

Docker概述

Docker为什么出现

一款产品从开发到上线，从操作系统，到运行环境，再到应用配置。作为开发+运维之间的协作我们需要

关心很多东西，这也是很多互联网公司都不得不面对的问题，特别是各种版本的迭代之后，不同版本环

境的兼容，对运维人员是极大的考验！

环境配置如此麻烦，换一台机器，就要重来一次，费力费时。很多人想到，能不能从根本上解决问题，

软件可以带环境安装？也就是说，安装的时候，把原始环境一模一样地复制过来。解决开发人员说的“在

我的机器上可正常工作”的问题。

之前在服务器配置一个应用的运行环境，要安装各种软件，就拿一个基本的工程项目的环境来说吧，Java/Tomcat/MySQL/JDBC驱动包等。安装和配置这些东西有多麻烦就不说了，它还不能跨平台。假如我们是在 Windows 上安装的这些环境，到了 Linux 又得重新装。况且就算不跨操作系统，换另一台同样操作系统的服务器，要移植应用也是非常麻烦的。

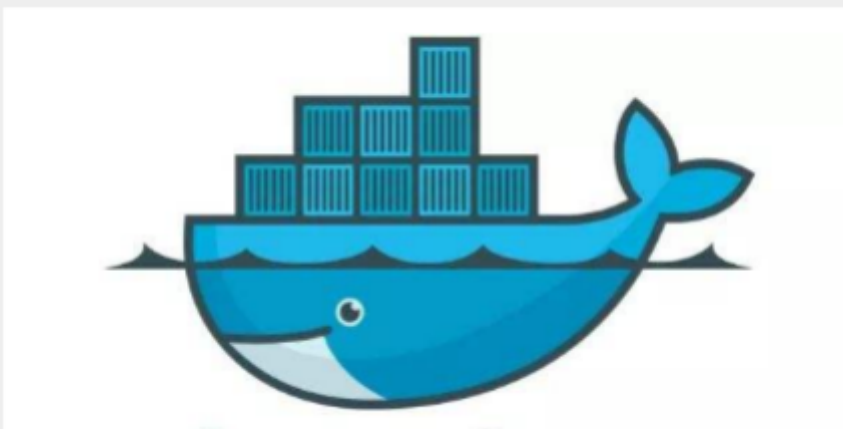
传统上认为，软件编码开发/测试结束后，所产出的成果即是程序或是能够编译执行的二进制字节码文件

等（Java为例）。而为了让这些程序可以顺利执行，开发团队也得准备完整的部署文件，让运维团队得以部署应用程式，开发需要清楚的告诉运维部署团队，用的全部配置文件+所有软件环境。不过，即便如此，

仍然常常发生部署失败的状况。

Docker之所以发展如此迅速，也是因为它对此给出了一个标准化的解决方案。

Docker镜像的设计，使得Docker得以打破过去「程序即应用」的观念。通过Docker镜像（images）将应用程序所需要的系统环境，由下而上打包，达到应用程序跨平台间的无缝接轨运作。



Docker的思想来自于集装箱，集装箱解决了什么问题？在一艘大船上，可以把货物规整的摆放起来。并

且各种各样的货物被集装箱标准化了，集装箱和集装箱之间不会互相影响。那么我就不需要专门运送水

果的船和专门运送化学品的船了。只要这些货物在集装箱里封装的好好的，那我就可以用一艘大船把

他
们都运走。

docker就是类似的理念。

docker 理念

Docker是基于Go语言实现的云开源项目。

Docker的主要目标是通过对应用组件的封装、分发、部署、运行等生命周期的管理，使用户的APP（可以是一个WEB应用或数据库应用等等）及其运行环境能够做到“一次封装，到处运行”。

Linux 容器技术的出现就解决了这样一个问题，而 Docker 就是在它的基础上发展过来的。将应用运行在

Docker 容器上面，而 Docker 容器在任何操作系统上都是一致的，这就实现了跨平台、跨服务器。只需

要一次配置好环境，换到别的机子上就可以一键部署好，大大简化了操作。

Docker能干嘛

之前的虚拟机技术

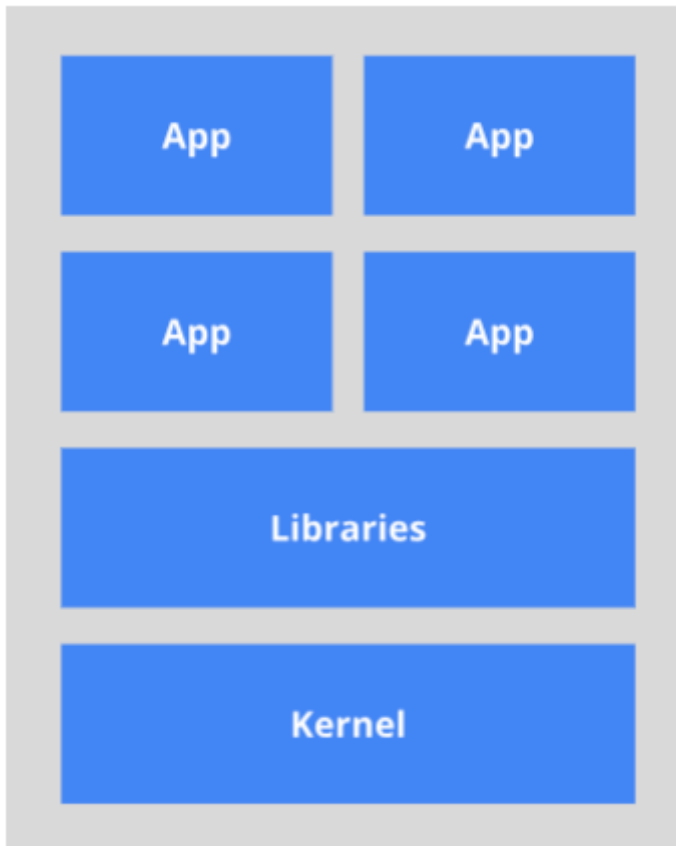
虚拟机（virtual machine）就是带环境安装的一种解决方案。

它可以在一种操作系统里面运行另一种操作系统，比如在Windows 系统里面运行Linux 系统。应用程序

对此毫无感知，因为虚拟机看上去跟真实系统一模一样，而对于底层系统来说，虚拟机就是一个普通文

件，不需要了就删掉，对其他部分毫无影响。这类虚拟机完美的运行了另一套系统，能够使应用程序，

操作系统和硬件三者之间的逻辑不变。

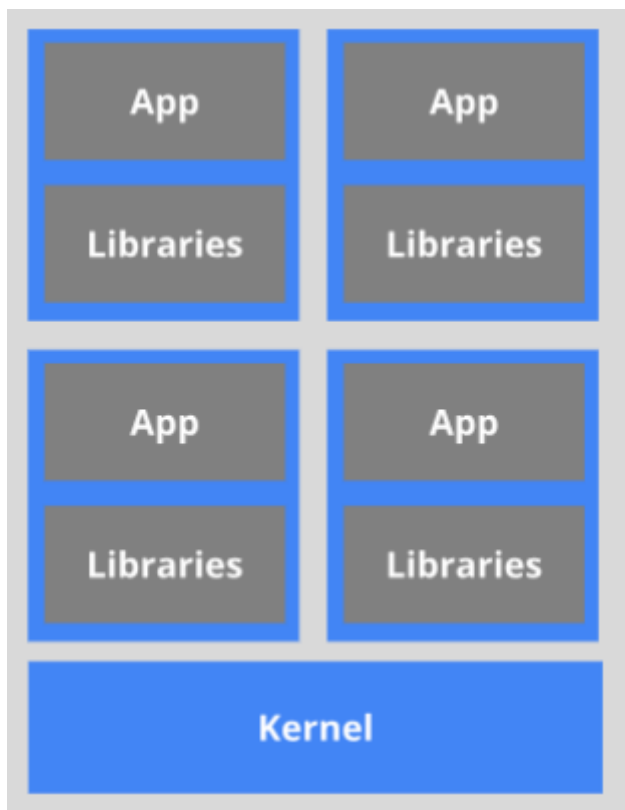


- 虚拟机的缺点：
 - 1、资源占用多
 - 2、冗余步骤多
 - 3、启动慢

容器虚拟化技术

由于前面虚拟机存在这些缺点，Linux 发展出了另一种虚拟化技术：Linux 容器（Linux Containers，缩写为 LXC）。

Linux 容器不是模拟一个完整的操作系统，而是对进程进行隔离。有了容器，就可以将软件运行所需的所有资源打包到一个隔离的容器中。容器与虚拟机不同，不需要捆绑一整套操作系统，只需要软件工作所需的库资源和设置。系统因此而变得高效轻量并保证部署在任何环境中的软件都能始终如一地运行。



比较了 Docker 和传统虚拟化方式的不同之处：

- 传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程；
- 而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。

- 每个容器之间互相隔离，每个容器有自己的文件系统，容器之间进程不会相互影响，能区分计算资源。

开发/运维（DevOps）

更快速的应用交付和部署：

传统的应用开发完成后，需要提供一堆安装程序和配置说明文档，安装部署后需根据配置文档进行繁杂的配置才能正常运行。Docker化之后只需要交付少量容器镜像文件，在正式生产环境加载镜像并运行即可，应用安装配置在镜像里已经内置好，大大节省部署配置和测试验证时间。

更便捷的升级和扩缩容：

随着微服务架构和Docker的发展，大量的应用会通过微服务方式架构，应用的开发构建将变成搭乐高积木一样，每个Docker容器将变成一块“积木”，应用的升级将变得非常容易。当现有的容器不足以支撑业务处理时，可通过镜像运行新的容器进行快速扩容，使应用系统的扩容从原先的天级变成分钟级甚至秒级。

更简单的系统运维：

应用容器化运行后，生产环境运行的应用可与开发、测试环境的应用高度一致，容器会将应用程序相关的环境和状态完全封装起来，不会因为底层基础架构和操作系统的 inconsistency 给应用带来影响，产生新的BUG。当出现程序异常时，也可以通过测试环境的相同容器进行快速定位和修复。

更高效的计算资源利用：

Docker是内核级虚拟化，其不像传统的虚拟化技术一样需要额外的Hypervisor [管理程序] 支持，所以在一台物理机上可以运行很多个容器实例，可大大提升物理服务器的CPU和内存的利用率。

学习途径

Docker官网：<http://www.docker.com>

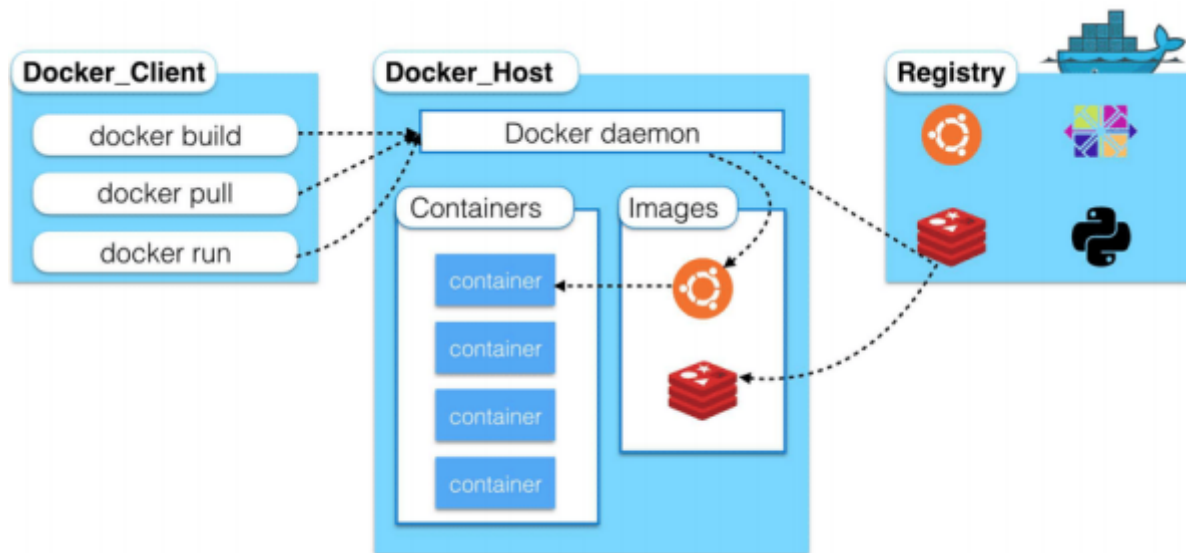
Docker中文网站：<https://www.docker-cn.com>

Docker Hub官网：<https://hub.docker.com>（仓库）

Docker安装

Docker的基本组成

Docker的架构图



镜像（image）：

Docker 镜像（Image）就是一个只读的模板。镜像可以用来创建 Docker 容器，一个镜像可以创建很多容器。就好比 Java 中的 类和对象，类就是镜像，容器就是对象！

容器（container）：

Docker 利用容器（Container）独立运行的一个或一组应用。容器是用镜像创建的运行实例。

它可以被启动、开始、停止、删除。每个容器都是相互隔离的，保证安全的平台。

可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。

仓库（repository）：

仓库（Repository）是集中存放镜像文件的场所。

仓库（Repository）和仓库注册服务器（Registry）是有区别的。仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。

仓库分为公开仓库（Public）和私有仓库（Private）两种形式。

最大的公开仓库是 Docker Hub(<https://hub.docker.com/>)，存放了数量庞大的镜像供用户下载。

国内的公开仓库包括阿里云、网易云等

小结：

- Docker 本身是一个容器运行载体或称之为管理引擎。我们把应用程序和配置依赖打包好形成一个可交付的运行环境，这个打包好的运行环境就似乎 image 镜像文件。只有通过这个镜像文件才能生成 Docker 容器。image 文件可以看作是容器的模板。Docker 根据 image 文件生成容器的实例。同一个 image 文件，可以生成多个同时运行的容器实例。
- image 文件生成的容器实例，本身也是一个文件，称为镜像文件。

- 一个容器运行一种服务，当我们需要的时候，就可以通过docker客户端创建一个对应的运行实例，也就是我们的容器
- 至于仓库，就是放了一堆镜像的地方，我们可以把镜像发布到仓库中，需要的时候从仓库中拉下来就可以了。

安装步骤

1、官网安装参考手册：<https://docs.docker.com/engine/install/centos/>

2、确定CentOS7版本

3、yum安装gcc相关环境（需要确保虚拟机可以上外网）

```
yum -y install gcc
yum -y install gcc-c++
```

4、卸载旧版本

```
yum remove docker \
           docker-client \
           docker-client-latest \
           docker-common \
           docker-latest \
           docker-latest-logrotate \
           docker-logrotate \
           docker-engine
```

5、安装需要的软件包

```
yum install -y yum-utils
```

6、设置镜像仓库

```
# 错误
yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-
ce.repo
## 报错
[Errno 14] curl#35 - TCP connection reset by peer
[Errno 12] curl#35 - Timeout

# 正确推荐使用国内的
yum-config-manager --add-repo http://mirrors.aliyun.com/docker-
ce/linux/centos/docker-ce.repo
```

7、更新yum软件包索引

```
yum makecache fast
```

8、安装 Docker CE

```
yum install docker-ce docker-ce-cli containerd.io
```

9、启动 Docker

```
systemctl start docker
```

10、测试命令

```
docker version  
docker run hello-world  
docker images
```

```
[root@kuangshen ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	fce289e99eb9	16 months ago	1.84kB

11、卸载

