**docker 学习文档**

# Docker 简介

## docker 是什么

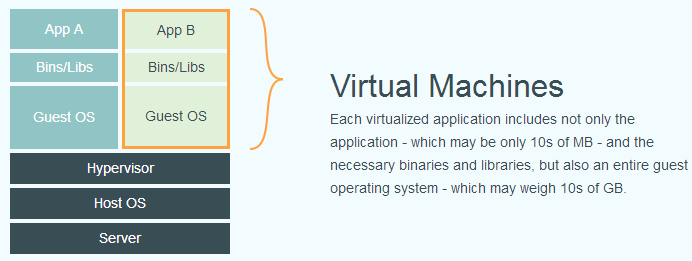
Docker 最初是 dotCloud 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目，它是基于 dotCloud 公司多年云服务技术的一次革新，并于 2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在 [GitHub](https://github.com/moby/moby) 上进行维护。Docker 项目后来还加入了 Linux 基金会，并成立推动 [开放容器联盟（OCI）](https://www.opencontainers.org/)。

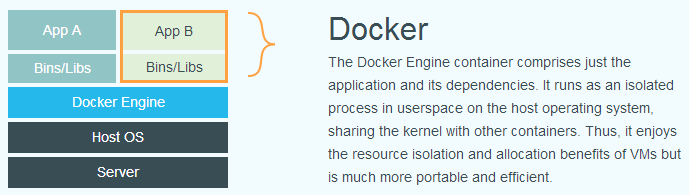
Docker 自开源后受到广泛的关注和讨论，至今其 GitHub 项目已经超过 4 万 6 千个星标和一万多个 fork。甚至由于 Docker 项目的火爆，在 2013 年底，dotCloud 公司决定改名为 Docker。Docker 最初是在 Ubuntu 12.04 上开发实现的；Red Hat 则从 RHEL 6.5 开始对 Docker 进行支持；Google 也在其 PaaS 产品中广泛应用 Docker。

Docker 使用 Google 公司推出的 Go 语言 进行开发实现，基于 Linux 内核的 cgroup，namespace，以及 AUFS 类的 Union FS 等技术，对进程进行封装隔离，属于 操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。最初实现是基于 LXC，从 0.7 版本以后开始去除 LXC，转而使用自行开发的 libcontainer，从 1.11 开始，则进一步演进为使用 runC 和 containerd。

Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护。使得 Docker 技术比虚拟机技术更为轻便、快捷。

下面的图片比较了 Docker 和传统虚拟化方式的不同之处。传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。





## 为什么要使用 docker

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

**更高效的利用系统资源**

由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，Docker 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

**更快速的启动时间**

传统的虚拟机技术启动应用服务往往需要数分钟，而 Docker 容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。

**一致的运行环境**

开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 bug 并未在开发过程中被发现。而 Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现 「这段代码在我机器上没问题啊」 这类问题。

**持续交付和部署**

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

使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。开发人员可以通过 Dockerfile 来进行镜像构建，并结合 持续集成(Continuous Integration) 系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜像，甚至结合 持续部署(Continuous Delivery/Deployment) 系统进行自动部署。

而且使用 Dockerfile 使镜像构建透明化，不仅仅开发团队可以理解应用运行环境，也方便运维团队理解应用运行所需条件，帮助更好的生产环境中部署该镜像。

**更轻松的迁移**

由于 Docker 确保了执行环境的一致性，使得应用的迁移更加容易。Docker 可以在很多平台上运行，无论是物理机、虚拟机、公有云、私有云，甚至是笔记本，其运行结果是一致的。因此用户可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

**更轻松的维护和扩展**

Docker 使用的分层存储以及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker 团队同各个开源项目团队一起维护了一大批高质量的 官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

**对比传统虚拟机总结**

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

# docker 基本概念

Docker 包括三个基本概念

* 镜像（Image）
* 容器（Container）
* 仓库（Repository）

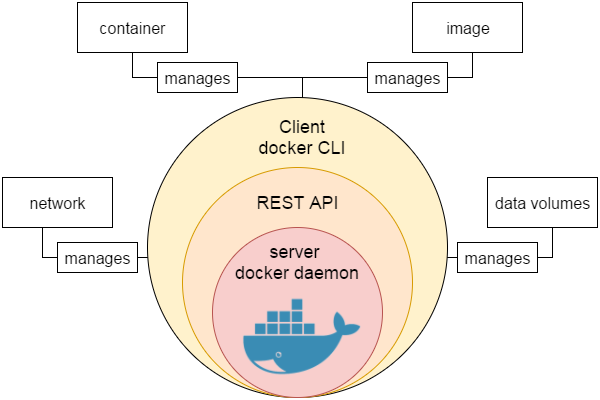
理解了这三个概念，就理解了 Docker 的整个生命周期。

## docker 引擎

Docker 引擎是一个包含以下主要组件的客户端服务器应用程序。

* 一种服务器，它是一种称为守护进程并且长时间运行的程序。
* REST API用于指定程序可以用来与守护进程通信的接口，并指示它做什么。
* 一个有命令行界面 (CLI) 工具的客户端。

Docker 引擎组件的流程如下图所示：



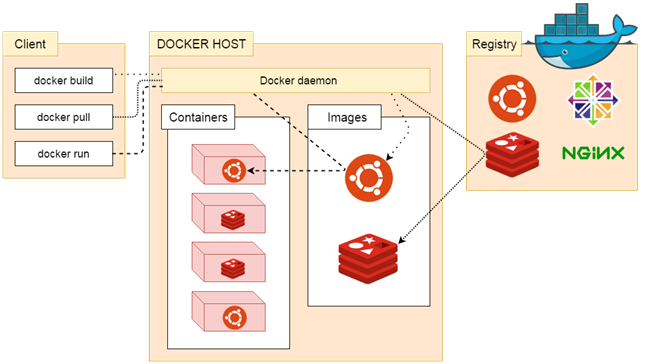
## docker 系统架构

Docker 使用客户端-服务器 (C/S) 架构模式，使用远程 API 来管理和创建 Docker 容器。

Docker 容器通过 Docker 镜像来创建。

容器与镜像的关系类似于面向对象编程中的对象与类。

|  |  |
| --- | --- |
| **Docker** | **面向对象** |
| 容器 | 对象 |
| 镜像 | 类 |



|  |  |
| --- | --- |
| **标题** | **说明** |
| 镜像(Images) | Docker 镜像是用于创建 Docker 容器的模板。 |
| 容器(Container) | 容器是独立运行的一个或一组应用。 |
| 客户端(Client) | Docker 客户端通过命令行或者其他工具使用 Docker API (https://docs.docker.com/reference/api/docker\_remote\_api) 与 Docker 的守护进程通信。 |
| 主机(Host) | 一个物理或者虚拟的机器用于执行 Docker 守护进程和容器。 |
| 仓库(Registry) | Docker 仓库用来保存镜像，可以理解为代码控制中的代码仓库。Docker Hub(https://hub.docker.com) 提供了庞大的镜像集合供使用。 |
| Docker Machine | Docker Machine是一个简化Docker安装的命令行工具，通过一个简单的命令行即可在相应的平台上安装Docker，比如VirtualBox、 Digital Ocean、Microsoft Azure。 |

## docker 镜像

我们都知道，操作系统分为内核和用户空间。对于 Linux 而言，内核启动后，会挂载 root 文件系统为其提供用户空间支持。而 Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu 16.04 最小系统的 root 文件系统。

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像不包含任何动态数据，其内容在构建之后也不会被改变。

**分层存储**

因为镜像包含操作系统完整的 root 文件系统，其体积往往是庞大的，因此在 Docker 设计时，就充分利用 Union FS 的技术，将其设计为分层存储的架构。所以严格来说，镜像并非是像一个 ISO 那样的打包文件，镜像只是一个虚拟的概念，其实际体现并非由一个文件组成，而是由一组文件系统组成，或者说，由多层文件系统联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

关于镜像构建，将会在后续相关章节中做进一步的讲解。

## docker 容器

镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的 类 和 实例 一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的 命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。也因为这种隔离的特性，很多人初学 Docker 时常常会混淆容器和虚拟机。

前面讲过镜像使用的是分层存储，容器也是如此。每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层，我们可以称这个为容器运行时读写而准备的存储层为容器存储层。

容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。

按照 Docker 最佳实践的要求，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用 数据卷（Volume）、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主（或网络存储）发生读写，其性能和稳定性更高。

数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡。因此，使用数据卷后，容器删除或者重新运行之后，数据却不会丢失。

## docker 仓库

镜像构建完成后，可以很容易的在当前宿主机上运行，但是，如果需要在其它服务器上使用这个镜像，我们就需要一个集中的存储、分发镜像的服务，Docker Registry 就是这样的服务。

一个 Docker Registry 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。

通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 <仓库名>:<标签> 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 latest 作为默认标签。

以 Ubuntu 镜像 为例，ubuntu 是仓库的名字，其内包含有不同的版本标签，如，14.04, 16.04。我们可以通过 ubuntu:14.04，或者 ubuntu:16.04 来具体指定所需哪个版本的镜像。如果忽略了标签，比如 ubuntu，那将视为 ubuntu:latest。

仓库名经常以 两段式路径 形式出现，比如 jwilder/nginx-proxy，前者往往意味着 Docker Registry 多用户环境下的用户名，后者则往往是对应的软件名。但这并非绝对，取决于所使用的具体 Docker Registry 的软件或服务。

### 公有 Docker Registry

Docker Registry 公开服务是开放给用户使用、允许用户管理镜像的 Registry 服务。一般这类公开服务允许用户免费上传、下载公开的镜像，并可能提供收费服务供用户管理私有镜像。

最常使用的 Registry 公开服务是官方的 [Docker Hub](https://hub.docker.com/)，这也是默认的 Registry，并拥有大量的高质量的官方镜像。除此以外，还有 [CoreOS](https://coreos.com/) 的 [Quay.io](https://quay.io/repository/)，CoreOS 相关的镜像存储在这里；Google 的 [Google Container Registry](https://cloud.google.com/container-registry/)，Kubernetes 的镜像使用的就是这个服务。

由于某些原因，在国内访问这些服务可能会比较慢。国内的一些云服务商提供了针对 Docker Hub 的镜像服务（Registry Mirror），这些镜像服务被称为加速器。常见的有 [阿里云加速器](https://cr.console.aliyun.com/" \l "/accelerator)、[DaoCloud 加速器](https://www.daocloud.io/mirror" \l "accelerator-doc) 等。使用加速器会直接从国内的地址下载 Docker Hub 的镜像，比直接从 Docker Hub 下载速度会提高很多。

国内也有一些云服务商提供类似于 Docker Hub 的公开服务。比如 [时速云镜像仓库](https://hub.tenxcloud.com/)、[网易云镜像服务](https://c.163.com/hub" \l "/m/library/)、[DaoCloud 镜像市场](https://hub.daocloud.io/)、[阿里云镜像库](https://cr.console.aliyun.com/) 等。

### 私有 Docker Registry

除了使用公开服务外，用户还可以在本地搭建私有 Docker Registry。Docker 官方提供了 [Docker Registry](https://store.docker.com/images/registry/) 镜像，可以直接使用做为私有 Registry 服务。

开源的 Docker Registry 镜像只提供了 [Docker Registry API](https://docs.docker.com/registry/spec/api/) 的服务端实现，足以支持 docker 命令，不影响使用。但不包含图形界面，以及镜像维护、用户管理、访问控制等高级功能。在官方的商业化版本 [Docker Trusted Registry](https://docs.docker.com/datacenter/dtr/2.0/) 中，提供了这些高级功能。

除了官方的 Docker Registry 外，还有第三方软件实现了 Docker Registry API，甚至提供了用户界面以及一些高级功能。比如，[VMWare Harbor](https://github.com/vmware/harbor) 和 [Sonatype Nexus](https://www.sonatype.com/docker)。

# docker 安装

Docker 在 1.13 版本之后，从 2017 年的 3 月 1 日开始，版本命名规则变为如下：

|  |  |
| --- | --- |
| 项目 | 说明 |
| 版本格式 | YY.MM |
| Stable 版本 | 每个季度发行 |
| Edge 版本 | 每个月发行 |

同时 Docker 划分为 CE 和 EE。CE 即社区版（免费，支持周期三个月），EE 即企业版，强调安全，付费使用。

Docker CE 每月发布一个 Edge 版本 (17.03, 17.04, 17.05...)，每三个月发布一个 Stable 版本 (17.03, 17.06, 17.09...)，Docker EE 和 Stable 版本号保持一致，但每个版本提供一年维护。

官方网站上有各种环境下的 [安装指南](https://docs.docker.com/engine/installation/)，这里主要介绍 Docker CE 在 Linux 、Windows 10 (PC) 和 macOS 上的安装。

## ubuntu 安装docker

警告：切勿在没有配置 Docker APT 源的情况下直接使用 apt 命令安装 Docker.

### **准备工作**

**1、系统要求**

Docker CE 支持以下版本的 Ubuntu 操作系统：

* Artful 17.10 (Docker CE 17.11 Edge +)
* Xenial 16.04 (LTS)
* Trusty 14.04 (LTS)

Docker CE 可以安装在 64 位的 x86 平台或 ARM 平台上。Ubuntu 发行版中，LTS（Long-Term-Support）长期支持版本，会获得 5 年的升级维护支持，这样的版本会更稳定，因此在生产环境中推荐使用 LTS 版本,当前最新的 LTS 版本为 Ubuntu 16.04。

**2、卸载旧版本**

旧版本的 Docker 称为 docker 或者 docker-engine，使用以下命令卸载旧版本：

sudo apt-get remove docker docker-engine docker.io

**3、Ubuntu 14.04 可选内核模块**

从 Ubuntu 14.04 开始，一部分内核模块移到了可选内核模块包 (linux-image-extra-\*) ，以减少内核软件包的体积。正常安装的系统应该会包含可选内核模块包，而一些裁剪后的系统可能会将其精简掉。AUFS 内核驱动属于可选内核模块的一部分，作为推荐的 Docker 存储层驱动，一般建议安装可选内核模块包以使用 AUFS。

如果系统没有安装可选内核模块的话，可以执行下面的命令来安装可选内核模块包：

sudo apt-get update

sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual

**4、Ubuntu 16.04 +**

Ubuntu 16.04 + 上的 Docker CE 默认使用 overlay2 存储层驱动,无需手动配置。

### **使用 APT 安装**

安装必要的一些系统工具

sudo apt-get update

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

安装 GPG 证书

curl -fsSL http://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -

写入软件源信息

sudo add-apt-repository "deb [arch=amd64] http://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb\_release -cs) stable"

更新并安装 Docker CE

sudo apt-get -y update

sudo apt-get -y install docker-ce

以上命令会添加稳定版本的 Docker CE APT 镜像源，如果需要最新或者测试版本的 Docker CE 请将 stable 改为 edge 或者 test。从 Docker 17.06 开始，edge test 版本的 APT 镜像源也会包含稳定版本的 Docker。

### **使用脚本自动安装**

在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，Ubuntu 系统上可以使用这套脚本安装：

curl -fsSL get.docker.com -o get-docker.sh

# 可能会出现 404 错误，请移步下面的特别说明

sudo sh get-docker.sh --mirror Aliyun

执行这个命令后，脚本就会自动的将一切准备工作做好，并且把 Docker CE 的 Edge 版本安装在系统中。

**特别说明**

2018 年 7 月 21 日，貌似阿里云这边在做调整，故导致 Docker 的 Aliyun 安装脚本不可用，是永久性还是临时性的尚不清除，如果你已经按照之前的操作安装 Docker，请按以下步骤进行修复并重新安装

* 如果已经使用了 Aliyun 脚本安装并成功的
* 请先卸载 Docker，命令为：apt-get autoremove docker-ce
* 删除 /etc/apt/sources.list.d 目录下的 docker.list 文件
* 使用 AzureChinaCloud 镜像脚本重新安装，命令为：sudo sh get-docker.sh --mirror AzureChinaCloud

### 启动 Docker CE

sudo systemctl enable docker

sudo systemctl start docker

Ubuntu 14.04 请使用以下命令启动：

sudo service docker start

### 建立 docker 用户组

默认情况下，docker 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 root 用户和 docker 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 root 用户。因此，更好地做法是将需要使用 docker 的用户加入 docker 用户组。

建立 docker 组：

sudo groupadd docker

将当前用户加入 docker 组：

sudo usermod -aG docker $USER

退出当前终端并重新登录，进行如下测试。

### 测试 Docker 是否安装正确

docker run hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

ca4f61b1923c: Pull complete

Digest: sha256:be0cd392e45be79ffeffa6b05338b98ebb16c87b255f48e297ec7f98e123905c

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

(amd64)

3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

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

$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:

https://cloud.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/engine/userguide/

若能正常输出以上信息，则说明安装成功。

### 镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速。

### 参考文档

[Docker 官方 Ubuntu 安装文档](https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/)

## centOS 安装 docker

警告：切勿在没有配置 Docker YUM 源的情况下直接使用 yum 命令安装 Docker.

### 准备工作

**1、系统要求**

Docker CE 支持 64 位版本 CentOS 7，并且要求内核版本不低于 3.10。 CentOS 7 满足最低内核的要求，但由于内核版本比较低，部分功能（如 overlay2 存储层驱动）无法使用，并且部分功能可能不太稳定。

**2、卸载旧版本**

旧版本的 Docker 称为 docker 或者 docker-engine，使用以下命令卸载旧版本：

sudo yum remove docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-selinux \

docker-engine-selinux \

docker-engine

### 使用 yum 安装

执行以下命令安装依赖包：

sudo yum install -y yum-utils \

device-mapper-persistent-data \

lvm2

鉴于国内网络问题，强烈建议使用国内源，官方源请在注释中查看。

执行下面的命令添加 yum 软件源：

sudo yum-config-manager \

--add-repo \

https://mirrors.ustc.edu.cn/docker-ce/linux/centos/docker-ce.repo

# 官方源

# sudo yum-config-manager \

# --add-repo \

# https://download.docker.com/linux/centos/docker-ce.repo

如果需要最新版本的 Docker CE 请使用以下命令：

sudo yum-config-manager --enable docker-ce-edge

如果需要测试版本的 Docker CE 请使用以下命令：

sudo yum-config-manager --enable docker-ce-test

**安装 Docker CE**

更新 yum 软件源缓存，并安装 docker-ce。

sudo yum makecache fast

sudo yum install docker-ce

### 使用脚本自动安装

在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，CentOS 系统上可以使用这套脚本安装：

curl -fsSL get.docker.com -o get-docker.sh

sudo sh get-docker.sh --mirror Aliyun

执行这个命令后，脚本就会自动的将一切准备工作做好，并且把 Docker CE 的 Edge 版本安装在系统中。

### 启动 Docker CE

sudo systemctl enable docker

sudo systemctl start docker

### 建立 docker 用户组

默认情况下，docker 命令会使用 Unix socket 与 Docker 引擎通讯。而只有 root 用户和 docker 组的用户才可以访问 Docker 引擎的 Unix socket。出于安全考虑，一般 Linux 系统上不会直接使用 root 用户。因此，更好地做法是将需要使用 docker 的用户加入 docker 用户组。

建立 docker 组：

sudo groupadd docker

将当前用户加入 docker 组：

sudo usermod -aG docker $USER

退出当前终端并重新登录，进行如下测试。

### 测试 Docker 是否安装正确

docker run hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

ca4f61b1923c: Pull complete

Digest: sha256:be0cd392e45be79ffeffa6b05338b98ebb16c87b255f48e297ec7f98e123905c

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

(amd64)

3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

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

$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:

https://cloud.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/engine/userguide/

若能正常输出以上信息，则说明安装成功。

### 镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速。

### 添加内核参数

默认配置下，如果在 CentOS 使用 Docker CE 看到下面的这些警告信息：

WARNING: bridge-nf-call-iptables is disabled

WARNING: bridge-nf-call-ip6tables is disabled

请添加内核配置参数以启用这些功能。

sudo tee -a /etc/sysctl.conf <<-EOF

net.bridge.bridge-nf-call-ip6tables = 1

net.bridge.bridge-nf-call-iptables = 1

EOF

然后重新加载 sysctl.conf 即可

sudo sysctl -p

### 参考文档

[Docker 官方 CentOS 安装文档](https://docs.docker.com/engine/installation/linux/docker-ce/centos/)。

## mac 安装 docker

### 系统要求

[Docker for Mac](https://docs.docker.com/docker-for-mac/) 要求系统最低为 macOS 10.10.3 Yosemite。如果系统不满足需求，可以安装 [Docker Toolbox](https://docs.docker.com/toolbox/overview/)。

### 安装

**1、使用 Homebrew 安装**

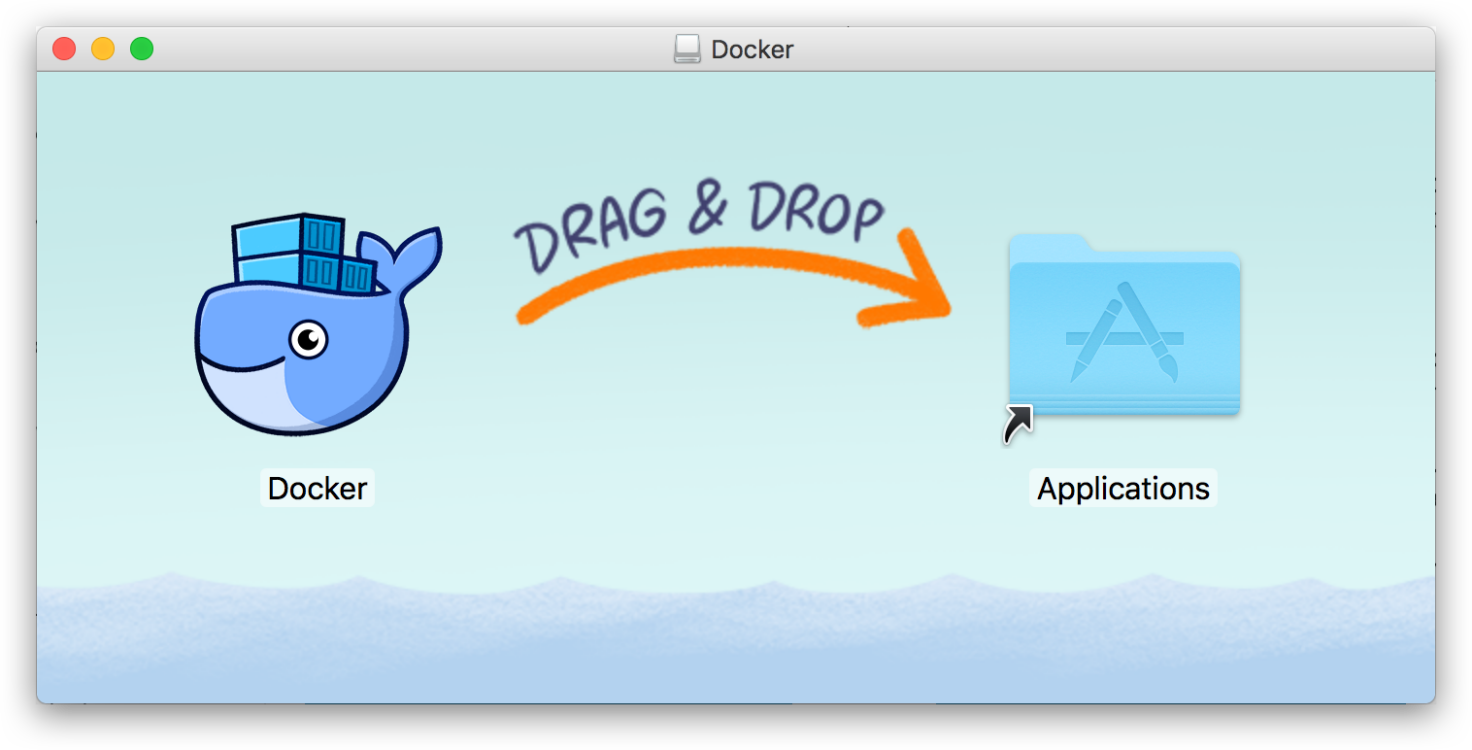
Homebrew 的 Cask 已经支持 Docker for Mac，因此可以很方便的使用 Homebrew Cask 来进行安装：

brew cask install docker

**2、手动下载安装**

如果需要手动下载，请点击以下链接下载 Stable 或 Edge 版本的 Docker for Mac。

如同 macOS 其它软件一样，安装也非常简单，双击下载的 .dmg 文件，然后将那只叫 Moby 的鲸鱼图标拖拽到 Application 文件夹即可（其间需要输入用户密码）。



