Docker常用命令

帮助命令

```
docker version       #显示docker的版本信息
docker info          #显示docker的系统信息，包括镜像和容器的数量
docker 命令 --help    #帮助命令
```

帮助文档的地址：https://docs.docker.com/reference/

镜像命令

docker images 查看所有本地的主机上的镜像

```
root@tcheng:/home/tcheng# docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-world   latest    d1165f221234   4 months ago    13.3kB

#解释
REPOSITORY        镜像的仓库源
TAG               镜像的标签
IMAGE ID          镜像的id
CREATED           镜像的创建时间
SIZE              镜像的大小

#可选项
  -a, --all               #列出所有的镜像
  -q, --quiet             #只显示镜像的id
```

docker search 搜索镜像

```
root@tcheng:/home/tcheng# docker search mysql
NAME                                 DESCRIPTION
STARS     OFFICIAL   AUTOMATED
mysql                                MySQL is a widely used, open-source relation…
11147     [OK]
mariadb                              MariaDB Server is a high performing open sou…
4228      [OK]
mysql/mysql-server                   Optimized MySQL Server Docker images. Create…
829                  [OK]

#可选项，通过搜索来过滤
--filter=STARS=3000      #搜索出来的镜像就是STARS大于3000的
root@tcheng:/home/tcheng# docker search mysql --filter=STARS=3000
NAME      DESCRIPTION                                     STARS     OFFICIAL
AUTOMATED
mysql     MySQL is a widely used, open-source relation…   11147     [OK]
mariadb   MariaDB Server is a high performing open sou…   4228      [OK]
root@tcheng:/home/tcheng# docker search mysql --filter=STARS=5000
```

```
NAME       DESCRIPTION                                      STARS     OFFICIAL
AUTOMATED
mysql      MySQL is a widely used, open-source relation…   11147     [OK]
```

docker pull 下载镜像

```
#下载镜像 docker pull 镜像名[:tag]
root@tcheng:/home/tcheng# docker pull mysql
Using default tag: latest       #如果不写tag，默认就是latest
latest: Pulling from library/mysql
b4d181a07f80: Pull complete     #分层下载，docker image的核心，联合文件系统
a462b60610f5: Pull complete
578fafb77ab8: Pull complete
524046006037: Pull complete
d0cbe54c8855: Pull complete
aa18e05cc46d: Pull complete
fd6f649b1d0a: Pull complete
2a97d48c2fdc: Pull complete
30f0c7db48fc: Pull complete
f5dda8df049e: Pull complete
671b83fd7448: Pull complete
5d9cc55fa997: Pull complete
Digest: sha256:18d8d109aa64673c78aebfb845b929cfdac97a553332f4310f4de8d67ceb03d2
#签名
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest  #真实地址

#等价于它
docker pull mysql
docker pull docker.io/library/mysql:latest

#指定版本下载
root@tcheng:/home/tcheng# docker pull mysql:5.7
5.7: Pulling from library/mysql
b4d181a07f80: Already exists
a462b60610f5: Already exists
578fafb77ab8: Already exists
524046006037: Already exists
d0cbe54c8855: Already exists
aa18e05cc46d: Already exists
fd6f649b1d0a: Already exists
8a2b858b000b: Pull complete
322182b17422: Pull complete
070e28050a88: Pull complete
613bdfd8796e: Pull complete
Digest: sha256:956e11ac581cad9ac8747a9a1d61b8ffcfa6845e0f23bdbab6ba20a2ad792cbf
Status: Downloaded newer image for mysql:5.7
docker.io/library/mysql:5.7
```

```
root@tcheng:/home/tcheng# docker images
REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
mysql          5.7       9f1d21c1025a   19 hours ago     448MB
mysql          latest    95db2e2bd882   19 hours ago     514MB
hello-world    latest    d1165f221234   4 months ago     13.3kB
root@tcheng:/home/tcheng#
```

docker rmi 删除镜像

```
root@tcheng:/home/tcheng# docker rmi -f 容器id                        #删除指定id的容器
root@tcheng:/home/tcheng# docker rmi -f 容器id 容器id 容器id 容器id      #删除多个容器
root@tcheng:/home/tcheng# docker rmi -f $(docker images -aq)          #删除所有容器
```

容器命令

说明：我们有了镜像才可以创建容器，linux，下载一个centos镜像来测试学习

```
docker pull centos
```

新建容器并启动

```
docker run [可选参数] image

#参数说明
--name Name     容器名字  tomcat01  tomcat02，用来区分容器
-d                  后台方式运行
-it                 使用交互方式运行，进入容器查看内容
-p                  指定容器的端口 -p 8080:8080
    -p ip:主机端口:容器端口
    -p 主机端口:容器端口     (常用)
    -p 容器端口
    容器端口
-P                  随机指定端口

#测试，启动并进入容器
root@tcheng:/home/tcheng# docker run -it centos /bin/bash
[root@2119931c0839 /]# ls   #查看容器内的centos，基础版本，很多命令都是不完善的
bin  etc   lib    lost+found  mnt  proc  run   srv  tmp  var
dev  home  lib64  media       opt  root  sbin  sys  usr

#从容器中退回主机
[root@2119931c0839 /]# exit
exit
root@tcheng:/home/tcheng# ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

列出运行的容器

```
# docker ps 命令
        #列出当前正在运行的容器
-a      #列出当前正在运行的容器+带出历史运行过的容器
-n=?    #显示最近创建的容器，比如 -n=2 意思就是列出最近创建的2个容器
-q      #只显示容器的编号

root@tcheng:/home/tcheng# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
root@tcheng:/home/tcheng# docker ps -a
CONTAINER ID   IMAGE          COMMAND       CREATED          STATUS
      PORTS     NAMES
2119931c0839   centos         "/bin/bash"   8 minutes ago    Exited (0) 5 minutes
ago             crazy_grothendieck
26d6b3db1bab   d1165f221234   "/hello"      17 hours ago     Exited (0) 17 hours
ago              blissful_diffie
```

退出容器

```
exit            #直接容器停止并退出
Ctrl + P + Q    #容器不停止退出
```

删除容器

```
docker rm 容器id                #删除指定的容器，不能删除正在运行的容器，如果要强制删除
rm -f
docker rm -f $(docker ps -aq)   #删除所有的容器
docker ps -a -q|xargs docker rm #删除所有的容器
```

启动和停止容器的操作

```
docker start 容器id   # 启动容器
docker restart 容器id # 重启容器
docker stop 容器id    # 停止当前正在运行的容器
docker kill 容器id    # 强制停止当前的容器
```

常用其他命令

后台启动容器

```
#命令 docker run -d 镜像名!
root@tcheng:/home/tcheng# docker run -d centos

#问题 运行docker ps时，发现 centos 停止了

#常见的坑，docker 容器使用后台运行，就必须要有一个前台进程，不然docker发现没有应用，就会自动停
止
```

查看日志

```
#命令 docker logs -tf --tail number 容器id

# 自己编写一段shell脚本
root@tcheng:/home/tcheng# docker run -d centos /bin/sh -c "while true; do echo
chenghu; sleep 1; done"
```

```
e841c6f318aed599135e9d54c9dfdacca3b7beeefd2d1f24afc5a1d505e525eb

root@tcheng:/home/tcheng# docker ps
CONTAINER ID   IMAGE
e841c6f318ae   centos

#显示日志
-tf              # 显示日志
--tail number    # 要显示的日志条数
root@tcheng:/home/tcheng# docker logs -tf --tail 10 e841c6f318ae
```

查看容器中进程信息

```
#命令 docker top 容器id
root@tcheng:/home/tcheng# docker top e841c6f318ae
UID                 PID                 PPID                C
STIME
root                2831                2807                0
10:18
root                3311                2831                0
10:24
```

查看镜像的元数据

```
#命令 docker inspect 容器id

#测试
root@tcheng:/home/tcheng# docker inspect e841c6f318ae
[
    {
        "Id":
"e841c6f318aed599135e9d54c9dfdacca3b7beeefd2d1f24afc5a1d505e525eb",
        "Created": "2021-07-21T02:18:07.074098349Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "while true; do echo chenghu; sleep 1; done"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 2831,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2021-07-21T02:18:07.406954499Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image":
"sha256:300e315adb2f96afe5f0b2780b87f28ae95231fe3bdd1e16b9ba606307728f55",
        "ResolvConfPath":
"/var/lib/docker/containers/e841c6f318aed599135e9d54c9dfdacca3b7beeefd2d1f24afc5
a1d505e525eb/resolv.conf",
```

```
        "HostnamePath":
"/var/lib/docker/containers/e841c6f318aed599135e9d54c9dfdacca3b7beeefd2d1f24afc5
a1d505e525eb/hostname",
        "HostsPath":
"/var/lib/docker/containers/e841c6f318aed599135e9d54c9dfdacca3b7beeefd2d1f24afc5
a1d505e525eb/hosts",
        "LogPath":
"/var/lib/docker/containers/e841c6f318aed599135e9d54c9dfdacca3b7beeefd2d1f24afc5
a1d505e525eb/e841c6f318aed599135e9d54c9dfdacca3b7beeefd2d1f24afc5a1d505e525eb-
json.log",
        "Name": "/stoic_gagarin",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "docker-default",
        "ExecIDs": null,
        "HostConfig": {
            "Binds": null,
            "ContainerIDFile": "",
            "LogConfig": {
                "Type": "json-file",
                "Config": {}
            },
            "NetworkMode": "default",
            "PortBindings": {},
            "RestartPolicy": {
                "Name": "no",
                "MaximumRetryCount": 0
            },
            "AutoRemove": false,
            "VolumeDriver": "",
            "VolumesFrom": null,
            "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",
            "ConsoleSize": [
                0,
                0
```

