1 概念

Docker 守护进程会解析整个 dockerfile 文件内容，执行一条条的指令，Docker 守护进程会翻译成真正的 linux 指令

2 语法说明

2.1 整体说明

2.1.1 常用指令

指令	说明
FROM	设置镜像使用的基础镜像
MAINTAINER	设置镜像的作者
RUN	编译镜像时运行的脚本
CMD	设置容器的启动命令
LABEL	设置镜像的标签
EXPOESE	设置镜像暴露的端口
ENV	设置容器的环境变量
ADD	编译镜像时复制文件到镜像中
COPY	编译镜像时复制文件到镜像中
ENTRYPOINT	设置容器的入口程序
VOLUME	设置容器的挂载卷
USER	设置运行RUN CMD ENTRYPOINT的用户名
WORKDIR	设置RUN CMD ENTRYPOINT COPY ADD指令的工作目录
ARG	设置编译镜像时加入的参数
ONBUILD	设置镜像的ONBUILD指令
STOPSIGNAL	设置容器的退出信号量

FROM　基于哪个镜像构建新镜像
MAINTAINER（弃用）　镜像维护者信息
LABEL　同上，但用法更加灵活
RUN 构建镜像时运行的 Shell 命令
COPY　拷贝文件或目录到镜像中
CMD　运行容器时执行，如果有多个 CMD 指令，最后一个生效

ENTRYPOINT　运行容器时执行，如果有多个 CMD 指令，最后一个生效。可单独使用，也可与 CMD 配合使用

USER　　为 RUN、CMD、ENTRYPOINT 执行指令指定运行用户

EXPOSE　声明容器运行的服务端口

WORKDIR　　为 RUN、CMD、ENTRYPOINT、COPY 和 ADD 设置工作目录

VOLUME 指定挂载点，使容器中的一个目录具有持久化存储数据的功能

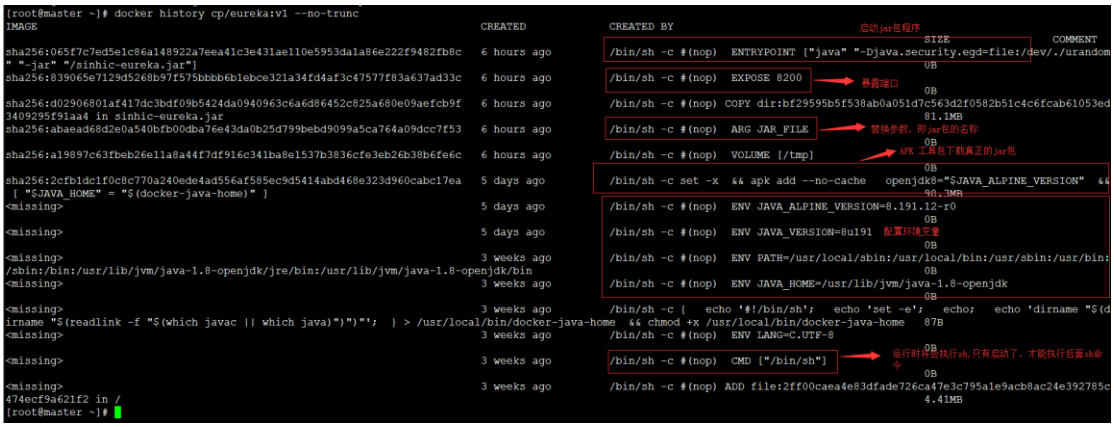
ENV 设置环境变量

ENTRYPOINT 与 CMD 的区别在于 ENTRYPOINT 可以使用 CMD 作为参数，通常都是用作启动后台服务。

- 理解

（1） 镜像的层级，类似于子类继承父类，最顶层的子类则有所有继承类的属性和方法，多个不同的镜像，可以从不同的层级开始继承，这样就非常有利于共享；

（2） 但是镜像并不是单纯的类和对象，有些依赖于一些环境变量，有些又需要启动一些组件，有些还要传递各种参数，构建自定义镜像时还需要复制各种文件，定义各种参数等，所有的这些都带来的复杂性，这时就会有上述命令来满足这些需求；



2.2 FROM

使用 Dockerfile 构建镜像时最好是将 Dockerfile 放置在一个新建的空目录下。然后将构建镜像所需要的文件添加到该目录中。为了提高构建镜像的效率，你可以在目录下新建一个.dockerignore 文件来指定要忽略的文件和目录。.dockerignore 文件的排除模式语法和 Git 的.gitignore 文件相似。