### 运行

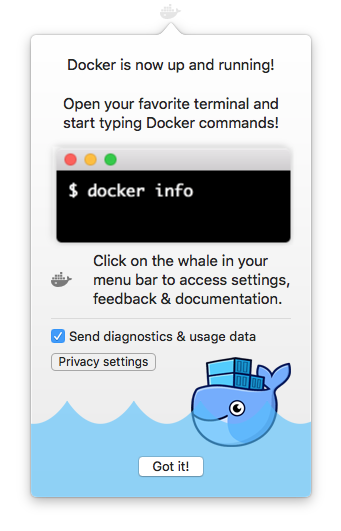
从应用中找到 Docker 图标并点击运行。



运行之后，会在右上角菜单栏看到多了一个鲸鱼图标，这个图标表明了 Docker 的运行状态。



第一次点击图标，可能会看到这个安装成功的界面，点击 "Got it!" 可以关闭这个窗口。



以后每次点击鲸鱼图标会弹出操作菜单。



启动终端后，通过命令可以检查安装后的 Docker 版本。

docker --version

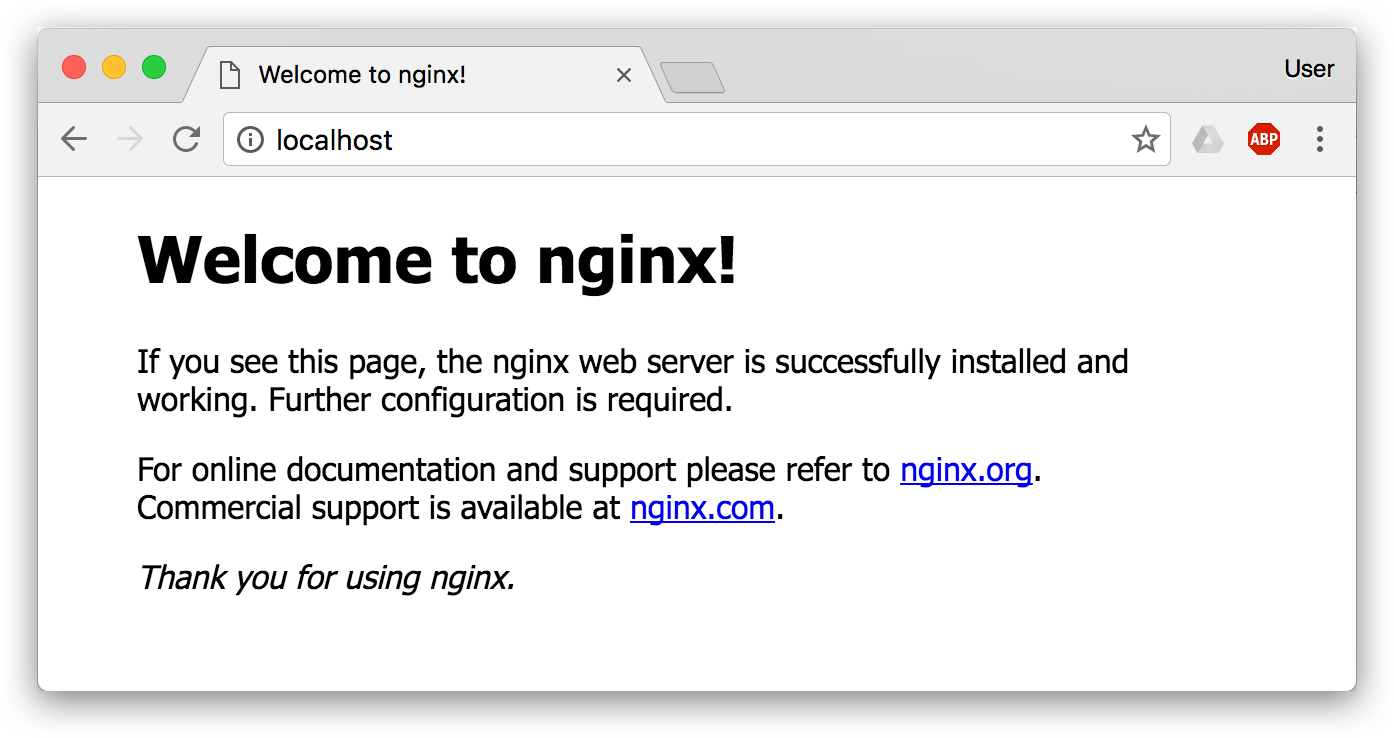
docker-compose --version

docker-machine --version

如果 docker version、docker info 都正常的话，可以尝试运行一个 Nginx 服务器：

docker run -d -p 80:80 --name webserver nginx

服务运行后，可以访问 http://localhost，如果看到了 "Welcome to nginx!"，就说明 Docker for Mac 安装成功了。



要停止 Nginx 服务器并删除执行下面的命令：

docker stop webserver

docker rm webserver

### 镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速。

## windos 安装 docker

### 系统要求

[Docker for Windows](https://docs.docker.com/docker-for-windows/install/) 支持 64 位版本的 Windows 10 Pro，且必须开启 Hyper-V。

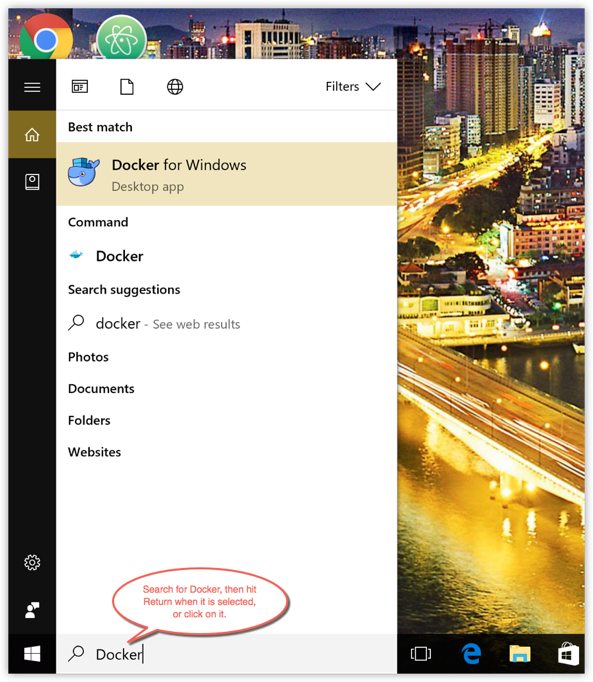
### 安装

点击以下链接下载 Stable 或 Edge 版本的 Docker for Windows。

下载好之后双击 Docker for Windows Installer.exe 开始安装。

### 运行

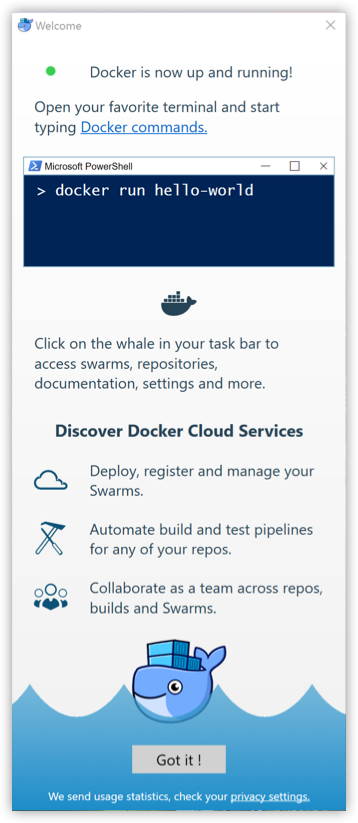
在 Windows 搜索栏输入 Docker 点击 Docker for Windows 开始运行。



Docker CE 启动之后会在 Windows 任务栏出现鲸鱼图标。

https://www.funtl.com/assets/install-win-taskbar-circle.png

等待片刻，点击 Got it 开始使用 Docker CE。



### 镜像加速

鉴于国内网络问题，后续拉取 Docker 镜像十分缓慢，强烈建议安装 Docker 之后配置 国内镜像加速。

## 镜像加速

国内从 Docker Hub 拉取镜像有时会遇到困难，此时可以配置镜像加速器。Docker 官方和国内很多云服务商都提供了国内加速器服务，例如：

* Docker 官方提供的中国 registry mirror
* 阿里云加速器
* DaoCloud 加速器

我们以 Docker 官方加速器为例进行介绍。

### Ubuntu 14.04、Debian 7 Wheezy

对于使用 upstart 的系统而言，编辑 /etc/default/docker 文件，在其中的 DOCKER\_OPTS 中配置加速器地址：

DOCKER\_OPTS="--registry-mirror=https://registry.docker-cn.com"

重新启动服务。

sudo service docker restart

### Ubuntu 16.04+、Debian 8+、CentOS 7

对于使用 systemd 的系统，请在 /etc/docker/daemon.json 中写入如下内容（如果文件不存在请新建该文件）

{

"registry-mirrors": [

"https://registry.docker-cn.com"

]

}

注意，一定要保证该文件符合 json 规范，否则 Docker 将不能启动。

之后重新启动服务。

sudo systemctl daemon-reload

sudo systemctl restart docker

注意：如果您之前查看旧教程，修改了 docker.service 文件内容，请去掉您添加的内容（--registry-mirror=https://registry.docker-cn.com），这里不再赘述。

### Windows 10

对于使用 Windows 10 的系统，在系统右下角托盘 Docker 图标内右键菜单选择 Settings，打开配置窗口后左侧导航菜单选择 Daemon。在 Registry mirrors 一栏中填写加速器地址 https://registry.docker-cn.com，之后点击 Apply 保存后 Docker 就会重启并应用配置的镜像地址了。

### macOS

对于使用 macOS 的用户，在任务栏点击 Docker for mac 应用图标 -> Perferences... -> Daemon -> Registry mirrors。在列表中填写加速器地址 https://registry.docker-cn.com。修改完成之后，点击 Apply & Restart 按钮，Docker 就会重启并应用配置的镜像地址了。

### 检查加速器是否生效

配置加速器之后，如果拉取镜像仍然十分缓慢，请手动检查加速器配置是否生效，在命令行执行 docker info，如果从结果中看到了如下内容，说明配置成功。

Registry Mirrors:

https://registry.docker-cn.com/

# docker 镜像

在之前的介绍中，我们知道镜像是 Docker 的三大组件之一。

Docker 运行容器前需要本地存在对应的镜像，如果本地不存在该镜像，Docker 会从镜像仓库下载该镜像。

本章将介绍更多关于镜像的内容，包括：

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

## Docker 获取镜像

之前提到过，Docker Hub 上有大量的高质量的镜像可以用，这里我们就说一下怎么获取这些镜像。

从 Docker 镜像仓库获取镜像的命令是 docker pull。其命令格式为：

docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]

具体的选项可以通过 docker pull --help 命令看到，这里我们说一下镜像名称的格式。

* Docker 镜像仓库地址：地址的格式一般是 <域名/IP>[:端口号]。默认地址是 Docker Hub。
* 仓库名：如之前所说，这里的仓库名是两段式名称，即 <用户名>/<软件名>。对于 Docker Hub，如果不给出用户名，则默认为 library，也就是官方镜像。

比如：

docker pull ubuntu:16.04

16.04: Pulling from library/ubuntu

bf5d46315322: Pull complete

9f13e0ac480c: Pull complete

e8988b5b3097: Pull complete

40af181810e7: Pull complete

e6f7c7e5c03e: Pull complete

Digest: sha256:147913621d9cdea08853f6ba9116c2e27a3ceffecf3b492983ae97c3d643fbbe

Status: Downloaded newer image for ubuntu:16.04

上面的命令中没有给出 Docker 镜像仓库地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 ubuntu:16.04，因此将会获取官方镜像 library/ubuntu 仓库中标签为 16.04 的镜像。

从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储所构成。下载也是一层层的去下载，并非单一文件。下载过程中给出了每一层的 ID 的前 12 位。并且下载结束后，给出该镜像完整的 sha256 的摘要，以确保下载一致性。

在使用上面命令的时候，你可能会发现，你所看到的层 ID 以及 sha256 的摘要和这里的不一样。这是因为官方镜像是一直在维护的，有任何新的 bug，或者版本更新，都会进行修复再以原来的标签发布，这样可以确保任何使用这个标签的用户可以获得更安全、更稳定的镜像。

如果从 Docker Hub 下载镜像非常缓慢，可以参照 镜像加速器 一节配置加速器。

### 运行

有了镜像后，我们就能够以这个镜像为基础启动并运行一个容器。以上面的 ubuntu:16.04 为例，如果我们打算启动里面的 bash 并且进行交互式操作的话，可以执行下面的命令。

docker run -it --rm \

ubuntu:16.04 \

bash

root@e7009c6ce357:/# cat /etc/os-release

NAME="Ubuntu"

VERSION="16.04.4 LTS, Trusty Tahr"

ID=ubuntu

ID\_LIKE=debian

PRETTY\_NAME="Ubuntu 16.04.4 LTS"

VERSION\_ID="16.04"

HOME\_URL="http://www.ubuntu.com/"

SUPPORT\_URL="http://help.ubuntu.com/"

BUG\_REPORT\_URL="http://bugs.launchpad.net/ubuntu/"

docker run 就是运行容器的命令，我们这里简要的说明一下上面用到的参数。

* -it：这是两个参数，一个是 -i：交互式操作，一个是 -t 终端。我们这里打算进入 bash 执行一些命令并查看返回结果，因此我们需要交互式终端。
* --rm：这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 docker rm。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 --rm 可以避免浪费空间。
* ubuntu:16.04：这是指用 ubuntu:16.04 镜像为基础来启动容器。
* bash：放在镜像名后的是命令，这里我们希望有个交互式 Shell，因此用的是 bash。

进入容器后，我们可以在 Shell 下操作，执行任何所需的命令。这里，我们执行了 cat /etc/os-release，这是 Linux 常用的查看当前系统版本的命令，从返回的结果可以看到容器内是 Ubuntu 16.04.4 LTS 系统。

最后我们通过 exit 退出了这个容器。

## docker 列出镜像

Docker 列出镜像

要想列出已经下载下来的镜像，可以使用 docker image ls 命令。

docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

redis latest 5f515359c7f8 5 days ago 183 MB

nginx latest 05a60462f8ba 5 days ago 181 MB

mongo 3.2 fe9198c04d62 5 days ago 342 MB

<none> <none> 00285df0df87 5 days ago 342 MB

ubuntu 16.04 f753707788c5 4 weeks ago 127 MB

ubuntu latest f753707788c5 4 weeks ago 127 MB

ubuntu 14.04 1e0c3dd64ccd 4 weeks ago 188 MB

列表包含了 仓库名、标签、镜像 ID、创建时间 以及 所占用的空间。

其中仓库名、标签在之前的基础概念章节已经介绍过了。镜像 ID 则是镜像的唯一标识，一个镜像可以对应多个标签。因此，在上面的例子中，我们可以看到 ubuntu:16.04 和 ubuntu:latest 拥有相同的 ID，因为它们对应的是同一个镜像。

### 镜像体积

如果仔细观察，会注意到，这里标识的所占用空间和在 Docker Hub 上看到的镜像大小不同。比如，ubuntu:16.04 镜像大小，在这里是 127 MB，但是在 Docker Hub 显示的却是 50 MB。这是因为 Docker Hub 中显示的体积是压缩后的体积。在镜像下载和上传过程中镜像是保持着压缩状态的，因此 Docker Hub 所显示的大小是网络传输中更关心的流量大小。而 docker image ls 显示的是镜像下载到本地后，展开的大小，准确说，是展开后的各层所占空间的总和，因为镜像到本地后，查看空间的时候，更关心的是本地磁盘空间占用的大小。

另外一个需要注意的问题是，docker image ls 列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于 Docker 镜像是多层存储结构，并且可以继承、复用，因此不同镜像可能会因为使用相同的基础镜像，从而拥有共同的层。由于 Docker 使用 Union FS，相同的层只需要保存一份即可，因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小的多。

你可以通过以下命令来便捷的查看镜像、容器、数据卷所占用的空间。

docker system df

TYPE TOTAL ACTIVE SIZE RECLAIMABLE

Images 24 0 1.992GB 1.992GB (100%)

Containers 1 0 62.82MB 62.82MB (100%)

Local Volumes 9 0 652.2MB 652.2MB (100%)

Build Cache 0B 0B

### 虚悬镜像

上面的镜像列表中，还可以看到一个特殊的镜像，这个镜像既没有仓库名，也没有标签，均为 <none>。：

<none> <none> 00285df0df87 5 days ago 342 MB

这个镜像原本是有镜像名和标签的，原来为 mongo:3.2，随着官方镜像维护，发布了新版本后，重新 docker pull mongo:3.2 时，mongo:3.2 这个镜像名被转移到了新下载的镜像身上，而旧的镜像上的这个名称则被取消，从而成为了 <none>。除了 docker pull 可能导致这种情况，docker build 也同样可以导致这种现象。由于新旧镜像同名，旧镜像名称被取消，从而出现仓库名、标签均为 <none> 的镜像。这类无标签镜像也被称为 虚悬镜像(dangling image) ，可以用下面的命令专门显示这类镜像：

docker image ls -f dangling=true

REPOSITORY TAG IMAGE ID CREATED SIZE

<none> <none> 00285df0df87 5 days ago 342 MB

一般来说，虚悬镜像已经失去了存在的价值，是可以随意删除的，可以用下面的命令删除。

docker image prune

### 中间层镜像

为了加速镜像构建、重复利用资源，Docker 会利用 中间层镜像。所以在使用一段时间后，可能会看到一些依赖的中间层镜像。默认的 docker image ls 列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像的话，需要加 -a 参数。

docker image ls -a

这样会看到很多无标签的镜像，与之前的虚悬镜像不同，这些无标签的镜像很多都是中间层镜像，是其它镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍，而这些镜像是别的镜像的依赖，因此并不会因为它们被列出来而多存了一份，无论如何你也会需要它们。只要删除那些依赖它们的镜像后，这些依赖的中间层镜像也会被连带删除。

### 列出部分镜像

不加任何参数的情况下，docker image ls 会列出所有顶级镜像，但是有时候我们只希望列出部分镜像。docker image ls 有好几个参数可以帮助做到这个事情。

根据仓库名列出镜像

docker image ls ubuntu

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu 16.04 f753707788c5 4 weeks ago 127 MB

ubuntu latest f753707788c5 4 weeks ago 127 MB

ubuntu 14.04 1e0c3dd64ccd 4 weeks ago 188 MB

列出特定的某个镜像，也就是说指定仓库名和标签

docker image ls ubuntu:16.04

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu 16.04 f753707788c5 4 weeks ago 127 MB

除此以外，docker image ls 还支持强大的过滤器参数 --filter，或者简写 -f。之前我们已经看到了使用过滤器来列出虚悬镜像的用法，它还有更多的用法。比如，我们希望看到在 mongo:3.2 之后建立的镜像，可以用下面的命令：

docker image ls -f since=mongo:3.2

REPOSITORY TAG IMAGE ID CREATED SIZE

redis latest 5f515359c7f8 5 days ago 183 MB

nginx latest 05a60462f8ba 5 days ago 181 MB

想查看某个位置之前的镜像也可以，只需要把 since 换成 before 即可。

此外，如果镜像构建时，定义了 LABEL，还可以通过 LABEL 来过滤。

docker image ls -f label=com.example.version=0.1

### 以特定格式显示

默认情况下，docker image ls 会输出一个完整的表格，但是我们并非所有时候都会需要这些内容。比如，刚才删除虚悬镜像的时候，我们需要利用 docker image ls 把所有的虚悬镜像的 ID 列出来，然后才可以交给 docker image rm 命令作为参数来删除指定的这些镜像，这个时候就用到了 -q 参数。

docker image ls -q

5f515359c7f8

05a60462f8ba

fe9198c04d62

00285df0df87

f753707788c5

f753707788c5

1e0c3dd64ccd

--filter 配合 -q 产生出指定范围的 ID 列表，然后送给另一个 docker 命令作为参数，从而针对这组实体成批的进行某种操作的做法在 Docker 命令行使用过程中非常常见，不仅仅是镜像，将来我们会在各个命令中看到这类搭配以完成很强大的功能。因此每次在文档看到过滤器后，可以多注意一下它们的用法。

另外一些时候，我们可能只是对表格的结构不满意，希望自己组织列；或者不希望有标题，这样方便其它程序解析结果等，这就用到了 Go 的模板语法。

比如，下面的命令会直接列出镜像结果，并且只包含镜像ID和仓库名：

docker image ls --format "{{.ID}}: {{.Repository}}"

5f515359c7f8: redis

05a60462f8ba: nginx

fe9198c04d62: mongo

00285df0df87: <none>

f753707788c5: ubuntu

f753707788c5: ubuntu

1e0c3dd64ccd: ubuntu

或者打算以表格等距显示，并且有标题行，和默认一样，不过自己定义列：

docker image ls --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"

IMAGE ID REPOSITORY TAG

5f515359c7f8 redis latest

05a60462f8ba nginx latest

fe9198c04d62 mongo 3.2

00285df0df87 <none> <none>

f753707788c5 ubuntu 16.04

f753707788c5 ubuntu latest

1e0c3dd64ccd ubuntu 14.04

## docker 删除本地镜像

如果要删除本地的镜像，可以使用 docker image rm 命令，其格式为：

docker image rm [选项] <镜像1> [<镜像2> ...]

### 用 ID、镜像名、摘要删除镜像

其中，<镜像> 可以是 镜像短 ID、镜像长 ID、镜像名 或者 镜像摘要。

比如我们有这么一些镜像：

docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

centos latest 0584b3d2cf6d 3 weeks ago 196.5 MB

redis alpine 501ad78535f0 3 weeks ago 21.03 MB

docker latest cf693ec9b5c7 3 weeks ago 105.1 MB

nginx latest e43d811ce2f4 5 weeks ago 181.5 MB

我们可以用镜像的完整 ID，也称为 长 ID，来删除镜像。使用脚本的时候可能会用长 ID，但是人工输入就太累了，所以更多的时候是用 短 ID 来删除镜像。docker image ls 默认列出的就已经是短 ID 了，一般取前3个字符以上，只要足够区分于别的镜像就可以了。

比如这里，如果我们要删除 redis:alpine 镜像，可以执行：

docker image rm 501

Untagged: redis:alpine

Untagged: redis@sha256:f1ed3708f538b537eb9c2a7dd50dc90a706f7debd7e1196c9264edeea521a86d

Deleted: sha256:501ad78535f015d88872e13fa87a828425117e3d28075d0c117932b05bf189b7

Deleted: sha256:96167737e29ca8e9d74982ef2a0dda76ed7b430da55e321c071f0dbff8c2899b

Deleted: sha256:32770d1dcf835f192cafd6b9263b7b597a1778a403a109e2cc2ee866f74adf23

Deleted: sha256:127227698ad74a5846ff5153475e03439d96d4b1c7f2a449c7a826ef74a2d2fa

Deleted: sha256:1333ecc582459bac54e1437335c0816bc17634e131ea0cc48daa27d32c75eab3

Deleted: sha256:4fc455b921edf9c4aea207c51ab39b10b06540c8b4825ba57b3feed1668fa7c7

我们也可以用镜像名，也就是 <仓库名>:<标签>，来删除镜像。

docker image rm centos

Untagged: centos:latest

Untagged: centos@sha256:b2f9d1c0ff5f87a4743104d099a3d561002ac500db1b9bfa02a783a46e0d366c

Deleted: sha256:0584b3d2cf6d235ee310cf14b54667d889887b838d3f3d3033acd70fc3c48b8a

Deleted: sha256:97ca462ad9eeae25941546209454496e1d66749d53dfa2ee32bf1faabd239d38

当然，更精确的是使用 镜像摘要 删除镜像。

docker image ls --digests

REPOSITORY TAG DIGEST IMAGE ID CREATED SIZE

node slim sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228 6e0c4c8e3913 3 weeks ago 214 MB

docker image rm node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228

Untagged: node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228

### Untagged 和 Deleted

如果观察上面这几个命令的运行输出信息的话，你会注意到删除行为分为两类，一类是 Untagged，另一类是 Deleted。我们之前介绍过，镜像的唯一标识是其 ID 和摘要，而一个镜像可以有多个标签。

因此当我们使用上面命令删除镜像的时候，实际上是在要求删除某个标签的镜像。所以首先需要做的是将满足我们要求的所有镜像标签都取消，这就是我们看到的 Untagged 的信息。因为一个镜像可以对应多个标签，因此当我们删除了所指定的标签后，可能还有别的标签指向了这个镜像，如果是这种情况，那么 Delete 行为就不会发生。所以并非所有的 docker image rm 都会产生删除镜像的行为，有可能仅仅是取消了某个标签而已。

当该镜像所有的标签都被取消了，该镜像很可能会失去了存在的意义，因此会触发删除行为。镜像是多层存储结构，因此在删除的时候也是从上层向基础层方向依次进行判断删除。镜像的多层结构让镜像复用变动非常容易，因此很有可能某个其它镜像正依赖于当前镜像的某一层。这种情况，依旧不会触发删除该层的行为。直到没有任何层依赖当前层时，才会真实的删除当前层。这就是为什么，有时候会奇怪，为什么明明没有别的标签指向这个镜像，但是它还是存在的原因，也是为什么有时候会发现所删除的层数和自己 docker pull 看到的层数不一样的源。

除了镜像依赖以外，还需要注意的是容器对镜像的依赖。如果有用这个镜像启动的容器存在（即使容器没有运行），那么同样不可以删除这个镜像。之前讲过，容器是以镜像为基础，再加一层容器存储层，组成这样的多层存储结构去运行的。因此该镜像如果被这个容器所依赖的，那么删除必然会导致故障。如果这些容器是不需要的，应该先将它们删除，然后再来删除镜像。

### 用 docker image ls 命令来配合

像其它可以承接多个实体的命令一样，可以使用 docker image ls -q 来配合使用 docker image rm，这样可以成批的删除希望删除的镜像。我们在“镜像列表”章节介绍过很多过滤镜像列表的方式都可以拿过来使用。

比如，我们需要删除所有仓库名为 redis 的镜像：

docker image rm $(docker image ls -q redis)

或者删除所有在 mongo:3.2 之前的镜像：

docker image rm $(docker image ls -q -f before=mongo:3.2)

充分利用你的想象力和 Linux 命令行的强大，你可以完成很多非常赞的功能。

### CentOS/RHEL 的用户需要注意的事项

在 Ubuntu/Debian 上有 UnionFS 可以使用，如 aufs 或者 overlay2，而 CentOS 和 RHEL 的内核中没有相关驱动。因此对于这类系统，一般使用 devicemapper 驱动利用 LVM 的一些机制来模拟分层存储。这样的做法除了性能比较差外，稳定性一般也不好，而且配置相对复杂。Docker 安装在 CentOS/RHEL 上后，会默认选择 devicemapper，但是为了简化配置，其 devicemapper 是跑在一个稀疏文件模拟的块设备上，也被称为 loop-lvm。这样的选择是因为不需要额外配置就可以运行 Docker，这是自动配置唯一能做到的事情。但是 loop-lvm 的做法非常不好，其稳定性、性能更差，无论是日志还是 docker info 中都会看到警告信息。官方文档有明确的文章讲解了如何配置块设备给 devicemapper 驱动做存储层的做法，这类做法也被称为配置 direct-lvm。

除了前面说到的问题外，devicemapper + loop-lvm 还有一个缺陷，因为它是稀疏文件，所以它会不断增长。用户在使用过程中会注意到 /var/lib/docker/devicemapper/devicemapper/data 不断增长，而且无法控制。很多人会希望删除镜像或者可以解决这个问题，结果发现效果并不明显。原因就是这个稀疏文件的空间释放后基本不进行垃圾回收的问题。因此往往会出现即使删除了文件内容，空间却无法回收，随着使用这个稀疏文件一直在不断增长。

所以对于 CentOS/RHEL 的用户来说，在没有办法使用 UnionFS 的情况下，一定要配置 direct-lvm 给 devicemapper，无论是为了性能、稳定性还是空间利用率。

或许有人注意到了 CentOS 7 中存在被 backports 回来的 overlay 驱动，不过 CentOS 里的这个驱动达不到生产环境使用的稳定程度，所以不推荐使用。

## 利用 commit 理解镜像构成

注意： docker commit 命令除了学习之外，还有一些特殊的应用场合，比如被入侵后保存现场等。但是，不要使用 docker commit 定制镜像，定制镜像应该使用 Dockerfile 来完成。如果你想要定制镜像请查看下一小节。

镜像是容器的基础，每次执行 docker run 的时候都会指定哪个镜像作为容器运行的基础。在之前的例子中，我们所使用的都是来自于 Docker Hub 的镜像。直接使用这些镜像是可以满足一定的需求，而当这些镜像无法直接满足需求时，我们就需要定制这些镜像。接下来的几节就将讲解如何定制镜像。

