

Docker中文指南

流年



目 录

[关于Docker](#)

[镜像简介](#)

[安装篇](#)

[Mac OS X](#)

[Ubuntu](#)

[Red Hat Enterprise Linux](#)

[CentOS](#)

[Debian](#)

[Gentoo](#)

[Google Cloud Platform](#)

[Rackspace Cloud](#)

[Amazon EC2](#)

[IBM Softlayer](#)

[Arch Linux](#)

[FrugalWare](#)

[Fedora](#)

[openSUSE](#)

[CRUX Linux](#)

[Microsoft Windows](#)

[Binaries](#)

[用户指南](#)

[使用Docker Hub](#)

[在Docker中运行应用](#)

[使用容器](#)

[使用docker镜像](#)

[连接容器](#)

[管理容器数据](#)

[使用Docker Hub](#)

[Docker Hub](#)

[账户](#)

[存储库](#)

[自动构建](#)

[官方案例](#)

[Docker中运行MongoDB](#)

[Docker中运行Redis服务](#)

[Docker中运行PostgreSQL](#)

[Docker中运行Riak服务](#)

[Docker中运行SSH进程服务](#)

[Docker中运行CouchDB服务](#)

Docker中运行Apt-Cacher-ng服务

关于Docker

Docker 已经出到1.6了，很多文章翻译的不对或者是已经过时，需要重新翻译和核对。从现在开始每天抽一点额外的时间来翻译文章。

联系方式

- 翻译讨论，点击 <https://docker.bearychat.com> 申请加入。
- 邮箱联系 admin#widuu.com(#换成@)

参与详细

1. 请发送邮件到 admin#widuu.com(#换成@)，说明参与 Docker 翻译。
2. 您会收到 Worktile 和 Breaychat 的 邀请链接，点击加入。
3. 在 Worktile 上领取自己要翻译的文章，开始翻译，使用 Worktile 是为了避免翻译冲突。
4. 在 Breaychat 上会提示大家每个人提交翻译的细节，并且作为团队沟通工具。

其他信息

Coding地址：https://coding.net/u/widuu/p/chinese_docker/git

gitHub地址：http://github.com/widuu/chinese_docker

blog地址：<http://www.widuu.com>

新浪微博：<http://weibo.com/widuu>

授权许可

除特别声明外，本书中的内容使用[CC BY-SA 3.0 License](#)（创作共用 署名-相同方式共享3.0许可协议）授权。

在任何地方开发、部署和运行任何应用

Docker是一款针对程序开发人员和系统管理员来开发、部署、运行应用的一款虚拟化平台。Docker可以让你像使用集装箱一样快速的组合成应用，并且可以像运输标准集装箱一样，尽可能的屏蔽代码层面的差异。Docker 会尽可能的缩短从代码测试到产品部署的时间。

Docker 组件

- The Docker Engine - Docker Engine 是一个基于虚拟化技术的轻量级并且功能强大的开源容器引擎管理工具。它可以将不同的 work flow 组合起来构建成你的应用。
- [Docker Hub](#) 可以分享和管理你的images镜像的一个 [Saas](#) 服务。

为什么选择Docker

快速交付应用程序

- 我们希望你的开发环境能够更好的提高你的工作效率。Docker容器能够帮助开发人员、系统管理员、QA和版本控制工程师在一个生产环节中一起协同工作。我们制定了一套容器标准，而这套容器标准能够使系统管理员更改容器的时候，程序员不需要关心容器的变化，而更专注自己的应用程序代码。从而隔离开了开发和管理，简化了开发和部署的成本。
- 我们使应用的构建方式更加简单，可以快速的迭代你的应用，并且可以可视化的来查看应用的细微更改。这能够帮助组织里边的成员来更好的理解一个应用从构建到运行的过程。
- Docker 是一个轻量级的容器，所以它的速度是非常快的，而容器的启动时间只需要一秒钟，从而大大的减少了开发、测试和部署的时间。

轻松部署和扩展

- Docker 容器可以运行在大多数的环境中，你可以在桌面环境、物理主机、虚拟主机再到数据中，私有或者公有云中部署。
- 因为 Docker 可以从多平台下运行。你可以很容器的迁移你的应用程序。如果需要，你可以非常简单的将应用程序从测试环境迁移到云，或者从云迁移到测试环境。
- Docker 是一个轻量级的容器，因此它可以在很短的时间内启动和关闭。当你需要的时候，你可以启动多个容器引擎，并且在不需要使用他们的时候，可以将他们全部关闭。

Get higher density and run more workloads

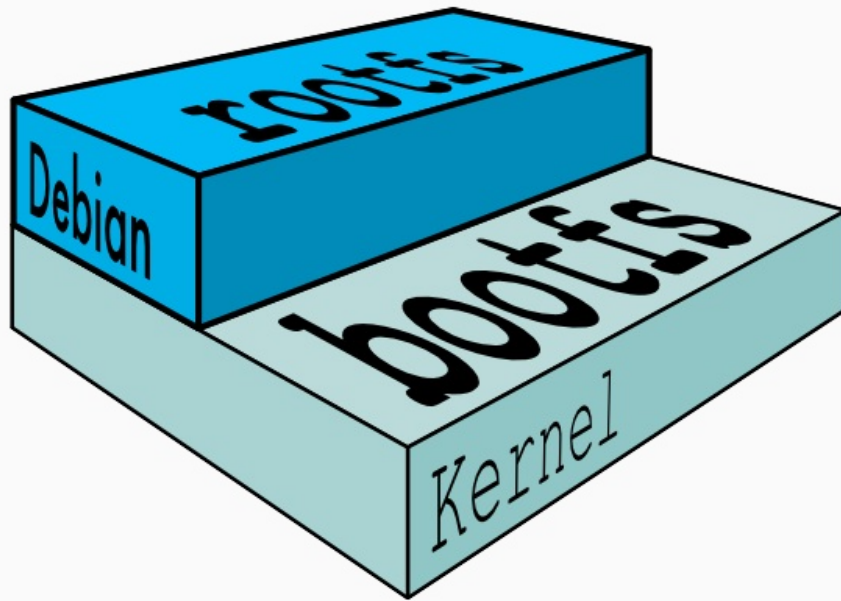
Docker的容器本身不需要额外创建虚拟机管理系统，因此你可以启动多套Docker容器，这样就可以充分发挥主机服务器的物理资源，也可以降低因为采购服务器licenses而带来的额外成本。

快速构建 轻松管理

因为Docker上述轻便，快速的特性。可以使您的应用达到快速迭代的目的。每次小的变更，马上就可以看到效果。而不用将若干个小变更积攒到一定程度再变更。每次变更一小部分其实是一种非常安全的方式。

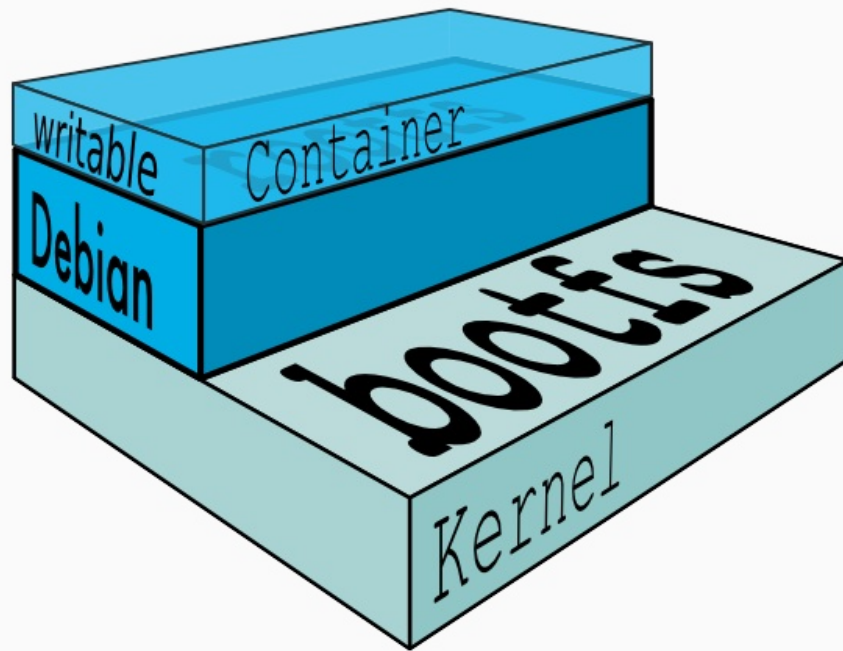
镜像简介

介绍

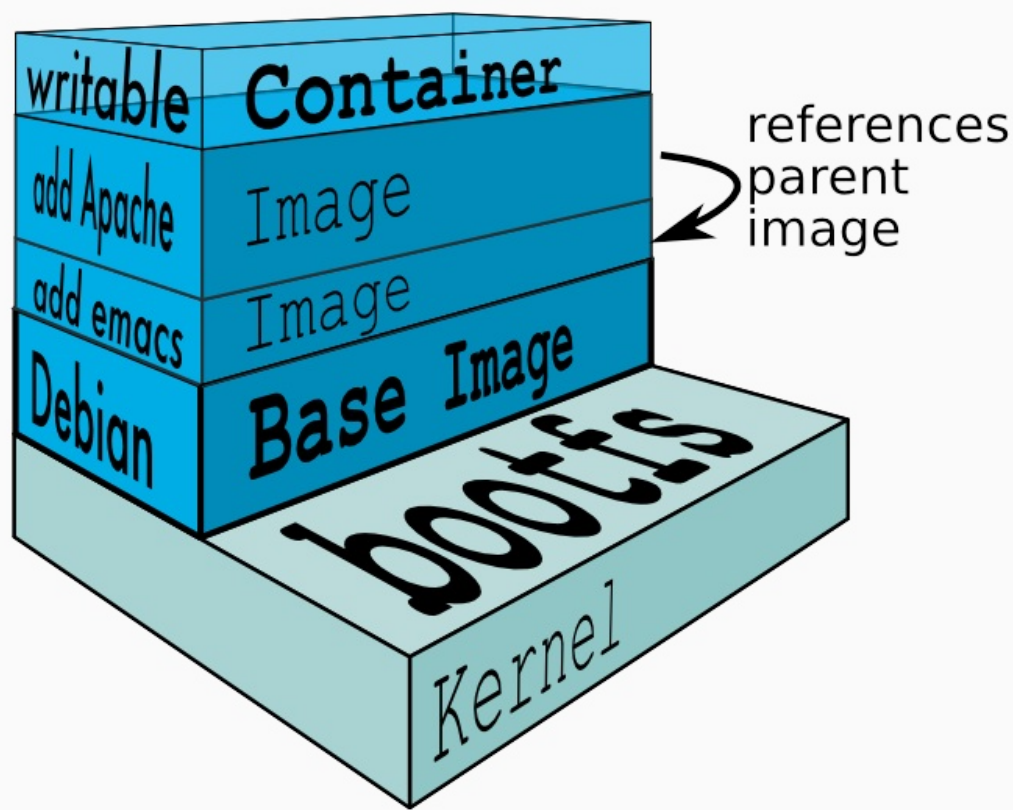


在 Docker 的术语里，一个只读层被称为镜像，一个镜像是永久不会变的。

由于 Docker 使用一个统一文件系统，Docker 进程认为整个文件系统是以读写方式挂载的。但是所有的变更都发生顶层的可写层，而下层的原始的只读镜像文件并未变化。由于镜像不可写，所以镜像是无状态的。



父镜像



每一个镜像都可能依赖于由一个或多个下层的组成的另一个镜像。我们有时说，下层那个 镜像 是上层镜像的父镜像。

基础镜像

一个没有任何父镜像的镜像，谓之基础镜像。

镜像ID

所有镜像都是通过一个 64 位十六进制字符串（内部是一个 256 bit 的值）来标识的。为简化使用，前 12 个字符可以组成一个短ID，可以在命令行中使用。短ID还是有一定的 碰撞机率，所以服务器总是返回长ID。

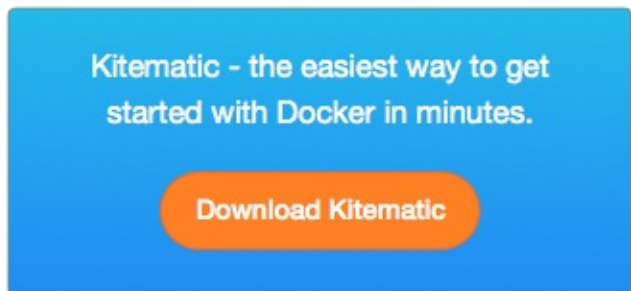
安装篇

[Mac OS X](#)
[Ubuntu](#)
[Red Hat Enterprise Linux](#)
[CentOS](#)
[Debian](#)
[Gentoo](#)
[Google Cloud Platform](#)
[Rackspace Cloud](#)
[Amazon EC2](#)
[IBM Softlayer](#)
[Arch Linux](#)
[FrugalWare](#)
[Fedora](#)
[openSUSE](#)
[CRUX Linux](#)
[Microsoft Windows](#)
[Binaries](#)

Mac OS X

你可以使用 Boot2Docker 来安装 Docker，然后在命令行运行 `docker`。如果你对命令行比较熟悉或者你打算在 Github 上贡献 Docker 项目，那么你就可以选择此安装方式。

或者，你可以使用 [Kitematic](#)，它是一款图形界面的应用程序（GUI），你可以通过图形界面来轻松地设置 Docker 和运行容器。



Command-line Docker with Boot2Docker

因为 Docker 进程使用的是 Linux 内核特性，所以你不能在原生的 OS X 中安装 Docker，如果你需要安装 Docker，你必须安装 Boot2Docker。这个程序中包含了 VirtualBox 虚拟主机(VM), Docker 和 Boot2Docker 管理工具。

Boot2Docker 是专门为 OS X 上运行 Docker 而开发的一个轻量级的虚拟主机管理工具。当 Virtual Box 在内存中启动后，它会下载一个大约 24MB 的 ISO 文件（`boot2docker.iso`），下载完成后，大约 5S 中就会启动了。

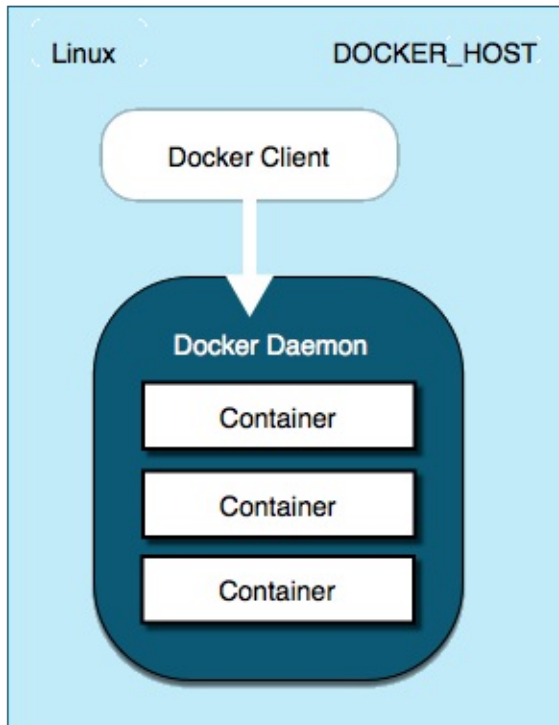
前提条件

你的 OS X 版本必须大于等于 10.6 "Snow Leopard" 才可以运行 Boot2Docker。

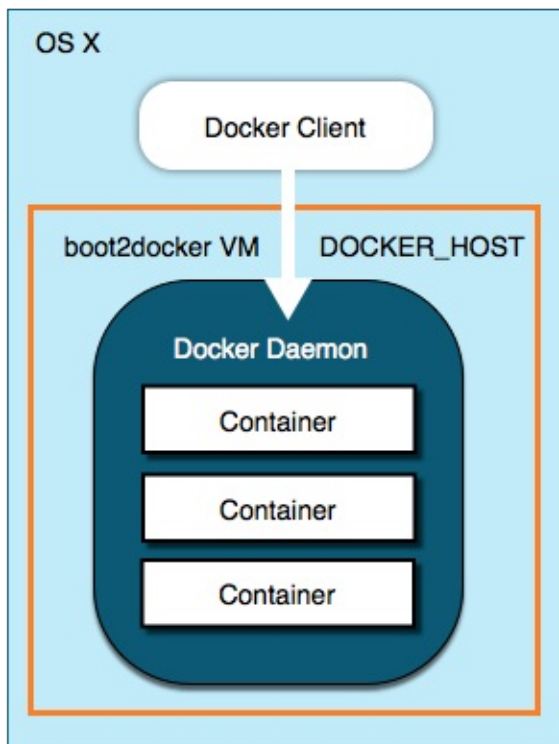
在安装之前了解一些概念

当我们在一台 Linux 主机上安装完 Docker 之后，我们的机器中就包含了本地主机和 Docker 主机。如果从网络层来划分，本地主机就代表你的电脑，而 Docker 主机就代表你运行的容器。

在一个典型的 Linux 主机上安装 Docker 客户端，运行 Docker daemon，并且在本地主机上直接运行一些容器。这就意味着你可以为 Docker 容器指定本地主机端口，例如 `localhost:8000` 或者 `0.0.0.0:8376`。



在 OS X 上安装的 Docker，`docker` 进程是通过 Boot2Docker 在 Linux 虚拟主机上运行的。



在 OS X 中，Docker 主机地址就是 Linux 虚拟主机地址。当你启动 `boot2docker` 进程的时候，虚拟
本文档使用 [看云](#) 构建

主机就会为它指定IP。在 `boot2docker` 下运行的容器，通过端口映射的方式将端口映射到虚拟主机上。你可以通过本页面上的操作实践来体会到这一点。

安装Docker

1. 点击进入[boot2docker/osx-installer release](#)页面。
2. 在下载页面中点击 `Boot2Docker-x.x.x.pkg` 来下载 Boot2Docker。
3. 双击安装包来安装 Boot2Docker

将 Boot2Docker 放到你的 "应用程序 (Applications)" 文件夹

安装程序会将 `docker` 和 `boot2docker` 二进制包放到 `/usr/local/bin` 文件夹下。

启动 Boot2Docker 程序

想要运行一个 Docker 容器，首先，你需要先启动 `boot2docker` 虚拟机，然后使用 `docker` 命令来加载、运行、管理容器。你可以从你的应用程序文件夹双击启动 `boot2docker`，或者使用命令行来启动。

提示：Boot2Docker 是被作为开发工具而设计的，不适用于生产环境中。

应用程序文件夹

当你从你的“应用程序文件夹(Applications)”来启动 "Boot2Docker" 程序, 程序会做如下事项：

- 打开一个命令行控制台。
- 创建 `$HOME/.boot2docker` 目录
- 创建 VirtualBox ISO 虚拟机 和 证书 (ssh key)
- 启动 VirtualBox 并运行 `docker` 进程

到这里就启动完毕了，你可以运行 `docker` 命令。你可以运行 `hello-word` 容器来验证你是否安装成功。

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

To generate this message, Docker took the following steps:
1\ The Docker client contacted the Docker daemon.
2\ The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (Assuming it was not already locally available.)
3\ The Docker daemon created a new container from that image which runs the
```

```
executable that produces the output you are currently reading.
4\.. The Docker daemon streamed that output to the Docker client, which sent it
to your terminal.
```

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

For more examples and ideas, visit:
<http://docs.docker.com/userguide/>

你可以使用命令行来启动和关闭 `boot2docker` 。

使用命令行

使用命令行来初始化和运行 `boot2docker` ，有如下步骤：

1. 创建一个新的 Boot2Docker 虚拟机

```
$ boot2docker init
```

这会创建一个新的虚拟主机，你只需要运行一次这个命令就可以了，以后就不需要了。

2. 启动 `boot2docker` 虚拟机。

```
$ boot2docker start
```

3. 通过 docker 客户端来查看环境变量

```
$ boot2docker shellinit
Writing /Users/mary/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/mary/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/mary/.boot2docker/certs/boot2docker-vm/key.pem
export DOCKER_HOST=tcp://192.168.59.103:2376
export DOCKER_CERT_PATH=/Users/mary/.boot2docker/certs/boot2docker-vm
export DOCKER_TLS_VERIFY=1
```

每台机器的具体路径和地址可能都不相同。

4. 使用 shell 命令来设置环境变量。

```
$ eval "$(boot2docker shellinit)"
```

5. 运行 `hello-world` 容器来验证安装。

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

Boot2Docker 基本练习

这一部分，需要你提前运行 `boot2docker` 并初始化 `docker` 客户端环境。你可以运行下边的命令来验证：

```
$ boot2docker status
$ docker version
```

本节我们通过使用 `boot2docker` 虚拟机来创建一些容器任务

容器端口访问

1. 在 Docker 主机上启动一个 Nginx 容器。

```
$ docker run -d -P --name web nginx
```

一般来说，`docker run` 命令会启动一个容器，运行这个容器，然后退出。`-d` 标识可以让容器在 `docker run` 命令完成之后继续在后台运行。`-P` 标识会将容器的端口暴露给主机，这样你就可以从你的 MAC 上访问它。

2. 使用 `docker ps` 命令来查看你运行的容器

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED	STATUS
5fb65ff765e9	nginx:latest	"nginx -g 'daemon of	web	3 minutes ago	Up 3 minutes
0.0.0.0:49156->443/tcp, 0.0.0.0:49157->80/tcp					

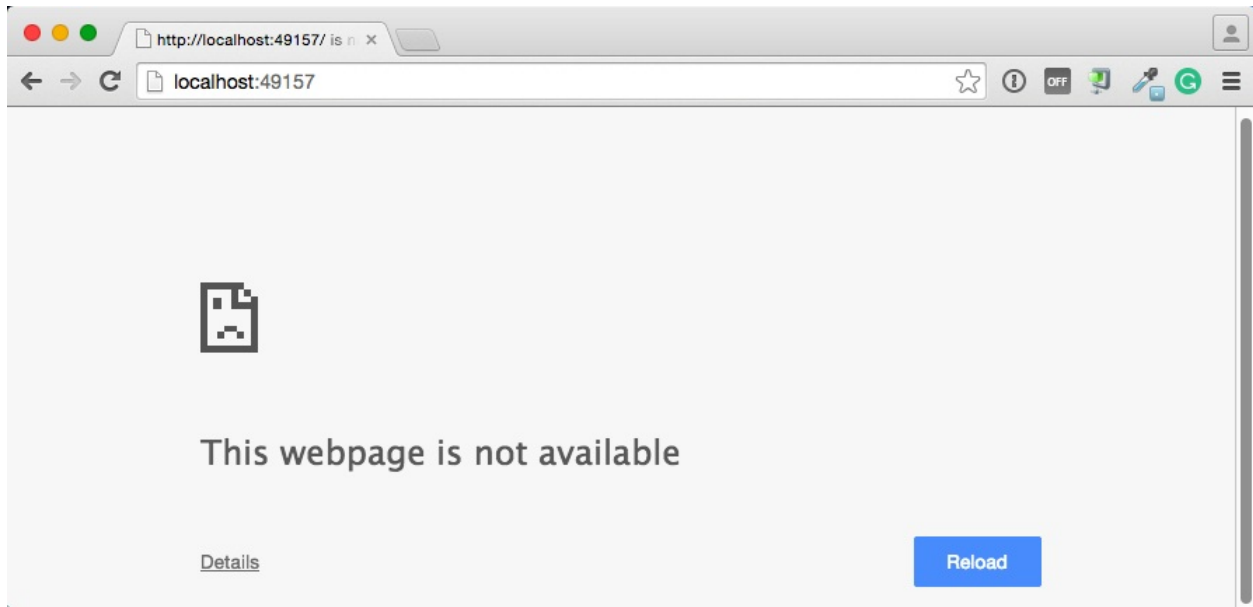
通过这一点我们可以看出 `nginx` 作为一个进程运行。

3. 查看容器端口

