Docker High Performance

高性能Docker

掌握Docker性能优化实践，更快更高效地部署容器，改善开发工作流

[美] Allan Espinosa 著
DockOne社区：陈杰 杨峰 夏彬 译

中国工信出版集团
電子工業出版社

高性能Docker
[美] Allan Espinosa 著
DockOne社区：陈杰 杨峰 夏彬 译
電子工業出版社

Docker High Performance

高性能Docker

[美] Allan Espinosa 著
DockOne社区：陈杰 杨峰 夏彬 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书共分8章，旨在帮助读者改善其Docker工作流，并保证应用在生产环境中顺利进行。

书中简单回顾了Docker是如何工作的。除了Docker的基础知识外，读者还会学到如何优化Docker基础架构和大规模应用。本书最后讲解的如何在基础架构中部署监控和故障排除系统，更是可以让读者更好地将学到的Docker的特性、概念等运用到实践中。

如果你对于管理Docker服务和Linux文件系统有充分的理解，并希望优化你的Docker容器，那本书将非常适合你。

版权贸易合同登记号图字：01-2016-3365

图书在版编目（**CIP**）数据

高性能Docker / （美）艾伦·埃斯皮诺萨（Allan　　Espinosa）著；陈杰，杨峰，夏彬译．—北京：电子工业出版社，2016.9

书名原文：Docker High Performance

ISBN 978-7-121-28963-7

Ⅰ．①高…　Ⅱ．①艾…②陈…③杨…④夏…　Ⅲ．①Linux操作系统－程序设计　Ⅳ．①TP316.89

中国版本图书馆CIP数据核字（2016）第123021号

策划编辑：张春雨　刘　芸

责任编辑：刘　舫

印　　刷：北京中新伟业印刷有限公司

装　　订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱

邮　　编：100036

开　　本：787×980　1/16

印　　张：9

字　　数：186千字

版　　次：2016年9月第1版

印　　次：2016年9月第1次印刷

定　　价：69.00元

译者序

Docker容器技术从LXC技术发展而来，是一个程序运行、测试和交付的开放平台。其中，可以将不同功能的虚拟主机以一个应用程序的方式进行管理，从而帮助用户实现快速测试、编码和交付。Docker从出现，到开源，到被新一代技术架构整合，到各种初创公司，无疑一直带有传奇色彩，为IT技术在中国“互联网+”创业浪潮中增添了一抹亮色。

围绕Docker的生态系统，目前在大厂商如Google，以及本地创业公司的共同推动下，在图像化管理、作业调度、作业编排等领域都有了长足发展。另外，目前最新的发展趋势是，直接在Mac和Windows中使用的Docker已经推出了beta版本，以一个后台运行的应用形式表现出来。

Docker更多与IaaS、PaaS的生态系统对接起来，围绕客户管理、维护、排错和作业调度等需求，出现各种管理方式，例如Kubernetes和Swarm等编排工具。其趋势也是将底层Docker的具体启动、运行、暂停和停止进行包装，以便客户更好地专注于Docker平台之上的应用，简化底层资源层的管理和维护。

但是对于万千技术爱好者而言，如果不能深入Docker内部进行运行机制研究、不能对Docker内部性能进行调优，那么就不能很好地应用这一新技术带来的益处。因为从深入研究技术的角度来看，不仅需要使用集成化工具带来的简便，更需要深入了解Docker底层的机制，

以便真正需要时，找到面临问题的解决方案。

电子工业出版社一直紧跟国外技术热点，适时引进了*Docker High Performance*原版书，它是一本面向有一定Linux基础、想深入了解Docker内部机理以及如何监控和提高容器性能的Docker初学者的书。

全书共分8章，每一章都有详细步骤讲解，从裸机开始，到最终运行一个模拟系统为止，读者如能全部操作下来，必将从中获得很大收获。

本书译者都是Docker社区很有经验的志愿者，在百忙之中抽出宝贵时间进行翻译、校对，希望能对容器技术在国内的推广和使用做出相应的贡献。因为容器技术发展很快，各位译者竭尽所能展现最新技术，但是水平有限，错误在所难免，希望读者能够批评指正。

杨峰

2016年7月

关于作者

Allan Espinosa是一名生活在东京的DevOps从业者，他是很多分布式系统工具的活跃的开源贡献者，比如Docker和Chef。Allan维护了若干个流行的开源软件的Docker镜像，这些镜像甚至比开源团体的官方发布版还要流行。

在他的职业生涯中，Allan还管理过一些大型分布式系统，包含生产环境中的数百到数千台服务器。他在不同的平台上构建了很多大规模应用，从美国的大型超级计算中心到日本的生产环境企业系统。

你可以通过Allan的Twitter账号@AllanEspinosa联系到他。他的个人网站是http://aespinosa.github.io，其中有很多关于Docker和分布式系统的博客文章。

我要感谢我的妻子Kana，她一如既往地支持我，使我能够花费大量的时间来写作。

关于审校者

Shashikant Bangera是一名DevOps架构师，具有16年的IT领域经验。他对于开源DevOps工具具有丰富的专业知识。Shashikant曾经参与大量的价值数百万英镑的项目。从传统的开发实践到敏捷工具和流程的使用的转变这方面，他具有丰富的实践经验，这可以提高发布频率和软件质量。此外，Shashikant已经使用开源工具设计了一个自动化的、按需的（on-demand）环境。对于大量的DevOps工具，他也具有丰富的实践经验。

Packt Publishing的另一本书*Learning Docker*也是由Shashikant审校的。

目　　录

前言

Docker是一款很好的用于构建和部署应用的工具。可移植的容器格式使我们可以在任何地方运行代码，从开发者工作站到著名的云计算提供商。Docker的工作流使开发、测试和部署更加容易和快速。然而，Docker核心和最佳实践的持续改善是很重要的，可帮助你实现Docker最大的潜在价值。

本书的主要内容

凡是对Docker有基本理解的工程师都可以按顺序一章一章地阅读本书。对Docker具有深入理解或者在生产环境中部署过应用的技术领导者们，可以直接阅读第8章的内容，了解Docker是如何适应已有应用的。以下是本书介绍的一系列主题。

第1章，简单介绍了如何搭建和运行Docker，介绍了贯穿本书都会用到的搭建步骤。

第2章，介绍了为什么调优Docker镜像是很重要的，介绍了多个调优小窍门，从而改善可部署性和Docker容器的性能。

第3章，介绍了如何自动化搭建Docker宿主机，并讨论了自动化的重要性以及它是如何促进Docker容器的大规模部署的。

第4章，介绍了如何使用Graphite搭建监控系统和使用ELK搭建日志系统。

第5章，介绍了如何使用Apache JMeter来创建负载，并测试Docker容器的性能。本章回顾了第4章中搭建的监控系统，并分析了若干Docker应用的性能基准结果，例如响应时间和吞吐量。

第6章，介绍了如何配置和部署基于Nginx的负载均衡容器。同时，也介绍了如何使用负载均衡器来水平扩展Docker应用的性能和可部署性。

第7章，介绍了典型Linux系统中的通用调试工具是如何调试Docker容器的。并介绍了每种工具是如何工作的以及是如何读取运行中的Docker容器的诊断信息的。

第8章，综合了前面几章中的性能优化方法，并介绍了如何在生产环境中使用Docker部署任何一个Web应用。

你需要做什么准备

你需要一台安装了最新版本内核的Linux工作站作为Docker 1.10.0的宿主机。本书使用Debian Jessie 8.2作为基础操作系统来安装和搭建Docker。

本书的目标读者

本书是为想要在生产环境中部署Docker应用和架构的开发者和运维者而写。如果你已经了解了Docker的基本知识，并且想进一步学习Docker的话，那么这本书就是适合你的。

惯例

在本书中，我们使用了不同的文本格式，用以区分不同类型的信息。以下是这些格式的例子以及它们的具体含义。

文本格式的代码、数据库表名、目录名、文件名、文件扩展名、路径名、URL、用户输入、Twitter用户名都是用以下格式书写的：“我们会使用--link<source>:<alias>来创建源容器source到另一个容器webapp的连接”。

代码块的格式如下：

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get \
       install -y -q python-all python-pip
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```

当我们希望着重表示代码块中的某一部分时，这一部分就会被设置为粗体。

```
import os
from flask import Flask
app = Flask(  name  )
@app.route('/')
def hello():
    provider = str(os.environ.get('PROVIDER', 'world'))
    return 'Hello '+provider+'!'
if   name   == '  main  ':
```

```
    # Bind to PORT if defined, otherwise default to 5000.
    port = int(os.environ.get('PORT', 5000))
    app.run(host='0.0.0.0', port=port)
```

命令行输入或输出的格式如下：

```
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddre
                          source
172.17.0.15
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddre
                          destination
172.17.0.28
dockerhost$ iptables -L DOCKER
Chain DOCKER (1 references)
target  prot opt source       destination
ACCEPT  tcp  --  172.17.0.28  172.17.0.15  tcp  dpt:5000
ACCEPT  tcp  --  172.17.0.15  172.17.0.28  tcp  spt:5000
```

警告或者重要注解会出现在这样的图标之后。

提示和技巧将会出现在这样的图标之后。

下载示例代码

你可以从http://www.broadview.com.cn下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

1　准备Docker宿主机

Docker使应用交付到客户更加便捷。它通过简单地创建和运行容器这种形式，简化了代码从开发环境到生产环境部署过程中的工作流。本章将快速学习如何准备一个基于Docker的开发环境，具体的操作步骤如下：

- 准备一个Docker宿主机
- Docker镜像的基本操作
- 运行Docker容器

本章的大部分内容是我们已经熟悉的知识，并且可以在Docker官方文档网站上查阅到。这里将介绍的是与Docker宿主机有关的部分命令和在后续几章中将使用到的交互操作。

准备一个Docker宿主机

假定读者熟悉如何创建一个Docker宿主机。对于本书的大部分章节，除非有特别说明的情况，我们都将在如下环境中运行示例：

- 操作系统：Debian 8.2 Jessie
- Docker版本：1.10.0

下面的命令显示了操作系统和Docker的版本:

```
$ ssh dockerhost
dockerhost$ lsb_release –a
No LSB modules are available.
Distributor ID: Debian
Description: Debian GNU/Linux 8.2 (jessie)
Release:       8.2
Codename:      jessie
dockerhost$ docker version
Client:
  Version:        1.10.0
  API version:    1.21
  Go version:     go1.4.2
  Git commit:     a34a1d5
  Built:          Fri Nov 20 12:59:02 UTC 2015
  OS/Arch:        linux/amd64

Server:
  Version:       1.10.0
  API version:   1.21
  Go version:    go1.4.2
  Git commit:    a34a1d5
  Built:         Fri Nov 20 12:59:02 UTC 2015
  OS/Arch:       linux/amd64
```

如果还没有创建Docker环境，可以按照Docker官方网站上的说明进行安装，网址是：https://docs.docker.com/installation/debian。

下载示例代码

你可以从你的账户中下载所有已购买的Packt书籍中的代码，地址是：http://www.packtpub.com。无论你是在什么地方购买的这本书，都可以访问http://www.packtpub.com/support并注册，文件将直接以邮件方式发送给你。

使用**Docker**镜像

Docker镜像包含了我们的应用和相应辅助其运行的组件，例如操作系统、运行环境、开发库等。它们被下载并部署到Docker宿主机上，以Docker容器的形式运行应用。本节将涉及在使用Docker镜像时需用到的几个命令：

- docker build
- docker images
- docker push
- docker pull

本节中的相关材料在Docker文档网站上可以访问，网址是：https://docs.docker.com/userguide/dockerimages。

编译Docker镜像

我们将采用Docker Education小组的training/webapp项目的Dockerfile来创建一个Docker镜像。接下来的步骤将展示如何编译这个网络应用。

1．首先，我们将通过如下命令来克隆webapp的Git仓库（https://github.com/docker-training/webapp）：

```
dockerhost$ git clone https://github.com/docker-traini
training-webapp
Cloning into 'training-webapp'...
remote: Counting objects: 45, done.
remote: Total 45 (delta 0), reused 0 (de..., pack-reus
Unpacking objects: 100% (45/45), done.
Checking connectivity... done.
```

2．然后，在执行下面的命令之后，使用docker build来编译Docker镜像：

```
dockerhost$ cd training-webapp
dockerhost$ docker build -t hubuser/webapp .
Sending build context to Docker daemon 121.3 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
Repository ubuntu already being ... another client. Wa
```

```
---> 6d4946999d4f
Step 1 : MAINTAINER Docker Education Team <education@d
---> Running in 0fd24c915568
---> e835d0c77b04
Removing intermediate container 0fd24c915568
Step 2 : RUN apt-get update
---> Running in 45b654e66939
Ign http://archive.ubuntu.com trusty InRelease
...
Removing intermediate container c08be35b1529
Step 9 : CMD python app.py
---> Running in 48632c5fa300
---> 55850135bada
Removing intermediate container 48632c5fa300
Successfully built 55850135bada
```

-t标签用于给镜像做标签，标记为hubuser/webapp。用<username>/<imagename>给容器做标签对于推送Docker镜像是一个重要的考虑，这部分内容将在后续章节中介绍。关于docker build命令的更多细节可以查阅相关文档，网址是：https://docs.docker.com/reference/commandline/ build或者运行docke build--help命令。

3．最后，来确认一下镜像已经在我们的Docker宿主机中，使

用docker images命令即可：

```
dockerhost$ docker images
REPOSITORY      TAG     IMAGE ID    CREATED         VIRT
hubuser/webapp  latest  55850135    5 minutes ago   360
Ubuntu          14.04   6d494699    3 weeks ago     188.
```

推送**Docker**镜像到资源库

我们已经制作完Docker镜像，接下来将其推送到资源库中，用于共享和在其他Docker宿主机上部署。Docker的默认配置是将镜像推送到Docker Hub。Docker Hub是Docker公司管理的一个开放资源库，任何人都可以推送和共享他们的镜像到这个资源库里。将镜像推送到资源库的步骤如下所示。

1. 在推送到Docker Hub之前，需要用docker login命令来获得授权，命令如下：

```
dockerhost$ docker login
Username: hubuser
Password: ********
Email: hubuser@hubemail.com
WARNING: login credentials saved in /home/hubuser/.dock
Succeeded
```

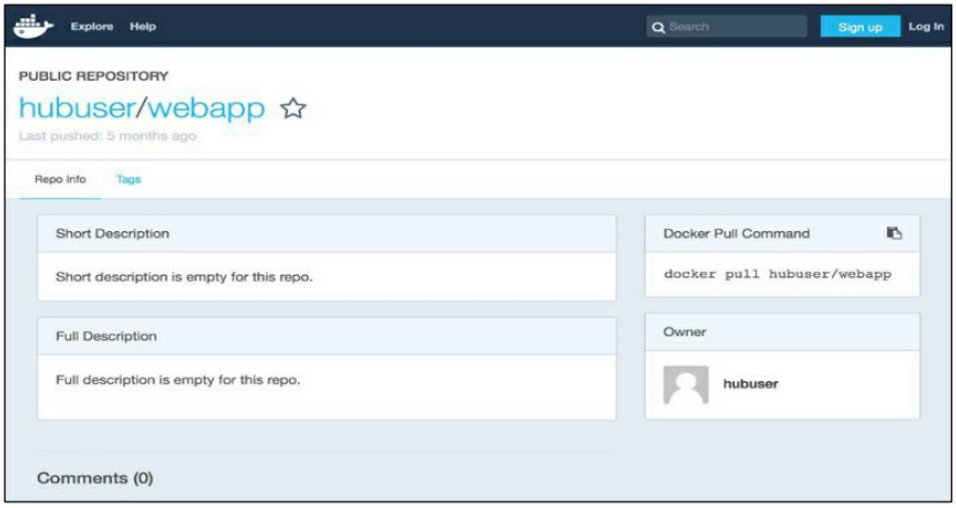
如果还没有Docker Hub的账号，可以参照说明去注册一个账

号，网址是：https://hub.docker.com/account/signup。

2．现在可以推送镜像到Docker Hub了。如前一节中所述，标签<username>/<imagename> 用于区分资源库中的镜像。通过docker push命令将镜像推送到Docker Hub中，执行情况如下：

```
dockerhost$ docker push hubuser/webapp
The push refers to a repository [hubuser/webapp] (len:
Sending image list
Pushing repository hubuser/webapp (1 tags)
428b411c28f0: Image already pushed, skipping
...
7d04572a66ec: Image successfully pushed
55850135bada: Image successfully pushed
latest: digest: sha256:b00a3d4e703b5f9571ad6a... size:
```

我们已经成功将Docker镜像推送到Docker Hub，它可以在Docker Hub中被搜索到。同样，我们可以在该镜像的Docker Hub的页面中获得更多关于它的信息。在本例中，该镜像的Docker Hub的网址是https://hub.docker.com/r/hubuser/webapp，如下图所示。



更多关于推送Docker镜像到资源库的信息可以通过执行docker push--help查看，或者参阅官方文档，网址是：https://docs.docker.com/reference/commandline/push。

尽管Docker Hub是一个存放Docker镜像的很好的地方，但很多情况下我们希望自己管理镜像资源库。例如，当我们希望节省拉取镜像到Docker宿主机的带宽时。另外一个原因可能是我们的Docker宿主机在数据中心内部，而防火墙阻断了其与互联网的连接。在第2章中，我们将进一步讨论如何运行自己的Docker registry，以便拥有一个私有的Docker镜像资源库。

从资源库中拉取**Docker**镜像

一旦Docker镜像编译完成并推送到资源库中，如Docker Hub，我们就可以拉取镜像到宿主机上。当在开发环境的Docker宿主机上编译

完Docker镜像，并希望在云端生产环境中的宿主机上部署时，这个工作流是非常有用的。这避免了在其他Docker宿主机上重新编译镜像。拉取镜像同样被用于从Docker Hub获取其他现有Docker镜像来构建我们自己的Docker镜像。于是，相对于之前做过的克隆Git资源库并在另外的Docker宿主机上重新编译，我们可以直接拉取它。拉取hubuser/webapp这个Docker镜像的操作如下所示。

1. 首先，清空现有Docker宿主机的镜像，以确保我们将从Docker Hub中拉取该镜像，操作如下：

```
dockerhost$ dockerhost rmi hubuser/webapp
```

2. 然后，我们将通过docker pull来下载这个镜像，操作如下：

```
dockerhost$ docker pull hubuser/webapp
latest: Pulling from hubuser/webapp
e9e06b06e14c: Pull complete
...
b37deb56df95: Pull complete
02a8815912ca: Already exists
Digest: sha256:06e9c1983bd6d5db5fba376ccd63bfa529e8d02f
Status: Downloaded newer image for hubuser/webapp:lates
```

3. 最后，确认已经成功下载到该镜像，操作如下：

```
dockerhost$ docker images
REPOSITORY      TAG     IMAGE ID  CREATED       VIRTUAL
ubuntu          14.04   6d494699  3 weeks ago   188.3 M
hubuser/webapp  latest  2a8815ca  7 weeks ago   348.8 M
```

关于拉取Docker镜像的更多细节，可以通过执行docker pull--help命令或者查阅官方文档，网址是：https://docs.docker.com/reference/commandline/pull。

运行Docker容器

由于已经拉取或者编译了Docker镜像，我们可以通过docker run命令运行并测试它们。本节将使用如下Docker命令来获得Docker宿主机上正在运行的容器的更多信息，并且它们将贯穿在接下来的几个章节中，命令如下：

- docker ps
- docker inspect

全部命令的更详细信息可以通过执行docker run--help命令或查阅官方文档，网址是：https://docs.docker.com/reference/commandline/run。

暴露容器端口

在training/webapp 例子中，它的Docker容器是作为一个Web服务器运行的。为了使它可以服务容器外的访问，Docker需要知道该应用的绑定端口。Docker将这种情况叫作暴露端口。本节将介绍在运行容器时如何暴露端口信息。

回到我们之前编译的training /webapp Docker镜像，该应用运行一个基于Python Flask的Web应用，它监听在5000端口，见webapp/app.py文件中的高亮显示部分:

```
import os
from flask import Flask
app = Flask(  name  )
@app.route('/')
def hello():
     provider = str(os.environ.get('PROVIDER', 'world'))
     return 'Hello '+provider+'!'
if __name_______== '__main__':
     # Bind to PORT if defined, otherwise default to 5000.
     port = int(os.environ.get('PORT', 5000))
     app.run(host='0.0.0.0', port=port)
```

相应的，通过Dockerfile中的EXPOSE指令，Docker镜像使得Docker宿主机了解到应用监听在5000端口，具体示例如下：

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get \
```

```
                install -y -q python-all python-pip
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/ WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```

既然我们已经掌握了Docker如何暴露容器的端口，那么接下来用如下命令来运行hubuser/webapp容器:

1. 运行带–d标签的docker　run命令来启动容器作为一个后台进程，操作如下：

```
dockerhost$ docker run --name ourapp -d hubuser/webapp
```

2. 通过docker　ps命令确认运行的容器暴露了5000端口给Docker宿主机，操作如下：

```
dockerhost:~/training-webapp$ docker ps
CONTAINER ID   IMAGE   ...  STATUS  PORTS  NAMES
df3e6b788fd8   hubuser...   Up 4 seconds   5000/tcp our
```

除了EXPOSE的说明，已暴露的端口可以在运行时通过--expose=[]标签进行覆盖。例如，使用如下命令可以让hubuser/webapp应用暴露端口4000-4500：

```
dockerhost$ docker run -d --expose=4000-4500 \
                          --name app hubuser/webapp
dockerhost $ docker ps
CONTAINER ID  IMAGE  ...                  PORTS
ca4dc1da26d   hubuser/webapp:latest  ...  4000-4500/tcp,50
```

```
df3e6b788fd8  hubuser/webapp:l...        5000/tcp
```

这个特殊的docker　run标志调试应用时十分有用。例如，假定我们的Web应用使用端口4000-4500，但是，我们并不希望这些端口在生产环境中都是可访问的，那么可以用--expose=[]临时启动一个用于调试的容器。更多关于如何使用这类技巧来解决Docker容器的问题，将在第7章中进行详细介绍。

发布容器端口

暴露端口只是使该端口在容器内可用，而对于那些服务Docker宿主机以外的应用，它们的端口则需要被发布出去。docker　run命令提供-P和-p标签来发布容器内暴露的端口。本节将介绍如何使用这两个标签发布端口到Docker宿主机。

--publish-all

-P或者--publish-all标签发布容器内所有已暴露的端口到Docker宿主机上的随机高位端口，具体端口范围定义在/proc/sys/net/ipv4/ip_local_port_range中。下面的步骤将继续使用hubuser/webapp的Docker镜像，并发布其暴露的端口。

1．首先，执行下面的命令运行容器，并发布所有已暴露的端口：

```
dockerhost$ docker run -P –d --name exposed hubuser/we
```

2．接下来，确认Docker宿主机发布端口32771将访问请求转向Docker容器暴露的5000端口。输入docker　ps命令来确认结果，操作如下：

```
dockerhost$ docker ps
CONTAINER  ID IMAGE   ...                PORTS
508cf1fb3e5   hubuser/webapp:latest ... 0.0.0.0:32771-
```

3. 同样可以验证分配的端口32771在Docker宿主机配置的临时端口范围之内：

```
dockerhost$ cat /proc/sys/net/ipv4/ip_local_port_range
32768   61000
```

4. 此外，可以通过如下命令来确认Docker宿主机确实监听在分配的32771端口上：

```
dockerhost$ ss -lt 'sport = *:32771'
State   Recv-Q Send-Q  Local Address:Port  Peer Addres
LISTEN  0  128         :::32771            :::*
```

5. 最后，可以验证Docker宿主机的32771端口确实被映射到运行的Docker容器中，方法是通过确认HTTP请求的返回信息是training/webapp这个Python应用的结果，操作如下：

```
$ curl  http://dockerhost:32771
Hello world!
```

--publish

-p或者--publish标签可将容器端口发布到Docker宿主机。如果容器的端口没有主动暴露，那么这个容器也可以被暴露端口。根据文档说明，-p标签可以采用如下格式发布容器端口：

- containerPort
- hostPort:containerPort
- ip::containerPort
- ip:hostPort:containerPort

通过指明hostPort，可以指定映射到Docker宿主机上的某个端口而不是随机分配一个临时端口。通过指明ip，可以限定从某个Docker宿主机的网络接口接收连接并返回相应数据包给映射的Docker容器端口。回到hubuser/webapp这个例子，通过如下命令将映射这个Python应用暴露的5000端口到Docker宿主机回环网络接口的80端口：

```
$ ssh dockerhost
dockerhost$ docker run -d -p 127.0.0.1:80:5000 training/we
dockerhost$ curl http://localhost
Hello world!
dockerhost$ exit
logout
Connection to dockerhost closed.
$ curl http://dockerhost
curl: (7) Failed connect to dockerhost:80; Connection refu
```

在调用docker run命令之后，Docker宿主机应用只能通过http://localhost响应HTTP请求。

链接容器

上一节介绍的端口发布功能只允许容器间通过Docker宿主机的端

口进行通信。另一种直接链接容器的方法是使用容器链接功能。容器链接在一起后，可以使源容器向目标容器发送消息，并且使通信中的容器以一种更安全的方式进行相互发现。

更多关于链接容器的内容可以在Docker文档网站上查阅，地址是：https://docs.docker.com/userguide/dockerlinks。

在本节中，我们将使用--link标签来安全地链接容器。接下来通过一个例子演示如何链接容器，具体步骤如下。

1. 首先，确认hubuser/webapp容器运行时仅暴露了指定端口。我们将创建一个容器，叫作source，作为源容器。创建源容器的操作如下：

```
dockerhost$ docker run --name source –d hubuser/webapp
```

2. 接下来，我们将创建一个目标容器。使用--link<source>:<alias>　创建一个链接，从source源容器指向webapp目标容器，操作如下：

```
dockerhost$ docker run -d --link source:webapp \
                    --name destination busybox /bin/ping
```

3. 最后，现在通过检查新创建的目标容器destination来确认这个链接已经被建立，操作如下：

```
dockerhost$ docker inspect -f "{{ .HostConfig.Links }}"
```

```
                                    destination
    [/source:/destination/webapp]
```

在链接过程中到底发生了什么呢？Docker宿主机在两个容器间创建了一个安全通道。我们可以通过Docker宿主机的iptable来确认，操作如下：

```
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddre
                               source
172.17.0.15
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddre
                               destination
172.17.0.28
dockerhost$ iptables -L DOCKER
Chain DOCKER (1 references)
 target  prot opt source  destination
 ACCEPT  tcp  --  172.17.0.28  172.17.0.15  tcp dpt:5000
 ACCEPT  tcp  --  172.17.0.15  172.17.0.28  tcp spt:5000
```

在前一个iptable中，Docker宿主机允许目标容器destination（172.17.0.28）访问源容器source（172.17.0.28）的5000端口的向外链接。第二个iptable的内容显示允许源容器source接收从destination容器向其5000端口发出的链接。

除了Docker宿主机在两个容器间建立安全链接，Docker宿主机通过如下两种形式暴露了源容器的信息给目标容器：

- 环境变量
- /etc/hosts中的条目

这两种信息源将在下一节中作为容器交互的例子进一步进行分析。

交互式容器

通过指定-i标签，可以使一个容器在前台运行，并接到标准输入流上。通过与-t标签联合使用，可以给容器添加一个虚拟的终端。利用这两者，我们可以使Docker容器像一个可交互的进程，类似于普通的shell窗口。这个特性在需要调试或者检查Docker容器内部时非常有用。接着上一节的例子，我们将调试当容器被链接到一起时它们内部发生了什么变化，步骤如下。

1. 首先，建立一个可交互的容器会话并链接到之前运行的source源容器，操作如下：

```
dockerhost$ docker run -i -t --link source:webapp \
                       --name interactive_container \
                       busybox /bin/sh
/ #
```

2. 接下来，我们检查暴露给目标容器的环境变量，操作如下：

```
/ # env | grep WEBAPP
WEBAPP_NAME=/interactive_container/webapp
WEBAPP_PORT_5000_TCP_ADDR=172.17.0.15
WEBAPP_PORT_5000_TCP_PORT=5000
WEBAPP_PORT_5000_TCP_PROTO=tcp
WEBAPP_PORT_5000_TCP=tcp://172.17.0.15:5000
WEBAPP_PORT=tcp://172.17.0.15:5000
```

一般来说，相互链接的容器间会创建如下环境变量。

- 对每个容器均有：<alias>_NAME=/container_name/alias_name
- 对每个已暴露端口的URL都有：<alias>_PORT_<port>_<protocol>。同时，它作为唯一前缀被添加到如下环境变量中。
  - <prefix>_ADDR：源容器的IP地址
  - <prefix>_PORT：已暴露的端口
  - <prefix>_PROTO：已暴露的端口采用的协议，TCP或UDP
- <alias>_PORT：源容器暴露的第一个端口

3．然后，在相互链接的容器中，目标容器的发现特性是/etc/hosts文件。webapp容器的别名被映射到源容器的IP地址同时，源容器的名字也被映射到相同的IP地址。下面的代码片段来自于可交互容器会话中的/etc/hosts文件，它包含如下映射：

```
172.17.0.29  d4509e3da954
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
172.17.0.15  webapp 85173b8686fc source
```

4. 最后，我们使用别名来链接源容器。在接下来的例子中，将通过创建一个HTTP请求到源容器的别名webapp，实现访问源容器中运行的Web应用，操作如下：

```
/ # nc webapp 5000
GET /

Hello world!
/ #
```

