Docker

简介

Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从Apache2.0协议开源。 Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中， 然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。 容器是完全使用沙箱机制，相互之间不会有任何接口,更重要的是容器性能开销极低。

Docker支持将软件编译成一个镜像；然后 在镜像中各种软件做好配置，将镜像发布 出去，其他使用者可以直接使用这个镜像。运行中的这个镜像称为容器，容器启动是 非常快速的。类似windows里面的ghost操 作系统，安装好后什么都有了

为什么使用docker:

由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，Docker 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

- 更快速的启动时间
- 一致的运行环境
- 持续交互和部署
- 更轻松的迁移
- 更轻松的维护和扩展

Docker引擎:

`是一个包含以下主要组件的客户端服务器的应用程序`

- 一种服务器，他是一个称为守护进程并且长时间运行的程序
- Rest Apl用于指定程序可以用来与守护进程通信的接口，并指示它做什么
- 一个有命令行界面工具的客户端

Docker系统镜像:

- Docker镜像(Images)：Docker 镜像是用于创建Docker 容器的 模板。
- Docker容器(Container)：容器是独立运行的一个或一组应用。
- Docker客户端(Client)：客户端通过命令行或者其他工具使用 Docker API(https://docs.docker.com/reference/api/docker_remote_api) 与Docker 的守护进程通信 docker主机(Host)：一个物理或者虚拟的机器用于执行 Docker 守护进程和容器。
- Docker仓库(Registry)：Docker 仓库用来保存镜像，可以理解 为代码控制中的代码仓库。Docker Hub(https://hub.docker.com) 提供了庞大的镜像集合供使用

Docker容器:

Docker容器是docker运行的实体，容器可以被创建，启动，停止，删除，暂停等
容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的 命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。
容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使

用 数据卷（Volume）、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主（或网络存储）发生读写，其性能和稳定性更高。数据卷的生存周期独立于容器，容器消亡，数据卷不会消亡

Docker仓库：

一个 Docker Registry 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。
通常，一个仓库会包含同一个软件不同版本的镜像，而标签就常用于对应该软件的各个版本。我们可以通过 <仓库名>:<标签> 的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，将以 latest 作为默认标签。

ubuntu安装docker:

- 命令: `wget -qO- http://get.docker.com/ |sh`
- 启动: `sudo service docker start`
- 测试运行: `docker run hello-world`
- 镜像加速: /etc/docker/daemon.json 在这里加入`{ "registry-mirrors":["http://hub-mirror.c.163.com"]}`

```
ubuntu安装：
        1.更换国内软件源，推荐中国科技大学的源，稳定速度快（可选）

            sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak
            sudo sed -i 's/archive.ubuntu.com/mirrors.ustc.edu.cn/g'
            /etc/apt/sources.list
            sudo apt update
        2.安装需要的包
            sudo apt install apt-transport-https ca-certificates software-
properties-common curl
        3.添加 GPG 密钥，并添加 Docker-ce 软件源，这里还是以中国科技大学的 Docker-ce 源为
例
            curl -fsSL https://mirrors.ustc.edu.cn/docker-ce/linux/ubuntu/gpg |
sudo apt-key add -
            sudo add-apt-repository "deb [arch=amd64]
https://mirrors.ustc.edu.cn/docker-ce/linux/ubuntu \
            $添加成功后更新软件包缓存安装设置开机自启动并启动（安装成功后默认已设置并启动，可
忽略）测试运行添加当前用户到用户组，可以不用运行（可选）(lsb_release -cs) stable"
        4.添加成功后更新软件包缓存sudo apt update
        5.安装 Docker-ce:::     sudo apt install docker-ce
        6.设置开机自启动并启动 Docker-ce（安装成功后默认已设置并启动，可忽略）
            sudo systemctl enable docker
            sudo systemctl start docker
        7.测试运行sudo docker run hello-world
        8.添加当前用户到 docker 用户组，可以不用 sudo 运行 docker（可选）
            sudo groupadd docker
            sudo usermod -aG docker $USER
        9.测试添加用户组（可选）docker run hello-world
```

ubuntu脚本自动安装：

- 1`.curl -fsSL get.docker.com -o get-docker.sh`
- 2.`sh get-docker.sh --mirror Aliyun`
  或者第二步使用：`sudo sh get-docker.sh --mirror AzureChinaCloud`
- 3.测试是否安装成功：
  `docker version`

ubuntu安装镜像加速器：

镜像加速: 如果没有则创建这个文件：

/etc/docker/daemon.json 在这里加入 `{ "registry-mirrors":["https://registry.docker-cn.com"]}`

重启服务：

`systemctl restart docker`

Centos安装Docker

centos命令安装

安装命令：`yum install docker`

检查是否安装成功: `docker version`

启动docker server: `systemctl start docker.service` 或者 `systemctl start docker`

设置docker开机自启：`systemctl enable docker.service`

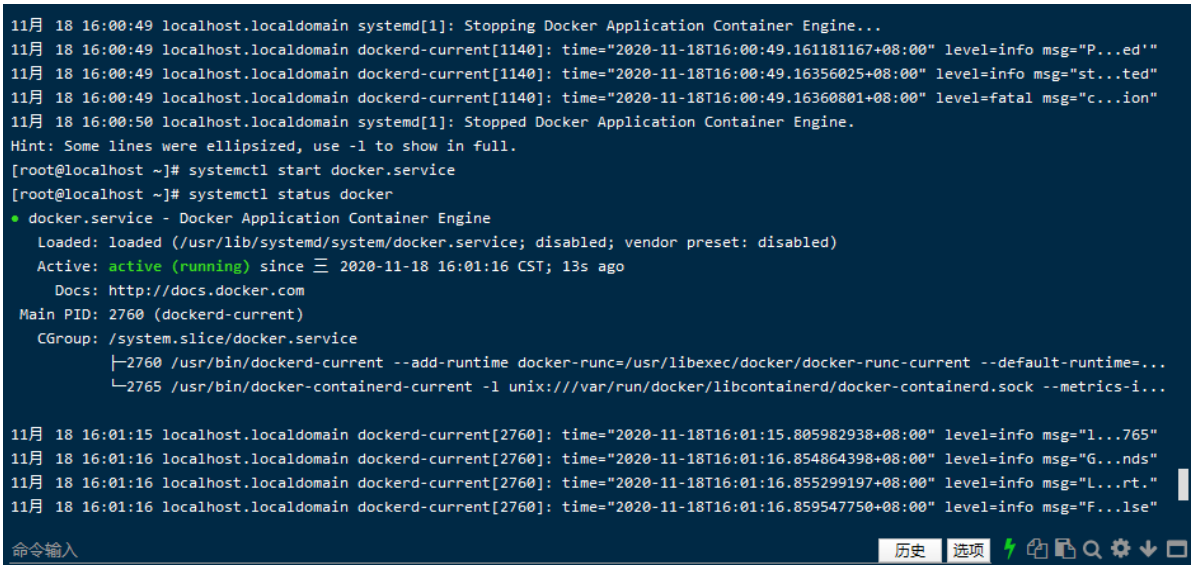
安装wget命令： `yum -y install wget`

搜索一些包：`yum search java|grep jdk`

安装vim编辑器：`yum - y install vim*`

检查docker状态：`systemctl status docker`

停止docker: `systemctl stop docker`



idea链接：

`vim /usr/lib/systemd/system/docker.service` 找到 `ExecStart=/usr/bin/dockerd-current` 然后直接加入如下

```
## ExecStart=/usr/bin/dockerd-current -H tcp://0.0.0.0:2375 -H
unix://var/run/docker.sock \
```

然后进行重启：

```
systemctl daemon-reload && systemctl restart docker
```

DockerFile：

镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决

dockerfile是一个文本文件，其内包含了一条条指令，每条指令构建一层，因此每条指令都应该描述如何构建.