回顾一下之前我们学到的知识，镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，是在以镜像为基础层，在其基础上加一层作为容器运行时的存储层。

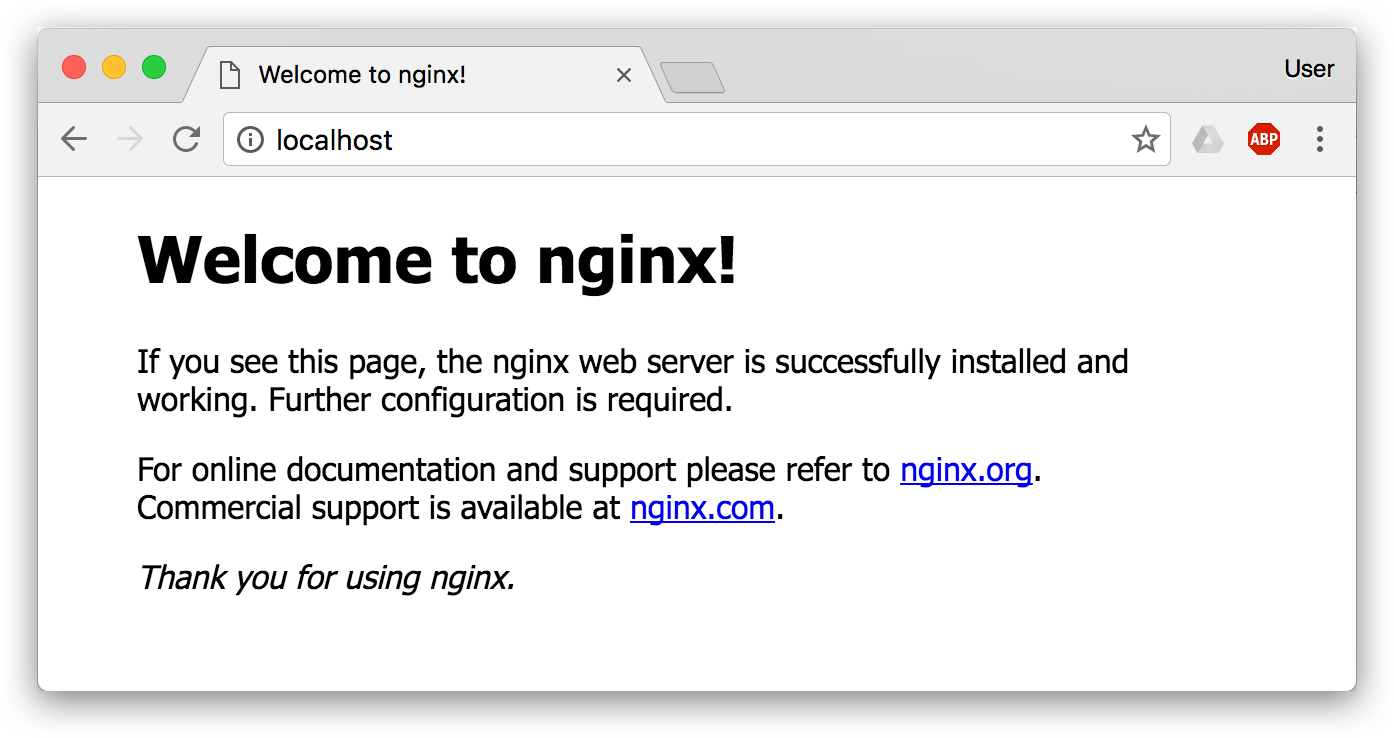
现在让我们以定制一个 Web 服务器为例子，来讲解镜像是如何构建的。

docker run --name webserver -d -p 80:80 nginx

这条命令会用 nginx 镜像启动一个容器，命名为 webserver，并且映射了 80 端口，这样我们可以用浏览器去访问这个 nginx 服务器。

如果是在 Linux 本机运行的 Docker，或者如果使用的是 Docker for Mac、Docker for Windows，那么可以直接访问：http://localhost；如果使用的是 Docker Toolbox，或者是在虚拟机、云服务器上安装的 Docker，则需要将 localhost 换为虚拟机地址或者实际云服务器地址。

直接用浏览器访问的话，我们会看到默认的 Nginx 欢迎页面。



现在，假设我们非常不喜欢这个欢迎页面，我们希望改成欢迎 Docker 的文字，我们可以使用 docker exec 命令进入容器，修改其内容。

docker exec -it webserver bash

root@3729b97e8226:/# echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

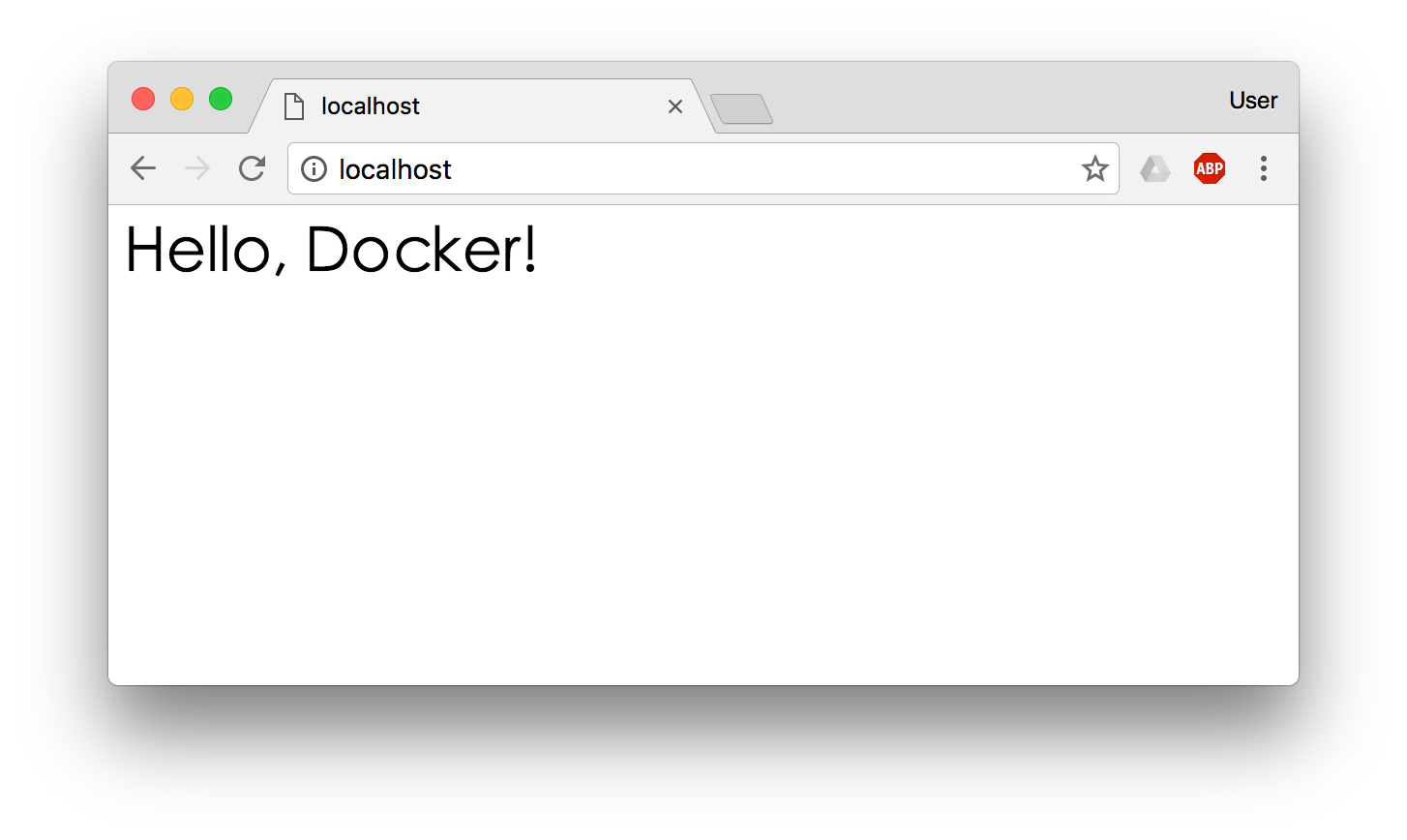
root@3729b97e8226:/# exit

exit

我们以交互式终端方式进入 webserver 容器，并执行了 bash 命令，也就是获得一个可操作的 Shell。

然后，我们用 <h1>Hello, Docker!</h1> 覆盖了 /usr/share/nginx/html/index.html 的内容。

现在我们再刷新浏览器的话，会发现内容被改变了。



我们修改了容器的文件，也就是改动了容器的存储层。我们可以通过 docker diff 命令看到具体的改动。

docker diff webserver

C /root

A /root/.bash\_history

C /run

C /usr

C /usr/share

C /usr/share/nginx

C /usr/share/nginx/html

C /usr/share/nginx/html/index.html

C /var

C /var/cache

C /var/cache/nginx

A /var/cache/nginx/client\_temp

A /var/cache/nginx/fastcgi\_temp

A /var/cache/nginx/proxy\_temp

A /var/cache/nginx/scgi\_temp

A /var/cache/nginx/uwsgi\_temp

现在我们定制好了变化，我们希望能将其保存下来形成镜像。

要知道，当我们运行一个容器的时候（如果不使用卷的话），我们做的任何文件修改都会被记录于容器存储层里。而 Docker 提供了一个 docker commit 命令，可以将容器的存储层保存下来成为镜像。换句话说，就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候，就会拥有原有容器最后的文件变化。

docker commit 的语法格式为：

docker commit [选项] <容器ID或容器名> [<仓库名>[:<标签>]]

我们可以用下面的命令将容器保存为镜像：

docker commit \

--author "Tao Wang <twang2218@gmail.com>" \

--message "修改了默认网页" \

webserver \

nginx:v2

其中 --author 是指定修改的作者，而 --message 则是记录本次修改的内容。这点和 git 版本控制相似，不过这里这些信息可以省略留空。

我们可以在 docker image ls 中看到这个新定制的镜像：

docker image ls nginx

REPOSITORY TAG IMAGE ID CREATED SIZE

nginx v2 07e334659748 9 seconds ago 181.5 MB

nginx 1.11 05a60462f8ba 12 days ago 181.5 MB

nginx latest e43d811ce2f4 4 weeks ago 181.5 MB

我们还可以用 `docker history` 具体查看镜像内的历史记录，如果比较 `nginx:latest` 的历史记录，我们会发现新增了我们刚刚提交的这一层。

docker history nginx:v2

IMAGE CREATED CREATED BY SIZE COMMENT

07e334659748 54 seconds ago nginx -g daemon off; 95 B 修改了默认网页

e43d811ce2f4 4 weeks ago /bin/sh -c #(nop) CMD ["nginx" "-g" "daemon 0 B

<missing> 4 weeks ago /bin/sh -c #(nop) EXPOSE 443/tcp 80/tcp 0 B

<missing> 4 weeks ago /bin/sh -c ln -sf /dev/stdout /var/log/nginx/ 22 B

<missing> 4 weeks ago /bin/sh -c apt-key adv --keyserver hkp://pgp. 58.46 MB

<missing> 4 weeks ago /bin/sh -c #(nop) ENV NGINX\_VERSION=1.11.5-1 0 B

<missing> 4 weeks ago /bin/sh -c #(nop) MAINTAINER NGINX Docker Ma 0 B

<missing> 4 weeks ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0 B

<missing> 4 weeks ago /bin/sh -c #(nop) ADD file:23aa4f893e3288698c 123 MB

新的镜像定制好后，我们可以来运行这个镜像。

docker run --name web2 -d -p 81:80 nginx:v2

这里我们命名为新的服务为 web2，并且映射到 81 端口。如果是 Docker for Mac/Windows 或 Linux 桌面的话，我们就可以直接访问 http://localhost:81 看到结果，其内容应该和之前修改后的 webserver 一样。

至此，我们第一次完成了定制镜像，使用的是 docker commit 命令，手动操作给旧的镜像添加了新的一层，形成新的镜像，对镜像多层存储应该有了更直观的感觉。

### 慎用 docker commit

使用 docker commit 命令虽然可以比较直观的帮助理解镜像分层存储的概念，但是实际环境中并不会这样使用。

首先，如果仔细观察之前的 docker diff webserver 的结果，你会发现除了真正想要修改的 /usr/share/nginx/html/index.html 文件外，由于命令的执行，还有很多文件被改动或添加了。这还仅仅是最简单的操作，如果是安装软件包、编译构建，那会有大量的无关内容被添加进来，如果不小心清理，将会导致镜像极为臃肿。

此外，使用 docker commit 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被称为黑箱镜像，换句话说，就是除了制作镜像的人知道执行过什么命令、怎么生成的镜像，别人根本无从得知。而且，即使是这个制作镜像的人，过一段时间后也无法记清具体在操作的。虽然 docker diff 或许可以告诉得到一些线索，但是远远不到可以确保生成一致镜像的地步。这种黑箱镜像的维护工作是非常痛苦的。

而且，回顾之前提及的镜像所使用的分层存储的概念，除当前层外，之前的每一层都是不会发生改变的，换句话说，任何修改的结果仅仅是在当前层进行标记、添加、修改，而不会改动上一层。如果使用 docker commit 制作镜像，以及后期修改的话，每一次修改都会让镜像更加臃肿一次，所删除的上一层的东西并不会丢失，会一直如影随形的跟着这个镜像，即使根本无法访问到。这会让镜像更加臃肿。

## 使用 Dockerfile 定制镜像

### Dockerfile 定制镜像

从刚才的 docker commit 的学习中，我们可以了解到，镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 Dockerfile。

Dockerfile 是一个文本文件，其内包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

还以之前定制 nginx 镜像为例，这次我们使用 Dockerfile 来定制。

在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：

mkdir mynginx

cd mynginx

touch Dockerfile

其内容为：

FROM nginx

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，FROM 和 RUN。

### FROM 指定基础镜像

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 nginx 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 FROM 就是指定基础镜像，因此一个 Dockerfile 中 FROM 是必备的指令，并且必须是第一条指令。

在 Docker Store 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如 nginx、redis、mongo、mysql、httpd、php、tomcat 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 node、openjdk、python、ruby、golang 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。

如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 ubuntu、debian、centos、fedora、alpine 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。

FROM scratch

...

如果你以 scratch 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。

不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，比如 swarm、coreos/etcd。对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 FROM scratch 会让镜像体积更加小巧。使用 Go 语言 开发的应用很多会使用这种方式来制作镜像，这也是为什么有人认为 Go 是特别适合容器微服务架构的语言的原因之一。

### RUN 执行命令

RUN 指令是用来执行命令行命令的。由于命令行的强大能力，RUN 指令在定制镜像时是最常用的指令之一。其格式有两种：

shell 格式：RUN <命令>，就像直接在命令行中输入的命令一样。刚才写的 Dockerfile 中的 RUN 指令就是这种格式。

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

exec 格式：RUN ["可执行文件", "参数1", "参数2"]，这更像是函数调用中的格式。

既然 RUN 就像 Shell 脚本一样可以执行命令，那么我们是否就可以像 Shell 脚本一样把每个命令对应一个 RUN 呢？比如这样：

FROM debian:jessie

RUN apt-get update

RUN apt-get install -y gcc libc6-dev make

RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz"

RUN mkdir -p /usr/src/redis

RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1

RUN make -C /usr/src/redis

RUN make -C /usr/src/redis install

之前说过，Dockerfile 中每一个指令都会建立一层，RUN 也不例外。每一个 RUN 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，commit 这一层的修改，构成新的镜像。

而上面的这种写法，创建了 7 层镜像。这是完全没有意义的，而且很多运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常臃肿、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错。 这是很多初学 Docker 的人常犯的一个错误。

Union FS 是有最大层数限制的，比如 AUFS，曾经是最大不得超过 42 层，现在是不得超过 127 层。

上面的 Dockerfile 正确的写法应该是这样：

FROM debian:jessie

RUN buildDeps='gcc libc6-dev make' \

&& apt-get update \

&& apt-get install -y $buildDeps \

&& wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz" \

&& mkdir -p /usr/src/redis \

&& tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \

&& make -C /usr/src/redis \

&& make -C /usr/src/redis install \

&& rm -rf /var/lib/apt/lists/\* \

&& rm redis.tar.gz \

&& rm -r /usr/src/redis \

&& apt-get purge -y --auto-remove $buildDeps

首先，之前所有的命令只有一个目的，就是编译、安装 redis 可执行文件。因此没有必要建立很多层，这只是一层的事情。因此，这里没有使用很多个 RUN 对一一对应不同的命令，而是仅仅使用一个 RUN 指令，并使用 && 将各个所需命令串联起来。将之前的 7 层，简化为了 1 层。在撰写 Dockerfile 的时候，要经常提醒自己，这并不是在写 Shell 脚本，而是在定义每一层该如何构建。

并且，这里为了格式化还进行了换行。Dockerfile 支持 Shell 类的行尾添加 \ 的命令换行方式，以及行首 # 进行注释的格式。良好的格式，比如换行、缩进、注释等，会让维护、排障更为容易，这是一个比较好的习惯。

此外，还可以看到这一组命令的最后添加了清理工作的命令，删除了为了编译构建所需要的软件，清理了所有下载、展开的文件，并且还清理了 apt 缓存文件。这是很重要的一步，我们之前说过，镜像是多层存储，每一层的东西并不会在下一层被删除，会一直跟随着镜像。因此镜像构建时，一定要确保每一层只添加真正需要添加的东西，任何无关的东西都应该清理掉。

很多人初学 Docker 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

### 构建镜像

好了，让我们再回到之前定制的 nginx 镜像的 Dockerfile 来。现在我们明白了这个 Dockerfile 的内容，那么让我们来构建这个镜像吧。

在 Dockerfile 文件所在目录执行：

docker build -t nginx:v3 .

从命令的输出结果中，我们可以清晰的看到镜像的构建过程。在 Step 2 中，如同我们之前所说的那样，RUN 指令启动了一个容器 9cdc27646c7b，执行了所要求的命令，并最后提交了这一层 44aa4490ce2c，随后删除了所用到的这个容器 9cdc27646c7b。

这里我们使用了 docker build 命令进行镜像构建。其格式为：

docker build [选项] <上下文路径/URL/->

在这里我们指定了最终镜像的名称 -t nginx:v3，构建成功后，我们可以像之前运行 nginx:v2 那样来运行这个镜像，其结果会和 nginx:v2 一样。

### 镜像构建上下文（Context）

如果注意，会看到 docker build 命令最后有一个 .。. 表示当前目录，而 Dockerfile 就在当前目录，因此不少初学者以为这个路径是在指定 Dockerfile 所在路径，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定上下文路径。那么什么是上下文呢？

首先我们要理解 docker build 的工作原理。Docker 在运行时分为 Docker 引擎（也就是服务端守护进程）和客户端工具。Docker 的引擎提供了一组 REST API，被称为 Docker Remote API，而如 docker 命令这样的客户端工具，则是通过这组 API 与 Docker 引擎交互，从而完成各种功能。因此，虽然表面上我们好像是在本机执行各种 docker 功能，但实际上，一切都是使用的远程调用形式在服务端（Docker 引擎）完成。也因为这种 C/S 设计，让我们操作远程服务器的 Docker 引擎变得轻而易举。

当我们进行镜像构建的时候，并非所有定制都会通过 RUN 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 COPY 指令、ADD 指令等。而 docker build 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？

这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。

如果在 Dockerfile 中这么写：

COPY ./package.json /app/

这并不是要复制执行 docker build 命令所在的目录下的 package.json，也不是复制 Dockerfile 所在目录下的 package.json，而是复制 上下文（context） 目录下的 package.json。

因此，COPY 这类指令中的源文件的路径都是相对路径。这也是初学者经常会问的为什么 COPY ../package.json /app 或者 COPY /opt/xxxx /app 无法工作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位置的文件。如果真的需要那些文件，应该将它们复制到上下文目录中去。

现在就可以理解刚才的命令 docker build -t nginx:v3 . 中的这个 .，实际上是在指定上下文的目录，docker build 命令会将该目录下的内容打包交给 Docker 引擎以帮助构建镜像。

如果观察 docker build 输出，我们其实已经看到了这个发送上下文的过程：

docker build -t nginx:v3 .

理解构建上下文对于镜像构建是很重要的，避免犯一些不应该的错误。比如有些初学者在发现 COPY /opt/xxxx /app 不工作后，于是干脆将 Dockerfile 放到了硬盘根目录去构建，结果发现 docker build 执行后，在发送一个几十 GB 的东西，极为缓慢而且很容易构建失败。那是因为这种做法是在让 docker build 打包整个硬盘，这显然是使用错误。

一般来说，应该会将 Dockerfile 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 .gitignore 一样的语法写一个 .dockerignore，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

那么为什么会有人误以为 . 是指定 Dockerfile 所在目录呢？这是因为在默认情况下，如果不额外指定 Dockerfile 的话，会将上下文目录下的名为 Dockerfile 的文件作为 Dockerfile。

这只是默认行为，实际上 Dockerfile 的文件名并不要求必须为 Dockerfile，而且并不要求必须位于上下文目录中，比如可以用 -f ../Dockerfile.php 参数指定某个文件作为 Dockerfile。

当然，一般大家习惯性的会使用默认的文件名 Dockerfile，以及会将其置于镜像构建上下文目录中。

### 其它 docker build 的用法

#### 直接用 Git repo 进行构建

或许你已经注意到了，docker build 还支持从 URL 构建，比如可以直接从 Git repo 中构建：

docker build https://github.com/twang2218/gitlab-ce-zh.git#:8.14

这行命令指定了构建所需的 Git repo，并且指定默认的 master 分支，构建目录为 /8.14/，然后 Docker 就会自己去 git clone 这个项目、切换到指定分支、并进入到指定目录后开始构建。

#### 用给定的 tar 压缩包构建

docker build http://server/context.tar.gz

如果所给出的 URL 不是个 Git repo，而是个 tar 压缩包，那么 Docker 引擎会下载这个包，并自动解压缩，以其作为上下文，开始构建。

#### 从标准输入中读取 Dockerfile 进行构建

docker build - < Dockerfile

或

cat Dockerfile | docker build -

如果标准输入传入的是文本文件，则将其视为 Dockerfile，并开始构建。这种形式由于直接从标准输入中读取 Dockerfile 的内容，它没有上下文，因此不可以像其他方法那样可以将本地文件 COPY 进镜像之类的事情。

#### 从标准输入中读取上下文压缩包进行构建

docker build - < context.tar.gz

如果发现标准输入的文件格式是 gzip、bzip2 以及 xz 的话，将会使其为上下文压缩包，直接将其展开，将里面视为上下文，并开始构建。

## Dockerfile 指令详解

我们已经介绍了 FROM，RUN，还提及了 COPY, ADD，其实 Dockerfile 功能很强大，它提供了十多个指令。下面我们继续讲解其他的指令。

### COPY 复制文件

格式：

* COPY <源路径>... <目标路径>
* COPY ["<源路径1>",... "<目标路径>"]

和 RUN 指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。

COPY 指令将从构建上下文目录中 <源路径> 的文件/目录复制到新的一层的镜像内的 <目标路径> 位置。比如：

COPY package.json /usr/src/app/

<源路径> 可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的 filepath.Match 规则，如：

COPY hom\* /mydir/

COPY hom?.txt /mydir/

<目标路径> 可以是容器内的绝对路径，也可以是相对于工作目录的相对路径（工作目录可以用 WORKDIR 指令来指定）。目标路径不需要事先创建，如果目录不存在会在复制文件前先行创建缺失目录。

此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限、文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git 进行管理的时候。

### ADD 更高级的复制文件

ADD 指令和 COPY 的格式和性质基本一致。但是在 COPY 基础上增加了一些功能。

比如 <源路径> 可以是一个 URL，这种情况下，Docker 引擎会试图去下载这个链接的文件放到 <目标路径> 去。下载后的文件权限自动设置为 600，如果这并不是想要的权限，那么还需要增加额外的一层 RUN 进行权限调整，另外，如果下载的是个压缩包，需要解压缩，也一样还需要额外的一层 RUN 指令进行解压缩。所以不如直接使用 RUN 指令，然后使用 wget 或者 curl 工具下载，处理权限、解压缩、然后清理无用文件更合理。因此，这个功能其实并不实用，而且不推荐使用。

如果 <源路径> 为一个 tar 压缩文件的话，压缩格式为 gzip, bzip2 以及 xz 的情况下，ADD 指令将会自动解压缩这个压缩文件到 <目标路径> 去。

在某些情况下，这个自动解压缩的功能非常有用，比如官方镜像 ubuntu 中：

FROM scratch

ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /

...

