- Docker 概述
- Docker 安装
- Docker 命令
    - 镜像命令
    - 容器命令
    - 操作命令
    - 。。。
- Docker 镜像
- 容器数据卷
- DockerFile
- Docker 网络原理
- IDEA 整合 Docker（单机 Docker)
- Docker Compose
- Docker Swarm
- CI\CD Jenkins

Docker 概述

Docker 为什么出现?

一款产品： 开发–上线 两套环境！应用环境，应用配置！

开发 — 运维。 问题：我在我的电脑上可以允许！版本更新，导致服务不可用！对于运维来说考验十分大?

环境配置是十分的麻烦，每一个及其都要部署环境(集群 Redis、ES、Hadoop...) !费事费力。

发布一个项目(jar + (Redis MySQL JDK ES)),项目能不能带上环境安装打包！

之前在服务器配置一个应用的环境 Redis MySQL JDK ES Hadoop 配置超麻烦了，不能够跨平台。

开发环境 Windows，最后发布到 Linux！

传统：开发 jar，运维来做！

现在：开发打包部署上线，一套流程做完！

安卓流程：java — apk —发布（应用商店）— 张三使用 apk 一安装即可用！

docker 流程： java-jar（环境） — 打包项目带上环境（镜像） —（ Docker 仓库：商店）-----

Docker 给以上的问题，提出了解决方案！

Docker 的思想就来自于集装箱！

JRE – 多个应用(端口冲突) – 原来都是交叉的！

隔离：Docker 核心思想！打包装箱！每个箱子是互相隔离的。

Docker 通过隔离机制，可以将服务器利用到极致！

本质：所有的技术都是因为出现了一些问题，我们需要去解决，才去学习！

Docker 历史

2010 年，几个的年轻人，就在美国成立了一家公司 `dotcloud`

做一些 pass 的云计算服务！LXC（Linux Container 容器）有关的容器技术！

> Linux Container 容器是一种内核虚拟化技术，可以提供轻量级的虚拟化，以便隔离进程和资源。

他们将自己的技术（容器化技术）命名就是 Docker

Docker 刚刚延生的时候，没有引起行业的注意！`dotCloud`，就活不下去！

`开源`

2013 年，Docker 开源！

越来越多的人发现 docker 的优点！火了。Docker 每个月都会更新一个版本！

2014 年 4 月 9 日，Docker1.0 发布！

docker 为什么这么火？十分的轻巧！

在容器技术出来之前，我们都是使用虚拟机技术！

虚拟机：在 window 中装一个 VMware，通过这个软件我们可以虚拟出来一台或者多台电脑！笨重！

虚拟机也属于虚拟化技术，Docker 容器技术，也是一种虚拟化技术！

```
vm : linux centos 原生镜像（一个电脑！） 隔离、需要开启多个虚拟机！ 几个 G 几分钟
docker: 隔离，镜像（最核心的环境 4m + jdk + mysql）十分的小巧，运行镜像就可以了！小巧！ 几个 M 秒级启动！
```

> 聊聊 Docker

Docker 基于 Go 语言开发的！开源项目！

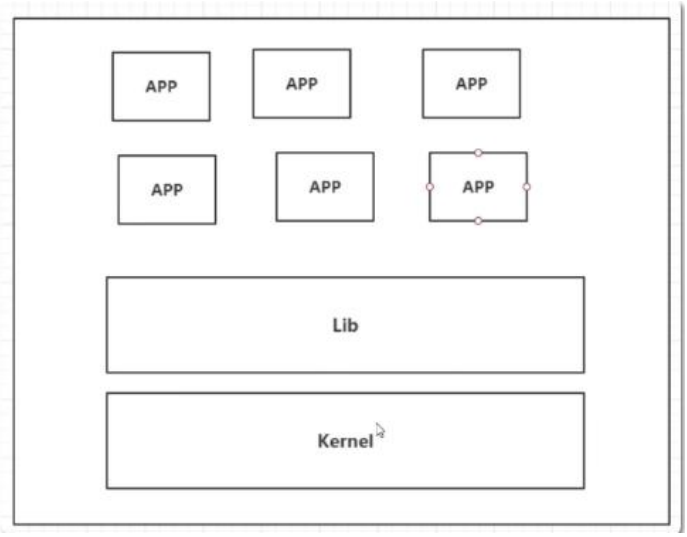
docker 官网：https://www.docker.com/

文档：https://docs.docker.com/ Docker 的文档是超级详细的！

仓库：https://hub.docker.com/

Docker 能干嘛

> 之前的虚拟机技术！



虚拟机技术缺点：

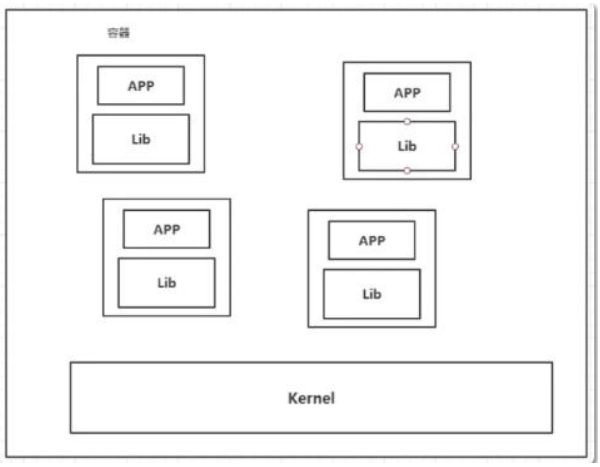
1、 资源占用十分多

2、 冗余步骤多

3、 启动很慢！

> 容器化技术

容器化技术不是模拟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，运行一个完整的操作系统，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容，容器是没有自己的内核的，也没有虚拟我们的硬件，所以就轻便了
- 每个容器间是互相隔离，每个容器内都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：一对帮助文档，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

项目打包为一个镜像，扩展服务器 A！服务器 B

更简单的系统运维

在容器化之后，我们的开发，测试环境都是高度一致的

更高效的计算资源利用

Docker 是内核级别的虚拟化，可以在一个物理机上可以运行很多的容器实例！服务器的性能可以被压榨到极致。

Docker 安装

Docker 的基本组成

镜像（image）：

docker 镜像就好比是一个目标，可以通过这个目标来创建容器服务，tomcat 镜像==>run==>容器（提供服务器），通过这个镜像可以创建多个容器（最终服务运行或者项目运行就是在容器中的）。

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就是一个简易的 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。(很类似 git)

Docker Hub 是国外的。

阿里云...都有容器服务器(配置镜像加速!)

安装 Docker

```
环境准备
```

Linux 要求内核 3.0 以上

```
➜  ~ uname -r
4.15.0-96-generic # 要求 3.0 以上
➜  ~ cat /etc/os-release
```

```
NAME="Ubuntu"
VERSION="18.04.4 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.4 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"VERSION
_CODENAME=bionic
UBUNTU_CODENAME=bionic
```

安装

帮助文档：https://docs.docker.com/engine/install/

```
#1.卸载旧版本
yum remove docker \
                docker-client \
                docker-client-latest \
                docker-common \
                docker-latest \
                docker-latest-logrotate \
                docker-logrotate \
                docker-engine
#2.需要的安装包
yum install -y yum-utils
#3.设置镜像的仓库
yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
#默认是从国外的，不推荐
#推荐使用国内的
yum-config-manager \
    --add-repo \
    https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
#更新 yum 软件包索引
yum makecache fast
#4.安装 docker 相关的 docker-ce 社区版 而 ee 是企业版
yum install docker-ce docker-ce-cli containerd.io
#5、启动 docker
docker systemctl start docker
#6. 使用 docker version 查看是否按照成功
```

```
docker version
#7. 测试
docker run hello-world
#7. 测试
➜  ~ docker run hello-world
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.查看一下下载的镜像
➜  ~ docker images
REPOSITORY           TAG                 IMAGE ID           CREATED            SIZE
hello-world          latest              bf756fb1ae65        4 months ago       13.3kB
```

了解：卸载 docker

```
#1. 卸载依赖
yum remove docker-ce docker-ce-cli containerd.io
#2. 删除资源
rm -rf /var/lib/docker
# /var/lib/docker 是 docker 的默认工作路径!
```

加速

https://www.cnblogs.com/nhdlb/p/12567154.html

创建或修改 **/etc/docker/daemon.json** 文件，修改为如下形式

```
{
  "registry-mirrors": [
    "https://registry.docker-cn.com",
    "http://hub-mirror.c.163.com",
    "https://docker.mirrors.ustc.edu.cn"
  ]
}
```

查看是否成功

```
$ docker info
```

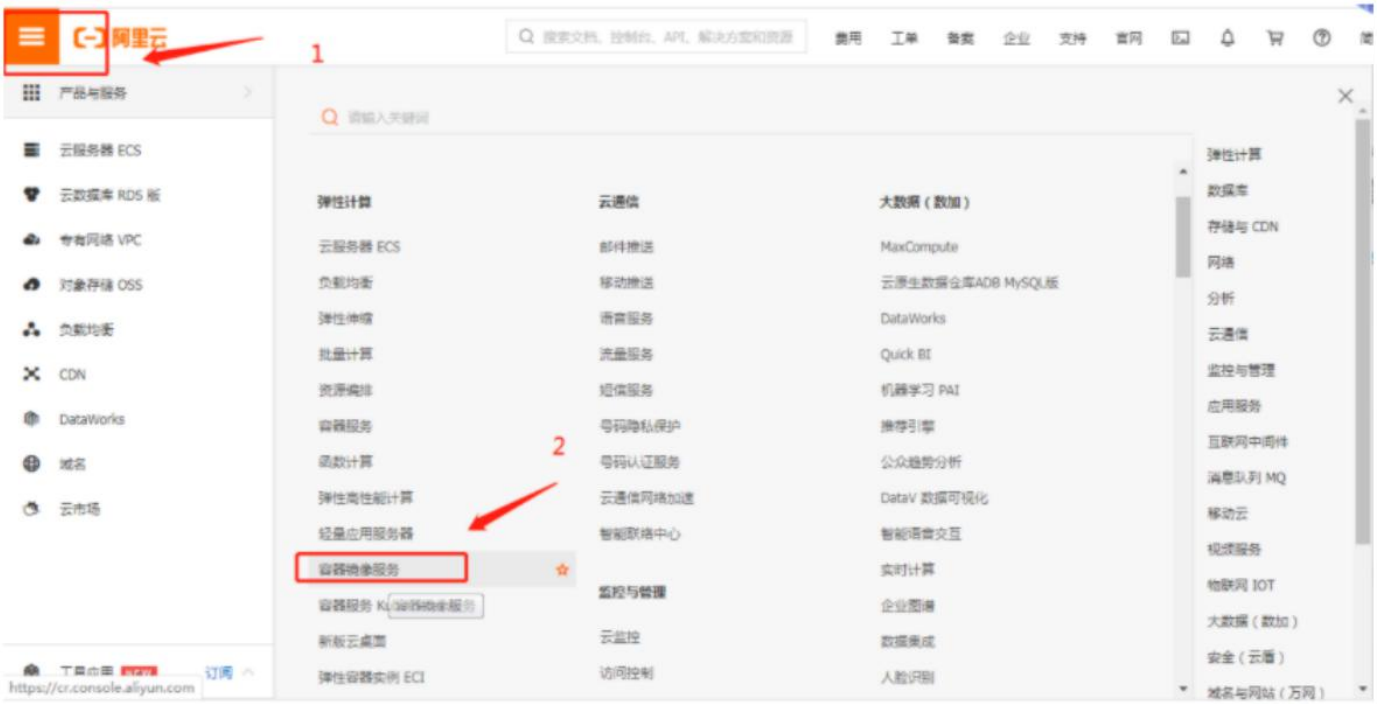
```
 Storage Driver: overlay2
  Backing Filesystem: xfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc version: dc9208a3303feef5b3839f4323d9beb36df0a9dd
 init version: fec3683
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 3.10.0-1062.12.1.el7.x86_64
 Operating System: CentOS Linux 7 (Core)
 OSType: linux
 Architecture: x86_64
 CPUs: 4
 Total Memory: 3.84GiB
 Name: localhost.localdomain
 ID: EASM:AFR3:YJDL:R6Z7:GG3H:IMVF:CSEL:CTAR:ZW2Y:6HZ3:3R6X:XHWT
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Registry Mirrors:
  http://hub-mirror.c.163.com/
 Live Restore Enabled: false

[root@localhost ~]#
```