```
systemctl stop docker  
yum -y remove docker-ce docker-ce-cli containerd.io  
rm -rf /var/lib/docker
```

阿里云镜像加速

- 1、介绍：<https://www.aliyun.com/product/acr>
- 2、注册一个属于自己的阿里云账户(可复用淘宝账号)
- 3、进入管理控制台设置密码，开通
- 4、查看镜像加速器自己的

阿里云

搜索文档、控制台、API、解决方案

容器镜像服务

镜像加速器

默认实例

镜像仓库

命名空间

授权管理

代理器

访问凭证

企业版实例

实例列表

镜像中心

镜像搜索

我的收藏

镜像加速器

加速器

使用加速器可以提升获取Docker官方镜像的速度

加速器地址

https://qiyb9988.mirror.aliyuncs.com

说明

操作文档

Ubuntu

CentOS

Mac

Windows

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://qiyb9988.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

5、配置镜像加速

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

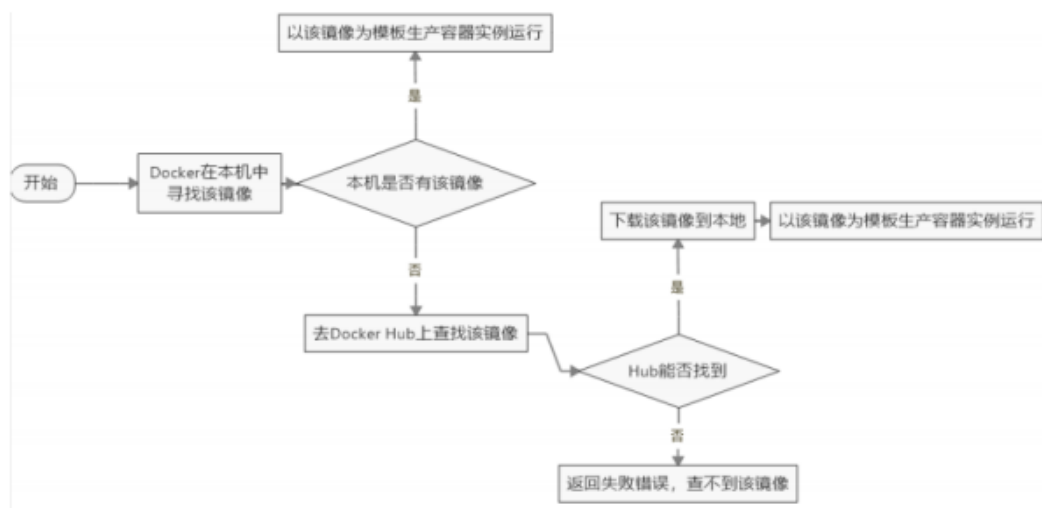
sudo systemctl daemon-reload
sudo systemctl restart docker
```

测试 HelloWorld

1、启动hello-world

```
docker run hello-world
```

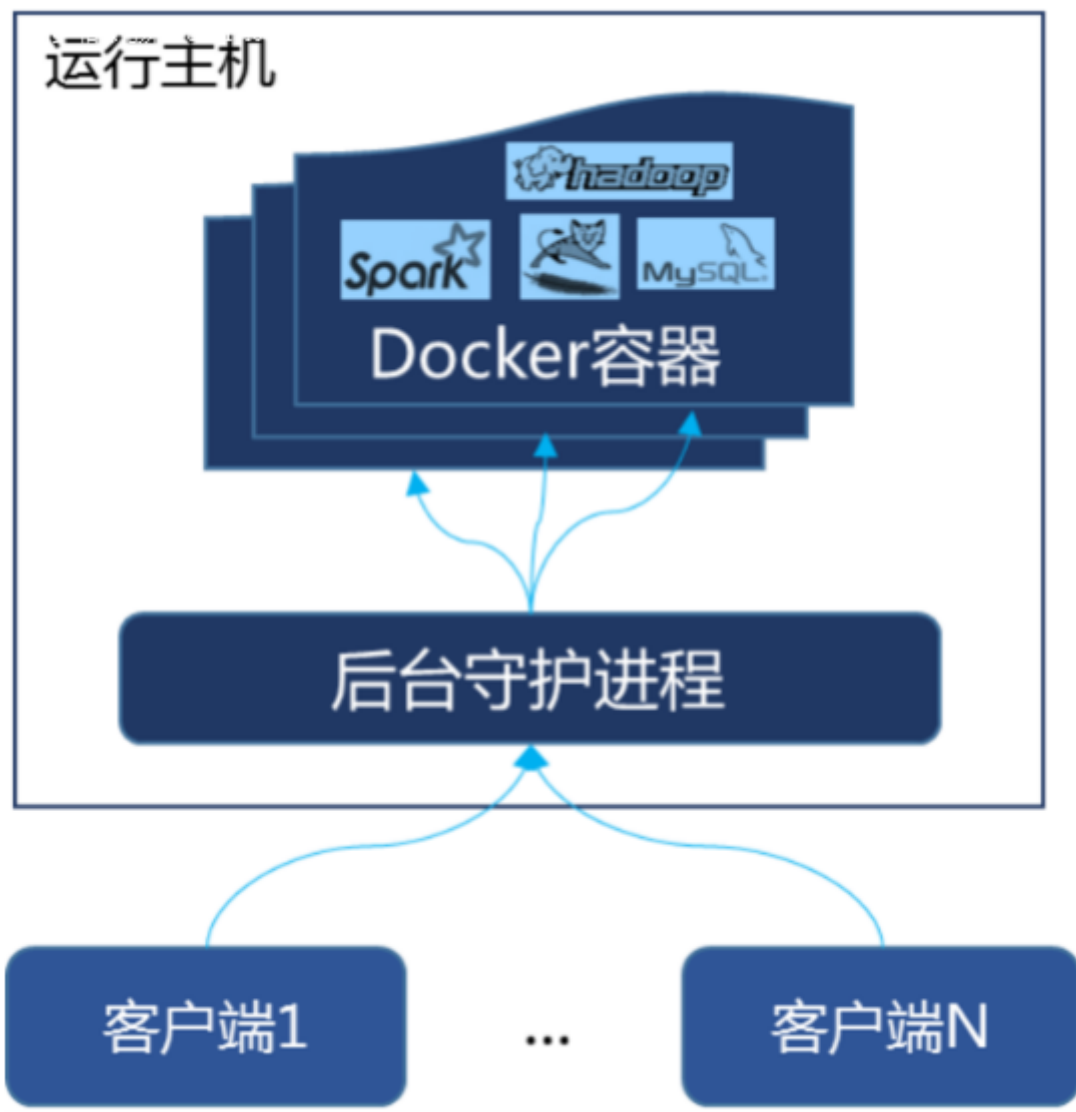
2、run干了什么？



底层原理

Docker是怎么工作的

Docker是一个Client-Server结构的系统，Docker守护进程运行在主机上，然后通过Socket连接从客户端访问，守护进程从客户端接受命令并管理运行在主机上的容器。容器，是一个运行时环境，就是我们前面说到的集装箱。



为什么Docker比较 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。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Guest OS,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,则省略了返个过程,因此新建一个docker容器只需要几秒钟。



Docker常用命令

帮助命令

```
docker version # 显示 Docker 版本信息。
docker info   # 显示 Docker 系统信息，包括镜像和容器数。。
docker --help # 帮助
```

镜像命令

docker images

```
# 列出本地主机上的镜像
[~]# docker images
REPOSITORY      TAG          IMAGE ID      CREATED      SIZE
hello-world     latest      bf756fblae65 4 months ago 13.3kB

# 解释
REPOSITORY  镜像的仓库源
TAG  镜像的标签
IMAGE ID  镜像的ID
CREATED  镜像创建时间
SIZE  镜像大小

# 同一个仓库源可以有多个 TAG，代表这个仓库源的不同版本，我们使用REPOSITORY: TAG 定义不同 的镜像，如果你不定义镜像的标签版本，docker将默认使用 latest 镜像！

# 可选项
-a:  列出本地所有镜像
-q:  只显示镜像id
--digests:  显示镜像的摘要信息
```

docker search

```
# 搜索镜像
[~]# docker search mysql

NAME                DESCRIPTION                                STARS          OFFICIAL
mysql              MySQL is a widely used, open-source relation...  9484          [OK]

# docker search 某个镜像的名称 对应DockerHub仓库中的镜像

# 可选项
--filter=stars=50 :  列出收藏数不小于指定值的镜像。
```

docker pull

```
# 下载镜像
[~]#
docker pull mysql
Using default tag: latest # 不写tag，默认是latest
latest: Pulling from library/mysql
54fec2fa59d0: Already exists # 分层下载
bcc6c6145912: Already exists
951c3d959c9d: Already exists
05de4d0e206e: Already exists
319f0394ef42: Already exists
d9185034607b: Already exists
013a9c64dadcd: Already exists
42f3f7d10903: Pull complete
c4a3851d9207: Pull complete
82a1cc65c182: Pull complete
a0a6b01efa55: Pull complete
bca5ce71f9ea: Pull complete
Digest:
```

```
sha256:61a2a33f4b8b4bc93b7b6b9e65e64044aaec594809f818aefbfff69a893d1944 # 签名
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest # 真实位置
# 指定版本下载
[~]# docker pull mysql:5.7
```

docker rmi

```
docker rmi -f 镜像id # 删除单个
docker rmi -f 镜像名:tag 镜像名:tag # 删除多个
docker rmi -f $(docker images -qa) # 删除全部
```

容器命令

说明：有镜像才能创建容器，我们这里使用 centos 的镜像来测试，就是虚拟一个 centos

```
docker pull centos
```

新建容器并启动

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

```
# 常用参数说明
--name="Name" # 给容器指定一个名字
-d # 后台方式运行容器，并返回容器的id!
-i # 以交互模式运行容器，通过和 -t 一起使用
-t # 给容器重新分配一个终端，通常和 -i 一起使用
-P # 随机端口映射（大写）
-p # 指定端口映射（小结），一般可以有四种写法
    ip:hostPort:containerPort
    ip::containerPort
    hostPort:containerPort （常用）
    containerPort
```

```
# 测试
[~]# docker images
REPOSITORY TAG IMAGE ID          CREATED      SIZE
centos latest 470671670cac 3 months ago 237MB
```

```
# 使用centos进行用交互模式启动容器，在容器内执行/bin/bash命令！
[~]# docker run -it centos /bin/bash
[root@66e5fcd006a3 /]# ls
bin dev etc home lib lib64 lost+found media mnt opt proc root run
sbin srv sys tmp usr var
[root@66e5fcd006a3 /]#

[root@66e5fcd006a3 /]# exit # 使用 exit 退出容器
exit
[~]#
```

列出所有运行的容器

```
# 命令
docker ps [OPTIONS]

# 常用参数说明
-a # 列出当前所有正在运行的容器 + 历史运行过的容器
-l # 显示最近创建的容器
-n=? # 显示最近n个创建的容器
-q # 静默模式，只显示容器编号。
```

退出容器

```
exit # 容器停止退出
ctrl+P+Q # 容器不停止退出
```

启动停止容器

```
docker start (容器id or 容器名) # 启动容器
docker restart (容器id or 容器名) # 重启容器
docker stop (容器id or 容器名) # 停止容器
docker kill (容器id or 容器名) # 强制停止容器
```

删除容器

```
docker rm 容器id # 删除指定容器
docker rm -f $(docker ps -a -q) # 删除所有容器
docker ps -a -q|xargs docker rm # 删除所有容器
```

常用其他命令

后台启动容器

```
# 命令
docker run -d 容器名

# 例子
docker run -d centos # 启动centos，使用后台方式启动

# 问题： 使用docker ps 查看，发现容器已经退出了！
# 解释：Docker容器后台运行，就必须有一个前台进程，容器运行的命令如果不是那些一直挂起的命令，就会自动退出。
# 比如，你运行了nginx服务，但是docker前台没有运行应用，这种情况下，容器启动后，会立即退出，因为他觉得没有程序了，所以最好的情况是，将你的应用使用前台进程的方式运行启动。
```

查看日志

```
# 命令
docker logs -f -t --tail 容器id

# 例子：我们启动 centos，并编写一段脚本来测试玩玩！最后查看日志
[~]# docker run -d centos /bin/sh -c "while true;do echo EINTR;sleep 1;done"
[~]# docker ps
CONTAINER ID        IMAGE
383b7e25c731        centos

# -t 显示时间戳
# -f 打印最新的日志
# --tail 数字 显示多少条！
[~]# docker logs -f 383b7e25c731

EINTR
EINTR
EINTR
EINTR
EINTR
EINTR
EINTR
EINTR
EINTR
^C
```

查看容器中运行的进程信息，支持 ps 命令参数。

```
# 命令
docker top 容器id

# 测试
[~]# docker top 383b7e25c731
UID      PID      PPID     C  STIME TTY  TIME      CMD
root     922721   922702   0  19:07 ?    00:00:00 /bin/sh -c ....
```

查看容器/镜像的元数据

```
# 命令
docker inspect 容器id

# 测试
[~]# docker inspect 383b7e25c731
[
  {
    "Id": "383b7e25c7310f45436d056e147144867291987a450fcd6790147fddbf0c69b4",
    "Created": "2023-08-08T11:07:33.669076455Z",
    "Path": "/bin/sh",
    "Args": [
      "-c",
      "while true;do echo  EINTR;sleep 1;done"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 922721,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2023-08-08T11:07:33.959210551Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image":
      "sha256:5d0da3dc976460b72c77d94c8a1ad043720b0416bfc16c52c45d4847e53fadb6",
      .....
  ]
```

进入正在运行的容器

```
# 命令1
docker exec -it 容器id /bin/bash

# 测试1
[~]# docker ps
CONTAINER ID  IMAGE                COMMAND              CREATED        STATUS PORTS NAMES
```



```

383b7e25c731 centos "/bin/sh -c 'while t..." 12 minutes ago Up 12 minutes
happy_chaum
[~]# docker exec -it 383b7e25c731 /bin/bash
docker exec -it 383b7e25c731 /bin/bash

[root@383b7e25c731 /]# ps
    PID TTY          TIME CMD
    687 pts/0        00:00:00 bash
    704 pts/0        00:00:00 ps
[root@383b7e25c731 /]# ps -ef
UID            PID     PPID  C  STIME TTY          TIME CMD
root             1         0  0  11:07 ?            00:00:00 /bin/sh -c while true;do echo
EINTR;sleep 1;done
root           687         0  0  11:18 pts/0        00:00:00 /bin/bash
root           706         1  0  11:18 ?            00:00:00 /usr/bin/coreutils --
coreutils-prog-shebang=sleep /usr/bin/sleep 1
root           707        687  0  11:18 pts/0        00:00:00 ps -ef
[root@383b7e25c731 /]#

# 命令2
docker attach 容器id

# 测试2
[~]# docker exec -it 383b7e25c731 /bin/bash
[root@383b7e25c731 /]# ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 08:43 ? 00:00:00 /bin/sh -c while true;do echo kuangshen;sleep
root 856 0 0 08:57 pts/0 00:00:00 /bin/bash
root 874 1 0 08:57 ? 00:00:00 /usr/bin/coreutils -- coreutils-prog-shebang=s
root 875 856 0 08:57 pts/0 00:00:00 ps -ef

# 区别
# exec 是在容器中打开新的终端，并且可以启动新的进程
# attach 直接进入容器启动命令的终端，不会启动新的进程

```

从容器内拷贝文件到主机上

```

# 命令
docker cp 容器id:容器内路径 目的主机路径

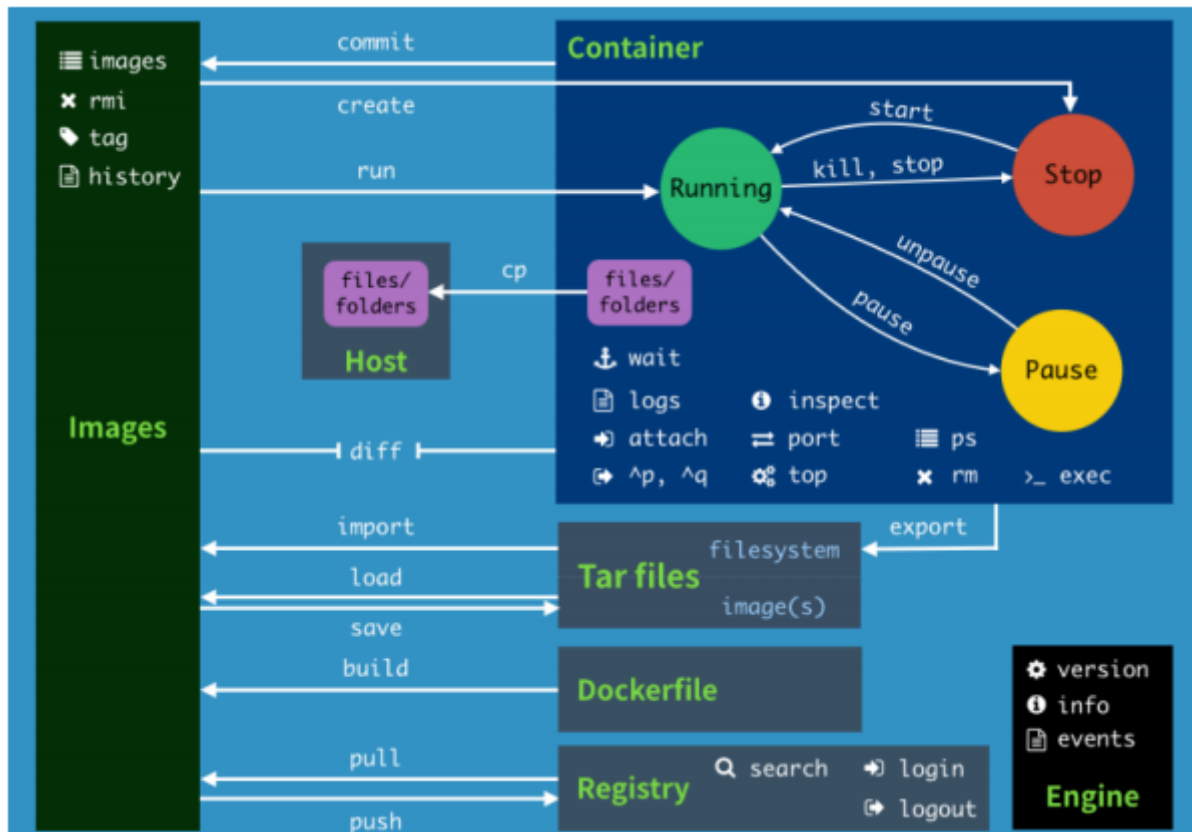
# 测试
# 容器内执行，创建一个文件测试
[root@383b7e25c731 /]# cd /home
[root@383b7e25c731 home]# touch f1
[root@383b7e25c731 home]# ls
f1
[root@383b7e25c731 home]# exit exit

# linux复制查看，是否复制成功
[~]# docker cp 383b7e25c731:/home/f1 /home
[~]# cd /home

