

centos7 docker CE（社区版）入门及安装

| | |
|------------------------------------|----|
| 一、环境设置..... | 4 |
| 1.docker 版本及安装环境要求..... | 4 |
| 2.centos7 把网卡名设置为 ethx..... | 4 |
| #1.修改网卡参数..... | 4 |
| #2.禁用该可预测命名规则..... | 4 |
| #3.重新生成 GRUB 配置并更新内核参数 | 4 |
| 3.其它配置..... | 5 |
| #1.关闭 selinux、防火墙（包括开机自启动）..... | 5 |
| #2.时间同步..... | 5 |
| #3.设置主机名..... | 5 |
| #4.epel 源安装..... | 5 |
| 二、知识点..... | 5 |
| 1.what-什么是容器 | 5 |
| 容器与虚拟机..... | 5 |
| 2.why-为什么需要容器? | 6 |
| 容器解决的问题..... | 6 |
| 如何让每种服务能够在所有的部署环境中顺利运行? | 7 |
| Docker 的特性 | 10 |
| 容器的优势..... | 10 |
| 3.How - 容器是如何工作的? | 10 |
| Docker 客户端 | 11 |
| Docker 服务器 | 11 |
| Docker 镜像 | 12 |
| Docker 容器 | 12 |
| Registry..... | 12 |
| Docker 组件如何协作? | 13 |
| 小结..... | 13 |
| 三、docker 安装..... | 14 |
| 1.安装及测试..... | 14 |
| 2.镜像加速器..... | 15 |
| 3.docker 常用命令及操作..... | 16 |
| #1.查找和获取镜像..... | 16 |
| #1) docker 镜像查询 docker search..... | 16 |
| #2) 下载镜像 docker pull..... | 19 |
| #3) 查看本地镜像..... | 19 |
| #2.运行容器..... | 19 |
| #1) 运行容器..... | 19 |
| #2) 启动/停止/删除容器..... | 21 |
| #3) 查看容器状态..... | 22 |
| #4) 登陆容器 docker exec..... | 22 |
| #3.其它常用命令 | 22 |
| 4.例子..... | 24 |

| | |
|--|----|
| #1.例子：安装 nginx 及相关操作 | 24 |
| #2.例子：安装 nginx 指定版本 | 25 |
| #3.例子：安装 nginx+php-fpm（一个镜像） | 29 |
| #4.例子：安装 nginx+php-fpm（两个镜像） | 31 |
| 四、构建镜像..... | 33 |
| 1.使用 docker commit 构建（不推荐） | 33 |
| #1.运行容器并修改容器..... | 33 |
| #2.将容器保存为新的镜像..... | 34 |
| #3.测试..... | 34 |
| 2.慎用 docker commit | 35 |
| 3.使用 Dockerfile 定制镜像 | 35 |
| FROM 表示指定基础镜像..... | 36 |
| RUN 执行命令 | 36 |
| 其它 Dockerfile 指令 | 37 |
| 用 docker build 命令构建镜像..... | 37 |
| 4. 其它生成镜像的方法..... | 38 |
| #1.从 rootfs 压缩包导入 | 38 |
| #2.docker save 和 docker load..... | 39 |
| 五、建立仓库..... | 39 |
| 1.建立公共仓库..... | 40 |
| 2.建立私有仓库..... | 41 |
| #1.安装及运行 registry v2(docker-distribution) | 41 |
| #2.通过 docker tag 重命名镜像，使之与 registry 匹配..... | 41 |
| #3.通过 docker pull 上传镜像..... | 41 |
| #1）解决方法一：修改/etc/docker/daemon.json 文件（客户端操作） | 42 |
| #2）解决方法二：修改/etc/sysconfig/docker 文件（客户端操作）(已弃用)..... | 42 |
| #3）解决方法三：把私有仓库改为 https..... | 43 |
| #4.测试（方法一） | 43 |
| #1）删除原有镜像..... | 43 |
| #2）查找本地所有镜像..... | 43 |
| #3）下载镜像..... | 43 |
| 六、数据卷..... | 44 |
| 1.数据卷介绍..... | 44 |
| 2.数据卷相关操作..... | 44 |
| #1.建立一个数据卷..... | 44 |
| #2.删除数据卷..... | 45 |
| #3.挂载一个主机目录作为数据卷..... | 45 |
| #4.查看数据卷的具体信息..... | 45 |
| #5.挂载一个本地主机文件作为数据卷..... | 46 |
| 3.数据卷容器..... | 46 |
| 4.利用数据卷容器来备份、恢复、迁移数据卷..... | 47 |
| #1.备份..... | 47 |
| #2.恢复..... | 47 |
| 七、使用网络..... | 47 |

| | |
|---------------------------|----|
| 1.外网访问容器..... | 47 |
| #1.参数-P -p | 47 |
| #2. 映射到指定地址的任意端口 | 48 |
| #3. 查看映射端口配置..... | 48 |
| 2.容器互联..... | 48 |
| 八、高级网络配置..... | 48 |
| 1.docker 的三种网络类型..... | 48 |
| #1.none 网络 | 49 |
| #2.host 网络..... | 49 |
| #3.bridge 网络（默认） | 49 |
| 2. 快速配置指南..... | 50 |
| 3.配置 DNS..... | 50 |
| 4. 容器访问控制..... | 51 |
| #1. 容器访问外部网络..... | 51 |
| #2. 容器之间访问..... | 51 |
| #3.访问所有端口 | 51 |
| #4.访问指定端口 | 51 |
| 5.端口映射实现..... | 52 |
| #1.映射容器端口到宿主主机的实现..... | 52 |
| #2.容器访问外部实现..... | 52 |
| #3.外部访问容器实现..... | 52 |
| 6.自定义网桥方式一 | 53 |
| #1.安装桥接..... | 53 |
| #2.创建及配置桥接..... | 53 |
| #3.修改启动参数..... | 54 |
| #4.测试..... | 54 |
| 7.自定义网桥方式二..... | 55 |
| 附录一、Dockerfile 指令详解 | 57 |
| copy 复制文件 | 57 |
| ADD 更高级的复制文件..... | 58 |
| CMD 容器启动命令..... | 58 |
| ENTRYPOINT 入口点 | 59 |
| ENV 设置环境变量 | 59 |
| ARG 构建参数..... | 60 |
| VOLUME 定义匿名卷 | 60 |
| EXPOSE 声明端口 | 61 |
| WORKDIR 指定工作目录..... | 61 |
| USER 指定当前用户 | 61 |
| HEALTHCHECK 健康检查 | 62 |
| ONBUILD 为他人做嫁衣裳..... | 63 |
| 参考文档..... | 65 |
| 附录二、私有仓库配置 https | 65 |
| 1.生成 ssl 证书..... | 65 |
| 2.运行 registry 容器 | 66 |

| | |
|---------------------------------|----|
| 3.测试 pull 镜像..... | 67 |
| 附录三、docker registry v2 api..... | 68 |
| 附录四、参考文章..... | 69 |

一、环境设置

1.docker 版本及安装环境要求

根据官方对 centos 安装 docker，链接地址：

<https://docs.docker.com/engine/installation/linux/centos/#os-requirements>

如果是 64 位（docker 不支持 32 位）的 Centos7

如果要升级 centos7 内核的话可以用下面方法

#升级内核，如无特殊要求请不要升级

```
rpm -Uvh http://www.elrepo.org/elrepo-release-7.0-2.el7.elrepo.noarch.rpm
yum --enablerepo=elrepo-kernel install kernel-ml-devel kernel-ml -y
```

#升级后重启后在进入系统菜单中选择新内核即可，

为了方便我用的是 yum 安装，目前最新版本为 docker-ce v17.06.0

2.centos7 把网卡名设置为 ethx

#1.修改网卡参数

#说白了就是把 enoxxx 形式改为 ethx 形式

```
cd /etc/sysconfig/network-scripts/
cp ifcfg-eno16777736 ifcfg-eno16777736.orig
cp ifcfg-eno33554960 ifcfg-eno33554960.orig
sed -i '/eno16777736/s#eno16777736#eth0#g' ifcfg-eno16777736
sed -n '/eth/p' ifcfg-eno16777736
mv ifcfg-eno16777736 ifcfg-eth0

sed -i '/eno33554960/s#eno33554960#eth1#g' ifcfg-eno33554960
sed -n '/eth/p' ifcfg-eno33554960
mv ifcfg-eno33554960 ifcfg-eth1
```

#2.禁用该可预测命名规则

#你可以在启动时传递 “net.ifnames=0 biosdevname=0 ” 的内核参数

```
grep 'GRUB_CMDLINE_LINUX' /etc/default/grub
sed -i '/GRUB_CMDLINE_LINUX/s/rhgb/rhgb net.ifnames=0 biosdevname=0/' /etc/default/grub
grep 'GRUB_CMDLINE_LINUX' /etc/default/grub
```

#3.重新生成 GRUB 配置并更新内核参数

#运行命令 grub2-mkconfig -o /boot/grub2/grub.cfg 来重新生成 GRUB 配置并更新内核参数

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

#重启

```
shutdown -r now
```

3.其它配置

#1.关闭 selinux、防火墙（包括开机自启动）

```
systemctl stop firewalld
systemctl disable firewalld
systemctl stop NetworkManager
systemctl disable NetworkManager
sed -i '/^SELINUX=/s/enforcing/disabled/' /etc/selinux/config
grep '^SELINUX=' /etc/selinux/config
setenforce off
```

#2.时间同步

```
yum install ntp -y
/usr/sbin/ntpdate pool.ntp.org
echo '#time sync by hua'>>/var/spool/cron/root
echo '*/* * * * * /usr/sbin/ntpdate pool.ntp.org >/dev/null 2>&1'>>/var/spool/cron/root
crontab -l
```

#3.设置主机名

```
cat /etc/hostname
#如果想起个名字可以用下面命令永久生效
hostnamectl set-hostname 主机名
```