- 语法

FROM <image>

FROM <image>:<tag>

FROM <image>:<digest>

如：

FROM java:8

FROM openjdk:8-jdk-alpine

2.2.1 查看镜像层级

docker history cp/eureka:v1

这里的 cp/eureka:v1 必须是本地已下载或构建好的镜像

```
docker history cp/eureka:v1
    CREATED             CREATED BY                                      SIZE            COMMENT
    5 hours ago         /bin/sh -c #(nop)  ENTRYPOINT ["java" "-Djav…   0B
    5 hours ago         /bin/sh -c #(nop)  EXPOSE 8200                  0B
    5 hours ago         /bin/sh -c #(nop) COPY dir:bf29595b5f538ab0a…   81.1MB
    5 hours ago         /bin/sh -c #(nop)  ARG JAR_FILE                 0B
    5 hours ago         /bin/sh -c #(nop)  VOLUME [/tmp]                0B
    5 days ago          /bin/sh -c set -x  && apk add --no-cache   o…   98.3MB
    5 days ago          /bin/sh -c #(nop)  ENV JAVA_ALPINE_VERSION=8…   0B
    5 days ago          /bin/sh -c #(nop)  ENV JAVA_VERSION=8u191       0B
    3 weeks ago         /bin/sh -c #(nop)  ENV PATH=/usr/local/sbin:…   0B
    3 weeks ago         /bin/sh -c #(nop)  ENV JAVA_HOME=/usr/lib/jv…   0B
    3 weeks ago         /bin/sh -c {   echo '#!/bin/sh';   echo 'set…   87B
    3 weeks ago         /bin/sh -c #(nop)  ENV LANG=C.UTF-8             0B
    3 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
    3 weeks ago         /bin/sh -c #(nop) ADD file:2ff00caea4e83dfad…   4.41MB
```

如查看完整信息，则加参数 no-trunc

docker history cp/eureka:v1 --no-trunc

```
[root@master ~]# docker history cp/eureka:v1 --no-trunc
IMAGE                                                                     CREATED         CREATED BY                                                                    SIZE        COMMENT
sha256:065f7c7ed5e1c86a148922a7eea41c3e431ae110e5953da1a86e222f9482fb8c   6 hours ago     /bin/sh -c #(nop)  ENTRYPOINT ["java" "-Djava.security.egd=file:/dev/./urandom
" "-jar" "/sinhic-eureka.jar"]                                                                                                                                          0B
sha256:839065e7129d5268b97f575bbbb6b1ebce321a34fd4af3c47577f83a637ad33c   6 hours ago     /bin/sh -c #(nop)  EXPOSE 8200                                                0B
sha256:d02906801af417dc3bdf09b5424da0940963c6a6d86452c825a680e09aefcb9f   6 hours ago     /bin/sh -c #(nop) COPY dir:bf29595b5f538ab0a051d7c563d2f0582b51c4c6fcab61053ed
3409295f91aa4 in sinhic-eureka.jar                                                                                                                                      81.1MB
sha256:abaead68d2e0a540bfb00dba76e43da0b25d799bebd9099a5ca764a09dcc7f53   6 hours ago     /bin/sh -c #(nop)  ARG JAR_FILE                                               0B
sha256:a19897c63fbeb26e11a8a44f7df916c341ba8e1537b3836cfe3eb26b38b6fe6c   6 hours ago     /bin/sh -c #(nop)  VOLUME [/tmp]                                              0B
sha256:2cfb1dc1f0c8c770a240ede4ad556af585ec9d5414abd468e323d960cabc17ea   5 days ago      /bin/sh -c set -x  && apk add --no-cache   openjdk8="$JAVA_ALPINE_VERSION"  &&
 [ "$JAVA_HOME" = "$(docker-java-home)" ]                                                                                                                               98.3MB
<missing>                                                                 5 days ago      /bin/sh -c #(nop)  ENV JAVA_ALPINE_VERSION=8.191.12-r0                        0B
<missing>                                                                 5 days ago      /bin/sh -c #(nop)  ENV JAVA_VERSION=8u191                                     0B
<missing>                                                                 3 weeks ago     /bin/sh -c #(nop)  ENV PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin:/usr/lib/jvm/java-1.8-openjdk/jre/bin:/usr/lib/jvm/java-1.8-openjdk/bin                                                                                      0B
<missing>                                                                 3 weeks ago     /bin/sh -c #(nop)  ENV JAVA_HOME=/usr/lib/jvm/java-1.8-openjdk                0B
<missing>                                                                 3 weeks ago     /bin/sh -c {   echo '#!/bin/sh';   echo 'set -e';   echo;   echo 'dirname "$(d
irname "$(readlink -f "$(which javac || which java)")")"';  } > /usr/local/bin/docker-java-home  && chmod +x /usr/local/bin/docker-java-home   87B
<missing>                                                                 3 weeks ago     /bin/sh -c #(nop)  ENV LANG=C.UTF-8                                           0B
<missing>                                                                 3 weeks ago     /bin/sh -c #(nop)  CMD ["/bin/sh"]                                            0B
<missing>                                                                 3 weeks ago     /bin/sh -c #(nop) ADD file:2ff00caea4e83dfade726ca47e3c795a1e9acb8ac24e392785c
474ecf9a621f2 in /                                                                                                                                                      4.41MB
[root@master ~]#
```

（图中标注：ENTRYPOINT —— 启动jar包程序；EXPOSE 8200 —— 暴露端口；ARG JAR_FILE —— 替换参数，即jar包的名称；apk add —— APK 工具包下载真正的jar包；ENV 各行 —— 配置环境变量；CMD ["/bin/sh"] —— 运行时将会执行sh，只有启动了，才能执行后面sh命令）

- 理解

镜像的层级，类似于子类继承父类，最顶层的子类则有所有继承类的属性和方法，多个不同的镜像，可以从不同的层级开始继承，这样就非常有利于共享；

2.3 RUN、CMD、ENTRYPOINT

2.3.1 CMD(启动容器时,即 docker run 时运行)

设置容器的启动命令，Dockerfile 中只能有一条 CMD 命令，如果写了多条则最后一条生效

- 样例：

CMD ["sh", "-c", "echo $HOME"

