### 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\_data" //windows上的docker配置文件则写成

// "data-root": "D:\\docker\_data"

}

**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

更新日期：2020-12-02

**1.镜像的使用**

# **docker images** //默认查看当前宿主机上的docker 镜像



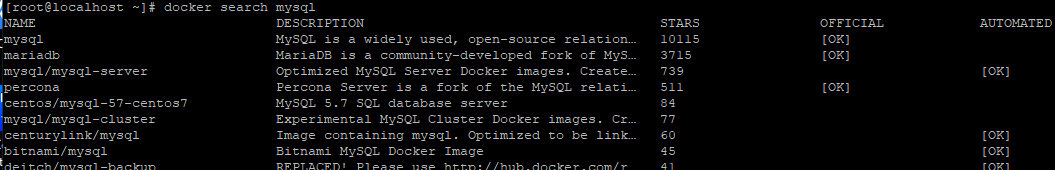
docker镜像全名由Repository和Tag组成（中间用冒号连接），如上图，显示有2个镜像：

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甚至几百KB

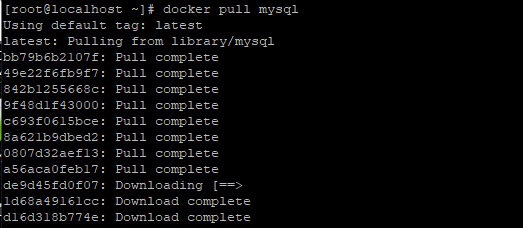
# **docker search 软件名** //默认是连接到docker hub上去查找目标软件的

// docker镜像

显示列表中，Name为镜像的名称，OFFICIAL下为[ok]时表示这个镜像是官方构建的，AutoMated下方为[ok]时表示这个镜像是其他人构建的

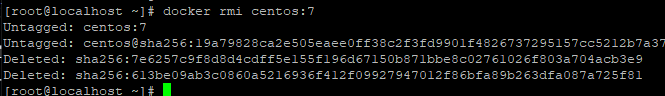
一般推荐下载官方的镜像

# **docker pull 镜像名** //下载目标镜像，镜像名要写search 里看到的名称



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



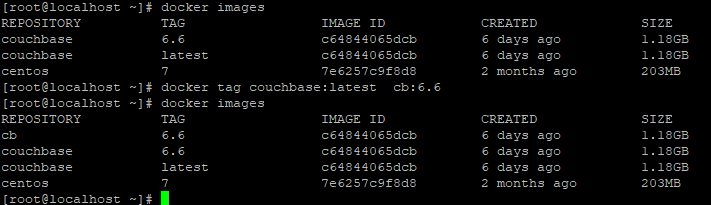
镜像名一定要写全，不写Tag的话，默认就是删除latest最新版本的，当镜像名不好写时，可写镜像的id，有时在构建镜像时会生成一些中间镜像，都没有名字，名称显示为<none>

一个一个地删除不方便，可用以下命令删除：

# docker rmi $(docker images | grep "none" | awk '{print $3}')

# **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

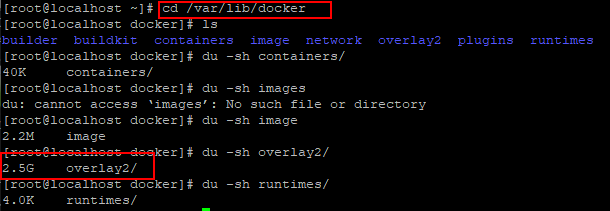


**2.镜像究竟是什么**？

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



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





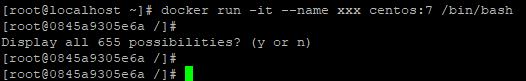
然后把该centos7.tar文件复制到其他装有docker服务的宿主机上

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



# docker images可见导入的镜像，可以导入后再修改镜像名称

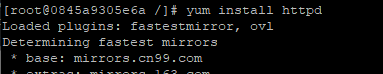
# **docker run -it --name xxx 镜像名 参数** //运行一个镜像，命名为xxx

