第三章：Docker容器进阶--创建镜像、数据管理、网络

**一、Docker镜像创建方法--基于已有镜像、基于本地模板；**

**二、Docker镜像创建方法--基于dockerfile创建；**

**三、构建本地私有仓库；**

**四、Docker的数据管理；**

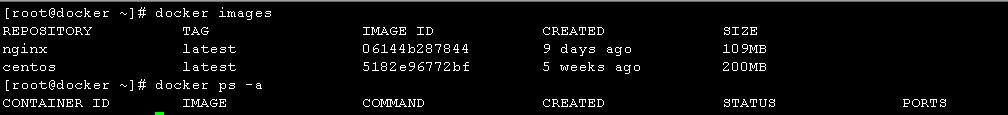
**五、Docker的网络管理；**

**六、Docker的存储类型；**

**一、Docker镜像创建方法--基于已有镜像、基于本地模板；**

**基于已有镜像：**

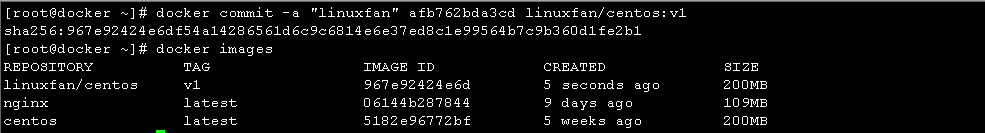
1.准备镜像：



2.准备容器：

****

**3.将容器创建成为docker镜像文件：**

****

**基于本地模板：**

1.下载本地操作系统模板：

https://download.openvz.org/template/precreated/ ##openvz开源项目

2.上传模板，导入到docker镜像中：

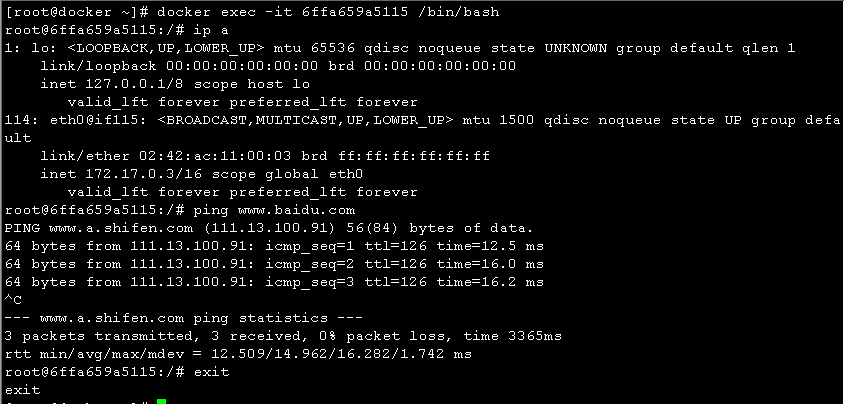
[root@docker ~]# ls ubuntu-14.04-x86\_64-minimal.tar.gz

ubuntu-14.04-x86\_64-minimal.tar.gz



3.使用镜像创建容器：





**二、Docker镜像创建方法--基于dockerfile创建；**

**概述：**除了手动生成docker镜像之外，还可以通过dockerfile自动生成镜像，dockerfile是由一组指定组成的文件，其中每条指令对应linux中的一条命令，docker程序将读取dockerfile中的指令生成指定镜像；

**结构：**

1.基础镜像信息；

2.维护者信息；

3.镜像操作指令；

4.容器启动时执行指令；

**规则：**dockerfile每行支持一条指令，每条指令可携带多个参数，支持使用以#号开头的注释；

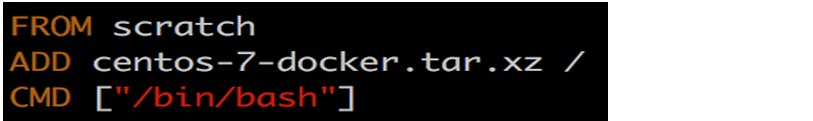
**\*注意：每一条指令就相当于给镜像加了一层，一个镜像不能超过 127 层，请惜字如金！**

**样例文件：**

1.最小的镜像hello-world，只有一个/hello程序，连基本的/bin、/etc/等都没有；



2. 基础base镜像，系统centos镜像，只有官方给予的最基本的目录，好多软件包是没有的；



3.用户手动基于centos编写的基础镜像；



**注解：**



|  |  |
| --- | --- |
| FROM 镜像 | 指定一个基础的镜像源，在此基础上，进行封装，scratch镜像表示为一无所有； |
| MAINTAINER 作者 | 指定新镜像的作者 |
| RUN 命令 | 在所基于镜像的基础上执行命令并且提交到新的镜像中； |
| WORKDIR 目录位置 | 类似于cd命令，更改执行命令时的基础环境； |
| EXPOSE 端口号 | 指定镜像创建后容器开启时中监听的端口，用于外界宿主机互联访问； |
| ENV 环境变量 变量值 | 设置容器的环境变量的值，被后边的RUN所使用； |
| ADD 源文件 /源目录 目标文件 /目标目录 | 将宿主机的文件或者目录复制到镜像的位置，注意：宿主机的文件或者目录必须与Dockerfile文件位于同位置，如若源文件是一个压缩包，会自动解压，源文件可以是一个URL； |
| COPY 源文件 /源目录 目标文件 /目标目录 | 与ADD功能类似，注：压缩包不能自动解压，也不能访问网络URL资源； |
| VOLUME ["目录"] | 实现数据卷资源持久化，将宿主机的目录资源挂载到容器中； |
| USER 用户名 | 指定运行该容器的用户； |
| CMD ["命令","参数1","参数2"] | 指定容器在开启时加载的命令，通常设置为一些自启动的服务； |
| ONBUILD | 指定当前镜像作为其他新镜像创建时指定基础镜像时，所需要指定的命令； |

在编写dockerfile时，有严格的格式需要遵循：

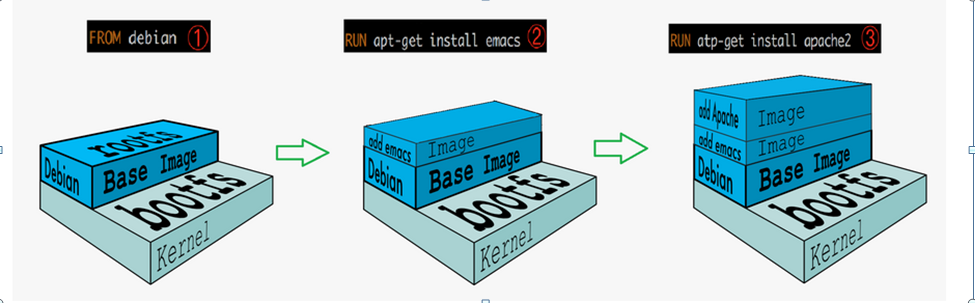
1.第一行必须使用FROM指定所基于镜像名称；

2.之后使用MAIN指定镜像的用户信息；

3.然后都是镜像操作相关指令，如RUN、ADD等，每执行一条，就会给基础镜像添加上新的一层；

4.最后指定CMD指令，用来指定容器运行时执行的命令操作；

**镜像分层结构详解：**



**解析：**新镜像是从 base 镜像一层一层叠加生成的。每安装一个软件，就在现有镜像的基础上增加一层，问为什么 Docker 镜像要采用这种分层结构呢？

最大的一个好处就是 - 共享资源。

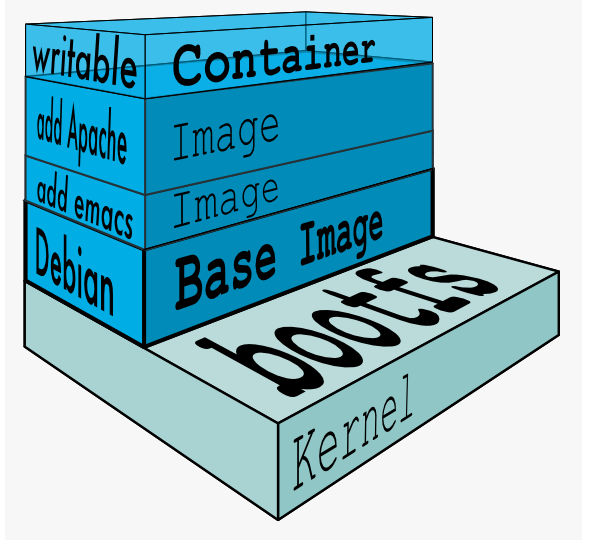
比如：有多个镜像都从相同的 base 镜像构建而来，那么 Docker Host 只需在磁盘上保存一份 base 镜像；同时内存中也只需加载一份 base 镜像，就可以为所有容器服务了。而且镜像的每一层都可以被共享；

