### Docker使用手册

**前言：**

1.蓝色字体表示命令行命令，正式执行时不要复制前面的#号，#号只是提示应该使用root权限操作

2.绿色字体表示注释，有时注释太多就不用绿色表示了

3.注意：本文档的所有操作请先在测环境进行实践，请不要直接在真实的服务器中操作！

**版权声明**：

本文档以开源的形式发布，所有条款如下：

1. 无担保：作者不保证文档内容的准确无误，亦不承担由于使用此文档所导致的任何后果

2. 自由使用：任何人可以出于任何目的而自由地 阅读/链接/打印/转载/引用/再创作 此文档，无需任何附加条件

若您 阅读/链接/打印/转载/引用/再创作 本文档，则说明接受以上2个条款。

作者：李茂福

时间：2023-10-12

**第0章、安装docker**

**★centos7安装docker**

centos7自带的yum源没有docker软件，要指定其他的源，比如阿里的源或docker官方的

# cd /etc/yum.repos.d/

# wget https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo #阿里的源

然后在/etc/yum.repos.d/目录下多了一个名为docker-ce.repo的文件

如果下载不了，可以在此目录下自己创建一个docker-ce.repo文件，内容如下：

[docker-ce-stable]

name=Docker CE Stable - $basearch

baseurl=https://mirrors.aliyun.com/docker-ce/linux/centos/7/$basearch/stable

enabled=1

gpgcheck=1

gpgkey=https://mirrors.aliyun.com/docker-ce/linux/centos/gpg

**官方的源：**

# cd /etc/yum.repos.d/

# wget https://download.docker.com/linux/centos/docker-ce.repo #官方的源

# cat docker-ce.repo

[docker-ce-stable]

name=Docker CE Stable - $basearch

baseurl=https://download.docker.com/linux/centos/$releasever/$basearch/stable

enabled=1

gpgcheck=1

gpgkey=https://download.docker.com/linux/centos/gpg

**#安装docker-ce**

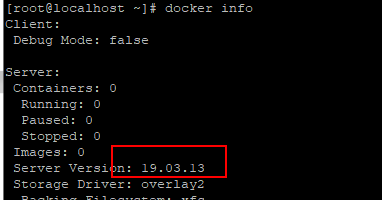
# yum install docker-ce docker-ce-cli containerd.io -y



# systemctl enable docker

# systemctl start docker

# docker info



**★ubuntu2204安装docker**

先下载docker官方gpg公钥

$ sudo mkdir /etc/apt/gpgkey

$ sudo wget https://download.docker.com/linux/ubuntu/gpg -O /etc/apt/gpgkey/docker-gpg

$ sudo gpg --dearmor -o /etc/apt/gpgkey/docker.gpg /etc/apt/gpgkey/docker-gpg

$ sudo chmod a+r /etc/apt/gpgkey/docker.gpg

$ sudo apt-key add docker-gpg #旧版本添加gpg key到系统

$ sudo vi /etc/apt/sources.list #添加docker官方的源

deb [arch=amd64 signed-by=/etc/apt/gpgkey/docker-gpg] https://download.docker.com/linux/ubuntu jammy stable

$ sudo apt update #更新apt的资源列表

$ sudo apt-get install docker-ce docker-ce-cli containerd.io #安装docker-ce

# sudo systemctl enable docker

# sudo systemctl start docker

**第1章、docker初始设置**

**★检查docker0的ip网段与环境现有ip网段是否有冲突，有则需要改**

**参考第8章、Docker网络模式**

**★操作docker的命令**：

要使用管理员或root的权限去操作docker命令！

**★指定存储images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，

**默认存储位置**：

Windows: C:\ProgramData\Docker

Linux: /var/lib/docker

**docker的配置文件**：

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": "/docker\_data"

}

#windows上的docker配置文件里路径写成 "data-root": "D:\\docker\_data"

**★log日志设置**

# vi /etc/docker/daemon.json

{

"log-level": "warn",

"log-driver": "json-file",

"log-opts": {

"max-size": "50m",

"max-file": "1"

}

}

**默认日志文件：**

# docker inspect 容器名 | grep "LogPath" #查看日志文件名

/docker\_data/containers/ID/ID-json.log #这个id为容器inspect查看到的长id

**★docker使用systemd服务管理**

# vi /etc/docker/daemon.json

{

"exec-opts": ["native.cgroupdriver=systemd"]

}

# mkdir -p /etc/systemd/system/docker.service.d

# systemctl daemon-reload

# systemctl enable docker

# systemctl restart docker

**★存储驱动overlay2及devicemapper**

devicemapper是红帽系列linux发行版上的默认存储驱动，它有两种配置模式：loop-lvm以及direct-lvm

direct-lvm是Docker官方推荐的生产环境中可以使用的模式，该模式使用块设备来构建精简池来存放镜像和Docker容器的数据。

Docker host使用loop-lvm的配置模式来运行devicemapper。该模式使用稀疏文件来创建thin池，这些池用于镜像和容器快照。并且这些模式是开箱即用的，无需额外配置。不过，不建议在产品部署中使用loop-lvm模式。

产品部署中应该使用direct-lvm模式，该模式使用块设备来创建thin池。

**★docker默认存储驱动为overlay2**

OverlayFS is the recommended storage driver, and supported if you meet the following prerequisites:

Version 4.0 or higher of the Linux kernel, or RHEL or CentOS using version 3.10.0-514 of the kernel or higher.

The overlay2 driver is supported on xfs backing filesystems, but only with d\_type=true enabled.

Use xfs\_info to verify that the ftype option is set to 1. To format an xfs filesystem correctly, use the flag -n ftype=1.

# xfs\_info /dev/mapper/rl-root | grep ftype

naming =version 2 bsize=4096 ascii-ci=0, ftype=1

# cat > /etc/docker/daemon.json <<EOF

{

"experimental": true,

"exec-opts": ["native.cgroupdriver=systemd"],

"insecure-registries": [ "cof-lee.com", "docker.io", "quay.io", "registry.redhat.io" ],

"data-root": "/data",

"storage-driver": "overlay2",

"storage-opts": [

"overlay2.size=50G"

]

}

EOF

#如果要对overlay2的容器空间限制大小，需要使用xfs文件系统，并且挂载时使用pquota

# xfs\_quota -x -c 'report' /data #查看docker的"data-root"目录的限额报告

# vi /etc/fstab

/dev/xxx /data xfs defaults,uquota,gquota,pquota,prjquota 0 0

# xfs\_quota -x -c 'report' /data

**2种存储驱动默认存储路径：**

/var/lib/docker/overlay2

/var/lib/docker/devicemapper

**★也可以修改存储驱动为Devicemapper模式**

官方指导链接： https://docs.docker.com/storage/storagedriver/device-mapper-driver/

**★direct-lvm自动配置**

# cat > /etc/docker/daemon.json <<EOF

{

"experimental": true,

"exec-opts": ["native.cgroupdriver=systemd"],

"insecure-registries": [ "cof-lee.com", "docker.io", "quay.io", "registry.redhat.io" ],

"storage-driver": "devicemapper",

"storage-opts": [

"dm.directlvm\_device=/dev/sdb",

"dm.thinp\_percent=95",

"dm.thinp\_metapercent=1",

"dm.thinp\_autoextend\_threshold=80",

"dm.thinp\_autoextend\_percent=20",

"dm.directlvm\_device\_force=false",

"dm.basesize=50G"

]

}

EOF

directlvm\_device 将要配置的direct-lvm的块设备的路径

thinp\_percent 指定需要创建的data thin pool 的大小

thinp\_metapercent 指定需要创建的metadata thin pool的大小

thinp\_autoextend\_threshold 指定自动扩容的百分比，其中100代表disable，百分比最小为50

thinp\_autoextend\_percent 指定每次扩容的大小，即实际值，上面的为百分比。需要注意的是也是100代表disable

directlvm\_device\_force 当需要使用的块设备中已经有文件系统时，是否需要对该设备执行格式化

systemctl daemon-reload

systemctl restart docker

自动创建名为docker的vg，名为thinpool的lv

**★direct-lvm手动配置**

direct-lvm自动配置目前还只能支持1个块设备，如果需要使用多个块设备，这个时候direct-lvm 自动配置就不适合了，需要进行手动配置

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

# vgcreate docker /dev/xxx #可以将多个pv加入此vg

# lvcreate --wipesignatures y -n thinpool -l 95%VG docker

# lvcreate --wipesignatures y -n thinpoolmeta -l 1%VG docker

#把池转换为thin池

# lvconvert -y --zero n -c 512K --thinpool docker/thinpool --poolmetadata docker/thinpoolmeta

#通过lvm文件来配置thin池的自动扩容

# cat > /etc/lvm/profile/docker-thinpool.profile <<EOF

activation {

thin\_pool\_autoextend\_threshold=80

thin\_pool\_autoextend\_percent=20

}

EOF

# lvchange --metadataprofile docker-thinpool docker/thinpool #应用新的lvm profile文件

# lvs -o+seg\_monitor #通过卷信息查看逻辑卷是否受监控；如果不受监控，则配置：

# lvchange --monitor y docker/thinpool

# cat > /etc/docker/daemon.json <<EOF

{

"experimental": true,

"exec-opts": ["native.cgroupdriver=systemd"],

"insecure-registries": [ "cof-lee.com", "docker.io", "quay.io", "registry.redhat.io" ],

"storage-driver": "devicemapper",

"storage-opts": [

"dm.thinpooldev=/dev/mapper/docker-thinpool",

"dm.use\_deferred\_removal=true",

"dm.use\_deferred\_deletion=true",

"dm.basesize=60G"

]

}

EOF

systemctl daemon-reload

systemctl restart docker

★ "dm.basesize=20G" 大小只可增加，不可缩容；只对新下载的镜像有效，已下载的镜像无效。

扩容时，直接扩容 /dev/mapper/docker-thinpool 即可

**第2章、镜像的使用**

#docker images #默认查看当前宿主机上的docker 镜像



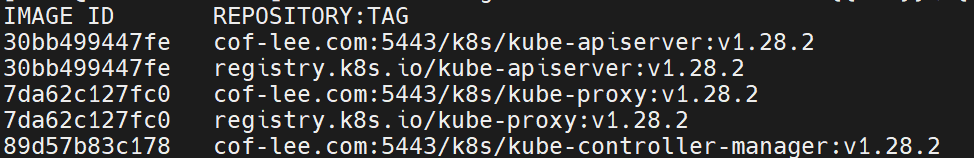
docker镜像全名由Repository和Tag组成（中间用冒号连接），如上图，显示有2个镜像：

couchbase:latest和registry:latest

第一列Repository表示仓库，即镜像主体名，第二列的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。

