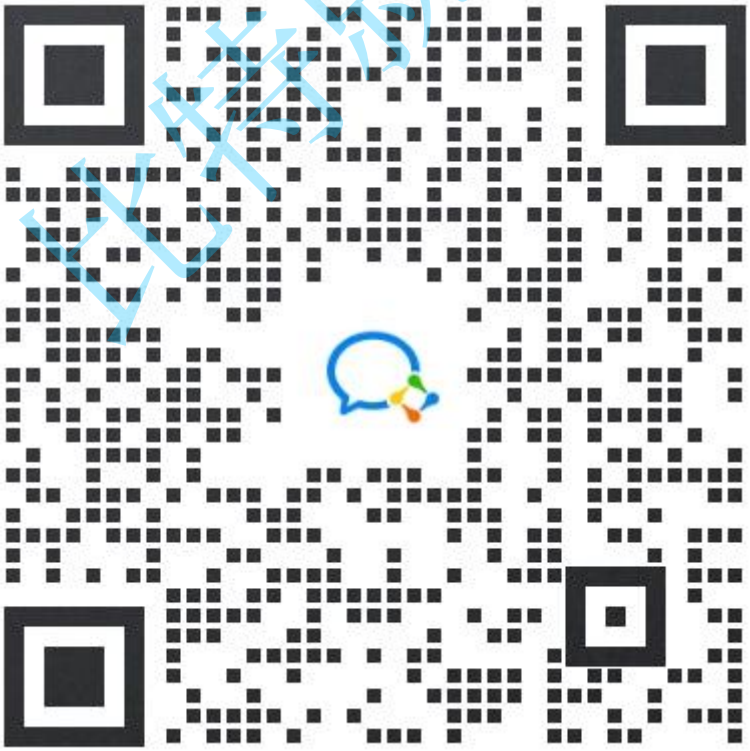
03 Docker 核心技术和实现原理

版权说明

对比特项目感兴趣，可以联系这个微信。



Docker 镜像制作

镜像制作及原因

镜像制作是因为某种需求，官方的镜像无法满足需求，需要我们通过一定手段来自定义镜像来满足要求。

制作镜像往往因为以下原因

1. 编写的代码如何打包到镜像中直接跟随镜像发布
2. 第三方制作的内容安全性未知，如含有安全漏洞
3. 特定的需求或者功能无法满足，如需要给数据库添加审计功能
4. 公司内部要求基于公司内部的系统制作镜像，如公司内部要求使用自己的操作系统作为基础镜像

Docker 镜像制作方式

制作容器镜像，主要有两种方法：

- 制作快照方式获得镜像（偶尔制作的镜像）：在基础镜像上（比如 Ubuntu），先登录容器中，然后安装镜像需要的所有软件，最后整体制作快照。
- Dockerfile 方式构建镜像（经常更新的镜像）：将软件安装的流程写成 Dockerfile，使用 docker build 构建成容器镜像。

快照方式制作镜像

制作命令

docker commit

- 功能

从容器创建一个新的镜像。

- 语法

```
Shell
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

- 参数
  - **-a :**提交的镜像作者；
  - **-c :**使用 Dockerfile 指令来创建镜像；可以修改启动指令
  - **-m :**提交时的说明文字；

- **-p :**在 commit 时，将容器暂停。

- 样例

```
Shell
docker commit c3f279d17e0a  maxhou/mynginx:v01
```

快照制作镜像实战

实战一、C++ HelloWorld 镜像制作

1. 创建临时工作目录

```
Shell
mkdir -p /data/maxhou/commitimage
cd /data/maxhou/commitimage
```

2. 编写 c++源代码文件,vi demo.c

```
Shell
#include <stdio.h>

int main()
{
    printf("hello docker!\n");
    return 0;
}
```

3. 启动一个 centos7 的容器

```
Shell
root@139-159-150-152:/data/maxhou/commitimage# docker run -it --
name mycppcommit centos:7 bash
[root@40de1bf45017 /]#
```

4. 替换为国内软件源

```
Shell
[root@40de1bf45017 /]# sed -e 's|^mirrorlist=|#mirrorlist=|g' \
>          -e
's|^#baseurl=http://mirror.centos.org/centos|baseurl=https://mirro
rs.ustc.edu.cn/centos|g' \
>          -i.bak \
```

```
>          /etc/yum.repos.d/CentOS-Base.repo
[root@40de1bf45017 /]# yum makecache
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
base
| 3.6 kB  00:00:00
extras
| 2.9 kB  00:00:00
updates
| 2.9 kB  00:00:00
(1/10): base/7/x86_64/group_gz
| 153 kB  00:00:00
(2/10): base/7/x86_64/primary_db
| 6.1 MB  00:00:00
(3/10): base/7/x86_64/filelists_db
| 7.2 MB  00:00:00
(4/10): base/7/x86_64/other_db
| 2.6 MB  00:00:00
(5/10): extras/7/x86_64/filelists_db
| 276 kB  00:00:00
(6/10): extras/7/x86_64/primary_db
| 249 kB  00:00:00
(7/10): extras/7/x86_64/other_db
| 149 kB  00:00:00
(8/10): updates/7/x86_64/filelists_db
|  11 MB  00:00:00
(9/10): updates/7/x86_64/other_db
| 1.3 MB  00:00:07
(10/10): updates/7/x86_64/primary_db
|  21 MB  00:00:16
Metadata Cache Created
```

5. 安装编译软件，并创建源代码目录

```
Shell
[root@40de1bf45017 /]#  yum install -y gcc
[root@40de1bf45017 /]#  mkdir /src/
```

6. 打开另外一个 shell，拷贝源代码到容器中

```
Shell
root@139-159-150-152:/data/maxhou/commitimage# docker cp ./demo.c
mycppcommit:/src
```

```
Successfully copied 2.048kB to mycppcommit:/src

#查看容器中
[root@40de1bf45017 /]# ls -l /src
total 4
-rw-r--r-- 1 root root 80 May 16 05:30 demo.c
```

7. 编译运行

```
Shell
[root@40de1bf45017 /]# cd /src
[root@40de1bf45017 src]# gcc demo.c -o demo
[root@40de1bf45017 src]# ./demo
hello docker!
```

8. 提交为一个镜像

```
Shell
root@139-159-150-152:/data/maxhou/commitimage# docker commit
mycppcommit mycppimg:v1.0
sha256:97d178ba9e5da794dd8276fe0ee23dc73510abea7f03f7f3c3a59a978dc
8fa2c

root@139-159-150-152:/data/maxhou/commitimage# docker images
mycppimg
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
mycppimg     v1.0      97d178ba9e5d   13 seconds ago   714MB
```

9. 测试镜像能否正常运行

```
Shell
root@139-159-150-152:/data/maxhou/commitimage# docker run -it
mycppimg:v1.0 ./src/demo
hello docker!
```

实战二、Springboot 镜像制作

第二章容器中《综合实战四、SpringBoot 容器制作》我们写好了一个简单的 springboot 服务，我们使用该服务来完成一个 springboot 镜像的制作

1. 启动一个 java8 的容器

```
Shell
root@139-159-150-152:/data/maxhou/commitimage#  docker run -it --
name myjavacommit java:8 bash
root@9ac4233dc4c8:/# java -version
openjdk version "1.8.0_111"
OpenJDK Runtime Environment (build 1.8.0_111-8u111-b14-2~bpo8+1-
b14)
OpenJDK 64-Bit Server VM (build 25.111-b14, mixed mode)
```

2. 打开另外一个 shell 窗口，拷贝 jar 包到容器的目录

```
Shell
docker cp ./springboot-demo-1.0-SNAPSHOT.jar
myjavacommit:/app.jar
```

3. java -jar 完成启动测试

```
Shell
root@9ac4233dc4c8:/# java -jar ./app.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.8)

2023-05-16 06:09:30.161  INFO 16 --- [           main]
com.bit.Main                               : Starting Main v1.0-
SNAPSHOT using Java 1.8.0_111 on 9ac4233dc4c8 with PID 16
(/app.jar started by root in /)
2023-05-16 06:09:30.173  INFO 16 --- [           main]
com.bit.Main                               : No active profile set,
falling back to 1 default profile: "default"
2023-05-16 06:09:32.912  INFO 16 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with
port(s): 8080 (http)
2023-05-16 06:09:32.961  INFO 16 --- [           main]
o.apache.catalina.core.StandardService   : Starting service
[Tomcat]
2023-05-16 06:09:32.962  INFO 16 --- [           main]
org.apache.catalina.core.StandardEngine  : Starting Servlet
engine: [Apache Tomcat/9.0.71]
```

```
2023-05-16 06:09:33.186  INFO 16 --- [           main]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring
embedded WebApplicationContext
2023-05-16 06:09:33.187  INFO 16 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root
WebApplicationContext: initialization completed in 2814 ms
2023-05-16 06:09:34.629  INFO 16 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on
port(s): 8080 (http) with context path ''
2023-05-16 06:09:34.654  INFO 16 --- [           main]
com.bit.Main                             : Started Main in 5.777
seconds (JVM running for 7.582)
```

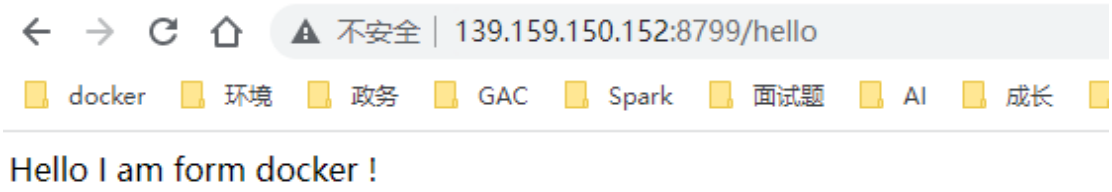
4. 执行 docker commit

```
Shell
root@139-159-150-152:/data/maxhou/commitimage# docker commit -c
'CMD ["java","-jar","/app.jar"]' myjavacommit myjavaimg:v1.0
sha256:865d428c7a87ec54819bb55d2e21d071f6149011d77a074ba638223ffc4
01824
root@139-159-150-152:/data/maxhou/commitimage# docker images
myjavaimg
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
myjavaimg    v1.0      865d428c7a87   18 seconds ago   661MB
```

5. 使用新创建的容器启动一个服务检查是否能够正常运行

```
Shell
root@139-159-150-152:/data/maxhou/commitimage#  docker run -p
8799:8080 -d --name mycommitjavaimg1 myjavaimg:v1.0
e336d7157d60ccd0fdb6a165aa01ebf7b3a2c19b449ddde6320fa7c280529b1d
root@139-159-150-152:/data/maxhou/commitimage# docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED
STATUS         PORTS                                     NAMES
e336d7157d60   myjavaimg:v1.0    "java -jar /app.jar"     2
seconds ago   Up 1 second    0.0.0.0:8799->8080/tcp, :::8799-
>8080/tcp   mycommitjavaimg1
```

6. 通过浏览器访问 8799 端口对应的接口



Dockerfile 制作镜像

Dockerfile 是什么

镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像,这个脚本就是 Dockerfile。

Dockerfile 是一个文本文件，其内**包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建**。

Dockerfile 格式

```Shell
# Comment
INSTRUCTION arguments
```

该指令不区分大小写。然而，约定是它们是大写的，以便更容易地将它们与参数区分开来。

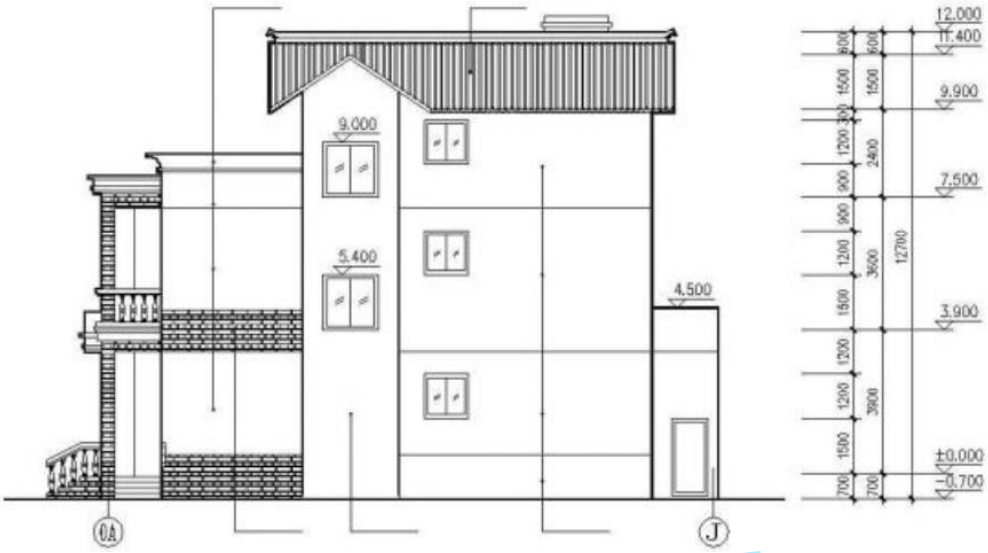
Docker 按顺序运行指令 `Dockerfile`

Docker 将以*开头*的行视为`#`注释，行中其他任何地方的标记`#`都被视为参数。这允许像这样的语句：

```Shell
# Comment
RUN echo 'we are running some # of cool things'
```

生活案例

Dockerfile 就像我们盖房子的时候的施工图纸，地基多深有多少层，每层是什么，以及多高。



为什么需要 Dockerfile

- 可以按照需求自定义镜像
  - 和 docker commit 一样能够自定义镜像，官方的镜像可以说很少能直接满足我们应用的，都需要我们自己打包自己的代码进去然后做成对应的应用镜像对外使用。
- 很方便的自动化构建，重复执行
  - 通过 dockerfile 可以自动化的完成镜像构建，而不是像 docker commit 一样，手动一个命令一个命令执行，而且可以重复执行，docker commit 的话很容易忘记执行了哪个命令，哪个命令没有执行。
- 维护修改方便，不再是黑箱操作
  - 使用 docker commit 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被称为黑箱镜像,dockerfile 很容易二次开发。
- 更加标准化，体积可以做的更小
  - docker 容器启动后，系统运行会生成很多运行时的文件，如果使用 commit 会导致这些文件也存储到镜像里面，而且 commit 的时候安装了很多的依赖文件，没有有效的清理机制的话会导致镜像非常的臃肿。使用 Dockerfile 则会更加标准化，而且提供多级构建，将编译和构建分开，不会有运行时的多余文件，更加的标准化。

Dockerfile 指令

指令清单

官方参考

Dockerfile 官方地址

指令	功能	备注
FROM	构建镜像基于哪个镜像，也就是基础镜像	必须掌握
MAINTAINER	镜像维护者姓名或邮箱地址	已经废弃，被 label 替代了
LABEL	为镜像添加元数据	
COPY	拷贝文件或目录到镜像中，跟 ADD 类似，但不具备自动下载或解压的功能	必须掌握
ADD	拷贝文件或目录到镜像中，如果是 URL 或压缩包便会自动下载或自动解压	必须掌握
WORKDIR	指定工作目录	必须掌握
RUN	指定 docker build 过程中运行的程序	必须掌握
VOLUME	指定容器挂载点	
EXPOSE	声明容器的服务端口（仅仅是声明）	
ENV	设置环境变量	必须掌握
CMD	运行容器时执行的命令	必须掌握
ENTRYPOINT	运行容器时程序入口	必须掌握
ARG	指定构建时的参数	
SHELL	指定采用哪个 shell	使用较少
USER	指定当前用户	
HEALTHCHECK	健康检测指令	

ONBUILD	在当前镜像构建时并不会被执行。只有当以当前镜像为基础镜像，去构建下一级镜像的时候才会被执行。	使用较少
STOPSIGNAL	允许您覆盖发送到容器的默认信号。	使用较少

下面我们详细学习 Dockerfile 的指令，并通过一个站点镜像的搭建来学习 Dockerfile 指令的使用。

FROM

- 功能
  - FROM 指令用于**为镜像文件构建过程指定基础镜像**，后续的指令运行于此基础镜像所提供的运行环境；

  注意事项
  - FROM 指令必须是 Dockerfile 中**非注释行或者 ARG 之后的第一个指令**；
  - 实践中，基准镜像可以是任何可用镜像文件，默认情况下，docker build 会在**docker 主机上查找指定的镜像文件**，在其不存在时，则**会自动从 Docker 的公共库** pull 镜像下来。如果找不到指定的镜像文件，docker build 会返回一个错误信息；
  - FROM 可以在一个 Dockerfile 中出现多次，如果有需求在一个 Dockerfile 中创建多个镜像,或将一个构建阶段作为另一个的依赖。
  - 如果 FROM 语句没有指定镜像标签，则**默认使用 latest 标签**。
- 语法

```
Shell
FROM [--platform=<platform>] <image> [AS <name>]
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

- 参数
  - <platform>:构建的 cpu 架构，如 `linux/amd64`, `linux/arm64`, `windows/amd64`
  - <image>：指定作为 base image 的名称；
  - <tag>：base image 的标签，省略时默认 latest；

- <digest>：是镜像的哈希码；
- AS <name>: 指定构建步骤的名称，配合 COPY --from=<name>可以完成多级构建

- 样例

```
Shell
FROM busybox:latest
```

- 实战

1. 创建 Docker 目录，确保目录中没有内容

```
Shell
mkdir -p /data/myworkdir/dockerfile/web1
cd /data/myworkdir/dockerfile/web1
```

2. 编辑 Dockerfile，测试 FROM 指令和注释，在 web1 目录中 vi Dockerfile，输入以下内容

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
```

3. 执行构建，打造镜像 v0.1 版本

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.1 .
[+] Building 0.1s (5/5) FINISHED
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 95B
0.0s
 => [internal] load .dockerignore
0.1s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => [1/1] FROM docker.io/library/ubuntu:22.04
0.0s
 => exporting to image
```

```
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:6c842b376e201bad109e4f12b9904dc98a5329acdddfb212134b961ee26
afe87
0.0s
 => => naming to docker.io/library/web1:v0.1
```

4. 运行制作的镜像，可以看到操作系统版本

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v0.1 cat /etc/*release*
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=22.04
DISTRIB_CODENAME=jammy
DISTRIB_DESCRIPTION="Ubuntu 22.04.2 LTS"
PRETTY_NAME="Ubuntu 22.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.2 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-
policies/privacy-policy"
UBUNTU_CODENAME=jammy
```


MAINTAINER

- 功能
  - 用于让 dockerfile 制作者提供本人的详细信息
  - 该功能已经废弃，由 label 替代
- 语法

```
Shell
MAINTAINER <authtor's detail>
```

- 参数
  - <authtor's detail>：作者信息
- 样例

```
Shell
MAINTAINER "maxhou <maxhou@bit.com>"
```

- 实战

1. 接 FROM 添加制作者信息，使用 MAINTAINER

```
Shell
 #我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
```

2. 再次编译 0.2 版本

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.2 .
[+] Building 0.1s (5/5) FINISHED
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 130B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => CACHED [1/1] FROM docker.io/library/ubuntu:22.04
0.0s
 => exporting to image
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:bd548ed8790827b7c37bfd1edf01796fa9cc2355dcf7463b5219b7bec41
34dd8
0.0s
 => => naming to docker.io/library/web1:v0.2
```

3. 查看镜像信息，可以看到作者信息已经添加完成

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker image
inspect web1:v0.2
[
    {
....
{
 "DockerVersion": "",
        "Author": "\"maxhou maxhou@bit.com\"",

}
....
```

LABEL

- 功能
  - 为镜像添加元数据，元数据是 kv 对形式
- 语法

```
Shell
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

- 样例

```
Shell
LABEL com.example.label-with-value="foo"
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

- 实战

1. 我们使用 LABEL 添加额外的元数据信息，继续 vi Dockerfile

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
```

2. 我们继续构建 v0.3 版本

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.3 .
[+] Building 0.1s (5/5) FINISHED
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 168B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => CACHED [1/1] FROM docker.io/library/ubuntu:22.04
0.0s
 => exporting to image
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:2d33a32643014f0ceb43f1cc264f487499f05db0ffe46e6e5db53967ea5
f34fb
0.0s
 => => naming to docker.io/library/web1:v0.3
```

3. 查看镜像元数据

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker
inspect web1:v0.3
[
....

      "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
```

```
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
"
            ],
            "Cmd": [
                "/bin/bash"
            ],
            "Image": "",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "app": "nginx",
                "company": "com.bit",
                "org.opencontainers.image.ref.name": "ubuntu",
                "org.opencontainers.image.version": "22.04"
            }
        },
```

COPY

- 功能
  - 用于从 docker 主机复制新文件或者目录至创建的新镜像指定路径中 。
- 语法

```
Shell
COPY [--chown=<user>:<group>] <src>... <dest>
COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

- 参数
  - <src>：要复制的源文件或目录，**支持使用通配符**；
  - <dest>：目标路径，即正在创建的 image 的文件系统路径；**建议<dest>使用绝对路径**，否则，COPY 指定以 WORKDIR 为当前路径

    在路径中有空白字符时，通常使用第 2 种格式；
  - --chown：修改用户和组
  - --from <name>可选项：

- 可以从之前构建的步骤中拷贝内容，结合 FROM .. AS <name>往往用作多级构建，后续我们有实战课专门完成多级构建

- 注意事项
  - <src>必须是 build 上下文中的路径，**不能是其父目录中的文件**；
  - 如果**<src>是目录**，则其**内部文件或子目录会被递归复制**，但**<src>目录自身不会被复制**；
  - 如果指定了多个<src>，或在<src>中使用了通配符，则<dest>必须是一个目录，且**必须以 / 结尾**；
  - 如果<dest>事先不存在，它将**会被自动创建**，这包括父目录路径。
- 样例

```
Shell
COPY index.html /data/web/html/  #要确保 dockerfile 同级路径下有 index.html 文件
```

- 实战

1. 创建一个 index.html，作为我们站点的首页 ,vi index.html 输入下面内容

```
Shell
<html>
        <h1>Hello ,My Home Page! by bit</h1>
</html>
```

2. 我们通过 COPY 命令添加到我们的镜像中，并且我们指定我们的根目录为 /data/web/www

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# cat Dockerfile
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
COPY index.html /data/web/www/
```

3. 我们再次编译 v0.4 版本镜像

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.4 .
[+] Building 0.2s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
0.1s
 => => transferring dockerfile: 199B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => [internal] load build context
0.0s
 => => transferring context: 90B
0.0s
 => CACHED [1/2] FROM docker.io/library/ubuntu:22.04
0.0s
 => [2/2] COPY index.html /data/web/www/
0.1s
 => exporting to image
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:a5e167b368fe1c42832e5012c8b0e417c7ba97458ea643f24ebcb5de938
1ee2b
0.0s
 => => naming to docker.io/library/web1:v0.4
```

4. 我们运行镜像，可以看到 index.html 已经进去了

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v0.4 ls /data/web/www
index.html
```

ENV

- 功能

- 用于为镜像定义所需的**环境变量**，并可被 Dockerfile 文件中**位于其后的**其它指令(如 ENV、ADD、COPY 等)所调用
- 调用格式为$variable_name 或 ${variable_name}

- 语法

```
Shell
ENV <key>=<value> ...
```

- 样例

```
Shell
ENV MY_NAME="John Doe"
```

- 实战

1. 目录后面可能复用，我们将目录的位置提取为变量，通过 ENV 来设置，我们再次编辑 Dockerfile

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
COPY index.html ${WEB_ROOT}
```

2. 我们再次编译 v0.5 版本镜像

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.5 .
[+] Building 0.1s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 224B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
```

```
 => [internal] load build context
0.0s
 => => transferring context: 31B
0.0s
 => [1/2] FROM docker.io/library/ubuntu:22.04
0.0s
 => CACHED [2/2] COPY index.html /data/web/www/
0.0s
 => exporting to image
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:c91a4e08ccfc93fecc10d66610f0e433e16ede20cb87b2a5dcb14a9a8d8
a3acb
0.0s
 => => naming to docker.io/library/web1:v0.5
```

3. 我们运行 v0.5 版本镜像，然后看下 index.html 是否在${WEB_ROOT}的目录下面

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v0.5 ls /data/web/www
index.html
```

4. 我们也可以通过 inspect 查看镜像，可以看到环境变量已经内置到镜像里面了

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker image
inspect web1:v0.5
[
....
"Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
```

```
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
",
                "WEB_ROOT=/data/web/www/"
            ],
```

WORKDIR

- 功能
  - 为 Dockerfile 中所有的 RUN、CMD、ENTRYPOINT、COPY 和 ADD 指定**设定工作目录**
- 语法

```
Shell
WORKDIR /path/to/workdir
```

- 注意事项
  - 默认的工作目录是/
  - 如果提供了相对路径，它将相对于前一条 `WORKDIR` 指令的路径。
  - `WORKDIR` 指令可以解析先前使用设置的环境变量 `ENV`
- 样例

```
Shell
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
#此处最终命令的输出 Dockerfile 为/a/b/c.
```

- 实战

1. 后面我们要下载 nginx，我们指定一个工作目录，通过 WORKDIR 来指定，后续我们就可以使用相对路径来执行 nginx 的编译了。我们再次编辑 Dockerfile

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
```

```
ENV WEB_ROOT=/data/web/www/
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
```

2. 我们再次编译镜像 v0.6

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.6 .
[+] Building 0.2s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 243B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => [internal] load build context
0.0s
 => => transferring context: 31B
0.0s
 => [1/3] FROM docker.io/library/ubuntu:22.04
0.0s
 => CACHED [2/3] COPY index.html /data/web/www/
0.0s
 => [3/3] WORKDIR /usr/local
0.1s
 => exporting to image
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:71e89a19350947993b8322a9164d23d0e5202c5ef4b21577f5d15036e17
fd738
0.0s
 => => naming to docker.io/library/web1:v0.6
```

