

# Docker 使用手册

docker 的安装教程请见作者的其他文档，地址：

<https://limaofu.github.io/>

## 0.设置存储 images 和容器的位置

Docker 安装完成后，首先要做的就是设置存储 images 和容器的位置，即我们之后要用到的 docker 镜像及容器的存储位置，docker 的配置文件为：

win server 下： C:\ProgramData\Docker\config\daemon.json

Linux 下： /etc/docker/daemon.json

修改其配置文件，添加一条配置（使用 json 的格式）

```
{  
  
  "data-root": "/data/docker",           //windows 上的 docker 配置文件则写成  
                                           // "data-root": "D:\\docker"  
  
}
```

**操作 docker 的命令：**

要使用管理员或 root 的权限去操作 docker 命令！

**其他说明：**

docker 的镜像是要依赖于宿主系统的内核的，它只是进程上的隔离，所以在某些系统类型或不同的内核上，有些镜像就无法使用了，使用 docker 镜像前一定要清楚它是否适配当前宿主主机系统的内核。

本文档主要是以 centos7 上的 docker-ce 作为讲解案例

作者：Cof-Lee

日期：2020-11-02

## 1. 镜像的使用

# **docker images**

//默认查看当前宿主机上的 docker 镜像

```
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
couchbase            latest             c64844065dcb       6 days ago         1.18GB
registry             latest             2d4f4b5309b1       4 months ago       26.2MB
[root@localhost ~]#
```

docker 镜像全名由 Repository 和 Tag 组成（中间用冒号连接），如上图，显示有 2 个镜像：  
couchbase:latest 和 registry:latest

前面的 Repository 表示仓库，即软件主体名，后面的 Tag 为版本号，latest 表示最新版本，  
当使用镜像时，若不指定 Tag 则默认表示使用最新版本 :latest 版本。

docker images 查看到的镜像的属性有名称，还有镜像 ID 及大小，镜像 ID 是唯一的，由 12  
个十六进制字符表示，大小不唯一，有大有小，大的可以有 10 来个 GB，小的 10 来 MB

# **docker search 软件名**

//默认是连接到 docker hub 上去查找目标软件的  
// docker 镜像

```
[root@localhost ~]# docker search mysql
NAME                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
mysql               MySQL is a widely used, open-source relation...  10115     [OK]
mariadb            MariaDB is a community-developed fork of MyS...  3715     [OK]
mysql/mysql-server  Optimized MySQL Server Docker images. Create...  739
percona            Percona Server is a fork of the MySQL relati...  511     [OK]
centos/mysql-57-centos7 MySQL 5.7 SQL database server                    84
mysql/mysql-cluster Experimental MySQL Cluster Docker images. Cr...  77
centurylink/mysql  Image containing mysql. Optimized to be link...  60
bitnami/mysql      Bitnami MySQL Docker Image                       45
deitch/mysql-backup REPLACED! Please use http://hub.docker.com/r...  41
```

显示列表中，Name 为镜像的名称，OFFICAL 下为[ok]时表示这个镜像是官方构建的，  
AutoMated 下方为[ok]时表示这个镜像其他人构建的  
一般推荐下载官方的镜像

# **docker pull 镜像名**

//下载目标镜像，镜像名要写 search 里看到的名称

```
[root@localhost ~]# docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql
bb79b6b2107f: Pull complete
49e22f6fb9f7: Pull complete
842b1255668c: Pull complete
9f48d1f43000: Pull complete
c693f0615bce: Pull complete
8a621b9dbed2: Pull complete
0807d32aef13: Pull complete
a56aca0feb17: Pull complete
de9d45fd0f07: Downloading [==>
1d68a49161cc: Download complete
dl6d318b774e: Download complete
```

下载镜像时默认也是从 docker hub 去下载的，国内下载的话可能有点慢，可以使用国内的镜像源，可以先看后面的“Docker 镜像仓库管理”章节。

# **docker rmi 镜像名** //默认是删除本地镜像

```
[root@localhost ~]# docker rmi centos:7
Untagged: centos:7
Untagged: centos@sha256:19a79828ca2e505eaae0ff38c2f3fd9901f4826737295157cc5212b7a37
Deleted: sha256:7e6257c9f8d8d4cdf5e155f196d67150b871bbe8c02761026f803a704acb3e9
Deleted: sha256:613be09ab3c0860a5216936f412f09927947012f86bfa89b263dfa087a725f81
[root@localhost ~]#
```

镜像名一定要写全，不写 Tag 的话，默认就是删除 latest 最新版本的

# **docker tag 原镜像名 新镜像名** //给本地镜像打上新的名称，

可以改 Repository 和 Tag 名，原来的镜像并不会被重命名，而是复制了一个副本，给副本打上新的名字

```
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
couchbase            6.6                c64844065dcb       6 days ago         1.18GB
couchbase            latest             c64844065dcb       6 days ago         1.18GB
centos               7                  7e6257c9f8d8       2 months ago       203MB
[root@localhost ~]# docker tag couchbase:latest cb:6.6
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
cb                  6.6                c64844065dcb       6 days ago         1.18GB
couchbase           6.6                c64844065dcb       6 days ago         1.18GB
couchbase           latest             c64844065dcb       6 days ago         1.18GB
centos              7                  7e6257c9f8d8       2 months ago       203MB
[root@localhost ~]#
```

## 2. 镜像究竟是什么？

docker 镜像就是一个没有内核的极简系统加上一个主要的服务软件，比如 mysql 镜像，就是由 linux 基础软件加上 mysql 软件组成的，只是没有内核而已，镜像在使用时，它是共用宿主系统的内核。再比如 centos 镜像，它就是 centos 最基本的一些工具软件集合体，没有内核而已。我们可以在 centos 镜像运行后，进入镜像运行空间（容器）里去添加自己的服务软件，然后再提交生成新的镜像。

