序将以程序的属主（或组）身份运行，而非运行该程序的用户。通常这用于以**root**运行那些需要特殊权限的程序。如果程序编写得很好，那么它们会尽快丢弃这个**root**状态，在解析任何用户输入之前，并尽可能缩小使用范围。如果不是这么做，就存在被破坏的危险。

典型的可**suid**成**root**的二进制文件包括**su**、**sudo**、**mount**及**ping**。这些文件大多数在容器内是不需要的，因此可以对其进行删除、移除**suid**位，或使用**nosuid**选项挂载容器根目录以忽略它们。这是安全测试套件可以测试的东西。

需要注意的是，专门设计用于运行在容器内的发行版可以解决这类问题，但尚不多见。目前的发行版会假定这些基础命令都是必需的。有些轻量级容器基础系统使用了**Busybox**核心小工具，这类工具没有安全地

实现suid程序，未丢弃特权，因此千万不要在运行时启用suid。

使用下面的命令可以查找系统中所有的suid和guid文件：

```
find / -xdev -perm -4000 -a -type f -print  
find / -xdev -perm -2000 -a -type f -print
```

6.6 容器内的root

容器的设计原则是保证容器内没有需要root权限的东西。尤其不要使用`docker run --privileged ...`，这将使用完全的root权限运行容器，并能执行宿主机可以操作的任何事情。

权能（见6.7节）是在需要时将root的权能子集赋予进程的一种方法。

用户命名空间（见6.12节）旨在提供一匹神奇的“不是root的root”独角兽，允许root的使用。6.12节中会详细讨论这一魔法。

遗憾的是，很多现存的容器都需要root权限，往往为了一些其实只需要修复即可的不好原因。其中一个例子是[Docker registry](#)，它会在一个root拥有的目录中创建锁文件，除非用户禁用搜索功能，否则这个问题依然存在。

6.7 权能

Linux对于root拥有的权能具有一些细粒度的权限，可以独立地分配给容器。capabilities(7)的[帮助页](#)中罗列了各个权能对应的操作。例如：

```
docker run --cap-add=NET_ADMIN ubuntu sh -c "ip link eth0 down"
```

将只使用NET_ADMIN权限来停止容器内的eth0接口，而这是完成此项操作的最低要求。可以以此运行那些需要suid的二进制文件，而不需要以root身份运行整个容器，但总的来说还是应该避免这么做，为了保持最

大化的安全性，容器应在不带权能的情况下运行。

6.8 seccomp

权能限制的是可以采取的操作类型，而seccomp过滤器则是完全移除了使用特定系统调用或特定参数调用的能力。

这一方案的困难之处在于确定应用程序需要使用的调用。用户可以使用跟踪的方式，不过必须覆盖100%的代码，而这很难做到。用户的代码可能会改变，使用的调用也可能改变。因此，对于一般用途的用例来说，最简单的策略是使用黑名单，过滤那些管理专用的，并且一般不为应用程序所用或完全过时的系统调用。大约25%的调用可以归入这些类别。

截至本书编写时，只有Docker lxc后端具有运行seccomp过滤器的钩子，默认的libcontainer后端则没有。在Docker仓库的contrib目录中有一些过滤器示例。为应用程序设置自身的过滤器也是可能的。

6.9 内核安全框架

Linux支持多个内核安全框架，其中最著名的是由NAS设计的与Red Hat Linux一起发行的SELinux。与Ubuntu一起发行的AppArmor与此类似。

SELinux是一个实现强制访问控制策略的框架。需要注意的是，它只是一个框架，必须定义实际的策略。但定义策略的人少之又少，其过程不仅复杂，且缺乏文档。因此，多数人使用的是供应商提供的策略。事实上，尽管存在一本解释SELinux的填色书，Google搜索建议中最受欢迎的依然是“关闭”。

如果你不是在一个长期运作的组织里工作，如这些策略的发源地——美国国防部，定义确实管用的安全策略是件很困难的事。不过，原则上可以将其应用在隔离不同类型数据的访问上，对于PCI合规而言，是HR数据或个人信息。遗憾的是，支持这些用途的工具还相当缺乏。

不过，我们建议尽可能不要禁用供应商策略，并且要理解如何标记允许访问的项目。供应商策略好过没有策略。容器的策略相对较新，可能不会一直很好地工作。

Docker从1.3版本开始支持SELinux，不过默认是关闭的。`docker --selinux-enabled`将启用这个功能，而类似`--security-opt="label:user:USER"`的选项可以在运行容器时设置用户、角色、类型及标签。

6.10 资源限制及cgroup

内核cgroup功能是由Google创建的，用于在Borg调度器（Kubernetes的前身）中运行规模化的应用程序。

一个cgroup限制着分配给一组进程（通常是一个容器）的资源。cgroup控制器集合数量庞大而复杂，不过重要的几个与CPU时间、内存和存储被限制有关。

最简单的是内存和CPU访问限制。可通过`docker run -m 128m`来设置内存用量。通过`docker run --cpuset=0-3`来设置容器运行所在的CPU，而通过`docker run --cpu-shares=512`来分配CPU时间共享。

重要的一点是，要停掉占用了所有内存、IO带宽和CPU时间的应用程序，它将影响同一台宿主机上运行的其他应用程序。

根据容器设置的方式，可能还存在一些干扰。例如，除非彻底地分配了CPU，否则缓存将是共享的，而如果共享了IO设备，如网络或磁盘，则存在IO竞争。这种情况会带来多大的影响取决于负载以及超额申请的资源数量，不过通常这是一个吞吐量的问题，而非安全问题，不过[旁路攻击](#)还是有可能发生。

Docker 1.6增加了`cgroup-parent`选项，它可以将容器附加到现存cgroup中。这意味着用户可以在Docker之外使用其他工具管理cgroup，然后选择要添加到容器中的cgroup。这使得用户可以使用所有的cgroup控制，而无需在意它们是否在Docker命令行中公开。

6.11 ulimit

Docker 1.6引入了控制每个容器**ulimit**的能力。这是一个在每个进程基础上控制资源的古老的Unix功能。需要注意的是**ulimit**也可以用来配置最大的处理器数量。出于不同目的，在资源控制上**ulimit**可能比**cgroup**更为简单，系统管理员更熟悉。

此前，容器会继承Docker进程的**ulimit**，这个限制一般都设置得相当高。现在可以用以下命令来设置可创建进程数的默认**ulimit**：软限制为1024，硬限制为2048。

```
docker -d --default-ulimit nproc=1024:2048
```

软限制是要强制执行的限制值，不过进程可以对其进行提高，最高至硬限制值。

然后，可以在每个容器级别上覆盖这些限制值，例如：

```
docker run -d --ulimit nproc=2048:4096 httpd
```

这将提高httpd容器的进程**ulimit**。

6.12 用户命名空间

相比其他命名空间，用户命名空间加入Linux内核的时间要迟一些，也较为复杂。

其思想与其他命名空间形式类似，只是用于用户ID（**uid**）和组ID（**gid**）。特别是，处于用户命名空间内的容器中的**root**用户和**uid 0**，可以映射到宿主机不同的非特权用户上。

这意味着，容器的**root**用户在宿主机系统中只是一个普通用户，无法做任何特殊的事情。那么它如何能称为**root**？对于属于其容器的资源来

说，它就是root，如容器的网卡，因此它可以重新配置容器的网卡，或绑定到80端口上。

这引入了更多复杂性，由于uid只是存储在文件系统中并分配权限给文件，因此对于不同的命名空间，它们的意义有所不同。这也意味着这项功能从引入到适合生产环境使用之间经历了长时间的延误。这样的延误意味着它错过了REHL 7.0的最后期限，无法得到来自Docker的直接支持，不过预计不会很久，在lxc驱动程序里也会得到部分支持。

用户命名空间另一个不太明显的优点是创建命名空间完全不需要root权限。这让Docker守护进程可以减少内部需要以root运行的代码量。

在拉取请求[#12648](#)中引入了最基础的功能：容器内root用户不对应宿主主机系统的root用户，但由于用户命名空间代码与libnetwork代码存在冲突，这一请求错过了Docker 1.7的最后期限，只能延后到Docker 1.8^[1]。用户命名空间更复杂的功能就差得更远了。

6.13 镜像验证

Docker 1.3引入了Docker[镜像验证](#)的路线图的最初部分。这仅是个开始，其目标是追寻Linux包管理器的路线，打造一个完整的模型。在这个模型中用户有一组信任的密钥，其中可能包括了受信任的供应商以及用户所在组织的签名，未经签名的镜像则不允许运行。

当前的实现还只是一个开始，验证签名失败时它给出警告，但不会阻止未签名包的安装，因此它无法提供任何实际的安全性价值。不过，它只是路线图的一个开端，可以在签名镜像问题[#2700](#)上查看其计划，并对其实现进行跟踪。

6.14 安全地运行Docker守护进程

默认情况下，Docker守护进程只能通过本地Unix域套接字进行访问，这意味着可以在本地通过套接字的权限控制其访问，而远程访问是完全不

可能的。

访问Docker守护进程将获取整台计算机完整的root权限，因此用户能以root身份运行Docker容器来执行宿主机上的任何命令，所以对访问的保护尤为重要。

如果用户使用-H选项强行将Docker绑定到一个TCP端口上以便进行远程控制（而不是通过ssh进行控制），那么用户需要使用iptables和SSL来控制其他访问。对多数用例而言，不推荐这么做。

6.15 监控

要了解容器的运行情况，包括发现安全相关的问题，容器的监控非常重要。本书中有专门的一章讲述监控，读者可以从那里开始设计自己的监控策略。

6.16 设备

如果容器需要访问提供硬件或虚拟设备访问的设备结点，可以通过--device选项传递所需的设备并设置权限。

例如，`docker run --device=/dev/snd:/dev/snd:r ...`将添加/dev/snd音频设备到容器中，使其在容器中只读。

由于设备节点允许ioctl访问，同时下层内核驱动程序中可能存在安全问题，它们成为了攻击面的一部分，尤其是特殊设备。因此，只提供需要的设备及最小的权限是最佳的策略。

6.17 挂载点

在使用默认的`libcontainer`驱动程序时，`Docker`会很小心地以只读权限挂载必要的虚拟文件系统。如果用户使用的是`lxc`驱动程序，则这一步需要自己完成。如果容器内具有`root`权限，对类似`/sys`和`/proc/bus`这类文件系统的写权限可能造成宿主机受损。

6.18 ssh

不要在容器里运行`ssh`。要习惯从宿主机上管理它们。这不仅简化了容器，还消除了权限的复杂度的级别。习惯在需要使用`Docker`提供的工具来查看容器运行情况。

简化的容器更易于管理，并且从宿主机进入容器也相当简单。实际上，`Docker`最终可以非常好地管理进程，而`ssh`是不必要的，还会增加复杂度。

6.19 私钥分发

服务需要使用密钥去访问其他服务，如访问AWS的密钥，或用于验证它们是否可以加入集群或访问资源。密钥管理很难，是个有待解决的问题，现在存在着或好或坏的方法，同时一些有用的工具正如雨后春笋般出现。

密钥应当只在必要时进行分发，如果它们被发现，最不可能的访问也会受到威胁。密钥应定期进行轮换，限制偶发缺口的持续时间。密钥不应被签入源代码中，因为更新密钥不应要求做一次新部署，而且它们最终将出现在公共的GitHub仓库中。

能够对密钥访问进行审计也是一个理想的目标，这样可以跟踪其使用。

出现的服务包括来自[SquareKeywhiz](#)和来自Hashicorp的[Vault](#)。`Kubernetes`也有一份优秀的关于密钥管理的[设计文档](#)，可作为密钥管理框架的基础。

6.20 位置

如果运行在宿主机或虚拟机上的所有服务对相同的数据都具有相同级别的访问权限，可能就不太需要担心隔离的问题了。毕竟，这不会比单体应用程序更糟，单体应用程序的组件并没有真正的隔离。

虽然微服务可以让用户构建一个具备高级别的特权分离的更安全的架构，但对于不需要访问敏感数据的应用程序，这并不是首要目标。用户需要将安全方面的努力放在能带来最大收益的地方。

面向用户的服务面临着不可信的输入，显然是一个薄弱点，应与重要数据的所有访问隔离开。与PCI合规相关的端点不应与其他服务运行在同一台宿主机上，并且应隔离到自己的集群，以减少审计边界。

在第7章中，我们将细述Docker中镜像的构建。

[1] 该请求最终合并入Docker 1.9。——译者注

第7章 构建镜像

镜像是所有容器的运行之本，因此，在构建Docker基础设施时，掌控构建镜像的艺术非常必要。

构建镜像的方式将决定容器部署的速度、从容器获取日志的难易程度、可配置的多少以及其安全程度。尽管构建镜像时的首要关注点是容器可按预期运行，但在生产环境中，这里所列举的因素将变得非常重要。

在着手构建镜像之前，我们需要理解其实现方式的几个方面。

7.1 此镜像非彼镜像

虽然表面上Docker镜像与虚拟机镜像并无太大差异，但它们的实现方式却完全不同。虚拟机镜像提供了完整的文件系统虚拟化：镜像中的文件系统可以与镜像所在宿主机的文件系统完全不同。虚拟机镜像通常以数据卷的形式实现，以大文件形式存储在宿主机操作系统中。一旦为虚拟机分配了数据卷，虚拟机里的访客操作系统就会在这个卷上创建并格式化出一个或多个分区。虚拟机管理程序会将这些文件作为原始磁盘展现给访客操作系统。

这种虚拟化文件系统的方法提供了巨大的隔离性和灵活性，但其效率可能不佳。例如，在使用同一个镜像运行多个虚拟机，或需要来自同一基础镜像的多个不同镜像时，效率就会比较低。克隆虚拟机的标准方法是为新镜像创建一份文件系统的新副本，使得两个镜像里的文件系统可以独立演进。这种方法在创建副本时的磁盘空间占用和时间花费上代价昂贵，也正因为这样，虚拟机厂商在这些普通复制无法正常工作的场景中，依靠写时复制（copy-on-write, CoW）技术来提高镜像的使用效率。

7.1.1 写时复制与高效的镜像存储与分发

在创建和运行多个从同一基线数据启动的进程时，写时复制技术可以节省大量时间和空间。对于虚拟化而言，假设有20台虚拟机需要使用相同的基础镜像，使用写时复制的话，用户就不需要创建20份该镜像的副本（每个虚拟机一份）。相反，所有虚拟机可以从相同的镜像文件启动，带来更快的启动速度，并节省运行所有虚拟机需要的大量磁盘空间。写时复制会让每个虚拟机里的访客操作系统以为它们是在基础镜像中对文件系统进行独立修改，它的实现方式是为每个虚拟机提供一个在共享基础镜像之上的叠加层，这个层可以独立于其他虚拟机进行修改。每当操作系统尝试对文件系统进行修改时，实际上是发生在这个叠加层上的，基础镜像保持原封不动。

在操作系统想要对文件系统进行修改时，处于这些场景之后的虚拟机会把这些被编辑的磁盘扇区复制到叠加层上，并将这些副本提供给访客操作系统，让其当作原始版本。然后，虚拟机管理程序就允许访客操作系统修改叠加层上的副本，保持基础镜像中的原始扇区不变。从此刻开始，这个虚拟机就再也看不到这些扇区的原始共享副本，只能看到叠加层中的副本（如图7-1所示）。虚拟机管理程序为访客操作系统提供了一个“幻象”：作为叠加层与基础镜像合并结果的文件系统会被当作一个单一的数据卷。

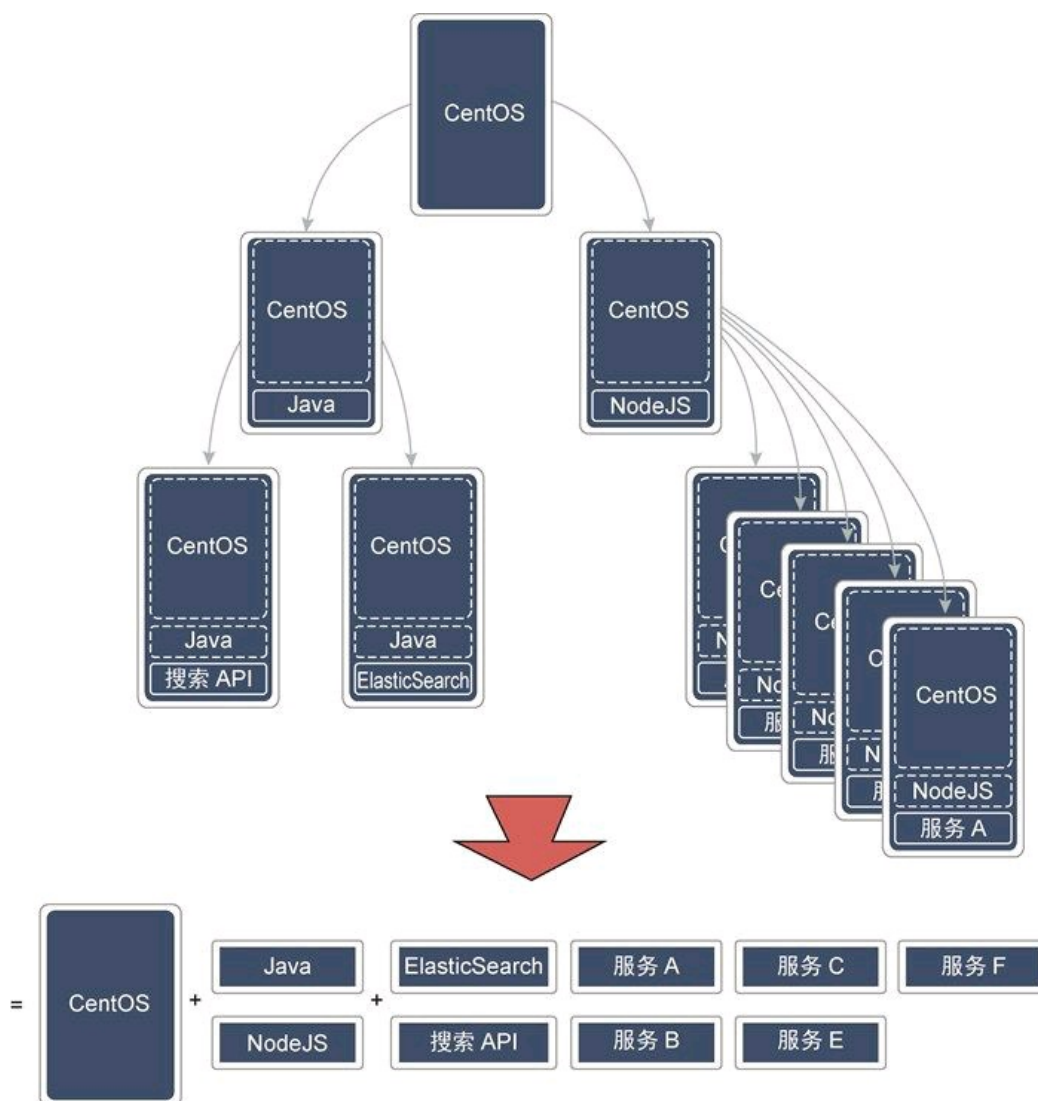


图7-1

Docker镜像生来就是基于写时复制技术的，且与标准的虚拟机不同，Docker的镜像并不是完全虚拟化的，它们是构建于宿主机的文件系统之上的。这种方法是否比完全虚拟化具有性能优势还有待考证，而且很大程度上取决于具体用例。例如，虚拟机世界里的写时复制通常是基于扇区的，也就是，只有基础镜像上有变动的文件磁盘扇区会被复制到叠加层上进行编辑，而对Docker而言，整个文件会被复制和编辑，因此即便只是大文件的一部分被修改，整个文件都需要被复制。另一方面，使用Docker的镜像，访客和宿主机操作系统之间不需要文件系统转换。我们讨论构建Docker镜像时重要的一点是，Docker进一步发挥了写时复制技术的作用，可以很容易地堆叠多个写时复制叠加层以创建一个镜像或一系列相关的镜像。

7.1.2 Docker对写时复制的使用

Docker使用写时复制的主要原因有二。其一是让用户可以交互地构建镜像，一次添加一个层。其二则具有更深远的意义，与镜像的存储和分发有关。在我们构建系统时，我们通常采取的方式是，所有服务都基于相同的操作系统的小集合，甚至是共享一些基础配置。以这种方式设置的容器镜像彼此的差异只在于配置的“最后一英里”，这最后一英里只包含了将该镜像与其他镜像区分开来的内容，有些时候，容器使用的是完全一样的镜像。在这些场景中，写时复制可以非常有效地节省时间和空间。

Docker还使用写时复制将容器运行于一个叠加层上，而非直接运行于镜像上。原始镜像是以只读模式被使用的，容器可能对文件系统做的任何修改都只会在这个叠加层上执行。读者可能阅读过“Docker镜像是不可变的”这样的Docker文献，其确切意思是：一旦镜像被创建，它就无法再被修改，因此你能做的事是在它的基础上构建新镜像。

Docker叠加层的使用其真正强大之处在于这些叠加层可以跨宿主机进行共享。每个叠加层包含了对基础镜像的引用，而后者又是另外一个叠加层。每个叠加层都拥有唯一的ID，以及一个可选的名称和版本号。Docker镜像的具名叠加层是在镜像仓库中存储与共享的。部署一个容器时，Docker会检查容器所需的镜像是否在本地仓库中已经存在。如果在本地不存在，Docker在镜像仓库检索这个镜像，并拉取该镜像所有叠加层的引用，然后确定哪些层已经在本地存在，并下载那些缺失的叠加层。

这种方法可以减少保存宿主机中所有镜像所需的空间，并能显著地减少新镜像的下载时间。例如，在一个运行10个容器的场景中，10个镜像的主干都来自于相同的基础CentOS 7镜像，宿主机只需要下载一次基础镜像，以及10个不同的叠加层，无须下载10个都包含CentOS 7完整副本的镜像。同样，下载更新的镜像只需要下载最新的几个叠加层。

本章后面将详细讨论如何利用这些特性，但我们先来看一下构建镜像的主要方面：使其工作。

7.2 镜像构建基本原理

从最基本的层面讲，构建一个容器镜像（后面简称为镜像）可以通过两种方法完成。第一个方法是从一个基础镜像（**ubuntu-14.04**）启动一个容器，在容器内运行一系列命令，如安装软件包、编辑配置文件等，一旦镜像处于期望状态，对其进行保存。

我们来看看它是如何工作的。在一个终端中，使用**ubuntu**的基础镜像启动一个运行**/bin/bash**可交互的容器。一旦进入容器内的**shell**，我们就在根目录中创建一个名为**docker-was-here**的文件。这项操作不应修改基础镜像。相反，新文件应被创建在容器的文件系统叠加层上：

```
$ docker run -ti ubuntu /bin/bash
root@4621ac608b25:/# pwd
/
root@4621ac608b25:/# ls
bin boot dev etc home lib lib64 media mnt opt proc
root run sbin srv sys tmp usr var
root@4621ac608b25:/# touch docker-was-here
root@4621ac608b25:/#
```

现在，我们在第二个终端中创建一个基于上述容器内容的新镜像，上述示例中其ID是**4621ac608b25**。

```
$ docker commit 4621ac608b25 my-new-image
6aeffe57ec698e0e5d618bd7b8202adad5c6a826694b26cb95448dda788d4ed8
```

最后，我们在这个终端中启动一个新容器，这一次使用的是我们新建的**my-new-image**镜像。我们可以验证镜像包含了我们自建的**docker-was-here**文件。

```
$ docker run -ti my-new-image /bin/bash
root@50d33db925e4:/# ls
bin boot dev docker-was-here etc home lib lib64 media
mnt opt proc root run sbin srv sys tmp usr var
root@50d33db925e4:/# ls -la
total 72
drwxr-xr-x 32 root root 4096 May  1 03:33 .
drwxr-xr-x 32 root root 4096 May  1 03:33 ..
-rwxr-xr-x  1 root root    0 May  1 03:33 .dockerenv
-rwxr-xr-x  1 root root    0 May  1 03:33 .dockerinit
drwxr-xr-x  2 root root 4096 Mar 20 05:22 bin
```

```
drwxr-xr-x  2 root root 4096 Apr 10  2014 boot
drwxr-xr-x  5 root root  380 May  1 03:33 dev
-rw-r--r--  1 root root    0 May  1 03:31 docker-was-here
drwxr-xr-x 64 root root 4096 May  1 03:33 etc
      ....
drwxr-xr-x 12 root root 4096 Apr 21 22:18 var
root@50d33db925e4:/#
```

尽管这种构建镜像的交互方法非常直观，但无助于可重现和自动化。对生产环境设置而言，很有必要使用可轻松重现的方法进行镜像自动化构建。**Docker**提供了一个方法来完成这件事，该方法基于一个名为**Dockerfile**的文件。

一个**Dockerfile**包含了一系列指令，**Docker**在一个容器中运行这些指令以产生一个镜像。这些指令可以分为两组：一组修改镜像的文件系统，一组修改镜像的元数据。修改文件系统的指令示例之一是**ADD**——将URL定义的远程地址文件写入到镜像文件系统中，或**RUN**——在镜像上运行一个命令。修改元数据的指令示例之一是**CMD**——设置了容器进程启动时要运行的默认命令及其参数。

在使用**docker build**时，**Docker**会以**Dockerfile**的**FROM**指令指定的基础镜像来启动一个临时的容器，然后在这个容器的上下文中运行每条指令。**Docker**会为每条指令创建一个中间镜像。这是为了方便用户渐进地构建镜像：在**Dockerfile**里修改或添加一条指令时，**Docker**知道在此之前的指令并未发生变化，所以它会使用运行完上一条指令后构建的镜像。

例如，要使用**Dockerfile**构建出和此前以交互方式构建而来的相同的镜像，首先创建一个新目录，然后在该目录中创建一个名为**Dockerfile**的文件，其内容为：

```
FROM ubuntu
MAINTAINER Me Myself and I
RUN touch /docker-was-here
```

接下来，我们告诉**Docker**使用这个**Dockerfile**来构建**my-new-image**镜像：

```
$ docker build -t my-new-image .
```

Docker默认会在当前目录中查找**Dockerfile**。如果用户使用的是其他文件名或Dockerfile位于其他位置，可以使用**-f**来告诉Docker所使用的Dockerfile路径：

```
$ docker build -t my-new-image -f my-other-dockerfile .
```

7.2.1 分层的文件系统和空间控管

正如前面所说，Docker镜像呈现出的是一个分层构架，镜像由一堆的文件系统叠加层组成。每一层都是源自前一层的一组文件的增加、修改和删除。在层里增加文件时，新文件将被创建。在层里删除文件时，这个文件会被标记为已删除，但是请注意，这个文件还包含在前面的层里。在层里修改文件时，取决于Docker所运行的存储驱动程序（在后面的存储章节中详述），要么整个文件在新层里被重新创建，或者只有这个文件的部分磁盘扇区在新层里被新扇区所替换。不论哪种方式，旧文件在前面的层中保持不变，而新层则包含了其新的修改。在每个层都有一个镜像，这个镜像是在基础镜像之上依次叠加先前层的结果。在所得到的镜像之上同样也是前面所有层的结果。

在构建Docker镜像时，通常会从一个现有基础镜像开始，这个镜像可能已经包含了很多层。Docker会按顺序运行Dockerfile里的每条指令，在每条指令结束时Docker会以运行该指令产生的文件系统的变化生成一个新层。这是一项非常好的功能，因为它允许渐进式开发镜像，而不必每次都等待所有的指令运行。例如，假设因为第10条指令包含一个错误，导致Docker构建镜像时失败，然后当用户在修复了这个失败的指令后重新尝试构建时，Docker将不需要再次运行前面的9条指令。相反，它将以上一次正确构建的层为起点，然后从之前的失败指令开始继续构建。这可大大节省时间，因为有些指令可能会运行一些比较耗时的命令。

这种分层架构在部署阶段同样具有优势，因为在部署一个新镜像时，某些较深的层可能已经存在于宿主机中，因此只有新层需要通过网络发送过去。在运行基于同一个或类似镜像的多个容器时，这项功能将极大地减少所需的时间和空间（如图7-2所示）。

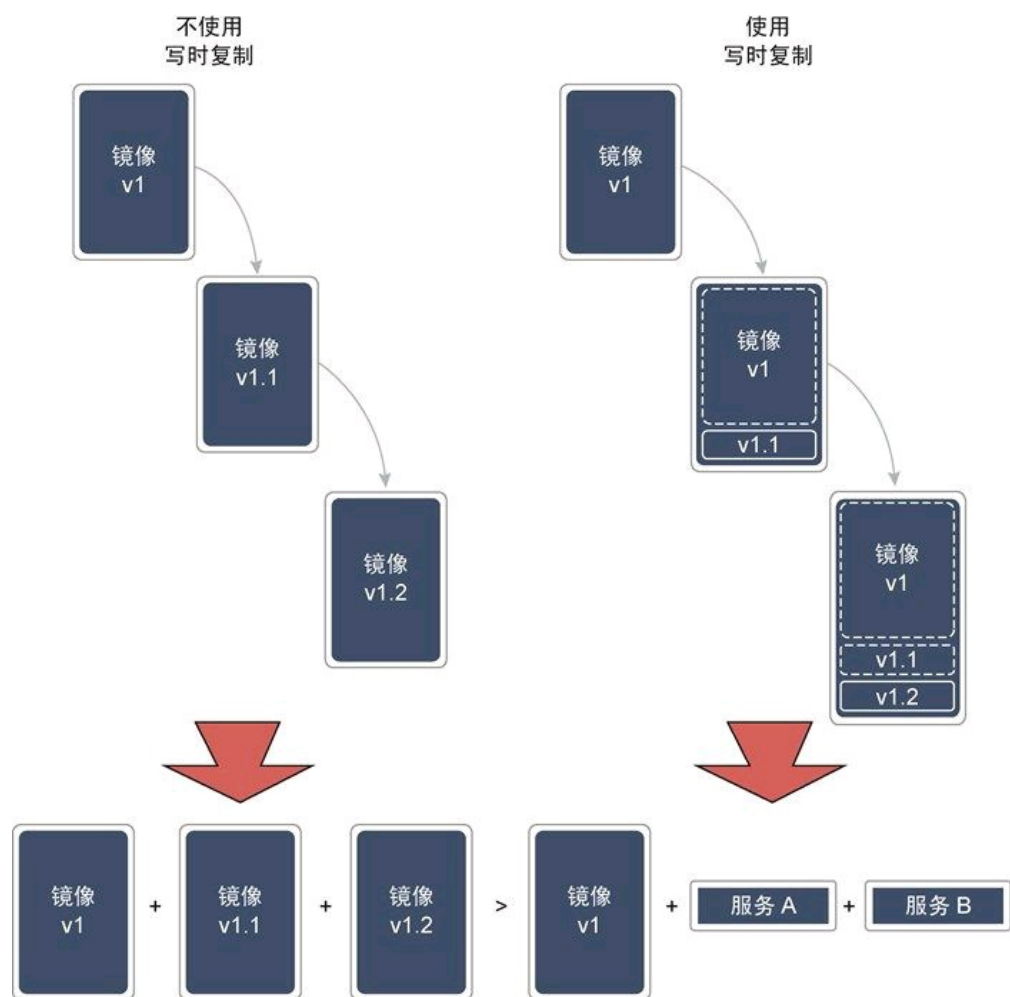


图7-2

这种分层架构也带来了一些在现实场景中需要考虑的注意事项。其中之一是，镜像无法缩小。如果一个镜像所有层加起来在文件系统中是500 MB，扩展它的任何自定义镜像在文件系统中至少需要500 MB，即使上层删除了下层的文件。

镜像大小相当重要，特别是在宿主机上安装这些镜像时，不仅因为下载镜像到宿主机所耗费的时间取决于它的大小，还因为镜像文件越大宿主机上所需的空间也越多。在开发过程中，其大小也很重要，因为对于一个新的开发人员，或对于启用一组新的Docker宿主机的人，甚至对持续集成/持续部署的服务器来说，都有可能需要花费大量时间下载开发过程所需容器的所有镜像。考虑到Docker是用来提升开发过程，这多少会让人有些沮丧。

注意，如果正在部署一个微服务架构，减小镜像大小尤为重要。但是，如果正在部署大型的虚拟机类容器，由于你可能使用了一个功能完整的操作系统，本节的大部分内容并不会给你带来多大作用。

从“小”做起

控制镜像空间要从最小化的基础镜像开始。在极端情况下，可以从一个空的文件系统开始，并在其中部署操作系统，不过这应该不是应用最广的方法。

下一个选项是类似[busybox](#)或[alpine](#)的微型发行版。[Busybox](#)大约2.5 MB，一开始是为嵌入式应用程序创建的。它包含了最基本的Unix工具供用户使用，不过用户也可以创建添加（或删除）了额外命令的自己的[busybox](#)发行版。[Busybox](#)对支持运行静态编译二进制文件的镜像支持良好，如用Go写的进程。

[Alpine](#)在[Busybox](#)基础上，扩展添加了一个以安全为重点的内核构建版本及一个名为[apk](#)的包管理器，并是基于[Musl libc](#)——一个更轻也可能更快的libc版本。[Alpine](#)可以作为容器通用的Linux发行版，不过用户将不得不花费更多的力气完成所需功能的配置：尽管[Alpine](#)提供了一个包管理器，可用的包列表比[Debian](#)或[CentOS](#)这类完整的发行版要小得多。相比全功能发行版，[Alpine](#)的优势在于它活动部件更少，因此更小巧，且更易于理解，而且作为一个必然结果，也更易于加固。

下一个选项是使用主流Linux发行版版本（如[Ubuntu](#)或[CentOS](#)）的容器优化版本。这些容器通常运行完整发行版的精简版本，它们删除了所有的桌面程序，且具有针对生产环境服务器优化的配置。这些镜像一般只有几百MB，如[Ubuntu 14.04](#)大约是190 MB，而[CentOS 7](#)大约是215 MB，不过它们提供了一个全功能的操作系统。这是目前在构建自定义镜像时最容易入手的地方，因为它们对打包服务的支持相当好，而且网络上存在大量的文档和手册可以作参考。[Docker registry](#)充满了这类镜像，大多数镜像由[Docker](#)公司直接支持。

一个明智的选择是，标准化出一个特定的镜像版本，并尽可能地将其用于所有容器。如果宿主机里所有的容器都具有相同的基础镜像，一旦第一个容器下载完成，下载其余容器的速度将更快，因为它们无须重新下载基础镜像层。不过需要注意的是，只有基础镜像已经下载好之后，才能在下载其他镜像时发挥作用。截至本文撰写时，如果在基础镜像尚未

下载完成时，同时下载多个镜像，那么每个镜像都会下载一次基础镜像，因为在开始下载镜像时本地仓库还不存在这个基础镜像。

7.2.2 保持镜像小巧

在选定一个小的基础镜像之后，下一步是在运行Dockerfile之后保持镜像的小巧。

每运行Dockerfile里的一个命令，就会生成一个新的镜像层。层生成时，一个新的最小镜像大小就被设定了：即使用户在Dockerfile的下一个命令中删除文件，也不会释放任何空间，位于宿主机文件系统上的镜像大小也不会缩小。

出于这个原因，如何在Dockerfile中组织命令将影响最终的镜像大小。例如，通过包管理器安装一个软件包：当调用包管理器时，它的索引会被更新，它会下载一些包到缓存目录，并在将包中文件放置到文件系统的最终位置前，将包展开到预演区域。如果用户像往常一样运行包安装命令，这些永远也用不上的缓存包文件将会永远地成为镜像的一部分。不过，如果用户在同一条安装命令中删除它们，这些文件就会像从未存在过一样。

例如，可以像这样在一个单一步骤里安装Scala并执行清理操作：

```
RUN curl -o /tmp/scala-2.10.2.tgz http://www.scala-lang.org/
files/archive/scala-2.10.2.tgz \
    && tar xzf /tmp/scala-2.10.2.tgz -C /usr/share/ \
    && ln -s /usr/share/scala-2.10.2 /usr/share/scala \
    && for i in scala scalc fsc scaladoc scalap; do ln -s /usr/
share/scala/bin/${i} /usr/bin/${i}; done \
    && rm -f /tmp/scala-2.10.2.tgz
```

在上面的示例中，如果像这样独立地运行上述命令：

```
RUN curl -o /tmp/scala-2.10.2.tgz http://www.scala-lang.org/
files/archive/scala-2.10.2.tgz
RUN tar xzf /tmp/scala-2.10.2.tgz -C /usr/share/
RUN ln -s /usr/share/scala-2.10.2 /usr/share/scala
RUN for i in scala scalc fsc scaladoc scalap; do ln -s /usr/
share/scala/bin/${i} /usr/bin/${i}; done
RUN rm -f /tmp/scala-2.10.2.tgz
```

其后的镜像将包含这个`.tgz`文件，尽管在最后一条命令之后无法在文件系统中看到这个文件。

7.2.3 让镜像可重用

有两种可以配置容器中运行进程的方法：一种方法是通过环境变量将配置传递给容器内部，另一种是将配置文件和/或目录挂载到容器中。两种方法都发生在容器启动时期。两种方法都是非常有用，有各自的应用场所，但它们本质上有很大不同。

通过环境变量配置

在Docker启动一个容器时，它可以将环境变量转发给容器进程，进而转发给运行于容器内的进程。我们来看一下它是如何工作的。启动一个容器，在shell中运行一个命令来打印环境变量`MY_VAR`的值：

```
$ docker run --rm busybox /bin/sh -c 'echo "my variable is $MY_VAR"'
my variable is
```

这个环境变量未在容器中预定义，因此并没有值。现在在容器内运行相同的命令，不过这次我们通过Docker传递一个环境变量给容器：

```
$ docker run -e "MY_VAR=docker-was-here" --rm busybox /bin/sh -c 'echo "my variable is $MY_VAR"'
my variable is docker-was-here
```

理想情况下，通过环境变量，容器内的进程是完全可配置的。有时，我们会容器化那些通过配置文件获取配置的服务。我们将在下一节中讨论如何处理这些场景，不过现在我们先专注于环境变量的直接使用。