```
$ docker port web
443/tcp -> 0.0.0.0:49156
80/tcp -> 0.0.0.0:49157
```

上边的显示告诉我们，`web` 容器将 80 端口映射到 Docker 主机的 49157 端口上。

4. 在浏览器输入地址 `http://localhost:49157` (localhost 是 0.0.0.0):

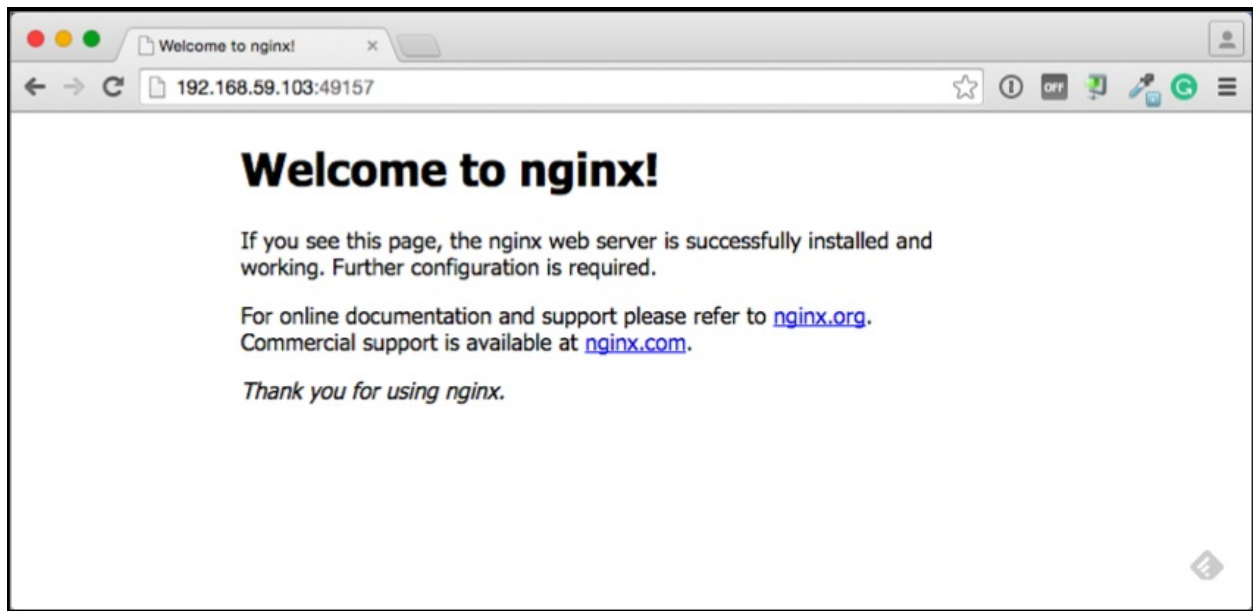


没有正常工作。没有正常工作的原因是 `DOCKER_HOST` 主机的地址不是 localhost (0.0.0.0), 但是你可以使用 `boot2docker` 虚拟机的IP地址来访问。

1. 获取 boot2docker 主机地址

```
$ boot2docker ip  
192.168.59.103
```

2. 在浏览器中输入 `http://192.168.59.103:49157`



成功运行！

1. 通过如下方法，停止并删除 `nginx` 容器。

```
$ docker stop web  
$ docker rm web
```

给容器挂载一个卷

当你启动 `boot2docker` 的时候，它会自动共享 `/Users` 目录给虚拟机。你可以利用这一点，将本地目录挂载到容器中。这个练习中我们将告诉你如何进行操作。

1. 回到你的 `$HOME` 目录

```
$ cd $HOME
```

2. 创建一个新目录，并命名为 `site`

```
$ mkdir site
```

3. 进入 `site` 目录。

```
$ cd site
```

4. 创建一个 `index.html` 文件。

```
$ echo "my new site" > index.html
```

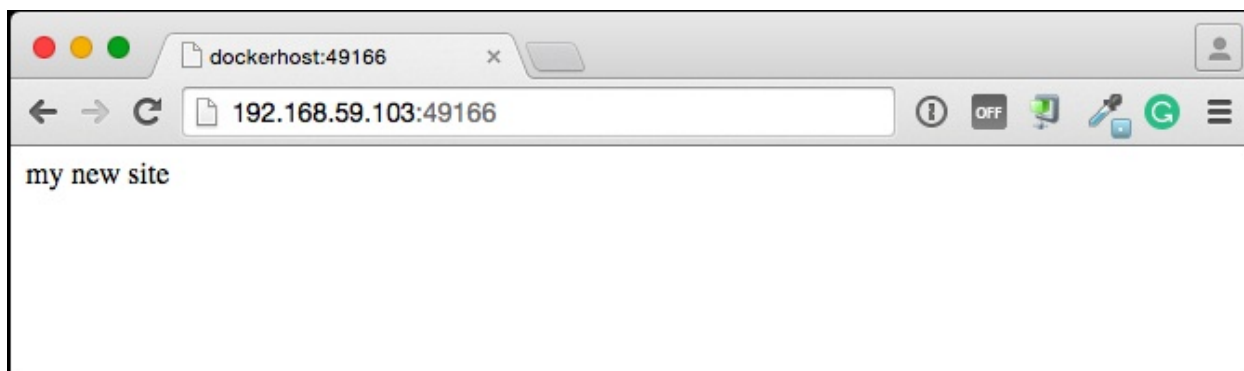
5. 启动一个新的 `nginx` 容器,并将本地的 `site` 目录替换容器中的 `html` 文件夹。

```
$ docker run -d -P -v $HOME/site:/usr/share/nginx/html --name mysite nginx
```

6. 获取 `mysite` 容器端口

```
$ docker port mysite  
80/tcp -> 0.0.0.0:49166  
443/tcp -> 0.0.0.0:49165
```

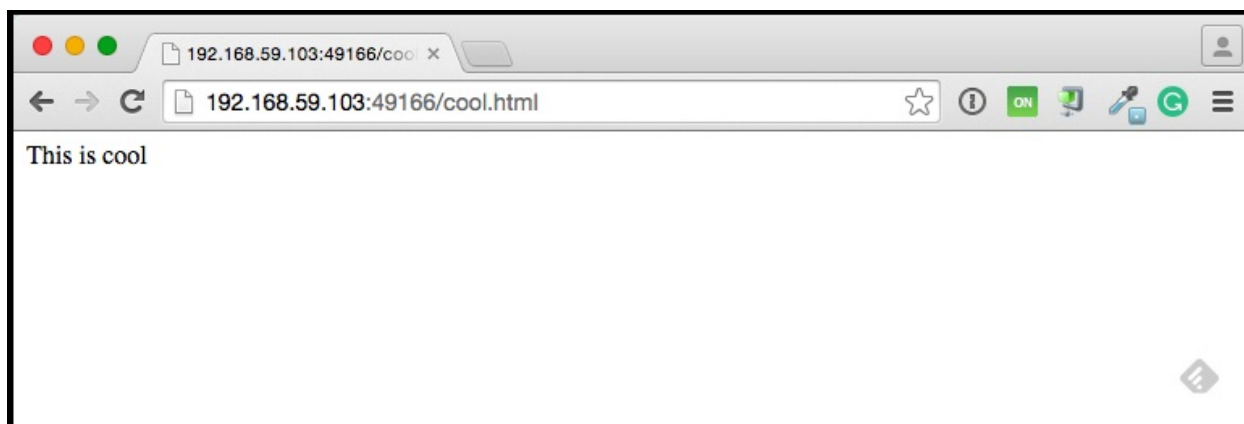
7. 在浏览器中打开站点。



1. 现在尝试在 `$HOME/site` 中创建一个页面

```
$ echo "This is cool" > cool.html
```

2. 在浏览器中打开新创建的页面。



1. 停止并删除 `mysite` 容器。

```
~~~  
$ docker stop mysite  
$ docker rm mysite  
~~~
```

升级 Boot2Docker

如果你现在运行的是1.4.1及以上版本 Boot2Docker，你可以使用命令行来升级 Boot2Docker。如果你运行的是老版本，你需要使用 `boot2docker` 仓库提供的包来升级。

命令行操作

你可以参照下边的操作来升级1.4.1以上版本：

1. 在你的机器中打开命令行。

2. 停止 `boot2docker` 应用。

```
$ boot2docker stop
```

3. 执行升级命令。

```
$ boot2docker upgrade
```

安装包方式升级

下边的操作可以升级任何版本的 Boot2Docker:

1. 在你的机器中打开命令行。
2. 停止 `boot2docker` 应用。

```
$ boot2docker stop
```

3. 打开 [boot2docker/osx-installer](#) 发布页。
4. 在下载部分点击 `Boot2Docker-x.x.x.pkg` 来下载 Boot2Docker。
5. 双击安装 Boot2Docker 包。

将 Boot2Docker 拖放到应用程序文件夹。

学习更多的知识

使用 `boot2docker help` 列出完整的命令行参考列表。更多关于使用 SSH 或 SCP 来访问 Boot2Docker 虚拟机的文档，请查看 [Boot2Docker repository](#)。

Ubuntu

Docker 支持以下的 Ubuntu 版本

- Ubuntu Trusty 14.04 (LTS) (64-bit)
- Ubuntu Precise 12.04 (LTS) (64-bit)
- Ubuntu Raring 13.04 and Saucy 13.10 (64 bit)

这个页面可以指导你安装 Docker 包管理器，并了解其中的安装机制。通过下边的安装方式可以确保你获取的是最新版本的 Docker。如果你想要使用 'Ubuntu包管理器' 安装，你可以查阅你的 Ubuntu 文档。

前提条件

Docker 需要在64位版本的Ubuntu上安装。此外，你还需要保证你的 Ubuntu 内核的最小版本不低于 3.10，其中3.10 小版本和更新维护版也是可以使用的。

在低于3.10版本的内核上运行 Docker 会丢失一部分功能。在这些旧的版本上运行 Docker 会出现一些 BUG，这些BUG在一定的条件里会导致数据的丢失，或者报一些严重的错误。

打开控制台使用 `uname -r` 命令来查看你当前的内核版本。

```
$ uname -r
3.11.0-15-generic
```

Docker 要求 Ubuntu 系统的内核版本高于 3.10，查看本页面的前提条件来验证你的Ubuntu版本是否支持 Docker。

Trusty 14.04

这个版本不需要考虑前提条件

Precise 12.04 (LTS)

对于Ubuntu Precise版本, 安装Docker需要内核在3.13及以上版本。如果你的内核版本低于3.13你需要升级你的内核。通过下边的表，请查阅下边的表来确认你的环境需要哪些包。

| linux-image-generic-lts-trusty | Generic Linux kernel image. This kernel has AUFS built in. This is required to run Docker. |

| linux-headers-generic-lts-trusty | Allows packages such as ZFS and VirtualBox guest additions which depend on them. If you didn't install the headers for your existing kernel, then you can skip these headers for the"trusty" kernel. If you're unsure, you should include this package for safety. |

| xserver-xorg-lts-trusty | Optional in non-graphical environments without

本文档使用 [看云](#) 构建

Unity/Xorg. Required when running Docker on machine with a graphical environment.

To learn more about the reasons for these packages, read the installation instructions for backported kernels, specifically the [LTS Enablement Stack](#) — refer to note 5 under each version.

```
|  
| libgl1-mesa-glx-lts-trusty |
```

通过下边的操作来升级你的内核和安装额外的包

1. 在Ubuntu系统中打开命令行控制台。
2. 升级你的包管理器

```
$ sudo apt-get update
```

3. 安装所有必须和可选的包

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

根据个人的系统环境来选择是否安装更多的包（前表列出）。

4. 重启系统

```
$ sudo reboot
```

5. 等到系统重启成功之后，查看[安装Docker](#)

Saucy 13.10 (64 bit)

Docker 使用 AUFS 作为默认的后端存储方式，如果你之前没有安装 AUFS，Docker 在安装过程中会自动添加。

Ubuntu安装Docker

首先要确认你的 Ubuntu 版本是否符合安装 Docker 的前提条件。如果没有问题，你可以通过下边的方式来安装 Docker：

1. 使用具有 `sudo` 权限的用户来登录你的Ubuntu。

2. 查看你是否安装了 wget

```
$ which wget
```

如果 `wget` 没有安装，先升级包管理器，然后再安装它。

```
$ sudo apt-get update $ sudo apt-get install wget
```

3. 获取最新版本的 Docker 安装包

```
$ wget -qO- https://get.docker.com/ | sh
```

系统会提示你输入 `sudo` 密码，输入完成之后，就会下载脚本并且安装 Docker 及依赖包。

4. 验证 Docker 是否被正确的安装

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

上边的命令会下载一个测试镜像，并在容器内运行这个镜像。

Ubuntu Docker 可选配置

这部分主要介绍了 Docker 的可选配置项，使用这些配置能够让 Docker 在 Ubuntu 上更好的工作。

- 创建 Docker 用户组
- 调整内存和交换空间(swap accounting)
- 启用防火墙的端口转发(UFW)
- 为 Docker 配置 DNS 服务

创建 Docker 用户组

docker 进程通过监听一个 Unix Socket 来替代 TCP 端口。在默认情况下，docker 的 Unix Socket 属于 `root` 用户，当然其他用户可以使用 `sudo` 方式来访问。因为这个原因，docker 进程就一直是 `root` 用户运行的。

为了在使用 `docker` 命令的时候前边不再加 `sudo`，我们需要创建一个叫 `docker` 的用户组，并且为用户组添加用户。然后在 `docker` 进程启动的时候，我们的 `docker` 群组有了 Unix Socket 的所有权，可以对 Socket 文件进行读写。

注意：`docker` 群组就相当于root用户。有关系统安全影响的细节，请查看 [Docker 进程表面攻击细节](#)

创建 `docker` 用户组并添加用户

1. 使用具有 `sudo` 权限的用户来登录你的Ubuntu。

在这过程中，我们假设你已经登录了Ubuntu。

2. 创建 `docker` 用户组并添加用户。

```
$ sudo usermod -aG docker ubuntu
```

3. 注销登录并重新登录

这里要确保你运行用户的权限。

4. 验证 `docker` 用户不使用 `sudo` 命令来执行 Docker

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

调整内存和交换空间(swap accounting)

当我们使用 Docker 运行一个镜像的时候，我们可能会看到如下的信息提示：

```
WARNING: Your kernel does not support cgroup swap limit. WARNING: Your kernel does not support swap limit capabilities. Limitation discarded..
```

为了防止以上错误信息提示的出现，我们需要在系统中启用内存和交换空间。我们需要修改系统的 GRUB (GNU GRand Unified Bootloader) 来启用内存和交换空间。开启方法如下：

1. 使用具有 `sudo` 权限的用户来登录你的Ubuntu。
2. 编辑 `/etc/default/grub` 文件
3. 设置 `GRUB_CMDLINE_LINUX` 的值如下：

```
GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

4. 保存和关闭文件

5. 更新 GRUB

```
$ sudo update-grub
```

6. 重启你的系统。

允许UFW端口转发

当你在运行 `docker` 的宿主主机上使用UFW（简单的防火墙）。你需要做一些额外的配置。Docker 使用桥接的方式来管理网络。默认情况下，UFW 过滤所有的端口转发策略。因此，当在UFW启用的情况下使用 `docker` ,你必须适当的设置UFW的端口转发策略。

默认情况下UFW是过滤掉所有的入站规则。如果其他的主机能够访问你的容器。你需要允许Docker的默认端口(2375)的所有连接。

设置 UFW 允许Docker 端口的入站规则：

1. 使用具有 `sudo` 权限的用户来登录你的Ubuntu。
2. 验证UFW的安装和启用状态

```
$ sudo ufw status
```

3. 打开和编辑 `/etc/default/ufw` 文件

```
$ sudo nano /etc/default/ufw
```

4. 设置 `DEFAULT_FORWARD_POLICY` 如下：

```
DEFAULT_FORWARD_POLICY="ACCEPT"
```

5. 保存关闭文件。
6. 重新加载UFW来使新规则生效。

```
$ sudo ufw reload
```

7. 允许 Docker 端口的入站规则

```
$ sudo ufw allow 2375/tcp
```

Docker 配置 DNS 服务

无论是Ubuntu还是Ubuntu 桌面繁衍版在系统运行的时候都是使用 `/etc/resolv.conf` 配置文件中的 `127.0.0.1`作为域名服务器(nameserver)。NetworkManager设置dnsmasq使用真实的dns服务器连接，并且设置 `/etc/resolv.conf`的域名服务为`127.0.0.1`。

在桌面环境下使用这些配置来运行 docker 容器的时候， Docker 用户会看到如下的警告：

```
WARNING: Local (127.0.0.1) DNS resolver found in resolv.conf and containers
can't use it. Using default external servers : [8.8.8.8 8.8.4.4]
```

该警告是因为 Docker 容器不能使用本地的DNS服务。相反 Docker 使用一个默认的外部域名服务器。

为了避免此警告，你可以给 Docker 容器指定一个DNS服务器。或者你可以禁用 NetworkManager 的 `dnsmasq`。不过当禁止 `dnsmasq` 可能是某些网络的DNS解析速度变慢。

为 Docker 指定一个DNS服务器

1. 使用具有 `sudo` 权限的用户来登录你的Ubuntu。
2. 打开并编辑 `/etc/default/docker`

```
$ sudo nano /etc/default/docker
```

3. 添加设置

```
DOCKER_OPTS="--dns 8.8.8.8"
```

使用`8.8.8.8`替换如`192.168.1.1`的本地DNS服务器。你可以指定多个DNS服务器，多个DNS服务器使用空格分割例如

```
--dns 8.8.8.8 --dns 192.168.1.1
```

警告:如果你正在使用的电脑需要连接到不同的网络,一定要选择一个公共DNS服务器。

4. 保存关闭文件。
5. 重启 Docker 进程

```
$ sudo restart docker
```

或者，作为替代先前的操作过程，禁止NetworkManager中的 `dnsmasq` (这样会使你的网络变慢)

1. 打开和编辑 `/etc/default/docker`

```
$ sudo nano /etc/NetworkManager/NetworkManager.conf
```

2. 注释掉 `dns = dnsmasq` :

```
dns=dnsmasq
```

3. 保存关闭文件
4. 重启NetworkManager 和 Docker

```
$ sudo restart network-manager $ sudo restart docker
```

升级Docker

在 `wget` 的时候使用 `-N` 参数来安装最新版本的Docker :

```
$ wget -N https://get.docker.com/ | sh
```

Red Hat Enterprise Linux

以下是支持 Docker 的 RHEL 版本：

- [Red Hat Enterprise Linux 7 \(64-bit\)](#)
- [Red Hat Enterprise Linux 6.5 \(64-bit\)](#) 或更高版本

内核支持

如果你的 RHEL 运行的是发行版内核。那就仅支持通过 extras 渠道或者 EPEL 包来安装 Docker。如果你打算在非发行版本的内核上运行 Docker，内核的改动可能会导致出错

Red Hat Enterprise Linux 7 installation

Red Hat Enterprise Linux 7（64位）[自带Docker](#)。你可以在[发行日志](#)中找到概述和指南。

Docker 包含在 extras 镜像源中，使用下面的方法可以安装 Docker:

1. 启用 extras 镜像源:

```
$ sudo subscription-manager repos --enable=rhel-7-server-extras-rpms
```

2. 安装 Docker：

```
$ sudo yum install docker
```

如果你是RHEL客户，更多的 RHEL-7 安装、配置和[用户指南](#)可以在[客户中心](#)中找到。

请继续阅读 [启动 Docker 进程](#)。

Red Hat Enterprise Linux 6.5 installation

你需要在 64位的 [RHEL 6.5](#) 或更高的版本上来安装 Docker，Docker 工作需要特定的内核补丁，因此 RHEL 的内核版本应为 2.6.32-431 或者更高。

Docker 已经包含在 RHEL 的 EPEL 源中。该源是 Extra Packages for Enterprise Linux (EPEL) 的一个额外包，社区中正在努力创建和维护相关镜像。

内核支持

如果你的 RHEL 运行的是发行版内核。那就仅支持通过 extras 渠道或者 EPEL 包来安装 Docker。如果你打算在非发行版本的内核上运行 Docker，内核的改动可能会导致出错

Warning: Please keep your system up to date using `yum update` and rebooting your

system. Keeping your system updated ensures critical security vulnerabilities and severe bugs (such as those found in kernel 2.6.32) are fixed.

首先，你需要安装EPEL镜像源，请查看 [EPEL installation instructions](#).

在EPEL中已经提供了 `docker-io` 包

如果你安装了(不相关)的 Docker 包，它将与 `docker-io` 冲突。在安装 `docker-io` 之前，请先卸载 Docker

下一步，我们将要在我们的主机中安装 Docker,也就是 `docker-io` 包:

```
$ sudo yum -y install docker-io
```

更新 `docker-io` 包:

```
$ sudo yum -y update docker-io
```

现在 Docker 已经安装好了，我们来启动 docker 进程:

```
$ sudo service docker start
```

设置开机启动:

```
$ sudo chkconfig docker on
```

现在，让我们确认 Docker 是否正常工作：

```
$ sudo docker run -i -t fedora /bin/bash
```

继续 [启动 Docker 进程](#)

启动 Docker 进程

现在 Docker 已经安装好了，让我们来启动 Docker 进程

```
$ sudo service docker start
```

如果我们想要开机启动 Docker ，我们需要执行如下的命令：

```
$ sudo chkconfig docker on
```

现在测试一下是否正常工作.

```
$ sudo docker run -i -t fedora /bin/bash
```

注意: 如果你运行的时候提示一个 `Cannot start container` 的错误, 错误中提到了 SELINUX 或者 权限不足。你需要更新 SELINUX 规则。你可以使用 `sudo yum upgrade selinux-policy` 然后重启。

自定义进程选项

如果你想要添加一个 HTTP 代理, 为 Docker 运行文件设置不同的目录或分区, 又或者定制一些其它的功能, 请阅读我们的系统文章, 了解[如何定制 Docker 进程](#)

问题

遇到问题请到 [Red Hat Bugzilla for docker-io component](#) 进行反馈。

CentOS

以下版本的CentOS 支持 Docker：

- [CentOS 7 \(64-bit\)](#)
- [CentOS 6.5 \(64-bit\)](#) or later

该指南可能会适用于其它的 EL6/EL7 的 Linux 发行版，譬如 Scientific Linux。但是我们没有做过任何测试。

请注意，由于 Docker 的局限性，Docker 只能运行在64位的系统中。

内核支持

目前的 CentOS 项目，仅发行版本中的内核支持 Docker。如果你打算在非发行版本的内核上运行 Docker，内核的改动可能会导致出错。

Docker 运行在 [CentOS-6.5](#) 或更高的版本的 CentOS 上，需要内核版本是 2.6.32-431 或者更高版本，因为这是允许它运行的指定内核补丁版本。

安装 - CentOS-7

Docker 软件包已经包含在默认的 CentOS-Extras 软件源里，安装命令如下：

```
$ sudo yum install docker
```

开始运行 [Docker daemon](#)。

Firewalld

CentOS-7 中介绍了 firewalld，firewall的底层是使用iptables进行数据过滤，建立在iptables之上，这可能会与 Docker 产生冲突。

当 firewalld 启动或者重启的时候，将会从 iptables 中移除 DOCKER 的规则，从而影响了 Docker 的正常工作。

当你使用的是 Systemd 的时候，firewalld 会在 Docker 之前启动，但是如果你在 Docker 启动之后再启动 或者重启 firewalld，你就需要重启 Docker 进程了。

安装 Docker - CentOS-6.5

在 CentOS-6.5 中，Docker 包含在 [Extra Packages for Enterprise Linux \(EPEL\)](#) 提供的镜像源中，该组织致力于为 RHEL 发行版创建和维护更多可用的软件包。

首先，你需要安装 EPEL 镜像源，请查看 [EPEL installation instructions](#)。

在 CentOS-6 中，一个系统自带的可执行的应用程序与 docker 包名字发生冲突，所以我们重新命名 docker 的RPM包名字为 `docker-io`。

CentOS-6 中 安装 `docker-io` 之前需要先卸载 `docker` 包。

```
$ sudo yum -y remove docker
```

下一步，安装 `docker-io` 包来为我们的主机安装 Docker。

```
$ sudo yum install docker-io
```

开始运行 [Docker daemon](#)。

手动安装最新版本的 Docker

当你使用推荐方法来安装 Docker 的时候，上述的 Docker 包可能不是最新发行版本。如果你想安装最新版本，[你可以直接安装二进制包](#)

当你使用二进制安装时，你可能想将 Docker 集成到 Systemd 的系统服务中。为了实现这一点，你需要从github中下载[service and socket](#)两个文件，然后安装到 `/etc/systemd/system` 中。

Please continue with the [Starting the Docker daemon](#).