注意：Dockerfile 的指令每执行一次都会在 docker 上新建一层。所以过多无意义的层，会造成镜像膨胀过大

在 /usr/local:

创建docker目录，然后创建一个tomcat的dockerfile目录，

`from:` 必须是第一条指令，用于指定基础的镜像

`run` ：用于执行命令行命令，由于命令行的强大，

命令有两种格式：shell格式： exec格式：

DokerFile指令

指令相对比较庞大 自行学习把，比较常用的有 `FROM，RUN，还提及了COPY,ADD`

参考：dockfile定制镜像

DockerFile轻量化方式

下面的这个的dockerfile 是从 Debian "jessie"系统下构建一个编译、安装 redis 可执行文件。

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y gcc libc6-dev make
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz"
RUN mkdir -p /usr/src/redis
RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1
RUN make -C /usr/src/redis
RUN make -C /usr/src/redis install
```

Dockerfile 中每一个指令都会建立一层，`RUN` 也不例外。每一个 `RUN` 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，`commit` 这一层的修改，构成新的镜像。

而上面的这种写法，创建了 7 层镜像。这是完全没有意义的，而且很多运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常臃肿、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错

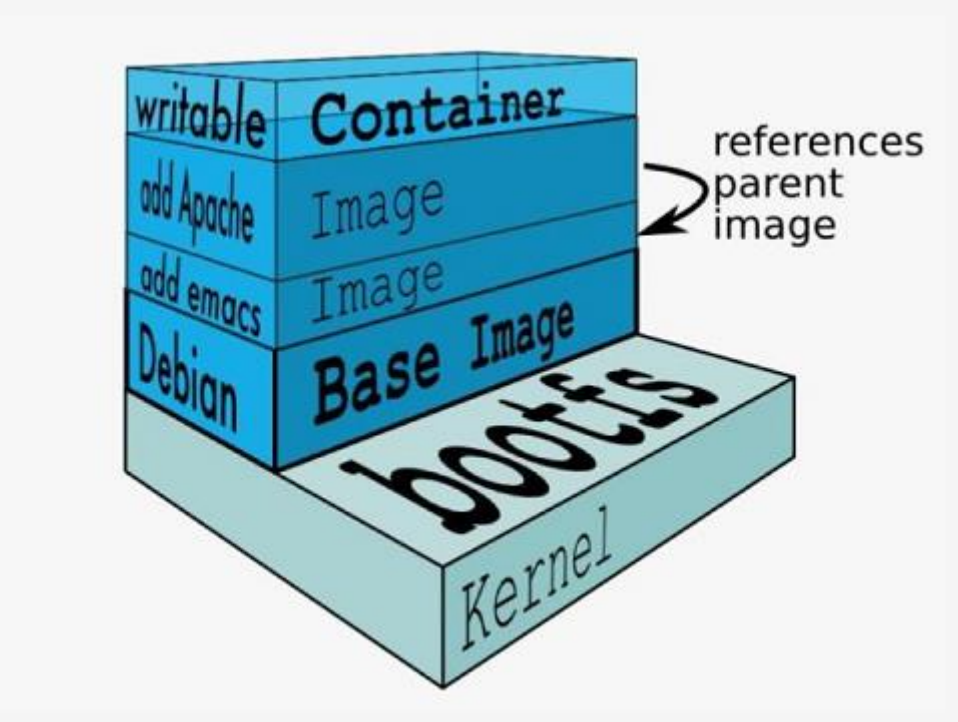
```
FROM debian:jessie

RUN buildDeps='gcc libc6-dev make' \
    && apt-get update \
    && apt-get install -y $buildDeps \
    && wget -O redis.tar.gz "http://download.redis.io/releases/redis-
3.2.5.tar.gz" \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -rf /var/lib/apt/lists/* \
    && rm redis.tar.gz \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps
```

这里没有使用很多个 RUN 对一一对应不同的命令，而是仅仅使用一个 RUN 指令，并使用 && 将各个所需命令串联起来。将之前的 7 层，简化为了 1 层。这并不是在写 Shell 脚本，而是在定义每一层该如何构建.并且，这里为了格式化还进行了换行。Dockerfile 支持 Shell 类的行尾添加 `\` 的命令换行方式，以及行首 `#` 进行注释的格式。良好的格式，比如换行、缩进、注释等，会让维护、排障更为容易，这是一个比较好的习惯。

此外，还可以看到这一组命令的最后添加了清理工作的命令，删除了为了编译构建所需要的软件，清理了所有下载、展开的文件，并且还清理了 `apt` 缓存文件。这是很重要的一步，我们之前说过，镜像是多层存储，每一层的东西并不会在下一层被删除，会一直跟随着镜像。因此镜像构建时，一定要确保每一层只添加真正需要添加的东西，任何无关的东西都应该清理掉

关于层的图片请看：

一个docker镜像由多个可读的镜像层组成，然后运行的容器会在这个docker的镜像上面多加一层可写的容器层，任何的对文件的更改都只存在此容器层。因此任何对容器的操作均不会影响到镜像



dockerfile 支持shell类后面添加\命令方式换行，以及首行#号进行注释格式，很多人 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件. **那就是使用&&进行链接**:

```
FROM centos
RUN yum install wget
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-5.0.3.tar.gz"
RUN tar -xvf redis.tar.gz
```

```
以上执行会创建 3 层镜像。可简化为以下格式：
FROM centos
RUN yum install wget \
    && wget -O redis.tar.gz "http://download.redis.io/releases/redis-
5.0.3.tar.gz" \
    && tar -xvf redis.tar.gz
```

`docker bulid -t centos:1.0` 进行构建

比如构建一个nginx:

在一个空白目录中，建立一个文本文件，并命名为 `Dockerfile`： `vim Dockerfile`

```
From nginx
Run echo '</h1>Hello, Docker!</h1>' /usr/share/nginx/html/index.html
```

使用以下命令进行构建:

```
docker build [选项] <上下文路径/URL/->
```

在这里我们指定了最终镜像的名称 `-t nginx:v3`，构建成功后，运行这个镜像

Dokerfile比较庞大具体的其他的构建可以去看 [docker file list]

更多关于docker层的概念请看：

Docker分层原理与内部结构

Dockerfile实践指南之层数与大小的控制

Docker多阶段构建最佳实践

docker镜像的优化

使用上下文环境构建：

会看到 `docker build` 命令最后有一个 `.`。`.` 表示当前目录，而 `Dockerfile` 就在当前目录，`这个路径不能认为是在指定 Dockerfile 所在的路径`，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定**上下文路径**。

一般来说，应该会将 `Dockerfile` 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 `.gitignore` 一样的语法写一个 `.dockerignore`，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

那么为什么会有人误以为 `.` 是指定 `Dockerfile` 所在目录呢？这是因为在默认情况下，如果不额外指定 `Dockerfile` 的话，会将上下文目录下的名为 `Dockerfile` 的文件作为 Dockerfile。

这只是默认行为，实际上 `Dockerfile` 的文件名并不要求必须为 `Dockerfile`，而且并不要求必须位于上下文目录中，比如可以用 `-f ../Dockerfile.php` 参数指定某个文件作为 `Dockerfile`。

当然，一般大家习惯性的会使用默认的文件名 `Dockerfile`，以及会将其置于镜像构建上下文目录中。

创建一个html文件：

创建一个Dockerfile 里面书写：`from tomcat copy /usr/local/tomcat/webapps/root`
然后执行一个为：`docker build -t myshop .`
里会自动的寻找到 Dockerfile可以看到打印的执行的语句然后进入 tomcat中会发现已经创建了个index.html