使用环境变量可以在进程及其配置间提供大量的隔离，这在“十二要素”（[12 factor](#)，一份用于构建基于服务的应用程序的宣言）中被认为是更好的方法。Docker为此设计了一个参数选项，在启动时将这些环境变量传递给容器。

这种分离的好处在于用户可以使用相同的镜像，而不管如何计算用于运

行容器的配置。当容器化的进程通过环境获取它的配置时，所有的配置责任都属于调用Docker来启动容器的那个进程。这个模式带来了极大的灵活性，因为配置可以来自容器启动脚本的硬编码中，或来自文件，或来自一些分布式配置服务，甚至是来自于调度器。

7.2.4 在进程无法被配置时，通过环境变量让镜像可配置

有时，用户需要包装一个无法通过环境变量配置的服务。最常见的场景是从一个或多个配置文件（如nginx）读取配置的进程。

1. 使用模板文件

有一个应用广泛的模式用于处理这种场景：使用一个入口点脚本，获取环境变量并在文件系统上生成配置文件，然后调用实际进程，该进程将在启动时读取那些新生成的配置文件。

我们来看一个示例。构建一个容器使用node-pushserver来给iOS和Android手机发送推送通知。对于这个例子，我们会创建一个名为entrypoint.sh的shell脚本，并在Dockerfile中将其添加到容器中：

```
from node:0.10

RUN npm install node-pushserver -g \
    && npm install debug -g

ADD entrypoint.sh /entrypoint.sh
ADD config.json.template /config.json.template
ADD cert-dev.pem /cert-dev.pem
ADD key-dev.pem /key-dev.pem

ENV APP_PORT 8000
ENV CERT_PATH /cert-dev.pem
ENV KEY_PATH /key-dev.pem
ENV GATEWAY_ADDRESS gateway.push.apple.com
ENV FEEDBACK_ADDRESS feedback.push.apple.com

CMD ["/entrypoint.sh"]
```

这个Dockerfile具有多个环境变量默认值。如上所见，这些环境变量决定了从服务自身端口到MongoDB的主机/端口，以及所需证书的位置，甚至是所使用的苹果服务器（在开发环境和生产环境中，它们可能会有所不同）。最后，容器将运行的进程是我们自己的entrypoint.sh，它看起来像是这样的：

```
#!/bin/sh
# 渲染一个模板配置文件
# 展开变量 + 保留格式
render_template() {
    eval "echo \"\$(cat $1)\""
}

## 如果没有MongoDB前缀，则拒绝启动
[ -z "MONGODB_CONNECT_URL" ] && echo "ERROR: you need to specify MONGODB_CONNECT_URL" && exit -1

## 对引号进行转义，以免在渲染时被删除
cat /config.json.template | sed s/"/\\\\"/g > /config.json.escaped
## 渲染模板
render_template /config.json.escaped > /config.json
cat /config.json
/usr/local/bin/pushserver -c /config.json
```

这个脚本文件有些需要注意的地方。首先，我们定义了一个函数render_template，参数是一个文件名，它将展开其中的环境变量，并返回其内容。

接着，我们对因为某些关键配置不存在就很快失败的情况做了严格把关。在这里，我们要求调用者提供一个名为MONGODB_CONNECT_URL的环境变量，它没有默认值。

最后是从模板生成配置文件的部分。模板看起来是这样的：

```
{
  "webPort": ${APP_PORT},

  "mongodbUrl": "${MONGODB_CONNECT_URL}",
  "apn": {
    "connection": {
      "gateway": "${GATEWAY_ADDRESS}",
      "cert": "${CERT_PATH}",
      "key": "${KEY_PATH}"
    }
  }
}
```

```

    },
    "feedback": {
      "address": "${FEEDBACK_ADDRESS}",
      "cert": "${CERT_PATH}",
      "key": "${KEY_PATH}",
      "interval": 43200,
      "batchFeedback": true
    }
  }
}

```

我们对双引号进行了转义，否则超级简单的渲染引擎 `render_template` 会将它们删除。然后，我们调用 `render_template`，它将获取转义过双引号的文件，并生成最终的配置文件。看起来就像这样：

```

{
  "webPort": 8300,

  "mongodbUrl":
  "mongodb://10.54.199.197/stagingpushserver,mongodb://10.54.199.209?
  replicaSet=rs0&readPreference=primaryPreferred",
  "apn": {
    "connection": {
      "gateway": "gateway.push.apple.com",
      "cert": "/certs/apn-cert.pem",
      "key": "/certs/apn-key.pem"
    },
    "feedback": {
      "address": "feedback.push.apple.com",
      "cert": "/certs/apn-cert.pem",
      "key": "/certs/apn-key.pem",
      "interval": 43200,
      "batchFeedback": true
    }
  }
}

```

最后，这个脚本通过 `/usr/local/bin/pushserver -c /config.json` 调用真正的服务，它将加载我们新生成的 `config.json` 文件。

2. 挂载配置文件

值得注意的是，我们前面生成的配置文件也加载了两个证书，虽然我们也可以把它们当作环境变量传递，如`echo ${CERT} > /certs/apn-cert.pem`，在这个实例中，我们还是以挂载文件的方式来提供。这是一个处理这些以文件进行配置的容器的替代方法。

在启动一个容器时，用户可以挂载本地目录或文件到容器文件系统中，并且这发生在容器进程启动之前。有鉴于此，配置上述容器的另一种方法是在容器启动之前运行生成配置文件的脚本，然后把文件挂载到容器里。这种方法的缺点是，用户需要在宿主机上找到一个合适的地方来写入这些配置文件，每个容器可能都要有个不同的版本，然后在销毁容器时正确地清理这些文件。在容器外定制配置文件与在容器内定制相比无更多益处，但是后者会更受欢迎，因为配置文件被包含在容器内。

7.2.5 让镜像在Docker变化时对自身进行重新配置

有时，我们需要容器能感知同一宿主机上的其他容器，并向它们提供服务。例如，提供日志收集服务的容器，它会把所有其他容器的日志发送给类似Kibana的日志聚合器。其他需求可能与这类容器的监控有关。

这类容器需要访问宿主机的Docker进程，以便与它进行通信，并查询现有容器及其配置。

在讨论如何实现这类容器之前，让我们使用一个日志示例来讨论这些容器可以解决的问题类型：我们希望将来自所有容器的日志发送给某些日志聚合服务，而且我们希望可以在一个容器里完成这件事。这要求我们运行一个日志收集进程，如logstash或fluentd，并且配置它为可以从每个容器获取日志。Docker通常将每个容器的日志存储在它自己的目录中，遵循这个模

式：`/var/log/docker/containers/${CONTAINER_ID}/${CONTAINER_ID}.json.log`——这里是json日志。

一个解决方案是构建我们自己的日志收集器，理解这个布局，并可以与Docker通信来查询现有容器的情况。

不过，多数时候用户不想编写自己的日志收集服务，而是选择配置一个现有的。这意味着当宿主机上添加或删除容器时，这个日志收集器的配置会发生改变。幸运的是，已经有一些工具可以根据来自宿主机的

Docker服务器的信息来重新构建配置文件。

这类工具之一是[docker-gen](#)。这个工具在Docker提供的容器信息基础上，使用提供的模板来生成配置文件。它所提供的模板语言对多数任务来说都足够强大，它运作的方式是它会监视或轮询Docker进程以获取容器内的变化（添加、删除等），并在发生变化时从模板重新生成配置文件。

在我们示例中，我们希望重新生成日志收集器配置以便所有容器的日志可以被正确地解析、标记并发送给我们所使用的日志聚合器。

让我们以一个现实世界的例子来看看这是如何工作的，该示例使用fluentd作为日志收集器，使用ElasticSearch/Kibana作为日志聚合器。

首先，我们需要创建我们的日志收集器容器：

```
FROM phusion/baseimage

# 设置正确的环境变量
ENV HOME /root

# 使用baseimage-docker的init系统
CMD ["/sbin/my_init"]

RUN apt-get update && apt-get -y upgrade \
    && apt-get install -y curl build-essential ruby ruby-dev wget \
    libcurl4-openssl-dev \
    && gem install fluentd --no-ri --no-rdoc \
    && gem install fluent-plugin-elasticsearch --no-ri --no-rdoc \
    && gem install fluent-plugin-record-reformer --no-ri --no-rdoc

ADD . /app
WORKDIR /app
RUN wget https://github.com/jwilder/docker-gen/releases/download/0.3.6/ \
    docker-gen-linux-amd64-0.3.6.tar.gz \
    && tar xvfz docker-gen-linux-amd64-0.3.6.tar.gz \
    && mkdir /etc/service/dockergen

ADD fluentd.sh /etc/service/fluentd/run
ADD dockergen.sh /etc/service/dockergen/run
```

这个Dockerfile的相关部分是我们安装了fluentd，然后为fluentd安装了一些插件。第一个用于将日志发送给ElasticSearch，另一个是record-

reformer，用于在发送日志给ElasticSearch之前对其进行转换和标记。最后，我们安装了**docker-gen**。

由于需要在同一个容器内同时运行**docker-gen**和**fluentd**，所以我们需要某种服务管理程序。在这个例子中，我们的镜像是基于**phusion/baseimage**的，这是Ubuntu一个Docker化的精简版本。相比常规Ubuntu，这个镜像提供的自定义项之一是使用**runit**作为进程管理程序。Dockerfile的最后两行中，有两个**ADD**指令用于添加**docker-gen**和**fluentd**的运行脚本。一旦容器启动，这两个脚本将运行，从而运行并监管**docker-gen**和**fluentd**。

docker-gen的启动脚本使用以下设置来启动**docker-gen**：它将监视Docker宿主机所运行容器的变化，如果出现任何变化，它将从模板**/app/templates/fluentd.conf.tpl**重新生成**/etc/fluent.conf**文件。一旦完成，它将运行**sv force-restart fluentd**（通过**runit**）强制重启**fluentd**，这将导致**fluentd**重载新配置。这是**docker-gen**的启动文件：

```
#!/bin/sh

exec /app/docker-gen \
    -watch \
    -notify "sv force-restart fluentd" \
    /app/templates/fluentd.conf.tpl \
    /etc/fluent.conf
```

fluentd的启动脚本更为直接一些，它只是使用**docker-gen**生成的配置文件**/etc/fluent.conf**来启动**fluentd**：

```
#!/bin/sh

exec /usr/local/bin/fluentd -c /etc/fluent.conf -v
```

接下来我们所需的是**docker-gen**用于生成FluentD配置的模板文件。这是现实世界中的一个详细示例：

```
## 文件输入
## 读取标签为docker.container的日志

{{range $key, $value := .}}
```

```

<source>
  type tail
  format json
  time_key time
  time_format %Y-%m-%dT%T.%LZ
  path /var/lib/docker/containers/{{ $value.ID }}/{{ $value.ID }}-json.log
  pos_file /var/lib/docker/containers/{{ $value.ID }}/{{ $value.ID }}-
json.log.pos
  tag docker.container.{{ $value.Name }}
  rotate_wait 5
  read_from_head true
</source>
{{end}}

{{range $key, $value := .}}
<match docker.container.{{ $value.Name }}>
  type record_reformer
  renew_record false
  enable_ruby false
  tag ps.{{ $value.Name }}
  <record>
    hostname {{ $.Env.HOSTNAME }}
    cluster_id {{ $.Env.CLUSTER_ID }}
    container_name {{ $value.Name }}
    image_name {{ $value.Image.Repository }}
    image_tag {{ $value.Image.Tag }}
  </record>
</match>
{{end}}

{{range $key, $value := .}}
<match ps.{{ $value.Name }}>
  type elasticsearch
  host {{ $.Env.ELASTIC_SEARCH_HOST }}
  port {{ $.Env.ELASTIC_SEARCH_PORT }}
  index_name fluentd type_name {{ $value.Name }}
  logstash_format true
  buffer_type memory
  flush_interval 3
  retry_limit 17
  retry_wait 1.0
  num_threads 1
</match>
{{end}}

```

这里内容较多，不过主要看一下模板的关键部分。我们为每个容器生成了3个条目：一个**source**类型，两个**match**类型。在这个例子中，我们

生成这些条目的方法是：对每个容器进行迭代（`{{range ...}}`）并为其构建相应条目（如`<source ...> ... </source>`）。在一个`range`块内，我们可以使用`$value`变量来读取当前容器的数据，这个变量是一个包含Docker有关该容器所有信息的字典。

例如，在`source`条目中，我们告诉fluentd上哪儿去查找每个容器的日志文件。所有容器的日志都位于`/var/lib/docker/containers`下以该容器ID命名的目录中，其文件名也是以容器ID命名。通过这句来完成：

```
path /var/lib/docker/containers/{{ $value.ID }}/{{ $value.ID }}-json.log
```

我们还为来自这个源的日志打上容器名标签，这在之后过滤时将非常有用：

```
tag docker.container.{{ $value.Name }}
```

模板里其他条目遵循相同的结构。第一个`match`条目用来重写日志条目并为其添加额外信息，如集群名称和宿主机名称。这两个值都来自于环境变量：

```
hostname {{ $.Env.HOSTNAME }}  
cluster_id {{ $.Env.CLUSTER_ID }}
```

我们还添加了来自Docker的其他有用的信息：容器所运行的镜像名称以及镜像标签。这样，我们可以在之后使用镜像名称甚至是镜像版本来过滤日志。在相同镜像的不同版本显示出不同的错误率时，这将非常有帮助，我相信读者也同意这个说法。这是添加了这些标签的代码：

```
image_name {{ $value.Image.Repository }}  
image_tag {{ $value.Image.Tag }}
```

最后一个`match`条目会将日志条目发送给ElasticSearch，而它的地址和端口也是来自于环境变量。

现在，使用这个设定，每次一个新容器被创建或被销毁，会使用上述3个条目为每个容器重新生成`/etc/fluent.conf`文件。这样我们从所有容器获取并发送给ElasticSearch的日志就被正确打上时间戳和标记。

7.2.6 信任与镜像

在使用Docker镜像时，一个共同的、备受关注的问题是它们到底有多可信。Docker和其他容器提供商正致力于为所下载、运行的镜像提供一个高层次的信任。这种信任来自两个层面。一个是镜像本身是否值得信任，镜像是否由受信任的开发者编写，如Docker公司或Red Hat。另一个是保证所下载的镜像确实是自己想要下载的镜像。截至本书编写时，Docker尚未提供端到端的信任链，不过已经有一部分功能存在。

要保护自己，避免运行包含恶意软件或其他风险的镜像，目前我们最好的选择是自己构建镜像。多数现存镜像都是开放源码的，并通过Dockerfile来生成。与下载镜像相反，复制这个Dockerfile，对其进行检查，然后从该Dockerfile构建镜像。要完全确保镜像不具有危害，用户需要检查所有镜像层，包含基础操作系统层。也就是说，如果一个镜像是基于另一个镜像，而后者又是基于一个知名的操作系统镜像，那么用户需要验证并构建前两层。

7.2.7 让镜像不可变

尽管容器文件系统是可写的，但最好还是将它们视为只读，并且只在启动时段进行写入，如需要在这段时间生成配置文件。将容器文件系统视为只读的主要原因是这些文件系统比宿主机的文件系统要慢，也因为在容器被销毁时数据极易丢失。很显然，如果一个容器运行着一个数据库，用户需要在某些地方写入数据。这种情况下，用户可以使用容器自身的文件系统，或写入宿主机挂载的数据卷中。

不过，多数容器可能完全不需要写入文件系统，因为它们不保存数据。多数情况下，进程还是会写入文件系统以生成日志。

在容器从业者中一个通用的模式是将日志写入进程的标准输出中，而非写入文件系统。这样，用户就依赖Docker自己的日志收集器来提取那些日志，不再需要对容器的文件系统进行写入。然后，用户在每台宿主机上运行一个日志收集器进程来提取这些Docker生成的日志，并将其发送给一个中央日志服务器进行存档、分析与查询。在运行微服务架构时，不同容器的数量非常巨大并且是动态的，这个模式十分普遍。

在为第三方服务构建镜像时，有时我们无法让服务将日志输出到标准输

出中。例如，多数Web服务器就不这么做。不过，也有一个相对简单的方法可以实现相同的行为：将日志文件链接到标准输出中。

例如，**nginx**只会把日志写到文件系统的日志文件中。在这种情况下，我们会指示**Nginx**继续这么做：

```
access_log /var/log/nginx/access.log main;
```

不过在Dockerfile中，我们将**/dev/stdout**链接到该文件中，因此当**Nginx**写入**access.log**时，它反而是写入到容器的标准输出：

```
RUN ln -sf /dev/stdout /var/log/nginx/access.log
```

7.3 小结

理解Docker对叠加文件系统的使用，以及如何构建轻便、可配置、可重用且迎合生态系统的镜像，能够为构建高效的Docker基础设施提供良好的基础。如我们所展示的，有时我们需要考虑不同配置的范式和运行时来进行软件设计，并以此打造对Docker友好的镜像，不过这些额外的付出从长远看是值得的，相比它为我们节省的时间和免去的麻烦要小得多。

Docker基础设施的重要一环是镜像仓库。下一章将详细讲述这个话题。

第8章 存储Docker镜像

如果读者已经在开发环境或生产环境中使用过Docker就会知道，使用Docker镜像可以很简单地完成存储镜像的任务。Docker从创建的那天起就有一个中央仓库可以让用户非常方便地存储镜像。这个中央模式简化了镜像的推送和拉取，同时方便工程师将代码和服务变得高度可移植。存储Docker镜像有3种方式：公共的、私有的、保存/载入。每个选项都有其优缺点，因此选用何种方式取决于所处环境类型、哪个最管用以及公司的安全性要求。

对于开源或公共项目，建议使用公共仓库。如果需要更高的安全性和更佳的性能，建议使用私有registry。如果需要定制某些东西，可以使用保存/载入。企业还应该考虑所存储镜像的数量和大小。在存储器中镜像大小通常有几百兆，因此需要确保在配备新容器时仓库具有高性能。

在使用Docker拉取或推送镜像时，除非指定了本地仓库，否则默认是Docker Hub。例如，以下命令会将镜像直接推送到Docker Hub上。

```
- docker push redis
```

如果在镜像名称之前指定命名空间，就可以重定向到一个内部私有仓库。例如，如果repo.domain.com托管在内部，以下命令将推送到一个本地仓库中。

```
- docker push repo.domain.com/redis
```

让我们来详细了解一下存储镜像的各种方式。

8.1 启动并运行存储的Docker镜像

对于开发环境和公共用途而言，存储Docker镜像最常见的方法是使用默认仓库hub.docker.com（Docker Hub）。可以将Docker Hub当作Docker镜像的GitHub。它提供了一个查看不同类型镜像的极佳途径，可以关注、收藏最好镜像，查看Dockerfile的内容，甚至是查看对该镜像用途的描述和评论。使用Docker Hub启动和运行非常简单。只需要创建一个账号，就能获得一个可以立即使用的单一仓库。完成账号设置后，就可以通过这个命令将新构建的镜像推送到Docker Hub中。

```
- docker push -t newrepository/webimage
```

从Docker Hub拉取镜像也非常简单：

```
- docker pull newrepository/webimage
```

在使用公共仓库时，需要注意几个安全设置。在使用Docker Hub时，镜像可以简单地置为公共或私有状态。如果在镜像中存储了源代码、安全密钥或环境细节，那么将镜像设置为公开状态应该非常慎重。如果用户的镜像包含敏感信息，Docker Hub能通过私有化仓库为镜像进行加固（只能通过管理员或协作账号访问）。Docker Hub提供了认证服务以保护可以对用户的仓库进行修改的人，还可以为用户的仓库分配协作者。

如果对Docker Hub背后的技术感兴趣，可以看一下Jérôme Petazzoni的[演讲视频](<http://blog.heavybit.com/blog/2015/3/23/dockermeetup>)。

如果刚开始使用Docker，可尝试一些[官方镜像](#)。

8.2 自动化构建

Docker Hub提供了一个不太出名但却非常有用的功能，用户可以使用Docker Hub的服务器自动构建容器镜像。要设置自动化构建，将Docker Hub仓库指向一个包含Dockerfile的GitHub、Bitbucket仓库或仓库的一个路径。

一旦设置了自动化构建，每当修改所配置的源代码时，Docker Hub就会自动化构建新容器镜像。接着，新构建的容器将被推送到registry中，并

被标记为自动构建（`automated build`）以供下载。

自动化构建有以下优点：

- 减少用户或用户的基础设施工作量；
- 自动化构建可以使用安全补丁自动更新基础Docker库镜像；
- 事件驱动方法（通过Webhooks）——镜像反映了应用程序代码的最新版本——对于开源项目这尤其有用。

8.3 私有仓库

存储Docker镜像的另一个常见方法是使用私有registry。Docker提供了一个开源服务器来存储镜像。私有registry让企业可以安全地将镜像保存在防火墙和VPN之后，以确保代码及镜像仓库的安全。启动并运行私有registry也非常简单。这是一个入门链接：<http://docs.docker.com/docker-hub-enterprise/install/>。

在Docker刚起步时，私有registry就已经是Docker主代码的一部分。由于Docker registry是其生态系统中极其重要的一部分，他们决定将私有registry抽出来作为独立产品。Docker自此将代码移动到自己的分支上，甚至将代码转化为可下载的Docker镜像。在过去一年中，私有registry得到了更好的发展。如果读者从早期就开始接触Docker，会发现私有registry已经从原先的非常不稳定变成了现在具有多种改进的稳定状态。

目前的私有仓库已经相当稳定，很多公司因为其性能提升和更高的安全要求已经开始使用自有的内部registry。如果你刚刚上手，打算运行自己的私有registry时，应考虑几个架构决策。需要考虑网络带宽、登录凭证、SSL安全性、监控以及磁盘存储要求。

私有仓库有以下优点：

- 速度——将仓库放置在自有网络内部可加快镜像的推送与拉取；
- 安全性。

8.4 私有registry的扩展

企业刚开始使用私有registry时，将其快速安装在服务器上后就放任不管了，任其运行。很快，企业就意识到镜像会占用大量存储空间、消耗大量网络带宽，并将遇到很多Docker所使用的不同类型文件系统所带来的文件系统问题（见4.3节）。如果计划在自有网络中运行私有Docker registry，应将其作为最高等级的服务器来对待。

在运行自己的内部私有registry时，建议使用基于网络的存储，并对仓库服务器做负载均衡以达成冗余。让我们来看一个现实中生产环境的私有仓库。

这个环境具有一个负载均衡器、两个设置成自动扩展组中的仓库Web服务器，并使用S3作为后端存储。这个环境包含了仓库中所有526 751个对象。对象由存储在仓库中的镜像及标签组成。这个环境的总大小是2 678 702 030 780字节，即可使用的存储空间为2.43 TB。

这个环境能应对的突发网络流量可以达到1 Gbit/s，不过通常在300 Mbit/s左右。这有采集Web服务器两周进出的网络吞吐量的多个镜像。

扩展私有registry并不难，不过要长期使用Docker，应考虑好存储及网络吞吐量。希望读者在实现自己的环境时，能从本书中得到一些与环境相关指标的灵感。让我们来探索一些更深的领域。

阅读Docker提供的[管理员手册](#)或[部署手册](#)也大有裨益。

8.4.1 S3

我们看到运行私有registry的大部分公司都使用S3来存储自己的镜像。Docker Hub也使用S3来存储镜像。其最大的优势在于其几乎无限的存储以及管理的简易性。如果使用的是AWS基础设施，其速度也将非常快，因为流量将通过本地路由到S3服务上。因为这个配置的持久性存储器是S3，这使得用户的私有仓库服务器保持不变，也可以对registry的Web部分进行自动扩展和负载均衡。在配置私有registry的设置时，可以使用S3存储驱动程序。

8.4.2 本地存储

本地存储是运行私有registry的另一个常用方法。如果用户更倾向这个选项或没有亚马逊账号，可以将registry文件存储在本地挂载点上，不过我们建议使用基于NFS或NAS的挂载点，以便在持续存储新镜像时能扩展读、写及容量。使用基于网络的存储也允许用户对registry的Web部分进行扩展。容量要求可能很快就会失控，因此应确保做相应计划。

8.4.3 对registry进行负载均衡

由于网络带宽的要求，单一的Docker仓库服务器可能逐渐不满足要求。Docker非常聪明，决定在私有registry中提供一个可插拔的存储驱动程序架构。可以在一个基于网络的可扩展文件存储器（如S3）中存储镜像，因此可以对仓库的Web部分做负载均衡。企业可以将registry镜像放置在用户选定的负载均衡器之后。将仓库放置在Web负载均衡器之后，可以轻松地扩展网络带宽，并减少单一仓库服务器失败带来的单点故障。

8.5 维护

随着时间推移，用户可能不再需要使用旧的镜像及标签。目前Docker仓库还没有自动清除的功能，因此可操作的最佳实践是，定期清理不再使用的标签和镜像。镜像或标签的删除无法通过[Docker API](#)进行（截至1.6版本），因此需要使用SSH登录到机器上，然后通过Docker CLI命令清除旧的镜像。

还需要注意的是，Docker registry是个非常活跃的项目，因此需要留意其升级和新功能。请查阅最新文档以使用正确的方法升级Docker仓库。

8.6 对私有仓库进行加固

Docker私有registry的网络加固很简单，因为用户只需开放5000端口。不过，由于registry是个标准的Web服务器，将其重新配置到80或443端口

上很容易。建议遵循公司的最佳实践来配置防火墙，阻止非必需端口的访问。

8.6.1 SSL

用户可以使用SSL证书来保护镜像，使其免于受到中间人攻击。使用SSL证书加固registry传输有多种方法。用户可以使用内置的Nginx服务器、配置TLS，或将证书部署在负载均衡器上。例如，如果用户使用AWS，可以在弹性负载均衡器（ELB）上配置SSL证书，然后以http的方式传输到registry的Web服务器上。在Docker registry上安装SSL证书十分简单。这个链接可以带读者入门：<https://docs.docker.com/registry/deploying/>。

8.6.2 认证

认证对于保护registry非常重要，可防止非法或不安全的镜像被上传。它同样可保护源代码的知识产权或镜像所提供的信息。对于高保密性环境，这是需要考虑的一个重要因素。目前私有registry提供了两种认证方式：silly和令牌（token）。^[1]

基于令牌的认证方式是唯一可供选择的安全选项。它是一个具有高度安全性的成熟范例，目前有很多公司正在使用。silly认证正如其名所暗示的那样不安全。它只在HTTP请求中检查是否存在Authorization头。如果未提供头信息，它依然需要认证。

8.7 保存/载入

存储镜像还有另外一种方法，使用Docker在1.0版本之前就已经实现的保存/载入功能。因为早期私有registry不稳定，有些公司采用了Dogestry模式并一直使用它。如今，这是移动镜像时最不受欢迎的方法，不过如果它适合用户的环境，当然也是可以使用的。用户可以通过Docker的内置命令使用其保存/载入功能。例如，可以这么做：

- `docker build redis`

- `docker save redis > /tmp/redis_docker_save.tar`
- 将镜像复制到远程服务器中（或在本地使用）
- `docker load < /tmp/redis_docker_save.tar`

通过使用`save/load`命令，用户可以拥有足够的灵活度。可以将镜像保存为一个`tar`包，并上传到仓库或网络共享中，以便集中使用。需要注意的是，用户可以使用Docker的`export`命令。它跟`save`相比有轻微区别。`export`命令会合并镜像，这意味着将丢失历史和元数据。它可以将镜像体积变得更小。在移动镜像使用保存/载入方法时，应牢记这一点。

8.8 最大限度地减小镜像体积

由于用于构建镜像的依赖项不同，Docker镜像体积可能会变得很大。例如，用户使用Ubuntu作为基础镜像，然后使用`apt-get`更新了任何一些库文件，同时安装了一个软件包，如`nginx`。`apt-get`将安装一堆容器构建完成后不需要的缓存库文件和依赖项。一个通用模式是删除这些缓存文件和库以尽量减小镜像。如果用户发现自己需要最小化Docker镜像，那可以使用一个名为`docker-squash`的优秀社区项目。以下是如何使用它的一个简单示例。

```
- docker save <镜像id> | sudo docker-squash -t newtag | docker load
```

我们对mesosphere/marathon里的一个公共镜像运行`docker-squash`，可以将其大小缩小11%。拉取下来的镜像初始大小是831.7 MB。在对该镜像运行`docker-squash`后，我们创建了一个大小是736.2 MB的新镜像。随着时间的累加空间性能逐步提升，同时节省了网络宽带，这将改善仓库的存储和性能。

如果想了解更多关于镜像压缩的信息，推荐看一下这篇博客文章：
<https://blog.jtlebi.fr/2015/04/25/how-i-shrunk-a-docker-image-by-98-8-featuring-fanotify/>。

8.9 其他镜像仓库方案

随着Docker生态系统的不断成长，在探索运行Docker的新环境时，读者应留意一下其他的镜像仓库：

- [Artifactory](#)；
- [Quay](#)；
- 由New Relic更新的[Dogestry](#)；
- [Google](#)容器仓库；
- Azure上的[Docker](#)。

在第9章中，我们将讲述如何结合Docker镜像使用CI/CD系统。

[1] 在2.1.0版本中增加了htpasswd认证方式。——译者注

第9章 CI/CD

现在，读者对容器的构建与存储已经比较熟悉，本章我们简要地谈一下结合Docker镜像的CI/CD系统的使用。如今很多企业已经采用了DevOps做法，并使用一个类似Jenkins、TravisCI或Teamcity的自动化构建系统来自动构建代码。当代码自动构建完毕，在Docker构建流程中将代码添加到容器里是很简单的。如果想在生产环境中运行Docker，强烈建议使用一个自动化持续集成和持续部署（CI/CD）构建系统来构建镜像。Docker构建和推送的简易性，再与CI/CD系统相结合，可能是迄今为止最强大的DevOps服务部署方式之一。

Docker本质上是一个应用程序交付框架。即便Docker网站的座右铭是：构建、交付与运行。如果能把代码交付过程中的构建和交付部分自动化，那么很可能可以更快地将产品交付给客户。开发人员都期望能迅速且频繁地交付代码，而企业交付的是产品。Docker允许将整个产品打包在一个容器中，这不仅能迅速且频繁地交付代码，也能交付整个产品。在接下来的几年时间，我们将看到越来越多的产品以容器镜像的方式进行交付，而不是一个个下载好的msi、jar或zip文件。如果你是一家需要通过SaaS更快地交付代码给客户的公司，或正在以容器方式交付产品，你就应该开始使用构建系统来构建Docker镜像。

如果你已经使用构建系统构建过代码和产品，下一步是通过构建系统来构建Docker镜像。展开想象，一个Web容器，它运行着Jetty，并且使用Java代码的jar文件运行它的应用程序。CI/CD系统将构建代码、将其打包，然后将最终的jar文件部署在像Artifactory、文件系统、AWS S3这样的产出位置上，或保存在自己的内部系统中。此时，当构建一个Docker镜像时，只需要在Dockerfile中使用`ADD code.jar /jetty/bin/code.jar`添加jar文件。在容器运行时，只需要配置Jetty使用`/jetty/bin/code.jar`这个JAR以载入应用程序中。[Rally Soft](#)有一个的具体示例。

CI/CD系统不仅对代码的构建和打包非常有用，在构建Docker镜像时表

现也非常完美。编译Docker镜像只需要Dockerfile（代码）和一个构建命令。一旦构建完成，用户只需要将包提交到仓库中。可以对CI/CD系统进行设置，在每次检测到GitHub提交时自动完成构建的全部过程。这将让用户的开发团队或基础设施运维团队可以在一个自动化部署系统中部署新组件、基础设施或应用程序。

使用CI/CD系统构建Docker镜像需要与Docker守护进程进行通信。一个简单的方法是在用户的构建代理上安装Docker，然后就可以构建并交付镜像。另外一种方法读者可能听说过，其术语叫Docker in Docker（DIND）。读者可以通过阅读一篇[文章](#)来了解更多信息。简单而言，DIND可以通过发送构建命令给另外一个Docker守护进程来完成构建。不论选择哪种方法，都需要一个自动化方法来构建Docker镜像并在构建进程中添加代码。

构建系统一般是按步骤进行的。我们把使用Docker的几种可能性构建做下分解。

构建Docker镜像。

- （1）从github.com拉取最新的Dockerfile代码。
- （2）运行`docker build -t repo.com/image .`来构建镜像。
- （3）使用`docker push repo.com/image`将镜像推送到仓库中。

使用代码构建Docker镜像。

- （1）从github.com拉取最新的Java代码。
- （2）使用Maven编译并测试Java代码，输出设置为code.jar。
- （3）从github.com拉取最新的Dockerfile代码（Dockerfile具有`ADD code.jar /jetty/bin/code.jar`命令）。
- （4）运行`docker build -t repo.com/image .`来构建镜像。
- （5）使用`docker push repo.com/image`将镜像推送到仓库中。

使用代码和集成测试构建Docker镜像。

- （1）从github.com拉取最新的Java代码。
- （2）使用Maven编译并测试Java代码，输出设置为code.jar。
- （3）从github.com拉取最新的Dockerfile代码（Dockerfile具有`ADD code.jar /jetty/bin/code.jar`命令）。

- (4) 运行`docker build -t repo.com/image .`来构建镜像。
- (5) 启动一个测试专用Docker基础设施。
- (6) 运行完整的端到端集成测试。
- (7) 停止该测试专用Docker基础设施。
- (8) 使用`docker push repo.com/image`将镜像推送到仓库中。

这些示例并不复杂，不过多数时候也不需要太复杂。构建和交付Docker镜像非常简单。最难的部分在于，理解Docker并让构建系统自动运行Docker命令。实现这一过程的自动化只会提高用户的工程成果。接下来，我们讨论几个有关使用自动化CI/CD系统构建Docker镜像的话题。

9.1 让所有人都进行镜像构建与推送

CI/CD系统已经可以自动化构建镜像了，并且现在生产环境中也运行着Docker。此时，用户大概会意识到自己的基础设施几乎可以运行所有丢给它的容器。为什么不让开发人员在编写新代码后构建并推送Docker镜像到生产环境中？或者，只是将其放置在构建系统里，然后自动将容器部署到生产环境中？这是理论上的完美情况，但现实并非如此。尤其是在放任所有开发人员在Docker镜像中使用任意代码构建自己的Web服务器容器，然后将镜像交给运维团队运行的时候，企业环境很快就会失控。运维团队会甩手不干、愤而离场。他们很快就会意识到情况不受控制，而且完全不清楚容器内有什么。如果他们坚守岗位，可能会开始问类似这样的问题：是否运行了Jetty？是否运行了Nginx？是否运行了Apache？Web服务器的版本是否是最新的？是否使用了最新的安全性强化最佳实践对其进行配置？里面是否有SSH？如何记录日志？是否使用了标准的日志格式？问题会源源不断地涌来。让所有人构建并推送容器是一些团队会想到的主意，但它经常会迅速地转变为一个糟糕的做法。我们强烈建议尽早选择更高的路线并着手制定标准。

以下是我们所认识的团队在生产环境中使用Docker的一些标准。

9.2 在一个构建系统中构建所有镜像

如果运维或开发团队始终从他们的工作站推送镜像，就会忽略一些信息或最佳实践。一致性是扩展环境和传播知识的关键。应仔细考虑有关构建、添加、使用及运行Docker容器的标准。在一个提供文档的中央系统中构建所有的Docker镜像。基础设施就像代码一样，只是目的是用于构建、编译和打包镜像。它也可以启动有关容器推送的目的地、指定基础镜像、代码验证等的标准化。

随着时间推移，你的安全团队将理解你所做的事情，并希望了解更多。新的安全方法让安全团队更多地参与到构建过程中。因此，应赋予安全团队权限并让他们审核构建的内容。他们还可以利用工具进行测试，找出容易受攻击的软件包、不安全的配置，甚至是一些很糟糕的做法，如在代码中包含密钥信息。

9.3 不要使用或禁止使用非标准做法

你的Docker镜像是运行Apache、Jetty、Nginx，还是三者都有？是否一个Docker镜像运行Gwizd，而另一个运行Dropwizard？如果你还没创建围绕容器的最佳工程实践，那就应该不断地去确认这些问题。在某些时候，你或你的运维团队会被叫去调试一个遗忘已久的镜像。让镜像内所使用的服务或软件包具有一致的标准，将为工程实践的成功奠定基础。如果你的工程团队倾向于运行Nginx和Jetty，那就使用这些服务来交付Web服务。如果你的团队倾向于使用Dropwizard而非Gwizd，那就使用前者的软件包。你的团队越早实现镜像内容标准化，成功的可能性就越大。

9.4 使用标准基础镜像

如果你遵循了第一个标准（见9.2节）和第二个标准（见9.3节），你就会意识到使用一个默认的基础Docker镜像是个好主意。假定你的团队使用Python 2.6作为代码语言。如果团队里一个新的开发人员拉取了最新版的Python基础镜像，但它是3.0版本，那你注定要遇到问题。树立一个规范，让所有镜像继承于一个标准基础镜像，将大有帮助。有些团队已

经开始让运维团队为他们构建基础镜像，他们只要继承即可。例如，运维团队创建了一个基础Ubuntu镜像，对日志做了适当的配置、设置了正确的安全性并安装了正确的Python 2.6版本。开发团队所要做的唯一事情就是用**FROM**来使用这个基础镜像，然后在构建时用**ADD**将代码添加到镜像中，就像这样：

```
FROM company.com/python_base:2015_02
```

```
ADD code.py /code.py
```

```
CMD [ "python", "./code.py" ]
```

这大大简化了镜像的创建过程，将大量的细节交给拥有并维护这个基础镜像的人负责。提供一个标准基础镜像将让运维团队清楚容器内运行着什么（在多数情况下），它简化了构建过程，让开发人员可以专注于代码，同时，因为它创建了一致性，有助于在用户的基础设施上扩展Docker镜像。