如果多个容器共享一份基础镜像，当某个容器修改了基础镜像的内容，比如 /etc 下的文件，这时其他容器的 /etc 是否也会被修改？

答案是不会！  
修改会被限制在单个容器内。

**容器的Copy-on-Write 特性：**

可写的容器层：当容器启动时，一个新的可写层被加载到镜像的顶部，这一层通常被称作“容器层”，“容器层”之下的都叫“镜像层”；



所有对容器的改动 - 无论添加、删除、还是修改文件都只会发生在容器层中；

只有容器层是可写的，容器层下面的所有镜像层都是只读的；

1.添加文件：在容器中创建文件时，新文件被添加到容器层中；

2.读取文件：在容器中读取某个文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后打开并读入内存；

3.修改文件：在容器中修改已存在的文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后修改之；

4.删除文件：在容器中删除文件时，Docker 也是从上往下依次在镜像层中查找此文件。找到后，会在容器层中记录下此删除操作；

只有当需要修改时才复制一份数据，这种特性被称作 Copy-on-Write。可见，容器层保存的是镜像变化的部分，不会对镜像本身进行任何修改；

这样就解释了我们前面提出的问题：容器层记录对镜像的修改，所有镜像层都是只读的，不会被容器修改，所以镜像可以被多个容器共享。

**案例：**

1.编辑Dockerfile文件：

[root@docker ~]# mkdir apache

[root@docker ~]# cd apache/

[root@docker apache]# vi Dockerfile

FROM centos

MAINTAINER The Centos Project <cloud@linuxfan.cn>

RUN yum -y update

RUN yum -y install httpd

EXPOSE 80

ADD index.html /var/www/html/index.html

ADD run.sh /run.sh

RUN chmod 755 /run.sh

CMD ["/run.sh"]