Starting the Docker daemon

当 Docker 安装完成之后，你需要启动 docker 进程。

```
$ sudo service docker start
```

如果我们希望 Docker 默认开机启动，如下操作：

```
$ sudo chkconfig docker on
```

现在，我们来验证 Docker 是否正常工作。第一步，我们需要下载最新的 `centos` 镜像。

```
$ sudo docker pull centos
```

下一步，我们运行下边的命令来查看镜像，确认镜像是否存在：

```
$ sudo docker images centos
```

这将会输出如下的信息：

```
$ sudo docker images centos
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
centos	latest	0b443ba03958	2 hours ago	297.6 MB

运行简单的脚本来测试镜像：

```
$ sudo docker run -i -t centos /bin/bash
```

如果正常运行，你将会获得一个简单的 bash 提示，输入 `exit` 来退出。

自定义进程选项

如果你想要添加一个 HTTP 代理，为 Docker 运行文件设置不同的目录或分区，又或者定制一些其它的功能，请阅读我们的系统文章，了解[如何定制 Docker 进程](#)

Dockerfiles

CentOS 项目为开发者提供了大量的的示例镜像，作为开发模板或者学习 Docker 的实例。你可以在这里找到这些示例：

<https://github.com/CentOS/CentOS-Dockerfiles>

好！现在你可以去查看[用户指南](#)，或者创建你自己的镜像了。

发现问题？

如果有关于在 CentOS 上的 Docker 问题，请直接在这里提交：[CentOS Bug Tracker](#).

Debian

以下版本的 Debian 支持 Docker：

- [Debian 8.0 Jessie \(64-bit\)](#)
- [Debian 7.7 Wheezy \(64-bit\)](#)

Debian Jessie 8.0 (64-bit)Debian

Debian 8 使用的是 3.14.0 的内核版本，可以从 Debian 的镜像源来安装 `docker.io` 包。

提示：Debian 包含一个特别老的KDE3/GNOME2包叫 `docker`，所以我们把这个包叫 `docker.io`。

安装

安装最新版的 Debian 软件包（可能不是最新版本 Docker）

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

验证 Docker 是否正常工作：

```
$ sudo docker run -i -t Ubuntu /bin/bash
```

该命令将下载 `Ubuntu` 镜像，并且在容器内运行 `bash`。

注意：如果你打算启用内存和交换空间设置，请查看[这里](#)

Debian Wheezy/Stable 7.x (64-bit)

安装 Docker 需要内核在3.8版本以上，但是 Wheezy 的内核版本是 3.2（[bug #407](#)对 Docker 需要3.8版本内核进行了讨论。）

幸运的是，官方提供了 `wheezy-backports`，它内核版本是3.16，可以支持 Docker。

安装

1. 从 `wheezy-backports` 镜像源来安装内核

在 `/etc/apt/sources.list` 文件下添加如下内容：

```
deb http://http.debian.net/debian wheezy-backports main
```


安装 `linux-image-amd64` 包 (注意使用 `-t wheezy-backports`)

```
$ sudo apt-get update
$ sudo apt-get install -t wheezy-backports linux-image-amd64
```

2. 从 get.docker.com 获取安装脚本并安装：

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

使用非root

docker 进程通过监听一个 Unix Socket 来替代 TCP 端口。在默认情况下，docker 的 Unix Socket 属于 root 用户，当然其他用户可以使用 `sudo` 方式来访问。因为这个原因，docker 进程就一直是 root 用户运行的。

如果你（或者说你安装 Docker 的时候）创建一个叫 `docker` 的用户组，并为用户组添加用户。这时候，当 Docker 进程启动的时候，`docker` 用户组对 Unix Socket 有了读/写权限。你必须使用 root 用户来运行 docker 进程，但你可以用 `docker` 群组用户来使用 docker 客户端，你再使用 `docker` 命令的时候前边就不需要加 `sudo` 了。从 Docker 0.9 版本开始，你可以使用 `-G` 来指定用户组。

警告：Docker 用户组（或者用 `-G` 指定的用户组）有等同于 root 用户的权限，有关系统安全影响的细节，请查看[Docker 进程表面攻击细节](#)

操作演示：

```
# Add the docker group if it doesn't already exist.
$ sudo groupadd docker

# Add the connected user "${USER}" to the docker group.
# Change the user name to match your preferred user.
# You may have to logout and log back in again for
# this to take effect.
$ sudo gpasswd -a ${USER} docker

# Restart the Docker daemon.
$ sudo service docker restart
##下一步
```

Gentoo

在 Gentoo Linux 上安装 Docker 可以通过以下两种方式的任一种实现：官方安装方法和 `docker-overlay` 方法。

官方 [Gentoo Docker](#) 团队页面。

官方方式

如果你正在寻找一种稳定的方案，最好的办法就是直接在 portage tree 上安装官方的 `app-emulation/docker` 包。

如果 `ebuild` 时出现任何问题，包括缺少内核配置标识或依赖，请到 Gentoo 的 [Bugzilla](#) 网站上指定的 `docker AT gentoo DOT org` 提交问题，或者加入 Freenode 的 Gentoo 官方 [IRC](#) 频道来提问。

docker-overlay 方法

如果你正在寻找一个 `-bin` `ebuild`, `live ebuild`, 或者 `bleeding edge ebuild`，可以使用 overlay 提供的 [docker-overlay](#)。使用 `app-portage/layman` 来添加第三方的 portage。查看最新的安装和使用 overlay 的文档请，请点击 [the overlay README](#)。

如果 `ebuild` 或者生成的二进制文件时出现任何问题，包括特别是缺少内核配置标识或依赖关系，请 [在 docker-overlay 仓库提交一个 issue](#) 或者直接在 freenode 网络的 `#docker` IRC 频道上联系 tianon。

安装

Available USE flags

USE Flag	Default	Description
---	:::	:::
aufs		Enables dependencies for the "aufs" graph driver, including necessary kernel flags.
btrfs		Enables dependencies for the "btrfs" graph driver, including necessary kernel flags.
contrib	Yes	Install additional contributed scripts and components.
device-mapper	Yes	Enables dependencies for the "devicemapper" graph driver, including necessary kernel flags.
doc		Add extra documentation (API, Javadoc, etc). It is recommended to enable per package instead of globally.
lxc		Enables dependencies for the "lxc" execution driver.
vim-syntax		Pulls in related vim syntax scripts.
zsh-completion		Enable zsh completion support.

这个包会适当的获取必要的依赖和提示的内核选项。

[tianon's](#) 的博客中有详细的使用标识的介绍。

```
$ sudo emerge -av app-emulation/docker
```

注：有时候官方的 Gentoo tree 和 docker-overlay 的最新版本还是有差距的，请耐心等待，最新版本会很快更新。

启动 Docker

请确保您运行的内核包含了所有必要的模块和配置（可选的 device-mapper 和 AUFS 或 Btrfs，这主要取决于你要使用的存储驱动程序）。

使用 Docker，docker 进程必须以 root 用户运行。

用非root用户使用 Docker，可以使用下边的命令，将你自己的用户添加到 docker 用户组。

```
$ sudo usermod -a -G docker user
```

OpenRC

启动 docker 进程：

```
$ sudo /etc/init.d/docker start
```

开机启动：

```
$ sudo rc-update add docker default
```

systemd

启动 docker 进程：

```
$ sudo systemctl start docker.service
```

开机启动：

```
$ sudo systemctl enable docker.service
```

如果你想要添加一个 HTTP 代理，为 Docker 运行文件设置不同的目录或分区，又或者定制一些其它的功能，请阅读我们的系统文章，了解[如何定制 Docker 进程](#)

Google Cloud Platform

Google Compute Engine 镜像快速入门

1.去谷歌云控制台，创建一个新的云计算项目，启用云引擎

2.使用如下的命令来下载谷歌云 SDK 并配置您的项目：

```
$ curl https://sdk.cloud.google.com | bash
$ gcloud auth login
$ gcloud config set project <google-cloud-project-id>
```

3.启动一个新实例，使用最新的 Container-optimized 镜像:(选择一个接近你所需实例大小的分区)

```
$ gcloud compute instances create docker-playground \
--image https://www.googleapis.com/compute/v1/projects/google-containers/global/images/container-vm-v20140522 \
--zone us-central1-a \
--machine-type f1-micro
```

4.用ssh来连接这个实例：

```
$ gcloud compute ssh --zone us-central1-a docker-playground
docker-playground:~$ sudo docker run hello-world
```

当 Docker 输出 hello word 消息的时候，说明你的 Docker 工作正常。

更多，请阅读 [google 云平台部署容器](#)

Rackspace Cloud

由 Rackspace 提供的 Ubuntu 安装 Docker 是非常简单的，你可以大多按照[Ubuntu](#)的安装指南。

不过请注意：

如果你使用的 Linux 发行版没有运行 3.8 内核，你就必须升级内核，这个在 Rackspace 上是有些困难的。

Rackspace 使用 grub 的 `menu.lst` 启动服务，虽然它们运行正常，但是并不像非虚拟软件包（如xen 兼容）内核那样。所以你必须手动设置内核。

不要在线部署的机器上尝试这样做：

```
# 更新apt
$ apt-get update

# 安装新内核
$ apt-get install linux-generic-lts-raring
```

非常好，现在你已经将内核安装到 `/boot/` 下，下一步你需要让它在重新启动后生效。

```
# find the exact names
$ find /boot/ -name '*3.8*'

# this should return some results
```

现在你需要手动编译 `/boot/grub/menu.lst`，在底部有相关选项。复制和替换成新内核，确保新内核在最上边，仔细检查内核和 `initrd` 指向的文件是否正确。

要特别注意检查内核和 `initrd` 条目。

```
# 现在编辑 /boot/grub/menu.lst
vi /boot/grub/menu.lst
```

这是配置好的样子：

```
## ## End Default Options ##

title          Ubuntu 12.04.2 LTS, kernel 3.8.x generic
root           (hd0)
```

```
kernel      /boot/vmlinuz-3.8.0-19-generic root=/dev/xvda1 ro quiet splash console=hvc0
initrd      /boot/initrd.img-3.8.0-19-generic

title       Ubuntu 12.04.2 LTS, kernel 3.2.0-38-virtual
root        (hd0)
kernel      /boot/vmlinuz-3.2.0-38-virtual root=/dev/xvda1 ro quiet splash console=hvc0
initrd      /boot/initrd.img-3.2.0-38-virtual

title       Ubuntu 12.04.2 LTS, kernel 3.2.0-38-virtual (recovery mode)
root        (hd0)
kernel      /boot/vmlinuz-3.2.0-38-virtual root=/dev/xvda1 ro quiet splash single
initrd      /boot/initrd.img-3.2.0-38-virtual
```

重启你的服务器（通过命令行或者控制台）：

```
reboot
```

验证你的内核是否升级成功

```
$ uname -a
# Linux docker-12-04 3.8.0-19-generic #30~precise1-Ubuntu SMP Wed May 1 22:26:36 UTC 2013 x86_64 x86_64
x86_64 GNU/Linux

# nice! 3.8.
```

现在升级内核完成，更多信息查看[ubuntu文档安装](#)

Amazon EC2

这里有几种方法可以在 AWS EC2 上安装 Docker。你可以使用 Amazon Linux，它的软件源中已经包含了 Docker 包，或者你也可以选择其它支持 Docker 的 Linux 镜像，例如：[标准的 Ubuntu 安装](#)。

当然，首先你要创建一个AWS帐号。

Amazon QuickStart with Amazon Linux AMI 2014.09.1

1. 选择一个镜像：

- 在你的 AWS 控制台选择 [Create Instance Wizard](#) 菜单。
- 在 Quick Start 按钮中，选择 Amazon 提供的Amazon Linux 2014.09.1 机器镜像(AMI)
- 作为测试，你可以使用默认的(可能免费) `t2.micro` 实例，（更多价格，[请查看这里](#)）。
- 单击右下角的 `Next: Configure Instance Details` 按钮。

2. 在几个标准的选项（这里一般默认选择就可以）之后，你的 Amazon Linux 实例可能就运行了。

3. 使用 SSH 登录你的实例中 (instance) 来安装 Docker：

```
`ssh -i <path to your private key> ec2-user@<your public IP address>`
```

4. 当你连接到你的实例（instance）之后，输入：

```
`sudo yum install -y docker ; sudo service docker start`
```

来安装和启动 Docker。

如果这是你第一个 AWS 实例，您可能需要配置您的安全组规则来允许 ssh 连接。默认情况下，新实例 (instance) 所有流入端口都会被 AWS 安全组过滤掉。所以当你尝试连接的时候，会出现超时。

在安装完成 Docker 之后，当你准备试用它的时候，你可以查看[用户指南](#)

标准Ubuntu安装

如果你想手动配置安装，请在 EC2 主机上根据 [Ubuntu](#) 文档安装 Docker。只要按照步骤1快速选择一个镜像（或者使用你自己现有的），并跳过用户数据的步骤。然后继续按照 [Ubuntu](#) 说明进行操作。

继续查看[用户指南](#)

IBM Softlayer

1. 创建一个 [IBM SoftLayer 账户](#).
2. 登录到 [SoftLayer Customer Portal](#).
3. 在 `Devices` 菜单中选择 [设备列表](#).
4. 点击位于菜单条下方、窗口顶部右侧的 Order Devices.
5. 在 `Virtual Server` 下点击 [Hourly](#).
6. 创建一个新的 SoftLayer Virtual Server Instance (VSI) , 使用全部字段的默认值:
 - `Datacenter` 的部署位置
 - Ubuntu Linux 12.04 LTS Precise Pangolin - 最小化安装的64位系统
7. 点击底部右侧的 Continue Your Order.
8. 填写 VSI 主机名和域名。
9. 填写用户所需的元数据和订单。
10. 接下来请继续阅读 [Ubuntu](#).

What next

更多信息请阅读[用户指南](#)。

Arch Linux

你可以使用 Arch Linux 社区发布的 Docker 软件包进行安装：

- [docker](#)

或者使用 AUR 包

- [docker-git](#)

`docker` 软件包将会安装最新版本的 Docker。`docker-git` 则是由当前master分支构建的包。

依赖关系

Docker 依赖于几个指定的安装包，核心的几个依赖包为：

- bridge-utils
- device-mapper
- iproute2
- lxc
- sqlite

安装

一般包的简单安装：

```
pacman -S docker
```

这就安装了你所需要的一切。

对于AUR包的执行：

```
yaourt -S docker-git
```

这里假设你已经安装好了 yaourt.如果你之前没有安装构建过这个包，请参考 [Arch User Repository](#) .

开启Docker

Docker 会创建一个系统服务，用下面命令来启动 Docker：

```
$ sudo systemctl start docker
```

设置开机启动：

本文档使用 [看云](#) 构建

```
$ sudo systemctl enable docker
```

自定义进程选项

如果你想要添加一个 HTTP 代理，为 Docker 运行文件设置不同的目录或分区，又或者定制一些其它的功能，请阅读我们的系统文章，了解[如何定制 Docker 进程](#)

FrugalWare

在FrugalWare上使用官方包来安装:

- [lxc-docker i686](#)
- [lxc-docker x86_64](#)

`lxc-docker` 包将会安装最新版本的 Docker。

依赖关系

Docker 有几个依赖包需要安装，核心依赖包如下：

- systemd
- lvm2
- sqlite3
- libguestfs
- lxc
- iproute2
- bridge-utils

安装

只需一步就可以完整安装：

```
pacman -S lxc-docker
```

开始用docker

Docker 会创建一个系统服务，使用下面的命令启动该服务：

```
$ sudo systemctl start lxc-docker
```

设置开机启动：

```
$ sudo systemctl enable lxc-docker
```

自定义进程选项

如果你想要添加一个 HTTP 代理，为 Docker 运行文件设置不同的目录或分区，又或者定制一些其它的功能，请阅读我们的系统文章，了解[如何定制 Docker 进程](#)

Fedora

Docker 已经支持以下版本的 Fedora :

- [Fedora 20 \(64-bit\)](#)
- [Fedora 21 and later \(64-bit\)](#)

目前的 Fedora 项目，仅发行版本中的内核支持 Docker。如果你打算在非发行版本的内核上运行 Docker，内核的改动可能会导致出错。

Fedora 21 或更高版本安装 Docker

在你的主机上安装 `docker` 包来安装 Docker。

```
$ sudo yum -y install docker
```

更新 `docker` :

```
$ sudo yum -y update docker
```

请继续阅读启动 Docker 进程 [Starting the Docker daemon](#)。

Fedora 20 安装 Docker

在 Fedora 20 中，一个系统自带的可执行的应用程序与 `docker` 包名字发生冲突，所以我们给 `docker` 的 RPM 包重命名为 `docker-io`。

Fedora 20 中安装 `docker-io` 之前需要先卸载 `docker` 包。

```
$ sudo yum -y remove docker  
$ sudo yum -y install docker-io
```

更新 `docker`

```
$ sudo yum -y update docker-io
```

请继续阅读启动 Docker 进程 [Starting the Docker daemon](#)。

Starting the Docker daemon

当 Docker 安装完成之后，你需要启动 `docker` 进程。

```
$ sudo systemctl start docker
```

如果我们希望开机时自动启动 Docker ，如下操作：

```
$ sudo systemctl enable docker
```

现在，我们来验证 Docker 是否正常工作。

```
$ sudo docker run -i -t fedora /bin/bash
```

注意：如果你使用的时候提示了 `Cannot start container` 错误，错误中提到了 SELINUX 或者权限不足，你需要更新 SELinux 策略，你可以使用 `sudo yum upgrade selinux-policy` 来改变 SELinux策略并重启。

为使用 Docker 用户授权

`docker` 命令行工具通过 `socket` 文件 `/var/run/docker.sock` 和 `docker` 守护进程进行通信的。而这个 `socket` 文件的用户权限是 `root:root`。虽然 [推荐](#) 使用 `sudo` 命令来使用 `docker` 命令，但是如果你不想使用 `sudo`，系统管理员可以创建一个 `docker` 用户组，并将 `/var/run/docker.sock` 赋予 `docker` 用户组权限，然后给 `docker` 用户组添加用户即可。

```
$ sudo groupadd docker
$ sudo chown root:docker /var/run/docker.sock
$ sudo usermod -a -G docker $USERNAME
```

自定义进程选项

如果你想要添加一个 HTTP 代理，为 Docker 运行文件设置不同的目录或分区，又或者定制一些其它的功能，请阅读我们的系统文章，了解[如何定制 Docker 进程](#)

下一步

阅读 [用户指南](#).

openSUSE

Docker 支持 openSUSE 12.3 或更高版本。由于 Docker 的限制，Docker 只能运行在64位的主机上。

Docker 不被包含在 openSUSE 12.3 和 openSUSE 13.1 的官方镜像仓库中。因此需要添加 OBS 的 [虚拟化仓库](#) 来安装 docker 包

执行下边的命令来添加虚拟化仓库(Virtualization repository)：

```
# openSUSE 12.3
$ sudo zypper ar -f http://download.opensuse.org/repositories/Virtualization/openSUSE_12.3/ Virtualization

# openSUSE 13.1
$ sudo zypper ar -f http://download.opensuse.org/repositories/Virtualization/openSUSE_13.1/ Virtualization
```

在 openSUSE 13.2版本以后就不需要添加额外的库了。

SUSE Linux Enterprise

可以在 SUSE Linux Enterprise 12 或 更高版本上来运行 Docker 。这里需要注意的是由于 Docker 当前的限制，只能在64位的主机上运行。

安装

安装 Docker 包

```
$ sudo zypper in docker
```

现在已经安装完毕，让我们来启动 docker 进程

```
$ sudo systemctl start docker
```

设置开机启动 docker：

```
$ sudo systemctl enable docker
```

Docker 包会创建一个的叫 docker 的群组,如果想使用非 root 用户来运行，这个用户需要是 docker 群组的成员才可以与 docker 进程进行交互，你可以使用如下命令添加用户：

```
$ sudo usermod -a -G docker <username>
```

本文档使用 [看云](#) 构建

确认一切都是否按照预期工作：

```
$ sudo docker run --rm -i -t opensuse /bin/bash
```

这条命令将下载和导入 `opensuse` 镜像，并且在容器内运行 `bash`，输入 `exit` 来退出容器。

如果你想要你的容器能够访问外部的网络，你就需要开启 `net.ipv4.ip_forward` 规则。这里你可以使用 YaST 工具查找 Network Devices -> Network Settings -> Routing 按钮来确认 IPv4 Forwarding 选择框是否被选中。

当由 Network Manager 来管理网络的时候，就不能按照上边的方法设置了。这里我们需要手动的编辑 `/etc/sysconfig/SuSEfirewall12` 文件来确保 `FW_ROUTE` 被设置成 `yes` ,如下：

```
FW_ROUTE="yes"
```

自定义进程选项

如果你想要添加一个 HTTP 代理，为 Docker 运行文件设置不同的目录或分区，又或者定制一些其它的功能，请阅读我们的系统文章，了解[如何定制 Docker 进程](#)

下一步

阅读[用户指南](#)。

CRUX Linux

在CRUX Linux可以通过由 [James Mills](#) 提供的 ports , 或者官方的 [contrib](#) ports.

- docker

`docker` port 将构建安装最新版本的 Docker。

安装

如果你的版本允许, 更新你的 ports 目录树并且安装docker(用root用户):

```
# prt-get depinst docker
```

如果你想节省你的编译时间, 你可以安装 `docker-bin`

内核要求

如果使 CRUX + Docker 主机正常工作, 你必须确保你的内核安装必要的模块来保证 Docker 进程的正常运行。

请阅读 `README` :

```
$ prt-get readme docker
```

`docker` ports 安装由 Docker 发行版提供的 `contrib/check-config.sh` 脚本, 以供检查你的内核配置是否适合安装 Docker 主机。

运行下面的命令来检查你的内核:

```
$ /usr/share/docker/check-config.sh
```

启动docker

我们提供一个 rc 脚本来创建 Docker.请使用下列命令来启动 Docker 服务(root用户):

```
# /etc/rc.d/docker start
```

设置开机启动

- 编辑 `/etc/rc.conf`
- 将 `docker` 放到 `SERVICES=(...)` 数组 `net` 之后.

本文档使用 [看云](#) 构建

镜像

“官方库”中提供了由 [James Mills](#) 制作的 CRUX 镜像。你可以使用 `pull` 命令来使用这个镜像，当然你也可以在 `Dockerfile(s)` 中的 `FROM` 部分来设置使用。

