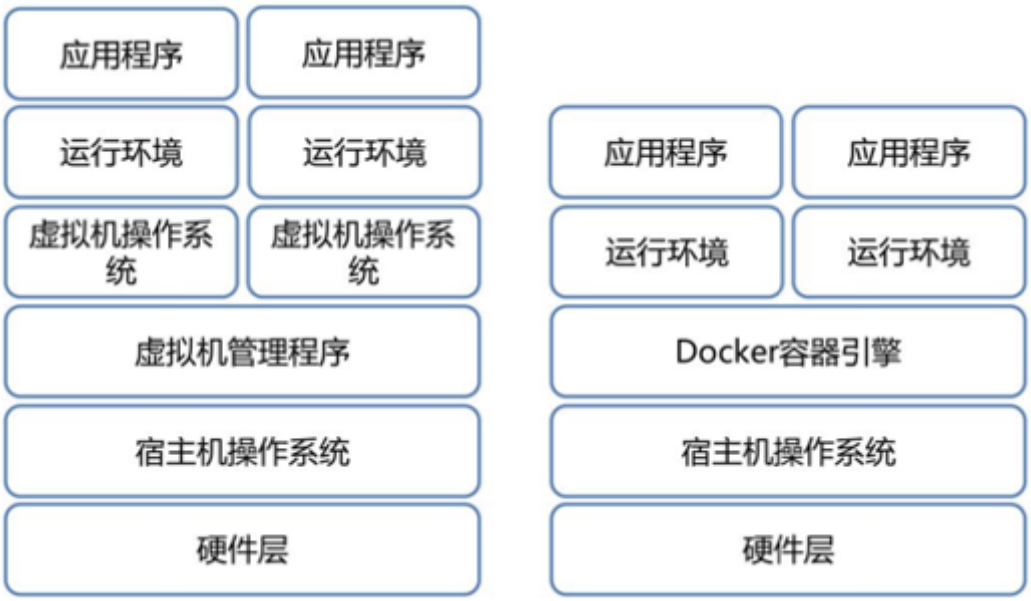
第1章 Docker容器介绍

1.Docker是什么

```
Docker是Docker.Inc 公司开源的一个基于LXC技术之上构建的Container容器引擎,源代码托管在GitHub 上,基于Go语言并遵从Apache2.0协议开源。
Docker是通过内核虚拟化技术（namespaces及cgroups等）来提供容器的资源隔离与安全保障等。
由于Docker通过操作系统层的虚拟化实现隔离，所以Docker容器在运行时，不需要类似虚拟机（VM）额外的操作系统开销，提高资源利用率。
```

2.容器与虚拟机对比

传统虚拟化和Docker分层对比:



VM虚拟化和Docker特性对比

特性	Docker	VM
启动速度	秒级	分钟级
硬盘使用	一般为MB	一般为GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个
隔离性	安全隔离	完全隔离

3.namespace和cgroup

namespace资源隔离

kernel提供了namespace的机制用来隔离相关资源。namespace设计之初就是为了实现轻量级的系统资源隔离。

可以让容器中的进程仿佛置身于一个独立的系统环境中。

namespace	系统调用参数	隔离内容
UTC	`CLONE_NEWUTS`	主机名和域名
IPC	`CLONE_NEWIPC`	信号量、消息队列和共享内存
PID	`CLONE_NEWPID`	进程编号
Network	`CLONE_NEWNET`	网络设备、网络栈、端口等
Mount	`CLONE_NEWNS`	文件系统
User	`CLONE_NEWUSER`	用户和用户组

cgroups资源限制

```
cgroup的作用主要是用来控制资源的使用，比如限制CPU内存和磁盘的使用等
```

cgroups的四大作用：

```
资源限制： 比如设定任务内存使用的上限。
优先级分配： 比如给任务分配CPU的时间片数量和磁盘IO的带宽大小来控制任务运行的优先级。
资源统计：比如统计CPU的使用时长、内存用量等。这个功能非常适用于计费。
任务控制：cgroups可以对任务执行挂起、恢复等操作。
```

4.docker的三个重要概念

Image(镜像):

```
那么镜像到底是什么呢？Docker 镜像可以看作是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。
```

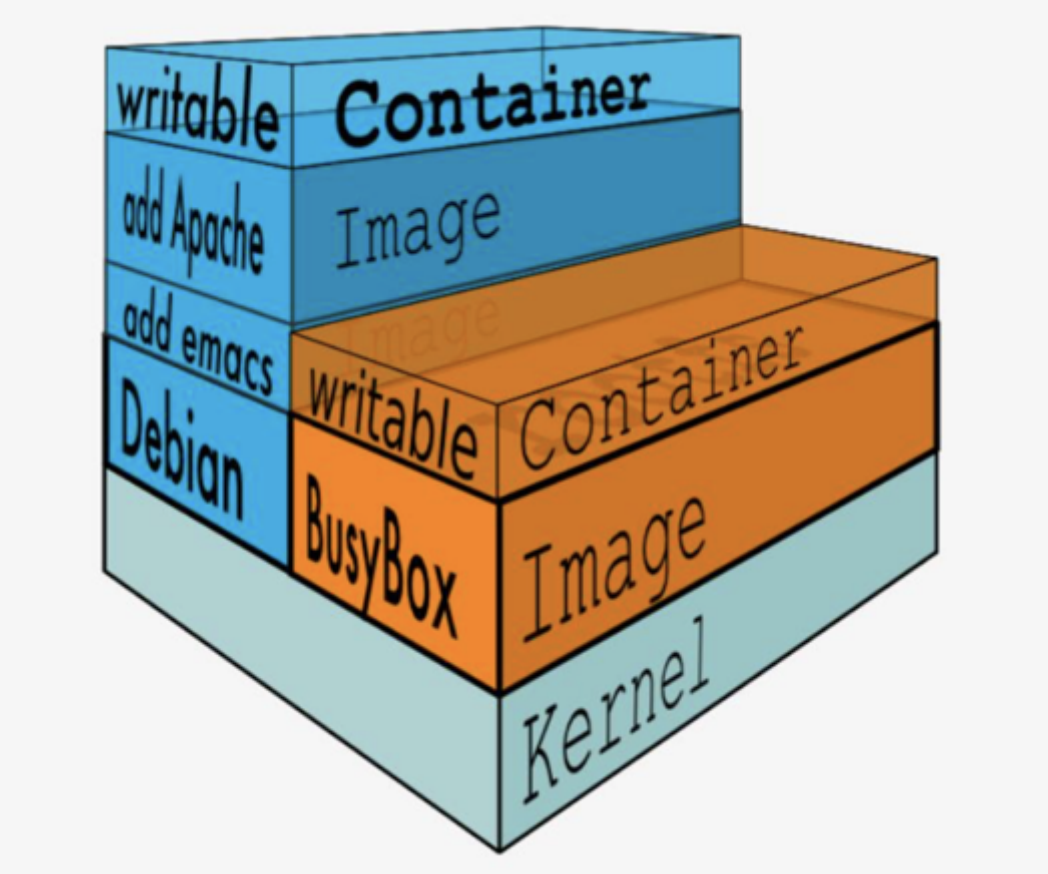
Container(容器):

```
容器(Container)的定义和镜像(Image)几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。
```

Repository(仓库):

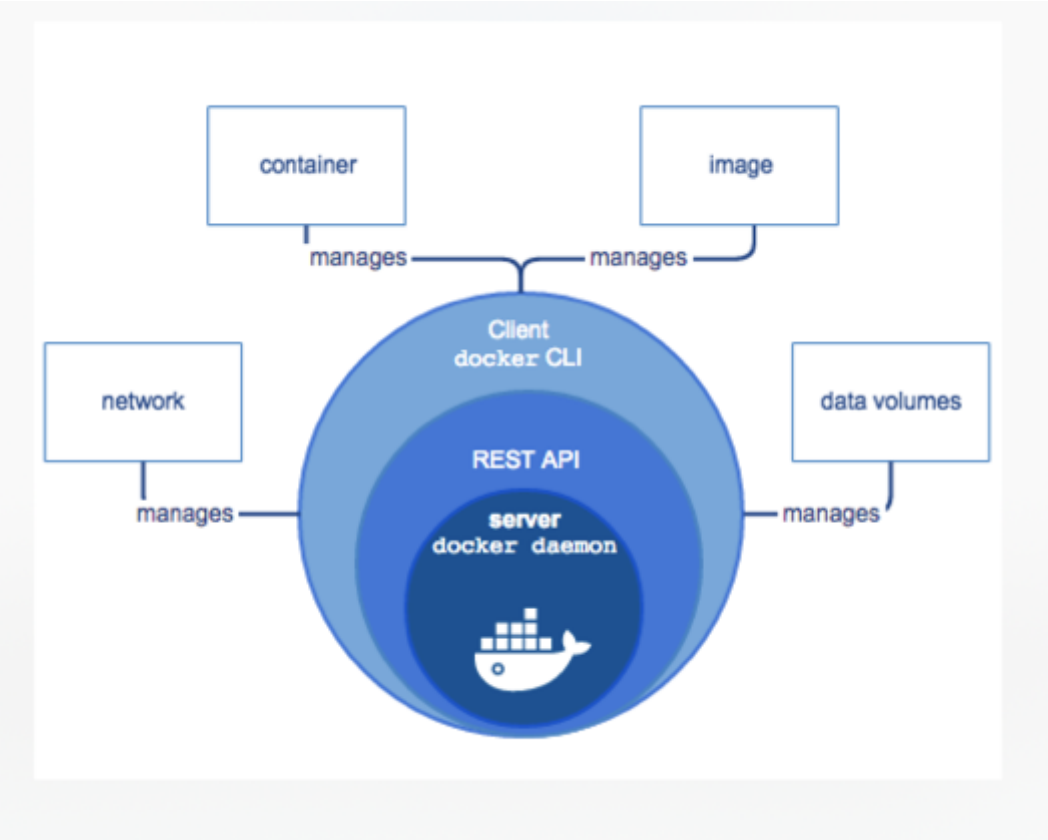
```
镜像仓库是 Docker 用来集中存放镜像文件的地方，类似于我们之前常用的代码仓库。
通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本 。
我们可以通过<仓库名>:<标签>的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 Latest 作为默认标签。
```

镜像和容器图解:



5.docker的组成部分

```
Docker是传统的CS架构分为docker client和docker server
Docker 客户端是 Docker 用户与 Docker 交互的主要方式。
当您使用 Docker 命令行运行命令时，Docker 客户端将这些命令发送给服务器端，服务端将执行这些命令。
Docker 命令使用 Docker API 。
Docker 客户端可以与多个服务端进行通信。
```



第2章 Docker安装部署

1.国内源安装docker-ce

这里我们使用清华源：

```
https://mirrors.tuna.tsinghua.edu.cn/help/docker-ce/
```

操作步骤：

```
yum remove docker docker-common docker-selinux docker-engine
yum install -y yum-utils device-mapper-persistent-data lvm2
wget -O /etc/yum.repos.d/docker-ce.repo
https://download.docker.com/linux/centos/docker-ce.repo
sed -i 's+download.docker.com+mirrors.tuna.tsinghua.edu.cn/docker-ce+'
/etc/yum.repos.d/docker-ce.repo
yum makecache fast
yum install docker-ce
systemctl start docker
```

2.国内远镜像加速配置

加速地址：

```
https://cr.console.aliyun.com/cn-hangzhou/instances/mirrors
```

配置命令：

```
mkdir -p /etc/docker
tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://ig2l319y.mirror.aliyuncs.com"]
}
EOF
systemctl daemon-reload
systemctl restart docker
```

3.运行第一个容器

运行一个Hello world

```
docker run alpine /bin/echo "Hello world"
```

第3章 Docker镜像和容器管理

1.镜像相关命令

1.1 搜索镜像

选择镜像建议：

```
1.优先选择官方的
2.选择星星多的
```

搜索命令：

```
docker search centos
```

使用curl命令获取镜像版本号：

```
yum install jq
curl -s https://registry.hub.docker.com/v1/repositories/centos/tags|jq
```

1.2 获取镜像

```
docker pull centos
docker pull busybox
docker pull busybox:1.29
```

