

# Docker

---

Docker

容器技术

简单介绍

容器的特性

容器虚拟化的核心技术

容器技术的发展史

容器的组成

容器核心技术介绍

Cgroup

Cgroup 是什么

Cgroup 的接口

Namespace

Namespace 介绍

Docker 容器和虚拟机的区别

1. 实现原理技术不同
2. 使用资源方面不同
3. 应用场景不同

Docker 容器和虚拟机合作

什么是Docker

Docker 的架构

Docker 功能和组件

Docker 客户端

Docker daemon

Docker 容器

Docker 镜像

Registry

## Docker 安装和启动服务

安装

启动服务

更多配置

## 使用国内源进行加速

## Docker 使用

获取命令行帮助信息

镜像管理

搜索镜像

获取镜像

查看本地的镜像

查看一个镜像的制作历程

备份本地仓库的镜像

删除镜像

容器管理

运行一个容器

查看正在运行中的容器

查看本地所有的容器

查看最近一次启动的容器

获取正在运行的容器的容器 ID

获取所有容器的容器 ID

查看容器的元数据

重命名容器名称

重新启动一个处于停止状态的容器

重启一个容器

停止一个容器

关闭和删除容器

在外部执行运行状态中容器的命令

进入到一个正在运行中的容器

监控容器的运行

在宿主机和容器之间交换文件

- 查看容器内发生改变的文件
- 实时输出Docker服务端的事件
- 创建一个容器但不启用
- 把容器制作作为一个镜像

## 镜像详解

- 创建自己的镜像
  - 通过容器创建
  - 通过 Dockerfile 创建
- 构建镜像的上下文(context)

## Dockerfile 详解

- FROM 指令
  - 制作自己的 Hello world
  - 关于 Alpine
- LABEL 指令
- ENV 指令
- RUN 指令
  - 执行命令的两种方式
- CMD 指令
- ENTRYPOINT 指令
- WORKDIR 指令
- COPY 指令
- ADD 指令
- USER 指令
- HEALTHCHECK 健康检查指令
- ONBUILD 指令

## 基础应用场景

### 高级应用场景

更多参考官方 Docker Demo 和官网

## 镜像发布

- 直接把镜像推到 docker hub 上
- 把 Dockerfile 推到 docker hub (推荐)

针对 Docker hub 上的多个仓库映射到 Git hub 上被关联仓库下的不同目录

管理 Docker 中的数据

三者的相同点和区别

相同之处

不同之处

`-v` 还是 `--mount`

关于使用数据卷和挂载主机目录的提示

数据卷 (volume) 详解

volume 的使用场景

volume 基本使用

bind mount 详解

`bind mount` 的使用场景

挂载一个主机目录作为数据卷

查看数据卷的具体信息

Docker 网络

配置 s 网桥

外部访问容器

查看端口映射配置信息

同一台主机上的容器互联和隔离

容器默认支持互联

在一个主机上隔离不同的 容器

配置 DNS

部署私用仓库

获取

以容器方式运行

在私有仓库上传、搜索、下载镜像

配置可以访问远程私有仓库的 `Docker` 主机

---

# 容器技术

## 简单介绍

容器技术又称为容器虚拟化

首先是一种虚拟化技术

虚拟化技术包括硬件虚拟化 半虚拟化 操作系统虚拟化

容器虚拟化就是操作系统虚拟化，是属于轻量级的虚拟化

容器虚拟化技术是已经集成到 Linux 内核中的

## 容器的特性

容器首先是一个相对独立的运行环境，并且在一个容器环境中，应该最小化对外界的影响，比如不能在容器中把宿主机上的资源全部消耗完，这就是资源控制。

## 容器虚拟化的核心技术

一般来说容器技术主要包含 Namespace 和 Cgroup 这两个内核特性

- Namespace 又称为命名空间（或名字空间），主要做访问隔离。其原理是针对一类资源进行抽象，并将其封装在一起提供给一个容器使用，对于这类资源，每个容器都有自己的抽象，而它们之间是不可见的，所以可以做到访问隔离。

- Cgroup 是 control group 的简称，又称为控制组，主要做资源控制。其原理是将一组进程放在一个控制组里，通过给这个控制组分配指定的可用资源，达到控制这一组进程可用资源的目的。

上述的两个核心技术两者并不存在依赖性，但是可以用组合的方式实现容器技术。

比如在一个 Namespace 中的进程恰好又在一个 Cgroup 中，那么这些进程就同时有了访问隔离和资源控制。这恰恰是容器的特性。

## 容器技术的发展史

### 1982年

在普通的目录结构中创建一个完整的子目录结构，可以称为抽象化目录结构。

通过 chroot 技术实现，就是把用户的文件系统根目录切换到某个指定的目录下。

缺点：

只是实现了视图上的虚拟化

用户实际上可以逃离设定的根目录

### 2000年

linux 内核版本 2.3.14 中引入了 pivot\_root 技术，它有效的避免了 chroot 带来的安全性问题。

目前的容器技术都是使用了 pivot\_root 技术来做根文件系统的切换。

缺点：

仅仅是对文件系统隔离的增强

市场上也出现了一些商用的容器技术，如 SWset(现在的 Odin) 开发的 Virtuozzo

## 2005年

Odin公司在 Virtuozzo 的基础上发布了 OpenVZ 技术，同时开始推动 OpenVZ 中的核心容器技术进入 Linux 内核主线，而此时 IBM 公司也在推动类似的技术，最后在社区的合作下，形成了目前大家看到的 Namespace 和 Cgroup

## 2013年

随着容器技术在内核主线中不断成熟和完善，Docker 诞生了，并且由于它的诞生，容器技术才真正引起了全世界技术公司和开发人员的关注。

# 容器的组成

对于 Linux 容器的最小组成：

**容器 = cgroup + namespace + rootfs + 容器引擎(用户态工具)**

- Cgroup 资源控制, 消耗的限制
- Namespace 访问隔离, 空间的限制
- rootfs 文件系统隔离
- 容器引擎 生命周期控制

## 容器核心技术介绍

### Cgroup

#### Cgroup 是什么

是 control group 的简写, 是属于 Linux 内核提供的一个特性, 用于限制和隔离一组进程对系统资源的使用。

这些资源包括 CPU、内存、block I/O 和网络带宽。

Cgroup 从 2.6.24 开始进入内核主线。

从实现的角度来看, Cgroup 实现了一个通用的进程分组的框架, 不同的资源由各个子系统管理。

- devices 设备权限控制
- cpuset 分配指定的 CPU 和内存节点。
- cpu 控制 CPU 占用率
- cpuacct 统计 CPU 使用情况
- memory 限制内存的使用上限

- freezer   冻结（暂停）Cgroup 中的进程
- net\_cls   配合 traffic controller 限制网络带宽
- net\_prio   设置进程的网络流量优先级
- huge\_tlb   限制 HugeTLB 使用
- perf\_event   允许 Perf 工具基于 Cgroup 分组做性能检测

## Cgroup 的接口

原生接口通过 cgroupfs 提供，是一种虚拟文件系统。

要想使用必须先挂载 cgroupfs 文件系统，命令如下：

```
# mount -t cgroup -o cpuset cpuset  
/sys/fs/cgroup/cpuset
```

注意：这个动作一般已经在启动时由系统自动完成了。

## 查看默认的 cgroupfs

```
[root@localhost ~]# ls -l  
/sys/fs/cgroup/cpuset
```

# Namespace

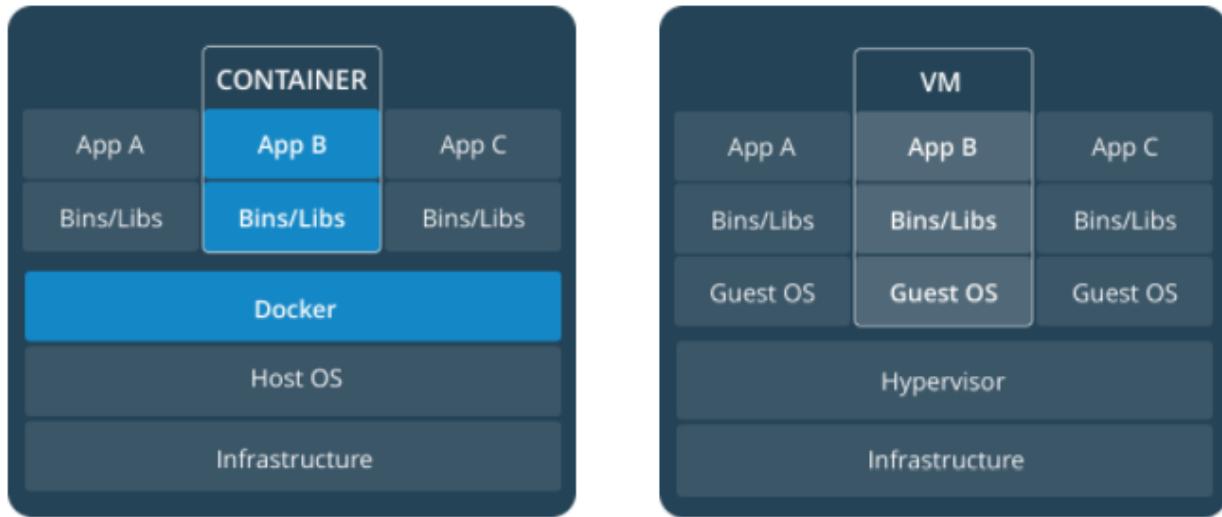
## Namespace 介绍

Namespace 是将内核的全局资源做封装，使得每个 Namespace 都有一份独立的资源，因此不同的进程在各自的 Namespace 内对同一种资源的使用不会互相干扰。

目前实现了以下几种 Namespace

- IPC 隔离 SystemV IPC 和 POSIX 消息队列，IPC 就是进程间通信
- Network 隔离网络资源，有独立的 网络设备, IP 地址, 路由表, /proc/net 目录。
- Mount 隔离文件系统挂载点，不同 Namespace 的进程看到的文件结构不同
- PID 隔离进程 ID
- UTS 隔离主机名和域名，UTS("UNIX Time-sharing System") 名字空间允许每个容器拥有独立的 hostname 和 domain name, 使其在网络上可以被视作一个独立的节点而非主机上的一个进程。
- User 隔离用户 ID 和组 ID，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。

## Docker 容器和虚拟机的区别



## 1. 实现原理技术不同

虚拟机是用来进行硬件资源划分的完美解决方案，利用的是硬件虚拟化技术，如此VT-x、AMD-V会通过一个 hypervisor 层来实现对资源的彻底隔离。

而容器则是操作系统级别的虚拟化，利用的是内核的 Cgroup 和 Namespace 特性，此功能通过软件来实现，仅仅是进程本身就可以实现互相隔离，不需要任何辅助。