阿里云镜像加速

1、登录阿里云找到容器服务


点击控制台
阿里云 企业应用中心
搜索
发布定制需求
文档
备案
控制台
登陆后，点击控制台
产品分类
企业建站
企业初创
多端小程序
解决方案
Teambition免费试用
开放云原生应用
云原生（Cloud Native）技术帮助企业在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。 构建并高效管理 符合云原生定义的应用，能帮助企业更快交付、降低未知风险、拓展业务边界。
联系我们
容器服务产品族
容器服务开源组件
文档
docker
镜像搜索

2、


阿里云
1
产品与服务
云服务器 ECS
云数据库 RDS 版
专有网络 VPC
对象存储 OSS
负载均衡
CDN
DataWorks
域名
云市场
弹性计算
云服务器 ECS
负载均衡
弹性伸缩
批量计算
资源编排
容器服务
函数计算
弹性高性能计算
轻量应用服务器
容器镜像服务
2
云通信
邮件推送
移动推送
语音服务
流量服务
短信服务
号码隐私保护
号码认证服务
云通信网络加速
智能联络中心
监控与管理
云监控
访问控制
大数据（数加）
MaxCompute
云原生数据仓库ADB MySQL版
DataWorks
Quick BI
机器学习 PAI
推荐引擎
公众趋势分析
DataV 数据可视化
智能语音交互
实时计算
企业图谱
数据集成
人脸识别
弹性计算
数据库
存储与 CDN
网络
分析
云通信
监控与管理
应用服务
互联网中间件
消息队列 MQ
移动云
视频服务
物联网 IOT
大数据（数加）
安全（云盾）
域名与网站（万网）
https://cr.console.aliyun.com

2、找到镜像加速器

1. 安装 / 升级Docker客户端

推荐安装 1.10.0 以上版本的Docker客户端，参考文档 docker-ce

2. 配置镜像加速器

针对Docker客户端版本大于 1.10.0 的用户

您可以通过修改daemon配置文件 /etc/docker/daemon.json 来使用加速器

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

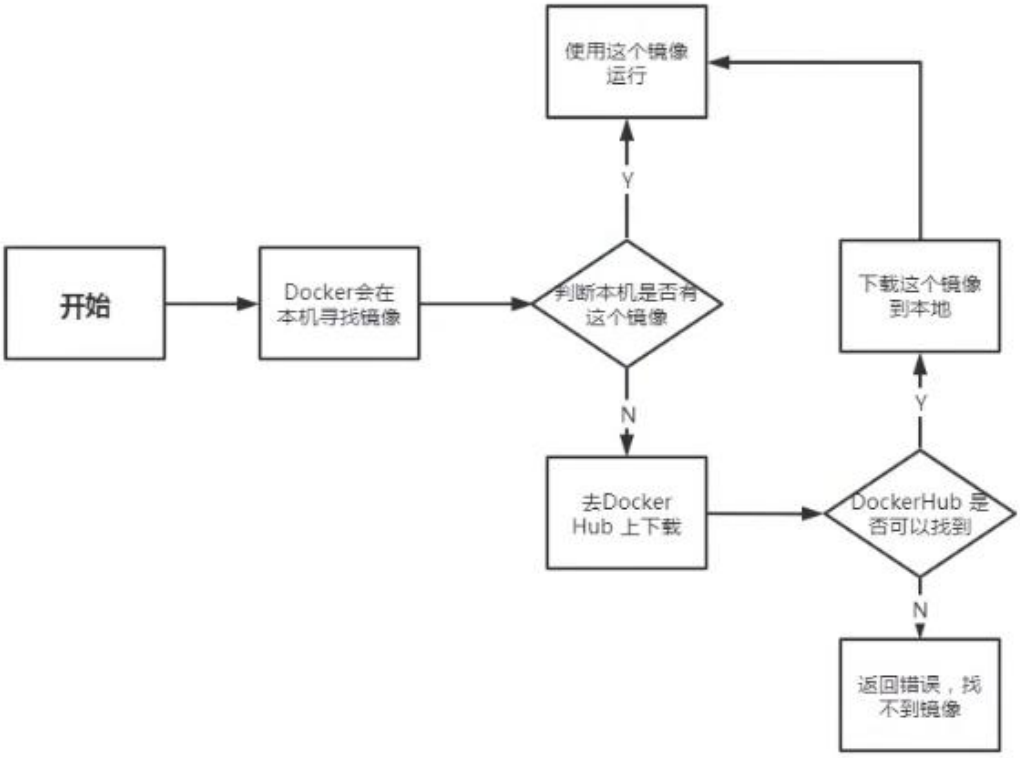
https://（...）.mirror.aliyuncs.com

3、配置使用

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://qiyb9988.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

回顾 HelloWorld 流程

docker run 流程图



底层原理

Docker **是怎么工作的**?

Docker 是一个 Client-Server 结构的系统，Docker 的守护进程运行在主机上。通过 Socket 从客户端访问!

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

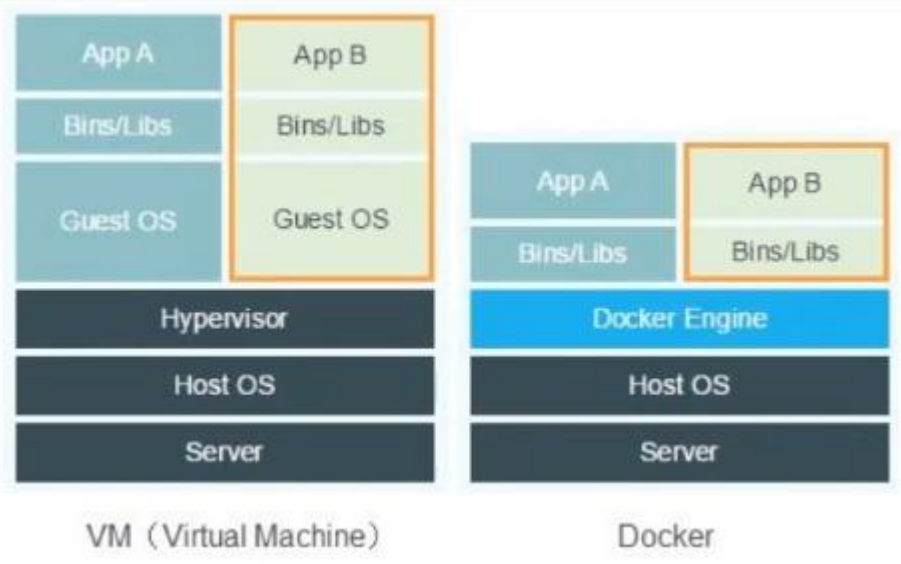
为什么 Docker 比 Vm 快

1、docker 有着比虚拟机更少的抽象层。由于 docker 不需要 Hypervisor 实现**硬件资源虚拟化**,运行在 docker 容器上的程序直接使用的都是实际物理机的硬件资源。因此在 CPU、内存利用率上 docker 将会在效率上有明显优势。

2、docker 利用的是宿主机的内核,而不需要 Guest OS。

```
GuestOS：  VM（虚拟机）里的的系统（OS）；
```

HostOS：物理机里的系统（OS）；



因此,当新建一个 容器时,docker 不需要和虚拟机一样重新加载一个操作系统内核。仍而避免引导、加载操作系统内核返个比较费时费资源的过程,当新建一个虚拟机时,虚拟机软件需要加载 GuestOS,返个新建过程是**分钟级别**的。而 docker 由于直接利用宿主机的操作系统,则省略了这个复杂的过程,因此新建一个 docker 容器只需要**几秒钟**。

Docker 的常用命令

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/engine/reference/commandline/build/

镜像命令

```
docker images #查看所有本地主机上的镜像  可以使用 docker image ls 代替

docker search  搜索镜像

docker pull  下载镜像  docker image pull

docker rmi  删除镜像  docker image rm
```

docker images 查看所有本地的主机上的镜像

```
➜  ~ docker images
REPOSITORY           TAG                 IMAGE ID            CREATED             SIZE
mysql                5.7                 e73346bdf465        24 hours ago        448MB

# 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
Options:
  -a, --all            Show all images (default hides intermediate images) #列出所有镜像
  -q, --quiet          Only show numeric IDs # 只显示镜像的 id

➜  ~ docker images -aq #显示所有镜像的 id
e73346bdf465
d03312117bb0
d03312117bb0
602e111c06b6
2869fc110bf7
470671670cac
bf756fb1ae65
5acf0e8da90b
```

docker search 搜索镜像

```
➜  ~ docker search mysql
NAME                            DESCRIPTION                                     STARS
OFFICIAL            AUTOMATED
mysql                           MySQL is a widely used, open-source relation…   9500
[OK]
mariadb                         MariaDB is a community-developed fork of MyS…   3444
[OK]
# --filter=STARS=3000 #搜索出来的镜像就是 STARS 大于 3000 的
Options:
  -f, --filter filter   Filter output based on conditions provided
      --format string   Pretty-print search using a Go template
      --limit int       Max number of search results (default 25)
      --no-trunc        Don't truncate output

➜  ~ docker search mysql --filter=STARS=3000
NAME                DESCRIPTION                                     STARS               OFFICIAL
AUTOMATED
```

```
mysql              MySQL is a widely used, open-source relation…   9500              [OK]
mariadb            MariaDB is a community-developed fork of MyS…   3444              [OK]
```

docker pull 下载镜像

```
# 下载镜像 docker pull 镜像名[:tag]
➜  ~ docker pull tomcat:8
8: Pulling from library/tomcat #如果不写 tag，默认就是 latest
90fe46dd8199: Already exists   #分层下载： docker image 的核心 联合文件系统
35a4f1977689: Already exists
bbc37f14aded: Already exists
74e27dc593d4: Already exists
93a01fbfad7f: Already exists
1478df405869: Pull complete
64f0dd11682b: Pull complete
68ff4e050d11: Pull complete
f576086003cf: Pull complete
3b72593ce10e: Pull complete
Digest: sha256:0c6234e7ec9d10ab32c06423ab829b32e3183ba5bf2620ee66de866df640a027  # 签名 防伪
Status: Downloaded newer image for tomcat:8
docker.io/library/tomcat:8 #真实地址

#等价于
docker pull tomcat:8
docker pull docker.io/library/tomcat:8
```

docker rmi 删除镜像

```
➜  ~ docker rmi -f 镜像 id #删除指定的镜像
➜  ~ docker rmi -f 镜像 id 镜像 id 镜像 id 镜像 id#删除指定的镜像
➜  ~ docker rmi -f $(docker images -aq) #删除全部的镜像
```

