Docker教程

学前准备

1.Linux (必要)

1.Docker 概述

Docker 为什么出现?

一款产品：开发一上线两套环境！应用环境，应用配置！

2.Docker的历史

2010年，几个年轻人，在美国成立了一家dotCloud

Docker开源以后，每个月都会更新一个版本。

Docker是基于Go语言开发的！开源项目

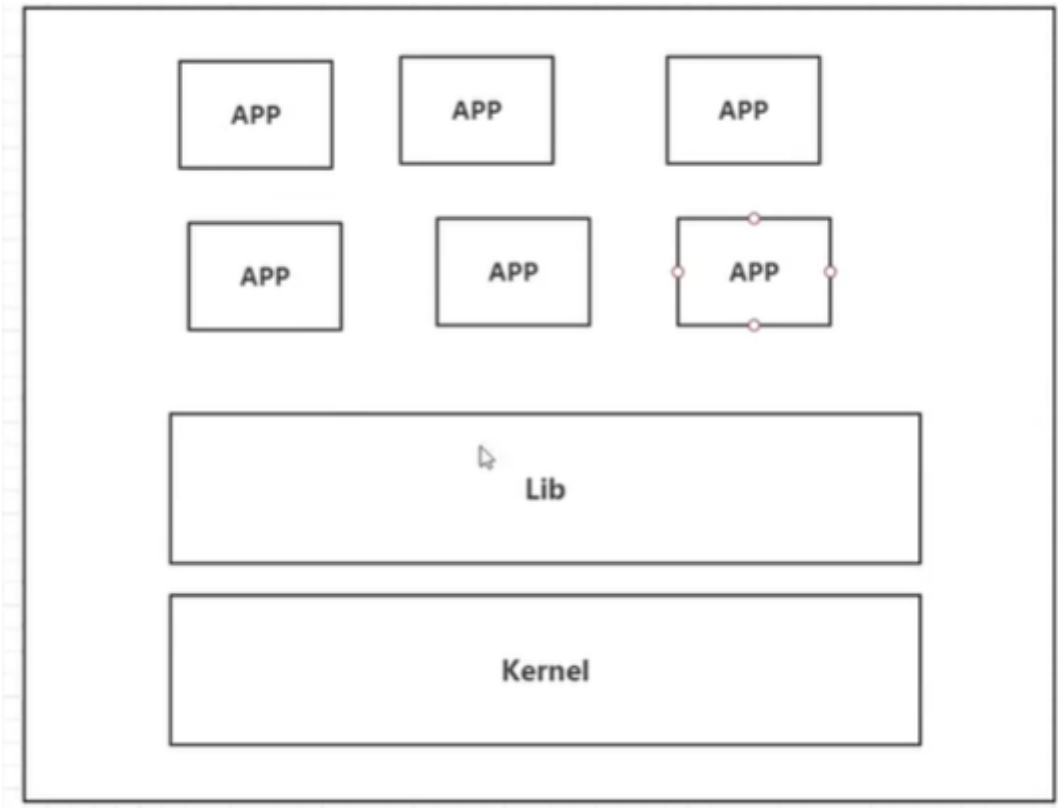
官网：https://www.docker.com/

文档地址：https://docs.docker.com/

仓库地址：https://hub.docker.com

3.Docker能干嘛

与虚拟机技术做对比（模拟一台电脑）

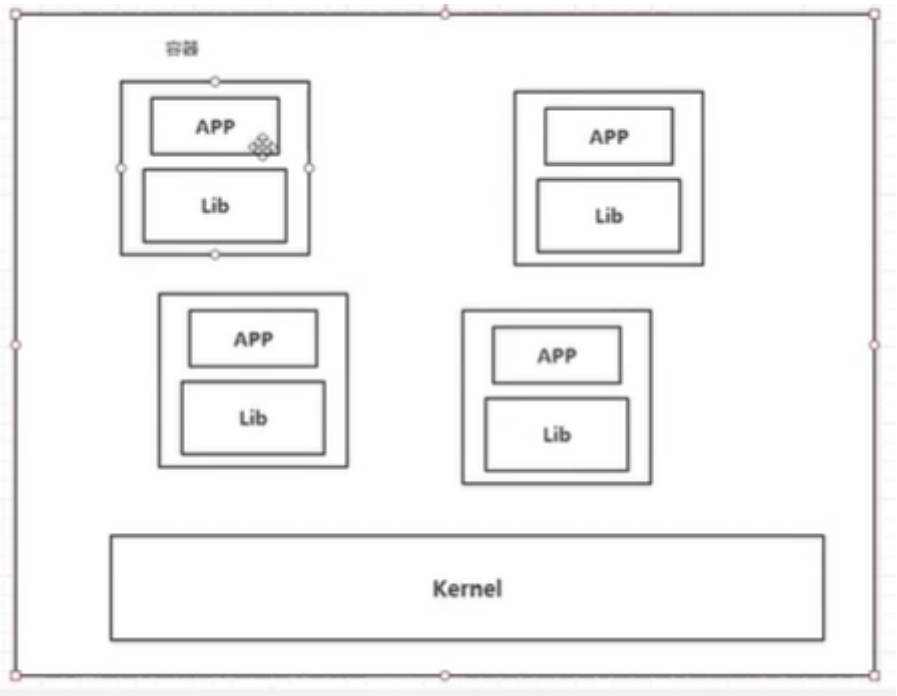


虚拟机技术缺点：

- 资源占用多
- 冗余步骤多
- 启动很慢

容器化技术

容器化技术不是模拟的一个完整的操作系统



比较Docker和虚拟机技术的不同

- 传统虚拟机，虚拟出一条硬件，运行个完整的操作系统，然后在这个系统上安装和运行软件
- 容器内的应用直接运行在宿主机的内核，容器是没有自己的内核的，也没有虚拟硬件，所以就轻便了
- 每个容器间是互相隔离，每个容器内部有一个属于自己的文件系统，互不影响。

DevOps(开发，运维)

应用更快速 的交付和部署

传统：一堆帮助文档，安装程序

Docker:打包镜像发布测试，一键运行，

更便捷的升级和扩缩容

使用了Docker之后，我们部署应用就和搭积木一样！

更简单的系统运维

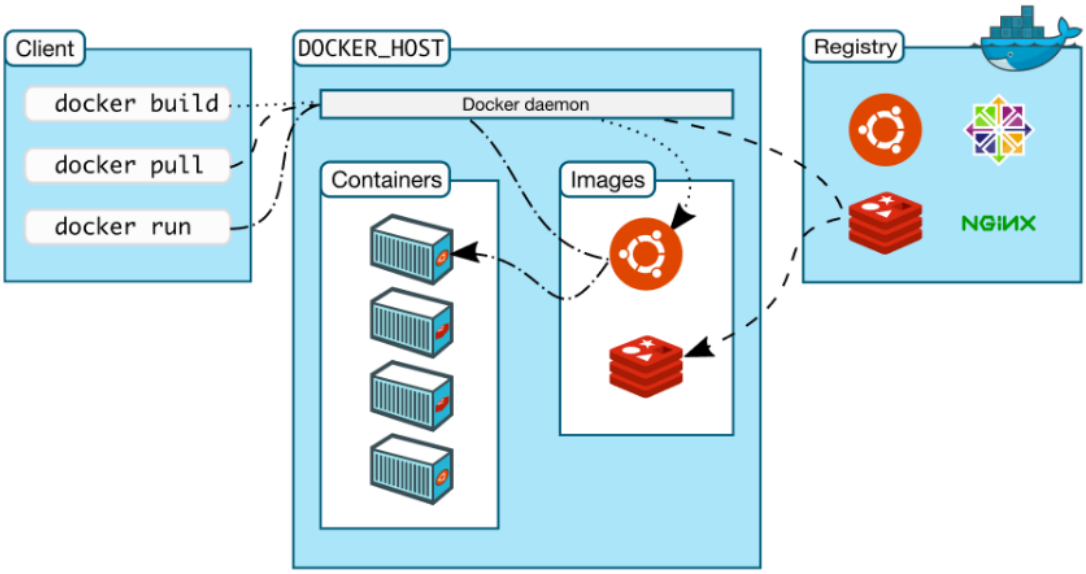
在容器化之后，我们的开发，测试环境是高度一致的

更高效的计算资源利用

Docker是内核级别的虚拟化，可以在一个虚拟机上运行很多的容器实例！服务器性能可以压榨到极致。

4.Docker安装

4.1.Docker的基本组成



镜像（Image）

docker镜像就好比一个模板，可以通过这个模板来创建容器服务，tomcat镜像 ===>run===>tomcat01()容器（提供服务器），通过这个镜像可以创建多个容器(最终服务运行或项目运行的地方)。

容器（container）

Docker利用容器技术，独立运行一个或者一组应用，通过镜像来创建的。

启动、停止、删除，基本命令！

目前就可以把这个容器理解为一个简易的Linux系统

仓库（repository）

仓库就是存放镜像的地方！

仓库分为公有仓库和私有仓库！

Docker Hub(默认是国外的)

阿里云...都有容器服务器（配置镜像加速！）

4.2 安装Docker

环境准备

1.需要一点点Linux基础

2.CentOS7 系统

3.使用Xshell连接远程服务器

环境查看

```
# 系统内核3.10以上
[root@vultr /]# uname -r
5.6.14-1.el7.elrepo.x86_64
```

```
[root@vultr /]# cat /etc/os-release
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
```

```
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"
```

安装

帮助文档:

```
# 1.卸载旧版的docker
sudo yum remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-engine
# 2.安装需要的安装包
sudo yum install -y yum-utils
# 3.设置镜像的仓库
sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo      # 默认是国外的仓库,
如果慢的话可以换成国内阿里云的镜像

# 4.更新yum
sudo yum makecache fast
# 5.安装docker 相关的内容 ，docker-ce社区版，ee企业版
sudo yum-config-manager --enable docker-ce-nightly

# 6.启动docker
sudo systemctl start docker
# 7.查看docker启动状态
sudo systemctl status docker

# 8.测试docker
sudo docker run hello-world
```

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:6a65f928fb91fcfbc963f7aa6d57c8eeb426ad9a20c7ee045538ef34847f44f1
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

```
# 8.查看一下下载的这个hello-world 镜像
sudo docker images
```

```
[root@vultr ~]# sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
 SIZE
hello-world         latest              bf756fb1ae65        4 months ago
 13.3kB
```

```
# 卸载docker
sudo yum remove docker-ce docker-ce-cli containerd.io
sudo rm -rf /var/lib/docker
```

阿里云镜像加速

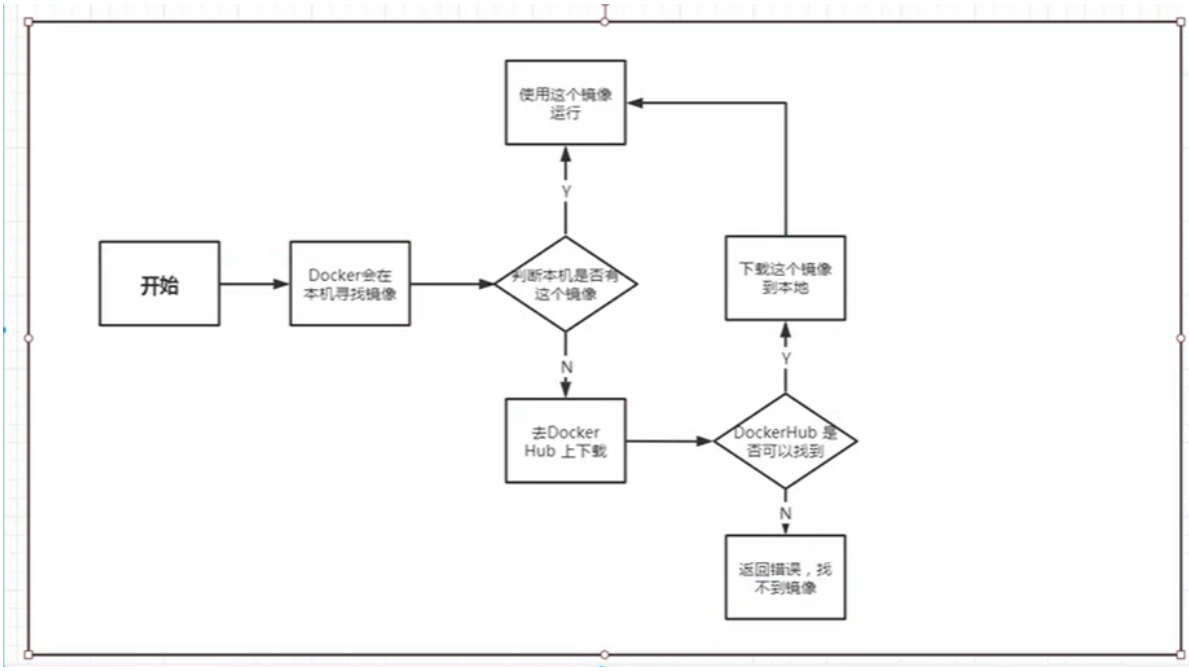
1.登录阿里云找到容器服务

2.找到镜像加速地址，里面有操作文档

3.根据配置文档进行配置

回顾HelloWorld的流程

```
docker run Hello-World
```

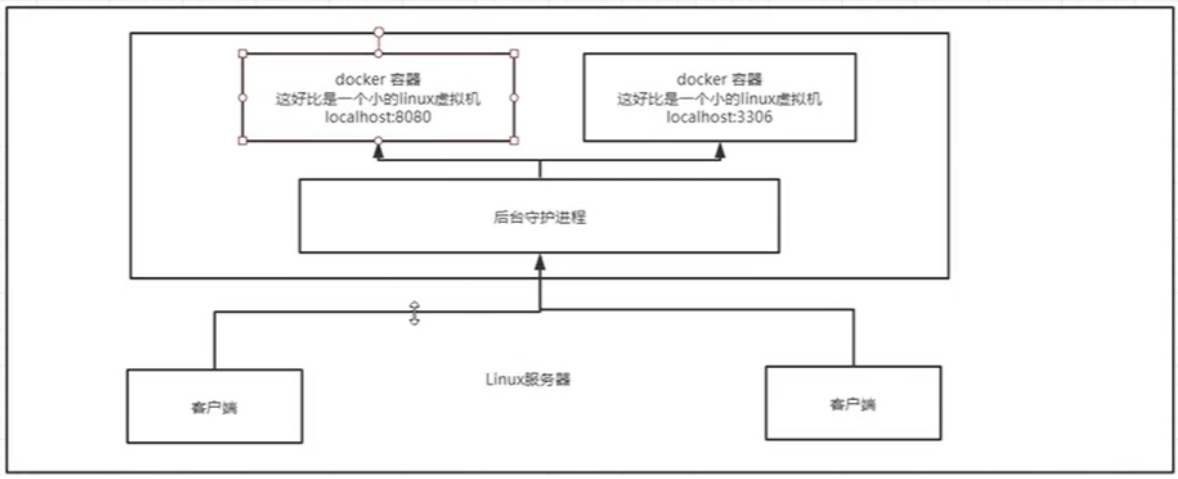


4.3 底层原理

Docker是怎么工作的?