## 2. 使用资源方面不同

Docker 容器与主机共享操作系统内核，不同的容器之间可以共享部分系统资源，因此更加轻量级，消耗的资源更少。

虚拟机会独占分配给自己的资源，不存在资源共享，各个虚拟机之间近乎完全隔离，更加重量级，也会消耗更多的资源。

## 3. 应用场景不同

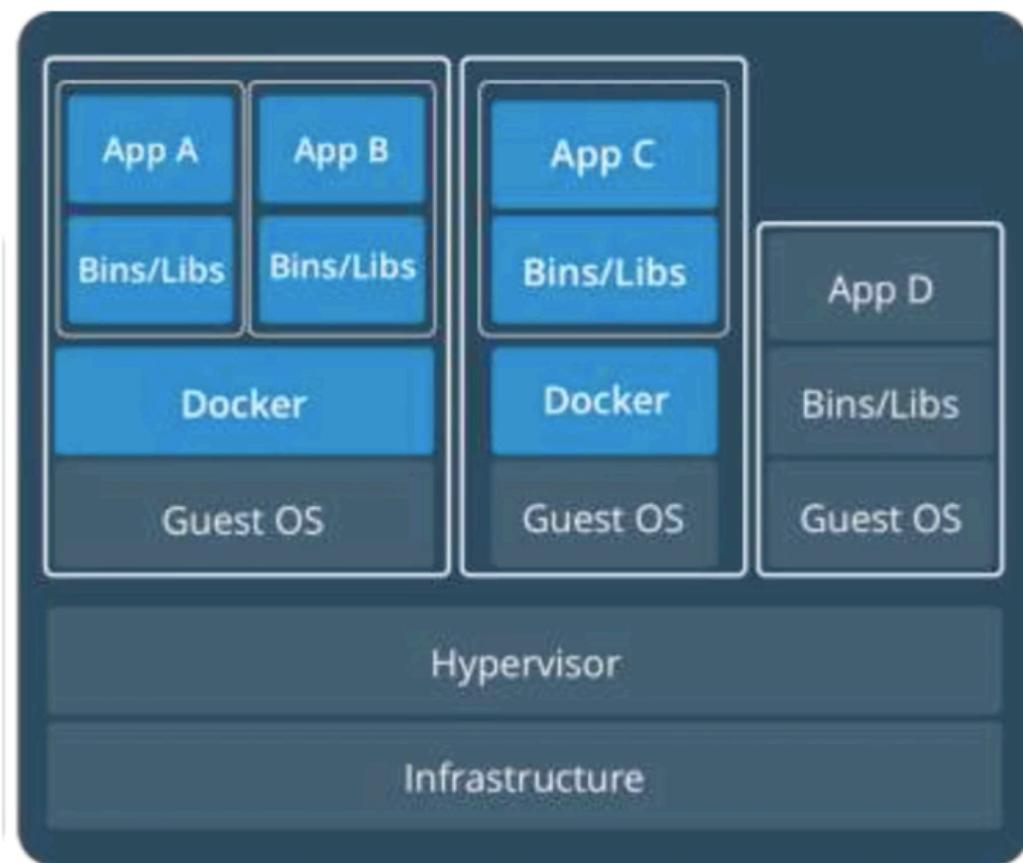
若需要资源的完全隔离并且不考虑资源的消耗，可以使用虚拟机。

若是想隔离进程并且需要运行大量进程实例，应该选择 Docker 容器。

## Docker 容器和虚拟机合作

可以在一台虚拟机上创建 Docker 容器

### 虚拟化+容器



## 什么是Docker

2013年初，一个叫dotCloud的PaaS服务提供商，将一个内部项目Docker开源。

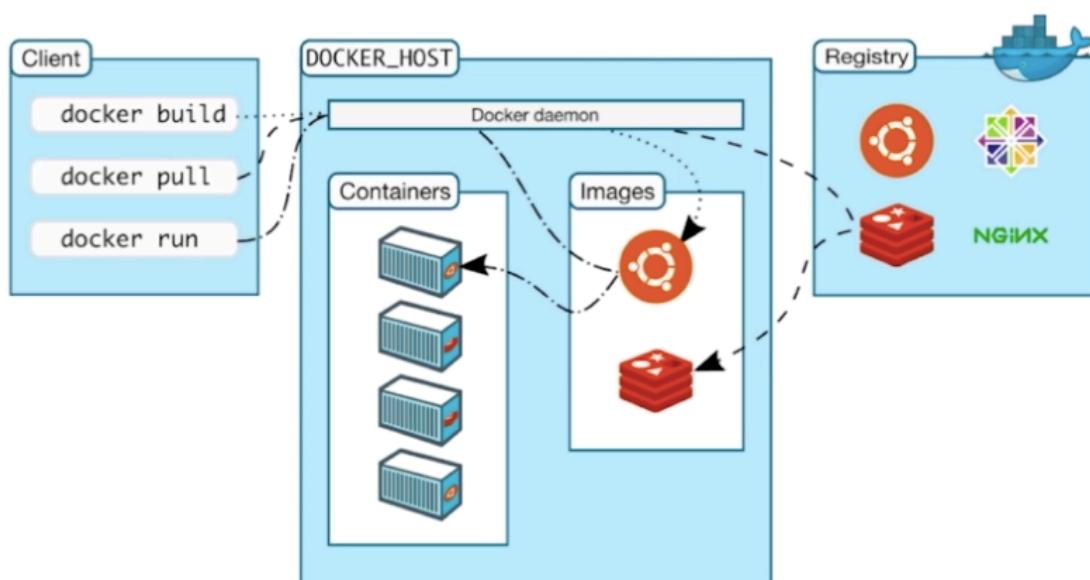
之后Docker一词迅速爆红。

这家公司干脆出售了其所持有的PaaS平台业务，并且改名为Docker.Inc，之后专注于Docker的开发和推广。

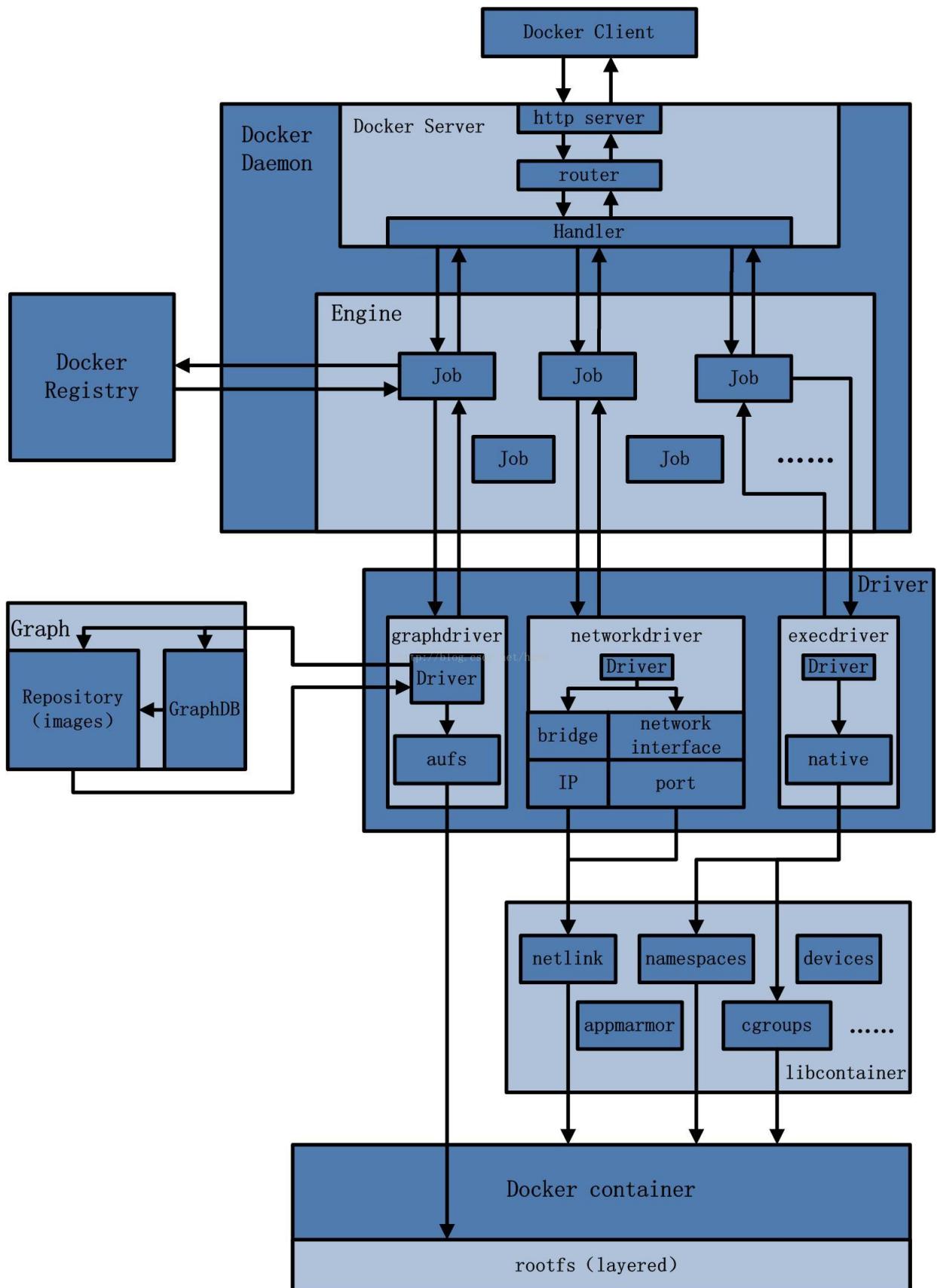
Docker实际上是一个开源的容器引擎，可以方便的对容器进行管理。

目前Docker已经加入Linux基金会，并遵循Apache 2.0协议，其代码托管于Github

<https://github.com/docker/docker>



# Docker 的架构



docker是一个C/S模式的架构，后端是一个松耦合架构，模块各司其职。

1. 用户是使用Docker Client与Docker Daemon建立通信，并发送请求给后者。
2. Docker Daemon作为Docker架构中的主体部分，首先提供Server的功能使其可以接受Docker Client的请求；
3. Engine执行Docker内部的一系列工作，每一项工作都是以一个Job的形式的存在。
4. Job的运行过程中，当需要容器镜像时，则从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载镜像以Graph的形式存储；
5. 当需要为Docker创建网络环境时，通过网络管理驱动networkdriver创建并配置Docker容器网络环境；
6. 当需要限制Docker容器运行资源或执行用户指令等操作时，则通过execdriver来完成。
7. libcontainer是一项独立的容器管理包，networkdriver以及execdriver都是通过libcontainer来实现具体对容器进行的操作

实际上 Docker 并不会直接于内核交互，是通过一个更底层的工具 Libcontainer 与内核交互的。

Libcontainer 才是真正意义上的容器引擎，它通过 clone 系统调用直接创建容器，通过 pivot\_root 系统调用进入容器，且通过直接操作 Cgroupfs 文件实现对资源的管控，而 Docker 更侧重于处理更上层的业务。

Docker 的另一个优势是对层级镜像的创新应用，即不同的容器可以共享底层的只读镜像。

通过写入自己特有的内容后添加新的镜像层，新增的镜像层和下层的镜像一起又可以作为基础镜像被更上层的镜像使用。

## Graph

先说 Union Mount，它支持把一个目录A叠加到另一个目录B之上；用户对目录B的读取就是A加上B的内容，而对B目录里文件写入和改写则会保存在目录A上，因为A在上一层。

比如一个最主要的应用是，把一张CD/DVD和一个硬盘目录给联合 mount在一起，然后，你就可以对这个只读的CD/DVD上的文件进行修改（当然，修改的文件存于硬盘上的目录里）。

Docker 对 Union mount 的应用，使得在多个容器使用通过一个基础镜像时，可以极大的减少内存占用，因为不同的容器访问同一个文件时，只会占用一份内存。这就需要使用支持 Union mount 的文件系统作为存储的 Graph设备，比如 AUFS 和 Overlay。

# Docker 功能和组件

- Docker 客户端
- Docker daemon
- Docker 容器
- Registry

## Docker 客户端

在 Linux 系统上，Docker 将客户端和服务端统一在同一个二进制文件中发布。

客户端用 docker command 来发起请求，也可以使用一整套 RESTful API 来发起请求。

## Docker daemon

可以被理解为 Docker server，另外也常常用 Docker Engine 来描述它，因为它实际上是驱动整个 Docker 功能的核心引擎。

接收客户端的请求，并实现请求所要求的功能，同时针对请求返回相应的结果。

## Docker 容器

在Docker 的功能和概念中，容器是一个核心内容，在性能上给 Docker 在虚拟化方面带来了极大的优势。

功能上 Docker 同过 Libcontainer 实现对容器生命周期的管理，信息的设置和查询，以及监控和通信等功能。

并且容器是对镜像的完美诠释，容器以镜像为基础，使镜像有了鲜活的生命，同时为镜像提供了一个标准的和隔离的运行环境。

## Docker 镜像

与容器对应，是一个还没有运行起来的环境。

包括了应用和这些应用运行时的环境。

它只是一个可定制的 rootfs。

Docker 镜像的另一创新就是 它是层级的并且可复用。

通常通过 Dockerfile 来创建，Dockerfile 提供了镜像内容的定制。

镜像就像是面向对象编程中的类，而容器就是这个类的实例

## Registry

是一个存放镜像的仓库，通常被部署在互联网服务器或者云端。

Registry 相当于传输软件的中转站。

Docker 公司提供了官方的 Registry，叫 Docker Hub。

Docker Hub 提供了大多数常用的软件和发行版的官方镜像，当然还有无数个人用户的个人镜像，都是免费的。

Registry 本身也是一个单独的开源项目，任何人都可以下载后部署自己的 Registry。

## Docker 安装和启动服务

### 安装

推荐按照官方文档安装

发行版本介绍

自2017年3月份 Docker 公司把 Docker 划分为了社区版和企业版

**Docker Community Edition(CE)** 为社区版, 免费

Docker CE有两个更新通道, 稳定和开发:

Stable每季度为您提供可靠的更新。 Edge每个月都会为您提供新功能。

**Docker Enterprise Edition** 为企业版, 收费

支持安全扫描, LDAP集成, 内容签名, 多云支持等

## CentOS7

<https://docs.docker.com/install/linux/docker-ce/centos/#install-using-the-repository>

### 1. 安装必要的依赖包

```
# yum install -y yum-utils device-mapper-persistent-data lvm2
```

- yum-utils 提供yum-config-manager实用程序
- devicemapper 存储驱动程序需要 device-mapper-persistent-data和lvm2。

### 2. 设置 Docker 标准发行版本的安装源

```
# yum-config-manager \
--add-repo \
https://download.docker.com/linux/centos/docker-ce.repo
```

### 3. 使用开发版本的源安装源

这些开发的存储库包含在上面的docker.repo文件中，但默认情况下处于禁用状态。您可以将它们与稳定的存储库一起启用。

```
# yum-config-manager --enable docker-ce-edge
```

当然也可以再次禁用它

```
# yum-config-manager --disable docker-ce-edge
```

## 4. 安装 Docker 社区版本

安装最新版本的Docker CE，或者转到下一步安装特定版本

```
# yum install -y docker-ce
```

如果提示接受GPG密钥，请验证指纹是否与060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35匹配，如果接受，请确认。

提示：

如果您启用了多个 Docker 存储库，比如有标准的安装源和开发测试源。则在 `yum install` 或 `yum update` 命令中安装或更新而不指定版本时，始终会安装尽可能高的版本，这可能不适合您的稳定性需求。

## 安装指定版本

## 1. 用下面的命令列出 Docker 的版本

```
# yum list docker-ce --showduplicates |  
sort -r
```

返回的列表取决于启用了哪些存储库，并且特定于您的 CentOS 版本（在本例中以.el7 后缀表示）。

## 2. 安装指定版本

通过完全限定的软件包名称（它是软件包名称（docker-ce）加上版本字符串（第二列））安装特定版本，例如 docker-ce-18.03.0.ce

```
# yum install docker-ce-<VERSION STRING>
```

## 启动服务

```
# systemctl start docker && systemctl enable  
docker
```

## 验证安装和服务是否正常

```
# docker run hello-world
```

这时，Docker 会主动去下载这个镜像，并用这个镜像启动一个容器；当容器运行时，它打印 `hello world` 并退出。

## 更多配置

参考官网

<https://docs.docker.com/install/linux/linux-postinstall/#configure-where-the-docker-daemon-listens-for-connections>

## 使用国内源进行加速

### 加速器

使用 Docker 的时候，需要经常从官方获取镜像，但是由于显而易见的网络原因，拉取镜像的过程非常耗时，严重影响使用 Docker 的体验。因此 DaoCloud 推出了加速器工具解决这个难题，通过智能路由和缓存机制，极大提升了国内网络访问 Docker Hub 的速度，目前已经拥有了广泛的用户群体，并得到了 Docker 官方的大力推荐。如果您是在国内的网络环境使用 Docker，那么 Docker 加速器一定能帮助到您。

### 版本要求

需要 Docker 1.8 或更高版本才能使用，如果您没有安装 Docker 或者版本较旧，请安装或升级。

## 支持的系统

Linux, MacOS 以及 Windows 平台。

## 是否收费

DaoCloud 为了降低国内用户使用 Docker 的门槛，提供永久免费的加速器服务，请放心使用。

## 使用阿里源

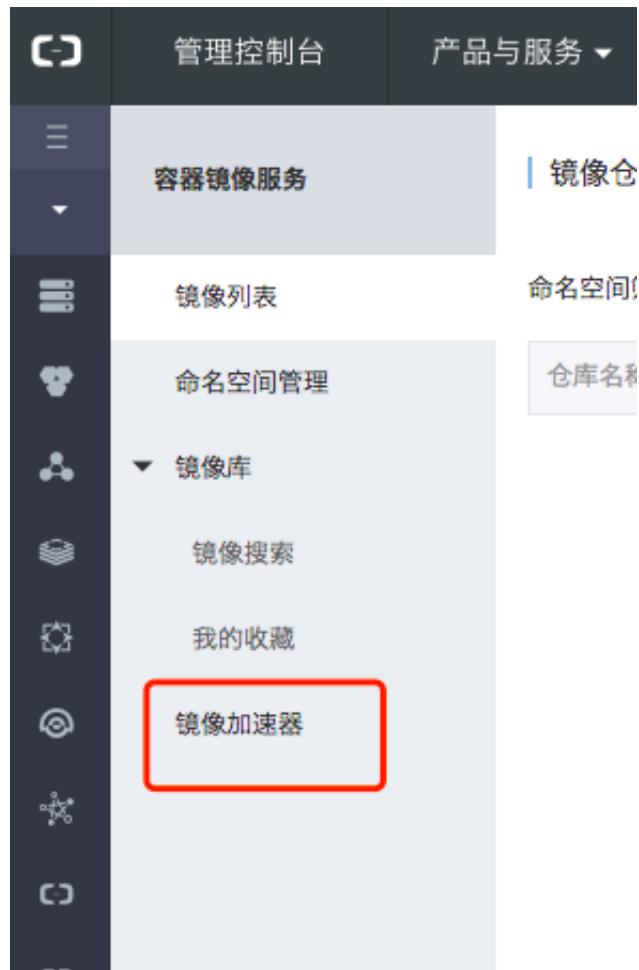
阿里云的镜像源有个加速器，可以加速你获取容器的速度。这个加速器地址是每个人专属的。

网址: <https://dev.aliyun.com/>

1. 需要注册个账号后登录上去，点管理中心



2. 再点击 镜像加速器



3. 再按照官方的操作文档修改配置文件即可

## 镜像加速器

使用加速器将会提升您在国内获取Docker官方镜像的速度！

您的专属加速器地址 [https://2023-03-14-14-40-11.uncs.com](https://<b>2023-03-14-14-40-11.uncs.com)



## 操作文档

Ubuntu

CentOS

Mac

Windows

### 安装/升级你的Docker客户端

推荐安装 **1.10.0** 以上版本的Docker客户端，参考文档 [docker-ce](#)

### 如何配置镜像加速器

针对Docker客户端版本大于1.10.0的用户

您可以通过修改daemon配置文件 `/etc/docker/daemon.json` 来使用加速器：

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
    "registry-mirrors": ["http://2023-03-14-14-40-11.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

# Docker 使用

## 获取命令行帮助信息

直接在命令行内输入 docker 命令后敲回车

部分截图：

```
[root@localhost ~]# docker
Usage: docker COMMAND
A self-sufficient runtime for containers
Options:
  --config string      Location of client config files (default "/root/.docker")
  -D, --debug          Enable debug mode
  -H, --host list      Daemon socket(s) to connect to
```

## 镜像管理

### 搜索镜像

#### docker search

```
[root@localhost ~]# docker search --help
Usage: docker search [OPTIONS] TERM
Search the Docker Hub for images

Options:
  -f, --filter filter    Filter output based on
conditions provided
                                         根据提供的条件过滤器输出
  --format string        Pretty-print search
using a Go template
                                         用Go模板打印出漂亮的搜索结
果
  --limit int           Max number of search
results (default 25)
```

搜索结果的最大数量（默认值  
为25）

--no-trunc

Don't truncate output

不要截断输出

实例：

默认搜索的是官方的镜像源仓库

```
[root@localhost ~]# docker search centos --  
limit 2  
NAME                  DESCRIPTION          STARS  
OFFICIAL              AUTOMATED  
centos                The official build of  
CentOS.               4308  
[ OK ]  
openshift/base-centos7 A CentOS7 derived  
base image for Source-To-I...    24
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
centos	The official build of CentOS.	4308	[OK]	
openshift/base-centos7	A CentOS7 derived base image for Source-To-I...	24		

获取镜像

使用 docker pull 命令可以将镜像下载到本地

```
[root@localhost ~]# docker pull centos
```

```
[root@localhost ~]# docker search centos --limit 3
NAME                                DESCRIPTION               STARS      OFFICIAL
centos                               The official build of CentOS.          4308      [OK]
openshift/base-centos7                A Centos7 derived base image for Source-To-I...   24
pivotaldata/centos-gpdb-dev          CentOS image for GPDB development. Tag names...   3

4 [root@localhost ~]# docker pull centos
Using default tag: latest
latest: Pulling from library/centos
469cfcc7a4b3: Pull complete
Digest: sha256:989b936d56b1ace20ddf855a301741e52abca38286382cba7f44443210e96d16
Status: Downloaded newer image for centos:latest
```

## 查看本地的镜像

```
[root@localhost ~]# docker images    # docker
image ls
REPOSITORY          TAG      IMAGE
ID                 CREATED     SIZE
hello-world         latest
e38bc07ac18e       6 weeks ago  1.85kB
centos              latest
e934aafc2206       6 weeks ago  199MB
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ansible/centos7-ansible	latest	51d539c192dd	2 weeks ago	480.2 MB
tomcat	latest	30d95ba23356	2 weeks ago	355.3 MB
nginx	latest	ba6bed934df2	2 weeks ago	181.3 MB
monitoringartist/zabbix-3.0-xxl	latest	f2632f31a98c	5 weeks ago	759.7 MB
centos	latest	980e0e4c79ec	5 weeks ago	196.7 MB
mysql	latest	4b3b6b994512	7 weeks ago	384.5 MB

TAG 表示镜像是最新  
的版本  
镜像名称  
镜像的ID

## 关于 TAG

当一个镜像的名称不足以分辨这个镜像所代表的含义时，你可以为具体的镜像添加 tag, 以便于区分特定的版本.