1.3 查看镜像

```
docker images
```

1.4 删除镜像

```
docker rmi centos
```

1.5 导出镜像

```
docker save -o centos.tar centos:latest
```

1.6 导入镜像

```
docker load  < centos.tar
```

2.容器相关命令

2.1 启动容器

```
docker run -d -p 80:80 nginx
docker run --name mydocker -t -i centos /bin/bash

#参数解释
run  运行容器
--name  指定容器的名称，但是不能和已经存在的重复
-d   后台运行容器，并返回容器ID
-p 80:80  端口映射 宿主机端口:容器端口
-i   以交互模式运行容器，通常与 -t 同时使用;
-t   为容器重新分配一个伪输入终端，通常与 -i 同时使用
```

2.2 停止容器

```
docker stop
```

2.3 查看容器

```
docker ps
docker ps -a
docker ps -q
docker ps -aq
docker stats 容器ID
docker rm $(docker ps -q -f 'STATUS=exited')
```

2.4 进入容器

```
docker exec 会分配一个新的终端tty
docker exec -it 容器ID  /bin/bash
docker attach会使用同一个终端
docker attach 容器ID

#参数解释
exec 在运行的容器中执行命令
-i    保持终端打开
-t    分配一个终端
```

2.5 删除容器

删除单个容器

```
docker rm 容器ID
```

批量删除容器

```
docker stop $(docker ps -q)
docker rm $(docker ps -aq)
```

第4章 Docker网络管理

1.随机映射端口

```
docker run -P
```

2.指定映射端口

-p 宿主机端口:容器端口

```
-p 80:80 -p 443:443
-p 宿主机IP:宿主机端口:容器端口
```

如果想多个容器使用8080端口，可以通过添加多个IP地址实现

```
ifconfig eth0:1 10.0.0.13 up
docker run -d -p 10.0.0.11:8080:80 nginx:latest
docker run -d -p 10.0.0.13:8080:80 nginx:latest
```

进入容器里修改站点目录，然后访问测试

```
docker exec -it bdb2a4e7e24d /bin/bash
echo "web01" > /usr/share/nginx/html/index.html
docker exec -it 31c1de138dda /bin/bash
echo "web02" > /usr/share/nginx/html/index.html
```

访问测试:

```
[root@docker-11 ~]# curl 10.0.0.11:8080
web02
[root@docker-11 ~]# curl 10.0.0.13:8080
web01
```

第5章 Docker数据目录管理

1.映射容器目录

```
-v 宿主机目录:容器内目录
```

1.1 创建游戏代码目录

```
mkdir /data/xiaoniao -p
cd /data/
unzip xiaoniaofeifei.zip  -d xiaoniao /
```

1.2 创建容器并映射数据目录

```
docker run -d  -p 80:80 -v /data/xiaoniao:/usr/share/nginx/html nginx:latest
docker ps
```

1.3 访问游戏页面

```
10.0.0.11
```

2.实验-访问不同端口展示不同页面

需求:

```
访问8080端口，展示xiaoniao首页
访问8090端口，展示shenjingmao的首页
```

2.1 准备nginx配置文件

```
[root@docker-11 ~]# cat /data/game.conf
server {
    listen       8080;
    server_name  localhost;
    location / {
        root   /opt/game/;
        index  index.html index.htm;
    }
}

server {
    listen       8090;
    server_name  localhost;
    location / {
        root   /opt/xiaoniao/;
```

```
        index  index.html index.htm;
    }
}
```

2.2 上传代码目录

```
[root@docker-11 /data]# ll
总用量 18896
drwxr-xr-x 5 root root       73 9月   7 23:03 game
-rw-r--r-- 1 root root      309 9月   7 22:57 game.conf
-rw-r--r-- 1 root root 19248295 8月  28 09:48 html5.zip
drwxr-xr-x 3 root root       92 9月   7 22:15 xiaoniao
-rw-r--r-- 1 root root    91014 9月   7 22:11 xiaoniaofeifei.zip
```

2.3 创建容器并挂载

需要挂载的内容：

```
1.nginx配置文件
2.游戏目录
```

创建容器命令:

```
docker run
-p 8080:8080 \
-p 8090:8090 \
-v /data/game:/opt/game \
-v /data/xiaoniao:/opt/xiaoniao \
-v /data/game.conf:/etc/nginx/conf.d/game.conf \
-d nginx:latest
```

2.4 访问测试

```
10.0.0.11:8080
10.0.0.11:8090
```

第6章 Docker镜像手动构建

1.手动制作游戏镜像

```
下面我们基于centos容器制作一个新镜像，并安装nginx服务
```

1.1 启动一个容器并安装nginx

```
[root@docker-11 ~]# docker run -it centos /bin/bash
[root@0ede2760ba65 /]# yum install wget install openssh-clients -y
[root@0ede2760ba65 /]# wget -O /etc/yum.repos.d/CentOS-Base.repo
http://mirrors.aliyun.com/repo/Centos-7.repo
[root@0ede2760ba65 /]# wget -O /etc/yum.repos.d/epel.repo
http://mirrors.aliyun.com/repo/epel-7.repo
```

```
[root@0ede2760ba65 /]# sed -i -e '/mirrors.cloud.aliyuncs.com/d' -e '/mirrors.aliyuncs.com/d' /etc/yum.repos.d/CentOS-Base.repo
[root@0ede2760ba65 /]# cat /etc/yum.repos.d/nginx.repo
[nginx-stable]
name=nginx stable repo
baseurl=http://nginx.org/packages/centos/$releasever/$basearch/
gpgcheck=1
enabled=1
gpgkey=https://nginx.org/keys/nginx_signing.key
[nginx-mainline]
name=nginx mainline repo
baseurl=http://nginx.org/packages/mainline/centos/$releasever/$basearch/
gpgcheck=1
enabled=0
gpgkey=https://nginx.org/keys/nginx_signing.key
[root@0ede2760ba65 /]# yum makecache fast
[root@0ede2760ba65 /]# yum install nginx -y
```

1.2 上传代码目录并配置nginx配置文件

```
[root@0ede2760ba65 /]# scp -r 10.0.0.11:/data/* /opt/
[root@0ede2760ba65 /]# ll /opt/
total 18896
drwxr-xr-x 5 root root       73 Sep  7 16:02 game
-rw-r--r-- 1 root root      303 Sep  7 16:02 game.conf
-rw-r--r-- 1 root root 19248295 Sep  7 16:02 html5.zip
drwxr-xr-x 3 root root       92 Sep  7 16:02 xiaoniao
-rw-r--r-- 1 root root    91014 Sep  7 16:02 xiaoniaofeifei.zip
[root@0ede2760ba65 /]# cp /opt/game.conf /etc/nginx/conf.d/
```

1.3 将容器提交为新的镜像

```
[root@docker-11 ~]# docker ps -aq
0ede2760ba65
[root@docker-11 ~]# docker commit 0ede2760ba65 game:v1
sha256:a61d28fbfe27ebe36d4b73825b55e5f94097083273ab56dccce0453ce2bd6d38
```

1.4 测试镜像功能是否可用

```
[root@docker-11 ~]# docker run -d -p 8080:8080 -p 8090:8090 game:v1  nginx -g 'daemon off;'
f58f209d4761c4bdd9bb164c0050a94a3273b1ee0e57eafe29e48b1517c72950
```

1.5 将新镜像导出

```
docker save -o game_v1.tar game:v1
```

2.手动制作云盘镜像

2.1 创建容器

```
docker run -d -p 80:80 --name clould game:v1  nginx -g 'daemon off;'
```

2.2 进入容器安装php并求改运行用户

```
[root@d0c987bcefa2 /]# yum install php-fpm -y
[root@d0c987bcefa2 /]# php-fpm -v
PHP 5.4.16 (fpm-fcgi) (built: Oct 30 2018 19:32:20)
Copyright (c) 1997-2013 The PHP Group
Zend Engine v2.4.0, Copyright (c) 1998-2013 Zend Technologies
[root@d0c987bcefa2 /]# sed -i '/^user/c user = nginx' /etc/php-fpm.d/www.conf
[root@d0c987bcefa2 /]# sed -i '/^group/c group = nginx' /etc/php-
fpm.d/www.conf
[root@d0c987bcefa2 /]# sed -i '/daemonize/s#no#yes#g' /etc/php-fpm.conf
[root@d0c987bcefa2 /]# php-fpm -c /etc/php.ini -y /etc/php-fpm.conf
[root@d0c987bcefa2 /]# php-fpm -c /etc/php.ini -y /etc/php-fpm.conf

[root@d0c987bcefa2 /]# ps -ef|grep php
root         77      0  0 21:43 ?        00:00:00 php-fpm: master process
(/etc/php-fpm.conf)
nginx        78     77  0 21:43 ?        00:00:00 php-fpm: pool www
nginx        79     77  0 21:43 ?        00:00:00 php-fpm: pool www
nginx        80     77  0 21:43 ?        00:00:00 php-fpm: pool www
nginx        81     77  0 21:43 ?        00:00:00 php-fpm: pool www
nginx        82     77  0 21:43 ?        00:00:00 php-fpm: pool www
```

2.3 配置nginx

```
[root@d0c987bcefa2 /]# cat /etc/nginx/conf.d/cloud.conf
server {
    listen 80;
    server_name localhost;
    root /code;
    index index.php index.html;

    location ~ \.php$ {
        root /code;
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
[root@d0c987bcefa2 /]# nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
[root@d0c987bcefa2 /]# nginx -s reload
```

2.4 下载代码目录

```
[root@d0c987bcefa2 /]# mkdir /code
[root@d0c987bcefa2 /]# cd /code/
[root@d0c987bcefa2 code]# scp -r 10.0.0.11:/data/kod/* /code/
[root@d0c987bcefa2 code]# ls
ChangeLog.md  README.MD  app  config  data  index.php  plugins  static
[root@d0c987bcefa2 code]# chown -R nginx:nginx /code/
```

2.5 测试

```
[root@d0c987bcefa2 code]# curl -I 127.0.0.1
HTTP/1.1 302 Moved Temporarily
Server: nginx/1.16.1
Date: Sat, 07 Sep 2019 21:53:17 GMT
Content-Type: text/html; charset=utf-8
Connection: keep-alive
X-Powered-By: PHP/5.4.16
Set-Cookie: KOD_SESSION_ID_9d6d9=ljq63o0tmcscon6eb3gdpqscf4; path=/
Set-Cookie: KOD_SESSION_ID_9d6d9=ljq63o0tmcscon6eb3gdpqscf4; path=/
Set-Cookie: KOD_SESSION_ID_9d6d9=ljq63o0tmcscon6eb3gdpqscf4; path=/
Set-Cookie: KOD_SESSION_SSO=bboh1p0h1uc50tfibrg67dnra7; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: KOD_SESSION_ID_9d6d9=deleted; expires=Thu, 01-Jan-1970 00:00:01
GMT; path=/
Set-Cookie: kod_name=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT
Set-Cookie: kodToken=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT
Set-Cookie: X-CSRF-TOKEN=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT
Location: ./index.php?user/login
```

2.6 提交新的镜像

```
[root@docker-11 ~]# docker commit d0c987bcefa2 kod:v1
sha256:169df6e8db11bd044e3e05237f2947783f9cc7a65b643dc9206ecf05fdc4a3ea
```

2.7 编写启动脚本并提交新镜像

```
[root@docker-11 ~]# docker exec -it c14835183fb5 /bin/bash
[root@c14835183fb5 /]# cat init.sh
#!/bin/bash
php-fpm -c /etc/php.ini -y /etc/php-fpm.conf
nginx -g 'daemon off;'
[root@c14835183fb5 /]# chmod +x init.sh
[root@docker-11 ~]# docker commit c14835183fb5 kod:v2
sha256:c05ebdf400aa7f7a27aa857df0d9c75c42943db89abca66f79101771db8e9585
```

2.8 启动测试

```
[root@docker-11 ~]# docker stop $(docker ps -qa)
[root@docker-11 ~]# docker run -d -p 80:80 kod:v2 /bin/bash /init.sh
dccf4aea5471713872e4fefaca45f7fac3bffec8f5f602570863ed14231dea1a
[root@docker-11 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
      STATUS              PORTS
NAMES
dccf4aea5471        kod:v2              "/bin/bash /init.sh"     36 seconds
ago      Up 35 seconds       0.0.0.0:80->80/tcp
magical_napier
```

2.9 添加GD库

```
此时打开页面提示缺少GD库，进入容器内安装php支持插件
[root@dccf4aea5471 /]yum install php-mbstring php-gd -y
然后重启容器
[root@docker-11 ~]# docker restart dccf4aea5471
```