#4.epel 源安装

```
rpm -Uvh https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

二、知识点

1.what-什么是容器

容器是一种轻量级、可移植、自包含的软件打包技术，使应用程序可以在几乎任何地方以相同的方式运行。开发人员在自己笔记本上创建并测试好的容器，无需任何修改就能够在生产系统的虚拟机、物理服务器或公有云主机上运行。

容器与虚拟机

谈到容器，就不得不将它与虚拟机进行对比，因为两者都是为应用提供封装和隔离。

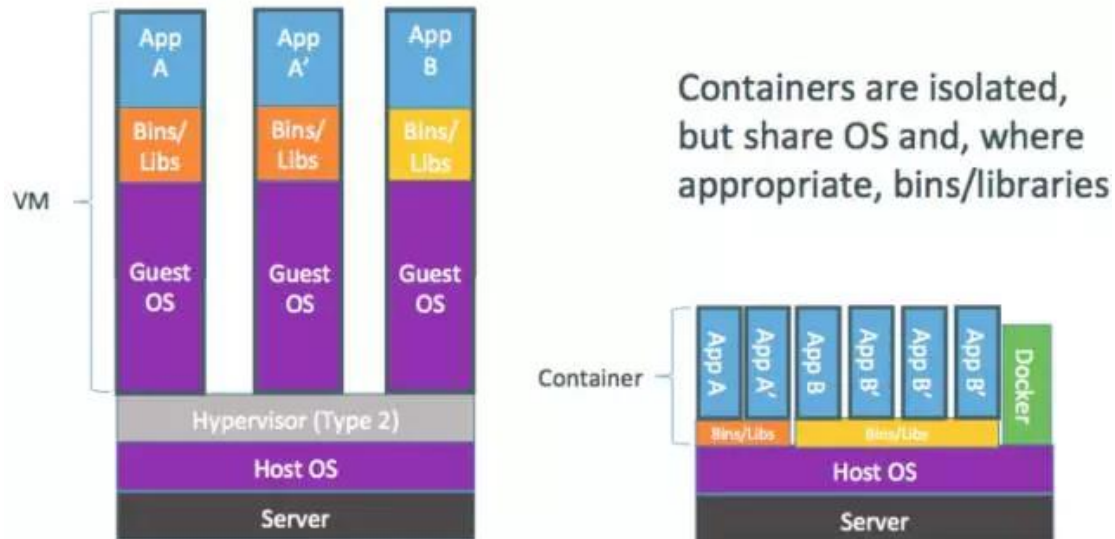
容器由两部分组成：

- 1.应用程序本身
- 2.依赖：比如应用程序需要的库或其他软件

容器在 Host 操作系统的用户空间中运行，与操作系统的其他进程隔离。这一点显著区别于的虚拟机。

传统的虚拟化技术，比如 VMWare, KVM, Xen，目标是创建完整的虚拟机。为了运行应用，除了部署应用本身及其依赖（通常几十 MB），还得安装整个操作系统（几十 GB）。

下图展示了二者的区别



上图所示，由于所有的容器共享同一个 Host OS，这使得容器在体积上要比虚拟机小很多。另外，启动容器不需要启动整个操作系统，所以容器部署和启动速度更快，开销更小，也更容易迁移。

2.why-为什么需要容器？

为什么需要容器？容器到底解决的是什么问题？

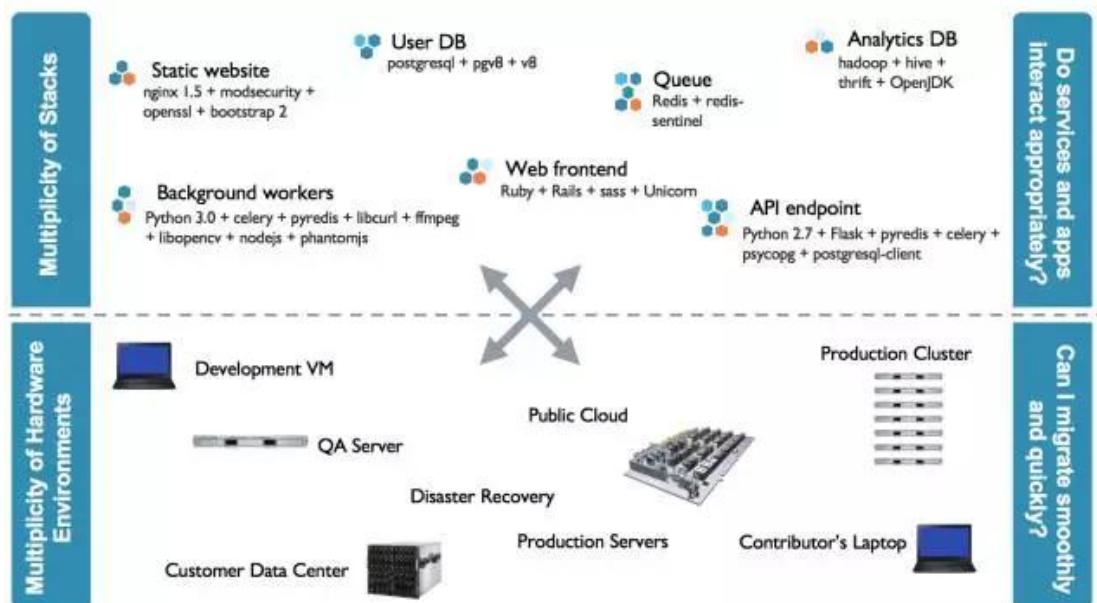
简要的答案是：容器使软件具备了超强的可移植能力。

容器解决的问题

我们来看看今天的软件开发面临着怎样的挑战？

如今的系统在架构上较十年前已经变得非常复杂了。以前几乎所有的应用都采用三层架构（Presentation/Application/Data），系统部署到有限的几台物理服务器上（Web Server/Application Server/Database Server）。

而今天，开发人员通常使用多种服务（比如 MQ，Cache，DB）构建和组装应用，而且应用很可能会部署到不同的环境，比如虚拟服务器，私有云和公有云。



一方面应用包含多种服务，这些服务有自己所依赖的库和软件包；另一方面存在多种部署环境，服务在运行时可能需要动态迁移到不同的环境中。这就产生了一个问题：

如何让每种服务能够在所有的部署环境中顺利运行？

于是我们得到了下面这个矩阵：

| | | | | | | | | |
|--|--------------------|----------------|-----------|--------------------|----------------|--------------|----------------------|------------------|
| | Static website | ? | ? | ? | ? | ? | ? | ? |
| | Web frontend | ? | ? | ? | ? | ? | ? | ? |
| | Background workers | ? | ? | ? | ? | ? | ? | ? |
| | User DB | ? | ? | ? | ? | ? | ? | ? |
| | Queue | ? | ? | ? | ? | ? | ? | ? |
| | | Development VM | QA Server | Single Prod Server | Onsite Cluster | Public Cloud | Contributor's laptop | Customer Servers |
| | | | | | | | | |

各种服务和环境通过排列组合产生了一个大矩阵。开发人员在编写代码时需要考虑不同的运行环境，运维人员则需要为不同的服务和平台配置环境。对他们双方来说，这都是一项困难而艰巨的任务。














如何解决这个问题呢？

聪明的技术人员从传统的运输行业找到了答案。几十年前，运输业面临着类似的问题。


























































每一次运输，货主与承运方都会担心因货物类型的不同而导致损失，比如几个铁桶错误地压在一堆香蕉上。另一方面，运输过程中需要使用不同的交通工具也让整个过程痛苦不堪：货物先装上车运到码头，卸货，然后装上船，到岸后又卸下船，再装上火车，到达目的地，最后卸货。一半以上的时间花费在装、卸货上，而且搬上搬下还容易损坏货物。

这同样也是一个 NxM 的矩阵。

| | | | | | | | |
|---|---|---|---|---|--|---|---|
|  | ? | ? | ? | ? | ? | ? | ? |
|  | ? | ? | ? | ? | ? | ? | ? |
|  | ? | ? | ? | ? | ? | ? | ? |
|  | ? | ? | ? | ? | ? | ? | ? |
|  | ? | ? | ? | ? | ? | ? | ? |
|  | ? | ? | ? | ? | ? | ? | ? |
| |  |  |  |  |  |  |  |

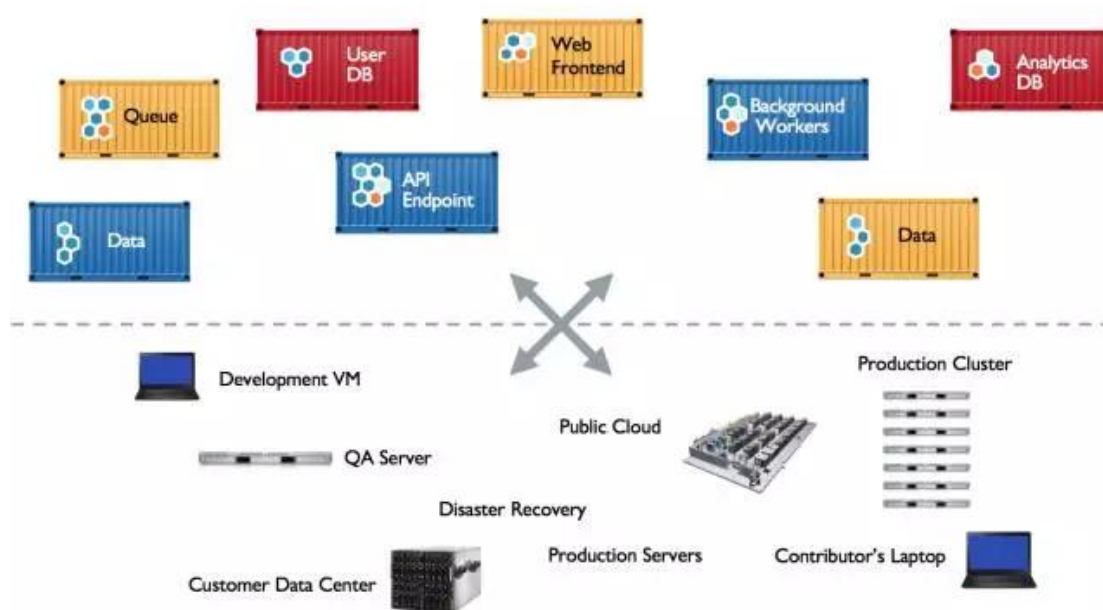
幸运的是，集装箱的发明解决这个难题。

| | | | | | | | |
|---|---|---|---|---|--|---|---|
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| |  |  |  |  |  |  |  |

任何货物，无论钢琴还是保时捷，都被放到各自的集装箱中。集装箱在整个运输过程中都是密封的，只有到达最终目的地才被打开。标准集装箱可以被高效地装卸、重叠和长途运输。现代化的起重机可以自动在卡车、轮船和火车之间移动集装箱。集装箱被誉为运输业与世界贸易最重要的发明。

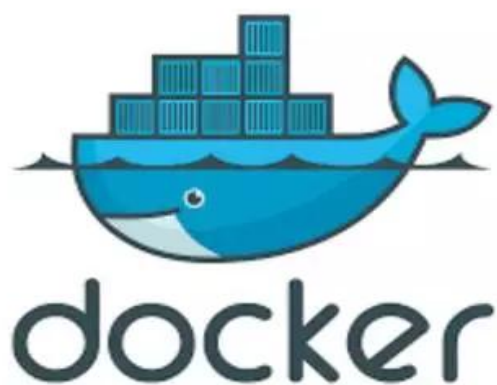


Docker 将集装箱思想运用到软件打包上，为代码提供了一个基于容器的标准化运输系统。Docker 可以将任何应用及其依赖打包成一个轻量级、可移植、自包含的容器。容器可以运行在几乎所有的操作系统上。



其实，“集装箱”和“容器”对应的英文单词都是“Container”。“容器”是国内约定俗成的叫法，可能是因为容器比集装箱更抽象，更适合软件领域的原故吧。

我个人认为：在老外的思维中，“Container”只用到了集装箱这一个意思，Docker 的 Logo 不就是一堆集装箱吗？



Docker 的特性

我们可以看看集装箱思想是如何与 Docker 各种特性相对应的。

| 特性 | 集装箱 | Docker |
|------|-----------------------------------|---|
| 打包对象 | 几乎任何货物 | 任何软件及其依赖 |
| 硬件依赖 | 标准形状和接口允许集装箱被装卸到各种交通工具，整个运输过程无需打开 | 容器无需修改便可运行在几乎所有的平台上 -- 虚拟机、物理机、公有云、私有云 |
| 隔离性 | 集装箱可以重叠起来一起运输，香蕉再也不会被铁桶压烂了 | 资源、网络、库都是隔离的，不会出现依赖问题 |
| 自动化 | 标准接口使集装箱很容易自动装卸和移动 | 提供 <code>run</code> , <code>start</code> , <code>stop</code> 等标准化操作，非常适合自动化 |
| 高效性 | 无需开箱，可在各种交通工具间快速搬运 | 轻量级，能够快速启动和迁移 |
| 职责分工 | 货主只需考虑把什么放到集装箱里；承运方只需关心怎样运输集装箱 | 开发人员只需考虑怎么写代码；运维人员只需关心如何配置基础环境 |

容器的优势

对于开发人员 - **Build Once, Run Anywhere**

容器意味着环境隔离和可重复性。开发人员只需为应用创建一次运行环境，然后打包成容器便可在其他机器上运行。另外，容器环境与所在的 **Host** 环境是隔离的，就像虚拟机一样，但更快更简单。

对于运维人员 - **Configure Once, Run Anything**

只需要配置好标准的 **runtime** 环境，服务器就可以运行任何容器。这使得运维人员的工作变得更高效，一致和可重复。容器消除了开发、测试、生产环境的不一致性。

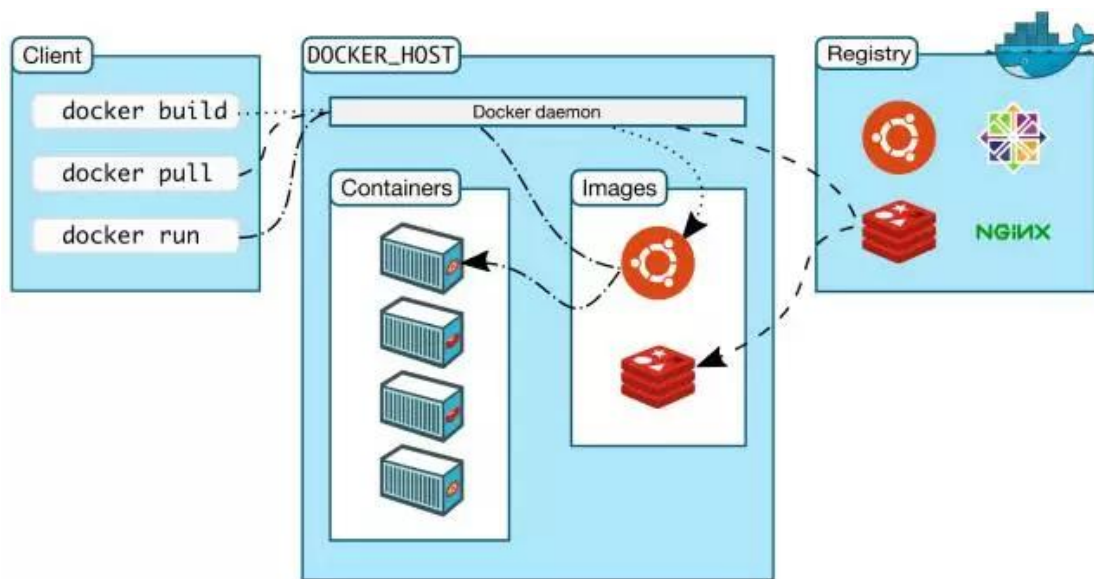
3.How - 容器是如何工作的？

首先会介绍 Docker 的架构，然后分章节详细讨论 Docker 的镜像、容器、网络 and 存储。

Docker 的核心组件包括：

1. Docker 客户端 - `Client`
2. Docker 服务器 - `Docker daemon`
3. Docker 镜像 - `Image`
4. `Registry`
5. Docker 容器 - `Container`

Docker 架构如下图所示：



Docker 采用的是 Client/Server 架构。客户端向服务器发送请求，服务器负责构建、运行和分发容器。客户端和服务端可以运行在同一个 Host 上，客户端也可以通过 socket 或 REST API 与远程的服务器通信。

Docker 客户端

最常用的 Docker 客户端是 `docker` 命令。通过 `docker` 我们可以方便地在 Host 上构建和运行容器。

`docker` 支持很多操作（子命令），后面会逐步用到。

```
[root@vm2 ~]# docker --help
```

Commands:

```
attach      Attach to a running container
build       Build an image from a Dockerfile
commit      Create a new image from a container's changes
cp          Copy files/folders between a container and the local filesystem
create      Create a new container
...
```

除了 `docker` 命令行工具，用户也可以通过 REST API 与服务器通信。

Docker 服务器

Docker daemon 是服务器组件，以 Linux 后台服务的方式运行。

```
[root@vm2 ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since 三 2017-05-24 11:17:45 CST; 3h 39min ago
     Docs: https://docs.docker.com
   Main PID: 4686 (dockerd)
```

Docker daemon 运行在 Docker host 上，负责创建、运行、监控容器，构建、存储镜像。

默认配置下，Docker daemon 只能响应来自本地 Host 的客户端请求。如果要允许远程客户端请求，需要在配置文件中打开 TCP 监听，步骤如下：

1. 编辑配置文件 `/etc/systemd/system/multi-user.target.wants/docker.service`，在环境变量 `ExecStart` 后面添加 `-H tcp://0.0.0.0`，允许来自任意 IP 的客户端连接。

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0
ExecReload=/bin/kill -s HUP $MAINPID
```

如果使用的是其他操作系统，配置文件的位置可能会不一样。

2. 重启 Docker daemon

```
[root@vm2 ~]# systemctl restart docker
Warning: docker.service changed on disk. Run 'systemctl daemon-reload' to reload units.
[root@vm2 ~]# systemctl daemon-reload
```

服务器 IP 为 192.168.56.102，客户端在命令行里加上 -H 参数，即可与远程服务器通信。

```
root@[REDACTED]~# docker -H 192.168.56.102 info
Containers: 3
  Running: 1
  Paused: 0
  Stopped: 2
Images: 1
```

info 子命令用于查看 Docker 服务器的信息。

Docker 镜像

可将 Docker 镜像看作只读模板，通过它可以创建 Docker 容器。

例如某个镜像可能包含一个 Ubuntu 操作系统、一个 Apache HTTP Server 以及用户开发的 Web 应用。

镜像有多种生成方法：

3. 可以从无到有开始创建镜像
4. 也可以下载并使用别人创建好的现成的镜像
5. 还可以在现有镜像上创建新的镜像

我们可以将镜像的内容和创建步骤描述在一个文本文件中，这个文件被称作 Dockerfile，通过执行 `docker build` `<docker-file>` 命令可以构建出 Docker 镜像，后面我们会讨论。

Docker 容器

Docker 容器就是 Docker 镜像的运行实例。

用户可以通过 CLI (docker) 或是 API 启动、停止、移动或删除容器。可以这么认为，对于应用软件，镜像是软件生命周期的构建和打包阶段，而容器则是启动和运行阶段。

Registry

Registry 是存放 Docker 镜像的仓库，Registry 分私有和公有两种。

Docker Hub (<https://hub.docker.com/>) 是默认的 Registry，由 Docker 公司维护，上面有数以万计的镜像，用户可以自由下载和使用。

出于对速度或安全的考虑，用户也可以创建自己的私有 Registry。后面我们会学习如何搭建私有 Registry。

`docker pull` 命令可以从 Registry 下载镜像。

`docker run` 命令则是先下载镜像（如果本地没有），然后再启动容器。

Docker 组件如何协作？

还记得我们运行的第一个容器吗？现在通过它来体会一下 Docker 各个组件是如何协作的。容器启动过程如下：

```
[root@vm2 ~]# docker run -d -p 80:80 httpd ①
Unable to find image 'httpd:latest' locally ②
latest: Pulling from library/httpd
10a267c67f42: Pull complete
0782edf7745a: Pull complete ③
f3a72c4d9d02: Pull complete
dd6ec65d8a55: Pull complete
1b7920e1c0df: Pull complete
5b99e4053015: Pull complete
e720548ad189: Pull complete
Digest: sha256:72b55a7c15a4ee3d56ff19f83b57b82287714f91070b1f556a54e90da5eee3fa
Status: Downloaded newer image for httpd:latest ④
209c55007dac7b2420a49a442ffe02260bf4bcbe54eccf5da2a87b32c39f8254 ⑤
```

1. Docker 客户端执行 `docker run` 命令。
2. Docker daemon 发现本地没有 `httpd` 镜像。
3. daemon 从 Docker Hub 下载镜像。
4. 下载完成，镜像 `httpd` 被保存到本地。
5. Docker daemon 启动容器。

`docker images` 可以查看到 `httpd` 已经下载到本地。

```
[root@vm2 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
httpd                latest             e0645af13ada       12 days ago        177 MB
```

`docker ps` 或者 `docker container ls` 显示容器正在运行。

```
[root@vm2 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
209c55007dac       httpd              "httpd-foreground" 14 minutes ago     Up 14 minutes      0.0.0.0:80->80/tcp  clever_almeida
```

小结

Docker 借鉴了集装箱的概念。标准集装箱将货物运往世界各地，Docker 将这个模型运用到自己的设计哲学中，唯一不同的是：集装箱运输货物，而 Docker 运输软件。

每个容器都有一个软件镜像，相当于集装箱中的货物。容器可以被创建、启动、关闭和销毁。和集装箱一样，Docker 在执行这些操作时，并不关心容器里到底装的什么，它不管里面是 Web Server，还是 Database。

用户不需要关心容器最终会在哪里运行，因为哪里都可以运行。

开发人员可以在笔记本上构建镜像并上传到 Registry，然后 QA 人员将镜像下载到物理或虚拟机做测试，最终容器会部署到生产环境。

使用 Docker 以及容器技术，我们可以快速构建一个应用服务器、一个消息中间件、一个数据库、一个持续集成环境。因为 Docker Hub 上有我们能想到的几乎所有的镜像。

不知大家是否意识到，潘多拉盒子已经被打开。容器不但降低了我们学习新技术的门槛，更提高了效率。

如果你是一个运维人员，想研究负载均衡软件 HAProxy，只需要执行 `docker run haproxy`，无

需繁琐的手工安装和配置既可以直接进入实战。

如果你是一个开发人员，想学习怎么用 `django` 开发 Python Web 应用，执行 `docker run django`，在容器里随便折腾吧，不用担心会搞乱 Host 的环境。

不夸张的说：容器大大提升了 IT 人员的幸福指数。

三、docker 安装

1. 安装及测试

官方链接：<https://docs.docker.com/engine/installation/linux/centos/#os-requirements>

#卸载旧版本，如果存在的话

```
yum remove docker \
    docker-common \
    container-selinux \
    docker-selinux \
    docker-engine
```

#yum 安装 docker CE

```
yum install -y yum-utils device-mapper-persistent-data lvm2
yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
yum makecache fast
yum install -y docker-ce
```

#因官方资源卡,也可以使用阿里云镜像

```
yum install -y yum-utils device-mapper-persistent-data lvm2
#如果 wget 卡住的话多弄几次
wget https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
mv docker-ce.repo /etc/yum.repos.d/
yum makecache fast
yum install -y docker-ce
```

#启动服务

```
systemctl start docker
systemctl enable docker
```

#查看相关信息

```
[root@vm6 docker]# docker version
```

Client:

```
Version:      17.06.0-ce
API version:  1.30
Go version:   go1.8.3
Git commit:   02c1d87
Built:        Fri Jun 23 21:20:36 2017
OS/Arch:      linux/amd64
```



```
Server:
  Version:      17.06.0-ce
  API version:  1.30 (minimum version 1.12)
  Go version:   go1.8.3
  Git commit:   02c1d87
  Built:        Fri Jun 23 21:21:56 2017
  OS/Arch:      linux/amd64
  Experimental: false
```

```
docker info
```

#警告如下：不理它

WARNING: overlay: the backing xfs filesystem is formatted without d_type support, which leads to incorrect behavior.