## 可以修改一个镜像的 TAG

命令格式为： docker tag IMAGEID 新镜像名称:新标签

实例如下：

```
[root@localhost ~]# docker tag 4842  
centos7:1.0
```

或者

```
docker tag 镜像名:tag 新镜像名称:新标签  
[root@localhost ~]# docker tag hello-  
world:latest hello-world:1.1
```

```
- container 0b1edfd5c4f/ is using its referenced image e38bc07ac18e  
[root@localhost ~]# docker images  
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE  
hello-world         latest    e38bc07ac18e   6 weeks ago   1.85kB  
centos              latest    e934aafc2206   6 weeks ago   199MB  
openshift/base-centos7  latest  4842f0bd3d61  15 months ago  383MB  
[root@localhost ~]# docker tag --help  
  
Usage: docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]  
  
Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE  
[root@localhost ~]# docker tag 4842 centos7:1.0  
[root@localhost ~]# docker images  
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE  
hello-world         latest    e38bc07ac18e   6 weeks ago   1.85kB  
centos              latest    e934aafc2206   6 weeks ago   199MB  
centos7             1.0      4842f0bd3d61  15 months ago  383MB  
openshift/base-centos7  latest  4842f0bd3d61  15 months ago  383MB  
[root@localhost ~]#
```

## 查看本地镜像的 IMAGE ID

```
[root@localhost ~]# docker images -q
```

## 查看一个镜像的制作历程

```
[root@localhost ~]# docker history centos
IMAGE           CREATED
CREATED BY
SIZE            COMMENT
e934aafc2206   6 weeks ago
/bin/sh -c #(nop)  CMD [ "/bin/bash" ]
0B
<missing>       6 weeks ago
/bin/sh -c #(nop) LABEL org.label-schema.sc...
0B
<missing>       6 weeks ago
/bin/sh -c #(nop) ADD file:f755805244a649ecc...
199MB
```

## 备份本地仓库的镜像

1. 用 save 子命令将本地仓库的镜像保存当前目录下

```
[root@localhost ~]# docker save -o
nginx.img.tar nginx
[root@localhost ~]# ls
nginx.img.tar
```

2. 将本地目录下的镜像备份文件导入到本地 Docker 仓库

```
# 方式一:  
[root@localhost ~]# docker load -i  
nginx.img.tar  
Loaded image: nginx:latest  
  
# -q 不输出信息  
  
# 方式二:  
[root@localhost ~]# docker load <  
nginx.img.tar  
82b81d779f83: Loading layer  54.21MB/54.21MB  
7ab428981537: Loading layer  3.584kB/3.584kB  
Loaded image: nginx:latest
```

## 删除镜像

要删除镜像必须确认此镜像目前没有被任何容器使用

docker rmi 镜像名:标签

```
[root@localhost ~]# docker rmi hello-world:1.1  
Untagged: hello-world:1.1
```

## 容器管理

运行一个容器

**docker run** 参数 镜像名称:tag 执行的命令

常用参数:

# 参数:

**-i** 保持和 docker 容器内的交互，启动容器时，运行的命令结束后，容器依然存活，没有退出（默认是会退出，即停止的）

**-t** 为容器的标准输入虚拟一个tty

**-d** 后台运行容器

**--rm** 容器在启动后，执行完成命令或程序后就销毁（不可于 **-d** 一起使用）

**--name** 给容器起一个自定义名称

**--restart** docker 1.12 新增加的参数，用来指定容器的重启策略，

当前提供的策略包括：

**no** 默认值，如果容器挂掉不自动重启。

**on-failure** 当容器以非 0 码退出时重启容器，同时可接受一个可选的最大重启次数参数（e.g. **on-failure:5**）。

**always** 不管退出码是多少都要重启，就算是你重启了 docker

daemon 服务，容器也会同时跟着重启。

--cap-add 允许容器里可以使用的功能或命令

运行一个容器时，若只是指定了镜像名，而没有指定其tag，docker默认会以tag为latest（最新版本）的镜像去启动容器，假如本地不存在这个镜像则先会报错；之后会尝试下载这个镜像，下载成功后再次运行容器和所要执行的命令。

## 示例：

### 1. 运行一个容器，并获得一个 tty

```
[root@localhost ~]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
nginx              latest   ae513a47849c  3 weeks ago  109MB
hello-world        latest   e38bc07ac18e  6 weeks ago  1.85kB
centos             latest   e934aafc2206  6 weeks ago  199MB
centos7            1.0     4842f0bd3d61  15 months ago 383MB
openshift/base-centos7  latest   4842f0bd3d61  15 months ago 383MB
```

```
[root@localhost ~]# docker run -it centos
[root@ff051d362141 /]# ls
anaconda-post.log  dev  home  lib64  mnt  proc
run    srv  tmp  var
bin          etc  lib  media  opt  root
sbin  sys  usr
[root@ff051d362141 /]# cat /etc/redhat-release
CentOS Linux release 7.4.1708 (Core)
[root@ff051d362141 /]#
```

# 退出容器，执行 exit 命令即可

使用 -it 选项运行的容器时，当退出当前容器后，或者容器内的程序（命令）执行结束后，容器会自动进入停止状态（除非运行时使用的参数 -d）

## 2. 运行一个容器，并且更改容器网络端口的状态

```
docker run --cap-add=NET_ADMIN ubuntu sh -c
"ip link eth0 down"
```

注意：后面要执行的命令需要容器内的系统有此命令。

## 3. 启动一个容器，并且禁止此容器内使用 chown 命令

```
[root@localhost ~]# docker run -it --cap-drop=CHOWN centos
[root@a139fa3eb118 /]# useradd shark
Setting mailbox file permissions: Operation
not permitted
[root@a139fa3eb118 /]# chown shark:shark
anaconda-post.log
chown: changing ownership of 'anaconda-
post.log': Operation not permitted
```

## 查看正在运行中的容器

```
[root@localhost ~]# docker ps
```

```
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           PORTS
TS                 centos             "/bin/bash"   20 minutes ago   Up 20 minutes
```

```
[root@x201t ~]# docker run -itd centos7 /bin/bash
cbe9955a3303612e56ebf6d2c2f26d7fa4c0330d61bab14f7b562740d7fd77d
[root@x201t ~]# docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           PORTS
cbe9955a3303       centos7          "/bin/bash"   11 seconds ago   Up 8 seconds    NAMES
r_swanson
```

启动镜像时运行的第一个命令  
可以看出容器是运行的状态  
采用的镜像名字  
COMMAND  
CREATED  
STATUS  
PORTS  
NAMES  
容器的名字

CONTAINER ID 容器ID

IMAGE 容器依赖的镜像

COMMAND 启动容器时执行的命令或程序

CREATED 容器启动时到现在的相隔时间

## STATUS 容器状态

### POR TS宿主机到容器的端口映射

注意: 当运行一个容器的时候, 没有用参数--name去指定容器名时, Docker会从自己的名称库中随机给这个容器起一个名字

.

### 查看本地所有的容器

```
[root@localhost ~]# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	退出状态
PORTS	NAMES				
78d6740efcfe	centos	"/bin/bash"	13 minutes ago	Exited	(1) 7 minutes ago
1aa15610dc8f	centos7:1.0	"container-entrypoint..."	16 minutes ago	Exited	(0) 16 minutes ago
b6c201a868a6	centos7:1.0	"container-entrypoint..."	19 minutes ago	Exited	(127) 19 minutes ago
5525f932659f	centos7:1.0	"container-entrypoint..."	25 minutes ago	Exited	(0) 23 minutes ago
ff051d362141	centos	"/bin/bash"	27 minutes ago	Up	27 minutes 运行状态
061edfd5c4f7	hello-world	"/hello"	23 hours ago	Exited	(0) 23 hours ago
	epic_proskuriakova				

### 查看最近一次启动的容器

不论此容器的目前状态是运行的或者停止的

```
[root@localhost ~]# docker ps -l
```

### 获取正在运行的容器的容器 ID

```
[root@localhost ~]# docker ps -q  
78d6740efcfe
```

## 获取所有容器的容器 ID

```
[root@localhost ~]# docker ps -aq  
78d6740efcfe  
1aa15610dc8f  
b6c201a868a6  
5525f932659f  
ff051d362141  
061edfd5c4f7
```

## 查看容器的元数据

查看容器的元数据信息，有启动时执行的命令或程序、运行时的IP、所使用的镜像等。

命令：docker inspect <容器ID|容器名>

```
命令：docker inspect <容器ID|容器名>
```

```
[root@localhost ~]# docker inspect 3f2a  
# 3f2a 是容器 ID
```

## 查看容器 IP

```
[root@localhost ~]# docker inspect --format='{{.NetworkSettings.IPAddress}}' 3f2a  
172.17.0.2
```

## 重命名容器名称

命令: docker rename OLD\_NAME NEW\_NAME

```
[root@localhost ~]# docker rename  
amazing_elion centos7.4
```

```
[root@localhost ~]# docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES  
78d6740efcfe        centos              "/bin/bash"         About an hour ago   Up 33 minutes          NAMES  
amazing_elion  
[root@localhost ~]# docker rename amazing_elion centos7.4  
[root@localhost ~]# docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES  
78d6740efcfe        centos              "/bin/bash"         About an hour ago   Up 33 minutes          NAMES  
centos7.4  
[root@localhost ~]#
```

## 重新启动一个处于停止状态的容器

```
[root@localhost ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
COMMAND             CREATED             STATUS
PORTS
NAMES
78d6740efcfe      centos
                  "/bin/bash"   6 hours ago
Exited (137) 5 hours ago
centos7.4
ff051d362141      centos
                  "/bin/bash"   7 hours ago
Exited (0) 6 hours ago
competent_shtern
```

## 启动容器，让其在后台运行

```
[root@localhost ~]# docker start 容器名/容器 ID
```

## 启动容器并获得一个终端

```
docker start -i 容器名/容器 ID
```

```
[root@localhost ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
78d6740efcfe      centos
                  "/bin/bash"   26 minutes ago
Up About a minute
g_elion
1aa15610dc8f      centos7:1.0
                  "container-entrap..." 29 minutes ago
Exited (0) 29 minutes ago
ed_morse
b6c201a868a6      centos7:1.0
                  "container-entrap..." 32 minutes ago
Exited (127) 32 minutes ago
lumiere
5525f932659f      centos7:1.0
                  "container-entrap..." 38 minutes ago
Exited (0) 36 minutes ago
blackwell
ff051d362141      centos
                  "/bin/bash"   40 minutes ago
Up 40 minutes
ent_shtern
061edfd5c4f7      hello-world
                  "/hello"     23 hours ago
Exited (0) 2 minutes ago
roskuriakova
[root@localhost ~]# docker start -i 78d
[root@78d6740efcfe ~]#
```

## 重启一个容器

```
docker restart 容器/容器 ID
```

## 停止一个容器

比如停止一个在后台运行的容器

```
docker stop 容器 ID
```

```
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE
COMMAND             CREATED
PORTS               NAMES
78d6740efcfe      centos
                  "/bin/bash"
                  29 minutes ago   Up 4
minutes
amazing_elion
[root@localhost ~]# docker stop 78d
78d
```

## 关闭和删除容器

```
docker rm 容器 ID
```

```
[root@localhost ~]# docker ps -a
CONTAINER ID        IMAGE
COMMAND             CREATED
STATUS              PORTS
NAMES
78d6740efcfe      centos
    "/bin/bash"          41 minutes ago
Up 4 minutes
    amazing_elion
061edfd5c4f7      hello-world
    "/hello"            23 hours ago
Exited (0) 16 minutes ago
    epic_proskuriakova
[root@localhost ~]# docker rm hello-world      #
名字是不好使的
Error: No such container: hello-world
[root@localhost ~]# docker rm 061                  #
用容器 ID
061
```

## 在外部执行运行状态中容器的命令

在宿主机上执行处于运行状态的容器的命令

语法：

docker exec 容器名/容器 ID (不是镜像名/镜像ID)

执行的命令 [参数]

示例如下：

```
[root@localhost ~]# docker exec amazing_elion  
cat /etc/redhat-release  
CentOS Linux release 7.4.1708 (Core)
```

```
[root@localhost ~]# docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES  
78d6740efcfe        centos              "/bin/bash"         45 minutes ago   Up 8 minutes            
[root@localhost ~]# docker exec amazing_elion cat /etc/redhat-release  
CentOS Linux release 7.4.1708 (Core) ←  
[root@localhost ~]#
```

## 进入到一个正在运行中的容器

当使用 -d 参数时，容器启动后会进入后台运行。某一些时候需要进入容器内操作.

目前有两种方法：

attach 命令和nsenter工具

### attach

```
docker attach 容器 ID
```

注意：这种方式，当你退出容器后（在容器里的终端里输入exit或[Ctrl]+d退出时），这个容器就会处于停止状态；不论之前他是否以后台方式运行。以为exit会向容器的主进程发送一个SIGKILL信号。

解决办法是:[Ctrl]-p+[Ctrl]-q

就是先按[Ctrl]+p键，接着再按[Ctrl]+q键

## nsenter(推荐)

使用nsenter工具，需要安装 util-linux 软件包

```
[root@localhost ~]# yum install util-linux  
-y
```

先获取到运行中的容器的pid

### 1. 先查看运行中的 容器

```
[root@localhost ~]# docker ps  
CONTAINER ID        IMAGE          COMMAND       CREATED          STATUS          PORTS          NAMES  
78d6740efcfe        centos         "/bin/bash"    7 hours ago     Up 8 minutes   centos7.4
```

### 2. 利用容器ID或容器名获得到容器到 pid

```
[root@localhost ~]# PID=$(docker inspect -f "  
{{ .State.Pid }}" 78d)
```

### 3. 通过或得的 PID连接到这个容器

```
[root@localhost ~]# nsenter --target $PID --  
mount --uts --ipc --net --pid
```

## 作业

把此功能写成一个脚本

## 监控容器的运行

### 1. 查看日志

可以通过使用docker logs命令来查看容器的运行日志

--tail 选项可以指定查看最后几条日志

-t 选项则可以对日志条目附加时间戳。

--until 显示在某个时间戳(例如：2018-05-25T 13:23:37)之前  
的日志，还可以相对时间 (例如：42m 42 minutes)

查看 2018年5月25日之前的日子

```
[root@localhost ~]# docker logs --until 2018-  
05-25 78d
```

## 2. 查看运行中容器的进程

```
[root@localhost ~]# docker top 78d
```

## 在宿主机和容器之间交换文件

--在宿主机和容器之间相互COPY文件 cp的用法如下：

```
docker cp [OPTIONS] CONTAINER:PATH LOCALPATH  
docker cp [OPTIONS] LOCALPATH|- CONTAINER:PATH
```

如：容器 centos7.4 中/root/a.txt, copy 到宿主机的当前目录下

```
[root@localhost ~]# docker cp  
centos7.4:/root/a.txt .  
[root@localhost ~]# ls  
anaconda-ks.cfg a.txt
```

修改完毕后，将该文件重新copy回容器

```
[root@localhost ~]# echo "update" > a.txt  
[root@localhost ~]# docker cp ./a.txt  
centos7.4:/root/a.txt
```

## 查看容器内发生改变的文件

使用 diff 子命令

### 1. 先在容器内容创建目录和文件

```
[root@localhost ~]# nsenter --target $PID --  
mount --uts --ipc --net --pid  
[root@78d6740efcfe /]# mkdir change_dir  
[root@78d6740efcfe /]# touch  
change_dir/file.txt
```

### 2. 在容器外部查看改变

```
[root@localhost ~]# docker diff 78d  
A /change_dir  
A /change_dir/file.txt  
C /root  
A /root/.bash_history  
A /test
```

A 表示在原来的基础上，做了添加

C 表示在原来的基础上，做了改变

## 实时输出Docker服务端的事件

events 子命令 实时输出Docker服务器端的事件，包括容器的创建，启动，关闭等

## 1. 先监听

```
[root@localhost ~]# docker events
```

## 2. 触发事件

启动一个容器

```
[root@localhost ~]# docker start -i myhell-docker
```

## 3. 输出的结果

```
[root@localhost ~]# docker events
2018-05-25T06:30:13.652207351+08:00 container
attach
2fc892450fe68a3b6e37012285643cb3b4a09f5b90baab
91b91dea6d3e6d68ff (image=hello-world:latest,
name=myhell-docker)
2018-05-25T06:30:13.664984219+08:00 network
connect
d145648388142745aa1bc99a8c6a6197b0bd3db67b85d9
c000f5e4744c4b20b3
(container=2fc892450fe68a3b6e37012285643cb3b4a
09f5b90baab91b91dea6d3e6d68ff, name=bridge,
type=bridge)
```

```
2018-05-25T06:30:14.398288392+08:00 container
start
2fc892450fe68a3b6e37012285643cb3b4a09f5b90baab
91b91dea6d3e6d68ff (image=hello-world:latest,
name=myhell-docker)
2018-05-25T06:30:14.494988113+08:00 container
die
2fc892450fe68a3b6e37012285643cb3b4a09f5b90baab
91b91dea6d3e6d68ff (exitCode=0, image=hello-
world:latest, name=myhell-docker)
2018-05-25T06:30:14.540358085+08:00 network
disconnect
d145648388142745aa1bc99a8c6a6197b0bd3db67b85d9
c000f5e4744c4b20b3
(container=2fc892450fe68a3b6e37012285643cb3b4a
09f5b90baab91b91dea6d3e6d68ff, name=bridge,
type=bridge)
```

## 创建一个容器但不启用

需要基于本地的镜像创建容器

### 1. 查看本地镜像

```
[root@localhost ~]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
nginx              latest   ae513a47849c  3 weeks ago  109MB
hello-world        latest   e38bc07ac18e  6 weeks ago  1.85kB
```

## 2. 创建容器

下面的示例是以 hello-world 为基础镜像，创建一个名为 myhello-world 的容器

```
[root@localhost ~]# docker create --name
"myhello-docker" hello-world:latest
2fc892450fe68a3b6e37012285643cb3b4a09f5b90baab
91b91dea6d3e6d68ff
[root@localhost ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
2fc892450fe6      hello-world:latest  "/hello"    4
minutes ago        Created            myhell-
docker
```

## 把容器制作成一个镜像

就像是给容器做快照和恢复快照

## 1. 首先需要导出容器，用 docker export 命令

