# 3-1、Docker镜像的原理讲解

## 1、docker镜像是什么？

- 是一种轻量级，可执行的独立软件包。包含了运行某个软件所需的所有内容，包括代码，运行时库，环境变量，配置文件。
- 所有的应用直接打包、构建成docker镜像、就可以直接运行使用。

```
[root@kuangshen ~]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
54fec2fa59d0: Already exists
9c94e11103d9: Pull complete
04ab1bfc453f: Pull complete
a22fde870392: Pull complete
def16cac9f02: Pull complete
1604f5999542: Pull complete
Digest: sha256:584db7c588cd6d1e0a97bcf67c49016d4f19320b614e07049404bea1d681965e
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
[root@kuangshen ~]#
```

### 2、怎么获取镜像？

- 可以从远程镜像仓库下载
- 朋友那拷贝
- 自己通过Dockerfile文件构建一个镜像。

### 3、docker镜像加载原理

- bootfs中的kernel内核被bootloader引导器引导启动后，内存使用权由bootfs交给kernel内核，会把bootfs卸载掉。
- docker镜像实际是一层一层文件系统组成的，rootfs就是在bootfs之上的一层，可以是Ubuntu、centos、Debian、fedora、redhat等文件系统，每个文件系统都是典型的linux系统的不同的发行版，包含了/dev，/proc，/bin，/etc等标准目录和文件，只是有些许轻微的差别。
- 为何Docker中centos镜像那么小，才200多M?
  - 因为对于一个精简的OS操作系统（这里可以是centos、ubuntu、debian、fedora、redhat），rootfs可以很小，只需要包含最基本的命令、工具、程序包就可以了，因为最底层都是直接使用bootfs的kernel内核，自己只需要提供对应所需的rootfs就可以了，由此可见对于不同的OS操作系统，它们的bootfs是公用的，只是使用的rootfs有所差别。
- 所以虚拟机启动就是引导linux内核的过程，分钟级别，比较慢；而容器的启动是秒级的，很快；

## UnionFS（联合文件系统）

我们下载的时候看到的一层层就是这个！

UnionFS（联合文件系统）：Union文件系统（UnionFS）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(unite several directories into a single virtual filesystem)。Union 文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。