```
$ docker pull crux
$ docker run -i -t crux
```

在 Docker Hub 中也有其他用户贡献的 [CRUX 基础镜像](#)。

Issues

如果你有一些问题，请在 [CRUX Bug Tracker](#) 提交。

支持

寻求更多技术支持，请查看 [CRUX 邮件列表](#) 或加入在 [FreeNode](#) 网络上的 [IRC Channels](#)。

Microsoft Windows

提示：Docker 已经在windows7.1和windows 8上通过测试，当然它也可以在低版本的windows上使用。但是你的处理器必须支持硬件虚拟化。

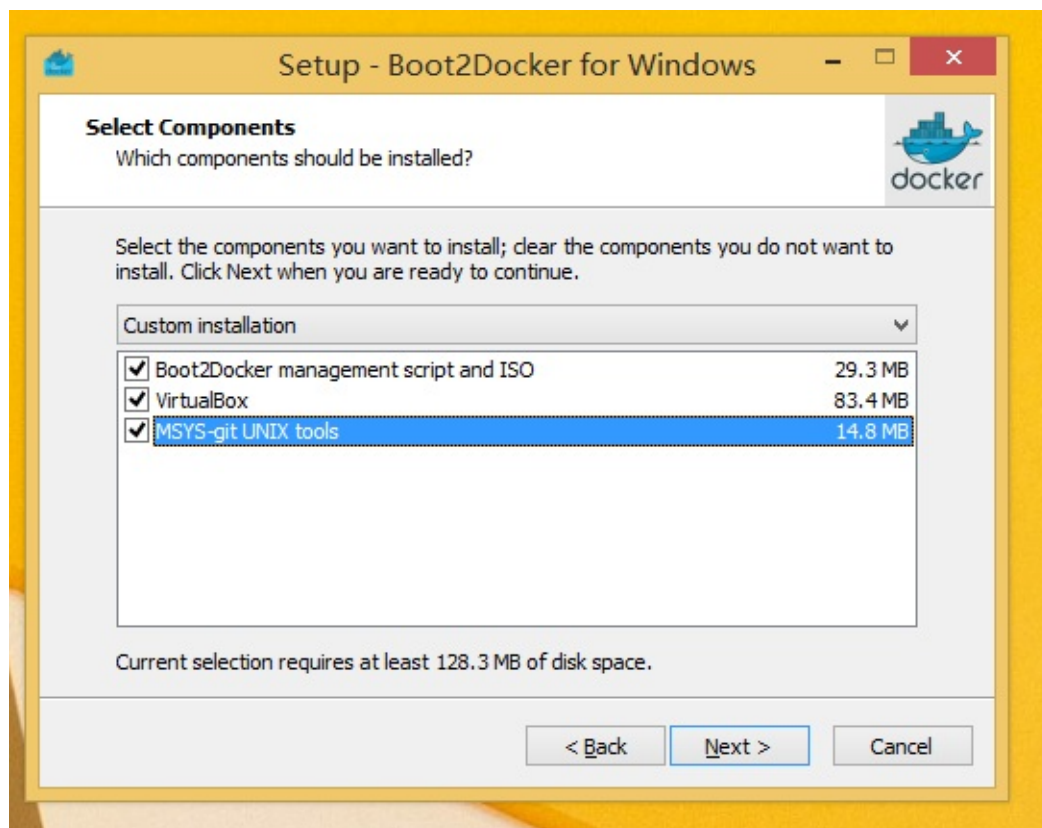
Docker 引擎使用的是Linux内核特性，所以我们需要在 Windows 上使用一个轻量级的虚拟机 (VM) 来运行 Docker。我们使用 Windows的Docker客户端来控制 Docker 虚拟化引擎的构建、运行和管理。

为了简化这个过程，我们设计了一个叫 [Boot2Docker](#) 的应用程序，你可以通过它来安装虚拟机和运行 Docker。

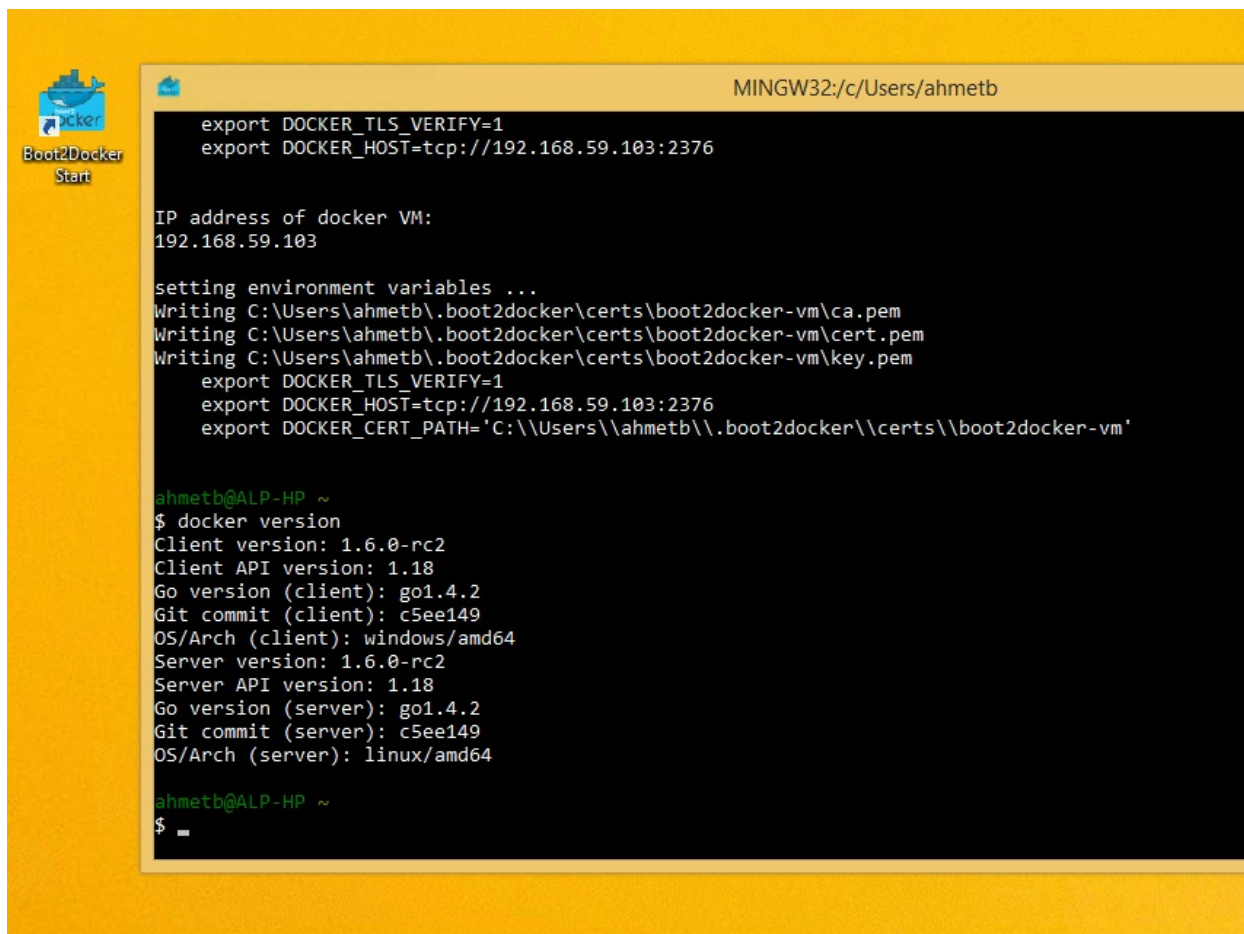
虽然你使用的是 Windows 的 Docker 客户端，但是 docker 引擎容器依然是运行在 Linux 宿主主机上（现在是通过Virtual box）。直到我们开发了 windows 版本的 Docker 引擎，你只需要在你的 Windows 主机上启动一个 Linux 容器。

安装

1. 下载最新版本的[Docker for Windows Installer](#)
2. 运行安装文件，它将会安装virtualbox、MSYS-git boot2docker Linux镜像和Boot2Docker的管理工具。



1. 从桌面上或者Program Files中找到Boot2Docker for Windows，运行 `Boot2Docker Start` 脚本。这个脚本会要求你输入 ssh 密钥密码 - 可以简单点（但是起码看起来比较安全），然后只需要按 [Enter]按钮即可。
2. `Boot2Docker Start` 将启动一个 Unix shell 来配置和管理运行在虚拟主机中的 Docker，运行 `docker version` 来查看它是否正常工作。



```

export DOCKER_TLS_VERIFY=1
export DOCKER_HOST=tcp://192.168.59.103:2376

IP address of docker VM:
192.168.59.103

setting environment variables ...
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\key.pem
export DOCKER_TLS_VERIFY=1
export DOCKER_HOST=tcp://192.168.59.103:2376
export DOCKER_CERT_PATH='C:\\Users\\ahmetb\\.boot2docker\\certs\\boot2docker-vm'

ahmetb@ALP-HP ~
$ docker version
Client version: 1.6.0-rc2
Client API version: 1.18
Go version (client): go1.4.2
Git commit (client): c5ee149
OS/Arch (client): windows/amd64
Server version: 1.6.0-rc2
Server API version: 1.18
Go version (server): go1.4.2
Git commit (server): c5ee149
OS/Arch (server): linux/amd64

ahmetb@ALP-HP ~
$ -

```

运行 Docker

注意：如果你使用的是一个远程的 Docker 进程，像 `Boot2docker`，你就不需要像前边的文档实例中那样在输入 Docker 命令之前输入 `sudo`。

`Boot2docker start` 将会自动启动一个 shell 命令框并配置好环境变量，以便您可以马上使用 Docker：

让我们尝试运行 `hello-world` 例子。运行：

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

这将会下载一个非常小的 `hello-world` 镜像，并且打印出 `Hello from Docker.` 信息。

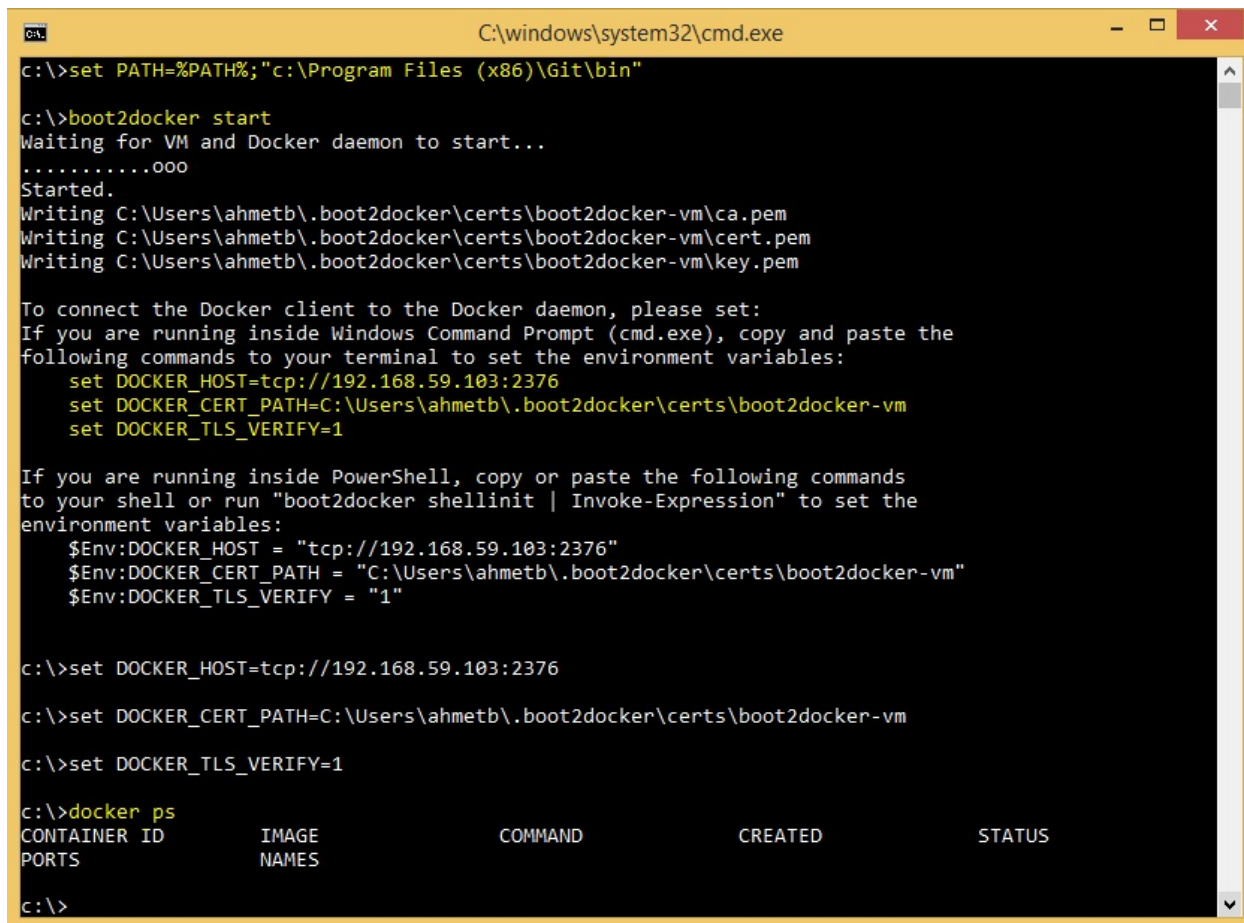
使用 Windows 的命令行(cmd.exe) 来管理运行 Docker

启动一个 Windows 命令行 (cmd.exe)。

运行 `Boot2docker` 命令，这需要你的 Windows PATH 环境变量中包含了 `ssh.exe`。因此我们需要将安装的 Git 的 bin 目录（其中包含了 `ssh.exe`）配置到我们的 `%PATH%` 环境变量中，运行如下命令：

```
set PATH=%PATH%;"c:\Program Files (x86)\Git\bin"
```

现在，我们可以运行 `boot2docker start` 命令来启动 Boot2docker 虚拟机。（如果有虚拟主机不存在的错误提示，你需要运行 `boot2docker init` 命令）。复制上边的指令到 `cmd.exe` 来设置你的 windows 控制台的环境变量，然后你就可以运行 `docker` 命令了，譬如 `docker ps`：



```
C:\windows\system32\cmd.exe
c:\>set PATH=%PATH%;"c:\Program Files (x86)\Git\bin"

c:\>boot2docker start
Waiting for VM and Docker daemon to start...
.....000
Started.
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\key.pem

To connect the Docker client to the Docker daemon, please set:
If you are running inside Windows Command Prompt (cmd.exe), copy and paste the
following commands to your terminal to set the environment variables:
    set DOCKER_HOST=tcp://192.168.59.103:2376
    set DOCKER_CERT_PATH=C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm
    set DOCKER_TLS_VERIFY=1

If you are running inside PowerShell, copy or paste the following commands
to your shell or run "boot2docker shellinit | Invoke-Expression" to set the
environment variables:
    $Env:DOCKER_HOST = "tcp://192.168.59.103:2376"
    $Env:DOCKER_CERT_PATH = "C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm"
    $Env:DOCKER_TLS_VERIFY = "1"

c:\>set DOCKER_HOST=tcp://192.168.59.103:2376

c:\>set DOCKER_CERT_PATH=C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm

c:\>set DOCKER_TLS_VERIFY=1

c:\>docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS              NAMES

c:\>
```

PowerShell 中使用 Docker

启动 PowerShell，你需要将 `ssh.exe` 添加到你的 PATH 中。

```
$Env:Path = "${Env:Path};c:\Program Files (x86)\Git\bin"
```

之后，运行 `boot2docker start` 命令行，它会打印出 PowerShell 命令，这些命令是用来设置环境变量来连接运行在虚拟机中 Docker 的。运行这些命令，然后你就可以运行 `docker` 命令了，譬如 `docker ps`：


```

C:\windows\system32\cmd.exe

c:\>set PATH=%PATH%;"c:\Program Files (x86)\Git\bin"

c:\>boot2docker start
Waiting for VM and Docker daemon to start...
.....000
Started.
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm\key.pem

To connect the Docker client to the Docker daemon, please set:
If you are running inside Windows Command Prompt (cmd.exe), copy and paste the
following commands to your terminal to set the environment variables:
    set DOCKER_HOST=tcp://192.168.59.103:2376
    set DOCKER_CERT_PATH=C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm
    set DOCKER_TLS_VERIFY=1

If you are running inside PowerShell, copy or paste the following commands
to your shell or run "boot2docker shellinit | Invoke-Expression" to set the
environment variables:
    $Env:DOCKER_HOST = "tcp://192.168.59.103:2376"
    $Env:DOCKER_CERT_PATH = "C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm"
    $Env:DOCKER_TLS_VERIFY = "1"

c:\>set DOCKER_HOST=tcp://192.168.59.103:2376

c:\>set DOCKER_CERT_PATH=C:\Users\ahmetb\.boot2docker\certs\boot2docker-vm

c:\>set DOCKER_TLS_VERIFY=1

c:\>docker ps
CONTAINER ID        IMAGE NAMES        COMMAND        CREATED        STATUS
PORTS
c:\>

```

提示:你可以使用 `boot2docker shellinit | Invoke-Expression` 来设置你的环境变量来代替复制粘贴 Powershell 命令。

进一步的细节

Boot2Docker 管理工具提供了如下几个命令：

```

$ boot2docker
Usage: boot2docker.exe [<options>] {help|init|up|ssh|save|down|poweroff|reset|restart|config|status|info|ip|shellinit|delete|download|upgrade|version} [<args>]

```

升级

- 下载最新的 [Docker for Windows Installer](#)
- 运行安装程序，这将升级 Boot2Docker 管理工具
- 打开终端输入如下的命令来升级你现有的虚拟机：

```
$ boot2docker stop $ boot2docker download $ boot2docker start
```

容器端口重定向

本文档使用 [看云](#) 构建

boot2Docker的默认用户是 `docker` 密码是 `tcuser` 。

最新版本的 boot2docker 可以设置网络适配器来给容器提供端口访问。

如你运行一个暴露内部端口的容器

```
docker run --rm -i -t -p 80:80 nginx
```

当你需要使用一个IP地址来访问 Nginx 服务器，你可以使用如下命令来查看 ip。

```
$ boot2docker ip
```

通常情况下，是192.168.59.103,但是它可以通过 virtualbox 的 dhcp 来改变。

更多细节信息，请查看[Boot2Docker site](#)

使用PUTTY登陆来代替CMD命令行

Boot2Docker使用 `%HOMEPATH%\ssh` 目录来生成你的共有和私有密钥。同样登陆的时候你也需要使用这个目录下的私有密钥。

这个私有密钥需要转换成 PuTTY 所需要的格式。

你可以使用 [puttygen](#)来生成，具体操作如下：

1. 打开 `puttygen.exe` 找到 ("File"->"Load") 按钮来加载 `%HOMEPATH%\ssh\id_boot2docker` 私有密钥文件。
2. 点击 "Save Private Key" 按钮。
3. 在PUTTY中使用刚才保存的文件来登陆 `docker@127.0.0.1:2022`

参考

如果你已经运行 Docker 主机或者你不希望使用 `Boot2docker` 安装，你可以安装 `docker.exe` 使用非官方的包管理器 `Chocolatey`。了解更多新，请查看 [Docker package on Chocolatey](#)。

Binaries

本安装说明是提供给那些想在多种环境中安装 Docker 的 hacker 们的。

在进行安装之前，请检查你的 Linux 发行版本是否有打包好的 Docker 安装包。我们已经发布了许多发行版包，这样会节省您很多时间。

检查运行时的依赖关系

如果想要 Docker 正常运行，需要安装以下软件：

- iptables version 1.4 or later
- Git version 1.7 or later
- procps (or similar provider of a "ps" executable)
- XZ Utils 4.9 or later
- a [properly mounted](#) cgroupfs hierarchy (having a single, all-encompassing "cgroup" mount point [is not sufficient](#))

检查内核的依赖关系

Docker 进程模式需要特定的内核环境支持。详情请检查您的[发行版](#)。

Docker 对 Linux 内核版本的最低要求是3.10，如果内核版本低于 3.10 会缺少一些运行 Docker 容器的功能。这些比较旧的内核，在一定条件下会导致数据丢失和频繁恐慌错误。

推荐使用版本号为 (3.x.y) 的 3.10 Linux 内核版本（或者新的维护版本），保持跟上内核的次要版本更新来确保内核的BUG已经被修复。

警告：安装自定义内核时，Linux 版本发行商可能不支持内核软件包。请务必在安装自定义内核之前，先咨询供应商是否支持 Docker。

警告：一些发行版本上还不能够安装新版本的内核，因为这些发行版本提供的包太老或者与新内核不兼容。

值得注意的是 Docker 可以以客户端模式存在，它几乎可以运行在任何的Linux内核（甚至 OS X 上）。

Enable AppArmor and SELinux when possible

如果你的 Linux 发行版上支持 AppArmor 或者 Selinux 请启用。这有助于提高安全性并阻止某些漏洞。关于如何启动设置推荐的安全机制，在发行版提供的文档上提供了详细的步骤。

某些 Linux 发行版上默认情况下是启用 AppArmor 或者 Selinux，但是它们的内核不符合安装 Docker 的最低要求（3.10或更高版本）。为了让系统能够启动 Docker 和 运行容器，需要更新内核到3.10或者更高版本。AppArmor/SELinux 用户空间(user space) 工具提供的系统与内核版本的不兼容性可能会阻止

Docker 的运行，容器的启动或者造成容器的意外退出。

警告：如果机器上启用了安全机制，它就不应该被禁用来使 Docker 和 容器运行。这样会使系统失去发行版供应商的支持，并可能打破严格的监管环境 and 安全策略。

获取Docker二进制文件

你可以下载最新版本或者特定版本的二进制版本。下载完二进制文件之后，你必须给文件可执行权限来运行。

在 Linux 或 OS X 上指定文件的执行权限：

```
$ chmod +x docker
```

从 Github 上获取稳定的发行版本号列表，请查看 [docker/docker](#) [发布页面](#)

注意

- 1) 你可以通过在 URL 中分别附加 MD5 和 SHA256 哈希值来获得二进制包。
- 2) 你可以通过 URL 中附加 .tgz 地址来获得压缩的二进制包。

获取 Linux 二进制包

通过下边的链接来下载最新版本的 Linux 二进制包：

```
https://get.docker.com/builds/Linux/i386/docker-latest
```

```
https://get.docker.com/builds/Linux/x86_64/docker-latest
```

使用下边的链接模式来下载指定版本的 Linux 二进制包：

```
https://get.docker.com/builds/Linux/i386/docker-<version>
```

```
https://get.docker.com/builds/Linux/x86_64/docker-<version>
```

举例：

```
https://get.docker.com/builds/Linux/i386/docker-1.6.0
```

```
https://get.docker.com/builds/Linux/x86_64/docker-1.6.0
```

获取 Mac OS X 二进制包

Mac OS X 的二进制文件仅仅是一个客户端。你不可以使用它来启动 docker 进程。通过下边的链接来下
本文档使用 [看云](#) 构建

载最新的 Mac OS X 版本：

```
https://get.docker.com/builds/Darwin/i386/docker-latest
https://get.docker.com/builds/Darwin/x86_64/docker-latest
```

通过下边的 URL 模式来下载指定的 Mac OS X 版本：

```
https://get.docker.com/builds/Darwin/i386/docker-<version>
https://get.docker.com/builds/Darwin/x86_64/docker-<version>
```

举例：

```
https://get.docker.com/builds/Darwin/i386/docker-1.6.0
https://get.docker.com/builds/Darwin/x86_64/docker-1.6.0
```

获取 Windows 的二进制包

从 1.60 版本开始，你可以只下载 Windows 客户端的二进制包。此外，二进制包仅是一个客户端，你不能用它来启动 docker 进程。通过下边的链接来下载最新的 Windows 版本：

```
https://get.docker.com/builds/Windows/i386/docker-latest.exe
https://get.docker.com/builds/Windows/x86_64/docker-latest.exe
```

通过下边的 URL 模式来下载指定的 Windows 版本：

```
https://get.docker.com/builds/Windows/i386/docker-<version>.exe
https://get.docker.com/builds/Windows/x86_64/docker-<version>.exe
```

举例：

```
https://get.docker.com/builds/Windows/i386/docker-1.6.0.exe
https://get.docker.com/builds/Windows/x86_64/docker-1.6.0.exe
```

运行Docker进程

```
# start the docker in daemon mode from the directory you unpacked
$ sudo ./docker -d &
```

非root用户运行

`docker` 进程一般来说默认用 `root` 用户运行，`docker` 进程绑定 `unix socket` 来代替 `TCP` 端口。默认情况下由用户 `root` 来管理 `unix socket`，但是你也可以使用 `sudo` 来使用。

如果你（你安装的 `Docker`）创建一个叫 `docker` 的 `unix` 群组，并且在群组中添加用户，当进程启动的时候，`Docker` 群组将有 `docker` 进程 `unix socket` 的读/写使用权。`docker` 进程必须使用 `root` 用户运行，但是当使用 `Docker` 群组的一个用户来运行 `Docker` 客户端的时候，你不需要在命令前添加 `sudo`。

警告：Docker 用户组（或者用 `-G` 指定的用户组）和 `root` 等效，

更新

升级你手动安装的 `Docker`，需要先关闭你的 `docker` 进程：

```
$ killall docker
```

然后按照常规的步骤安装。

运行你的第一个Docker容器

```
# check your docker version
$ sudo ./docker version

# run a container and open an interactive shell in the container
$ sudo ./docker run -i -t Ubuntu /bin/bash
```

继续阅读[用户指南](#)

用户指南

欢迎来到docker用户指南

通过这个介绍，你可以了解到 Docker 是什么，以及它是如何工作的。在本章节中，我们将 Docker 集成到你的环境中，并且通过使用 Docker 来了解一些基础知识。

我们教你如何使用docker:

- docker中运行你的应用程序。
- 运行你自己的容器。
- 创建docker镜像。
- 分享你的docker镜像。
- 和更多的信息!

我们已经将本指南分为几个主要部分：

开始使用Docker Hub

如何使用Docker Hub?

Docker Hub是docker的中心仓库。Docker Hub里存储了公共的 Docker 镜像，并且提供服务来帮助你构建和管理你的 Docker 环境。了解解更多。

阅读使用[Docker Hub](#)。

在Docker中运行 “hello Word” 应用

如何在容器内运行应用程序？

Docker 为你将要运行的应用程序提供了一个基于容器的虚拟化平台。学习如何使用 `Dockerize` 应用程序来运行他们。

阅读[Dockerize应用程序](#)

使用容器

如何管理我们的容器？

当你在docker容器中运行和管理你的应用程序，我们会展示如何管理这些容器。了解如何检查、监控和管理容器。

阅读[使用容器](#)

使用docker镜像

我是如何创建、访问和分享我自己的容器呢？

本文档使用 [看云](#) 构建

当你学会如何使用docker的时候，是时候进行下一步来学习如何在 Docker 中构建你自己应用程序镜像。

阅读[使用docker镜像](#)

容器连接

到这里我们学会了如何在 Docker 容器中构建一个单独的应用程序。而现在我们要学习如何将多个容器连接在一起构建一个完整的应用程序。

阅读[容器连接](#)

管理容器数据

现在我们知道如何连接 Docker 容器，下一步，我们学习如何管理容器数据，如何将卷挂载到我们的容器中。

阅读[管理容器数据](#)

使用Docker Hub

现在我们应该学习更多关于使用 Docker 的知识。例如通过 Docker Hub 提供的服务来构建私有仓库。

阅读[使用Docker Hub](#)

Docker Compose

Docker Compose 你只需要一个简单的配置文件就可以自定义你所需要的应用组件，包括容器、配置、网络链接和挂载卷。只需要一个简单的命令就可以启动和运行你的应用程序。

阅读[Docker Compose 用户指南](#).

Docker Machine

Docker Machine 可以帮助你快速的启动和运行 Docker 引擎。Machine 可以帮助你配置本地电脑、云服务商和你的个人数据中心上的 Docker 引擎主机，并且通过配置 Docker 客户端来让它们进行安全的通信。

查阅 [Go to Docker Machine user guide](#).

Docker 集群

Docker 集群是将多个 Docker 引擎池连接在一起组合成一个独立的主机来提供给外界。它是以 Docker API 作为服务标准的，所以任何已经在Docker上工作的工具，现在都可以透明地扩展到多个主机上。

阅读 [Go to Docker Swarm user guide](#).

使用Docker Hub

本节讲述了 Docker Hub 的快速入门,包括如何创建一个账户。

Docker Hub 存放着 Docker 及其组件的所有资源。Docker Hub 可以帮助你与同事之间协作，并获得功能完整的 Docker。为此，它提供的服务有：

- Docker 镜像主机
- 用户认证
- 自动镜像构建和工作流程工具，如构建触发器和 web hooks
- 整合了 GitHub 和 BitBucket

为了使用 Docker Hub ,首先需要注册创建一个账户。别担心，创建一个账户很简单并且是免费的。

创建 Docker Hub 账户

这里有两种访问可以创建和注册一个 Docker Hub 账户：

- 1.通过网站，或者
- 2.通过命令行

通过网站注册

填写[注册表单](#)，选择您的用户名和密码并指定您的电子邮箱。你也可以报名参加 Docker 邮件列表，会有很多关于 Docker 的信息。

Create your Docker account

Already have an account? [Login](#) instead.

Username:

Required. 4 to 30 lower case characters. Letters and digits only.

Password:

Password confirmation:

Enter the same password as above, for verification.

Email:

Mailing List:

☒ Subscribe to the Docker Weekly mailing list.

Sign up

or

 **Sign up with Github**

通过命令行

你可以通过使用命令行输入 `docker login` 命令来创建一个 Docker Hub 账号