CMD ["echo", "$HOME"]

- 示例如下：

```
[root@localhost df]# cat Dockerfile
FROM busybox
CMD ["/bin/echo", "this is a echo test"]
[root@localhost df]# docker build -t test .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM busybox
---> 6ad733544a63
Step 2/2 : CMD /bin/echo this is a echo test
---> Running in fa8af9fca520
---> c653707895ae
Removing intermediate container fa8af9fca520
Successfully built c653707895ae
Successfully tagged test:latest
[root@localhost df]# docker run test
this is a echo test
[root@localhost df]# docker run test /bin/echo hello
Hello
```

- 理解

如果想为容器设置默认的启动命令，可使用 CMD 指令。用户可在 docker run 命令行中替换此默认命令，但不支持参数传递，如果需要传参数则要用 ENTRYPOINT

2.3.2 ENTRYPOINT

作用是设置容器入口程序，类似于 CMD 指令，但其不会被 docker run 的命令行参数指定的指令所覆盖，而且这些命令行参数会被当作参数送给 ENTRYPOINT 指令指定的程序；但是，如果运行 docker run 时使用了 --entrypoint 选项，此选项的参数可当作要运行的程序覆盖 ENTRYPOINT 指令指定的程序；容器启动（运行）时才运行。

入口程序是容器启动时执行的程序，docker run 中最后的命令将作为参数传递给入口程序

- 语法

1. ENTRYPOINT ["executable", "param1", "param2"]
2. ENTRYPOINT command param1 param2

如果从上到下看到这里的话，那么你应该对这两种语法很熟悉啦。
第二种就是写 shell
第一种就是可执行文件加参数

- 实际演练

(1) 直接安装一个小软件 stress（一款对硬件的压力测试工具）

```
yum install -y epel-release
sudo yum install -y stress
```

查看效果：

运行 stress 会提示，缺少参数

```
[root@master ~]# stress
`stress' imposes certain types of compute stress on your system

Usage: stress [OPTION [ARG]] ...
 -?, --help         show this help statement
     --version      show version statement
 -v, --verbose      be verbose
 -q, --quiet        be quiet
 -n, --dry-run      show what would have been done
 -t, --timeout N    timeout after N seconds
     --backoff N    wait factor of N microseconds before work starts
 -c, --cpu N        spawn N workers spinning on sqrt()
 -i, --io N         spawn N workers spinning on sync()
 -m, --vm N         spawn N workers spinning on malloc()/free()
     --vm-bytes B   malloc B bytes per vm worker (default is 256MB)
     --vm-stride B  touch a byte every B bytes (default is 4096)
     --vm-hang N    sleep N secs before free (default none, 0 is inf)
     --vm-keep      redirty memory instead of freeing and reallocating
 -d, --hdd N        spawn N workers spinning on write()/unlink()
     --hdd-bytes B  write B bytes per hdd worker (default is 1GB)

Example: stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s
```

运行 stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s

```
[root@master ~]# stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s
stress: info: [27734] dispatching hogs: 8 cpu, 4 io, 2 vm, 0 hdd
stress: info: [27734] successful run completed in 10s
[root@master ~]#
```

反馈

(2) 使用镜像

Cd /opt

Mkdir –p dockerfiles/stress

Cd /opt/dockerfiles/stress

Touch Dockerfile

Vi Dockerfile

FROM ubuntu

RUN apt-get update && apt-get install -y stress

ENTRYPOINT ["/usr/bin/stress"]

CMD []

保存退出

docker build -t cp/stress:v1

docker images

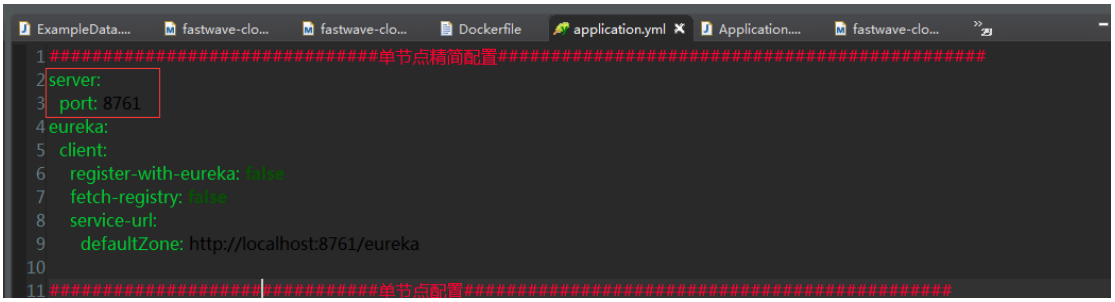
docker run -it cp/stress --vm 1 --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s