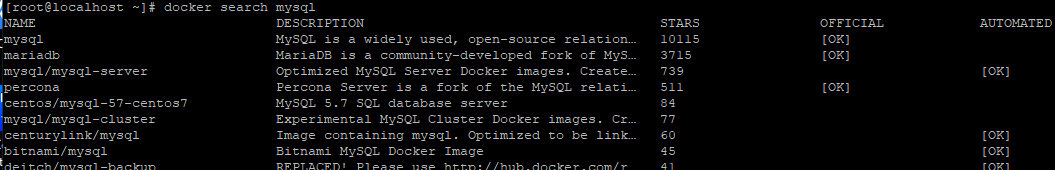
docker images查看到的镜像的属性有名称，还有镜像ID及大小，镜像ID是唯一的，由12个十六进制字符表示，镜像大小不固定，有大有小，大的可以有10来个GB，小的10来MB甚至几百KB

# docker image ls --format "table {{.ID}}\t{{.Repository}}:{{.Tag}}" #以指定格式显示



# docker inspect 镜像id #查看镜像详细信息

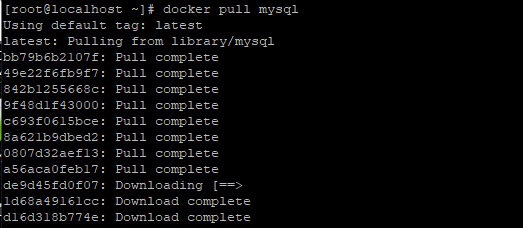
# docker search 软件名 #默认是连接到docker hub（docker.io）上去查找目标软件的docker镜像



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

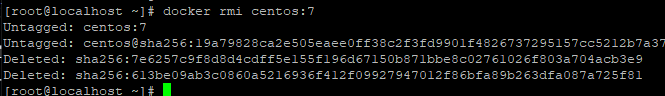
一般推荐下载官方的镜像

#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，可以先看后面第10章“镜像仓库管理”

# 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>

docker pull 及 docker build 可以导致这种现象。由于新旧镜像同名，旧镜像名称被取消，从而出现仓库名、标签均为 <none> 的镜像。

这类无标签镜像也被称为 虚悬镜像（dangling image）

# docker image ls -f dangling=true #显示虚悬镜像

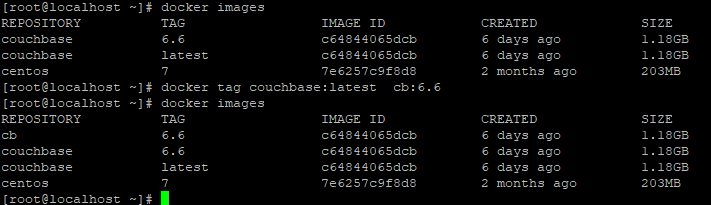
一个一个地删除不方便，可用以下命令删除：

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

# docker image prune #删除所有虚悬镜像

#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



**★下载/使用指定cpu架构的镜像**

docker下载或使用其他cpu架构/平台的镜像，需要使用 --platform 参数来指定cpu架构（amd64，arm64）

docker-19.03.9及以上版本才支持 --platform 参数，默认未开启 --platform 参数，需要手动开启，直接执行，会有下面的报错：

"--platform" is only supported on a Docker daemon with experimental features enabled

# docker info | grep -i "Experimental" #查看docker是否支持experimental

# vi /etc/docker/demon.json #增加下面的参数

"experimental": true

# systemctl daemon-reload

# systemctl restart docker

# docker pull --platform=arm64 imageName #下载指定cpu架构的镜像

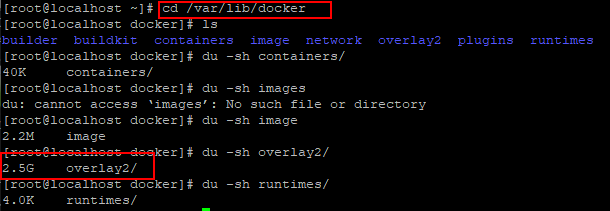
# docker inspect imageName | grep "Architecture"

**第3章、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程序软件，比如mysql镜像，就是由linux基础软件加上mysql软件组成的，镜像在使用时，一般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）或者在配置文件里指定的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

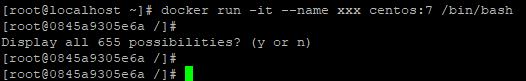
# docker load -i centos7.tar #从tar包导入镜像，导入时不可指定镜像名



# docker images #可查看导入的镜像，可以导入后再修改镜像名称

# docker save imgName1 imgName2 imgNamexx -o xxx.tar #导出多个镜像到一个tar文件中

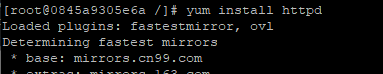
# docker run -it --name xxx 镜像名 参数 #运行一个镜像，命名为xxx



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，表示容器的shell提示符，这个0845a9305e6a就是容器实例的id

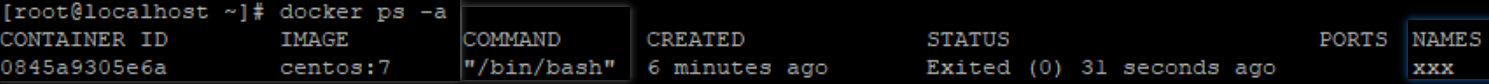
root@0845a9305e6a# yum install httpd #在容器实例里安装httpd软件





容器里输入exit 回车后就退出了容器实例，回到了宿主系统里

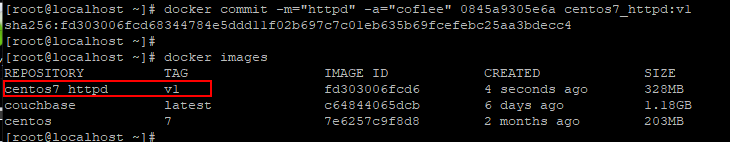
# docker ps -a  **#**查看刚刚运行过的容器，名为xxx的



在容器实例里进行的任何操作都不会保存到原始镜像里，原始镜像是静态的数据，不会改变的。容器实例是把镜像的数据做个快照，在容器里的所有操作都会保存到另外的地方（快照里），所以当我们删除容器时，这个容器里的所有数据就不存在了（快照删除了）要怎么做才能把容器变成静态的数据永久保存下来呢？

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



**★rootfs根文件系统**

根文件系统是一个从根/开始的目录结构，可以使用chroot命令进入指定的根文件系统

**★ramdisk**

RAMDisk将制作好的rootfs压缩后写入 Flash，启动的时候由Bootloader load到RAM，解压缩，然后挂载到/根目录。这种方法操作简单，但是在RAM中的文件系统不是压缩的，因此需要占用许多嵌入式系统中稀有资源RAM。

ramdisk就是用内存空间来模拟出硬盘分区，ramdisk通常使用磁盘文件系统的压缩存放在flash中，在系统初始化时，解压缩到SDRAM并挂载根文件系统，在linux系统中，ramdisk有二种，一种就是可以格式化并加载，在linux内核2.0/2.2就已经支持，其不足之处是大小固定;另一种是2.4的内核才支持，通过，ramfs来实现，他不能被格式化，但用起来方便，其大小随所需要的空间增加或减少，是目前linux常用的ramdisk技术。

**★initrd**

initrd 是 RAMDisk 的格式，kernel 2.4 之前都是 image-initrd，Kernel 2.5 引入了 cpio-initrd，大大简化了 Linux 的启动过程

RAMDisk的制作方法如下：

# dd if=/dev/zero of=init.img bs=4096 count=2048

#其中bs乘以count为块设备大小（单位：字节），生成init.img文件以后，还必须对该文件进行格式化。

# mke2fs –m0 –F init.img

# mkdir ram

# mount init.img ram/ -o loop

#这时，读写ram目录，等效于读写init.img文件。用户可以将根文系统所需的文件写入到ram目录中。往ram目录写完文件以后，还需要使用umount ram命令卸载init.img，才能将已写入的文件保存到init.img中。

# umount ram #添加完毕，需要卸载ram目录

**★dracut生成initramfs**

-H, --hostonly Host-Only mode: Install only what is needed for booting the local host instead of a generic host.

--keep Keep the temporary initramfs for debugging purposes

# dracut -H --keep initramfs-xxx.img

Not removing /var/tmp/dracut.bJEi2M.

# cd /var/tmp/dracut.bJEi2M/initramfs

# tar -czf initramfs.tar.gz \*

# docker import initramfs.tar.gz myinit:v1

# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

myinit v1 a2953ac6007b 4 seconds ago 66.1MB

# docker run -it myinit:v1 sh

sh-4.4#

**★scratch空镜像**

scratch是一个空镜像，只能用于构建镜像，通过FROM scratch，再构建一些基础镜像，比如busybox

scratch不可pull，也不可push，只可在构建镜像时FROM scratch

ldd命令是Linux中用于列出动态库依赖关系的工具。可以通过命令"ldd -v"或"--verbose"来查看详细的依赖关系信息

用 ldd 查出所需的 .so 文件将所有依赖压缩成 rootfs.tar 或 rootfs.tar.gz，再build到 scratch 基础镜像

**例如：**

# ldd /usr/bin/bash

linux-vdso.so.1 (0x00007fffb14f0000)

libtinfo.so.6 => /lib64/libtinfo.so.6 (0x00007f8fb6c67000) -> /lib64/libtinfo.so.6.1

libdl.so.2 => /lib64/libdl.so.2 (0x00007f8fb6a63000) -> /lib64/libdl-2.28.so

libc.so.6 => /lib64/libc.so.6 (0x00007f8fb669e000) -> /lib64/libc-2.28.so

/lib64/ld-linux-x86-64.so.2 (0x00007f8fb71b2000) -> /lib64/ld-2.28.so

# tar -zcvf rootfs.tar.gz /lib64/libtinfo.so.6 /lib64/libtinfo.so.6.1 /lib64/libdl.so.2 /lib64/libdl-2.28.so \

/lib64/libc.so.6 /lib64/libc-2.28.so /lib64/ld-linux-x86-64.so.2 /lib64/ld-2.28.so /usr/bin/bash

# cat > Dockerfile <<EOF

FROM scratch

ADD rootfs.tar.gz / #ADD作用同COPY，只是ADD会自动解压tar.gz文件，而copy不会自动解压

CMD ["/usr/bin/bash"]

EOF

# docker build --rm -t myapp:v7 .

# docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

myapp v7 985aa3ee1c29 1 minutes ago 4.51MB

# docker run -it myapp:v7 #运行镜像，默认进入bash

bash-4.4#

**#或者不使用docker build进行构建，用docker import导入一个文件系统为镜像**

# docker import rootfs.tar.gz myapp:v8

# docker run -it myapp:v8 bash

bash-4.4#

**第4章、容器的使用**

**①启动容器**

# 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shell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

# 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 #启动一个已经停止的容器

# docker restart 容器id #重启一个容器

**②传入环境变量**

# docker run 镜像名 -e TZ=Asia/Shanghai # -e 设置环境变量

**③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 #强制删除容器，包括正在运行的及已停止的

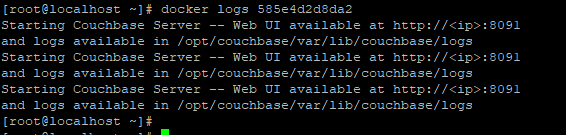
# docker container prune #删除所有已停止运行的容器，停止的容器也是要占用系统的存储空间的，确定不用了的容器就可删除