镜像在宿主系统上是怎么保存的呢？或者说是以什么形式进行存储的？

默认时，镜像及容器的存储位置为/var/lib/docker 目录下（windows 的 docker 镜像存在 C:\ProgramData\Docker 目录下）或者在我们指定的 data-root 目录下，里面有几个子目录，分别存储不同的数据，有存储镜像/容器实例信息的，也有单纯地存储镜像软件数据的

```
[root@localhost ~]# cd /var/lib/docker
[root@localhost docker]# ls
builder  buildkit  containers  image  network  overlay2  plugins  runtimes
[root@localhost docker]# du -sh containers/
40K      containers/
[root@localhost docker]# du -sh images
du: cannot access 'images': No such file or directory
[root@localhost docker]# du -sh image
2.2M     image
[root@localhost docker]# du -sh overlay2/
2.5G     overlay2/
[root@localhost docker]# du -sh runtimes/
4.0K     runtimes/
```

一个镜像不是以一个单独的文件形式进行存储的,所以我们没办法直接把镜像从当前宿主系统上提走,得导出镜像到某单独文件,再把该文件复制到其他 docker 宿主机上,再导入

# **docker save 镜像名 -o 导出的文件名.tar** //导出镜像到单一 tar 文件

```
[root@localhost ~]# docker save centos:7 -o centos7.tar
[root@localhost ~]#

[root@localhost ~]# ls -lh centos7.tar
-rw-----. 1 root root 202M Oct 30 11:00 centos7.tar
[root@localhost ~]#
```

然后把该 centos7.tar 文件复制到其他装有 docker 服务的宿主机上

# **docker load -i centos7.tar** //从 tar 包导入镜像,导入时不可指定镜像名

```
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
couchbase            latest              c64844065dcb       6 days ago         1.18GB
[root@localhost ~]#
[root@localhost ~]# docker load -i centos7.tar
613be09ab3c0: Loading layer [=====>] 211.1MB/2
Loaded image: centos:7
[root@localhost ~]#
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
couchbase            latest              c64844065dcb       6 days ago         1.18GB
centos                7                  7e6257c9f8d8       2 months ago       203MB
[root@localhost ~]#
```

# **docker images** 可见导入的镜像,可以导入后再修改镜像名称

# **docker run -it --name xxx 镜像名 参数** //运行一个镜像,命名为 xxx

```
[root@localhost ~]# docker run -it --name xxx centos:7 /bin/bash
[root@0845a9305e6a /]#
Display all 655 possibilities? (y or n)
[root@0845a9305e6a /]#
[root@0845a9305e6a /]#
```

本例中运行了 centos:7 这个镜像,运行该镜像后,镜像的运行体就是一个实例或者叫容器。该容器命名为 xxx,可以在容器实例里进行一些操作,-it 选项表示进入容器里并提供一个命令界面,这时宿主系统的 shell 提示符从 root@localhost 变成了 root@0845a9305e6a,这个 0845a9305e6a 就是容器实例的 id

root@0845a9305e6a# **yum install httpd** //在容器实例里安装 httpd 软件

```
[root@0845a9305e6a /]# yum install httpd
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
* base: mirrors.cn99.com
* extras: mirrors.163.com

Installed:
  httpd.x86_64 0:2.4.6-93.el7.centos

Dependency Installed:
  apr.x86_64 0:1.4.8-5.el7          apr-util.x86_64 0:1.5.2-6.el7
  httpd-tools.x86_64 0:2.4.6-93.el7.centos  mailcap.noarch 0:2.1.41-2.el7

Complete!
[root@0845a9305e6a /]#
[root@0845a9305e6a /]# exit
exit
[root@localhost ~]#
[root@localhost ~]#
```

输入 **exit** 后就退出了容器实例,回到了宿主系统里,



# **docker ps -a** //查看刚刚运行过的容器，名为 xxx 的

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

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
0845a9305e6a	centos:7	"/bin/bash"	6 minutes ago	Exited (0) 31 seconds ago		xxx

在容器实例里进行的任何操作都不会保存到镜像里，镜像是静态的数据，不会变的，容器实例是把镜像的数据加载到内存里去运行，在容器里的所有操作都会保存到另外的地方，所以当我们删除容器时，这个容器的所有数据就不存在了，要怎么才能把容器变成静态的数据呢？

可以把刚刚运行过的容器提交生成一个镜像，这样，生成的镜像以后也还可以用，数据也在

# **docker commit -m="描述信息" -a="提交者" 容器 ID 生成的镜像名**

```
[root@localhost ~]# docker commit -m="httpd" -a="coflee" 0845a9305e6a centos7_httpd:v1
sha256:fd303006fcd68344784e5ddd11f02b697c7c01eb635b69fcefefbc25aa3bdecc4
[root@localhost ~]#
[root@localhost ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
centos7 httpd	v1	fd303006fcd6	4 seconds ago	328MB
couchbase	latest	c64844065dcb	6 days ago	1.18GB
centos	7	7e6257c9f8d8	2 months ago	203MB

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

### 3.容器的使用

# **docker run 镜像名** //运行一个镜像，运行时的环境就是一个容器

run 后面的参数：

-d 后台运行

-i 交互式操作

-t 给容器提供一个终端

--name xxx 给容器实例命名，若不指定容器名，则系统会随机给它一个名字

# **docker run -d 镜像名** //让容器在后台运行

# **docker run -it 镜像名** //让容器在前台运行，并提供一个终端与容器交互，这时如果容器有默认的程序在运行的话，我们就可以和这个程序交互了，若没有默认程序运行或该程序不与用户交互，则我们就什么也做不了

# **docker run -it 镜像名 /bin/bash** //最后的/bin/bash 是传给容器的参数，表示运行这个程序与我们交互，即 linux 系统的 bash，然后就可以在 bash 命令行操作了

如果是 windows 上的容器，即传入 **cmd.exe** 参数

\*\*在容器里直接输入 **exit** 后，就退出容器了，容器就停止运行了！

```
# docker ps           //查看正在后台运行的容器
# docker ps -a        //查看所有运行过的容器及正在后台运行的容器
**容器的 id 与镜像的 id 是不一样的，别搞混了
```

```
# docker stop 容器 id      //停止正在运行的这个容器，容器 id 可以换成容器名
# docker start 容器 id     //启动一个已停止的容器
# docker restart 容器 id   //重启一个容器
```

## 进入容器并与之交互

如果容器在后台运行，即 **run -d** 的，如何再进入这个容器并与之交互呢？

```
# docker exec -it 容器 id      //进入正在后台运行的容器，并与之交互
```

....