容器命令

docker run 镜像 id 新建容器并启动

docker ps 列出所有运行的容器 docker container list

docker rm 容器 id 删除指定容器

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 #强制停止当前容器
```

说明：我们有了镜像才可以创建容器，Linux，下载 centos 镜像来学习

```
➜  ~ docker container
Usage:  docker container COMMAND
Manage containers
Commands:
  attach      Attach local standard input, output, and error streams to a running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  inspect     Display detailed information on one or more containers
  kill        Kill one or more running containers
  logs        Fetch the logs of a container
  ls          List containers
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  prune       Remove all stopped containers
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  run         Run a command in a new container
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  wait        Block until one or more containers stop, then print their exit codes
Run 'docker container COMMAND --help' for more information on a command.
```

新建容器并启动

```
docker run [可选参数] image | docker container run [可选参数] image
#参书说明
--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(宿主机):8080(容器)
                -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(大写)                            随机指定端口
# 测试、启动并进入容器
➜  ~ docker run -it centos /bin/bash
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
8a29a15cefae: Already exists
Digest: sha256:fe8d824220415eed5477b63addf40fb06c3b049404242b31982106ac204f6700
Status: Downloaded newer image for centos:latest
[root@95039813da8d /]# ls
bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv  sys
tmp  usr  var
[root@95039813da8d /]# exit #从容器退回主机
exit
➜  ~ ls
shell  user.txt
```

列出所有运行的容器

```
#docker ps 命令 #列出当前正在运行的容器
  -a, --all            Show all containers (default shows just running)
  -n, --last int       Show n last created containers (includes all states) (default -1)
  -q, --quiet          Only display numeric IDs

  ➜  ~ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                    NAMES
68729e9654d4        portainer/portainer   "/portainer"             14 hours ago        Up About
a minute   0.0.0.0:8088->9000/tcp   funny_curie
d506a017e951        nginx               "nginx -g 'daemon of…"   15 hours ago        Up 15
hours        0.0.0.0:3344->80/tcp     nginx01
➜  ~ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                    NAMES
95039813da8d        centos              "/bin/bash"              3 minutes ago       Exited (0)
2 minutes ago                           condescending_pike
1e46a426a5ba        tomcat              "catalina.sh run"        11 minutes ago      Exited
(130) 9 minutes ago                         sweet_gould
14bc9334d1b2        bf756fb1ae65        "/hello"                 3 hours ago         Exited (0)
3 hours ago                             amazing_stonebraker
```

```
f10d60f473f5        bf756fb1ae65          "/hello"                3 hours ago         Exited (0)
3 hours ago                               dreamy_germain
68729e9654d4        portainer/portainer   "/portainer"            14 hours ago        Up About
a minute            0.0.0.0:8088->9000/tcp   funny_curie
677cde5e4f1d        elasticsearch         "/docker-entrypoint.…"  15 hours ago        Exited
(143) 8 minutes ago                           elasticsearch
33eb3f70b4db        tomcat                "catalina.sh run"       15 hours ago        Exited
(143) 8 minutes ago                           tomcat01
d506a017e951        nginx                 "nginx -g 'daemon of…"  15 hours ago        Up 15
hours                 0.0.0.0:3344->80/tcp     nginx01
24ce2db02e45        centos                "/bin/bash"             16 hours ago        Exited (0)
15 hours ago                              hopeful_faraday
42267d1ad80b        bf756fb1ae65          "/hello"                16 hours ago        Exited (0)
16 hours ago                              ecstatic_sutherland
➜  ~ docker ps -aq
95039813da8d
1e46a426a5ba
14bc9334d1b2
f10d60f473f5
68729e9654d4
677cde5e4f1d
33eb3f70b4db
d506a017e951
24ce2db02e45
42267d1ad80b
```

退出容器

```
exit #容器直接退出
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 -rf
docker rm -f $(docker ps -aq)  #删除指定的容器
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      #强制停止当前容器
```

常用其他命令

后台启动命令

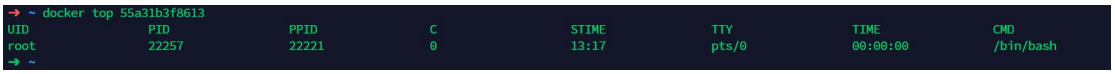
```
# 命令 docker run -d 镜像名
➜  ~ docker run -d centos
a8f922c255859622ac45ce3a535b7a0e8253329be4756ed6e32265d2dd2fac6c
➜  ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
# 问题 docker ps. 发现 centos 停止了
# 常见的坑，docker 容器使用后台运行，就必须要有要一个前台进程，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，就是没有程序了
```

查看日志

```
docker logs --help
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)
➜  ~ docker run -d centos /bin/sh -c "while true;do echo 6666;sleep 1;done" #模拟日志
#显示日志
-tf             #显示日志信息（一直更新）
--tail number #需要显示日志条数
docker logs -t --tail n 容器 id #查看 n 行日志
docker logs -ft 容器 id #跟着日志
```

查看容器中进程信息 ps

```
# 命令 docker top 容器 id
```



查看镜像的元数据

```
# 命令
docker inspect 容器 id

#测试
➜  ~ docker inspect 55321bcae33d
[
    {
        "Id": "55321bcae33d15da8280bcac1d2bc1141d213bcc8f8e792edfd832ff61ae5066",
        "Created": "2020-05-15T05:22:05.515909071Z",
```

```
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "while true;do echo 6666;sleep 1;done"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 22973,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2020-05-15T05:22:06.165904633Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:470671670cac686c7cf0081e0b37da2e9f4f768ddc5f6a26102ccd1c6954c1ee",
        "ResolvConfPath":
"/var/lib/docker/containers/55321bcae33d15da8280bcac1d2bc1141d213bcc8f8e792edfd832ff61ae50
66/resolv.conf",
        "HostnamePath":
"/var/lib/docker/containers/55321bcae33d15da8280bcac1d2bc1141d213bcc8f8e792edfd832ff61ae50
66/hostname",
        "HostsPath":
"/var/lib/docker/containers/55321bcae33d15da8280bcac1d2bc1141d213bcc8f8e792edfd832ff61ae50
66/hosts",
        "LogPath":
"/var/lib/docker/containers/55321bcae33d15da8280bcac1d2bc1141d213bcc8f8e792edfd832ff61ae50
66/55321bcae33d15da8280bcac1d2bc1141d213bcc8f8e792edfd832ff61ae5066-json.log",
        "Name": "/bold_bell",
        "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,
        "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/overlay2/1f347949ba49c4dbee70cea9ff3af39a14e602bc8fac8331c46347bf6708757a
-init/diff:/var/lib/docker/overlay2/5afcd8220c51854a847a36f52775b4ed0acb16fe6cfaec3bd2e5df
59863835ba/diff",
                "MergedDir":
"/var/lib/docker/overlay2/1f347949ba49c4dbee70cea9ff3af39a14e602bc8fac8331c46347bf6708757a
/merged",
                "UpperDir":
"/var/lib/docker/overlay2/1f347949ba49c4dbee70cea9ff3af39a14e602bc8fac8331c46347bf6708757a
/diff",
                "WorkDir":
"/var/lib/docker/overlay2/1f347949ba49c4dbee70cea9ff3af39a14e602bc8fac8331c46347bf6708757a
/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "55321bcae33d",
            "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 6666;sleep 1;done"
            ],
            "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": "63ed0c837f35c12453bae9661859f37a08541a0749afb86e881869bf6fd9031b",
        "HairpinMode": false,
        "LinkLocalIPv6Address": "",
        "LinkLocalIPv6PrefixLen": 0,
        "Ports": {},
        "SandboxKey": "/var/run/docker/netns/63ed0c837f35",
        "SecondaryIPAddresses": null,
        "SecondaryIPv6Addresses": null,
        "EndpointID":
"b129d9a5a2cbb92722a2101244bd81a9e3d8af034e83f338c13790a1a94552a1",
        "Gateway": "172.17.0.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAddress": "172.17.0.4",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "MacAddress": "02:42:ac:11:00:04",
        "Networks": {
            "bridge": {
                "IPAMConfig": null,
                "Links": null,
                "Aliases": null,
                "NetworkID":
"ad5ada6a106f5ba3dda9ce4bc1475a4bb593bf5f7fbead72196e66515e8ac36a",
                "EndpointID":
"b129d9a5a2cbb92722a2101244bd81a9e3d8af034e83f338c13790a1a94552a1",
                "Gateway": "172.17.0.1",
                "IPAddress": "172.17.0.4",
                "IPPrefixLen": 16,
                "IPv6Gateway": "",
                "GlobalIPv6Address": "",
                "GlobalIPv6PrefixLen": 0,
```

```
                "MacAddress": "02:42:ac:11:00:04",
                "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
➜  ~ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS               NAMES
55321bcae33d        centos              "/bin/sh -c 'while t…"   10 minutes ago      Up 10
minutes                         bold_bell
a7215824a4db        centos              "/bin/sh -c 'while t…"   13 minutes ago      Up 13
minutes                         zen_kepler
55a31b3f8613        centos              "/bin/bash"              15 minutes ago      Up 15
minutes                         lucid_clarke
➜  ~ docker exec -it 55321bcae33d /bin/bash
[root@55321bcae33d /]#
```

```
# 方式二
docker attach 容器id
#测试
docker attach 55321bcae33d
正在执行当前的代码...
区别
#docker exec #进入当前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
```

从容器内拷贝到主机上

```
docker cp 容器id:容器内路径   主机目的路径
#进入 docker 容器内部
➜  ~ docker exec -it  55321bcae33d /bin/bash
[root@55321bcae33d /]# ls
bin  etc   lib    lost+found  mnt  proc  run   srv  tmp  var
dev  home  lib64  media       opt  root  sbin  sys  usr
#新建一个文件
[root@55321bcae33d /]# echo "hello" > java.java
[root@55321bcae33d /]# cat java.java
hello
[root@55321bcae33d /]# exit
exit
➜  ~ docker cp 55321bcae33d:/java.java /    #拷贝
➜  ~ cd /
➜  / ls  #可以看见 java.java 存在
bin   home           lib         mnt   run       sys  vmlinuz
boot  initrd.img     lib64       opt   sbin      tmp  vmlinuz.old
dev   initrd.img.old  lost+found  proc  srv       usr  wget-log
etc   java.java      media       root  swapfile  var
```

学习方式：将我的所有笔记敲一遍，自己记录笔记！

小结：

```
  attach      Attach local standard input, output, and error streams to a running container
  #当前 shell 下 attach 连接指定运行的镜像
  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 #拷贝文件
  create      Create a new container #创建一个新的容器
  diff        Inspect changes to files or directories on a container's filesystem #查看 docker
容器的变化
  events      Get real time events from the server # 从服务获取容器实时时间
  exec        Run a command in a running container # 在运行中的容器上运行命令
  export      Export a container's filesystem as a tar archive #导出容器文件系统作为一个 tar 归
档文件[对应 import]
  history     Show the history of an image # 展示一个镜像形成历史
  images      List images #列出系统当前的镜像
  import      Import the contents from a tarball to create a filesystem image #从 tar 包中导入
内容创建一个文件系统镜像
```