```

```
[home]# ls
fl
```

小结



常用命令

命令	解释	翻译
attach	Attach to a running container	当前 shell 下 连接指定运行镜像
build	Build an image from a Dockerfile	通过 Dockerfile 定制镜像
commit	Create a new image from a container changes	提交当前容器为新的镜像
cp	Copy files/folders from the containers filesystem to the host path	从容器中拷贝指定文件或者目录到宿主机中
create	Create a new container	创建一个新的容器，同 run，但不启动容器
diff	Inspect changes on a container's filesystem	查看 docker 容器变化
events	Get real time events from the server	从 docker 服务获取容器实时事件

命令	解释	翻译
exec	Run a command in an existing container	在已存在的容器上运行命令
export	Stream the contents of a container as a tar archive	导出容器的内容流作为一个 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	列出系统当前镜像
import	Create a new filesystem image from the contents of a tarball	从 tar包中的内容创建一个新的文件系统映像[对应export]
info	Display system-wide information	显示系统相关信息
inspect	Return low-level information on a container	查看容器详细信息
kill	Kill a running container	kill 指定 docker 容器
load	Load an image from a tar archive	从一个 tar 包中加载一个镜像[对应 save]
login	Register or Login to the docker registry server	注册或者登陆一个 docker 源服务器
logout	Log out from a Docker registry server	从当前 Docker registry 退出
logs	Fetch the logs of a container	输出当前容器日志信息
port	Lookup the public-facing port which is NAT-ed to PRIVATE_PORT	查看映射端口对应的容器内部源端口
pause	Pause all processes within a container	暂停容器
ps	List containers	列出容器列表
pull	Pull an image or a repository from the docker registry server	从docker镜像源服务器拉取指定镜像或者库镜像
push	Push an image or a repository to the docker registry server	推送指定镜像或者库镜像至docker源服务器
restart	Restart a running container	重启运行的容器
rm	Remove one or more containers	移除一个或者多个容器

命令	解释	翻译
rmi	Remove one or more images	移除一个或多个镜像[无容器使用该 镜像才可删除，否则需删除相关容器才可继续或 -f 强制删除]
run	Run a command in a new container	创建一个新的容器并运行 一个命令
save	Save an image to a tar archive	保存一个镜像为一个 tar 包[对应 load]
search	Search for an image on the Docker Hub	在 docker hub 中搜 索镜像
start	Start a stopped containers	启动容器
stop	Stop a running containers	停止容器
tag	Tag an image into a repository	给源中镜像打标签
top	Lookup the running processes of a container	查看容器中运行的进程信 息
unpause	Unpause a paused container	取消暂停容器
version	Show the docker version information	查看 docker 版本号
wait	Block until a container stops, then print its exit code	截取容器停止时的退出状态值

实例：使用Docker 安装 Nginx

1、搜索镜像

```
[~]# docker search nginx
NAME      DESCRIPTION                               STARS  OFFICIAL
nginx     Official build of Nginx.                13159  [OK]
```

2、拉取镜像

```
[~]# docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
54fec2fa59d0: Pull complete
4ede6f09aeef: Pull complete
f9dc69acb465: Pull complete
Digest:
sha256:86ae264c3f4acb99b2dee4d0098c40cb8c46dcf9e1148f05d3a51c4df6758c12
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

3、启动容器

```
[~]# docker images
REPOSITORY TAG          IMAGE ID      CREATED        SIZE
diytomcat  latest    ffd6529937d  3 hours ago   636MB
```

```
nginx      latest 602e111c06b6 2 weeks ago 127MB
centos     latest 470671670cac 3 months ago 237MB
[~]# docker run -d --name mynginx -p 3500:80 nginx
a95d5f2f057fc609082cfa0de906bd690f95c43a26d38420d081f0e255b232ec
[~]# docker ps
CONTAINER ID IMAGE PORTS NAMES
a95d5f2f057f nginx 0.0.0.0:3500->80/tcp mynginx
```

4、测试访问

```
[~]# curl localhost:3500
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

5、进入容器

```
[~]# docker exec -it mynginx /bin/bash
root@a95d5f2f057f:/# whereis nginx # 寻找nginx
nginx: /usr/sbin/nginx /usr/lib/nginx /etc/nginx /usr/share/nginx
root@a95d5f2f057f:/# cd /usr/share/nginx # nginx 的路径
root@a95d5f2f057f:/usr/share/nginx# ls
html
root@a95d5f2f057f:/usr/share/nginx# cd html # 首页的位置
root@a95d5f2f057f:/usr/share/nginx/html# ls
50x.html index.html
root@a95d5f2f057f:/usr/share/nginx/html# cat index.html
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
```

```
width: 35em;
margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
</body>
</html>
```

使用docker安装 tomcat

```
# 官方文档解释
# -it : 交互模式
# --rm: 容器启动成功并退出以后容器就自动移除，一般在测试情况下使用！
docker run -it --rm tomcat:9.0

# 1、下载tomcat镜像
docker pull tomcat

# 2、启动
docker run -d -p 8080:8080 --name tomcat9 tomcat

# 3、进入tomcat
docker exec -it tomcat9 /bin/bash
```

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。

Docker镜像加载原理

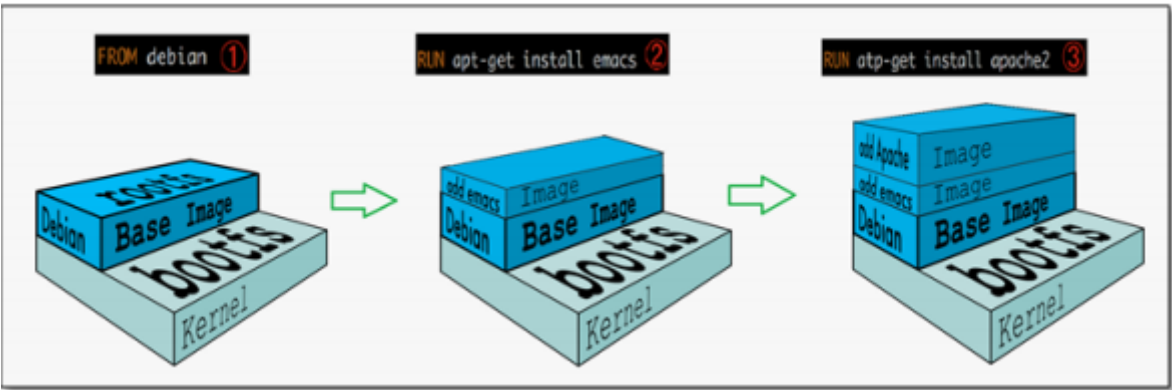
UnionFS（联合文件系统）

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虚拟文件系统下(unite several directories into a single virtual filesystem)。Union 文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。特性：一次同时加载多个文件系统，但从外面看起来，只能看到一个文件系统，联合加载会把各层文件系统叠加起来，这样最终的文件系统会包含所有底层的文件和目录

Docker镜像加载原理

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

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

```
[root@kuangshen home]# docker images centos
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
centos        latest    470671670cac   3 months ago   237MB
```

对于一个精简的OS，rootfs 可以很小，只需要包含最基本的命令，工具和程序库就可以了，因为底层直接用Host的kernel，自己只需要提供rootfs就可以了。由此可见对于不同的linux发行版, bootfs基本是一致的, 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@kuangshen home]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
54fec2fa59d0: Pull complete
9c94e11103d9: Pull complete
04ab1bfc453f: Pull complete
a22fde870392: Pull complete
def16cac9f02: Pull complete
1604f5999542: Pull complete
Digest: sha256:f7ee67d8d9050357a6ea362e2a7e8b65a6823d9b612bc430d057416788ef6df9
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
```

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

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

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！

理解：

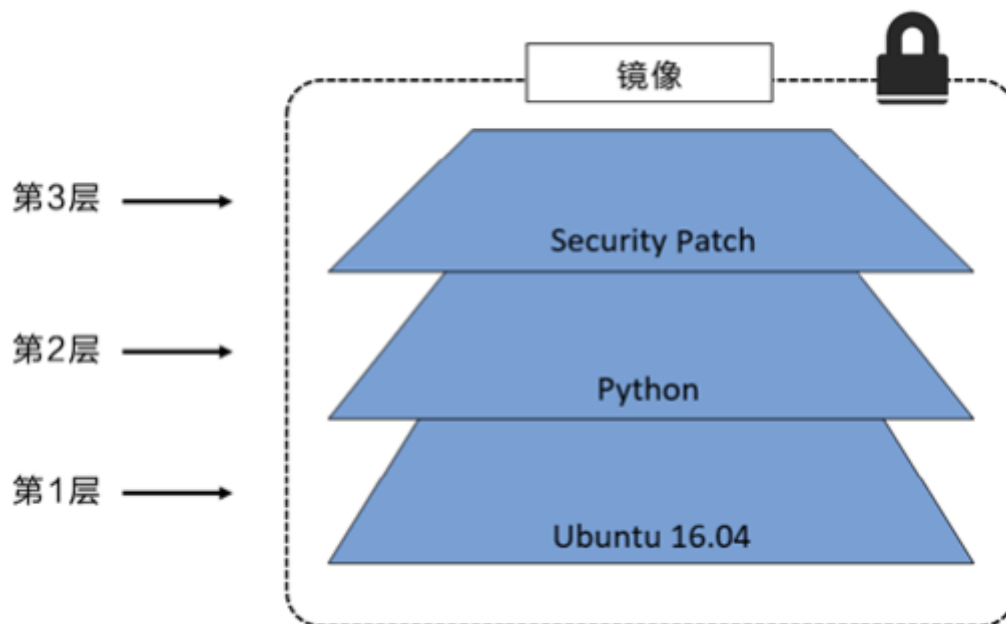
所有的 Docker 镜像都起始于一个基础镜像层，当进行修改或增加新的内容时，就会在当前镜像层之上，创建新的镜像层。

举一个简单的例子，假如基于 Ubuntu Linux 16.04 创建一个新的镜像，这就是新镜像的第一层；如果

在该镜像中添加 Python包，就会在基础镜像层之上创建第二个镜像层；如果继续添加一个安全补丁，就

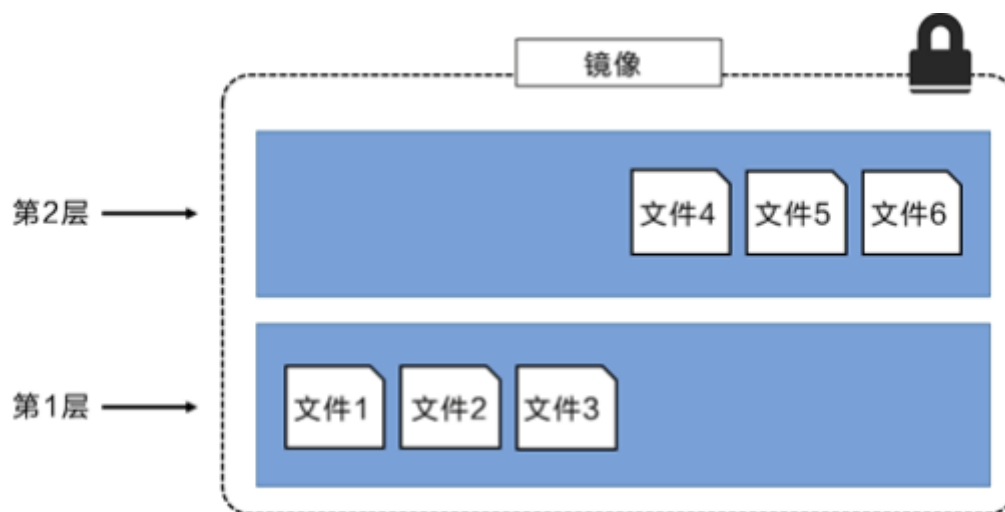
会创建第三个镜像层。

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



在添加额外的镜像层的同时，镜像始终保持是当前所有镜像的组合，理解这一点非常重要。下图中举了

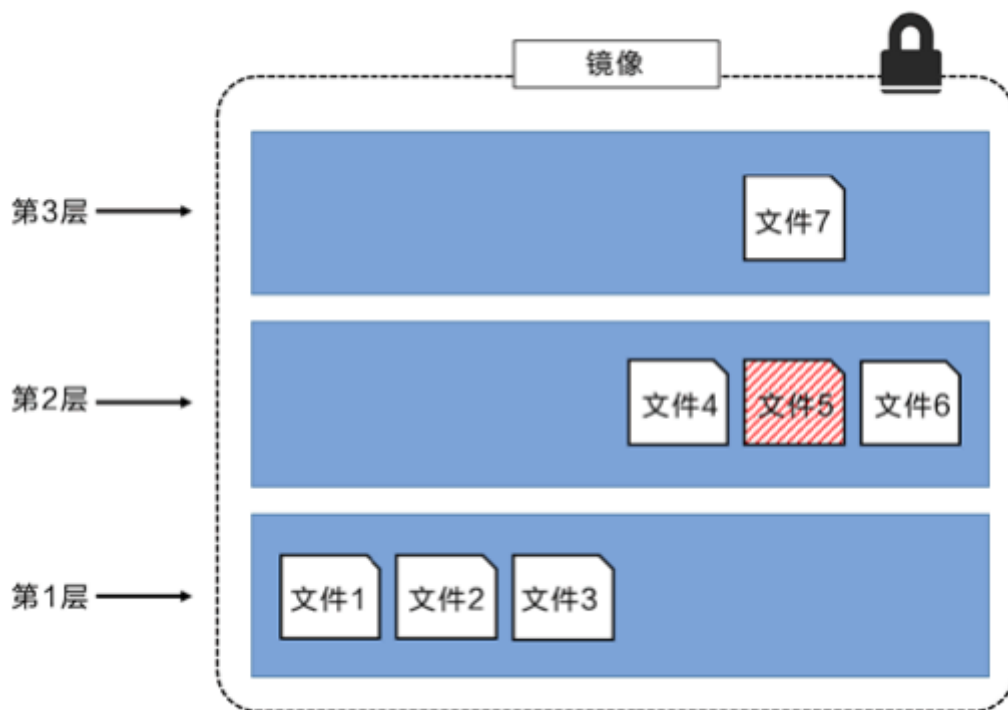
一个简单的例子，每个镜像层包含 3 个文件，而镜像包含了来自两个镜像层的 6 个文件。



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 的一个更新版本。



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

镜像层添加到镜像当中。

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

一的文件系统。

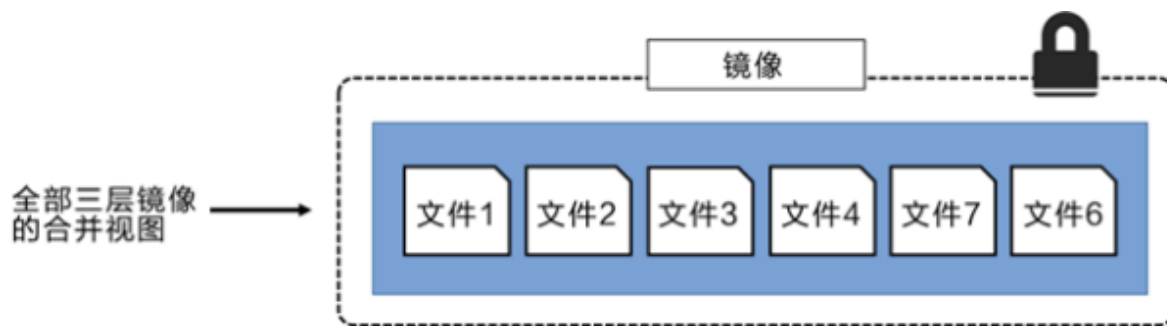
Linux 上可用的存储引擎有 AUFS、Overlay2、Device Mapper、Btrfs 以及 ZFS。顾名思义，每种存储

引擎都基于 Linux 中对应的文件系统或者块设备技术，并且每种存储引擎都有其独有的性能特点。

Docker 在 Windows 上仅支持 windowsfilter 一种存储引擎，该引擎基于 NTFS 文件系统之上实现了分

层和 CoW[1]。

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



特点

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

这一层就是我们通常说的容器层，容器之下的都叫镜像层！

镜像Commit

docker commit 从容器创建一个新的镜像。

docker commit 提交容器副本使之成为一个新的镜像！

语法

docker commit -m="提交的描述信息" -a="作者" 容器id 要创建的目标镜像名:[标签名]

测试

1、从Docker Hub 下载tomcat镜像到本地并运行 -it 交互终端 -p 端口映射