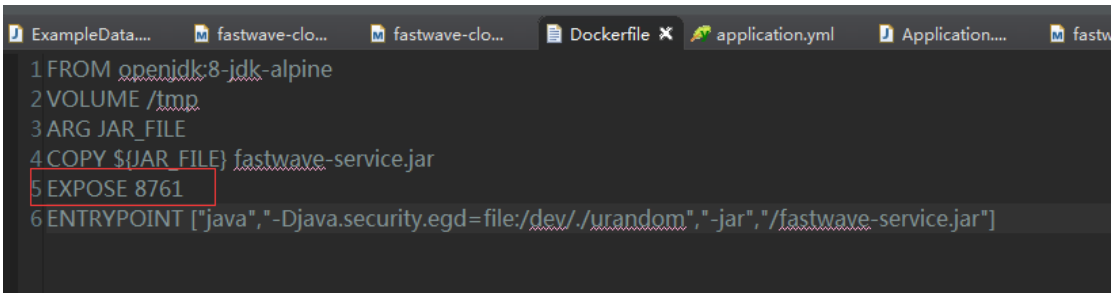
可以看到效果同上面一样，说明橙色部分参数起作

- 理解：

ENTRYPOINT 指定了容器启动时的入口可执行文件路径，并且只有一条生效，在 run 这个容器时，会把镜像后的所有参数传递给这个入口可执行程序，从而把 run 容器时的参数传递给容器内部的程序（如程序为一个 jar 包，则相当于 java –jar xxxx, xxx 表示参数）

2.3.3 ENTRYPOINT 实例(参数在镜像后)





在 shell 中：

-- cmd 中运行 OK

java -jar fastwave-cloud-eureka-4.0-SNAPSHOT.jar --server.port=8762

在 jenkins 的脚本中，则可以写如下代码，可以重新指定端口：

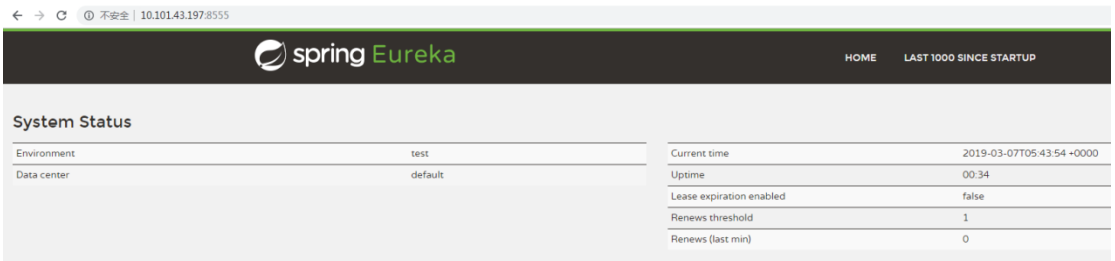
docker run -d --expose 8555 -p 8555:8555

-t 10.101.43.196:5000/fastwave/fastwave-cloud-eureka:latest --server.port=8555

下相同：

docker run -d --expose 8555 -p 8555:8555 -t 10.101.43.196:5000/fastwave/fastwave-cloud-eureka:latest --server.port=8555

效果：



2.3.4 ENTRYPOINT 实例(参数用多个-e 来表示)

- 如：`ENTRYPOINT ["/entrypoint.sh"]`
- 传参：

docker run --name mysqlserver -v $PWD/conf:/etc/mysql/conf.d -v $PWD/logs:/logs -v $PWD/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 -d -i -p 3306:3306 mysql:latest

2.3.5 RUN

RUN 用于指定 docker build 过程中要运行的命令，即是创建 Docker 镜像（image）的步骤。RUN 指令会生成容器，在容器中执行脚本，容器使用当前镜像，脚本指令完成后，Docker Daemon 会将该容器提交为一个中间镜像，供后面的指令使用

- 格式：

RUN <command>
RUN ["executable", "param1", "param2"]
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME
RUN ["/bin/bash", "-c", "echo hello"]

- 样例：

RUN yum install iproute nginx && yum clean all

- 案例：

RUN 的作用就是在构建镜像时，有时候需要在基础镜像上安装一些软件，比如 Nginx。例如：

FROM centos
MAINTAINER allocator
RUN yum install -y nginx
RUN echo ‘hello world’ > /usr/share/nginx/html/index.html
EXPOSE 80
ENTRYPOINT [“/usr/sbin/nginx”]

上面的两个 RUN 在构建镜像时，分别安装了 Nginx 和向 index.html 写入内容。

- 理解：

Run 可以执行各种命令，如为镜像添加一些组件等，每 run 一次，将会多一层，最顶层将有所有完整的组件集合。

2.4 ADD、COPY

2.4.1 ADD

一个复制命令，把文件复制到镜象中。
如果把虚拟机与容器想象成两台 linux 服务器的话，那么这个命令就类似于 scp，只是 scp 需要加用户名和密码的权限验证，而 ADD 不用。

语法如下：
1. ADD <src>... <dest>
2. ADD ["<src>",... "<dest>"]

示例：
ADD test relativeDir/
ADD test /relativeDir
ADD http://example.com/foobar /

```
FROM java:8
VOLUME /tmp
ADD fastwave-gateway-zuul-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

2.4.2 COPY

语法如下：
1. COPY <src>... <dest>
2. COPY ["<src>",... "<dest>"]
与 ADD 的区别
COPY 的<src>只能是本地文件，其他用法一致

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} fastwave-eureka.jar
EXPOSE 8200
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/fastwave-eureka.jar"]
```

理解:
Build 一个镜像时使用，可以从 Dockerfile 所在的目录（如一些 jar 包）复制到镜像中云，COPY 与 ADD 都可以实现，只是 ADD 功能更强大，可以从网络上下载文件，如果设置了 WORKDIR，