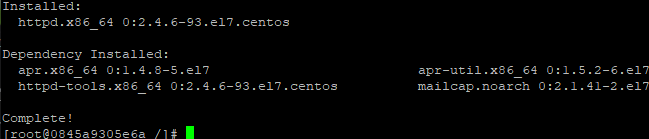


本例中运行了centos:7这个镜像，运行该镜像后，镜像的运行体就是一个实例或者叫容器。

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







输入exit后就退出了容器实例，回到了宿主系统里，

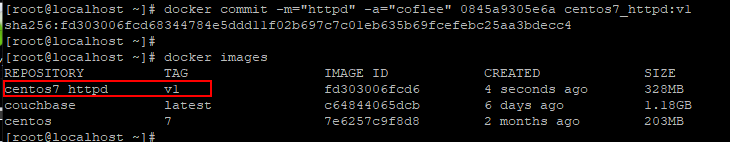
# **docker ps -a**  //查看刚刚运行过的容器，名为xxx的



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**



**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

--restart=always 当服务器重启后，让容器也跟随启动

# **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

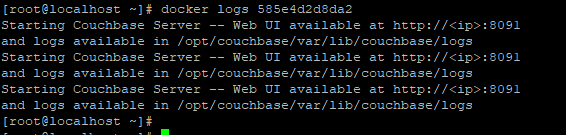
# **docker rm -f 容器id** //删除一个已经停止运行的容器

# **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的，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



# **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



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**

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了的端口相重复。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=ture

-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 -t 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 && \

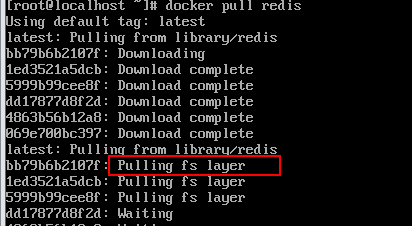
yum remove -y xx1 && \

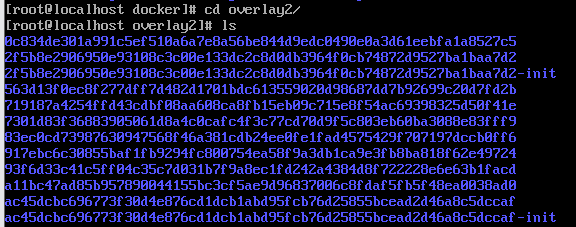
yum clean all

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





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/mysap-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/mysap-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，默认的仓库在https://docker.io

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：

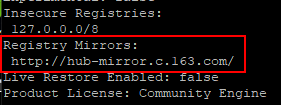
https://registry.docker-cn.com

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

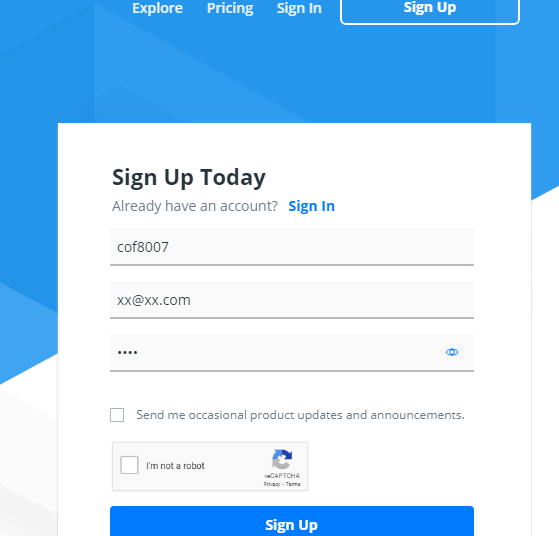


可见配置生效

**在宿主系统上登录到docker远程仓库**：

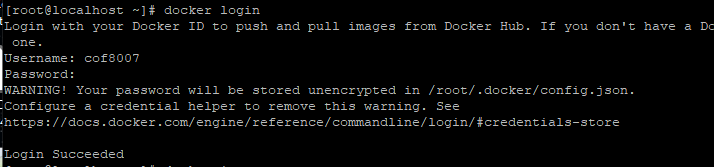
首先要在默认的仓库上创建一个账号，比如默认在docker.io时，我们先在https://hub.docker.com上创建自己的账号