配合docker commit命令，容器交互同样可以用于创建容器。但是，这是一个枯燥的过程并且这种开发过程的规模局限于单个开发者。因此，请使用docker build命令，并用版本控制工具来管理Dockerfile。

小结

希望通过本章可让大家熟悉本书中将使用的大部分命令。首先准

备一个可以进行Docker容器交互的Docker宿主机。然后，编译、下载、上传各式各样的Docker镜像，用于开发和部署容器到开发环境和生产环境。最后，通过编译或者下载Docker镜像来运行Docker容器。此外，我们还介绍了如何与运行中的容器进行通信和交互操作。

在下一章中，你将学会如何优化Docker镜像。那么，继续努力吧！

2　优化Docker镜像

既然我们已经编译并部署了Docker容器，那么就可以开始享受它带来的便利了。统一标准的包格式方便开发者和系统管理员相互配合，从而简化了管理应用代码的工作。Docker的容器格式使我们可以快速迭代应用程序的版本并与其他人共享。开发、测试和部署时间得到了降低，这归功于Docker容器的轻量和启动速度。Docker容器的移植能力使我们可以将应用程序从物理服务器上拓展到云主机上。

但是，我们将发现它逐渐背离我们最初使用Docker的原因。由于经常需要下载应用程序对应的最新Docker镜像运行库，使得开发时间在逐渐增加。而Docker　Hub的速度拖慢了部署的时间。更糟的是，Docker Hub有可能宕机，而我们就无法部署了。Docker镜像的尺寸已经达到GB级别了，对于如此大的镜像，一次简单的更新可能就要耗费一天。

本章将讨论那些Docker容器尺寸超出控制范围的场景，并针对这些问题给出了相应的解决方法，具体包括如下内容：

- 降低镜像部署时间
- 降低镜像编译时间
- 减小镜像的尺寸

降低部署时间

随着时间的推移，我们构建的Docker容器的尺寸变得越来越大。在现有Docker宿主机中更新运行的容器是没有问题的：Docker会合理利用Docker镜像层，这些镜像层是随着我们应用的增长而创建的。但是，考虑下我们准备水平扩展应用的情况：这意味着要部署更多的Docker容器到额外的Docker宿主机。每一个新的Docker宿主机需要下载我们创建的所有镜像层。本节将介绍一个“大”Docker应用是如何影响在新Docker宿主机上的部署时间的。首先，我们来构建这个Docker应用问题场景，步骤如下。

1. 编写Dockerfile创建一个“大”Docker镜像：

```
FROM debian:jessie
RUN dd if=/dev/urandom of=/largefile bs=1024 count=5242
```

2. 接下来，编译这个Dockerfile，并附上标签hubuser/largeapp，操作如下：

```
dockerhost$ docker build -t hubuser/largeapp.
```

3. 检查这个Docker镜像的尺寸。从下面的输出中我们可以看到，它占用662MB空间，操作如下：

```
dockerhost$ docker images
REPOSITORY          TAG     IMAGE ID  CREATED         V
hubuser/largeapp    latest  450e3123  5 minutes ago   6
debian              Jessie  9a61b6b1  4 days ago      1
```

4. 通过time命令记录它上传到Docker Hub和从Docker Hub拉取

的时间，操作如下：

```
dockerhost$ time docker push hubuser/largeapp
The push refers to a repository [hubuser/largeapp] (len
450e319e42c3: Image already exists
9a61b6b1315e: Image successfully pushed
902b87aaaec9: Image successfully pushed
Digest: sha256:18ef52e36996dd583f923673618483a4466aa2d1
9f0...

real 11m34.133s
user  0m0.164s
sys   0m0.104s
dockerhost$ time docker pull hubuser/largeapp
latest: Pulling from hubuser/largeapp
902b87aaaec9: Pull complete
9a61b6b1315e: Pull complete
450e319e42c3: Already exists
Digest: sha256:18ef52e36996dd583f923673618483a4466aa2d1
9f0...
Status: Downloaded newer image for hubuser/largeapp:lat

real 2m56.805s
user 0m0.204s
sys  0m0.188s
```

从上面高亮显示的时间值上可以看出，当执行docker push命令时，上传镜像到Docker Hub花费了大量时间。在部署过程中，docker pull拉取我们新建的镜像到生产环境的Docker宿主机同样花费了很长

的时间。而上传和下载的时间长短依赖于Docker宿主机与Docker Hub之间的网络。如果Docker Hub停机，我们就不能根据需求部署新的Docker容器，也不能部署现有容器到额外的Docker宿主机。

为了利用Docker的快速分发特性和便捷部署能力，上传和下载镜像的稳定性是十分重要的。幸运的是，我们可以运行自己私有的Docker registry用于存储和分发Docker镜像，而不再依赖公共的Docker Hub服务。接下来介绍如何创建私有Docker registry，并确认它在性能方面的提升，操作如下所示。

1. 通过下面的命令来运行私有Docker registry。它将运行在tcp://dockerhost:5000:

```
dockerhost$ docker run -p 5000:5000 -d registry:2
```

2. 接下来，让我们确认Docker镜像的部署速度已经被提升。首先，对先前创建的镜像加个标签，用于将它推送到本地Docker registry，操作如下：

```
dockerhost$ docker tag hubuser/largeapp \
                       dockerhost:5000/largeapp
```

3. 观察推送同样的Docker镜像到我们新建的Docker registry的速度有多快。测试显示，现在推送Docker镜像的速度至少是之前速度的10倍：

```
dockerhost$ time docker push dockerhost:5000/largeapp
The push refers to a ...[dockerhost:5000/largeapp] (len
...
real    0m52.928s
user    0m0.084s
```

```
sys     0m0.048s
```

4. 首先确保已经移除了之前编译的镜像，然后再测试从本地Docker registry拉取Docker镜像的性能。测试显示，拉取Docker镜像的速度是之前拉取速度的30倍：

```
dockerhost$ docker rmi dockerhost:5000/largeapp \
                       hubuser/largeapp
Untagged: dockerhost:5000/largeapp:latest
Untagged: hubuser/largeapp:latestDeleted:
549d099c0edaef424edb6cfca8f16f5609b066ba744638990daf3b4
dockerhost$ time docker pull dockerhost:5000/largeapp
latest: Pulling from dockerhost:5000/largeapp
549d099c0eda: Already exists
902b87aaaec9: Already exists
9a61b6b1315e: Already exists
Digest: sha256:323bed623625b3647a6c678ee6840be23616edc3
c68b62dd52ecf
Status: Downloaded newer image for dockerhost:5000/larg
real    0m10.444s
user    0m0.160s
sys     0m0.056s
```

性能提升如此之大的原因在于我们通过本地局域网络上传和拉取镜像，它节省了Docker宿主机的带宽，使得部署时间变短。最关键的是，我们的部署不再需要依赖Docker Hub。

为了部署Docker镜像到其他的Docker宿主机，我们需要创建安全的Docker registry。关于创建它的详细内容已经超出了本书的讨论范围，更多关于创建Docker registry的细节可以在官方网站查阅，地址是：
https://docs.docker.com/registry/deploying。

改善镜像编译时间

Docker镜像是开发者工作时生成的主要构件。简化Docker文件和加速容器技术使得我们可以快速迭代应用开发。但是，一旦编译Docker镜像所需时间不受控制，那么使用Docker带来的好处就将逐渐消失。本节我们将讨论某些花费大量时间构建Docker镜像的案例。然后，我们将给出几种技巧来解决这些问题。

采用**registry**镜像

构建镜像的时间有一大部分花费在获取上游镜像的过程。假定我们有如下所述的Dockerfile：

```
FROM java:8u45-jre
```

它将下载和编译java:8u45-jre这个镜像。当使用另外一个Docker宿主机，或者java:8u45-jre这个镜像在Docker Hub上更新了，我们的编译时间将显著增加。配置一个本地的registry镜像将降低这类镜像的编译时间。当每个开发者都有自己的Docker宿主机时，这将是一个非常有用的组织管理配置方案。该组织的网络将仅从Docker Hub下载此镜像一次，而该组织内的其他Docker宿主机可以直接从本地的registry

镜像中拉取镜像。

搭建一个registry镜像的过程跟前一节介绍的创建一个本地registry一样简单。但是，我们需要通过传递--registry-mirror选项给Docker守护进程，用以配置Docker宿主机知道本地的registry镜像。步骤如下所示。

1. 在装有Debian Jessie系统的Docker宿主机中，通过更新和创建一个Systemd插件文件来配置Docker守护进程，文件位置在：/etc/systemd/system/docker.service.d/10-syslog.conf，在该文件中加入如下内容：

```
service.d/10-syslog.conf，在该文件中加入如下内容：
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon-H fd:// \
                              --registry-mirror=http://doc
```

2. 然后，我们将重启Systemd来加载docker.service的新配置文件，操作如下：

```
dockerhost$ systemctl daemon-reload
```

3. 通过重启新配置的Systemd单元来重启Docker守护进程，操作如下：

```
dockerhost$ systemctl restartdocker.service
```

4. 最后，运行作为镜像的registry的Docker容器，操作如下：

```
dockerhost$ docker run -p 5000:5000 -d \
            -e STANDALONE=false \
```

```
-e MIRROR_SOURCE=https://registry-1.docker.
-e MIRROR_SOURCE_INDEX=https://index.docker
registry
```

接下来，确认该镜像registry已经正常工作，操作步骤如下所示。

1．编译本节开头描述的Dockerfile，并留意它的编译时间。可以发现，编译该镜像的大部分时间都用于下载其上游Docker镜像java:8u45-jre，操作如下：

```
dockerhost$ time docker build -t hubuser/mirrorups
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM java:8u45-jre
Pulling repository java
4ac125456dd3: Download complete
902b87aaaec9: Download complete
9a61b6b1315e: Download complete
1ff9f26f09fb: Download complete
6f6bffbbf095: Download complete
4b61c52d7fe4: Download complete
1a9b1e5c4dd5: Download complete
2e8cff440182: Download complete
46bc3bbea0ec: Download complete
3948efdeee11: Download complete
918f0691336e: Download complete
Status: Downloaded newer image for java:8u45-jre
---> 4ac125456dd3
Successfully built 4ac125456dd3
```

```
real  1m58.095s
user      0m0.036s
sys       0m0.028s
```

2. 然后，移除该镜像以及它的上游依赖镜像，重新编译该镜像，步骤如下：

```
dockerhost$ docker rmi java:8u45-jre hubuser/mirrorupst
dockerhost$ time docker build -t hubuser/mirrorupstream
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM java:8u45-jre
Pulling repository java
4ac125456dd3: Download complete
902b87aaaec9: Download complete
9a61b6b1315e: Download complete
1ff9f26f09fb: Download complete
6f6bffbbf095: Download complete
4b61c52d7fe4: Download complete
1a9b1e5c4dd5: Download complete
2e8cff440182: Download complete
46bc3bbea0ec: Download complete
3948efdeee11: Download complete
918f0691336e: Download complete
Status: Downloaded newer image for java:8u45-jre
---> 4ac125456dd3
Successfully built 4ac125456dd3

real    0m59.260s
```

```
user    0m0.032s
sys     0m0.028s
```

当java:8u45-jre这个Docker镜像第二次被下载时，它从本地的registry镜像中检索，而不是从Docker Hub上下载。搭建Docker registry镜像可以提升下载上游镜像的速度，通常速度可以提升两倍。如果将另外的Docker宿主机指向该registry镜像，它也会做同样的处理：忽略从Docker Hub的下载。

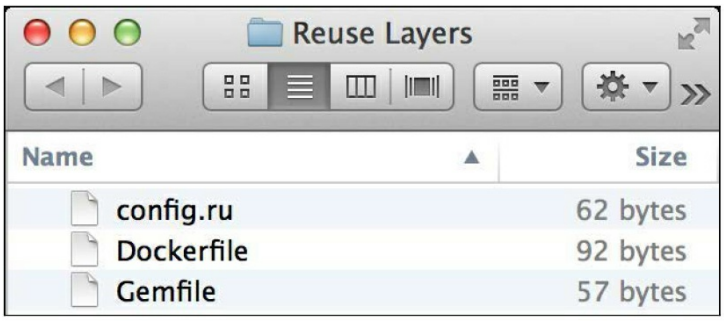
本教程中关于如何创建registry镜像的部分参考了Docker官方文档网站。更多细节内容可以在网站查阅，地址是：https://docs.docker.com/articles/registry_mirror。

复用镜像层

众所周知，一个Docker镜像是由一系列的层合并而成的，它们使用的是一个单一镜像联合文件系统（union filesystem）。当我们在构建Docker镜像时，Docker会检查Dockerfile中正在处理的指令，判断在它的缓存中是否已经存在可以复用的镜像，而不是重复创建一个镜像。通过学习构建过程中缓存的工作方式，可以提高后续创建Docker镜像的速度。在开发应用程序的过程中，我们对依赖库的操作方式跟这个过程很类似，并不是每次都需要添加应用程序的依赖库。在大多数情况下，我们只是更新应用程序的核心部分。了解这些之后，可以在开发工作流中围绕它设计一种创建Docker镜像的工作方式。

更多关于Dockerfile指令如何被缓存的内容可以从官方文档网站中查阅，网址是：http://docs.docker.com/articles/dockerfile_best-practices/#build-cache。

例如，假定我们正在编写一个Ruby应用程序，它的源代码结构如下图所示。



其中config.ru文件的内容如下所示：

```
app =  proc do |env|
  [200, {}, %w(hello world)]
end
run app
```

Gemfile文件的内容如下所示：

```
source 'https://rubygems.org'

gem 'rack'
gem 'nokogiri'
```

而Dockerfile 文件的内容如下所示：

```
FROM ruby:2.2.2

ADD . /app
WORKDIR /app
RUN bundle install

EXPOSE 9292
CMD rackup -E none
```

接下来的步骤将展示如何创建该Ruby应用程序的Docker镜像，操作如下所示。

1. 首先，通过下面的命令来创建Docker镜像。记录显示创建镜像的时间为1分钟左右：

```
dockerhost$ time docker build -t slowdependencies
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ruby:2.2.2
---> d763add83c94
Step 1 : ADD . /app
---> 6663d8b8b5d4
Removing intermediate container 2fda8dc40966
Step 2 : WORKDIR /app
---> Running in f2bec0dea1c9
---> 289108c6655f
Removing intermediate container f2bec0dea1c9
Step 3 : RUN bundle install
```

```
---> Running in 7025de40c01d
Don't run Bundler as root. Bundler can ask for sud
Fetching gem metadata from https://rubygems.org/..
Fetching version metadata from https://rubygems.or
 Resolving dependencies...
Installing mini_portile 0.6.2
Installing nokogiri 1.6.6.2 with native extensions
Installing rack 1.6.4
Using bundler 1.10.5
Bundle complete! 2 Gemfile dependencies, 4 gems no
Bundled gems are installed into /usr/local/bundle.
---> ab26818ccd85
Removing intermediate container 7025de40c01d
Step 4 : EXPOSE 9292
---> Running in e4d7647e978b
---> a602159cb786
Removing intermediate container e4d7647e978b
Step 5 : CMD rackup -E none
---> Running in 407308682d13
---> bffce44702f8
Removing intermediate container 407308682d13
Successfully built bffce44702f8

real    0m54.428s
user    0m0.004s
sys     0m0.008s
```

2．接下来，更新config.ru文件来改变应用程序的功能，操作如下：

```
app =  proc do |env|
  [200, {}, %w(hello other world)]
end
run app
```

3. 让我们重新创建这个Docker镜像，同时注意创建过程花费的时间，操作如下：

```
dockerhost$ time docker build -t slowdependencies .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ruby:2.2.2
---> d763add83c94
Step 1 : ADD . /app
---> 05234a367589
Removing intermediate container e9d33db67914
Step 2 : WORKDIR /app
---> Running in 65b3f40d6228
---> c656079a833f
Removing intermediate container 65b3f40d6228
Step 3 : RUN bundle install
---> Running in c84bd4aa70a0
Don't run Bundler as root. Bundler can ask for sudo ...
Fetching gem metadata from https://rubygems.org/.......
Fetching version metadata from https://rubygems.org/..
Resolving dependencies...
Installing mini_portile 0.6.2
Installing nokogiri 1.6.6.2 with native extensions
Installing rack 1.6.4
```

```
Using bundler 1.10.5
Bundle complete! 2 Gemfile dep..., 4 gems now installed
Bundled gems are installed into /usr/local/bundle.
---> 68f5dc363171
Removing intermediate container c84bd4aa70a0
Step 4 : EXPOSE 9292
---> Running in 68c1462c2018
---> c257c74eb7a8
Removing intermediate container 68c1462c2018
Step 5 : CMD rackup -E none
---> Running in 7e13fd0c26f0
---> e31f97d2d96a
Removing intermediate container 7e13fd0c26f0
Successfully built e31f97d2d96a

real    0m57.468s
user    0m0.008s
sys     0m0.004s
```

可以发现，尽管只有一行代码改变，但是还是需要在创建过程中为Docker镜像的每一次迭代执行bundle install命令。这样效率很低，而且它打断了我们的开发状态，因为需要占用1分钟的时间去编译并运行Docker应用。对于没有耐心的开发者来说，这简直不能忍受！

为了优化这个工作流，我们可以将准备应用程序依赖的阶段从整个应用程序的镜像构建中剥离出来，步骤如下所示。

1．首先，变更Dockerfile内容，操作如下：

```
FROM ruby:2.2.2
```

```
ADD Gemfile /app/Gemfile
WORKDIR /app
RUN bundle install
ADD . /app

EXPOSE 9292
CMD rackup -E none
```

2．然后，编译刚改造过的Docker镜像，操作如下：

```
dockerhost$ time docker build -t separatedependencies .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
...
Step 3 : RUN bundle install
---> Running in b4cbc6803947
Don't run Bundler as root. Bundler can ask for sudo if needed, and
installing your bundle as root will break this applicat non-root
users on this machine.
Fetching gem metadata from https://rubygems.org/.......
Fetching version metadata from https://rubygems.org/..
Resolving dependencies...
Installing mini_portile 0.6.2
Installing nokogiri 1.6.6.2 with native extensions
Installing rack 1.6.4
Using bundler 1.10.5
```

```
Bundle complete! 2 Gemfile dependencies, 4 gems now ins
gems are installed into /usr/local/bundle.
---> 5c009ed03934
Removing intermediate container b4cbc6803947
Step 4 : ADD . /app
...
Successfully built ff2d4efd233f

real   0m57.908s
user   0m0.008s
sys    0m0.004s
```

3. 编译时间与之前相同，同时记录下Step3中创建的镜像ID。然后，重新修改config.ru文件并重新编译整个镜像，操作如下：

```
dockerhost$ vi config.ru # edit as we please
dockerhost$ time docker build -t separatedependencies .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ruby:2.2.2
 ---> d763add83c94
Step 1 : ADD Gemfile /app/Gemfile
 ---> Using cache
 ---> a7f68475cf92
Step 2 : WORKDIR /app
 ---> Using cache
 ---> 203b5b800611
Step 3 : RUN bundle install
```

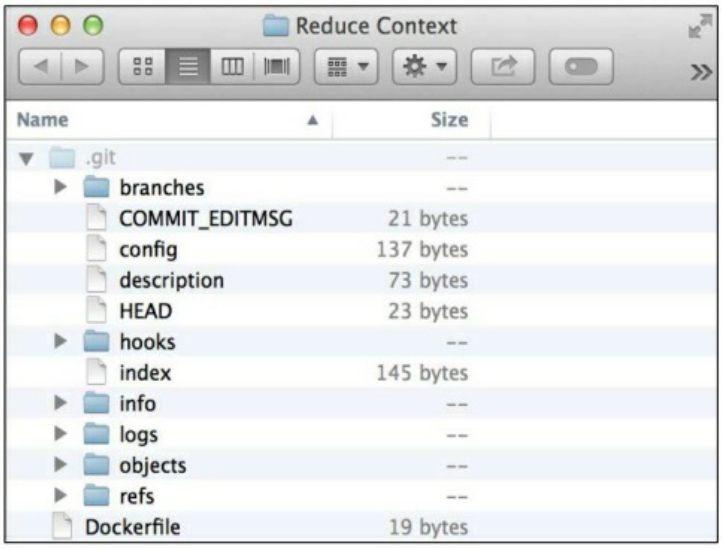
```
 ---> Using cache
 ---> 5c009ed03934
Step 4 : ADD . /app
 ---> 30b2bfc3f313
Removing intermediate container cd643f871828
Step 5 : EXPOSE 9292
 ---> Running in a56bfd37f721
 ---> 553ae65c061c
Removing intermediate container a56bfd37f721
Step 6 : CMD rackup -E none
 ---> Running in 0ceaa70bee6c
 ---> 762b7ccf7860
Removing intermediate container 0ceaa70bee6c...
Successfully built 762b7ccf7860

real   0m0.734s
user   0m0.008s
sys    0m0.000s
```

从输出的记录中可以看到，docker build复用了Step3中的缓存，这是因为对于Gemfile并没有改动。同时，Docker镜像的编译时间降低了约80倍。

这类重构Docker镜像的方法在降低部署时间方面是非常有效的。由于生产环境中的Docker宿主机已经使用Step3之前的镜像层，因此新版本的Docker应用只需Docker宿主机拉取Step4到Step6的镜像层来更新应用程序即可。

减小构建上下文大小

假定我们在基于Git的版本控制管理中有一个Dockerfile文件，如下图所示。



某种程度上，我们会发现．git文件夹占用了太多硬盘空间。这可能是我们提交了过多次代码之后的结果，查看方法如下：

```
dockerhost$ du -hsc .git
1001M  .git
1001M  total
```

当编译Docker应用时，我们发现编译Docker镜像的时间也非常久，查看方法如下：

```
dockerhost$ time docker build -t hubuser/largecontext .
Sending build context to Docker daemon 1.049 GB
Sending build context to Docker daemon
...
Successfully built 9a61b6b1315e

real   0m17.342s
user   0m0.408s
```

```
sys    0m1.360s
```

在仔细查看输出记录时，我们会发现，Docker客户端上传了整个．git文件夹（约1GB），而仅仅是因为它在镜像的编译路径下。于是，在编译Docker镜像的过程中，Docker守护进程花费了大量时间来接收这部分内容。

但是，这些文件对于编译该应用程序的Docker镜像是无用的。而且，这些Git相关的文件对于在生产环境中运行应用程序也是无用的。我们可以在编译Docker镜像之前设定Docker去忽略一些文件，操作方法如下。

1．在Dockerfile所在目录下创建一个．dockerignore文件，并将如下内容写入该文件：

```
.git
```

2．然后，重新编译Docker镜像，操作如下：

```
dockerhost$ time docker build -t hubuser/largecontext .
Sending build context to Docker daemon 3.072 kB
...
Successfully built 9a61b6b1315e

real   0m0.030s
user   0m0.004s
sys    0m0.004s
```

现在，编译时间提升了约500倍，同时减小了编译内容的大小。

更多关于如何使用.dockerignore文件的方法可以在官方文档网站查阅，网址是：https://docs.docker.com/reference/builder/#dockerignore-file。

使用缓存代理

另一个导致编译Docker镜像缓慢的原因是那些下载系统依赖库的指令。例如，一个基于Debian系统的Docker镜像需要从APT资源库中下载依赖包。编译镜像过程中，apt-get install指令运行时间的长短取决于所需要下载的依赖包的大小。降低这类编译指令消耗时间的技巧是引入这些依赖包的缓存代理。比较流行的缓存代理是apt-cacher-ng工具。本节将对其进行介绍并搭建它，以便提高Docker镜像编译工作流的效率。

在下面的Dockerfile例子中，镜像里安装了一系列Debian依赖包，内容如下：

```
FROM debian:jessie

RUN echo deb http://httpredir.debian.org/debian \
        jessie-backports main > \
        /etc/apt/sources.list.d/jessie-backports.list
RUN apt-get update &&\
    apt-get --no-install-recommends \
```

```
install -y openjdk-8-jre-headless
```

运行记录显示，它的编译时间非常长，这是因为Dockerfile下载了大量与Java (openjdk-8-jre-headless)相关的依赖包，操作如下：

```
dockerhost$ time docker build -t beforecaching .
...
Successfully built 476f2ebd35f6

real    3m22.949s
user    0m0.048s
sys     0m0.020s
```

为了提高编译该Docker镜像的速度，我们将搭建apt-cacher-ng缓存代理。幸运的是，在Docker Hub上有一个可以直接使用的镜像。准备apt-cacher-ng的步骤如下所示。

1. 在Docker宿主机中启动apt-cacher-ng，操作如下：

```
dockerhost$ docker run -d -p 3142:3142 sameersbn/apt-ca
```

2. 接下来，将使用该缓存代理修改Dockerfile，内容如下：

```
FROM debian:jessie

RUN echo Acquire::http { \
        Proxy\"http://dockerhost:3142\"\; \
        }\;>/etc/apt/apt.conf.d/01proxy
```

3. 编译之前创建并标记为hubuser/debian:jessie的Dockerfile，操作如下：

```
dockerhost$ docker build -t hubuser/debian:jessie
```

4. 最后，更新hubuser/debian:jessie为新的Docker基础镜像，其中包含了需要安装的一系列Debian依赖包，内容如下：

```
FROM hubuser/debian:jessie

RUN echo deb http://httpredir.debian.org/debian \
        jessie-backports main > \
        /etc/apt/sources.list.d/jessie-backports.list
RUN apt-get update && \
    apt-get --no-install-recommends \
            install -y openjdk-8-jre-headless
```

5. 确认新的工作流，执行初始化编译来使缓存生效，操作如下：

```
dockerhost$ docker build -t aftercaching .
```

6. 最后，移除刚才编译创建的镜像，并重新编译该镜像验证缓存生效，操作如下：

```
dockerhost$ docker rmi aftercaching
dockerhost$ time docker build -t aftercaching .
...
Removing intermediate container 461637e26e05
Successfully built 2b80ca0d16fd

real    0m31.049s
user    0m0.044s
sys     0m0.024s
```

尽管我们没有使用Docker的编译缓存，但记录显示第二次的编译速度还是很快。当我们为团队或者公司开发Docker基础镜像时，这个技巧很有用。团队成员在重新编译镜像时将获得6.5倍速度的提升，因为他们所需的依赖包将通过公司内部的缓存代理下载。在持续整合服务器上的编译同样将被提速，因为在开发过程中我们已经让缓存生效了。

本节讨论了如何使用一个具体的缓存代理服务，还有其他可供选择的缓存代理以及相应的说明文档，列表如下所示。

- **apt-cacher-ng**：它支持缓存Debian、RPM和其他特定系统包，网址为：https://www.unix-ag.uni-kl.de/~bloch/acng。
- **Sonatype Nexus**：它支持Maven、Ruby Gems、PyPI和NuGet的依赖包缓存，网址为：http://www.sonatype.org/nexus。
- **Polipo**：它是一个开发过程中通用的缓存代理，网址为：http://www.pps.univ-paris-diderot.fr/~jch/software/polipo。
- **Squid**：它是另一个流行的缓存代理，同样可以处理其他几种网络流量场景，网址为：http://www.squid-cache.org。

减小Docker镜像的尺寸

随着我们对Docker应用的持续使用，如果不加注意，那么镜像的尺寸就会变得越来越大。很多人在使用Docker时会发现，团队定制化的Docker镜像尺寸都至少有1GB大。镜像越大就意味着编译和部署Docker应用的时间会越长。因此，我们需要减小需要部署的镜像的尺

寸。它会抵消使用Docker带来的好处，失去快速迭代开发和部署应用的能力。

本节将深入讨论Docker镜像层的技术细节以及它们是如何影响最终镜像的大小的。接下来，我们将在研究Docker镜像工作原理的过程中，学习如何优化这些镜像层。

链式指令

Docker镜像尺寸变大的一个原因是很多对编译或运行无关的指令被引入到镜像中。一个常见的案例是打包元数据和缓存。在安装完编译和运行相关的依赖包之后，这些下载的文件就没有存在的必要了。类似*clean*的指令可以在很多仓库（如Docker Hub）的Dockerfile中发现，它们用于清理这类文件，例如：

```
FROM debian:jessie

RUN echo deb http://httpredir.debian.org/debian \
jessie-backports main \
> /etc/apt/sources.list.d/jessie-backports.list
RUN apt-get update
RUN apt-get --no-install-recommends \
install -y openjdk-8-jre-headless
RUN rm -rfv /var/lib/apt/lists/*
```

但是，一个Docker镜像的尺寸是每一个独立镜像层的尺寸之和，这也就是联合文件系统的工作机制。因此，*clean*步骤并没有真正删掉相应的硬盘空间，可通过如下命令来查看：

```
dockerhost$ docker build -t fakeclean .
```

```
dockerhost$ docker history fakeclean

IMAGE         CREATED         CREATED BY
33c8eedfc24a  2  minutes ago  /bin/sh  -c rm -rfv /var/lib
48b87c35b369  2  minutes ago  /bin/sh  -c apt-get install
dad9efad9e2d  4  minutes ago  /bin/sh  -c apt-get update
a8f7bf731a7d  5  minutes ago  /bin/sh  -c echo 'deb http:/
9a61b6b1315e  6  days ago     /bin/sh  -c #(nop) CMD ["/bi
902b87aaaec9  6  days ago     /bin/sh  -c #(nop) ADD file:
```

记录显示，这里并不存在“负”的镜像层尺寸。于是，Dockerfile中每一个指令要么保持镜像尺寸不变，要么增加它的尺寸。同时，每一步还会引入新的元数据信息，使得整体尺寸在增大。