```
>exit           //在容器里输入 exit 后退出容器
```

\*使用 **docker exec** 进入容器后，再 **exit** 退出的话，该容器仍在后台运行，并不会停止

```
# docker rm -f 容器 id      //删除一个已经停止运行的容器
# docker container prune    //删除所有已停止运行的容器，停止的容器也是要占用
                             系统的存储空间，确定不用了的容器就删除吧
```

## 容器信息查看：

```
# docker inspect 容器 id/容器名      //查看容器的详细信息，如状态，配置
```

```
# docker logs 容器 id/容器名          //查看容器里的标准输出信息
```

```
[root@localhost ~]# docker logs 585e4d2d8da2
Starting Couchbase Server -- Web UI available at http://<ip>:8091
and logs available in /opt/couchbase/var/lib/couchbase/logs
Starting Couchbase Server -- Web UI available at http://<ip>:8091
and logs available in /opt/couchbase/var/lib/couchbase/logs
Starting Couchbase Server -- Web UI available at http://<ip>:8091
and logs available in /opt/couchbase/var/lib/couchbase/logs
[root@localhost ~]#
```

```
# docker logs -f 容器 id/容器名      //不断地输出容器里的标准输出信息，
//就像 top 命令那样，不退出显示界面，动态输出信息按 Ctrl+C 退出
```

```
# docker export 容器 id > xx.tar      //导出容器快照到 xx.tar 文件里
# docker import xxx.tar 镜像名        //导入容器快照为镜像，必须指定镜像名
*导入容器快照后，生成的是镜像，而不是运行的容器
```

## 4.端口映射:

容器里的网络环境默认为 nat 的,也就是所有容器自己会获得一个私有 ip,然后它访问外部时是要转为宿主系统的 ip,外部没法直接访问容器里的 ip 和端口号,得做个 dNat 端口映射

# **docker run -d -p 宿主端口:容器端口 镜像名** //用 -p 指定端口映射

例如:

**-p 9999:80** //把宿主上的 9999 端口映射到容器里的 80 端口

端口组映射: (端口数量要一致)

**-p 800-810:900-910** //把宿主上的 800 到 810 端口组映射到容器里的 900-910 端口

-p xx:xx 可有多个,做**多个端口的映射**:

**-p 100:100 -p 800-820:800-820 -p 999:999**

默认是 tcp 的端口,可在端口后加 /udp 表示使用 udp 协议的端口

**-p 53:53/udp**

# **docker port 容器 id/容器名** //查看容器的端口映射情况

## 5.目录映射:

容器里的数据是保存的临时的容器存储位置的,如果不小心把容器删除或清理了,那数据就再也没有了,而且数据直接放在容器里,也不方便管理和备份。最好是做个目录映射,把宿主系统上的某个目录映射到容器里的某目录下,然后容器的服务指定存放数据的目录到映射的目录下就行了。这些容器产生的数据就会直接保存到宿主系统的某目录下。

# **docker run -d -v /宿主目录:/容器里的目录 镜像名** //要在容器运行时指定映射的目录,2 个目录中间是一个冒号,冒号左右无空格

```
root@localhost ~]# docker run -d -v /cb_data:/opt/couchbase/var couchbase:6.6
```

如果是 windows 系统,则路径写成 **-v D:\xxdir:/opt/xxdir**

-v 参数指定映射的目录,最好是写在 docker run 命令后,中间不要隔太远

# **docker cp 宿主目录/xx 文件 容器 id:/xxfile** //从宿主目录复制某文件到容器里

# **docker cp 容器 id:/xxxx 宿主目录/xxfile** //从容器里复制文件到宿主系统下

## 6.容器占用资源限额

容器在使用时是尽可能地占用宿主系统的所有可用资源,如果有多个容器在运行时,宿主系统的 docker 进程会自动调配,我们也可以手动指定某容器运行时可用的最大资源

内存限额:

在 `docker run` 后指定 `-m` 和 `--memory-swap=` 参数

```
# docker run -d -m 500M 镜像名 //指定运行的容器最多只能用宿主系统的 500M 内存
```

```
# docker run -d -m 500M --memory-swap=600M 镜像名 //表示最多使用 500M 内存，内存和 swap 一共最多 600M，也就是 100M 的 swap 空间
```

```
# docker run -d -m 500M --memory-swap=-1 xxx //表示容器对 swap 的使用没有限制
```

\*注意，windows 系统没有 swap 空间，所以不能使用 `--memory-swap` 参数

## CPU 优先使用权:

在 `docker run` 后指定 `--cpu-shares` 参数

```
# docker run -d --cpu-shares 2048 镜像名 //表示容器使用 cpu 的优先权值为 2048，默认时每个容器都是 1024 的优先权值，2048 则表示这个容器有 2 倍的机会获得 cpu 的使用权，这个 --cpu-shares 参数只指定使用权的先后，而不指定能使用多少个 cpu
```

## CPU 使用周期控制:

在 `docker run` 后指定 `--cpu-quota` 及 `--cpu-period` 参数

`--cpu-quota` 指定容器对 cpu 的使用要在多长时间内做一次重新分配

`--cpu-period` 指定在一个使用周期内 (`--cpu-quota`) 最多有多少时间来给这个容器运行

```
# docker run -d --cpu-quota 200000 --cpu-period 100000 镜像名
```

表示在 200 000 微秒内至少有 100 000 微秒的时间给这个容器使用 cpu  
1000 微秒为 1 毫秒

## CPU 个数限制:

在 `docker run` 后指定 `--cpus=` 参数

```
# docker run -d --cpus=4 镜像名 //这个容器最多可以使用 4 个 cpu，可指定小数点的，如 --cpus=1.5 个
```

## 磁盘读写优先级:

在 `docker run` 后指定 `--blkio-weight` 参数

```
# docker run -d --blkio-weight 1000 镜像名 //指定容器使用磁盘的优先级  
默认优先级为 500，1000 则表示此容器有 2 倍的机会使用宿主系统的磁盘
```

## 磁盘读写速度的限制:

在 `docker run` 后指定 `--device-read-bps` 和 `--device-write-bps`

或 `--device-read-iops` 和 `--device-write-iops`