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

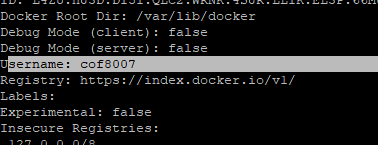
# docker login //默认是登录到docs.docker.com仓库，与我们配置的registry-mirrors指定的地址无关，那个只是默认下载镜像的地址，而不是登录公共仓库的



输入用户名和密码就顺利登录了

然后查看一下，

# docker info



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docs.docker.com的cof8007账户里

# docker logout //退出登录



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.给docker设置代理**

在国内访问默认的docker.com或docker.io时 速度比较慢，可以使用国内其他的registry，但有时国内的registry可能不含某些镜像，所以我们还是得想其他办法，比如给docker设置一个代理，就是让docker把请求的数据发给代理服务器，这个代理就是指http(s)的正向代理，关于 正向代理的原理请见作者其他文档。

前提是在本地内网搭建一个代理服务器，如http的正向代理服务器，ip和端口号为：

192.168.0.105:10809

# mkdir /etc/systemd/system/docker.service.d //创建docker.service.d子目录



# vi /etc/systemd/system/docker.service.d/http-proxy.conf //创建文件并编辑



内容如下：

[Service]

Environment="HTTP\_PROXY=http://192.168.0.105:10809/" "NO\_PROXY=localhost,127.0.0.1,192.168.0.0/16" //这与上面是一行



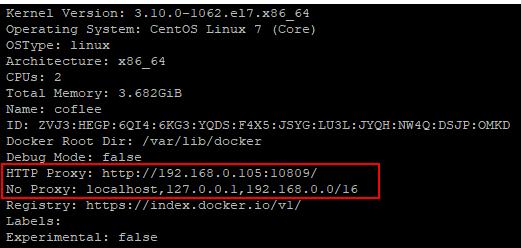
保存，

# systemctl daemon-reload

# systemctl restart docker



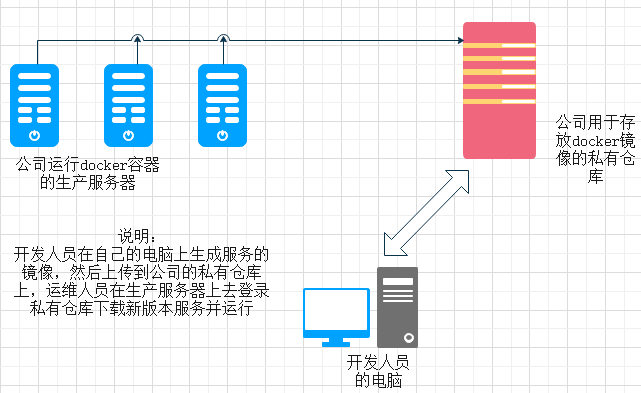
# docker info //查看信息，如下图可见已经生效了



**11.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：



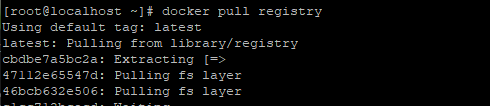
当然，这只是一个简单的流程（实际中不建议直接用registry，而是建议用harbor）

**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



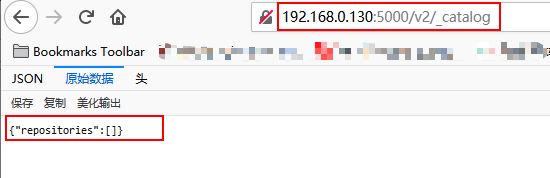
这个registry服务软件也是docker公司开发的，并且是开源的，它运行后，我们就可以上传镜像到它那里，它会把我们上传的镜像保存到容器里的/var/lib/registry目录下，所以我们得在目标服务器上做个目录映射，让数据存在宿主系统的磁盘里，而且此服务默认端口号为5000，所以还要做个端口映射

