Docker实战应用（二）

Docker 中的网络功能介绍

Docker 允许通过外部访问容器或容器互联的方式来提供网络服务。

外部访问容器

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 `-P` 或 `-p` 参数来指定端口映射。

当使用 `-P` 标记时，Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口。

使用 `docker ps -l` 可以看到，各个容器的端口映射。

`-p` 则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。支持的格式有 `ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort` 。

映射所有接口地址

使用 `hostPort:containerPort` 格式本地的 5000 端口映射到容器的 5000 端口，可以执行

```
docker run -d -p 5000:5000 --name 容器名 镜像名:Tag
```

映射到指定地址的指定端口

可以使用 `ip:hostPort:containerPort` 格式指定映射使用一个特定地址，比如 localhost 地址 127.0.0.1

```
docker run -d -p 127.0.0.1:5000:5000 --name 容器名 镜像名:Tag
```

映射到指定地址的任意端口

使用 `ip::containerPort` 绑定 localhost 的任意端口到容器的 5000 端口，本地主机会自动分配一个端口。

```
docker run -d -p 127.0.0.1::5000 --name 容器名 镜像名:Tag
```

注意：

- 容器有自己的内部网络和 ip 地址（使用 docker inspect 可以获取所有的变量，Docker还可以有一个可变的网络配置。）
- `-p` 标记可以多次使用来绑定多个端口

例如:

```
docker run -d \
-p 5000:5000 \
-p 3000:80 \
training/webapp \
python app.py
```

容器互联

新建网络

下面先创建一个新的 Docker 网络。

```
docker network create -d bridge my-net
```

`-d` 参数指定 Docker 网络类型，有 bridge overlay 。其中 overlay 网络类型用于Swarm mode，资料中可查看到。

使用`docker network ls`可以查看当前创建的网络列表。

连接容器

运行一个容器并连接到新建的 my-net 网络

```
docker run -it --name busybox1 --network my-net busybox sh
```

打开新的终端，再运行一个容器并加入到 my-net 网络

```
docker run -it --name busybox2 --network my-net busybox sh
```

下面通过 `ping` 来证明 `busybox1` 容器和 `busybox2` 容器建立了互联关系。

```
# 在busybox1中执行
ping busybox2
# 在busybox2中执行
ping busybox1
```

如果有多个容器之间需要互相连接，推荐使用 Docker Compose。

解决docker中无法使用vim

首先进入到需要使用vim命令行的容器

1 更新系统

```
apt-get update
```

2 安装vim

```
apt-get install -y vim
```

3 如果安装失败，尝试修改镜像源为国内镜像，然后再从第1步执行

```
mv /etc/apt/sources.list /etc/apt/sources.list.bak
echo "deb http://mirrors.163.com/debian/ jessie main non-free contrib" >/etc/apt/sources.list
echo "deb http://mirrors.163.com/debian/ jessie-proposed-updates main non-free contrib" >>/etc/apt/sources.list
echo "deb-src http://mirrors.163.com/debian/ jessie main non-free contrib" >>/etc/apt/sources.list
echo "deb-src http://mirrors.163.com/debian/ jessie-proposed-updates main non-free contrib" >>/etc/apt/sources.list
```

Dockerfile

Linux系统组成部分

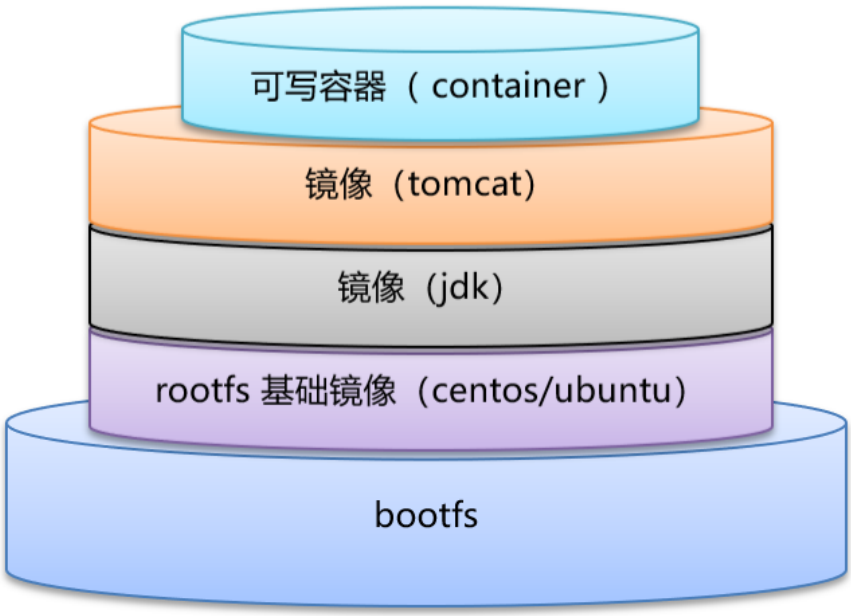
- bootfs：包含bootloader（引导加载程序）和 kernel（内核）
- rootfs： root文件系统，包含的就是典型 Linux 系统中的/dev，/proc，/bin，/etc等标准目录和文件
- 不同的linux发行版，bootfs基本一样，而rootfs不同，如ubuntu，centos等