但在某些情况下，如果我们真的是希望复制个压缩文件进去，而不解压缩，这时就不可以使用 ADD 命令了。

在 Docker 官方的 Dockerfile 最佳实践文档 中要求，尽可能的使用 COPY，因为 COPY 的语义很明确，就是复制文件而已，而 ADD 则包含了更复杂的功能，其行为也不一定很清晰。最适合使用 ADD 的场合，就是所提及的需要自动解压缩的场合。

另外需要注意的是，ADD 指令会令镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢。

因此在 COPY 和 ADD 指令中选择的时候，可以遵循这样的原则，所有的文件复制均使用 COPY 指令，仅在需要自动解压缩的场合使用 ADD。

### CMD 容器启动命令

CMD 指令的格式和 RUN 相似，也是两种格式：

* shell 格式：CMD <命令>
* exec 格式：CMD ["可执行文件", "参数1", "参数2"...]
* 参数列表格式：CMD ["参数1", "参数2"...]。在指定了 ENTRYPOINT 指令后，用 CMD 指定具体的参数。

之前介绍容器的时候曾经说过，Docker 不是虚拟机，容器就是进程。既然是进程，那么在启动容器的时候，需要指定所运行的程序及参数。CMD 指令就是用于指定默认的容器主进程的启动命令的。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令，比如，ubuntu 镜像默认的 CMD 是 /bin/bash，如果我们直接 docker run -it ubuntu 的话，会直接进入 bash。我们也可以在运行时指定运行别的命令，如 docker run -it ubuntu cat /etc/os-release。这就是用 cat /etc/os-release 命令替换了默认的 /bin/bash 命令了，输出了系统版本信息。

在指令格式上，一般推荐使用 exec 格式，这类格式在解析时会被解析为 JSON 数组，因此一定要使用双引号 "，而不要使用单引号。

如果使用 shell 格式的话，实际的命令会被包装为 sh -c 的参数的形式进行执行。比如：

CMD echo $HOME

在实际执行中，会将其变更为：

CMD [ "sh", "-c", "echo $HOME" ]

这就是为什么我们可以使用环境变量的原因，因为这些环境变量会被 shell 进行解析处理。

提到 CMD 就不得不提容器中应用在前台执行和后台执行的问题。这是初学者常出现的一个混淆。

Docker 不是虚拟机，容器中的应用都应该以前台执行，而不是像虚拟机、物理机里面那样，用 upstart/systemd 去启动后台服务，容器内没有后台服务的概念。

一些初学者将 CMD 写为：

CMD service nginx start

然后发现容器执行后就立即退出了。甚至在容器内去使用 systemctl 命令结果却发现根本执行不了。这就是因为没有搞明白前台、后台的概念，没有区分容器和虚拟机的差异，依旧在以传统虚拟机的角度去理解容器。

对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其它辅助进程不是它需要关心的东西。

而使用 service nginx start 命令，则是希望 upstart 来以后台守护进程形式启动 nginx 服务。而刚才说了 CMD service nginx start 会被理解为 CMD [ "sh", "-c", "service nginx start"]，因此主进程实际上是 sh。那么当 service nginx start 命令结束后，sh 也就结束了，sh 作为主进程退出了，自然就会令容器退出。

正确的做法是直接执行 nginx 可执行文件，并且要求以前台形式运行。比如：

CMD ["nginx", "-g", "daemon off;"]

### ENTRYPOINT 入口点

ENTRYPOINT 的格式和 RUN 指令格式一样，分为 exec 格式和 shell 格式。

ENTRYPOINT 的目的和 CMD 一样，都是在指定容器启动程序及参数。ENTRYPOINT 在运行时也可以替代，不过比 CMD 要略显繁琐，需要通过 docker run 的参数 --entrypoint 来指定。

当指定了 ENTRYPOINT 后，CMD 的含义就发生了改变，不再是直接的运行其命令，而是将 CMD 的内容作为参数传给 ENTRYPOINT 指令，换句话说实际执行时，将变为：

<ENTRYPOINT> "<CMD>"

那么有了 CMD 后，为什么还要有 ENTRYPOINT 呢？这种 <ENTRYPOINT> "<CMD>" 有什么好处么？让我们来看几个场景。

**场景一：让镜像变成像命令一样使用**

假设我们需要一个得知自己当前公网 IP 的镜像，那么可以先用 CMD 来实现：

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

CMD [ "curl", "-s", "http://ip.cn" ]

假如我们使用 docker build -t myip . 来构建镜像的话，如果我们需要查询当前公网 IP，只需要执行：

docker run myip

当前 IP：61.148.226.66 来自：北京市 联通

嗯，这么看起来好像可以直接把镜像当做命令使用了，不过命令总有参数，如果我们希望加参数呢？比如从上面的 CMD 中可以看到实质的命令是 curl，那么如果我们希望显示 HTTP 头信息，就需要加上 -i 参数。那么我们可以直接加 -i 参数给 docker run myip 么？

docker run myip -i

docker: Error response from daemon: invalid header field value "oci runtime error: container\_linux.go:247: starting container process caused \"exec: \\\"-i\\\": executable file not found in $PATH\"\n".

我们可以看到可执行文件找不到的报错，executable file not found。之前我们说过，跟在镜像名后面的是 command，运行时会替换 CMD 的默认值。因此这里的 -i 替换了原来的 CMD，而不是添加在原来的 curl -s http://ip.cn 后面。而 -i 根本不是命令，所以自然找不到。

那么如果我们希望加入 -i 这参数，我们就必须重新完整的输入这个命令：

docker run myip curl -s http://ip.cn -i

这显然不是很好的解决方案，而使用 ENTRYPOINT 就可以解决这个问题。现在我们重新用 ENTRYPOINT 来实现这个镜像：

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

ENTRYPOINT [ "curl", "-s", "http://ip.cn" ]

这次我们再来尝试直接使用 docker run myip -i：

docker run myip

当前 IP：61.148.226.66 来自：北京市 联通

docker run myip -i

HTTP/1.1 200 OK

Server: nginx/1.8.0

Date: Tue, 22 Nov 2016 05:12:40 GMT

Content-Type: text/html; charset=UTF-8

Vary: Accept-Encoding

X-Powered-By: PHP/5.6.24-1~dotdeb+7.1

X-Cache: MISS from cache-2

X-Cache-Lookup: MISS from cache-2:80

X-Cache: MISS from proxy-2\_6

Transfer-Encoding: chunked

Via: 1.1 cache-2:80, 1.1 proxy-2\_6:8006

Connection: keep-alive

当前 IP：61.148.226.66 来自：北京市 联通

可以看到，这次成功了。这是因为当存在 ENTRYPOINT 后，CMD 的内容将会作为参数传给 ENTRYPOINT，而这里 -i 就是新的 CMD，因此会作为参数传给 curl，从而达到了我们预期的效果。

**场景二：应用运行前的准备工作**

启动容器就是启动主进程，但有些时候，启动主进程前，需要一些准备工作。

比如 mysql 类的数据库，可能需要一些数据库配置、初始化的工作，这些工作要在最终的 mysql 服务器运行之前解决。

此外，可能希望避免使用 root 用户去启动服务，从而提高安全性，而在启动服务前还需要以 root 身份执行一些必要的准备工作，最后切换到服务用户身份启动服务。或者除了服务外，其它命令依旧可以使用 root 身份执行，方便调试等。

这些准备工作是和容器 CMD 无关的，无论 CMD 为什么，都需要事先进行一个预处理的工作。这种情况下，可以写一个脚本，然后放入 ENTRYPOINT 中去执行，而这个脚本会将接到的参数（也就是 <CMD>）作为命令，在脚本最后执行。比如官方镜像 redis 中就是这么做的：

FROM alpine:3.4

...

RUN addgroup -S redis && adduser -S -G redis redis

...

ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 6379

CMD [ "redis-server" ]

可以看到其中为了 redis 服务创建了 redis 用户，并在最后指定了 ENTRYPOINT 为 docker-entrypoint.sh 脚本。

#!/bin/sh

...

# allow the container to be started with `--user`

if [ "$1" = 'redis-server' -a "$(id -u)" = '0' ]; then

chown -R redis .

exec su-exec redis "$0" "$@"

fi

exec "$@"

该脚本的内容就是根据 CMD 的内容来判断，如果是 redis-server 的话，则切换到 redis 用户身份启动服务器，否则依旧使用 root 身份执行。比如：

docker run -it redis id

uid=0(root) gid=0(root) groups=0(root)

### ENV 设置环境变量

格式有两种：

* ENV <key> <value>
* ENV <key1>=<value1> <key2>=<value2>...

这个指令很简单，就是设置环境变量而已，无论是后面的其它指令，如 RUN，还是运行时的应用，都可以直接使用这里定义的环境变量。

ENV VERSION=1.0 DEBUG=on NAME="Happy Feet"

这个例子中演示了如何换行，以及对含有空格的值用双引号括起来的办法，这和 Shell 下的行为是一致的。

定义了环境变量，那么在后续的指令中，就可以使用这个环境变量。比如在官方 node 镜像 Dockerfile 中，就有类似这样的代码：

ENV NODE\_VERSION 7.2.0

RUN curl -SLO "https://nodejs.org/dist/v$NODE\_VERSION/node-v$NODE\_VERSION-linux-x64.tar.xz" \

&& curl -SLO "https://nodejs.org/dist/v$NODE\_VERSION/SHASUMS256.txt.asc" \

&& gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \

&& grep " node-v$NODE\_VERSION-linux-x64.tar.xz\$" SHASUMS256.txt | sha256sum -c - \

&& tar -xJf "node-v$NODE\_VERSION-linux-x64.tar.xz" -C /usr/local --strip-components=1 \

&& rm "node-v$NODE\_VERSION-linux-x64.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt \

&& ln -s /usr/local/bin/node /usr/local/bin/nodejs

在这里先定义了环境变量 NODE\_VERSION，其后的 RUN 这层里，多次使用 $NODE\_VERSION 来进行操作定制。可以看到，将来升级镜像构建版本的时候，只需要更新 7.2.0 即可，Dockerfile 构建维护变得更轻松了。

下列指令可以支持环境变量展开： ADD、COPY、ENV、EXPOSE、LABEL、USER、WORKDIR、VOLUME、STOPSIGNAL、ONBUILD。

可以从这个指令列表里感觉到，环境变量可以使用的地方很多，很强大。通过环境变量，我们可以让一份 Dockerfile 制作更多的镜像，只需使用不同的环境变量即可。

### ARG 构建参数

格式：ARG <参数名>[=<默认值>]

构建参数和 ENV 的效果一样，都是设置环境变量。所不同的是，ARG 所设置的构建环境的环境变量，在将来容器运行时是不会存在这些环境变量的。但是不要因此就使用 ARG 保存密码之类的信息，因为 docker history 还是可以看到所有值的。

Dockerfile 中的 ARG 指令是定义参数名称，以及定义其默认值。该默认值可以在构建命令 docker build 中用 --build-arg <参数名>=<值> 来覆盖。

在 1.13 之前的版本，要求 --build-arg 中的参数名，必须在 Dockerfile 中用 ARG 定义过了，换句话说，就是 --build-arg 指定的参数，必须在 Dockerfile 中使用了。如果对应参数没有被使用，则会报错退出构建。从 1.13 开始，这种严格的限制被放开，不再报错退出，而是显示警告信息，并继续构建。这对于使用 CI 系统，用同样的构建流程构建不同的 Dockerfile 的时候比较有帮助，避免构建命令必须根据每个 Dockerfile 的内容修改。

### VOLUME 定义匿名卷

格式为：

* VOLUME ["<路径1>", "<路径2>"...]
* VOLUME <路径>

之前我们说过，容器运行时应该尽量保持容器存储层不发生写操作，对于数据库类需要保存动态数据的应用，其数据库文件应该保存于卷(volume)中，后面的章节我们会进一步介绍 Docker 卷的概念。为了防止运行时用户忘记将动态文件所保存目录挂载为卷，在 Dockerfile 中，我们可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

VOLUME /data

这里的 /data 目录就会在运行时自动挂载为匿名卷，任何向 /data 中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。当然，运行时可以覆盖这个挂载设置。比如：

docker run -d -v mydata:/data xxxx

在这行命令中，就使用了 mydata 这个命名卷挂载到了 /data 这个位置，替代了 Dockerfile 中定义的匿名卷的挂载配置。

### EXPOSE 暴露端口

格式为 EXPOSE <端口1> [<端口2>...]。

EXPOSE 指令是声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是 docker run -P 时，会自动随机映射 EXPOSE 的端口。

此外，在早期 Docker 版本中还有一个特殊的用处。以前所有容器都运行于默认桥接网络中，因此所有容器互相之间都可以直接访问，这样存在一定的安全性问题。于是有了一个 Docker 引擎参数 --icc=false，当指定该参数后，容器间将默认无法互访，除非互相间使用了 --links 参数的容器才可以互通，并且只有镜像中 EXPOSE 所声明的端口才可以被访问。这个 --icc=false 的用法，在引入了 docker network 后已经基本不用了，通过自定义网络可以很轻松的实现容器间的互联与隔离。

要将 EXPOSE 和在运行时使用 -p <宿主端口>:<容器端口> 区分开来。-p，是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而 EXPOSE 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

### WORKDIR 指定工作目录

格式为 WORKDIR <工作目录路径>。

使用 WORKDIR 指令可以来指定工作目录（或者称为当前目录），以后各层的当前目录就被改为指定的目录，如该目录不存在，WORKDIR 会帮你建立目录。

之前提到一些初学者常犯的错误是把 Dockerfile 等同于 Shell 脚本来书写，这种错误的理解还可能会导致出现下面这样的错误：

RUN cd /app

RUN echo "hello" > world.txt

如果将这个 Dockerfile 进行构建镜像运行后，会发现找不到 /app/world.txt 文件，或者其内容不是 hello。原因其实很简单，在 Shell 中，连续两行是同一个进程执行环境，因此前一个命令修改的内存状态，会直接影响后一个命令；而在 Dockerfile 中，这两行 RUN 命令的执行环境根本不同，是两个完全不同的容器。这就是对 Dockerfile 构建分层存储的概念不了解所导致的错误。

之前说过每一个 RUN 都是启动一个容器、执行命令、然后提交存储层文件变更。第一层 RUN cd /app 的执行仅仅是当前进程的工作目录变更，一个内存上的变化而已，其结果不会造成任何文件变更。而到第二层的时候，启动的是一个全新的容器，跟第一层的容器更完全没关系，自然不可能继承前一层构建过程中的内存变化。

因此如果需要改变以后各层的工作目录的位置，那么应该使用 WORKDIR 指令。

### USER 指定当前用户

格式：USER <用户名>

USER 指令和 WORKDIR 相似，都是改变环境状态并影响以后的层。WORKDIR 是改变工作目录，USER 则是改变之后层的执行 RUN, CMD 以及 ENTRYPOINT 这类命令的身份。

当然，和 WORKDIR 一样，USER 只是帮助你切换到指定用户而已，这个用户必须是事先建立好的，否则无法切换。

RUN groupadd -r redis && useradd -r -g redis redis

USER redis

RUN [ "redis-server" ]

如果以 root 执行的脚本，在执行期间希望改变身份，比如希望以某个已经建立好的用户来运行某个服务进程，不要使用 su 或者 sudo，这些都需要比较麻烦的配置，而且在 TTY 缺失的环境下经常出错。建议使用 gosu。

# 建立 redis 用户，并使用 gosu 换另一个用户执行命令

RUN groupadd -r redis && useradd -r -g redis redis

# 下载 gosu

RUN wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/1.7/gosu-amd64" \

&& chmod +x /usr/local/bin/gosu \

&& gosu nobody true

# 设置 CMD，并以另外的用户执行

CMD [ "exec", "gosu", "redis", "redis-server" ]

### HEALTHCHECK 健康检查

格式：

* HEALTHCHECK [选项] CMD <命令>：设置检查容器健康状况的命令
* HEALTHCHECK NONE：如果基础镜像有健康检查指令，使用这行可以屏蔽掉其健康检查指令

HEALTHCHECK 指令是告诉 Docker 应该如何进行判断容器的状态是否正常，这是 Docker 1.12 引入的新指令。

在没有 HEALTHCHECK 指令前，Docker 引擎只可以通过容器内主进程是否退出来判断容器是否状态异常。很多情况下这没问题，但是如果程序进入死锁状态，或者死循环状态，应用进程并不退出，但是该容器已经无法提供服务了。在 1.12 以前，Docker 不会检测到容器的这种状态，从而不会重新调度，导致可能会有部分容器已经无法提供服务了却还在接受用户请求。

而自 1.12 之后，Docker 提供了 HEALTHCHECK 指令，通过该指令指定一行命令，用这行命令来判断容器主进程的服务状态是否还正常，从而比较真实的反应容器实际状态。

当在一个镜像指定了 HEALTHCHECK 指令后，用其启动容器，初始状态会为 starting，在 HEALTHCHECK 指令检查成功后变为 healthy，如果连续一定次数失败，则会变为 unhealthy。

HEALTHCHECK 支持下列选项：

* --interval=<间隔>：两次健康检查的间隔，默认为 30 秒；
* --timeout=<时长>：健康检查命令运行超时时间，如果超过这个时间，本次健康检查就被视为失败，默认 30 秒；
* --retries=<次数>：当连续失败指定次数后，则将容器状态视为 unhealthy，默认 3 次。

和 CMD, ENTRYPOINT 一样，HEALTHCHECK 只可以出现一次，如果写了多个，只有最后一个生效。

在 HEALTHCHECK [选项] CMD 后面的命令，格式和 ENTRYPOINT 一样，分为 shell 格式，和 exec 格式。命令的返回值决定了该次健康检查的成功与否：0：成功；1：失败；2：保留，不要使用这个值。

假设我们有个镜像是个最简单的 Web 服务，我们希望增加健康检查来判断其 Web 服务是否在正常工作，我们可以用 curl 来帮助判断，其 Dockerfile 的 HEALTHCHECK 可以这么写：

FROM nginx

RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/\*

HEALTHCHECK --interval=5s --timeout=3s \

CMD curl -fs http://localhost/ || exit 1

这里我们设置了每 5 秒检查一次（这里为了试验所以间隔非常短，实际应该相对较长），如果健康检查命令超过 3 秒没响应就视为失败，并且使用 curl -fs http://localhost/ || exit 1 作为健康检查命令。

使用 docker build 来构建这个镜像：

docker build -t myweb:v1 .

构建好了后，我们启动一个容器：

docker run -d --name web -p 80:80 myweb:v1

当运行该镜像后，可以通过 docker container ls 看到最初的状态为 (health: starting)：

docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

03e28eb00bd0 myweb:v1 "nginx -g 'daemon off" 3 seconds ago Up 2 seconds (health: starting) 80/tcp, 443/tcp web

在等待几秒钟后，再次 docker container ls，就会看到健康状态变化为了 (healthy)：

docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

03e28eb00bd0 myweb:v1 "nginx -g 'daemon off" 18 seconds ago Up 16 seconds (healthy) 80/tcp, 443/tcp web

如果健康检查连续失败超过了重试次数，状态就会变为 (unhealthy)。

为了帮助排障，健康检查命令的输出（包括 stdout 以及 stderr）都会被存储于健康状态里，可以用 docker inspect 来查看。

docker inspect --format '{{json .State.Health}}' web | python -m json.tool

{

"FailingStreak": 0,

"Log": [

{

"End": "2016-11-25T14:35:37.940957051Z",

"ExitCode": 0,

"Output": "<!DOCTYPE html>\n<html>…… </html>\n",

"Start": "2019-11-25T14:35:37.780192565Z"

}

],

"Status": "healthy"

}

### ONBUILD 为他人作嫁衣

格式：ONBUILD <其它指令>。

ONBUILD 是一个特殊的指令，它后面跟的是其它指令，比如 RUN, COPY 等，而这些指令，在当前镜像构建时并不会被执行。只有当以当前镜像为基础镜像，去构建下一级镜像的时候才会被执行。

Dockerfile 中的其它指令都是为了定制当前镜像而准备的，唯有 ONBUILD 是为了帮助别人定制自己而准备的。

假设我们要制作 Node.js 所写的应用的镜像。我们都知道 Node.js 使用 npm 进行包管理，所有依赖、配置、启动信息等会放到 package.json 文件里。在拿到程序代码后，需要先进行 npm install 才可以获得所有需要的依赖。然后就可以通过 npm start 来启动应用。因此，一般来说会这样写 Dockerfile：

FROM node:slim

RUN mkdir /app

WORKDIR /app

COPY ./package.json /app

RUN [ "npm", "install" ]

COPY . /app/

CMD [ "npm", "start" ]

把这个 Dockerfile 放到 Node.js 项目的根目录，构建好镜像后，就可以直接拿来启动容器运行。但是如果我们还有第二个 Node.js 项目也差不多呢？好吧，那就再把这个 Dockerfile 复制到第二个项目里。那如果有第三个项目呢？再复制么？文件的副本越多，版本控制就越困难，让我们继续看这样的场景维护的问题。

如果第一个 Node.js 项目在开发过程中，发现这个 Dockerfile 里存在问题，比如敲错字了、或者需要安装额外的包，然后开发人员修复了这个 Dockerfile，再次构建，问题解决。第一个项目没问题了，但是第二个项目呢？虽然最初 Dockerfile 是复制、粘贴自第一个项目的，但是并不会因为第一个项目修复了他们的 Dockerfile，而第二个项目的 Dockerfile 就会被自动修复。

那么我们可不可以做一个基础镜像，然后各个项目使用这个基础镜像呢？这样基础镜像更新，各个项目不用同步 Dockerfile 的变化，重新构建后就继承了基础镜像的更新？好吧，可以，让我们看看这样的结果。那么上面的这个 Dockerfile 就会变为：

FROM node:slim

RUN mkdir /app

WORKDIR /app

CMD [ "npm", "start" ]

这里我们把项目相关的构建指令拿出来，放到子项目里去。假设这个基础镜像的名字为 my-node 的话，各个项目内的自己的 Dockerfile 就变为：

FROM my-node

COPY ./package.json /app

RUN [ "npm", "install" ]

COPY . /app/

基础镜像变化后，各个项目都用这个 Dockerfile 重新构建镜像，会继承基础镜像的更新。

那么，问题解决了么？没有。准确说，只解决了一半。如果这个 Dockerfile 里面有些东西需要调整呢？比如 npm install 都需要加一些参数，那怎么办？这一行 RUN 是不可能放入基础镜像的，因为涉及到了当前项目的 ./package.json，难道又要一个个修改么？所以说，这样制作基础镜像，只解决了原来的 Dockerfile 的前4条指令的变化问题，而后面三条指令的变化则完全没办法处理。

ONBUILD 可以解决这个问题。让我们用 ONBUILD 重新写一下基础镜像的 Dockerfile:

FROM node:slim

RUN mkdir /app

WORKDIR /app

ONBUILD COPY ./package.json /app

ONBUILD RUN [ "npm", "install" ]

ONBUILD COPY . /app/

CMD [ "npm", "start" ]

这次我们回到原始的 Dockerfile，但是这次将项目相关的指令加上 ONBUILD，这样在构建基础镜像的时候，这三行并不会被执行。然后各个项目的 Dockerfile 就变成了简单地：

FROM my-node

是的，只有这么一行。当在各个项目目录中，用这个只有一行的 Dockerfile 构建镜像时，之前基础镜像的那三行 ONBUILD 就会开始执行，成功的将当前项目的代码复制进镜像、并且针对本项目执行 npm install，生成应用镜像。

### 参考文档

[Dockerfie 官方文档](https://docs.docker.com/engine/reference/builder/)

[Dockerfile 最佳实践文档](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/)

[Docker 官方镜像 Dockerfile](https://github.com/docker-library/docs)

## Dockerfile 多阶段构建

### 之前的做法

在 Docker 17.05 版本之前，我们构建 Docker 镜像时，通常会采用两种方式：

#### 全部放入一个 Dockerfile

一种方式是将所有的构建过程编包含在一个 Dockerfile 中，包括项目及其依赖库的编译、测试、打包等流程，这里可能会带来的一些问题：

Dockerfile 特别长，可维护性降低

镜像层次多，镜像体积较大，部署时间变长

源代码存在泄露的风险

例如

编写 app.go 文件，该程序输出 Hello World!

package main

import "fmt"

func main(){

fmt.Printf("Hello World!");

}

编写 Dockerfile.one 文件

FROM golang:1.9-alpine

RUN apk --no-cache add git ca-certificates

WORKDIR /go/src/github.com/go/helloworld/

COPY app.go .

RUN go get -d -v github.com/go-sql-driver/mysql \

&& CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app . \

&& cp /go/src/github.com/go/helloworld/app /root

WORKDIR /root/

CMD ["./app"]

构建镜像

docker build -t go/helloworld:1 -f Dockerfile.one

#### 分散到多个 Dockerfile

另一种方式，就是我们事先在一个 Dockerfile 将项目及其依赖库编译测试打包好后，再将其拷贝到运行环境中，这种方式需要我们编写两个 Dockerfile 和一些编译脚本才能将其两个阶段自动整合起来，这种方式虽然可以很好地规避第一种方式存在的风险，但明显部署过程较复杂。

例如

编写 Dockerfile.build 文件

FROM golang:1.9-alpine

RUN apk --no-cache add git

WORKDIR /go/src/github.com/go/helloworld

COPY app.go .

RUN go get -d -v github.com/go-sql-driver/mysql \

&& CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

编写 Dockerfile.copy 文件

FROM alpine:latest

RUN apk --no-cache add ca-certificates

WORKDIR /root/

COPY app .

CMD ["./app"]

新建 build.sh

#!/bin/sh

echo Building go/helloworld:build

docker build -t go/helloworld:build . -f Dockerfile.build

docker create --name extract go/helloworld:build

docker cp extract:/go/src/github.com/go/helloworld/app ./app

docker rm -f extract

echo Building go/helloworld:2

docker build --no-cache -t go/helloworld:2 . -f Dockerfile.copy

rm ./app

现在运行脚本即可构建镜像

chmod +x build.sh

./build.sh

对比两种方式生成的镜像大小

docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

go/helloworld 2 f7cf3465432c 22 seconds ago 6.47MB

go/helloworld 1 f55d3e16affc 2 minutes ago 295MB

### 使用多阶段构建

为解决以上问题，Docker v17.05 开始支持多阶段构建 (multistage builds)。使用多阶段构建我们就可以很容易解决前面提到的问题，并且只需要编写一个 Dockerfile：

例如

编写 Dockerfile 文件

FROM golang:1.9-alpine as builder

RUN apk --no-cache add git

WORKDIR /go/src/github.com/go/helloworld/

RUN go get -d -v github.com/go-sql-driver/mysql

COPY app.go .

RUN CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest as prod

RUN apk --no-cache add ca-certificates

WORKDIR /root/

COPY --from=0 /go/src/github.com/go/helloworld/app .

CMD ["./app"]

构建镜像

docker build -t go/helloworld:3 .

对比三个镜像大小

docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

go/helloworld 3 d6911ed9c846 7 seconds ago 6.47MB

go/helloworld 2 f7cf3465432c 22 seconds ago 6.47MB

go/helloworld 1 f55d3e16affc 2 minutes ago 295MB

很明显使用多阶段构建的镜像体积小，同时也完美解决了上边提到的问题。

**只构建某一阶段的镜像**

我们可以使用 as 来为某一阶段命名，例如

FROM golang:1.9-alpine as builder

例如当我们只想构建 builder 阶段的镜像时，我们可以在使用 docker build 命令时加上 --target 参数即可

docker build --target builder -t username/imagename:tag .