特性：一次同时加载多个文件系统，但从外面看起来，只能看到一个文件系统，联合加载会把各层文件系统叠加起来，这样最终的文件系统会包含所有底层的文件和目录

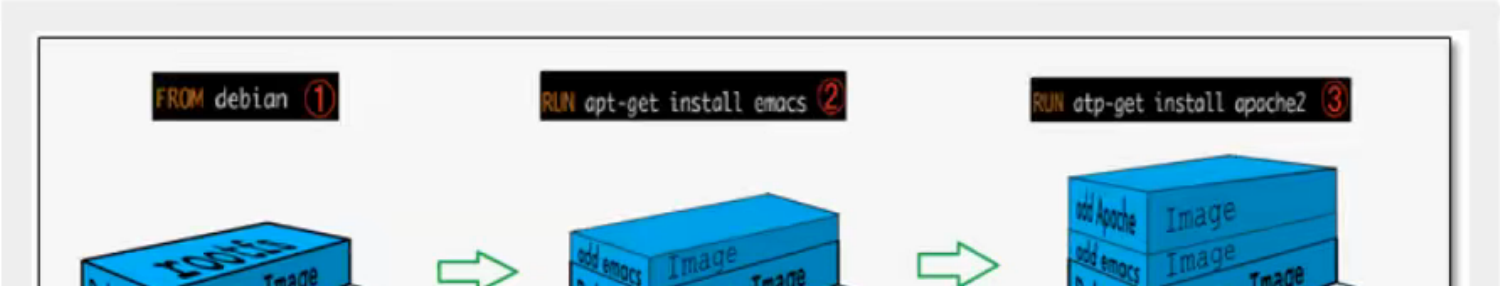
## Docker镜像加载原理

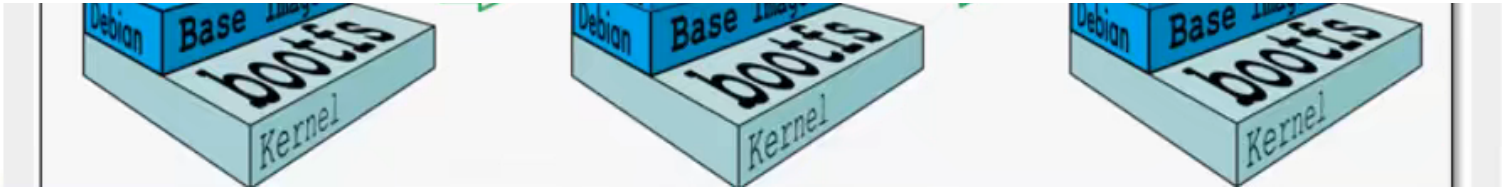
docker的镜像实际上由一层一层的文件系统组成，这种层级的文件系统UnionFS。

bootfs(boot file system)主要包含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。

黑屏-加载--开机进入系统

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。






**4、docker镜像分层原理**

- 已经拉取过的镜像层（redis的kernel内核层被拉取过，因为内核层是共用的），无需再次拉取

```
[root@kuangshen ~]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
54fec2fa59d0: Already exists
9c94e11103d9: Pull complete
04ab1bfc453f: Pull complete
a22fde870392: Pull complete
def16cac9f02: Pull complete
1604f5999542: Pull complete
Digest: sha256:584db7c588cd6d1e0a97bcf67c49016d4f19320b614e07049404bea1d681965e
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
[root@kuangshen ~]#
```

- 查看下构建整个redis镜像的每一层（Layer）镜像的信息。

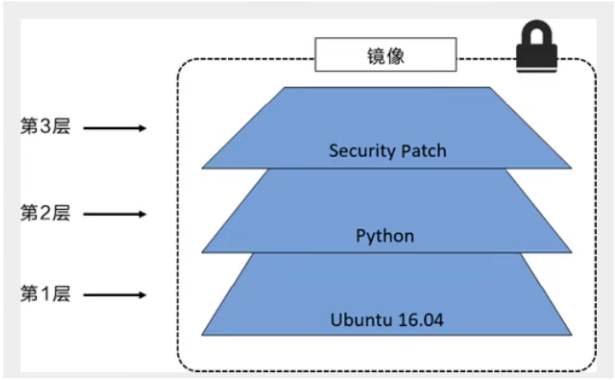
```
[root@kuangshen ~]# docker image inspect redis:latest
"RootFS": {
    rootfs参数，可以看到构建redis镜像的每一层
    "Layers": [
        "sha256:c2adabaecedbda0af72b153c6499a0555f3a769d52370469d8f6bd6328af9b13
        "sha256:744315296a49be711c312dfa1b3a80516116f78c437367ff0bc678da1123e990
        "sha256:379ef5d5cb402a5538413d7285b21aa58a560882d15f1f553f7868dc4b66afa8
        "sha256:d00fd460effb7b066760f97447c071492d471c5176d05b8af1751806a1f905f8
        "sha256:4d0c196331523cfed7bf5bafd616ecb3855256838d850b6f3d5fba911f6c4123
        "sha256:98b4a6242af2536383425ba2d6de033a510e049d9ca07ff501b95052da76e894
    ]
},
```

- 每一层，都类似于git的某一个提交版本，做的是功能累加操作（文件增加，删除，更新）。

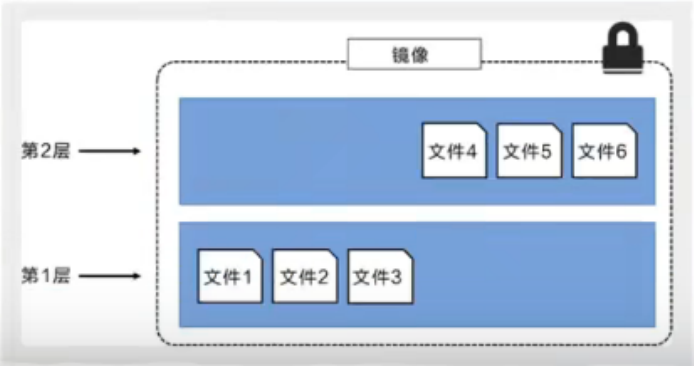
所有的 Docker 镜像都起始于一个基础镜像层，当进行修改或增加新的内容时，就会在当前镜像层之上，创建新的镜像层。