```
            ],
            "Isolation": "",
            "CpuShares": 0,
            "Memory": 0,
            "NanoCpus": 0,
            "CgroupParent": "",
            "BlkioWeight": 0,
            "BlkioWeightDevice": [],
            "BlkioDeviceReadBps": null,
            "BlkioDeviceWriteBps": null,
            "BlkioDeviceReadIOps": null,
            "BlkioDeviceWriteIOps": null,
            "CpuPeriod": 0,
            "CpuQuota": 0,
            "CpuRealtimePeriod": 0,
            "CpuRealtimeRuntime": 0,
            "CpusetCpus": "",
            "CpusetMems": "",
            "Devices": [],
            "DeviceCgroupRules": null,
            "DeviceRequests": null,
            "KernelMemory": 0,
            "KernelMemoryTCP": 0,
            "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":
"/var/lib/docker/overlay2/0b9426cf5a48bf51db0d410fae1de848f739906d8e05316707d142
98eb3f780c-
init/diff:/var/lib/docker/overlay2/baa640a0531943d59945d9837e92c00c847588e0cc3f1
2dafd49c8a8c9353e71/diff",
                "MergedDir":
"/var/lib/docker/overlay2/0b9426cf5a48bf51db0d410fae1de848f739906d8e05316707d142
98eb3f780c/merged",
                "UpperDir":
"/var/lib/docker/overlay2/0b9426cf5a48bf51db0d410fae1de848f739906d8e05316707d142
98eb3f780c/diff",
                "WorkDir":
"/var/lib/docker/overlay2/0b9426cf5a48bf51db0d410fae1de848f739906d8e05316707d142
98eb3f780c/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "e841c6f318ae",
            "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/sh",
                "-c",
                "while true; do echo chenghu; sleep 1; done"
            ],
            "Image": "centos",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "org.label-schema.build-date": "20201204",
                "org.label-schema.license": "GPLv2",
                "org.label-schema.name": "CentOS Base Image",
                "org.label-schema.schema-version": "1.0",
                "org.label-schema.vendor": "CentOS"
            }
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID":
"3df2b3435462b14907edc2d92eb22f5dd88b065505bbe66d15092e8c455cf590",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {},
```

```
            "SandboxKey": "/var/run/docker/netns/3df2b3435462",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID":
"30b4ddba5b21650422f0c3f32674131e8301250e2fca63d7eccbab9441f88d2a",
            "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":
"bc27bef241ae9b458260708f222617c4049a26a2f0700dc2bb44a9cca1a39dd8",
                    "EndpointID":
"30b4ddba5b21650422f0c3f32674131e8301250e2fca63d7eccbab9441f88d2a",
                    "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
                }
            }
        }
    }
]
```

进入当前正在运行的容器

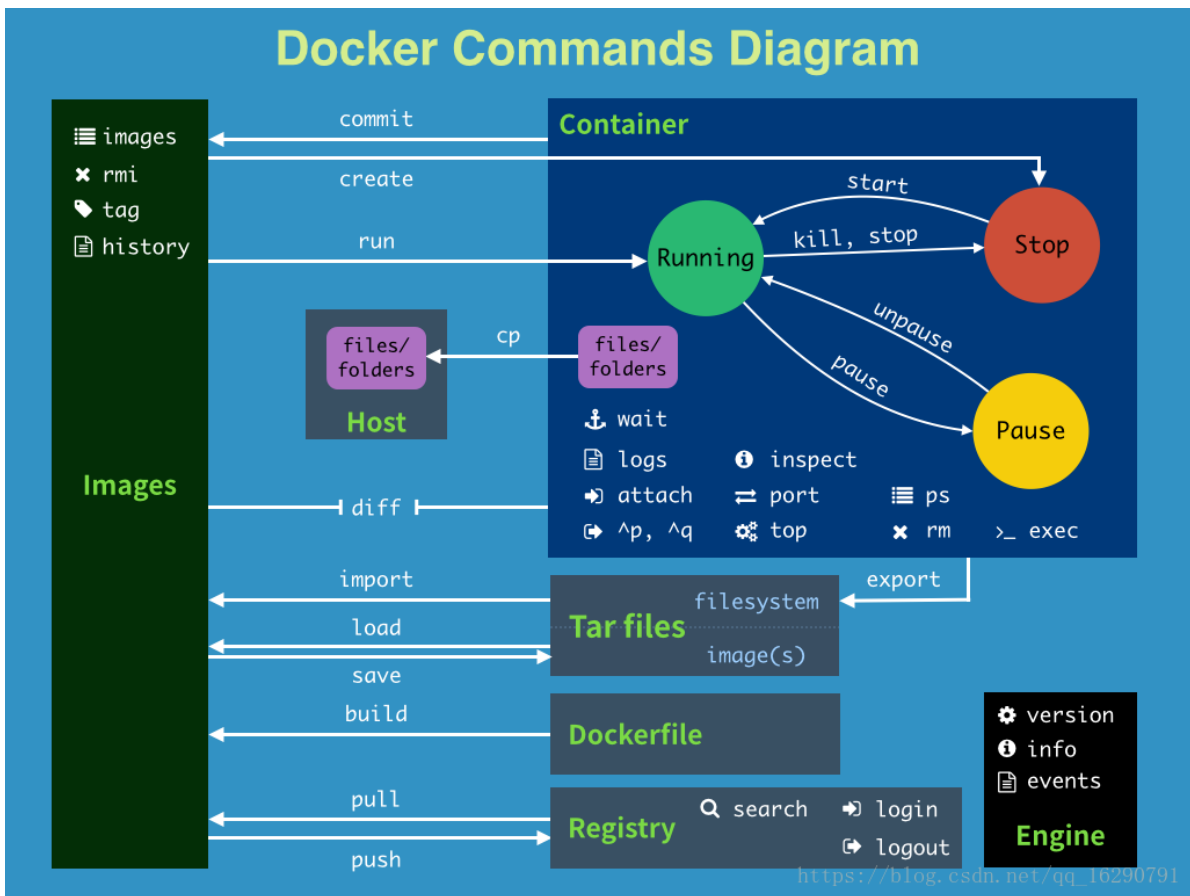
```
# 我们通常容器都是使用后台方式运行的，需要进入容器，修改一些配置

#命令
docker exec -it 容器id bashShell

#测试
root@tcheng:/home/tcheng# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
PORTS     NAMES
e841c6f318ae   centos    "/bin/sh -c 'while t…"   28 minutes ago   Up 28 minutes
            stoic_gagarin
root@tcheng:/home/tcheng# docker exec -it e841c6f318ae /bin/bash
[root@e841c6f318ae /]# ls
bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run
sbin  srv  sys  tmp  usr  var
[root@e841c6f318ae /]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 02:18 ?        00:00:01 /bin/sh -c while true; do
echo chenghu; sleep 1; done
root        1714       0  0 02:46 pts/0    00:00:00 /bin/bash
```

```
root         1749       1  0 02:47 ?        00:00:00 /usr/bin/coreutils --
coreutils-prog-shebang=sleep /usr/bin/sleep 1
root         1750    1714  0 02:47 pts/0    00:00:00 ps -ef

#方式二
docker attach 容器id

#测试
root@tcheng:/home/tcheng# docker attach e841c6f318ae
正在执行当前的代码...

#两个命令的区别
docker exec     #进入容器后开启一个新的终端，可以在里面操作（常用）
docker attach   #进入容器正在执行的终端，不会启动新的进程
```

从容器内拷贝文件到主机上

```
#命令
docker cp 容器id:容器内路径 目的主机路径

#查看当前主机目录下
root@tcheng:/home# ls
tcheng
root@tcheng:/home# docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS
NAMES
0c504c42e552   centos    "/bin/bash"   10 minutes ago   Up 2 minutes
goofy_saha

#进入docker容器内部
root@tcheng:/home# docker attach 0c504c42e552
[root@0c504c42e552 home]# ls

#在容器内新建一个文件
[root@0c504c42e552 home]# touch test.py
[root@0c504c42e552 home]# exit
exit
root@tcheng:/home# docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS
  PORTS     NAMES
0c504c42e552   centos    "/bin/bash"   11 minutes ago   Exited (0) 6 seconds ago
            goofy_saha

#将这个文件拷贝到主机上
root@tcheng:/home# docker cp 0c504c42e552:/home/test.py /home
root@tcheng:/home# ls
tcheng  test.py
root@tcheng:/home#
```

小结