3. 执行 pwd 命令查看当前目录

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
```

```
-name web1 --rm -it web1:v0.6 pwd
/usr/local
```

ADD

- 功能
  - ADD 指令类似于 COPY 指令，ADD 支持使用 TAR 文件和 URL 路径,会自动完成解压和下载
- 语法

```
Shell
ADD [--chown=<user>:<group>]  <src>... <dest>
ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

- 参数
  - <src>：要复制的源文件或目录，**支持使用通配符**；
  - <dest>：目标路径，即正在创建的 image 的文件系统路径；**建议<dest>使用绝对路径**，否则，ADD 指定以 WORKDIR 为其实路径；在路径中有空白字符时，通常使用第 2 种格式；
  - --chown：修改用户和组
- 实战

1. 我们登录 nginx 官网 https://nginx.org/，找到最新稳定版本的 nginx 的下载地址 https://nginx.org/en/download.html，可以看到最新版本为 1.22.1，我们复制该链接 https://nginx.org/download/nginx-1.22.1.tar.gz

2. 有了链接地址后，我们发现这是一个 URL 地址，我们通过 ADD 命令下载，因为 nginx 的未来版本还会变，所以我们可以提取 ningx 的版本为环境变量，我们再次编辑

我们的 Dockerfile

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
```

3. 我们执行编译 v0.7

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.7 .
[+] Building 2.3s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
0.1s
 => => transferring dockerfile: 337B
0.1s
 => [internal] load .dockerignore
0.1s
 => => transferring context: 2B
0.1s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => [internal] load build context
0.1s
 => => transferring context: 31B
0.1s
 => https://nginx.org/download/nginx-1.22.1.tar.gz
2.0s
 => [1/4] FROM docker.io/library/ubuntu:22.04
0.0s
 => CACHED [2/4] COPY index.html /data/web/www/
0.0s
 => CACHED [3/4] WORKDIR /usr/local
0.0s
 => [4/4] ADD https://nginx.org/download/nginx-1.22.1.tar.gz ./src
0.1s
```

```
 => exporting to image
0.0s
 => => exporting layers
0.0s
 => => writing image
sha256:8c235c36336579b3c3f3fe6f1131684025a9fa7917fac349ef9af7cae2a
b7380
0.0s
 => => naming to docker.io/library/web1:v0.7
```

4. 我们运行 v0.7 镜像查看，看是否已经完成了下载

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v0.7 ls -l /usr/local/src
total 1052
-rw------- 1 root root 1073948 Oct 19 09:23 nginx-1.22.1.tar.gz
```

5. 可以看到此时并没有被解压

6. 我们手动下载下来这个压缩包，放到我们的服务器目录，此时目录的结构如下：

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# wget
https://nginx.org/download/nginx-1.22.1.tar.gz
--2023-03-14 15:32:34--  https://nginx.org/download/nginx-
1.22.1.tar.gz
Resolving nginx.org (nginx.org)... 52.58.199.22, 3.125.197.172,
2a05:d014:edb:5704::6, ...
Connecting to nginx.org (nginx.org)|52.58.199.22|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 1073948 (1.0M) [application/octet-stream]
Saving to: ‘nginx-1.22.1.tar.gz’

nginx-1.22.1.tar.gz
100%[=================================================================
=====================================>]   1.02M  1.05MB/s    in
1.0s

2023-03-14 15:32:36 (1.05 MB/s) - ‘nginx-1.22.1.tar.gz’ saved
[1073948/1073948]
root@139-159-150-152:/data/myworkdir/dockerfile/web1# ll -h
total 1.1M
```

```
drwxr-xr-x 2 root root 4.0K Mar 14 15:32 ./
drwxr-xr-x 4 root root 4.0K Mar 14 14:37 ../
-rw-r--r-- 1 root root  298 Mar 14 15:26 Dockerfile
-rw-r--r-- 1 root root   53 Mar 14 15:01 index.html
-rw-r--r-- 1 root root 1.1M Oct 19 17:23 nginx-1.22.1.tar.gz
```

7. 我们再次编辑 Dockerfile，添加将 nginx 放到 src2 目录

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
ADD ${NGINX_VERSION}.tar.gz ./src2
```

8. 执行命令编译 v0.8

```
Shell
docker build -t web1:v0.8 .
```

9. 我们运行 v0.8 版本的镜像，然后查看该镜像的/usr/local/src2 目录，可以看到 nginx 已经被解压

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v0.8 ls -l /usr/local/src2
total 4
drwxr-xr-x 8 1001 1001 4096 Oct 19 08:02 nginx-1.22.1
```

RUN

- 功能
  - 用于指定 docker build 过程中运行的程序，其可以是任何命令
- 语法

```Shell
#shell form
RUN <command>
#exec form
RUN ["executable", "param1", "param2"]
```

- 参数
  - 第一种格式中，<command>通常是一个 **shell 命令**， 且**以“/bin/sh -c”来运行**它,Windows 默认为 cmd /S /C。如果一个脚本 test.sh 不能自己执行，必须要 `/bin/sh -c test.sh` 的方式来执行，那么，如果使用 `RUN 的 shell 形式`，最后得到的命令相当于：

```Bash
/bin/sh -c "/bin/sh -c 'test.sh'"
```

  - 第二种语法格式中的参数是一个 **JSON 格式的数组**，其中<executable>为要运行的命令，后面的 <paramN>为传递给命令的选项或参数；然而，此种格式指定的命令**不会以“/bin/sh -c”来发起**，因此**常见的 shell 操作**如变量替换以及通配符(?,*等)替换**将不会进行**；不过，如果要运行的命令依赖于此 shell 特性的话，可以将其替换为类似下面的格式。`RUN ["/bin/bash", "-c", "<executable>", "<param1>"]`
- 样例

```Shell
ENV WEB_SERVER_PACKAGE nginx-1.21.1.tar.gz
RUN cd ./src && tar -xf ${WEB_SERVER_PACKAGE}
```

- 实战

1. 接上，因为我们 nginx 是源码所以我们需要先解压 src 文件，通过 RUN 命令可以完成 nginx 的解压

```Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
```

```
ADD ${NGINX_VERSION}.tar.gz ./src2
#解压
RUN cd ./src && tar zxvf ${NGINX_VERSION}.tar.gz
```

2. 再次编译镜像 v0.9，然后查看镜像/usr/local/src 目录是否已经解压

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v0.9 .
[+] Building 4.3s (12/12) FINISHED
 => [internal] load build definition from Dockerfile
0.1s
 => => transferring dockerfile: 429B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => [1/6] FROM docker.io/library/ubuntu:22.04
0.0s
 => [internal] load build context
0.0s
 => => transferring context: 71B
0.0s
 => https://nginx.org/download/nginx-1.22.1.tar.gz
1.9s
 => CACHED [2/6] COPY index.html /data/web/www/
0.0s
 => CACHED [3/6] WORKDIR /usr/local
0.0s
 => CACHED [4/6] ADD https://nginx.org/download/nginx-
1.22.1.tar.gz ./src
0.0s
 => CACHED [5/6] ADD nginx-1.22.1.tar.gz ./src2
0.0s
 => [6/6] RUN cd ./src && tar zxvf nginx-1.22.1.tar.gz
1.9s
 => exporting to image
0.1s
 => => exporting layers
0.1s
```

```
 => => writing image
sha256:940d8dd72fd3f5e810a652480a0b414bed6a69a14450581a46cc94f3e64
bccbd
0.0s
 => => naming to docker.io/library/web1:v0.9
0.0s
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v0.9 ls -l /usr/local/src
total 1056
drwxr-xr-x 8 1001 1001    4096 Oct 19 08:02 nginx-1.22.1
-rw------- 1 root root 1073948 Oct 19 09:23 nginx-1.22.1.tar.gz
```

3. 因为是源码安装所以我们要编译安装 nginx，需要下载编译工具，已经依赖库信息，并通过 make 来完成编译，我们再次修改 dockerfile

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
ADD ${NGINX_VERSION}.tar.gz ./src2
#解压
RUN cd ./src && tar zxvf ${NGINX_VERSION}.tar.gz

#1.安装 build-essential 构建工具
#2.安装依赖包 libpcre3 libpcre3-dev zlib1g-dev 依赖库
#3.进入 nginx 目录
#4.执行编译和构建
RUN apt-get update -y && apt install -y build-essential libpcre3
libpcre3-dev zlib1g-dev
RUN cd ./src/${NGINX_VERSION} \
                && ./configure --prefix=/usr/local/nginx \
                && make && make install
```

4. 我们再次构建，然后查看构建的 nginx 是否成功生成

```
Shell
```

```
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v1.0 .                                        。。。。
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v1.0 /usr/local/nginx/sbin/nginx -V
nginx version: nginx/1.22.1
built by gcc 11.3.0 (Ubuntu 11.3.0-1ubuntu1~22.04)
configure arguments: --prefix=/usr/local/nginx
```

5. nginx 的默认配置文件为/usr/local/nginx/conf/nginx.conf,我们修改 server 部分，配置一个我们自己的配置文件，然后覆盖它

```
Shell
#user  nobody;
worker_processes  1;

#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;

#pid        logs/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       mime.types;
    default_type  application/octet-stream;

    #log_format  main  '$remote_addr - $remote_user [$time_local]
"$request" '
    #                  '$status $body_bytes_sent "$http_referer" '
    #                  '"$http_user_agent"
"$http_x_forwarded_for"';

    #access_log  logs/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    #keepalive_timeout  0;
```

```
    keepalive_timeout  65;

    #gzip  on;

    server {
        listen       80;
        server_name  localhost;

        #charset koi8-r;

        #access_log  logs/host.access.log  main;

        location / {
            root   /data/web/html/;
            index  index.html index.htm;
        }

        #error_page  404              /404.html;

        # redirect server error pages to the static page /50x.html
        #
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }

        # proxy the PHP scripts to Apache listening on
127.0.0.1:80
        #
        #location ~ \.php$ {
        #    proxy_pass   http://127.0.0.1;
        #}

        # pass the PHP scripts to FastCGI server listening on
127.0.0.1:9000
        #
        #location ~ \.php$ {
        #    root           html;
        #    fastcgi_pass   127.0.0.1:9000;
        #    fastcgi_index  index.php;
        #    fastcgi_param  SCRIPT_FILENAME
/scripts$fastcgi_script_name;
        #    include        fastcgi_params;
        #}
```

```
        # deny access to .htaccess files, if Apache's document
root
        # concurs with nginx's one
        #
        #location ~ /\.ht {
        #    deny  all;
        #}
    }

}
```

6. 我们可以看到我们的包的下载和编译特别耗时，我们调整下 Dockerfile 的顺序，然后添加我们配置的 nginx.conf，此时 Dockerfile 如下

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
#1.安装 build-essential 构建工具
#2.安装依赖包 libpcre3 libpcre3-dev zlib1g-dev 依赖库
RUN apt-get update -y && apt install -y build-essential libpcre3
libpcre3-dev zlib1g-dev
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
ADD ${NGINX_VERSION}.tar.gz ./src2
#解压
RUN cd ./src && tar zxvf ${NGINX_VERSION}.tar.gz

#3.进入 nginx 目录
#4.执行编译和构建
RUN cd ./src/${NGINX_VERSION} \
                && ./configure --prefix=/usr/local/nginx \
                && make && make install
COPY nginx.conf ./nginx/conf
```

目录结构如下

```
drwxr-xr-x 2 root root    4096 Mar 14 16:47 ./
drwxr-xr-x 4 root root    4096 Mar 14 14:37 ../
-rw-r--r-- 1 root root     730 Mar 14 16:31 Dockerfile
-rw-r--r-- 1 root root      53 Mar 14 15:01 index.html
-rw-r--r-- 1 root root 1073948 Oct 19 17:23 nginx-1.22.1.tar.gz
-rw-r--r-- 1 root root    1879 Mar 14 16:47 nginx.conf
```

7. 我们再次构建镜像 v1.1

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v1.1 .
```

8. 我们再次运行 nginx 镜像查看是否正常运行

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker run -
-name web1 --rm -it web1:v1.1 /usr/local/nginx/sbin/nginx -V
nginx version: nginx/1.22.1
built by gcc 11.3.0 (Ubuntu 11.3.0-1ubuntu1~22.04)
configure arguments: --prefix=/usr/local/nginx
```

CMD

- 功能
  - **类似于 RUN 指令**，CMD 指令也可用于运行任何命令或应用程序，不过，二者的运行时间点不同
  - **RUN 指令运行于映像文件构建过程**中，而 **CMD 指令运行于基于 Dockerfile 构建出的新映像文件启动一个容器时**
  - CMD 指令的首要**目的在于为启动的容器指定默认要运行的程序**，且其运行结束后，容器也将终止；不过，CMD 指定的命令其可以被 docker run 的命令行选项所覆盖
  - 在 Dockerfile 中可以存在**多个 CMD 指令，但仅最后一个会生效**
- 语法

```
Shell
CMD ["executable","param1","param2"] (exec form, this is the
preferred form)
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
CMD command param1 param2 (shell form)
```

- 注意事项
  - 第二种则用于为 ENTRYPOINT 指令提供默认参数
  - json 数组中，要使用**双引号**，单引号会出错
- 样例

```
Shell
CMD ["/usr/bin/wc","--help"]
```

- 实战

1. 接上，此时因为 docker 是需要一个长时间后台运行的,我们让 nginx 进入前台运行，这就需要我们的 CMD 或者 EntryPoint 来完成，我们先用 CMD 来配置，修改 Dockerfile 如下

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
#1.安装 build-essential 构建工具
#2.安装依赖包 libpcre3 libpcre3-dev zlib1g-dev 依赖库
RUN apt-get update -y && apt install -y build-essential libpcre3
libpcre3-dev zlib1g-dev
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
ADD ${NGINX_VERSION}.tar.gz ./src2
#解压
RUN cd ./src && tar zxvf ${NGINX_VERSION}.tar.gz

#3.进入 nginx 目录
#4.执行编译和构建
RUN cd ./src/${NGINX_VERSION} \
                && ./configure --prefix=/usr/local/nginx \
                && make && make install
COPY nginx.conf ./nginx/conf/
CMD ["/usr/local/nginx/sbin/nginx","-g daemon off;"]
```

2. 我们再次编译 v1.2 版本

```Shell
docker build -t web1:v1.2 .
```

3. 我们再次运行，然后 docker ps 可以看到容器已经在运行了

```Shell
docker run --name web1 --rm -d web1:v1.2
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS          PORTS     NAMES
eb88a30deaa4   web1:v1.2      "/usr/local/nginx/sb…"   3 seconds
ago   Up 2 seconds              web1
8f53962c6338   mybusybox:v1   "/bin/sh"                22 hours
ago    Up 22 hours               mybusybox1
```

EXPOSE

- 功能
  - 用于为容器声明**打开指定要监听的端口**以实现与外部通信
  - 该 `EXPOSE` 指令实际上并不发布端口。它充当构建图像的人和运行容器的人之间的一种文档，关于要发布哪些端口。要在运行容器时实际发布端口，使用`-p` 参数发布和映射一个或多个端口，或者使用`-P`flag 发布所有暴露的端口并将它们映射宿主机端口。
- 语法

```Shell
EXPOSE <port> [<port>/<protocol>...]
```

- 参数
  - <protocol>：tcp/udp 协议
  - <port>：端口
- 样例

```Shell
EXPOSE 80/tcp
```

实战

1. 接上一个指令，我们通过外部的浏览器访问，然后发现无法访问，是因为我们端

口还没对外开放

2. 我们通过 EXPOSE 暴露端口看下，调整 Dockerfile 如下

```
Shell
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
#1.安装 build-essential 构建工具
#2.安装依赖包 libpcre3 libpcre3-dev zlib1g-dev 依赖库
RUN apt-get update -y && apt install -y build-essential libpcre3
libpcre3-dev zlib1g-dev
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
ADD ${NGINX_VERSION}.tar.gz ./src2
#解压
RUN cd ./src && tar zxvf ${NGINX_VERSION}.tar.gz

#3.进入 nginx 目录
#4.执行编译和构建
RUN cd ./src/${NGINX_VERSION} \
                && ./configure --prefix=/usr/local/nginx \
                && make && make install
COPY nginx.conf ./nginx/conf/
EXPOSE 80/tcp
CMD ["/usr/local/nginx/sbin/nginx","-g daemon off;"]
```

3. 此时我们构建 v1.3 版本

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/web1# docker build
-t web1:v1.3
```

4. 我们再次运行，然后通过浏览器访问，不过我们要先停止之前的容器

```
Shell
docker stop web1
docker run --name web1 --rm -d web1:v1.3
```

5. 再次通过 docker ps 查看容器正常运行，但是通过浏览器访问还是不行，说明 EXPOSE 仅仅是用于声明，并没有起什么作用

6. 我们通过 docker -p 来对外映射端口

```
Shell
docker stop web1
docker run --name web1 --rm -p 80:80 -d web1:v1.3
```

7. 此时通过浏览器访问可以看到我们的首页

Hello ,My Home Page! by bit

ENTRYPOINT

- 功能
  - 用于指定容器的启动入口
- 语法

```
Shell
#exec from
ENTRYPOINT ["executable", "param1", "param2"]
# shell form
ENTRYPOINT command param1 param2
```

- 参数

- json 数组中，要使用**双引号**，单引号会出错

- 样例

```
Shell
ENTRYPOINT ["nginx","-g","daemon off;"]
```

- 实战

1. 我们将 CMD 调整为 EntryPoint 重新测试下，此时 Dockerfile 如下

```
Shell
#我的 web 站点 by maxhou
FROM ubuntu:22.04 as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
#1.安装 build-essential 构建工具
#2.安装依赖包 libpcre3 libpcre3-dev zlib1g-dev 依赖库
RUN apt-get update -y && apt install -y build-essential libpcre3
libpcre3-dev zlib1g-dev
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
ADD ${NGINX_VERSION}.tar.gz ./src2
#解压
RUN cd ./src && tar zxvf ${NGINX_VERSION}.tar.gz

#3.进入 nginx 目录
#4.执行编译和构建
RUN cd ./src/${NGINX_VERSION} \
                && ./configure --prefix=/usr/local/nginx \
                && make && make install
COPY nginx.conf ./nginx/conf/
EXPOSE 80/tcp
#CMD ["/usr/local/nginx/sbin/nginx","-g daemon off;"]
ENTRYPOINT ["/usr/local/nginx/sbin/nginx","-g daemon off;"]
```

2. 再次构建，运行我们的镜像 v1.4

```
Shell
docker build -t web1:v1.4 .
```

```
docker stop web1
docker run --name web1 --rm -p 80:80 -d web1:v1.4
```

3. 此时通过浏览器可以再次访问，发现可以正常访问

Hello ,My Home Page! by bit

ARG

- 功能
  - ARG 指令类似 ENV，定义了一个变量；区别于 ENV：用户可以在构建时 docker build --build-arg <varname> = <value> 进行对变量的修改；ENV 不可以；
  - 如果用户指定了未在 Dockerfile 中定义的构建参数，那么构建输出警告。
- 语法

```
Shell
ARG <name>[=<default value>]
```

- 注意事项
  - Dockerfile 可以包含一个或多个 ARG 指令
  - ARG 支持指定默认值
  - 使用范围：定义之后才能使用，定义之前为空，如下面的案例，执行命令 `docker build --build-arg username=what_user .`第二行计算结果为 `some_user`，不是我们指定的 build-arg 中的参数值 `what_user`

```
Shell
FROM busybox
USER ${username:-some_user}
ARG username
USER $username
# ...
```

  - ENV 和 ARG 同时存在，ENV 会覆盖 ARG

```
Shell
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=v1.0.0
```

```
RUN echo $CONT_IMG_VER
```

执行下面指令输出 v1.0.0

```
Plaintext
docker build --build-arg CONT_IMG_VER=v2.0.1 .
```

我们可以优化写法为

```
Shell
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=${CONT_IMG_VER:-v1.0.0}
RUN echo $CONT_IMG_VER
```

- 系统内置了一些 ARG 变量
  - `HTTP_PROXY`
  - `http_proxy`
  - `HTTPS_PROXY`
  - `https_proxy`
  - `FTP_PROXY`
  - `ftp_proxy`
  - `NO_PROXY`
  - `no_proxy`
  - `ALL_PROXY`
  - `all_proxy`

- 样例

```
Shell
FROM busybox
ARG user1=someuser
ARG buildno=1
```

- 实战

1. 突然有一天老板说把基础镜像升级到 22.10 吧，这个时候我们的 ARG 就排上用场了，我们定义一个 ARG 变量指定操作系统版本，修改后的 dockerfile 如下

```
Shell
#我的 web 站点 by maxhou
```

```
ARG UBUNTU_VERSION=22.04
FROM ubuntu:${UBUNTU_VERSION} as buildbase
MAINTAINER "maxhou maxhou@bit.com"
LABEL company="com.bit" app="nginx"
ENV WEB_ROOT=/data/web/www/
ENV NGINX_VERSION="nginx-1.22.1"
#1.安装 build-essential 构建工具
#2.安装依赖包 libpcre3 libpcre3-dev zlib1g-dev 依赖库
RUN apt-get update -y && apt install -y build-essential libpcre3
libpcre3-dev zlib1g-dev
COPY index.html ${WEB_ROOT}
WORKDIR /usr/local
ADD https://nginx.org/download/${NGINX_VERSION}.tar.gz ./src
ADD ${NGINX_VERSION}.tar.gz ./src2
#解压
RUN cd ./src && tar zxvf ${NGINX_VERSION}.tar.gz

#3.进入 nginx 目录
#4.执行编译和构建
RUN cd ./src/${NGINX_VERSION} \
                && ./configure --prefix=/usr/local/nginx \
                && make && make install
COPY nginx.conf ./nginx/conf/
EXPOSE 80/tcp
#CMD ["/usr/local/nginx/sbin/nginx","-g daemon off;"]
ENTRYPOINT ["/usr/local/nginx/sbin/nginx","-g daemon off;"]
```

2. 我们再次构建 v1.5 版本的镜像，指定 ARG 参数值，可以看到此时镜像重新构建

```
Shell
docker build --build-arg UBUNTU_VERSION=22.10 -t web1:v1.5 .
```

3. 我们再次运行 v1.5 镜像可以看到容器正常运行

```
Shell
docker stop web1
docker run --name web1 --rm -p 80:80 -d web1:v1.5
```

通过浏览器可以访问到我们的 web 站点

Hello ,My Home Page! by bit

VOLUME

- 功能
  - 用于在 image 中创建一个挂载点目录
  - 通过 VOLUME 指令创建的挂载点，无法指定主机上对应的目录，是自动生成的。
- 语法

```
Shell
VOLUME <mountpoint>
VOLUME ["<mountpoint>"]
```

- 参数
  - mountpoint:挂载点目录
  - 注意事项
    - 如果挂载点目录路径下此前有文件存在，docker run 命令会在卷挂载完成后将此前的所有文件复制到新挂载的卷中
    - 其实 VOLUME 指令只是起到了声明了容器中的目录作为匿名卷，但是并没有将匿名卷绑定到宿主机指定目录的功能。
    - volume 只是指定了一个目录，用以在用户忘记启动时指定-v 参数也可以保证容器的正常运行。比如 mysql，你不能说用户启动时没有指定-v，然后删了容器，就把 mysql 的数据文件都删了，那样生产上是会出大事故的，所以 mysql 的 dockerfile 里面就需要配置 volume，这样即使用户没有指定-v，容器被删后也不会导致数据文件都不在了。还是可以恢复的。
    - volume 与-v 指令一样，容器被删除以后映射在主机上的文件不会被删除。
    - 如果-v 和 volume 指定了同一个位置，会以-v 设定的目录为准，其实 volume 指令的设定的目的就是为了避免用户忘记指定-v 的时候导致的数据丢失，那么如果用户指定了-v，自然而然就不需要 volume 指定的位置了。
- 样例

```
Shell
VOLUME ["/data1","/data2"]
```

- 实战

1. 我们创建一个 docker file,如下指定/data 为匿名卷

```
FROM ubuntu:22.04
RUN mkdir -p /data
```

```
VOLUME ["/data"]
RUN echo "Hello bit by Dockerfile volume" > /data/myvolume.txt
CMD ["tail","-f","/etc/hosts"]
```

2. 构建镜像

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/volume# docker
build -t myvolume:v0.1 .
[+] Building 1.1s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
0.2s
 => => transferring dockerfile: 187B
0.1s
 => [internal] load .dockerignore
0.1s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04
0.0s
 => [1/3] FROM docker.io/library/ubuntu:22.04
0.0s
 => [2/3] RUN mkdir -p /data
0.4s
 => [3/3] RUN echo "Hello bit by Dockerfile volume" >
/data/myvolume.txt
0.4s
 => exporting to image
0.1s
 => => exporting layers
0.1s
 => => writing image
sha256:02f9adb20d5b9c22acae4251818a6cef1f669b33df93c1fb22b369df17d
92f52
0.0s
 => => naming to docker.io/library/myvolume:v0.1
```

3. 启动容器

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/volume# docker run
--name myvolume -d myvolume:v0.1
5d0992016082823f4f62342f57339c033d30e5b53d8a9025e3246b6cb0ebcfcc
root@139-159-150-152:/data/myworkdir/dockerfile/volume# docker ps
```

```
CONTAINER ID   IMAGE           COMMAND                  CREATED
STATUS         PORTS     NAMES
5d0992016082   myvolume:v0.1   "tail -f /etc/hosts"   5 seconds
ago   Up 4 seconds             myvolume
```

4. 查看匿名卷目录