2.准备Dockerfile文件中调用的脚本：

[root@docker apache]# vi run.sh

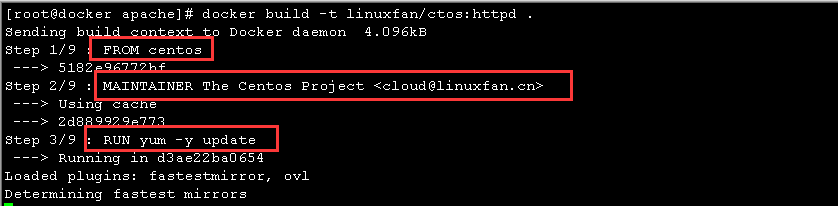
#!/bin/bash

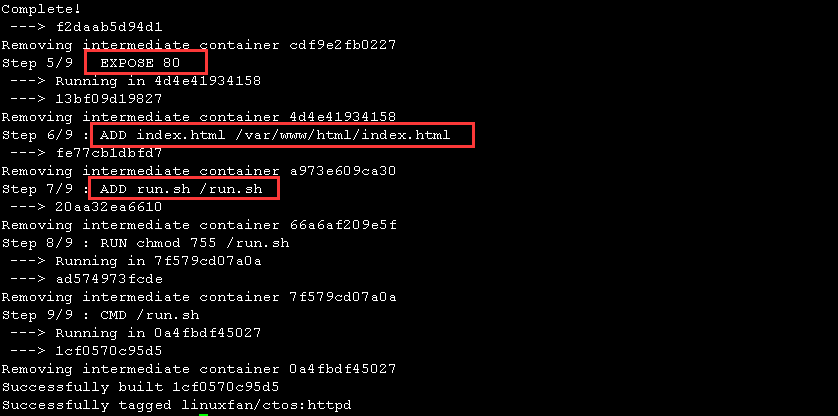
rm -rf /run/httpd/\*

exec /usr/sbin/apachectl -D FOREGROUND

[root@docker apache]# echo "welcome to docker" >index.html

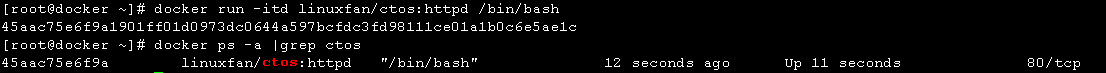
3.使用Dockfile文件生成镜像：



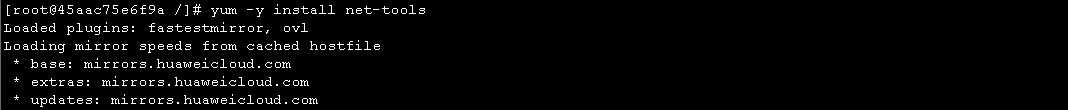




4.使用已经存在的镜像文件创建新的容器：

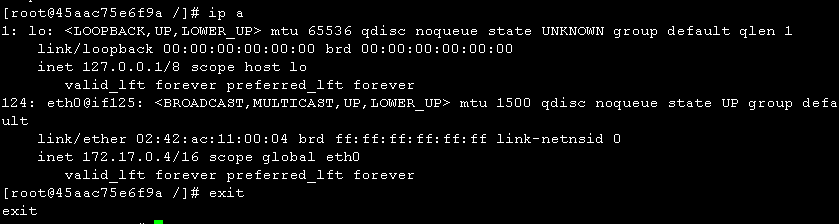




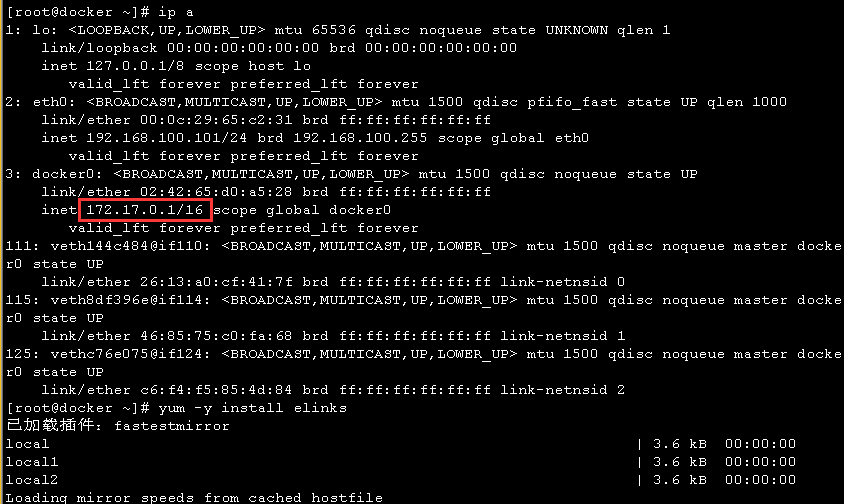








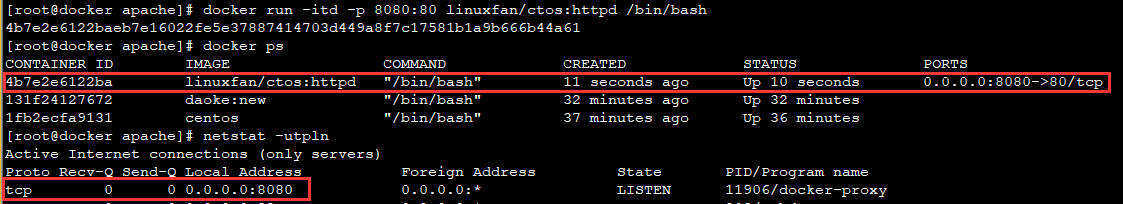
5.使用docker节点测试访问容器的httpd服务：

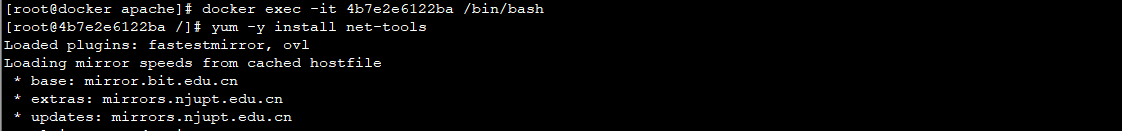


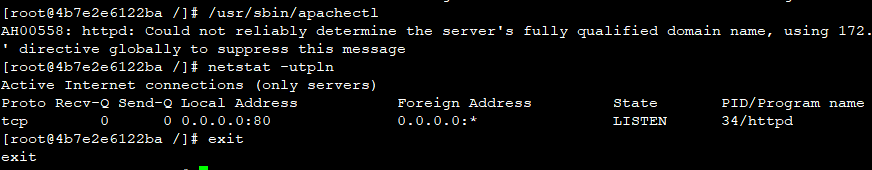


6.将容器的80端口映射到docker节点的8080端口进行发布，外网客户端测试连接：



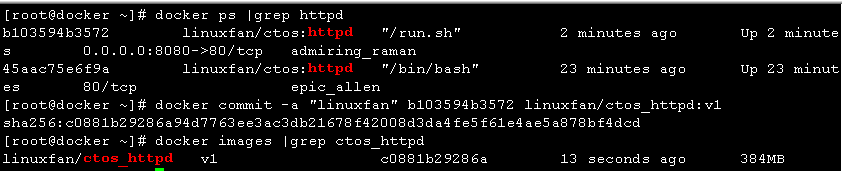




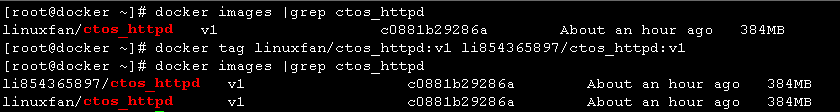


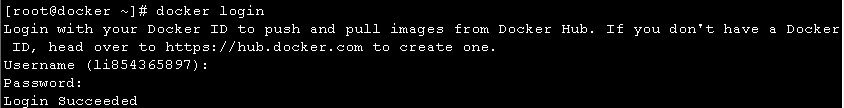


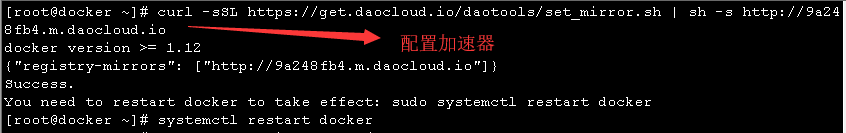
7.将容器的当前状态制作成为镜像文件：

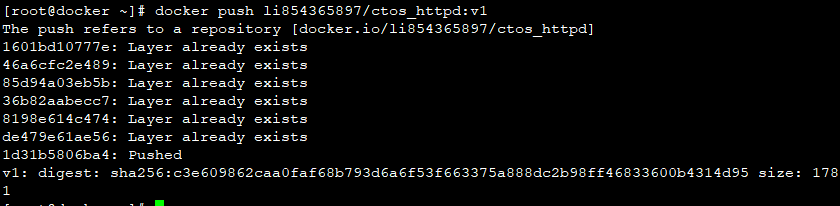


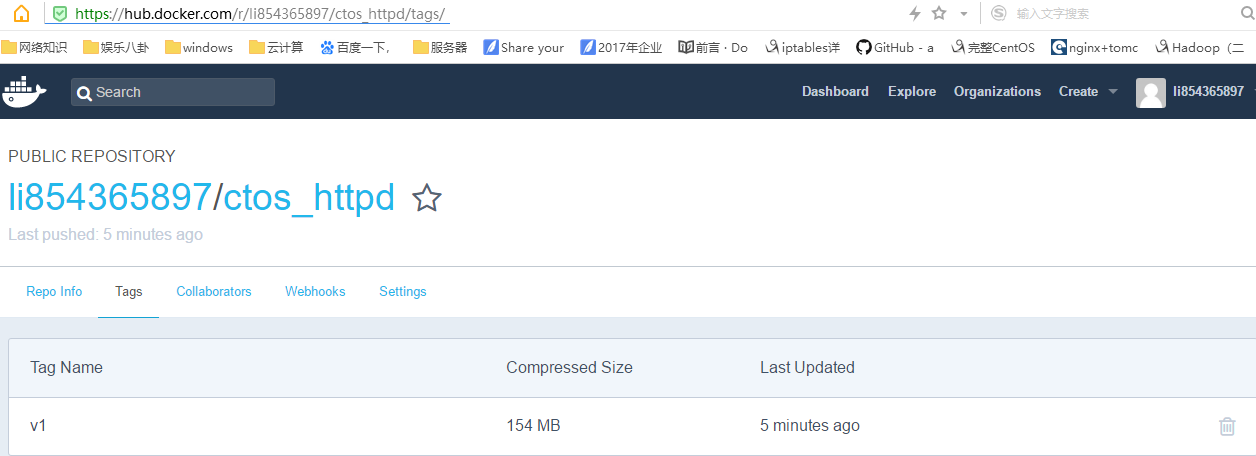
8.将上述镜像文件上传到docker hub公共仓库：







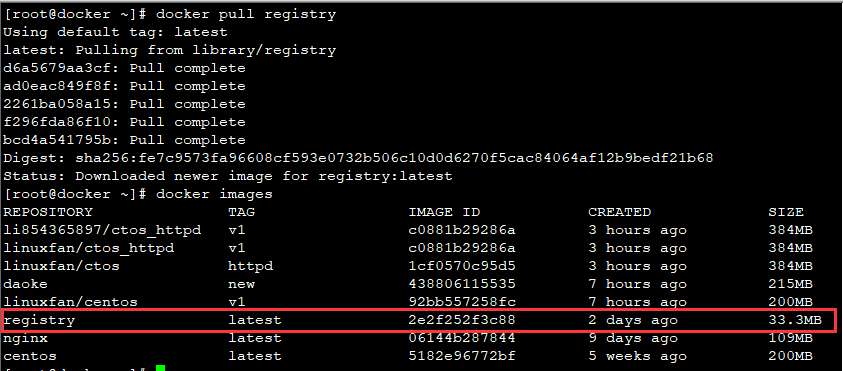




**三、构建本地私有仓库：**

**概述：**搭建本地私有仓库容器，进行存放本地镜像；

1.下载私有仓库镜像：



2.编辑json文件指定私有仓库http地址和端口：

[root@docker ~]# vi /etc/docker/daemon.json ##全部删除替换掉

{"insecure-registries": ["192.168.100.101:5000"]}

[root@docker ~]# systemctl restart docker

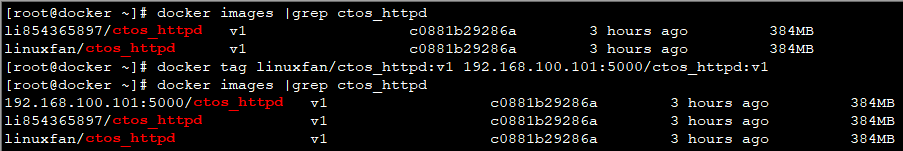
3.使用下载好的私有仓库的镜像文件创建容器：



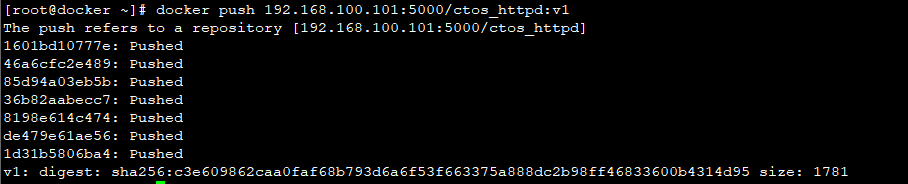


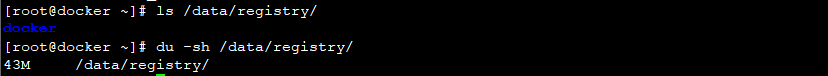
选项：-v /var/lib/registry/指定私有仓库容器中存放镜像的位置；

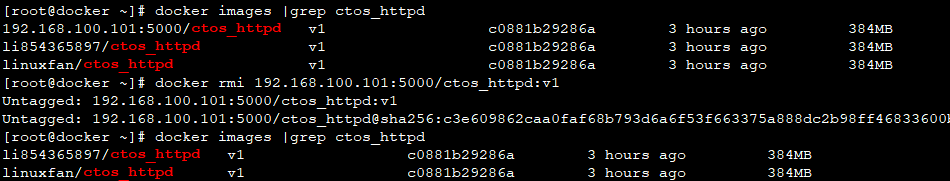
/data/registry/ 将私有仓库容器中的上述位置映射到docker节点的本地目录；

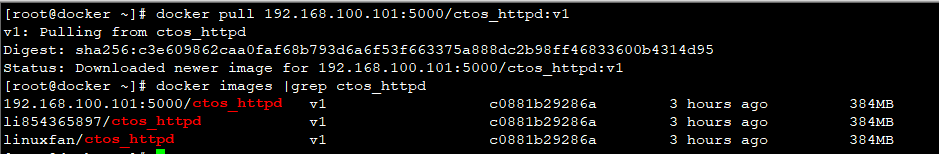


4.上传镜像文件到私有仓库容器中并删除镜像文件验证；









**四、Docker的数据管理：**

**概述：**在docker中，为了方便查看容器内产生的数据或者将多个容器内的数据进行共享，就会涉及到容器的数据管理概念；

**方式：**

1.数据卷：data volumes

2.数据卷容器：data volumes containers

3.数据cp复制：docker cp ...