Docker镜像组成

回顾一下之前我们学到的知识，镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，是在以镜像为基础层，在其基础上加一层作为容器运行时的存储层。

- Docker镜像是由特殊的文件系统叠加而成
- 最底端是 bootfs，并使用宿主机的bootfs
- 第二层是 root文件系统rootfs,称为base image
- 然后再往上可以叠加其他的镜像文件
- 统一文件系统（Union File System）技术能够将不同的层整合成一个文件系统，为这些层提供了一个统一的视角，这样就隐藏了多层的存在，在用户的角度看来，只存在一个文件系统。
- 一个镜像可以放在另一个镜像的上面。位于下面的镜像称为父镜像，最底部的镜像成为基础镜像。
- 当从一个镜像启动容器时，Docker会在最顶层加载一个读写文件系统作为容器

如Tomcat镜像：



也就是说镜像的本质就是一个分层的文件系统。

Docker镜像制作

容器转为镜像

```
# 将容器转换成镜像,注意：数据卷不会写到镜像
docker commit 容器ID或容器名 镜像名称:版本号
# 将镜像压缩成文件
docker save -o 文件名称 镜像名称:版本号
# 将压缩文件转换成镜像
docker load -i 压缩文件名称
```

使用示例

```
docker commit 121332 mytomcat:1.0
docker save -o mytomcat.tar mytomcat:1.0
docker load -i mytomcat.tar
```

还可使用`--author`指定修改作者，`--message`记录修改内容，这和git版本控制像相似。

注意:

1. 使用 docker commit 命令虽然可以比较直观的帮助理解镜像分层存储的概念，但是实际环境中并不会这样使用。
2. 首先，如果值很小的在容器内进行文件操作不会有太多影响，如果是安装软件包、编译构建，那么会有大量的无关内容添加进来，如果没有清理，那么镜像会比较臃肿。
3. 此外，使用 docker commit 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被称为黑箱镜像，换句话说，就是除了制作镜像的人知道执行过什么命令、怎么生成的镜像，别人根本无从得知。
4. 而且，镜像时分层存储的文件系统，除当前层可以被修改外，其他层都是不变的，如果反复的对镜像进行修改，会让镜像越来越臃肿。
5. `docker commit` 命令除了学习之外，还有一些特殊的应用场合，比如被入侵后保存现场等。如果要定制镜像，应该使用 Dockerfile 来完成。

使用 Dockerfile 定制镜像

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。这个脚本就是 Dockerfile。

Dockerfile概述

- Dockerfile 是一个文本文件
- 包含了一条条的指令
- 每一条指令构建一层，基于基础镜像，最终构建出一个新的镜像
- 对于开发人员：可以为开发团队提供一个完全一致的开发环境
- 对于测试人员：可以直接拿开发时所构建的镜像或者通过Dockerfile文件构建一个新的镜像开始工作了
- 对于运维人员：在部署时，可以实现应用的无缝移植

Dockerfile关键词

关键字	作用	备注
FROM	指定父镜像	指定dockerfile基于那个image构建
MAINTAINER	作者信息	用来标明这个dockerfile谁写的
LABEL	标签	用来标明dockerfile的标签 可以使用Label代替Maintainer 最终都是在docker image基本信息中可以查看
RUN	执行命令	执行一段命令 默认是/bin/sh 格式: RUN command 或者 RUN ["command" , "param1","param2"]
CMD	容器启动命令	提供启动容器时候的默认命令 和ENTRYPOINT配合使用.格式 CMD command param1 param2 或者 CMD ["command" , "param1","param2"]
ENTRYPOINT	入口	一般在制作一些执行就关闭的容器中会使用
COPY	复制文件	build的时候复制文件到image中
ADD	添加文件	build的时候添加文件到image中 不仅仅局限于当前build上下文 可以来源于远程服务
ENV	环境变量	指定build时候的环境变量 可以在启动的容器的时候 通过-e覆盖 格式ENV name=value
ARG	构建参数	构建参数 只在构建的时候使用的参数 如果有ENV 那么ENV的相同名字的值始终覆盖arg的参数
VOLUME	定义外部可以挂载的数据卷	指定build的image那些目录可以启动的时候挂载到文件系统中 启动容器的时候使用 -v 绑定 格式 VOLUME ["目录"]
EXPOSE	暴露端口	定义容器运行的时候监听的端口 启动容器的使用-p来绑定暴露端口 格式: EXPOSE 8080 或者 EXPOSE 8080/udp
WORKDIR	工作目录	指定容器内部的工作目录 如果没有创建则自动创建 如果指定/使用的是绝对地址 如果不是/开头那么是在上一条workdir的路径的相对路径
USER	指定执行用户	指定build或者启动的时候 用户 在RUN CMD ENTRYPONT执行的时候的用户
HEALTHCHECK	健康检查	指定监测当前容器的健康监测的命令 基本上没用 因为很多时候应用本身有健康监测机制
ONBUILD	触发器	当存在ONBUILD关键字的镜像作为基础镜像的时候 当执行FROM完成之后 会执行 ONBUILD的命令 但是不影响当前镜像用处也不怎么大
STOPSIGNAL	发送信号量到宿主机	该STOPSIGNAL指令设置将发送到容器的系统调用信号以退出。