Docker是一个C / S架构系统，Docker的守护进程运行在主机上，通过Socket从客户端访问

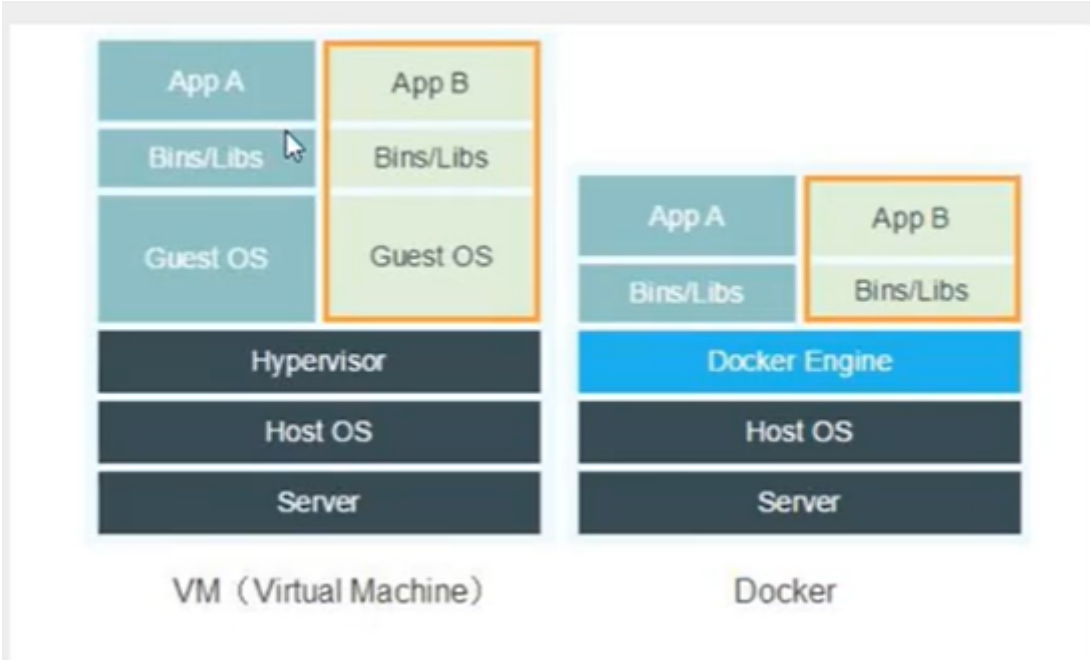
DockerServer接收到Docker-Client的指令，就会执行这个命令!



Docker为什么比虚拟机快

1、Docker有着比虚拟机更少的抽象层

2、docker利用的是宿主机的内核，Vm需要GuestOS



所以说，新建一个容器的时候，docker不需要想虚拟机一样重新加载一个操作系统内核，避免引导，虚拟机是加载Guest OS ,分钟级别的，而docker利用宿主机的操作系统。

4.4 Docker的常用命令

帮助命令

```
docker version      # docker 版本
docker info         # 显示docker 系统信息，包括镜像和容器数量
docker 命令 --help  # 帮助命令
```

帮助文档地址： https://docs.docker.com/engine/reference/builder/

4.4.1 镜像命令

```
[root@vultr ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
hello-world         latest              bf756fb1ae65        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
CREATE     镜像的创建时间
SIZE       镜像的大小
# 可选参数
-a --all           # 列出所有镜像
-q --quite         # 只列出镜像的ID
```

镜像搜索

```
[root@vultr ~]# docker search mysql
NAME                              DESCRIPTION
STARS               OFFICIAL            AUTOMATED
mysql                             MySQL is a widely used, open-source relation…
9550                [OK]
mariadb                           MariaDB is a community-developed fork of MyS…
3469                [OK]
mysql/mysql-server                Optimized MySQL Server Docker images. Create…
700                 [OK]
.........................

# 可选项
docker search 搜索的是docker商店里面的内容，和在官网搜索框搜索的一样。
--filter=STARS=3000    # 搜索出来的镜像就是STARS>3000的
```

镜像下载

```
[root@vultr ~]# docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql
afb6ec6fdc1c: Pull complete      # 分层下载，docker image的核心 联合文件系统
0bdc5971ba40: Pull complete
97ae94a2c729: Pull complete
f777521d340e: Pull complete
1393ff7fc871: Pull complete
a499b89994d9: Pull complete
7ebe8eefbafe: Pull complete
597069368ef1: Pull complete
ce39a5501878: Pull complete
7d545bca14bf: Pull complete
211e5bb2ae7b: Pull complete
5914e537c077: Pull complete
Digest: sha256:a31a277d8d39450220c722c1302a345c84206e7fd4cdb619e7face046e89031d
       # 签名信息（防伪标志）
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest                    # docker真实地址

# 等价命令
docker pull mysql
docker pull docker.io/library/mysql:latest

可选参数
docker pull 镜像名[:tag]          # 指定镜像的版本号，默认是下载最新版
```

删除镜像

```
docker rmi -f IMAGE ID   # 删除指定ID的镜像（删除多个时要用空格隔开）
docker rmi -f $(docker images -aq)  # 删除所有镜像（仔细体会该命令）
```

4.4.2 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

下载centos镜像

```
[root@vultr ~]# docker pull centos
Using default tag: latest
latest: Pulling from library/centos
8a29a15cefae: Pull complete
Digest: sha256:fe8d824220415eed5477b63addf40fb06c3b049404242b31982106ac204f6700
Status: Downloaded newer image for centos:latest
docker.io/library/centos:latest
```

新建容器并启动

```
docker run [可选参数] image

# 参数说明
--name="Name"  # 跑起来后容器的名字
-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 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@vultr ~]# docker run -it centos /bin/bash
[root@5aaf0c538b30 /]# ls
bin  etc   lib    lost+found  mnt  proc  run   srv  tmp  var
dev  home  lib64  media       opt  root  sbin  sys  usr

# 从容器中退回到主机(容器停止并退出)   ctrl + p + q   容器不停止退出
[root@5aaf0c538b30 /]# exit
exit
[root@vultr ~]#
```

列出所有运行的容器

```
#参数
docker ps          # 列出当前正在运行的容器
docker ps -a       # 列出当前正在运行的容器 + 带出历史运行的容器
docker ps -a -n=1  # 查出最近创建的1个容器
docker ps -q       # 只显示容器的编号
[root@vultr ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
[root@vultr ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                            PORTS               NAMES
5aaf0c538b30        centos              "/bin/bash"         8 minutes ago
Exited (130) About a minute ago                       determined_clarke
1da242b618f5        bf756fb1ae65        "/hello"            17 hours ago
Exited (0) 17 hours ago                               wonderful_einstein
[root@vultr ~]#
```

删除容器

```
docker rm 容器id                                    # 删除指定的容器，不能删除运行的容器，可以加
-f 强制删除
docker rm -f $(docker ps -aq)                       # 删除所有的容器
docker ps -a -q | xargs docker rm                   # 管道的方式删除所有的容器
```

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                  # 停止容器
docker kill 容器id                  # 强制停止当前容器
```

4.4.3 常用的其它命令

后台启动容器

```
# 命令 docker run -d 镜像名

[root@vultr ~]# docker run -d centos
9758cf3c87149b312d1094bae6c6f53aecf958197e391908e6915d55c8e38798
[root@vultr ~]#

# 问题docker ps，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，docker 发现没有应用，就会自动停止
# nginx,容器启动后，发现自己没有提供服务，就会立刻停止
```

查看日志

```
[root@vultr ~]# docker logs --help

Usage:  docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

Options:
      --details        Show extra details provided to logs
  -f, --follow         Follow log output
      --since string   Show logs since timestamp (e.g.
                       2013-01-02T13:23:37) or relative (e.g. 42m for 42
                       minutes)
      --tail string    Number of lines to show from the end of the logs
                       (default "all")
  -t, --timestamps     Show timestamps
      --until string   Show logs before a timestamp (e.g.
                       2013-01-02T13:23:37) or relative (e.g. 42m for 42
                       minutes)
```

查看容器中的进程信息

```
docker top 容器id
```

查看镜像的原数据

```
# 命令
```

```
docker inspect 容器id

# 测试
[root@vultr ~]# docker inspect 5aaf0c538b30
[
    {
        "Id":
"5aaf0c538b30a6c08837a32d0a2b569650fb708f092d376699115a6cb753fff9",
        "Created": "2020-05-28T01:04:55.14362426Z",
        "Path": "/bin/bash",
        "Args": [],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 0,
            "ExitCode": 130,
            "Error": "",
            "StartedAt": "2020-05-28T01:04:55.414849364Z",
            "FinishedAt": "2020-05-28T01:11:22.265728216Z"
        },
        "Image":
"sha256:47067167cac686c7cf0081e0b37da2e9f4f768ddc5f6a26102ccd1c6954c1ee",
        "ResolvConfPath":
"/var/lib/docker/containers/5aaf0c538b30a6c08837a32d0a2b569650fb708f092d37669911
5a6cb753fff9/resolv.conf",
        "HostnamePath":
"/var/lib/docker/containers/5aaf0c538b30a6c08837a32d0a2b569650fb708f092d37669911
5a6cb753fff9/hostname",
        "HostsPath":
"/var/lib/docker/containers/5aaf0c538b30a6c08837a32d0a2b569650fb708f092d37669911
5a6cb753fff9/hosts",
        "LogPath":
"/var/lib/docker/containers/5aaf0c538b30a6c08837a32d0a2b569650fb708f092d37669911
5a6cb753fff9/5aaf0c538b30a6c08837a32d0a2b569650fb708f092d376699115a6cb753fff9-
json.log",
        "Name": "/determined_clarke",
        "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,
        "CapAdd": null,
        "CapDrop": null,
        "Capabilities": null,
        "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/overlay2/44c40bd84b09c8f877caccb99c9e79a5c066976acaafdefabcd755
32a35f2d6a-
init/diff:/var/lib/docker/overlay2/4e52dc0891072fc1d03e26e011f46c5b6632fc85a5722
3d3d8cafc308ee62e7f/diff",
                "MergedDir":
"/var/lib/docker/overlay2/44c40bd84b09c8f877caccb99c9e79a5c066976acaafdefabcd755
32a35f2d6a/merged",
                "UpperDir":
"/var/lib/docker/overlay2/44c40bd84b09c8f877caccb99c9e79a5c066976acaafdefabcd755
32a35f2d6a/diff",
                "WorkDir":
"/var/lib/docker/overlay2/44c40bd84b09c8f877caccb99c9e79a5c066976acaafdefabcd755
32a35f2d6a/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "5aaf0c538b30",
            "Domainname": "",
            "User": "",
            "AttachStdin": true,
            "AttachStdout": true,
            "AttachStderr": true,
            "Tty": true,
            "OpenStdin": true,
            "StdinOnce": true,
            "Env": [

 "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
```

```
            "Cmd": [
                "/bin/bash"
            ],
            "Image": "centos",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "org.label-schema.build-date": "20200114",
                "org.label-schema.license": "GPLv2",
                "org.label-schema.name": "CentOS Base Image",
                "org.label-schema.schema-version": "1.0",
                "org.label-schema.vendor": "CentOS",
                "org.opencontainers.image.created": "2020-01-14 00:00:00-08:00",
                "org.opencontainers.image.licenses": "GPL-2.0-only",
                "org.opencontainers.image.title": "CentOS Base Image",
                "org.opencontainers.image.vendor": "CentOS"
            }
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID":
"844655d0c237102ab0a33deadce662381f3b749992c6d9362f3e191be568e6e9",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {},
            "SandboxKey": "/var/run/docker/netns/844655d0c237",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID": "",
            "Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "",
            "IPPrefixLen": 0,
            "IPv6Gateway": "",
            "MacAddress": "",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID":
"fbba2e0e5337c940fbbdc1d5038a5756522300e692460dc6217cda791bd07130",
                    "EndpointID": "",
                    "Gateway": "",
                    "IPAddress": "",
                    "IPPrefixLen": 0,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "",
                    "DriverOpts": null
                }
            }
        }
```