2.10 访问测试没问题后提交新镜像

```
[root@docker-11 ~]# docker commit dccf4aea5471 kod:v2
sha256:23051ce545a2eb6bb50bb2307bd9cfbaf6139e52f205a4126fb1d8d974c417f4
```

第7章 Dockerfile自动构建Docker镜像

1.Dockerfile介绍

```
通过前面的练习我们已经掌握了手动制作镜像的方法，但是这种方法命令繁多，而且不能自动化，操作起来比较复杂。
实际上在企业中我们会使用Dockerfile来自动化构建镜像。
Dockerfile是一种可以被docker解释并执行的脚本，拥有自己固定的指令。
有了这个利器之后我们运维就可以解放双手了，只要编写好脚本就可以按照我们期望的结果构建相应的镜像。
如果需要更新，只需要修改Dockerfile里很少的代码就可以重复的构建。
```

2.Dockerfile操作命令说明

```
Docker通过对于在Dockerfile中的一系列指令的顺序解析实现自动的image的构建
    通过使用build命令，根据Dockerfiel的描述来构建镜像
    通过源代码路径的方式
    通过标准输入流的方式
Dockerfile指令:
    只支持Docker自己定义的一套指令，不支持自定义
    大小写不敏感，但是建议全部使用大写
    根据Dockerfile的内容顺序执行
FROM:
    FROM {base镜像}
    必须放在DOckerfile的第一行，表示从哪个baseimage开始构建
MAINTAINER:
    可选的，用来标识image作者的地方
```

RUN:
    每一个RUN指令都会是在一个新的container里面运行，并提交为一个image作为下一个RUN的base
    一个Dockerfile中可以包含多个RUN，按定义顺序执行
    RUN支持两种运行方式：
        RUN <cmd> 这个会当作/bin/sh -c “cmd” 运行
        RUN [“executable”，“arg1”，。。。]，Docker把他当作json的顺序来解析，因此必须使用双引号，而且executable需要是完整路径
    RUN 都是启动一个容器、执行命令、然后提交存储层文件变更。第一层 RUN command1 的执行仅仅是当前进程，一个内存上的变化而已，其结果不会造成任何文件。而到第二层的时候，启动的是一个全新的容器，跟第一层的容器更完全没关系，自然不可能继承前一层构建过程中的内存变化。而如果需要将两条命令或者多条命令联合起来执行需要加上&&。如：cd /usr/local/src && wget xxxxxxx
CMD:
    CMD的作用是作为执行container时候的默认行为（容器默认的启动命令）
    当运行container的时候声明了command，则不再用image中的CMD默认所定义的命令
    一个Dockerfile中只能有一个有效的CMD，当定义多个CMD的时候，只有最后一个才会起作用
CMD定义的三种方式：
    CMD <cmd> 这个会当作/bin/sh -c "cmd"来执行
    CMD ["executable","arg1",....]
    CMD ["arg1","arg2"]，这个时候CMD作为ENTRYPOINT的参数
EXPOSE 声明端口
    格式为 EXPOSE <端口1> [<端口2>...]。
    EXPOSE 指令是声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是 docker run -P 时，会自动随机映射 EXPOSE 的端口。
ENTRYPOINT:
    entrypoint的作用是，把整个container变成了一个可执行的文件，这样不能够通过替换CMD的方法来改变创建container的方式。但是可以通过参数传递的方法影响到container内部
    每个Dockerfile只能够包含一个entrypoint，多个entrypoint只有最后一个有效
    当定义了entrypoint以后，CMD只能够作为参数进行传递
entrypoint定义方式：
    entrypoint ["executable","arg1","arg2"]，这种定义方式下，CMD可以通过json的方式来定义entrypoint的参数，可以通过在运行container的时候通过指定command的方式传递参数
    entrypoint <cmd>，当作/bin/bash -c "cmd"运行命令
ADD & COPY:
    当在源代码构建的方式下，可以通过ADD和COPY的方式，把host上的文件或者目录复制到image中
    ADD和COPY的源必须在context路径下
    当src为网络URL的情况下，ADD指令可以把它下载到dest的指定位置，这个在任何build的方式下都可以work
    ADD相对COPY还有一个多的功能，能够进行自动解压压缩包
ENV:
    ENV key value
    用来设置环境变量，后续的RUN可以使用它所创建的环境变量
    当创建基于该镜像的container的时候，会自动拥有设置的环境变量
WORKDIR:
    用来指定当前工作目录（或者称为当前目录）
    当使用相对目录的情况下，采用上一个WORKDIR指定的目录作为基准
USER:
    指定UID或者username，来决定运行RUN指令的用户
ONBUILD:
    ONBUILD作为一个trigger的标记，可以用来trigger任何Dockerfile中的指令
    可以定义多个ONBUILD指令
    当下一个镜像B使用镜像A作为base的时候，在FROM A指令前，会先按照顺序执行在构建A时候定义的ONBUILD指令
    ONBUILD <DOCKERFILE 指令> <content>

```
VOLUME:
    用来创建一个在image之外的mount point，用来在多个container之间实现数据共享
    运行使用json array的方式定义多个volume
    VOLUME ["/var/data1","/var/data2"]
    或者plain text的情况下定义多个VOLUME指令
```

4.Dockerfile小试身手

构建思路：

```
1.先不要着急写Dockerfile，首先手动进入容器，然后正常执行安装步骤。并确保运行正常
2.收集好安装步骤的命令以及正确的配置文件
3.将收集的配置文件按照一定规范保存在相应的目录下
4.根据收集到的安装步骤编写Dockerfile
5.构建镜像并启动测试
```

小项目：使用Dockerfile构建Centos7+Nginx镜像

```
#1.创建目录
[root@docker-11 ~]# mkdir dockerfile
[root@docker-11 ~]# cd dockerfile/
[root@docker-11 ~/dockerfile]# mkdir nginx_base
[root@docker-11 ~/dockerfile]# cd nginx_base/

#2.准备文件
[root@docker-11 ~/dockerfile/nginx_base]# cat > local.repo << 'EOF'
[local]
name=local
enable=1
gpgcheck=0
baseurl=http://10.0.0.100
EOF
[root@docker-11 ~/dockerfile/nginx_base]# ll
total 8
-rw-r--r-- 1 root root 292 Jul 22 15:01 Dockerfile
-rw-r--r-- 1 root root  65 Jul 22 15:47 local.repo

#3.编写Dockerfile
[root@docker-11 ~/dockerfile/nginx_base]# cat > Dockerfile << 'EOF'
FROM centos:7
RUN rm -rf /etc/yum.repos.d/*
ADD local.repo /etc/yum.repos.d/local.repo
RUN yum makecache fast \
    yum install nginx -y \
      yum clean all
EXPOSE 80
CMD ["nginx","-g","daemon off;"]
EOF

#4.构建镜像
[root@docker-11 ~/dockerfile/nginx_base]# cat build.sh
#!/bin/bash
docker build -t centos7_nginx:1.20 .
[root@docker-11 ~/dockerfile/nginx_base]# bash build.sh

#5.启动测试
```

```
[root@docker-11 ~/dockerfile/nginx_base]# docker run -d -p 10.0.0.11:80:80 centos7_nginx:1.20
[root@docker-11 ~/dockerfile/nginx_base]# curl -I 10.0.0.11
```

5.Dockerfile分段构建的最佳实践

避免安装不必要的软件包

```
只安装容器必须的软件，不是必须要的不要安装，比如一些工具类的命令，wget net-tools等
```

安装完成后清除缓存

```
安装完成后我们可以把不用的压缩包以及软件缓存等文件删除掉，以减少镜像的体积
```

一个容器内不要跑太多的应用

```
Docker倡导一个容器应该只关注一件事，应该保证一个容器只有一个进程。
但是这也不是硬性要求，比如nginx+php+mysql的应用，那么我们也可以将nginx+php放在一个容器里，mysql单独放一个容器。
但是我们还是希望一个容器专注干一件事，尽量保持干净和模块化。
```

最小化镜像层数

```
Dockerfile里每条RUN，COPY，ADD指令都会创建指令层，所以我们可以将多条RUN命令尽可能的合并成一个RUN指令，这样就减少了构建镜像的层数。
```

增加可读性

```
我们刚才说了可以讲多个RUN指令合并成一个RUN指令，但是这样做可能会导致RUN指令很长很长，那么我们可以像shell命令那样使用\来换行，增加可读性，例如：
RUN yum install git \
    wget \
    net-tools \
    tree
```

使用supervisor控制多个进程

```
#1.安装软件
yum -y install supervisor

#2.编写进程配置文件
[root@b7757c17bf36 ~]# cat /etc/supervisord.d/nginx_php.ini
[program:nginx]
command=nginx -g 'daemon off;'
autostart=true
autorestart=true
startsecs = 5
redirect_stderr = true
stdout_logfile_maxbytes = 20MB
stdout_logfile_backups = 20
stdout_logfile = /var/log/supervisor/nginx.log
```

```

[program:php-fpm]
command=php-fpm -c /etc/php.ini -y /etc/php-fpm.conf
autostart=true
autorestart=true
startsecs = 5
redirect_stderr = true
stdout_logfile_maxbytes = 20MB
stdout_logfile_backups = 20
stdout_logfile = /var/log/supervisor/php-fpm.log

#3.修改supervisord配置文件放在前台启动
sed -i "s#nodaemon=false#nodaemon=true#g" /etc/supervisord.conf

#4.启动supervisord程序
supervisord -c /etc/supervisord.conf

#5.使用命令
supervisorctl update
supervisorctl status
supervisorctl start nginx
supervisorctl restart nginx
supervisorctl stop nginx
```

6.使用Dockerfile创建云盘镜像

```
基于Dockerfile构建云盘镜像

1.先创建目录
[root@docker-11 ~]# cd dockerfile/
[root@docker-11 ~/dockerfile]# ls
nginx_base
[root@docker-11 ~/dockerfile]# mkdir kod
[root@docker-11 ~/dockerfile]# cd kod/
[root@docker-11 ~/dockerfile/kod]#

2.收集配置文件
mkdir conf
cd conf
cat > local.repo << EOF
[local]
name=local
enable=1
gpgcheck=0
baseurl=http://10.0.0.100
EOF
docker cp b7757c17bf36:/etc/php-fpm.d/www.conf  .
docker cp b7757c17bf36:/etc/nginx/conf.d/cloud.conf  .
docker cp b7757c17bf36:/etc/supervisord.conf  .
docker cp b7757c17bf36:/etc/supervisord.d/nginx_php.ini  .

3.准备代码目录
mkdir code/
cd code
docker cp b7757c17bf36:/code/ .
tar zcvf code.tar.gz code

```

```
4.编写Dockerfile
FROM centos:7
RUN rm -rf /etc/yum.repos.d/*
ADD conf/local.repo /etc/yum.repos.d/local.repo
RUN yum install nginx php-fpm php-mbstring php-gd supervisor -y
ADD conf/www.conf /etc/php-fpm.d/www.conf
ADD conf/cloud.conf /etc/nginx/conf.d/cloud.conf
ADD conf/supervisord.conf /etc/supervisord.conf
ADD conf/nginx_php.ini /etc/supervisord.d/nginx_php.ini
ADD code/code.tar.gz /
RUN chown -R nginx:nginx /code/
EXPOSE 80
CMD ["supervisord","-c","/etc/supervisord.conf"]

5.构建镜像
docker build -t kod:v2 .

6.启动容器测试
docker run -d --name kod_v1 -p 8080:80 kod:v1
```

7.使用Dockerfile创建KVM图形化管理工具镜像

```

```

8.构建Tomcat镜像

8.1 构建基础CentOS7镜像

```
#1.基础镜像需要的操作
安装网络工具包
配置yum源
更改时区

#2.创建目录
mkdir centos7
cd centos7/

#3.准备配置文件
[root@docker-11 centos7]# ll
总用量 16
-rw-r--r-- 1 root root 1759 7月  22 21:02 CentOS-Base.repo
-rw-r--r-- 1 root root  664 7月  22 21:02 epel.repo

#4.编写dockerfile
FROM centos:7
RUN rm -f /etc/yum.repos.d/*
ADD CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo
ADD epel.repo /etc/yum.repos.d/epel.repo
ADD supervisord.conf /etc/supervisord.conf
RUN yum install net-tools bash-completion supervisor -y \
    && yum clean all \
    && rm -f /etc/localtime \
    && ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime \
    && groupadd -g 1000 www \
    && useradd -u 1000 -g 1000 www -M -s /sbin/nologin
```

```

#5.构建命令
docker build -t centos7_base:v1 .
```

