### Dockerfile自定义镜像：

#### 前言

前面已经学了如何拉取镜像，如何基于镜像去创建并运行容器，如何进行数据卷操作，不过前面操作的镜像都是由DockerHub官方制作的镜像，但是我们自己的项目，比如写的微服务代码，DockerHub官方显然是不会帮我们制作镜像，所以说我们的微服务项目一定是自己制作镜像的，方法就是**可以基于DockerFile来制作、自定义镜像**。

#### 镜像结构

因为要制作镜像，必须得先了解镜像内部的构成

镜像的组成：镜像就是将应用程序及其需要的系统函数库、环境、配置、依赖打包到一起而成。

镜像的结构要看镜像的组成怎么要去组合：比如这些组成有没有相互的依赖关系、顺序？显然是有的，如果没有底层的系统函数库、底层的文件系统，怎么去完成环境的配置？如果没有配置环境变量，怎么去做依赖的安装？如果没有依赖的安装，怎么去完成应用的安装？如果没有应用安装，怎么去做应用配置？可见，镜像不仅仅是吧以上组成糅合在一起，而且是按照一定的顺序去分层构建的。

以mysql镜像为例，讲解镜像结构：

如下图，mysql镜像就是按照上面分析的依赖顺序去分层的：

1：要想构建一个镜像，最底层，一定是镜像所依赖的系统函数库，此图中mysql就是用了ubutun的操作系统，当然不是完整系统，只是mysql依赖的部分系统函数库和部分文件，此层就像整个大楼的地基一样，只有把这层搞定，才能继续向上，所以这层我们称为：**BASE IMAGE（基础镜像：应用依赖的系统函数库、环境、配置、文件等）**

