# 部署

ps -ef|grep tomcat80\*|grep -v grep|awk '{print $2}'|xargs kill -9

权限

sudo chmod a+rw /var/run/docker.sock

## remove old version

$ sudo yum remove docker\*

install

$ sudo yum search docker

$ sudo yum -y install <FULLY-QUALIFIED-PACKAGE-NAME>

Service start

$ sudo systemctl start docker

testing

$ sudo docker run hello-world

镜像(https://hub.alauda.cn/repos/xuxinkun/kubernetes)

docker pull registry.alauda.cn/xuxinkun/kubernetes

/usr/sbin/sestatus -v

修改/etc/selinux/config 文件

将SELINUX=enforcing改为SELINUX=disabled

/etc/hosts

/etc/hostname

systemctl disable firewalld

systemctl stop firewalld

源

curl -o /etc/yum.repos.d/aliyum.repo http://mirrors.aliyun.com/repo/Centos-7.repo

curl -o /etc/yum.repos.d/docker-ce.repo https://download.docker.com/linux/centos/docker-ce.repo

cat> /etc/yum.repos.d/docker.repo<<EOF

[dockerrepo]  
name=Docker Repository  
baseurl=https://yum.dockerproject.org/repo/main/centos/7/  
enabled=1  
gpgcheck=1  
gpgkey=https://yum.dockerproject.org/gpg

EOF

yum install bridge-utils

yum install docker-engine-1.12.3-1.el7.centos

yumdownloader docker-engine-1.12.3-1.el7.centos

镜像

[root@rancher ~]# systemctl enable docker.service

Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service to /usr/lib/systemd/system/docker.service

vi /usr/lib/systemd/system/docker.service

--registry-mirror=https://pee6w651.mirror.aliyuncs.com

重新加载

[root@rancher ~]# systemctl daemon-reload

镜像

vi /etc/init.d/docker

exec="/usr/bin/$prog --registry-mirror=https://pee6w651.mirror.aliyuncs.com"

C:\Users\BIN\Documents\Tencent Files\20796698\Image\Group\5$}VC5Z`UVI8{`C6F(~C0%J.png

systemctl enable docker.service

systemctl docker start

# 命令

docker build创建镜像时需要Dockerfile文件. Dockerfile文件里装着一些命令集, build的过程, 会从上到下依次执行里面的命令.   
整个创建过程是首先从基础镜像启一个容器, 然后执行一条命令, 修改之后提交此容器为新镜像. 直到所有命令都执行完. 所以在得到最终构建的镜像时, 会生成很多"临时镜像". 如果因为某条命令出错而中断, 之前创建的"临时镜像"还在, 重新build就会从中断处继续, 而不是从头再来一遍, 很方便.

## Dockerfile里常用命令

* FROM <image name>: 指定构建使用的基础镜像
  + 例: FROM ubuntu:14.04
* MAINTAINER <author name>: 创建者信息
  + 例: MAINTAINER Laowen "qq05629@126.com"
* ENV: 设置环境变量
  + 例: ENV REFRESHED \_AT 2017-03-16
* RUN <command>: 在shell或者exec的环境下执行一条命令.RUN指令会在新创建的镜像上添加新的层面，接下来提交的结果可以用在Dockerfile的下一条指令中
  + 例: RUN apt-get -yqq update
* ADD <source> <destination>: 从当前目录复制文件到容器, source可以是URL或者是启动配置上下文中的一个文件, destination是容器内的路径. 会自动处理目录, 压缩包等情况
  + 例: ``
* COPY: 从当前目录复制文件到容器. 只是单纯地复制文件.
* VOLUME [ "/data" ]: 声明一个数据卷, 可用于挂载, []里面是路径
  + 例: VOLUME [ "/var/lib/redis", "/var/log/redis" ]
* USER <uid>: 镜像正在运行时设置的一个UID,RUN命令执行时的用户
* WORKDIR: 指定RUN、CMD与ENTRYPOINT命令的工作目录
  + 例: WORKDIR /opt/nodeapp
* ONBUILD: 前缀命令, 放在上面这些命令前面, 表示生成的镜像再次作为"基础镜像"被用于构建时要执行的命令
* ENTRYPOINT: 配置给容器一个可执行的命令,这意味着在每次使用镜像创建容器时一个特定的应用程序可以被设置为默认程序.同时也意味着该镜像每次被调用时仅能运行指定的应用.类似于CMD,Docker只允许一个ENTRYPOINT,多个ENTRYPOINT会只执行最后的ENTRYPOINT指令
  + 例: ENTRYPOINT [ "nodejs", "server.js" ]
* CMD: 提供了容器默认的执行命令,Dockerfile只允许使用一次CMD指令. 使用多个CMD只有最后一个指令生效
  + 例: CMD [ "/bin/true" ]
* EXPOSE <port>: 指定容器在运行时监听的端口
  + 例: EXPOSE 3000