```
attach      Attach local standard input, output, and error streams to a running
container   #当前 shell 下 attach 连接指定运行镜像
build       Build an image from a Dockerfile                #通过 Dockerfile 定制
镜像
commit      Create a new image from a container's changes   #提交当前容器为新的镜像
cp          Copy files/folders between a container and the local filesystem #从容
器中拷贝指定文件或目录到主机中
create      Create a new container                          #创建一个新的容器，同
run，但不启动容器
diff        Inspect changes to files or directories on a container's filesystem
#查看docker容器变化
events      Get real time events from the server    #从 docker 服务获取容器实时事件
exec        Run a command in a running container    #在已存在的容器中运行命令
export      Export a container's filesystem as a tar archive    #导出容器的内容作为
一个 tar [对应import]
history     Show the history of an image        #展示一个镜像形成历史
images      List images                         #列出系统当前镜像
import      Import the contents from a tarball to create a filesystem image #从
tar包中的内容创建一个新的文件系统镜像[对应export]
info        Display system-wide information #显示系统相关信息
inspect     Return low-level information on Docker objects      #查看容器详细信息
kill        Kill one or more running containers                 # kill 指定
docker 容器
load        Load an image from a tar archive or STDIN           #从一个tar包中加载
一个镜像[对应save]
login       Log in to a Docker registry                         #注册或登录一个
docker 源服务器
logout      Log out from a Docker registry                      #从当前 Docker
registry 退出
logs        Fetch the logs of a container                       #输出当前容器日志信
息
pause       Pause all processes within one or more containers   #暂停一个或多个容器
```

```
port        List port mappings or a specific mapping for the container  #列出端口映射或容器的特定映射
ps          List containers                                    #列出容器列表
pull        Pull an image or a repository from a registry      #从docker镜像源服务器拉取指定镜像或者镜像库
push        Push an image or a repository to a registry        #推送指定镜像或者镜像库至docker源服务器
rename      Rename a container                        #重命名容器
restart     Restart one or more containers            #重启一个或多个容器
rm          Remove one or more containers             #删除一个或多个容器
rmi         Remove one or more images                 #删除一个或多个镜像[无容器使用该镜像才可删除，否则需删除相关容器才可继续或使用 -f 选项强制删除]
run         Run a command in a new container                       #创建一个新的容器并运行命令行
save        Save one or more images to a tar archive (streamed to STDOUT by default)   #保存一个镜像为一个 tar 包[对应load]
search      Search the Docker Hub for images                       #在 Docker Hub 中搜索镜像
start       Start one or more stopped containers                   #启动容器
stats       Display a live stream of container(s) resource usage statistics #显示容器资源使用统计的实时流
stop        Stop one or more running containers                        #停止容器
tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE   #创建一个指向SOURCE_IMAGE 的标签 TARGET_IMAGE
top         Display the running processes of a container           #查看容器中运行的进程信息
unpause     Unpause all processes within one or more containers #取消暂停容器
update      Update configuration of one or more containers         #更新一个或多个容器的配置
version     Show the Docker version information                    #查看 docker 版本号
wait        Block until one or more containers stop, then print their exit codes #截取容器停止时的退出状态值
```

接下来就是一堆的练习

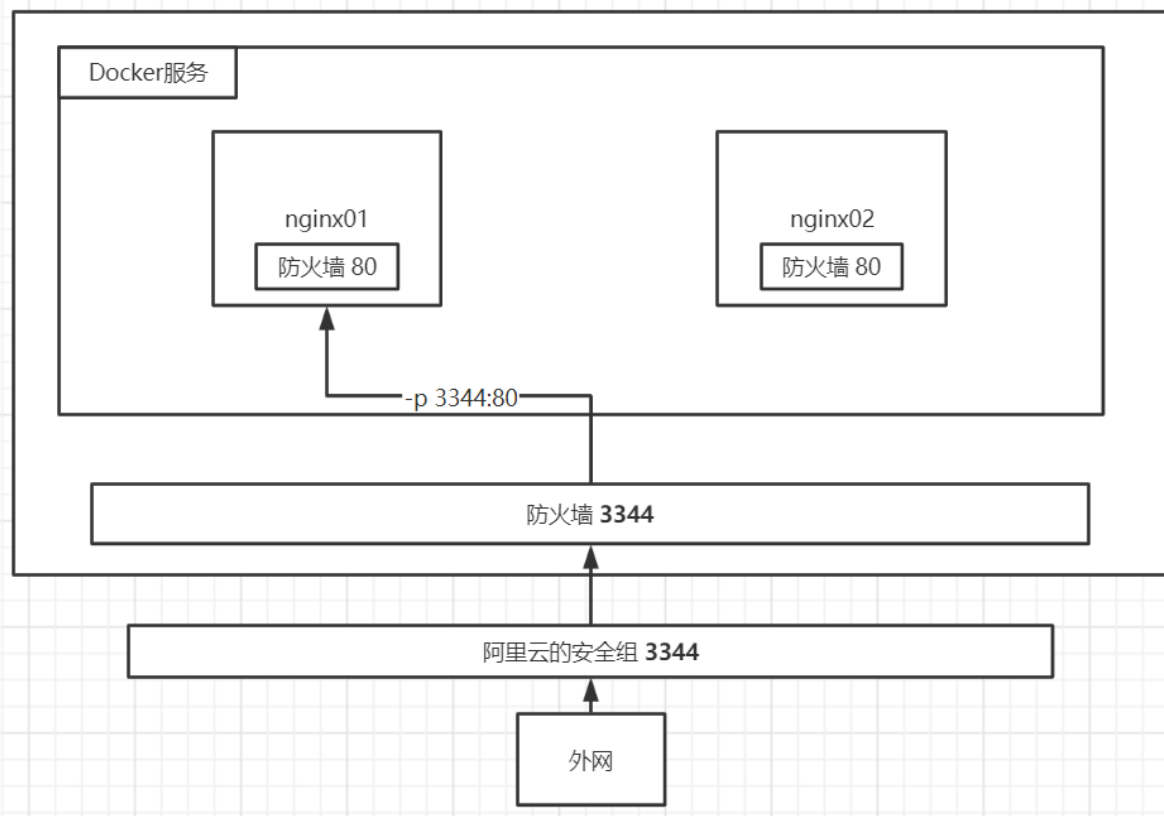
作业练习

> Docker安装Nginx

```
# 1、搜索镜像 search 建议去 Docker Hub 搜索，可以看到帮助文档
# 2、下载镜像 pull
# 3、运行测试

# -d 后台运行
# --name 给容器命名
# -p 宿主机端口:容器内部端口
root@tcheng:/home# docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
nginx         latest    4cdc5dd7eaad   2 weeks ago    133MB
hello-world   latest    d1165f221234   4 months ago   13.3kB
centos        latest    300e315adb2f   7 months ago   209MB
root@tcheng:/home# docker run -d --name nginx01 -p 3344:80 nginx
```

```
1a99374cef4dff3dc58a6c0d90d0501ccbe57fc208938fb01bfbbd71f029f0fd
root@tcheng:/home# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
PORTS                                   NAMES
1a99374cef4d   nginx     "/docker-entrypoint.…"   20 seconds ago   Up 20 seconds
  0.0.0.0:3344->80/tcp, :::3344->80/tcp   nginx01
root@tcheng:/home# curl localhost:3344
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
root@tcheng:/home#

#进入容器
root@tcheng:/home# docker exec -it nginx01 /bin/bash
root@1a99374cef4d:/# whereis nginx
nginx: /usr/sbin/nginx /usr/lib/nginx /etc/nginx /usr/share/nginx
root@1a99374cef4d:/# cd /etc/nginx/
root@1a99374cef4d:/etc/nginx# ls
conf.d  fastcgi_params  mime.types  modules  nginx.conf  scgi_params
uwsgi_params
root@1a99374cef4d:/etc/nginx#
```

暴露端口的概念



思考问题：我们每次改动nginx配置文件，都需要进入容器内部？十分麻烦，我要是在容器外部提供一个映射路径，达到在容器外修改文件，容器内部就可以自动同步自动修改？-v 数据卷！

Docker镜像讲解

镜像是什么

镜像是一种轻量级、可执行的独立软件包，用来打包软件运行环境和基于运行环境开发的软件，它包含运行某个软件所需的所有内容，包括代码、运行时库、环境变量和配置文件。

所有的应用，直接打包docker镜像，就可以直接跑起来！

如何得到镜像：

- 从远程仓库下载
- 通过拷贝其他人的
- 自己制作一个镜像 DockerFile

Docker镜像加载原理

> UnionFS（联合文件系统）

我们下载的时候看到的一层一层的就是这个！

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。镜像可以童年故宫分层来进行继承，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

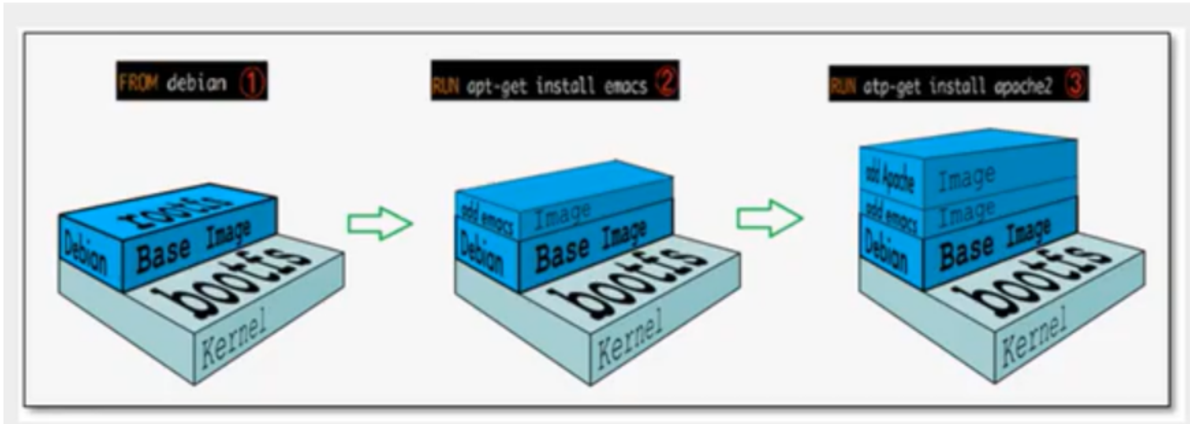
> Docker镜像加载原理

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就是各种不同操作系统的发行版，比如Ubuntu，Centos等等。



平时我们安装进虚拟机的CentOS都是好几个G，为什么Docker这里才200多M？

对于一个精简的OS，rootfs可以很小，只需要包含最基本的命令、工具和程序库就可以了，因为底层直接用Host的kernel，自己只需要提供rootfs就可以了，由此可见对于不同的linux发行版，bootfs基本都是一致的，rootfs会有差别，因此不同的发行版可以公用bootfs。

虚拟机是分钟级别的，容器是秒级的！

分层理解

> 分层的镜像

我们可以去下载一个镜像，注意观察下载的日志输出，可以看到是一层一层的在下载！