**构建时从其他镜像复制文件**

上面例子中我们使用 COPY --from=0 /go/src/github.com/go/helloworld/app . 从上一阶段的镜像中复制文件，我们也可以复制任意镜像中的文件。

COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf

## 其他制作镜像方式

除了标准的使用 Dockerfile 生成镜像的方法外，由于各种特殊需求和历史原因，还提供了一些其它方法用以生成镜像。

### 从 rootfs 压缩包导入

格式：docker import [选项] <文件>|<URL>|- [<仓库名>[:<标签>]]

压缩包可以是本地文件、远程 Web 文件，甚至是从标准输入中得到。压缩包将会在镜像 / 目录展开，并直接作为镜像第一层提交。

比如我们想要创建一个 OpenVZ 的 Ubuntu 14.04 模板的镜像：

docker import \

http://download.openvz.org/template/precreated/ubuntu-14.04-x86\_64-minimal.tar.gz \

openvz/ubuntu:14.04

这条命令自动下载了 ubuntu-14.04-x86\_64-minimal.tar.gz 文件，并且作为根文件系统展开导入，并保存为镜像 openvz/ubuntu:14.04。

导入成功后，我们可以用 docker image ls 看到这个导入的镜像：

docker image ls openvz/ubuntu

REPOSITORY TAG IMAGE ID CREATED SIZE

openvz/ubuntu 14.04 f477a6e18e98 55 seconds ago 214.9 MB

如果我们查看其历史的话，会看到描述中有导入的文件链接：

docker history openvz/ubuntu:14.04

IMAGE CREATED CREATED BY SIZE COMMENT

f477a6e18e98 About a minute ago 214.9 MB Imported from http://download.openvz.org/template/precreated/ubuntu-14.04-x86\_64-minimal.tar.gz

### docker save 和 docker load

Docker 还提供了 docker load 和 docker save 命令，用以将镜像保存为一个 tar 文件，然后传输到另一个位置上，再加载进来。这是在没有 Docker Registry 时的做法，现在已经不推荐，镜像迁移应该直接使用 Docker Registry，无论是直接使用 Docker Hub 还是使用内网私有 Registry 都可以。

**保存镜像**

使用 docker save 命令可以将镜像保存为归档文件。

比如我们希望保存这个 alpine 镜像。

docker image ls alpine

REPOSITORY TAG IMAGE ID CREATED SIZE

alpine latest baa5d63471ea 5 weeks ago 4.803 MB

保存镜像的命令为：

docker save alpine | gzip > alpine-latest.tar.gz

然后我们将 alpine-latest.tar.gz 文件复制到了到了另一个机器上，可以用下面这个命令加载镜像：

docker load -i alpine-latest.tar.gz

Loaded image: alpine:latest

如果我们结合这两个命令以及 ssh 甚至 pv 的话，利用 Linux 强大的管道，我们可以写一个命令完成从一个机器将镜像迁移到另一个机器，并且带进度条的功能：

docker save <镜像名> | bzip2 | pv | ssh <用户名>@<主机名> 'cat | docker load'

## 镜像的实现原理

Docker 镜像是怎么实现增量的修改和维护的？

每个镜像都由很多层次构成，Docker 使用 Union FS 将这些不同的层结合到一个镜像中去。

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

Docker 在 AUFS 上构建的容器也是利用了类似的原理。

# docker 容器

容器是 Docker 又一核心概念。

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

本章将具体介绍如何来管理一个容器，包括创建、启动和停止等。

## 启动容器

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

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

### 新建并启动

所需要的命令主要为 docker run。

例如，下面的命令输出一个 “Hello World”，之后终止容器。

docker run ubuntu:14.04 /bin/echo 'Hello world'

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

下面的命令则启动一个 bash 终端，允许用户进行交互。

docker run -t -i ubuntu:14.04 /bin/bash

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

在交互模式下，用户可以通过所创建的终端来输入命令，例如

root@af8bae53bdd3:/# pwd

/

root@af8bae53bdd3:/# ls

bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

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

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

### 启动已终止容器

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

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

root@ba267838cc1b:/# ps

PID TTY TIME CMD

1 ? 00:00:00 bash

11 ? 00:00:00 ps

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

## 守护状态运行

更多的时候，需要让 Docker 在后台运行而不是直接把执行命令的结果输出在当前宿主机下。此时，可以通过添加 -d 参数来实现。

下面举两个例子来说明一下。

如果不使用 -d 参数运行容器。

docker run ubuntu:17.10 /bin/sh -c "while true; do echo hello world; sleep 1; done"

hello world

hello world

hello world

hello world

容器会把输出的结果 (STDOUT) 打印到宿主机上面

如果使用了 -d 参数运行容器。

docker run -d ubuntu:17.10 /bin/sh -c "while true; do echo hello world; sleep 1; done"

此时容器会在后台运行并不会把输出的结果 (STDOUT) 打印到宿主机上面(输出结果可以用 docker logs 查看)。

注： 容器是否会长久运行，是和 docker run 指定的命令有关，和 -d 参数无关。

使用 -d 参数启动后会返回一个唯一的 id，也可以通过 docker container ls 命令来查看容器信息。

docker container ls

要获取容器的输出信息，可以通过 docker container logs 命令。

docker container logs [container ID or NAMES]

## 终止容器

可以使用 docker container stop 来终止一个运行中的容器。

此外，当 Docker 容器中指定的应用终结时，容器也自动终止。

例如对于上一章节中只启动了一个终端的容器，用户通过 exit 命令或 Ctrl+d 来退出终端时，所创建的容器立刻终止。

终止状态的容器可以用 docker container ls -a 命令看到。例如

docker container ls -a

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

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

## 进入容器

在使用 -d 参数时，容器启动后会进入后台。

某些时候需要进入容器进行操作，包括使用 docker attach 命令或 docker exec 命令，推荐大家使用 docker exec 命令，原因会在下面说明。

### attach 命令

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

docker run -dit ubuntu

docker run -dit Ubuntu

docker attach 243c

注意： 如果从这个 stdin 中 exit，会导致容器的停止。

### exec 命令

**-i -t 参数**

docker exec 后边可以跟多个参数，这里主要说明 -i -t 参数。

只用 -i 参数时，由于没有分配伪终端，界面没有我们熟悉的 Linux 命令提示符，但命令执行结果仍然可以返回。

当 -i -t 参数一起使用时，则可以看到我们熟悉的 Linux 命令提示符。

docker run -dit ubuntu

docker container ls

docker exec -it 69d1 bash

docker exec -i 69d1 bash

如果从这个 stdin 中 exit，不会导致容器的停止。这就是为什么推荐大家使用 docker exec 的原因。

更多参数说明请使用 docker exec --help 查看。

## 导出和导入容器

### 导出容器

如果要导出本地某个容器，可以使用 docker export 命令。

docker container ls -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

7691a814370e ubuntu:14.04 "/bin/bash" 36 hours ago Exited (0) 21 hours ago test

docker export 7691a814370e > ubuntu.tar

这样将导出容器快照到本地文件。

### 导入容器快照

可以使用 docker import 从容器快照文件中再导入为镜像，例如

cat ubuntu.tar | docker import - test/ubuntu:v1.0

docker image ls

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

test/ubuntu v1.0 9d37a6082e97 About a minute ago 171.3 MB

此外，也可以通过指定 URL 或者某个目录来导入，例如

docker import http://example.com/exampleimage.tgz example/imagerepo

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

## 删除容器

可以使用 docker container rm 来删除一个处于终止状态的容器。例如

docker container rm trusting\_newton

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

#清理所有处于终止状态的容器

用 docker container ls -a 命令可以查看所有已经创建的包括终止状态的容器，如果数量太多要一个个删除可能会很麻烦，用下面的命令可以清理掉所有处于终止状态的容器。

docker container prune

# docker 仓库

仓库（Repository）是集中存放镜像的地方。

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

大部分时候，并不需要严格区分这两者的概念。

## docker hub

### 注册

你可以在 https://hub.docker.com/ 免费注册一个 Docker 账号。

### 登录

可以通过执行 docker login 命令交互式的输入用户名及密码来完成在命令行界面登录 Docker Hub。

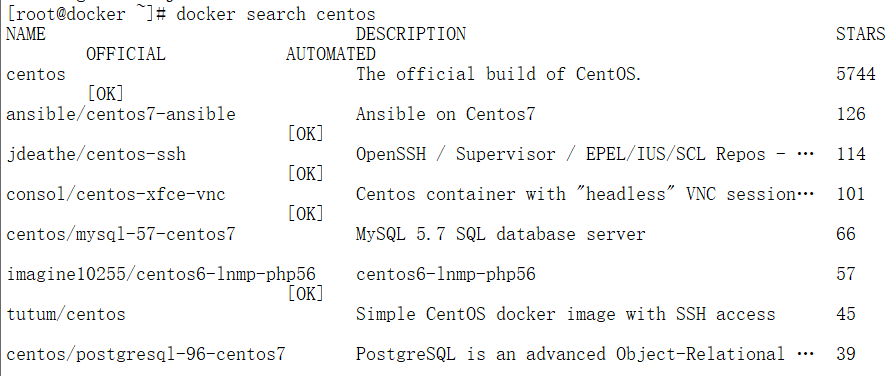
你可以通过 docker logout 退出登录。

### 拉取镜像

你可以通过 docker search 命令来查找官方仓库中的镜像，并利用 docker pull 命令来将它下载到本地。

例如以 centos 为关键词进行搜索：

docker search centos



可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、收藏数（表示该镜像的受关注程度）、是否官方创建、是否自动创建。

官方的镜像说明是官方项目组创建和维护的，automated 资源允许用户验证镜像的来源和内容。

根据是否是官方提供，可将镜像资源分为两类。

一种是类似 centos 这样的镜像，被称为基础镜像或根镜像。这些基础镜像由 Docker 公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。

还有一种类型，比如 tianon/centos 镜像，它是由 Docker 的用户创建并维护的，往往带有用户名称前缀。可以通过前缀 username/ 来指定使用某个用户提供的镜像，比如 tianon 用户。

另外，在查找的时候通过 --filter=stars=N 参数可以指定仅显示收藏数量为 N 以上的镜像。

下载官方 centos 镜像到本地。

docker pull centos

Pulling repository centos

0b443ba03958: Download complete

539c0211cd76: Download complete

511136ea3c5a: Download complete

7064731afe90: Download complete

### 推送镜像

用户也可以在登录后通过 docker push 命令来将自己的镜像推送到 Docker Hub。

以下命令中的 username 请替换为你的 Docker 账号用户名

docker tag ubuntu:17.10 username/ubuntu:17.10

docker image ls

docker push username/ubuntu:17.10

docker search username

### 自动创建

自动创建（Automated Builds）功能对于需要经常升级镜像内程序来说，十分方便。

有时候，用户创建了镜像，安装了某个软件，如果软件发布新版本则需要手动更新镜像。

而自动创建允许用户通过 Docker Hub 指定跟踪一个目标网站（目前支持 GitHub 或 BitBucket）上的项目，一旦项目发生新的提交或者创建新的标签（tag），Docker Hub 会自动构建镜像并推送到 Docker Hub 中。

要配置自动创建，包括如下的步骤：

创建并登录 Docker Hub，以及目标网站；

在目标网站中连接帐户到 Docker Hub；

在 Docker Hub 中 配置一个自动创建；

选取一个目标网站中的项目（需要含 Dockerfile）和分支；

指定 Dockerfile 的位置，并提交创建。

之后，可以在 Docker Hub 的 自动创建页面 中跟踪每次创建的状态。

## docker 私有仓库

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

本节介绍如何使用本地仓库。

[docker-registry](https://docs.docker.com/registry/spec/api/) 是官方提供的工具，可以用于构建私有的镜像仓库。本文内容基于 [docker-registry v2.x](https://github.com/docker/distribution) 版本。

### 安装运行 docker-registry

你可以通过获取官方 registry 镜像来运行。

docker run -d -p 5000:5000 --restart=always --name registry registry

这将使用官方的 registry 镜像来启动私有仓库。默认情况下，仓库会被创建在容器的 /var/lib/registry 目录下。你可以通过 -v 参数来将镜像文件存放在本地的指定路径。例如下面的例子将上传的镜像放到本地的 /opt/data/registry 目录。

docker run -d \

-p 5000:5000 \

-v /opt/data/registry:/var/lib/registry \

registry

### 在私有仓库上传、搜索、下载镜像

创建好私有仓库之后，就可以使用 docker tag 来标记一个镜像，然后推送它到仓库。例如私有仓库地址为 127.0.0.1:5000。

先在本机查看已有的镜像。

docker image ls

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

ubuntu latest ba5877dc9bec 6 weeks ago 192.7 MB

使用 docker tag 将 ubuntu:latest 这个镜像标记为 127.0.0.1:5000/ubuntu:latest。

格式为 docker tag IMAGE[:TAG] [REGISTRY\_HOST[:REGISTRY\_PORT]/]REPOSITORY[:TAG]。

docker tag ubuntu:latest 127.0.0.1:5000/ubuntu:latest

docker image ls

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

ubuntu latest ba5877dc9bec 6 weeks ago 192.7 MB

127.0.0.1:5000/ubuntu:latest latest ba5877dc9bec 6 weeks ago 192.7 MB

使用 docker push 上传标记的镜像。

docker push 127.0.0.1:5000/ubuntu:latest

用 curl 查看仓库中的镜像。

curl 127.0.0.1:5000/v2/\_catalog

{"repositories":["ubuntu"]}

这里可以看到 {"repositories":["ubuntu"]}，表明镜像已经被成功上传了。

先删除已有镜像，再尝试从私有仓库中下载这个镜像。

docker image rm 127.0.0.1:5000/ubuntu:latest

docker pull 127.0.0.1:5000/ubuntu:latestdocker image ls

docker image ls

### 注意事项

如果你不想使用 127.0.0.1:5000 作为仓库地址，比如想让本网段的其他主机也能把镜像推送到私有仓库。你就得把例如 192.168.199.100:5000 这样的内网地址作为私有仓库地址，这时你会发现无法成功推送镜像。

这是因为 Docker 默认不允许非 HTTPS 方式推送镜像。我们可以通过 Docker 的配置选项来取消这个限制，或者查看下一节配置能够通过 HTTPS 访问的私有仓库。

#### Ubuntu 14.04, Debian 7 Wheezy

对于使用 upstart 的系统而言，编辑 /etc/default/docker 文件，在其中的 DOCKER\_OPTS 中增加如下内容：

DOCKER\_OPTS="--registry-mirror=https://registry.docker-cn.com --insecure-registries=192.168.199.100:5000"

重新启动服务。

sudo service docker restart

#### Ubuntu 16.04+, Debian 8+, centos 7

对于使用 systemd 的系统，请在 /etc/docker/daemon.json 中写入如下内容（如果文件不存在请新建该文件）

{

"registry-mirrors": [

"https://registry.docker-cn.com"

],

"insecure-registries": [

"192.168.199.100:5000"

]

}

注意：该文件必须符合 json 规范，否则 Docker 将不能启动。

### 其他

对于 Docker for Windows 、 Docker for Mac 在设置中编辑 daemon.json 增加和上边一样的字符串即可。

### docker registry 配置证书

上一节我们搭建了一个具有基础功能的私有仓库，本小节我们来使用 Docker Compose 搭建一个拥有权限认证、TLS 的私有仓库。

新建一个文件夹，以下步骤均在该文件夹中进行。

#### 准备站点证书

如果你拥有一个域名，国内各大云服务商均提供免费的站点证书。你也可以使用 openssl 自行签发证书。

这里假设我们将要搭建的私有仓库地址为 docker.domain.com，下面我们介绍使用 openssl 自行签发 docker.domain.com 的站点 SSL 证书。

第一步创建 CA 私钥。

openssl genrsa -out "root-ca.key" 4096

第二步利用私钥创建 CA 根证书请求文件。

openssl req \

-new -key "root-ca.key" \

-out "root-ca.csr" -sha256 \

-subj '/C=CN/ST=Shanxi/L=Datong/O=Your Company Name/CN=Your Company Name Docker Registry CA'

以上命令中 -subj 参数里的 /C 表示国家，如 CN；/ST 表示省；/L 表示城市或者地区；/O 表示组织名；/CN 通用名称。

第三步配置 CA 根证书，新建 root-ca.cnf。

[root\_ca]

basicConstraints = critical,CA:TRUE,pathlen:1

keyUsage = critical, nonRepudiation, cRLSign, keyCertSign

subjectKeyIdentifier=hash

第四步签发根证书。

openssl x509 -req -days 3650 -in "root-ca.csr" \

-signkey "root-ca.key" -sha256 -out "root-ca.crt" \

-extfile "root-ca.cnf" -extensions \

root\_ca

第五步生成站点 SSL 私钥。

openssl genrsa -out "docker.domain.com.key" 4096

第六步使用私钥生成证书请求文件。

openssl req -new -key "docker.domain.com.key" -out "site.csr" -sha256 \

-subj '/C=CN/ST=Shanxi/L=Datong/O=Your Company Name/CN=docker.domain.com'

第七步配置证书，新建 site.cnf 文件。

[server]

authorityKeyIdentifier=keyid,issuer

basicConstraints = critical,CA:FALSE

extendedKeyUsage=serverAuth

keyUsage = critical, digitalSignature, keyEncipherment

subjectAltName = DNS:docker.domain.com, IP:127.0.0.1

subjectKeyIdentifier=hash

第八步签署站点 SSL 证书。

openssl x509 -req -days 750 -in "site.csr" -sha256 \

-CA "root-ca.crt" -CAkey "root-ca.key" -CAcreateserial \

-out "docker.domain.com.crt" -extfile "site.cnf" -extensions server

这样已经拥有了 docker.domain.com 的网站 SSL 私钥 docker.domain.com.key 和 SSL 证书 docker.domain.com.crt。

新建 ssl 文件夹并将 docker.domain.com.key docker.domain.com.crt 这两个文件移入，删除其他文件。

#### 配置私有仓库

私有仓库默认的配置文件位于 /etc/docker/registry/config.yml，我们先在本地编辑 config.yml，之后挂载到容器中。

version: 0.1

log:

accesslog:

disabled: true

level: debug

formatter: text

fields:

service: registry

environment: staging

storage:

delete:

enabled: true

cache:

blobdescriptor: inmemory

filesystem:

rootdirectory: /var/lib/registry

auth:

htpasswd:

realm: basic-realm

path: /etc/docker/registry/auth/nginx.htpasswd

http:

addr: :443

host: https://docker.domain.com

headers:

X-Content-Type-Options: [nosniff]

http2:

disabled: false

tls:

certificate: /etc/docker/registry/ssl/docker.domain.com.crt

key: /etc/docker/registry/ssl/docker.domain.com.key

health:

storagedriver:

enabled: true

interval: 10s

threshold: 3

#### 生成 http 认证文件

mkdir auth

docker run --rm \

--entrypoint htpasswd \

registry \

-Bbn username password > auth/nginx.htpasswd

将上面的 username password 替换为你自己的用户名和密码。

#### 编辑 docker-compose.yml

version: '3'

services:

registry:

image: registry

ports:

- "443:443"

volumes:

- ./:/etc/docker/registry

- registry-data:/var/lib/registry

volumes:

registry-data:

#### 修改 hosts

编辑 /etc/hosts

docker.domain.com 127.0.0.1

#### 启动

docker-compose up -d

这样我们就搭建好了一个具有权限认证、TLS 的私有仓库，接下来我们测试其功能是否正常。

#### 测试私有仓库功能

登录到私有仓库。

docker login docker.domain.com

尝试推送、拉取镜像。

docker pull ubuntu:17.10

docker tag ubuntu:17.10 docker.domain.com/username/ubuntu:17.10

docker push docker.domain.com/username/ubuntu:17.10

docker image rm docker.domain.com/username/ubuntu:17.10

docker pull docker.domain.com/username/ubuntu:17.10

如果我们退出登录，尝试推送镜像。

docker logout docker.domain.com

docker push docker.domain.com/username/ubuntu:17.10

发现会提示没有登录，不能将镜像推送到私有仓库中。

#### 注意事项

如果你本机占用了 443 端口，你可以配置 [Nginx 代理](https://docs.docker.com/registry/recipes/nginx/)，这里不再赘述。

## docker 企业级仓库 Harbor

在学完 compose 后再来安装部署。

# docker 数据管理

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

* 数据卷（Volumes）
* 挂载主机目录 (Bind mounts)

## docker 数据卷

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

* 数据卷 可以在容器之间共享和重用
* 对 数据卷 的修改会立马生效
* 对 数据卷 的更新，不会影响镜像
* 数据卷 默认会一直存在，即使容器被删除

注意：数据卷 的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的 数据卷。

### 选择 -v 还是 -–mount 参数

Docker 新用户应该选择 --mount 参数，经验丰富的 Docker 使用者对 -v 或者 --volume 已经很熟悉了，但是推荐使用 --mount 参数。

### 创建一个数据卷

docker volume create my-vol

查看所有的 数据卷

docker volume ls

在主机里使用以下命令可以查看指定 数据卷 的信息

docker volume inspect my-vol

[

{

"Driver": "local",

"Labels": {},

"Mountpoint": "/var/lib/docker/volumes/my-vol/\_data",

"Name": "my-vol",

"Options": {},

"Scope": "local"

}

]

### 启动一个挂载数据卷的容器

在用 docker run 命令的时候，使用 --mount 标记来将 数据卷 挂载到容器里。在一次 docker run 中可以挂载多个 数据卷。

下面创建一个名为 web 的容器，并加载一个 数据卷 到容器的 /webapp 目录。

docker run -d -P \

--name web \

# -v my-vol:/wepapp \

--mount source=my-vol,target=/webapp \

training/webapp \

python app.py

### 查看数据卷的具体信息

在主机里使用以下命令可以查看 web 容器的信息

docker inspect web

数据卷 信息在 "Mounts" Key 下面

"Mounts": [

{

"Type": "volume",

"Name": "my-vol",

"Source": "/var/lib/docker/volumes/my-vol/\_data",

"Destination": "/app",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

],

### 删除数据卷

docker volume rm my-vol

数据卷 是被设计用来持久化数据的，它的生命周期独立于容器，Docker 不会在容器被删除后自动删除 数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的 数据卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用 docker rm -v 这个命令。

无主的数据卷可能会占据很多空间，要清理请使用以下命令

docker volume prune

## 监听主机目录

### -v 还是 -–mount 参数

Docker 新用户应该选择 --mount 参数，经验丰富的 Docker 使用者对 -v 或者 --volume 已经很熟悉了，但是推荐使用 --mount 参数。

### 挂载一个主机目录作为数据卷

使用 --mount 标记可以指定挂载一个本地主机的目录到容器中去。

docker run -d -P \

--name web \

# -v /src/webapp:/opt/webapp \

--mount type=bind,source=/src/webapp,target=/opt/webapp \

training/webapp \

python app.py

上面的命令加载主机的 /src/webapp 目录到容器的 /opt/webapp目录。这个功能在进行测试的时候十分方便，比如用户可以放置一些程序到本地目录中，来查看容器是否正常工作。本地目录的路径必须是绝对路径，以前使用 -v 参数时如果本地目录不存在 Docker 会自动为你创建一个文件夹，现在使用 --mount 参数时如果本地目录不存在，Docker 会报错。

Docker 挂载主机目录的默认权限是 读写，用户也可以通过增加 readonly 指定为 只读。

docker run -d -P \

--name web \

# -v /src/webapp:/opt/webapp:ro \

--mount type=bind,source=/src/webapp,target=/opt/webapp,readonly \

training/webapp \

python app.py

加了 readonly 之后，就挂载为 只读 了。如果你在容器内 /opt/webapp 目录新建文件，会显示如下错误

/opt/webapp # touch new.txt

touch: new.txt: Read-only file system

### 查看数据卷的具体信息

在主机里使用以下命令可以查看 web 容器的信息

docker inspect web

挂载主机目录 的配置信息在 "Mounts" Key 下面

"Mounts": [

{

"Type": "bind",

"Source": "/src/webapp",

"Destination": "/opt/webapp",

"Mode": "",

"RW": true,

"Propagation": "rprivate"

}

],

### 挂载一个本地主机文件作为数据卷

--mount 标记也可以从主机挂载单个文件到容器中

docker run --rm -it \

# -v $HOME/.bash\_history:/root/.bash\_history \

--mount type=bind,source=$HOME/.bash\_history,target=/root/.bash\_history \

ubuntu:17.10 \

bash

root@2affd44b4667:/# history

1 ls

2 diskutil list

这样就可以记录在容器输入过的命令了。

# docker 网络配置

Docker 允许通过外部访问容器或容器互联的方式来提供网络服务

## Docker 外部访问容器

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

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

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

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

docker container ls -l

同样的，可以通过 docker logs 命令来查看应用的信息。

docker logs -f nostalgic\_morse

\* Running on http://0.0.0.0:5000/

10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -

10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -

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

### 映射所有接口地址

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

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

此时默认会绑定本地所有接口上的所有地址。

### 映射到指定地址的指定端口

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

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

### 映射到指定地址的任意端口

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

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

还可以使用 udp 标记来指定 udp 端口

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

### 查看映射端口配置

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

docker port nostalgic\_morse 5000

注意：

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

例如

docker run -d \

-p 5000:5000 \

-p 3000:80 \

training/webapp \

python app.py

## Docker 容器互连

如果你之前有 Docker 使用经验，你可能已经习惯了使用 --link 参数来使容器互联。

随着 Docker 网络的完善，强烈建议大家将容器加入自定义的 Docker 网络来连接多个容器，而不是使用 --link 参数。

### 新建网络

下面先创建一个新的 Docker 网络。

docker network create -d bridge my-net

-d 参数指定 Docker 网络类型，有 bridge overlay。其中 overlay 网络类型用于 Swarm mode，在本小节中你可以忽略它。

### 连接容器

运行一个容器并连接到新建的 my-net 网络

docker run -it --rm --name busybox1 --network my-net busybox sh

打开新的终端，再运行一个容器并加入到 my-net 网络

docker run -it --rm --name busybox2 --network my-net busybox sh

再打开一个新的终端查看容器信息

docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

b47060aca56b busybox "sh" 11 minutes ago Up 11 minutes busybox2

8720575823ec busybox "sh" 16 minutes ago Up 16 minutes busybox1

下面通过 ping 来证明 busybox1 容器和 busybox2 容器建立了互联关系。

在 busybox1 容器输入以下命令

/ # ping busybox2

PING busybox2 (172.19.0.3): 56 data bytes

64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.072 ms

64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.118 ms

用 ping 来测试连接 busybox2 容器，它会解析成 172.19.0.3。

同理在 busybox2 容器执行 ping busybox1，也会成功连接到。

/ # ping busybox1

PING busybox1 (172.19.0.2): 56 data bytes

64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.064 ms

64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.143 ms

这样，busybox1 容器和 busybox2 容器建立了互联关系。

### Docker Compose

如果你有多个容器之间需要互相连接，推荐使用 Docker Compose。

## Docker 配置 DNS

如何自定义配置容器的主机名和 DNS 呢？秘诀就是 Docker 利用虚拟文件来挂载容器的 3 个相关配置文件。

在容器中使用 mount 命令可以看到挂载信息：

mount

/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...

/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...

tmpfs on /etc/resolv.conf type tmpfs ...