# **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



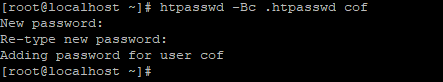


然后可以使用Httpd-tools的htpasswd命令创建用户

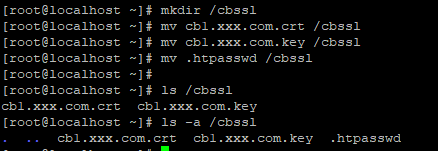


# yum install httpd-tools

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



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



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记了**

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

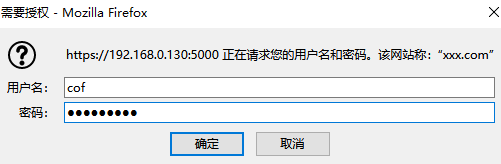
（这个 **`** 反撇号英文名叫grave 或backtick）



运行成功后，在其他电脑上访问https://x.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？

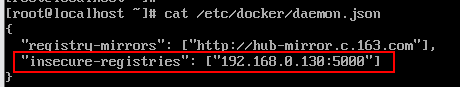
首先也要让客户端的docker信任registry服务器的ssl证书，

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



添加一条配置：

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



保存，

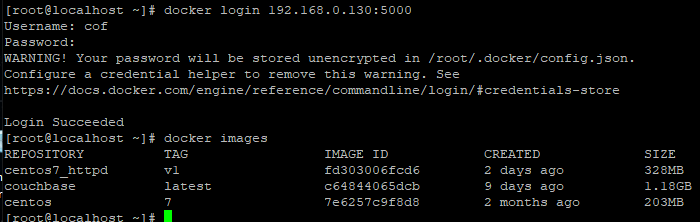
重启docker服务（前提是确保当前运行的容器都已正确停止）

# 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**

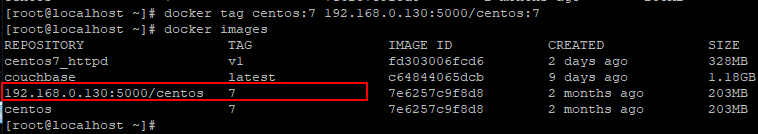


输入用户名和密码后就成功登录了，再查看当前本地的镜像

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

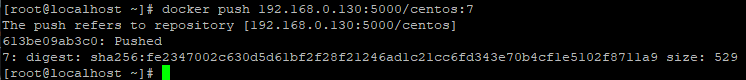
# **docker tage centos:7 192.168.0.130:5000/centos:7**



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



最后登录https://x.x.x.x:5000/v2/\_catalog 查看一下

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



**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）它是一套服务系统。

**12.搭建本地Harbor镜像仓库**

上一章讲了registry仓库的搭建，了解了基本的原理，现在讲一下Harbor的搭建

安装harbor前的准备事项：

1.首先要安装有docker-compose工具，这个工具默认不和docker-ce一起安装，得自己去下载，地址为：<https://github.com/docker/compose/releases/>

选择较新版本，

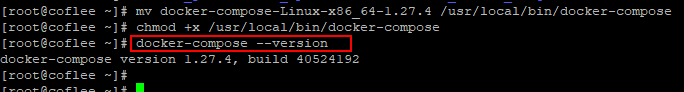


如果是要装在centos7上，则下载docker-compose-Linux-x86\_64这个文件，它就是可执行的二进制文件，下载到电脑上，然后复制到centos7上的/usr/local/bin/目录下，重命名为docker-compose

