目錄

项目说明

环境准备

从零开始学 **docker**

Develop with Docker

配置容器网络

管理应用数据

生产环境中应用

docker-compose

997.docker-compose 实战

998.dockerfiles 实战

Docker 17.09 官方文档中文笔记

基于官网文档 17.09

> 建议有英语能力的人直接到 docker 官网文档 去看。

官方目录结构

小白入门目录结构

项目地址

- **GitHub** 项目地址: https://github.com/octowhale/gitbooks-docker2-docs/
- **GitBooks** 在线阅读: https://www.gitbook.com/read/book/octowhale/docker-doc-cn
  - 下载 PDF
  - 下载 EPUB
  - 下载 MOBI
- 国内 七牛 镜像(可能过期): http://p05r4f6dx.bkt.clouddn.com

> 尽量到 `GitHub` 和 `GitBook` 上面阅读

常见问题

- 如果你对 Docker 没有整体概念，或者你想了解更过的 Docker 生态，参考 **Cloudman** 的 《每天 5 分钟玩转 Docker
- 如果你对 Docker 不是很了解，还有很多问题，可以先去看看 蜗牛大哥 的 《常见 docker 100 问》。在这里几乎能找到你入门的所有答案。
- 学会如何聪明的提问 《提问的智慧》
- 学会了 《提问的智慧》， 那就 《别提傻逼问题》

勘误

如果你有心帮助后来人，发现文章里面有错误的地方，

- **GitBooks**: 请通过 `gitbooks 下方的 disqus` 留言。

- **GitHub Issus**: https://github.com/octowhale/gitbooks-docker2-docs/issues

准备工作

如果你还没有安装 docker， 可以选择直接到 docker 官网 寻找操作系统发行版的安装方式。

- 为了保证 Docker 最新功能，尽量选择 `ubuntu 16.04` 上安装，可以直接看 在 ubuntu 16.04 上安装 docker-ce。
- 不要 **CentOS 6** 安装 Docker，不要安装 **CentOS 7** 自带源中的 Docker

代理服务器

在本地测试环境中，最好搭建一个『仓库代理服务器』。如果你用的是大水管，那当我没说。搭建 `代理服务器` 的时候，使用 `国内镜像源` ，可以加速下载。

- 启动『仓库代理服务器』

镜像服务器

国内镜像源。其实用诸如 aliyun、 daocloud 的源也没关系。这里提供两个不用注册即可使用的官方镜像源。配置方法参考 启动『仓库代理服务器』

- docker-cn 官方： `https://registry.docker-cn.com`
- 中科大： `https://docker.mirrors.ustc.edu.cn`

Lisences

本项目遵循 `GNU GENERAL PUBLIC LICENSE Version 3`

捐助

公益捐助

当然，更推荐各位进行公益捐助，让更多的孩子有书可读。

- 腾讯公益平台 (http://gongyi.qq.com)
- 阿里公益平台 (https://love.alipay.com/)

安装 docker ce

基于 `ubuntu 16.04`

修改 ubuntu 镜像源

1. 下载 ubuntu_tuna_sources.sh
2. `bash ubuntu_tuna_sources.sh`

安装 docker ce

1. 下载 install_dockerce_ubuntu.sh
2. `bash install_dockerce_ubuntu.sh cn`

```
#!/bin/bash
#
# install_dockerce_ubuntu.sh
#
# for ubuntu 16.04.3
#


echo $1 |grep -i cn

[ $? -eq 0 ] && area=cn


sudo apt-get update

sudo apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo a
```

```
pt-key add -


if [ "$area" == "cn" ]
then
{
    # 使用 aliyun repo
    # sudo add-apt-repository \
    #    "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/
linux/ubuntu \
    #    $(lsb_release -cs) \
    #    stable"

    # 使用 中科大 repo
    sudo add-apt-repository \
        "deb [arch=amd64] https://mirrors.ustc.edu.cn/docker-ce/
linux/ubuntu \
        $(lsb_release -cs) \
        stable"
}
else
{
    # 使用 docker repo
    sudo add-apt-repository \
       "deb [arch=amd64] https://download.docker.com/linux/ubunt
u \
       $(lsb_release -cs) \
       stable"
}
fi


sudo apt-get update
sudo apt-get install -y docker-ce

## 修改 image 源
if [ "$area" == "cn" ]
then
{
    sudo sed -i --follow-symlinks '/^ExecStart/s/\(.*\)/\1 --reg
```

```
istry-mirror=https:\/\/docker.mirrors.ustc.edu.cn/' /etc/systemd
/system/multi-user.target.wants/docker.service

    sudo systemctl daemon-reload
    sudo systemctl restart docker
}
fi
```

修改 docker 仓库源

编辑 `/etc/systemd/system/multi-user.target.wants/docker.service` 文件，在启动命令行 `ExecStart` 行后面加上镜像仓库。

国内两个公共镜像仓库

- 中科大： `https://docker.mirrors.ustc.edu.cn/`
- docker中国： `https://registry.docker-cn.com`

```
ExecStart ... --registry-mirror=https://docker.mirrors.ustc.edu.
cn/
```

修改完成后，重启 docker

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

Docker 仓库代理服务器

镜像源与配置

由于国情原因，在国内从过来 `pull` 镜像是非常痛苦的。

好在国内的前辈提供了一些国内镜像源，可以很好的加速。 两个公开镜像源，不需要注册用户

- docker-cn 官方： `https://registry.docker-cn.com`
- 中科大： `https://docker.mirrors.ustc.edu.cn`

增加镜像源

`ubuntu 16.04` / `docker 17.06`

```
# 1. 修改 daemon 配置
$ sudo vi  /etc/systemd/system/multi-user.target.wants/docker.service
ExecStart=/usr/bin/dockerd -H fd:// --registry-mirror=https://docker.mirrors.ustc.edu.cn/

# 2. 重启
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

镜像仓库代理

即使配置了国内源，但反复下载大镜像或者有个小水管也是很恼火的事情。

一般这种情况下有三种考虑：

- 自建镜像仓库， `registry`
- 自建镜像仓库代理， `registry:2`
- 二合一。 官方 `registry` 镜像无法实现，可以通过 `nexus` 镜像实现。

启动仓库

1. 下载代理配置好的镜像仓库代理包 registry_proxy.tar.gz
2. 解压并运行包中的 `bash start.sh`

> 注意: 需要依赖 docker-compose 组件

1. 将前面介绍的 `docker daemon` 配置的镜像，替换成你当前的镜像仓库代理路径即可。

以下是具体配置

镜像仓库代理服务器配置

```
# config.yml

version: 0.1
log:
  fields:
    service: registry
storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
http:
  addr: :5000
  headers:
    X-Content-Type-Options: [nosniff]
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
proxy:
  remoteurl: https://registry.docker-cn.com
```

docker-compose 发布文件

```
# docker-compose.yml

version: '2'
services:
  mirror:
    # restart: always 永远保持运行
    restart: always
    image: registry:2
    ports:
      - "5000:5000"
    volumes:
      - ./config.yml:/etc/docker/registry/config.yml
```

启动

```
#!/bin/bash

source ~/.bashrc

which docker-compose || {

    echo "sudo curl -L https://github.com/docker/compose/releases/download/1.16.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose"
    echo "sudo chmod +x /usr/local/bin/docker-compose"

}

cd $(dirname $0)

docker-compose -f docker-compose-registry-proxy.yml  up -d
```

docker daemon.json 配置文件

`daemon.json` 配置方式

- `Linux`：`/etc/docker/daemon.json`
- `Windows Server`：`C:\ProgramData\docker\config\daemon.json`
- `Docker for Mac` / `Docker for Windows`：Click the Docker icon in the toolbar, select `Preferences`, then select `Daemon`. Click `Advanced`.

daemon.json 配置

镜像加速器

```
// 配置一个
{
  "registry-mirrors": ["https://registry.docker-cn.com"]
}

// 配置多个
{
  "registry-mirrors": ["https://registry.docker-cn.com","https://docker.mirrors.ustc.edu.cn"]
}
```

> 镜像加速器常用值：
>
> > `docker-cn 官方`：`https://registry.docker-cn.com`
> >
> > `中科大`：`https://docker.mirrors.ustc.edu.cn`

日志

```
{
  "debug": true,
  "log-level": "info"
}
```

> `log-level` 的有效值包括: `debug` , `info` , `warn` , `error` , `fatal`

监控 Prometheus

> https://docs.docker.com/engine/admin/prometheus/#configure-docker

```
{
  "metrics-addr" : "127.0.0.1:9323",
  "experimental" : true
}
```

保持容器在线

> https://docs.docker.com/engine/admin/live-restore/#enable-the-live-restore-option

当 `dockerd` 进程死掉后，依旧保持容器存活。

```
{
  "live-restore": true
}
```

Linux 重载 docker daemon

```
$ sudo kill -SIGHUP $(pidof dockerd)
```

设置 `镜像、容器、卷` 存放目录和驱动

> https://docs.docker.com/engine/admin/systemd/#runtime-directory-and-storage-driver

下述两个参数可以单独使用

```
{
    "graph": "/mnt/docker-data",
    "storage-driver": "overlay"
}
```

`graph`：设置存放目录

- `Docker Root Dir: /mnt/docker-data`

`storage-driver`：设置存储驱动

- `Storage Driver: overlay`

user namespace remap

> https://docs.docker.com/engine/security/userns-remap/#enable-userns-remap-on-the-daemon

安全设置： 用户空间重映射

`userns-remap` 的值可以是 如果值字段 `只有` 一个值，那么该字段表示 `组`。如果需要同时指定 `用户` 和 `组`，需要使用 `冒号` 分隔，格式为 `用户:组`

- `组`
- `用户:组`
- `组` 或 `用户` 的值可以是组或用户的 `名称` 或 `ID`。
  - `testuser`
  - `testuser:testuser`
  - `1001`
  - `1001:1001`
  - `testuser:1001`
  - `1001:testuser`

```
{
  "userns-remap": "testuser"
}

// 或同时指定 用户和组，且使用 名称和ID
{
  "userns-remap": "testuser:1001"
}
```

```
$ dockerd --userns-remap="testuser:testuser"
```

> `userns-remap` 使用不多，但并不是不重要。目前不是默认启用的原因是因为一些应用会假定 uid 0 的用户拥有特殊能力，从而导致假定失败，然后报错退出。所以如果要启用 **user id remap**，你要充分测试一下。但是启用 uid remap 的安全性提高是明显的。

1. Edit `/etc/docker/daemon.json`. Assuming the file was previously empty, the following entry will enable `userns-remap` using user and group called `testuser`. You can address the user and group by ID or name. You only need to specify the group name or ID if it is different from the user name or ID. If you provide both the user and group name or ID, separate them by a colon (`:`) character. The following formats will all work for the value, assuming the UID and GID of `testuser` are `1001`:
   - `testuser`
   - `testuser:testuser`
   - `1001`
   - `1001:1001`
   - `testuser:1001`
   - `1001:testuser`

Get started, Par1: Orientation and set up

欢迎！很高兴您希望了解如何使用 Docker。

本教程包含六个部分，您将在其中：

1. 在此页面上了解基本设置与入门。
2. 构建并运行您的第一个应用
3. 将您的应用变为扩展服务
4. 在多台机器之间扩展您的服务
5. 添加持久保存数据的访客计数器
6. 将您的 swarm 部署到生产中

应用自身非常简单，以确保您不会因代码的操作而分心。但是，Docker 的价值在于如何构建、交付和运行应用；您的应用的实际操作完全不可知。

先决条件

虽然我们将在教程中定义概念，但您最好在开始之前了解 Docker 是什么以及为何使用 Docker。

另外，在继续之前，我们还需要假定您熟悉一些概念：

- IP 地址和端口
- 虚拟机
- 编辑配置文件
- 基本熟悉代码依赖项和构建的概念
- 机器资源使用情况属于，例如 CPU 百分比、RAM 使用量（字节）等。

对容器的简要说明

镜像 是一种轻量级、可执行的独立软件包，它包含运行某个软件所需的所有内容，包括代码、运行时、库、环境变量和配置文件。

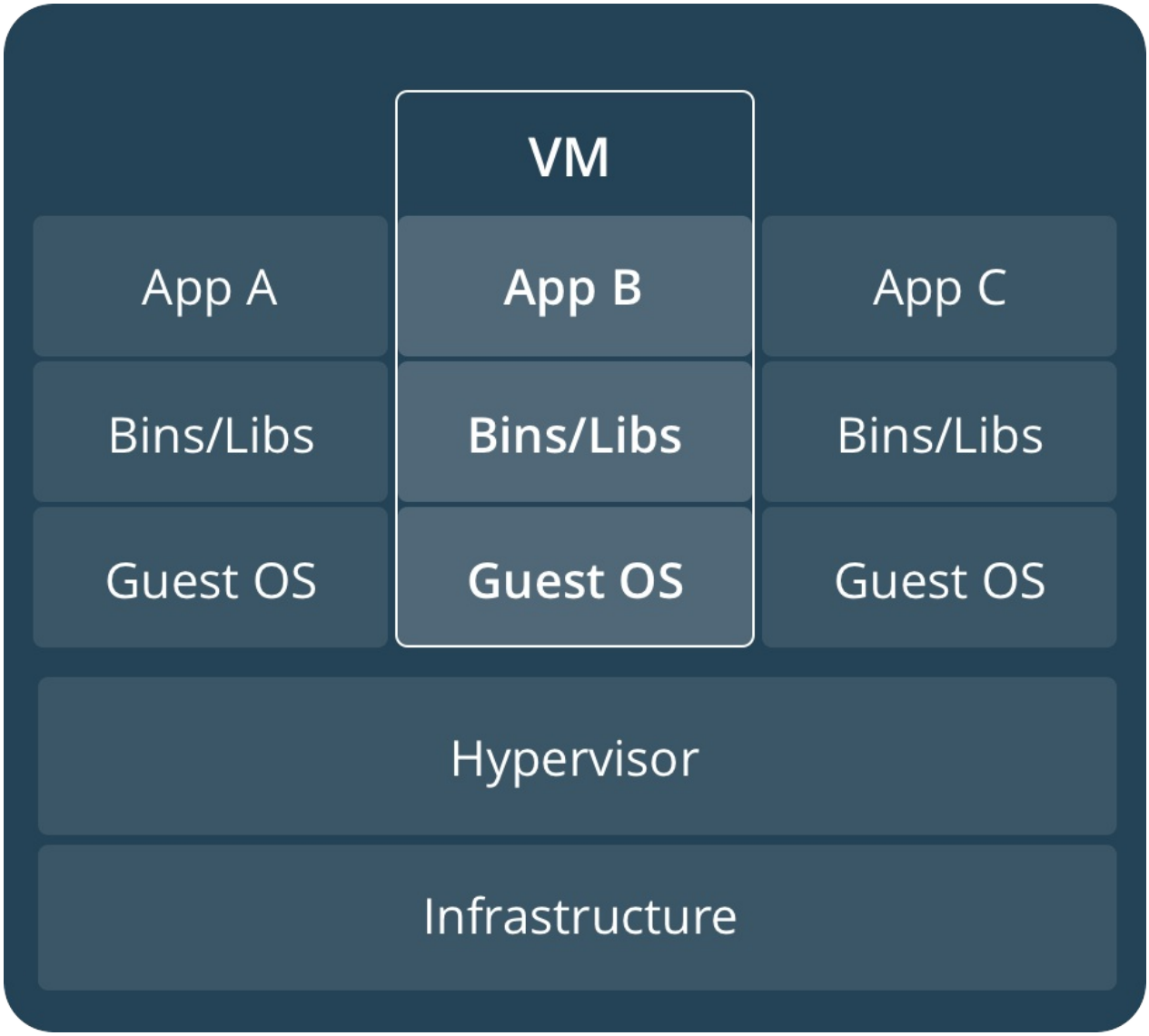
容器 是镜像的运行时实例 - 实际执行时镜像会在内存中变成什么。默认情况下，它完全独立于主机环境运行，仅在配置为访问主机文件和端口的情况下才执行此操作。

容器在主机内核上以本机方式运行应用。与仅通过管理程序对主机资源进行虚拟访问的虚拟机相比，它们具有更好的性能特征。容器可以获取本机访问，每个容器都在独立进程中运行，占用的内存不超过任何其他可执行文件。

容器与虚拟机

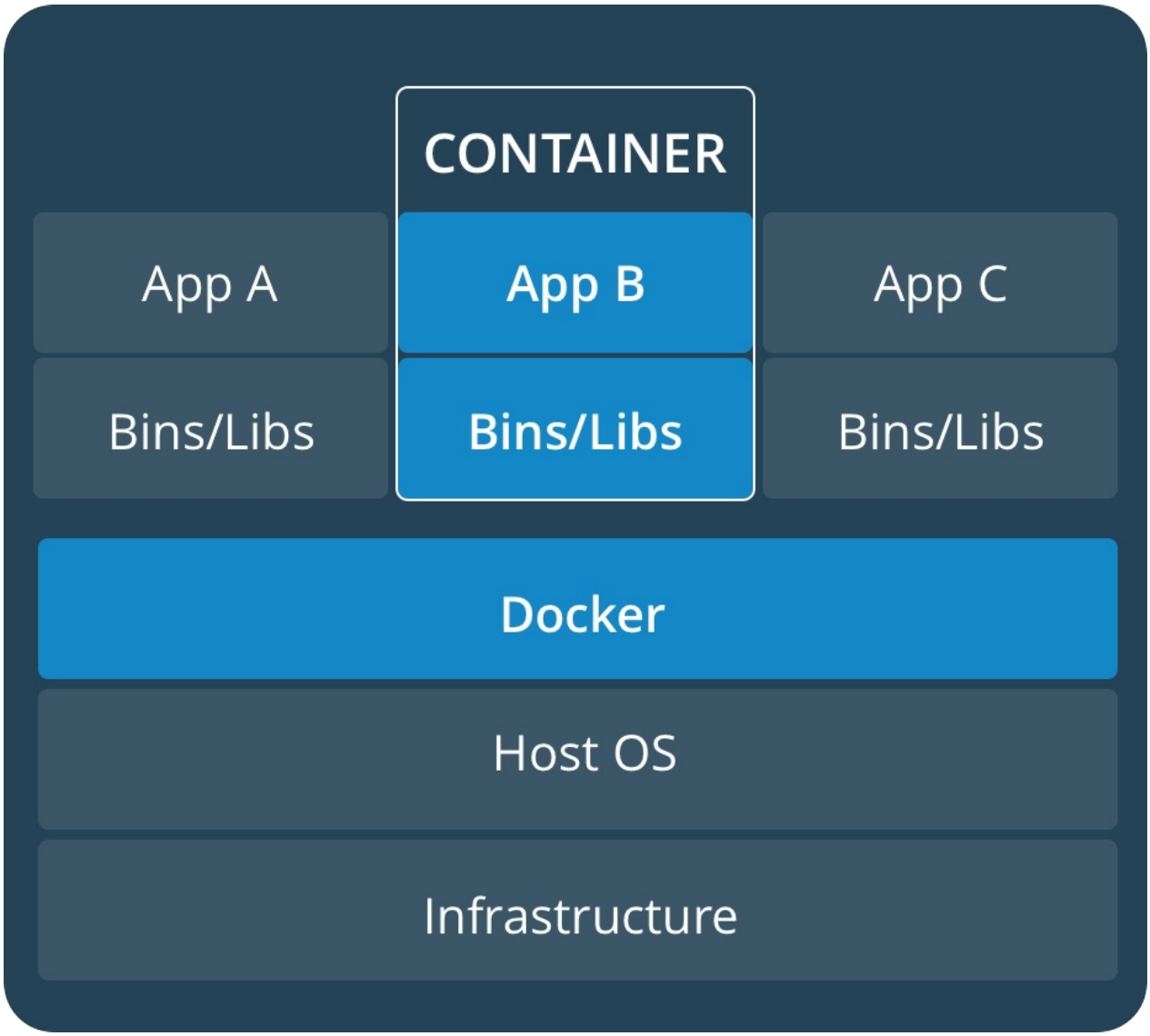
在比较虚拟机和容器时，请考虑此图：

虚拟机图



虚拟机运行来宾操作系统 - 请注意每个框中的操作系统层。此项为资源密集型，并且生成的磁盘镜像和应用状态与操作系统设置、系统安装的依赖项、操作系统安全补丁以及其他容易丢失且难以复制的临时配置相关联。

容器图



容器可以 `共享单个内核` ，并且需要存在于容器镜像中的唯一信息是可执行文件及其软件包依赖项，这些都不需要在主机系统上安装。这些进程的运行方式类似于原生进程，并且您可以通过运行 `docker ps` 等命令来逐一管理它们 - 如同您在 Linux 上运行 `ps` 以查看活动进程一样。最后，由于它们包含所有依赖项，因此不存在配置关联；容器化应用“可以随处运行”。

设置

在开始之前，请确保您的系统安装了最新版本的 Docker。

- 官方文档中提供所有操作系统的安装方法，看这里安装 Docker
- 如果你想在 ubuntu 下安装最新版本的 docker，可以直接看这里 ubuntu 16.04 下安装 docker-ce 17.09

> 注：需要版本 1.13 或更高版本

您将能够运行 `docker run hello-world` 并看到类似于以下内容的响应：

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working
correctly.

To generate this message, Docker took the following steps:
...(snipped)...
```

现在，应确保您使用的是版本 1.13 或更高版本。运行 `docker --version` 进行确认。

```
$ docker --version
Docker version 17.05.0-ce-rc1, build 2878a85
```

如果您看到类似于以上内容的消息，表示已做好准备工作。

总结

将可扩展单位做成单个可移植的可执行文件，具有广泛的意义。它表示，`CI/CD` 可以将更新推送到分布式应用程序的任何部分，您无需担心系统依赖项，并且资源密度将增加。编排扩展动作将启动新的可执行文件而不是新的虚拟主机。

我们将了解所有这些内容，但首先需要从基础内容开始。

Get started, par2: Containers

先决条件

- 安装 Docker 版本 1.13 或更高版本。
- 阅读 第 1 部分 中的新用户导引。
- 对环境进行快速的测试运行，以确保您已做好准备：

```
docker run hello-world
```

简介

现在可以开始按照 Docker 方式构建应用。我们将从此类应用的层次结构的底层（即，`容器`）开始，这是本页面上涵盖的内容。在此级别之上是服务，它定义了容器在生产中的行为方式（请参阅 第 3 部分）。最后，处于最高级别的是技术栈，用于定义所有服务的交互（请参阅 第 5 部分）。

- 技术栈
- 服务
- 容器（您在此处）

您的新开发环境

过去，如果要开始编写 Python 应用，您的第一项业务是将 Python 运行时安装到机器上。但是，这会导致机器上的环境必须如此，才能使应用按预期运行；对于运行应用的服务器来说，也同样如此。

借助 Docker，您只需将可移植的 Python 运行时抓取为镜像，而无需进行安装。然后，您的构建可以将基本 Python 镜像与应用代码包含在一起，从而确保应用、其依赖项及运行时都一起提供。

这些可移植的镜像由 `Dockerfile` 定义。

使用 `Dockerfile` 定义容器

`Dockerfile` 将在您的容器内定义环境中执行的操作。对网络接口和磁盘驱动器等资源的访问在此环境内实现虚拟化，这将独立于系统的其余部分，因此您必须将端口映射到外部，并具体说明您要“复制”到该环境的文件。但是，在执行此操作后，您可以期望此 `Dockerfile` 中定义的应用构建的行为在运行时始终相同。

`Dockerfile`

创建空目录并将名为 `Dockerfile` 的此文件放入其中。记录说明每个语句的注释。

```
# 将官方 Python 运行时用作父镜像
FROM python:2.7-slim

# 将工作目录设置为 /app
WORKDIR /app

# 将当前目录内容复制到位于 /app 中的容器中
ADD . /app

# 安装 requirements.txt 中指定的任何所需软件包
RUN pip install -r requirements.txt

# 使端口 80 可供此容器外的环境使用
EXPOSE 80

# 定义环境变量
ENV NAME World

# 在容器启动时运行 app.py
CMD ["python", "app.py"]
```

此 `Dockerfile` 引用了我们尚未创建的内容，名为 `app.py` 和 `requirements.txt` 。在后续步骤中，我们会准备好这些内容。

> 设置代理服务器 >
>
> > 如果你在代理服务器后面，需要在 `Dockerfile` 中添加以下行，使用 `ENV` 命令指定代理服务器的主机和端口。

```
# 设置代理服务器， 替换 host:/port 为你代理服务器对应的值

ENV http_proxy host:/port
ENV https_proxy host:/port
```

The app itself

抓取这两个文件并将其放入 `Dockerfile` 所在的文件夹。这将完成我们的应用，如您所见它非常简单。将上述 `Dockerfile` 构建到镜像中时，由于 `Dockerfile` 的 `ADD` 命令而显示 `app.py` 和 `requirements.txt`，并且借助 `EXPOSE` 命令，将能够通过 HTTP 访问 `app.py` 的输出。

`requirements.txt`

```
Flask
Redis
```

`app.py`

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, sock
et_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</
i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname
=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

现在，我们可以看到，`pip install -r requirements.txt` 将安装 Python 的 Flask 和 Redis 库，而此应用将输出环境变量 `NAME` 以及对 `socket.gethostname()` 调用的输出。最后，由于 Redis 未在运行（因为我们仅安装了 Python 库，而未安装 Redis 自身），因此我们应期望尝试在此处使用它将失败并生成错误消息。

> 注：在容器内时访问主机的名称将检索容器 ID，这类似于正在运行的可执行文件的进程 ID。

构建应用

好的！您不需要 Python 或系统上 `requirements.txt` 中的任何内容，也不会构建或运行此镜像来将它们安装在系统上。似乎您尚未真正使用 Python 和 Flask 设置环境，但已进行设置。

以下是 ls 应显示的内容：

```
$ ls
Dockerfile        app.py            requirements.txt
```

现在，运行构建命令。这将创建 Docker 镜像，我们将使用 `-t` 对其进行标记，为创建的镜像命令。

```
docker build -t friendlyhello .
```

您已构建的镜像在何处？它位于您的机器上的本地 Docker 镜像库中：

```
$ docker images

REPOSITORY            TAG                 IMAGE ID
friendlyhello         latest              326387cea398
```

运行应用

运行应用，使用 `-p` 参数将机器的 4000 端口映射到容器暴露的 80 端口：

```
docker run -p 4000:80 friendlyhello
```

您将看到容器中的 Python 正在为应用提供服务（网址为 `http://0.0.0.0:80`）的通知。但是，该消息来自容器内部，但容器不知道你将宿主机的 `4000` 端口映射到了容器的 `80` 端口，因此需要将正确 URL 更改为 `http://localhost:4000`。

在 Web 浏览器中访问该 URL，以查看网页上提供的显示内容，包括“Hello World”文本、容器 ID以及 Redis 错误消息。

浏览器中的 Hello World

您还可以在 shell 中使用 curl 命令查看相同内容。

```
$ curl http://localhost:4000

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits
:</b> <i>cannot connect to Redis, counter disabled</i>
```

> 注：此端口重映射 `4000:80` 用于说明您在 `Dockerfile` 中暴露的内容与使用 `docker run -p` 发布的内容之间的差异。在后续步骤中，我们只需将主机上的 80 端口映射到容器中的 80 端口并使用 `http://localhost` 。

在终端上按 `CTRL+C` 退出。

使用 `-d` 在后台运行容器。

```
docker run -d -p 4000:80 friendlyhello
```

您将获得应用的长容器 ID，然后将返回到终端。容器现在在后台运行。您还可以使用 `docker ps` 查看缩写容器 ID（这两种 ID 可以在运行命令时交换工作）：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
TED
1fa4ab2cf395        friendlyhello       "python app.py"     28 s
econds ago
```

您将看到 `CONTAINER ID` 与 `http://localhost:4000` 上的内容相匹配。

现在，使用 `docker stop` 及 `CONTAINER ID` 结束该进程，如下所示：

```
docker stop 1fa4ab2cf395
```

共享镜像

为了说明我们刚才创建的可移植性，可以上传已构建的镜像并在其他地方运行它。但是，在您要将容器部署到生产环境中时，需要了解如何推送到镜像库。

镜像库是镜像仓库的集合，而镜像仓库是镜像的集合 - 除了代码已构建之外，类似于 GitHub 镜像仓库。镜像库中的一个帐户可以创建很多镜像仓库。默认情况下，`docker CLI` 使用 Docker 的公用镜像库。

> 注：我们将在此处使用 Docker 的公用镜像库，仅仅因为它是免费的并且经过预先配置，但有许多公用镜像库可供选择，并且您甚至可以使用 Docker Trusted Registry 设置您自己的专用镜像库。

使用 Docker ID 登录

如果您没有 Docker 帐户，请在 cloud.docker.com 中进行注册。记录您的用户名。

登录本地机器上的 Docker 公用镜像库。

```
docker login
```

标记镜像

用于将本地镜像与镜像库中的镜像仓库相关联的表示法为 `username/repository:tag` 。tag 是可选项，但建议使用它，因为这是镜像库用于为 Docker 镜像指定版本的机制。针对上下文为镜像库和 tag 指定有意义的名称，例如 `get-started:part1` .这会将镜像放入 `get-started` 镜像仓库并将其标记为 `part1` 。

现在，将其合并到一起，以标记镜像。使用您的用户名、镜像仓库和标签名称运行 `docker tag image` ，以便镜像将上传到所需目的地。此命令的语法为：

```
docker tag image username/repository:tag
```

例如：

```
docker tag friendlyhello john/get-started:part1
```

运行 `docker images` 以查看新标记的镜像。（您还可以使用 `docker image ls` 。）

> 更推荐使用 `docker image ls` 这种方法。docker 功能现在更加模块化，所有模块中，第二参数基本一致，但有一些用法不同。使用这种方法能方便，且更容易回顾操作动作。

```
$ docker images
REPOSITORY                TAG                 IMAGE ID
 CREATED             SIZE
friendlyhello             latest              d9e555c53008
 3 minutes ago       195MB
john/get-started          part1               d9e555c53008
 3 minutes ago       195MB
python                    2.7-slim            1c7128a655f6
 5 days ago          183MB
```

发布镜像

将已标记的镜像上传到镜像仓库：

```
docker push username/repository:tag
```

完成后，将公开此上传的结果。如果登录 Docker Hub，可以使用其 pull 命令看到新的镜像。

从远程镜像仓库中拉取并运行镜像

从现在开始，您可以使用 `docker run` ，并且可以使用以下命令在任何机器上运行您的应用：

```
docker run -p 4000:80 username/repository:tag
```

如果镜像在机器本地不可用，Docker 将从镜像仓库中拉取它。

```
$ docker run -p 4000:80 john/get-started:part1
Unable to find image 'john/get-started:part1' locally
part1:Pulling from orangesnap/get-started
10a267c67f42:Already exists
f68a39a6a5e4:Already exists
9beaffc0cf19:Already exists
3c1fe835fb6b:Already exists
4c9f1fa8fcb8:Already exists
ee7d8f576a14:Already exists
fbccdcced46e:Already exists
Digest: sha256:0601c866aab2adcc6498200efd0f754037e909e5fd42069ad
eff72d1e2439068
Status: Downloaded newer image for john/get-started:part1
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

> 注：如果您未指定这些命令中的 `:tag` 部分，在进行构建和运行镜像时，将使用标签 `:latest` 。Docker 将使用在未指定标签的情况下运行的镜像的最新版本（可以不是最新镜像）。 无论 `docker run` 在何处执行，它将从 `requirements.txt` 拉取您的镜像及 Python 和所有依赖项，然后运行代码。所有内容都在一个小软件包中提供，并且主机只需安装 Docker 来运行它。

第 2 部分总结 以上是此页面的所有内容。在下一节中，我们学习如何通过在服务中运行此容器来扩展应用。

继续阅读第 3 部分 »

概要和速查表（可选）

以下是关于本页面所涵盖内容的终端记录：

这里有一个视频，自行到官网查看。

以下是此页面上的基本 Docker 命令列表，以及一些相关命令（如果您要在继续之前进行进一步探索）。

```
docker build -t friendlyname .# 使用此目录的 Dockerfile 创建镜像
docker run -p 4000:80 friendlyname  # 运行端口 4000 到 90 的“友好名称”映射
docker run -d -p 4000:80 friendlyname         # 内容相同，但在分离模式下
docker ps                                 # 查看所有正在运行的容器的列表
docker stop <hash>                     # 平稳地停止指定的容器
docker ps -a           # 查看所有容器的列表，甚至包含未运行的容器
docker kill <hash>                   # 强制关闭指定的容器
docker rm <hash>                # 从此机器中删除指定的容器
docker rm $(docker ps -a -q)           # 从此机器中删除所有容器
docker images -a                               # 显示此机器上的所有镜像
docker rmi <imagename>            # 从此机器中删除指定的镜像
docker rmi $(docker images -q)             # 从此机器中删除所有镜像
docker login             # 使用您的 Docker 凭证登录此 CLI 会话
docker tag <image> username/repository:tag  # 标记 <image> 以上传到镜像库
docker push username/repository:tag            # 将已标记的镜像上传到镜像库
docker run username/repository:tag                   # 运行镜像库中的镜像
```

service

创建 swarm

```
$ docker swarm init --advertise-addr 192.168.56.205
Swarm initialized: current node (9fu8b0ot055ayazhcxu61jv33) is n
ow a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-1onpqfhmh9krhfkdw5bucfvr4
vriulm0ber2f1gmbrqp7eh0m0-ewm3hnddibbq45492igzx2rxr 192.168.56.2
05:2377

To add a manager to this swarm, run 'docker swarm join-token man
ager' and follow the instructions.
```

发布 stack

```
$ docker stack deploy -c docker-compose.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_web
```

Let's inspect one task and limit the ouput to container ID:

通过 `docker service` 查询 `service` 信息

Get the service ID for the one service in our application:

```
$ docker service ls
ID                  NAME                MODE                REPL
ICAS            IMAGE                          PORTS
jhkiqvpxov65        getstartedlab_web   replicated          3/3
                uyinn28/friendlyhello:latest   *:80->80/tcp
```

Docker swarms run tasks that spawn containers. Tasks have state and their own IDs:

```
$ docker service ps getstartedlab_web

# 结果中的 ID 就是 task ID
ID                  NAME                      IMAGE
             NODE                DESIRED STATE       CURRENT STA
TE            ERROR                PORTS
igyp06sdql0s        getstartedlab_web.1       uyinn28/friendlyhe
llo:latest   S005_Ubuntu1604S    Running             Running 6 m
inutes ago
wuc02ndv48f7         \_ getstartedlab_web.1   uyinn28/friendlyhe
llo:latest   S005_Ubuntu1604S    Shutdown            Shutdown 6
minutes ago
qsssyx6rwz07        getstartedlab_web.6       uyinn28/friendlyhe
llo:latest   S005_Ubuntu1604S    Running             Running 6 m
inutes ago
to0xg7tp59bk        getstartedlab_web.10      uyinn28/friendlyhe
llo:latest   S005_Ubuntu1604S    Running             Running 6 m
inutes ago
```

通过 **`docker container`** 查询 **`container`** 的信息

```
$ docker container ps -a
CONTAINER ID        IMAGE                             COMMAND
            CREATED             STATUS                          POR
TS                      NAMES
ea8167447298        uyinn28/friendlyhello:latest   "python app.p
y"          10 minutes ago      Up 9 minutes                    80/
tcp                     getstartedlab_web.6.qsssyx6rwz07ppch07jud2
rnk
100bcb7eb051        uyinn28/friendlyhello:latest   "python app.p
y"          10 minutes ago      Up 9 minutes                    80/
tcp                     getstartedlab_web.10.to0xg7tp59bkshrrvgf4y
hutp
e12bb83371fd        uyinn28/friendlyhello:latest   "python app.p
y"          10 minutes ago      Up 9 minutes                    80/
tcp                     getstartedlab_web.1.igyp06sdql0s8eo0eyg6dz
9lq
354f388506c7        uyinn28/friendlyhello:latest   "python app.p
y"          36 minutes ago      Exited (137) 9 minutes ago
                        getstartedlab_web.1.wuc02ndv48f7jp71sm0s0j
ya6
b712972c07d5        registry:2                        "/entrypoint.
sh /e..."   24 hours ago        Up 24 hours                     0.0
.0.0:5000->5000/tcp   registry

# 注意，<task> 是 NAMES 对应值中，以 `.` 分割的最后一位。
```

task_id 和 container_id 互查

```
# 通过 task 值查询 container id
$ docker inspect --format='{{.Status.ContainerStatus.ContainerID
}}' <task>

$ docker inspect --format='{{.Status.ContainerStatus.ContainerID
}}' igyp06sdql0s
e12bb83371fd9a661e34d16e69dc7c56bb0a832d92ecd620042823da6a807725
```

Vice versa, inspect the container ID, and extract the task ID:

```
# 通过 container id 查询 task id
$ docker inspect --format="{{index .Config.Labels \"com.docker.s
warm.task.id\"}}" <container>

$ docker inspect --format="{{index .Config.Labels \"com.docker.s
warm.task.id\"}}" 100bcb7eb051
to0xg7tp59bkshrrvgf4yhutp
```

Scale the app

You can scale the app by changing the `replicas` value in docker-compose.yml, saving the change, and re-running the docker stack deploy command:

```
$ docker stack deploy -c docker-compose.yml getstartedlab

Updating service getstartedlab_web (id: oauk2ifq9mfkg1k2kdr8cqgu
r)
```

Docker will do an `in-place update`, `no need to tear the stack down` first or kill any containers.

```
$ docker container ls -q

ea8167447298
100bcb7eb051
e12bb83371fd
b712972c07d5
```

Take down the app and the swarm

Take the app down with docker stack rm:

```
$ docker stack rm getstartedlab

Removing service getstartedlab_web
Removing network getstartedlab_webnet
```

This removes the app, but our one-node swarm is still up and running (as shown by `docker node ls`). Take down the swarm with `docker swarm leave --force`.