```
# docker run -d --device-read-bps /dev/sda:50MB --device-write-bps /dev/sda:50MB 镜像名
```



表示这个容器对宿主系统的/dev/sda 磁盘读速度最多为 50MB/秒，写速度最多为 50MB/秒  
一般一块机械硬盘读写速度为 200MB/s 左右，多块硬盘做了 raid 可能有 800MB/s 左右，具体得到实际中的宿主机去测试一下

```
# docker run -d --device-read-iops /dev/sda:1000 --device-write-iops /dev/sda:1000 镜像名
```

表示这个容器对宿主系统的/dev/sda 磁盘的读操作次数最多为 1000 次/秒，写 1000 次/秒，至于每次读写多少数据，就不知道了

## 7. Docker 网络模式

docker 的网络模式没有太多的可玩性，不深入讲了

默认给容器用的网络环境为 nat 的环境，当容器访问外部环境时，docker 进程把容器的 ip 转为宿主系统的 ip，如果宿主系统有多个网卡（ip），则随机转换成任一个。当然也可以在做容器的端口映射时指定要使用的宿主系统的 ip

```
# docker run -d -p 宿主 ip:port:port 镜像名
```

windows 上的 docker 是在安装后，初次启动时就在宿主系统上创建了一个虚拟网卡，名为 vEthernet(nat)，然后共享宿主系统的网卡到这块网卡上，然后每创建一个容器就给容器分配一个临时虚拟网卡，此容器的网卡以 vEthernet(nat)为网关，这样当容器访问外部环境时，就以宿主机的真实的网卡去访问，做了端口映射的话，就由 docker 进程去转换

Linux 上的 docker 是只有当 docker 服务启动时才会在系统里创建一个临时的网桥，名为 docker0，当宿主系统重启后，如果 docker 服务未启动则不会有这个 docker0 网桥  
然后每创建一个容器就给容器分配一个临时虚拟网卡，在宿主系统上也能看得到这些容器的网卡。docker 进程再把容器的虚拟网卡放入这个 docker0 网桥里，网桥就是一个交换机，docker0 可以认为是这个交换机的 svi 管理 ip。容器的网卡以这个交换机的 svi 管理 ip 为网关。其他的网络地址的转换就是在宿主系统上使用 iptables 创建相应的 nat 规则，临时的。所以我们启动 docker 的容器时，若做了端口映射，就千万不要保存宿主系统上的 iptables 配置，不然这些临时的端口映射规则就被保存了。下次想再运行容器，可能就会有冲突。

linux 下的 docker 默认还有一种网络模式，名为 host，表示容器就直接使用宿主系统的网卡，不另外给容器创建一个临时的虚拟网卡，所以容器要用到的端口不能和其他已经被使用的端口相重复。这个模式一般只用于当宿主系统上仅运行一个 docker 容器时。且当这个容器要做的端口映射非常多，不好写，或者不太清楚这个容器到底要做哪些端口映射。

运行容器时如何指定其要使用的网络模式呢？

可在 docker run 后指定--network 参数

```
# docker run -d --network bridge 镜像名 //表示使用 bridge 模式，就是默认缺省的 Nat 模式，在 windows 上要写成--network nat
```

```
# docker run -d --network host 镜像名 //表示使用 host 模式，共用宿主 ip
```

## 8.Dockerfile 的编写

Dockerfile 就是一个 根据某原始镜像生成另一个新镜像的配置文件，可以在原始镜像的基础上加入其他的文件或添加其他软件，修改启动入口程序等就生成了新的镜像了。这个只要是方便自动生成新镜像，如果我们运行基础镜像，再进入容器去操作的话，可能操作过程中会出现一些错误，而导致生成的镜像比较大，而且有时步骤非常多，人工操作比较麻烦，一般是在我们自己写的程序项目目录下创建一个 Dockerfile 文件，名字就叫 Dockerfile

内容示例如下：

```
FROM xxx/xxx:latest
WORKDIR /app
COPY ./xx /app
RUN xxxxxxCMDxxxx
CMD xxx
ENTRYPOINT xxx
```

### Dockerfile 内容讲解：

dockerfile 的内容就是构建新镜像的全过程以及指定新镜像的某些运行时参数

```
FROM *** //表示使用***指定的镜像作为基础镜像，之后的操作是在此镜像运行时的容器里进行的
LABEL description="xxx" //表示给生成的新镜像加上某个标签，如描述内容等
WORKDIR /dir //指定容器内的工作目录，在基础镜像运行时的容器里
COPY 宿主目录文件 /dir //复制宿主里的某文件或目录到容器里的/app
//这里的宿主目录为相对路径，是由构建时最后指定的
RUN xxx //RUN 指定在容器里运行的命令，比如安装或编译某软件等
//RUN 命令可以有多行
ARG xxx=xxx //设置容器里的环境变量，仅在构建时有效
ENV xxx=xxx //设置生成的新镜像的环境变量，在新镜像运行时有效
ADD xxx xxx //作用同 COPY，只是 add 可以从 url 下载文件，而 copy 不行
VOLUME /xx //定义匿名数据卷
EXPOSE 80,443 //声明要导出的端口，此镜像服务要用的端口，仅仅是声明，给用户看的，实际上可能用到的不止是这几个端口
USER user:group //指定在容器里执行后续命令的用户/用户组
ENTRYPOINT xxx //在生成的新镜像运行时的入口程序
CMD xxx //在生成的新镜像运行时的入口程序的变参
```

### 根据 dockerfile 构建新镜像：

dockerfile 写完后，就可以复制到某项目目录下，一个目录下只能有一个 dockerfile 文件，然后在此目录下运行构建命令：

```
# docker build --rm -t 生成的镜像名 . //注意末尾有个. 点
--rm 表示构建结束后，删除中间环节的容器，默认是--rm=true
-t 表示指定生成的新镜像名，可以有多个名字，比如：
# docker build --rm -t appname:v1 appname:latest .
```

表示生成的当前 app 版本为 v1，且复制一个副本作为最新版本

`docker build` 末尾的点表示构建镜像时的宿主环境位置，比如在 `dockerfile` 里这样写：