# 例子

FROM centos:latest

镜像的作者

MAINTAINER benpao "[benpao@benpao.com](mailto:benpao@benpao.com)"

安装openssh-server和sudo软件包，并且将sshd的UsePAM参数设置成no

RUN yum install -y openssh-server sudo  
RUN sed -i 's/UsePAM yes/UsePAM no/g' /etc/ssh/sshd\_config

添加测试用户admin，密码admin，并且将此用户添加到sudoers里

RUN useradd admin

RUN echo "admin:admin" | chpasswd

RUN echo "admin ALL=(ALL) ALL" >> /etc/sudoers

修改root用户密码

RUN echo "root:benpao" | chpasswd

下面这两句比较特殊，在centos6上必须要有，否则创建出来的容器sshd不能登录

RUN ssh-keygen -t dsa -f /etc/ssh/ssh\_host\_dsa\_key  
RUN ssh-keygen -t rsa -f /etc/ssh/ssh\_host\_rsa\_key

启动sshd服务并且暴露22端口

RUN mkdir /var/run/sshd  
EXPOSE 22  
CMD ["/usr/sbin/sshd", "-D"]

docker build –t name:version .

# 云

docker run -d --restart=always --name apache\_web\_service --privileged=true -p 8888:80 -v/apps/webapp/html:/var/www/html/ docker.io/fdisk123/httpd

docker run -d --restart=always --name apache\_upload\_simple\_file --privileged=true -p 8081:80 -v /apps/upload:/var/www/html/ docker.io/fdisk123/httpd

docker run -d --restart=always --name mysql\_service --privileged=true -v /apps/database:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=chenshikun@123 -p 3306:3306 docker.io/mysql

docker run -d --restart=always --name redis\_service --privileged=true -v /apps/redis:/data -p 6379:6379 docker.io/redis

docker run -d --restart=always --name tomcat\_simple\_service --privileged=true -v /apps/webapp/webroot/simple:/apps/webapp/webroot/ -v /apps/conf/simple:/apps/conf/ -v /apps/upload/simple-web/file:/apps/webapp/upload/ -v /apps/log/simple:/usr/local/tomcat/logs/ -p 8080:8080 --link=redis\_service:redis --link=mysql\_service:mysql -e WRITE\_LOG=false docker.io/fdisk123/tomcat:8

curl 127.0.0.1:8080/simple-web/manager/appointment/get/1

docker run -d --restart=always --name nginx\_service --privileged=true -v /apps/conf/nginx:/etc/nginx/conf -p 80:80 --link=tomcat\_simple\_service:simple\_web --link=apache\_upload\_simple\_file:simple\_web\_file --link=apache\_web\_service:html docker.io/fdisk123/nginx

docker exec –it xx /bin/bash

mysql -u root -p

create database simple;

sudo docker run -d --restart=always --name mysql\_sonarqube\_service --privileged=true -v /apps/db:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=sonarqube -p 3306:3306 docker.io/mysql

docker run -d --restart=always --name sonarqube --link= mysql\_sonarqube\_service:db –e SONARQUBE\_JDBC\_USERNAME=root –e SONARQUBE\_JDBC\_PASSWORD=sonarqube –e SONARQUBE\_JDBC\_URL= jdbc:mysql://db:3306/sonar?useUnicode=true&characterEncoding=utf8 -p 9000:9000 –v /apps/sonarqube/data:/opt/sonarqube/data sonarqube

docker run -d --restart=always --privileged=true --name sonarqube --link mysql\_sonarqube\_service:db -e SONARQUBE\_JDBC\_USERNAME=root -e SONARQUBE\_JDBC\_PASSWORD=sonarqube -e SONARQUBE\_JDBC\_URL="jdbc:mysql://db:3306/sonar?useUnicode=true&characterEncoding=utf8" -p 9000:9000 -v /apps/sonarqube/data:/opt/sonarqube/data sonarqube

