

1.Nginx 服务

第一章 Nginx概述

nginx 是一个高性能的 HTTP 和反向代理 web 服务，Nginx 是由伊戈尔·塞索耶夫为俄罗斯访问量第二的 Rambler.ru 站点（俄文：Рамблер）开发的，第一个公开版本 0.1.0 发布于 2004 年 10 月 4 日，其将源代码以类 BSD 许可证的形式发布，因它的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名，2011 年 6 月 1 日，nginx 1.0.4 发布。

在 BSD-like 协议下发行，其特点是占有内存少，并发能力强，事实上 nginx 的并发能力在同类型的网页服务器中表现较好，国内大型网站：百度、京东、新浪、网易、腾讯、淘宝等都在使用 Nginx。并且支持 TCP、UDP、HTTP、gRPC、fastcgi、SCG 等等协议处理，实现了相应通信协议的请求解析、长连接、代理转发负载均衡、等互联网常见的应用功能，同时它还是一个高并发服务软件。

1.Nginx 的优点

1. **高性能**：采用事件驱动架构，能够高效处理大量并发连接。
2. **低资源消耗**：内存和 CPU 占用低，官方数据显示，它可以在仅占用 2.5MB 内存的情况下维持 1 万个无活动连接，有效抵御 DOS 攻击。
3. **反向代理和负载均衡**：支持多种负载均衡算法，提升系统可用性和性能。
4. **静态文件处理**：对静态文件（如图片、CSS）响应速度快，效率高。
5. **支持多种协议**：除了 HTTP/HTTPS，还支持 TCP、UDP、gRPC 等。
6. **模块化架构**：可通过丰富的模块扩展功能，满足不同需求。
7. **SSL/TLS 支持**：内置支持安全传输，加密 HTTPS 请求。
8. **热部署能力**：支持无缝更新和配置更改，保证高可用性。

2.Nginx 的缺点

1. **动态内容处理**：处理动态内容时，通常需要与后端应用服务器。

3.Nginx应用场景？

- http 服务器。
 - Nginx 是一个 http 服务可以独立提供 http 服务。可以做网页静态服务器。
- 虚拟主机。
 - 可以实现在一台服务器虚拟出多个网站，例如个人网站使用的虚拟机。
- 反向代理和负载均衡。
 - 当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使 Nginx 做反向代理。并且多台服务器可以平均分担负载，不会因为某台服务器负载高宕机而某台服务器闲置的情况。
 - API 网关：nginx 中也可以配置安全管理、比如可以使用 Nginx 搭建 API 接口网关，对每个接口服务进行拦截，在微服务架构中处理请求路由、认证和流量控制。

4.正向代理

客户端通过代理服务器访问互联网。客户端与代理服务器之间直接通信，代理服务器代表客户端向目标服务器发送请求。例如在内网当中的客户端想要上网，就需要通过正向代理服务器将请求转发到互联网当中的服务器，而对于互联网的服务器来看，请求是代理服务器发送的，而隐藏后端服务器；最常见的就是我们的上外网挂梯子（VPN）。

5.反向代理

反向代理是客户端请求被代理服务器接收，然后由代理服务器将请求转发给后端服务器，然后再将服务器的相应结果反馈给客户端，客户端并不知道实际服务的后端服务器。

1. 保护了真实的 web 服务器，web 服务器对外不可见，外网只能看到反向代理服务器，而反向代理服务器上并没有真实数据，因此，保证了 web 服务器的资源安全。
2. 反向代理为基础产生了动静资源分离以及负载均衡的方式，减轻 web 服务器的负担，加速了对网站访问速度。
3. 节约了有限的 IP 地址资源，企业内所有的网站共享一个在 internet 中注册的IP地址，这些服务器分配私有地址，采用虚拟主机的方式对外提供服务。

6.为什么使用Nginx

- 跨平台、配置简单、方向代理、高并发连接：处理2-3万并发连接数，官方监测能支持5万并发，内存消耗小：开启10个nginx才占150M内存，nginx处理静态文件好，耗费内存少，
- 而且Nginx内置的健康检查功能：如果有一个服务器宕机，会做一个健康检查，再发送的请求就不会发送到宕机的服务器了。重新将请求提交到其他的节点上。
- 使用Nginx的话还能：
 1. 节省宽带：支持GZIP压缩，可以添加浏览器本地缓存
 2. 稳定性高：宕机的概率非常小
 3. 接收用户请求是异步的

7.什么是C10K问题

所谓 c10k 问题，指的是服务器如何支持 10k 个并发连接，也就是 concurrent 10000 connection（这也是 c10k 这个名字的由来）。

早期的互联网可以说是一个小群体的集合。互联网还不够普及，用户也不多，一台服务器同时在线 100 个用户，在当时已经算是大型应用了，所以并不存在 C10K 的难题。互联网的爆发期是在 www 网站、浏览器出现后。最早的互联网称之为 Web1.0，大部分的使用场景是下载一个 HTML 页面，用户在浏览器中查看网页上的信息，这个时期也不存在 C10K 问题。

Web2.0 时代到来后，就不同了。一方面是，互联网普及率大大提高了，用户群体几何倍增长。另一方面是，互联网不再是单纯地浏览 www 网页，逐渐开始进行交互，而且应用程序的逻辑也变得更复杂。从简单的表单提交，到即时通信和在线实时互动，C10K 的问题才体现出来了。因为每一个用户都必须与服务器保持连接，才能进行实时数据交互。例如在淘宝双11这样节日网站，同一时间的并发 TCP 连接很可能已经过亿。

最初的服务器都是基于进程/线程模型的，新到来一个 TCP 连接，就需要分配 1 个进程（或者线程）。进程又是操作系统最昂贵的资源，一台机器无法创建很多进程。如果是 C10K，就要创建 1 万个进程，那么就单机而言，操作系统是无法承受的（往往出现效率低下、甚至完全瘫痪）。如果是采用分布式系统，维持 1 亿用户在线需要 10 万台服务器，成本巨大，也只有企业巨头们，才有财力购买如此多的服务器。

与其它web服务器对比

Apache 是一个免费的开源的 Web 服务器软件，广泛使用，是最流行的 Web 服务器之一。它能够在多种平台上运行，支持多种编程语言，如 PHP、Python 和 Perl。通过丰富的模块，Apache 可以扩展功能，包括身份验证、URL 重写、缓存等。其灵活性使得 Apache 在动态内容处理方面表现出色，但在高并发环境下可能不如 Nginx。

Nginx 是一个免费的开源的 Web 服务器软件，主要用于高性能的 Web 服务器和反向代理。它在高并发环境下表现优异，是 Apache 的有力竞争者。Nginx 以轻量级、低资源消耗和高效的事件驱动架构著称，适合处理静态文件和反向代理请求，同时支持负载均衡和缓存等功能。

Tomcat 是一个免费的开源的 Web 服务器软件，主要用于运行 Java Servlet 和 JavaServer Pages (JSP) 应用。与 Apache 和 Nginx 不同，Tomcat 主要处理 Java 相关的动态内容，而不是静态文件。因此，通常需要与其他 Web 服务器（如 Apache 或 Nginx）配合使用，以实现更好的性能和处理能力。Tomcat 的主要优势在于其对 Java EE 规范的支持，使其成为 Java 开发者的首选。

第二章 部署Nginx服务器

1. 下载Nginx软件包

```
wget https://nginx.org/download/nginx-1.18.0.tar.gz
```

2. 安装依赖软件

```
1 | yum -y install gcc make pcre-devel zlib-devel openssl-devel
```

3. 编译安装nginx

```
1 # 解压nginx-1.18.0.tar.gz包
2 tar -xf nginx-1.18.0.tar.gz
3 # 进入nginx-1.18.0目录
4 cd ./nginx-1.18.0
5
6
7 #开始编译nginx
8 ./configure \
9   --prefix=/usr/local/nginx \
10  --user=nginx \
11  --group=nginx \
12  --with-http_ssl_module
13
14 make -j4 && make install          # 编译使用4核CPU编译 和编译安装
15
16 # 查看nginx目录
17 ls /usr/local/nginx
18 conf                                #配置文件目录
19 html                                 #网页目录
20 logs                                 #日志目录
21 sbin                                 #程序目录
22
23 # 进入到主程序目录
24 cd /usr/local/nginx/
25
26 # 查看nginx编译参数
```

```

27 sbin/nginx -v
28 nginx version: nginx/1.18.0
29 built by gcc 7.3.0 (GCC)
30 built with OpenSSL 1.1.1f  31 Mar 2020
31 TLS SNI support enabled
32 configure arguments: --prefix=/usr/local/nginx --user=nginx --group=nginx --with-
33 http_ssl_module
34
35 # 开启nginx服务
36 sbin/nginx
37 nginx: [emerg] getpwnam("nginx") failed    #启动报错，没有nginx用户需要自己创建。
38
39 # 创建nginx用户
40 useradd -s /sbin/nologin nginx           # 创建用户nginx不能登录系统当中
41
42 # 在启动nginx程序
43 sbin/nginx
44
45 # 查看nginx进程
46 ps -ef | grep nginx
47 root      34028      1  0 11:33 ?        00:00:00 nginx: master process ./nginx
48 nginx     34029  34028  0 11:33 ?        00:00:00 nginx: worker process
49 root      34032  27883  0 11:33 pts/2    00:00:00 grep nginx
50 # 访问测试
51 curl localhost

```

浏览器访问测试



4. Nginx 命令使用

```

1 # 启动 Nginx
2 sbin/nginx
3
4 # 停止 Nginx
5 sbin/nginx -s stop
6
7 # 重新加载配置
8 sbin/nginx -s reload
9
10 # 测试配置文件
11 sbin/nginx -t
12
13 # 查看 Nginx 版本
14 sbin/nginx -v

```

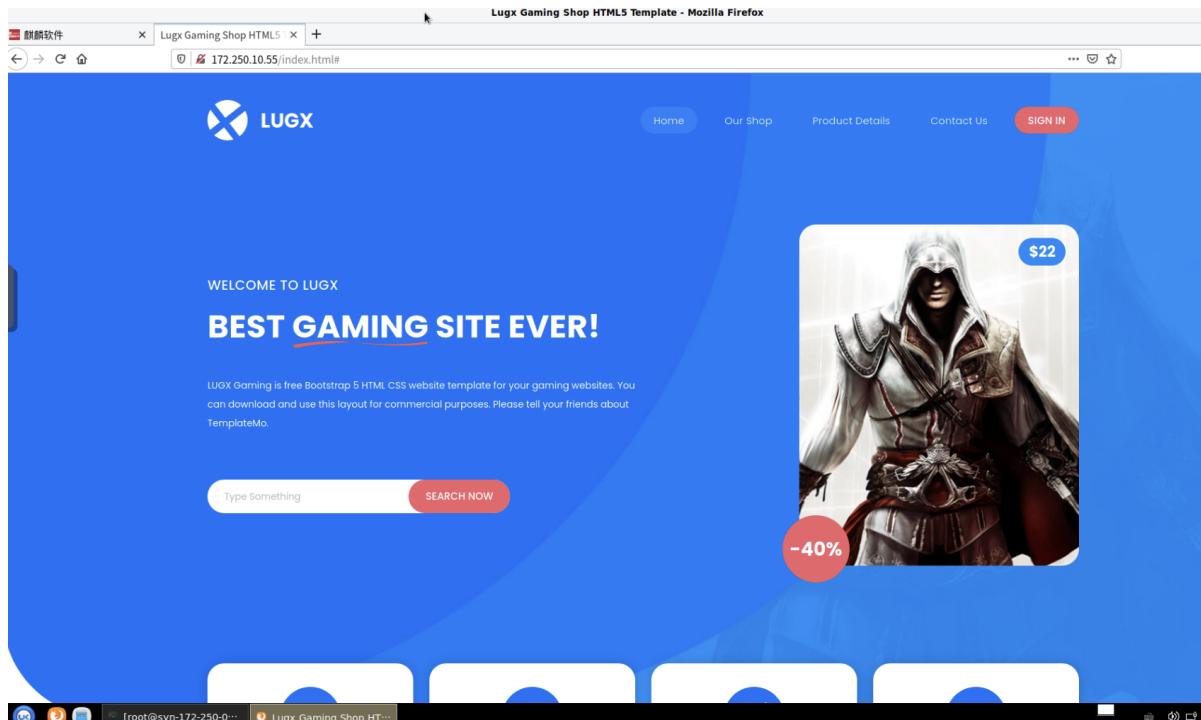
5. 测试网页目录

```
1 echo "hello world" >> html/test.html
2 curl 172.250.10.55/test.html
3 hello world
```

6. 测试图片网页网站

```
1 # 查看网站网页
2 ls /opt/exam/kylin/App_Web/templateemo_589_lugx_gaming/
3 assets contact.html index.html product-details.html shop.html vendor
4 # 将网站网页拷贝到html目录下
5 cp -r /opt/exam/kylin/App_Web/templateemo_589_lugx_gaming/* /usr/local/nginx/html
```

使用浏览器访问<http://172.250.10.55>



显示服务端nginx端口和查看客户端访问IP和随机端口，需要先快速访问网页，在执行这个命令才可以查看到。

```
1 [root@PXLX-02 web]# ss -antup | grep nginx
2 tcp      LISTEN      0      511          0.0.0.0:80          0.0.0.0:*
3 users:(("nginx",pid=35991,fd=8),("nginx",pid=35990,fd=8),("nginx",pid=35989,fd=8),
4 ("nginx",pid=35988,fd=8),("nginx",pid=34028,fd=8))
5 tcp      ESTAB      0      0          172.250.10.55:80      172.250.10.55:34438
6 users:(("nginx",pid=35988,fd=6))
7 tcp      ESTAB      0      0          172.250.10.55:80      172.250.10.55:34452
8 users:(("nginx",pid=35988,fd=21))
9 tcp      ESTAB      0      0          172.250.10.55:80      172.250.10.55:34450
10 users:(("nginx",pid=35988,fd=20))
11 tcp      ESTAB      0      0          172.250.10.55:80      172.250.10.55:34446
12 users:(("nginx",pid=35988,fd=18))
13 tcp      ESTAB      0      0          172.250.10.55:80      172.250.10.55:34448
14 users:(("nginx",pid=35988,fd=19))
15 tcp      ESTAB      0      0          172.250.10.55:80      172.250.10.55:34442
16 users:(("nginx",pid=35988,fd=17))
```

第三章 Nginx 配置

1.Nginx用户认证配置

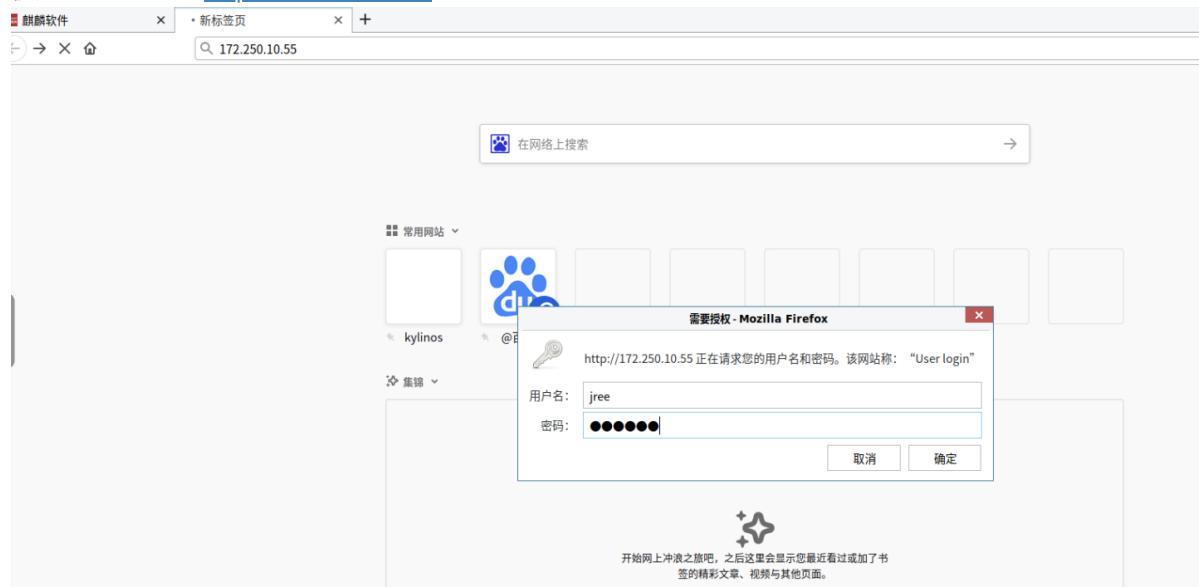
理论知识内容补充：

要求：配置nginx用户认证，登录web界面需要用户身份验证。用户名为 jree 密码为123456

```
1 # 打开nginx 配置文件
2 cd /usr/local/nginx/
3 vim conf/nginx.conf
4 .....
5 location / {
6     root html;
7     index index.html index.htm;
8     auth_basic "User login";          # 开启认证
9     auth_basic_user_file /usr/local/nginx/.htpasswd;    # 指定用户认证文件
在/usr/local/nginx/.htpasswd
10 }
11
12 # 下载httpd-tools 测试工具
13 yum -y install httpd-tools
14
15
16 # 创建用户
17 htpasswd -c /usr/local/nginx/.htpasswd jree      # 创建用户jree输入两次密码
18 New password:
19 Re-type new password:
20 Adding password for user jree
21
22 # 重新加载配置文件
23 cd /usr/local/nginx
24 sbin/nginx -t          #检查配置文件是否正确
25 sbin/nginx -s reload  #重新加载配置文件
```

先清除浏览器缓存Windows/Linux：按下 Ctrl + Shift + Delete

使用浏览器访问：<http://172.250.10.55>



可以在追加多个用户认证验证。

```
1 htpasswd /usr/local/nginx/.htpasswd admin
2 New password:
3 Re-type new password:
4 Adding password for user admin
```

2.web虚拟机配置

2.1基于域名的虚拟机主机

示例配置：

```
1 # 编辑配置文件
2 vim /usr/local/nginx/conf/nginx.conf
3 .....
4 server {
5     listen 80;
6     server_name www.exe.com; #域名
7
8     location / {
9         root /apps/web/nginx/www.exe.com/html;
10        index index.html;
11    }
12 }
13
14 # 创建网页目录
15 mkdir /apps/web/nginx/www.exe.com/html -p
16 # 配置网页内容
17 echo "hello world www.exe.com" > /apps/web/nginx/www.exe.com/html/index.html
18
19 # 重新加载配置文件
20 sbin/nginx -s reload
21
22 # 访问测试01
23 curl www.exe.com #访问不了，因为没有配置域名解析
24
25 # 配置域名解析
26 vim /etc/hosts
27 添加本地IP www.exe.com
28
29 # 访问测试02
30 curl www.exe.com
31 hello world example.com # 成功
```

2.2基于端口的虚拟主机

内容补充：

示例配置：

```
1 # 修改配置文件
2 vim /usr/local/nginx/conf/nginx.conf
3 .....
4 server {
5     listen 8888; #端口
6     server_name www.exe.com ; #域名
7
8     location / {
9         root /apps/web/nginx/port/html;
10        index index.html;
```

```

11     }
12 }
13
14
15 # 创建网页目录
16 mkdir /apps/web/nginx/port/html -p
17
18 # 创建网页内容
19 echo "port 8888" > /apps/web/nginx/port/html/index.html
20
21
22 # 重新加载配置文件
23 sbin/nginx -s reload
24
25 # 访问测试
26 curl localhost:8888

```

2.3基于IP的虚拟机主机

内容补充：

示例配置：

```

1 # 修改配置文件
2 vim /usr/local/nginx/conf/nginx.conf
3 .....
4 server {
5     listen 172.250.10.79:8889; #IP/端口
6     server_name localhost;
7
8     location / {
9         root /apps/web/nginx/ip_addr/html;
10        index index.html;
11    }
12 }
13
14 # 创建网页目录
15 mkdir /apps/web/nginx/ip_addr/html -p
16
17 # 创建网页内容
18 echo "172.250.10.79" > /apps/web/nginx/ip_addr/html/index.html
19
20 # 重启服务器
21 sbin/nginx -s reload
22
23 # 访问测试
24 curl 172.250.10.79:8889
25 172.250.10.79 # 网页内容

```

3.nginx反向代理和负载均衡

3.1 基本反向代理配置、

以下是一个简单的反向代理配置示例，Nginx 会将所有请求转发到后端应用服务器（例如运行在 `http://localhost:3000` 的应用）。

使用 `nohup python3 -m http.server 3000 --bind 127.0.0.1 -d /apps/web/nginx/www/html/ &` 命令可以在当前目录下启动一个简单的 HTTP 服务器。

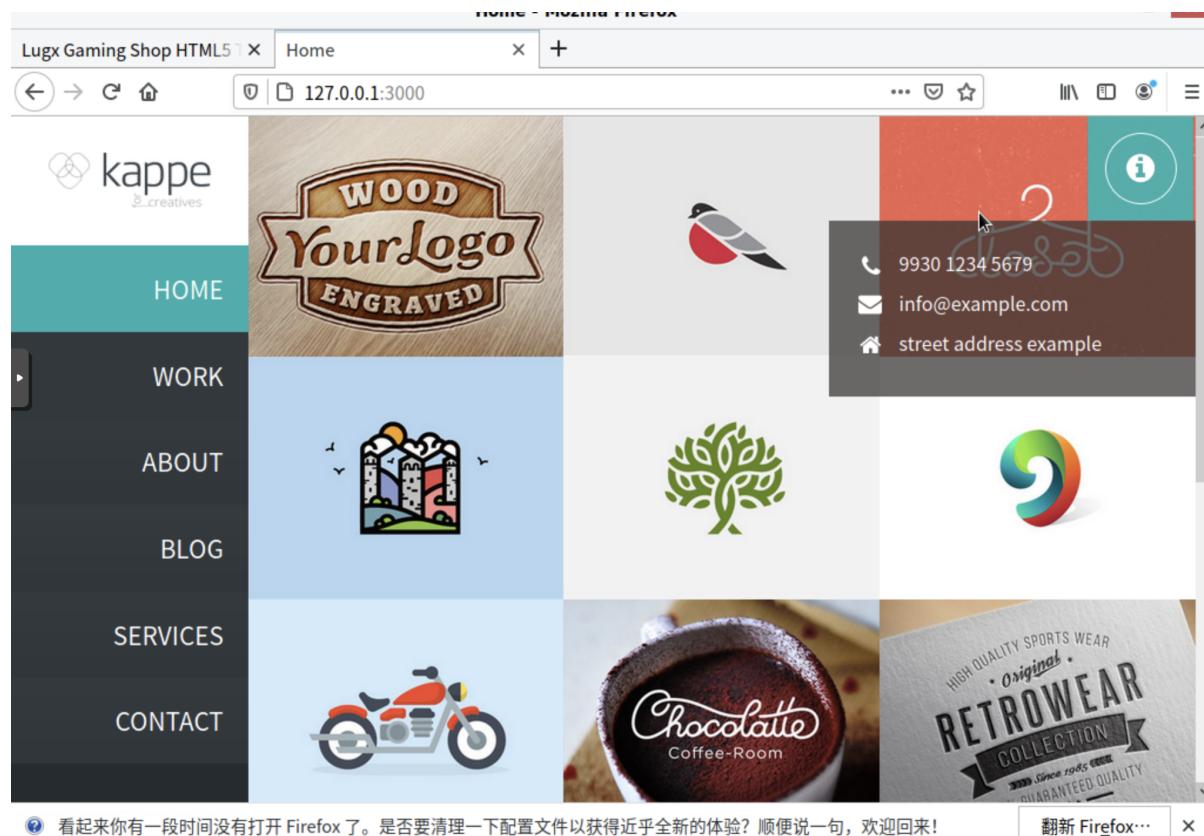
示例配置：

```

1 # 创建网页目录
2 mkdir /apps/web/nginx/www/html/ -p
3 # 解压网页目录
4 unzip /opt/exam/kylin/App_web/www_template.zip -d /apps/web/nginx/www/html/
5 mv /apps/web/nginx/www/html/www_template/* /apps/web/nginx/www/html
6
7 # 查看网页目录
8 ls /apps/web/nginx/www/html
9 404.html about.html blog.html contact.html css images index.html js
single.html work.html 说明.txt
10
11 # 使用python3当中http模块创建一个虚拟主机
12 nohup python3 -m http.server 3000 --bind 127.0.0.1 -d /apps/web/nginx/www/html/ & # 放入后台运行
13
14 # 访问测试
15 curl 127.0.0.1:3000

```

使用浏览器访问127.0.0.1:3000测试



配置nginx反向代理

```

1 # 修改配置文件
2 vim /usr/local/nginx/conf/nginx.conf
3 .....
4 server {
5     listen 8848;
6     server_name www.exe.com;
7
8     location / {
9         proxy_pass http://localhost:3000; # 后端服务器地址
10    }
11 }

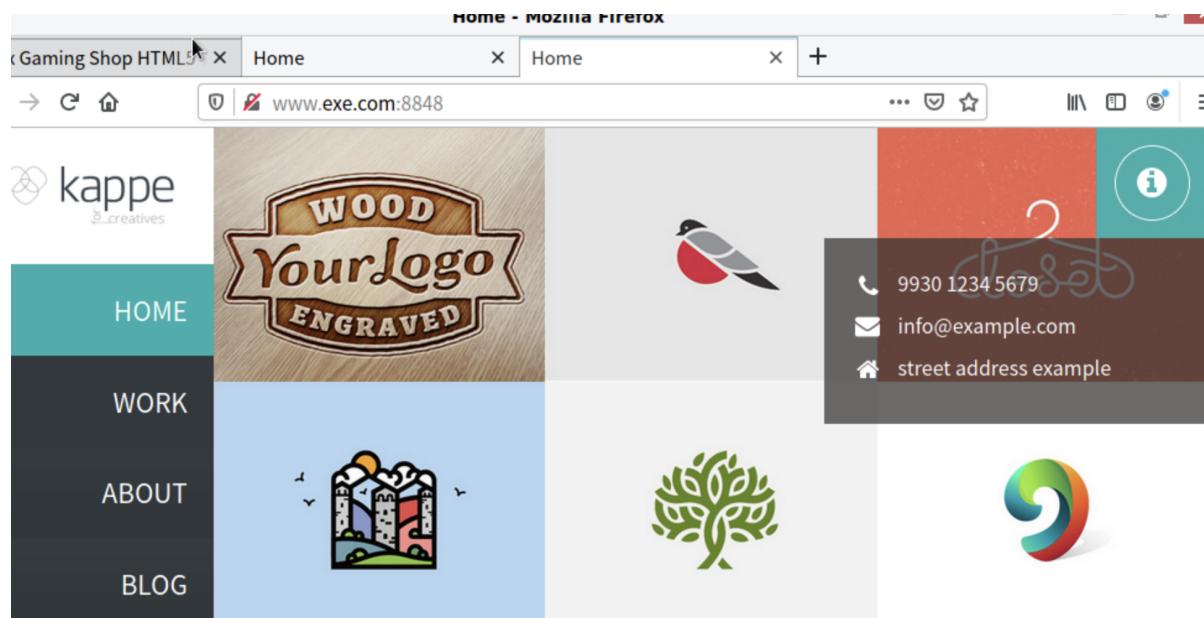
```

```

12
13 # 重启nginx
14 sbin/nginx -s reload      #以防止nginx不生效可以重启nginx
15 sbin/nginx -s stop
16 sbin/nginx -s start
17
18 curl localhost:8848
19 curl www.exe.com:8848

```

浏览器访问测试www.exe.com:8848



3.2 反向代理与负载均衡

Nginx 还可以用于将请求分发到多个后端服务器，以实现负载均衡。

示例配置：

```

1 # 创建网页内容
2 mkdir /apps/web/nginx/web1/html/ -p
3 mkdir /apps/web/nginx/web2/html/ -p
4 echo "hellow world web1" > /apps/web/nginx/web1/html/index.html
5 echo "hellow world web2" > /apps/web/nginx/web2/html/index.html
6
7 # 在使用python3当中http模块创建一个虚拟主机
8 nohup python3 -m http.server 3001 --bind 127.0.0.1 -d /apps/web/nginx/web1/html &
9 nohup python3 -m http.server 3002 --bind 127.0.0.1 -d /apps/web/nginx/web2/html &
10 # 修改nginx文件
11 vim /usr/local/nginx/conf/nginx.conf
12 .....
13 upstream backend {          #定义源服务器组
14     server 127.0.0.1:3001;   # 后端服务器 1
15     server 127.0.0.1:3002;   # 后端服务器 2
16     # 默认轮询
17 }
18
19 server {
20     listen 8880;
21     server_name www.upstream.com;
22     location / {
23         proxy_pass http://backend; # 调用 upstream 定义的服务器组

```

```
24     }
25 }
```

3.3 Nginx 调度算法

1. 轮询 (Round Robin) 后端服务器的健康检查

`max_fails`：在 `fail_timeout` 时间段内，允许的最大失败次数。这里设置为 `2`，表示如果在 `fail_timeout` 指定的时间内，该服务器连续失败达到 2 次，则会被认为是不可用的。

`fail_timeout`：指定服务器被认为不可用的持续时间（以秒为单位）。在此时间段内，Nginx 不会尝试请求该服务器。这里设置为 `30`，表示如果服务器在 30 秒内失败 2 次，则在接下来的 30 秒内不再请求该服务器。

```
1 upstream backend {
2     server 127.0.0.1:3001;      # 后端服务器 1
3     server 127.0.0.1:3002  max_fails=2  fail_timeout=30;      # 后端服务器 2
4     # 默认使用轮询
5 }
6
7 # 重启服务器
8 sbin/nginx -s reload
9
10 # 查找python3 3002端口进程
11 ps -ef | grep python3 # 查找进程PID
12 root      1170      1  0 Oct30 ?          00:00:08 /usr/bin/python3 -Es
13 /usr/sbin/tuned -l -P
14 root      2285      2055  0 Oct30 ?          00:00:00 /usr/bin/python3
15 /usr/local/lib64/mate-indicators/libexec/reset_applet_position.py
16 root      17765     1  0 09:15 ?          00:00:04 python3 -m http.server 3000 --
17 bind 127.0.0.1 -d /apps/web/nginx/www/html/
18 root      27245     27163  0 15:37 pts/0    00:00:00 python3 -m http.server 3001 --
19 bind 127.0.0.1 -d /apps/web/nginx/web1/html
20 root      27248     27163  0 15:38 pts/0    00:00:00 python3 -m http.server 3002 --
21 bind 127.0.0.1 -d /apps/web/nginx/web2/html
22 root      28075     27163  0 15:51 pts/0    00:00:00 grep python3
23
24 # 杀死进程
25 kill -9 27248
26
27 # 访问测试
28 curl localhost:8880
29 hellow world web1 #只有web1，不会向web2发送请求了。
30
31 # 将服务加回去
32 nohup python3 -m http.server 3002 --bind 127.0.0.1 -d /apps/web/nginx/web2/html &
33
34 # 服务正常了
35 curl localhost:8880
36 hellow world web2
37 curl localhost:8880
38 hellow world web1
```