为了降低整个镜像的尺寸，清除操作应该在同一镜像层中执行。于是，解决方案是将先前的多条指令合并成一条。当Docker使用/bin/sh来执行每一条指令时，我们可以使用Bourne shell提供的&&操作符来实现链接，例如：

```
FROM debian:jessie

RUN echo deb http://httpredir.debian.org/debian \
         jessie-backports main \
         > /etc/apt/sources.list.d/jessie-backports.lis
RUN apt-get update && \
apt-get --no-install-recommends \
        install -y openjdk-8-jre-headless && \
rm -rfv /var/lib/apt/lists/*
```

现在每一个独立层的尺寸已经足够小了。由于独立镜像层的尺寸

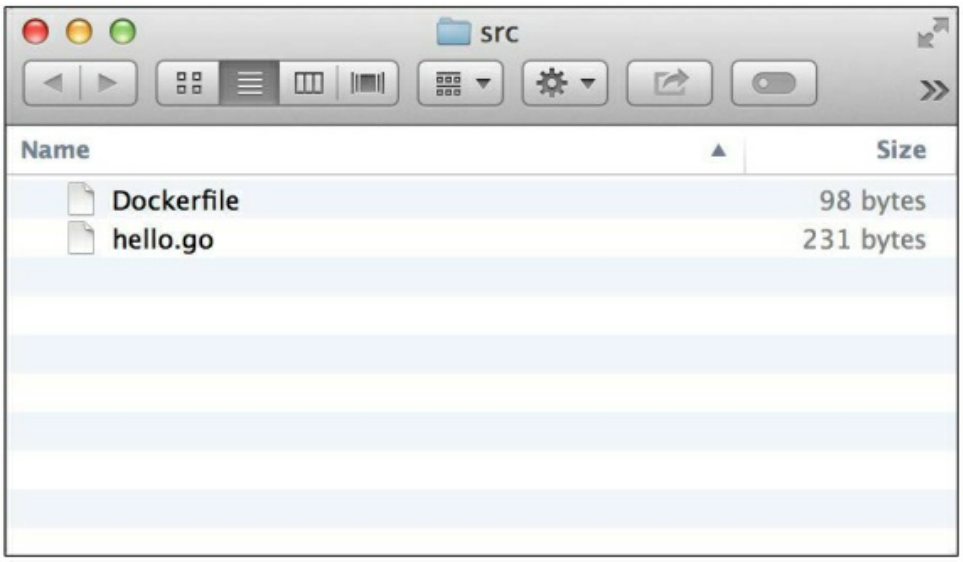
被减小，于是整个镜像的尺寸也随之减小。让我们来确认一下它们的尺寸，操作如下：

```
dockerhost$ docker build -t trueclean .
dockerhost$ docker history trueclean
IMAGE         CREATED             CREATED BY
 03d0b15bad7f  About a minute ago /bin/sh  -c apt-get upda
 a8f7bf731a7d  9  minutes ago     /bin/sh  -c echo deb h..
 9a61b6b1315e  6  days ago        /bin/sh  -c #(nop) CMD..
 902b87aaaec9  6  days ago        /bin/sh  -c #(nop) ADD..
```

分离编译镜像和部署镜像

Docker镜像中另一类无用文件是编译过程中的依赖文件，例如在编译应用程序过程中所依赖的源代码库，如编译文件和头文件。一旦应用程序编译完毕，这些文件就不再有用，因为运行该应用仅需要相关的依赖库。

例如，编译下面这个应用程序，它已经开发完毕并准备部署到Docker云主机上。这是一个简单的Web应用程序，采用Go语言开发，代码树如下图所示。



hello.go的内容如下：

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
     fmt.Fprintf(w, "hello world")
}
func main() {
     http.HandleFunc("/", handler)
     http.ListenAndServe(":8080", nil)
}
```

相应的Dockerfile中记录了如何编译源代码和运行编译结果，内容如下：

```
FROM golang:1.4.2

ADD hello.go hello.go
RUN go build hello.go
EXPOSE 8080
ENTRYPOINT ["./hello"]
```

接下来，我们将展示这个Docker镜像的尺寸是如何变大的，操作如下所示。

1．首先，编译这个Docker镜像并记录它的尺寸，操作如下：

```
dockerhost$ docker  bild -t largeapp .
dockerhost$ docker  images
REPOSITORY  TAG     IMAGE ID      CREATED       VIRTUAL SI
largeapp    latest  47a64e67fb81  4 minute...   523.1 MB
golang      1.4.2   124e2127157f  5 days ago    517.3 MB
```

2．然后，对比运行时实际应用程序的尺寸，操作如下：

```
dockerhost$ docker run --name large -d largeapp
dockerhost$ docker exec -it large/bin/ls -lh
total 5.6M
drwxrwxrwx 2 root root 4.0K Jul 14 06:26 bin
-rwxr-xr-x 1 root root 5.6M Jul 20 02:40 hello
-rw-r--r-- 1 root root  231 Jul 18 05:59 hello.go
drwxrwxrwx 2 root root 4.0K Jul 14 06:26 src
```

用Go语言编写应用程序以及编译代码的一个优势是，它可以生成一个单一可执行文件，这对部署非常方便。Docker镜像中除去该可执行文件占据的空间，全都是Docker基础镜像引入的无用文件。可以

发现，来自基础镜像的文件使得整个镜像尺寸增加了将近100倍。

同样，我们可以优化这个最终的Docker镜像并仅打包最后的hello可执行文件和相关的依赖包，然后部署到生产环境。优化步骤如下所示。

1. 首先，复制运行容器中的可执行文件到Docker宿主机，操作如下：

```
dockerhost$ docker cp -L large:/go/hello ../build
```

2. 如果前面的依赖库是一个静态库，那这一步就已经完成了，直接进入下一步。但是，Go工具编译时默认采用共享库机制，为了让二进制文件直接运行，还需要这些共享库，操作如下：

```
dockerhost$ docker exec -it large /usr/bin/ldd hello
linux-vdso.so.1 (0x00007ffd84747000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.
(0x00007f32f3793000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f32f33ea000)
/lib64/ld-linux-x86-64.so.2 (0x00007f32f39b0000)
```

3. 接下来，保存全部共享库到Docker宿主机，采用docker cp –L命令，操作如下：

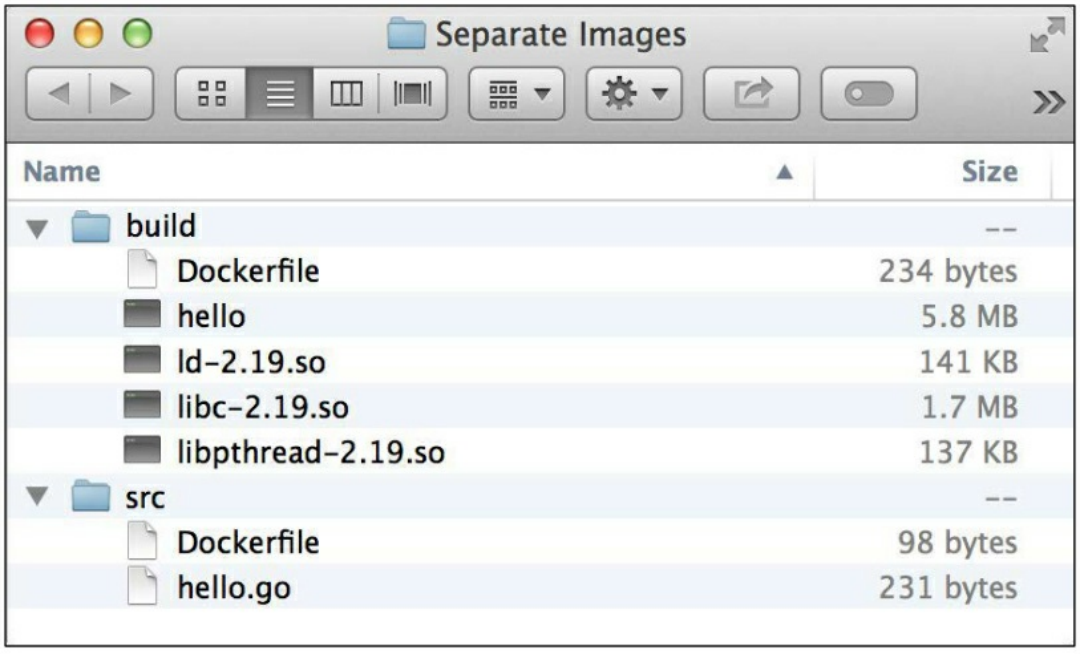
```
dockerhost$ docker cp -L large:/lib/x86_64-linux-gnu/li
                          ../build
dockerhost$ docker cp -L large:/lib/x86_64-linux-gnu/li
                          ../build
```

```
dockerhost$ docker cp -L large:/lib64/ld-linux-x86-64.s
../build
```

4. 创建一个新的Dockerfile用于编译这个只有二进制（binary-only）的镜像。注意，如何使用ADD指令将共享库添加到Docker镜像中，操作如下：

```
FROM scratch
ADD hello /app/hello
ADD libpthread-2.19.so \
/lib/x86_64-linux-gnu/libpthread.so.0
ADD libc-2.19.so /lib/x86_64-linux-gnu/libc.so.6
ADD ld-2.19.so /lib64/ld-linux-x86-64.so.2
EXPOSE 8080
ENTRYPOINT ["/app/hello"]
```

5. 现在，所有必要的文件都在这个“binary-only”的镜像中，文件夹中的目录结构树如下图所示。



6．最后，采用build/Dockerfile文件编译这个用于部署的二进制Docker镜像，最终生成的镜像将比原来的小，操作如下：

```
dockerhost$ docker build -t binary .
dockerhost$ docker images
REPOSITORY  TAG     IMAGE ID    CREATED          VIRTUAL
binary      latest  45c327c815  seconds ago      7.853 M
largeapp    latest  47a64e67f   52 minutes ago   523.1 M
golang      1.4.2   124e21271   5 days ago       517.3 M
```

同样的方法可以用于编译其他应用，例如通常采用．/configure&& make&& make install方式安装的那些软件。同样，可以用于那些解释性编程语言的应用程序，如Python、Ruby或者PHP。但是，创建一个“运行时”的Ruby语言的Docker镜像还需要进行一些额外处理。这种优化技术的最佳实践案例是在一个可持续开发流程中的应用程序的场景，并且它由于镜像太大导致传输时间太长。

小结

在本章中，我们学习了关于Docker如何编译镜像并应用这些知识来改进各种参数，包括部署时间、编译时间和镜像尺寸。本章的技术并不全面，肯定还有其他人针对自己的应用对Docker做出相应的优化。更多的新技术将会随着Docker技术的成熟和开发而被发现。驱使我们做这些优化的动力在于不断地问自己：我们是否从使用Docker中得到了帮助。其他类似的问题如下：

- 部署时间减少了吗？
- 在运行应用的过程中，开发团队从运营团队中得到反馈的速

度是否有提升？

- 当用户在使用应用的过程中发现问题时，我们是否能够快速迭代新的产品特性？

时刻记住使用Docker的出发点和目标，我们旨在提高工作流效率。

利用先前提到的优化技巧将需要对现有Docker宿主机进行调整。为了批量管理多台Docker宿主机，我们需要使用一些自动化管理配置工具。在下一章中，我们将学习如何用配置管理工具来自动化配置Docker宿主机。

3　用Chef自动化部署Docker

学到现在，我们已经了解了Docker生态系统具有多样性的特点，Docker宿主机有很多配置参数。但是，手动配置Docker宿主机通常费时费力且容易出错。如果没有自动化的策略，那么在生产环境中批量部署Docker将会出现很多隐患。

在本章，我们将学习用配置管理来解决这个问题。我们将采用Chef来批量管理Docker宿主机，Chef是一个配置管理软件。本章将涉及以下话题：

- 配置管理的重要性
- Chef的介绍
- 自动化配置Docker宿主机
- 部署Docker容器
- 可选的自动化工具

配置管理简介

Docker引擎有很多参数需要进行配置，如cgroups、内存、CPU、

文件系统、网络环境等。识别Docker容器运行在哪个Docker宿主机上是另一方面的配置。Docker容器需要配置不同的cgroups、共享卷、链接容器、公开端口等，组合这些参数去优化应用程序将很耗时。

手动复制一份上述配置到另一台Docker宿主机是很困难的。我们可能会忘记创建一个宿主机的全部步骤，这个过程不仅慢并且容易出错。创建一个文档来记录全部过程同样不会起作用，因为这个文件会随着时间的推移而过期失效。

如果不能以一种及时且可靠的方式提供配置给新的Docker宿主机，我们将没有办法批量部署Docker应用。一致且快速的准备和配置Docker宿主机对我们同样重要。否则，对于应用程序来说，Docker创建容器包的能力也将很快变得没有意义。

配置管理是一种策略，用于管理应用程序的变更、报告和审核系统的变更。它不仅仅在开发应用系统时使用，而且在我们的案例中，它同时记录Docker宿主机的变更，以及Docker容器应用本身的变更。在某种意义上，Docker为我们的应用程序完成了配置管理，包括如下内容：

- 从应用开发、演示、测试到部署到生产环境，Docker容器可以复现应用程序的全部环境。
- 编译Docker镜像是一种简便使应用程序更新并部署到所有环境中的方法。
- Docker允许团队成员掌握应用程序运行所需要的全部信息，同时可以将必要的变更高效地交付给客户。深入Dockerfile，他们可以知道应用程序的哪部分需要更新以及正常运行所需的依赖。

- Docker记录我们对Docker镜像所做的全部环境变更。然后，它通过相应版本的Dockerfile记录，同时记录了变更的内容、变更的操作人以及变更的发生时间。

但是，运行我们应用的Docker宿主机的变化呢？就像Dockerfile用版本控制帮助管理应用程序运行环境一样，配置管理工具可以用代码记录Docker宿主机的变更。它简化了创建Docker宿主机的过程。在批量部署Docker应用程序时，我们可以便捷地创建新Docker宿主机。当有硬件错误时，可以从它们原有的配置基础上启动新的Docker宿主机。如果需要部署Docker容器的新版本，我们只需将Docker宿主机的配置指向新镜像。配置管理使我们可以管理Docker批量部署。

使用Chef

Chef是一个配置管理工具，它提供了特定领域的语言对基础设施的配置进行建模。基础设施中的每一个配置项被定义为一个资源。一个资源基本上是一个Ruby方法，它可以在一个块内接收多个参数。下面例子中的资源描述的是安装docker-engine软件包：

```
package 'docker-engine' do
  action :install
end
```

这样的资源汇集在Ruby源代码文件中，叫作recipe。当在一个服务器（在我们的案例中指Docker宿主机）上执行recipe时，所有已经定义过的资源将被执行并达到预期配置的状态。

有些Chef的recipe可能会依赖某些支持项，如配置模板和其他recipe。这类recipe和相应的信息都已经记录在cookbook中。cookbook

是服务器分布式配置和策略的基础单元。

我们将编写Chef的recipe来表示Docker宿主机的预期状态。而我们的recipe将被组织在cookbook中，用于分发到基础设施。但是，让我们先来准备Chef的运行环境，然后开始在recipe中描述基于Docker的基础设施环境。一个Chef的环境包括如下部分：

- 一个Chef服务器
- 一个工作站
- 一个节点

接下来的几节将详细介绍每个组件。然后，我们将搭建它们来准备Chef环境，用于管理Docker宿主机。

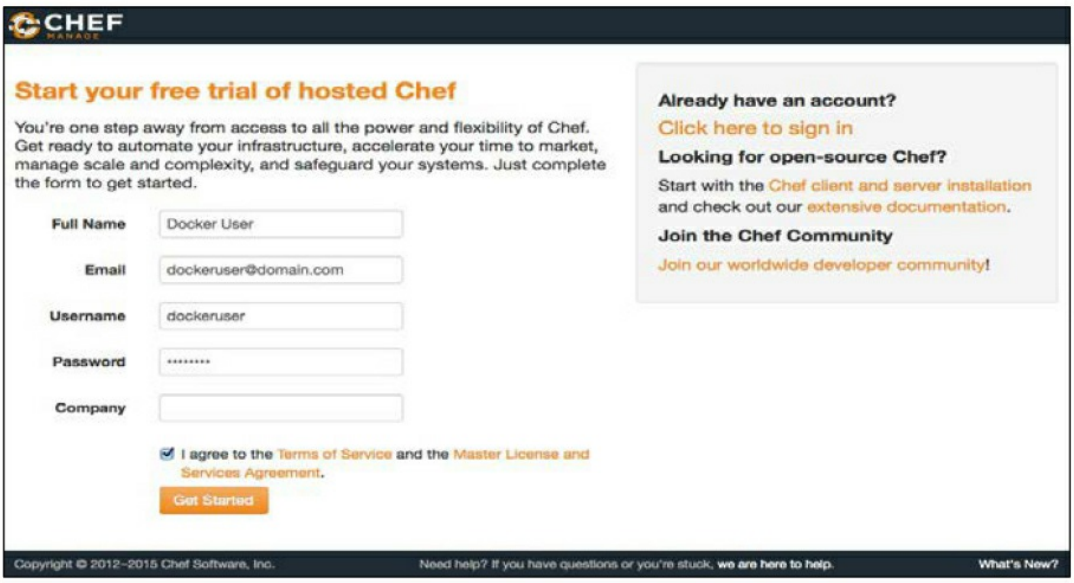
搭建Chef环境的更多细节超出了本书讨论的范围，更多相关信息可以在Chef官方文档网站查阅，地址为：http://docs.chef.io。

注册**Chef**服务器

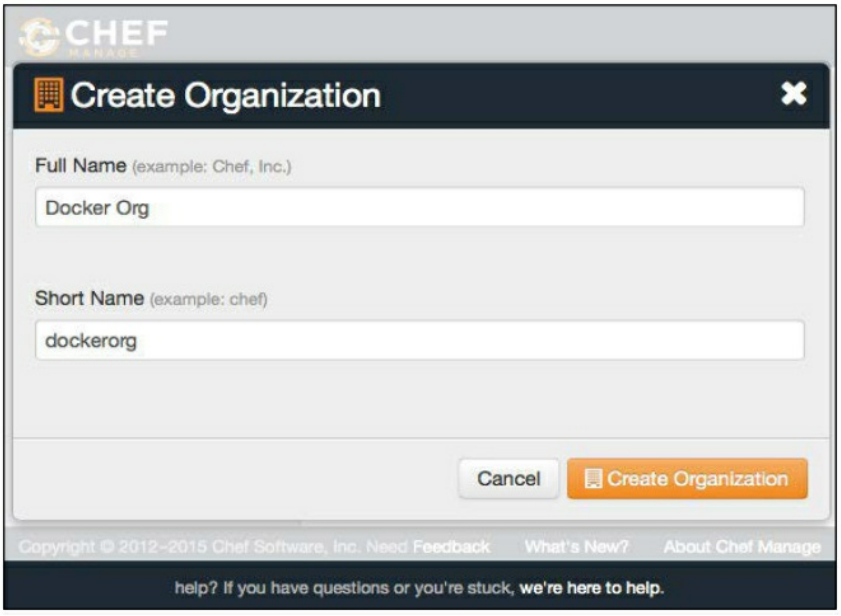
Chef服务器是我们整个基础设施的cookbook和其他规则项的中心资源库。它包含了我们管理的基础设施的元信息。在这个案例中，Chef服务器包括cookbook、策略、Docker宿主机的元信息。

准备Chef服务器前，我们需要简单地注册一个私有Chef服务器。一个免费的Chef服务器允许我们在基础设施中管理最多5个节点。准备一个私有Chef服务器账号的步骤如下所示。

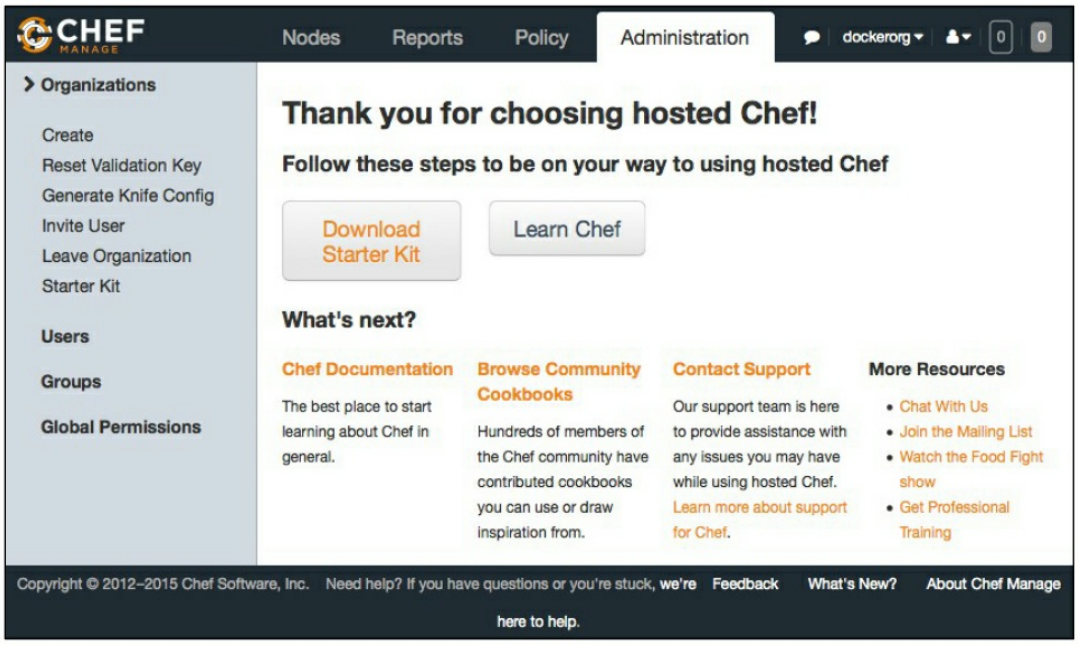
1．打开https://manage.chef.io/signup网页并填写账号信息，截图如下所示。



2．创建完用户账号之后，私有Chef服务器允许我们创建一个组织。组织仅仅是Chef服务器基于角色控制的访问权限管理方式。创建组织需要提供表单上的详细信息并单击“Create Organization”按钮，截图如下所示。



3．我们基本完成了创建私有Chef服务器账号的操作。最后，单击下图中的“Download Starter Kit”按钮，它将下载一个包含chef-repo的压缩文件。在下一节中，我们将对chef-repo进行详细介绍。



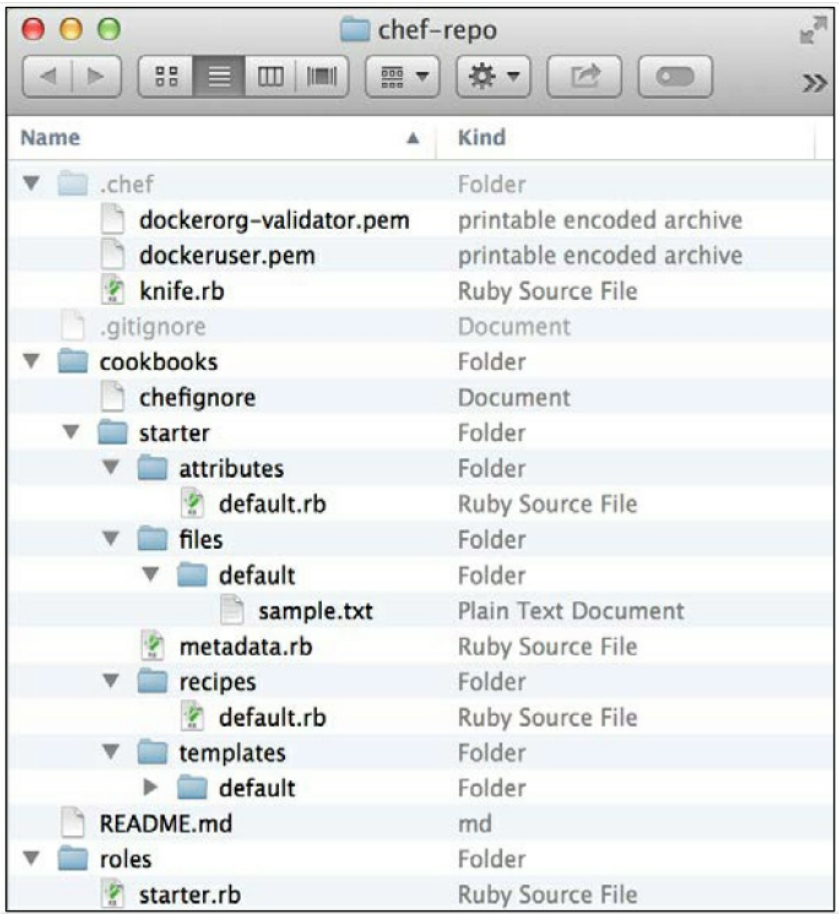
搭建工作站

创建Chef环境的第二步是搭建工作站，工作站用于跟Chef服务器进行交互，我们的大部分准备工作和代码都是由它发送给Chef服务器的。在工作站中，我们将把基础设施的配置项放置在一个Chef资源库中。

Chef资源库包含了所有需要交互的信息并且保持与Chef服务器同步。它还保存了私有密钥以及其他需要授权的配置文件，并与Chef服务器保持交互。这些文件存储在Chef资源库的．chef文件夹中。它还包括我们将要编写和与Chef服务器同步的cookbook，并存储在cookbooks/文件夹中。在Chef资源库中还包括其他文件和文件夹，例如数据包、角色和环境变量等。但是，目前了解cookbook和授权文件可以配置Docker宿主机已经足够了。

还记得我们在前一节下载的启动工具包吗？解压到chef-repo中，目录树结构如下图所示。

工作站的另一个重要组件是Chef开发工具包。它包含所有与读取chef-repo中配置信息有关的程序，并与Chef服务器进行交互。Chef开发工具包同样提供了便于开发的代码、测试和cookbook。在本章随后的内容中，我们将使用到开发工具包中的各类程序。



接下来，根据我们工作站的平台环境下载所需要的Chef开发工具包，网址为：https://downloads.chef.io/chef-dk，如下图所示。

然后，打开安装工具，根据平台提示进行Chef开发工具包的安装。最后，确认正确安装完成，操作如下：

```
$ chef -v
Chef Development Kit Version: 0.6.2
chef-client version: 12.3.0
berks version: 3.2.4
kitchen version: 1.4.0
```

既然已经完成了工作站的安装，让我们切换到chef-repo/文件夹并完成Chef环境的最后一项准备工作。

启动节点

创建Chef环境的最后一部分是节点，一个节点可以是Chef管理下的任何一台电脑。它可以是一个物理主机、虚拟主机、云主机或者是一个网络设备。在我们的案例中，这里的节点是Docker宿主机。

对于Chef管理的节点来说，最重要的组件是chef-client。它负责连接Chef服务器，并下载必要的配置文件，同时将节点配置到预期状态。当一个chef-client在节点上运行时，它的工作流程如下所示：

1．它在Chef服务器上注册并授权管理当前节点。

2．它收集节点上的系统信息用于创建一个节点对象。

3．然后，根据节点需要同步相应的Chef cookbook。

4．通过加载节点所需的recipe来编译资源。

5．然后，执行所有的资源并按照相应的操作去配置节点。

6．最后，它报告chef-client的结果给Chef服务器和其他已配置的消息终端。

现在，让我们将Docker宿主机通过工作站启动它成为一个节点。引导过程将安装并配置chef-client，启动引导过程操作如下所示：