```
root@tcheng:/home# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
33847f680f63: Pull complete
26a746039521: Pull complete
18d87da94363: Pull complete
78e9d65cb9ae: Pull complete
985fcd1202ac: Pull complete
ffbec49e5b6a: Pull complete
Digest: sha256:7ca8673b08156d6253bb3a3262267d58b0ae7afc8985fd3ae0f91d5425ad8126
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
root@tcheng:/home# 
```

思考：为什么Docker镜像要采用这种分层的结构呢？

最大的好处，我觉得莫过于资源共享了！比如有多个镜像都从相同的Base镜像构建而来，那么宿主机只需要在磁盘上保留一份Base镜像，同时内存中也只需要加载一份Base镜像，这样就可以为所有的容器服务了，而且镜像的每一层都可以被共享。

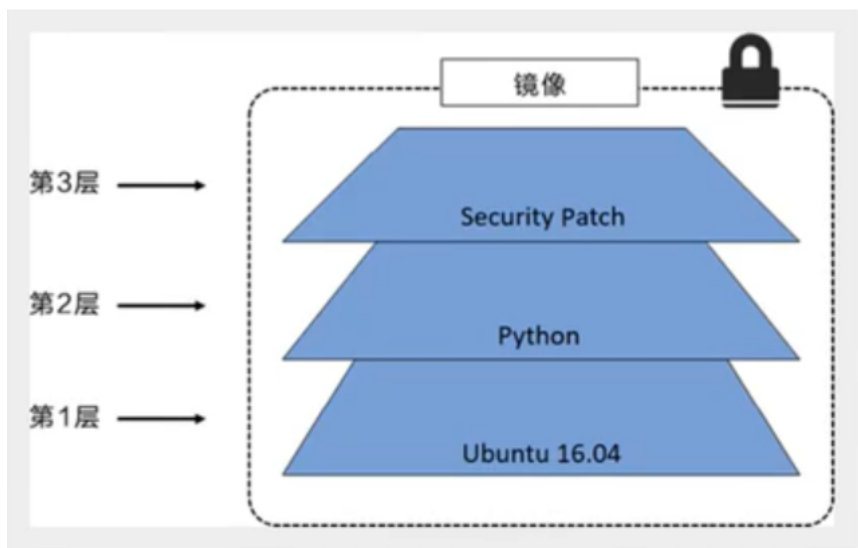
查看镜像分层的方式可以通过 docker image inspect 命令！

```
root@tcheng:/home# docker image inspect redis
[
    //..........
        "RootFS": {
            "Type": "layers",
            "Layers": [

 "sha256:814bff7343242acfd20a2c841e041dd57c50f0cf844d4abd2329f78b992197f4",

"sha256:dd1ebb1f5319785e34838c7332a71e5255bda9ccf61d2a0bf3bff3d2c3f4cdb4",

 "sha256:11f99184504048b93dc2bdabf1999d6bc7d9d9ded54d15a5f09e36d8c571c32d",

 "sha256:bbd0862909ab72bf6a2805b373cf20138b0ad2a41ed47c7355f06a7857baefd0",

 "sha256:5f29c448b19b8e6a8dac1c47f26f34a9ddb6ab1f7d4c5d0cf5488c382cd9e84e",

 "sha256:728ce664f6d39d6ad6a84fc51212779371cf570fb2b694ce1f20b68023809b87"
            ]
        },
        "Metadata": {
            "LastTagTime": "0001-01-01T00:00:00Z"
        }
    }
]
```

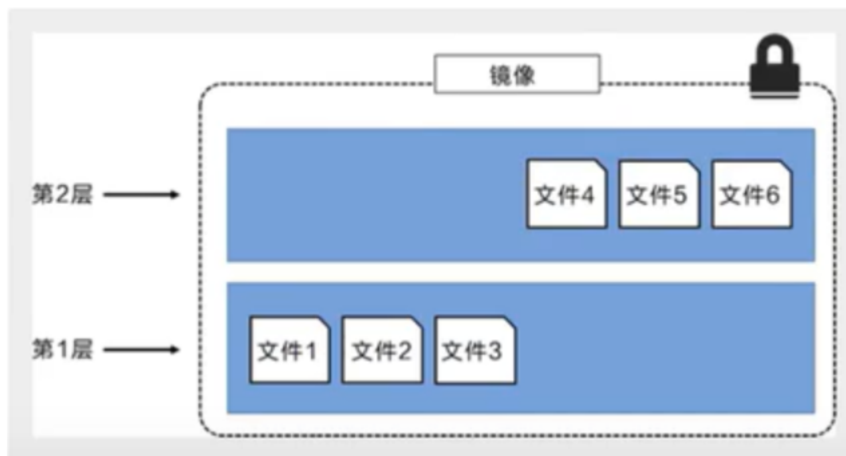
理解：

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，就会创建第三各镜像层。

该镜像当前已包含3个镜像层，如下图所示（这只是用于演示的很简单的例子）。

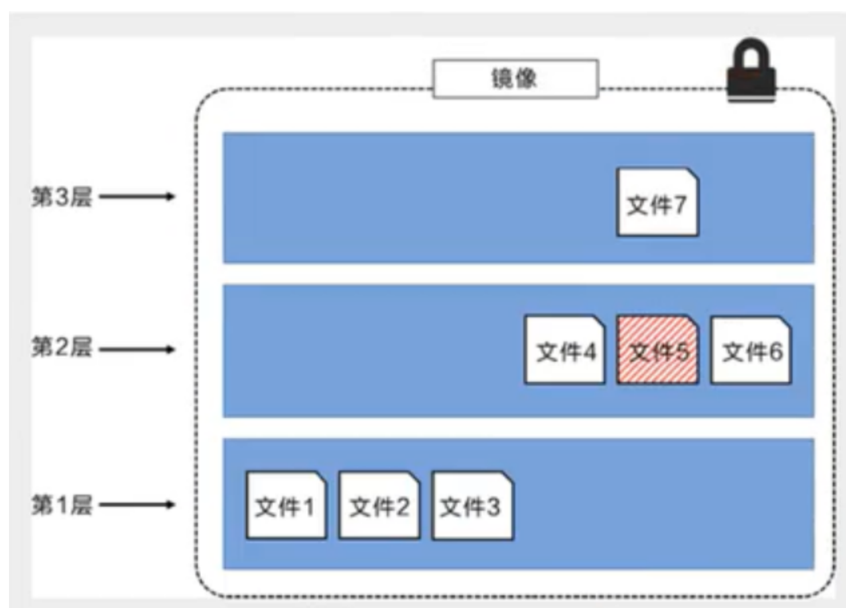


在添加额外镜像层的同时，镜像始终保持是当前所有镜像的组合，理解这一点非常重要。下图中举了一个简单的例子，每个镜像层包含 3 个文件，而镜像包含了来自两个镜像层的 6 个文件。



上图中的镜像层和之前图中的略有区别，主要目的是便于展示文件。

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，这是因为最上层的文件 7 是文件 5 的一个更新版本。



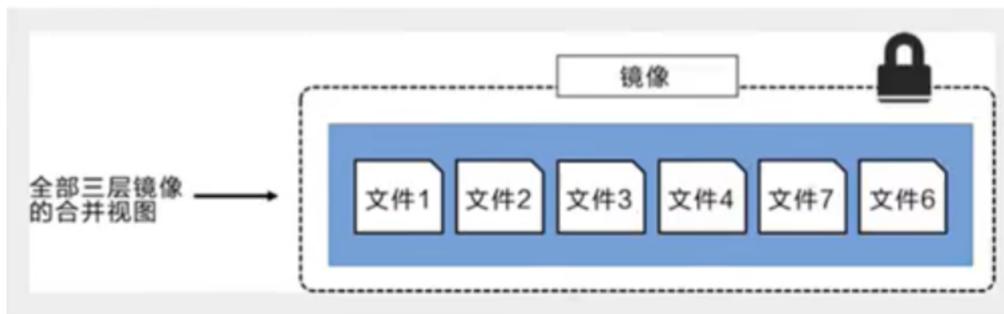
这种情况下，上曾镜像中的文件覆盖了底层镜像层中的文件。这样就使得文件的更新版本作为一个新镜像层添加到镜像当中。

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]。

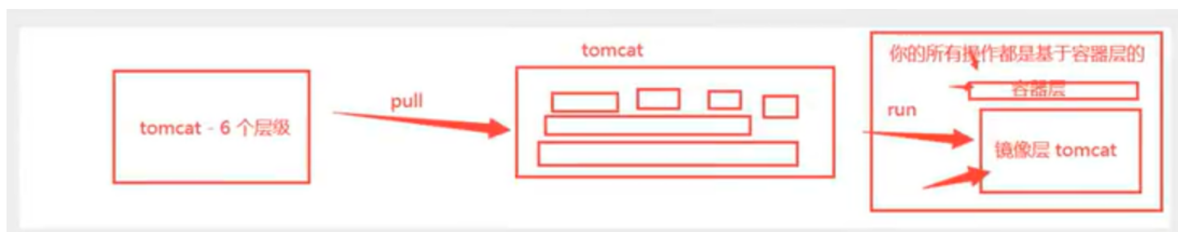
下图展示了与系统显示相同的三层镜像。所有镜像层堆叠并合并，对外提供统一的视图。



> 特点

Docker 镜像都是只读的，当容器启动时，一个新的可写层被加载到镜像的顶部！

这一层就是我们通常说的容器层，容器之下的都叫镜像层！



如何提交一个自己的镜像

commit 镜像