举一个简单的例子，假如基于 Ubuntu Linux 16.04 创建一个新的镜像，这就是新镜像的第一层；如果在该镜像中添加 Python包，就会在基础镜像层之上创建第二个镜像层；如果继续添加一个安全补丁，就会创建第三个镜像层。

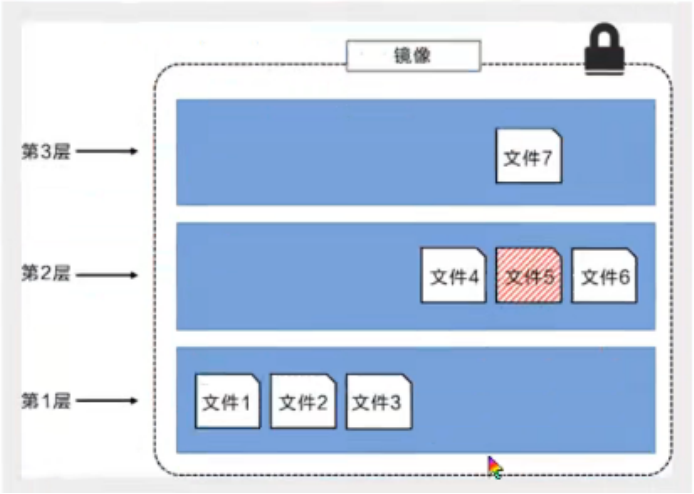
该镜像当前已经包含 3 个镜像层，如下图所示（这只是一个用于演示的很简单的例子）。




在添加额外的镜像层的同时，镜像始终保持是当前所有镜像的组合，理解这一点非常重要。下图中举了一个简单的例子，每个镜像层包含 3 个文件，而镜像包含了来自两个镜像层的 6 个文件。



下图中展示了一个稍微复杂的三层镜像，在外部看来整个镜像只有 6 个文件，这是因为最上层中的文件 7 是文件 5 的一个更新版本。



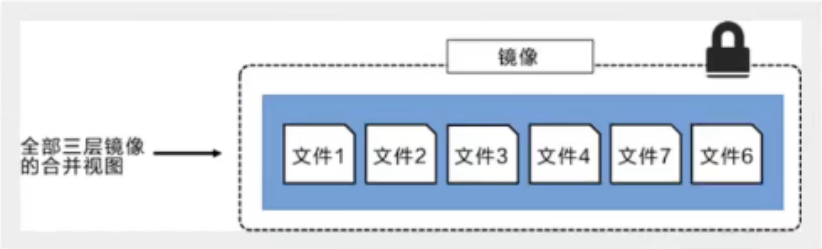

这种情况下，上层镜像层中的文件覆盖了底层镜像层中的文件。这样就使得文件的更新版本作为一个新镜像层添加到镜像当中。

Docker 通过存储引擎（新版本采用快照机制）的方式来实现镜像层堆栈，并保证多镜像层对外展示为统一的文件系统。

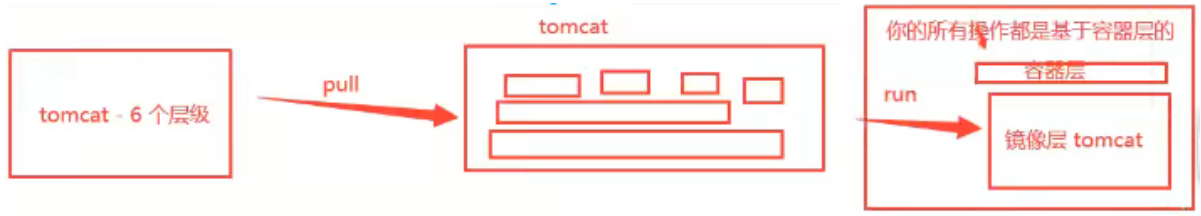
Linux 上可用的存储引擎有 AUFS、Overlay2、Device Mapper、Btrfs 以及 ZFS。顾名思义，每种存储引擎都基于 Linux 中对应的文件系统或者块设备技术，并且每种存储引擎都有其独有的性能特点。