```
在Dockerfile 中的使用      构建时先声明
    from tomcat
    workdir /usr/local/tomcat/webapps/root/
    run rm -fr *
    copy spring-boot-institute.jar .
    Run  unzip spring-boot-institute.jar
    run rm -fr spring-boot-institute.jar
    workdir /usr/local/tomcat
```

运行这个docker容器：`docker build -t institute` 这里会自动的寻找到Dockerfile

比如一个项目中这样使用：

```
FROM php:apache
#Install git
RUN apt-get update
    && apt-get install -y git
    && docker-php-ext-install pdo pdo_mysql mysqli
    && a2enmod rewrite
#Install Composer
RUN php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
&& php composer-setup.php --install-dir=. --filename=composer
&& mv composer /usr/local/bin/
COPY ./html/ /var/www/html/
EXPOSE 80
```

执行命令启动： `docker build -f dockerfile -t myapp:0.1`

比如一个maven项目：

创建一个DockerFile `vim dockerfile`

```
# First stage: complete build environment
FROM maven:3.5.0-jdk-8-alpine AS builder

# add pom.xml and source code
ADD ./pom.xml pom.xml
ADD ./src src/

# package jar
RUN mvn clean package

# Second stage: minimal runtime environment
From openjdk:8-jre-alpine

# copy jar from the first stage
COPY --from=builder target/my-app-1.0-SNAPSHOT.jar my-app-1.0-SNAPSHOT.jar

EXPOSE 8080

CMD ["java", "-jar", "my-app-1.0-SNAPSHOT.jar"]
```

执行命令启动： `docker build -f dockerfile -t myapp:0.1`

Docker数据卷：

数据卷 是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

数据卷 可以在容器之间共享和重用

对 数据卷 的修改会立马生效

对 数据卷 的更新，不会影响镜像

数据卷 默认会一直存在，即使容器被删除

注意：数据卷 的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的 数据卷。

- 创建一个数据卷：docker volume create my-vol
- `查看所有数据卷：docker volume ls
- 在主机里查看指定的数据卷：docker volume inspect my-vol启动一个挂载的数据容器：在用 docker run 命令的时候，使用 --mount 标记来将 数据卷 挂载到容器里。在一次 docker run 中可以挂载多个 数据卷。
- 下面创建一个名为 web 的容器，并加载一个 数据卷 到容器的 /webapp 目录。：

```
$ docker run -d -P \
  --name web \
    # -v my-vol:/wepapp \
    --mount source=my-vol,target=/webapp \
    training/webapp \
    python app.py
```

查看数据卷的信息：

在主机里使用以下命令可以查看 web 容器的信息
docker inspect web

删除数据卷：

命令：docker volume rm my-vol

数据卷 是被设计用来持久化数据的，它的生命周期独立于容器，Docker 不会在容器被删除后自动删除 数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的 数据卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用 docker rm -v 这个命令。

无主的数据卷的删除用以下命令清理：

docker volume pruse

实例共享数据卷：

1. 在/usr/loca/docker/tomcat下创建一个ROOT目录，并在里面书写一个index.html
2. 然后启动docker容器的tomcat：

   docker run -p 8080:8080 --name tomcat -d -v
   /usr/local/docker/tomcat/ROOT:/usr/local/tomcat/webapps/ROOT tomcat
3. 通过交互式进入容器中的tomcat：docker exec -it tomcat bash
   可以在webapps中的ROOT目录看到index.html，
4. 然后在启动一个tomcat:docker run -p 8081:8080 --name tomcat1 -d -v
   /usr/local/docker/tomcat/ROOT:/usr/local/tomcat/webapps/ROOT tomcat
   这时访问就可以看到两个同时访问了一个数据卷了，这时就可以共享到数据卷了

Docker镜像：

docker下载镜像：

1.docker pull ubuntu:16.04

//docker image ls :查看镜像列表列出的是顶级的镜像

// docker ps :查看容器列表

2.运行这个ubuntu容器：

```
docker run  -it --rm \
ubuntu:16.04 \
bash
```

其实上面的一串命令等于：`docker run  -it --rm ubuntu:16.04 bash`

3.说明：

it：这是两个参数，一个是 -i：交互式操作，一个是 -t 终端。我们这里打算进入 bash 执行一些命令并查看返回结果，因此我们需要交互式终端。

--rm：这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 docker rm。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 --rm 可以避免浪费空间。ubuntu:16.04：这是指用 ubuntu:16.04 镜像为基础来启动容器。bash：放在镜像名后的是命令，这里我们希望有个交互式 Shell，因此用的是 bash运行一个容器等于运行一个对象，

Docker镜像操作

列出镜像：`docker image ls`

拉取镜像：`docker pull [ubuntu:13.10]`

查找镜像：`docker search [httpd]`

镜像是不能够单独启动的他是依靠容器的 根据镜像运行一个容器：`docker run --name container-name -d image-name`

eg：`docker run –name myredis –d redis`

> --name：自定义容器名
>
> -d：后台运行 image-name:指定镜像模板 列表 docker ps（查看运行中的容器）；
>
> 加上-a；可以查看所有容器
> //docker image ls :查看镜像列表列出的是顶级的镜像
> //docker ps :查看容器列表

删除镜像：`docker image rm [选项] <镜像1> [镜像2，...]` 其中，`<镜像>` 可以是 `镜像短 ID`、`镜像长 ID`、`镜像名` 或者 `镜像摘要`。或者：`docker rmi [镜像name]`

构建镜像：`docker build -t runoob/centos:6.7 .`

参数说明：

- **-t**：指定要创建的目标镜像名
- **.**：Dockerfile 文件所在目录，可以指定Dockerfile 的绝对路径

❤*注*：[]内均为参数，也就是必选的

虚悬镜像：

镜像列表中，还可以看到一个特殊的镜像，这个镜像既没有仓库名，也没有标签，均为 。

原因：

这个镜像原本是有镜像名和标签的，原来为 mongo:3.2，随着官方镜像维护，发布了新版本后，重新 docker pull mongo:3.2 时，mongo:3.2 这个镜像名被转移到了新下载的镜像身上，而旧的镜像上的这个名称则被取消，从而成为了 。除了 docker pull 可能导致这种情况，docker build 也同样可以导致这种现象。

由于新旧镜像同名，旧镜像名称被取消，从而出现仓库名、标签均为 的镜像。

删除虚悬镜像：

```
$ docker image prune
```

查看中间层镜像：

```
docker image ls -a
```

这些无标签的镜像很多都是中间层镜像，是其它镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。

实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍，而这些镜像是别的镜像的依赖，因此并不会因为它们被列出来而多存了一份，无论如何你也会需要它们

docker image ls 还支持强大的过滤器参数 --filter，或者简写 -f。之前我们已经看到了使用过滤器来列出虚悬镜像的用法，它还有更多的用法。比如，我们希望看到在 mongo:3.2 之后建立的镜像

```
$ docker image ls -f since=mongo:3.2
想看某个位置之前的镜像 ：只需要把 since 换成 before
```

删除指定的镜像：

docker image rm [选项] <镜像1> [<镜像2> ...] 其中，<镜像> 可以是 镜像短 ID、镜像长 ID、镜像名或者 镜像摘要。

```
停止 docker stop container-name/container-id 停止当前你运行的容器
启动 docker start container-name/container-id 启动容器
删除 docker rm container-id 删除指定容器
端口映射 -p 6379:6379 eg:docker run -d -p 6379:6379 --name myredis docker.io/redis
-p:主机端口(映射到)容器内部的端口
容器日志 docker logs container-name/container-id
```

Docker构建tomact:

进入docker目录：`docker pull tomcat`

启动：