**④容器信息查看**

# docker inspect 容器id/容器名 #查看容器的详细信息

#容器的信息储存在docker数据目录下的containers子目录下

# 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

**⑤容器快照导入导出**

# docker export 容器id > xxx.tar #导出容器快照到xxx.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

**第5章、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到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

**第6章、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

-w /dir/xx # -w指定容器的工作目录，即cd到此目录下再运行某命令

# 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

**第7章、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=

# 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

# 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

--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200000微秒内至少有100000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=

# 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

# 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，就不可知了

**第8章、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:宿主port:容器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可以认为是这个网桥的管理ip。容器的网卡以这个网桥管理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。

**★docker容器根据网络模型分类**

|  |  |  |
| --- | --- | --- |
| Bridge container | Bridge | 桥接，借助虚拟网桥为容器建立网络连接 |
| Open container | Host | 容器直接共享使用宿主结点的网络名称空间 |
| Joined container | Container | 多个容器共享同一个网络名称空间 |
| Closed container | none | 没有对外通信的网络接口，仅loopback口 |

**★修改docker0的网段及指定容器dns**

Centos7下docker0默认网段： 172.17.0.0/16

WinServ下默认网段： 172.22.32.0/20

# vi /etc/docker/daemon.json

{

"bip": "10.1.2.1/24", //指定docker0网段ip，ip的最后一字节不能为0

"dns": [ //指定容器的dns

"114.114.114.114",

"8.8.8.8"

],

"dns-search": ["cof-lee.com"] //指定容器的搜索域

}

**★指定容器的网络模式**

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

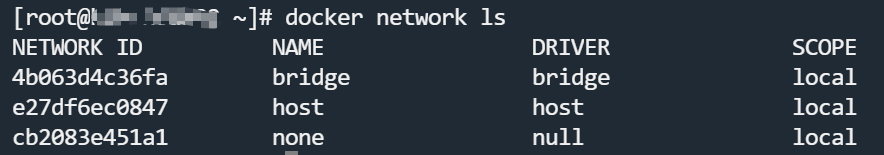
# docker run -d --network bridge 镜像名 #表示使用bridge模式，就是默认缺省的Nat模式，在windows上要写成--network nat

# docker run -d --network host 镜像名 #表示使用host模式，共用宿主系统的网络环境

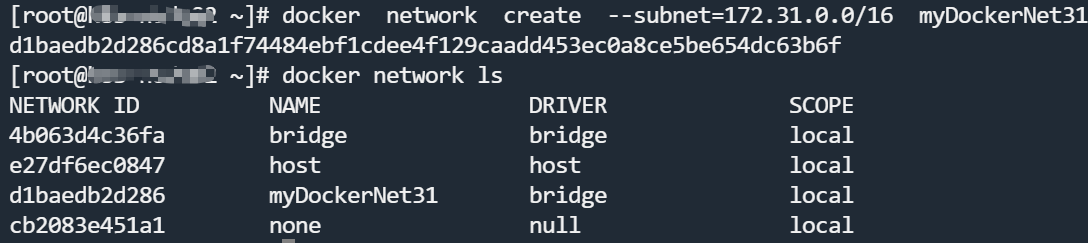
# docker run -d --network none 镜像名 #容器里只有loopback网卡，无其他网卡

**★创建容器网络（bridge模式）**

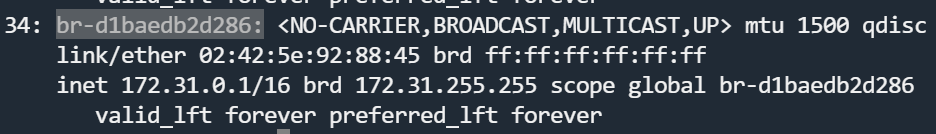
# docker network ls #查看容器网络



# docker network create --subnet=172.31.0.0/16 myDockerNet31 #默认为bridge类型的网络



#会生成一个以br-networkID为名的网桥



# docker run -d --network myDockerNet31 --ip 172.31.1.2 xxxx

# docker run -d -h hxx --dns=223.5.5.5 --dns-search=cof-lee.com imgName # -h指定容器的hostname

# docker network connect myDockerNet31 containerName #给容器增加一个网口

★默认类型的网络（bridge），不能指定parent接口，需要手动创建路由或nat规则

容器里的mac和（宿主机网卡及其他容器）的mac不一样，容器网络有网关ip

**★macvlan网络**

★macvlan-bridge模式（flat直通）

# docker network create -d macvlan \

--subnet=172.30.1.0/24 --gateway=172.30.1.1 \

-o parent=ens33 macvlan\_bridge\_net30

特点：容器里的mac和（宿主机网卡及其他容器）的mac不一样，容器网络有网关ip

★macvlan\_802.1q模式（有vlan标签）

# docker network create -d macvlan \

--subnet=172.20.1.0/24 --gateway=172.20.1.1 \

-o parent=ens33.20 macvlan\_vlan20

特点：容器里的mac和（宿主机网卡及其他容器）的mac不一样，容器网络有网关ip

**★ipvlan网络**

★ipvlan-l2模式

# docker network create -d ipvlan \

--subnet=172.23.1.0/24 --gateway=172.23.1.1 \

-o ipvlan\_mode=l2 -o parent=ens36 ipvlan\_l2\_net23

特点：容器里的mac和（宿主机网卡及其他容器）的mac一样，容器网络有网关ip

★ipvlan-l3模式

# docker network create -d ipvlan \

--subnet=172.24.1.0/24 \

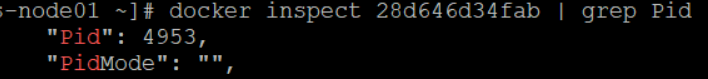
-o ipvlan\_mode=l3 -o parent=ens34 ipvlan\_l3\_net24

特点：容器里的mac和（宿主机网卡及其他容器）的mac一样，容器网络没有网关ip，下一跳指向容器的eth0接口

容器不能指定网关ip，外部网络要访问容器网络，需要创建路由，下一跳指向parent接口，即宿主机的ens34

**★操作容器网络netns**

# docker inspect 28d646d34fab | grep Pid #查看容器Pid



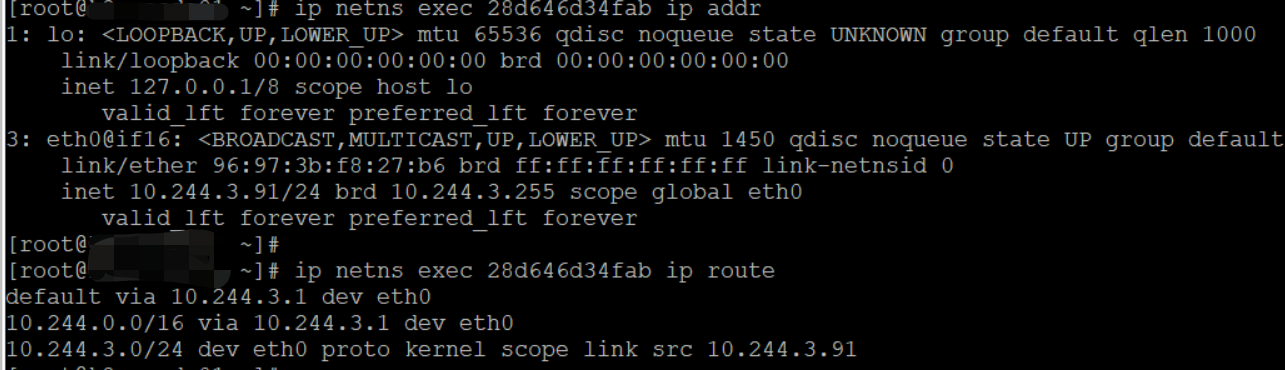
# mkdir /var/run/netns #创建网络命名空间目录

# ln -s /proc/4953/ns/net /var/run/netns/28d646d34fab #链接容器网络空间到宿主机空间

# ip netns list #查看网络空间



# ip netns exec 28d646d34fab ip addr #操作容器网络空间，查看及配置都可以



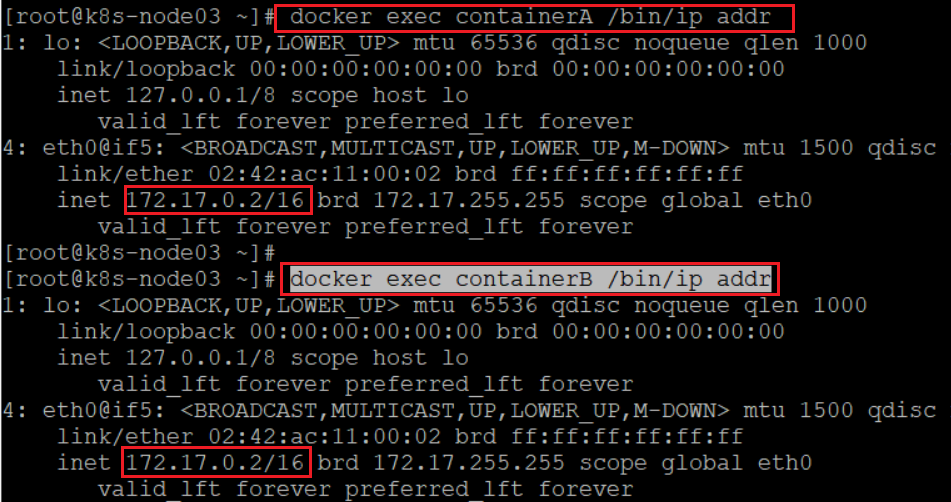
**★共享容器的网络栈及pid**

# docker run -d --name containerA 镜像名 参数

# mkdir /var/run/netns #创建网络命名空间目录

# ln -s /proc/4953/ns/net /var/run/netns/28d646d34fab #链接容器网络空间到宿主机空间

# docker run -d --name containerB --net=container:containerA 镜像名 参数



可见2个容器的ip地址是一样的，共享一个网络空间

**★共享容器的pid**

# docker run -d --name containerC --pid=container:containerA 镜像名 参数

#在2个容器里ps -ef查看到的信息是一样的

**★docker不生成默认的docker0网桥，不修改iptables规则**

# cat > /etc/docker/daemon.json <<EOF

{

"bridge": "none",

"iptables": false,

"experimental": true,

"exec-opts": ["native.cgroupdriver=systemd"],

"insecure-registries": [ "cof-lee.com", "docker.io", "quay.io", "registry.redhat.io" ]

}

EOF

# systemctl daemon-reload

# systemctl restart docker

**第9章、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 image:ver

WORKDIR /appdir

COPY ./xxdir /appdir

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 xxx #表示使用xxx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/dir

#这里的宿主目录为相对路径，是由构建时最后一个参数指定的

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路径，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 && \