2. IP 哈希 (IP Hash)

根据客户端 IP 地址的哈希值来分配请求，确保同一客户端的请求总是发送到同一台后端服务器。

```
1 upstream backend {
2     ip_hash; # 使用 IP 哈希算法
3     server 127.0.0.1:3001;
4     server 127.0.0.1:3002;
```

```
5 }
6
7 # 重新加载nginx配置文件
8 sbin/nginx -s reload
9
10 #访问测试
11 curl localhost:8880
12 helloworld web1
13 curl localhost:8880
14 helloworld web1
```

3. 加权轮询 (Weighted Round Robin)

根据服务器的权重来决定请求的分配比例，权重值越大，接收到的请求越多。

```
1 upstream backend {
2     server 127.0.0.1:3001 weight=3;    # 权重为 3
3     server 127.0.0.1:3002 weight=1;    # 权重为 1
4 }
```

4. 重定向和重写

4.1. 重定向 (Redirect)

内容补充：

重定向是将用户从一个 URL 转到另一个 URL。常见的重定向有 301（永久性重定向）和 302（临时重定向）。重定向会改变用户的请求 URL。

示例配置

在 Nginx 配置中实现 301 重定向：

```
1 # 配置文件修改
2 server {
3     listen      8090;
4     server_name www.test.com localhost;
5     location / {
6         return 301 http://www.webxc.com; # 访问 www.test.com 重定向到www.webxc.com
7     }
8 }
9
10 server {
11     listen      80;
12     server_name www.webxc.com;
13     location / {
14         root  html;
15         index index.html index.htm;
16     }
17 }
18
19 #配置网页内容
20 echo "helloworld web1" > /usr/local/nginx/html/index.html
21
22
23 # 重启服务
24 sbin/nginx -s reload
25
26 # 配置域名解析
27 vim/etc/hosts
```

```

28 127.0.0.1 www.test.com www.webxc.com
29
30 # 访问测试
31 curl -I www.test.com:8090
32 HTTP/1.1 301 Moved Permanently
33 Server: nginx/1.18.0
34 Date: Thu, 31 Oct 2024 09:41:59 GMT
35 Content-Type: text/html
36 Content-Length: 169
37 Connection: keep-alive
38 Location: http://www.webxc.com # 看到重定向地址
39
40 # 访问测试
41 curl -L www.test.com:8090 # -L选项 重定向跟随 不加-L 选项使用curl访问
42 www.test.com:8090 不会跟随。
42 hello wrold web1

```

浏览器访问测试www.test.com:8090 会自动重定向跟随网页



4.2 重写 (Rewrite)

内容补充：

重写是修改请求的 URI，而不改变用户的请求 URL。重写常用于将请求转发到不同的处理程序或文件。

示例配置

在 Nginx 配置中使用重写规则：

```

1
2 #第1一个测试
3 server {
4     listen 80;
5     server_name localhost;
6
7     location / {
8         # 可以提供新的内容或静态文件
9         root html; # 网站根目录
10        index index.html;
11        rewrite / http://www.taobao.com;
12    }
13 }
14
15 # 重启nginx
16 sbin/nginx -s reload
17
18 # 访问测试
19 curl -L localhost #转到淘宝
20
21
22
23
24 # 第2个测试
25 server {
26     listen 80;
27     server_name localhost;

```

```
28
29     location / {
30         # 可以提供新的内容或静态文件
31         root html;    # 网站根目录
32         index index.html;
33         rewrite /a.html  /web;  # 访问a.html 跳转到web目录
34     }
35
36     location /web {
37         root   html;
38         index index.html index.htm;
39     }
40
41 }
42
43 # 添加网页内容
44 mkdir  /usr/local/nginx/html/web
45 echo "hello world web index.html" > /usr/local/nginx/html/web/index.html
46
47 # 重启服务器
48 sbin/nginx -s reload
49
50 # 验证测试
51 curl -L localhost/a.html
52 hello world web index.html
53
54 curl -L localhost/a.htmldc
55 hello world web index.html
56
57 curl -L localhost/dc/a.html
58 hello world web index.html
59
60 curl -L localhost/aahtml
61 hello world web index.html
62
63
64 # 第3个正则匹配测试
65 server {
66     listen 80;
67     server_name localhost;
68
69     location / {
70         # 可以提供新的内容或静态文件
71         root html;    # 网站根目录
72         index index.html;
73         rewrite ^/a\.html(.*)$  /web$1    redirect;  # # 将 /a.html 的请求重写到 /web
    目录
74     }
75 }
76
77 # 重启nginx
78 sbin/nginx -s reload
79 # 验证测试
80 curl localhost/a.html
81 hello world web index.html
82
83 curl -L localhost/dc/a.html
84 报404错误
85
86 curl -L localhost/a.html/aaa
```

```
87 报404错误
88
89 # 添加网页内容
90 echo "aaa" > html/web/aaa
91
92 # 访问测试
93 curl -L localhost/a.html/aaa
94 aaa
```

参数解释

rewrite ^/a.html(.*)\$ /web\$1;:

- `^/a.html(.*)$`: 正则表达式，匹配请求路径为 /a.html 的请求。`(.)` 会捕获 /a.html 后面的所有内容。
- `/web$1`: 重写后的路径，其中 `$1` 表示正则表达式中第一个捕获组的内容。如果 /a.html 后面没有其他内容，则请求将被重写为 /web；如果有内容，则重写为 /web 加上相应的内容。

测试

访问 /a.html: 如果您在浏览器中访问 <http://localhost/a.html>，将会被重写到 <http://localhost/web>。

访问 /a.html 后加路径: 如果您访问 <http://localhost/a.html/aaa>，将被重写为 <http://localhost/web/aaa>。

5.配置 HTTPS 和 SSL/TLS 证书。

5.1 配置SSI 虚拟主机

内容补充：

示例配置

```
1 server {
2     listen      443 ssl;
3     server_name localhost;
4
5     ssl_certificate      cert.pem;
6     ssl_certificate_key  cert.key;
7
8     ssl_session_cache    shared:SSL:1m;
9     ssl_session_timeout  5m;
10
11    ssl_ciphers  HIGH:!aNULL:!MD5;
12    ssl_prefer_server_ciphers  on;
13
14    location / {
15        root  https;
16        index index.html index.htm;
17    }
18 }
19
20
21 # 配置私钥
22 openssl genrsa > conf/cert.key 2048 #生成一个 2048 位的 RSA 私钥
23 # 配置公钥/生成自签名证书
24 openssl req -x509 -key conf/cert.key > conf/cert.pem -days 365 #生成一个自签名的证书,
使用指定的私钥，并保存到 conf/cert.pem 文件中，有效期为 365 天
25 You are about to be asked to enter information that will be incorporated
into your certificate request.
26 what you are about to enter is what is called a Distinguished Name or a DN.
27 There are quite a few fields but you can leave some blank
28 For some fields there will be a default value,
```

```

30 If you enter '.', the field will be left blank.
31 -----
32 Country Name (2 letter code) [AU]:CN # 国家名称
33 State or Province Name (full name) [Some-State]:JC #省/州名称
34 Locality Name (eg, city) []:NJ #城市名称
35 Organization Name (eg, company) [Internet Widgits Pty Ltd]:XYD #公司名称
36 Organizational Unit Name (eg, section) []:JFB #部门名称
37 Common Name (e.g. server FQDN or YOUR name) []:web1 #服务器名称
38 Email Address []:1234@qq.com #邮箱名称
39
40
41 # 验证配置
42 nginx -t
43
44 # 创建网页目录和网页内容
45 mkdir /usr/local/nginx/https
46 echo "web https~" > /usr/local/nginx/https/index.html
47
48 # 重启
49 nginx -s stop
50 nginx
51
52 # 访问测试
53 curl -k https://localhost    #-k 忽略证书验证。
54 web https~

```

5.2 重定向 HTTP 到 HTTPS

内容补充：

示例配置

```

1 # HTTP 重定向到 HTTPS
2 server {
3     listen 80;
4     server_name localhost;
5     # 重定向所有 HTTP 请求到 HTTPS
6     return 301 https://$host$request_uri;
7 }
8
9 # 重启nginx
10 sbin/nginx -t
11
12 # 验证测试内容
13 curl -L -k http://localhost
14 web https~

```

Nginx 报错信息

错误代码	错误信息	描述	解决方法
400	Bad Request	请求无效，服务器无法理解请求。	检查请求的格式和语法。
401	Unauthorized	未授权，用户未提供有效的身份验证凭据。	检查认证信息是否正确。

错误代码	错误信息	描述	解决方法
403	Forbidden	禁止访问，服务器拒绝了请求。	检查文件权限和 Nginx 配置，确保用户有访问权限。
404	Not Found	请求的资源未找到。	检查 URL 是否正确，确认资源是否存在。
405	Method Not Allowed	请求方法不被允许。	检查配置，确保使用的 HTTP 方法（如 GET、POST）被允许。
408	Request Timeout	请求超时，服务器未在规定时间内接收到完整请求。	检查网络连接或增加请求超时时间。
500	Internal Server Error	服务器内部错误，未处理的异常或配置问题。	查看 Nginx 错误日志以获取详细信息。
502	Bad Gateway	网关错误，Nginx 作为反向代理时，后端服务器无响应。	检查后端服务是否正常运行，并确认网络连接。
503	Service Unavailable	服务不可用，服务器无法处理请求，通常是由于过载或维护。	检查服务器负载情况，考虑增加资源或修复服务问题。
504	Gateway Timeout	网关超时，Nginx 作为反向代理时，后端服务器响应超时。	检查后端服务的响应时间，可能需要优化后端服务或调整超时设置。

第四章练习题

实验题1：启动一台虚拟机web1，源码安装部署nginx软件

- YUM安装nginx源码包的依赖软件包
- 将素材 /opt/exam/kylin/APP_src/nginx-1.18.0.tar.gz 源码编译安装
- 源码编译安装要求：
 - 安装目录：/usr/local/nginx
 - 安装用户和组：nginx
 - 安装编译模块：--with-http_ssl_modu
- 客户端测试访问nignxy应当出现用认证DC:和密码为123456 默认页面为"hello world "
- 启动nginx服务

操作步骤

```

1 # 解压nginx.tar.gz
2 tar -xf /opt/exam/kylin/APP_src/nginx-1.18.0.tar.gz
3
4 #编译安装
5 cd /opt/exam/kylin/APP_src/nginx-1.18.0
6 ./configure --prefix=/usr/local/nginx --user=nginx --group=nginx --with-
  http_ssl_modu
7 make && make install
8
9 # 创建nginx 用户
10 useradd -s /sbin/nologin nginx
11
12 # 进入nginx目录
13 cd /usr/local/nginx

```

```

14 vim conf/nginx.conf
15 ....
16 location / {
17     root    html;
18     index   index.html index.htm;
19     auth_basic "User login"; # 开启认证
20     auth_basic_user_file /usr/local/nginx/.htpasswd; # 指定用户认证文件
在/usr/local/nginx/.htpasswd
21 }
22 # 下载测试工具
23 yum -y install httpptools
24 # 创建认证用户
25 htpasswd -c /usr/local/nginx/conf/.htpasswd DC
26 New password:
27 Re-type new password:
28 Adding password for user DC
29
30 # 启动nginx
31 sbin/nginx -t #验证配置文件是否正确
32 sbin/nginx
33
34 # 创建网页内容
35 echo "hello world" > html/index.html
36
37 # 访问测试
38 curl -u DC:123456 localhost
39 hello world

```

实验题2：创建基于域名的Web虚拟主机和反向代理

- 创建2个基于域名的虚拟主机，域名分别为：www.link.com和www.destroy.com 端口分别是8081和8082
- 创建2个虚拟主机的根目录，分别为/usr/local/nginx/link
- 将素材/opt/exam/kylin/App_Web/www_template.zip，解压后，放到网站根目录usr/local/nginx/link
- 当访问www.destroy.com转发给本地3307和3308端口，权重是每访问3次3307才访问一次3308。
- 将本地/web/html/web1和web2网页使用python3 http_servcer 模块做成http服务器
- 网页web1/index.html内容为hello web1，网页web2/index.html内容为hello web2
- 客户端修改hosts本地解析文件，实现域名解析
- 客户端测试访问2个虚拟主机域名

```

1 # 修改配置文件
2 vim /usr/local/nginx/conf/nginx.conf
3 ....
4 server {
5     listen      8081;
6     server_name www.link.com;
7     location / {
8         root    link;
9         index   index.html index.htm;
10    }
11 }
12
13 upstream webserver {
14     server 127.0.0.1:3307 weight=3 ;
15     server 127.0.0.1:3008 weight=1 ;
16 }
17

```

```

18 server {
19     listen      8082;
20     server_name www.destroy.com;
21     location / {
22         proxy_pass http://webserver;
23     }
24 }
25
26 #创建网页内容
27 mkdir /usr/local/nginx/link
28 unzip /opt/exam/kylin/App_Web/www_template.zip
29 cp -r www_template/* /usr/local/nginx/link
30
31
32 # 创建负载均衡后端调用的服务器
33 mkdir /web/html/web1 -p
34 mkdir /web/html/web2 -p
35 echo "hello web1" > /web/html/web1/index.html
36 echo "hello web2" > /web/html/web2/index.html
37 nohup python3 -m http.server 3307 --bind 127.0.0.1 -d /web/html/web1/ &
38 nohup python3 -m http.server 3308 --bind 127.0.0.1 -d /web/html/web2/ &
39
40 # 重启nginx
41 sbin/nginx -t
42 sbin/nginx -s reload
43
44 # 配置域名解析
45 vim /etc/hosts
46 127.0.0.1 www.link.com www.destroy.com
47
48 # 访问测试
49 curl www.link.com:8081
50 应该有网页内容
51 #
52 curl www.destroy.com:8082
53 # 写个循环访问测试 权重是3-1
54 for i in {1..10};do
55 > curl www.destroy.com:8082
56 > done
57 hello web1
58 hello web1
59 hello web1
60 hello web2
61 hello web1
62 hello web1
63 hello web1
64 hello web2
65 hello web1
66 hello web1

```

实验题3：访问www.myssl.com重定向到本地ssl

- https虚拟主机根目录为/usr/local/nginx/https
- 生成私钥/usr/local/nginx/conf/myssl.key
- 生成证书/usr/local/nginx/conf/myssl.pem
- 修改nginx配置文件，虚拟主机名为：www.myssl.com端口:8900
- https网页内容为：webserver ssl

- 客户端访问<http://www.myssl.com>测试网站访问效果

```

1 # 修改配置文件
2 server {
3     listen 8900;
4     server_name www.myssl.com;
5     location / {
6         return 301 https://$host$request_uri ;
7     }
8 }
9
10 server {
11     listen      443 ssl;
12     server_name localhost;
13
14     ssl_certificate      myssl.pem;
15     ssl_certificate_key  myssl.key;
16
17     ssl_session_cache    shared:SSL:1m;
18     ssl_session_timeout  5m;
19
20     ssl_ciphers  HIGH:!aNULL:!MD5;
21     ssl_prefer_server_ciphers  on;
22
23     location / {
24         root  https;
25         index index.html index.htm;
26     }
27 }
28
29 # 配置私钥
30 openssl genrsa > conf/myssl.key 2048 #生成一个 2048 位的 RSA 私钥
31 # 配置公钥/生成自签名证书
32 openssl req -x509 -key conf/myssl.key > conf/myssl.pem -days 365 #生成一个自签名的证书，使用指定的私钥，有效期为 365 天
33 You are about to be asked to enter information that will be incorporated
34 into your certificate request.
35 What you are about to enter is what is called a Distinguished Name or a DN.
36 There are quite a few fields but you can leave some blank
37 For some fields there will be a default value,
38 If you enter '.', the field will be left blank.
39 -----
40 Country Name (2 letter code) [AU]:CN # 国家名称
41 State or Province Name (full name) [Some-State]:JC #省/州名称
42 Locality Name (eg, city) []:NJ #城市名称
43 Organization Name (eg, company) [Internet Widgits Pty Ltd]:XYD #公司名称
44 Organizational Unit Name (eg, section) []:JFB #部门名称
45 Common Name (e.g. server FQDN or YOUR name) []:web1 #服务器名称
46 Email Address []:1234@qq.com #邮箱名称
47
48 # 重启nginx
49 sbin/nginx -s reload
50
51 # 创建网页目录和内容
52 mkdir https
53 echo "webserver ssl" > https/index.html
54
55 # 配置域名解析
56 vim /etc/hosts
57 127.0.0.1 www.myssl.com

```

```
58  
59 # 访问测试  
60 curl -k -L http://www.myssl.com:8900  
61 webserver ssl
```

2.RPM包管理

第一章RPM包管理

制作 RPM 软件包

制作 RPM 包是一个相对复杂的过程，但它为软件开发者和系统管理员提供了一个标准化的方法来分发、安装和管理软件。以下是关于如何制作 RPM 包的简要介绍：

准备源代码

确保你有软件的源代码，并且它可以在目标系统上正确编译。

创建 SPEC 文件

SPEC 文件描述了如何编译源代码、安装软件以及包的元数据（如名称、版本、描述等）。

构建 RPM 包

使用 `rpmbuild` 命令和 SPEC 文件来构建 RPM 包。

安装软件

`rpmdevtools` 这个工具还包含一些其他的工具，同时它依赖 `rpm-build`，所以直接安装的话会同时把 `rpm-build` 装上。

主要功能和工具

1. `rpmdev-setuptree`

- 描述：此命令创建 RPM 构建目录结构，通常在 `~/rpmbuild/` 下。
- 使用：`rpmdev-setuptree`

2. `rpmdev-newspec`

- 描述：创建一个新的 SPEC 文件模板。
- 使用：`rpmdev-newspec [name]`

3. `rpmdev-diff`

- 描述：比较两个 tarball 或两个 RPM 包。
- 使用：`rpmdev-diff [options] oldfile newfile`

4. `rpmdev-extract`

- 描述：在当前目录下解压 tarball。
- 使用：`rpmdev-extract archive.tar.gz`

5. `rpmdev-wipetree`

- 描述：清除 `~/rpmbuild/` 目录下的构建内容，但保留 SOURCES 和 SPECS。
- 使用：`rpmdev-wipetree`

6. `rpmdev-rmdevelrpms`

- 描述：删除所有 "-devel" RPM 包。
- 使用：`rpmdev-rmdevelrpms`

7. `rpmdev-checksig`

- 描述：检查源代码 tarball、patch 和 RPM 的签名。

- 使用: `rpmdev-checksig file.rpm`

8. `rpmdev-vercmp`

- 描述: 比较两个 RPM 版本和发布号, 确定它们的排序关系。
- 使用: `rpmdev-vercmp 1.0-1 1.0-2`

9. `rpmdev-packager`

- 描述: 打印包维护者的全名和电子邮件地址。
- 使用: `rpmdev-packager`

10. 设置 RPM 构建环境

- `rpmdev-setuptree` 命令将会在用户的家目录中创建一个 `rpmbuild` 目录, 该目录包含 RPM 构建所需的所有子目录。

默认位置	宏代码	名称	用途
<code>~/rpmbuild/SPECS</code>	<code>%_specdir</code>	Spec 文件目录	保存 RPM 包配置 (.spec) 文件
<code>~/rpmbuild/SOURCES</code>	<code>%_sourcedir</code>	源代码目录	保存源码包 (如 .tar 包) 和所有 patch 补丁
<code>~/rpmbuild/BUILD</code>	<code>%_builddir</code>	构建目录	源码包被解压至此, 并在该目录的子目录完成编译
<code>~/rpmbuild/BUILDROOT</code>	<code>%_buildrootdir</code>	最终安装目录	保存 %install 阶段安装的文件
<code>~/rpmbuild/RPMS</code>	<code>%_rpmdir</code>	标准 RPM 包目录	生成/保存二进制 RPM 包
<code>~/rpmbuild/SRPMS</code>	<code>%_srcrpmdir</code>	源代码 RPM 包目录	生成/保存源码 RPM 包 (SRPM)

宏代码这一列可以在 SPEC 文件中用来指所对应的目录, 类似于全局变量。当然 `~/rpmbuild` 这个文件夹也是有宏代码的, 叫做 `%_topdir`。

创建 SPEC 文件模板

```

1 使用命令 `rpmdev-newspec nginx.spec` 这将在 `~/rpmbuild/SPECS` 目录中创建一个新的 SPEC 文
2 件模板。
3 基本信息:
4 Name: 软件的名称, 这里是 nginx。
5 Version: 软件的版本号。
6 Release: RPM 包的发布版本。通常从 1 开始, 每次 RPM 包更新时递增。
7 Summary: 对软件的简短描述。
8 License: 软件的许可证类型。
9 URL: 软件的官方网站。
10 Source0: 软件的源代码 tarball 的 URL。
11
12 依赖关系:
13 BuildRequires: 构建软件所需的依赖包。
14 Requires: 运行软件所需的依赖包。
15
16 描述:
17 %description: 对软件的详细描述。
18
19 准备阶段:
20 %prep: 为构建做准备, 如解压源代码 tarball。

```

```

21 %autosetup: 一个宏，通常用于自动解压和为构建配置源代码。
22
23 构建阶段：
24 %build: 包含编译软件所需的命令。
25 %configure: 一个宏，通常用于运行 `./configure` 脚本。
26 %make_build: 一个宏，通常用于运行 `make` 命令。
27
28 安装阶段：
29
30 %install: 包含将软件安装到 RPM 构建目录的命令。
31 rm -rf $RPM_BUILD_ROOT: 清理 RPM 构建目录。
32 %make_install: 一个宏，通常用于运行 `make install` 命令。
33
34 文件列表：
35 %files: 列出应包含在 RPM 包中的文件和目录。
36 %license 和 %doc: 宏，用于指定许可证文件和文档。
37
38 变更日志：
39 %changelog: 包含 RPM 包的变更历史。
40
41 这只是 SPEC 文件的基本组成部分。实际的 SPEC 文件可能会包含更多的部分和细节，具体取决于所打包软件的复杂性和需求。
42
43 #构建 RPM 包
44 使用命令 `rpmbuild -ba ~/rpmbuild/SPECS/nginx.spec` 这将会在 `~/rpmbuild/RPMS/` 和 `~/rpmbuild/SRPMS/` 目录下生成二进制和源代码的 RPM 包。

```

第二章 制作nginx rpm软件包

```

1 #安装 rpmbuild
2 yum -y install rpmbuild
3 yum -y install rpm-build
4
5 # 在当前目录下创建rpmbuild目录和nginx.spec文件
6 rpmbuild -ba SPECS/nginx.spec
7 cd /root/rpmbuild/SPECS
8 rpmdev-newspec nginx # 生成nginx.spec 模板
9
10 # 将给nginx源码包复制到rpmbuild的SOURCES目录
11 cp /opt/exam/kylin/APP_src/nginx-1.18.0.tar.gz /root/rpmbuild/SOURCES
12
13
14 # 编辑nginx.spec文件来为构建RPM
15 vim /root/rpmbuild/SPECS/nginx.spec
16
17 Name:      nginx          # 软件名称
18 Version:   1.18.0          # 指定软件版本号
19 Release:   1               # 指定软件包的发布号
20 Summary:   High-performance web # 简单描述软件包概要信息
21
22
23 License:   GPL            # 指定软件包的许可证类型
24 URL:       http://nginx.org # 指定软件包官方网址
25 Source0:   nginx-1.18.0.tar.gz # 源码文件非常重要，要写SOURCES 目录下源码包一样
26
27 BuildRequires: pcre-devel, zlib-devel, openssl-devel, libxml2-devel, libxslt-devel,
28 gd-devel # 编译时依赖包，包括开发库，编译器等
29 Requires:   pcre, zlib, openssl # 指定运行时所需要的依赖关系，包括其他软件，库等

```

```
30 %description # 描述软件包详细信息
31 Nginx is a high-performance HTTP server
32
33 %prep # 安装前准备, 解压
34 %autosetup # 解压源码包
35
36 # 配置安装目录为 /usr/local/nginx, 并启用 SSL 和 GZIP 模块
37 %build
38 ./configure --prefix=/usr/local/nginx \
39             --with-http_ssl_module \
40             --with-http_gzip_static_module \
41             --user=nginx \
42             --group=nginx
43 %make_build # 用于调用 make 命令, 并自动启用多核并行编译, 加快编
44     译速度。
45
46 %install # 将构建好的文件安装到临时构建根目录。
47 rm -rf $RPM_BUILD_ROOT # 作用是在RPM打包过程中清理构建根目录。
48 %make_install # 用于将构建好的文件安装到临时的构建根目录 (%
49 {builddroot}) 中, 以便后续打包。
50
51 # 在软件安装之前执行, 用于创建 nginx 用户和组。如果用户或组已经存在, 它不会重复创建。
52 %pre
53 getent group nginx >/dev/null || groupadd -r nginx
54 getent passwd nginx >/dev/null || useradd -r -g nginx -d /usr/local/nginx -s
55 /sbin/nologin -c "Nginx web server" nginx
56
57 # 在软件安装之后执行, 这个脚本的作用是将 /usr/local/nginx 目录及其内容的所有权赋予 nginx 用户和
58 # 组。
59 %post
60 chown -R nginx:nginx $RPM_BUILD_ROOT/usr/local/nginx
61
62 %files # 定义打包文件列表
63 %license LICENSE # 指定软件的许可证文件通常是 /usr/share/licenses/nginx
64 %doc README # 指定软件的文档文件, 通常是 /usr/share/doc/nginx
65 /usr/local/nginx/* # 将整个/usr/local/nginx 目录及其所有内容打包到RPM包当中
66 %config(noreplace) /usr/local/nginx/conf/nginx.conf # 将nginx.conf 标记为配置文件, 且
67     在软件升级时不会覆盖用户的修改版本。
68
69
70 rpmbuild -ba /root/rpmbuild/SPECS/nginx.spec # 基于编辑好的SPEC文件再次执行RPM构建
71
72 ls /root/rpmbuild/RPMS/x86_64/nginx-1.18.0-1.ky10.x86_64.rpm
73
74 mkdir /usr/lcoal/nginx/html/Mynginx # 创建/Mynginx目录
75 cp /root/rpmbuild/RPMS/x86_64/nginx-1.18.0-1.ky10.x86_64.rpm
76 /usr/lcoal/nginx/html/Mynginx #将构建好的RPM包复制到 /Mynginx
77
78 createrepo /usr/lcoal/nginx/html/Mynginx # 构建自定义YUM仓库
79 vim /usr/lcoal/nginx/conf/nginx.conf # 修改nginx配置文件
80 server {
81     listen 80; # 监听 80 端口, HTTP 默认端口
82     server_name localhost; # 服务器名称, 指向 localhost (本地服务器)
83
84     location /Mynginx/ { # 定义一个访问路径 `/Mynginx/` 的 location
85         autoindex on; # 启用目录浏览功能, 列
86        出/usr/local/nginx/html/Mynginx/目录下的文件
87     }
88 }
```

```

82         alias /usr/local/nginx/html/Mynginx/; # 将请求路径 `/Mynginx/` 映射到实际文件
83         系统路径 `/usr/local/nginx/html/Mynginx/
84     }
85 }
86 curl http://127.0.0.1/Mynginx/          #验证是否能访问到目录下内容
87
88 # 编辑YUM配置文件
89 vim /etc/yum.repos.d/nginx.repo
90 [Mynginx]
91 name=Mynginx
92 baseurl=http://127.0.0.1/Mynginx/
93 enable=1
94
95 # 验证YUM仓库
96 yum repolist -v

```

3.容器docker

第一章 认知docker

1. 初识docker

1.1 什么是容器？

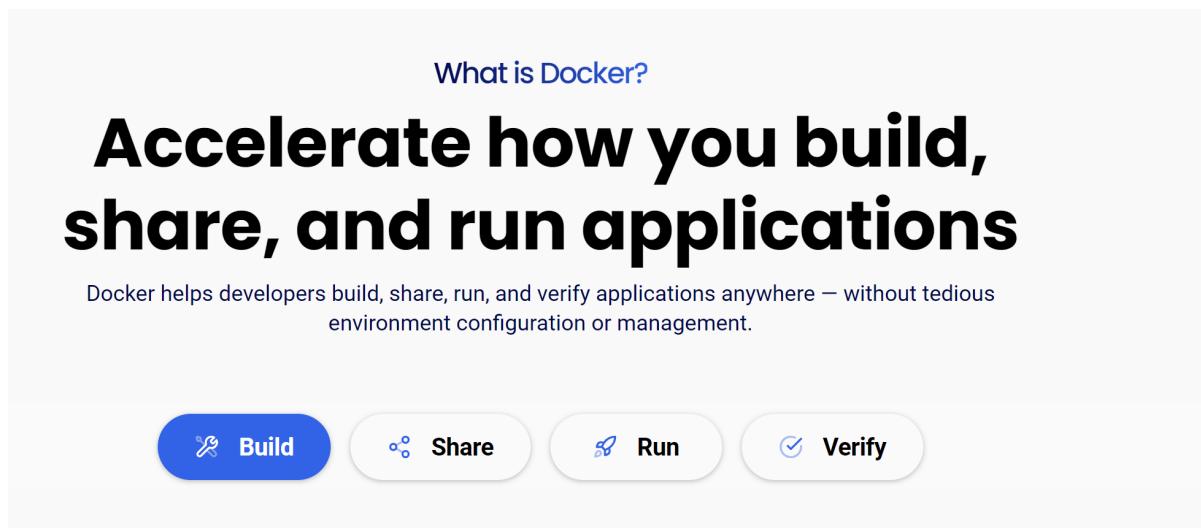
容器是一种工具，容器就好比是容纳物品的箱子。我们可以在这个箱子里面放入一些物品，同时也可用来存纳物品和帮助我们运输物品等，它也像我们生活当中的行李箱、书包等等一些列。

容器技术是虚拟化、云计算、大数据之后的一门新技术，容器技术提升了硬件资源利用率，方便了企业的业务横向扩容，实现了业务宕机自愈功能，因此未来数年会是一个容器愈发流行的时代，对于IT行业的从业者，熟练掌握容器技术无疑是给自己的前途和就业机会开辟了更加宽阔的道路。

1.2 docker 简介

docker是在2013年开源的应用程序是一个有GO语言编写的一个开源paas服务（平台即服务），GO语言是有google开发，docker是基于linux内核实现的。docker最早采用LXC技术（LXC是linux原生支持的容器技术，可以提供轻量级的虚拟化，可以说docker是基于LXC发展起来的）。

docker相比虚拟机的交付速度更快，资源消耗更低，docker采用客户端和服务端架构，使用远程API来管理和创建的docker容器，使其可以创建一个轻量级、可移植的，只给自足的容器。docker三大理念是build（构建）、share（共享）、run（运行）、docker遵从apache2.0协议（是一种开源软件许可证，由Apache软件基金会提供。它是一种自由、宽松的许可证，允许用户以商业和非商业用途自由使用、修改和重新分发受许可的软件。）



通过 (namespace) 来提供容器的资源隔离和 (cgroup) 限制容器使用一定数量硬件功能等一系列安全保障。所以docke容器在运行是不需类似虚拟机（空运行的虚拟机占用物理机的一定性能开销）额外的资源开销问题，docker可以大幅度提高服务器资源利用率，可以理解docker是一种新颖方式实现轻量级的虚拟机，类似VM但在原理和应用上他俩差别还是很大的，并且docker专业叫法为应用型容器。