```
    }
]
```

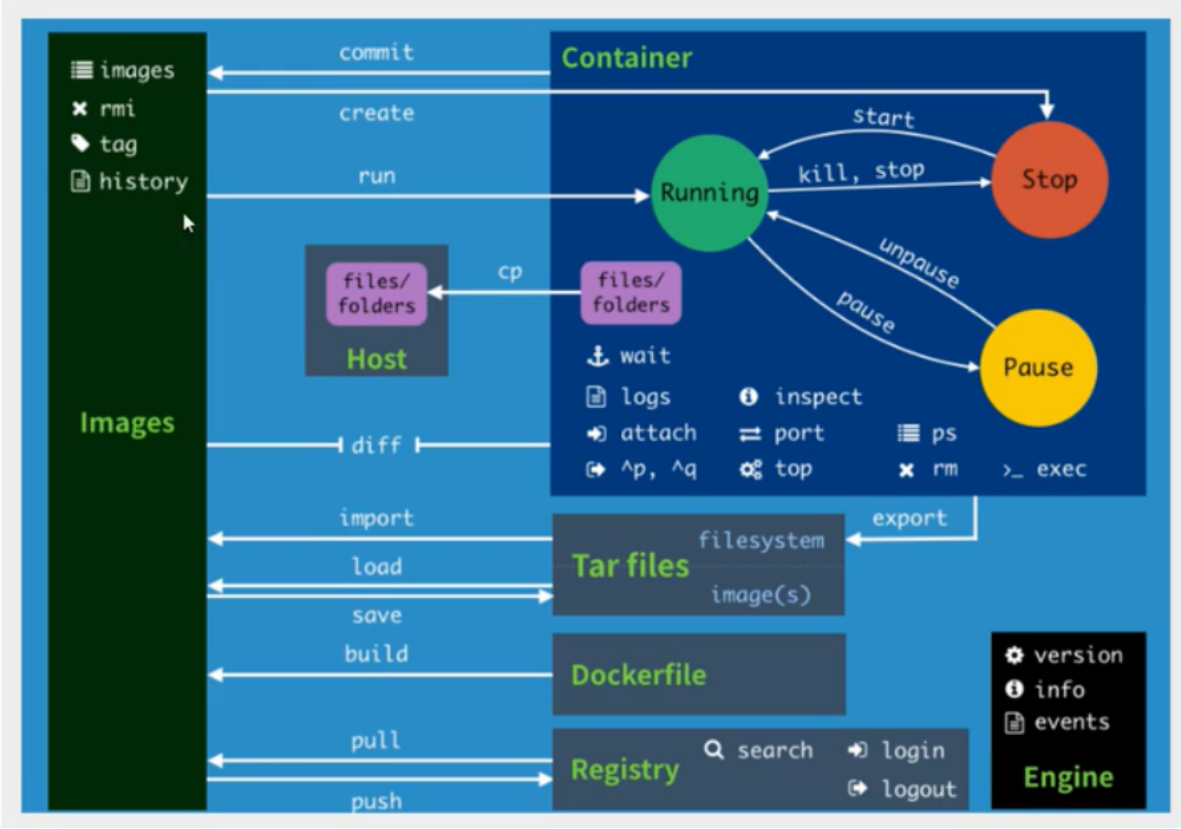
进入当前正在运行的容器

```
# 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
# 方式1（进入容器后开启一个新的终端，可以在里面操作）
docker exec -it 容器id bashShell
# 方式2(进入正在执行的终端，不会启动新的进程)
docker attach 容器id

# 测试
[root@vultr ~]# docker exec -it b6c14bcc1032 /bin/bash
[root@b6c14bcc1032 /]#
```

从容器内拷贝文件到主机

```
# 拷贝是一个手动过程，未来我们可以使用 -v 卷的技术实现自动同步
docker cp 容器id：容器内路径 目的主机路径
```

4.4.4 小结



4.5 Docker 可视化

- portainer(先用这个)

```
docker run -d -p 8088:9000 --restart=always -v
/var/run/docker.sock:/var/run/docker.sock --privileged=true
portainer/portainer
```

- Rancher(CI/CD再用)

什么是portainer?

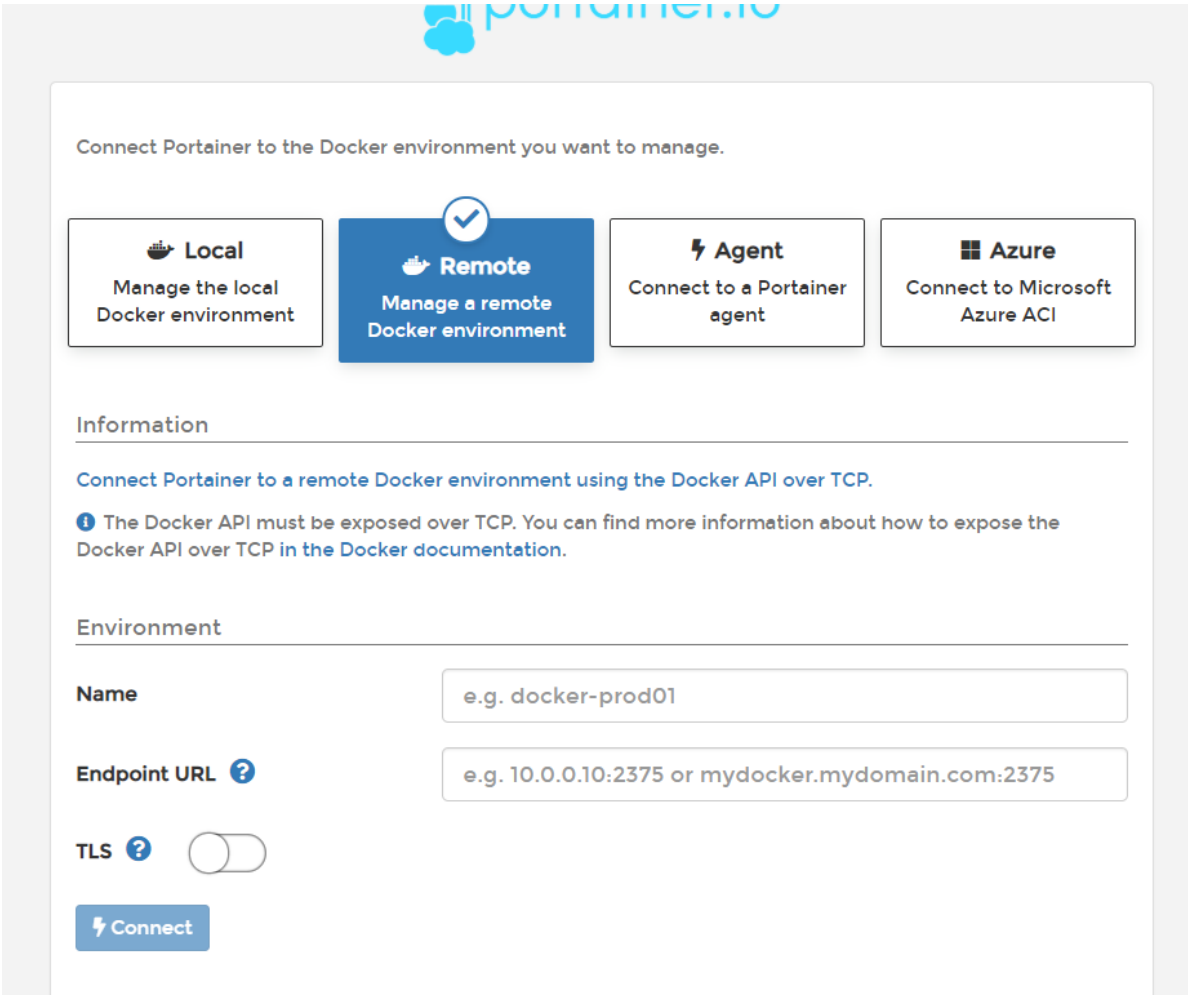
Docker 图形化界面管理工具！提供一个后台面板供我们操作！

下载安装：

```
[root@vultr ~]# docker run -d -p 8088:9000 --restart=always -v
/var/run/docker.sock:/var/run/docker.sock --privileged=true portainer/portainer
Unable to find image 'portainer/portainer:latest' locally
latest: Pulling from portainer/portainer
d1e017099d17: Pull complete
a7dca5b5a9e8: Pull complete
Digest: sha256:4ae7f14330b56ffc8728e63d355bc4bc7381417fa45ba0597e5dd32682901080
Status: Downloaded newer image for portainer/portainer:latest
2121d2dc15a3c2a4c2e2867b38091d03d5d10b459d4615e13e7ddb6b65092a5e
[root@vultr ~]#
```

访问测试：

```
http://ip:8088
```



5 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。

未来所有应用，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
- 朋友拷贝给你
- 自己制作一个镜像DockerFile

Docker镜像加载原理

UnionFS(联合文件系统)

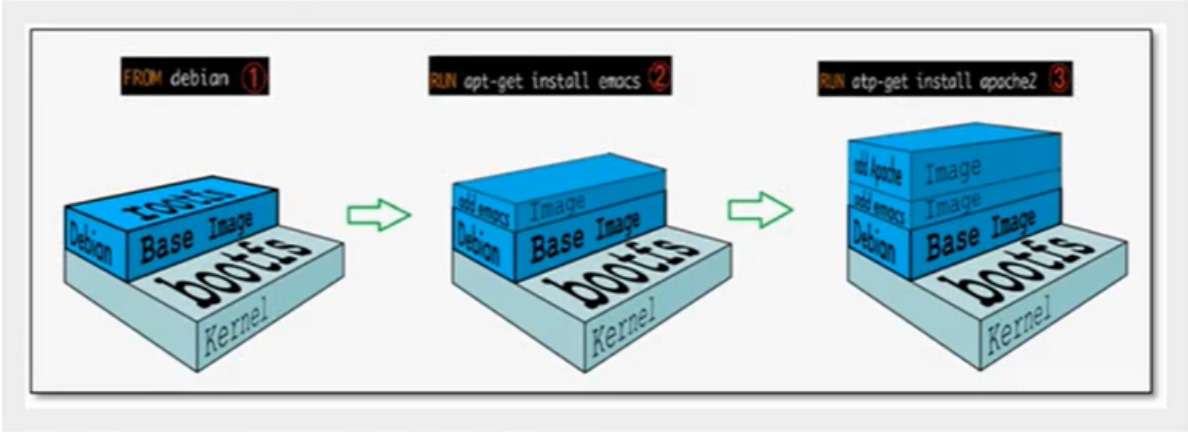
Unionfs（联合文件系统）： Union文件系统（UnionFS）是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个拟文件系统下（unite several directories into a single virtualfilesystem), Union文件系统是 Docker像的基础，镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。
特性：一次同时加载多个文件系统，但从外面看起来，只能看到一个文件系统，联合加载会把各层文件系统叠加起来，这样最终的文件系统会包含所有底层的文件和目录。

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kernal, Linux刚启动时会加载bootfs文件系统，在 Docker境像的最底层是 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，此时系统也会下载bootfs。

rootfs(root file system),在 boots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。



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

```
[root@vultr ~]# docker images centos
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              latest              470671670cac        4 months ago        237MB
[root@vultr ~]#
```

对于个精简的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inux发行版， boots基本是一致的， rootfs会有差别因此不同的发行版可以公用bootfs

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@vultr ~]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
afb6ec6fdc1c: Already exists
608641ee4c3f: Pull complete
668ab9e1f4bc: Pull complete
ea9ab8bf5f73: Pull complete
137e0d1a14d9: Pull complete
b2c5e1be4a59: Pull complete
Digest: sha256:89051d5ec46a89d4a708467af38eaaf4029450c4b1b9835ffd413cf70625b22e
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
[root@vultr ~]#
```

思考： 为什么Docker镜像要采用这种分层的结构呢?

最大的好处，我感觉莫过于是资源共享了！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!

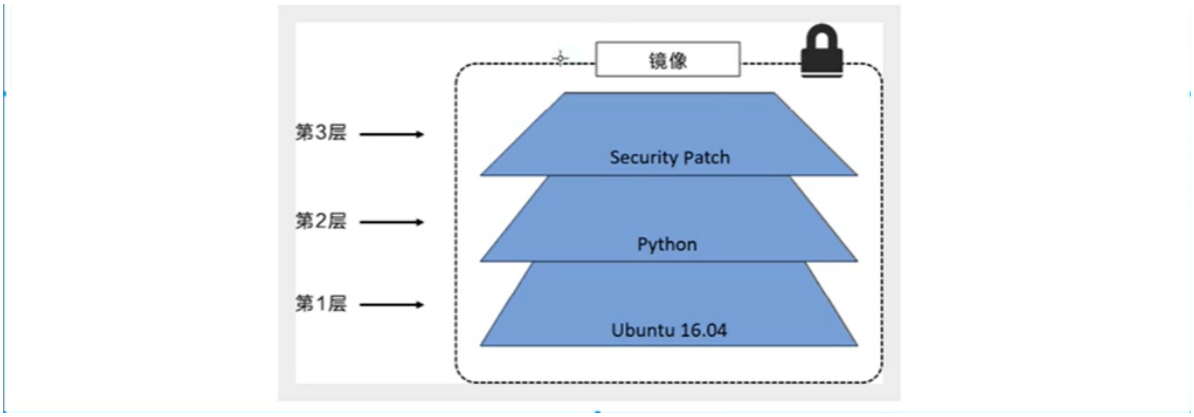
```
[root@vultr ~]# docker image inspect redis:latest
[
    //.........
        "RootFS": {
            "Type": "layers",
            "Layers": [

 "sha256:ffc9b21953f4cd7956cdf532a5db04ff0a2daa7475ad796f1bad58cfbaf77a07",

 "sha256:d4e681f320297add0ede0554524eb9106d8c3eb3a43e6e99d79db6f76f020248",

 "sha256:59bd5a888296b623ae5a9efc8f18285c8ac1a8662e5d3775a0d2d736c66ba825",

 "sha256:abef44452659f23ec349153a796bb160cefa667cb8d6d16d064fa9a0ab7f1dbb",

 "sha256:2f8fcc565367faa65192da55ce7c3ae8d73b92d69b5c0b0dbd5bd01215a11ae0",

 "sha256:5e107edf3216c060540ca8d2703a4c6fa836a6a9589033da9364bea8e0ea5a2a"
            ]
        },
        "Metadata": {
            "LastTagTime": "0001-01-01T00:00:00Z"
        }
    }
]
```