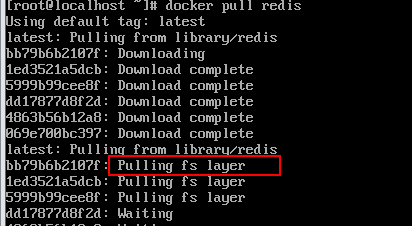
yum remove -y xx1 && \

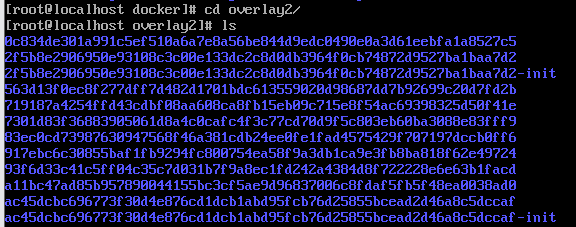
yum clean all

#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yum 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比较大（占用存储空间较大）

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。

**★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，项目所在目录

# 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。

**第10章、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（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，则下载速度会很慢，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

**★修改默认镜像仓库源**

# 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：

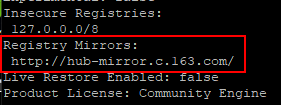
https://registry.docker-cn.com

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

# systemctl restart docker

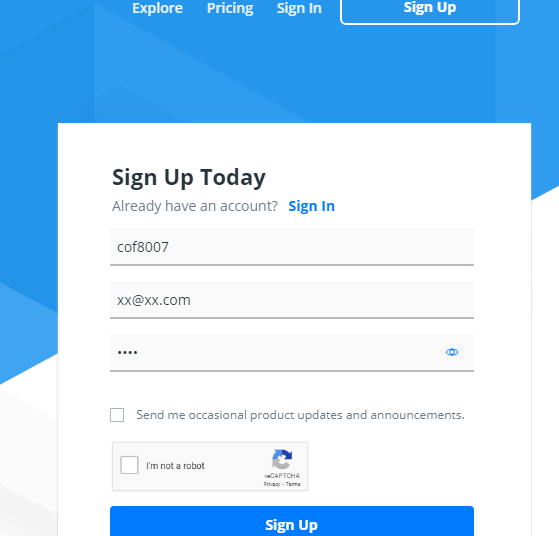
# docker info #可见配置生效



**★在宿主系统上登录到docker远程仓库**

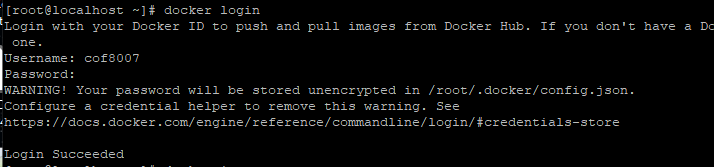
首先要在默认的仓库上创建一个账号，比如默认在docker.io时，我们先在https://hub.docker.com上创建自己的账号

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



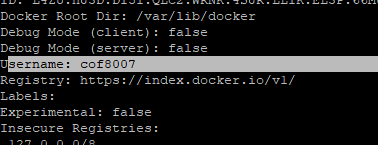
在宿主系统上登录账号

# 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

# 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



**第11章、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 #创建代理配置文件并编辑，内容如下：

[Service]

Environment="HTTP\_PROXY=http://192.168.0.105:10809/"

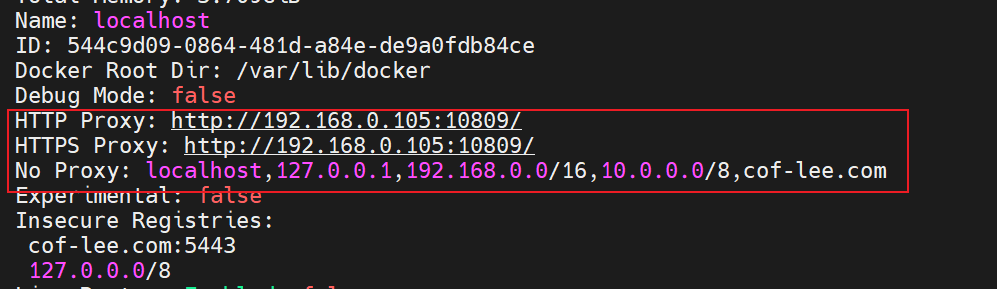
Environment="HTTPS\_PROXY=http://192.168.0.105:10809/"

Environment="NO\_PROXY=localhost,127.0.0.1,192.168.0.0/16,10.0.0.0/8,cof-lee.com"

# systemctl daemon-reload

# systemctl restart docker

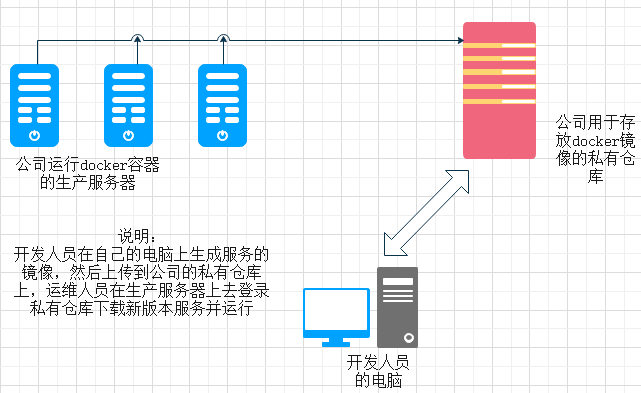
# docker info #查看信息，如下图可见已经生效了



**第12章、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：



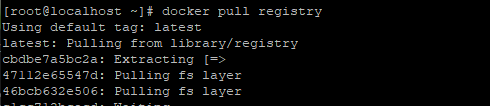
当然，这只是一个简单的流程（实际中不建议直接用registry，而是建议用harbor）

**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



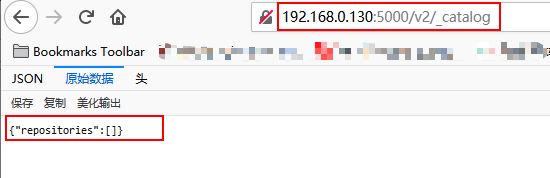
这个registry服务软件也是docker公司开发的，并且是开源的，它运行后，我们就可以上传镜像到它那里，它会把我们上传的镜像保存到容器里的/var/lib/registry目录下，所以我们得在目标服务器上做个目录映射，让数据存在宿主系统的磁盘里，而且此服务默认端口号为5000，所以还要做个端口映射

# 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。

★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



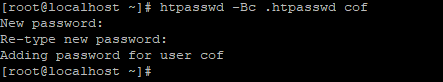


然后可以使用Httpd-tools的htpasswd命令创建用户

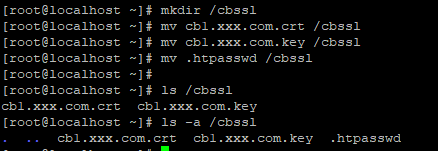


# yum install httpd-tools

# 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" \

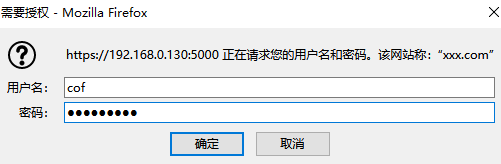
registry #**最后的镜像名不要忘记了**

#以上为一行，最后的**\**表示可以折行，在windows命令行里使用键盘左上角的**`**反撇号去折行（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





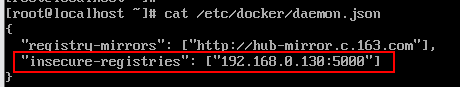
到此，本地的私有docker 镜像仓库服务器就配置好了，那么**客户端**（其他计算机上的docker服务）怎么用它呢？

首先也要让客户端的docker信任registry服务器的ssl证书，

**★客户端信任私有registry仓库**

#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

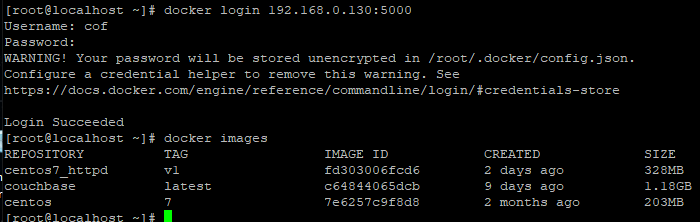


#保存，重启docker服务（前提是确保当前运行的容器都已正确停止）

# systemctl restart docker

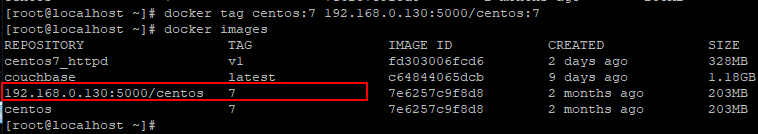
#之后就可以使用了，在客户端，登录我们的私有registry

# docker login 目标registry的ip:5000



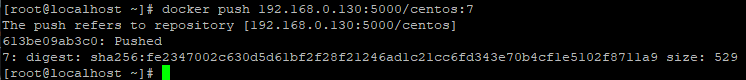
输入用户名和密码后就成功登录了，再查看当前本地的镜像，把要上传到registry的镜像重新打个tag，比如我们要上传centos:7到私有registry服务器，就把centos:7打上 x.x.x.x:5000/新镜像名 x.x.x.x指代registry服务器的Ip

# 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。

**第13章、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，这个工具默认和docker-ce一起安装了

docker自带有docker-compose，路径为/usr/libexec/docker/cli-plugins/docker-compose

# ln -s /usr/libexec/docker/cli-plugins/docker-compose /usr/bin/docker-compose

# docker-compose --version

#如果默认没有docker-compose，需要自己去下载，地址为：<https://github.com/docker/compose/releases/>

选择较新版本（若要支持多种cpu架构的镜像，需要使用v2.0及以上版本的harbor）



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/bin/目录下，重命名为docker-compose

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

# chmod +x /usr/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

# hostnamectl set-hostname harbor.test.com

#证书所在目录为/etc/harbor\_cert

# mkdir /etc/harbor\_cert

# ls /etc/harbor\_cert



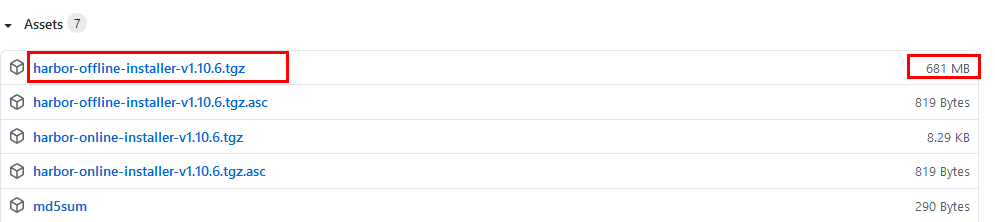
3.确定harbor仓库数据目录，目录可以单独挂载一个磁盘，空间要足够，比如在/data目录下