1.3 docker组成

主机 (host)：一个物理机或虚拟机，用于运行docker服务进程和容器。

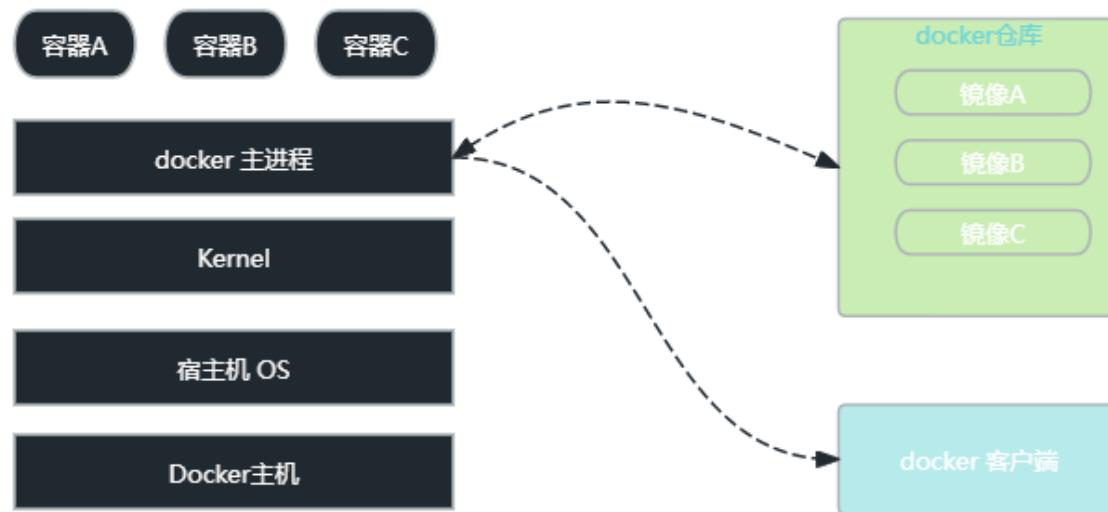
服务端 (server)：docker守护进程 (systemd服务)，运行docker容器

客户端 (Client)：客户端使用docker命令或其它工具调用dockerAPI。

仓库 (registry)：存储镜像的仓库是一种集中管理和存储镜像文件的系统类似YUM仓库。

镜像 (images)：镜像可以理解为一个轻量级、独立的软件包，其中包含了应用程序运行所需的所有环境。

容器 (Container)：容器时从镜像生成对外提供服务的一个或多组服务，通常是一个容器一个服务。



docker 生成镜像原理： docker客户端(client) 执行 build、 pull、 run等操作通过API调用服务端(server)通过网络从镜像仓库(registry)中拉取镜像 (images) 下载到本地服务器，在通过镜像 (images) 生成容器 (container) 。

1.4 docker和虚拟机对比

资源利用率：

Docker： Docker利用容器化技术，在宿主操作系统上共享内核，因此每个容器只需占用少量的系统资源，如内存和磁盘空间。

虚拟机： 虚拟机需要模拟完整的操作系统，并独立分配一定的资源给每个虚拟机，因此需要更多的系统资源。

启动时间：

Docker： 由于容器共享宿主操作系统的内核，因此启动Docker容器的时间通常比启动虚拟机快得多。

虚拟机： 虚拟机需要启动独立的操作系统内核，因此启动时间通常较长。

隔离性：

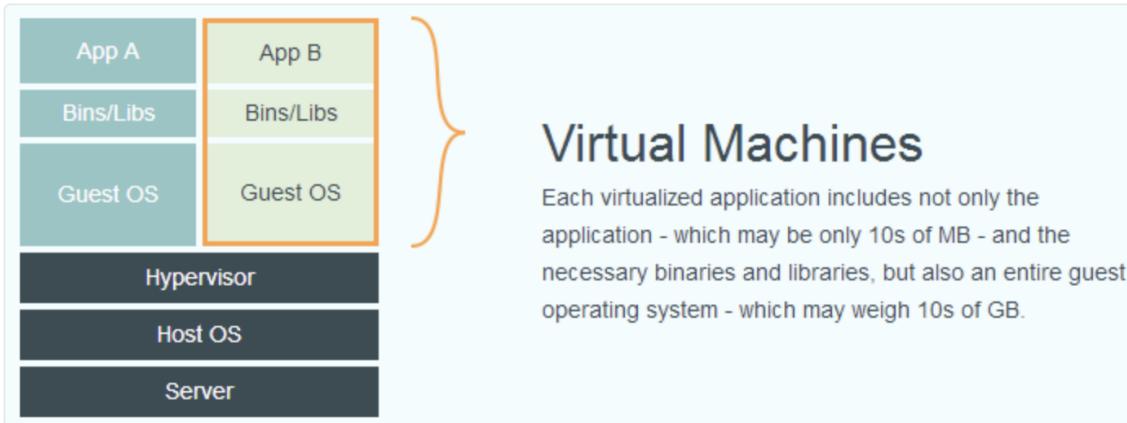
Docker： Docker容器提供了一定程度的隔离性，但容器共享宿主操作系统的内核，因此容器之间的隔离性相对较弱。

虚拟机： 虚拟机是完全隔离的，每个虚拟机都拥有自己的操作系统内核和独立的资源分配，因此具有更强的隔离性。

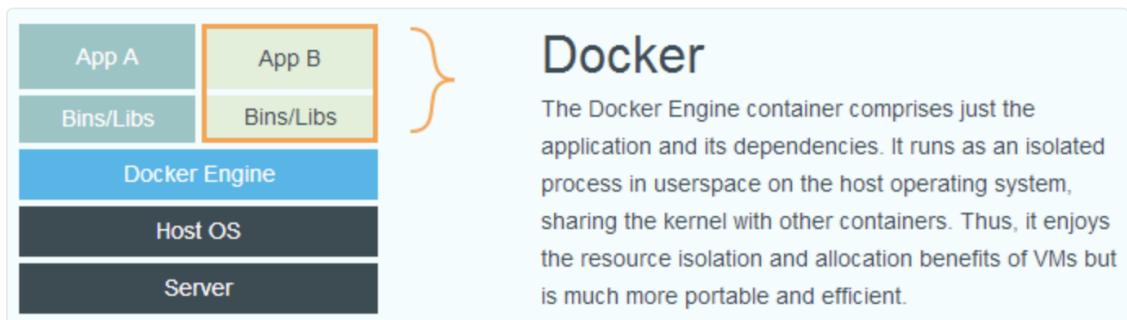
跨平台性：

Docker: Docker容器可以在几乎任何操作系统上运行，包括Windows、Mac和Linux。

虚拟机：虚拟机通常需要特定的虚拟化软件来运行，并且在不同的操作系统上可能有不同的版本。



传统虚拟化



Docker

1.5 Namespace技术：

namespace 是linux系统底层概念，在装完操作系统之后直接在内核层实现，用于在相同系统上运行的进程之间创建独立的资源实例。这些资源包括进程ID、网络、文件系统、挂载点等。每种命名空间都为其内部的进程提供了一个独立的视图，使得它们可以在不同的命名空间中享有相互独立的资源。

例如：想象一所大学，这所大学有许多不同的系（命名空间）。每个系都有自己的学生、教授和教室，它们之间相互独立，不会相互干扰。例如，计算机系的学生只能在计算机系的教室上课，而不能进入其他系的教室。同样，化学系的学生只能在化学系的教室上课。

命名空间技术通常被用于容器化技术中，如Docker和Kubernetes，以实现容器之间的隔离，目前主要通过一下技术实现容器运行空间的相互隔离：

隔离系统	功能	系统调用	内核版本
MNT Namespace	提供磁盘挂着点和文件系统隔离	CLONE_NEWNS	linux 2.4.19
IPC Namespace	提供进程间通信的隔离	CLONE_NEWIPC	linux 2.6.19
UTS Namespace	提供主机名隔离	CLONE_NEWUTS	linux 2.6.19
PID Namespace	提供进程隔离	CLONE_NEWPID	linux 2.6.24
NET Namespace	提供 网络隔离	CLONE_NEWWNET	linux 2.6.29
User Namespace	提供 用户隔离	CLONE_NEWUSER	linux 3.8

1.5.1 MNT Namespace

每个容器都要有独立的根文件系统有独立的用户空间，以实现在容器里面启动服务并且使用容器的运行环境，即一个宿主机是ubuntu的服务器，可以在里面启动一个centos运行环境的容器并且在容器里面启动一个Nginx服务，此Nginx运行时使用的运行环境就是centos系统目录的运行环境，但是在容器里面是不能访问宿主机的资源，宿主机是使用了chroot技术把容器锁定到一个指定的运行目录里面。

这其中的原因就是宿主机是使用了chroot技术把容器锁定到一个指定的运行目录，这意味着在宿主机上运行的容器进程被限制在一个特定的根目录中，而无法访问宿主机上的其他目录和文件系统。。

例如：/var/lib/containerd/io.containerd.runtime.v2.task/moby/容器ID

启动一个容器验证：

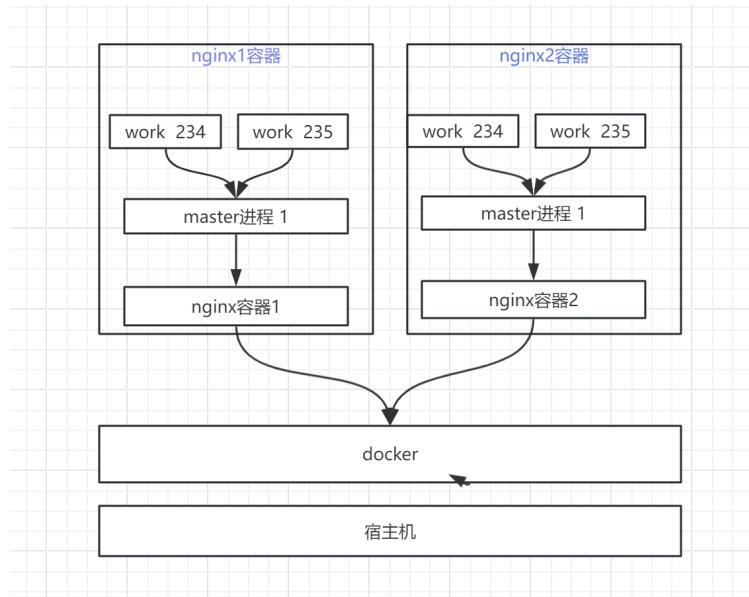
```
1 docker load -i /opt/exam/kylin/docker/storehouse/nginx.tar.gz
2
3 docker run -itd --name nginx01 -p 80:80 nginx:latest
4 docker exec -it nginx01 bash
```

查看容器基于什么系统打包

```
1 root@4857f84ef140:/#cat /etc/kylin-release
2 Kylin Linux Advanced Server release V10 (Tercel)
3 # 安装系统基础命令
4 yum repolist -v
5 yum -y install procps (top命令)
6 yum -y install net-tools (网络工具)
7 验证容器根文件系统
8 root@4857f84ef140:/# ls
9 bin boot dev docker-entrypoint.d docker-entrypoint.sh etc home lib lib64
media
10 mnt opt proc root run sbin srv sys tmp usr var
```

1.5.2 IPC Namespace:

IPC Namespace 的作用是在容器内部创建一个独立的通信环境，使得容器内的进程可以相互通信，但是不能直接访问其他容器内的进程，确保了容器之间的隔离性。



1.5.3 UST Namespace:

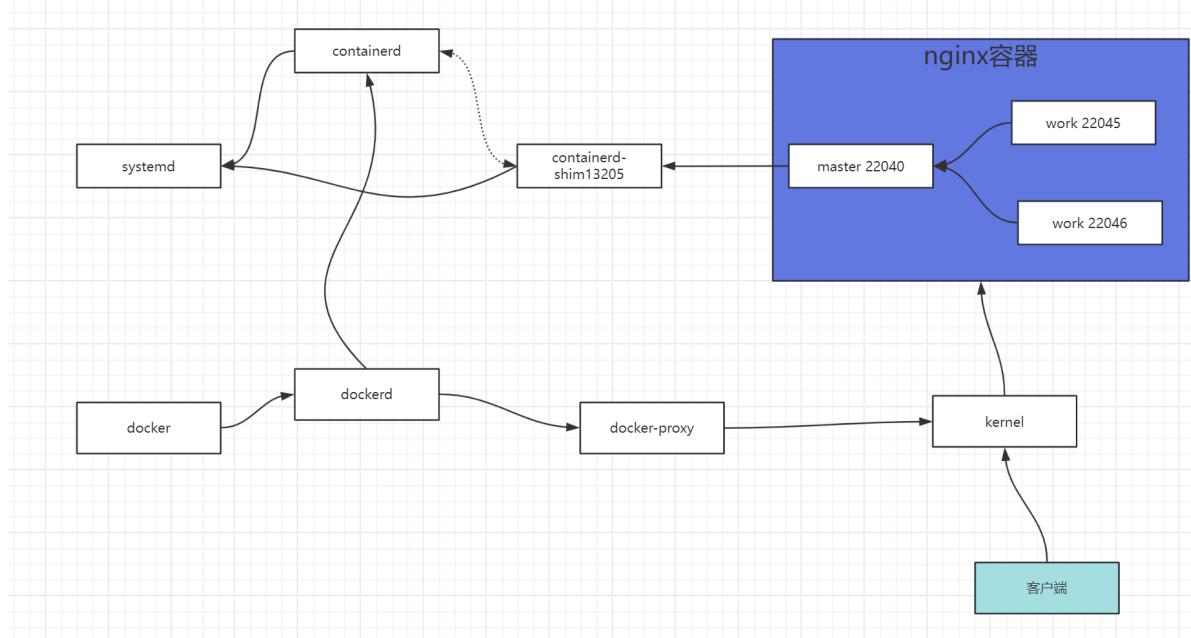
UST Namespace 就像是容器的个人身份证，它包含容器的个人信息，比如容器的名字（主机名）和地址（域名）。这些信息只在容器内部有效，就像你的身份证信息只在你自己的身份证上有效一样。

`cat /etc/issue` 显示了容器的发行版信息。

`uname -a` 显示了容器的内核信息和其他系统信息。

1.5.4 PID Namespace

Docker等容器技术使用 **PID Namespace** 来实现进程隔离，使容器内的进程与宿主机和其他容器的进程相互隔离。



1.5.5 Net Namespace:

每一个容器都类似于虚拟机一样有自己的网卡，监听端口，TCP/IP协议等，docker使用 Network namespace会把它分配一个虚拟的网络 (vethX) 接口，就好像给它插了一张虚拟的网卡。这个虚拟网卡连接到容器内部的网络空间，让容器能够与外部世界通信。

而`docker0`则相当于一根神奇的网线，它连接着宿主机和所有的容器，就像家里的路由器一样。它负责在不同的网络之间传输数据，确保容器之间和宿主机之间能够互相沟通。而它所处的位置，就像是整个网络中的交通枢纽一样，负责处理数据的转发和传输。

注意：需要把 IPv4 转发功能打开。

使用命令：`echo net.ipv4.ip_forward=1 >> /etc/sysctl.conf`

使用 `sysctl -p` 命令重新加载 `/etc/sysctl.conf` 文件，以使更改生效。

使用 `sysctl net.ipv4.ip_forward` 命令来确认该功能是否已经打开。

安装：`yum -y install net-tools` (网络工具)

使用 `ifconfig` 命令查看宿主机网卡信息。

```

ens19: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      ether c6:24:5f:bd:26:33 txqueuelen 1000 (Ethernet)
        RX packets 27604 bytes 9557930 (9.1 MiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        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 366797 bytes 55810130 (53.2 MiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 366797 bytes 55810130 (53.2 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth4ad888: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      ↗
      inet6 fe80::6030:11ff:fe55:3b03 prefixlen 64 scopeid 0x20<link>
      ether 62:30:11:55:3b:03 txqueuelen 0 (Ethernet)
        RX packets 2950 bytes 205063 (200.2 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 3742 bytes 10572620 (10.0 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

使用 `arp -a` 命令把ip转换为MAC地址。

安装: `yum -y install bridge-utils` (查卡网卡桥接)

使用 `brctl show` 命令查看宿主机网桥连接设备。

```
[root@localhost ~]# brctl show
bridge name     bridge id          STP enabled    interfaces
docker0         8000.0242e36ca5ea   no           vethe4ad888
```

也可以使用 `iptables -t nat -vnL` 从宿主机上查看iptables规则。

Chain POSTROUTING (policy ACCEPT 32 packets, 2193 bytes)						
pkts	bytes	target	prot	opt	in	out
0	0	MASQUERADE	all	--	*	!docker0 172.17.0.0/16 0.0.0.0/0
0	0	MASQUERADE	tcp	--	*	172.17.0.2 172.17.0.2 tcp dpt:80
0	0	MASQUERADE	tcp	--	*	172.17.0.3 172.17.0.3 tcp dpt:80

Chain DOCKER (2 references)						
pkts	bytes	target	prot	opt	in	out
0	0	RETURN	all	--	docker0	*
1	60	DNAT	tcp	--	!docker0	*
1	60	DNAT	tcp	--	!docker0	*

1.5.6 User namespace

各个容器内可能会出现重名的用户和用户组名称，或重复的用户UID或GID，那么通过什么方式来隔离呢？

User namespace 运行在各个宿主机的各个容器空间内创建相同的用户名以及相同的用户UID和GID，只是会把用户的作用范围限制在每个容器内，即A用户和B用户可以用于相同的用户名和ID账户，但是此用户的有效范围仅是当前的容器内，不能访问另一个容器内的文件系统，即相互隔离，互不影响。

1.6 资源限制

在一个容器，如果不对其做限制资源限制，则宿主机会运行其占用无限的内存空间，有时候因为代码BUG程序会一直申请内存，直到把宿主机内存占完，为了避免此类的问题出现，宿主机有必要对容器进行资源限制，例如CPU，内存等等，cgroups主要的作用，就是限制一个进程能够使用的资源上限，包括cpu，内存，磁盘，网络带宽等等。此外，还能够对进程进行优先级设置，以及将进程挂起和恢复等操作。

1.6.1 验证系统cgroups

cgroups在内核层默认已经开启，无需额外的配置或编译选项。

查看当前系统的 Red Hat Linux 版本信息。

```
1 [root@k8s-node1 ~]# cat /etc/redhat-release
2 CentOS Linux release 7.9.2009 (Core)
```

用于显示当前系统内核的版本号。

```
1 [root@k8s-node1 ~]# uname -r
2 3.10.0-1160.el7.x86_64
```

查看位于 `/boot/config-3.10.0-1160.el7.x86_64` 文件中是否包含关于 cgroup 的配置。

```
1 [root@k8s-node1 ~]# grep -i cgroup /boot/config-3.10.0-1160.el7.x86_64
2 CONFIG_CGROUPS=y
3 # CONFIG_CGROUP_DEBUG is not set
4 CONFIG_CGROUP_FREEZER=y
5 CONFIG_CGROUP_PIDS=y
6 CONFIG_CGROUP_DEVICE=y
7 CONFIG_CGROUP_CPUACCT=y
8 CONFIG_CGROUP_HUGETLB=y
9 CONFIG_CGROUP_PERF=y
10 CONFIG_CGROUP_SCHED=y
11 CONFIG_BLK_CGROUP=y
12 # CONFIG_DEBUG_BLK_CGROUP is not set
13 CONFIG_NETFILTER_XT_MATCH_CGROUP=m
14 CONFIG_NET_CLS_CGROUP=y
15 CONFIG_NETPRIORITY_CGROUP=y
```

查看cgroups中内核配置中与内存 (MEM) 和 cgroup (CG) 相关的选项。

```
1 [root@k8s-node1 ~]# cat /boot/config-3.10.0-1160.el7.x86_64 | grep MEM | grep CG
2 CONFIG_MEMCG=y
3 CONFIG_MEMCG_SWAP=y
4 CONFIG_MEMCG_SWAP_ENABLED=y
5 CONFIG_MEMCG_KMEM=y
```

1.6.2 查看系统cgroups

```
1 [root@k8s-node1 ~]# ll /sys/fs/cgroup/
2 总用量 0
3 drwxr-xr-x 6 root root 0 2月 29 10:52 blkio
4 lrxwxrwxrwx 1 root root 11 2月 29 10:52 cpu -> cpu,cpuacct
5 lrxwxrwxrwx 1 root root 11 2月 29 10:52 cpuacct -> cpu,cpuacct
6 drwxr-xr-x 6 root root 0 2月 29 10:52 cpu,cpuacct
7 drwxr-xr-x 3 root root 0 2月 29 10:52 cpuset
8 drwxr-xr-x 6 root root 0 2月 29 10:52 devices
9 drwxr-xr-x 3 root root 0 2月 29 10:52 freezer
10 drwxr-xr-x 3 root root 0 2月 29 10:52 hugetlb
11 drwxr-xr-x 6 root root 0 2月 29 10:52 memory
12 lrxwxrwxrwx 1 root root 16 2月 29 10:52 net_cls -> net_cls,net_prio
13 drwxr-xr-x 3 root root 0 2月 29 10:52 net_cls,net_prio
14 lrxwxrwxrwx 1 root root 16 2月 29 10:52 net_prio -> net_cls,net_prio
15 drwxr-xr-x 3 root root 0 2月 29 10:52 perf_event
16 drwxr-xr-x 6 root root 0 2月 29 10:52 pids
```

```
17 drwxr-xr-x 6 root root 0 2月 29 10:52 systemd
```

这些是在 `/sys/fs/cgroup/` 目录下的子目录，它们代表了 `cgroups` 的各个子系统，每个子目录都对应一个 `cgroup` 子系统。这些子系统包括：

1. **blkio**: 磁盘块 I/O 子系统，用于限制和控制块设备的 I/O。
2. **cpu, cpuacct**: CPU 子系统，用于限制和控制 CPU 的使用。`cpuacct` 子系统用于记录 CPU 使用情况。
3. **cpuset**: CPU 亲和性子系统，用于将进程绑定到特定的 CPU 核心。
4. **devices**: 设备子系统，用于限制和控制进程对设备的访问。
5. **freezer**: 冻结子系统，用于冻结和恢复进程的状态。
6. **hugetlb**: 大页内存子系统，用于控制大页内存的分配和使用。
7. **memory**: 内存子系统，用于限制和控制内存的使用。
8. **net_cls, net_prio**: 网络类和网络优先级子系统，用于限制和控制进程的网络流量。
9. **perf_event**: 性能事件子系统，用于管理进程的性能事件。
10. **pids**: 进程 ID 子系统，用于限制和控制进程的数量。
11. **systemd**: `systemd` 子系统，用于与 `systemd` 进行集成和管理。

这些子目录中包含了与各个 `cgroup` 子系统相关的控制文件和配置文件，通过修改这些文件，可以对相应的 `cgroup` 进行资源控制和管理。

1.6.3 总结：

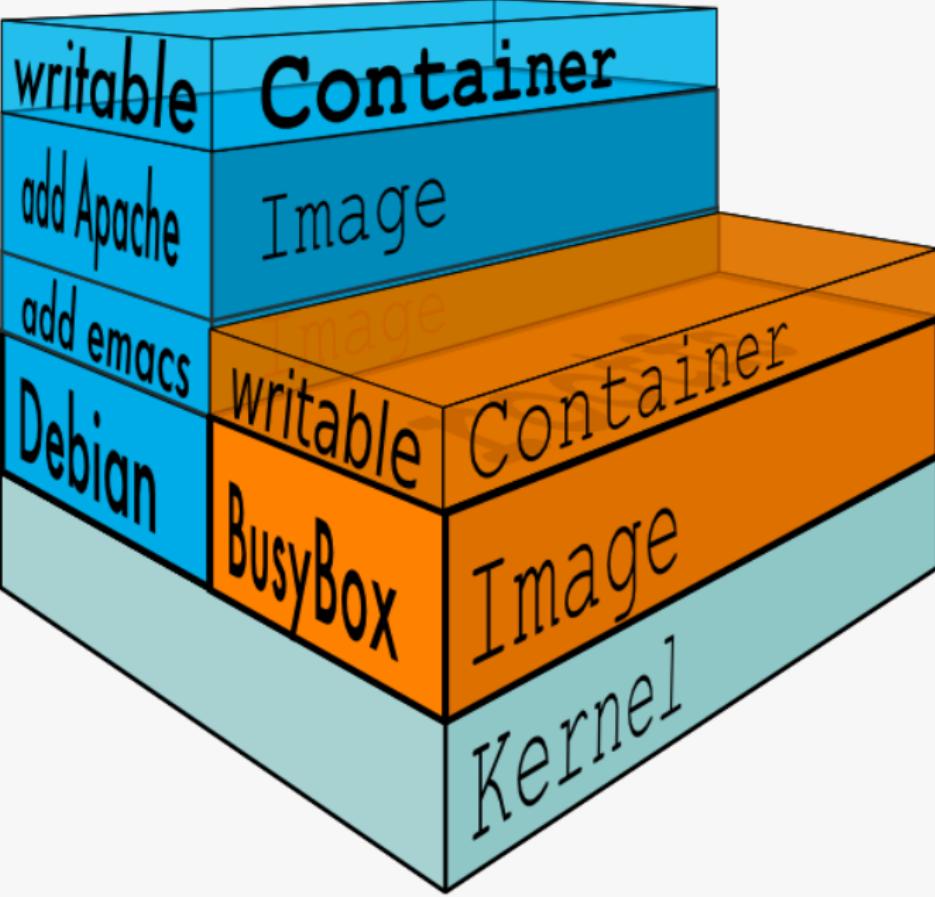
有了以上的chroot、namespace、groups 是容器实现的核心技术，它们共同构成了容器的隔离环境和资源管理机制，使得容器可以在一个独立的、隔离的环境中运行，并实现对资源的精细化控制和管理。但是还需要有相应的容器创建与删除的管理工具，以及怎样把容器运行起来，容器数据怎么处理，怎么进行启动与关闭等问题需要解决，于是容器管理技术出现了。

2. 容器的管理工具

2.1 docker

Docker 启动一个容器需要镜像，docker的镜像可以保存在一个公共的地方共享使用，只要把镜像下载来就可以使用，最主要的可以在镜像基础之上做自定义配置并且可以再把其提交为一个镜像，一个镜像可以被启动为多个容器。

Docker的镜像是分层的，镜像底层为库文件且只读层既不能写入也不能删除数据，从镜像加载启动为一个容器后会生成一个可写层，其写入的数据会复制到容器目录，但是容器的数据在删除容器后也会被随之删除。



我们可以将 Docker 镜像的分层比喻为一种类似于建筑结构的层次化组织。这种组织方式采用了一种称为"Union File System"的技术，类似于在建筑物中堆叠不同的楼层。

1. **基础层 (Base Layer)**：就像建筑物的地基一样，基础层是 Docker 镜像的最底层。这一层包含了操作系统和基本的运行时库，类似于建筑物的地基和基础结构。
2. **第二层 (Second Layer)**：在基础层之上，我们可以添加额外的层次，就像在建筑物中添加了更多的楼层一样。第二层包含了在基础层之上安装的软件包、依赖项和其他文件，使得镜像具备了特定的功能或应用。
3. **第三层 (Third Layer)**：如果需要进一步定制镜像，我们可以在第二层之上添加更多的层次。第三层可能包含了应用程序的代码、配置文件等。每一层都是基于前一层构建的，但它们是相互独立的，并且是只读的，这意味着修改一个层不会影响到其他层。

通过这种层次化的组织方式，Docker 镜像可以有效地管理文件系统。如果对容器进行了修改，Docker 会在新的层次中保存这些更改，而不会影响到底层的镜像，这样就能够轻松地创建、共享和更新容器，并且节省存储空间。

2.2 docker 优势

轻量快速：Docker 容器启动迅速，占用资源少，使得应用程序部署更加高效。

跨平台：Docker 容器可以在各种不同的操作系统和云平台上运行，保持应用程序在不同环境中的一致性。

简化部署：通过 Docker，可以简化应用程序的打包、交付和部署流程，提高开发和运维效率。

资源利用率高：由于 Docker 容器共享主机内核，并且使用轻量级的隔离机制，因此可以更有效地利用系统资源，提高了服务器的利用率。

环境一致性：Docker 容器打包了应用程序及其依赖项，确保在不同环境中的一致性和可移植性。

持续集成和持续交付：Docker 可与 CI/CD 工具集成，实现自动化构建、测试和部署，加速软件开发和发布过程。

2.3 docker 的缺点

隔离性：在 Docker 中，尽管容器之间存在一定程度的隔离，但与传统虚拟机相比，其隔离性并不是完全彻底的。这主要是因为 Docker 容器共享主机的操作系统内核，而虚拟机则是在独立的虚拟硬件上运行完整的操作系统。

3. 容器规范

在2015年6月，一些主要的科技公司联合成立了 Open Container Initiative (OCI) 组织，其主要目的是制定容器技术的标准，以促进容器生态系统的发展和互操作性。

OCI 组织成立后，发布了两个主要的规范：

1. **Runtime Specification** (运行时规范)：这个规范定义了容器运行时的标准接口和行为，包括容器的生命周期管理、文件系统和网络的隔离、进程和资源管理等。该规范旨在确保不同容器运行时之间的兼容性，使得容器可以在不同的运行时环境中无缝运行。
2. **Image Format Specification** (镜像格式规范)：这个规范定义了容器镜像的标准格式和结构，包括镜像的元数据、文件系统层和配置信息等。该规范旨在实现容器镜像在不同平台和环境中的互操作性，使得容器镜像可以在不同的容器运行时中被正确地解析和执行。

这两个规范为容器技术的发展和应用提供了基础，使得不同厂商和开发者能够共同遵循统一的标准，不同的容器公司开发的容器只要兼容这两个规范，就可以保证容器的可移植性和相互操作性。

4. runtime

容器的"runtime"指的是容器在运行时所使用的容器引擎或容器运行时环境。它负责管理容器的生命周期、创建和运行容器、以及提供容器隔离和资源管理等功能。容器的运行时环境通常包括容器引擎本身以及相关的库、工具和配置。

目前主流的容器运行时包括：

1. **Docker Runtime**：Docker 是最流行的容器引擎之一，它提供了完整的容器生态系统，包括容器运行时、镜像管理、网络和存储等功能。Docker 的容器运行时是基于 runc (以前是基于 Docker 自己的容器运行时 libcontainer) 实现的。
2. **containerd**：containerd 是一个面向容器运行时的核心组件，它由 Docker 开源的一部分，用于管理容器的生命周期、镜像和容器的存储、网络等。containerd 可以作为一个独立的容器运行时，也可以作为其他容器平台和工具的基础组件。
3. **cri-o**：cri-o 是一个轻量级的容器运行时，专门针对 Kubernetes 平台而设计，采用了 OCI 规范，以 containerd 为基础，并提供了 Kubernetes CRI (Container Runtime Interface) 的实现。
4. **rkt (Rocket)**：rkt 是由 CoreOS 开发的另一个容器引擎，旨在提供更简洁、安全和模块化的容器运行时环境。rkt 支持 OCI 规范，并提供了一种与 Docker 兼容的容器格式。

这些容器运行时环境都是根据 OCI 规范实现的，因此它们之间具有一定的互操作性和兼容性，可以在不同的容器平台和工具之间共享和使用容器镜像。

runtime主要定义了以下规范，并以json格式保存在/run/docker/runtime-runc/moby/容器ID/state.json

注意！ 后期可以通过监控软件来结合这个配置文件做到对容器的状态监控，此文件存储了 Docker 运行时（通常是 runc）管理的特定容器的状态信息。

