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
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

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

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

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.IPAddress }}" \
                           source
172.17.0.15
dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" \
                           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。

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=524288
```

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         VIRTUAL SIZE
hubuser/largeapp    latest  450e3123    5 minutes ago   662 MB
debian              Jessie  9a61b6b1    4 days ago      125.2 MB
```

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: 1)
450e319e42c3: Image already exists
9a61b6b1315e: Image successfully pushed
902b87aaaec9: Image successfully pushed
Digest: sha256:18ef52e36996dd583f923673618483a4466aa2d1d0d6ce
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: sha256:18ef52e36996dd583f923673618483a4466aa2d1d0d6ce
9f0...
```

```
Status: Downloaded newer image for hubuser/largeapp:latest

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: 1)
...
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:
549d099c0edaef424edb6cfca8f16f5609b066ba744638990daf3b43...
dockerhost$ time docker pull dockerhost:5000/largeapp
latest: Pulling from dockerhost:5000/largeapp
549d099c0eda: Already exists
902b87aaaec9: Already exists
9a61b6b1315e: Already exists
Digest: sha256:323bed623625b3647a6c678ee6840be23616edc357dbe07c5a0
c68b62dd52ecf
Status: Downloaded newer image for dockerhost:5000/largeapp:latest
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]
ExecStart=
ExecStart=/usr/bin/docker daemon-H fd:// \
                          --registry-mirror=http://dockerhost:5000
```

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.io \
            -e MIRROR_SOURCE_INDEX=https://index.docker.io \
            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/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 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/mirrorupstream
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`。

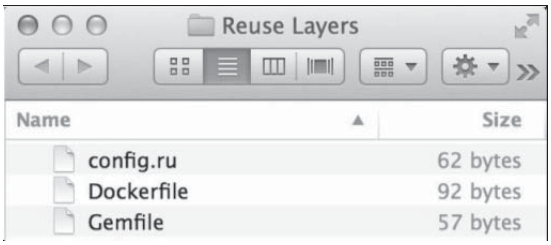
复用镜像层

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 sudo if ...
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 installed.
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 it is
needed, and
installing your bundle as root will break this application for all
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 installed. Bundled
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
```

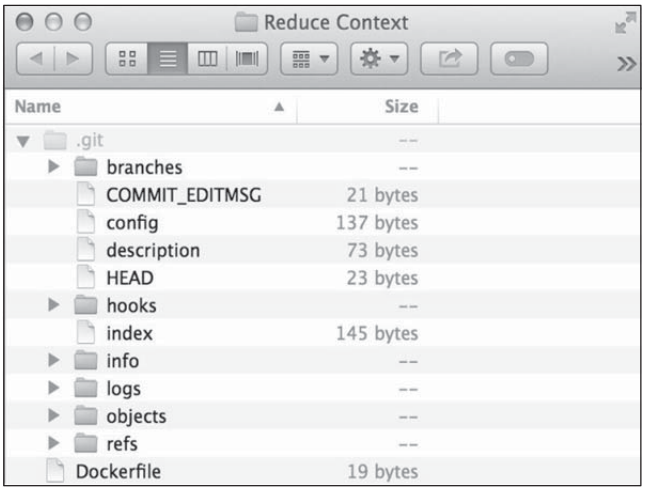
```
 ---> d763add83c94
Step 1 : ADD Gemfile /app/Gemfile
 ---> Using cache
 ---> a7f68475cf92
Step 2 : WORKDIR /app
 ---> Using cache
 ---> 203b5b800611
Step 3 : RUN bundle install
 ---> Using cache
 ---> 5c009ed03934
Step 4 : ADD . /app
 ---> 30b2bfc3f313
Removing intermediate container cd643f871828
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-cacher-ng
   ```