9.5 使用Docker进行集成测试

Docker不仅可以让用户将应用程序打包成一个镜像并进行交付，还可丰富基础设施的方方面面。假定你在生产环境构建中运行集成测试，同时具有一个携带静态数据的静态环境。很可能你会构建代码，然后在这个静态环境中运行集成测试。在使用Docker的世界里，用户可以把这个静态环境转变成一个在测试基础设施里集成Docker镜像的动态环境。

Docker可以非常快速地启动与停止，加上制作优质镜像的简易性，可根据需要对测试基础设施进行支撑。在完成一次构建时，用户可以运行同一个代理或另一台宿主机系统（调用Docker API）上的集成基础设施，然后在结束后将其关闭。这甚至可以为公司省下保持静态基础设施运行的费用。

到今天为止，在结合Docker的CI/CD测试领域我们还没看到过多创新，不过已经有一些新项目开始涌现。我们见识过在镜像中运行Selenium浏览器测试，甚至为企业提供完整的端到端测试，但是我们还没看到任何标准。我们期待在这个领域很快能看到更多成果。如果想了解Docker和Selenium，可以查看名为Docker Selenium的GitHub项目。

9.6 小结

DevOps完全就是一种“以一个团队的身份交付代码和基础设施”的文化。当你的工程团队实践DevOps，并能使用单一系统来配置和部署Docker镜像到基础设施上时，你已经开始渐入佳境了。最近在DockerCon 15大会上，我们看到Jenkins全面拥抱了Docker，甚至是微软Visual Studio团队也展示了使用Docker镜像构建和部署应用程序和基础设施的威力。我们希望看到越来越多的公司开始在他们的CI/CD环境里使用Docker，并树立更多最佳实践。

配置管理对Docker来说很重要，第10章将对其进行说明。

第10章 配置管理

过去的10年里，配置管理（Configuration Management，CM）已然成为基础设施工具链中一个被广泛应用的工具，像Chef、Puppet和CFEngine这样的配置管理平台也成为了绝大多数服务器上的必备软件。伴随着企业的基础设施变得更加动态地去适配不同的像亚马逊或者Rackspace这些厂商提供的云服务，配置管理工具的地位也显得越加重要。一般来说，配置管理工具的目标是完成新上线服务器的配置和现有线上服务配置的更新等任务的标准化和自动化。配置管理的应用场景小到增加一个新用户，大到新建一个复杂的运行大数据服务的计算机集群。这些平台需要处理种类繁多的操作系统以及不同的系统版本、应用服务和各式各样的用例。随着时间的推移，这些配置管理工具在数据中心的地位变得越来越重要，基于它们实现的自动化配置任务也变得越来越复杂。从某种程度上来讲，容器技术的一大魅力正是在于它承诺可以通过面向容器的管理来避免配置管理带来的复杂性。

10.1 配置管理与容器

相对于物理机或者虚拟机（virtual machine，VM），容器化基础设施的配置管理则要简单得多。当尝试以配置变更发生频率的角度去看待配置管理本身的时候，不难发现这里面实际上存在着3种不同层级的配置变更，分别对应着不同等级的熵^[1]。

最上面的一层是作为系统一部分的宿主机的数量，和宿主机的容量、配置以及它们之间的相互关系等。这类配置通常很少会发生变更，除非有硬件/虚拟机故障等突发情况，其余最常见的变更就是替换故障元件这样的日常事宜。

第二层便是宿主机本身的配置。这包括安装软件包、打补丁和编写配置

文件。这类配置变更通常要比上一层频繁许多。

最后，更加常见的一层应该是运行在宿主机上的应用相关的配置变更。这里面包括bug的修复、性能调优、新功能的发布、新版本的迭代以及新的软件配置等。一般来说，现代基础设施绝大部分的配置变化均集中于此。

容器技术的应用会使得这3个层级之间变更频率的差异变得更大。在这种情况下，运行容器的所有宿主机看上去并没有什么变化，而且它们实际上也很少变动，反而是运行在每台宿主机上的容器会经常发生变化。

由于配置管理擅长的是装配宿主机，因而它们在新的容器化的世界里并没有多少用武之地。它们的作用也从负责配置整个系统转变成只是负责配置运行这些应用服务的基础设施，这包括从Docker宿主机的配置到Mesos集群的搭建等。

组建容器化的基础设施实际上是一项非常固定的、通用和重复的工作，以至于用户不得不重新考虑采用传统配置管理工具去做这些事情是否真的值得。毕竟这些配置管理工具本身是相当复杂的，而伴随着这种复杂性而来的是配置管理工具本身的学习成本、运维成本以及企业内部个性化定制的开销，如今因为Docker的出现，它们当初设计时的目标也已经远远超出有实际需求。

此外，由于绝大部分所需的配置变更均转移到了Docker容器这一层，传统的配置管理也变得没有那么关键。如果基础设施层面没有发生变更，用户便无需再借助配置管理工具来推送这些服务的新版本或者新配置。

10.2 面向容器的配置管理

尽管配置管理在容器环境下已经变得不那么核心，但它仍然是十分重要的，而更关键的问题在于用户很有可能无法工作在一个全容器的环境里，这便导致用户仍然需要配置管理，因此将容器整合进配置管理就变得十分有必要。

配置管理在以下3个方面能帮到Docker用户。

(1) 配置和维护Docker宿主机。这包括从带有基础操作系统的新硬件的上线到Docker服务的安装和配置，以及确保这些宿主机上安装了最新的安全补丁。这样一来就使得用户能够快速配置新主机来扩容，同时以集中式的管理方式维护Docker主机的容器集群。

(2) Docker容器和Docker镜像的管理。这涵盖了从容器镜像的整个生命周期（镜像的创建、推送等）到实际运行这些镜像的容器的运行和管理。

(3) 构建镜像。虽然Docker提供了通过Dockerfile来构建镜像的简单方式，但是很多时候在容器里运行的软件的大部分还是需要通过配置管理指令集来安装和配置的。用户可以绕过Dockerfile，直接使用在虚拟机上安装这些软件的方式在Docker容器上安装这些软件。

在配置管理工具提供的3个功能中，最为Docker用户所熟知的是第一个——配置Docker宿主机。对事先没有建立一套完备的配置管理的用户而言，另外两个也许并没有多大的吸引力，毕竟它们跟Docker原生提供的相差无几。

在本节中，我们将探讨如何使用一些主流的配置管理系统来集成Docker的支持。

10.2.1 Chef

Chef官方为我们提供了专门的[docker-chef cookbook](#)来完成Docker的宿主机安装和镜像及容器的管理。该cookbook对于不同的Docker宿主机配置参数（存储引擎和守护进程的启动参数等）以及各种宿主机操作系统的支持已经相当完备。

使用Chef在宿主机上安装Docker是一件再轻松不过的事情，只需要在你的cookbook里增加下面这行代码：

```
include_recipe 'docker'
```

基于Chef的镜像和容器的管理同样简洁明了，还可以非常简便地映射到对应的docker命令。例如，拉取一个镜像可以通过如下的资源轻松搞定：

```
docker_image 'nginx'
```

上述指令将完成最新Nginx镜像的下载。如果想拉取一个指定版本的镜像，可以这样做：

```
docker_image 'nginx' do
  tag '1.9.1'
end
```

使用Chef从镜像实例化容器同样也不是一件难事。例如，以下命令即实现了发起一个运行前面的nginx镜像的容器实例，开放对应80端口的访问权限并且挂载一个宿主机本地目录到对应容器内部的/WWW：

```
docker_container 'nginx' do
  detach true
  port '80:80'
  volume '/mnt/docker:/www'
end
```

最后，还可以使用与原生Docker完全相同的方式来操作容器。如下指令即等同于运行一个commit命令来将现有的容器提交为一个新镜像，并且命名为my-company/nginx:my-new-version：

```
docker_container 'nginx' do
  repository 'my-company'
  tag 'my-new-version'
  action :commit
end
```

不只是上述提到的这些，实际上，docker-chef cookbook还提供了更多对Docker功能方面的支持，它们分别对应着现有Docker的一些原生功能，像push、cp和export等，而一般它们的设计主旨也和原生Docker的基本保持一致，甚至于说它们可以非常工整地对应到原生Docker提供的各种功能。

Chef还支持通过标准的配置基础设施的方式来构建镜像。它们所提供的就是[Chef容器](#)，一个安装了Chef客户端并且将镜像本身配置运行在一个以runit作为进程管理程序的完整操作系统里的Docker镜像。运行这个镜像的容器将会连接到Chef服务器上并根据事先定义好的cookbook来完成自身的配置。这样一来便提供了一个很好的从虚拟机/物理机到Docker的过渡，它允许用户使用一套相同的cookbook来配置不同的虚拟

机、容器或是物理宿主机。

Chef社区还有很多其他的cookbook用来完成Docker生态圈的其他组件的自动化配置，包括服务发现（`etcd`、`consul`等）、调度器和像Mesos及Kubernetes这样的资源管理组件。Docker官方还提供了一个Ruby版本的`docker-api`，这使得Chef的recipe可以通过它的API直接与Docker守护进程交互。

综上所述，如果你已经是Chef用户，你也许会想探索一下Chef究竟能为容器做些什么。Chef可以提供的是一个简单直接的从现有基础设施到容器的迁移方案，而一旦进入Docker世界，如何使用Chef则很大程度上取决于用户所处的环境（例如，用户所在的企业有许多其他的用户），以及用户接受Docker哲学的程度，Docker的理念正是推崇从虚拟化迁移到单进程容器为主的微服务架构。

10.2.2 Ansible

与Chef类似，Ansible同样为Docker提供了从宿主机配置到镜像和容器管理的支持。

Ansible本身在配置管理方面和Docker十分契合，它们都推崇简单、直接的做事方式。因此，当Chef和Puppet纷纷选择“客户端运行在主机（及容器）上并主动向服务器拉取自己所需的配置变更”这样一套主从架构（当然，它们也正在尝试支持独立客户端）时，Ansible采取了一种更为直接的方式，即通过SSH将配置从远程主机推送到目标机器上。坦白讲，这种方法和之前的客户端模型相比，与Docker的契合度更高。

使用Ansible在宿主机上安装Docker同样是一件非常简单的事情。尽管Ansible官方不直接提供专用的playbook，但读者可以参考Paul Durivage发布在GitHub中的用来在Ubuntu上安装Docker的`docker.ubuntu`的做法。例如，添加如下指令到你的Ansible playbook，即可实现在宿主机上安装Docker并且监听7890端口：

```
- name: Install Docker on Port
  hosts: all
  roles:
    - role: angstwad.docker_ubuntu
      docker_opts: "-H tcp://0.0.0.0:7890"
```

```
kernel_pkg_state: present
```

Ansible同样提供了对Docker宿主机管理的官方支持——[Docker模块](#)。通过调用这个模块，可以实现对Docker镜像的管理以及对容器的创建、启动、停止和销毁，这同直接使用Docker几乎没什么两样。当涉及容器方面的操作时，用户也可以设置一些重启策略，这样一来，在容器发生故障时，Ansible便知道该如何去应对。Ansible的Docker模块在多容器管理方面同样提供了不错的支持。例如，用户可以轻松定位到所有运行同一镜像的容器然后通过Ansible来完成一键重启。

用户也可以直接在自己的playbook里编写对应的Docker操作。例如，用户可以在自己的playbook里追加如下指令，如此一来，Ansible便会先下载好myimage:1.2.3镜像，然后再创建一个名为mycontainer的容器，接着它会在/usr/data里挂载一个卷，并运行这个下载好的镜像：

```
- name: data container
  docker:
    name: mycontainer
    image: myimage:1.2.3
    state: present
    volumes:
      - /usr/data
    command: myservice --myparam myvalue
    state: started
    expose:
      - 1234
```

这里面其他参数的含义是显而易见的。容器将会在启动时运行一个myservice的命令，配上对应的运行参数--myparam myvalue。然后该容器还会对外公开1234端口。

用户也可以重启一台主机上现有运行同一镜像的容器。例如，可以通过下面这些指令去重启所有运行myimage:1.2.3镜像的容器实例：

```
- name: restart myimage:1.2.3
  docker:
    image: myimage:1.2.3
    state: restarted
```

另外，Ansible对于用户的Docker基础设施的另一大助益便是用户可以借

助其playbook来完成镜像的构建。为了达成这个目的，用户需要编写一个Dockerfile，它会在本地安装Ansible并且将需要运行的playbook复制到容器里然后运行：

```
FROM ubuntu
# 安装 Ansible
RUN apt-get -y update
RUN apt-get install -y python-yaml python-jinja2 git
RUN git clone http://github.com/ansible/ansible.git /usr/lib/ansible
WORKDIR /usr/lib/ansible
ENV PATH /usr/lib/ansible/bin:/sbin:/usr/sbin:/usr/bin
ENV ANSIBLE_LIBRARY /usr/lib/ansible/library
ENV PYTHONPATH /usr/lib/ansible/lib:$PYTHON_PATH

# 下载并复制 playbooks 和 hosts
ADD playbooks /usr/lib/ansible-playbooks
ADD inventory /etc/ansible/hosts
WORKDIR /usr/lib/ansible-playbooks

# 运行 playbook，用 Ansible 配置镜像
RUN ansible-playbook my-playbook.yml -c local

# 其他Docker配置
EXPOSE 22 4000
ENTRYPOINT ["myservice"]
```

在上述Dockerfile里，Ansible将去执行一个事先已经复制到镜像的指定playbook（这里是my-playbook），用户可以把所有的配置任务都放到这里面来，Ansible会负责接下来的所有事情。如果在用户的基础设施里已经有了很多现成的playbook，用户完全可以很方便地继续使用它们来配置相应的容器而无需再烦恼是否应该直接用Dockerfile来重新实现这些自动化配置任务。

Ansible官方的Docker模块同样也提供了镜像管理方面的支持，如上传镜像、拉取镜像、删除和下载镜像等。值得一提的是，上述绝大部分的功能也可以很轻松地通过Ansible直接运行Docker命令行来实现。

10.2.3 Salt Stack

Salt Stack于2014.7.0版本完成了对Docker的功能支持，这其中不仅包括常见的Docker操作，还加入了从Docker获取信息到Salt Mine的支持。

Salt通过DOCKERIO模块来管理Docker容器。有了它，用户可以很方便地定义一些自己所需的Docker State，而Salt Minion将会负责和Docker交互，并实现用户期许的配置状态。例如，如果用户想实现这样的一个配置状态：创建一个运行myorg/myimage:1.2.3镜像，名字叫做mycontainer的容器。那么可以根据所需定义如下的state：

```
my_service:
  docker.running:
    - container: mycontainer
    - image: myorg/myimage:1.2.3
    - port_bindings:
        "5000/tcp":
          HostIp: ""
          HostPort: "5000"
```

其他的Docker操作与上述类似。与Ansible类似，用户同样也可以非常简便地使用Salt Stack调用Docker的命令行来操作Docker。例如，如下内容大致等价于前面的state：

```
my_service:
  cmd.run:
    - name: docker run -p5000 --name mycontainer
      myorg/myimage:1.2.3
```

10.2.4 Puppet

用户可以使用Puppet的一个由Gareth Rushgrove提供的已然相当完备的Docker模块来完成Docker宿主机、镜像以及容器的安装和管理等工作。如果用户已经在用Puppet，那么通过这个模块来操作Docker将是一件再简单不过的事情。

使用该模块安装Docker非常简单，这一功能初步在Ubuntu 12.04和14.04以及Centos6.6和7.0测试通过，当然它也可能未经修改便可以直接在其他Debian或者RHEL派系的Linux发行版上正常运行。以下便是如何在宿主机上安装最新版Docker的详细指令：

```
include 'docker'
class { 'docker':
  version => 'latest',
}
```

该安装类提供了很多参数选项。例如，用户可以修改Docker监听的端口，或者去定义它的套接字具体的路径：

```
class { 'docker':  
  version => 'latest',  
  tcp_bind    => 'tcp://127.0.0.1:4243',  
  socket_bind => 'unix:///var/run/docker.sock',  
}
```

一旦完成了Docker的安装，管理它的镜像及容器自然也是水到渠成的事情。例如，我们可以通过如下指令来拉取所需的镜像：

```
docker::image { 'myorg/myimage':  
  image_tag => '1.2.3'  
}
```

还可以轻松地将它删除：

```
docker::image { 'myorg/myimage':  
  ensure    => 'absent',  
  image_tag => '1.2.3'  
}
```

从镜像运行一个容器也不再是一件困难的事情：

```
docker::run { 'myservice':  
  image    => 'myorg/myimage:1.2.3',  
  command => 'myservice --myparam myvalue',  
}
```

`docker::run`提供的很多配置选项都能够很好地直接对应到原生Docker `run`的选项，像公开的服务端口、环境变量、重启策略等。另外，它还补充了一些Docker原生所不具备的、额外的配置参数，像容器的依赖关系等。这些依赖关系会被编码到`initd`或者`systemd`^[2]。

最后，这个模块还提供了一个很贴心的功能，那便是用户可以直接在正在运行的容器里执行`exec`命令：

```
docker::exec { 'myservice-ls':
```

```
detach    => true,  
container => 'myservice',  
command   => 'ls',  
tty       => true,  
}
```

10.3 小结

时下的配置管理工具提供了对Docker一定层面上的支持。然而，这些功能是否足够满足需求又或者说使用这些工具所带来的价值是否抵得上它们的投入成本都取决于用户所处的具体环境。当然，这些工具也为已经使用它们来管理虚拟机的企业提供了一些过渡到容器的解决方案。

Docker的存储引擎为其提供了优质的性能体验，第11章我们将就此展开讨论。

[1] 这里的熵指的是基础设施的配置变化程度的度量。——译者注

[2] 即容器启动时便会根据这个依赖关系顺序启动。——译者注

第11章 Docker存储引擎

使用Docker的最大的好处之一在于它可以从一个现有的镜像快速的实例化一个新的容器。作为Docker的前身，历史上的LXC容器，它的做法是将会为每个新创建的容器分配宿主机上一个单独的目录，然后将镜像的根文件系统复制到该目录下。这显然是相当低效的。在这种情况下，磁盘空间的消耗会随着每一个新创建的容器而不断的增长，并且容器的启动时间也取决于从一个目录复制到另一个目录下的数据量的大小。与之不同的是，Docker利用镜像分层技术来解决这些难题。

从整体上来讲，镜像层就是一个简单的文件树结构，它可以按需挂载和更改。新的镜像层可以是全新的也可以基于现有的镜像即所谓的父镜像层之上创建。基于一个现有镜像层创建出来的新的镜像层实际上算是它的一个副本——他们两个都可以被Docker根据一个相同且唯一的名字所定位。而一旦这个新的镜像层发生更改，那么Docker将会为它立马生成和分配一个新的唯一的名字。从这一刻起，父镜像层将会保持不变，而未来对该镜像做出的任何更改都只会应用到新的这一层。如果这让你想到了著名的写时复制（CoW）机制，那么也许你会立刻恍然大悟！

Docker这一镜像分层技术的实现有赖于众多的写时复制文件系统的支持，其中有一些已经内置到了原生的Linux内核里。与之前直接复制父镜像的做法不同的是，Docker只关注父镜像及基于它所创建的新的镜像层之间的变更内容（又称为增量）。这样一来便节省了大量的磁盘空间。整个镜像文件系统主要的增长点只在于各镜像层之间的增量的大小。

Docker在容器的存储管理方面采取了类似的概念。每一个容器都分为以下两层。

- 初始层——基于父镜像的基础镜像层。它包含了每个Docker容器都会出现的一些基本文件：`/etc/hosts`、`/etc/resolv.conf`等。
- 容器文件系统层——初始层之上的镜像层。它包含容器本身存储的

一些数据。

Docker通过它所提供的远程API对外公开了它的镜像分层，这里面也提供了一些比较贴心的功能，例如，用户可以实现容器的版本控制以及可以为镜像打上标签等。如果用户想从一个现有的容器保存出一个新的镜像层，只需要简单的调用`docker commit container_id`命令，随后Docker便会自动去定位自初始层起一路到该容器层所应用的所有变更，然后在父镜像层之上创建一个新的分层。用户也可以为刚提交的这一分层打上一个标签，要么据此构建新镜像，又或者是自此实例化新的容器。由于在容器文件系统上应用了相同的写时复制概念，Docker容器的启动时间也因此大大缩短。

一图胜千言——如果读者有兴趣想挖掘一下存储在Docker Hub里的任一Docker镜像对应具体的镜像分层，可以试试由[Centurylink实验室](#)研发的一个非常棒的[Image Layers](#)工具，它使用户可以手动检查该镜像具体可用的分层，而它实际提供的功能还远不止这些。

到目前为止，上述所讲的内容已经覆盖了关于Docker如何处理镜像和容器文件系统必备的基础知识，接下来，我们将探讨所有Docker原生支持的存储引擎。我们将深入剖析里面的一些核心概念，从而让读者能够更好地理解其中的原理，并且我们还会提供一些实际的例子，这里的每条命令读者都可以在自己的Docker宿主机上一个个地直接执行。

Docker原生提供了不少开箱即用的存储引擎。用户需要做的只是选用其中之一罢了。一旦决定了要使用哪款引擎，用户就需要在环境变量`DOCKER_OPTS`里追加一个`--storage-driver`命令行参数来告知Docker守护进程。跟其他的服务一样，用户必须重启一次Docker守护进程来使得新的配置参数生效。下面，我们的探索之旅将首先从Docker默认的存储引擎[aufs](#)开始。

11.1 AUFS

就像之前所提到的那样，[aufs](#)是Docker提供的默认的存储引擎。选择它的部分原因在于Docker团队最开始在[dotCloud](#)内部便是使用它来运行的容器，因此他们对于如何在生产环境下应用它已经有了一个比较坚实的

理论基础和运维经验。

顾名思义（该引擎的正式名称尚且待定），**aufs**使用**AUFS**文件系统来存储镜像和容器。**AUFS**的工作原理是通过层层“堆叠”多个称为分支的文件系统层，然后每一层都对外公开一个单独的挂载点以使用户可以独立访问它们。每个分支都是一个简单的目录，里面包含一些普通的文件和元数据。最上层的分支则是唯一的一个可读写的文件层。**AUFS**正是靠元数据在所有堆叠的镜像层之间查找和定位文件的具体位置。它每一次的查找操作总是先从最顶层开始，而当某个文件需要做读写相关的操作时该文件也将会被复制到最顶层。一旦这个文件本身很大的时候，这类操作所耗费的时间可能就会比较长。

理论就先讲到这里。接下来，我们来看一个具体的实战例子，它将为我们展示**Docker**是如何运用**AUFS**存储引擎的。首先，先确认一下**Docker**是否的确配置了**aufs**为存储引擎：

```
# sudo docker info
Containers: 10
Images: 60
Storage Driver: aufs
  Root Dir: /var/lib/docker/aufs
  Backing Filesystem: extfs
  Dirs: 80
Execution Driver: native-0.2
Kernel Version: 3.13.0-40-generic
Operating System: Ubuntu 14.04.1 LTS
CPUs: 1
Total Memory: 490 MiB
Name: docker-hacks
ID: DK4P:GBM6:NWWP:VOWT:PNDF:A66E:B4FZ:MXA:LSNB:JLGB:TUOL:J3IH
```

正如我们所看到的，**AUFS**引擎默认的基础镜像存储目录便是**/var/lib/docker/aufs**。让我们一起来看看这个目录里包含了哪些内容：

```
# ls -l /var/lib/docker/aufs/
total 36
drwxr-xr-x 82 root root 12288 Apr  6 15:29 diff
drwxr-xr-x  2 root root 12288 Apr  6 15:29 layers
drwxr-xr-x 82 root root 12288 Apr  6 15:29 mnt
```

如果用户还没有创建任何容器，那么所有相关的目录都将是空的。顾名思义，**mnt**子目录里面包含的内容便是所有容器文件系统的挂载点。他们只会在容器运行的时候才会被挂载上。既然如此，让我们先试试创建一个新容器，然后实际来看看它里面生成的内容吧。我们将在一个新的Docker容器里运行一个**top**命令，这样一来它便会一直保持运行的状态，直到我们主动停止它：

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47
```

从上面的输出可以看到，**busybox**镜像由5个镜像层组成，它们分别对应了5个AUFS分支。AUFS分支的数据则存放在**diff**目录，用户可以很轻松的通过如下命令来验证该**diff**目录下的每个子目录对应的镜像层：

```
# ls -l /var/lib/docker/aufs/diff/
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b
ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47-init
```

读者也可以看到当我们启动该容器时，像本章开头讲述的那样，初始镜像层在最上层最先被创建出来。如今，该容器已经处于运行状态，它的文件系统也应该被挂载上，让我们一起来确认一下：

```
# grep f534838e081e /proc/mounts
/var/lib/docker/aufs/mnt/
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47
aufs rw,relatime,si=fa8a65c73692f82b 0 0
```

该运行中的容器文件系统的挂载点会被映射

到/var/lib/docker/aufs/mnt/container_id并且它会被挂载成读-写模式。我们不妨试试修改该容器的文件系统，在它里面创建一个简单的文件（/etc/test）然后将这个修改提交：

```
# docker exec -it f534838e081e touch /etc/test
# docker commit f534838e081e
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2
```

上述操作应该会创建一个新的镜像层，它将包含刚刚我们创建的新文件并且会把该增量存放到一个特定的diff目录。这一点可以很方便的通过列出diff目录下的子目录里的内容来确认：

```
# find /var/lib/docker/aufs/diff/
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2/
-type f

/var/lib/docker/aufs/diff/
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2/
etc/test
```

现在，可以基于刚创建的包含/etc/test这个文件的镜像层启动一个新容器：

```
# docker run -d
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2
top
9ce0bef93b3ac8c3d37118c0cff08ea698c66c153d78e0d8ab040edd34bc0ed9
# docker ps -q
9ce0bef93b3a
f534838e081e
```

可以通过如下命令来确认该文件是否的确存在于新创建的这个容器内：

```
# docker exec -it 9ce0bef93b3a ls -l /etc/test
-rw-r--r--    1 root    root          0 Apr  7 00:27 /etc/
test
```

那么，让我们再来看看如果删除容器里的一个文件并且提交这一更改会发生什么：

```
# docker exec -it 9ce0bef93b3a rm /etc/test
```

```
# docker commit 9ce0bef93b3a
e3b7c789792da957c4785190a5044a773c972717f6c2ba555a579ee68f4a4472
```

当删除一个文件时，AUFS会创建一个所谓的“写出”文件，基本上就是一个重命名的加上“.wh.”前缀的文件。这便是AUFS将文件标记为已删除的方式。这一点同样也非常容易验证，只需要检索对应镜像层目录里的内容：

```
ls -a /var/lib/docker/aufs/diff/
e3b7c789792da957c4785190a5044a773c972717f6c2ba555a579ee68f4a4472/
etc/
.  ..  .wh.test
```

该隐藏文件实际上仍然存放在宿主机的文件系统上，但是当用户基于这一创建的镜像层启动一个新容器时，AUFS会非常智能地将其剔除，而这个文件将不会再出现在新运行的容器的文件系统里：

```
# docker run --rm -it
e3b7c789792da957c4785190a5044a773c972717f6c2ba555a579ee68f4a4472
test -f /etc/.wh.test || echo "File does not exist"
File does not exist
```

以上便是我们介绍的aufs存储引擎的全部内容。在这里，我们一起探讨了Docker是如何利用AUFS文件系统所提供的一些功能特性来创建的容器并且展示了一些实战案例。下面我们对这一节做一个简单的总结，然后转到下一个存储引擎的介绍。

AUFS的挂载速度是相当快的，因此它们能够非常快速的创建出新容器。它们的读/写速度也几乎跟原生的差不了多少。这使得它成为众多运行容器的Docker存储引擎里一个比较合适和成熟的方案。AUFS的性能瓶颈主要在于需要写入大文件的场景，因此使用aufs存储引擎来存放数据库文件可能不是一个好主意。同样地，太多的镜像层可能会导致文件查找时间过长，因此最好不要让自己的容器有太多的分层。

虽然用户可以通过一些变通手段来解决先前所提到的一些不足之处（如用卷来挂载数据目录和减少镜像层数），但是aufs存储引擎的最大的问题还是在于AUFS文件系统本身还没有被容纳到主流的Linux内核版本里，而且以后也不太可能。这就是说它不大可能会出现在主流的Linux发行版中，而它在使用上就可能需要用到一些黑科技，这对于用户而言

实在是不太方便，并且由于它没有被内置到内核里，如何将其更新补丁应用到Linux内核也是一件令人头疼的事情。即便是曾经在他们的内核里加入了对AUFS的默认支持的Ubuntu，如今也决定了在12.04版本里[禁用](#)这一特性，并且明确鼓励用户迁移到[OverlayFS](#)，该文件系统自11.10版本起被内嵌到了Ubuntu的内核里，而且仍然在积极地迭代更新。关于OverlayFS，我们将在这一章的稍后部分详细讨论。现在，让我们一起来看看另外一个建立在相对成熟的Linux存储技术基础上的存储引擎devicemapper。

11.2 DeviceMapper

DeviceMapper是一个由Linux内核提供的先进的存储框架，它能够将物理块设备映射为虚拟块设备。它也是[LVM2](#)，块级别存储加密，[多路径](#)以及许多其他的Linux存储工具等诸多技术实现的基础。用户可以在Linux内核的[官方文档](#)里获取更多有关DeviceMapper的信息。在这一小节里，我们将把重点放在Docker是如何使用DeviceMapper来管理容器以及镜像的存储。

devicemapper存储引擎使用了DeviceMapper的预分配（thin provisioning^[1]）模块来实现镜像的分层。从整体上来说，预分配机制（也被称之为thinp）会对外提供一组原始的物理存储（块），用户可以据此创建任意大小的虚拟块设备或是虚拟磁盘。thinp技术比较神奇的一点在于直到用户实际开始将数据写到它们里面之前，这些设备不会占用任何实际的磁盘空间，也不会有任何的原始存储块会被标记为正在使用。

另外，thinp技术支持创建数据卷的快照功能。用户也可以据此创建一个现有卷的副本，而新的快照卷将不会占用任何额外的存储空间。值得再次强调的是，在用户开始写入数据之前，它将不会从存储池里申请任何额外的存储空间。

预分配技术本身使用两种块设备：

- 数据设备——用作存储池的设备，一般都很大；
- 元数据设备——用来存放已创建的卷（包括快照点）的存储块和存

储池之间的映射关系等信息。

devicemapper存储引擎的写时复制技术是基于单个块设备级别实现的，这和**aufs**引擎基于文件系统层面的实现略有不同。当**Docker**守护进程启动时，它会为之自动创建预分配机制正常工作所必需的两个块设备：

- 用作存储池的数据设备；
- 维护元数据的设备。

默认情况下，这些设备都只是一些绑定到回环设备上的**稀疏文件**。这些文件大小上一般看上去是100 GB和2 GB，但是因为它们是稀疏文件，因此实际上并不会用去宿主机上太多的磁盘空间。

一个实际的例子可能会更好地理解这些概念。首先，我们需要设置一下环境变量**DOCKER_OPTS**从而告知**Docker**守护进程采用**devicemapper**存储引擎作为默认的存储选项，然后重启服务。在服务重启完成之后，我们可以通过如下方式来验证它现在是否真的使用**devicemapper**作为存储引擎：

```
# docker info
Containers: 0
Images: 0
Storage Driver: devicemapper
  Pool Name: docker-253:1-143980-pool
  Pool Blocksize: 65.54 kB
  Backing Filesystem: extfs
  Data file: /dev/loop0
  Metadata file: /dev/loop1
  Data Space Used: 305.7 MB
  Data Space Total: 107.4 GB
  Metadata Space Used: 729.1 kB
  Metadata Space Total: 2.147 GB
  Udev Sync Supported: false
  Data loop file: /var/lib/docker/devicemapper/devicemapper/data
  Metadata loop file: /var/lib/docker/devicemapper/devicemapper/
metadata
  Library Version: 1.02.82-git (2013-10-04)
Execution Driver: native-0.2
Kernel Version: 3.13.0-40-generic
Operating System: Ubuntu 14.04.1 LTS
CPUs: 1
```

```
Total Memory: 490 MiB
Name: docker-book
ID: IZT7:TU36:TNKP:RELL:2Q2J:CA24:OK6Z:A5KZ:HP5Q:WBPG:X4UJ:WB6A
```

让我们一起来看看devicemapper

在/var/lib/docker/devicemapper/这个目录下都做了哪些动作:

```
# ls -alhs /var/lib/docker/devicemapper/devicemapper/
total 292M
4.0K drwx----- 2 root root 4.0K Apr  7 20:58 .
4.0K drwx----- 4 root root 4.0K Apr  7 20:58 ..
291M -rw----- 1 root root 100G Apr  7 20:58 data
752K -rw----- 1 root root 2.0G Apr  7 21:07 metadata
```

如上所示，Docker创建的data和metadata文件只占用了很少的磁盘空间。我们可以通过执行如下命令来确认这两个文件实际上是否真的被用作了回环设备的后端存储:

```
# lsblk
NAME                                MAJ:MIN RM  SIZE RO TYPE
MOUNTPOINT
loop0                                7:0      0   100G  0 loop
  docker-253:1-143980-pool (dm-0)    252:0      0   100G  0 dm
  docker-253:1-143980-base (dm-1)    252:1      0    10G  0 dm
loop1                                7:1      0     2G  0 loop
  docker-253:1-143980-pool (dm-0)    252:0      0   100G  0 dm
  docker-253:1-143980-base (dm-1)    252:1      0    10G  0 dm
```

除了为预分配创建必要的稀疏文件之外，Docker守护进程还会在预分配的存储池上自动创建一个包含一个空白的ext4文件系统的基础设备。所有新的镜像层都是基础设备的一个快照，这意味着每个容器和镜像都拥有一个属于它自己的块设备。这样一来，用户在任何时间点都可以为任意现有镜像或者容器创建一个新的快照点。基础设备的默认大小设置是10 GB，这也是一个容器或镜像的最大空间大小，但是由于使用了预分配机制，它们实际占用空间会小很多。用户可以很轻松地通过执行如下命令来验证基础设备的存在:

```
# dmsetup ls
docker-253:1-143980-base    (252:1)
docker-253:1-143980-pool    (252:0)
```

让我们来试试创建一个简单的容器，然后在里面执行我们在11.1节里做过的类似测试：

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
```

如果列出/var/lib/docker/devicemapper/mnt/目录下的具体内容，读者会发现这里面有一列对应每个镜像层的文件目录：

```
# ls -l /var/lib/docker/devicemapper/mnt/
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b
ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01-init
```

这些目录即是对应特定devicemapper镜像层的挂载点。除非哪个特定的镜像层被实际挂载，例如，当某个容器正在运行时，用户会发现它目录下面的内容全部是空的。同样地，用户也可以非常简单地通过检查正在运行的容器对应的挂载目录里的内容来验证这一点：

```
# docker ps -q
f5a805967279
# grep f5a805967279 /proc/mounts
/dev/mapper/docker-253:1-143980-
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01/
var/lib/docker/devicemapper/mnt/
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
ext4 rw,relatime,discard,stripe=16,data=ordered 0 0
```

现在，如果用户列出该目录下的内容，将能够看到它里面包含的容器文件系统的具体内容。而一旦用户停止了该容器，那么它的附加块设备都

将会被卸载而对应挂载点所在的目录里的内容都会被置为空：

```
# docker stop f5a805967279
f5a805967279
# ls -l /var/lib/docker/devicemapper/mnt/
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
total 0
```

这就意味着当用户使用**devicemapper**作为存储引擎的时候，将不太容易直观的得到镜像层之间的差异。

此外，当采用**devicemapper**作为Docker的默认存储引擎时，它将会使用稀疏文件作为容器和镜像的后端存储。这会对性能方面产生显著的影响。每当一个容器修改一次它的文件系统时，Docker就必须从存储池里申请一个新的存储块，这在采用稀疏文件的情况下可能需要耗费一定的时间。试想下如果同时运行数百个容器并且它们正在修改各自的文件系统时会是怎样的一个场景吧。

幸运的是，Docker允许用户通过传入特定的命令行参数**--storage-opt**来告知Docker守护进程使用真实的块设备来存储**devicemapper**的数据和元数据设备。它们是：

- 针对数据块设备的**dm.datadev**；
- 针对元数据设备的**dm.metadatadev**。

在生产环境下应用**devicemapper**引擎时请记得一定要设置成使用真实的块设备来存储数据和元数据设备。当运行了很多的容器时这一点尤其重要！

读者可以转到

<https://github.com/docker/docker/tree/master/daemon/graphdriver/devmapper>了解**devicemapper**引擎提供的所有可用的选项。

提示：如果想从**aufs**转到**devicemapper**引擎，用户必须首先通过执行**docker save**命令将所有镜像都对对应保存到一个个单独的**tar**包里，一旦**devicemapper**存储引擎被Docker守护进程启用，用户便可以轻松地使用**docker load**命令加载这些已经保存的镜像。

正如我们所看到的，**devicemapper**存储引擎为Docker提供了一个非常

有意思的容器存储的备选方案。如果用户对LVM以及它周边丰富的工具集非常熟悉，可以很轻松地使用类似的方式来管理Docker的存储。然而，当用户决定要使用这个存储引擎的时候，往往在使用上有一些需要特别注意的地方：

- 使用devicemapper引擎至少需要了解devicemapper各个子系统的一些基础运维知识；
- 更改devicemapper的任何选项都需要先停止Docker守护进程并且擦除/var/lib/docker目录下的所有内容；
- 正如我们之前所提到的那样，容器文件系统的大小默认设置为10 GB（可以在守护进程启动时通过dm.basesize参数来修改这一配置）；
- 通常很难去扩展一个已经超出其基础设备大小的运行中的容器的空间；
- 不能轻易的扩展镜像的空间大小——提交一个大过它基础设备大小的容器真的不是那么容易。

现在，我们对于如何使用devicemapper存储引擎应该有了一些不错的想法，那么是时候继续前行，接着讨论下一个可选方案了，下一个存储引擎的背后是一个在Linux社区长久以来拥有着广泛影响力（积极的和负面的兼而有之）的文件系统btrfs。