# mkdir /data

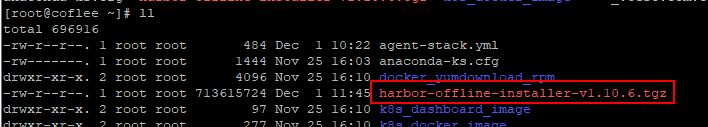
# mount /dev/xxxx /data

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



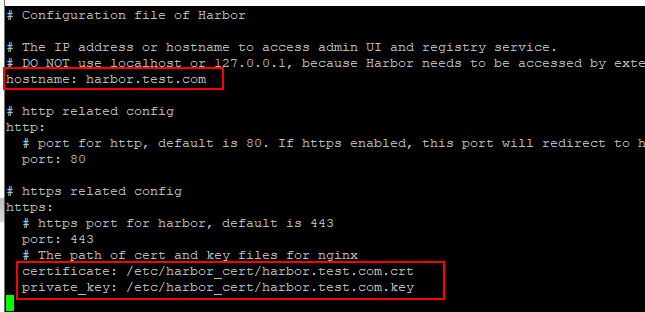
# tar -xf harbor-offline-installer-v1.10.6.tgz #解压缩离线安装包；解压后，生成一个名为harbor的目录



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前要先修改一下配置：

# cd harbor #进入解压目录

# vi harbor.yml #修改配置文件



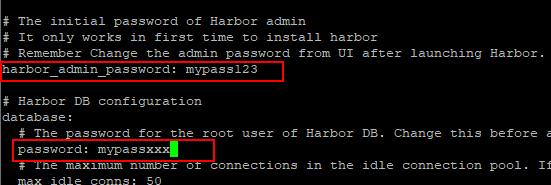
首先改hostname:为事先规划的 harbor.test.com

https:下面，指定使用的ssl证书及key

证书最终被复制到“data\_volume”指定的目录下的子目录secret/cert/里，并重命名为server.crt和server.key，比如：

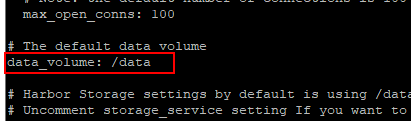
data\_volume: 值为/data

则证书最终存放在：/data/secret/cert/



接下来使用harbor\_admin\_password: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

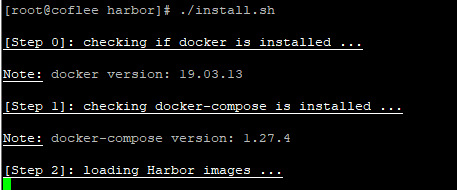


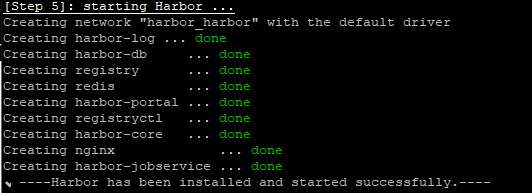
最后修改data\_volume:为事先规划的目录，如/data

#保存，退出

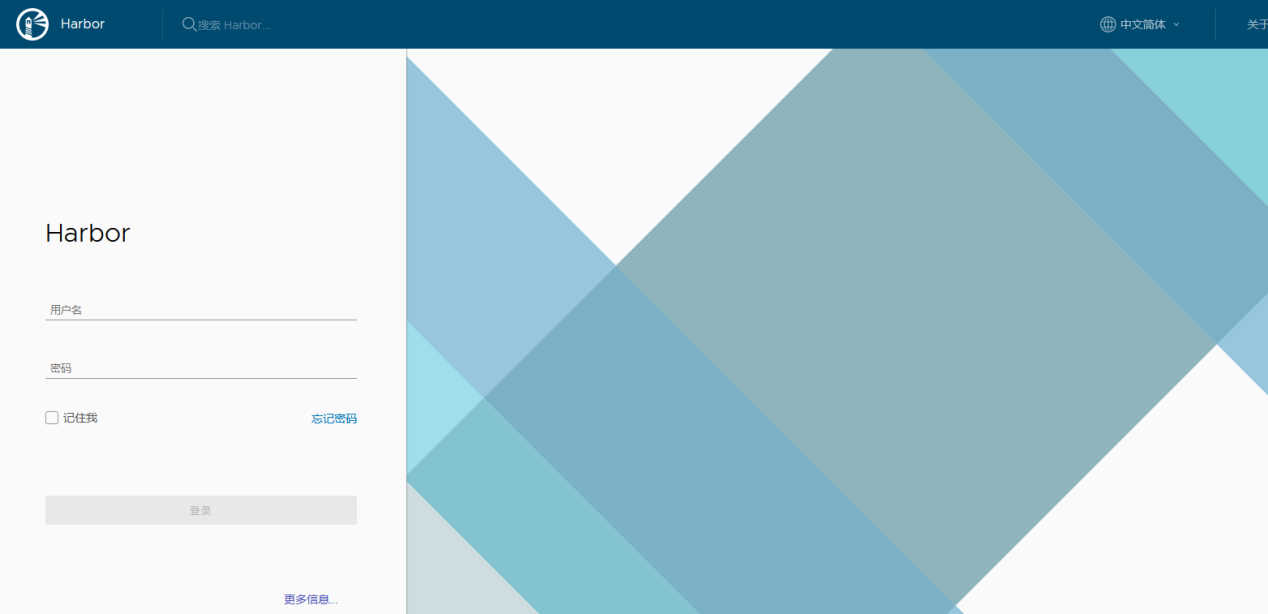
# ./install.sh #在harbor解压目录里执行此脚本，进行安装

安装的过程就是检测docker是否安装，docker-compose命令是否安装，再load导入相关镜像（镜像已在离线包里），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



#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

# vi /etc/docker/daemon.json #添加以下配置

{

"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

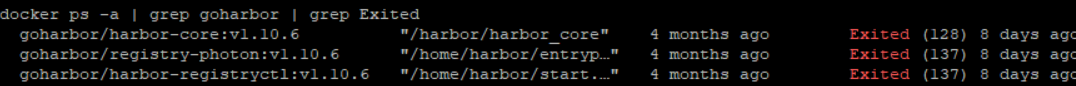
# docker container prune

# vi harbor.yml #重新配置

# ./install.sh #重新安装harbor

**★将harbor做成systemd服务**

**★当重启服务器后harbor并未自动重启**



#重启harbor

cd /root/docker\_portainer\_harbor/harbor

docker-compose -f docker-compose.yml stop #先停止

docker-compose -f docker-compose.yml up -d #重启

**#或者将harbor做成开机自启服务**

cat > /usr/lib/systemd/system/harbor.service <<EOF

[Unit]

Description=Harbor

After=docker.service network.target

Requires=docker.service

[Service]

Type=simple

Restart=on-failure

RestartSec=5

ExecStart=/usr/bin/docker-compose -f /root/docker\_portainer\_harbor/harbor/docker-compose.yml up

ExecStop=/usr/bin/docker-compose -f /root/docker\_portainer\_harbor/harbor/docker-compose.yml stop

[Install]

WantedBy=multi-user.target

EOF

systemctl daemon-reload

systemctl enable harbor

systemctl start harbor

**★制作支持多cpu架构的镜像**

docker manifest是用于管理Docker多平台镜像的工具，可以创建、查看和删除多架构镜像，以便在多种操作系统和CPU架构上运行Docker镜像。

**示例：**

#下载2个cpu架构的busybox镜像（x86\_64和aarch64）

# docker pull busybox --platform=amd64 #或者 --platform=linux/amd64 amd64也可写为 x86\_64

# docker tag busybox:latest cof-lee.com/tools/busybox-amd64:2023-07-18

# docker rmi busybox:latest

# docker pull busybox --platform=arm64 #或者 --platform=linux/arm64 arm64也可写为 aarch64

# docker tag busybox:latest cof-lee.com/tools/busybox-arm64:2023-07-18

# docker rmi busybox:latest

#将镜像名改为本地harbor仓库的hostname前缀

# docker tag busybox-amd64:2023-07-18 cof-lee.com/tools/busybox-amd64:2023-07-18

# docker tag busybox-arm64:2023-07-18 cof-lee.com/tools/busybox-arm64:2023-07-18

# docker manifest inspect imageName #查看镜像的manifest

#创建支持多cpu架构的镜像manifest（要求先登录docker并上传前面下载好了的2个busybox镜像）

# docker login -u admin -p xxxx cof-lee.com

# docker pull cof-lee.com/tools/busybox-amd64:2023-07-18

# docker pull cof-lee.com/tools/busybox-arm64:2023-07-18

# docker manifest create cof-lee.com/tools/busybox-multi-arch:2023-07-18 \

cof-lee.com/tools/busybox-arm64:2023-07-18 \

cof-lee.com/tools/busybox-amd64:2023-07-18 \

--insecure

提示：Created manifest list cof-lee.com/tools/busybox-multi-arch:2023-07-18 表示创建成功

在 ~/.docker/manifests/目录下面的 cof-lee.com\_tools\_busybox-multi-arch-2023-07-18

# docker manifest inspect cof-lee.com/tools/busybox-multi-arch:2023-07-18 #查看manifest

# docker manifest push cof-lee.com/tools/busybox-multi-arch:2023-07-18 --insecure #上传manifest到harbor

#上传后，在harbor上有此镜像，在本地没有，可以pull下载，pull下载时只可下载其中的一个arch版本

若报错：failed to put manifest \* unknown: Manifest.list is not supported.

原因是harbor1.x版本不支持多系统架构manifest，harbor2.x版本支持多系统架构manifest

**第14章、portainer UI**

portainer是可视化的docker容器管理工具，其本身也是以docker容器的方式运行。官方镜像在 docker.io/portainer/protainer-ce 可下载到本地并上传到自己环境的镜像仓库里

**★以http运行**

# 提前创建 /etc/portainer-data 目录

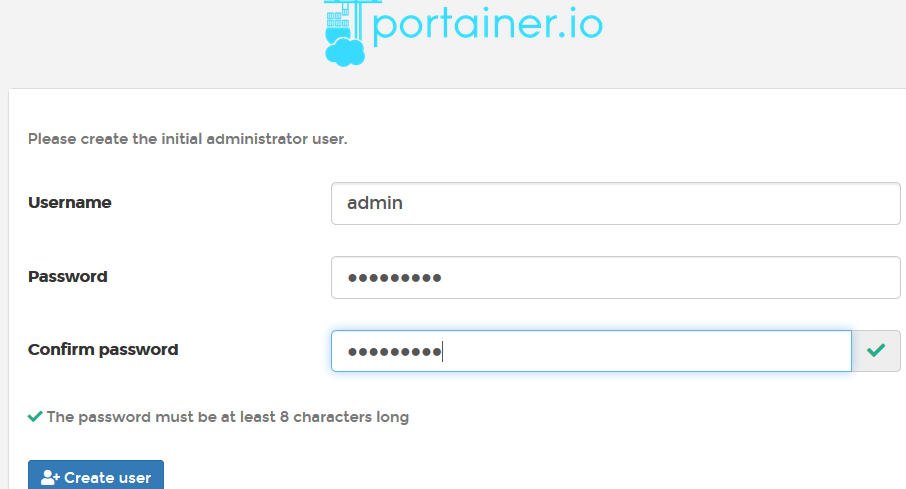
|  |  |
| --- | --- |
| docker run -d -p 9000:9000 --restart=always \ |  |
| -v /var/run/docker.sock:/var/run/docker.sock \ |  |
| -v /etc/portainer-data:/data \ | #数据存放目录 |
| --name portainer-ui-node1 \ | #容器名称 |
| cof-lee.com:5443/docker/portainer-ce:2.0.0 | #镜像名称 |