2：在Base Image基础上，就可以给mysql应用配置环境变量、以及mysql需要依赖的其他东西（此图中没有画出来）

3：在前面基础上，就可以安装mysql了，那么就可以把mysql应用的安装包拷进来

4：然后在前面基础上，就可以去rpm去安装mysql

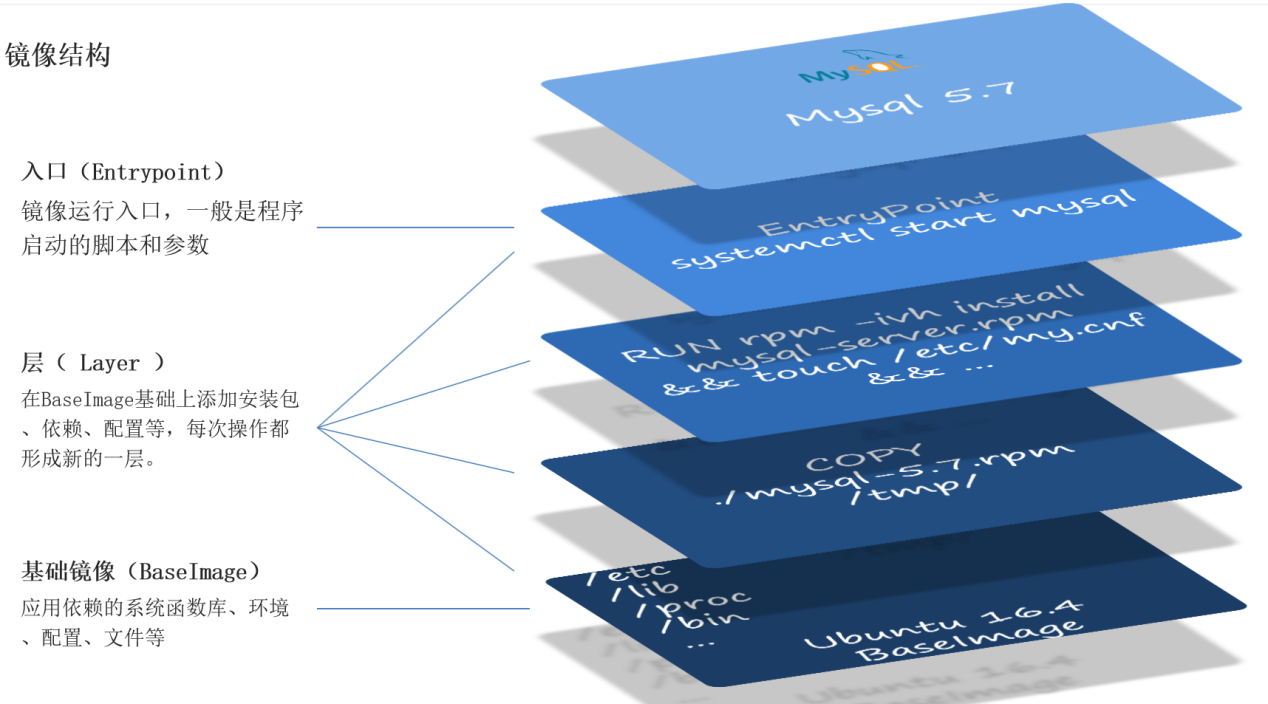
5：安装完mysql，还要去配置MySQLd的配置文件

6：等所有安装步骤都做完，这个楼就盖得差不多了，还差一个东西：入口（EntryPoint：镜像运行入口，一般是程序启动的脚本和参数），因为你把这一堆文件揉成一块，打成一个包，将来别人怎么启动里面的应用呢？你得给他一个入口（或者理解成引线，引爆点），也就是启动的脚本，任何的镜像都有这一层。

到此，镜像就构建完成了，我们这个过程中，从基础开始，完成了环境变量的配置、依赖的安装、应用的安装等等，这每一次的操作，都会产生新的一层：Layer（层：在Base Image基础上添加安装包、依赖、配置等，每次操作都形成新的一层）

问题：有先后顺序可以理解，有依赖关系嘛，有先有后，但为什么要分层呢？

比如，如果不分层，mysql5.7的镜像要换成5.8的，那么就不能在原来的基础上改了，因为不分层，全混成一团了，就只能从零开始再构建一个；但是分层了，就不同了，因为逐层构建，要构建5.8的时候，会发现只是不需要基础镜像以上的那些层，那么我们就可以基于基础镜像继续构建5.8，提高复用性，提高效率。所以，将来我们自己构建镜像，也应该从基础开始逐层构建（就是为了复用！）。



##### 总结：

镜像是分层结构，每一层成为一个Layer:

1：BaseImage层：包含基本的系统函数库、环境变量、文件系统；

2：EntryPoint：入口，是镜像中应用启动的命令；

3：其他：在BaseImage基础上添加依赖、安装程序、完成整个应用的安装与配置。

#### Dockerfile语法：

##### 什么是Dockerfile?

自定义镜像一定会用到Dockerfile，Dockerfile是一个文本文件，其中包含一个个的指令（Instruction），用指令来说明要执行什么操作来构建镜像，于是Dockerfile可以当作是镜像构建的说明书，dockerh会安装Dockerfile中的指令去构建镜像。每一个指令都会形成一层Layer。