```
docker commit   #提交容器成为一个新的镜像

#命令和 git 原理类似
docker commit -m="提交的描述信息" -a="作者" 容器id 目标镜像名:[TAG]
```

实战测试

```
# 1、启动一个默认的tomcat

# 2、发现这个默认的tomcat 是没有webapps应用， 镜像的原因，官方的镜像默认 webapps 下面是没有文件的！

# 3、自己拷贝基本的文件到 webapps 目录下

# 4、将操作过的容器通过 commit 提交为一个新的镜像！以后就可以使用修改过的镜像即可。
```

```
root@tcheng:/home# docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS         PORTS                                       NAMES
1a83920f9666   tomcat    "/bin/bash"   4 minutes ago   Up 4 minutes   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   focused_shannon
root@tcheng:/home# docker commit -a="leaf" -m="add webapps application" 1a83920f9666 tomcat-leaf:1.0
sha256:91caf43cd7b6db7e4abb4251835757daa37efc3c10220af63cc6d14520a7dc58
root@tcheng:/home# docker images
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
tomcat-leaf   1.0       91caf43cd7b6   4 seconds ago    673MB
redis         latest    9dae5b22eb39   10 hours ago     105MB
tomcat        latest    921ef208ab56   30 hours ago     668MB
nginx         latest    4cdc5dd7eaad   2 weeks ago      133MB
hello-world   latest    d1165f221234   4 months ago     13.3kB
centos        latest    300e315adb2f   7 months ago     209MB
root@tcheng:/home#
```

到这里才算是入门Docker！

容器数据卷

什么是容器数据卷

docker理念回顾

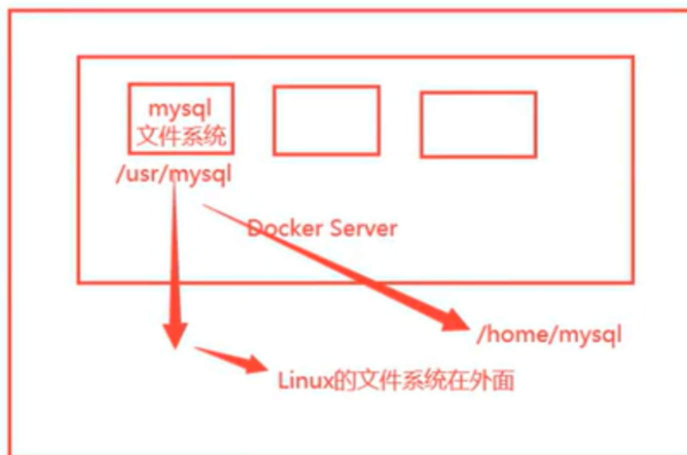
将应用和环境打包成一个镜像！

数据？如果数据都在容器中，那么我们把容器删除，就会导致数据丢失！需求：数据可以持久化！

MySQL，容器删了会导致数据都被删除了！需求：MySQL数据可以存储在本地！

容器之间可以有一个数据共享的技术！Docker 容器中产生的数据，同步到本地！

这就是卷技术！目录的挂载，将我们容器内的目录，挂载到Linux上面！



总结一句话：容器的持久化和同步操作！容器间也是可以数据共享的！

使用数据卷

> 方式一：直接使用命令来挂载 -v

```
#命令
docker run -it -v 宿主机目录:容器内目录

#测试
root@tcheng:/home# docker run -it -v /home/test:/home centos /bin/bash

#启动之后我们可以通过 docker inspect 容器id 来查看卷挂载的目录
```

```
        "Name": "overlay2"
    },
    "Mounts": [
        {
            "Type": "bind",
            "Source": "/home/test",
            "Destination": "/home",
            "Mode": "",
            "RW": true,
            "Propagation": "rprivate"
        }
    ],
    "Config": {
```

挂载 -v 卷

宿主机地址

容器内地址

```
#测试文件的同步
```

```
root@tcheng:/home# docker run -it -v /home/test:/home centos /bin/bash
[root@a9dbeb5b7824 /]# ls
bin  etc   lib    lost+found  mnt  proc  run   srv  tmp  var
dev  home  lib64  media       opt  root  sbin  sys  usr
[root@a9dbeb5b7824 /]# cd /home
[root@a9dbeb5b7824 home]# ls
[root@a9dbeb5b7824 home]#
[root@a9dbeb5b7824 home]# touch test.py
[root@a9dbeb5b7824 home]# ls
test.py
[root@a9dbeb5b7824 home]#
```

```
root@tcheng:/home# ls
tcheng  test
root@tcheng:/home# cd test
root@tcheng:/home/test# ls
root@tcheng:/home/test# ls
test.py
root@tcheng:/home/test#
```

```
#再一次测试
1、停止容器
2、在宿主机上修改文件
3、启动容器
4、容器内的数据依旧是同步的
```

```
root@tcheng:/home# docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS                       PORTS     NAMES
a9dbeb5b7824   centos    "/bin/bash"              20 minutes ago   Exited (0) 9 minutes ago               pensive_saha
1a83920f9666   tomcat    "/bin/bash"              6 hours ago      Exited (0) 6 hours ago                 focused_shannon
1a99374cef4d   nginx     "/docker-entrypoint.…"   2 days ago       Exited (0) 24 hours ago                nginx01
0c504c42e552   centos    "/bin/bash"              2 days ago       Exited (127) 27 seconds ago            goofy_saha
root@tcheng:/home# docker start a9dbeb5b7824
a9dbeb5b7824
root@tcheng:/home# docker attach a9dbeb5b7824
[root@a9dbeb5b7824 /]# cd /home
[root@a9dbeb5b7824 home]# ls
test.py
[root@a9dbeb5b7824 home]# cat test.py
linux data update
[root@a9dbeb5b7824 home]#
```

```
root@tcheng:/home/test# ls
test.py
root@tcheng:/home/test# cat test.py
linux data update
root@tcheng:/home/test#
```

好处：只需要修改本地文件，容器内会自动同步！

实战：安装MySQL

思考：MySQL的数据持久化问题！

```
#获取镜像
root@tcheng:/home# docker pull mysql:5.7

#运行容器，需要做数据挂载  #安装启动MySQL，需要配置密码的，这是要注意的一个点！
#官方测试   docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag

#启动我们的MySQL
-d       后台运行
-p       端口映射
-v       卷挂载
-e       环境配置
--name   容器名称
root@tcheng:/home# docker run -d -p 3310:3306 -v /home/mysql/conf:/etc/mysql/conf.d -v /home/mysql/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 --name=mysql01 mysql:5.7

#启动成功之后，我们在本地使用 sqlyog 来测试一下
#sqlyog-连接到服务器的3310 --- 3310 和 容器内的 3306 映射，这个时候我们就可以连接上了！
```

假设我们将容器删除

```
root@tcheng:/home# docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          S
TATUS          PORTS
NAMES
bada30697507   mysql:5.7   "docker-entrypoint.s…"   33 minutes ago   U
p 33 minutes   33060/tcp, 0.0.0.0:3310->3306/tcp, :::3310->3306/tcp
mysql01
root@tcheng:/home# docker rm -f mysql01
mysql01
root@tcheng:/home# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
root@tcheng:/home# docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED       STATUS
                   PORTS     NAMES
a9dbeb5b7824   centos    "/bin/bash"              6 days ago    Exited
(127) 6 days ago             pensive_saha
1a83920f9666   tomcat    "/bin/bash"              7 days ago    Exited
(0) 7 days ago               focused_shannon
1a99374cef4d   nginx     "/docker-entrypoint.…"   8 days ago    Exited
(0) 7 days ago               nginx01
0c504c42e552   centos    "/bin/bash"              9 days ago    Exited
(127) 6 days ago             goofy_saha
root@tcheng:/home#
```