理解：

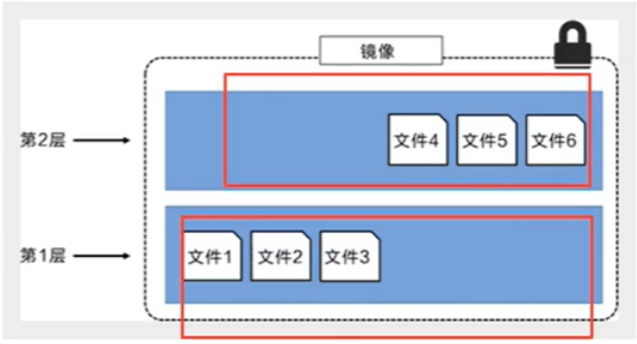
所有的Docker镜像都起始于一个基础镜像层，当进行修改或增加新的内容时，就会在当前镜像层之上，创建新的镜像层。

举一个简单的例子：假如基于Ubuntu Linux 16.04创建一个新的镜像，这就是新镜像层的第一层；如果在该镜像层中添加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）

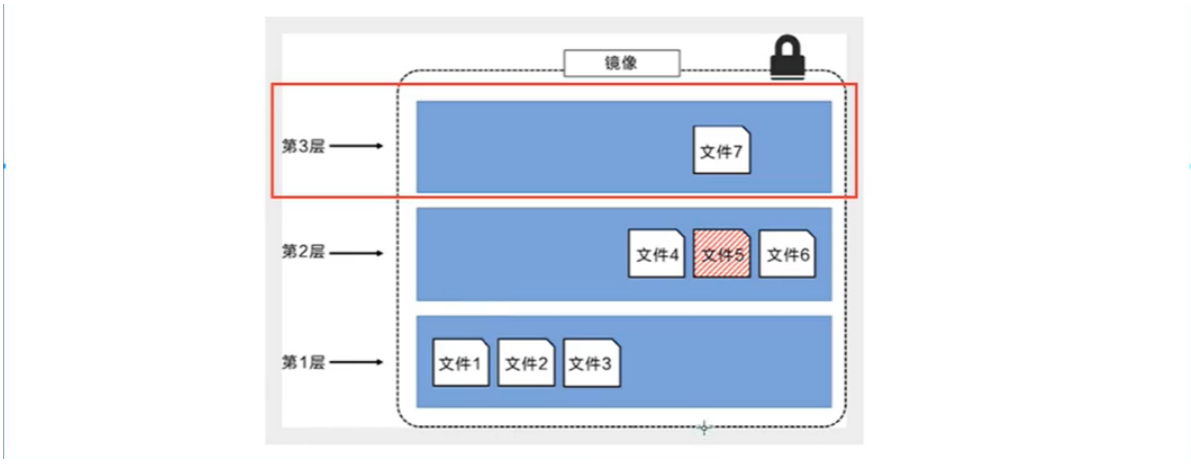


在添加额外的镜像层的同时，镜像始终保持是当前所有镜像的组合，理解这一点非常重要，下图中举了一个简单的例子，每个镜像层包含3个文件，而镜像包含了来自两个镜像层的六个文件。



上图中的镜像层跟之前图中的略有区别，主要目的便于展示文件

下图中展示了一个稍微复咋的三层镜像，在外部来看整个镜像只有六个文件，这是因为最上层中的文件7是文件5的一个更新版本。



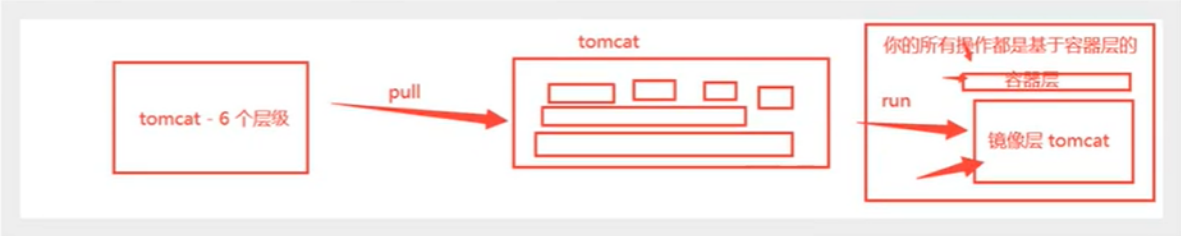
这种情况下，上层镜像层中的文件覆盖了底层镜像中的文件，这样就使得文件的更新版本作为一个新镜像层添加到镜像中。

Docker i通过存储引摩（新版本采用快照机制）的方式来实现像层堆栈，并保证多镜像层对外展示为统一的文件系统
Linux上可用的存储引摩有AUFS、 Overlay2、 Device Mapper,、Btrfs以及ZFS。顾名思义，每种存储引都基于 Linux中对应的文件系统或者块设备技术，并且每种存储引摩都有具独有的性能特点
在 Windows上仅支持 windowsfilter-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。



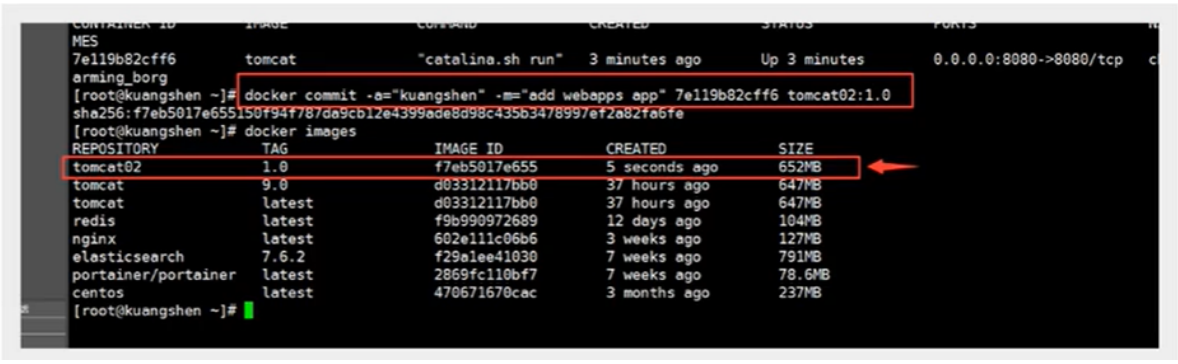
如何提交一个自己的镜像

commit镜像

```
docker commit 提交容器成为一个新的副本
# 命令和git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下面没有文件的

# 3自己拷贝进去基本的文件

# 4将修改后的tomcat提交上去
```



学习方式：理解概念，但是一定要实践，最后实践和理论知识相结合。

如果你想要保存当前容器的状态，就可以通过commit来提交，获得一个镜像，就好比我们之前学习VM时候的快照。

==========恭喜你，docker入门了==========

6 容器数据卷

6.1 什么是容器数据卷

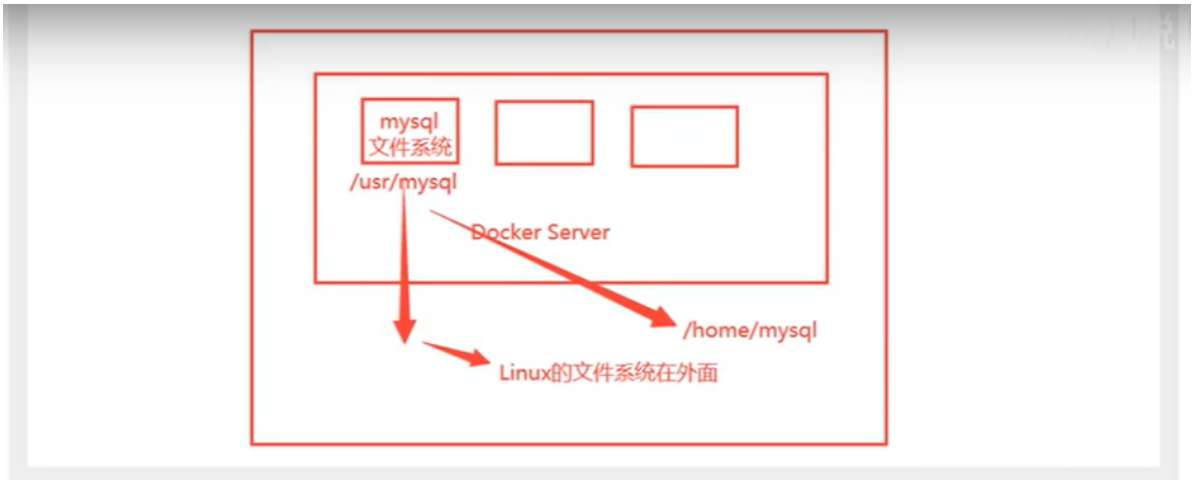
docker 理念回顾

将应用和环境打包成一个镜像

如果把数据放在容器中，删除容器后数据就没了！需求：数据可以持久化

Mysql ，容器删了就好比删库跑路，因此需要MySQL数据可以存在本地。

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！将容器内的目录，挂载到Linux上



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！

6.2 使用数据卷

方式1： 直接使用命令来挂载 -v

```
docker run -it -v 主机目录:容器目录

# 测试
[root@vultr ~]# docker run -it -v /home/ceshi:/home centos /bin/bash
[root@vultr home]# docker ps -a

# 启动起来时候我们可以通过docker inspect 查看挂载信息
[root@vultr home]# docker inspect 982e78fd5927
```

重点： 不管容器是否运行，都能实现同步。

使用容器卷的好处：我们以后修改只需要在本地修改即可，容器内自动同步。

6.3 实战：安装Mysql

思考：Mysql的数据持久化问题。

```
# 获取镜像
docker pull mysql:5.7
# 运行容器，需要挂载镜像,需要配置mysql的密码
# 解释
-d 后台运行
-p 端口映射
-v 数据卷挂载
-e 环境配置
--name  设置名字
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 （使用该命令 MySQL会被自动杀死）

(以下命令MySQL不会被自动杀死)
docker run -d --restart=always --name mysql01 -p 3310:3306 -e
MYSQL_ROOT_PASSWORD=123456 mysql:5.7

# 启动之后测试连接
```

具名挂载和匿名挂载

```
# 匿名挂载
-v 容器内路径
docker run -d -P --name nginx01 -v /etc/nginx nginx

# 查看所有volume的情况
docker volume ls

# 具名挂载(命令中 juming-nginx代表卷名)
docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx nginx

# 查看卷的具体位置
docker volume inspect juming-nginx
```

```
[root@vultr conf]# docker run -d -P --name nginx01 -v /etc/nginx nginx
be0e799728e71fa2362a0e29ecf36a84afcfaffe44955828d68955ecc1940f9c
[root@vultr conf]# docker volume ls
DRIVER              VOLUME NAME
local               8ea53ddaae0baccdfb144f16b7f248257122a7fcb8b952ba30731f2dc6f01201
local               545ace2adb51944383671181e9c573f45155bb9c1413157e272da9627aae2a36
[root@vultr conf]#
```

匿名卷

```
[root@vultr conf]# docker run -d -P --name nginx01 -v /etc/nginx nginx
be0e799728e71fa2362a0e29ecf36a84afcfaffe44955828d68955ecc1940f9c
[root@vultr conf]# docker volume ls
DRIVER              VOLUME NAME
local               8ea53ddaae0baccdfb144f16b7f248257122a7fcb8b952ba30731f2dc6f01201
local               545ace2adb51944383671181e9c573f45155bb9c1413157e272da9627aae2a36
[root@vultr conf]# docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx nginx
69e63d7657c79bb971263f32bdb318d7b3b3df7489153341f641d93257e73a00
[root@vultr conf]# docker volume ls
DRIVER              VOLUME NAME
local               8ea53ddaae0baccdfb144f16b7f248257122a7fcb8b952ba30731f2dc6f01201
local               545ace2adb51944383671181e9c573f45155bb9c1413157e272da9627aae2a36
local               juming-nginx
[root@vultr conf]#
```

匿名挂载

具名挂载

```
[root@vultr conf]# docker volume inspect juming-nginx
[
    {
        "CreatedAt": "2020-05-28T19:21:13+08:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/juming-nginx/_data",
        "Name": "juming-nginx",
        "Options": null,
        "Scope": "local"
    }
]
[root@vultr conf]#
```

挂载的具体位置

所有的docker容器内部卷，没有指定目录的情况下都会挂载在

`/var/lib/docker/volumes/xxxx/_data`

我们通过具名挂载可以方便的找到我们的一个卷，大多数情况下使用**具名挂载**

```
[root@vultr conf]# cd /var/lib/docker/
[root@vultr docker]# ll
total 48
drwx------.  2 root root 4096 May 27 16:22 builder
drwx--x--x.  4 root root 4096 May 27 16:22 buildkit
drwx------. 10 root root 4096 May 28 19:21 containers
drwx------.  3 root root 4096 May 27 16:22 image
drwxr-x---.  3 root root 4096 May 27 16:22 network
drwx------. 46 root root 4096 May 28 19:21 overlay2
drwx------.  4 root root 4096 May 27 16:22 plugins
drwx------.  2 root root 4096 May 27 16:22 runtimes
drwx------.  2 root root 4096 May 27 16:22 swarm
drwx------.  2 root root 4096 May 28 19:01 tmp
drwx------.  2 root root 4096 May 27 16:22 trust
drwx------.  5 root root 4096 May 28 19:21 volumes
[root@vultr docker]# cd volumes
[root@vultr volumes]# ll
total 36
drwxr-xr-x. 3 root root  4096 May 28 19:14 545ace2adb51944383671181e9c573f45155bb9c1413157e2
aae2a36
drwxr-xr-x. 3 root root  4096 May 28 12:46 8ea53ddaae0baccdfb144f16b7f248257122a7fcb8b952ba3
6f01201
drwxr-xr-x. 3 root root  4096 May 28 19:21 juming-nginx
-rw-------. 1 root root 32768 May 28 19:21 metadata.db
[root@vultr volumes]#
```

docker所有相关内容

挂载的卷

判断具名挂载和匿名挂载

```
-v 容器内路径               # 匿名挂载
-v 卷名:容器内路径          # 具名挂载
-v  /宿主机路径:容器内路径    # 指定路径挂载
```