# mv docker-compose-Linux-x86\_64 /usr/local/bin/docker-compose

# chmod +x /usr/local/bin/docker-compose //添加可执行权限

# docker-compose --version //测试命令是否能用



2.确定harbor主机的名称，并创建harbor的ssl证书，可使用自签名证书，本例中harbor主机名称为harbor.test.com



证书所在目录为/etc/harbor\_cert



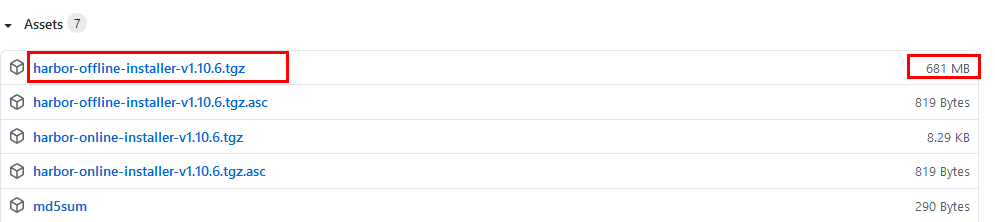


3.确定harbor仓库数据目录，目录可以单独挂载一个磁盘，空间要足够，比如在/data目录下

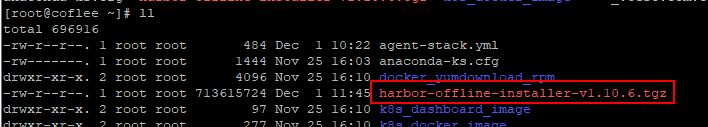


4.然后可以安装harbor了，建议使用offline离线安装版去安装，下载地址：

<https://github.com/goharbor/harbor/releases> 下载较新版本



下载680MB左右大小的那个，本例使用v1.10.6的版本，下载后上传到centos7上



# tar -xf harbor-offline-installer-v1.10.6.tgz //解压缩

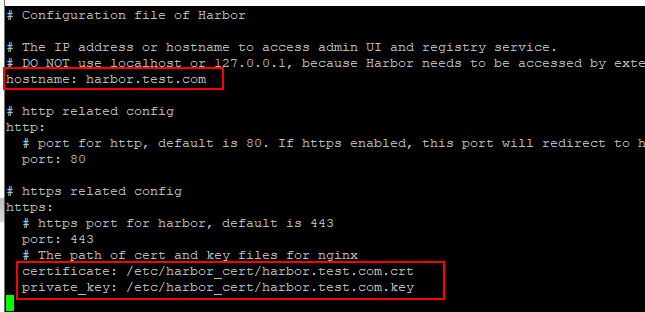
解压后，生成一个目录 名为harbor



目录里的harbor.v1.10.6.tar.gz为harbor要用到的docker镜像，都打包好了，在执行./install.sh脚本时会自动去解压并load进docker里，在执行install.sh前要先设置一下，