```
$ sudo docker login
```

邮箱确认

本文档使用 [看云](#) 构建

一旦你填写完毕表格，请查看你的电子邮件，通过点击欢迎信息中的链接来激活您的账户。



To activate your account, please verify your email address

Confirm Your Email

登陆

在完成确认过程之后，您可以使用web控制台登陆

docker

Home Learn More Getting started Community Documentation Blog INDEX

sign up login

Login

Username:

Password:

Log in

[Forgot Password?](#)

或者通过在命令行中输入 `docker login` 命令来登录：

```
$ sudo docker login
```

你的 Docker 账户现在已经生效，并随时可以使用。

在Docker中运行应用

Docker 允许你在容器内运行应用程序，使用 `docker run` 命令来在容器内运行一个应用程序。

Hello world

现在让我们来试试

```
$ sudo docker run ubuntu:14.04 /bin/echo 'Hello world'
Hello world
```

刚才你启动了你的第一个容器！

那么刚才发生了什么？我们逐步来分析 `docker run` 命令做了哪些事情。

首先，我们指定了 `docker` 二进制执行文件和我们想要执行的命令 `run`。`docker run` 组合会运行容器。

接下来，我们指定一个镜像：`ubuntu 14.04`。这是我们运行容器的来源。Docker 称此为镜像。在本例中，我们使用一个 `Ubuntu 14.04` 操作系统镜像。

当你指定一个镜像，Docker 首先会先从你的 Docker 本地主机上查看镜像是否存在，如果没有，Docker 就会从镜像仓库 [Docker Hub](#) 下载公共镜像。

接下来，我们告诉 Docker 在容器内我们需要运行什么命令：

```
/bin/echo 'Hello world'
```

当我们的容器启动了 Docker 创建的新的 Ubuntu 14.04 环境，并在容器内执行 `/bin/echo` 命令后。我们会在命令行看到如下结果：

```
hello world
```

那么，我们创建容器之后会发生什么呢？当命令状态处于激活状态的时候 Docker 容器就会一直运行。这里只要 "hello world" 被输出，容器就会停止。

一个交互式的容器

让我们尝试再次运行 `docker run`，这次我们指定一个新的命令来运行我们的容器。

```
$ sudo docker run -t -i ubuntu:14.04 /bin/bash
root@af8bae53bdd3:/#
```

我们继续指定了 `docker run` 命令，并启动了 `ubuntu:14.04` 镜像。但是我们添加了两个新的标识 (参数flags)： `-t` 和 `-i` 。 `-t` 表示在新容器内指定一个伪终端或终端， `-i` 表示允许我们对容器内的 (`STDIN`) 进行交互。

我们在容器内还指定了一个新的命令： `/bin/bash` 。这将在容器内启动 `bash shell`

所以当容器 (`container`) 启动之后，我们会获取到一个命令提示符：

```
root@af8bae53bdd3:/#
```

我们尝试在容器内运行一些命令：

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

你可以看到我们运行 `pwd` 来显示当前目录，这时候显示的是我们的根目录。我们还列出了根目录的文件列表，通过目录列表我们看出来这是一个典型的 Linux 文件系统。

你可以在容器内随便的玩耍，你可以使用 `exit` 命令或者使用 `CTRL-D` 来退出容器。

```
root@af8bae53bdd3:/# exit
```

与我们之前的容器一样，一旦你的 Bash shell 退出之后，你的容器就停止了。

Hello world 守护进程

现在当一个容器运行完一个命令后就会退出，但是这样看起来有时候并不好。让我们创建一个容器以进程的方式运行，就像大多数我们运行在 Docker 中的应用程序一样，这里我们可以使用 `docker run` 命令：

```
$ sudo docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147
```

等等，怎么回事？我们的 “hello world” 输出呢？让我们看看它是怎么运行的。这个命令看起来应该很熟悉。我们运行 `docker run` ，但是我们指定了一个 `-d` 标识。 `-d` 标识告诉 docker 在容器内以后台进程模式运行。

我们也指定了一个相同的镜像： `ubuntu:14.04`

最终，我们指定命令行运行：

```
/bin/sh -c "while true; do echo hello world; sleep 1; done"
```

这是一个忠实的 hello world 进程：一个脚本会一直输出 "hello world"

为什么我们看不到的一大堆的 "hello world"？而是docker返回的一个很长的字符串：

```
1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147
```

这个长的字符串叫做容器ID（container ID）。它对于每一个容器来说都是唯一的，所以我们可以使用它。

注意：容器 ID 是有点长并且非常的笨拙，稍后我们会看到一个短点的 ID,某些方面来说它是容器 ID 的简化版。

我们可以根据容器 ID 查看 "hello world" 进程发生了什么

首先，我们要确保容器正在运行。我们可以使用 `docker ps` 命令来查看。`docker ps` 命令可以查询 docker 进程的所有容器。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
1e5535038e28	ubuntu:14.04	/bin/sh -c 'while tr	2 minutes ago	Up 1 minute		insane_babbage

这里我们看到了以进程模式运行的容器。`docker ps` 命令会返回一些有用的信息，这里包括一个短的容器 ID：1e5535038e28。

我们还可以查看到构建容器时使用的镜像，`ubuntu:14.04`，当命令运行之后，容器的状态随之改变并且被系统自动分配了名称 `insane_babbage`。

注意：当容器启动的时候 Docker 会自动给这些容器命名，稍后我们可以看到我们如何给容器指定名称。

好了，现在我们知道它正在运行。但是我们能要求它做什么呢？做到这，我们需要在我们容器内使用 `docker logs` 命令。让我们使用容器的名称来填充 `docker logs` 命令。

```
$ sudo docker logs insane_babbage
hello world
hello world
hello world
...
```


`docker logs` 命令会查看容器内的标准输出：这个例子里输出的是我们的命令 `hello world`

太棒了！我们的 docker 进程是工作的，并且我们创建了我们第一个 docker 应用。

现在我们已经可以创建我们自己的容器了，让我们处理正在运行的进程容器并停止它。我们使用 `docker stop` 命令来停止容器。

```
$ sudo docker stop insane_babbage
insane_babbage
```

`docker stop` 命令会通知 Docker 停止正在运行的容器。如果它成功了，它将返回刚刚停止的容器名称。

让我们通过 `docker ps` 命令来检查它是否还工作。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

太好了，我们的容器停止了。

使用容器

在上一节的 Docker 用户指南中，我们启动了我们的第一个容器。而后边的例子中我们使用 `docker run` 命令启动了两个容器

- 与前台进行交互的容器
- 以进程方式在后台运行的容器

在这个过程中，我们学习到了几个 Docker 命令：

- `docker ps` 列出容器
- `docker logs` 显示容器的标准输出
- `docker stop` 停止正在运行的容器

提示：另一种学习 `docker` 命令的方式就是查看我们的 [交互式教程页面](#)。

`docker` 客户端非常简单。Docker 的每一项操作都是通过命令行来实现的，而每一条命令行都可以使用一系列的标识（flags）和参数。

```
# Usage: [sudo] docker [flags] [command] [arguments] ..  
# Example:  
$ docker run -i -t ubuntu /bin/bash
```

让我们看看这个使用 `docker version` 命令的操作，它将返回当前安装的 Docker 客户端和进程的版本信息。

```
$ sudo docker version
```

这个命令不仅返回了您使用的 Docker 客户端和进程的版本信息，还返回了 GO 语言的版本信息(Docker 的编程语言)。

```
Client version: 0.8.0  
Go version (client): go1.2  
  
Git commit (client): cc3a8c8  
Server version: 0.8.0  
  
Git commit (server): cc3a8c8  
Go version (server): go1.2  
  
Last stable version: 0.8.0
```

查看一下 Docker 客户端都能做什么

我们可以通过只输入不附加任何参数的 `docker` 命令来运行 docker 二进制文件，这样我们就会查看到 Docker 客户端的所有命令选项。

```
$ sudo docker
```

会看到当前可用的所有命令行列表：

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

查看 Docker 命令用法

你可以更深入的去了解指定的 Docker 命令使用方法。

试着输入 Docker `[command]`，这里会看到 docker 命令的使用方法：

```
$ sudo docker attach
Help output . . .
```

或者你可以通过在 docker 命令中使用 `--help` 标识(flags)

```
$ sudo docker images --help
```

这将返回所有的帮助信息和可用的标识(flags)：

```
Usage: docker attach [OPTIONS] CONTAINER

Attach to a running container

--no-stdin=false: Do not attach stdin
--sig-proxy=true: Proxify all received signal to the process (even in non-tty mode)
```

注意：你可以点击[这里](#) 来查看完整的 Docker 命令行列表和使用方法。

在Docker中运行一个web应用

到这里我们了解了更多关于 docker 客户端的知识，而现在我们需要将学习的焦点转移到重要的部分：运行多个容器。到目前为止我们发现运行的容器并没有一些什么特别的用处。让我们通过使用 docker 构建一个 web 应用程序来运行一个web应用程序来体验一下。

在这个 web 应用中，我们将运行一个 Python Flask 应用。使用 `docker run` 命令。

```
$ sudo docker run -d -P training/webapp python app.py
```

让我们回顾一下我们的命令都做了什么。我们指定两个标识(flags) `-d` 和 `-P`。我们已知是 `-d` 标识是让 docker 容器在后台运行。新的 `-P` 标识通知 Docker 将容器内部使用的网络端口映射到我们使用的主机上。现在让我们看看我们的 web 应用。

This image is a pre-built image we've created that contains a simple Python Flask web application.

我们指定了 `training/web` 镜像。我们创建容器的时候使用的是这个预先构建好的镜像，并且这个镜像已经包含了简单的 Python Flask web 应用程序。

最后，我们指定了我们容器要运行的命令：`python app.py`。这样我们的 web 应用就启动了。

注意：你可以在[命令参考](#)和[Docker run参考](#)查看更多 `docker run` 命令细节

查看 WEB 应用容器

现在我们使用 `docker ps` 来查看我们正在运行的容器。

```
$ sudo docker ps -l
CONTAINER ID   IMAGE                      COMMAND                  CREATED        STATUS        PORTS
bc533791f3f5   training/webapp:latest    python app.py           5 seconds ago Up 2 seconds  0.0.0.0:49155->5000/tcp
p_nostalgic_morse
```

你可以看到我们在 `docker ps` 命令中指定了新的标识 `-l`。这样组合的 `docker ps` 命令会返回最后启动容器的详细信息。

注意：默认情况下，`docker ps` 命令只显示运行中的容器。如果你还想看已经停止的容器，请加上 `-a` 标示。

我们这里可以看到一些细节，与我们第一次运行 `docker ps` 命令的时候相比，这里多了一个 `PORTS` 列。

```
PORTS
0.0.0.0:49155->5000/tcp
```

我们通过在 `docker run` 中使用 `-P` 标示(flags) 来将我们 Docker 镜像内部容器端口暴露给主机。

提示：当我们学习[如何构建镜像的时候](#)，我们将了解更多关于如何开放 Docker 镜像端口。

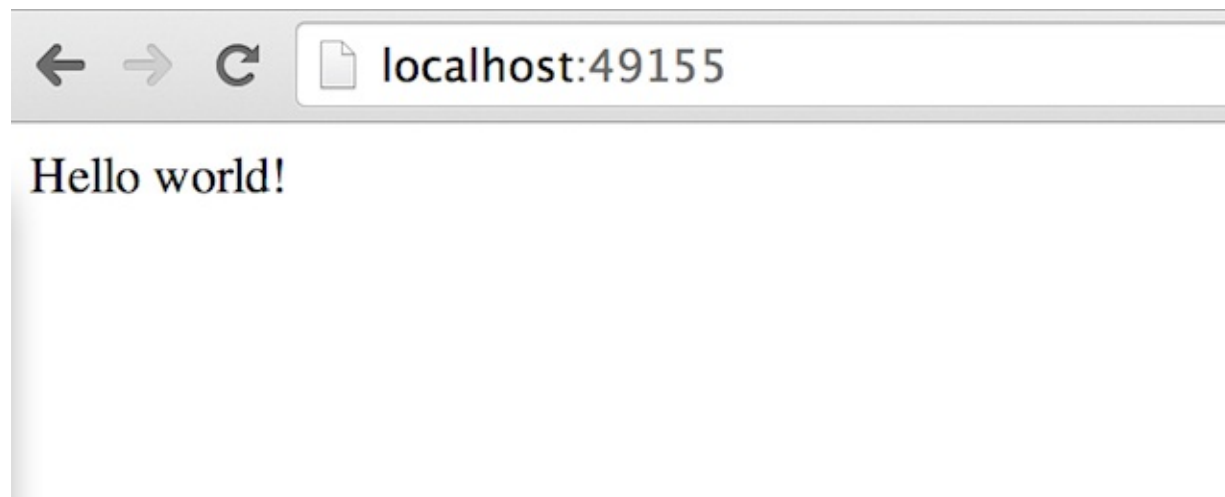
在这种情况下，Docker 开放了 5000 端口（默认 Python Flask 端口）映射到主机端口 49155 上。

Docker 能够很容易的配置和绑定网络端口。在最后一个例子中 `-P` 标识(flags)是 `-p 5000` 的缩写，它将会把容器内部的 5000 端口映射到本地 Docker 主机的高位端口上(这个端口的通常范围是 32768 至 61000)。我们也可以指定 `-p` 标识来绑定指定端口。举例：

```
$ sudo docker run -d -p 5000:5000 training/webapp python app.py
```

这将会把容器内部的 5000 端口映射到我们本地主机的 5000 端口上。你可能现在会问：为什么我们只使用 1对1端口映射的方式将端口映射到 Docker 容器，而不是采用自动映射高位端口的方式？这里 1:1 映射方式能够保证映射到本地主机端口的唯一性。假设你想要测试两个 Python 应用程序，两个容器内部都绑定了端口5000，这样就没有足够的 Docker 的端口映射，你只能访问其中一个。

所以，现在我们打开浏览器访问端口49155。



我们的应用程序可以访问了！

注意：如果你在 OS X windows或者Linux上使用 boot2docker 虚拟机，你需要获取虚拟机的 ip 来代替localhost 使用，你可以通过运行 boot2docker shell 来获取 ip。

```
$ boot2docker ip
The VM's Host only interface IP address is: 192.168.59.103
```

在这种情况下,你可以通过输入 <http://192.168.59.103:49155> 来访问上面的例子。

查看网络端口快捷方式

使用 `docker ps` 命令来会返回端口的映射是一种比较笨拙的方法。为此，Docker 提供了一种快捷方式：`docker port`，使用 `docker port` 可以查看指定（ID或者名字的）容器的某个确定端口映射到宿主机的端口号。

```
$ sudo docker port nostalgic_morse 5000
0.0.0.0:49155
```

在这种情况下，我们看到容器的 5000 端口映射到了宿主机的 49155 端口。

查看WEB应用程序日志

让我们看看我们的容器中的应用程序都发生了什么，这里我们使用学习到的另一个命令 `docker logs` 来查看。

```
$ sudo docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -
```

这次我们添加了一个 `-f` 标识。`docker log` 命令就像使用 `tail -f` 一样来输出容器内部的标准输出。这里我们从显示屏上可以看到应用程序使用的是 5000 端口并且能够查看到应用程序的访问日志。

查看WEB应用程序容器的进程

我们除了可以查看容器日志，我们还可以使用 `docker top` 来查看容器内部运行的进程：

```
$ sudo docker top nostalgic_morse
PID          USER         COMMAND
854          root         python app.py
```

这里我们可以看到 `python app.py` 在容器里唯一进程。

检查WEB应用程序

最后，我们可以使用 `docker inspect` 来查看Docker的底层信息。它会返回一个 JSON 文件记录着 Docker 容器的配置和状态信息。

```
$ sudo docker inspect nostalgic_morse
```

来让我们看下JSON的输出。

```
[{
  "ID": "bc533791f3f500b280a9626688bc79e342e3ea0d528efe3a86a51ecb28ea20",
```

```
"Created": "2014-05-26T05:52:40.808952951Z",
"Path": "python",
"Args": [
  "app.py"
],
"Config": {
  "Hostname": "bc533791f3f5",
  "Domainname": "",
  "User": "",
  . . .
```

我们也可以针对我们想要的信息进行过滤，例如，返回容器的 IP 地址，如下：

```
$ sudo docker inspect -f '{{ .NetworkSettings.IPAddress }}' nostalgic_morse
172.17.0.5
```

停止WEB应用容器

现在，我们的WEB应用程序处于工作状态。现在我们通过使用 `docker stop` 命令来停止名为 `nostalgic_morse` 的容器：

```
$ sudo docker stop nostalgic_morse
nostalgic_morse
```

现在我们使用 `docker ps` 命令来检查容器是否停止了。

```
$ sudo docker ps -l
```

重启WEB应用容器

哎呀！刚才你停止了另一个开发人员所使用的容器。这里你现在有两个选择：您可以创建一个新的容器或者重新启动旧的。让我们启动我们之前的容器：

```
$ sudo docker start nostalgic_morse
nostalgic_morse
```

现在再次运行 `docker ps -l` 来查看正在运行的容器，或者通过URL访问来查看我们的应用程序是否响应。

注意：也可以使用 `docker restart` 命令来停止容器然后再启动容器。

移除WEB应用容器

你的同事告诉你他们已经完成了在容器上的工作，不再需要容器了。让我们使用 `docker rm` 命令来删除它：

```
$ sudo docker rm nostalgic_morse
Error: Impossible to remove a running container, please stop it first or use -f
2014/05/24 08:12:56 Error: failed to remove one or more containers
```

发生了什么？实际上，我们不能删除正在运行的容器。这避免你意外删除了正在使用并且运行中的容器。让我们先停止容器，然后再试一试删除容器。

```
$ sudo docker stop nostalgic_morse
nostalgic_morse
$ sudo docker rm nostalgic_morse
nostalgic_morse
```

现在我们停止并删除了容器。

注意：删除容器是最后一步！

使用docker镜像

在[了解Docker](#)这部分中，我们知道了 Docker 镜像是容器的基础。。在[前面的部分](#)我们使用的是已经构建好的 Docker 镜像，例如：`ubuntu` 镜像和 `training/webapp` 镜像。

我们还了解到 Docker 商店下载镜像到本地的 Docker 主机上。如果一个镜像不存在，他就会自动从 Docker 镜像仓库去下载，默认是从 `Docker Hub` 公共镜像源下载。

在这一节中，我们将探讨更多的关于 Docker 镜像的东西：

- 管理和使用本地 Docker 主机镜像。
- 创建基本镜像
- 上传 Docker 镜像到 [Docker Hub Registry](#)。

在主机上列出镜像列表

让我们列出本地主机上的镜像。你可以使用 `docker images` 来完成这项任务：

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
training/webapp	latest	fc77f57ad303	3 weeks ago	280.5 MB
ubuntu	13.10	5e019ab7bf6d	4 weeks ago	180 MB
ubuntu	saucy	5e019ab7bf6d	4 weeks ago	180 MB
ubuntu	12.04	74fe38d11401	4 weeks ago	209.6 MB
ubuntu	precise	74fe38d11401	4 weeks ago	209.6 MB
ubuntu	12.10	a7cf8ae4e998	4 weeks ago	171.3 MB
ubuntu	quantal	a7cf8ae4e998	4 weeks ago	171.3 MB
ubuntu	14.04	99ec81b80c55	4 weeks ago	266 MB
ubuntu	latest	99ec81b80c55	4 weeks ago	266 MB
ubuntu	trusty	99ec81b80c55	4 weeks ago	266 MB
ubuntu	13.04	316b678ddf48	4 weeks ago	169.4 MB
ubuntu	raring	316b678ddf48	4 weeks ago	169.4 MB
ubuntu	10.04	3db9c44f4520	4 weeks ago	183 MB
ubuntu	lucid	3db9c44f4520	4 weeks ago	183 MB

我们可以看到之前使用的镜像。当我们每次要使用镜像启动一个容器的时候都会从 [Docker Hub](#) 下载对应的镜像。

我们在镜像列表中看到三个至关重要的东西。

- 来自什么镜像源，例如 `ubuntu`
- 每个镜像都有标签(tags)，例如 `14.04`
- 每个镜像都有镜像ID

镜像源中可能存储这一个镜像源的多个版本。我们会看到 `Ubuntu` 的多个版本：`10.04`, `12.04`, `12.10`, `13.04`, `13.10` and `14.04`。每个容器有一个唯一的标签，让我们来识别为不同的镜像，例如：

```
ubuntu:14.04
```

所以我们可以运行一个带标签镜像的容器：

```
$ sudo docker run -t -i ubuntu:14.04 /bin/bash
```

如果我们想要使用 `Ubuntu 12.04` 的镜像来构建，我们可以这样做

```
$ sudo docker run -t -i ubuntu:12.04 /bin/bash
```

如果你不指定一个镜像的版本标签，例如你只使用 `Ubuntu`，Docker将默认使用 `Ubuntu:latest` 镜像。

提示：我们建议使用镜像时指定一个标签，例如 `ubuntu:12.04`。这样你知道你使用的是一个什么版本的镜像。

获取一个新的镜像

现在如何获取一个新的镜像？当我们在本地主机上使用一个不存在的镜像时 Docker 就会自动下载这个镜像。但是这需要一段时间来下载这个镜像。如果我们想预先下载这个镜像，我们可以使用 `docker pull` 命令来下载它。这里我们下载 `centos` 镜像。

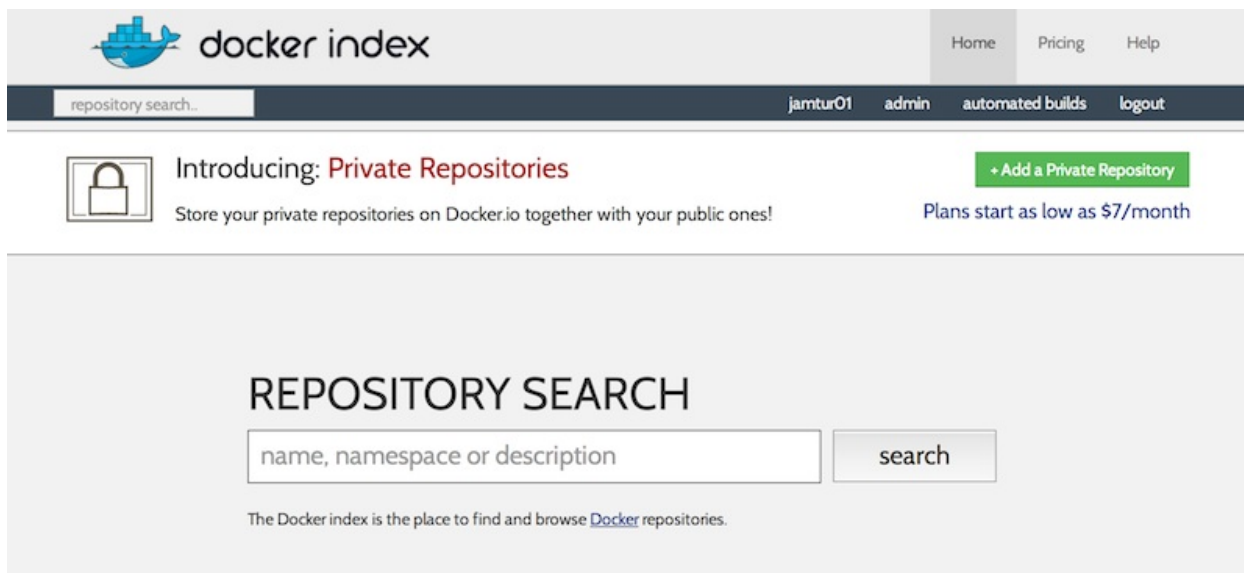
```
$ sudo docker pull centos
Pulling repository centos
b7de3133ff98: Pulling dependent layers
5cc9e91966f7: Pulling fs layer
511136ea3c5a: Download complete
ef52fb1fe610: Download complete
. . .
```

我们看到镜像的每一层都被下载下来了，现在我们可以直接使用这个镜像来运行容器，而不需要在下载这个镜像了。