8.2 构建基础JDK镜像

```
#1.创建目录
mkdir dockerfile/jdk/

#2.准备配置文件
[root@docker-11 ~]# cd dockerfile/jdk/
[root@docker-11 jdk]# docker cp 28e7ff621f8c:/etc/profile .
[root@docker-11 jdk]# vim profile
[root@docker-11 jdk]# tail -4 dockerfile/jdk/profile
export JAVA_HOME=/opt/jdk
export TOMCAT_HOME=/opt/tomcat
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$TOMCAT_HOME/bin:$PATH
export
CLASSPATH=.$CLASSPATH:$JAVA_HOME/lib:$JAVA_HOME/jre/lib:$JAVA_HOME/lib/tools.
jar

#3.将jdk上传到指定目录
[root@docker-11 ~]# tree dockerfile/
dockerfile/
└── jdk
    ├── jdk-8u60-linux-x64.tar.gz
    └── profile

#4.编写Dockerfile
[root@docker-11 jdk]# cat Dockerfile
FROM centos7_base:v1
ADD jdk-8u60-linux-x64.tar.gz /opt/
ADD profile /etc/profile
RUN ln -s /opt/jdk1.8.0_60 /opt/jdk
ENV JAVA_HOME /opt/jdk
ENV JRE_HOME $JAVA_HOME/jre
ENV CLASSPATH $JAVA_HOME/lib/:$JRE_HOME/lib/
ENV PATH $PATH:$JAVA_HOME/bin

#5.编写构建命令脚本
[root@docker-11 jdk]#  cat > /root/dockerfile/jdk/build.sh << 'EOF'
#!/bin/bash
docker build -t centos7_jdk:8u60 .
EOF
[root@docker-11 jdk]# bash build.sh

#6.查看构建后的镜像
[root@docker-11 jdk]# docker images|grep jdk
centos7_jdk   8u60      58a880ff9253   19 seconds ago   569MB

#7.使用jdk镜像启动容器测试
[root@docker-11 jdk]# docker run -it --rm centos7_jdk:8u60 java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

8.3 构建Tomcat镜像

```
#1.创建目录
[root@docker-11 ~]# mkdir dockerfile/tomcat

#2.上传压缩包
[root@docker-11 ~]# cd dockerfile/tomcat
[root@docker-11 tomcat]# wget
https://mirrors.tuna.tsinghua.edu.cn/apache/tomcat/tomcat-
8/v8.5.69/bin/apache-tomcat-8.5.69.tar.gz

#3.编写Dockerfile
[root@docker-11 tomcat]# cat > Dockerfile << 'EOF'
FROM centos7_jdk:8u60
ADD apache-tomcat-8.5.69.tar.gz /opt/
RUN ln -s /opt/apache-tomcat-8.5.69 /opt/tomcat
EOF

#4.编写构建命令脚本
[root@docker-11 jdk]#  cat > /root/dockerfile/jdk/build.sh << 'EOF'
#!/bin/bash
docker build -t tomcat_base:8.5.69 .
EOF
[root@docker-11 tomcat]# tree
.
├── apache-tomcat-8.5.69.tar.gz
├── build.sh
└── Dockerfile

#5.构建镜像
[root@docker-11 tomcat]# bash build.sh

#6.测试访问
[root@docker-11 tomcat]# docker run -d -it tomcat_base:8.5.69 /bin/bash
[root@docker-11 tomcat]# docker exec -it b2011b0b3eb3 /bin/bash
[root@b2011b0b3eb3 /]# /opt/tomcat/bin/catalina.sh start
```

8.4 构建业务镜像

```
#1.创建目录
[root@docker-11 ~]# mkdir dockerfile/webapp1

#2.编写业务文件
[root@docker-11 ~]# cd dockerfile/webapp1
[root@docker-11 webapp1]# mkdir app
[root@docker-11 webapp1]# echo "V1" > app/index.jsp
[root@docker-11 webapp1]# tar zcf app.tar.gz app/

#3.修改tomcat配置文件
[root@docker-11 webapp1]# docker cp b2011b0b3eb3:/opt/tomcat/conf/server.xml
.
[root@docker-11 webapp1]# vim server.xml
 <Host name="localhost"  appBase="/opt/tomcat/webapps"

#4.编写supervisor配置文件
[root@docker-11 webapp1]# cat tomcat.ini
```

```
[program:tomcat]
command=/opt/tomcat/bin/catalina.sh run
autostart=true
autorestart=true
startsecs = 5
redirect_stderr = true
stdout_logfile_maxbytes = 20MB
stdout_logfile_backups = 20
stdout_logfile = /var/log/supervisor/tomcat.log

#5.编写Dockerfile文件
[root@docker-11 webapp1]# cat Dockerfile
FROM tomcat_base:8.5.69
ADD tomcat.ini /etc/supervisord.d/tomcat.ini
ADD server.xml /opt/tomcat/conf/server.xml
ADD app.tar.gz /opt/tomcat/webapps/
EXPOSE 8080
CMD ["supervisord","-c","/etc/supervisord.conf"]

#6.编写构建脚本
[root@docker-11 webapp1]# cat > build.sh << 'EOF'
#!/bin/bash
docker build -t tomcat_app:v1 .
EOF

#7.运行容器测试
docker run -d -it -p 8080:8080 tomcat_app:v1
```

第8章 企业级私有仓库Docker-harbor

1.部署步骤

```
第一步：安装docker和docker-compose
第二步：下载harbor-offline-installer-v1.9.0-rc1.tgz
第三步：上传到/opt,并解压
第四步：修改harbor.yml配置文件 hostname = 10.0.0.11 harbor_admin_password = 123456
第五步：执行install.sh
```

2.安装docker-compose

1.安装docker-compose

```
yum install -y docker-compose
```

2.检查

```
docker-compose version
```

3.上传解压docker-harbor

```
[root@docker-11 ~]# cd /opt/
[root@docker-11 /opt]# ls
harbor-offline-installer-v1.9.0-rc1.tgz
[root@docker-11 /opt]# tar zxf harbor-offline-installer-v1.9.0-rc1.tgz
[root@docker-11 /opt]# ls
harbor  harbor-offline-installer-v1.9.0-rc1.tgz
[root@docker-11 /opt]# cd harbor/
```

4.修改配置文件

修改2个地方：

```
[root@docker-11 /opt/harbor]# egrep "10.0.0.11|123456" harbor.yml
hostname: 10.0.0.11
harbor_admin_password: 123456
```

5.安装

```
[root@docker-11 /opt/harbor]# ./install.sh
```

6.修改docker信任仓库

```
[root@docker-11 /opt/harbor]# cat /etc/docker/daemon.json
{
      "registry-mirrors": ["http://hub-mirror.c.163.com"],
      "insecure-registries": ["http://10.0.0.11"]
}
```

7.重启docker

```
systemctl restart docker
```

8.给镜像打标签并提交到harbor

docker登陆harbor

```
[root@docker-11 /opt/harbor]# docker login 10.0.0.11
Username: zhangya
Password:
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

运行一个容器：

```
[root@docker-11 ~]# docker run -d -p 8080:80 centos_kod:v1
78be80f7c2029b68e8943e38fa99131ec6709f798e63c94afb5a7fdfa4a8047c
```

查看容器ID:

```
[root@docker-11 ~]# docker ps|grep kod
78be80f7c202        centos_kod:v1
"/bin/bash /init.sh"     15 seconds ago      Up 13 seconds
0.0.0.0:8080->80/tcp        tender_dirac
```

将容器提交为新镜像并且更改为harbor仓库的地址

```
[root@docker-11 ~]# docker commit 78be80f7c202 10.0.0.11/linux/centos_kod:v1
sha256:6bf1e1eef1969bcd4c82472aed945d4dda74a923c0d7dae91e38539676f8c240
```

查看镜像

```
[root@docker-11 ~/dockerfile/kod]# docker images
REPOSITORY                      TAG                         IMAGE ID
CREATED             SIZE
10.0.0.11/linux/centos_kod      v1                          6bf1e1eef196
13 minutes ago      465MB
```

将新镜像推送到harbor上

```
[root@docker-11 /opt/harbor]# docker push 10.0.0.11/linux/centos_kod:v1
```

9.在docker-harbor上查看

```
10.0.0.11
账号:admin
密码:123456
```

10.其他主机上下载镜像

配置docker信任仓库

```
[root@docker-12 ~]# cat /etc/docker/daemon.json
{
      "registry-mirrors": ["http://hub-mirror.c.163.com"],
      "insecure-registries": ["http://10.0.0.11"],
}
```

从Harbor仓库拉取镜像

```
[root@docker-12 ~]# docker pull 10.0.0.11/linux/centos_kod:v1
```

第9章 Docker网络模式

1.Docker网络的四种模式

```
Host            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。
Bridge     此模式会为每一个容器分配、设置IP等，并将容器连接到一个docker0虚拟网桥，通过docker0网桥以及Iptables nat表配置与宿主机通信。
None            此模式关闭了容器的网络功能。
Container    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。
```

查看网络模式命令:

```
[root@node-51 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
3791e4fc9c18        bridge              bridge              local
b494337929ef        host                host                local
a153ac0003e3        none                null                local
```

查看网卡命令:

```
[root@node-51 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:11:6b:18 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.51/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe11:6b18/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:bb:96:63:c7 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```

查看桥接网卡命令

```
yum install bridge-utils -y
brctl show
```

2.Bridge模式

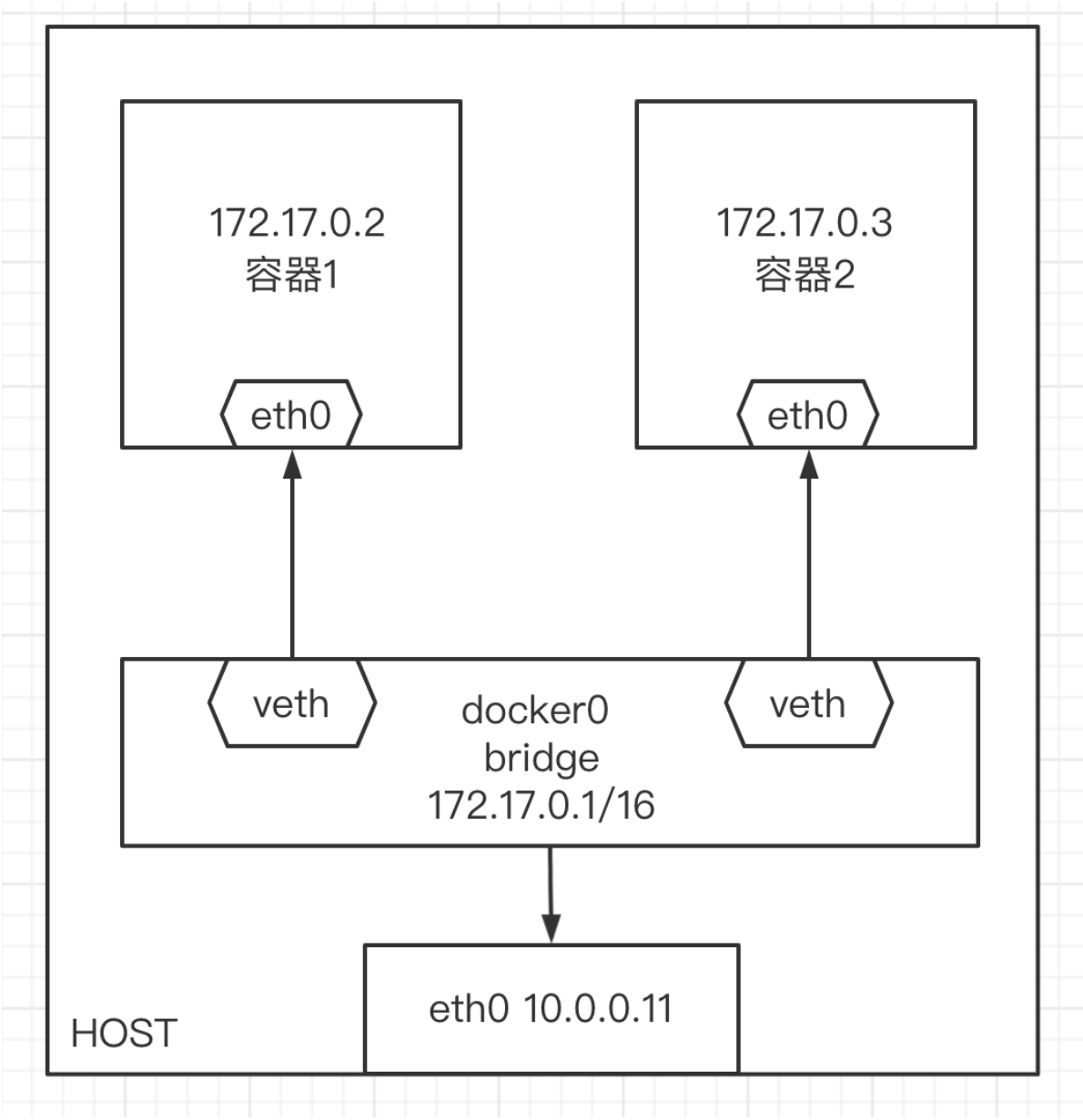
2.1 Bridge模式介绍