11.3 BTRFS

即便是一个普通人都能猜到，btrfs存储引擎使用的是BTRFS文件系统来实现Docker镜像层写时复制的功能。为了进一步理解btrfs存储引擎，让我们先来了解下BTRFS文件系统所具备的一些功能特性，然后通过一些实战案例来讲解如何在Docker中应用它。

btrfs是一个已经内置到Linux内核主干中很长一段时间的叠加文件系统，但是它依然没有达到一个生产环境文件系统所需的质量或者说成熟度。它设计的初衷是为了用来和Sun公司的ZFS文件系统提供的一些功能特性相抗衡。BTRFS具备的一些显著特性包括：

- 快照；

- 子卷；
- 无间断地添加或删除块设备；
- 透明压缩。

btrfs以块为单位存储数据。一个块就是简单的一段原始存储，一般大小在1 GB左右，**BTRFS**即是使用它来存放实际的数据。数据块一般都是均匀的分布到所有底层的块设备里。所以，即使物理磁盘上有存储空间，数据块仍然可能提前耗尽。当这样的情况发生时，用户唯一能做的便是重新调整自己的文件系统，这样一来它将会从空白或者接近空白的存储块里挪走数据从而释放一些磁盘空间。这一操作过程中没有任何的宕机成本。

以上便是我们需要介绍的关于**BTRFS**的一些理论基础。现在，让我们一起来看看**btrfs**存储引擎的一些实际应用案例。为了能够使用这一引擎，**Docker**要求将**/var/lib/docker**目录挂载到一个**BTRFS**文件系统中。我们就不讲解如何去做的详细步骤了——这里我们假定用户已经事先准备好了一个**BTRFS**分区并且在其上面创建好了**btrfs**引擎所需的特定目录：

```
# grep btrfs /proc/mounts
/dev/sdb1 /var/lib/docker btrfs rw,relatime,space_cache 0 0
```

现在，需要通过修改环境变量**DOCKER_OPTS**来告知**Docker**守护进程使用**btrfs**作为其存储引擎。在守护进程重启后，用户可以通过如下命令来查询当前**Docker**正在使用的存储引擎的信息：

```
# docker info
Containers: 0
Images: 0
Storage Driver: btrfs
Execution Driver: native-0.2
Kernel Version: 3.13.0-24-generic
Operating System: Ubuntu 14.04 LTS
CPUs: 1
Total Memory: 490.1 MiB
Name: docker
ID: NQJM:HHFZ:5636:VGNJ:ICQA:FK4U:6A7F:EUDC:VFQL:PJFF:MI7N:TX7L
WARNING: No swap limit support
```

用户可以通过执行如下命令来检索**BTRFS**文件系统对应的一些信息，它

可以展示该文件系统使用情况的概要：

```
# btrfs filesystem show /var/lib/docker
Label: none  uuid: 1d65647c-b920-4dc5-b2f4-de96f14fe5af
    Total devices 1 FS bytes used 14.63MiB
    devid    1 size 5.00GiB used 1.03GiB path /dev/sdb1

Btrfs v3.12

# btrfs filesystem df /var/lib/docker
Data, single: total=520.00MiB, used=14.51MiB
System, DUP: total=8.00MiB, used=16.00KiB
System, single: total=4.00MiB, used=0.00
Metadata, DUP: total=255.94MiB, used=112.00KiB
Metadata, single: total=8.00MiB, used=0.00
```

Docker还利用了BTRFS的子卷特性。用户可以在<https://lwn.net/Articles/579009/>了解更多有关子卷的知识。整体上说，子卷是一个相当复杂的概念，可以简单地认为它是一个可以通过文件系统顶层子卷访问的POSIX文件命名空间，或者它也能够以自己的方式挂载。

每一个新创建的Docker容器都会被分配一个新的BTRFS子卷，并且如果存在父镜像层，那么它会以父镜像层子卷的一个快照的形式创建。Docker镜像同样如此。我们不妨先创建一个新的容器然后再通过具体的案例深入理解这相关的概念：

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
86ab6d8602036cadb842d3a030adf2b05598ac0e178ada876da84489c7ebc612
```

用户可以很方便地通过如下命令来验证每一层是否真的对应分配了一个新的BTRFS子卷：

```
# btrfs subvolume list /var/lib/docker/  
ID 258 gen 9 top level 5 path btrfs/subvolumes/  
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158  
ID 259 gen 10 top level 5 path btrfs/subvolumes/  
df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b  
ID 260 gen 11 top level 5 path btrfs/subvolumes/  
ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2  
ID 261 gen 12 top level 5 path btrfs/subvolumes/  
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125  
ID 262 gen 13 top level 5 path btrfs/subvolumes/  
86ab6d8602036cadb842d3a030adf2b05598ac0e178ada876da84489c7ebc612-init  
ID 263 gen 14 top level 5 path btrfs/subvolumes/  
86ab6d8602036cadb842d3a030adf2b05598ac0e178ada876da84489c7ebc612
```

用户可以在任何时间为任意一层镜像设置快照点。一个快照点等同于一个新的子卷，它和其他的子卷共享它的数据（和元数据）。由于一个快照点便是一个子卷，因此还可以创建快照的快照。一个实际的案例也许可以帮助我们理解的更加透彻。我们将对一个正在运行中的容器做变更然后提交它：

```
# docker exec -it 86ab6d860203 touch /etc/testfile  
# docker commit 86ab6d860203  
32cb186de0d0890c807873a3126e797964c0117ce814204bcbf7dc143c812a33
```

如预期那样，被提交的容器

在/var/lib/docker/btrfs/subvolumes/32cb186de0d0890c807873目录里创建了一个新的BTRFS子卷。如果进一步查看这个子卷里的具体内容，会发现它实际上包含了一个完整的镜像文件系统，而不只是一个增量镜像。这是由于BTRFS没有读写分层的概念并且也很难去列出不同的快照点之间的差异。关于这一点也许以后会得到改善。

现在，让我们探讨一下当运用btrfs存储引擎作为Docker存储时还有哪些值得关注的地方。正如之前所提到的，btrfs引擎要求

将/var/lib/docker目录挂载到BTRFS文件系统上。这样做的优势在于可以让用户的整个宿主操作系统的其他部分免受潜在的文件系统中断的影响。在这里，我们也推荐把/var/lib/docker/vfs/目录挂载到像ext4或者vfs这样久经考验的文件系统。

BTRFS文件系统对于磁盘空间不足的问题非常敏感。用户必须确保能随时监控存储块的使用情况并且在需要的时候不断地重新调整文件系统。

这可能会对运维人员造成一些负担，尤其是当用户运行了大量的容器然后必须在宿主机上保留大量容器镜像的时候。从另一方面来看，用户也因此得获一个可以轻松扩展而无需中断服务的存储服务。

通过运用它的快照特性，BTRFS的写时复制功能使得备份容器和镜像变得超级简单。但是，话说回来，BTRFS的写时复制特性并不适用于创建和修改大量小文件的容器，例如数据库。这会导致经常出现文件系统碎片，因而需要频繁地进行文件系统的调整。针对那些绑定挂载到高IO类应用的容器的目录或者卷，用户可能需要通过禁用它们的写时复制来避免上述的这些问题。

如果下定决心要删除Docker的BTRFS子卷，在通过`rm -rf`命令实际删除底下的目录内容之前，别忘了先确保已经使用**`btrfs subvolume delete subvolume_directory`**删除了对应的子卷，否则这可能会导致文件系统的损坏。这种情况有时候也会在尝试删除镜像或者销毁容器的时候发生，所以请一定要留意这一点。

综上所述，btrfs存储引擎最大的优势和弊端正是BTRFS文件系统本身。它要求使用者在使用方面有充足的操作经验，而且它在多样的生产环境下性能不是很好。BTRFS也不允许共享页面缓存，因为这可能会导致过高的内存使用率。以上这些也正是为什么[CoreOS团队最近](#)决定在他们的操作系统发行版里放弃内置对btrfs的支持的主要原因。

11.4 OverlayFS

overlay是Docker最新引入的存储引擎。它使用OverlayFS文件系统来为Docker镜像分层提供写时复制的支持。和之前一样，需要将环境变量**`DOCKER_OPTS`**修改为指向该引擎，然后重启Docker守护进程。在深入讲解一些实际案例之前，让我们先来介绍一下OverlayFS文件系统。

OverlayFS是一个联合文件系统，Linux内核在3.18版本起就已经内置了对此文件系统的支持。它实际上结合了两种文件系统：

- 上层——子文件系统；
- 下层——父文件系统。

OverlayFS引入了工作目录的概念，所谓的工作目录指的便是一个驻留在上层文件系统的目录，而它通常用于完成上、下层之间文件的原子复制。

下层文件系统可以是任意一个Linux内核（包括overlayFS本身）支持的文件系统并且通常都是只读的。事实上，我们可以通过层层堆叠或者说互相在彼此的顶部“叠加”来划分出多个下层文件系统层。上层文件系统一般是可写的。作为一个联合文件系统，overlayFS也实现了通过经典的“写出”文件来标记将要删除的文件，这与我们在第一节里讲述的AUFS文件系统的做法有些类似。

叠加的主要操作便是目录的合并。每个目录树下相同路径的两个文件将会被合并到叠加后的文件系统里的相同目录下。与之前容器的镜像层数越多，AUFS需要耗费的查找时间则越长的情况相比，OverlayFS在查找效率方面做出了不小的改进。每当有针对合并目录的查找请求时，它会在被合并的目录中同时进行查找，然后将最终的结果缓存到叠加文件系统的固定条目里。如果它们均找到对应的文件，那么这些查找记录会被保存下来，并且会为此创建一个新的合并目录将这些相同的文件合并，否则便只有其中一个会被保存：如果上层查找得到便是上层，否则便是下层。

那么，Docker是如何使用OverlayFS的呢？它将下层的文件系统作为基础镜像层，当创建一个新的Docker容器时，它会自动为之创建一个新的只包含两个镜像层之间增量的上层文件系统。这和我们之前介绍的AUFS的做法简直一模一样。正如我们预期的那样，提交一个容器所创建的新的镜像层同样也只是包含基础镜像层和新镜像层之间的差异而已。

好了，理论就先讲到这里。接下来，让我们来看一个具体案例。在此之前，为了满足Overlayfs的基本使用条件，用户的Linux内核必须是3.18或以上版本。如果用户的内核版本比这个低，Docker将会选择下一个可用的存储引擎，而不使用overlay。

与之前一样，用户需要告知Docker守护进程采用overlay存储引擎。记住，存储引擎的名字是**overlay**而不是**overlayfs**：

```
# docker info
Containers: 0
```

```
Images: 0
Storage Driver: overlay
  Backing Filesystem: extfs
Execution Driver: native-0.2
Kernel Version: 3.18.0-031800-generic
Operating System: Ubuntu 14.04 LTS
CPUs: 1
Total Memory: 489.2 MiB
Name: docker
ID: NQJM:HHFZ:5636:VGNJ:ICQA:FK4U:6A7F:EUDC:VFQL:PJFF:MI7N:TX7L
WARNING: No swap limit support
```

现在，让我们创建一个新容器，随后看下`/var/lib/docker`目录里的内容：

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Important:
image verification is a tech preview feature and
should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8accaa02090f32ee0b95c1
```

所有的镜像层都如预期那样出现在了`/var/lib/docker/overlay`目录里。查找基础镜像和容器镜像层实际上和检索挂载好的该容器文件系统一样简单，而且用户可以在这里看到下层和上层目录里的内容：

```
# grep eb9e1a68c705 /proc/mounts
overlay /var/lib/docker/overlay/
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8ac-
caa02090f32ee0b95c1/merged overlay
rw,relatime,lowerdir=/var/lib/docker/overlay/
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125/
root,upperdir=/var/lib/docker/overlay/
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8ac-
caa02090f32ee0b95c1/upper,workdir=/var/lib/docker/overlay/
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8ac-
caa02090f32ee0b95c1/work 0 0
```

和我们测试之前的存储引擎一样，我们将会在一个正在运行的容器里创

建一个空文件然后提交，而这会立刻触发创建一个新的镜像层：

```
# docker exec -it eb9e1a68c705 touch /etc/testfile
# docker ps
CONTAINER ID          IMAGE               COMMAND             9
CREATED              STATUS             PORTS
NAMES
eb9e1a68c705         busybox:latest     "top"
minutes ago          Up 9 minutes
cocky_bohr
# docker commit eb9e1a68c705
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b
```

用户可以简单地通过如下命令来确认它是否已经创建好新的镜像层以及里面是否包含我们刚刚创建的那个新的空白文件：

```
# ls -l /var/lib/docker/overlay/
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b/
root/etc/testfile
-rw-r--r-- 1 root root 0 Apr  8 21:44 /var/lib/docker/overlay/
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b/
root/etc/testfile
```

上述所有结果完全符合我们的预期，而它最终的表现和[aufs](#)存储引擎所做的几乎完全一样。但是，这里面也有些许的不同。**overly**这里没有**diff**子目录。然而这并不意味着我们就没有办法检索文件系统的增量——它们只是被“隐藏”在了某个地方罢了。**overlay**存储引擎会为每个容器创建3个子目录：

- **upper**——这个便是可读写的上层文件系统层；
- **work**——原子复制所需的临时目录；
- **merged**——正在运行中的容器的挂载点。

此外，该引擎会创建一个叫做**lower-id**的文件，这里面会包含父镜像层的**id**而它的“根”目录将会被**overlay**当做下层目录——该文件一般用于存放父镜像层的查找**id**。

让我们从之前提交的一个镜像层启动一个新容器：

```
# docker run -d
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b
```

```
007c9ca6bb483474f1677349a25c769ee7435f7b22473305f18cccb2fca21333
```

我们可以很方便地通过查看“lower-id”文件来确认容器的父镜像层的id:

```
# cat /var/lib/docker/overlay/  
007c9ca6bb483474f1677349a25c769ee7435f7b22473305f18cccb2fca21333/lower-id  
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b
```

由于overlay存储引擎和之前评测过的aufs引擎有非常多的相似之处，因此这里不再赘述。话说回来，为什么overlay引擎能提供如此多令人兴奋的特性呢？这里面有几个重要的原因：

- Linux内核主干内置了对Overlay文件系统的支持——我们不需要再安装任何额外的内核补丁；
- 因为采用了页面缓存共享的机制，所以它能够占用更少的内存资源；
- 虽然它在下层和上层文件系统之间的复制速度方面有一定的问题，但是综合来说，它依旧比aufs引擎要快上不少；
- 对于镜像之间相同的文件是以硬链接的方式关联在一起，如此就可以避免重复的覆盖并且容许更快的删除/销毁。

尽管有上述这些优点，但OverlayFS仍旧是一个非常年轻的文件系统。虽然从初步的测试结果来看它是十分理想的，但是我们仍然没有看到它实际投入生产环境下应用的具体案例。有鉴于此，越来越多像CoreOS这样的企业和OverlayFS站到了同一阵营，因此我们可以预见到的是未来它会有更多积极的开发和改进。

现在，是时候放下对写时复制这一特性的迷恋了，接下来我们来看一看最后一个Docker存储引擎vfs，这是一个不提供写时复制机制的存储引擎。

11.5 VFS

就像前面提到的，vfs存储引擎是唯一的一个不采用任何写时复制机制的存储引擎。每个镜像层就是一个单一的目录。当Docker创建一个新的

镜像层时它所做的便是将基础镜像层所在的目录全量地物理复制到新建镜像的目录里。这将导致这个引擎运转非常缓慢并且磁盘空间的利用率也会很低。

让我们来看一个使用**vfs**的实际案例。和之前一样，我们首先需要修改一下环境变量**DOCKER_OPTS**，告知Docker守护进程采用**vfs**作为存储引擎，然后重启服务：

```
# ps -ef|grep docker
root      2680      1  0 21:51 ?                00:00:02 /usr/bin/docker
-d --storage-driver=vfs
```

我们将通过一个很小的只运行**top**命令的**busybox**容器来讲解**vfs**引擎的一些特性。

```
# docker run -d busybox top
...
[FILTERED OUTPUT]
...
de8c6e2684acefa1e84791b163212d92003886ba8cb73eccef4b2c4d167a59a4
```

VFS镜像层存放在**/var/lib/docker/vfs/dir**目录里。现在，我们将测试一下使用这一引擎时的一些具体速度和磁盘空间方面的性能表现。我们会通过如下命令启动一个新容器然后在里面生成一个合适大小的文件：

```
# docker run -ti busybox /bin/sh
/ # dd if=/dev/zero of=sample.txt bs=200M count=1
1+0 records in
1+0 records out
/ # du -sh sample.txt
200.0M sample.txt
/#
```

紧接着，提交这个容器来触发Docker创建一个新的镜像层，并且观察一下对应的执行速度：

```
# time docker commit 24247ae7c1c0
7f3a2989b52e0430e6372e34399757a47180080b409b94cb46d6cd0a7c46396e

real    0m1.029s
```

```
user    0m0.008s
sys     0m0.008s
```

结果表明，它耗费了1秒左右的时间来创建一个新的镜像层——这要归咎于基础镜像层里我们创建的那个200 MB大小的文件。最后，让我们从新提交的这一层镜像中再创建一个新容器，然后再看看它的速度是怎样的：

```
# time docker run --rm -it
7f3a2989b52e0430e6372e34399757a47180080b409b94cb46d6cd0a7c46396e
echo VFS is slow

VFS is slow

real    0m3.124s
user    0m0.021s
sys     0m0.007s
```

从先前那个提交的镜像创建一个新容器竟然花了将近3秒！此外，我们拥有的两个容器在宿主机文件系统中每个均占用了200 MB的磁盘空间！

上述的这个例子已经证明VFS是不太适用于生产环境的。它更像是当宿主主机上没有任何支持写时复制特性的文件系统的時候的一个备选方案。尽管如此，VFS依旧算是挂载Docker卷的一个不错的解决方案，原因在于它具备良好的平台兼容性并且当用户打算在像FreeBSD这样的非Linux平台上运行Docker时它会是一个不错的选择。

11.6 小结

Docker在存储引擎方面提供了一个相当全面的选择。这也许是一个喜忧参半的情况，因为通常初学者一旦知道了有这么多的备选后，可能会困惑到底应该选哪个。Donald Knuth关于“[过早优化](#)”的名言常常会在我们讨论技术选型的时候出现在脑海里。

实际上，人们往往会选择自己在生产环境里最有把握运维的那个工具。最后，我们以一张简短的表（表11-1）来结束本章，表中列出了一些在

选择使用哪个存储引擎之前需要评估的要点。

表11-1

	AUFS	OverlayFS	BTRFS	DeviceMapper
上线	非常快	非常快	快	快
小文件 I/O	非常快	非常快	快	快
大文件 I/O	慢	慢	快	快
内存使用率	高效	高效	高效	不是很高效
缺陷	没有在内核主干里，有层数限制，随机并发度的问题	不成熟	接近成熟，但是实际上还不是，存储扩容会很费劲，需要一些扎实的运维知识	高密度的容器及很高的磁盘占用，周边生态不好，大部分需要的是运维经验

Docker在网络领域同样带来了革命性的挑战，我们将在第12章讲解这方面的内容。

[1] 是DeviceMapper提供的一项特性，它允许在实际使用时实报实销的资源利用，即类似虚拟内存的一项技术。——译者注

第12章 Docker 网络实现

在第11章里，我们了解了各种Docker原生支持的存储引擎和它们是如何帮助Docker打包镜像以及更高效地从已构建的镜像中运行容器等内容。实际上，Docker真正的强大之处在于使用它来构建分布式应用。这些分布式应用通常由散布在整个计算机网络内为了完成一些计算任务而交互的一定数量的服务程序组成。在本章中，我们将解答一个看上去很简单的问题：如何才能使得运行在Docker容器内部的应用程序在网络上能够实现相互访问？为了解答这一问题，我们首先需要了解一下Docker的网络模型。

Docker的网络实现包含3方面的内容。分别是IP的分配、（域）名的解析以及容器或服务的发现这3块。上述所有的概念即是著名的[零配置网络](#)（即zerocnf）的理论基础。简而言之，零配置网络即是一组在没有任何人工干预的情况下自动创建和配置一个TCP/IP网络的技术。

在每台宿主机上，Docker容器的分布密度快速波动（并且一般也是如此）。对于这样的复杂情况，自动化网络配置的概念就显得尤为重要。管理和运维这样的环境，以及针对其有时候横跨多个云服务平台的复杂情况建立起一套理想而安全的网络基础设施，想想都觉得会是一个非常艰巨的挑战。在这些复杂场景下实现零配置网络的“愿景”是我们在计算机网络方面的终极目标。它可以帮助开发和运维人员从以前手工配置网络的重担中解放出来。接下来，将探讨Docker为了尽可能的实现这一愿景所提出的众多可选的网络解决方案。

当外界的计算网络访问这些在Docker容器内部运行的应用服务时，用户们常常面临如下问题：

- 该如何去访问一个运行在Docker容器里的应用或者服务？
- 如果应用程序在Docker容器内部运行，那么该如何以一种相对安全的方式去连接或者发现所依赖的应用服务？
- 在网络上，该如何加固这些运行在Docker容器内部的应用？

在这一章里，我们将试图寻找上述问题的答案。我们将会从Docker内部的网络实现原理讲起，这应该可以让你对当前的网络模型有一个很好的大体上的了解。接下来，我们将研究一些具体的实战案例，用户可以在任意一台Docker宿主机上执行里面的命令。最后，我们将探讨在现有的网络模型不满足需求的情况下的一些可选的替代方案。话不多说，是时候实际行动了！

12.1 网络基础知识

Docker的网络模型非常简单，但同时也相当强大。默认情况下（即Docker守护进程的默认配置），无需用户任何人工的干预，所有新创建的Docker容器都会自动地连上Docker的内部网络：你简单地运行一下类似于`docker run <image> <cmd>`的命令，一旦你的容器启动，它便会自动地出现在网络上。这听上去挺神奇的，不妨让我们看一下它背后的实现原理吧。

当Docker守护进程以其默认的配置参数在一台宿主机上启动时，它会创建一个Linux网桥设备并将其命名为`docker0`。该网桥随后会自动地分配一个满足RFC 1918定义的私有IP段的随机IP地址和子网。该子网决定了所有新创建容器将被分配的容器IP地址所属的网段。当前的网络模型如图12-1所示。

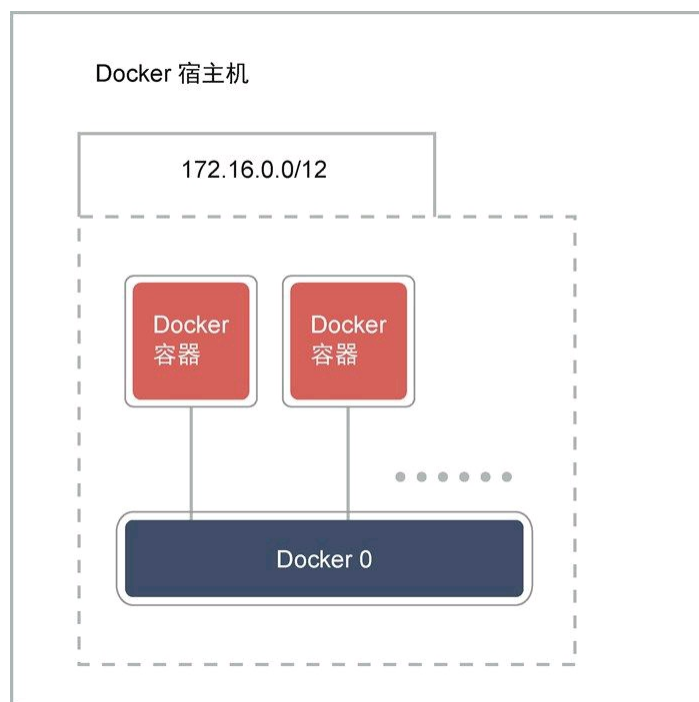


图12-1

除了创建网桥设备，Docker守护进程还会在宿主机上修改一些**iptables**规则。它会创建一个叫做**DOCKER**的特殊过滤链并且把它插入到**FORWARD**链的最上面。它也修改了一些**iptables nat**表的规则使得容器可以建立对外的连接。设置了这些规则以后Docker容器内部之间的网络连接在接收端便会显示对方的内网IP地址，不过，容器对外的网络连接仍由宿主机上的一个IP地址发出，而不是最开始发起连接的那个容器的IP地址。这一点有时候会让初学者感到困惑。

如果不希望Docker修改宿主机上的**iptables**规则，那么必须以**--iptables**参数设置为**false**的方式启动Docker服务。也可以通过简单地设置环境变量**DOCKER_OPTS**，然后重启服务进程达成这一点。默认情况下Docker配置里的这个参数是设置为**true**的。

注意：**DOCKER_OPTS**必须在Docker守护进程启动前设置好。如果你希望你所做的更改能够持久化，甚至于在Docker宿主机重启后仍然能生效，那么你必须修改一些**init**服务配置或脚本文件。这在各个Linux发行版之间存在着些许不同，在Ubuntu上你可以通过修改**/etc/default/docker**文件来完成这一需求。

Docker通过创建一个单独的**iptables**链来管理容器之间的访问，这使

得系统管理员们可以很方便地以修改**DOCKER**链的方式来管理容器的外部访问，在此过程中他们无需接触宿主机上任何其他的**iptables**规则，这也就避免了一些意外修改的情况。在**DOCKER**链里追加和修改规则本质上即是**Docker**如何管理容器之间的链接的具体实现，这些内容我们将稍后介绍。

Docker守护进程会为每个新建的容器创建一个新的网络命名空间。然后它会生成一对**veth**设备。**veth**（**Virtual Ethernet**的简写）是一类特殊的**Linux**网络设备，它通常是成对（或者说结对）出现的，而且大致上来说它扮演的角色类似于是一个“网络管道”：从一端传入的任何数据都会被传到另外一端。事实证明，这很方便地实现了在**Linux**内核里不同网络命名空间之间的相互通信。**Docker**会将**veth**对等接口中的其中一个连接到容器的网络命名空间里然后在宿主机的网络命名空间里持有另外一个，后者的名称是一个带有**veth**前缀的随机生成的字串。**veth**对里的每一个对等接口只有在它当前所属的命名空间下才能看到。

随后，**Docker**会将宿主机上的**veth**对等接口绑定到**docker0**网桥上，然后为另外一个容器的**veth**对等接口分配一个之前**Docker**守护进程启动时选定的私有IP网段里的IP地址。一旦宿主机上的**veth**对等接口桥接上了**docker0**设备，**Docker**会立马在宿主机上的路由表上为之前选定的私有IP网段插入一条新的路由记录，并且开启宿主机上的IP转发。这便使得用户可以在宿主机上直接与容器通信。关于这一设定的详细过程如图12-2所示。

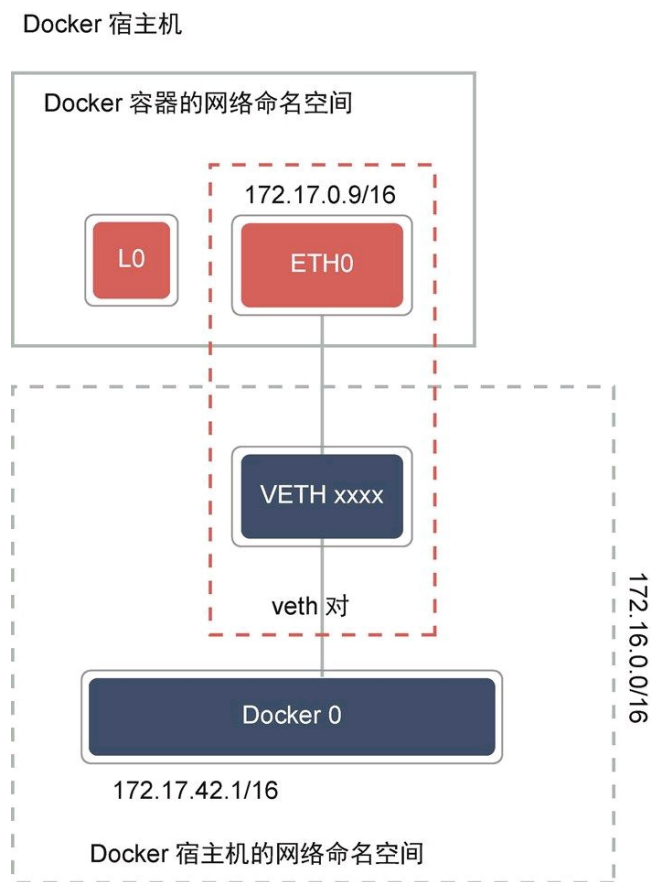


图12-2

默认情况下，Docker容器只能从内网访问，它们一般不对外提供路由。Docker并不提倡从外网访问容器，因此外界的宿主机一般很难直接和它们通信。

注意：如果以将`--iptables`参数设置为`false`的方式启动Docker服务，它将不会再在宿主机上操作任何`iptables`的规则。它也不再会配置IP转发，这就意味着你的容器将无法再访问外界的应用甚至本地上其他的容器。如果这不是你预期的结果，可能就要考虑一下在系统上手动开启IP转发的配置。在Linux上，你可以通过设置`/proc/sys/net/ipv4/ip_forward`内核参数为“1”来实现这一点。

12.2 IP地址的分配

在容器更换迭代频繁的情况下通过手工的方式来管理IP地址毕竟不是一个长久之计，一旦容器的数量上了规模将没办法继续这样做。这正是Docker在IP地址分配方面需要解决的问题。正如之前提到的那样，IP自动分配是零配置网络实现的基石之一，并且Docker手上已然有了一个实现的方案。Docker能够做到在没有任何人工干预的情况下为新创建的容器自动地分配IP地址。Docker守护进程会维护一组已经分配给那些正在运行的Docker容器的IP地址以避免为新容器再分配相同的IP地址。当一个容器停止或者被删除的时候，它的IP地址也会被回收到由Docker守护进程维护的IP地址池里，这样，当新容器启动时便可以直接复用这些IP资源。

如果在容器销毁后，释放的IP地址映射到容器对应MAC地址的缓存ARP上的记录没有被立即清除，立刻复用该IP可能会导致宿主机本地网络的ARP冲突。Docker采取的办法是为每个已分配的IP地址生成一串随机MAC地址来解决这个问题。该MAC地址生成器确保是强一致的：相同的IP地址生成的MAC地址将会是完全一致的。Docker也允许用户在创建新容器时手动指定MAC地址，然而，由于上述所提到的ARP冲突的问题，我们并不建议这样做，除非你想出一些其他的机制可以规避它。

太好了！IP（和MAC地址）的分配都是在没有任何用户手动干涉的情况下“自动”完成。一旦新容器被创建出来，它们便会以其自动分配的IP地址出现在Docker的私有网络里。而这正是你想要借助零配置网络实现的。Docker甚至还更进了一步：为了使得运行在容器里的服务之间能够相互通信，Docker还必须支持端口的分配。

端口的分配

当一个容器启动时，Docker可以为它自动分配任意的UDP或者TCP端口并且使之能够在宿主机上被访问。用户可以在构建容器的镜像时通过在Dockerfile里使用EXPOSE指令来指定对外公开的端口，也可以通过--expose命令在容器启动的时候显式声明。该命令允许用户定义某个范围内的端口而不只是单个的端口映射。然而，要知道声明大范围的端口段的影响在于所有相关的信息都是可以通过Docker服务的远程API获取的，因此查询一个占有巨大端口段的容器的话很容易就会暴露Docker守护进程。

Docker守护进程随后会从Linux宿主机上的文件

——`/proc/sys/net/ipv4/ip_local_port_range`定义的一个端口范围里挑选出一个随机的端口号。如果失败，如当Docker守护进程运行在非Linux宿主机上时，Docker将会改为从这个端口范围（49153~65535）里申请对应的端口。这并不是自动完成的：Docker守护进程只维护正在宿主机上运行的容器公开的端口号。用户必须明确地通过被Docker称之为发布端口的方式触发一次宿主机的端口映射。发布端口使得用户可以绑定任意公开的端口到宿主机上任何一个对外可路由的IP地址。如果用户在之前构建镜像的时候没有设置任何公开的端口，那么发布端口将对运行在Docker容器里的服务的对外可用性没有任何影响。

用户可以通过执行如下命令来找出指定容器对外公开的具体端口信息：

```
# docker inspect -f '{{.Config.ExposedPorts}}' <container_id>
```

警告：用户只能在启动新容器的时候为其发布对应的公开端口。一旦容器已经开始运行，我们将没有办法再发布其他的公开端口。必须从头开始重新创建容器！

用户可以选择发布所有的公开端口或者只发布那些用户挑选出来的愿意让它对外可访问的。Docker提供了非常便利的命令行参数来实现各种组合。用户可以通过Docker帮助来了解所有可用的参数选项。

端口分配的背后主要在于iptables的妙用，Docker就是通过灵活运用之前提到的DOCKER链和nat表实现这一点的。为了帮助读者更好地理解这项功能，我们将通过图12-3所示的具体案例来讲解相关内容。

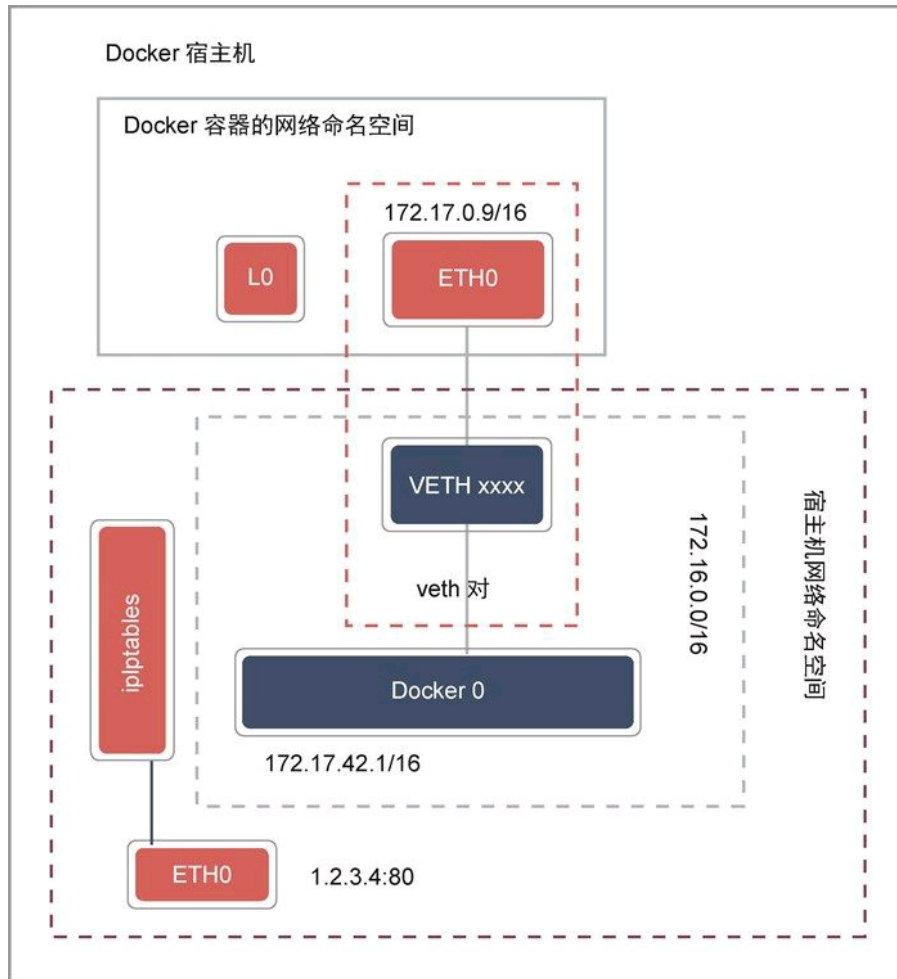


图12-3

假设，我们想在Docker容器里运行nginx Web服务并且需要通过宿主机上的可对外路由的IP地址1.2.3.4和对应的TCP端口80使其能够被外界访问。我们需要用到library/nginx的Docker镜像，实际上，当用户运行docker pull nginx命令的时候，Docker从Docker Hub上拉取的默认镜像便是我们需要用到的library/nginx镜像。我们通过执行如下命令（记得带上-p参数）来完成上述任务：

```
# sudo docker run -d -p 1.2.3.4:80:80 nginx
a10e2dc0fdb2dc8259e9671dccc6853d77c831b3a19e3c5863b133976ca4691
#
```

可以通过执行以下命令来验证由这个镜像创建出的容器是否的确公开了TCP 80端口：

```
# sudo docker inspect -f '{{.Config.ExposedPorts}}' a10e2dc0fdfb
map[443/tcp:map[] 80/tcp:map[]]
```

可以看到，我们用来实例化容器的**nginx**镜像本身还公开了443端口。现在，容器已经开始运行，用户可以通过执行以下命令轻松地来检索它公开的所有端口（也称为主机端口绑定）：

```
# sudo docker inspect -f '{{.HostConfig.PortBindings}}'
a10e2dc0fdfb
map[80/tcp:[map[HostIp:1.2.3.4 HostPort:80]]]
```

提示：这里有一个简便命令行来检查一个特定**Docker**容器IP：*port*绑定的内容：**docker port container_id**。

太棒了！**nginx**如今在**Docker**容器里运行并且它能够通过之前给定的IP地址和端口对外提供服务。我们现在可以通过执行简单的**curl**命令在外面的宿主机（当然，我们假定你的防火墙没有禁止外界对80端口的访问）上访问其默认的**nginx**站点：

```
# curl -I 1.2.3.4:80
HTTP/1.1 200 OK
Server: nginx/1.7.11
Date: Wed, 01 Apr 2015 12:48:47 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 24 Mar 2015 16:49:43 GMT
Connection: keep-alive
ETag: "551195a7-264"
Accept-Ranges: bytes
```