```
COPY cert /app/cert //表示把构建环境指定的宿主目录下的 cert 子目录文件复制到当前容器的/app/cert 目录下
```

这个 `cert` 就是相对于 `#docker build xx` 末尾的点，末尾的点表示当前目录，即我们在执行 `docker build` 命令时所处的宿主系统的目录，假如是 `/myPro`

则上面的 `copy cert /app/cert` 的最终效果就是把 `/myPro` 目录下的 `cert` 子目录里的文件复制到容器里的 `/app/cert` 里，当然，此时的 `dockerfile` 文件也是在 `/myPro` 目录下

所以当我们不处于宿主系统的 `/myPro` 目录时，若要构建此目录下的 `dockerfile`，可以这样执行：

```
# docker build --rm -t -f /myPro/dockerfile appname:v1 /myPro
```

## 新镜像构建的原理：

`dockerfile` 里的每一行命令执行后就保存为一层新的数据，直到所有命令运行完成，这样多层数据打包成一个镜像，每一层数据在构建完成后，就不会再改变，比如上一层的数据，就算在下一个命令把它删除了，其实它也是存在镜像里的，只是在下一层打个标记（表示删除），所以如果在某一层的临时数据比较多的话，就得在该层完成之后把临时数据删除

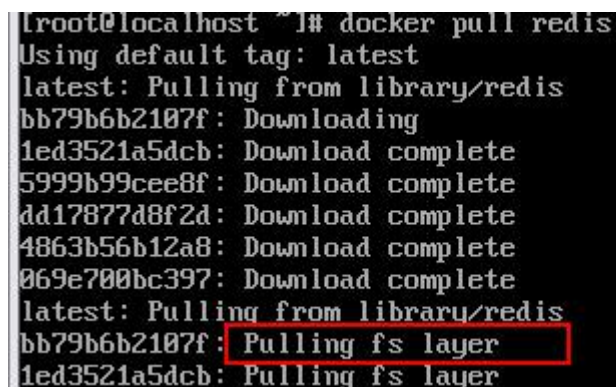
一般 `RUN` 命令后面的一行命令为一层，所以最好是把清理当前层无用数据的命令也放在这一行，命令之间用 `&&` 符号连接，在 `dockerfile` 里也可以用反斜杠断行，示例：

```
RUN yum install -y xxx1 xx2 && \
    其他命令 && \
    yum remove -y xx1 && \
    yum clean all
```

以上为一行，表示用 `yum` 安装某几个工具，用完其中的工具后再删除，因为 `yum` 在安装软件时默认是会缓存 `rpm` 安装包到本地的，所以最后得 `clean all`

以上的操作得放在一个 `RUN` 命令后，才算是同一层，这样中间过程的临时数据才能有效清除，如果 `yum clean all` 放在另一个 `RUN` 命令后，则只是对上层数据打上“删除”的标记，真实的数据并未删除，还在镜像层里，这会导致生成的镜像总大小很大。

我们在下载镜像时也是一层一层地下载的，也就是说镜像在构建生成时是一层一层地，在存储时也是一层一层的，每一个基础数据层都有它唯一的 id



```
[root@localhost ~]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
bb79b6b2107f: Downloading
1ed3521a5dcb: Download complete
5999b99cee8f: Download complete
dd17877d8f2d: Download complete
4863b56b12a8: Download complete
069e700bc397: Download complete
latest: Pulling from library/redis
bb79b6b2107f: Pulling fs layer
1ed3521a5dcb: Pulling fs layer
```

```

[root@localhost docker]# cd overlay2/
[root@localhost overlay2]# ls
0c834de301a991c5ef510a6a7e8a56be844d9edc0490e0a3d61eebfa1a8527c5
2f5b8e2906950e93108c3c00e133dc2c8d0db3964f0cb74872d9527ba1baa7d2
2f5b8e2906950e93108c3c00e133dc2c8d0db3964f0cb74872d9527ba1baa7d2-init
563d13f0ec8f277dff7d482d1701bdc613559020d98687dd7b92699c20d7fd2b
719187a4254fffd43cdbf08aa608ca8fb15eb09c715e8f54ac69398325d50f41e
7301d83f36883905061d8a4c0cafc4f3c77cd70d9f5c803eb60ba3088e83fff9
83ec0cd73987630947568f46a381cdb24ee0fe1fad4575429f707197dccb0ff6
917ebc6c30855baf1fb9294fc800754ea58f9a3db1ca9e3fb8ba818f62e49724

```

所以在我们下载同一镜像软件的不同版本时，如果本地已有旧版本，则下载的新版本镜像一般只会下载更新的层，不变的层就不下载了，就用本地的。

#### 附：dockerfile 构建 dotnet 应用

```

FROM mcr.microsoft.com/dotnet/core/sdk:2.1 AS build-env
WORKDIR /app
COPY *.csproj ./
RUN dotnet restore
COPY . ./
RUN dotnet publish -c Release -o out
FROM mcr.microsoft.com/dotnet/core/aspnet:2.1 AS runtime
WORKDIR /app
COPY --from=build-env /app/out ./
ENTRYPOINT ["dotnet", "myasp-netapp.dll"]

```

#### 构建时：

```

# cd /myasp-netappDir //dockerfile 保存在此，项目目录
# docker build --rm -t myaspApp:v1 .

```

#### 构建过程：

首先以 xxx 为开发环境容器，把宿主机上的/myasp-netappDir 里的\*.csproj 文件们复制到构建容器里的/app 里，再 dotnet restore，再把宿主机上的/myasp-netappDir 里所有文件复制到容器里的/app 里，执行发布操作，生成的 dll 放到/app/out 目录里

然后另取一原始镜像作为 runtime 容器，从 build-env 容器里把/app/out 目录下所有文件复制到当前容器的/app 里，设置入口程序，将此容器提交生成最终镜像。然后清理中间过程的容器。

## 9. 镜像仓库管理

我们在使用 `docker search` 和 `docker pull` 命令时操作的镜像都是存放在 `docker` 的公共仓库里，仓库：Registry，默认的仓库为 `https://hub.docker.com`

即宿主系统上的 `docker` 程序会访问这个远程仓库，如果网络状况不佳，则下载速度很慢，我们可以修改 `docker` 的配置文件，让它访问国内的其他 `docker` 镜像仓库