Reformat the filesystem with ftype=1 to enable d_type support.

Running without d_type support will not be supported in future releases.

WARNING: bridge-nf-call-iptables is disabled

WARNING: bridge-nf-call-ip6tables is disabled

#验证 docker，需要能上网，因为本地没有，会自动从 Docker Hub 下载 hello-world 镜像

```
docker run hello-world
```

```
[root@v ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
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.
 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/
```

2. 镜像加速器

国内访问 Docker Hub 有时会遇到困难，此时可以配置镜像加速器。国内很多云服

务商都提供了加速器服务，例如：

- [阿里云加速器](#)
- [DaoCloud 加速器](#)
- [灵雀云加速器](#)

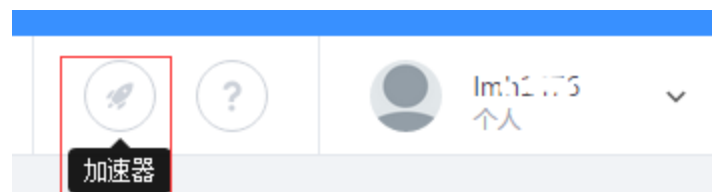
注册用户并且申请加速器，会获得如

<https://jxus37ad.mirror.aliyuncs.com> 这样的地址。我们需要将其配置给 Docker 引擎。

我这里使用免费的 DaoCloud 加速器，注册一个账号，登陆账号后，打开下面链接

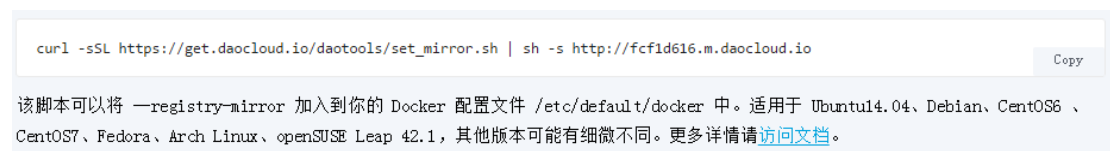
<https://www.daocloud.io/mirror#accelerator-doc>

或者登陆后点账号 logo 旁边的加速器，如下图



会有一个已经生成的加速器脚本：

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://fcfd616.m.daocloud.io
```



#复制上面的命令，执行重启 docker，完成，很简单。

```
[root@vm6 ~]# curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://fcfd616.m.daocloud.io
docker version >= 1.12
{"registry-mirrors": ["http://fcfd616.m.daocloud.io"]}
Success.
You need to restart docker to take effect: sudo systemctl restart docker
[root@vm6 ~]# systemctl restart docker
```

3.docker 常用命令及操作

#1.查找和获取镜像

#1) docker 镜像查询 docker search




我们用 vm 安装 centos 操作系统一样，首先要搜索一下系统的版本。docker 提供了一个官方专业放镜像的地方叫仓库，就是 Docker Hub, Docker Hub (hub.docker.com/explore) 上有大量的高质量的镜像可以用，找开此网址 <https://hub.docker.com/> 点 “Explore”，能搜索到的仓库是免费的



进入如图：

Explore Official Repositories

这个是仓库列表：仓库里面包含有要的镜像，而且是免费的像git pull一样，如果下载就用docker pull命令

| | | | |
|---|---------------|---------------|-------------------------|
|  <div> <div>nginx</div> <div>official</div> </div> | 6.4K STARS | 10M+ PULLS | DETAILS |
|  <div> <div>redis</div> <div>official</div> </div> | 4.0K STARS | 10M+ PULLS | DETAILS |
|  <div> <div>busybox</div> <div>official</div> </div> | 1.1K STARS | 10M+ PULLS | DETAILS |

上图中可以搜索镜像也可以直接点下面的镜像列表中的镜像，如我点第一个 nginx

OFFICIAL REPOSITORY



☆

Last pushed: 4 days ago

Repo Info

Tags

这个是nginx镜像的使用说明，可以了解一下

Short Description

Official build of Nginx.

Full Description

Supported tags and respective Dockerfile links

- 1.13.2, mainline, 1, 1.13, latest ([mainline/stretch/Dockerfile](#))
- 1.13.2-perl, mainline-perl, 1-perl, 1.13-perl, perl ([mainline/stretk](#))

Repo Info

Tags

标签列表：我们看到标签列表中有很多标签表示nginx有很多版本供我们选择，这个比较重要，后面用到的pull命令下载镜像需要用到标签名

| Tag Name | Compressed Size | Last Updated |
|--------------------|-----------------|--------------|
| 1.12-alpine-perl | 18 MB | 4 days ago |
| stable-alpine-perl | 18 MB | 4 days ago |

为什么要有标签，这里就用到镜像和仓库的关系了，如下：

镜像=<仓库>:[标签]

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。上面的截图就是表示 nginx 两个镜像。

#查看帮助

```
[root@vm6 ~]# docker search --help
```

Usage: docker search [OPTIONS] TERM

Search the Docker Hub for images

Options:

-f, --filter filter Filter output based on conditions provided
--help Print usage
--limit int Max number of search results (default 25)
--no-trunc Don't truncate output

简单理解 docker search，语法为：

docker search [选项] <镜像 id/镜像名>[:版本]

注：

1. 上面[]表示可选，<>表示必选，
2. “镜像 id”要到后面用 docker pull 命令下载下来，再用 docker images 才能看到

```
[root@vm6 ~]# docker images nginx
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx                1.12.0             313ec0a602bc       2 weeks ago        107MB
nginx                latest              c246cd3dd41d       2 weeks ago        107MB
```

我们还是拿搜索 nginx 例子，命令如下：

docker search nginx

```
[root@vm6 ~]# docker search nginx 镜像名字一般为“用户名/名字”，第一个只有名字的是官方镜像
NAME                DESCRIPTION          STARS     OFFICIAL   AUTOMATED
nginx               Official build of Nginx. 6377      [OK]
jwilder/nginx-proxy Automated Nginx reverse proxy for docker c... 1062
richarvey/nginx-php-fpm Container running Nginx + PHP-FPM capable ... 403
jrcs/letsencrypt-nginx-proxy-companion LetsEncrypt container to use with nginx as... 198
webdevops/php-nginx Nginx with PHP-FPM 85
million12/nginx-php Nginx + PHP-FPM 5.5, 5.6, 7.0 (NG), CentOS... 77
bitnami/nginx       Bitnami nginx Docker Image 30
evild/alpine-nginx Minimalistic Docker image with Nginx 16
funkygibbon/nginx-pagespeed nginx + ngx_pagespeed + openssl on docker-... 11
webdevops/nginx     Nginx container 8
webdevops/php-nginx-dev PHP with Nginx for Development (eg. with x... 7
blacklabelops/nginx Dockerized Nginx Reverse Proxy Server. 5
lscience/nginx      Nginx Docker images that include Consul Te... 4
frekele/nginx        docker run --rm --name nginx -p 80:80 -p 4... 3
ixbox/nginx          Nginx on Alpine Linux. 3
```

#从上面截图我们看到，用命令是看不到镜像的标签名字的，所在我们要下载特定镜像还是得去

<https://hub.docker.com> 搜索再查看标签名

我们也可以指定版本,根据上面我们用 web 登陆看看 “Repo info” 知道 nginx 默认为 1.13, 如果我想要搜索 1.12 版本呢，那我就就要用 web 看一下 nginx 的 “tags” 中有没有 1.12 版本了，随便找下都发现有好几个，如图：

| | | |
|--------|-------|------------|
| 1.12 | 44 MB | 4 days ago |
| stable | 44 MB | 4 days ago |
| 1.12.0 | 44 MB | 4 days ago |

也可以用命令搜索尝试一下

docker search nginx:1.12

```
[root@vm6 ~]# docker search nginx:1.12
NAME                DESCRIPTION          STARS     OFFICIAL   AUTOMATED
codecoke/nginx.1.12                                0
ark74/nginx-rtmp-1.12                                0
83problems/nginx-centos NGINX 1.12 running on CentOS 6 and 7. 0 [OK]
```

从上图中没有列出 nginx 官方镜像，只能搜索包含 nginx 和 1.12 两个条件的镜像名而已。

#2) 下载镜像 docker pull

上面讲了如何搜索镜像，现在讲如果下载镜像，可以用“docker pull”命令命令格式为：

```
docker pull [选项] [Docker Registry 地址]<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 Registry 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，既 <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。

还是以 nginx 为例子我把上面的 nginx 下载下来

```
#如果不加标签默认就是 latest
```

```
docker pull nginx
```

```
#如果下下载上面的 nginx 1.12 版本怎办，上面有 1.12 和 1.12.0 这里下载 1.12.0
```

```
docker pull nginx:1.12.0
```

#3) 查看本地镜像

```
#帮助
```

```
docker images --help
```

```
#查看全部镜像
```

```
docker images
```

```
#查看指定镜像
```

```
#查看本地名字为 nginx 标签为任意名的镜像
```

```
docker images nginx
```

```
docker images nginx:latest
```

```
docker images nginx:1.12.0
```

#2.运行容器

#1) 运行容器

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。

容器安装用 docker run 命令执行，也可以简单理解为用“docker run”命令来运行镜像得到的东西叫“容器”。

```
#帮助命令
```

```
docker run --help
```

```
语法：
```

```
docker run [选项] <镜像> [命令] [参数...]
```

```
#换成另一个形式就是
```

```
docker run [选项] <镜像 id> [命令] [参数...]
```

```
或
```

```
docker run [选项] <仓库>[:<标签>] [命令] [参数...]
```

```
#注:没写标签默认就是 latest
```

#常用的参数如下:

-d: 表示以“守护模式”执行, 日志不会出现在输出终端上。
--name: 给容器起一个名字, 以后可以用这个名字去操作容易
-i: 表示以“交互模式”运行容器, **-i** 则让容器的标准输入保持打开
--rm: 表示停止退出容器时自动移除容器, 这样就不用先 `docker stop` 再 `docker rm`
-t: 表示容器启动后会进入其命令行, **-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
-v: 表示需要将本地哪个目录挂载到容器中, 格式: **-v** <宿主机目录>:<容器目录>, **-v** 标记来创建一个数据卷并挂载到容器里。在一次 `run` 中多次使用可以挂载多个数据卷。
-p: 表示宿主机与容器的端口映射, **[ip:]<宿主机端口>:[ip:]<docker 端口>**, 如果不写 IP 地址表示宿主所有可用接口相当于 0.0.0.0, 做了映射之后就可以通过外网访问 docker 相关端口了
-p 参数可以用多个, 表示映射多个端口
-P: 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。
不一定要使用“镜像 ID”, 也可以使用“仓库名:标签名”

-v 是最常用的参数, 为什么呢?

运行 docker 镜像变成容器之后, 一般尽量不要让容器有改动, 这样就和镜像一致, 这样的好处有:

1. 方便迁移, 因为我没有改过, 所以万一那台服务器挂了下载一个镜像再运行马上又可以了。
2. 如果过多的更新, 重要的还要做备份, 产生额外的 IO, 还有容器要是坏了怎搞? 用镜像起一个也不能一样?! 最坏的情况万一那个容器被误删了怎搞?!

一般经常性变化的不太建议用容器如数据库, 本身产生的 IO, 内存, CPU 比较容易造成容器退出! 如果有些东西需要改变呢, 那就放在宿主主机目录中用 **-v** 做映射这样方便很多。

比如做 nginx 配置, 自己做一个镜像把网站代码和 nginx 配置放进去, 而网站文件就用 **-v** 映射到 docker 中, 这样就不怕 docker 坏掉, 大不了我再用那个镜像再起一个容器。

什么时候才用到 `docker run -it` 呢, 当 Dockerfile 看默认命令是 `CMD ["/bin/bash"]`, 关于 Dockerfile 是 `CMD` 命令后面会讲到, 这类一般为操作系统比较多, 就要用 **-it** 参数, 要一下就跑完, 容器就退出了。

不能用下面命令运行系统如 centos

#下面运行命令是一执行容器就马上结束了

```
docker run --name centos -d centos:6.8
```

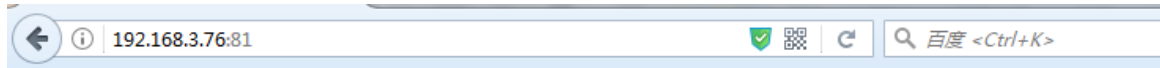
#正确的命令

```
docker run --name centos -it -d centos:6.8 /bin/bash
```

#例子: 最简单的

```
docker run --name nginx -p 81:80 -d nginx
netstat -anltp|grep 81
docker ps
```

#用浏览器测试一下



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

#如果上面的例子想本地主机端口是随机的可以把小写的-p 改为大写的-P
#停止原来的 nginx，为了不和上面的 nginx 容器名称冲突我改为了 t1，
#加多了--rm,表示容器停止就删除

```
[root@vm6 ~]# docker stop nginx
```

nginx

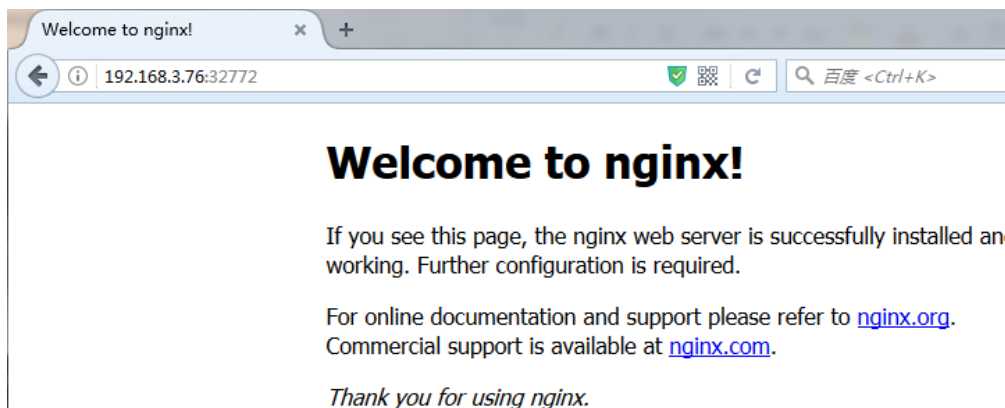
```
[root@vm6 ~]# docker run --rm --name t1 -P -d nginx
```

```
25f2efde4a5f1222ef327024b9bf291d502559d0af83d84a7a7509d986f65668
```

```
[root@vm6 ~]# docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|-------|------------------------|---------------|--------------|-----------------------|-------|
| 25f2efde4a5f | nginx | "nginx -g 'daemon ..." | 3 seconds ago | Up 2 seconds | 0.0.0.0:32772->80/tcp | t1 |

从上面知道是把本机的一个随机端口 32772 映射为 docker 的 80 端口，打开浏览器访问试下：



#2) 启动/停止/删除容器

#start 启动容器