```JSON
 #查看容器信息
 root@139-159-150-152:/data/var/lib/docker/volumes# docker inspect
5d0992016082
[
◦◦◦◦
 "Mounts": [
            {
                "Type": "volume",
                "Name":
"1b0b6a1a5f39dacc68acb03d144d67a90f853e8b42d760415faccf4c9a4b83fd"
,
                "Source":
"/data/var/lib/docker/volumes/1b0b6a1a5f39dacc68acb03d144d67a90f85
3e8b42d760415faccf4c9a4b83fd/_data",
                "Destination": "/data",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            }
        ],
```

5. 可以看到挂载卷目录，进入目录可以看到我们的文件

```JavaScript
root@139-159-150-152:/data/var/lib/docker/volumes# ll
/data/var/lib/docker/volumes/1b0b6a1a5f39dacc68acb03d144d67a90f853
e8b42d760415faccf4c9a4b83fd/_data
total 12
drwxr-xr-x 2 root root 4096 Mar 25 10:09 ./
drwx-----x 3 root root 4096 Mar 25 10:09 ../
-rw-r--r-- 1 root root   31 Mar 25 10:05 myvolume.txt
root@139-159-150-152:/data/var/lib/docker/volumes# cat
/data/var/lib/docker/volumes/1b0b6a1a5f39dacc68acb03d144d67a90f853
```

```
e8b42d760415faccf4c9a4b83fd/_data/myvolume.txt
Hello bit by Dockerfile volume
```

6. 删除容器，查看文件是没有被删除的。

```
JavaScript
root@139-159-150-152:/data/var/lib/docker/volumes# docker stop
myvolume
myvolume
root@139-159-150-152:/data/var/lib/docker/volumes# docker rm
myvolume
myvolume
root@139-159-150-152:/data/var/lib/docker/volumes# ll
/data/var/lib/docker/volumes/1b0b6a1a5f39dacc68acb03d144d67a90f853
e8b42d760415faccf4c9a4b83fd/_data
total 12
drwxr-xr-x 2 root root 4096 Mar 25 10:09 ./
drwx-----x 3 root root 4096 Mar 25 10:09 ../
-rw-r--r-- 1 root root   31 Mar 25 10:05 myvolume.txt
```

SHELL

- 功能
  - SHELL 指令允许覆盖用于 shell 命令形式的默认 shell。
  - Linux 上的默认 shell 是["/bin/sh"，"-c"]，在 Windows 上是["cmd"，"/S"，"/C"]
  - SHELL 指令必须以 JSON 格式写入 Dockerfile。
- 语法

```
Shell
SHELL ["executable", "parameters"]
```

- 参数
  - executable：shell 可执行文件的位置
  - parameters：shell 执行的参数
  - 注意事项
    - SHELL 指令可以多次出现。

- 每个 SHELL 指令都会覆盖所有先前的 SHELL 指令，并影响所有后续指令。
- 该 `SHELL` 指令在 Windows 上特别有用，因为 windows 行有两种不同的 shell：`cmd` 和 `powershell`

- 样例

```
Shell
FROM microsoft/windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S", "/C"]
RUN echo hello
```

- 实战

1. 我们可以在构建的时候切换使用不同的 shell
2. 新建一个目录

```
Shell
mkdir -p /data/myworkdir/dockerfile/shell
```

3. 新建 Dockerfile，vi Dockerfile

```
Shell
FROM ubuntu:22.04
RUN ls -l / > /test1.txt
SHELL ["/bin/bash","-cvx"]
RUN ls -l / > /test2.txt
```

4. 构建镜像

```
Shell
```

```
docker build -t shell:v0.1 --no-cache --progress=plain .
```

5. 运行查看结果

```
Shell
#1 [internal] load .dockerignore
#1 transferring context: 2B done
#1 DONE 0.0s

#2 [internal] load build definition from Dockerfile2
#2 transferring dockerfile: 133B 0.0s done
#2 DONE 0.1s

#3 [internal] load metadata for docker.io/library/ubuntu:22.04
#3 DONE 0.0s

#4 [1/3] FROM docker.io/library/ubuntu:22.04
#4 CACHED

#5 [2/3] RUN ls -l / > /test1.txt
#5 DONE 0.4s

#6 [3/3] RUN ls -l / > /test2.txt
#6 0.372 ls -l / > /test2.txt
#6 0.372 + ls -l /
#6 DONE 0.4s
```

USER

- 功能
  - 用于指定运行 image 时的或运行 Dockerfile 中任何 RUN、CMD 或 ENTRYPOINT 指令定的程序时的用户名或 UID
  - 默认情况下，container 的运行身份为 root 用户
- 语法

```
Shell
USER <user>[:<group>]
USER <UID>[:<GID>]
```

- 参数
  - user:用户

- group：用户组
- uid:用户 id
- gid:组 id
- 注意事项
  - <UID>可以为任意数字，但实践中其必须为/etc/passwd 中某用户的有效 UID,否则将运行失败

- 样例

```
Shell
USER docker:docker
```

- 实战

1. user 用于指定后续命令的运行用户，通常我们的程序建议不要直接用 root 用户操作
2. 我们创建一个目录

```
Shell
mkdir -p /data/myworkdir/dockerfile/user
```

3. 创建 Dockerfile，添加以下内容

```
Shell
FROM ubuntu:22.04 as buildbase
RUN groupadd nginx
RUN useradd nginx -g nginx
USER nginx:nginx
RUN whoami > /tmp/user1.txt
USER root:root
RUN groupadd mysql
RUN useradd mysql -g mysql
USER mysql:mysql
RUN whoami > /tmp/user2.txt
```

4. 执行编译

```
Shell
docker build -t user:v0.1 .
```

5. 运行查看我们的用户

```Shell
root@139-159-150-152:/data/myworkdir/dockerfile/user# docker run --name user1 --rm  -it  user:v0.1 cat /tmp/user1.txt /tmp/user2.txt
nginx
mysql
```

HEALTHCHECK

- 功能
  - HEALTHCHECK 指令告诉 Docker 如何测试容器以检查它是否仍在工作。
  - 即使服务器进程仍在运行，这也可以检测出陷入无限循环且无法处理新连接的 Web 服务器等情况。
- 语法

```Shell
HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the container)
HEALTHCHECK NONE (disable any healthcheck inherited from the base image)
```

- 参数
  - `OPTIONS` 选项有：
    - --interval=DURATION (default: 30s)：每隔多长时间探测一次，默认 30 秒
    - -- timeout= DURATION (default: 30s)：服务响应超时时长，默认 30 秒
    - --start-period= DURATION (default: 0s)：服务启动多久后开始探测，默认 0 秒
    - --retries=N (default: 3)：认为检测失败几次为宕机，默认 3 次
  - 返回值
    - 0：容器成功是健康的，随时可以使用
    - 1：不健康的容器无法正常工作
    - 2：保留不使用此退出代码
- 样例

```Shell
```

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

1. 创建目录

```
Shell
mkdir -p /data/myworkdir/dockerfile/healthcheck
```

2. 我们拉取一个 nginx 镜像，然后安装 curl，首先检查 80 端口，编辑 Dockerfile 如下

```
Shell
FROM nginx:1.22.1
#更换国内镜像源
RUN sed -i 's/deb.debian.org/mirrors.ustc.edu.cn/g'
/etc/apt/sources.list
RUN apt-get update && apt-get install -y curl && rm -rf
/var/lib/apt/lists/*
HEALTHCHECK --interval=5s --timeout=3s \
  CMD curl -fs http://localhost/ || exit 1
```

3. 我们构建镜像，然后运行

```
Shell
docker build -t healthcheck:v0.1 .
docker run --name hc1 --rm -d healthcheck:v0.1
```

4. 通过 docker ps 可以看到镜像成功运行,因为 nginx 默认是 80 端口

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/healthcheck#
docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED
STATUS                    PORTS
NAMES
4b4700a08235   healthcheck:v0.1   "/docker-entrypoint.…"   56
seconds ago      Up 55 seconds (healthy)   80/tcp
hc1
124c1abfd1dd   web1:v1.5          "/usr/local/nginx/sb…"   About
an hour ago   Up About an hour          0.0.0.0:80->80/tcp, :::80-
>80/tcp   web1
8f53962c6338   mybusybox:v1       "/bin/sh"                24
hours ago        Up 24 hours
```

5. 通过 docker inspect 我们可以看到我们配置的参数

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/healthcheck#
docker inspect hc1
[
    {
....
      "Healthcheck": {
                "Test": [
                    "CMD-SHELL",
                    "curl -fs http://localhost/ || exit 1"
                ],
                "Interval": 5000000000,
                "Timeout": 3000000000
            },

...
```

6. 我们调整 Dockerfile 端口为 10080，再次运行镜像

```
Shell
FROM nginx:1.22.1
#更换国内镜像源
RUN sed -i 's/deb.debian.org/mirrors.ustc.edu.cn/g'
/etc/apt/sources.list
RUN apt-get update && apt-get install -y curl && rm -rf
/var/lib/apt/lists/*
HEALTHCHECK --interval=5s --timeout=3s \
  CMD curl -fs http://localhost:10080/ || exit 1
```

7. 我们再次构建镜像 v0.2,停止第一个镜像

```
Shell
docker build -t healthcheck:v0.2 .
docker stop hc1
docker run --name hc2  -d healthcheck:v0.2
```

8. 过一段时间后，我们查看 docker ps 可以看到显示未不健康

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/healthcheck#
docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED
```

```
STATUS                      PORTS     NAMES
713d9b805fdd   healthcheck:v0.2   "/docker-entrypoint.…"   2
minutes ago   Up 2 minutes (unhealthy)   80/tcp    hc2
8f53962c6338   mybusybox:v1       "/bin/sh"                24
hours ago    Up 24 hours                           mybusybox1
```

9. 我们通过 docker inspect 查看可以看到已经发生了多次的失败

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/healthcheck#
docker inspect hc2
[
....

 "Health": {
                "Status": "unhealthy",
                "FailingStreak": 11,
                "Log": [
                    {
                        "Start": "2023-03-
14T18:52:47.242951421+08:00",
                        "End": "2023-03-
14T18:52:47.317606451+08:00",
                        "ExitCode": 1,
                        "Output": ""
                    },
                    {
                        "Start": "2023-03-
14T18:52:52.323168751+08:00",
                        "End": "2023-03-
14T18:52:52.395180253+08:00",
                        "ExitCode": 1,
                        "Output": ""
                    },
                    {
                        "Start": "2023-03-
14T18:52:57.40049726+08:00",
                        "End": "2023-03-
14T18:52:57.473780025+08:00",
                        "ExitCode": 1,
                        "Output": ""
                    },
                    {
```

```
                        "Start": "2023-03-
14T18:53:02.478480287+08:00",
                        "End": "2023-03-
14T18:53:02.550407578+08:00",
                        "ExitCode": 1,
                        "Output": ""
                    },
                    {
                        "Start": "2023-03-
14T18:53:07.558706019+08:00",
                        "End": "2023-03-
14T18:53:07.631105312+08:00",
                        "ExitCode": 1,
                        "Output": ""
                    }
                ]
            }
```

10. 删除容器释放空间

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/healthcheck#
docker stop hc2
hc2
root@139-159-150-152:/data/myworkdir/dockerfile/healthcheck#
docker rm hc2
hc2
```

ONBUILD

- 功能
  - 用于在 Dockerfile 中**定义一个触发器**
  - **以该 Dockerfile 中的作为基础镜像由 FROM 指令在 build 过程中被执行**时，将**会“触发”**创建其 base image 的 **Dockerfile 文件中的 ONBUILD 指令**定义的触发器
- 语法

```
Shell
ONBUILD <INSTRUCTION>
```

- 参数：
  - INSTRUCTION：dockerfile 的一条指令
- 样例

```
Shell
ONBUILD ADD . /app/src
```

- 实战

1. 创建目录

```
Shell
mkdir -p /data/myworkdir/dockerfile/build
```

2. 在里面创建 Dockerfile1 构建第一个基础镜像，设置 ONBUILD 的时候写入一次文件，Dockerfile1 内容如下

```
Shell
FROM ubuntu:22.04
LABEL version="0.1"
ONBUILD RUN echo "in build" >> /tmp/build.txt
```

3. 我们构建作为基础镜像

```
Shell
 docker build -t build:v0.1 -f Dockerfile1 .
```

4. 我们使用 build:v0.1 作为基础镜像，新建一个 Dockerfile2，我们来配置构建 build:v0.2 的镜像，Dockerfile2 的内容如下

```
Shell
FROM build:v0.1
LABEL version="0.2"
```

5. 构建 v0.2 镜像,可以看到我们在 v0.1 中设置的钩子自动执行了

```
Shell
root@139-159-150-152:/data/myworkdir/dockerfile/build# docker
build -t build:v0.2 -f Dockerfile2 .
[+] Building 1.6s (6/6) FINISHED
 => [internal] load build definition from Dockerfile2
0.1s
 => => transferring dockerfile: 74B
```

```
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/build:v0.1
0.0s
 => CACHED [1/1] FROM docker.io/library/build:v0.1
0.0s
 => [2/1] RUN echo "in build" >> /tmp/build.txt
1.5s
 => exporting to image
0.1s
 => => exporting layers
0.0s
 => => writing image
sha256:ead612c9f088d5c595e6c9e4db0d7e992748aa0c8b0968ea0fea069d8f6
23668
0.0s
 => => naming to docker.io/library/build:v0.2
```

STOPSIGNAL

- 功能
  - STOPSIGNAL 指令设置将发送到容器的系统调用信号。
  - 此信号可以是与内核的系统调用表中的位置匹配的有效无符号数，例如 9，或者 SIGNAME 格式的信号名，例如 SIGKILL。
- 语法

```
Shell
STOPSIGNAL signal
```

- 参数
  - STOPSIGNAL 指令设置将发送到容器出口的系统调用信号。 此信号可以是与内核的系统调用表中的位置匹配的有效无符号数，例如 9，或者 SIGNAME 格式的信号名，例如 SIGKILL。常见的信号如下：

代号	名称	内容
1	SIGHUP	启动被终止的程序，可让该进程重新读取自己的配置文

		件，类似重新启动。
2	SIGINT	相当于用键盘输入 [ctrl]-c 来中断一个程序的进行。
9	SIGKILL	代表强制中断一个程序的进行，如果该程序进行到一半，那么尚未完成的部分可能会有“半产品”产生，类似 vim 会有 .filename.swp 保留下来。
15	SIGTERM	以正常的方式来终止该程序。由于是正常的终止，所以后续的动作会将他完成。不过，如果该程序已经发生问题，就是无法使用正常的方法终止时，输入这个 signal 也是没有用的。
19	SIGSTOP	相当于用键盘输入 [ctrl]-z 来暂停一个程序的进行。

- 样例

```
Shell
STOPSIGNAL 9
```

- 实战

1. 创建目录

```
Shell
mkdir -p /data/myworkdir/dockerfile/ss
```

2. 编辑 Dockerfile 如下

```
Shell
FROM nginx:1.22.1
STOPSIGNAL 9
ENTRYPOINT ["nginx","-g daemon off;"]
```

3. 执行镜像构建

```
Shell
docker build -t stopsignal:v0.1 .
```

4. 运行镜像

```
Shell
docker run --name stopsingal1 --rm -d stopsignal:v0.1
```

5. 通过 docker ps 查看

```
Shell
root@139-159-150-152:~# docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED
STATUS         PORTS     NAMES
990df945b604   stopsignal:v0.1   "nginx '-g daemon of…"   3
seconds ago    Up 3 seconds    80/tcp    stopsingal1
8f53962c6338   mybusybox:v1      "/bin/sh"                24 hours
ago    Up 24 hours              mybusybox1
```

6. 打开另外一个 shell 窗口 B,执行 docker logs -f 查看日志

```
Shell
root@139-159-150-152:~# docker logs -f stopsingal1
2023/03/14 11:33:09 [notice] 1#1: using the "epoll" event method
2023/03/14 11:33:09 [notice] 1#1: nginx/1.22.1
2023/03/14 11:33:09 [notice] 1#1: built by gcc 10.2.1 20210110
(Debian 10.2.1-6)
2023/03/14 11:33:09 [notice] 1#1: OS: Linux 5.4.0-100-generic
2023/03/14 11:33:09 [notice] 1#1: getrlimit(RLIMIT_NOFILE):
1048576:1048576
2023/03/14 11:33:09 [notice] 1#1: start worker processes
2023/03/14 11:33:09 [notice] 1#1: start worker process 6
```

7. 在原来的 shell 窗口中执行 docker stop，然后查看日志是突然退出

```
Shell
root@139-159-150-152:~# docker logs -f stopsingal1
2023/03/14 11:33:09 [notice] 1#1: using the "epoll" event method
2023/03/14 11:33:09 [notice] 1#1: nginx/1.22.1
2023/03/14 11:33:09 [notice] 1#1: built by gcc 10.2.1 20210110
(Debian 10.2.1-6)
2023/03/14 11:33:09 [notice] 1#1: OS: Linux 5.4.0-100-generic
2023/03/14 11:33:09 [notice] 1#1: getrlimit(RLIMIT_NOFILE):
1048576:1048576
2023/03/14 11:33:09 [notice] 1#1: start worker processes
2023/03/14 11:33:09 [notice] 1#1: start worker process 6
root@139-159-150-152:~#
```

8. 我们再来创建一个 Dockerfile2，这个里面我们不配置停止信号

```Shell
FROM nginx:1.22.1
ENTRYPOINT ["nginx","-g daemon off;"]
```

9. 构建然后运行我们的容器

```Shell
docker build -t stopsignal:v0.2 -f Dockerfile2 .
docker run --name stopsingal2 --rm -d stopsignal:v0.2
```

10. 在 shell 窗口 B 中执行 docker logs 查看日志

```Shell
root@139-159-150-152:~# docker logs -f stopsingal2
2023/03/14 11:38:53 [notice] 1#1: using the "epoll" event method
2023/03/14 11:38:53 [notice] 1#1: nginx/1.22.1
2023/03/14 11:38:53 [notice] 1#1: built by gcc 10.2.1 20210110
(Debian 10.2.1-6)
2023/03/14 11:38:53 [notice] 1#1: OS: Linux 5.4.0-100-generic
2023/03/14 11:38:53 [notice] 1#1: getrlimit(RLIMIT_NOFILE):
1048576:1048576
2023/03/14 11:38:53 [notice] 1#1: start worker processes
2023/03/14 11:38:53 [notice] 1#1: start worker process 6
2023/03/14 11:39:46 [notice] 1#1: signal 3 (SIGQUIT) received,
shutting down
2023/03/14 11:39:46 [notice] 6#6: gracefully shutting down
2023/03/14 11:39:46 [notice] 6#6: exiting
2023/03/14 11:39:46 [notice] 6#6: exit
2023/03/14 11:39:46 [notice] 1#1: signal 17 (SIGCHLD) received
from 6
2023/03/14 11:39:46 [notice] 1#1: worker process 6 exited with
code 0
2023/03/14 11:39:46 [notice] 1#1: exit
```

11. 我们可以看到正常的退出 nginx 会打印优雅的退出，先退出子进程，再退出主进程，而不是像我们设置的强制退出信号一样，直接进程消失了，没有任何反应。

制作命令

docker build

- 功能

docker build 命令用于使用 Dockerfile 创建镜像。

- 语法

```
Shell
 docker build [OPTIONS] PATH | URL | -
```

- 关键参数
  - **--build-arg=[] :**设置镜像创建时的变量；
  - **-f :**指定要使用的 Dockerfile 路径；
  - **--label=[] :**设置镜像使用的元数据；
  - **--no-cache :**创建镜像的过程不使用缓存；
  - **--pull :**尝试去更新镜像的新版本；
  - **--quiet, -q :**安静模式，成功后只输出镜像 ID；
  - **--tag, -t:** 镜像的名字及标签，通常 name:tag 或者 name 格式；可以在一次构建中为一个镜像设置多个标签。
  - **--network:** 默认 default。在构建期间设置 RUN 指令的网络模式
- 样例

```
Shell
docker build -t mynginx:v1 .
```

Dockerfile 编写优秀实践

1. **善用.dockerignore 文件**

使用它可以标记在执行 docker build 时忽略的路径和文件，**避免**发送**不必要的数据**内容，从而**加快**整个镜像**创建**过程。

2. **镜像的多阶段构建**

通过多步骤创建，可以将编译和运行等过程分开，保证最终生成的镜像只包括运行应用所需要的最小化环境。当然，用户也可以通过分别构造编译镜像和运行镜像来达到类似的结果，但这种方式需要维护多个 Dockerfile。

3. **合理使用缓存**

如合理使用 cache，减少内容目录下的文件，**内容不变的指令尽量放在前面**，这样可以尽量复用；

4. **基础镜像尽量使用官方镜像，并选择体积较小镜像**

容器的核心是应用，大的平台微服务可能几十上百个。选择过大的父镜像（如 Ubuntu 系统镜像）会造成最终生成应用镜像的臃肿，**推荐选用瘦身过的应用镜像（如 node:slim），或者较为小巧的系统镜像（如 alpine、busybox 或 debian）**；

5. **减少镜像层数**

如果希望所生成镜像的层数尽量少，则要**尽量合并 RUN、ADD 和 COPY 指令**。通常情况下，多个 RUN 指令可以合并为一条 RUN 指令；如 apt get update&&apt install 尽量写到一行

6. **精简镜像用途**

尽量让每个镜像的用途都比较**集中单**一，避免构造大而复杂、多功能的镜像；

7. **减少外部源的干扰**

如果确实要从外部引入数据，需要指定持久的地址，并带版本信息等，让他人可以复用而不出错。

8. **减少不必要的包安装**

只安装需要的包，不要安装无用的包，减少镜像体积。

操作实战

实战一、C++ HelloWorld 镜像制作

C++镜像制作

实战二、Springboot 镜像制作

SpringBoot 微服务镜像制作

实战三、正确使用 CMD 和 EntryPoint

CMD 与 EntryPoint 实战

实战四、优秀实践一合理使用 dokerignore

合理使用 dokerignore

实战五、优秀实践二多阶段构建

多阶段构建

实战六、优秀实践三合理使用缓存

合理使用缓存

实战七、Dockerfile 结合 docker compose 搭建 mysql 主从同步

Dockerfile 搭建 mysql 主从集群

实战八、Dockerfile 结合 docker compose 搭建 redis 集群

Dockerfile 构建 Redis 集群

实战九、Dockerfile 结合 docker compose 搭建 nginx、mysql、springboot 微服务站点

Dockerfile 配合 docker-compose 搭建微服务

实战十、Dockerfile 结合 docker compose 搭建 c++微服务站点

Dockerfile 结合 dockercompose 搭建 C++微服务

镜像制作常见问题

1. ADD 与 COPY 的区别

ADD：不仅能够将构建命令所在的主机本地的文件或目录，而且能够将远程 URL 所对应的文件或目录，作为资源复制到镜像文件系统。所以，可以认为 ADD 是增强版的 COPY，支持将远程 URL 的资源加入到镜像的文件系统。

COPY：COPY 指令能够将构建命令所在的主机本地的文件或目录，复制到镜像文件系统。

有的时候就是只需要拷贝压缩包，那么我们就要用 COPY 指令了

2. CMD 与 EntryPoint 的区别

ENTRYPOINT 容器启动后执行的命令,让容器执行表现的像一个可执行程序一样,与 CMD 的区别是不可以被 docker run 覆盖，会把 docker run 后面的参数当作传递给 ENTRYPOINT 指令的参数。

Dockerfile 中只能指定一个 ENTRYPOINT,如果指定了很多,只有最后一个有效。docker run 命令的 -entrypoint 参数可以把指定的参数继续传递给 ENTRYPOINT

组合使用 ENTRYPOINT 和 CMD, ENTRYPOINT 指定默认的运行命令, CMD 指定默认的运行参数

3. 多个 From 指令如何使用

多个 FROM 指令并不是为了生成多根的层关系，最后生成的镜像，仍以最后一条

FROM 为准，之前的 FROM 会被抛弃，那么之前的 FROM 又有什么意义呢？

每一条 FROM 指令都是一个构建阶段，多条 FROM 就是多阶段构建，虽然最后生成的镜像只能是最后一个阶段的结果，但是，能够将前置阶段中的文件拷贝到后边的阶段中，这就是多阶段构建的最大意义。

最大的使用场景是将编译环境和运行环境分离.