```
[root@x201t ~]# docker ps -a
CONTAINER ID        IMAGE       COMMAND      CREATED     STATUS          PORTS
 NAMES
f7d6cb83d6fa      centos7    "/bin/bash"   4 minutes ago   Exited (137) 7 seconds ago
grave_ramanujan

[root@x201t ~]# docker export f7d6cb83d6fa >centos7.tar
[root@x201t ~]# ls
1.txt              anaconda-ks.cfg  docker_nsenter.sh  gns3-gui-1.4.6-1.1.src.rpm  set.sh      下载
a                  a.txt           dynagen-0.11.0    gns3-gui-1.4.6-1.1.x86_64.rpm  sharkyun.log
aa.sh              centos7.tar    dynagen-0.11.0.tar.gz  netstat.txt  ss.txt
access.log-20160710 ddos.sh        file            pass_file  testip.txt
a.file             Desktop        GNS3            rpmbuild  test.sh
[root@x201t ~]#
```

## 2. 把导出的tar文件制作成镜像，使用 docker import 命令

```
[root@x201t ~]# cat centos7.tar | docker import - test/centos7:v1.0
sha256:0355d1d322a26a5f9ecfe5a6f2570cb512fe6877f505989b03c3d80e393ce20
[root@x201t ~]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED     SIZE
test/centos7        v1.0    0355d1d322a2  16 seconds ago  196.7 MB
ansible/centos7-ansible latest  51d539c192dd  2 weeks ago   480.2 MB
tomcat              latest  30d95ba23356  3 weeks ago   355.3 MB
nginx               latest  ba6bed934df2  3 weeks ago   181.3 MB
zabbix              3.0    f2632f31a98c  5 weeks ago   759.7 MB
centos7             latest  980e0e4c79ec  5 weeks ago   196.7 MB
centos              7.0    980e0e4c79ec  5 weeks ago   196.7 MB
centos              7.2    980e0e4c79ec  5 weeks ago   196.7 MB
mysql               latest  4b3b6b994512  7 weeks ago   384.5 MB
oracle              latest  1988eb5b3fc6  12 weeks ago  278.2 MB
hello-world         latest  c54a2cc56cbb  3 months ago  1.848 kB
wnameless/oracle-xe-11g latest  b4d052e20bda  3 months ago  2.227 GB
treasureboat/centos6 latest  869ead6d5987  20 months ago 465.7 MB
[root@x201t ~]#
```

表示把cat的内容作为docker import 的标准输入

注意：

docker import 和 docker load 的区别在于：

load 是用来导入镜像存储文件到本地镜像库的，镜像存储文件是用 save 从本地镜像库保存到本地硬盘的镜像备份文件，一般容量相对容器的快照文件较大，保存的是完整的记录，导入时，不能重新指定标签（tag）等元数据信息；

而 import 导入的是容器的快照文件，容器的快照文件体积较小，它丢弃了历史记录和元数据信息，仅仅保存容器当时的快照状态。

# 镜像详解

镜像是容器的基础，是容器静态的运行环境。就像是一个程序代码，当你运行了这个程序，此时它的动态表现形式就是在内存中，叫进程。

## 创建自己的镜像

创建自己的镜像可以使用 `docker commit` 命令，从一个你配置好环境的容器中创建一个镜像，这个镜像会自动放到你的本地仓库。但是强烈建议不应该这样自定义你的镜像。应为这会带来一些安全隐患，也不利于分享传输你的环境。

正确的做法是通过 Dockerfile 来定制镜像。

提示： 镜像是多层的，每一层都是在前一层的基础上进行的修改。

当然容器也是多层存储的，当容器运行时，会以镜像为基础层，之后在镜像的基础层上加一层，来作为容器运行时的存储层。

下面我们就以在 Centos 这个基础镜像启动一个容器，之后在容器里安装一下我们所要的工具，之后再保存容器中的操作，分别用 commit 方式和 Dockerfile 方式来演示这个过程。

## 通过容器创建

### 1. 启动容器，并安装软件

```
[root@localhost ~]# docker run -it  
centos:latest /bin/bash  
[root@89b5c240ba58 /]# rpm -qa |grep vim  
vim-minimal-7.4.160-2.el7.x86_64  
[root@89b5c240ba58 /]# vim a  
bash: vim: command not found  
[root@89b5c240ba58 /]# rpm -qa |grep tree  
[root@89b5c240ba58 /]# yum install tree vim*
```

### 1. 安装完毕，验证

```
[root@89b5c240ba58 /]# rpm -qa |grep tree  
tree-1.6.0-10.el7.x86_64  
[root@89b5c240ba58 /]# rpm -qa |grep vim  
vim-filesystem-7.4.160-4.el7.x86_64  
vim-enhanced-7.4.160-4.el7.x86_64  
vim-minimal-7.4.160-4.el7.x86_64  
vim-common-7.4.160-4.el7.x86_64  
vim-X11-7.4.160-4.el7.x86_64  
[root@89b5c240ba58 /]#
```

## 1. commit 提交到本地仓库

commit 语法：

```
docker commit [选项] <容器ID或容器名> [<镜像名>[:<标签>]]
```

示例：

打开另外一个终端，或者退出容器，在宿主机上执行如下命令：

### 1. 先查看本地仓库的镜像情况

```
[root@localhost ~]# docker image ls | grep centos
centos
centos          latest
e934aafc2206    7 weeks ago      199MB
centos7         1.0
4842f0bd3d61    16 months ago     383MB
openshift/base-centos7   latest
4842f0bd3d61    16 months ago     383MB
```

### 1. 再确认一下要提交的容器 ID 或者 容器名

```
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE
COMMAND            CREATED             STATUS
                 PORTS
89b5c240ba58      centos:latest
"/bin/bash"         20 minutes ago   Up 20
minutes
zealous_nobel
```

## 1. 开始提交到本地仓库

```
[root@localhost ~]# docker commit \
> --author "yangge <yangge@qf.com>" \
> --message "Install tree vim*" \
> 89b5c240ba58 \
> centos:1.10

sha256:8f1a0dbcef683c3ca22522f3dbac63633734f3d
832a21cc425feaac8d4a21857
```

参数说明：

- --author 作者
- --message 说明信息

自定义镜像最佳实战，必须养成写作者和说明信息的习惯

## 1. 验证

```
[root@localhost ~]# docker image ls centos
REPOSITORY          TAG      IMAGE
ID                  CREATED   SIZE
centos              1.10
8f1a0dbcef68      7 minutes ago   372MB
centos              latest
e934aafc2206      7 weeks ago    199MB
```

思考：

刚才我们安装了软件之后并没有清理缓存等环境，这样有可能导致镜像的臃肿。

还有可能是，你制作好的镜像别人不一定能信任，就像你不会随便从其他地方拉取一个镜像来作为自己的生产环境的容器一样。

更可怕的是，容器和镜像都是分层的，对于当前层的修改，是不会影响之前层的，之前层都是只读的。加入某一次对其修改有误，想再修改回来，那就会又增加一层，导致更加臃肿。

所以正确的做法是采用下面的方式 Dockerfile。

## 通过 Dockerfile 创建

我们可以把刚才的对容器的所有操作命令都记录到一个文件里，就像写更脚本程序。

之后用 docker build 命令以此文件为基础制作一个镜像，并会自动提交到本地仓库。

这样的话镜像的构建会变的透明化，为维护也更加简单，只修改这个文件即可。

同时分享也更加简单快捷，因为只要分享这个文件即可。

Dockerfile 是一个普通的文本文件，文件名必须叫 Dockerfile 其中包含了一系列的指令(Instruction)，每一条指令都会构建一层，就是描述该层是如何创建的。

示例：

## 1. 编辑 Dockerfile 文件

```
[root@localhost ~]# mkdir centos_dockerfile
[root@localhost ~]# cd centos_dockerfile/
[root@localhost centos_dockerfile]# vi
Dockerfile
FROM centos:latest
LABEL maintainer="yangge <yangge@qf.com>"
description="Install tree vim*"
RUN rpm -qa | grep tree || yum install -y
tree \
vim*
```

指令介绍：

- FORM 定义一个基础镜像
- LABEL 定义一些元数据信息，比如作者、版本、关于镜像的描述信息
- RUN 执行命令行的命令

编辑完，保存退出

## 2.开始构建镜像

命令语法格式：

```
docker bulid -t 仓库名/镜像名:tag .
```

```
docker build [选项] <上下文路径/URL/->
```

示例：

```
[root@localhost centos_dockerfile]# docker
build -t centos:1.20 .
Sending build context to Docker daemon
2.048kB
Step 1/3 : FROM centos:latest
--> e934aafc2206
Step 2/3 : LABEL
maintainer="shark<dockerhub@163.com>"
description="Install tree vim*"
--> Using cache
```

```
---> 1207b2848015
Step 3/3 : RUN rpm -qa | grep tree || yum
install -y tree      vim*
---> Running in 33d321b249d7
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
...略...
Complete!
Removing intermediate container 33d321b249d7
---> adc30981bc84
Successfully built adc30981bc84      # 表示构建成功
Successfully tagged centos:1.20      # TAG 标签
[root@localhost centos_dockerfile]#
```

## 1. 验证

```
[root@localhost centos_dockerfile]# docker
image ls centos
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
centos              1.20    adc30981bc84  3 minutes ago  372MB
centos              1.10    8f1a0dbcef68  About an hour ago  372MB
centos              latest   e934aaafc2206  7 weeks ago   199MB
```

## 构建镜像的上下文(context)

这个 `.` 表示当前目录，这实际上是在指定上下文的目录是当前目录，`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。

`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。

## 最佳实战

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

`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.qf` 参数指定某个文件作为 `Dockerfile`。

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。

```
[root@localhost dockerfile_qf_ignore]# tree .
.
├── Dockerfile.qf
└── test
    ├── a.txt
    ├── b.txt
    └── test.qf

[root@localhost dockerfile_qf_ignore]# cat Dockerfile.qf
FROM alpine
COPY ./test.qf /root/test.qf
```

```
[root@localhost dockerfile_qf_ignore]# docker  
build -f ../Dockerfile.qf -t alpine:test.qf .
```

```
[root@localhost dockerfile_qf_ignore]# docker  
run -it alpine:test.qf /bin/sh
```

## Dockerfile 详解

### FROM 指令

主要作用是指定一个镜像作为构建自定义镜像的基础镜像，在这个基础镜像之上进行修改定制。

这个指令是 Dockerfile 中的必备指令，同时也必须是第一条指令。

在 [Docker Store](#) 上有很多高质量的官方镜像，可以直接作为我们的基础镜像。

作为服务类的，如 [Nginx](#) [Mongo](#) 等

用于开发的，如 [Python](#) [golang](#)

操作系统类，如 [Centos](#) [ubuntu](#)

除了一些现有的镜像，Docker 还有一个特殊的镜像 `scratch`

这个镜像是虚拟的，表示空白镜像

```
FROM scratch
```

```
...
```

这以为着这将不以任何镜像为基础镜像。

可以把可执行的二进制文件复制到镜像中直接执行，容器本身就是和宿主机共享 Linux 内核的。

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。

## 制作自己的 Hello world

1. 在任意一台 Linux 机器上，安装 gcc

查看有没有安装

```
[root@localhost hello_qf]# rpm -qa gcc glibc-static  
glibc-static-2.17-222.el7.x86_64  
gcc-4.8.5-11.el7.x86_64
```

没有的话，进行安装即可

```
[root@localhost hello_qf]# yum install gcc  
glibc-static
```

## 1. 编辑 C 源代码文件

```
[root@localhost docker]# cat hello.c  
#include <stdio.h>  
  
int main()  
{  
    printf("Hello, Yangge! \n");  
    return 0;  
}
```

## 1. 将源代码文件编译为可执行的二进制文件

```
[root@localhost hello_qf]# gcc --static  
hello.c -o hello
```

编译好后，测试一下

```
[root@localhost hello_qf]# ls  
hello  hello.c  
# hello 是我们编译好的二进制文件  
# hello.c 是 c 的源码文件  
[root@localhost hello_qf]# ./hello  
Hello Yangge      # 输出结果，说明编译成功
```

## 1. 编辑 Dockerfile

在有 hello 二进制的文件目录下，编译 Dockerfile 文件，内容如下：

```
[root@localhost hello_qf]# ls  
Dockerfile  hello  hello.c  
[root@localhost hello_qf]# cat Dockerfile  
FROM scratch  
ADD hello /  
CMD [ "/hello" ]
```

- ADD 是把当前目录下的 hello 文档拷贝到 容器中的根目录下
- CMD 执行根目录下的 hello 文件

## 2. 构建新的镜像

注意命令的最后有个 `.`

```
[root@hello_qf]# docker build -t  
xiguatian/hello-yange .  
Sending build context to Docker daemon  
868.9kB  
Step 1/3 : FROM scratch  
-->  
Step 2/3 : ADD hello /  
--> 63ed3c13b7fd  
Step 3/3 : CMD [ "/hello" ]  
--> Running in a26622affa68  
Removing intermediate container a26622affa68  
--> dfadd4a86525  
Successfully built dfadd4a86525  
Successfully tagged xiguatian/hello-  
yange:latest
```

### 3. 查看本地仓库验证

```
[root@localhost hello_qf]# docker image ls  
xiguatian/hello-yange  
REPOSITORY          TAG      IMAGE ID  
CREATED            SIZE  
xiguatian/hello-yange latest  dfadd4a86525  3  
minutes ago        865kB
```

可以看到镜像很小

### 4. 利用新的镜像运行一个容器

```
[root@localhost hello_qf]# docker run --rm  
qf/hello-yange  
Hello Yangge
```

## 关于 Alpine

官网：<https://alpinelinux.org/>

WIKI [https://wiki.alpinelinux.org/wiki/Main\\_Page](https://wiki.alpinelinux.org/wiki/Main_Page)

Alpine Linux是一款独立的非商业性通用Linux发行版，专为那些了解安全性，简单性和资源效率的高级用户而设计。

### 小

Alpine Linux围绕musl libc和busybox构建。这使得它比传统的GNU / Linux发行版更小，更节省资源。一个容器需要不超过8 MB的空间，而对磁盘的最小安装需要大约130 MB的存储空间。您不仅可以获得完整的Linux环境，还可以从存储库中选择大量的软件包。

二进制软件包被缩减和拆分，使您可以更好地控制安装的内容，从而使您的环境尽可能地小巧高效。

### 简单

Alpine Linux是一个非常简单的发行版，它会尽量避免使用。它使用自己的包管理器，称为apk，OpenRC init系统，脚本驱动的设置，就是这样！这为您提供了一个简单，清晰的Linux环境，没有任何噪音。然后，您可以添加项目所需的软件包，因此无论是构建家用PVR还是iSCSI存储控制器，薄型邮件服务器容器或坚如磐石的嵌入式交换机，其他都不会挡道。

## 安全

Alpine Linux的设计考虑到了安全性。内核修补了一个非官方的grsecurity / PaX端口，并且所有的用户级二进制文件被编译为位置独立可执行文件（PIE）和堆栈粉碎保护。这些主动安全功能可防止利用整个类别的零日等漏洞。

## LABEL 指令

LABEL 指令用于指定一个镜像的描述信息

该 LABEL 指令将元数据添加到镜像中。

LABEL 是一个键值对。

要在 LABEL 值中包含空格，请像在命令行解析中一样使用引号和续行符 \ 。

几个用法示例：

```
LABEL maintainer="yangge@qf.com"
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

一个镜像可以有多个 `LABEL` 标签。您可以在一行中指定多个标签。这减少了最终映像的大小，但现在不再会影响到镜像的大小了。

但是仍然可以把他们写在一行或用反斜线进行续航

```
LABEL multi.label1="value1"
multi.label2="value2" other="value3"
```

```
LABEL multi.label1="value1" \
multi.label2="value2" \
other="value3"
```

有继承关系的镜像，标签也会有面向对象编程中继承的关系和特性

要查看镜像的 `LABEL` 信息，请使用该 `docker inspect` 命令。

## ENV 指令

### 用于设置环境变量

格式有两种：

- `ENV <key> <value>`
- `ENV <key1>=<value1> <key2>=<value2>...`

示例：

### 推荐方式

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

不推荐

```
ENV NODE_VERSION 7.2.0
```

其他指令使用：

```
RUN echo $NODE_VERSION
```

```
...
```

下列指令可以支持环境变量：

`ADD`、`COPY`、`ENV`、`EXPOSE`、`LABEL`、`USER`、`WORKDIR`、`VOLUME`、`STOPSIGNAL`、`ONBUILD`。

## RUN 指令

`RUN` 指令是在容器内执行 shell 命令，默认会是用 `/bin/sh -c` 的方式执行。

### 执行命令的两种方式

- `RUN <command>` (*shell*形式，该命令在*shell*中运行)
- `RUN [ "executable", "param1", "param2" ]` (*exec*形式)

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。

注意：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。

所以，在使用 *shell* 方式，尽量多的使用续行符 \

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
```

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。

注意当使用 `exec` 方式时，需要明确指定 `shell` 路径，否则变量可能不会生效

```
FROM centos
ENV name="yangge"
RUN [ "/bin/echo", "$name" ]
```

```
[root@localhost exec]# vi dockerfile
[root@localhost exec]# docker build -t alpine:exec .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM centos
--> 892caf4398fc
Step 2/3 : ENV name="yangge"
--> Using cache
--> b7447ada8998
Step 3/3 : RUN ["/bin/echo", "$name"]
--> Running in 56df7d339802
$name
Removing intermediate container 56df7d339802
--> 22335d341309
Successfully built 22335d341309
Successfully tagged alpine:exec
```