**方式一：数据卷**

**概述：**数据卷是供容器使用的特殊目录，位于容器中，可将宿主机的目录挂载到数据卷上，对数据卷的修改操作立即可见，并且更新数据不会影响镜像，从而实现数据在宿主机与容器之间的迁移，数据卷的使用类似于linux系统中的mount操作；

1.创建数据卷的容器基于centos镜像：

[root@docker ~]# docker run -itd -v /linuxfan/data1:/data1 -v /linuxfan/data2:/data2 --name linuxcon centos /bin/bash

前者：宿主机位置

后者：容器位置

b43ef13d7ce8f57ddafe3aa2d35e497fe5947c92aa1123502c3e389e16698119

[root@docker ~]# docker ps |grep linuxcon

b43ef13d7ce8 centos "/bin/bash" 15 seconds ago Up 15 seconds linuxcon

[root@docker ~]# ls /linuxfan/

data1 data2

选项：-v指定容器内的数据卷，将本地目录/linuxfan/data1和/linuxfan/data2挂载到容器目录/data1和/data2中；

2.连接容器，查看数据卷目录：

[root@docker ~]# docker exec -it b43ef13d7ce8 /bin/bash

[root@b43ef13d7ce8 /]# ls -ld /data1/

drwxr-xr-x 2 root root 6 Sep 14 22:53 /data1/

[root@b43ef13d7ce8 /]# ls -ld /data2/

drwxr-xr-x 2 root root 6 Sep 14 22:53 /data2/

[root@b43ef13d7ce8 /]# exit

3.在宿主机创建文件验证容器内的文件变化：

[root@docker ~]# touch /linuxfan/data1/data1.file

[root@docker ~]# touch /linuxfan/data2/data2.file

[root@docker ~]# docker exec -it b43ef13d7ce8 /bin/bash

[root@b43ef13d7ce8 /]# ls /data1/

data1.file

[root@b43ef13d7ce8 /]# ls /data2/

data2.file

[root@b43ef13d7ce8 /]# exit

4.在容器内创建文件验证宿主机内的文件变化：

[root@docker ~]# docker exec -it b43ef13d7ce8 /bin/bash

[root@b43ef13d7ce8 /]# touch /data1/data1.txt

[root@b43ef13d7ce8 /]# touch /data2/data2.txt

[root@b43ef13d7ce8 /]# exit

exit

[root@docker ~]# ls /linuxfan/data1/

data1.file data1.txt

[root@docker ~]# ls /linuxfan/data2/

data2.file data2.txt

**方式二：数据卷容器**

**概述：**数据卷容器就是一个普通的容器，专门提供数据卷给其他容器挂载使用；

**使用方法：**首先需要创建一个容器作为数据卷（可以在方式一数据卷的基础上），之后在其他容器创建时使用--volumes-from挂载源数据卷容器使用，如此就可以实现多个容器之间的数据共享；

1.基于centos镜像创建容器linuxcon-1，创建时指定linuxcon容器作为数据卷容器；

[root@docker ~]# docker run -itd --volumes-from linuxcon --name linuxcon-1 centos /bin/bash

e4b9e0ef5063133aaf088cbc03b10c6788d21974c91b4f2c20f649f34f576b5e

[root@docker ~]# docker ps |grep linuxcon

e4b9e0ef5063 centos "/bin/bash" 7 seconds ago Up 6 seconds linuxcon-1

b43ef13d7ce8 centos "/bin/bash" 11 minutes ago Up 11 minutes

2.进入linuxcon-1容器，查看目录是否存在；

[root@docker ~]# docker exec -it e4b9e0ef5063 /bin/bash

[root@e4b9e0ef5063 /]# ls -ld /data1/

drwxr-xr-x 2 root root 39 Sep 14 22:57 /data1/

[root@e4b9e0ef5063 /]# ls -ld /data2/

drwxr-xr-x 2 root root 39 Sep 14 22:58 /data2/

[root@e4b9e0ef5063 /]# exit

3.在宿主机和linuxcon容器分别创建文件验证通过“宿主机--linuxcon数据卷容器--linuxcon-1”容器是否同步数据；

[root@docker ~]# touch /linuxfan/data1/data1.bak

[root@docker ~]# docker exec -it b43ef13d7ce8 /bin/bash

[root@b43ef13d7ce8 /]# touch /data2/data2.bak

[root@b43ef13d7ce8 /]# exit

exit

[root@docker ~]# docker exec -it e4b9e0ef5063 /bin/bash

[root@e4b9e0ef5063 /]# ls /data1/

data1.bak data1.file data1.txt

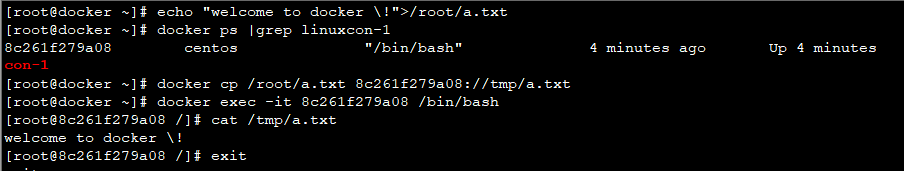
[root@e4b9e0ef5063 /]# ls /data2/

data2.bak data2.file data2.txt

[root@e4b9e0ef5063 /]# exit

**方式三：数据cp复制：docker cp ...**

上传：



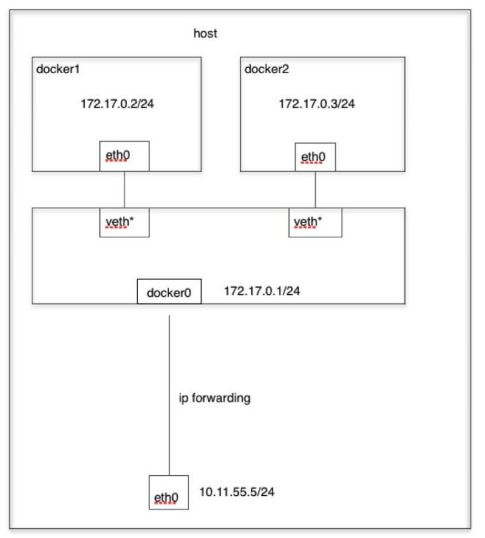
下载：



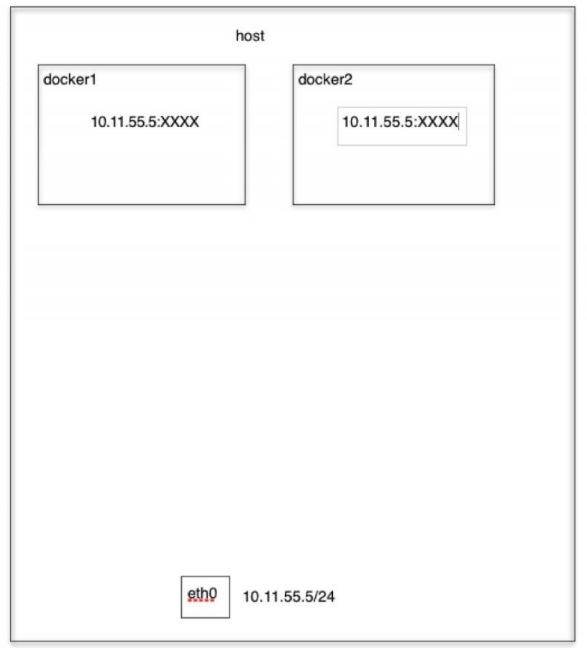
**五、Docker的网络管理；**

**网络模式：**

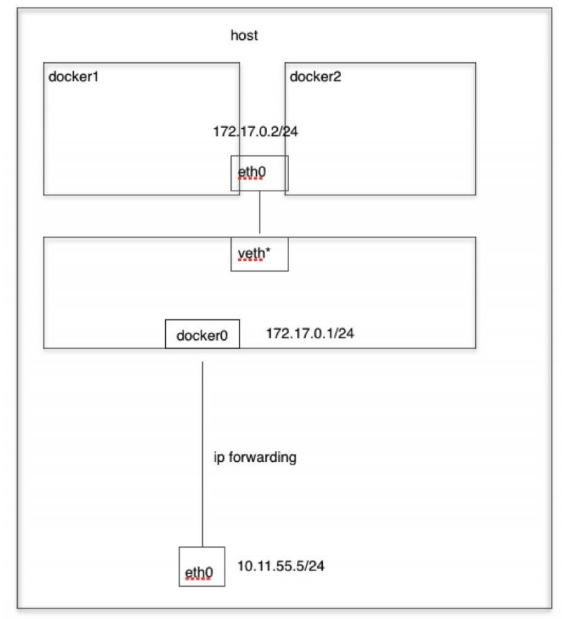
1.Bridge模式：默认的网络模式，在启动docker进程后，宿主机会创建名为docker0的虚拟网桥，ip地址为172.17.0.1，所有容器的默认网络会自动获取一个172.17.0.0网络的地址，实际上宿主机在此模式默认自动做了一条NAT转换规则并且开启路由转发；