```
docker start <容器 ID>|<容器名>
```

#stop 停止正在运行的容器

```
docker stop <容器 ID>|<容器名>
```

#restart 重启容器

```
docker restart <容器 ID>|<容器名>
```

#rm 删除容器

```
docker rm <容器 ID>|<容器名>
```

#3) 查看容器状态

#列出当前所有正在运行的容器

```
docker ps
```

#列出所有的容器

```
docker ps -a
```

#列出最近一次启动的容器

```
docker ps -l
```

#4) 登陆容器 **docker exec**

我们知道容器也是有自己的操作系统，如果我们想登陆进去看一下，那怎么办？

首先保持容器在运行，然后执行 **docker exec**，语法为：

```
docker exec [选项] <容器> <命令> [参数...]
```

一般情况当使用 **-it /bin/bash** 时就会登陆容器，如登陆上面的 **nginx** 命令为

```
[root@vm6 ~]# docker exec -it nginx /bin/bash
```

```
root@eeb328d0e565:/#pwd
```

```
/
```

```
root@eeb328d0e565:/# exit
```

```
exit
```

```
[root@vm6 ~]#
```

#3.其它常用命令

摘自：<http://blog.csdn.net/birdben/article/details/49873725>

在 ubuntu 中安装 docker

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

查看 docker 的版本信息

```
$ docker version
```

查看安装 docker 的信息

```
$ docker info
```

查看本机 Docker 中存在哪些镜像

```
$ docker images
```

检索 image

```
$ docker search ubuntu:14.04
```

在 docker 中获取 ubuntu 镜像

```
$ docker pull ubuntu:14.04
```

显示一个镜像的历史

```
$ docker history birdben/ubuntu:v1
```



```

# 列出一个容器里面被改变的文件或者目
$ docker diff birdben/ubuntu:v1

# 从一个容器中取日志
$ docker logs birdben/ubuntu:v1

# 显示一个运行的容器里面的进程信息
$ docker top birdben/ubuntu:v1

# 从容器里面拷贝文件/目录到本地一个路径
$ docker cp ID:/container_path to_path

# 查看容器的相关信息
$ docker inspect $CONTAINER_ID

# 显示容器 IP 地址和端口号，如果输出是空的说明没有配置 IP 地址（不同的 Docker 容器可以通过
此 IP 地址互相访问）
$ docker inspect --format='{{.NetworkSettings.IPAddress}}' $CONTAINER_ID

# 保存对容器的修改
$ docker commit -m "Added ssh from ubuntu14.04" -a "birdben" 6s56d43f627f3 birdben/ubuntu:
v1

# 参数:
# -m 参数用来指定提交的说明信息;
# -a 可以指定用户信息的;
# 6s56d43f627f3 代表的容器的 id;
# birdben/ubuntu:v1 指定目标镜像的用户名、仓库名和 tag 信息。

# 构建一个容器
$ docker build -t="birdben/ubuntu:v1" .

# 参数:
# -t 为构建的镜像制定一个标签，便于记忆/索引等
# . 指定 Dockerfile 文件在当前目录下，也可以替换为一个具体的 Dockerfile 的路径。

# 在 docker 中运行 ubuntu 镜像
$ docker run <相关参数> <镜像 ID> <初始命令>

# 守护模式启动
$ docker run -it ubuntu:14.04

# 交互模式启动
$ docker run -it ubuntu:14.04 /bin/bash

```

```
# 指定端口号启动
$ docker run -p 80:80 birdben/ubuntu:v1

# rmi 删除镜像
$ docker rmi ed9c93747fe1Deleted

# 登录 Docker Hub 中心
$ docker login

# 发布上传 image (push)
$ docker push birdben/ubuntu:v1

#查看 docker 状态
docker stats

#清理所有处于终止状态的容器
#用 docker ps -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，可以用下面的命令全部清理掉。
docker rm $(docker ps -a -q)
#注：这个命令其实会试图删除所有的包括还在运行中的容器，
#不过就像上面提过的 docker rm 默认并不会删除运行中的容器。

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

```

4.例子

#1.例子：安装 nginx 及相关操作

```
#查看 nginx 版本
docker search 镜像名
如：docker search nginx
#不能可以指定版本搜索
docker search 镜像名:版本
如：docker search nginx:1.12
#下面搜索出了 nginx 的版本，第一个就是
```

```
[root@vm6 ~]# docker search nginx
```

| NAME | DESCRIPTION | STARS | OFFICIAL | AUTOMATED |
|--|---|-------|----------|-----------|
| nginx | Official build of Nginx. | 6320 | [OK] | |
| jwilder/nginx-proxy | Automated Nginx reverse proxy for docker c... | 1060 | | [OK] |
| richarvey/nginx-php-fpm | Container running Nginx + PHP-FPM capable ... | 399 | | [OK] |
| jrcs/letsencrypt-nginx-proxy-companion | LetsEncrypt container to use with nginx as... | 195 | | [OK] |
| webdevops/php-nginx | Nginx with PHP-FPM | 82 | | [OK] |
| million12/nginx-php | Nginx + PHP-FPM 5.5, 5.6, 7.0 (NG), CentOS... | 77 | | [OK] |
| h3nrik/nginx-ldap | NGINX web server with LDAP/AD, SSL and pro... | 38 | | [OK] |
| bitnami/nginx | Bitnami nginx Docker Image | 30 | | [OK] |
| evild/alpine-nginx | Minimalistic Docker image with Nginx | 16 | | [OK] |
| funkygibbon/nginx-pagespeed | nginx + ngx_pagespeed + openssl on docker-... | 11 | | [OK] |

#像 github 一样获取镜像用的是 docker pull，那推送当然就用 docker push 啦

docker pull nginx

#获取完之后会自动安装，查看镜像是否存

docker images nginx

#运行 nginx 端口为把 docker 的 80 端口映射成本机的 81 端口 docker run -d -p 0.0.0.0:81:80 nginx
netstat -alntp|grep 81

#打开浏览器输入：http://192.168.3.12:81，结果如下图：

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

#2.例子：安装 nginx 指定版本

假如我们想安装 nginx1.12 版本怎办

打开 <https://hub.docker.com/explore/>，第一行为 nginx，点击查看跳到链接

<https://hub.docker.com/r/library/nginx/tags/>里面有所有 nginx 版本号的 tag

| | | |
|-------------|-------|------------|
| 1.12.0-perl | 54 MB | 9 days ago |
| 1.12 | 44 MB | 9 days ago |
| stable | 44 MB | 9 days ago |
| 1.12.0 | 44 MB | 9 days ago |
| perl | 54 MB | 9 days ago |
| 1.13-perl | 54 MB | 9 days ago |

我这里就安装 1.12 版本

也可以直接用 docker search 搜索 nginx:1.12.0，但资源名为“nginx”不会出来

#知道一个仓库名为“nginx”的，版本有很多，找到要的版本，用 docker pull 命令安装即可。

```
[root@vm6 ~]# docker pull nginx:1.12.0
```

```
1.12.0: Pulling from library/nginx
e6e142a99202: Pull complete
60807a6589a7: Pull complete
55c917c49935: Pull complete
Digest: sha256:aea0e686832d38c32a33ea6c6fcd0070598d7f09dce33d3bf7b2ce27b347f600
Status: Downloaded newer image for nginx:1.12.0
```

```
[root@vm6 ~]# docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|-------------|--------|--------------|-------------|--------|
| nginx | 1.12.0 | 313ec0a602bc | 7 days ago | 107MB |
| nginx | latest | c246cd3dd41d | 7 days ago | 107MB |
| hello-world | latest | 1815c82652c0 | 2 weeks ago | 1.84kB |

现在想做的是

- 1) 把 80 端口映射为本机的 80
参数为 `-p <本机端口>:<docker 端口>`
- 2) 给容器起一个名为 `nginx12` 方便记忆和管理。
参数为 `--name 容器名`
- 3) 把本地网站目录 `/disk1/www/hualinux.com` 映射为 docker 相同目录路径
映射用参 `-v <本机目录>:<docker 目录>`
- 4) 把本地日志目录 `/disk1/logs/nginx` 映射为 docker 相同目录路径
- 5) 登陆 docker nginx 和配置 nginx 域名为 `www.hualinux.com`，用 `docker exec -it` 实现

操作如下：

#因本地没有安装 nginx，所在生成一个 nginx 账号

```
groupadd nginx
useradd -s /sbin/nologin -g nginx -M nginx
```

#生成网站目录及日志目录

```
mkdir -p /disk1/www/hualinux.com
mkdir -p /disk1/logs/nginx
chown nginx.nginx -R /disk1/www/hualinux.com
chown nginx.nginx -R /disk1/logs/nginx
```

#建立一个测试文件

```
echo '76 index.html'> /disk1/www/hualinux.com/index.html
```

#docker 相关操作

#停止之前的 docker nginx

```
docker stop nginx
docker rm nginx
docker ps -a|grep nginx
```

#运行 docker 并按上面的按照映射端口和目录，注意下面是一条命令

```
docker run -p 80:80 --name nginx12 \
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com:ro \
-v /disk1/logs/nginx:/disk1/logs/nginx:rw -d nginx:1.12.0
```

```
docker ps
```

```
[root@vm6 ~]# docker run -p 80:80 --name nginx12 -v /disk1/www/hualinux.com:/disk1/www/hualinux.com:ro -v /disk1/logs/nginx:/disk1/logs/nginx:rw -d nginx:1.12.0
737c4153167e9e79863d90bdf184da130d886d40037ec1504e89437dde0ff2b
[root@vm6 ~]# docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|--------------|------------------------|---------------|--------------|--------------------|---------|
| 737c4153167e | nginx:1.12.0 | "nginx -g 'daemon ..." | 3 minutes ago | Up 3 minutes | 0.0.0.0:80->80/tcp | nginx12 |

#从上截图得知容器 ID 为“737c4153167e”名为“nginx12”，这两个都可以对容器进行操作

#重启容器，同理停止为 stop，开始为 start

#docker restart 737c4153167e 也行，为了方便一般用名字操作

docker restart nginx12

#进入容器

[root@vm6 ~]# docker exec -it nginx12 /bin/bash

root@737c4153167e:/# pwd

/

root@737c4153167e:/# ls -l /disk1/

total 0

drwxr-xr-x 3 root root 18 Jun 30 11:12 logs

drwxr-xr-x 3 root root 25 Jun 30 11:12 www

root@737c4153167e:/# cat /disk1/www/hualinux.com/index.html

76 index.html

#生成 nginx

cd /etc/nginx/conf.d/

cp default.conf hualinux.conf

#退出 docker

exit

#因 docker nginx 没有 vi 和 vim 工具，需要复制 docker nginx 到本机上配置，打开另一个 ssh

docker cp nginx12:/etc/nginx/conf.d/hualinux.conf .

#修改 hualinux.conf 配置

sed -i '/server_name/s/localhost/www.hualinux.com/' hualinux.conf

sed -i '/access_log/a\access_log /disk1/logs/nginx/hualinux_com.access.log main;' hualinux.conf

#因为有 2 个 share 结果，只修改第 1 个

sed -i '0,/share/s#/usr/share/nginx/html#/disk1/www/hualinux.com#' hualinux.conf

egrep 'server_name|access_log|share|hualinux' hualinux.conf

#把修改好的文件上传到 docker nginx

docker cp hualinux.conf nginx12:/etc/nginx/conf.d/hualinux.conf

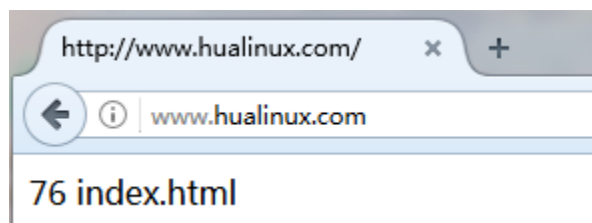
#重启

docker restart nginx12

#测试

在 win 电脑上添加 hosts 绑定添加一行“192.168.3.76 www.hualinux.com”

打开浏览器访问，发现能访问，测试成功



#如果日志映射成功的话，本地 linux 也会有日志生成

```
cat /disk1/logs/nginx/hualinux_com.access.log
```

#也可以登陆 docker 查看一下是否有日志生成，能在本地看一般不登陆

```
docker exec -it nginx12 /bin/bash
```

```
cat /disk1/logs/nginx/hualinux_com.access.log
```

特别注意：容器一但删除里面所有的配置都会消失!!

容器就有点像虚拟化的虚拟主机那个，你删除了虚拟主机那么相应配置自然被删除了。停止容器就是停止虚拟主机那样，存储还在。

#删除 nginx 为 12 的容器

```
docker stop nginx12
```

```
docker rm nginx12
```

```
docker ps -a|grep nginx12
```

#再生成一个容器，看一下刚才配置的 nginx 还在不？

```
docker run -p 80:80 --name nginx12 \
```

```
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com:ro \
```

```
-v /disk1/logs/nginx:/disk1/logs/nginx:rw -d nginx:1.12.0
```

再次访问 <http://www.hualinux.com/>变成了 nginx 首页了，`/etc/nginx/conf.d/hualinux.conf` 配置文件没有了！



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

有人肯定问了，如果有一台 centos 有几十个 docker，如果配置文件保存在 docker 中，万一误删了不是很惨！

解决方式是统一放在本地磁盘中，docker 尽量不做修改，然后用 -v 映射到 docker 中，这样方便管理。

操作如下：在本机 centos7.2 中操作

#为了方便在本机进行测试，在本机也安装一个 nginx，也可以选择不安装。

```
rpm -ivh http://nginx.org/packages/centos/7/noarch/RPMS/nginx-release-centos-7-0.el7ngx.noarch.rpm
```

```
yum install nginx -y
```

```
mkdir -p /disk1/dockerconf/nginx12/nginx/conf.d
```

#把 hualinux.conf 配置放在 `/disk1/dockerconf/nginx12/nginx/conf.d/` 下

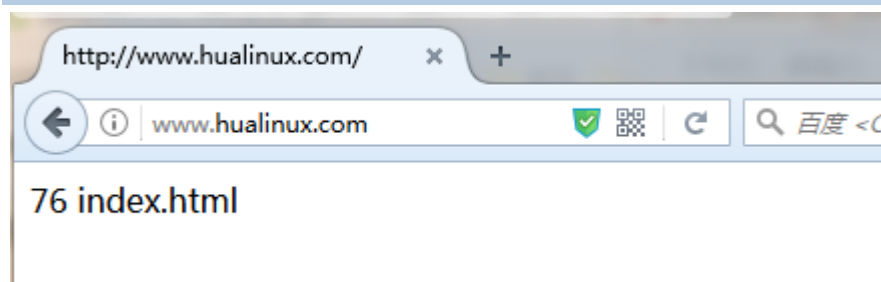
#删除 nginx 为 12 的容器