退出 swarm

- 节点退出 `docker swarm leave`
- swarm manager 必须使用 `--force` 强制退出

```
# 查看节点
$ docker node ls
ID                            HOSTNAME            STATUS
      AVAILABILITY        MANAGER STATUS
9fu8b0ot055ayazhcxu61jv33 *   S005_Ubuntu1604S    Ready
      Active              Leader

$ docker swarm leave

# swarm manager 必须使用 `--force` 强制退出
Error response from daemon: You are attempting to leave the swar
m on a node that is participating as a manager. Removing the las
t manager erases all current state of the swarm. Use `--force` t
o ignore this message.



# 退出
$ docker swarm leave --force
Node left the swarm.

$ docker node ls
Error response from daemon: This node is not a swarm manager. Us
e "docker swarm init" or "docker swarm join" to connect this nod
e to swarm and try again.
```

本节命令

```
docker stack ls                                            # Lis
t stacks or apps
docker stack deploy -c <composefile> <appname>  # Run the specif
ied Compose file
docker service ls                 # List running services associ
ated with an app
docker service ps <service>                  # List tasks associ
ated with an app
docker inspect <task or container>                  # Inspect t
ask or container
docker container ls -q                                      # Li
st container IDs
docker stack rm <appname>                             # Tear dow
n an application
```

swarms

Understanding Swarm clusters

A `swarm` is `a group of machines` that are running Docker and joined into a `cluster`. After that has happened, you **continue** to run the Docker commands you're used to, but now they are executed on a cluster **by a** **`swarm manager`**. The machines in a swarm can be `physical or virtual`. After joining a swarm, they are `referred to as nodes`.

`swarm` 就是 docker 集群模式。启用 swarm 之后，命令只能在 `swarm manager` 上执行。加入集群的机器可以是物理机或虚拟机，加入之后，被称为节点。

Swarm managers can use `several strategies` to run containers, such as

- `"emptiest node"` – which fills the least utilized machines with containers. 为使用率最低的机器分配容器
- Or `"global"`, which ensures that each machine gets exactly one instance of the specified container. 为每个机器都分配一个容器

You instruct the swarm manager to use these strategies in the `Compose file`, just like the one you have already been using.

Swarm managers are the `only machines` in a swarm that `can execute your commands`, or `authorize` other machines `to join` the swarm as `workers`. Workers are `just` there `to provide capacity` and `do not have the authority to tell` any other machine what it can and cannot do.

Up until now, you have been using Docker in a `single-host mode` on your local machine. But Docker also can be `switched into swarm mode`, and that's what enables the use of swarms. Enabling swarm mode `instantly 立即` makes the `current machine` a `swarm manager`. From then on, Docker will run the commands you execute on the swarm you're managing, rather than just on the current machine.

Set up your swarm

A swarm is `made up of multiple nodes`, which can be either `physical or virtual machines`. The basic concept is simple enough: run `docker swarm init` to enable swarm mode and make your `current` machine a `swarm manager`, then run `docker swarm join` on other machines to have them join the swarm as `workers`.

- `docker swarm init`：创建 swarm 且当前机器做为 `swarm manager (地主)`
- `docker swarm join`：加入已经创建好的 swarm 做 `worker (苦工)`

Create a cluster

原文中是使用 `docker-mechine` 创建了两个虚拟机。

由于我运行 `docker-ce` 的 `Ubuntu1604Server` 已经是虚拟机了，无法再在里面安装 `virtualbox`。因此，这里开了两个 `Ubuntu1604Server`，并且都安装了 `docker-ce`

节点名	节点角色	**IP** 地址	系统版本	**Docker** 版本
S12	Manager	192.168.56.212	Ubuntu 1604.03	Docker-ce 17.06
S13	Worker	192.168.56.213	Ubuntu 1604.03	Docker-ce 17.06

初始化 swarm

在 `S12` 上执行命令 `docker swarm init`。命令执行后，`S12` 便成为这个集群的 `swarm manager`，控制集群的所有命令都通过 `S12` 发出。

由于 `S12` 上有多个 IP，因此在初始化 swarm 的时候，必须使用 `--advertise-addr 192.168.56.212` 指定 swarm 绑定的 IP。

```
[user@S012 04.swarm_sample]$ docker swarm init --advertise-addr
192.168.56.212
Swarm initialized: current node (z2yzvrbh0mv2w2yzhoc9ryzmb) is n
ow a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4750ptov1hfil71va7ofrwvki
hvwupn4gs8akpz4hqis13y7u5-59w992ywlsrxme1glcr6axc9a 192.168.56.2
12:2377

To add a manager to this swarm, run 'docker swarm join-token man
ager' and follow the instructions.
```

初始化完成后，可以看到系统提示，当前主机已经是 `manager 地主` 了。

并且显示 `docker swarm join` 命令提示其他主机如何加入该集群。

```
To add a worker to this swarm, run the following command:

  docker swarm join \
  --token <token> \
  <ip>:<port>
```

在 `S13` 上执行 `docker swarm join` 命令加入刚才创建的 swarm。

```
user@S013:~$ docker swarm join --token SWMTKN-1-4750ptov1hfil71v
a7ofrwvkihvwupn4gs8akpz4hqis13y7u5-59w992ywlsrxme1glcr6axc9a 192
.168.56.212:2377

This node joined a swarm as a worker.
# 该节点已经成功加入 swarm ， 卖身为奴。
```

> 注意：在使用 `docker swarm join` 时，如果 `node` 也是多 IP 环境，也必须使用 `--advertise-addr ipaddr` 选项。否则 docker 会根据路由或者其他条件随机选择一个网卡，然而这个可能不是你所期待的。
>
> 参考《docker 99 问》多宿主网络怎么配置

现在回到 `S12`，使用 `docker node ls` 查看当前节点状态

```
$ docker node ls
ID                            HOSTNAME            STATUS
      AVAILABILITY         MANAGER STATUS
re0g5dw09cwd7h4ciab68uaw1     S013     Ready                Active

z2yzvrbh0mv2w2yzhoc9ryzmb *   S012     Ready                Active
               Leader
```

可以看到，S12 和 S13 都已经成为集群的节点了。并且，在 `MANAGER STATUS` 一栏，S12 被标为 `Leader`。

Deploy your app on a cluster

继续留在 S12 上，通过 swarm manager 的身份向集群发布应用。找到 `03.service` 节中的 `docker-compose.yml`。

使用 `docker stack deploy` 命令发布应用。

```
[user@S012 03.service_sample]$ docker stack deploy -c docker-com
pose.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_web
```

> 需要注意的是：由于 S13上面没有 `octowhale/friendlyhello:tag` 镜像，因此所有在 S13 节点上启动容器都失败了，提示 **No such image:...**。最后发现 5 个容器都启动在 S12 上了。
>
> 注意：这里其实是我将镜像名字写错了。本来应该是 `latest` 而写成了 `tag`。

```
[user@S012 03.service_sample]$ docker stack ps getstartedlab
ID                  NAME                          IMAGE
          NODE                DESIRED STATE       CURRENT STATE
            ERROR                                  PORTS
tqxgpkg81j3d        getstartedlab_web.1       octowhale/friendly
hello:tag   S013    Ready                Preparing 1 second ago

lylttxrahjsm         \_ getstartedlab_web.1   octowhale/friendly
hello:tag   S012    Shutdown             Rejected 1 second ago
  "No such image: octowhale/friend…"

... 略 ...
```

首先，在 S12 上执行命令 `docker stack rm` 关闭服务。

```
[user@S012 03.service_sample]$ docker stack rm getstartedlab
Removing service getstartedlab_web
Removing network getstartedlab_webnet
```

然后，切回到 `S13` ，使用 `docker pull` 命令拉取镜像 `octowhale/friendlyhello:lastest` 。

```
$ docker pull octowhale/friendlyhello:latest
latest: Pulling from octowhale/friendlyhello
ad74af05f5a2: Pull complete
a36a1c51ab4d: Pull complete
be169522399f: Pull complete
286703095347: Pull complete
1c1fda0fa4c6: Pull complete
0951c86a6675: Pull complete
0302b40f6cd6: Pull complete
```

最后，再次回到 `S12` 上发布应用

使用 `docker stack ps getstartedlab` 查看结果

```
[user@S012 03.service_sample]$ docker stack ps getstartedlab
ID                  NAME                  IMAGE
         NODE                DESIRED STATE       CURRENT STATE
         ERROR               PORTS
tcw0rpmg3jmh        getstartedlab_web.1   octowhale/friendlyhell
o:latest   S013    Running             Running 8 seconds ago

v2ww0v172uch        getstartedlab_web.2   octowhale/friendlyhell
o:latest   S013    Running             Running 8 seconds ago

yhlyjw4quyol        getstartedlab_web.3   octowhale/friendlyhell
o:latest   S012    Running             Running 9 seconds ago

5yyyq98y7ad4        getstartedlab_web.4   octowhale/friendlyhell
o:latest   S013    Running             Running 8 seconds ago

qpbonq3g97sv        getstartedlab_web.5   octowhale/friendlyhell
o:latest   S012    Running             Running 9 seconds ago
```

可以发现，容器已经在两台节点上分别启动起来了。

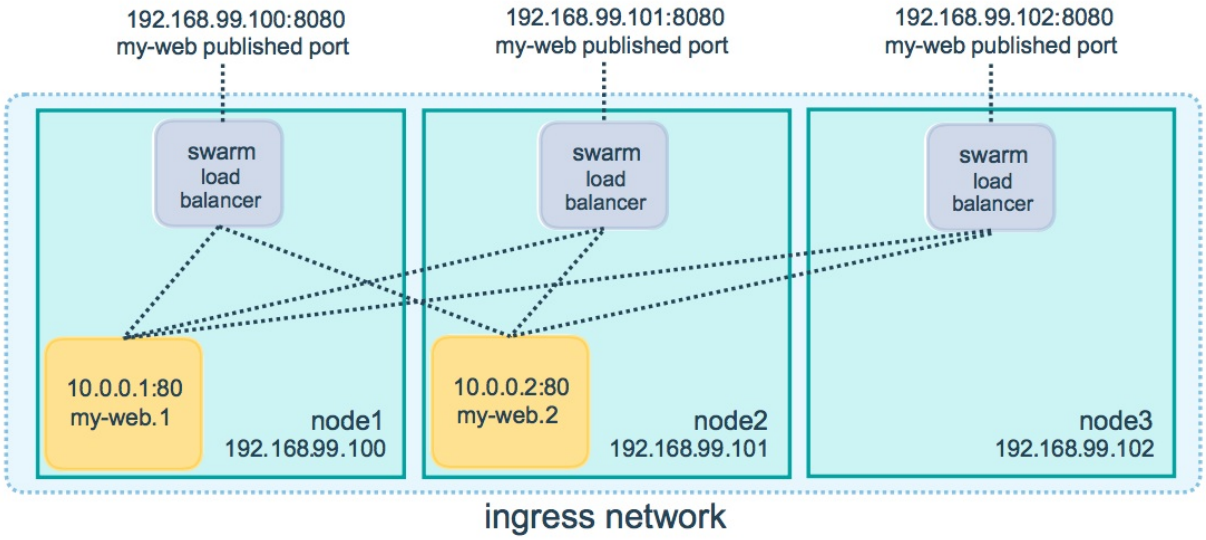
> 注意：第二次发布应用的时候，`S13` 依旧失败，提示 `"starting container failed: Ad…"`，经分析，应该是觉得 `docker-compose.yml` 发布是端口映射的原因 `- "80:80"`。由于用户普通用户，不能打开 `80` 端口。修改端口为 `8080` 后，重新发布正常了。

```
# 错误提示
j9hnjmfghnmb         \_ getstartedlab_web.2   octowhale/friendly
hello:latest   S013    Shutdown            Failed 31 seconds ago
    "starting container failed: Ad…"
```

Accessing your cluster

You can access your app from the IP address of **either** 同时 `S12` or `S13`. The network you created is `shared` between them and load-balancing. 共享网络和 **LB**

The reason both IP addresses work is that nodes in a swarm `participate in an ingress routing mesh 共享路由入口`. This ensures that a service deployed at a certain port within your swarm always has that port reserved to itself, no matter what node is actually running the container. Here's a diagram of how a routing mesh for a service called my-web published at port 8080 on a three-node swarm would look:



Having connectivity trouble?

Keep in mind that in order to use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode: 如果要实现 共享路由入口 **ingress-routing-mesh**，需要在 节点 直接相互允许以下端口访问：

```
Port 7946 TCP/UDP for container network discovery.
Port 4789 UDP for the container ingress network.
```

Iterating and scaling your app

From here you can do everything you learned about in part 3.

Scale the app by changing the `docker-compose.yml` file.

Change the app behavior by editing code.

In either case, simply run `docker stack deploy` again to deploy these changes.

You can join any machine, physical or virtual, to this swarm, using the same `docker swarm join` command you used on `S13`, and capacity will be added to your cluster. Just run `docker stack deploy` afterwards, and your app will take advantage of the new resources.

Cleanup

You can tear down the stack with docker stack rm. For example:

```
# 在 S12 上執行 rm 刪除 service
docker stack rm getstartedlab
```

stack

> https://docs.docker.com/get-started/part5/

Introduction

A `stack` is a group of `interrelated(相关的) services` that **share** dependencies, and can be **orchestrated(编排)** and **scaled** together. A single stack is capable of `defining(定义) and coordinating(协调)` the functionality of an entire application (though very complex applications may want to use multiple stacks).

> 所以，stack 就是，不同任务的调度器。

发布 `stack` 使用命令 `docker stack deploy`

Add a new service and redeploy

和 上一节 一样，我们这里需要两个节点 `S12` 和 `S13`.

节点名	节点角色	IP 地址	系统版本	Docker 版本
S12	Manager	192.168.56.212	Ubuntu 1604.03	Docker-ce 17.06
S12	Worker	192.168.56.213	Ubuntu 1604.03	Docker-ce 17.06

上一节中，最后已经删除了 `stack`, 退出了 `swarm` 。

创建集群

注意执行命令的机器

```
# 创建集群
[user@S012 05.stack_sample]$ docker swarm init --advertise-addr
192.168.56.212

# S13 加入集群
```

```
[user@S013 ~]$ docker swarm join --token SWMTKN-1-3096uq30z7u8x5
qt5uy7zfhevd5uet5os5s7a84a5b908clxh2-dattjfqr3pw9sjcsqvfiyk4ri 1
92.168.56.212:2377

# 查看节点信息
[user@S012 05.stack_sample]$ docker node ls
ID                            HOSTNAME            STATUS
      AVAILABILITY        MANAGER STATUS
69im9oe8rljfjbh600dnyulks *   S012    Ready               Active
              Leader
nqhgz9accdqorokp6ct47c262     S013    Ready               Active


# 创建 stack 服务
[user@S012 05.stack_sample]$ docker stack deploy -c docker-compo
se.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_web

# 查看 stack 信息
[user@S012 05.stack_sample]$ docker stack  ps getstartedlab
ID                  NAME                  IMAGE
         NODE                DESIRED STATE       CURRENT STATE
         ERROR               PORTS
jco8fikcn0zf        getstartedlab_web.1   octowhale/friendlyhell
o:latest   S013    Running             Running 2 minutes ago

x0w43zxui0oa        getstartedlab_web.2   octowhale/friendlyhell
o:latest   S012    Running             Running 2 minutes ago

ofbupicgj1af        getstartedlab_web.3   octowhale/friendlyhell
o:latest   S013    Running             Running 2 minutes ago

30a16pv6ov4k        getstartedlab_web.4   octowhale/friendlyhell
o:latest   S013    Running             Running 2 minutes ago

iv33enuysiu2        getstartedlab_web.5   octowhale/friendlyhell
o:latest   S012    Running             Running 2 minutes ago
```

更新 docker-compose 文件

```
# docker-compose-visualizer.yml
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
networks:
  webnet:
```

The only thing **new** here is the peer service to `web`, named `visualizer`. You'll see two new things here:

- a `volumes` key, giving the visualizer access to the host's socket file for Docker,

- and a `placement` key, ensuring that this service **only ever runs on a swarm manager** – never a worker. That's because this container, built from an open source project created by Docker, displays Docker services running on a swarm in a diagram.

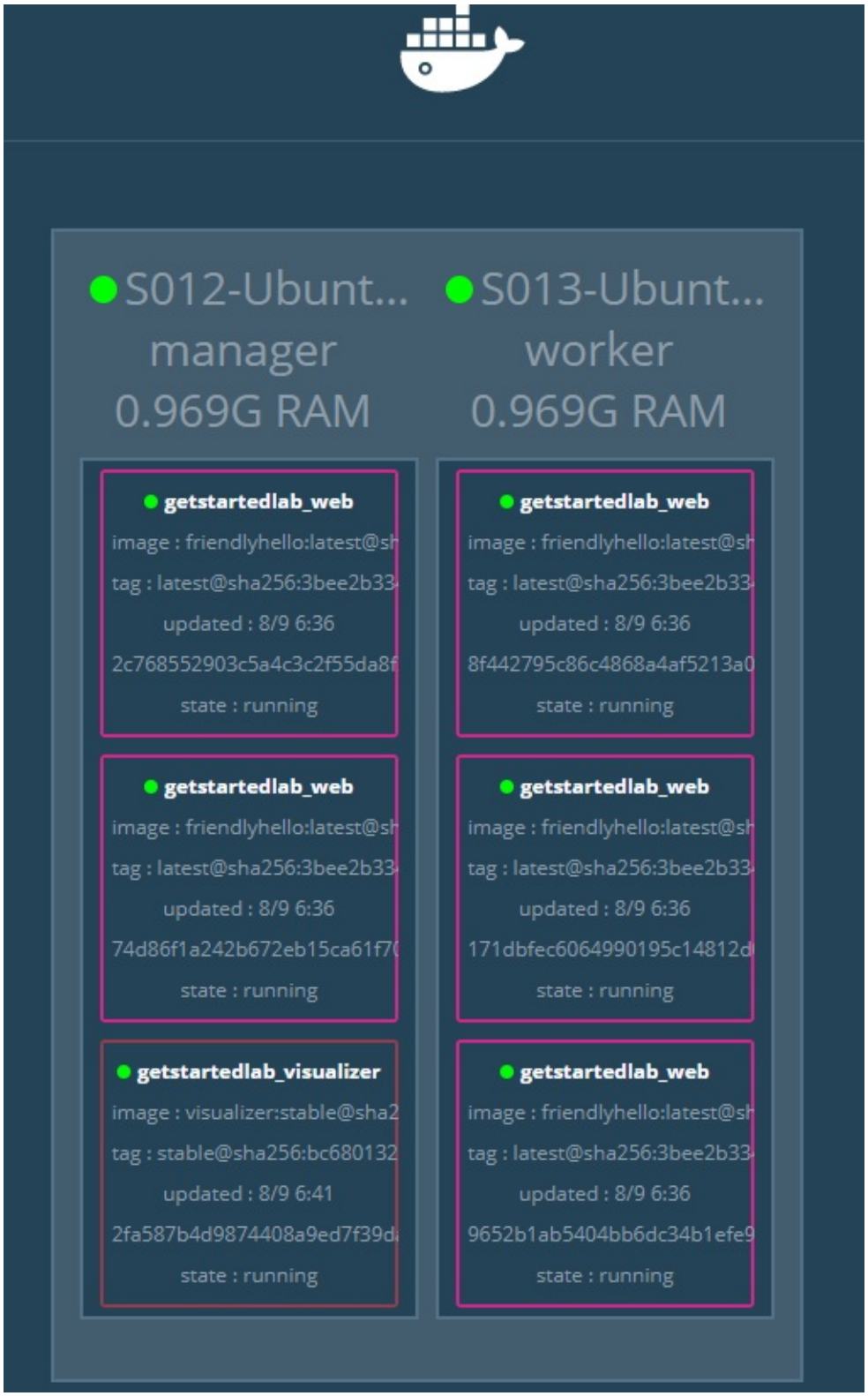
重新执行 `docker stack deploy` 更新 **getstartedlab** 服务。

```
# 更新 getstartedlab
[user@S012 05.stack_sample]$ docker stack deploy -c docker-compo
se-visualizer.yml getstartedlab
Updating service getstartedlab_web (id: 7m4i1ily612f8mk9jh7pt7jp
v)
Creating service getstartedlab_visualizer

# 查看节点信息
[user@S012 05.stack_sample]$ docker stack ps getstartedlab
ID                  NAME                            IMAGE
                   NODE                DESIRED STATE       CURRE
NT STATE            ERROR                PORTS
7p8pt9dqp2ww        getstartedlab_visualizer.1   dockersamples/v
isualizer:stable   S012    Running             Running 16 second
s ago
jco8fikcn0zf        getstartedlab_web.1          octowhale/frien
dlyhello:latest      S013    Running             Running 5 minut
es ago
x0w43zxui0oa        getstartedlab_web.2          octowhale/frien
dlyhello:latest      S012    Running             Running 5 minut
es ago
ofbupicgj1af        getstartedlab_web.3          octowhale/frien
dlyhello:latest      S013    Running             Running 5 minut
es ago
30a16pv6ov4k        getstartedlab_web.4          octowhale/frien
dlyhello:latest      S013    Running             Running 5 minut
es ago
iv33enuysiu2        getstartedlab_web.5          octowhale/frien
dlyhello:latest      S012    Running             Running 5 minut
es ago
```

可以看到 `getstartedlab_visualizer.1` 只启动了一个，并且运行在 `S12` 上。 The visualizer is a standalone service that can run in any app that includes it in the stack. It doesn't depend on anything else.

通过 `http://192.168.56.212:8080/` 或 `http://192.168.56.213:8080/` 都可以看到 `visualizer` 界面。因为 `swarm` 共享了入口(`ingress routing mesh`)。



Persist the data

为容器添加持久化数据 `redis` 。

修改 `docker-compose.yml` 增加 `redis` 部分

```
# docker-compose-visualizer-redis.yml
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
  redis:
    image: redis
    ports:
      - "6379:6379"
    volumes:
```

```
      - ./data:/data
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
networks:
  webnet:
```

here in our Compose file we expose it from the host to the world, so you can `actually enter the IP` for any of your nodes into Redis Desktop Manager and manage this Redis instance, if you so choose.

`Most importantly`, there are a couple of things in the redis specification that `make data persist` between deployments of this stack:

- redis always `runs on the manager`, so it's always using the `same filesystem`.
- redis accesses an **arbitrary(任意)** directory in the host's file system `as /data inside` the container, which is where Redis stores data.

Together, this is creating a "source of truth" in your host's physical filesystem for the Redis data. `Without this`, Redis would store its data in `/data` inside the container's filesystem, which would `get wiped out` if that container were ever redeployed.

This source of truth has two components:

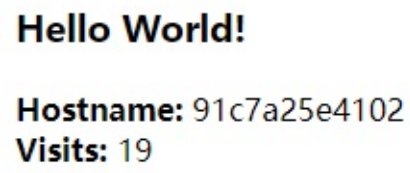
- The placement constraint you put on the Redis service, ensuring that it `always uses the same host`.
- The volume you created that lets the container access `./data (on the host)` as `/data (inside the Redis container)`. While containers come and go, the files stored on ./data on the specified host will persist, enabling continuity.**(**如果不映射本地路径到 **redis** 容器 **/data** 中，那么当容器关闭后，**redis** 数据就会消失，无法达到数据持久化的目的**)**

更新 **getstartedlab** 服务

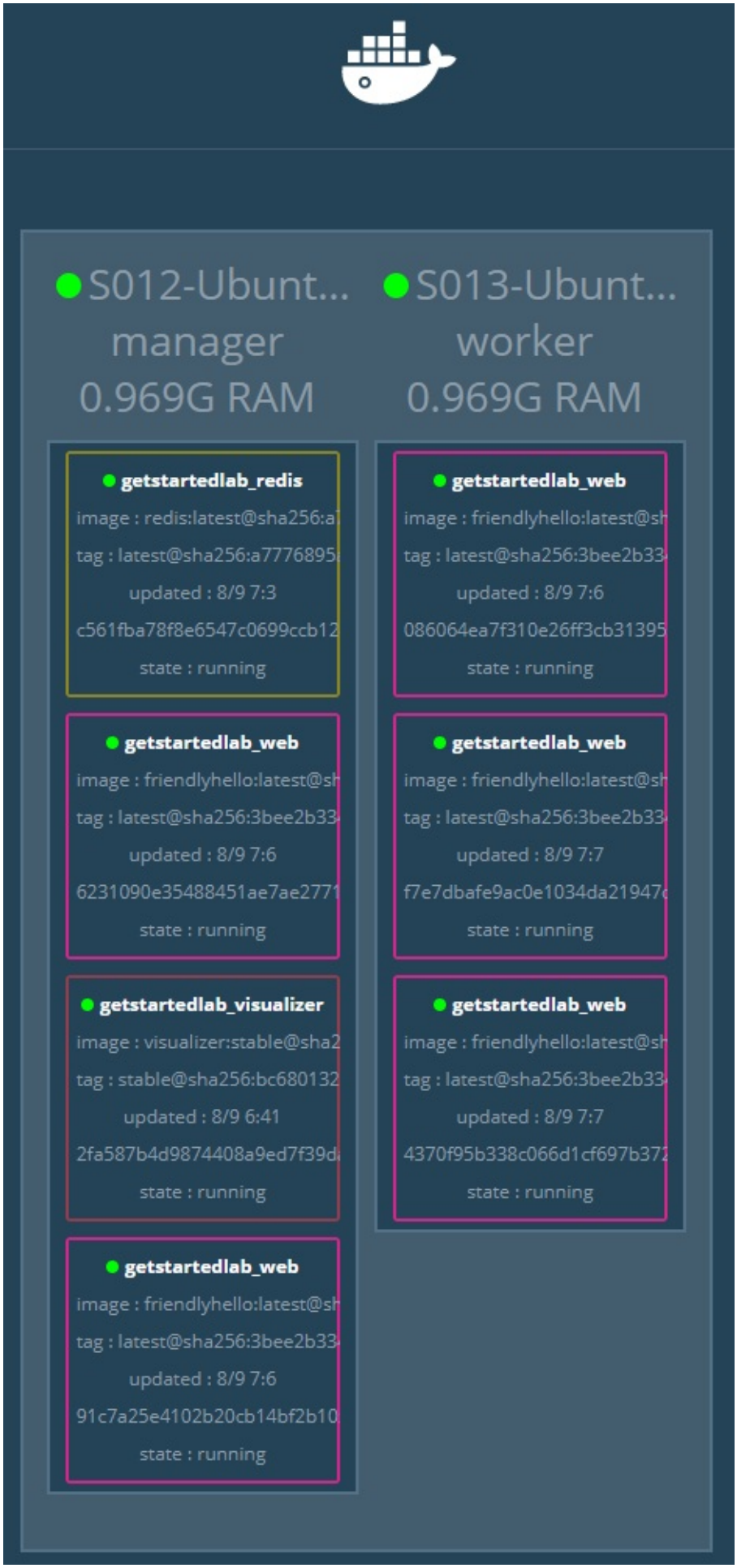
```
# 创建 redis 数据目录
[user@S012 05.stack_sample]$ mkdir -p ./data

# 更新服务
[user@S012 05.stack_sample]$ docker stack deploy -c docker-compo
se-visualizer-redis.yml getstartedlab
Updating service getstartedlab_web (id: 7m4i1ily612f8mk9jh7pt7jp
v)
Updating service getstartedlab_visualizer (id: gjxq5cczcypoeoad3
e41befzy)
Creating service getstartedlab_redis
```

访问 `http://192.168.56.212:8800/` 就可以看到页面已经支持计数器了。



Also, check the visualizer at port 8080 on either node's IP address, and you'll see the `redis` service running along with the `web` and `visualizer` services.


S012-Ubunt...
manager
0.969G RAM
S013-Ubunt...
worker
0.969G RAM
getstartedlab_redis
image : redis:latest@sha256:a
tag : latest@sha256:a7776895
updated : 8/9 7:3
c561fba78f8e6547c0699ccb12
state : running
getstartedlab_web
image : friendlyhello:latest@sh
tag : latest@sha256:3bee2b33
updated : 8/9 7:6
086064ea7f310e26ff3cb31395
state : running
getstartedlab_web
image : friendlyhello:latest@sh
tag : latest@sha256:3bee2b33
updated : 8/9 7:6
6231090e35488451ae7ae2771
state : running
getstartedlab_web
image : friendlyhello:latest@sh
tag : latest@sha256:3bee2b33
updated : 8/9 7:7
f7e7dbafe9ac0e1034da21947
state : running
getstartedlab_visualizer
image : visualizer:stable@sha2
tag : stable@sha256:bc680132
updated : 8/9 6:41
2fa587b4d9874408a9ed7f39d
state : running
getstartedlab_web
image : friendlyhello:latest@sh
tag : latest@sha256:3bee2b33
updated : 8/9 7:7
4370f95b338c066d1cf697b372
state : running
getstartedlab_web
image : friendlyhello:latest@sh
tag : latest@sha256:3bee2b33
updated : 8/9 7:6
91c7a25e4102b20cb14bf2b10
state : running

Get Started, Part 6: Deploy your app

这一篇真的没有什么好写的。都是引导性的内容。

更主要问题在于我也没用过。

还是直接看官网文档的翻译吧。

祝贺您！

您已完成对整个 Docker 平台的完整技术栈、开发到部署浏览。

Docker 平台的内容远不止此处涵盖的内容，但您已了解容器、镜像、服务、swarm、技术栈、扩展、负载均衡、存储卷和布局约束的基本内容。

想要深入了解更多内容？以下是我们推荐的一些资源：

样板：我们的样板包含容器中运行的常用软件的多个示例，以及一些提供最佳实践的优秀实验室。 用户指南：用户指南中提供了一些示例，它们更深入地说明了网络和存储。 管理指南：涵盖如何管理 Docker 化生产环境。 训练：官方 Docker 课程，提供了现场指导和虚拟教室环境。 博客：涵盖 Docker 的最新内容。

Network container

Launch a container on the default network

Docker includes support for networking containers through the use of **network drivers**. By default, Docker provides two network drivers for you, the `bridge` (default) and the `overlay` drivers. You can also write a network driver plugin so that you can create your own drivers but that is an advanced task.