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                      SIZE
33c8eedfc24a   2  minutes ago   /bin/sh  -c rm -rfv /var/lib...  0 B
48b87c35b369   2  minutes ago   /bin/sh  -c apt-get install ...  318.6 MB
dad9efad9e2d   4  minutes ago   /bin/sh  -c apt-get update       9.847 MB
a8f7bf731a7d   5  minutes ago   /bin/sh  -c echo 'deb http:/...  61 B
9a61b6b1315e   6  days ago      /bin/sh  -c #(nop) CMD ["/bi...  0 B
902b87aaaec9   6  days ago      /bin/sh  -c #(nop) ADD file:...  125.2 MB
```

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.list
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：

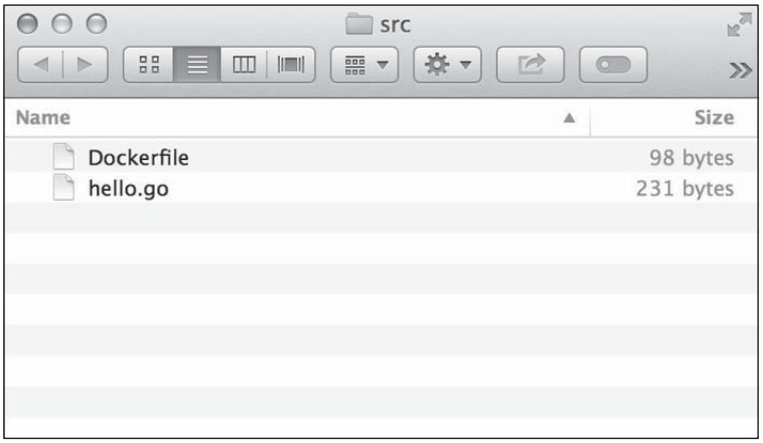
```
dockerhost$ docker build -t trueclean .
dockerhost$ docker history trueclean
IMAGE          CREATED              CREATED BY                    SIZE
 03d0b15bad7f  About a minute ago   /bin/sh   -c apt-get update...318.6 MB
 a8f7bf731a7d  9   minutes ago      /bin/sh   -c echo deb h...    61 B
```

```
9a61b6b1315e  6   days ago          /bin/sh   -c #(nop) CMD...    0 B
902b87aaaec9  6   days ago          /bin/sh   -c #(nop) ADD...    125.2    MB
```

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 SIZE
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.0
(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：

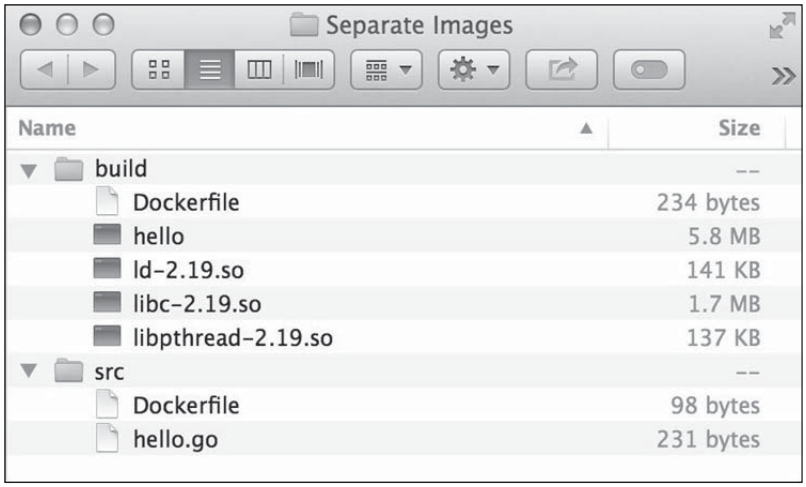
```
dockerhost$ docker cp -L large:/lib/x86_64-linux-gnu/libpthread.so.0 \
                         ../build
dockerhost$ docker cp -L large:/lib/x86_64-linux-gnu/libc.so.6 \
                          ../build
dockerhost$ docker cp -L large:/lib64/ld-linux-x86-64.so.2 \
                          ../build
```

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 SIZE
binary       latest   45c327c815  seconds ago      7.853 MB
largeapp     latest   47a64e67f   52 minutes ago   523.1 MB
golang       1.4.2    124e21271   5 days ago       517.3 MB
```

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。