4. 快照和 dockerfile 制作镜像有什么区别？

等同于为什么要使用 Dockerfile。

5. 什么是空悬镜像（dangling ）

仓库名、标签均为<none> 的镜像被称为虚悬镜像，一般来说，虚悬镜像已经失去了存在的价值，是可以随意删除的。

造成虚悬镜像的原因：

原因一：

原本有镜像名和标签的镜像，发布了新版本后，重新 docker pull *** 时，旧的镜像名被转移到了新下载的镜像身上，而旧的镜像上的这个名称则被取消；

原因二：

docker build 同样可以导致这种现象。比如用 dockerfile1 构建了个镜像 tnone1:v1,又用另外一个 Dockerfile2 构建了一个镜像 tnone1:v1，这样之前的那个镜像就会变成空悬镜像。

可以用下面的命令专门显示这类镜像：

```Shell
docker image ls -f dangling=true
```

6. 中间层镜像是什么？

为了加速镜像构建、重复利用资源，Docker 会利用 **中间层镜像**。所以在使用一段时间后，可能会看到一些依赖的中间层镜像。默认的 `docker image ls` 列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像的话，需要加 `-a` 参数。

```Shell
 docker image ls -a
```

这样会看到很多无标签的镜像，与之前的虚悬镜像不同，这些无标签的镜像很多都是中间层镜像，是其它镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍，而这些镜像是别的镜像的依赖，因此并不会因为它们被列出来而多存了一份，无论如何你也会需要它们。只要删除那些依赖它们的镜像后，这些依赖的

中间层镜像也会被连带删除。

Docker 镜像原理

docker 是操作系统层的虚拟化，所以 docker 镜像的本质是在模拟操作系统。我们先看下操作系统是什么。

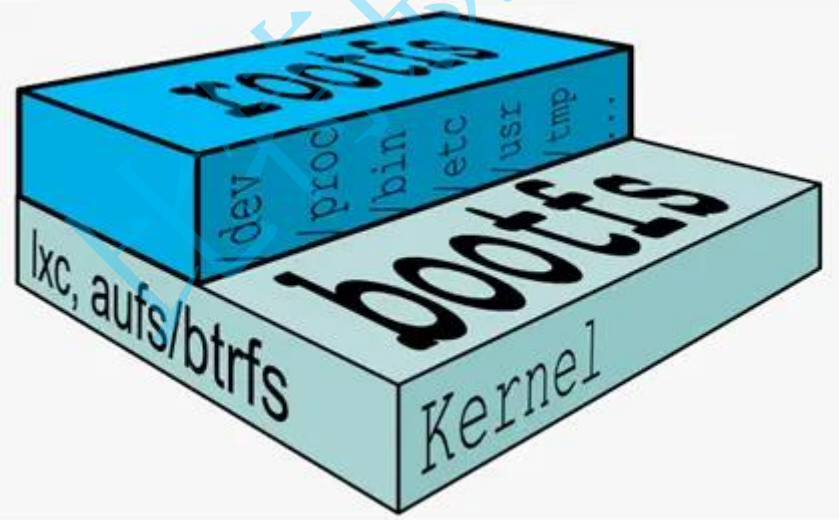
操作系统基础

操作系统由：进程调度子系统、进程通信子系统、内存管理子系统、设备管理子系统、文件管理子系统、网络通信子系统、作业控制子系统组成。

Linux 的文件管理子系统由 bootfs 和 rootfs 组成。

(1). bootfs：要包含 bootloader 和 kernel, bootloader 主要是引导加载 kernel, Linux 刚启动时会加载 bootfs 文件系统，在 Docker 镜像的最底层是引导文件系统 bootfs。这一层与我们典型的 Linux/Unix 系统是一样的，包含 boot 加载器和内核。当 boot 加载完成之后整个内核就都在内存中了，此时内存的使用权已由 bootfs 转交给内核，此时系统也会卸载 bootfs。

(2). rootfs： 在 bootfs 之上。包含的就是典型 Linux 系统中的 /dev, /proc, /bin, /etc 等标准目录和文件。rootfs 就是各种不同的操作系统发行版，比如 Ubuntu，Centos 等等。



Union FS(联合文件系统)

联合文件系统（Union File System），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。

UnionFS（联合文件系统）是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下。UnionFS 是一种为 Linux，FreeBSD 和 NetBSD 操作系统设计的把其他文件系统联合到一个联合挂载点的文件系统服务。它使用 branch 把不同文件系统的文件和目录“透明地”覆盖，形成一个单一一致的文件系统。这些 branches 或者是 read-only 或者是 read-write 的，所以当对这个虚拟后的联合文件系统进行写操作的时候，系统是真正写到了一个新的文件中。看起来这个虚拟后的联合文件系统是可以对任何文件进行操作的，但是其实它并没有改变原来的文件，这是因为 unionfs 用到了一个重要的资管管理技术叫写时复制。

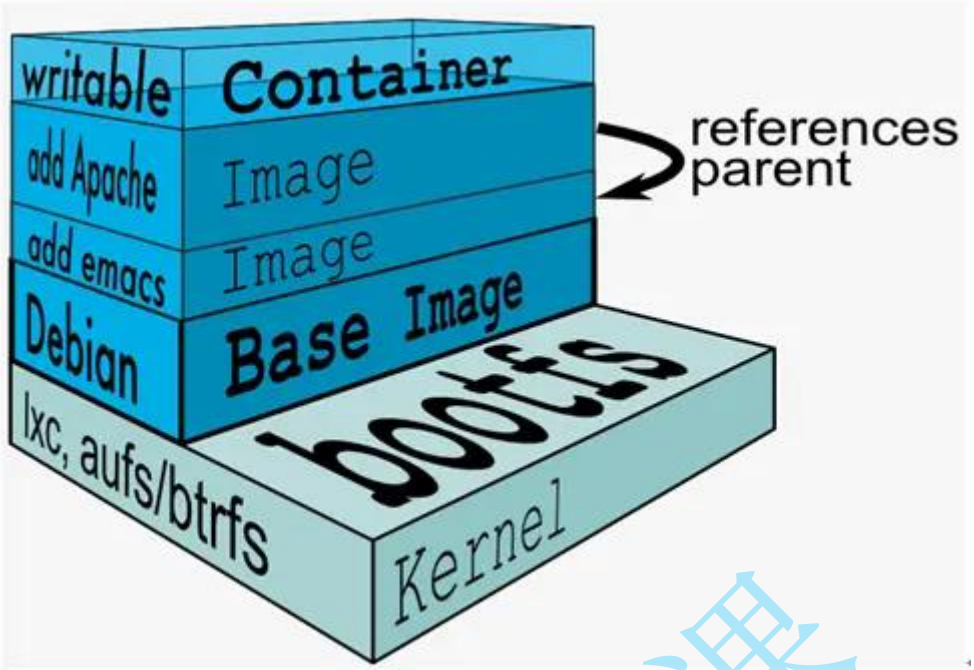
写时复制（copy-on-write，下文简称 CoW），也叫隐式共享，是一种对可修改资源实现高效复制的资源管理技术。它的思想是，如果一个资源是重复的，但没有任何修改，这时候并不需要立即创建一个新的资源；这个资源可以被新旧实例共享。创建新资源发生在第一次写操作，也就是对资源进行修改的时候。通过这种资源共享的方式，可以显著地减少未修改资源复制带来的消耗，但是也会在进行资源修改的时候增加小部分的开销。

再看 Docker 镜像是什么

image 里面是一层层文件系统 Union FS。联合文件系统，可以将几层目录挂载到一起，形成一个虚拟文件系统。虚拟文件系统的目录结构就像普通 linux 的目录结构一样，docker 通过这些文件再加上宿主机的内核提供了一个 linux 的虚拟环境。

每一层文件系统我们叫做一层 layer，联合文件系统可以对每一层文件系统设置三种权限，只读（readonly）、读写（readwrite）和写出（whiteout-able），但是 docker 镜像中每一层文件系统都是只读的。

构建镜像的时候，从一个最基本的操作系统开始，每个构建的操作都相当于做一层的修改，增加了一层文件系统。一层层往上叠加，上层的修改会覆盖底层该位置的可见性，这也很容易理解，就像上层把底层遮住了一样。当你使用的时候，你只会看到一个完全的整体，你不知道里面有几层，也不清楚每一层所做的修改是什么。



可以看到镜像分层结构有以下特性

（1）镜像共享宿主机的 kernel

（2）base 镜像是 linux 的最小发行版

（3）同一个 docker 主机支持不同的 Linux 发行版

（4）采用分层结构，可以上层引用下层，最大化的共享资源

（5）容器层位于可写层，采用 cow 技术进行修改，该层仅仅保持变化的部分，并不修改镜像下面的部分

（6）容器层以下都是只读层

（7）docker 从上到下找文件

镜像实现原理

Docker 分层存储实现原理

1. 分层存储实现方式

docker 镜像技术的基础是联合文件系统(UnionFS)，其文件系统是分层的

```
Shell
目前 docker 支持的联合文件系统有很多种，包括：AUFS、overlay、overlay2、DeviceMapper、VSF 等
```

Linux 中各发行版实现的 UnionFS 各不相同，所以 docker 在不同 linux 发行版中使用

的也不同。通过 docker info 命令可以查看当前系统所使用哪种 UnionFS，常见的几种发行版使用如下：

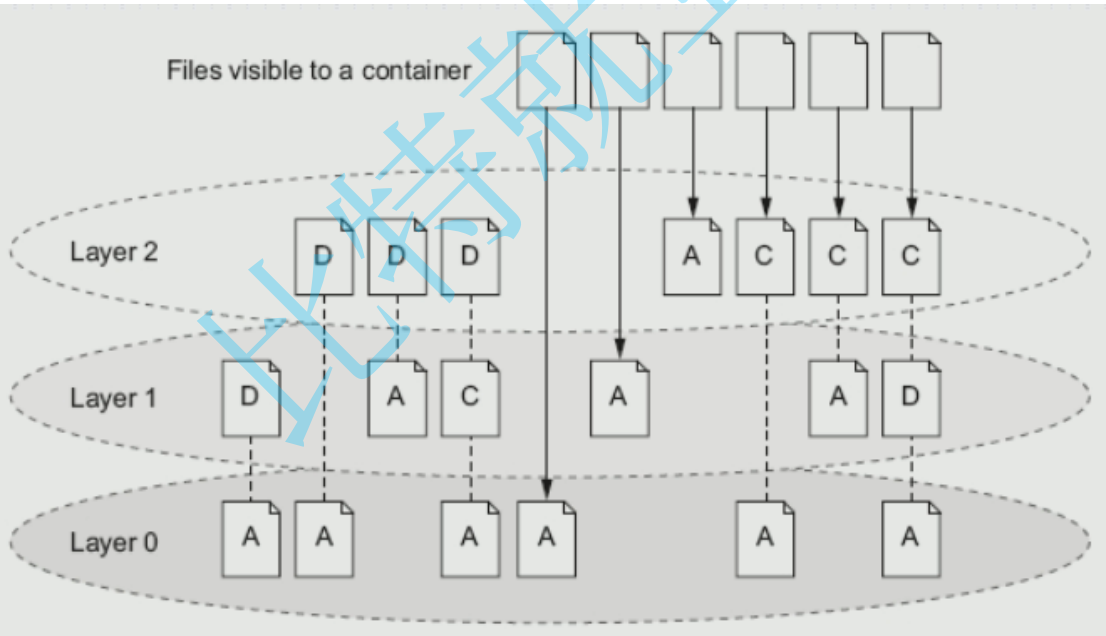
```
Shell
CentOS, Storage Driver: overlay2、overlay
debain, Storage Driver: aufs
RedHat, Storage Driver: devicemapper
```

overlay2 是 overlay 的升级版，官方推荐，更加稳定，而新版的 docker 默认也是这个驱动，linux 的内核 4.0 以上或者或使用 3.10.0-514 或更高版本内核的 RHEL 或 CentOS。

2. Union FS 的原理

docker 镜像由**多个只读层叠加面成**，启动容器时，docker 会**加载只读镜像层并在镜像栈顶部加一个读写层；**

如果运行中的容器修改了现有的一个已经存在的文件，那该文件将会从读写层下面的只读层复制到读写层，该文件版本仍然存在，只是已经被读写层中该文件的副本所隐藏，此即**“写时复制(COW)”机制**。

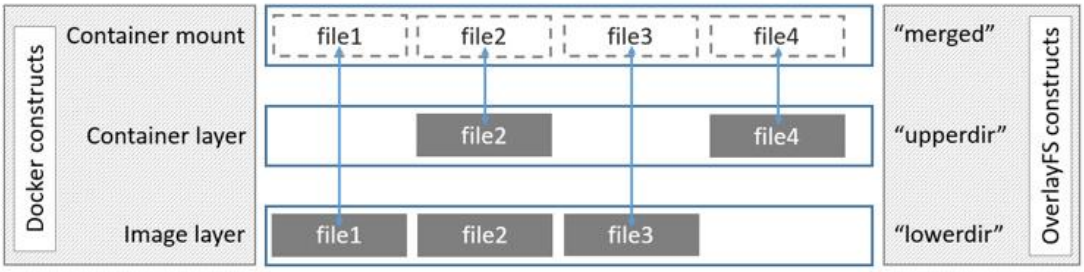


如果一个文件在最底层是可见的，如果在 layer1 上标记为删除，最高的层是用户看到的 Layer2 的层，在 layer0 上的文件，在 layer2 上可以删除，但是只是标记删除，用户是不可见的，总之在到达最顶层之前，把它标记来删除，对于最上层的用户是不可见的，当标记一删除，只有用户在最上层建一个同名一样的文件，才是可见的。

3. overlay2 实现

1. **架构**

`OverlayFS` 将单个 `Linux` 主机上的两个目录合并成一个目录。这些目录被称为层，统一过程被称为联合挂载 `OverlayFS` 底层目录称为 `lowerdir`， 高层目录称为 `upperdir`,合并统一视图称为 `merged`



图中可以看到三个层结构，即 lowerdir、upperdir、merged 层

2. **分层**

- lowerdir 层

其中 lowerdir 是只读的镜像层(image layer)，其中就包含 bootfs/rootfs 层，bootfs(boot file system)主要包含 bootloader 和 kernel，bootloader 主要是引导加载 kernel，当 boot 成功 kernel 被加载到内存中，bootfs 就被 umount 了，rootfs(root file system)包含的就是典型 Linux 系统中的/dev、/proc、/bin、/etc 等标准目录。

lowerdir 是可以分很多层的，除了 bootfs/rootfs 层以外，还可以通过 Dockerfile 建立很多 image 层

- upperdir 层

`upper` 是容器的读写层,采用了 `CoW`(写时复制)机制,只有对文件进行修改才会将文件拷贝到 `upper` 层,之后所有的修改操作都会对 `upper` 层的副本进行修改。upperdir 层是 lowerdir 的上一层，只有这一层可读可写的，其实就是 Container 层，在启动一个容器的时候会在最后的 image 层的上一层自动创建，所有对容器数据的更改都会发生在这一层。

- workdir 层

它的作用是充当一个中间层的作用,每当对 `upper` 层里面的副本进行修改时,会先当到 `workdir`,然后再从 `workdir` 移动 `upper` 层

- merged 层

是一个统一图层,从 `mergedir` 可以看到 `lower`,`upper`,`workdir` 中所有数据的整合,整个容器展现出来的就是 `mergedir` 层.merged 层就是联合挂载层，也就是给用户暴露的统一视觉，将 image 层和 container 层结合，就如最上边的图中描述一致，同一文件，在此层会展示离它最近的层级里的文件内容，或者可以理解为，只要 container 层中有此文件，便展示 container 层中的文件内容，若 container 层中没有，则展示

image 层中的。

3. **如何完成读写**

1、读：

如果文件在 upperdir(容器)层，直接读取文件；

如果文件不在 upperdir(容器)层，则从镜像层(lowerdir)读取；

2、写：

首次写入：如果 upperdir 中不存在，overlay 和 overlay2 执行 copy_up 操作，把文件从 lowdir 拷贝到 upperdir 中，由于 overlayfs 是文件级别的(即使只有很少的一点修改，也会产生 copy_up 的动作)，后续对同一文件的再次写入操作将对已经复制到容器层的文件副本进行修改，这也就是常常说的写时复制(copy-on-write)。

删除文件或目录：当文件被删除时，在容器层(upperdir)创建 whiteout 文件，镜像层(lowerdir)的文件是不会被删除的，因为它们是只读的，但 whiteout 文件会阻止它们显示，当目录被删除时，在容器层(upperdir)一个不透明的目录，这个和上边的 whiteout 的原理一样，组织用户继续访问，image 层不会发生改变

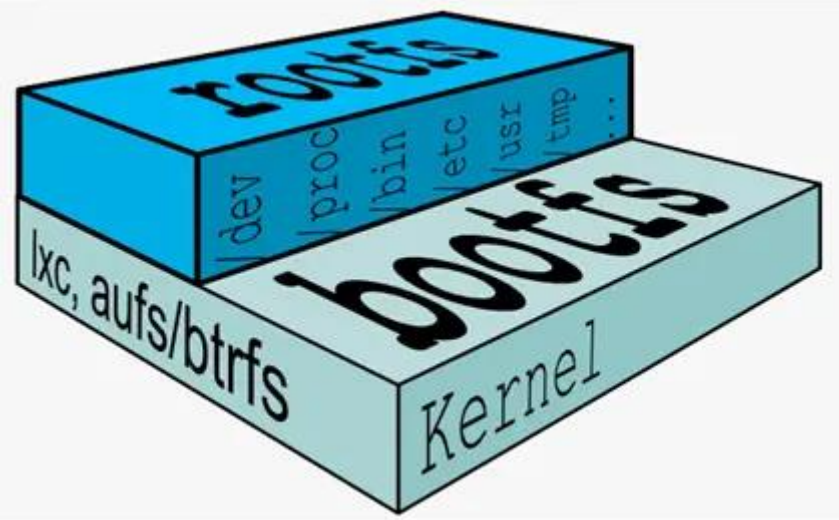
3、注意事项

copy_up 操作只发生在文件首次写入，以后都是只修改副本。

容器层的文件删除只是一个“障眼法”，是靠 whiteout 文件将其遮挡,image 层并没有删除，这也就是为什么使用 docker commit 提交保存的镜像会越来越大，无论在容器层怎么删除数据，image 层都不会改变。

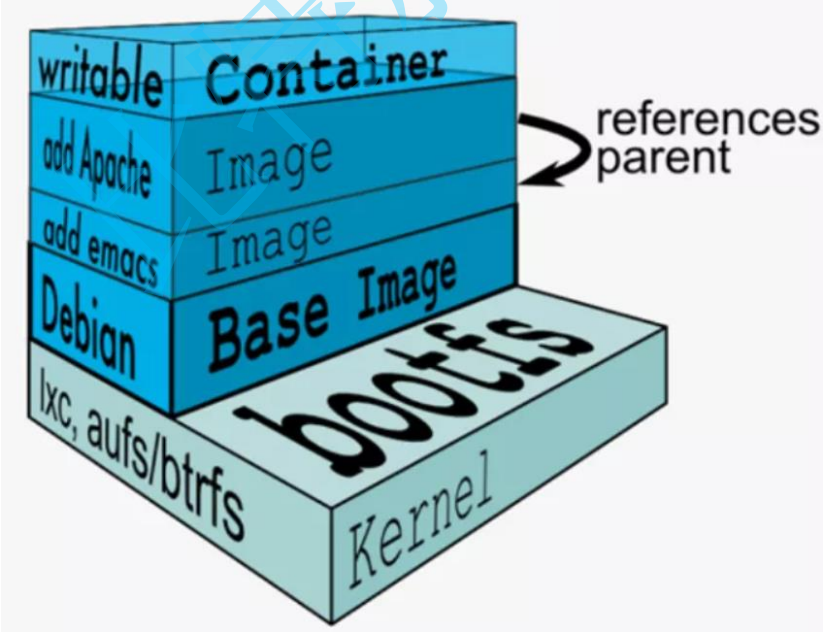
docker 镜像加载原理

boots(boot file system）主要包含 bootloader 和 Kernel, bootloader 主要是引导加 kernel,Linux 刚启动时会加 bootfs 文件系统，在 Docker 镜像的最底层是 bootfs。这一层与我们典型的 Linux/Unix 系统是一样的，包含 boot 加載器和内核。当 boot 加载完成之后整个内核就都在内存中了，此时内存的使用权已由 bootfs 转交给内核，此时系统也会卸载 bootfs。



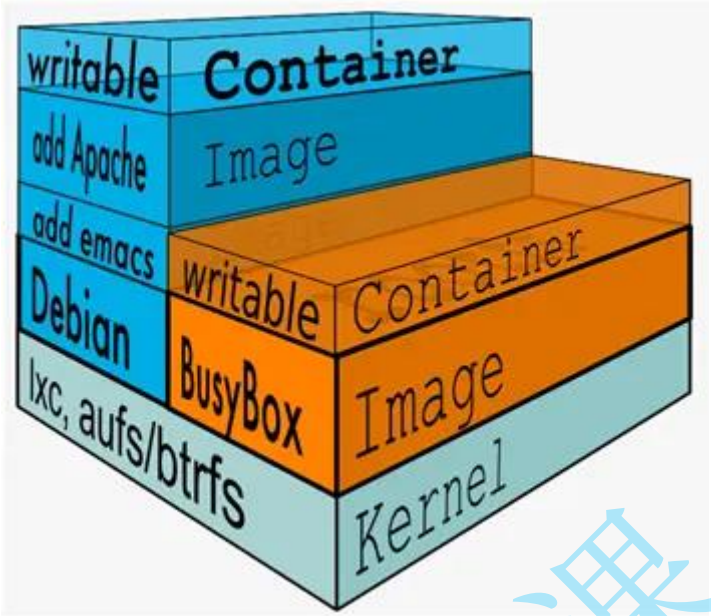
rootfs（root file system),在 bootfs 之上。包含的就是典型 Linux 系统中的 /dev,/proc,/bin,/etc 等标准目录和文件。 rootfs 就是各种不同的操作系统发行版，比如 Ubuntu,Centos 等等。

典型的 Linux 在启动后，首先将 rootfs 置为 readonly, 进行一系列检查, 然后将其切换为 “readwrite” 供用户使用。在 docker 中，起初也是将 rootfs 以 readonly 方式加载并检查，然而接下来利用 union mount 的将一个 readwrite 文件系统挂载在 readonly 的 rootfs 之上，并且允许再次将下层的 file system 设定为 readonly 并且向上叠加, 这样一组 readonly 和一个 writeable 的结构构成一个 container 的运行目录, 每一个被称作一个 Layer



下面的这张图片非常好的展示了组装的过程，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。



操作实战

实战一：镜像分层存储实战

实战目的

了解 overlay 文件系统的镜像底层实现逻辑。

基础知识

tree 命令详解

Linux tree 命令用于以树状图列出目录的内容。

执行 tree 指令，它会列出指定目录下的所有文件，包括子目录里的文件。

安装

ubuntu 安装

```Shell
apt install tree -y
```

centos 安装

```Shell
yum install tree -y
```

语法

```
Shell
tree [-aACdDfFgilnNpqstux][-I <范本样式>][-P <范本样式>][目录...]
```

常用参数说明:

- -a 显示所有文件和目录。
- -d 显示目录名称而非内容。
- -D 列出文件或目录的更改时间。
- -f 在每个文件或目录之前，显示完整的相对路径名称。
- -i 不以阶梯状列出文件或目录名称。
- -L level 限制目录显示层级。
- -l 如遇到性质为符号连接的目录，直接列出该连接所指向的原始目录。
- -P<范本样式> 只显示符合范本样式的文件或目录名称。

操作案例

```
Shell
#显示全部
[root@VM-0-6-centos mydockerfile]# tree -a
.
`-- mycppms
    |-- cppweb
    |   |-- Dockerfile
    |   |-- main.cpp
    |   `-- mycppweb
    |-- docker-compose.yml
    `-- nginx
        |-- bit.conf
        `-- Dockerfile

3 directories, 6 files
#显示目录
[root@VM-0-6-centos mydockerfile]# tree -d
.
`-- mycppms
    |-- cppweb
    `-- nginx

3 directories
```

```
#显示时间
[root@VM-0-6-centos mydockerfile]# tree -D
.
`-- [Jan 18 12:08]  mycppms
    |-- [Jan 18 14:22]  cppweb
    |   |-- [Jan 18  9:34]  Dockerfile
    |   |-- [Jan 18  9:19]  main.cpp
    |   `-- [Jan 18 11:48]  mycppweb
    |-- [Jan 18 12:08]  docker-compose.yml
    `-- [Jan 18 12:07]  nginx
        |-- [Jan 18  9:35]  bit.conf
        `-- [Jan 18  9:35]  Dockerfile

3 directories, 6 files

#显示路径
[root@VM-0-6-centos mydockerfile]# tree -f
.
`-- ./mycppms
    |-- ./mycppms/cppweb
    |   |-- ./mycppms/cppweb/Dockerfile
    |   |-- ./mycppms/cppweb/main.cpp
    |   `-- ./mycppms/cppweb/mycppweb
    |-- ./mycppms/docker-compose.yml
    `-- ./mycppms/nginx
        |-- ./mycppms/nginx/bit.conf
        `-- ./mycppms/nginx/Dockerfile

3 directories, 6 files

#取消层级
[root@VM-0-6-centos mydockerfile]# tree -i
.
mycppms
cppweb
Dockerfile
main.cpp
mycppweb
docker-compose.yml
nginx
bit.conf
Dockerfile

3 directories, 6 file
```

```
#控制显示层级
[root@VM-0-6-centos mydockerfile]# tree -L 1
.
`-- mycppms

1 directory, 0 files
#模糊匹配
[root@VM-0-6-centos mydockerfile]# tree -P bit*
.
`-- mycppms
    |-- cppweb
    `-- nginx
        `-- bit.conf
#打印软链接目录中的内容
[root@VM-0-6-centos mydockerfile]# tree -l
.
|-- myapp -> /data/maxhou/myuserapp
|   `-- data.txt
|-- mycppms
|   |-- cppweb
|   |   |-- Dockerfile
|   |   |-- main.cpp
|   |   `-- mycppweb
|   |-- docker-compose.yml
|   `-- nginx
|       |-- bit.conf
|       `-- Dockerfile
`-- test -> /usr/bin/python

4 directories, 8 files
```


实战过程

1. 拉取镜像

```
docker pull nginx:1.21.1
```

2. docker image history 可以查看镜像的分层信息，nginx 的 1.21.1 的官方基于 debian 的镜像制作文件如下

[Dockerfile-nginx1.21.1.txt]