```
docker run --name tomcat -p 8080:8080 -v $PWD/test:/usr/local/tomcat/webapps/test
-d tomcat
说明：
-p 8080:8080：将容器的8080端口映射到主机的8080端口
-v $PWD/test:/usr/local/tomcat/webapps/test：将主机中当前目录下的test挂载到容器的/test
```

查看容器启动情况：docker ps

备注：

- 1.命令：`docker pull tomcat`
  下载tomcat 9： `docker pull tomcat:9-jre8`
- 2.docker中运行tomcat：需要制定端口
  `docker run -p 8080:8080 tomcat`

Docker构建mysql:

进入docker目录：`docker pull mysql:5.7.22`
启动运行：

```
docker run -p 3306:3306 --name mysql \
            -v /usr/local/docker/mysql/conf:/etc/mysql \
            -v /usr/local/docker/mysql/logs:/var/log/mysql \
            -v /usr/local/docker/mysql/data:/var/lib/mysql \
            -e MYSQL_ROOT_PASSWORD=lkl555 \
            -d mysql:5.7.22
```

说明：-p 3306:3306：将容器的3306端口映射到主机的3306端口
-v /usr/local/docker/mysql/conf:/etc/mysql：将主机当前目录下的 conf 挂载到容器的 /etc/mysql
-v /usr/local/docker/mysql/logs:/var/log/mysql：将主机当前目录下的 logs 目录挂载到容器的 /var/log/mysql
-v /usr/local/docker/mysql/data:/var/lib/mysql：将主机当前目录下的 data 目录挂载到容器的 /var/lib/mysql
-e MYSQL_ROOT_PASSWORD=123456：初始化root用户的密码
查看：docker ps -a 查看刚刚启动的服务

出现的错误：

Error response from daemon: driver failed programming external connectivity on endpoint mysql (c28bcf099d63d5f3b2affd38a033a42d60e3cf7054fc1bd520342b73f6a2987b): Error starting userland proxy: listen tcp 0.0.0.0:3306: bind: address already in use

原因是本地的mysql已经启动了端口不能够进行映射了、

所以停掉本地的mysql服务：sudo service mysql stop

或者使用：docker ps -a

然后使用 docker rm xxxxxxxxx

然后再从新启动

需要进入：docker容器内

```
docker run -it --rm mysql:5.7.22 bash
ls  -al
查看安装到容器的哪个位置：whereis mysql
进入、etc/mysql/mysql.conf.d

查看 mysqld.cnf是否有 max_allowd_packet=128M

没有的话则进行追加：echo "max_allowd_packet=128M" >> mysqld.cnf

这样是为了设置mysql可以执行文件的大小。

修改之后退出，进入ubuntu界面，然后重启mysql容器 docker restart mysql
将容器里面的配置文件移到宿主机 mysql/conf 目录下: docker cp  mysql:/etc/mysql .
将这些文件移动到conf目录下
```

注意：这里出现了错误并不能够去copy过去。。。。

Docker构建oracle

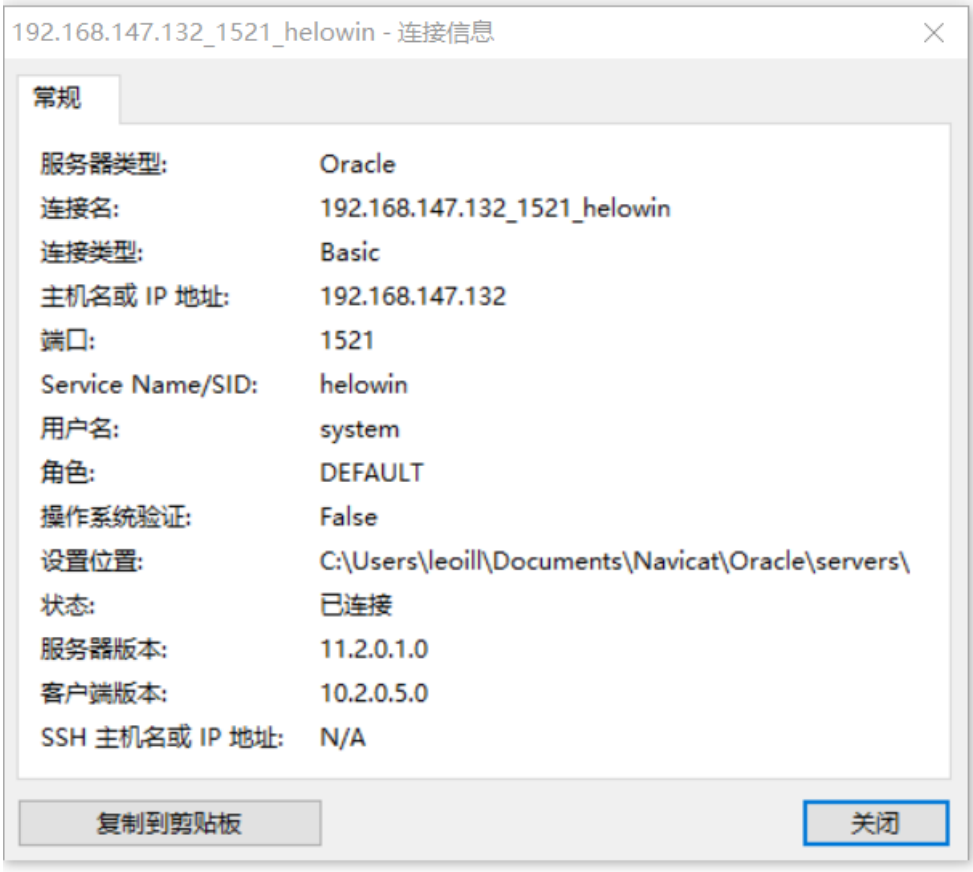
- 1.拉去oracle数据库镜像
  `docker pull registry.cn-hangzhou.aliyuncs.com/helowin/oracle_11g`

- 2.启动oracle 自动启动镜像 --restart=always
  `docker run -p 1521:1521 --name oracle_11g -d --restart=always registry.cn-hangzhou.aliyuncs.com/helowin/oracle_11g`
- 3.启动服务：`docker start oracle_11g`
- 4.进入控制台设置用户信息：`docker exec -it oracle_11g bash`
- 5.切换到root用户模式下
  su root
  输入密码helowin
- 6.编辑profile文件配置ORACLE环境变量

  > export ORACLE_HOME=/home/oracle/app/oracle/product/11.2.0/dbhome_2
  > export ORACLE_SID=helowin
  > export PATH=$ORACLE_HOME/bin$:PATH
- 7.重启配置文件服务
  source /etc/profile
- 8.建立sqlplus软连接
  ln -s $ORACLE_HOME/bin/sqlplus /usr/bin
- 9.切换到oracle用户，修改oracle的相关账号密码

> su oracle
> //登录sqlplus并修改sys、system用户密码
> sqlplus /nolog
>
> conn /as sysdba
>
> alter user system identified by oracle; # 将system的密码改为oracle
>
> alter user sys identified by oracle; # 将sys的密码改为oracle
>
> ALTER PROFILE DEFAULT LIMIT PASSWORD_LIFE_TIME UNLIMITED; # 更改配置文件默认限制密码

- 10.使用navicat进行连接：



主页 | Ubuntu-server x64 -oracle

```
        . "$i" >/dev/null 2>&1
      fi
    fi
done

unset i
unset -f pathmunge
export ORACLE_HOME=/home/oracle/app/oracle/product/11.2.0/dbhome_2
export ORACLE_SID=helowin
export PATH=$OREACLE_HOME/bin:$PATH
"etc/profile" 81L, 1925C written
[root@805e99c47643 /]# source /etc/profile
[root@805e99c47643 /]# ln -s $ORACLE_HOME/bin/sqlplus /usr/bin
[root@805e99c47643 /]# su oracle
[oracle@805e99c47643 /]$ sqlplus /nolog

SQL*Plus: Release 11.2.0.1.0 Production on Thu Oct 8 14:38:06 2020

Copyright (c) 1982, 2009, Oracle.  All rights reserved.