当用户在宿主机上对外公开一个端口时，**Docker**守护进程会在**DOCKER**链里追加一条新的**iptables**规则，它将会把宿主机上所有目标是**1.2.3.4:80**的流量重定向到一个特定容器的80端口上，并且会据此修改**nat**表的规则。用户可以通过运行如下命令轻松地检索这些信息：

```
# iptables -nL DOCKER
Chain DOCKER (1 references)
target      prot opt source          destination
**ACCEPT    tcp  --  0.0.0.0/0        1.2.3.4
tcp dpt:80**
```

```
# iptables -nL -t nat
Chain PREROUTING (policy ACCEPT)
target      prot opt source                destination
DOCKER      all  --  0.0.0.0/0             0.0.0.0/0
ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target      prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target      prot opt source                destination
DOCKER      all  --  0.0.0.0/0             !127.0.0.0/8
ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target      prot opt source                destination
MASQUERADE  all  --  172.17.0.0/16         0.0.0.0/0
**MASQUERADE tcp --  172.17.0.5
172.17.0.5      tcp dpt:80**

Chain DOCKER (2 references)
target      prot opt source                destination
**DNAT      tcp  --  0.0.0.0/0             1.2.3.4          tcp
dpt:80 to:172.17.0.5:80**
```

注意：当关闭或删除一个发布了一定数量端口的容器时，*Docker*会删除它最初创建的所有必需的*iptables*规则，因此用户不必再担心这些。

最后，让我们来看一看如果我们使用`-P`选项来启动一个新容器的话会发生什么，使用该参数意味着我们无需再为公开端口指定任何主机端口或者IP地址映射信息，*Docker*将自动帮我们完成端口的映射。我们将和之前的例子一样使用相同的*nginx*镜像：

```
# docker run -d -P nginx
995faf55ede505c001202b7ee197a552cb4f507bc40203a3d86705e9d08ee71d
```

同之前提到的那样，由于我们没有显式地指定将容器端口绑定到宿主机的接口上，因此*Docker*将会把这些端口映射到宿主机的一个随机端口上。可以通过执行以下命令来发现该容器绑定的端口信息：

```
# docker port 995faf55ede5
443/tcp -> 0.0.0.0:49153
80/tcp -> 0.0.0.0:49154
```

当然，也可以通过执行以下命令直接检索网络端口的信息：

```
# docker inspect -f '{{ .NetworkSettings.Ports }}' 995faf55ede5
map[443/tcp:[map[HostIp:0.0.0.0 HostPort:49153]] 80/tcp:
[map[HostIp:0.0.0.0 HostPort:49154]]]
```

可以看到两个公开的TCP端口分别被绑定到了宿主机上所有网络接口的**49153**和**49154**端口。读者可能已经猜到了，**Docker**正是通过它的远程API公开的这些信息，它为需要相互通信的各个容器实现了一个非常简单的服务发现。当用户启动一个新容器时可以通过一个环境变量来传入对应的宿主机端口映射信息。关于这一点，我们还可以通过一个更好也更安全的方式来实现，这部分内容我们将在本章的稍后部分详细介绍。

所有的这一切看起来棒极了。我们不用再去手动管理运行在宿主机上的**Docker**容器的IP地址：**Docker**包揽了一切！然而，当我们开始扩大**Docker**基础设施的规模的时候，一些问题可能就会暴露出来。由于**Docker**守护进程接管了特定的**Docker**宿主机上所有IP分配的活，我们该如何确保它不会在不同的宿主机上针对不同的容器分配相同的IP地址呢？甚至于我们是否需要关注这个问题呢？针对这些问题的答案也是很常见的：这得看情况。我们将在12.5节进一步探讨相关话题。现在让我们先转到零配置网络实现的下一个重要组成部分：域名解析。

12.3 域名解析

Docker提供了一些配置参数允许用户在容器基础设施里管理域名的解析。同以往一样，为了利用好这些功能而又不会引发一些不可预料的结果，我们需要先了解一下**Docker**内部是如何处理域名解析的。

当**Docker**创建新容器时，默认情况下它会给该容器分配一个随机生成的主机名以及一个唯一的容器名。这是两个完全不同但是又容易搞混的概念，尤其对于新手来讲，他们在使用时常常会混淆这两个概念。

主机名可以理解为就是一个普通的Linux主机名：它允许运行在容器里的进程通过解析该容器的主机名来获取它对应的IP地址。容器的主机名并不是一个可以在容器外部环境解析的域名，而且默认情况下它会被

设置为容器的**ID**，即一串允许用户从命令行或者通过远程API在宿主机上定位任意容器的唯一字符串^[1]。容器名，从另一方面来说，是一个Docker内部的概念，它与Linux无关。

可以通过使用Docker命令行工具执行如下命令来查询容器名：

```
docker inspect -f '{{.Name}}' container_id
```

容器名在Docker里主要有两个用途：

- 与容器ID相比，它使得用户可以通过远程API使用友好、可读的名称来查找容器；
- 它有助于构建一个基本的基于Docker的容器发现。

用户可以使用Docker客户端提供的一些特定的命令行参数来重载Docker守护进程设定的默认值。

容器的主机名和容器名可以被设置成相同的值，但是除非你有一些自动管理它们的方法，否则我们建议不要这样做，因为在一个高密度的容器部署的条件下，这可能会变成一个难以维护的情况并将成为运维人员的噩梦。

在容器镜像构建时无论是主机名还是容器名都不会硬编码到容器镜像里面。Docker实际上会主动在Docker宿主机上生成/etc/hostname和/etc/hosts文件，然后在新创建的容器启动时将两者绑定挂载到里面。用户可以通过检查宿主机里下述文件的内容来确认这一点：

```
# cat /var/lib/docker/containers/container_id/hostname
# cat /var/lib/docker/containers/container_id/hosts
```

提示：可以通过在命令行里运行`docker inspect -f '{{printf "%s\n%s" .HostnamePath .HostsPath}}' container_id`命令得到相应文件的具体路径信息。

我们将在本章的后面部分讨论更多关于容器名概念的详细内容；现在我们只需要记住，如果用户想通过一个友好、可读的名字而不是随机生成的容器ID来查找Docker容器，就可以给它们分配一个自定义的名字。用户无法修改一个已经创建好的容器的名称——遇到这种情况只能从头重

新创建它。

注意：从0.7版本起，*Docker*会用一些著名的科学家和黑客的名字来命名容器。如果你也想为*Docker*项目做一份贡献，可以在[GitHub](#)上开一个[Pull Request](#)加入你认为值得推荐的名人。在*Docker*里，处理这个的Go包叫做*namesgenerator*，在*Docker*的代码库中的*pkg*子目录下找到它。

现在你已经知道容器是如何将它的主机名解析为对应的IP地址了，那么是时候再去了解下它是如何解析外部DNS域名了。如果你猜到容器可能是像平常Linux主机那样使用/etc/resolv.conf文件来实现，那么恭喜你，答对了！*Docker*会在宿主机上为每个新创建的容器生成/etc/resolv.conf文件，然后当启动该容器时将这个文件挂载到容器里。用户可以通过在*Docker*宿主机上检索以下内容来验证这一点：

```
# sudo cat /var/lib/docker/containers/container_id/resolv.conf
```

提示：可以通过在命令行执行`docker inspect -f '{{.ResolvConfPath}}' container_id`来找出上述文件的具体路径。

默认情况下，*Docker*会将宿主机上的/etc/resolv.conf文件复用到新创建的容器里。在1.5版以后，当用户在宿主机上通过修改这个文件更改了DNS配置时，如果希望这些变动也应用到现有已经在运行的容器（那些使用之前的配置创建的），则必须重启这些容器方能使之生效。如果你还在用老版本，那就不是这样了，你必须从头重新创建这些容器。

如果不想*Docker*容器采用宿主机的DNS配置，可以通过修改环境变量DOCKER_OPTS来重载它们，可以在启动守护进程时通过命令行参数或者通过修改宿主机上一个特定的配置文件将变动固化下来（在Ubuntu Linux发行版上这个文件是/etc/default/docker）。

假设用户现在有一个专门的DNS服务器，并且希望自己的*Docker*容器都使用它来完成DNS的解析。再假定这个DNS服务器可以用1.2.3.4 IP地址访问并且它管理了example.comexample.com域。这样的话，用户可能需要按如下方式修改环境变量DOCKER_OPTS：

```
DOCKER_OPTS="--dns 1.2.3.4 --dns-search example.com"
```

为了让Docker守护进程应用新配置，需要重启一次守护进程。用户可以通过执行以下命令检查`/etc/resolv.conf`的内容来验证新创建的容器现在是否真的采用了新的DNS配置：

```
# sudo cat /var/lib/docker/containers/container_id/resolv.conf
nameserver 1.2.3.4
search example.com
```

所有在Docker守护进程的DNS配置修改之前已经启动的容器将仍然保持原来的配置。如果希望这些容器采用新配置，就必须要从头重新创建它们，简单的重启容器无法实现预期效果！

注意：之前所提到的Docker守护进程的DNS选项参数将不会重载宿主机上的DNS配置。它们只会作为自Docker守护进程开始应用这一新配置起在宿主机上创建的所有容器的默认DNS配置。然而用户也可以针对每个容器显式地重载他们的配置。

Docker甚至允许更细粒度地控制容器的DNS配置。用户可以在启动新容器时像这样重载其全局的DNS配置：

```
# sudo docker run -d --dns 8.8.8.8 nginx
995faf55ede505c001202b7ee197a552cb4f507bc40203a3d86705e9d08ee71d
# sudo cat $(docker inspect -f '{{.ResolvConfPath}}'
995faf55ede5)
nameserver 8.8.8.8
search example.com
```

读者可能已经注意到了这里面的一个小细节。正如在上述命令的输出中所能看到的，我们只是为新容器设置了`--dns`的配置，但是`/etc/resolv.conf`里面的`search`指令已经被修改了。Docker将把命令行上指定的参数配置和Docker守护进程本身规定的配置两者做一次合并。当Docker守护进程设置`--dns-search`为一些特定域时，如果用户在启动新容器的时候只重载了`--dns`参数，容器将会继承Docker服务设定的搜索域配置，而不是宿主机上的配置。目前我们没有办法改变这一行为，因此必须留意这一行为！

此外，当Docker守护进程指定的DNS设置发生变动时，采用自定义DNS

配置创建的容器将不会受到任何影响，即使在这之后重启这些容器也同样如此。如果想让它们转为采用Docker守护进程的配置，那么必须以不指定任何自定义参数的方式重新创建它们。

警告：如果用户在正在运行的容器里直接修改了`/etc/resolv.conf`、`/etc/hostname`或者`/etc/hosts`文件，要注意`docker commit`不会保存用户所做的这些变更，并且在容器重启之后，这些变更也不会被保留下来！

正如在本章中所看到的，Docker在无需任何人工干预的情况下自动地完成了每个容器的DNS配置。这再一次完美的践行了零配置网络的准则。甚至于如果用户把容器导出并迁移到其他的宿主机，Docker仍会采用之前的DNS配置，所以用户无需担心再为它们从头配置一次DNS。

能够在容器启动后在其内部直接解析外部DNS域名固然是非常方便的，但是如果我们希望能从容器里访问其他容器里的服务呢？正如之前所了解的，容器的主机名在他们外部是无法解析的，因此不能使用容器的主机名来完成容器之间的通信。而为了使一个容器能够和另外一个容器通信，就必须知道其他容器的IP地址，可以通过查询Docker远程API得到这些信息。但是这一点在容器内部是没有办法做到的，除非用户在容器启动时绑定挂载Docker守护进程的套接字或者在`docker0`网桥接口上公开Docker API服务。此外，查询API本身有一定的额外开销，这会带来不必要的复杂度，并且有点背离我们最初设定零配置网络的初衷。要解决这个问题归根结底还是得依靠零配置网络的最后一个核心组成：服务发现。

12.4 服务发现

Docker提供了一个开箱即用的、虽然基础，但是功能却非常强大的服务发现机制：**Docker**链接。正如我们之前所了解到的，在Docker的内网里，所有的容器都是可以访问得到的，因此默认只要它们知道彼此的IP地址，相互之间便能够直接通信。但是仅仅发现其他容器的IP地址还不够，还得找出容器化的服务接受外界请求时连接所需的端口信息。

Docker链接使得用户可以让任意容器发现其他Docker容器的IP地址和公

开的端口。**Docker**提供了一个非常方便的命令行参数来实现这一点，我们不必再大费周折，它会帮我们自动搞定这一切。该参数即**--link**。当创建一个新容器并将它链接到其他容器时，**Docker**会将所有连接方面的具体数据以多个环境变量的形式导出到源容器里。这可能比较难理解。让我们通过以下的具体案例来讲解得更清楚些。

我们将通过**nginx**容器来讲解如何完成**Docker**链接，首先从检索IP地址和分配端口开始。可以通过执行如下命令来找出容器的IP地址：

```
# docker ps -q
a10e2dc0fdfb
# docker inspect -f '{{.NetworkSettings.IPAddress}}'
a10e2dc0fdfb
172.17.0.2
```

--link参数遵循如下语法格式：**container_id:alias**。这里面的**container_id**是运行中的容器的id，而**alias**（别名）是一个随便起的名字，关于这块内容我们将在后面部分详细解释。我们将会试着在一个新的“一次性”容器（带上**--rm**标志即意味着一旦容器退出便会将容器删除）里ping **nginx**容器的IP地址。

```
# docker run --rm -it --link=a10e2dc0fdfb:test busybox ping -c
2 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.615 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.248 ms

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.248/0.431/0.615 ms
```

正如预期那样，默认情况下，在内网里容器之间可以相互通信（除非容器之间的通信被禁用了，关于这一点，可以在12.5节中了解更多的内容）。

当容器被链接时，**Docker**会自动更新源容器上的**/etc/hosts**文件，将命令行里带的链接别名和目标容器的IP地址关联上，在我们这个例子里便是简单的**"test"**。可以通过运行如下命令来验证这一点（见输出最底下那条记录）：

```
# docker run --rm -it --link=a10e2dc0fdfb:**test** busybox
cat /etc/hosts
172.17.0.13    a51e855bac00
127.0.0.1     localhost
::1          localhost ip6-localhost ip6-loopback
fe00::0       ip6-localnet
ff00::0       ip6-mcastprefix
ff02::1       ip6-allnodes
ff02::2       ip6-allrouters
**172.17.0.2   test**
```

这意味着相比于之前做的ping测试那样直接使用目标容器的IP地址，还可以像下面这样通过被链接的容器的别名来引用它：

```
# docker run --rm -it --link=edb055f7f592:test busybox ping -c
2 test
PING test (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.492 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.230 ms

--- test ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.230/0.361/0.492 ms
```

我们前面还提到Docker在链接两个容器时还会为之创建一些环境变量，它们可以帮助源容器轻松发现目标容器公开的服务端口。当源容器尝试和目标容器建立连接时它无需知道对方的IP地址和公开的端口，只需读取环境变量里的对应内容即可，它们是Docker为每个公开的端口自动创建的，并且会推送到源容器的执行环境里。这些环境变量的名字遵循以下格式：

```
ALIAS_NAME
ALIAS_PORT
ALIAS_PORT_<EXPOSEDPORT>_TCP
ALIAS_PORT_<EXPOSEDPORT>_TCP_PROTO
ALIAS_PORT_<EXPOSEDPORT>_TCP_PORT
ALIAS_PORT_<EXPOSEDPORT>_TCP_ADDR
...
...
```

可以通过运行以下命令和检查对应的输出结果来验证这一点：

```
# docker run --rm -it --link=a10e2dc0fdfb:test busybox env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin
HOSTNAME=ff03fba501ea
TERM=xterm
TEST_PORT=tcp://172.17.0.2:80
TEST_PORT_80_TCP=tcp://172.17.0.2:80
TEST_PORT_80_TCP_ADDR=172.17.0.2
TEST_PORT_80_TCP_PORT=80
TEST_PORT_80_TCP_PROTO=tcp
TEST_PORT_443_TCP=tcp://172.17.0.2:443
TEST_PORT_443_TCP_ADDR=172.17.0.2
TEST_PORT_443_TCP_PORT=443
TEST_PORT_443_TCP_PROTO=tcp
TEST_NAME=/mad_euclid/test
TEST_ENV_NGINX_VERSION=1.7.11-1~wheezy
HOME=/root
```

从上述案例可以看出，我们用来测试链接的nginx容器，公开了两个TCP端口：80和443。Docker在这里创建了两个环境变量：针对每个公开的端口分别是TEST_PORT_80_TCP和TEST_PORT_443_TCP。通过链接的方式导出这些环境变量，Docker在实现了一个简单可移植的服务发现功能的同时又保证了容器的安全可靠。然而天上永远不会掉馅饼。链接虽然是一个很棒的概念，但是你会发现它又是相对静态的。

当链接的目标容器消亡时，源容器便会丢失同链接容器所提供的服务的连接。在容器更替非常频繁的动态环境下这可能会是一个问题。除非用户在自己的应用里实现了一些基本的健康检测机制，或者说至少会做一些故障转移，否则还是应该时刻关注这方面的情况。

此外，当用户恢复发生故障的目标容器时，源容器的/etc/hosts文件将会自动加上目标容器新申请的IP地址，但是通过链接的方式注入源容器的环境变量将不会被更新，因此如果用户事先不知道服务的端口信息的话可能还是会不太理想。另外，最好不要依赖这些环境变量来发现目标容器的IP地址，我们更建议使用链接别名的方式，它会通过/etc/hosts文件自动解析更新后的IP地址。

解决链接本身数据更新不及时问题的一种办法是使用一个名为[docker-grand-ambassador](#)的工具。它致力于解决在使用Docker链接时可能会遇到的一些问题。更高级并且广泛适用的一个方案便是采用现有的一些不错的DNS服务软件来解决容器的服务发现，毕竟它不会引入额外的复杂

度。目前，开源界已经有一些现成的DNS服务的实现方案，他们提供了对Docker容器开箱即用的服务发现的支持，并且不用耗费太大力气便能将其集成到Docker基础设施里。

加上这一节讨论的Docker的服务发现，至此我们关于零配置网络的几个核心组件的介绍基本告一段落。可以看到，当用户在同一台宿主机上运行所有的容器时，Docker能够高水准地满足零配置网络的所有需求。然而，遗憾的是，一旦用户把自己的容器扩展到多台机器或者甚至是多个云服务厂商时Docker便无法完美地交付。我们非常期待Docker Swarm即将发布的新特性以及新的网络模型，它们应该能解决这一节所遇到的问题^[2]。

现在，让我们转到一些更高级的网络主题，深入探讨下Docker的核心网络模型，并且我们会在这里讨论下前面章节中遇到的一些问题的解决方案。

12.5 Docker高级网络

这一节我们将首先从网络安全方面谈起，然后我们会转到探讨关于多个Docker宿主机跨主机的容器间如何通信的话题，最后我们将讨论一下网络命名空间的共享。

12.5.1 网络安全

网络安全实际上是一个相当复杂的话题，关于它的内容甚至可以出一本单独的书来讲解。在本节中我们将会只覆盖到里面的几个小部分，主要是在设计Docker基础设施或者是使用外部供应商提供的Docker基础服务时需要注意的一些地方。我们的重点会放在Docker原生提供的一些解决方案，当然我们会在本节的末尾部分介绍一些其他的替代方案。

默认情况下，Docker允许容器之间可以不受限制地随意通信。很显然，正如你所预见，这可能会存在一些潜在的安全风险。一旦Docker网络上的某个容器出了问题并且对同一网络内的其他容器发动了拒绝服务攻击的话该怎么办？这些网络攻击，有些可能是恶意发起的，也有可能只是软件bug所引发的。这类情况在多租户的环境下尤其应该得到重视。

幸运的是，Docker允许完全禁用容器之间的通信，只要在Docker守护进程启动时传入一个特定的参数即可。该参数名为`--icc`^[3]，默认情况下该参数是设置为`true`的。如果想完全禁用Docker容器之间的通信，那么必须通过修改环境变量`DOCKER_OPTS`将该参数设置成`false`，随后重启Docker守护进程使之生效。此方法的实现原理是，Docker守护进程会在FORWARD链里插入一条新的DROP策略的iptables规则，它会丢弃所有目标是Docker容器的包。在Docker守护进程重启完成后，可以通过执行如下命令来验证这一点：

```
# sudo iptables -nL FORWARD
Chain FORWARD (policy ACCEPT)
target      prot opt source                destination
DOCKER      all  --  0.0.0.0/0              0.0.0.0/0
**DROP      all  --  0.0.0.0/0              0.0.0.0/0**
ACCEPT      all  --  0.0.0.0/0              0.0.0.0/0
ctstate RELATED,ESTABLISHED
ACCEPT      all  --  0.0.0.0/0              0.0.0.0/0
```

从这一刻起，容器之间就再也无法互相通信了。我们可以对之前已经在运行的nginx容器再作一次简单的ping测试来验证这一点。首先，我们得拿到nginx容器对应的IP地址：

```
# docker inspect -f '{{.NetworkSettings.IPAddress}}'
a10e2dc0fdfb
172.17.0.2
```

现在，让我们从一个一次性容器里ping它：

```
# docker run --rm -it busybox ping -c 2 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

正如所见，由于容器间的通信已经被禁用了，因此该ping测试的结果显示的是完全失败。然而，我们依旧可以在宿主机上连接nginx容器：

```
# ping -c 2 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.098 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.066 ms
```

```
--- 172.17.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.066/0.082/0.098/0.016 ms
```

此外，所有之前已经发布的端口保持不变，因此运行在nginx容器里的nginx服务仍然可以正常访问：

```
# curl -I 1.2.3.4:80
HTTP/1.1 200 OK
Server: nginx/1.7.11
Date: Wed, 01 Apr 2015 12:48:47 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 24 Mar 2015 16:49:43 GMT
Connection: keep-alive
ETag: "551195a7-264"
Accept-Ranges: bytes
```

那么我们现在该怎样只启用那些需要相互通信的容器之间的访问呢？答案当然还是**iptables**。如果不想手工管理这些**iptables**规则，别担心，**Docker**提供了一个很方便的命令行参数，这样的话，当用户创建新容器时，**Docker**便会帮助用户自动地完成这些配置。事实上，这个参数我们在服务发现一节里便已经介绍过，没错，它就是：**--link**。**Docker**容器之间的链接不仅可以为我们提供一个简单的服务发现机制，它也为链接的容器之间提供了一个安全的网络通信：只有相互链接的容器之间才能相互通信，并且它们只能访问那些公开的服务端口。**Docker**正是通过在**DOCKER**链里插入一个“双向通信”的**iptables**规则来实现的这一点。

一些实际案例可能更有助于我们加强对这部分内容的理解。下面，我们仍然复用之前已经在运行的**nginx**容器，并且将它链接到一个一次性容器。让我们首先验证一下链接是否真的只允许特定的公开端口的通信。因为**nginx**容器仍然在运行，所以它的IP地址自然还是之前分配的那个：

```
# docker inspect -f '{{.NetworkSettings.IPAddress}}'
a10e2dc0fdfb
172.17.0.2
```

然后，我们再用一个一次性容器去ping它：

```
# docker run --rm -it -link=a10e2dc0fdb:test busybox ping -c 2
172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

由于ping使用的是[ICMP协议](#)，它是一个网络层协议并且没有端口的概念，因此出现上述的ping完全失败的结果也就不足为奇了。我们可以通过执行如下命令来验证nginx容器里工作的默认站点是否仍然如预期那样完美地提供服务：

```
# docker run --rm -link=a10e2dc0fdb:test -ti busybox wget
172.17.0.2
--2015-04-01 14:11:58-- http://172.17.0.2/
Connecting to 172.17.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 612 [text/html]
Saving to: 'index.html'

100%
[=====>]
612          --.-K/s   in 0s

2015-04-01 14:11:58 (16.4 MB/s) - 'index.html' saved [612/612]
```

还可以通过执行如下命令来检查DOCKER链的内容，从而查看Docker在容器被链接时创建的iptables规则：

```
# iptables -nL DOCKER
Chain DOCKER (1 references)
target      prot opt source                destination
ACCEPT      tcp  --  0.0.0.0/0              172.17.0.2
tcp dpt:80
ACCEPT      tcp  --  172.17.0.9             172.17.0.2
tcp dpt:443
ACCEPT      tcp  --  172.17.0.2             172.17.0.9
tcp spt:443
ACCEPT      tcp  --  172.17.0.9             172.17.0.2
tcp dpt:80
ACCEPT      tcp  --  172.17.0.2             172.17.0.9
```

就像我们之前所讲的那样，加固容器间通信的另外一种做法便是直接修改**DOCKER**链里的**iptables**规则。当然，建议最好不要这么做，因为当宿主机上运行了大量高密度容器的时候，这样做的话维护会是一个很大的问题，毕竟用户必须跟踪每个容器的生命周期，然后在容器交替迭代过程中不断地增删规则。

关于安全性另外一个很重要的点便是网络分段的问题。目前**Docker**在容器网络的分段方面没有提供任何原生的支持。所有容器之间的网络流量都是经过**docker0**网桥传输的。**Linux**里面的网桥是一种运行在混杂模式下的特殊的网络设备。这意味着任意一个在宿主机拥有**root**权限的人都有能力查看所有容器之间的网络交互的情况。这在多租户的环境下可能尤其需要注意，因此用户应该始终确保在自己的容器间两端往返的任意网络传输都是加密的。

正如之前所说，关于网络安全有说不完的内容。在这里，我们只是讨论了一些皮毛，并且重点放在一些基础概念上。现在，我们将继续下一个部分，探讨一下我们该如何完成跨**Docker**宿主机的容器间网络通信。

12.5.2 多主机的容器间通信

与之前一样，有几种选择可供我们实现这一目标。有了之前介绍的端口发布和容器链接方面的理论基础，我们可以推广到**Docker**社区里著名的大使模式（**ambassador pattern**），它巧妙地结合了这些概念。

大使模式的工作原理是在所有你想互联的**Docker**宿主机上运行一个特殊的容器，然后使用它来完成不同**Docker**宿主机之间容器的相互通信。这个特殊容器会运行一个**socat**网络代理，它负责**Docker**宿主机之间连接的代理转发。图12-4清晰地展现了这一具体过程。

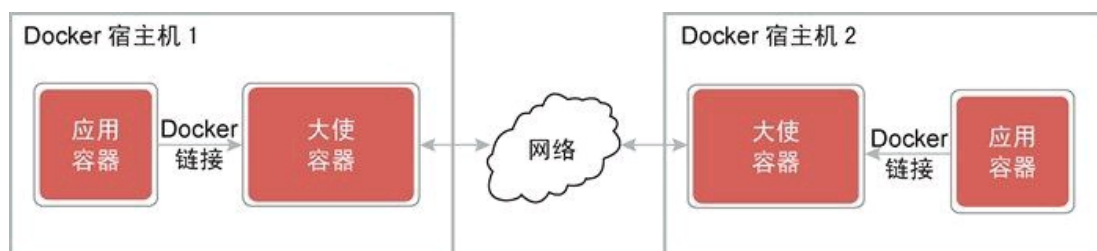


图12-4

读者可通过阅读Docker的[官方文档](#)来了解这一模式的更多内容。该模式核心理念可以归结为在宿主机上发布端口，然后通过链接到在其他宿主机上运行的大使容器，根据该宿主机上的环境变量找出发布并对我们公开的端口服务信息，以此来实现跨宿主机的容器间的网络通信。该模式的弊端在于，用户必须在每台宿主机上运行一个额外的容器来处理代理转发。而好处是，如果用户是通过容器名称而不是容器的ID引用目标容器，那么此模式有助于提高容器的可移植性。另外，当想导出一个容器并且把它迁移到其他的Docker宿主机时，如果那台宿主机刚好运行了一个容器，并且它和用户所导出的容器应该链接的那个容器名字相同的话，用户只需要启动那个迁移好的容器即可。

既然我们已经介绍了一些Docker原生提供的实现跨宿主机的容器间网络通信的解决方案，那么是时候来看一些更复杂的概念了。具体来说，我们将介绍如何在用户自己的专有内网里运行和集成容器服务。

Docker允许用户为自己想要运行的容器明确指定一个IP网段。正如之前所了解到的，容器的IP地址是从Docker守护进程启动时随机选择的一个内网IP网段里申请和分配的。如果用户希望将容器运行在自己的内网环境里，可以在Docker守护进程启动时传入一个特定参数来指定自己的IP网段。这似乎是在用户的私有内网里实现跨宿主机的容器间通信的首选方案。

必须首先为docker0网桥分配一个IP地址，然后再为Docker容器指定一个IP地址段。这个IP地址段必须是该网桥所在网段的一个子网。因此，如果我们想让我们的容器在192.168.20.0/24网络上，我们可以将DOCKER_OPTS环境变量设置成下面的值：

```
DOCKER_OPTS="--bip=192.168.20.5/24--fixed-cidr=192.168.20.0/25"
```

这看起来似乎是一个非常简便的实现Docker容器自定义内网配置的方法，实际上还需要为之在宿主机上添加一些特殊路由和iptables规则。这一方案的另外一个弊端或许你已经想到了，那便是IP地址的自动分配问题，这一点我们在之前讨论过。由于Docker守护进程之间不会通信，因此IP地址的分配可能会导致地址冲突。这也正是Docker期望解决的问题，在之后的版本如果引入了libnetwork的话，这个问题应该可以迎刃而解。在此期间，可以利用Docker的第三方工具的集成来解决这个问题。

针对Docker私有多主机网络的实现，业界还有一种流行的解决方案便是使用开放**虚拟交换机**（OVS）和**GRE隧道**。这背后的想法便是用OVS创建一个网桥来取代默认的**docker0**，然后在内网的宿主机之间创建一个安全的GRE隧道。这样做的话，用户至少需要对OVS如何运转有一些基本的了解。而且，希望互联的Docker宿主机越多，其复杂程度也越高（关于这一点，在Docker以后的版本里会通过引入**libnetwork**加以解决，或者也可以使用**Swarm**来规避这些问题）。默认情况下，使用OVS仍然无法解决我们早些时候提到的Docker跨多宿主机的IP地址分配的问题，因此用户需要采取一些额外的措施。关于如何将OVS用于Docker的更多内容请阅读这篇文章：<https://goldmann.pl/blog/2014/01/21/connecting-docker-containers-on-multiple-hosts/>。

Docker还提供了另外一个更为简单的方案来实现跨多主机的容器间网络通信：共享网络命名空间。

12.5.3 共享网络命名空间

共享网络命名空间的概念自**Kubernetes**项目发起以来开始被广泛大众接受，而它还有一个众所周知的名字，那便是**pod**。下面，我们将介绍如何将网络命名空间共享用于Docker宿主机的互联和通信。

当启动一个新容器时，可以为之指定一个特殊的“网络模式”。网络模式允许用户指定Docker守护进程采用何种方式为新建的容器创建网络命名空间。Docker客户端也提供了**--net**命令行参数来实现这一点。默认情况下它设置为**bridge**，这一点本章的开头部分便已经介绍过。

为了能够共享任意数量的容器之间的网络命名空间，我们传入**--net**命令行参数时必须遵循下面的格式：**container:NAME_or_ID**。

为了使用这一参数，必须事先创建好一个“源”容器，该容器将创建一个其他容器都能加入的“基础”网络命名空间。同样地，我们将通过一个简单的案例来详细说明这一点。

我们将会创建一个新容器，它将加入我们之前运行的**nginx**容器所创建的网络命名空间里。但是在此之前，让我们先列出**nginx**容器里所有的网络接口的情况：

```
# docker exec -it a10e2dc0fdfb ip link list
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN-
KNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
71: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT
    link/ether 02:42:ac:11:00:05 brd ff:ff:ff:ff:ff:ff
```

上述网络接口信息应该没什么特别的地方。现在，让我们新建一个新的
一次性容器，它将加入上述容器的网络空间里并随后列出它所有可用的
网络接口情况：

```
# docker run --rm -it --net=container:a10e2dc0fd9b busybox ip
link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN-
KNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
71: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT group default
    link/ether 02:42:ac:11:00:05 brd ff:ff:ff:ff:ff:ff
```

与文档里所描述的一样，这也正是我们所预见的情况：新创建的容器可
以看到并且绑定到**nginx**容器的网络接口上。新容器没有创建任何新的
网络命名空间——它所做的只是加入到一个现有的命名空间而已。

网络空间共享的优势在于它只占用Linux内核很小一部分的资源，并且
它可以作为一些场景下简单的网络管理的解决方案：它不需要用户关注
任何**iptables**的规则，并且如果在内网上已经建立了网络连接，那么
新进来的网络连接可以直接复用源自同一容器IP的连接。这一方案的另
一个优势在于用户可以通过回环接口与运行在容器里的服务通信。但
是，用户无法再把不同的服务绑定到相同的IP地址和端口上。

此外，如果源容器停止运行，用户必须重新创建它并让所有其他的容器
重新加入它建立的新的网络命名空间里：网络命名空间是不能循环利用
的！当然，用户可以通过创建一个“已命名”的网络命名空间来解决这
个问题。读者可以在<https://speakerdeck.com/gyre007/exploring-networking-in-linux-containers>这篇演讲中找到关于网络命名空间的更多内容。目
前，**Docker**并没有一个很好的办法来列举出所有共享了网络命名空间的
容器。

现在，我们对于**Docker**容器之间的网络命名空间共享已经有了一个很好

的认识，接下来，让我们一起来看看如何利用它来连接多个Docker宿主机以实现相互通信。窍门同样在于**--net**命令行参数的灵活运用。

Docker允许用户在创建容器的时候和它的宿主机共享网络命名空间。该宿主机本身在PID 1进程的命名空间里运行它的网络栈，这样的话，对于容器来说它们可以很轻松地加入宿主机的网络命名空间里。直白点儿讲便是：容器的网络栈和宿主机之间是共享的。

这就意味着加入宿主机的网络命名空间的容器将具备宿主机网络的只读权限（除非带上**--privileged**标志）。当用户在与宿主机共享网络命名空间的容器里启动一个新服务时，该服务会绑定到宿主机上任何可用的IP地址上，因此用户不必费多大劲便能实现它在宿主机上对外提供服务。

警告：尽量避免在宿主机上共享网络命名空间，因为这可能会导致用户的宿主机存在一些潜在的安全风险。在Docker的安全问题完全解决之前用户都必须时刻小心这一点！

下面，我们将通过一个非常简单的案例来加深对这一方案的实际理解。创建一个新容器，列出里面所有可用的网络接口，然后和宿主机上的网络接口做对比。首先，我们通过运行如下命令来列举宿主机上所有的网络接口：

```
# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP mode DEFAULT group default qlen 1000
    link/ether 04:01:47:4f:c6:01 brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
```

这里面没什么特殊的地方：我们有一个回环设备、一个以太网设备和一个Docker网桥。现在，让我们创建一个新的一次性容器，并且使得它和宿主机共享网络命名空间，然后列举该容器里所有可用的网络接口信息：

```
# docker run --rm --net=host -it busybox ip link list
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP mode DEFAULT group default qlen 1000
    link/ether 04:01:47:4f:c6:01 brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
```

如你所见，该容器没有创建任何新的网络接口，并且宿主机上的所有接口对它都是可见的。这样一来便轻松实现了宿主机之间容器的相互连接和通信，它只是省去了Docker私有网络这一部分而已。然而有利就有弊，这同样也使得用户的容器直接暴露在了网络上，并且抹去了很多在私有网络命名空间里运行容器时所具备的优势。

此外，当我们在宿主机上共享网络命名空间时必须留意一个看上去不是那么显眼的事情。网络命名空间里包含了Unix套接字，这意味着宿主机上所有各种操作系统服务所公开的Unix套接字同样对容器可见。这有时候可能会导致一些意想不到的事情发生。当用户在一个与宿主机共享网络命名空间的容器里准备关闭服务的时候切记这一点，并且尽量避免将这些容器运行在授权模式下。

上述情况的一个简单的可用场景便是当用户没有私有的SSH密钥时，在Docker构建时（build time）用户可以通过SSH客户端进行网络代理，如果将它打包到容器的镜像里会造成一些安全隐患（除非用户在每次构建后回收密钥）。这个问题的解决办法之一是运行一个socat代理容器，它与宿主机共享网络命名空间，然后访问宿主机上的ssh-agent。随后该代理将会监听一个TCP端口，在构建时用户可以通过这个端口代理所有的内部传输。在这篇[gist](#)中你可以看到和这相关的一个具体实例。

关于网络命名空间共享更安全的一个版本是先照常创建一个网络命名空间，然后使用`ip link set netns`命令把宿主机上的接口移到新的网络命名空间中，随后在容器里配置它。目前，Docker还没有对此提供任何命令行方面的支持。

这种类型的配置在大多数的云环境下都无法正常运行，因为它们很少会提供多个物理网络端口，并且也不接受在现有的端口里添加额外的MAC地址。然而，如果在带有多个物理端口的物理硬件下运行，像SR-

IOV端口、VLAN或者macvtap接口，这种办法应该是可行的。

12.6 IPv6

大多数的Docker部署都是使用标准的IPv4，但是许多大型数据中心的运营商，像Google和Facebook，在内部已经开始转向使用新的IPv6协议。IPv6地址不会短缺，它的地址位长足足有128位，而且为单个主机分配的IP地址是一个/64，或者是18 446 744 073 709 551 616个地址！

这意味着完全可以为每个容器分配一个它自己的全局可路由的IPv6地址，这也代表着跨主机的容器之间的通信会变得简单很多。

当前版本的Docker已经加入了对IPv6的支持，并且相应的[配置文档](#)也已经整理了出来。Docker需要为容器分配一个至少/80的地址段，这意味着它可以在一个/64位地址的宿主机上工作，或者甚至在更大规模的时候还可以把单个的/64段分到不同的多台宿主机上，在这个配置下理论上最多可以有4096台宿主机。不同宿主机之间的容器甚至可以直路由。

同样，在云环境下，至今仍然鲜有提供IPv6支持的厂商，但是这一点在未来有望得到改善。Digital Ocean每台宿主机上只提供了最小仅16个的IPv6地址段，虽然它宣称支持这一功能，但是实际实施起来还是有一定的难度，关于启用IPv6地址支持的方法，在其官方文档里的NDP代理部分有相应介绍。然而，也有像来自Bytemark的BigV和Vultr，他们提供的云服务支持一个全长的/64段IPv6的支持，有了它们，为Docker引入IPv6的支持也将不再是一件难事。