```
  info        Display system-wide information # 显示全系统信息
  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 包或标准输入中加载一个镜像[对
应 save]
  login       Log in to a Docker registry #
  logout      Log out from a Docker registry
  logs        Fetch the logs of a container
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  ps          List containers
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  rmi         Remove one or more images
  run         Run a command in a new container
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  search      Search the Docker Hub for images
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  version     Show the Docker version information
  wait        Block until one or more containers stop, then print their exit codes
```

作业练习

Docker 安装 Nginx

```
#1. 搜索镜像 search 建议大家去 docker 搜索，可以看到帮助文档
#2. 拉取镜像 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
➜  ~ docker run -d --name nginx00 -p 82:80 nginx
75943663c116f5ed006a0042c42f78e9a1a6a52eba66311666eee12e1c8a4502
➜  ~ docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS               NAMES
75943663c116        nginx               "nginx -g 'daemon of…"   41 seconds ago      Up 40
seconds       0.0.0.0:82->80/tcp   nginx00
➜  ~ curl localhost:82   #测试
<!DOCTYPE html>,,,,
```

思考问题：我们每次改动 nginx 配置文件，都需要进入容器内部？十分的麻烦，要是可以在容器外部提供一个映射路径，达到在容器修改文件名，容器内部就可以自动修改？√数据卷！

作用：docker 来装一个 tomcat

```
# 官方的使用
docker run -it --rm tomcat:9.0
# 之前的启动都是后台，停止了容器，容器还是可以查到， docker run -it --rm image 一般是用来测试，用完就删除
--rm       Automatically remove the container when it exits
#下载
docker pull tomcat
#启动运行
docker run -d -p 8080:8080 --name tomcat01 tomcat
#测试访问有没有问题
curl localhost:8080

#进入容器
➜  ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS                    NAMES
db09851cf82e        tomcat              "catalina.sh run"   28 seconds ago      Up 27 seconds
0.0.0.0:8080->8080/tcp   tomcat01
➜  ~ docker exec -it db09851cf82e /bin/bash
root@db09851cf82e:/usr/local/tomcat#
# 发现问题：1、linux 命令少了。 2.没有 webapps
```

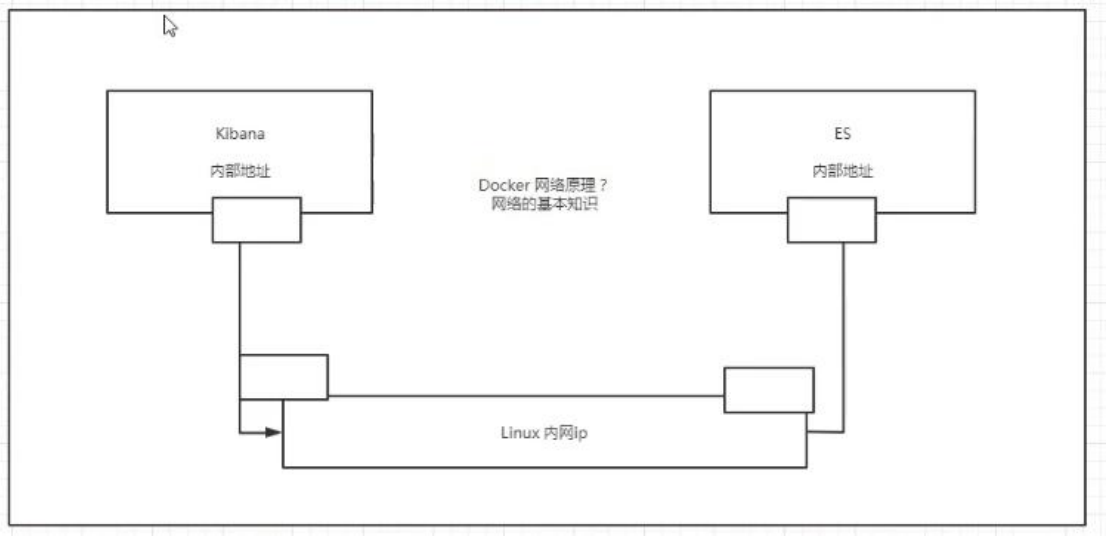
思考问题：我们以后要部署项目，如果每次都要进入容器是不是十分麻烦？要是可以在容器外部提供一个映射路径，webapps，我们在外部放置项目，就自动同步内部就好了！

作业：部署 es+kibana

```
# es 暴露的端口很多!
# es 的数据一般需要放置到安全目录! 挂载
# --net somenetwork ? 网络配置

# 启动 elasticsearch
docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node"
elasticsearch:7.6.2
# 测试一下 es 是否成功启动
➜  ~ curl localhost:9200
{
  "name" : "d73ad2f22dd3",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "atFKgANxS8CzgIyCB8PGxA",
  "version" : {
    "number" : "7.6.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "ef48eb35cf30adf4db14086e8aabd07ef6fb113f",
    "build_date" : "2020-03-26T06:34:37.794943Z",
    "build_snapshot" : false,
    "lucene_version" : "8.4.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
➜  ~ docker stats # 查看 docker 容器使用内存情况


#关闭，添加内存的限制，修改配置文件 -e 环境配置修改
➜  ~ docker rm -f d73ad2f22dd3
➜  ~ docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e
"discovery.type=single-node" -e ES_JAVA_OPTS="-Xms64m -Xmx512m" elasticsearch:7.6.2

➜  ~ curl localhost:9200
{
  "name" : "b72c9847ec48",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "yNAK0EORSvq3Wtaqe2QqAg",
  "version" : {
    "number" : "7.6.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "ef48eb35cf30adf4db14086e8aabd07ef6fb113f",
```

```
    "build_date" : "2020-03-26T06:34:37.794943Z",
    "build_snapshot" : false,
    "lucene_version" : "8.4.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

作业：使用 kibana 连接 es？思考网络如何才能连接



可视化

- portainer(先用这个)

```
docker run -d -p 8080:9000 \
--restart=always -v /var/run/docker.sock:/var/run/docker.sock --privileged=true
portainer/portainer
```

- Rancher(CI/CD 再用)

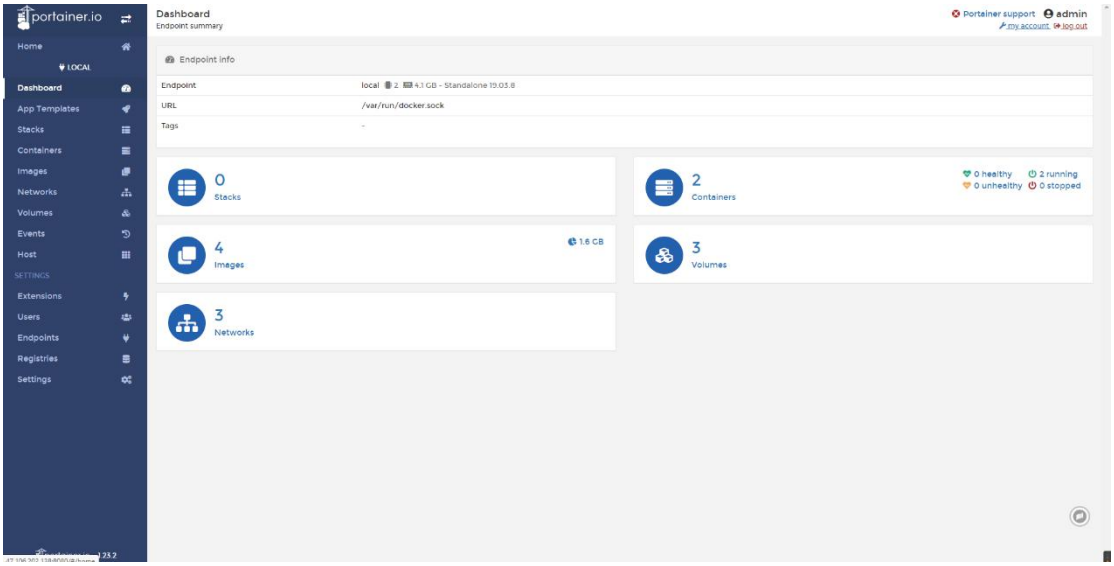
什么是 portainer?

Docker 图形化界面管理工具！提供一个后台面板供我们操作！

```
docker run -d -p 8080:9000 \
--restart=always -v /var/run/docker.sock:/var/run/docker.sock --privileged=true
portainer/portainer
```

测试访问： 外网：8080

进入之后的面板



Docker 镜像讲解

镜像是什么

镜像是一种轻量级、可执行的独立软件保，用来打包软件运行环境和基于运行环境开发的软件，他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

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，他支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下（ unite several directories into a single virtual filesystem)。Union 文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像

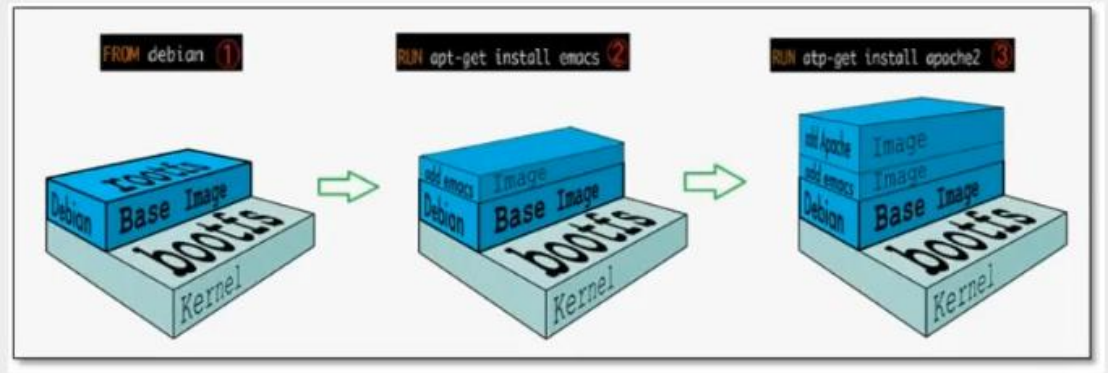
特性：一次同时加载多个文件系统，但从外面看起来，只能看到一个文件系统，联合加载会把各层文件系统叠加起来，这样最终的文件系统会包含所有底层的文件和目录

Docker 镜像加载原理

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

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

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



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 200M?

对于个精简的 OS,rootfs 可以很小，只需要包合最基本的命令，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， 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.