```
$ knife bootstrap dockerhost
...
Connecting to dockerhost
dockerhost Installing Chef Client...
...
dockerhost trying wget...
dockerhost Comparing checksum with sha256sum...
dockerhost Installing Chef 12.3.0
dockerhost installing with dpkg...
...
dockerhost Thank you for installing Chef!
dockerhost Starting first Chef Client run...
dockerhost Starting Chef Client, version 12.3.0
dockerhost Creating a new client identity for dockerhost u
validator key.
dockerhost resolving cookbooks for run list: []
dockerhost Synchronizing Cookbooks:
dockerhost Compiling Cookbooks...
dockerhost ... WARN: Node dockerhost has an empty run list
dockerhost Converging 0 resources
dockerhost
dockerhost Running handlers:
dockerhost Running handlers complete
```

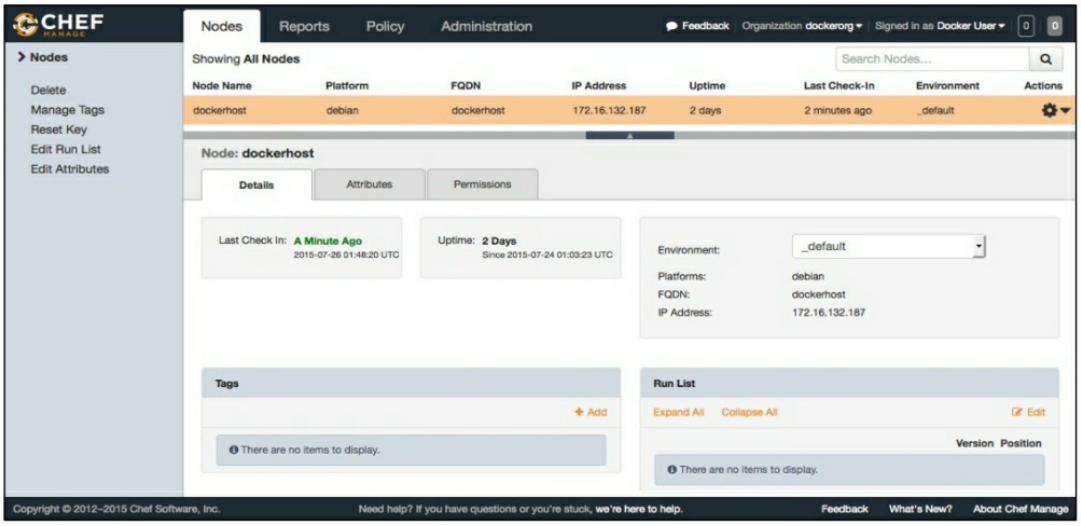
```
dockerhost Chef Client finished, 0/0 resources updated in
```

上面运行的记录显示，引导过程分为两步：首先，它在Docker宿主机节点安装并配置了chef-client。然后，它启动chef-client来与Chef服务器同步预期的状态。由于我们还没有配置任何状态给Docker宿主机，所以它没有同步任何配置。

可以根据自己的需要配置引导过程。更多关于如何使用knife bootstrap的说明可以在官方文档上查阅，网址为：http://docs.chef.io/knife_bootstrap.html。

在很多案例中，云提供商已经对Chef做了深度整合。所以，我们只需要使用云提供商的SDK即可，不需要用knife bootstrap。并且，我们只需要声明需要Chef集成，只需提供给它相应的配置信息，如chef-client的client.rb文件和一些授权密钥。

我们的Docker宿主机已经在Chef服务器中注册完毕后，可以同步其配置信息。访问网址：https://manage.chef.io/organizations/dockerorg/nodes/dockerhost来确认Docker宿主机已经是Chef环境中的一个节点，截图如下所示。



配置Docker宿主机

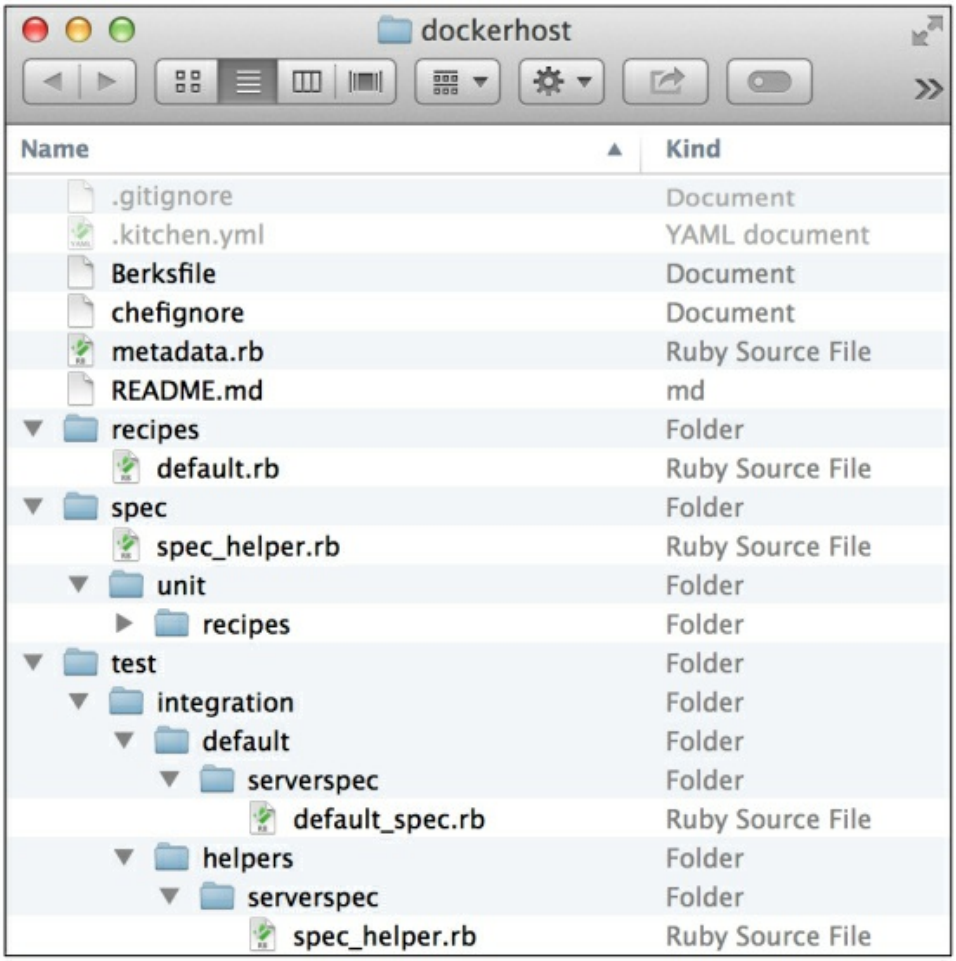
现在Chef环境所需的全部组件已经搭建完毕，可以启动Chef recipe实际描述Docker宿主机所需的配置。此外，我们将利用Chef生态系统中现有的Chef cookbook来超越现有生产能力。由于Docker是用于部署的一个流行的基础设施，所以我们可以用已有的cookbook对Docker宿主机进行配置。在Chef市场中可以找到社区提供的Chef cookbook。在Chef市场中可以找到其他可用的cookbook，网址为：http://supermarket.chef.io。

在本节中，你将学会如何编写Chef recipe并应用到节点中。为Docker宿主机编写recipe的步骤如下所示。

1. 使用Chef开发工具包中的chef generate cook命令生成cookbook模板。切换到cookbook的目录之后，执行如下命令：

```
$ cd cookbooks
$ chef generate cookbook dockerhost
```

命令生成的cookbook模板文件夹结构截图如下所示。



2. 然后，我们开始编辑cookbook。将工作目录切换到之前创建的cookbook目录下，操作如下：

```
$ cd dockerhost
```

3. 从Chef市场中安装相应的cookbook作为依赖，包括apt和docker。这些cookbook由第三方提供，它们可以在我们的recipe中使用。随后将使用它们作为编译块来搭建Docker宿主机。继续添加其他依赖到metadata.rb文件，内容如下：

```
name 'dockerhost'
maintainer 'The Authors'
maintainer_email 'you@example.com'
license 'all_rights'
```

```
description 'Installs/Configures dockerhost'
long_description 'Installs/Configures dockerhost'
version '0.1.0'

depends 'apt', '~> 2.7.0'
depends 'docker', '~> 0.40.3'
```

metadata.rb文件记录Chef cookbook中的元数据信息，而这些元数据信息将帮助Chef服务器指导cookbook正确部署到节点上。更多关于如何配置元数据信息到Chef cookbook上的内容请查阅官方文档，网址为：http://docs.chef.io/config_rb_metadata.html。

4. 既然已经声明了依赖关系，就可以开始下载并安装它们，操作如下：

```
$ berks install
Resolving cookbook dependencies...
Fetching 'dockerhost' from source at .
Fetching cookbook index from https://supermarket.chef.i
Installing apt (2.7.0)
Installing docker (0.40.3)
Using dockerhost (0.1.0) from source at .
```

5. 最后，我们将编写Chef recipe，它等同于安装说明书，可参

考网站文档：http:// blog.docker.com/2015/07/new-apt-and-yum-repos。我们将使用apt依赖cookbook中提供的apt_repository资源，然后将如下内容添加到recipes/default.rb:

```
  recipes/default.rb:
  apt_repository 'docker' do
    uri 'http://apt.dockerproject.org/repo'
    components %w(debian-jessie main)
    keyserver 'p80.pool.sks-keyservers.net'
    key '58118E89F3A912897C070ADBF76221572C52609D'
    cache_rebuild true
end

package 'docker-engine'
```

现在，我们已经完成了dockerhost/的Chef cookbook准备工作。最后一步是将它应用到我们的Docker宿主机上，使它可以选择所需配置，操作步骤如下所示。

1. 首先，上传Chef cookbook到Chef服务器。请注意，在下面的命令输出记录中，apt和docker的cookbook将会被自动上传到Chef服务器：

```
$ berks upload
Uploaded apt (2.7.0) to: 'https://api.opscode.../docker
Uploaded docker (0.40.3) to: 'https://api.ops.../docker
Uploaded dockerhost (0.1.0) to: 'https://api.opscode.co
organizations/dockerorg'
```

2. 然后，通过设置run_list应用dockerhost recipe到节点，也就是

我们的Docker宿主机，操作如下：

```
$ knife node run_list set dockerhost dockerhost
dockerhost:
    run_list: recipe[dockerhost]
```

3. 最后，在dockerhost中执行chef-client。chef-client将会抓取Docker宿主机的节点对象并应用所需配置，操作如下：

```
$ ssh dockerhost
dockerhost$ sudo chef-client
Starting Chef Client, version 12.3.0
resolving cookbooks for run list: ["dockerhost"]
Synchronizing Cookbooks:
 - apt
 - dockerhost
 - docker
Compiling Cookbooks...
Converging 2
resources Recipe:
dockerhost::default
 * apt_repository[docker] action add
   * execute[install-key 58118E89F3A912897C...] action
   ...
* apt_package[docker-engine] action install
  - install version 1.7.1-0~j... of package docker-engi
Running handlers:
Running handlers complete
Chef Cl... finished, 6/7 resources updated in 24.69 sec
```

现在，Docker已经通过Chef在我们的Docker宿主机上安装并配置完成。无论何时我们需要添加一个新的Docker宿主机，只需在云提供商中添加另一个服务器并用之前编写的名为dockerhost的Chef recipe来引导它即可。当需要更新全部Docker宿主机中的Docker守护进程的配置时，只需更新Chef的cookbook并重新启动chef-client即可。

在生产环境中，在宿主机中安装配置管理软件的目的是为了不需要登录就可修改配置变更。手动运行chef-client只是一个半自动化方式。我们希望chef-client作为一个守护进程运行，这样就不需要在每次变更之后都重新启动它。chef-client守护进程将连接Chef服务器来检查它管理的节点是否有新的配置变更。默认的配置中，同步请求为每30分钟一次。

更多关于如何配置chef-client为守护进程的信息请查阅官方文档，网址为：https://docs.chef.io/chef_client.html。

部署Docker容器

批量管理Docker的下一步就是在Docker宿主机池中自动部署Docker容器。到目前为止，我们已经编译完成多个Docker应用。我们有一个简单的架构关于这些容器间如何通信和相互调用。Chef recipe可以用代码来表示这个架构，它对于批量管理整个应用程序和基础设施是很关键的。我们可以标示哪些容器需要运行，以及容器间如何连接。可以定位Docker容器需要被部署在哪个宿主机。将这些架构用代

码记录下来可以为我们的应用部署编制策略。

在本节中，将创建一个Chef recipe来编排部署Nginx的Docker镜像到我们的Docker宿主机上。在前一节中，由docker cookbook提供用来配置Docker宿主机的Chef资源将会被用到。部署的具体步骤如下所示。

1. 首先，创建Chef recipe文件。如下命令将会在dockerhost/目录下创建recipes/containers.rb文件：

```
$ chef generate recipe . containers
```

2. 然后，从https://registry.hub.docker.com/_/nginx拉取官方的Nginx Docker镜像文件到我们的宿主机。然后在recipes/containers.rb中加入如下代码：

```
docker_image 'nginx' do
   tag '1.9.3'
end
```

3. 下载完Docker镜像之后，配置Docker宿主机运行该容器。在0.40.3版本之后的docker cookbook，我们需要指定安装Debian Jessie系统的Docker宿主机使用systemd作为它的init系统工具。同时，添加如下内容到recipes/containers.rb：

```
node.set['docker']['container_init_type'] = 'systemd'

directory '/usr/lib/systemd/system'

docker_container 'nginx' do
  tag '1.9.3'
```

```
    container_name 'webserver'
    detach true
    ort '80:80'
  end
```

docker_container和docker_image有很多可选项需要指定，用于标示对容器的操作。docker cookbook同样有其他资源与Docker宿主机交互。更多关于如何使用这些配置项的内容可以在项目主页找到，网址为：https://github.com/bflad/chef-docker。

4. 然后，我们将为发布而准备新版本的cookbook，可以通过在metadata.rb中指定版本信息来实现，内容如下：

```
name 'dockerhost'
maintainer 'The Authors'
maintainer_email 'you@example.com'
license 'all_rights'
description 'Installs/Configures dockerhost'
long_description 'Installs/Configures dockerhost'
version '0.2.0'

depends 'apt', '~> 2.7.0'
depends 'docker', '~> 0.40.3'
```

5. 更新Berksfile.lock文件来锁定所有将要上传到Chef服务器的

cookbook文件版本。操作如下：

```
$ berks install
Resolving cookbook dependencies...
Fetching 'dockerhost' from source at .
Fetching cookbook index from https://supermarket.chef.i
Using dockerhost (0.2.0) from source at .
Using apt (2.7.0)
Using docker (0.40.3)
```

6. 目前新的cookbook的所有修改均已完成，我们需要输入下面的指令来上传cookbook到Chef服务器。请注意，berk upload命令自动识别出只有dockerhost这个cookbook需要更新，而跳过了apt和docker这两个cookbook：

```
$ berks upload
Skipping apt (2.7.0) (frozen)


Skipping docker (0.40.3) (frozen)
Uploaded dockerhost (0.2.0) to: 'https://ap.../dockeror
```

7. 接下来，添加recipes/containers.rb到Docker宿主机的运行列表。更新节点表示的Docker宿主机，操作如下：

```
$ knife node run_list add dockerhost dockerhost::contai
dockerhost:
  run_list: recipe[dockerhost]
  recipe[dockerhost::containers]
```

8. 最后，重新运行chef-client来更新Docker宿主机的配置。如果已经配置了chef-client的守护进程，我们也可以等着chef-

client自动重启。操作如下：

```
$ ssh dockerhost
dockerhost$ sudo chef-client
Starting Chef Client, version 12.3.0
resolving cookbooks for run list: ["dockerhost",
"dockerhost::containers"]
Synchronizing Cookbooks:
  - dockerhost
  - apt
  - docker
Compiling Cookbooks...
Converging 5 resources Recipe:
dockerhost::default
...
Recipe: dockerhost::containers
  * docker_image[nginx] action pull

  * directory[/usr/lib/systemd/system] action create
    - create new directory /usr/lib/systemd/system
  * docker_container[nginx] action run
   * template[/usr/lib/.../webserver.socket] action cre
    ...
   * service[webserver] action enable (up to date)
   * service[webserver] action start
     - start service service[webserver]
  * template[webserver.socket] action nothing ...
  * template[webserver.service] action nothing ...
  * service[webserver] action nothing ...
```

```
Running handlers:
Running handlers complete
Chef Client finished, 6/10 resources updated in 42.83 s
```

现在Docker宿主机已经运行起Nginx容器。我们可以通过访问http://dockerhost来确认它已经正常工作，应该可以看到如下图所示的页面。

可选方案

这里还有其他用于帮助配置Docker宿主机的通用配置管理工具。列表如下所示。

- **Puppet**：网址为http://puppetlabs.com。
- **Ansible**：网址为http://ansible.com。
- **CFEngine**：网址为http://cfengine.com。

- **SaltStack**：网址为http://saltstack.com。

- **The Docker machine:** 这是一个非常特定的配置管理工具，它允许我们在基础设施中提供和配置Docker宿主机。关于Docker machine的更多信息可以从Docker的官方文档网站查阅，网址为：https://docs.docker.com/machine。

如果你完全不想管理Docker宿主机的基础设施，可以使用Docker托管服务。很多主流的云提供商开始提供以Docker宿主机为云镜像的服务。其他提供更加完善功能的解决方案使我们可以操作所有云端Docker宿主机像在一个Docker虚拟宿主机中一样。主流云提供商在Docker生态系统中描述了他们的整合方案，列表如下所示。

- Google Container Engine (https://cloud.google.com/container-engine)

- Amazon EC2 Container Service (http://aws.amazon.com/documentation/ecs)

- Azure Docker VM Extension (https://github.com/Azure/azure-docker-extension)

- Joyent Elastic Container Service (https://www.joyent.com/public-cloud)

在部署Docker容器方面，同样还有很多容器工具可以使用。它们通过提供API的方式来运行和部署Docker容器。有些API甚至可以与Docker engine媲美。这可以使我们在与Docker宿主机池交互时像使用一台虚拟Docker宿主机一样方便。将容器部署到Docker宿主机池的工具列表如下所示。

- Docker Swarm (https://www.docker.com/docker-swarm)

- Google Kubernetes (http://kubernetes.io)
- CoreOS fleet (https://coreos.com/fleet)
- Mesophere Marathon (https://mesosphere.github.io/marathon)
- SmartDataCenter Docker Engine (https://github.com/joyent/ sdc-docker)

然而，还是需要类似于Chef的配置管理工具在Docker宿主机池上部署和配置我们的编排系统。

小结

在本章，我们学习了如何自动化配置Docker部署工作。使用Chef可以帮助批量配置并提供多个Docker宿主机，它使我们可以为应用程序部署和编排Docker容器到宿主机池中。从这个角度看，你需要编写Chef recipe来持久化所有已经学过的Docker优化技巧。

在下一章中，我们将引入仪表盘（instrumentation）来管理整个Docker基础设施和应用。这将给我们带来进一步的关于如何优化Docker部署以得到更高效率的反馈。

4　监控Docker宿主机和容器

前几章介绍了优化Docker部署的方法，以及如何扩展以提高性能，但是如何知道我们的调整是正确的呢？能够监控Docker架构和应用对发现为什么需要和何时需要调优至关重要。评测系统性能使我们能够发现扩展和调整的上下限。

除了监控底层Docker信息外，评测应用的业务相关性能也很重要。通过跟踪应用的某些系统值，可以将系统值跟应用相关值对应起来，这样Docker开发和运维团队可以对业务部门展示Docker如何节省了成本并增加了业务价值。

在本章中，会针对如下专题进行讨论。

- 监控的重要性
- 在Graphite中搜集监控数据
- 在colleted中监控数据
- 在ELK栈中整合日志
- 从Docker容器中发送日志

监控的重要性

监控提供了对Docker部署的一种反馈，回答了从底层操作系统性能到高层业务目标的一系列问题。在Docker宿主机中插入合适的工具对于确定系统工作状态很关键，我们可以使用这种反馈来确定应用是否工作正常。

如果初始假设不正确，就可以根据反馈调整计划、改变系统设置，或者更新运行中的Docker应用。投产后，也可以使用同样的监控流程来区分错误和bug。

Docker有内置的日志和监控功能，默认Docker宿主机将Docker容器的标准输出和错误输出存放在/var/lib/docker/<container_id>/<container_id>-json.log文件中，docker logs命令请求Docker引擎读取相关文件内容并显示出来。

另一个监控命令是docker stats，通过向Docker引擎远程API /containers/<container_id>/stats服务点发送请求，将运行中容器的control group信息（CPU、内存、网络使用）实时读取出来，如下示例就是docker stats的输出：

```
dockerhost$ docker run --name running –d busybox \
    /bin/sh -c 'while true; do echo hello && sleep 1; don
dockerhost$ docker stats running
CONTAINER   CPU %  MEM USAGE/LIMIT   MEM %  NET I/O
running     0.00%  0 B/518.5 MB      0.00%  17.06 MB/119.8
```

内置的docker logs和docker stats对于监控开发目的的和小规模部署的容器应用很有用。但是当管理一个有成千上万台容器主机的大型生产系统时，此方法就不再有效了，因为不可能登入每个容器都敲入

这两个命令。

逐个登入会对了解全局Docker部署带来困难，而且并不是所有对容器应用感兴趣的人都可以登入Docker宿主机。业务人员可能只对Docker部署应用如何提高企业处理能力感兴趣，他们不需要了解底层命令。

因此，能够将容器部署的所有事件和数据集中到一个统一管理界面非常重要，这使得运维扩展只需要在这里看系统发生了什么。集中化的仪表盘使得各方人员，例如业务人员都可以访问监控系统。本章其他小节将展示如何整合docker logs和docker stats的数据。

收集数据到Graphite

为了开始监控Docker部署，首先要设置一个服务点以便发送监控数据。Graphite是一个常用的搜集数据的工具栈，明文协议使得它使用起来非常简单。许多第三方工具都使用明文协议，后面，我们将展示将数据发往Graphite的操作多么简单。

Graphite的另一个功能是可以将实时收集的数据渲染出来，根据这些图形整合成仪表盘，仪表盘可以显示我们关心的各种监控数据。

在本节中，我们将设置如下Graphite构成模块，生成一个最小工作栈。

- **carbon-cache**：这是Graphite通过网络接收数据的模块。除了实现明文协议，也监听叫作pickle的二进制协议，这种协议更先进而且简单，极大地优化了数据格式。

- **whisper**：这是一个基于时序的数据库，存放carbon-cache接收

的数据。固定大小的特性非常适合作为监控的组件。随着时间流逝，监控数据越来越多，数据库也会越来越大。

实际上，更多时候想监控某个时点的数据，因此，可以按照监控计划来设计whisper数据库，而不仅将其作为一个存储来装大量的数据。

- **graphite-web**：这是读取whisper数据库中的数据，以图形显示在仪表盘的模块。也可以用于创建，并实时渲染显示carbon-cache对服务请求返回的数据（它们已经被写入whisper库中）。

carbon中还有不少其他模块，例如carbon-aggregator和carbon-relay。它们都是扩展Graphite必需的模块。更多信息可以从如下网址获得：https://github.com/graphite-project/carbon。目前，我们只需要carbo-cache创建一个简单的Graphite集群。

下面描述如何部署carbon-cache和whisper数据库。

1．准备一个Docker镜像，生成如下Dockerfile：

```
FROM debian:jessie

RUN apt-get update && \
    apt-get --no-install-recommends \
        install -y graphite-carbon
```

```
ENV GRAPHITE_ROOT /graphite

ADD carbon.conf /graphite/conf/carbon.conf
RUN mkdir -p $GRAPHITE_ROOT/conf && \
    mkdir -p $GRAPHITE_ROOT/storage && \
    touch $GRAPHITE_ROOT/conf/storage-aggregation.conf
    touch $GRAPHITE_ROOT/conf/storage-schemas.conf

VOLUME /whisper
EXPOSE 2003 2004 7002

ENTRYPOINT ["/usr/bin/twistd", "--nodaemon", \ "--react
    "--no_save"]
CMD ["carbon-cache"]
```

2. 使用第一步中的Dockerfile生成 hubuser/carbon映像：

```
dockerhost$ docker build -t hubuser/carbon .
```

3. 在carbon.conf中，需要配置carbon-cache使用Docker卷/whisper来持久化数据，存放whisper数据库：

```
[cache]

CARBON_METRIC_INTERVAL = 0
LOCAL_DATA_DIR = /whisper
```

4. 生成hubuser/carbon映像后，准备生成数据容器存放whisper数据库：

```
dockerhost$ docker create --name whisper \
    --entrypoint='whisper database for graphite' \
    hubuser/carbon
```

5. 最后，运行早先生成的carbon-cache服务映像。我们将使用定制化名称和暴露的端口收发数据：

```
dockerhost$ docker run --volumes-from whisper -p 2003:2
    --name=carboncache hubuser/carbon
```

我们现在有一个存放Docker相关数据的地方了，为了使用这些数据，需要读取和显示的方法。下面需要部署graphite-web模块，步骤如下所示。

1. 生成创建hubuser/graphite-web容器的Dockerfile：

```
FROM debian:jessie
RUN apt-get update && \
    apt-get --no-install-recommends install -y \
    graphite-web \
    apache2 \
    libapache2-mod-wsgi

ADD local_settings.py /etc/graphite/local_settings.py
RUN ln -sf /usr/share/graphite-web/apache2-graphite.co
       /etc/apache2/sites-available/100-graphite.conf
    a2dissite 000-default && a2ensite 100-graphite &&
    mkdir -p /graphite/storage && \
    graphite-manage syncdb --noinput && \
    chown -R _graphite:_graphite /graphite
```

```
EXPOSE 80
ENTRYPOINT ["apachectl", "-DFOREGROUND"]
```

2. 第一步中的Docker映像中提到local_settings.py，将如下定义放入carbon-cache容器和whisper卷中：

```
import os
# --link-from carboncache:carbon
CARBONLINK_HOSTS = ['carbon:7002']
# --volumes-from whisper
WHISPER_DIR = '/whisper'

GRAPHITE_ROOT = '/graphite'
SECRET_KEY = os.environ.get('SECRET_KEY', 'replacekey')
LOG_RENDERING_PERFORMANCE = False
LOG_CACHE_PERFORMANCE = False
LOG_METRIC_ACCESS = False
LOG_DIR = '/var/log/graphite'
```

3. 准备好Dockerfile和local_settings.py配置文件后，创建hubuser/graphite-web容器：

```
dockerhost$ docker build -t hubuser/graphite-web .
```

4. 最后，运行hubuser/graphite-web Docker映像，链接carbon-cache容器和whisper卷：

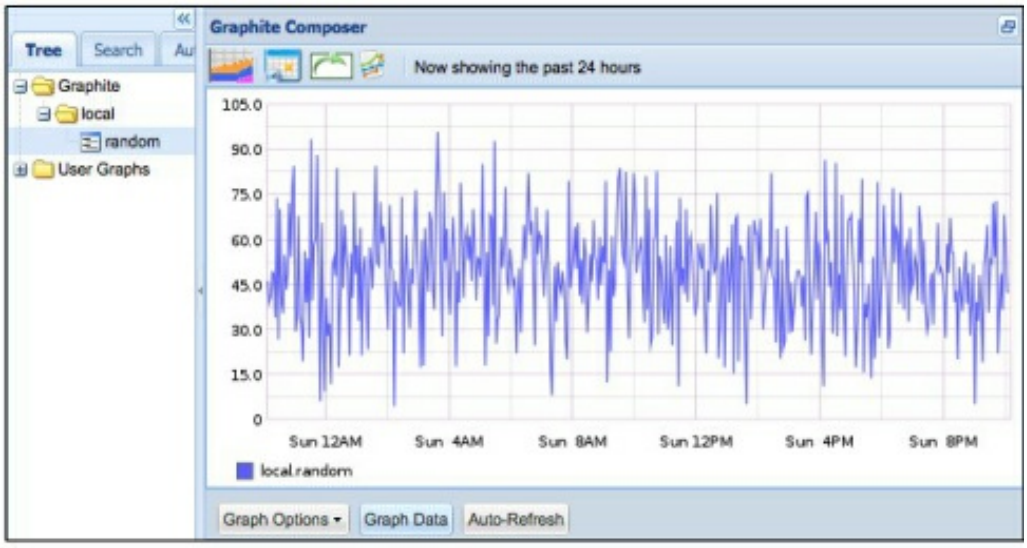
```
dockerhost$ docker run --rm --env SECRET_KEY=somestring
    --volumes-from whisper --link carboncache:carbon \
    -p 80:80 hubuser/graphite-web
```

如果想扩展graphite-web实例的话，SECRET_KEY环境变量是必不可少的。更多关于graphite-web扩展的信息可参见：http://graphite.readthedocs.org/en/latest/config–local-settings.html。

现在，我们有了一个完成的Graphite部署，可以运行一个简单测试来看看是否运转正常。如下命令将向carbon-cache服务点发送叫作local.random的随机数据。

```
dockerhost$ seq `date +%s` -60 $((`date +%s` - 24*60*60))
    | perl -n -e \
    'print "local.random ". int(rand(100)) . " " .  $_' \
    | docker run --link carboncache:carbon -i --rm \
    busybox nc carbon 2003
```

最后，确认数据都存放好了，可以访问hubuser/graphite-web的URLhttp://dockerhost/compose，在Tree选项卡中，可以展开Graphite/local目录，就可以访问到random数据。如下图所示。



生产系统中的**Graphite**

在生产系统中，简单配置的Graphite随着监控越来越多的Docker部署，数据会越来越多而达到极限，这时就需要扩展Graphite为集群配置。

为了扩展carbon-cache的处理能力，需要carbon-relay和carbon-aggregator两个模块。为了提高graphite-web的响应能力，需要跟其他缓存组件，例如memcached一同水平扩展。也可能需要级联graphite-web的实例以便获得统一的视图。whisper数据库也需要与carbon-cache和graphite-web一同扩展。

更多关于扩展Graphite集群的信息可以从如下网址获得：http://graphite.readthedocs.org。

用collectd监控

配置完监控数据存放处后，下面需要从Docker应用中获取具体数据了。本节，我们会使用collectd，这是一个常用的系统统计数据搜集保护进程，也是一个很轻量但高性能的C程序，几乎不对监控的系统造成资源上的影响。因为是轻量级应用，部署起来很简单。有很多插件可以监控几乎所有系统的模块。

现在开始监控Docker宿主机，用如下步骤安装collectd，将数据发送到Graphite：

1．在Docker宿主机内安装collectd：

```
dockerhost$ apt-get install collectd-core
```

2．创建一个最小collectd配置，发送数据到Graphite。在之前配置carbon-cache时，暴露了默认明文端口2003，
在/etc/collectd/collectd.conf中进行如下配置：

```
LoadPlugin "write_graphite"
<Plugin write_graphite>
  <Node "carboncache">
    Host "dockerhost"
  </Node>
</Plugin>
```

3．下一步需要从Docker宿主机内监控一些参数，将如下插件定义放到/etc/collectd/collectd.conf中：