使用 `docker network ls` 查看当前网络

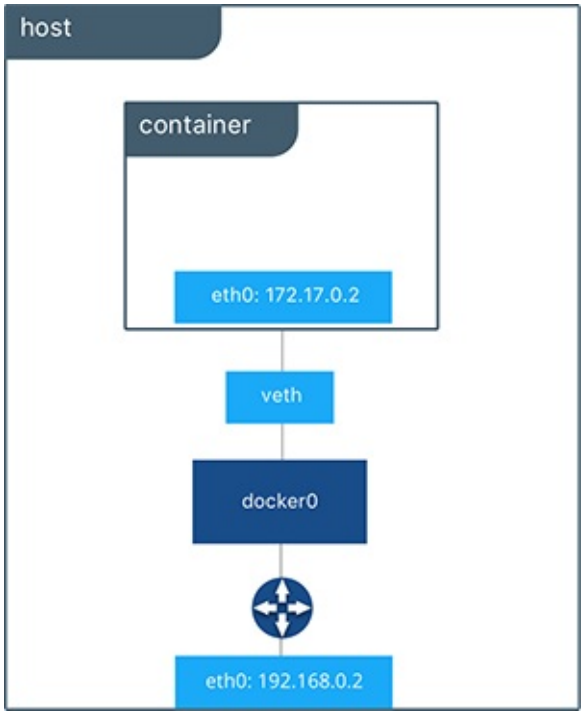
```
$ docker network ls

NETWORK ID          NAME                DRIVER
18a2866682b8        none                null
c288470c46f6        host                host
7b369448dccb        bridge              bridge
```

The network named `bridge` is a special network, 当不指定 `--net` 的时候，默认使用该网络.

```
$ docker run -itd --name=networktest ubuntu

74695c9cea6d9810718fddadc01a727a5dd3ce6a69d09752239736
```



使用命令 `docker network inspect <network_name>` 查看网络信息

```
$ docker network inspect bridge
```

Create your own bridge network

- Docker Engine natively supports `both` `bridge` networks and `overlay` networks.
- A `bridge` network is `limited to a single host` running Docker Engine.
- An `overlay` network can `include multiple hosts` and is a more advanced topic.

创建一个 **bridge** 网络

```
# 创建一个网络
$ docker network create -d bridge my_bridge
bca088e559338498829496538f1e58b414cfe22be0737bf7b6d9c36bfd2954a5
```

The `-d` flag tells Docker to `use` the `bridge driver` for the new network. You could have left this flag off as bridge is the `default` value for this flag.

查看当前所有网络

使用命令 `docker network ls`

```
$ docker network ls

NETWORK ID          NAME                DRIVER
7b369448dccb        bridge              bridge
615d565d498c        my_bridge           bridge
18a2866682b8        none                null
c288470c46f6        host                host
```

查看网络详细信息

使用命令 `docker network inspect <network_name>`

```
# 查看网络信息
$ docker network inspect my_bridge
[
    {
        "Name": "my_bridge",
        "Id": "bca088e559338498829496538f1e58b414cfe22be0737bf7b
6d9c36bfd2954a5",
        "Created": "2017-09-08T17:18:10.67709847+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.20.0.0/16",
                    "Gateway": "172.20.0.1"
                }
            ]
        },
```

```
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "c1f13c8b73e75a3e424bb55bc98158348090a8121404541c371
f3201a28ab113": {
                "Name": "alpine_test",
                "EndpointID": "61eb93407ef612356ca566d5b2e7ce2f2
46890e8551781e35f5246f88f7cb51b",
                "MacAddress": "02:42:ac:14:00:02",
                "IPv4Address": "172.20.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Add containers to a network

Networks, by definition, provide `complete isolation(完全隔离)` for containers.

创建容器时，使用 `--net=<network_name>` flag 为容器指定网络

```
# 将一个容器加入网络
$ docker run --rm -itd --net=my_bridge --name alpine_test alpine

c1f13c8b73e75a3e424bb55bc98158348090a8121404541c371f3201a28ab113
```

查看容器网络信息

使用命令 `docker inspect --format='{{json .NetworkSettings.Networks}}' <container_name>`

```
# 通过容器查看网络信息
$ docker inspect --format='{{json .NetworkSettings.Networks}}' a
lpine_test
{"my_bridge":{"IPAMConfig":null,"Links":null,"Aliases":["c1f13c8
b73e7"],"NetworkID":"bca088e5593384988294965 38f1e58b414cfe22be07
37bf7b6d9c36bfd2954a5","EndpointID":"61eb93407ef612356ca566d5b2e
7ce2f246890e8551781e35f5246f88f7cb51b","Gateway":"172.20.0.1","I
PAddress":"172.20.0.2","IPPrefixLen":16,"IPv6Gateway":"","Global
IPv6Address":"","GlobalIPv6PrefixLen":0,"MacAddress":"02:42:ac:1
4:00:02","DriverOpts":null}}
```

将容器移除网络

You can remove a `container` from a `network` by disconnecting the container. To do this, you supply `both` the `network name` and the `container name`. You can also use the `container ID`. In this example, though, the name is faster.

将一个 `container` 从给一个网络中移除时，使用命令 `docker network disconnect <network_name> <container_name/container_id>`

```
# 将容器移除网络
$ docker network disconnect my_bridge alpine_test

# 重新通过容器查看网络信息
$ docker inspect --format='{{json .NetworkSettings.Networks}}' a
lpine_test
{}

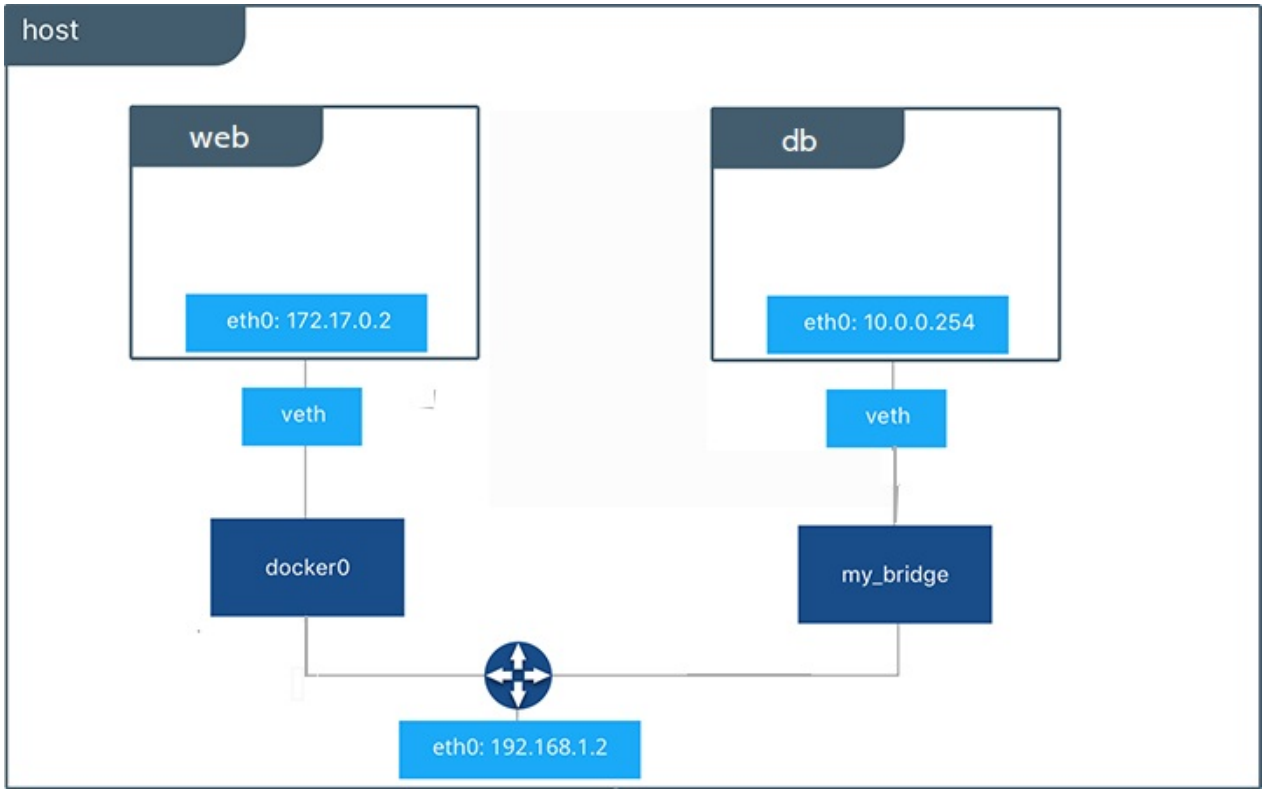
# 查看容器状态
# 容器虽然脱离网络，但是依然存活
$ docker container ps -a
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS                          PORTS
        NAMES
c1f13c8b73e7        alpine              "/bin/sh"
 5 minutes ago       Up 5 minutes
        alpine_test
```

容器的网络隔离

创建两个容器，

- 容器 `db` 使用 `my_bridge` 网络
- 容器 `web` 使用默认的 `bridge` 网络

```
# 创建容器
$ docker run -d --net=my_bridge --name db training/postgres
53365bd018e5525703ac8230717c1e5df6fa020f72e4f90b45f9d87dc6e7d725
$ docker run -itd --name web ubuntu
774887a1a4d4cd5b254f6797dadebd0388b6707f1f051435ec881251bf273320
```



进入容器 `db` ，尝试 ping 容器 `web` 。以失败而告终

```
# 查看IP地址
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{
.IPAddress}}{{end}}' db
172.20.0.2
$ docker inspect --format='{{range .NetworkSettings.Networks}}{{
.IPAddress}}{{end}}' web
172.17.0.2

$ docker exec -it db bash
root@53365bd018e5:/# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
^C
--- 172.17.0.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1007ms
```

将容器加入网络

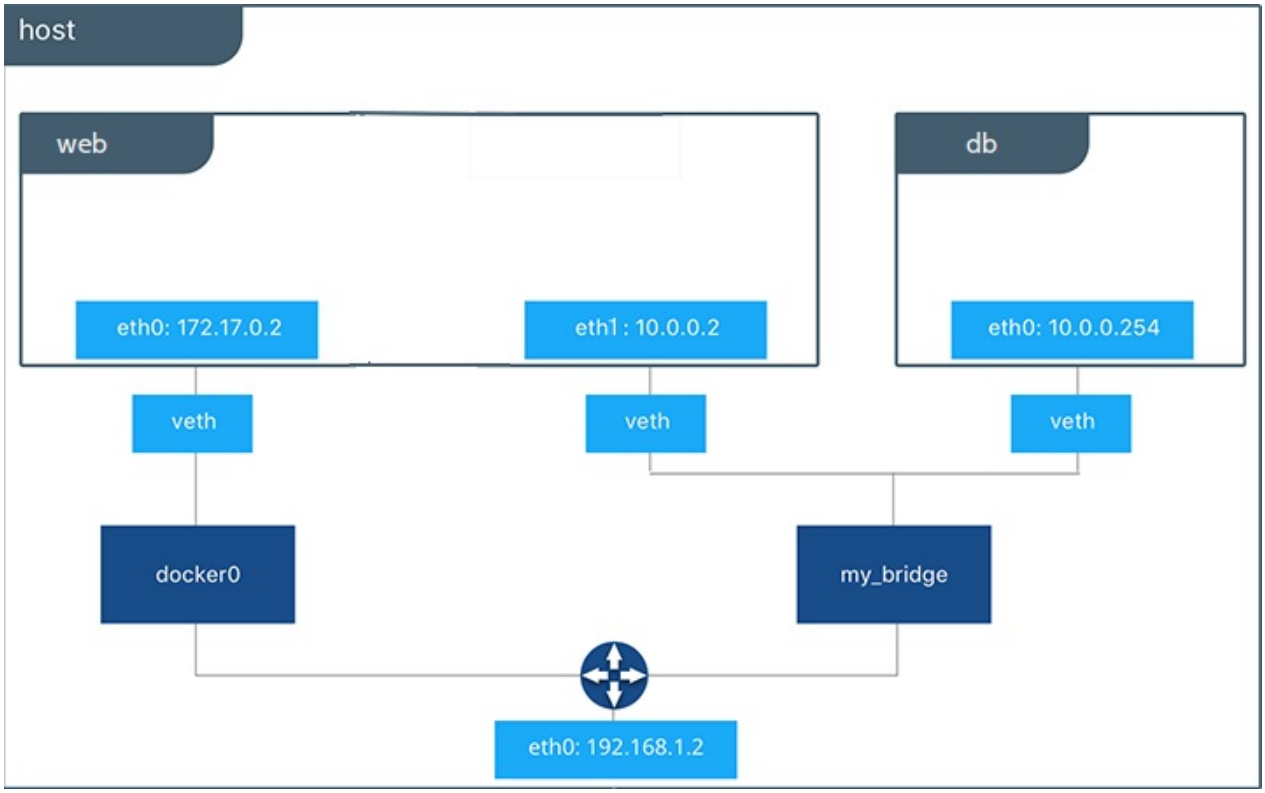
使用命令 `docker network connect <network_name> <container_name>`

```
$ docker network connect my_bridge web

$ docker inspect --format='{{range .NetworkSettings.Networks}}{{
.IPAddress}}{{end}}' web
172.17.0.2
172.20.0.3
```

重新 ping `web` 的新地址

```
$ docker exec -it db bash
root@53365bd018e5:/# ping 172.20.0.3
PING 172.20.0.3 (172.20.0.3) 56(84) bytes of data.
64 bytes from 172.20.0.3: icmp_seq=1 ttl=64 time=0.129 ms
64 bytes from 172.20.0.3: icmp_seq=2 ttl=64 time=0.087 ms
64 bytes from 172.20.0.3: icmp_seq=3 ttl=64 time=0.076 ms
^C
--- 172.20.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.076/0.097/0.129/0.024 ms
```



删除网路

使用命令 `docker network rm <network_name>`

```
$ docker network rm my_bridge
my_bridge

$ docker network ls
NETWORK ID          NAME                              DRIVER
       SCOPE
b2807e40afc6        bridge                            bridge
       local
1f069296a405        docker_gwbridge                   bridge
       local
dfea931891b4        dockerregistrymirror_default      bridge
       local
3c2b536039ac        host                              host
       local
ffa7c88de9e4        none                              null
       local
```

Best practices for writing Dockerfiles

dockerfile 是一个包含特殊命令格式的文件，描述了如何 `build` 一个镜像。

This document covers the best practices and methods recommended by Docker, Inc. and the Docker community for creating `easy-to-use` , `effective` Dockerfiles. We `strongly suggest` you `follow` these recommendations (**in fact, if you're creating an Official Image, you must adhere to these practices**).

> Note: for more detailed explanations of any of the Dockerfile commands mentioned here, visit the Dockerfile Reference page.

General guidelines and recommendations

Containers should be ephemeral(无状态容器)

`dockerfile` 创建的容器应该是 `无状态` 的，容器可以被关闭销毁，并且可以被新建的容器替代。参考 Processes

使用 .dockerignore

为了提高性能，可以使用 `.dockerignore` 忽略编译时不必要的文件。用法与 `.gitignore` 相似。 参考.dockerignore

避免安装不必要的包

降低复杂性、依赖以、包大小以及 `build` 时间。

Each container should have only one concern (容器功能单一性)

将多个应用放入多个容器中，可以更好的横向扩展和服用容器。

`one process per container` 是个不错的宗旨，但并不是一个 `快捷的方法` 。应该让容器尽可能的保持 `clean` 与 `模块化(modular)`

You may have heard that there should be “ `one process per container` ”. While this mantra has good intentions, it is `not necessarily true` that there should be only one operating system process per container. In addition to the fact that containers can now be spawned with an init process, some programs might spawn additional processes of their own accord.

减少容器层数

保持 `dockerfile` 的可读性与 `容器层数` 之间的平衡。

参数排序换行

使用 `\` 换行后，可以方便的修改、去重、增加可读性。

```
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion
```

善用 **build** 缓存

docker build 的时候按照 `dockerfile` 中指令顺序执行。每个指令 `执行之前` ，docker 检查该指令是否存在缓存镜像，以及是否可以 `复用` ，而不是重新创建一个重复镜像。If you do not want to use the cache at all you can use the `--no-cache=true` option on the docker build command.

复用镜像缓存的基本原则：

- `Starting with a parent image that is already in the cache` , the next instruction is compared against `all child images` derived(自..衍生) from that base image to see if one of them was built using the exact same instruction. If not, the cache is invalidated

- In most cases simply comparing the instruction in the `Dockerfile` with one of the child images is sufficient. However, certain instructions require a little more examination and explanation.
- For the `ADD` and `COPY` instructions, `the contents of the file(s) in the image` are examined and a `checksum` is calculated `for each file`. The `last-modified and last-accessed` times of the file(s) are `not` considered in these checksums. During the cache lookup, the checksum is compared against the checksum in the existing images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated.
- Aside from the `ADD` and `COPY` commands, cache checking `will not` look at the files in the container to `determine a cache match`. 缓存一致性检查不会涉及容器中的缓存文件。

容器指令

FROM

Dockerfile reference for the FROM instruction

指定父镜像。任何时候，都尽量选择官方镜像。推荐使用 `Debian images`

LABEL

Understanding object labels

在项目中，使用 `LABEL` 标记 `镜像` 标签是一个个 `键值对`

> **Note**: 如果值有空格，使用 `双引号 (")` 括起开。尽量避免 `值` 里面本身就包含双引号，

三种有效格式

```
# Set one or more individual labels
LABEL com.example.version="0.0.1-beta"
LABEL vendor="ACME Incorporated"
LABEL com.example.release-date="2015-02-12"
LABEL com.example.version.is-production=""

# Set multiple labels on one line
LABEL com.example.version="0.0.1-beta" com.example.release-date=
"2015-02-12"

# Set multiple labels at once, using line-continuation character
s to break long lines
LABEL vendor=ACME\ Incorporated \
      com.example.is-beta= \
      com.example.is-production="" \
      com.example.version="0.0.1-beta" \
      com.example.release-date="2015-02-12"
```

RUN

Dockerfile reference for the RUN instruction

记住，写 Dockerfile 的基本规则

- 可读性高
- 容易理解
- 方便维护
- 将 长而复杂的 RUN 语句 使用 换行符 (\) 分割成多行。

APT-GET

避免使用 `RUN apt-get upgrade` or `dist-upgrade` 。而应该指定特定的包，例如 `apt-get install -y foo` 。

Always `combine(联合)` `RUN apt-get update` with `apt-get install` in the same RUN statement, for example:

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo
```

Using `apt-get update alone` in a `RUN` statement causes `caching issues` and subsequent `apt-get install` instructions `fail`.(由 build cache 引起的)。Using `RUN apt-get update && apt-get install -y` ensures your Dockerfile installs the `latest package versions` with no further coding or manual intervention.

Below is a `well-formed` `RUN` instruction that demonstrates all the `apt-get recommendations`.

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
 && rm -rf /var/lib/apt/lists/*
```

USING PIPES

部分 `RUN` 命令支持 `管道 (|)`。

例如：

```
RUN wget -O - https://some.site | wc -l > /number
```

上诉命令使用 `/bin/sh -c` 解释器，`最后一个`(这里为 `wc -l`) `退出值` 是否成功决定了此次 `RUN` 命令是否成功。此案例中，即使 `wget` 失败了，`RUN` 也是成功的。

使用 `set -o pipefail &&` 可以保证管道失败的时候，`RUN` 也失败。

```
RUN set -o pipefail && wget -O - https://some.site | wc -l > /nu
mber
```

> 注意：`不是所有` shell 都支持 `-o pipefail` 。
>
> > `dash shell` 就不支持支持。因此可以考虑 `exec` 格式的 `RUN` 。

```
RUN ["/bin/bash", "-c", "set -o pipefail && wget -O - https://so
me.site | wc -l > /number"]
```

CMD

Dockerfile reference for the CMD instruction

- `CMD` 命令只应该出现在 `dockerfile` 中一次。
  - 如果重复出现，只有最后一个会生效
- `CMD` 三格式：
  - `exec` form：`CMD ["executable","param1","param2"]` ，推荐使用
    - 传递 `json` 数组。必须使用 `双引号`
  - 作为 `ENTRYPOINT` 默认参数：`CMD ["param1","param2"]`
    - 参考 《docker 99 问》 CMD 和 ENTRYPOINT 到底有什么不同
  - `shell` form：`CMD command param1 param2`
    - 默认使用 `/bin/sh -c` 解释器
- `CMD` 命令会在容器内部执行。
- `CMD` 命令应该始终使用如下格式
  - `CMD ["executable", "param1", "param2"…]`
  - ex: `CMD ["apache2","-DFOREGROUND"]`
- `CMD` 命令大多数情况下应该指定一个 `可交互` 的 `shell` ，例如 `bash` ，`python` 等
  - ex: `CMD ["perl", "-de0"]` ，`CMD ["python"]`
  - 这样，创建容器时可以使用 `docker run -it python`

- 尽量不要使用 `CMD ["param1","param2"]` 与 `ENTRYPOINT` 组合格式，除非你清楚的知道你在做什么。

EXPOSE

Dockerfile reference for the EXPOSE instruction

指定容器提供服务的端口

```
EXPOSE <port> [<port>...]
```

- 使用：`docker run -p out_port:inter_port` 实现映射
- For container linking, Docker provides environment variables for the path from the recipient container back to the source (ie, `MYSQL_PORT_3306_TCP`).

延展阅读 EXPOSE 与 PUBLISH 的区别

ENV

Dockerfile reference for the ENV instruction

- 设置容器的 `环境变量`
  - For example, `ENV PATH /usr/local/nginx/bin:$PATH` will ensure that CMD ["nginx"] just works.
- 定义 `dockerfile` 变量

```
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
```

AND or COPY

Dockerfile reference for the ADD instruction

Dockerfile reference for the COPY instruction

- `ADD` 和 `COPY` 功能相似，但推荐使用 `COPY`
- `COPY` 只支持将文件从 `本地` 复制到 `镜像`
- `ADD` 支持：
  - URL 支持：`ADD http://example.com/big.tar.xz /usr/src/things/`
  - 解压支持：`ADD rootfs.tar.xz /`
- 使用多个 `COPY` 命令将文件分别复制到 `镜像` 中能更好的使用 `build cache` 机制
- 在文件不需要被解压时，使用 `COPY` 而不是 `ADD`
- 出于 `镜像容量` 考虑，应该使用 `curl / wget` 代替 `ADD` 的 `URL` 支持
  - 删除多余的文件
  - 减少镜像层数

```
# ADD 命令
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all

# 使用 curl / wget 代替
RUN mkdir -p /usr/src/things \
    && curl -SL http://example.com/big.tar.xz | tar -xJC /usr/src/things \
    && make -C /usr/src/things all
```

ENTRYPOINT

Dockerfile reference for the ENTRYPOINT instruction

指定容器 `默认入口`

- `ENTRYPOINT` 与 `CMD`：`ENTRYPOINT CMD`

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]

# 执行效果相当于 s3cmd --help
```

- 优化 `ENTRYPOINT` 入口

For example, the Postgres Official Image uses the following script as its `ENTRYPOINT`:

```
#!/bin/bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

> 注意： Note: This script uses the `exec` Bash command so that the final running application becomes the `container's PID 1`. This `allows` the application to `receive` `any Unix signals` sent to the container. See the `ENTRYPOINT` help for more details.
>
> 注意2： `gosu` 用于替代 `sudo` 切换用户权限。更多信息参考 `USER` 命令新入口

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
```

VOLUME

Dockerfile reference for the VOLUME instruction

- The `VOLUME` instruction creates `a mount point` with the specified name and marks it as `holding externally mounted` volumes from native host or other containers.

- `VOLUEME` 指令创建了一个 `挂载点` ，该挂载点可以挂在来自于 `本地目录` 或 `其他容器` 的卷。
  - 就像 `EXPOSE` 指令暴露了一个可以被映射的端口一样。
- 需要将容器可变数据放入到挂载目录中
  - 数据盘
  - 日志盘
  - ...

USER

Dockerfile reference for the USER instruction

- 如果容器服务可以运行与非特权用户，可以使用 `USER` 切换用户

```
USER <user>[:<group>] or
USER <UID>[:<GID>]
```

创建用户

```
RUN groupadd -r postgres && useradd --no-log-init -r -g postgres
 postgres
```

> 注意：上述命令在添加用户时将的到一个 `非固定` 的 `UID/GID` 。如果有需要，可以使用命令指定 `UID/GID` .
>
> 注意2： Due to an unresolved bug in the Go archive/tar package's handling of sparse files, attempting to create a user with a sufficiently `large UID` inside a Docker container can lead to disk exhaustion as `/var/log/faillog` in the container layer is `filled with NUL (\0)` characters. Passing the `--no-log-init flag` to useradd works around this issue. The `Debian/Ubuntu` `adduser` wrapper `does not` support the `--no-log-init` flag and should be avoided.

- 在容器中，使用 `gosu` 代替 `sudo`
- 不要 `频繁` 的在 `dockerfile` 中切换用户

WORKDIR

Dockerfile reference for the WORKDIR instruction

```
WORKDIR /path/to/workdir
```

- `WORKDIR` 的参数应该使用 `绝对路径`
- 使用 `WORKDIR` 而不是 `RUN cd /some/path && do something`

ONBUILD

Dockerfile reference for the ONBUILD instruction

在 `镜像上` 操作。

不是很懂。

Examples for Official Repositories

These Official Repositories have exemplary Dockerfiles:

- Go
- Perl
- Hy
- Ruby

创建基础镜像

- 大多数时候，只需要使用 `FROM` 抓取官方镜像进行修改
- 在某些时候，可能需要自己创建镜像

使用 **tar** 创建完整镜像

打包压缩当前系统，创建完整初始镜像。

以 `ubuntu` 为例

```
$ sudo debootstrap xenial xenial > /dev/null
$ sudo tar -C xenial -c . | docker import - xenial

a29c15f1bf7a

$ docker run xenial cat /etc/lsb-release

DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04 LTS"
```

There are more example scripts for creating parent images in the Docker GitHub Repo:

- BusyBox
- CentOS / Scientific Linux CERN (SLC) on Debian/Ubuntu or on CentOS/RHEL/SLC/etc.
- Debian / Ubuntu

使用 **scratch** 创建一个简单父镜像

`scratch` 是 docker 维护的一个小型镜像。

没多大意义

其他

更多创建 `基本镜像` 或者 `黑箱镜像` 可以参考

- 使用 `import` 创建镜像
- 使用 `commit` 创建镜像

多阶构建 multi-stage-build

> https://docs.docker.com/engine/userguide/eng-image/multistage-build/#use-multi-stage-builds
>
> 推荐链接：http://blog.alexellis.io/mutli-stage-docker-builds/

- 使用一个 `Dockerfile`
- 可以使用多个 `FROM`
- 每个 `FROM` 的父镜像可以不同
- 从每个 `FROM` 开始都是一个相对独立的 `build` 过程
- 从之前的 `build` 结果中只 `提取` 需要的内容，其他舍弃

如何使用

multi-stage dockerfile

```
# builder
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o a
pp .

# publisher
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

在 `publisher` 中，使用了 `COPY --from=0` 从上一次 `build` 的结果中提取了编译结果。源代码和其他不需要的文件就放弃了。

build stage 别名

```
# builder
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o a
pp .

# publisher
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/ap
p .
CMD ["./app"]
```

- 在 `builder` 阶段，对 `FROM` 指定使用了 `别名` `(as builder)`。
- 相应的，在 `publisher` 阶段， `COPY` 指令使用了 `--from=builder` 指定提取来源。
    - 使用 `as <name>` 别名，即使调整 `build` 顺序也不会影响结果。

Docker container networking

Default networks

- docker 提供了三个 `内建(built-in)` 网络

```
$ docker network ls
NETWORK ID          NAME                DRIVER
    SCOPE
ad747e2a983c        bridge              bridge
    local
3c2b536039ac        host                host
    local
ffa7c88de9e4        none                null
    local
```

- `内建 bridge` 网络为 `docker0`，可以通过 `ip addr show` 或 `ip a` 或 `ifconfig` 查看

```
$ ip addr show docker0
5: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdi
sc noqueue state DOWN group default
  link/ether 02:42:68:2f:d5:ec brd ff:ff:ff:ff:ff:ff
  inet 172.17.0.1/16 scope global docker0
     valid_lft forever preferred_lft forever
  inet6 fe80::42:68ff:fe2f:d5ec/64 scope link
     valid_lft forever preferred_lft forever
```

- 启动容器的时候，通过 `docker run --network=<network>` 指定容器所在的网络。否则默认使用 `内建` 的 `bridge`。

> **Running on Docker for Mac or Docker for Windows?**
>
> > If you are using Docker for Mac (or running Linux containers on Docker for Windows), the `docker network ls` command will work as described above, but the `ip addr show` and `ifconfig` commands may be present, but will give you information about the IP addresses for your local host, `not Docker container networks`. `This is because Docker uses network interfaces running inside a thin VM, instead of on the host machine itself.`
> >
> > To use the `ip addr show` or `ifconfig` commands to browse Docker networks, log on to a Docker machine such as a local VM or on a cloud provider like a Docker machine on AWS or a Docker machine on Digital Ocean. You can use `docker-machine ssh <machine-name>` to log on to your local or cloud hosted machines, or a direct `ssh` as described on the cloud provider site.

- `none` 网络将容器加入到一个 `container-specific` 网络堆栈中。这些容器 `没有` 网络接口

```
### 将容器加入 none 网络
$ docker run -it --rm --network=none busybox

### 以下为容器内
/ #
/ # cat /etc/hosts
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
/ #
/ # ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

> 注意：使用 `CTRL-p CTRL-q` 命令可以退出容器同时保持容器运行。

- `host` 网络将容器加入到 `宿主机` 网路堆栈。
  - 容器一旦加入 `host` 网络中，那么 `容器` 和 `宿主机` 之间 `不再有网络隔离` 。
  - 举个栗子，`host` 网络中的某个容器启用了 80 端口，那么宿主机也就启用了 80 端口

```
## default bridge network

$ docker run --rm -itd --name=nginx_docker0 nginx
88d13e3a1bc23d0f03d17c5dce6bd12d32643c9930c96f51b38ae1462be35177

$ netstat -tunpl
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
```

```
Proto Recv-Q Send-Q Local Address           Foreign Address
    State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*
    LISTEN      -

$ curl 127.0.0.1
curl: (7) Failed to connect to 127.0.0.1 port 80: Connection ref
used

$ docker stop nginx_docker0
nginx_docker0



## host network

$ docker run --rm -itd --name=nginx_host --network=host nginx
af543af4dc4fdf44142642cc5978591fd56ee4dd4acddc16426102ae24971d40

$ netstat -tunpl
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
    State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*
    LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*
    LISTEN      -

$ curl 127.0.0.1
<!DOCTYPE html>
...省略..
</html>

$ docker stop nginx_host
nginx_host
```

- `none` 和 `host` 网络不能通过 Docker 直接管理，但是 `built-in`

`bridge` 可以。

- `用户自建(user-defined) bridge` 网络也可以通过 Docker 直接管理

The default bridge network

- 使用 `docker network inspect <network_name>` 查看网络信息

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "ad747e2a983c7004e92ec878e7c634ee3f0312278b71366f3
f0285a713ea8f52",
        "Created": "2017-09-18T09:23:24.488923085+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "t
```

```
rue",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.
0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

启动两个 `busybox` 容器，并加入到 `默认 bridge` 网络

```
$ docker run --rm -itd --name=container1 busybox
b09c8abd1eb1cb1d12deb0da841bba45392e1ee3c0f4b6222c472568c6e4d932

$ docker run --rm -itd --name=container2 busybox
22a09d5501560c26096d985cc8be074a778dd047a960de3c8380419dfe50794f
```

启动容器后，重新 `inspect` 查看网络信息，会发现两个容器的网络信息也在里面

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "ad747e2a983c7004e92ec878e7c634ee3f0312278b71366f3
f0285a713ea8f52",
        "Created": "2017-09-18T09:23:24.488923085+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
```

```
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "22a09d5501560c26096d985cc8be074a778dd047a960de3c8380419dfe50794f": {
                "Name": "container2",
                "EndpointID": "e961a9a1d1690026b0e0be0b82d3c5f192728bd4f994571bbc7be70102c8c467",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },
            "b09c8abd1eb1cb1d12deb0da841bba45392e1ee3c0f4b6222c472568c6e4d932": {
                "Name": "container1",
                "EndpointID": "48969713f853fcebe8ed578420f0d9935f3f41c7911a09aedc61b0aa016c52e0",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
```

