

# *Docker*

——从入门到实践



# Docker —— 从入门到实践

v0.2.9

[Docker](#) 是个伟大的项目，它彻底释放了虚拟化的威力，让应用的分发、部署和管理都变得前所未有的高效和轻松！

本书既适用于具备基础 Linux 知识的 Docker 初学者，也可供希望理解原理和实现的高级用户参考。同时，书中给出的实践案例，可供在进行实际部署时借鉴。

本书源于 [WaitFish](#) 的《[Docker 学习手册 v1.0](#)》内容。后来，[yeasy](#) 根据最新 Docker 版本对内容进行了修订和重写，并增加内容；经协商将所有内容开源，采用互联网合作的方式进行维护。

前六章为基础内容，供用户理解 Docker 的基本概念和操作；7~9 章介绍一些高级操作；第 10 章给出典型的应用场景和实践案例；11~13 章介绍关于 Docker 实现的相关技术。

最新版本在线阅读：[GitBook](#) 或 [DockerPool](#)。

另外，欢迎加入 DockerPoolQQ 群（341410255），分享 Docker 资源，交流 Docker 技术。

本书源码在 Github 上维护，欢迎参与：[https://github.com/yeasy/docker\\_practice](https://github.com/yeasy/docker_practice)。

感谢所有的 [贡献者](#)。

## 主要版本历史

- 0.3: 2014-10-TODO
  - 完成仓库章节；
  - 重写安全章节；
  - 修正底层实现章节的架构、名字空间、控制组、文件系统、容器格式等内容；
  - 添加对常见仓库和镜像的介绍；
  - 添加 Dockerfile 的介绍；
- 重新校订中英文混排格式。
- 0.2: 2014-09-18
  - 对照官方文档重写介绍、基本概念、安装、镜像、容器、仓库、数据管理、网络等章节；
  - 添加底层实现章节；
  - 添加命令查询和资源链接章节；
- 其它修正。
- 0.1: 2014-09-05
  - 添加基本内容；
  - 修正错别字和表达不通顺的地方。

## 参加步骤

在 GitHub 上 fork 到自己的仓库，如 `docker_user/docker_practice`，然后 clone 到本地，并设置用户信息。

```
git clone git@github.com:docker_user/docker_practice.git  
cd docker_practice  
git config user.name "Docker User"  
git config user.email docker_user@dockcer.com
```

- 修改代码后提交，并推送到自己的仓库。

```
$ #do some change on the content  
$ git commit -am "Fix issue #1: change helo to hello"  
$ git push
```

- 在 GitHub 网站上提交 pull request。
- 定期使用项目仓库内容更新自己仓库内容。

```
git remote add upstream https://github.com/yeasy/docker_practice  
git fetch upstream  
git checkout master  
git rebase upstream/master  
git push -f origin master
```

# 简介

---

本章将带领你进入 Docker 的世界。

什么是 Docker？

用它会带来什么样的好处？

好吧，让我们带着问题开始这神奇之旅。

# 什么是 Docker

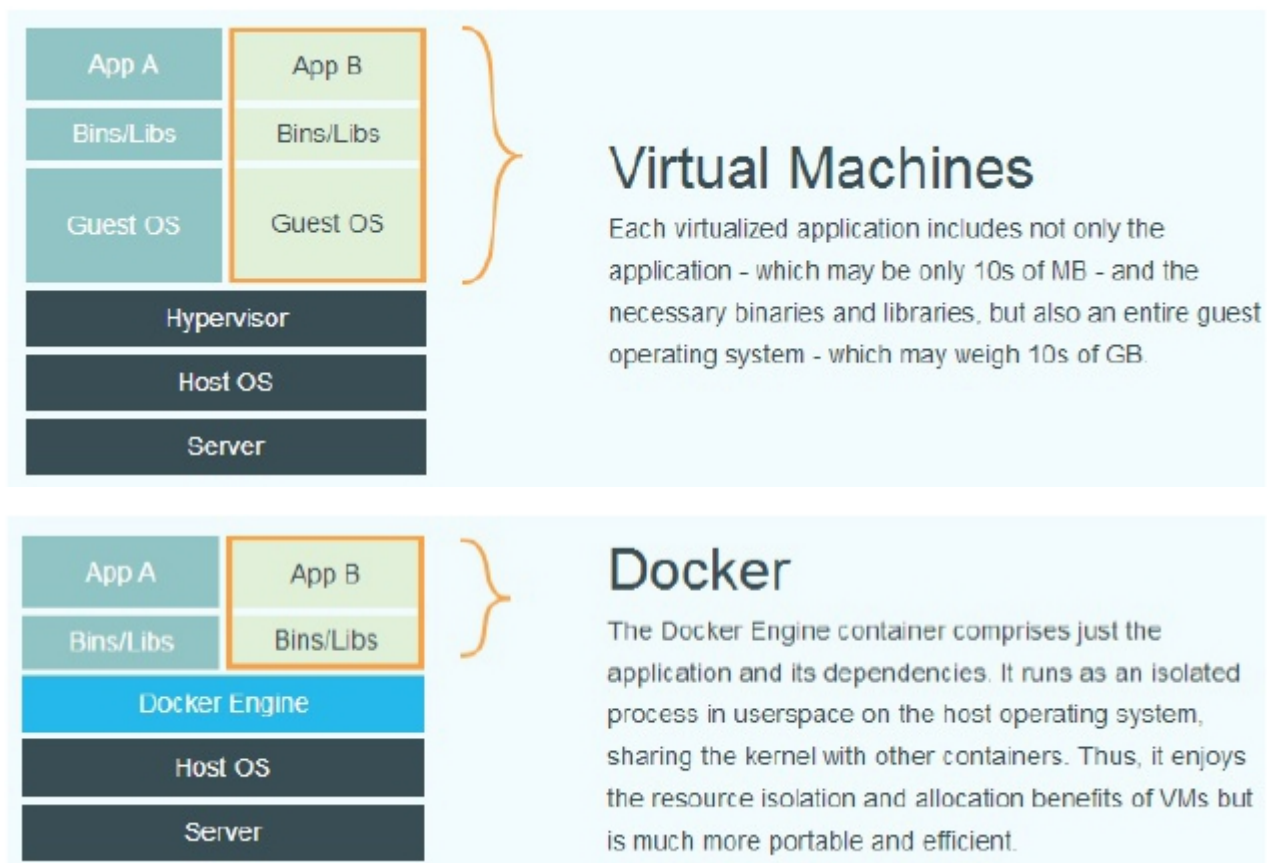
Docker 是一个开源项目，诞生于 2013 年初，最初是 dotCloud 公司内部的一个业余项目。它基于 Google 公司推出的 Go 语言实现。项目后来加入了 Linux 基金会，遵从了 Apache 2.0 协议，项目代码在 [GitHub](#) 上进行维护。

Docker 自开源后受到广泛的关注和讨论，以至于 dotCloud 公司后来都改名为 DockerInc。Redhat 已经在其 RHEL6.5 中集中支持 Docker；Google 也在其 PaaS 产品中广泛应用。

Docker 项目的目标是实现轻量级的操作系统虚拟化解决方案。Docker 的基础是 Linux 容器（LXC）等技术。

在 LXC 的基础上 Docker 进行了进一步的封装，让用户不需要去关心容器的管理，使得操作更为简便。用户操作 Docker 的容器就像操作一个快速轻量级的虚拟机一样简单。

下面的图片比较了 Docker 和传统虚拟化方式的不同之处，可见容器是在操作系统层面上实现虚拟化，直接复用本地主机的操作系统，而传统方式则是在硬件层面实现。



# 为什么要使用 Docker？

作为一种新兴的虚拟化方式，Docker 跟传统的虚拟化方式相比具有众多的优势。

首先，Docker 容器的启动可以在秒级实现，这相比传统的虚拟机方式要快得多。其次，Docker 对系统资源的利用率很高，一台主机上可以同时运行数千个 Docker 容器。

容器除了运行其中应用外，基本不消耗额外的系统资源，使得应用的性能很高，同时系统的开销尽量小。传统虚拟机方式运行 10 个不同的应用就要起 10 个虚拟机，而Docker 只需要启动 10 个隔离的应用即可。

具体说来，Docker 在如下几个方面具有较大的优势。

## 更快速的交付和部署

对开发和运维（devop）人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。

开发者可以使用一个标准的镜像来构建一套开发容器，开发完成之后，运维人员可以直接使用这个容器来部署代码。Docker 可以快速创建容器，快速迭代应用程序，并让整个过程全程可见，使团队中的其他成员更容易理解应用程序是如何创建和工作的。Docker 容器很轻很快！容器的启动时间是秒级的，大量地节约开发、测试、部署的时间。

## 更高效的虚拟化

Docker 容器的运行不需要额外的 hypervisor 支持，它是内核级的虚拟化，因此可以实现更高的性能和效率。

## 更轻松的迁移和扩展

Docker 容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等。这种兼容性可以让用户把一个应用程序从一个平台直接迁移到另外一个。

## 更简单的管理

使用 Docker，只需要小小的修改，就可以替代以往大量的更新工作。所有的修改都以增量的方式被分发和更新，从而实现自动化并且高效的管理。

## 对比传统虚拟机总结

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

# 基本概念

---

- 

- 

Docker 包括三个基本概念

- 

- 镜像 ( Image )

- 容器 ( Container )

- 仓库 ( Repository )

理解了这三个概念，就理解了 Docker 的整个生命周期。

# Docker 镜像

---

Docker 镜像就是一个只读的模板。

例如：一个镜像可以包含一个完整的 ubuntu 操作系统环境，里面仅安装了 Apache 或用户需要的其它应用程序。

镜像可以用来创建 Docker 容器。

Docker 提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。



# Docker 容器

---

Docker 利用容器来运行应用。

容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。

可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

\*注：镜像是只读的，容器在启动的时候创建一层可写层作为最上层。

# Docker 仓库

---

仓库是集中存放镜像文件的场所。有时候会把仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。

仓库分为公开仓库（Public）和私有仓库（Private）两种形式。

最大的公开仓库是 [Docker Hub](#)，存放了数量庞大的镜像供用户下载。国内的公开仓库包括 [DockerPool](#) 等，可以提供大陆用户更稳定快速的访问。

当然，用户也可以在本地网络内创建一个私有仓库。

当用户创建了自己的镜像之后就可以使用 `push` 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上 `pull` 下来就可以了。

\*注：Docker 仓库的概念跟 [Git](#) 类似，注册服务器可以理解为 GitHub 这样的托管服务。

# 安装

---

官方网站上有各种环境下的 [安装指南](#)，这里主要介绍下Ubuntu和CentOS系列的安装。

# Ubuntu 系列安装 Docker

## 通过系统自带包安装

Ubuntu 14.04 版本系统中已经自带了 Docker 包，可以直接安装。

```
$ apt-get update
$ apt-get install -y docker.io
$ ln -sf /usr/bin/docker.io /usr/local/bin/docker
$ sed -i '$a complete -F _docker docker' /etc/bash_completion.d/docker.io
```

如果使用操作系统自带包安装 Docker，目前安装的版本是比较旧的 0.9.1。要安装更新的版本，可以通过使用 Docker 源的方式。

## 通过 Docker 源安装最新版本

要安装最新的 Docker 版本，首先需要安装 apt-transport-https 支持，之后通过添加源来安装。

```
$ apt-get install apt-transport-https
$ apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 36A1D7869245C8950F966E92D8576A8BA8
$ bash -c "echo deb https://get.docker.io/ubuntu docker main > /etc/apt/sources.list.d/docker.list"
$ apt-get update
$ apt-get install lxc-docker
```

## 14.04 之前版本

如果是较低版本的 Ubuntu 系统，需要先更新内核。

```
$ sudo apt-get update
$ sudo apt-get install linux-image-generic-lts-raring linux-headers-generic-lts-raring
$ sudo reboot
```

然后重复上面的步骤即可。

安装之后启动 Docker 服务。

```
$ sudo service docker start
```

# CentOS 系列安装 Docker

Docker 支持 CentOS6 及以后的版本。

## CentOS6

对于 CentOS6，可以使用 [EPEL](#) 库安装 Docker，命令如下

```
$ sudo yum install http://mirrors.yun-idc.com/epel/6/i386/epel-release-6-8.noarch.rpm
$ sudo yum install docker-io
```

## CentOS7

CentOS7 系统 `CentOS-Extras` 库中已带 Docker，可以直接安装：

```
$ sudo yum install docker
```

安装之后启动 Docker 服务，并让它随系统启动自动加载。

```
$ sudo service docker start
$ sudo chkconfig docker on
```

# Docker 镜像

---

在之前的介绍中，我们知道镜像是 Docker 的三大组件之一。

Docker 运行容器前需要本地存在对应的镜像，如果镜像不存在本地，Docker 会从镜像仓库下载（默认是 Docker Hub 公共注册服务器中的仓库）。

本章将介绍更多关于镜像的内容，包括：

- 
- 从仓库获取镜像；  
管理本地主机上的镜像；  
介绍镜像实现的基本原理。

# 获取镜像

可以使用 `docker pull` 命令来从仓库获取所需要的镜像。

下面的例子将从 Docker Hub 仓库下载一个 Ubuntu 12.04 操作系统的镜像。

```
$ sudo docker pull ubuntu:12.04
Pulling repository ubuntu
ab8e2728644c: Pulling dependent layers
511136ea3c5a: Download complete
5f0ffaa9455e: Download complete
a300658979be: Download complete
904483ae0c30: Download complete
ffdaafd1ca50: Download complete
d047ae21eeaf: Download complete
```

下载过程中，会输出获取镜像的每一层信息。

该命令实际上相当于 `$ sudo docker pull registry.hub.docker.com/ubuntu:12.04` 命令，即从注册服务器 `registry.hub.docker.com` 中的 `ubuntu` 仓库来下载标记为 `12.04` 的镜像。

有时候官方仓库注册服务器下载较慢，可以从其他仓库下载。从其它仓库下载时需要指定完整的仓库注册服务器地址。例如

```
$ sudo docker pull dl.dockerpool.com:5000/ubuntu:12.04
Pulling dl.dockerpool.com:5000/ubuntu
ab8e2728644c: Pulling dependent layers
511136ea3c5a: Download complete
5f0ffaa9455e: Download complete
a300658979be: Download complete
904483ae0c30: Download complete
ffdaafd1ca50: Download complete
d047ae21eeaf: Download complete
```

完成后，即可随时使用该镜像了，例如创建一个容器，让其中运行 `bash` 应用。

```
$ sudo docker run -t -i ubuntu:12.04 /bin/bash
root@fe7fc4bd8fc9:/#
```

# 列出本地镜像

使用 `docker images` 显示本地已有的镜像。

```
$ sudo docker images
REPOSITORY      TAG        IMAGEID       CREATED        VIRTUAL SIZE
ubuntu          12.04      74fe38d11401 4 weeks ago    209.6 MB
ubuntu          precise   74fe38d11401 4 weeks ago    209.6 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
...
```

在列出信息中，可以看到几个字段信息

- 来自于哪个仓库，比如 `ubuntu`
- 镜像的标记，比如 `14.04`
- 它的 ID 号（唯一）
- 创建时间
- 镜像大小

其中镜像的 ID 唯一标识了镜像，注意到 `ubuntu:14.04` 和 `ubuntu:trusty` 具有相同的镜像 ID，说明它们实际上是同一镜像。

TAG 信息用来标记来自同一个仓库的不同镜像。例如 `ubuntu` 仓库中有多个镜像，通过 TAG 信息来区分发行版本，例如 `10.04`、`12.04`、`12.10`、`13.04`、`14.04` 等。例如下面的命令指定使用镜像 `ubuntu:14.04` 来启动一个容器。