```
/etc/collectd/collectd.conf 中：
```

```
LoadPlugin "cpu"
LoadPlugin "memory"
LoadPlugin "disk"
LoadPlugin "interface"
```

4. 配置完成后，重启collectd：

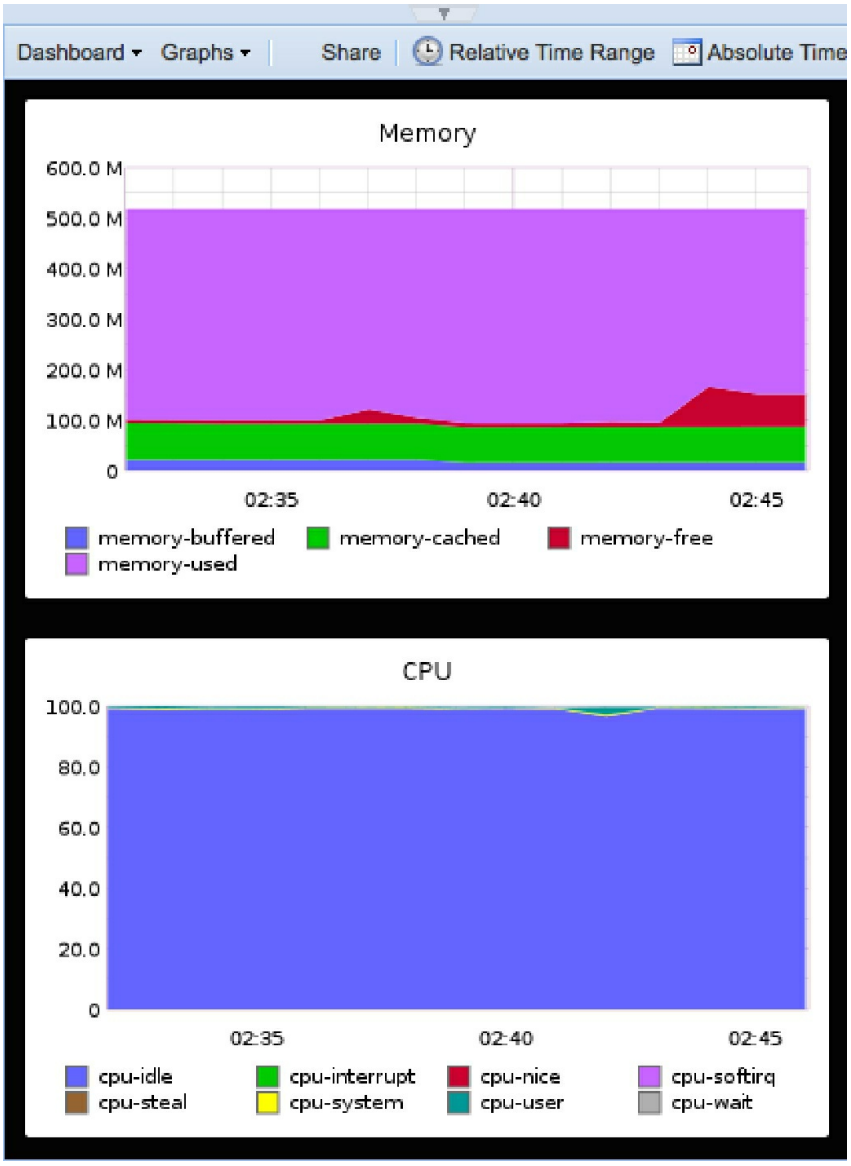
```
dockerhost$ systemctl restart collectd.service
```

5. 最后，在graphite-web部署内生成一个图形化仪表盘来表示之前收集的数据。访问http://dockerhost/dashboard，单击“Dashboard”，再单击“Edit　Dashboard”链接，会弹出仪表盘配置文件，输入如下JSON格式的文本：

```
[
  {
    "areaMode": "stacked",
    "yMin": "0",
    "target": [
      "aliasByMetric(dockerhost.memory.*)"
    ],
    "title": "Memory"
  },
  {
    "areaMode": "stacked",
    "yMin": "0",
    "target": [
      "aliasByMetric(dockerhost.cpu-0.*)"
    ],
    "title": "CPU"
```

```
    }
  ]
```

我们现在有了一个监控Docker宿主机的基础栈，最后一步显示的仪表盘截图如下所示。



收集Docker相关数据

现在，我们评测一些跟应用相关的性能指标，但是如何深入容器内部呢？在基于Debian Jessie的Docker宿主机上，容器运行在docker-

[container_id].scope控制组内，信息可以从Docker宿主机sysfs下的/sys/fs/cgroup/cpu、cpuacct/system.slice获得。幸运的是，collectd也有一个cgroups插件跟之前暴露的sysfs信息交互，如下步骤将展示如何使用此插件来评测被监控Docker容器的CPU性能。

1. 首先在/etc/collectd/collectd.conf中插入如下配置：

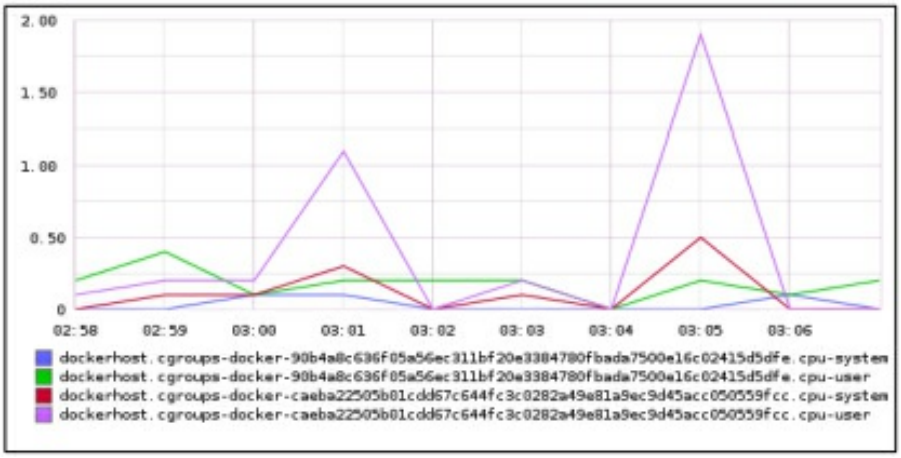
```
LoadPlugin "cgroups"


<Plugin cgroups>
  CGroup "/^docker.*.scope/"
</Plugin>
```

2. 重启collectd服务：

```
dockerhost$ systemctl restart collectd.service
```

3. 最后，等几分钟，Graphite会从collectd中获得足够多的数据，我们可以获得初始图形化Docker容器CPU的性能数据。

我们可以访问Graphite部署的渲染API dockerhost.cgroups-docker*.*，从而检查Docker容器内CPU的数据。下图是访问如下API URL获得的图像化输出：http://dockerhost/render/?target=dockerhost.cgroups-docker-*.*。



更多关于cgroups插件的信息可以从如下网址获得：
https://collectd.org/documentation/manpages/collectd.conf.5.shtml#p

现在，cgroups插件只评测Docker容器的CPU数据，虽然还有其他的监控项目在进行中，但写书时并没有完成。幸运的是，有一个基于Python的collectd插件可以跟docker stats交互，以下是安装步骤。

1．首先，下载运行插件的依赖包：

```
dockerhost$ apt-get install python-pip libpython2.7
```

2．其次，下载并安装Github的插件安装包：

```
dockerhost$ cd /opt
dockerhost$ git clone https://github.com/lebauce/docker
plugin.git
dockerhost$ cd docker-collectd-plugin dockerhost$ pip i
requirements.txt
```

3．添加如下信息到/etc/collectd/collectd.conf配置文件中：

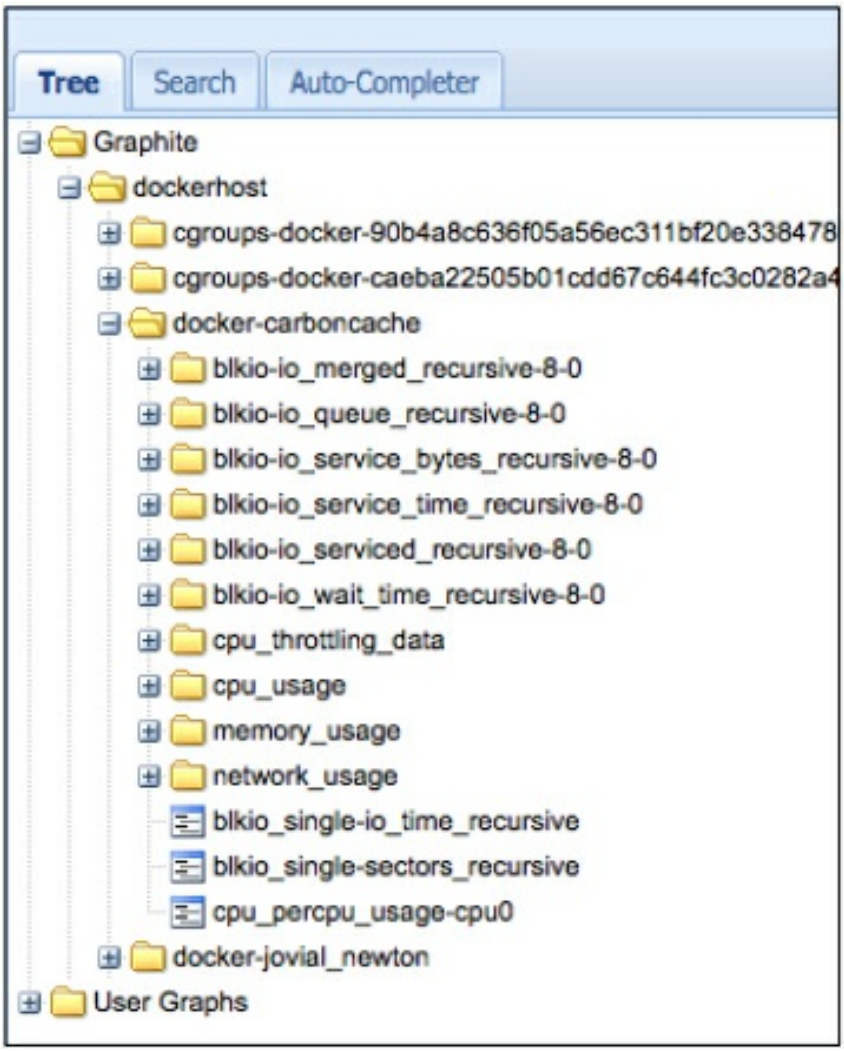
```
TypesDB "/opt/docker-collectd-plugin/dockerplugin.db"
LoadPlugin python

<Plugin python>
  ModulePath "/opt/docker-collectd-plugin"
  Import "dockerplugin"

 <Module dockerplugin>
  BaseURL "unix://var/run/docker.sock"
  Timeout 3
 </Module>
</Plugin>
```

4．最后，重启collectd服务：

```
dockerhost$ systemctl restart collectd.service
```

有时候不想安装完整Python栈访问Docker stat服务点，这时，可以使用collectd底层的curl_json插件从容器中获取统计信息。可以配置向容器stat服务点发出请求，解析出JSON格式的collectd数据。更多关于插件的信息可以从如下网址获得：https://collectd.org/documentation/manpages/collectd.conf.5.shtml#p

在如下截图中，我们可以利用Graphite部署http://docker/compose从cgroups插件中获得的数据：



在**Docker**内部运行**collectd**

如果想像其他应用一样部署collectd，可以在Docker内部运行它。如下是初始化Dockerfile的操作，可以用于启动一个collectd部署的Docker容器。

```
FROM debian:jessie

RUN apt-get update && \
```

```
    apt-get --no-install-recommends install -y \
        collectd-core

ADD collectd.conf /etc/collectd/collectd.conf
ENTRYPOINT ["collectd", "-f"]
```

许多插件会检查/proc和/sys文件系统，容器内部的collectd为了访问这些文件，需要挂载Docker卷，例如--volume/proc:/host/proc。然而，现在很多插件直接读取硬编码的/proc和/sys目录，有很多关于如何使得这种配置更加灵活的讨论。可参见如下Github页面：https://github.com/collectd/collectd/issues/1169。

在**ELK**栈中整合日志

在我们现有的collectd和Graphite监控方案中，不是所有Docker宿主机和容器都可以被获得。有些事件和数据只是作为日志文件存在，但我们需要将这些文本转化为易懂的非结构化日志，以便跟裸数据一样，可以从中分析出基于Docker应用的更高层的问题。

ELK栈是一个弹性的，解决此类问题的包组合，每个字母代表一个功能。其解释如下。

- **Logstash**：Logstash是用于收集管理日志和事件的模块，是收

集不同日志源（多个Docker宿主机和容器）的中心点。我们可以使用Logstash转化和注释接收的日志，并且允许日志中包含丰富的信息。

- **Elasticsearch**：Elasticsearch是一个高扩展性的分布式搜索引擎，分片功能允许随着数据的增长，扩展日志存储。它的数据库引擎是面向文档的，使得我们存储和注释日志更加方便。

- **Kibana**：Kibana是Elasticsearch的分析和搜索仪表盘。简易性使得我们可以为Docker应用生成不同仪表盘。Kibana非常易于定制化，因此可以给关心的人们提供有价值的内部关系，技术上说是底层的，但是业务上很有意义。

在本节剩余部分中，我们设置组件的每个模块，将Docker宿主机和容器日志发送到其中，以下步骤是如何配置ELK栈。

1. 首先，激活官方Elasticsearch映像，使用容器名字以便后续步骤引用：

```
dockerhost$ docker run -d --name=elastic elasticsearch
```

2. 运行Kibana映像跟前一步中的Elasticsearch容器链接起来。现在可以公开将5601端口映射到80端口，这样Kibana URL看起来很美观：

```
dockerhost$ docker run -d --link elastic:elasticsearch
   -p 80:5601 kibana:4.1.1
```

3. 准备Logstash Docker映像和配置文件Dockerfile：

```
FROM logstash:1.5.3
```

```
ADD logstash.conf /etc/logstash.conf
EXPOSE 1514/udp
```

4. 在Docker映像中，配置Logstash为一个Syslog服务器，解释了上一步中暴露UDP端口的原因。在如下logstash.conf文件中，基础配置将侦听Syslog服务。后一部分配置是发送日志到称为elasticsearch的Elasticsearch实例，我们将使用这个作为链接Elasticsearch容器的主机名：

```
input {
  syslog {
    port => 1514 type => syslog
  }
}

output {
  elasticsearch {
    host => "elasticsearch"
  }
}
```

Logstash有很多有价值的插件，可以读取很多信息源的日志数据。特别的，有一个collectd codec插件，可以使用ELK栈来代替Graphite监控数据。

更多配置信息可参见：

https://www.elastic.co/guide/en/logstash/current/plugins-codecs-collectd.html。

5．我们准备了所有需要的文件，生成hubuser/logstash映像：

```
dockerhost$ docker build -t hubuser/logstash .
```

6．执行Logstash。注意，我们暴露1514端口给Docker宿主机作为Syslog端口，还链接了之前生成的称为elastic的容器。目标名称为elasticsearch，因为其主机名在之前配置的logstash.conf中配置为Elasticsearch，并将接收日志数据：

```
dockerhost$ docker run --link elastic:elasticsearch -d \
    -p 1514:1514/udp hubuser/logstash -f /etc/logstash.con
```

7．配置Docker宿主机的Syslog服务转发到Logstash容器。作为基本配置，我们设置Rsyslog转发所有日志，也包括所有从Docker引擎发来的日志。需要编辑/etc/rsyslog.d/100-logstash.conf，加入如下内容：

```
/etc/rsyslog.d/100-logstash.conf，加入如下内容：
*.* @dockerhost:1514
```

8．最后，重启Syslog服务：

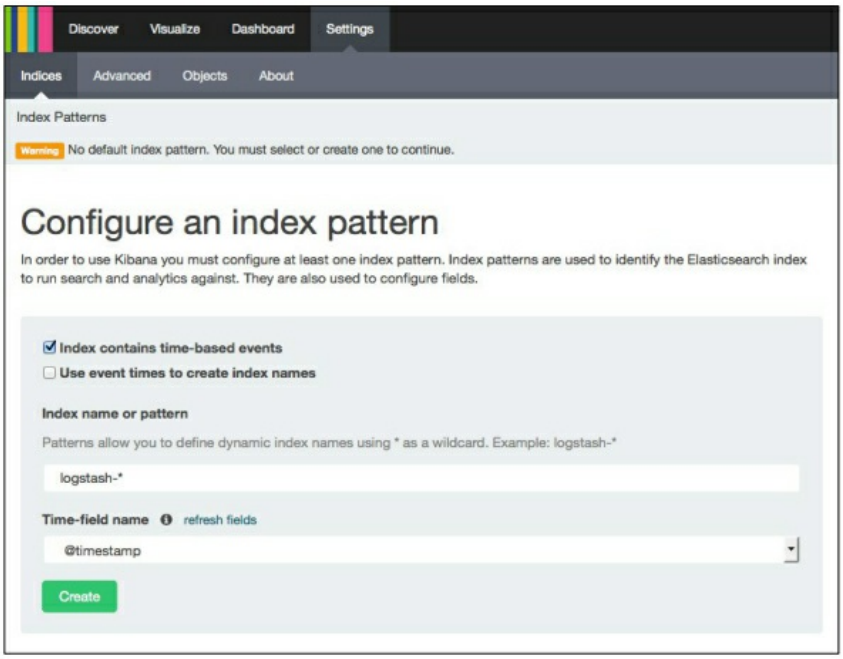
```
dockerhost$ systemctl restart rsyslog.service
```

我们现在有了一个具有基本功能的ELK栈，现在发送一个简单消息到Logstash，确认其是否会出现在Kibana仪表盘。

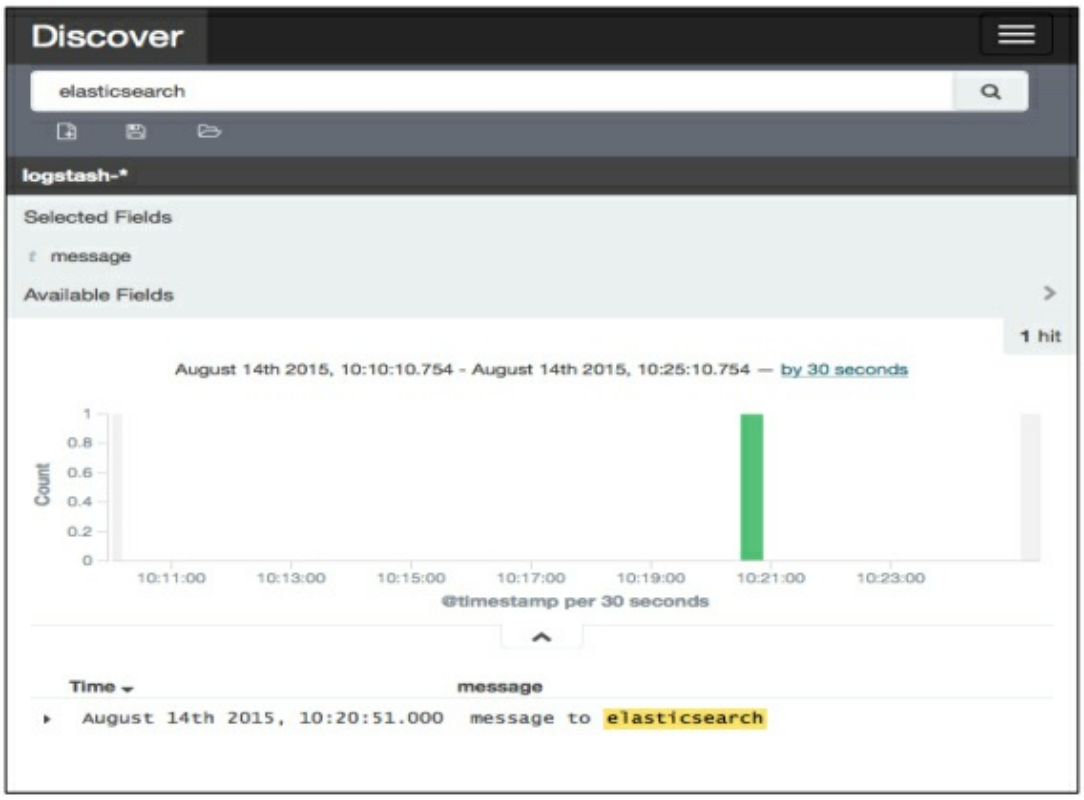
1．首先用如下命令发送消息：

```
dockerhost$ logger -t test 'message to elasticsearch'
```

2．访问Kibana仪表盘http://dockerhost。Kibana要求设置默认索引格式，使用如下默认值，单击“Create”按钮，开始索引，如下图所示。



3．访问http://dockerhost/#discover，在搜索框中键入elasticsearch。如下图所示展示了刚才发送的Syslog信息：



ELK栈中有很多用于日志架构优化的组件。可以添加Logstash插件，过滤和注释从Docker容器和主机中接收的日志。Elasticsearch随着规模扩大，可以进行扩展，还可以生成Kibana仪表盘分享信息。更多关于ELK栈的信息可以访问如下网址：https://www.elastic.co/guide。

转发**Docker**容器日志

现在我们有了一个具有基本功能的ELK栈，可以将Docker日志转发到其中。在Docker　1.7以后，定制化日志驱动已经被支持。在本节

中，我们配置Docker宿主机使用Syslog驱动。默认的，Docker中的Syslog事件将会发往Docker宿主机的Syslog服务中，因为配置了宿主机Syslog转发消息到ELK栈，我们会在ELK栈中看到容器日志。配置步骤如下所示。

1. Docker引擎已经通过Debian　Jessie主机Systemd配置完毕。更新在Docker宿主机中运行，生成Systemd文件/etc/systemd/system/docker.service.d/10-syslog.conf，代码如下所示。

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// \
    --log-driver=syslog
```

2. 重启Systemd服务，以便新配置生效：

```
dockerhost$ systemctl daemon-reload
```

3. 最后，重启Docker引擎：

```
dockerhost$ systemctl restart docker.service
```

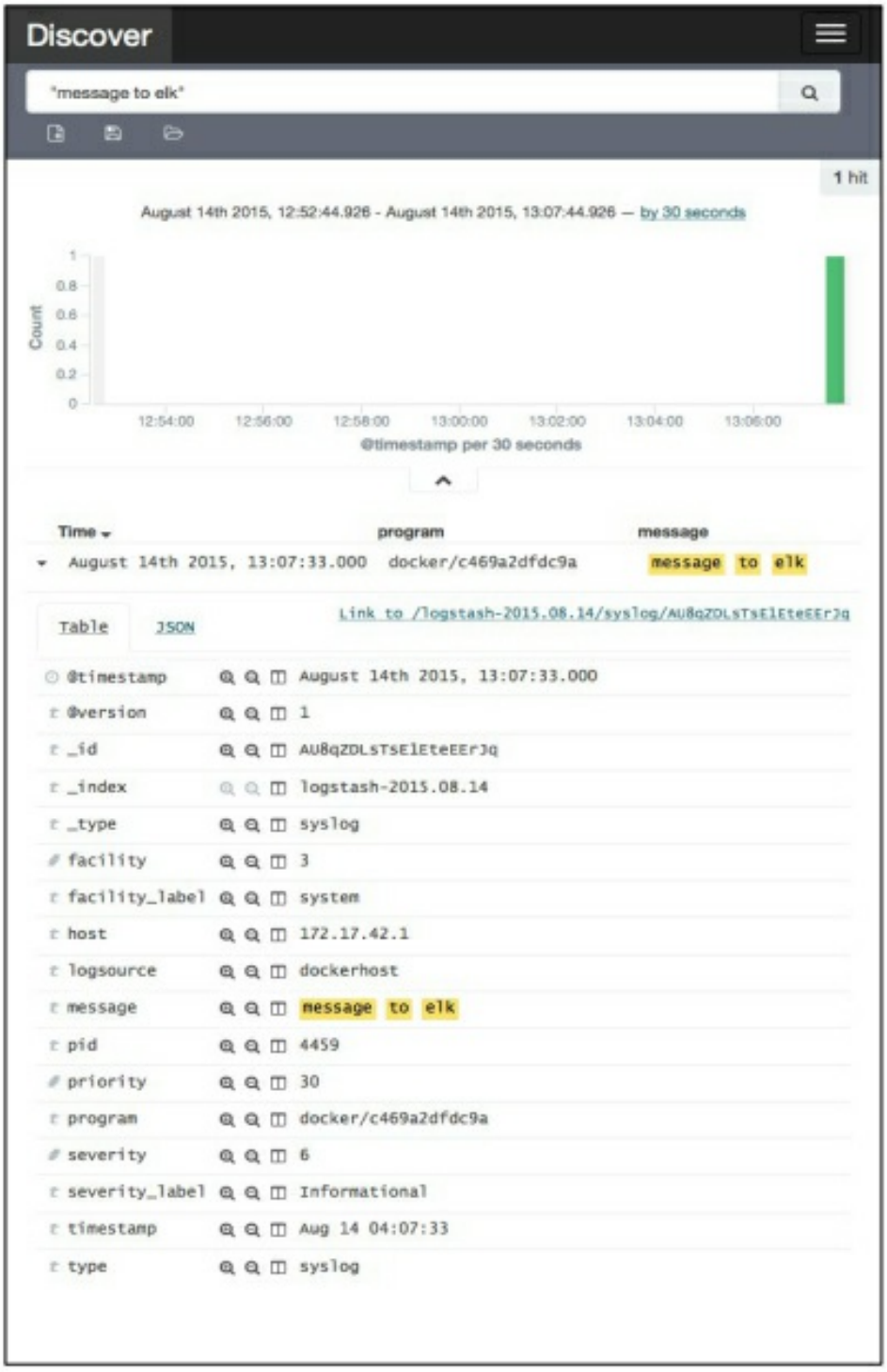
4. 可选的，如果想定制化Docker容器中的日志注释，可以应用任何Logstash过滤器。

现在，任何标准输出和错误输出都通过Docker容器转发到ELK栈，我们可以做一些简单测试来验证工作是否正常：

```
dockerhost$ docker run --rm busybox echo message to elk
```

docker run命令也支持在希望运行的容器中使用--log-driver和--log-opt=[]命令行选项设置日志卷。使用这些选项可进行深层次的容器主机调试。

执行完上述测试后，消息应该存在Elasticsearch中。访问Kibana访问点http://dockerhost，搜索关键词“message to elk”，显示出之前发送的消息。如下截图展示了Kibana中的结果。



在上图中，可以看到发送的测试消息，其中也含有其他关于Syslog的消息。Docker　Syslog驱动设置默认Syslog注释和优先级分别为**system**和**informational**。除此之外，前一个程序被设置为：**docker/c469a2dfdc9a**。

c469a2dfdc9a字符串是我们之前busybox映像的容器ID。默认容器标签被设置为：docker/<container-id>　格式。所有之前的默认注释都可以通过传递--log-opt=[]选项来实现。

除了Syslog和JSON文件日志驱动外，Docker还支持其他几种访问点来接收日志。更多关于日志驱动和各自的使用方法可参见如下网址：https://docs.docker.com/reference/logging。

其他监控和日志方案

还有其他几种方案可部署支持基于Docker应用的监控和日志架构。某些已经内置支持监控Docker容器，其他应该也有各种组合解决方案。例如之前我们描述的，因为它们只是专注于监控或者日志的某一个方面。

对于其他方案，我们应该也有二次开发方案。但是，相对于它们带来的便捷性来说，这些付出还是值得的。如下列表包括但不仅限于这些，可以用于日志和监控方案：

- cAdvisor (http://github.com/google/cadvisor)
- InfluxDB (http://influxdb.com)
- Sensu (http://sensuapp.org)

- Fluentd (http://www.fluentd.org/)
- Graylog (http://www.graylog.org)
- Splunk (http://www.splunk.com)

有时候，运维团队和开发人员对开发和运行Docker的应用并不很熟练，或者说并不想维护监控和日志架构。还有一些监控和日志平台，我们可以直接使用它们，这样就可以将注意力更多地放在优化Docker应用上。

某些可以跟现存的监控和日志代理一同工作，例如Syslog和collectd。而对于其他的，我们需要下载和部署它们的代理以便将事件和数据转发到平台上，如下列表列出了一些但不仅限于它们：

- New Relic (http://www.newrelic.com)
- Datadog (http://www.datadoghq.com)
- Librato (http://www.librato.com)
- Elastic's Found (http://www.elastic.co/found)
- Treasure Data (http://www.treasuredata.com)
- Splunk Cloud (http://www.splunk.com)

小结

我们现在知道为什么以可扩展和可访问模式监控Docker部署非常重要了。我们部署collectd和Graphite来监控Docker容器数据，并将不

同Docker宿主机和容器上的日志整合到ELK栈上进行分析。

除了裸数据和事件，知道对应用来说意味着什么也很重要。Graphite-web和Kibana允许我们生成定制化的仪表盘和分析，并提供了对Docker应用深层次的了解。因为有了这些监控工具和技术，我们才可以将Docker运行和部署于生产系统中。

在下一章中，我们将开始做一些性能测试，提供一个在高负载情况下应用的基准。我们应该能够使用监控系统观察和验证性能行为。

5 性能基准测试

为了优化应用，确认每个参数是否优化很重要。性能基准对于确定Docker容器中每个元素是否表现正常很重要。应用本身有很多需要调整的地方。Docker宿主机有诸如内存、网络、CPU和存储等参数可以调整。根据应用的实际情况，某个或者某些参数成为瓶颈，针对不同元素进行测试对于指导调优策略至关重要。

此外，生成合适的性能测试，也可以了解基于现在配置的Docker应用能力的上限，由此来决定是否需要部署更多Docker宿主机来扩展应用处理能力，或者是将现有应用转移到更大存储、内存或者CPU的Docker宿主机。如果我们有混合云部署，就可以使用这些评测来确定哪个云提供商可以为应用提供更好的支持。