```
]
```

- 同在 `default bridge` 网络中的容器，只能通过 `ip addr` 互相 `通信(communicate)`
- docker `default bridge` 网络 `不支持` `自动发现(automatic service discovery)` 。
    - 如果你想通过 `容器名(container name)` `解析(resolve)` 容器的 `ip addr` ，则需要使用 `自定义网络`
    - 或者，在启动容器的时候使用 `--link` 标签。
    - 不推荐在 `default bridge` 以外的网络中使用 `--link` 。`--link` 以后可能会被废除。
- 使用 `docker attach <container_name>/<container_id>` 进入一个正在运行的容器
    - `ping` container2 的 ip 地址 `(172.17.0.3/16)` , `通`
    - `ping` container2 的 容器名 `(container2)` , `不通`
    - 查看 container1 的 `/etc/hosts` 文件

```
$ docker attach container1
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:828 (828.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping -c 3 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
```

```
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.114 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.079 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.102 ms

--- 172.17.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.079/0.098/0.114 ms
/ #
/ # ping -c 3 container2
ping: bad address 'container2'
/ #

/ # cat /etc/hosts
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
172.17.0.2    b09c8abd1eb1
```

- `default bridge` 支持 `端口映射(port mapping)`
- `default bridge` 支持使用 `--link` 实现通过 `container name` 进行容器见的通信。
  - 更推荐使用 `自建网络` 代替 `--link`

User-defined networks

使用 `自定义 bridge 网络`

- 支持容器之间可以互相通信
- 支持 `容器名` 与 `容器IP` 的 `自动 DNS 解析(automatic DNS resolution)`
- 可以通过默认的 `network driver` 创建 `bridge network`，`overlay network` 和 `MACVLAN network`.
- 自定网络数量无上限
- 一个容器可以在任意时刻加入 `0个` 或 `多个` 网络

- 容器 `加入` 或 `退出` 网络不需要重启。
- 当一个容器加入 `多个` 网络时，容器 `对外交互` 使用按 `字母排序( in lexical order)` 的 `第一个` `非内部(non-internal)` 网络。

Bridge networks

自定义 `bridge` 网络与默认 `bridge` 网络相似

- 但 `增加了` 一些功能，并且 `移除了` 一些 `老旧` 的功能
- 使用 `docker network create` 创建网络
- 使用 `docker network inspect` 查看网络信息

```
$ docker network create --driver bridge isolated_nw
ea31d15e26277e0e7a7f4a8945d5e4cbe2545ab5d4fd3c384e1bec70783475d0

$ docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "ea31d15e26277e0e7a7f4a8945d5e4cbe2545ab5d4fd3c384
e1bec70783475d0",
        "Created": "2017-09-18T20:33:55.745048069+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

- 创建容器时，使用 `--network=<network_name>` 加入所选网络

```
$ docker run -itd --rm --network=isolated_nw --name=container3 b
```

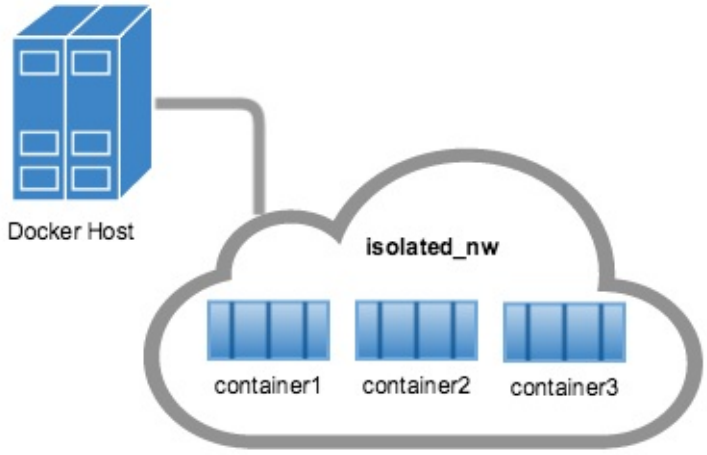
```
usybox

ac378978b8da5e32ebe3289bc7f2a151f4b2349c06fc7136b78052d47aeab403
$
$ docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "ea31d15e26277e0e7a7f4a8945d5e4cbe2545ab5d4fd3c384
e1bec70783475d0",
        "Created": "2017-09-18T20:33:55.745048069+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "ac378978b8da5e32ebe3289bc7f2a151f4b2349c06fc7136b78
052d47aeab403": {
                "Name": "container3",
                "EndpointID": "7b6d9f64f1631656e56d5368070a69ed2
04502b36e0871b88fee4a2902ee6775",
                "MacAddress": "02:42:ac:15:00:02",
                "IPv4Address": "172.21.0.2/16",
                "IPv6Address": ""
```

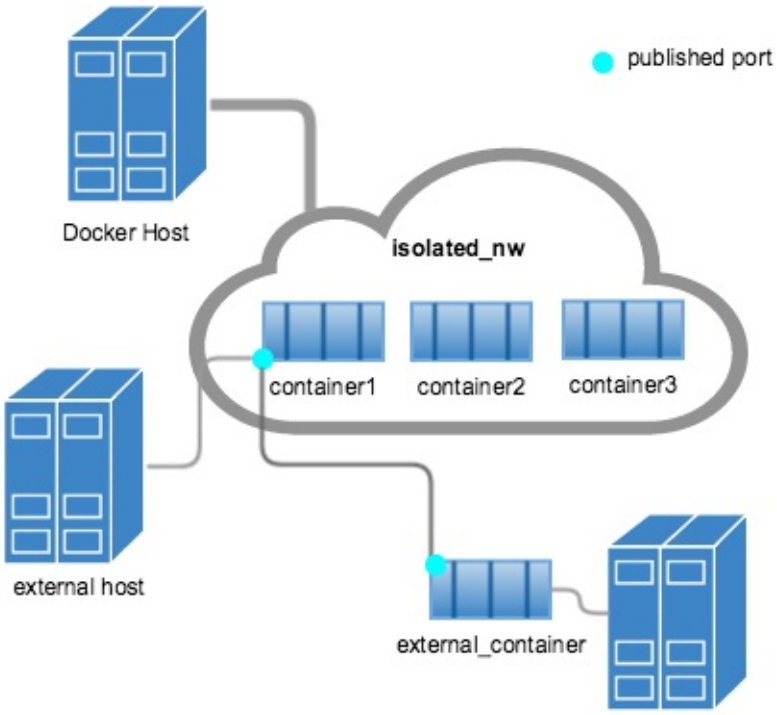
```
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

就 `bridge` 网络而言

- 在同一 `宿主机` 上的容器被创建后，可以 `立即` 与彼此通信。
- `bridge` 网络与 `外部(external)` 网络隔离。



- `user-defined bridge` 网络，不支持 `--link`
- 可以为网络中的容器 `expose` 和 `publish` 端口
  - `expose` 对外暴露端口
  - `publish` 映射 `宿主机` 端口到 `容器` 暴露的端口



- `bridge network` 适合在 `单宿主机` 上创建一个 `相对较小` 的网络。
- 如果要创建一个跨 `多宿主机` 的 `大型` 网络，可以通过 `overlay` 网络实现。

The `docker_gwbridge` network

`docker_gwbridge` 网络是一个 `本地桥接网络(local bridge network)` ；在以下两种情况下，有 `Docker` 自动创建：

- 当 `初始化(initialize)` 或 `加入(join)` 一个 `swarm` , docker 会创建一个 `docker_gwbridge` 网络，并通过该网络在 `不同主机` 的 `swarm nodes` 之间通信。
- 当容器不能提供外部链接的时候，除了容器的本身的网络之外， Docker 还会将容器连接到 `docker_gwbridge` 网络中， 这样容器就可以连接到外部网络或其他 `swarm nodes` 。
- 可以提前创建一个 `docker_gwbridge` 网络，并设置 `自动有配置(custom configuration)`
- 否认， Docker 会在需要的时候自行创建。

举个例子，如何创建一个 `docker_gwbridge` 网络

```
$ docker network create --subnet 172.30.0.0/16 \
```

```
                        --opt com.docker.network.bridge.name=doc
ker_gwbridge \
                        --opt com.docker.network.bridge.enable_i
cc=false \
                        docker_gwbridge
ebbab3861c25cf862c2733602f46c147aab41c701321319c7a6d03907a22850f

$ docker network inspect docker_gwbridge
[
    {
        "Name": "docker_gwbridge",
        "Id": "ebbab3861c25cf862c2733602f46c147aab41c701321319c7
a6d03907a22850f",
        "Created": "2017-09-18T21:09:59.948604073+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.30.0.0/16"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.enable_icc": "false",
            "com.docker.network.bridge.name": "docker_gwbridge"
        },
        "Labels": {}
    }
```

```
]
```

- 当使用 `overlay` 网络时， `docker_gwbridge` 始终会出现

Overlay networks in swarm mode

You can create an `overlay` network on a `manager node` running in `swarm mode` `without` an `external key-value store` . The swarm makes the overlay network `available` `only` to nodes `in` the swarm that require it for a service. When you create a service that uses the overlay network, the manager node `automatically` `extends the overlay network to nodes` that run service tasks.

使用 `swarm mode` 时，

- 通过 `manager node` 创建一个 `overlay` 网络，且 `不使用` 外部 `key-value 存储`
- 只有加入 `swarm` 了的 `node` 运行相同服务时， `overlay` 网络才会生效。
- `manager node` `自动` 将 `overlay` 网络扩展到所有运行相同服务的 `node`

学习 Swarm mode overview.

举个例子，创建一个 `overlay` 网络，并用于 `swarm` .

```
## 当前主机必须为 manager node 才能创建 overlay 网络 ，否则会报错。
$ docker network create     \
      --driver overlay      \
      --subnet 10.0.9.0/24  \
      my-multi-host-network

#### 报错
Error response from daemon: This node is not a swarm manager. Us
e "docker swarm init" or "docker swarm join" to connect this nod
e to swarm and try again.


## 初始化 swarm
$ docker swarm init  --advertise-addr 192.168.56.212
```

```
Swarm initialized: current node (s9c14hz5hfwqmvvl2spe6xkri) is n
ow a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5p9kychp1zobk33828qqallmo
z35y8qqe2xh60mj66s8n2ly3o-3pc84g6dcrwdtnk96jtfgrr24 192.168.56.2
12:2377

To add a manager to this swarm, run 'docker swarm join-token man
ager' and follow the instructions.


## 创建 overlay 网络
$ docker network create      \
       --driver overlay      \
       --subnet 10.0.9.0/24  \
       my-multi-host-network
wy9ddlzjyx6ljetv6f134dapq


## 在 overlay 网络上创建一个 service
$ docker service create --replicas 2 --network my-multi-host-net
work --name my-web nginx
ws6vwpptohjutafbaxzxonw49
Since --detach=false was not specified, tasks will be created in
 the background.
In a future release, --detach=false will become the default.
```

- `Only` swarm `services` `can` connect to overlay networks, `not standalone` containers.

An overlay network without swarm mode

- If you are not using Docker Engine in swarm mode, the `overlay` network requires `a valid key-value store` service.
- Supported key-value stores include `Consul`, `Etcd`, and `ZooKeeper` (Distributed store).

- `Before` creating a network in this way, you `must install` and `configure` your chosen key-value store service.
- The Docker hosts that you intend to network and the service must `be able to communicate` .

> Note: Docker Engine running in `swarm mode` is `not compatible` with `networking with an external key-value store` .

- 对于大多数用户而言，不推荐使用这种方式
- 这种方式在以后可能会被废弃。
- 如果还是想使用这种方法，看 guide

Custom network plugins

自己写驱动。

创建方式与其他网络相同。

Embedded DNS server

- Docker daemon runs an `embedded DNS` server which provides DNS resolution `among containers` connected to the `same user-defined network` , so that these containers can resolve container `names to IP addresses` .
- If the embedded DNS server is `unable` to resolve the request, it will be `forwarded` to `any external DNS servers` configured for the container.
- To facilitate this when the container is created, only the embedded DNS server reachable at `127.0.0.11` will be listed in the container's `resolv.conf` file. (看不懂)
  - 在测试机上，所有容器的 `resolv.conf` 都与 `宿主机` 的 `resolv.conf` 相同
- For more information on embedded DNS server on user-defined networks, see embedded DNS server in user-defined networks

Exposing and publishing ports

docker 中有两个机制直接涉及网络端口: `expose` 和 `publish` 。适用于 `default bridge` 和 `user-defined bridge` 网络

- 暴露容器端口
  - `Dockerfile` 中的 `EXPOSE` 关键字
  - `docker run` 命令中的 `--expose` 标识。
  - Exposing ports is `a way of documenting` which ports are used, but does `not` actually `map or open` any ports.
  - 可选
- 映射容器端口
  - `Dockerfile` 中的 `PUBLISH` 关键字
  - `docker run` 命令中的 `--publish / -p` 标识。
  - This tells Docker which ports to `open` on the container's network interface
  - 宿主机随机映射一个高端口号(大于 `30000`)到容器内部。
  - 使用 `-p local_port:container_port` 指定端口映射。
  - 不能在 `Dockerfile` 中映射端口，因为无法保证启动容器时端口没被占用。

举个例子，启动容器开放80端口，本机随机映射一个 `高端口 37268`

```
$ docker run -itd --rm -p 80 nginx
4a606e40ebfc300fefd8c98c7ee0f46090138fce8bcdd90b46d92910f1980525

$ docker container ls
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
 NAMES
4a606e40ebfc        nginx               "nginx -g 'daemon ..."
 54 seconds ago      Up 53 seconds       0.0.0.0:32768->80/tcp
 unruffled_kowalevsk
```

在举个例子，指定 `宿主机` 的 `8080` 端口映射到 `容器` 的 `80` 端口。

```
$ docker run -itd --rm --name=nginx -p 8080:80 nginx
e398ac99f370a40c321a7dfc2dbc2237c2c6f078691ea8273485804a16de8d01
```

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
NAMES
e398ac99f370        nginx               "nginx -g 'daemon ..."
 15 seconds ago      Up 14 seconds       0.0.0.0:8080->80/tcp
nginx


## Use a proxy server with containers

为容器配置 `HTTP`, `HTTPS`, `FTP` 代理服务器。

+ In Docker `17.07` and `higher`, you can `configure the Docker
client` `to pass` proxy information to containers `automatically
`.

+ In Docker `17.06 and lower`, you must `set` appropriate `envir
onment` variables `within the container`. You can do this when y
ou build the image (which makes the image less portable) or when
 you create or run the container.


### Configure the Docker Client

> **Edge only**: This option is only available in Docker CE Edge
 versions. See Docker CE Edge.

+ 编辑文件 `~/.config.json`, 启动容器的的用户家目录。
  + 配置如下，按需求替换 `httpPorxy` 为 `httpsProxy` 或 `ftpProxy`
  + 可以同时配置多个代理服务器
+ 启动容器时，会自动为容器添加代理

```json
{
  "proxies":
  {
    "httpProxy": "http://127.0.0.1:3001",
    "noProxy": "*.test.example.com,.example2.com"
  }
```
```

```
}
```

Set the environment variables manually

- 在使用 `dockerfile` 创建镜像时
- 在使用 `--env` 标识启动容器时
- 可以同时指定多个
- 17.07 版本以后，应该使用上述配置方式实现。

variable	`dockerfile` example	`docker run`
HTTP_PROXY	`ENV HTTP_PROXY "http://127.0.0.1:3001"`	`--env HTTP_P "http://127.0`
HTTPS_PROXY	`ENV HTTPS_PROXY "https://127.0.0.1:3001"`	`--env HTTPS_ "https://127.`
FTP_PROXY	`ENV FTP_PROXY "ftp://127.0.0.1:3001"`	`--env FTP_PR "ftp://127.0.`
NO_PROXY	`ENV NO_PROXY "*.test.example.com,.example2.com"`	`--env NO_PRO "*.test.examp`

Links

- `--link` 实现了在 `default bridge` 网络中， `容器` 之间使用 `容器名` 进行通信。
- 尽量避免使用 `--link`

更多 Legacy Links

Docker and iptables

- 在 `linux 主机` 中，docker 使用 `iptables` 管理网络设备，包括 `routing`，`port forwarding`，`network address translate(NAT)` 和 `其他`
- docker 会修改 `iptables`
  - 在启动或关闭 `映射端口` 的容器时
  - 创建或修改网络
  - 容器加入网络

    - 其他网络相关操作
- `iptables` 需要开机启动 `run-level 3`
    - 或者网络初始化后启动

Docker `dynamically` manages `iptables` rules for the daemon, as well as your containers, services, and networks. In Docker 17.06 and higher, you can `add` rules to a new table called `DOCKER-USER`, and these rules will be `loaded before` any rules Docker creates `automatically`. This can be useful if you need to `pre-populate` iptables rules that need to be in place before Docker runs.

Work with network commands

> https://docs.docker.com/engine/userguide/networking/work-with-networks/

本文案例都是在 `bridge` 网络下实现的。如果要了解 `overlay` 网络，看 003.get_start_with_multi-host_networking.md

可用命令：

```
$ docker network --help

Usage:    docker network COMMAND

Manage networks

Options:
      --help   Print usage

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networ
ks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks

Run 'docker network COMMAND --help' for more information on a co
mmand.
```

Create networks

- 安装 Docker 之后，会默认创建一个 `bridge` ，网络接口是 `docker0` 。这个你已经知道了。
- 使用 `docker network create` 创建自定义 `bridge` 或 `overlay` 网络。

- `bridge` 网络在 `单机` 环境
- `overlay` 网络可以跨多主机。但需要 `key-value store` 的支持。

创建网络

在不指定 `--driver` 的情况下，默认使用 `bridge driver` 。

```
$ docker network create simple-network
544c3cd2638b03f647f4d2332cc261c6a3898dc454a6b324b91878efe5047049

$ docker network inspect simple-network
[
    {
        "Name": "simple-network",
        "Id": "544c3cd2638b03f647f4d2332cc261c6a3898dc454a6b324b
91878efe5047049",
        "Created": "2017-09-19T11:24:42.370489908+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

与创建 `bridge` 网络不同，创建 `overlay` 网络需要一些前置条件，包括：

- 可访问的 `key-value store` ，支持使用 `Consul` , `Etcd` , `Zookeeper`

- 集群中的所有主机都可以与 `key-value store` 通信
- swarm 模式下，适当的配置所有主机 `docker engine` 的 `daemon` 。

`dockerd` 对 `overlay` 网络支持的参数，包括：

- `--cluster-store`
- `--cluster-store-opt`
- `--cluster-advertise`

- 当创建网络时， `docker engine` 会创建一个 `非覆盖子网(non-overlapping subnetwork)`
  - 非覆盖子网 指的是通过 `docker engine` 创建的
  - 与 `主机` 网络无关
- 在创建的时候，可以通过 `--subnet` 指定网络范围
  - 在创建 `bridge` 网络时，只能指定 `一个` 网段。
  - 但创建 `overlay` 网络时，可以创建 `多个` 网段。

> 注意: 强烈推荐使用 `--subnet` 。如果不适用 `--subnet` 的话，默认创建的子网可能会与物理主机所在的子网重叠。从而产生不可预料的问题。

除了可以使用 `--subnet` ，还可以：

```
$ docker network create --help

Usage:    docker network create [OPTIONS] NETWORK

Create a network

Options:
      --attachable             Enable manual container attachmen
t
      --aux-address map        Auxiliary IPv4 or IPv6 addresses
used by Network driver (default map[])
      --config-from string     The network from which copying th
e configuration
      --config-only            Create a configuration only netwo
rk
  -d, --driver string          Driver to manage the Network (def
ault "bridge")
      --gateway stringSlice    IPv4 or IPv6 Gateway for the mast
er subnet
      --help                   Print usage
      --ingress                Create swarm routing-mesh network
      --internal               Restrict external access to the n
etwork
      --ip-range stringSlice   Allocate container ip from a sub-
range
      --ipam-driver string     IP Address Management Driver (def
ault "default")
      --ipam-opt map           Set IPAM driver specific options
(default map[])
      --ipv6                   Enable IPv6 networking
      --label list             Set metadata on a network
  -o, --opt map                Set driver specific options (defa
ult map[])
      --scope string           Control the network\'s scope
      --subnet stringSlice     Subnet in CIDR format that repres
ents a network segment
```

创建一个 **overlay** 网络

在执行以下命令前，须确认命令中的 `ip 段` 不会覆盖你的主机网络。

```
## 创建一个覆盖网络
$ docker network create -d overlay \
 --subnet=192.168.0.0/16 \
 --subnet=192.170.0.0/16 \
 --gateway=192.168.0.100 \
 --gateway=192.170.0.100 \
 --ip-range=192.168.1.0/24 \
 --aux-address="my-route=192.168.1.5" --aux-address="my-switch=1
92.168.1.6" \
 --aux-address="my-printer=192.170.1.5" --aux-address="my-nas=19
2.170.1.6" \
 my-multihost-network

## 由于没有 key-value store 也不在 swarm mode 下因此报错
Error response from daemon: This node is not a swarm manager. Us
e "docker swarm init" or "docker swarm join" to connect this nod
e to swarm and try again.

## 初始化 swarm
$ docker swarm init --advertise-addr 192.168.56.205

## 重新创建
$ docker network create -d overlay \
  --subnet=192.168.0.0/16 \
  --subnet=192.170.0.0/16 \
  --gateway=192.168.0.100 \
  --gateway=192.170.0.100 \
  --ip-range=192.168.1.0/24 \
  --aux-address="my-route=192.168.1.5" --aux-address="my-switch=
192.168.1.6" \
  --aux-address="my-printer=192.170.1.5" --aux-address="my-nas=1
92.170.1.6" \
  my-multihost-network

uzjd9y9dneh63s38bf17zncg7
```

创建用户网络时，可以传递额外的选项。 `bridge` 支持:

Option	Equivalent	Equ
`com.docker.network.bridge.name`	`-`	网络
`com.docker.network.bridge.enable_ip_masquerade`	`--ip-masq`	启用
`com.docker.network.bridge.enable_icc`	`--icc`	开启内部通性
`com.docker.network.bridge.host_binding_ipv4`	`--ip`	bir 容器默认宿主
`com.docker.network.driver.mtu`	`--mtu`	设置络

The `com.docker.network.driver.mtu` option is also supported by the overlay driver

所有 `network driver` 共用参数

- `--internal` : Restrict(限制) external access to the network
- `--ipv6` : 启动 IPv6 网络

创建网络时，使用 `-o` 将端口映射时绑定到指定 `IP` (uses `-o` to bind to a specific IP address when binding ports)

- 网络创建时，`host_binding_ipv4` 不存在是不会报错的
- 容器创建时，`host_binding_ipv4` 不存在，无法正常创建容器

```
## 绑定一个不存在的 宿主机 IP，不会报错。
$ docker network create     -o "com.docker.network.bridge.host_b
inding_ipv4"="172.233.0.1"     simple-network2
f0d572fb3a0f903f4d3a37d400b815d17375735518bcfddc824b582123f4732f

## 查看网络信息时，可以看出，网段和 bind 的ip没有特别的的关系
$ docker network inspect simple-network2
[
    {
        "Name": "simple-network2",
        "Id": "f0d572fb3a0f903f4d3a37d400b815d17375735518bcfddc8
```

```
24b582123f4732f",
        "Created": "2017-09-19T13:10:03.303424685+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.host_binding_ipv4": "172.
233.0.1"
        },
        "Labels": {}
    }
]

## network 创建成功后，也没有自动生成任何有关 172.233 的 ip 信息
$ ip a |grep 172.233


## 在 simple-network2 中创建容器时，由于无法将 映射的端口绑定到 172.233
.0.1 上，因此报错，创建失败
$ docker run --rm --network=simple-network2 -itd --name=nginx  -
p 8081:80 nginx
e5b03e37ed77dd5bb8d9fefe1d059506d10db9b44e4bf4451b9772fc3614125e
```

```
docker: Error response from daemon: driver failed programming ex
ternal connectivity on endpoint nginx (6f4c9e05e494cce098563d997
77f105ede1485b326df187155936cb233fa6d97): Error starting userlan
d proxy: listen tcp 172.233.0.1:8081: bind: cannot assign reques
ted address.
```

正常绑定一个 宿主机 **IP**

```
$ docker network create -o "com.docker.network.bridge.host_bindi
ng_ipv4"="192.168.56.205" simple-network3
5c1a9d5eed7316ef4fc8d05a76056750df97c08bc07a67a451053d256366352a

$ docker run --rm --network=simple-network3 -itd --name=nginx  -
p 8081:80 nginx
cc0f03ef9cc06d6477dfde1ce1807326b995bd6c991aa96734bdeae10e872155

$ netstat -tunpl
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
    State       PID/Program name
tcp        0      0 192.168.56.205:8081     0.0.0.0:*
    LISTEN      -

$ ip a  |grep 'inet '
    inet 127.0.0.1/8 scope host lo
    inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
    inet 192.168.56.205/24 brd 192.168.56.255 scope global enp0s
8
    inet 172.20.0.1/16 scope global br-792f2fc2cdc4
    inet 172.17.0.1/16 scope global docker0
    inet 172.19.0.1/16 scope global docker_gwbridge
    inet 172.18.0.1/16 scope global br-5c1a9d5eed73

## 只能通过 host_binding_ipv4 指定的ip访问
$ curl 127.0.0.1:8081
curl: (7) Failed to connect to 127.0.0.1 port 8081: Connection r
efused

$ curl 192.168.56.205:8081
<!DOCTYPE html>
...省略...
</html>
```

Connect containers

- 一个容器可以加入多个不同类型的网络
- 一旦容器加入网络，通网络之间的容器就可以通过 `容器名` 或 `容器ip` 进行通信。

For `overlay` networks or `custom plugins` that support `multi-host connectivity`, 容器在相同的网络，但属于不同的主机，也可以相互通信。

Basic container networking example

1. 在默认网络中，创建 `container1` 和 `container2`

```
$ docker run -itd --name=container1 busybox
$ docker run -itd --name=container2 busybox
```

1. 创建一个用户隔离网络

```
$ docker create network -d bridge --subnet 172.25.0.0/16 iso
lated_nw
```

2. 将 `container2` 加入到 `isolated_nw` 网络 。
3. 创建 `container3` ，并加入 `isolated_nw` ，使用 `--ip` 指定 ip 地址。

```
$ docker run --network=isolated_nw --ip=172.52.3.3 -itd --na
me=container3 busybox
```

4. 使用 `--ip` 或 `--ip6` 给容器指定 IP 地址
    - 在 `user-defined network` 中，指定容易ip后，重启容器会保留指定的 IP。
    - docker daemon 重启后，只有默认网络保留中的指定 ip 会被保留。
        - 因为 `user-defined network` 可能发生变化。
5. 查看 `container3` 的网络信息

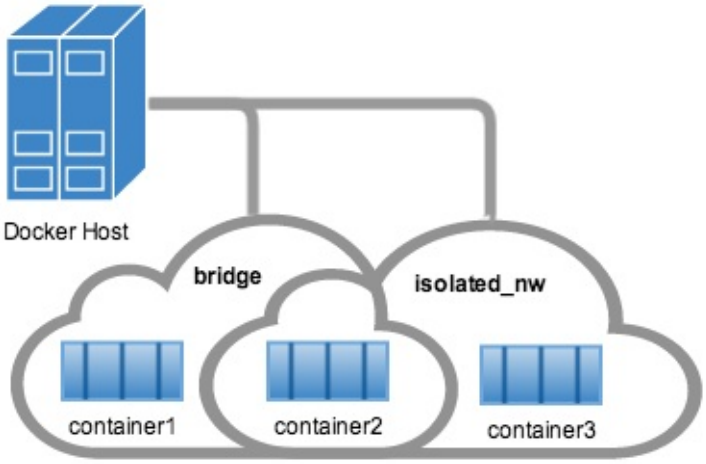
```
$ docker inspect --format='' container3
```

6. 查看 `container2` 的网络信息，并使用 `python` 做美化输出

```
$ docker inspect --format='' container2 | python -m json.too
l
```

注意，容器所属网络如下

- `container1` 只连接到 `default bridge`
- `container2` 同时连接到 `default bridge` 和 `isolated_nw`
- `container3` 只连接到 `isolated_nw`



1. 使用 `docker attach <container_name>` 进入 `container2`

   ```
   $ docker attach container2
   ```

   在容器中使用 `ifconfig` 命令查看容器网络信息

2. docker 嵌入的 dns 服务器可以将 `user-defined` 网络中的 `容器名` 解析为 `容器IP` 。因此，可以通过 `container2` ping 通 `containr3`

```
## in container2
/ # ping -w 4 container3
PING container3 (172.25.3.3): 56 data bytes
64 bytes from 172.25.3.3: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.3.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.3.3: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.3.3: seq=3 ttl=64 time=0.097 ms

--- container3 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

由于 `container2` 和 `container1` 属于 `default bridge network`，因此不能直接通过 `容器名` 进行交互。但可以使用 `容器IP`

```
## in container2
/ # ping -w 4 container1
ping: bad address 'container1'

/ # ping -w 4 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.095 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.075 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.072 ms
64 bytes from 172.17.0.2: seq=3 ttl=64 time=0.101 ms

--- 172.17.0.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.072/0.085/0.101 ms
```

1. 由于 `container1` 和 `container3` 不再同一个网络，因此即使使用 `容器IP` 也无法通信

```
$ docker attach container3

$ ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
^C

--- 172.17.0.2 ping statistics ---
10 packets transmitted, 0 packets received, 100% packet loss
```

> 注意：即使容器不是 `running` 状态，也可以将其连接到某个网络。
>
> > 但 `docker network inspect <network_name>` 只显示 `running` 状态的容器信息。

Link containers without using user-defined networks

- `default bridge network` 中的容器不能自动将容器名解析为容器 IP
- `default bridge network` 中的容器相互使用容器名进行访问，需要使用 `--link` 。
  - 只建议将 `--link` 用在这里。
  - 建议使用 `user-defined network` 替代 `--link`

为 `default bridge network` 中的容器使用 `--link` 之后:

- 可以解析 `contaner name` 为 `container ip` 了
- 可以为容器指定一个 `网络别名( network alias)` ， `--link=container_name:alias_name`
- 安全的容器连接性，(in isolation via `--icc=false`)
- 环境变量注入(environment variable injection)

再次强调，以下功能在 `user-defined network` 中不需要额外配置就可以使用。此外，动态的加入或退出多个网络。 **Additionally, you get the ability to dynamically attach to and detach from multiple networks.**

- 内嵌 DNS， 自动 `容器名` 解析
- 支持 `--link` 的别名功能
- automatic secured isolated environment for the containers in a network

- 环境变量注入

怎么使用 `--link`

1. 继续之前的案例，创建 `container4` 并加入 `isolated_nw`，使用 `--link` 为 `container5` *(尚未创建)* 指定别名。

   ```
   $ docker run --network=isolated_nw -itd --name=container4 --
   link container5:c5 busybox
   ```

   一旦 `container5` 被创建，`container4` 就能将 `c5` 解析到 `container5` 的 IP 上。

> 注意：
>
> 1. 在 `default bridge network` 中，使用 `legacy link` 创建的 `link` 是 `静态的(static)`，且容器与别名是 `硬绑定的(hard-binds)`，因此能 `容忍` 容器重启(`restarts`)。
> 2. 在 `user-defined network` 中， `new link` 功能支持容器间的 `动态(dynamic)` 连接。 因此，支持容器启动，并重新解析容器IP。
>
> 注意2：
>
> 在 `default bridge network` 中，使用 `legacy link` 不能为不存在的容器指定别名。
>
> 在 `user-defined network` 中，使用 `new link` 可以为不存在的容器指定别名。

```
$ docker run -itd --name container4 --link container5:c5 busybox
docker: Error response from daemon: Could not get container for
container5.
See 'docker run --help'.

$ docker run -itd --name container4 --network=isolated_nw --link
 container5:c5 busybox
fcfbe26df2db9b5bc6c9645bc7b32753d5abccc2f74d1f58ca3382692ee1a140
```

1. 创建 `container5` ，加入 `isolated_nw` ，使用 `--link` 为

`container4` 指定别名

```
$ docker run --network=isolated_nw -itd --name=container5 --
link container4:c4 busybox
```

在创建 `container5` 之前，`container4` 不能 ping `c5/container5` 。创建之后，二者可以互 ping 了。

Network alias scoping example

- `--link` 别名只作用于指定的 `container` ，在 `其他 container` 中无效。
- `--link` 别名只作用于指定的 `network` 中，即使一个容器处于多个网络，在其他网络中也不能使用 `别名`

书接上文，举个例子

1. 创建一个 `bridge` 网络名为 `local_alias`

```
$ docker network create -d bridge --subnet 172.26.0.0/24 loc
al_alias
```

2. 把 `container4` 和 `container5` 分别加入到该网络中，并做别名 ```bash

$ docker network connect --link container5:foo local_alias container4 $ docker network connect --link container4:bar local_alias container5

3. 现在，`container4`, `container5` 都在 `isolated_nw` 和 `local_alias` 网络中。因此，进入 `container4` 可以 ping 通 `container5` 的两个别名。**但是可以看到，两个别名解析的 `IP地址` 是不同的**

```bash
$ docker attach container4

 / # ping -w 4 foo
 PING foo (172.26.0.3): 56 data bytes
 64 bytes from 172.26.0.3: seq=0 ttl=64 time=0.070 ms
 64 bytes from 172.26.0.3: seq=1 ttl=64 time=0.080 ms
 64 bytes from 172.26.0.3: seq=2 ttl=64 time=0.080 ms
 64 bytes from 172.26.0.3: seq=3 ttl=64 time=0.097 ms

 --- foo ping statistics ---
 4 packets transmitted, 4 packets received, 0% packet loss
 round-trip min/avg/max = 0.070/0.081/0.097 ms

 / # ping -w 4 c5
 PING c5 (172.25.0.5): 56 data bytes
 64 bytes from 172.25.0.5: seq=0 ttl=64 time=0.070 ms
 64 bytes from 172.25.0.5: seq=1 ttl=64 time=0.080 ms
 64 bytes from 172.25.0.5: seq=2 ttl=64 time=0.080 ms
 64 bytes from 172.25.0.5: seq=3 ttl=64 time=0.097 ms

 --- c5 ping statistics ---
 4 packets transmitted, 4 packets received, 0% packet loss
 round-trip min/avg/max = 0.070/0.081/0.097 ms
```