##### 指令说明：

**FROM**：指定基础镜像（Base Image），一般是操作系统

eg：FROM centos:7

**ENV**：设置环境变量，一般是键值对，一旦配好，在后续的过程中可以使用环境变量中配好的东西，比如我们可以配好一个目录，那么后续使用过程中，大家都可以用这个目录。

eg：ENV kay value

**COPY**：构建镜像时拷贝本地文件到镜像的指定目录，比如我们本地的java项目包要拷贝到镜像里面去，就可以用这个

eg：COPY ./mysql-5.7.rpm /tmp

**RUN**：执行linux的shell命令，一般是安装过程的命令，比如安装什么依赖、或者解压，都可以用RUN后面跟上对应的命令就行

eg：RUN yum install gcc

**EXPOSE**：指定容器运行时监听的端口，即暴露容器端口，是给镜像使用者看的

eg：EXPOSE 8080

**ENTRYPOINT**：镜像中应用的启动命令，容器运行时调用，就是前面说的镜像的启动脚本

eg：ENTRYPOINT java -jar xx.jar

更新详细语法说明，请参考官网文档：

<https://docs.docker.com/engine/reference/builder>

#### 将java项目构建成镜像

##### eg：基于Ubuntu镜像构建一个新镜像，运行一个java项目

步骤一：新建一个空文件夹：docker-demo：mkdir -p /tmp/docker-demo

步骤二：拷贝课前资料中的docker-demo.jar文件到docker-demo这个目录

步骤三：拷贝课前资料中的jar8.tar.gz文件到docker-demo这个目录

步骤四：拷贝课前资料提供的Dockerfile到docker-demo这个目录

# 指定基础镜像

FROM ubuntu:16.04

# 配置环境变量，指定JDK的安装目录，目录是镜像中的文件系统中的目录

ENV JAVA\_DIR=/usr/local

# 拷贝jdk和java项目的包，拷贝的源是宿主机文件系统中的目录，目的地是镜# 像文件系统的目录

COPY ./jdk8.tar.gz $JAVA\_DIR/

COPY ./docker-demo.jar /tmp/app.jar

# 安装JDK （1：进入安装目录；2：解压缩jdk压缩包；3：重命名）

RUN cd $JAVA\_DIR \

&& tar -xf ./jdk8.tar.gz \

&& mv ./jdk1.8.0\_144 ./java8

# 配置环境变量

ENV JAVA\_HOME=$JAVA\_DIR/java8

ENV PATH=$PATH:$JAVA\_HOME/bin

# 暴露端口

EXPOSE 8090

# 入口，java项目的启动命令

ENTRYPOINT java -jar /tmp/app.jar

步骤五：进入docker-demo目录

步骤六：利用dockerfile构建镜像：**docker build -t javaweb:1.0 .**

**-t**：tag的缩写

**javaweb**：镜像名

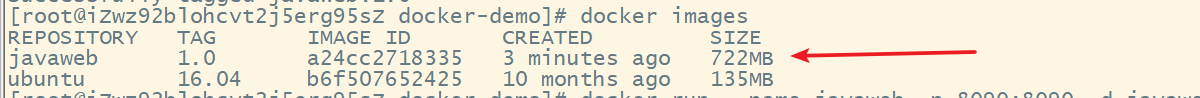
**1.0**：版本

**.** 注意，一定要加点，这个点代表的是dockerfile所在的目录，点表示dockerfile就在当前目录中。



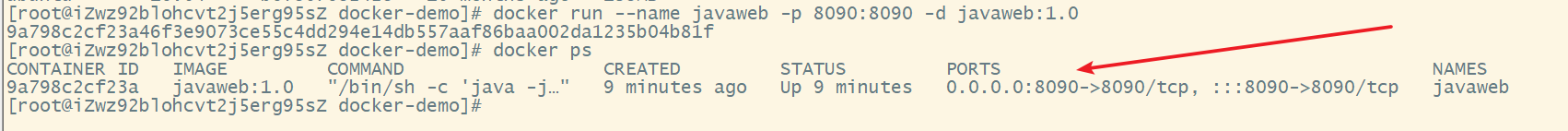
上截图可以看到构建镜像时会逐步执行指令！没执行一个指令就会有一个新的层（Layer）出现！

步骤七：查看镜像：docker images



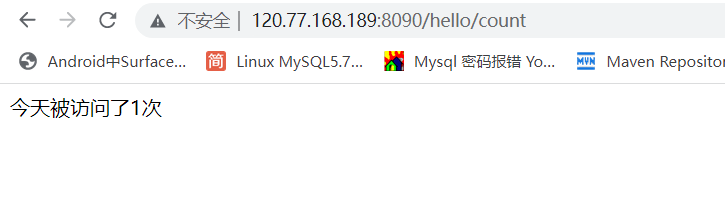
步骤八：运行镜像：

docker run --name javaweb -p 8090:8090 -d javaweb:1.0



步骤九：验证，浏览器访问项目：

http://120.77.168.189:8090/hello/count



##### 上面案例问题

上面的案例的镜像我们是基于Ubuntu镜像（Base Image）构建的，这个案例镜像的构建会发现还是挺麻烦的，由dockerfile文件可看到，构建过程分为很多步。而在这么多步骤中，其实真正来构建java项目的只有下面黑色标识的两行：