IPv4地址的短缺问题迫使更多的人把目光放到了IPv6的身上，它提供了一种没有NAT回归简单的网络拓扑结构，但是由于缺乏广泛的支持，IPv6的实际应用仍然存在问题。

12.7 小结

正如本章所介绍的，**Docker**引擎提供了大量可供选择的方案来帮助连接基础架构中的容器。遗憾的是，更高级的配置需要大量的人力投入，并且需要用户对于像路由和**iptables**这样的网络内部原理有深刻的理解，而这对于只想运行应用而不关心底层网络细节的用户却是一件很痛苦的事情。此外，**Docker**提供的原生参数是相对静态的，并且在跨多宿主主机方面的扩展性并不是很好。我们希望所有的问题最终都能通过将在新版**Docker**里发布的网络API来解决，该API基于早前提过的**libnetwork**实现。当然，在实际发布前也不必太过担心，快速迭代的**Docker**生态环境会帮你解决遇到的问题。

在标准的**Docker**网络试图将网络配置抽象化的同时，使用**iptables**以及潜在的隧道协议是有性能开销的。**New Relic**发现在某些情况下，对于需要高性能网络的应用程序来说，使用标准的**Docker**网络配置也许会比原生的本地网络要慢上20多倍。

业界已经有大量可供使用的工具来解决上述讨论的**Docker**网络方面的问题。最简单的，但也是非常强大的便是一个由**Docker**公司的**Jérôme Petazzoni**独立开发的名为**pipework**的工具。**pipework**本质上是一个简单的shell脚本，它支持像容器多IP地址配置等高级网络配置，允许使用Mac VLAN设备甚至是DHCP来完成IP地址的分配。甚至有人制作了一个专门的**Docker**镜像，使得可以在容器中运行**pipework**。可以利用**docker compose**将其整合到应用程序中。

另一个非常有用的工具是由Weaveworks公司提供的，名为**weave**。如果详细介绍**weave**可能需要一个单独的章节，但是在这里，我们只提供一些简单的介绍，读者可以通过本节提到的各种链接地址进一步了解与它相关的内容。**weave**可以跨多宿主机和云服务创建覆盖网络，如此一来用户便可以轻松地连接到各个**Docker**容器。**weave**还提供了一些非常强大的功能，如安全性方面的**增强**和早前提到过的**weave-dns**，它实现了一个简易的基于DNS的服务发现。根据经验来看，**weave**算是市场上目前最易上手的工具，因为它不需要用户考虑太多底层网络方面的细节。该企业最近发布了一些非常有意思的功能，如**weave scope**，这是一个为容器提供更好的可视化效果的工具，它在高密度容器的环境里可以说是无价之宝。读者可以通过这篇[文章](<http://thenewstack.io/how-to-detect-map-and-monitor-docker-containers-with-weave-scope-from-weaveworks/>)对**scope**有一个大致的了解。**weave**版本1.0发布了一个重大特性，即**快速网络路径**，它是基于我们之前讨论过的OVS实现的。使用该快速路径就

意味着用户的网络不再是强加密的，但是这仍然是一个值得权衡的事情。最后，随着Docker近期发布的[Docker插件系统](#)的支持，原生Docker的集成也变得更加容易，用户只要把Weave当做[Docker的插件](#)使用就行了。

CoreOS也开发了一个名为[flannel](#)的覆盖网络解决方案。flannel最开始开发是为了解决Kubernetes原生依靠Google云平台为中央网络的网络依赖问题，但是自第一个版本发布以来该项目发生了很大的转变，如今它提供了一些如[VXLAN](#)这样很有意思的特性。flannel将它的配置信息存储在[etcd](#)中，这使得它可以和CoreOS操作系统轻松地整合在一起。

最后，在Docker网络领域最新的一员是一个名为[calico](#)的项目，它和OpenStack一样为Docker提供了一个3层的解决方案。目前该项目利用的是ClusterHQ的[powerstrip](#)工具，使用该工具可以在Docker引擎的原生插件实现前完成简单的插件注册功能。calico项目的最大优势便是它提供了原生的IPv6支持，并且易于在多Docker宿主机之间扩展。calico本身是通过某种跨多宿主机的分布式BGP路由来实现的这一点。这是一个值得长期关注的项目。

介绍完这些第三方工具之后，我们也将结束本次的Docker网络之旅。希望这里所描述的内容能够真正地帮助你更好地了解该如何建立适合自己的Docker基础设施的网络模型。

现在，我们将继续前行，在第13章中我们将讨论如何完成跨多宿主机的Docker容器的调度。

[1] 从1.10版起，Docker会根据镜像及镜像层数据的安全散列生成一个内容可寻址的安全ID。——译者注

[2] 在本书纸版发行前，Docker v1.9已经发布，实现了跨主机的网络互通，并且支持不同的插件方式。——译者注

[3] [icc](#)是Inter Container Communication的缩写。——译者注

第13章 调度

到目前为止，我们已经看了很多在单台主机上运行Docker的案例。然而，你的基础设施必定会在某一刻增长到单台宿主机不足以承载其服务的程度或者说你想要在一些高可用的环境下运行Docker容器。这时候就需要将它扩展到多台机器上。这会带来很多的挑战，不仅在于多宿主机的Docker网络配置，更重要的是容器的编排和集群的管理。

如果没有合适的集群和编排的管理工具，你会发现要么是自己被容器基础设施所绑架，深陷于繁琐的日常事务中，而不是管理和操纵它，要么便是重复造轮子，最终写了很多自己的容器管理工具。事实上，Docker已经意识到了这方面的需求并投入了巨大的精力到它的网络模型的改造当中，而且还创建了一个名为Swarm的Docker原生集群管理工具。

关于集群管理这个话题，如果要深入讲解的话可以单独写一本书了。这里面包含了大量需要关注的内容：如何增删宿主机，如何确保它们的健康状态，以及如何伸缩扩展来管理负载等。在本章中，我们将把重点放在集群管理和容器编排中一个特别关键的部分：调度。

作业调度的基本理念很简单，而且它的历史可以追溯到大型机与高性能计算（HPC）。与其把计算机的资源投入到各自特定的工作负载，还不如将整个数据中心看做是一个巨大的计算和存储的资源池，然后只需要在上面设置需要运行的计算作业即可，就像它是一台巨大的计算机一样。事实上，Mesosphere作为Docker领域里基础设施的一员，它们的主打产品正是“数据中心操作系统”。

现在，让我们先把具体实现方面的内容放一放，把重点放在根本问题上。为了更好的理解Docker主机集群里容器调度问题的本质，我们需要先做一些铺垫工作，并且了解一下调度的真正意义。

13.1 什么是调度

如果给调度问题作一个30秒的简单概括，我们就可以说调度问题是为一组计算任务分配一组硬件资源（如CPU、内存、存储和网络容量），在满足任务需求的同时完成这些任务。

可以影响调度决策的因素有很多。执行成本显然是需要考虑进去的，但是我們也需要考量延迟、吞吐量、错误率、完成时间以及服务的交付质量等因素。因此需要根据重要程度（面向用户）为任务排优先级，或者在调度一些带有明显负载波动的关键作业时安排较少的时间窗口。

每个任务可能由多个进程组成，对应的是不同的数据存储和网络需求，网络需求也许还有一定的网络带宽或者地域要求，因此用户得把它们放在靠近彼此的区域，由于可能还会有一些冗余的要求，所以用户还得在其他地方存放几个冗余的副本，如放到另一个数据中心或者可用区。

应用所需的资源量在不同时间点会因为负载的不同发生变化，由于它们做过扩容和缩容，因此需要根据其扩容或缩容做重新调度。此外，随着时间的推移，新的应用也将被部署到环境里，因此必须定期重新考虑调度策略。有些任务可能只是临时执行的，而有些则是长期运行的，如果基于生命周期来分配资源也能产生一定的收益。

度量和监控是关键：这是吞吐量及延迟与需求之间的博弈，资源利用率和容量的规划很重要，但是同样也得考虑成本和预算，还有那些即将产生的价值。当用户把调优作为最终目标（如最大化吞吐量、最低延迟等）的时候，调度问题本身就成了一类[调优问题](#)。

通过几十年来的不断优化，操作系统这一层的调度已经做得相当出色了，因此在绝大多数情况下，我们所做的调度都是基于单台主机的。就目前而言，跨主机的调度还远未成熟。

与虚拟机（VM）的调度不同，容器通常都不固定大小，它们随时可以修改自己的内存使用量。这使得容器的调度比将一些固定大小的虚拟机装配到物理裸机的[装箱问题](#)要困难得多。

最优调度是一个非常棘手的问题，只有在一个非常大的规模下投入巨大的工程人力才可能看到收益。

13.2 调度策略

正如我们之前提到的那样，与单台主机的调度问题相比，跨集群之间的调度会更加复杂。单台主机的调度重点关注少数CPU上运行大量线程和进程的问题，其目的在于避免资源分配不均，保证单个进程不会运行太长的时间，并且确保交互进程可以在超时之前命中资源。除此之外，它还主张一些资源利用方面的公平调度，如引入IO带宽等概念。在NUMA系统中，隔离性也会被考虑进去，但它不是主要的驱动因素。

在单个或者多个数据中心中，乃至在云环境下，该问题的规模将会扩大好几个数量级。延迟会变得至关重要，主机在执行作业时会出现异常，并且每个任务的时间跨度都会相应变长。此外，任务的负载强度方面会有很大的差异，有Hadoop的MapReduce作业，其工作量一般覆盖到整个集群；也有MPI类型的作业，它们需要在任务开始之前准备好相应的可用资源；传统的Web应用，这些通常只要提供的容量超过它们所需即可。

因此，一个集群里容器的调度真的是一个很令人头疼的问题，这涉及到了异构环境和不同的工作负载两方面。这个世界不存在“一刀切”的完美方案，因此各类机构创建了各式各样一整套的调度工具，它们之中的绝大多数均是从自身的角度出发致力于解决一类特定的负载问题。

13.3 Mesos

Mesos是由加利福尼亚大学伯克利实验室的研究人员开发的一款特别有意思的解决方案，该实验室给业界创造了太多的惊喜，而这次它依然没有让业界失望。Mesos以一个高级抽象的概念实现了所谓的二级调度，即Mesos本身在计算集群里提供了一层对底层计算资源的抽象，随后再把它们提供[或者说调度]给任意数量的在它之上运行的[异构]框架服

务。然后每个框架再根据它自己特定的应用场景实现属于它自己的一套任务调度策略。

这意味着Mesos只维护集群的资源，然后基于优势资源[公平算法](#)决定把哪些资源分配给哪个框架服务。重要的是Mesos将实际的任务调度委托给了特定的框架服务，就这一点来说自有其利弊。

这一模型为用户提供了很大的灵活性，用户无需再操心底层的计算资源的处理，而可以根据特定的应用场景专心地编写属于自己的、合适的任务调度器。从另一方面来说，用户无法看到整个集群状态的简要概况。除此之外，在Mesos认为资源分配是相对公平的情况下，用户也无法在自己需要它们的时候强制让Mesos为用户分配更多的资源。Mesos评估资源公平的标准是基于框架启动时提交给Mesos资源分配器的一些指标。

当用户运行了大量长期运行的任务时事情可能会变得比较麻烦，它们可能会占用大量的计算资源并因此导致那些尚未调度的任务因为没有资源消费而阻塞。换句话说：Mesos在用于同构的批量处理型应用时表现得最好，这一点已经成功地被像[Apache Spark](#)这样的项目所验证，它在Mesos之上实现了一个异常快速的数据处理框架。

运行在Mesos之上的框架服务，它们所调度的计算任务实际上是由Mesos称之为容器类^[1]（Containerizer）的进程来调度的。Mesos自0.20版本起，便在容器类的选择方面加入了对Docker的支持。如今Mesos仍然还没有完全实现对Docker的全面支持，但是该版本实现的功能已经足以用来调度一些复杂的Docker任务。

[Twitter](#)证明了Mesos可以成功驾驭大规模基础设施下的异构应用。他们已经为长期运行的服务开发了一个叫[Aurora](#)的框架，它支持资源抢占和其他许多实用的功能。业界另一个很受欢迎的框架是由[Mesosphere](#)开发的[Marathon](#)，它提供了一个远程的REST API来方便用户编排。

如果想了解有关Mesos的更多详细内容，不妨去读一读它的这篇[白皮书](#)。

13.4 Kubernetes

与Mesos相比，Kubernetes算是一个非常年轻的产品，它2014年6月才开始启动，在2015年7月发布了1.0版本。然而，它有一段很长的发展历史，实际上它是Google自Omega和之后的Borg这样的内部调度系统创建的一个开源版本。

这篇由Google发表的关于Borg的论文简要讲述了该产品宏大的发展历程，还有Google是如何在内部基于容器的架构来完成调度的。凭借如此宏大的背景，Kubernetes自然也引起了很多人的注意，而如今它已经被CoreOS和Red Hat这样的企业开发和采纳。

Kubernetes覆盖了应用部署涉及的整个范畴：调度、更新、维护和扩展。它支持Docker容器，并且将它们归类到名为“pod”的一个个小集群的组，“pod”共享一个网络命名空间，如此一来它们之间的网络通信就变得简单很多；这也就意味着一个小组的容器可以共同组成一个单独的应用，如Web应用和Redis存储。

使用Kubernetes的API可以管理容器、pod、备份、卷、私密信息、元数据以及所有其他组件，而且支持多方面的用途，这一点也正是Kubernetes的主打功能。由于该API的设计原型源自于Google的早期项目，因此可以说它已经经受了很好的实际考验。

Kubernetes目前还不是一个易于安装的组件。业界比较清晰的教程当属Red Hat的Kubernetes[初学向导](#)了，然而这里面仍然有很多手动配置的步骤。引入Docker可能会减轻必要的配置管理的工作量，但是为它构建一个完整的调度环境仍然需要花费很大力气。

到目前为止，最容易上手的体验方案当然还是使用Google容器引擎的实现。有一篇必看的[Wordpress入门教程](#)，快速、简单、易于上手，在部署完成时它将会为用户安装和配置好Kubernetes标准的“kubectl”命令行工具，如此一来用户便可以在集群里轻松运行属于自己的Docker容器了。

13.5 OpenShift

调度系统可以说是平台即服务（PaaS）的一块基石，而Kubernetes正成为这一领域里的核心成员。

Red Hat的OpenShift是一个开源（并且已经托管了）的平台即服务的产物。在它发行版本3之前它可以说是一个相当于标准的开源版的“Heroku”PaaS产品，但是版本3是在Atomic项目，特别是Docker和Kubernetes的基础之上做了一次完全重写，Atomic项目本身是Red Hat应对CoreOS的举措，它是其他一大堆开源项目的载体。

OpenShift于2015年6月发布后全面上市，当你希望得到的是一个完全成熟的PaaS时，它会是一个很好的切入Kubernetes的方法。

Red Hat公司首席工程师Clayton Coleman的想法

OpenShift是怎样适配容器的？为了获得一个完整的了解，我们专门采访了Red Hat公司OpenShift项目的首席工程师Clayton Coleman先生。

记者：“OpenShift v3是否真的是一款自吹自擂的产品又或者说是一堆开源项目打包在一起的可以随时被替换掉的‘杰作’？显然在某种程度上来说它是的，但是我更感兴趣的是它是如何在面向你的客户的产品以及面向你的开发过程的工具两者间协调工作的。”

Clayton Coleman：“OpenShift是一系列建立在Kubernetes之上的部署、构建和应用生命周期管理工具的集合。对于每个单独的组件，我们尽量不去采用那些明摆着非常偏门的技术。例如，针对边缘路由^[2]和代理我们写了一个通用的模式，它兼容了Apache、Nginx、F5和许多其他的可配置的负载均衡器，但是在默认情况下我们还是采用最常见的HAProxy来配置。我们自然希望可以从多方途径获取支持，但是我们目前主要还是采用Git。Kubernetes正在加入对Rocket容器引擎的支持，因此OpenShift可以利用这一点，也许最终可以替代Docker。我们试图成为众多不同的概念和开发工作流之间关联的纽带，而不是假定所有应用都是一个自顶向下的生命周期。”

记者：“PaaS如何才能在企业里真正落地？谁是最早的先驱，在此过程中又作了哪些决策？容器和Docker的加入会加快推进的步伐吗？显然就目前而言，它是一个很大的革新，并且这中间变化的过程也相当有趣。”

Clayton Coleman: “企业内的PaaS一般都是由大量的内部应用的部署成本所驱动，他们希望借助PaaS可以将部署模式和流程标准化，并且希望可以一定程度地提高基础设施和工具的灵活性。我想Docker带来的更多的是自下而上的变革，原因在于它是一项低门槛的技术，对于一些简单的任务它可以完成的很好。我们已经看到许多企业将Docker应用到了平常频繁接触的操作流程里，在这里运维和开发团队在以镜像为原子单位的应用部署中受益。当然，它也有不利的一面。在Docker引入容器带来巨大的提升的同时仍然需要实施大量的定制/脚本化的流程，相比之下我们认为引入更大规模的集群管理模式（通过集群管理/PaaS工具）可以带来更大的收益（一个是呈倍数级的改善，一个则是附加增量形式的迭代）。 ”

记者：“你把Kubernetes里的pod模型描述成一个微型的虚拟机，例如，一组应用通常是通过磁盘和Unix套接字在本地进行通信来协同工作的，而不是通过远程的网络IO传输。有没有什么工具可以用来做这种结构的现有应用的迁移，将它们按照现有的[Red Hat容器安全标准](#)的规格改造成具备更高安全性的标准结构？所以，你有没有看到过一个现有应用的实际迁移的案例，或者说业界有没有大规模在PaaS上运行新的应用的案例？”

Clayton Coleman: “如今我们所看到的结构是新旧模式之间^[3]的分裂，因此选择其中任何一种应用模式都会趋向于在某方面设计的过分简单化，这就导致它在另外一种模式下很难正常运转。一个好应用的组成应该比‘仅仅只是一个容器’要复杂得多，并且它们通常需要抽象的是共享的磁盘资源、本地网络，或者是不应该耦合到主容器里的‘侧舱’形式的客户端。因此从第一天萌生组建它的想法时你就需要建立一个本地聚合的概念来帮助将新旧应用模型有机的结合到一起。”

记者：“不可变基础架构和Atomic项目都是时下非常新兴和热门的，在你看来，对不可变基础架构的市场需求有多大？你如何看待这个趋势？”

Clayton Coleman: “为了使得基础架构的不可变性真正落地，你必须以一种可重复的方式来不断地实施变更。新的不可变镜像的流动完全取决于一个自动化的且易于理解的构建流。不可变基础架构的优点是减少了重现问题或者跟踪故障时需要理解的变量的数量，但是反过来，不可变基础架构本身的成本在于将一切事物自动化。”

记者：“其他的一些厂商似乎在推容器遗留的一些东西，即在一个完全可变的系统里运行多个应用，而Red Hat似乎是想推动一个应用对应一个容器的结构，必要的话再围绕着构建一系列的pod。这样的描述正确吗？这是否是由客户需求驱动的呢？”

Clayton Coleman: “有时候，构建一些较复杂的容器是很有必要并且也很有价值的。一般来说，减少复杂度是一个很合理的做法，但是常常会有一些实际的需求必须通过一些复杂的容器来满足，而我们仍然可以从围绕着它们所构建的弹性基础设施中受益。我认为容器化的基础设施所取得的收益在很大程度上取决于你所处的环境将责任域切分成多个子组件的实现能力（这便是人们提倡的SOA，或者说最近比较热门的微服务概念）。但是这仍然不能排除不存在大的、可变的，或者复杂的容器。”

在第14章里，我们将介绍Docker中的服务发现，它可以使一个计算机网络上的任意服务都可以找到它需要通信的其他服务。

[1] 它是Mesos Slave节点的一部分，可以将其理解为是Mesos中任务的容器。——译者注

[2] 将客户连接到因特网的路由器被称为边缘路由器。——译者注

[3] 即把Docker当做简单的微型虚拟机和完全容器化的应用架构两套模式。——译者注

第14章 服务发现

服务发现是一种容许计算机网络上的任意服务均可以找到它所需要通信的其他服务的机制。它是大多数分布式系统的核心组件。如果用户的基础设施是运行或者遵循的面向服务的**架构**（Service Oriented Architecture, SOA），那么毫无疑问，用户需要部署某种服务发现方案。同时，服务发现也是软件应用设计方面的一个新兴概念，这和传统的SOA有许多的相似之处，而如今它有一个更广为人知的名字叫做**微服务**。

服务发现的定义相当简单（如图14-1所示）：一个客户端如何才能找到它想要与之通信的**IP**地址和服务端口？

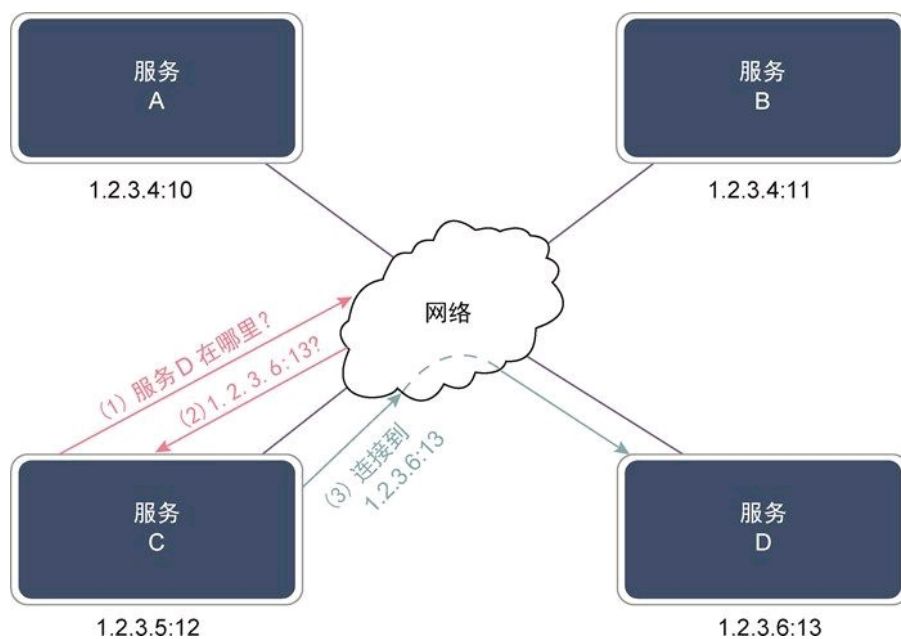


图14-1

针对这个问题，不同的解决方案往往隐藏了很多用户不知道的微妙之处。为了能更好地理解服务发现的理念，我们首先至少需要为它约定一

些基本的要求。一个服务发现方案必须遵循的最低要求如下。

- 服务注册/服务声明：即服务在它所属的网络上声明自己存在的过程。其通常做法是在某种服务数据库里加上这行记录，该数据库通常也被称为服务中心或服务目录。服务中心里的每行条目必须至少包含服务的IP地址和端口信息，最好还包含服务的一些元数据，如使用的协议、具体的环境、版本等。
- 服务查找/服务发现：即从网络上找出想要与之通信的服务的具体连接信息的过程。该过程具体可以归结为查询服务目录数据库然后找出给定服务的具体IP地址和端口信息。理想情况下，应该可以通过不同的维度如上述所提到的几个元数据，来查询服务目录里的数据。

服务注册的过程其实比我们上述所提到的情况要复杂许多。一般来说，有以下两种实现方式。

- 把服务注册的模块直接嵌入到应用服务的源代码里。
- 使用一个伙伴进程（或者说是协同进程）来帮忙处理注册的任务。

将服务注册嵌入到应用源代码中，这会对客户端的库有一定的要求。一般来说，一个特定编程语言可用的客户端的库也是有限的，因此为了实现这一点，用户可能会写很多额外的代码，这可能会给用户的代码库引入很多不必要的复杂度。在应用的源代码里嵌入服务发现的代码常常会导致产生所谓的胖客户端，这部分代码不易编写，并且常常会造成调试方面的困扰。这一方案会将用户绑定到一个特定的服务发现方案上，这可能会导致用户的应用和服务缺乏一定的可移植性。最后，如果用户想要在自己的基础设施上运行和集成一些第三方的服务，如redis，那么可能需要费些力气。但是，如果能够建立一套稳定的、积极维护的方案，用户可以从这里面得到很多好处，如客户端服务的负载均衡、连接的池化、自动的服务心跳检测和故障迁移等。

另一种被广泛采用的服务注册的方案是在应用服务一侧运行一个伙伴进程，然后用它来代表应用服务的注册。这一方案的好处也是显而易见的。它非常实用的一点在于不需要应用程序的作者为此编写任何额外的代码。通过把伙伴进程集成到init系统^[1]里，可以确保（一定程度上来说）服务只有当它完全启动并且正常运行时候方才注册。采用这种方案来注册服务同样也使它可以轻松地和第三方服务整合到一起。但是，它的缺点在于用户不得不为每个需要注册的服务运行一个单独的进程。

此外，使用伙伴进程也会对实际的服务注册过程提出额外的要求：你怎么为自己的伙伴进程提供你想要注册的服务的配置呢？由于服务注册经常需要更新应用服务的一些元数据，因此了解应用的配置是非常有必要的。显然，伙伴进程这个方法会带来一些好处，但是它也同样会引入一些运维方面的挑战。

基础设施越复杂，我们要求放到服务目录数据库里的数据也会越多。一个更加全面的解决方案理应该可以提供更多的内容（与之前的两种方案相比），特别是在以下几个方面：

- 服务目录数据库的高可用性；
- 能够轻松将服务目录数据库扩展到多台主机，同时保证一定级别的数据一致性；
- 告知相关服务的可用性，以便在服务不可用时及时地将它从服务目录中删除；
- 告知一些特定服务目录条目的变更，如当一些服务的元数据变更时。

什么，使用Docker是不是这些事情都得做？好吧，实际上Docker会使服务发现的问题更加明显。特别是在开始将容器基础设施扩展到多台宿主机的时候这一点更为关键。一旦用户开始使用Docker来打包和运行应用，用户会发现自己不停地查找运行在Docker容器里的特定服务正在监听的端口和IP地址。幸运的是，Docker使得查找服务连接信息变得很简单：用户需要做的只是通过Docker守护进程去查询一下远程API暴露给自己的信息。很多时候用户最终选择的是编写一些简单的shell脚本，然后将服务连接的具体信息以环境变量的形式通过Docker API传入新容器里。虽然对于本地环境而言这是一个不错的便捷方法，但是它不是一个易于扩展和可持续的方案。这不仅在于定制脚本的维护的艰难，更主要的是当应用或服务实例分布到多台宿主机或者多个数据中心时情况会变得更加糟糕。这个问题在云环境这样的服务更替频繁的场景下会暴露得更加明显。

在本章中，我们会尽力覆盖到服务发现这个话题的方方面面，并且从实际情况出发，探讨在运行Docker容器时可以采用的多种开源方案。那么，打起精神来，让我们开始吧。

14.1 DNS服务发现

DNS是支撑起万维网的核心技术之一。从整体上来说，DNS是一个主要用于将人类可读的名称解析为机器可读的IP地址的一致性的分布式数据库。众所周知，它也被用于查找负责一个指定域的电子邮件服务器。用DNS做服务发现似乎是一个很自然的选择，因为它一个已经被充分检验过、被广泛部署并且非常易于理解的技术。对此，业界有丰富的可供选择的开源服务的实现，与此同时几乎能想到的任何一门编程语言都会提供相当不错的全套客户端库。DNS支持客户端缓存和基础的域名代理，这也使得它成为一个可扩展性很强的解决方案。

DNS最易于理解的部分莫过于早前所提到的通过DNS的A记录实现从域名解析到IP地址的过程。然而只使用A记录做服务发现是远远不够的，因为它们不会提供指定服务所监听端口的任何信息。此外，A记录无法提供关于运行在用户的基础设施里的服务的任何元数据信息。为了能够用DNS实现全功能的服务发现，我们至少需要用到以下两个额外的DNS记录：

- **SRV**——通常用来提供网络上的服务位置信息，如端口号等；
- **TXT**——通常用于提供多个服务的元数据信息，如环境，版本等。

这些资源记录可以说是使用DNS作为服务发现的解决方案的最低要求。当然也可以根据国际惯例在公知端口号^[2]下运行服务，这样一来便可以简单地只使用A记录来实现服务发现，然而，这样一来用户将无法完成一些自己本来需要的复杂查询操作。

注册和注销服务时需要在DNS服务器配置里添加或删除指定的DNS记录，并且常常需要重新加载一次服务以使得变动生效。用户必须实现一定程度的自动化，以保持DNS服务端的配置和线上自己基础设施里正在运行的应用服务的配置是一致的。DNS诞生于一个相对“静态”的互联网时代，与如今新的“云时代”下服务器和服务的更替都非常频繁的情况相比，当时它并不需要频繁地去更新DNS记录。由于DNS基础设施采取的是多层缓存机制，因此DNS记录的修改也需要一定的传播时间才能使之完全生效。用户常常试图采取把TTL值降低到最小的方法来解决这个传播时间的问题，但这样做的话可能会产生太多不必要的网络流量，进而

拖慢通信的速度，并且会因为服务的查找动作过于频繁而给DNS服务器增加额外的不必要的负载。

与经过实战检验的[bind](#)服务相比，虽然业界一些著名的DNS服务器实现在配置方面采取了更加灵活的方式，但是用户常常不愿意运行他们自己的DNS服务器，因为除了原有方案已经实现的大量的自动化工作以外，他们还需要花费相当大的运营和维护成本来维护它们。云服务商们还提供了很丰富的带有简单API控制的DNS服务，如AWS的[Route53](#)，使用它们的话可能需要花费额外的精力来编写和维护代码，这样看来这些方案也不是非常可取的，并且它们始终没有解决服务繁杂的问题。的确，天上可不会掉馅饼。

随着Docker和微服务架构的兴起，DNS服务器也从现有模式中衍生出了新品种。这些DNS服务器解决了部分之前讨论过的问题，而且常常可以被用来提供简单的服务发现，并且它的服务对象还不只是运行在Docker容器里的服务。在下一节里，我们将介绍一些最广为人知的实现方案，并且探讨下应该如何在基础设施里使用它们。

DNS服务器的重新发明

最广为人知的“新生代”DNS服务实现之一是一款名为[SkyDNS](#)的软件，它可以被用在基础设施里实现服务发现。可以从源码编译它也可以将它打包成Docker容器部署。可以从[Docker Hub](#)上找到它的Docker镜像。SkyDNS的最新版本采用的是[etcd](#)服务来存储它的DNS记录。我们会在本章的稍后部分详细介绍[etcd](#)。现在，让我们姑且认为[etcd](#)是一个分布式的键值对存储吧。

SkyDNS提供了一个远程的JSON API，它允许用户通过发送HTTP POST请求到指定的API端点的方式来动态地完成服务的注册。这样的话它会自动创建一个SRVDNS记录，正如我们之前所了解的，它可以被用于发现在网络上运行的服务的连接端口信息。自己也可以将TTL设置成任意值，如此一来，一旦TTL过期了它会自动注销该服务。SkyDNS也提供了[DNSSEC](#)的功能。要想设置它的话自己需要做一些额外的配置。可以在GitHub上阅读该项目的[文档](#)。

如果想把SkyDNS和Docker一起使用，就得写一个专门的SkyDNS客户端库来帮忙完成服务的注册。幸运的是，正如前面提到的，多亏SkyDNS

本身提供远程API，这应该不会是一个大问题。一旦服务完成了注册，便可以用任一DNS库来查询它的具体信息。丰富的开源DNS库在这时体现出了巨大的价值。一个更简单的使用SkyDNS集成到Docker的方案便是使用Docker公司的Michael Crosby开发的skydock，虽然目前来说它的可扩展性还不是很强。Skydock将会监听Docker API的事件，然后自动地为用户处理所有Docker容器的服务注册工作：会帮用户自动完成服务的注册和注销。Skydock的唯一问题便是它目前只能用在一台Docker宿主机上，然而，这一点可能在以后会得到改善。Skydock的确是一款值得密切关注的工具。如果想找到更多关于Skydock以及它是如何应用到Docker容器的内容，这里有一个非常棒的由Michael贡献的YouTube视频，他会带你领略所有的Skydock特性。

Docker DNS服务器领域的另外一员便是weave-dns。weave-dns最大的好处在于用户不必花费太多精力就能使用它提供的一些开箱即用的功能。和Skydock一样，它会监听Docker API的事件然后通过自动添加和删除特定的DNS记录来完成容器服务的注册。与Skydock只能在一台Docker宿主机上使用不同，weave-dns允许跨多台Docker宿主机工作。然而，如果想要充分发挥它的优势，就必须在专有的weave覆盖网络上使用它，这对于一些用户而言可能是一个不太好的消息。weave网络是一个软件定义网络（SDN），它可以为用户提供一个简单而安全的跨多台Docker宿主机的覆盖网络，然而，如果用户已经使用了其他的SDN方案，那么weave-dns也许不是一个最佳选择。此外，目前weave-dns依赖于用户使用公知端口号来提供服务，而不是查询SRV或TXT记录。

随着Docker生态系统的快速扩张和成长，以后可能还会有更多可供使用的DNS服务器实现，有的可能作为独立的服务，有的可能是作为成熟的SDN产品的一部分提供。由于篇幅的限制，本书不可能覆盖到所有可供选择的方案，其他的备选方案就留待读者自己继续挖掘和探索。接下来我们将介绍一些已经成为分布式系统领域事实标准的服务发现方案。我们将从著名的Zookeeper项目开始。

14.2 Zookeeper

Zookeeper是一个Apache基金会的项目，它为分布式系统提供分布式的协同服务。它还提供了简单的基本授权功能，允许客户端建立更加复杂

的协同功能。Zookeeper旨在成为一个分布式服务的核心协调组件或者是搭建强大的分布式应用时的一个基础组件。关于Zookeeper的介绍我们甚至可以写一整本书来讲解，而且实际上这样的书已经有了。在这里我们将不再探讨那些基础的概念了，取而代之的是，我们将会把重点放在如何在基础设施里采用Zookeeper作为用户的服务发现方案。

从整体上来说，Zookeeper提供了一个名为**znode**的分布式内存数据存储寄存器。它们以类似标准文件系统的组织方式存放在分级的命名空间里。**znode**这样的层次结构通常被称为“数据树”。**znode**通常有两种类型：

- 普通的——可以被客户端显式创建和删除；
- 临时的——与普通的一样，此外客户端还可以选择委托，一旦客户端会话终止便会自动把它们从集群里删除。

客户端可以在集群里的任意**znode**上设置监听，它可以让Zookeeper自动地通知客户端任何数据上的更改或删除。可以说，Zookeeper最大的优势之一在于它提供的API的简洁性：它只提供了7种**znode**操作。借助于Zookeeper原子广播（ZAB）一致性算法的实现，Zookeeper提供了健壮的数据一致性保证和分区容忍性。简单来说，ZAB定义了一个领导者和一群追随者（他们可以共同选举出领导者）。所有的写请求都会转发到领导者这边，随后领导者会将它们应用到系统中。读请求则可以被追随者们消费。Zookeeper只能在服务器的法定人数（大多数）都正常的情况下正常工作，因此用户必须保证Zookeeper集群的部署数量总是保持在3、5等这样的奇数单位。或者更官方地说，用户必须保证运行的集群有 $2n+1$ 个节点（ n 是一个代表服务器数量的正整数）。这样规模的集群可以容忍 n 个节点的故障。前面所提到的属性对Zookeeper的可扩展性有一定的影响。增加新节点可以提高读的吞吐量，但是会降低写的吞吐能力。此外，当仲裁发生时，它必须等待远程站点选举领导者的投票结束，这样也就导致写的速度会有所下降，因此，如果想跨多个数据中心运行Zookeeper集群，用户应该事先考虑好这些问题的应对措施。

14.3 基于Zookeeper的服务发现

可以通过利用Zookeeper原生提供的临时**znode**特性来实现服务发现。服

务在注册时会在集群里的一个在其启动时给定的命名空间下创建一个临时**znode**，然后将它在网络上的位置信息（IP地址和端口）填充进去。**Zookeeper**层级式的命名空间为同类服务的集合提供了一个简单的实现机制，当基础设施里运行了多个同种服务的实例时，这个实现机制相当有用。

服务注册必须要嵌入到相关服务的源代码中，或者用户也可以编写一个简单的伙伴服务，它将使用**Zookeeper**协议并处理服务本身的注册工作。事实上，无论选择哪种方式都无可避免地需要编写一些额外的代码。客户端可以通过检索特定的**Zookeeper znode**的命名空间里的信息来发现已注册的服务数据。和之前提到的一样，只要被注册的服务创建的**TCP**会话仍然是活动状态，那么临时**znode**便不会被删除。而一旦服务从**Zookeeper**断开连接，**znode**就会被删除并且该服务马上会被注销。客户端可以在他们想要被告知相关情况的**znode**端[设置监听](#)。当**znode**发生变化时监听会被马上触发然后删除。

Zookeeper是完全使用Java编写的。该项目也提供了一个完备的Java客户端库来完成和**Zookeeper**集群的交互。虽然也有通过其他[编程语言实现](#)的客户端，但是并不是所有的客户端都会提供全面的功能特性的支持，而且他们的实现也各有差异，这有时候会使终端用户产生一些困扰。客户端必须处理服务发现和自动服务故障迁移两方面的负载平衡，以防出现客户端查找时一些发现的服务不再响应或它们的**Zookeeper**会话已经关闭的情况。如果用的是Java编程语言，那么这里有一个很棒的库，它在**Zookeeper**客户端库的基础上进行了一下包装，并且提供了很多额外的开箱即用的功能。它叫做[Curator](#)。同**Zookeeper**一样，它也是一个Apache基金会项目。关于如何使用Curator，可以查看Curator的[入门文档](#)。