则 RUN、CMD、COPY、ADD 会把文件复制到该目录下，但当没有指定呢？文件复制到哪了？这里要看当前工作目录，即 WORKDIR 指令指定目录，还有-v 命令指定的目录，如果都没有指定则是根目录，即“/”。

2.5 VOLUME

只读层及在顶部的读写层的组合被称为 Union File System（联合文件系统），为了能够保存（持久化）数据以及共享容器间的数据，Docker 提出了 Volume 的概念。简单来说，Volume 就是目录或者文件，它可以绕过默认的联合文件系统，而以正常的文件或者目录的形式存在于宿主机上

可实现挂载功能，可以将内地文件夹或者其他容器种得文件夹挂在到这个容器种

语法为：

```
VOLUME ["/data"]
```

说明：

["/data"]可以是一个 JsonArray ，也可以是多个值。所以如下几种写法都是正确的

VOLUME ["/var/log/"]
VOLUME /var/log
VOLUME /var/log /var/db

http://www.cnblogs.com/51kata/p/5266626.html

器是基于镜像创建的，最后的容器文件系统包括镜像的只读层+可写层，容器中的进程操作的数据持久化都是保存在容器的可写层上。一旦容器删除后，这些数据就没了，除非我们人工备份下来（或者基于容器创建新的镜像）。能否可以让容器进程持久化的数据保存在主机上呢？这样即使容器删除了，数据还在

一、通过 docker run 命令

1、运行命令：docker run --name test -it -v /home/xqh/myimage:/data ubuntu /bin/bash

其中的 -v 标记 在容器中设置了一个挂载点 /data（就是容器中的一个目录），并将主机上的 /home/xqh/myimage 目录中的内容关联到 /data 下

映射到主机上了

1、不指定

docker run --name test1 -it -v /data ubuntu /bin/bash

```
xqh@ubuntu:~/myimage$ docker inspect test1
[
{
    "Id": "1fd6c2c4bc545163d8c5c5b02d60052ea41900a781a82c20a8f02059cb82c30c",
..............................
    "Mounts": [
        {
            "Name": "0ab0aaf0d6ef391cb68b72bd8c43216a8f8ae9205f0ae941ef16ebe32dc9fc01",
            "Source": "/var/lib/docker/volumes/0ab0aaf0d6ef391cb68b72bd8c43216a8f8ae9205f0ae941ef16ebe32dc9fc01/_data",
            "Destination": "/data",
            "Driver": "local",
            "Mode": "",
            "RW": true
        }
    ],
............................
```

主机目录是自动创建

二、通过 dockerfile 创建挂载点

```
#test

FROM ubuntu

MAINTAINER hello1

VOLUME ["/data1","/data2"]
```

我们通过docker inspect 查看通过该dockerfile创建的镜像生成的容器，可以看到如下信息

```
    "Mounts": [
        {
            "Name": "d411f6b8f17f4418629d4e5a1ab69679dee369b39e13bb68bed77aa4a0d12d21",
            "Source": "/var/lib/docker/volumes/d411f6b8f17f4418629d4e5a1ab69679dee369b39e13bb68bed77aa4a0d12d21/_data",
            "Destination": "/data1",
            "Driver": "local",
            "Mode": "",
            "RW": true
        },
        {
            "Name": "6d3badcf47c4ac5955deda6f6ae56f4aaf1037a871275f46220c14ebd762fc36",
            "Source": "/var/lib/docker/volumes/6d3badcf47c4ac5955deda6f6ae56f4aaf1037a871275f46220c14ebd762fc36/_data",
            "Destination": "/data2",
            "Driver": "local",
            "Mode": "",
            "RW": true
        }
    ],
```

也是自动创建的

三、容器共享卷（挂载点）

以是来源不同镜像，如：

docker run --name test2 -it --volumes-from test1 ubuntu /bin/bash

也可以是同一镜像，如：

docker run --name test3 -it --volumes-from test1 myimage /bin/bash

上面的三个容器 test1 , test2 , test3 均有 /data1 和 /data2 两个目录，且目录中内容是共享的，任何一个容器修改了内容，别的容器都能获取到。

在几个容器层面共享文件夹

- 理解：

Volume 不仅仅是为了持久化，因为容器最上层有读写层，可以用于数据的持久化，但这不利于各容器之间相互访问，也不能在宿主机中访问，相当于这些都被容器“封起来了”，如果容器删除了，容器内部的数据也都删除了，而一般情况下不推荐对容器内部进行什么操作，所以对于一些配置文件、数据文件、共享目录都无法处理。而 Volume 提供了各种情况下共享数据的方式：

（1） run 容器时的-v 参数，这时可以映射到宿主机上的具体目录，如：

```
docker run --name mysqlserver -v $PWD/conf:/etc/mysql/conf.d -v $PWD/logs:/logs -v $PWD/data:/var/lib/mysq -e
MYSQL_ROOT_PASSWORD=123456 -d -i -p 3306:3306  mysql:5.7
```

（2） run 指定-v 但没有指定宿主机目录，这时会随机的自动生成一个目录

```
docker run --name test1 -it -v /data ubuntu /bin/bash
```

（3） run 指定--volumes-from test1，可以在容器之间共享，但宿主机不能修改

```
docker run --name test2 -it --volumes-from test1  ubuntu  /bin/bash
docker run --name test3 -it --volumes-from test1  myimage  /bin/bash
```