评测应用如何对性能基准进行响应，对规划容器架构能力非常重要。生成测试计划模拟峰值和正常时候的场景，使得我们可以在应用投产后预测到应用将会如何运转。

在本章中，我们会讨论如下问题：

- 为性能基准配置Apache JMeter
- 生成和设计性能基准负载
- 分析应用性能

配置Apache JMeter

Apache JMeter是用于测试Web应用最常用的性能工具。除了负载测试，此开源项目还支持其他网络协议测试，例如LDAP、FTP和裸TCP包。其可配置性很好，在设计复杂应用场景时很强大，可以模拟成千上万用户突然访问Web应用产生峰值的场景。

负载测试软件具有另外一个功能，就是数据截获和分析能力。JMeter支持从数据复制到制图和分析等一系列功能，可以立刻用于分析性能结果。它还支持很多插件，包括负载模式、分析或者网络连接等，可以直接使用它们。

更多如何使用Apache JMeter功能的知识可以在如下网址中获得：http://jmeter.apache.org。

在本节中，我们部署一个用于性能分析的应用，并且准备一个运行JMeter性能测试的工作站。

部署一个简单应用

为了描述方便，本章我们会部署一个简单Web应用，此应用是使用Unicorn（一个常用的Ruby应用服务器）部署的，是基于Ruby的Web应用，其主要功能是通过UNIX socket接收从Nginx发来的数据，

这种Ruby应用架构非常常见。

在本节中，我们将Ruby应用部署在称为webapp的Docker宿主机中，我们将应用、性能工具和监控分别部署在不同Docker宿主机中，这样性能和监控就不会影响到应用本身。

如下是部署Ruby Web应用的步骤：

1．生成Rack config.ru文件，配置Ruby应用：

```
app =  proc do |env|

  Math.sqrt rand
  [200, {}, %w(hello world)]
end
run app
```

2．将应用作为Docker容器打包：

```
FROM ruby:2.2.3

RUN gem install unicorn
WORKDIR /app
COPY . /app

VOLUME /var/run/unicorn

CMD unicorn -l /var/run/unicorn/unicorn.sock
```

3．生成nginx.conf配置文件，它将通过上一步生成的UNIX socket转发请求到Unicorn应用服务；记录请求的同时，将复

制$remote_addr和$response_time。后续将着重分析这两个参数：

```
events { }
http {
  log_format unicorn ''$remote_addr [$time_local]''
             '' ""$request"" $status''
             '' $body_bytes_sent $request_time'';
  access_log /var/log/nginx/access.log unicorn;
  upstream app_server {
    server unix:/var/run/unicorn/unicorn.sock;
  }
  server {
    location / {
      proxy_pass http://app_server;
    }
  }
}
```

4. 将之前生成的Nginx配置用如下Dockerfile打包成Docker容器：

```
FROM nginx:1.9.4
COPY nginx.conf /etc/nginx/nginx.conf
```

5. 通过docker-compose.yml将前两个容器组合起来进行部署：

```
web:
  log_opt:
    syslog-tag: nginx
```

```
    build: ./nginx
    ports:
      - 80:80
    volumes_from:
      - app
  app:
    build: ./unicorn
```

最后，在代码里可以看到如下图所示的目录结构：

现在开始部署Docker宿主机：

```
webapp$ docker-compose up -d
```

Docker　Compose是一个生成多容器应用的工具，通过在YML文件中定义的schema描述Docker容器如何运行以及如何连接。

Docker Compose支持curl|bash安装方法，要进行快速安装，代码如下所示：

```
dockerhost$ curl -L https:// github.com/docker/compose/
releases/download/1.5.2/docker-compose-`uname -s`-`uname
-m` \
                    > /usr/local/bin/docker-compose
```

我们只是简单使用Docker compose，如果想深入了解，可以参见如下网址：http://docs.docker.com/compose。

最后，简单测试安装是否成功：

```
$ curl http://webapp.dev
hello world
```

准备工作完成，下一节，我们将通过安装JMeter准备基准测试的工作站。

安装JMeter

本章剩余部分都是用Apache JMeter 2.13进行测试的。本节，我们将下载和安装工作站，步骤如下：

1．前往JMeter网站的下载页面http://jmeter.apache.org/download_jmeter.cgi。

2．选择apache-jmeter-2.13.tgz开始下载。

3．下载完毕，展开压缩文件：

```
$ tar -xzf apache-jmeter-2.13.tgz
```

4．将bin目录加入$PATH环境变量：

```
$ export PATH=$PATH:`pwd`/apache-jmeter-2.13/bin
```

5．运行JMeter：

```
$ jmeter
```

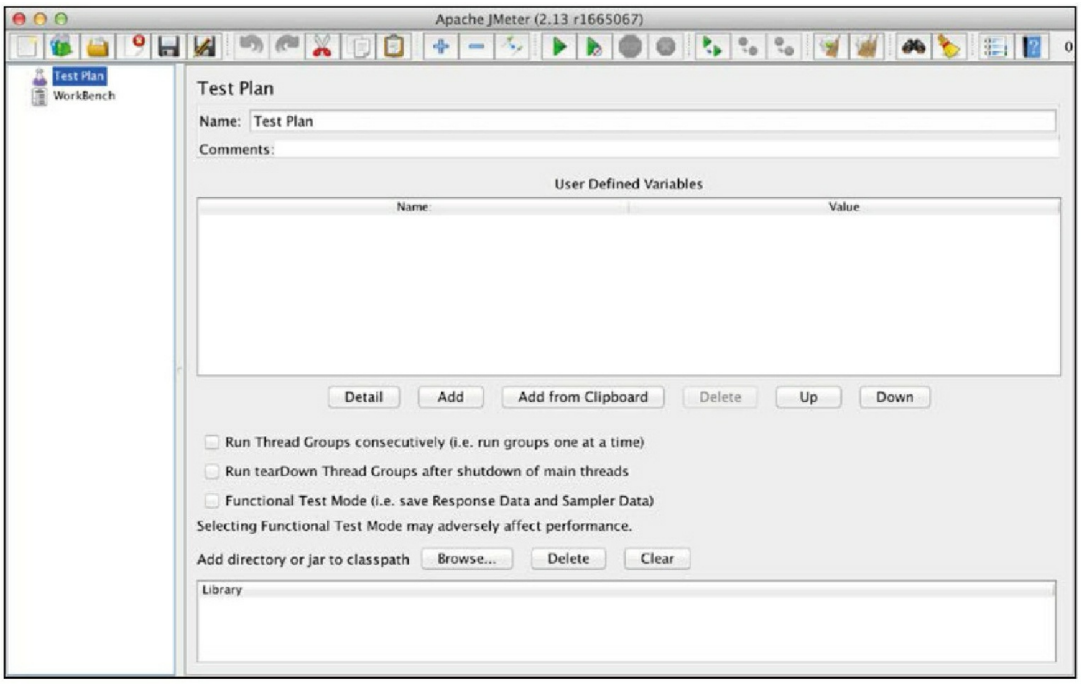
JMeter UI的截图如下所示，现在已经准备好基准测试工具。

注意，Apache JMeter是Java应用，请确保工作站Java Runtime Environment至少是1.6版本以上。

如果是Mac OSX，可以使用Homebrew进行安装：

```
$ brew install jmeter
```

如需更多JMeter的相关信息，可以参见如下网址：http://jmeter.apache.org/usermanual/get-started.html。



生成性能负载

为应用生成性能基准是很开放的问题。Apache JMeter在此领域独树一帜，有不少可以调整的参数。作为开端，我们可以使用如下问题作为开始：

- 应用是做什么的？
- 我们的用户有什么特点？
- 用户和应用之间如何交互？

从这些问题开始，可以将它们转换成对应用的需求。

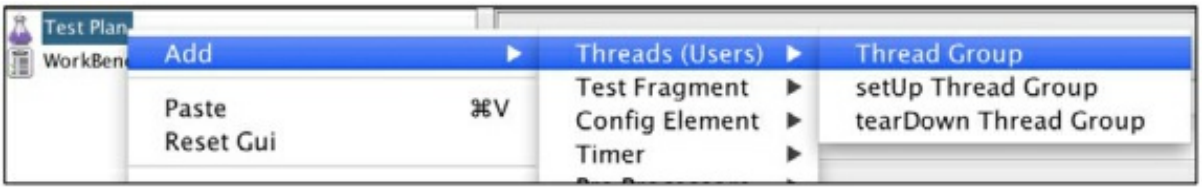
在前一节的简单应用中，我们生成了一个Web应用，给用户请求返回“hello　world”。对于Web应用，我们一般更关心吞吐量和响应时间，吞吐量意味着有多少用户可以同时接收到“hello　world”；响应时间则是客户在接收到“hello world”前要等待的时间。

本节中，我们使用Apache JMeter生成一个基本性能基准，然后使用这个基准和第4章中的监控工具对应用进行初始分析，然后对应用进行调整，使我们对应用有更好的了解。

在JMeter中生成测试计划

Apache JMeter中的性能测试都是通过测试计划（test plan）来描述的，一个测试计划描述了JMeter要做的一系列动作，例如向应用发送请求。每一步都被称为一个元素，每个元素又包含若干其他元素，看起来更像树状结构。

为了在测试计划中添加元素，需要在父元素上单击右键，然后点选“Add”项，在打开的文本菜单中选择需要加入的元素。下图所示的是在测试计划中加入了Thread Group。



下面的几个步骤展示了如何创建我们想要的测试计划：

1. 首先，重命名“test plan”为“Unicorn Capacity”，单击“test plan”元素，即可直接进行更新。

2. 在Unicorn Capacity测试计划中，创建thread group，将其命名为Application Users。我们配置此thread group每秒单线程发送10,000个请求：

   — **Number of Threads:** 1

— **Ramp-up Period:** 0 seconds

— **Loop Count:** 120,000 times

当生成测试计划时，将Loop Count设置得低一些是有意义的。我们可以从10,000，甚至10开始。这样响应时间会很短，可以立刻得到响应。当完成整个测试计划后，可以返回来调整这些参数。

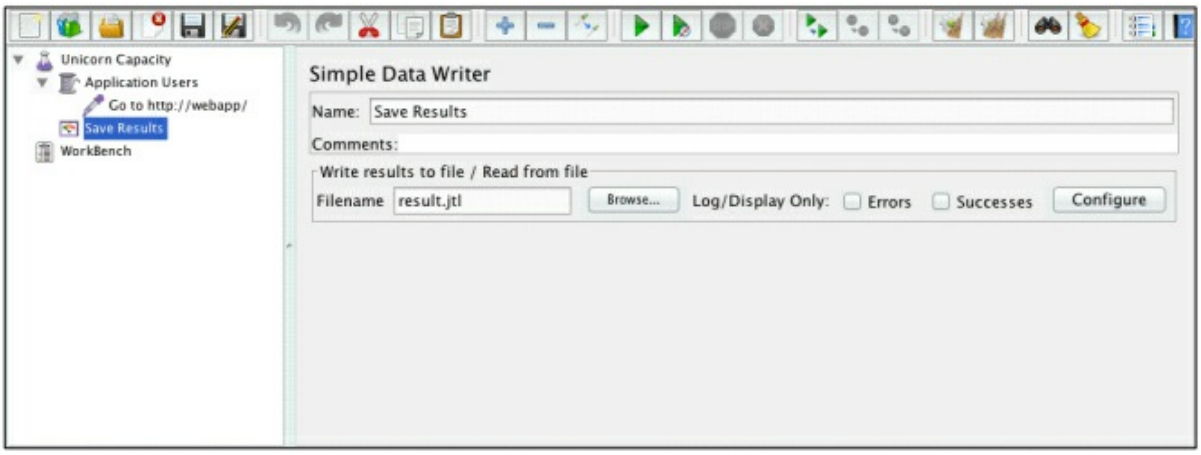
3．在“Application Users" thread group下添加“Sampler, HTTP Request”，生成真正的请求。配置如下：

— **Name:** Go to http://webapp/

— **Server Name:** webapp

4．最后，配置如何保存数据。在“Unicron Capacity”下添加一个侦听器，添加“Simple Data Writer”，命名为“Save Result”。设置文件名为result.jtl来保存性能测试结果。我们分析性能结果时需要用到这个文件名。

现在我们生成了基本测试负载，在http://webapp/产生了120,000个HTTP请求，测试计划将数据保存到result.jtl文件中，如下图所示：



可以运行性能测试了。在“Run”菜单中选择“Start”项开始执行测试计划。计划执行时，Start按钮变灰即为不可执行，执行结束后，按钮重新变为可用。

执行性能测试后，下一节我们将分析存储在result.jtl文件中的内容。

JMeter测试计划中有各种类型的元素可用，除了本节中用到的外，还有很多可用于正规请求、网络请求和分析数据。如需更多信息，可以参见如下网址：
http://jmeter.apache.org/usermanual/component_reference.html。

分析基准测试结果

本节我们来分析基准测试结果，以确定120,000个请求是如何影响应用的。当生成Web应用性能基准时，我们一般对两类事情很感兴

趣：

- 应用同时能处理多少个请求？
- 应用对每个请求的响应时间是多少？

这两类Web性能数据很容易转换成应用对业务的影响。例如，多少用户在使用应用？另外，从用户角度来看，它们又是如何对应用的响应起作用的？我们可以将它们与系统CPU、内存和网络对应起来，从而决定系统的能力。

检查JMeter运行结果

JMeter中有几个侦听器元素有渲染图形的能力，部署测试计划时激活这些功能非常有用。但是UI实时图形处理会影响系统真正的运行性能，因此，最好将运行和分析功能分开。在本节中，我们将生成新的测试计划，分析从result.jtl中获得的数据，看看JMeter侦听器元素的作用。

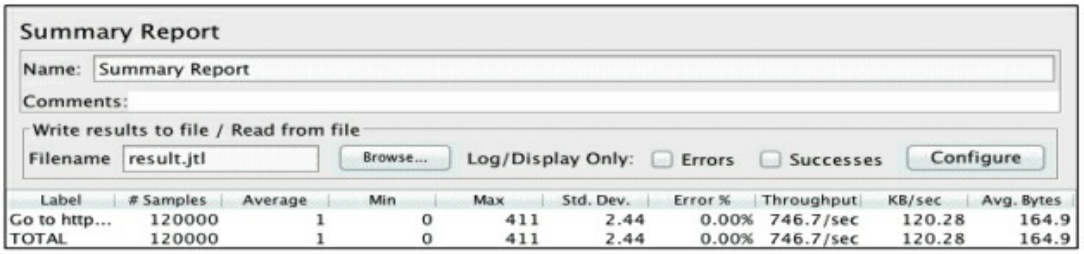
首先我们需要生成一个新的测试计划，称为Analyze Results，我们将在此父元素下添加若干侦听器元素。之后，将添加各种可以用于分析性能结果的JMeter侦听器。

计算带宽

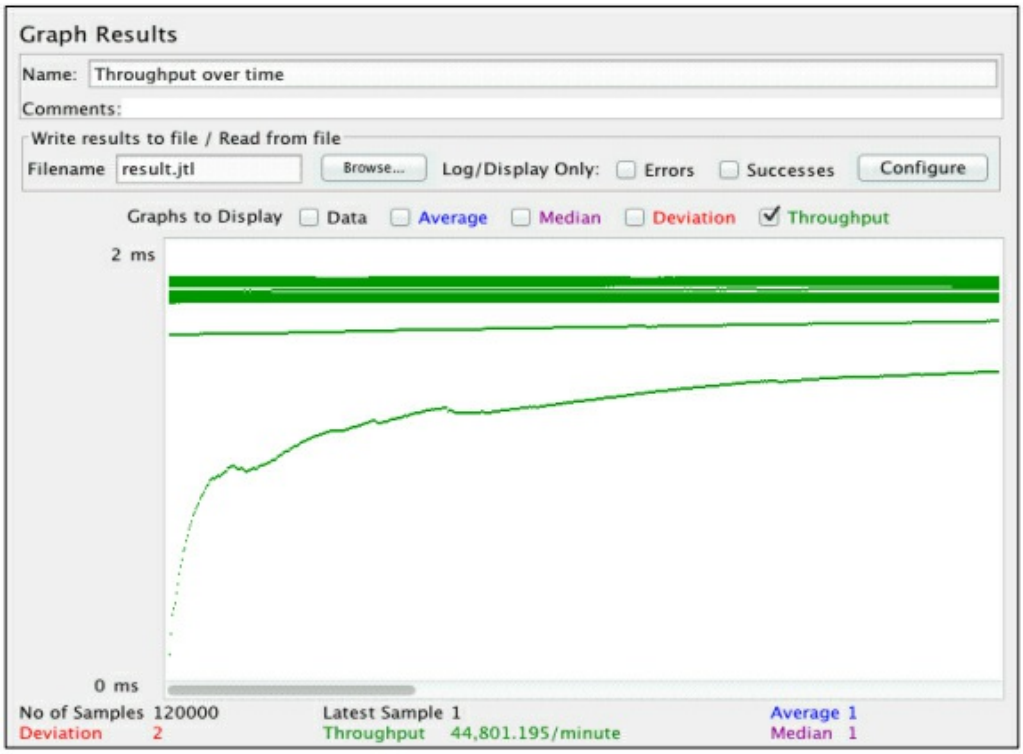
对于第一次分析，我们将使用Summary Report侦听器，此侦听器可展示应用的吞吐量。性能评测将展示应用每秒可以处理多少次交易。

要显示吞吐量内容，需要执行以下步骤。

加载侦听器后，在“Filename”域中输入result.jtl，其中有我们之前生成的测试数据。下面的屏幕截图显示了120,000个HTTP请求被送往http//webapp，吞吐量为每秒746.7次请求：



我们也可以看看吞吐量如何随着Graph Results侦听器而变化。在Analyze Results测试计划元素下生成名为“Throughput over time”的侦听器。确保只有“Throughput”复选框被标记（后续可以点选其他变量）。创建侦听器后，再次载入我们的result.jtl测试结果。下面的截图显示了吞吐量是如何随着时间而变化的：



正如我们在上图中看到的，随着JMeter在单线程池中不断加压（warm-up），吞吐量开始上升很慢。但随着测试继续运行，吞吐量

稳定在一个水平。如果调整thread group下的loop count，可以使测试时间大大缩短。

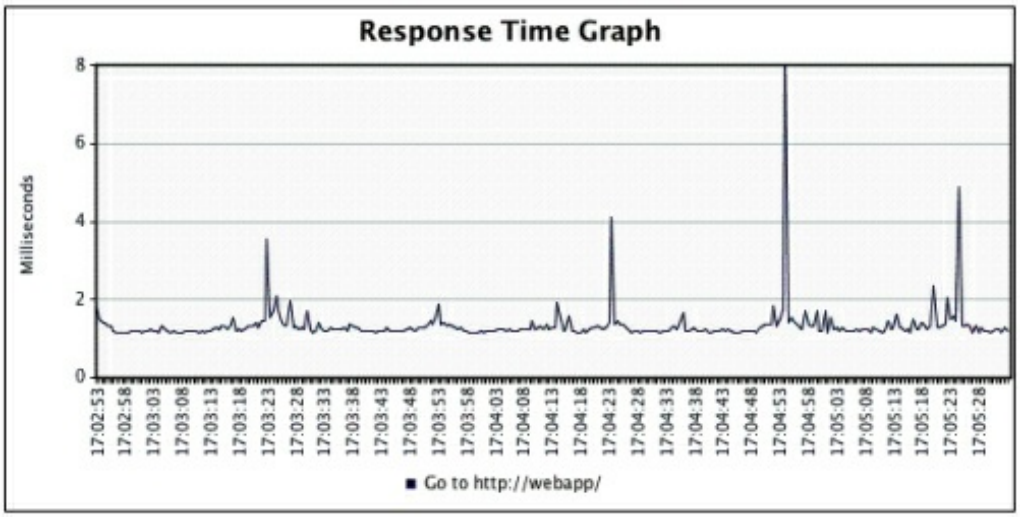
这样，在Summary Report下显示的吞吐量结果或多或少是一致的。需要注意的是，Graph Results侦听器在初始抽样后基本围绕在数据点周围。

请记住，在基准测试中，如果有更多的样本，我们会得到更精确的观察结果！

制作响应时间图

另外一个我们感兴趣的数据是响应时间（response time）。响应时间展示JMeter接收到应用响应前需要等待的时间。从用户角度可以理解为，客户敲入网址到所有内容显示在浏览器的时间（有时候并不准确，例如JavaScript画图会消耗时间，但是对于我们的实例，这种模拟应该足够精确）。

要看应用的响应时间，需要使用Response Time Graph侦听器。进行初始化设置时，可以将间隔设为500毫秒，会将每500毫秒的响应平均值记录到result.jtl中。在下图中，可以看到应用程序响应时间大多是在大约1毫秒：



如果要显示更细致的响应时间，可以将间隔减小至1毫秒。注意，这将需要更多的时间，因为JMeter的UI试图绘制更多的点。当有太多的样本时，JMeter可能会崩溃，因为我们的工作站没有足够的内存来显示整个图形。在大的基准情况下，我们能更好地观察我们的监控系统。我们将在下一节观察这些数据。

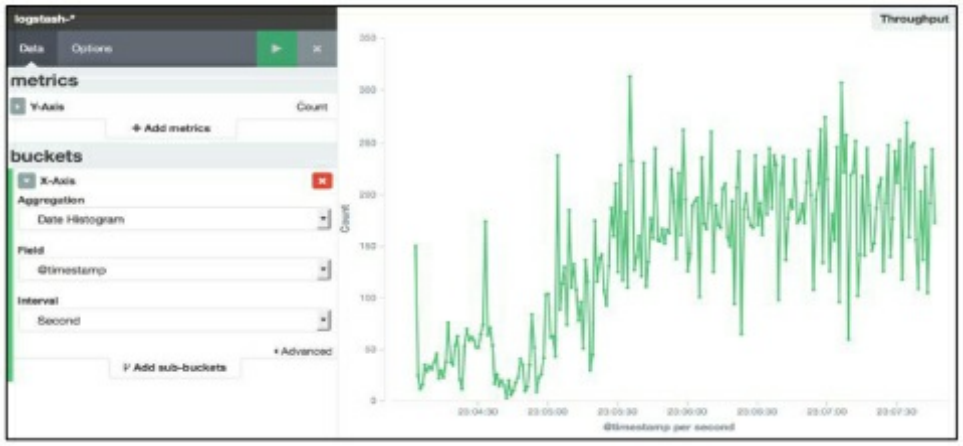
在**Graphite**和**Kibana**中观察性能

有时候，当我们的工作站太旧了，Java不能处理其JMeter的UI显示的120,000个数据点。为了解决这个问题，当画响应时间图时，可以减少数据量，例如产生少量的请求，或像我们前面做的那样，对数据取平均。不过，有时我们要看数据的全谱，这对于想查看应用更详细的行为很重要。幸运的是，我们已经拥有了在第4章中建立的监控系统。

在本节中，我们的监控和日志系统部署在被称为monitoring的

Docker宿主机中。运行应用容器的Docker宿主机webapp中，配置有collectd和Rsyslog，它们负责将事件发送到monitoring。

还记得描述性能基准时我们提到过的Nginx配置吗？标准Nginx容器中产生的访问日志被Docker获取，如果我们使用跟第4章中同样的Docker守护进程配置，这些日志事件被本地Rsyslog服务获取。Syslog日志会被转发到Logstash Syslog收集器，存放在Elasticsearch，然后使用Kibana图形化功能查看应用的响应能力。下图所示的分析就是通过计数Elasticsearch每秒收到访问日志数量生成的：



我们也可以在Kibana中画出应用的响应时间。首先需要重新配置Logstash，从接收到的访问日志中提取数据。更新第4章中的logstash.conf文件，加入grok{}过滤器，代码如下所示：

```
input {
  syslog {
    port => 1514
    type => syslog
  }
}
```

```
filter {
  if [program] == ""docker/nginx"" {
    grok {
    patterns_dir => [""/etc/logstash/patterns""]
    match => {
      ""message"" => ""%{NGINXACCESS}""
    }
  }
}
}

output {
  elasticsearch {
    host => ""elasticsearch""
  }
}
```

Logstash的过滤器插件用于数据真正存放到Elasticsearch前就过滤事件，将原始数据转化为JSON格式的富文本数据，以便于后续分析。更多关于Logstash过滤器的插件信息，可以从如下网址获得：https://www.elastic.co/guide/en/logstash/current/filter-plugins.html。

grok{}过滤器中定义的NGINXACCESS模式被称为Patterns文件，其内容如下所示：

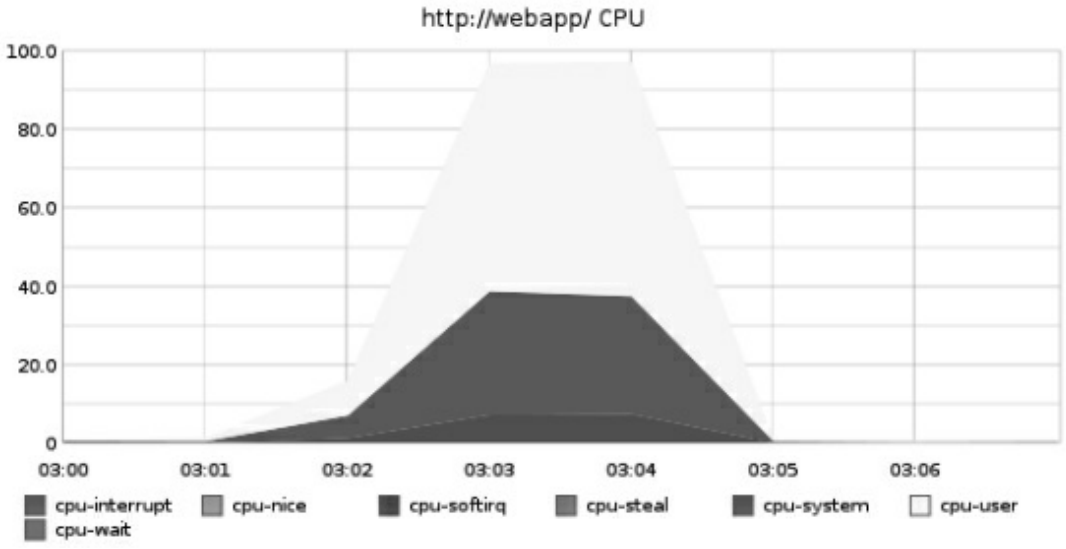
```
REQUESTPATH \""%{WORD:method} %{URIPATHPARAM} HTTP.*\""
HTTPREQUEST %{REQUESTPATH} %{NUMBER:response_code}
WEBMETRICS %{NUMBER:bytes_sent:int} %{NUMBER:response_time
NGINXSOURCE %{IP:client} \[%{HTTPDATE:requested_at}\]
NGINXACCESS %{NGINXSOURCE} %{HTTPREQUEST} %{WEBMETRICS}
```

最后，如第4章中所示的步骤，重新生成hubuser/logstash Docker容器。还需要更新Dockerfile，代码如下所示：

```
FROM logstash:1.5.3

ADD logstash.conf /etc/logstash.conf
ADD patterns /etc/logstash/patterns/nginx
EXPOSE 1514/udp
EXPOSE 25826/udp
```

现在我们可以从Nginx访问日志中提取响应时间，可以在Kibana中把这些数据画出来。下图就是Kibana中展示的每秒平均响应时间的示例：

另外可以演示的结果就是，Docker宿主机webapp对不同性能基准测试压力的响应图。首先，可以查看Web应用对Docker宿主机CPU的消耗情况。我们可以登入graphite-web仪表盘，画出webapp.cpu-0.cpu-*（排除cpu-idle数据）的图像。可以看出，Docker宿主机CPU在收到大量请求时消耗达到100%，如下图所示。



也可以通过同样的方式查看其他系统指标在HTTP请求压力下的表现。最重要的是，我们可以使用这些数据，然后分析出Web应用与它们之间的关系。

Apache JMeter 2.13与之后版本，包括一个后台侦听器，会将JMeter接收到的数据实时发送到外部服务点。默认的，会支持Graphite wire协议。可以用这个支持将性能基准数据发送到第4章中搭建的Graphite监控架构进行分析。更多数据可以参见如下网址：http://jmeter.apache.org/usermanual/realtime-results.html。

性能调优

现在，我们已经有了在Apache JMeter中生成测试计划和分析基准结果的基本工作流。还有其他一些参数可以调整，这一节，我们重新审视测试计划，来看看Docker应用的限制在哪里。

增加并发

第一个需要调整的参数是loop　count。增加更多并发请求可以使我们看出应用对负载的响应情况。这会增加性能基准的精度，因为网络连接和硬件失效（除非特意先排除了这种可能性）都可能影响我们的测试。

当我们搜集到足够多的数据后，开始意识到产生的负载并不足以压测Docker应用。例如，从我们的第一个分析中看到的吞吐量并不足以模拟正式的用户行为，如果想每秒有2000个请求，为了增加JMeter产生的压力，可以增加thread group中的线程数量，这会增加JMeter每次生成的并发请求数量。如果想模拟渐次增加用户数量，可以将ramp-up时段变长一些。

有时候想瞬时产生大量用户访问，可以设置ramp-up时间为0，这样所有线程立刻开始工作。有时候想调整其他行为，比如稳定负载后突然出现一个峰值，可以使用Stepping Thread Group

插件。

对于限制每秒100次的访问场景，可以使用Timer元素控制若干线程产生请求。为了限制吞吐量，可以使用Constant Throughput Timer，此参数使JMeter自动限制Web应用的吞吐量。

某些性能测试技术并不能跟Apache JMeter很顺畅地集成起来，但是有不少可用的插件可以很简单地产生负载，对应用进行压测。Apache JMeter列出常用插件的网址如下：http://jmeter-plugins.org。