# 指定基础镜像

FROM ubuntu:16.04

# 配置环境变量，JDK的安装目录，目录时镜像中的文件系统中的目录

ENV JAVA\_DIR=/usr/local

# 拷贝jdk和java项目的包，拷贝的源是宿主机文件系统中的目录，目的地是镜# 像文件系统的目录

COPY ./jdk8.tar.gz $JAVA\_DIR/

**COPY ./docker-demo.jar /tmp/app.jar**

# 安装JDK

RUN cd $JAVA\_DIR \

&& tar -xf ./jdk8.tar.gz \

&& mv ./jdk1.8.0\_144 ./java8

# 配置环境变量

ENV JAVA\_HOME=$JAVA\_DIR/java8

ENV PATH=$PATH:$JAVA\_HOME/bin

# 暴露端口

EXPOSE 8090

# 入口，java项目的启动命令

**ENTRYPOINT java -jar /tmp/app.jar**

**当我们把DockerFile中拷贝项目jar包动作放到下面时，更直观地知道，红字部分的动作都是在安装jdk，那么问题来了，如果说我们又有一个微服务要构建了，将来也会有成千上万的微服务要构建，每个微服务在做构建时，都要做红色部分的流程，太麻烦了！**

**于是，按照之前说的，我们的镜像即然已经做了分层，而以下例子中黑色部分的前几层，不管是这个java项目，还是再换一个java项目，这红色部分的前几层都是一样的，那么我们可以把红色部分的这几层提前构建好作为一个镜像（Base Image），以后都在这个镜像的基础上在进行构建，就方便多了，这就是分层的好处！**

# 指定基础镜像

FROM ubuntu:16.04

# 配置环境变量，JDK的安装目录，目录时镜像中的文件系统中的目录

ENV JAVA\_DIR=/usr/local

# 拷贝jdk和java项目的包，拷贝的源是宿主机文件系统中的目录，目的地是镜# 像文件系统的目录

COPY ./jdk8.tar.gz $JAVA\_DIR/

# 安装JDK

RUN cd $JAVA\_DIR \

&& tar -xf ./jdk8.tar.gz \

&& mv ./jdk1.8.0\_144 ./java8

# 配置环境变量

ENV JAVA\_HOME=$JAVA\_DIR/java8

ENV PATH=$PATH:$JAVA\_HOME/bin

**COPY ./docker-demo.jar /tmp/app.jar**

# 暴露端口

EXPOSE 8090

# 入口，java项目的启动命令

**ENTRYPOINT java -jar /tmp/app.jar**

#### eg：基于java:8-alpine镜像，将一个项目构建为镜像

**java:8-apline镜像是一个体积非常小的jdk的镜像，就是说这个镜像已经帮我们把上面红色部分的全部做了，那么，如果我们构建镜像是以java:8-apline为基础，那么红色部分的就可以省掉不做了！**

**那么，新的dockerFile就变成了：**

# 指定基础镜像

FROM java:8-alpine

**COPY ./docker-demo.jar /tmp/app.jar**

# 暴露端口

EXPOSE 8090

# 入口，java项目的启动命令

**ENTRYPOINT java -jar /tmp/app.jar**

**实现思路如下**

**1：新建一个空的目录，然后在目录中新建一个文件，命名为Dockerfile;**

**2：拷贝课前资料提供的docker-demo.jar到这个目录；**

**3：编写dockerfile文件：**

# 指定基础镜像

FROM java:8-alpine

**COPY ./docker-demo.jar /tmp/app.jar**

# 暴露端口

EXPOSE 8090

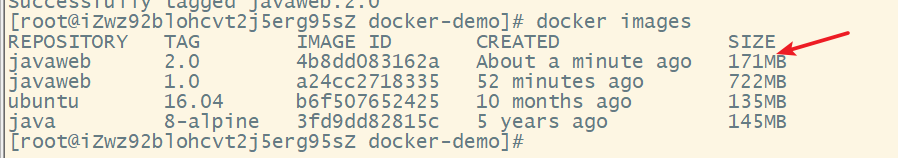
# 入口，java项目的启动命令

**ENTRYPOINT java -jar /tmp/app.jar**

**4：使用docker build命令构建镜像：docker build -t javaweb:2.0 .**



可以发现，步骤只剩下四步了，而且镜像体积也比1.0版本小了很多



**5使用docker run创建容器并运行**

docker run --name javaweb -p 8090:8090 -d javaweb:2.0

#### 总结：

Dockerfile的本质就是一个文件，通过指令描述镜像的构建过程；

DockerFile的第一行必须是From，从一个基础镜像来构建；

基础镜像可以是基本操作系统，比如Ubuntu。也可以是其他人制作好的镜像，例如：java:8-alpine。