3. 通过 docker image history 查看如下,我们可以看到 dockerfile 和做出来的镜像是对应的，而且不是每一层都占用空间的。

```
Shell
[root@VM-0-6-centos ~]# docker history nginx:1.21.1
IMAGE          CREATED       CREATED BY
SIZE      COMMENT
822b7ec2aaf2   2 years ago   /bin/sh -c #(nop)  CMD ["nginx" "-g"
"daemon…   0B
<missing>      2 years ago   /bin/sh -c #(nop)  STOPSIGNAL SIGQUIT
0B
<missing>      2 years ago   /bin/sh -c #(nop)  EXPOSE 80
0B
<missing>      2 years ago   /bin/sh -c #(nop)  ENTRYPOINT
["/docker-entr…   0B
<missing>      2 years ago   /bin/sh -c #(nop) COPY
file:09a214a3e07c919a…   4.61kB
<missing>      2 years ago   /bin/sh -c #(nop) COPY
file:0fd5fca330dcd6a7…   1.04kB
<missing>      2 years ago   /bin/sh -c #(nop) COPY
file:0b866ff3fc1ef5b0…   1.96kB
<missing>      2 years ago   /bin/sh -c #(nop) COPY
file:65504f71f5855ca0…   1.2kB
<missing>      2 years ago   /bin/sh -c set -x     && addgroup --
system -…   63.9MB
<missing>      2 years ago   /bin/sh -c #(nop)  ENV
PKG_RELEASE=1~buster      0B
<missing>      2 years ago   /bin/sh -c #(nop)  ENV
NJS_VERSION=0.6.1         0B
<missing>      2 years ago   /bin/sh -c #(nop)  ENV
NGINX_VERSION=1.21.1      0B
<missing>      2 years ago   /bin/sh -c #(nop)  LABEL
maintainer=NGINX Do…   0B
<missing>      2 years ago   /bin/sh -c #(nop)  CMD ["bash"]
0B
<missing>      2 years ago   /bin/sh -c #(nop) ADD
file:4ff85d9f6aa246746…   69.3MB
```

4. 我们通过 inspcet 命令查看，该镜像的存储位置，可以看到 GraphDriver 也就是我们的存储驱动，是 overlay2 的存储驱动，nginx 的 overlay2 的四个目录也都显示出来了，我们知道 docker 的默认目录是/var/lib/docker，之所以在/data/var/lib/docker 下面

是因为我们规划了磁盘，调整了默认的存储目录。

```
Shell
docker image inspect nginx:1.21.1
[
  ...
          "GraphDriver": {
            "Data": {
                "LowerDir":
"/data/var/lib/docker/overlay2/ceb3d04e263d1d0df854a9fd6001dda5473
65dd6ce0a10b35d2a73759f191449/diff:/data/var/lib/docker/overlay2/f
902c819258349724531e9e37bfc18667a50678a58b42bb25eca4c22e6c00907/di
ff:/data/var/lib/docker/overlay2/14096822a444ae4469397896ad70aedb1
b1e61799d3ead85089ab83914735e76/diff:/data/var/lib/docker/overlay2
/41caf1dd40c939b7821571dee0396aebe4cd4f7cd423342f85f21ae00350d287/
diff:/data/var/lib/docker/overlay2/e7b75a724095e83434f726e4a70478a
ecf35e79eb4ff8d72457616ee21df76d8/diff",
                "MergedDir":
"/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2
c279fc1682c3f3b7924ef3fb03bc5/merged",
                "UpperDir":
"/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2
c279fc1682c3f3b7924ef3fb03bc5/diff",
                "WorkDir":
"/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2
c279fc1682c3f3b7924ef3fb03bc5/work"
            },
            "Name": "overlay2"
        },

  ...
]
```

5. 我们可以看到 lowerdir，upperdir，都位于/data/var/lib/docker/overlay2 下面，因为我们调整过默认存储位置所以对比默认的/var/lib/docker 多了/data,我们在这个目录下查找我们的 nginx，看下文件是如何被存储的。搜索后可以看到我们找到了多个 nginx 文件，因为我们本地有多个 nginx 镜像所以搜到了多个 nginx 文件，通过 lowerdir 的值我们可以确定有一个是和我们 nginx:1.12.1 的匹配上的，红色部分。

```
Shell
[root@VM-0-6-centos overlay2]# tree -P nginx -f |grep
```

```
"/sbin/nginx"
|   |   |   |   `--
 ./25d1c25420c9a58e0c02d3e40e8008d5848a25f9e284b0239efe2f8d25e832a
a/diff/usr/sbin/nginx
|   |   |   |   `--
 ./41caf1dd40c939b7821571dee0396aebe4cd4f7cd423342f85f21ae00350d28
7/diff/usr/sbin/nginx
```

6. 同样的方式我们通过 Dockerfile 发现，nginx 还存储了个 docker-entrypoint.sh，我们搜索这个文件，我们发现这个文件也被放到了 diff 目录下面，和我们的 lowerdir 中一个 layer 是对应的。查看内容我们确定并没有什么加密之类的，也就是是镜像的文件分层后放到了 diff 目录下面。

```
Shell
[root@VM-0-6-centos overlay2]#  tree -P "docker-entrypoint.sh" -f
|grep "docker-entrypoint.sh"
|   |   `--
 ./14096822a444ae4469397896ad70aedb1b1e61799d3ead85089ab83914735e7
6/diff/docker-entrypoint.sh
|   |   `--
 ./f29dde59fead1ae457b19affe0e02f1ab86ee818a853380ff1ec10b07e7f7ff
1/diff/docker-entrypoint.sh
|   |               `--
 ./f9d6c7b04bbd428a61ec3acf43e0e3bbfacb451432cb72fdfbf449f5fcba75e
5/diff/usr/local/bin/docker-entrypoint.sh
```

7. 接下来我们进入 diff 的上一级目录查看，可以看到 link 文件，里面是每一个 diff 目录的短名称，或者说软链接。

```
Shell
[root@VM-0-6-centos overlay2]#
ll  ./14096822a444ae4469397896ad70aedb1b1e61799d3ead85089ab8391473
5e76/
total 16
-rw------- 1 root root    0 Jan 18 18:48 committed
drwxr-xr-x 2 root root 4096 Jan 18 18:48 diff
-rw-r--r-- 1 root root   26 Jan 18 18:48 link
-rw-r--r-- 1 root root   57 Jan 18 18:48 lower
drwx------ 2 root root 4096 Jan 18 18:48 work
[root@VM-0-6-centos overlay2]#
cat  ./14096822a444ae4469397896ad70aedb1b1e61799d3ead85089ab839147
35e76/link
QJTI7BHG2F37TD3V4VR5FOFBYV[root@VM-0-6-centos overlay2]#
```

```
[root@VM-0-6-centos overlay2]# ll ./l/QJTI7BHG2F37TD3V4VR5FOFBYV
lrwxrwxrwx 1 root root 72 Jan 18
18:48 ./l/QJTI7BHG2F37TD3V4VR5FOFBYV -
> ../14096822a444ae4469397896ad70aedb1b1e61799d3ead85089ab83914735
e76/diff
```

8. 通过遍历 l 目录我们会发现，整个 docker 的镜像的 diff 目录都被做了对应的软链接，或者说起了个短名称。

```
[root@VM-0-6-centos overlay2]# ll l
total 132
lrwxrwxrwx 1 root root 33 Jan 17 14:37 32MANU2SR3HKMYP5YTURHDKV4R -> ../fpz75txqu8s1hgkm5t449l1di/diff
lrwxrwxrwx 1 root root 33 Jan 18 09:39 3HNBNTOZ7H2WVN7FTXGKJP6F5P -> ../wf81sk8f2l2fcaoauh7yr315r/diff
lrwxrwxrwx 1 root root 72 Jan 15 15:26 3R2DYT6F72F7XBA2HXOBIDVJK6 -> ../0e354063d3efda441ce2b3655585d455bea4ac56265b8d3d6bec8
lrwxrwxrwx 1 root root 33 Jan 18 09:39 3SOTV2UYNKOFL7GDPJOSDVIAES -> ../do320gchmdrgsjjy3eezhtrm6/diff
lrwxrwxrwx 1 root root 33 Jan 16 14:44 3YLILKTVEXGVFI2VOBOXVINEEI -> ../loo7oe4w7yovj903xwjjyalsq/diff
lrwxrwxrwx 1 root root 33 Jan 17 14:37 3Z6BFNZGBTOQFXMW7YWQ667L5M -> ../bo3pgtsn754d0bsydkjjfe5yy/diff
lrwxrwxrwx 1 root root 72 Jan 17 15:55 4DYZSL57T62FCPYNAKM5VIBCQM -> ../01b99a101461025ea405e97f3810ad74b6ef87221565a3c362626
lrwxrwxrwx 1 root root 33 Jan 17 16:08 5RVE2GQRRA4PLAIRKJ2K4UBXUS -> ../uxsnp5qj50fmf6j1674kqwxi7/diff
lrwxrwxrwx 1 root root 33 Jan 17 14:39 65MBTPP5R2HHMWM4ZLYPOXKSQP -> ../3ye41pl9zy8fuu8l6pptyb15l/diff
lrwxrwxrwx 1 root root 77 Jan 20 09:22 6IX6WANY2W43ZPJ5YV3YOAZ7EL -> ../c33a829d82af9de1837cc36b580c55efa79185212bd2336fcb63a
lrwxrwxrwx 1 root root 33 Jan 16 15:18 7F7NW7BFBJTVYXRPPQDYJB4LDM -> ../gbcvaz98qyxx2txd32hukyi3b/diff
lrwxrwxrwx 1 root root 72 Jan 17 15:55 7KXFLSPJ42CFCZC7UE4LEIH5UR -> ../9f2312d6c439f5ccd71ce3157a073903ea7e15bcc390c06c47428
lrwxrwxrwx 1 root root 33 Jan 17 16:08 7OEXAXC3TWKJDMJI6KYOKOQNQJ -> ../jv69q8id3owz2qz4pbegiud05/diff
```

9. 每一个 diff 是一个层级的内容，层级的关系是存放到了 lower 文件中，里面存放着父级的层级。

```
Shell
[root@VM-0-6-centos overlay2]#
ll  ./14096822a444ae4469397896ad70aedb1b1e61799d3ead85089ab8391473
5e76/
total 16
-rw------- 1 root root    0 Jan 18 18:48 committed
drwxr-xr-x 2 root root 4096 Jan 18 18:48 diff
-rw-r--r-- 1 root root   26 Jan 18 18:48 link
-rw-r--r-- 1 root root   57 Jan 18 18:48 lower
drwx------ 2 root root 4096 Jan 18 18:48 work
[root@VM-0-6-centos overlay2]#
cat  ./14096822a444ae4469397896ad70aedb1b1e61799d3ead85089ab839147
35e76/lower
l/OTMLVWGYWVFOBDWPG7FDXYI2N5:l/WXT5VHSKBEQXADHTKXC6VMRVNQ
```

10. 最后我们查看下 mergeddir，发现 merged dir 是不存在的，当我们启动为容器的时候才是有有效的。

```
Shell
      "GraphDriver": {
            "Data": {
                "LowerDir":
"/data/var/lib/docker/overlay2/ceb3d04e263d1d0df854a9fd6001dda5473
65dd6ce0a10b35d2a73759f191449/diff:/data/var/lib/docker/overlay2/f
```

```
902c819258349724531e9e37bfc18667a50678a58b42bb25eca4c22e6c00907/di
ff:/data/var/lib/docker/overlay2/14096822a444ae4469397896ad70aedb1
b1e61799d3ead85089ab83914735e76/diff:/data/var/lib/docker/overlay2
/41caf1dd40c939b7821571dee0396aebe4cd4f7cd423342f85f21ae00350d287/
diff:/data/var/lib/docker/overlay2/e7b75a724095e83434f726e4a70478a
ecf35e79eb4ff8d72457616ee21df76d8/diff",
                "MergedDir":
"/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2
c279fc1682c3f3b7924ef3fb03bc5/merged",
                "UpperDir":
"/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2
c279fc1682c3f3b7924ef3fb03bc5/diff",
                "WorkDir":
"/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2
c279fc1682c3f3b7924ef3fb03bc5/work"
            },
            "Name": "overlay2"
        },

    }
]
[root@VM-0-6-centos overlay2]# ll
/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2c
279fc1682c3f3b7924ef3fb03bc5/merged
ls: cannot access
/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d2160f0cdabb2c
279fc1682c3f3b7924ef3fb03bc5/merged: No such file or directory
[root@VM-0-6-centos overlay2]# docker run -d --name mylayer
nginx:1.21.1
66d41536d204060f8a2ccd4a639c97902ae848803825cd696b15e81b1b0e7b27
[root@VM-0-6-centos overlay2]# docker inspect mylayer
[
    {
        "Id":
"66d41536d204060f8a2ccd4a639c97902ae848803825cd696b15e81b1b0e7b27"
,
        "Created": "2024-01-20T02:31:14.214325451Z",
        "Path": "/docker-entrypoint.sh",
        "Args": [
            "nginx",
            "-g",
            "daemon off;"
        ],
        "State": {
```

```
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 3214,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2024-01-20T02:31:14.580138624Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image":
"sha256:822b7ec2aaf2122b8f80f9c7f45ca62ea3379bf33af4e042b67aafbf6e
ac1941",
        "ResolvConfPath":
"/data/var/lib/docker/containers/66d41536d204060f8a2ccd4a639c97902
ae848803825cd696b15e81b1b0e7b27/resolv.conf",
        "HostnamePath":
"/data/var/lib/docker/containers/66d41536d204060f8a2ccd4a639c97902
ae848803825cd696b15e81b1b0e7b27/hostname",
        "HostsPath":
"/data/var/lib/docker/containers/66d41536d204060f8a2ccd4a639c97902
ae848803825cd696b15e81b1b0e7b27/hosts",
        "LogPath":
"/data/var/lib/docker/containers/66d41536d204060f8a2ccd4a639c97902
ae848803825cd696b15e81b1b0e7b27/66d41536d204060f8a2ccd4a639c97902a
e848803825cd696b15e81b1b0e7b27-json.log",
        "Name": "/mylayer",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "",
        "ExecIDs": null,
        "HostConfig": {
            "Binds": null,
            "ContainerIDFile": "",
            "LogConfig": {
                "Type": "json-file",
                "Config": {}
            },
            "NetworkMode": "default",
```

```
            "PortBindings": {},
            "RestartPolicy": {
                "Name": "no",
                "MaximumRetryCount": 0
            },
            "AutoRemove": false,
            "VolumeDriver": "",
            "VolumesFrom": null,
            "ConsoleSize": [
                34,
                156
            ],
            "CapAdd": null,
            "CapDrop": null,
            "CgroupnsMode": "host",
            "Dns": [],
            "DnsOptions": [],
            "DnsSearch": [],
            "ExtraHosts": null,
            "GroupAdd": null,
            "IpcMode": "private",
            "Cgroup": "",
            "Links": null,
            "OomScoreAdj": 0,
            "PidMode": "",
            "Privileged": false,
            "PublishAllPorts": false,
            "ReadonlyRootfs": false,
            "SecurityOpt": null,
            "UTSMode": "",
            "UsernsMode": "",
            "ShmSize": 67108864,
            "Runtime": "runc",
            "Isolation": "",
            "CpuShares": 0,
            "Memory": 0,
            "NanoCpus": 0,
            "CgroupParent": "",
            "BlkioWeight": 0,
            "BlkioWeightDevice": [],
            "BlkioDeviceReadBps": [],
            "BlkioDeviceWriteBps": [],
            "BlkioDeviceReadIOps": [],
            "BlkioDeviceWriteIOps": [],
```

```
            "CpuPeriod": 0,
            "CpuQuota": 0,
            "CpuRealtimePeriod": 0,
            "CpuRealtimeRuntime": 0,
            "CpusetCpus": "",
            "CpusetMems": "",
            "Devices": [],
            "DeviceCgroupRules": null,
            "DeviceRequests": null,
            "MemoryReservation": 0,
            "MemorySwap": 0,
            "MemorySwappiness": null,
            "OomKillDisable": false,
            "PidsLimit": null,
            "Ulimits": null,
            "CpuCount": 0,
            "CpuPercent": 0,
            "IOMaximumIOps": 0,
            "IOMaximumBandwidth": 0,
            "MaskedPaths": [
                "/proc/asound",
                "/proc/acpi",
                "/proc/kcore",
                "/proc/keys",
                "/proc/latency_stats",
                "/proc/timer_list",
                "/proc/timer_stats",
                "/proc/sched_debug",
                "/proc/scsi",
                "/sys/firmware"
            ],
            "ReadonlyPaths": [
                "/proc/bus",
                "/proc/fs",
                "/proc/irq",
                "/proc/sys",
                "/proc/sysrq-trigger"
            ]
        },
        "GraphDriver": {
            "Data": {
                "LowerDir":
"/data/var/lib/docker/overlay2/a5ab751b2f3c06f19549a75f360b9538016
bf22ad8f3247277c889cc522cf0eb-
```

```
init/diff:/data/var/lib/docker/overlay2/a9c4ea5af20e9ddd481574d216
0f0cdabb2c279fc1682c3f3b7924ef3fb03bc5/diff:/data/var/lib/docker/o
verlay2/ceb3d04e263d1d0df854a9fd6001dda547365dd6ce0a10b35d2a73759f
191449/diff:/data/var/lib/docker/overlay2/f902c819258349724531e9e3
7bfc18667a50678a58b42bb25eca4c22e6c00907/diff:/data/var/lib/docker
/overlay2/14096822a444ae4469397896ad70aedb1b1e61799d3ead85089ab839
14735e76/diff:/data/var/lib/docker/overlay2/41caf1dd40c939b7821571
dee0396aebe4cd4f7cd423342f85f21ae00350d287/diff:/data/var/lib/dock
er/overlay2/e7b75a724095e83434f726e4a70478aecf35e79eb4ff8d72457616
ee21df76d8/diff",
                "MergedDir":
"/data/var/lib/docker/overlay2/a5ab751b2f3c06f19549a75f360b9538016
bf22ad8f3247277c889cc522cf0eb/merged",
                "UpperDir":
"/data/var/lib/docker/overlay2/a5ab751b2f3c06f19549a75f360b9538016
bf22ad8f3247277c889cc522cf0eb/diff",
                "WorkDir":
"/data/var/lib/docker/overlay2/a5ab751b2f3c06f19549a75f360b9538016
bf22ad8f3247277c889cc522cf0eb/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "66d41536d204",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "80/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
",
                "NGINX_VERSION=1.21.1",
                "NJS_VERSION=0.6.1",
                "PKG_RELEASE=1~buster"
            ],
```

```
            "Cmd": [
                "nginx",
                "-g",
                "daemon off;"
            ],
            "Image": "nginx:1.21.1",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": [
                "/docker-entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": {
                "maintainer": "NGINX Docker Maintainers <docker-
maint@nginx.com>"
            },
            "StopSignal": "SIGQUIT"
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID":
"c20f300a192d42284ca3913faba6f42a0f1de78821a973ee1ebe85ab12654be8"
,
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {
                "80/tcp": null
            },
            "SandboxKey": "/var/run/docker/netns/c20f300a192d",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID":
"9b92de9ba5b457af6dc5759eb42070c938593597942a873a2777a08cab6f60c8"
,
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:02",
            "Networks": {
                "bridge": {
```

```
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID":
"5178b26f83ee47eba1331668af11cb12d2f3985fed5a77cc65562a017678ae22"
,
                    "EndpointID":
"9b92de9ba5b457af6dc5759eb42070c938593597942a873a2777a08cab6f60c8"
,
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:02",
                    "DriverOpts": null
                }
            }
        }
    }
]
[root@VM-0-6-centos overlay2]# ll
/data/var/lib/docker/overlay2/a5ab751b2f3c06f19549a75f360b9538016b
f22ad8f3247277c889cc522cf0eb/merged
total 92
drwxr-xr-x 2 root root 4096 Sep  2  2021 bin
drwxr-xr-x 2 root root 4096 Jun 13  2021 boot
drwxr-xr-x 1 root root 4096 Jan 20 10:31 dev
drwxr-xr-x 1 root root 4096 Sep  3  2021 docker-entrypoint.d
-rwxrwxr-x 1 root root 1202 Sep  3  2021 docker-entrypoint.sh
drwxr-xr-x 1 root root 4096 Jan 20 10:31 etc
drwxr-xr-x 2 root root 4096 Jun 13  2021 home
drwxr-xr-x 1 root root 4096 Sep  3  2021 lib
drwxr-xr-x 2 root root 4096 Sep  2  2021 lib64
drwxr-xr-x 2 root root 4096 Sep  2  2021 media
drwxr-xr-x 2 root root 4096 Sep  2  2021 mnt
drwxr-xr-x 2 root root 4096 Sep  2  2021 opt
drwxr-xr-x 2 root root 4096 Jun 13  2021 proc
drwx------ 2 root root 4096 Sep  2  2021 root
drwxr-xr-x 1 root root 4096 Jan 20 10:31 run
drwxr-xr-x 2 root root 4096 Sep  2  2021 sbin
drwxr-xr-x 2 root root 4096 Sep  2  2021 srv
drwxr-xr-x 2 root root 4096 Jun 13  2021 sys
```

```
drwxrwxrwt 1 root root 4096 Sep  3  2021 tmp
drwxr-xr-x 1 root root 4096 Sep  2  2021 usr
drwxr-xr-x 1 root root 4096 Sep  2  2021 var
```

实战总结

我们通过镜像实际存储位置可以看到镜像在存储的时候，通过分层来实现，并通过 link 和 lower 完成层与层之间链接关系配置，diff 存放了我们的内容，并且没有什么加密。

实战二：overlay 文件系统工作实战

实战目的

了解 overlay 的底层工作逻辑，对 docker 的镜像底层工作有深度掌握

实战步骤

1. 我们首先创建一个目录用来挂载我们的文件系统

```
Shell
mkdir -p /data/myworkdir/fs
```

2. 我们创建文件系统的工作目录

```
Shell
cd /data/myworkdir/fs
mkdir upper lower merged work
```

3. 准备一些文件

```
Shell
echo "in lower" > lower/in_lower.txt
echo "in upper" > upper/in_upper.txt

echo "In both. from lower" > lower/in_both.txt
echo "In both. from upper" > upper/in_both.txt
```

4. 挂载 overlay 目录

```
Shell
root@139-159-150-152:/data/myworkdir/fs# mount -t overlay overlay
```

```
-o lowerdir=./lower,upperdir=./upper,workdir=./work ./merged
```

5. 通过 df -h 可以看到我们完成了挂载

```
Shell
root@139-159-150-152:/data/myworkdir/fs# df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            948M     0  948M   0% /dev
tmpfs           199M  1.2M  198M   1% /run
/dev/vda1        40G   20G   19G  52% /
tmpfs           992M     0  992M   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           992M     0  992M   0% /sys/fs/cgroup
/dev/loop0       50M   50M     0 100% /snap/snapd/18357
/dev/loop2       55M   55M     0 100% /snap/erlang/101
/dev/loop3       56M   56M     0 100% /snap/core18/2708
/dev/loop4       56M   56M     0 100% /snap/core18/2714
tmpfs           199M  8.0K  199M   1% /run/user/1000
tmpfs           199M     0  199M   0% /run/user/1002
tmpfs           199M     0  199M   0% /run/user/0
overlay          40G   20G   19G  52% /data/myworkdir/fs/merged
```

6. 我们此时看下目录结构，然后发现 merged 目录自动生成了 3 个文件，可以看到 boteh,lower,upper 都在。merged 目录其实就是用户看到的目录，用户的实际文件操作在这里进行。

```
Shell
root@139-159-150-152:/data/myworkdir/fs# tree -a
.
├── lower
│   ├── in_both.txt
│   └── in_lower.txt
├── merged
│   ├── in_both.txt
│   ├── in_lower.txt
│   └── in_upper.txt
├── upper
│   ├── in_both.txt
│   └── in_upper.txt
└── work
    └── work
```

7. **编辑：**

merge 目录下编辑一下 in_lower.txt，upper 目录下就会马上出现一个 in_lower.txt，而且内容就是编辑后的。而 lower 目录下的 in_lower.txt 内容不变

```
Shell
root@139-159-150-152:/data/myworkdir/fs# cd merged/
root@139-159-150-152:/data/myworkdir/fs/merged# vi in_lower.txt

#修改内容如下：
in lower! after edit!
root@139-159-150-152:/data/myworkdir/fs/merged# cat in_lower.txt
in lower! after edit!

#再次查看目录树，可以看到 in_lower.txt 在 upper 里面生成了，发生了
copy_up
root@139-159-150-152:/data/myworkdir/fs/merged# cd ..
root@139-159-150-152:/data/myworkdir/fs# tree
.
├── lower
│   ├── in_both.txt
│   └── in_lower.txt
├── merged
│   ├── in_both.txt
│   ├── in_lower.txt
│   └── in_upper.txt
├── upper
│   ├── in_both.txt
│   ├── in_lower.txt
│   └── in_upper.txt
└── work
    └── work

5 directories, 8 files

#查看文件内容，发现是编辑后的
root@139-159-150-152:/data/myworkdir/fs# cat upper/in_lower.txt
in lower! after edit!
```

8. 如果我们删除 in_lower.txt，lower 目录里的"in_lower.txt"文件不会有变化，只是在 upper/ 目录中增加了一个特殊文件来告诉 OverlayFS，"in_lower.txt'这个文件不能出现在 merged/ 里了，类似 AuFS 的 whiteout

```
Shell
root@139-159-150-152:/data/myworkdir/fs# rm -f merged/in_lower.txt
```

```
root@139-159-150-152:/data/myworkdir/fs# tree
../
├── lower
│   ├── in_both.txt
│   └── in_lower.txt
├── merged
│   ├── in_both.txt
│   └── in_upper.txt
├── upper
│   ├── in_both.txt
│   ├── in_lower.txt
│   └── in_upper.txt
└── work
    └── work
root@139-159-150-152:/data/myworkdir/fs# ll upper/
total 16
drwxr-xr-x 2 root root 4096 Mar 16 10:39 ./
drwxr-xr-x 6 root root 4096 Mar 16 10:29 ../
-rw-r--r-- 1 root root   21 Mar 16 10:29 in_both.txt
c--------- 1 root root 0, 0 Mar 16 10:39 in_lower.txt
-rw-r--r-- 1 root root   10 Mar 16 10:29 in_upper.txt
```

注意到 upper 下 in_lower.txt 的文件类型没有 ,而是 c 不是-或者 d

实战总结

可以看到这种文件系统对于底层来说不影响，共享比较容易，但是如果编辑，删除频繁的话，性能还是比较差的。要不停的拷贝或者标记。

Docker 卷原理

Docker 卷机制

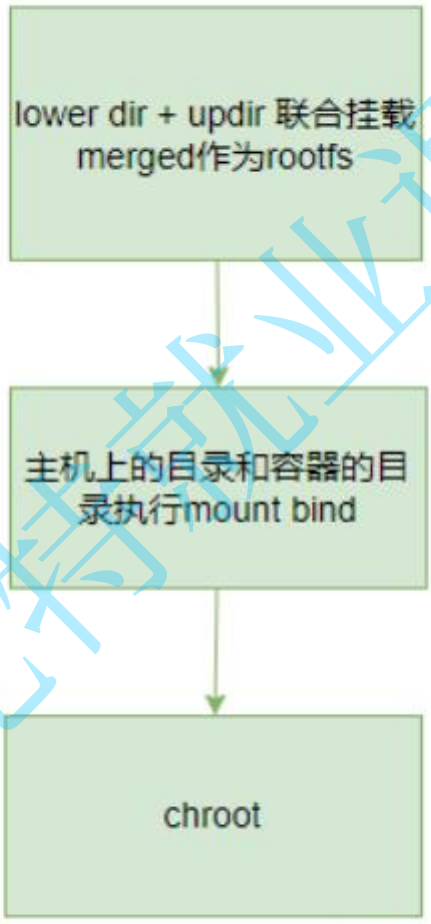
Docker 又是如何做到把一个宿主机上的目录或者文件，挂载到容器里面去呢?