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

如果不指定具体的标记，则默认使用 `latest` 标记信息。



# 创建镜像

创建镜像有很多方法，用户可以从 DockerHub 获取已有镜像并更新，也可以利用本地文件系统创建一个。

## 修改已有镜像

先使用下载的镜像启动容器。

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

注意：记住容器的 ID，稍后还会用到。

在容器中添加 json 和 gem 两个应用。

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

当结束后，我们使用 exit 来退出，现在我们的容器已经被我们改变了，使用 docker commit 命令来提交更新后的副本。

```
$ sudo docker commit -m "Added json gem" -a "Docker Newbee" 0b2616b0e5a8 ouruser/sinatra:v2
4f177bd27a9ff0f6dc2a830403925b5360bfe0b93d476f7fc3231110e7f71b1c
```

其中，-m 来指定提交的说明信息，跟我们使用的版本控制工具一样；-a 可以指定更新的用户信息；之后是用来创建镜像的容器的 ID；最后指定目标镜像的仓库名和 tag 信息。创建成功后会返回这个镜像的 ID 信息。

使用 docker images 来查看新创建的镜像。

```
$ sudo docker images
REPOSITORY          TAG         IMAGEID         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
```

之后，可以使用新的镜像来启动容器

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

## 利用 Dockerfile 来创建镜像

使用 docker commit 来扩展一个镜像比较简单，但它不容易在一个团队中分享它。我们可以使用 docker build 来创建一个新的镜像。为此，首先需要创建一个 Dockerfile，包含一些如何创建镜像的指令。

新建一个目录和一个 Dockerfile

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

Dockerfile 中每一条指令都创建镜像的一层，例如：

```
# This is a comment
FROM ubuntu:14.04
MAINTAINER Docker Newbee <newbee@docker.com>
RUN apt-get -qq update
RUN apt-get -qq install ruby ruby-dev
RUN gem install sinatra
```

Dockerfile 基本的语法是

- 使用 # 来注释
- FROM 指令告诉 Docker 使用哪个镜像作为基础
- 接着是维护者的信息
- RUN 开头的指令会在创建中运行，比如安装一个软件包，在这里使用 apt-get 来安装了一些软件

编写完成 Dockerfile 后可以使用 docker build 来生成镜像。

```
$ 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 -qq 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
```

其中 `-t` 标记来添加 tag，指定新的镜像的用户信息。 `."` 是 Dockerfile 所在的路径（当前目录），也可以替换为一个具体的 Dockerfile 的路径。

可以看到 build 进程在执行操作。它要做的第一件事情就是上传这个 Dockerfile 内容，因为所有的操作都要依据 Dockerfile 来进行。然后，Dockerfile 中的指令被一条一条的执行。每一步都创建了一个新的容器，在容器中执行指令并提交修改（就跟之前介绍过的 `docker commit` 一样）。当所有的指令都执行完毕之后，返回了最终的镜像 id。所有的中间步骤所产生的容器都被删除和清理了。

\*注意一个镜像不能超过 127 层

此外，还可以利用 `ADD` 命令复制本地文件到镜像；用 `EXPOSE` 命令来向外部开放端口；用 `CMD` 命令来描述容器启动后运行的程序等。例如

```
# put my local web site in myApp folder to /var/www
ADD myApp /var/www
# expose httpd port
EXPOSE 80
# the command to run
CMD ["/usr/sbin/apachectl", "-D", "FOREGROUND"]
```

现在可以利用新创建的镜像来启动一个容器。

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

还可以用 `docker tag` 命令来修改镜像的标签。

```
$ sudo docker tag 5db5f8471261 ouruser/sinatra:devel
$ sudo docker images ouruser/sinatra
REPOSITORY          TAG         IMAGEID         CREATED          VIRTUALSIZE
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
```

\*注：更多用法，请参考 [Dockerfile](#) 章节。

## 从本地文件系统导入

要从本地文件系统导入一个镜像，可以使用 `openvz`（容器虚拟化的先锋技术）的模板来创建：`openvz` 的模板下载地址为 **错误！超链接引用无效。错误！超链接引用无效。**

比如，先下载了一个 `ubuntu-14.04` 的镜像，之后使用以下命令导入：

```
sudo cat ubuntu-14.04-x86_64-minimal.tar.gz | docker import - ubuntu:14.04
```

然后查看新导入的镜像。

docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	14.04	05ac7c0b9383	17 seconds ago	215.5 MB

## 上传镜像

用户可以通过 `docker push` 命令，把自己创建的镜像上传到仓库中来共享。例如，用户在 Docker Hub 上完成注册后，可以推送自己的镜像到仓库中。

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

# Dockerfile

---

使用 Dockerfile 可以允许用户创建自定义的镜像。

## 基本结构

Dockerfile 由一行行命令语句组成，并且支持以 `#` 开头的注释行。

一般的，Dockerfile 分为四部分：基础镜像信息、维护者信息、镜像操作指令和容器启动时执行指令。

例如

```
This dockerfile uses the ubuntu image
VERSION 2 - EDITION 1
Author: docker_user
Command format: Instruction [arguments / command] ..

# Base image to use, this must be set as the first line
FROM ubuntu

# Maintainer: docker_user <docker_user at email.com> (@docker_user)
MAINTAINER docker_user docker_user@email.com

# Commands to update the image
RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >> /etc/apt/sources.list
RUN apt-get update && apt-get install -y nginx
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf

# Commands when creating a new container
CMD /usr/sbin/nginx
```

其中，一开始必须指明所基于的镜像名称，接下来推荐说明维护者信息。

后面则是镜像操作指令，例如 `RUN` 指令，`RUN` 指令将对镜像执行跟随的命令。每运行一条 `RUN` 指令，镜像添加新的一层，并提交。

最后是 `CMD` 指令，来指定运行容器时的操作命令。

下面是一个更复杂的例子

```

# Nginx
#
# VERSION            0.0.1

FROM ubuntu
MAINTAINER Victor Vieux <victor@docker.com>

RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server

# Firefox over VNC
#
# VERSION            0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir /.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example
#
# VERSION            0.1

FROM ubuntu
RUN echo foo > bar
# Will output something like ==> 907ad6c2736f

FROM ubuntu
RUN echo moo > oink
# Will output something like ==> 695d7793cbe4

# You ll now have two images, 907ad6c2736f with /bar, and 695d7793cbe4 with
# /oink.

```

指令的一般格式为 `INSTRUCTION arguments`，指令包括 `FROM`、`MAINTAINER`、`RUN` 等。

## FROM

格式为 `FROM <image>` 或 `FROM <image>:<tag>`。

第一条指令必须为 `FROM` 指令。并且，如果在同一个 Dockerfile 中创建多个镜像时，可以使用多个 `FROM` 指令（每个镜像一次）。

## MAINTAINER

格式为 `MAINTAINER <name>`，指定维护者信息。

## RUN

格式为 `RUN <command>` 或 `RUN ["executable", "param1", "param2"]`。

前者将在 `shell` 终端中运行命令，即 `/bin/sh -c`；后者则使用 `exec` 执行。指定使用其它终端可以通过第二种方式实现，例如 `RUN ["/bin/bash", "-c", "echo hello"]`。

每条 `RUN` 指令将在当前镜像基础上执行指定命令，并提交为新的镜像。当命令较长时可以使用 `\` 来换行。

## CMD

支持三种格式

- `CMD ["executable", "param1", "param2"]` 使用 `exec` 执行，推荐方式；
- `CMD command param1 param2` 在 `/bin/sh` 中执行，提供给需要交互的应用；
- `CMD ["param1", "param2"]` 提供给 `ENTRYPOINT` 的默认参数；

指定启动容器时执行的命令，每个 Dockerfile 只能有一条 `CMD` 命令。如果指定了多条命令，只有最后一条会被执行。

如果用户启动容器时候指定了运行的命令，则会覆盖掉 `CMD` 指定的命令。

## EXPOSE

格式为 `EXPOSE <port> [<port>...]`。

告诉 Docker 服务端容器暴露的端口号，供互联系统使用。在启动容器时需要通过 `-P`，Docker 主机会自动分配一个端口转发到指定的端口。

## ENV

格式为 `ENV <key> <value>`。指定一个环境变量，会被后续 `RUN` 指令使用，并在容器运行时保持。

例如



```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJC /usr/src/postgress && ...
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

## ADD

格式为 `ADD <src> <dest>` 。

该命令将复制指定的 `<src>` 到容器中的 `<dest>` 。其中 `<src>` 可以是 Dockerfile 所在目录的一个相对路径；也可以是一个 URL；还可以是一个 `tar` 文件（自动解压为目录）。则。

## COPY

格式为 `COPY <src> <dest>` 。

复制本地主机的 `<src>` （为 Dockerfile 所在目录的相对路径）到容器中的 `<dest>` 。

当使用本地目录为源目录时，推荐使用 `COPY` 。

## ENTRYPOINT

两种格式：

- `ENTRYPOINT["executable", "param1", "param2"]`
- `ENTRYPOINT command param1 param2` （shell 中执行）。

配置容器启动后执行的命令，并且不可被 `dockerrun` 提供的参数覆盖。

每个 Dockerfile 中只能有一个 `ENTRYPOINT`，当指定多个时，只有最后一个起效。

## VOLUME

格式为 `VOLUME ["/data"]` 。

创建一个可以从本地主机或其他容器挂载的挂载点，一般用来存放数据库和需要保持的数据等。

## USER

格式为 `USER daemon` 。

指定运行容器时的用户名或 UID，后续的 `RUN` 也会使用指定用户。

当服务不需要管理员权限时，可以通过该命令指定运行用户。并且可以在之前创建所需要的用户，例如：`RUN groupadd -r postgres && useradd -r -g postgres postgres`。要临时获取管理员权限可以使用 `gosu`，而不推荐 `sudo`。

## WORKDIR

格式为 `WORKDIR /path/to/workdir` 。

为后续的 `RUN` 、 `CMD` 、 `ENTRYPOINT` 指令配置工作目录。

可以使用多个 `WORKDIR` 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。例如

```
WORKDIR/a
WORKDIRb
WORKDIRc
RUNpwd
```

则最终路径为 `/a/b/c` 。

## ONBUILD

格式为 `ONBUILD [INSTRUCTION]` 。

配置当所创建的镜像作为其它新创建镜像的基础镜像时，所执行的操作指令。

例如，`Dockerfile` 使用如下的内容创建了镜像 `image-A` 。

```
[...]
ONBUILD ADD ./app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

如果基于 `image-A` 创建新的镜像时，新的`Dockerfile`中使用 `FROM image-A` 指定基础镜像时，会自动执行 `ONBUILD` 指令内容，等价于在后面添加了两条指令。

```
FROM image-A

#Automatically run the following
ADD ./app/src
RUN /usr/local/bin/python-build --dir /app/src
```

使用 `ONBUILD` 指令的镜像，推荐在标签中注明，例如 `ruby:1.9-onbuild` 。

## 创建镜像

---

编写完成 Dockerfile 之后，可以通过 `docker build` 命令来创建镜像。

基本的格式为 `docker build [选项] 路径`，该命令将读取指定路径下（包括子目录）的 Dockerfile，并将该路径下所有内容发送给 Docker 服务端，由服务端来创建镜像。因此一般建议放置 Dockerfile 的目录为空目录。也可以通过 `.dockerignore` 文件（每一行添加一条匹配模式）来让 Docker 忽略路径下的目录和文件。

要指定镜像的标签信息，可以通过 `-t` 选项，例如

```
$ sudo docker build -t myrepo/myapp/tmp/test1/
```

Docker 底层的核心技术包括 Linux 上的名字空间 ( Namespaces )、控制组 ( Control groups )、Union 文件系统 ( Union file systems ) 和容器格式 ( Container format )。

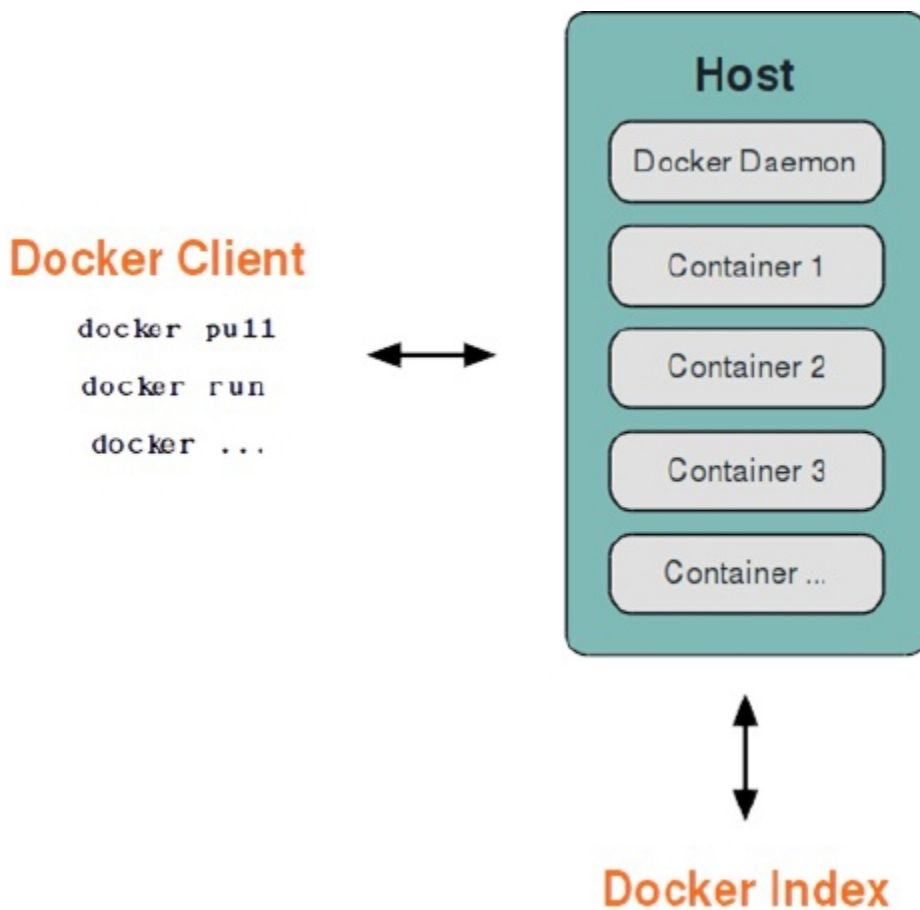
我们知道，传统的虚拟机通过在宿主主机中运行 hypervisor 来模拟一整套完整的硬件环境提供给虚拟机的操作系统。虚拟机系统看到的环境是可限制的，也是彼此隔离的。这种直接的做法实现了对资源最完整的封装，但很多时候往往意味着系统资源的浪费。例如，以宿主机和虚拟机系统都为 Linux 系统为例，虚拟机中运行的应用其实可以利用宿主机系统中的运行环境。

我们知道，在操作系统中，包括内核、文件系统、网络、PID、UID、IPC、内存、硬盘、CPU 等等，所有的资源都是应用进程直接共享的。要想实现虚拟化，除了要实现内存、CPU、网络IO、硬盘IO、存储空间等的限制外，还要实现文件系统、网络、PID、UID、IPC等等的相互隔离。前者相对容易实现一些，后者则需要宿主机系统的深入支持。

随着 Linux 系统对于名字空间功能的完善实现，程序员已经可以实现上面的所有需求，让某些进程在彼此隔离的名字空间中运行。大家虽然都共用一个内核和某些运行时环境（例如一些系统命令和系统库），但是彼此却看不到，都以为系统中只有自己的存在。这种机制就是容器 ( Container )，例如名字空间来做权限的隔离控制，利用 cgroups 来做资源分配。

## 基本架构

Docker 采用了 C/S架构，包括客户端和服务端。 Dockerdaemon 作为服务端接受来自客户的请求，并处理这些请求（创建、运行、分发容器）。 客户端和服务端既可以运行在一个机器上，也可通过 socket 或者 RESTfulAPI 来进行通信。



Dockerdaemon 一般在宿主主机后台运行，等待接收来自客户端的消息。 Docker 客户端则为用户提供一系列可执行命令，用户用这些命令实现跟 Dockerdaemon 交互。