```
docker stop nginx12
```

```

docker rm nginx12
docker ps -a|grep nginx12
#再生成一个容器，把本地的 nginx 配置映射到 docker 上
docker run -p 80:80 --name nginx12 \
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com:ro \
-v /disk1/logs/nginx:/disk1/logs/nginx:rw \
-v /disk1/dockerconf/nginx12/nginx/conf.d:/etc/nginx/conf.d \
-d nginx:1.12.0
#查看进程
docker ps
#登陆 docker 查看一下配置文件
docker exec -it nginx12 /bin/bash
root@1f919bdd9922:/# ls -l /etc/nginx/conf.d/hualinux.conf
-rw-r--r-- 1 root root 1166 Jul  3 10:31 /etc/nginx/conf.d/hualinux.conf
root@1f919bdd9922:/# exit
exit

```



#3.例子：安装 nginx+php-fpm（一个镜像）

同理也可以安装 nginx+php-fpm 可以如下搜索，php 版本为 5.6

1.先用“docker search nginx-php”命令搜索

```

[root@vm6 ~]# docker search nginx-php
NAME                                DESCRIPTION                                STARS     OFFICIAL   AUTOMATED
richarvey/nginx-php-fpm            Container running Nginx + PHP-FPM capable ... 399       [OK]
webdevops/php-nginx                Nginx with PHP-FPM                        82        [OK]
million12/nginx-php                Nginx + PHP-FPM 5.5, 5.6, 7.0 (NG), CentOS... 77        [OK]

```

2.打网网站: <https://hub.docker.com/explore/>，也搜索“nginx+php-fpm”

找到“million12/nginx-php”并点击进去查看 tag 是多少

<https://hub.docker.com/r/million12/nginx-php/tags/>

PUBLIC | AUTOMATED BUILD

million12/nginx-php ☆

Last pushed: 5 months ago

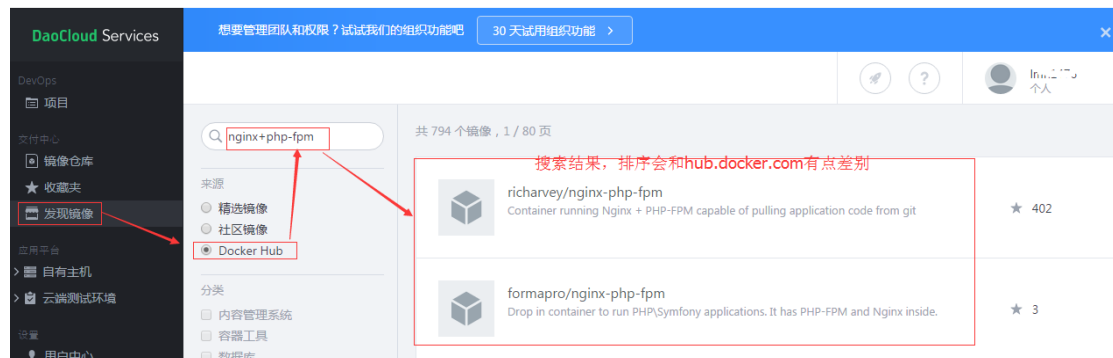
Repo Info **Tags** Dockerfile Build Details

| Tag Name | Compressed Size | Last Updated |
|----------|-----------------|---------------|
| php56 | 361 MB | 5 months ago |
| php70 | 408 MB | 5 months ago |
| latest | 456 MB | 5 months ago |
| php55 | 295 MB | 10 months ago |

现在知道 tag 了，可以用命令安装

```
docker pull million12/nginx-php:php56
```

注：docker 官方的 hub.docker.com 经常搜索有问题，打开慢，可以用 daocloud 搜索
登陆 daocloud



安装项目官方说明：<https://hub.docker.com/r/million12/nginx-php/>

配置文件保存在：`/data/www/default`

#查看/etc/php-fpm.d/www.conf 配置发现 php-fpm 用的是 sock 连接

#发现没有 netstat 命令 `yum install net-tools vim`

#停止上面的 nginx12

```
docker stop nginx12
```

#安装 docker

#因配置文件不一样，所以就不映射/disk1/dockerconf/nginx12/nginx/conf.d 到 docker 上

```
docker run -p 80:80 --name ng-php56 \
```

```
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com:ro \
```

```
-v /disk1/logs/nginx:/disk1/logs/nginx:rw \
```

```
-d million12/nginx-php:php56
```

```
docker ps
```

```
cd /disk1/dockerconf/nginx12/nginx/conf.d/
```

#修改 hualinux.conf 使用让支持 php

```
echo '<?php phpinfo() ?>' >/disk1/www/hualinux.com/info.php
```

#登陆 docker，建立 nginx 配置文件

```
docker exec -it ng-php56 /bin/bash
```

```
cd /etc/nginx/hosts.d/
```

```
vi hualinux.conf
```

```
server {
    listen      80;
    server_name www.hualinux.com;
    #access_log /disk1/logs/nginx/hualinux_com.access.log main;

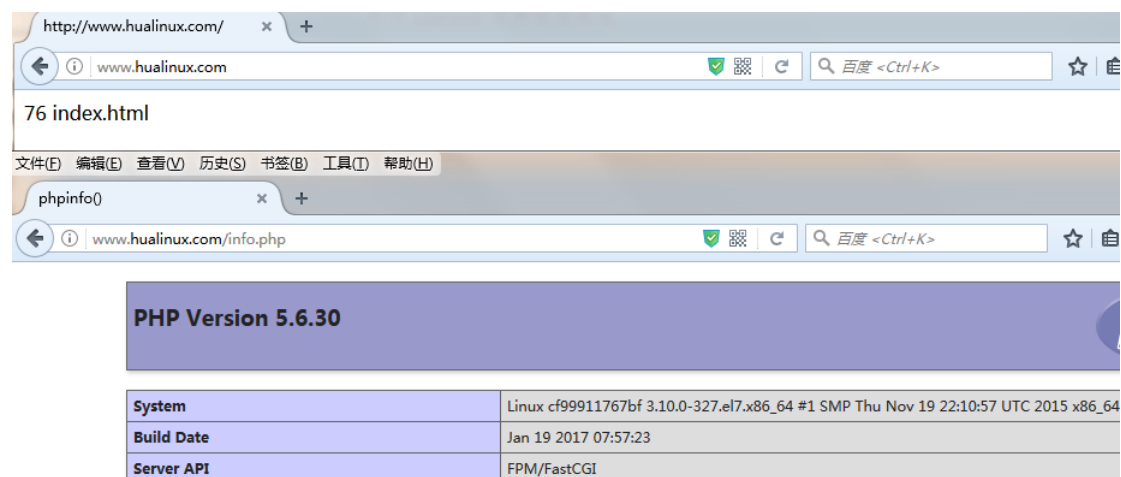
    location / {
        root    /disk1/www/hualinux.com;
        index   index.html index.htm;
    }
}
```

```
# pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
#
location ~ \.php$ {
    root           /disk1/www/hualinux.com;
    #fastcgi_pass   127.0.0.1:9000;
    fastcgi_pass    unix:/var/run/php-fpm-www.sock;
    #fastcgi_index  index.php;
    fastcgi_param   SCRIPT_FILENAME /scripts$fastcgi_script_name;
    include         fastcgi_params;
}
}
```

```
nginx -t
#退出 docker
exit
docker restart ng-php56
```

#测试

打开浏览器输入 <http://www.hualinux.com/>和 <http://www.hualinux.com/info.php>



#4.例子：安装 nginx+php-fpm（两个镜像）

[这个](#)是分开的一个 dokcer 安装 nginx 一个 dokcer 安装 php-fpmdocker nginx 镜像已经安装了，现在安装 php-fpm 用的是 5.6 版本。

分开安装 nginx 和 php-fpm 关联的关键是--link 参数

#1.安装 docker php，在官网中人 php tag 中有 php-fpm 版本

```
docker pull php:5.6.30-fpm
```

#2.运行 docker php，端口映射和目录映射不能少

```
docker run --name php56 -p 9000:9000 \
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com \
-d php:5.6.30-fpm
```

#3.修改 php 配置文件，把监控地址改为 0.0.0.0 或者内网地址也行

```
docker exec -it php56 /bin/bash
```

```
#得知 php-fpm 配置文件为/usr/local/etc/php-fpm.d/www.conf
```

```
ps -ef |grep php-fpm
cd /usr/local/etc/php-fpm.d/
grep '^listen' www.conf
sed -i '/^listen/s/127.0.0.1/0.0.0.0/' www.conf
#退出 docker
exit
```

#4.运行 docker nginx，并用--link 参数关联 php56 容器

#运行之前修改一下 nginx 中 hualinux.conf 配置中的 php 部分，把 127.0.0.1 改为内网 IP 地址

```
cd /disk1/dockerconf/nginx12/nginx/conf.d/
vim hualinux.conf
```

```
...
location ~ \.php$ {
    root          /disk1/www/hualinux.com;
    #fastcgi_pass  127.0.0.1:9000;
    fastcgi_pass   192.168.3.76:9000;
    fastcgi_index  index.php;
    fastcgi_param  SCRIPT_FILENAME  /disk1/www/hualinux.com$fastcgi_script_name;
    include        fastcgi_params;
    fastcgi_intercept_errors on;
}
...
```

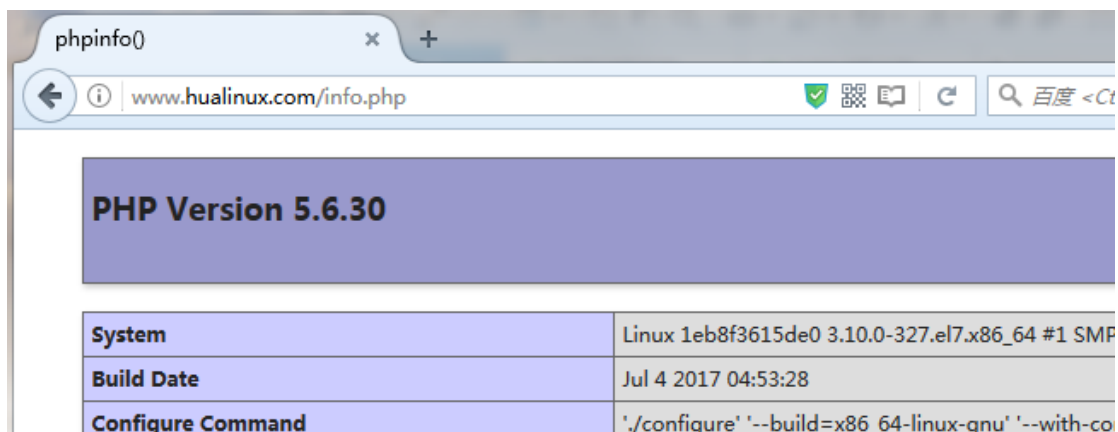
#因 nginx 要关联 php56 容器，所以 php56 容器得先运行！

```
docker run -p 80:80 --name nginx \
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com \
-v /disk1/logs/nginx:/disk1/logs/nginx:rw \
-v /disk1/dockerconf/nginx12/nginx/conf.d:/etc/nginx/conf.d \
--link php56 -d nginx
docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|----------------|------------------------|----------------|---------------|------------------------|-------|
| 1eb8f3615de0 | php:5.6.38-fpm | "docker-php-entryp..." | 9 minutes ago | Up 6 minutes | 0.0.0.0:9000->9000/tcp | php56 |
| 4ccb9144f08b | nginx | "nginx -g 'daemon ..." | 23 minutes ago | Up 23 minutes | 0.0.0.0:80->80/tcp | nginx |

#5.测试效果

打开浏览器输入 <http://www.hualinux.com/info.php>



四、构建镜像

对于 Docker 用户来说,最好的情况是不需要自己创建镜像。几乎所有常用的数据库、中间件、应用软件等都有现成的 Docker 官方镜像或其他人和组织创建的镜像,我们只需要稍作配置就可以直接使用。

使用现成镜像的好处除了省去自己做镜像的工作量外,更重要的是可以利用前人的经验。特别是使用那些官方镜像,因为 Docker 的工程师知道如何更好的在容器中运行软件。

当然,某些情况下我们也不得不自己构建镜像,比如:

- 找不到现成的镜像,比如自己开发的应用程序。
- 需要在镜像中加入特定的功能,比如官方镜像几乎都不提供 ssh。
-

Docker 提供了两种构建镜像的方法:

- 1) docker commit 命令
- 2) Dockerfile 构建文件

1.使用 docker commit 构建（不推荐）

docker commit 命令是创建新镜像最直观的方法,其过程包含三个步骤:

- 1) 运行容器
- 2) 修改容器
- 3) 将容器保存为新的镜像（docker commit 命令）

#1.运行容器并修改容器

我们就以上面 nginx12 为例子,容器已经运行过了,安装命令为

```
docker run -p 80:80 --name nginx12 \  
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com:ro \  
-v /disk1/logs/nginx:/disk1/logs/nginx:rw \  
-v /disk1/dockerconf/nginx12/nginx/conf.d:/etc/nginx/conf.d \  
-d nginx:1.12.0
```

注意:

修改容器要修改原容器中的内容才生效,映射的修改实际上只是修改本机上的内容,如修改“/etc/nginx/conf.d”目录下的文件,生成新容器后是没有变化的。

例如修改 nginx 首页内容,操作如下:

```
#停止所有运行的容器
```

```

docker ps|awk '{print $1}'|grep -v 'CONTAINER'|xargs docker stop
docker start nginx12
docker ps
#登陆容器并修改内容
docker exec -it nginx12 /bin/bash
cd /usr/share/nginx/html
cp index.html index.html.orig
echo "welcome to hua docker nginx">index.html
exit

```

#生成新的镜像的帮助命令

```
[root@vm6 ~]# docker commit -h
```

Flag shorthand -h has been deprecated, please use --help

Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

Options:

```

-a, --author string      Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")
-c, --change list        Apply Dockerfile instruction to the created image
    --help                Print usage
-m, --message string     Commit message
-p, --pause              Pause container during commit (default true)

```

#2.将容器保存为新的镜像

docker commit 命令用生成新镜像，该命令就相当于 VM 虚拟机中生成新的快照一样。

```

[root@vm6 ~]# docker commit -m "nginx 1.12.0 centos7" nginx12 hua:nginx
sha256:91f2eefb953cb713a44a3179de2d95af9605757427e605660d296b6b0f52d176
[root@vm6 ~]# docker images hua

```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|-------|--------------|----------------|-------|
| hua | nginx | 91f2eefb953c | 15 seconds ago | 107MB |

注解：

-m: 为描述

nginx12: 使用的容器名字

hua:nginx: 仓库名为 hua, tag 为 nginx,如果不写默认 latest

#3.测试

#安装刚刚生成的 hua:nginx 镜像

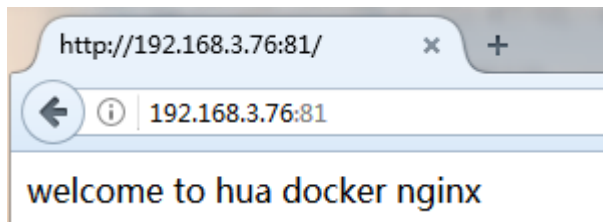
#因为 80 端口被 nginx12 占了，我用 81 端口

```

docker run --name hua-ng -p 81:80 -d hua:nginx
docker ps|grep hua-ng

```

#打开浏览器输入 <http://192.168.3.76:81/>，效果如下：说明新容器成功了。



2.慎用 docker commit

使用 `docker commit` 命令虽然可以比较直观的帮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。

我们刚才只是创建了一个很小的 `html` 文件，所以镜像大小变化小，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。

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。

`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，定制行为应该使用 `Dockerfile` 来完成。

3.使用 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 `nginx12` 镜像为例，这次我们使用 `Dockerfile` 来定制

#删除 hua-ng 容器及刚刚建立的 hua:nginx 镜像

```
docker stop hua-ng
docker rm hua-ng
[root@vm6 ~]# docker images hua
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hua                  nginx               91f2eefb953c       4 hours ago        107MB
[root@~]# docker rmi hua:nginx
Untagged: hua:nginx
Deleted: sha256:91f2eefb953cb713a44a3179de2d95af9605757427e605660d296b6b0f52d176
Deleted: sha256:9f88f9328fdc9d651552ccc9d243b63bb8323b9056a5544f01bb02dc14a33848
```