虚拟机是分钟级别，容器是秒级！

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，可以看到是一层层的在下载

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

```
➜  / docker image inspect redis
[
    {
        "Id": "sha256:f9b9909726890b00d2098081642edf32e5211b7ab53563929a47f250bcdc1d7c",
        "RepoTags": [
            "redis:latest"
        ],
        "RepoDigests": [
            "redis@sha256:399a9b17b8522e24fbe2fd3b42474d4bb668d3994153c4b5d38c3dafd5903e32"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2020-05-02T01:40:19.112130797Z",
        "Container": "d30c0bcea88561bc5139821227d2199bb027eeba9083f90c701891b4affce3bc",
        "ContainerConfig": {
            "Hostname": "d30c0bcea885",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "6379/tcp": {}
```

```
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "GOSU_VERSION=1.12",
                "REDIS_VERSION=6.0.1",
                "REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-6.0.1.tar.gz",

"REDIS_DOWNLOAD_SHA=b8756e430479edc162ba9c44dc89ac394316cd482f2dc6b91bcd5fe12593f273"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"redis-server\"]"
            ],
            "ArgsEscaped": true,
            "Image":
"sha256:704c602fa36f41a6d2d08e49bd2319ccd6915418f545c838416318b3c29811e0",
            "Volumes": {
                "/data": {}
            },
            "WorkingDir": "/data",
            "Entrypoint": [
                "docker-entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "18.09.7",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "6379/tcp": {}
            },
            "Tty": false,
```

```
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "GOSU_VERSION=1.12",
                "REDIS_VERSION=6.0.1",
                "REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-6.0.1.tar.gz",

"REDIS_DOWNLOAD_SHA=b8756e430479edc162ba9c44dc89ac394316cd482f2dc6b91bcd5fe12593f273"
            ],
            "Cmd": [
                "redis-server"
            ],
            "ArgsEscaped": true,
            "Image":
"sha256:704c602fa36f41a6d2d08e49bd2319ccd6915418f545c838416318b3c29811e0",
            "Volumes": {
                "/data": {}
            },
            "WorkingDir": "/data",
            "Entrypoint": [
                "docker-entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": null
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 104101893,
        "VirtualSize": 104101893,
        "GraphDriver": {
            "Data": {
                "LowerDir":
"/var/lib/docker/overlay2/adea96bbe6518657dc2d4c6331a807eea70567144abda686588ef6c3bb0d778a
/diff:/var/lib/docker/overlay2/66abd822d34dc6446e6bebe73721dfd1dc497c2c8063c43ffb8cf8140e2
caeb6/diff:/var/lib/docker/overlay2/d19d24fb6a24801c5fa639c1d979d19f3f17196b3c6dde96d3b69c
d2ad07ba8a/diff:/var/lib/docker/overlay2/a1e95aae5e09ca6df4f71b542c86c677b884f5280c1d3e3a1
111b13644b221f9/diff:/var/lib/docker/overlay2/cd90f7a9cd0227c1db29ea992e889e4e6af057d9ab28
35dd18a67a019c18bab4/diff",
                "MergedDir":
"/var/lib/docker/overlay2/afa1de233453b60686a3847854624ef191d7bc317fb01e015b4f06671139fb11
/merged",
```

```
            "UpperDir":
"/var/lib/docker/overlay2/afa1de233453b60686a3847854624ef191d7bc317fb01e015b4f06671139fb11
/diff",
            "WorkDir":
"/var/lib/docker/overlay2/afa1de233453b60686a3847854624ef191d7bc317fb01e015b4f06671139fb11
/work"
        },
        "Name": "overlay2"
    },
    "RootFS": {
        "Type": "layers",
        "Layers": [
            "sha256:c2adabaecedbda0af72b153c6499a0555f3a769d52370469d8f6bd6328af9b13",
            "sha256:744315296a49be711c312dfa1b3a80516116f78c437367ff0bc678da1123e990",
            "sha256:379ef5d5cb402a5538413d7285b21aa58a560882d15f1f553f7868dc4b66afa8",
            "sha256:d00fd460effb7b066760f97447c071492d471c5176d05b8af1751806a1f905f8",
            "sha256:4d0c196331523cfed7bf5bafd616ecb3855256838d850b6f3d5fba911f6c4123",
            "sha256:98b4a6242af2536383425ba2d6de033a510e049d9ca07ff501b95052da76e894"
        ]
    },
    "Metadata": {
        "LastTagTime": "0001-01-01T00:00:00Z"
    }
}
]
```

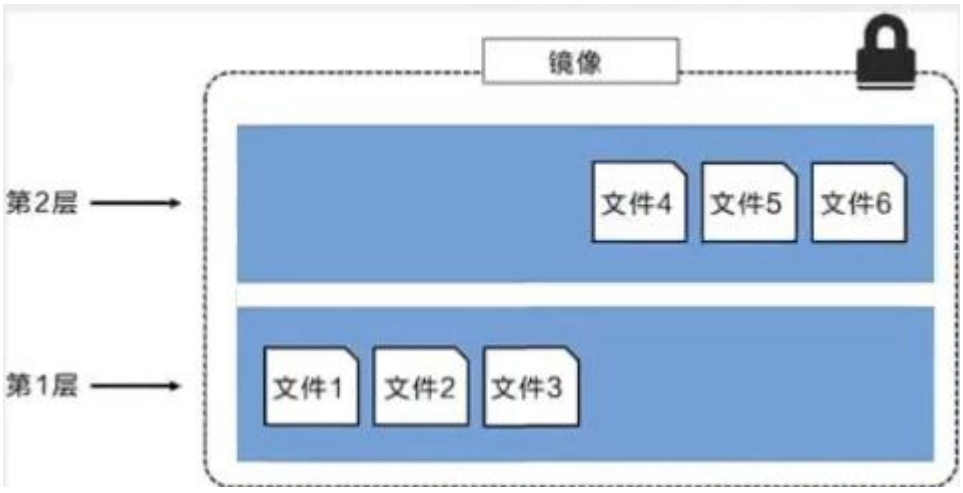
理解：

所有的 Docker 镜像都起始于一个基础镜像层，当进行修改或培加新的内容时，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 Linux16.04 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，

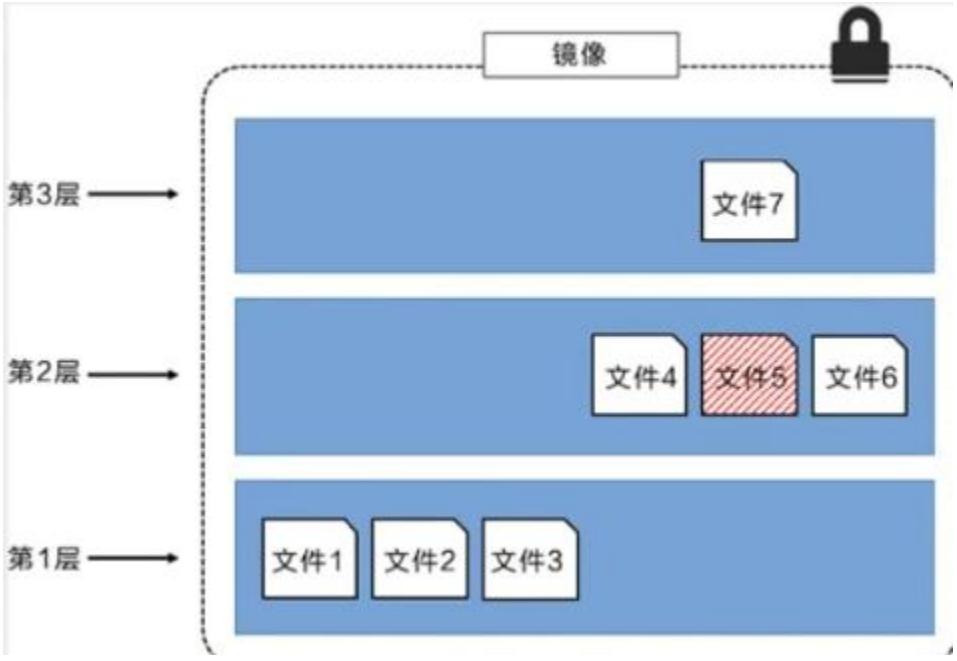
就会在基础镜像层之上创建第二个镜像层；如果继续添加一个安全补丁，就会创健第三个镜像层该像当前已经包含 3 个镜像层，如下图所示（这只是一个用于演示的很简单的例子）。

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



上图中的镜像层跟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，这是因为最上层中的文件 7 是文件 5 的一个更新版



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

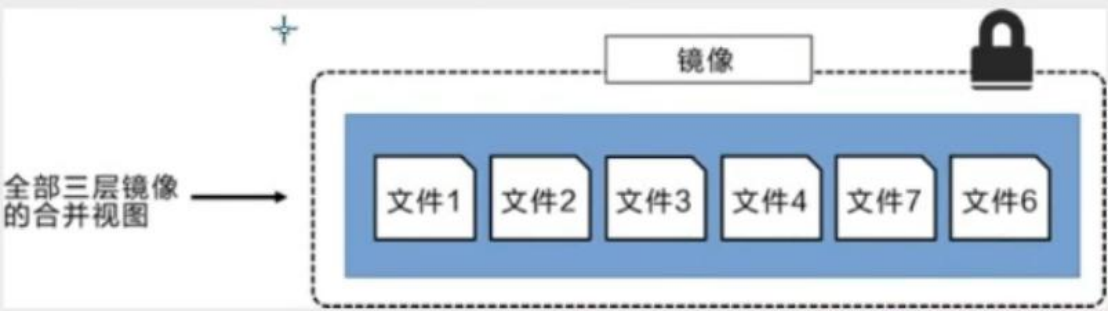
Docker 通过存储引擎（新版本采用快照机制）的方式来实现镜像层堆栈，并保证多镜像层对外展示为统一的文件系统

Linux 上可用的存储引擎有 AUFS、 Overlay2、 Device Mapper、Btrfs 以及 ZFS。顾名思义，每种存储引擎都基于 Linux 中对应的

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]。

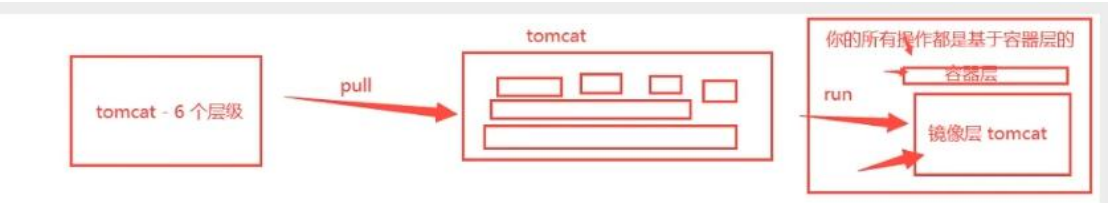
下图展示了与系统显示相同的三层镜像。所有镜像层堆并合并，对外提供统一的视图



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！



commit 镜像

```
docker commit 提交容器成为一个新的副本

# 命令和 git 原理类似
docker commit -m="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
docker run -d -p 8080:8080 tomcat
# 2、发现这个默认的 tomcat 是没有 webapps 应用，官方的镜像默认 webapps 下面是没有文件的！
docker exec -it 容器 id
# 3、拷贝文件进去

# 4、将操作过的容器通过 commit 调教为一个镜像！我们以后就使用我们修改过的镜像即可，这就是我们自己的一个修改的镜像。
docker commit -m="描述信息" -a="作者" 容器 id 目标镜像名:[TAG]
docker commit -a="kuangshen" -m="add webapps app" 容器 id tomcat02:1.0
```