（4） 仅指定一个 volum，可在不同容器中使用，但宿主机不能修改

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
```

上述设置后的数据，在清除容器后仍存在。

(5)Dockerfile 中使用了 VOLUME /tmp 指令后,容器发生写数据时,都会写到该目录下吗?

2.6 ENV、ARG

什么时候用 ARG，什么时候用 ENV?
如果想保存为环境变量，就用 ENV；如果只想在 Dockerfile 中临时使用，就用 ARG
ENV 指令是在 dockerfile 里面设置环境变量，不能在编译时或运行时传递

2.6.1 ENV

格式： ENV <key> <value>或 ENV <key>=<value>

通过 ENV 定义的环境变量，可以被后面的所有指令中使用，如上面的例子

2、但是不能被 CMD 指令使用，也不能被 docker run 的命令参数引用。这个需要注意

3、通过 ENV 定义的环境变量，会永久的保存到该镜像创建的任何容器中。这样除了不能在上面说的启动命令中使用外，可以在后续容器的操作中使用。

4、可以在 docker run 命令中通过 -e 标记来传递环境变量，这样容器运行时就可以使用该变量。如：

```
docker run -i -t -e "TEST=hello" ubuntu /bin/bash
```

- 案例

```
#test
FROM ubuntu
MAINTAINER hello
ENV MYDIR /mydir
RUN mkdir $MYDIR
RUN echo hello world > $MYDIR/test.txt
```

假设用上面的 dockerfile 构建了一个叫 myimage 的镜像。
运行 docker run -i -t myimage /bin/bash
我们发现新建的容器中有了 /mydir 目录，并有了/mydir/test.txt 文件，文件内容为 hello world

下方为查看一个镜像的层级，也有很多环境变量的定义

```
[root@master ~]# docker history cp/eureka:v1
IMAGE               CREATED             CREATED BY                                      SIZE
065f7c7ed5e1        5 hours ago         /bin/sh -c #(nop)  ENTRYPOINT ["java" "-Djav…    0B
839065e7129d        5 hours ago         /bin/sh -c #(nop)  EXPOSE 8200                   0B
d02906801af4        5 hours ago         /bin/sh -c #(nop) COPY dir:bf29595b5f538ab0a…    81.1MB
abaead68d2e0        5 hours ago         /bin/sh -c #(nop)  ARG JAR_FILE                  0B
a19897c63fbe        5 hours ago         /bin/sh -c #(nop)  VOLUME [/tmp]                 0B
2cfb1dc1f0c8        5 days ago          /bin/sh -c set -x  && apk add --no-cache    o…    98.3MB
<missing>           5 days ago          /bin/sh -c #(nop)  ENV JAVA_ALPINE_VERSION=8…    0B
<missing>           5 days ago          /bin/sh -c #(nop)  ENV JAVA_VERSION=8u191        0B
<missing>           3 weeks ago         /bin/sh -c #(nop)  ENV PATH=/usr/local/sbin:…    0B
<missing>           3 weeks ago         /bin/sh -c #(nop)  ENV JAVA_HOME=/usr/lib/jv…    0B
<missing>           3 weeks ago         /bin/sh -c {    echo '#!/bin/sh';    echo 'set…    87B
<missing>           3 weeks ago         /bin/sh -c #(nop)  ENV LANG=C.UTF-8              0B
<missing>           3 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/sh"]               0B
<missing>           3 weeks ago         /bin/sh -c #(nop) ADD file:2ff00caea4e83dfad…    4.41MB
[root@master ~]#
```

2.6.2 ARG

格式： ARG <参数名>[=<默认值>]

如：ARG CODE_VERSION=latest

- 案例

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} fastwave-eureka.jar
EXPOSE 8200
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/fastwave-eureka.jar"]
```

在 build 时，需要指定参数才能正确编译：

```
docker build --build-arg JAR_FILE=fastwave-eureka-0.0.1-SNAPSHOT.jar -t cp/eureka:v2 .
```

- 理解：

都可以理解为定义变量，然后就可以在各种场景下使用了，但 ARG 定义的变量在局部使用，且可以通过命令行传递进来，但 ENV 定义的范围更大一些，可以在下一层中使用。

2.7 EXPOSE

格式为 EXPOSE port [port2,port3,...]，例如 EXPOSE 80 这条指令告诉 Docker 服务器暴露 80 端口，供容器外部连接使用。

在启动容器的使用使用-P，Docker 会自动分配一个端口和转发指定的端口，使用-p 可以具体指定使用哪个本地的端口来映射对外开放的端口。

EXPOSE 只设置暴露端口并不导出端口，Docker Daemon 会把镜像中所有暴露端口导出，并为每个暴露端口分配一个随机的主机端口，只有启动容器时使用-P/-p 才导出端口，这个时

候才能通过外部访问容器提供的服务。

- 案例：

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} fastwave-service-admin.jar
EXPOSE 8203
ENTRYPOINT
["java","-Djava.security.egd=file:/dev/./urandom","-jar","/fastwave-service-admin.jar"]
```

- 理解：

暴露一个或多个端口，这些端口要与运行的 jar 包中的启动的监听端口一致，这里暴露后，只能说容器中暴露了端口（如果 dockerfile 中没有 expose 语句，则容器将不会有任何监听端口可以映射？），并不能在外部访问，因为外部访问的是宿主机上的端口。在 run 一个镜像时 Docker Daemon 会映射端口为宿主机端口，如果运行时没有用-P/-p 指定宿主机端口，则会随机映射一个未使用的宿主机端口。

- 当希望在 run 中指定暴露的端口时，可以使用如下命令：