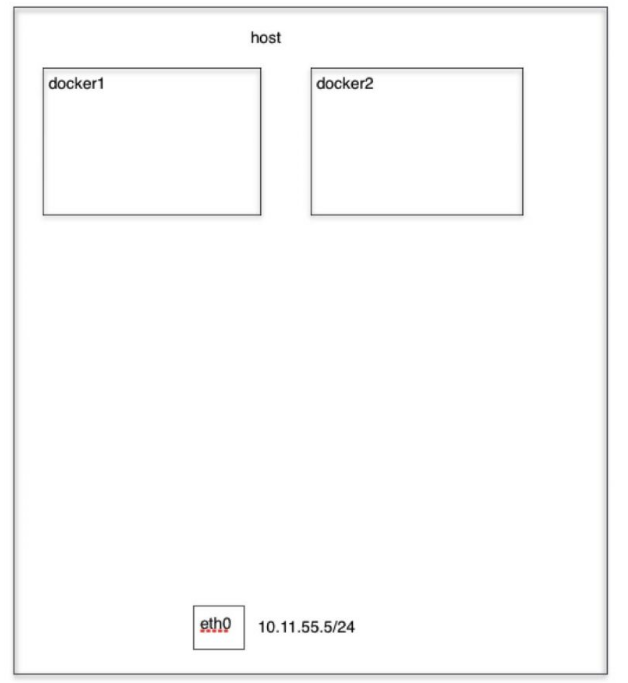
2.Host模式：如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的；



3.Container模式：指定新创建的容器和已经存在的一个容器共享一个 Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 IP，而是和一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信；



4.None模式： Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息。需要我们自己为Docker容器添加网卡、配置IP等；



**配置网络模式：**

* 配置默认桥接模式：默认；
* 配置host模式：使用 --net=host指定；

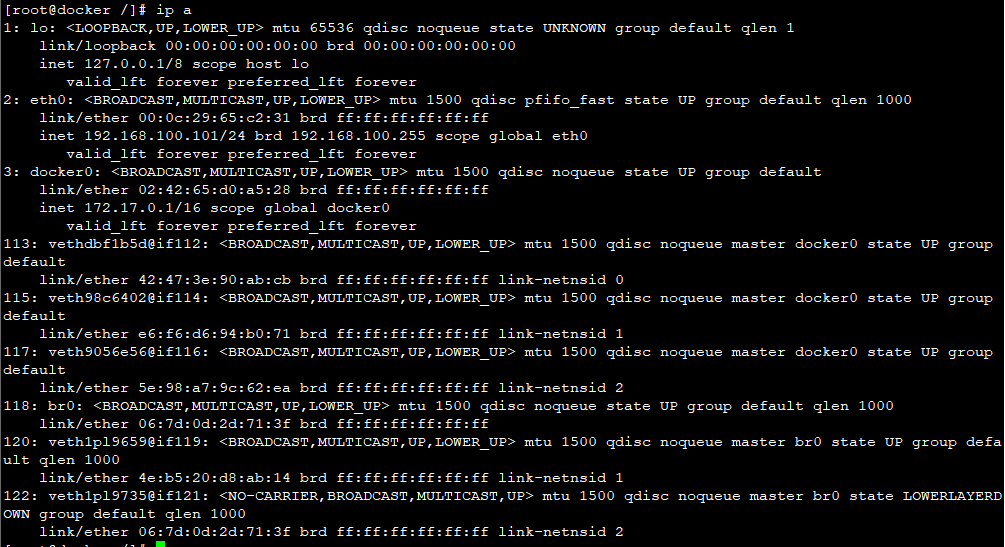
[root@docker ~]# docker run -itd --net=host --name centos7-3 centos /bin/bash

[root@docker ~]# docker ps |grep centos7-3

8b8585dd3ec2 centos "/bin/bash" 3 minutes ago Up 3 minutes centos7-3

[root@docker ~]# docker exec -it 8b8585dd3ec2 /bin/bash

[root@docker /]# yum -y install iproute net-tools httpd



[root@docker /]# /usr/sbin/httpd

[root@docker /]# netstat -utpln |grep 80

tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 127/httpd

tcp 0 0 0.0.0.0:8081 0.0.0.0:\* LISTEN -

[root@docker /]# exit

exit

[root@docker ~]# netstat -utpln |grep 80

tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 10684/httpd

tcp 0 0 0.0.0.0:8081 0.0.0.0:\* LISTEN 9202/docker-proxy

* 配置自定义桥接模式：容器与宿主机处于同一网络；

[root@docker ~]# yum -y install bridge-utils

[root@docker ~]# cp /etc/sysconfig/network-scripts/ifcfg-eth0 /etc/sysconfig/network-scripts/ifcfg-br0

[root@docker ~]# vi /etc/sysconfig/network-scripts/ifcfg-br0

TYPE=Bridge

BOOTPROTO=static

DEFROUTE=yes

PEERDNS=yes

PEERROUTES=yes

IPV4\_FAILURE\_FATAL=yes

NAME=br0

DEVICE=br0

ONBOOT=yes

NM\_CONTROLLED=no

IPADDR=192.168.100.101

PREFIX=24

GATEWAY=192.168.100.100

DNS1=192.168.100.100

:wq

[root@docker ~]# vi /etc/sysconfig/network-scripts/ifcfg-eth0

TYPE=Ethernet

BOOTPROTO=none

DEFROUTE=yes

PEERDNS=yes

PEERROUTES=yes

IPV4\_FAILURE\_FATAL=yes

NAME=eth0

DEVICE=eth0

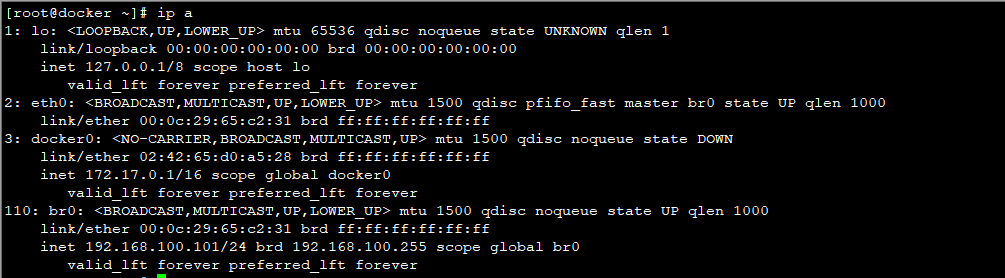
ONBOOT=yes

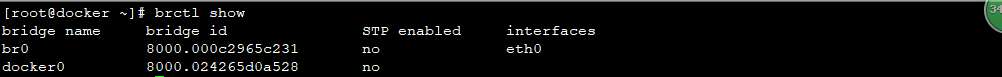
NM\_CONTROLLED=no

BRIDGE=br0

:wq

[root@docker ~]# systemctl restart network





[root@docker ~]# git clone https://github.com/jpetazzo/pipework

正克隆到 'pipework'...

remote: Counting objects: 501, done.

remote: Total 501 (delta 0), reused 0 (delta 0), pack-reused 501

接收对象中: 100% (501/501), 172.97 KiB | 161.00 KiB/s, done.

处理 delta 中: 100% (264/264), done.