```
# vi /etc/docker/daemon.json
```

```
{
  ..
  "registry-mirrors": [
    "https://xxx.com",
    "http://xxxxxxx.com"
  ],
  ..
}
```

保存，

\*注意，配置要使用 `json` 的规范格式，

`registry-mirrors` 后的地址可以指定多个，国内的仓库地址有：

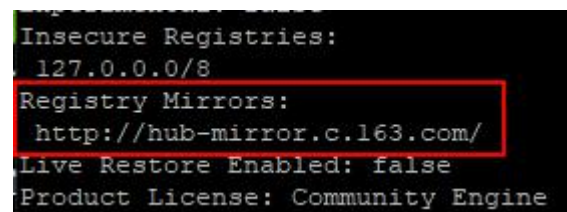
<https://registry.docker-cn.com>

<http://hub-mirror.c.163.com>

然后先查看有无正在运行的容器，先停止运行，再重启 `docker` 服务

最后查看一下：

```
# docker info
```



```
Insecure Registries:
 127.0.0.0/8
Registry Mirrors:
 http://hub-mirror.c.163.com/
Live Restore Enabled: false
Product License: Community Engine
```

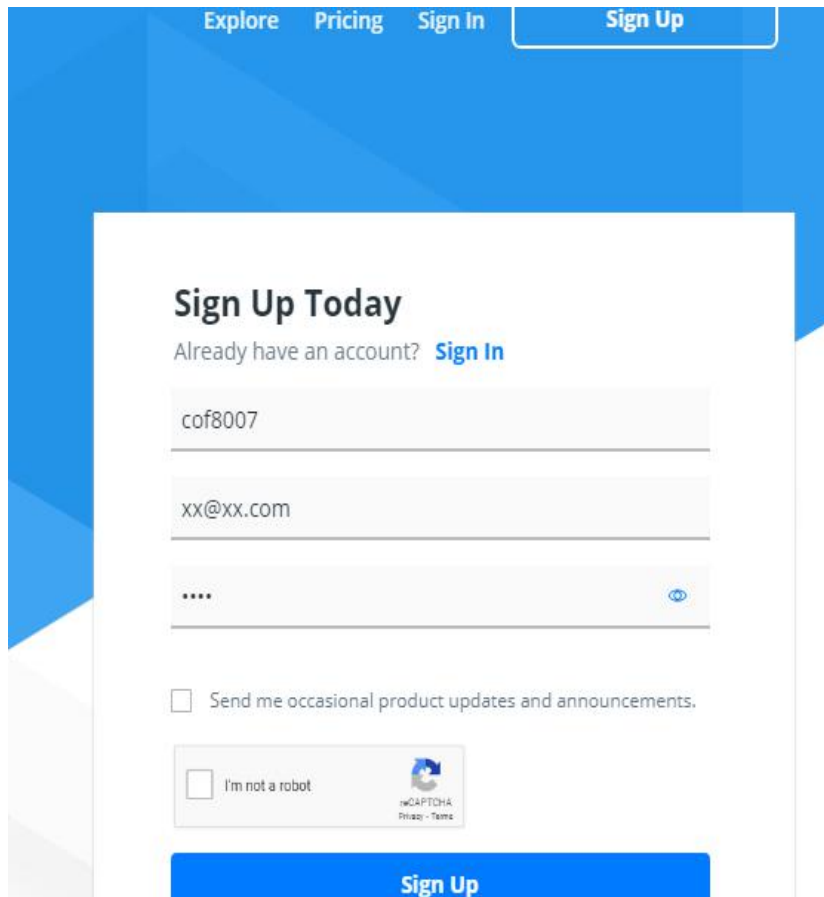
可见配置生效

### 在宿主系统上登录到 `docker` 远程仓库：

首先要在默认的仓库上创建一个账号，比如默认在 `hub.docker.com` 时，我们先在 `https://hub.docker.com` 上创建自己的账号

再创建一个或多个仓库名（即软件名），之后在宿主系统上使用 `docker` 登录这个账号，上传宿主机本地上的镜像到远程账户里





在宿主系统上登录账号

# docker login //默认是登录到 hub.docker.com 仓库，与我们配置的 registry-mirrors 指定的地址无关，那个只是默认下载镜像的地址，而不是登录公共仓库的

```
[root@localhost ~]# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, go to https://docs.docker.com/engine/reference/commandline/login/#credentials-store to get one.
Username: cof8007
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

输入用户名和密码就顺利登录了

然后查看一下，

docker info

```
ID: 1420:RO3B:D131:QEC2:W4NR:43CK:ELIK:EL3F:00M
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Username: cof8007
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
127.0.0.0/8
```

可见当前登录到的账户

再进行 `docker push` 操作就能把本地的镜像上传到 `hub.docker.com` 的 `cof8007` 账户里

# `docker logout` //退出登录

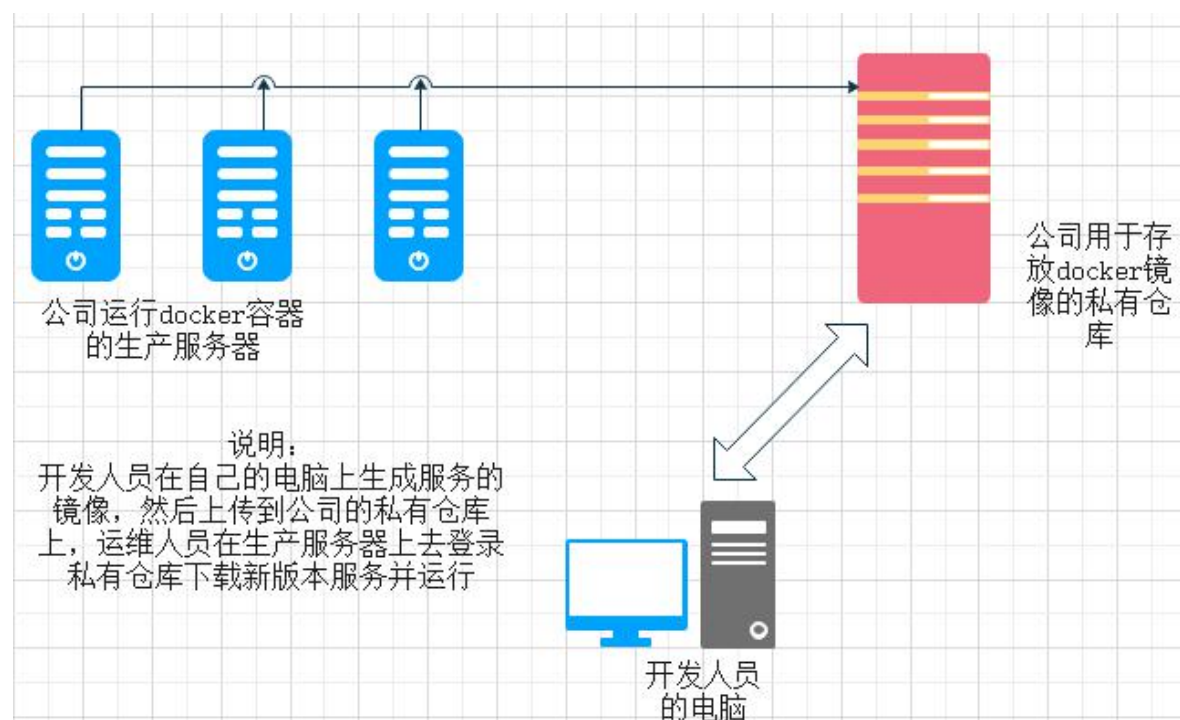
```
[root@localhost ~]# docker logout
Removing login credentials for https://index.docker.io/v1/
[root@localhost ~]#
```