# 名字空间

---

名字空间是 Linux 内核一个强大的特性。每个容器都有自己单独的名字空间，运行在其中的应用都像是在独立的操作系统中运行一样。名字空间保证了容器之间彼此互不影响。

## pid 名字空间

不同用户的进程就是通过 pid 名字空间隔离开的，且不同名字空间中可以有相同 pid。所有的 LXC 进程在 Docker 中的父进程为 Docker 进程，每个 LXC 进程具有不同的名字空间。同时由于允许嵌套，因此可以很方便的实现嵌套的 Docker 容器。

## net 名字空间

有了 pid 名字空间，每个名字空间中的 pid 能够相互隔离，但是网络端口还是共享 host 的端口。网络隔离是通过 net 名字空间实现的，每个 net 名字空间有独立的网络设备、IP 地址、路由表、/proc/net 目录。这样每个容器的网络就能隔离开来。Docker 默认采用 veth 的方式，将容器中的虚拟网卡同 host 上的一个 Docker 网桥 docker0 连接在一起。

## ipc 名字空间

容器中进程交互还是采用了 Linux 常见的进程间交互方法(interprocess communication - IPC)，包括信号量、消息队列和共享内存等。然而同 VM 不同的是，容器的进程间交互实际上还是 host 上具有相同 pid 名字空间中的进程间交互，因此需要在 IPC 资源申请时加入名字空间信息，每个 IPC 资源有一个唯一的 32 位 id。

## mnt 名字空间

类似 chroot，将一个进程放到一个特定的目录执行。mnt 名字空间允许不同名字空间的进程看到的文件结构不同，这样每个名字空间中的进程所看到的文件目录就被隔离开了。同 chroot 不同，每个名字空间中的容器在 /proc/mounts 的信息只包含所在名字空间的 mount point。

## uts 名字空间

UTS("UNIX Time-sharing System") 名字空间允许每个容器拥有独立的 hostname 和 domain name，使其在网络上可以被视作一个独立的节点而非主机上的一个进程。

## user 名字空间

每个容器可以有不同的用户和组 id，也就是说可以在容器内用容器内部的用户执行程序而非主机上的用户。

\*注：关于 Linux 上的名字空间，[这篇文章](#) 介绍的很好。

# 控制组

---

控制组 ( [cgroups](#) ) 是 Linux 内核的一个特性，主要用来对共享资源进行隔离、限制、审计等。只有能控制分配到容器的资源，才能避免当多个容器同时运行时的对系统资源的竞争。

控制组技术最早是由 Google 的程序员 2006 年起提出，Linux 内核自 2.6.24 开始支持。

控制组可以提供对容器内内存、CPU、磁盘 IO 等资源的限制和计费管理。

# Union 文件系统

---

Union文件系统 ( [UnionFS](#) ) 是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)。

Union 文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。

另外，不同 Docker 容器就可以共享一些基础的文件系统层，同时再加上自己独有的改动层，大大提高了存储的效率。

Docker 中使用的 AUFS ( AnotherUnionFS ) 就是一种 UnionFS。 AUFS 支持为每一个成员目录（类似 Git 的分支）设定只读（readonly）、读写（readwrite）和写出（whiteout-able）权限，同时 AUFS 里有一个类似分层的概念，对只读权限的分支可以逻辑上进行增量地修改(不影响只读部分的)。

Docker 目前支持的 Union 文件系统种类包括 AUFS,btrfs,vfs 和 DeviceMapper。



## 容器格式

---

最初，Docker 采用了 LXC 中的容器格式。自 1.20 版本开始，Docker 也开始支持新的 libcontainer 格式，并作为默认选项。

对更多容器格式的支持，还在进一步的发展中。

Docker 的网络实现其实就是利用了 Linux 上的网络名字空间和虚拟网络设备（特别是 veth pair）。建议先熟悉了解这两部分的基本概念再阅读本章。

## 基本原理

首先，要实现网络通信，机器需要至少一个网络接口（物理接口或虚拟接口）来收发数据包；此外，如果不同子网之间要进行通信，需要路由机制。

Docker 中的网络接口默认都是虚拟的接口。虚拟接口的优势之一是转发效率较高。Linux 通过在内核中进行数据复制来实现虚拟接口之间的数据转发，发送接口的发送缓存中的数据包被直接复制到接收接口的接收缓存中。对于本地系统和容器内系统看来就像是一个正常的以太网卡，只是它不需要真正同外部网络设备通信，速度要很快。

Docker 容器网络就利用了这项技术。它在本地主机和容器内分别创建一个虚拟接口，并让它们彼此连通（这样的一对接口叫做 vethpair）。

## 创建网络参数

Docker 创建一个容器的时候，会执行如下操作：

- 创建一对虚拟接口，分别放到本地主机和新容器中；
- 本地主机一端桥接到默认的 docker0 或指定网桥上，并具有一个唯一的名字，如 veth65f9；
- 容器一端放到新容器中，并修改名字作为 eth0。这个接口只在容器的名字空间可见；  
从网桥可用地址段中获取一个空闲地址分配给容器的 eth0，并配置默认路由到桥接网卡 veth65f9。

完成这些之后，容器就可以使用 eth0 虚拟网卡来连接其他容器和其他网络。

可以在 dockerrun 的时候通过 --net 参数来指定容器的网络配置，有4个可选值：

- --net=bridge 这个是默认值，连接到默认的网桥。  
--net=host 告诉 Docker 不要将容器网络放到隔离的名字空间中，即不要容器化容器内的网络。此时容器使用本地主机的网络，它拥有完全的本地主机接口访问权限。容器进程可以跟主机其它 root 进程一样可以打开低范围的端口，可以访问本地网络服务比如 D-bus，还可以让容器做一些影响整个主机系统的事情，比如重启主机。因此使用这个选项的时候要非常小心。如果进一步的使用 --
- privileged=true，容器会被允许直接配置主机的网络堆栈。  
--net=container:NAME\_or\_ID 让 Docker 将新建容器的进程放到一个已存在容器的网络栈中，新容器进程有自己的文件系统、进程列表和资源限制，但会和已存在的容器共享 IP 地址和端口等网络资源，两者进程可以直接通过 lo 环回接口通信。  
--net=none 让 Docker 将新容器放到隔离的网络栈中，但是不进行网络配置。之后，用户可以自己进行配置。

## 网络配置细节

用户使用 --net=none 后，可以自行配置网络，让容器达到跟平常一样具有访问网络的权限。通过这个过程，可以了解 Docker 配置网络的细节。

首先，启动一个 /bin/bash 容器，指定 --net=none 参数。

```
$ sudo docker run -i -t --rm --net=none base/bin/bash
root@63f36fc01b5f:/#
```

在本地主机查找容器的进程 id，并为它创建网络命名空间。

```
$ sudo docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid
```






检查桥接网卡的 IP 和子网掩码信息。

```
$ ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...
```

创建一对 “veth pair” 接口 A 和 B，绑定 A 到网桥 docker0，并启用它

```
$ sudo ip link add A type veth peer name B
$ sudo brctl addif docker0 A
$ sudo ip link set A up
```

将B放到容器的网络命名空间，命名为 eth0，启动它并配置一个可用 IP（桥接网段）和默认网关。

```
 link set B netns $pid
 netns exec $pid ip link set dev B name eth0
 netns exec $pid ip link set eth0 up
 netns exec $pid ip addr add 172.17.42.99/16 dev eth0
 netns exec $pid ip route add default via 172.17.42.1
```

以上，就是 Docker 配置网络的具体过程。

当容器结束后，Docker 会清空容器，容器内的 eth0 会随网络命名空间一起被摧毁，A 接口也被自动从 docker0 卸载。

此外，用户可以使用 `ip netns exec` 命令来在指定网络名字空间中进行配置，从而配置容器内的网络。

# Docker 命令查询

---

## 基本语法

```
docker[OPTIONS] COMMAND [arg...]
```

一般来说，Docker 命令可以用来管理 daemon，或者通过 CLI 命令管理镜像和容器。可以通过 `man docker` 来查看这些命令。

## 选项

`-D=true|false`  
使用 debug 模式。默认为 false。

`-H, --host=[unix:///var/run/docker.sock]:tcp://[host:port]`来绑定或者 `unix://[path/to/socket]` 来使用。  
在 daemon 模式下绑定的 socket , 通过一个或多个 `tcp://host:port,unix:///path/to/socket,fd://* or fd://socketfd` 来指定。

`--api-enable-cors=true|false`  
在远端 API 中启用 CORS 头。缺省为 false。

`-b=""`  
将容器挂载到一个已存在的网桥上。指定为 'none' 时则禁用容器的网络。

`--bip=""`  
让动态创建的 docker0 采用给定的 CIDR 地址; 与 -b 选项互斥。

`-d=true|false`  
使用 daemon 模式。缺省为 false。

`--dns=""`  
让 Docker 使用给定的 DNS 服务器。

`-g=""`  
指定 Docker 运行时的 root 路径。缺省为 /var/lib/docker。

`--icc=true|false`  
启用容器间通信。默认为 true。

`--ip=""`  
绑定端口时候的默认 IP 地址。缺省为 0.0.0.0。

`--iptables=true|false`  
禁止 Docker 添加 iptables 规则。缺省为 true。

`--mtu=VALUE`  
指定容器网络的 mtu。缺省为 1500。

`-p=""`  
指定 daemon 的 PID 文件路径。缺省为 /var/run/docker.pid。

`-s=""`  
强制 Docker 运行时使用给定的存储驱动。

`-v=true|false`  
输出版本信息并退出。缺省值为 false。

`--selinux-enabled=true|false`  
启用 SELinux 支持。缺省值为 false。SELinux 目前不支持 BTRFS 存储驱动。

## 命令

Docker 的命令可以采用 `docker-CMD` 或者 `dockerCMD` 的方式执行。两者一致。

`docker-attach(1)`  
依附到一个正在运行的容器中。

`docker-build(1)`  
从一个 Dockerfile 创建一个镜像

`docker-commit(1)`  
从一个容器的修改中创建一个新的镜像

`docker-cp(1)`  
从容器中复制文件到宿主系统中

`docker-diff(1)`  
检查一个容器文件系统的修改

`docker-events(1)`  
从服务端获取实时的事件

`docker-export(1)`  
导出容器内容为一个 tar 包

`docker-history(1)`  
显示一个镜像的历史

`docker-images(1)`  
列出存在的镜像

`docker-import(1)`  
导入一个文件（典型为 tar 包）路径或目录来创建一个镜像

`docker-info(1)`  
显示一些相关的系统信息

`docker-inspect(1)`  
显示一个容器的底层具体信息。

`docker-kill(1)`  
关闭一个运行中的容器（包括进程和所有资源）

`docker-load(1)`  
从一个 tar 包中加载一个镜像

`docker-login(1)`  
注册或登录到一个 Docker 的仓库服务器

`docker-logout(1)`  
从 Docker 的仓库服务器登出

`docker-logs(1)`  
获取容器的 log 信息

`docker-pause(1)`  
暂停一个容器中的所有进程

`docker-port(1)`  
查找一个 nat 到一个私有网口的公共口

`docker-ps(1)`  
列出容器

`docker-pull(1)`  
从一个Docker的仓库服务器下拉一个镜像或仓库

`docker-push(1)`  
将一个镜像或者仓库推送到一个 Docker 的注册服务器

`docker-restart(1)`  
重启一个运行中的容器

`docker-rm(1)`  
删除给定的若干个容器

`docker-rmi(1)`  
删除给定的若干个镜像

`docker-run(1)`  
创建一个新容器，并在其中运行给定命令

`docker-save(1)`  
保存一个镜像为 tar 包文件

`docker-search(1)`  
在 Docker index 中搜索一个镜像

`docker-start(1)`  
启动一个容器

`docker-stop(1)`  
终止一个运行中的容器

`docker-tag(1)`  
为一个镜像打标签

`docker-top(1)`  
查看一个容器中的正在运行的进程信息

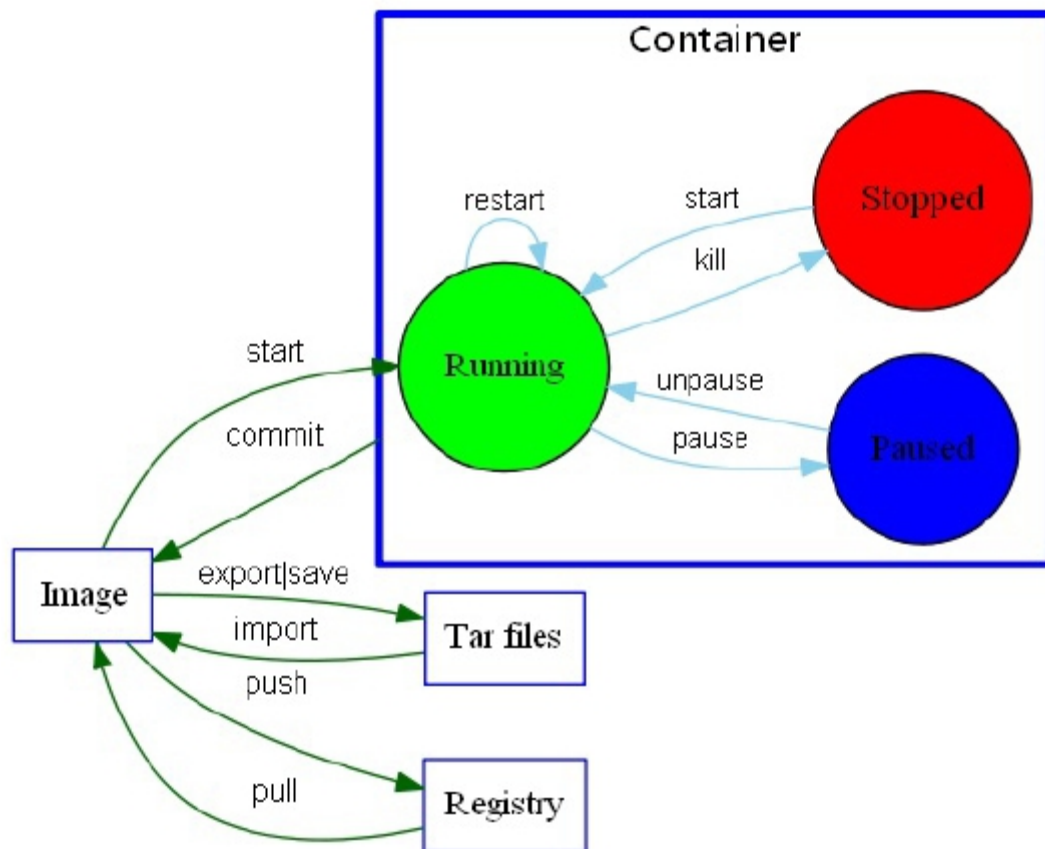
`docker-unpause(1)`  
将一个容器内所有的进程从暂停状态中恢复

`docker-version(1)`  
输出 Docker 的版本信息

`docker-wait(1)`  
阻塞直到一个容器终止，然后输出它的退出符

## 一张图总结 **Docker** 的命令

---





# 常见仓库介绍

---

本章将介绍常见的一些仓库和镜像的功能，使用方法和生成它们的 Dockerfile 等。包括 Ubuntu、CentOS、MySQL、MongoDB、Redis、Nginx、Wordpress、Node.js 等。

## 基本信息

[Ubuntu](#) 是流行的 Linux 发行版，其自带软件版本往往较新一些。该仓库提供了 Ubuntu从12.04 ~ 14.10 各个版本的镜像。

## 使用方法

默认会启动一个最小化的 Ubuntu 环境。

```
$ sudo docker run --name some-ubuntu -i -t ubuntu  
root@523c70904d54:/#
```

## Dockerfile

- [12.04 版本](#)
- [14.04 版本](#)
- [14.10 版本](#)
-

## 基本信息

[CentOS](#) 是流行的 Linux 发行版，其软件包大多跟 RedHat 系列保持一致。该仓库提供了 CentOS 从 5 ~ 7 各个版本的镜像。

## 使用方法

默认会启动一个最小化的 CentOS 环境。

```
$ sudo docker run --name some-centos -i -t centos bash
bash-4.2#
```