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

配置全部容器的 DNS ，也可以在 /etc/docker/daemon.json 文件中增加以下内容来设置。

{

"dns" : [

"114.114.114.114",

"8.8.8.8"

]

}

这样每次启动的容器 DNS 自动配置为 114.114.114.114 和 8.8.8.8。使用以下命令来证明其已经生效。

docker run -it --rm ubuntu:17.10 cat etc/resolv.conf

nameserver 114.114.114.114

nameserver 8.8.8.8

如果用户想要手动指定容器的配置，可以在使用 docker run 命令启动容器时加入如下参数：

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

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

--dns-search=DOMAIN 设定容器的搜索域，当设定搜索域为 .example.com 时，在搜索一个名为 host 的主机时，DNS 不仅搜索 host，还会搜索 host.example.com。

注意：如果在容器启动时没有指定最后两个参数，Docker 会默认用主机上的 /etc/resolv.conf 来配置容器。

# docker 高级网络配置

注意：本章属于 Docker 高级配置，如果您是初学者，您可以暂时跳过本章节，直接学习 Docker Compose 一节。

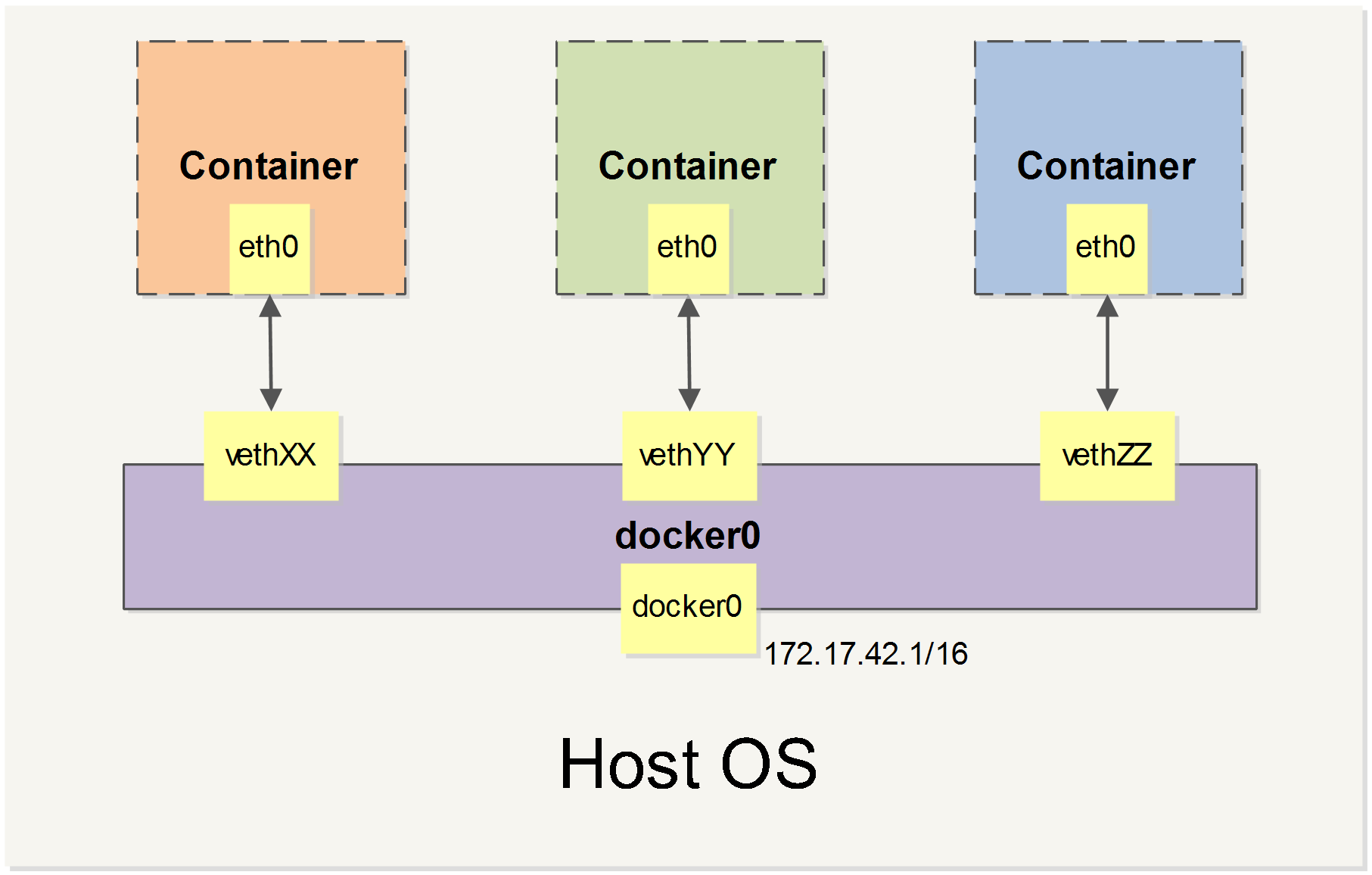
本章将介绍 Docker 的一些高级网络配置和选项。

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

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

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

Docker 网络



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

## 快速配置指南

下面是一个跟 Docker 网络相关的命令列表。

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

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

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

* --dns=IP\_ADDRESS... 使用指定的DNS服务器
* --dns-search=DOMAIN... 指定DNS搜索域

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

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

## 容器访问控制

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

### 容器访问外部网络

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

sysctl net.ipv4.ip\_forward

net.ipv4.ip\_forward = 1

如果为 0，说明没有开启转发，则需要手动打开。

sysctl -w net.ipv4.ip\_forward=1

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

### 容器之间访问

容器之间相互访问，需要两方面的支持。

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

**访问所有端口**

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

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

**访问指定端口**

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

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

此时，系统中的 iptables 规则可能是类似

sudo iptables -nL

...

Chain FORWARD (policy ACCEPT)

target prot opt source destination

DROP all -- 0.0.0.0/0 0.0.0.0/0

...

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

当添加了 --link=CONTAINER\_NAME:ALIAS 选项后，添加了 iptables 规则。

sudo iptables -nL

Chain FORWARD (policy ACCEPT)

target prot opt source destination

ACCEPT tcp -- 172.17.0.2 172.17.0.3 tcp spt:80

ACCEPT tcp -- 172.17.0.3 172.17.0.2 tcp dpt:80

DROP all -- 0.0.0.0/0 0.0.0.0/0

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

## 端口映射实现

默认情况下，容器可以主动访问到外部网络的连接，但是外部网络无法访问到容器。

### 容器访问外部实现

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

查看主机的 NAT 规则。

sudo iptables -t nat -nL

Chain POSTROUTING (policy ACCEPT)

target prot opt source destination

MASQUERADE all -- 172.17.0.0/16 !172.17.0.0/16

...

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

### 外部访问容器实现

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

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

使用 -P 时：

iptables -t nat -nL

Chain DOCKER (2 references)

target prot opt source destination

DNAT tcp -- 0.0.0.0/0 0.0.0.0/0 tcp dpt:49153 to:172.17.0.2:80

使用 -p 80:80 时：

iptables -t nat -nL

Chain DOCKER (2 references)

target prot opt source destination

DNAT tcp -- 0.0.0.0/0 0.0.0.0/0 tcp dpt:80 to:172.17.0.2:80

注意：

* 这里的规则映射了 0.0.0.0，意味着将接受主机来自所有接口的流量。用户可以通过 -p IP:host\_port:container\_port 或 -p IP::port 来指定允许访问容器的主机上的 IP、接口等，以制定更严格的规则。
* 如果希望永久绑定到某个固定的 IP 地址，可以在 Docker 配置文件 /etc/docker/daemon.json 中添加如下内容。

{

"ip": "0.0.0.0"

}

## 配置 docker0 网桥

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

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

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

也可以在配置文件中配置 DOCKER\_OPTS，然后重启服务。

由于目前 Docker 网桥是 Linux 网桥，用户可以使用 brctl show 来查看网桥和端口连接信息。

sudo brctl show

bridge name bridge id STP enabled interfaces

docker0 8000.3a1d7362b4ee no veth65f9 vethdda6

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

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

sudo docker run -i -t --rm base /bin/bash

ip addr show eth0

24: eth0: <BROADCAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UP group default qlen 1000

link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff

inet 172.17.0.3/16 scope global eth0

valid\_lft forever preferred\_lft forever

inet6 fe80::306f:e0ff:fe35:5791/64 scope link

valid\_lft forever preferred\_lft forever

ip route

default via 172.17.42.1 dev eth0

172.17.0.0/16 dev eth0 proto kernel scope link src 172.17.0.3

## 自定义网桥

除了默认的 docker0 网桥，用户也可以指定网桥来连接各个容器。

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

如果服务已经运行，那需要先停止服务，并删除旧的网桥。

sudo systemctl stop docker

sudo ip link set dev docker0 down

sudo brctl delbr docker0

然后创建一个网桥 bridge0。

sudo brctl addbr bridge0

sudo ip addr add 192.168.5.1/24 dev bridge0

sudo ip link set dev bridge0 up

查看确认网桥创建并启动。

ip addr show bridge0

4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default

link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff

inet 192.168.5.1/24 scope global bridge0

valid\_lft forever preferred\_lft forever

在 Docker 配置文件 /etc/docker/daemon.json 中添加如下内容，即可将 Docker 默认桥接到创建的网桥上。

{

"bridge": "bridge0"

}

启动 Docker 服务。

新建一个容器，可以看到它已经桥接到了 bridge0 上。

可以继续用 brctl show 命令查看桥接的信息。另外，在容器中可以使用 ip addr 和 ip route 命令来查看 IP 地址配置和路由信息。

## 工具和示例

在介绍自定义网络拓扑之前，你可能会对一些外部工具和例子感兴趣：

### pipework

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

### playground

Brandon Rhodes 创建了一个提供完整的 Docker 容器网络拓扑管理的 [Python库](https://github.com/brandon-rhodes/fopnp/tree/m/playground)，包括路由、NAT 防火墙；以及一些提供 HTTP, SMTP, POP, IMAP, Telnet, SSH, FTP 的服务器。

## 编辑网络配置文件

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

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

## 实例：创建一个点到点连接

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

用户有时候需要两个容器之间可以直连通信，而不用通过主机网桥进行桥接。

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

首先启动 2 个容器：

docker run -i -t --rm --net=none base /bin/bash

docker run -i -t --rm --net=none base /bin/bash

找到进程号，然后创建网络命名空间的跟踪文件。

docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a

2989

docker inspect -f '{{.State.Pid}}' 12e343489d2f

3004

sudo mkdir -p /var/run/netns

sudo ln -s /proc/2989/ns/net /var/run/netns/2989

sudo ln -s /proc/3004/ns/net /var/run/netns/3004

创建一对 peer 接口，然后配置路由

sudo ip link add A type veth peer name B

sudo ip link set A netns 2989

sudo ip netns exec 2989 ip addr add 10.1.1.1/32 dev A

sudo ip netns exec 2989 ip link set A up

sudo ip netns exec 2989 ip route add 10.1.1.2/32 dev A

sudo ip link set B netns 3004

sudo ip netns exec 3004 ip addr add 10.1.1.2/32 dev B

sudo ip netns exec 3004 ip link set B up

sudo ip netns exec 3004 ip route add 10.1.1.1/32 dev B

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

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

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

# docker 三剑客 compose

Docker Compose 是 Docker 官方编排（Orchestration）项目之一，负责快速的部署分布式应用。

## docker compose 简介

Compose 项目是 Docker 官方的开源项目，负责实现对 Docker 容器集群的快速编排。从功能上看，跟 OpenStack 中的 Heat 十分类似。

其代码目前在 <https://github.com/docker/compose> 上开源。

Compose 定位是 「定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker applications）」，其前身是开源项目 Fig。

通过第一部分中的介绍，我们知道使用一个 Dockerfile 模板文件，可以让用户很方便的定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等。

Compose 恰好满足了这样的需求。它允许用户通过一个单独的 docker-compose.yml 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

Compose 中有两个重要的概念：

* 服务 (service)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
* 项目 (project)：由一组关联的应用容器组成的一个完整业务单元，在 docker-compose.yml 文件中定义。

Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。

Compose 项目由 Python 编写，实现上调用了 Docker 服务提供的 API 来对容器进行管理。因此，只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

## docker compose 安装和卸载

Compose 支持 Linux、macOS、Windows 10 三大平台。

Compose 可以通过 Python 的包管理工具 pip 进行安装，也可以直接下载编译好的二进制文件使用，甚至能够直接在 Docker 容器中运行。

前两种方式是传统方式，适合本地环境下安装使用；最后一种方式则不破坏系统环境，更适合云计算场景。

Docker for Mac 、Docker for Windows 自带 docker-compose 二进制文件，安装 Docker 之后可以直接使用。

docker-compose --version

Linux 系统请使用以下介绍的方法安装。

### 二进制包

在 Linux 上的也安装十分简单，从 官方 [GitHub Release](https://github.com/docker/compose/releases) 处直接下载编译好的二进制文件即可。

例如，在 Linux 64 位系统上直接下载对应的二进制包。

sudo curl -L https://github.com/docker/compose/releases/download/1.17.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

### PIP 安装

注： x86\_64 架构的 Linux 建议按照上边的方法下载二进制包进行安装，如果您计算机的架构是 ARM (例如，树莓派)，再使用 pip 安装。

这种方式是将 Compose 当作一个 Python 应用来从 pip 源中安装。

执行安装命令：

sudo pip install -U docker-compose

可以看到类似如下输出，说明安装成功。

Collecting docker-compose

Downloading docker-compose-1.17.1.tar.gz (149kB): 149kB downloaded

...

Successfully installed docker-compose cached-property requests texttable websocket-client docker-py dockerpty six enum34 backports.ssl-match-hostname ipaddress

### bash 补全命令

curl -L https://raw.githubusercontent.com/docker/compose/1.8.0/contrib/completion/bash/docker-compose > /etc/bash\_completion.d/docker-compose

### 容器中执行

Compose 既然是一个 Python 应用，自然也可以直接用容器来执行它。

curl -L https://github.com/docker/compose/releases/download/1.8.0/run.sh > /usr/local/bin/docker-compose

chmod +x /usr/local/bin/docker-compose

实际上，查看下载的 run.sh 脚本内容，如下

set -e

VERSION="1.8.0"

IMAGE="docker/compose:$VERSION"

# Setup options for connecting to docker host

if [ -z "$DOCKER\_HOST" ]; then

DOCKER\_HOST="/var/run/docker.sock"

fi

if [ -S "$DOCKER\_HOST" ]; then

DOCKER\_ADDR="-v $DOCKER\_HOST:$DOCKER\_HOST -e DOCKER\_HOST"

else

DOCKER\_ADDR="-e DOCKER\_HOST -e DOCKER\_TLS\_VERIFY -e DOCKER\_CERT\_PATH"

fi

# Setup volume mounts for compose config and context

if [ "$(pwd)" != '/' ]; then

VOLUMES="-v $(pwd):$(pwd)"

fi

if [ -n "$COMPOSE\_FILE" ]; then

compose\_dir=$(dirname $COMPOSE\_FILE)

fi

# TODO: also check --file argument

if [ -n "$compose\_dir" ]; then

VOLUMES="$VOLUMES -v $compose\_dir:$compose\_dir"

fi

if [ -n "$HOME" ]; then

VOLUMES="$VOLUMES -v $HOME:$HOME -v $HOME:/root" # mount $HOME in /root to share docker.config

fi

# Only allocate tty if we detect one

if [ -t 1 ]; then

DOCKER\_RUN\_OPTIONS="-t"

fi

if [ -t 0 ]; then

DOCKER\_RUN\_OPTIONS="$DOCKER\_RUN\_OPTIONS -i"

fi

exec docker run --rm $DOCKER\_RUN\_OPTIONS $DOCKER\_ADDR $COMPOSE\_OPTIONS $VOLUMES -w "$(pwd)" $IMAGE "$@"

可以看到，它其实是下载了 docker/compose 镜像并运行。

### 卸载

如果是二进制包方式安装的，删除二进制文件即可。

sudo rm /usr/local/bin/docker-compose

如果是通过 pip 安装的，则执行如下命令即可删除。

sudo pip uninstall docker-compose

## docker compose 使用

### 术语

首先介绍几个术语。

* 服务 (service)：一个应用容器，实际上可以运行多个相同镜像的实例。
* 项目 (project)：由一组关联的应用容器组成的一个完整业务单元。

可见，一个项目可以由多个服务（容器）关联而成，Compose 面向项目进行管理。

### 场景

最常见的项目是 web 网站，该项目应该包含 web 应用和缓存。

下面我们用 Python 来建立一个能够记录页面访问次数的 web 网站。

**web 应用**

新建文件夹，在该目录中编写 app.py 文件

from flask import Flask

from redis import Redis

app = Flask(\_\_name\_\_)

redis = Redis(host='redis', port=6379)

@app.route('/')

def hello():

count = redis.incr('hits')

return 'Hello World! 该页面已被访问 {} 次。\n'.format(count)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", debug=True)

**Dockerfile**

编写 Dockerfile 文件，内容为

FROM python:3.6-alpine

ADD . /code

WORKDIR /code

RUN pip install redis flask

CMD ["python", "app.py"]

**docker-compose.yml**

编写 docker-compose.yml 文件，这个是 Compose 使用的主模板文件。

version: '3'

services:

web:

build: .

ports:

- "5000:5000"

redis:

image: "redis:alpine"

**运行 compose 项目**

docker-compose up

此时访问本地 5000 端口，每次刷新页面，计数就会加 1。

## docker compose 命令说明

### 命令对象与格式

对于 Compose 来说，大部分命令的对象既可以是项目本身，也可以指定为项目中的服务或者容器。如果没有特别的说明，命令对象将是项目，这意味着项目中所有的服务都会受到命令影响。

执行 docker-compose [COMMAND] --help 或者 docker-compose help [COMMAND] 可以查看具体某个命令的使用格式。

docker-compose 命令的基本的使用格式是

docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]

### 命令选项

* -f, --file FILE 指定使用的 Compose 模板文件，默认为 docker-compose.yml，可以多次指定。
* -p, --project-name NAME 指定项目名称，默认将使用所在目录名称作为项目名。
* --x-networking 使用 Docker 的可拔插网络后端特性
* --x-network-driver DRIVER 指定网络后端的驱动，默认为 bridge
* --verbose 输出更多调试信息。
* -v, --version 打印版本并退出。

### 命令使用说明

#### build

格式为 docker-compose build [options] [SERVICE...]。

构建（重新构建）项目中的服务容器。

服务容器一旦构建后，将会带上一个标记名，例如对于 web 项目中的一个 db 容器，可能是 web\_db。

可以随时在项目目录下运行 docker-compose build 来重新构建服务。

选项包括：

* --force-rm 删除构建过程中的临时容器。
* --no-cache 构建镜像过程中不使用 cache（这将加长构建过程）。
* --pull 始终尝试通过 pull 来获取更新版本的镜像。

#### config

验证 Compose 文件格式是否正确，若正确则显示配置，若格式错误显示错误原因。

#### down

此命令将会停止 up 命令所启动的容器，并移除网络

#### exec

进入指定的容器。

#### help

获得一个命令的帮助。

#### images

列出 Compose 文件中包含的镜像。

#### kill

格式为 docker-compose kill [options] [SERVICE...]。

通过发送 SIGKILL 信号来强制停止服务容器。

支持通过 -s 参数来指定发送的信号，例如通过如下指令发送 SIGINT 信号。

docker-compose kill -s SIGINT

#### logs

格式为 docker-compose logs [options] [SERVICE...]。

查看服务容器的输出。默认情况下，docker-compose 将对不同的服务输出使用不同的颜色来区分。可以通过 --no-color 来关闭颜色。

该命令在调试问题的时候十分有用。

#### pause

格式为 docker-compose pause [SERVICE...]。

暂停一个服务容器。

#### port

格式为 docker-compose port [options] SERVICE PRIVATE\_PORT。

打印某个容器端口所映射的公共端口。

选项：

* --protocol=proto 指定端口协议，tcp（默认值）或者 udp。
* --index=index 如果同一服务存在多个容器，指定命令对象容器的序号（默认为 1）。

#### ps

格式为 docker-compose ps [options] [SERVICE...]。

列出项目中目前的所有容器。

选项：

* -q 只打印容器的 ID 信息。

#### pull

格式为 docker-compose pull [options] [SERVICE...]。

拉取服务依赖的镜像。

选项：

* --ignore-pull-failures 忽略拉取镜像过程中的错误。

#### push

推送服务依赖的镜像到 Docker 镜像仓库。

#### restart

格式为 docker-compose restart [options] [SERVICE...]。

重启项目中的服务。

选项：

* -t, --timeout TIMEOUT 指定重启前停止容器的超时（默认为 10 秒）。

#### rm

格式为 docker-compose rm [options] [SERVICE...]。

删除所有（停止状态的）服务容器。推荐先执行 docker-compose stop 命令来停止容器。

选项：

* -f, --force 强制直接删除，包括非停止状态的容器。一般尽量不要使用该选项。
* -v 删除容器所挂载的数据卷。

#### run

格式为 docker-compose run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]。

在指定服务上执行一个命令。

例如：

docker-compose run ubuntu ping docker.com

将会启动一个 ubuntu 服务容器，并执行 ping docker.com 命令。

默认情况下，如果存在关联，则所有关联的服务将会自动被启动，除非这些服务已经在运行中。

该命令类似启动容器后运行指定的命令，相关卷、链接等等都将会按照配置自动创建。

两个不同点：

* 给定命令将会覆盖原有的自动运行命令；
* 不会自动创建端口，以避免冲突。

如果不希望自动启动关联的容器，可以使用 --no-deps 选项，例如

docker-compose run --no-deps web python manage.py shell

将不会启动 web 容器所关联的其它容器。

选项：

* -d 后台运行容器。
* --name NAME 为容器指定一个名字。
* --entrypoint CMD 覆盖默认的容器启动指令。
* -e KEY=VAL 设置环境变量值，可多次使用选项来设置多个环境变量。
* -u, --user="" 指定运行容器的用户名或者 uid。
* --no-deps 不自动启动关联的服务容器。
* --rm 运行命令后自动删除容器，d 模式下将忽略。
* -p, --publish=[] 映射容器端口到本地主机。
* --service-ports 配置服务端口并映射到本地主机。
* -T 不分配伪 tty，意味着依赖 tty 的指令将无法运行。

#### scale

格式为 docker-compose scale [options] [SERVICE=NUM...]。

设置指定服务运行的容器个数。

通过 service=num 的参数来设置数量。例如：

docker-compose scale web=3 db=2

将启动 3 个容器运行 web 服务，2 个容器运行 db 服务。

一般的，当指定数目多于该服务当前实际运行容器，将新创建并启动容器；反之，将停止容器。

选项：

* -t, --timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）。

#### start

格式为 docker-compose start [SERVICE...]。

启动已经存在的服务容器。

#### stop

格式为 docker-compose stop [options] [SERVICE...]。

停止已经处于运行状态的容器，但不删除它。通过 docker-compose start 可以再次启动这些容器。

选项：

* -t, --timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）。

#### top

查看各个服务容器内运行的进程。

#### unpause

格式为 docker-compose unpause [SERVICE...]。

恢复处于暂停状态中的服务。

#### up

格式为 docker-compose up [options] [SERVICE...]。

该命令十分强大，它将尝试自动完成包括构建镜像，（重新）创建服务，启动服务，并关联服务相关容器的一系列操作。

链接的服务都将会被自动启动，除非已经处于运行状态。

可以说，大部分时候都可以直接通过该命令来启动一个项目。

默认情况，docker-compose up 启动的容器都在前台，控制台将会同时打印所有容器的输出信息，可以很方便进行调试。

当通过 Ctrl-C 停止命令时，所有容器将会停止。

如果使用 docker-compose up -d，将会在后台启动并运行所有的容器。一般推荐生产环境下使用该选项。

默认情况，如果服务容器已经存在，docker-compose up 将会尝试停止容器，然后重新创建（保持使用 volumes-from 挂载的卷），以保证新启动的服务匹配 docker-compose.yml 文件的最新内容。如果用户不希望容器被停止并重新创建，可以使用 docker-compose up --no-recreate。这样将只会启动处于停止状态的容器，而忽略已经运行的服务。如果用户只想重新部署某个服务，可以使用 docker-compose up --no-deps -d <SERVICE\_NAME> 来重新创建服务并后台停止旧服务，启动新服务，并不会影响到其所依赖的服务。

选项：

* -d 在后台运行服务容器。
* --no-color 不使用颜色来区分不同的服务的控制台输出。
* --no-deps 不启动服务所链接的容器。
* --force-recreate 强制重新创建容器，不能与 --no-recreate 同时使用。
* --no-recreate 如果容器已经存在了，则不重新创建，不能与 --force-recreate 同时使用。
* --no-build 不自动构建缺失的服务镜像。
* -t, --timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）。

#### version

格式为 docker-compose version。

打印版本信息。

## docker compose 模板文件

模板文件是使用 Compose 的核心，涉及到的指令关键字也比较多。但大家不用担心，这里面大部分指令跟 docker run 相关参数的含义都是类似的。

默认的模板文件名称为 docker-compose.yml，格式为 YAML 格式。

version: "3"

services:

webapp:

image: examples/web

ports:

- "80:80"

volumes:

- "/data"

注意每个服务都必须通过 image 指令指定镜像或 build 指令（需要 Dockerfile）等来自动构建生成镜像。

如果使用 build 指令，在 Dockerfile 中设置的选项(例如：CMD, EXPOSE, VOLUME, ENV 等) 将会自动被获取，无需在 docker-compose.yml 中再次设置。

下面分别介绍各个指令的用法。

### build

指定 Dockerfile 所在文件夹的路径（可以是绝对路径，或者相对 docker-compose.yml 文件的路径）。 Compose 将会利用它自动构建这个镜像，然后使用这个镜像。

version: '3'

services:

webapp:

build: ./dir

你也可以使用 context 指令指定 Dockerfile 所在文件夹的路径。

使用 dockerfile 指令指定 Dockerfile 文件名。

使用 arg 指令指定构建镜像时的变量。

version: '3'

services:

webapp:

build:

context: ./dir

dockerfile: Dockerfile-alternate

args:

buildno: 1

使用 cache\_from 指定构建镜像的缓存

build:

context: .

cache\_from:

- alpine:latest

- corp/web\_app:3.14

### cap\_add, cap\_drop

指定容器的内核能力（capacity）分配。

例如，让容器拥有所有能力可以指定为：

cap\_add:

- ALL

去掉 NET\_ADMIN 能力可以指定为：

cap\_drop:

- NET\_ADMIN

### command

覆盖容器启动后默认执行的命令。

command: echo "hello world"

### configs

仅用于 Swarm mode

### cgroup\_parent

指定父 cgroup 组，意味着将继承该组的资源限制。

例如，创建了一个 cgroup 组名称为 cgroups\_1。

cgroup\_parent: cgroups\_1

### container\_name

指定容器名称。默认将会使用 项目名称\_服务名称\_序号 这样的格式。

container\_name: docker-web-container

注意: 指定容器名称后，该服务将无法进行扩展（scale），因为 Docker 不允许多个容器具有相同的名称。

### deploy

仅用于 Swarm mode

### devices

指定设备映射关系。

devices:

- "/dev/ttyUSB1:/dev/ttyUSB0"