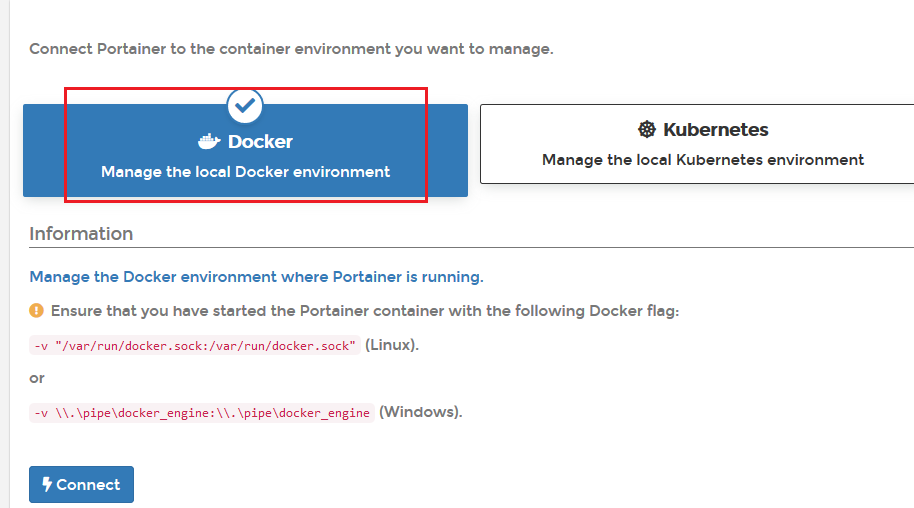
**★以https运行**

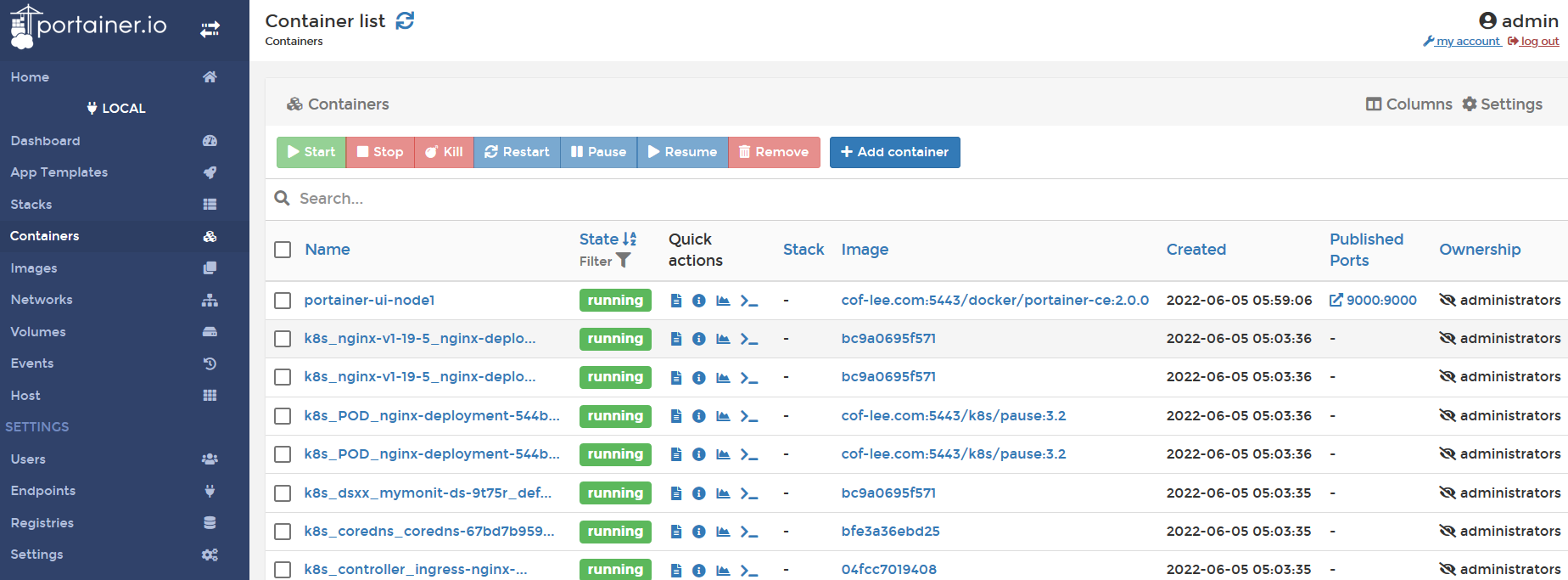
#使用ssl，提前准备证书并放在 /etc/portainer-cert/ 目录下

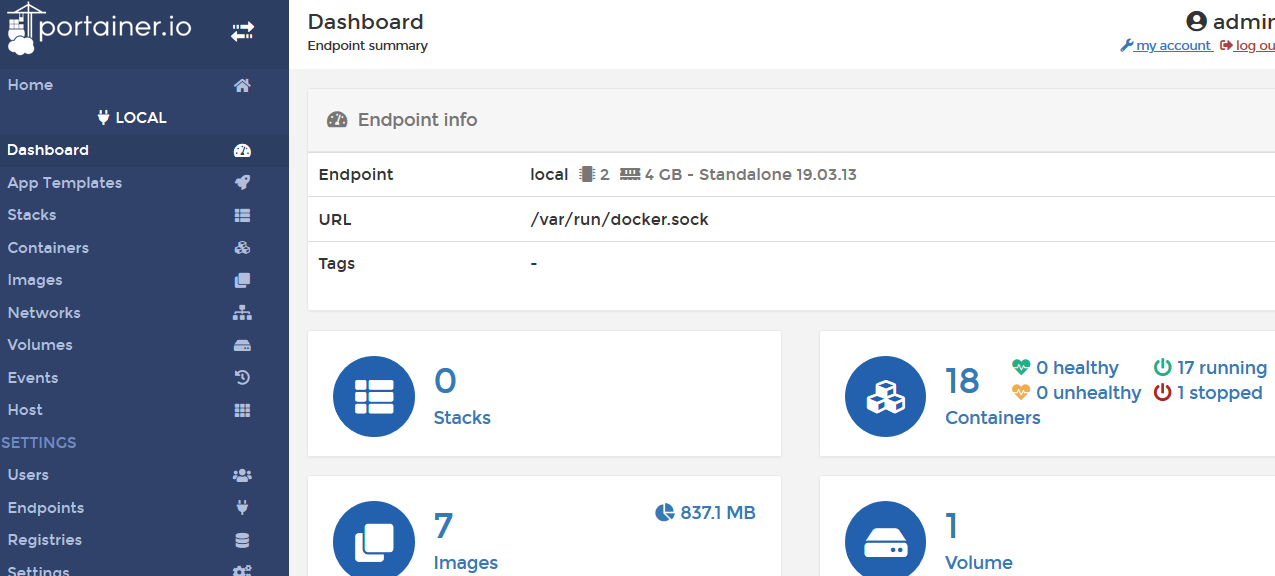
|  |  |
| --- | --- |
| docker run -d -p 9443:9000 --restart=always \ |  |
| -v /var/run/docker.sock:/var/run/docker.sock \ |  |
| -v /etc/portainer-data:/data \ |  |
| -v /etc/portainer-cert:/etc/portainer-cert \ |  |
| --name portainer-ui-node1-ssl \ |  |
| cof-lee.com:5443/docker/portainer-ce:2.0.0 \ | #以下三行是传给此容器的参数，必须放最后 |
| --ssl \ |  |
| --sslcert /etc/portainer-cert/portainer-ui.crt \ |  |
| --sslkey /etc/portainer-cert/portainer-ui.key |  |

#容器启动后，登录https://x.x.x.x:9443/ 初次登录要求创建管理员账号









**第15章、docker-compose**

首先要安装有docker-compose工具，这个工具默认和docker-ce一起安装了

docker自带有docker-compose，路径为/usr/libexec/docker/cli-plugins/docker-compose

# ln -s /usr/libexec/docker/cli-plugins/docker-compose /usr/bin/docker-compose

# docker-compose --version

#如果默认没有docker-compose，需要自己去下载，地址为：<https://github.com/docker/compose/releases/>

选择较新版本（若要支持多种cpu架构的镜像，需要使用v2.0及以上版本的harbor）



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/bin/目录下，重命名为docker-compose

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

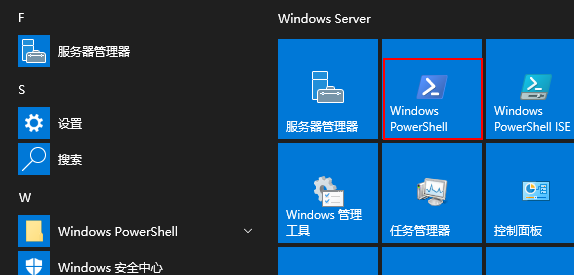
# chmod +x /usr/bin/docker-compose #添加可执行权限