当容器进程被创建之后，尽管开启了 Mount Namespace，但是在它执行 chroot（chroot 就是可以改变某进程的根目录，使这个程序不能访问目录之外的其他目录，这个跟我们在一个容器中是很相似的）之前，容器进程一直可以看到宿主机上的整个文件系统。

而宿主机上的文件系统，也自然包括了我们要使用的容器镜像。这个镜像的各个层，保存在 /var/lib/docker/overlay2/{layer id}/diff 目录下，在容器进程启动后，它们会被联合挂载在 /var/lib/docker/{layerid}/merged/ 目录中，这样容器所需的 rootfs 就准备好了。

所以，我们只需要在 rootfs 准备好之后，在执行 chroot 之前，把 Volume 指定的宿主机目录（比如 /home 目录），挂载到指定的容器目录（比如 /test 目录）在宿主机上对应的目录（即 /var/lib/docker/aufs/mnt/[可读写层 ID]/test）上，这个 Volume 的挂载工作就完成了。

由于执行这个挂载操作时，“容器进程”已经创建了，也就意味着此时 Mount Namespace 已经开启了。所以，这个挂载事件只在这个容器里可见。你在宿主机上，是看不见容器内部的这个挂载点的。这就保证了容器的隔离性不会被 Volume 打破。



操作实战

实战一、Linux mount bind

- 功能

通过 mount --bind 命令来将两个目录连接起来，mount --bind 命令是将前一个目录挂载到后一个目录上，所有对后一个目录的访问其实都是对前一个目录的访问。可以理解为硬链接。

- 语法

```
Bash
mount --bind src dest
```

- 样例

1. 创建目录

```
Bash
root@139-159-150-152:/data/myworkdir/mount# mkdir -p
/data/myworkdir/mount/data1
root@139-159-150-152:/data/myworkdir/mount# mkdir -p
/data/myworkdir/mount/data2
```

2. 目录 1 中创建文件

```
Bash
echo "Hello My bit" > /data/myworkdir/mount/data1/testfile.txt
```

3. 查看两个目录

```
Bash
root@139-159-150-152:/data/myworkdir/mount# ll
/data/myworkdir/mount/data1
total 12
drwxr-xr-x 2 root root 4096 Mar 25 17:39 ./
drwxr-xr-x 6 root root 4096 Mar 25 17:38 ../
-rw-r--r-- 1 root root   13 Mar 25 17:39 testfile.txt
root@139-159-150-152:/data/myworkdir/mount# ll
/data/myworkdir/mount/data2
total 8
drwxr-xr-x 2 root root 4096 Mar 25 17:38 ./
```

```
drwxr-xr-x 6 root root 4096 Mar 25 17:38 ../
```

4. 绑定目录

```
Shell
mount --bind /data/myworkdir/mount/data1
/data/myworkdir/mount/data2
```

5. 查看目录

```
Bash
root@139-159-150-152:/data/myworkdir/mount# ll
/data/myworkdir/mount/data2
total 12
drwxr-xr-x 2 root root 4096 Mar 25 17:39 ./
drwxr-xr-x 6 root root 4096 Mar 25 17:38 ../
-rw-r--r-- 1 root root   13 Mar 25 17:39 testfile.txt
root@139-159-150-152:/data/myworkdir/mount# ll
/data/myworkdir/mount/data1
total 12
drwxr-xr-x 2 root root 4096 Mar 25 17:39 ./
drwxr-xr-x 6 root root 4096 Mar 25 17:38 ../
```

6. 修改文件，可以看到马上反应到另外一个目录

```
JavaScript
root@139-159-150-152:/data/myworkdir/mount# echo "Hello bit after
my edit" >> /data/myworkdir/mount/data1/testfile.txt
root@139-159-150-152:/data/myworkdir/mount# cat
/data/myworkdir/mount/data2/testfile.txt
Hello My bit
Hello bit after my edit
```

实战二、查看 Docker 联合挂载

1. 启动一个容器，挂载一个目录

```
Bash
docker run --rm -d --name mynginx -p 80:80 -v
test_volume:/usr/share/nginx/html nginx:1.22.1
```

2. 查看 docker inspect 容器查看容器存储信息信息

```JSON
docker inspect mynginx

...
"GraphDriver": {
            "Data": {
                "LowerDir":
"/data/var/lib/docker/overlay2/27bc85a04695d475066edb2573cf6b0dc5b
4fdea45665a88303275cadcc6c257-
init/diff:/data/var/lib/docker/overlay2/c34284778740cc002763d1a9b8
5b808bc840d2a84d97a0322d4e33392d3cd931/diff:/data/var/lib/docker/o
verlay2/53b30605c7d4af184c010ada7247cc9eed2a710c6a6415256de855eb3a
4100cb/diff:/data/var/lib/docker/overlay2/912ac8acad715e79bdc49cb1
1201ad6b5c747933f1a91aa64525d774ccf44a42/diff:/data/var/lib/docker
/overlay2/b4f013e22ed0ca22c2a9277303a017285dbeda5ec86f166aeaf0e45c
cf5038f2/diff:/data/var/lib/docker/overlay2/bf3249bf7fe5efb140234b
eb7b38de54865f3a8bc5dc7c720499d2ed7e66135f/diff:/data/var/lib/dock
er/overlay2/68c8286bbe73c386eaefc263a21049dfc62a3b20f2c6c0874e9baa
3e684b0d9e/diff",
                "MergedDir":
"/data/var/lib/docker/overlay2/27bc85a04695d475066edb2573cf6b0dc5b
4fdea45665a88303275cadcc6c257/merged",
                "UpperDir":
"/data/var/lib/docker/overlay2/27bc85a04695d475066edb2573cf6b0dc5b
4fdea45665a88303275cadcc6c257/diff",
                "WorkDir":
"/data/var/lib/docker/overlay2/27bc85a04695d475066edb2573cf6b0dc5b
4fdea45665a88303275cadcc6c257/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [
            {
                "Type": "volume",
                "Name": "test_volume",
                "Source":
"/data/var/lib/docker/volumes/test_volume/_data",
                "Destination": "/usr/share/nginx/html",
                "Driver": "local",
                "Mode": "z",
                "RW": true,
                "Propagation": ""
```

```
            }
        ],
```

3. 执行 mount 查看一个容器的 meged 其实是 upper 和 lower 挂载而成

```
Plain Text
root@139-159-150-152:/data/myworkdir/mount# mount |grep overlay
|grep
27bc85a04695d475066edb2573cf6b0dc5b4fdea45665a88303275cadcc6c257
overlay on
/data/var/lib/docker/overlay2/27bc85a04695d475066edb2573cf6b0dc5b4
fdea45665a88303275cadcc6c257/merged type overlay
(rw,relatime,lowerdir=/data/var/lib/docker/overlay2/l/KBX6CEMOXXF7
K5V2LJX45D2ARF:/data/var/lib/docker/overlay2/l/AFNMYBLUBEEGANRBCH2
X6BMGNE:/data/var/lib/docker/overlay2/l/SU4ZC43MX63BIGVJUVINXNSUZ3
:/data/var/lib/docker/overlay2/l/RWKLSQEXMLQ2MSDFCYBQ4ZJ6C5:/data/
var/lib/docker/overlay2/l/F2SWB5ER5A34JFQ636AMQ3D7Q5:/data/var/lib
/docker/overlay2/l/FNY2R6K3QLFC7A5NTZX5AXJXZB:/data/var/lib/docker
/overlay2/l/7LURKUQVHOFX4CUNJ5QA2SV3J4,upperdir=/data/var/lib/dock
er/overlay2/27bc85a04695d475066edb2573cf6b0dc5b4fdea45665a88303275
cadcc6c25/diff,workdir=/data/var/lib/docker/overlay2/27bc85a04695d
475066edb2573cf6b0dc5b4fdea45665a88303275cadcc6c257/work,xino=off)
```

实战三、Docker 卷深度思考（docker commit 能提交卷里面的内容么）

容器的镜像操作，比如 docker commit，都是发生在宿主机空间的。而由于 Mount Namespace 的隔离作用，宿主机并不知道这个绑定挂载的存在。所以，在宿主机看来，容器中可读写层的 /test 目录（/var/lib/docker/overlay2/{layerid}/[可读写层 ID]/test），始终是空的。在宿主机的眼里 /test 的 inode 就是原本没有重定向的 inode，你所有的修改都不在这个文件下。

不过，由于 Docker 一开始还是要创建 /test 这个目录作为挂载点，所以执行了 docker commit 之后，你会发现新产生的镜像里，会多出来一个空的 /test 目录。

实际跑个案例看下

1. 启动容器，挂载下文件

```
Bash
docker run --rm -d --name mynginx -p 80:80 -v test_volume:/test
```

```
nginx:1.22.1
```

2. 进入容器的挂载宿主机目录，修改 index.html

```
JavaScript
root@139-159-150-152:/data/myworkdir/mount# cd
/data/var/lib/docker/volumes/test_volume/_data
root@139-159-150-
152:/data/var/lib/docker/volumes/test_volume/_data# echo "Hello
bit for test commit " > index.html
```

3. 容器中查看

```
Bash
root@139-159-150-
152:/data/var/lib/docker/volumes/test_volume/_data# docker exec
mynginx cat /test/index.html
Hello bit for test commit
```

4. 执行 docker commit 提交容器为新镜像

```
JavaScript
root@139-159-150-
152:/data/var/lib/docker/volumes/test_volume/_data# docker commit
mynginx mycommit:v0.1
sha256:94ffd1b2035b9130893de027a1770eaef1e7c3cb7c79706c9766eb9d0d7
96044
root@139-159-150-
152:/data/var/lib/docker/volumes/test_volume/_data# docker images
|grep mycommit:v0.1
root@139-159-150-
152:/data/var/lib/docker/volumes/test_volume/_data# docker images
|grep mycomm
mycommit                                            v0.1
94ffd1b2035b   28 seconds ago   142MB
```

5. 然后查看新镜像中文件，可以看到只有一个空目录

```
SQL
root@139-159-150-
152:/data/var/lib/docker/volumes/test_volume/_data# docker run --
rm -it mycommit:v0.1 ls -l /test
```

```
total 0
root@139-159-150-
152:/data/var/lib/docker/volumes/test_volume/_data# docker run --
rm -it mycommit:v0.1 ls -l /
total 76
drwxr-xr-x   2 root root 4096 Feb 27 00:00 bin
drwxr-xr-x   2 root root 4096 Dec  9 19:15 boot
drwxr-xr-x   5 root root  360 Mar 25 10:11 dev
drwxr-xr-x   1 root root 4096 Mar  1 18:43 docker-entrypoint.d
-rwxrwxr-x   1 root root 1616 Mar  1 18:43 docker-entrypoint.sh
drwxr-xr-x   1 root root 4096 Mar 25 10:11 etc
drwxr-xr-x   2 root root 4096 Dec  9 19:15 home
drwxr-xr-x   1 root root 4096 Feb 27 00:00 lib
drwxr-xr-x   2 root root 4096 Feb 27 00:00 lib64
drwxr-xr-x   2 root root 4096 Feb 27 00:00 media
drwxr-xr-x   2 root root 4096 Feb 27 00:00 mnt
drwxr-xr-x   2 root root 4096 Feb 27 00:00 opt
dr-xr-xr-x 221 root root    0 Mar 25 10:11 proc
drwx------   2 root root 4096 Feb 27 00:00 root
drwxr-xr-x   1 root root 4096 Mar 25 10:06 run
drwxr-xr-x   2 root root 4096 Feb 27 00:00 sbin
drwxr-xr-x   2 root root 4096 Feb 27 00:00 srv
dr-xr-xr-x  13 root root    0 Mar 25 10:11 sys
drwxr-xr-x   2 root root 4096 Mar 25 10:06 test
drwxrwxrwt   1 root root 4096 Mar  1 18:43 tmp
drwxr-xr-x   1 root root 4096 Feb 27 00:00 usr
drwxr-xr-x   1 root root 4096 Feb 27 00:00 var
```

Docker 网络原理

Linux 常见网络虚拟化

什么是虚拟网卡

虚拟网卡（又称虚拟网络适配器），即用软件模拟网络环境，模拟网络适配器。简单来说就是软件模拟的网卡。

虚拟网卡:tun/tap

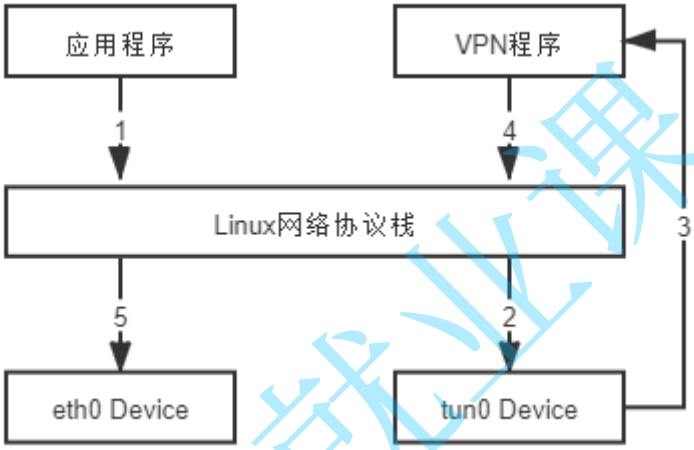
- **简介**

tap/tun 虚拟了一套网络接口，这套接口和物理的接口无任何区别，可以配置 IP，可以路由流量，不同的是，它的流量只在主机内流通。

tun 和 tap 是两个相对独立的虚拟网络设备，其中 tap 模拟了以太网设备，操作二层数据包（以太帧），tun 则模拟了网络层设备，操作三层数据包（IP 报文）。

使用 tun/tap 设备的目的是实现把来自协议栈的数据包先交由某个打开了 /dev/net/tun 字符设备的用户进程处理后，再把数据包重新发回到链路中。你可以通俗地将它理解为这块虚拟化网卡驱动一端连接着网络协议栈，另一端连接着用户态程序，而普通的网卡驱动则是一端连接着网络协议栈，另一端连接着物理网卡。

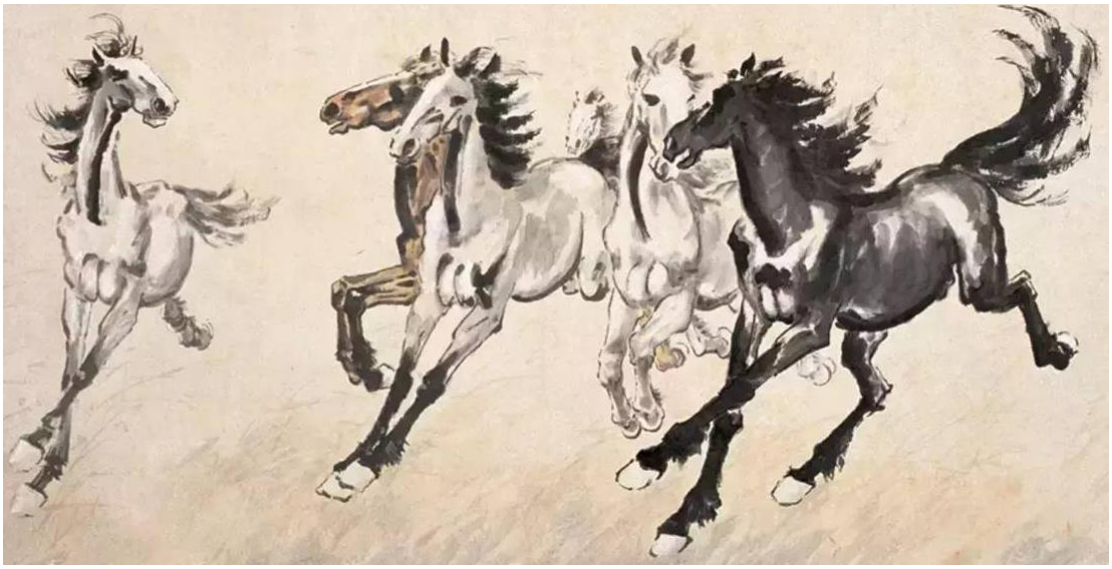
最典型的 VPN 应用程序为例，程序发送给 tun 设备的数据包



应用程序通过 tun 设备对外发送数据包后，tun 设备，便会把数据包通过字符设备发送给 VPN 程序，VPN 收到数据包，会修改后再重新封装成新报文，譬如数据包原本是发送给 A 地址的，VPN 把整个包进行加密，然后作为报文体，封装到另一个发送给 B 地址的新数据包当中。然后通过协议栈发送到物理网卡发送出去。

使用 tun/tap 设备传输数据需要经过两次协议栈，不可避免地会有一定的性能损耗，所以引入了新的网卡实现方式 veth.

- **生活案例**



虚拟网卡就像游戏中的战马至于实际生活中的战马，在游戏中，人物可以骑着虚拟的马战斗。

- **实战**

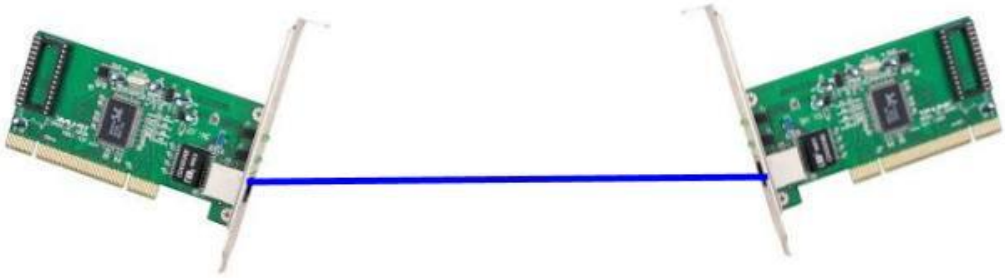
虚拟网卡实战

虚拟网卡:veth

- **简介**

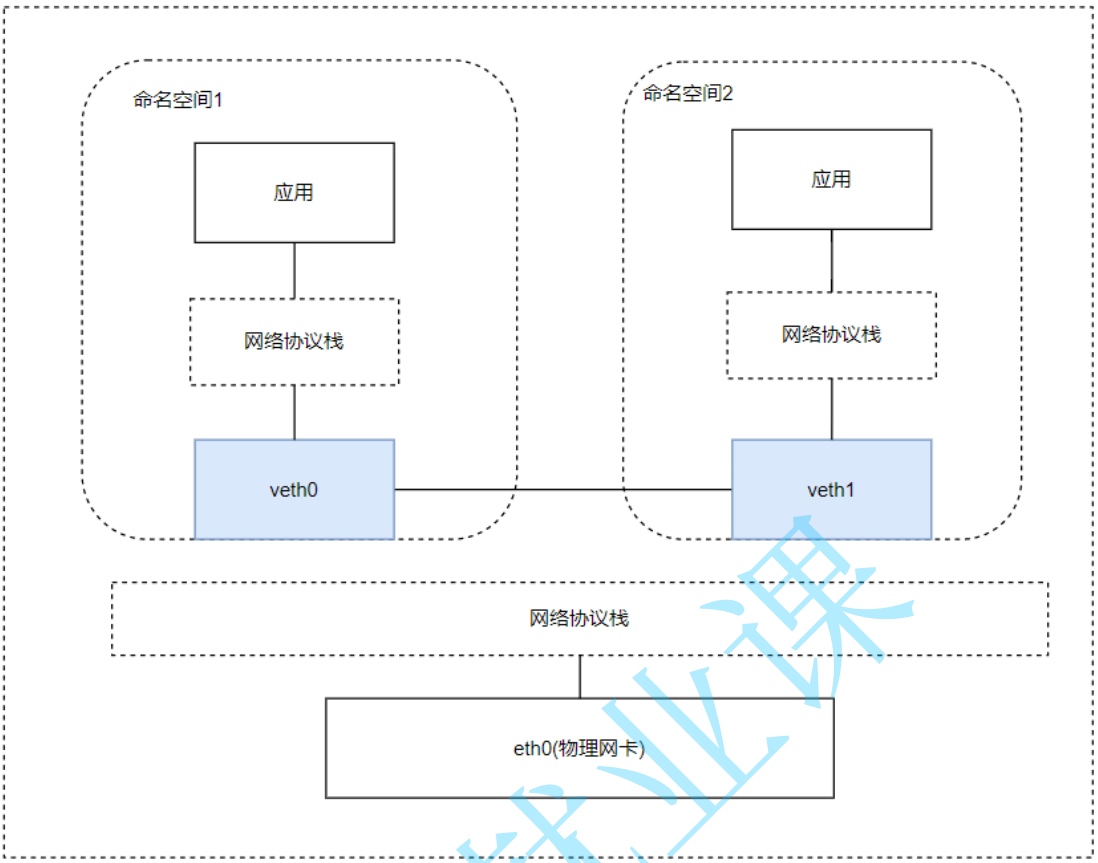
在 Linux Kernel 2.6 版本，Linux 开始支持网络名空间隔离的同时，也提供了专门的虚拟以太网（Virtual Ethernet，习惯简写做 veth）让两个隔离的网络名称空间之间可以互相通信。

直接把 veth 比喻成是虚拟网卡其实并不十分准确，如果要和物理设备类比，它应该相当于由交叉网线连接的一对物理网卡。形象化的理解如下:



veth 实际上不是一个设备，而是一对设备，因而也常被称作 veth pair。要使用 veth，必须在两个独立的网络名称空间中进行才有意义，因为 veth pair 是一端连着协议栈，另一端彼此相连的，在 veth 设备的其中一端输入数据，这些数据就会从设备的另外一端原样不变地流出.

veth pair



veth 通信不需要反复多次经过网络协议栈，这让 veth 比起 tap/tun 具有更好的性能。veth 实现了点对点的虚拟连接，可以通过 veth 连接两个 namespace，如果我们需要将 3 个或者多个 namespace 接入同一个二层网络时，就不能只使用 veth 了。在物理网络中，如果需要连接多个主机，我们会使用网桥，或者又称为交换机。Linux 也提供了网桥的虚拟实现。

- **生活案例**

veth 就像一座桥，直接连接两个地方。

- **实战**

veth 实战

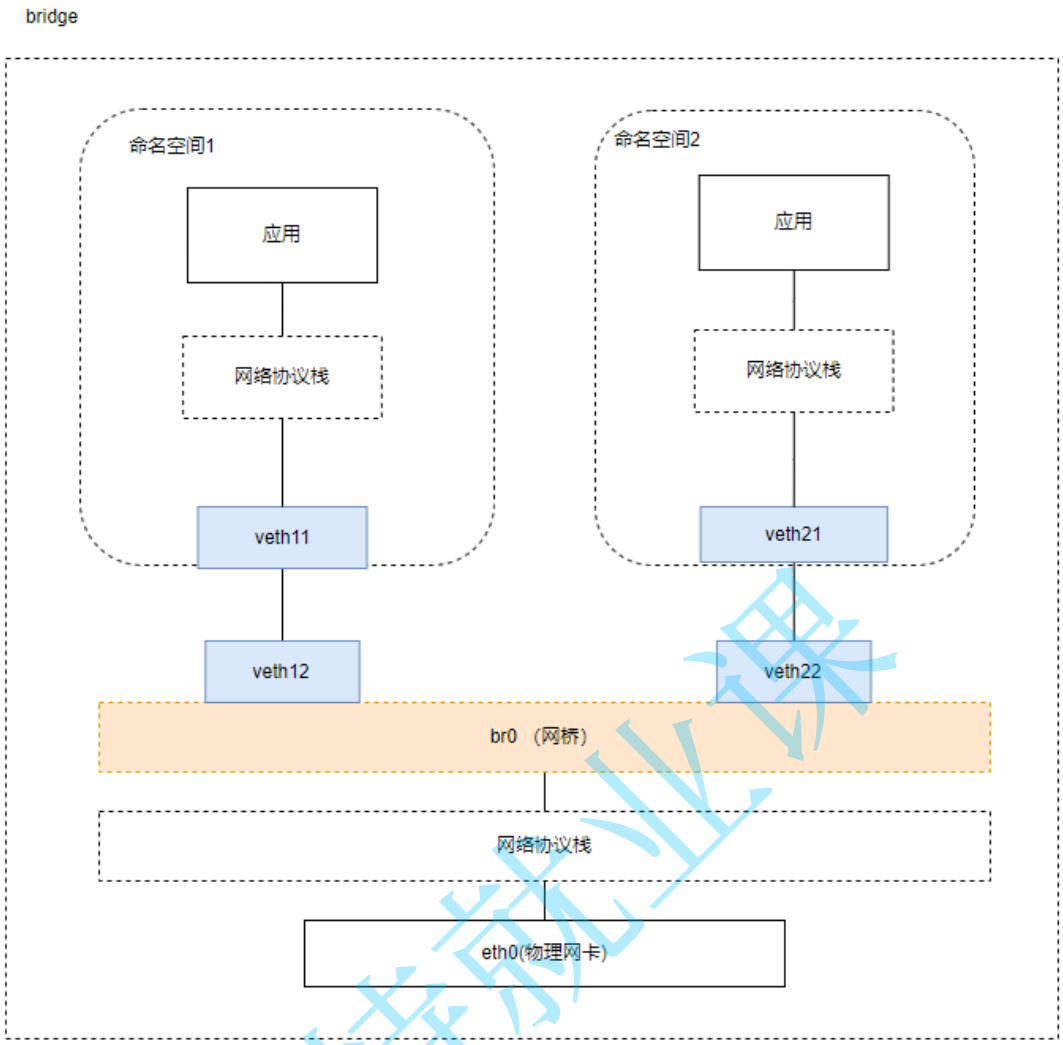
虚拟交换机

- **简介**

使用 veth pair 将两个隔离的 netns 连接在了一起,在现实世界里等同于用一根网线把两台电脑连接在了一起，但是在现实世界里往往很少会有人这样使用。因为一台设备不仅仅只需要和另一台设备通信，它需要和很多很多的网络设备进行通信，如果还使用这样的方式，需要十分复杂的网络接线，并且现实世界中的普通网络设备也没有那么多网络接口。

那么，想要让某一台设备和很多网络设备都可以通信需要如何去做呢？在我们的日常生活中，除了手机和电脑，最常见的网络设备就是路由器了，我们的手机连上 WI-FI，电脑插到路由器上，等待从路由器的 DHCP 服务器上获取到 IP，他们就可以相互通信了，这便是路由器的二层交换功能在工作。Linux Bridge 最主要的功能就是二层交换，是对现实世界二层交换机的模拟.