拓展

```
# 通过 -v 容器内路径：ro rw 改变读写权限
ro  read only
rw  read and write

# 一旦设定了容器权限，容器对我们挂载出来的内容就有限定了。
docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx:ro nginx
docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx:rw nginx

# ro 只要看到 ro 就说明这个路径只能通过宿主机操作，容器内部无法操作。
```

7 DockerFile

Dockerfile 就是用来构建docker 镜像的构建文件！命令脚本！

通过这个脚本来生成一个镜像，镜像是一层一层的，脚本就是一个个命令，每个命令都是一层一层的！

```
# 编写dockerfile 文件 该文件位于 /home/docker-volume/dockerfile
# 以下为dockerfile内容
FROM centos

VOLUME ["volume01","volume02"]

CMD echo "----end----"

CMD /bin/bash

# 这里的每个命令，就是镜像的一层。
```

构建Docker镜像

```
[root@vultr docker-volume]# docker build -f dockfile -t cxk/centos:1.0 .
unable to prepare context: unable to evaluate symlinks in Dockerfile path: lstat
/home/docker-volume/dockfile: no such file or directory
[root@vultr docker-volume]# docker build -f dockerfile -t cxk/centos:1.0 .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM centos
 ---> 470671670cac
Step 2/4 : VOLUME ["volume01","volume02"]
 ---> Running in b79650272c13
Removing intermediate container b79650272c13
 ---> 2601708045c7
Step 3/4 : CMD echo "----end----"
 ---> Running in 76821870cc40
Removing intermediate container 76821870cc40
 ---> 1d81382eda61
Step 4/4 : CMD /bin/bash
 ---> Running in 9e21e89897dd
Removing intermediate container 9e21e89897dd
 ---> d223fd456147
Successfully built d223fd456147
Successfully tagged cxk/centos:1.0
```

```
[root@vultr docker-volume]# docker images
REPOSITORY            TAG                 IMAGE ID            CREATED
SIZE
cxk/centos            1.0                 d223fd456147        2 minutes ago
  237MB
mysql                 latest              30f937e841c8        7 days ago
541MB
redis                 latest              987b78fc9e38        9 days ago
104MB
nginx                 latest              9beeba249f3e        12 days ago
  127MB
portainer/portainer   latest              2869fc110bf7        2 months ago
 78.6MB
centos                latest              470671670cac        4 months ago
237MB
[root@vultr docker-volume]#
```

```
[root@vultr docker-volume]# docker images
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
cxk/centos            1.0                 d223fd456147        2 minutes ago       237MB
mysql                 latest              30f937e841c8        7 days ago          541MB
redis                 latest              987b78fc9e38        9 days ago          104MB
nginx                 latest              9beeba249f3e        12 days ago         127MB
portainer/portainer   latest              2869fc110bf7        2 months ago        78.6MB
centos                latest              470671670cac        4 months ago        237MB
[root@vultr docker-volume]#
```

启动自己构建的容器

```
[root@vultr docker-volume]# docker images
REPOSITORY            TAG                 IMAGE ID            CREATED
SIZE
cxk/centos            1.0                 d223fd456147        2 minutes ago
  237MB
mysql                 latest              30f937e841c8        7 days ago
541MB
redis                 latest              987b78fc9e38        9 days ago
104MB
nginx                 latest              9beeba249f3e        12 days ago
  127MB
portainer/portainer   latest              2869fc110bf7        2 months ago
 78.6MB
centos                latest              470671670cac        4 months ago
237MB
[root@vultr docker-volume]# docker run -it d223fd456147 /bin/bash
[root@87a9b18e881d /]#
```

```
[root@vultr docker-volume]# docker images
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
cxk/centos            1.0                 d223fd456147        2 minutes ago       237MB
mysql                 latest              30f937e841c8        7 days ago          541MB
redis                 latest              987b78fc9e38        9 days ago          104MB
nginx                 latest              9beeba249f3e        12 days ago         127MB
portainer/portainer   latest              2869fc110bf7        2 months ago        78.6MB
centos                latest              470671670cac        4 months ago        237MB
[root@vultr docker-volume]# docker run -it d223fd456147 /bin/bash
[root@87a9b18e881d /]# ls
bin  etc   lib    lost+found  mnt  proc  run   srv  tmp  var      volume02
dev  home  lib64  media       opt  root  sbin  sys  usr  volume01
[root@87a9b18e881d /]#
```

这个卷和外部有一个同步的目录：

```
FROM centos

VOLUME ["volume01","volume02"]    匿名挂载，出现的应该是一串数

CMD echo "----end----"

CMD /bin/bash
```

查看卷的挂载路径

```
[root@vultr docker-volume]# docker inspect 87a9b18e881d
```

```
            "Name": "overlay2"
        },
        "Mounts": [
            {
                "Type": "volume",
                "Name": "d2a28499e6de8206ffd71d35e4159240d980913a15988731f843134811487d85",
                "Source": "/var/lib/docker/volumes/d2a28499e6de8206ffd71d35e4159240d980913a1
843134811487d85/_data",
                "Destination": "volume01",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            },
            {
                "Type": "volume",
                "Name": "733fd04720a3142b565aaf39e3fb619d015174de9a804f2505332217d050ce40",
                "Source": "/var/lib/docker/volumes/733fd04720a3142b565aaf39e3fb619d015174de9
5332217d050ce40/_data",
                "Destination": "volume02",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
```

这种挂载方式使用的比较多，因为我们通常会构建自己的镜像！

加入构建镜像时没有挂载卷，要使用-v 参数进行手动挂载。

数据卷容器

多个CentOS之间同步数据


```
# 启动3个容器，通过我们刚才自己写的镜像启动
docker run -it --name docker01 cxk/centos:1.0
docker run -it --name docker02 --volumes-from cxk/centos:1.0
docker run -it --name docker03 --volumes-from cxk/centos:1.0
```

注意： 数据共享只要有一个容器还在使用数据，该数据就不会丢失。

多个MySQL之间同步数据

```
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

docker run -d -p 3311:3306 -e MYSQL_ROOT_PASSWORD=123456 --name mysql02
mysql:5.7
```

结论:

容器之间配置信息的传递，数据卷的生命周期一直持续到没有人使用为止。

但是一旦持久化到了本地，这个时候，本地数据是不会删除的。

DockerFile介绍

dockerfile是用来构建docker镜像的文件！命令参数脚本！

构建步骤：

1、编写一个dockerfile文件

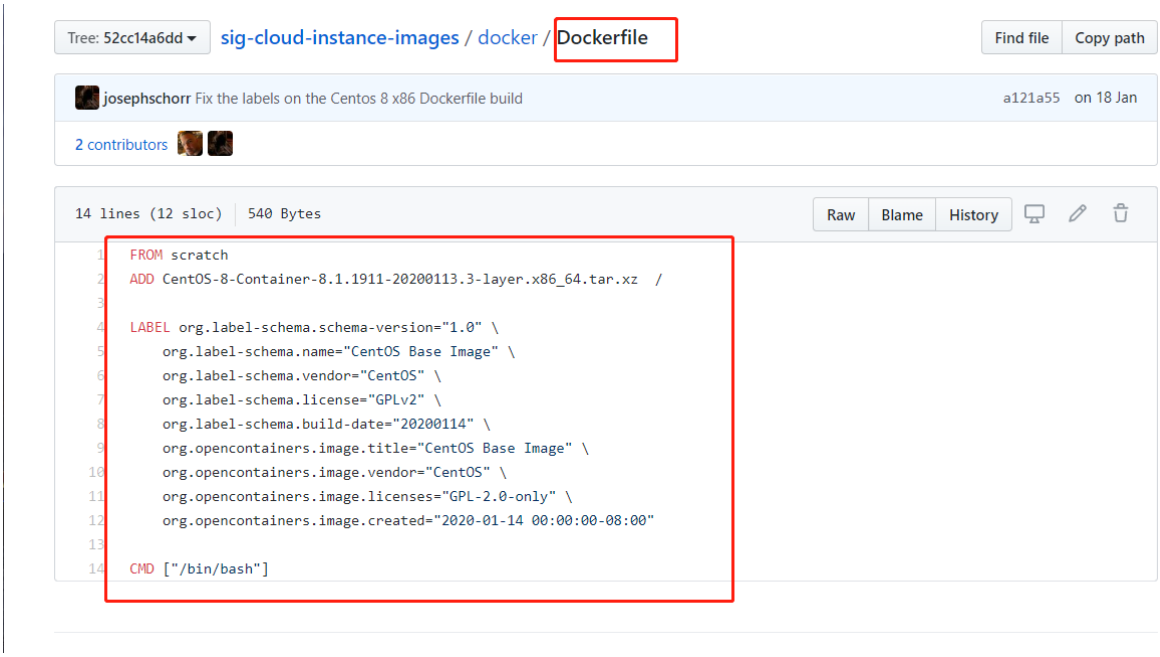
2、docker build 构建成为一个镜像

3、docker run 运行镜像

4、docker push 发布镜像（DockerHub、阿里云镜像）

官方centos8 Dockerfile

Docker Hub 中99%的镜像从这个基础镜像过来的 `FROM searth`



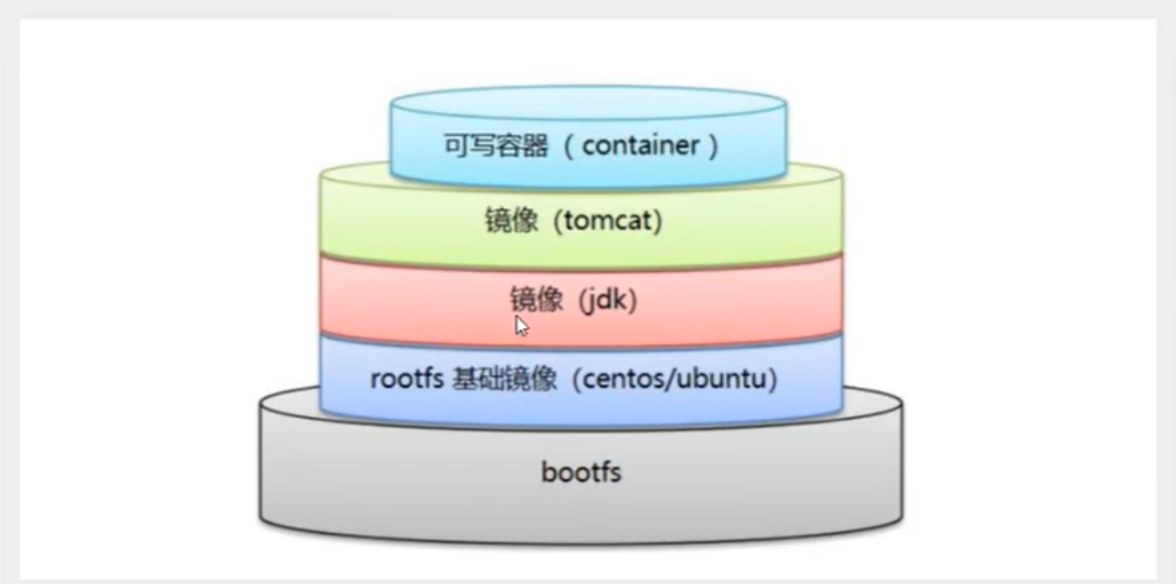
很多官方镜像都是基础包，很多功能没有，通常自己搭建自己的镜像。

官方既然可以制作镜像，我们也可以

DockerFile构建过程

基础知识：

- 每个保留关键字（指令）都是必须是大写字母
- 执行从上到下顺序执行
- "#"表示注释
- 每一个指令都会创建提交一个新的镜像层，并提交



dockerfile是面向提交的，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

Docker镜像逐渐成为企业交付的标准，必须要掌握

步骤：开发，部署，运维。。。。缺一不可！

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的。

DockerFile的指令

```
FROM            # 基础镜像，一切从这里构建
MAINTAINER      # 镜像是谁写的，留姓名+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压缩包就是我们要添加进去的内容
WORKDIR         # 镜像的工作目录
VOLUME          # 挂载的目录
EXPOSE          # 暴露端口，此时不暴露，需要在运行的时候使用-p参数暴露
CMD             # 指定这个容器启动时要运行的命令，只有最后一个会生效，可被替代
ENTRYPOINT      # 指定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。
COPY            # 类似ADD，将文件拷贝到镜像中
ENV             # 构建的时候设置环境变量。
```

以前我们使用别人的，现在我们知道这些指令后，练习自己写一个DockerFile

实战DockerFile测试

创建一个自己的centos,原版的没有 ifconfig 和vim。自己编写的添加进去。

- 编写DockerFile文件

```
[root@vultr dockerfile]# vim mydockerfile
[root@vultr dockerfile]# cat mydockerfile
FROM centos
MAINTAINER cxk<15230034878@163.com>

ENV MYPATH /user/local
WORKDIR $MYPATH
RUN yum -y install vim
RUN yum -y install net-tools

EXPOSE 80

CMD echo $MYPATH
CMD echo "-------end------"
CMD /bin/bash
[root@vultr dockerfile]#
```