```
docker run -it -p 8080:8080 tomcat
```

2、删除上一步镜像产生的tomcat容器的文档

```
docker ps # 查看容器id
```

```
docker exec -it 容器id /bin/bash
```

```
/usr/local/tomcat # cd webapps/
```

```
/usr/local/tomcat/webapps # ls -l # 查看是否存在 docs文件夹
```

```
/usr/local/tomcat/webapps # curl localhost:8080/docs/ # 可以看到 docs 返回的内容
```

```
/usr/local/tomcat/webapps # rm -rf docs # 删除它
```

```
/usr/local/tomcat/webapps # curl localhost:8080/docs/ # 再次访问返回 404
```

3、当前运行的tomcat实例就是一个没有docs的容器，我们使用它为模板commit一个没有docs的tomcat新镜像， tomcat02

```
docker ps -l # 查看容器的id
```

注意：commit的时候，容器的名字不能有大写，否则报错：invalid reference format

```
docker commit -a="kuangshen" -m="no tomcat docs" 1e98a2f815b0 tomcat02:1.1  
sha256:cdccd4674f93ad34bf73d9db577a20f027a6d03fd1944dc0e628ee4bf17ec748
```

```
[/]  
# docker images # 查看，我们自己提交的镜像已经OK了！
```

```
REPOSITORY TAG IMAGE ID CREATED
```

```
SIZE
```

```
tomcat02 1 .1 cdccd4674f93 About a minute  
ago 649MB  
redis latest f9b990972689 9 days ago  
104MB  
tomcat latest 927899a31456 2 weeks ago  
647MB  
centos latest 470671670cac 3 months ago  
237MB
```

4、这个时候，我们的镜像都是可以使用的，大家可以启动原来的tomcat，和我们新的tomcat02来 测试看看！

```
[~]# docker run -it -p 8080:8080 tomcat02:1.1
```

如果你想要保存你当前的状态，可以通过commit，来提交镜像，方便使用，类似于 VM 中的快照！

容器数据卷

什么是容器数据卷

docker的理念回顾：

将应用和运行的环境打包形成容器运行，运行可以伴随着容器，但是我们对于数据的要求，是希望能够持久化，就好比，安装一个MySQL，我们不会希望把容器删了，就相当于删库跑路了

我们希望容器之间有可能可以共享数据，Docker容器产生的数据，如果不通过docker commit 生成

新的镜像，使得数据作为镜像的一部分保存下来，那么当容器删除后，数据自然也就没有了！这样是行不通的！

为了能保存数据在Docker中我们就可以使用卷！让数据挂载到我们本地！这样数据就不会因为容器删除而丢失了！

作用：

卷就是目录或者文件，存在一个或者多个容器中，由docker挂载到容器，但不属于联合文件系统，因此

能够绕过 Union File System ，提供一些用于持续存储或共享数据的特性：

卷的设计目的就是数据的持久化，完全独立于容器的生存周期，因此Docker不会在容器删除时删除其挂载的数据卷。

特点：

- 1、数据卷可在容器之间共享或重用数据
- 2、卷中的更改可以直接生效
- 3、数据卷中的更改不会包含在镜像的更新中
- 4、数据卷的生命周期一直持续到没有容器使用它为止

所以：总结一句话：就是容器的持久化，以及容器间的继承和数据共享！

使用数据卷

方式一：容器中直接使用命令来添加

挂载

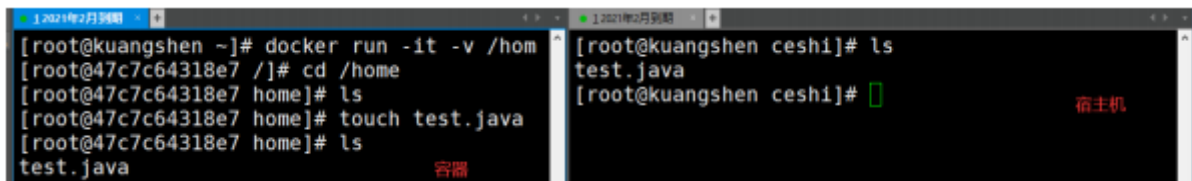
```
# 命令
docker run -it -v 宿主机绝对路径目录:容器内目录 镜像名

# 测试
[~]# docker run -it -v /home/ceshi:/home centos /bin/bash
```

查看数据卷是否挂载成功 docker inspect 容器id

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

测试容器和宿主机之间数据共享：可以发现，在容器中，创建的会在宿主机中看到！



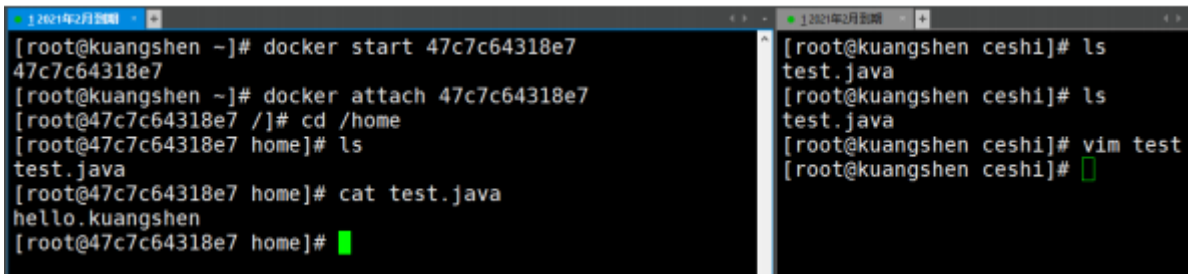
```
[root@kuangshen ~]# docker run -it -v /home
[root@47c7c64318e7 /]# cd /home
[root@47c7c64318e7 home]# ls
[root@47c7c64318e7 home]# touch test.java
[root@47c7c64318e7 home]# ls
test.java
```

```
[root@kuangshen ceshi]# ls
test.java
[root@kuangshen ceshi]#
```

测试容器停止退出后，主机修改数据是否会同步！

- 1. 停止容器
- 2. 在宿主机上修改文件，增加些内容
- 3. 启动刚才停止的容器

- 4. 然后查看对应的文件，发现数据依旧同步！ok



```
[root@kuangshen ~]# docker start 47c7c64318e7
47c7c64318e7
[root@kuangshen ~]# docker attach 47c7c64318e7
[root@47c7c64318e7 /]# cd /home
[root@47c7c64318e7 home]# ls
test.java
[root@47c7c64318e7 home]# cat test.java
hello.kuangshen
[root@47c7c64318e7 home]#

[root@kuangshen ceshi]# ls
test.java
[root@kuangshen ceshi]# ls
test.java
[root@kuangshen ceshi]# vim test
[root@kuangshen ceshi]#
```

使用 docker 安装 mysql

1、搜索镜像

```
[~]# docker search mysql
NAME DESCRIPTION
STARS
mysql MySQL is a widely used, open-source
relation... 9488
```

2、拉取镜像

```
[~]# docker pull mysql:5.7
5.7: Pulling from library/mysql
54fec2fa59d0: Already exists
bcc6c6145912: Pull complete
951c3d959c9d: Pull complete
05de4d0e206e: Pull complete
319f0394ef42: Pull complete
d9185034607b: Pull complete
013a9c64dad0: Pull complete
e745b3361626: Pull complete
03145d87b451: Pull complete
3991a6b182ee: Pull complete
62335de06f7d: Pull complete
Digest:
sha256:e821ca8cc7a44d354486f30c6a193ec6b70a4eed8c8362aeede4e9b8d74b8ebb
Status: Downloaded newer image for mysql:5.7
docker.io/library/mysql:5.7
```

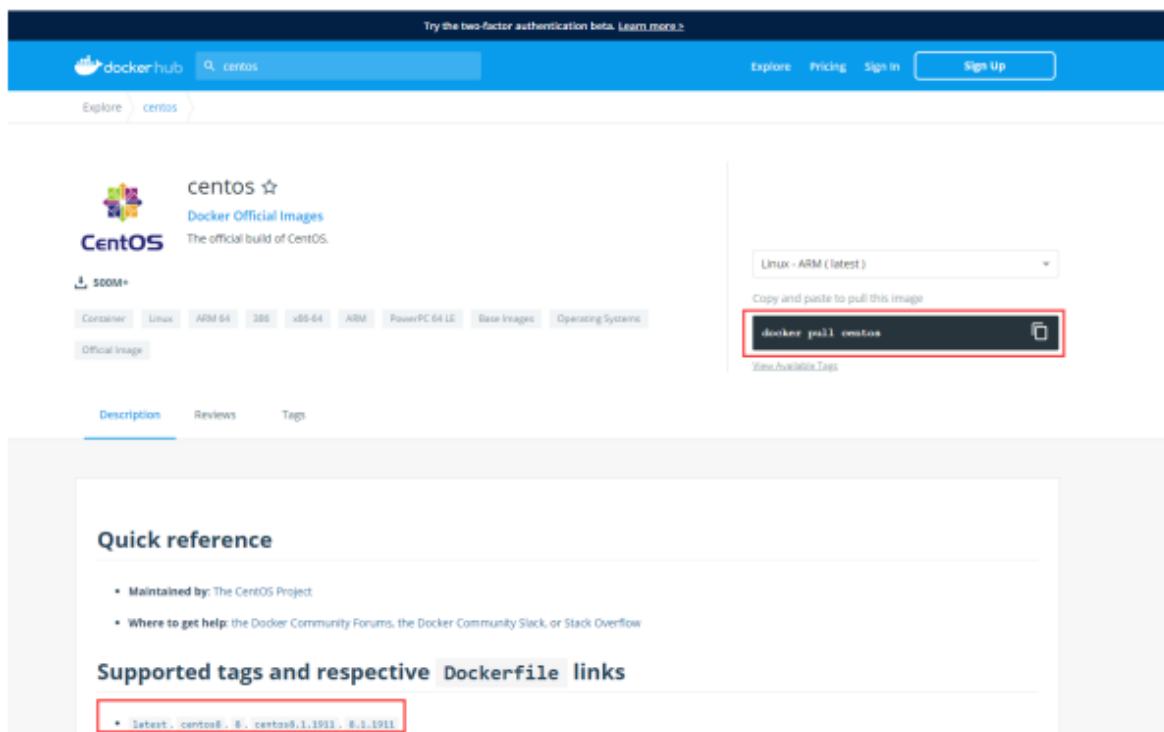
3、启动容器 -e 环境变量

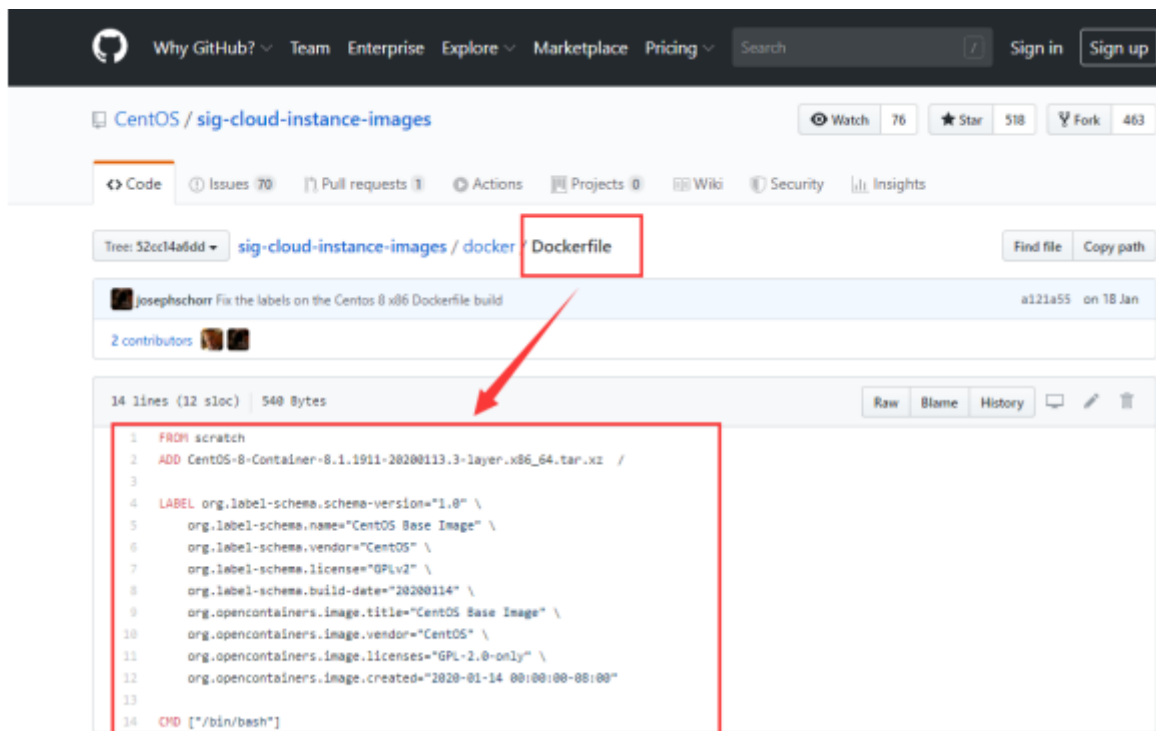
```
[home]# docker run -d -p 3310:3306 -v /home/mysql/conf:/etc/mysql/conf.d -v
/home/eintr/mysql/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 --name mysql01
mysql:5.7
fb06b80faa82e0931d6f0f3d9c3137d1ce5ae6d619d8dea82a4f01c7a8574028
```

4、使用本地的sql yog连接测试一下 3310

5、查看本地的 /home/mysql 目录

```
[data]# pwd
/home/eintr/mysql/data
[data]# ls
auto.cnf      ca.pem          client-key.pem  ib_logfile0  ibdata1  mysql
private_key.pem server-cert.pem  sys
```





DockerFile构建过程

基础知识：

- 1、每条保留字指令都必须为大写字母且后面要跟随至少一个参数
- 2、指令按照从上到下，顺序执行
- 3、# 表示注释
- 4、每条指令都会创建一个新的镜像层，并对镜像进行提交

流程：

- 1、docker从基础镜像运行一个容器
- 2、执行一条指令并对容器做出修改
- 3、执行类似 docker commit 的操作提交一个新的镜像层
- 4、Docker再基于刚提交的镜像运行一个新容器
- 5、执行dockerfile中的下一条指令直到所有指令都执行完成！

说明：

从应用软件的角度来看，DockerFile，docker镜像与docker容器分别代表软件的三个不同阶段。

- DockerFile 是软件的原材料（代码）
- Docker 镜像则是软件的交付品（.apk）
- Docker 容器则是软件的运行状态（客户下载安装执行）

DockerFile 面向开发，Docker镜像成为交付标准，Docker容器则涉及部署与运维，三者缺一不可！



DockerFile：需要定义一个DockerFile，DockerFile定义了进程需要的一切东西。DockerFile涉及的内容

包括执行代码或者是文件、环境变量、依赖包、运行时环境、动态链接库、操作系统的发行版、服务进

程和内核进程（当引用进行需要和系统服务和内核进程打交道，这时需要考虑如何设计 namespace的权

限控制）等等。

Docker镜像：在DockerFile 定义了一个文件之后，Docker build 时会产生一个Docker镜像，当运行

Docker 镜像时，会真正开始提供服务；

Docker容器：容器是直接提供服务的。

DockerFile指令

关键字：