1. 将 `container5` 从 `isolated_nw` 中退出。重新在 `container4` 中 ping `container5` 中的两个别名。可以看到 `c5` 不行了，但是 `foo` 还可以

```
$ docker network disconnect isolated_nw container5

$ docker attach container4

/ # ping -w 4 c5
ping: bad address 'c5'

/ # ping -w 4 foo
PING foo (172.26.0.3): 56 data bytes
64 bytes from 172.26.0.3: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.26.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.26.0.3: seq=3 ttl=64 time=0.097 ms

--- foo ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

Limitations of docker network

`docker network` 有一些局限性：

环境变量注入(ENVIRONMENT VARIABLE INJECTION)

- 环境变量注入是静态的，且环境变量在容器启动后就不能改变
- `legacy --link` 可以共享连接容器之间的 `所有` 环境变量，而 `docker network` 命令则不同
- 当使用 `docker network` 连接容器时，环境变量不会动态的在容器间共享。

使用 网络扩展别名(network-scoped alias)

- `legacy link` 提供的 `传出名称解析(outging name resolution)` 隔离是基于容器配置的。
- 而 `network-scoped alias` 是提供给网络中所有容器的，不支持 `单向(one-way)` 隔离。

书接上文，举个例子

1. 创建容器 `container6` ，加入 `isolated_nw` 网络，并使用 `--network-alias <alias_name>` 选项为容器取别名

```
$ docker run --network=isolated_nw -itd --name=container6 --netw
ork-alias app busybox
```

1. 进入 `container4` 中，ping `container6` 的容器名(*`container6`*)和别名(*`app`*)，可以发现都通，且 `IP` 地址一样

```
$ docker attach container4

/ # ping -w 4 app
PING app (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms

--- app ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

/ # ping -w 4 container6
PING container5 (172.25.0.6): 56 data bytes
64 bytes from 172.25.0.6: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.25.0.6: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.25.0.6: seq=3 ttl=64 time=0.097 ms

--- container6 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms
```

1. 将 `container6` 连接到 `local_alias` 网络中，并使用 `--network-alias <alias_name>` 给容器取一个网络别名。

```
$ docker network connect --alias scoped-app local_alias containe
r6
```

现在，容器所在的网络如下：

- `container4` : `isolated_nw` , `local_alias`
- `container5` : `isolated_nw`
- `container6` : `isolated_nw` / `app` , `local_alias` / `scoped-app`
- 现在分别进入 `container4` 和 `container5` 中 ping `container6` 的别名 `scoped-app` 。可以发现，别名只对在同一个网络(*local_alias*)中的 `containaenr4` 有效，而对 `container5` 无效。

```
$ docker attach container4

/ # ping -w 4 scoped-app
PING foo (172.26.0.5): 56 data bytes
64 bytes from 172.26.0.5: seq=0 ttl=64 time=0.070 ms
64 bytes from 172.26.0.5: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.26.0.5: seq=2 ttl=64 time=0.080 ms
64 bytes from 172.26.0.5: seq=3 ttl=64 time=0.097 ms

--- foo ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.081/0.097 ms

$ docker attach container5

/ # ping -w 4 scoped-app
ping: bad address 'scoped-app'
```

因此，`网络扩展别名 network-scoped alias` 生效的条件是：`容器` 、`别名` 必须处于同一个 `网络` 中。

RESOLVE MULTIPLE CONTAINERS TO A SINGLE ALIAS

多个容器可以在 `同一个` 网络中， `同时` 共享 `同一个` `网络扩展别名` 。因此，就是先了一个的 `DNS 轮询(round-robin)` 高可用。这个对某些缓存客户端 *ip* 的软件可能不适用，比如，*nginx*

> 注意：使用 `swarm` 替代这种 `dns 轮询` 高可用。
>
> > swarm 在外部提供了一个类似 `负载均衡器(load-balancing)` 的功能。你可以访问 swarm 的任何节点，即使该节点没有提供你所访问的 `服务 service` 。docker 也可以将请求转发到 `任何一个提供了该服务的节点上`

书接上文，举个例子

1. 创建容器 `container7` ，加入 `isolated_nw` 网络中，并设置网络扩展别名 `app` ,与 `container6` 相同。

```
$ docker run --network=isolated_nw -itd --name=container7 --netw
ork-alias app busybox
```

现在多个容器共享一个别名了，别名会被解析到 `其中一个` 容器。当其中一个容器不可用的时候，依旧还有另一个容器可以提供服务。

> 原文如下：
>
> > When multiple containers share the same alias, one of those containers will resolve to the alias. If that container is unavailable, another container with the alias will be resolved. This provides a sort of high availability within the cluster.
> >
> > 个人觉得有歧义 这个说法，感觉在两个容器同时存活的时候，这个别名『只会』解析到固定的一个容器上。当且仅当，这个容器挂掉之后，才会解析到第二个容器上。
> >
> > 这个不符合轮询的定义。也不符合实验结果

```
$  docker run --rm -it --network isolated_nw --name c4 busybox

/ # ping -c 3 app
PING app (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.054 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.159 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.153 ms

--- app ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.054/0.122/0.159 ms

/ # ping -c 3 app
PING app (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.050 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.163 ms
64 bytes from 172.20.0.3: seq=2 ttl=64 time=0.203 ms

--- app ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.050/0.138/0.203 ms
```

> 补充说明

北京-MD-蜗牛
你用的是 ping，你应该用 nslookup 或者 dig 去分析 dns

北京-MD-蜗牛 2017/9/20 8:36:47
另外，你应该使用 docker swarm 中的 service 的副本来做高可用。每个 service 会对应一个 VIP，这样没有 DNS 轮询的麻烦，如果客户端不重新解析，一样可以重定向到新的容器。

北京-MD-蜗牛
ping 并不是一个很好地用来测试 DNS 的工具，因为它有缓存。对于这种高可用的情况下，它不见得每次都会触发 DNS 请求。

北京-MD-蜗牛 2017/9/20 8:49:45
另外，ping 也不适合测试 docker 网络，因为有些情况，比如 IPVS，是4层交换网络，而 ping 只是3层，会导致测试出问题。

北京-MD-蜗牛 2017/9/20 8:50:46
所以一般是 nslookup 测试 dns 是否正常，然后用 curl 之类的进行4层网络的测试。ping 用一下可以，但是如果不通，不意味着 4 层一定不通。如果通，也不见得4层就通

Disconnect containers

使用命令 `docker network disconnect <network_name> <container_name>` 断开容器网络连接

```
$ docker network disconnect isolated_nw container2

$ docker inspect --format=''  container2 | python -m json.tool

{
    "bridge": {
        "NetworkID":"7ea29fc1412292a2d7bba362f9253545fecdfa8ce9a
6e37dd10ba8bee7129812",
        "EndpointID": "9e4575f7f61c0f9d69317b7a4b92eefc133347836
dd83ef65deffa16b9985dc0",
        "Gateway": "172.17.0.1",
        "GlobalIPv6Address": "",
```

```
        "GlobalIPv6PrefixLen": 0,
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:11:00:03"
    }
}


$ docker network inspect isolated_nw

[
    {
        "Name": "isolated_nw",
        "Id": "06a62f1c73c4e3107c0f555b7a5f163309827bfbbf9998401
66065a8f35455a8",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Containers": {
            "467a7863c3f0277ef8e661b38427737f28099b61fa55622d6c3
0fb288d88c551": {
                "Name": "container3",
                "EndpointID": "dffc7ec2915af58cc827d995e6ebdc897
342be0420123277103c40ae35579103",
                "MacAddress": "02:42:ac:19:03:03",
                "IPv4Address": "172.25.3.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {}
    }
```

```
]
```

当一个容器从一个网络中退出之后，就不能在通过 该网络 与其他网络中的容器通信了。

- 使用命令 `docker container stop <container_name>` 关闭容器
- 使用命令 `docker container rm <container_name>` 删除容器

Handle stale network endpoints

在某些情况下，例如，在多主机网络中，非正常重启 docker daemon 进程，可能会造成 daemon 不能完全清理 `失效(stale)` 的 `终端节点(endpoint)`。当网络中存在失效节点是，当一个新容器加入到该网络，且容器与失效节点名相同的名称时，会导致一下问题：

> 原文：
>
> > In some scenarios, such as ungraceful docker daemon restarts in a multi-host network, the daemon cannot clean up stale connectivity endpoints. Such stale endpoints may cause an error if a new container is connected to that network with the same name as the stale endpoint:

```
ERROR: Cannot start container bc0b19c089978f7845633027aa3435624c
a3d12dd4f4f764b61eac4c0610f32e: container already connected to n
etwork multihost
```

使用命令 `docker network disconnect -f <network_name> <container_name>` 强制将容器从网络中删除

```
$ docker run -d --name redis_db --network multihost redis

ERROR: Cannot start container bc0b19c089978f7845633027aa3435624c
a3d12dd4f4f764b61eac4c0610f32e: container already connected to n
etwork multihost

$ docker rm -f redis_db

$ docker network disconnect -f multihost redis_db

$ docker run -d --name redis_db --network multihost redis

7d986da974aeea5e9f7aca7e510bdb216d58682faa83a9040c2f2adc0544795a
```

Remove a network

当所有容器都退出网络之后，可以将网络删除。否则，会报错。

使用命令 `docker network rm <network_name>` 删除网络。

Manage swarm service networks

数据流量分类

Docker swarm 数据流量分为两个层面：

- `控制管理流量` (**control and management plane traffic**): 包括 swarm 管理消息，例如加入/退出 swarm 的请求。这些流量总是被加密的。
- `应用数据流量` (**Application data plane traffic**): 包括容器之间的数据交换，以及容器与外部网络的数据交换。

更多关于 swarm 网络的信息

swarm 的三个重要网络概念

swarm service 三个重要的网络概念:

- `Overlay network`：管理参数到 swarm 中的 Docker Daemon 之间的通信。
  - 可以添加 service 到 `一个或多个` `overlay` 网络中, 从而实现 `service to service` 的通信交互。
  - 使用 `overlay network driver` 创建的网络就是 `overlay` 网络
- `Ingress network`：是一个特殊的 `overlay` 网络，用于协调 service 节点之间的 `load balancing`
  - 当 `swarm node` 收到一个请求包时，会调用 `IPVS` 模块，经过 `ingress` 网络，将请求包路由到 `service node`
  - `ingress` 网络在 `init` 或 `join` swarm 的时候自动创建。`17.05` 及之后的版本允许用户自定义参数，但一般用不到。
- `docker_gwbridge`：是一个 `bridge` 网络，用于连接 `overlay` 网络(包括 `ingress` 网络) 到 `docker daemon` 的物理网络。
  - 默认情况下，每个跑服务的容器都是连接到本地 docker dameon 主机的 `docker_gwbridge` 网络上。
  - `docker_gwbridge` 网络在 `init` 或 `join` swarm 的时候自动创建。一般不用去管，但用户可以自定义参数。

Firewall considerations

swarm 中的 docker daemon 相互通信需通过以下端口：

- `7946` `TCP/UDP`：用于容器网络发现
- `4789` `UDP`：用于容器 overlay 网络

创建 overlay 网络

使用命令 `docker network create -d overlay` 创建

```
docker network create -d overlay my-network
```

使用命令 `docker network inspect` 查看网络信息

```
$ docker network inspect my-network
[
    {
        "Name": "my-network",
        "Id": "fsf1dmx3i9q75an49z36jycxd",
        "Created": "0001-01-01T00:00:00Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": []
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "Containers": null,
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "4
097"
        },
        "Labels": null
    }
]
```

- `driver` 是 `overlay`
- `scope` 是 `swarm`
  - 在 docker 的其他网络中，还可能是 `local`，`host`，`global`
  - `scope` 表示，只有在相同 `swarm` 中的主机才能访问该网络。
- 当 service `第一次` 加入到该网络中时，网络的子网和网关会自动被配置。

Customize an overlay network

使用命令 `docker network create --help` 查看更多配置参数。

配置子网和网关

You can configure these when creating a network using the `--subnet` and `--gateway` flags.

```
$ docker network create \
  --driver overlay \
  --subnet 10.0.9.0/24 \
  --gateway 10.0.9.99 \
  my-network
```

CONFIGURE ENCRYPTION OF APPLICATION DATA

Management and control plane data related to a swarm is always encrypted. Docker swarm mode overlay network security model.

Application data among swarm nodes is not encrypted by default. 在 `docker network create` 的时候使用 `--opt encrypted` 标识可以为 `overlay` 网络加密

- 在 `vxlan` 层允许 `IPSEC` 加密方式。这种加密对性能造成不可忽视的影响，因此您应该在生产过程中使用该选项之前测试此选项。

Attach a service to an overlay network

将 `service` 加入 `overlay` 网络中

- `docker service create --network <network_name>`
- `docker service update --network-add <network_name>`

```
$ docker service create \
  --replicas 3 \
  --name my-web \
  --network my-network \
  nginx
```

`service` 下的容器可以通过 `overlay` 网络中的容器进行通信。

查看 **service** 所加入的网络

- 使用命令 `docker service ls` 查看有哪些 `service`
- 使用命令 `docker service ps <service_name>` 查看 `service` 加入了哪些网络。

查看 **network** 有哪些 **service**

- `docker network inspect <network_name>` 可以看到当前网络下处于 `running` 的容器。

Configure service discovery

Service discovery is the mechanism Docker uses to `route` a request `from` your service's external clients `to` an individual swarm node。笼统的说就是包转发：

- `IPVS` 模式 / `Virtual IP (VIP)`
- `DNS` 轮询模式 / `DNS Round Robin (DNSRR)`

Customize the ingress network

Most users `never need` to configure the `ingress` network, but Docker 17.05 and higher allow you to do so.

- 解决子网冲突
- 修改 `MTU` 大小

自定义 `ingress` 参数意味着 `删除(removing)` 和 `重建(recreating)` 。

- 如果 `ingress` 关联了 `对外的 service (service which publish ports)` ，那首先要将 `对外服务` 全部删除，才能删除 `ingress` 。
- 在删除 `ingress` 之后，重建之前， `不对外的 service (services which do not publish ports)` 依旧正常运行，但无法进行负载均衡。
- 使用命令 `docker network inspect ingress` 查看 `ingress` 的信息，删除所有连接到 `ingress` 的 `service` 。否则之后操作会报错。
- 使用命令 `docker network rm ingress` 删除 `ingress` 网络。

```
$ docker network rm ingress

WARNING! Before removing the routing-mesh network, make sure all
 the nodes in your swarm run the same docker engine version. Oth
erwise, removal may not be effective and functionality of newly
create ingress networks will be impaired.
Are you sure you want to continue? [y/N]
```

1. 使用 `--ingress` 标识重建一个 `overlay` 网络，并指定需要使用的参数和值。例如设置
2. `MUT` : 1200，
3. 子网: `10.11.0.0/16` ，
4. 网关: `10.11.0.2`

```
$ docker network create \
  -d overlay \
  --ingress \
  --subnet=10.11.0.0/16 \
  --gateway=10.11.0.2 \
  --opt com.docker.network.mtu=1200 \
  my-ingress
```

> 注意： `ingress` 网络只能有一个，但是网络名可以设置为任何值。

1. 重启第一步停掉的服务。

Customize the docker_gwbridge

The `docker_gwbridge` is a `virtual bridge` that connects the `overlay networks` (**including the ingress network**) to an `individual` Docker daemon's `physical` network.

- `docker_gwbridge` 不是 `docker device`
- `docker_gwbridge` 存在于 `docker 主机` 的 `内核` 中

如果想要自定义 `docker_gwbridge` 的参数， `docker host 不能处于 swarm 中`

- 必须在 docker 主机加入 swarm 之前

- 或暂时将主机从 swarm 中删除

You need to have the `brctl` application installed on your operating system in order to delete an existing bridge. The package name is `bridge-utils`.

1. install `brctl`

```
# 如果不存在 brctl ，ubuntu 16.04 使用命令安装
sudo apt-get install bridge-utils
```

1. stop docker

```
$ sudo systemctl stop docker
Warning: Stopping docker.service, but it can still be activated
by:
  docker.socket
```

1. 使用 `brctl show <network_device_name>` 查看名为 `docker_gwbridge` 的 桥接设备(bridge device) 是否存在。如存在，则使用 `brctl delbr <network_device_name>` 删除

```
$ sudo brctl show docker_gwbridge
bridge name    bridge id        STP enabled    interfaces
docker_gwbridge        8000.0242b9cc96b2    no

# 网卡删除失败
$ sudo brctl delbr docker_gwbridge
bridge docker_gwbridge is still up; can not delete it

# 关闭网卡
$ sudo ifconfig docker_gwbridge down

# 重新删除
$ sudo brctl delbr docker_gwbridge
```

> 注意：删除 `docker_gwbridge` 网卡之前，需要先使用命令 `sudo ifocnfig docker_gwbridge down` 关闭它，否则会报错
>
> > https://unix.stackexchange.com/questions/62751/cannot-delete-bridge-bridge-br0-is-still-up-cant-delete-it

1. 启动 docker 但是不要 `join` 或 `init` swarm

```
$ sudo systemctl start docker
```

1. 使用自定义参数重新创建 `docker_gwbridge`
2. 更多自定义参数 bridge driver options

```
$ docker network create \
--subnet 10.11.0.0/16 \
--opt com.docker.network.bridge.name=docker_gwbridge \
--opt com.docker.network.bridge.enable_icc=false \
docker_gwbridge
```

> 注意：ubuntu 16.04 /docker 17.06 中，如果 `docker network ls` 中如果存在 `docker_gwbridge`，不使用 `docker network rm docker_gwbridge` 删除网络。即使按照之前的步骤，使用 `sudo btctl delbr <network_device_name>` 删除网卡设备，之后启动 docker 时已经会自动重建 `docker_gwbridge`。前后过程可以通过 `ip a|grep docker_gwbridge` 查看
>
> 注意：
>
> > 在 ubuntu16.04 / docker 17.06 中，实际操作应该如下

1. 不用安装 `bridge-utils` 也不同停止 `docker daemon`
2. 使用命令 `docker network rm docker_gwbridge` 删除网络，与此同时，系统对应的 `docker_gwbridge` 设备也会被删除。
3. 使用 `docker network create ...` 命令创建自定义网卡即可。

Use a separate interface for control and data traffic

By `default`, all swarm traffic is sent over the `same interface`, including `control and management` traffic for maintaining the swarm itself and `data traffic` to and from the service containers.

In Docker `17.06 and higher`, it is possible to `separate` this traffic by passing the `--datapath-addr` flag when `initializing` or `joining` the swarm.

多网卡时

- `--advertise-addr`：必须被指定
  - Traffic about `joining`, `leaving`, and `managing` the swarm will be sent over the `--advertise-addr interface`
- `--datapath-addr`：如果不指定，则与 `--advertise-addr` 同
  - traffic `among a service's containers` will be sent over the `--datapath-addr interface`
- `--advertise-addr` 和 `--datapath-addr` 的值，可以是
  - `ip 地址`：`192.168.1.1`，`--advertise-addr 192.168.1.1`
  - `接口名称`：`enp0s3`，`--datapath-addr enp0s3`

举个例子：

初始化 **swarm**

- `eth0` 做管理，`eth1` 做数据交换
- `eth0`：`10.0.0.1`
- `enp0s8`：`192.168.0.1`

```
$ docker swarm init --advertise-addr 10.0.2.15 --data-path-addr
enp0s8

Swarm initialized: current node (tejhf9eji9eaw3m268rjx94cw) is n
ow a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-token-string 10.0.2.15:23
77

To add a manager to this swarm, run 'docker swarm join-token man
ager' and follow the instructions.
```

加入 **swarm**

- swarm manager: `192.168.99.100:2377`
- `eth0` 做管理，`eth1` 做数据交换

```
$ docker swarm join \
  --token SWMTKN-1-token-string \
  --advertise-addr eth0 \
  --datapath-addr eth1 \
  192.168.99.100:2377
```

swarm join-token 与 join

join-token

可以通过命令 `docker swarm join-token` 查询 swarm-token

```
$ docker swarm join-token --help

Usage:    docker swarm join-token [OPTIONS] (worker|manager)

Manage join tokens

Options:
      --help     Print usage
  -q, --quiet    Only display token
      --rotate   更新token(Rotate join token)

# 查询 token
$ docker swarm join-token worker
$ docker swarm join-token manager
# 更新 token
$ docker swarm join-token --rotate worker
$ docker swarm join-token --rotate worker
```

join

使用 `docker swarm join --token <token_string> <manager ip:port>` 命令可以加入 swarm。

- 根据 `<token_string>` 的值不同，加入后的角色(`worker` / `manager`)也不同。
- `<token_string>` 由 `docker swarm join-token <worker|manager>` 命令生成

```
$ docker swarm join --help

Usage:    docker swarm join [OPTIONS] HOST:PORT

Join a swarm as a node and/or manager

Options:
      --advertise-addr string   Advertised address (format: <ip|
interface>[:port])
      --availability string     Availability of the node ("activ
e"|"pause"|"drain") (default "active")
      --data-path-addr string   Address or interface to use for
data path traffic (format: <ip|interface>)
      --help                    Print usage
      --listen-addr node-addr   Listen address (format: <ip|inte
rface>[:port]) (default 0.0.0.0:2377)
      --token string            Token for entry into the swarm
```

Multi-host networking with standalone swarms

第一代 swarm ，已经过期了。

很多地方不方便，不推荐使用了。

swarm mode overlay network security model

- Overlay networking for Docker Engine `swarm mode` `comes secure out of the box` .
- swarm nodes 交换 `overlay` 网络信息使用 `八卦协议(gossip protcol)` 。
- 在 `GCM` 模式下，节点使用 AES 算法对所交换的信息进行 `加密` 和 `解密` 。
- `Manager Node` 每 `12小时` 更换一次 `gossip` 的加密密钥。
- 在 `overlay` 网络中，你可以在不同 `node` 的容器上加密数据。
    - 创建网络时，使用 `--opt encrypted` 标识启用加密。

```
$ docker network create --opt encrypted --driver overlay my-mult
i-host-network

dt0zvqn0saezzinc8a5g4worx
```

- 启用 `overlay` 网络加密之后， docker 在根据任务调度，会在执行 service 任务的 `node` 上与 `overlay` 网络之间创建一个 `IPSEC` 通道。参考 (IPVS 的 TUN 模式)
    - 在 `GCM` 模式下，这些 `tunnel` 同样适用 `AES` 算法加密
    - `manager node` 每 `12小时` 更换一次 `key`

> 警告：不要将 **windows** 节点加入到 `加密` 后的 **overlay** 网络中 。
>
> > `overlay 网络加密` 不支持 windows。 如果尝试将 windows 节点加入加密后的 overlay 网络时， `不会被检测到` 但 `节点不能通信`

Swarm mode overlay networks and unmanaged containers

- 由于 `swarm mode` 的 `overlay` 网络中， `manager nodes` 使用 `加密密钥` 对 `gossip communication` 进行了加密，只有 swarm 中执行任务的的

容器才拥有该 `密钥` 。
    - 因此，在 swarm mode 外启动的容器(非托管容器) 不能加入到该 `overlay` 网络中。

举个例子:

```
$ docker run --network my-multi-host-network nginx

docker: Error response from daemon: swarm-scoped network
(my-multi-host-network) is not compatible with `docker create` o
r `docker
run`. This network can only be used by a docker service.
```

为了解决这个问题，可以将非托管容器迁移到托管服务中

举个例子：使用镜像 `my-image` 创建一个服务

```
$ docker service create --network my-multi-host-network my-image

# 查看当前服务
$ docker service ls
```

- 因为 `swarm mode` 是一个可选功能， Docker Engine 向后兼容。你可以继续使用第三方 `key-value store` 创建 `overlay` 网络。但强烈建议使用 `swarm mode` 。
- 除了本文中描述的安全性好处之外， `swarm mode` 还可以利用新的服务API提供的更大的可扩展性。

Embedded DNS server in user-defined networks

DNS lookup for containers connected to user-defined networks works `differently` compared to the containers connected to `default bridge` network.

> 注意：为了向后兼容，`default bridge` 网络的 DNS 配置没有改变。
>
> > 看这里 DNS in default bridge network 了解 `default bridge network` 的 DNS 配置

- 从 1.10 开始，Docker 提供了一个 `内置的 DNS 服务器`，因此可以为容器设置一个有效的 `name`，`network-alias` 或通过 `link` 指定别名。
- 在 `container` 内，Docker 如何管理 DNS 配置的具体细节可以从一个 Docker 版本更改为下一个。
- 因此，不要修改容器内的 `/etc/hosts`，`/etc/resolv.conf`。同时，使用以下命令进行管理：

options	note
`--name=<container_name>`	通过 `--name` 指定容器名，可以在 `user-defined` 网络中使用。 内置 DNS 服务器 维护 `container` 所连接的网络中， `container_name` 与 `container_ip` 之间的映射关系
`--network-alias=<alias_name>`	除了 `--name` 中描述的之外， `container` 在连接到 `user-defined network` 时，还可以指定在该网络中使用的 一个或多个 别名 `alias_name` 。 内置 DNS 服务器 维护网络中的 `alias_name` 与 `container_ip` 的映射关系。可以使用 `docker network connect --alias alias_name1 --alias alias_name2 network_name container_name` 将运行中的容器加入网络时配置多个别名
`--link=<container_name>:<alias_name>`	在使用 `docker run` 时，为 `container_name` 在 内置 DNS 中 指定一个额外的 `alias_name` 映射到 `container_ip` 上。只是当前使用 `--link` 的 container 才能通过 `alias_name` 对目标容器进行访问。这实现了 `容器1` 中的进程可以在不知道 `容器2` 的名称或IP的情况下连接到 `容器2` 。
`--dns=[IP_ADDRESS...]`	`--dns` 指定了当 内置 DNS 服务器 无法解析主机名时，所转发 DNS 请求的 目标服务器 `ip` 地址 。 `--dns` IP 地址由 内置 DNS 服务器 维护，但是不会更新容器内的 `/etc/resolv.conf` 文件
`--dns-search=DOMAIN...`	设置在容器内使用 裸的不合格的主机名(`bare unqualified hostname`) 时搜索的的域名。 `--dns-search` 选项由 内置 DNS 服务器 维护，但是不会更新容器内的 `/etc/resolv.conf` 文件。当一个进程尝试访问 `host` 且设置了域名 `example.com` ， DNS 会搜索 `host` 和 `host.example.com` 。
`--dns-opt=OPTION...`	Sets the options used by DNS resolvers. These options are managed by the embedded DNS server and will not be updated in the container's `/etc/resolv.conf` file. See documentation for `resolv.conf` for a list of valid options.

- 当没有指定 `--dns=<ipaddr>` , `--dns-search=<domain.com>` , `--dns-opt=OPTION...` 时， Docker 使用所在 宿主机 的 `/etc/resolv.conf` 。
    - 这种情况下， docker daemon 会过滤掉 宿主机上的 `resolv.conf` 文件中所有的 `localhost ip address` `nameserver` 入口(entry)
- 过滤是非常有必要的，因为主机的 localhost address 如法从容器中访问。

- 过滤只是，如果容器中的 `/etc/resolv.conf` 没有其他的 `nameserver` 入口存在，那么daemon会添加 `google dns nameserver (8.8.8.8 and 8.8.4.4)` ，如果启用了 ipv6 ，会添加 `(2001:4860:4860::8888 and 2001:4860:4860::8844)` 。

  > 注意：如果需要访问本机的 `DNS 服务器` ，必须将 DNS 进程服务监听在 `non-localhost` ip 上，这样容器才能访问

> 住址：容器中的 `/etc/resolv.conf` 中的 DNS 服务器永远是 `127.0.0.11` 。

Manage data in Docker

将数据放在容器的 `writable layer` 中有以下缺点：

- 数据无法 `持久化` 。当然一个容器关闭后，另一个容器不能再使用这些数据。
- `writable layer` 与宿主机 `紧耦合` 。不能方便的迁移到其他地方。
- 在 `writable layer` 中写数据需要 `storage driver` 管理文件系统。 `storage driver` 使用 `Linux kernel` 提供了一个 `union filesystem` 。与使用 `data volumes` 直接写入主机文件系统相比，这种方式会额外降低性能。

`Docker` 提供了 `3` 中不同的方式挂载挂载数据到容器中：

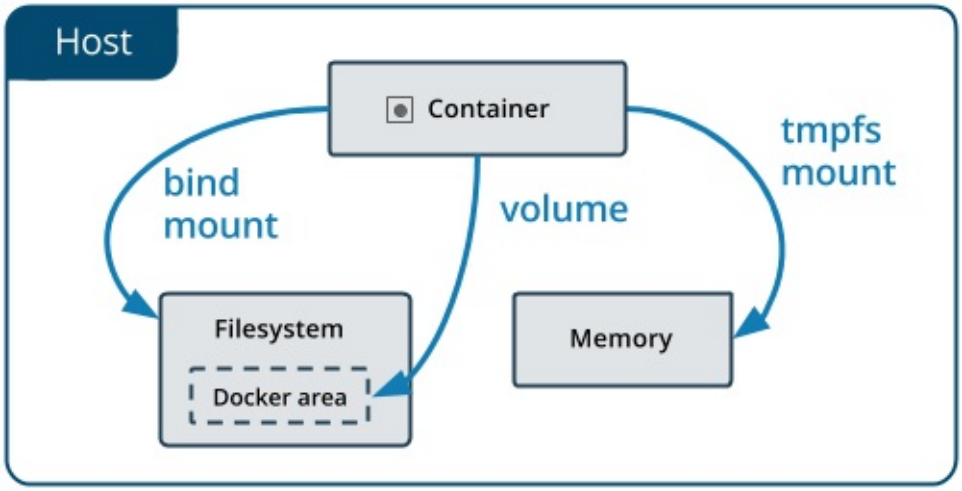
- `volumes`
- `bind mounts`
- `tmpfs`

当不能明确选择使用哪种方式的时候，就是用 `volumes` 。

Choose the right type of mount

无论选择哪种挂载方式，数据在容器中看起来都一样；要么是个文件夹，要么是一个独立的文件。

下图很好的说明了， `volumes` , `bind mounts` 和 `tmpfs` 三种方式在宿主机上存放数据的位置。



- **volumes**：将数据存放在由 docker 管理 的 文件系统 的一部分中 (linux 下为 /var/lib/docker/volumes/)。 Non-Docker 进程无法修改改文件系统。 **volumes** 是最佳的 持久化 数据的方案。
- **bind mounts**：数据存在于 宿主机 的 任何地方 。这些数据对于宿主机也可能是很重要的文件或目录。 Non-docker 进程在任何时间都可以修改这些文件。
- **tmpfs**：将数据存放在 宿主机 的 内存 中。并且 永远 不会落盘写入文件系统。

More details about mount types

Volumes

- 由 Docker 创建并管理。
  - 使用 docker volume create <volume_name> 创建
  - 或在创建 容器 或 服务 时附带创建。

```
$ docker volume create -d local test-vol
test-vol

$ ll /var/lib/docker/volumes/
drwxr-xr-x  3 root root  4096 Oct  3 14:44 test-vol/

$ docker volume create
8d360a1944d08e1ddb3e96981642504ca33b4d4d9c630733fd7c5c6eef7b3503
$ docker volume create
f2cf1aabeadc2568a2ca186722840b249bf89db2aac78c0812c2baf31d985a78
```

- 当创建创建一个 volume 的时候， Docker 在宿主机上相应位置创建一个 目录 。
  - 当 mount volume 到容器中时，实际就是 mount 该目录到容器中。
  - 与 bind mounts 类似，但 volume 有 Docker 管理，并与宿主机的核心功能 隔离 。
- 一个 volume 可以同时 mount 到多个容器。
  - 当没有容器挂载， volume 依旧可以通过 Docker 管理，并不会自动

删除。
    - 可以使用 `docker volume prune` 命令删除没用的 `volume` 。