编写的DockerFile

- 通过编写的DockerFile构建镜像

```
[root@vultr dockerfile]# docker build -f mydockerfile -t mycentos:0.1 .
Sending build context to Docker daemon  2.048kB
Step 1/10 : FROM centos
 ---> 470671670cac
Step 2/10 : MAINTAINER cxk<15230034878@163.com>
 ---> Running in 7a1f11825cdf
Removing intermediate container 7a1f11825cdf
 ---> 8ce134390415
Step 3/10 : ENV MYPATH /user/local
 ---> Running in bd6a6741ed05
Removing intermediate container bd6a6741ed05
 ---> 81cfcbfe3ab6
Step 4/10 : WORKDIR $MYPATH
 ---> Running in 2a2f67e85058
Removing intermediate container 2a2f67e85058
 ---> 9a4cd6bb72a5
Step 5/10 : RUN yum -y install vim
 ---> Running in 20fe1806161b
CentOS-8 - AppStream                            2.2 MB/s | 7.0 MB     00:03

CentOS-8 - Base                                 1.7 MB/s | 2.2 MB     00:01

CentOS-8 - Extras                               8.2 kB/s | 6.5 kB     00:00

Dependencies resolved.
================================================================================
====
 Package             Arch        Version                   Repository
Size
================================================================================
====
Installing:
 vim-enhanced        x86_64      2:8.0.1763-13.el8         AppStream
 1.4 M
Installing dependencies:
 gpm-libs            x86_64      1.20.7-15.el8             AppStream
39 k
 vim-common          x86_64      2:8.0.1763-13.el8         AppStream
 6.3 M
 vim-filesystem      noarch      2:8.0.1763-13.el8         AppStream
48 k
```

```
 which               x86_64      2.21-10.el8               BaseOS
 49 k

Transaction Summary
================================================================================
====
Install  5 Packages

Total download size: 7.8 M
Installed size: 31 M
Downloading Packages:
(1/5): gpm-libs-1.20.7-15.el8.x86_64.rpm        226 kB/s |  39 kB     00:00

(2/5): vim-filesystem-8.0.1763-13.el8.noarch.rp 610 kB/s |  48 kB     00:00

(3/5): which-2.21-10.el8.x86_64.rpm             289 kB/s |  49 kB     00:00

(4/5): vim-enhanced-8.0.1763-13.el8.x86_64.rpm  2.4 MB/s | 1.4 MB     00:00

(5/5): vim-common-8.0.1763-13.el8.x86_64.rpm    6.1 MB/s | 6.3 MB     00:01

--------------------------------------------------------------------------------
----
Total                                           4.5 MB/s | 7.8 MB     00:01

warning: /var/cache/dnf/AppStream-02e86d1c976ab532/packages/gpm-libs-1.20.7-
15.el8.x86_64.rpm: Header V3 RSA/SHA256 Signature, key ID 8483c65d: NOKEY
CentOS-8 - AppStream                            1.6 MB/s | 1.6 kB     00:00

Importing GPG key 0x8483C65D:
 Userid     : "CentOS (CentOS Official Signing Key) <security@centos.org>"
 Fingerprint: 99DB 70FA E1D7 CE22 7FB6 4882 05B5 55B3 8483 C65D
 From       : /etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial
Key imported successfully
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
 1/1
  Installing       : which-2.21-10.el8.x86_64
1/5
  Installing       : vim-filesystem-2:8.0.1763-13.el8.noarch
 2/5
  Installing       : vim-common-2:8.0.1763-13.el8.x86_64
 3/5
  Installing       : gpm-libs-1.20.7-15.el8.x86_64
 4/5
  Running scriptlet: gpm-libs-1.20.7-15.el8.x86_64
 4/5
  Installing       : vim-enhanced-2:8.0.1763-13.el8.x86_64
 5/5
  Running scriptlet: vim-enhanced-2:8.0.1763-13.el8.x86_64
 5/5
  Running scriptlet: vim-common-2:8.0.1763-13.el8.x86_64
 5/5
```

```
  Verifying        : gpm-libs-1.20.7-15.el8.x86_64
 1/5
  Verifying        : vim-common-2:8.0.1763-13.el8.x86_64
 2/5
  Verifying        : vim-enhanced-2:8.0.1763-13.el8.x86_64
 3/5
  Verifying        : vim-filesystem-2:8.0.1763-13.el8.noarch
 4/5
  Verifying        : which-2.21-10.el8.x86_64
5/5

Installed:
  vim-enhanced-2:8.0.1763-13.el8.x86_64 gpm-libs-1.20.7-15.el8.x86_64

  vim-common-2:8.0.1763-13.el8.x86_64   vim-filesystem-2:8.0.1763-
13.el8.noarch
  which-2.21-10.el8.x86_64

Complete!
Removing intermediate container 20fe1806161b
 ---> e52093302c6d
Step 6/10 : RUN yum -y install net-tools
 ---> Running in 1cd9e2028c0b
Last metadata expiration check: 0:00:08 ago on Fri May 29 02:38:59 2020.
Dependencies resolved.
================================================================================
====
 Package         Architecture Version                       Repository
Size
================================================================================
====
Installing:
 net-tools       x86_64       2.0-0.51.20160912git.el8      BaseOS
323 k

Transaction Summary
================================================================================
====
Install  1 Package

Total download size: 323 k
Installed size: 1.0 M
Downloading Packages:
net-tools-2.0-0.51.20160912git.el8.x86_64.rpm   838 kB/s | 323 kB     00:00

--------------------------------------------------------------------------------
----
Total                                           283 kB/s | 323 kB     00:01

Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
 1/1
  Installing       : net-tools-2.0-0.51.20160912git.el8.x86_64
 1/1
```

```
  Running scriptlet: net-tools-2.0-0.51.20160912git.el8.x86_64
 1/1
  Verifying        : net-tools-2.0-0.51.20160912git.el8.x86_64
 1/1

Installed:
  net-tools-2.0-0.51.20160912git.el8.x86_64


Complete!
Removing intermediate container 1cd9e2028c0b
 ---> 30e2a1a95ad7
Step 7/10 : EXPOSE 80
 ---> Running in 837c3a740662
Removing intermediate container 837c3a740662
 ---> af9dacb35350
Step 8/10 : CMD echo $MYPATH
 ---> Running in 0a4d08357260
Removing intermediate container 0a4d08357260
 ---> c0e5402041a6
Step 9/10 : CMD echo "-------end------"
 ---> Running in 84424d2e1c2c
Removing intermediate container 84424d2e1c2c
 ---> 6398ec87f25f
Step 10/10 : CMD /bin/bash
 ---> Running in c1af3629104d
Removing intermediate container c1af3629104d
 ---> ae983b27f787
Successfully built ae983b27f787
Successfully tagged mycentos:0.1
```

以上是整个构建流程，可以看到构建成功。ID号为ae983b27f787

注意： 本地有的镜像，直接会被复用。没有的才会下载。同时看到下载了vim和net-tools。

测试运行自己构建的centos

```
[root@vultr dockerfile]# docker images
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
mycentos              0.1                 ae983b27f787        10 minutes ago      321MB
cxk/centos            1.0                 d223fd456147        15 hours ago        237MB
mysql                 latest              30f937e841c8        7 days ago          541MB
redis                 latest              987b78fc9e38        10 days ago         104MB
nginx                 latest              9beeba249f3e        13 days ago         127MB
portainer/portainer   latest              2869fc110bf7        2 months ago        78.6MB
centos                latest              470671670cac        4 months ago        237MB
[root@vultr dockerfile]# docker run -it mysentos:0.1
Unable to find image 'mysentos:0.1' locally
docker: Error response from daemon: pull access denied for mysentos, repository does not exist or ma
y require 'docker login': denied: requested access to the resource is denied.
See 'docker run --help'.
[root@vultr dockerfile]# docker run -it mycentos:0.1
[root@ff295a273cc6 local]# pwd
/user/local
[root@ff295a273cc6 local]#
```

查看docker镜像构建的历史

```
docker history 镜像ID
```

CMD和ENTRYPOINT区别

```
CMD       # 指定这个容器启动的时候要运行的命令，只有最后一个生效，可被替代
ENTRYPOINT  # 指定这个容器启动的时候要运行的命令，可以追加命令
```

Dockerfile中很多命令都十分相似，我们需要了解他们的区别，最好的学习方式就是对比他们然后测试效果。

实战：Tomcat镜像

1.准备镜像文件tomcat压缩包，jdk的压缩包

2.编写dockerfile文件(官方命名DockerFile, build 会自动寻找这个文件，无需-f 指定)。

```
FROM centos
MAINTAINER cxk<15230034878@163.com>
COPY readme.txt /usr/local/readme.txt
ADD jdk-8u11-linux-x64.tar.gz /usr/local/
ADD apache-tomcat-9.0.22.tar.gz /usr/local/
RUN yum -y install vim
ENV MYPATH /usr/local
WORKDIR $MYPATH

ENV JAVA_HOME /usr/local/jdk1.8.0_11
ENV CLASSPATH $JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
ENV CATALINA_HOME /usr/local/apache-tomcat-9.0.22
ENV CATALINA_BASH /usr/local/apache-tomcat-9.0.22
ENV PATH
$PATH:$JAVA_HOME/bin:$CATALINA_HOME/lib:$CATALINA_BASH/lib:$CATALINA_BASH/bi
n

EXPOSE 8080

CMD /usr/local/apache-tomcat-9.0.22/bin/startup.sh &&tail -F
/usr/local/apache-tomcat-9.0.22/bin/logs/catalina.out
```

3. 构建镜像

```
# docker build 命令
docker build -t diytomcat .
```

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

把镜像发布到DockerHub

1.地址 https://www.docker.com/products/docker-hub 注册自己的账号

2.确定这个账号可以登陆

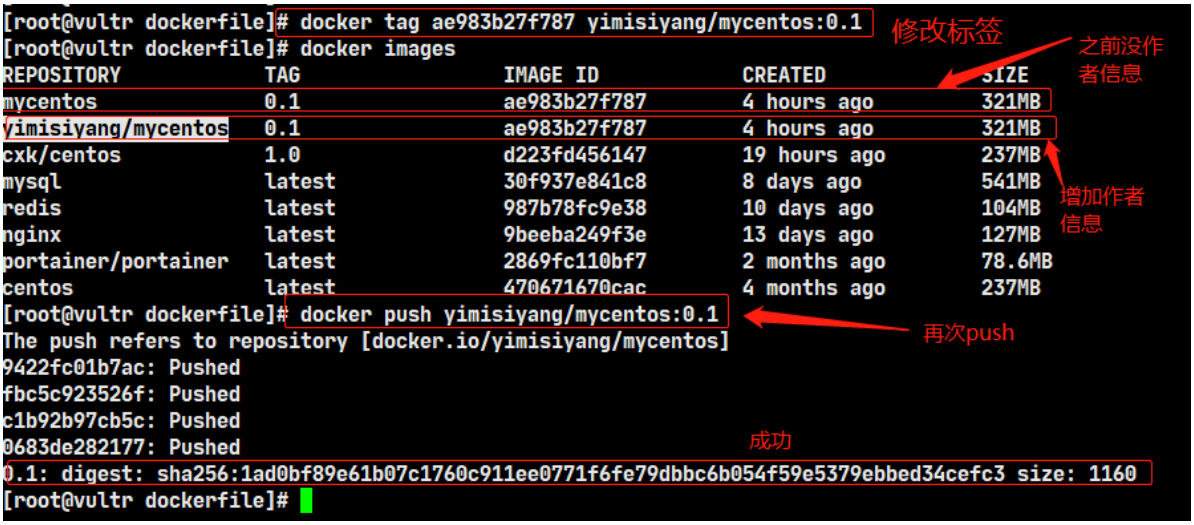
```
[root@vultr dockerfile]# docker login -u yimisiyang
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
```

3.在我们服务器上提交自己的镜像

```
[root@vultr dockerfile]# 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
```

4.登陆完成之后就可以提交镜像了，就是一步 docker push

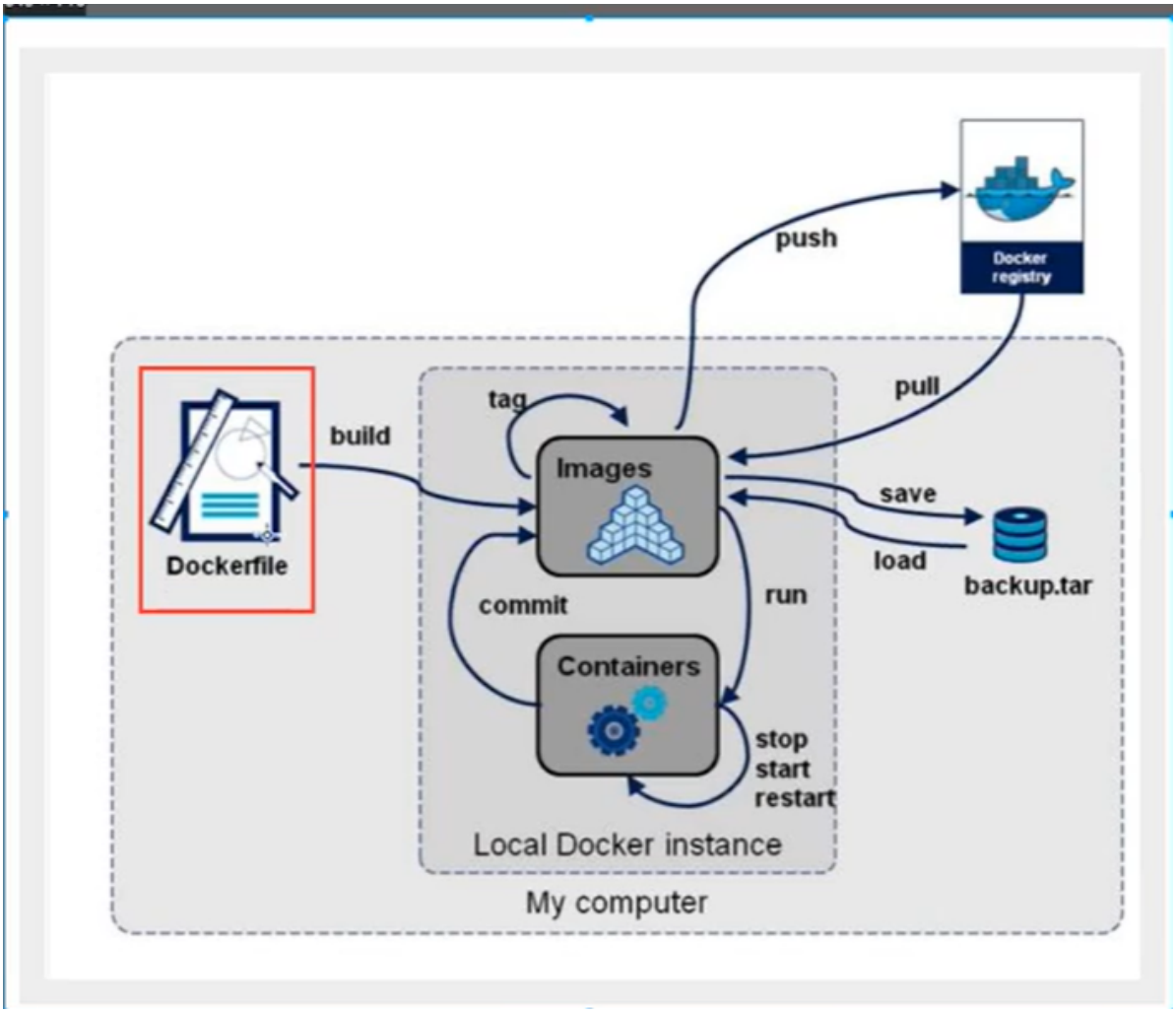
```
[root@vultr dockerfile]# docker push mycentos:0.1
The push refers to repository [docker.io/library/mycentos]
9422fc01b7ac: Preparing
fbc5c923526f: Preparing
c1b92b97cb5c: Preparing
0683de282177: Preparing
denied: requested access to the resource is denied  # 拒绝了

#  解决方法 ，给自己制作的镜像加一个标签
docker tag 镜像ID 作者信息/镜像信息:版本号

# 退出docker登陆(如果不再使用)
docker logout
```