```
1 yum -y install jq 按照jq工具
2 # 查看此文件并以json格式输出
3 cat /run/docker/runtime-runc/moby/容器ID/state.json | jq .
4 # 内容如下:
5 version: 版本信息, 表示 JSON 数据的版本号。
6 container_id: 容器ID, 表示容器的唯一标识符。
7 pid: PID, 表示容器的进程ID。
8 container_directory: 容器文件目录, 表示容器在文件系统中的路径。
9 container_created_at: 容器创建时间, 表示容器的创建时间戳。
10 container.lifecycle: 容器的生命周期, 表示容器当前的状态, 例如 running、paused、stopped 等。
```

5. 容器镜像 (image)

OCI Image Specification 的一部分, 它定义了容器镜像的底层存储格式和布局, 包括镜像的索引、图层和配置等。镜像格式规范为容器镜像提供了一个统一的存储结构, 以便不同的容器工具和平台可以正确地读取和处理容器镜像。

```
1 # 使用命令
2 docker inspect 镜像
3 # 内容如下
4 镜像元数据: 包括了容器镜像的基本信息, 比如名称、版本、创建者等。
5 文件系统层: 容器镜像由多个文件系统层组成, 每个层包含了容器运行所需的文件和目录。
6 容器配置: 定义了容器运行时的配置信息, 比如启动命令、环境变量、网络设置等。
7 镜像索引: 对于多架构或多平台的镜像, 可以使用镜像索引来管理不同版本。
8 签名和验证信息: 为了确保镜像的完整性和安全性, 可以对镜像进行数字签名。
```

6. 容器的管理工具

runc管理工具是docker engine, Docker engine 是一个典型的容器管理工具, 它包含了多个组件, 包括后台守护进程 (Daemon) 、客户端命令行接口 (CLI) 等。Docker Engine 的后台守护进程负责管理容器的创建、运行和销毁, 而客户端命令行接口提供了用户与 Docker Engine 交互的方式。大家经常提到的docker就是只docker engine。

1. **客户端 (Client)** : Docker 客户端是与用户交互的界面, 它提供了一组命令行工具, 用户可以通过这些命令行工具与 Docker Engine 进行交互。客户端负责接收用户的命令, 并将这些命令发送给 Docker Engine 执行。这些命令可以包括镜像操作、容器操作、网络操作等。
2. **服务端 (Daemon)** : Docker 服务端是运行在后台的守护进程, 也称为 Docker 守护进程 (Docker Daemon) 。它负责管理容器的生命周期、镜像的构建和存储、网络的管理等。Docker 服务端接收来自客户端的命令, 执行相应的操作, 并返回结果给客户端。

7. 容器定义工具

容器定义工具允许用户定义容器的属性核内容, 以方便容器能够被保存, 共享和重建。

Docker Image: Docker 镜像是一个只读的模板, 用于创建容器。它包含了运行容器所需的所有文件和配置, 包括基础操作系统、应用程序、库文件、环境变量等。Docker 镜像可以从 Dockerfile 构建而来, 也可以从 Docker Hub 或其他镜像仓库中拉取。

Dockerfile: Dockerfile 是一个文本文件, 用于定义 Docker 镜像的构建过程。它包含了一系列的指令, 用来描述如何构建镜像, 包括基础镜像、软件包安装、文件复制、环境变量设置等操作。通过编写 Dockerfile, 用户可以自定义镜像, 以满足特定的应用需求。

容器定义工具是用来定义容器镜像的结构和内容的工具。它们允许用户按照一定的格式和规范来描述容器镜像, 以便后续能够使用这些定义来构建和管理容器。除了 Dockerfile 外, 还有其他一些工具可以用来定义容器, 比如 Podman Podfile、Kubernetes YAML 文件、这些工具提供了一种灵活和标准化的方式来描述容器镜像, 使得用户能够方便地创建、共享和管理容器镜像。

8. 镜像仓库

镜像仓库 (Image Repository) 是用于存储和管理容器镜像的地方。它提供了一个中心化的存储库，使得用户可以方便地上传、下载和共享容器镜像。镜像仓库通常提供了一些基本的功能，包括镜像的存储、版本管理、权限控制等。

image registry: docker 官方提供的私有仓库部署工具。

docker hub: docker官方的公共仓库，已经保存了大量的常用镜像。

harbor: vmware 提供的自带web界面上带认证功能的镜像仓库。

9. docker (容器) 的依赖技术

1. 容器网络

docker自带的网络docker network仅支持管理单机上的容器网络，当多台主机运行时需要使用第三方开源网络，例如 calico、flannel、traefik等

2. 服务发现

容器的动态扩容特性决定了容器IP也会随之变化，因此需要有一种机制可以自动识别将用户请求动态转发到新创建的容器上，Kubernetes自带服务发现功能，需要结合 kube-dns服务解析内部域名。

3. 容器监控

可以通过原生命令docke ps/top/stats 查看容器运行状态，另外也可以使用heapster/Prometheus 等第三方监控容器的运行状态。

4. 数据管理

容器的动态迁移会导致其在不同的host之间迁移，因此如何保证与容器相关的数据也能随之迁移或随时访问，可以使用逻辑卷/存储挂载等方式解决。

5. 日志收集

docker 原生的日志查看工具 `docker logs`，但是容器内容的日志需要同ELK等专门的日志收集分析和展示工具进行处理。

第二章 安装docker及基础命令使用

1. 系统版本选择

[官方网站](#) docker支持多种操作系统安装运行，比如 ubuntu、centos、redhat、debian、等等，甚至还支持 windows和MAC，在linux系统上需要内核版本在3.10 或以上。查看内核版本命令 `hostnamectl`。

docker之前没有区分版本，但是2017年处推出将docker更名为新的项目Moby, [github地址](#)，moby项目属于Docker项目的全新上游，docker 将属于Moby的子产品，而之后的版本开始区分为CE版本（社区版本）和EE版本（企业收费版本），CE社区版本和EE企业版本都是每一个季度发布一个新的版本，但是EE版本提供后期安全维护1年，而CE版本是4个月。

2. docker 安装

2.1 下载 RPM 包安装

[阿里云镜像下载地址](#)

2.2 二进制下载地址:

[docker-ce-linux-static-stable-x86_64安装包下载 开源镜像站-阿里云\(aliyun.com\)](https://mirrors.aliyun.com/docker-ce/linux/static/stable/x86_64/docker-24.0.2.tgz)

```
1 # 下载docker二进程包
2 [root@localhost ~]# wget https://mirrors.aliyun.com/docker-
ce/linux/static/stable/x86_64/docker-24.0.2.tgz
3
4 # 解压tar包
5 [root@localhost ~]# tar -xf docker-24.0.2.tgz
6
7 # 将解压出的docker/bin目录下的可执行文件复制到/usr/local/bin/
8 [root@localhost ~]# cp ./docker/* /usr/local/bin/
```

将docker注册为系统服务 dockers.service

```
1 [root@localhost ~]# vim /usr/lib/systemd/system/docker.service
2
3 [Unit]
4 Description=Docker Application Container Engine
5 Documentation=https://docs.docker.com
6 BindsTo=containerd.service
7 After=network-online.target firewalld.service containerd.service
8 Wants=network-online.target
9 Requires=docker.socket
10
11 [Service]
12 Type=notify
13 # the default is not to use systemd for cgroups because the delegate issues still
14 # exists and systemd currently does not support the cgroup feature set required
15 # for containers run by docker
16 ExecStart=/usr/local/bin/dockerd -H fd:// --
17 containerd=/run/containerd/containerd.sock
18 ExecReload=/bin/kill -s HUP $MAINPID
19 TimeoutSec=0
20 RestartSec=2
21 Restart=always
22
23 # Note that StartLimit* options were moved from "Service" to "Unit" in systemd 229.
24 # Both the old, and new location are accepted by systemd 229 and up, so using the
25 # old location
26 # to make them work for either version of systemd.
27 StartLimitBurst=3
28
29 # Note that StartLimitInterval was renamed to StartLimitIntervalSec in systemd 230.
30 # Both the old, and new name are accepted by systemd 230 and up, so using the old
31 # name to make
32 # this option work for either version of systemd.
33 StartLimitIntervalSec=60s
34
35 # Having non-zero Limit*s causes performance problems due to accounting overhead
36 # in the kernel. we recommend using cgroups to do container-local accounting.
37 LimitNOFILE=infinity
38 LimitNPROC=infinity
39 LimitCORE=infinity
40
41 # Comment TasksMax if your systemd version does not support it.
42 # only systemd 226 and above support this option.
43 TasksMax=infinity
```

```

41 # set delegate yes so that systemd does not reset the cgroups of docker containers
42 Delegate=yes
43
44 # kill only the docker process, not all processes in the cgroup
45 KillMode=process
46
47 [Install]
48 WantedBy=multi-user.target
49
50 # 为 docker.service添加可执行权限
51 chmod +x /usr/lib/systemd/system/docker.service

```

定义 Docker 守护进程的通信接口

```

1 [root@localhost ~]# vim /usr/lib/systemd/system/docker.socket
2
3 [Unit]
4 Description=Docker Socket for the API
5 PartOf=docker.service
6
7 [Socket]
8 ListenStream=/var/run/docker.sock
9 SocketMode=0660
10 SocketUser=root
11 SocketGroup=docker
12
13 [Install]
14 WantedBy=sockets.target
15
[root@localhost ~]# chmod a+x /usr/lib/systemd/system/docker.socket

```

配置containerd.service文件

```

1 vim /lib/systemd/system/containerd.service
2
3 [Unit]
4 Description=containerd container runtime
5 Documentation=https://containerd.io
6 After=network.target local-fs.target
7
8 [Service]
9 ExecStartPre=-/usr/sbin/modprobe overlay
10 ExecStart=/usr/local/bin/containerd
11
12 Type=notify
13 Delegate=yes
14 KillMode=process
15 Restart=always
16 RestartSec=5
17 # Having non-zero Limit*s causes performance problems due to accounting overhead
18 # in the kernel. We recommend using cgroups to do container-local accounting.
19 LimitNPROC=infinity
20 LimitCORE=infinity
21 LimitNOFILE=infinity
22 # Comment TasksMax if your systemd version does not support it.
23 # only systemd 226 and above support this version.
24 TasksMax=infinity
25 OOMScoreAdjust=-999
26
27 [Install]

```

```
28 WantedBy=multi-user.target  
29  
30  
31 chmod a+x /lib/systemd/system/containerd.service
```

设置存储目录，设置私有镜像仓库地址。

```
1 vim /etc/docker/daemon.json  
2  
3 {  
4     "data-root": "/var/lib/docker",  
5     "storage-driver": "overlay2",  
6     "registry-mirrors": ["https://docker.aityp.com/"],  
7     "exec-opts": ["native.cgroupdriver=systemd"],  
8     "live-restore": false,  
9     "log-opt": {  
10         "max-file": "5",  
11         "max-size": "100m"  
12     }  
13 }  
14  
15 # 启动docker并设置开机自启  
16 [root@localhost ~]# groupadd docker  
17 [root@localhost ~]# systemctl daemon-reload  
18 [root@localhost ~]# systemctl start docker  
19  
20 # 查看docker版本  
21 [root@localhost ~]# docker version
```

2.3 使用YUM安装

```
yumdownloader --resolve [ 软件包名称 ] #下载指定软件包及其依赖项
```

```
1 wget -O /etc/yum.repos.d/CentOS-Base.repo https://mirrors.aliyun.com/repo/Centos-  
2 7.repo  
3 wget -O /etc/yum.repos.d/epel.repo https://mirrors.aliyun.com/repo/epel-7.repo  
4 sudo yum-config-manager --add-repo https://mirrors.aliyun.com/docker-  
5 ce/linux/centos/docker-ce.repo  
6  
7 yum repolist -v  
8 yum list docker-ce.x86_64 --showduplicates | sort -r #查询docker版本  
9 yum install docker-ce-24.0.0-1.el7 docker-ce-cli-26.1.0-1.el7  
10 # 安装docker-ce  
11  
12 systemctl enabled docker-ce --now  
13 systemctl enabled docker-ce-cli --now  
14 # 开启docker
```

2.4 验证docker版本

```
1 # 查看docker版本  
2 [root@docker-server04 ~]# docker version  
3 Client: Docker Engine - Community  
4 Version:          26.0.2  
5 API version:      1.45  
6 Go version:       go1.21.9
```

```
7  Git commit:      3c863ff
8  Built:          Thu Apr 18 16:30:00 2024
9  OS/Arch:        linux/amd64
10 Context:        default
11
12 Server: Docker Engine - Community
13 Engine:
14   Version:       26.0.2
15   API version:  1.45 (minimum version 1.24)
16   Go version:   go1.21.9
17   Git commit:   7cef0d9
18   Built:         Thu Apr 18 16:28:58 2024
19   OS/Arch:       linux/amd64
20   Experimental: false
21 containerd:
22   Version:       1.6.31
23   GitCommit:    e377cd56a71523140ca6ae87e30244719194a521
24 runc:
25   Version:       1.1.12
26   GitCommit:    v1.1.12-0-g51d5e94
27 docker-init:
28   Version:       0.19.0
29   GitCommit:    de40ad0
```

2.4 docker存储引擎

官方文档存储引擎的介绍

Docker存储引擎是指Docker用于管理镜像和容器数据的后端技术。在Docker中，有几种存储引擎可供选择，每种引擎都有其独特的特点和适用场景。以下是一些常见的Docker存储引擎：

1. **AUFS**是一种联合文件系统，最初由谢尔盖·佩特罗维奇于2006年开发。它被广泛用于Linux发行版和容器技术中，包括Docker。
 - 多层存储**: AUFS允许将多个文件系统层级联起来，形成一个统一的文件系统视图。这样可以在不同的文件系统层之间共享文件，提高存储效率和灵活性。
 - 写时复制**: AUFS使用写时复制 (copy-on-write) 技术，当对文件进行写操作时，它会在新的层中创建副本，而不是直接修改原始文件。这样可以提高性能和数据保护能力。
 - 快速启动**: 由于AUFS支持多层联合挂载，因此在创建和启动容器时可以快速加载镜像和文件系统层，加速容器的启动速度。
 - 分层管理**: AUFS将每个容器的文件系统分成多个层，可以独立管理和更新每个层，使得容器镜像的构建和管理更加灵活和高效。
2. **devicemapper**: 是一种基于块设备的存储驱动，可以使用LVM（逻辑卷管理器）或直接使用块设备作为后端存储。它支持写时复制 (copy-on-write) 技术，适用于需要高级存储功能的场景。需要注意！使用这个存储引擎会出现以下情况；使用Device Mapper作为Docker的存储驱动程序，Docker服务启动时会在/var/lib/docker/devicemapper/devicemapper/目录下创建一个100G大小的数据文件。这个数据文件用于存储所有容器的数据变更。

但当容器的数据变更超过了这个100G的限制时，会导致容器根目录变为只读，同时会限制每个容器最大为10GB。这个限制可能会对一些应用造成不便，特别是需要大量存储空间的应用。因此，在一些情况下，需要对Docker的存储驱动进行定制配置以满足特定需求。推荐还是使用overlay2引擎。
3. **overlay**: 是另一种基于overlay的存储驱动，与overlay2类似，但性能略低。通常在旧版本的Docker中使用。
4. **overlay2**: 是Docker默认的存储驱动，支持多层镜像的联合挂载，具有高效的写入性能和快速的容器启动速度。它是overlay的改进版本，适用于大多数场景。
5. 官方文档关于存储引擎的描述文档：
[官方docker文档对存储驱动程序](#)

注意！在一操作系统低版本内核中，例如centos-7.2版本中，如果docker数据目录是一个单独的磁盘而且是xfs格式那么需要在格式化的时候加入 -n -ftype=1，否则在一些低版本操作系统中 ftype=0 导致容器在启动时候会报错不支持d-type，centos7.3 版本之后解决了这个问题。

```
1 [root@docker-master ~]# xfs_info /
2 meta-data=/dev/mapper/k1as-root  isize=512    agcount=4, agsize=9375232 blks
3          =                      sectsz=512   attr=2, projid32bit=1
4          =                      crc=1      finobt=1, sparse=1, rmapbt=0
5          =                      reflink=1
6 data     =                      bsize=4096   blocks=37500928, imaxpct=25
7          =                      sunit=0     swidth=0 blks
8 naming   =version 2           bsize=4096   ascii-ci=0, ftype=1
9 log      =internal log       bsize=4096   blocks=18311, version=2
10         =                      sectsz=512   sunit=0 blks, lazy-count=1
11 realtime=none               extsz=4096   blocks=0, rtextents=0
```

2.4.1 修改存储引擎

注意！修改存储引擎原有的镜像容器都会没有，因此在更改存储引擎之前的需要将容器镜像/var/lib/docker目录备份，或将容器镜像上传到harbor仓库中。

通过使用 docker info 命令查看当前使用的存储引擎。

```
Server:
Containers: 0
Running: 0
Paused: 0
Stopped: 0
Images: 0
Server Version: 26.0.2
Storage Driver: overlay2
  Backing Filesystem: xfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: false
  Logging Driver: json-file
  Cgroup Driver: cgroupfs
  Cgroup Version: 1
  Plugins:
```

通过 dockerd --help 查看docker存储引擎修改方法。

```
[root@docker-master mytest]# dockerd --help | grep driver
--log-driver string                  Default driver for container logs (default "json-file")
--log-opt map                         Default log driver options for containers (default map[])
-s, --storage-driver string          Storage driver to use
--storage-opt list                    Storage driver options
```

第一种修改存储引擎方法：更改存储引擎可以通过修改 vim /lib/systemd/system/docker.service配置文件。

```
[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --insecure-registry 172.250.1.83 --storage-driver overlay2
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutStartSec=0
RestartSec=2
Restart=always
```

第二种修改存储引擎方法：可以通过/etc/docker/daemon.json文件指定存储引擎。

```
{
  "registry-mirrors": ["https://48r3burk.mirror.aliyuncs.com"],
  "storage-driver": "overlay2"
}
```

如果文件包含无效的 JSON，则 Docker 不会启动。 daemon.json

2.4.2修改docker存储目录

docker 默认将数据存储在 /var/lib/docker 目录中，为了安全起见我们服务器会有两块或多块硬盘等情况，一块是系统盘，一块是数据盘，而我们可以将数据盘格式化挂载到一个新创建的数据目录给docker存储数据使用，使用dockerd -help 查看帮助过滤出修改使用的参数。

```
[root@docker-server04 ~]# dockerd --help | grep data-root
--data-root string          Root directory of persistent Docker state (default "/var/lib/docker")
```

修改 /etc/docker/daemon.json文件指定docker数据存储目录，注意目录要存在。

```
{
  "registry-mirrors": ["https://48r3burk.mirror.aliyuncs.com"],
  "storage-driver": "overlay2",
  --data-root": "/data/docker"
}
```

使用docker info查看修改的存储目录

```
Total Memory: 7.515GiB
Name: docker-server03
ID: de086027-cd1e-42b2-a18a-302a86527b17
Docker Root Dir: /data/docker
Debug Mode: false
```

在使用 ls 命令查看/data/docker 目录，目录当中会生成一些数据存放目录。

```
[root@docker-server03 ~]# ls /data/docker/
buildkit  containers  engine-id  image  network  overlay2  plugins  runtimes  swarm  tmp  user  volumes
```

2.5 docker服务进程

通过查看docker 进程了解docker的运行方式。有一下四个进程：

```
1 # 查看进程命令有:
2 pstree -p
3 ps -ef | grep containerd
4 ps -ef | grep dockerd
```

dockerd：被client直接访问，其父进程为宿主机的systemd 守护进程。

docker-proxy：实现容器通信用来创建和维护容器iptables规则，其父进程为dockerd。

containerd：被docker进程调用以实现与runc交互，主要用来创建容器的。

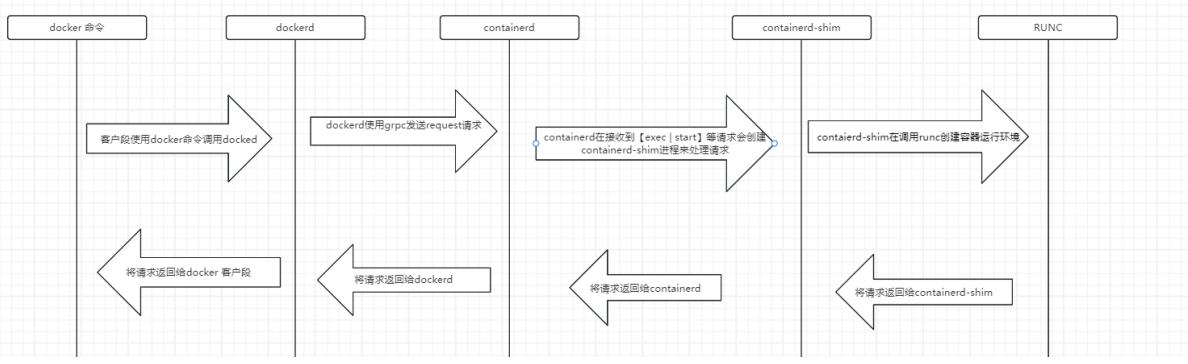
conatinerd-shim：真正运行容器的载体，其父进程为containerd.

docker的进程其实是在systemd进程中启动两个进程，一个是dockerd，我们通过dockr命令调用dockerd进程，而docker进程则调用containerd进程。containerd是启动容器每一个容器通过containerd来封装容器运行环境，而dockerd进程下面有一个docker-proxy进程，用来创建和维护容器的iptables的规则，如果容器被删除了docker-proxy还需要把容器的iptables给删除。

2.6 容器创建与管理过程

1. 我们通过docker命令调用dockerd，而dockerd 通过grpc和containerd模块通信（runc）交换，而通信的原因是在/lib/systemd/system/docker.service文件中指定了/run/containerd/conatiner.sock，让dockerd 知道containerd在什么位置，在把请求转发给containerd，dockerd和containerd通信的socket文件：/run/containerd/conatiner.sock。
2. containerd 在dockerd启动时被启动，然后conatinerd接收到dockerd的请求，来启动grpc 请求监听，containerd处理grpc请求，根据请求做相应动作。 /usr/bin/dockerd -H fd:// -- containerd=/run/containerd/containerd.sock
3. 这个动作就是若是创建容器， containerd拉起一个containerd-shim容器进程，并进行相应的创建操作。
4. container-shim被创建后， start/exec/create 创建RUNC进程，通过exit， control文件夹和containerd通信，通过父子进程关系和SLGGHLD信号监控容器中进程状态。

5. 在整个容器生命周期中，containerd通过epoll监控容器文件，监控容器事件。



2.6.1 gRPC简介

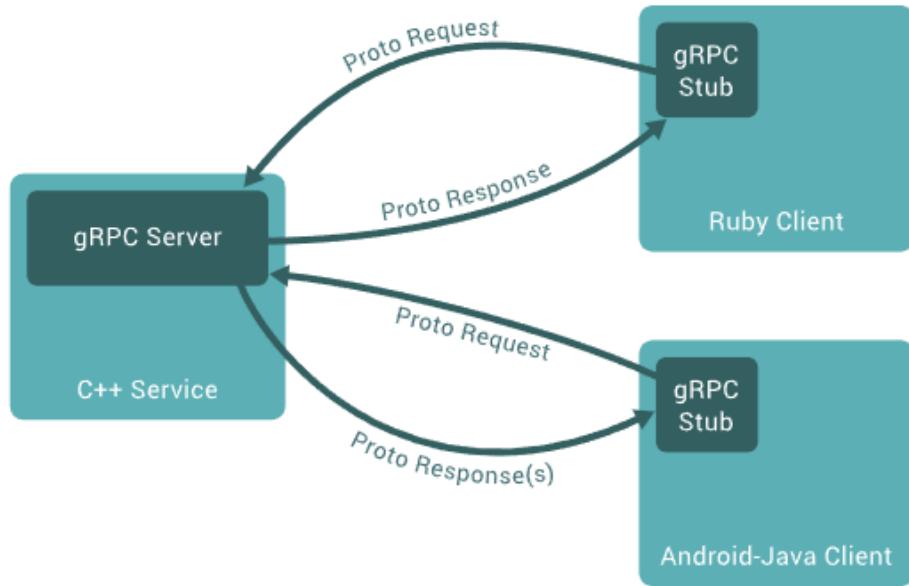
[gRPC官网](#)

gRPC 是一种高性能、开源和通用的远程过程调用 (RPC) 框架，最初由 Google 开发并开源。它基于 HTTP/2 协议进行通信，使用 Protocol Buffers (通常简称为 Protobuf) 作为接口定义语言 (IDL)。

gRPC 提供了一种简单的方法来定义服务和消息类型，然后可以自动生成客户端和服务端代码，使得客户端和服务端可以像调用本地对象一样进行通信。它支持多种编程语言，包括 C++, Java, Python, Go, Ruby, C#, Node.js 等。

gRPC 具有以下特点：

- 高性能**：基于 HTTP/2 协议，支持双向流、流控制、头部压缩等特性，提供了更高效的网络传输性能。
- 简单易用**：使用 Protocol Buffers 定义接口，自动生成客户端和服务端代码，提供了简洁的 API 接口。
- 多语言支持**：支持多种编程语言，使得不同语言的系统可以轻松进行通信。
- 可插拔性**：支持各种认证、负载均衡、错误处理等扩展功能，可以根据需要进行定制。
- 跨平台支持**：可以在不同的平台上部署，包括服务器、移动设备和浏览器。



3. docker 镜像加速配置

国内下载国外的镜像有时候会很慢，因此可以更改docker配置文件添加一个加速器，可以通过加速器达到加速下载的目的。

3.1 获取加速地址

浏览器打开 [阿里云官网](#) 登录或注册账号，点击产品与服务 ----> 点击容器镜像服务ACR ----> 点击镜像工具 ----> 点击镜像加速器，将会得到一个加速地址，当中还有配置方法。

The screenshot shows the Alibaba Cloud Container Registry (ACR) interface. In the top navigation bar, there is a '阿里云' logo and a '工作台' button. On the left sidebar, under '容器镜像服务', the '镜像加速器' option is selected. The main content area is titled '镜像加速器'. A yellow warning box at the top states: '由于当前运营商网络问题，可能会导致您拉取 Docker Hub 镜像变慢。建议您手动拉取镜像到本地节点，然后重启Pod。您也可以将镜像上传到 ACR 中或使用订阅海外源镜像功能，再从 ACR 拉取对应镜像。' Below this, there is a section titled '加速器' with a text input field labeled '加速器地址' containing the URL 'https://48r3burk.mirror.aliyuncs.com'. A red arrow points to the '复制' (Copy) button next to the URL. Further down, there is a '操作文档' section for 'CentOS' with steps 1 and 2:

1. 安装 / 升级Docker客户端
推荐安装 1.10.0 以上版本的Docker客户端，参考文档[docker-ce](#)
2. 配置镜像加速器
针对Docker客户端版本大于 1.10.0 的用户
您可以修改 daemon 配置文件 /etc/docker/daemon.json 来使用加速器

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<- 'EOF'
{
  "registry-mirrors": ["https://48r3burk.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

3.2 生成配置文件

```
1 sudo mkdir -p /etc/docker
2 sudo tee /etc/docker/daemon.json <<- 'EOF'
3 {
4   "registry-mirrors": ["https://48r3burk.mirror.aliyuncs.com"]
5 }
6 EOF
7 sudo systemctl daemon-reload
8 sudo systemctl restart docker
```

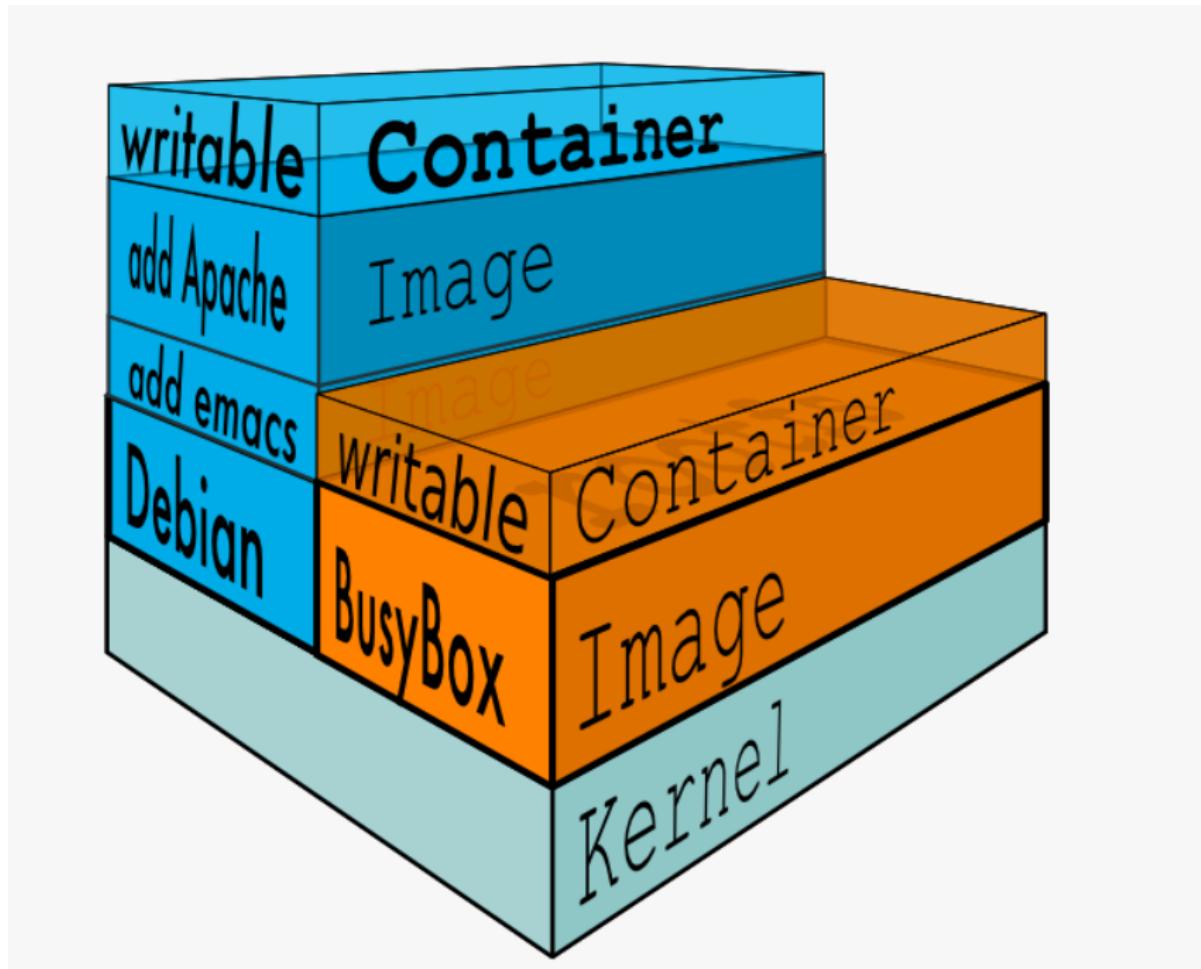
4. docker 镜像管理

docker 镜像含有启动容器所需要的文件系统以及所需要的应用，因此镜像主要用于创建并启动docker容器。

docker镜像里面是一层层文件系统，实际上Docker 镜像使用的是种叫做 UnionFS (Union File System) 的技术。(Union File System) 联合文件系统，2004年由纽约州立大学开发，它可以把多个目录内容联合挂载到同一个目录下，而目录的物理位置是分开的。UnionFS可以把只读和可读写文件系统合并在一起，具有写时复制功能，允许只读文件系统的修改可以保存到可写文件系统当中。

docker通过这些文件在加上宿主机的内核提供了Linux的虚拟环境，每一层文件系统我们叫做layer，联合文件夹系统可以对每一层文件系统设置三种权限，只读 (readonly)、读写 (readwrite)、写出 (whiteout-able)，但是docekr镜像中每层文件系统都是只读的，从基础镜像开始，每次构建镜像时所做的更改都会添加一个新的层，每个镜像层都是只读的，这意味着它们在创建后不能被修改。任何对文件系统的更改都会创建一个新的镜像层，并将其叠加在已有的层之上 一层层往上叠加。

当容器运行时，Docker 将这些层以只读模式加载到容器中。容器中的任何文件修改都会被记录在容器的可写层中，而不会影响到镜像本身。最终，用户在容器中看到的是完全的文件系统视图，不需要知道其中具体有多少层或每一层的具体内容。这样，用户就可以像在一个独立的 Linux 环境中一样使用容器，而不必担心底层的细节，结构如下：



上面的这张图展示了组装的过程，每一个镜像层都是建立在另一个镜像层之上的，同时所有的镜像层都是只读的，只有每个容器最顶层的容器层才可以被用户直接读写，所有的容器都建立在一些底层服务（Kernel）上，包括命名空间、控制组、rootfs 等等，这种容器的组装方式提供了非常大的灵活性，只读的镜像层通过共享也能够减少磁盘的占用。

这个联合文件系统在现阶段的容器技术体系中，更多的是是一类存储引擎概念，实际上各linux发行版会支持不同的联合文件系统实现，例如docker 的 Overlay2 存储引擎也是一种联合文件系统，它通过将多个文件系统层组合在一起以提供容器文件系统的构建和管理。这些文件系统层包括四个主要部分，分别是 LowerDir、MergedDir、UpperDir 和 WorkDir。

就是将原有的文件系统中的不同目录进行**合并（merge）**，最后向我们呈现出一个合并后文件系统。在 overlay2 文件结构中，联合挂载技术通过联合三个不同的目录来实现：lower目录、upper目录和work目录，这三个目录联合挂载后得到merged目录。

使用 docker inspect 【镜像ID】 查看镜像文件系统。