## Dockerfile

- [CentOS 5 版本](#)
- [CentOS 6 版本](#)
- [CentOS 7 版本](#)

## 基本信息

MySQL 是开源的关系数据库实现。 该仓库提供了 MySQL 各个版本的镜像，包括 5.6 系列、5.7 系列等。

## 使用方法

默认会在 3306 端口启动数据库。

```
$ sudo docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
```

之后就可以使用其它应用来连接到该容器。

```
$ sudo docker run --name some-app --link some-mysql:mysql -d application-that-uses-mysql
```

或者通过 mysql 。

```
$ sudo docker run -it --link some-mysql:mysql --rm mysql sh -c 'exec mysql -h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYS
```



## Dockerfile

- 5.6 版本
- 5.7 版本

## 基本信息

MongoDB 是开源的 NoSQL 数据库实现。该仓库提供了 MongoDB 2.2 ~ 2.7 各个版本的镜像。

## 使用方法

默认会在 27017 端口启动数据库。

```
$ sudo docker run --name some-mongo -d mongo
```

使用其他应用连接到容器，可以用

```
$ sudo docker run --name some-app --link some-mongo:mongo -d application-that-uses-mongo
```

或者通过 mongo

```
$ sudo docker run -it --link some-mongo:mongo --rm mongo sh -c 'exec mongo "$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT"'
```

## Dockerfile

- [2.2 版本](#)
- [2.4 版本](#)
- [2.6 版本](#)
- [2.7 版本](#)

## 基本信息

Redis 是开源的内存 Key-Value 数据库实现。该仓库提供了 Redis 2.6 ~ 2.8.9 各个版本的镜像。

## 使用方法

默认会在 6379 端口启动数据库。

```
$ sudo docker run --name some-redis -d redis
```

另外还可以启用 [持久存储](#)。

```
$ sudo docker run --name some-redis -d redis redis-server --appendonly yes
```

默认数据存储位置在 VOLUME/data 。可以使用 `--volumes-from some-volume-container` 或 `-v /docker/host/dir:/data` 将数据存放到本地。

使用其他应用连接到容器，可以用

```
$ sudo docker run --name some-app --link some-redis:redis -d application-that-uses-redis
```

或者通过 `redis-cli`

```
$ sudo docker run -it --link some-redis:redis --rm redis sh -c 'exec redis-cli -h "$REDIS_PORT_6379_TCP_ADDR" -p "$REDIS'
```

## Dockerfile

- [2.6 版本](#)
- [最新 2.8 版本](#)

## 基本信息

[Nginx](#) 是开源的高效的 Web 服务器实现，支持 HTTP、HTTPS、SMTP、POP3、IMAP 等协议。该仓库提供了 Nginx 1.0 ~ 1.7 各个版本的镜像。

## 使用方法

下面的命令将作为一个静态页面服务器启动。

```
$ sudo docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx
```

用户也可以不使用这种映射方式，通过利用 Dockerfile 来直接将静态页面内容放到镜像中，内容为

```
FROM nginx
COPY static-html-directory /usr/share/nginx/html
```

之后生成新的镜像，并启动一个容器。

```
$ sudo docker build -t some-content-nginx .
$ sudo docker run --name some-nginx -d some-content-nginx
```

开放端口，并映射到本地的 8080 端口。

```
sudo docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

Nginx 的默认配置文件路径为 `/etc/nginx/nginx.conf`，可以通过映射它来使用本地的配置文件，例如

```
docker run --name some-nginx -v /some/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx
```

使用配置文件时，为了在容器中正常运行，需要保持 `daemon off;`。

## Dockerfile

- [1 ~ 1.7 版本](#)

## 基本信息

[WordPress](#) 是开源的 Blog 和内容管理系统框架，它基于 PHP 和 MySQL。该仓库提供了 WordPress 4.0 版本的镜像。

## 使用方法

启动容器需要 MySQL 的支持，默认端口为 80。

```
$ sudo docker run --name some-wordpress --link some-mysql:mysql -d wordpress
```

启动 WordPress 容器时可以指定的一些环境参数包括

- `-eWORDPRESS_DB_USER=...` 缺省为 “root”
- `-eWORDPRESS_DB_PASSWORD=...` 缺省为连接 mysql 容器的环境变量 `MYSQL_ROOT_PASSWORD` 的值
- `-eWORDPRESS_DB_NAME=...` 缺省为 “wordpress”
- `-eWORDPRESS_AUTH_KEY=..., -eWORDPRESS_SECURE_AUTH_KEY=..., -eWORDPRESS_LOGGED_IN_KEY=..., -eWORDPRESS_NONCE_KEY=..., -eWORDPRESS_AUTH_SALT=..., -eWORDPRESS_SECURE_AUTH_SALT=..., -eWORDPRESS_LOGGED_IN_SALT=..., -eWORDPRESS_NONCE_SALT=...` 缺省为随机 sha1 串

## Dockerfile

- [4.0 版本](#)



## 基本信息

[Node.js](#) 是基于 JavaScript 的可扩展服务端和网络软件开发平台。该仓库提供了 Node.js 0.8 ~ 0.11 各个版本的镜像。

## 使用方法

在项目中创建一个 Dockerfile。

```
FROM node:0.10-onbuild
# replace this with your application's default port
EXPOSE 8888
```

然后创建镜像，并启动容器

```
$ sudo docker build -t my-nodejs-app
$ sudo docker run -it --rm --name my-running-app my-nodejs-app
```

也可以直接运行一个简单容器。

```
$ sudo docker run -it --rm --name my-running-script -v "$(pwd)":/usr/src/myapp -w /usr/src/myapp node:0.10 node your
```

## Dockerfile

- [0.8 版本](#)
- [0.10 版本](#)
- [0.11 版本](#)

# 资源链接

---

- Docker 主站点: <https://www.docker.io>
- Docker 注册中心API: [http://docs.docker.com/reference/api/registry\\_api/](http://docs.docker.com/reference/api/registry_api/)
- DockerHub API: [http://docs.docker.com/reference/api/docker-io\\_api/](http://docs.docker.com/reference/api/docker-io_api/)
- Docker 远端应用API: [http://docs.docker.com/reference/api/docker\\_remote\\_api/](http://docs.docker.com/reference/api/docker_remote_api/)
- Dockerfile 参考 : <https://docs.docker.com/reference/builder/>
- Dockerfile 最佳实践 : [https://docs.docker.com/articles/dockerfile\\_best-practices/](https://docs.docker.com/articles/dockerfile_best-practices/)

# 存出和载入镜像

## 存出镜像

如果要导出镜像到本地文件，可以使用 `dockersave` 命令。

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu               14.04              c4ff7513909d       5 weeks ago        225.4 MB
...
$ sudo docker save -o ubuntu_14.04.tar ubuntu:14.04
```

## 载入镜像

可以使用 `docker load` 从导出的本地文件中再导入到本地镜像库，例如

```
$ sudo docker load --input ubuntu_14.04.tar
```

或

```
$ sudo docker load < ubuntu_14.04.tar
```

这将导入镜像以及其相关的元数据信息（包括标签等）。

## 移除本地镜像

---

如果要移除本地的镜像，可以使用 `docker rmi` 命令。注意 `docker rm` 命令是移除容器。

```
$ sudo docker rmi training/sinatra
Untagged: training/sinatra:latest
Deleted: 5bc342fa0b91cabf65246837015197eecfa24b2213ed6a51a8974ae250fedd8d
Deleted: ed0ffdcdae5eb2c3a55549857a8be7fc8bc4241fb19ad714364cbfd7a56b22f
Deleted: 5c58979d73ae448df5af1d8142436d81116187a7633082650549c52c3a2418f0
```

\*注意：在删除镜像之前要先用 `docker rm` 删掉依赖于这个镜像的所有容器。

## 镜像的实现原理

---

Docker 镜像是怎么实现增量的修改和维护的？每个镜像都由很多层次构成，Docker 使用 [Union FS](#) 将这些不同的层结合到一个镜像中去。

通常 Union FS 有两个用途，一方面可以实现不借助 LVM、RAID 将多个 disk 挂到同一个目录下，另一个更常用的就是将一个只读的分支和一个可写的分支联合在一起，LiveCD 正是基于此方法可以允许在镜像不变的基础上允许用户在其上进行一些写操作。Docker 在 AUFS 上构建的容器也是利用了类似的原理。

# Docker 容器

---

容器是 Docker 又一核心概念。

简单的说，容器是独立运行的一个或一组应用，以及它们的运行态环境。对应的，虚拟机可以理解为模拟运行的一整套操作系统（提供了运行态环境和其他系统环境）和跑在上面的应用。

本章将具体介绍如何来管理一个容器，包括创建、启动和停止等。

# 启动容器

启动容器有两种方式，一种是基于镜像新建一个容器并启动，另外一个是在终止状态（stopped）的容器重新启动。

因为 Docker 的容器实在太轻量级了，很多时候用户都是随时删除和新创建容器。

## 新建并启动

所需要的命令主要为 `docker run`。

例如，下面的命令输出一个 “Hello World”，之后终止容器。

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

这跟在本地直接执行 `/bin/echo 'helloworld'` 几乎感觉不出任何区别。

下面的命令则启动一个 `bash` 终端，允许用户进行交互。

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

其中，`-t` 选项让 Docker 分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上，`-i` 则让容器的标准输入保持打开。

在交互模式下，用户可以通过所创建的终端来输入命令，例如

```
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
```

当利用 `docker run` 来创建容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载
- 利用镜像创建并启动一个容器
- 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
- 从地址池配置一个 ip 地址给容器
- 执行用户指定的应用程序
- 执行完毕后容器被终止

## 启动已终止容器

可以利用 `docker start` 命令，直接将一个已经终止的容器启动运行。

容器的核心为所执行的应用程序，所需要的资源都是应用程序运行所必需的。除此之外，并没有其它的资源。可以在伪终端中利用 `ps` 或 `top` 来查看进程信息。

```
root@ba267838cc1b:/# ps
PID TTY          TIME CMD
  1 ?        00:00:00 bash
 11 ?        00:00:00 ps
```

可见，容器中仅运行了指定的 `bash` 应用。这种特点使得 `Docker` 对资源的利用率极高，是货真价实的轻量级虚拟化。



## 守护态运行

更多的时候，需要让 Docker 容器在后台以守护态（Daemonized）形式运行。此时，可以通过添加 `-d` 参数来实现。

例如下面的命令会在后台运行容器。

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

容器启动后会返回一个唯一的 id，也可以通过 `docker ps` 命令来查看容器信息。

```
$ 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
```

要获取容器的输出信息，可以通过 `docker logs` 命令。

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

# 终止容器

可以使用 `docker stop` 来终止一个运行中的容器。

此外，当Docker容器中指定的应用终结时，容器也自动终止。 例如对于上一章节中只启动了一个终端的容器，用户通过 `exit` 命令或 `Ctrl+d` 来退出终端时，所创建的容器立刻终止。

终止状态的容器可以用 `docker ps -a` 命令看到。例如

```
sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
ba267838cc1b	ubuntu:14.04	"/bin/bash"	30 minutes ago	Exited (0)	About a minute ago
98e5efa7d997	training/webapp:latest	"python app.py"	About an hour ago	Exited (0)	34 minutes ago

处于终止状态的容器，可以通过 `docker start` 命令来重新启动。

此外， `docker restart` 命令会将一个运行态的容器终止，然后再重新启动它。

# 进入容器

在使用 `-d` 参数时，容器启动后会进入后台。某些时候需要进入容器进行操作，有很多种方法，包括使用 `docker attach` 命令或 `nsenter` 工具等。

## attach 命令

`docker attach` 是 Docker 自带的命令。下面示例如何使用该命令。

```
$ sudo docker run -idt ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ sudo docker ps
CONTAINERID      IMAGE           COMMAND         CREATED        STATUS        PORTS          NAMES
243c32535da7     ubuntu:latest   "/bin/bash"     18 seconds ago Up 17 seconds  nostalgic_hy
$ sudo docker attach nostalgic_hypatia
root@243c32535da7:/#
```

但是使用 `attach` 命令有时候并不方便。当多个窗口同时 `attach` 到同一个容器的时候，所有窗口都会同步显示。当某个窗口因命令阻塞时，其他窗口也无法执行操作了。

## nsenter 命令

### 安装

`nsenter` 工具在 `util-linux` 包 2.23 版本后包含。如果系统中 `util-linux` 包没有该命令，可以按照下面的方法从源码安装。

```
$ cd /tmp; curl https://www.kernel.org/pub/linux/utils/util-linux/v2.24/util-linux-2.24.tar.gz | tar -zxf-; cd util-linux-2.24;
$ ./configure --without-ncurses
$ make nsenter && sudo cp nsenter /usr/local/bin
```

### 使用

`nsenter` 可以访问另一个进程的名字空间。`nsenter` 要正常工作需要有 `root` 权限。很不幸，Ubuntu 14.4 仍然使用的是 `util-linux 2.20`。安装最新版本的 `util-linux (2.24)` 版，请按照以下步骤：

```
$ curl https://www.kernel.org/pub/linux/utils/util-linux/v2.24/util-linux-2.24.tar.gz | tar xzvf util-linux-2.24.tar.gz
$ cd util-linux-2.24
$ ./configure --without-ncurses && make nsenter
$ sudo cp nsenter /usr/local/bin
```

为了连接到容器，你还需要找到容器的第一个进程的 `PID`，可以通过下面的命令获取。

```
PID=$(docker inspect --format "{{.State.Pid}}" <container>)
```

通过这个 `PID`，就可以连接到这个容器：

```
$ nsenter --target $PID --mount --uts --ipc --net --pid
```

下面给出一个完整的例子。

```
$ sudo docker run -idt ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ sudo docker ps
CONTAINERID      IMAGE           COMMAND         CREATED        STATUS        PORTS          NAMES
243c32535da7    ubuntu:latest   "/bin/bash"     18 seconds ago Up 17 seconds          nostalgic_hy
$ PID=$(docker-pid 243c32535da7)
10981
$ sudo nsenter --target 10981 --mount --uts --ipc --net --pid
root@243c32535da7:/#
```

更简单的，建议大家下载 [.bashrc\\_docker](#)，并将内容放到 `.bashrc` 中。

```
$ wget -P ~ https://github.com/yeasy/docker_practice/raw/master/_local/.bashrc_docker;
$ echo "[ -f ~/.bashrc_docker ] && . ~/.bashrc_docker" >> ~/.bashrc; source ~/.bashrc
```

这个文件中定义了很多方便使用 Docker 的命令，例如 `docker-pid` 可以获取某个容器的 PID；而 `docker-enter` 可以进入容器或直接在容器内执行命令。

```
$ echo $(docker-pid <container>)
$ docker-enter <container> ls
```

# 导出和导入容器

## 导出容器

如果要导出本地某个容器，可以使用 `docker export` 命令。

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7691a814370e	ubuntu:14.04	"/bin/bash"	36 hours ago	Exited (0) 21 hours ago		test

```
$ sudo docker export 7691a814370e > ubuntu.tar
```

这样将导出容器快照到本地文件。

## 导入容器快照

可以使用 `docker import` 从容器快照文件中再导入为镜像，例如

```
$ cat ubuntu.tar | sudo docker import - test/buntu:v1.0
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
test/ubuntu	v1.0	9d37a6082e97	About a minute ago	171.3 MB

此外，也可以通过指定 URL 或者某个目录来导入，例如

```
$ sudo docker import http://example.com/exampleimage.tgz example/imagerepo
```

\*注：用户既可以使用 `docker load` 来导入镜像存储文件到本地镜像库，也可以使用 `docker import` 来导入一个容器快照到本地镜像库。这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息（即仅保存容器当时的快照状态），而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

## 删除容器

---

可以使用 `docker rm` 来删除一个处于终止状态的容器。 例如

```
$sudo dockerrmtrusting_newton
trusting_newton
```