```
FROM # 基础镜像，当前新镜像是基于哪个镜像的
MAINTAINER # 镜像维护者的姓名混合邮箱地址
RUN # 容器构建时需要运行的命令
EXPOSE # 当前容器对外保留出的端口
WORKDIR # 指定在创建容器后，终端默认登录的进来工作目录，一个落脚点
ENV # 用来在构建镜像过程中设置环境变量
ADD # 将宿主机目录下的文件拷贝进镜像且ADD命令会自动处理URL和解压tar压缩包
COPY # 类似ADD，拷贝文件和目录到镜像中！
VOLUME # 容器数据卷，用于数据保存和持久化工作
CMD # 指定一个容器启动时要运行的命令，dockerFile中可以有多个CMD指令，但只有最后一个生效！
ENTRYPOINT # 指定一个容器启动时要运行的命令！和CMD一样
ONBUILD # 当构建一个被继承的DockerFile时运行命令，父镜像在被子镜像继承后，父镜像的ONBUILD被触发
```

Docker File

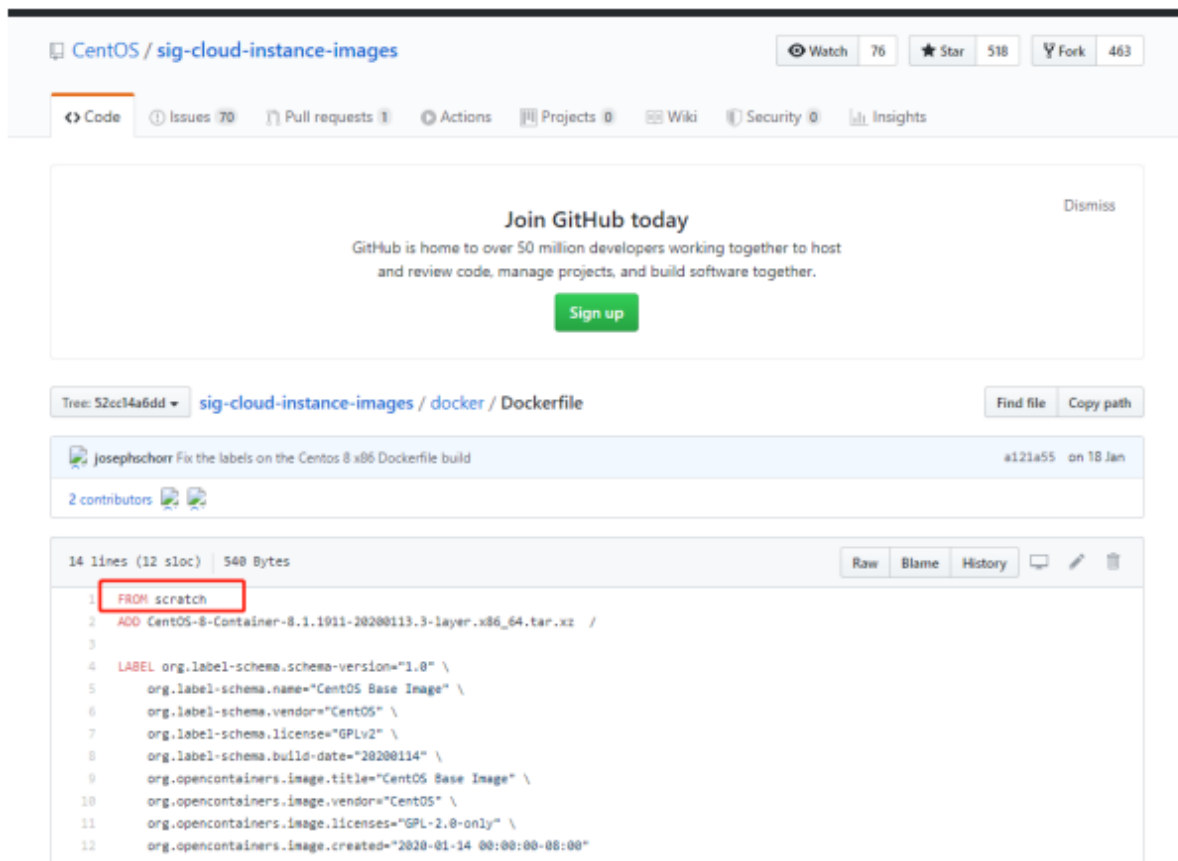
FROM	• 这个镜像的妈妈是谁？（指定基础镜像）
MAINTAINER	• 告诉别人，谁负责养它？（指定维护者信息）
RUN	• 你想让它干啥（在命令前面加上RUN即可）
ADD	• 给它点创业资金（COPY文件，会自动解压）
WORKDIR	• 我是cd,今天刚化了妆（设置当前工作目录）
VOLUME	• 给它一个存放行李的地方（设置卷，挂载主机目录）
EXPOSE	• 它要打开
CMD	• 奔跑吧，兄弟！（指定容器启动后的要干的事情）

Dockerfile

BUILD	Both	RUN
FROM	WORKDIR	CMD
MAINTAINER	USER	ENV
COPY		EXPOSE
ADD		VOLUME
RUN		ENTRYPOINT
ONBUILD		
.dockerignore		

实战测试

Docker Hub 中99% 的镜像都是通过在base镜像（Scratch）中安装和配置需要的软件构建出来的



自定义一个 centos

1、编写DockerFile

查看下官方默认的CentOS的情况：

```
[root@kuangshen ~]# docker images centos
REPOSITORY    TAG       IMAGE ID      CREATED        SIZE
centos        latest    470671670cac  3 months ago  237MB
[root@kuangshen ~]# docker run -it centos
[root@6d224cd97634 /]# pwd
/
[root@6d224cd97634 /]# vim a.txt
bash: vim: command not found
[root@6d224cd97634 /]# ifconfig
bash: ifconfig: command not found
```

Initial centos image default path is /

Default does not support vim

Default does not support ifconfig

目的：使我们自己的镜像具备如下：登陆后的默认路径、vim编辑器、查看网络配置ifconfig支持

准备编写DockerFile文件

```
[home]# mkdir dockerfile-test
[home]# ls
ceshi dockerfile-test docker-test-volume fl
[home]#
[home]# vim mydockerfile-centos # 编辑文件
```

```
[home]# cat centos.dockerfile
FROM centos
MAINTAINER <yixingwei4@gmail.com>
ENV MYPATH /usr/local
WORKDIR $MYPATH

# 添加软件源
RUN cd /etc/yum.repos.d/
RUN sed -i 's/mirrorlist/#mirrorlist/g' /etc/yum.repos.d/CentOS-*
RUN sed -i 's|#baseurl=http://mirror.centos.org|baseurl=http://vault.centos.org|g' /etc/yum.repos.d/CentOS-*

# 安装网络工具包
RUN yum -y install net-tools

EXPOSE 80

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

2、构建

docker build -f dockerfile地址 -t 新镜像名字:TAG .

会看到 **docker build** 命令最后有一个 **.** 表示当前目录

```
[home]# docker build -f centos.dockerfile -t mycentos:0.1 .

DEPRECATED: The legacy builder is deprecated and will be removed in a future
release.

          Install the buildx component to build images with BuildKit:
          https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  3.072kB
Step 1/12 : FROM centos
---> 5d0da3dc9764
Step 2/12 : MAINTAINER <yixingwei4@gmail.com>
---> Running in 5500702375cd
Removing intermediate container 5500702375cd
---> 2434e4089bb4
Step 3/12 : ENV MYPATH /usr/local
---> Running in 1bd5e0e28354
Removing intermediate container 1bd5e0e28354
---> 288e5341adfb
Step 4/12 : WORKDIR $MYPATH
---> Running in 0e0f8e67bed4
Removing intermediate container 0e0f8e67bed4
---> ae504b9b6f9e
Step 5/12 : RUN cd /etc/yum.repos.d/
---> Running in 9b37b75d9959
```

```

Removing intermediate container 9b37b75d9959
---> 8d39335ee324
Step 6/12 : RUN sed -i 's/mirrorlist/#mirrorlist/g' /etc/yum.repos.d/CentOS-*
---> Running in 779de050fef8
Removing intermediate container 779de050fef8
---> c8cb39062bf5
Step 7/12 : RUN sed -i
's|#baseurl=http://mirror.centos.org|baseurl=http://vault.centos.org|g'
/etc/yum.repos.d/CentOS-*
---> Running in 57b571cebaef
Removing intermediate container 57b571cebaef
---> 293347f31a24
Step 8/12 : RUN yum -y install net-tools
---> Running in ec857a6fd893
CentOS Linux 8 - AppStream          502 kB/s | 8.4 MB      00:17
CentOS Linux 8 - BaseOS            228 kB/s | 4.6 MB      00:20
CentOS Linux 8 - Extras            874 B/s | 10 kB       00:12
Dependencies resolved.

=====
Package                Architecture Version                      Repository      Size
=====
Installing:
net-tools              x86_64      2.0-0.52.20160912git.el8     baseos          322 k

Transaction Summary
=====
Install 1 Package

Total download size: 322 k
Installed size: 942 k
Downloading Packages:
net-tools-2.0-0.52.20160912git.el8.x86_64.rpm 27 kB/s | 322 kB      00:11
-----
Total                                           27 kB/s | 322 kB      00:11
warning: /var/cache/dnf/baseos-398269605bdca3dc/packages/net-tools-2.0-
0.52.20160912git.el8.x86_64.rpm: Header V3 RSA/SHA256 Signature, key ID 8483c65d:
NOKEY
CentOS Linux 8 - BaseOS                    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     : net-tools-2.0-0.52.20160912git.el8.x86_64 1/1
  Running scriptlet: net-tools-2.0-0.52.20160912git.el8.x86_64 1/1
  Verifying      : net-tools-2.0-0.52.20160912git.el8.x86_64 1/1

```

```
Installed:
  net-tools-2.0-0.52.20160912git.el8.x86_64

Complete!
Removing intermediate container ec857a6fd893
  ---> b42c5d1e4f26
Step 9/12 : EXPOSE 80
  ---> Running in 2845ad50ccf3
Removing intermediate container 2845ad50ccf3
  ---> adaf7fd46fbf
Step 10/12 : CMD echo $MYPATH
  ---> Running in c64d98bfef0e
Removing intermediate container c64d98bfef0e
  ---> d773370e2060
Step 11/12 : CMD echo "-----end-----"
  ---> Running in 6485e4653732
Removing intermediate container 6485e4653732
  ---> 229e35d25b45
Step 12/12 : CMD /bin/bash
  ---> Running in 460866cd1308
Removing intermediate container 460866cd1308
  ---> 965ddee297e0
Successfully built 965ddee297e0
Successfully tagged mycentos:0.1ma
```

3、运行

```
docker run -it 新镜像名字:TAG

ocker run -it mycentos:0.1
[root@6458fc806947 local]# ls
bin etc games include lib lib64 libexec sbin share src
[root@6458fc806947 local]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.2 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:ac:11:00:02 txqueuelen 0 (Ethernet)
    RX packets 26 bytes 4648 (4.5 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@6458fc806947 local]#
```

可以看到，我们自己的新镜像已经支持 ifconfig 的命令，扩展OK！

4、列出镜像地的变更历史

```
docker history 镜像名
```

```
[root@kuangshen home]# docker history 18888023317c
IMAGE          COMMENT          CREATED          CREATED BY          SIZE
18888023317c   4 minutes ago   /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "/bin... 0B
c5b5391196c7   4 minutes ago   /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "echo... 0B
cd3c78f8143e    4 minutes ago   /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "echo... 0B
dd8bc8a4e724    4 minutes ago   /bin/sh -c #(nop) EXPOSE 80                      0B
3055316b02b9    4 minutes ago   /bin/sh -c yum -y install net-tools                24MB
6f6449a55974    4 minutes ago   /bin/sh -c yum -y install vim                      59.8MB
022384682f07    4 minutes ago   /bin/sh -c #(nop) WORKDIR /usr/local                0B
7a89b945c3a6    4 minutes ago   /bin/sh -c #(nop) ENV MYPATH=/usr/local             0B
9d37c7174860    4 minutes ago   /bin/sh -c #(nop) MAINTAINER kuangshen<2473...    0B
470671670cac    3 months ago    /bin/sh -c #(nop) CMD ["/bin/bash"]                0B
<missing>       3 months ago    /bin/sh -c #(nop) LABEL org.label-schema.sc...    0B
<missing>       3 months ago    /bin/sh -c #(nop) ADD file:aa54047c80ba30064...  237MB
```

CMD 和 ENTRYPOINT 的区别

我们之前说过，两个命令都是指定一个容器启动时要运行的命令

CMD：Dockerfile 中可以有多条 CMD 指令，但只有最后一条生效，CMD 会被 docker run 之后的参数替换！

ENTRYPOINT：docker run 之后的参数会被当做参数传递给 ENTRYPOINT，之后形成新的命令组合！

测试

CMD命令

```
# 1、构建dockerfile
[home]# vim dockerfile-cmd-test
[home]# cat centos_cmd.dockerfile
FROM mycentos:0.1

CMD ["ls", "-a"]

# 2、build 镜像
[home]# docker build -f centos_cmd.dockerfile -t cmdtest:0.1 .
```

DEPRECATED: The legacy builder is deprecated and will be removed in a future release.

Install the buildx component to build images with BuildKit:
<https://docs.docker.com/go/buildx/>

Sending build context to Docker daemon 4.096kB

Step 1/2 : FROM mycentos:0.1

---> 965ddee297e0

Step 2/2 : CMD ["ls", "-a"]

---> Running in f57cdddadf60

Removing intermediate container f57cdddadf60

---> 8f1ced5d9f78

Successfully built 8f1ced5d9f78

Successfully tagged cmdtest:0.1

3、执行

```
[home]# docker run cmdtest:0.1
```

```
.  
..  
bin  
etc  
games  
include  
lib  
lib64  
libexec  
sbin  
share  
src
```

4、如果我们希望用 -l 列表展示信息，我们就需要加上 -l参数

```
[home]# docker run cmdtest:0.1 -l
```

```
docker: Error response from daemon: failed to create task for container: failed to  
create shim task: OCI runtime create failed: runc create failed: unable to start  
container process: exec: "-l": executable file not found in $PATH: unknown.  
ERRO[0000] error waiting for container:
```

问题：我们可以看到可执行文件找不到的报错，executable file not found。

之前我们说过，跟在镜像名后面的是 command，运行时替换 CMD 的默认值。

因此这里的 -l 替换了原来的 CMD，而不是添加在原来的 ls -a 后面。而 -l 根本不是命令，所以自然找不到。

那么如果我们希望加入 -l 这参数，我们就必须重新完整的输入这个命令：

```
docker run cmdtest ls -al
```

ENTRYPOINT命令

1、构建dockerfile

```
[home]# vim dockerfile-entrypoint-test
```

```
[home]# cat dockerfile-entrypoint-test
```

```
FROM centos
```



```
ENTRYPOINT [ "ls", "-a" ]
```

2、build 镜像

```
[home]# docker build -f dockerfile-entrypoint-test -t  
entrypointtest.  
Sending build context to Docker daemon 23 .04kB  
Step 1 /2 : FROM centos  
--> 470671670cac  
Step 2 /2 : ENTRYPOINT [ "ls", "-a" ]  
--> Running in bac4ae055630  
Removing intermediate container bac4ae055630  
--> ae07199f9144  
Successfully built ae07199f9144  
Successfully tagged entrypointtest:latest
```

3、执行

```
[home]# docker run cmdtest:0.2  
.dockerenv  
bin  
dev  
etc  
home  
lib  
lib64  
.....
```

4、测试-l参数，发现可以直接使用，这里就是一种追加，我们可以明显的知道 CMD 和 ENTRYPOINT 的区别了

```
[home]# docker run cmdtest:0.2 -l  
total 52  
drwxr-xr-x 12 root root 4096 Sep 15 2021 .  
drwxr-xr-x  1 root root 4096 Sep 15 2021 ..  
drwxr-xr-x  2 root root 4096 Nov  3 2020 bin  
drwxr-xr-x  2 root root 4096 Nov  3 2020 etc  
drwxr-xr-x  2 root root 4096 Nov  3 2020 games  
drwxr-xr-x  2 root root 4096 Nov  3 2020 include  
drwxr-xr-x  2 root root 4096 Nov  3 2020 lib  
drwxr-xr-x  3 root root 4096 Sep 15 2021 lib64  
drwxr-xr-x  2 root root 4096 Nov  3 2020 libexec  
drwxr-xr-x  2 root root 4096 Nov  3 2020 sbin  
drwxr-xr-x  5 root root 4096 Sep 15 2021 share  
drwxr-xr-x  2 root root 4096 Nov  3 2020 src
```

自定义镜像 tomcat

- 1、mkdir -p kuangshen/build/tomcat
- 2、在上述目录下 touch read.txt
- 3、将JDK和tomcat安装的压缩包拷贝进上一步目录
- 4、在 /kuangshen/build/tomcat 目录下新建一个Dockerfile文件

```

# vim Dockerfile

FROM centos
MAINTAINER kuangshen<24736743@qq.com>
#把宿主机当前上下文的read.txt拷贝到容器/usr/local/路径下
COPY read.txt /usr/local/cincontainer.txt
#把java与tomcat添加到容器中
ADD jdk-8u11-linux-x64.tar.gz /usr/local/
ADD apache-tomcat-9.0.22.tar.gz /usr/local/
#安装vim编辑器
RUN yum -y install vim
#设置工作访问时候的WORKDIR路径，登录落脚点
ENV MYPATH /usr/local
WORKDIR $MYPATH
#配置java与tomcat环境变量
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_BASE /usr/local/apache-tomcat-9.0.22
ENV PATH $PATH:$JAVA_HOME/bin:$CATALINA_HOME/lib:$CATALINA_HOME/bin
#容器运行时监听的端口
EXPOSE 8080
#启动时运行tomcat
# ENTRYPOINT ["/usr/local/apache-tomcat-9.0.22/bin/startup.sh" ]
# CMD ["/usr/local/apache-tomcat-9.0.22/bin/catalina.sh","run"]
CMD /usr/local/apache-tomcat-9.0.22/bin/startup.sh && tail -F
/usr/local/apache-tomcat-9.0.22/bin/logs/catalina.out