```
$ sudo docker run -t -i centos /bin/bash
bash-4.1#
```

查找镜像

Docker 的特点之一是人们创建了各种各样的 Docker 镜像。而且这些镜像已经被上传到了 `Docker Hub`。我们可以从 `Docker Hub` 网站来搜索镜像。



我们也可以使用 `docker search` 命令来搜索镜像。譬如说我们的团队需要一个安装了 Ruby 和 Sinatra 的镜像来做我们的 web 应用程序开发。我们可以通过 `docker search` 命令来搜索所有的 `sinatra` 来寻找适合我们的镜像

```
$ sudo docker search sinatra
```

NAME	DESCRIPTION	STARS	OFFICIAL
alpine/sinatra	Sinatra training image	0	
marceldegraaf/sinatra	Sinatra test app	0	
mattwarren/docker-sinatra-demo		0	
luisbebop/docker-sinatra-hello-world		0	
bmorearty/handson-sinatra	handson-ruby + Sinatra for Hands on with D...	0	
subwiz/sinatra		0	
bmorearty/sinatra		0	
...			

我们看到了返回了大量的 `sinatra` 镜像。我们看到列表中有镜像名称、描述、Stars(衡量镜像的流行程度-如果用户喜欢这个镜像他就会点击 stars)和 是否是正式以及构建状态。[官方镜像仓库](#) 是官方精心整理出来服务 Docker 的 Docker 镜像库。自动化构建的镜像仓库是允许你验证镜像的内容和来源。

翻译到这里

我们回顾以前使用的镜像，我们决定使用 `sinatra` 镜像。到目前为止，我们已经看到了两种类型的镜像，像 `ubuntu` 镜像，我们称它为基础镜像或者根镜像。这些镜像是由docker公司提供建立、验证和支持。这些镜像都可以通过自己的名字来标示。

我们还可以看到用户的镜像，例如 `training/sinatra`，并且我们可以使用 `docker pull` 命令来下载它。

```
$ sudo docker pull training/sinatra
```

现在我们的团队可以在自己的容器内使用这个镜像了。

```
$ sudo docker run -t -i training/sinatra /bin/bash
root@a8cb6ce02d85:/#
```

创建自己的镜像

我们的团队发现 `training/sinatra` 镜像虽然有用但是不是我们需要的，我们需要针对这个镜像做出更改。现在又两种方法，我们可以更新和创建镜像。

- 1.我们可以从已经创建的容器中更新镜像，并且提交这个镜像。
- 2.我们可以使用 `Dockerfile` 指令来创建一个镜像。

更新并且提交镜像

更新一个镜像，首先我们要创建一个我们想更新的容器。

```
$ sudo docker run -t -i training/sinatra /bin/bash
root@0b2616b0e5a8:/#
```

注意：创建容器ID `0b2616b0e5a8` ,我们在后边还需要使用它。

在我们的容器内添加 `json`

```
root@0b2616b0e5a8:/# gem install json
```

当我们完成的时候，输入 `exit` 命令来退出这个容器。

现在我们有一个根据我们需求做出改变的容器。我们可以使用 `docker commit` 来提交这个容器。

```
$ sudo docker commit -m="Added json gem" -a="Kate Smith" \
0b2616b0e5a8 ouruser/sinatra:v2
4f177bd27a9ff0f6dc2a830403925b5360bfe0b93d476f7fc3231110e7f71b1c
```

这里我们使用了 `docker commit` 命令。我们可以指定 `-m` 和 `-a` 标示。`-m` 标示是允许我们指定提交的信息，就像你提交一个版本控制。`-a` 标示允许对我们的更新指定一个用户。

我们也指定了我们想要创建的新镜像来自(我们先前记录的ID) `0b2616b0e5a8` 和我们指定的目标镜像：

```
ouruser/sinatra:v2
```

让我们分解这个步骤。我们先给这个镜像分配了一个新用户名字 `ouruser`；接着，未修改镜像名称，保留了原镜像名称 `sinatra`；最后为镜像指定了标签 `v2`。

我们可以使用 `docker images` 命令来查看我们的新镜像 `ouruser/sinatra`。

```
$ sudo docker images
REPOSITORY          TAG       IMAGE ID       CREATED        VIRTUAL SIZE
training/sinatra     latest    5bc342fa0b91   10 hours ago   446.7 MB
ouruser/sinatra      v2        3c59e02ddd1a   10 hours ago   446.7 MB
ouruser/sinatra      latest    5db5f8471261   10 hours ago   446.7 MB
```

使用我们的新镜像来创建一个容器：

```
$ sudo docker run -t -i ouruser/sinatra:v2 /bin/bash
root@78e82f680994:/#
```

使用 Dockerfile 创建镜像

使用 `docker commit` 命令能非常简单的扩展镜像，但是它有点麻烦：在一个团队中不容易共享它的开发过程。为解决这个问题，我们可以使用一个新的命令来创建新的镜像。

为此，我们创建一个 `Dockerfile`，其中包含一组指令告诉docker如何创建我们的镜像。

现在让我们创建一个目录，并且创建一个 `Dockerfile`

```
$ mkdir sinatra
$ cd sinatra
$ touch Dockerfile
```

每一个指令镜像就会创建一个新的层，让我们看一个简单的例子，我们的开发团队创建一个自己的 `Sinatra` 镜像：

```
# This is a comment
FROM ubuntu:14.04
MAINTAINER Kate Smith <ksmith@example.com>
RUN apt-get -qq update
RUN apt-get -qq install ruby ruby-dev
RUN gem install sinatra
```

【注意】：1)、每个指令前缀都必须大写：INSTRUCTION statement 2)、可以使用 `#` 注释；

让我们看看 `Dockerfile` 做了什么：第一个指令 `FROM`，告诉Docker使用哪个镜像源，在这个案例中我们使用了一个 `Ubuntu 14.04` 基础镜像。下一步，我们使用 `MAINTAINER` 指令指定谁是维护者。最后，我们指定三个 `RUN` 指令，一个 `RUN` 指令在镜像内执行命令。例如安装包。这里我们在 `Sinatra` 中更新了APT缓存，安装了 `Ruby` 和 `RubyGems`。

注意：我们还提供了更多的Dockerfile指令参数。

现在我们使用 `docker build` 命令和 `Dockerfile` 命令来创建一个镜像。

```
$ sudo docker build -t="ouruser/sinatra:v2" .
Uploading context 2.56 kB
Uploading context
Step 0 : FROM ubuntu:14.04
--> 99ec81b80c55
Step 1 : MAINTAINER Kate Smith <ksmith@example.com>
--> Running in 7c5664a8a0c1
--> 2fa8ca4e2a13
Removing intermediate container 7c5664a8a0c1
Step 2 : RUN apt-get -qq update
--> Running in b07cc3fb4256
--> 50d21070ec0c
Removing intermediate container b07cc3fb4256
Step 3 : RUN apt-get -qqy install ruby ruby-dev
--> Running in a5b038dd127e
Selecting previously unselected package libasan0:amd64.
(Reading database ... 11518 files and directories currently installed.)
Preparing to unpack .../libasan0_4.8.2-19ubuntu1_amd64.deb ...
. . .
Setting up ruby (1:1.9.3.4) ...
Setting up ruby1.9.1 (1.9.3.484-2ubuntu1) ...
Processing triggers for libc-bin (2.19-0ubuntu6) ...
--> 2acb20f17878
Removing intermediate container a5b038dd127e
Step 4 : RUN gem install sinatra
--> Running in 5e9d0065c1f7
. . .
Successfully installed rack-protection-1.5.3
Successfully installed sinatra-1.4.5
4 gems installed
--> 324104cde6ad
Removing intermediate container 5e9d0065c1f7
Successfully built 324104cde6ad
```

我们使用 `docker build` 命令和 `-t` 来创建我们的新镜像，用户是 `ouruser`、仓库源名称 `sinatra`、标签是 `v2`。

如果 `Dockerfile` 在我们当前目录下，我们可以使用 `.` 来指定 `Dockerfile`

提示：你也可以指定 `Dockerfile` 路径

现在我们可以看到构建过程。`docker`做的第一件事是通过你的上下文构建。基本上是目录的内容构建。`docker`会根据本地的内容来在`docker`进程中去构建。

下一步，我们 `Dockerfile` 一步一步执行命令。我们可以看到，每个步骤可以创建一个新的容器，在容器内运行指令并且提交改变，就像我们早期看到的 `docker commit` 流程、当所有的指令执行完成之后，我们会得到 `324104cde6ad` 镜像（有助于标记`ouruser/sinatra:v2`），然后所有中间容器会被删除

干净。

我们可以从我们的新镜像中创建一个容器：

```
$ sudo docker run -t -i ouruser/sinatra:v2 /bin/bash
root@8196968dac35:/#
```

注意：这是比较简单的创建镜像方法。我们跳过了你可以使用的一大堆指令。在后面的部门我们将会看到更多的指令指南，或者你可以参考 [Dockerfile](#) 参考的例子和详细描述每一个指令。

设置镜像标签

你可以给现有的镜像添加标记，然后提交和构建。我们可以使用 `docker tag` 命令。让我们给 `ouruser/sinatra` 镜像添加一个新的标签。

```
$ sudo docker tag 5db5f8471261 ouruser/sinatra:devel
```

`docker tag` 指令标记镜像ID，这里是 `5db5f8471261`，设定我们的用户名称、镜像源名称和新的标签。

让我们使用 `docker images` 命令查看新的标签。

```
$ sudo docker images ouruser/sinatra
REPOSITORY          TAG         IMAGE ID      CREATED       VIRTUAL SIZE
ouruser/sinatra     latest     5db5f8471261  11 hours ago  446.7 MB
ouruser/sinatra     devel     5db5f8471261  11 hours ago  446.7 MB
ouruser/sinatra     v2        5db5f8471261  11 hours ago  446.7 MB
```

向Docker Hub推送镜像

一旦你构建或创建一个新的镜像，你可以使用 `docker push` 命令推送到Docker Hub。可以对其他人公开进行分享，或把它添加到你的私人仓库中。

```
$ sudo docker push ouruser/sinatra
The push refers to a repository [ouruser/sinatra] (len: 1)
Sending image list
Pushing repository ouruser/sinatra (3 tags)
. . .
```

主机中移除镜像

你也可以删除你主机上的镜像，某种程度上我们可以使用 `docker rmi` 命令。

让我们删除已经不需要的容器：`training/sinatra`。

```
$ sudo docker rmi training/sinatra
Untagged: training/sinatra:latest
Deleted: 5bc342fa0b91cabf65246837015197eecfa24b2213ed6a51a8974ae250fedd8d
Deleted: ed0fffdcdade5eb2c3a55549857a8be7fc8bc4241fb19ad714364cbfd7a56b22f
Deleted: 5c58979d73ae448df5af1d8142436d81116187a7633082650549c52c3a2418f0
```

提示：在容器从主机中移除前，请确定容器没有被使用。

连接容器

在使用 [Docker 部分](#), 我们谈到了如何通过网络端口来访问运行在 Docker 容器内的服务。这是与docker容器内运行应用程序交互的一种方法。在本节中, 我们打算通过端口连接到一个docker容器, 并向您介绍容器连接概念。

网络端口映射

在[使用docker](#)部分,我们创建了一个python应用的容器。

```
$ sudo docker run -d -P training/webapp python app.py
```

注：容器有一个内部网络和IP地址（在使用Docker部分我们使用 `docker inspect` 命令显示容器的IP地址）。Docker可以有各种网络配置方式。你可以再[这里](#)学到更多docker网络信息。

我们使用 `-P` 标记创建一个容器, 将容器的内部端口随机映射到主机的高端口49000到49900。这时我们可以使用 `docker ps` 来看到端口5000绑定主机端口49155。

```
$ sudo docker ps nostalgic_morse
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
bc533791f3f5	training/webapp:latest	python app.py	5 seconds ago	Up 2 seconds	0.0.0.0:49155->5000/tcp

p nostalgic_morse

我们也可以使用 `-p` 标识来指定容器端口绑定到主机端口

```
$ sudo docker run -d -p 5000:5000 training/webapp python app.py
```

我们看这为什么不是一个好的主意呢？因为它限制了我们容器的一个端口。

我们还有很多设置 `-p` 标识的方法。默认 `-p` 标识会绑定本地主机上的指定端口。并且我们可以指定绑定的网络地址。举例设置 `localhost`

```
$ sudo docker run -d -p 127.0.0.1:5001:5002 training/webapp python app.py
```

这将绑定容器内部5002端口到主机的 `localhost` 或者 `127.0.0.1` 的5001端口。

如果要绑定容器端口5002到宿主机动态端口, 并且让 `localhost` 访问, 我们可以这样做：

```
$ sudo docker run -d -p 127.0.0.1::5002 training/webapp python app.py
```

本文档使用 [看云](#) 构建

我们也可以绑定UDP端口，我们可以在后面添加 `/udp` ,举例：

```
$ sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

我们可以使用 `docker port` 快捷方式来绑定我们的端口，这有助于向我们展示特定的端口。例如我们绑定 `localhost` ,如下是 `docker port` 输出：

```
$ docker port nostalgic_morse 5000
127.0.0.1:49155
```

注：`-p` 可以使用多次配置多个端口。

Docker容器连接

端口映射并不是唯一把docker连接到另一个容器的方法。docker有一个连接系统允许将多个容器连接在一起，共享连接信息。docker连接会创建一个父子关系，其中父容器可以看到子容器的信息。

容器命名

执行此连接需要依靠你的docker的名字，这里我们可以看到当我们创建每一个容器的时候，它都会自动被命名。事实上我们已经熟悉了老的 `nostalgic_morse` 指南。你也可以自己命名容器。这种命名提供了两个有用的功能：

- 1.给容器特定的名字使你更容易记住他们，例如：命名web应用程序为web容器。
- 2.它为docker提供一个参考，允许其他容器引用，举例连接web容器到db容器。

你可以使用 `--name` 标识来命名容器，举例：

```
$ sudo docker run -d -P --name web training/webapp python app.py
```

我们可以看到我们启动了的容器，就是我们使用 `--name` 标识命名为 `web` 的容器。我们可以使用 `docker ps` 命令来查看容器名称。

```
$ sudo docker ps -l
CONTAINER ID   IMAGE                                COMMAND                  CREATED          STATUS          PORTS
NAMES
aed84ee21bde   training/webapp:latest              python app.py           12 hours ago    Up 2 seconds   0.0.0.0:49154->5000/tcp
web
```

我们也可以使用 `docker inspect` 来返回容器名字。

```
$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde
/web
```

注：容器的名称必须是唯一的。这意味着你只能调用一个web容器。如果你想使用重复的名称来命名容器，你需要使用 `docker rm` 命令删除以前的容器。在容器停止后删除。

容器连接

连接允许容器之间可见并且安全地进行通信。使用 `--link` 创建连接。我们创建一个新容器，这个容器是数据库。

```
$ sudo docker run -d --name db training/postgres
```

这里我们使用 `training/postgres` 容器创建一个新的容器。容器是PostgreSQL数据库。

现在我们创建一个 `web` 容器来连接 `db` 容器。

```
$ sudo docker run -d -P --name web --link db:db training/webapp python app.py
```

这将使我们的web容器和db容器连接起来。`--link` 的形式

```
--link name:alias
```

`name` 是我们连接容器的名字，`alias` 是link的别名。让我们看如何使用alias。

让我们使用 `docker ps` 来查看容器连接。

```
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
RTS           NAMES
349169744e49   training/postgres:latest            su postgres -c '/usr    About a minute ago    Up About a minute    54
32/tcp        db
aed84ee21bde   training/webapp:latest              python app.py            16 hours ago    Up 2 minutes    0.
0.0.0:49154->5000/tcp    db/web,web
```

我们可以看到我们命名的容器，`db` 和 `web`，我们还在名字列中可以看到web容器也显示 `db/web`。这告诉我们web容器和db容器是父/子关系。

我们连接容器做什么？我们发现连接的两个容器是父子关系。这里的父容器是 `db` 可以访问子容器 `web`。为此docker在容器之间打开一个安全连接隧道不需要暴露任何端口在容器外部。你会注意到当你启动db容器的时候我们没有使用 `-P` 或者 `-p` 标识。我们连接容器的时候我们不需要通过网络给PostgreSQL

数据库开放端口。

Docker在父容器中开放子容器连接信息有两种方法：

- 环境变量
- 更新 `/etc/hosts` 文件。

让我们先看看docker的环境变量。我们运行 `env` 命令来查看列表容器的环境变量。

```
$ sudo docker run --rm --name web2 --link db:db training/webapp env
. . .
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5000_TCP=tcp://172.17.0.5:5432
DB_PORT_5000_TCP_PROTO=tcp
DB_PORT_5000_TCP_PORT=5432
DB_PORT_5000_TCP_ADDR=172.17.0.5
. . .
```

注：这些环境变量只设置顶一个进程的容器。同样，一些守护进程(例如sshd)进行shell连接时就会去除。

我们可以看到docker为我们的数据库容器创建了一系列环境变量。每个前缀变量是 `DB_` 填充我们指定的别名。如果我们的别名是 `db1`，前缀别名就是 `DB1_`。您可以使用这些环境变量来配置您的应用程序连接到你的数据库db容器。该连接时安全、私有的，只能在web容器和db容器之间通信。

docker除了环境变量，可以添加信息到父主机的 `/etc/hosts`

```
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7 aed84ee21bde
. . .
172.17.0.5 db
```

这里我们可以看到两个主机项。第一项是web容器用容器ID作为主机名字。第二项是使用别名引用IP地址连接数据库容器。现在我们试试ping这个主机：

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db
PING db (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

注：我们不得不安装 `ping`，因为容器内没有它。

我们使用 `ping` 命令来ping db 容器，使用它解析到 `127.17.0.5` 主机。我们可以利用这个主机项配置应用程序来使用我们的 db 容器。

注：你可以使用一个父容器连接多个子容器。例如，我们可以有多个web容器连接到我们的db数据库容器。

管理容器数据

到目前为止，我们已经介绍了docker的一些基本概念，了解了如何使用docker镜像，以及容器之间如何通过网络连接。本节，我们来讨论如何管理容器和容器间的共享数据。

接下来，我们将主要介绍Docker管理数据的两种主要的方法：

- 数据卷
- 数据卷容器

数据卷

数据卷是指在存在于一个或多个容器中的特定目录，此目录能够绕过[Union File System](#)提供一些用于持续存储或共享数据的特性。

- 数据卷可在容器之间共享或重用
- 数据卷中的更改可以直接生效
- 数据卷中的更改不会包含在镜像的更新中
- 数据卷的生命周期一直持续到没有容器使用它为止

添加一个数据卷

你可以在 `docker run` 命令中使用 `-v` 标识来给容器内添加一个数据卷，你也可以在一次 `docker run` 命令中多次使用 `-v` 标识挂载多个数据卷。现在我们在web容器应用中创建单个数据卷。

```
$ sudo docker run -d -P --name web -v /webapp training/webapp python app.py
```

这会在容器内部创建一个新的卷 `/webapp`

注：类似的，你可以在 `Dockerfile` 中使用 `VOLUME` 指令来给创建的镜像添加一个或多个数据卷。

挂载一个主机目录作为卷

使用 `-v`，除了可以创建一个数据卷，还可以挂载本地主机目录到容器中：

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python app.py
```

这将会把本地目录 `/src/webapp` 挂载到容器的 `/opt/webapp` 目录。这在做测试时是非常有用的，例如我们可以挂载宿主机的源代码到容器内部，这样我们就可以看到改变源代码时的应用时如何工作的。宿主机上的目录必须是绝对路径，如果目录不存在docker会自动创建它。

注：出于可移植和分享的考虑，这种方法不能够直接在 `Dockerfile` 中实现。作为宿主机目录——其性质——是依赖于特定宿主机的，并不能够保证在所有的宿主机上都存在这样的特定目录。

docker默认情况下是对数据卷有读写权限，但是我们通过这样的方式让数据卷只读：

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp:ro training/webapp python app.py
```

这里我们同样挂载了 `/src/webapp` 目录，只是添加了 `ro` 选项来限制它只读。

将宿主机上的特定文件挂载为数据卷

除了能挂载目录外，`-v` 标识还可以将宿主机的一个特定文件挂载为数据卷：

```
$ sudo docker run --rm -it -v ~/.bash_history:/bash_history ubuntu /bin/bash
```

上述命令会在容器中运行一个bash shell，当你退出此容器时在主机上也能够看到容器中bash的命令历史。

注：很多文件编辑工具如 `vi` 和 `sed --in-place` 会导致inode change。Docker v1.1.0之后的版本，会产生一个错误：“`sed cannot rename ./sedKdJ9Dy: Device or resource busy`”。这种情况下如果想要更改挂载的文件，最好是直接挂载它的父目录。

创建、挂载数据卷容器

如果你想要容器之间数据共享，或者从非持久化容器中使用一些持久化数据，最好创建一个指定名称的数据卷容器，然后用它来挂载数据。

让我们创建一个指定名称的数据卷容器：

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

你可以在另外一个容器使用 `--volumes-from` 标识，通过刚刚创建的数据卷容器来挂载对应的数据卷。

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

可以将对应的数据卷挂载到更多的容器中：

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

当然，您也可以对一个容器使用多个 `--volumes-from` 标识，来将多个数据卷桥接到这个容器中。

本文档使用 [看云](#) 构建

数据卷容器是可以进行链式扩展的，之前的 dbdata 数据卷依次挂载到了dbdata、db1和db2容器，我们还可以使用这样的方式来将数据卷挂载到新的容器db3：

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

即使你删除所有挂载了数据卷dbdata的容器（包括最初的 dbdata 容器和后续的 db1 和 db2 ），数据卷本身也不会被删除。要删在磁盘上删除这个数据卷，只能针对最后一个挂载了数据卷的容器显式地调用 `docker rm -v` 命令。这种方式可使你在容器之间方便的更新和迁移数据。

备份、恢复或者迁移数据卷

数据卷还可以用来备份、恢复或迁移数据。为此我们使用 `--volumes-from` 参数来创建一个挂载数据卷的容器，像这样：

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

这里我们启动了一个挂载 dbdata 卷的新容器，并且挂载了一个本地目录作为 /backup 卷。最后，我们通过使用tar命令将 dbdata 卷的内容备份到容器中的 /backup 目录下的 backup.tar 文件中。当命令完成或者容器停止，我们会留下我们的 dbdata 卷的备份。

然后，你可以在同一容器或在另外的容器中恢复此数据。创建一个新的容器

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

然后在新的容器中的数据卷里un-tar此备份文件。

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

你可以对熟悉的目录应用此技术，来测试自动备份、迁移和恢复。

使用Docker Hub

现在你已经学习了如何利用命令行在本地运行Docker，还学习了如何[拉取镜像](#)用于从现成的镜像中构建容器，并且还学习了如何[创建自己的镜像](#)。

接下来，你将会学到如何利用[Docker Hub](#)简化和提高你的Docker工作流程。

[Docker Hub](#)是一个由Docker公司负责维护的公共注册中心，它包含了超过15,000个可用来下载和构建容器的镜像，并且还提供认证、工作组结构、工作流工具（比如webhooks）、构建触发器以及私有工具（比如私有仓库可用于存储你并不想公开分享的镜像）。

Docker命令和Docker Hub

Docker通过 `docker search`、`pull`、`login` 和 `push` 等命令提供了连接Docker Hub服务的功能，本页将展示这些命令如何工作的。

账号注册和登陆

一般，你需要先在docker中心创建一个账户（如果您尚未有）。你可以直接在[Docker Hub](#)创建你的账户，或通过运行：

```
$ sudo docker login
```

这将提示您输入用户名，这个用户名将成为你的公共存储库的命名空间名称。如果你的名字可用，docker会提示您输入一个密码和你的邮箱，然后会自动登录到Docker Hub，你现在可以提交和推送镜像到Docker Hub的你的存储库。

注：你的身份验证凭证将被存储在你本地目录的 `.dockercfg` 文件中。

搜索镜像

你可以通过使用搜索接口或者通过使用命令行接口在Docker Hub中搜索，可对镜像名称、用户名或者描述等进行搜索：

```
$ sudo docker search centos
```

NAME	DESCRIPTION	STARS	OFFICIAL	TRUSTED
centos	Official CentOS 6 Image as of 12 April 2014	88		
tianon/centos	CentOS 5 and 6, created using rinse instea...	21		
...				

这里你可以看到两个搜索的示例结果：`centos` 和 `tianon/centos`。第二个结果是从名为 `tianon/` 的用户仓库搜索到的，而第一个结果 `centos` 没有用户空间这就意味着它是可信的顶级命名空间。`/` 字符分割用户镜像和存储库的名称。

本文档使用 [看云](#) 构建

当你发现你想要的镜像时，便可以用 `docker pull` 来下载它。

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

现在你有一个镜像，基于它你可以运行容器。

向Docker Hub贡献

任何人都可以从 Docker Hub 仓库下载镜像，但是如果你想要分享你的镜像，你就必须先注册，就像你在[第一部分](#)的[docker用户指南](#)看到的一样。

推送镜像到Docker Hub

为了推送到仓库的公共注册库中，你需要一个命名的镜像或者将你的容器提到为一个命名的镜像，正像[这里](#)我们所看到的。

你可以将此仓库推送到公共注册库中，并以镜像名字或者标签来对其进行标记。