```
tcheng@tcheng:/home$ ls
mysql  tcheng  test
tcheng@tcheng:/home$ cd mysql/
tcheng@tcheng:/home/mysql$ ls
conf  data
tcheng@tcheng:/home/mysql$ ls
conf  data
tcheng@tcheng:/home/mysql$ cd data/
tcheng@tcheng:/home/mysql/data$ ls
auto.cnf         ib_buffer_pool  mysql               server-key.pem
ca-key.pem       ibdata1         performance_schema  sys
ca.pem           ib_logfile0     private_key.pem
client-cert.pem  ib_logfile1     public_key.pem
client-key.pem   ibtmp1          server-cert.pem
tcheng@tcheng:/home/mysql/data$
```

发现，我们挂载到本地的数据卷依旧没有丢失，这就实现了容器数据持久化功能!

Dockerfile

Dockerfile介绍

dockerfile是用来构建docker镜像的文件！是一个命令参数脚本!

构建步骤：

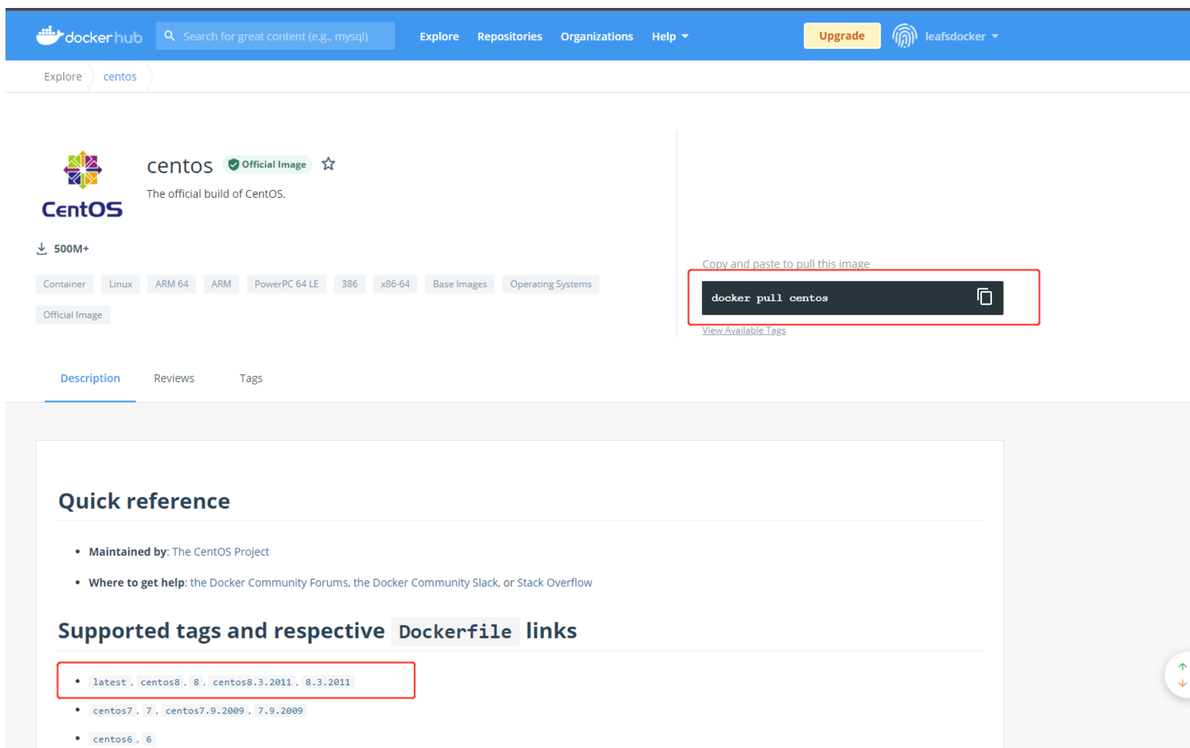
1、编写一个 dockerfile 文件

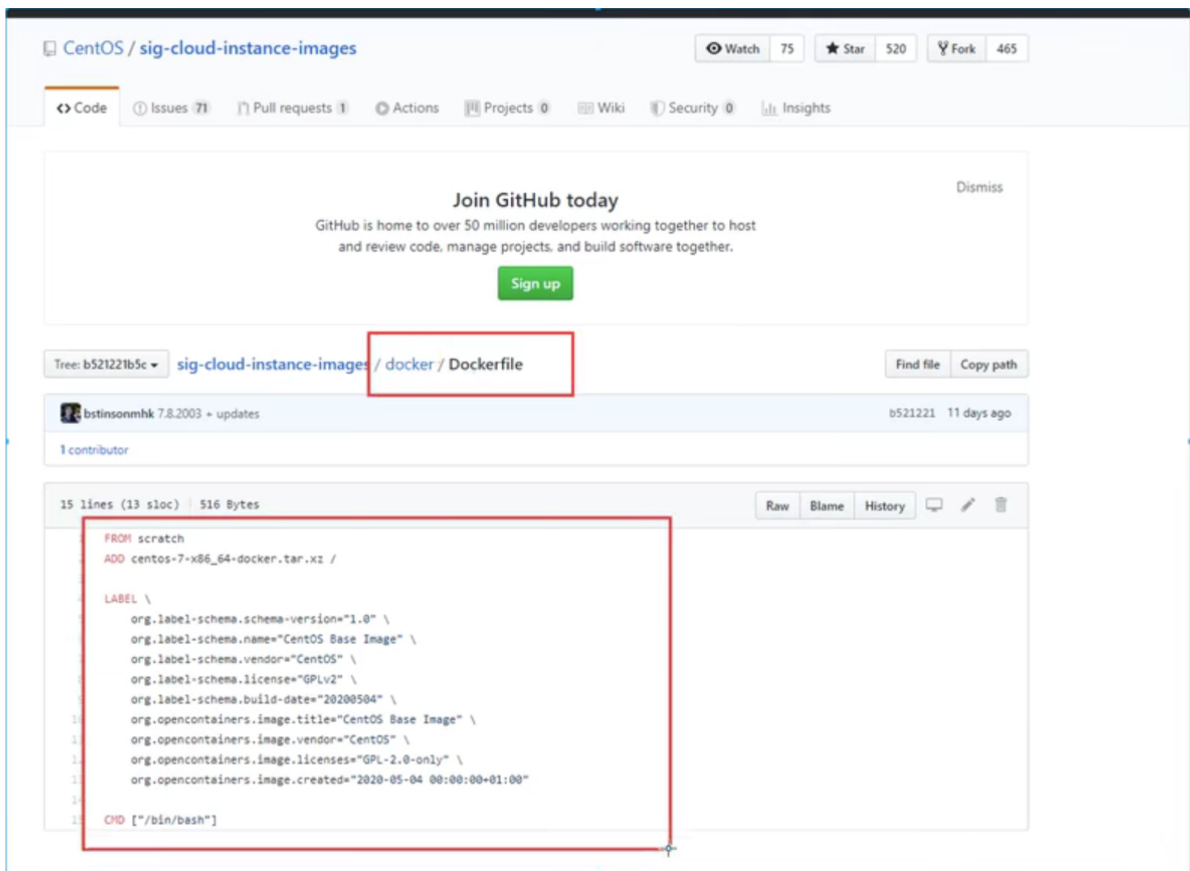
2、docker build 构建成为一个镜像

3、docker run 运行镜像

4、docker push 发布镜像（DocekrHub、阿里云镜像仓库！）

查看一下官方是怎么做的?





很多官方镜像都是基础包，很多功能都没有，我们通常会自己搭建自己的镜像！

官方既然可以制作镜像！那我们也可以！

DockerFile构建过程

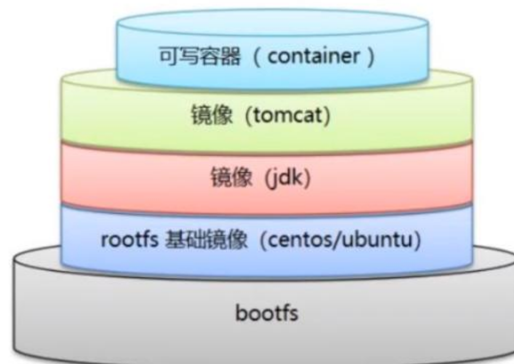
基础知识：

1、每个保留关键字（指令）都必须是大写字母

2、执行从上到下顺序执行

3、# 表示注释

4、每一个指令都会创建一个新的镜像层！并提交！



dockerfile 是面向开发的，我们以后要发布项目，做镜像，就需要编写 dockerfile 文件，这个文件十分简单！

步骤：开发、部署、运维。。。缺一不可！

Docker 镜像逐渐成为企业交付的标准，必须要掌握！

DockerFile：构建文件，定义了一切的步骤，源代码

DockerImages：通过 DockerFile 构建生成的镜像，最终发布和运行的产品！

Docker容器：容器就是镜像运行起来提供服务器

DockerFile的指令

以前的话我们就是使用别人的镜像，现在我们知道了这些指令后，我们来练习自己写一个镜像！