关键字	作用	备注
SHELL	指定执行脚本的shell	指定RUN CMD ENTRYPOINT 执行命令的时候 使用的shell

我们来定制一个简单Nginx镜像

1 在一个空白目录中，建立一个文本文件，并命名为 Dockerfile

```
mkdir mynginx
cd mynginx
touch Dockerfile
```

2 编辑Dockerfile，输入下面内容

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

在 Docker Store 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如nginx 、redis 、 mongo 、 mysql 、 httpd 、 php 、 tomcat 等；除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 `scratch` 。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。

构建镜像，执行命令

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

- -f指定dockerfile文件名
- -t指定镜像名（镜像名:tag）

3 使用案例

```
docker build -f ./Dockerfile -t mynginx:1.0  .
```

4 启动mynginx

```
#复制之前的nginx配置文件
cp -r ~/nginx/conf ~/mynginx/
#构建容器
docker run -it --name mynginx \
-p 81:80 \
-v $PWD/conf/nginx.conf:/etc/nginx/nginx.conf \
mynginx:1.0
```

5 测试访问

```
http://ip:81/
```

Docker 服务编排

服务编排

微服务架构的应用系统中一般包含若干个微服务，每个微服务一般都会部署多个实例，如果每个微服务都要手动启停，维护的工作量会很大。

包括：

- 要从Dockerfile build image 或者去dockerhub拉取image
- 要创建多个container
- 要管理这些container（启动停止删除）

所谓服务编排就是按照一定的业务规则批量管理容器。

Docker Compose

概述

Docker Compose是一个编排多容器分布式部署的工具，提供命令集管理容器化应用的完整开发周期，包括服务构建，启动和停止。使用步骤：

1.利用 Dockerfile 定制镜像

2.使用 docker-compose.yml 定义组成应用的各服务

3.运行 docker-compose up 启动应用

模板文件

模板文件是使用 Compose 的核心，涉及到的指令关键字也比较多。但大家不用担心，这里面大部分指令跟 docker run 相关参数的含义都是类似的。
默认的模板文件名称为 docker-compose.yml，格式为 YAML 格式。

比如：

```
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。

build

指定 Dockerfile 所在文件夹的路径（可以是绝对路径，或者相对 docker-compose.yml 文件的路径）。 Compose 将会利用它自动构建这个镜像，然后使用这个镜像。

例如：

```
version: '3'
services:
 webapp:
  build: ./dir
```

你也可以使用 `context` 指令指定 Dockerfile 所在文件夹的路径。

使用 `dockerfile` 指令指定 Dockerfile 文件名。

使用 `arg` 指令指定构建镜像时的变量。

例如：

```
version: '3'
 services:
  webapp:
   build:
    context: ./dir
    dockerfile: Dockerfile-alternate
    args:
     buildno: 1
```

image

指定为镜像名称或镜像 ID。如果镜像在本地不存在， Compose 将会尝试拉去这个镜像。

```
version: '3'
 services:
  webapp:
   image: centos:7
```

command

覆盖容器启动后默认执行的命令。

```
command: echo "hello world"
```

depends_on

解决容器的依赖、启动先后的问题。以下例子中会先启动 redis db 再启动 web

```
version: '3'
services:
 web:
  build: .
  depends_on:
   - db
   - redis
 redis:
  image: redis
 db:
  image: postgres
```

注意： web 服务不会等待 redis db 「完全启动」之后才启动。

tmpfs

挂载一个 tmpfs 文件系统到容器。

```
tmpfs: /run
tmpfs:
- /run
- /tmp
```

expose

暴露容器端口，但不映射到宿主机，只被连接的服务访问。
仅可以指定内部端口为参数

```
expose:
 - "3000"
 - "8000"
```

extra_hosts

类似 Docker 中的 --add-host 参数，指定额外的 host 名称映射信息。