```
$ docker volume prune
WARNING! This will remove all volumes not used by at least one c
ontainer.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
8d360a1944d08e1ddb3e96981642504ca33b4d4d9c630733fd7c5c6eef7b3503
f2cf1aabeadc2568a2ca186722840b249bf89db2aac78c0812c2baf31d985a78
test-vol
```

- 当挂载 `volume` 时， `volume` 可以被 `命名` 或 `匿名` 。
    - `匿名卷 (anonymous volumes)` 是指那些第一次挂载时，没有为其指定名称的卷，因此 Docker 为这些卷分配了一个随机名称，以此保证卷名的唯一性。
    - `命名卷 (name volumes)` 和 `匿名卷` 除了名称上的差别，实际行为都一样。
- `Volumes` 同样支持 `volumes drivers` ，因此允许将数据放在远程主机或云服务器上。

Bind mounts

- 与 `volumes` 相比， `bind mounts` 有一些功能限制。
- 挂载的时候，需要指定 `宿主机` 上的 `源路径` 必须是 `绝对路径` 。
- 挂载之前， `容器` 中的 `目标路径` 可以不存在。如不存在，挂载时会自动创建。
- `bind mounts` 效率可以很高，但是依赖宿主机具有特殊结构的文件系统。
- 发布 Docker 应用时，优先考虑 `volumes` 。
- `bind mounts` 不能通过 `Docker CLI` 直接进行管理。

> 警告: 使用 `bind mounts` 挂载的目录或文件可以在宿主机上直接进行修改，删除等操作。

tmpfs mounts

通过 `tmpfs` 挂载的数据，无论是在宿主机或者容器中都不会被持久化。可以用在容器的生命周期内，保存一些非持久状态或敏感信息。For instance, internally, swarm services use tmpfs mounts to mount secrets into a service's containers.

difference between three mount type

- `bind mount` 和 `volume mount` 都是使用 `-v / --volume` 进行挂载，但参数有点不同。
- `tmpfs` 使用 `--tmpfs` 。
- 在 `docker 17.06` 之后，建议以上三种类型都是用使用 `--mount` 进行挂载。

Good use cases for volumes

建议使用 `volume` 的场景包括：

- 当需要多个不同容器之间共享数据。
  - 如果不特别指定，容器启动时会自动创建一个 `volume`
  - 该容器删除后， `volume` 依旧存在。
  - 一个或多个其他容器可以挂载这个 `volume` , `rw/ro` 权限。
  - `volume` 在明确指出删除的情况下，才会被删除。
- 当宿主机不能提供 `bind mount` 挂载的 `目录或文件结构` 时。
  - `volume` 可帮助您将Docker主机的配置与容器运行时分离。
- 当需要在 `远程主机` 或 `云存储服务` 上保存数据时。
- 当需要将数据在宿主机之间 `备份` 、 `恢复` 和 `迁移` 时。
  - 首先，停止容器
  - 再次，备份 `volume` 目录（ex: `/var/lib/docker/volumes/<volume_name>` ）

Good use case for bind mounts

建议使用 `bind mount` 的场景包括：

- 在宿主机和容器之间共享 `配置文件` 。
  - 例如， Docker 本身的 DNS 解析。 `/etc/resolv.conf` 在宿主机和所有容器之间共享

- 在开发环境下宿主机的和容器之间共享 `代码` 和 `编译组件` 。
  - 使用 `bind mount` 快速挂载，而不是将代码 `COPY` build 到镜像中。
- 当宿主机的文件或目录结构保证与容器 `bind mount` 一致时。

Good use cases for tmpfs mount

建议使用 `bind mount` 的场景包括：

- 处于安全或性能考虑，不想数据被保存在宿主机硬盘或容器中。

Tips for using bind mounts or volumes

如果同时使用 `bind mount` 和 `volumes` ，需要注意一下几点：

- 在使用 `volume` 时，如果镜像中的挂载点目录原本就有文件，那么这些文件会复制到 `volume` 中。
  - This is a good way to pre-populate data that the Docker host needs (in the case of bind mounts) or that another container needs (in the case of volumes).
  - `bind mount` 不会， `docker-ce 17.09` 中测试

```
# 创建新 volume
$ docker volume create testvol
testvol

# 查看 volume 中的内容
$ tree /var/lib/docker/volumes/testvol/
/var/lib/docker/volumes/testvol/
└── _data

1 directory, 0 files

# 将 volume 挂载到 container 中， 挂载点中原来就有内容
$ docker run --rm -v testvol:/etc/nginx/ nginx ls /etc/nginx
conf.d
fastcgi_params
koi-utf
koi-win
```

```
mime.types
modules
nginx.conf
scgi_params
uwsgi_params
win-utf

# 再次查看挂载点，容器中的内容被复制到了 volume 中
$ tree /var/lib/docker/volumes/testvol/
/var/lib/docker/volumes/testvol/
└── _data
    ├── conf.d
    │   └── default.conf
    ├── fastcgi_params
    ├── koi-utf
    ├── koi-win
    ├── mime.types
    ├── modules -> /usr/lib/nginx/modules
    ├── nginx.conf
    ├── scgi_params
    ├── uwsgi_params
    └── win-utf

2 directories, 10 files
```

```
# 创建一个 bind mount 挂载源目录
$ mkdir -p testdir
$ ls testdir/

# 查看镜像中挂载点内容
$ docker run --rm nginx ls /etc/nginx
conf.d
fastcgi_params
koi-utf
koi-win
mime.types
modules
nginx.conf
scgi_params
uwsgi_params
win-utf

# 挂载时，查看挂载点无内容
$ docker run --rm -v /root/testdir/:/etc/nginx/ nginx ls /etc/ng
inx

# 挂载后，查看宿主机挂载路径无内容
$ ls testdir/
```

- 在使用 `bind mount` 或 `volume` 挂载时，如果 `源目录` 或 `volume` 中有数据，那么，容器中挂载点原本的数据将会被隐藏，类似 linux 下的 `mount` 。

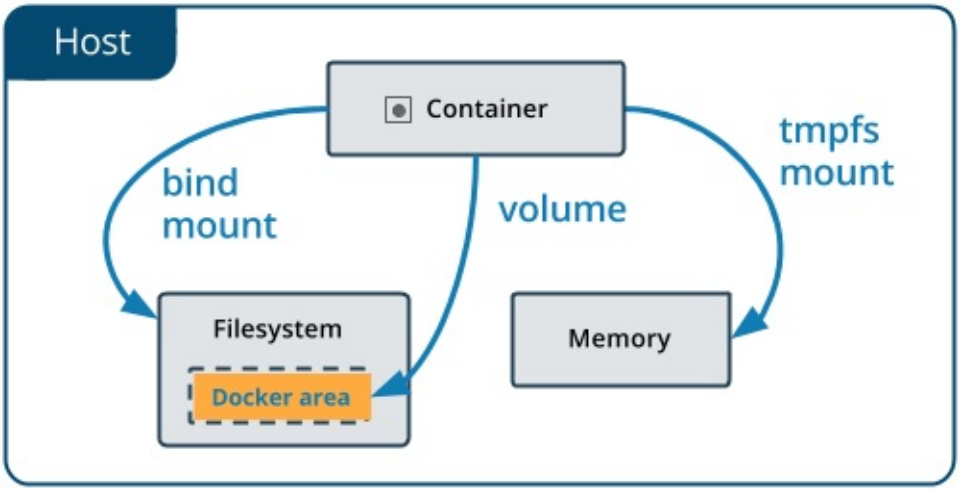
Use volumes

`Volumes` 在 docker 中更推荐使用。虽然 `bind mount` 也可以用，但是 `bind mount` 更依赖于宿主机的目录结构。此外 `volumes` 完全通过 docker 管理。

`volumes` 比 `bind mount` 有以下几个优势：

- `volumes` 更容易备份和迁移。
- `volumes` 可以通过 `docker CLI` 和 `API` 进行管理。
- `volumes` 可以在用于 `linux` 和 `windows` 下的容器。
- `volumes` 在多个容器间共享更安全。
- `volumes drivers` 允许你将 `volumes` 放在 `远程主机` 或 `云服务` 上，因此实现 `volumes` 内容加密或实现 `额外功能`。
- 新 `volumes` 的内容可以由容器 `预先填充(pre-populated)`。
    - 即，当 `volumes` 为空，而挂载目录不为空的时候，目录中的内容会被自动复制到 `volumes` 中。

此外，将容器持久化数据放入 `volumes` 中比放入 `writable layer` 中更好，这样不会增加容器占用磁盘空间大小。当容器删除后，`volumes` 依然存在。



如果容器会产生的数据不需要被持久化，建议使用 `tmpfs mount`

- 以避免这些数据在某些地方被保存
- 替代使用容器的 `writable layer` 从而提高效率。

Volumes use `rprivate` bind propagation, and bind propagation is not configurable for volumes.

使用 `-v` 还是 `--mount`

- 最开始，`-v` 或 `--volume` 用于 `standalone` 容器，而 `--mount` 用于 `swarm services`；但从 `17.06` 开始 `--mount` 也可以用于 `standalone` 容器了。
- `--mount` 可读性更高，意义更明确。
  - `-v` 将所有参数组合成一个；而 `--mount` 将他们分开
- 如果主要指定 `volume driver`，必须使用 `--mount`

> 建议：新人用 `--mount`；老鸟可能更熟悉 `-v / --volume`，但建议使用 `--mount`

- `-v / --volume`：由 `3` 部分组成，用 `冒号 :` 分隔。每部分的排序必须正确。`volume_name:mount_point:flag`
  - `volume_name`：第一部分，指定被挂载的卷名。如果是 `命名卷` 这部分就是 `卷名称`，如果是 `匿名卷` 这部分省略。
  - `mount_point`：第二部分，指定 `volume` 在容器中的挂载路径。
  - `flag`：第三部分，可选。如果有多个字段，以 `逗号 ,` 分隔。字段将在下面进行讨论。
- `--mount`：由多个 `key-value` 对组成；以 `逗号 ,` 分隔；每个字段格式为 `<key>=<value>`。虽然 `--mount` 比 `-v` 更冗长，但是每个字段意义更明确，而且字段之间没有顺序限制。
  - `type`：指定挂载类型，值为 `bind`，`volume`，`tmpfs`。此处为 `volume`。
  - `source / src`：被挂载的卷名。如果为 `命名卷` 则为卷名。如果省略，则为 `匿名卷`。
  - `destination / dst / target`：指定 `volume` 在容器中的挂载路径。
  - `readonly`：如果存在，则所挂载的 `volume` 为只读。
  - `volume-opt`：可以出现多次，使用 `key-value` 对指定 `option name` 和 `option value`。

`-v` 与 `--mount` 的行为的异同

- 与 `bind mount` 不同，`volume` 的所有参数都可以同时用于 `--mount` 和 `-v`。

- 当 `volumes` 用于 `services` 时，只能使用 `--mount` 。

创建与管理 volumes

与 `bind mount` 不同，可以在容器之外 `创建` 和 `管理` volume。

创建 volume

```
$ docker volume create my-vol
```

查看存在 volume

```
$ docker volume ls

local               my-vol
```

查看 volume 详细信息

```
$ docker volume inspect my-vol
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

删除 volume

```
$ docker volume rm my-vol
```

Start a container with a volume

在 `run` 一个容器时，如果 `volume` 不存在，则会自动被创建。下例中 `myvol2` 被挂载到容器中的 `/app/` 目录下。

下面两种写法是等价的 **`--mount`**

```
$ docker run -d \
  -it  \
  --name devtest  \
  --mount source=myvol2,target=/app  \
  nginx:latest
```

`--volume`

```
$ docker run -d \
  -it \
  --name devtest
  -v myvol2:/app  \
  nginx:latest
```

使用命令 `docker container inspect devtest` 验证 volume 已经被创建并且正确挂载。查看 `Mounts` 部分：

```
"Mounts": [
    {
        "Type": "volume",
        "Name": "myvol2",
        "Source": "/var/lib/docker/volumes/myvol2/_data",
        "Destination": "/app",
        "Driver": "local",
        "Mode": "",
        "RW": true,
        "Propagation": ""
    }
],
```

结果表明了，挂载类型为 `volume` ， 卷和挂载点都正确，且挂载权限为 `read-write` 。

停止容器并删除卷

```
$ docker container stop devtest

$ docker container rm devtest

$ docker volume rm myvol2
```

start a service with volumes

当启动一个 `serice` 并定一个 `volume` 时， 每个容器会使用各自的 `local volume` 。 当使用 `local` `volume driver` 时，所有容器都不能共享数据；但其他 `volume driver` 可以支持共享存储。 Docker for AWS 和 Docker for Azure 都支持使用 `Cloudstor plugin` 保存数据。

下例中， 创建了一个 4 个 nginx 容器实例的 `service` , 每个容器都使用各自的 `local volume` ，名称都为 `myvol2`

```
$ docker swarm init

$ docker service create -d  \
  --name devtest-service  \
  --mount source=myvol2,target=app  \
  --replicas 4  \
  nginx:latest
```

使用命令 `docker service ps devtest-service` 验证 `service` 正常启动

```
$ docker service ps devtest-service
ID                  NAME                IMAGE               NODE
                DESIRED STATE       CURRENT STATE           ERRO
R               PORTS
11ckavzyee76        devtest-service.1   nginx:latest        inst
ance-4          Running             Running 2 minutes ago

o7fmb0363h75        devtest-service.2   nginx:latest        inst
ance-4          Running             Running 2 minutes ago

ftiz0l91v9wb        devtest-service.3   nginx:latest        inst
ance-4          Running             Running 2 minutes ago

cbnd9h2hczqv        devtest-service.4   nginx:latest        inst
ance-4          Running             Running 2 minutes ago
```

删除 service

```
$ docker service rm devtest-service
```

syntax differences for service

命令 `docker service create` 不支持使用 `-v` 或 `--volume`。当要挂载 `volume` 时，必须使用 `--mount`。

Populate a volume using a container

- 当挂载一个 `空 volume` 到容器，且容器中挂载点中 `有` 文件时
  - Docker 会将 挂载点 中的文件复制到 `volume` 中
  - 容器挂载该 `volume`
  - 容器使用 `volume` 中的文件
  - 其他容器有权访问 `volume` 中的文件。

为了说明这个现象，下例中启动了一个 `nginx` 容器，并将 `空 volume` `nginx-vol` 挂载到容器中的 `/usr/share/nginx/html` 目录上，且该目录中那个有默认的 HTML 文件。

--mount

```
$ docker run -d  \
  -it  \
  --name=nginxtest  \
  --mount source=nginx-vol,dst=/usr/share/nginx/html  \
  nginx:latest
```

--volume

```
$ docker run -d  \
  -it  \
  --name=nginxtest
  -v nginx-vol:/usr/share/nginx/html  \
  nginx:latest
```

当通过上面的命令创建容器之后，下面的几条命令可以删除容器与卷：

```

$ docker container stop nginxtest

$ docker container stop nginxtest

$ docker volume rm nginx-vol
```

使用只读卷

某些情况下，你可能希望挂载的卷为 `只读` 状态，这个时候，

- `-v / --volume` : `ro` , 以 `冒号 :` 分割
- `--mount` : `readonly` ， 以 `逗号 ,` 分割

例如

--mount

```
$ docker run -d  \
  -it  \
  --name=nginxtest  \
  --mount src=nginx-vol,dst=/usr/share/nginx/html,readonly  \
  nginx:latest
```

`-v / --volume`

```
$ docker run -d  \
  -it  \
  --name=nginxtest  \
  -v nginx-vol:/usr/share/nginx/html:ro  \
  nginx:latest
```

使用命令 `docker container inspect nginxtest` 可以看到 volume 已经正常被挂载。查看 `Mounts` 部分：

```
"Mounts": [
    {
        "Type": "volume",
        "Name": "nginx-vol",
        "Source": "/var/lib/docker/volumes/nginx-vol/_data",
        "Destination": "/usr/share/nginx/html",
        "Driver": "local",
        "Mode": "",
        "RW": false,
        "Propagation": ""
    }
],
```

停止并移除容器，删除卷命令

```
$ docker container stop nginxtest

$ docker container rm nginxtest

$ docker volume rm nginx-vol
```

Usa a volume driver

当使用命令 `docker volume create` 创建卷 或当启动容器时使用一个 `没有被创建的卷` 时，可以为新创建的卷指定 `volume driver` 。

下例中使用 `vieux/sshfs` volume driver 。

Inital set-up

假设有两个节点，第一个节点为 Docker 宿主机，并可以通过 `ssh` 连接到第二个节点。

在 Docker 宿主机上，安装 `vieux/sshfs` 插件：

```
$ docker plugin install --grant-all-permissions vieux/sshfs
```

Create a volume using a volume driver

下例中使用了 SSH 密码进行授权，如果两个节点之间使用 `密钥` 进行授权的话，密码部分可以省略。

不同的 `volume dirver` 可能有 `0 个或多个` 配置选项，每个选项配置时使用 `-o` 指定。

```
$ docker volume create --driver vieux/sshfs  \
  -o sshcmd=test@node2:/home/test  \
  -o password=testpassword  \
  sshvolume
```

start a container which creates a volume using a volume driver

下例中使用了 SSH 密码进行授权，如果两个节点之间使用 `密钥` 进行授权的话，密码部分可以省略。

不同的 `volume dirver` 可能有 `0 个或多个` 配置选项，每个选项配置时使用 `-o` 指定。

> 如果所使用的 `volume driver` 需要配置选项，那么创建时必须使用 `--mount` ，而不是 `-v / --volume` 。

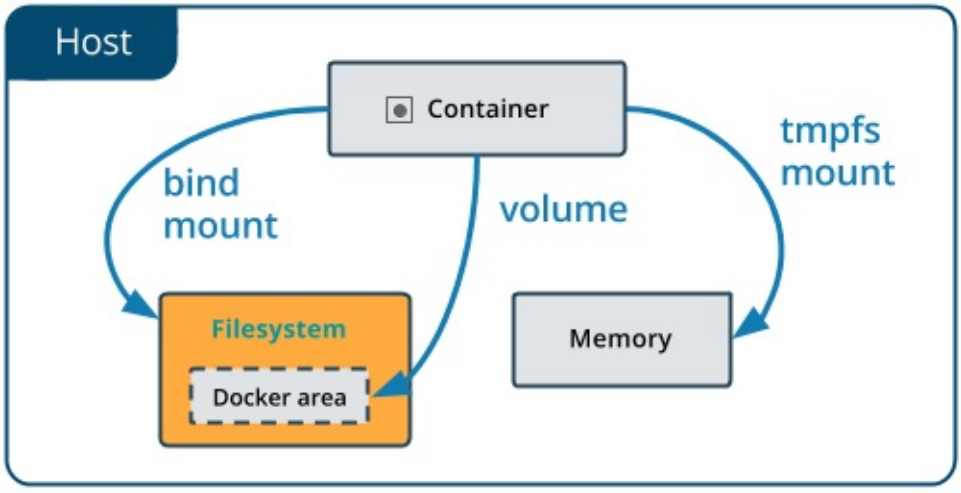
```
$ docker run -d  \
  -it  \
  --name sshfs-container  \
  --volume-driver vieux/sshfs  \
  --mount src=sshvolume,dst=/app,volume-opt=sshcmd=test@node2:/h
ome/test,volume-opt=password=testpassword  \
  nginx:latest
```

Use bind mounts

- `bind mounts` 在 docker 早期版本就已经在使用了。
- `bind mounts` 较 `volumes` 而言，会有一些功能限制。
- 当使用 `bind mounts` 时， 宿主机中的 `已存在的文件或目录` 被挂载到容器中。
- 在挂载时，使用 `绝对路径` 或 `相对路径` 指定所挂载的文件或目录。

```
$ docker run --rm -v python3:/root/python3/ --name nginx_test -d
  nginx
bfd85451aa6e96e304341c8c80ac16af64482d793a2b00e866f0bae06ec2e83f
```

- 与 `bind mounts` 不同，当创建一个 `volume` 时，会在宿主机的 `Docker's storage directory` 创建一个新文件夹，并有 docker 管理其内容。
- 容器中的挂载点在不需要预先存在。
  - 如果挂载点不存在，则会自动创建。
  - 如果挂载点存在且不会空，那么挂载点中的内容将被隐藏。（参考 linux mount）
- `bind mount` 在研发阶段非常有效，但它们依赖于具有特定目录结构的主机的文件系统。
- 如果要发布一个 docker 应用，应该使用 `命名卷` .
- docker CLI 命令不能直接作用于 `bind mounts` 。



用 `-v` 还是 `--mount`

- 最开始，`-v` 或 `--volume` 用于 `standalone` 容器，而 `--mount` 用于 `swarm services`；但从 `17.06` 开始 `--mount` 也可以用于 `standalone` 容器了。
- `--mount` 可读性更高，意义更明确。
  - `-v` 将所有参数组合成一个；而 `--mount` 将他们分开

> 建议：新人用 `--mount`；老鸟可能更熟悉 `-v / --volume`，但建议使用 `--mount`

- `-v / --volume`：有三个部分组成，以冒号 `:` 分隔。`-v local_path:mount_point:flag`
  - local_path：宿主机文件或目录的 `绝对路径或相对路径`。
  - mount_point：容器中的挂载点。
  - flag: 可选部分。如果有个多，使用逗号 `,` 分隔。例如 `ro`，`consistent`，`delegated`，`cached`，`z`，`Z`。
- `--mount`：以多个 `<key>=<value>` 对组成，使用逗号 `,` 分隔。虽然 `--mount` 比 `-v` 更冗长，但是每个字段意义更明确，而且字段之间没有顺序限制。
  - `type`：挂载类型 `bind`，`volume`，`tmpfs`，此处为 `bind`。
  - `source / src`：宿主机文件或目录的 `绝对路径`。
  - `destination / dst/ target`：容器中的挂载点
  - `readonly`：只读挂载
  - `bind-propagation`：如果存在，则修改 bind 的传播方式。可选值为 `rprivate`，`private`，`rshared`，`shared`，`rslave`，`slave`。
  - `consistency`：如果存在，可选值为 `consistent`，`delegated`，`cached`。该选项仅对 `Docker for Mac` 有效，其他平台将会被忽略。
  - `--mount` 不支持 `z` 或 `Z` 修改 selinux 标签。

`--mount` 的 **`src`** 不支持相对路径:

```
$ docker run -d --rm --name nginx_test --mount type=bind,src=pyt
hon3,dst=/root/python3 nginx
docker: Error response from daemon: invalid mount config for type
 "bind": invalid mount path: 'python3' mount path must be absolu
te.
See 'docker run --help'.

$ docker run -d --rm --name nginx_test --mount type=bind,src=/ho
me/python3,dst=/root/python3 nginx
880c35d9d2cfb01db921030451ff2a4f0b6da7baf2b1f5b8ce3fe4da2985ed8e
```

Differences between -v and --mount behavior

Because the `-v` and `--volume` flags have been a part of Docker for a long time, their behavior cannot be changed. This means that there is one behavior that is different between `-v` and `--mount` .

- 使用 `-v / --volume` 挂载时，如果 `source` 不存在，docker 会在指定路径创建一个 `目录` 作为 `source` 。
- 使用 `--mount` 挂载是，如果 `source` 不存在，docker 不会创建任何东西，并报错。

Start a container with a bind mount

`--mount` :

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app \
  nginx:latest
```

`-v` :

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app \
  nginx:latest
```

使用命令 `docker container inspect devtest` 验证挂载是否正确，并查看 `Mounts` 部分：

```
"Mounts": [
    {
        "Type": "bind",
        "Source": "/tmp/source/target",
        "Destination": "/app",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
    }
],
```

结果显示，挂载方式为 `bind`，`source` 和 `Destination` 位置正确，挂载方式为 `read-write`，传播方式为 `rprivate` 。

关闭容器：

```
$ docker container stop devtest

$ docker container rm devtest
```

Mounting into a non-empty directory on the container

使用 `bind mounts` 挂载本地目录到容器中的一个非空目录时，容器中的目录内容将会被隐藏。

这个非常适合用于研发阶段，经常变动。

下例非常极端，把宿主机的中的 `/tmp` 目录挂载到了容器中的 `/usr` 目录。

`--mount`：

```
$ docker run -d \
  -it \
  --name broken-container \
  --mount type=bind,source=/tmp,target=/usr \
  nginx:latest

docker: Error response from daemon: oci runtime error: container
_linux.go:262:
starting container process caused "exec: \"nginx\": executable f
ile not found in $PATH".
```

`-v`：

```
$ docker run -d \
  -it \
  --name broken-container \
  -v /tmp:/usr \
  nginx:latest

docker: Error response from daemon: oci runtime error: container
_linux.go:262:
starting container process caused "exec: \"nginx\": executable f
ile not found in $PATH".
```

上例中，容器创建了，但是没有被启动。使用命令删除

```
$ docker container rm broken-container
```

Use a read-only bind mount

当需要被挂在的目录为 `只读` 时：

- `-v`：`ro`

- `--mount`：`readonly`

`--mount`：

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app,readonly
\
  nginx:latest
```

`-v`：

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:ro \
  nginx:latest
```

使用命令 `docker container inspect devtest` 查看容器信息，并查看 `Mounts` 部分

```
"Mounts": [
    {
        "Type": "bind",
        "Source": "/tmp/source/target",
        "Destination": "/app",
        "Mode": "ro",
        "RW": false,
        "Propagation": "rprivate"
    }
],
```

关闭容器

```
$ docker container stop devtest

$ docker container rm devtest
```

Configure bind propagation

- 在 `bind mount` 和 `volumes` 中，`Bind propagation` 默认为 `rprivate` 。
- 只有 `linux` 上能为 `bind mounts` 配置 `bind-propagation` 的值。
- `bind propagation` 是一个高阶功能，大部分人不会配置到这部分。

> 更多信息，直接看官方文档
> https://docs.docker.com/engine/admin/volumes/bind-mounts/#configure-bind-propagation

Configure the selinux label

如果使用 `selinux` ，你可以使用 `z` 或 `Z` 来修改 mount 到容器中的主机文件或目录的 selinux 标签。并且可能会在 Docker 的范围之外产生后果。

- `z` ：`bind mount` 的内容可以在多个容器之间共享
- `Z` ：`bind mount` 的内容是 `私有的` ，不能被共享。

在 极端情况下，如果挂载宿主机的 `/home` 或 `/usr/` 到容器中，并使用了 `Z` ，那么会导致宿主机无法操作，且你需要手动 `relabel` 这些宿主机文件。

> `z` 或 `Z` 不能搭配 `--mount` 使用

This example sets the `z` option to specify that multiple containers can share the bind mount's contents:

```
$ docker run -d \
  -it \
  --name devtest \
  -v "$(pwd)"/target:/app:z \
  nginx:latest
```

Configure mount consistency for macOS

> 只作用于 MacOS ，自己看官网
> https://docs.docker.com/engine/admin/volumes/bind-mounts/#configure-mount-consistency-for-macos

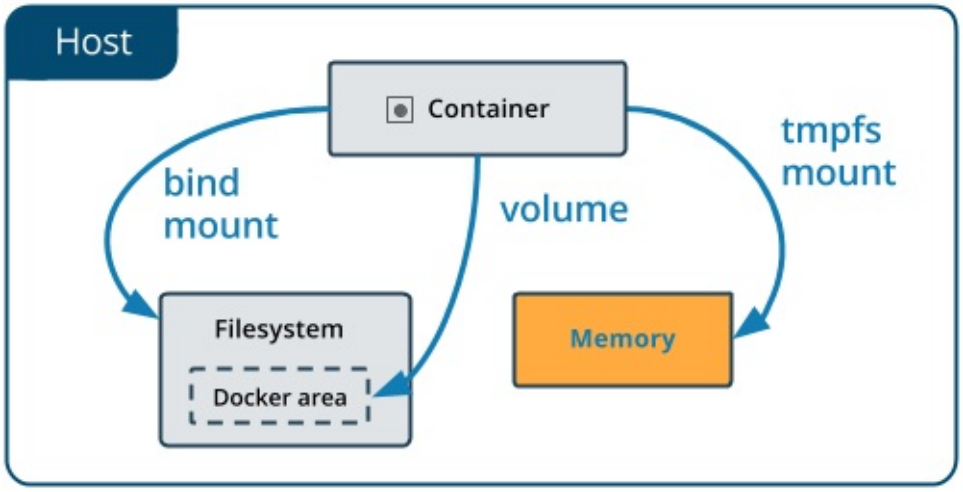
Use tmpfs mounts

`volume` 和 `bind mounts` 都是挂载到容器的文件系统中，且内容保存在宿主机上。

在某些情况下，你可能不希望将容器数据放在宿主机上，也不想放在容器的 `writable layer` 中，或者处于性能或安全考虑，或者数据涉及非持久性应用程序状态。

使用 `tmpfs` 挂载，将数据写入 `内存` 中（内存小时，写入 `swap` 中）。

- 当容器停止后， `tmpfs` mount 被删除。
- 当容器被 commit 是， `tmpfs` mount 不会被保存。



选 `--tmpfs` 还是 `--mount`

- 最开始， `--tmpfs` 只用于 standalone 容器， 而 `--mount` 用于 `swarm services` 。 然而从 `17.06` 开始， `--mount` 也可以用于 standalone 容器了。
- `--mount` 可读性更高，意义更明确。
  - `-v` 将所有参数组合成一个；而 `--mount` 将他们分开

> 建议： 新人用 `--mount` ；老鸟可能更熟悉 `-v / --volume` ，但建议使用 `--mount`

- `--tmpfs`：挂载 `tmpfs mount`，但不能指定任何配置，只能用于 standalone 容器。
- `--mount`：由多个 `<key>=<value>` 组成。
  - `type`：指定类型；可选值为 `bind`，`volume`，`tmpfs`。此处为 `tmpfs`。
  - `destination / dst / target`：指定挂载点。
  - `tmpfs-type` 和 `tmpfs-mode`：看下面的介绍

Differences between `--tmpfs` and `--mount` behavior

- `--tmpfs` 不允许指定任何配置。
- `--tmpfs` 不能用于 `swarm services`。必须使用 `--mount`。

Limitations of tmpfs containers

- `tmpfs` mounts 不能再容器间共享。
- `tmpfs` mounts 只能作用于 linux 容器，不支持 windows 容器。

Use a tmpfs mount in a container

容器中使用 `tmpfs` mounts 时，可以使用 `--tmpfs` 或 `--mount type=tmpfs`。`tmpfs` 没有 `source` 选项。

下例中为 nginx 容器创建了一个 `tmpfs` mount 并挂载到了 `/app` 目录。

`--mount`：

```
$ docker run -d  \
  -it  \
  --name tmptest  \
  --mount type=tmpfs,dst=/app  \
  nginx:latest
```

`--tmpfs`：

```
$ docker run -d  \
  -it  \
  --tmpfs /app  \
  nginx:latest
```

使用命令 `docker container inspect tmptest` 查看容器信息，并查看 `Mounts` 部分。

```
"Tmpfs": {
    "/app": ""
},
```

删除容器

```
$ docker container stop tmptest

$ docker container rm tmptest
```

Specify tmpfs options

`tmpfs` mounts 有两个配置选项；两个选项都不是必须的。如果要使用这两个选项，必须使用 `--mount` 而不是 `--tmpfs` 。

Option	Description
`tmpfs-size`	设置 `tmpfs mounts` 的大小，单位 `bytes` 。默认 `无限制`
`tmpfs-mode`	设置 tmpfs 的 `8 进制 rwx 权限。例如 700 或 0770 。默认为 1777 或 world-writable 。`

下例中设置 `tmpfs-mode` 为 `1770` 。因此在容器中，非 `world-readable` 。

```
$ docker run -d  \
  -it  \
  --name tmptest  \
  --mount type=tmpfs,dst=/app,tmpfs-mode=1770  \
  nginx:latest
```

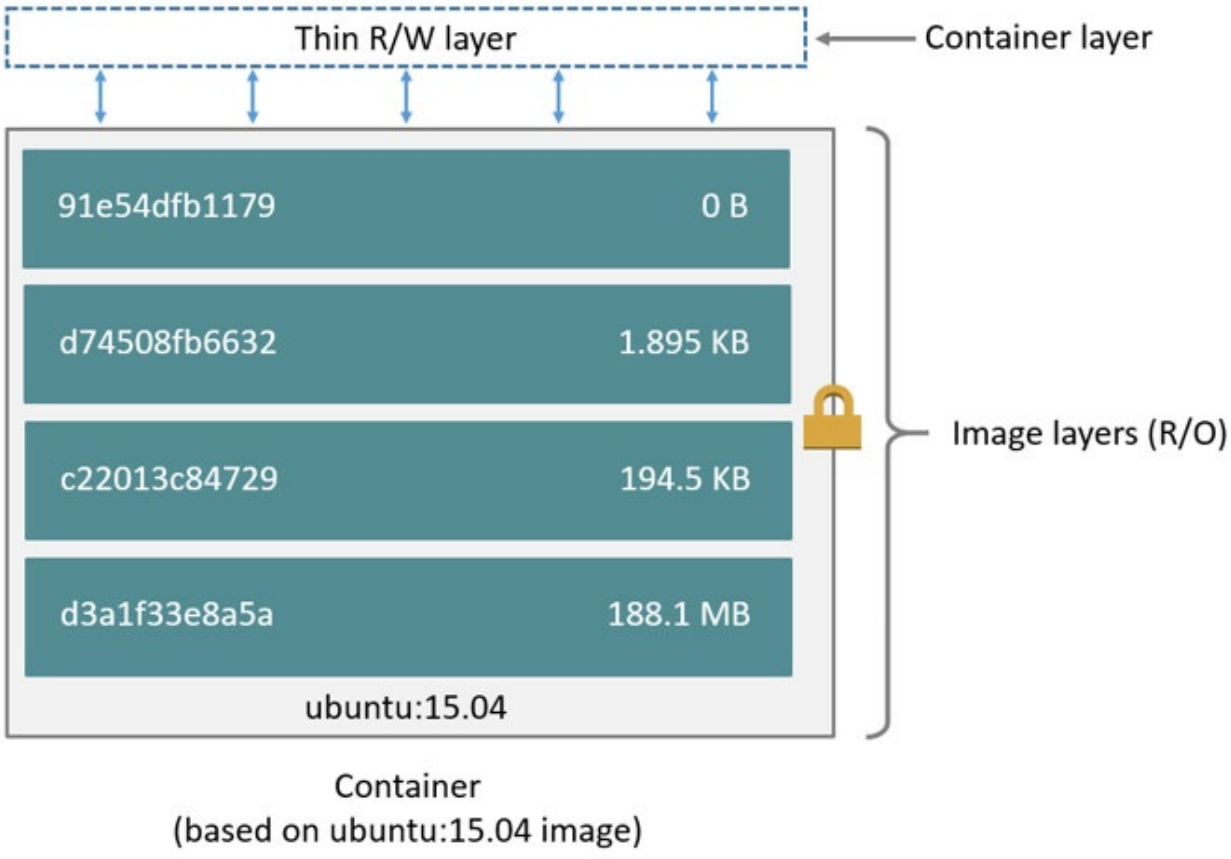
镜像、容器与存储

> https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers

镜像与层

- 一个镜像包含多个 `层 (layer)` 堆叠 `(stack)` 而成
- 每一层代表一个 `dockerfile` 指令
- 除了最后一层，其他都为 `只读 (read-only)`
- 每个 `layer` 都与之前的不一样
- 当创建 `container` 时，会在顶端新建一个 `可写层(writable layer)`，被称为 `容器层 (container layer)`
- 所有文件 `更改` 都发生在 `容器层`