具体的不想多讲（国内连接 `docker hub` 比较慢），除非你有开源精神，可以做一些开源的镜像然后上传到自己的 `docker hub` 账号上，并 `public` 开放。

## 10.搭建本地 docker 镜像仓库

有时个人或公司的开发人员把 某服务做成了 `docker` 镜像，然后这个镜像要存放在什么地方呢？总得有个地方去统一保管吧，但由于网络的限制，可能无法正常使用默认的 `hub.docker.com` 这个仓库，所以我们可以自己搭建一个公司本地的 `docker` 镜像仓库。

主要运作流程：



当然，这只是一个简单的流程

本章侧重于私有仓库的搭建

“`docker` 镜像仓库”也是一个服务，可以在 `docker` 公共仓库下载这个服务镜像（名为 `registry`），然后在“要做成私有仓库”的目标服务器上运行这个镜像，再做一些配置和目录映射即可。

目标服务器上要先安装 docker 服务：

# **docker pull registry** //下载 registry 这个镜像

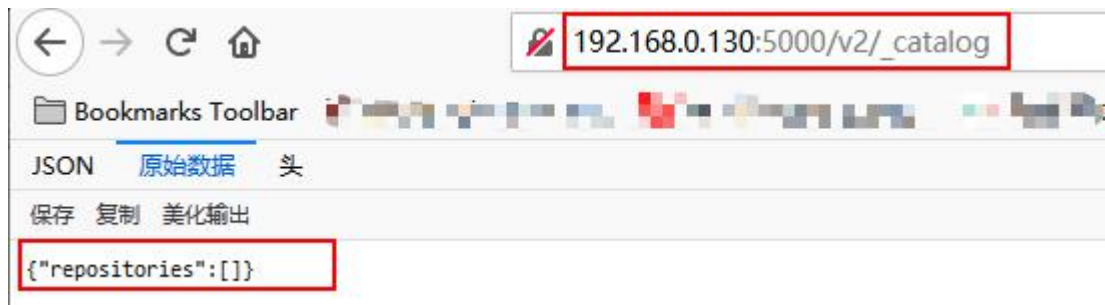
```
[root@localhost ~]# docker pull registry
Using default tag: latest
latest: Pulling from library/registry
cbdbe7a5bc2a: Extracting [=>
```

这个 registry 服务软件也是 docker 公司开发的，并且是开源的，它运行后，我们就可以上传镜像到它那里，它会把我们上传的镜像保存到容器里的/var/lib/registry 目录下，所以我们得在目标服务器上做个目录映射，让数据存在宿主系统的磁盘里，而且此服务默认端口号为 5000，所以还要做个端口映射

# **docker run -d -v /localregistry:/var/lib/registry -p 5000:5000 registry**

运行 registry 服务，将宿主系统的/localregistry 目录映射到容器里的/var/lib/registry 容器运行后，在其他电脑上用浏览器访问目标服务的 5000 端口，url 为：

http://x.x.x.x:5000/v2/\_catalog



可以查看到这个私有仓库里面的 repositories 为空，说明什么都没有，我们之后就可以上传镜像到这个仓库里了，到时再查看就有了。

\*不过，为了安全，一般也不建议用 Http 的明文协议，得给这个 registry 服务配置 ssl 和创建用户。

首先自己申请一个 ssl 证书或自签名生成一个 ssl 证书，域名可以随便取，比如 cb1.xxx.com

cb1.xxx.com.crt	2020/11/2 9:43	X.509 Certificate	2 KB
cb1.xxx.com.key	2020/11/2 9:43	KEY 文件	2 KB

颁发给: cb1.xxx.com

颁发者: cb1.xxx.com

有效期从 2020/11/2 到 2030/11/2

然后可以使用 Httpd-tools 的 htpasswd 命令创建用户

```
[root@localhost ~]# yum install httpd-tools
```

# **yum install httpd-tools**

# **htpasswd -Bc .htpasswd cof** //保存文件名为 .htpasswd，创建用户 cof

```
[root@localhost ~]# htpasswd -Bc .htpasswd cof
New password:
Re-type new password:
Adding password for user cof
[root@localhost ~]#
```

最后把 cb1.xxx.com.crt、cb1.xxx.com.key 及.htpasswd 这三个文件放到宿主系统同一目录下，比如/cbssl 目录

```
[root@localhost ~]# mkdir /cbssl
[root@localhost ~]# mv cb1.xxx.com.crt /cbssl
[root@localhost ~]# mv cb1.xxx.com.key /cbssl
[root@localhost ~]# mv .htpasswd /cbssl
[root@localhost ~]#
[root@localhost ~]# ls /cbssl
cb1.xxx.com.crt  cb1.xxx.com.key
[root@localhost ~]# ls -a /cbssl
.  ..  cb1.xxx.com.crt  cb1.xxx.com.key  .htpasswd
```