如果要删除一个运行中的容器，可以添加 `-f` 参数。Docker 会发送 `SIGKILL` 信号给容器。

# 仓库

---

仓库（Repository）是集中存放镜像的地方。

一个容易混淆的概念是注册服务器（Registry）。实际上注册服务器是管理仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像。从这方面来说，仓库可以被认为是一个具体的项目或目录。例如对于仓库地址 `dl.dockerpool.com/ubuntu` 来说，`dl.dockerpool.com` 是注册服务器地址，`ubuntu` 是仓库名。

大部分时候，并不需要严格区分这两者的概念。

# Docker Hub

目前 Docker 官方维护了一个公共仓库 [Docker Hub](#)，其中已经包括了超过 15,000 的镜像。大部分需求，都可以通过在 Docker Hub 中直接下载镜像来实现。

## 登录

可以通过执行 `docker login` 命令来输入用户名、密码和邮箱来完成注册和登录。注册成功后，本地用户目录的 `.dockercfg` 中将保存用户的认证信息。

## 基本操作

用户无需登录即可通过 `docker search` 命令来查找官方仓库中的镜像，并利用 `docker pull` 命令来将它下载到本地。

例如以 `centos` 为关键词进行搜索：

```
$ sudo docker search centos
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
centos	The official build of CentOS.	465	[OK]	
tianon/centos	CentOS 5 and 6, created using rinse instea...		28	
blalor/centos	Bare-bones base CentOS 6.5 image		6	[OK]
saltstack/centos-6-minimal		6	[OK]	
tutum/centos-6.4	DEPRECATED. Use tutum/centos:6.4 instead. ...		5	[OK]
...				

可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、星级（表示该镜像的受欢迎程度）、是否官方创建、是否自动创建。官方的镜像说明是官方项目组创建和维护的，`automated` 资源允许用户验证镜像的来源和内容。

根据是否是官方提供，可将镜像资源分为两类。一种是类似 `centos` 这样的基础镜像，被称为基础或根镜像。这些基础镜像是由 Docker 公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。还有一种类型，比如 `tianon/centos` 镜像，它是由 Docker 的用户创建并维护的，往往带有用户名称前缀。可以通过前缀 `user_name/` 来指定使用某个用户提供的镜像，比如 `tianon` 用户。

另外，在查找的时候通过 `-sN` 参数可以指定仅显示评价为 `N` 星以上的镜像。

下载官方 `centos` 镜像到本地。

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

用户也可以在登录后通过 `docker push` 命令来将镜像推送到 Docker Hub。

## 自动创建



自动创建 ( Automated Builds ) 功能对于需要经常升级镜像内程序来说，十分方便。 有时候，用户创建了镜像，安装了某个软件，如果软件发布新版本则需要手动更新镜像。。

而自动创建允许用户通过 DockerHub 指定跟踪一个目标网站 ( 目前支持 [GitHub](#) 或 [BitBucket](#) ) 上的项目，一旦项目发生新的提交，则自动执行创建。

要配置自动创建，包括如下的步骤：

- 创建并登陆 DockerHub，以及目标网站；
- 在目标网站中连接帐户到 DockerHub；
- 在 DockerHub 中 [配置一个自动创建](#)；
- 选取一个目标网站中的项目 ( 需要含 Dockerfile ) 和分支；
- 指定 Dockerfile 的位置，并提交创建。

之后，可以 在DockerHub 的 [自动创建页面](#) 中跟踪每次创建的状态。

# 私有仓库

有时候使用 DockerHub 这样的公共仓库可能不方便，用户可以创建一个本地仓库供私人使用。

本节介绍如何使用本地仓库。

`docker-registry` 是官方提供的工具，可以用于构建私有的镜像仓库。

## 安装运行 `docker-registry`

### 容器运行

在安装了 Docker 后，可以通过获取官方 `registry` 镜像来运行。

```
$ sudo docker run -d -p 5000:5000 registry
```

这将使用官方的 `registry` 镜像来启动本地的私有仓库。用户可以通过指定参数来配置私有仓库位置，例如配置镜像存储到 Amazon S3 服务。

```
$ sudo docker run \
  -e SETTINGS_FLAVOR=s3 \
  -e AWS_BUCKET=acme-docker \
  -e STORAGE_PATH=/registry \
  -e AWS_KEY=AKIAHSB43HS3J92MXZ \
  -e AWS_SECRET=xdDowwIK7TJajV1Y7EoOZrmuPEJIHYcNP2k4j49T \
  -e SEARCH_BACKEND=sqlalchemy \
  -p 5000:5000 \
  registry
```

此外，还可以指定本地路径（如 `/home/user/registry-conf`）下的配置文件。

```
$ sudo docker run -d -p 5000:5000 -v /home/user/registry-conf:/registry-conf -e DOCKER_REGISTRY_CONFIG=/registry-co
```

默认情况下，仓库会被创建在容器的 `/tmp/registry` 下。可以通过 `-v` 参数来将镜像文件存放在本地的指定路径。例如下面的例子将上传的镜像放到 `/opt/data/registry` 目录。

```
$ sudo docker run -d -p 5000:5000 -v /opt/data/registry:/tmp/registry registry
```

### 本地安装

对于 Ubuntu 或 CentOS 等发型包，可以直接通过源安装。

- Ubuntu

```
$ sudo apt-get install -y build-essential python-dev libevent-dev python-pip liblzma-dev
$ sudo pip install docker-registry
```

- CentOS

```
$ sudo yum install -y python-devel libevent-devel python-pip gcc xz-devel  
$ sudo python-pip install docker-registry
```

也可以从 [docker-registry](#) 项目下载源码进行安装。

```
$ sudo apt-get install build-essential python-dev libevent-dev python-pip libssl-dev liblzma-dev libffi-dev  
$ git clone https://github.com/docker/docker-registry.git  
$ cd registry  
$ pip install.
```

然后修改配置文件，主要修改dev模板段的 `storage_path` 到本地的存储仓库的路径。

```
$ cp config/config_sample.yml config/config.yml
```

之后启动 Web 服务。

```
$ sudo gunicorn -c contrib/gunicorn.py docker_registry.wsgi:application
```

或者

```
$ sudo gunicorn --access-logfile - --error-logfile - -k gevent -b 0.0.0.0:5000 -w 4 --max-requests 100 docker_registry.wsgi
```

此时使用访问本地的 5000 端口，看到输出 `docker-registry` 的版本信息说明运行成功。

\*注：`config/config_sample.yml` 文件是示例配置文件。

## 在私有仓库上传、下载、搜索镜像

创建好私有仓库之后，就可以使用 `docker tag` 来标记一个镜像，然后推送它到仓库，别的机器上就可以下载下来了。例如私有仓库地址为 `192.168.7.26:5000`。

先在本机查看已有的镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGEID	CREATED	VIRTUALSIZE
ubuntu	latest	ba5877dc9bec	6 weeks ago	192.7 MB
ubuntu	14.04	ba5877dc9bec	6 weeks ago	192.7 MB

使用 `docker tag` 将 `ba58` 这个镜像标记为 `192.168.7.26:5000/test`（格式为 `docker tag IMAGE[:TAG] [REGISTRYHOST]/[USERNAME/]NAME[:TAG]`）。

```
$ sudo docker tag ba58 192.168.7.26:5000/test
root ~# docker images
```

REPOSITORY	TAG	IMAGEID	CREATED	VIRTUAL SIZE
ubuntu	14.04	ba5877dc9bec	6 weeks ago	192.7 MB
ubuntu	latest	ba5877dc9bec	6 weeks ago	192.7 MB
192.168.7.26:5000/test	latest	ba5877dc9bec	6 weeks ago	192.7 MB

使用 `docker push` 上传标记的镜像。

```
$ sudo docker push 192.168.7.26:5000/test
The push refers to a repository [192.168.7.26:5000/test] (len: 1)
Sending image list
Pushing repository 192.168.7.26:5000/test (1 tags)
Image 511136ea3c5a already pushed, skipping
Image 9bad880da3d2 already pushed, skipping
Image 25f11f5fb0cb already pushed, skipping
Image ebc34468f71d already pushed, skipping
Image 2318d26665ef already pushed, skipping
Image ba5877dc9bec already pushed, skipping
Pushing tag for rev [ba5877dc9bec] on http://192.168.7.26:5000/v1/repositories/test/tags/latest
```

用 `curl` 查看仓库中的镜像。

```
$ curl http://192.168.7.26:5000/v1/search
{"num_results":7,"query":"","results":[{"description":"","name":"library/miaxis_j2ee"}, {"description":"","name"
```



这里可以看到 `{"description":"","name":"library/test"}`，表明镜像已经被成功上传了。

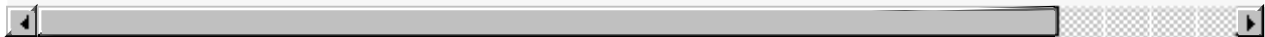
现在可以到另外一台机器去下载这个镜像。

```
$ sudo docker pull 192.168.7.26:5000/test
Pulling repository 192.168.7.26:5000/test
ba5877dc9bec: Download complete
511136ea3c5a: Download complete
9bad880da3d2: Download complete
25f11f5fb0cb: Download complete
ebc34468f71d: Download complete
2318d26665ef: Download complete
$ sudo docker images
```

REPOSITORY	TAG	IMAGEID	CREATED	VIRTUAL SIZE
192.168.7.26:5000/test	latest	ba5877dc9bec	6 weeks ago	192.7 MB

可以使用 [这个脚本](#) 批量上传本地的镜像到注册服务器中，默认是本地注册服务器 `127.0.0.1:5000`。例如：

```
$ wget https://github.com/yeasy/docker_practice/raw/master/_local/push_images.sh;sudo chmod a+xpsh_images.sh
$ ./push_images.sh ubuntu:latest centos:centos7
The registry server is 127.0.0.1
Uploading ubuntu:latest...
The push refers to a repository [127.0.0.1:5000/ubuntu] (len: 1)
Sending image list
Pushing repository 127.0.0.1:5000/ubuntu (1 tags)
Image 511136ea3c5a already pushed, skipping
Image bfb8b5a2ad34 already pushed, skipping
Image c1f3bdbd8355 already pushed, skipping
Image 897578f527ae already pushed, skipping
Image 9387bcc9826e already pushed, skipping
Image 809ed259f845 already pushed, skipping
Image 96864a7d2df3 already pushed, skipping
Pushing tag for rev [96864a7d2df3] on {http://127.0.0.1:5000/v1/repositories/ubuntu/tags/latest}
Untagged: 127.0.0.1:5000/ubuntu:latest
Done
Uploading centos:centos7...
The push refers to a repository [127.0.0.1:5000/centos] (len: 1)
Sending image list
Pushing repository 127.0.0.1:5000/centos (1 tags)
Image 511136ea3c5a already pushed, skipping
34e94e67e63a: Image successfully pushed
70214e5d0a90: Image successfully pushed
Pushing tag for rev [70214e5d0a90] on {http://127.0.0.1:5000/v1/repositories/centos/tags/centos7}
Untagged: 127.0.0.1:5000/centos:centos7
Done
```



# 仓库配置文件

Docker 的 Registry 利用配置文件提供了一些仓库的模板（flavor），用户可以直接使用它们来进行开发或生产部署。

## 模板

在 `config_sample.yml` 文件中，可以看到一些现成的模板段：

- `common` ：基础配置
- `local` ：存储数据到本地文件系统
- `s3` ：存储数据到 AWS S3 中
- `dev` ：使用 `local` 模板的基本配置
- `test` ：单元测试使用
- `prod` ：生产环境配置（基本上跟s3配置类似）
- `gcs` ：存储数据到 Google 的云存储
- `swift` ：存储数据到 OpenStack Swift 服务
- `glance` ：存储数据到 OpenStack Glance 服务，本地文件系统为后备
- `glance-swift` ：存储数据到 OpenStack Glance 服务，Swift 为后备
- `elliptics` ：存储数据到 Elliptics key/value 存储

用户也可以添加自定义的模版段。

默认情况下使用的模板是 `dev`，要使用某个模板作为默认值，可以添加 `SETTINGS_FLAVOR` 到环境变量中，例如

```
export SETTINGS_FLAVOR=dev
```

另外，配置文件中支持从环境变量中加载值，语法格式为 `_env:VARIABLENAME[:DEFAULT]`。

## 示例配置

```
common:
  loglevel: info
  search_backend: "_env:SEARCH_BACKEND:"
  sqlalchemy_index_database:
    "_env:SQLALCHEMY_INDEX_DATABASE:sqlite:///tmp/docker-registry.db"

prod:
  loglevel: warn
  storage: s3
  s3_access_key: _env:AWS_S3_ACCESS_KEY
  s3_secret_key: _env:AWS_S3_SECRET_KEY
  s3_bucket: _env:AWS_S3_BUCKET
  boto_bucket: _env:AWS_S3_BUCKET
  storage_path: /srv/docker
  smtp_host: localhost
  from_addr: docker@myself.com
  to_addr: my@myself.com

dev:
  loglevel: debug
  storage: local
  storage_path: /home/myself/docker

test:
  storage: local
  storage_path: /tmp/tmpdockertmp
```

## 选项

# Docker 数据管理

---

这一章介绍如何在 Docker 内部以及容器之间管理数据，在容器中管理数据主要有两种方式：

- 数据卷（Data volumes）

- 数据卷容器（Data volume containers）



# 数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

- 数据卷可以在容器之间共享和重用
- 对数据卷的修改会立马生效
- 对数据卷的更新，不会影响镜像
- 卷会一直存在，直到没有容器使用

\*数据卷的使用，类似于 Linux 下对目录或文件进行 mount。

## 创建一个数据卷

在用 `docker run` 命令的时候，使用 `-v` 标记来创建一个数据卷并挂载到容器里。在一次 `run` 中多次使用可以挂载多个数据卷。

下面创建一个 `web` 容器，并加载一个数据卷到容器的 `/webapp` 目录。

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

\*注意：也可以在 `Dockerfile` 中使用 `VOLUME` 来添加一个或者多个新的卷到由该镜像创建的任意容器。

## 挂载一个主机目录作为数据卷

使用 `-v` 标记也可以指定挂载一个本地主机的目录到容器中去。

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

上面的命令加载主机的 `/src/webapp` 目录到容器的 `/opt/webapp` 目录。这个功能在进行测试的时候十分方便，比如用户可以放置一些程序到本地目录中，来查看容器是否正常工作。本地目录的路径必须是绝对路径，如果目录不存在 Docker 会自动为你创建它。

\*注意：`Dockerfile` 中不支持这种用法，这是因为 `Dockerfile` 是为了移植和分享用的。然而，不同操作系统的路径格式不一样，所以目前还不能支持。

Docker 挂载数据卷的默认权限是读写，用户也可以通过 `:ro` 指定为只读。

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

加了 `:ro` 之后，就挂载为只读了。

## 挂载一个本地主机文件作为数据卷

`-v` 标记也可以从主机挂载单个文件到容器中

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

这样就可以记录在容器输入过的命令了。

\*注意：如果直接挂载一个文件，很多文件编辑工具，包括 `vi` 或者 `sed --in-place`，可能会造成文件 `inode` 的改变，从 `Docker 1.1.0`起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。


## 数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。

数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。

首先，创建一个命名的数据卷容器 `dbdata`：

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



然后，在其他容器中使用 `--volumes-from` 来挂载 `dbdata` 容器中的数据卷。