# 网络

### 1 host模式

众所周知，Docker使用了Linux的Namespaces技术来进行资源隔离，如PID Namespace隔离进程，Mount Namespace隔离文件系统，Network Namespace隔离网络等。一个Network Namespace提供了一份独立的网络环境，包括网卡、路由、Iptable规则等都与其他的Network Namespace隔离。一个Docker容器一般会分配一个独立的Network Namespace。但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。

例如，我们在10.10.101.105/24的机器上用host模式启动一个含有web应用的Docker容器，监听tcp80端口。当我们在容器中执行任何类似ifconfig命令查看网络环境时，看到的都是宿主机上的信息。而外界访问容器中的应用，则直接使用10.10.101.105:80即可，不用任何NAT转换，就如直接跑在宿主机中一样。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。

### 2 container模式

在理解了host模式后，这个模式也就好理解了。这个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。

### 3 none模式

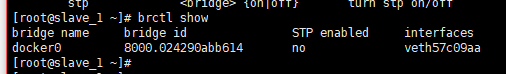
这个模式和前两个不同。在这种模式下，Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息。需要我们自己为Docker容器添加网卡、配置IP等。

### 4 bridge模式

bridge模式是Docker默认的网络设置，此模式会为每一个容器分配Network Namespace、设置IP等，并将一个主机上的Docker容器连接到一个虚拟网桥上。下面着重介绍一下此模式。

## **bridge模式下容器的通信**

brctl show



在bridge模式下，连在同一网桥上的容器可以相互通信（若出于安全考虑，也可以禁止它们之间通信，方法是在DOCKER\_OPTS变量中设置--icc=false，这样只有使用--link才能使两个容器通信）。

容器也可以与外部通信，我们看一下主机上的Iptable规则，可以看到这么一条

-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE

这条规则会将源地址为172.17.0.0/16的包（也就是从Docker容器产生的包），并且不是从docker0网卡发出的，进行源地址转换，转换成主机网卡的地址。这么说可能不太好理解，举一个例子说明一下。假设主机有一块网卡为eth0，IP地址为10.10.101.105/24，网关为10.10.101.254。从主机上一个IP为172.17.0.1/16的容器中ping百度（180.76.3.151）。IP包首先从容器发往自己的默认网关docker0，包到达docker0后，也就到达了主机上。然后会查询主机的路由表，发现包应该从主机的eth0发往主机的网关10.10.105.254/24。接着包会转发给eth0，并从eth0发出去（主机的ip\_forward转发应该已经打开）。这时候，上面的Iptable规则就会起作用，对包做SNAT转换，将源地址换为eth0的地址。这样，在外界看来，这个包就是从10.10.101.105上发出来的，Docker容器对外是不可见的。

那么，外面的机器是如何访问Docker容器的服务呢？我们首先用下面命令创建一个含有web应用的容器，将容器的80端口映射到主机的80端口。

docker run -d --name web -p 80:80 fmzhen/simpleweb

然后查看Iptable规则的变化，发现多了这样一条规则：

-A DOCKER ! -i docker0 -p tcp -m tcp --dport 80 -j DNAT --to-destination 172.17.0.5:80

此条规则就是对主机eth0收到的目的端口为80的tcp流量进行DNAT转换，将流量发往172.17.0.5:80，也就是我们上面创建的Docker容器。所以，外界只需访问10.10.101.105:80就可以访问到容器中得服务。

除此之外，我们还可以自定义Docker使用的IP地址、DNS等信息，甚至使用自己定义的网桥，但是其工作方式还是一样的。