把镜像发布到阿里云

阿里云可以参考官方文档。

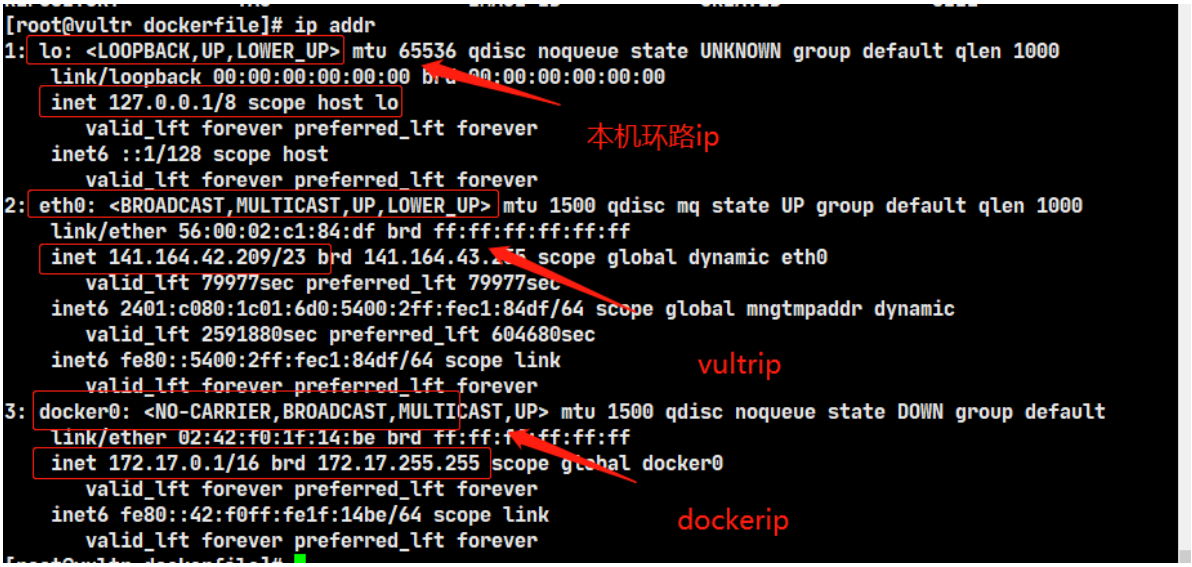
小结



8 Docker网络

理解Docker网络

学习之前删除所有的容器和镜像。



三个网络

问题1：Docker是如何处理容器网络访问的?


```
# 测试
[root@vultr dockerfile]# docker run -d -P --name tomcat01 tomcat
Unable to find image 'tomcat:latest' locally
latest: Pulling from library/tomcat
376057ac6fa1: Pull complete
5a63a0a859d8: Pull complete
496548a8c952: Pull complete
2adae3950d4d: Pull complete
0a297eafb9ac: Pull complete
09a4142c5c9d: Pull complete
9e78d9befa39: Pull complete
18f492f90b9c: Pull complete
7834493ec6cd: Pull complete
216b2be21722: Pull complete
Digest: sha256:ce753be7b61d86f877fe5065eb20c23491f783f283f25f6914ba769fee57886b
Status: Downloaded newer image for tomcat:latest
777f53ba782f6ca1b7a66f941b093e9d8fa0f46125cd5a31db2d2298bdcfd80f

# 查看ip
[root@vultr dockerfile]# 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
54: eth0@if55: <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能ping 通容器内部吗？ （能ping通）
[root@vultr dockerfile]# ping 127.17.0.2
PING 127.17.0.2 (127.17.0.2) 56(84) bytes of data.
64 bytes from 127.17.0.2: icmp_seq=1 ttl=64 time=0.079 ms
64 bytes from 127.17.0.2: icmp_seq=2 ttl=64 time=0.064 ms
64 bytes from 127.17.0.2: icmp_seq=3 ttl=64 time=0.100 ms
64 bytes from 127.17.0.2: icmp_seq=4 ttl=64 time=0.085 ms
64 bytes from 127.17.0.2: icmp_seq=5 ttl=64 time=0.068 ms
64 bytes from 127.17.0.2: icmp_seq=6 ttl=64 time=0.071 ms
64 bytes from 127.17.0.2: icmp_seq=7 ttl=64 time=0.070 ms
^C
--- 127.17.0.2 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6145ms
rtt min/avg/max/mdev = 0.064/0.076/0.100/0.015 ms
[root@vultr dockerfile]#

# 启动一个tomcat
[root@vultr dockerfile]# docker run -d -P --name tomcat02 tomcat
6a1881d8702af58ef82dedc2813a4f9bf14ca70a6eb0f107943bf0bfa017832a
[root@vultr dockerfile]# docker exec -it tomcat02 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
56: eth0@if57: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
[root@vultr dockerfile]#
```

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，只要安装了docker 就会有一个网卡docker0。桥接模式，使用的就是veth-pair技术。

2.再次测试ip addr，发现又增加了一对网卡。

启动第一个tomcat看到的网卡及IP

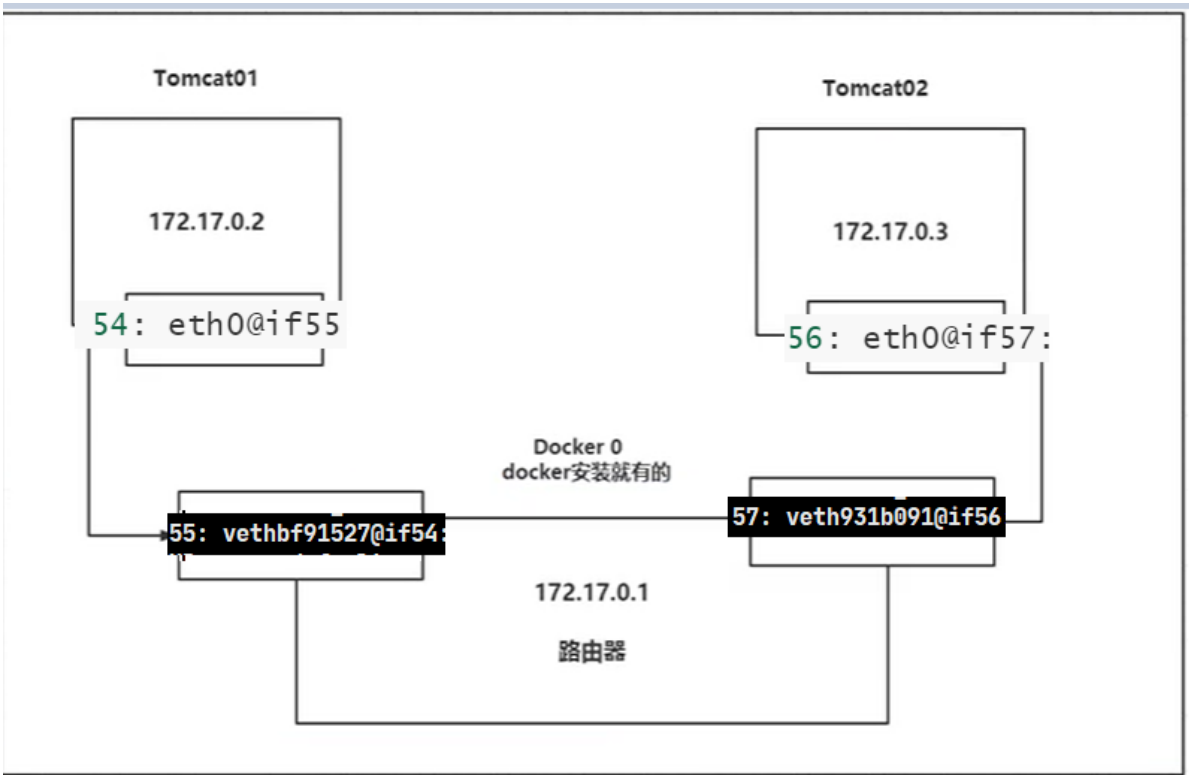
```
[root@vultr dockerfile]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 56:00:02:c1:84:df brd ff:ff:ff:ff:ff:ff
    inet 141.164.42.209/23 brd 141.164.43.255 scope global dynamic eth0
       valid_lft 79170sec preferred_lft 79170sec
    inet6 2401:c080:1c01:6d0:5400:2ff:fec1:84df/64 scope global mngtmpaddr dynamic
       valid_lft 2591642sec preferred_lft 604442sec
    inet6 fe80::5400:2ff:fec1:84df/64 scope link
       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:f0:1f:14:be 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:f0ff:fe1f:14be/64 scope link
       valid_lft forever preferred_lft forever
55: vethbf91527@if54: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state
UP group default
    link/ether ae:5e:02:23:f3:dd brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::ac5e:2ff:fe23:f3dd/64 scope link
       valid_lft forever preferred_lft forever
```

启动第二个tomcat看到的IP

```
[root@vultr dockerfile]# docker run -d -P --name tomcat02 tomcat
6a1881d8702af58ef82dedc2813a4f9bf14ca70a6eb0f107943bf0bfa017832a
[root@vultr dockerfile]# docker exec -it tomcat02 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
56: eth0@if57: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

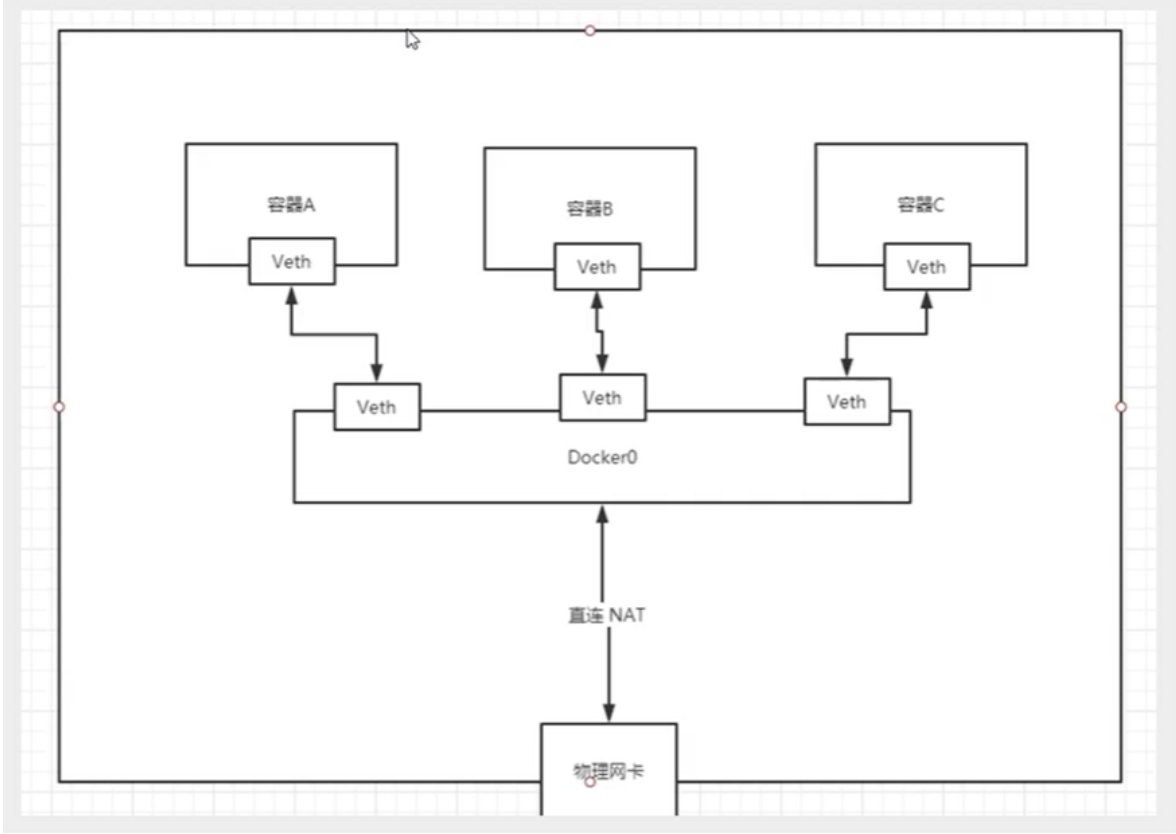
```
# 我们发现容器带来的网卡，都是一对一对的：图中的55 54是一对  56 57 是一对
# veth-pair 就是一对虚拟设备接口。他们都是成对出现的：一段连接着协议，一段彼此相连。
```

3.通过测试发现各个容器之间是可以ping通的，因为都桥接到了docker0这个网卡上。



结论：tomcat01和tomcat02是公用一个路由器，docker0

所有的容器不指定网卡的情况下，都是docker0路由的，docker会给我们的容器分配一个默认的可用IP



Docker中所有网络接口都是虚拟的，虚拟的转发效率高。

只要容器删除，对应网桥就没了。

--link

通过容器名字来实现容器之间的访问