Zookeeper的确能够提供一个健壮的、经过实战检验的服务发现方案。然而，在用户的基础设施里运行**Zookeeper**集群会引入一定程度的复杂性，它要求用户具备一些基本的**Zookeeper**运维经验并且会带来一些额外的维护成本。由于**Zookeeper**提供了一个强一致性的保证，当网络出现阻断时位于非仲裁方的服务即使仍然正常工作也将无法完成注册或寻找已经注册的服务。就服务更替非常频繁的写负载较重的环境而言，**Zookeeper**也许不是一个最佳选择。使用**Zookeeper**来完成服务发现的最棘手的问题之一便是它依赖已注册服务创建的**TCP**会话的持久性来保证服务发现的可用性。而仅仅只存在**TCP**会话也不能保证服务一定是健康

的。应用服务可以执行的任务非常多元化。只检查TCP会话是否存活的话很难验证这些任务的健康性。因此用户不能只靠TCP连接的活跃度来判断服务是否健康！很多用户低估了这一点并且因此引发了很多意想不到的事情。

如果把Zookeeper当做IT架构里的一个基础组件，它也许会让用户眼前一亮。虽然用户无法直接将其运用到Docker基础设施里，但是它可以通过其他构建在它之上的系统“悄然”贡献自己的一份力。经典案例莫过于[Apache Mesos](#)，用户可以使用它的一个插件然后借助Zookeeper来完成Docker宿主机上Mesos集群的容器之间的调度。如果想把Zookeeper作为一个独立的服务发现方案来使用，可能需要编写一个简单的伙伴客户端，它会和“Docker化”的应用服务一起运行并且处理服务自身的注册工作。然而，还有更简单的方法。用户可以使用一些基于Zookeeper之上实现的一些解决方案，如Smartstack，关于这块内容我们将在本章的稍后部分详细介绍。

接下来，我们将涉足的是分布式键值存储领域，相对于新人而言，它可以说是比Zookeeper更容易的、用作Docker基础设施里服务发现的方案。我们讨论的第一个主题即是这其中一款名为etcd的工具。

14.4 etcd

etcd是CoreOS团队使用Go语言编写的一款分布式键值存储软件。它和Zookeeper有许多的相似之处。我将先介绍它的一些基本特性然后再讲述如何将它用于服务发现。

同Zookeeper类似，etcd将数据存放在层级的命名空间里。它定义了目录和键的概念（这并不是etcd独创的）。任何目录都可以包含多个键，实际上它们是用来查找存储在etcd中的数据的唯一标识符。存储在etcd中的数据可以是临时性的也可以是持久化的。etcd和Zookeeper主要不同之处在于针对临时数据的实现方式。在Zookeeper里，临时数据的生命周期等同于客户端创建的TCP会话的寿命，而etcd采取的是和DNS的做法有些相似的一个方案。任何一个存放在etcd里的键都会设置一个TTL（Time to Live，存活时间）值。该TTL值定义了对应键在被设置值以后多长时间会过期，过期的同时该键值即被永久删除。客户端可

以在任意时刻刷新TTL从而延长存储数据的寿命。客户端还可以在任意键或者目录上设置监听，这样一来当这些数据发生变更时它们便能获取到相应的通知。**etcd**的监听机制是通过[HTTP长轮询](#)来实现的。

使用**etcd**的最大的好处之一在于它通过提供一个远程的JSON API抽象了底层的数据操作。这对于应用开发人员来说是一个天大的喜讯，他们不再需要使用特定的编程语言客户端来实现这些操作。所有需要与**etcd**交互的操作都可以通过一个简单的HTTP客户端来实现（另外，每个**etcd**发行版都默认自带一个名为**etcdctl**的命令行客户端工具）。正如[官方文档](#)中的很多例子中展示的，用户甚至可以简单地使用**curl**命令来和**etcd**集群进行交互。**etcd**使用[Raft](#)一致性算法，通过在集群之间同步日志来管理数据。和Zookeeper使用的ZAB类似，Raft定义了领导者和追随者节点。所有的写请求都必须经由通过领导者使之生效，随后它会将操作以日志形式同步/重演到其余的追随者节点。

为了使**etcd**能够正常工作，集群里必须部署 $2n+1$ 个节点，即3、5、7等。可以通过<https://coreos.com/os/docs/latest/cluster-architectures.html>来查看推荐的生产环境集群设置方案。同Zookeeper一样，**etcd**也提供了强大的数据一致性保证和分区容忍。此外，**etcd**还建立了一套强大的[安全模型](#)，它使得客户端与集群之间以及集群中各节点之间的通信都可以采用SSL/TLS作为认证方式来完成客户端的授权。虽然说在**etcd**集群的管理方面可能会遇到一些小小的挑战。但值得庆幸的是，**etcd**官方提供了很棒的[管理](#)和[集群部署](#)指导手册，在将**etcd**集群部署到基础设施之前，必须得去读一读这些手册。当然，**etcd**不只限于这里简要介绍的内容，因此我建议如果感兴趣的读者不妨去看看它的官方文档。接下来，我们将讨论该如何将**etcd**用于服务发现。

基于**etcd**的服务发现

可以通过利用**etcd**的TTL特性来实现服务发现的功能。注册服务会在**etcd**集群里创建一个新的键值对，然后把连接的详细信息填到里面。这里，可以创建一个单独的键也可以插入到一个现有的键的目录里。**etcd**的目录提供了一个很好的分组方式，它可以将运行在基础设施里的相同服务的多个实例归类到一起（目录本身也是以一个键的形式存在）。该服务随后可以在一个给定的键上设置一个TTL，这样一来用户无需对它做任何进一步的操作，存放在其中的数据便会在达到TTL的值后自动过期。借助于TTL，**etcd**实现了一个简单的自动注销服务的机

制。用户还可以通过更新TTL的方式来延迟数据的过期时间，这也是一些长期运行的服务必须做的，它们可以借助这一手段来避免不断的注册/注销。客户端则可以通过查询etcd集群里的特定键或者目录来找出已注册服务的连接信息。就像之前提到的那样，客户端可以在任意一个键上设置监听，然后可以在该键存储的数据发生变化时收到通知。

多亏了有etcd提供的远程JSON API，服务的注册甚至还可以嵌入到服务的源代码里，或者也可以使用一个伙伴进程，通过实现一个简单的HTTP客户端——`etcdctl`或者`curl`就能办到——来和etcd集群交互。由于使用简单的命令行工具便能完成服务的注册，这使得集成第三方服务也变得相当简单：可以在服务启动时在etcd里添加一个新条目，然后在服务关闭时删掉该键。借助于SystemD，可以很容易地实现伙伴进程同主服务进程一起启动、停止。

在使用Docker时用户也可以轻松地采取相同的策略来实现服务的注册。伙伴进程会去检索服务容器的信息，然后将解析后的IP地址和端口数据填充到一个特定的etcd键里。这个键随后可以被其他Docker容器通过查询etcd集群的方式来读取。

如果采用伙伴进程来实现服务注册，最终可能在维护方面会比较伤脑筋，因为用户需要不断地更新TTL值并监控已注册服务的健康状况。用户一般是通过一个简单的shell脚本来实现这一点，它会运行一个无限循环，并且以一定的时间间隔不断检查正在运行的服务的健康性，然后据此更新特定的etcd键对应的值。可以通过一个简单的[实例](#)来了解这个方案。

至于客户端，业界已经有大量现成的、不同的编程语言实现的工具和[库可供选择](#)，这里面的大多数工具的社区仍然相当活跃。再说，同其他工具一样，etcd原生的Go语言库应该会提供所有功能特性的支持。遗憾的是，它仍然无法提供一些服务负载均衡或者故障转移方面的功能支持，因此用户需要自己去解决这些问题。

etcd提供了一个非常不错的针对服务发现的解决方案。因此，尽管它仍然还在不断的迭代完善，很多企业已经开始将它应用到他们的生产环境中。具备编程语言无关性的远程API接口对于应用开发者而言有很大的推动作用，因为它给了他们更多的选择空间。然而，同Zookeeper一样，采用etcd的话会给用户的基础设施引入额外的管理复杂度。用户需要了解如何操作etcd集群，而这一点并不是那么容易就能办到。通常来

说，缺乏对etcd内部原理的理解，往往可能导致一些意想不到的事情发生，有些情况下甚至可能会丢失数据。

因此，对于一个初学者而言，根据用户存储在集群里的数据容量来完成etcd的扩容工作可能会是一个不小的挑战。服务目录里的记录必须有它们各自的TTL值，然后用户需要通过已注册的服务不断地刷新该值，这需要开发者投入一些额外的精力。如果所处环境里运行的服务本身生命周期很短，那么频繁更新TTL值会产生相当大的网络流量。我们这里所提到的etcd实际上是许多其他开源项目实现的基石，像之前提到过的SkyDNS或者是Kubernetes，并且它已经默认被内置到了CoreOS Linux发行版里。该项目很有可能会得到进一步的发展和显著的提高。

etcd已然成为Docker基础设施里一个非常流行的、用来实现服务发现解决方案的基本构建组件。一些新的、全套的解决方案都受到了它的启发。值得一提的是，这里面有一个项目是Jason Wilder本人创建的。它把etcd和另外一款非常流行的名为HAProxy的开源软件结合了起来。它采取伙伴进程的方式来实现服务发现并且利用Docker API发送相应的事件，随后用它来生成HAProxy的配置，借此完成Docker容器里运行的服务与其他服务之间请求的路由和负载均衡工作。关于这部分内容，读者可以在<http://jasonwilder.com/blog/2014/07/15/docker-service-discovery/>了解更多详细内容。再强调一次，业界可能已经有很多方案是基于etcd实现的服务发现，然而Jason的这个项目把服务发现本身的一些基础概念讲解得非常透彻，并且定义了一个已经在Docker社区里被反复验证的服务发现模型。图14-2对该项目进行了简单地讲解说明。

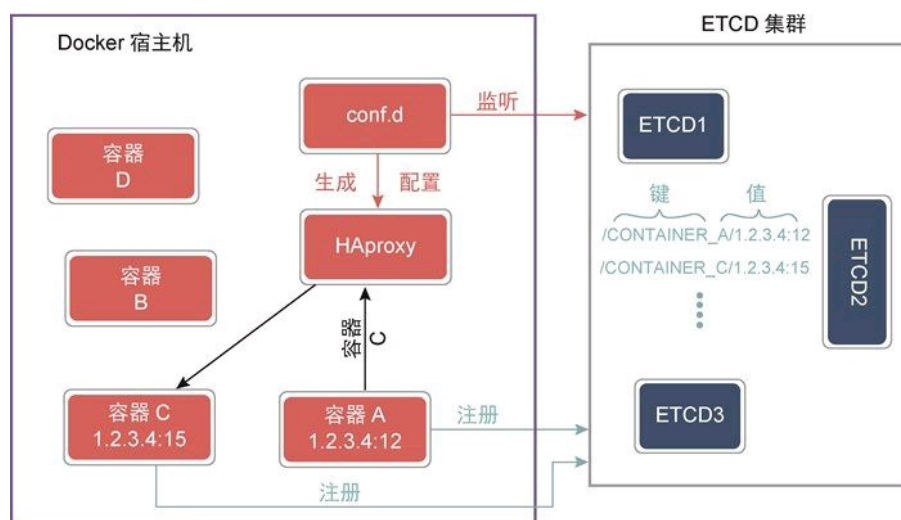


图14-2

在以上设定中，HAproxy直接运行在宿主机上（即不是运行在Docker容器里）并且为所有运行在Docker容器里的服务提供了一个单一的入口。运行在Docker容器里的服务会在服务启动时通过在etcd里创建一个特定的键条目来完成注册。我们可以利用一个特殊的服务进程（如conf.d）来监控etcd集群里的键命名空间，一旦发生变化的话它会马上为之生成新的HAproxy配置并随后重新加载HAproxy服务使之生效。针对该服务的请求会自动被路由和负载均衡到运行在Docker容器里的其他服务。这有点像Smartstack所推崇的模式，它是另外一款服务发现的解决方案，我们将在本章的后面部分详细介绍。

由于篇幅有限，关于Docker生态圈里其他那些围绕etcd建立的服务发现方案便留待读者朋友探索。接下来，我们将介绍一款比etcd更加年轻也可以说更加强大的兄弟软件：consul。

14.5 consul

consul是一款由HashiCorp公司编写的多功能分布式系统工具。同之前介绍过的etcd一样，consul也是使用Go语言实现。consul很好地将它所有的特性集成为一个可定制化软件，并易于使用和运维。在这里，我们不会花太多篇幅去介绍什么是consul，关于这一点读者可以在它的[官方网站](#)上找到一个非常全面的文档，里面包含了大量的实际案例。取而代之的是，我们将会去总结它的主要特性并且探讨在Docker基础设施里如何借助它来实现服务发现。最后，在本章的末尾我们将会介绍一个实际案例，它利用consul提供的功能特性，将consul作为一个插件式的后端服务，为运行在Docker容器里的应用提供了即插即用的服务发现功能。

我们选择用多功能一词来描述consul的目的正是在于consul的确可以无需花费其他任何额外的精力，作为一款单独运行的工具提供下述任意一项功能：

- 分布式键值存储；
- 分布式监控工具；
- DNS服务器。

上述功能及其易用性使得consul成为DevOps社区里一款非常强大和流行

的工具。让我们快速过目一下，看看consul的背后究竟隐藏了些什么奥秘使得它可以用在如此多的场合。

consul，同etcd类似，也是基于Raft一致性算法实现的，这也就是说，它所在的集群里的节点部署数量同样应该满足 $2n+1$ （ n 表示一个正整数）以保证其正常工作。和etcd一样，consul也提供了一个远程的JSON API，这使得各种不同的编程语言实现的客户端访问该服务更加简单。通过提供远程API的支持，consul允许用户自行在其之上构建新服务或直接使用它原生提供的开箱即用的功能。

就部署而言，consul定义了一个代理（agent）的概念。该代理能够在以下两种模式运行：

- 服务端——提供分布式键值存储和DNS服务器；
- 客户端——提供服务的注册、运行健康监测以及转发请求给服务器。

服务端和客户端代理共同组成一个完整的集群。consul通过利用HashiCorp编写的另外一款名为serf的工具来实现集群成员身份和节点发现。serf是基于SWIM一致性协议实现的，并且在一些性能方面做了优化。您可以通过[consul官网](#)来了解consul中的gossip详细的内部实现原理。利用gossip协议并通过将它和本地服务的健康监测结合到一起，这使得consul可以实现一个简单但是异常强大的分布式故障检测机制。这对于开发者和运维人员而言实在是一个巨大的福音。开发人员可以在他们的应用程序里公开健康监测的端点然后轻松地将应用服务添加到consul的分布式服务集合里。运维人员也可以编写简单的工具，使用consul的API来监控服务的健康性，或者他们只是使用consul原生提供的[Web UI](#)去操作。

这里讨论到的只是consul一些基本的内容，实际上它所提供的还远不止这些，因此强烈建议读者去细读一下consul强大的官方文档。如果想知道consul和市面上其他工具的差异，不要犹豫，赶紧去看一下专门讨论这一话题的[官方文档](#)吧。接下来，让我们一起来看看我们该如何将consul用于基础设施里的服务发现。

14.5.1 基于consul的服务发现

迄今为止，在我们介绍过的工具中，作为一款可定制的服务发现解决方案，**consul**无疑是最容易上手的一个。用户可以通过以下几种方式将自己的服务注册到**consul**的服务目录里：

- 利用**consul**的远程API将服务注册嵌入到用户的应用代码里；
- 使用一个简单的伙伴脚本/客户端工具，在应用服务启动时通过远程API来完成注册；
- 创建一个简单的声明服务的**配置文件**，**consul**代理可以在服务启动或重新加载服务后读取该配置。

已注册的服务可以通过**consul**的远程API直接去查找，当然用户也可以使用**consul**提供的开箱即用的DNS服务来检索它们的信息。这一点尤其方便，因为用户不必再受限于一个特定的服务查找方案，而且甚至可以不费任何力气地同时使用这两套方案。此外，**consul**还允许用户为自己的应用服务设计一些自定义的健康检测机制。如此一来，用户不必再像之前的Zookeeper那样和TCP会话周期绑定到一起，也不必像**etcd**那样和TTL值挂钩。**consul**代理程序会持续不断地在本地监控已注册服务的健康性并且一旦健康检测失败它会立马自动将其从服务目录中抹除。

Consul提供了一个非常完备的服务发现解决方案，并且令人意外的是它的成本其实非常低。在**consul**中，应用服务可以通过远程API或DNS来定位和检索。为了完成服务的注册，需要避免使用远程API而可以采用更简单的基于JSON的配置文件来实现这一点。这使得**consul**能够很方便地和传统配置管理工具集成在一起。使用**consul**会给用户的基础设施引入一些额外的复杂度，但是作为回报，用户也从中获得了大量的收益。与**etcd**相比，**consul**集群无疑是更易于维护 and 管理的。**Consul**在多数据中心方面也具备很好的扩展能力，事实上，**consul**提供了一些额外的工具专门负责多数据中心的扩容工作。**Consul**可以作为一个单独的工具使用，也可以作为构建一个复杂的分布式系统的基本组件。如今，业界围绕它已然形成了一个完整的工具生态圈。在下一节里，我们将会介绍一款名为**registrator**的工具，它使用**consul**作为它的一个可插拔的后端服务，它为运行在Docker容器里的应用提供了一个非常易于上手的、自动服务注册的解决方案。

14.5.2 registrator

从整体上来说，**registrator**会去监听Docker的Unix套接字来获取

Docker容器启动和消亡时的事件，并且它会通过在事先配置好的一个可插拔的后端服务中创建新记录的形式自动完成容器的服务注册。这就意味着它必须以一个Docker容器的身份来运行。读者可以在[Docker Hub](#)上找到**registrator**的Docker镜像。**registrator**提供了相当多的配置参数选择，因此，尽情去[GitHub项目页面](#)的文档库里去查找关于它们的详细解释吧。

下面，让我们一起来看一个简短的实际案例。我们将会使用**registrator**把**redis**内存数据库打包到Docker容器里运行，用户可以很方便地通过**consul**发现它在网络上所提供的服务。当然，用户也可以使用类似的方法在Docker基础设施里运行任意的应用服务。

回到这个例子，首先我们需要启动一个**consul**容器。这里，需要用到**registrator**之父[Jeff Lindsay](#)创建的镜像来实例化具体的容器：

```
# docker run -d -p 8400:8400 -p 8500:8500 -p 8600:53/udp -h
node1 progrium/consul -server -bootstrap
37c136e493a60a2f5cef4220f0b38fa9ace76e2c332dbe49b1b9bb596e3ead39
#
```

现在，后端的发现服务已经开始运行，接下来我们将会启动一个**registrator**容器，并且同时传给它一个**consul**的连接URL作为参数：

```
# docker run -d -v /var/run/docker.sock:/tmp/docker.sock -h
$HOSTNAME gliderlabs/registrator consul://$CONSUL_IP:8500
e2452c138dfa9414e907a9aef0eb8a473e8f6e28d303e8a374245ea6cd0e9cdd
```

如下所示，我们可以看到容器均已成功启动，并且我们假定所有容器都是注册的**redis**服务：

```
docker ps
CONTAINER ID          IMAGE                                COM-
MAND                 CREATED                STATUS
PORTS
NAMES
e2452c138dfa          gliderlabs/registrator:latest      "/bin/regis-
trator co    3 seconds ago         Up 2 sec-
onds
distracted_sammet
```

```
37c136e493a6      progrium/consul:latest      "/bin/start
-server      2 minutes ago      Up 2 minutes      53/tcp,
0.0.0.0:8400->8400/tcp, 8300-8302/tcp, 8301-8302/udp,
0.0.0.0:8500->8500/tcp, 0.0.0.0:8600->53/udp      furious_kirch
```

考虑到整个例子的完整性，我们不妨介绍一下最初的情况，下列命令展示了我们正在运行的只有一个节点的consul集群并且在该时刻没有任何已注册的服务运行：

```
# curl $CONSUL_IP:8500/v1/catalog/nodes
[{"Node": "consul1", "Address": "172.17.0.2"}]
# curl $CONSUL_IP:8500:8500/v1/catalog/services
{"consul": []}
```

现在，让我们先启动一个redis容器，然后公开它所有需要对外提供服务的端口：

```
# docker run -d -P redis
55136c98150ac7c44179da035be1705a8c295cd82cd452fb30267d2f1e0830d6
```

如果一切顺利，我们应该可以在consul的服务目录里找到该redis服务的信息：

```
# curl -s localhost:8500/v1/catalog/service/redis |python -
mjson.tool
[
  {
    "Address": "172.17.0.6",
    "Node": "node1",
    "ServiceAddress": "",
    "ServiceID": "docker-hacks:hungry_archimedes:6379",
    "ServiceName": "redis",
    "ServicePort": 32769,
    "ServiceTags": null
  }
]
```

从上面的输出可以看到registrator定义服务所采用的格式。关于这一点可以转到[项目文档](#)了解更多的细节。正如我们在前面章节所了解到的，consul提供了一个原生的开箱即用的DNS服务的支持，因此所有已注册的服务可以很轻松地通过DNS来查找和定位。要验证这一点也非常

简单。首先，我们需要找出consul提供的DNS服务器将哪些端口映射到了宿主机上：

```
# docker port 37c136e493a6
53/udp -> 0.0.0.0:8600
8400/tcp -> 0.0.0.0:8400
8500/tcp -> 0.0.0.0:8500
```

太棒了，我们可以看到容器的DNS服务被映射到了宿主机的所有网络接口上，并且监听了8600端口。现在，我们可以使用Linux上著名的**dig**工具来完成一些DNS的查询操作。从consul的官方文档中我们可以了解到，consul里已注册服务对应的默认的DNS记录会以NAME.service.consul的格式命名。因此，在这个例子中，当注册一个新服务时**registrator**使用的Docker镜像名便会是**redis.service.consul**（当然，必要的话也可以修改这个设置）。

那么，现在让我们来试着运行一下DNS的查询吧：

```
# dig @172.17.42.1 -p 8600 redis.service.consul +short
172.17.0.6
```

如今我们已经获得了**redis**服务器的IP地址，但是同该服务通信所需的信息还远不止这些。我们还需要找出该服务器监听的TCP端口。幸运的是，这一点很容易办到。我们需要做的只是通过查询consul的DNS来寻找对应的使用相同的DNS名称的**SRV**记录。如果一切顺利，我们应该可以看到返回的端口号是32769，当然我们也可以通过它提供的远程API以检索consul服务目录的方式来获取这个信息：

```
# dig @172.17.42.1 -p 8600 -t SRV redis.service.consul +short
1 1 32769 node1.node.dc1.consul.
```

真的是太棒了！借助consul，我们成功地为我们的Docker容器实施了一整套完备的服务发现方案，而且所有我们需要做的配置只是运行两个简单的命令而已！我们甚至无需编写任何代码。

如果我们现在停止**redis**容器，consul会将它标记为已停止的状态，如此一来，它将不会再响应我们的任何请求。这一点同样也非常容易验证：

```
# docker stop 55136c98150a
55136c98150a
# dig @172.17.42.1 -p 8600 -t SRV redis.service.consul +short
# dig @172.17.42.1 -p 8600 -t SRV redis.service.consul

; <<>> DiG 9.9.5-3ubuntu0.1-Ubuntu <<>> @172.17.42.1 -p 8600 -t
SRV redis.service.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 56543
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDI-
TIONAL: 0

;; QUESTION SECTION:
;redis.service.consul.      IN      SRV

;; Query time: 3 msec
;; SERVER: 172.17.42.1#8600(172.17.42.1)
;; WHEN: Tue May 05 17:59:35 EDT 2015
;; MSG SIZE rcvd: 38
```

如果正在寻找一个简单而容易上手的服务发现的解决方案，**registrator**无疑是一个非常省力的选择，尽管它仍然需要用户运行一些像**consul**或者**etcd**这样的存储后端。然而，由于其本身具备简单部署的优势以及它提供的对**Docker**的原生集成支持，选用它无疑是利大于弊的。

现在，我们将结束这一节，接着介绍一个新的不依赖任何一致性算法来保证数据的强一致性的解决方案，然而它仍然提供了一些有趣的特性，因此不妨将它当作实现用户的基础设施里的服务发现的一个备选方案。

14.6 Eureka

在过去的几年里，**Netflix**的**工程师**团队开发了大量的开源工具来帮助他们管理微服务云基础架构，以方便那些以前必须创建他们自己的**开源软件中心**心才能消费他们的用户快速上手。大部分的工具都是用**Java**编程语言编写的，如果用户没有复用其**OSS**工具箱里的一些其他工具往往会很难将其集成到私有的基础设施里。在这一节里，我们将会一起来看

看Eureka究竟提供了怎样的一个有意思的服务发现的选择。

Eureka是一个基于REST的服务，它在设计之初最主要的目的是用来提供“中间层”的负载均衡、服务发现和服务的故障转移。官方推荐的用例场景是，如果用户在AWS云中运行自己的基础设施，AWS云是Eureka经过实战检验的地方，而用户的基础设施里又存在大量的内部服务，同时又不希望注册到AWS ELB中或者对外公开这些服务的话，这时候可以考虑采用Eureka实现服务发现。可以说，促成Eureka的最主要的动力便是在于ELB不支持内部服务的负载均衡。理想情况下，由于Eureka本身不提供会话保持的功能，因此通过Eureka发现的服务本身应该都是无状态的。从架构上来说Eureka主要有两个组件：

- 服务端——提供服务的注册；
- 客户端——处理服务的注册，并且提供基本的、轮换式的负载均衡和故障转移功能。

官方推荐的部署方案是每个AWS地理区域部署一个Eureka的服务集群，或者至少每一个AWS可用区域部署一台服务器。实际上，Eureka服务端根本不知道其他AWS区域有哪些服务器。它保存信息最主要的目的是为了保证一个AWS区域内的负载均衡。Eureka集群里的服务器会以异步的形式同步他们彼此间的服务注册信息。这一同步操作可能需要花一些时间才能完全生效。由于服务端可能有缓存，因此该操作有时候甚至要耗费数分钟的时间才能完成。与Zookeeper、etcd或consul相比，Eureka更偏重于服务的可用性而不是数据的强一致性，因此客户端往往可能需要处理脏数据的读取。没办法，Eureka本身便是这样设计的。它关注的是经常发生故障的云环境下的弹性伸缩问题。采用这种方案就意味着Eureka甚至可以在集群因为某种故障出现网络分区时仍能工作，不过这样会牺牲数据的一致性。

Eureka客户端会将应用服务的信息注册到服务端，并且随后每30秒更新一次他们的“租约”信息。如果客户端没有在90秒内更新它的租约，那么它会自动从服务端的注册中心里删除，然后必须重新注册。这跟etcd里面通过TTL实现的机制有些类似，但是如果使用Eureka，用户无法设置注册中心里的记录的生命周期。Eureka客户端对于服务端故障具备一定的弹性耐受能力。它们会将本地的注册信息缓存起来，因此即使注册服务器发生故障，它们依旧可以正常工作，这显然要求客户端这一方可以处理一些服务的故障转移工作。一旦网络分区的故障排除，客户端本地的状态会被合并到服务端。而为了易于排障，Netflix OSS库里提供了另

外一款名为[Ribbon](https://github.com/Netflix/ribbon)的工具专门用来解决这个问题。

Eureka旧版本的客户端主要是基于pull模式的，然而最新发布的版本已经加入了大量对于服务端和客户端两方面的改进。它将集群的读写分离开来，从而改善了性能并提高了可扩展性。Eureka客户端可以订阅一组特定的服务，随后，就像之前介绍过的工具一样，他们便可以在服务端发生任何更改的时候收到通知。Eureka官方还提供了一个简单的仪表盘，使其更容易部署到其他的云服务上。您可以转到如下[链接](#)来了解新版本设计理念方面的更多内容。

基于Eureka的服务发现

就像之前所提到的那样，Eureka的设计目标是客户端的中间层负载均衡。要实现服务之间的负载均衡，必须首先看看它们是否在服务端的注册中心里。由于Eureka是使用Java编程语言实现的，因此用户可以相当容易地将Java客户端集成到自己的应用代码里。原生的客户端提供了非常全面的功能特性，包括一个简单的基于轮换式的负载均衡的支持。每个应用服务可以在它启动的时候到Eureka服务端注册自己的信息，然后每隔30秒发送一个心跳包到服务端。如果超过90秒该服务仍然没有动作，Eureka会自动将其注销。

Eureka还对外开放了一个REST API，如此一来用户便可以轻松实现自己的客户端。虽然开源界目前已经有几个不同编程语言实现的客户端库，但是没有哪个客户端能够真正在功能覆盖面及质量上胜过原生客户端的实现。另外一个选择是用户可以实现一个小型的Java伙伴程序，和自己的服务一起运行，然后由它来调用原生的客户端库帮忙处理服务的注册和心跳汇报工作。这将带来额外的工作量以及不必要的维护复杂度，当然用户将因此获得原生客户端库的全部功能支持。

使用Eureka来实现服务发现的最大优点在于它对故障方面具备一定的容忍能力，这使得它成为云环境下的一个非常不错的选择，当然，前提是用户可以在客户端这一边处理服务的故障转移以及脏数据的读取。事实上，云环境的部署需求正是它设计最主要的动力所在。然而，遗憾的是，用户无法控制服务注册中心里记录的生命周期，而且必须不断地发送心跳包，这样做的话可能会为自己的基础设施带来不小的流量压力，并且会给注册服务器带来一些额外的负载。客户端查询的往往也是全量

的服务列表数据，在查找服务时也没有过滤或者搜索的细粒度之分等。关于这一点可能会在2.0版本得到改善，而这一版本也会引入一个服务订阅的概念，即用户可以收到有关服务的通知信息。

Eureka在最开始设计的时候便考虑到了自动扩展，因此它的可扩展性相当不错。此外，Eureka的最新版本对它的扩展能力做了进一步的改善。它的致命要害在于，如果想充分利用Eureka，就必须要用到Netflix的一些其他OSS组件，如之前提到的Ribbon库或Archaius配置服务，该服务又依赖于Zookeeper。这一点也许读者早就意识到了，它可能会为基础设施引入大量不必要的复杂度。

接下来，我们将把视角从Netflix的OSS深渊里挪开，转而讨论一个已经非常流行的不同的变种方案，借助它用户可以非常便捷地实现服务发现，而且它还有一个非常有趣的名字。那么，让我们一起来见识一下Smartstack吧！

14.7 Smartstack

Smartstack是一个由AirBnb的工程师团队创建的服务发现解决方案。Smartstack在整个服务发现的生态圈里的地位非常特殊，原因在于它的设计理念启发了其他的解决方案。正如它的名字所暗示的，smartstack真的是一组智能服务组成的技术栈，其中包括Nerve和Synapse两个部分。

Nerve和Synapse都是使用Ruby编程语言编写的，并且以Ruby gem的形式发布。他们可以和HAproxy以及之前介绍过的Zookeeper交互。<http://www.haproxy.org>上有一篇很棒的入门性质的博客文章，读者有兴趣的话不妨去读一读，在这里可以了解到更多关于Smartstack背后的一些创造动机。在本节里，我们将介绍它的一些主要特性，并在最后做一个简短的总结。请记住，在打算将Smartstack部署到自己的基础设施之前，千万不要犹豫，多看一些它的在线文档会有很大帮助的。

14.7.1 基于Smartstack的服务发现

Smartstack在服务发现的注册和发现方面是伙伴进程模型的拥护

者：**synapse**和**nerve**均是以独立进程和应用服务一起运行的，并且代表应用服务自动处理服务的注册及查找工作。在生产环境里，一台宿主机上运行一个Synapse实例应该就够了。**Smartstack**会利用**Zookeeper**作为自己服务目录的后端并且采用**HAproxy**作为已发现的服务的唯一入口和负载均衡器。**Smartstack**可以非常轻松地集成到用户的**Docker**基础设施里。图14-3所示就是采用**Smartstack**实现服务发现的一个简单架构。

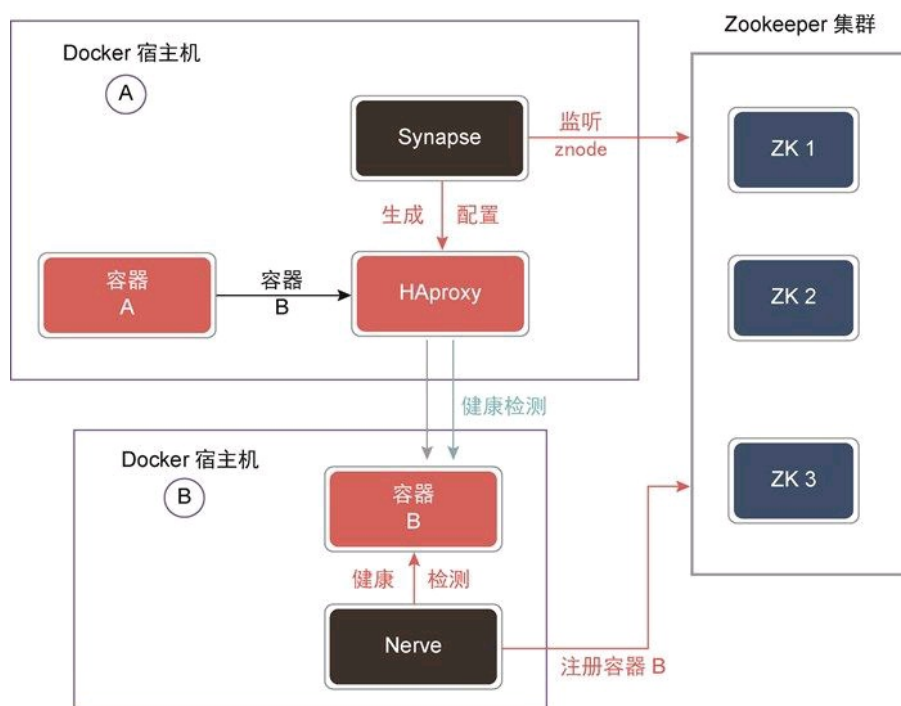


图14-3

应用开发人员无需编写任何服务发现的代码并且他们还可以得到免费的开箱即用的负载均衡和服务故障自动转移的功能支持。接下来，让我们进一步看看**Smartstack**的这两个核心服务，从而更好地理解**Smartstack**究竟是怎样完成服务发现工作的。

14.7.2 Nerve

Nerve是一款简单的用来监控机器和服务的健康状况的工具。它将服务的健康信息保存在一些分布式存储里。后端方面目前只完全支持**Zookeeper**，但是官方也正在努力做针对**etcd**的完全适配工作。**Nerve**负责服务发现里的服务注册部分，它会根据服务的健康状态添加和删除**Zookeeper**集群里的**znode**。如果使用**etcd**作为后端存储，**Nerve**将会

在etcd集群里添加一个键值记录然后设置它的TTL为30秒。之后，它会根据服务的健康状况不断地更新该记录。

Nerve给服务部署方面指明了一条道路，即要求应用开发人员提供一些合适的机制来监控服务的健康状况。这一点很重要，而这不仅是为了提高服务发现实现的可靠性。Nerve利用应用服务提供的健康检测方案来驱动服务注册流程。最后，Nerve还可以从服务发现解决方案的整体中分离出来，作为单独的监控服务的看门狗来使用。

如果想了解Nerve更多的内容，不妨去GitHub上读一读它的[官方文档](#)。

14.7.3 Synapse

Synapse是一款简单的服务发现的实现方案，它定义了服务watcher的概念，让用户可以从指定的后端中监听相应的事件。Synapse会根据收到的事件生成相应的HAproxy配置文件。Synapse中提供了一些可用的服务watcher：

- **stub**——没有监听的概念，用户只能手动指定服务的列表；
- **zookeeper**——zookeeper会在集群里的特定znode节点上注册监听；
- **docker**——监听Docker API的事件；
- **EC2**——根据AWS EC2的标签来监听服务器。

每当监听的服务不可用时，Synapse会重写HAproxy的配置，随后重新加载HAproxy服务使之生效。所有客户端的请求都是通过HAproxy代理的，它负责将请求路由到真正的特定应用服务。这对于应用开发人员和运维人员来说都是一个共赢的局面：

- 开发人员不必编写任何的服务发现代码；
- 运维人员也可以通过一些经过充分验证的解决方案来实现服务的负载均衡以及故障迁移。

再强调一下，这些内容均可以在GitHub上的[官方文档](#)里找到。

Smartstack是一个具备技术无关性的绝佳方案，借助它，用户可以在基础设施里实现服务发现。它不需要用户编写任何额外的应用代码并且可以轻松地部署到裸机、虚拟机或Docker容器里。Smartstack本身相当简单，但是整个套件至少需要维护4个不同的部分：Zookeeper、

HAproxy、Synapse和Nerve。例如，如果没有在基础设施中事先运行Zookeeper，用户可能觉得运行全套的Smartstack方案会很难。此外，虽然在用户的服务之前运行HAproxy可以为用户提供一个不错的服务层抽象以及负载均衡和服务的故障迁移功能，然而用户需要在每台宿主机上至少管理一个HAproxy实例，这会引入一定的复杂度，并且往往需要一定的维护成本。

14.8 nslookupd

在本章的最后，将简单介绍一款由bitly的工程师团队开发的名为nslookupd的工具。nslookupd并不是一个完整的服务发现解决方案，它只是提供了一种发现nsqd实例的创新方式，或是在应用运行时跑在基础设施里的一个分布式消息队列。

实际上，nsqd守护进程在向那些nslookupd实例声明自己启动的时候就已经完成了服务注册，随后nsqd会定期发送带有它们状态信息的心跳包给每个nslookupd实例。

nslookupd实例担任的角色是直接为客户端提供查询的服务注册中心。它们提供的只是一个网络上周期性同步的nsqd实例的数据库。客户端通常需要检索每个可用的实例信息，然后合并这些结果。