```
$ sudo docker push yourname/newimage
```

镜像上传之后你的团队或者社区的人都可以使用它。

Docker Hub特征

让我们再进一步看看Docker Hub的特色，[这里](#)你可以看到更多的信息。

- 私有仓库
- 组织和团队
- 自动构建
- Webhooks

私有仓库

有时候你不想公开或者分享你的镜像，所以Docker Hub允许你有私有仓库，你可以在[这里](#)登录设置它。

组织和机构

私人仓库一个较有用的地方在于你可以将仓库分享给你团队或者你的组织。Docker Hub支持创建组织，这样你可以和你的同事来管理你的私有仓库，在[这里](#)你可以学到如何创建和管理一个组织。

自动构建

自动构建功能会自动从[Github](#)和[BitBucket](#)直接将镜像构建或更新至Docker Hub，通过为Github或Bitbucket的仓库添加一个提交的hook来实现，当你推送提交的时候就会触发构建和更新。

设置一个自动化构建你需要：

- 1.创建一个[Docker Hub](#)账户并且登陆
- 2.通过[Link Accounts](#)菜单连接你的GitHub或者BitBucket
- 3.[配置自动化构建](#)
- 4.选择一个包含 Dockerfile 的Github或BitBucket项目
- 5.选择你想用于构建的分支（默认是 master 分支）
- 6.给自动构建创建一个名称
- 7.指定一个Docker标签来构建
- 8.指定 Dockerfile 的路径，默认是 / 。

一旦配置好自动构建，在几分钟内就会自动触发构建，你将会在[Docker Hub](#)仓库源看到你新的构建，并且它将会和你的Github或者BitBucket保持同步更新直到你解除自动构建。

如果你想看到你自动化构建的状态，你可以去你的Docker Hub[自动化构建页面](#)，它将会想展示你构建的状态和构建历史。

一旦你创建了一个自动化构建，你可以禁用或删除它。但是，你不能通过 `docker push` 推送一个自动化构建，而只能通过你在Github或者BitBucket提交你的代码来管理它。

你可以在一个仓库中创建多个自动构建，配置它们只指定的 Dockerfile 或Git 分支。

构建触发器

自动构建也可以通过Docker Hub的Url来触发，这样你就可以通过命令重构自动构建镜像。

Webhooks

webhooks属于你的存储库的一部分，当一个镜像更新或者推送到你的存储库时允许你触发一个事件。当你的镜像被推送的时候，webhook可以根据你指定的url和一个有效的Json来递送。

Docker Hub

先来看看 Docker Hub 的界面：

The screenshot displays the Docker Hub interface for a user named jamtur01. The top navigation bar includes a search bar, links for Browse Repos, Documentation, Community, and Help, and the user's profile icon. The left sidebar contains navigation options: Summary (selected), Repositories, Starred, Manage Settings, and Private Repositories (with a 'Buy more!' button indicating 1 of 11 used). The main content area is titled 'Your Recently Updated Repositories' and features a grid of repository cards. Each card shows the repository name, update time, Docker pull count, and star count. Below the grid, there are two tables: 'Contributed Repositories' and 'Starred Repositories'.

Repository	Stars
johnston/java	2 ★
dockercon/demo-app	4 ★
training/notes	1 ★

Repository	Stars
johnston/apache	8 ★
johnston/nginx-test	9 ★
dockercon/demo-app	4 ★

在这个章节，我们来学习 Docker Hub 的相关话题：

账户

学习如何创建一个 Docker Hub 账户来管理你的组织和机构。

仓库

了解如何分享你 Docker Hub 上的 Docker 镜像，以及如何存储和管理你的私人镜像。

自动构建

学习如何自动化构建、部署和管理

账户

Docker Hub账户

当没有数字签名和账户的时候，你只能从 [Docker Hub](#) 上 `search` 和 `pull` 一个 Docker 镜像。然后，想要 `push` 镜像到服务器、发表评论或者 `star` 一个仓库，你就需要去创建一个 [Docker Hub](#) 账户。

注册一个Docker Hub账户

你可以通过电子邮件来[注册](#)一个 Docker Hub 账户，验证邮箱后即可使用。

电子邮件激活过程

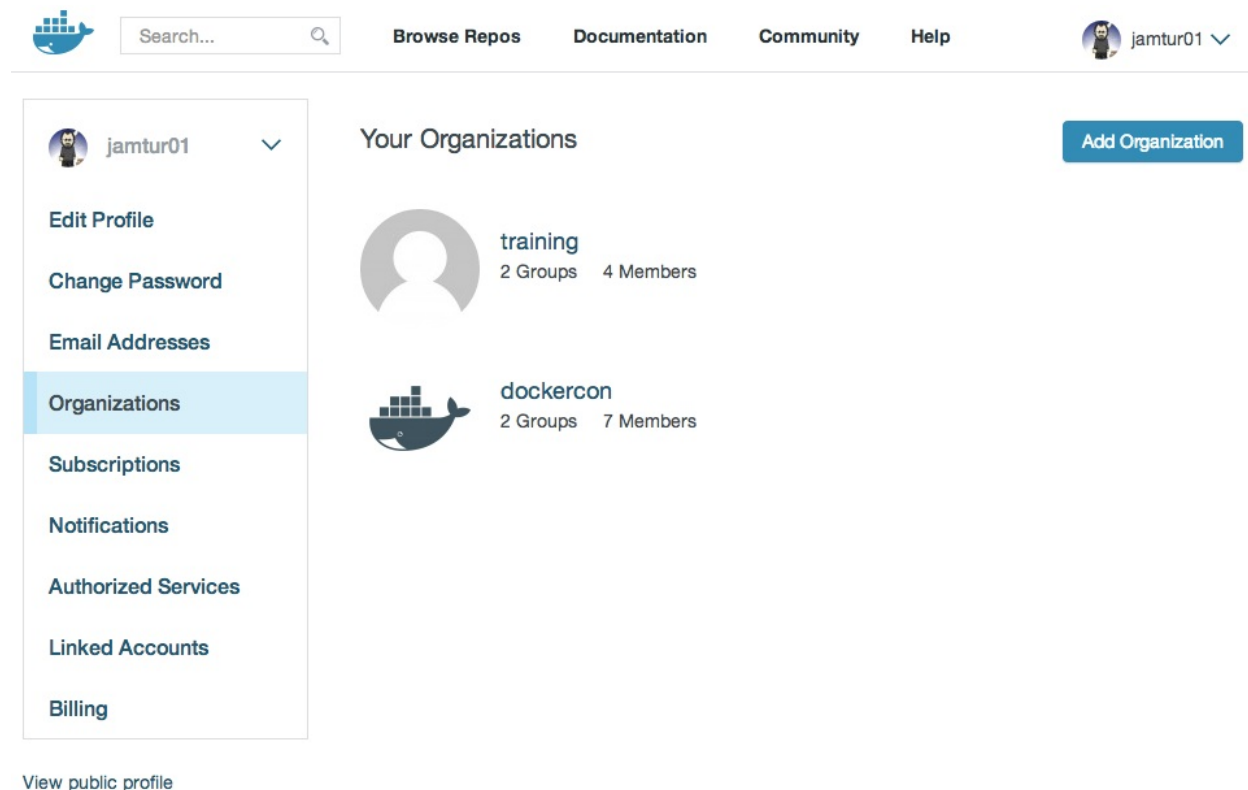
你至少需要有一个有效的电子邮件地址来验证你的账户。如果你未收到验证邮件，你可以通过访问[此页面](#)来请求重新发送确认邮件。

密码重置流程


如果由于某种原因，你忘记密码，不能访问您的账户，你可以从[密码重置页面](#)来重置你的密码。

组织&机构


Docker Hub 的机构和群组允许你加入组织和团队。你可以在[这里](#)查看你属于哪个组织，你也可以从选项卡中添加新的组织。




在你的组织中，你可以创建群组，让您进一步管理可以与你的版本库进行交互的人员。



[Browse Repos](#)
[Documentation](#)
[Community](#)
[Help](#)


jamtur01


training


[Groups](#)
[Edit Profile](#)
[Billing](#)

[View public profile](#)

Groups


[+ Create group](#)

owners



jamtur01 - James Turnbull

[Edit Group](#)


trainers


huslage - Aaron Huslage


[Edit Group](#)
[Delete group](#)


jamtur01 - James Turnbull

[Remove Member](#)


jpetazzo - Jérôme Petazzoni

[Leave](#)


nathanleclaire - Nathan LeClaire

[Remove Member](#)

存储库

搜索仓库和镜像

你可以使用 Docker 来搜索所有公开可用的仓库和镜像。

```
$ docker search ubuntu
```

这将通过 Docker 提供的关键字匹配来显示您可用的仓库列表。

私有仓库将不会显示到仓库搜索结果上。你可以通过 Docker Hub 的简况页面来查看仓库的状态。

仓库

你的 Docker Hub 仓库有许多特性。

stars

你的仓库可以用星被标记，你也可以用星标记别的仓库。Stars 也是显示你喜欢这个仓库的一种方法，也是一种简单的方法来标记你喜欢的仓库。

评论

你可以与其他 Docker 社区的成员和维护者留下评论。如果你发现有不当的评论，你可以标记他们以供审核。

合作者及其作用

指定的合作者可以通过你提供的权限访问你的私人仓库。一旦指定，他们可以 `push` 和 `pull` 你的仓库。但他们将不会被允许执行任何管理任务，如从删除仓库或者改变其状态。

注：一个合作者不能添加其他合作者。只有仓库的所有者才有管理权限。

你也可以与在 Docker Hub 上的组织和团队进行协作，更多信息。

官方仓库

Docker Hub 包含了许多[官方仓库](#)。这些都是 Docker 供应商和 Docker 贡献者提供的认证库，他们包含了来自供应商，如 Oracle 和 Red Hat 的镜像，您可以使用它们来构建应用程序和服务。

如果使用官方库，供应商会对镜像进行持续维护、升级和优化，从而为项目提供强大的驱动力。

注：如果你的组织、产品或者团队想要给官方资源库做贡献。可以再[这里](#)查看更多信息。

私有Docker仓库

私人仓库用来存储你的私有镜像，前提是你需要一个 Docker 账户，或者你已经属于 Docker Hub 上的某个组织或群组。

要使用 Docker Hub 私有仓库，首先在[这里](#)进行添加。你的 Docker Hub 账户会免费获得一个私人仓库。如果你需要更多的账户，你需要升级你的 [Docker Hub 计划](#)。

私有仓库建立好后，你可以使用 Docker 来 `push` 和 `pull` 你的镜像。

注：你需要先登录并获得权限来访问你的私人仓库。

私有仓库和公共仓库基本相同，但是以公共身份是无法浏览或者搜索到私有仓库及其内容的，他们也不会以同样的方式被缓存。

在设置页面你可以指定哪些人有权访问（如合作者），在这里你可以切换仓库状态（公共到私有，或者反过来）。你需要有一个可用的私有仓库，并开启相关设置才能做这样的转换。如果你无法进行相关操作，请升级你的 [Docker Hub 计划](#)。

Webhooks

您可以在仓库设置页面来配置你的 webhooks。只有成功 `push` 以后，`webhook` 才会生效。webhooks 会调用 HTTP POST 请求一个 json，类似如下所示的例子：

你可以使用 http 工具进行测试，例如 [requestb.in](#)。

webhook json例子:

```
{
  "push_data": {
    "pushed_at": 1385141110,
    "images": [
      "imagehash1",
      "imagehash2",
      "imagehash3"
    ],
    "pusher": "username"
  },
  "repository": {
    "status": "Active",
    "description": "my docker repo that does cool things",
    "is_automated": false,
    "full_description": "This is my full description",
    "repo_url": "https://registry.hub.docker.com/u/username/reponame/",
    "owner": "username",
    "is_official": false,
    "is_private": false,
    "name": "reponame",
    "namespace": "username",
    "star_count": 1,
    "comment_count": 1,
    "date_created": 1370174400,
  }
}
```

```
    "dockerfile": "my full dockerfile is listed here",  
    "repo_name": "username/reponame"  
  }  
}
```

Webhooks 允许你将你镜像和仓库的更新信息通知指定用户、服务以及其他应用程序。

自动构建

关于自动化构建

自动化构建是一个特殊的功能，它允许您在 Docker Hub 上使用构建集群，根据指定的 `Dockerfile` 或者 GitHub、BitBucket 仓库（或环境）来自动创建镜像。该系统将从仓库复制一份，并根据以仓库为环境的 `Dockerfile` 的描述构建镜像。由此产生的镜像将被上传到注册表，并且自动生成标记。

自动化构建有许多优势：

- 你的自动化构建项目一定是准确按照预期构建的
- 在 Docker Hub 注册表上，任何拥有你仓库访问权限的用户都乐意浏览 `Dockerfile`
- 自动化构建保证了你的仓库总是最新的

自动化构建支持 [GitHub](#) 和 [BitBucket](#) 的私有和公有的仓库。

要使用自动化构建，你必须拥有经过验证有效的 Docker Hub 账户和 GitHub/Bitbucket 账户。

设置GitHub自动化构建

首先，你需要将 GitHub 账户链接到你的 [Docker Hub](#) 账户，以允许注册表查看你的仓库。

注：目前我们需要有读写权限以建立 Docker Hub 和 GitHub 的挂钩服务，这是GitHub管理权限的方式，我们别无选择。抱歉！我们将保护您的账户及隐私，确保不会被他人非法获取。

开始构建！登录到你的 Docker Hub 账户，点击屏幕右上方的 "+ Add Repository" 按钮，选择[自动化构建](#)。

选择[GitHub服务](#)

然后按照说明授权和连接你的 GitHub 账户到 Docker Hub。连接成功后，你就可以选择用来自动化构建的仓库了。

创建一个自动化构建项目

你可以用你的 `Dockerfile` 从你的公共或者私有仓库[创建一个自动化构建项目](#)。

GitHub子模块

如果你的 GitHub 仓库包含了私有子模块的连接，你需要在 Docker Hub 上添加部署秘钥。

部署秘钥位于自动化构建主页的 "Build Details" 菜单。访问设置 GitHub 仓库的页面，选择 "Deploy keys" 来添加秘钥。

Step	Screenshot	Description

No description set
☆ 0 0 0

InformationBuild DetailsTags

Build Details

TypeNameDockerfile LocationTag Name

Branchmaster/latest

Builds History

build IdStatusCreated DateLast Updated

byekzdla7eewqzms4trym6Building2014-08-01 15:26:042014-08-01 15:26:06

Start a Build

Build Details

Links

Source Project PageSource Repository

Files

Dockerfile

Settings

DescriptionAutomated BuildWebhooksCollaboratorsBuild TriggersRepository LinksDeploy KeysMake PublicDelete Repository

你的自动化构建部署秘钥位于“Build Details” 菜单的 “Deploy keys” 下。

OptionsCollaboratorsWebhooks & ServicesDeploy keys

Deploy keys

Add deploy key

There are no deploy keys for this repository

Add a deploy key

Title

Key

Add key

在你的 GitHub 子模块仓库设置页，添加部署秘钥。


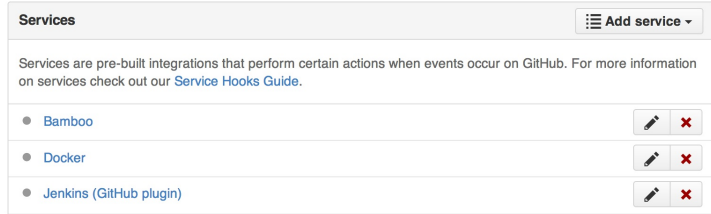
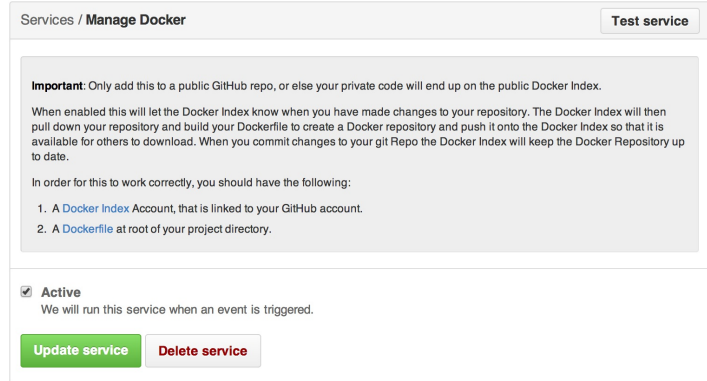
GitHub组织

一旦你的组织成员身份设置为公开，对应的 GitHub 组织状态便会被公开在你的 GitHub 上。为了验证，你可以查看 GitHub 上你的组织的成员选项卡。

GitHub服务挂钩

按照以下步骤配置自动化构建的 GitHub 服务挂钩:

Step	Screenshot	Description
1		登录到 GitHub.com，并转到您的仓库页面，点击右侧页面 “Settings”。执行该操作要求你有该仓库的管理员权限。

2		<p>点击页面左侧的“Webhooks & Services”。</p>
3		<p>找到 "Docker" 并点击它。</p>
4		<p>确认 "Active" 被选中，然后点击“Update service”按钮以保存您的更改。</p>

设置BitBucket自动化构建

为了设置自动化构建，你需要先把 BitBucket 连接到你的 Docker Hub 账户，以允许其访问你的仓库。

登录到你的 Docker Hub 账户，点击屏幕右上方的 "+ Add Repository" 按钮，选择[自动化构建](#)。

选择的 [Bitbucket 服务](#)。

然后按照说明授权和连接你的 Bitbucket 账户到 Docker Hub。连接成功后，你就可以选择用来自动化构建的仓库了。

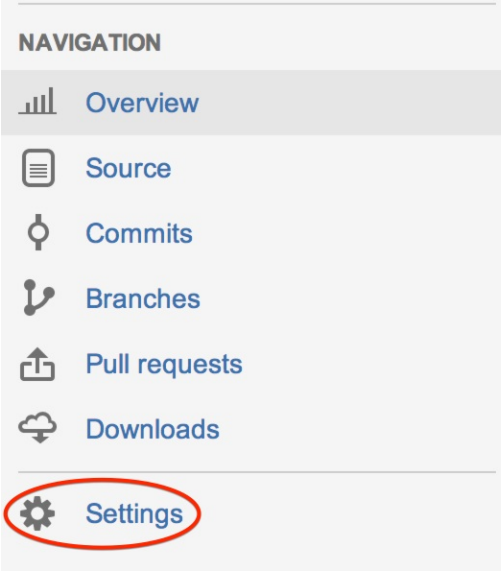
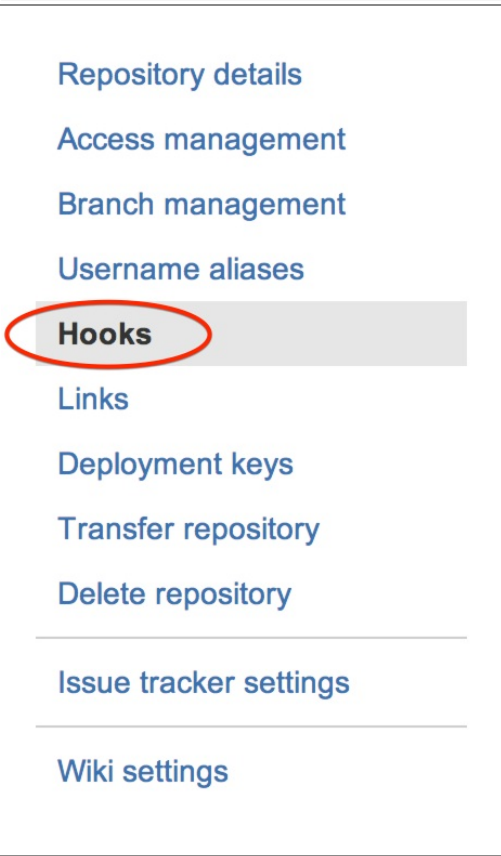
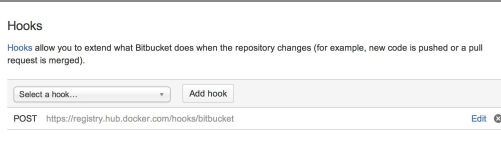
创建自动化构建项目

你可以用你的 `Dockerfile` 从你的公共或者私有仓库[创建一个自动化构建项目](#)。

Bitbucket服务挂钩

当你成功连接账户以后，一个 `POST` 挂钩将会自动被添加到你的仓库。请按照以下步骤确认或者更改你的挂钩设置：

Step	Screenshot	Description
------	------------	-------------

1		<p>登录到 Bitbucket.org 进入仓库页面。点击左侧导航下的 “Settings”。执行该操作要求你有该仓库的管理员权限。</p>
2		<p>点击左侧 “Settings” 下的 “Hooks”。</p>
3		<p>现在你应该能看到关联了该仓库的挂钩列表，包括一个指向 <code>registry.hub.docker.com/hooks/bitbucket</code> 的 POST 挂钩。</p>

Dockerfile和自动化构建

在构建过程中，我们将复制 `Dockerfile` 的内容。我们也将添加它到 Docker Hub 上，使得 Docker

社区（公共仓库）或者得到许可的团队成员可以访问仓库页面。

README.md

如果你的仓库有一个 `README.md` 文件，我们将使用它作为仓库的描述。构建过程中会寻找 `Dockerfile` 同一目录下的 `README.md`。

警告：如果你需要在创建之后修改描述，它会在下一次自动化构建完成之后生效。

建立触发器

如果你需要 GitHub 或者 BitBucket 以外的方式来触发自动化构建，你可以创建一个构建触发器。当你打开构建触发器，它会提供给你一个 url 来发送 POST 请求。这将触发自动化构建过程，类似于 GitHub webhook。

建立触发器可在自动化构建项目的 Settings 菜单中设置。

注：你在五分钟内只能触发一个构建，如果你已经进行一个构建，或你最近提交了构建请求，这些请求将被忽略。你可以在设置页面来找到最后10条触发日志来验证是否一切正常工作。

Webhooks

也可以使用 Webhooks 来自动化构建，Webhooks 会在仓库推送成功后被调用。

此webhook调用将生成一个 HTTP POST，JSON样例如下：

```
{
  "push_data": {
    "pushed_at": 1385141110,
    "images": [
      "imagehash1",
      "imagehash2",
      "imagehash3"
    ],
    "pusher": "username"
  },
  "repository": {
    "status": "Active",
    "description": "my docker repo that does cool things",
    "is_automated": false,
    "full_description": "This is my full description",
    "repo_url": "https://registry.hub.docker.com/u/username/reponame/",
    "owner": "username",
    "is_official": false,
    "is_private": false,
    "name": "reponame",
    "namespace": "username",
    "star_count": 1,
    "comment_count": 1,
    "date_created": 1370174400,
    "dockerfile": "my full dockerfile is listed here",
    "repo_name": "username/reponame"
  }
}
```

```
}
```

Webhooks 可在自动化构建项目的 Settings 菜单中设置。

注意：如果你想测试你的 webhook，我们建议使用像 requestb.in 的工具。

仓库链接

仓库链接是一种建立自动化项目与项目之间关联的方式。如果一个项目得到更新，连接系统还会触发另一个项目的更新构建。这使得你可以轻松地让所有关联项目保持更新同步。

要添加链接的话，访问你想要添加链接的项目的仓库设置页面，在设置菜单下地右侧点击 “Repository Links”。然后输入你想要与之链接的仓库名称。

警告：您可以添加多个仓库的链接，但要小心。自动化构建之间的双向关系会造成一个永不停止的构建循环。

官方案例

Docker中运行Node.js web应用

注意:——如果你不喜欢 `sudo` 可以查看[使用非root用户](#)

这个例子的目的是向您展示如何通过使用 `Dockerfile` 来构建自己的docker镜像。我们将在 Centos 上运行一个简单 `node.js` web应用并输出'hello word'。您可以在<https://github.com/enokd/docker-node-hello>/获得完整的源代码。

创建Node.js应用

首先,先创建一个文件存放目录 `src` 。然后创建 `package.json` 文件来描述你的应用程序和依赖关系:

```
{
  "name": "docker-centos-hello",
  "private": true,
  "version": "0.0.1",
  "description": "Node.js Hello world app on CentOS using docker",
  "author": "Daniel Gasienica <daniel@gasienica.ch>",
  "dependencies": {
    "express": "3.2.4"
  }
}
```

然后,创建一个 `index.js` 文件使用 [Express.js](#) 框架来创建一个web应用程序:

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

// Constants
var PORT = 8080;

// App
var app = express();
app.get('/', function (req, res) {
  res.send('Hello world\n');
});

app.listen(PORT);
console.log('Running on http://localhost:' + PORT);
```