```
FROM centos
ENV name="yangge"
RUN [ "/bin/bash", "-c", "/bin/echo $name" ]
```

注意：exec表单被解析为JSON数组，这意味着您必须在单词周围使用双引号（"）而非单引号（'）。

```
[root@localhost exec]# docker build -t alpine:exec .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM centos
--> 892caf4398fc
Step 2/3 : ENV name="yangge"#
--> Using cache
--> b7447ada8998
Step 3/3 : RUN ["/bin/bash", "-c", "/bin/echo $name"]
--> Running in af47433e750f
yangge
Removing intermediate container af47433e750f
--> 8ac11215bed8
Successfully built 8ac11215bed8
Successfully tagged alpine:exec
```

## CMD 指令

Dockerfile 中只能有一条 CMD 指令。如果列出多个，CMD 则只有最后一个 CMD 会生效。

### CMD 主要目的是为运行容器时提供默认值

Docker 不是虚拟机，容器就是进程，CMD 指令就是用于指定默认的容器主进程的启动命令的。在启动(运行)一个容器时可以指定新的命令来替代镜像设置中的这个默认命令。

可以包含可执行文件，当然也可以省略。

`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。

**注意：**不要混淆 `RUN` 使用 `CMD`。`RUN` 实际上运行一个命令并提交结果；`CMD` 在构建时不执行任何操作，但指定镜像的默认命令。

Docker 不是虚拟机，容器内没有后台服务的概念。

不要视图这样启动一个程序到后台：

```
CMD systemctl start nginx
```

这行被 `Docker` 理解为：

```
CMD ["sh" "-c" "systemctl start nginx"]
```

对于容器而言，其启动程序就是容器的应用进程，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。

就像上面的示例中，主进程是 `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，自然就会使容器退出。

正确的做法是直接执行 `nginx` 这个可执行文件，并且不以后台守护进程的方式运行。

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

## ENTRYPOINT 指令

`ENTRYPOINT` 的目的和 `CMD` 一样，都是在指定容器的启动程序及参数。

`ENTRYPOINT` 在运行时也可以被替代，不过比 `CMD` 要略显繁琐，需要通过 `docker run` 的参数 `--entrypoint` 来指定。

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

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，也就是实际执行时，将变为：

```
<ENTRYPOINT> "<CMD>"
```

有了 `CMD` 后，为什么还要有 `ENTRYPOINT` 呢？

这种 `<ENTRYPOINT> "<CMD>"` 给我们带来了什么好处么？

让我们来看几个场景。

- 场景一：让镜像变成像命令一样使用

### CMD 方式

```
FROM centos
RUN yum update \
    && yum install -y curl
CMD [ "curl", "-s", "http://ip.cn" ]
```

构建镜像后，运行容器

```
# docker run --rm centos-echo-ip-cmd
```

执行下面命令会报错

```
# docker run --rm centos-echo-ip-cmd -i
```

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。

## ENTRYPOINT 方式

```
FROM centos
RUN yum install -y curl
ENTRYPOINT ["curl", "-s", "http://ip.cn"]
```

再次构建镜像后，运行容器

```
# docker run --rm centos-echo-ip-entrypoint
```

```
# docker run --rm centos-echo-ip-entrypoint -i
```

- 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。

官方镜像 `redis` 中的示例：

```
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 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 gosu 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)
```

还有 `ENTRYPOINT` 指令不会被 `RUN` 指令覆盖，而 `CMD` 指令会被 `RUN` 指令覆盖

## WORKDIR 指令

用于声明当前的工作目录，以后各层的当前目录就被改为指定的目录。

格式为 `WORKDIR <工作目录路径>`。

如该目录不存在，`WORKDIR` 会帮你建立目录。

再次强调！不要以为编写 `Dockerfile` 是在写 `shell` 脚本。

下面是一个错误示例：

```
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，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。

两行 `RUN` 分别构建了并启动了各自全新的容器。

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。

```
FROM alpine
WORKDIR /a/b
RUN touch a_b_f.txt
WORKDIR /a
RUN touch a_f.txt
```

```
[root@localhost workdir]# docker run -it
alpine:workdir /bin/sh
/a # ls
a_f.txt  b
/a # cd b
/a/b # ls
a_b_f.txt
```

# COPY 指令

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。

<目标路径> 可以是容器内的绝对路径，也可以是相对于 WORKDIR 指定的工作目录的相对路径。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被创建。

COPY 指令将会从构建的上下文目录中，把源路径的文件或目录复制到新的一层的镜像内的 <目标路径> 位置。比如：

```
COPY qf.json /usr/src/app/
```

注意下面是错误的

```
COPY qf.json /usr/src/app
```

这样会把 qf.json 拷贝成为 /usr/src/ 目录下的 app 文件

<源路径> 可以是多个，支持通配符，如：

```
COPY qf* /app/  
COPY q?.txt /app/
```

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。

## ADD 指令

`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。

支持自动解压缩，压缩格式支持：`gzip`, `bzip2` 以及 `xz`

官方推荐使用 `COPY` 进行文件的复制。

`ADD` 指定会使构建镜像时的缓存失效，导致构建镜像的速度很慢。

`COPY` 和 `ADD` 指令中选择的原则，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`。

```
ADD qf.tar.gz /
```

## USER 指令

`USER` 则是改变之后层的执行 `RUN`, `CMD` 以及 `ENTRYPOINT` 这类命令的身份。

这个用户必须是事先建立好的，否则无法切换。

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 健康检查指令

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。

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，用其启动容器后的状态变化会是下面的演变过程：

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。
- `--start-period=<时长>`：容器的初始化时长，默认0秒，不计入健康检测时间内。

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

后面的命令同样支持 `shell` 方式和 `exec` 方式。

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。

示例：

使用 `curl` 命令来判断 nginx 提供的 web 服务是否正常。

其 `Dockerfile` 的 `HEALTHCHECK` 可以这么写：

```
FROM centos
RUN rpm -ivh \
http://mirrors.aliyun.com/epel/epel-release-
latest-7.noarch.rpm && yum install nginx \
curl -y
ADD index.html
/usr/share/nginx/html/index.html
HEALTHCHECK --interval=5s --timeout=3s CMD
curl -fs \
http://localhost/ || exit 1
CMD ["nginx", "-g", "daemon off;"]
EXPOSE 80
```

这里设置了每 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。

构建镜像后，启动容器，并观察容器的状态变化

```
# docker build -t ali_nginx .
# docker run -d ali_nginx
```

```
[root@localhost ~]# docker ps
CONTAINER ID IMAGE COMMAND
CREATED STATUS
PORTS NAMES
09a8b90b0f67 ali_nginx "nginx -g 'daemon
of..." 4 seconds ago Up 3 seconds
(health: starting) 80/tcp vigorous_jang
[root@localhost ~]# docker ps
CONTAINER ID IMAGE
COMMAND CREATED
STATUS PORTS
NAMES
09a8b90b0f67 ali_nginx "nginx
-g 'daemon of..." 19 seconds ago Up 18
seconds (healthy) 80/tcp
vigorous_jang
```

## 利用元数据查看容器的健康状态

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

## ONBILUD 指令

`ONBILUD` 指令用于当其他 Dockerfile 以自己为基础镜像时将会运行的命令。

格式：`ONBUILD <其它指令>`。

其他指令可以是：比如 `RUN`, `COPY` 等。

## 基础应用场景

假如有两个项目 A 和 B

两个项目想分别有不同的文件

**A 项目下的文件：**

```
[root@docker onbulid]# tree A/
A/
├── a1.txt
└── a.txt

0 directories, 2 files
```

**B 项目下的文件**

```
[root@docker onbulid]# tree B
B
├── b1.txt
├── b2.txt
└── b.txt

0 directories, 3 files
```

现在任意的空目录下创建一个 `Dockerfile`

文件内容：

```
[root@docker onbulid]# cat Dockerfile
FROM alpine
ONBUILD COPY . /root/
```

接着用这个 `Dockerfile` 来构建一个所有项目都要使用的一个基础镜像

镜像名字： `alpine-base`

```
[root@docker onbulid]# docker build -t alpine-
base .

Sending build context to Docker daemon
6.144kB

Step 1/2 : FROM alpine
--> 3fd9065eaf02

Step 2/2 : ONBUILD COPY . /root/
--> Running in 4d6fad2809be
Removing intermediate container 4d6fad2809be
--> 804bfc0b47be

Successfully built 804bfc0b47be
Successfully tagged alpine-base:latest
```

当使用这个镜像去运行容器的时候。查看 `/root` 目录下，可发现并没有任何东西，

说明 `COPY . /root/` 并没有此次构建镜像的过程中去执行。

```
[root@docker onbulid]# docker run --rm alpine-
base:latest ls /root/
[root@docker onbulid]#
```

现在我们在使用刚才构建的镜像为项目 A 的基础镜像，来构建 A 项目的镜像

想看看目前 A 项目下的文件：

```
[root@docker onbulid]# cd A  
[root@docker A]# ls  
a1.txt  a.txt
```

在项目的 A 目录下编写 Dockerfile 文件内容如下：

```
[root@docker A]# cat Dockerfile  
FROM alpine-base:latest
```

是的只需要这一行即可

现在让我们来构建 A 项目的镜像

```
[root@docker A]# docker build -t alpine-a .  
Sending build context to Docker daemon  
3.072kB  
Step 1/1 : FROM alpine-base:latest  
# Executing 1 build trigger  
--> 5a003e1dc65f  
Successfully built 5a003e1dc65f  
Successfully tagged alpine-a:latest
```

接着运行以这个镜像 alpine-a:latest 为基础镜像而运行的容器中的 /root/ 目录下会有 A 项目目录下的所有文件：

```
[root@docker A]# docker run --rm alpine-a:latest ls /root/
Dockerfile
a.txt
a1.txt
```

B 项目的 `Dockerfile` 的内容：

```
[root@docker B]# ls
b1.txt  b2.txt  b.txt
[root@docker B]# cat Dockerfile
FROM alpine-base:latest
```

同样构建 B 项目的镜像，运行容器后可以看到 `/root/` 目录下会有 B 项目目录下的所有文件

```
[root@docker B]# docker build -t alpine-b .
Sending build context to Docker daemon
3.584kB
Step 1/1 : FROM alpine-base:latest
# Executing 1 build trigger
---> e66b6ee561a9
Successfully built e66b6ee561a9
Successfully tagged alpine-b:latest
[root@docker B]# docker run --rm alpine-
b:latest ls /root/
Dockerfile
b.txt
b1.txt
b2.txt
```

可以看出，`ONBUILD` 指令后面内容会在，其他镜像以此镜像为基础镜像构建的时候执行。

## 高级应用场景

python 项目都有自己的依赖包，通常会放在项目根目录下的一个文件，这个文件名叫：`requirements.txt`

此文件可以通过如下命令得到：

```
(django) [root@localhost ~]# pip3 freeze >
requirement.txt
```

内容一般为：

```
(django) [root@localhost ~]# head -3  
requirement.txt  
Django==1.11  
PyMySQL==0.8.1
```

可以使用如下命令来安装这些项目的依赖模块。

```
pip3 install -r requirement.txt
```

现在假设公司有多个 python3 的项目，每个项目都有自己不同的依赖模块。需要为每个项目制定一个 Dockerfile 或者镜像吗？

比如有两个项目： CMDB 和 SUPERMAN

下面我们使用 `ONBUILD` 指令来构建一个基础 `python` 镜像，

之后两个项目可以不必修改原来的 `Dockerfile` 就可以部署自己的环境依赖包了。

## CMDB 的 Dockerfile

### CMDB

```
[root@docker onbulid]# tree CMDB/
CMDB/
├── requirments.txt
└── run.py
[root@docker onbulid]# cat
CMDB/requirmants.txt
django==1.11
[root@docker onbulid]# cat CMDB/run.py
import django
print(django.VERSION
```

## SUPERMAN

```
[root@docker onbulid]# tree SUPERMAN/
SUPERMAN/
├── requirments.txt
└── run.py
[root@docker onbulid]# cat
SUPERMAN/requirmants.txt
django==1.11
[root@docker onbulid]# cat SUPERMAN/run.py
import django
print(django.VERSION
```

## 使用 ONBUILD 指令构建 Python 基础镜像

```
[root@docker onbulid]# cat Dockerfile
FROM python
ONBUILD COPY . /opt/
ONBUILD RUN pip3 install -r
/opt/requirements.txt
ONBUILD CMD ["python3", "/opt/run.py"]
[root@docker onbulid]# docker build -t
python3-base .
```

之后分别在各自的项目目录下创建自己的 `Dockerfile`

## CMDB 的 `Dockerfile`

```
FROM python3-base
```

## SUPERMAN 的 `Dockerfile`

```
FROM python3-base
```

这样就可以很简单的实现不同的项目只需要创建一个同样内容的镜像，而会得到自己的环境了。

另外下面的是在 shell 中的执行 python 的命令：

```
FROM python
ONBUILD COPY ./requirement.txt /
ONBUILD RUN pip install -r /requirement.txt
ONBUILD CMD ["python", "-c" "import
django;print(django.VERSION)"]
```

把这个构建成所有项目的基础镜像，名字为： python-onbuild:v1.0

```
# docker build -t python-onbuild:v1.0 .
```

其他 `python` 项目再使用此镜像为基础镜像时，`Dockerfile` 中只需一行即可：

```
FROM python-onbuild:v1.0
```

**更多参考官方 Docker Demo 和官网**

1. doker demo

<https://github.com/docker-library>

1. 官网

<https://docs.docker.com/engine/reference/builder/>

[https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)

## 镜像发布

### 直接把镜像推到 docker hub 上

1. 在自己的本地宿主机上登录你的 docker hub 账号

```
[root@localhost hello_qf]# docker login
Login with your Docker ID to push and pull
images from Docker Hub. If you don't have a
Docker ID, head over to https://hub.docker.com
to create one.

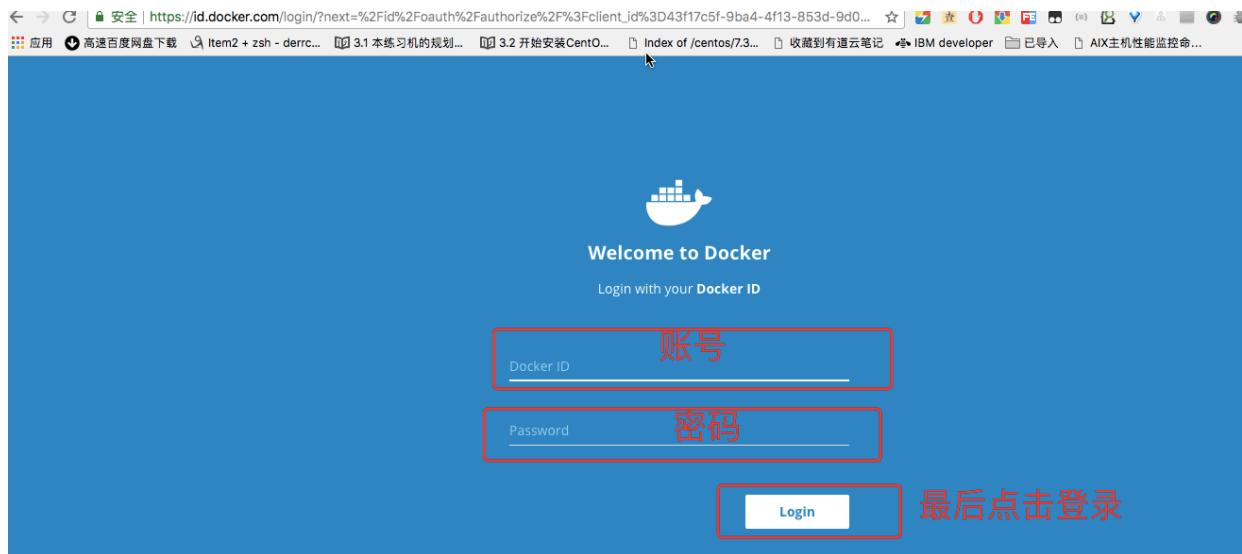
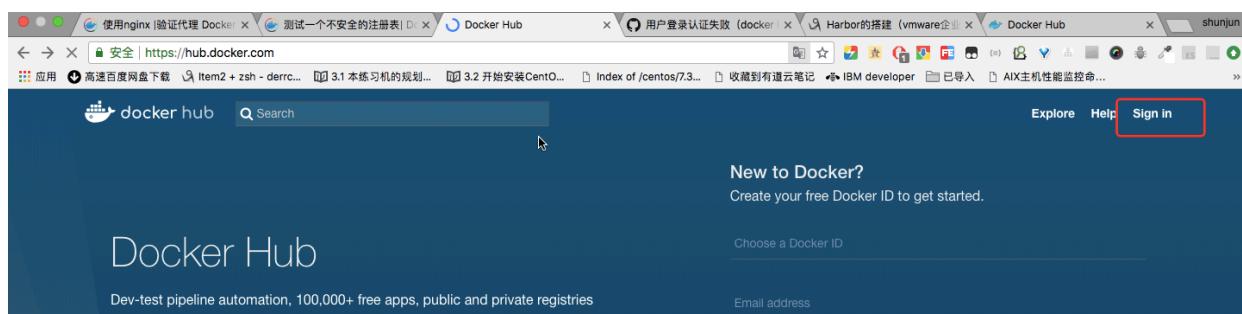
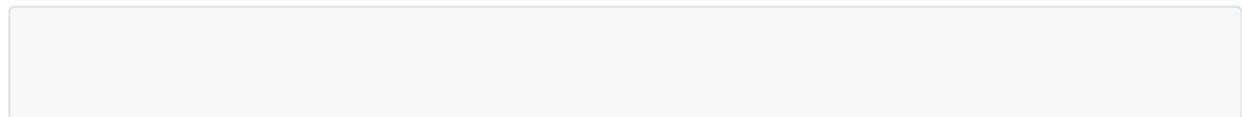
Username: test          # 输入你的 docker id
Password:
Login Succeeded        # 表示成功登录
[root@localhost hello_qf]#
```

假如你需要退出登录时

```
[root@localhost ~]# docker logout
```

1. 查看本地仓库的镜像

## 2. 修改标签



Docker Hub Dashboard showing repositories for user 'xiguatian'. The repository 'xiguatian/alpine' is highlighted with a red border.