```
1.当Docker Daemon第一次启动时会创建一个虚拟的网桥，默认名称是Docker0
2.创建完后会给这个网桥分配一个子网,默认是172.17.0.1/16
3.由Docker创建的每一个容器,都会创建一个一个Veth设备对,其中一端关联到网桥上,另一端放在容器里映射为eth0,然后从网桥的地址段内给容器内的eth0分配一个IP地址,这样容器之间就可以互通了。
```

网络模式特点：

```
1.同一宿主机的容器之间可以互相通信，不同宿主机之间不能互相通信
2.桥接模式的容器可以自动获取172.17.0.0/16网段的IP地址
3.其他机器不能直接访问容器，可以通过映射端口的形式访问
4.每个容器映射到宿主机的端口不能重复
5.容器可以借助宿主机的网络访问其他机器
```

2.2 Bridge模式示意图



2.3 查看Bridge的详细信息

查看桥接模式的详细信息：

```
[root@docker-11 ~]# docker network inspect bridge
```

容器内查看：

```
[root@docker-11 ~]# docker run -it busybox /bin/sh
/ # cd
~ # ip a
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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
14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:05 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.5/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
~ #
~ # route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.17.0.1      0.0.0.0         UG    0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0

~ # ping 10.0.0.12 -c 1
PING 10.0.0.12 (10.0.0.12): 56 data bytes
64 bytes from 10.0.0.12: seq=0 ttl=63 time=0.471 ms

--- 10.0.0.12 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.471/0.471/0.471 ms

~ # traceroute 10.0.0.12
traceroute to 10.0.0.12 (10.0.0.12), 30 hops max, 46 byte packets
 1  172.17.0.1 (172.17.0.1)  0.010 ms  0.005 ms  0.005 ms
 2  10.0.0.12 (10.0.0.12)  0.257 ms  0.246 ms  0.192 ms
```

2.4 修改桥接模式默认的网络配置

方法1：修改systemd文件添加bip参数

```
[root@docker-11 ~]# vim /lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --bip=192.168.1.1/24
[root@docker-11 ~]# systemctl daemon-reload
[root@docker-11 ~]# systemctl restart docker.service
[root@docker-11 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:9a:74:d9 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.11/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe9a:74d9/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:ed:fc:67:1f brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.1/24 brd 192.168.1.255 scope global docker0
```

```
       valid_lft forever preferred_lft forever
    inet6 fe80::42:edff:fefc:671f/64 scope link
       valid_lft forever preferred_lft forever

[root@docker-11 ~]# docker run -it busybox /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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
18: eth0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:42:c0:a8:01:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.2/24 brd 192.168.1.255 scope global eth0
       valid_lft forever preferred_lft forever
```

方法2：修改daemon.json文件

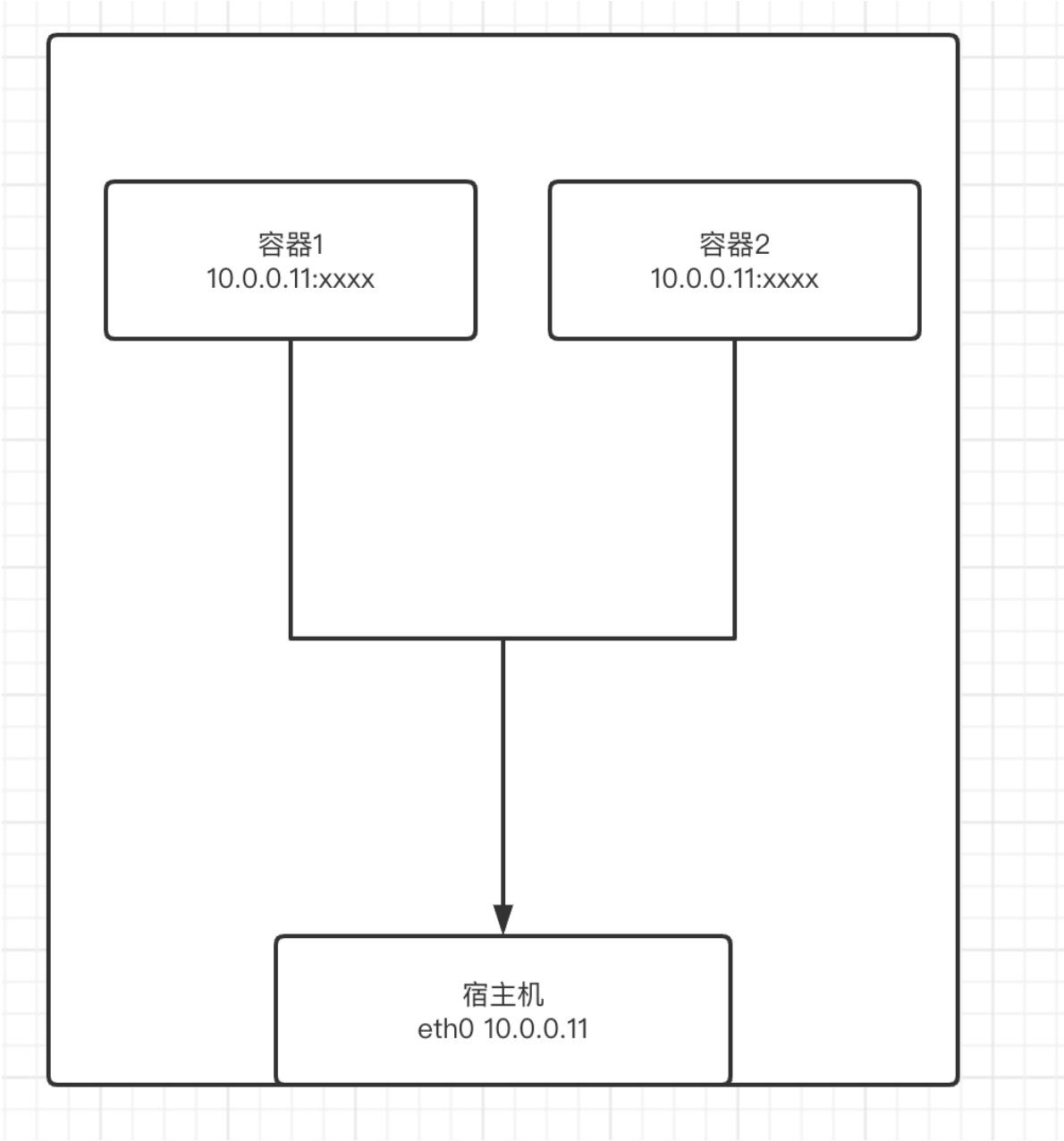
```
[root@docker-11 ~]# cat /etc/docker/daemon.json
{
  "bip": "192.168.2.1/24",
  "registry-mirrors": ["https://ig2l319y.mirror.aliyuncs.com"]
}
```

3.Host模式

3.1 Host模式说明

```
1.Host模式启动的容器不会虚拟出自己的网卡和IP，而是使用宿主机的IP和端口。
2.但是其他的资源比如文件系统和进程列表还是和宿主机隔离的。
3.启动容器需要使用指定的参数 --network host
4.Host模式不支持端口映射
5.因为直接使用宿主机的网络资源，所以性能较好
```

3.2 Host模式示意图



3.3 Host模式演示

```
#查看当前宿主机的端口和容器运行情况
[root@docker-11 ~]# netstat -lntup|grep 80
[root@docker-11 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

#运行一个nginx容器
[root@docker-11 ~]# docker run -d --network host nginx
f6f44b316317f1727d648801836653866fe25f2ad8c24bf6fe9e7e2e8ee1b6ea
[root@docker-11 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS
    PORTS     NAMES
f6f44b316317   nginx     "/docker-entrypoint.…"   5 seconds ago   Up 4
seconds             charming_blackwell

#再次查看宿主机端口情况
[root@docker-11 ~]# netstat -lntup|grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
   20842/nginx: master
```

```
tcp6       0      0 :::80                   :::*                    LISTEN
   20842/nginx: master

#进入容器后配置源信息
[root@docker-11 ~]# docker exec -it f6f44b316317 /bin/bash
root@docker-11:/# cat >/etc/apt/sources.list << 'EOF'
> deb https://mirrors.tuna.tsinghua.edu.cn/debian/ buster main contrib non-
free
> deb https://mirrors.tuna.tsinghua.edu.cn/debian/ buster-updates main
contrib non-free
> deb https://mirrors.tuna.tsinghua.edu.cn/debian/ buster-backports main
contrib non-free
> deb https://mirrors.tuna.tsinghua.edu.cn/debian-security buster/updates
main contrib non-free
> EOF
root@docker-11:/# apt update

#在容器内安装网络命令
root@docker-11:/# apt install iproute2 net-tools -y

#查看网络信息
root@docker-11:/# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.1.1  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::42:edff:fefc:671f  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ed:fc:67:1f  txqueuelen 0  (Ethernet)
        RX packets 22  bytes 1106 (1.0 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 26  bytes 2154 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.11  netmask 255.255.255.0  broadcast 10.0.0.255
        inet6 fe80::20c:29ff:fe9a:74d9  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:9a:74:d9  txqueuelen 1000  (Ethernet)
        RX packets 208690  bytes 293033849 (279.4 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 67922  bytes 4640476 (4.4 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 12  bytes 1563 (1.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 12  bytes 1563 (1.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3.4 Host模式注意

不能端口映射

```
[root@docker-11 ~]# docker run -p 8080:80 --network host -d nginx
WARNING: Published ports are discarded when using host network mode
e4fc457b171fa488db1c2bb0293d2eaaf8948b5ecbac339feb92f943aa565bf1
```