# docker-compose --version #测试命令是否能用

**第16章、winServ 2019安装docker**

**①Power Shell命令行安装docker**

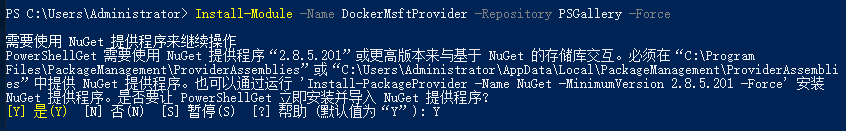
首先以管理员身份打开powershell

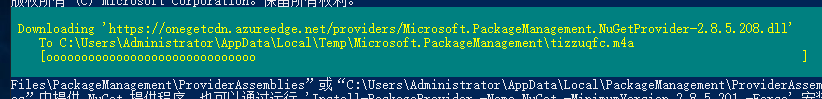


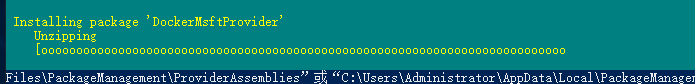
1.安装docker厂商支持模块

PS> Install-Module -Name DockerMsftProvider -Repository PSGallery -Force





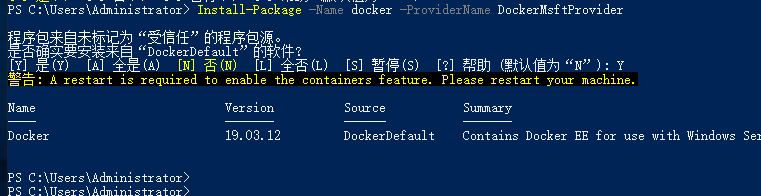




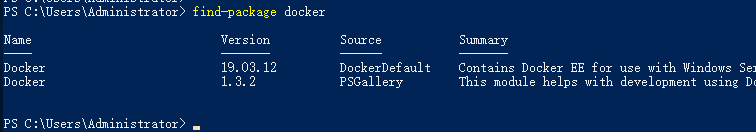
2.使用此厂商模块安装docker

PS> Install-Package -Name docker -ProviderName DockerMsftProvider





>find-package docker 查看一下安装的包，有19.03.xx的版本就表示安装成功了

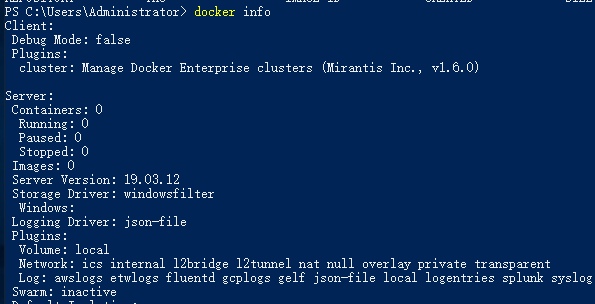


3.安装完成后，需要重启

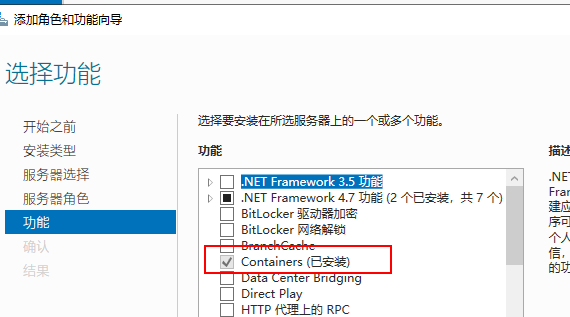


4.重启之后，以管理员身份运行PS命令行

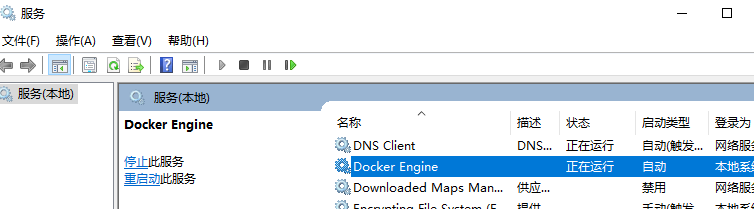
PS> docker info #查看docker相关信息



注意：使用ps命令安装是要求联网的，且自动帮我们安装了Containers容器服务



5. services.msc 打开服务管理，确保Docker Engine服务是自动启动的就行了



**②离线安装docker**

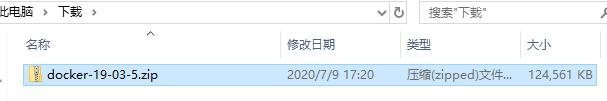
1.先在能联网的计算机上下载适合windows server 2019的docker离线安装包，比如19.03版

下载地址：（大概125MB）

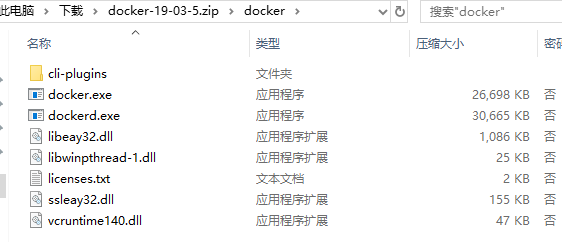
<https://download.docker.com/components/engine/windows-server/19.03/docker-19.03.5.zip>

或：<https://dockermsft.blob.core.windows.net/dockercontainer/docker-19-03-5.zip>

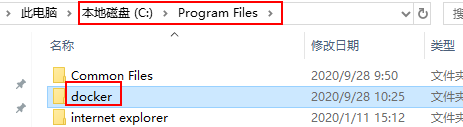
然后想办法复制安装包到目标win server 2019服务器上

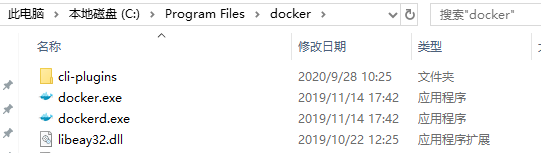


下载的zip包大概125MB，里面是一个名为docker的文件夹，文件夹里是docker服务相关的程序和数据文件

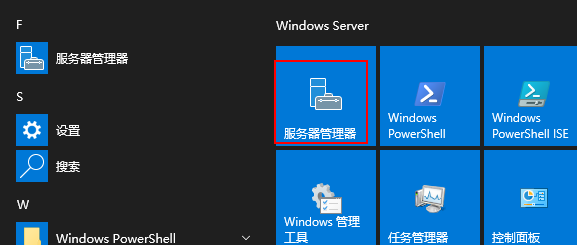


2.把这个压缩包里的文件解压到C:\ProgramFiles\docker\目录下

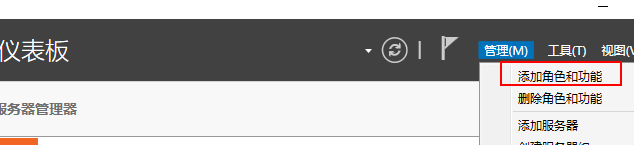




3.接下来先安装Containers服务

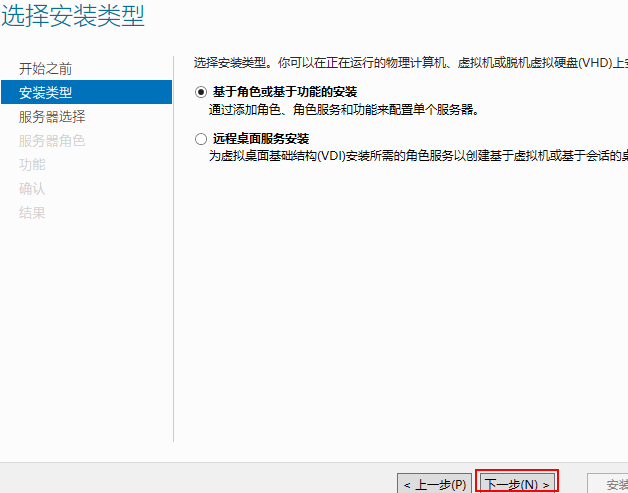


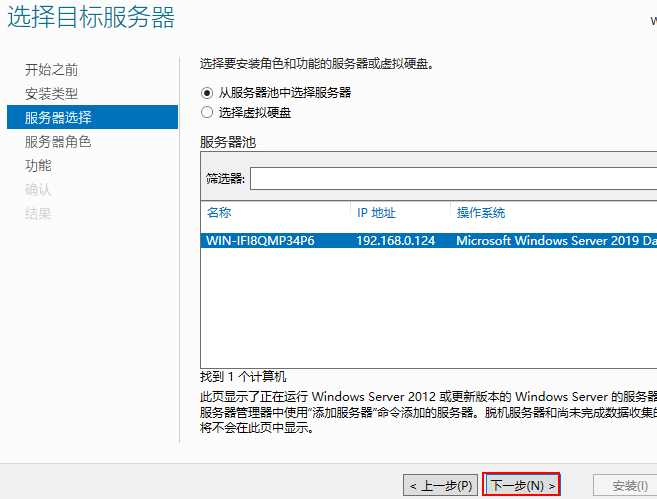
打开服务器管理器

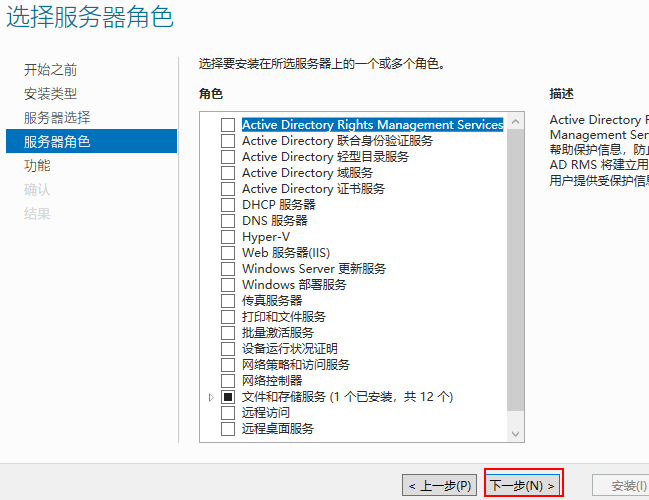


点击仪表板右上角的“管理”→“添加角色和功能”

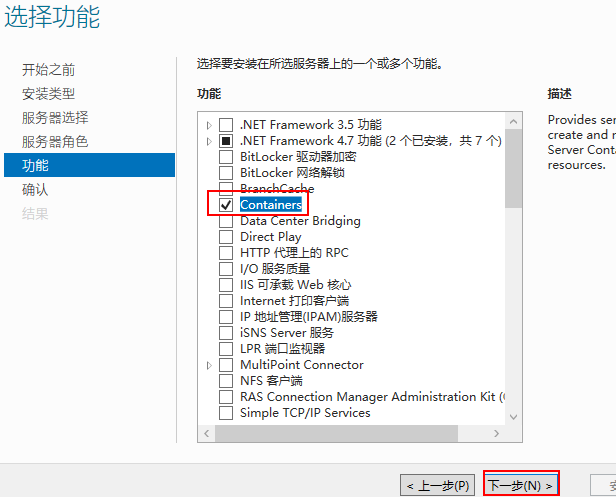


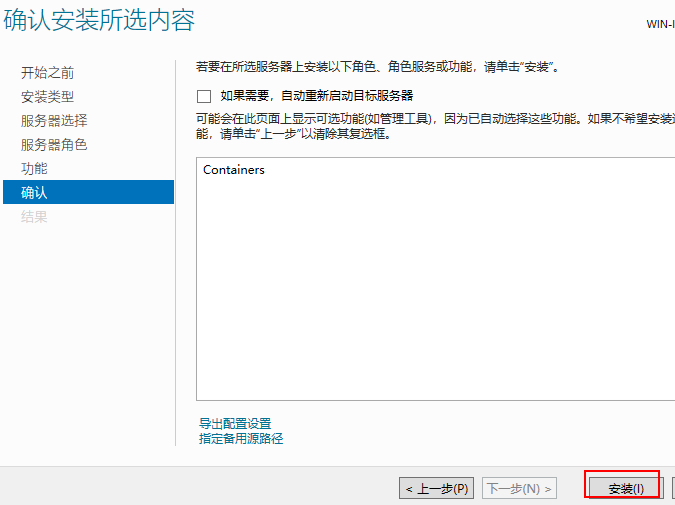




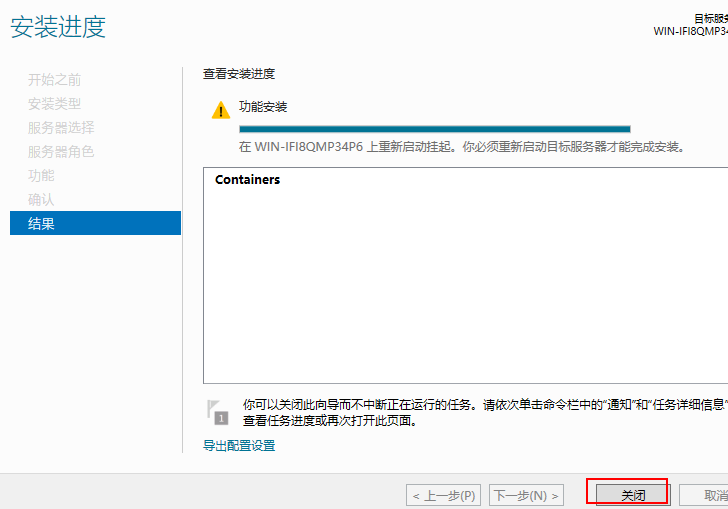


4.以上几步直接点“下一步”，在“功能”这里，勾选Containers，点击下一步





5.点击“安装”