```
FROM          # 基础镜像，一切从这里开始构建
MAINTAINER    # 镜像是谁写的，姓名+邮箱
RUN           # 镜像构建的时候需要运行的命令
ADD           # 步骤，tomcat镜像，这个tomcat压缩包！添加内容
WORKDIR       # 镜像的工作目录
VOLUME        # 挂载的目录
EXPOSE        # 暴露端口配置
CMD           # 指定这个容器启动的时候要运行的命令，只有最后一个会生效，可被替代
ENTRYPOINT    # 指点这个容器启动的时候要运行的命令，可以追加命令
ONBUILD       # 当构建一个被继承 DockerFile 这个时候就会运行 ONBUILD 的指令，触发指令
COPY          # 类似ADD，将我们的文件拷贝到镜像中
ENV           # 构建的时候设置环境变量！
```

实战测试

DockerHub中 99% 的镜像都是从这个基础镜像过来的 FROM scratch，然后配置需要的软件和配置来进行构建

b2d195220e sig-cloud-instance-images / docker / Dockerfile

Go to file ···

bstinsonmhk CentOS 7.9.2009 images Latest commit b2d1952 on 14 Nov 2020 History

1 contributor

15 lines (13 sloc) | 516 Bytes

Raw Blame

```
FROM scratch
ADD centos-7-x86_64-docker.tar.xz /

LABEL \
    org.label-schema.schema-version="1.0" \
    org.label-schema.name="CentOS Base Image" \
    org.label-schema.vendor="CentOS" \
    org.label-schema.license="GPLv2" \
    org.label-schema.build-date="20201113" \
    org.opencontainers.image.title="CentOS Base Image" \
    org.opencontainers.image.vendor="CentOS" \
    org.opencontainers.image.licenses="GPL-2.0-only" \
    org.opencontainers.image.created="2020-11-13 00:00:00+00:00"

CMD ["/bin/bash"]
```

> 创建一个自己的centos

```
# 1.编写DockerFile文件
root@tcheng:/home/dockerfile# cat mydockerfile
FROM centos
MAINTAINER leaf<5824612@qq.com>

ENV MYPATH /usr/local
WORKDIR $MYPATH

RUN apt-get install vim
RUN apt-get install net-tools

EXPOSE 80

CMD echo $MYPATH
CMD echo "------end------"
CMD /bin/bash

# 2.通过这个文件构建镜像
# 命令 docker build -f dockerfile文件路径 -t 镜像名:[tag]

# 3.测试运行
```

> CMD 和 ENTRYPOINT 区别

```
CMD                     #指定这个容器启动的时候要运行的命令，只有最后一个会生效，可别替代
ENTRYPONIT              #指定这个容器启动的时候要运行的命令，可以追加命令
```

实战：Romcat镜像

1、准备镜像文件 tomcat 压缩包， jdk的压缩包！

```
root@tcheng:/home/tomcat# ll
total 153672
drwxr-xr-x 2 root root        4096 7月  31 11:05 ./
drwxr-xr-x 7 root root        4096 7月  31 11:05 ../
-rwxrwxrwx 1 root vboxsf  11825772 7月  31 11:02 apache-tomcat-10.1.0-M2.tar.gz*
-rwxrwxrwx 1 root vboxsf 145520298 7月  31 10:58 jdk-8u301-linux-x64.tar.gz*
root@tcheng:/home/tomcat#
```

2、编写dockerfile文件，官方命名 Dockerfile ， build时会自动寻找这个文件，就不需要 -f 指定了！

```
FROM centos
MAINTAINER leaf<123564898@qq.com>

COPY readme.txt /usr/local/readme.txt

ADD jdk-8u301-linux-x64.tar.gz /usr/local
ADD apache-tomcat-10.1.0-M2.tar.gz /usr/local

RUN yum -y  install vim

ENV MYPATH /usr/local
WORKDIR $MYPATH

ENV JAVA_HOME /usr/local/jdk1.8.0_301
ENV CLASSPATH $JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
ENV CATALINA_HOME /usr/local/apache-tomcat-10.1.0-M2
ENV CATALINA_BASH /usr/local/apache-tomcat-10.1.0-M2
ENV PATH $PATH:$JAVA_HOME/bin:$CATALINA_HOME/lib:$CATALINA_HOME/bin

EXPOSE 8080

CMD /usr/local/apache-tomcat-10.1.0-M2/bin/startup.sh && tail -F
/usr/local/apache-tomcat-10.1.0-M2/bin/logs/catalina.out
```

3、构建镜像

```
# docker build -t mytomcat .
```

4、启动镜像

5、访问测试

6、发布项目（由于做了卷挂载，我们直接在本地编写项目就可以发布了！）

发布自己的镜像

> DockerHub

1、地址 https://hub.docker.com/ 注册自己的账号！

2、确定这个账号可以登录

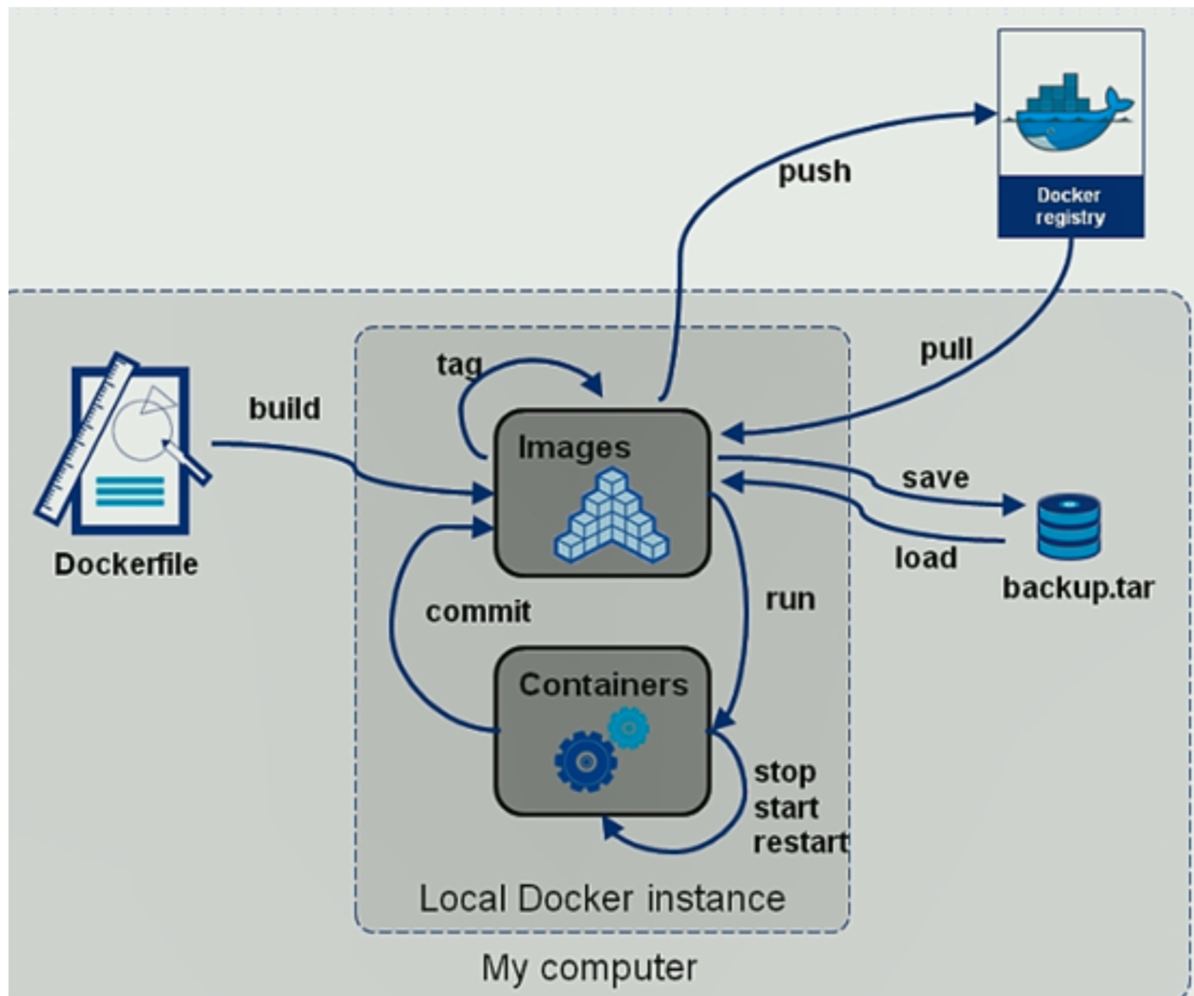
```
root@tcheng:/home# docker login --help

Usage:  docker login [OPTIONS] [SERVER]

Log in to a Docker registry.
If no server is specified, the default is defined by the daemon.