[root@docker ~]# cp pipework/pipework /usr/local/bin/

[root@docker ~]# vi /etc/sysconfig/docker

OPTIONS=' --selinux-enabled -b=br0 --fixed-cidr=192.168.100.0/24 --log-driver=journald --signature-verification=false'

:wq

[root@docker ~]# systemctl restart docker

[root@docker ~]# docker run -itd --net=none --name centos7-101 centos /bin/bash ##注意创建容器时指定--net=none不采用默认桥接方式自动产生ip，使用pipeword工具设置ip地址

ade2fb172937c97516c500d0b05284069b7e67b385aae84672e498594a82ea4d

[root@docker ~]# docker ps |grep centos7-101

ade2fb172937 centos "/bin/bash" 9 seconds ago Up 9 seconds centos7-101

[root@docker ~]# pipework br0 centos7-101 192.168.100.110/24@192.168.100.100 ##使用pipework命令为容器配置ip地址。指定桥接网卡、容器ID、ip地址、子网掩码、网关

，pipeword工具功能强大，划分vlan等；

[root@docker ~]# docker exec -it centos7-101 /bin/bash

[root@ade2fb172937 /]# yum -y install iproute httpd net-tools

[root@ade2fb172937 /]# ip a

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

119: eth1@if120: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000

link/ether a6:a6:92:78:01:d8 brd ff:ff:ff:ff:ff:ff link-netnsid 0

inet 192.168.100.110/24 brd 192.168.100.255 scope global eth1

valid\_lft forever preferred\_lft forever

[root@ade2fb172937 /]# ping -c 2 www.baidu.com

PING www.a.shifen.com (61.135.169.121) 56(84) bytes of data.

64 bytes from 61.135.169.121 (61.135.169.121): icmp\_seq=1 ttl=128 time=2.62 ms

[root@ade2fb172937 /]# /usr/sbin/httpd

[root@ade2fb172937 /]# netstat -utpln

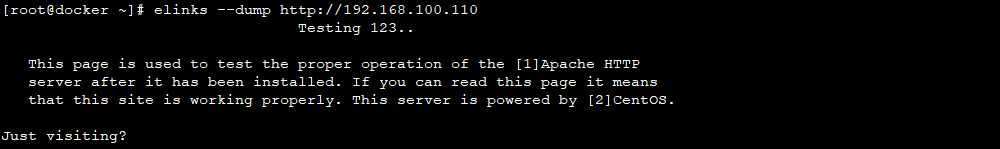
Active Internet connections (only servers)

Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name

tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 106/httpd

[root@ade2fb172937 /]# exit

exit



* 配置container模式：使用 --net=container:container\_id/container\_name；
* 配置none模式：使用 --net=none；

**方式：**

一、端口映射：将容器内部的端口号码映射到docker宿主节点的端口，实现外部主机访问docker容器内的服务；

二、容器互联：通过容器的名称在容器之间建立一条专门的网络通信隧道从而实现容器的互联，简单点说：就是在源容器和接收容器之间建立一条隧道，接收容器可以看到源容器指定的信息；

**方式一：端口映射：**

**注：在run 容器时，-P选项可以将docker容器内的端口随机映射到宿主机的49000-49900范围；**

1.创建基于centos镜像的容器并且实现端口映射，将容器的httpd的80端口映射到宿主机的8081端口；

[root@docker ~]# docker run -itd -e"container=docker" --privileged=true -p 8081:80 centos /usr/sbin/init

727fd0bea64f5b7e28903d727dc23f61b2b7a065818bf03417a4d2c072728b5d

[root@docker ~]# docker ps |grep init

bfc36fcc7857 centos "/usr/sbin/init" About a minute ago Up About a minute 0.0.0.0:8081->80/tcp pensive\_lovelace

2.为容器安装httpd服务并且启动服务；

[root@docker ~]# docker exec -it 727fd0bea64f /bin/bash

[root@727fd0bea64f /]# yum -y install net-tools iproute httpd

[root@727fd0bea64f /]# systemctl start httpd

[root@727fd0bea64f /]# ip a |grep 172

inet 172.17.0.5/16 scope global eth0

[root@727fd0bea64f /]# netstat -utpln |grep 80

tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 183/httpd

[root@727fd0bea64f /]# exit

[root@docker ~]# docker ps |grep init

bfc36fcc7857 centos "/usr/sbin/init" About a minute ago Up About a minute 0.0.0.0:8081->80/tcp pensive\_lovelace

[root@docker ~]# netstat -utpln |grep 8081

tcp 0 0 0.0.0.0:8081 0.0.0.0:\* LISTEN 14123/docker-proxy

4.外部节点访问测试服务；

[root@docker ~]# elinks --dump http://192.168.100.101:8081

...

4. http://www.centos.org/

5. http://www.internic.net/whois.html

6. http://www.centos.org/

**方式二：容器互联：**

1.创建源容器：

[root@docker ~]# docker images |grep centos |grep latest

centos latest 5182e96772bf 5 weeks ago 200MB

[root@docker ~]# docker run -itd --name centos7-1 centos /bin/bash

12a1d1312abf261bae7036143b5a8575851b99a42e05b769ff428b5f8698f14f

[root@docker ~]# docker ps |grep centos7-1

12a1d1312abf centos "/bin/bash" 54 seconds ago Up 53 seconds

2.创建接收容器，指定centos容器为源容器：

[root@docker ~]# docker run -itd --name centos7-2 --link centos7-1:centos7-1 centos /bin/bash

ea4baf4bdd1d041620a31c47fa32b37b4e4d3511d64fb0333c1c9ca39c7c0367

[root@docker ~]# docker ps |grep centos7-2

ea4baf4bdd1d centos "/bin/bash" 5 seconds ago Up 4 seconds

3.连接到接收容器进行测试容器之间的连通性：

[root@docker ~]# docker exec -it ea4baf4bdd1d /bin/bash

[root@ea4baf4bdd1d /]# ping centos7-1

PING centos7-1 (172.17.0.6) 56(84) bytes of data.

64 bytes from centos7-1 (172.17.0.6): icmp\_seq=1 ttl=64 time=0.121 ms

64 bytes from centos7-1 (172.17.0.6): icmp\_seq=2 ttl=64 time=0.036 ms

^C

--- centos7-1 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 999ms

rtt min/avg/max/mdev = 0.036/0.078/0.121/0.043 ms

[root@ea4baf4bdd1d /]# exit

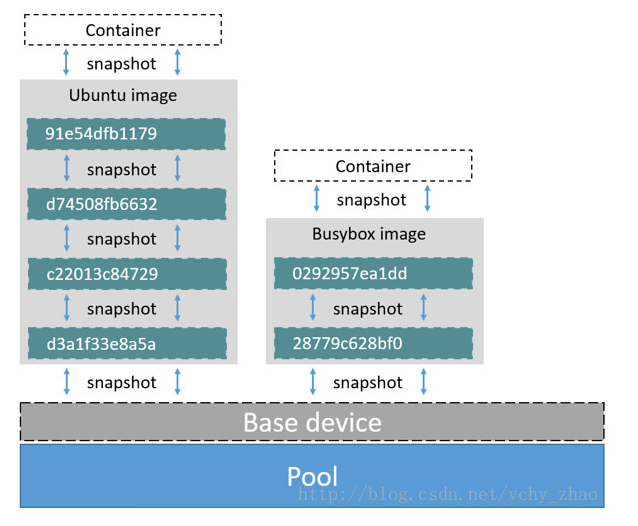
**六、Docker的存储类型：**

**存储类型：https://blog.csdn.net/vchy\_zhao/article/details/70238690#1**

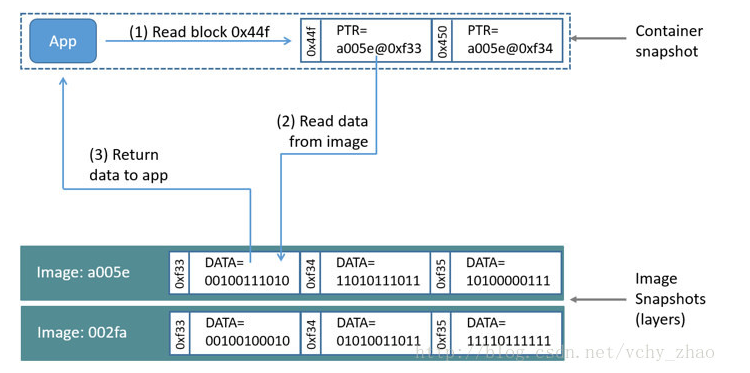
* **aufs：**AUFS是Docker最先使用的storage driver，Docker在Debian，Ubuntu系的系统中默认使用aufs，它技术很成熟，社区支持也很好，特点如下：   
  - 容器启动速度很快   
  - 存储空间利用很高效   
  - 内存的利用很高效