SQL> conn /as sysdba
Connected.
SQL> alert user system identified by oracle;
SP2-0734: unknown command beginning "alert user..." - rest of line ignored.
SQL> alter user system identified by oracle;

User altered.

SQL> alter user sys identified by  oracle;

User altered.

SQL> alter profile default limit password_life_time unlimited;

Profile altered.

SQL>
```

oracle中的sys用户和system用户的区别：

- 【sys】所有 oracle 的数据字典的基表和视图都存放在 sys 用户中，这些基表和视图对于 oracle 的运行是至关重要的，由数据库自己维护，任何用户都不能手动更改。 sys 用户拥有 dba ， sysdba ， sysoper 等角色或权限，是 oracle 权限最高的用户。
- 【 system】 用户用于存放次一级的内部数据，如 oracle 的一些特性或工具的管理信息。 system 用户拥有普通 dba 角色权限。
- 【system】用户只能用 normal 身份登陆 em ，除非你对它授予了 sysdba 的系统权限(grant sysdba to system)或者 sysoper 系统权限。
- 【 sys 】用户具有 “SYSDBA” 或者 “SYSOPER” 系统权限，登陆 em 也只能用这两个身份，不能用 normal 。

Docker 容器命令操作

新建启动 `docker run [container ID or NAMES]`

查看日志： `docker container logs [container ID or NAMES]` **例：docker container logs 635c90c02f22**

查看容器列表： `docker container ls`

守护态运行： `docker run -d ubuntu:17.10 /bin/sh -c "while true; do echo hello world; sleep 1; done"`

查看容器： `docker container ls -a`

停止容器运行： `docker container stop [id]` 等价于---> `docker stop [id]` 例如：**docker container stop 635c90c02f22**

进入容器： `docker exec -t [ubuntu]`

删除容器： `docker container rm [container-name]` 或者： `docker rm -f [continer Id 或者name]`

清理所有处于终止的容器： `docker container prune`

查看当前运行的容器： `docker ps`

启动容器： `docker start [container ID or NAMES]` **docker start 26c7868e652f**

❤*注*：[]内均为参数，也就是必选的，

Docker Compose:

- Docker Compose：是docker官方编排的快速，对于容器集群很有利
- Compose定位是：定义和运行多个Docker容器的应用，主要是为了解决多个容器间的相互配合来完成某项任务。他允许用户通过docker-compose.yml的定义一组相关连的应用容器为一个项目
- 服务：一个应用容器，实际上可以包括若干个运行相同镜像的容器实例
- 项目：由一组关联的应用容器组成的一个完整业务单元，在docker-compose.yml中定义compose默认管理的是对象是项目，通过子命令对项目快捷的生命周期的管理

安装docker compose:

`如果你已经安装了docker那么就先把docker给删除了，`

`先进行卸载：apt-get autoremove docker-cedocker`

`然后删除dokcerlist文件：在/etc/apt/sources.list.d`

安装docker：

安装docker：`sudo curl -fsSL get.docker.com -o get-docker.sh`

因为有的aliyun上面没有具体的升级版本，所以把镜像给改写：`sh get-docker.sh --mirror AzureChinaCloudcd`

这时只需等待即可。。。。。。

检查镜像加速文件是否存在：在/etc/docker/daemon.json中
/etc/docker/daemon.json 在这里加入`{ "registry-mirrors":["https://registry.docker-cn.com"]}`

检查dockerversion:

然后安装docker-compose看下面的就可以了

安装docker 和docker compose要同时安装，

linux安装docker compose：采用二进制包进行安装：

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
加入可执行的权限：
$ sudo chmod +x /usr/local/bin/docker-compose
```

卸载docker compose:

`$ sudo rm /usr/local/bin/docker-compose`

Docker Compose 使用：

模板文件：

在docker 文件夹中的tomcat中的创建 docker-compose.yml

```
version: "3"
services:
  webapp:
  image: examples/web
    ports:
    - "80:80"
      volumes:
    - "/data"
```

注意每个服务都必须通过 image 指令指定镜像或 build 指令（需要 Dockerfile）等来自动构建生成镜像。

运行 compose 项目： docker-compose up

Docker compose的命令的使用：

```
基本的命令的格式的：docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
    命令选项：
        -f, --file FILE 指定使用的 Compose 模板文件，默认为 docker-compose.yml，可以多次指定。
        -p, --project-name NAME 指定项目名称，默认将使用所在目录名称作为项目名。
            --x-networking 使用 Docker 的可拔插网络后端特性
            --x-network-driver DRIVER 指定网络后端的驱动，默认为 bridge
            --verbose 输出更多调试信息。
            -v, --version 打印版本并退出。
        build的命令：
            docker-compose build [options] [SERVICE...]
            option：
                --force-rm 删除构建过程中的临时容器。
                --no-cache 构建镜像过程中不使用 cache（这将加长构建过程）。
                --pull 始终尝试通过 pull 来获取更新版本的镜像。
            config:
                验证文件格式是否正确，若正确则显示配置，若错误则进行错误原因
            down：此命令将会停止 up 命令所启动的容器，并移除网络
            exec：进入指定的容器。
            help：获得一个命令的帮助。
            images：列出 Compose 文件中包含的镜像。
            kill：格式为 docker-compose kill [options] [SERVICE...]。通过发送 SIGKILL 信号来强制停止服务容器。
            rm:格式为 docker-compose rm [options] [SERVICE...]。
            删除所有（停止状态的）服务容器。推荐先执行 docker-compose stop 命令来停止容器。
            -f：强制直接删除
            -v:删除容器所挂载的数据卷
            run:docker-compose run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]。
            参数：
                -d 后台运行容器。
                --name NAME 为容器指定一个名字。
                --entrypoint CMD 覆盖默认的容器启动指令。
                -e KEY=VAL 设置环境变量值，可多次使用选项来设置多个环境变量。
                -u, --user="" 指定运行容器的用户名或者 uid。
                --no-deps 不自动启动关联的服务容器。
                --rm 运行命令后自动删除容器，d 模式下将忽略。
                -p, --publish=[] 映射容器端口到本地主机。
                --service-ports 配置服务端口并映射到本地主机。
                -T 不分配伪 tty，意味着依赖 tty 的指令将无法运行。
                start,stop,top,pause,port 和docker中的使用的都一样
```

`前台运行：docker-compose up`

`后台运行：docker-compose up -d`

`启动：docker-compose start`

`停止：docker-compose stop`

`停止并移除容器：docker-compose down`

Docker compose实战Nginx

```
version: '3.1'
  services:
  nginx:
    restart: always
    image: nginx
    container_name: nginx
    ports:
      - 81:80
    volumes:
      - ./conf/nginx.conf:/etc/nginx/nginx.conf
      - ./wwwroot:/usr/share/nginx/wwwroot
```

配置nginx的数据卷：

在dokcer文件下创建：nginx的目录下创建：nginx.conf文件：

在当前目录创建wwwroot目录

这里直接书写为
/usr/local/dokcer/nginx/conf/nginx.conf
/usr/local/dokcer/nginx/wwwroot

在nginx.conf目录配置nginx的虚拟主机：然后去配置nginx即可

Docker compose实战tomcat:

```
version: '3.1'
services:
  tomcat:
    restart: always
    image: tomcat
    container_name: tomcat
    ports:
      - 8080:8080
    volumes:
      - /usr/local/docker/tomcat/webapps/test:/usr/local/tomcat/webapps/test
    environment:
      TZ: Asia/Shanghai
```