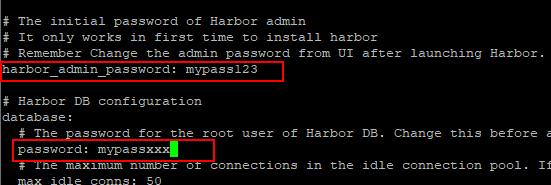
# cd harbor //进入解压目录

# vi harbor.yml //修改配置文件



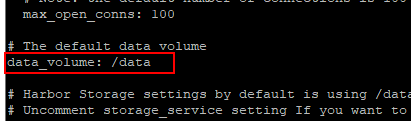
首先改hostname为事先规划的 harbor.test.com

https:下面，指定使用的ssl证书及key



接下来指定harbor admin用户的密码，如mypass123

database下面的password:为harbor要用到的数据库的密码，最好也是要改一下

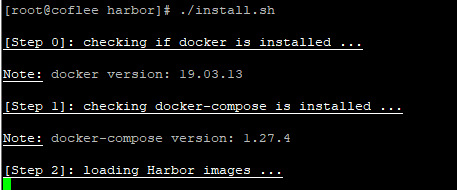


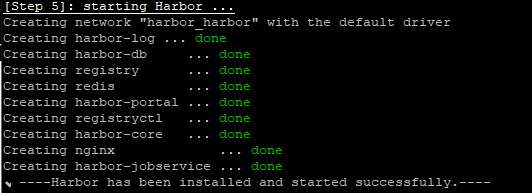
最后修改data\_volume:为事先规划的目录，如/data

保存

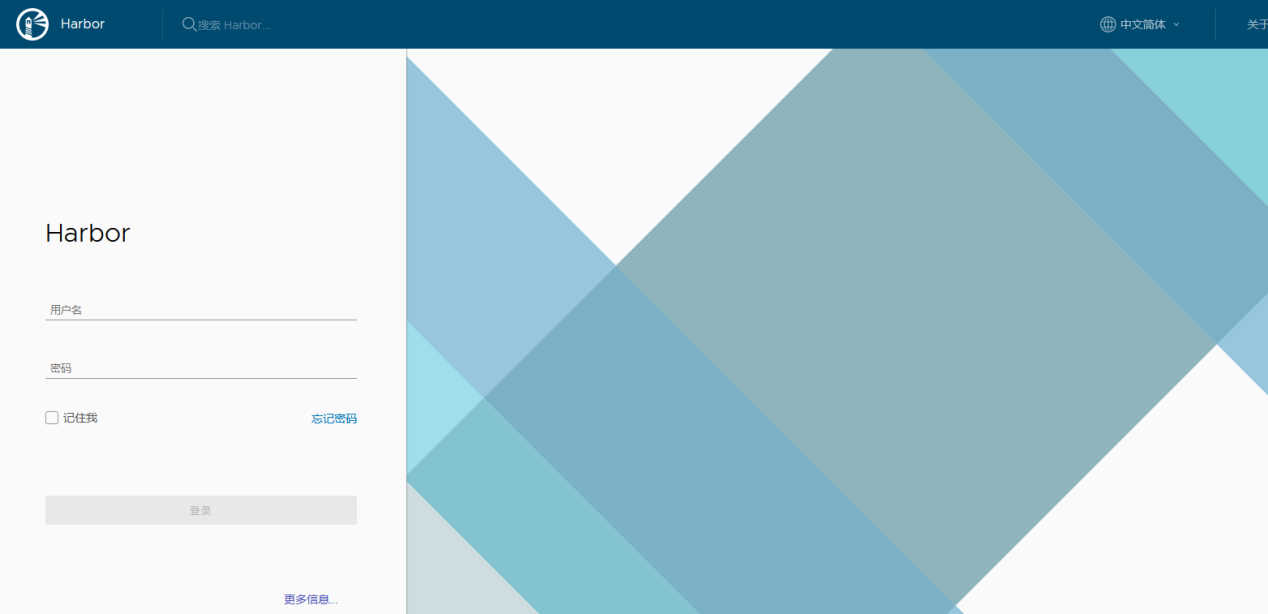
# ./install.sh //在harbor解压目录里执行此脚本，进行安装

安装的过程就是检测docker是否安装，docker-compose命令是否安装，再load相关镜像，再根据harbor.yml配置去生成docker-compose.yml文件，最后用docker-compose up命令去启动所有相关容器





当出现----harbor has been installed and started successfully.时，说明安装成功并已启动了



在浏览器输入https://harbor主机的ip 就可以访问了，默认就是443端口号

输入用户名admin和刚刚在配置文件设置的密码，就可登录了



harbor具体该怎么用就不多说了，和前面一章的registry差不多，

只是在其他docker客户端进行docker login和push时要用配置文件里指定的名称，而不能用ip，否则登录失败，所以要在客户端的docker主机上添加harbor.test.com的解析。

# vi /etc/hosts

x.x.x.x harbor.test.com

还要在docker配置文件里添加以下配置

# vi /etc/docker/daemon.json

{

"insecure-registries": ["harbor.test.com" ]

}

用法示例：

# docker login harbor.test.com //登录，先在harbor上创建用户

# docker tag xxxxx harbor.test.com/myPro/xxxx:v1 //打标签

# docker push harbor.test.com/myPro/xxxx:v1 //上传镜像到harbor上

# docker logout harbor.test.com //退出登录

**harbor的重置**：



在初次成功安装harbor时，会在harbor解压目录下生成一个docker-compose.yml配置文件，要停止harbor相关的容器时，就用docker-compose命令去操作

# docker-compose stop //在harbor解压目录下执行，停止与harbor相关

//的所有容器

# docker-compose start //启动与harbor相关的所有容器

当容器都stop时，先删除已停止的容器

# docker container prune

# vi harbor.yml //重新配置

# ./install.sh //重新安装，