host模式下端口不能重复.如果再次启动相同端口的虚拟机就会失败

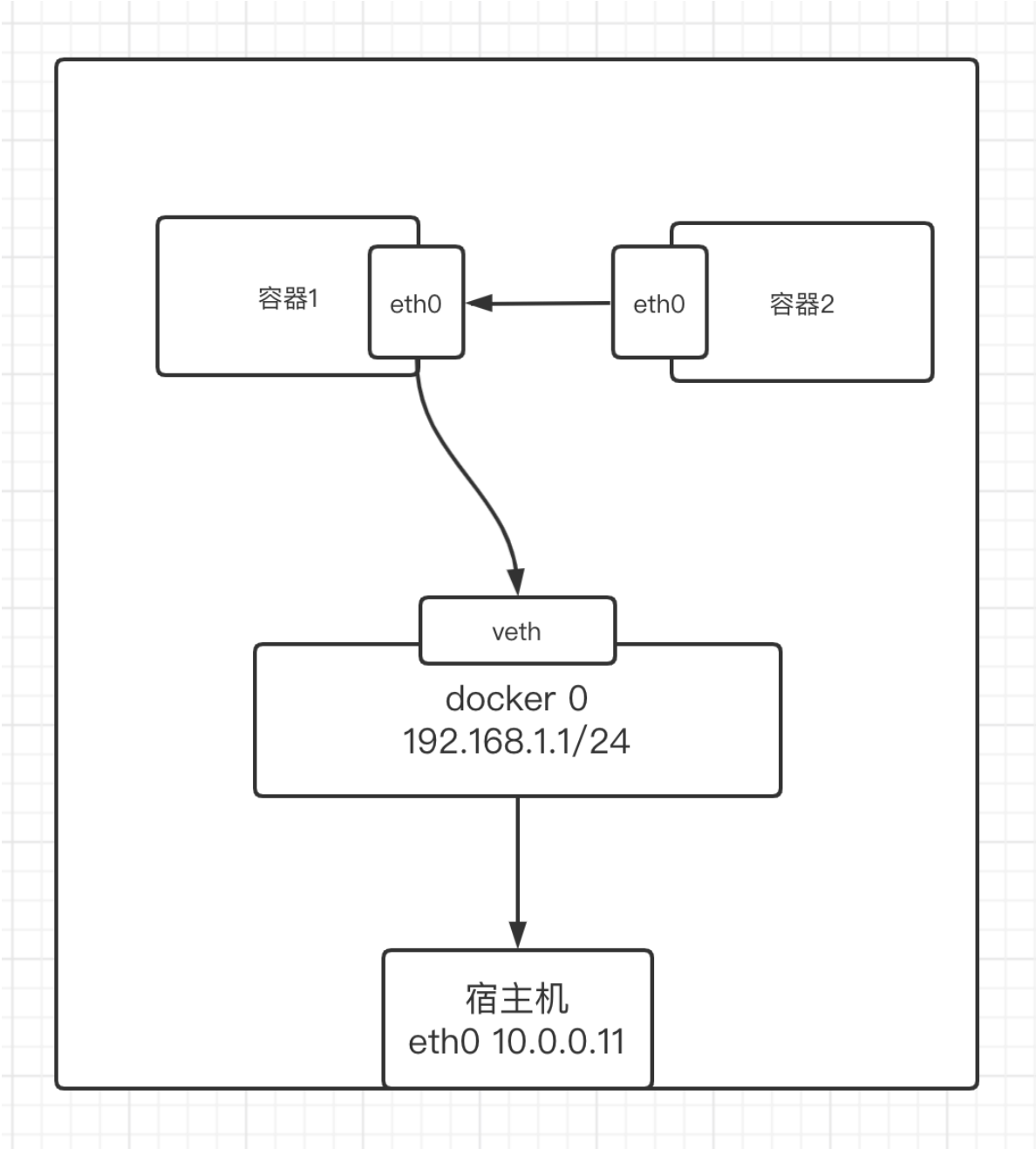
```
#启动多个host模式的容器
[root@docker-11 ~]# docker run -d --network host nginx
638c60c90e5d17f75de1805f81e8b064c4e144616e186b57a8170fc789f1a71b
[root@docker-11 ~]# docker run -d --network host nginx
9866a4ac70f4a08bd91e3a0f820ee796f1353d3fbf93f5f8ff48a52fb84850e7

#查看容器发现并没有启动成功
[root@docker-11 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
     PORTS     NAMES
f6f44b316317   nginx     "/docker-entrypoint.…"   11 minutes ago   Up 11
minutes             charming_blackwell

#通过查看失败的容器日志发现是端口冲突了
[root@docker-11 ~]# docker logs -f 638c60c90e5d
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to
perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-
default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of
/etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in
/etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-
templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-
processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2021/07/21 13:21:17 [emerg] 1#1: bind() to 0.0.0.0:80 failed (98: Address
already in use)
nginx: [emerg] bind() to 0.0.0.0:80 failed (98: Address already in use)
```

4.Container模式

4.1 Container模式说明

```
1.Container模式创建的容器不会创建自己的网卡和IP，而是和一个已经存在的容器共享同一个网络空间
2.Container模式的容器和宿主机网络空间互相隔离。
```

4.2 Container模式示意图



4.3 Container模式演示

```
#运行第一个容器没有nginx服务
docker run -it --name web1 -p 80:80 nginx
docker exec -it web1 /bin/bash
curl 127.0.0.1

#运行第二个容器拥有nginx服务
docker run -d --name web2 --network container:web1 nginx
curl 127.0.0.1
```

5.None模式

5.1 None模式介绍

```
如果使用None模式，则容器不会创建任何网络配置，没有网卡也没有IP地址，因此机会不会使用这种模式
```

5.2 使用None模式

```
[root@docker-11 ~]# docker run --network none -it busybox:latest /bin/sh
/ #
/ # ip a
/ # ping 10.0.0.11
PING 10.0.0.11 (10.0.0.11): 56 data bytes
ping: sendto: Network is unreachable
/ # route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
```

6.自定义网络模式

6.1 自定义网络模式特点

```
自定义网络可以独立设置容器的使用的网段，而且在同一网络里的虚拟机不需要link就可以直接使用容器名进行互相访问
```

6.2 自定义网络模式语法

```
#创建自定义网络
docker network create -d <mode> --subnet <CIDR> --gateway <网关> <自定义网络名称>

#引用自定义网络
docker run --network <自定义网络名称> <镜像名称>

#删除自定义网络
doccker network rm <自定义网络名称或网络ID>
```

6.3 自定义网络模式实验

```
#创建自定义网络
docker network create -d bridge --subnet 192.168.100.0/24 --gateway 192.168.1.1 sz-net

#查看信息
docker inspect my-net

#查看网卡
ip a

#查看网桥
brctl show

#利用自定义的网络创建容器
##运行第一个容器
docker run --name busybox_1 -it --network my-net busybox /bin/sh
ip a
route -n

##运行第二个容器
```

```
docker run --name busybox_2 -it --network my-net busybox /bin/sh
ip a
route -n
ping busybox_1
```

第10章 Docker容器单机编排工具

1.docker-compose介绍

```
Compose 是用于定义和运行多容器 Docker 应用程序的工具。
通过Compose，您可以使用YML文件来配置应用程序需要的所有服务。
写好yaml文件之后，只需要运行一条命令，就会按照资源清单里的配置运行相应的容器服务。
```

Compose 使用的三个步骤：

```
1.使用 Dockerfile 定义应用程序的环境。
2.使用 docker-compose.yml 定义构成应用程序的服务，这样它们可以在隔离环境中一起运行。
3.最后，执行 docker-compose up 命令来启动并运行整个应用程序。
```

官方版本说明:

```
https://docs.docker.com/compose/compose-file/compose-versioning/
```

2.安装docker-compose

方法1:直接yum安装-版本比较老

```
yum install docker-compose
```

方法2:使用官方脚本安装

```
curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

3.docker-compose命令格式

```
build       #构建镜像
bundle      #从当前docker compose 文件生成一个以<当前目录>为名称的json格式的Docker Bundle 备 份文件
config -q   #查看当前配置，没有错误不输出任何信息
create      #创建服务，较少使用
down        #停止和删除所有容器、网络、镜像和卷
events      #从容器接收实时事件，可以指定json 日志格式，较少使用
exec        #进入指定容器进行操作
help        #显示帮助细信息
images      #显示镜像信息
kill        #强制终止运行中的容器
logs        #查看容器的日志
pause       #暂停服务
port        #查看端口
```

```
ps          #列出容器
pull        #重新拉取镜像，镜像发生变化后，需要重新拉取镜像，较少使用
push        #上传镜像
restart     #重启服务
rm          #删除已经停止的服务
run         #一次性运行容器
scale       #设置指定服务运行的容器个数
start       #启动服务
stop        #停止服务
top         #显示容器运行状态
unpause     #取消暂定
up          #创建并启动容器
```

4.docker-compose语法介绍

官方英文参考文档:

```
https://github.com/compose-spec/compose-spec/blob/master/spec.md
```

菜鸟教程翻译文档:

```
https://www.runoob.com/docker/docker-compose.html
```

模板案例:

```
version: '版本号'
services:
  服务名称1:
    image: 容器镜像
    container_name: 容器名称
    environment:
      - 环境变量1=值1
      - 环境变量2=值2
    volumes:
      - 存储驱动1:容器内的数据目录路径
      - 宿主机目录路径:容器内的数据目录路径
    ports:
      - 宿主机端口:映射到容器内的端口
    networks:
      - 自定义网络的名称
    links:
      - namenode

  服务名称2:
    image: 容器镜像
    container_name: 容器名称
    environment:
      - 环境变量1=值1
      - 环境变量2=值2
    volumes:
      - 存储驱动2:对应容器内的数据目录路径
    ports:
      - 宿主机端口:映射到容器内的端口
    networks:
      - 自定义网络的名称
```

```
    links:
      - namenode

networks:
  default:
    external: true
    name: 自定义网络名称
```

5.使用docker-compose部署zabbix

官方文档:

```
https://www.zabbix.com/documentation/5.0/zh/manual/installation/containers
```

使用自定义网络:

```
docker network create -d bridge --subnet 172.16.1.0/24 --gateway 172.16.1.1 zabbix-net
```

修改后的docker-compose文件:

```
version: '3.9'
services:
  mysql:
    image: mysql:5.7
    container_name: mysql
    user: 2000:2000
    environment:
      - "MYSQL_ROOT_PASSWORD=123"
      - "MYSQL_DATABASE=zabbix"
      - "MYSQL_USER=zabbix"
      - "MYSQL_PASSWORD=zabbix"
    command:
      --character-set-server=utf8
      --collation-server=utf8_bin
    volumes:
      - /data/mysql:/var/lib/mysql

  zabbix-server-mysql:
    image: zabbix/zabbix-server-mysql
    container_name: zabbix-server-mysql
    environment:
      - "DB_SERVER_HOST=mysql"
      - "MYSQL_USER=zabbix"
      - "MYSQL_PASSWORD=zabbix"
    ports:
      - "10051:10051"
    depends_on:
      - mysql

  zabbix-web-nginx-mysql:
    image: zabbix/zabbix-web-nginx-mysql
    container_name: zabbix-web-nginx-mysql
    environment:
      - "DB_SERVER_HOST=mysql"
```

```
      - "MYSQL_USER=zabbix"
      - "MYSQL_PASSWORD=zabbix"
      - "ZBX_SERVER_HOST=zabbix-server-mysql"
      - "PHP_TZ=Asia/Shanghai"
    ports:
      - "80:8080"

networks:
  default:
    external: true
    name: zabbix-net
```

6.使用docker-compose部署wordpress

```
version: '3'
services:
  mysql:
    image: mysql:5.7
    container_name: mysql
    user: 2000:2000
    environment:
      - "MYSQL_ROOT_PASSWORD=123"
      - "MYSQL_DATABASE=wordpress"
      - "MYSQL_USER=wordpress"
      - "MYSQL_PASSWORD=wordpress"
    volumes:
      - "/data/wordpress:/var/lib/mysql"
    ports:
      - "3306:3306"
    command:
      --character-set-server=utf8
      --collation-server=utf8_bin

  nginx_php:
    image: nginx_php:v1
    container_name: nginx_php
    ports:
      - "80:80"

networks:
  default:
    external: true
    name: wordpress
```

7.docker-compose部署jenkins

```
进入jenkins容器里运行dockeri
```

8.docker-compose运行EBK

```
version: '2.2'
services:
  es01:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.13.4
```

```
    container_name: es01
    environment:
      - node.name=es01
      - cluster.name=es-docker-cluster
      - discovery.seed_hosts=es02
      - cluster.initial_master_nodes=es01,es02
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    volumes:
      - /data/es_01:/usr/share/elasticsearch/data
    ports:
      - 9200:9200
    networks:
      - elastic
  es02:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.13.4
    container_name: es02
    environment:
      - node.name=es02
      - cluster.name=es-docker-cluster
      - discovery.seed_hosts=es01
      - cluster.initial_master_nodes=es01,es02
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    volumes:
      - /data/es_02:/usr/share/elasticsearch/data
    networks:
      - elastic

  kibana:
    image: docker.elastic.co/kibana/kibana:7.13.4
    container_name: kibana
    environment:
      SERVER_NAME: 10.0.0.12
      ELASTICSEARCH_HOSTS: http://es01
    ports:
      - 5601:5601
    networks:
      - elastic