#查看 `nginx12` 容器对应的镜像，在这里是第一行的那个

```
[root@vm6 ~]# docker images nginx
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|--------|--------------|-------------|-------|
| nginx | 1.12.0 | 313ec0a602bc | 12 days ago | 107MB |
| nginx | latest | c246cd3dd41d | 12 days ago | 107MB |

在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：

```
mkdir /disk1/hua-docker
cd /disk1/hua-docker/
cat>Dockerfile<<EOF
FROM nginx:1.12.0
RUN echo 'welcome to hua docker nginx' > /usr/share/nginx/html/index.html
EOF
cat Dockerfile
```

其中 **FROM** 接镜基础镜像，可以是官方远程仓库中的，也可以位于本地仓库。

格式为：**FROM** <仓库>:<标签>

FROM 表示指定基础镜像

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。

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 'welcome to hua docker nginx' > /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。

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。

```
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
```


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。

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。

其它 Dockerfile 指令

请看附录一

用 docker build 命令构建镜像

#查看帮助

```
docker build --help
```

```
docker build -t hua:nginx .
```

注：最后有一个小圆点，不要忘记了！点不能用 Dockerfile 代替，也不能用 /disk1/hua-docker/Dockerfile，小圆点是上下文路径（都是相对路径）

否则运行会报错，结果如下：

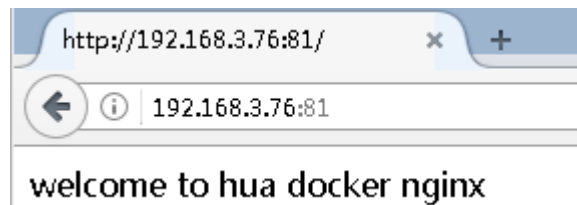
```
[root@vm6 hua-docker]# docker build -t hua:nginx .
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM nginx:1.12.0
--> 313ec0a602bc
Step 2/2 : RUN echo 'welcome to hua docker nginx' > /usr/share/nginx/html/index.html
--> Running in 1f275ce19348
--> 8f7a11a1630f
Removing intermediate container 1f275ce19348
Successfully built 8f7a11a1630f
Successfully tagged hua:nginx
[root@vm6 hua-docker]# docker images hua 成功生成了一个hua的镜像
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|-------|--------------|--------------------|-------|
| hua | nginx | 8f7a11a1630f | About a minute ago | 107MB |

#运行 hua:nginx 镜像看一下效果

```
docker run --name hua-ng -p 81:80 -d hua:nginx
docker ps|grep hua-ng
```

#打开浏览器查看一下效果，和 docker commit 生成的镜像效果一样，说明成功了。



上面中“.”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。

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。

4. 其它生成镜像的方法

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。

#1.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
```

```
Downloading from http://download.openvz.org/template/precreated/ubuntu-14.04-x86_64-minimal.tar.gz  
sha256:f477a6e18e989839d25223f301ef738b69621c4877600ae6467c4e528  
9822a79B/78.42 MB
```

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`

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/templa |
| te/precreated/ubuntu-14.04-x86_64-minimal.tar.gz | | | | |

#2.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 `DockerRegistry` 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 `nginx` 镜像。

```
docker images nginx:latest
```

```
docker save nginx:latest | gzip > nginx-latest.tar.gz
```

然后我们将 `nginx-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 nginx-latest.tar.gz
```

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'
```

五、建立仓库

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。

目前 `Docker` 官方维护了一个公共仓库，链接为 `https://hub.docker.com/`，大部分需求，都可以通过在 `Docker Hub` 中直接下载镜像来实现。

用户可以将自己的镜像保存到 `Docker Hub` 免费的 `repository` 中。如果不希望别人访问自己的镜像，也可以购买私有 `repository`。

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

1.建立公共仓库

#1.注册账号

建立公共仓库去 <https://hub.docker.com> 上注册一个账号，我的为 hualinux

#2.在 Docker Host 上登录，我用的是 hualinux 账号

```
[root@vm6 ~]# docker login -u hualinux
```

Password:

Login Succeeded

#3. 修改镜像的 repository 使之与 Docker Hub 账号匹配。

Docker Hub 为了区分不同用户的同名镜像，镜像的 registry 中要包含用户名，完整格式为: [username]/xxx:tag，我们通过 docker tag 命令重命名镜像。

```
[root@vm6 ~]# docker tag hua:nginx hualinux/hua:nginx
```

```
[root@vm6 ~]# docker images hualinux/hua
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--------------|-------|--------------|--------------|-------|
| hualinux/hua | nginx | 8f7a11a1630f | 42 hours ago | 107MB |

#4. 通过 docker push 将镜像上传到 Docker Hub，国内访问很卡，需要尝试 N 次方可成功!!


```
docker push hualinux/hua:nginx
```

Docker 会上传镜像的每一层，如果想上传同一 repository 中所有镜像，省略 tag 部分就可以了，例如：

```
docker push hualinux/hua
```

#5.登录 <https://hub.docker.com>，在 Public Repository 中就可以看到上传的镜像。

Repositories



hualinux/hua
public

0
STARS

1
PULLS

>
DETAILS

[Repo Info](#) [Tags](#) [Collaborators](#) [Webhooks](#) [Settings](#)

| Tag Name | Compressed Size | Last Updated |
|----------|-----------------|----------------|
| nginx | 44 MB | 25 minutes ago |

#如果要删除上传的镜像，只能在 Docker Hub 界面上操作

#6.这样，所有人就可以用下面命令下载了

```
docker push hualinux/hua:nginx
```

2. 建立私有仓库

Docker Hub 虽然非常方便，但还是有些限制，比如：

- 1) 需要 internet 连接，而且下载和上传速度慢。
- 2) 上传到 Docker Hub 的镜像任何人都能够访问，虽然可以用私有 repository，但不是免费的。
- 3) 安全原因很多组织不允许将镜像放到外网。

解决方案就是搭建本地的 Registry。

Docker 已经将 Registry 开源了，同时在 Docker Hub 上也有官方的镜像 registry。下面我们就在 Docker 中运行自己的 registry。

#1. 安装及运行 registry v2(docker-distribution)

#docker-registry v2 版本即是 docker-distribution 代替，为了简单方便起见，我们直接安装 docker-distribution，省去了配置的麻烦。

#建立存放镜像数据目录

```
mkdir -p /disk1/myregistry
```

#拉 registry 镜像下来，第一个只有 registry，没用用户名的就是官方镜像目前为版本为 v2.6.1

| NAME | DESCRIPTION | STARS | OFFICIAL | AUTOMATED |
|---------------------------------------|---|-------|----------|-----------|
| registry 没用户名的则为官方版本 | The Docker Registry 2.0 implementation for... | 1556 | [OK] | |
| konradkleine/docker-registry-frontend | Browse and modify your Docker registry in ... | 152 | | [OK] |
| hyper/docker-registry-web | Web UI, authentication service and event r... | 99 | | [OK] |
| atcol/docker-registry-ui | A web UI for easy private/local Docker Reg... | 94 | | [OK] |
| distribution/registry | WARNING: NOT the registry official image!!... | 49 | | [OK] |
| marvambass/nginx-registry-proxy | Docker Registry Reverse Proxy with Basic A... | 42 | | [OK] |
| h3nr1k/registry-ldap-auth | LDAP and Active Directory authentication p... | 19 | | [OK] |
| jhipster/jhipster-registry | JHipster Registry, based on Netflix Eureka... | 16 | | [OK] |

```
docker search registry
```

```
docker pull registry
```

```
[root@vm6 ~]# docker images registry
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|--------|--------------|------------|--------|
| registry | latest | c2a449c9f834 | 8 days ago | 33.2MB |

#运行容器

```
docker run -d -p 5000:5000 --name hua-reg -v /disk1/myregistry:/var/lib/registry registry
```

#查看 hua-reg 仓库状态

```
docker ps
```

```
netstat -altnp|grep 5000
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|----------|------------------------|----------------|---------------|------------------------|---------|
| 4473cf7228d9 | registry | "/entrypoint.sh /e..." | 13 seconds ago | Up 13 seconds | 0.0.0.0:5000->5000/tcp | hua-reg |
| tcp6 | 0 | 0 :::5000 | :::* | LISTEN | 20567/docker-proxy | |

#2. 通过 docker tag 重命名镜像，使之与 registry 匹配

#镜像的前面加上了运行 registry 的主机名称和端口，下面 IP 也可以用域名代替

```
docker tag hualinux/hua:nginx 192.168.3.76:5000/hualinux/hua:nginx
```

镜像名称由 repository 和 tag 两部分组成。repository 的完整格式为：

[registry-host]:[port]/[username]/xxx，只有 Docker Hub 上的镜像可以省略 [registry-host]:[port] 。

#3. 通过 docker pull 上传镜像

#如果 IP 地址用的是域名的话则要修改为应用的域名

```
docker push 192.168.3.76:5000/hualinux/hua:nginx
```

#报错:

```
[root@vm6 ~]# docker push 192.168.3.76:5000/hualinux/hua:nginx
The push refers to a repository [192.168.3.76:5000/hualinux/hua]
Get https://192.168.3.76:5000/v2/: http: server gave HTTP response to HTTPS client
```

分析:

上面报错是因为用 https 访问，而私有仓库没有配置 https 引起的。

#1) 解决方法一: 修改/etc/docker/daemon.json 文件 (客户端操作)

因为我这台服务器客户端和服务端一起，所以在本地上修改

因为我之前用了 docker 镜像加速，所以修改一下

#如果不要 docker 加速也可以，输入下面命令

```
# echo '{"insecure-registries": ["192.168.3.76:5000"]}' > /etc/docker/daemon.json
```

```
cd /etc/docker/
```

```
cp daemon.json daemon.json.orig
```

```
vim /etc/docker/daemon.json
```

```
{"registry-mirrors": ["http://fcf1d616.m.daocloud.io"],
"insecure-registries": ["192.168.3.76:5000"]}
}
```

```
{"registry-mirrors": ["http://fcf1d616.m.daocloud.io"],
"insecure-registries": ["192.168.3.76:5000"]}
}
```

```
cat /etc/docker/daemon.json
```

```
systemctl restart docker
```

```
docker start hua-reg
```

```
docker ps
```

#重新提交

```
docker push 192.168.3.76:5000/hualinux/hua:nginx
```

```
[root@vm6 ~]# docker push 192.168.3.76:5000/hualinux/hua:nginx
The push refers to a repository [192.168.3.76:5000/hualinux/hua]
98328461d632: Pushed
caef1bbc7dcb: Pushed
8246593fe84c: Pushed
54522c622682: Pushed
nginx: digest: sha256:88b35551654955e5b15dcbe98806d6f6e5257fa2d9bf7f4f161512f03bcc2b6e size: 1155
```

#注: 如果别的 docker 主机上要获取私人仓库，也一平要修改一下/etc/docker/daemon.json

#2) 解决方法二: 修改/etc/sysconfig/docker 文件 (客户端操作) (已弃用)

此方法早期一些版本可以使用，现版本已失败，都没有/etc/sysconfig/docker 配置文件!

```
echo 'OPTIONS="--selinux-enabled --insecure-registry 192.168.3.76:5000"'>>/etc/sysconfig/docker
```

```
cat /etc/sysconfig/docker
```

#重启 docker

```
systemctl restart docker
```

#再启动 registry

```
docker start hua-reg
```

#测试


```
newer image for hello-world:latest
docker tag hello-world 192.168.3.76:5000/hualinux/hua:hello
docker push 192.168.3.76:5000/hualinux/hua:hello
```

#3) 解决方法三：把私有仓库改为 https

上面两种解决方法做的缺点是你的私有仓库不安全，其次，其他要下载或者上传镜像的机器都要修改相应的配置文件。

具体配置查附录二

#4.测试（方法一）

#1) 删除原有镜像

#删除 192.168.3.76:5000/hualinux/hua:nginx

```
docker rmi 192.168.3.76:5000/hualinux/hua:nginx
docker images 192.168.3.76:5000/hualinux/hua:nginx
```

```
[root@vm6 ~]# docker rmi 192.168.3.76:5000/hualinux/hua:nginx
Untagged: 192.168.3.76:5000/hualinux/hua:nginx
Untagged: 192.168.3.76:5000/hualinux/hua@sha256:88b35551654955e5b15dcbe98806d6f6e5257fa2d9bf7f4f161512f03bcc2b6e
[root@vm6 ~]# docker images 192.168.3.76:5000/hualinux/hua:nginx 没有镜像了，删除成功
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
[root@vm6 ~]#
```

#2) 查找本地所有镜像

#根据 docker registry v2 api 接口列表，请看“附录三”得知查看镜像方式如下：

#v1 版本为 curl 192.168.3.76:5000/v1/search，现在 v2 版本用 _catalog

```
[root@vm6 v2]# curl 192.168.3.76:5000/v2/_catalog
{"repositories":["hualinux/hua"]}
[root@vm6 v2]# curl http://192.168.3.76:5000/v2/hualinux/hua/tags/list
{"name":"hualinux/hua","tags":["nginx"]}
```

上面也可以用浏览器访问

#3) 下载镜像

#除了镜像的名称长一些（包含 registry host 和 port），使用方式完全一样。

```
[root@vm6 ~]# docker pull 192.168.3.76:5000/hualinux/hua:nginx
nginx: Pulling from hualinux/hua
Digest: sha256:88b35551654955e5b15dcbe98806d6f6e5257fa2d9bf7f4f161512f03bcc2b6e
Status: Downloaded newer image for 192.168.3.76:5000/hualinux/hua:nginx
docker images 192.168.3.76:5000/hualinux/hua:nginx
```

#同理添加多一个镜像试下，这里用的是 hello-world

```
[root@vm6 ~]# docker tag hello-world 192.168.3.76:5000/hualinux/hua:hello
[root@vm6 ~]# docker push 192.168.3.76:5000/hualinux/hua:hello
The push refers to a repository [192.168.3.76:5000/hualinux/hua]
45761469c965: Pushed
hello: digest: sha256:9fa82f24cbb11b6b80d5c88e0e10c3306707d97ff862a3018f22f9b49cef303a size: 524
[root@vm6 ~]# curl http://192.168.3.76:5000/v2/hualinux/hua/tags/list
{"name":"hualinux/hua","tags":["nginx","hello"]}
[root@vm6 ~]# curl 192.168.3.76:5000/v2/_catalog
```



```
{"repositories":["hualinux/hua"]}
```

六、数据卷

1.数据卷介绍

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。

2.数据卷相关操作

#1.建立一个数据卷

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

例如：

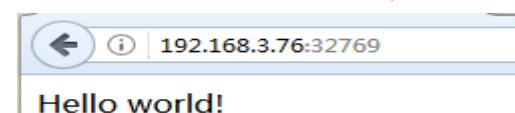
```
docker run -d -P --name web -v /webapp training/webapp python app.py
```

上面的命令意思为：

用“training/webapp”镜像创建一个名字为 web 的容器，端口为随机端口，建立一个“/webapp”目录，并在容器当前目录执行“python app.py”命令

```
[root@vm6 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
e8d56bca16e0      training/webapp    "python app.py"     4 minutes ago      Up 4 seconds       0.0.0.0:32769->5000/tcp   web
[root@vm6 ~]# docker exec -it web /bin/bash
root@e8d56bca16e0:/opt/webapp# pwd
/opt/webapp
root@e8d56bca16e0:/opt/webapp# ll
total 20
drwxr-xr-x 2 root root  89 Jul 10 09:37 ./
drwxr-xr-x 3 root root 19 Jul 10 09:37 ../
-rw-r--r-- 4 root root 11 May 15 2015 .gitignore
-rw-r--r-- 4 root root 19 May 15 2015 Procfile
-rw-r--r-- 4 root root 347 May 15 2015 app.py
-rw-r--r-- 4 root root 41 May 15 2015 requirements.txt
-rw-r--r-- 4 root root 285 May 15 2015 tests.py
root@e8d56bca16e0:/opt/webapp# ll /webapp/
total 0
drwxr-xr-x 2 root root  6 Jul 10 09:37 ./
drwxr-xr-x 1 root root 67 Jul 10 09:37 ../
root@e8d56bca16e0:/opt/webapp# exit
exit
```