AUFS是一种联合文件系统，意思是它将同一个主机下的不同目录堆叠起来（类似于栈）成为一个整体，对外提供统一的视图。AUFS是用联合挂载来做到这一点。   
AUFS使用单一挂载点将多个目录挂载到一起，组成一个栈，对外提供统一的视图，栈中的每个目录作为一个分支。栈中的每个目录包括联合挂载点都必须在同一个主机上。   
在Docker中，AUFS实现了镜像的分层。AUFS中的分支对应镜像中的层。   
此外，容器启动时创建的读写层也作为AUFS的一个分支挂载在联合挂载点上。

* **devicemapper：**在RedHat系中使用device mapper。device mapper在Linux2.6内核中被并入内核，它很稳定，也有很好的社区支持。device mapper将所有的镜像和容器存储在它自己的虚拟设备上，这些虚拟设备是一些支持写时复制策略的快照设备。device mapper工作在块层次上而不是文件层次上，这意味着它的写时复制策略不需要拷贝整个文件。 device mapper创建镜像的过程如下：   
  - 使用device mapper的storge driver创建一个精简配置池；精简配置池由块设备创建；- 接下来创建一个基础设备；   
  - 每个镜像和镜像层都是基础设备的快照；这写快照支持写时复制策略，这意味着它们起始都是空的，当有数据写入时才耗费空间。在device mapper作为storage driver的系统中，容器层container layer是它依赖的镜像的快照。与镜像一样，container layer也支持写时复制策略，它保存了所有对容器的更改。当有数据需要写入时，device mapper就为它们在资源池中分配空间；

****

**读取数据：**

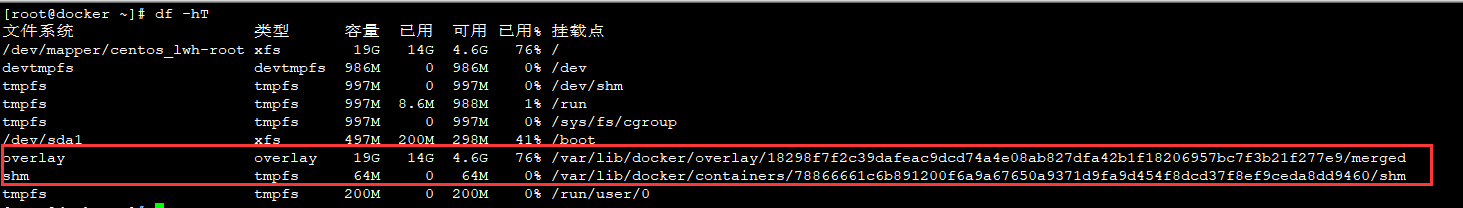
****

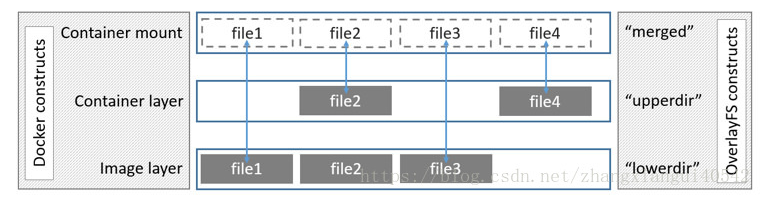
写入数据：

对容器的写操作由“需要时分配”策略完成。更新已有数据由“写时复制”策略完成，这些操作都在块的层次上完成，每个块的大小为64KB；   
向容器写入56KB的新数据的步骤如下：   
- 进程向容器发出写56KB数据的请求；   
- device mapper的“需要时分配”策略分配一个64KB的块给容器快照(container snapshot)；如果要写入的数据大于64KB，就分配多个大小为64KB的块。   
- 将数据写入新分配的块中；

* **overlay存储驱动：**与AUFS相似，也是一种联合文件系统(union filesystem)，与AUFS相比，其稳定性还有待测试；

OverlayFS在Linux主机上有两个目录但呈现给用户就像只有一个目录一样。这些目录称之为层（layers），这些层的统一呈现通过联合挂载实现。OverlayFS将底层目录视作lowerdir，将上层目录称作upperdir，通过叫做merged目录对外暴露统一的视图。

以下图片展示了Docker的镜像和Docker容器的分层结构。 镜像层就是lowerdir目录，而uppperdir目录就是容器层。统一对外暴露的视图就是merged目录，该目录也是容器的挂载点。Docker结构映射OverlayFS结构图如下；



**读取数据：**

如果文件在容器层不存在，则从lowdir中读取；

只在容器层存在，则直接从容器中读取改文件；

文件存在容器和镜像层，容器层upperdir会覆盖镜像层lowdir中的文件；

**修改数据：**

首次写入： 在upperdir中不存在，overlay和overlay2执行copy\_up操作，把文件从lowdir拷贝到upperdir，由于overlayfs是文件级别的（即使文件只有很少的一点修改，也会产生的copy\_up的行为）；  
 copy\_up操作只发生在文件首次写入，以后都是只修改副本 ；  
 overlayfs只适用两层，因此性能很好，查找搜索都更快；

**性能测试：**

总体上，overlay要比aufs和device mapper快一点。下面是对overlay性能影响较大的几个方面：   
- 页缓存(page caching)：overlayfs支持页缓存的共享，这意味着多个使用同一文件的容器可以共享同一页缓存，这使得overlayfs具有很高的内存使用效率；   
- copy-up操作：overlay的拷贝操作工作在文件层面上，也就是对文件的第一次修改需要复制整个文件，这回带来一些性能开销，在修改大文件时尤其明显；  
但overlay的拷贝操作比aufs还是快一点，因为aufs有很多层，而overlay只有两层，所以overlay在文件的搜索方面相对于aufs具有优势；  
- i节点限制：使用overlay作为storage driver会消耗大量的i节点，随着镜像和容器数量的增长这种消耗尤其显著，这在一定程度上限制了overlay的使用；

* **overlay2存储驱动：**与overlay存储模式的本质区别是镜像层之间共享数据的方法不同

overlay共享数据方式是通过硬连接，而overlay2是通过每层的 lower文件；

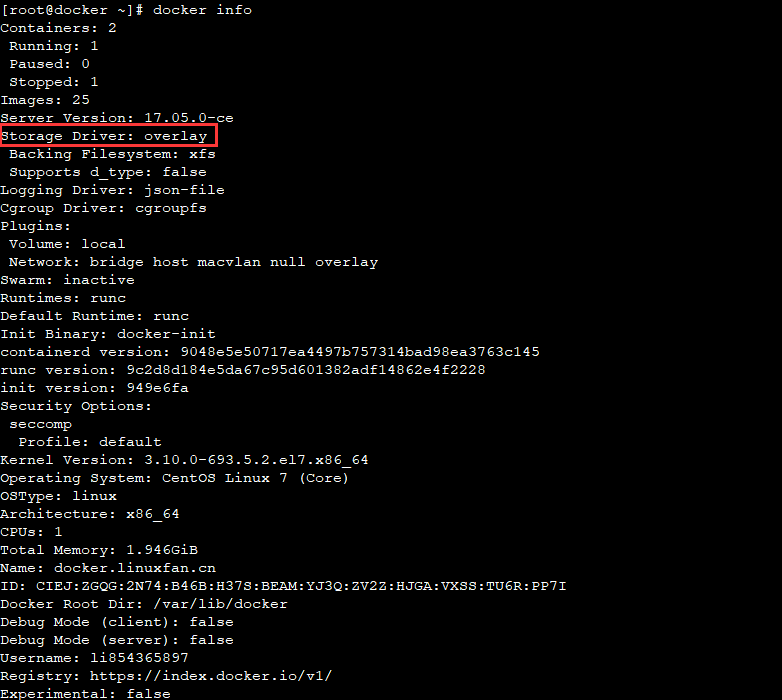


**案例：**

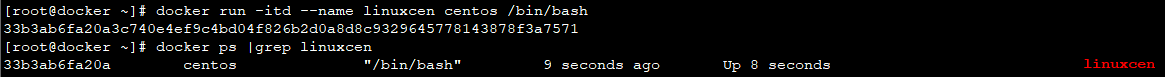
http://blog.51cto.com/nosmoking/1881033

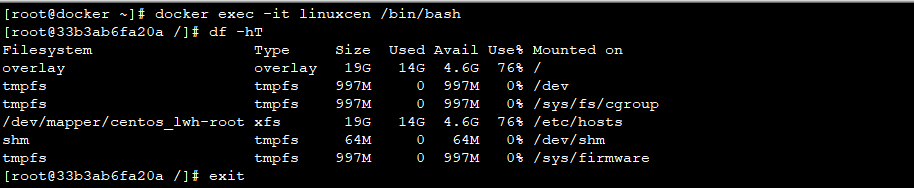
https://www.linuxidc.com/Linux/2015-01/112245.htm