```
$ sudo docker run -d --volumes-from dbdata --name db1 training/postgres  
$ sudo docker run -d --volumes-from dbdata --name db2 training/postgres
```

还可以使用多个 `--volumes-from` 参数来从多个容器挂载多个数据卷。也可以从其他已经挂载了容器卷的容器来挂载数据卷。

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

\*注意：使用 `--volumes-from` 参数所挂载数据卷的容器自己并不需要保持在运行状态。

如果删除了挂载的容器（包括 `dbdata`、`db1` 和 `db2`），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 `docker rm -v` 命令来指定同时删除关联的容器。这可以让用户在容器之间升级和移动数据卷。具体的操作将在下一节中进行讲解。

# 利用数据卷容器来备份、恢复、迁移数据卷

可以利用数据卷对其中的数据进行进行备份、恢复和迁移。

## 备份

首先使用 `--volumes-from` 标记来创建一个加载 `dbdata` 容器卷的容器，并从本地主机挂载当前到容器的 `/backup` 目录。命令如下：

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

容器启动后，使用了 `tar` 命令来将 `dbdata` 卷备份为本地的 `/backup/backup.tar`。

## 恢复

如果要恢复数据到一个容器，首先创建一个带有数据卷的容器 `dbdata2`。

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

然后创建另一个容器，挂载 `dbdata2` 的容器，并使用 `untar` 解压备份文件到挂载的容器卷中。

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

# Docker 中的网络功能介绍

---

Docker 允许通过外部访问容器或容器互联的方式来提供网络服务。

## 外部访问容器

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 `-P` 或 `-p` 参数来指定端口映射。

当使用 `-P` 标记时，Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口。

使用 `docker ps` 可以看到，本地主机的 49155 被映射到了容器的 5000 端口。此时访问本机的 49155 端口即可访问容器内 web 应用提供的界面。

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

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

同样的，可以通过 `docker logs` 命令来查看应用的信息。

```
$ 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 -
```

`-p`（小写的）则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。支持的格式有 `ip:hostPort:containerPort` | `ip::containerPort` | `hostPort:containerPort`。

## 映射所有接口地址

使用 `hostPort:containerPort` 格式本地的 5000 端口映射到容器的 5000 端口，可以执行

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

此时默认会绑定本地所有接口上的所有地址。

## 映射到指定地址的指定端口

可以使用 `ip:hostPort:containerPort` 格式指定映射使用一个特定地址，比如 `localhost` 地址 127.0.0.1

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

## 映射到指定地址的任意端口

使用 `ip::containerPort` 绑定 `localhost` 的任意端口到容器的 5000 端口，本地主机会自动分配一个端口。

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

还可以使用 `udp` 标记来指定 `udp` 端口

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

## 查看映射端口配置

使用 `docker port` 来查看当前映射的端口配置，也可以查看到绑定的地址

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

注意：

- 容器有自己的内部网络和 ip 地址（使用 `docker inspect` 可以获取所有的变量，Docker 还可以有一个可变的网络配置。）
- `-p` 标记可以多次使用来绑定多个端口

例如

```
$ sudo docker run -d -p 5000:5000 -p 3000:80 training/webapp python app.py
```

## 容器互联

容器的连接（linking）系统是除了端口映射外，另一种跟容器中应用交互的方式。

该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

## 自定义容器命名

连接系统依据容器的名称来执行。因此，首先需要自定义一个好记的容器命名。

虽然当创建容器的时候，系统默认会分配一个名字。自定义命名容器有2个好处：

- 自定义的命名，比较好记，比如一个web应用容器我们可以给它起名叫web
- 当要连接其他容器时候，可以作为一个有用的参考点，比如连接web容器到db容器

使用 `--name` 标记可以为容器自定义命名。

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

使用 `docker ps` 来验证设定的命名。

```
$ 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

也可以使用 `docker inspect` 来查看容器的名字

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

注意：容器的名称是唯一的。如果已经命名了一个叫 `web` 的容器，当你要再次使用 `web` 这个名称的时候，需要先用 `docker rm` 来删除之前创建的同名容器。

在执行 `docker run` 的时候如果添加 `--rm` 标记，则容器在终止后会立刻删除。注意，`--rm` 和 `-d` 参数不能同时使用。

## 容器互联

使用 `--link` 参数可以让容器之间安全的进行交互。

下面先创建一个新的数据库容器。

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

删除之前创建的 `web` 容器



```
$ docker rm -f web
```

然后创建一个新的 web 容器，并将它连接到 db 容器

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

此时，db 容器和 web 容器建立互联关系。

--link 参数的格式为 --link name:alias，其中 name 是要链接的容器的名称，alias 是这个连接的别名。

使用 docker ps 来查看容器的连接

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
349169744e49	training/postgres:latest	su postgres -c 'usr	About a minute ago	Up About a minute	5432/tcp	
aed84ee21bde	training/webapp:latest	python app.py	16 hours ago	Up 2 minutes	0.0.0.0:49154->5000	

可以看到自定义命名的容器，db 和 web，db 容器的 names 列有 db 也有 web/db。这表示 web 容器链接到 db 容器，web 容器将被允许访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主主机上。在启动 db 容器的时候并没有使用 -p 和 -P 标记，从而避免了暴露数据库端口到外部网络上。

Docker 通过 2 种方式为容器公开连接信息：

- 环境变量
- 更新 /etc/hosts 文件

使用 env 命令来查看 web 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
...
```

其中 DB\_ 开头的环境变量是供 web 容器连接 db 容器使用，前缀采用大写的连接别名。

除了环境变量，Docker 还添加 host 信息到父容器的 /etc/hosts 的文件。下面是父容器 web 的 hosts 文件

```
$ sudo docker run -t -i --rm --link db:db training/webapp/bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7 aed84ee21bde
...
172.17.0.5 db
```

这里有 2 个 hosts，第一个是 web 容器，web 容器用 id 作为他的主机名，第二个是 db 容器的 ip 和主机名。可以在 web 容器中安装 ping 命令来测试跟db容器的连通。

```
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
```

用 ping 来测试db容器，它会解析成 172.17.0.5 。\*注意：官方的 ubuntu 镜像默认没有安装 ping，需要自行安装。

用户可以链接多个子容器到父容器，比如可以链接多个 web 到 db 容器上。

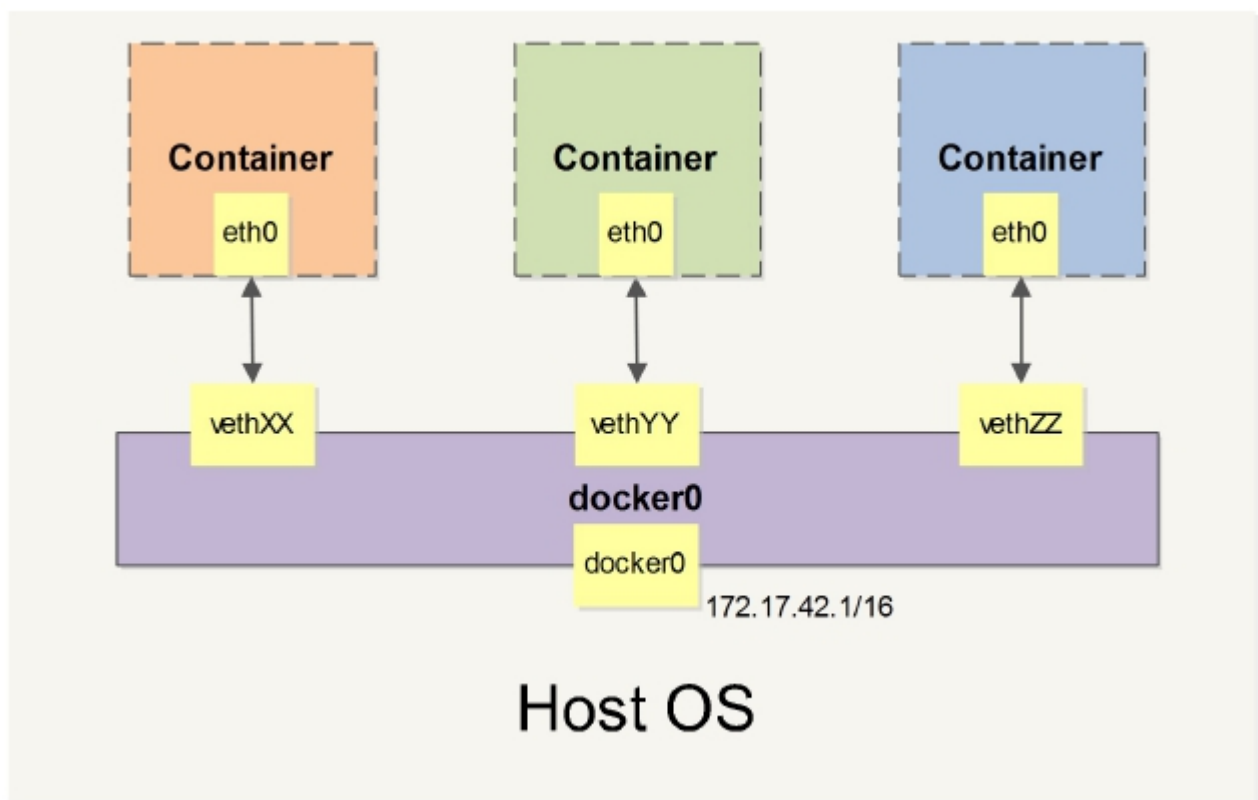
# 高级网络配置

本章将介绍 Docker 的一些高级网络配置和选项。

当 Docker 启动时，会自动在主机上创建一个 `docker0` 虚拟网桥，实际上是 Linux 的一个 bridge，可以理解为一个软件交换机。它会在挂载到它的网口之间进行转发。

同时，Docker 随机分配一个本地未占用的私有网段（在 [RFC1918](#) 中定义）中的一个地址给 `docker0` 接口。比如典型的 `172.17.42.1`，掩码为 `255.255.0.0`。此后启动的容器内的网口也会自动分配一个同一网段（`172.17.0.0/16`）的地址。

当创建一个 Docker 容器的时候，同时会创建了一对 `vethpair` 接口（当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包）。这对接口一端在容器内，即 `eth0`；另一端在本地并被挂载到 `docker0` 网桥，名称以 `veth` 开头（例如 `vethAQI2QT`）。通过这种方式，主机可以跟容器通信，容器之间也可以相互通信。Docker 就创建了在主机和所有容器之间一个虚拟共享网络。



接下来的部分将介绍在一些场景中，Docker 所有的网络定制配置。以及通过 Linux 命令来调整、补充、甚至替换 Docker 默认的网络配置。

# 快速配置指南

---

下面是一个跟 Docker 网络相关的命令列表。

其中有些命令选项只有在 Docker 服务启动的时候才能配置，而且不能马上生效。

- `-b BRIDGE` or `--bridge=BRIDGE` --指定容器挂载的网桥
- `--bip=CIDR` --定制 `docker0` 的掩码
- `-H SOCKET...` or `--host=SOCKET...` --Docker 服务端接收命令的通道
- `--icc=true|false` --是否支持容器之间进行通信
- `--ip-forward=true|false` --请看下文容器之间的通信
- `--iptables=true|false` --禁止 Docker 添加 iptables 规则
- `--mtu=BYTES` --容器网络中的 MTU

下面2个命令选项既可以在启动服务时指定，也可以 Docker 容器启动（`docker run`）时候指定。在 Docker 服务启动的时候指定则会成为默认值，后面执行 `docker run` 时可以覆盖设置的默认值。

- `--dns=IP_ADDRESS...` --使用指定的DNS服务器
- `--dns-search=DOMAIN...` --指定DNS搜索域

最后这些选项只有在 `docker run` 执行时使用，因为它是针对容器的特性内容。

- `-h HOSTNAME` or `--hostname=HOSTNAME` --配置容器主机名
- `--link=CONTAINER_NAME:ALIAS` --添加到另一个容器的连接
- `--net=bridge|none|container:NAME_or_ID|host` --配置容器的桥接模式
- `-p SPEC` or `--publish=SPEC` --映射容器端口到宿主主机
- `-P` or `--publish-all=true|false` --映射容器所有端口到宿主主机

## 配置 DNS

Docker 没有每个容器专门定制镜像，那么怎么自定义配置容器的主机名和 DNS 配置呢？秘诀就是它利用虚拟文件来挂载到来容器的 3 个相关配置文件。

在容器中使用 `mount` 命令可以看到挂载信息：

```
$ mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
tmpfs on /etc/resolv.conf type tmpfs ...
...
```

这种机制可以让宿主主机 DNS 信息发生更新后，所有 Docker 容器的 dns 配置通过 `/etc/resolv.conf` 文件立刻得到更新。

如果用户想要手动指定容器的配置，可以利用下面的选项。

`-h HOSTNAME` or `--hostname=HOSTNAME` 设定容器的主机名，它会被写到容器内的 `/etc/hostname` 和 `/etc/hosts`。但它在容器外部看不到，既不会在 `docker ps` 中显示，也不会其他的容器的 `/etc/hosts` 看到。

`--link=CONTAINER_NAME:ALIAS` 选项会在创建容器的时候，添加一个其他容器的主机名到 `/etc/hosts` 文件中，让新容器的进程可以使用主机名 `ALIAS` 就可以连接它。

`--dns=IP_ADDRESS` 添加 DNS 服务器到容器的 `/etc/resolv.conf` 中，让容器用这个服务器来解析所有不在 `/etc/hosts` 中的主机名。

`--dns-search=DOMAIN` 设定容器的搜索域，当设定搜索域为 `.example.com` 时，在搜索一个名为 `host` 的主机时，DNS 不仅搜索 `host`，还会搜索 `host.example.com`。注意：如果没有上述最后 2 个选项，Docker 会默认用主机上的 `/etc/resolv.conf` 来配置容器。

## 容器访问控制

容器的访问控制，主要通过 Linux 上的 iptables 防火墙来进行管理和实现。iptables 是 Linux 上默认的防火墙软件，在大部分发行版中都自带。

## 容器访问外部网络

容器要想访问外部网络，需要本地系统的转发支持。在Linux 系统中，检查转发是否打开。

```
$sysctl net.ipv4.ip_forward  
net.ipv4.ip_forward = 1
```

如果为 0，说明没有开启转发，则需要手动打开。

```
$sysctl -w net.ipv4.ip_forward=1
```

如果在启动 Docker 服务的时候设定 `--ip-forward=true`，Docker 就会自动设定系统的 `ip_forward` 参数为 1。

## 容器之间访问

容器之间相互访问，需要两方面的支持。

- 容器的网络拓扑是否已经互联。默认情况下，所有容器都会被连接到 `docker0` 网桥上。
- 本地系统的防火墙软件 `--iptables` 是否允许通过。

## 访问所有端口

当启动 Docker 服务时候，默认会添加一条转发策略到 iptables 的 FORWARD 链上。策略为通过（ACCEPT）还是禁止（DROP）取决于配置 `--icc=true`（缺省值）还是 `--icc=false`。当然，如果手动指定 `--iptables=false` 则不会添加 iptables 规则。

可见，默认情况下，不同容器之间是允许网络互通的。如果为了安全考虑，可以在 `/etc/default/docker` 文件中配置 `DOCKER_OPTS=--icc=false` 来禁止它。

## 访问指定端口

在通过 `--icc=false` 关闭网络访问后，还可以通过 `--link=CONTAINER_NAME:ALIAS` 选项来访问容器的开放端口。

例如，在启动 Docker 服务时，可以同时使用 `icc=false--iptables=true` 参数来关闭允许相互的网络访问，并让 Docker 可以修改系统中的 iptables 规则。

此时，系统中的 iptables 规则可能是类似

```
$ sudo iptables -nL
...
Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
DROP      all  --  0.0.0.0/0            0.0.0.0/0
...
```

之后，启动容器（`docker run`）时使用 `--link=CONTAINER_NAME:ALIAS` 选项。Docker 会在 iptable 中为两个容器分别添加一条 `ACCEPT` 规则，允许相互访问开放的端口（取决于 Dockerfile 中的 `EXPOSE` 行）。

当添加了 `--link=CONTAINER_NAME:ALIAS` 选项后，添加了 iptables 规则。