如果要找的是一个可以在大规模基础设施这样的拓扑架构里运行的分布式消息队列解决方案，不妨去看一看nsqd和nslookupd项目的官方文档，了解更多详细内容。

14.9 小结

在本章里，我们介绍了一系列的服务发现解决方案。服务发现没有捷径可寻，只能根据任务的类型和它必须满足的需求来选择一款合适的工具。和传统的推荐一个特定解决方案的形式不同，我们给出一个包含多种解决方案的概述表（见表14-1），对本章的内容做出总结，希望这能够帮助读者做出正确的选择，选出最适用于自己的基础设施的服务发现

工具。

表14-1

名称	注册机制	数据一致性	语言
SkyDNS	客户端	强一致	Go
weave-dns	自动注册	强一致	Go
ZooKeeper	客户端	强一致	Java
etcd	伙伴程序 + 客户端	强一致	Go
consul	客户端 + 配置 + 自动注册	强一致	Go
eureka	客户端	终端一致	Java
nslookupd	客户端	终端一致	Go

在第15章里，我们将介绍Docker的日志采集和监控。

[1] 即像systemd这样的服务管理程序，你可以将自己的应用和伙伴进程关联起来，设定伙伴进程在应用启动之后方才启动。——译者注

[2] IANA机构设定了一个Linux下常见服务端口注册列表（<https://zh.wikipedia.org/wiki/TCP/UDP%E7%AB%AF%E5%8F%A3%E5>），它指定了这些著名服务的默认注册端口号。——译者注

第15章 日志和监控

在虚拟化时代来临时，很多企业迫切需要一种新手段来监控它们的新的虚拟环境。业内的监控工具也因此需要作出更新，以支持监控宿主机和虚拟机两种不同的环境。如今，相同的问题在部署容器化的基础设施时再次发生。Docker环境下的日志收集和监控乍听起来可能会让人头疼不已，但是如果可以打破现有环境的束缚，并且使用新兴的工具的话，便可以让读者从这一新的虚拟环境中真正受益。

运行在容器里的系统和应用与如今部署在虚拟服务器上的容器也不尽相同。对许多人来说，监控和记录虚拟化系统上的另外的抽象层可能会是一件非常难以想象的事情。无论环境有多复杂、多抽象，我们始终需要建立一套监控和日志系统来了解应用的健康状态及性能，并满足各种审计的要求。事实上，对于企业来说似乎其最大的问题在于，针对这一抽象层现有的工具缺乏足够的可视化能力。接下来，我们重点关注日志和监控这两部分，只要做到这两点用户就可以让Docker容器在自己的环境里成功地运行起来。

15.1 日志

需要查看Docker容器里的日志时，有几种不同的方案可供选择。这些方案取决于你的容器所处的状态，是正在运行还是已销毁/已删除。当考虑的是正在运行中的容器的日志方案时，有如下方案可供选择：

- Docker原生提供的日志模块支持；
- 通过连接正在运行的容器来提取日志；
- 将日志导出到Docker宿主机；
- 将日志发送到集中式的日志平台；
- 将日志装载到其他Docker容器里。

对于一个已经销毁的/已删除的容器来说，用户的选择非常有限。由于容器默认是临时性的，因此用户可能会在它们终止服务时丢失相应的数据。日志数据的丢失可能会不利于对已发生故障的应用的故障排查及定位。为了能够在容器终止时将其日志保存下来，用户可能需要将这些日志从容器导出到宿主机或者线上系统里。

15.1.1 Docker原生的日志支持

Docker会自动捕获运行在Docker容器里的进程的标准输出和标准出错信息，并且将它们重定向到容器的根文件系统上的一个指定的日志路径里，宿主机上拥有**root**权限的用户可以直接访问这一容器日志。可以通过运行以下命令来获取宿主机上容器默认的日志路径：

```
docker inspect -f '{{.LogPath}}' <container_id>
```

因此，运行在Docker容器里的进程若以前台模式运行会让用户能够很方便地利用Docker原生提供的日志功能来记录其输出信息。

Docker的1.6版本还引入了一些开箱即用的日志引擎。默认情况下，Docker采用的是**json-file**，它会将应用的日志以**json**格式存储到前面提到的容器根文件系统的日志路径下。这一点非常人性化，因为许多的集中式日志方案都需要一个**json**格式的数据流来方便建立搜索索引。

可以通过运行Docker客户端内置的**logs**命令来查看容器日志的内容。请注意，运行**logs**命令将会输出从容器启动开始所有的日志，所以最好将它重定向到其他的类似于**more**或**less**这样的查看工具：

```
docker logs redis
```

如上所见，Docker命令行客户端会自动将**json**编码的文件解码并且以文本的格式输出到界面。

如果用户正在检查一个正在运行的容器发生的问题，那么一般便是运行上面这条命令。该命令会告诉Docker从宿主机记录日志的地方找出一个名为**redis**的容器的日志。在默认情况下，如果没有变动，容器的日志会存放到保存容器本身的系统文件的容器根文件系统里。切记在生产环境下运行容器时，如果应用日志的输出量很大，一定要对日志配置轮换

或将日志文件存放在一个可扩展的磁盘存储里。如果运行下面这条命令，容器的日志会被源源不断地写到文件系统里，因此用户将可以看到一个持续不断的日志输出：

```
docker logs redis --follow (or -f for short)
```

Docker的**logs**命令还支持一个额外的参数以显示文件的时间戳并限制**logs**命令可以查看的最多行数。当需要调试时间敏感的问题时可以使用**-timestamps**（或者**-t**）参数来显示日志里每行的具体时间戳。也可以使用**--tail**加上一个数值，如**--tail 10**，来显示容器标准输出和标准错误输出的最后10行内容。

Docker原生的**logs**命令目前所能提供的就是这些。例如，用户正在运行**redis**数据库服务器，它会将日志输出到该容器的**/logs/redis.log**里。然而，由于Docker只会捕获**stdout**和**stderr**的输出，因此执行**docker logs redis**将无法显示该文件的输出内容。我们将在本节的后面部分解释该如何处理这些日志。

Docker的第二个日志驱动的选择便是**syslog**。一旦使用它，Docker会将所有的应用日志发送到运行在宿主机上的**syslog**。当然，用户必须在启动容器时明确指定该参数。在**redis**例子中，想要运行的命令应该是这样的：

```
docker run -d --log-driver=syslog redis
```

如果现在去查看宿主机上**syslog**的内容，用户会看到我们刚启动的**redis**容器生成的日志内容。由于许多开源的日志传输工具都是用来解析和传输**syslog**里的日志数据的，所以**syslog**是一个非常方便的日志方案。用户可以轻松地以传统的方式复用这些工具，通过将它们直接运行在宿主机上，然后指定宿主机的**syslog**作为它们的日志源，如此一来用户会感觉自己好像回到了没有Docker的时代。

最后，Docker还支持参数为**none**的日志选项，也就是说它会忽略应用程序生成的所有日志。同样地，和**syslog**驱动类似，必须在启动容器时明确指定该参数才能生效。虽然我们不建议在生产环境中运行的应用采取这样的不捕获任何日志的方式，但是它在某些场景下的确是一个非常方便的选择，例如一些容器会生成大量的无用日志然后可能导致宿主机

上的文件系统出现真实的I/O灾难。

15.1.2 连接到Docker容器

通常，我们不建议在容器或者宿主机操作系统里查看日志文件。然而，在排障时，用户可能需要根据自己的排查顺序查看一些正在运行的容器里的额外日志信息。接下来，为了继续调查，用户可能需要连接到一个正在运行的容器上。为了连接一个正在运行的容器，用户需要做一些额外的、繁琐的且重复的操作来实现安全访问（**root**和**ssh**）。但是，如果的确需要这样做，用户可以使用Docker原生提供的**attach**命令。下面是一个使用该命令的实际用例：

```
docker attach redis
```

连接容器即是连接到该容器里正在运行的进程的**shell**上。当连接到容器内部时，由于大多数的镜像都是非常轻量地在运行单个进程，因此用户可以在容器操作系统里执行的命令往往也会非常有限。

Docker在1.3版本推出了**docker exec**子命令，它允许用户在正在运行的Docker容器里执行任何命令，包括用户最喜欢的**shell**。这种方式比将TTY连接到正在运行的容器要优雅得多。用户可以通过执行如下命令轻松来运行**bash**（如果容器的镜像里装了**bash**的话），随后便可以在容器文件系统里查看日志或任何其他文件：

```
docker exec -it redis /bin/bash
```

上述命令将会在运行**redis**服务的容器里再运行一个新进程。一旦进入到容器里，用户便可以看到一些可能不会被记录到**stderr**或者**stdout**的额外的日志文件。这可以用来帮助调试故障的应用或根据日志文件的内容来评估性能。然而，这并不是一个适用于生产系统的可扩展方案。生产系统一般需要的是一个集中式的日志平台来查看汇总的日志流。

15.1.3 将日志导出到宿主机

如今，企业通常运行或使用的都是集中式的日志系统。在传统的服务器环境里，人们通常会使用各种客户端软件（如**syslog**）来读取文件系统上存放日志的目录或路径，然后将它们发送到一个集中平台上。为了

利用集中式日志系统，用户可能需要将容器里的日志文件转移到宿主系统上。实际上，将Docker日志存放在宿主系统上并不是一件难事。将日志记录到宿主系统里常用的有两种方法。用户可以在Dockerfile镜像里使用VOLUME指令或使用docker run -v参数将容器里的一个文件路径挂载到宿主文件系统上的一个位置上。

当需要将日志存放在宿主系统上时，Docker官方推荐的做法是采用docker run -v volume选项来挂载卷。使用-v参数便可以灵活地指定文件系统重定向输出的位置。默认情况下，Dockerfile里的VOLUME指令使用的是容器里的路径，其对应的一般是一个根文件系统的位置。如果用户的根文件系统已经满了，这就会导致用户的宿主机出现一些大问题，而且它很难被清理干净。例如，运行一个redis容器并且将它配置为日志存放到/redislogs/redis.log，然后将日志文件存放在宿主系统中。

如果redis配置文件被设置为将日志放到/redislogs/redis.log文件里，用户可以很轻松地在Dockerfile里通过VOLUME命令（即VOLUME /redislogs）将日志归档。在镜像里使用VOLUME指令会导致Docker运行时在容器里的/redislogs位置上创建一个新的挂载点，然后将内容复制到宿主系统上新创建的卷目录。然而，宿主系统上的卷目录并不是一个众所周知的位置（一般地，它可能会是/var/lib/docker/volumes/7d3b93bf75d687530a159ae77e61b03这样的路径）。在这种情况下便很难再在宿主机上去配置一个集中式的日志收集客户端来收集数据卷里的文件内容。这也正是我们推荐使用docker run -v的方式来将日志持久化的原因。

通过在启动容器时使用docker run -v选项，用户可以从本质上将容器里的目录或者文件路径重定向到宿主文件系统上的某个位置上。一般将容器里的日志目录“重定向”到宿主系统的日志位置的做法是使用如下指令：docker run -v /logs:/apps/logs。

当用户把容器的日志重定向到宿主机上的某个集中的地方时，如/logs/apps，用户便可以轻松地配置一个日志采集客户端，从/logs/apps里收集所有的日志，然后将它们发送到集中的日志系统里。这也适用于在单台宿主机上运行多个容器实例的场景。假设用户在同一台宿主机上运行了nginx、redis以及一个worker容器的实例。通过执行如下docker命令用户将可以在/apps目录里得到3个日志文件：

```
docker run -v /logs/apps/nginx/nginx.log nginx
docker run -v /logs/apps/redis/redis.log redis
docker run -v /logs/apps/worker/worker.log worker
```

运行在宿主机上的日志采集客户端，如果它被配置为从`/logs/apps/**/*.log`读取日志，便能够收集所有的3个日志文件然后将它们发送出去。以上便是一个简单的整合用户的容器及日志采集客户端的方法，这样一来，开发/运维团队便都能轻松记住该到哪去查找日志文件。

15.1.4 发送日志到集中式的日志平台

如果要将日志发送到一个集中式的日志系统里，除了将它导出到宿主机似乎别无他法。实际上，用户无需把日志文件重定向到宿主系统。取而代之的是可以采取其他的办法将日志导出，例如，用户可以在容器里的应用进程一侧运行一个日志采集客户端，然后将收集到的日志通过网络直接发送出去。如果运维团队支持在一个容器里运行多个进程，用户可以把容器简单地看成是一台服务器，然后在容器启动的时候安装和配置该客户端。

这里以标准的**syslog**为例。运行在容器里的应用会把日志传到`/logs/apps`里，然后安装和配置一个**syslog**守护进程，从该目录下读取任何存在的日志信息。随后，它被配置成可以将日志发送到一个集中式的日志服务器里。一旦该目录下有新文件加进来，**syslog**守护进程将会如预期那样读取和传输对应的日志数据。

这一模式的确也很有价值，然而它存在一些明显的短板。这些短板包括，若在一个容器里运行多个进程，这可能会导致容器在进程利用方面变得更加笨重，而且它在服务启动脚本里增加了一些额外的复杂度，并使得应用隔离的便利性荡然无存。在这个模式下，如果一个系统要运行10个容器，用户就得在宿主机上运行10个**syslog**服务。如果这些**syslog**客户端能够被整合到一个单一的进程的话性能也许会更好。

将日志导出到在线系统的另一种办法是直接从应用中获取。假设在容器里运行一个Java应用，该Java代码被配置成向代码函数中写入日志文件。该函数还可以设置成将日志发送到远端服务器上的一个集中式队列里（如**Kafka**），之后这些日志数据会交由其他系统处理。这一方案使得开发人员能够很轻松地配置一个可扩展到任意集中式系统的日志模

型。然而，这一模式只适用于应用在远程日志采集函数方面均采用同样一段Java代码的场景。如果另外有一个容器运行的是一段Python脚本，那么便意味着就得再写一个Python实现的远程日志函数。

15.1.5 在其他容器一侧收集日志

这最后一个从容器里获取日志的方案在业界正逐渐变得流行起来。通过在系统上的其他容器之间使用共享的数据卷，用户便可以在其他正在运行的容器里提取自己所需的数据（在这个例子里即日志）。我们喜欢将这个从其他容器一侧挂载日志的做法比作在一辆摩托车上安装了一个侧座。在一个系统上有多个容器的场景下采用这一方案可以节省一些不必要的处理时间。例如，如果系统上的每个容器都通过运行一个服务的方式来发送它的日志，那么由于每个容器里运行的是冗余的服务，因此会浪费一些不必要的资源。用户可以把各个容器里所有的日志放到一个单独的容器里面，然后将其集中起来并发送到一个集中式的日志系统。由于将所有的事务都整合到了一个统一的日志采集服务，因此用户便可以节省一大笔的资源。

我们不妨用两个容器来举一个简单的例子。一个容器运行一个**redis**服务，并且将日志事件发送到它的容器中的/logs/redis.log位置，对应的即是一个数据卷/logs。如果我们在**docker run**执行时运行标志**--volumes-from**来启动另外一个容器，用户就能在这个新容器里提取/logs数据卷的数据。当然，这样做可能在概念上不容易理解，所以接下来我们会列举一些实际的命令来帮助用户更好地理解这一过程。

首先，我们会启动一个新的名为**redis**的容器。随后，我们会使用**-v**参数创建一个新的数据卷，由于我们之前通过**--name**参数指定了该容器的名称，因此其他容器可以很方便地从它这里提取数据。

```
docker run -d -v /logs --name redis registry/redis
```

然后，我们可以启动一个日志收集容器，使用**--volumes-from**将第一个容器创建的/logs数据卷挂载到容器里面。注意别忘了带上**redis**的名字。

```
docker run -d --volumes-from redis --name log_collector  
registry/logcollector
```

这使得一个运行了大量Docker容器的系统具备了访问共享资源的能力。这类案例在像Mesosphere这样的网格式部署场景中尤为常见。

Docker提供了多种不同的方案来采集和查看日志。我们建议的做法是选择一个最合适自己的基础设施的、易于维护且可扩展的方案。一般公司最终选择的也会是那套在他们环境里工作最稳定的模式。那么，现在让我们一起来看看该如何监控Docker容器吧。

15.2 监控

对于Docker容器的监控归根到底是用户要怎样监控运行在容器里的服务以及采集哪些指标数据。用户所采取的Docker容器监控的手段实际上取决于现有的工具以及监控的形式。我们建议选择那款团队喜欢并且用的舒心的工具。大公司里一般可能采用的会是一些业界成熟的监控工具，如Nagios，而当需要监控一项像Docker这样的新兴技术时可能就需要发挥自己的创造力了。创业型公司则往往一开始使用的就是最新潮、最棒的技术，而它们一般都已经加入了对Docker的支持，如New Relic、Datadog或Sysdig。无论哪款工具，在绝大部分的场景下一般都能够工作得很棒，因此这只关乎团队究竟偏向于哪个以及它本身的成本收益比。

从某种意义上讲，监控和从容器里获取日志的形式非常相似。用户完全可以通过像使用一个init文件来管理运行在系统上的服务那样去监控一个正在运行中的容器。另外一种选择是去监控Docker的子系统（使用`docker ps`或`docker stats`），只要容器仍然在运行，它就可以被认为是健康的。有些公司甚至把监控客户端像传统服务器那样放到了容器里面。用户可以通过访问应用层的如`http://<service>/health`这样的端点服务或一些其他措施来监控容器的健康状况。最后，如果用户采用的是Etsy工程方案，用户可能会对系统里的所有事情做些相应的权衡，最终可能会选择通过在宿主机上、Docker服务以及容器本身的健康状况方面采集相应的指标数据来完成监控。

许多已经开始使用Docker容器的企业就等于已经切换到了一个面向服务的架构（SOA），这可能就需要建立一套全新的监控标准。我们打算深入探讨如何监控SOA环境，但是读者应该意识到这里面的区别。SOA里的服务通常都是以临时模式运行的。这就意味着，如果服务运行在容

器里的话可能会挂掉，但是，这并没有什么太大问题。一个服务可以使用运行相同服务的多个容器以负载均衡的方式承载，然后如果它们停止服务的话会自动重启，并自动去重新注册自己的服务发现，随后它们便可以恢复并且这中间没有任何的宕机损失。一个负载均衡的临时系统通常监控的重点会是应用服务本身的健康状况而不是单台服务器上的单个服务。绝大多数情况下这便意味着服务需要一个外部系统来监控其应用层的状态或监控它服务端口的可用性，这可能就需要利用一些新的监控工具才能办到。

监控服务器、服务及应用的方式多种多样。下面我们通过一款名为Datadog的工具来监控一个简单的环境，然后运行一个简单的Python脚本来监控HTTP端点服务（如图15-1所示）。Datadog提供了一个基于主机的客户端来监控服务器，然后上面运行一个statsd的后端服务，用来完成应用级别的监控。在这个例子中，我们将会运行一台单独的服务器，然后上面运行一个单独的Docker容器。我们会监控宿主机的CPU、内存以及磁盘IO，并且会从Docker守护进程里收集一些统计数据。紧接着，我们会使用一个运行在其他系统上的Python脚本来监控应用的HTTP端点服务，它会有一些指标数据发送到StatsD后端。最后，我们将端对端地监控一个在Docker容器里运行的系统。

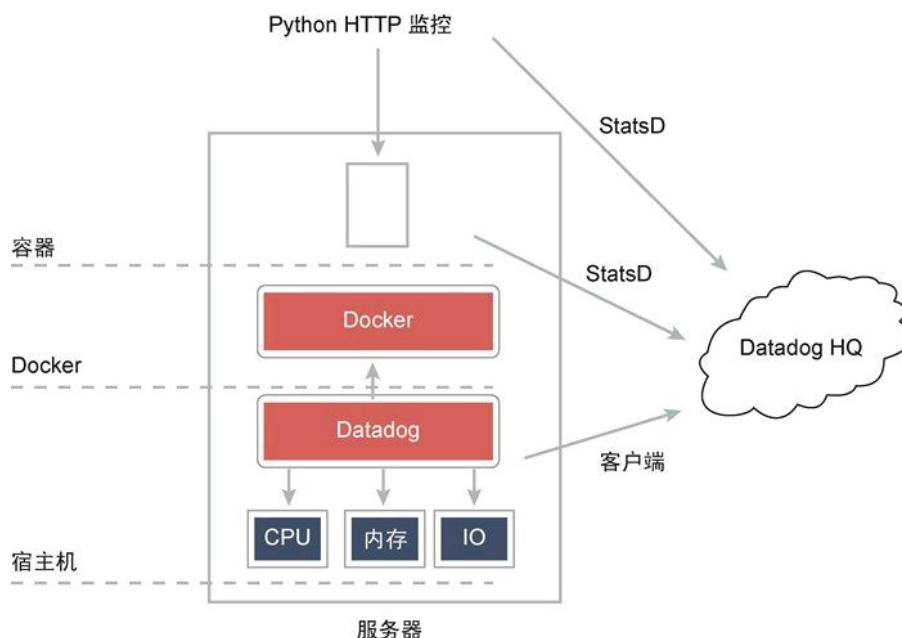


图15-1

15.2.1 基于宿主机的监控

大多数服务器所处的环境里都会部署一个监控系统，它会负责CPU、内存以及磁盘IO这些传统的监控指标的处理和告警。当它们监控的是宿主机上的这些指标时，大部分监控系统也许还能正常运转。但是如今应用服务是运行在Docker里的一个单独进程，使用一些如`ps aux|grep nagios`或`service nagios status`之类的传统命令来监控它们不太容易。这时候，用户就需要查看自己的监控系统是否能监控Docker进程的状态或者能否调用它提供的API来提取健康状况的数据。如果它还没有集成Docker的话，要是用户觉得有必要，那么可以考虑自行采取一些措施来监控它。

在这个案例场景里，使用Datadog来监控宿主机是一件非常简单的事情。Datadog的客户端和绝大多数传统的监控系统客户端一样，它会安装到本地然后将采集到的监控指标数据发送到一个集中式的系统里。这就跟Nagios、Zabbix或Hyperic监控客户端做的事情没什么两样。安装Datadog客户端很简单，用户只需要照着Datadog的安装命令就能把它们的主机客户端（在这里即Ubuntu）快速地配置起来。这里我们使用的命令是`DD_API_KEY=cdfadffdad9a... bash -c "$(curl -L https://raw.githubusercontent.com/DataDog/dd-agent/master/packaging/datadog-agent/source/install_agent.sh)"`。在客户端安装成功后，它便会立即开始将所收集到的数据发送到Datadog的基础设施中。几分钟后，宿主机便能够将用户所有的CPU、内存和磁盘IO的具体指标收集到它们的监控数据浏览器和汇总看板里。一旦数据汇总到了系统里，我们便可以据此设置告警邮件，或如果某些员工需要还可以定制一些告警页面。图15-2所示是采集后的数据展示的一个简单看板。

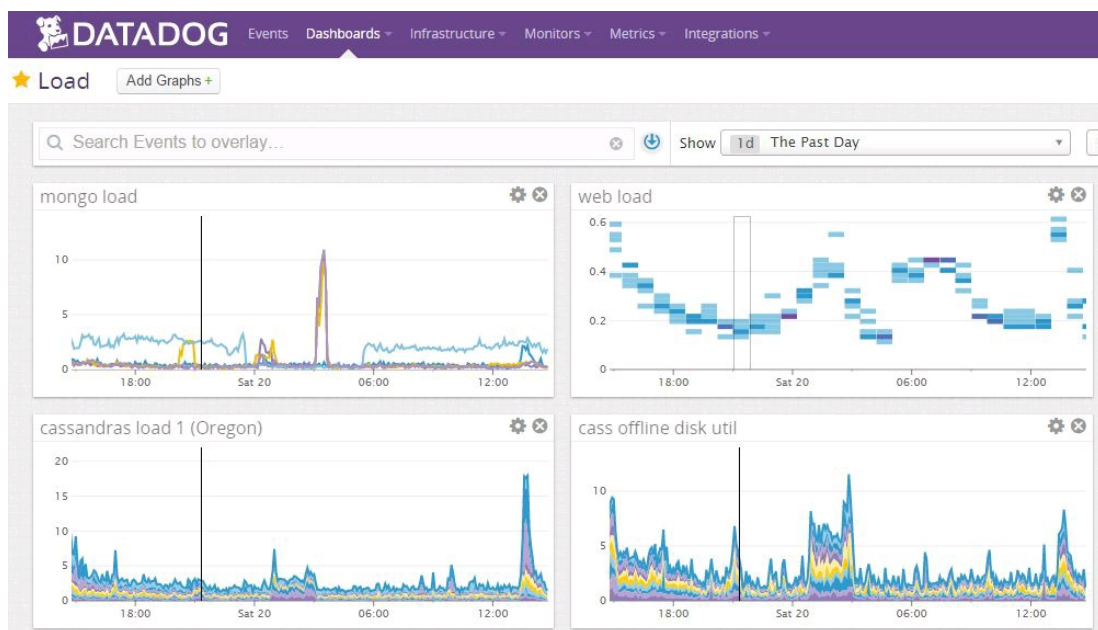


图15-2

在这种情况下，用户最多只能看到大多数基础设施团队针对应用环境想要监控内容的三分之一。在这个场景里，我们只能看到主机层面的监控指标，如CPU、内存和磁盘IO。我们无法知道系统里单个进程或服务的使用情况。最可能出现的场景便是当CPU占用率过高时，用户想知道大部分资源具体是被哪个容器，进程或者服务占用的。问题的核心在于我们需要采集系统上运行的每个Docker容器的性能指标，然后据此制定相应的策略，以减少平均维护时间（MTTR）。就目前来说，我们对于系统上运行的Docker服务以及正在运行的容器的性能或者健康状况几乎一无所知。因此，接下来让我们一起来看看如何监控Docker服务以及如何才能采集到相应的指标数据。

15.2.2 基于Docker守护进程的监控

Docker提供了多种方式来监控容器以及从系统中采集数据。首先，我们将介绍Docker原生提供的几个不同的命令。随后，我们将介绍如何使用Datadog来监控Docker守护进程以及怎样采集容器的监控指标。默认情况下，Datadog不会通过它们的API去采集Docker的监控指标。用户需要到它们的网站上开启集成支持。一旦开启该集成，Datadog便能从Docker中采集相应的监控数据，这样一来用户便可以实现大规模的容器的监控。在我们讲解例子之前不妨先使用`docker ps`看一下容器当前的状态。

当在Docker容器里运行服务时，用户将无法再依赖以前的一些众所周知的工具来监控容器的进程。由于服务或进程是以容器的形式在宿主系统中运行，因此除了Docker服务本身，用户将无法查看它们具体的健康状况。以前像ps这样用来检查一个进程是否运行的工具，如今针对运行在Docker下的服务则被抽象需要运行docker ps或者docker info才能获取到当前作为容器运行的服务的具体信息。

docker ps是一个简单的、返回当前所运行容器的状态的命令。要运行它也非常简单。用户只需要在其安装了Docker的系统上输入docker ps即可。用户可能无法看到容器相关的所有信息，但是可以得到绝大部分的重要指标以及容器当前所处的状态信息。有些企业甚至只需要运行docker ps便能满足他们所有的需求。它允许用户查看当前一个特定容器是否在运行，如果没有的话可以将它重启。下面这个简单案例即是运行一段bash脚本来监控一个特定的Redis容器的运行状态。如果该容器没有处于运行状态就会去重启它：

```
#!/bin/bash

# Check for running Redis container, if its not running then
start it
STATE=$(docker inspect redis | jq ".[0].State.Running")
if [[ "$STATE" != "true" ]]; then
    docker run -p 6379:6379 --name redis -v /logs/apps/redis:/
logs -d hub.docker.com/redis
fi
```

这是一个非常简单和初级的展示如何监控容器运行状态的用例，而这也是许多企业目前的做法。绝大多数生产环境的编排组件采取类似方法来维护和自动化监控系统的健康状况。当用户所处的环境里Docker的部署已经达到了一定的规模，并且在任意给定的时间内均有大量的容器运行，那么用户很可能需要一款更棒的工具和编排手段。如果想构建自己的容器监控组件（很多公司已经这么做了），Docker除了使用Bash脚本外还提供了其他方式来获取容器的状态，如使用[Docker Python API](#) `docker-py`或使用Docker的远程API。用户甚至可以使用Docker的集群管理系统，如[Shipyards](#)，来查看多个系统里的不同容器的状态。

现在，读者应该了解了该如何获取系统上容器的运行状态。那么，接下来我们需要做的便是获取容器的性能指标。就以之前系统出现过的高CPU占用率的情况为例。我们需要找出系统里是哪个容器占用了大部分

的CPU资源。在这个场景下，用户需要了解Docker提供的另外一个命令。Docker自1.5以上的版本提供了一个很棒的统计API及命令行工具，即**docker stats**，它可以获取容器的实时性能指标，如CPU、内存、网络IO、磁盘IO和块IO。在该API开放之前，它实际上是cgroup使用的一些性能指标的原始导出数据。大部分监控公司采纳了这个新的API，并尝试将其用于从Docker系统中获取更多的监控指标。用户也可以借此命令推出自己的监控系统。**docker stats**命令为用户提供了大量的容器运行时的信息，因此用户可以据此构建整合了所需信息的更复杂的看板。

使用**docker ps**和**docker stats**命令已经让用户在监控容器的基础状态的道路上前进了一大步。然而，当用户将容器从单个系统扩展到更大规模时，用户还需要一个工具来把采集的监控指标聚合起来并扩展到一个易于使用的集中看板上。让我们再回顾下使用Datadog的这个案例，看看如何在大规模场景下用它来完成监控。

Datadog是通过使用Docker守护进程的套接字来访问Docker API从而实现与Docker的集成。通过访问该API，Datadog能够监控当前宿主系统上运行了多少容器，Docker的CPU和内存占用情况，Docker容器状态更改的事件流，cgroup一些不常见的指标数据，以及Docker镜像的统计信息等。它会为用户提供非常丰富的指标信息，展示当前在容器上运行的容器化服务的健康状态和性能的具体情况。再次重申一遍，默认情况下，Datadog不会使用Docker API来监控，因此用户需要到他们的官方网站上去手动开启这一功能。在开启该功能之后，用户还需要将它加到系统上的Docker组才能让Datadog客户端有权限访问Docker。一旦准备工作就绪，Datadog就会开始从系统里采集监控数据，并自动将这些数据发送到基础设施里。如此一来，用户便能设置自身团队所需的特定Docker监控数据的看板和告警。依照[Datadog提供的具体例子](#)，用户可以使用他们的Metrics Explorer查看对应的监控指标（如图15-3所示）。

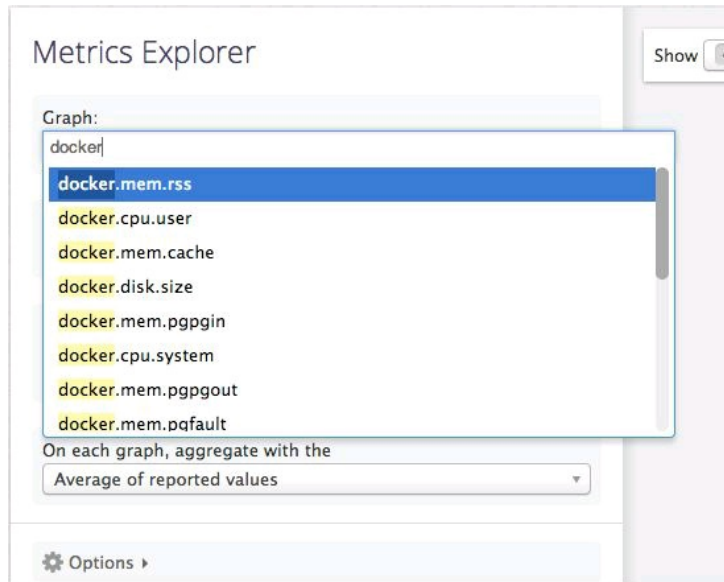


图15-3

通过Datadog与Docker的集成，我们得以了解单个宿主机或多主机上有多少容器在运行，容器状态变更的事件流，以及特定镜像的一些指标信息。运维和开发团队如今均可以使用这些信息来实现他们容器级的服务状态及性能的监控。如果有大的变动，我们可以给运维团队发出告警并通知他们这些问题。通过Datadog提供的Docker监控集成支持，用户可以获取一些普通主机级监控客户端不能提供的抽象指标参数和信息。如今，我们能够采集基于宿主机和Docker两方面的监控数据，这样一来整个系统具备了更高的可视化程度。这里面唯一漏掉的便是运行在容器里的应用程序本身的健康状况。现在，让我们一起来看看整个案例的最后一个部分吧。

15.2.3 基于容器的监控

用户可以通过多种途径来监控运行在容器里的应用的健康状态。由于容器本身是一个完整打包的操作系统，因此可以在容器里面运行任意类型的应用程序。用户可以把正在运行的容器简单地看做是一台普通的服务器，然后在镜像创建过程中在容器里安装一个基于主机层面的监控客户端。但是，我们强烈建议将容器尽可能地保持在比较轻量水准，最好只运行单个进程。虽然我们不建议一个容器里运行多个进程，但是有些公司成功地这样做了，并且它的好处在于有时候可以很方便地与用户现有的监控设施集成。在这个例子里，系统上的容器会运行一个简单

的HTTP服务并且对外开放一个/health的端点。为了确认运行在容器里的应用的健康状态是好的，我们可以通过多种手段对它进行监控。用户可以使用StatsD或其他自定义监控框架，如bash或者Python脚本，来监控它。我们先来看看StatsD的方案。

StatsD是一款轻量级的监控工具，它是由Etsy创建的用来简化监控数据的统计和聚合，可以通过网上的一篇[博客](#)了解到更多细节。Statsd因它轻量和大规模场景下良好的扩展性的特点，已然成为一款非常受欢迎的应用层监控服务工具。针对一个运行在容器里的应用的健康状况的监控，其中一种方案是调用Statsd的库将应用的指标数据发送出去。例如，有个运行在容器里的Web应用，它会获取POST请求，并据此处理一些信用卡数据。为了追踪究竟收到了多少个POST请求，每当接收到一个POST请求时它会把对应的指标数据发送到StatsD服务器。该服务器随后会通过连接到信用卡中心提供的API来处理相应的信用卡数据。用户可以在连接该API的代码里设置记录开始访问的时间。然后在调用返回且信用卡数据被处理之后，代码部分再次记录对应的结束时间。该代码随后计算两个记录时间之间的差值从而获取该任务总共的耗时，然后将另外一个StatsD指标发送出去。而每当它成功响应一次客户端提交的POST请求时，它会发送最后一个标志成功的StatsD指标以记录总共处理的信用卡数量。在本例中，如果将所有采集到的StatsD监控指标绘制到一个看板上，用户便能几乎实时地观察应用的健康状况。用户能看到该容器接收到了多少个POST请求，调用API所花费的时间，以及完成处理的信用卡数量。这是测量运行在容器里的应用健康状态的一种方式。例如，若看板上已完成的信用卡处理数量突然下降，用户可以给它发送一个包含其详细信息的告警。

在本例的最开始我们曾经讲过一个/health的端点服务，那么接下来我们将具体介绍一下这方面的内容。StatsD（和Datadog的具体实现）会将任意字符串的输入看做是一个指标，然后将它们聚合汇总后展示给用户。例如，如果我们想监控http://myservice.corp/health这个http端点服务，可以使用`echo "myservice.status- code:curl -sL -w "%{http_code}" "http://myservice.corp/ health"|c" | nc -u -w0 statsd.server.com 8125`。该命令会检查服务器上这一http端点服务返回的状态码，然后借助netcat将数据发送给StatsD。Datadog还可以和StatsD集成在一起，专门针对用户发送给statsd的监控数据做监控和告警。在本例中，如果该服务是健康的，它会返回一个200的状态码。Datadog将会收到myservice.statuscode的监控数据，而它的值即是

200。如果该状态码返回的是一个500或者404错误，我们随后便可以使用Datadog通过邮件或专门的呼叫系统去发送一个故障通知。下面的这个Python脚本即是一个采用Datadog结合StatsD实现的具体例子。

```
import requests # For URL monitoring
import statsd # We installed the Datadog statsd module
import sys
import time

sites = ['http://myservice.corp/health']

def check_web_response_code(url):
    r = requests.get(url, allow_redirects=True, verify=False, stream=True)
    return str(r.status_code)

def send_dogstatsd(options, site):
    c = statsd.DogStatsd(options.statsd, options.statsport)
    c.connect(host=options.statsd, port=options.statsport)
    statname = 'httpmonitor'
    tags = []
    tags += ['site:'+site]
    r = check_web_response_code(site)
    c.gauge(statname, r, tags=tags)

def monitor_sites(options):
    for site in sites:
        send_dogstatsd(options, site)

def main():
    while True:
        monitor_sites(options)
        time.sleep(30);

if __name__ == '__main__':
    sys.exit(main())
```

用户可以让上面这个简单的Python脚本在一个容器里运行，然后通过它来监控其他容器的健康状况。这是一个非常简单、轻松地获取应用健康状况的方法。读者可以到StatsD的GitHub项目中了解更多详细内容以及它所提供的额外的核心类库。

15.3 小结

总体来说，上述案例已经较为完整地介绍了如何端对端地监控一个系统上的Docker以及正在运行的容器。借助一个单独的工具，我们便能监控服务器本身、Docker守护进程以及容器内的应用服务。在上述案例中，通过使用Datadog，应用服务的监控数据可以被整合到Datadog的复杂看板上，Datadog提供了宿主机、Docker服务及应用健康状态的告警。这是一个非常基础的例子，但是我们希望它可以展示该如何着手监控运行在Docker基础设施里的应用服务。

而这一点也贯穿于整本书。我们希望读者已经了解了在生产环境使用Docker所需要的工具和信息。

DockOne社区简介



DockOne.io成立于2014年，是国内最大的容器技术社区。社区主要关注 Docker、Mesos、CoreOS、Kubernetes、Ceph、OpenStack等容器生态圈相关软件，致力于为广大容器爱好者提供一个分享、学习和交流的平台，目前已有活跃用户逾50000，精品文章1000余篇。



看完了

如果您对本书内容有任何疑问，可发邮件至contact@epubit.com.cn，会有编辑或作译者协助答疑。也可访问异步社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@epubit.com.cn。

在这里可以找到我们：

- 微博：@人邮异步社区
 - QQ群：368449889
-