下图展示了 `ubuntu 15.04` 镜像的层级关系

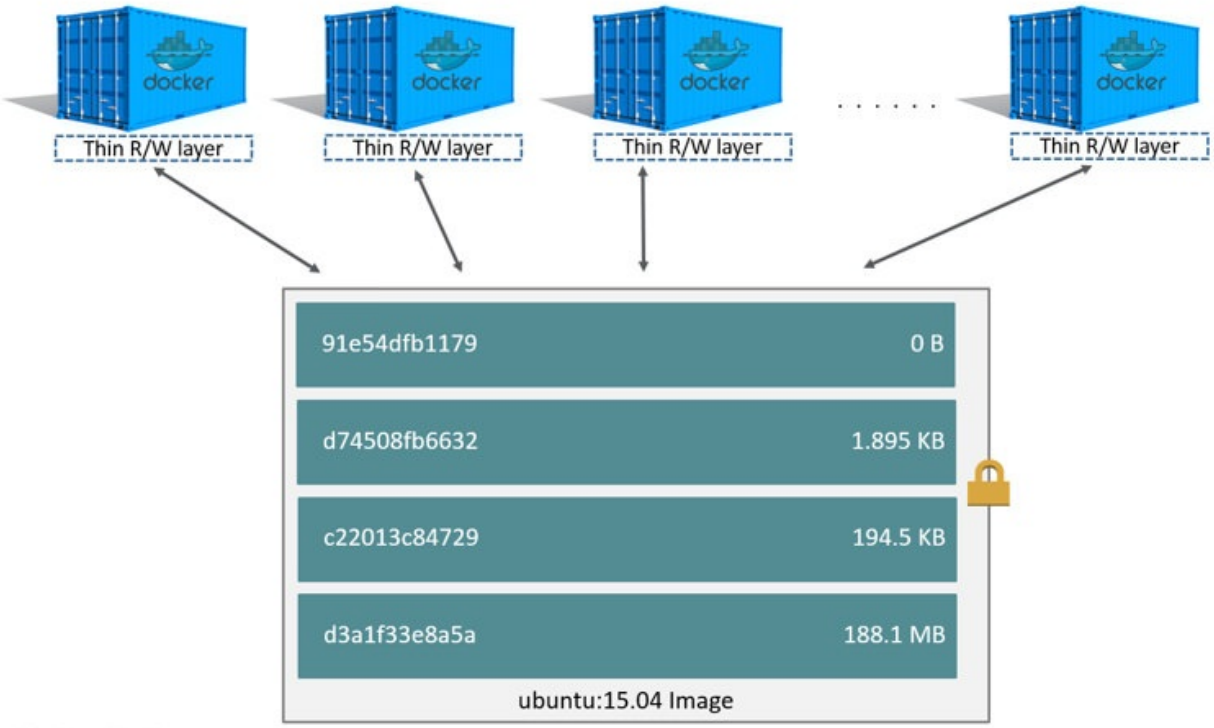


- `存储驱动 (storage driver)` 管理这些层之间的关系。
- 不同的 存储驱动 有各自不同的特性。

容器与层

- 多个 `container` 可以共享一个 `image`
- 每个 `container` 在 `image` 的基础上创建一个属于各自的 `writable layer`
- 所有文件变动都 `只会` 发生在 `writable layer` 上
- 任何文件变动都 `不会` 影响到 `image`
- `container` 在被删除的时候，相应的 `writable layer` 也被删除

下图展示了 `ubuntu 15.04` 镜像创建多个 `容器` 之间的关系



> 注意：如果你有 `多个` 镜像需要 `共享访问` `相同的数据` ，那么需要将这些数据放在 `docker volume` 中，并 `mount` 到你的容器中。

容器在磁盘上的大小

- 使用命令 `docker container ps -s` 查看运行中容器的大小
- `SIZE`：当前容器的 `writable layer` 大小
- `virtual size`：容器使用的 `read-only` 镜像大小
  - 还可以通过 `docker image ls` 查看
  - 不同的容器可能使用相同的镜像，因此统计 `virtual size` 是不能简单

的相加

```
$ docker container ps -s

CONTAINER ID        IMAGE                 COMMAND
     CREATED             STATUS              PORTS
          NAMES                       SIZE
714135f3cac4        registry:2            "/entrypoint.sh /e..
."   28 hours ago        Up 28 hours         0.0.0.0:5000->5
000/tcp   registryproxy_mirror_1   0B (virtual 33.2MB)
```

- 容器使用的总容量为：所有容器的 `size` 与一个 `virtual size` 的和。
  - 这种说法并不包含以下几点：
  - 使用 `json-file` logging drive 的日志文件大小。
  - 容器使用的 `volume`
  - 容器的配置，通常很小
  - Memory written to disk (if swapping is enabled)
  - checkpoint, if you're useing the experimental checkpoint/restore feature.

copy-on-write(CoW) 策略

- `CoW` 策略是一种高效的 `sharing and copying` 策略。
- 尽可能的降低了 `I/O` 消耗
- 当一个文件在低层 `layer` 中存在时，可以直接访问并读取。
- 当一个文件 `第一次` 被修改时，先复制到当前 `writable layer`，然后在修改

sharing promotes smaller images

使用命令 `docker pull <image>` 的时候

- `image` 的每一次独立下载
- `Linux` 系统存放在 `/var/lib/docker/` 目录下
  - `/var/lib/docker/<storage-driver>/layers/`
- 使用 `docker history <image>` 查看镜像的 `build` 过程
  - 输出中含有 `<missing>` 的 `layer` 表示：这些 `layer` 在其他系统上

`build` 并且在本地不可用。可以被忽略

Copying makes containers efficient

当发生 `copy-on-write` 操作时，具体步骤和具体的 `storage driver` 有关。

默认的 `aufs` driver 以及 `overlay`，`overlay2` drivers 的 `CoW` 步骤如下：

- 首先，搜索 `image layer` 中是否包含此文件。从 `最上层` 开始，到 `最下层` 为止。当找到后，添加到 `cache` 中等待下一步操作。
- 随后，`copy_up` 操作将找到的文件复制到容器的 `writable layer`。
- 最后，针对此文件的所有操作，容器都不会再次进行搜索底层 `layer` 了。

`Btrfs`，`ZFS` 和其他 storage dirver 的 `CoW` 实现方式不同。

- 有 `write` 操作的容器会消耗更多的资源，因为 `write` 操作的结果会保存在 `writable layer` 中。

> 注意：对于 `write-heavy` 应用，不应该将 `data` 保存在容器中。而应该使用 `Docker volume`。

A `copy_up` operation can incur a `noticeable` performance `overhead`. This `overhead` is `different depending on` which `storage driver` is in use. `Large files`, `lots of layers`, and `deep directory trees` can make the impact more `noticeable`. This is mitigated by the fact that each `copy_up` operation only occurs the `first time` a given file is modified.

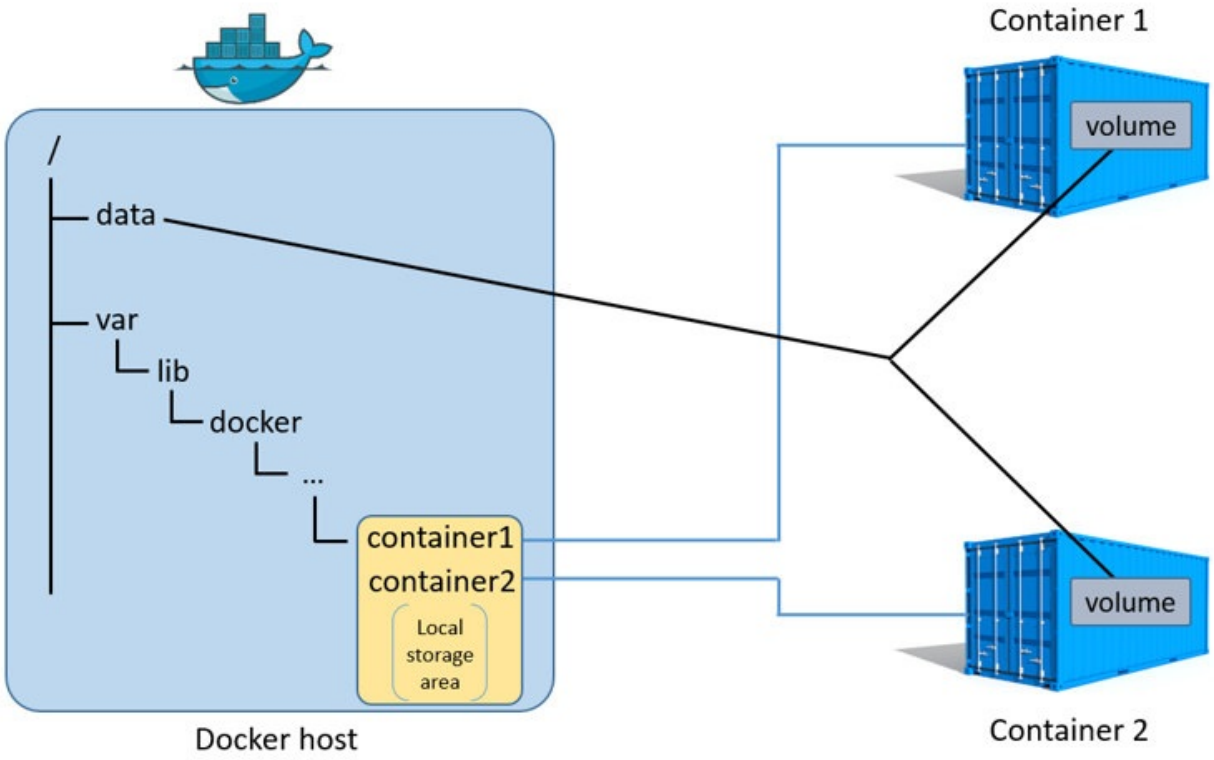
- 容器在启动时就就会创建 `writable layer`，而不仅是在有 `CoW` 操作时。

Data volumes and the storage driver

- 当容器被删除时，没有保存在 `data volume` 中的数据也将被删除。
- `data volume` 是直接被挂载到容器中的 `directory` 或 `file`
- `data volume` 不受 `storage driver` 控制
  - `Reads` and `writes` to `data volumes` `bypass` the storage driver and operate at native host
- 一个容器可以挂载多个 `data volume`
- 一个 `data volume` 可以被多个容器挂载

下图展示了：

- 一个主机启动了两个容器
- 每个容器在 `/var/lib/docker` 下有自己的磁盘空间
- 每个容器还分别挂载了一个 `data volume` 到容器的 `/data`



- Data volumes reside outside of the local storage area on the Docker host, further reinforcing their independence from the storage driver's control.
- When a container is deleted, any data stored in data volumes persists on the Docker host.

选择合适的 storage driver

Docker 支持多种不同的 storage driver, 使用 `pluggable architecture` 。

- `storage driver` 决定了如何 `保存` 和 `管理` `image` 和 `container`

选择合适的 `storage driver` ，以下是几个比较重要的参考因素：

- 如果系统内核支持多种 `storage driver` ，在没有指定 `storage driver` 的情况下，系统会按照以下优先级选择：
  - `aufs` : 默认。 `最老` 的 `storage dirver` ，但是并不是所有系统都支持。
  - `btrfs` , `zfs` : 使用配置最少的 。这些都依赖于 `文件系统(backing system )` 正确配置
  - 否则，尝试在最常见的情况下使用具有最佳整体性能和稳定性的 `storage driver` 。
    - `overlay2` 最佳，其次是 `overlay` 。这两者都不需要外配置。
    - `devicemapper` 再次，但要求 `direct-lvm` 的 生产环境 **(production environments)**，因为 `loopback-lvm` 在无配置的情况下，性能很低
  - Docker 源码决定了 `storage driver` 的顺序。
- `storage dirver` 可选列表依赖于 `docker 版本` 和 `系统版本`
  - `aufs` : Ubuntu and Debian
  - `btrfs` : SLES, 只支持 Docker EE
  - Support storage drivers per Linux distribution
- 部分 `storage driver` 对 `文件系统` 有要求。
  - Supported backing filesystems
- 结合以上限制，根据业务工作量选择合适的 `storage driver` 。
  - other considerations

Supported storage drivers per Linux distribution

不推荐使用需要 `禁用(disable)` 安全策略的 `storage driver`

- 在 `CentOS` 上使用 `overlay` or `overlay2` 需要禁用 `selinux`

Docker EE and CS-Engine

看 Product compatibility matrix

Docker CE

In general, the following configurations work on recent versions of the Linux distribution

Linux distribution	Recommended storage drivers
Docker CE on Ubuntu	`aufs` , `devicemapper` , `overlay2` (Ubuntu 14.04.4 or later, 16.04 or later), `overlay` , `zfs` , `vfs`
Docker CE on Debian	`aufs` , `devicemapper` , `overlay2` (Debian Stretch), `overlay` , `vfs`
Docker CE on CentOS	`devicemapper` , `vfs`
Docker CE on Fedora	`devicemapper` , `overlay2` (Fedora 26 or later, experimental), `overlay` (experimental), `vfs`

- 犹豫的时候，最好选用支持 `overlay2` 的 linux 系统。
- 使用 `Docker volume` 代替在 `writable layer` 上进行频繁的写入。
- `vfs` 不要选。除非你清楚的知道自己在做什么 its performance and storage characteristics and limitations.

> 注意：用常用的 `storage drive` 才能方便的排错。

Docker for Mac and Docker for Windows

Docker for Mac and Docker for Windows are intended for development, rather than production. Modifying the storage driver on these platforms is not supported.

Supported backing filesystems

对 Docker 而言， backing filesystem 就是 /var/lib/docker/ 目录 所在 的 文件系统格式 。

一些 storage dirver 只能在特定的 backing filesystem 上工作

Storage Driver	Supported backing filesystems
overlay , overlay2	ext4 , xfs
aufs	ext4 , xfs
devicemapper	direct-lvm
btrfs	btrfs
zfs	zfs

Other considerations

根据工作量选择

不同 storage driver 有不同的 特性 。

- aufs , overlay , overlay2 更适合操作 文件(file) 而不是 block 。使用 内存 更有效率，但大量写入操作会使 container's writable layer 快速增大。
- devicemapper , btrfs , zfs 更适合 block 操作。比如用做 docker volumes
- For lots of small writes or containers with many layers or deep filesystems , overlay 比 overlay2 更合适.
- btrfs 和 zfs 对内存需求很高
- zfs is a good choice for high-density workloads such as PaaS.

Shared storage systems and the storage driver

If your enterprise uses SAN , NAS , hardware RAID , or other shared storage systems, they may provide high availability , increased performance , thin provisioning , deduplication , and compression . In many cases, Docker can work on top of these storage systems, but Docker does not closely integrate(整合,一体化) with them.

- 每个 `docker storage driver` 都基于 `linux filesystem 或 volume manager` 。
- 使用时，确保所选的 `dockter storage dirver` 在所在 `shared storage system` 上是最好的。

Stability

出于稳定性考虑，一般而言 `aufs`，`overlay` 和 `devicemapper` 优先级最高。

Experience and expertise

出于 `maintaining (维护)` 方便考虑。

Test with your own workloads

测试工作量后在决定。

Check and set your current storage driver

> 重要：有些 `storage driver`，例如 `devicemapper`，`btrfs`，`zfs` 需要对系统进行额外的配置。

使用 `docker info` 命令查看当前 `Storage driver` 信息

```
docker info

Containers: 0
Images: 0
Storage Driver: overlay
 Backing Filesystem: extfs

...
<output truncated>
```

- 在 Docker 启动命令中，使用 `--storage-driver` 标志可以设置 `storage driver`

- (推荐) 配置 `daemon.json`
  - Linux: `/etc/docker/daemon.json`
  - Windows: `C:\programData\docker\config\daemon.json`

例如，指定 `devicemapper` driver。

```
{
  "storage-driver": "devicemapper"
}
```

> 注意：在 `ubuntu 16.04.3 x86_64 / docker-ce 17.06` 通过 `daemon.json` 配置 `registry-mirrors` 不成功。
>
> > 不知道 `storage driver` 是否可以通过 `daemon` 配置，未测试。

Docker daemon 的配置和排错

本文教你如何定制 `dockerd` 的参数。

Start the daemon using operating system utilities

- 安装 docker
- 配置开机启动

Start the daemon manually

出于 DEBUG 的目的，可以使用 `dockerd` 手工启动 Docker服务。如果非管理员，需要使用 `sudo` 命令。当通过这种方式启动 Docker 后， Docker 运行在前台，并直接在控制端上打印日志。

```
$ dockerd

INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
...
...
```

按 `Ctrl+C` 停止 Docker

Configure the Docker daemon

Docker daemon 包含了很多参数，

- 手工启动的时候，你可以通过给 `dockerd` 传参
- 还可以在 `daemon.json` 中配置

当然，更推荐使用 `daemon.json` 这种方式。

查看 dockerd 获得更多配置选项。

手动传参启动方式：

```
$ dockerd -D --tls=true --tlscert=/var/docker/server.pem --tlske
y=/var/docker/serverkey.pem -H tcp://192.168.59.3:2376
```

- `-D` : debugging
- `-tls` : 启用 TLS 证书
  - `--tlscert` , `--tlskey` 指定证书
- `-H` : 指定 daemon 监听的 `网络接口`

更好的方式是所有参数放入 `daemon.json` 中，并重启 Docker daemon。这种方式适用于各种 Docker平台。 例如将上面的参数放 `daemon.json` 中：

```
{
  "debug": true,
  "tls": true,
  "tlscert": "/var/docker/server.pem",
  "tlskey": "/var/docker/serverkey.pem",
  "hosts": ["tcp://192.168.59.3:2376"]
}
```

Docker 文档中有许多具体的配置选项。例如：

- Automatic start containr
- Limit a container's resources
- Configure storage drivers
- Container security

Troubleshoot the daemon

开启 debugging 模式可以帮助排错。如果 docker daemon 完全不响应，您还 可以通过个 docker daemon 发送信号 `SIGUSR` ， 强制将所有线程的堆栈信息 添加到守护程序日志中。

Out Of Memory Exceptions (OOME)

如果你的容器尝试使用的内存超过了系统上限，可能会触发 `Out Of Memory Exception (OOME)`，并且某个容器或**Docker daemon**可能会被 `kernel OOM killer` 关闭。为了防止这种情况的发生，确保容器主机有足够的内存，另外，查阅 Understand the risks of running out of memory.

Read the logs

排错，要善于看日志，不同的系统，日志存储的位置不一样。

Operating system	Location
RHEL, Oracle Linux	`/var/log/messages`
Debian	`/var/log/daemon.log`
Ubuntu 16.04+, CentOS	Use the command `journalctl -u docker.service`
Ubuntu 14.10-	`/var/log/upstart/docker.log`
macOS	`~/Library/Containers/com.docker.docker/Data/com.docker linux/console-ring`
Windows	`AppData\Local`

Enable debugging

有两种方式可以开启 debugging 模式。

其一，为 `dockerd` 传参 `-D`。

其二，为 `daemon.json` 增加 `"debug": true` 字段。这种方式适用于 Docker 各个平台。

1. 编辑 `/etc/docker/daemon.json`。如果是 **Windows** 或 **MacOS**，不能直接编辑，需要在 **Preferences / Daemon / Advanced** 中处理。
2. 如果文件为空，`daemon.json` 如下：

```
{
  "debug": true
}
```

如果文件存在，则直接添加 `"debug": true` 字段，注意换行出的 `逗号` 。另外，同时可以指定 `log-level` 等级；默认为 `info` ；有效值为 `debug` , `info` , `warm` , `error` , `fatal` 。

1. 如果是 Linux 主机， `dockerd` 发送 `HUP` 信号， reload 配置。如果是 Windows 主机，重启 docker。

```
$ sudo kill -SIGHUP $(pidof dockerd)
```

Force a stack trace to be logged

如果 daemon 无响应，你可以向 daemon 发送 `SIGUSR1` 信号强制将所有堆栈信息保存下来。

- **Linux**:

  ```
  $ sudo kill -SIGUSR1 $(pidof dockerd)
  ```

- **Windows Server**: 下载 [docker-signal]https://github.com/jhowardmsft/docker-signal)，使用参数 `--pid=<PID of daemon` 执行命令。

这样便会强制保存堆栈信息，但不会停止 daemon。 Daemon 的日志中会保存上述堆栈信息，或记录保存上述堆栈信息文件的路径。

The daemon will continue operating after handling the `SIGUSR1` signal and dumping the stack traces to the log. The stack traces can be used to determine the state of all goroutines and threads within the daemon.

View stack traces

Docker daemon 日志可以通过以下方式查看：

- Linux 系统使用 `systemctl` : `journalctl -u docker.service`

- 早期的 Linux系统： `/var/log/message` ， `/var/log/daemon.log` ， `/var/log/docker.log`
- Windows Server 的 DockerEE： `Get-EventLog -LogName Application -Source Docker -After (Get-Date).AddMinutes(-5) | Sort-Object Time`

> 注意: 在 windows 或 MacOS 上不能手动生成堆栈信息。但是，可以通过 Docker 任务栏中 **Diagnose and feddback** 向 Docker 团队反馈你的问题。

Docker 日志如下：

```
...goroutine stacks written to /var/run/docker/goroutine-stacks-2017-06-02T193336z.log
...daemon datastructure dump written to /var/run/docker/daemon-data-2017-06-02T193336z.log
```

上述信息展示了 Docker 将堆栈信息保存的路径。

Check whether Docker is running

- 使用命令 `docker info` 来检查 Docker 是否正在运行。
- 你还可以使用操作系统指令来检查，例如 `sudo systemctl is-active docker` 或 `sudo status docker` 或 `sudo service docker status` 或使用 Windows 工具
- 当然，你还可以使用 `ps` 或 `top` 查看 `dockerd` 进程是否存在。

Start containers automatically

Docker 提供 restart 策略 控制容器在某些情境想是否自动重启。 重启策略保证了通过 `--link` 连接的容器以正确的顺序启动，而不在使用进程管理器控制启动流程。

重启策略与 `dockerd` 的 `--live-restore` 参数不一样。使用 `--live-restore` 允许你在 Docker升级期间保持容器运行，但 网络 和 用户输入 会被中断。

Use a restart policy

在使用 `docker run` 命令启动容器时，使用 `--restart` 可以为容器指定重启策略。 `--restart` 的可选值包括以下

Flag	描述
`no`	不要自动重新启动容器。（默认）
`on-failure`	如果由于出现错误而重新启动容器，该错误显示为非零退出代码。
`unless-stopped`	重新启动容器，除非它被显示停止或Docker本身被停止或重新启动。
`always`	如果停止，请务必重新启动容器。

以 redis 容器为例， 除非明确停止或 docker 重启， 容器始终会重启。

```
$ docker run -dit --restart unless-stopped redis
```

Restart policy details

使用重新启动策略时，请记住以下几点：

- 重新启动策略仅在容器启动成功后生效。在这种情况下，启动成功意味着容器至少持续 10 秒钟，Docker 已经开始监视它。这样可以防止根本不启动的容器进入重启循环。
- 如果您 手动 停止容器，则重新启动策略将被忽略，直到 Docker 守护程序重新

启动或容器被手动重新启动。这是防止重新启动循环的另一个尝试。

- 重新启动策略仅适用于容器。群组服务的重新启动策略配置不同。看 flags related to service restart

Use a process manager

如果重启策略满足要求，可以使用 进程管理工具，例如 `upstart`，`systemd` 或 `supervisor` 。

进程管理工具运行在容器内，并检查容器内的进程是否启动，如果没有则启动它。这不是 Docker-aware 的，仅监控容器内的进程。 不推荐使用这种方式，因为这是基于系统平台的，不同 Linux 发行版之间。

> 警告: 不要将 docker 重启策略 与 主机级别 的进程管理工具混用，二者可能冲突。

如果要使用进程管理器，在启动容器或服务时，需要将其配置成为类似手动使用 `docker start` 或 `docker service` 启动容器或服务。o use a process manager, configure it to start your container or service using the same docker start or docker service command you would normally use to start the container manually.

使用 Promethues 收集 Docker 性能指标

> 只讲如何在 Linux 下配置 Prometheus。其他操作系统，直接看 原文 。其实也没什么也看的

Prometheus 是一个开源监控和报警工具。可以将 Docker 作为其监控目标。本文将启动一个 Prometheus 容器，并兼容 Docker 实例。

> 警告：本文中所有可用的指标和其名字都可能在将来的版本中发生变化。

目前，你只能监控 Docker 本身，而不能监控 Docker 容器中的程序。

环境信息

```
#  操作系统版本
Kernel Version: 4.4.0-93-generic
Operating System: Ubuntu 16.04.3 LTS

# Docker 版本
Server Version: 17.09.0-ce

# prometheus 版本
prometheus, version 1.8.1 (branch: HEAD, revision: 3a7c51ab70fc7
615cd318204d3aa7c078b7c5b20)
  build user:       root@ab78fb101775
  build date:       20171023-15:50:57
  go version:       go1.9.1
```

配置 Docker

将 Docker daemon 配置为 Prometheus 的目标，你需要指定 `metrics-address` 。当然最为推荐的方法就配置 `/etc/docker/daemon.json` 。注意 `json` 文件格式。

```
{
  "metrics-addr" : "127.0.0.1:9323",
  "experimental" : true
}
```

加载 dockerd 配置

```
$ sudo kill -SIGHUP $(pidof dockerd)
```

现在 Docker 在 9323 端口上公开 `Prometheu-兼容指标` 。

配置和启动 Prometheus

本例中， Prometheus 将通过 Docker 容器提供服务。

将下述配置文件保存到 `/tmp/prometheus.yml` 中

```
# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15
 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. Th
e default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

  # Attach these labels to any time series or alerts when commun
icating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
      monitor: 'codelab-monitor'

# Load rules once and periodically evaluate them according to th
e global 'evaluation_interval'.
rule_files:
  # - "first.rules"
  # - "second.rules"
```

```
# A scrape configuration containing exactly one endpoint to scra
pe:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any tim
eseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'docker'
        # metrics_path defaults to '/metrics'
        # scheme defaults to 'http'.

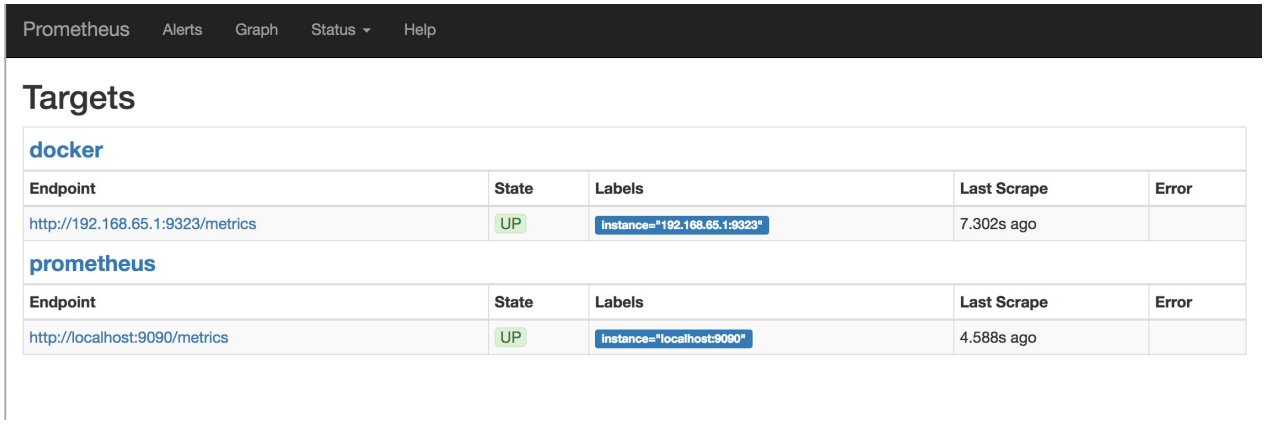
    static_configs:
      - targets: ['localhost:9323']
```

启动一个 单副本 的 Promethues 服务

```
$ docker service create --replicas 1 --name my-prometheus \
    --mount type=bind,source=/tmp/prometheus.yml,destination=/et
c/prometheus/prometheus.yml \
    --publish 9090:9090/tcp \
    prom/prometheus
```

> 注意：如果你的 Docker Host 没有加入到集群中，系统会提示你先 `docker swarm init`

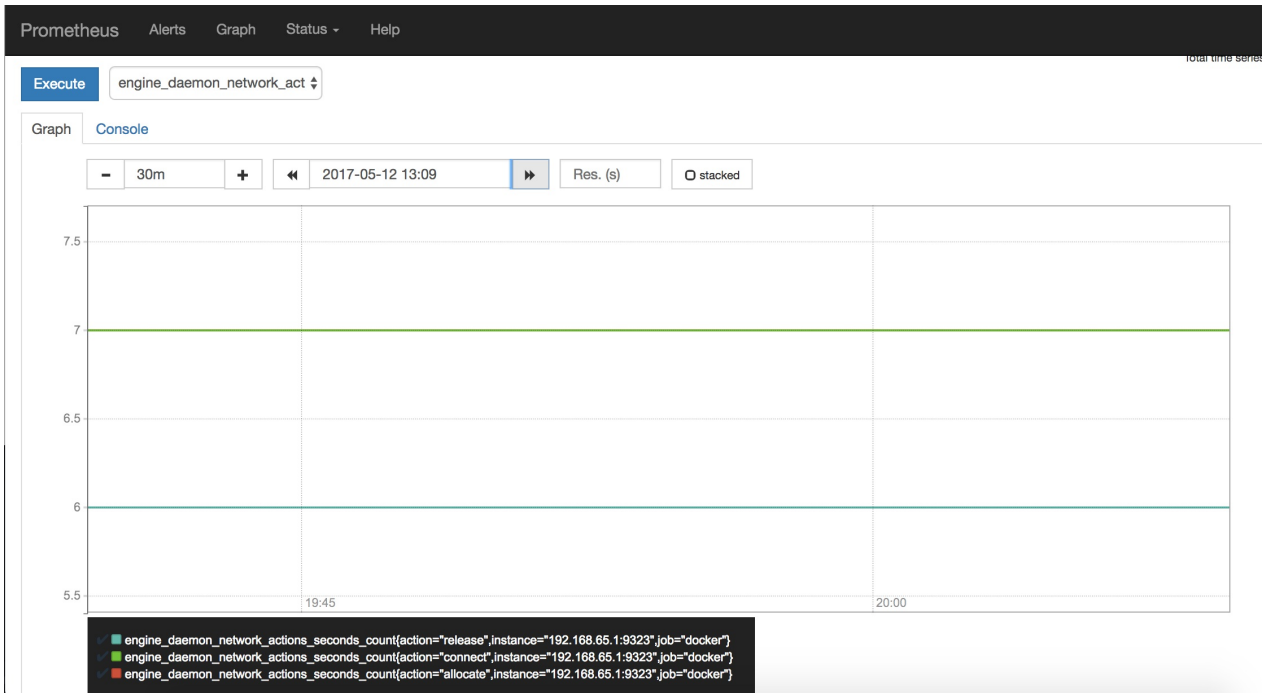
通过 web 页面访问查看结果



使用 **prometheus**

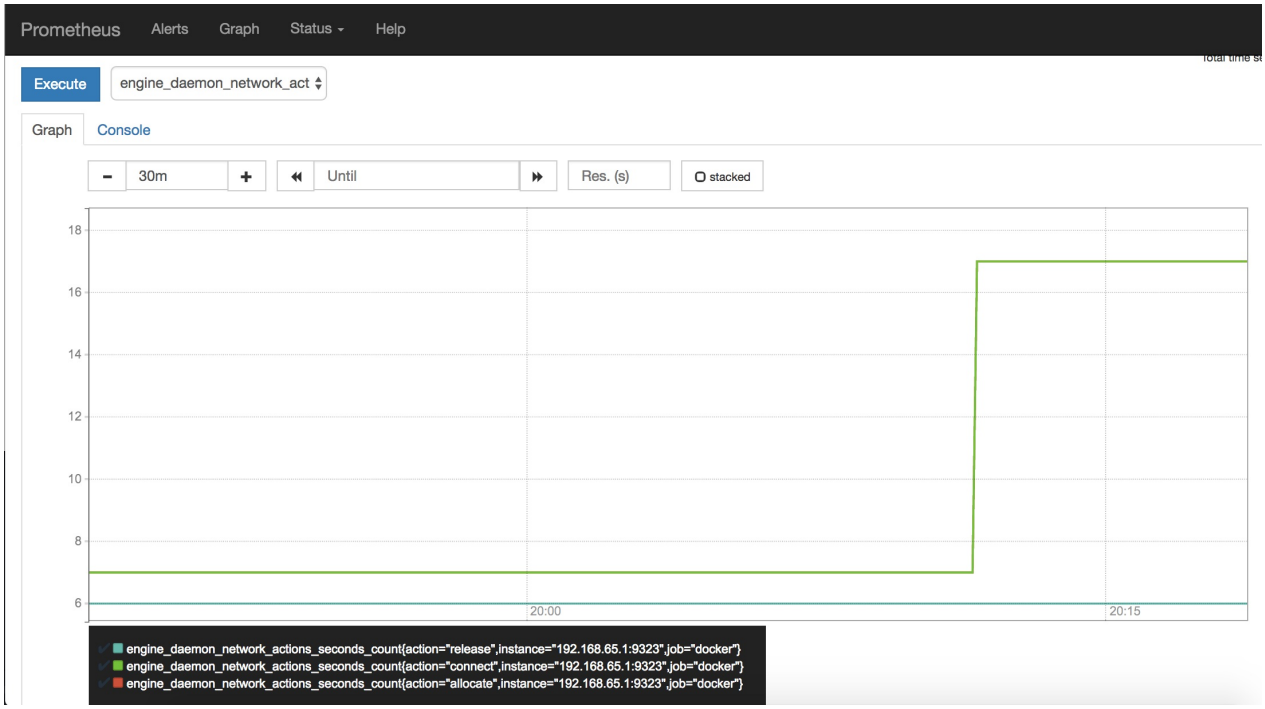
点击 **Graphs** ， 在 下拉菜单 中选择一个 指标参数 , 点击 **Execute** 。