```
$ sudo iptables -nL
...
Chain FORWARD (policy ACCEPT)
target     prot opt source                destination      tcp spt:80
ACCEPT    tcp  --  172.17.0.2            172.17.0.3
ACCEPT    tcp  --  172.17.0.3            172.17.0.2      tcp dpt:80
DROP      all  --  0.0.0.0/0            0.0.0.0/0
```

注意：`--link=CONTAINER_NAME:ALIAS` 中的 `CONTAINER_NAME` 目前必须是 Docker 分配的名字，或使用 `--name` 参数指定的名字。主机名则不会被识别。

# 映射容器端口到宿主主机的实现

默认情况下，容器可以主动访问到外部网络的连接，但是外部网络无法访问到容器。

## 容器访问外部实现

容器所有到外部网络的连接，源地址都会被NAT成本地系统的IP地址。这是使用 iptables 的源地址伪装操作实现的。

查看主机的 NAT 规则。

```
$ sudo iptables -t nat -nL
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source                destination
MASQUERADE all -- 172.17.0.0/16          !172.17.0.0/16
...
```

其中，上述规则将所有源地址在 172.17.0.0/16 网段，目标地址为其他网段（外部网络）的流量动态伪装为从系统网卡发出。MASQUERADE 跟传统 SNAT 的好处是它能动态从网卡获取地址。

## 外部访问容器实现

容器允许外部访问，可以在 docker run 时候通过 -p 或 -P 参数来启用。

不管用那种办法，其实也是在本地的 iptable 的 nat 表中添加相应的规则。

使用 -P 时：

```
$ iptables -t nat -nL
...
Chain DOCKER (2 references)
target     prot opt source                destination
DNAT       tcp -- 0.0.0.0/0             0.0.0.0/0             tcp dpt:49153 to:172.17.0.2:80
```

使用 -p 80:80 时：

```
$ iptables -t nat -nL
Chain DOCKER (2 references)
target     prot opt source                destination
DNAT       tcp -- 0.0.0.0/0             0.0.0.0/0             tcp dpt:80 to:172.17.0.2:80
```

注意：

- 这里的规则映射了 0.0.0.0，意味着将接受主机来自所有接口的流量。用户可以通过 -p IP:host\_port:container\_port 或 -p IP::port 来指定允许访问容器的主机上的 IP、接口等，以制定更严格的规则。
- 如果希望永久绑定到某个固定的 IP 地址，可以在 Docker 配置文件 /etc/default/docker 中指定 DOCKER\_OPTS="--ip=IP\_ADDRESS"，之后重启 Docker 服务即可生效。



## 配置 docker0 网桥

Docker 服务默认会创建一个 docker0 网桥（其上有一个 docker0 内部接口），它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和本地主机都放到同一个物理网络。

Docker 默认指定了 docker0 接口的 IP 地址和子网掩码，让主机和容器之间可以通过网桥相互通信，它还给出了 MTU（接口允许接收的最大传输单元），通常是 1500 Bytes，或宿主主机网络路由上支持的默认值。这些值都可以在服务启动的时候进行配置。

- `--bip=CIDR` -- IP 地址加掩码格式，例如 192.168.1.5/24
- `--mtu=BYTES` -- 覆盖默认的 Docker mtu 配置

也可以在配置文件中配置 DOCKER\_OPTS，然后重启服务。由于目前 Docker 网桥是 Linux 网桥，用户可以使用 `brctlshow` 来查看网桥和端口连接信息。

```
$ sudo brctlshow
bridge name      bridge id        STP enabled      interfaces
docker0          8000.3a1d7362b4ee  no               veth65f9
                  vethdda6
```

\*注：`brctl` 命令在 Debian、Ubuntu 中可以使用 `sudo apt-get install bridge-utils` 来安装。

每次创建一个新容器的时候，Docker 从可用的地址段中选择一个空闲的 IP 地址分配给容器的 eth0 端口。使用本地主机上 docker0 接口的 IP 作为所有容器的默认网关。

```
$ sudo docker run -i -t --rm base/bin/bash
$ ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
        valid_lft forever preferred_lft forever
$ ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0 proto kernel scope link src 172.17.0.3
$ exit
```

# 自定义网桥

除了默认的 `docker0` 网桥，用户也可以指定网桥来连接各个容器。

在启动 Docker 服务的时候，使用 `-b BRIDGE` 或 `--bridge=BRIDGE` 来指定使用的网桥。

如果服务已经运行，那需要先停止服务，并删除旧的网桥。

```
$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
```

然后创建一个网桥 `bridge0` 。

```
$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.5.1/24 dev bridge0
$ sudo ip link set dev bridge0 up
```

查看确认网桥创建并启动。

```
$ ip addr show bridge0
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global bridge0
        valid_lft forever preferred_lft forever
```

配置 Docker 服务，默认桥接到创建的网桥上。

```
$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker start
```

启动 Docker 服务。新建一个容器，可以看到它已经桥接到了 `bridge0` 上。

可以继续用 `brctl show` 命令查看桥接的信息。另外，在容器中使用 `ipaddr` 和 `iproute` 命令来查看 IP 地址配置和路由信息。

## 工具和示例

---

在介绍自定义网络拓扑之前，你可能会对一些外部工具和例子感兴趣：

### **pipework**

Jérôme Petazzoni 编写了一个叫 [pipework](#) 的 shell 脚本，可以帮助用户在比较复杂的场景中完成容器的连接。

### **playground**

Brandon Rhodes 创建了一个提供完整的 Docker 容器网络拓扑管理的 [Python库](#)，包括路由、NAT 防火墙；以及一些提供 HTTP, SMTP, POP, IMAP, Telnet, SSH, FTP 的服务器。

## 编辑网络配置文件

---

Docker 1.2.0 开始支持在运行中的容器里编辑 `/etc/hosts`, `/etc/hostname` 和 `/etc/resolv.conf` 文件。

但是这些修改是临时的，只在运行的容器中保留，容器终止或重启后并不会被保存下来。也不会被 `docker commit` 提交。

## 示例：创建一个点到点连接

默认情况下，Docker 会将所有容器连接到由 docker0 提供的虚拟子网中。

用户有时候需要两个容器之间可以直连通信，而不用通过主机网桥进行桥接。

解决办法很简单：创建一对 peer 接口，分别放到两个容器中，配置成点到点链路类型即可。

首先启动 2 个容器：

```
$ sudo docker run -i -t --rm --net=none base/bin/bash
root@1f1f4c1f931a:/#
$ sudo docker run -i -t --rm --net=none base/bin/bash
root@12e343489d2f:/#
```

找到进程号，然后创建网络名字空间的跟踪文件。

```
$ sudo docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a
2989
$ sudo docker inspect -f '{{.State.Pid}}' 12e343489d2f
3004
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/2989/ns/net /var/run/netns/2989
$ sudo ln -s /proc/3004/ns/net /var/run/netns/3004
```

创建一对 peer 接口，然后配置路由

```
$ sudo ip link add A type veth peer name B

link set A netns 2989
netns exec 2989 ip addr add 10.1.1.1/32 dev A
netns exec 2989 ip link set A up
netns exec 2989 ip route add 10.1.1.2/32 dev A

link set B netns 3004
netns exec 3004 ip addr add 10.1.1.2/32 dev B
netns exec 3004 ip link set B up
netns exec 3004 ip route add 10.1.1.1/32 dev B
```

现在这 2 个容器就可以相互 ping 通，并成功建立连接。点到点链路不需要子网和子网掩码。

此外，也可以不指定 `--net=none` 来创建点到点链路。这样容器还可以通过原先的网络来通信。

利用类似的办法，可以创建一个只跟主机通信的容器。但是一般情况下，更推荐使用 `--icc=false` 来关闭容器之间的通信。

## 实战案例

---

介绍一些典型的应用场景和案例。

# 使用 Supervisor 来管理进程

Docker 容器在启动的时候开启单个进程，比如，一个 ssh 或者 apache 的 daemon 服务。但我们经常需要在一个机器上开启多个服务，这可以有很多方法，最简单的就是把多个启动命令方到一个启动脚本里面，启动的时候直接启动这个脚本，另外就是安装进程管理工具。

本小节将使用进程管理工具 supervisor 来管理容器中的多个进程。使用 Supervisor 可以更好的控制、管理、重启我们希望运行的进程。在这里我们演示一下如何同时使用 ssh 和 apache 服务。

## 配置

首先创建一个 Dockerfile，内容和各部分的解释如下。

```
FROM ubuntu:13.04
MAINTAINER examples@docker.com
RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/sources.list
RUN apt-get update
RUN apt-get upgrade -y
```

安装 supervisor 安装 ssh、apache 和 supervisor。

```
RUN apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/run/sshd
RUN mkdir -p /var/log/supervisor
```

这里安装 3 个软件，还创建了 2 个 ssh 和 supervisor 服务正常运行所需要的目录。

```
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

添加 supervisord 的配置文件，并复制配置文件到对应目录下。

```
EXPOSE 22 80
CMD ["/usr/bin/supervisord"]
```

这里我们映射了 22 和 80 端口，使用 supervisord 的可执行路径启动服务。

## supervisor 配置文件内容

```
[supervisord]
nodaemon=true
[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -D FOREGROUND"
```

配置文件包含目录和进程，第一段 supervisord 配置软件本身，使用 nodaemon 参数来运行。第二段包含要控制的 2 个服务。每一段包含一个服务的目录和启动这个服务的命令。

## 使用方法

创建镜像。

```
$ sudo docker build -t test/supervisord .
```

启动 supervisor 容器。

```
$ sudo docker run -p 22 -p 80 -t -i test/supervisord
2013-11-25 18:53:22,312 CRIT Supervisor running as root (no user in config file)
2013-11-25 18:53:22,312 WARN Included extra file "/etc/supervisor/conf.d/supervisord.conf" during parsing
2013-11-25 18:53:22,342 INFO supervisord started with pid 1
2013-11-25 18:53:23,346 INFO spawned: 'sshd' with pid 6
2013-11-25 18:53:23,349 INFO spawned: 'apache2' with pid 7
```

使用 `dockerrun` 来启动我们创建的容器。使用多个 `-p` 来映射多个端口，这样我们就能同时访问 `ssh` 和 `apache` 服务了。

可以使用这个方法创建一个只有 `ssh` 服务的基础镜像，之后创建镜像可以以这个镜像为基础来创建



## 创建 **tomcat/weblogic** 集群

### 安装 **tomcat** 镜像

准备好需要的 jdk、tomcat 等软件放到 home 目录下面，启动一个虚拟机

```
docker run -t -i -v /home:/opt/data --name mk_tomcat ubuntu /bin/bash
```

这条命令挂载本地 home 目录到虚拟机的 /opt/data 目录，虚拟机内目录若不存在，则会自动创建。接下来就是 tomcat 的基本配置，jdk 环境变量设置好之后，将 tomcat 程序放到 /opt/apache-tomcat 下面 编辑 /etc/supervisor/conf.d/supervisor.conf 文件，添加 tomcat 项

```
[supervisord]
nodaemon=true

[program:tomcat]
command=/opt/apache-tomcat/bin/startup.sh

[program:sshd]
command=/usr/sbin/sshd -D
docker commit ac6474aeb31d tomcat
```

新建 tomcat 文件夹，新建 Dockerfile。

```
FROM mk_tomcat
EXPOSE 22 8080
CMD ["/usr/bin/supervisord"]
```

根据 Dockerfile 创建镜像。

```
docker build tomcat tomcat
```

### 安装 **weblogic** 镜像

步骤和 tomcat 基本一致，这里贴一下配置文件

```
supervisor.conf
[supervisord]
nodaemon=true

[program:weblogic]
command=/opt/Middleware/user_projects/domains/base_domain/bin/startWebLogic.sh

[program:sshd]
command=/usr/sbin/sshd -D
dockerfile
FROM weblogic
EXPOSE 22 7001
CMD ["/usr/bin/supervisord"]
```

## tomcat/weblogic 镜像的使用

### 存储的使用

在启动的时候，使用 `-v` 参数

`-v, --volume=[]` Bind mount a volume (e.g. from the host: `-v /host:/container`, from docker: `-v /container`)

将本地磁盘映射到虚拟机内部，它在主机和虚拟机容器之间是实时变化的，所以我们更新程序、上传代码只需要更新物理主机的目录就可以了

### tomcat 和 weblogic 集群的实现

tomcat 只要开启多个容器即可

```
docker run -d -v -p 204:22 -p 7003:8080 -v /home/data:/opt/data --name tm1 tomcat:/usr/bin/supervisord
docker run -d -v -p 205:22 -p 7004:8080 -v /home/data:/opt/data --name tm2 tomcat:/usr/bin/supervisord
docker run -d -v -p 206:22 -p 7005:8080 -v /home/data:/opt/data --name tm3 tomcat:/usr/bin/supervisord
```

这里说一下 weblogic 的配置，大家知道 weblogic 有一个域的概念。如果要使用常规的 administrator + node 的方式部署，就需要在 supervisord 中分别写出 administratorserver 和 node server 的启动脚本，这样做的优点是：

- 可以使用 weblogic 的集群，同步等概念
- 部署一个集群应用程序，只需要安装一次应用到集群上即可

缺点是：

- Docker 配置复杂了
- 没办法自动扩展集群的计算容量，如需添加节点，需要在 administrator 上先创建节点，然后再配置新的容器 supervisor 启动脚本，然后再启动容器 另外种方法是把所有的程序都安装在 adminiserver 上面，需要扩展的时候，启动多个节点即可，它的优点和缺点和上一种方法恰恰相反。（建议使用这种方式来部署开发和测试环境）

```
docker run -d -v -p 204:22 -p 7001:7001 -v /home/data:/opt/data --name node1 weblogic/usr/bin/supervisord  
docker run -d -v -p 205:22 -p 7002:7001 -v /home/data:/opt/data --name node2 weblogic/usr/bin/supervisord  
docker run -d -v -p 206:22 -p 7003:7001 -v /home/data:/opt/data --name node3 weblogic/usr/bin/supervisord
```

这样在前端使用 nginx 来做负载均衡就可以完成配置了

# 多台物理主机之间的容器互联（暴露容器到真实网络中）

Docker 默认的桥接网卡是 docker0。它只会在本机桥接所有的容器网卡，举例来说容器的虚拟网卡在主机上看一般叫做 veth\* 而 Docker 只是把所有这些网卡桥接在一起，如下：

```
[root@opnvz ~]# brctl show
bridge name      bridge id        STP enabled      interfaces
docker0          8000.56847afe9799    no               veth0889
                  veth3c7b
                  veth4061
```

在容器中看到的地址一般是像下面这样的地址：

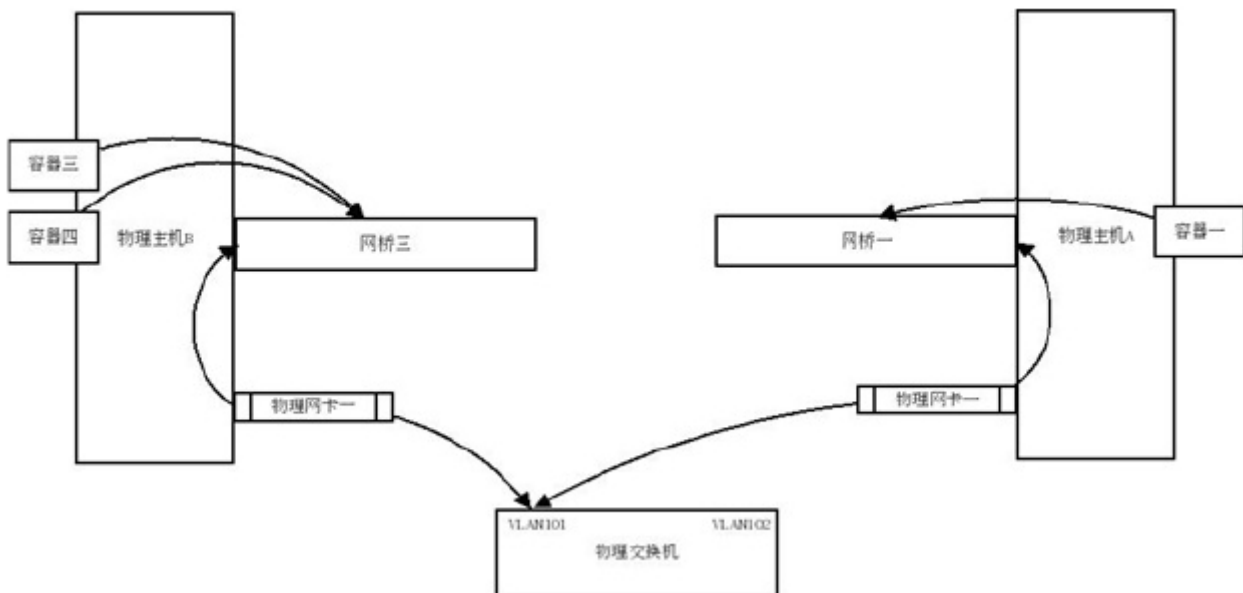
```
root@ac6474aeb31d:~# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
11: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 4a:7d:68:da:09:cf brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::487d:68ff:feda:9cf/64 scope link
        valid_lft forever preferred_lft forever
```