```
"Data": {
  "LowerDir": "/data/docker/overlay2/629d50bd11345bd84caab0116f3f91e66daae16b335b5935f544e5e96fa4c55d/diff",
  "MergedDir": "/data/docker/overlay2/278e551a0cd81ffe1f61fe34b83102e66e03db19d96164f3bbd27e1326c5dbcc/merged",
  "UpperDir": "/data/docker/overlay2/278e551a0cd81ffe1f61fe34b83102e66e03db19d96164f3bbd27e1326c5dbcc/diff",
  "WorkDir": "/data/docker/overlay2/278e551a0cd81ffe1f61fe34b83102e66e03db19d96164f3bbd27e1326c5dbcc/work"
}
```

1. LowerDir (底层目录)

- 概念：LowerDir 是底层镜像的文件系统。它包含了 Docker 镜像的基础内容，即构建新镜像所需的所有文件和目录。
- 作用：LowerDir 中的文件和目录提供了容器所需的基本文件系统结构和内容，如操作系统文件、库、二进制文件等。
- 例子：LowerDir 可能对应于已有的 Docker 基础镜像，比如 CentOS、Ubuntu 等。

2. MergedDir (合并目录)

- 概念：MergedDir 是所有镜像层（包括 LowerDir 和 UpperDir）合并后的视图。

- 作用：MergedDir 是容器启动时实际加载的文件系统。它包含了 LowerDir 和 UpperDir 中的所有内容，提供了容器的完整文件系统视图。

- 例子：MergedDir 为容器提供了在启动时可见的完整文件系统，包括基础镜像和任何修改。

3. UpperDir (顶层目录)

- 概念：UpperDir 是容器的可写层。当容器启动并对文件系统进行修改时，这些修改会在 UpperDir 中进行记录。
- 作用：UpperDir 记录了容器运行时对文件系统所做的所有更改，如新增文件、修改文件等。
- 例子：UpperDir 使得容器的文件系统修改是可持久化的，而不会影响到底层的基础镜像或其他容器实例。

4. WorkDir (工作目录)

- 概念：WorkDir 是 Overlay2 驱动程序用于执行写入操作时的临时工作目录。
- 作用：WorkDir 在写入时复制 (Copy-on-Write) 过程中起到缓冲作用，用于存储写入操作的临时数据。
- 例子：WorkDir 是 Overlay2 驱动程序内部使用的临时目录，通常不会直接暴露给用户，而是用于管理写入操作的临时数据。

4.1 overlayFS

在讲 overlay2 之前，我们需要先简单了解下什么是 rootfs：rootfs 也叫 **根文件系统**，是 Linux 使用的最基本的文件系统，是内核启动时挂载的第一个文件系统，提供了根目录 /，根文件系统包含了系统启动时所必须的目录和关键性文件，以及使其他文件系统得以挂载所必要的文件。在根目录下有根文件系统的各个目录，例如 /bin、/etc、/mnt 等，再将其他分区挂载到 /mnt，/mnt 目录下就有了这个分区的各个目录和文件。

在 docker 容器中使用的同样也是 rootfs 这种文件系统，当我们通过 docker exec 命令进入到容器内部时也可以看到在根目录下有 /bin、/etc、/tmp 等目录，但是在 docker 容器中与 Linux 不同的是，在挂载 rootfs 后，docker deamon 会利用**联合挂载技术**在已有的 rootfs 上再挂载一个读写层，容器在运行过程中文件系统发生的变化只会在读写层进行修改，并通过 whiteout 文件隐藏只读层中的旧版本文件。

在介绍 docker 中使用的 overlay2 文件结构前，我们先通过对 overlay 文件系统进行简单的操作演示以便更深入理解不同层不同目录之间的关系。

1. 先创建几个文件夹和文件

```
1 mkdir A B C worker
2 echo "test-A" > A/a.txt
3 echo "test-A" > A/b.txt
4 echo "test-A" > A/c.txt
5 echo "test-B" > B/a.txt
6 echo "test-B" > B/d.txt
7 echo "test-C" > C/b.txt
8 echo "test-C" > C/e.txt
9
10 tree .
11 .
12 └── A
13     ├── a.txt
14     ├── b.txt
15     └── c.txt
16 └── B
17     ├── a.txt
18     └── d.txt
19 └── C
20     ├── b.txt
21     └── e.txt
22 └── worker
23
```

```
24 | 4 directories, 7 files
```

2. 使用 `mount` 命令挂载成 overlayFS 文件系统，格式如下

```
mount -t overlay overlay -o lowerdir=lower1:lower2:lower3,upperdir=upper,workdir=work  
merged_dir
```

在这个例子中，我们用 A 和 B 两个文件夹作为 lower 目录，用 C 作为 upper 目录，worker 作为 work 目录，挂载到 /mnt/merged 目录下

```
1 [root@docker-server03 mnt]# mkdir merged  
2 [root@docker-server03 mnt]# mount -t overlay overlay -o  
lowerdir=A:B,upperdir=C,workdir=worker ./merged  
3 [root@docker-server03 mnt]# tree merged/  
4 merged/  
5   ├── a.txt  
6   ├── b.txt  
7   ├── c.txt  
8   ├── d.txt  
9   └── e.txt  
10  
11 0 directories, 5 files
```

可以看到我们原本 A B C 三个目录下的文件已经被合并，相同文件名的文件将会有选择性的显示，在 merged 中会显示离 merged 层更近的文件，upper 层比 lower 层更近，同样的 lower 层中，排序靠前比排序靠后更近，在这个例子中就是 A 比 B 更靠近 merged 层。

根据这个规律，我们可以先分析下 merged 层中的文件来源，`a.txt` 在 A 和 B 中都有，但 A 比 B 更优先，所以 merged 中的 `a.txt` 应该来自 A 目录，`b.txt` 在 A 和 C 中都有，但 C 是 upper 层，所以 `b.txt` 应该来自 C 目录，我们可以核实一下。

```
1 [root@docker-server03 mnt]# cd merged/  
2 [root@docker-server03 merged]# cat a.txt  
3 test-A  
4 [root@docker-server03 merged]# cat b.txt  
5 test-C
```

接下来我们可以看下 upper 层、lower 层和 merged 层之间的关系，上面已经提到了 upper 层是 **读写层** 而 lower 层是 **只读层**，merged 层是联合挂载后的视图，那如果我们在 merged 层中对文件进行操作会发生什么？

```
1 [root@docker-server03 merged]# echo "DDD" > a.txt  
2 [root@docker-server03 merged]# cat a.txt  
3 DDD  
4 [root@docker-server03 merged]# cat ./A/a.txt  
5 test-A  
6 [root@docker-server03 merged]# cd ..  
7 [root@docker-server03 mnt]# tree .  
8 .  
9   └── A  
10    |   ├── a.txt  #没变  
11    |   ├── b.txt  
12    |   └── c.txt  
13    └── B  
14      |   ├── a.txt  
15      |   └── d.txt  
16    └── C  
17      |   ├── a.txt  #变了  
18      |   └── b.txt
```

```

19 |   └── e.txt
20 └── merged
21 |   ├── a.txt
22 |   ├── b.txt
23 |   ├── c.txt
24 |   ├── d.txt
25 |   └── e.txt
26 └── worker
27     └── work
28
29 6 directories, 13 files
30 [root@docker-server03 mnt]#

```

我们修改 merged 层的 `a.txt` 文件，可以看到 merged 层的 `a.txt` 内容虽然改变，但 A 目录（只读层）下的 `a.txt` 内容并没有发生变化，而在 C 目录（读写层）下多了一个 `a.txt` 文件，内容就是我们修改过的 `a.txt` 的内容，这就是只读层与读写层的关系，在 **merged** 目录对文件进行修改并不会影响到只读层的源文件，只会在 **读写层进行编辑**

如果我们在 merged 目录删除文件会发生什么

```

1 [root@docker-server03 mnt]# rm -rf merged/c.txt
2 [root@docker-server03 mnt]# ls -l ./*

```

```

[root@docker-server03 mnt]# ls -l ./*
./A:
总用量 12
-rw-r--r-- 1 root root 7 4月 21 17:29 a.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 b.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 c.txt

./B:
总用量 8
-rw-r--r-- 1 root root 7 4月 21 17:29 a.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 d.txt

./C:
总用量 12
-rw-r--r-- 1 root root 4 4月 21 17:37 a.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 b.txt
c----- 1 root root 0, 0 4月 21 17:58 c.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 e.txt

./merged:
总用量 16
-rw-r--r-- 1 root root 4 4月 21 17:37 a.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 b.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 d.txt
-rw-r--r-- 1 root root 7 4月 21 17:29 e.txt

./worker:
总用量 0
d----- 2 root root 6 4月 21 17:58 work

```

可以看到在 merged 目录中已经没有 `c.txt` 文件，但在 C 目录下却多了一个 `c.txt`，这个文件就是的 **whiteout文件**，它是一种主/次设备号都为0的字符设备，overlay 文件结构通过使用这种特殊文件来实现文件删除功能，在 merged 目录下使用 `ls` 命令来查看文件时，overlay 会自动过滤掉 upper 目录下的 whiteout 文件以及在 lower 目录下的同名文件，以此来实现文件删除效果。

whiteout 文件：

"whiteout"文件是UnionFS中的一种特殊文件，用于表示在上层文件系统中删除了一个文件或目录。在 UnionFS中，当需要删除底层文件系统中的文件或目录时，并不会直接在底层文件系统中删除，而是在上层文件系统中创建一个名为".wh.{被删除文件名}"的特殊文件，这个特殊文件即为whiteout文件。这样，当访问文件系统时，会检查上层文件系统中是否存在whiteout文件，如果存在，则会将其视为底层文件系统中相应文件或目录的删除操作。

还有一个值得提到的点：overlay在对文件进行操作时用到了**写时复制（Copy on Write）**技术，在没有对文件进行修改时，merged 目录直接使用 lower 目录下的文件，只有当我们在 merged 目录对文件进行修改时，才会把修改的文件复制到 upper 目录。

4.2 Docker overlay2

有了对 overlayFS 的基本了解，我们接下来就可以着手分析 Docker 的 overlay2 文件结构了，实际上 Docker 支持的存储驱动有很多种：overlay、overlay2、aufs、vfs，devicemappe 等，在 Ubuntu 较新版本的 Docker 中普遍采用了 overlay2 这种文件结构，其具有更优越的驱动性能，而 overlay 与 overlay2 的本质区别就是二者在镜像层之间的共享数据方法不同：

- overlay 通过 **硬链接** 的方式共享数据，只支持，增加了磁盘 inode 负担
- overlay2 通过将多层 lower 文件联合在一起

简而言之，overlay2 就是 overlay 的改进版本，我们可以通过 `docker info | grep " Storage Driver"` 命令查看。

```
[root@docker-server03 mnt]# docker info | grep " Storage Driver"
Storage Driver: overlay2
```

在 Docker 中，我们日常操作主要涉及两个层面：镜像层与容器层，镜像层就是我们通过 `docker pull` 等命令下载到本机中的镜像，而容器层则是我们通过 `docker exec` 等命令进入的交互式终端，如果你使用过 Docker，你会发现我们只用一个镜像，通过 `docker run` 可以产生很多个容器，这就可以类比 upper 与 lower 两层，镜像作为 lower 层，只读提供文件系统基础，而容器作为 upper 层，我们可以在其中进行任意文件操作，只用同一个镜像就可以引申出不同的容器。通过这种机制，我们可以节约空间资源，因为多个容器可以共享同一个镜像层，而容器层只包含容器特定的修改和数据，不同的容器可以基于同一个镜像启动，每个容器都有自己独立的文件系统。

4.2.1 镜像层

通过 `docker inspect` 【镜像ID】 来查看镜像配置

```
"GraphDriver": {
    "Data": {
        "LowerDir": "/var/lib/docker/overlay2/3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa9c2ef659c22c5c6/diff:/var/lib/docker/overlay2/63b1fd65cd292a803d22cc89cd3999fc89ebd88334872b0cf1c5916d85603820/diff:/var/lib/docker/overlay2/12f1e58e359274373828c367fe0cf26f7a422834b93a4d3fd5254fbe8af376f7/diff:/var/lib/docker/overlay2/355e046273ab88e4898624c45c3db22ae4806ed2610d1c1cad5b1cf5ad7503d/diff:/var/lib/docker/overlay2/e2507fefd275983b6e56855cd48c5118500e4d2847405e7e08e44ed4e13f7d25/diff:/var/lib/docker/overlay2/9052aa3377b45f65ab7c3eca6ef00c18a43e702d00136d3ae5772fc0e8ab24/diff:/var/lib/docker/overlay2/99de2110a3d25229444153e7470ceb350d9f8ee5f979e4cd10bb791790b2cf/diff:/var/lib/docker/overlay2/0475b0d70638514f0713bb217239a7ec3453d5c4f98912873f0e58589fa43100/diff:/var/lib/docker/overlay2/ce1b630606113c23903890567e0d79301c3bddce03d1e4abe28e822415b0400/diff",
        "MergedDir": "/var/lib/docker/overlay2/8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb877fbe9b7f010536/merged",
        "UpperDir": "/var/lib/docker/overlay2/8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb877fbe9b7f010536/diff",
        "WorkDir": "/var/lib/docker/overlay2/8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb877fbe9b7f010536/work"
    },
    "Name": "overlay2"
},
```

其中的 `GraphDriver` 字段中有关于 overlay2 文件结构的目录信息

每一层的对应都在配置信息中体现的非常清楚，但是有一点问题，我们在实际查看文件夹的时候，可以发现镜像层其实并没有 `merged` 目录。镜像层作为只读层，不需要有 `merged` 目录，因为它不会进行写操作，也不需要对文件系统进行修改。它的主要作用是提供文件系统的基础，供容器层使用。因此，镜像层在配置信息中显示的主要是为了展示完整的 Overlay2 文件结构，以便管理和查看容器的文件系统。

可以看到镜像的目录是在 `/var/lib/docker/overlay2` 下，我们打开一个镜像层看一看其中都有哪些文件。

```
root@ftpd:/var/lib/docker/overlay2# cd 3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa9c2ef659c22c5c6
root@ftpd:/var/lib/docker/overlay2/3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa9c2ef659c22c5c6# ls -al
total 24
drwx----x 4 root root 4096 Sep 29 20:50 .
drwx----x 15 root root 4096 Nov 16 23:15 ..
-rw----- 1 root root 0 Sep 29 20:50 committed
drwxr-xr-x 2 root root 4096 Sep 29 20:50 diff
-rw-r--r-- 1 root root 26 Sep 29 20:50 link
-rw-r--r-- 1 root root 231 Sep 29 20:50 lower
drwx----x 2 root root 4096 Sep 29 20:50 work
root@ftpd:/var/lib/docker/overlay2/3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa9c2ef659c22c5c6#
```

其中我们关注下 `diff` 目录、`link` 和 `lower` 文件。

4.2.2 diff 目录

在这个目录中存放的是当前镜像层的文件，刚刚在介绍 overlay2 与 overlay 区别的时候提到了 overlay2 是将多个 lower 层联合到一起，在上面的图中也可以看到，**多个 lower 层之间用 : 分割**，在这些层中每一层都有一部分文件，把他们联合到一起就得到了完整rootfs。

```
root@ftpd:/var/lib/docker/overlay2# ls 3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa9c2ef659c22c5c6/diff
run.sh
root@ftpd:/var/lib/docker/overlay2# ls 63b1fd65cd292a803d22cc89cd3999fc89ebd88334872b0cf1c5916d85603820/diff
run.sh
root@ftpd:/var/lib/docker/overlay2# ls e2507fefd275983b6e56855cd48c5118500e4d2847405e7e08e44ed4e13f7d25/diff
tmp var
root@ftpd:/var/lib/docker/overlay2# ls ce13b630606113c23903890567e0d79301c3bddce03d1e4abe28e822415b0400/diff
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@ftpd:/var/lib/docker/overlay2#
```

4.2.3 link 文件

link 文件中的内容是当前层的软链接名称。

```
root@ftpd:/var/lib/docker/overlay2# cat 0475b0d70638514f0713bb217239a7ec3453d5c4f98912873f0e58589fa43100/link
G25JAP7XBMIDCRA2FTZ7AEVFKMroot@ftpd:/var/lib/docker/overlay2#
```

这些链接都在 `/var/lib/docker/overlay2/l/` 目录下，使用软链接的目的是为了避免达到 mount 命令参数的长度限制。

```
G25JAP7XBMIDCRA2FTZ7AEVFKMroot@ftpd:/var/lib/docker/overlay2# cd l
root@ftpd:/var/lib/docker/overlay2/l# ls -al
total 56
drwx-----x 2 root root 4096 Nov 16 23:15 .
drwx-----x 15 root root 4096 Nov 16 23:15 ..
lrwxrwxrwx 1 root root 72 Sep 29 20:50 BLSYC43MQCXDCBVCEPAH72KM3 -> .../9052aa3377b45f65ab7c3eca6ef00c18a43e702d00136d3ae
5772fe9b0e8a0b24/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 BLTJMAE6HQ2SJPABNC7RF7EZX -> .../63b1fd65cd292a803d22cc89cd3999fc89ebd88334872b0cf
lc5916d85603820/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 FRJ422QTCT3DH07AGAAGJLWALJK -> .../8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb8
77fbe9b7f010536/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 G25JAP7XBMIDCRA2FTZ7AEVFKM -> .../0475b0d70638514f0713bb217239a7ec3453d5c4f98912873
f0e58589fa43100/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 GTADZFMXTG3NEEN404N5YFR4H3 -> .../3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa
9c2ef659c22c5c6/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 IWRMM2YPVQNEGTUNFN2WBM63A -> .../355e046273ab88e4898624c45c3db22ae4806ed2610d1c1dc
ad5b1cf5ad7503d/diff
lrwxrwxrwx 1 root root 77 Sep 29 20:59 LOU3CPT5DIMGG236HCTDPGWAK -> .../575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418
ffdd415cf754e53-init/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 MXF5HX3GYMBXETLM4X3VEY03W -> .../99de2110a3d25229444153e7470cbeb350d9f8ee5f979e4cd
c10bb791790b2cf/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 TXARCPWBRGF6T3KXG0CKSXHFHW -> .../12f1e58e359274373828c367fe8cf26f7a422834b93a4d3fd
5254fe8af376f77/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 WE2X0LGK4YRRM5NJG0FRKDILM -> .../e2507fefd275983b6e56855cd48c5118500e4d2847405e7e0
8e44ed4e13f7d25/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:59 XSA6RT6WGVB27JW46ZV33FL3JA -> .../575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418
ffdd415cf754e53/diff
lrwxrwxrwx 1 root root 72 Sep 29 20:50 YTESJVNLFIGI3C6A70QFMQDIWT -> .../ce13b630606113c23903890567e0d79301c3bddce03d1e4ab
e28e822415b0400/diff
root@ftpd:/var/lib/docker/overlay2/l#
```

4.2.4 lower 文件

lower 文件中的内容是在此层之下所有层的软链接名称，最底层不存在该文件，我们知道 upper 层在 lower 层之上，而 lower 层中越靠后则越在底层。

我们查看 upper 层对应目录下 lower 文件，可以发现其中有9个软链接

```
root@ftpd:/var/lib/docker/overlay2# cat 8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb877fe9b7f010536/lower | grep 'l'
l/GTADZFMXTG3NEEN404N5YFR4H3:1/BLTJMAE6HQ2SJPABNC7RF7EZX:1/TXARCPWBRGF6T3KXG0CKSXHFHW:1/IWRMM2YPVQNEGTUNFN2WBM63A:1/WE2X0
LKG4YRRM5NJG0FRKDILM:1/BLSYC43MQCXDCBVCEPAH72KM3:1/MXF5HX3GYMBXETLM4X3VEY03W:1/G25JAP7XBMIDCRA2FTZ7AEVFKM:1/YTESJVNLFIGI3C
6A70QFMQDIWT
root@ftpd:/var/lib/docker/overlay2#
```

恰好 lower 目录中有9个镜像层

```
"LowerDir": "/var/lib/docker/overlay2/3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa9c2ef659c22c5c6/diff:
/var/lib/docker/overlay2/63b1fd65cd292a803d22cc89cd3999fc89ebd88334872b0cf1c5916d85603820/diff:/var/lib/docker/overlay2/12f1
e58e359274373828c367fe8cf26f7a422834b93a4d3fd5254fe8af376f7/diff:/var/lib/docker/overlay2/e2507fefd275983b6e56855cd48c5118500e4d2847405e7e08e44ed4e13f7d
25/diff:/var/lib/docker/overlay2/9052aa3377b45f65ab7c3eca6ef00c18a43e702d00136d3ae5772fe0e8a0b24/diff:/var/lib/docker/overlay2/0475b0d70638514f0713bb217
239a7ec3453d5c4f98912873f0e58589fa43100/diff:/var/lib/docker/overlay2/ce13b630606113c23903890567e0d79301c3bddce03d1e4abe28e8
22415b0400/diff",
```

在 lower 层中，处于最底层的则应该是在 : 最后的目录。

即 `/var/lib/docker/overlay2/ce13b630606113c23903890567e0d79301c3bddce03d1e4abe28e822415b040`

查看这一目录下的文件，可以发现它并没有 lower 文件。这是因为位于最底层的镜像层（只读层）不需要具有 lower 文件。Lower 文件实际上记录了其他层（包括容器层和其他镜像层）相对于当前层的挂载点。由于镜像层位于最底层，它没有任何其他层挂载在它的上面，因此不需要 lower 文件。Lower 文件通常只存在于容器层和基于镜像的容器层中。

```
root@ftpd:/var/lib/docker/overlay2# ls -al ce13b630606113c23903890567e0d79301c3bddce03d1e4abe28e822415b0400/
total 16
drwx----x 3 root root 4096 Sep 29 20:50 .
drwx----x 15 root root 4096 Nov 16 23:15 ..
-rw----- 1 root root 0 Sep 29 20:50 committed
drwxr-xr-x 21 root root 4096 Sep 29 20:50 diff
-rw-r--r-- 1 root root 26 Sep 29 20:50 link
root@ftpd:/var/lib/docker/overlay2#
```

我们查看其上一层的 lower 文件内容，这一层对应的软链接即 link 文件内容为 `YTESJVNLFIGI3C6A70QFMQDIWT`，可以发现确实对应了最底层目录的软链接。

```
root@ftpd:/var/lib/docker/overlay2# cat 0475b0d70638514f0713bb217239a7ec3453d5c4f98912873f0e58589fa43100/lower
1/YTESJVNLFIGI3C6A70QFMQDIWT root@ftpd:/var/lib/docker/overlay2#
```

元数据

Docker 的元数据存储目录为 `/var/lib/docker/image/overlay2`，我们主要看 imagedb 和 layerdb 这两个文件夹。

```
root@ftpd:/var/lib/docker/image/overlay2# ls -al
total 24
drwx---- 5 root root 4096 Sep 29 20:50 .
drwx---- 3 root root 4096 Sep 29 20:50 ..
drwx---- 4 root root 4096 Sep 29 20:50 distribution
drwx---- 4 root root 4096 Sep 29 20:50 imagedb
drwx---- 5 root root 4096 Sep 29 20:59 layerdb
-rw----- 1 root root 313 Sep 29 20:50 repositories.json
root@ftpd:/var/lib/docker/image/overlay2#
```

imagedb

这个文件夹中存储了镜像相关的元数据，具体位置是在 `/imagedb/content/sha256` 目录下，这个目录下的文件以 **镜像ID** 来命名。

```
root@ftpd:/var/lib/docker/image/overlay2/imagedb/content/sha256# docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
stilliard/pure-ftpd    latest   04e2c228c8e9    4 months ago   148MB
root@ftpd:/var/lib/docker/image/overlay2/imagedb/content/sha256# ls
04e2c228c8e97b1628a1f9dce3786e74bf52875b38b567be5ecc6a74198ba8a
root@ftpd:/var/lib/docker/image/overlay2/imagedb/content/sha256#
```

这个文件的内容就是我们通过 `docker inspect [镜像 ID]` 命令查看到的信息，其中我们关注 `RootFS` 字段。

```
"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7",
    "sha256:71931e5ac1f875e61a93d6c43aab8176bc1be6b38fed0e1681e5d38c196732a5",
    "sha256:afffe3290d8cb2e178d425e944acb89a77957f3156fd637dce6b9d32a27a495",
    "sha256:70dbe6df341e974424d762d196b82136a64ff2da2cd04f7750aeef813ad73c6a1",
    "sha256:c550a39815750265085026b4c6688834843962127442589e9807ba194063e9cd",
    "sha256:77e310dc441ab3832859eb74e802cd5157a2e502c7275703b98843bc97c95e3",
    "sha256:f697f4f0184d95702f81aa5180e168a54bac1c0bea076d109c43935374e1e528",
    "sha256:01ced56987c0bcb6c34d87ae16e5661daf5eb2520lbe1391c6d717c064f068",
    "sha256:435f7e161b7e8eb81celb69575b2ee36b722891f9b196e456b9b0739519da88",
    "sha256:bd657d29e36dc4bc702a5e9c8ed96bbba7e7520a5717f1363beca2528dd0949d"
  ],
}
```

可以看到这个字段中有许多 sha256 值，这些哈希值称为 **diff_id**，其从上至下的顺序就表示镜像层最底层到最顶层，也就是说每个 diff_id 都对应了一个镜像层，实际上，对应每一个镜像层的还有另外两个 id： **cache_id** 和 **chain_id**

- cache_id 就是在 `docker/overlay2` 目录下看到的文件夹名称，也是我们通过 `docker inspect [镜像 ID]` 命令查看 GraphDriver 字段对应不同的 Dir，其本质是宿主机随机生成的uuid。

```
"GraphDriver": {
  "Data": {
    "LowerDir": "/var/lib/docker/overlay2/3b232efd9fafbdd6f6b0cc6b0ab13cf08c02a5671a73c26fa9c2ef659c22c5c6/diff",
    "UpperDir": "/var/lib/docker/overlay2/63b1fd65cd292a803d22c89cd3999fc89ebd88334872b0cfc1c5916d85603820/diff",
    "MergedDir": "/var/lib/docker/overlay2/8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb877fbe9b7f010536/merged",
    "CacheDir": "/var/lib/docker/overlay2/8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb877fbe9b7f010536/cache",
    "WorkDir": "/var/lib/docker/overlay2/8b63e9e3cdc41a140d28a5d434b311dc81d746d43b237acb877fbe9b7f010536/work"
  }
}
```

- chain_id 是通过 diff_id 计算出来的，是 Docker 内容寻址机制采用的索引 ID
 - chain_id 在目录 `/docker/image/overlay2/layerdb/sha256` 查看
 - 如果当前镜像层为最底层，则其 chain_id 与 diff_id 相同
 - 如果当前镜像层不是最底层，则其 chain_id 计算方式为：`sha256(上层chain_id + " " + 本层diff_id)`

这三个 id 之间存在一一对应的关系，我们可以通过 diff_id 计算得到 chain_id，又可以通过 chain_id 找到对应的 cache_id，下面我们举个栗子说明一下：

我们刚刚提到了 diff_id 从上至下是最底层到最顶层。

```
"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7" 最底层
    "sha256:71931e5ac1f875e61a93d6c43aab8176bc1be6b38fed0e1681e5d38c196732a5",
    "sha256:a7ffe3290d8cb2e2178d425e944acb89a77957f3156fd37dce6b9d32a7a495",
    "sha256:70dbe6df341e974424d762d196682136a64ff2da2cd04f7750aef813ad73c6a1",
    "sha256:c550a39815750265085026b4c6688834843962127442589e9807ba194063e9cd",
    "sha256:77e31c0dc441ab3832859eb74e802cd5157a2e502c7275703b98843bc97c95e3",
    "sha256:f697f4f0184d95792f81aa5180e168a54bac1c0bea076d109c43935374e1e528",
    "sha256:01ced56987c0bc6c34d87ae16e5661daf5eb25201be13911c6d717c064f068",
    "sha256:435f7e161b7e8eb81celcb69575b2ee36b722891f9b196e456b9b0739519da88",
    "sha256:bd657d29e36dc4bc702a5e9c8ed96bbba7e7520a5717f1363beca2528dd094d9" 最顶层
  ],
}
```

查看 chain_id