```

当前文件状态

```

[root@kuangshen tomcat]# pwd
/home/kuangshen/build/tomcat
[root@kuangshen tomcat]# ll
total 165976
-rw-r--r-- 1 root root 10929702 May 12 12:31 apache-tomcat-9.0.22.tar.gz
-rw-r--r-- 1 root root 1041 May 12 12:41 Dockerfile
-rw-r--r-- 1 root root 159019376 May 12 12:39 jdk-8u11-linux-x64.tar.gz
-rw-r--r-- 1 root root 0 May 12 12:31 read.txt

```

5、构建镜像

```

[tomcat]# docker build -t diytomcat.
.....
Successfully built ffd6f6529937d
Successfully tagged diytomcat:latest # 构建完成

# 查看确定构建完毕！
[tomcat]# docker images
REPOSITORY TAG IMAGE ID CREATED
SIZE
diytomcat latest ffd6f6529937d 20 seconds ago
636MB

```

6、运行启动 run

```
docker run -d -p 8080:8080 -v /tmp/tomcat/test:/usr/local/apache-tomcat-9.0.79/webapps/test -v /tmp/tomcat/logs:/usr/local/apache-tomcat-9.0.79/logs mytomcat:0.1
```

```
[root@kuangshen tomcat]# docker run -d -p 9090:8080 --name mydiytomcat -v /home/kuangshen/build/tomcat/test:/usr/local/apache-tomcat-9.0.22/webapps/test -v /home/kuangshen/build/tomcat/tomcat9logs:/usr/local/apache-tomcat-9.0.22/logs --privileged=true diytomcat
ee46e8476eab3fe8d11ed44c8aff4543d93611b06550238d9aeae91aba2e656e
[root@kuangshen tomcat]# ll
total 165984
-rw-r--r-- 1 root root 10929702 May 12 12:31 apache-tomcat-9.0.22.tar.gz
-rw-r--r-- 1 root root 1041 May 12 12:41 Dockerfile
-rw-r--r-- 1 root root 159019376 May 12 12:39 jdk-8u11-linux-x64.tar.gz
-rw-r--r-- 1 root root 0 May 12 12:31 read.txt
drwxr-xr-x 2 root root 4096 May 12 12:47 test
drwxr-xr-x 2 root root 4096 May 12 12:47 tomcat9logs
```

备注：Docker挂载主机目录Docker访问出现cannot open directory.: Permission denied

解决办法：在挂载目录后多加一个--privileged=true参数即可

7、验证测试访问

Home Documentation Configuration Examples Wiki Mailing Lists Find Help

Apache Tomcat/9.0.79

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:

- [Security Considerations How-To](#)
- [Manager Application How-To](#)
- [Clustering/Session Replication How-To](#)

Developer Quick Start

- [Tomcat Setup](#)
- [First Web Application](#)
- [Realms & AAA](#)
- [JDBC DataSources](#)
- [Examples](#)
- [Servlet Specifications](#)
- [Tomcat Versions](#)

Managing Tomcat

For security, access to the [manager webapp](#) is restricted. Users are defined in:

```
$CATALINA_HOME/conf/tomcat-users.xml
```

In Tomcat 9.0 access to the manager application is split between different users. [Read more...](#)

- [Release Notes](#)
- [Changelog](#)
- [Migration Guide](#)
- [Security Notices](#)

Documentation

- [Tomcat 9.0 Documentation](#)
- [Tomcat 9.0 Configuration](#)
- [Tomcat Wiki](#)

Find additional important configuration information in:

```
$CATALINA_HOME/RUNNING.txt
```

Developers may be interested in:

- [Tomcat 9.0 Bug Database](#)
- [Tomcat 9.0 JavaDocs](#)
- [Tomcat 9.0 Git Repository at GitHub](#)

Getting Help

FAQ and Mailing Lists

The following mailing lists are available:

- [tomcat-announce](#)
Important announcements, releases, security vulnerability notifications. (Low volume).
- [tomcat-users](#)
User support and discussion
- [taglibs-user](#)
User support and discussion for [Apache Taglibs](#)
- [tomcat-dev](#)
Development mailing list, including commit messages

Other Downloads

- [Tomcat Connectors](#)
- [Tomcat Native](#)
- [Taglibs](#)
- [Deployer](#)

Other Documentation

- [Tomcat Connectors](#)
- [mod_jk Documentation](#)
- [Tomcat Native](#)
- [Deployer](#)

Get Involved

- [Overview](#)
- [Source Repositories](#)
- [Mailing Lists](#)
- [Wiki](#)

Miscellaneous

- [Contact](#)
- [Legal](#)
- [Sponsorship](#)
- [Thanks](#)

Apache Software Foundation

- [Who We Are](#)
- [Heritage](#)
- [Apache Home](#)
- [Resources](#)

Copyright ©1999-2023 Apache Software Foundation. All Rights Reserved

8、结合前面学习的容器卷将测试的web服务test发布

```
[root@kuangshen test]# pwd
/home/kuangshen/build/tomcat/test
[root@kuangshen test]# mkdir WEB-INF
[root@kuangshen test]# cd WEB-INF/
[root@kuangshen WEB-INF]# vim web.xml
[root@kuangshen WEB-INF]# cd ..
[root@kuangshen test]# vim a.jsp
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID" version="2.5">

<display-name>test</display-name>

</web-app>
```

a.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>hello, kuangshen</title>
</head>
<body>
-----welcome-----
<%= " my docker tomcat, impact-eintr "%>
<br>
<br>
<% System.out.println("-----my docker tomcat-----");%>
</body>
</html>
```

9、测试



查看日志

```
[tomcat]# cd tomcat9logs/
[tomcat9logs]# ll
total 24
-rw-r----- 6.3k root 18 8月 15:15 catalina.2023-08-18.log
-rw-r----- 6.3k root 18 8月 15:21 catalina.out
-rw-r----- 0 root 18 8月 15:15 host-manager.2023-08-18.log
-rw-r----- 408 root 18 8月 15:15 localhost.2023-08-18.log
-rw-r----- 961 root 18 8月 15:21 localhost_access_log.2023-08-18.txt
-rw-r----- 0 root 18 8月 15:15 manager.2023-08-18.log
[tomcat9logs]# sudo cat catalina.out

[sudo] eintr 的密码:
18-Aug-2023 07:15:56.335 INFO [main]
org.apache.catalina.startup.VersionLoggerListener.log Server version name:
Apache Tomcat/9.0.79
...
18-Aug-2023 07:15:57.001 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["http-nio-8080"]
18-Aug-2023 07:15:57.012 INFO [main] org.apache.catalina.startup.Catalina.start
Server startup in [445] milliseconds
-----my docker tomcat-----
-----my docker tomcat-----
```

Docker-网络讲解

理解Docker0

准备工作：清空所有的容器，清空所有的镜像

```
docker rm -f $(docker ps -a -q) # 删除所有容器
docker rmi -f $(docker images -qa) # 删除全部镜像
```

Docker的网络也是十分重要的一个点，希望大家可以认真理解！

我们先来做个测试

查看本地ip ip addr

```
[root@kuangshen ~]# 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
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:16:3e:30:27:f4 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.138/20 brd 172.17.95.255 scope global dynamic eth0
        valid_lft 310959134sec preferred_lft 310959134sec
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:bb:71:07:06 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global docker0
        valid_lft forever preferred_lft forever
```

这里我们分析可得，有三个网络：

```
lo 127.0.0.1 # 本机回环地址
eth0 172.17.90.138 # 阿里云的私有IP
docker0 172.18.0.1 # docker网桥
# 问题：Docker 是如何处理容器网络访问的？
```

我们之前安装ES的时候，留过一个问题，就是安装Kibana的问题，Kibana得指定ES的地址！或者我们

实际场景中，我们开发了很多微服务项目，那些微服务项目都要连接数据库，需要指定数据库的url地

址，通过ip。但是我们用Docker管理的话，假设数据库出问题了，我们重新启动运行一个，这个时候数

据库的地址就会发生变化，docker会给每个容器都分配一个ip，且容器和容器之间是可以互相访问的。

我们可以测试下容器之间能不能ping通过：

```
# 启动tomcat01
[~]# docker run -d -P --name tomcat01 tomcat

# 查看tomcat01的ip地址，docker会给每个容器都分配一个ip!
[~]# 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
122 : eth0@if123: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
link/ether 02 :42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 172 .18.0.2/16 brd 172 .18.255.255 scope global eth0
valid_lft forever preferred_lft forever

# 思考，我们的linux服务器是否可以ping通容器内的tomcat ?
[~]# ping 172.18.0.2
PING 172 .18.0.2 (172.18.0.2) 56 (84) bytes of data.
64 bytes from 172 .18.0.2: icmp_seq= 1 ttl= 64 time= 0 .070 ms # 可以ping通！
```

原理

1、每一个安装了Docker的linux主机都有一个docker0的虚拟网卡。这是个桥接网卡，使用了veth-pair技术！

```
# 我们再次查看主机的 ip addr
[~]# 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
2 : eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
link/ether 00 :16:3e:30:27:f4 brd ff:ff:ff:ff:ff:ff
inet 172.17.90.138/20 brd 172.17.95.255 scope global dynamic eth0
valid_lft 310954997sec preferred_lft 310954997sec
3 : docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP group default
link/ether 02 :42:bb:71:07:06 brd ff:ff:ff:ff:ff:ff
inet 172.18.0.1/16 brd 172.18.255.255 scope global docker0
valid_lft forever preferred_lft forever
...
95: veth61a5b6c@if94: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master docker0 state UP group default
    link/ether fe:06:99:5f:be:c5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::fc06:99ff:fe5f:bec5/64 scope link proto kernel_ll
    valid_lft forever preferred_lft foreve
# 发现：本来我们有三个网络，我们在启动了个tomcat容器之后，多了一个！ 95 的网络！
```

2、每启动一个容器，linux主机就会多了一个虚拟网卡。

```
# 我们启动了一个tomcat01，主机的ip地址多了一个 123: vethc8584ea@if122
# 然后我们在tomcat01容器中查看容器的ip是 122: eth0@if123

# 我们再启动一个tomcat02观察
[~]# docker run -d -P --name tomcat02 tomcat

# 然后发现linux主机上又多了一个网卡 125: veth021eeea@if124:
# 我们看下tomcat02的容器内ip地址是 124: eth0@if125:
[~]# docker exec -it tomcat02 ip addr

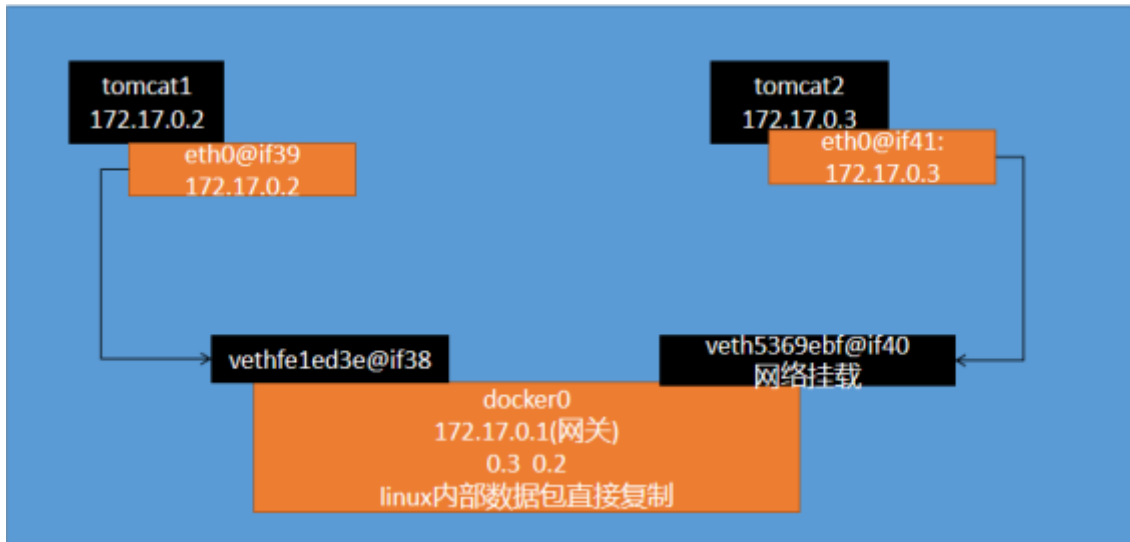
# 观察现象：
# tomcat --- linux主机 vethc8584ea@if122 ---- 容器内 eth0@if123
# tomcat --- linux主机 veth021eeea@if124 ---- 容器内 eth0@if125
# 相信到了这里，大家应该能看出点小猫腻了吧！只要启动一个容器，就有一对网卡

# veth-pair 就是一对的虚拟设备接口，它都是成对出现的。一端连着协议栈，一端彼此相连着。
# 正因为有这个特性，它常常充当着一个桥梁，连接着各种虚拟网络设备！
# “Bridge、OVS 之间的连接”，“Docker 容器之间的连接”等等，以此构建出非常复杂的虚拟网络结构，比如 OpenStack Neutron。
```

3、我们来测试下tomcat01和tomcat02容器间是否可以互相ping通

```
[~]# docker exec -it tomcat02 ping 172.18.0.2
PING 172.18.0.2 (172.18.0.2) 56 (84) bytes of data.
64 bytes from 172.18.0.2: icmp_seq= 1 ttl= 64 time= 0.110 ms
# 结论：容器和容器之间是可以互相访问的。
```

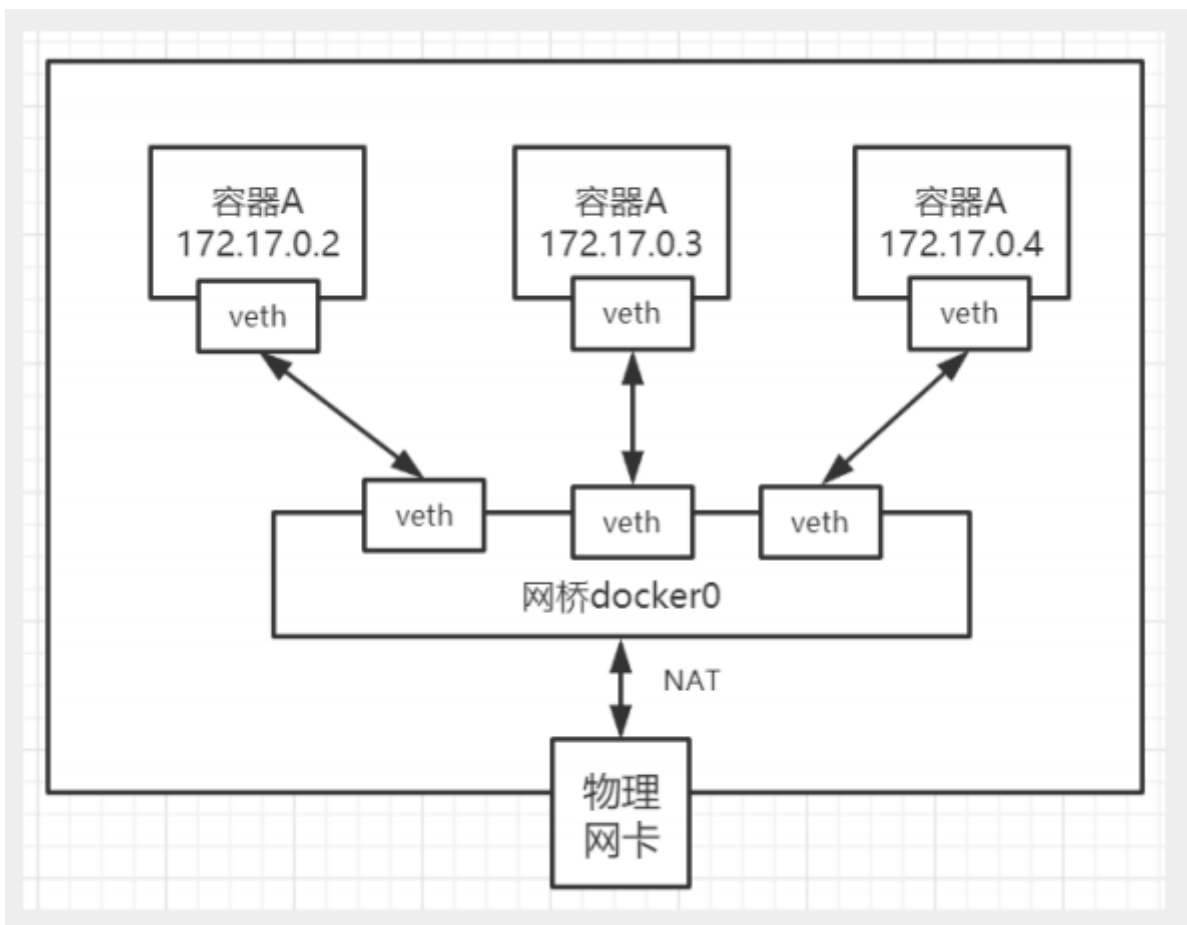
4、我们来画一个网络模型图



结论：tomcat1和tomcat2共用一个路由器。是的，他们使用的一个，就是docker0。任何一个容器启动默认都是docker0网络。
docker默认会给容器分配一个可用ip。

小结

Docker使用Linux桥接，在宿主机虚拟一个Docker容器网桥(docker0)，Docker启动一个容器时会根据Docker网桥的网段分配给容器一个IP地址，称为Container-IP，同时Docker网桥是每个容器的默认网关。因为同一宿主机内的容器都接入同一个网桥，这样容器之间就能够通过容器的Container-IP直接通信。



Docker容器网络就很好的利用了Linux虚拟网络技术，在本地主机和容器内分别创建一个虚拟接口，并让他们彼此联通（这样一对接口叫veth pair）；

Docker中的网络接口默认都是虚拟的接口。虚拟接口的优势就是转发效率极高（因为Linux是在内核中进行数据的复制来实现虚拟接口之间的数据转发，无需通过外部的网络设备交换），对于本地系统和容

器系统来说，虚拟接口跟一个正常的以太网卡相比并没有区别，只是他的速度快很多。

--Link

思考一个场景，我们编写一个微服务，数据库连接地址原来是使用ip的，如果ip变化就不行了，那我们能不能使用服务名访问呢？

jdbc:mysql://mysql:3306，这样的话哪怕mysql重启，我们也不需要修改配置了！docker提供了 --link 的操作！