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，就好比我们我们使用虚拟机的快照。

入门成功！！！！

docker 进阶 之容器数据卷 DockerFile Docker 网络 — 狂神说

容器数据卷

DockerFile

Docker 网络讲解

IDEA 整合 Docker

Docker Compose

Docker Swarm

CI、CD 之 Jenkins

容器数据卷

什么是容器数据卷

docker 的理念回顾

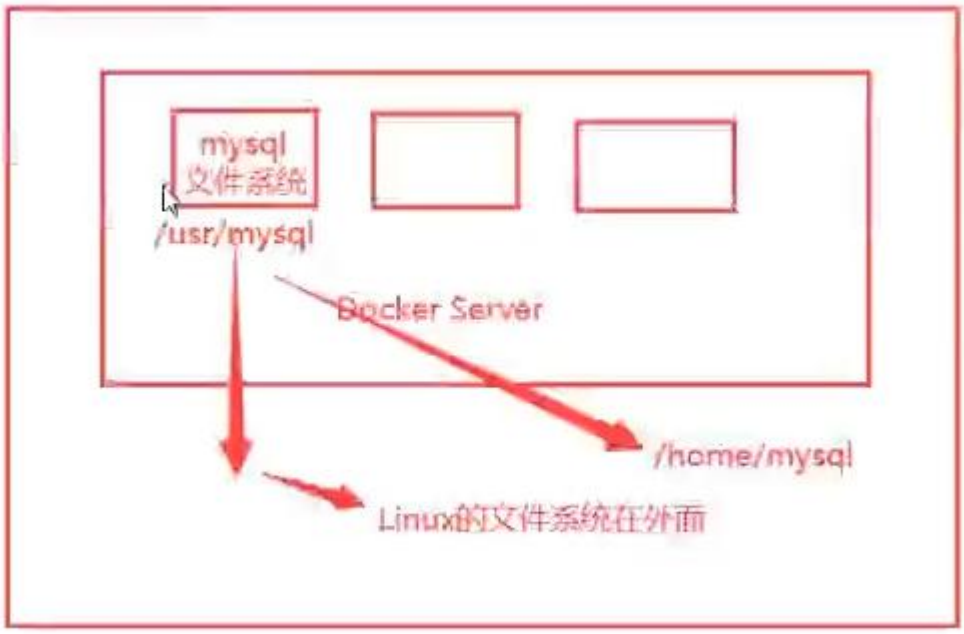
将应用和环境打包成一个镜像！

数据？如果数据都在容器中，那么我们容器删除，数据就会丢失！需求：数据可以持久化

MySQL，容器删除了，删库跑路！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 ： 直接使用命令挂载 -v
```

```
-v, --volume list                    Bind mount a volume
```

```
docker run -it -v 主机目录:容器内目录  -p 主机端口:容器内端口
➜ ~ docker run -it -v /home/ceshi:/home centos /bin/bash
#通过 docker inspect 容器 id 查看
```

测试文件的同步

再来测试!

1、停止容器

2、宿主机修改文件

3、启动容器

4、容器内的数据依旧是同步的

好处：我们以后修改只需要在本地修改即可，容器内会自动同步!

实战：安装 MySQL

思考：MySQL 的数据持久化的问题

```
# 获取 mysql 镜像
➜  ~ 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，这是要注意点!
# 参考官网 hub
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag

#启动我们得
-d 后台运行
-p 端口映射
-v 卷挂载
```

```
-e 环境配置
-- name 容器名字
➜  ~ docker run -d -p 3306:3306 -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 3306--和容器内的 3306 映射

# 在本地测试创建一个数据库，查看一下我们映射的路径是否 ok!
```

假设我们将容器删除

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。

具名和匿名挂载

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

# 查看所有的 volume 的情况
➜  ~ docker volume ls
DRIVER              VOLUME NAME
local               33ae588fae6d34f511a769948f0d3d123c9d45c442ac7728cb85599c2657e50d
local
# 这里发现，这种就是匿名挂载，我们在 -v 只写了容器内的路径，没有写容器外的路劲!

# 具名挂载
➜  ~ docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx nginx
➜  ~ docker volume ls
DRIVER              VOLUME NAME
local               juming-nginx

# 通过 -v 卷名：容器内路径
# 查看一下这个卷
```

所有的 docker 容器内的卷，没有指定目录的情况下都是在`/var/lib/docker/volumes/xxxx/_data`下

如果指定了目录，docker volume ls 是查看不到的

```
# 三种挂载： 匿名挂载、具名挂载、指定路径挂载
-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 docker volume ls 是查看不到的
```

拓展：

```
# 通过 -v 容器内路径： ro rw 改变读写权限
ro #readonly 只读
rw #readwrite 可读可写
docker run -d -P --name nginx05 -v juming:/etc/nginx:ro nginx
docker run -d -P --name nginx05 -v juming:/etc/nginx:rw nginx

# ro 只要看到 ro 就说明这个路径只能通过宿主机来操作，容器内部是无法操作！
```

初始 Dockerfile

Dockerfile 就是用来构建 docker 镜像的构建文件！命令脚本！先体验一下！

通过这个脚本可以生成镜像，镜像

```
# 创建一个 dockerfile 文件，名字可以随便 建议 Dockerfile
# 文件中的内容 指令(大写) 参数
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！
```

```
# 启动自己写的镜像
```

1

这个卷和外部一定有一个同步的目录

```
FROM centos

VOLUME ["volume01","volume02"]

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

匿名挂载

查看一下卷挂载

```
docker inspect 容器id
```

测试一下刚才的文件是否同步出去了！

这种方式使用的十分多，因为我们通常会构建自己的镜像！

假设构建镜像时候没有挂载卷，要手动镜像挂载 -v 卷名：容器内路径！

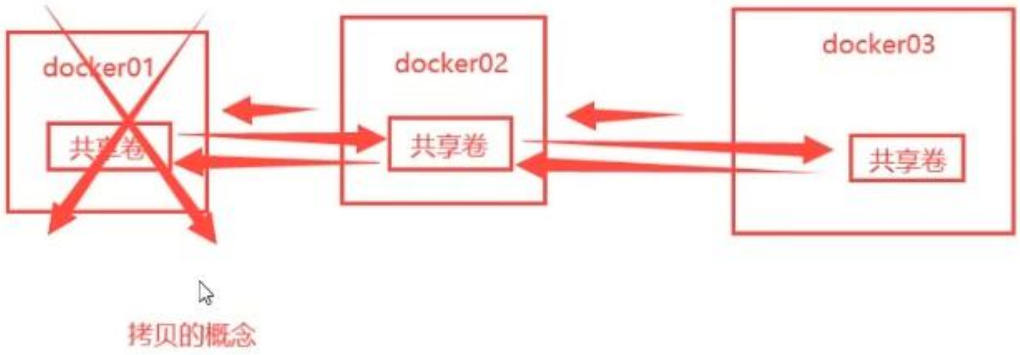
数据卷容器

多个 MySQL 同步数据！

命名的容器挂载数据卷！


```
--volumes-from list            Mount volumes from the specified container(s)
# 测试，我们通过刚才启动的
```

```
# 测试：可以删除 docker01，查看一下 docker02 和 docker03 是否可以访问这个文件
# 测试依旧可以访问
```



多个 mysql 实现数据共享

```
➜  ~ docker run -d -p 3306:3306 -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 3307:3306 -e MYSQL_ROOT_PASSWORD=123456 --name mysql02 --volumes-from
mysql01  mysql:5.7
# 这个时候，可以实现两个容器数据同步！
```

结论：

容器之间的配置信息的传递，数据卷容器的生命周期一直持续到没有容器使用为止。

但是一旦你持久化到了本地，这个时候，本地的数据是不会删除的！

DockerFile

DockerFile 介绍

`dockerfile` 是用来构建 docker 镜像的文件！命令参数脚本！

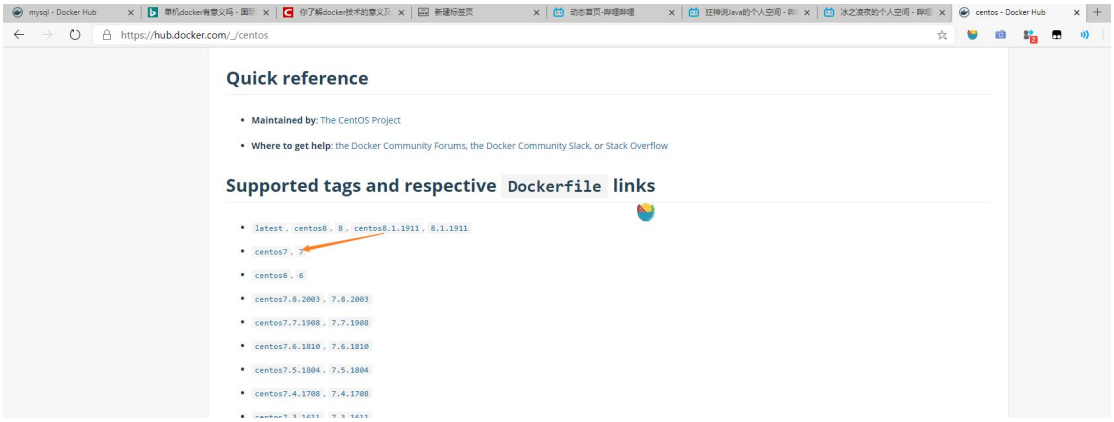
构建步骤：

1、 编写一个 dockerfile 文件

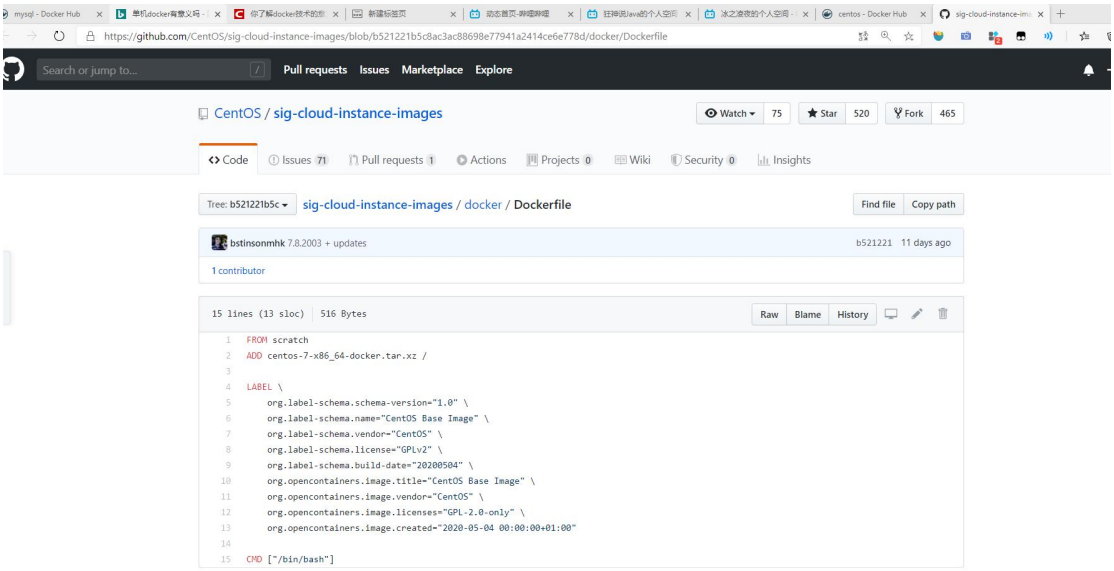
2、 docker build 构建称为一个镜像

3、 docker run 运行镜像

4、 docker push 发布镜像（DockerHub 、阿里云仓库)



点击后跳到一个 Dockerfile



很多官方镜像都是基础包，很多功能没有，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

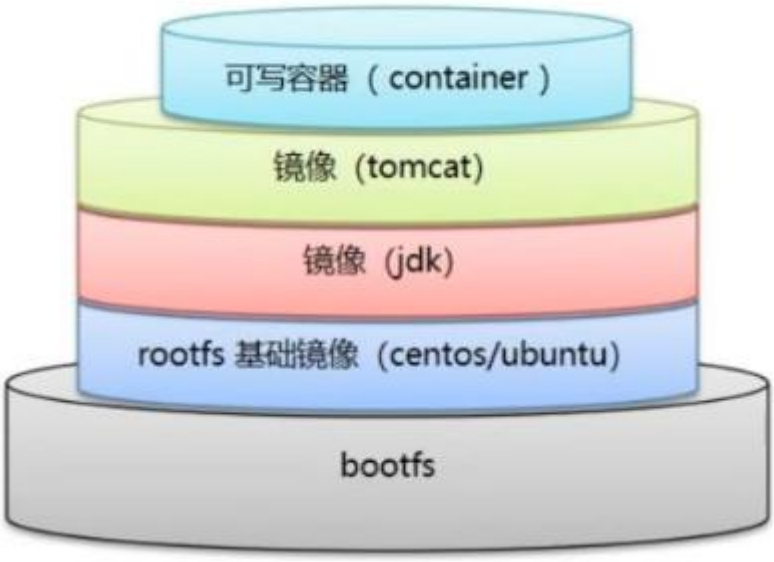
基础知识：

1、每个保留关键字(指令）都是必须是大写字母

2、执行从上到下顺序

3、#表示注释

4、每一个指令都会创建提交一个新的镜像曾，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！

Docker 镜像逐渐成企业交付的标准，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，最终发布和运行产品。

Docker 容器：容器就是镜像运行起来提供服务。

DockerFile 的指令