```
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# ls
1904ea1ad0d36d50beb8d2c5faf39a2caa197f786eef3c2b8300472ba2185228
21975be7804f1b4fd726b0d7664efbf954a6287dce481c7e087b0022e944433b
3dbe253f8a2bc44900b046c1f54b1bb56d2b61a6d3eb2d740910eee92d642930
44f1a2b9e8052fe06ed9f7c601fc57820d96a5025af95b92e56b850b8219e7
764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7
7d06128b1d3a4f6ae4eae94af7b7df3759f981e80e0d0e4a4bc0fa84a6640
841b336d41eed1353a95a776a4279514eb232fd5c7d63b8e9906e77e63c1fa
85b5336a2ae86b41c42264930ab3cb08c574218aff7347e96d73bfa8773e9c25
94f59313e010badc95dadd0fa2bedbd6e83121fb624be46e301a7248306ece0a
f28066a04d32a8782c9765cf1fb55ab750d8948d69a71ea7023cd315a50e0787
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256#
```

可以看到其中确实有一个 chain_id 与 最底层的 diff_id 相同（红框标出），有了最底层的 chain_id 我们就可以计算出下一层的 chain_id，至于具体如何计算，以及如何通过 chain_id 找到对应的 cache_id，我们需要先了解 layerdb 目录下的内容

4.2.5 layerdb

我们现在已知 Docker 的镜像层作为只读层，容器层作为读写层，而 Docker 实际上定义了 roLayer 接口与 mountLayer 接口，分别用来描述（只读）镜像层与（读写）容器层，这两个接口的元数据就在目录 `docker/image/overlay2/layerdb` 下

4.2.6 roLayer

rolayer 接口用来描述镜像层，元数据的具体目录在 `layerdb/sha256/` 下，在此目录下每个文件夹都以每个镜像层的 chain_id 命名

```
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# ls -l 3dbe253f8a2bc44900b046c1f54b1bb56d2b61a6d3eb2d740910eee92d642930
total 20
-rw-r--r-- 1 root root 64 Sep 29 20:50 cache-id
-rw-r--r-- 1 root root 71 Sep 29 20:50 diff
-rw-r--r-- 1 root root 71 Sep 29 20:50 parent
-rw-r--r-- 1 root root 4 Sep 29 20:50 size
-rw-r--r-- 1 root root 299 Sep 29 20:50 tar-split.json.gz
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256#
```

在文件夹中主要有这5个文件，我们简单介绍一下：

- cache-id：当前 chain_id 对应的 cache_id，用来索引镜像层。
- diff：当前 chain_id 对应的 diff_id。
- parent：当前 chain_id 对应的镜像层的下一层（父层）镜像 chain_id，最底层不存在该文件。
- size：当前 chain_id 对应的镜像层物理大小，单位是字节。
- tar-split.json.gz：当前 chain_id 对应镜像层压缩包的 split 文件，可以用来还原镜像层的 tar 包，通过 `docker save` 命令导出镜像时会用到。

我们在上文中已经判断出了最底层镜像对应的 chain_id，不妨查看下对应目录下的文件。

```
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# ls -l 764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7
total 256
-rw-r--r-- 1 root root      64 Sep 29 20:50 cache-id
-rw-r--r-- 1 root root     71 Sep 29 20:50 diff
-rw-r--r-- 1 root root      8 Sep 29 20:50 size
-rw-r--r-- 1 root root 249737 Sep 29 20:50 tar-split.json.gz
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256#
```

可以看到该目录下确实没有 parent 文件，那么我们再查看其下一层，通过 diff_id 的顺序我们可以得知其下一层的 diff_id 为 `71931e5ac1f875e61a93d6c43aab8176bc1be6b38fed0e1681e5d38c196732a5`，通过计算 sha256，我们可以得出下一层的 chain_id。

明文：

```
sha256:764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7 sha256:71931e5ac1f875e61a93d6c43aab8176bc1be6b38fed0e1681e5d38c196732a5
上层镜像的chain_id, 即parent文件内容                                当前镜像层的diff_id, 即diff文件内容
注意中间用空格连接
```

散列/哈希算法：

SHA1 SHA224 SHA256 SHA384 SHA512 MD5

HmacSHA1 HmacSHA224 HmacSHA256 HmacSHA384 HmacSHA512 HmacMD5 PBKDF2

哈希值：

```
7d06128b1d3a4f6aef4eae94afb7bdf3759f981e80e0dd0e4a4bc0cfa84a6640
```

计算得到最底层的下一层镜像 chain_id 为

```
7d06128b1d3a4f6aef4eae94afb7bdf3759f981e80e0dd0e4a4bc0cfa84a6640。
```

```
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# ls
1904ea1ad0d36d50bеб8d2c5faf39a2caa197f786eef3c2b8300472ba2185228
219f5be7804fb14fd726b0d7664eфbf954a6287dce481c7e087b0022e944433b
3dbe253f8a2bc4900b046c1f54b1b56d2b61a6d3eb2d740910eee92d642930
44f1a2b9e8052fe6edf9f7c601fc57820d96a5025af9d592e56b850b8219e7
764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7
7d06128b1d3a4f6aef4eae94afb7bdf3759f981e80e0dd0e4a4bc0cfa84a6640
841b336d41eed1353aa95a776a4279514eb232fd5c7d63b8e906e77e63c1fa
85b536a2ae86b41c42264930ab3cb08c574218aff7347e96d73bfa8773e90c25
94f50313e010badc95dadd0fa2bedbd6e83121fb624be46e301a7248306ece0a
f28066a04d32a8782c9765c1fb55ab750d8948d69a71ea7023cd315a50e0787
```

确实存在该目录，证明计算无误，再查看此目录下 diff 文件与 parent 文件内容。

```
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# cd 7d06128b1d3a4f6aef4eae94afb7bdf3759f981e80e0dd0e4a4bc0cfa84a6640# cat diff
sha256:71931e5ac1f875e61a93d6c43aab8176bc1be6b38fed0e1681e5d38c196732a5# root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256:764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7# root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256:7d06128b1d3a4f6aef4eae94afb7bdf3759f981e80e0dd0e4a4bc0cfa84a6640# parent
```

可以看到与我们计算用到的两个值也完全相同

4.2.7 mountLayer

mountLayer 接口用来描述容器层，元数据的具体目录在 `/layerdb/mounts/`，在此目录下的文件夹以每个容器的容器ID (CONTAINER ID) 命名。

```
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts# ls -l
total 4
drwxr-xr-x 2 root root 4096 Sep 29 20:59 0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
0b6f2be915fb        stilliard/pure-ftpd   "/bin/sh -c '/run.sh'"   7 weeks ago        Up 2 days          30000-30009/tcp, 0.0.0.0:30010-30019
->30010-30019/tcp, :::30010-30019->30010-30019/tcp, 0.0.0.0:23->21/tcp, :::23->21/tcp
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts# docker inspect 0b6f2be915fb
{
    "Id": "0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52",
    "Created": "2021-09-29T12:59:16.17049283Z",
    "Path": "/bin/sh",
```

在这个文件夹下只有3个文件，内容如下：

```

root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52# ls
-l
total 12
-rw-r--r-- 1 root root 69 Sep 29 20:59 init-id
-rw-r--r-- 1 root root 64 Sep 29 20:59 mount-id
-rw-r--r-- 1 root root 71 Sep 29 20:59 parent
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52# cat init-id
575db3795cb5ff7cadb347f10fcfaad5fd5fe3e5b3e31418ffdd415cf754e53-init
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52# cat mount-id
575db3795cb5ff7cadb347f10fcfaad5fd5fe3e5b3e31418ffdd415cf754e53
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52# cat parent
sha256:1904ea1add036d0be6bd2c5faf39a2caa197f786eeef3c2b8300472ba2185228
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52#

```

简单介绍一下这3个文件：

- init-id: 对应容器 init 层目录名，源文件在 `/var/lib/docker/overlay2` 目录下
- mount-id: 容器层存储在 `/var/lib/docker/overlay2` 目录下的名称
- parent: 容器的镜像层最顶层镜像的 chain_id

我们可以查看 parent 这个文件中 chain_id 对应目录下的 diff 文件。

```

root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# ls
1904ea1add036d0be6bd2c5faf39a2caa197f786eeef3c2b8300472ba2185228 7d06128b1d3a4f6ae94fa7bf7d3759f981e80e0dd0e4a4bc0cfa84a6640
219f5b6e7804fb1b4fd726b0d7664efb954a6287dce4817e087b0022e944333b 841b336d41eed1353aa95a776a4279514eb232fd5c7d3b8e9e906e77e631cfa
3de25378a2bc44900b046c1f154b1b56d2b61e6d3be2d740910eee92d642930 85b536a2ae86b41c42264930ab3cb08c574218aff7347e96d73bfa8773e90c25
44f1a2b9e8052fe0edf9f7c601fc5782d96a5025afdf95b9e56b850h8219e7 94f50313e010badc95dad0fa2bedhd6e83121fh624be46e301a7248306ece0a
764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7 128066a04d32a8782c9765cf1fb55ab750d8948d69a71ea7023cd315a50e0787
root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# cd ../../mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52/
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52# ls -l
total 12
-rw-r--r-- 1 root root 69 Sep 29 20:59 init-id
-rw-r--r-- 1 root root 64 Sep 29 20:59 mount-id
-rw-r--r-- 1 root root 71 Sep 29 20:59 parent
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52# cat ../../sha256/1904ea1
add036d0be6bd2c5faf39a2caa197f786eeef3c2b8300472ba2185228/diff
sha256:bd657d29e36dc4bc702a5e9c8ed96bba7e7520a517f1363beca2528dd0949d
root@ftpd:/var/lib/docker/image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52#

```

根据 diff_id 从上至下的顺序，我们可以确定这个 diff_id 的确是镜像层的最顶层

```

root@ftpd:/var/lib/docker/image/overlay2/layerdb/sha256# docker image inspect 04e2 | grep -i "rootfs" -A 13
"RootFS": {
    "Type": "layers",
    "Layers": [
        "sha256:764055ebc9a7a290b64d17cf9ea550f1099c202d83795aa967428ebdf335c9f7",
        "sha256:71931e5ac1f875e61a93d6c43aaab8176b1be6b38fed0e1681e5d38196732a5",
        "sha256:afffe3290d8cb2e2178d425e944acb89a7795773156fd637dce6b9d32a27495",
        "sha256:70dbe6df341e74424d762d196b82136a64ff2da2cd04f7750ae8f13ad73c6a1",
        "sha256:c550a39815750265085026b4c6688834843962127442589e9807ba194063e9cd",
        "sha256:7e31c0dc41ab3832859eb74e802cd5157a2e502c7275703b98843bc97c95e3",
        "sha256:697f4f0184d95702f81aa5180e168a54ba0cbe0a076d109c43935374e1e528",
        "sha256:01ced6987c0bc6c34087ae116e561dafb5201be13911c6d717c064f068",
        "sha256:435f7e161b7e8eb81c1cb69575b2ee36b722891f9b196e456b9b0739519da88",
        "sha256:bd657d29e36dc4bc702a5e9c8ed96bba7e7520a517f1363beca2528dd0949d"
    ]
}

```

在这里我们引入了一个叫做 **init层** 的概念，实际上，一个完整的容器分为3层：镜像层、init层和容器层，镜像层提供完整的文件系统基础（rootfs），容器层提供给用户进行交互操作与读写权限，而 init 层则是对应每个容器自己的一些系统配置文件，我们可以看一下 init 层的内容。

```

root@ftpd:/var/lib/docker/overlay2# tree 575db3795cb5ff7cadb347f10fcfaad5fd5fe3e5b3e31418ffdd415cf754e53-init
575db3795cb5ff7cadb347f10fcfaad5fd5fe3e5b3e31418ffdd415cf754e53-init
+-- committed
+-- diff
|   +-- dev
|   |   +-- console
|   |   +-- pts
|   |   +-- shm
|   +-- etc
|       +-- hostname
|       +-- hosts
|       +-- mtab -> /proc/mounts
|       +-- resolv.conf
+-- link
+-- lower
+-- work
    +-- work
        +-- #a

7 directories, 9 files
root@ftpd:/var/lib/docker/overlay2#

```

可以看到在 diff 目录中有一些 `/etc/hosts`、`/etc/resolv.conf` 等配置文件，需要这一层的原因是当容器启动的时候，有一些每个容器特定的配置文件（例如 `hostname`），但由于镜像层是只读层无法进行修改，所以就在镜像层之上单独挂载一层 init 层，用户通过修改每个容器对应 init 层中的一些配置文件从而达到修改镜像配置文件的目的，而在 init 层中的配置文件也仅对当前容器生效，通过 `docker commit` 命令创建镜像时也不会提交 init 层。

4.2.8 容器层

最后我们来看一看容器层的构造，刚刚我们在 `mountLayer` 中提到了 **mount-id** 这个文件，而这个文件的内容就是容器层目录的名称，我们通过 `docker inspect [CONTAINER ID]` 命令也可以判断。

```

root@ftpd:/var/lib/docker/overlay2# cat ..image/overlay2/layerdb/mounts/0b6f2be915fb55eb62b8d7721e6c13c7d57eb01db2c7654f8037d87b219cbc52/mount-id
575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53#root@ftpd:/var/lib/docker/overlay2#
root@ftpd:/var/lib/docker/overlay2# docker inspect 0b6f | grep "GraphDriver" -A 10
    "GraphDriver": {
        "Data": {
            "LowerDir": "/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53-init/diff:/var/lib/docker/o
verlay2/8b63e9e3cd41a140d28a5d43b311cd81d746d43b237acb877fbe9b7f010536/diff:/var/lib/docker/overlay2/5b232efd9fafb6d6fb0cc6b6bab13cf08c02a5671a
3c26fa9c2ef659c22c5c6/diff:/var/lib/docker/overlay2/63b1fd65cd292a803d22c89cd3999fc89ebd8334872b0fc5916d85603820/diff:/var/lib/docker/overlay2
/12f1e58e359274373828c367fe8cf26f7a422834b93ad4d3fd5254fbe8af376f7/diff:/var/lib/docker/overlay2/355e046273ab88e4898624c45c3db2ae4806ed2610d1c1dca
d5b1cf5ad7503d/diff:/var/lib/docker/overlay2/e2507fef275983b6e56855cd48c5118500e4d2847405e7e08e44ed4e13f7d25/diff:/var/lib/docker/overlay2/9952aa
3377b45f65ab7c3eca0ef00e18a43e07200136d3ae577fec0e8a0b24/diff:/var/lib/docker/overlay2/99de2110a3d25229444153e7470cbeb358d9f8ee5f979e4cd10bb791
796b2cf/diff:/var/lib/docker/overlay2/0475b6d70638514f0713bb217239a/ec3453d5c4f08912873f0e58589fa43100/diff:/var/lib/docker/overlay2/cel1b636060611
3c23903890567e0d79301c3bdce03d1ed4abe28e822415b0400/diff".
        "MergedDir": "/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53/merged",
        "UpperDir": "/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53/diff",
        "WorkDir": "/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53/work"
    },
    "Name": "overlay2"
},
"Mounts": [
{
root@ftpd:/var/lib/docker/overlay2#

```

可以看到其实容器层的目录与镜像层、init层都在同一目录下，其实也就说明了他们在文件结构上都是相同的。

```

root@ftpd:/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53# ls -l
total 20
drwxr-xr-x 7 root root 4096 Sep 29 20:59 diff
-rw-r--r-- 1 root root 26 Sep 29 20:59 link
-rw-r--r-- 1 root root 318 Sep 29 20:59 lower
drwxr-xr-x 1 root root 4096 Sep 29 20:59 merged
drwx----- 3 root root 4096 Nov 16 23:15 work
root@ftpd:/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53#

```

同样都是这几个文件，但不同的是，我们可以看到在容器层确实有了 merged 这个目录，与我们在文章一开始实现的 overlayFS 是相同的。

merged 目录

```

root@ftpd:/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53# ls merged/
bin boot dev etc home lib lib64 media mnt opt proc root run run.sh sbin srv sys tmp usr var
root@ftpd:/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53#

```

在 merged 目录下展现了完整的 rootfs 文件系统，这就是 overlay2 通过联合挂载技术，将镜像层、init层与容器层挂载到一起呈现的结果，这也是我们通过 `docker exec` 命令进入容器的交互式终端看到的结果，也就是所谓的视图

```

root@ftpd:/var/lib/docker/overlay2/575db3795cb5fff7cadb347f10cfcaad5fd5fe3e5b3e31418ffdd415cf754e53# docker exec -it 0b6f /bin/bash
root@0b6f2be915fb:#

```

5.docker 命令使用

5.1 搜索镜像：

```
[root@localhost ~]# docker search centos:7.9 #带指定版本号
```

```
[root@localhost ~]# docker search centos #不到版本号默认latest
```

5.2 下载镜像：

命令格式： `docker pull 仓库服务器:端口/项目名称/镜像名称:tag(版本号)`

```
[root@localhost ~]# docker pull cnetos
```

```
[root@localhost ~]# docker pull nginx
```

```
[root@localhost ~]# docker pull ubuntu
```

5.3 查看本地镜像：

注意：下载完成的镜像要比下载的大，因为下载完成后会解压。

```
[root@localhost ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	stable	b0e509d07cdc	4 months ago	188MB

REPOSITORY #镜像仓库名称

TAG #镜像版本号

```
IMAGE ID      #镜像唯一ID  
CREATED      #镜像创建时间  
SIZE         #镜像大小
```

5.4 镜像导出：

可以将镜像从本地导出为一个压缩文件，然后复制到其他服务器进行导入使用。

导出方法1：

```
[root@localhost ~]# docker save nginx -o /root/nginx.tar.gz
```

导出方法2：

```
[root@localhost ~]# docker save nginx > /root/nginx.tar.gz
```

5.5 镜像导入：

将镜像导入docker

```
[root@localhost ~]# docker load < /opt/nginx.tar.gz
```

5.6 删除镜像：

命令格式： docker rmi 镜像ID/镜像名称

```
[root@localhost ~]# docker rmi centos:centos7.9.2009
```

5.7 容器操作基础命令：

命令格式： docker run [选项] [镜像名] [shell 命令] [参数]

docker run [参数选项] [镜像名称必须放在所有的选项的后面] [/bin/echo "hello world "] 单次执行，没有自定义容器名称。

直接进入容器，并随机生成容器ID和名称

```
[root@localhost ~]# docker run -it centos:centos7.9.2009 bash
```

```
[root@42a7f1db4fd4 ~]#
```

```
ctrl+p+q #退出容器不注销
```

5.8 显示容器：

```
[root@localhost ~]# docker ps #显示正在运行的容器
```

```
[root@localhost ~]# docker ps -a #显示所有容器包含以及关闭的容器
```

5.8 删除容器：

```
[root@localhost ~]# docker rm centosV1 #删除容器
```

```
[root@localhost ~]# docker rm -f centosV1 #强制删除容器，即使容器在运行
```

5.9 随机映射端口：

```
[root@localhost ~]# docker run -P nginx:stable #前台启动并随机映射本地端口到容器的80
```

前台启动窗口无法进行其他操作，除非退出容器，但是容器也会退出

```
[root@42a7f1db4fd4 /]# exit
[root@localhosts ~]# docker run -P nginx:stable
/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: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/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
2024/09/29 07:39:19 [notice] 1#1: using the "epoll" event method
2024/09/29 07:39:19 [notice] 1#1: nginx/1.26.1
2024/09/29 07:39:19 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/09/29 07:39:19 [notice] 1#1: OS: Linux 4.19.90-24.4.v2101.ky10.x86_64
2024/09/29 07:39:19 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1073741816:1073741816
2024/09/29 07:39:19 [notice] 1#1: start worker processes
2024/09/29 07:39:19 [notice] 1#1: start worker process 29
2024/09/29 07:39:19 [notice] 1#1: start worker process 30
2024/09/29 07:39:19 [notice] 1#1: start worker process 31
2024/09/29 07:39:19 [notice] 1#1: start worker process 32
```

Docker 使用 `-P` 选项时，会随机将容器的端口映射到主机上的一个可用端口。该端口的范围通常是在 **32768 到 60999** 之间，这是操作系统为动态/临时端口分配的默认范围。

具体的端口范围取决于主机系统的配置，Linux 系统可以通过以下命令查看和修改该范围：

```
[root@localhosts ~]# cat /proc/sys/net/ipv4/ip_local_port_range
32768     60999
```

```
[root@localhosts ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
286e32696cbe7      nginx:stable       "/docker-entrypoi...   7 minutes ago    Up 7 minutes          0.0.0.0:49153->80/tcp, :::49153->80/tcp   confident_poitras
```

5.10 指定端口映射：

方式1：本地端口81映射到容器80端口

```
[root@localhosts ~]# docker run -d --name nginx01 -p 81:80 nginx:latest
```

方式2：本地IP:本地端口:容器端口

```
[root@localhosts ~]# docker run -d --name nginx02 -p 172.250.10.233:82:80 nginx:latest
```

方式3：本地IP:本地随机端口:容器端口

```
[root@localhosts ~]# docker run -d --name nginx03 -p 172.250.10.233::80 nginx:latest
```

方式4：本地IP:本地端口:容器端口/协议， 默认为tcp协议

```
[root@localhosts ~]# docker run -d --name nginx04 -p 172.250.10.233:83:80/udp nginx:latest
```

方式5：映射多个端口

```
[root@localhosts ~]# docker run -d --name nginx05 -p 85:80/tcp -p 443:443/tcp -p 53:53/udp nginx:latest
```

```
[root@localhosts ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
f72c14d7c748      nginx:latest       "/docker-entrypoi...   38 seconds ago    Up 37 seconds          172.250.10.233:49153->80/tcp
3bedef1f155      nginx:latest       "/docker-entrypoi...   About a minute ago  Up About a minute  0.0.0.0:53->53/udp, :::53->53/udp, 0.0.0.0:443->443/tcp, :::443->443/tcp, 0.0.0.0:85->80/tcp, :::85->80/tcp   nginx03
b76162373642      nginx:latest       "/docker-entrypoi...   3 minutes ago     Up 3 minutes          80/tcp, 172.250.10.233:83->80/udp
b214627e4bef      nginx:latest       "/docker-entrypoi...   25 minutes ago    Up 25 minutes         172.250.10.233:82->80/tcp
27e20bf67eb5      nginx:latest       "/docker-entrypoi...   29 minutes ago    Up 29 minutes          0.0.0.0:81->80/tcp, :::81->80/tcp   nginx05
nginx05
nginx04
nginx02
nginx01
```

查看Nginx容器访问日志：

```
[root@localhosts ~]# docker logs nginx01 #一次查看
```

```
[root@localhosts ~]# docker logs -f nginx01 #持续查看
```

5.11 查看容器已经映射的端口：

```
[root@localhosts ~]# docker port nginx01  
80/tcp -> 0.0.0.0:81  
80/tcp -> :::81
```

5.12 自定义容器名称：

```
[root@localhosts ~]# docker run -it --name nginx07 nginx:latest
```

5.13 后台启动容器：

```
[root@localhosts ~]# docker run -d -P --name nginx08 nginx:latest  
42edec6f61aca6c8c2d3a3e61d91e1384b80aeaf9e148e134d98afde06951cbc
```

```
[root@localhosts ~]# docker run -d -P --name nginx08 nginx:latest  
42edec6f61aca6c8c2d3a3e61d91e1384b80aeaf9e148e134d98afde06951cbc  
[root@localhosts ~]# docker ps -a  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
42edec6f61ac nginx:latest "/docker-entrypoint..." 22 seconds ago Up 21 seconds 0.0.0.0:49154->80/tcp, :::49154->80/tcp nginx08
```

5.14 创建容器并进入容器

```
[root@localhosts ~]# docker run -i -t --name centos02 centos:latest /bin/bash
```

[root@006056fa0697 /]# 创建容器后直接进入，输入exit退出后容器会关闭

```
[root@006056fa0697 /]# ps -aux  
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND  
root 1 0.1 0.1 11840 2848 pts/0 Ss 08:44 0:00 /bin/bash
```

5.15 单次运行：

容器退出后自动删除

```
[root@localhosts ~]# docker run -it --rm --name nginx09
```

5.16 传递运行命令：

```
[root@localhosts ~]# docker run -d centos /usr/bin/tail -f '/etc/hosts'
```

```
[root@localhosts ~]# docker ps -a  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
6be92d72c5b8 centos "/usr/bin/tail -f /e..." 13 seconds ago Up 12 seconds magical_napier
```

5.17 容器的启动和关闭：

```
[root@localhosts ~]# docker stop nginx01
```

```
[root@localhosts ~]# docker start nginx01
```

5.18 进入正在运行的容器：

```
[root@localhosts ~]# docker exec -it nginx08 /bin/bash
```

5.19 批量关闭容器：

```
[root@localhosts ~]# docker stop $(docker ps -a -q) #正常关闭所有运行中的容器
```

```
[root@localhosts ~]# docker kill $(docker ps -a -q) #强制关闭所有运行中的容器
```

5.19 批量删除容器：

```
[root@localhosts ~]# docker rm -f $(docker ps -aq -f status=exited) #批量删除已退出的容器  
[root@localhosts ~]# docker rm -f $(docker ps -aq) #批量删除所有容器
```

[harbor官方下载地址](#)

harbor当中的harbor.yml 这个配置文件中的配置参数后面不要有空格，不然在在使用docker login 连接horbor 仓管会存在一些问题。

```
# Configuration file of Harbor  
  
# The IP address or hostname to access admin UI and registry service.  
# DO NOT use localhost or 127.0.0.1, because Harbor needs to be accessed by external clients.  
hostname: 172.250.1.83
```



6.构建镜像

6.1 手动制作YUM版本nginx镜像：

Docker制作类似于虚拟机的模板制作，即按照公司的实际业务需求将需要安装的软件、相关配置等基础环境配置完成，然后将虚拟机再提交为模板，最后再批量从模板批量创建新的虚拟机，这样可以极大的简化业务中相同环境的虚拟机运行环境的部署工作，Docker的镜像制作分为手动制作和自动制作(基于DockerFile)，企业通常都是基于Dockerfile制作镜像，其中手动制作镜像步骤具体如下：

6.2 导入镜像并初始化系统：

基于某个基础镜像之上重新制作，因此需要先有一个基础镜像，本次使用官方提供的centos镜像为基础：

```
[root@localhosts ~]# docker load -i /opt/exam/kylin/APP_images/kylin-10-sp1-x86_64-b04.07.11.tar.xz  
[root@localhosts ~]# docker run -it kylin-10-x86_64:v10-b04.07.11 /bin/bash  
[root@ecc82802ca9d/]# yum repolist -v
```

6.3 使用Yum安装Nginx

```
[root@ecc82802ca9d /]# yum -y install nginx #使用YUM安装nginx  
[root@ecc82802ca9d /]# yum install -y vim wget pcre pcre-devel zlib zlib-devel openssl openssl-devel  
iproute net-tools iotop #安装常用命令
```

6.4 关闭Nginx后台运行:

```
[root@ecc82802ca9d /]# vim /etc/nginx/nginx.conf  
  
user root;  
worker_processes auto;  
error_log /var/log/nginx/error.log;  
pid /run/nginx.pid;\  
  
daemon off; #关闭Nginx后台运行
```

6.5 自定义web页面:

```
[root@ecc82802ca9d /]# echo "Docker Yum Nginx" > /usr/share/nginx/html/index.html  
[root@ecc82802ca9d /]# cat /usr/share/nginx/html/index.html  
  
Docker Yum Nginx #自定义web界面
```

6.6 提交为镜像:

在开个终端在宿主机基于容器ID提交为镜像

```
[root@localhosts ~]# docker commit -m "first commit" ecc82802ca9d kynginx:v1
```

6.7 从镜像启动容器:

```
[root@localhosts ~]# docker run -d -p 8088:80 --name nginxV1 kynginx:v1 /usr/sbin/nginx  
[root@localhosts ~]# curl localhost:8088  
  
Docker Yum Nginx # 出现一下界面
```

7. 使用DockerFile制作镜像

DockerFile可以说是一种可以被Docker程序解释的脚本，DockerFile是由一条条的命令组成的，每条命令对应linux下面的一条命令，Docker程序将这些DockerFile指令再翻译成真正的linux命令，其有自己的书写方式和支持的命令，Docker程序读取DockerFile并根据指令生成Docker镜像，相比手动制作镜像的方式，DockerFile更能直观的展示镜像是怎么产生的，有了写好的各种各样DockerFile文件，当后期某个镜像有额外的需求时，只要在之前的DockerFile添加或者修改相应的操作即可重新生成新的Docker镜像，避免了重复手动制作镜像的麻烦，具体如下：

官方DockerFile使用

以下是 Dockerfile 中常用指令的解释，以表格形式列出：

指令	作用	示例
FROM	指定基础镜像，每个 Dockerfile 必须有且只能有一个 FROM 指令。	<code>FROM ubuntu:20.04</code>
ADD	将本地或远程文件添加到镜像中，支持解压归档文件。	<code>ADD ./app.tar.gz /app/</code>

指令	作用	示例
COPY	将文件从宿主机复制到镜像中，不支持解压归档文件。	<code>COPY ./app /app</code>
ENV	设置环境变量，容器运行时生效。	<code>ENV APP_ENV=production</code>
EXPOSE	声明容器暴露的端口号（仅为文档性质，实际暴露需通过 <code>docker run</code> 配置端口映射）。	<code>EXPOSE 8080</code>
LABEL	为镜像添加元数据（如版本、维护者信息）。	<code>LABEL version="1.0" maintainer="youremail@example.com"</code>
STOP SIGNAL	指定容器接收到的停止信号，控制容器如何优雅停止。	<code>STOP SIGNAL SIGKILL</code>
USER	指定容器内运行的用户，默认是 <code>root</code> ，但可以更改为其他非 <code>root</code> 用户以提升安全性。	<code>USER nginx</code>
VOLUME	声明一个挂载点，指定路径将作为卷，便于数据持久化或与宿主机共享。	<code>VOLUME /data</code>
WORKDIR	设置工作目录，后续指令将在此目录下执行。	<code>WORKDIR /app</code>
RUN	在构建镜像时执行命令，通常用于安装软件包或运行脚本。	<code>RUN apt-get update && apt-get install -y nginx</code>
CMD	指定容器启动时的默认命令，通常是容器中应用的主进程。	<code>CMD ["nginx", "-g", "daemon off;"]</code>

构建centos基础镜像：

```

1 # 创建项目目录
2 mkdir /opt/dockerfile/{web/{nginx,tomcat,jdk,apache},system/{centos,ubuntu,debian}}
-pv
3 # 进入到centos系统目录中
4 cd /opt/dockerfile/system/centos
5 # 编辑dockerfile
6 vim dockerfile
7 from centos:centos7.9.2009
8 run curl -o /etc/yum.repos.d/CentOS-Base.repo
https://mirrors.aliyun.com/repo/Centos-7.repo && yum repolist -v
9 run yum -y install vim wget tree lrzsz gcc gcc-c++ automake pcre pcre-devel zlib
zlib-devel openssl openssl-devel net-tools make iproute iotop net-tools \
&& groupadd www -g 2022 \
11 && useradd www -u 2020 -g www
12 # 容器时区和宿主机保持一样
13 run rm -rf /etc/localtime && ln -snf /usr/share/zoneinfo/Asia/Shanghai
/etc/localtime
14
15 vim centos.sh
16 #!/bin/bash
17 docker build -t centos7.9.2009:v1 .
18 chmod 777 *.sh