Docker compose 实战mysql5：

```
version: '3.1'
services:
  mysql:
    restart: always
    image: mysql:5.7.22
    container_name: mysql
    ports:
      - - 3306:3306
        environment:
          TZ: Asia/Shanghai
          MYSQL_ROOT_PASSWORD: 123456
```

```
        command:
          --character-set-server=utf8mb4
          --collation-server=utf8mb4_general_ci
          --explicit_defaults_for_timestamp=true
          --lower_case_table_names=1
          --max_allowed_packet=128M
          --sql-
mode="STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,NO_ZERO_DAT
E,NO_ZERO_IN_DATE,ERROR_FOR_DIVISION_BY_ZERO"
        volumes:
          - mysql-data:/var/lib/mysql
volumes:
  mysql-data:
```

就是为了简化docker的使用：

直接运行 docker-compose.yml中的就可以使用。运行即可以

`docker-compose up`

Docker compose 实战oracle

```
version: '3.1'
services:
  oracle:
    # sid: xe
    # username: system
    # password: oracle
    image: sath89/oracle-xe-11g
    restart: always   #如果docker容器由于一些问题挂掉的化，docker-composer会自动把容器给
启动起来
    container_name: oracle  #启动之后容器的名称
    volumes:
      - /my-docker-data/oracle-11g/data:/u01/app/oracle
    ports:
      - 1521:1521
```

- `启动服务：docker-compose -f oracle.yml up -d`
- `查看运行的组件：docker ps -a`

Docker compose 实战RabbitMQ

```
version: '3.1'
services:
  rabbitmq:
    restart: always
    image: rabbitmq:management
    container_name: rabbitmq
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      TZ: Asia/Shanghai
      RABBITMQ_DEFAULT_USER: rabbit
      RABBITMQ_DEFAULT_PASS: 123456
    volumes:
```

```
      - ./data:/var/lib/rabbitmq
```

centos用上面的会出现 `chown: changing ownership of '/var/lib/rabbitmq': Permission denied`

解决方案：https://jingyan.baidu.com/article/9c69d48f7821b853c9024ef8.html

Docker compose 实战RocketMq

docker-compose.yml

注意：启动RocketMQ server + Broker+ Console至少需要2G内容。

```
version: '3.5'
services:
  rmqnamesrv:
    image: foxiswho/rocketmq:server
    container_name: rmqnamesrv
    ports:
      - 9876:9876
    volumes:
      - ./data/logs:/opt/logs
      - ./data/store:/opt/store
    networks:
        rmq:
          aliases:
            - rmqnamesrv

  rmqbroker:
    image: foxiswho/rocketmq:broker
    container_name: rmqbroker
    ports:
      - 10909:10909
      - 10911:10911
    volumes:
      - ./data/logs:/opt/logs
      - ./data/store:/opt/store
      - ./data/brokerconf/broker.conf:/etc/rocketmq/broker.conf
    environment:
        NAMESRV_ADDR: "rmqnamesrv:9876"
        JAVA_OPTS: " -Duser.home=/opt"
        JAVA_OPT_EXT: "-server -Xms128m -Xmx128m -Xmn128m"
    command: mqbroker -c /etc/rocketmq/broker.conf
    depends_on:
      - rmqnamesrv
    networks:
      rmq:
        aliases:
          - rmqbroker

  rmqconsole:
    image: styletang/rocketmq-console-ng
    container_name: rmqconsole
    ports:
      - 8080:8080
    environment:
```

```
        JAVA_OPTS: "-Drocketmq.namesrv.addr=rmqnamesrv:9876 -
Dcom.rocketmq.sendMessageWithVIPChannel=false"
    depends_on:
      - rmqnamesrv
    networks:
      rmq:
        aliases:
          - rmqconsole

networks:
  rmq:
    name: rmq
    driver: bridge
```

需要配置一下broker.conf 我们需要在./data/brokerconf/ 目录下创建一个名为broker.conf的配置文件。

```
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
#  Unless required by applicable law or agreed to in writing, software
#  distributed under the License is distributed on an "AS IS" BASIS,
#  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#  See the License for the specific language governing permissions and
#  limitations under the License.


# 所属集群名字
brokerClusterName=DefaultCluster

# broker 名字，注意此处不同的配置文件填写的不一样，如果在 broker-a.properties 使用:
broker-a,
# 在 broker-b.properties 使用: broker-b
brokerName=broker-a

# 0 表示 Master，> 0 表示 Slave
brokerId=0

# nameServer地址，分号分割
# namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876

# 启动IP,如果 docker 报
com.alibaba.rocketmq.remoting.exception.RemotingConnectException: connect to
<192.168.0.120:10909> failed
# 解决方式1 加上一句 producer.setVipChannelEnabled(false);，解决方式2 brokerIP1 设置宿
主机IP，不要使用docker 内部IP
# brokerIP1=192.168.0.253

# 在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
defaultTopicQueueNums=4
```

```
# 是否允许 Broker 自动创建 Topic，建议线下开启，线上关闭 ！！！这里仔细看是 false，false，
false
autoCreateTopicEnable=true

# 是否允许 Broker 自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true

# Broker 对外服务的监听端口
listenPort=10911

# 删除文件时间点，默认凌晨4点
deleteWhen=04

# 文件保留时间，默认48小时
fileReservedTime=120

# commitLog 每个文件的大小默认1G
mapedFileSizeCommitLog=1073741824

# ConsumeQueue 每个文件默认存 30W 条，根据业务情况调整
mapedFileSizeConsumeQueue=300000

# destroyMapedFileIntervalForcibly=120000
# redeleteHangedFileInterval=120000
# 检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
# 存储路径
# storePathRootDir=/home/ztztdata/rocketmq-all-4.1.0-incubating/store
# commitLog 存储路径
# storePathCommitLog=/home/ztztdata/rocketmq-all-4.1.0-
incubating/store/commitlog
# 消费队列存储
# storePathConsumeQueue=/home/ztztdata/rocketmq-all-4.1.0-
incubating/store/consumequeue
# 消息索引存储路径
# storePathIndex=/home/ztztdata/rocketmq-all-4.1.0-incubating/store/index
# checkpoint 文件存储路径
# storeCheckpoint=/home/ztztdata/rocketmq-all-4.1.0-incubating/store/checkpoint
# abort 文件存储路径
# abortFile=/home/ztztdata/rocketmq-all-4.1.0-incubating/store/abort
# 限制的消息大小
maxMessageSize=65536

# flushCommitLogLeastPages=4
# flushConsumeQueueLeastPages=2
# flushCommitLogThoroughInterval=10000
# flushConsumeQueueThoroughInterval=60000

# Broker 的角色
# - ASYNC_MASTER 异步复制Master
# - SYNC_MASTER 同步双写Master
# - SLAVE
brokerRole=ASYNC_MASTER

# 刷盘方式
# - ASYNC_FLUSH 异步刷盘
# - SYNC_FLUSH 同步刷盘
flushDiskType=ASYNC_FLUSH
```

```
# 发消息线程池数量
# sendMessageThreadPoolNums=128
# 拉消息线程池数量
# pullMessageThreadPoolNums=128
```

Docker compose实战Redis:

```yaml
version: '3.1'
services:
  redis:
    image: redis:latest
    container_name: redis
    restart: always
    ports:
      - 6379:6379
    networks:
      - mynetwork
    volumes:
      - ./redis.conf:/usr/local/etc/redis/redis.conf:rw
      - ./data:/data:rw
    command:
      /bin/bash -c "redis-server /usr/local/etc/redis/redis.conf "
networks:
  mynetwork:
    external: true
```

Docker compose实战Redis集群:

```yaml
version: '3.1'
  services:
  master:
    image: redis
    container_name: redis-master
    ports:
      - 6379:6379

    slave1:
      image: redis
      container_name: redis-slave-1
      ports:
        - 6380:6379
      command: redis-server --slaveof redis-master 6379

    slave2:
      image: redis
      container_name: redis-slave-2
      ports:
        - 6381:6379
      command: redis-server --slaveof redis-master 6379
```