```
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 添加同目录
WORKDIR                 # 镜像的工作目录
VOLUME                  # 挂载的目录
EXPOSE                  # 保留端口配置
```

```
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，触发指令。
COPY                                    # 类似 ADD，将我们文件拷贝到镜像中
ENV                                             # 构建的时候设置环境变量！
```

实战测试

创建一个自己的 centos

```
# 1.编写 Dockerfile 文件
vim mydockerfile-centos
FROM centos
MAINTAINER cheng<1204598429@qq.com>

ENV MYPATH /usr/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
```

```
# 2、通过这个文件构建镜像
# 命令 docker build -f 文件路径 -t 镜像名:[tag] .
docker build -f mydockerfile-centos -t mycentos:0.1 .
```

测试运行

我们可以列出本地进行的变更历史

我们平时拿到一个镜像，可以研究一下是什么做的

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
```

测试 cmd

```
# 编写 dockerfile 文件
$ vim dockerfile-test-cmd
FROM centos
CMD ["ls","-a"]
# 构建镜像
$ docker build  -f dockerfile-test-cmd -t cmd-test:0.1 .
# 运行镜像
$ docker run cmd-test:0.1
.
..
.dockerenv
bin
dev

# 想追加一个命令  -l 成为 ls -al
$ docker run cmd-test:0.1 -l
docker: Error response from daemon: OCI runtime create failed: container_linux.go:349: starting container process caused "exec: \"-l\":
 executable file not found in $PATH": unknown.
ERRO[0000] error waiting for container: context canceled
# cmd 的情况下 -l 替换了 CMD["ls","-l"]。 -l  不是命令所有报错
```

测试 ENTRYPOINT

```
# 编写 dockerfile 文件
$ vim dockerfile-test-entrypoint
FROM centos
ENTRYPOINT ["ls","-a"]
$ docker run entrypoint-test:0.1
.
..
.dockerenv
bin
dev
etc
```

```
home
lib
lib64
lost+found ...
# 我们的命令，是直接拼接在我们得 ENTRYPOINT 命令后面的
$ docker run entrypoint-test:0.1 -l
total 56
drwxr-xr-x   1 root root 4096 May 16 06:32 .
drwxr-xr-x   1 root root 4096 May 16 06:32 ..
-rwxr-xr-x   1 root root    0 May 16 06:32 .dockerenv
lrwxrwxrwx   1 root root    7 May 11  2019 bin -> usr/bin
drwxr-xr-x   5 root root  340 May 16 06:32 dev
drwxr-xr-x   1 root root 4096 May 16 06:32 etc
drwxr-xr-x   2 root root 4096 May 11  2019 home
lrwxrwxrwx   1 root root    7 May 11  2019 lib -> usr/lib
lrwxrwxrwx   1 root root    9 May 11  2019 lib64 -> usr/lib64 ....
```

Dockerfile 中很多命令都十分的相似，我们需要了解它们的区别，我们最好的学习就是对比他们然后测试效果！

实战：Tomcat 镜像

1、准备镜像文件

准备 tomcat 和 jdk 到当前目录，编写好 README

2、编写 dokerfile

```
FROM centos #
MAINTAINER cheng<1204598429@qq.com>
COPY README /usr/local/README #复制文件
ADD jdk-8u231-linux-x64.tar.gz /usr/local/ #复制解压
ADD apache-tomcat-9.0.35.tar.gz /usr/local/ #复制解压
RUN yum -y install vim
ENV MYPATH /usr/local #设置环境变量
WORKDIR $MYPATH #设置工作目录
ENV JAVA_HOME /usr/local/jdk1.8.0_231 #设置环境变量
ENV CATALINA_HOME /usr/local/apache-tomcat-9.0.35 #设置环境变量
```

```
ENV PATH $PATH:$JAVA_HOME/bin:$CATALINA_HOME/lib #设置环境变量 分隔符是:
EXPOSE 8080 #设置暴露的端口
CMD /usr/local/apache-tomcat-9.0.35/bin/startup.sh && tail -F
/usr/local/apache-tomcat-9.0.35/logs/catalina.out # 设置默认命令
```

3、构建镜像

```
# 因为 dockerfile 命名使用默认命名 因此不用使用-f 指定文件
$ docker build -t mytomcat:0.1 .
```

4、run 镜像

```
$ docker run -d -p 8080:8080 --name tomcat01 -v
/home/kuangshen/build/tomcat/test:/usr/local/apache-tomcat-9.0.35/webapps/test -v
/home/kuangshen/build/tomcat/tomcatlogs/:/usr/local/apache-tomcat-9.0.35/logs mytomcat:0.1
```

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！)

发现：项目部署成功，可以直接访问！

我们以后开发的步骤：需要掌握 Dockerfile 的编写！我们之后的一切都是使用 docker 镜像来发布运行！

发布自己的镜像

```
Dockerhub
```

1、地址 https://hub.docker.com/

2、确定这个账号可以登录

3、登录

```
$ 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、提交 push 镜像

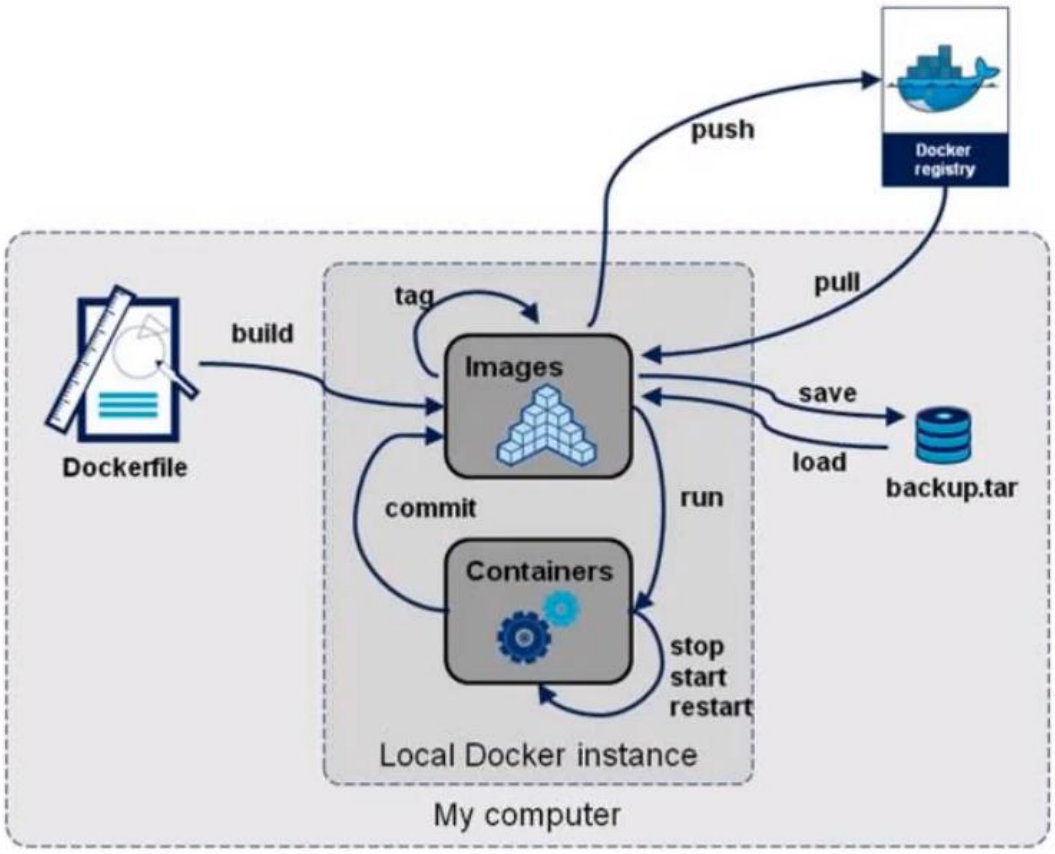
```
# 会发现 push 不上去，因为如果没有前缀的话默认是 push 到 官方的 library
# 解决方法
# 第一种 build 的时候添加你的 dockerhub 用户名，然后在 push 就可以放到自己的仓库了
$ docker build -t chengcoder/mytomcat:0.1 .
# 第二种 使用 docker tag #然后再次 push
$ docker tag 容器 id chengcoder/mytomcat:1.0 #然后再次 push
```

阿里云镜像服务上

看官网 很详细 https://cr.console.aliyun.com/repository/

```
$ sudo docker login --username=zchengx registry.cn-shenzhen.aliyuncs.com
$ sudo docker tag [ImageId] registry.cn-shenzhen.aliyuncs.com/dsadxzc/cheng:[镜像版本号]
# 修改 id 和 版本
sudo docker tag a5ef1f32aaae registry.cn-shenzhen.aliyuncs.com/dsadxzc/cheng:1.0
# 修改版本
$ sudo docker push registry.cn-shenzhen.aliyuncs.com/dsadxzc/cheng:[镜像版本号]
```

小结



Docker 网络

理解 Docker 0

清空所有网络

```
测试
```

三个网络

```
# 问题： docker 是如果处理容器网络访问的？
```

```
# 测试  运行一个 tomcat
$ docker run -d --name tomcat01 tomcat

$ 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
551: vethbfc37e3@if550: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
state UP group default
    link/ether 1a:81:06:13:ec:a1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::1881:6ff:fe13:eca1/64 scope link
       valid_lft forever preferred_lft forever