运行分布式测试

调整完并发参数后，看起来结果并没有变化。我们可以设置JMeter每秒产生10,000个请求，但是很可能使UI崩溃，因为此时达到了工作站极限。我们需要建立一个JMeter服务器池，使用JMeter产生分布式压测。分布式压测很有用，因为可以从云端产生高性能负载来模拟峰值。对于生成来自不同源的压力也很有用。分布式测试对于模拟高延迟场景很有用，可以模拟用户从世界各地访问我们的Docker应用。

执行如下命令将JMeter部署在多台服务器上，模拟分布式性能基准测试：

1．首先，创建hubuser/jmeter的Dockerfile：

```
FROM java:8u66-jre

# Download URL for JMeter
RUN curl http://www.apache.org/dist/jmeter/binaries/ap
```

```
jmeter-2.13.tgz | tar xz
WORKDIR /apache-jmeter-2.13

EXPOSE 1099
EXPOSE 1100

ENTRYPOINT ["./bin/jmeter", "-j", "/dev/stdout", "-s",
            "-Dserver_port=1099", "-Jserver.rmi.localp
```

2．其次，部署足够数量的分布式服务器，记下每台Docker宿主机的主机名或者IP地址。本例使用两个Docker主机：dockerhost1和dockerhost2：

3．在Dokcer主机上运行JMeter，登入每一台主机并键入如下命令：

```
dockerhost1$ docker run -p 1099:1099 -p 1100:1100 \
        hubuser/jmeter -Djava.rmi.server.hostname=doc
dockerhost2$ docker run -p 1099:1099 -p 1100:1100 \
        hubuser/jmeter -Djava.rmi.server.hostname=doc
```

4．运行JMeter客户端，连接JMeter服务端：

```
$ jmeter -Jremote_hosts=dockerhost1,dockerhost2
```

设置好JMeter集群后，运行分布式测试。注意，测试计划中线程数量定义了每个JMeter服务器上的线程数。在前几节中的测试计划中，JMeter性能基准会产生240,000个请求，我们会根据测试压力来调整此参数。前几节中有些指南可以指导用户进行远程测试。

最后，开始远程测试过程，从Run菜单中选择Remote Start

All，dockerhost1和dockerhost2上测试计划中定义的JMeter服务会发起多个线程组。从Nginx的访问日志中可以看到不同请求源的IP地址，如下所示：

```
172.16.132.216 [14/Sep/2015:16:...] ""GET / HTTP/1.1"" 200
172.16.132.187 [14/Sep/2015:16:...] ""GET / HTTP/1.1"" 200
172.16.132.216 [14/Sep/2015:16:...] ""GET / HTTP/1.1"" 200
172.16.132.187 [14/Sep/2015:16:...] ""GET / HTTP/1.1"" 200
172.16.132.216 [14/Sep/2015:16:...] ""GET / HTTP/1.1"" 200
172.16.132.187 [14/Sep/2015:16:...] ""GET / HTTP/1.1"" 200
```

更多关于分布式和远程测试的知识可以从如下网址获得：

http://jmeter.apache.org/usermanual/remote-test.html。

其他性能基准工具

其他可供参考的性能基准测试工具可以参考如下链接。

- **Apache Bench**：http://httpd.apache.org/docs/2.4/en/programs/ab.html
- **HP Lab's Httperf**：http://www.hpl.hp.com/research/linux/httperf
- **Siege**：https://www.joedog.org/siege-home

小结

在本章中，我们创建了评测Docker应用的性能基准。使用Apache JMeter和第4章配置的监控系统，可以分析不同场景下的应用行为，可帮助我们建立应用能力上限，并可以用于更深层次的优化和扩展。

在下一章中，我们会讨论增加应用能力采用扩展架构时的负载均衡问题。

6 负载均衡

无论我们如何调优Docker应用，总会碰到性能瓶颈。使用前一章介绍的性能基准技术，可以了解到应用的能力。在不久的将来，Docker应用的用户将超过这个能力上限。总不能因为我们的Docker应用不能处理更多请求，就不让这些用户访问吧？所以，为了服务不断增长的用户，我们需要水平扩展Docker应用。

在本章中，我们将讨论如何水平扩展Docker应用，从而提高应用的处理能力。我们将会使用负载均衡器，这是各种Web扩展应用架构的关键组件。负载均衡器会将用户分发至Docker宿主机集群的不同应用上。本章包含以下内容：

- 准备Docker宿主机集群
- 使用Nginx来做负载均衡
- 水平扩展Docker应用
- 使用负载均衡器实现零停机发布

准备Docker宿主机集群

要对Docker应用进行负载均衡，首先要有一个服务器集群。在我们的架构中，需要准备一个Docker宿主机集群，在其中部署我们的应

用。一种可扩展的方式是使用配置管理工具来维护一个公共的基础配置，可以参照第3章中的介绍。

在准备好Docker宿主机集群之后，就可以准备运行应用了。在本节中，我们会扩展一个简单的NodeJS应用，下面将介绍该应用是如何运行的。

该Web应用是一个很小的NodeJS应用app.js。为了更好地体现应用是如何进行负载均衡的，我们会打印一些日志信息，来表明该应用是运行在哪一台Docker宿主机上的。app.js文件包含以下代码：

```
var http = require('http');

var server = http.createServer(function (request, response
  response.writeHead(200, {"Content-Type": "text/plain"});
  var version = "1.0.0";
  var log = {};
  log.header = 'mywebapp';
  log.name = process.env.HOSTNAME;
  log.version = version; console.log(JSON.stringify(log));
  response.end(version + " Hello World  "+ process.env.HOS
});
server.listen(8000);
```

为了部署这些应用代码，我们通过以下的Dockerfile，把它打包成一个Docker镜像hubuser/app:1.0.0：

```
FROM node:4.0.0

COPY app.js /app/app.js
```

```
EXPOSE 8000
CMD ["node", "/app/app.js"]
```

我们需要确保Docker镜像创建成功，并发布到Docker Hub中。这样一来，我们就可以很容易地部署它。使用以下命令构建镜像，并发布到Docker Hub：

```
dockerhost$ docker build -t hubuser/app:1.0.0 .
dockerhost$ docker push hubuser/app:1.0.0
```

最后一步，需要将应用部署到三个不同的Docker宿主机上：greenhost00、greenhost01和greenhost02。登录到每台宿主机上，执行以下命令：

```
greenhost00$ docker run -d -p 8000:8000 hubuser/app:1.0.0
greenhost01$ docker run -d -p 8000:8000 hubuser/app:1.0.0
greenhost02$ docker run -d -p 8000:8000 hubuser/app:1.0.0
```

更好的方法是，我们可以写一个Chef cookbook来部署Docker应用。

使用Nginx来做负载均衡

现在，已经有了一个Docker应用池可以接收请求流量，然后我们

要准备负载均衡器。在本节中，我们会简要介绍Nginx，它是一个流行的、高并发的、高性能的Web服务器。通常来说，Nginx可以被用作反向代理，来请求转发到多个Web应用，比如我们前面写的NodeJS应用。只要将Docker应用池配置为Nginx的反向代理目的地，Nginx就可以将请求负载均衡到每个应用上。

在部署负载均衡器时，我们会在Docker宿主机dockerhost上部署Nginx Docker容器。部署完成后，Nginx就开始向以greenhost*开头的Docker宿主机池转发流量。

以下是Nginx的一些简单配置。这些配置生效之后，Nginx会向之前部署的Docker应用池转发流量。将该配置保存在dockerhost主机的/root/nginx.conf中：

```
events { }

http {
  upstream app_server {
    server greenhost00:8000;
    server greenhost01:8000;
    server greenhost02:8000;
  }
  server {
    location / {
      proxy_pass http://app_server;
    }
  }
}
```

以上的Nginx配置主要包含了多条指令，每条指令都有相应的作

用。我们可以使用upstream指令来定义应用池。在该应用池中，使用server指令定义其中的服务器。资源池中的服务器通常定义为<hostname-or-ip>:<port> 这种格式。

上面提到的指令可以参考以下网址:

- upstream—http://nginx.org/en/docs/http/ngx_http_upstream_module.h
- server—http://nginx.org/en/docs/http/ngx_http_upstream_module.h
- proxy_pass—http://nginx.org/en/docs/http/ngx_http_proxy_module.html

更多介绍这些基本指令的相关材料，可以访问：http://nginx.org/en/docs/beginners_guide.html#conf_structure。

我们现在已经准备好了nginx.conf文件，可以将该配置文件部署到Nginx容器中了。可以在dockerhost主机上运行以下命令：

```
dockerhost$ docker run -p 80:80 -d --name=balancer \
        --volume=/root/nginx.conf:/etc/nginx/nginx.conf:r
```

我们的Web应用可以通过http://dockerhost访问。每条请求都会被路由至hubuser/webapp:1.0.0容器池中的其中一个。

为了确认我们的部署，可以通过Kibana来显示三个主机上的流量分布。首先，我们需要产生访问应用的流量。可以使用第5章中介绍的JMeter测试框架。为了更快地测试，也可以长时间运行类似下面的指令来产生流量：

```
$ while true; do curl http://dockerhost && sleep 0.1; done
1.0.0 Hello World  56547aceb063
1.0.0 Hello World  af272c6968f0
1.0.0 Hello World  7791edeefb8c
1.0.0 Hello World  56547aceb063
1.0.0 Hello World  af272c6968f0
1.0.0 Hello World  7791edeefb8c
1.0.0 Hello World  56547aceb063
1.0.0 Hello World  af272c6968f0
1.0.0 Hello World  7791edeefb8c
```

回忆一下，在前面准备的应用中，我们将$HOSTNAME打印到HTTP响应中。在之前的例子里，响应会显示Docker容器的主机名。注意，在默认情况下，Docker容器会把容器ID的短哈希值作为主机名。从测试流量的输出可以看出，我们得到了三个不同容器的响应。

就像第4章中那样建立日志系统，我们可以使用Kibana来更好地可视化响应。在下图中，我们可以计算出每分钟每个Docker宿主机返回的响应数目。

从上图可以看出，负载被Nginx均匀地分配到了三个Docker宿主机上：greenhost00、greenhost01和greenhost02。

为了使Kibana正确地可视化部署过程，需要标记Docker容器，并在Logstash中过滤这些日志项，从而使它们正确地被Elasticsearch处理。步骤如下：

首先，我们要确保在部署Docker容器时使用syslog-tag选项。这可以使Logstash更容易过滤容器日志。运行以下命令：

```
greenhost01$ docker run -d -p 8000:8000 \
  --log-driver syslog \
  --log-opt syslogtag=webapp \
  hubuser/app:1.0.0
```

这样一来，Logstash将会收到标记了docker/webapp标签的容器日志。然后，我们在Elasticsearch中使用Logstash过滤器来获得

这些信息：

```
filter {
  if [program] == "docker/webapp" {
    json {
      source => "message"
    }
  }
}
```

水平扩展**Docker**应用

现在，假设三台Docker宿主机上的负载开始提高。如果没有我们建立的Nginx负载均衡器，那么应用性能就会下降，这意味着服务质量会下降。对应用用户来说意味着低质量的服务，甚至半夜会被叫起来做系统调整。然而，通过负载均衡器来管理应用连接，水平扩展应用和提高应用能力就变得简单了。

因为我们的应用已经被负载均衡了，所以水平扩展过程是很简单的。以下的几个步骤是如何提高应用能力的典型工作流。

1．使用与Docker宿主机池相同的基础配置来创建新的Docker宿主机。在本节中，我们将创建两台新的Docker宿主机，分别叫作greenhost03和greenhost04。

2．在新的Docker宿主机上部署应用。使用与之前相同的命令：

```
greenhost03$ docker run -d -p 8000:8000 hubuser/app:1.0.0
greenhost04$ docker run -d -p 8000:8000 hubuser/app:1.0.0
```

3. 至此，资源池中新的应用服务器已经准备好接收连接了。现在，我们需要将它们添加到Nginx负载均衡器的目的地中。更新/root/nginx.conf文件，将它们添加到upstream服务器池中：

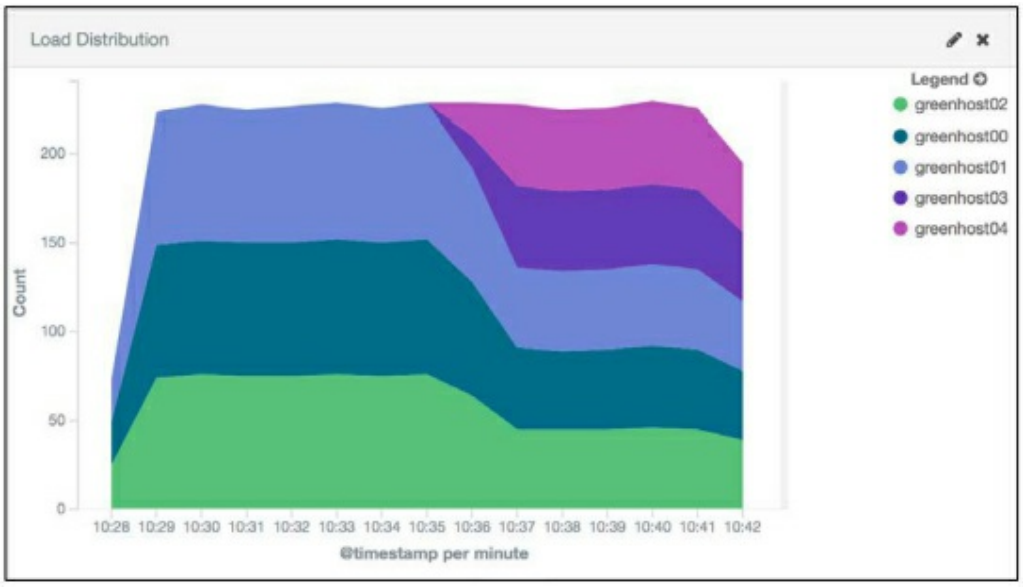
```
events { }

http {
  upstream app_server {
    server greenhost00:8000;
    server greenhost01:8000;
    server greenhost02:8000;
    server greenhost03:8000;
    server greenhost04:8000;
  }
  server {
    location / {
      proxy_pass http://app_server;
    }
  }
}
```

4. 最后，我们需要向Nginx主进程发送HUP UNIX信号，来通知Nginx容器重新加载配置文件。使用以下命令来发送重加载信号：

```
dockerhost$ docker kill -s HUP balancer
```

关于如何使用UNIX信号来控制Nginx，请参考：http://nginx.org/en/docs/control.html。

现在，我们已经完成了水平扩展Docker应用，让我们再使用Kibana可视化工具来观察一下效果。下图显示了流量是如何负载均衡到5台Docker宿主机上的：



从上图中可以看出，Nginx重载之后，它就开始将流量分发到新的Docker容器上。在此之前，每个Docker容器会接收三分之一的流量，现在每个Docker应用仅仅接收五分之一的流量。

零停机部署

使用负载均衡技术还有另一个好处，就是可使用它来更新应用。

传统上，运维工程师需要安排停机时间或维护窗口来更新生产环境中的应用。然而，流量在到达应用之前会先经过负载均衡器，我们可以利用这一点。在本节中，我们会使用blue-green部署技术来零停机地更新应用。

目前，Docker容器池hubuser/app:1.0.0被称为*green*宿主机池，因为它正在接收来自Nginx负载均衡器的请求。我们将更新后的应用部署为Docker容器池hubuser/app: 2.0.0。

请用以下步骤完成更新：

1．更新应用app.js中的版本号：

```
var http = require('http');

var server = http.createServer(function (request, resp
  response.writeHead(200, {"Content-Type": "text/plain
  var version = "2.0.0";
  var log = {};
  log.header = 'mywebapp';
  log.name = process.env.HOSTNAME;
  log.version = version;
  console.log(JSON.stringify(log));
  response.end(version + " Hello World  "+ process.env
});

server.listen(8000);
```

2．在更新了应用内容之后，我们需要准备一个新版本的Docker镜像hubuser/app:2.0.0，并发布到Docker Hub上：

```
dockerhost$ docker build -t hubuser/app:2.0.0 .
dockerhost$ docker push hubuser/app:2.0.0
```

3．然后，需要准备Docker宿主机池，分别是bluehost01、bluehost02和bluehost03，可以在云提供商中创建，也可以购买物理硬件。这个集群称为Docker宿主机池*blue*。

4．最后，我们要使用以下命令，在每个宿主机上部署新版本的Docker应用：

```
bluehost00$ docker run -d -p 8000:8000 hubuser/app:2.0
bluehost01$ docker run -d -p 8000:8000 hubuser/app:2.0
bluehost02$ docker run -d -p 8000:8000 hubuser/app:2.0
```

Docker宿主机池*blue*已经准备好了。虽然它已经上线运行，但尚未真正接收用户流量。此时，在将用户流量导向新版本应用之前，我们可以对这个集群做任何事，比如进行预检（preflight checks）和测试。

在确认Docker宿主机池*blue*可以正常工作之后，就可以将流量导向*blue*了。我们可以很方便地在配置文件/root/nginx.conf里添加blue宿主机到server列表里，从而水平扩展Docker宿主机池。

```
events { }

http {
  upstream app_server {
    server greenhost00:8000;
    server greenhost01:8000;
    server greenhost02:8000;
```

```
    server greenhost03:8000;
    server greenhost04:8000;
    server bluehost00:8000;
    server bluehost01:8000;
    server bluehost02:8000;
  }
  server {
    location / {
      proxy_pass http://app_server;
    }
  }
}
```

通过以下命令发送HUP信号，来重加载Nginx负载均衡器，激活上述配置：

```
dockerhost$ docker kill -s HUP balancer
```

此时，Nginx会把流量同时发往Docker应用的老版本（hubuser/app:1.0.0）和新版本（hubuser/app:2.0.0）。因为新版本应用承载的是应用用户的真实流量，所以我们可以完全确认新版本应用是否按预期工作。如果不能正常工作，可以删除Docker宿主机池中的bluehost*主机，向Nginx重新发送HUP信号，从而安全地回滚。

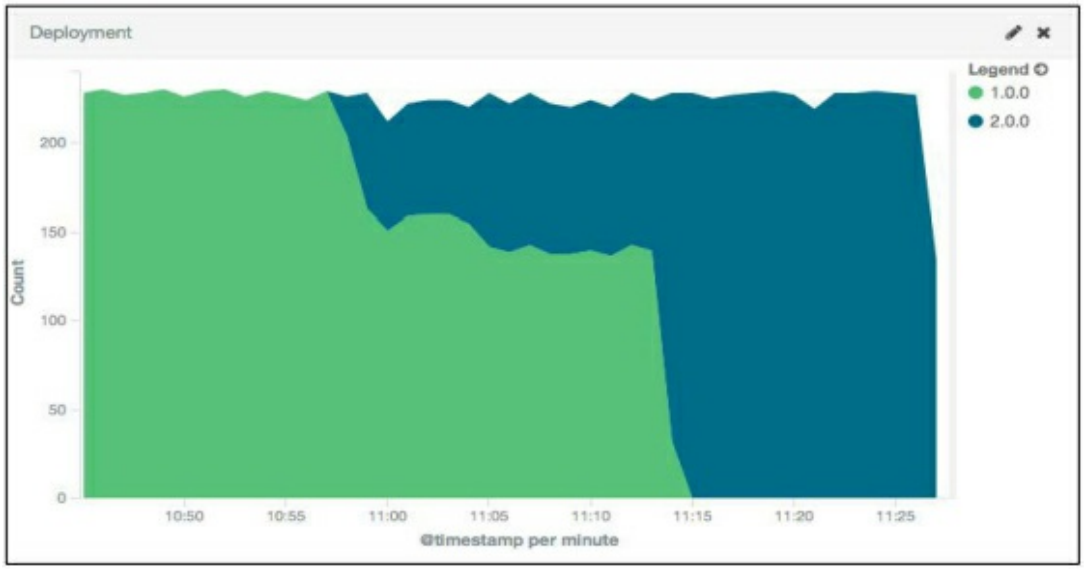
然而，假设我们已经对新版本应用满意了，那么就可以从负载均衡器的配置文件中，安全地删除老版本的Docker应用。我们需要从配置文件/root/nginx.conf中，删除所有的greenhost*服务器。

```
http {
  upstream app_server {
```

```
    server bluehost00:8000;
    server bluehost01:8000;
    server bluehost02:8000;
  }
  server {
    location / {
      proxy_pass http://app_server;
    }
  }
}
```

现在，我们只要再次向Nginx发送HUP信号，就可以完成零停机部署。此时，Docker宿主机池blue将承载应用所有的生产环境流量。因此，Docker宿主机池blue变成了新的Docker宿主机池green。为了节约资源，我们可以选择删除老的Docker宿主机池green。

完整的blue-green部署过程可以用Kibana的可视化图来总结：



注意，在上图中，即便在更新过程中，应用仍然在接收用户流量。在更新之前，所有的流量被分配到5个1.0.0版本的应用上。在激

活Docker宿主机池blue之后，3/8的流量流向2.0.0版本的应用上。最后，我们释放了老的Docker宿主机池green的终结点，所有的应用流量全部流向2.0.0版本的应用上。

更多关于blue-green部署和其他类型的零停机发布技术，可以参考Jez Humble和Dave Farley编写的*Continuous Delivery*一书。这本书可以在http://continuousdelivery.com上找到。

其他负载均衡器

其他工具也可以作为应用的负载均衡器。有的工具类似于Nginx，使用外部配置文件。我们可以发送信号来重加载更新的配置。有的工具将配置文件储存在外部数据库中，比如Redis、etcd和一些常规数据库，负载均衡器会动态地重加载配置文件。商业版的Nginx也提供了其中一些功能。还有一些开源项目，以第三方模块的方式扩展Nginx。

以下列表中的负载均衡器都可以在我们的架构中部署成Docker容器：

- Redx(https://github.com/rstudio/redx)
- HAProxy (http://www.haproxy.org)
- Apache HTTP Server (http://httpd.apache.org)

- Vulcand (http://vulcand.github.io/)
- CloudFoundry's GoRouter (https://github.com/cloudfoundry/gorouter)
- dotCloud's Hipache (https://github.com/hipache/hipache)

同时，还有一些基于硬件的负载均衡器，可以通过它们特定格式的API进行配置。如果使用云提供商的话，其中有些负载均衡器也提供相应的云API。

小结

在本章中，我们了解到了负载均衡器的好处，以及如何使用它们。我们以Docker容器的形式，部署和配置Nginx作为负载均衡器，从而水平扩展了Docker应用。同时，在更新应用的过程中，我们使用负载均衡器实现了零停机发布。

在下一章中，我们将继续学习如何调试Docker容器，提高Docker优化技术。

7 容器的故障检测和排除

有时候，仪表化并不够，比如第4章中建立的监控和日志系统。比较理想的是，我们应该以一种可扩展方式对Docker部署进行故障检测和排除。然而，有时候，我们只能登录到Docker宿主机上来查看Docker容器，除此之外，别无他法。

在本章中，我们将介绍以下主题：

- 使用docker exec检查容器
- 从Docker的外部进行调试
- 其他调试套件

检查容器

对服务器进行故障检测和排除时，传统的调试方式就是登录到机器上去查看。在Docker中，典型的工作流分为两步：首先使用标准的远程登录工具（如ssh）登录到Docker宿主机上，然后使用docker exec进入指定的、运行中的容器进程命名空间中。作为调试应用的最后一种手段，这是很有效的。

在本节的大部分例子中，我们会对运行HAProxy的Docker容器进行故障检测、排除和调试。为了准备该容器，根据以下内容创建

HAProxy的配置文件haproxy.cfg：

```
defaults
  mode http
  timeout connect 5000ms
  timeout client 50000ms
  timeout server 50000ms

frontend stats
  bind 127.0.0.1:80
  stats enable

listen http-in
  bind *:80
  server server1 www.debian.org:80
```

然后，使用官方的HAProxy Docker镜像（haproxy:1.5.14）和之前创建的配置，我们就可以运行容器了。在Docker宿主机上，运行以下命令来启动HAProxy容器：

```
dockerhost$ docker run -d -p 80:80 --name haproxy \
    -v `pwd`/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cf
    haproxy:1.5.14
```

现在，我们可以开始检查容器并调试。例如，确认HAProxy容器是否侦听在80端口上。ss程序可以在大多数Linux发行版（如Debian）上打印出套接字统计信息的小结。我们可以运行以下命令，来显示Docker容器内正在侦听的套接字的统计信息：

```
dockerhost$ docker exec haproxy /bin/ss -l
```

```
State   Recv-Q  Send-Q  Local Address:Port  Peer Address:P
LISTEN  0       128                 *:http
LISTEN  0       128         127.0.0.1:http
```

由于ss已经在haproxy:1.5.14的父容器debian:jessie中默认安装了，所以docker exec这种方法才能起作用。我们不能使用类似的工具（如netstat），因为它默认没有安装。运行netstat命令会引起以下错误：

```
dockerhost$ docker exec haproxy /usr/bin/netstat -an
dockerhost$ echo $?
255
```

我们可以查看Docker引擎服务的日志，来调查发生了什么。输入以下命令会显示，在我们的容器中netstat程序不存在：

```
dockerhost$ journalctl -u docker.service –o cat
...
time="..." level=info msg="POST /v1.20/containers/haproxy/
time="..." level=info msg="POST /v1.20/exec/c64fcf22b5c4..
time="..." level=warning msg="signal: killed"
time="..." level=error msg="Error running command in exist
       " [8] System error: exec: \"/usr/bin/netstat\":"
       " stat /usr/bin/netstat: no such file or directory"
time="..." level=error msg="Handler for POST /exec/{n.../s
time="..." level=error msg="HTTP Error" err="Cannot run ex
2015/11/18 17:58:12 http: response.WriteHeader on hijacked
2015/11/18 17:58:12 http: response.Write on hijacked conne
time="..." level=info msg="GET /v1.20/exec/c64fcf22b5c47be
...
```

另一种查看netstat是否安装的方法就是交互式地进入容器中。docker exec命令有一个选项-it，我们可以用它打开一个交互式的shell会话，进行调试。在bash中输入以下命令：

```
dockerhost$ docker exec -it haproxy /bin/bash
root@b397ffb9df13:/#
```

现在，我们在一个标准的shell环境中，使用标准的Linux工具进行调试。在下一节中，我们会陆续介绍这些命令。暂时，我们看一下为什么netstat不能工作：

```
root@b397ffb9df13:/# netstat
bash: netstat: command not found
root@b397ffb9df13:/# /usr/bin/netstat -an
bash: /usr/bin/netstat: No such file or directory
```

可以从bash的输出看到，netstat并未安装。

我们可以提供另一个方法将netstat安装到容器中，就像我们在一般的Debian环境中那样。在容器中输入以下命令来安装netstat：

```
root@b397ffb9df13:/# apt-get update
root@b397ffb9df13:/# apt-get install -y net-tools
```

现在，我们可以正常运行netstat命令了：

```
root@b397ffb9df13:/# netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address
tcp        0      0 0.0.0.0:80          0.0.0.0:*
tcp        0      0 127.0.0.1:80        0.0.0.0:*
Active UNIX domain sockets (servers and established)
```

```
Proto RefCnt Flags       Type       State         I-Node
```

并不推荐这种容器调试方法。在设计Docker架构时，应该使用正确的仪表化工具和监控工具。下次，让我们改善一下在第4章中构建的工具。

以下是用docker exec命令这个最后手段（最佳实践last-resort approach）的一些局限性：

1. 当停止并重建容器时，我们安装的netstat就又失效了。这是因为原本的HAProxy镜像并未包含该命令。在容器中临时安装包会影响Docker的主要功能，导致一个不可变的架构。

2. 如果我们希望在Docker镜像中打包所有的调试工具，那么镜像大小就会相应增大，部署也会变慢。像第2章中介绍的，我们需要优化并减小容器的大小。

3. 最小化的容器只包含必要的二进制文件，甚至连bash shell都没有。输入以下命令会发现，我们无法进入容器：

```
dockerhost$ docker exec -it minimal_image /bin/bash
dockerhost$ echo $?
255
```

总之，docker exec是一款强大的工具，可以进入容器中运行各种命令进行调试。使用-it选项，还可以通过交互式命令行进行更深入的调试。这种方法也有很多局限性，原因是我们假设所有的工具都已经默认安装在Docker容器中了。

关于docker exec命令的更多信息，可以查看官方文档：

https://docs.docker.com/reference/commandline/ exec。

下一节我们将介绍如何解决这些局限性，从Docker外部使用工具来检查运行中的容器的状态。我们将简要介绍如何使用这些工具。

从外部调试

尽管Docker隔离了容器的网络、内存、存储资源，但是每个单独的容器仍然需要Docker宿主机操作系统来执行真正的命令。我们可以利用这一点，从外部来检查和调试Docker容器。在本节中，我们会介绍其中的一些工具如何作用于Docker容器。可以在Docker宿主机或者具有一定权限（高级权限，elevated privileges）的兄弟容器（sibling container）中，查看Docker宿主机中的组件。

追踪系统调用

在服务器操作中，系统调用踪迹system call tracer是一个关键工具。它可以检查并记录被调用的系统调用，每个操作系统都有它的变形。即使我们在Docker容器中运行不同的应用和进程，归根到底都会以系统调用的形式进入Docker宿主机的Linux操作系统。

在Linux系统中，strace程序可以用来记录系统调用。strace的拦截和日志功能可以从外部检查Docker容器。容器生命周期中调用的系统调用列表可以勾画出容器是如何运行的。