```

```
19 ./centos.sh
```

构建JDK基础镜像：

```
1 # 进入web目录下的java-jdk目录
2 cd /opt/dockerfile/web/jdk
3 cp /etc/profile .
4 ls
5 jdk-8u401-linux-i586.tar.gz /profile
6
7 # 编辑dockerfile
8 vim dockerfile
9 from centos7.9.2009:v1
10 add jdk-8u401-linux-i586.tar.gz /usr/local/src/
11 run ln -sv /usr/local/src/jdk1.8.0_401 /usr/local/jdk
12 add profile /etc/profile
13 env JRE_HOME="$JAVA_HOME/jre"
14 env JAVA_HOME="/usr/local/jdk"
15 env CLASSPATH="$JAVA_HOME/lib/:$JRE_HOME/lib/"
16 env PATH="$PATH:$JAVA_HOME/bin"
17
18 #准备构建脚本
19 vim centos-ava-jdk.sh
20 docker build -t centos:jdk
21 ./app_java-jdk.sh
22
23 docker run -it --rm centos:jdk bash
24 java -version
```

构建tomcat基础镜像

```
1 # 编辑dockerfile
2 vim dockerfile
3 FROM centos:jdk
4
5 FROM centos:jdk
6 # 设置时区、语言和终端环境变量
7 ENV TZ "Asia/Shanghai"
8 ENV LANG en_US.UTF-8
9 ENV TERM xterm
10 # 设置时区、语言和终端环境变量
11 ENV TOMCAT_MAJOR_VERSION 8
12 ENV TOMCAT_MINOR_VERSION 8.5.100
13 ENV CATALINA_HOME /apps/tomcat
14 ENV APP_DIR ${CATALINA_HOME}/webapps
15
16 RUN mkdir /apps
17 ADD apache-tomcat-8.5.100.tar.gz /apps
18 #将解压后的 Tomcat 目录软链接到 /apps/tomcat
19 RUN ln -sv /apps/apache-tomcat-8.5.100 /apps/tomcat
20
21 # vim build-tomcat.sh
22#!/bin/bash
23 docker build -t tomcat:v1 .
24
25 #给脚本赋予权限
26 chmod 777 *.sh
27 #执行脚本
```

```
28 ./build-tomcat.sh
```

构建tomcat业务镜像

```
1 # 编辑dockerfile
2 vim dockerfile
3 FROM tomcat:v1
4 ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh
5 ADD myapp/* /apps/tomcat/webapps/myapp/
6 RUN useradd tomcat && chown tomcat.tomcat /apps/ -R
7
8 EXPOSE 8080 8009
9
10 CMD ["/apps/tomcat/bin/run_tomcat.sh"]
11
12 #准备自定义myapp页面:
13 mkdir myapp
14 echo "Tomcat web-01" > myapp/index.html
15 #准备启动脚本
16 vim run_tomcat.sh
17 #!/bin/bash
18 echo "1.1.1.1 abc.test.com" >> /etc/hosts
19 echo "anmeserver 223.5.5.5" > /etc/resolv.conf
20
21 su - tomcat -c "/apps/tomcat/bin/catalina.sh start"
22 su - tomcat -c "tail -f /etc/hosts"
23
24 # 编辑镜像构建脚本
25 vim build-tomcat.sh
26 #!/bin/bash
27 docker build -t tomcat-web:app1 .
28
29 #给脚本赋予权限
30 chmod 777 *.sh
31 #执行脚本
32 ./build-tomcat.sh
33
34 docker run -it -d -p 8081:8080 tomcat-web:app1
35
36 curl localhost:8081/myapp/
```

构建nginx基础镜像

```
1 # 官网下载nginx源码包
2 wget https://nginx.org/download/nginx-1.18.0.tar.gz ./
3 vim dockerfile
4
5 # 编辑dockerfile
6 FROM centos7.9.2009:v1
7 ADD nginx-1.18.0.tar.gz /usr/local
8 RUN cd /usr/local/nginx-1.18.0 \
9     && ./configure --prefix=/usr/local/nginx --with-http_sub_module \
10    && make \
11    && make install
12 COPY index.html /usr/local/nginx/html
13 CMD ["/usr/local/nginx/sbin/nginx", "-g", "daemon off;"]
14
15 # 编辑镜像构建脚本
16 vim build-nginx.sh
```

```

17 #!/bin/bash
18 docker build -t nginx-image:v1 .
19
20 chmod 777 *.sh
21 ./build-nginx.sh
22 docker run -itd --name nginx -p 80:80 nginx-image:v1
23
24 curl localhost

```

构建haproxy镜像

```

1 mkdir /opt/web/haproxy
2 cd /opt/web/haproxy
3
4 # 下载haproxy 包
5 wget https://www.haproxy.org/download/2.8/src/haproxy-2.8.9.tar.gz
6
7 # 编辑dockerfile
8 vim dockerfile
9 FROM centos7.9.2009:v1
10 RUN yum -y install glibc glibc-devel zlib-devel systemd-devel wget vim wget tree
    lrzsz gcc gcc-c++ automake pcre pcre-devel zlib zlib-devel openssl openssl-devel
    net-tools make iproute iotop net-tools
11 ADD haproxy-2.8.9.tar.gz /usr/local/src
12 RUN cd /usr/local/src/haproxy-2.8.9 \
    && make ARCH=x86_64 TARGET=linux-glibc USE_PCRE=1 USE_OPENSSL=1 USE_ZLIB=1
    USE_SYSTEMD=1 USE_CPU_AFFINITY=1 PREFIX=/usr/local/haproxy
    && make install PREFIX=/usr/local/haproxy
    && cp haproxy /usr/sbin/
13
14 ADD proxy.cfg /etc/haproxy/
15 EXPOSE 80 9999
16 CMD ["/usr/local/haproxy/sbin/run_haproxy.sh"]
17
18 # 编辑haproxy.cfg 配置文件
19 vim haproxy.cfg
20 # 全局配置部分
21 global
22 chroot /usr/local/haproxy #将进程的锁到/usr/local/haproxy目录，限制进程的文件访问范围。
23 #stats socket /var/lib/haproxy/haproxy.sock mode 600 level admin
24 uid 99
25 gid 99
26 daemon #将HAProxy 后台运行
27 # nbproc 1 #指定启动的进程数，此处为 1 个进程
28 pidfile /usr/local/haproxy/haproxy.pid #指定HAProxy进程的PID文件路径
29 log 127.0.0.1 local3 info #配置日志的输出到本地，日志级别为 local3，输出信息级别
30
31 # 默认配置部分
32 defaults
33 option http-keep-alive #启用 HTTP 连接保持活动状态，允许客户端和服务器之间保持长连接
34 option forwardfor # 头部，将原始客户端 IP 地址添加到请求头中，用于透明代理。
35 mode http # 设置默认的代理模式为 HTTP，用于处理 HTTP 请求和响应
36 timeout connect 300000ms # 设置客户端与服务器建立连接的超时时间为 300 秒
37 timeout client 300000ms # 设置客户端的超时时间为 300 秒，包括接收客户端请求和发送响应的时间
38 timeout server 300000ms # 设置服务器的超时时间为 300 秒，包括与后端服务器建立连接和接收响应的时间
39
40 # 监听配置部分
41 listen stats

```

```

45 mode http
46 bind 0.0.0.0:9999 #监听地址和端口
47 stats enable #启用 HAProxy 的统计功能，允许通过特定URI访问统计信息
48 log global # 指定使用全局定义的日志设置
49 stats uri docker run -it --name haproxy-v1 -p 81:80 -p 9999:9999 centos-haproxy:v1
50 s # 设置统计页面的URI为/haproxy-status，通过访问该URI可以查看 HAProxy 的统计信息
51 stats auth haadmin:123456 # 设置访问统计页面时的认证用户名和密码，用户名为 haadmin，密码为
52 123456
53
54 listen web_port #
55 bind 0.0.0.0:80 #监听地址和端口
56 mode http
57 log global
58 balance roundrobin # 设置负载均衡算法为轮询模式
59 server web1 172.250.1.112:8080 check inter 3000 fall 2 rise 5
60 server web2 172.250.1.79:8080 check inter 3000 fall 2 rise 5
61 # 定义后端服务器 web1，指定服务器地址为 192.168.7.101:8080，并设置健康检查参数。
62
63 # 编辑容器启动脚本
64 vim run_haproxy.sh
65 #!/bin/bash
66 /usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg
67 tail -f /etc/hosts
68
69 # 编辑镜像构建脚本
70 vim build-haproxy.sh
71 #!/bin/bash
72 docker build -t centos-haproxy:v1 .
73
74 chmod 777 *.sh
75 ./build-haproxy.sh
76
77 docker run -itd --name haproxy-v1 -p 80:80 -p 9999:9999 -v
78 /opt/web/haproxy.haproxy.cfg:/usr/local/haproxy.haproxy.cfg centos-haproxy:v1
79
80 curl http://172.250.1.113:81/myapp/
81
82 http://172.250.1.112:9999/haproxy-status

```

搭建单台服务器站点网站

1、构建tomcat业务镜像

```

1 # 构建tomcat1 业务镜像
2 cd /opt/web/tomcat-app1
3 # 编辑dockerfile
4 vim dockerfile
5 FROM tomcat:v1
6 ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh
7 ADD myapp/* /apps/tomcat/webapps/myapp/
8 RUN useradd tomcat && chown tomcat.tomcat /apps/ -R
9 CMD ["/apps/tomcat/bin/run_tomcat.sh"]
10
11 #准备自定义myapp页面:
12 mkdir myapp
13 echo "Tomcat web-01" > myapp/index.html
14 #准备启动脚本

```

```

15 vim run_tomcat.sh
16 #!/bin/bash
17 echo "1.1.1.1 abc.test.com" >> /etc/hosts
18 echo "nameserver 223.5.5.5" > /etc/resolv.conf
19 su - tomcat -c "/apps/tomcat/bin/catalina.sh start"
20 su - tomcat -c "tail -f /etc/hosts"
21
22 # 编辑镜像构建脚本
23 vim build-tomcat.sh
24 #!/bin/bash
25 docker build -t tomcat-web:app1 .
26
27 #给脚本赋予权限
28 chmod 777 *.sh
29 #执行脚本
30 ./build-tomcat.sh
31
32 # tomcat-1容器镜像运行
33 docker run -it -d -p 8081:8080 -name tomcat-web-v1 tomcat-web:app1
34 curl localhost:8081/myapp/
35
36
37 # 构建tomcat业务2镜像
38 cd /opt/web/tomcat-app2
39 cp -r /opt/web/tomcat-app1/* ./
40
41 # 修改网页内容
42 echo "Tomcat web-02" > myapp/index.html
43
44 # tomcat-2容器镜像运行
45 docker run -it -d -name tomcat-web-v2 -p 8082:8080 -v /opt/web/tomcat-
app2/myapp/index.html:/apps/tomcat/webapps/myapp/index.html tomcat-web:app1
46 curl localhost:8082/myapp/

```

2、构建nginx动静分离镜像

```

1 cd /opt/web/nginx
2 # 官网下载nginx源码包
3 wget https://nginx.org/download/nginx-1.18.0.tar.gz ./
4 vim dockerfile
5
6 # 编辑dockerfile
7 FROM centos7.9.2009:v1
8 ADD nginx-1.18.0.tar.gz /usr/local
9 RUN cd /usr/local/nginx-1.18.0 \
10 && ./configure --prefix=/usr/local/nginx --with-http_sub_module \
11 && make \
12 && make install
13 COPY index.html /usr/local/nginx/html
14 CMD ["/usr/local/nginx/sbin/nginx", "-g", "daemon off;"]
15
16 # 构建静态网页内容
17 echo "hello world nginx" > index.html
18
19 # 编辑镜像构建脚本
20 vim build-nginx.sh
21 #!/bin/bash
22 docker build -t nginx-image:v1 .
23 # 脚本赋予权限执行
24 chmod 777 *.sh

```

```

25 ./build-nginx.sh
26
27 # 编辑nginx.conf配置文件
28 grep -v "#" nginx.conf | grep -v "^\$"
29 worker_processes 1;
30 events {
31     worker_connections 1024;
32 }
33 http {
34     include mime.types;
35     default_type application/octet-stream;
36     sendfile on;
37     keepalive_timeout 65;
38
39 upstream tomcat_webserver {
40     server 172.250.1.113:8081;
41     server 172.250.1.113:8082;
42 }
43     server {
44         listen 80;
45         server_name localhost;
46         location /myapp {
47             proxy_pass http://tomcat_webserver;
48             proxy_set_header Host $host;
49             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
50             proxy_set_header X-Real-IP $remote_addr;
51         }
52         location / {
53             root html;
54             index index.html index.htm;
55         }
56         error_page 500 502 503 504 /50x.html;
57         location = /50x.html {
58             root html;
59         }
60     }
61 }
62
63 #nginx容器镜像运行
64 docker run -itd --name nginx-web-v1 -p 8080:80 -v
65 /opt/web/nginx/nginx.conf:/usr/local/nginx/conf/nginx.conf nginx-image:v1
66
67 # 验证是否动静分离
68 curl localhost:8080
69 curl localhost:8080/myapp/

```

3、配置haproxy容器镜像

```

1 mkdir /opt/web/haproxy
2 cd /opt/web/haproxy
3
4 # 下载haproxy 包
5 wget https://www.haproxy.org/download/2.8/src/haproxy-2.8.9.tar.gz
6
7 # 编辑dockerfile
8 vim dockerfile
9 FROM centos7.9.2009:v1
10 RUN yum -y install glibc glibc-devel zlib-devel systemd-devel wget vim wget tree
11 lrzsz gcc gcc-c++ automake pcre pcre-devel zlib zlib-devel openssl openssl-devel
12 net-tools make iproute iotop net-tools

```

```
11 ADD haproxy-2.8.9.tar.gz /usr/local/src
12 RUN cd /usr/local/src/haproxy-2.8.9 \
13 && make ARCH=x86_64 TARGET=linux-glibc USE_PCRE=1 USE_OPENSSL=1 USE_ZLIB=1
14 && USE_SYSTEMD=1 USE_CPU_AFFINITY=1 PREFIX=/usr/local/haproxy
15 && make install PREFIX=/usr/local/haproxy
16 && cp haproxy /usr/sbin/
17 ADD haproxy.cfg /etc/haproxy/
18 EXPOSE 80 9999
19 CMD ["/usr/local/haproxy/sbin/run_haproxy.sh"]

20 # 编辑haproxy.cfg 配置文件
21 vim haproxy.cfg
22 # 全局配置部分
23 global
24 maxconn 100000
25 chroot /usr/local/haproxy #将进程的锁到/usr/local/haproxy目录，限制进程的文件访问范围。
26 #stats socket /var/lib/haproxy/haproxy.sock mode 600 level admin
27 uid 99
28 gid 99
29 daemon #将HAProxy 后台运行
30 log 127.0.0.1 local3 info #配置日志的输出到本地，日志级别为 local3，输出信息级别
31
32 # 默认配置部分
33 defaults
34 option http-keep-alive #启用 HTTP 连接保持活动状态，允许客户端和服务器之间保持长连接
35 option forwardfor # 头部，将原始客户端 IP 地址添加到请求头中，用于透明代理。
36 maxconn 100000
37 mode tcp # 设置默认的代理模式为 HTTP，用于处理 HTTP 请求和响应
38 timeout connect 300000ms # 设置客户端与服务器建立连接的超时时间为 300 秒
39 timeout client 300000ms # 设置客户端的超时时间为 300 秒，包括接收客户端请求和发送响应的时间
40 timeout server 300000ms # 设置服务器的超时时间为 300 秒，包括与后端服务器建立连接和接收响应
41 的时间
42 # 监听配置部分
43 listen stats
44 mode http
45 bind 0.0.0.0:9999 #监听地址和端口
46 stats enable #启用 HAProxy 的统计功能，允许通过特定URI访问统计信息
47 log global # 指定使用全局定义的日志设置
48 stats uri /haproxy-status # 通过访问该URI可以查看 HAProxy 的统计信息
49 stats auth haadmin:123456 # 设置访问用户名和密码，用户名为 haadmin，密码为 123456
50
51 frontend docker_nginx_web #
52 bind 0.0.0.0:80 #监听地址和端口
53 mode http
54 default_backend docker_nginx_hosts
55
56 backend docker_nginx_hosts
57 #balance roundrobin # 设置负载均衡算法为轮询模式
58 server web1 172.250.1.113:8080 check inter 3000 fall 2 rise 5
59 # 定义后端服务器 web1，指定服务器地址为 192.168.7.101:8080，并设置健康检查参数。
60
61
62 # 编辑容器启动脚本
63 vim run_haproxy.sh
64#!/bin/bash
65 /usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg
66 tail -f /etc/hosts
67
68 # 编辑镜像构建脚本
```

```

69 vim build-haproxy.sh
70 #!/bin/bash
71 docker build -t centos-haproxy:v1 .
72 # 赋予脚本权限
73 chmod 777 *.sh
74 ./build-haproxy.sh
75 # haproxy容器镜像运行
76 docker run -itd --name haproxy-v1 -p 80:80 -p 9999:9999 -v
    /opt/web/haproxy/haproxy.cfg:/usr/local/haproxy/haproxy.cfg centos-haproxy:v1
77
78 curl http://172.250.1.113:80/myapp/
79 http://172.250.1.113:9999/haproxy-status

```

搭建多服务器小型站点网站

1、构建tomcat-业务镜像

```

1 # dockers-v1主机
2 # 编辑dockerfile
3 vim dockerfile
4 FROM tomcat:v1
5 ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh
6 ADD myapp/* /apps/tomcat/webapps/myapp/
7 RUN useradd tomcat && chown tomcat.tomcat /apps/ -R
8 CMD ["/apps/tomcat/bin/run_tomcat.sh"]

9
10 #准备自定义myapp页面:
11 mkdir myapp
12 echo "Tomcat web-01" > myapp/index.html
13
14 #准备启动脚本
15 vim run_tomcat.sh
16 #!/bin/bash
17 su - tomcat -c "/apps/tomcat/bin/catalina.sh start"
18 su - tomcat -c "tail -f /etc/hosts"
19
20 # 编辑镜像构建脚本
21 vim build-tomcat.sh
22 #!/bin/bash
23 docker build -t tomcat-web:app1 .
24
25 #给脚本赋予权限执行
26 chmod 777 *.sh
27 ./build-tomcat.sh
28
29 # tomcat-1容器镜像运行
30 docker run -it -d -p 8081:8080 -name tomcat-web-v1 tomcat-web:app1
31 curl localhost:8081/myapp/
32
33 =====
34 =====
35 # dockers-v2主机
36 # 编辑dockerfile
37 vim dockerfile
38 FROM tomcat:v1
39 ADD run_tomcat.sh /apps/tomcat/bin/run_tomcat.sh
40 ADD myapp/* /apps/tomcat/webapps/myapp/
41 RUN useradd tomcat && chown tomcat.tomcat /apps/ -R
42 CMD ["/apps/tomcat/bin/run_tomcat.sh"]

```

```

43
44 #准备自定义myapp页面:
45 mkdir myapp
46 echo "Tomcat web-02" > myapp/index.html
47
48 #准备启动脚本
49 vim run_tomcat.sh
50 #!/bin/bash
51 su - tomcat -c "/apps/tomcat/bin/catalina.sh start"
52 su - tomcat -c "tail -f /etc/hosts"
53
54 # 编辑镜像构建脚本
55 vim build-tomcat.sh
56 #!/bin/bash
57 docker build -t tomcat-web:app2 .
58
59 #给脚本赋予权限执行
60 chmod 777 *.sh
61 ./build-tomcat.sh
62
63 # tomcat-2容器镜像运行
64 docker run -it -d -p 8081:8080 -name tomcat-web-v2 tomcat-web:app2
65 curl localhost:8081/myapp/

```

2、构建nginx-动静分离镜像

```

1 # dockers-v1主机
2 # 官网下载nginx源码包
3 wget https://nginx.org/download/nginx-1.18.0.tar.gz ./
4 vim dockerfile
5
6 # 编辑dockerfile
7 FROM centos7.9.2009:v1
8 ADD nginx-1.18.0.tar.gz /usr/local
9 RUN cd /usr/local/nginx-1.18.0 \
10 && ./configure --prefix=/usr/local/nginx --with-http_sub_module \
11 && make \
12 && make install
13 COPY index.html /usr/local/nginx/html
14 CMD ["/usr/local/nginx/sbin/nginx", "-g", "daemon off;"]
15
16 # 编辑镜像构建脚本
17 vim build-nginx.sh
18 #!/bin/bash
19 docker build -t nginx-image:v1 .
20 # 脚本赋予权限执行
21 chmod 777 *.sh
22 ./build-nginx.sh
23
24 # 编辑nginx.conf配置文件
25 grep -v "#" nginx.conf | grep -v "^$"
26 worker_processes 1;
27 events {
28     worker_connections 1024;
29 }
30 http {
31     include mime.types;
32     default_type application/octet-stream;
33     sendfile on;
34     keepalive_timeout 65;

```

```
35
36 upstream tomcat_webserver {
37     server 172.250.1.113:8081;
38     server 172.250.1.112:8081;
39 }
40     server {
41         listen      80;
42         server_name localhost;
43         location /myapp {
44             proxy_pass http://tomcat_webserver;
45             proxy_set_header Host $host;
46             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
47             proxy_set_header X-Real-IP $remote_addr;
48         }
49         location / {
50             root   html;
51             index  index.html index.htm;
52         }
53         error_page  500 502 503 504  /50x.html;
54         location = /50x.html {
55             root   html;
56         }
57     }
58 }
59
60 #nginx容器镜像运行
61 docker run -itd --name nginx-web-v1 -p 8080:80 -v
62 /opt/web/nginx/nginx.conf:/usr/local/nginx/conf/nginx.conf nginx-image:v1
63
64 # 验证是否动静分离
65 curl localhost:8080
66 curl localhost:8080/myapp/
67 =====
68 =====
69 # dockers-v2主机
70 # 官网下载nginx源码包
71 wget https://nginx.org/download/nginx-1.18.0.tar.gz  ../
72 vim dockerfile
73
74 # 编辑dockerfile
75 FROM centos7.9.2009:v1
76 ADD nginx-1.18.0.tar.gz  /usr/local
77 RUN cd /usr/local/nginx-1.18.0 \
78 && ./configure --prefix=/usr/local/nginx --with-http_sub_module \
79 && make \
80 && make install
81 COPY index.html /usr/local/nginx/html
82 CMD ["/usr/local/nginx/sbin/nginx", "-g", "daemon off;"]
83
84 # 编辑镜像构建脚本
85 vim build-nginx.sh
86 #!/bin/bash
87 docker build -t nginx-image:v2 .
88 # 脚本赋予权限执行
89 chmod 777 *.sh
90 ./build-nginx.sh
91
92 # 编辑nginx.conf配置文件
```

```

93 grep -v "#" nginx.conf | grep -v "^\$"
94 worker_processes 1;
95 events {
96     worker_connections 1024;
97 }
98 http {
99     include mime.types;
100    default_type application/octet-stream;
101    sendfile on;
102    keepalive_timeout 65;
103
104 upstream tomcat_webserver {
105     server 172.250.1.113:8081;
106     server 172.250.1.112:8081;
107 }
108     server {
109         listen 80;
110         server_name localhost;
111         location /myapp {
112             proxy_pass http://tomcat_webserver;
113             proxy_set_header Host $host;
114             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
115             proxy_set_header X-Real-IP $remote_addr;
116         }
117         location / {
118             root html;
119             index index.html index.htm;
120         }
121         error_page 500 502 503 504 /50x.html;
122         location = /50x.html {
123             root html;
124         }
125     }
126 }
127
128 #nginx容器镜像运行
129 docker run -itd --name nginx-web-v2 -p 8080:80 -v
130 /opt/web/nginx/nginx.conf:/usr/local/nginx/conf/nginx.conf nginx-image:v2
131
132 # 验证是否动静分离
133 curl localhost:8080
134 curl localhost:8080/myapp/

```

3、配置haprox容器镜像

```

1 # docker-server01主机
2 mkdir /opt/web/haproxy
3 cd /opt/web/haproxy
4
5 # 下载haproxy 包
6 wget https://www.haproxy.org/download/2.8/src/haproxy-2.8.9.tar.gz
7
8 # 编辑dockerfile
9 vim dockerfile
10 FROM centos7.9.2009:v1
11 RUN yum -y install glibc glibc-devel zlib-devel systemd-devel wget vim wget tree
12 lrzsz gcc gcc-c++ automake pcre pcre-devel zlib zlib-devel openssl openssl-
13 devel net-tools make iproute iotop net-tools
14 ADD haproxy-2.8.9.tar.gz /usr/local/src
15 RUN cd /usr/local/src/haproxy-2.8.9 \

```

```
14 && make ARCH=x86_64 TARGET=linux-glibc USE_PCRE=1 USE_OPENSSL=1 USE_ZLIB=1  
15 USE_SYSTEMD=1 USE_CPU_AFFINITY=1 PREFIX=/usr/local/haproxy  
16 && make install PREFIX=/usr/local/haproxy  
17 & cp haproxy /usr/sbin/  
18 ADD aproxy.cfg /etc/haproxy/  
19 EXPOSE 80 9999  
20 CMD ["/usr/local/haproxy/sbin/run_haproxy.sh"]  
21 # 编辑haproxy.cfg 配置文件  
22 vim haproxy.cfg  
23 # 全局配置部分  
24 global  
25 maxconn 100000  
26 chroot /usr/local/haproxy #将进程的锁到/usr/local/haproxy目录，限制进程的文件访问范围。  
27 #stats socket /var/lib/haproxy/haproxy.sock mode 600 level admin  
28 uid 99  
29 gid 99  
30 daemon #将HAProxy 后台运行  
31 # nbproc 1 #指定启动的进程数，此处为 1 个进程  
32 log 127.0.0.1 local3 info #配置日志的输出到本地，日志级别为 local3，输出信息级别  
33  
34 # 默认配置部分  
35 defaults  
36 option http-keep-alive #启用 HTTP 连接保持活动状态，允许客户端和服务器之间保持长连接  
37 option forwardfor # 头部，将原始客户端 IP 地址添加到请求头中，用于透明代理。  
38 maxconn 100000  
39 mode tcp # 设置默认的代理模式为 HTTP，用于处理 HTTP 请求和响应  
40 timeout connect 300000ms # 设置客户端与服务器建立连接的超时时间为 300 秒  
41 timeout client 300000ms # 设置客户端的超时时间为 300 秒，包括接收客户端请求和发送响应的时间  
42 timeout server 300000ms # 设置服务器的超时时间为 300 秒，包括与后端服务器建立连接和接收响应的时间  
43  
44 # 监听配置部分  
45 listen stats  
46 mode http  
47 bind 0.0.0.0:9999 #监听地址和端口  
48 stats enable #启用 HAProxy 的统计功能，允许通过特定URI访问统计信息  
49 log global # 指定使用全局定义的日志设置  
50 stats uri /haproxy-status # 通过访问该URI可以查看 HAProxy 的统计信息  
51 stats auth haadmin:123456 # 设置访问用户名和密码，用户名为 haadmin，密码为 123456  
52  
53 frontend docker_nginx_web #  
54 bind 0.0.0.0:80 #监听地址和端口  
55 mode http  
56 default_backend docker_nginx_hosts  
57  
58 backend docker_nginx_hosts  
59 #balance roundrobin # 设置负载均衡算法为轮询模式  
60 server web1 172.250.1.113:8080 check inter 3000 fall 2 rise 5  
61 server web2 172.250.1.112:8080 check inter 3000 fall 2 rise 5  
62 # 定义后端服务器 指定服务器地址，并设置健康检查参数。  
63  
64  
65 # 编辑容器启动脚本  
66 vim run_haproxy.sh  
67 #!/bin/bash  
68 /usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg  
69 tail -f /etc/hosts  
70
```

```
71 # 编辑镜像构建脚本
72 vim build-haproxy.sh
73#!/bin/bash
74 docker build -t centos-haproxy:v1 .
75 # 赋予脚本权限
76 chmod 777 *.sh
77 ./build-haproxy.sh
78 # haproxy容器镜像运行
79 docker run -itd --name haproxy-v1 -p 80:80 -p 9999:9999 -v
80 /opt/web/haproxy/haproxy.cfg:/usr/local/haproxy/haproxy.cfg centos-haproxy:v1
81
82 curl http://172.250.1.113:80/myapp/
83 http://172.250.1.113:9999/haproxy-status
84 =====
85 =====
86 # docker-server02主机
87 mkdir /opt/web/haproxy
88 cd /opt/web/haproxy
89
90 # 下载haproxy 包
91 wget https://www.haproxy.org/download/2.8/src/haproxy-2.8.9.tar.gz
92
93 # 编辑dockerfile
94 vim dockerfile
95 FROM centos7.9.2009:v1
96 RUN yum -y install glibc glibc-devel zlib-devel systemd-devel wget vim wget tree
97 lrzsz gcc gcc-c++ automake pcre pcre-devel zlib zlib-devel openssl openssl-
98 devel net-tools make iproute iotop net-tools
99 ADD haproxy-2.8.9.tar.gz /usr/local/src
100 RUN cd /usr/local/src/haproxy-2.8.9 \
101 && make ARCH=x86_64 TARGET=linux-glibc USE_PCRE=1 USE_OPENSSL=1 USE_ZLIB=1
102 USE_SYSTEMD=1 USE_CPU_AFFINITY=1 PREFIX=/usr/local/haproxy
103 && make install PREFIX=/usr/local/haproxy
104 && cp haproxy /usr/sbin/
105 ADD haproxy.cfg /etc/haproxy/
106 EXPOSE 80 9999
107 CMD ["/usr/local/sbin/run_haproxy.sh"]
108
109 # 编辑haproxy.cfg 配置文件
110 vim haproxy.cfg
111 # 全局配置部分
112 global
113 maxconn 100000
114 chroot /usr/local/haproxy #将进程的锁到/usr/local/haproxy目录，限制进程的文件访问范围。
115 #stats socket /var/lib/haproxy/haproxy.sock mode 600 level admin
116 uid 99
117 gid 99
118 daemon #将HAProxy 后台运行
119 log 127.0.0.1 local3 info #配置日志的输出到本地，日志级别为 local3，输出信息级别
120
121 # 默认配置部分
122 defaults
123 option http-keep-alive #启用 HTTP 连接保持活动状态，允许客户端和服务器之间保持长连接
124 option forwardfor # 头部，将原始客户端 IP 地址添加到请求头中，用于透明代理。
125 maxconn 100000
126 mode tcp # 设置默认的代理模式为 HTTP，用于处理 HTTP 请求和响应
127 timeout connect 300000ms # 设置客户端与服务器建立连接的超时时间为 300 秒
```

```

125    timeout client 300000ms      # 设置客户端的超时时间为 300 秒，包括接收客户端请求和发送响应的时间
126    timeout server 300000ms     # 设置服务器的超时时间为 300 秒，包括与后端服务器建立连接和接收响应的时间
127
128    # 监听配置部分
129    listen stats
130        mode http
131        bind 0.0.0.0:9999      #监听地址和端口
132        stats enable          #启用 HAProxy 的统计功能，允许通过特定URI访问统计信息
133        log global            # 指定使用全局定义的日志设置
134        stats uri /haproxy-status # 通过访问该URI可以查看 HAProxy 的统计信息
135        stats auth haadmin:123456 # 设置访问用户名和密码，用户名为 haadmin，密码为 123456
136
137    frontend docker_nginx_web #
138        bind 0.0.0.0:80 #监听地址和端口
139        mode http
140        default_backend docker_nginx_hosts
141
142    backend docker_nginx_hosts
143        #balance roundrobin # 设置负载均衡算法为轮询模式
144        server web1 172.250.1.113:8080 check inter 3000 fall 2 rise 5
145        server web2 172.250.1.112:8080 check inter 3000 fall 2 rise 5
146        # 定义后端服务器，指定服务器地址，并设置健康检查参数。
147
148
149    # 编辑容器启动脚本
150    vim run_haproxy.sh
151    #!/bin/bash
152    /usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg
153    tail -f /etc/hosts
154
155    # 编辑镜像构建脚本
156    vim build-haproxy.sh
157    #!/bin/bash
158    docker build -t centos-haproxy:v2 .
159    # 赋予脚本权限
160    chmod 777 *.sh
161    ./build-haproxy.sh
162    # haproxy容器镜像运行
163    docker run -itd --name haproxy-v2 -p 80:80 -p 9999:9999 -v
164        /opt/web/haproxy/haproxy.cfg:/usr/local/haproxy/haproxy.cfg centos-haproxy:v2
165    curl http://172.250.1.113:80/myapp/
166    http://172.250.1.113:9999/haproxy-status