在接下来的步骤中,我们将看到如何使用 docker 的 centos 容器来运行这个应用程序。首先,你需要为你的应用程序构建一个 docker 镜像。

创建Dockerfile

创建一个空文件叫 `Dockerfile` :

本文档使用 [看云](#) 构建

```
touch Dockerfile
```

使用你喜欢的编辑器打开Dockerfile

接下来，定义构建自己镜像的顶级镜像。在这里我们使用 Docker Hub 中 Centos (tag : 6) 镜像：

```
FROM centos:6
```

因为我们要构建一个 Node.js 应用，你需要在你的 Centos 镜像中安装Node.js。Node.js运行应用程序需要使用 npm 安装你 package.json 中定义的依赖关系。安装 Centos 包，你可以查看 Node.js 维基指令：

```
# Enable EPEL for Node.js
RUN rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
# Install Node.js and npm
RUN yum install -y npm
```

将你的应用程序源代码添加到你的Docker镜像中，使用 ADD 指令：

```
# Bundle app source
ADD . /src
```

使用npm安装你的应用程序依赖：

```
# Install app dependencies
RUN cd /src; npm install
```

应用程序绑定到端口8080，您将使用 EXPOSE 指令对 docker 端口进程映射：

```
EXPOSE 8080
```

最后，定义命令，使用 CMD 定义运行时的node服务和应用 src/index.js 的路径（看我们添加源代码到容器的步骤）：

```
CMD ["node", "/src/index.js"]
```

你的 Dockerfile 现在看起来像如下这样：

```
FROM centos:centos6
```

```
# Enable EPEL for Node.js
RUN rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
# Install Node.js and npm
RUN yum install -y npm

# Bundle app source
COPY . /src
# Install app dependencies
RUN cd /src; npm install

EXPOSE 8080
CMD ["node", "/src/index.js"]
```

创建你的个人镜像

到你的 `Dockerfile` 目录下，运行命令来构建镜像。 `-t` 参数给镜像添加标签，为了让我们在 `docker images` 命令更容易查找到它：

```
$ sudo docker build -t <your username>/centos-node-hello .
```

你的镜像现在将在列表中：

```
$ sudo docker images
```

# Example				
REPOSITORY	TAG	ID	CREATED	
centos	centos6	539c0211cd76	8 weeks ago	
<your username>/centos-node-hello	latest	d64d3505b0d2	2 hours ago	

运行镜像

使用 `-d` 参数来运行你的镜像并将容器在后台运行。使用 `-p` 参数来绑定一个公共端口到私有容器端口上。运行你之前构建的镜像：

```
$ sudo docker run -p 49160:8080 -d <your username>/centos-node-hello
```

打印应用输出：

```
# Get container ID
$ sudo docker ps

# Print app output
$ sudo docker logs <container id>

# Example
Running on http://localhost:8080
```

测试

要测试应用程序,先得到 docker 应用程序映射的端口：

本文档使用 [看云](#) 构建

```
$ sudo docker ps
```

# Example				
ID	IMAGE	COMMAND	...	PORTS
ecce33b30ebf	<your username>/centos-node-hello:latest	node /src/index.js		49160->8080

在上面的示例中，docker 映射容器的 49160 端口到 8080 端口。

现在你可以使用 curl 来访问你的app (安装curl : sudo apt-get install curl) :

```
$ curl -i localhost:49160
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
Date: Sun, 02 Jun 2013 03:53:22 GMT
Connection: keep-alive

Hello world
```

如果你使用的是 OS X 的 Boot2docker，实际上端口映射到的是 Docker 虚拟主机，你需要使用如下命令来测试：

```
$ curl $(boot2docker ip):49160
```

我们希望本教程能够帮助您在 Docker 上构建的 Centos 镜像来运行Node.js。你可以获得全部源代码<https://github.com/gasi/docker-node-hello>。

Docker中运行MongoDB

描述

在这个例子里，我们会学到如何构建一个预装 MongoDB 的镜像。我们还将会看到如何推送镜像到 Docker Hub 并分享给其他人。

使用 Docker 容器来部署 MongoDB 实例将会给你带来许多好处，例如：

- 易于维护、高可配置的 MongoDB 实例
- 准备好运行和毫秒级内开始工作
- 基于全球访问的共享镜像

注意: 如果你不喜欢sudo,可以查看[非 root 用户使用](#)

为 MongoDB 创建一个 Dockerfile

让我们创建一个 Dockerfile 并构建：

```
$ nano Dockerfile
```

虽然这部分是可选的，但是在 Dockerfile 开头处添加注释，可以更好的去解释其目的：

```
# Dockerizing MongoDB: Dockerfile for building MongoDB images
# Based on ubuntu:latest, installs MongoDB following the instructions from:
# http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/
```

小贴士：Dockerfile 文件是灵活的。然而，他们遵循一定的格式。第一项定义的是镜像的名称，这里是 Dockerized MongoDB 应用的父镜像。

我们将使用 Docker Hub 中最新版本的 Ubuntu 镜像来构建。

```
# Format: FROM repository[:version]
FROM ubuntu:latest
```

继续，我们将在 Dockerfile 中指定 MAINTAINER

```
# Format: MAINTAINER Name <email@addr.ess>
MAINTAINER M.Y. Name <myname@addr.ess>
```

注：尽管 Ubuntu 系统已经有 MongoDB 包，但是它们可能过时，因此，在这个例子中，我们将使用 MongoDB 的官方包。

我们将开始导入 MongoDB 公共 GPG 密钥。我们还将创建一个 MongoDB 库包管理器

```
# Installation:
# Import MongoDB public GPG key AND create a MongoDB list file
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sources.list.d/10gen.list
```

这个初步的准备后，我们将更新我们的包并且安装 MongoDB 。

```
# Update apt-get sources AND install MongoDB
RUN apt-get update
RUN apt-get install -y -q mongodb-org
```

小贴士: 您可以通过使用所需的软件包版本列表来指定 MongoDB 的特定版本,例如:

```
RUN apt-get install -y -q mongodb-org=2.6.1 mongodb-org-server=2.6.1 mongodb-org-shell=2.6.1 mongodb-org-tools=2.6.1
```

MongoDB 需要数据目录，让我们在最后一步中执行

```
# Create the MongoDB data directory
RUN mkdir -p /data/db
```

最后我们设置 `ENTRYPOINT` 指令来告诉 Docker 如何在我们的 MongoDB 镜像容器内运行 `mongod`。并且我们将使用 `EXPOSE` 命令来指定端口：

```
# Expose port 27017 from the container to the host
EXPOSE 27017

# Set usr/bin/mongod as the dockerized entry-point application
ENTRYPOINT usr/bin/mongod
```

现在保存我们的文件并且构建我们的镜像。

注：完整版的 `Dockerfile` 可以在这里[下载](#)

构建 MongoDB 的 Docker 镜像

本文档使用 [看云](#) 构建

我们可以使用我们的 `Dockerfile` 来构建我们的 MongoDB 镜像。除非实验，使用 `docker build` 的 `--tag` 参数来给 docker 镜像指定标签始终是一个很好的做法。

```
# Format: sudo docker build --tag/-t <user-name>/<repository> .
# Example:
$ sudo docker build --tag my/repo .
```

当我们发出这个命令时，Docker 将会通过 `Dockerfile` 来构建镜像。最终镜像将被标记成 `my/repo`。

推送 MongoDB 镜像到 Docker Hub

`docker push` 命令将会把镜像推送到 Docker Hub 上，你可以在 Docker Hub 上托管和分享推送的镜像。为此，你需要先登录：

```
# Log-in
$ sudo docker login
Username:
..

# Push the image
# Format: sudo docker push <user-name>/<repository>
$ sudo docker push my/repo
The push refers to a repository [my/repo] (len: 1)
Sending image list
Pushing repository my/repo (1 tags)
..
```

使用 MongoDB 的镜像

使用我们创建的 MongoDB 镜像，我们可以运行一个或多个的 MongoDB 守护进程。

```
# Basic way
# Usage: sudo docker run --name <name for container> -d <user-name>/<repository>
$ sudo docker run --name mongo_instance_001 -d my/repo

# Dockerized MongoDB, lean and mean!
# Usage: sudo docker run --name <name for container> -d <user-name>/<repository> --noprealloc --smallfiles
$ sudo docker run --name mongo_instance_001 -d my/repo --noprealloc --smallfiles

# Checking out the logs of a MongoDB container
# Usage: sudo docker logs <name for container>
$ sudo docker logs mongo_instance_001

# Playing with MongoDB
# Usage: mongo --port <port you get from `docker ps`>
$ mongo --port 12345
```


Docker中运行Redis服务

非常简单，没有任何修饰，redis是使用一个连接附加到一个web应用程序。

创建一个redis docker容器

首先，我们先为redis创建一个Dockerfile

```
FROM      ubuntu:12.10
RUN       apt-get update
RUN       apt-get -y install redis-server
EXPOSE    6379
ENTRYPOINT ["/usr/bin/redis-server"]
```

现在你需要通过Dockerfile创建一个镜像，将替换成你自己的名字。

```
sudo docker build -t <your username>/redis .
```

运行服务

使用我们刚才创建的redis镜像

使用 -d 运行这个服务分离模式，让容器在后台运行。

重要的是我们没有开放容器端口，相反，我们将使用一个容器来连接redis容器数据库

```
sudo docker run -name redis -d <your username>/redis
```

创建你的web应用容器

现在我们可以创建我们的应用程序容器，我们使用-link参数来创建一个连接redis容器，我们使用别名db，这将会在redis容器和redis实例容器中创建一个安全的通信隧道

```
sudo docker run -link redis:db -i -t ubuntu:12.10 /bin/bash
```

进入我们刚才创建的容器，我们需要安装redis的redis-cli的二进制包来测试连接

```
apt-get update
apt-get -y install redis-server
service redis-server stop
```

现在我们可以测试连接，首先我要先查看下web应用程序容器的环境变量，我们可以用我们的ip和端口

来连接redis容器

```
env
. . .
DB_NAME=/violet_wolf/db
DB_PORT_6379_TCP_PORT=6379
DB_PORT=tcp://172.17.0.33:6379
DB_PORT_6379_TCP=tcp://172.17.0.33:6379
DB_PORT_6379_TCP_ADDR=172.17.0.33
DB_PORT_6379_TCP_PROTO=tcp
```

我们可以看到我们有一个DB为前缀的环境变量列表，DB来自指定别名连接我们的现在的容器，让我们使用DB_PORT_6379_TCP_ADDR变量连接到Redis容器。

```
redis-cli -h $DB_PORT_6379_TCP_ADDR
redis 172.17.0.33:6379>
redis 172.17.0.33:6379> set docker awesome
OK
redis 172.17.0.33:6379> get docker
"awesome"
redis 172.17.0.33:6379> exit
```

我们可以很容易的使用这个或者其他环境变量在我们的web应用程序容器上连接到redis容器

Docker中运行PostgreSQL

注意:——如果你不喜欢sudo,可以查看[非root用户使用](#)

在Docker中安装PostgreSQL

如果Docker Hub中没有你需要的Docker镜像,你可以创建自己的镜像,开始先创建一个 Dockerfile :

注意:这个PostgreSQL仅设置用途。请参阅PostgreSQL文档来调整这些设置,以便它是安全的。

```
#
# example Dockerfile for http://docs.docker.com/examples/postgresql_service/
#

FROM ubuntu
MAINTAINER SvenDowideit@docker.com

# Add the PostgreSQL PGP key to verify their Debian packages.
# It should be the same key as https://www.postgresql.org/media/keys/ACCC4CF8.asc
RUN apt-key adv --keyserver keyserver.ubuntu.com --recv-keys B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8

# Add PostgreSQL's repository. It contains the most recent stable release
#   of PostgreSQL, ``9.3``.
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" > /etc/apt/sources.list.d/pgdg.list

# Update the Ubuntu and PostgreSQL repository indexes
RUN apt-get update

# Install ``python-software-properties``, ``software-properties-common`` and PostgreSQL 9.3
# There are some warnings (in red) that show up during the build. You can hide
# them by prefixing each apt-get statement with DEBIAN_FRONTEND=noninteractive
RUN apt-get -y -q install python-software-properties software-properties-common
RUN apt-get -y -q install postgresql-9.3 postgresql-client-9.3 postgresql-contrib-9.3

# Note: The official Debian and Ubuntu images automatically ``apt-get clean``
# after each ``apt-get``

# Run the rest of the commands as the ``postgres`` user created by the ``postgres-9.3`` package when it
# was ``apt-get installed``
USER postgres

# Create a PostgreSQL role named ``docker`` with ``docker`` as the password and
# then create a database `docker` owned by the ``docker`` role.
# Note: here we use ``&&\`` to run commands one after the other - the ``\``
#   allows the RUN command to span multiple lines.
RUN /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';" &&\
    createdb -O docker docker

# Adjust PostgreSQL configuration so that remote connections to the
# database are possible.
RUN echo "host all all 0.0.0.0/0 md5" >> /etc/postgresql/9.3/main/pg_hba.conf
```

```
# And add ``listen_addresses`` to ``/etc/postgresql/9.3/main/postgresql.conf``
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

# Expose the PostgreSQL port
EXPOSE 5432

# Add VOLUMES to allow backup of config, logs and databases
VOLUME ["/etc/postgresql", "/var/log/postgresql", "/var/lib/postgresql"]

# Set the default command to run when starting the container
CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D", "/var/lib/postgresql/9.3/main", "-c", "config_file=/etc/postgresql/9.3/main/postgresql.conf"]
```

使用Dockerfile构建镜像并且指定名称

```
$ sudo docker build -t eg_postgresql .
```

并且运行PostgreSQL服务容器

```
$ sudo docker run --rm -P --name pg_test eg_postgresql
```

有2种方法可以连接到PostgreSQL服务器。我们可以使用链接容器,或者我们可以从我们的主机(或网络)访问它。

注： `--rm` 删除容器，当容器存在时成功。

使用容器连接

在客户端 `docker run` 中直接使用 `-link remote_name:local_alias` 使容器连接到另一个容器端口。

```
$ sudo docker run --rm -t -i --link pg_test:pg eg_postgresql bash

postgres@7ef98b1b7243:/# psql -h $PG_PORT_5432_TCP_ADDR -p $PG_PORT_5432_TCP_PORT -d docker -U docker -
-password
```

连接到你的主机系统

假设你有安装postgresql客户端,您可以使用主机端口映射测试。您需要使用 `docker ps` 找出映射到本地主机端口:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
PORTS
5e24362f27f6       eg_postgresql:latest /usr/lib/postgresql/
0.0.0.0:49153->5432/tcp      pg_test
$ psql -h localhost -p 49153 -d docker -U docker --password
```

测试数据

一旦你已经通过身份验证，并且有 `docker =>` 提示，您可以创建一个表并填充它。

```
psql (9.3.1)
Type "help" for help.

$ docker=> CREATE TABLE cities (
docker(#      name          varchar(80),
docker(#      location      point
docker(# );
CREATE TABLE
$ docker=> INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
INSERT 0 1
$ docker=> select * from cities;
      name      | location
-----+-----
San Francisco | (-194,53)
(1 row)
```

使用容器卷

您可以使用PostgreSQL卷检查定义日志文件、备份配置和数据:

```
$ docker run --rm --volumes-from pg_test -t -i busybox sh

/ # ls
bin      etc      lib      linuxrc  mnt      proc     run      sys      usr
dev      home     lib64    media    opt      root     sbins    tmp      var
/ # ls /etc/postgresql/9.3/main/
environment  pg_hba.conf  postgresql.conf
pg_ctl.conf  pg_ident.conf  start.conf
/tmp # ls /var/log
ldconfig  postgresql
```

Docker中运行Riak服务

这个例子的目的是向您展示如何构建一个预装Riak的docker镜像。

创建Dockerfile

创建一个空文件 Dockerfile

```
$ touch Dockerfile
```

接下来，定义你想要来建立你镜像的父镜像。我们将使用Ubuntu（tag：最新版），从Docker Hub中下载：

```
# Riak
#
# VERSION          0.1.0

# Use the Ubuntu base image provided by dotCloud
FROM ubuntu:latest
MAINTAINER Hector Castro hector@basho.com
```

接下来,我们更新APT缓存和应用更新:

```
# Update the APT cache
RUN sed -i.bak 's/main$/main universe/' /etc/apt/sources.list
RUN apt-get update
RUN apt-get upgrade -y
```

之后，我们安装和设置一些依赖关系：

- `CURL` 来下载 Basho's APT存储库密钥。
- `lsb-release` 帮助我们查看Ubuntu版本。
- `openssh-server` 允许我们登陆远程容器，加入Riak节点组成一个集群。
- `supervisor` 用于管理 OpenSSH 和 Riak 进程。

Install and setup project dependencies

```
RUN apt-get install -y curl lsb-release supervisor openssh-server
```

```
RUN mkdir -p /var/run/sshd RUN mkdir -p /var/log/supervisor
```

```
RUN locale-gen en_US en_US.UTF-8
```

```
ADD supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

`RUN echo 'root:basho' | chpasswd`

下一步，添加 Basho's APT仓库：

```
RUN curl -s http://apt.basho.com/gpg/basho apt-key add --
RUN echo "deb http://apt.basho.com $(lsb_release -cs) main" > /etc/apt/sources.list.d/basho.list
RUN apt-get update
```

之后，我们安装Riak和改变一些默认值：

```
# Install Riak and prepare it to run
RUN apt-get install -y riak
RUN sed -i.bak 's/127.0.0.1/0.0.0.0/' /etc/riak/app.config
RUN echo "ulimit -n 4096" >> /etc/default/riak
```

接下来，我们为缺少的 `initctl` 来添加一个软连接：

```
# Hack for initctl
# See: https://github.com/dotcloud/docker/issues/1024
RUN dpkg-divert --local --rename --add /sbin/initctl
RUN ln -s /bin/true /sbin/initctl
```

然后我们开发Riak协议缓冲区、HTTP接口以及SSH：

```
# Expose Riak Protocol Buffers and HTTP interfaces, along with SSH
EXPOSE 8087 8098 22
```

最后，运行 `supervisord` 这里Riak和OpenSSH将启动：

```
CMD ["/usr/bin/supervisord"]
```

创建一个supervisord配置文件

创建一个supervisord.conf空文件，并且保证和Dockerfile是同级目录：

```
touch supervisord.conf
```

填充下面定义的程序：

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D
```

```
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
autorestart=true

[program:riak]
command=bash -c ". /etc/default/riak && /usr/sbin/riak console"
pidfile=/var/log/riak/riak.pid
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
```

构建Riak的Docker镜像

现在你应该能够构建一个Riak的docker镜像:

```
$ docker build -t "<yourname>/riak" .
```

下一步

Riak是分布式数据库。很多生产部署包括至少5个节点。查看docker-riak<https://github.com/hectcastro/docker-riak>项目细节来使用Docker和Pipework部署Riak集群。

Docker中运行SSH进程服务

以下是用 Dockerfile 设置sshd服务容器，您可以使用连接并检查其他容器的卷,或者可以快速访问测试容器。

```
# sshd
#
# VERSION          0.0.1

FROM      ubuntu:12.04
MAINTAINER Thatcher R. Peskens "thatcher@dotcloud.com"

# make sure the package repository is up to date
RUN apt-get update

RUN apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:screencast' |chpasswd

EXPOSE 22
CMD      ["/usr/sbin/sshd", "-D"]
```

使用如下命令构建镜像：

```
$ sudo docker build --rm -t eg_sshd .
```

然后运行它，你可以使用 `docker port` 来找出容器端口22映射到主机的端口

```
$ sudo docker run -d -P --name test_sshd eg_sshd
$ sudo docker port test_sshd 22
0.0.0.0:49154
```

现在你可以使用ssh登陆Docker进程的主机IP地址，端口是49154(IP地址可以使用ifconfig获取)：

```
$ ssh root@192.168.1.2 -p 49154
# The password is ``screencast``.
$
```

最后，清理停止的容器，并且删除容器，然后删除镜像。

```
$ sudo docker stop test_sshd
$ sudo docker rm test_sshd
$ sudo docker rmi eg_sshd
```


Docker中运行CouchDB服务

注：如果你不喜欢使用sudo，你可以查看[这里非root用户运行](#)

这里有一个例子，使用数据卷在两个CouchDb之间共享相同的数据容器，这个可以用于热升级，测试不同版本的CouchDB数据等等。

创建第一个数据库

现在我们创建/var/lib/couchdb作为数据卷

```
COUCH1=$(sudo docker run -d -p 5984 -v /var/lib/couchdb shykes/couchdb:2013-05-03)
```

添加一条数据在第一个数据库中

我们假设你的docker主机默认是本地localhost.如果不是localhost请换到你docker的公共IP

```
HOST=localhost
URL="http://$HOST:$(sudo docker port $COUCH1 5984 | grep -Po '\d+$')/_utils/"
echo "Navigate to $URL in your browser, and use the couch interface to add data"
```

创建第二个数据库

这次，我们请求共享访问\$COUCH1的卷。

```
COUCH2=$(sudo docker run -d -p 5984 -volumes-from $COUCH1 shykes/couchdb:2013-05-03)
```

在第二个数据库上来浏览数据

```
HOST=localhost
URL="http://$HOST:$(sudo docker port $COUCH2 5984 | grep -Po '\d+$')/_utils/"
echo "Navigate to $URL in your browser. You should see the same data as in the first database"''
```

祝贺你，你已经运行了两个Couchdb容器，并且两个都相互独立，除了他们的数据

Docker中运行Apt-Cacher-ng服务

注意:——如果你不喜欢sudo,可以查看[非root用户使用](#), --如何你使用OS X或者通过TCP使用Docker, 你需要使用sudo

当你有许多docker服务器, 或者不能使用Docker缓存来构建不相干的Docker容器, 他可以为你的包缓存代理, 这是非常有用的。该容器使第二个下载的任何包几乎瞬间下载。

使用下边的Dockerfile

```
#
# Build: docker build -t apt-cacher .
# Run: docker run -d -p 3142:3142 --name apt-cacher-run apt-cacher
#
# and then you can run containers with:
#   docker run -t -i --rm -e http_proxy http://dockerhost:3142/ debian bash
#
FROM          ubuntu
MAINTAINER    SvenDowideit@docker.com

VOLUME        ["/var/cache/apt-cacher-ng"]
RUN           apt-get update ; apt-get install -yq apt-cacher-ng

EXPOSE        3142
CMD           chmod 777 /var/cache/apt-cacher-ng ; /etc/init.d/apt-cacher-ng start ; tail -f /var/log/apt-cacher-ng/*
```

使用下边的命令构建镜像：

```
$ sudo docker build -t eg_apt_cacher_ng .
```

现在运行它, 映射内部端口到主机

```
$ sudo docker run -d -p 3142:3142 --name test_apt_cacher_ng eg_apt_cacher_ng
```

查看日志文件, 默认使用 `tailed` 命令, 你可以使用

```
$ sudo docker logs -f test_apt_cacher_ng
```

让你Debian-based容器使用代理,你可以做三件事之一：

本文档使用 [看云](#) 构建

1.添加一个apt代理设置

```
echo 'Acquire::http { Proxy "http://dockerhost:3142"; };' >> /etc/apt/conf.d/
```

2.设置环境变量： http_proxy=http://dockerhost:3142/

3.修改你的 sources.list 来开始 http://dockerhost:3142/

选项1注入是安全设置到你的apt配置在本地的公共基础版本。

```
FROM ubuntu
RUN echo 'Acquire::http { Proxy "http://dockerhost:3142"; };' >> /etc/apt/apt.conf.d/01proxy
RUN apt-get update ; apt-get install vim git

# docker build -t my_ubuntu .
```

选项2针对测试时非常好的，但是会破坏其它从http代理的HTTP客户端，如curl、wget或者其他：

```
$ sudo docker run --rm -t -i -e http_proxy=http://dockerhost:3142/ debian bash
```

选项3是最轻便的，但是有时候你可能需要做很多次，你也可以在你的 Dockerfile 这样做：

```
$ sudo docker run --rm -t -i --volumes-from test_apt_cacher_ng eg_apt_cacher_ng bash

$ /usr/lib/apt-cacher-ng/distkill.pl
Scanning /var/cache/apt-cacher-ng, please wait...
Found distributions:
bla, taggedcount: 0
  1\. precise-security (36 index files)
  2\. wheezy (25 index files)
  3\. precise-updates (36 index files)
  4\. precise (36 index files)
  5\. wheezy-updates (18 index files)

Found architectures:
  6\. amd64 (36 index files)
  7\. i386 (24 index files)

WARNING: The removal action may wipe out whole directories containing
index files. Select d to see detailed list.

(Number nn: tag distribution or architecture nn; 0: exit; d: show details; r: remove tagged; q: quit):
q
```

最后，停止测试容器，删除容器，删除镜像：

```
$ sudo docker stop test_apt_cacher_ng
$ sudo docker rm test_apt_cacher_ng
$ sudo docker rmi eg_apt_cacher_ng
```