输入以下命令，在装有Debian的Docker宿主机上安装strace：

```
dockerhost$ apt-get install strace
```

调用docker run的时候使用--pid=host这个选项，可以将容器的PID命名空间放到Docker宿主机中。通过这种方式，我们可以在Docker容器中安装strace，来检查Docker宿主机中的所有进程。如果我们想使用相应的基础镜像，也可以在不同的Linux发行版（比如CentOS或Ubuntu）中安装strace。

关于该选项，更多信息可以参考 http://docs.docker.com/engine/reference/run/#pid-settings-pid。

现在Docker宿主机上已经安装了strace，我们可以用它来检查HAProxy容器的系统调用了。输入以下命令来追踪HAProxy容器的系统调用：

```
dockerhost$ PID=`docker inspect -f '{{.State.Pid}}' haprox
dockerhost$ strace -p $PID
epoll_wait(3, {}, 200, 1000)  = 0
epoll_wait(3, {}, 200, 1000)  = 0
epoll_wait(3, {}, 200, 1000)  = 0
```

```
epoll_wait(3, {}, 200, 1000)  = 0
...
```

你可以看到，HAProxy容器调用了epoll_wait()等待网络连接。现在，在另一个单独的终端里，输入以下命令，向容器发送HTTP请求：

```
$ curl http://dockerhost
```

然后，我们重新回到strace程序，可以看到以下内容：

```
...
epoll_wait(3, {}, 200, 1000)  = 0
epoll_wait(3, {{EPOLLIN, {u32=5, u64=5}}}, 200, 1000) = 1
accept4(5, {sa_family=AF_INET, sin_port=htons(56470), sin_
setsockopt(6, SOL_TCP, TCP_NODELAY, [1], 4) = 0
accept4(5, 0x7ffc087a6a50, [128], SOCK_NONBLOCK) = -1 EAGA
recvfrom(6, "GET / HTTP/1.1\r\nUser-Agent: curl"..., 8192,
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 7
fcntl(7, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0
setsockopt(7, SOL_TCP, TCP_NODELAY, [1], 4) = 0
connect(7, {sa_family=AF_INET, sin_port=htons(80), sin_add
epoll_wait(3, {}, 200, 0)        = 0
sendto(7, "GET / HTTP/1.1\r\nUser-Agent: curl"..., 74, MSG
recvfrom(6, 0x18c488e, 8118, 0, 0, 0)  = -1 EAGAIN (Resour
epoll_ctl(3, EPOLL_CTL_ADD, 7, {EPOLLOUT, {u32=7, u64=7}})
epoll_ctl(3, EPOLL_CTL_ADD, 6, {EPOLLIN|EPOLLRDHUP, {u32=6
epoll_wait(3, {{EPOLLOUT, {u32=7, u64=7}}}, 200, 1000) = 1
sendto(7, "GET / HTTP/1.1\r\nUser-Agent: curl"..., 74, MSG
epoll_ctl(3, EPOLL_CTL_DEL, 7, 6bbc1c)  = 0
```

```
epoll_wait(3, {}, 200, 0)        = 0
recvfrom(7, 0x18c88d4, 8192, 0, 0, 0)  = -1 EAGAIN (Resour
epoll_ctl(3, EPOLL_CTL_ADD, 7, {EPOLLIN|EPOLLRDHUP, {u32=7
epoll_wait(3, {{EPOLLIN, {u32=7, u64=7}}}, 200, 1000) = 1
recvfrom(7, "HTTP/1.1 200 OK\r\nDate: Fri, 20 N"..., 8192,
epoll_wait(3, {}, 200, 0)                = 0
sendto(6, "HTTP/1.1 200 OK\r\nDate: Fri, 20 N"..., 742, MS
epoll_wait(3, {{EPOLLIN|EPOLLRDHUP, {u32=6, u64=6}}}, 200,
recvfrom(6, "", 8192, 0, NULL, NULL)    = 0
shutdown(6, SHUT_WR)  = 0 close(6)      = 0
setsockopt(7, SOL_SOCKET, SO_LINGER, {onoff=1, linger=0},
close(7)                                = 0
epoll_wait(3, {}, 200, 1000)            = 0
...
```

我们可以看到，HAProxy调用了标准的、BSD风格的套接字系统调用，比如accept4()、socket()和close ()，来接收、处理和关闭来自HTTP客户端的网络连接。同时，请注意，即便HAProxy是在处理连接的过程中，epoll_wait()仍然在不断被调用，从这里可以看出HAProxy是如何处理并发连接的。

追踪系统调用是非常有用的技术，可以用来调试线上的生产系统。运维人员有时候没有权限访问源码，只有生产环境中编译好的二进制文件（或者Docker镜像），没有源码（或者Dockerfile）。从运行中的应用中，可以获得的唯一线索就是应用调用的系统调用。

strace的官方网页是http://sourceforge.net/projects/strace/。更多信息可以参考该主页，也可以输入以下命令：

```
dockerhost$ man 1 strace
```

想要更全面地了解Linux系统中的系统调用，可以参考http://man7.org/linux/man-pages/man2/syscalls.2.html，这对于理解strace的输出很有帮助。

分析网络数据包

我们部署的大多数Docker容器都提供了某种形式的网络服务。在本章的HAProxy例子中，容器主要承载的就是HTTP网络流量。无论我们运行的是何种容器，网络数据包最终都会由Docker宿主机发出，从而完成请求。通过打印并分析数据包的内容，我们就可以了解Docker容器的本质。在本小节中，我们将使用数据包分析器tcpdump来观察Docker容器接收和发送的网络数据包流量。

在装有Debian的Docker宿主机中，输入以下命令来安装tcpdump：

```
dockerhost$ apt-get install -y tcpdump
```

我们也可以将Docker宿主机的网卡接口暴露给容器。通过这种方法，我们可以在容器内安装tcpdump，不会让临时安装的调

试工具污染Docker宿主机的环境。这可以通过运行docker run时添加--net=host选项。我们可以在Docker容器中，使用tcpdump访问docker0网卡接口。

下面使用tcpdump的例子是针对Vagrant VMware Fusion 7.0的。假设我们已经有一个Debian的Docker宿主机作为Vagrant VMware Fusion box，运行以下命令来暂停和恢复Docker宿主机的虚拟机：

```
$ vagrant suspend
$ vagrant up
$ vagrant ssh
dockerhost$
```

注意，我们是在Docker宿主机上运行以下命令。我们发现，在debian:jessie容器中不能解析www.google.com了：

```
dockerhost$ docker run -it debian:jessie /bin/bash
root@fce09c8c0e16:/# ping www.google.com
ping: unknown host
```

现在，我们可以在另外一个单独的终端运行tcpdump。当运行ping命令的时候，可以注意到tcpdump的输出：

```
dockerhost$ tcpdump -i docker0
tcpdump: verbose output suppressed, use -v or -vv for full
listening on docker0, link-type EN10MB (Ethernet), capture
22:03:34.512942 ARP, Request who-has 172.17.42.1 tell 172.
22:03:35.512931 ARP, Request who-has 172.17.42.1 tell 172.
22:03:38.520681 ARP, Request who-has 172.17.42.1 tell 172.
22:03:39.520099 ARP, Request who-has 172.17.42.1 tell 172.
```

```
22:03:40.520927 ARP, Request who-has 172.17.42.1 tell 172.
22:03:43.527069 ARP, Request who-has 172.17.42.1 tell 172.
```

可以看到，/bin/bash正在寻找172.17.42.1，这通常是Docker引擎的网卡接口docker0的IP地址。然后，我们输入以下命令查看一下docker0：

```
dockerhost$ ip addr show dev docker0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdi
state UP group default
    link/ether 02:42:46:66:64:b8 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::42:46ff:fe66:64b8/64 scope link
       valid_lft forever preferred_lft forever
```

现在，我们知道问题出在哪里了。网卡接口docker0没有被指定IPv4地址。有时候，当VMware恢复Docker宿主机时，docker0上映射的IP地址会被莫名其妙地删除。幸运的是，解决方案就是简单地重启Docker引擎，然后Docker会重新初始化docker0网卡接口。在Docker宿主机上，输入以下命令来重启Docker引擎：

```
dockerhost$ systemctl restart docker.service
```

现在，我们运行与之前相同的命令，可以看到IP地址已经被指定了：

```
dockerhost$ ip addr show dev docker0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 q
   link/ether 02:42:46:66:64:b8 brd ff:ff:ff:ff:ff:ff
   inet 172.17.42.1/16 scope global docker0
      valid_lft forever preferred_lft forever
   inet6 fe80::42:46ff:fe66:64b8/64 scope link
```

```
   valid_lft forever preferred_lft forever
```

我们回到最初的命令行里，可以看到问题解决了：

```
root@fce09c8c0e16:/# ping www.google.com
PING www.google.com (74.125.21.105): 56 data bytes
64 bytes from 74.125.21.105: icmp_seq=0 ttl=127 time=65.55
64 bytes from 74.125.21.105: icmp_seq=1 ttl=127 time=38.27
...
```

更多关于tcpdump数据包分析器的信息，可以参考http://www.tcpdump.org。我们也可以通过以下命令查看tcpdump的文档：

```
dockerhost$ man 8 tcpdump
```

观察块设备

Docker容器将数据存储在物理存储设备上，比如硬盘或者SSD。Docker底层的写时复制文件系统是一个随机访问的物理设备，这些设备被组织为块设备，而数据被组织为固定大小的、可以随机访问的块（blocks）。

由于Docker容器有一些特殊的I/O行为和性能问题，我们可以使用blktrace工具来追踪、检查故障和排除块设备中的问题。这个程序

可以检测到进程如何和块设备进行交互的内核事件。在本小节中，我们将设置Docker宿主机，来观察容器底层的块设备。

首先，在Docker宿主机上，输入以下命令安装blktrace程序：

```
dockerhost$ apt-get install -y blktrace
```

另外，我们需要开启文件系统的调试选项。在Docker宿主机上，输入以下命令：

```
dockerhost$ mount -t debugfs debugfs /sys/kernel/debug
```

准备好之后，我们要告诉blktrace 在哪里监听I/O事件。为了追踪容器的I/O事件，我们需要知道Docker运行时的根文件目录在哪。在Docker宿主机的默认配置里，运行时指向了/var/lib/docker。运行以下命令，来查看它属于哪个分区：

```
dockerhost$ df -h
Filesystem  Size   Used   Avail   Use%   Mounted on
/dev/dm-0   9.0G   7.6G    966M    89%   /
udev         10M      0     10M     0%   /dev
tmpfs        99M    13M     87M    13%   /run
tmpfs       248M    52K    248M     1%   /dev/shm
tmpfs       5.0M      0    5.0M     0%   /run/lock
tmpfs       248M      0    248M     0%   /sys/fs/cgroup
/dev/sda1   236M    34M    190M    15%   /boot
```

从输出可以看出，Docker宿主机的/var/lib/docker目录是在 / 这个分区下，这就是blktrace监听事件的位置。输入以下命令，开始监听该设备上的I/O事件：

```
dockerhost$ blktrace -d /dev/dm-0 -o dump
```

运行docker　run的时候加上--privileged选项，我们就可以在容器中使用blktrace命令。通过该选项，可以获得提升的权限，从而挂载被调试的文件系统。更多关于扩展容器权限的内容，可以参见https://docs.docker.com/engine/reference/　run/#runtime-privilege-linux-capabilities-and-lxc-configuration。

我们将在容器内创建空文件，直到 / 分区耗尽了空闲资源，从而在磁盘上产生I/O事件。输入以下命令产生负载：

```
dockerhost$ docker run -d --name dump  debian:jessie \
    /bin/dd if=/dev/zero of=/root/dump bs=65000
```

该命令可能很快就结束了，这取决于根分区的空闲空间。我们可以使用以下命令获取容器的PID：

```
dockerhost$ docker inspect -f '{{.State.Pid}}' dump
11099
```

现在，我们得到了产生I/O事件的Docker容器的PID，可以使用blktrace的辅助工具blkparse。blktrace程序只监听Linux内核的块I/O层，并将结果输出到文件中。blkparse程序是一个辅助工具，可以查看和分析这些事件。输入以下命令，在产生的负载中寻找Docker容器的PID对应的I/O事件：

```
dockerhost$ blkparse -i dump.blktrace.0 | grep --color " $
...
```

```
254,0  0  730   10.6267  11099  Q  R   13667072 + 24 [exe]
254,0  0  732   10.6293  11099  Q  R   5042728 + 16 [exe]
254,0  0  734   10.6299  11099  Q  R   13900768 + 152 [exe
254,0  0  736   10.6313  11099  Q  RM  4988776 + 8 [exe]
254,0  0  1090  10.671   11099  C  W   11001856 +  1024  [
254,0  0  1091  10.6712  11099  C  W   11002880 +  1024  [
254,0  0  1092  10.6712  11099  C  W   11003904 +  1024  [
254,0  0  1093  10.6712  11099  C  W   11004928 +  1024  [
254,0  0  1094  10.6713  11099  C  W   11006976 +  1024  [
254,0  0  1095  10.6714  11099  C  W   11005952 +  1024  [
254,0  0  1138  10.6930  11099  C  W   11239424 +  1024  [
254,0  0  1139  10.6931  11099  C  W   11240448 +  1024  [
...
```

从上面的输出可以看到，在块设备/dev/dm-0上偏移量为11001856的位置，有一次1024字节的写入（W），并且已经完成（C）。为了更进一步地调查，我们可以查看事件产生的偏移量位置。输入以下命令，过滤出这个偏移量位置：

```
dockerhost$ blkparse -i dump.blktrace.0 | grep 11001856
...
254,0  0  1066  10.667  8207  Q  W 11001856 + 1024 [kwork
254,0  0  1090  10.671 11099  C  W 11001856 + 1024 [0]
...
```

我们可以看到写入（W）已经被kworker进程放入了设备的队列（Q）中，也就是说，写入操作被内核加入到了队列中。在40毫秒之后，Docker容器进程的该写入请求完成了。

我们刚刚进行的调试流程是使用blktrace追踪块I/O事件的一个小

例子。我们也可以查看Docker容器中更详细的I/O行为，并找出应用的瓶颈。是不是产生了太多的写请求？大量的读请求是否需要缓存？相对于内置的docker stats产生的性能指标，这些真实的事件对于深入的故障检查和排除更有用。

更多关于blkparse的输出和blktrace的选项，可以参考用户手册http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html。

故障检测和排除工具

调试Docker容器中的应用与调试Linux中的普通应用是不同的。然而，真正使用的程序是相同的，原因是容器中发出的系统调用最终都会进入Docker宿主机的操作系统。通过了解容器产生的系统调用，我们可以使用任意的调试工具来进行故障检测和排除。

除了标准的Linux工具之外，有很多针对容器的工具集，其集合了一系列标准工具，对于容器使用更加友好。以下是工具列表：

- Red Hat的rhel-tools Docker镜像是一个巨大的容器，集成了我们之前提到的工具。https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/version-7/getting-started-with-containers/#using_the_atomic_tools_container_image 中的文档显示了如何以正确的权限使用该容器。

- CoreOS的toolbox程序是一个精简的脚本工具，可以用Systemd的systemd-nspawn程序创建一个小的Linux容器。通过复制流行的Docker容器的根文件系统，我们可以安装任意一款工具，而不会污染Docker宿主机的文件系统。更多信息可以参考https://coreos.com/os/docs/latest/install-debugging-tools.html。

- nsenter工具可以进入Linux控制组（control group）的进程空间，它是docker exec的前身，已经不维护了。如果想了解docker exec的历史，可以参考nsenter项目的主页https://github.com/jpetazzo/nsenter。

小结

请记住，登录到Docker宿主机中进行调试，并不是一个可扩展的方式。在应用层面上添加仪表化工具，可以快速、高效地诊断我们未来会碰到的问题。记住，没有人愿意半夜两点醒来，去运行tcpdump来调试Docker容器。

在下一章中，我们将讨论如何将基于Docker的工作负载用于生产环境中。

8 应用到生产环境

Docker出自于dotCloud的PaaS，以一种快速的、可扩展的方式满足IT对于开发和部署Web应用的需求，并满足日益增长的Web需求。在生产环境中，将所有应用运行在Docker容器中，不是一件简单的事情。

在本章中，我们将涉及之前学到的优化Docker的知识，并阐述如何使用它们在生产环境中运维Web应用。主要包含以下主题：

- Web运维
- 使用Docker支持应用
- 部署应用
- 扩展应用
- 关于Web操作的更多参考资料

Web运维

在互联网上维持Web应用的7Í24小时运行，对于软件开发和系统管理都提出了挑战。Docker致力于联合软件开发和系统管理，通过创建Docker镜像的方式，使得构建和部署变得统一。

然而，Docker也不是银弹。由于Web应用越来越复杂，因此，了解软件开发和系统管理中的基础概念是非常重要的。近来随着互联网技术的发展，大量的Web应用在人们的生活中无处不在，复杂性也越来越高。

为了处理Web应用维护的复杂性，我们需要掌握Web运维的细节。为了掌握这些内容，Theo Schlossnagle将其归结为4个方面：知识、工具、经验和规程。知识是指像海绵一样吸收互联网、会议和科技会议中关于Web运维的信息。理解它们，并从中过滤出有用的信息，这可以帮助我们设计生产环境中的应用架构。随着Docker和Linux容器的流行，有必要了解支持它的不同技术。在第7章中，我们已经看到常用的Linux调试工具在调试Docker容器时仍是有效的。通过了解容器是如何与Docker宿主机操作系统交互的，我们可以调试Docker中发生的问题。

第二个方面是掌握相应的工具。本书主要涉及的就是如何使用Docker，以及如何优化它的用法。在第2章中，我们了解到如何根据Docker底层的写时复制技术，来优化Docker镜像。Web运维知识告诉我们，为什么优化后的Docker镜像对于可扩展性和可部署性是很重要的。掌握如何高效地使用Docker不是一朝一夕的事情，需要在生产环境中不断实践。当然，我们可能被要求凌晨2点起来进行生产环境中的第一次Docker部署。但是，就像Schlossnagle说的那样，随着时间的流逝，在持续使用Docker的过程中获得的经验可以大大扩展我们对它的掌控能力。

通过应用知识和持续使用工具，我们获取了相关的经验，未来可能用得上。根据过去做出的错误决定，我们可以做出更好的判断。容器技术理论和Docker的运行实践可能会发生冲突。Schloassnagle提到，我们需要在Web运维中获取经验、学习如何避免错误的判断，并从中吸取经验。他建议在有限的（测试）环境中做出坏决定，这种影

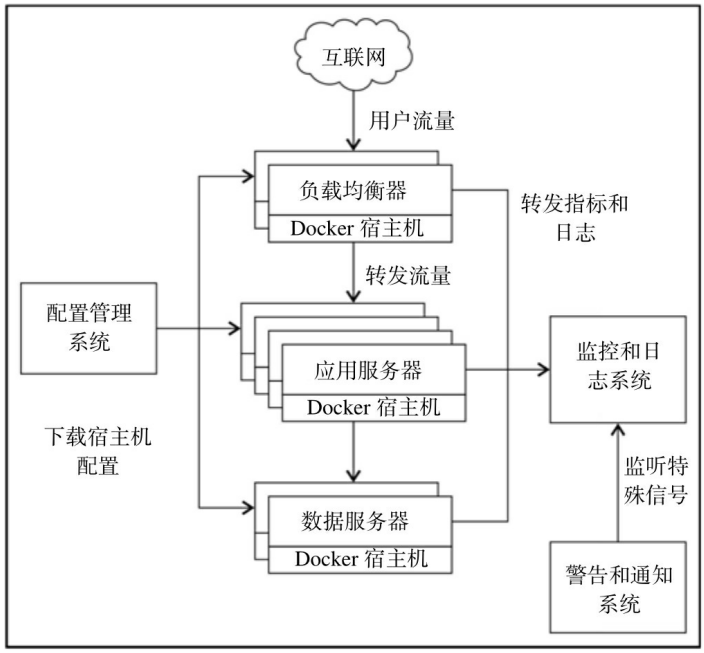
响是最小的。Docker是获取这种类型的经验的最好方式。只要有一个标准格式的、立即可用的Docker镜像，初级Web运维工程师也可以创建出他们自己的环境，并从错误中学习经验和教训。同时，当转向生产环境时，Docker环境是很类似的，因此工程师们就已经有了类似的经验。

掌握Web运维的最后一个方面是规程。然而，这些规程还未严格定义。即便是Docker，也花费了几年的时间，人们才意识到使用容器技术的最好方式。在此之前，人们普遍认为大而全的Docker镜像是比较方便的。然而，第2章中提到，减少Docker镜像的足迹（footprint）有利于管理应用的复杂性。这样一来，我们只需要考虑更少的组件，这会使调试经验变得更加简单。这些使用Docker的规程并不是读一读Docker相关的博客就可以一蹴而就的，这需要长期地学习Docker社区的知识，并在生产环境中实践如何使用Docker。

在剩下的小节里，我们将讨论Docker容器的理论和实践对于运维Web应用的帮助。

使用Docker支持Web应用

下图展示了Web应用的典型架构。负载均衡器层接收来自互联网的流量，这些流量主要包含用户请求，被转发到Web应用服务器。根据请求的特性，Web应用会从持久化存储层获取一些状态，类似于数据库服务器：



从上图中可以看到，每一层都是运行在一个Docker宿主机上的Docker容器内。对于每个组件的布局，我们可以统一使用Docker部署负载均衡器、应用和数据库，就像第2章和第6章中介绍的那样。然而，除了Docker宿主机中的Docker守护进程，我们需要以一种可扩展的方式观察和管理Web架构的全栈。在图的右边，我们可以看到，每个Docker宿主机都会发送诊断信息（如应用和系统事件、日志信息和指标）到一个集中式的日志和监控系统。我们在第4章中使用Graphite和ELK部署了这样一个系统。除此之外，可能还需要另外一个系统，用来监听日志和日志中的特定信号，并发送警报给负责基于Docker的Web应用栈维护的工程师们。这些事件与一些决定性的事件有关，比如应用的可用性和性能，我们需要采取措施来保证应用满足商业上的需求。一个内部管理系统，如Nagios或者一个第三方系统PagerDuty，可以用在我们的Docker部署中，就像第4章和第7章中介绍的。

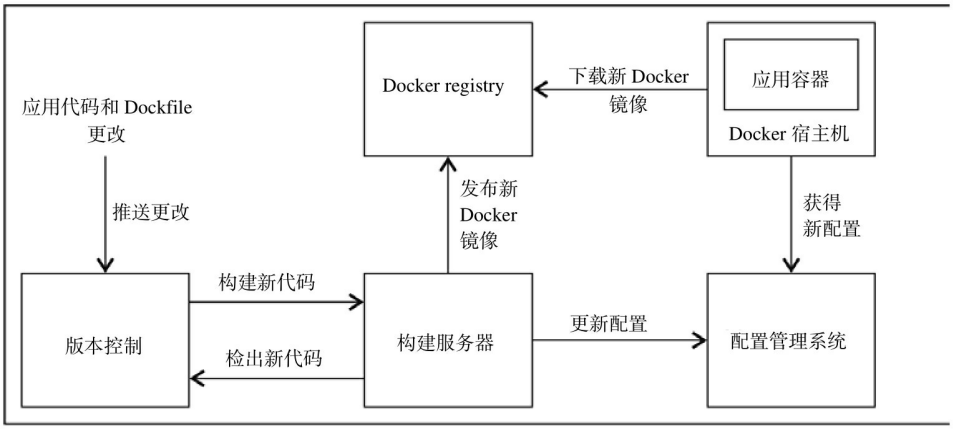
上图的左边包含了一个配置管理系统，所有Docker宿主机都会从这里下载所有需要的配置。在第3章中我们使用Chef服务器来存储Docker宿主机的配置，它包含了Docker宿主机在架构中的角色等信

息。Chef服务器保存了Docker容器如何运行的配置信息。最后，配置管理系统还会告诉Docker宿主机Graphite和Logstash监控和记录系统的终结点。

总之，除了Docker，还有各种组件来支持生产环境中的Web应用。由于容器部署的可扩展性和灵活性，Docker使我们很容易地就可以搭建整个架构。尽管如此，我们也不能跳过搭建配套基础设施的环节。在下一小节中，我们将用之前章节中学到的技巧部署Web应用的配套基础设施。

部署应用

调优Docker容器性能的一个重要组件是反馈系统，它可以告诉我们如何正确提高Web应用性能。在第4章中介绍的Graphite和ELK栈可以用可视化的方式告诉我们基于Docker的Web应用的变化。尽管收集反馈非常重要，但是及时地收集反馈更为重要。就像第3章中介绍的那样，自动配置Docker宿主机应该是一个自动的、快速的部署系统。剩余组件如下图所示：



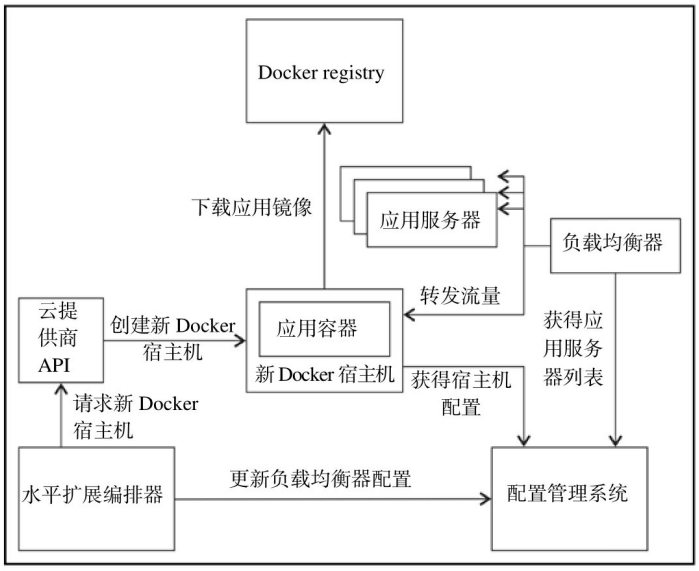
无论我们何时提交应用代码的变更，或者用Dockerfile描述如何运行和编译，都需要配套技术设施将这些变更传播到所有Docker宿主

机上。如上图所示，代码变更被提交到版本控制系统，比如Git，触发了一次新版本代码的构建。这是通过Git的postreceive钩子实现的。这些触发器会被构建服务器接收，比如Jenkins。传播代码变更的流程类似于第6章中介绍的blue-green部署流程。在接收到新代码构建的触发器之后，Jenkins将查看新版本代码，并运行docker build来创建Docker镜像。在构建成功之后，Jenkins会将新的Docker镜像推送到Docker registry中，比如Docker Hub。另外，它会更新第3章中建立的Chef服务器中的条目，从而间接地更新Docker宿主机。由于Chef服务器和Docker registry中的内容发生了变化，Docker宿主机会注意到新的配置文件，并下载、部署、运行新版本的Docker容器和Web应用。

在下一小节中，我们会讨论类似的水平扩展Docker应用的流程。

扩展应用

当我们收到如第4章中的监控系统的警报信息之后，Docker容器池不堪重负，此时就需要水平扩展它了。就像第6章中介绍的那样，我们使用负载均衡器来进行水平扩展。下图显示了第6章中的高层架构：



一旦我们决定水平扩展和添加额外的Docker宿主机，就可以通过一个水平扩展的编排器来自动化这一过程。这可以通过在构建服务器（如Jenkins）上安装一系列简单shell脚本来实现。编排器将请求云提供商API来创建一个新的Docker宿主机。该请求将创建一台Docker宿主机，并运行基本的自举（bootstrap）脚本，从配置管理系统（参见第3章）下载配置文件。Docker宿主机会从Docker registry自动下载应用的Docker镜像。在整个创建过程完成之后，水平扩展编排器将更新Chef服务器中的负载均衡器配置，添加新的应用服务器列表。以此类推，负载均衡器从Chef服务器中拉取新配置，添加新Docker宿主机，然后开始向它们分发流量。

可以看到，学习第3章的自动化设置Docker宿主机的知识对于实现第6章中介绍的可扩展的负载均衡架构是很有必要的。

更多阅读资料

使用Docker部署Web应用的配套基础设施只是冰山一角。本章中的基础概念在以下书籍中有更详细的描述。

- *Web Operations: Keeping the Data On Time*一书由J. Allspaw和J.Robbins编辑，2010年由O'Reill出版。
- *Continuous Delivery*一书由J. Humble和D. Farley编写，2010年由Addison-Wesley出版。
- *Jenkins: The Definitive Guide*一书由J. F. Smart编写，2011年由O'Reilly出版。
- *The Art of Capacity Planning: Scaling Web Resources*一书由J. Allspaw编写，2008年由O'Reilly出版。
- *Pro Git*一书由S. Chacon和B. Straub编写，2014年由Apress出版。

小结

在本书中，我们已经学习了Docker是如何工作的。除了Docker的基本使用方法之外，我们还学习了Web运维的基础概念，并帮助挖掘Docker的全部潜力。你学到了Docker和操作系统的关键概念，并对它们有了更深入的理解。另外，我们对于应用如何调用Docker宿主机操作系统的系统调用也有了了解。同时，我们还学习了很多工具，以可扩展的、可管理的方式来部署和调试生产环境中的Docker容器。

然而，你还需要在生产环境中不断地实践如何使用Docker来运行Web应用。不要害怕犯错，因为这可以获取如何在生产环境中运行Docker的最佳实践。随着Docker社区的发展，社区的实践经验也在不断增长。因此，我们应该从基础的一点一滴开始学习，勇敢地在生产环境中使用Docker！

目录