### depends\_on

解决容器的依赖、启动先后的问题。以下例子中会先启动 redis db 再启动 web

version: '3'

services:

web:

build: .

depends\_on:

- db

- redis

redis:

image: redis

db:

image: postgres

注意：web 服务不会等待 redis db 「完全启动」之后才启动。

### dns

自定义 DNS 服务器。可以是一个值，也可以是一个列表。

dns: 8.8.8.8

dns:

- 8.8.8.8

- 114.114.114.114

### dns\_search

配置 DNS 搜索域。可以是一个值，也可以是一个列表。

dns\_search: example.com

dns\_search:

- domain1.example.com

- domain2.example.com

### tmpfs

挂载一个 tmpfs 文件系统到容器。

tmpfs: /run

tmpfs:

- /run

- /tmp

### env\_file

从文件中获取环境变量，可以为单独的文件路径或列表。

如果通过 docker-compose -f FILE 方式来指定 Compose 模板文件，则 env\_file 中变量的路径会基于模板文件路径。

如果有变量名称与 environment 指令冲突，则按照惯例，以后者为准。

env\_file: .env

env\_file:

- ./common.env

- ./apps/web.env

- /opt/secrets.env

环境变量文件中每一行必须符合格式，支持 # 开头的注释行。

# common.env: Set development environment

PROG\_ENV=development

### environment

设置环境变量。你可以使用数组或字典两种格式。

只给定名称的变量会自动获取运行 Compose 主机上对应变量的值，可以用来防止泄露不必要的数据。

environment:

RACK\_ENV: development

SESSION\_SECRET:

environment:

- RACK\_ENV=development

- SESSION\_SECRET

如果变量名称或者值中用到 true|false，yes|no 等表达 布尔 含义的词汇，最好放到引号里，避免 YAML 自动解析某些内容为对应的布尔语义。这些特定词汇，包括

y|Y|yes|Yes|YES|n|N|no|No|NO|true|True|TRUE|false|False|FALSE|on|On|ON|off|Off|OFF

### expose

暴露端口，但不映射到宿主机，只被连接的服务访问。

仅可以指定内部端口为参数

expose:

- "3000"

- "8000"

### external\_links

注意：不建议使用该指令。

链接到 docker-compose.yml 外部的容器，甚至并非 Compose 管理的外部容器。

external\_links:

- redis\_1

- project\_db\_1:mysql

- project\_db\_1:postgresql

### extra\_hosts

类似 Docker 中的 --add-host 参数，指定额外的 host 名称映射信息。

extra\_hosts:

- "googledns:8.8.8.8"

- "dockerhub:52.1.157.61"

会在启动后的服务容器中 /etc/hosts 文件中添加如下两条条目。

8.8.8.8 googledns

52.1.157.61 dockerhub:

### healthcheck

通过命令检查容器是否健康运行。

healthcheck:

test: ["CMD", "curl", "-f", "http://localhost"]

interval: 1m30s

timeout: 10s

retries: 3

### image

指定为镜像名称或镜像 ID。如果镜像在本地不存在，Compose 将会尝试拉取这个镜像。

image: ubuntu

image: orchardup/postgresql

image: a4bc65fd

### labels

为容器添加 Docker 元数据（metadata）信息。例如可以为容器添加辅助说明信息。

labels:

com.startupteam.description: "webapp for a startup team"

com.startupteam.department: "devops department"

com.startupteam.release: "rc3 for v1.0"

### links

注意：不推荐使用该指令。

### logging

配置日志选项。

logging:

driver: syslog

options:

syslog-address: "tcp://192.168.0.42:123"

目前支持三种日志驱动类型。

driver: "json-file"

driver: "syslog"

driver: "none"

options 配置日志驱动的相关参数。

options:

max-size: "200k"

max-file: "10"

### network\_mode

设置网络模式。使用和 docker run 的 --network 参数一样的值。

network\_mode: "bridge"

network\_mode: "host"

network\_mode: "none"

network\_mode: "service:[service name]"

network\_mode: "container:[container name/id]"

### networks

配置容器连接的网络。

version: "3"

services:

some-service:

networks:

- some-network

- other-network

networks:

some-network:

other-network:

### pid

跟主机系统共享进程命名空间。打开该选项的容器之间，以及容器和宿主机系统之间可以通过进程 ID 来相互访问和操作。

pid: "host"

### ports

暴露端口信息。

使用宿主端口：容器端口 (HOST:CONTAINER) 格式，或者仅仅指定容器的端口（宿主将会随机选择端口）都可以。

ports:

- "3000"

- "8000:8000"

- "49100:22"

- "127.0.0.1:8001:8001"

注意：当使用 HOST:CONTAINER 格式来映射端口时，如果你使用的容器端口小于 60 并且没放到引号里，可能会得到错误结果，因为 YAML 会自动解析 xx:yy 这种数字格式为 60 进制。为避免出现这种问题，建议数字串都采用引号包括起来的字符串格式。

### secrets

存储敏感数据，例如 mysql 服务密码。

version: "3.1"

services:

mysql:

image: mysql

environment:

MYSQL\_ROOT\_PASSWORD\_FILE: /run/secrets/db\_root\_password

secrets:

- db\_root\_password

- my\_other\_secret

secrets:

my\_secret:

file: ./my\_secret.txt

my\_other\_secret:

external: true

### security\_opt

指定容器模板标签（label）机制的默认属性（用户、角色、类型、级别等）。例如配置标签的用户名和角色名。

security\_opt:

- label:user:USER

- label:role:ROLE

### stop\_signal

设置另一个信号来停止容器。在默认情况下使用的是 SIGTERM 停止容器。

stop\_signal: SIGUSR1

### sysctls

配置容器内核参数。

sysctls:

net.core.somaxconn: 1024

net.ipv4.tcp\_syncookies: 0

sysctls:

- net.core.somaxconn=1024

- net.ipv4.tcp\_syncookies=0

### ulimits

指定容器的 ulimits 限制值。

例如，指定最大进程数为 65535，指定文件句柄数为 20000（软限制，应用可以随时修改，不能超过硬限制） 和 40000（系统硬限制，只能 root 用户提高）。

ulimits:

nproc: 65535

nofile:

soft: 20000

hard: 40000

### volumes

数据卷所挂载路径设置。可以设置宿主机路径 （HOST:CONTAINER） 或加上访问模式 （HOST:CONTAINER:ro）。

该指令中路径支持相对路径。

volumes:

- /var/lib/mysql

- cache/:/tmp/cache

- ~/configs:/etc/configs/:ro

### 其它指令

此外，还有包括 domainname, entrypoint, hostname, ipc, mac\_address, privileged, read\_only, shm\_size, restart, stdin\_open, tty, user, working\_dir 等指令，基本跟 docker run 中对应参数的功能一致。

指定服务容器启动后执行的入口文件。

entrypoint: /code/entrypoint.sh

指定容器中运行应用的用户名。

user: nginx

指定容器中工作目录。

working\_dir: /code

指定容器中搜索域名、主机名、mac 地址等。

domainname: your\_website.com

hostname: test

mac\_address: 08-00-27-00-0C-0A

允许容器中运行一些特权命令。

privileged: true

指定容器退出后的重启策略为始终重启。该命令对保持服务始终运行十分有效，在生产环境中推荐配置为 always 或者 unless-stopped。

restart: always

以只读模式挂载容器的 root 文件系统，意味着不能对容器内容进行修改。

read\_only: true

打开标准输入，可以接受外部输入。

stdin\_open: true

模拟一个伪终端。

tty: true

### 读取变量

Compose 模板文件支持动态读取主机的系统环境变量和当前目录下的 .env 文件中的变量。

例如，下面的 Compose 文件将从运行它的环境中读取变量 ${MONGO\_VERSION} 的值，并写入执行的指令中。

version: "3"

services:

db:

image: "mongo:${MONGO\_VERSION}"

如果执行 MONGO\_VERSION=3.2 docker-compose up 则会启动一个 mongo:3.2 镜像的容器；如果执行 MONGO\_VERSION=2.8 docker-compose up 则会启动一个 mongo:2.8 镜像的容器。

若当前目录存在 .env 文件，执行 docker-compose 命令时将从该文件中读取变量。

在当前目录新建 .env 文件并写入以下内容。

# 支持 # 号注释

MONGO\_VERSION=3.6

执行 docker-compose up 则会启动一个 mongo:3.6 镜像的容器。

## docker compose 实战 dubbo-admin

compose 可以很便捷的让 dubbo-admin 运行在一个独立的环境中。

### 创建空文件夹

假设新建一个名为 dubbo-admin 的文件夹，然后进入这个文件夹。

### 创建 docker-compose.yml 文件

docker-compose.yml 文件将开启一个 dubbo-admin 服务和一个独立的 zookeeper 实例：

version: '3.4'

services:

zk\_server:

image: zookeeper:3.4

ports:

- 2181:2181

dubbo-admin:

image: xiaochunping/dubbo-admin:1.0

links:

- zk\_server:zookeeper

depends\_on:

- zk\_server

ports:

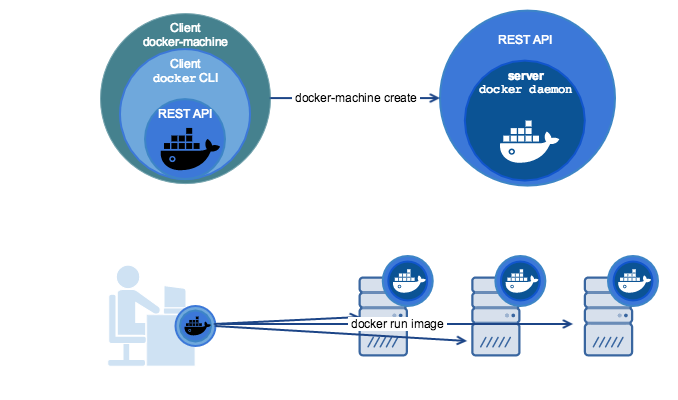
- 7001:7001

### 构建并运行项目

运行 docker-compose up -d 就会拉取镜像再创建我们所需要的镜像，然后启动 dubbo-admin 和数据库容器。 接着浏览器访问 127.0.0.1:7001 端口就能看到 dubbo-admin 安装界面了。

# docker 三剑客 machine

## Docker Machine 简介



Docker Machine 是 Docker 官方编排（Orchestration）项目之一，负责在多种平台上快速安装 Docker 环境。

Docker Machine 项目基于 Go 语言实现，目前在 [Github](https://github.com/docker/machine) 上进行维护。

本章将介绍 Docker Machine 的安装及使用。

## Docker Machine 安装

Docker Machine 可以在多种操作系统平台上安装，包括 Linux、macOS，以及 Windows。

### macOS、Windows

Docker for Mac、Docker for Windows 自带 docker-machine 二进制包，安装之后即可使用。

查看版本信息。

docker-machine -v

### Linux

在 Linux 上的也安装十分简单，从 官方 GitHub Release 处直接下载编译好的二进制文件即可。

例如，在 Linux 64 位系统上直接下载对应的二进制包。

sudo curl -L https://github.com/docker/machine/releases/download/v0.13.0/docker-machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine

sudo chmod +x /usr/local/bin/docker-machine

完成后，查看版本信息。

docker-machine -v

## Docker Machine 使用

Docker Machine 支持多种后端驱动，包括虚拟机、本地主机和云平台等。

### 创建本地主机实例

**Virtualbox 驱动**

使用 virtualbox 类型的驱动，创建一台 Docker 主机，命名为 test。

docker-machine create -d virtualbox test

你也可以在创建时加上如下参数，来配置主机或者主机上的 Docker。

--engine-opt dns=114.114.114.114 配置 Docker 的默认 DNS

--engine-registry-mirror https://registry.docker-cn.com 配置 Docker 的仓库镜像

--virtualbox-memory 2048 配置主机内存

--virtualbox-cpu-count 2 配置主机 CPU

更多参数请使用 docker-machine create --driver virtualbox --help 命令查看。

**macOS xhyve 驱动**

xhyve 驱动 GitHub: https://github.com/zchee/docker-machine-driver-xhyve

xhyve 是 macOS 上轻量化的虚拟引擎，使用其创建的 Docker Machine 较 VirtualBox 驱动创建的运行效率要高。

brew install docker-machine-driver-xhyve

docker-machine create \

-d xhyve \

# --xhyve-boot2docker-url ~/.docker/machine/cache/boot2docker.iso \

--engine-opt dns=114.114.114.114 \

--engine-registry-mirror https://registry.docker-cn.com \

--xhyve-memory-size 2048 \

--xhyve-rawdisk \

--xhyve-cpu-count 2 \

xhyve

注意：非首次创建时建议加上 --xhyve-boot2docker-url ~/.docker/machine/cache/boot2docker.iso 参数，避免每次创建时都从 GitHub 下载 ISO 镜像。

更多参数请使用 docker-machine create --driver xhyve --help 命令查看。

**Windows 10**

Windows 10 安装 Docker for Windows 之后不能再安装 VirtualBox，也就不能使用 virtualbox 驱动来创建 Docker Machine，我们可以选择使用 hyperv 驱动。

docker-machine create --driver hyperv vm

更多参数请使用 docker-machine create --driver hyperv --help 命令查看。

### 使用介绍

创建好主机之后，查看主机

docker-machine ls

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

test - virtualbox Running tcp://192.168.99.187:2376 v17.10.0-ce

创建主机成功后，可以通过 env 命令来让后续操作对象都是目标主机。

docker-machine env test

后续根据提示在命令行输入命令之后就可以操作 test 主机。

也可以通过 SSH 登录到主机。

docker-machine ssh test

docker@test:~$ docker --version

Docker version 17.10.0-ce, build f4ffd25

连接到主机之后你就可以在其上使用 Docker 了。

### 官方支持驱动

通过 -d 选项可以选择支持的驱动类型。

* amazonec2
* azure
* digitalocean
* exoscale
* generic
* google
* hyperv
* none
* openstack
* rackspace
* softlayer
* virtualbox
* vmwarevcloudair
* vmwarefusion
* vmwarevsphere

### 第三方驱动

请到 [第三方驱动列表](https://github.com/docker/docker.github.io/blob/master/machine/AVAILABLE_DRIVER_PLUGINS.md) 查看

### 操作命令

* active 查看活跃的 Docker 主机
* config 输出连接的配置信息
* create 创建一个 Docker 主机
* env 显示连接到某个主机需要的环境变量
* inspect 输出主机更多信息
* ip 获取主机地址
* kill 停止某个主机
* ls 列出所有管理的主机
* provision 重新设置一个已存在的主机
* regenerate-certs 为某个主机重新生成 TLS 认证信息
* restart 重启主机
* rm 删除某台主机
* ssh SSH 到主机上执行命令
* scp 在主机之间复制文件
* mount 挂载主机目录到本地
* start 启动一个主机
* status 查看主机状态
* stop 停止一个主机
* upgrade 更新主机 Docker 版本为最新
* url 获取主机的 URL
* version 输出 docker-machine 版本信息
* help 输出帮助信息

每个命令，又带有不同的参数，可以通过

docker-machine COMMAND --help

来查看具体的用法。

# docker 三剑客 swarm

## Docker Swarm 简介

Docker Swarm 是 Docker 官方三剑客项目之一，提供 Docker 容器集群服务，是 Docker 官方对容器云生态进行支持的核心方案。

使用它，用户可以将多个 Docker 主机封装为单个大型的虚拟 Docker 主机，快速打造一套容器云平台。

注意：Docker 1.12.0+ Swarm mode 已经内嵌入 Docker 引擎，成为了 docker 子命令 docker swarm，绝大多数用户已经开始使用 Swarm mode，Docker 引擎 API 已经删除 Docker Swarm。为避免大家混淆旧的 Docker Swarm 与新的 Swarm mode，旧的 Docker Swarm 内容已经删除，请查看 Swarm mode 一节。

## Docker Swarm mode

Docker 1.12 Swarm mode 已经内嵌入 Docker 引擎，成为了 docker 子命令 docker swarm。请注意与旧的 Docker Swarm 区分开来。

Swarm mode 内置 kv 存储功能，提供了众多的新特性，比如：具有容错能力的去中心化设计、内置服务发现、负载均衡、路由网格、动态伸缩、滚动更新、安全传输等。使得 Docker 原生的 Swarm 集群具备与 Mesos、Kubernetes 竞争的实力。

## Swarm mode 基本概念

Swarm 是使用 [SwarmKit](https://github.com/docker/swarmkit/) 构建的 Docker 引擎内置（原生）的集群管理和编排工具。

使用 Swarm 集群之前需要了解以下几个概念。

### 节点

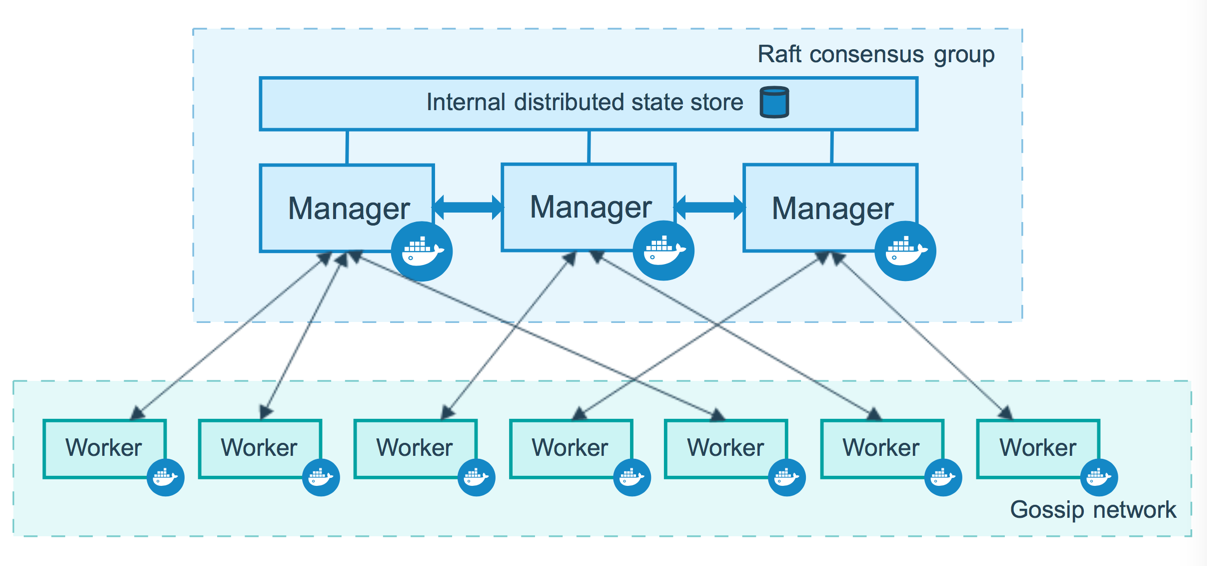
运行 Docker 的主机可以主动初始化一个 Swarm 集群或者加入一个已存在的 Swarm 集群，这样这个运行 Docker 的主机就成为一个 Swarm 集群的节点 (node) 。

节点分为管理 (manager) 节点和工作 (worker) 节点。

管理节点用于 Swarm 集群的管理，docker swarm 命令基本只能在管理节点执行（节点退出集群命令 docker swarm leave 可以在工作节点执行）。一个 Swarm 集群可以有多个管理节点，但只有一个管理节点可以成为 leader，leader 通过 raft 协议实现。

工作节点是任务执行节点，管理节点将服务 (service) 下发至工作节点执行。管理节点默认也作为工作节点。你也可以通过配置让服务只运行在管理节点。

来自 Docker 官网的这张图片形象的展示了集群中管理节点与工作节点的关系。



### 服务和任务

任务 （Task）是 Swarm 中的最小的调度单位，目前来说就是一个单一的容器。

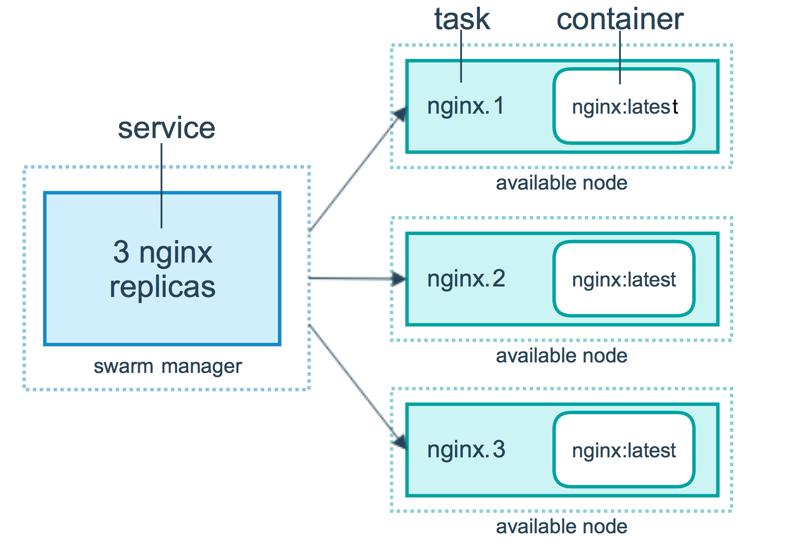
服务 （Services） 是指一组任务的集合，服务定义了任务的属性。服务有两种模式：

replicated services 按照一定规则在各个工作节点上运行指定个数的任务。

global services 每个工作节点上运行一个任务

两种模式通过 docker service create 的 --mode 参数指定。

来自 Docker 官网的这张图片形象的展示了容器、任务、服务的关系。



## 创建 Swarm 集群

阅读 基本概念 一节我们知道 Swarm 集群由 管理节点 和 工作节点 组成。本节我们来创建一个包含一个管理节点和两个工作节点的最小 Swarm 集群。

### 初始化集群

在 Docker Machine 一节中我们了解到 Docker Machine 可以在数秒内创建一个虚拟的 Docker 主机，下面我们使用它来创建三个 Docker 主机，并加入到集群中。

我们首先创建一个 Docker 主机作为管理节点。

docker-machine create -d virtualbox manager

我们使用 docker swarm init 在管理节点初始化一个 Swarm 集群。

docker-machine ssh manager

docker@manager:~$ docker swarm init --advertise-addr 192.168.99.100

Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

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

docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

192.168.99.100:2377

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

如果你的 Docker 主机有多个网卡，拥有多个 IP，必须使用 --advertise-addr 指定 IP。

执行 docker swarm init 命令的节点自动成为管理节点。

### 增加工作节点

上一步我们初始化了一个 Swarm 集群，拥有了一个管理节点，下面我们继续创建两个 Docker 主机作为工作节点，并加入到集群中。

$ docker-machine create -d virtualbox worker1

$ docker-machine ssh worker1

docker@worker1:~$ docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

192.168.99.100:2377

This node joined a swarm as a worker.

$ docker-machine create -d virtualbox worker2

$ docker-machine ssh worker2

docker@worker1:~$ docker swarm join \

--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \

192.168.99.100:2377

This node joined a swarm as a worker.