```
# 我们使用tomcat02，直接通过容器名ping tomcat01，不使用ip
[~]# docker exec -it tomcat02 ping tomcat01
ping: tomcat01: Name or service not known # 发现ping不通
```

```
# 我们再启动一个tomcat03，但是启动的时候连接tomcat02
[~]# docker run -d -P --name tomcat03 --link tomcat02 tomcat
a3a4a17a2b707766ad4f2bb967ce1d94f658cd4cccef3bb8707395cdc71fa1e7

# 这个时候，我们就可以使用tomcat03 ping通tomcat02 了
[~]# docker exec -it tomcat03 ping tomcat02
PING tomcat02 (172.18.0.3) 56 (84) bytes of data.
64 bytes from tomcat02 (172.18.0.3): icmp_seq= 1 ttl= 64 time= 0 .093 ms
64 bytes from tomcat02 (172.18.0.3): icmp_seq= 2 ttl= 64 time= 0 .066 ms

# 再来测试，tomcat03 是否可以ping tomcat01 失败
[~]# docker exec -it tomcat03 ping tomcat01
ping: tomcat01: Name or service not known

# 再来测试，tomcat02 是否可以ping tomcat03 反向也ping不通
[~]# docker exec -it tomcat02 ping tomcat03
ping: tomcat03: Name or service not known
```

思考，这个原理是什么呢？我们进入tomcat03中查看下host配置文件

```
[~]# 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
172.18.0.3 tomcat02 b80da266a3ad # 发现tomcat2直接被写在这里
172.18.0.4 a3a4a17a2b70

# 所以这里其实就是配置了一个 hosts 地址而已！
# 原因：--link的时候，直接把需要link的主机的域名和ip直接配置到了hosts文件中了。
```

--link早都过时了，我们不推荐使用！我们可以使用自定义网络的方式

自定义网络

基本命令查看

命令如下：

```
[root@kuangshen ~]# 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.
```

查看所有网络

```
[~]# docker network ls
NETWORK ID   NAME          DRIVER  SCOPE
4eb2182ac4b2 bridge        bridge  local
ae2b6209c2ab host          host    local
c037f7ec7e57 none          null    local
```

网络模式

网络模式	配置	说明
bridge模式	--net=bridge	默认值，在Docker网桥docker0上为容器创建新的网络栈
none模式	--net=none	不配置网络，用户可以稍后进入容器，自行配置
container模式	--net=container:name/id	容器和另外一个容器共享Network namespace。kubernetes中的pod就是多个容器共享一个Network namespace。
host模式	--net=host	容器和宿主机共享Network namespace
用户自定义	--net=自定义网络	用户自己使用network相关命令定义网络，创建容器的时候可以指定为自己定义的网络

查看一个具体的网络的详细信息

```
# 命令
[~]# docker network inspect eb633065d908
[
  {
    "Name": "es",
    "Id": "eb633065d908a352d74965ec5586ab4ce038779b23184a20cb0d478c403e57cf",
    "Created": "2022-01-07T15:46:27.60870142+08:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
```

```

        "Driver": "default",
        "Options": {},
        "Config": [
            {
                "Subnet": "172.18.0.0/16",
                "Gateway": "172.18.0.1"
            }
        ],
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]

```

自定义网卡

1、删除原来的所有容器

```

[~]# docker rm -f $(docker ps -aq)

# 恢复到了最开始的样子
[~]# 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
2 : eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
link/ether 00 :16:3e:30:27:f4 brd ff:ff:ff:ff:ff:ff
inet 172 .17.90.138/20 brd 172 .17.95.255 scope global dynamic eth0
valid_lft 310951436sec preferred_lft 310951436secc
3 : docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN group default
link/ether 02 :42:bb:71:07:06 brd ff:ff:ff:ff:ff:ff
inet 172 .18.0.1/16 brd 172 .18.255.255 scope global docker0
valid_lft forever preferred_lft forever

```

2、接下来我们来创建容器，但是我们知道默认创建的容器都是docker0网卡的

```
# 默认我们不配置网络，也就相当于默认值 --net bridge 使用的docker0
docker run -d -P --name tomcat01 --net bridge tomcat
```

docker0网络的特点

1. 它是默认的
2. 域名访问不通
3. --link 域名通了，但是删了又不行

3、我们可以让容器创建的时候使用自定义网络

```
[root@kuangshen ~]# docker network create --help

Usage:  docker network create [OPTIONS] NETWORK

Create a network

Options:
  --attachable          Enable manual container attachment
  --aux-address map     Auxiliary IPv4 or IPv6 addresses used by Network driver (default map[])
  --config-from string  The network from which copying the configuration
  --config-only        Create a configuration only network
  -d, --driver string   Driver to manage the Network (default "bridge")
  --gateway strings     IPv4 or IPv6 Gateway for the master subnet
  --ingress            Create swarm routing-mesh network
  --internal           Restrict external access to the network
  --ip-range strings   Allocate container ip from a sub-range
  --ipam-driver string  IP Address Management Driver (default "default")
  --ipam-opt map       Set IPAM driver specific options (default map[])
  --ipv6              Enable IPv6 networking
  --label list         Set metadata on a network
  -o, --opt map        Set driver specific options (default map[])
  --scope string       Control the network's scope
  --subnet strings     Subnet in CIDR format that represents a network segment
```

自定义创建的默认default "bridge"

自定义创建一个网络网络

```
[~]# docker network create --driver bridge --subnet 192.168.0.0/16 --gateway
192.168.0.1 mynet
09bd09d8d3a6b33e6d19f49643dab551e5a45818baf4d5328aa7320c6dcfc236
```

确认下

```
[~]# docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
47e40f310757	bridge	bridge	local
c3a45b0b0e10	host	host	local
d2163fface9b	mynet	bridge	local
d0b8308f2f21	none	null	local

```
[~]# docker network inspect mynet
```

```
[
  {
    "Name": "mynet",
    "Id": "d2163fface9b7722174e506f99f4c9d4e480bbf5b14ab49ed479a01021f105b4",
    "Created": "2023-08-18T19:58:33.191478196+08:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
```

```

    "IPAM": {
        "Driver": "default",
        "Options": {},
        "Config": [
            {
                "Subnet": "192.168.0.0/16",
                "Gateway": "192.168.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
}
]

```

我们来启动两个容器测试，使用自己的 mynet!

```

[~]# docker run -d -p 8081:8080 --net mynet --name tomcat01 mytomcat:0.1
2dbdccb80de1e8b80bb55979d0d671940bd5d5252f8fe2aa31206d3fdabcfdbc
[~]# docker run -d -p 8082:8080 --net mynet --name tomcat02 mytomcat:0.1
66289f37fb274d066224ac6c25e85b54452e1631857396edc8c7fe714b651a50
[~]# docker ps

```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
66289f37fb27	mytomcat:0.1	"/bin/sh -c '/usr/lo..."	5 seconds ago	Up 4 seconds
0.0.0.0:8082->8080/tcp, :::8082->8080/tcp		tomcat02		
2dbdccb80de1	mytomcat:0.1	"/bin/sh -c '/usr/lo..."	39 seconds ago	Up 38 seconds
0.0.0.0:8081->8080/tcp, :::8081->8080/tcp		tomcat01		

再来查看下

```

[~]# docker network inspect mynet
[
    {
        "Name": "mynet",
        "Id": "d2163fface9b7722174e506f99f4c9d4e480bbf5b14ab49ed479a01021f105b4",
        "Created": "2023-08-18T19:58:33.191478196+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "192.168.0.0/16",

```

```

        "Gateway": "192.168.0.1"
    }
}
],
"Internal": false,
"Attachable": false,
"Ingress": false,
"ConfigFrom": {
    "Network": ""
},
"ConfigOnly": false,
"Containers": {
    "2dbdccc80de1e8b80bb55979d0d671940bd5d5252f8fe2aa31206d3fdabcfdbc": {
        "Name": "tomcat01",
        "EndpointID":
"164a4cc8354cdc252e5a79e2986eaf462eba3efa06171ff5007edd0f82460955",
        "MacAddress": "02:42:c0:a8:00:02",
        "IPv4Address": "192.168.0.2/16",
        "IPv6Address": ""
    },
    "66289f37fb274d066224ac6c25e85b54452e1631857396edc8c7fe714b651a50": {
        "Name": "tomcat02",
        "EndpointID":
"499678e229f02b2397644d449d400f25269fbe0bcb2b8415f5b237a6d4786ecc",
        "MacAddress": "02:42:c0:a8:00:03",
        "IPv4Address": "192.168.0.3/16",
        "IPv6Address": ""
    }
},
"Options": {},
"Labels": {}
}
]

```

我们来测试ping容器名和ip试试, 都可以ping通

```
[~]# docker exec -it tomcat01 ping tomcat02
```

```
PING tomcat02 (192.168.0.3) 56(84) bytes of data.
```

```
64 bytes from tomcat02.mynet (192.168.0.3): icmp_seq=1 ttl=64 time=0.076 ms
```

```
64 bytes from tomcat02.mynet (192.168.0.3): icmp_seq=2 ttl=64 time=0.099 ms
```

```
64 bytes from tomcat02.mynet (192.168.0.3): icmp_seq=3 ttl=64 time=0.127 ms
```

```
64 bytes from tomcat02.mynet (192.168.0.3): icmp_seq=4 ttl=64 time=0.072 ms
```

```
^C
```

```
--- tomcat02 ping statistics ---
```

```
4 packets transmitted, 4 received, 0% packet loss, time 3044ms
```

```
rtt min/avg/max/mdev = 0.072/0.093/0.127/0.023 ms
```

```
[~]# docker exec -it tomcat01 ping 192.168.0.3
```

```
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
```

```
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=0.078 ms
```

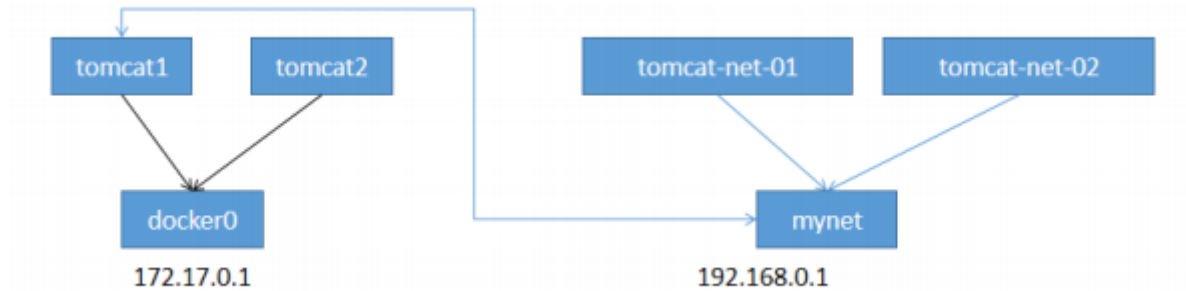
```
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.097 ms
```

```
^C
```

```
--- 192.168.0.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1025ms
rtt min/avg/max/mdev = 0.078/0.087/0.097/0.013 ms
```

发现，我们自定义的网络docker都已经帮我们维护好了对应的关系
所以我们平时都可以这样使用网络，不使用--link效果一样，所有东西实时维护好，直接域名 ping通。

网络连通



docker0和自定义网络肯定不通，我们使用自定义网络的好处就是网络隔离：

大家公司项目部署的业务都非常多，假设我们有一个商城，我们会有订单业务（操作不同数据），会有订单业务购物车业务（操作不同缓存）。如果在一个网络下，有的程序猿的恶意代码就不能防止了，所以我们就在部署的时候网络隔离，创建两个桥接网卡，比如订单业务（里面的数据库，redis，mq，全部业务都在order-net网络下）其他业务在其他网络。

那关键的问题来了，如何让 tomcat01 访问 tomcat1？

启动默认的容器，在docker0网络下

```
[~]# docker run -d -p 8083:8080 --name tomcat1 mytomcat:0.1
2ef46433ed1b29b67ce2045db1f58630dfbdcc228fa4ed009a5e02129bed1a14
[~]# docker run -d -p 8084:8080 --name tomcat2 mytomcat:0.1
28a787ba2d42c14c855b87d852a3dcb785203a177512b641dfa8b7890c95acac
#### # 查看当前的容器
```

```
[~]# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS			NAMES	
28a787ba2d42	mytomcat:0.1	"/bin/sh -c '/usr/lo..."	8 seconds ago	Up 8
seconds	0.0.0.0:8084->8080/tcp, :::8084->8080/tcp		tomcat2	
2ef46433ed1b	mytomcat:0.1	"/bin/sh -c '/usr/lo..."	29 seconds ago	Up 29
seconds	0.0.0.0:8083->8080/tcp, :::8083->8080/tcp		tomcat1	
66289f37fb27	mytomcat:0.1	"/bin/sh -c '/usr/lo..."	9 minutes ago	Up 9
minutes	0.0.0.0:8082->8080/tcp, :::8082->8080/tcp		tomcat02	
2dbdccb80de1	mytomcat:0.1	"/bin/sh -c '/usr/lo..."	9 minutes ago	Up 9
minutes	0.0.0.0:8081->8080/tcp, :::8081->8080/tcp		tomcat01	

```
[~]# docker network --help
```

Commands:

connect Connect a container to a network # 连接一个容器到一个网络

命令 docker network connect [OPTIONS] NETWORK CONTAINER


```
[~]# docker network connect mynet tomcat01
[~]# docker network inspect mynet
[
  {
    "Name": "mynet",
    "Id": "d2163fface9b7722174e506f99f4c9d4e480bbf5b14ab49ed479a01021f105b4",
    "Created": "2023-08-18T19:58:33.191478196+08:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "192.168.0.0/16",
          "Gateway": "192.168.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "2dbdccc80de1e8b80bb55979d0d671940bd5d5252f8fe2aa31206d3fdabcfdbc": {
        "Name": "tomcat01",
        "EndpointID":
"164a4cc8354cdc252e5a79e2986eaf462eba3efa06171ff5007edd0f82460955",
        "MacAddress": "02:42:c0:a8:00:02",
        "IPv4Address": "192.168.0.2/16",
        "IPv6Address": ""
      },
      "2ef46433ed1b29b67ce2045db1f58630dfbdcc228fa4ed009a5e02129bed1a14": {
        "Name": "tomcat1",
        "EndpointID":
"f117818e25ad7932f7a3dfb09a5dbceb71b4d35cd27a933c9df9e8f52d2425d1",
        "MacAddress": "02:42:c0:a8:00:04",
        "IPv4Address": "192.168.0.4/16",
        "IPv6Address": ""
      },
      "66289f37fb274d066224ac6c25e85b54452e1631857396edc8c7fe714b651a50": {
        "Name": "tomcat02",
        "EndpointID":
"499678e229f02b2397644d449d400f25269fbe0bcb2b8415f5b237a6d4786ecc",
        "MacAddress": "02:42:c0:a8:00:03",
        "IPv4Address": "192.168.0.3/16",
        "IPv6Address": ""
      }
    }
  }
]
```