```
docker run -d --expose 8555 -p 8555:8555
-t 10.101.43.196:5000/fastwave/fastwave-cloud-eureka:latest --server.port=8555
```

设置之后，就会在容器内部暴露 8555 端口了，使用运行时标志--expose 是附加的，因此会在 Dockerfile 的 EXPOSE 指令定义的端口之外暴露添加的端口

2.8 WORKDIR

WORKDIR 指令为 Dockerfile 中的任何 RUN，CMD，ENTRYPOINT，COPY 和 ADD 指令设置工作目录（或称当前目录）。(也就是说以后各层的当前目录就被改为 WORKDIR 指定的目录)如果 WORKDIR 对应的目录不存在，将会自动被创建

- dockerfile 案例 1：

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

则最终路径为/a/b/c

- run 中案例 2：

-w 参数覆盖掉 WORKDIR 指令的设置，如：

执行 docker run -w / myimage

上面的-w 参数将容器的工作目录设置成了根目录

查看当前的工作目录，命令：

docker exec -it a6c5180a08c6 pwd

- 理解

使用 workdir 相当于，相当于 cd 命令，后面的 RUN、CMD，ENTRYPOINT，COPY 和 ADD 指令会按这个目录执行。在 dockerfile 中可以指定，在 run 容器的时候也可以指定，并且优先级更高，会覆盖 dockerfile 中的设置，如果都没有设置，则默认工作目录为根目录，即“/”。

2.9 LABEL、ONBUILD、USER

2.9.1 LABEL

- 示例

LABEL <key>=<value> <key>=<value> <key>=<value>

一个 LABEL 指定一层，尽量合并为一个指令，同名覆盖。

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
LABEL maintainer="SvenDowideit@home.org.au"
```

docker inspect 查看 images 的 LABEL

```
"Labels": {
   "com.example.vendor": "ACME Incorporated"
   "com.example.label-with-value": "foo",
   "version": "1.0",
   "description": "This text illustrates that label-values can span multiple lines.",
   "multi.label1": "value1",
   "multi.label2": "value2",
   "other": "value3"
}
```

一个镜像可以有多个 label。要指定多个 labels，Docker 推荐尽可能地把多个 labels 合并到一个 LABEL 指令中去。每一个 LABEL 指令会生成一个新的镜像层，如果你使用多个 label，将导致构建出一个低效的镜像。这个示例只生成单个镜像层，下面为推荐写法，这个示例只生成单个镜像层：

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

上面的示例可以重写为：

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

- 理解

Label 基本上是一些元数据信息，如版本号之类的，没有“实际”的作用，主要是一些文本

说明信息，可以在 inspect 中查看到。

2.9.2 ONBUILD

ONBUILD 该指令的作用就是，它后面往往放的一些可执行的命令，但是它进行镜像构建时，ONBUILD 后面的命令不会执行，而是别人把它构建的镜像作为基础镜像拿来再构建镜像时，就会执行 ONBUILD 后面的指令。

（1）父镜像的 Dockerfile 文件内容如下

```
FROM node:0.12.6
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
ONBUILD COPY package.json /usr/src/app/
ONBUILD RUN npm install
ONBUILD COPY . /usr/src/app
CMD [ "npm", "start" ]
```