Repository	Description	Stars	Pulls
xiguatian/hello-yange	public	0 STARS	2 PULLS
xiguatian/dockerfiles	public   automated build	0 STARS	1 PULLS
xiguatian/alpine	public	0 STARS	1 PULLS

Docker Hub Repository page for 'xiguatian/alpine'. The 'Docker Pull Command' field is highlighted with a red border and labeled '拉取命令' (Pull Command).

PUBLIC REPOSITORY  
xiguatian/alpine ☆  
Last pushed: 4 minutes ago

Repo Info Tags Collaborators Webhooks Settings

Short Description  
Short description is empty for this repo.

Full Description  
Full description is empty for this repo.

Docker Pull Command  
docker pull xiguatian/alpine

Owner  
xiguatian

```
[root@docker docker_nginx]# docker rmi
xiguatian/alpine:v1.0
Untagged: xiguatian/alpine:v1.0
Untagged:
xiguatian/alpine@sha256:8c03bb07a531c53ad7d0f6
e7041b64d81f99c6e493cb39abba56d956b40eacbc
[root@docker docker_nginx]# docker image ls
xiguatian/alpine:v1.0
REPOSITORY          TAG      IMAGE
ID                 CREATED   SIZE
[root@docker docker_nginx]# docker pull
xiguatian/alpine
Using default tag: latest
Error response from daemon: manifest for
xiguatian/alpine:latest not found
```

## 需要指定 tag

The screenshot shows the Docker Hub website at <https://hub.docker.com/r/xiguatian/alpine/tags/>. The page displays the 'PUBLIC REPOSITORY' for 'xiguatian/alpine'. The 'Tags' tab is active and highlighted with a red box. A table below lists the tags, showing 'v1.0' as the current selection.

Tag Name	Compressed Size	Last Updated
v1.0	2 MB	6 minutes ago

语法 docker push 自己的 docker id/镜像名[:TAG]

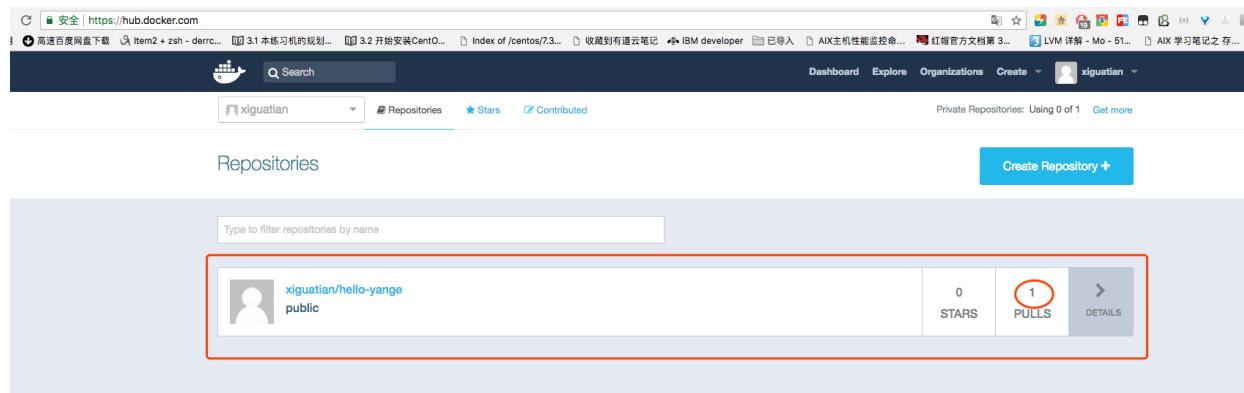
```
[root@docker docker_nginx]# docker pull
xiguatian/alpine:v1.0
v1.0: Pulling from xiguatian/alpine
Digest:
sha256:8c03bb07a531c53ad7d0f6e7041b64d81f99c6e
493cb39abba56d956b40eacbc
Status: Downloaded newer image for
xiguatian/alpine:v1.0
[root@docker docker_nginx]# docker image ls
xiguatian/alpine:v1.0
REPOSITORY          TAG      IMAGE
ID                CREATED    SIZE
xiguatian/alpine   v1.0
3fd9065eaf02       4 months ago  4.15MB
```

```
[root@localhost hello_qf]# docker image ls
REPOSITORY          TAG      IMAGE
IMAGE ID            CREATED    SIZE
gcc                latest
7c6c8b9d6e2f       3 hours ago  8.55kB
xiguatian/hello-yange  latest
dfadd4a86525       5 hours ago  865kB
```

2. 开始把需要发布的镜像，发布到 Docker Hub 自己的仓库

```
[root@localhost hello_qf]# docker push
xiguatian/hello-yangge
The push refers to repository
[docker.io/xiguatian/hello-yangge]
1313930c66a7: Pushed
latest: digest:
sha256:245c1193222eced00346fed63b6a4f9e270c82a
741f01384f860c95d84bb840e size: 527
[root@localhost hello_qf]#
```

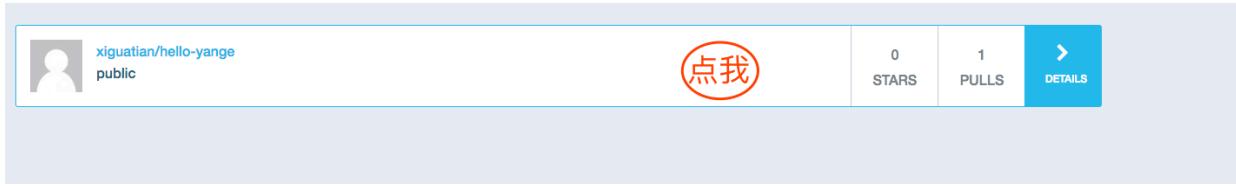
## 1. 登录到 doker hub 上查看



## 1. 把本地从库的这个镜像删除

```
[root@localhost hello_qf]# docker rmi
xiguatian/hello-yangge
```

## 1. 把 docker hub 上的进行 pull 下来



Short Description

Short description is empty for this repo.

Full Description

Full description is empty for this repo.

Docker Pull Command

docker pull xiguatian/hello-yange

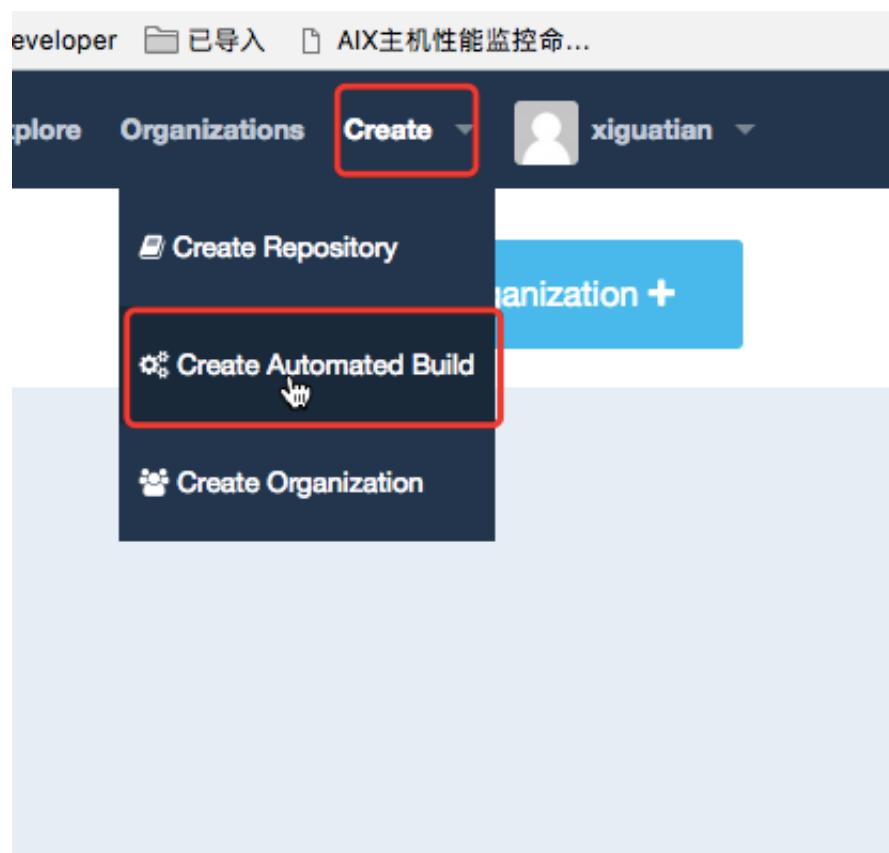
赋值此行到本地执行

Owner

xiguatian

```
[root@localhost hello_qf]# docker pull
xiguatian/hello-yange
Using default tag: latest
latest: Pulling from xiguatian/hello-yange
cf11dfb04e74: Pull complete
Digest:
sha256:245c1193222eced00346fed63b6a4f9e270c82a
741f01384f860c95d84bb840e
Status: Downloaded newer image for
xiguatian/hello-yange:latest
[root@localhost hello_qf]# docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
xiguatian/hello-yange    latest    dfadd4a86525  6 hours ago  865kB
```

## 把 Dockerfile 推到 docker hub (推荐)



You haven't linked to  GitHub or  Bitbucket yet.

[Link Accounts](#)

## Linked Accounts & Services

### Linked Accounts

These account links are currently used for Automated Builds, so that we can access your project lists and help you configure your Automated Builds. **Please note:** A github/bitbucket account can be connected to only one docker hub account at a time.

点我 

Link Github



Link Bitbucket

### 连接到GitHub

我们让您选择我们对您的GitHub帐户有多少访问权限。

#### 公共和私人 (推荐)

- 读取和写入访问公共和私人存储库。（我们只使用写访问来添加服务钩子并添加部署密钥）
- 如果您想从私人GitHub存储库设置自动构建，那么这是必需的。
- 如果您想使用私人GitHub组织，那么这是必需的。
- 我们将自动为您配置服务挂钩和部署密钥。

选择

#### 有限访问

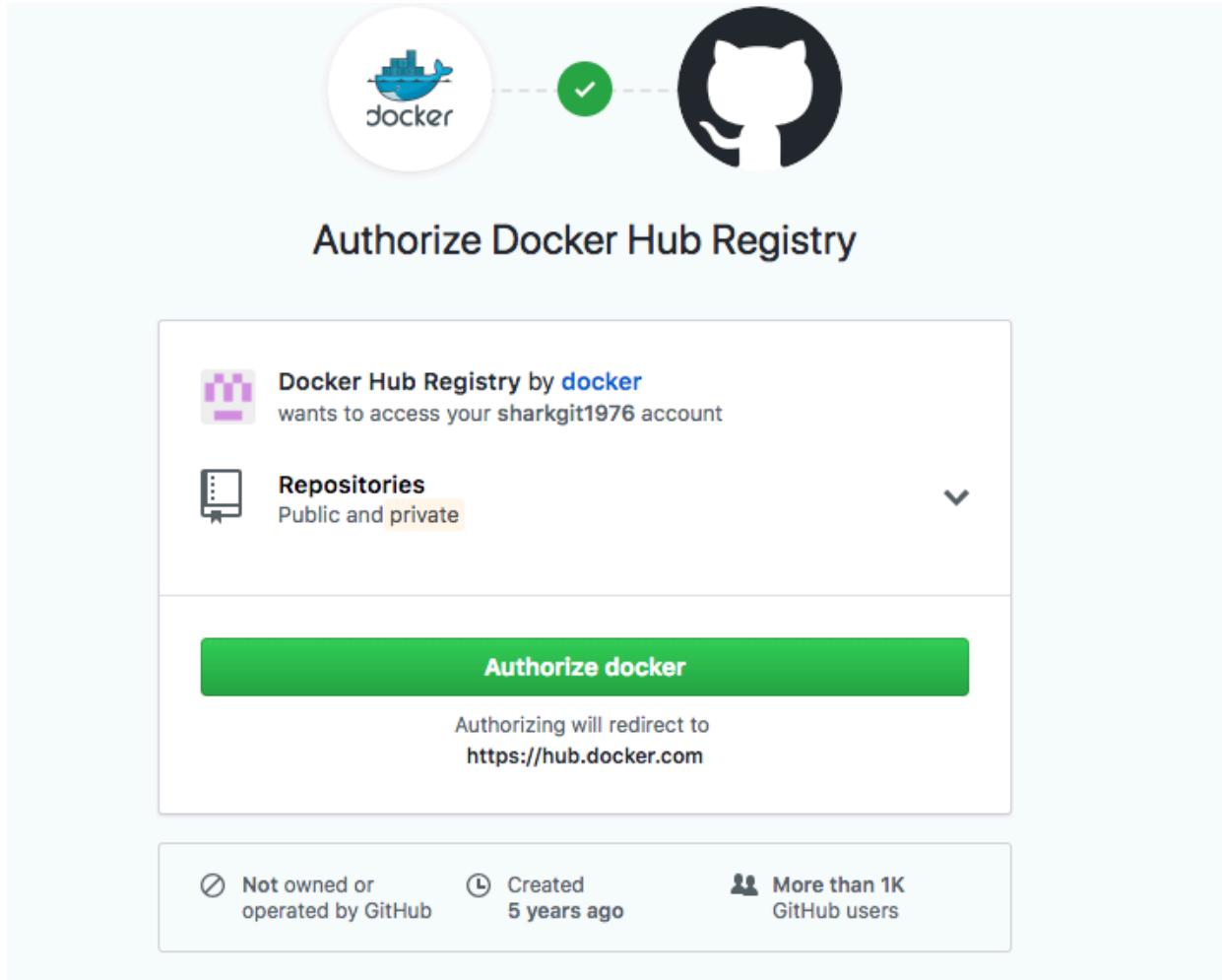
- 公开只读访问。
- 只适用于公共存储库和组织。
- 您需要手动更改存储库才能使用自动构建。

选择

中文版：



英文版：



关联帐户和服务

关联账户

这些帐户链接目前用于自动构建，以便我们可以访问您的项目列表并帮助您配置自动构建。请注意：github / bitbucket 账户一次只能连接到一个码头中心账户。

**成功**

读/写访问

取消链接Github

链接Bitbucket

New repository

Import repository

New gist

New organization

Followers 0 Following 1

Customize your pinned repositories

## Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: sharkgit1976

Repository name: **仓库名称** dockerfile ✓

Great repository names are short and memorable. Need inspiration? How about [musical-invention](#).

Description (optional): **描述信息** 存放 Dockerfile

**Public**  
Anyone can see this repository. You choose who can commit.

**Private**  
You choose who can see and commit to this repository.

Initialize this repository with a README  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None | ⓘ

**点击创建** Create repository

安装下图提示，在本地创建一个新的 git 仓库，用于和这个仓库进行关联同步

按图提示点击，以复制多行命名

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH git@github.com:sharkgit1976/dockerfile.git

We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# dockerfile" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git remote add origin git@github.com:sharkgit1976/dockerfile.git  
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin git@github.com:sharkgit1976/dockerfile.git  
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

这里是在本地创建一个新仓库

点击这里

A red arrow points to the 'Create repository' button (a folder icon) in the top right corner of the command-line section.

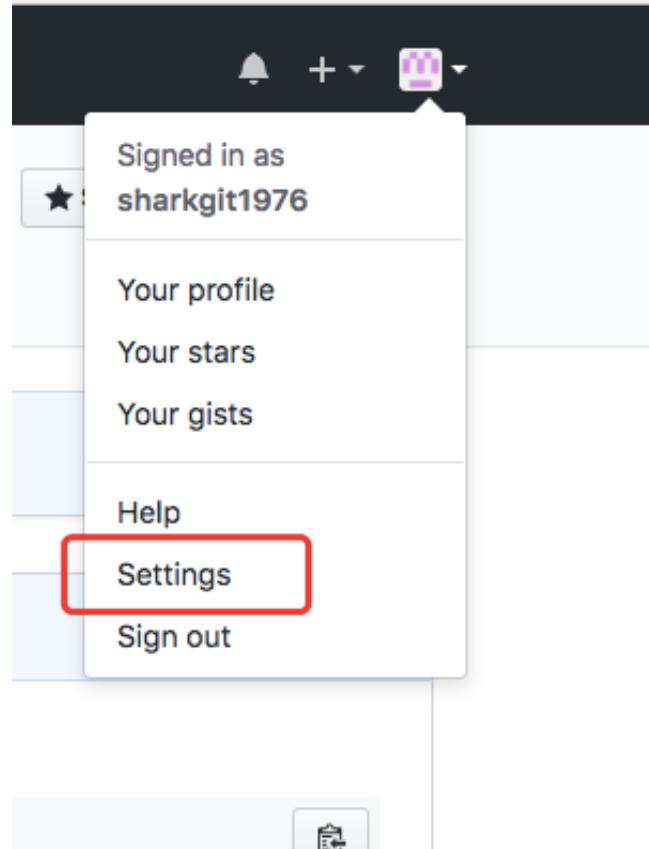
在本地选择一个目录，作为本地的 git 仓库，之后进入这个目录。

依次执行刚才复制的命令

```
echo "# dockerfile" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git remote add origin git@github.com:你的git账号/dockerfile.git  
git push -u origin master
```

最后一般会提示，权限问题。

可以把本地当前用户的公钥传到 git hub 上，以建立信任关系



1

Personal settings

Profile

Account

Emails

Notifications

Billing

SSH and GPG keys

Security

Blocked users

Repositories

Organizations

Saved replies

Applications

Developer settings

2

## SSH keys

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

My mac	Fingerprint: f7:7f:4c:b5:d1:3d:db:06:af:a9:cf:97:2a:f4:61:30	<button>Delete</button>
SSH	Added on 18 Oct 2017	
	Last used within the last 5 months — Read/write	
testing	Fingerprint: a4:55:86:43:be:ac:7c:e2:08:66:79:87:cd:61:c8:ce	<button>Delete</button>
SSH	Added on 29 May 2018	
	Last used within the last week — Read/write	

New SSH key

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#).