```
# 未设置之前是ping不通的
```

```
[root@vultr dockerfile]# docker exec -it tomcat02 ping tomcat01
ping: tomcat01: Name or service not known
[root@vultr dockerfile]#

# 解决上述问题
[root@vultr dockerfile]# docker run -d -P --name tomcat03 --link tomcat02 tomcat
0d7082ee3de2ef2424ee0d06d17ada6ade0e7ac00d2362f3b4f50a6f1f4a1815
[root@vultr dockerfile]#

# 通过上述命令tomcat03就可以ping通tomcat02了
[root@vultr dockerfile]# docker exec -it tomcat03 ping tomcat02
PING tomcat02 (172.17.0.3) 56(84) bytes of data.
64 bytes from tomcat02 (172.17.0.3): icmp_seq=1 ttl=64 time=0.134 ms
64 bytes from tomcat02 (172.17.0.3): icmp_seq=2 ttl=64 time=0.091 ms
64 bytes from tomcat02 (172.17.0.3): icmp_seq=3 ttl=64 time=0.122 ms
64 bytes from tomcat02 (172.17.0.3): icmp_seq=4 ttl=64 time=0.083 ms
^C
--- tomcat02 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 102ms
rtt min/avg/max/mdev = 0.083/0.107/0.134/0.023 ms
[root@vultr dockerfile]#

# 反向ping不通 tomcat02 ping tomcat03
[root@vultr dockerfile]# docker exec -it tomcat02 ping tomcat03
ping: tomcat03: Name or service not known
[root@vultr dockerfile]#
```

出现上述问题的原因就是tomcat03配置了tomcat02的ip,而tomcat02并没有配合tomcat03的IP

```
[root@vultr dockerfile]# docker exec -it tomcat03 cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3      tomcat02 6a1881d8702a
172.17.0.4      0d7082ee3de2
[root@vultr dockerfile]#
```

```
[root@vultr dockerfile]# docker exec -it tomcat02 cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3      6a1881d8702a
[root@vultr dockerfile]#
```

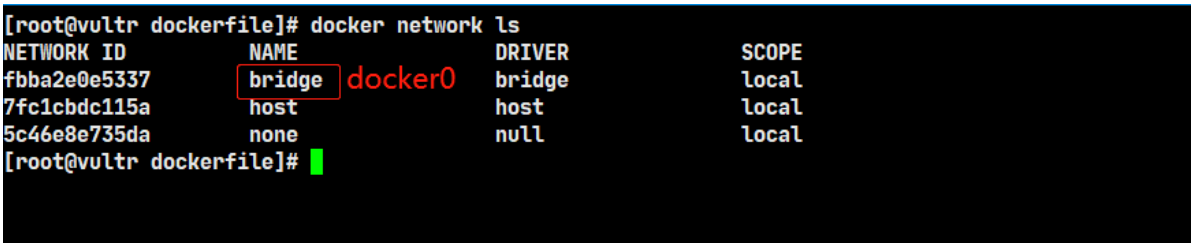
并没有绑定tomcat03ip

本质探究：--link就是再hosts配置了容器IP，该方法太笨了，已经不推荐使用了。

通常使用自定义网络解决上述问题，不使用docker0

自定义网络

查看所有的docker网络



网络模式：

bridge:桥接 docker(默认)

none:不配置网络

host: 和宿主机共享网络

container:容器间网络连接（用到比较少，局限大）

测试

```
#  测试之前删掉所有容器，保证干净，对新手友好
[root@vultr dockerfile]# docker rm -f $(docker ps -qa)
0d7082ee3de2
6a1881d8702a
777f53ba782f
[root@vultr dockerfile]#

# 查看IP
[root@vultr dockerfile]# 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: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group
default qlen 1000
    link/ether 56:00:02:c1:84:df brd ff:ff:ff:ff:ff:ff
    inet 141.164.42.209/23 brd 141.164.43.255 scope global dynamic eth0
       valid_lft 74611sec preferred_lft 74611sec
    inet6 2401:c080:1c01:6d0:5400:2ff:fec1:84df/64 scope global mngtmpaddr
dynamic
       valid_lft 2591805sec preferred_lft 604605sec
    inet6 fe80::5400:2ff:fec1:84df/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN group default
    link/ether 02:42:f0:1f:14:be 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:f0ff:fe1f:14be/64 scope link
       valid_lft forever preferred_lft forever
[root@vultr dockerfile]#

# 之前直接启动的命令 默认有  --net bridge
docker run -d -P --name tomcat01 --net bridge tomcat

# 下面进行自定义网络
```

```
[root@vultr dockerfile]# docker network create --driver bridge --subnet
192.168.0.0/16 --gateway 192.168.0.1 mynet
118f8c8cc7d00bc6ccc8ba6816dbfe0594871f55ff19a9f215aab89f8016ff3b
[root@vultr dockerfile]#

# 查看是否创建成功
[root@vultr dockerfile]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
fbba2e0e5337        bridge              bridge              local
7fc1cbdc115a        host                host                local
118f8c8cc7d0        mynet               bridge              local
5c46e8e735da        none                null                local
[root@vultr dockerfile]#
```

使用以下命令也可以查看

```
docker network inspect mynet
```

```
# 将tomcat放再自己创建的网络进行启动
[root@vultr dockerfile]# docker run -d -P --name tomcat01-mynet --net mynet
tomcat
49dcfccdd8b20a44a2a57eb21f0f1a87da2429faf487c1a3cac4d76e2a1bd3fa
[root@vultr dockerfile]# docker run -d -P --name tomcat02-mynet --net mynet
tomcat
93e6394eaa47ac7e2ce1847f871c104ac07912c3559ad261a8620d09af48d743
[root@vultr dockerfile]#

# 直接用名字进行互相ping
[root@vultr ~]# docker exec -it tomcat01-mynet ping tomcat02-mynet
PING tomcat02-mynet (192.168.0.3) 56(84) bytes of data.
64 bytes from tomcat02-mynet.mynet (192.168.0.3): icmp_seq=1 ttl=64 time=0.117
ms
64 bytes from tomcat02-mynet.mynet (192.168.0.3): icmp_seq=2 ttl=64 time=0.178
ms
64 bytes from tomcat02-mynet.mynet (192.168.0.3): icmp_seq=3 ttl=64 time=0.088
ms
64 bytes from tomcat02-mynet.mynet (192.168.0.3): icmp_seq=4 ttl=64 time=0.096
ms
^C
--- tomcat02-mynet ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 91ms
rtt min/avg/max/mdev = 0.088/0.119/0.178/0.037 ms
```

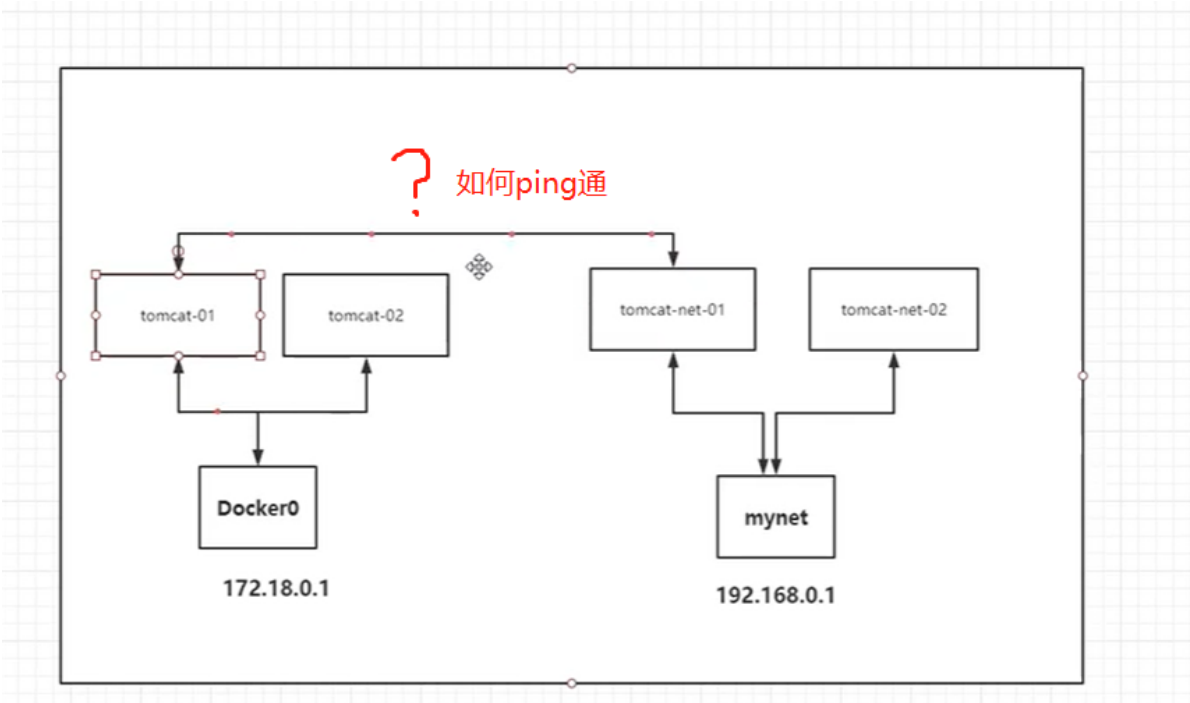
好处

直接通过ping名字的方式就可以实现连通

不同的集群使用不同的网络，保证集群是安全和健康的。

以上就是自定义网络的好处。

网络连通



connect 功能能够实现（实现一个容器两个IP）

```
[root@vultr ~]# docker network --help

Usage:  docker network COMMAND

Manage networks

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.
[root@vultr ~]#
```

连接一个容器到一个网络

结论：假设要跨网络操作别人，就需要使用connect连通。

实战：部署Redis集群

shell 脚本(redis.sh)

```
# 创建网卡
docker network create redis --subnet 172.38.0.0/16
echo "=========网卡创建成功========"
# 通过脚本创建6个redis并配置
for port in $(seq 1 6);
do
mkdir -p /mydata/redis/node-${port}/conf
touch /mydata/redis/node-${port}/conf/redis.conf
cat << EOF >/mydata/redis/node-${port}/conf/redis.conf
port 6379
bind 0.0.0.0
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
cluster-announce-ip 172.38.0.1${port}
cluster-announce-port 6379
```

```
appendonly yes
EOF
docker run -p 637${port}:6379 -p 1637${port}:16379 --name redis-${port} \
-v /mydata/redis/node-${port}/data:/data \
-v /mydata/redis/node-${port}/conf/redis.conf:/etc/redis/redis.conf \
-d --net redis --ip 172.38.0.1${port} redis:5.0.9-alpine3.11 redis-server
/etc/redis/redis.conf;
done
```

```
[root@vultr shell]# ./redis.sh
511b286e230a6a38a46ad6c53efee8fd15625d74ae397ccbab0fb234040d3217
82cc99d1cf295b94a9a96d89d8e050f784971df6f322f63c8e7e4500efcd9620
f0efd3a94e274c7285bd92443eaa1428be12032e9f2ed28f947a957424770ab0
d94afae3919d73cd5dc2930fb3f1893118255d77fa2a25ec7eab72156c7689bd
48abc37b00c4ddd5c0537a916a07bebae3f05fc163242915257b5893c940f965
2f44f21284a82d69983190a3e633309e61bb44dd198ec67659c59830f42aea39
[root@vultr shell]# docker exec -it redis-1 /bin/sh
/data # ls
appendonly.aof  nodes.conf
/data #
```

执行完上述脚本以后创建集群(该命令进入redis-* 进行操作)

```
# 创建集群（执行该命令中途需要输入yes）
redis-cli --cluster create 172.38.0.11:6379 172.38.0.12:6379 172.38.0.13:6379
172.38.0.14:6379 172.38.0.15:6379 172.38.0.16:6379 --cluster-replicas 1
```

```
/data # redis-cli --cluster create 172.38.0.11:6379 172.38.0.12:6379 172.38.0.13:6379 172.38.0.
79 172.38.0.15:6379 172.38.0.16:6379 --cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 172.38.0.15:6379 to 172.38.0.11:6379
Adding replica 172.38.0.16:6379 to 172.38.0.12:6379
Adding replica 172.38.0.14:6379 to 172.38.0.13:6379
M: 774df83b72ab0d182b3ff959121f47c2f2810cc6 172.38.0.11:6379
   slots:[0-5460] (5461 slots) master
M: a740673cd40b7e03607d07a06770a1cf1d5ef42d 172.38.0.12:6379
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
M: e1b8283b6a2c159c007a14539c2e5495811e2cfa 172.38.0.13:6379
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: b448797bac2e9edb80b6e8caef3b8076c83135b3 172.38.0.16:6379
   slots: (0 slots) slave
   replicates 774df83b72ab0d182b3ff959121f47c2f2810cc6
S: a24e6a43b636145ddcb3dc3333b72ab260dfff7b 172.38.0.14:6379
   slots: (0 slots) slave
   replicates e1b8283b6a2c159c007a14539c2e5495811e2cfa
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

从上图可以看出，该redis集群搭建成功。

测试集群

```
redis-cli -c
```

```
/data # redis-cli -c
127.0.0.1:6379> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:1
cluster_stats_messages_ping_sent:560
cluster_stats_messages_pong_sent:566
cluster_stats_messages_sent:1126
```