```
extra_hosts:
 - "googledns:8.8.8.8"
 - "dockerhub:52.1.157.61"
```

会在启动后的服务容器中 /etc/hosts 文件中添加如下两条条目。

```
8.8.8.8 googledns
52.1.157.61 dockerhub
```

logging

配置日志选项。

```
logging:
 driver: syslog
 options:
  syslog-address: "tcp://192.168.0.42:123"
```

目前支持三种日志驱动类型。

```
driver: "json-file"
driver: "syslog"
driver: "none"
```

network_mode

设置网络模式。使用和 `docker run` 的 --network 参数一样的值。

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

networks

配置容器连接的网络。

```
version: "3"
services:
 some-service:
  networks:
   - some-network
   - other-network
 networks:
  some-network:
  other-network:
```

volumes

数据卷所挂载路径设置。可以设置宿主机路径 （ HOST:CONTAINER ） 或加上访问模式 （ HOST:CONTAINER:ro ）。

该指令中路径支持相对路径。

```
volumes:
 - /var/lib/mysql
 - cache/:/tmp/cache
 - ~/configs:/etc/configs/:ro
```

通过官网查看更多用法：https://docs.docker.com/compose/compose-file/

安装与使用

Compose目前已经完全支持Linux、Mac OS和Windows，在我们安装Compose之前，需要先安装Docker。

二进制包版本查看：https://github.com/docker/compose/releases

1、安装Docker Compose

```
# 下面我们以编译好的二进制包方式安装在Linux系统中。
curl -L https://github.com/docker/compose/releases/download/1.26.0/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
# 设置文件可执行权限
chmod +x /usr/local/bin/docker-compose
# 查看版本信息
docker-compose -v
```

2、卸载Docker Compose

```
# 二进制包方式安装的，删除二进制文件即可
rm /usr/local/bin/docker-compose
```

3、Docker compose的使用

```
比如来编排nginx + springboot项目
```

1. 创建docker-compose目录

```
mkdir ~/docker-compose
cd ~/docker-compose
```

2. 编写 docker-compose.yml 文件

```
version: '3'
services:
  project-front:
   image: project-front:1.0
   ports:
    - 81:80
   links:
    - project-gateway
   volumes:
    - ./nginx/conf.d:/etc/nginx/conf.d
  project-gateway:
    image: project-gateway:1.0
    expose:
      - "10096"
  project-login:
    image: project-login:1.0
    expose:
      - "10106"
```

3. 创建./nginx/conf.d目录

```
mkdir -p ./nginx/conf.d
```

4. 在./nginx/conf.d目录下编写springboot.conf文件

```
server {
    listen 80;
    access_log off;
    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
        try_files $uri $uri/ /index.html;
    }
    error_page 500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
```

```
    }
    location /api/ {
        rewrite ^/api/(.*) /$1 break;
        proxy_pass http://project-gateway:10096;
    }
}
```

5. 在~/docker-compose 目录下 使用docker-compose 启动容器

```
# 前台启动
docker-compose up
# 后台启动
docker-compose up -d
```

6. 测试访问

```
http://192.168.28.133:81
```

Docker 私有仓库

Docker官方的Docker hub（https://hub.docker.com）是一个用于管理公共镜像的仓库，我们可以从上面拉取镜像 到本地，也可以把我们自己的镜像推送上去。但是，有时候我们的服务器无法访问互联网，或者你不希望将自己的镜 像放到公网当中，那么我们就需要搭建自己的私有仓库来存储和管理自己的镜像。

私有仓库搭建

```
# 1、拉取私有仓库镜像
docker pull registry
# 2、启动私有仓库容器
docker run -id --name registry -p 5000:5000 registry
# 3、打开浏览器 输入地址http://私有仓库服务器ip:5000/v2/_catalog，看到{"repositories":[]} 表示私有仓库 搭建成功
# 4、修改daemon.json
vim /etc/docker/daemon.json
# 在上述文件中添加一个key，保存退出。此步用于让 docker 信任私有仓库地址；注意将私有仓库服务器ip修改为自己私有仓库服务器真实ip
{"insecure-registries":["私有仓库服务器ip:5000"]}
# 5、重新加载配置
systemctl daemon-reload
# 6、重启docker 服务
systemctl restart docker
# 7、启动私有仓库容器
docker start registry
```

将镜像上传至私有仓库

```
# 1、标记镜像为私有仓库的镜像
docker tag centos:7 私有仓库服务器IP:5000/centos:7
# 2、上传标记的镜像
docker push 私有仓库服务器IP:5000/centos:7
```

从私有仓库拉取镜像

```
#拉取镜像
docker pull 私有仓库服务器ip:5000/centos:7
```