（2）子镜像的 Dockerfile 文件内容如下

```
FROM node:0.12.6-onbuild
```

- 理解

当有子镜像 FROM 本镜像时，才会执行 ONBUILD 后的语句，避免镜像过大

2.9.3 USER

USER 指令用于指定容器执行程序的用户身份，默认是 root 用户。在 docker run 中可以通过 -u 选项来覆盖 USER 指令的设置。例如：

docker run -it -u mysql newmysqldb /bin/bash

docker 容器中的 root 用户密码是随机分配的

- 理解

指定容器内部执行指令的用户，默认为 root 用户，一般不需要修改。

3 镜像层级

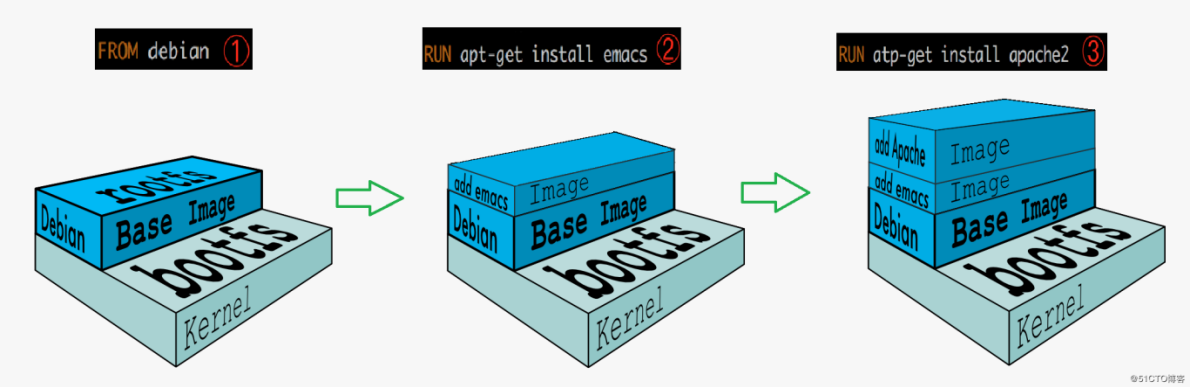
3.1 Dockerfile 案例

scratch 是一非常特别的镜像，这个镜像很有意思的是他是一个空镜像。
也就是说大小几乎是 0,但是不能单独跑起来，只能做基础镜像，然后把 main 程序放上去。
说白就是是一个线程了，这样有个非常好的优势，镜像小，没有依赖。
开始还以为是 alpine 比较小呢，看了这个是更小了。
实际上，Docker Hub 中 99% 的镜像都是通过在 base 镜像中安装和配置需要的软件构建出来的。比如我们现在构建一个新的镜像，Dockerfile 如下：

新镜像不再是从 scratch 开始，而是直接在 Debian base 镜像上构建

```
1. # Version: 0.0.1
2. FROM debian                1.新镜像不再是从 scratch 开始，而是直接在 Debian base 镜像上构建。
3. MAINTAINER wzlinux
4. RUN apt-get update && apt-get install -y emacs        2.安装 emacs 编辑器。
5. RUN apt-get install -y apache2              3.安装 apache2。
6. CMD ["/bin/bash"]              4.容器启动时运行 bash。
```

构建过程如下图所示：

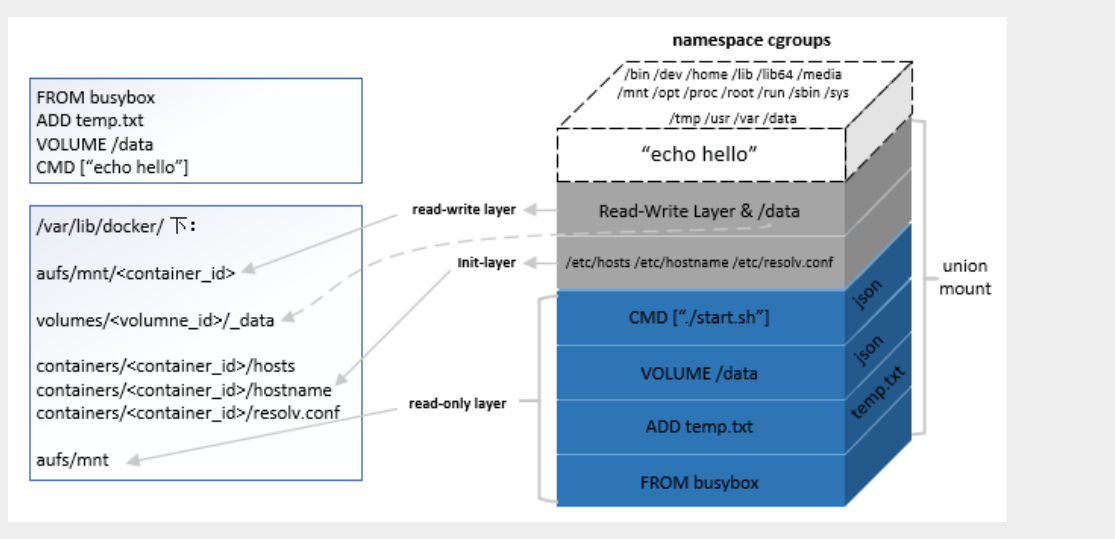


可以看到，新镜像是从 base 镜像一层一层叠加生成的。每安装一个软件，就在现有镜像的基础上增加一层

3.2 镜像层级示意图

https://www.cnblogs.com/sparkdev/p/9092082.html

综合考虑镜像的层级结构，以及 volume、init-layer、可读写层这些概念，一个完整的、在运行的容器的所有文件系统结构可以用下图来描述：



3.3 Mysql Dockerfile 案例

https://github.com/mysql/mysql-docker/blob/mysql-server/5.7/Dockerfile

```
FROM oraclelinux:7-slim
ARG MYSQL_SERVER_PACKAGE=mysql-community-server-minimal-5.7.25
ARG MYSQL_SHELL_PACKAGE=mysql-shell-8.0.15

# Install server
RUN yum install -y https://repo.mysql.com/mysql-community-minimal-release-el7.rpm \
        https://repo.mysql.com/mysql-community-release-el7.rpm \
    && yum-config-manager --enable mysql57-server-minimal \
    && yum install -y \
        $MYSQL_SERVER_PACKAGE \
        $MYSQL_SHELL_PACKAGE \
        libpwquality \
    && yum clean all \
    && mkdir /docker-entrypoint-initdb.d

VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh
COPY healthcheck.sh /healthcheck.sh
ENTRYPOINT ["/entrypoint.sh"]
HEALTHCHECK CMD /healthcheck.sh
EXPOSE 3306 33060
CMD ["mysqld"]
```

```
docker run --name mysqlserver -v $PWD/conf:/etc/mysql/conf.d -v $PWD/logs:/logs -v $PWD/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 -d -i -p 3306:3306
```

mysql:latest

3.4 容器如镜像的关系

可写的容器层
当容器启动时，一个新的可写层被加载到镜像的顶部。这一层通常被称作“容器层”，“容器层”之下的都叫“镜像层”