这样就可以把这个网络看成是一个私有的网络，通过 nat 连接外网，如果要让外网连接到容器中，就需要做端口映射，即 -p 参数。

如果在企业内部应用，或者做多个物理主机的集群，可能需要将多个物理主机的容器组到一个物理网络中来，那么就需要将这个网桥桥接到我们指定的网卡上。

## 拓扑图

主机 A 和主机 B 的网卡一般都连着物理交换机的同一个 vlan 101,这样网桥一和网桥三就相当于在同一个物理网络中了，而容器一、容器三、容器四也在同一物理网络中了，他们之间可以相互通信，而且可以跟同一 vlan 中的其他物理机器互联。



## ubuntu 示例

下面以 ubuntu 为例创建多个主机的容器联网: 创建自己的网桥,编辑 /etc/network/interface 文件

```
auto br0
iface br0 inet static
address 192.168.7.31
netmask 255.255.240.0
gateway 192.168.7.254
bridge_ports em1
bridge_stp off
dns-nameservers 8.8.8.8 192.168.6.1
```

将 Docker 的默认网桥绑定到这个新建的 br0 上面, 这样就将这台机器上容器绑定到 em1 这个网卡所对应的物理网络上了。

ubuntu 修改 /etc/default/docker 文件, 添加最后一行内容

```
# Docker Upstart and SysVinit configuration file
# Customize location of Docker binary (especially for development testing).
#DOCKER="/usr/local/bin/docker"
# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"

# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"

DOCKER_OPTS="-b=br0"
```

在启动 Docker 的时候 使用 -b 参数 将容器绑定到物理网络上。重启 Docker 服务后, 再进入容器可以看到它已经绑定到你的物理网络上了。

```
root@ubuntudocker:~# docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
58b043aa05eb    desk_hz:v1    "/startup.sh"    5 days ago    Up 2 seconds    5900/tcp, 6080/tcp, 22/tcp

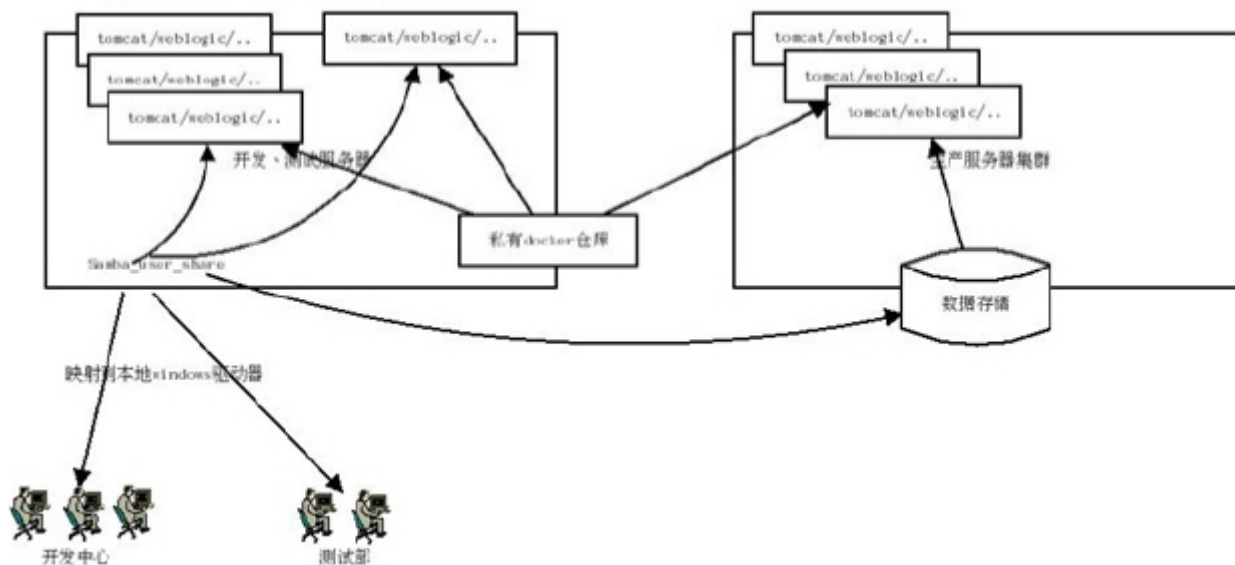
root@ubuntudocker:~# brctl show
bridge name    bridge id    STP enabled    interfaces
br0            8000.7e6e617c8d53    no            em1
               vethe6e5
```

这样就直接把容器暴露到物理网络上了，多台物理主机的容器也可以相互联网了。需要注意的是，这样就需要自己来保证容器的网络安全了。

# 标准化开发测试和生产环境

对于大部分企业来说，搭建 PaaS 既没有那个精力，也没那个必要，用 Docker 做个人的 sandbox 用处又小了点。

可以用 Docker 来标准化开发、测试、生产环境。



Docker 占用资源小，在一台 E5 128 G 内存的服务器上部署 100 个容器都绰绰有余，可以单独抽一个容器或者直接在宿主物理主机上部署 samba，利用 samba 的 home 分享方案将每个用户的 home 目录映射到开发中心和测试部门的 Windows 机器上。

针对某个项目组，由架构师搭建好一个标准的容器环境供项目组 and 测试部门使用，每个开发工程师可以拥有自己单独的容器，通过 `dockerrun -v` 将用户的 home 目录映射到容器中。需要提交测试时，只需要将代码移交给测试部门，然后分配一个容器使用 `-v` 加载测试部门的 home 目录启动即可。这样，在公司内部的开发、测试基本就统一了，不会出现开发部门提交的代码，测试部门部署不了的问题。

测试部门发布测试通过的报告后，架构师再一次检测容器环境，就可以直接交由部署工程师将代码和容器分别部署到生产环境中了。这种方式的部署横向性能的扩展性也极好。

# 安全

---

评估 Docker 的安全性时，主要考虑三个方面：

- 由内核的名字空间和控制组机制提供的容器内在安全
- Docker程序（特别是服务端）本身的抗攻击性
- 内核安全性的加强机制对容器安全性的影响
-



# 内核名字空间

---

Docker 容器和 LXC 容器很相似，所提供的安全特性也差不多。当用 `dockerrun` 启动一个容器时，在后台 Docker 为容器创建了一个独立的名字空间和控制组集合。

名字空间提供了最基础也是最直接的隔离，在容器中运行的进程不会被运行在主机上的进程和其它容器发现和作用。

每个容器都有自己独有的网络栈，意味着它们不能访问其他容器的 `sockets` 或接口。不过，如果主机系统上做了相应的设置，容器可以像跟主机交互一样的和其他容器交互。当指定公共端口或使用 `links` 来连接 2 个容器时，容器就可以相互通信了（可以根据配置来限制通信的策略）。

从网络架构的角度来看，所有的容器通过本地主机的网桥接口相互通信，就像物理机器通过物理交换机通信一样。

那么，内核中实现名字空间和私有网络的代码是否足够成熟？

内核名字空间从 2.6.15 版本（2008 年 7 月发布）之后被引入，数年间，这些机制的可靠性在诸多大型生产系统中被实践验证。

实际上，名字空间的想法和设计提出的时间要更早，最初是为了在内核中引入一种机制来实现 [OpenVZ](#) 的特性。而 OpenVZ 项目早在 2005 年就发布了，其设计和实现都已经十分成熟。

# 控制组

---

控制组是 Linux 容器机制的另外一个关键组件，负责实现资源的审计和限制。

它提供了很多有用的特性；以及确保各个容器可以公平地分享主机的内存、CPU、磁盘 IO 等资源；当然，更重要的是，控制组确保了当发生在容器内的资源压力不会连累主机系统。

尽管控制组不负责隔离容器之间相互访问、处理数据和进程，它在防止拒绝服务（DDOS）攻击方面是必不可少的。尤其是在多用户的平台（比如公有或私有的 PaaS）上，控制组十分重要。例如，当某些应用程序表现异常的时候，可以保证一致地正常运行和性能。

控制组机制始于 2006 年，内核从 2.6.24 版本开始被引入。

# Docker服务端的防护

运行一个容器或应用程序的核心是通过 Docker 服务端。Docker 服务的运行目前需要 root 权限，因此其安全性十分关键。

首先，确保只有可信的用户才可以访问 Docker 服务。Docker 允许用户在主机和容器间共享文件夹，同时不需要限制容器的访问权限，这就容易让容器突破资源限制。例如，恶意用户启动容器的时候将主机的根目录 / 映射到容器的 /host 目录中，那么容器理论上就可以对主机的文件系统进行任意修改了。这听起来很疯狂？但是事实上几乎所有虚拟化系统都允许类似的资源共享，而没法禁止用户共享主机根文件系统到虚拟机系统。

这将会造成很严重的安全后果。因此，当提供容器创建服务时（例如通过一个 web 服务器），要更加注意进行参数的安全检查，防止恶意的用户用特定参数来创建一些破坏性的容器

为了加强对服务端的保护，Docker 的 REST API（客户端用来跟服务端通信）在 0.5.2 之后使用本地的 Unix 套接字机制替代了原先绑定在 127.0.0.1 上的 TCP 套接字，因为后者容易遭受跨站脚本攻击。现在用户使用 Unix 权限检查来加强套接字的访问安全。

用户仍可以利用 HTTP 提供 REST API 访问。建议使用安全机制，确保只有可信的网络或 VPN，或证书保护机制（例如受保护的 stunnel 和 ssl 认证）下的访问可以进行。此外，还可以使用 HTTPS 和证书来加强保护。

最近改进的 Linux 名字空间机制将可以实现使用非 root 用户来运行全功能的容器。这将从根本上解决了容器和主机之间共享文件系统而引起的安全问题。

终极目标是改进 2 个重要的安全特性：

- 将容器的 root 用户映射到本地主机上的非 root 用户，减轻容器和主机之间因权限提升而引起的安全问题；
- 允许 Docker 服务端在非 root 权限下运行，利用安全可靠的子进程来代理执行需要特权权限的操作。这些子进程将只允许在限定范围内进行操作，例如仅仅负责虚拟网络设定或文件系统管理、配置操作等。

最后，建议采用专用的服务器来运行 Docker 和相关的管理服务（例如管理服务比如 ssh 监控和进程监控、管理工具 nrpe、collectd 等）。其它的业务服务都放到容器中去运行。

# 内核能力机制

能力机制 ( Capability ) 是 Linux 内核一个强大的特性，可以提供细粒度的权限访问控制。Linux 内核自 2.1 版本起就支持能力机制，它将权限划分为更加细粒度的操作能力，既可以作用在进程上，也可以作用在文件上。

例如，一个 Web 服务进程只需要绑定一个低于 1024 的端口的权限，并不需要 root 权限。那么它只需要被授权 `net_bind_service` 能力即可。此外，还有很多其他的类似能力来避免进程获取 root 权限。

默认情况下，Docker 启动的容器被严格限制只允许使用内核的一部分能力。

使用能力机制对加强 Docker 容器的安全有很多好处。通常，在服务器上会运行一堆需要特权权限的进程，包括有 ssh、cron、syslogd、硬件管理工具模块（例如负载模块）、网络配置工具等等。容器跟这些进程是不同的，因为几乎所有的特权进程都由容器以外的支持系统来进行管理。

- ssh 访问被主机上ssh服务来管理；
- cron 通常应该作为用户进程执行，权限交给使用它服务的应用来处理；
- 日志系统可由 Docker 或第三方服务管理；
- 硬件管理无关紧要，容器中也就不需执行 udevd 以及类似服务；
- 网络管理也都在主机上设置，除非特殊需求，容器不需要对网络进行配置。

从上面的例子可以看出，大部分情况下，容器并不需要“真正的” root 权限，容器只需要少数的能力即可。为了加强安全，容器可以禁用一些没必要的权限。

- 完全禁止任何 mount 操作；
- 禁止直接访问本地主机的套接字；
- 禁止访问一些文件系统的操作，比如创建新的设备、修改文件属性等；
- 禁止模块加载。

这样，就算攻击者在容器中取得了 root 权限，也不能获得本地主机的较高权限，能进行的破坏也有限。

默认情况下，Docker采用 [白名单](#) 机制，禁用 [必需功能](#) 之外的其它权限。当然，用户也可以根据自身需求来为 Docker 容器启用额外的权限。

## 其它安全特性

---

除了能力机制之外，还可以利用一些现有的安全机制来增强使用 Docker 的安全性，例如 TOMOYO, AppArmor, SELinux, GRSEC 等。

Docker 当前默认只启用了能力机制。用户可以采用多种方案来加强 Docker 主机的安全，例如：

- 在内核中启用 GRSEC 和 PAX，这将增加很多编译和运行时的安全检查；通过地址随机化避免恶意探测等。并且，启用该特性不需要 Docker 进行任何配置。
- 使用一些有增强安全特性的容器模板，比如带 AppArmor 的模板和 Redhat 带 SELinux 策略的模板。这些模板提供了额外的安全特性。
- 用户可以自定义访问控制机制来定制安全策略。

跟其它添加到 Docker 容器的第三方工具一样（比如网络拓扑和文件系统共享），有很多类似的机制，在不改变 Docker 内核情况下就可以加固现有的容器。

## 总结

---

总体来看，Docker 容器还是十分安全的，特别是在容器内不使用 root 权限来运行进程的话。

另外，用户可以使用现有工具，比如 Apparmor, SELinux, GRSEC 来增强安全性；甚至自己在内核中实现更复杂的安全机制。

# Table of Contents

前言	2
Docker 简介	4
什么是 Docker	5
为什么要用 Docker	6
基本概念	7
镜像	8
容器	9
仓库	10
安装	11
Ubuntu	12
CentOS	13
镜像	14
获取镜像	15
列出	16
创建	17
Dockerfile	21
基本结构	22
指令	24
创建镜像	27
底层实现	28
基本架构	29
名字空间	30
控制组	31
Union 文件系统	32
容器格式	33
网络	34
附录一：命令查询	36
附录二：常见仓库介绍	41
Ubuntu	42
CentOS	43
MySQL	44
MongoDB	45
Redis	46
Nginx	47
WordPress	48
Node.js	49
附录三：资源链接	50
存出和载入	51
移除	52
实现原理	53
容器	54
启动	55
守护态运行	57
终止	58
进入容器	59
导出和导入	61
删除	62
仓库	63
DockerHub	64
私有仓库	66
配置文件	70
数据管理	72
数据卷	73
数据卷容器	75
备份、恢复、迁移数据卷	76

使用网络	77
外部访问容器	78
容器互联	80
高级网络配置	83
快速配置指南	84
配置 DNS	85
容器访问控制	86
端口映射实现	88
配置 docker0 网桥	89
自定义网桥	90
工具和示例	91
编辑网络配置文件	92
实例：创建一个点到点连接	93
实战案例	94
使用 Supervisor 来管理进程	95
创建 tomcat/weblogic 集群	97
多台物理主机之间的容器互联	100
标准化开发测试和生产环境	103
安全	104
内核名字空间	105
控制组	106
服务端防护	107
内核能力机制	108
其它安全特性	109
总结	110