Docker 在 Windows 上仅支持 windowsfilter 一种存储引擎，该引擎基于 NTFS 文件系统之上实现了分层和 CoW[1]。

下图展示了与系统显示相同的三层镜像。所有镜像层堆叠并合并，对外提供统一的视图。




- docker镜像都是只读的，一个docker镜像由多个可读的镜像层组成，然后通过镜像运行出的容器会在这个docker镜像顶部多加一层可写的容器层，任何的对文件的更改都只存在此容器层，因此任何对容器的操作均不会影响到镜像。
- 如果别人要使用你的新镜像，则你将运行出的容器，更改后，commit即可.（相当于git提交了一个版本），新镜像是包含了修改后的容器层+原有的所有镜像层。




- 镜像分层的好处

> ①基本上每个软件都是基于某个镜像去运行的，因此一旦某个底层环境出了问题，就不需要去修改全部基于该镜像的软件的镜像，只需要修改底层环境的镜像。
>
> ②这个好处也是最大好处，就是可以共享资源，其他相同环境的软件镜像都共同去享用同一个环境镜像，而不需要每个软件镜像要去创建一个底层环境。

**5、docker的commit，基于原有的镜像，生成一个新镜像**

- 思考？镜像是可读的，怎么会生成一个新镜像？
  - 是通过镜像运行出一个容器，进入容器内部，修改内容，将容器进行commit，得到一个新镜像（修改后的容器+原有所有镜像层），类似于git commit一个新提交，但是整个项目包含了新提交+原有所有提交。
- 语法

> 提交容器成为一个新的副本
> docker commit -m "提交的描述信息" -a "作者" 容器id 目标镜像名:[tag]

- 测试
  - 启动一个默认的tomcat容器，发现默认的tomcat容器的webapps下是没有文件的（没有tomcat欢迎页），因为官方的镜像默认webapps是没有文件的，我自己拷贝进去了一些文件进去（tomcat欢迎页），这一步拷贝操作就是对容器的一个修改。

```
[root@kuangshen ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
MES
7e119b82cff6        tomcat              "catalina.sh run"   16 seconds ago      Up 15 seconds       0.0.0.0:8080->8080/tcp
arming_borg
[root@kuangshen ~]# docker exec -it 7e119b82cff6 /bin/bash
root@7e119b82cff6:/usr/local/tomcat#
```

```
root@7e119b82cff6:/usr/local/tomcat# ls
BUILDING.txt     LICENSE  README.md      RUNNING.txt  conf  logs            temp     webapps.dist
CONTRIBUTING.md  NOTICE   RELEASE-NOTES  bin          lib   native-jni-lib  webapps  work
root@7e119b82cff6:/usr/local/tomcat# cp -r webapps.dist/* webapps
root@7e119b82cff6:/usr/local/tomcat#

root@7e119b82cff6:/usr/local/tomcat# cd webapps
root@7e119b82cff6:/usr/local/tomcat/webapps# ls
ROOT  docs  examples  host-manager  manager
root@7e119b82cff6:/usr/local/tomcat/webapps#
```

因为webapps目录下没文件，拷贝一些

- 然后将我们修改过的容器，commit提交为一个新镜像（包含了修改后的容器层+原有的所有镜像层）。发现新镜像tomcat02比官方镜像大了一点。

```
[root@kuangshen ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NA
MES
7e119b82cff6        tomcat              "catalina.sh run"   3 minutes ago       Up 3 minutes        0.0.0.0:8080->8080/tcp   ch
arming_borg
[root@kuangshen ~]# docker commit -a="kuangshen" -m="add webapps app" 7e119b82cff6 tomcat02:1.0
sha256:f7eb5017e655150f94f787da9cb12e4399ade8d98c435b3478997ef2a82fa6fe
[root@kuangshen ~]# docker images
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
tomcat02              1.0                 f7eb5017e655        5 seconds ago       652MB
tomcat                9.0                 d03312117bb0        37 hours ago        647MB
tomcat                latest              d03312117bb0        37 hours ago        647MB
redis                 latest              f9b990972689        12 days ago         104MB
nginx                 latest              602e111c06b6        3 weeks ago         127MB
elasticsearch         7.6.2               f29a1ee41030        7 weeks ago         791MB
portainer/portainer   latest              2869fc110bf7        7 weeks ago         78.6MB
centos                latest              470671670cac        3 months ago        237MB
[root@kuangshen ~]#
```

- 运行新镜像为一个容器，浏览器访问8080就可以到欢迎页