屏幕上就会绘制所选择的监控图 。



为了让监控图看起来更有趣一个点， 创建一个 10 个副本的服务，不停的 `ping` 。

```
$ docker service create \
  --replicas 10 \
  --name ping_service \
  alpine ping docker.com
```



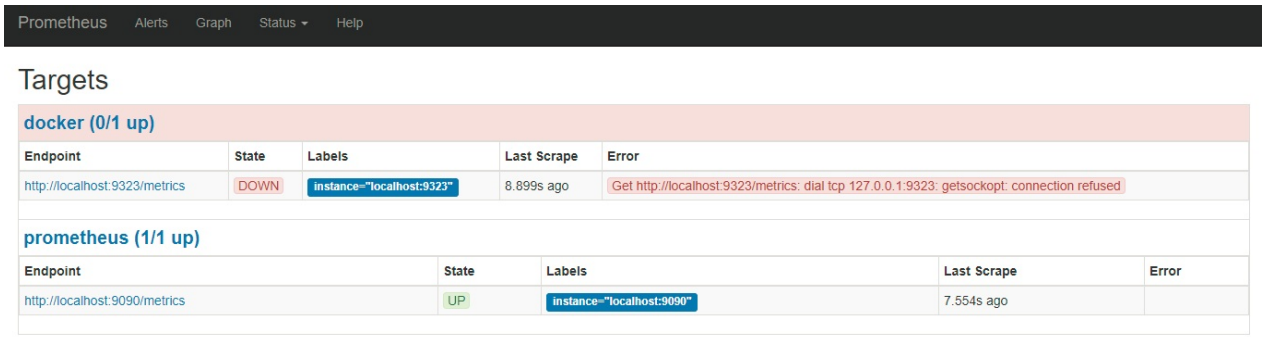
观察完成后，停止并删除 `ping_service` 。

```
$ docker service rm pingservice
```

等几分钟后，你应该开可以看到监控图又回到了空闲级别。

实验说明

1. `daemon.json` 中的配置的 `metrics-address` 指标，在我的监控图中报错



可能正如文章开头所言，监控的 `metrics` 已经失效了。

1. Prometheus 中 **Graph** 下拉菜单中的监控项目名称与官网文档中的不一样。通过本文，可以初步搭建一个监控。

下一步

访问 Promethues 获取更多信息。

Docker daemon 宕机期间保持容器存活

默认情况下，当 Docker daemon 退出的时候， 会关闭正在运行的容器。从 `1.12` 开始，可以配置 daemon 参数，使容器在 daemon 进程不可用的时候已经保持运行。改参数降低了在 daemon 崩溃、计划性维护以及升级时容器的停止时间。

> 注意： `live restore` 不支持 windows 容器。但是支持在 Docker for Windows 上跑的 Linux 容器。

启用 live restore 选项

有两种方法配置：

1. 配置 `/etc/docker/daemon.json` ，推荐这种方式，但要注意 `json` 格式

```
{
    "live-restore": true
}
```

发送命令，热加载新配置的参数

```
$ kill -SIGHUP $(pidof dockerd)
```

1. 或者给 `dockerd` 传参。

```
$ sudo dockerd --live-restore
```

升级时的 live restore

`live restore` 支持在 `minor` 版本之间升级时恢复容器与 daemon 之间的连接关系。例如 Docker Engine `1.12.1` 到 `1.12.2` 。

如果版本跨越较大的升级， daemon 可能不能恢复与容器之间的连接。如果出现这种情况， daemon 会忽略正在运行的容器，而你必须手动进行管理。

重启时的 live restore

只有在 daemon 配置不变的的情况下重启 daemon `live restore` 才会生效。例如， `live restore` 在 daemon 更换了 `bridge ip` 和 `graphdriver` 时不会生效。

> 并不是所有的参数发生变化了 `live restore` 都不会生效。应该处于比较底层的一些参数发生变化了，才不会生效。例如，增加或修改 `registry-mirrors` 就有效。

live restore 对运行中的容器的影响

docker daemon 进程长时间不活动会对容器产生不良影响。 容器进程会日志写入到一个 FIFO 日志文件中，以供 daemon 恢复之后处理。如果 daemon 长时间不处理这些日志文件， buffer 缓冲去会填满并停止写入新的日志。一个被写满了的日志在有更多空间钱，会阻扰进程。 buffer 默认大小是 `64K` 。

刷新 buffer 必须要重启 Docker 。

改变 buffer 大小需要修改 `/proc/sys/fs/pipe-max-size` 的值

live restore 与 swarm mode

`live store` 与 Docker swarm mode 不兼容。When the Docker Engine runs in swarm mode, the orchestration feature manages tasks and keeps containers running according to a service specification.

排错

简单记录一下。

1. 在配置 `"live-restore" : true` 之后重启 Docker
2. 不知道在什么时候做测试的时候，开启了 swarm mode，及 `docker swarm init`
3. 由于 `live-restore` 与 swarm mode 冲突，因此重启 Docker 的时候出现以下报错：

```
Nov 01 10:33:59 instance-4 dockerd[30728]: time="2017-11-01T10:3
3:59.310839742Z" level=info msg="There are old running container
s, the network config will not take affect"
Nov 01 10:33:59 instance-4 dockerd[30728]: time="2017-11-01T10:3
3:59.322647040Z" level=info msg="Loading containers: done."
Nov 01 10:33:59 instance-4 dockerd[30728]: time="2017-11-01T10:3
3:59.344132196Z" level=info msg="Docker daemon" commit=afdb6d4 g
raphdriver(s)=overlay2 version=17.09.0-ce
Nov 01 10:33:59 instance-4 dockerd[30728]: time="2017-11-01T10:3
3:59.344906513Z" level=fatal msg="Error starting cluster compone
nt: --live-restore daemon configuration is incompatible with swa
rm mode"
```

> 注意：在使用 `journalctl -xe` 命令查看系统日志时，可能会出现日志过长，一个屏幕不能完全显示的情况。但不会自动换行，这个时候可以使用键盘 `左右` 键调整日志位置。

Install Docker Compose

安装很简单，到 docker-compose 的 github release 页面下载后，将 `二进制` 文件放入 `/usr/local/bin/` 目录下即可

安装

```
curl -L https://github.com/docker/compose/releases/download/1.16.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

升级

下载最新版本覆盖原来的即可

卸载

删除二进制文件即可

Docker Compose 案例

准备

- 安装 docker
- 安装 docker-compose

Step1: Setup

1. 创建一个目录

```
$ mkdir compose_test
$ cd compose_test
```

1. 创建一个 `app.py` ，运维 web 程序

```
#!/usr/bin/python
#
# filename: app.py

from flask import Flask
from redis import Redis

app = Flask(__name__)
redis = Redis(host='redis', port=6379)  # host='redis' redis 为
`redis container` 的 `容器名`

@app.route('/')
def hello():
    count = redis.incr('hits')
    return 'Hello World! I have been seen {} times.\n'.format(co
unt)

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

> 注意： `host='redis'` 中的 `redis` 为 运行 ***redis*** 程序的容器的容器名，端口使用默认的 `6789`

1. 创建 `requirements.txt` ，安装 python 组件

```
flask
redis
```

Step 2: Create a Dockerfile

在同级目录中，创建 Dockerfile， 启动任务时，可以构建镜像

```
FROM python:3.4-alpine
ADD . /code
WORKDIR /code
RUN pip install -i https://mirrors.ustc.edu.cn/pypi/web/simple -
r requirements.txt
CMD ["python", "app.py"]
```

- `FROM` 使用 `python:3.4-alpine` 镜像作为基础镜像
- `ADD` 将当前目录 `.` 放入镜像 `/code`
- `WORKDIR` 将工作目录设置为 `/code`
- `RUN` 安装 python 依赖，在国内使用 `中科大` 的源，加速下载
- `CMD` 设置镜像入口为 `python app.py`

Step 3: Define services in a Compose file

创建 `docker-compose.yml` ， 编写发布步骤

```
# docker-compose.yaml

version: '3'
services:
  web:
    build: .
    ports:
     - "5001:5000"
  redis:
    image: "redis:alpine"
```

`compose` 定义了两个 `服务`：`web` 和 `redis`

- `web` 服务
  - `build`：使用当前目录的 `Dockerfile` build 的镜像
  - `ports`：映射宿主机 5001 端口到容器的 5000 端口，与 `docker run -p 5001:5000` 同。
    - `Flask` 默认端口 5000
    - 由于我本机的 5000 端口已经跑了 `docker registry proxy`，因此修改了端口映射。
- `redis` 服务
  - 使用 Docker Hub 中的 `redis` 镜像

Step 4: Build and run your app with Compose

1. 使用命令 `docker-compose up` 服务

```
$ docker-compose up

Starting composetest_web_1 ...
composetest_redis_1 is up-to-date
Starting composetest_web_1 ... error

ERROR: for composetest_web_1  Cannot start service web: driver f
ailed programming external connectivity on endpoint composetest_
web_1 (92e55e324676835b2d5053a8aa6eafbb639f96d1f9dfe5203e2d7a66d
eb793c1): Bind for 0.0.0.0:5000 failed: port is already allocate
```

```
d

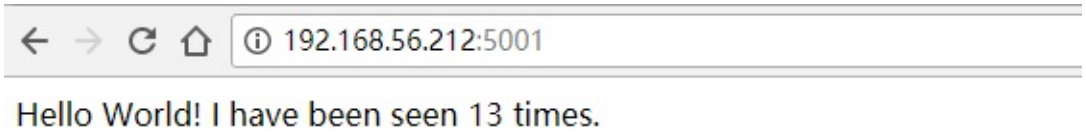
ERROR: for web  Cannot start service web: driver failed programm
ing external connectivity on endpoint composetest_web_1 (92e55e3
24676835b2d5053a8aa6eafbb639f96d1f9dfe5203e2d7a66deb793c1): Bind
for 0.0.0.0:5000 failed: port is already allocated
ERROR: Encountered errors while bringing up the project.

######################################################################
############
# 第一此启动出错，是因为本机已经跑了一个容器 `registry-proxy` 占用了 500
0 端口
# 因此，将 docker-compose 中的端口映射从 `5000:5000` 修改为了 `5001:5
000`
######################################################################
############

$ docker-compose up
Recreating composetest_web_1 ...
composetest_redis_1 is up-to-date
Recreating composetest_web_1 ... done
Attaching to composetest_redis_1, composetest_web_1
redis_1  | 1:C 25 Sep 08:08:25.420 # oO0OoO0OoO0Oo Redis is star
ting oO0OoO0OoO0Oo
redis_1  | 1:C 25 Sep 08:08:25.477 # Redis version=4.0.2, bits=6
4, commit=00000000, modified=0, pid=1, just started
redis_1  | 1:C 25 Sep 08:08:25.477 # Warning: no config file spe
cified, using the default config. In order to specify a config f
ile use redis-server /path/to/redis.conf
redis_1  | 1:M 25 Sep 08:08:25.478 * Running mode=standalone, po
rt=6379.
redis_1  | 1:M 25 Sep 08:08:25.478 # WARNING: The TCP backlog se
tting of 511 cannot be enforced because /proc/sys/net/core/somax
conn is set to the lower value of 128.
redis_1  | 1:M 25 Sep 08:08:25.478 # Server initialized
redis_1  | 1:M 25 Sep 08:08:25.478 # WARNING overcommit_memory i
s set to 0! Background save may fail under low memory condition.
 To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl
.conf and then reboot or run the command 'sysctl vm.overcommit_m
emory=1' for this to take effect.
```

```
redis_1  | 1:M 25 Sep 08:08:25.478 # WARNING you have Transparen
t Huge Pages (THP) support enabled in your kernel. This will cre
ate latency and memory usage issues with Redis. To fix this issu
e run the command 'echo never > /sys/kernel/mm/transparent_hugep
age/enabled' as root, and add it to your /etc/rc.local in order
to retain the setting after a reboot. Redis must be restarted af
ter THP is disabled.
redis_1  | 1:M 25 Sep 08:08:25.478 * Ready to accept connections
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to q
uit)
web_1    |  * Restarting with stat
web_1    |  * Debugger is active!
web_1    |  * Debugger PIN: 326-358-579
```

访问主机 `htpp://docker_host_ip:5001` 查看结果



使用命令 `docker container ls` 查看当前运行的主机

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND
 CREATED              STATUS              PORTS
   NAMES
54870af6fd43        redis:alpine        "docker-entrypoint..."
 About a minute ago   Up 7 seconds        6379/tcp
   composetest_redis_1
c80b129f2730        composetest_web     "python app.py"
 About a minute ago   Up 7 seconds        0.0.0.0:5001->5000/tcp
   composetest_web_1
```

1. 使用 `docker image ls` 命令查看本地镜像

```
$ docker image ls
REPOSITORY              TAG                 IMAGE ID
CREATED             SIZE
composetest_web         latest              e2c21aa48cc1
4 minutes ago       93.8MB
python                  3.4-alpine          84e6077c7ab6
7 days ago          82.5MB
redis                   alpine              9d8fa9aa0e5b
3 weeks ago         27.5MB
```

使用命令 `docker inspect <image_tag>/<image_id>` 查看具体信息

1. 重新打开一个终端，进入到 `docker-compose.yml` 所在目录，使用命令 `docker-compose down` 关闭服务

```
$ docker-compose down
Stopping composetest_web_1   ... done
Stopping composetest_redis_1 ... done
Removing composetest_web_1   ... done
Removing composetest_redis_1 ... done
Removing network composetest_default
```

或者直接在 `docker-compose up` 的终端中 `Ctrl+C` 退出进程

使用 `docker-compose stop` 后， `容器关闭，但不删除`

```
$ docker container ps -a
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS                      PORTS
        NAMES
78b2acf2d71f        redis:alpine        "docker-entrypoint..."
 25 seconds ago      Exited (0) 9 seconds ago
        composetest_redis_1
7d34fdcd3f91        composetest_web     "python app.py"
 25 seconds ago      Exited (0) 9 seconds ago
        composetest_web_1
```

Step 5: Edit the Compose file to add a bind mount

编辑 `docker-compose.yml` ，增加 volumes bind mount

`volumes` 将 `.:/code` 本地目录映射到了容器中

- 与 `docker run -v host_path:container_path` 相同
- 与 `docker run` 不同的是， `docker-compose` 可以使用相对目录，而 `docker run` 只能使用绝对目录

```
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"

    ## 映射本地目录到容器中
    volumes:
     - .:/code

  redis:
    image: "redis:alpine"
```

通过目录映射，可以方便的在本地修改文件并查看效果，而不用 `rebuild` 镜像

Step 6: Re-build and run the app with Compose

```
$ docker-compose up
Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to q
uit)
...
```

> Shared folders, volumes, and bind mounts
>
> - If your project is outside of the `Users` directory (`cd ~`), then you need to share the drive or location of the Dockerfile and volume you are using. If you get runtime errors indicating an application file is not found, a volume mount is denied, or a service cannot start, try enabling file or drive sharing. Volume mounting requires shared drives for projects that live outside of `C:\Users` (Windows) or `/Users` (Mac), and is required for any project on Docker for Windows that uses Linux containers. For more information, see Shared Drives on Docker for Windows, File sharing on Docker for Mac, and the general examples on how to Manage data in containers.
> - If you are using Oracle VirtualBox on an older Windows OS, you might encounter an issue with shared folders as described in this VB trouble ticket. Newer Windows systems meet the requirements for Docker for Windows and do not need VirtualBox.

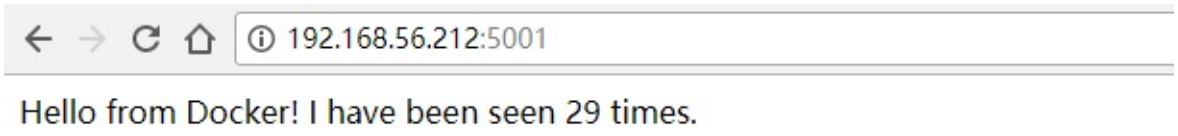
Step 7: Update the application

现在代码目录已经映射到容器中了，因此我们不需要 rebuild 惊喜就可以看到修改后的效果。

1. 重开一个终端窗口，修改 `app.py` 文件。 `Hello World` 改为 `Hello from Docker` .

```
# return 'Hello World! I have been seen {} times.\n'.format(coun
t)
return 'Hello from Docker! I have been seen {} times.\n'.format(
count)
```

1. 刷新窗口，查看结果



Step 8: Experiment with some other commands

- `docker-compose up -d`：后台运行
- `docker-compose ps`

```
$ docker-compose ps
       Name                     Command               State
       Ports
------------------------------------------------------------------
---------------------
composetest_redis_1   docker-entrypoint.sh redis ...   Up      6
379/tcp
composetest_web_1     python app.py                    Up      0
.0.0.0:5001->5000/tcp
```

- `docker-compose run`：一次性命令。例如查看 `web` 服务的环境变量

```
$ docker-compose run web env
PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/us
r/bin:/sbin:/bin
HOSTNAME=8ad8bfde60ca
TERM=xterm
LANG=C.UTF-8
GPG_KEY=97FC712E4C024BBEA48A61ED3A5CA953F73C700D
PYTHON_VERSION=3.4.7
PYTHON_PIP_VERSION=9.0.1
HOME=/root
```

- `docker-compose stop`：停止服务
- `docker-compose down`：关闭所有容器并删除，默认保留 `数据卷`
  - `docker-compose down --volumes`：同时也删除容器使用的 `数据卷`

> 注意：`docker-compose` 都必须指定 `docker-compose.yml` 。否则会报错
>
> - `default`：使用当前目录下的 `docker-compose.yml` 或 `docker-compose.yaml`
> - 或者使用 `docker-compose -f path/compose_name.yml subcommand`

```
$ docker-compose ps
ERROR:
        Can't find a suitable configuration file in this directo
ry or any
        parent. Are you in the right directory?

        Supported filenames: docker-compose.yml, docker-compose.
yaml
```

Overview of docker-compose CLI

Command options overview and help

使用 `docker-compose --help` 查看相关子命令

完整命令 `docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]` 实现构建管理服务。

使用 `-f` 指定一个或多个 compose 文件

使用 `-f` 指定 `compose file` 的 `路径` 与 `文件名`

Specifying multiple Compose files

`docker-compose` 支持使用 `-f` 同时指定多个 `compose file` 。

- 当指定多个文件时， `docker-compose` 会将做个文件组合成一个文件。
- `build` 的时候，按照指定文件的顺序进行。
- 如果前后文件中都对同一个服务进行了配置，如果有冲突参数，那么后者将 `重载 override` 前者的参数。 其他非冲突参数 `同时生效`

举个例子：

如命令，指定了两个 `compose file`

```
$ docker-compose -f docker-compose.yml -f docker-compose.admin.y
ml run backup_db
```

两个 `compose file` 都存在 `webapp` 服务

1. `docker-compose.yml` 指定了一个 `webapp` 服务 ```yaml

docker-compose.yml

webapp: image: examples/web ports:

```
- "8000:8000"
```

volumes:

```
- "/data"
```

```
2. `docker-compose.admin.yml` 同样指定了一个 `webapp` 服务。 相同 `filed` 的配置，将会被 `重载`。其他没有冲突的地方，则同时生效。

```yaml
# docker-compose.admin.yml

webapp:
  build: .
  environment:
    - DEBUG=1
```

- `-f -` ，使用 `- (dash)` 表示 `标准输入(stdin)` 作为 `compose file` 。当使用 `stdin` 时，配置中的所有路径都是以 `当前目录` 的 `相对路径` .
- `-f` 是可选的。如果没有使用 `-f` ，那么 `docker-compose` 会搜索 `当前目录` 及 `上级目录` 中搜索 `docker-compose.yml` 和 `docker-compose.override.yml` 文件
  - 目录结构说明
    - `compose_test` 为一个标准 `docker-compose` 结构目录。所有文件都在 `compose_test` 目录下，可以正常运行。
    - `compose_t3` 为 `compose_test` 的同级目录
    - `compose_t2` , `compose_t4` 为 `compose_test` 的子、孙目录
  - 实验结果：使用 `docker-compose up` 命令可实现搜索的
    - [ ] ~同级目录： 不行~
    - [x] 子目录: 可以
    - [x] 孙目录: 可以
```

```
$ tree -a
.
├── compose_t3/
└── compose_test/
    ├── compose_t2/
    │   └── compose_t4/
    ├── app.py
    ├── docker-compose.v1.yml
    ├── docker-compose.yml
    ├── Dockerfile
    └── requirements.txt
```

- 搜索路径时，必须存在 `docker-compose.yml` 文件。如果 `docker-compose.override.yml` 同时存在。两个文件会合并为一个文件。
  - 如果存在冲突，`docker-compose.override.yml` 中的配置会 `重载` `docker-compose.yml` 中的。

删除 `docker-compose.yml` ，只保留 `docker-compose.override.yml` 文件。启动时报错。

```
$ mv docker-compose.yml docker-compose.yml__

$ ls docker-compose.*
docker-compose.override.yml  docker-compose.v1.yml  docker-compo
se.yml__

$ docker-compose up -d
ERROR:
        Can not find a suitable configuration file in this direc
tory or any
        parent. Are you in the right directory?

        Supported filenames: docker-compose.yml, docker-compose.
yaml
```

同时存在 `docker-compose.yml` 和 `docker-compose.override.yml` 文件。

```
$ cat docker-compose.override.yml
version: '3'
services:
  web:
    build: .
    ports:
     - "5003:5000"
    volumes:
     - .:/code
  redis:
    image: "redis:alpine"


$ docker-compose up -d


$ docker container ls
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
                          NAMES
dfa3dce01fa9        redis:alpine        "docker-entrypoint..."
 7 seconds ago       Up 6 seconds        6379/tcp
                          composetest_redis_1
34a9f6466e1d        composetest_web     "python app.py"
 7 seconds ago       Up 6 seconds        0.0.0.0:5001->5000/tcp,
 0.0.0.0:5003->5000/tcp   composetest_web_1


$ curl 127.0.0.1:5001
Hello from Docker! I have been seen 1 times.
$ curl 127.0.0.1:5001
Hello from Docker! I have been seen 2 times.
$ curl 127.0.0.1:5003
Hello from Docker! I have been seen 3 times.
$ curl 127.0.0.1:5003
Hello from Docker! I have been seen 4 times.
$ curl 127.0.0.1:5001
Hello from Docker! I have been seen 5 times.
```

> 注意：
>
> - 像端口这样的，因为宿主机不能同时监听两个相同的端口， 因此，映射端口是，宿主机端口相同，会报错。 宿主机端口不同，就属于两个不同的配置了，因此，叠加。
> - 类似于 volumes 这类配置，目录 bind 到容器中相同的时候，就产生了 override 的效果。

```
$ cp -a ./* ../compose_t3/

$ cat docker-compose.yml
version: '3'
services:
  web:
    build: .
    ports:
     - "5001:5000"
    volumes:
     - .:/code
  redis:
    image: "redis:alpine"

$ cat docker-compose.override2.yml
version: '3'
services:
  web:
    build: .
    ports:
     - "5001:5000"
    volumes:
     - ../compose_t3:/code
  redis:
    image: "redis:alpine"

$ grep return app.py ../compose_t3/app.py
app.py:    return 'Hello from Docker compose_test! I have been s
een {} times.\n'.format(count)
../compose_t3/app.py:    return 'Hello from Docker compose_t3! I
 have been seen {} times.\n'.format(count)
```

```
$ docker-compose -f docker-compose.yml  -f docker-compose.overri
de2.yml up -d
Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_web_1 ... done

$ docker container ls
CONTAINER ID        IMAGE               COMMAND
 CREATED             STATUS              PORTS
  NAMES
6ea717e23b74        redis:alpine        "docker-entrypoint..."
 9 seconds ago       Up 8 seconds        6379/tcp
  composetest_redis_1
6bec5939d3dd        composetest_web     "python app.py"
 9 seconds ago       Up 8 seconds        0.0.0.0:5001->5000/tcp
  composetest_web_1
df21e8152501        registry:2          "/entrypoint.sh /e..."
 5 days ago          Up 47 hours         0.0.0.0:5000->5000/tcp
  registryproxy_mirror_1

## 被 重载 了
$ curl 127.0.0.1:5001
Hello from Docker compose_t3! I have been seen 1 times.
```

Specifying a path to a single Compose file

使用 `-f` 指定一个非当前目录的 `compose file`

- 可能使用过 `命令行`
- 可能是通过 `COMPOSE_FILE 环境变量` 定义
- 或者是通过 `系统环境变量` 定义

For an example of using the `-f` option at the command line, suppose you are running the Compose Rails sample, and have a `docker-compose.yml` file in a directory called `sandbox/rails`. You can use a command like `docker-compose pull` to get the `postgress` image for the `db` service from anywhere by using the `-f` flag as follows:

```
$ docker-compose -f ~/sandbox/rails/docker-compose.yml pull db
Pulling db (postgres:latest)...
latest: Pulling from library/postgres
ef0380f84d05: Pull complete
50cf91dc1db8: Pull complete
d3add4cd115c: Pull complete
467830d8a616: Pull complete
089b9db7dc57: Pull complete
6fba0a36935c: Pull complete
81ef0e73c953: Pull complete
338a6c4894dc: Pull complete
15853f32f67c: Pull complete
044c83d92898: Pull complete
17301519f133: Pull complete
dcca70822752: Pull complete
cecf11b8ccf3: Pull complete
Digest: sha256:1364924c753d5ff7e2260cd34dc4ba05ebd40ee8193391220be0f9901d4e1651
Status: Downloaded newer image for postgres:latest
```

Use `-p` to specify a project name

每个配置文件爱你都有个项目命令

- 通过 `-p` 指定一个 `自定义` 的项目名。
- 如果不使用 `-p` ， `docker-compose` 会使用 `当前目录名称` 作为项目名。
- 更多信息看 COMPOSE_PROJECT_NAME 环境变量

Set up environment variables

You can set environment variables for various `docker-compose` options, including the `-f` and `-p` flags.

For example,

- the COMPOSE_FILE environment variable relates to the `-f` flag,
- COMPOSE_PROJECT_NAME environment variable relates to the `-p` flag.

- Also, you can set some of these variables in an environment file.

Compose CLI environment variables

Several environment variables are available for you to configure the Docker Compose command-line behaviour.

Variables starting with DOCKER_ are the same as those used to configure the Docker command-line client.

COMPOSE_PROJECT_NAME

设置 `project_name` 。

This value is prepended along with the service name to the container on start up.

- 例如， `project_name` 为 `myapp`
  - 两个服务： `db` 和 `web`
  - 那么启动后，容器名称分别为 `myapp_db_1` 和 `myapp_web_1`
- `COMPOSE_PROJECT_NAME` 是可选的。如果不设置那么， `COMPOSE_PROJECT_NAME` 默认会使用 `docker-compose.yml` 所在目录名称。

```
$ pwd
/data/docker-compose/compose_test/compose_t2/compose_t4

$ docker-compose up -d
Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_web_1 ... done
```

- 在命令行中，可以通过 `docker-compose -p project_name` 指定

```
$ docker-compose -p myapp  up -d
Creating network "myapp_default" with the default driver
Creating myapp_web_1 ...
Creating myapp_redis_1 ...
Creating myapp_web_1
Creating myapp_web_1 ... done

$ docker-compose -p myapp down --volumes
Stopping myapp_redis_1 ... done
Stopping myapp_web_1   ... done
Removing myapp_redis_1 ... done
Removing myapp_web_1   ... done
Removing network myapp_default
```

COMPOSE_FILE

指定 `compose file` 的位置。

- 如果没有指定 `COMPOSE_FILE` ，`docker-compose` 会一次从 `当前目录` 向 `上级目录` 找 `docker-compose.yml` 文件，知道找到为止。
- `COMPOSE_FILE` 支持指定多个 `compose file`
  - `COMPOSE_FILE=docker-compose.yml:docker-compose.prod.yml`
  - `linux` / `macOS` 中使用 `:(冒号)` 分隔
  - `windows` 中使用 `;(分号)` 分隔。
  - 可以使用 `COMPOSE_PATH_SEPARATOR` 变量自定义上述的 `分隔符` `(: / ;)`
  - 或者使用 `多个` `-f` 实现多 `compose file` 的情形。

COMPOSE_PATH_SEPARATOR

为 `COMPOSE_FILE` 自定义多文件分隔符。

COMPOSE_API_VERSION

Docker API 只支持 `确定版本号` 的客户端请求。如果使用 `docker-compose` 是返回 `client and server don't have same version` ，可以通过修改环境变量的方式，设置 `docker api` 和 `client` 版本号一致。

如果 `docker-compose` 和 `docker api` 版本号不一致，可能引发未知异常。

DOCKER_HOST

Sets the `URL` of the `docker daemon` . As with the Docker client, defaults to `unix:///var/run/docker.sock` .

DOCKER_TLS_VERIFY

当 `DOCKER_TLS_VERIFY` 值为 `非空字符串` 时，表示 `docker daemon` 启用 `TLS` 通信。

DOCKER_CERT_PATH

为 `TLS` 验证指定证书路径。

- 默认值： `~/.docker`

证书包括：

- `ca.pem`
- `cert.pem`
- `key.pem`

COMPOSE_HTTP_TIMEOUT

设置 `docker-compose` 请求 `docker daemon` 的超时时间

- 单位 `秒` 。
- 默认值：60

COMPOSE_TLS_VERSION

为 `TLS` 通信指定 `TLS 版本`

- 默认值： `TLSv1`
- 可选值： `TLSv1` , `TLSv1_1` , `TLSv1_2`

COMPOSE_CONVERT_WINDOWS_PATHS

允许在定义 `volume` 时，将 `windows-style` 路径格式转换为 `Unix-style` 路径格式。Users of Docker Machine and Docker Toolbox on Windows should always set this.

- 默认值：0
- 可选值：
  - 启用： `true` , `1`
  - 禁用： `false` , `0`

docker-compose 命令行补全

基本上，在现在的 `linux` 发行版中 `bash-completion` 都默认安装了。

```
$ dpkg -l |grep bash
ii  bash                                4.3-14ubuntu1.2
                  amd64        GNU Bourne Again SHell
ii  bash-completion                     1:2.1-4.2ubuntu1.1
                  all          programmable completion for the b
ash shell
ii  command-not-found                   0.3ubuntu16.04.2
                  all          Suggest installation of packages
in interactive bash sessions
```

docker-compose file practice

- 001.nginx-sample
- 002.nginx-php-dokuwiki

Dockerfile practice

1. 尽量不是使用 `ADD`，使用 `COPY` 代替
    - 其实尽量使用 `RUN wget http://path/file.tar.gz ; do something ; rm -f file.tar.gz`
2. 删除没用的临时文件
    - **`ubuntu`** 缓存:
        - `apt-get install -y --no-install-recommends $packages`
        - `apt-get purge -y --auto-remove $packages`
        - `rm -rf /var/lib/apt/lists/*`
    - 删除临时文件规则
        - 类似于 `wget` 这样的辅助命令，或者包管理器的缓存
            - 在 `每一步` `RUN` 命令之前安装
            - 在 `每一步` `RUN` 命令之后删除
            - 否则，就会保留在该 `layer` 中
3. 将不变的内容放在前面
    - 前面的镜像内容变了，后面就要重新构建，而不能使用 `cache` 了。
4. `COPY`：将文件放入镜像的时候会保留当前的文件权限。
    - 因此，类似于 `entrypoint.sh` 这类执行文件，需要在 `build` 之间就赋予 `chmod +x` 的权限

docker-entrypoint.sh

这是 `redis` 官方的 `entrypoint.sh`。

dockerfile 中，

- 结尾的时候将 `ENTRYPOINT` 和 `CMD` 分开了。并且，在 `entrypoint.sh` 结尾使用 `exec $@` 命令。
    - 这样，用户在不添加任何参数的时候，可以默认执行 `redis-server`
    - 如果使用自定义个 `docker run --rm -it redis sh` 就可以进入命令行界面进行交互

```
ENTRYPOINT ["entrypoint.sh"]
CMD ["redis-server"]
```

```
#!/bin/sh
set -e

# first arg is `-f` or `--some-option`
# or first arg is `something.conf`
if [ "${1#-}" != "$1" ] || [ "${1%.conf}" != "$1" ]; then
    set -- redis-server "$@"
fi

# allow the container to be started with `--user`
if [ "$1" = 'redis-server' -a "$(id -u)" = '0' ]; then
    chown -R redis .
    exec gosu redis "$0" "$@"
fi

exec "$@"
```

v2ray

- v2ray-alpine
- v2ray-ubuntu

ubuntu 替换源

- ubuntu-soruces

redis 编译安装

- redis:ubuntu1604