在上截中可以用浏览器输入 <http://192.168.3.76:32769/> 进行访问



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

#2.删除数据卷

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，要清理会很麻烦。Docker 官方正在试图解决这个问题，相关工作的进度可以查看这个 [PR](#)。

#3.挂载一个主机目录作为数据卷

使用 `-v` 标记也可以指定挂载一个本地主机的目录到容器中去。如第三章例子安装 `nginx12`

```
docker run -p 80:80 --name nginx12 \  
-v /disk1/www/hualinux.com:/www/hualinux.com -d nginx:1.12.0
```

上面的命令是加载主机的 `/disk1/www/hualinux.com` 目录到容器的 `/www/hualinux.com` 目录。这个功能在进行测试的时候十分方便，比如用户可以放置一些程序到本地目录中，来查看容器是否正常工作。本地目录的路径必须是绝对路径，如果目录不存在 Docker 会自动为你创建它。

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

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

```
docker run -p 80:80 --name nginx12 \  
-v /disk1/www/hualinux.com:/www/hualinux.com:ro -d nginx:1.12.0
```

加了 `:ro` 之后，就挂载为只读了。

#4.查看数据卷的具体信息

```
docker inspect web
```

在输出的内容中找到其中和数据卷相关的部分，可以看到所有的数据卷都是创建在主机的 `/var/lib/docker/volumes/` 下面的

```
"Mounts": [  
  {  
    "Type": "volume",  
    "Name": "54fb6...9bfdb6adccf00e64",  
    "Source": "/var/lib/docker/volumes/54fb...00e64/_data",  
    "Destination": "/webapp",  
    "Driver": "local",  
    "Mode": "",  
    "RW": true,  
    "Propagation": ""  
  }  
],
```

#5.挂载一个本地主机文件作为数据卷

-v 标记也可以从主机挂载单个文件到容器中

```
docker run --name nginx --rm -it -v ~/.bash_history:/bash_history nginx /bin/bash
```

这样就可以记录在容器输入过的命令了。

***注意：**如果直接挂载一个文件，很多文件编辑工具，包括 **vi** 或者 **sed --inplace**，可能会造成文件 **inode** 的改变，从 **Docker 1.1 .0** 起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

3.数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。

数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载。

首先，创建一个名为 **dbdata** 的数据卷容器：

#该容器运行一下就会自动停止，这是正常的

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

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

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

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

设置完之后你，你可以进入 **db1** 的容器在 **/dbdata** 里面建立一个文件，再去 **db2** 查看结果是一样的就像 **NFS** 挂载一样。

```
[root@vm6 ~]# docker exec -it db1 /bin/bash
root@988d51ca3c89:/# cd /dbdata/
root@988d51ca3c89:/dbdata# echo 'this is db1'>db1.txt
root@988d51ca3c89:/dbdata# exit
exit
[root@vm6 ~]# docker exec -it db2 cat /dbdata/db1.txt
this is db1
```

可以使用超过一个的 **--volumes-from** 参数来指定从多个容器挂载不同的数据卷。也可以从其他已经挂载了数据卷的容器来级联挂载数据卷。

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

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

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

4.利用数据卷容器来备份、恢复、迁移数据卷

#1.备份

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

```
[root@vm6 ~]# docker run --volumes-from dbdata -v $(pwd):/backup centos:6.8 tar cvf /backup/backup.tar /dbdata
tar: Removing leading '/' from member names
/dbdata/
/dbdata/db1.txt
[root@vm6 ~]# tar -xf backup.tar
[root@vm6 ~]# cat dbdata/db1.txt
this is db1
```

容器启动后，使用了 `tar` 命令来将 `dbdata` 卷备份为容器中 `/backup/backup.tar` 文件，也就是主机当前目录下的名为 `backup.tar` 的文件

#2.恢复

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

```
docker run -v /dbdata --name dbdata2 centos:6.8 /bin/bash
```

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

```
[root@vm6 ~]# docker run --volumes-from dbdata2 -v $(pwd):/backup busybox tar xvf /backup/backup.tar
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
27144aa8f1b9: Pull complete
Digest: sha256:be3c11fdb7cfe299214e46edc642e09514dbb9bbefcd0d3836c05a1e0cd0642
Status: Downloaded newer image for busybox:latest
dbdata/
dbdata/db1.txt
```

为了查看/验证恢复的数据，可以再启动一个容器挂载同样的容器卷来查看

```
[root@vm6 ~]# docker run --volumes-from dbdata2 busybox /bin/ls /dbdata
db1.txt
```

七、使用网络

1.外网访问容器

#1.参数-P -p

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。

`-P`(大写): 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。

格式 `-P`

`-p`(小写): 映射指定的网络，格式 `-p [ip:]<本机端口>:[ip:]<docker 端口>`，也可以 IPv6

```
-p [ip::]<本机端口>:[ip::]<docker 端口>
```

如果不写地址就表示映射本地所有接口地址

#2. 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 --rm --name web -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 --rm --name web2 -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

#3. 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

```
[root@vm6 ~]# docker port web
```

```
5000/tcp -> 127.0.0.1:32768
```

```
[root@vm6 ~]# docker port web 5000
```

```
127.0.0.1:32768 127.0.0.1:49155.
```

注意：

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

例如

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

2. 容器互联

使用 `--link` 参数可以让容器之间安全的进行交互。

例子请看“[第三章 4 例子，例子 4](#)”

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

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

```
docker run -p 80:80 --name t2 \  
-v /disk1/www/hualinux.com:/disk1/www/hualinux.com \  
-v /disk1/logs/nginx:/disk1/logs/nginx:rw \  
-v /disk1/dockerconf/nginx12/nginx/conf.d:/etc/nginx/conf.d \  
--link php56 nginx env
```

八、高级网络配置

1.docker 的三种网络类型

`Docker` 安装时会自动在 `host` 上创建三个网络，我们可用 `docker network ls` 命令查看：

```
[root@vm6 ~]# docker network ls
```

| NETWORK ID | NAME | DRIVER | SCOPE |
|--------------|--------|--------|-------|
| 49a29406f457 | bridge | bridge | local |
| da0dcddd5892 | host | host | local |
| c68a89dab29a | none | null | local |

#1.none 网络

顾名思义，none 网络就是什么都没有的网络。挂在这个网络下的容器除了 lo，没有其他任何网卡。容器创建时，可以通过 `--network=none` 指定使用 none 网络。

我们不禁会问，这样一个封闭的网络有什么用呢？

其实还真有应用场景。封闭意味着隔离，一些对安全性要求高并且不需要联网的应用可以使用 none 网络。比如某个容器的唯一用途是生成随机密码，就可以放到 none 网络中避免密码被窃取。当然大部分容器是需要网络的，我们接着看 host 网络。

#2.host 网络

连接到 host 网络的容器共享 Docker host 的网络栈，容器的网络配置与 host 完全一样。可以通过 `--network=host` 指定使用 host 网络。

```
[root@vm6 ~]# docker run -it --network=host busybox
/ # ip l
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:0c:29:24:32:b4 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:0c:29:c6:e8:18 brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:68:2d:06:78 brd ff:ff:ff:ff:ff:ff
314: veth3e519ba@if3113: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master docker0
    link/ether c6:75:63:08:8a:23 brd ff:ff:ff:ff:ff:ff
318: veth4ea0999@if3117: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master docker0
    link/ether 4a:f0:7a:5c:6a:d6 brd ff:ff:ff:ff:ff:ff
/ #
/ # hostname
vm6
/ # exit
```

在容器中可以看到 host 的所有网卡，并且连 hostname 也是 host 的。host 网络的使用场景又是什么呢？

直接使用 Docker host 的网络最大的好处就是性能，如果容器对网络传输效率有较高要求，则可以选择 host 网络。当然不便之处就是牺牲一些灵活性，比如要考虑端口冲突问题，Docker host 上已经使用的端口就不能再用了。

Docker host 的另一个用途是让容器可以直接配置 host 网络。比如某些跨 host 的网络解决方案，其本身也是以容器方式运行的，这些方案需要对网络进行配置，比如管理 iptables

#3.bridge 网络（默认）

Docker 安装时会创建一个 命名为 docker0 的 linux bridge。如果不指定 `--network`，创建的容器默认都会挂到 docker0 上。

```
yum install bridge-utils -y
```

#没有运行容器时

```
[root@vm6 ~]# brctl show
```

| bridge name | bridge id | STP enabled | interfaces |
|-------------|-------------------|-------------|------------|
| docker0 | 8000.0242682d0678 | no | |

```
[root@vm6 ~]# docker run --rm --name t1 -p 80:80 -d nginx
ca3866795cd0d842e0ccd3a2dd43d6103f140049deb4f638e2eac7080af76108
#运行一个容器时再查看
[root@vm6 ~]# brctl show
```

| bridge name | bridge id | STP enabled | interfaces |
|-------------|-------------------|-------------|-------------|
| docker0 | 8000.0242682d0678 | no | veth2da790d |

2. 快速配置指南

下面是一个跟 Docker 网络相关的命令列表。

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

```
--b BRIDGE or --bridge=BRIDGE --指定容器挂载的网桥
--bip=CIDR --定制 docker0 的掩码
--H SOCKET... or --host=SOCKET... --Docker 服务端接收命令的通道
--icc=true|false --是否支持容器之间进行通信
--ip-forward=true|false --请看下文容器之间的通信
--iptables=true|false --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 Docker 容器启动（`docker run`）时候指定。在 Docker 服务启动的时候指定则会成为默认值，后面执行 `docker run` 时可以覆盖设置的默认值。

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

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

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

3. 配置 DNS

默认情况下宿主主机 DNS 信息发生更新后，所有 Docker 容器的 dns 配置通过 `/etc/resolv.conf` 文件立刻得到更新。

如果用户想要手动指定容器的配置，可以利用下面的选项。

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

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

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

`--dns-search=DOMAIN` 设定容器的搜索域，当设定搜索域为 `.example.com` 时，在搜索一个名为 `host` 的主机时，DNS 不仅搜索 `host`，还会搜索 `host.example.com`。注意：如果没有上述最后 2 个

选项，Docker 会默认用主机上的 `/etc/resolv.conf` 来配置容器。

4. 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。

#1. 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。

#2. 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`

#3. 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。

#4. 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。

```
[root@vm6 certs]# iptables -nvL
```

```
...
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 iptables 中为两个容器分别添加一条 ACCEPT 规则，允许相互访问开放的端口（取决于 Dockerfile 中的 EXPOSE 行）。当添加了 `--link=CONTAINER_NAME:ALIAS` 选项后，添加了 iptables 规则。

```
[root@vm6 certs]# iptables -nvL
```

```
...
```

```
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。

5.端口映射实现

#1.映射容器端口到宿主主机的实现

默认情况下，容器可以主动访问到外部网络的连接，但是外部网络无法访问到容器。

#2.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。

```
[root@vm6 ~]# iptables -t nat -nL
...
Chain POSTROUTING (policy ACCEPT)
target      prot opt source                destination
MASQUERADE  all  --  172.17.0.0/16          0.0.0.0/0
MASQUERADE  tcp  --  172.17.0.2             172.17.0.2             tcp dpt:5000
..
```

其中，上述规则将所有源地址在 172.17.0.0/16 网段，目标地址为任意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。

#3.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：

```
[root@vm6 ~]# iptables -t nat -nL
...
Chain DOCKER (2 references)
target      prot opt source                destination
RETURN      all  --  0.0.0.0/0             0.0.0.0/0
使用 -p 80:80 时
[root@vm6 ~]# iptables -t nat -nL
...
Chain DOCKER (2 references)
target      prot opt source                destination
RETURN      all  --  0.0.0.0/0             0.0.0.0/0
DNAT        tcp  --  0.0.0.0/0             0.0.0.0/0             tcp dpt:80 to:172.17.0.2:80
```

注意:

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

6.自定义网桥方式一

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。

#1.安装桥接

```
yum install bridge-utils -y
```

#2.创建及配置桥接

#如果服务已经运行，那需要先停止服务，并删除旧的网桥。

```
systemctl stop docker
```

```
ip link set dev docker0 down
```

```
brctl delbr docker0
```

#然后创建一个网桥 `br0`

```
brctl addbr br0
```

```
ip addr add 192.168.5.1/24 dev br0
```

```
ip link set dev br0 up
```

#查看确认网桥创建并启动

```
[root@vm6 ~]# ip addr show br0
```

```
431: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
    link/ether 86:b3:10:23:a8:a7 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global br0
        valid_lft forever preferred_lft forever
    inet6 fe80::84b3:10ff:fe23:a8a7/64 scope link
        valid_lft forever preferred_lft forever
```

#上面命令方式一旦重启就会失效，如果用永久方式可以写成配置文件，操作如下：

```
cd /etc/sysconfig/network-scripts/
```

```
cat>ifcfg-br0<<EOF
```

```
DEVICE="br0"
```

```
ONBOOT="yes"
```

```
TYPE="Bridge"
```

```
BOOTPROTO="static"
```

```
IPADDR="192.168.5.1"
```

```
NETMASK="255.255.255.0"
```

```
GATEWAY="192.192.5.1"
```

```
DEFROUTE="yes"
```

```
NM_CONTROLLED="no"
```

```
EOF
/etc/init.d/network restart
ifconfig
```

#3.修改启动参数

#配置 Docker 服务，默认桥接到创建的网桥上(centos7 上设置无效)

```
echo 'DOCKER_OPTS="-b=br0"' >> /etc/default/docker
systemctl start docker
```

#设置桥接参数

```
echo 'OPTIONS="-b br0"' >> /etc/sysconfig/docker
```

#修改启动脚本，设置启动项桥接 br0，这个时候在 **OPTIONS** 中添加的参数就有效了

#把 “ExecStart=/usr/bin/dockerd” (不同版本会有所不同)，后面添加一个 “**\$OPTIONS**”，

#-/etc/sysconfig/docker 其中 “-” 表示 ignore_errors=yes 的意思

```
cd /usr/lib/systemd/system
cp docker.service docker.service.orig
sed -i '/ExecStart/s/^/#/' docker.service
sed -i '/ExecStart/a\EnvironmentFile=-/etc/sysconfig/docker\nExecStart=/usr/bin/dockerd $OPTIONS' docker.service
grep 'ExecStart' docker.service
```

#重启 docker

```
systemctl daemon-reload
systemctl restart docker
systemctl status docker
```

#4.测试

#新建一个容器，可以看到它已经桥接到了 bridge0 上可以继续用 brctl show 命令查看桥接的信息。
另外，在容器中使用 ip addr 和 ip route 命令来查看 IP 地址配置和路由信息。

```
docker run --rm --name t1 -it -d centos:6.8
```

```
[root@vm6 ~]# brctl show
```

| bridge name | bridge id | STP enabled | interfaces |
|-------------|-------------------|-------------|-------------|
| br0 | 8000.a6ad9c263730 | no | veth3af7283 |
| docker0 | 8000.0242c41b5028 | no | |

#因为我的 dokcer 是 centos6 的，所以用 ifconfig 而不用 ip addr

```
[root@vm6 system]# docker exec -it t1 ifconfig eth0
```

```
eth0      Link encap:Ethernet  HWaddr 02:42:C0:A8:05:02
          inet addr:192.168.5.2  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 b)  TX bytes:0 (0.0 b)
```

#因为我的 dokcer 是 centos6 的，所以用 route 而不用 ip route