## GPG keys

New GPG key

There are no GPG keys associated with your account.

Learn how to [generate a GPG key and add it to your account](#).

Personal settings

Profile

Account

Emails

Notifications

Billing

**SSH and GPG keys**

Security

Blocked users

Repositories

Organizations

Saved replies

Applications

**SSH keys / Add new**

Title  
test **这里为这个公钥起个名字**

Key  
Begins with 'ssh-rsa', 'ssh-dss', 'ssh-ed25519', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'

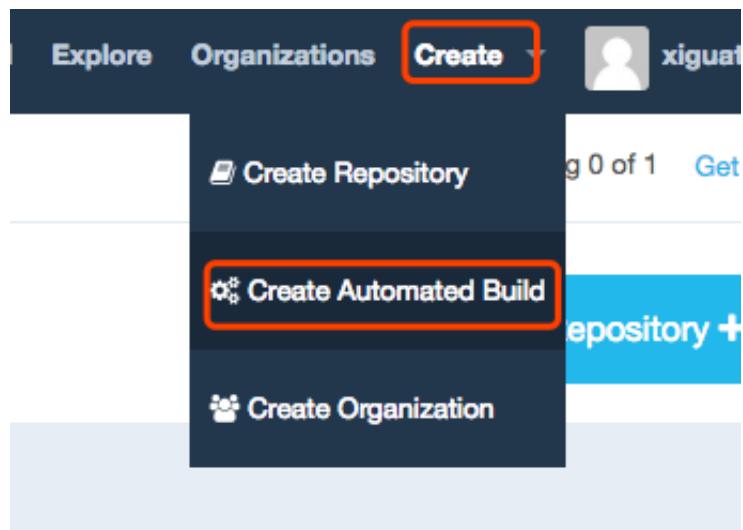
**把本地的公钥粘贴到此处**

**Add SSH key** 最后点击添加

重新执行最后一条命令：

```
git push -u origin master
```

在 github 上创建完仓库后，再来到 docker hub 上进行设置





这个就是刚才在 git hub 上创建的仓库名

dockerfiles 点我

Repository Namespace & Name\*

Visibility

Short Description\*

填写描述信息

By default Automated Builds will match branch names to Docker build tags. [Click here to customize behavior.](#)

Create

- 填写该auto-build的名称（名称任意自取）

- 填写short description
- 点击create

下图表示成功

The screenshot shows the Docker Hub repository creation form. It includes fields for Repository Namespace & Name (xiguatian/dockerfiles), Visibility (public), and Short Description (填写描述信息). A note at the bottom states: "By default Automated Builds will match branch names to Docker build tags. [Click here to customize behavior.](#)". A green success message box contains the text: "Successfully configured an automated build repository." A blue "Create" button is visible on the right.

**注意：只有github中内容发生变化时， dockerhub才会进行 auto-build**

在 git hub 上改变 Dockerfile 后，在 docker hub 上可以看到会正在自动构建

The screenshot shows the Docker Hub repository details page for the 'xiguatian/dockerfiles' repository. The 'Build Details' tab is selected. A table displays the build status:

Status	Actions	Tag	Created	Last Updated
Building	<a href="#">Cancel</a>	latest	a few seconds ago	a few seconds ago

# 针对 Docker hub 上的多个仓库映射到 Git hub 上被关联仓库下的不同目录

## 示例

1. 在本地 git 仓库的目录下创建相应的目录和 Dockerfile

```
[root@docker docker_git]# tree
```

```
.
```

```
+- alpine
```

```
  +- v1.1
```

```
    \-- Dockerfile
```

```
  +- v1.2
```

```
    \-- Dockerfile
```

```
+- nginx
```

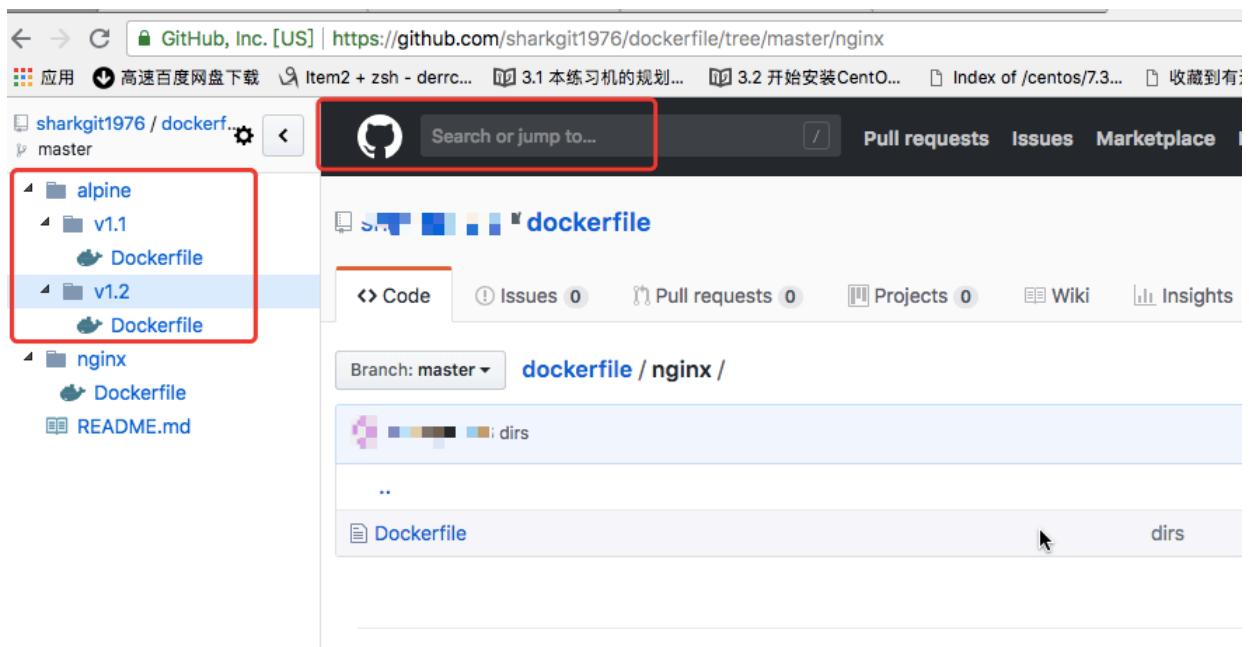
```
  \-- Dockerfile
```

```
+- README.md
```

2. 同步本地 git 仓库的文件到远程仓库

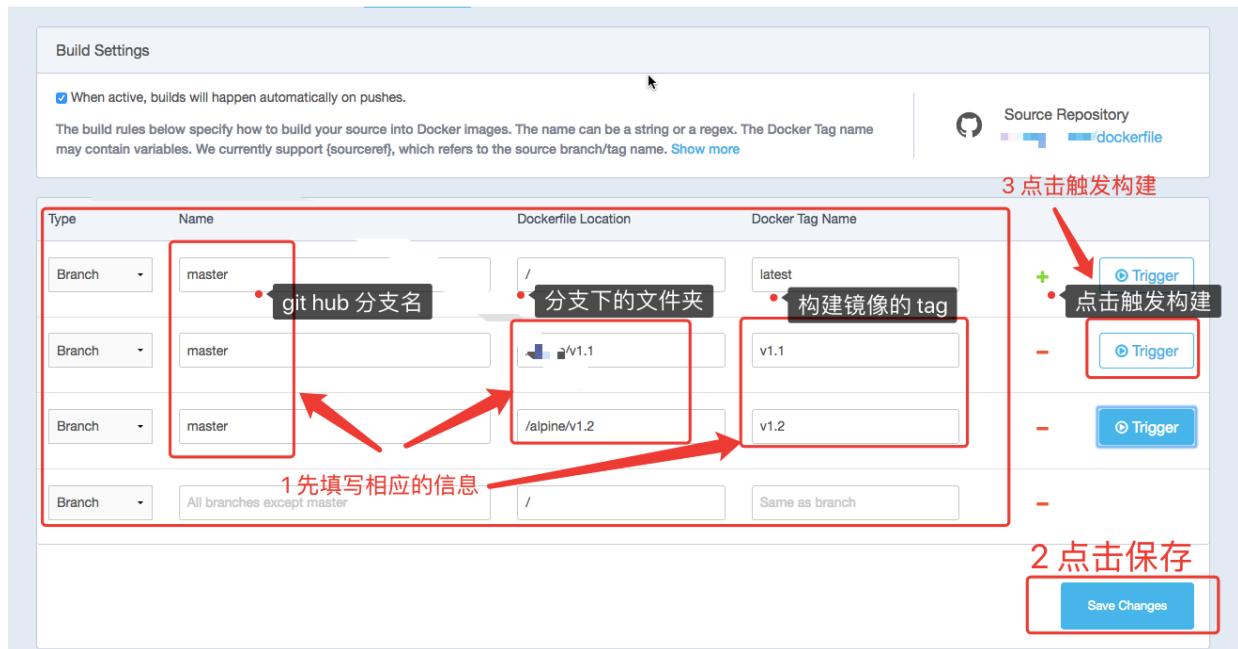
```
[root@docker docker_git]# git add --all
[root@docker docker_git]# git commit -m "dd"
[master 5c44ad5] dd
3 files changed, 3 insertions(+), 1 deletion(-)
delete mode 100644 Dockerfile
rename alpine/{ => v1.1}/Dockerfile (100%)
create mode 100644 alpine/v1.2/Dockerfile
[root@docker docker_git]# git push origin master
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 477 bytes | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@github.com:s.../.../.../dockerfile.git
 4c36521..5c44ad5  master -> master
```

### 3. 检查 git hub 仓库是否有同步的文件目录



### 4. 在 Docker hub 上配置监控 git hub 仓库的不同目录，并在

## 配置保存后，点击触发构建



## 5. 在 Docker hub 上检查自动构建详情

This screenshot shows the 'AUTOMATED BUILD' section for a repository on Docker Hub. It displays the last pushed image and a table of build logs.

**Repo Info**, **Tags**, **Dockerfile**, **Build Details**, **Build Settings**, **Collaborators**, **Webhooks**, **Settings**

Status	Actions	Tag	Created	Last Updated
queued	<button>Cancel</button>	v1.2	a few seconds ago	a few seconds ago
queued	<button>Cancel</button>	v1.1	a few seconds ago	a few seconds ago

**Source Repository**: /dockerfile

## 6. 等待几秒后，刷新网页，看到结果

PUBLIC | AUTOMATED BUILD

/dockerfile

Last pushed: 10 minutes ago

Repo Info Tags Dockerfile Build Details Build Settings Collaborators Webhooks Settings

Status	Actions	Tag	Created	Last Updated
✓ Success	自动构建成功	v1.2	15 minutes ago	9 minutes ago
✓ Success		v1.1	15 minutes ago	12 minutes ago

Source Repository

/dockerfile

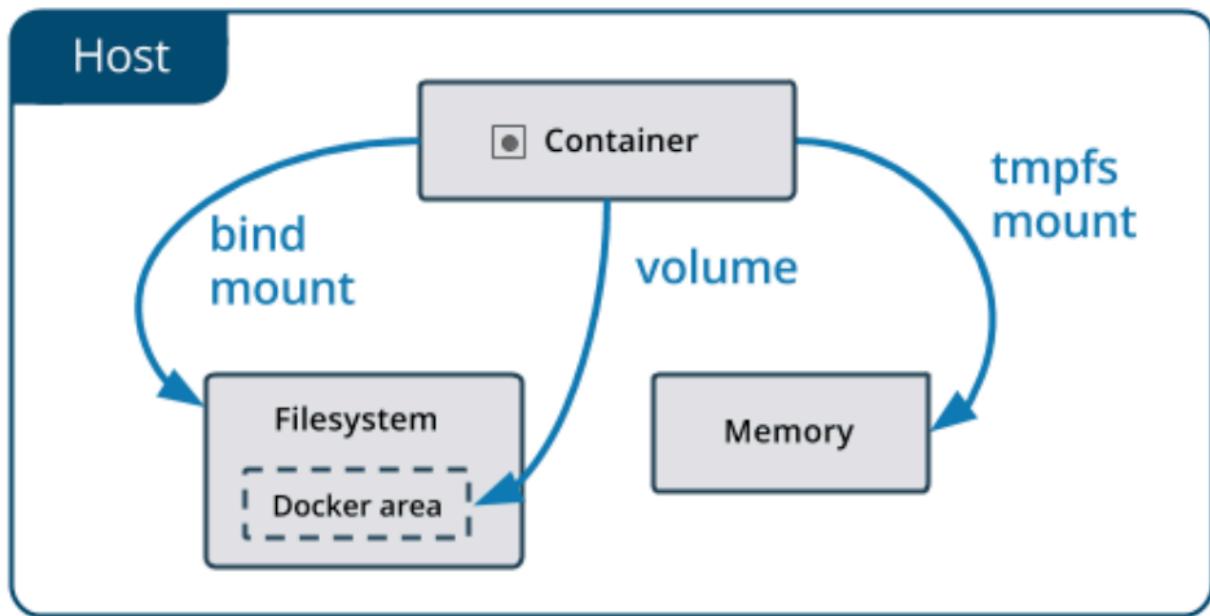
## 管理 Docker 中的数据

默认情况下，容器内创建的所有文件都存储在可写容器层上。这意味着：

- 当容器不再运行时，数据不会持续存在，并且如果另一个进程需要数据，则很难从容器中获取数据。
- 容器的可写层紧密耦合到容器运行的主机。您无法轻松地将数据移到其他地方。
- 写入容器的可写层需要 [存储驱动程序](#) 来管理文件系统。存储驱动程序使用 Linux 内核提供联合文件系统。与使用直接写入主机文件系统的 [数据卷](#) 相比，这种额外的抽象性能会降低性能。

Docker 有两种容器将文件存储在主机中的选项，这样即使在容器停止之后这些文件也会被保留：[卷](#) 和 [绑定挂载](#)。如果你在 Linux 上运行 Docker，你也可以使用 [tmpfs](#) 挂载。

# 三者的相同点和区别



## 相同之处

无论您选择使用哪种类型去使用，数据在容器内看起来都是相同的。它被视为容器文件系统中的目录或单个文件。

## 不同之处

- **卷 (volume)** 存储在由Docker管理的主机文件系统的一部分中（在Linux上是：`/var/lib/docker/volumes/`）。非Docker进程不应该修改这部分文件系统。卷是在Docker中保留数据的最佳方式。
- **绑定挂载 (bind mount)** 也就是把主机的本地目录挂载到容器中某个挂载点。可以存储在主机系统的任何位置。他们甚至可能是重要的系统文件或目录。Docker主机或Docker容器上的非Docker进程可以随时修改它们。
- **tmpfs挂载 (tmpfs mount)** 仅存储在主机系统的内存中，而不会写入主机系统的文件系统。

## `-v` 还是 `--mount`

使用 `-v` or `--volume` 标志可将绑定挂载和卷挂载到容器中

对于 `tmpfs` 可以使用 `--tmpfs`

在Docker 17.06及更高版本中，官方建议将 `--mount` 容器和服务的标志用于绑定挂载，卷或 `tmpfs` 挂载，因为语法更清晰。

## 关于使用数据卷和挂载主机目录的提示

- 如果将空卷挂载到容器中的含有内容的目录中，则会将这些内容复制到卷中。同样，如果您启动容器并指定一个尚不存在的卷，则会为您创建一个空卷。
- 如果将一个**bind mount** 或非空的数据卷挂载到容器中的一个非空目录中，则这些内容会被遮盖隐藏。隐藏的内容不会被删除或更改，此时也不可被访问。就像在 Linux 机器中使用 `mount` 命令一样的效果

## 数据卷（volume）详解

由Docker创建和管理。

可以使用该 `docker volume create` 命令显式创建一个卷，或者在创建容器或服务期间Docker可以创建一个卷。这与绑定挂载的工作方式类似，区别在于卷由Docker管理。

卷还支持使用卷驱动程序，这些卷驱动程序可让您将数据存储在远程主机或云提供程序中，以及其他可能性。

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**。

## volume 的使用场景

卷是在Docker容器和服务中保持数据的首选方式。卷的一些用例包括：

- 在多个运行容器之间共享数据。如果您没有明确创建它，则会在第一次将其装入容器时创建卷。当该容器停止或被移除时，该卷仍然存在。多个容器可以同时安装相同的卷，无论是读写还是只读。仅当您明确删除卷时才会删除卷。
- 当您想要将容器的数据存储在远程主机或云提供商上而不是本地时。
- 当您需要备份，还原或将数据从一台Docker主机迁移到另一台时，卷是更好的选择。您可以使用卷停止容器，然后备