$ docker exec -it 容器 id
$ ip addr
# 查看容器内部网络地址 发现容器启动的时候会得到一个 eth0@if551 ip 地址，docker 分配！
550: eth0@if551: <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 能不能 ping 通容器内部！ 可以 容器内部可以 ping 通外界吗？ 可以！
$ 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=0.069 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.074 ms
```

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，我们只要按照了 docker，

就会有一个 docker0 桥接模式，使用的技术是 veth-pair 技术！

https://www.cnblogs.com/bakari/p/10613710.html

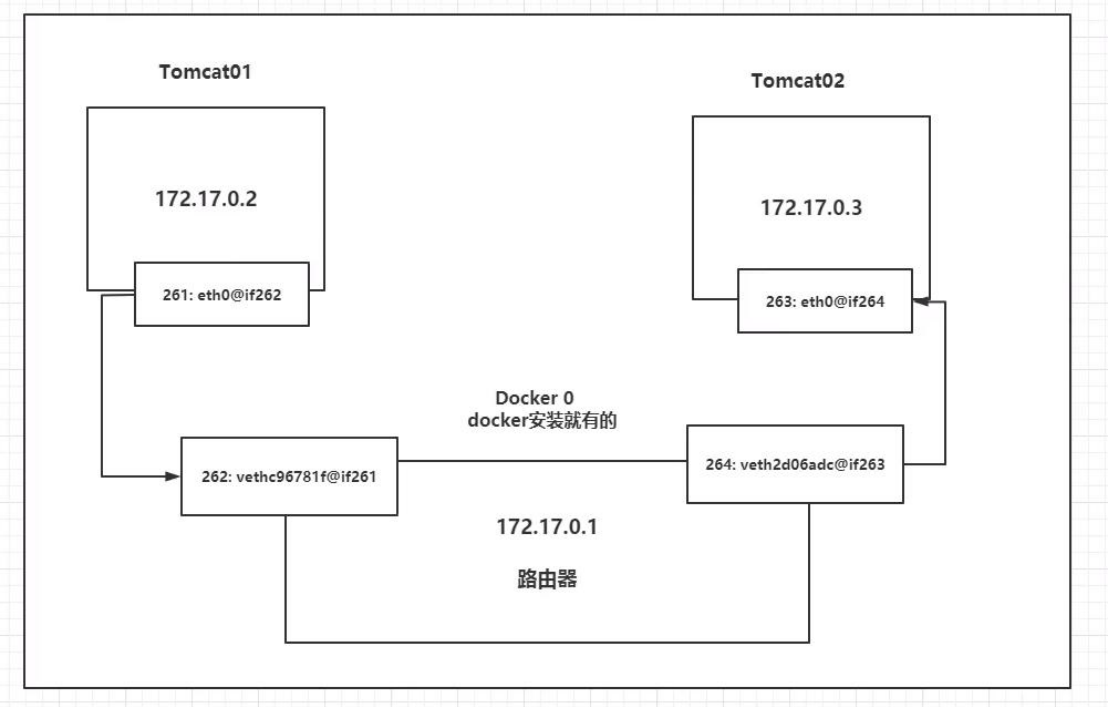
再次测试 ip add

2 、在启动一个容器测试，发现又多了一对网络

```
# 我们发现这个容器带来网卡，都是一对对的
# veth-pair 就是一对的虚拟设备接口，他们都是成对出现的，一端连着协议，一端彼此相连
# 正因为有这个特性 veth-pair 充当一个桥梁，连接各种虚拟网络设备的
# OpenStac,Docker 容器之间的连接，OVS 的连接，都是使用 evth-pair 技术
```

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

```
$ docker-tomcat docker exec -it tomcat01 ip addr  #获取 tomcat01 的 ip 172.17.0.2
550: eth0@if551: <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
$ docker-tomcat docker exec -it tomcat02 ping 172.17.0.2#让 tomcat02ping tomcat01
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=0.098 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.071 ms
# 可以 ping 通
```

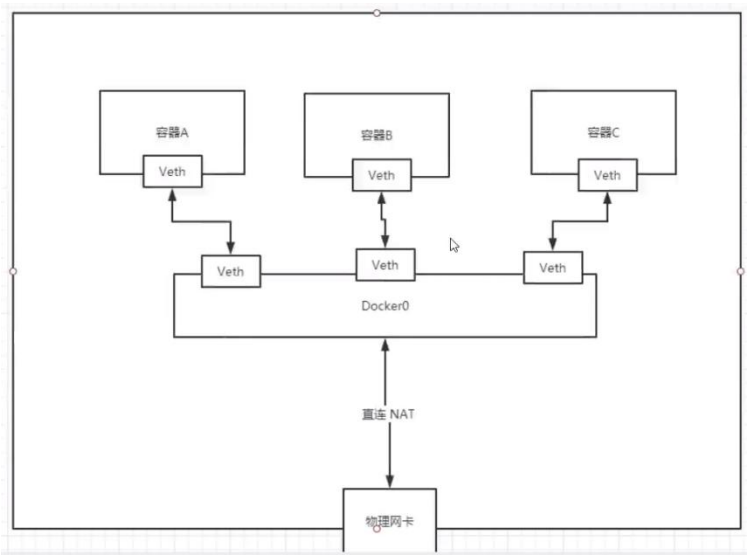


结论：tomcat01 和 tomcat02 公用一个路由器，docker0。

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

> 小结：

Docker 使用的是 Linux 的桥接，宿主机是一个 Docker 容器的网桥 docker0



Docker 中所有网络接口都是虚拟的，虚拟的转发效率高（内网传递文件）

只要容器删除，对应的网桥一对就没了！

> 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 ip 换了，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？

–link

```
$ docker exec -it tomcat02 ping tomca01   # ping 不通
ping: tomca01: Name or service not known
# 运行一个 tomcat03 --link tomcat02
$ docker run -d -P --name tomcat03 --link tomcat02 tomcat
5f9331566980a9e92bc54681caaac14e9fc993f14ad13d98534026c08c0a9aef
# 用 tomcat03 ping tomcat02 可以 ping 通
$ 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=0.115 ms
64 bytes from tomcat02 (172.17.0.3): icmp_seq=2 ttl=64 time=0.080 ms
```

```
# 用 tomcat02 ping tomcat03 ping 不通
```

探究：

docker network inspect 网络 id 网段相同

docker inspect tomcat03

查看 tomcat03 里面的/etc/hosts 发现有 tomcat02 的配置

–link 本质就是在 hosts 配置中添加映射

现在使用 Docker 已经不建议使用–link 了！

自定义网络，不适用 docker0！

docker0 问题：不支持容器名连接访问！

自定义网络

```
docker network
connect     -- Connect a container to a network
create      -- Creates a new network with a name specified by the
disconnect  -- Disconnects a container from a network
inspect     -- Displays detailed information on a network
ls          -- Lists all the networks created by the user
prune       -- Remove all unused networks
rm          -- Deletes one or more networks
```

查看所有的 docker 网络

网络模式

bridge ： 桥接 docker（默认，自己创建也是用 bridge 模式）

none ： 不配置网络，一般不用

host ： 和所主机共享网络

container ： 容器网络连通（用得少！局限很大）

测试

```
# 我们直接启动的命令 --net bridge,而这个就是我们得 docker0
# bridge 就是 docker0
$ docker run -d -P --name tomcat01 tomcat
等价于 => docker run -d -P --name tomcat01 --net bridge tomcat

# docker0，特点：默认，域名不能访问。 --link 可以打通连接，但是很麻烦！
# 我们可以 自定义一个网络
$ docker network create --driver bridge --subnet 192.168.0.0/16 --gateway 192.168.0.1 mynet
```

```
$ docker network inspect mynet;
```

启动两个 tomcat,再次查看网络情况

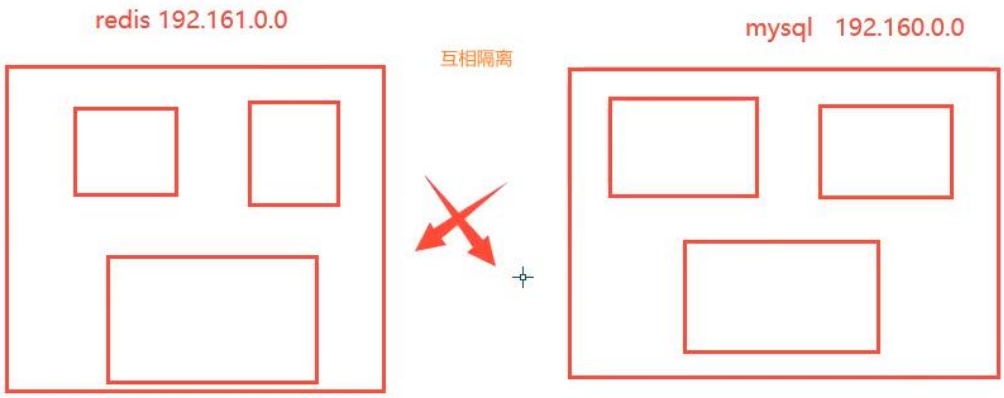
在自定义的网络下，服务可以互相 ping 通，不用使用–link

我们自定义的网络 docker 当我们维护好了对应的关系，推荐我们平时这样使用网络！

好处：

redis -不同的集群使用不同的网络，保证集群是安全和健康的

mysql-不同的集群使用不同的网络，保证集群是安全和健康的



网络连通

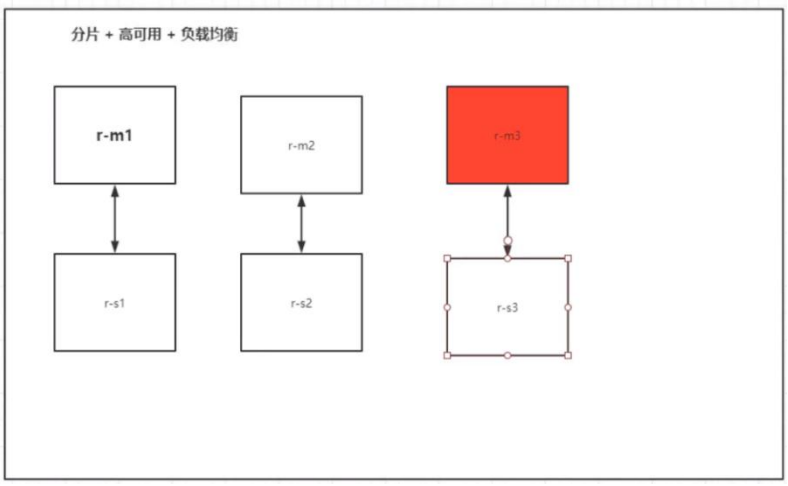
```
# 测试两个不同的网络连通  再启动两个 tomcat 使用默认网络，即 docker0
$ docker run -d -P --name tomcat01 tomcat
$ docker run -d -P --name tomcat02 tomcat
# 此时 ping 不通
```

```
# 要将 tomcat01 连通 tomcat-net-01 ，连通就是将 tomcat01 加到 mynet 网络
# 一个容器两个 ip（tomcat01）
```

```
# 01 连通 ，加入后此时，已经可以 tomcat01 和 tomcat-01-net ping 通了
# 02 是依旧不通的
```

结论：假设要跨网络操作别人，就需要使用 docker network connect 连通！

实战：部署 Redis 集群



```
# 创建网卡
docker network create redis --subnet 172.38.0.0/16
# 通过脚本创建六个 redis 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
cluster-announce-bus-port 16379
appendonly yes
EOF
done

# 通过脚本运行六个 redis
for port in $(seq 1 6);\
docker run -p 637${port}:6379 -p 1667${port}:16379 --name redis-${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
docker exec -it redis-1 /bin/sh #redis 默认没有 bash
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
```

docker 搭建 redis 集群完成!

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传

(img-i60F3UA4-1589635687509)(C:/Users/xiao/Pictures/md/image-20200516203010672.p

ng)]

我们使用 docker 之后，所有的技术都会慢慢变得简单起来！

SpringBoot 微服务打包 Docker 镜像

1、构建 SpringBoot 项目

2、打包运行

```
mvn package
```

3、编写 dockerfile

```
FROM java:8
COPY *.jar /app.jar
CMD ["--server.port=8080"]
EXPOSE 8080
ENTRYPOINT ["java","-jar","app.jar"]
```

4、构建镜像

```
# 1.复制 jar 和 DockerFIle 到服务器
# 2.构建镜像
$ docker build -t xxxxx:xx  .
```

5、发布运行　以后我们使用了 Docker 之后，给别人交付就是一个镜像即可！