最后启动 registry 容器

```
# docker run -d -p 5000:5000 --name localregistry --restart=always \
-v /cbssl:/cbssl \
-v /localregistry:/var/lib/registry \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/cbssl/cb1.xxx.com.crt \
-e REGISTRY_HTTP_TLS_KEY=/cbssl/cb1.xxx.com.key \
-e REGISTRY_AUTH_HTPASSWD_PATH=/cbssl/.htpasswd \
-e REGISTRY_AUTH_HTPASSWD_REALM="cb1.xxx.com" \
registry //最后的镜像名不要忘了
```

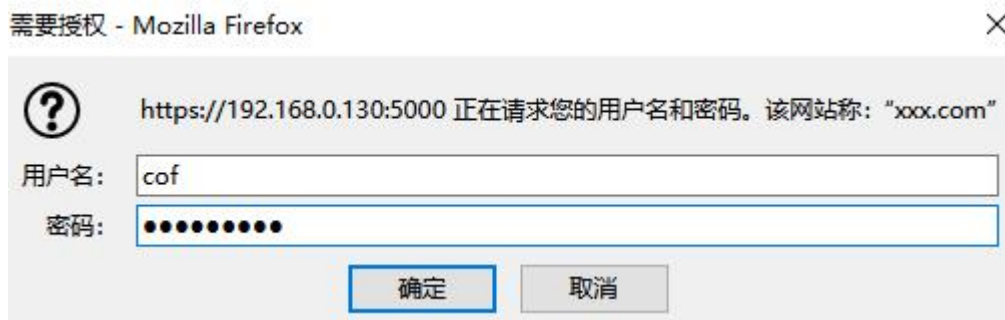
以上为一行，最后的\表示可以换行，在 windows 命令行里使用键盘左上角的`反撇号去换行（这个`反撇号英文名叫 grave 或 backtick）

```
[root@localhost ~]# docker run -d -p 5000:5000 --name localregistry --restart=always -v /cbssl:/cbssl -v /localregistry:/var/lib/registry -e REGISTRY_HTTP_TLS_CERTIFICATE=/cbssl/cb1.xxx.com.crt -e REGISTRY_HTTP_TLS_KEY=/cbssl/cb1.xxx.com.key -e REGISTRY_AUTH_HTPASSWD_PATH=/cbssl/.htpasswd -e REGISTRY_AUTH_HTPASSWD_REALM="cb1.xxx.com" registry
```

运行成功后，在其他电脑上访问 https://x.x.x:5000/v2/\_catalog

然后提示不安全的连接，因为证书是我们自签名的，可以信任它，继续访问

然后就提示要输入用户名和密码了，用户名就是前面创建的那个 cof





到此本地的私有 docker 镜像仓库服务器就配置好了，

那么客户端（其他计算机上的 docker 服务）怎么用它呢？

首先也要让客户端的 docker 信任 registry 服务器的 ssl 证书，

`# vi /etc/docker/daemon.json` //修改 docker 配置文件

```
st ~]# vi /etc/docker/daemon.json
```

添加一条配置：

`"insecure-registries": [ "192.168.0.130:5000" ]` //信任此 registry 服务器

```
[root@localhost ~]# cat /etc/docker/daemon.json
{
  "registry-mirrors": ["http://hub-mirror.c.163.com"],
  "insecure-registries": ["192.168.0.130:5000"]
}
```

保存，

重启 docker 服务（前提是确保当前运行的容器都已正确停止）

`# systemctl restart docker`

之后就可以使用了，

在客户端，登录我们的私有 registry

`# docker login 目标 registry 的 ip:5000`

```
[root@localhost ~]# docker login 192.168.0.130:5000
Username: cof
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos7_httpd       v1                 fd303006fcd6       2 days ago         328MB
couchbase            latest             c64844065dcb       9 days ago         1.18GB
centos               7                 7e6257c9f8d8       2 months ago       203MB
```

输入用户名和密码后就成功登录了，再查看当前本地的镜像

把要上传到 registry 的镜像重新打个 tag，比如我们要上传 centos:7 到私有 registry 服务器

就把 centos:7 打上 `x.x.x.x:5000/新镜像名`

//x.x.x.x 指代 registry 服务器的 ip

`# docker tag centos:7 192.168.0.130:5000/centos:7`



```
[root@localhost ~]# docker tag centos:7 192.168.0.130:5000/centos:7
[root@localhost ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
centos7_httpd	v1	fd303006fcd6	2 days ago
couchbase	latest	c64844065dcb	9 days ago
192.168.0.130:5000/centos	7	7e6257c9f8d8	2 months ago
centos	7	7e6257c9f8d8	2 months ago

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

再使用 push 命令上传

直接 docker push 目标镜像名 就行，这时的目标镜像名就是 x.x.x.x:5000/centos:7

# **docker push 192.168.0.130:5000/centos:7** //前提是已经登录了目标 registry

```
[root@localhost ~]# docker push 192.168.0.130:5000/centos:7
The push refers to repository [192.168.0.130:5000/centos]
613be09ab3c0: Pushed
7: digest: sha256:fe2347002c630d5d61bf2f28f21246ad1c21cc6fd343e70b4cfl5102f8711a9 size: 529
[root@localhost ~]#
```

最后登录 [https://x.x.x.x:5000/v2/\\_catalog](https://x.x.x.x:5000/v2/_catalog) 看一下

如下图，repositories 下有一个名为"centos"的仓库，说明刚刚上传成功了



退出登录:

# **docker logout x.x.x.x:5000**

```
[root@localhost ~]# docker logout 192.168.0.130:5000
Removing login credentials for 192.168.0.130:5000
```

小结:

这个 registry 服务没有管理界面，上传的镜像也只有 repository 名，无 tag 名，所以以后要是想从这个 registry 下载某镜像都得问一下其他人之前上传的镜像名 tag 是多少，或者每次上传时都是只上传 latest 版本的。

本例只是让大家了解这个 registry 仓库的基本运作原理，如果想更方便地使用，可以搭建 harbor，harbor 是一个有图形界面管理的镜像仓库套件，它含好几个镜像（软件），搭建比较麻烦，当然了，使用起来就简单了。