networks:
  elastic:
    driver: bridge
```

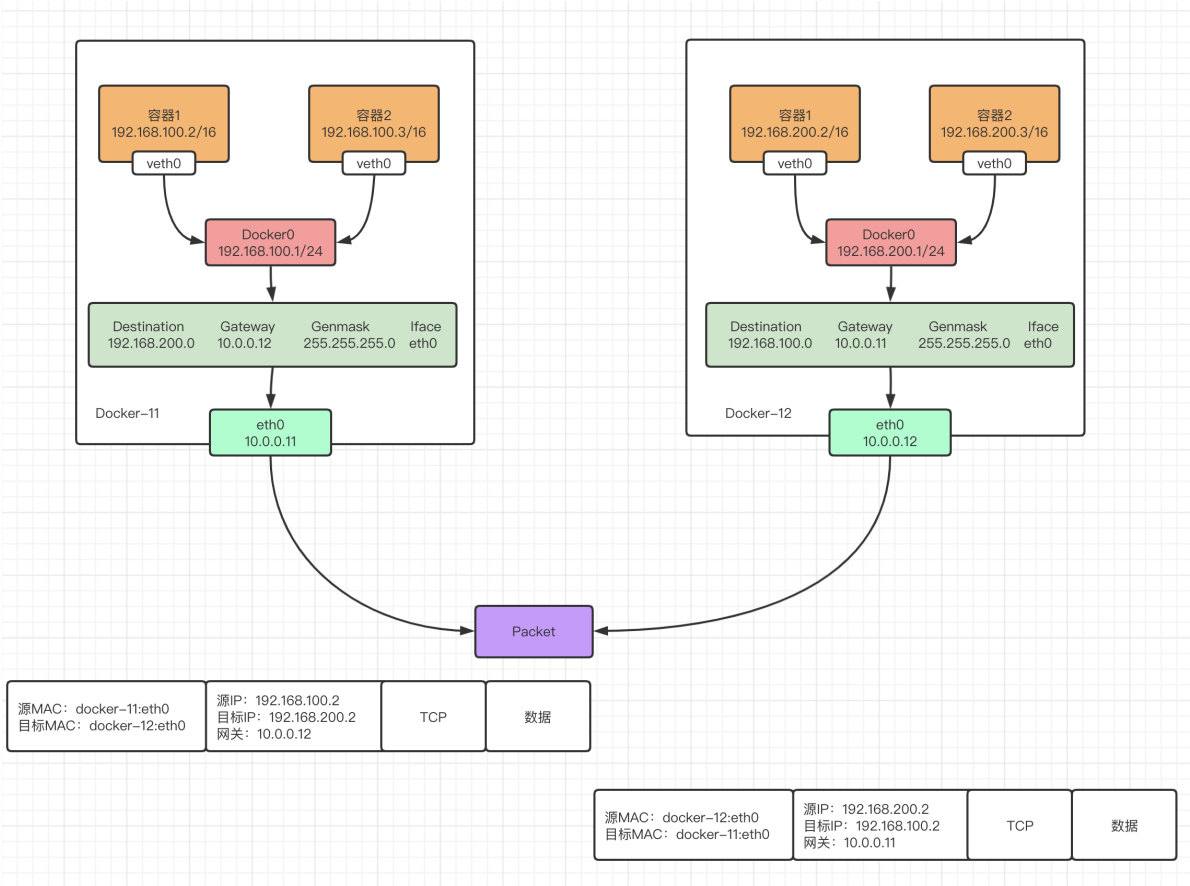
第11章 Docker容器夸主机通信

1.Docker跨主机网络类型

```
静态路由
flannel
Overlay
macvlan
calico
```

2.静态路由模式

2.1 静态路由模式说明



配置说明:

```
1.两台宿主机的容器IP处于不同的网段
2.两台宿主机都配置了静态路由，发给对方的网段的数据包通过eth0网卡，网关指向对方的eth0地址
3.防火墙开发内网转发规则
```

2.2 两台主机创建不同的docker0网段

docker-11配置

```
cat > /etc/docker/daemon.json << 'EOF'
{
  "bip": "192.168.100.1/24",
  "registry-mirrors": ["https://ig2l319y.mirror.aliyuncs.com"]
}
EOF
systemctl daemon-reload
systemctl restart docker
ip a|grep 192.168.100
```

docker-12配置

```
cat > /etc/docker/daemon.json << 'EOF'
{
  "bip": "192.168.200.1/24",
  "registry-mirrors": ["https://ig2l319y.mirror.aliyuncs.com"]
}
EOF
systemctl daemon-reload
systemctl restart docker
ip a|grep 192.168.200
```

2.3 添加静态路由和iptables规则

docker-11配置

```
route add -net 192.168.200.0/24 gw 10.0.0.12
iptables -A FORWARD -s 10.0.0.0/24 -j ACCEPT
```

docker-12配置

```
route add -net 192.168.100.0/24 gw 10.0.0.11
iptables -A FORWARD -s 10.0.0.0/24 -j ACCEPT
```

2.4 跨主机容器通信测试

docker-11启动容器

```
[root@docker-11 ~]# docker run -it busybox /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:42:c0:a8:64:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.2/24 brd 192.168.100.255 scope global eth0
       valid_lft forever preferred_lft forever
```

docker-12启动容器

```
[root@docker-12 ~]# docker run -it busybox /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:42:c0:a8:c8:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.200.2/24 brd 192.168.200.255 scope global eth0
       valid_lft forever preferred_lft forever
```

docker-11启动容器访问docker-12容器测试

```
/ # ping -c 1 192.168.200.2
PING 192.168.200.2 (192.168.200.2): 56 data bytes
64 bytes from 192.168.200.2: seq=0 ttl=62 time=0.531 ms
```

docker-12启动容器访问docker-11容器测试

```
/ # ping -c 1 192.168.100.2
PING 192.168.100.2 (192.168.100.2): 56 data bytes
64 bytes from 192.168.100.2: seq=0 ttl=62 time=0.631 ms
```

2.5 抓包查看

docker-11抓包

```
[root@docker-11 ~]# yum install tcpdump -y
[root@docker-11 ~]# tcpdump -i eth0 -nn icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
19:59:41.665444 IP 10.0.0.11 > 192.168.200.2: ICMP echo request, id 7, seq 0, length 64
19:59:41.665791 IP 192.168.200.2 > 10.0.0.11: ICMP echo reply, id 7, seq 0, length 64
```

docker-12抓包

```
[root@docker-12 ~]# yum install tcpdump -y
[root@docker-12 ~]# tcpdump -i eth0 -nn icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:01:20.616844 IP 10.0.0.12 > 192.168.100.2: ICMP echo request, id 7, seq 0, length 64
20:01:20.617351 IP 192.168.100.2 > 10.0.0.12: ICMP echo reply, id 7, seq 0, length 64
```

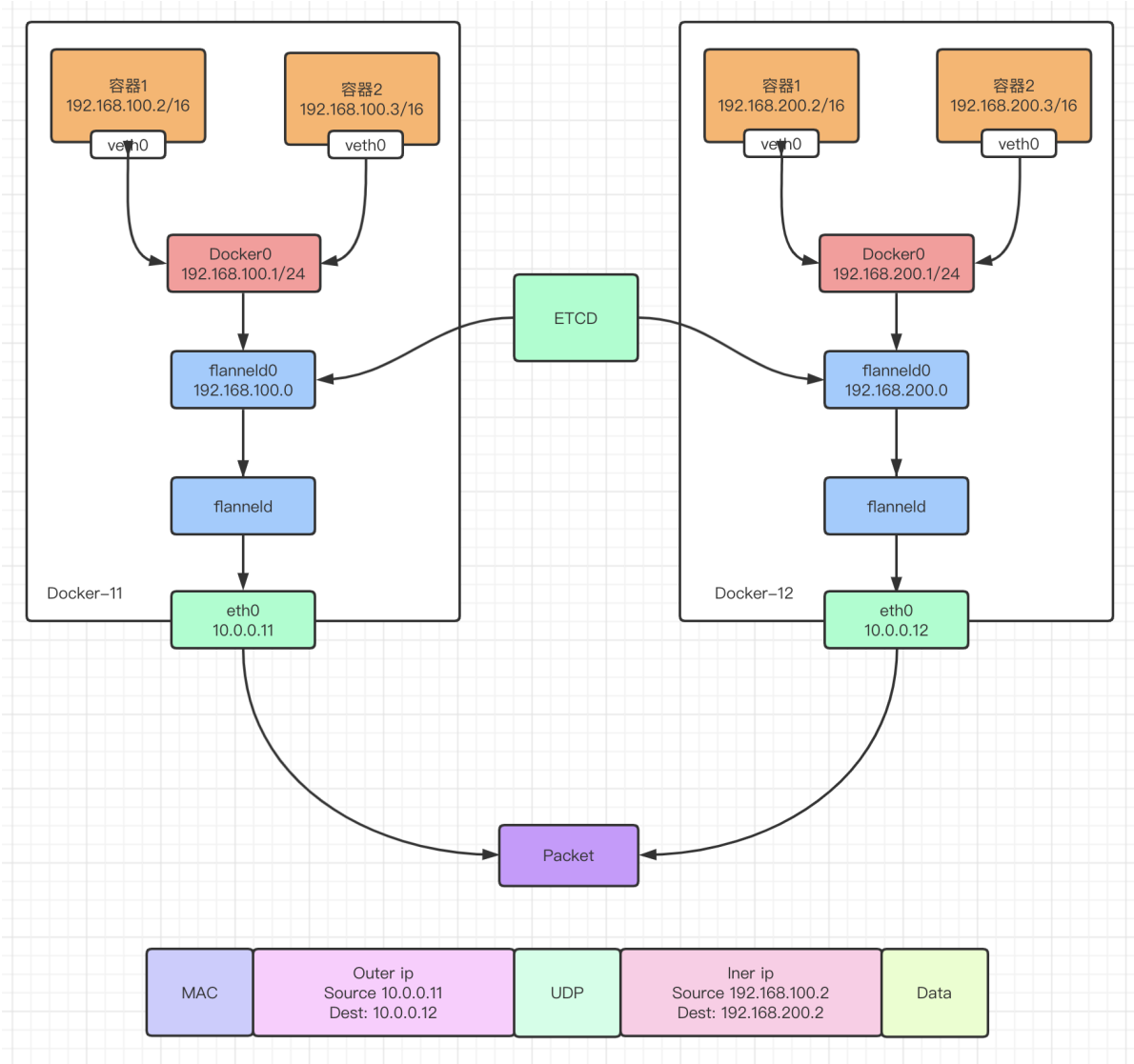
3.跨主机通信-flannel实现

3.1 flannel介绍

```
Flannel是一种基于overlay网络的跨主机容器网络解决方案，即将TCP数据包封装在另一种网络包里面进行路由转发和通信，Flannel是CoreOS开发,专门用于docker多机互联的一个工具,让集群中的不同节点主机创建的容器都具有全集群唯一的虚拟ip地址
```

3.2 flannel通信原理

流程图解



文字说明

```
1.数据从源容器中发出后，经由所在主机的docker0虚拟网卡转发到flannel0虚拟网卡。

2.源主机的flanneld服务将原本的数据内容UDP封装后根据自己的路由表投递给目的节点的flanneld服务，数据到达目标主机后被解包，然后直接进入目的节点的flannel0虚拟网卡，然后被转发到目的主机的docker0虚拟网卡，最后就像本机容器通信一样由docker0路由到达目标容器。

3.使每个结点上的容器分配的地址不冲突。Flannel通过Etcd分配了每个节点可用的IP地址段后，再修改Docker的启动参数。“--bip=X.X.X.X/X”这个参数，它限制了所在节点容器获得的IP范围。
```

3.3 实验环境

```
10.0.0.11  etcd,flannel,docker
10.0.0.12  flannel,docker
```

3.4 docker-11安装配置etcd

单节点安装etcd

```
yum install etcd -y
```

编辑配置文件

```
cat > /etc/etcd/etcd.conf << 'EOF'
# [member]
ETCD_NAME=default
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_CLIENT_URLS="http://10.0.0.11:2379,http://127.0.0.1:2379"

# #[cluster]
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.11:2379"
EOF
```

启动etcd

```
systemctl start etcd
systemctl enable etcd
```

测试etcd功能

```
etcdctl -C http://10.0.0.11:2379 cluster-health
etcdctl -C http://10.0.0.11:2379 set /testdir/testkey "Hello world"
etcdctl -C http://10.0.0.11:2379 get /testdir/testkey
```

防火墙

```
iptables -A INPUT -p tcp -m tcp --dport 2379 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A INPUT -p tcp -m tcp --dport 2380 -m state --state NEW,ESTABLISHED -j ACCEPT
```

3.5 安装配置Flannel-两台机器都操作

安装Flannel

```
yum install flannel -y
```

配置Flannel

```
cp /etc/sysconfig/flanneld /opt/flanneld.bak
cat > /etc/sysconfig/flanneld << 'EOF'
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD_ENDPOINTS="http://10.0.0.11:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_PREFIX="/atomic.io/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
EOF
```

配置etcd数据库

```
etcdctl mk /atomic.io/network/config '{ "Network": "192.168.0.0/16" }'
```

启动flanneld

```
systemctl start flanneld.service
systemctl enable flanneld.service
```

检查端口

```
netstat -lntup|grep flannel
```

3.6 配置Docker关联Flannel网络

修改docker配置文件:

```
vim /usr/lib/systemd/system/docker.service
.................
EnvironmentFile=/run/flannel/docker
ExecStart=/usr/bin/dockerd -H fd:// $DOCKER_NETWORK_OPTIONS
.................
systemctl daemon-reload
systemctl restart docker
```

3.7 创建防火墙规则

```
iptables -P FORWARD ACCEPT
```

3.8 创建容器测试

docker-11创建容器:

```
docker run -it busybox /bin/sh
```

查看IP地址:

```
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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
11: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1472 qdisc noqueue
    link/ether 02:42:c0:a8:38:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.2/24 brd 192.168.56.255 scope global eth0
       valid_lft forever preferred_lft forever
```

docker-12创建容器:

```
docker run -it busybox /bin/sh
```

查看IP地址:

```
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    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
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1472 qdisc noqueue
    link/ether 02:42:c0:a8:24:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.36.2/24 brd 192.168.36.255 scope global eth0
       valid_lft forever preferred_lft forever
```

测试容器间可否通讯:

```
ping 192.168.58.2
ping 192.168.58.3
```

4.macvlan模式

4.1 创建网络

```
docker network create -d macvlan --subnet 10.0.0.0/24 --gateway 10.0.0.2 -o
parent=eth0 macvlan_1
```

4.2 启动容器

docker-11启动容器

```
docker run -it --network macvlan_1 --ip 10.0.0.100 alpine
```

docker-12启动容器

```
docker run -it --network macvlan_1 --ip 10.0.0.200 alpine
```

启动后互相ping发现可以正常通讯

```
ping 10.0.0.100
ping 10.0.0.200
```

5.跨主机通信-Consul实现

5.1 Consul介绍

```
Consul是一个服务网格（微服务间的 TCP/IP，负责服务之间的网络调用、限流、熔断和监控）解决方
案，它是一个一个分布式的，高度可用的系统，而且开发使用都很简便。
它提供了一个功能齐全的控制平面，主要特点是：服务发现、健康检查、键值存储、安全服务通信、多
数据中心。
```

5.2 二进制安装步骤

```
wget https://releases.hashicorp.com/consul/1.4.4/consul_1.4.4_linux_amd64.zip
unzip consul_1.4.4_linux_amd64.zip
mv consul /usr/bin/
chmod +x /usr/bin/consul
nohup consul agent -server -bootstrap -ui -data-dir /var/lib/consul -client=10.0.0.11 -bind=10.0.0.11 &>/var/log/consul.log &
tail -f /var/log/consul.log
```

5.3 修改docker-11启动文件

```
[root@docker-11 ~]# vim /lib/systemd/system/docker.service
#ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --cluster-store consul://10.0.0.11:8500 --cluster-advertise 10.0.0.11:2375
```

5.4 重启docker-11

```
systemctl daemon-reload
systemctl restart docker.service
```

5.5 同样方法修改docker-12的配置

```
[root@docker-12 ~]# vim /lib/systemd/system/docker.service
#ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --cluster-store consul://10.0.0.11:8500 --cluster-advertise 10.0.0.12:2375
```

5.6 重启docker2

```
systemctl daemon-reload
systemctl restart docker.service
```

5.7 在docker主机上创建overlay网络

在docker1上创建网络，然后会自动同步到docker2上

```
docker network create -d overlay overlay_net
```

5.8 分别在两个节点上创建容器

docker1上运行命令

```
docker run -it --net=overlay_net --name busybox01 busybox:latest
```

docker2上运行命令

```
docker run -it --net=overlay_net --name busybox02 busybox:latest
```

5.9 测试联通性

```
docker run -it --net=overlay_net --name busybox01 busybox:latest
#ping 10.0.0.3
```

第12章 Docker资源限制

1.Docker资源限制说明

```

```

2.容器的内存限制

官方文档:

```
https://docs.docker.com/config/containers/resource_constraints/
```

Docker限制内存相关参数:

```
-m  允许容器使用的最大内存，单位有k,m,g
--oom-kill-disable
```

下载压测工具镜像:

```
docker pull lorel/docker-stress-ng
```

压测工具参数说明:

```
#查看帮助说明
docker run --name mem_test -it --rm lorel/docker-stress-ng

#常用参数
 -m N, --vm N            启动N个workers，默认一个256M内
```

创建一个没有内存限制的容器:

```
#启动一个前台窗口任务
docker run --name mem_test -it lorel/docker-stress-ng --vm 2

#另开一个窗口查看
CONTAINER ID   NAME      CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
BLOCK I/O   PIDS
49493229356b   c1        99.46%    514.2MiB / 1.934GiB   25.96%    1.1kB / 0B
  0B / 0B     5
```

创建一个限制了内存大小的容器

```
#启动一个前台窗口任务
docker run --name mem_test --rm -m 300m -it lorel/docker-stress-ng --vm 2

#新开窗口查看
CONTAINER ID   NAME       CPU %     MEM USAGE / LIMIT   MEM %     NET I/O
BLOCK I/O   PIDS
7d1a3b482a3a   mem_test   98.75%    294.9MiB / 300MiB   98.30%    656B / 0B
0B / 0B     5
```

3.容器的CPU限制

官方文档:

```
https://docs.docker.com/config/containers/resource_constraints/
```

Docker限制CPU相关参数:

```
--cpus=<value>
```

查看宿主机CPU核数:

```
[root@docker-11 ~]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
```

压测工具命令:

```
#不限制容器的CPU使用，压测工具开启4个CPU
docker run --name cpu_test -it --rm lorel/docker-stress-ng --cpu 4

#新开窗口查看CPU占用情况
CONTAINER ID   NAME       CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
   BLOCK I/O     PIDS
8701e7f14f6f   cpu_test   402.31%   9.973MiB / 1.934GiB   0.50%     1.1kB /
0B   1.58MB / 0B   5

#限制容器只能使用1.5个CPU
docker run --cpus 1.5 --name cpu_test -it --rm lorel/docker-stress-ng --cpu 4

#查看容器运行状态
CONTAINER ID   NAME       CPU %     MEM USAGE / LIMIT    MEM %     NET I/O
  BLOCK I/O   PIDS
ae710912bb3e   cpu_test   149.31%   14.9MiB / 1.934GiB   0.75%     656B / 0B
  0B / 0B     5
```

第13章 Docker监控

1.docker自带的监控命令

```
docker container ps  :查看正在运行的容器
docker container top  :知道某个容器运行了哪些进程
docker container stats :显示每个容器各种资源使用情况
```

2.cAdvisor+ prometheus+ grafana组件介绍

2.1 cAdvisor介绍

```
1.cAdvisor是google开发的容器监控工具，cAdvisor会显示当前host的资源使用情况，包括CPU，内存，网络，文件系统。
2.不过cAdvisor提供的操作界面略显简陋，而且需要在不同页面之间跳转，并且只能监控一个host，这不免让人质疑他的实用性，但cAdvisor有一个亮点是可以将监控到的数据导出给第三方工具，有这些工具进一步加工处理。
3.所以我们可以把cAdvisor定位为一个监控数据收集器，收集和导出数据是他的强项，而非展示数据。
cAdvisor支持很多第三方工具，其中就包含prometheus
```

2.2 prometheus

```
Prometheus是一个非常优秀的监控工具。提供了监控数据搜集，存储，处理，可视化和告警一系列完整的解决方案。
包含组件:
Node Exporter :负责收集host硬件和操作系统数据，以容器的形式运行在所有host上
cAdvisor :负责收集容器数据，以容器的形式运行在所有host上
```

2.3 grafana

```
grafana是一款支持多种数据源的图形展示工具
```

3.使用docker-compose部署

3.1 地址规划

```
10.0.0.11   cAdvisor+ Node Exporter +prometheus+ grafana
10.0.0.12   cAdvisor+ Node Exporter
```

3.2 编写prometheus配置文件

```
cat > prometheus.yml << 'EOF'
scrape_configs:
- job_name: cadvisor
  scrape_interval: 5s
  static_configs:
  - targets:
    - 10.0.0.11:8080
    - 10.0.0.12:8080

- job_name: prometheus
  scrape_interval: 5s
  static_configs:
  - targets:
```

```
    - 10.0.0.11:9090

- job_name: node_exporter
  scrape_interval: 5s
  static_configs:
  - targets:
    - 10.0.0.11:9100
    - 10.0.0.12:9100
EOF
```

3.2 编写docker-compose文件

docker-11配置

```
cat >docker-compose.yml<<EOF
version: '3.2'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
    - 9090:9090
    command:
    - --config.file=/etc/prometheus/prometheus.yml
    volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
    depends_on:
    - cadvisor

  node-exporter:
    image: prom/node-exporter:latest
    container_name: node_exporter
    ports:
    - 9100:9100

  cadvisor:
    image: google/cadvisor:latest
    container_name: cadvisor
    ports:
    - 8080:8080
    volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    ports:
    - 3000:3000
EOF
```

docker-12配置:

```
cat >docker-compose.yml<<EOF
```

```
version: '3.2'
services:
  node-exporter:
    image: prom/node-exporter:latest
    container_name: node_exporter
    ports:
    - 9100:9100

  cadvisor:
    image: google/cadvisor:latest
    container_name: cadvisor
    ports:
    - 8080:8080
    volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
EOF
```

运行命令：

```
docker-compose -f docker-compose.yml up -d
```

4.web页面操作

访问地址:

```
10.0.0.11:3000
admin admin
```

添加数据源：

```
DataSources
Name:Prometheus
URL:http://10.0.0.11:9090
```

下载监控面板文件:

```
https://grafana.com/api/dashboards/10619/revisions/1/download
```

第13章 Jenkins自动化部署Docker

1.部署流程

部署流程：

```
1.下载代码
2.编译镜像
3.推送镜像
4.停止正在运行的容器
5.启动新容器
6.清理jenkins主机上的镜像
```

回滚流程：

```
1.选择需要回滚的版本
2.停止正在运行的容器
3.启动新容器
```

2.pipeline脚本

```
pipeline{
    agent any

    parameters {
        gitParameter name: 'git_version',
                     branchFilter: 'origin/(.*)',
                     type: 'PT_TAG',
                     defaultValue: 'v1.0',
                     description: '发布新版本'
        choice(name: 'base_image', choices: ['nginx:1.16','nginx:1.17'],description: '请选择基础镜像版本')
        choice(name: 'deploy_env', choices: ['deploy','rollback'],description: 'deploy: 发布版本\nrollback: 回滚版本')


    }

    stages{
        stage("下载代码"){
            steps{
                checkout([$class: 'GitSCM',
                                  branches: [[name: '*/master']],
                                  doGenerateSubmoduleConfigurations: false,
                                  extensions: [[$class: 'RelativeTargetDirectory',
                                  relativeTargetDir: 'game']],
                                  submoduleCfg: [],
                                  userRemoteConfigs: [[credentialsId: 'b8c1f793-47ed-4903-995d-2273673d8f87',
                                  url: 'git@10.0.0.200:dev/docker-pipeline.git']]])
            }
        }

        stage("编译镜像"){
            when {
                environment name: 'deploy_env', value: 'deploy'
            }
            steps{
                writeFile file: "Dockerfile", text: """FROM 10.0.0.205/base_image/${params.base_image}\nADD game /usr/share/nginx/html/"""
                sh "docker build -t 10.0.0.205/image/game:${params.git_version} . && docker push 10.0.0.205/image/game:${params.git_version}"
            }
        }

```

```
        stage("推送镜像"){
            when {
                environment name: 'deploy_env', value: 'deploy'
            }
            steps{
                sh "docker build -t 10.0.0.205/image/game:${params.git_version} . && docker push 10.0.0.205/image/game:${params.git_version}"
            }
        }

        stage("部署容器"){
            when {
                environment name: 'deploy_env', value: 'deploy'
            }
            steps{
                sh 'ssh 10.0.0.204 "docker stop game && docker rm game && docker run --name game -p 80:80 -d 10.0.0.205/image/game:${git_version} && docker ps"'
            }
        }

        stage("清理构建镜像"){
            when {
                environment name: 'deploy_env', value: 'deploy'
            }
            steps{
                sh "docker rmi 10.0.0.205/image/game:${params.git_version}"
            }
        }

        stage("回滚镜像"){
            when {
                environment name: 'deploy_env', value: 'rollback'
            }

            steps{
                sh 'ssh 10.0.0.204 "docker stop game && docker rm game && docker run --name game -p 80:80 -d 10.0.0.205/image/game:${git_version} && docker ps"'
            }
        }
    }
}
```