Linux Bridge，由 `brctl` 命令创建和管理。Linux Bridge 创建以后，真实的物理设备（如 eth0）抑或是虚拟的设备（veth 或者 tap）都能与 Linux Bridge 配合工作。



有了虚拟化网络设备后，下一步就是要使用这些设备组成网络。

- **生活案例**

交换机就像立交桥，有很多直接相连的道路组成。

- **实战**

网桥实战

虚拟组网 VxLan

物理网络的拓扑结构是相对固定的。云原生时代的分布式系统的逻辑拓扑结构变动频率，譬如服务的扩缩、断路、限流，等等，都可能要求网络跟随做出相应的变化。

正因如此，软件定义网络（Software Defined Network，SDN）的需求在云计算和分布式时代变得前所未有地迫切，SDN 的核心思路是在物理的网络之上再构造一层虚拟化的网络。

SDN 里位于下层的物理网络被称为 Underlay，它着重解决网络的连通性与可管理性，位于上层的逻辑网络被称为 Overlay，它着重为应用提供与软件需求相符的传输服务和网络拓扑。

vlan

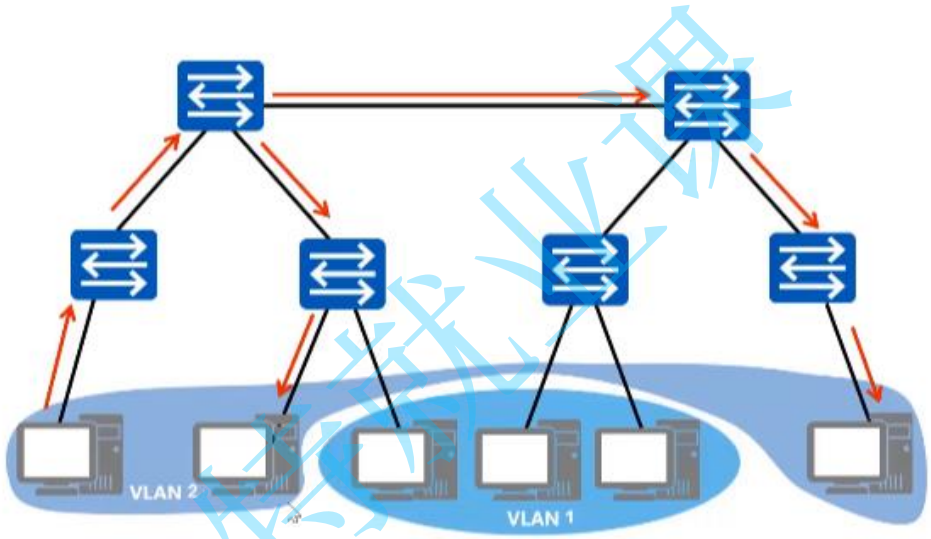
交换机是一个 L2 设备，插在同一个交换机的网络设备组成了一个 L2 网络，L2 网络之间通过 MAC 地址通信，同时这个 L2 网络也是一个广播域。

同属于一个广播域的两个设备想要通信，一设备须得向网络中的所有设备发送请求信息，只有对应 MAC 地址的设备才是真正的接收方，但实际上却是数据帧传遍整个网络，所有设备都会收到，且直接丢弃。

如此一来，将造成一系列不好的后果：网络整体带宽被占用、潜在的信息安全风险、占用 CPU 资源……

因此，VLAN 应运而生！

Vlan(Virtual Local Area Network)即虚拟局域网，是一个将物理局域网在逻辑上划分成多个广播域技术。通过在交换机上配置 Vlan，可以实现在同一 Vlan 用户可以进行二层互访，在不同 Vlan 间的用户被二层隔离，这样既能够隔离广播域，又可以提升网络安全性。



VLAN 究竟能够解决什么问题？

1、限制广播域。广播域被限制在一个局域网内，节省了带宽，提高了网络处理能力。

2、增强局域网的安全性。不同局域网内的报文在传输时是相互隔离的，即一个 VLAN 内的用户不能和其它 VLAN 内的用户直接通信，如果不同 VLAN 要进行通信，则需要通过路由器或三层交换机等三层设备。

3、灵活构建虚拟工作组。用局域网可以划分不同的用户到不同的工作组，同一工作组的用户也不必局限于某一固定的物理范围，网络构建和维护更方便灵活。

不过 VLAN 也并非没有缺点。

1、随着虚拟化技术的发展，一台物理服务器往往承载了多台虚拟机，公有云或其它大型虚拟化云数据中心动辄需容纳上万甚至更多租户，VLAN 技术最多支持 4000 多个 VLAN，逐渐无法满足需求。

2、公有云提供商的业务要求将实体网络租借给多个不同的用户，这些用户对于网络的

要求有所不同，而不同用户租借的网络有很大的可能会出现 IP 地址、MAC 地址的重叠。传统的 VLAN 并没有涉及这个问题，因此需要一种新的技术来保证在多个租户网络中存在地址重叠的情况下依旧能有效通信的技术。

3、虚拟化技术使得单台主机可以虚拟化出多台虚拟机同时运行，而每台虚拟机都会有其唯一的 MAC 地址。这样，为了保证集群中所有虚机可以正常通信，交换机必须保存每台虚机的 MAC 地址，这样就导致了交换机中的 MAC 表异常庞大，从而影响交换机的转发性能。

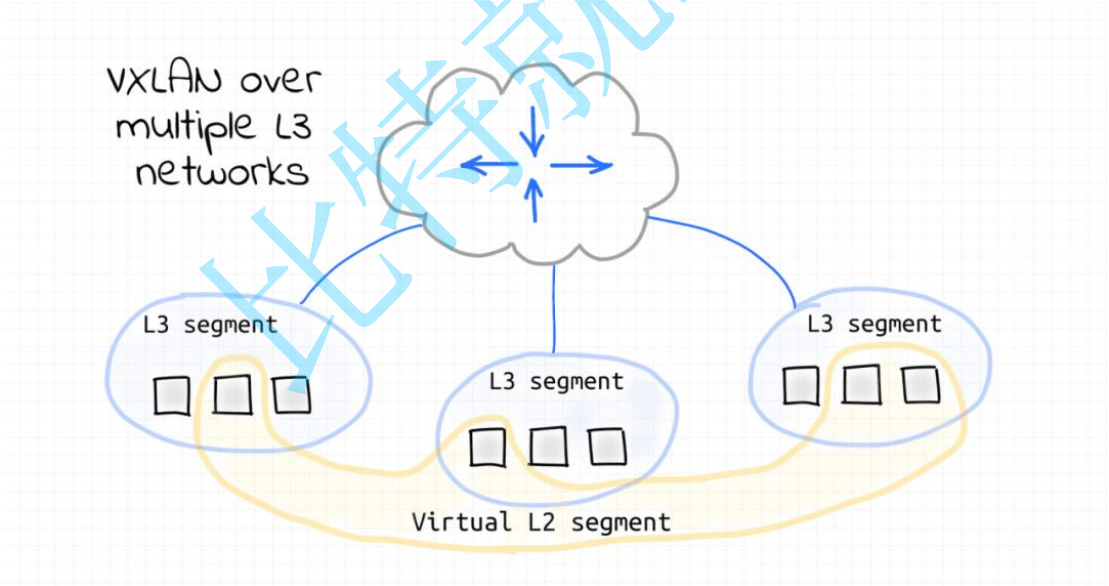
vxlan

VXLAN 是另一种网络虚拟化技术，有点类似于 VLAN，但功能更强大。

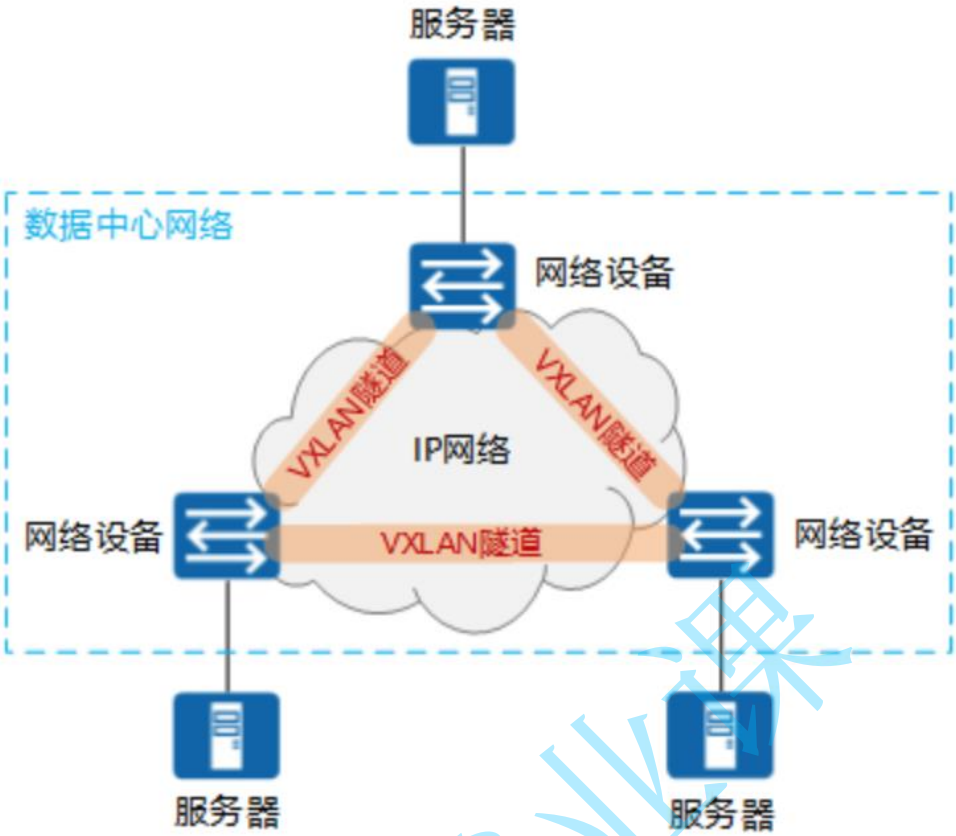
在传统的 VLAN 网络中，共享同一底层 L2 网段的 VLAN 不能超过 4096 个。只有 12 比特用于对 Ethernet Frame 格式中的 VLAN ID 字段进行编码。

VXLAN 协议定义了 8 个字节的 VXLAN Header，引入了类似 VLAN ID 的网络标识，称为 VNI（VXLAN Network ID），由 24 比特组成，这样总共是 1600 多万个，从而满足了大规模不同租户之间的标识、隔离需求。

VXLAN 是基于 L3 网络构建的虚拟 L2 网络，是一种 Overlay 网络。VXLAN 不关心底层物理网络拓扑，它将 Ethernet Frame 封装在 UDP 包中，只要它能承载 UDP 数据包，就可以在远端网段之间提供以太网 L2 连接。



每个 VXLAN 节点上的出站 L2 Ethernet Frame 都会被捕获，然后封装成 UDP 数据包，并通过 L3 网络发送到目标 VXLAN 节点。当 L2 Ethernet Frame 到达 VXLAN 节点时，就从 UDP 数据包中提取（解封装），并注入目标设备的网络接口。这种技术称为隧道。因此，VXLAN 节点会创建一个虚拟 L2 网段，从而创建一个 L2 广播域。

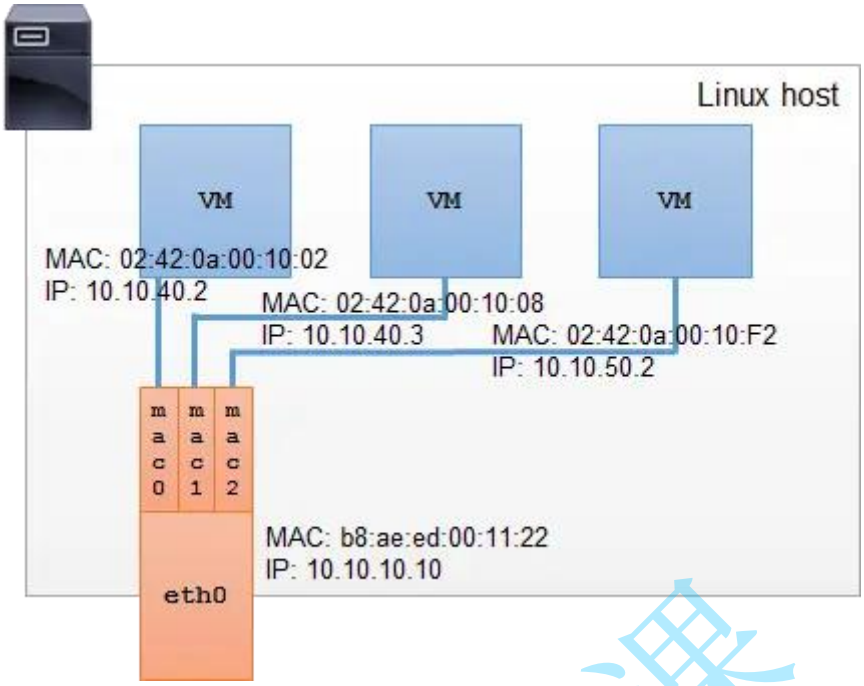

服务器
数据中心网络
网络设备
VXLAN隧道
VXLAN隧道
IP网络
网络设备
VXLAN隧道
网络设备
服务器
服务器

- 生活案例

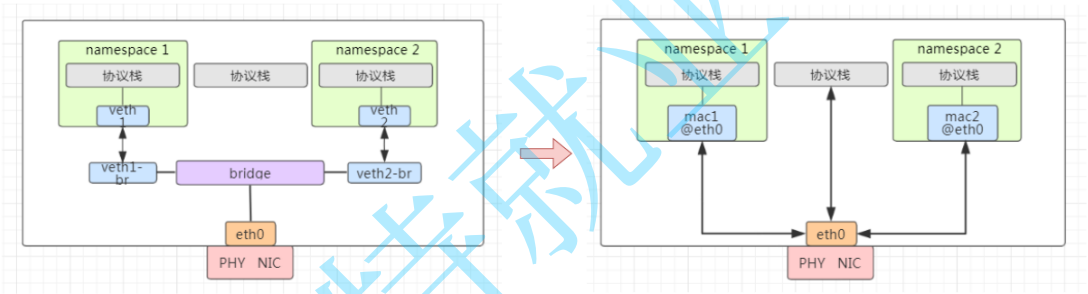
vxlan 就像中国的路网，一样连接了很多城市。

MacVLan

MACVLAN 允许对同一个网卡设置多个 IP 地址，还允许对同一张网卡上设置多个 MAC 地址，这也是 MACVLAN 名字的由来。原本 MAC 地址是网卡接口的“身份证”，应该是严格的一对一关系，而 MACVLAN 打破这层关系，方法是在物理设备之上、网络栈之下生成多个虚拟的 Device，每个 Device 都有一个 MAC 地址，新增 Device 的操作本质上相当于在系统内核中注册了一个收发特定数据包的回调函数，每个回调函数都能对一个 MAC 地址的数据包进行响应，当物理设备收到数据包时，会先根据 MAC 地址进行一次判断，确定交给哪个 Device 来处理。



可以看到 bridge 到 macvlan 缩短了通讯链路，性能比较高



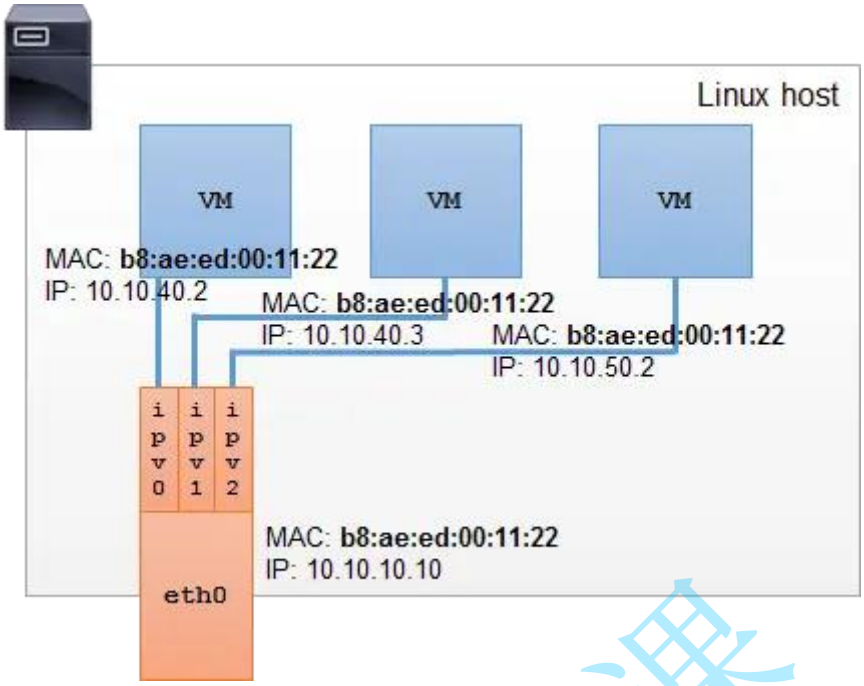
许多网卡在硬件层面对支持的 MAC 地址数量存在限制。超过该限制会导致性能的下降。所以 macvlan 虽然近乎完美还是没有实际推广完成。

- 生活案例

macvlan 相当于一个人有多个身份证。

IPVLan

Ipvlan 与 macvlan 非常相似，但又存在显著不同。Ipvlan 的子接口上并不拥有独立的 MAC 地址。所有共享父接口 MAC 地址的子接口拥有各自独立的 IP。



共享 MAC 地址会影响 DHCP 相关的操作。如果虚拟机、容器需通过 DHCP 获取网络配置，请确保它们在 DHCP 请求中使用各自独立的 ClientID；DHCP 服务器会根据请求中的 ClientID 而非 MAC 地址来分配 IP 地址。某些设备不支持分配多个 ip 地址所以 IPVlan 也没有实际大规模推广。

- 生活案例

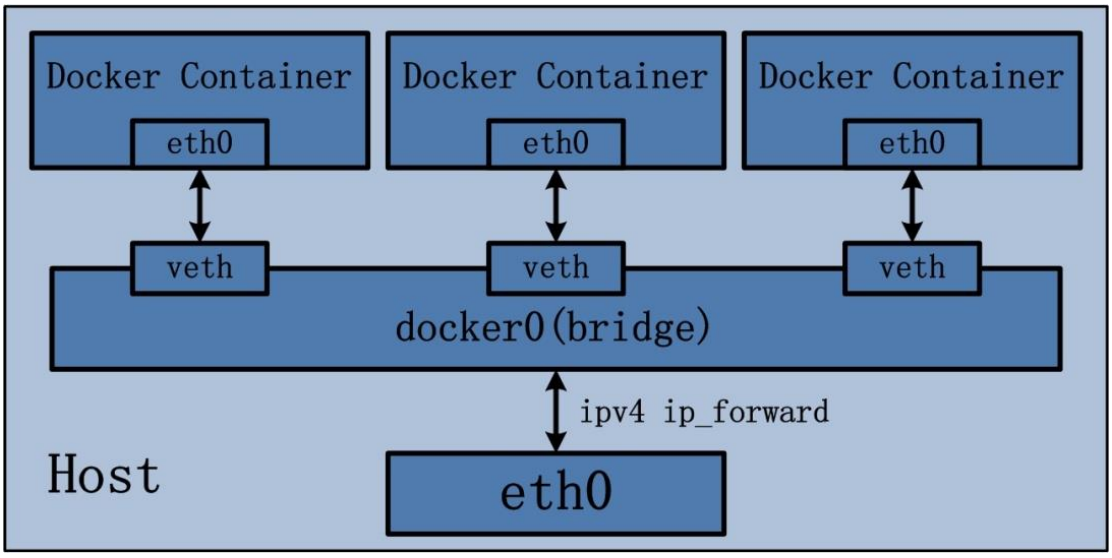
ipvlan 相当于一个身份证对应了多个人。

docker 网络分类

docker 常见的网络的分类如下，其中 bridge 网络、host 网络、container 网络、none 网络、overlay 网络、macvlan 网络、ipvlan 网络。其中 overlay 网络往往配合 swarm 和 k8s 来使用。

1. bridge 网络

bridge 驱动会在 Docker 管理的主机上创建一个 Linux 网桥。默认情况下，网桥上的容器可以相互通信。也可以通过 bridge 驱动程序配置，实现对外部容器的访问。桥接网络如下

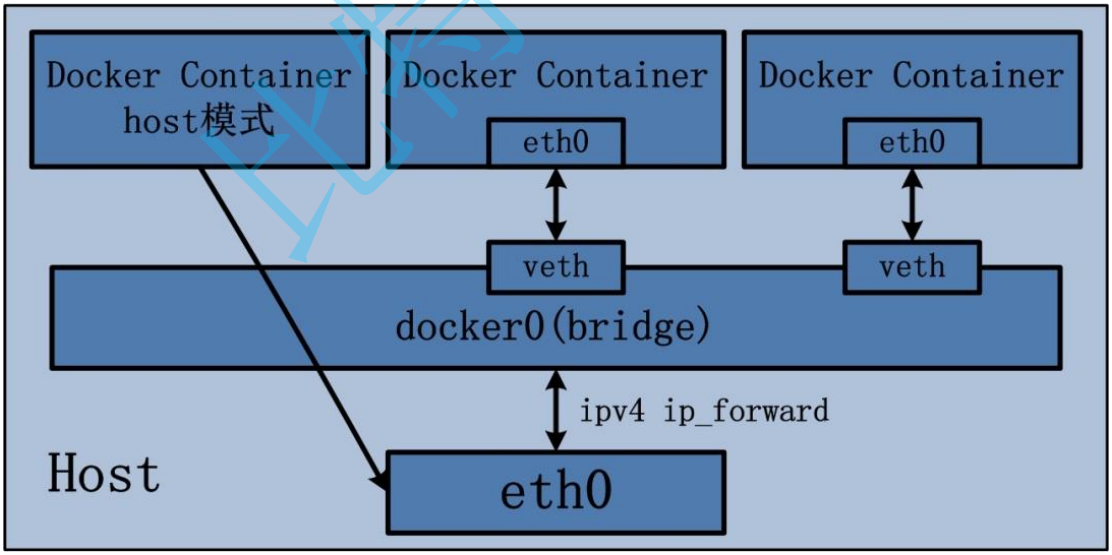


创建命令如下：

```
Shell
 docker network create -d bridge bridgenet1
```

2. host 网络

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

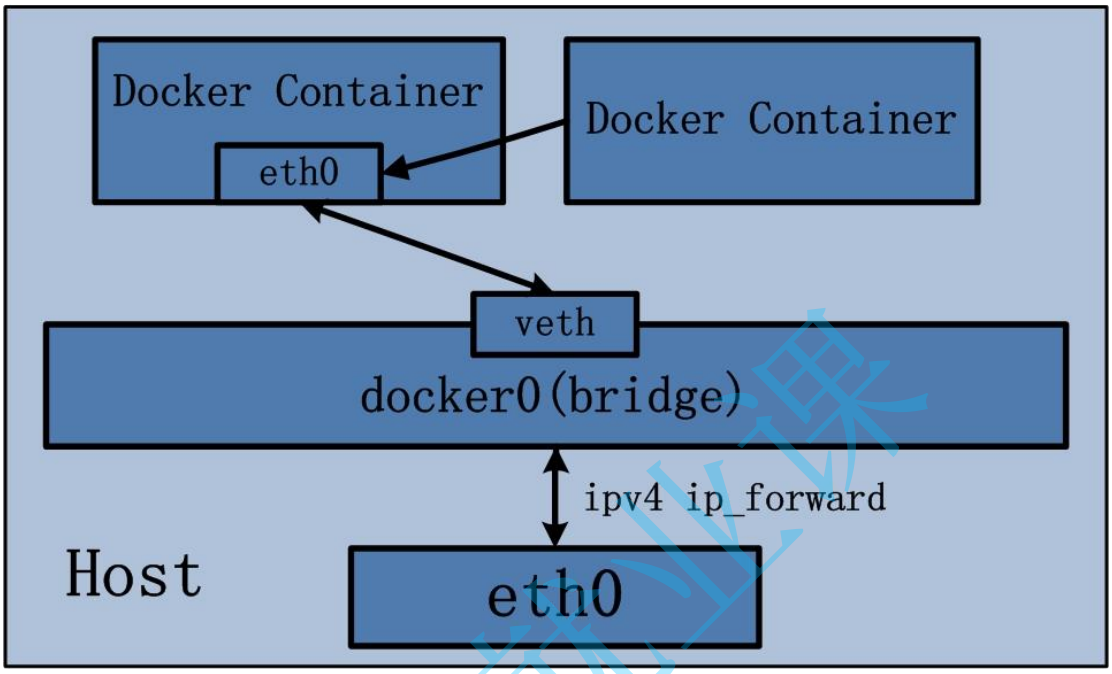


host 网络创建命令如下

```
Shell
 docker run --name c2 -itd --network=host busybox
```

3. container 网络

这个模式指定新创建的容器和引进存在的一个容器共享一个 network namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 ip，而是和一个指定的容器共享 ip，端口等，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信。容器网络原理如下:



容器网络创建命令参考如下:

```Shell
docker run -itd --name netcontainer2 --network
container:netcontainer1 busybox
```

4. none 网络

Docker 容器拥有自己的 Network Namespace，但是，并不为 Docker 容器进行任何网络配置。也就是说，这个 Docker 容器没有网卡、IP、路由等信息。需要我们自己为 Docker 容器添加网卡、配置 IP 等。

```Shell
docker  run -itd --name c3 --network none busybox
```

5. overlay 网络

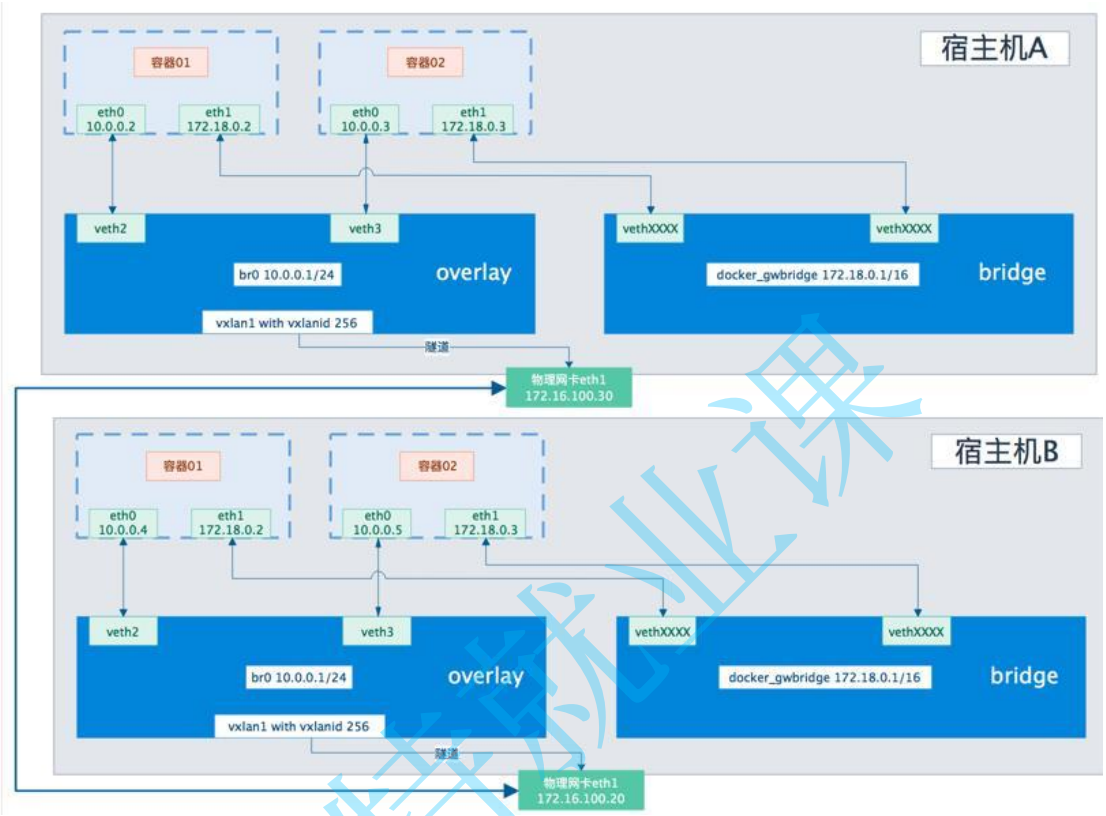
Overlay 驱动创建一个支持多主机网络的覆盖网络。

在不改变现有网络基础设施的前提下，通过某种约定通信协议，把二层报文封装在 IP 报文之上的新的数据格式。

Overlay 网络实际上是目前最主流的容器跨节点数据传输和路由方案，底层原理

VXLAN.

Overlay 网络将多个 Docker 守护进程连接在一起，允许不同机器上相互通讯，同时支持对消息进行加密，实现跨主机的 docker 容器之间的通信，Overlay 网络将多个 Docker 守护进程连接在一起，使 swarm 服务能够相互通信。这种策略消除了在这些容器之间进行操作系统级路由的需要。下图是个典型的 overlay 网络



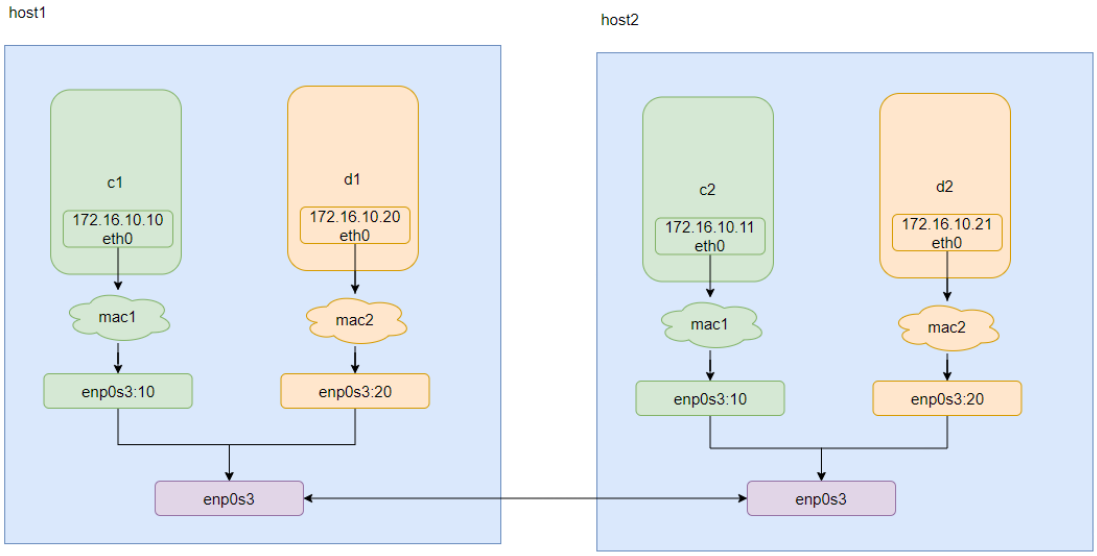
overlay 网络的创建命令如下

```Shell
docker network create -d overlay ovnet1
```

6. macvlan 网络

为每个容器的虚拟网络接口分配一个 MAC 地址，使其看起来是直接连接到物理网络的物理网络接口。在这种情况下，您需要在 Docker 主机上指定一个物理接口以用于 .macvlan 以及 子网和网关 macvlan。您甚至可以 macvlan 使用不同的物理网络接口隔离您的网络。注意要把对应的网卡开启混杂模式命令为 ifconfig 网卡名称(如 eth0) promisc

macvlan 典型的拓扑如下:

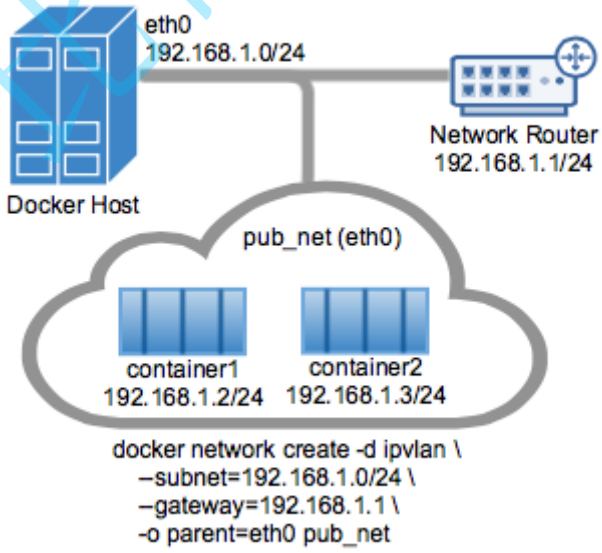


macvlan 的创建命令如下

```shell
docker network create -d macvlan --subnet=172.16.10.0/24 --
gateway=172.16.10.1 -o parent=eth0 mac1
```

7. ipvlan 网络

ipvlan 和 overlay 都可以实现不同主机上的容器之间的通讯，但是 ipvlan 是所有容器都在一个网段，相当于在一个 vlan 里面，然后可以通过不同的子接口对应不同网段，实现不同容器之间的通讯。而 overlay 可以实现不同网段之间的通讯。ipvlan 的 l2 网络示例如下：



ipvlan 的创建命令参考如下：

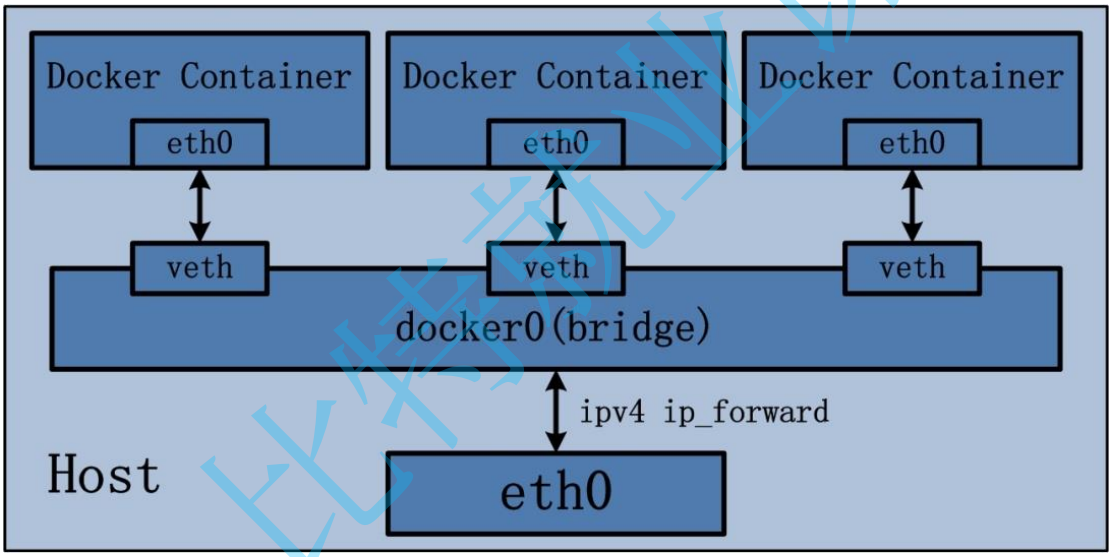
```
Shell
docker network create -d ipvlan  --subnet=192.168.1.0/24   --
gateway=192.168.1.1  -o ipvlan_mode=l2  -o parent=eth0 pub_net
```

深入理解 docker Bridge 网络

网络介绍

Docker Bridge 网络采用内置的 bridge 驱动，bridge 驱动底层采用的是 Linux 内核中 Linux bridge 技术。就网络而言，bridge 网络是在网络段之间转发流量的链路层设备，而网桥可以是在主机内核中运行的硬件设备或软件设备；就 Docker 而言，桥接网络使用软件网桥 `docker0`，它允许连接到同一网桥网络的容器进行通信，同时提供与未连接到该网桥网络容器的隔离。

Docker Container 的 bridge 桥接模式可以参考下图：



- 创建步骤

Bridge 桥接模式的实现步骤主要如下：

1. Docker Daemon 利用 veth pair 技术，在宿主机上创建两个虚拟网络接口设备，假设为 veth0 和 veth1。而 veth pair 技术的特性可以保证无论哪一个 veth 接收到网络报文，都会将报文传输给另一方。

2. Docker Daemon 将 veth0 附加到 Docker Daemon 创建的 docker0 网桥上。保证宿主机的网络报文可以发往 veth0；

3. Docker Daemon 将 veth1 添加到 Docker Container 所属的 namespace 下，并被改名为 eth0。如此一来，保证宿主机的网络报文若发往 veth0，则立即会被 eth0 接收，实现宿主机到 Docker Container 网络的联通性；同时，也保证 Docker Container 单独

使用 eth0，实现容器网络环境的隔离性。

操作实战

实战一、docker bridge 的 veth pair 如何对应

1. 我们运行一个 busybox 容器

```
Shell
docker run --name netcontainer1 -itd busybox
```

2. 我们在容器中上查看 pair 信息

```
Shell
root@139-159-150-152:/data/myworkdir/mysql-data/user# docker exec
-it netcontainer1 sh
/ # ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
536: eth0@if537: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500
qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
/ # cat /sys/class/net/eth0/iflink
537
```

3. 宿主机上我们执行 ip link,找到 id 为 537 的这样两个 pair 就对应上了。

```
Shell
root@139-159-150-152:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel
state UP mode DEFAULT group default qlen 1000
    link/ether fa:16:3e:9d:f4:ac brd ff:ff:ff:ff:ff:ff
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP mode DEFAULT group default
    link/ether 02:42:d7:87:d1:1b brd ff:ff:ff:ff:ff:ff
537: veth600bcb3@if536: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether 3e:a8:bc:98:4d:90 brd ff:ff:ff:ff:ff:ff link-
netnsid 0
539: veth81c87ff@if538: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
```

```
qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether 5a:35:04:aa:f3:71 brd ff:ff:ff:ff:ff:ff link-
netnsid 1
```

实战二、容器与宿主机/外界主机通信原理

- 实现原理

Bridge 桥接模式，从原理上实现了 Docker Container 到宿主机乃至其他机器的网络连通性。然而，由于宿主机的 IP 地址与 veth pair 的 IP 地址均不在同一个网段，故仅仅依靠 veth pair 和 namespace 的技术，还不足以是宿主机以外的网络主动发现 Docker Container 的存在。为了使得 Docker Container 可以让宿主机以外的世界感知到容器内部暴露的服务，Docker 采用 NAT（Network Address Translation，网络地址转换）的方式，让宿主机以外的世界可以主动将网络报文发送至容器内部。

具体来讲，当 Docker Container 需要暴露服务时，内部服务必须监听容器 IP 和端口号 port_0，以便外界主动发起访问请求。由于宿主机以外的世界，只知道宿主机 eth0 的网络地址，而并不知道 Docker Container 的 IP 地址，哪怕就算知道 Docker Container 的 IP 地址，从二层网络的角度来讲，外界也无法直接通过 Docker Container 的 IP 地址访问容器内部应用。因此，Docker 使用 NAT 方法，将容器内部的服务监听的端口与宿主机的某一个端口 port_1 进行“绑定”。

如此一来，外界访问 Docker Container 内部服务的流程为：

1. 外界访问宿主机的 IP 以及宿主机的端口 port_1；
2. 当宿主机接收到这样的请求之后，由于 DNAT 规则的存在，会将该请求的目的 IP（宿主机 eth0 的 IP）和目的端口 port_1 进行转换，转换为容器 IP 和容器的端口 port_0；
3. 由于宿主机认识容器 IP，故可以将请求发送给 veth pair；
4. veth pair 的 veth0 将请求发送至容器内部的 eth0，最终交给内部服务进行处理。

使用 DNAT 方法，可以使得 Docker 宿主机以外的世界主动访问 Docker Container 内部服务。那么 Docker Container 如何访问宿主机以外的世界呢。以下简要分析 Docker Container 访问宿主机以外世界的流程：

1. Docker Container 内部进程获悉宿主机以外服务的 IP 地址和端口 port_2，于是 Docker Container 发起请求。容器的独立网络环境保证了请求中报文的源 IP 地址为容器 IP（即容器内部 eth0），另外 Linux 内核会自动为进程分配一个可用源端口（假设为 port_3）；
2. 请求通过容器内部 eth0 发送至 veth pair 的另一端，到达 veth0，也就是到达了网桥（docker0）处；

3. docker0 网桥开启了数据报转发功能（/proc/sys/net/ipv4/ip_forward），故将请求发送至宿主机的 eth0 处；

4. 宿主机处理请求时，使用 SNAT 对请求进行源地址 IP 转换，即将请求中源地址 IP（容器 IP 地址）转换为宿主机 eth0 的 IP 地址；

5. 宿主机将经过 SNAT 转换后的报文通过请求的目的 IP 地址（宿主机以外世界的 IP 地址）发送至外界。

在这里，很多人肯定会问：对于 Docker Container 内部主动发起对外的网络请求，当请求到达宿主机进行 SNAT 处理后发给外界，当外界响应请求时，响应报文中的目的 IP 地址肯定是 Docker 宿主机的 IP 地址，那响应报文回到宿主机的时候，宿主机又是如何转给 Docker Container 的呢？关于这样的响应，由于 port_3 端口并没有在宿主机上做相应的 DNAT 转换，原则上不会被发送至容器内部。为什么说对于这样的响应，不会做 DNAT 转换呢。原因很简单，DNAT 转换是针对容器内部服务监听的特定端口做的，该端口是供服务监听使用，而容器内部发起的请求报文中，源端口号肯定不会占用服务监听的端口，故容器内部发起请求的响应不会在宿主机上经过 DNAT 处理。

其实，这一环节的内容是由 iptables 规则来完成，具体的 iptables 规则如下：

```
Shell
iptables -I FORWARD -o docker0 -m conntrack --ctstate
RELATED,ESTABLISHED -j ACCEPT
```

这条规则的意思是，在宿主机上发往 docker0 网桥的网络数据报文，如果是该数据报文所处的连接已经建立的话，则无条件接受，并由 Linux 内核将其发送到原来的连接上，即回到 Docker Container 内部。

`IP-Forwarding` IP 转发。 一种路由协议。IP 转发是操作系统的一种选项，支持主机起到路由器的功能。在一个系统中含有两块以上的网卡，并将 IP 转发选项打开，这样该系统就可以作为路由器进行使用了。

- **实战操作**

`iptables` 组成 Linux 平台下的包过滤防火墙，与大多数的 Linux 软件一样，这个包过滤防火墙是免费的，它可以代替昂贵的商业防火墙解决方案，完成封包过滤、封包重定向和网络地址转换等功能。

- `-t` 指定表名，在 iptables 中，有四张表：filter：这里面的链条，规则，可以决定一个数据包是否可以到达目标进程端口；mangle: 这里面的链条，规则，可以修改数据包的内容，比如 ttl；nat：这里面的链条，规则，可以修改源和目标的 ip 地址，从而进行包路由；raw：这里面的链条，规则，能基于数据包的状态进行规则设定
- `-n` 使用数字形式（numeric）显示输出结果
- `-v` 查看规则表详细信息（verbose）的信息

- `-L` 列出（list）指定链中所有的规则进行查看

下面我们可以通过 `iptables` 命令来查看 `NAT` 转换表

```
Shell
[zsc@VM-8-12-centos ~]$ sudo iptables -t nat -nvL
[sudo] password for zsc:
Chain PREROUTING (policy ACCEPT 410 packets, 19228 bytes)
 pkts bytes target     prot opt in     out     source
destination
1791K  104M DOCKER     all  --  *      *       0.0.0.0/0
0.0.0.0/0            ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 410 packets, 19228 bytes)
 pkts bytes target     prot opt in     out     source
destination

Chain OUTPUT (policy ACCEPT 359 packets, 22606 bytes)
 pkts bytes target     prot opt in     out     source
destination
 1308 78480 DOCKER     all  --  *      *
0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type
LOCAL

Chain POSTROUTING (policy ACCEPT 360 packets, 22646 bytes)
 pkts bytes target     prot opt in     out     source
destination
    0     0 MASQUERADE  all  --  *      !br-0aa8840cec0c
172.18.0.0/16        0.0.0.0/0
    2    96 MASQUERADE  all  --  *      !docker0  172.17.0.0/16
0.0.0.0/0
    0     0 MASQUERADE  all  --  *      !br-45ac8acddba4
172.19.0.0/16        0.0.0.0/0
    2   168 MASQUERADE  all  --  *      !br-5ab488f2287c
172.18.0.0/16        0.0.0.0/0
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.7
172.17.0.7           tcp dpt:80

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source
destination
    0     0 RETURN     all  --  br-0aa8840cec0c *       0.0.0.0/0
0.0.0.0/0
    0     0 RETURN     all  --  docker0 *       0.0.0.0/0
0.0.0.0/0
```

```
    4   204 DNAT       tcp  --  !docker0 *       0.0.0.0/0
0.0.0.0/0           tcp dpt:8088 to:172.17.0.7:80
```

看到红色部分的表项，该表项类型为 `DNAT(Destination NAT)`，`tcp dpt:8088 to:172.17.0.7:80` 表示将发往本机 `8088` 端口的流量转发到容器的 `80` 端口。

实战三、容器的网络命名空间查找

基础知识

nsenter 命令是一个可以在指定进程的命名空间下运行指定程序的命令。它位于 util-linux 包中。一个最典型的用途就是进入容器的网络命令空间。相当多的容器为了轻量级，是不包含较为基础的命令的，比如说 ip address，ping，telnet，ss，tcpdump 等等命令，这就给调试容器网络带来相当大的困扰：只能通过 docker inspect ContainerID 命令获取到容器 IP，以及无法测试和其他网络的连通性。这时就可以使用 nsenter 命令仅进入该容器的网络命名空间，使用宿主机的命令调试容器网络。

- 语法

```
Shell
nsenter [options] <program> [<argument>...]
```

- 常用参数

参数	含义
-t, --target <pid>	进入目标进程的命名空间
-i, --ipc[=<file>]	进入 IPC 空间
-m, --mount[=<file>]	进入 Mount 空间
-n, --net[=<file>]	进入 Net 空间
-p, --pid[=<file>]	进入 Pid 空间
-u, --uts[=<file>]	进入 UTS 空间
-U, --user[=<file>]	进入用户空间
-V, --version	版本查看

- 案例

```
Shell

ip netns add ns111
#看到的是宿主机的网络信息
ifconfig -a
ll /var/run/netns/
nsenter --net=/var/run/netns/ns111
#看到的是隔离空间的信息
ifconfig -a
```

实战

1. ip netns list 查找 docker 网络命名空间

docker 使用 namespace 实现容器网络，但是使用 ip netns 命令却无法在主机上看到任何 docker 容器的 network namespace，这是因为默认 docker 把创建的网络命名空间链接文件隐藏起来了。

ip netns 默认是去检查/var/run/netns 目录的。而 Docker 容器对应的 ns 信息记录到了/var/run/docker/netns 目录。所以 ip netns 查出来就是空的

```
Shell
root@139-159-150-152:/var/run/docker/netns# ll
/var/run/docker/netns/
total 0
drwxr-xr-x 2 root root 100 Mar 17 09:51 ./
drwx------ 8 root root 180 Mar 10 16:40 ../
-r--r--r-- 1 root root   0 Mar 16 21:56 22e66c104e59
-r--r--r-- 1 root root   0 Mar 16 22:15 42d0ab39867a
-r--r--r-- 1 root root   0 Mar 17 09:51 a2d2e95c8b31
```

2. 容器 net ns 进入

我们首先启动一个容器

```
Shell
root@139-159-150-152:/var/run/docker/netns# docker run -itd --name
net4 -p 81:80 busybox
36dd63a05de607d2a4d271060d01dd96ea53b49162f825c7625e268461e98e0e
```

查看容器的详细信息，可以看到 SandboxKey，该目录就是我们的容器所在的网络命名空间

```
Shell
root@139-159-150-152:/var/run/docker/netns# docker inspect net4
|grep Sand -A 10
            "SandboxID":
"00899eb26020c8222cb78b60b8bcf053d1086e28c9e26c625a2b59f352ef6231"
,
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {
                "80/tcp": [
                    {
                        "HostIp": "0.0.0.0",
                        "HostPort": "81"
                    },
                    {
--
            "SandboxKey": "/var/run/docker/netns/00899eb26020",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID":
"d8bdbef789874cc052c0df9ea3390cd6120f74ea9c39d783dc877588baf6c398"
,
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.4",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:04",
```

3. 通过 nsenter 进入命名空间，查看 ip 地址

```
Shell
root@139-159-150-152:/var/run/docker/netns# nsenter --
net=/var/run/docker/netns/00899eb26020 sh
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
559: eth0@if560: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
```

```
    link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff link-
netnsid 0
    inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

4. 通过 shell 进入容器，可以看到两个打印的 ip 地址是一样的，也就是说我们现在位于容器的命名空间了。

```
Shell
root@139-159-150-152:/var/run/docker/netns# docker exec -it net4
sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
559: eth0@if560: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500
qdisc noqueue
    link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ #
```