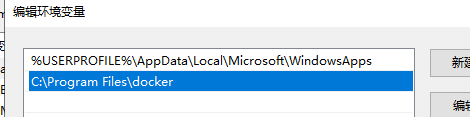


6.然后手动重启服务器

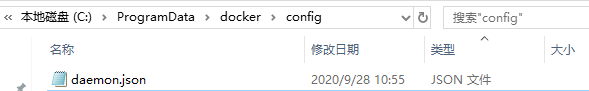
7.重启之后，添加环境变量



在Path变量后追加一个值C:\Program Files\docker



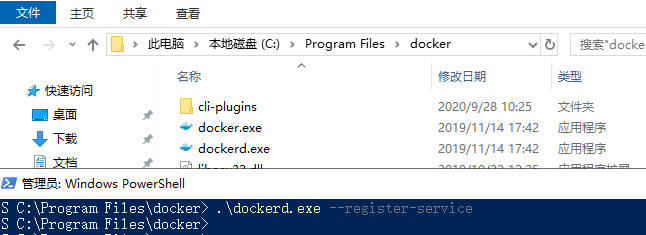
8.然后在 C:\ProgramData\目录下创建docker\config子目录，在config子目录下创建一个名为daemon.json的文件，内容可为空



9.将dockerd注册为服务

在C:\Program Files\docker目录下打开命令行，输入命令：

> docker.exe --register-service

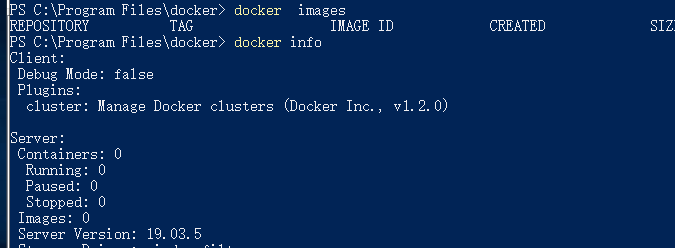


10.启动docker服务



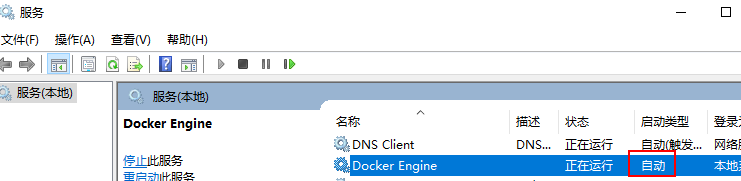
11.验证docker命令是否可用

输入docker info 等命令



可见Docker已经启动了，

12.确保Docker Engine服务为自动启动的就行了



**③切换至Linux内核**

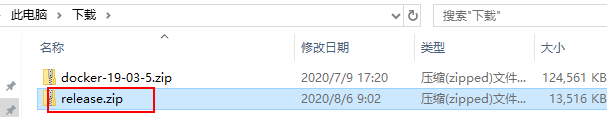
如果是在win 10或win server 2019上使用Docker Desktop安装包进行安装的话，默认是必须要求开启Hyper-V服务，才能安装成功，可以设置为使用Linux内核



，但在win server 2019里，使用PS命令安装docker或使用docker离线安装包进行安装的话，可以不开启Hyper-V服务，只需要开启Containers容器服务，这样默认就只是windows内核，只可使用基于windows内核的docker镜像

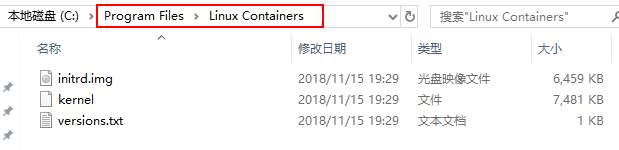
1.若也想使用Linux内核的镜像，得安装LCOW（Linux Container over Windows）

下载地址：<https://github.com/linuxkit/lcow/releases> 去找较新版本的release.zip

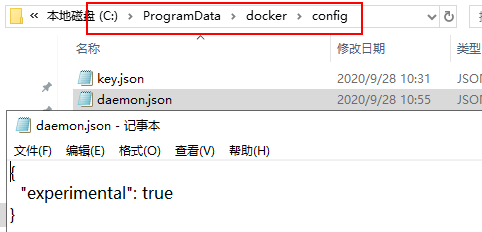


下载的release.zip大概14MB，里面有3个文件，是Linux内核和启动文件

2.创建 C:\Program Files\Linux Containers\目录，并将release.zip里的文件复制到此目录下



3.再编辑docker配置文件 C:\ProgramData\docker\config\daemon.json文件，内容如下：



4.然后设置PS的环境变量

PS> [Environment]::SetEnvironmentVariable("LCOW\_SUPPORTED","1","Machine")



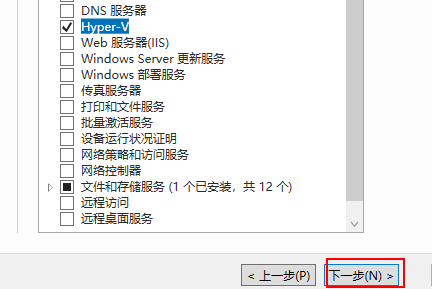
5.再安装Hyper-V服务

在管理器控制器里，点击右上角的“管理”→“添加角色和功能”

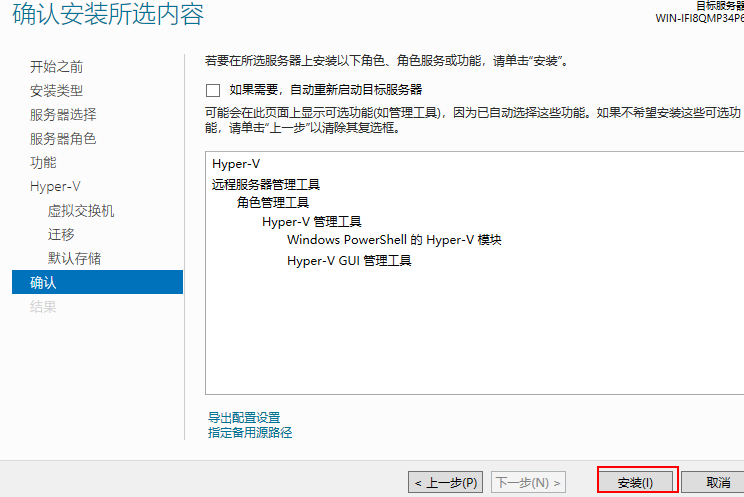


在“服务器角色”里，勾选Hyper-V





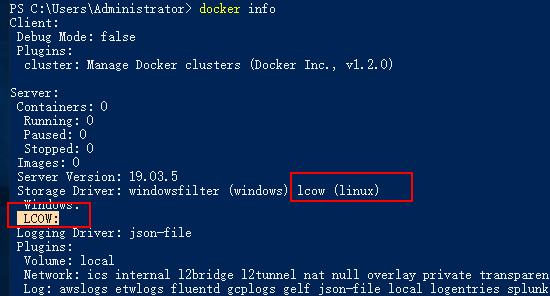
然后一直默认，下一步

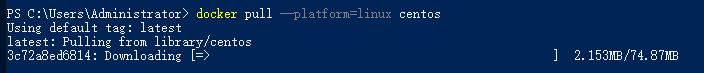


点击“安装”

6.安装完成后，重启

7.重启之后就可验证是否切换为Linux内核了，



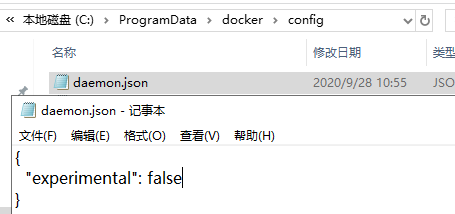


要运行linux内核的镜像时要带上--platform=linux 参数

8.若想切换回windows内核，可以将环境变量去除

PS> [Environment]::SetEnvironmentVariable("LCOW\_SUPPORTED",$null,"Machine")

并修改配置C:\ProgramData\docker\config\daemon.json



最后重启docker服务即可

PS> restart-service docker

**④ps命令卸载docker**

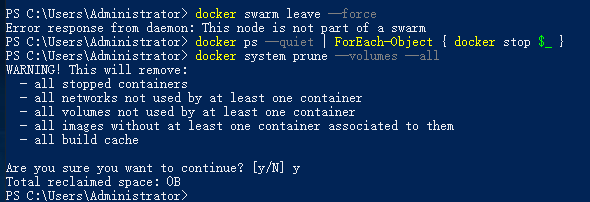
前提是使用ps命令在线安装的docker

1.关闭docker运行的相关服务

> docker swarm leave --force

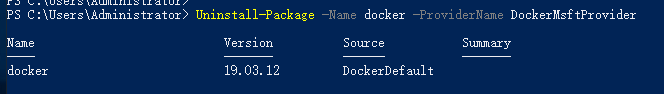
> docker ps --quiet | ForEach-Object { docker stop $\_ }

> docker system prune --volumes --all



2.卸载docker安装包

> Uninstall-Package -Name docker -ProviderName DockerMsftProvider



3.移除厂商支持模块

> Uninstall-Module -Name DockerMsftProvider



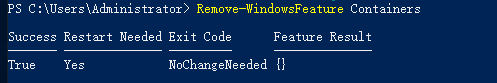
4.删除docker数据目录

> Remove-Item "C:\ProgramData\Docker" -Recurse



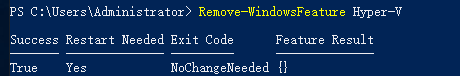
5.移除Containers服务

> Remove-WindowsFeature Containers



6.移除Hyper-V服务，（非必选）

> Remove-WindowsFeature Hyper-V



7.如果使用了Linux内核，则要删除C:\Program Files\Linux Containers\目录

8.重启操作系统