使用sentinel实现高可用:

```yaml
version: '3.1'
  services:
    sentinel1:
      image: redis
      container_name: redis-sentinel-1
      ports:
        - 26379:26379
      command: redis-sentinel /usr/local/etc/redis/sentinel.conf
      volumes:
        #数据卷
        - ./sentinel1.conf:/usr/local/etc/redis/sentinel.conf
    sentinel2:
      image: redis
      container_name: redis-sentinel-2
      ports:
        - 26380:26379
      command: redis-sentinel /usr/local/etc/redis/sentinel.conf
      volumes:
        - ./sentinel2.conf:/usr/local/etc/redis/sentinel.conf
    sentinel3:
      image: redis
      container_name: redis-sentinel-3
      ports:
        - 26381:26379
      command: redis-sentinel /usr/local/etc/redis/sentinel.conf
      volumes:
        - ./sentinel3.conf:/usr/local/etc/redis/sentinel.conf
```

修改Sentinel文件：

分别为 sentinel1.conf，sentinel2.conf，sentinel3.conf，配置文件内容相同
配置内容如下：

```
port 26379
dir /tmp
# 自定义集群名，其中 127.0.0.1 为 redis-master 的 ip，6379 为 redis-master 的端口，2 为最小投票数（因为有 3 台 Sentinel 所以可以设置成 2）
sentinel monitor mymaster 192.168.147.132 6379 2
sentinel down-after-milliseconds mymaster 30000
sentinel parallel-syncs mymaster 1
sentinel failover-timeout mymaster 180000
sentinel deny-scripts-reconfig yes
```

启动集群：

docker-compose up -d

查看集群是否生效：

> 进入 Sentinel 容器，使用 Sentinel API 查看监控情况：
> docker exec -it redis-sentinel-1 /bin/bash
> redis-cli -p 26379
> sentinel master mymaster
> sentinel slaves mymaster
> 生效后即可：

Docker compose实战FastDFS

参考下的 SpringClouditoken 项目下的FastDFS一节：http://www.kaysanshi.top/learning-note/

Docker compose实战Dubbo zookeeper

单节点：

单机模式

新建zookeeper文件夹。在zookeeper文件夹中新建dokcer-compose.yml

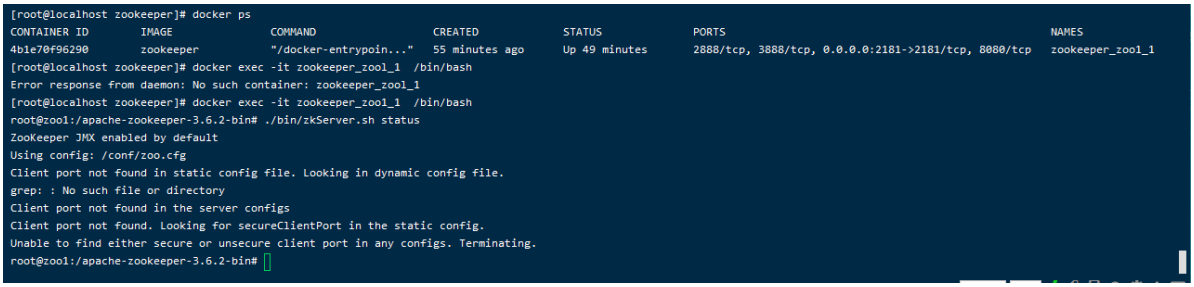
```
version: '3.1'
services:
  zoo1:
    image: zookeeper
    restart: always
    hostname: zookeeper
    ports:
      - 2181:2181
```

进入容器。

`docker exec -it zookeeper_zoo1_1 /bin/bash`

检查是否安装成功：l

`./bin/zkServer.sh status`



进入bin目录后查看bin目录下的文件：

```
root@zookeeper:/apache-zookeeper-3.6.2-bin# ls -l bin
total 64
-rwxr-xr-x. 1 zookeeper zookeeper   232 Sep  4 12:43 README.txt
-rwxr-xr-x. 1 zookeeper zookeeper  2066 Sep  4 12:43 zkCleanup.sh
-rwxr-xr-x. 1 zookeeper zookeeper  1158 Sep  4 12:43 zkCli.cmd
-rwxr-xr-x. 1 zookeeper zookeeper  1620 Sep  4 12:43 zkCli.sh
-rwxr-xr-x. 1 zookeeper zookeeper  1843 Sep  4 12:43 zkEnv.cmd
-rwxr-xr-x. 1 zookeeper zookeeper  3690 Sep  4 12:43 zkEnv.sh
-rwxr-xr-x. 1 zookeeper zookeeper  4559 Sep  4 12:43 zkServer-initialize.sh
-rwxr-xr-x. 1 zookeeper zookeeper  1286 Sep  4 12:43 zkServer.cmd
-rwxr-xr-x. 1 zookeeper zookeeper 11116 Sep  4 12:43 zkServer.sh
-rwxr-xr-x. 1 zookeeper zookeeper   988 Sep  4 12:43 zkSnapShotToolkit.cmd
-rwxr-xr-x. 1 zookeeper zookeeper  1377 Sep  4 12:43 zkSnapShotToolkit.sh
-rwxr-xr-x. 1 zookeeper zookeeper   996 Sep  4 12:43 zkTxnLogToolkit.cmd
-rwxr-xr-x. 1 zookeeper zookeeper  1385 Sep  4 12:43 zkTxnLogToolkit.sh
root@zookeeper:/apache-zookeeper-3.6.2-bin#
```

通过 `zkCli` 来访问 Zookeeper 的控制台来进行管理。

```
root@zookeeper:/apache-zookeeper-3.6.2-bin# zkCli.sh -server 127.0.0.1:2181
Connecting to 127.0.0.1:2181
2020-11-20 02:26:42,393 [myid:] - INFO  [main:Environment@98] - Client
environment:zookeeper.version=3.6.2--803c7f1a12f85978cb049af5e4ef23bd8b688715,
built on 09/04/2020 12:44 GMT
2020-11-20 02:26:42,397 [myid:] - INFO  [main:Environment@98] - Client
environment:host.name=zookeeper
2020-11-20 02:26:42,397 [myid:] - INFO  [main:Environment@98] - Client
environment:java.version=11.0.9.1
2020-11-20 02:26:42,398 [myid:] - INFO  [main:Environment@98] - Client
environment:java.vendor=Oracle Corporation
2020-11-20 02:26:42,398 [myid:] - INFO  [main:Environment@98] - Client
environment:java.home=/usr/local/openjdk-11
...
...
Welcome to ZooKeeper!
2020-11-20 02:26:42,475 [myid:127.0.0.1:2181] - INFO  [main-
SendThread(127.0.0.1:2181):ClientCnxn$SendThread@1167] - Opening socket
connection to server localhost/127.0.0.1:2181.
2020-11-20 02:26:42,475 [myid:127.0.0.1:2181] - INFO  [main-
SendThread(127.0.0.1:2181):ClientCnxn$SendThread@1169] - SASL config status:
Will not attempt to authenticate using SASL (unknown error)
JLine support is enabled
2020-11-20 02:26:42,491 [myid:127.0.0.1:2181] - INFO  [main-
SendThread(127.0.0.1:2181):ClientCnxn$SendThread@999] - Socket connection
established, initiating session, client: /127.0.0.1:40896, server:
localhost/127.0.0.1:2181
2020-11-20 02:26:42,511 [myid:127.0.0.1:2181] - INFO  [main-
SendThread(127.0.0.1:2181):ClientCnxn$SendThread@1433] - Session establishment
complete on server localhost/127.0.0.1:2181, session id = 0x10013cdbf990003,
negotiated timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null

[zk: 127.0.0.1:2181(CONNECTED) 0] create /hello-zone 'world'
Created /hello-zone
[zk: 127.0.0.1:2181(CONNECTED) 1] ls /
[dubbo, hello-zone, zookeeper]
[zk: 127.0.0.1:2181(CONNECTED) 2]
```