份卷的目录（如 `/var/lib/docker/volumes/<volume-name>`）。

## volume 基本使用

### 实验场景一：

1. 创建一个新的数据卷，之后运行一个容器，并把数据卷挂载到容器的一个目录 `/webapp` 下。
2. 在容器的目录 `/webapp` 下创建文件 `nginx.txt` 并写入一些内容
3. 接着退出这个容器，并删除。
4. 运行一个新的容器，并挂载刚才的数据卷到容器的目录 `/yangge`
5. 观察容器目录 `/yangge` 下是否有之前创建的文件 `nginx.txt`

### --mount 语法：

`--mount type=volume, source=数据卷名, target=容器中的挂载点`

`type` 的值可以是 `volume`, `bind`, `tmpfs` 默认是 `volume`

`source` 也可以写成 `src`

`target` 可以写成 `dst`

## 创建一个数据卷

```
$ docker volume create yangge_vol
```

## 启动一个容器并挂载一个已创建好的数据卷

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。

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

```
[root@docker data_volume]# docker run -it --  
mount source=yangge_vol,target=/webapp --name  
mynginx nginx /bin/sh
```

## 进入容器后，创建文件

此时在容器的 `/webapp` 目录下创建文件 `nginx.txt`

```
/ # touch /webapp/nginx.txt  
/ # echo "nginx server" > /webapp/nginx.txt
```

## 退出容器并删除容器

```
/ # exit  
[root@docker data_volume]# docker rm mynginx
```

启动一个新的容器，并且把刚才的数据卷 `yangge_vol` 挂载到新的容器里

```
[root@docker data_volume]# docker run -it --  
name nginx_new \  
> --mount source=yangge_vol,target=/yangge \  
> alpine \  
> /bin/sh
```

接着查看容器中目录 `/yangge` 下是否有个刚才的文件

`nginx.txt`

```
/ # cat /yangge/nginx.txt  
nginx server
```

## 实验场景二：（作业）

1. 直接运行一个容器，并且挂载两个数据卷到容器中。

- 数据卷一：`qf_vol` 目标挂载点：`/qf_data`
- 数据卷二：`shark_vol` 目标挂载点：`/shark_data`

2. 之后分别在容器的两个目录创建一些文件。

3. 退出容器，并删除容器
4. 把其中的一个数据卷挂载到一个新运行的容器。
5. 观察原来建立的数据是否还存在

## bind mount 详解

就是挂载主机目录

与卷相比，绑定安装具有有限的功能。

当您使用绑定挂载时，主机上的文件或目录被挂载到容器中。  
主机上的文件或目录必须是完整路径。

不存在，它会根据需求被创建。

您不能使用Docker CLI命令直接管理被挂载的目录或文件。

会非常高效，但这点，依赖于具有特定目录结构的主机的文件系统。

**副作用**是您可以通过容器中运行的进程更改主机文件系统，包括创建，修改或删除重要的系统文件或目录。这是一个强大的能力，可能会对安全产生影响，包括影响主机系统上的非Docker进程。

## bind mount 的使用场景

- 从主机共享配置文件到容器。这是默认情况下，通过`/etc/resolv.conf`从主机挂载到每个容器，Docker如何为容器提供DNS解析。
- 在Docker主机上的开发环境与容器之间共享源码。
- 当需要保证Docker主机的文件或目录结构与容器所需的一致时。

## 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。

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

```
$ docker run -it \
  --name web \
  --mount
  type=bind,source=/tmp/,target=/opt/webapp/ \
  alpine \
  /bin/sh
```

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 -it \
  --name web \
  # -v /src/webapp:/opt/webapp:ro \
  --mount
type=bind,source=/tmp/,target=/opt/webapp/,rea
donly \
  alpine \
  /bin/sh
```

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

```
\# touch new.txt
touch: new.txt: Read-only file system
```

## 挂载 Docker 主机的本地文件到容器中

```
[root@docker ~]# docker run -it --rm \
--mount type=bind,source=/root/.bash_history,target=/root/.bash_history
alpine \
/bin/sh
/ # ls /root/.bash_history
/root/.bash_history
/ #
```

## 查看数据卷的具体信息

### 查看所有的 `数据卷`

```
$ docker volume ls
```

local	yangge_vol
-------	------------

在主机里使用以下命令可以查看指定 `数据卷` 的信息

```
[root@docker data_volume]# docker volume
inspect yangge_vol
[
  {
    "CreatedAt": "2018-06-
01T11:48:14+08:00",
    "Driver": "local",
    "Labels": {},
    "Mountpoint":
"/var/lib/docker/volumes/yangge_vol/_data",
    "Name": "yangge_vol",
    "Options": {},
    "Scope": "local"
  }
]
```

## 通过容器查看数据卷挂载的具体信息

在主机里使用以下命令可以查看 `nginx_new` 容器的信息

`数据卷` 信息在 "Mounts" Key 下面

```
[root@docker ~]# docker inspect nginx_new -f "
{{json .Mounts}}" |python -m json.tool
[
{
    "Destination": "/yangge",
    "Driver": "local",
    "Mode": "z",
    "Name": "yangge_vol",
    "Propagation": "",
    "RW": true,
    "Source":
    "/var/lib/docker/volumes/yangge_vol/_data",
    "Type": "volume"
}
]
```

## 删除数据卷

```
$ docker volume rm yangge_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。

数据卷可能会占据很多空间，可以使用以下命令清理掉没有容器使用的 数据卷

```
$ docker volume prune
```

## Docker 网络

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。

### 配置 `s` 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，可以在服务启动时改变它：

- `--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

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。

`brctl` 命令需要安装 `bridge-utils`

```
[root@docker ~]# yum install bridge-utils
[root@docker ~]# brctl show
bridge name bridge id          STP enabled
interfaces
br-592bf266d1de      8000.0242d53c8ae4    no
veth1357758
                           vetha2ea524
br-8f91b60157b9      8000.02421002c747    no
docker0      8000.0242b23bc676    no
vethcc2dfbe
                           vethe52debb
```

## 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。

当使用大写的 `-P` 标记时，Docker 会随机映射一个物理机的 `49000~49900` 之间的端口到内部容器开放的网络端口。

`-p` 则可以指定想要映射的物理机端口，并且，在一个指定端口上只可以绑定一个容器。

示例：

映射指定的本地 IP 和端口到容器端口

```
ip:hostPort:containerPort
```

```
[root@docker ~]# docker run -d -p  
10.18.42.174:8080:80 --name mynginx2  
nginx:latest
```

映射本地指定 IP 的任意端口到容器的一个端口，本地主机会自动分配一个端口

ip:::containerPort

```
[root@docker ~]# docker run -d -p  
10.18.42.174::80 --name mynginx3 nginx:latest
```

映射本机的所有的地址的指定端口到容器的指定端口

hostPort:containerPort

```
[root@docker ~]# docker run -d -p 8000:80  
nginx:latest
```

-p 标记可以多次使用来绑定多个端口

例如：

```
$ docker run -d \
  --name nginx110
  -p 5001:5000 \
  -p 3000:80 \
nginx
```

## 查看端口映射配置信息

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

格式：

```
docker port 容器名/容器 ID 容器的端口号
```

```
[root@docker ~]# docker port mynginx
80/tcp -> 0.0.0.0:32768
```

```
[root@docker ~]# docker port nginx110
5000/tcp -> 0.0.0.0:5001
80/tcp -> 0.0.0.0:3000
[root@docker ~]# docker port nginx110 80
0.0.0.0:3000
[root@docker ~]# docker port nginx110 5000
0.0.0.0:5001
```

# 同一台主机上的容器互联和隔离

## 容器默认支持互联

默认情况下，在同一台主机上的所有容器在网络层都会互相通信。

这是因为，创建的每一个容器都会把它加入到一个网桥上，即 `bridge`

```
[root@docker ~]# docker network ls
NETWORK ID          NAME
DRIVER              SCOPE
8f91b60157b9        auth_docker_default
bridge              local
dc242d2dfee5        bridge
bridge              local
77095181cedc        host
                      host
                      local
9793d4ec6df2        none
                      null
                      local
```

可以尝试在容器中互相 `ping` 对方的容器名，可以发现都是可以通信的。

## 在一个主机上隔离不同的容器

## 1. 先创建一个 **Docker** 的网络

```
$ docker network create -d bridge qf-net
```

- -d 是指定了网络类型是个网桥

```
[root@docker onbulid]# docker network create -d bridge qf-net
[root@docker onbulid]# docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
8f91b60157b9    auth_docker_default
bridge          local
dc242d2dfee5   bridge
bridge          local
77095181cedc   host      local
9793d4ec6df2   null      local
592bf266d1de   qf-net
bridge          local
```

## 2. 将容器加入到新的网络

打开一个终端，运行容器1

```
[root@docker ~]# docker run -it --name host1 \
--network qf-net alpine:latest /bin/sh
/ # ip a |grep inet
    inet 127.0.0.1/8 scope host lo
        inet 172.19.0.2/16 brd 172.19.255.255
            scope global eth0
```

再打开一个新的终端，运行容器2

```
[root@docker ~]# docker run -it --name host2 \
--network qf-net alpine:latest
/ # ip a |grep inet
    inet 127.0.0.1/8 scope host lo
        inet 172.19.0.3/16 brd 172.19.255.255
            scope global eth0
```

在容器 1 和容器 2 中尝试互相 `ping` 对方

容器 2 的操作

```
/ # ping host1
PING host1 (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: seq=0 ttl=64
time=0.102 ms
64 bytes from 172.19.0.2: seq=1 ttl=64
time=0.103 ms
```

## 容器 1 的操作

```
/ # ping host2
PING host2 (172.19.0.3): 56 data bytes
64 bytes from 172.19.0.3: seq=0 ttl=64
time=0.099 ms
64 bytes from 172.19.0.3: seq=1 ttl=64
time=0.145 ms
64 bytes from 172.19.0.3: seq=2 ttl=64
time=0.133 ms
```

## 配置 DNS

1. 配置全部容器的 DNS，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"
  ]
}
```

重启 docker server 服务

```
[root@docker ~]# systemctl restart docker
```

这样每次启动的容器 DNS 自动配置为 114.114.114.114 和 8.8.8.8。

确认

```
$ docker run -it --rm alpine cat /etc/resolv.conf
```

## 2. 配置单个容器的 DNS

--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。

```
[root@docker ~]# docker run --rm --dns=8.8.8.8 alpine cat /etc/resolv.conf
search localdomain qf.com
nameserver 8.8.8.8
```

注意：如果在容器启动时没有指定 --dns=IP\_ADDRESS 参数，Docker 会默认用主机上的 /etc/resolv.conf 来配置容器。

# 部署私用仓库

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](#) 是官方提供的工具，可以用于构建私有的镜像仓库。

基于 [docker-registry](#) v2.x 版本。

API <https://docs.docker.com/registry/spec/api/>

## 获取

```
[root@localhost ~]# docker search registry --  
limit 1  
[root@localhost ~]# docker pull registry
```

## 以容器方式运行

```
# docker run -d -p 5000:5000 --restart=always  
\  
--name registry-test registry
```

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 \
  --restart=always \
  -p 5000:5000 \
  --name registry-test \
  -v /opt/data/registry:/var/lib/registry \
  registry
```

还可以配置私有仓库内部的监听端口

```
# docker run -d \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:5001 \
  -p 5001:5001 \
  --name registry-test \
  registry
```

## 在私有仓库上传、搜索、下载镜像

仓库建好后，就可以使用 `docker push` 命令上传本地仓库的镜像到私有仓库了。

注意：

首先需要指定需要被推送镜像的标签为 `私有仓库地址:port/镜像名` 的格式

更改标签语法：

示例：

```
[root@localhost ~]# docker image ls alpine
REPOSITORY          TAG      IMAGE
ID                 CREATED   SIZE
alpine              3.7      4.15MB
3fd9065eaf02        4 months ago
[root@localhost ~]# docker tag alpine:3.7
127.0.0.1:5000/alpine:3.7
```

推送：

```
[root@localhost ~]# docker push
127.0.0.1:5000/alpine:3.7
The push refers to repository
[127.0.0.1:5000/alpine]
cd7100a72410: Pushed
...
```

用 `curl` 命令查看私有仓库的镜像

```
[root@localhost ~]# curl  
127.0.0.1:5000/v2/_catalog  
{"repositories":["alpine"]}
```

尝试删除本地仓库的镜像 127.0.0.1:5000/alpine:3.7

```
[root@localhost ~]# docker rmi  
127.0.0.1:5000/alpine:3.7  
[root@localhost ~]# docker image ls | grep  
alpine
```

从私有仓库拉取刚才推送的镜像

```
[root@localhost ~]# docker pull  
127.0.0.1:5000/alpine:3.7  
3.7: Pulling from alpine  
Digest:  
sha256:8c03bb07a531c53ad7d0f6e7041b64d81f99c6e  
493cb39abba56d956b40eacbc  
Status: Downloaded newer image for  
127.0.0.1:5000/alpine:3.7  
[root@localhost ~]# docker image ls | grep  
alpine  
127.0.0.1:5000/alpine    3.7  
3fd9065eaf02          4 months ago        4.15MB  
alpine                  3.7  
3fd9065eaf02          4 months ago        4.15MB
```

## 配置可以访问远程私有仓库的 Docker 主机

其实是要在访问远程仓库的 Docker 机器上配置

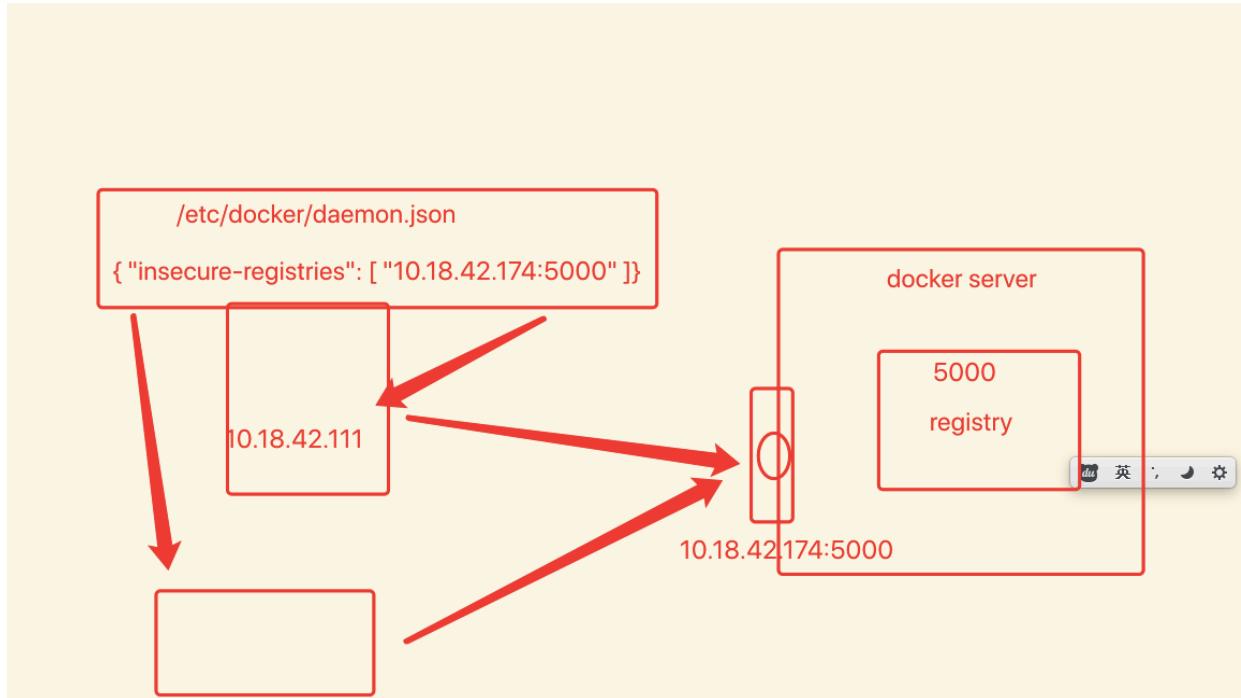
假如现在私用仓库部署在主机 172.16.153.136 上。

之后在其他 Docker 机器上配置如下内容

对于 Centos7.x 系统，在 /etc/docker/daemon.json 中写入如下内容：

```
{ "insecure-registries": [  
    "172.16.153.136:5000" ]}
```

```
[root@gougou ~]# cat /etc/docker/daemon.json  
{  
    "registry-mirrors": ["http://172.16.153.136:5000"],  
    "insecure-registries" : ["10.18.42.174:5000"]  
}
```



之后在 `/lib/systemd/system/docker.service` 添加如下内容

```
EnvironmentFile=/etc/docker/daemon.json
```

执行如下命令重启 Docker

```
[root@localhost ~]# systemctl daemon-reload  
[root@localhost ~]# systemctl restart docker
```

```
[root@gougou ~]# docker tag zhuzhu/hello-
yanmeili 10.18.42.174:5000/hello
[root@gougou ~]# docker push
10.18.42.174:5000/hello
The push refers to repository
[10.18.42.174:5000/hello]
4daac5e398bf: Pushed
latest: digest:
sha256:5efd8d38bc16299bb61d4096ebf64ce46a4600a
efda405a886e1b0cbc145922d size: 525
```