注意：一些细心的读者可能通过 docker-machine create --help 查看到 --swarm\* 等一系列参数。该参数是用于旧的 Docker Swarm,与本章所讲的 Swarm mode 没有关系。

### 查看集群

经过上边的两步，我们已经拥有了一个最小的 Swarm 集群，包含一个管理节点和两个工作节点。

在管理节点使用 docker node ls 查看集群。

docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

03g1y59jwfg7cf99w4lt0f662 worker2 Ready Active

9j68exjopxe7wfl6yuxml7a7j worker1 Ready Active

dxn1zf6l61qsb1josjja83ngz \* manager Ready Active Leader

## Swarm mode 部署服务

我们使用 docker service 命令来管理 Swarm 集群中的服务，该命令只能在管理节点运行。

### 新建服务

现在我们在上一节创建的 Swarm 集群中运行一个名为 nginx 服务。

docker service create --replicas 3 -p 80:80 --name nginx nginx:1.13.7-alpine

现在我们使用浏览器，输入任意节点 IP ,即可看到 nginx 默认页面。

### 查看服务

使用 docker service ls 来查看当前 Swarm 集群运行的服务。

docker service ls

ID NAME MODE REPLICAS IMAGE PORTS

kc57xffvhul5 nginx replicated 3/3 nginx:1.13.7-alpine \*:80->80/tcp

使用 docker service ps 来查看某个服务的详情。

docker service ps nginx

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

pjfzd39buzlt nginx.1 nginx:1.13.7-alpine swarm2 Running Running about a minute ago

hy9eeivdxlaa nginx.2 nginx:1.13.7-alpine swarm1 Running Running about a minute ago

36wmpiv7gmfo nginx.3 nginx:1.13.7-alpine swarm3 Running Running about a minute ago

使用 docker service logs 来查看某个服务的日志。

docker service logs nginx

nginx.3.36wmpiv7gmfo@swarm3 | 10.255.0.4 - - [25/Nov/2017:02:10:30 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:58.0) Gecko/20100101 Firefox/58.0" "-"

nginx.3.36wmpiv7gmfo@swarm3 | 10.255.0.4 - - [25/Nov/2017:02:10:30 +0000] "GET /favicon.ico HTTP/1.1" 404 169 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:58.0) Gecko/20100101 Firefox/58.0" "-"

nginx.3.36wmpiv7gmfo@swarm3 | 2017/11/25 02:10:30 [error] 5#5: \*1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 10.255.0.4, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.99.102"

nginx.1.pjfzd39buzlt@swarm2 | 10.255.0.2 - - [25/Nov/2017:02:10:26 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:58.0) Gecko/20100101 Firefox/58.0" "-"

nginx.1.pjfzd39buzlt@swarm2 | 10.255.0.2 - - [25/Nov/2017:02:10:27 +0000] "GET /favicon.ico HTTP/1.1" 404 169 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:58.0) Gecko/20100101 Firefox/58.0" "-"

nginx.1.pjfzd39buzlt@swarm2 | 2017/11/25 02:10:27 [error] 5#5: \*1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 10.255.0.2, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.99.101"

### 删除服务

使用 docker service rm 来从 Swarm 集群移除某个服务。

docker service rm nginx

## 使用 compose 文件

正如之前使用 docker-compose.yml 来一次配置、启动多个容器，在 Swarm 集群中也可以使用 compose 文件 （docker-compose.yml） 来配置、启动多个服务。

上一节中，我们使用 docker service create 一次只能部署一个服务，使用 docker-compose.yml 我们可以一次启动多个关联的服务。

我们以在 Swarm 集群中部署 WordPress 为例进行说明。

version: "3"

services:

wordpress:

image: wordpress

ports:

- 80:80

networks:

- overlay

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD: wordpress

deploy:

mode: replicated

replicas: 3

db:

image: mysql

networks:

- overlay

volumes:

- db-data:/var/lib/mysql

environment:

MYSQL\_ROOT\_PASSWORD: somewordpress

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD: wordpress

deploy:

placement:

constraints: [node.role == manager]

visualizer:

image: dockersamples/visualizer:stable

ports:

- "8080:8080"

stop\_grace\_period: 1m30s

volumes:

- "/var/run/docker.sock:/var/run/docker.sock"

deploy:

placement:

constraints: [node.role == manager]

volumes:

db-data:

networks:

overlay:

在 Swarm 集群管理节点新建该文件，其中的 visualizer 服务提供一个可视化页面，我们可以从浏览器中很直观的查看集群中各个服务的运行节点。

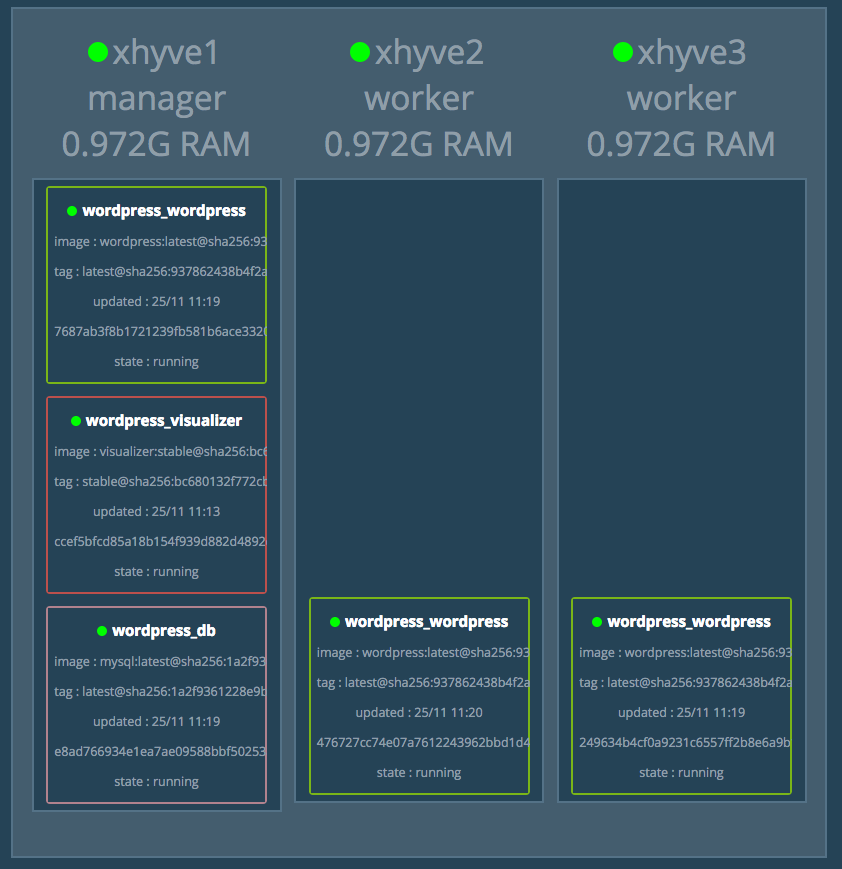
在 Swarm 集群中使用 docker-compose.yml 我们用 docker stack 命令，下面我们对该命令进行详细讲解。

### 部署服务

部署服务使用 docker stack deploy，其中 -c 参数指定 compose 文件名。

docker stack deploy -c docker-compose.yml wordpress

现在我们打开浏览器输入 任一节点IP:8080 即可看到各节点运行状态。如下图所示：



在浏览器新的标签页输入 任一节点IP 即可看到 WordPress 安装界面，安装完成之后，输入 任一节点IP 即可看到 WordPress 页面。

### 查看服务

docker stack ls

NAME SERVICES

wordpress 3

### 移除服务

要移除服务，使用 docker stack down

docker stack down wordpress

Removing service wordpress\_db

Removing service wordpress\_visualizer

Removing service wordpress\_wordpress

Removing network wordpress\_overlay

Removing network wordpress\_default

该命令不会移除服务所使用的 数据卷，如果你想移除数据卷请使用 docker volume rm

## Swarm mode 管理敏感数据

在动态的、大规模的分布式集群上，管理和分发 密码、证书 等敏感信息是极其重要的工作。传统的密钥分发方式（如密钥放入镜像中，设置环境变量，volume 动态挂载等）都存在着潜在的巨大的安全风险。

Docker 目前已经提供了 secrets 管理功能，用户可以在 Swarm 集群中安全地管理密码、密钥证书等敏感数据，并允许在多个 Docker 容器实例之间共享访问指定的敏感数据。

注意： secret 也可以在 Docker Compose 中使用。

我们可以用 docker secret 命令来管理敏感信息。接下来我们在上面章节中创建好的 Swarm 集群中介绍该命令的使用。

这里我们以在 Swarm 集群中部署 mysql 和 wordpress 服务为例。

### 创建 secret

我们使用 docker secret create 命令以管道符的形式创建 secret

openssl rand -base64 20 | docker secret create mysql\_password -

openssl rand -base64 20 | docker secret create mysql\_root\_password -

### 查看 secret

使用 docker secret ls 命令来查看 secret

docker secret ls-

ID NAME CREATED UPDATED

l1vinzevzhj4goakjap5ya409 mysql\_password 41 seconds ago 41 seconds ago

yvsczlx9votfw3l0nz5rlidig mysql\_root\_password 12 seconds ago 12 seconds ago

### 创建 MySQL 服务

创建服务相关命令已经在前边章节进行了介绍，这里直接列出命令。

docker network create -d overlay mysql\_private

docker service create \

--name mysql \

--replicas 1 \

--network mysql\_private \

--mount type=volume,source=mydata,destination=/var/lib/mysql \

--secret source=mysql\_root\_password,target=mysql\_root\_password \

--secret source=mysql\_password,target=mysql\_password \

-e MYSQL\_ROOT\_PASSWORD\_FILE="/run/secrets/mysql\_root\_password" \

-e MYSQL\_PASSWORD\_FILE="/run/secrets/mysql\_password" \

-e MYSQL\_USER="wordpress" \

-e MYSQL\_DATABASE="wordpress" \

mysql:latest

如果你没有在 target 中显式的指定路径时，secret 默认通过 tmpfs 文件系统挂载到容器的 /run/secrets 目录中。

docker service create \

--name wordpress \

--replicas 1 \

--network mysql\_private \

--publish target=30000,port=80 \

--mount type=volume,source=wpdata,destination=/var/www/html \

--secret source=mysql\_password,target=wp\_db\_password,mode=0400 \

-e WORDPRESS\_DB\_USER="wordpress" \

-e WORDPRESS\_DB\_PASSWORD\_FILE="/run/secrets/wp\_db\_password" \

-e WORDPRESS\_DB\_HOST="mysql:3306" \

-e WORDPRESS\_DB\_NAME="wordpress" \

wordpress:latest

查看服务

docker service ls

ID NAME MODE REPLICAS IMAGE

wvnh0siktqr3 mysql replicated 1/1 mysql:latest

nzt5xzae4n62 wordpress replicated 1/1 wordpress:latest

现在浏览器访问 IP:30000，即可开始 WordPress 的安装与使用。

通过以上方法，我们没有像以前通过设置环境变量来设置 MySQL 密码， 而是采用 docker secret 来设置密码，防范了密码泄露的风险。

## Swarm mode 管理配置信息

在动态的、大规模的分布式集群上，管理和分发配置文件也是很重要的工作。传统的配置文件分发方式（如配置文件放入镜像中，设置环境变量，volume 动态挂载等）都降低了镜像的通用性。

在 Docker 17.06 以上版本中，Docker 新增了 docker config 子命令来管理集群中的配置信息，以后你无需将配置文件放入镜像或挂载到容器中就可实现对服务的配置。

注意：config 仅能在 Swarm 集群中使用。

这里我们以在 Swarm 集群中部署 redis 服务为例。

### 创建 config

新建 redis.conf 文件

port 6380

此项配置 Redis 监听 6380 端口

我们使用 docker config create 命令创建 config

docker config create redis.conf redis.conf

### 查看 config

使用 docker config ls 命令来查看 config

docker config ls

ID NAME CREATED UPDATED

yod8fx8iiqtoo84jgwadp86yk redis.conf 4 seconds ago 4 seconds ago

### 创建 redis 服务

docker service create \

--name redis \

# --config source=redis.conf,target=/etc/redis.conf \

--config redis.conf \

-p 6379:6380 \

redis:latest \

redis-server /redis.conf

如果你没有在 target 中显式的指定路径时，默认的 redis.conf 以 tmpfs 文件系统挂载到容器的 /config.conf。

经过测试，redis 可以正常使用。

以前我们通过监听主机目录来配置 Redis，就需要在集群的每个节点放置该文件，如果采用 docker config 来管理服务的配置信息，我们只需在集群中的管理节点创建 config，当部署服务时，集群会自动的将配置文件分发到运行服务的各个节点中，大大降低了配置信息的管理和分发难度。

# docker 安全

评估 Docker 的安全性时，主要考虑三个方面:

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

## 内核命名空间

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

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

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

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

那么，内核中实现命名空间和私有网络的代码是否足够成熟？

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

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

## 控制组

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

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

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

控制组机制始于 2006 年，内核从 2.6.24 版本开始被引入。

## 服务端防护

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

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

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

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

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

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

终极目标是改进 2 个重要的安全特性：

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

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

## 内核能力机制

能力机制（Capability）是 Linux 内核一个强大的特性，可以提供细粒度的权限访问控制。 Linux 内核自 2.2 版本起就支持能力机制，它将权限划分为更加细粒度的操作能力，既可以作用在进程上，也可以作用在文件上。

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

默认情况下，Docker 启动的容器被严格限制只允许使用内核的一部分能力。

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

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

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

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

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

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

## 其它安全特性

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

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

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

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

## 安全总结

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

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

# docker 底层实现

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

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

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

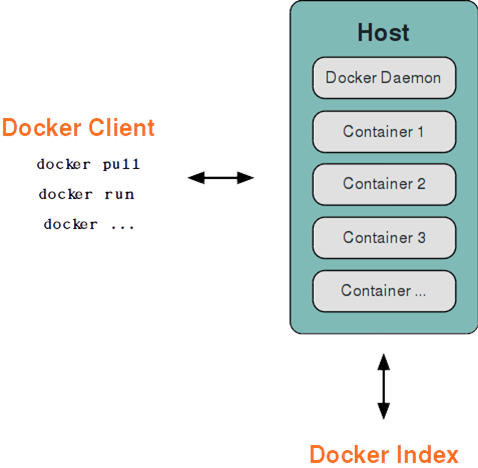
随着 Linux 系统对于命名空间功能的完善实现，程序员已经可以实现上面的所有需求，让某些进程在彼此隔离的命名空间中运行。大家虽然都共用一个内核和某些运行时环境（例如一些系统命令和系统库），但是彼此却看不到，都以为系统中只有自己的存在。这种机制就是容器（Container），利用命名空间来做权限的隔离控制，利用 cgroups 来做资源分配。

## 基本架构

Docker 采用了 C/S 架构，包括客户端和服务端。Docker 守护进程 （Daemon）作为服务端接受来自客户端的请求，并处理这些请求（创建、运行、分发容器）。

客户端和服务端既可以运行在一个机器上，也可通过 socket 或者 RESTful API 来进行通信。

Docker 基本架构



Docker 守护进程一般在宿主主机后台运行，等待接收来自客户端的消息。

Docker 客户端则为用户提供一系列可执行命令，用户用这些命令实现跟 Docker 守护进程交互。

## 命名空间

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

### pid 命名空间

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

### net 命名空间

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

### ipc 命名空间

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

### mnt 命名空间

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

### uts 命名空间

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

### user 命名空间

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

## 控制组

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

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

控制组可以提供对容器的内存、CPU、磁盘 IO 等资源的限制和审计管理。

## 联合文件系统

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

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

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

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

Docker 目前支持的联合文件系统包括 OverlayFS, AUFS, Btrfs, VFS, ZFS 和 Device Mapper。

各 Linux 发行版 Docker 推荐使用的存储驱动如下表。

|  |  |
| --- | --- |
| **Linux 发行版** | **Docker 推荐使用的存储驱动** |
| Docker CE on Ubuntu | aufs, devicemapper, overlay2 (Ubuntu 14.04.4 +, 16.04 +), 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 +), overlay (实验性支持), vfs |

在可能的情况下，推荐使用 overlay2 存储驱动，overlay2 是目前 Docker 默认的存储驱动，以前则是 aufs。你可以通过配置来使用以上提到的其他类型的存储驱动。

## 容器格式

最初，Docker 采用了 LXC 中的容器格式。从 0.7 版本以后开始去除 LXC，转而使用自行开发的 [libcontainer](https://github.com/docker/libcontainer)，从 1.11 开始，则进一步演进为使用 [runC](https://github.com/opencontainers/runc) 和 [containerd](https://containerd.tools/)。

对更多容器格式的支持，还在进一步的发展中。

## 网络

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

### 基本原理

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

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

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

### 创建网络参数

Docker 创建一个容器的时候，会执行如下操作：

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

完成这些之后，容器就可以使用 eth0 虚拟网卡来连接其他容器和其他网络。

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

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

### 网络配置细节

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

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

docker run -i -t --rm --net=none base /bin/bash

root@63f36fc01b5f:/#

在本地主机查找容器的进程 id，并为它创建网络命名空间。

docker inspect -f '{{.State.Pid}}' 63f36fc01b5f

2778

pid=2778

sudo mkdir -p /var/run/netns

sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid

检查桥接网卡的 IP 和子网掩码信息。

ip addr show docker0

21: docker0: ...

inet 172.17.42.1/16 scope global docker0

...

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

sudo ip link add A type veth peer name B

sudo brctl addif docker0 A

sudo ip link set A up

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

sudo ip link set B netns $pid

sudo ip netns exec $pid ip link set dev B name eth0

sudo ip netns exec $pid ip link set eth0 up

sudo ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0

sudo ip netns exec $pid ip route add default via 172.17.42.1

以上，就是 Docker 配置网络的具体过程。

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

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

# 附录

## Docker 命令查询

### 基本语法

Docker 命令有两大类，客户端命令和服务端命令。前者是主要的操作接口，后者用来启动 Docker Daemon。

客户端命令：基本命令格式为 docker [OPTIONS] COMMAND [arg...]；

服务端命令：基本命令格式为 dockerd [OPTIONS]。

可以通过 man docker 或 docker help 来查看这些命令。

### 客户端命令选项

--config=""：指定客户端配置文件，默认为 `/.docker`；

-D=true|false：是否使用 debug 模式。默认不开启；

-H, --host=[]：指定命令对应 Docker 守护进程的监听接口，可以为 unix 套接字（unix:///path/to/socket），文件句柄（fd://socketfd）或 tcp 套接字（tcp://[host[:port]]），默认为 unix:///var/run/docker.sock；

-l, --log-level="debug|info|warn|error|fatal"：指定日志输出级别；

--tls=true|false：是否对 Docker 守护进程启用 TLS 安全机制，默认为否；

--tlscacert= /.docker/ca.pem：TLS CA 签名的可信证书文件路径；

--tlscert= /.docker/cert.pem：TLS 可信证书文件路径；

--tlscert= /.docker/key.pem：TLS 密钥文件路径；

--tlsverify=true|false：启用 TLS 校验，默认为否。

### dockerd 命令选项

--api-cors-header=""：CORS 头部域，默认不允许 CORS，要允许任意的跨域访问，可以指定为 “\*”；

--authorization-plugin=""：载入认证的插件；

-b=""：将容器挂载到一个已存在的网桥上。指定为 'none' 时则禁用容器的网络，与 --bip 选项互斥；

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

--cgroup-parent=""：指定 cgroup 的父组，默认 fs cgroup 驱动为 `/docker`，systemd cgroup 驱动为 `system.slice`；

--cluster-store=""：构成集群（如 Swarm）时，集群键值数据库服务地址；

--cluster-advertise=""：构成集群时，自身的被访问地址，可以为 `host:port` 或 `interface:port`；

--cluster-store-opt=""：构成集群时，键值数据库的配置选项；

--config-file="/etc/docker/daemon.json"：daemon 配置文件路径；

--containerd=""：containerd 文件的路径；

-D, --debug=true|false：是否使用 Debug 模式。缺省为 false；

--default-gateway=""：容器的 IPv4 网关地址，必须在网桥的子网段内；

--default-gateway-v6=""：容器的 IPv6 网关地址；

--default-ulimit=[]：默认的 ulimit 值；

--disable-legacy-registry=true|false：是否允许访问旧版本的镜像仓库服务器；

--dns=""：指定容器使用的 DNS 服务器地址；

--dns-opt=""：DNS 选项；

--dns-search=[]：DNS 搜索域；

--exec-opt=[]：运行时的执行选项；

--exec-root=""：容器执行状态文件的根路径，默认为 `/var/run/docker`；

--fixed-cidr=""：限定分配 IPv4 地址范围；

--fixed-cidr-v6=""：限定分配 IPv6 地址范围；

-G, --group=""：分配给 unix 套接字的组，默认为 `docker`；

-g, --graph=""：Docker 运行时的根路径，默认为 `/var/lib/docker`；

-H, --host=[]：指定命令对应 Docker daemon 的监听接口，可以为 unix 套接字（unix:///path/to/socket），文件句柄（fd://socketfd）或 tcp 套接字（tcp://[host[:port]]），默认为 unix:///var/run/docker.sock；

--icc=true|false：是否启用容器间以及跟 daemon 所在主机的通信。默认为 true。

--insecure-registry=[]：允许访问给定的非安全仓库服务；

--ip=""：绑定容器端口时候的默认 IP 地址。缺省为 0.0.0.0；

--ip-forward=true|false：是否检查启动在 Docker 主机上的启用 IP 转发服务，默认开启。注意关闭该选项将不对系统转发能力进行任何检查修改；

--ip-masq=true|false：是否进行地址伪装，用于容器访问外部网络，默认开启；

--iptables=true|false：是否允许 Docker 添加 iptables 规则。缺省为 true；

--ipv6=true|false：是否启用 IPv6 支持，默认关闭；

-l, --log-level="debug|info|warn|error|fatal"：指定日志输出级别；

--label="[]"：添加指定的键值对标注；

--log-driver="json-file|syslog|journald|gelf|fluentd|awslogs|splunk|etwlogs|gcplogs|none"：指定日志后端驱动，默认为 json-file；

--log-opt=[]：日志后端的选项；

--mtu=VALUE：指定容器网络的 mtu；

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

--raw-logs：输出原始，未加色彩的日志信息；

--registry-mirror=<scheme>://<host>：指定 `docker pull` 时使用的注册服务器镜像地址；

-s, --storage-driver=""：指定使用给定的存储后端；

--selinux-enabled=true|false：是否启用 SELinux 支持。缺省值为 false。SELinux 目前尚不支持 overlay 存储驱动；

--storage-opt=[]：驱动后端选项；

--tls=true|false：是否对 Docker daemon 启用 TLS 安全机制，默认为否；

--tlscacert= /.docker/ca.pem：TLS CA 签名的可信证书文件路径；

--tlscert= /.docker/cert.pem：TLS 可信证书文件路径；

--tlscert= /.docker/key.pem：TLS 密钥文件路径；

--tlsverify=true|false：启用 TLS 校验，默认为否；

--userland-proxy=true|false：是否使用用户态代理来实现容器间和出容器的回环通信，默认为 true；

--userns-remap=default|uid:gid|user:group|user|uid：指定容器的用户命名空间，默认是创建新的 UID 和 GID 映射到容器内进程。

### 客户端命令

可以通过 docker COMMAND --help 来查看这些命令的具体用法。

attach：依附到一个正在运行的容器中；

build：从一个 Dockerfile 创建一个镜像；

commit：从一个容器的修改中创建一个新的镜像；

cp：在容器和本地宿主系统之间复制文件中；

create：创建一个新容器，但并不运行它；

diff：检查一个容器内文件系统的修改，包括修改和增加；

events：从服务端获取实时的事件；

exec：在运行的容器内执行命令；

export：导出容器内容为一个 tar 包；

history：显示一个镜像的历史信息；

images：列出存在的镜像；

import：导入一个文件（典型为 tar 包）路径或目录来创建一个本地镜像；

info：显示一些相关的系统信息；

inspect：显示一个容器的具体配置信息；

kill：关闭一个运行中的容器 (包括进程和所有相关资源)；

load：从一个 tar 包中加载一个镜像；

login：注册或登录到一个 Docker 的仓库服务器；

logout：从 Docker 的仓库服务器登出；

logs：获取容器的 log 信息；

network：管理 Docker 的网络，包括查看、创建、删除、挂载、卸载等；

node：管理 swarm 集群中的节点，包括查看、更新、删除、提升/取消管理节点等；

pause：暂停一个容器中的所有进程；

port：查找一个 nat 到一个私有网口的公共口；

ps：列出主机上的容器；

pull：从一个Docker的仓库服务器下拉一个镜像或仓库；

push：将一个镜像或者仓库推送到一个 Docker 的注册服务器；

rename：重命名一个容器；

restart：重启一个运行中的容器；

rm：删除给定的若干个容器；

rmi：删除给定的若干个镜像；

run：创建一个新容器，并在其中运行给定命令；

save：保存一个镜像为 tar 包文件；

search：在 Docker index 中搜索一个镜像；

service：管理 Docker 所启动的应用服务，包括创建、更新、删除等；

start：启动一个容器；

stats：输出（一个或多个）容器的资源使用统计信息；

stop：终止一个运行中的容器；

swarm：管理 Docker swarm 集群，包括创建、加入、退出、更新等；

tag：为一个镜像打标签；

top：查看一个容器中的正在运行的进程信息；

unpause：将一个容器内所有的进程从暂停状态中恢复；

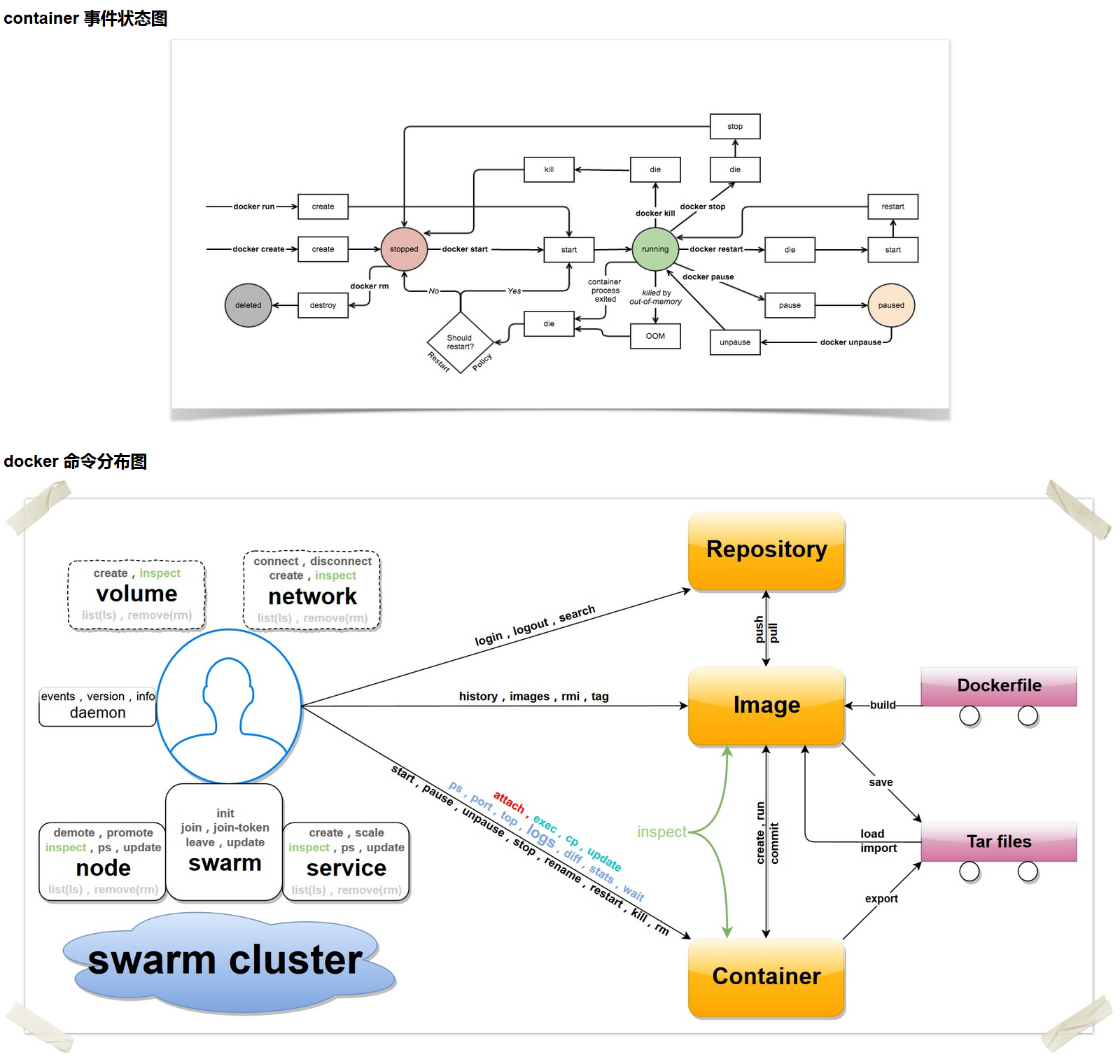
update：更新指定的若干容器的配置信息；

version：输出 Docker 的版本信息；

volume：管理 Docker volume，包括查看、创建、删除等；

wait：阻塞直到一个容器终止，然后输出它的退出符。

### 一张图总结 Docker 的命令



## Docker 资源链接

### 官方网站

* Docker 官方主页：https://www.docker.com
* Docker 官方博客：https://blog.docker.com/
* Docker 官方文档：https://docs.docker.com/
* Docker Store：https://store.docker.com
* Docker Cloud：https://cloud.docker.com
* Docker Hub：https://hub.docker.com
* Docker 的源代码仓库：https://github.com/moby/moby
* Docker 发布版本历史：https://docs.docker.com/release-notes/
* Docker 常见问题：https://docs.docker.com/engine/faq/
* Docker 远端应用 API：https://docs.docker.com/develop/sdk/

### 实践参考

* Dockerfile 参考：https://docs.docker.com/engine/reference/builder/
* Dockerfile 最佳实践：
* https://docs.docker.com/engine/userguide/eng-image/dockerfile\_best-practices/

### 技术交流

* Docker 邮件列表： https://groups.google.com/forum/#!forum/docker-user
* Docker 的 IRC 频道：https://chat.freenode.net#docker
* Docker 的 Twitter 主页：https://twitter.com/docker

### 其它

* Docker 的 StackOverflow 问答主页：https://stackoverflow.com/search?q=docker