Options:
  -p, --password string   Password
      --password-stdin    Take the password from stdin
  -u, --username string   Username
```

```
root@tcheng:/home# docker login -u leafsdocker
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

3、在我们服务器上提交自己的镜像

4、登录完毕后就可以提交镜像了，就是一步 docker push

小结



Docker网络

理解Docker0

清空所有

> 测试

```
root@tcheng: /home
root@tcheng:/home# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:53:2c:41 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 85749sec preferred_lft 85749sec
    inet6 fe80::7577:6c93:bfbe:eeeb/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:9b:69:8e:d8 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
root@tcheng:/home#
```

本机回环地址

docker0 地址

```
# 问题， docker 是如何处理容器网络访问的?
```

```
root@tcheng:/home# docker run -d -P --name tomcat01 tomcat

# 查看容器的内部网络地址 ip addr  发现容器启动时会得到一个 eth0@if5 ip地址，docker分配的!
root@tcheng:/home# docker exec -it tomcat01 ip addr
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
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever

# 思考，Linux主机能不能 ping 通容器内部的ip!
root@tcheng:/home# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.078 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.107 ms

# Linux 可以 ping 通 docker 容器内部ip
```

> 原理

1、我们每启动一个docker容器，docker就会给docker容器分配一个ip，我们只要安装了docker，就会有一个网卡 docker0 桥接模式，使用的技术是 evth-pair 技术！

```
# 再次测试 ip addr
```

```
root@tcheng:/home# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:53:2c:41 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 85097sec preferred_lft 85097sec
    inet6 fe80::7577:6c93:bfbe:eeeb/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:9b:69:8e:d8 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:9bff:fe69:8ed8/64 scope link
       valid_lft forever preferred_lft forever
5: veth89fe07b@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether f2:d2:b4:98:e7:6e brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::f0d2:b4ff:fe98:e76e/64 scope link
       valid_lft forever preferred_lft forever
root@tcheng:/home#
```

2、再启动一个容器测试，发现又多了一对网卡！

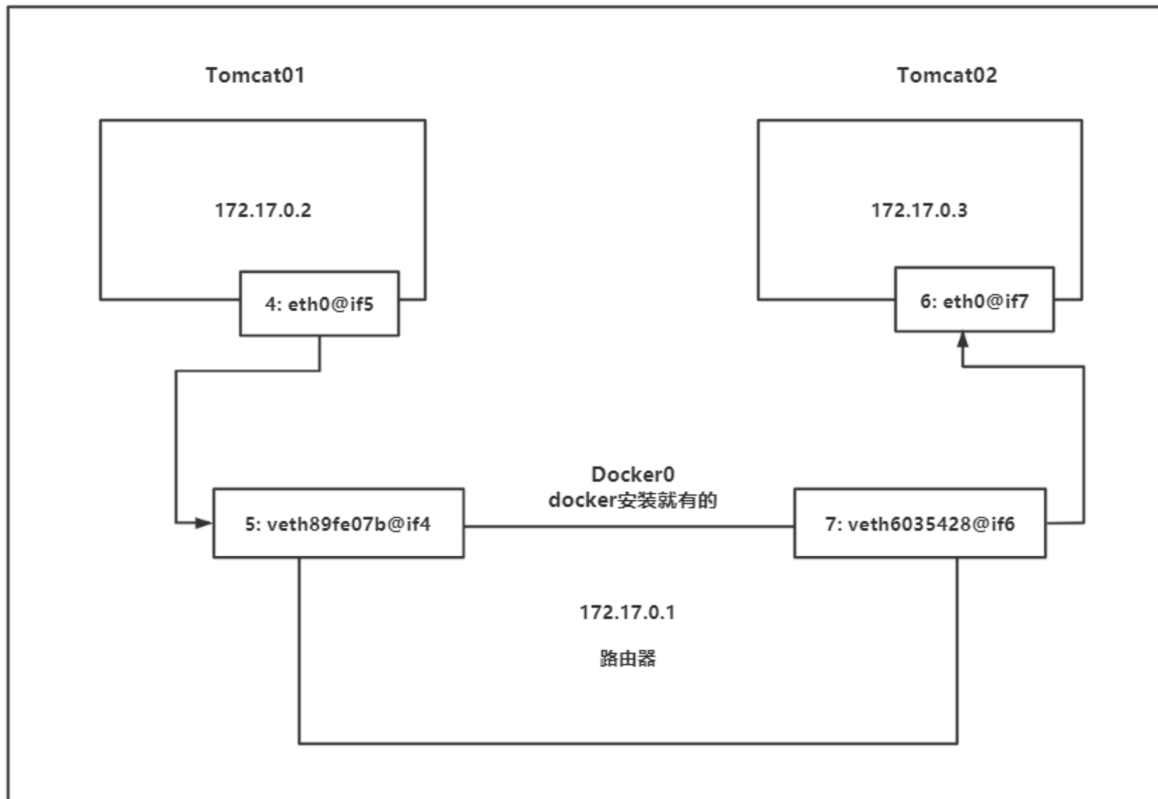
```
root@tcheng:/home# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:53:2c:41 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 84950sec preferred_lft 84950sec
    inet6 fe80::7577:6c93:bfbe:eeeb/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:9b:69:8e:d8 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:9bff:fe69:8ed8/64 scope link
       valid_lft forever preferred_lft forever
5: veth89fe07b@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether f2:d2:b4:98:e7:6e brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::f0d2:b4ff:fe98:e76e/64 scope link
       valid_lft forever preferred_lft forever
7: veth6035428@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether e2:2a:28:c5:4f:c8 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::e02a:28ff:fec5:4fc8/64 scope link
       valid_lft forever preferred_lft forever
```

```
# 我们发现这个容器带来的网卡，都是一对一对的
# evth-pair 就是一对的虚拟设备接口，它们都是成对出现的，一段连着协议，一段彼此相连
# 正因为有这个特性，evth-pair 充当一个桥梁，连接各种虚拟网络设备的
```

3、我们来测试下 tomcat01 和 tomcat02 是否可以 ping 通！

```
root@tcheng:/home# docker exec -it tomcat02 ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.195 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.137 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.139 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.252 ms

# 结论， 容器和容器之间是可以互相 ping 通的！
```

绘制一个网络模型图：

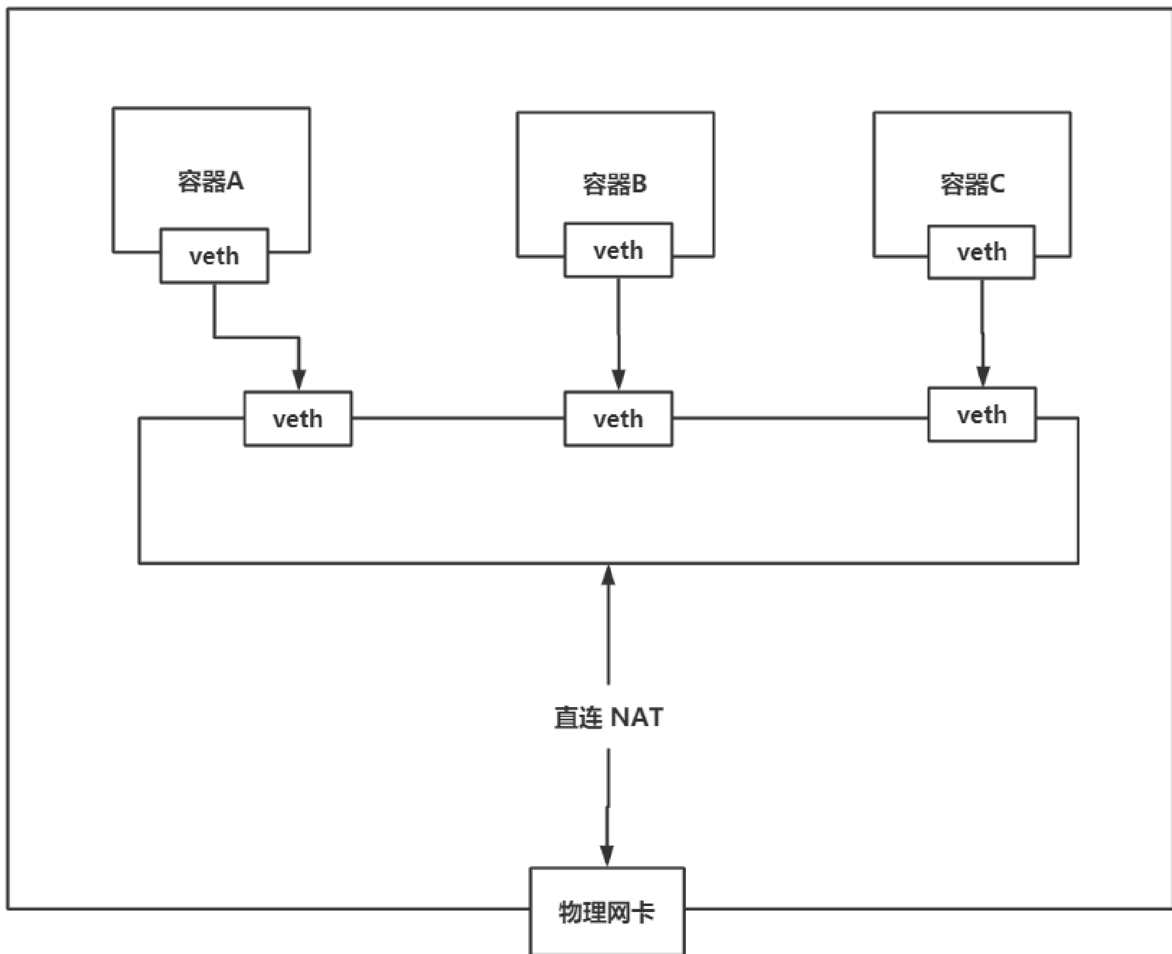


结论：tomcat01 和 tomcat02 是公用的一个路由器，也就是docker0。

所有的容器在不指定网络的情况下，都是 docker0 路由的，docker会给每一个容器分配一个默认的可用ip

> 小结

Docker 使用的是 Linux 的桥接，宿主机中是一个 Docker 容器的网桥，也就是 docker0



Docker 中的所有网络接口都是虚拟化的，虚拟的转发效率高！（内网传递文件！）

只要容器删除，对应的一对网桥也就没了！

> 思考一个场景，我们编写了一个微服务，database url=ip: 项目不重启，数据可ip换掉了，我们希望可以处理这个问题，可以通过名字来进行访问容器?