```

4、搭建keepalived 高可用

```

1    # docker-server01主机
2    yum -y install keepalived
3
4    # 编辑keepalived 配置文件
5    vim /etc/keepalived/keepalived.conf
6    vrrp_instance VI_1 {
7        state MASTER      #主节点
8        interface eth1 #绑定的网络接口
9        virtual_router_id 1
10       priority 100 优先级
11       advert_int 1
12       unicast_src_ip 172.250.1.113 #指定 VRRP 实例中当前节点用于单播通信的源 IP 地址

```

```

13     unicast_peer { # 指定 VRRP 实例中当前节点的单播对等节点。
14         172.250.1.112
15     }
16     authentication {
17         auth_type PASS
18         auth_pass 1111
19     }
20     virtual_ipaddress {
21         172.250.1.114 dev eth1 label eth1:1
22     } #配置了虚拟 IP 地址 172.250.1.113 绑定到 eth1 接口上，并设置了标签为 eth1:1。
23
24 # 开启服务并临时开启内核参数
25 systemctl restart keepalived && systemctl enable keepalived
26 sysctl -w net.ipv4.ip_nonlocal_bind=1 #临时修改内核参数，控制是否允许绑定到非本地 IP 地址的套接字
27
28 =====
29 =====
30 # docker-server02主机
31 yum -y install keepalived
32
33 # 编辑keepalived 配置文件
34 vim /etc/keepalived/keepalived.conf
35 vrrp_instance VI_1 {
36     state BACKER #主节点
37     interface eth1 #绑定的网络接口
38     virtual_router_id 1
39     priority 50 优先级
40     advert_int 1
41     unicast_src_ip 172.250.1.112 #指定 VRRP 实例中当前节点用于单播通信的源 IP 地址
42     unicast_peer { # 指定 VRRP 实例中当前节点的单播对等节点。
43         172.250.1.113
44     }
45     authentication {
46         auth_type PASS
47         auth_pass 1111
48     }
49     virtual_ipaddress {
50         172.250.1.114 dev eth1 label eth1:1
51     } #配置了虚拟 IP 地址 192.168.222.69 绑定到 eth1 接口上，并设置了标签为 eth1:1。
52
53 systemctl restart keepalived && systemctl enable keepalived
54 sysctl -w net.ipv4.ip_nonlocal_bind=1 #临时修改内核参数，控制是否允许绑定到非本地 IP 地址的套接字
55
56 # 使用VIP访问
57 curl 172.250.1.114
58 curl 172.250.1.114:80/myapp/

```

第三章 Docker 数据管理：

在 Docker 容器中管理数据是非常重要的，因为容器本质上是临时性的，容器一旦删除，数据也会丢失。而 Docker 的镜像是分层设计的，镜像层是只读的，通过镜像启动的容器添加了一层可读写的文件系统，用户写入的数据都保存在这一层当中。

例如以下Mysql实验：

```

1 [root@localhosts ~]# mkdir /data/mysql -p

```

```

2 [root@localhosts ~]# docker run -it -p 3306:3306 -v /data/mysql:/var/lib/mysql
-e MYSQL_ROOT_PASSWORD='123qqq...A' mysql:5.7.40
3 [root@localhosts ~]# docker exec -it 00f5ec23ffba bash
bash-4.2# mysql -uroot -p123qqq...A
mysql> create database db01;
6
7 [root@localhosts ~]# ls /data/mysql
8 auto.cnf      client-cert.pem  ib_buffer_pool  ib_logfile1  mysql.sock
public_key.pem  sys
9 ca-key.pem    client-key.pem   ibdata1        ibtmp1       performance_schema
server-cert.pem
10 ca.pem        db01           ib_logfile0    mysql        private_key.pem
server-key.pem
11 [root@localhosts ~]# docker rm -f [root@localhosts ~]#
12 [root@localhosts ~]# docker run -it -p 3307:3306 -v /data/mysql:/var/lib/mysql -
e MYSQL_ROOT_PASSWORD='123qqq...A' mysql:5.7.40
13 [root@localhosts ~]# docker exec -it d7c49ea3c7fc bash
14 bash-4.2# mysql -uroot -p123qqq...A
15 mysql> show databases;
16 +-----+
17 | Database          |
18 +-----+
19 | information_schema |
20 | db01              |
21 | mysql              |
22 | performance_schema |
23 | sys                |
24 +-----+
25 5 rows in set (0.00 sec)
26
27 [root@localhosts ~]# docker inspect d7c49ea3c7fc

```

1.数据类型：

如果要将写入到容器的数据永久保存，则需要将容器中的数据保存到宿主机的指定目录，目前Docker的数据类型分为两种：

1. 是数据卷(data volume),数据卷类似于挂载的一块磁盘，数据容器是将数据保存在一个容器上。
2. 是数据卷容器(Data volume container),数据卷容器是将宿主机的目录挂载至一个专门的数据卷容器，然后让其他容器通过数据卷容器读写宿主机的数据。

2.什么是数据卷

数据卷 (Volume) 是 Docker 提供的一种用于持久化数据的机制，可以说宿主机上的目录或文件。允许容器之间共享数据，或者将数据存储在宿主机上，使得即使容器删除或重新创建，数据仍然能够保留和使用。

3.创建web目录并生成web页面

使用以下tomcat为例：

```

1 [root@localhosts ~]# mkdir /data/testapp
2 [root@localhosts ~]# echo "testapp web" > /data/testapp/index.html
3 [root@localhosts ~]# cat /data/testapp/index.html
4 testapp web
5

```

4.启动容器并验证数据：

启动两个容器， web1容器和web2容器， 分别测试是否能在宿主机访问到宿主机的数据。

注意： 使用 -V 参数， 将宿主机目录映射到容器内部， web的ro 标识在容器内对该目录只读， 默认是可读写的。

```
1 [root@localhosts ~]# docker run -d --name web1 --memory="2g" --ulimit  
nofile=65535:65535 -v /data/testapp/:/usr/local/tomcat/webapps/testapp -p 8080:8080  
tomcat:8.0  
2 [root@localhosts ~]# docker run -d --name web2 --memory="2g" --ulimit  
nofile=65535:65535 -v /data/testapp/:/usr/local/tomcat/webapps/testapp -p 8081:8080  
tomcat:8.0
```

5.在宿主机或容器修改数据：

```
1 [root@localhosts ~]# echo "web server2" > /data/testapp/index.html  
2 [root@localhosts ~]# cat /data/testapp/index.html  
3 web server2  
4  
5  
6 # 验证测试：  
7 curl localhost:8080/testapp  
8 curl localhost:8081/testapp
```

第四章 私有镜像Registry仓库搭建

Docker Registry作为Docker的核心组件之一负责镜像内容的存储与分发，客户端的docker pull以及push命令都将直接与registry进行交互，最初版本的registry由Python实现，由于设计初期在安全性，性能以及API的设计上有着诸多的缺陷，该版本在0.9之后停止了开发，由新的项目distribution(新的dockerregister被称为Distribution)来重新设计并开发下一代registry,新的项目由go语言开发，所有的API,底层存储方式，系统架构都进行了全面的重新设计已解决上一代registry中存在的问题，2016年4月份registry 2.0正式发布，docker 1.6版本开始支持registry 2.0,而八月份随着docker 1.8发布，docker hub正式启用2.1版本registry全面替代之前版本registry,新版registry对镜像存储格式进行了重新设计并和旧版不兼容，docker 1.5和之前的版本无法读取2.0的镜像，另外，Registry 2.4版本之后支持了回收站机制，也就是可以删除镜像了，在2.4版本之前是无法支持删除镜像的，所以如果你要使用最好是大于Registry 2.4版本的,Docker 的 Registry 开源实现已捐赠给 CNCF，现称为Distribution。

registry镜像：/opt/exam/kylin/docker/storehouse/registry.tar.gz

搭建配置：

```
1 # 导入镜像  
2 docker load -i /opt/exam/kylin/docker/storehouse/registry.tar.gz  
3  
4 # 创建授权使用目录  
5 mkdir /docker/auth -p  
6 cd /docker  
7  
8 # 创建用户  
9 htpasswd -Bbn jack 123456 > auth/htpasswd # 创建用户并生成密码。  
10  
11 # 验证用户和密码  
12 cat auth/htpasswd  
13  
14 # 启动docker registry  
15 docker run -d \
```

```

16 -p 5000:5000 \
17 --restart=always \
18 --name registry1 \
19 -v /docker/auth:/auth \
20 -e REGISTRY_AUTH=hpasswd \
21 -e REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm" \
22 -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/hpasswd \
23 registry:2
24
25 # 验证测试
26 docker login 172.250.10.79:5000
27 Username: jack
28 Password:
29 Error response from daemon: Get "https://172.250.10.79:5000/v2/": http: server gave
HTTP response to HTTPS client #出现一下报错解决方法。
30
31 # 添加配置参数
32 vim /lib/systemd/system/docker.service
33 .....
34 ExecStart=/usr/local/bin/dockerd -H fd:// --
35 containerd=/run/containerd/containerd.sock --insecure-registry=172.250.10.79:5000
36
37 # 重启服务
38 systemctl daemon-reload
39 systemctl restart docker
40
41 # 验证测试
42 docker login 172.250.10.79:5000
43 Login Succeeded # 出现一下表示登录成功
44
45 docker login localhost:5000
46 Login Succeeded # 出现一下表示登录成功

```

验证导入镜像测试

```

1 # 导入镜像
2 docker load -i /opt/exam/kylin/APP_images/tomcat8.0.tar.gz
3 # 修改镜像名字
4 docker tag ef6a7c98d192 172.250.10.106:5000/tomcat8.0:v1
5 # 推送镜像到仓库
6 docker push 172.250.10.79:5000/tomcat8.0:v1
7 The push refers to repository [172.250.10.79:5000/tomcat8.0]
8 d0f3f4011f28: Pushed
9 583dc95d65c9: Pushed
10 f26731984f9b: Pushed
11 9f052711b40a: Pushed
12 81242e1e644e: Pushed
13 39a6e47c4ae6: Pushed
14 fc6174f0df4a: Pushed
15 425325c72d90: Pushed
16 c596d5191368: Pushed
17 daf45b2cad9a: Pushed
18 8c466bf4ca6f: Pushed
19 v1: digest: sha256:47274bc304687e21217ca6a33e3244c43c6d5cfade76c4b478023f2aca3b2
size: 2625
20
21 # 验证
22 curl -u jack:123456 http://172.250.10.79:5000/v2/tomcat8.0/tags/list
23 {"name":"tomcat8.0","tags":["v1"]}
24

```

```
25 # 下载镜像
26 docker pull 172.250.10.79:5000/tomcat8.0:v1
27
28
29 # 删除镜像
```

第五章 练习题目

题目 1：创建 Nginx 反向代理配置

要求：

1. 创建一个 Nginx 容器，配置为反向代理两个 Tomcat 容器。
2. Tomcat 容器分别在 8081 和 8082 端口上运行，提供不同的网页内容。
3. Nginx 容器需要映射本地的 80 端口，并将请求转发到这两个 Tomcat 容器。

内容描述：

- Tomcat 1 在 `localhost:8081` 上提供静态网页 `index.html`，内容为 "Welcome to Tomcat 1"。
- Tomcat 2 在 `localhost:8082` 上提供静态网页 `index.html`，内容为 "Welcome to Tomcat 2"。
- Nginx 配置将 `http://localhost/tomcat1` 转发到 Tomcat 1，`http://localhost/tomcat2` 转发到 Tomcat 2。

```
1 # 导入tomcat镜像
2 docker load -i /opt/exam/kylin/APP_images/tomcat8.0.tar.gz
3 docker tag ef6a7c98d192 tomcat:tomcat8.0
4
5 # 创建 Tomcat 1 容器0
6 docker run -d --name tomcat1 --memory="2g" --ulimit nofile=65535:65535 -p 8081:8080
tomcat:tomcat8.0
7 # 在 Tomcat 1 容器内添加 index1.html
8 docker exec -it tomcat1 bash -c 'echo "welcome to Tomcat 1" >
/usr/local/tomcat/webapps/ROOT/index.html'
9
10 # 创建 Tomcat 2 容器docker
11 docker run -d --name tomcat2 --memory="2g" --ulimit nofile=65535:65535 -p 8082:8080
tomcat:tomcat8.0
12 # 在 Tomcat 2 容器内添加 index2.html
13 docker exec -it tomcat2 bash -c 'echo "welcome to Tomcat 2" >
/usr/local/tomcat/webapps/ROOT/index.html'
14
15
16 #创建 Nginx 配置文件
17 mkdir -p /app/nginx/conf/
18 docker run -d --name nginx01 nginx:latest
19 docker cp nginx01:/usr/local/nginx/conf/nginx.conf /app/nginx/conf
20 docker rm -f nginx01
21
22 # 修改nginx 配置文件
23 vim /app/nginx/conf/nginx.conf
24     server {
25         listen 80;
26             location /tomcat1 {
27                 proxy_pass http://172.250.10.79:8081/;
28             }
29             location /tomcat2 {
30                 proxy_pass http://172.250.10.79:8082/;
31         }
```

```

32
33 # 创建 Nginx 容器
34 docker run -d --name nginx -p 80:80 --link tomcat1 --link tomcat2 -v
35 /app/nginx/conf/nginx.conf:/usr/local/nginx/conf/nginx.conf nginx:latest
36
37 # 访问测试
38 curl localhost/tomcat1
39 Welcome to Tomcat 1
40
41 curl localhost/tomcat2
42 Welcome to Tomcat 2

```

题目 2：Nginx 动静分离和负载均衡配置

任务要求：

1. 静态站点配置：

- 当用户访问 `http://www.exli.com/optio` 时，Nginx 直接返回本地的静态页面，内容为 `hello world nginx`。

2. 动态站点配置和负载均衡：

- 当用户访问 `http://www.servertm.com/myserver` 时，Nginx 以加权轮询的方式将请求转发到两个后端 Tomcat 服务器上，来实现负载均衡。
- Tomcat 服务器页面内容为：
 - Tomcat 1 返回内容：`tomcat web1`
 - Tomcat 2 返回内容：`tomcat web2`
- 设置 Tomcat 1 和 Tomcat 2 的加权比例为 3:1，即 Tomcat 1 承担更多请求。

3. 环境要求：

- 两个 Tomcat 容器分别在宿主机上 `9081` 和 `9082` 端口上运行。
- Nginx 监听本机的 `80` 端口。
- 请确保 Nginx 容器可以访问到两个 Tomcat 容器。
- 将本地 `/app/nginx/conf/nginx.conf` 配置文件映射到 Nginx 容器中。

```

1 # 启动后端服务
2 docker run -d --name tomcat1 --memory="2g" --ulimit nofile=65535:65535 -p 9081:8080
3 tomcat:tomcat8.0
4 docker exec -it tomcat1 bash -c 'echo "tomcat web1" >
5 /usr/local/tomcat/webapps/ROOT/index.html'
6
7 docker run -d --name tomcat2 --memory="2g" --ulimit nofile=65535:65535 -p 9082:8080
8 tomcat:tomcat8.0
9 docker exec -it tomcat2 bash -c 'echo "tomcat web2" >
10 /usr/local/tomcat/webapps/ROOT/index.html'
11
12 #创建 Nginx 配置文件
13 mkdir -p /app/nginx/conf
14 docker run -d --name nginx01 nginx:latest
15 docker cp nginx01:/usr/local/nginx/conf/nginx.conf /app/nginx/conf/
16 docker rm -f nginx01

```

```
16
17 # 修改配置文件
18 vim /app/nginx/conf/nginx.conf
19 .....
20 # 定义后端负载均衡组
21 upstream webtomcat {
22     server 172.250.10.79:9081 weight=3;
23     server 172.250.10.79:9082 weight=1;
24 }
25
26 # 静态站点配置
27 server {
28     listen 80;
29     server_name www.exli.com;
30
31     location /option {
32         root html;
33         index index.html index.htm;
34     }
35 }
36
37 # 动态站点负载均衡配置
38 server {
39     listen 80;
40     server_name www.servertm.com;
41
42     location /myserver {
43         proxy_pass http://webtomcat/;
44     }
45 }
46
47
48 # 启动Nginx服务
49 docker run -d --name nginx -p 80:80 -v
50 /app/nginx/conf/nginx.conf:/usr/local/nginx/conf/nginx.conf nginx:latest
51 docker exec -it nginx bash -c 'mkdir /usr/local/nginx/html/option/'
52 docker exec -it nginx bash -c 'echo "hello world nginx" >
53 /usr/local/nginx/html/option/index.html'
54
55 # 访问测试
56 curl www.exli.com/option/
57 hello world nginx
58
59 # 访问tomcat
60 for i in {1..10};do
61 > curl www.servertm.com/myserver
62 > done
63 tomcat web1
64 tomcat web1
65 tomcat web2
66 tomcat web1
67 tomcat web1
68 tomcat web2
69 tomcat web1
70 tomcat web1
71 tomcat web1
```

题目3：使用nginx HTTPS反向代理私有镜像仓库

你的任务是搭建一个nginx 反向代理，并配置后端服务。

【任务要求】：

1. 搭建使用镜像仓库

- 使用 `registry:2` 镜像启动名为 `my-registry` 的容器，映射容器的 5000 端口到主机的 5000 端口，设置容器重启策略为 `always`，并映射容器的 `/var/lib/registry` 目录到主机的 `/var/lib/registry` 目录。
- 使用htpasswd工具，配置registry镜像仓库权限认证：认证用户：tom 密码：123456
- 用户认证文件应映射为：`/opt/exam/kylin/docker/htpasswd:/auth/htpasswd`

2. 搭建nginx反向代理

- 使用nginx:latest 镜像启动名为 `nginx-haproxy` 的容器，要求访问本地端口443端口，转发到私有镜像仓库的5000端口。
- nginx 的配置文件应该在`/web_server/nginx/conf/`目录当中。
- 密钥证书应该在`/web_server/nginx/conf`目录当中。

3. 推送镜像到私有镜像到仓库当中

- 请通过https方式将`/opt/exam/kylin/storehouse/hello-world.tar.gz` 镜像推送到私有镜像仓库。
- 并在`/etc/docker/daemon.json` 文件配置代理镜像地址。

```
1 # 创建认证用户
2 htpasswd -Bc /opt/exam/kylin/docker/htpasswd tom
3 # 创建私有镜像容器
4 docker run -d \
5   --restart always \
6   --name my-registry \
7   -e REGISTRY_AUTH=htpasswd \
8   -e REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm" \
9   -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd\
10  -p 5000:5000 \
11  -v /var/lib/registry:/var/lib/registry \
12  -v /opt/exam/kylin/docker/htpasswd:/auth/htpasswd \
13  registry:2
14
15
16 # 登录验证
17 docker login localhost:5000
18 输入用户名: tom
19 输入用户密码: 123456
20 # 看到Login Succeeded 代表登录成功
21
22 # 拷贝nginx配置文件到本地
23 docker run -d --name nginx01 nginx:latest
24 mkdir /web_server/nginx/conf -p
25 docker cp nginx01:/usr/local/nginx/conf/nginx.conf /web_server/nginx/conf
26
27 # 编辑nginx
28 vim /web_server/nginx/conf/nginx.conf
29 .....
30   server {
31     listen      443 ssl;
32     server_name localhost;
33     client_max_body_size 200M;
34 }
```

```

35      ssl_certificate      /usr/local/nginx/conf/server.crt;
36      ssl_certificate_key  /usr/local/nginx/conf/server.key;
37
38      ssl_session_cache    shared:SSL:1m;
39      ssl_session_timeout   5m;
40
41      ssl_ciphers  HIGH:!aNULL:!MD5;
42      ssl_prefer_server_ciphers  on;
43
44      location / {
45          proxy_pass http://宿主机IP:5000;
46      }
47  }
48
49 # 生成私钥
50 openssl genrsa -out /web_server/nginx/conf/server.key 2048
51 # 生成自签名证书
52 openssl req -new -x509 -key /web_server/nginx/conf/server.key -out
53 /web_server/nginx/conf/server.crt -days 365
54
55 # 创建nginx_proxy容器
56 docker run -d \
57   --name nginx_proxy \
58   -p 443:443 \
59   -v /web_server/nginx/conf/nginx.conf:/usr/local/nginx/conf/nginx.conf \
60   -v /web_server/nginx/conf/server.key:/usr/local/nginx/conf/server.key \
61   -v /web_server/nginx/conf/server.crt:/usr/local/nginx/conf/server.crt \
62   nginx:latest
63
64 # 配置镜像地址
65 vim /lib/systemd/system/docker.service
66 ExecStart=/usr/local/bin/dockerd -H fd:// --
67 containerd=/run/containerd/containerd.sock --insecure-registry=https://localhost
68
69 # 重新加载生效
70 systemctl daemon-reload
71 systemctl restart docker
72
73 # 推送镜像到私有仓库
74 docker load -i /opt/exam/kylin/storehouse/hello-world.tar.gz
75 docker tag 24e2f7dbd68a localhost/hello-world:latest
76 docker login https://localhost
77 docker push localhost/hello-world:latest
78
79 # 验证
80 curl -u tom:123456 -k https://localhost/v2/_catalog
81 {"repositories":["hello-world"]}

```

网络高可用

第一章 搭建高可用网络bond

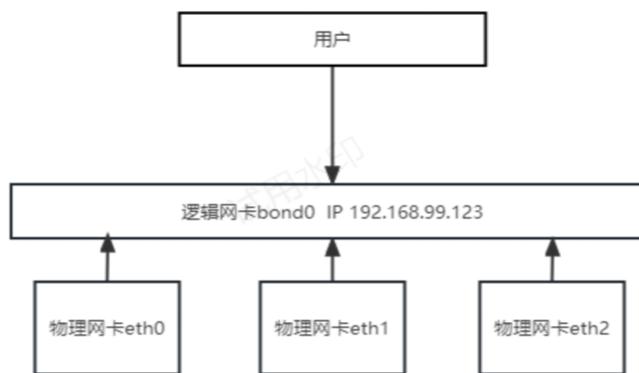
1.1 Bond概述

bond技术是将多块物理网卡绑定同一IP地址对外提供服务，通过不同的模式配置，从而达到高可用、负载均衡及链路冗余等效果。我们知道，两块物理网卡是不可以直接配置同一个IP地址的，多网卡bond技术的基本原理就是通过虚拟出一块逻辑网卡对外提供网络连接。

1.2 bond模式

Bond技术的基本原理是将多个网络接口聚合成一个bond接口，并在这个bond接口上实现负载均衡和故障切换。以下是对bond技术的概述：

1. 负载均衡模式 (balance-rr)：这是最常用的bond模式之一，它将数据流量均匀地分发到所有绑定的网络接口上，实现负载均衡。每个数据包都会按照一定的算法（如数据包计数、源IP地址等）选择一个网络接口进行传输，从而提高网络的吞吐量。
2. 主备模式 (active-backup)：在主备模式下，一个网络接口作为活动接口 (active)，负责传输数据，而其他接口作为备份接口 (backup)。如果活动接口发生故障或失效，系统会自动切换到备份接口，保证网络的连通性。这种模式提供了冗余的备份，但没有实现负载均衡。



1.3 通过nmcli命令配置网卡bond主备模式

1. 创建Bond接口：

- nmcli connection add type bond con-nam 【BOND_NAME】 ifname 【BOND_NAME】 mode 【BOND_MODE】
- 【BOND_NAME】：Bond接口的名称
- 【BOND_MODE】：Bond的工作模式，例如active-backup或balance-rr

2. 配置Bond接口的IP地址：

- nmcli connection modify 【BOND_NAME】 ipv4.addresses 【IP_ADDRESS/SUBNET_MASK】 ipv4.gateway 【GATEWAY_ADDRESS】 ipv4.dns 【DNS_ADDRESS】 ipv4.method manual
- 【BOND_NAME】：Bond接口的名称
- 【IP_ADDRESS/SUBNET_MASK】：网卡ip和子网掩码
- 【GATEWAY_ADDRESS】：默认网关
- 【DNS_ADDRESS】：DNS服务器

3. 将物理网卡添加到Bond接口：

- nmcli connection add type ethernet ifname 【PHYSICAL_INTERFACE_1】 master 【BOND_NAME】
- nmcli connection add type ethernet ifname 【PHYSICAL_INTERFACE_2】 master 【BOND_NAME】
- 【PHYSICAL_INTERFACE_1】：想要添加到Bond接口的第一块物理网卡的名称
- 【BOND_NAME】：Bond接口设置的名称

4. 启动从属接口和Bond接口：

- nmcli connection up [PHYSICAL_INTERFACE_1]

- nmcli connection up [PHYSICAL_INTERFACE_2]
- nmcli connection up [BOND_NAME]

1.4搭建bond网络

```

1 # 启用设备: nmcli device connect ens19
2
3 操作步骤1: 创建一个名为bond0的bond接口, 并设置为active-backup模式
4 [root@localhost ~]# nmcli connection add type bond con-name bond0 ifname bond0 mode
active-backup
5
6 操作步骤2: 为bond0接口配置IP地址、网关和DNS
7 [root@localhost ~]# nmcli connection modify bond0 ipv4.addresses 172.250.10.106/24
8     ipv4.gateway 172.250.10.254 ipv4.dns 8.8.8.8,114.114.114.114 ipv4.method manual
9
10 操作步骤3: 将两块物理网卡添加到bond0接口
11 [root@localhost ~]# nmcli connection add type ethernet ifname ens18 master bond0
12 [root@localhost ~]# nmcli connection add type ethernet ifname ens19 master bond0
13
14 操作步骤4: 启动从属接口和bond0接口。
15 [root@localhost ~]# nmcli con up bond-slave-ens18
16 [root@localhost ~]# nmcli con up bond-slave-ens19
17 [root@localhost ~]# nmcli con up bond0
18
19 操作步骤5: 查看bond连接状态
20 [root@localhost ~]# cat /proc/net/bonding/bond0
21
22 操作步骤6: 测试
23 ping -I bond0 172.250.10.106
24
25 操作步骤6: 验证两张网卡是否有流量
26 ip -s link show ens18
27 ip -s link show ens19
28
29 操作步骤7: 禁用一个网卡
30 nmcli con down bond-slave-ens19
31
32 操作步骤7: 在ens19网卡流量
33 ip -s link show ens19

```

删除操作

```
nmcli connection delete bond0
```

第二章 高可用网络team

2.1. 网卡Teaming原理:

网卡Teaming技术允许将多个物理网卡组合成一个逻辑网卡, 从而实现网络冗余、负载均衡。这种技术确保了在某个物理网卡出现故障时, 网络通信不会中断, 同时还可以提高网络的整体性能。

2.2. Teaming模式:

1. 轮询 (Round Robin)模式:

- 原理: 按照固定的顺序, 轮流地从组成的多块网卡中发送数据包。
- 优点: 均衡利用所有网卡, 实现负载均衡。
- 缺点: 可能导致数据包顺序混乱, 不适合需要连续数据流的应用。

2. 备份模式 (Active-Backup):

- 原理：仅有一块网卡在工作，当其出现故障时，备用网卡会立即接手工作。
- 优点：高可靠性，简单配置。
- 缺点：未充分利用所有网卡资源。

2.3 基本命令使用：

1. 创建team接口：使用nmcli工具创建一个新的team接口，并指定其运行模式。
 - nmcli con add type team con-name TEAM0 ifname team0, config '{"runner': {"name": "roundrobin"}}'
2. 添加网卡到team：将物理网卡添加到已创建的team接口中。
 - nmcli con add type team-slave con-name TEAM0-eth0 ifname eth0 master team0
 - nmcli con add type team-slave con-name TEAM0-eth1 ifname eth1 master team0
3. 修改team配置：根据需要更改team的配置。
 - nmcli con mod TEAM0 team.config '{"runner": {"name": "activebackup"}}'
4. 查看team状态：查看team接口的当前状态和配置。
 - nmcli con show team0
5. 启动/停止team接口：根据需要启动或停止team接口。
 - nmcli con up team0
 - nmcli con down team0
6. 删除team接口：删除不再需要的team接口。
 - nmcli con delete team0

2.4 搭建team配置

```

1 操作步骤1：首先，他需要创建一个新的team接口：
2 [root@localhost ~]# nmcli con add type team con-name TEAM0 ifname team0 config
3   '{"runner": {"name": "roundrobin"}}' #创建名为TEAM0的team接口
4
5 操作步骤2：添加物理网卡到team接口：
6 [root@localhost ~]# nmcli connection add type team-slave con-name team0-ens18 ifname
7   ens18 master team0
8 [root@localhost ~]# nmcli connection add type team-slave con-name team0-ens19 ifname
9   ens19 master team0
10
11 操作步骤3：配置team接口的网络参数：
12 [root@localhost ~]# nmcli con mod TEAM0 ipv4.addresses 172.250.10.106/24
13   ipv4.gateway 172.250.10.254 ipv4.dns "8.8.8.8 114.114.114.114" ipv4.method manual
14   connection.autoconnect yes
15
16 操作步骤4：启动team接口及其子接口：
17 [root@localhost ~]# nmcli con up TEAM0 #启动TEAM0接口
18 [root@localhost ~]# nmcli con up team-ens18 #启用子接口
19 [root@localhost ~]# nmcli con up team-ens19 #启用子接口
20
21 操作步骤5：验证team接口的配置：
22 [root@localhost ~]# nmcli con show TEAM0 #查看TEAM0的状态及其成员
23 [root@localhost ~]# teamdctl team0 state
24
25 操作步骤6：故障模拟：
26 为了验证Teaming的冗余功能，可以尝试断开一个物理网卡的连接，然后观察是否仍然有网络连通性。
27 [root@localhost ~]# nmcli con down TEAM0-eth1
28
29 操作步骤7：网络测试：
```

```
24 [root@localhost ~]# ping 192.168.98.254 #使用ping命令测试网络连通性，确保即使断开一个物理网卡的连接，网络仍然可用。  
25 操作步骤8：验证team接口的配置：  
26 [root@localhost ~]# nmcli con up TEAM0-eth1 #重新启用TEAM0-eth1接口，恢复网络的完整性
```