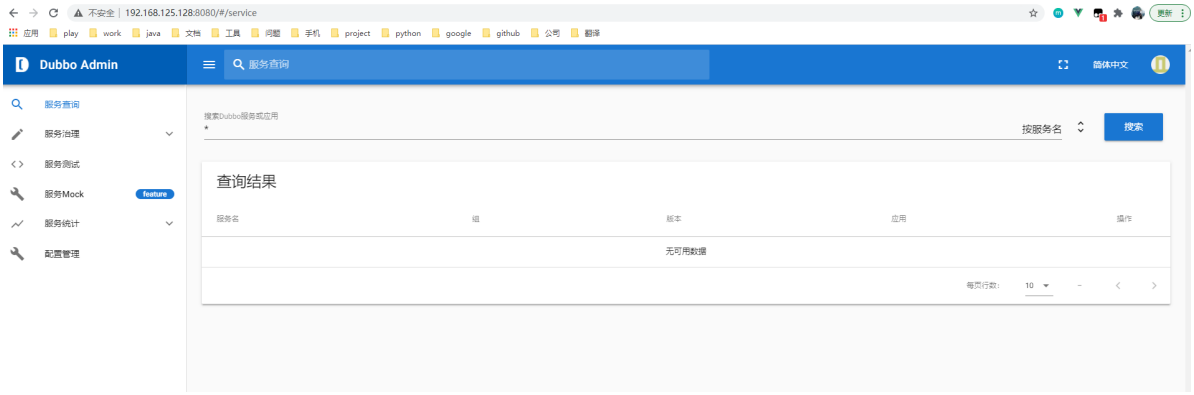
去git 的dubbo 源码， 然后打jar包，上传到服务器。

这个时候直接启动jar 就能访问到了。 `java -jar  dubbo-admin-0.2.0-SNAPSHOT.jar`

注意这个地方我的那个application.properties并未改动。

```
admin.registry.address=zookeeper://127.0.0.1:2181
admin.config-center=zookeeper://127.0.0.1:2181
admin.metadata-report.address=zookeeper://127.0.0.1:2181
```

请看页面效果：



Docker compose实战zookeeper集群 （有问题 -未解决）

伪集群模式：

在原来的使用是`server zoo1:3888`总是出现问题。新版本推荐使用以下

```
version: '3.1'
services:
  zoo1:
    image: zookeeper
    restart: always
    hostname: zoo1
    ports:
      - 2181:2181
    environment:
      ZOO_MY_ID: 1
      ZOO_SERVERS: server.1=0.0.0.0:2888:3888 server.2=zoo2:2888:3888
server.3=zoo3:2888:3888

  zoo2:
    image: zookeeper
    restart: always
    hostname: zoo2
    ports:
      - 2182:2181
    environment:
      ZOO_MY_ID: 2
      ZOO_SERVERS: server.1=zoo1:2888:3888 server.2=0.0.0.0:2888:3888
server.3=zoo3:2888:3888

  zoo3:
    image: zookeeper
    restart: always
    hostname: zoo3
    ports:
      - 2183:2181
    environment:
      ZOO_MY_ID: 3
      ZOO_SERVERS: server.1=zoo1:2888:3888 server.2=zoo2:2888:3888
server.3=0.0.0.0:2888:3888
```

- 分别以交互方式进入容器查看

```
docker exec -it zookeeper_zoo1_1 /bin/bash
docker exec -it zookeeper_zoo1_2 /bin/bash
docker exec -it zookeeper_zoo1_3 /bin/bash
```

Docker compose实战 gitlab

docker-compose.yml

```
version: '3'
  services:
    gitlab:
    restart: always
    image: twang2218/gitlab-ce-zh
    hostname: '192.168.147.132'
    environment:
      TZ: 'Asia/Shanghai'
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'http://192.168.147.132:8080'
        gitlab_rails['gitlab_shell_ssh_port'] = 2222
        unicorn['port'] = 8888
        nginx['listen_port'] = 8080
    ports:
      - '8080:8080'
      - '8443:443'
      - '2222:22'
      volumes:
        - /usr/local/docker/gitlab/config:/etc/gitlab
        - /usr/local/docker/gitlab/data:/var/opt/gitlab
        - /usr/local/docker/gitlab/logs:/var/log/gitlab
```

Docker compose实战 nexus

```
version: '3.1'
  services:
    nexus:
      restart: always
      image: sonatype/nexus3
      container_name: nexus
      ports:
        - 8081:8081
        volumes:
          - /usr/local/docker/nexus/data:/nexus-data
```

通过：http://ip:port/用户名：密码：验证

Docker compose 实战ElasticSecarch

docker-compose.yml

```yaml
version: '3.3'
services:
  elasticsearch:
    image: wutang/elasticsearch-shanghai-zone:6.3.2
    container_name: elasticsearch
    restart: always
    ports:
      - 9200:9200
      - 9300:9300
    environment:
      cluster.name: elasticsearch
```

访问： http://elasticsearchIP:9200/ 查看是否成功

Docker compose编排容器

一般使用先进行安装环境然后通过docker-compose.yml进行编排启动各个服务。

比如我们有一个 spring cloud项目

Docker-Registry:

Docker 私服:

cd /usr/local/docker
创建 registry目录
使用docker compose命令直接使用:

```yaml
version: '3.1'
services:
    registry:
        image: registry
        restart: always
        container_name: registry
        ports:
            - 5000:5000
        volumes:
            - /usr/local/docker/registry/data:/var/lib/registry
```

只要安装之后就可以了，这里只是服务端，同样我们需要客户端测试：http://ip:5000/v2/
我们在开发时构建服务端之后，在客户端构建一个服务之后上传到这个docker私服中有其他人使用的话直接在这个地方拉取：同时需要在客户端配置：/etc/docker/daemon.json `daemon.json中配置:`

```
{
    "registry-mirrors": [
        "https://registry.docker-cn.com"
    ],
    "insecure-registries": [
        "ip:5000"
    ]
}
```

之后重启就可以：

`sudo systemctl daemon-reload`

`sudo systemctl restart docker`

测试镜像上传：

拉取一个镜像

docker pull nginx

```
## 查看全部镜像 docker images
## 标记本地镜像并指向目标仓库（ip:port/image_name:tag，该格式为标记版本号）
    docker tag nginx 192.168.75.133:5000/nginx
## 提交镜像到仓库
    docker push 192.168.75.133:5000/nginx
查看全部镜像：
    curl -XGET http://192.168.75.133:5000/v2/_catalog
部署Docker Registry WebUI
私服安装成功后就可以使用 docker 命令行工具对 registry 做各种操作了。然而不太方便的地方是不能直观的查看 registry 中的资源情况。如果可以使用 UI 工具管理镜像就更好了。这里介绍两个 Docker Registry WebUI 工具
    docker-registry-frontend
    docker-registry-web
```

在客户端中docker-compose中使用这个：

```
version: '3.1'
services:
  frontend:
    image: konradkleine/docker-registry-frontend:v2
    ports:
      - 8080:80
    volumes:
      - ./certs/frontend.crt:/etc/apache2/server.crt:ro
      - ./certs/frontend.key:/etc/apache2/server.key:ro
    environment:
      - ENV_DOCKER_REGISTRY_HOST=192.168.75.133
      - ENV_DOCKER_REGISTRY_PORT=5000
```