```

    },
    "Options": {},
    "Labels": {}
  }
]

# tomcat1 可以ping通了
[~]# docker exec -it tomcat1 ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=0.115 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.151 ms
64 bytes from 192.168.0.3: icmp_seq=3 ttl=64 time=0.157 ms
^C
--- 192.168.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2018ms
rtt min/avg/max/mdev = 0.115/0.141/0.157/0.018 ms

# tomcat02 依旧ping不通, 大家应该就理解了
[~]# docker exec -it tomcat2 ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
^C
--- 192.168.0.2 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2021ms

```

结论：如果要跨网络操作别人，就需要使用 docker network connect [OPTIONS] NETWORK CONTAINER 连接

发布镜像

DockerHub

注册dockerhub <https://hub.docker.com/signup>，需要有一个账号

```

# 1、查看登录命令
[tomcat]# docker login --help
Usage: docker login [OPTIONS] [SERVER]

# 2、登录
[tomcat]# docker login -u kuangshen
Password:
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-
store
Login Succeeded

# 3、将镜像发布出去
[tomcat]# docker push kuangshen/diytomcat:1.0
The push refers to repository [docker.io/library/diytomcat]

```

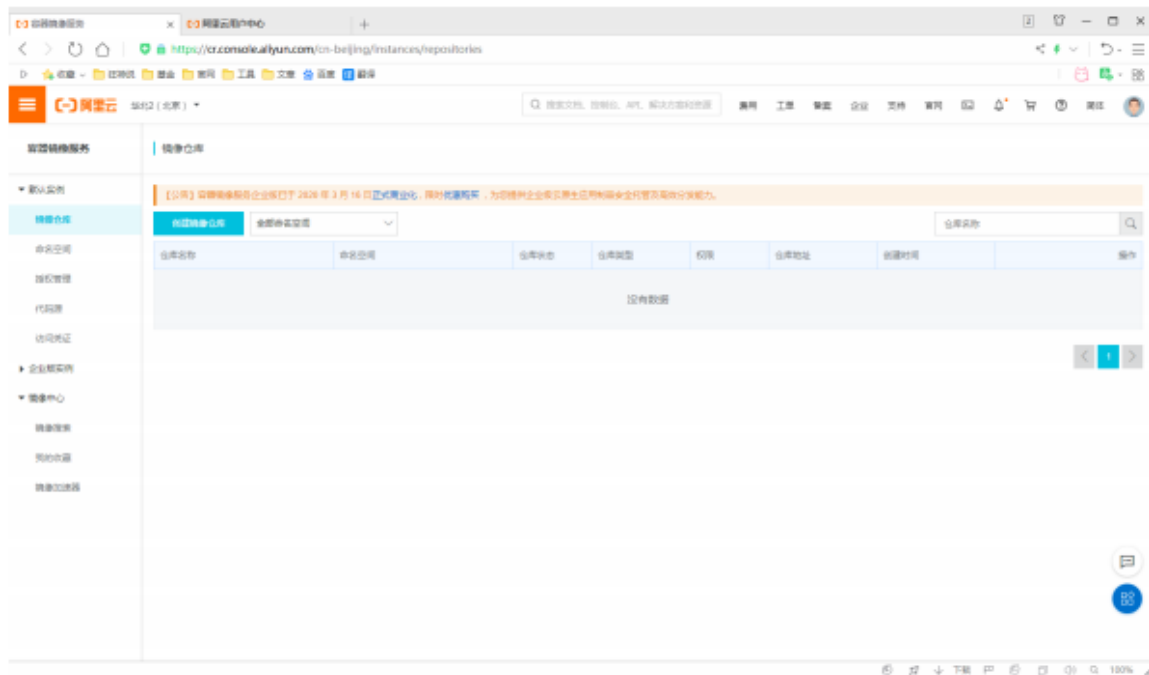
```
0f02399c6fdf: Preparing
e79ea0c3a34e: Preparing
09281fa8fe38: Preparing
b56a902b0aef: Preparing
0683de282177: Preparing
# 拒绝: 请求的资源访问被拒绝
denied: requested access to the resource is denied

# 问题: 本地镜像名无帐号信息, 解决加 tag即可
docker tag 251ca4419332 kuangshen/diytomcat:1.0

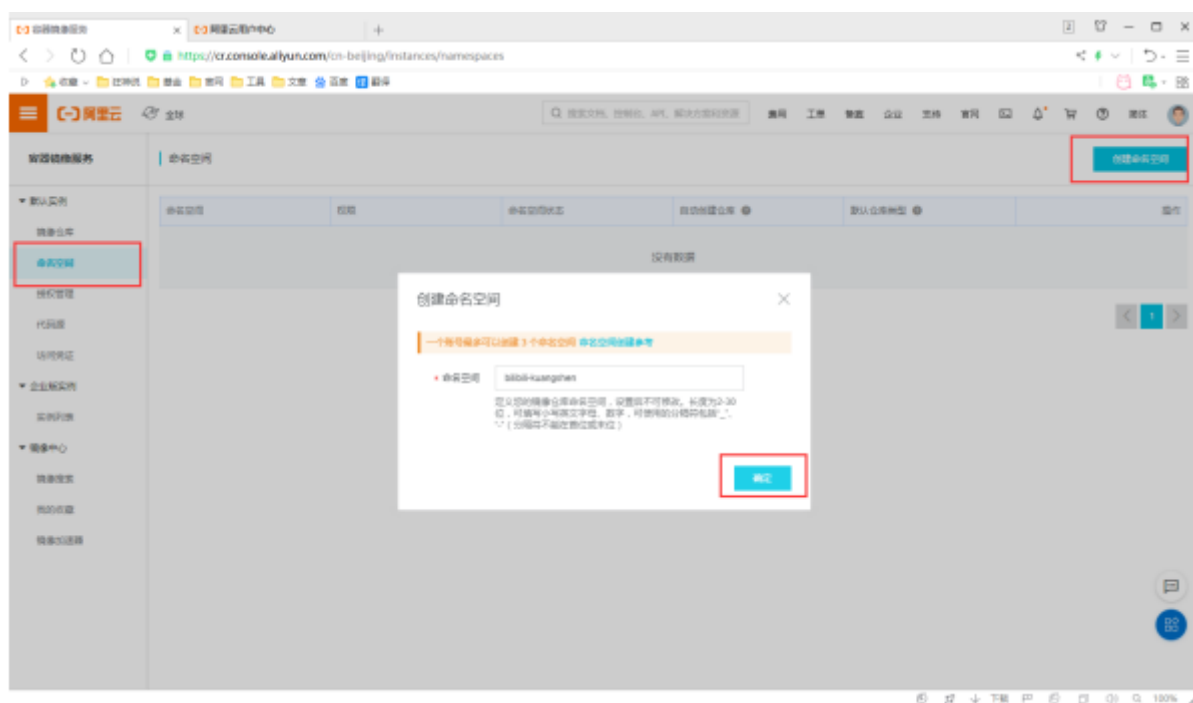
# 再次 push, ok
[tomcat]# docker push kuangshen/diytomcat:1.0
The push refers to repository [docker.io/kuangshen/diytomcat]
0f02399c6fdf: Pushing [=====> ]
   9 .729MB/59.76MB
e79ea0c3a34e: Pushing [=====> ]
   3 .188MB/15.41MB
09281fa8fe38: Pushing [> ]
   3 .823MB/324MB
b56a902b0aef: Pushed
0683de282177: Pushing [=> ]
   5 .997MB/237.1MB
```

阿里云镜像服务

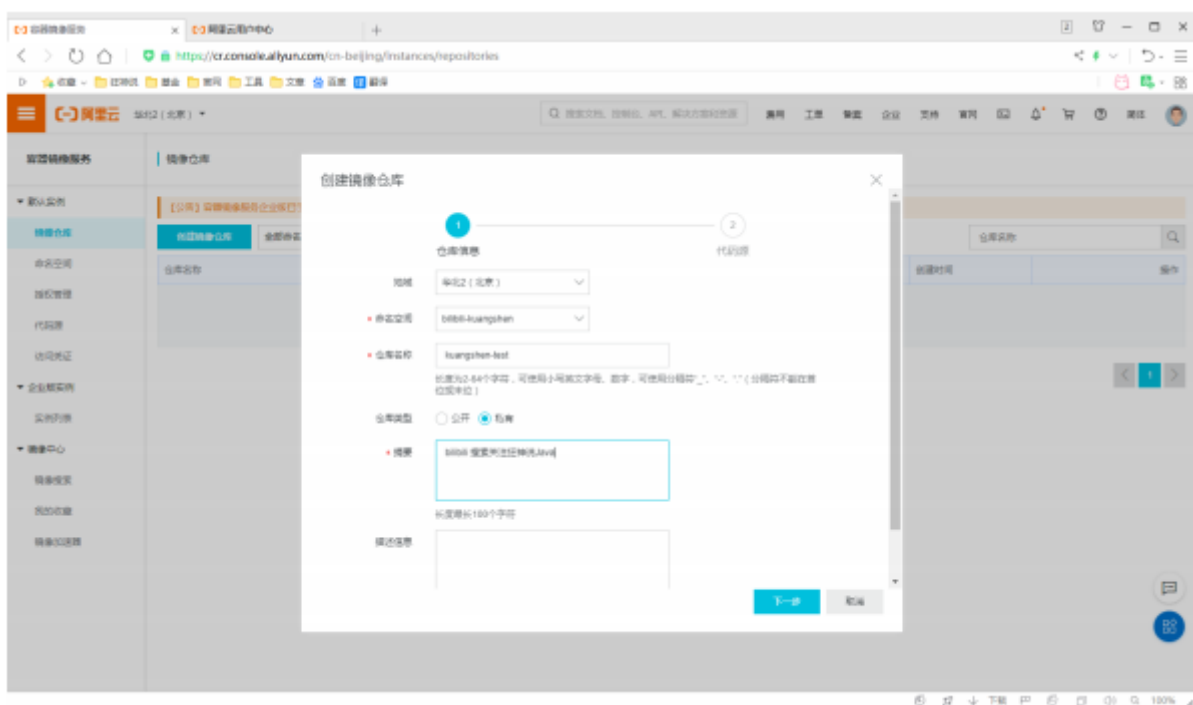
- 1、登录阿里云
- 2、找到容器镜像服务

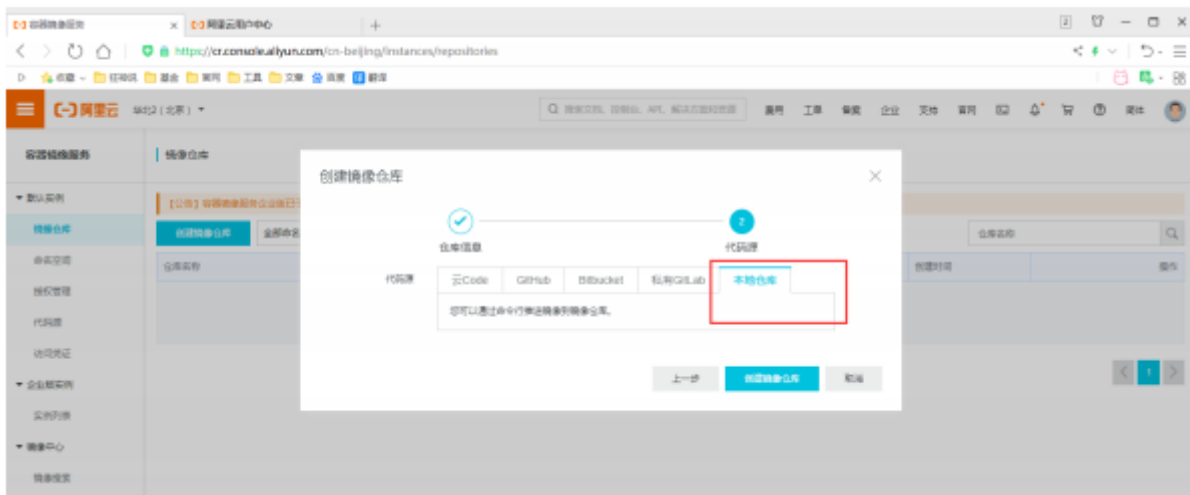


3、创建命名空间

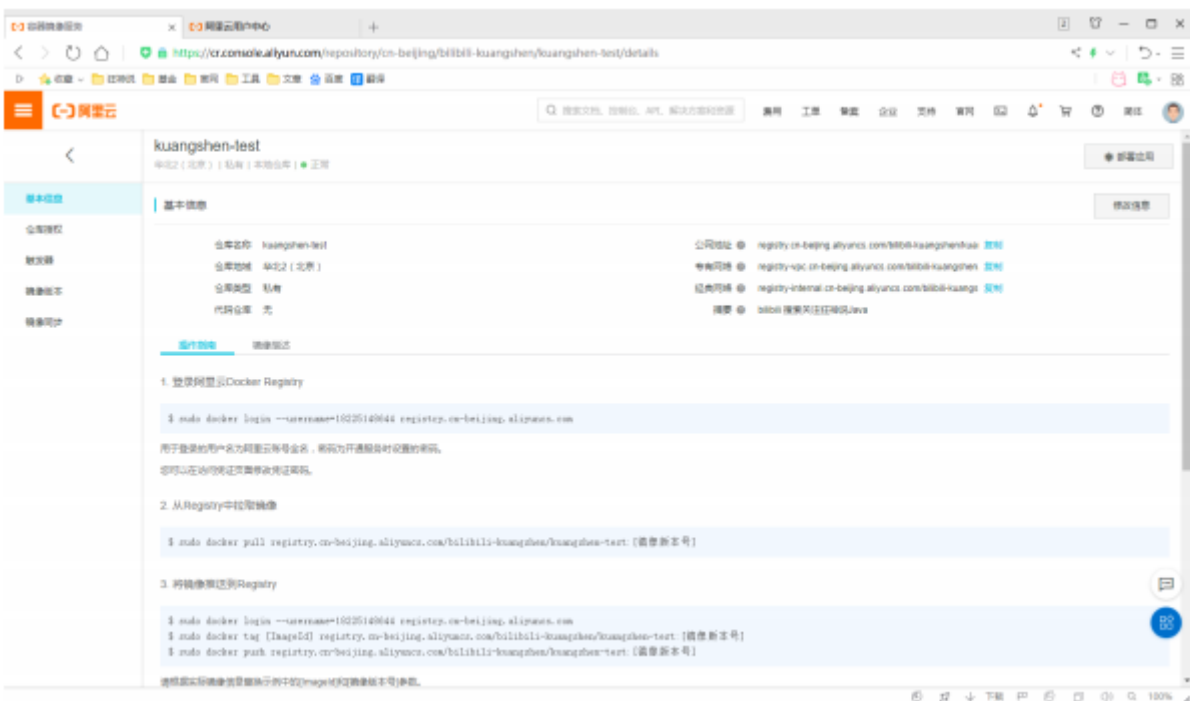


4、创建镜像仓库





5、点击进入这个镜像仓库，可以看到所有的信息



6、测试推送发布

```
[tomcat]# docker login --username=18225148644 registry.cn-beijing.aliyuncs.com
Password:
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-
store

Login Succeeded
```

2、设置 tag

```
docker tag [ImageId] registry.cn-beijing.aliyuncs.com/bilibili-
kuangshen/kuangshen-test:[镜像版本号]
```

```
[tomcat]# docker tag 251ca4419332 registry.cn-beijing.aliyuncs.com/bilibili-kuangshen/kuangshen-test:v1.0
```

3、推送命令

```
docker push registry.cn-beijing.aliyuncs.com/bilibili-kuangshen/kuangshen-test:[镜像版本号]
```

```
[tomcat]# docker push registry.cn-beijing.aliyuncs.com/bilibili-kuangshen/kuangshen-test:v1.0
```

7、在阿里云镜像仓库查看效果！



版本	镜像ID	状态	Digest	镜像大小	最后更新时间	操作
v1.0	251ca4419332...	正常	5f2706a8790c2e70b0682w5 9d8730a627a21709637411 7273659880ceeb5	266.706 MB	2020-09-13 19:48:21	安全扫描 镜像历史 同步 删除

总结

总结