1.查看当前docker服务的存储模式：

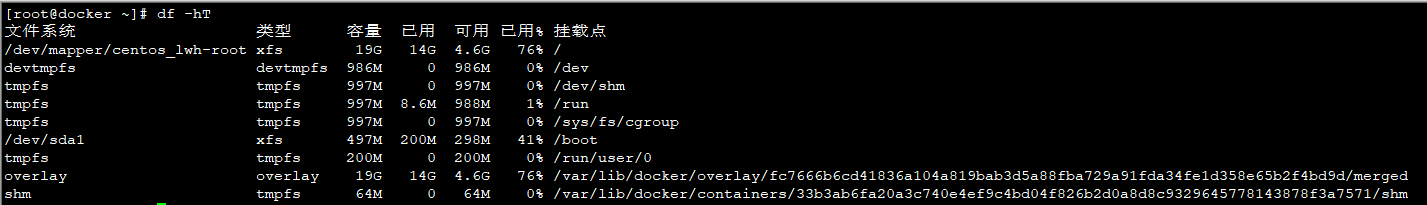


2.创建容器查看存储状态：

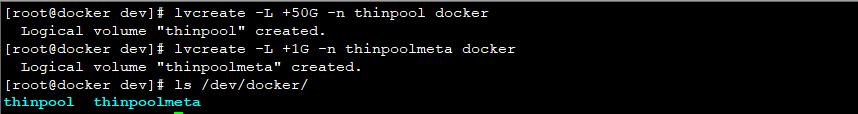


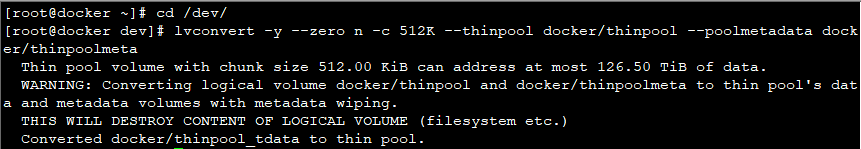


3.查看docker宿主机的磁盘存储状态：



4.添加三块20G的硬盘制作lvm逻辑卷，为devicemapper存储模式做数据存储；







5.修改docker的daemon配置文件指定存储模式和存储设备等；

第一种方式：

[root@docker ~]# vi /etc/docker/daemon.json ##不生效，所以修改docker启动脚本设置选项进行修改

{"registry-mirrors": ["http://9a248fb4.m.daocloud.io"]}

{

"storage-driver": "devicemapper",

"storage-opts": [

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

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

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

"dm.basesize=25G"

]

}

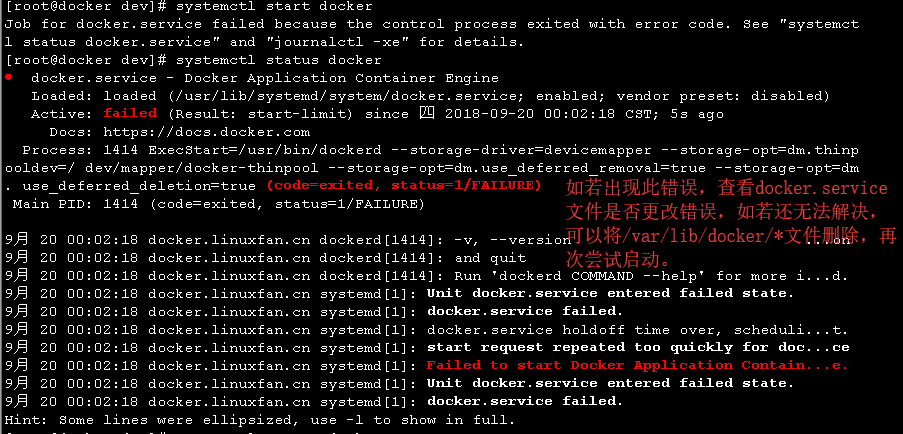
第二种方式（推荐）：

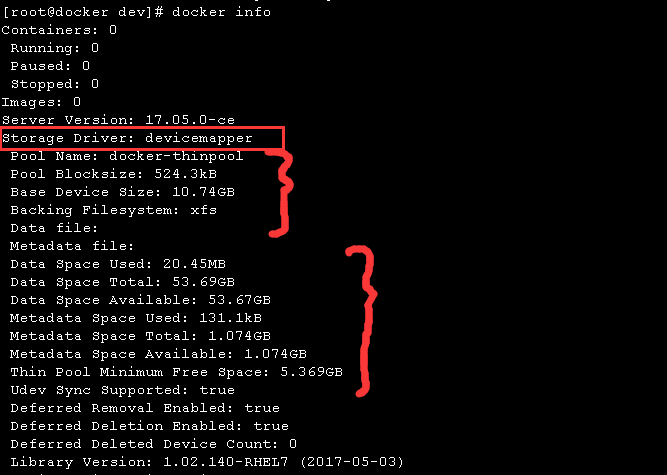
[root@docker ~]# vi /lib/systemd/system/docker.service

ExecStart=/usr/bin/dockerd --storage-driver=devicemapper --storage-opt=dm.thinpooldev=/dev/mapper/docker-thinpool --storage-opt=dm.use\_deferred\_removal=true --storage-opt=dm.use\_deferred\_deletion=true

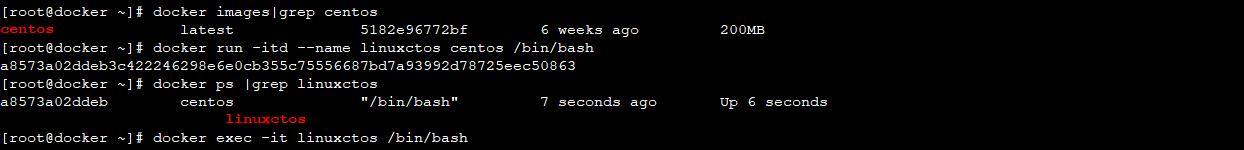
6.验证当前docker的overlay存储模式切换成为devicemapper；

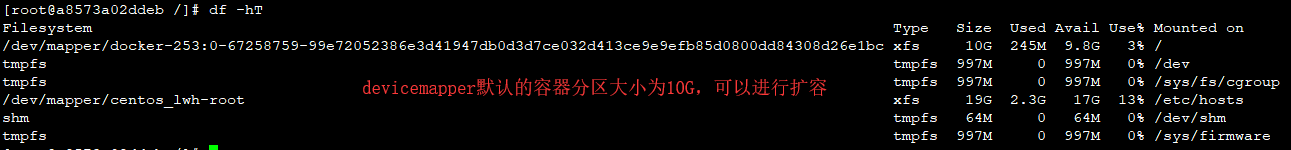


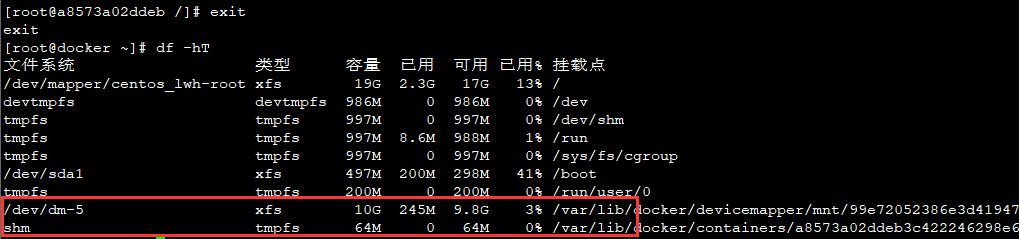




7.创建新的容器，验证磁盘存储的变化：







8.为容器磁盘存储空间进行扩容：

冷扩容：将容器删除（删除前可以进行备份，可以将其打包成为镜像，上传至仓库），重新创建容器，指定选项 -d --storage-opt dm.basesize=20G进行扩容；

热扩容：如下；

9.为容器磁盘存储空间进行热扩容：