```
[root@vm6 system]# docker exec -it t1 route
```

```
Kernel IP routing table
```

| Destination | Gateway | Genmask | Flags | Metric | Ref | Use | Iface |
|-------------|-------------|---------------|-------|--------|-----|-----|-------|
| default | 192.168.5.1 | 0.0.0.0 | UG | 0 | 0 | 0 | eth0 |
| 192.168.5.0 | * | 255.255.255.0 | U | 0 | 0 | 0 | eth0 |

#测试一下能不能联网

```
docker exec -it t1 yum install -y vim
```

7.自定义网桥方式二

上面“6. 自定义网桥方式一”，只是把桥接的名字换了，把 IP 段也改了，如果我们想同时弄几个网段，docker 分类，比如按域名，hualinux.com 域名及二级域名在 192.168.5.0/24 网段。其它的在默认 192.168.3.0/24 网段。操作如下：

```
#创建一个网络叫 hualinux
[root@vm6 ~]# docker network create --driver bridge hualinux
aff3151c3b375038eb0b0acd874298d336b256cd8f73fb48f55bc472a3fde435
```

#如果 ifconfig 没有出现新网卡重启一下网络，centos7.2 试过重启网络不生效，就重启服务器！

```
[root@vm6 ~]# ifconfig br-aff3151c3b37
br-aff3151c3b37: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.18.0.1 netmask 255.255.0.0 broadcast 0.0.0.0
    ether 02:42:b7:ad:c1:6c txqueuelen 0 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
[root@vm6 ~]# brctl show
```

| bridge name | bridge id | STP enabled | interfaces |
|------------------------|-------------------|-------------|------------|
| br-aff3151c3b37 | 8000.0242b7adc16c | no | |
| docker0 | 8000.024273233ffc | no | |

从上面可以看出这里增加了一个“**br-aff3151c3b37**”，正好是 hualinux 的短 ID

```
docker network inspect hualinux
```

```
[root@vm6 ~]# docker network inspect hualinux
[
  {
    "Name": "hualinux",
    "Id": "aff3151c3b375038eb0b0acd874298d336b256cd8f73fb48f55bc472a3fde435",
    "Created": "2017-07-17T10:56:43.65586531+08:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    }
  }
],
```

上面的 IP 地址是自动分配的，需要修改为我们自己的 IP 地址段怎么办？

只需在创建网段时指定 `--subnet` 和 `--gateway` 参数:

#删除 hualinux

```
[root@vm6 ~]# docker network ls
```

| NETWORK ID | NAME | DRIVER | SCOPE |
|--------------|----------|--------|-------|
| b2431ef67591 | bridge | bridge | local |
| da0dcddd5892 | host | host | local |
| aff3151c3b37 | hualinux | bridge | local |
| c68a89dab29a | none | null | local |

#删除可以是 ID 也可以是名字, 只要能看出是唯一的一般都可以

```
docker network rm hualinux
```

#建立指定网段的网关的

```
docker network create --driver bridge --subnet 192.168.5.0/24 --gateway 192.168.5.1 hualinux
```

```
[root@vm6 ~]# ifconfig br-8b3638a842ec
```

```
br-8b3638a842ec: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.5.1 netmask 255.255.255.0 broadcast 0.0.0.0
    ether 02:42:f6:43:83:24 txqueuelen 0 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

#容器要使用新的网络, 需要在启动时通过 `--network` 指定, 比如

```
docker run -it --rm --name t1 --network=hualinux -d busybox
```

```
docker exec -it t1 ifconfig eth0
```

```
docker exec -it t1 ping www.baidu.com -c 3
```

```
[root@vm6 ~]# docker exec -it t1 ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 02:42:C0:A8:05:02
          inet addr:192.168.5.2 Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:0 (0.0 B)

[root@vm6 ~]# docker exec -it t1 ping www.baidu.com -c 3
PING www.baidu.com (14.215.177.38): 56 data bytes
64 bytes from 14.215.177.38: seq=0 ttl=55 time=4.297 ms
64 bytes from 14.215.177.38: seq=1 ttl=55 time=3.784 ms
64 bytes from 14.215.177.38: seq=2 ttl=55 time=3.851 ms

--- www.baidu.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 3.784/3.977/4.297 ms
```

从上图看出能 t1 容器分配了 192.168.5.2 的 ip 地址, 可以 ping 通外界。

到目前为止, 容器的 IP 都是 docker 自动从 subnet 中分配, 我们可以用 `--ip` 指定指定一个静态 IP

```

docker stop t1
[root@vm6 ~]# docker run -it --rm --name t1 --network=hualinux --ip 192.168.5.100 busybox
/ # ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 02:42:C0:A8:05:64
          inet addr:192.168.5.100  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:508 (508.0 B)  TX bytes:0 (0.0 B)

/ # ping www.baidu.com -c 3
PING www.baidu.com (14.215.177.37): 56 data bytes
64 bytes from 14.215.177.37: seq=0 ttl=55 time=3.513 ms
64 bytes from 14.215.177.37: seq=1 ttl=55 time=3.655 ms
64 bytes from 14.215.177.37: seq=2 ttl=55 time=3.618 ms

--- www.baidu.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 3.513/3.595/3.655 ms
/ # exit
[root@vm6 ~]#

```

注：只有使用 `--subnet` 创建的网络才能指定静态 IP。

#hualinux 建时没有指定 `--subnet`，如果指定静态 IP 报错如下：

```

docker network rm hualinux
docker network create --driver bridge hualinux
[root@vm6 ~]# docker network ls

```

| NETWORK ID | NAME | DRIVER | SCOPE |
|---------------------|-----------------|---------------|--------------|
| b2431ef67591 | bridge | bridge | local |
| da0dcddd5892 | host | host | local |
| 389380e4c5ba | hualinux | bridge | local |
| c68a89dab29a | none | null | local |

```

[root@vm6 ~]# docker run -it --rm --name t1 --network=hualinux --ip 192.168.5.100 busybox
docker: Error response from daemon: user specified IP address is supported only when connecting to networks with user configured subnets.

```

附录一、Dockerfile 指令详解

官方链接：<https://docs.docker.com/engine/reference/builder/#dockerfile-examples>

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。

在 Docker 官方的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>"
```

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 声明端口

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，该目录需要已经存在，`WORKDIR` 并不会帮你建立目录。

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`，可以从其项目网站看到进一步的信息：<https://github.com/tianon/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 ps` 看到最初的状态为 (health:starting)：

```
$ docker ps
```

| 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 ps`，就会看到健康状态变化为了 (healthy)：

```
$ docker ps
```

| 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>\n<head>\n<title>Welcome to nginx!</title>\n<style>\n  body {\n    width: 35em;\n    margin: 0 auto;\n    font-family: Tahoma, Verdana, Arial, sans-serif;\n  }\n</style>\n</head>\n<body>\n<h1>Welcome to nginx!</h1>\n<p>If you see this page, the nginx web server is successfully installed and\nworking. Further configuration is required.</p>\n\n<p>For online documentation a\nnd support please refer to\n<a href=\"http://nginx.org/\">nginx.org</a>.<br/>\nCommercial support i\ns available at\n<a href=\"http://nginx.com/\">nginx.com</a>.</p>\n\n<p><em>Thank you for using nginx.</em></p>\n</body>\n</html>\n",
      "Start": "2016-11-25T14:35:37.780192565Z"
    }
  ],
  "Status": "healthy"
}
```

ONBUILD 为他人做嫁衣裳

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

Dockerfile 官方文档: <https://docs.docker.com/engine/reference/builder/>

Dockerfile 最佳实践文档: https://docs.docker.com/engine/userguide/engimage/dockerfile_best-practices/

附录二、私有仓库配置 https

1.生成 ssl 证书

ssl 证书一般是去专业的机构购买如：

沃通（<https://www.wosign.com/>）、Symantec（www.symantec.com/zh/cn/ssl-certificates）、GlobalSign（cn.globalsign.com）、entrust（www.entrust.com.cn）等，当然也可以手工生成不过会提示不安全，实验中可以手工生成，操作如下：

```
cd /etc/docker
mkdir certs
cd certs
openssl req -newkey rsa:2048 -x509 -nodes -days 3560 -out hualinux.crt -keyout hualinux.key
```

#如果下面的输入错了想删除可以用 ctrl+退格键，单按退格键是不行的

```
Generating a 2048 bit RSA private key
.....+++
.....+++
writing new private key to 'hualinux.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:cn 国家代码, 中国的为cn
State or Province Name (full name) [:]:guangdong省份
Locality Name (eg, city) [Default City]:guangzhou市
Organization Name (eg, company) [Default Company Ltd]:flying公司名
Organizational Unit Name (eg, section) [:]:it部门
Common Name (eg, your name or your server's hostname) [:]:192.168.3.76:5000名字
Email Address [:]:flying@163.com 邮箱
```

#上面主要是 192.168.3.76:5000，其它的可以直接回车

```
chmod o-r hualinux.key
```

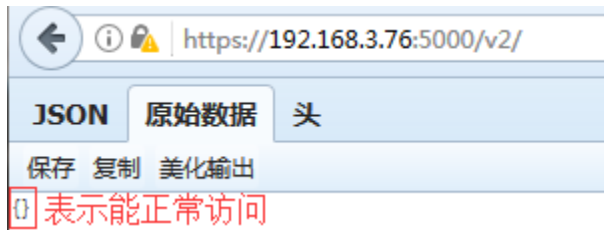
2.运行 registry 容器

```
docker stop hua-reg
docker run -p 192.168.3.76:5000:5000 --name hua-reg-ssl \
-u root \
-v /etc/docker/certs:/certs \
-v /disk1/myregistry:/var/lib/registry \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/hualinux.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/hualinux.key \
-d registry
```

#打开浏览器查看是否能正常访问：<https://192.168.3.76:5000/v2>

#因为是手工生成的证书所以不安全，正式环境中需要去专业的证书机构购买，用域名方式访问。





3.测试 pull 镜像

```
[root@vm6 ~]# docker pull 192.168.3.76:5000/hualinux/hua:nginx
```

```
Error response from daemon: Get https://192.168.3.76:5000/v2/: x509: cannot validate certificate for 192.168.3.76 because it doesn't contain any IP SANs
```

解决:

#修改/etc/pki/tls/openssl.cnf 配置, 在该文件中找到[v3_ca], 在它下面添加如下内容:

```
cd /etc/pki/tls/
```

```
cp openssl.cnf openssl.cnf.orig
```

```
vim openssl.cnf +226
```

```
[ v3_ca ]
```

```
# Extensions for a typical CA
```

```
subjectAltName = IP:192.168.3.76
```

#重新再生成证书, 这个很重要不能用之前的证书了

```
cd /etc/docker/certs/
```

```
rm -f hualinux.*
```

```
openssl req -newkey rsa:2048 -x509 -nodes -days 3560 -out hualinux.crt -keyout hualinux.key
```

```
chmod o-r hualinux.key
```

#重启 docker

```
systemctl restart docker
```

```
docker start hua-reg-ssl
```

#push 一个镜像

```
[root@vm6 certs]# docker pull 192.168.3.76:5000/hualinux/hua:nginx
```

```
Error response from daemon: Get https://192.168.3.76:5000/v2/: x509: certificate signed by unknown authority
```

报错解决:

这是因为客户端宿主主机上没有相应的证书。需要把 registry 所在主机上, 刚生成的证书 /etc/docker/certs/hualinux.crt 复制到/etc/docker/certs.d/192.168.3.76:5000/ca.crt 操作如下:

```
mkdir -p /etc/docker/certs.d/192.168.3.76:5000
```

```
cp /etc/docker/certs/hualinux.crt /etc/docker/certs.d/192.168.3.76:5000/ca.crt
```

#再次测试: 成功

```
[root@vm6 certs]# docker pull 192.168.3.76:5000/hualinux/hua:nginx
```

```
nginx: Pulling from hualinux/hua
```

```
Digest: sha256:88b35551654955e5b15dcbe98806d6f6e5257fa2d9bf7f4f161512f03bcc2b6e
```

```
Status: Image is up to date for 192.168.3.76:5000/hualinux/hua:nginx
```

#开多一台 centos7.2 的 docker，测试一下是否下载，这台主机名为 vm5

#内网 ip 地直来 192.168.3.75

```
mkdir -p /etc/docker/certs.d/192.168.3.76:5000
```

```
cd /etc/docker/certs.d/192.168.3.76\5000/
```

```
scp root@192.168.3.76:/etc/docker/certs.d/192.168.3.76\5000/ca.crt .
```

```
systemctl restart docker
```

```
docker pull 192.168.3.76:5000/hualinux/hua:nginx
```

```
[root@vm5 192.168.3.76:5000]# docker pull 192.168.3.76:5000/hualinux/hua:nginx
nginx: Pulling from hualinux/hua
177c8d195b28: Pull complete
80407d76f511: Pull complete
fa697bbf7113: Pull complete
337271467e53: Pull complete
Digest: sha256:88b35551654955e5b15dcbe98806d6f6e5257fa2d9bf7f4f161512f03bcc2b6e
Status: Downloaded newer image for 192.168.3.76:5000/hualinux/hua:nginx
[root@vm5 192.168.3.76:5000]# docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--------------------------------|--------|--------------|-------------|--------|
| 192.168.3.76:5000/hualinux/hua | nginx | 8f7a11a1630f | 6 days ago | 107MB |
| hello-world | latest | 1815c82652c0 | 3 weeks ago | 1.84kB |

附录三、docker registry v2 api

引用: <http://www.jianshu.com/p/6a7b80122602>

本篇总结 docker registry v2 api 描述和使用 [docker-registry v2](#)

API 清单:

| method | path | Entity | Description |
|--------|-----------------------------------|----------------------------|--|
| GET | /v2/ | Base | Check that the endpoint implements Docker Registry API V2. |
| GET | /v2/<image>/tags/list | Tags | Fetch the tags under the repository identified by name. |
| GET | /v2/<image>/manifests/<reference> | Manifest | Fetch the manifest identified by name and reference where reference can be a tag or digest. A HEAD request can also be issued to this endpoint to obtain resource information without receiving all data. |
| put | /v2/<image>/manifests/<reference> | Manifest | Put the manifest identified by name and reference where reference can be a tag or digest. |
| delete | /v2/<image>/manifests/<reference> | Manifest | Delete the manifest identified by name and reference. Note that a manifest can only be deleted by digest. |
| GET | /v2/<image>/blobs/<digest> | Blob | Retrieve the blob from the registry identified by digest. A HEAD request can also be issued to this endpoint to obtain resource information without receiving all data. |
| DELETE | /v2/<image>/blobs/<digest> | Blob | Delete the blob identified by name and digest |
| POST | /v2/<image>/blobs/uploads/ | Initiate Blob Upload | Initiate a resumable blob upload. If successful, an upload location will be provided to complete the upload. Optionally, if the digest parameter is present, the request body will be used to complete the upload in a single request. |
| GET | /v2/<image>/blobs/uploads/<uuid> | Blob Upload | Retrieve status of upload identified by uuid. The primary purpose of this endpoint is to resolve the current status of a resumable upload. |
| PATCH | /v2/<image>/blobs/uploads/<uuid> | Blob | Upload a chunk of data for the specified upload. |

| | | | |
|--------|----------------------------------|-------------|---|
| | | Upload | |
| PUT | /v2/<image>/blobs/uploads/<uuid> | Blob Upload | Complete the upload specified by uuid, optionally appending the body as the final chunk. |
| DELETE | /v2/<image>/blobs/uploads/<uuid> | Blob Upload | Cancel outstanding upload processes, releasing associated resources. If this is not called, the unfinished uploads will eventually timeout. |
| GET | /v2/_catalog Catalog | Catalog | Retrieve a sorted, json list of repositories available in the registry. |

附录四、参考文章

Docker 技术入门与实战 第 2 版

Docker 容器之 nginx(官方使用配置篇)

<https://www.lvtao.net/config/docker-nginx.html>

服务器使用 docker 部署 nginx 和 php

<http://www.jianshu.com/p/852d14812ba9>

03 搭建 docker 私有仓库

<http://blog.csdn.net/gqtcgq/article/details/51163558>

flying 飞翔

Q:715031064

Q 群: 478477301

2017.07