Docker(三)：Dockerfile 命令详解

原创： 纯洁的微笑 纯洁的微笑 2018-03-16

上一篇文章Docker(二)：Dockerfile 使用介绍介绍了 Dockerfile 的使用，这篇文章我们来继续了解 Dockerfile ，学习 Dockerfile 各种命令的使用。

Dockerfile 指令详解

1 FROM 指定基础镜像

FROM 指令用于指定其后构建新镜像所使用的基础镜像。FROM 指令必是 Dockerfile 文件中的首条命令，启动构建流程后，Docker 将会基于该镜像构建新镜像，FROM 后的命令也会基于这个基础镜像。

FROM语法格式为：

```
1. FROM <image>
```

或

```
1. FROM <image>:<tag>
```

或

```
1. FROM <image>:<digest>
```

通过 FROM 指定的镜像，可以是任何有效的基础镜像。FROM 有以下限制：

- FROM 必须 是 Dockerfile 中第一条非注释命令
- 在一个 Dockerfile 文件中创建多个镜像时，FROM 可以多次出现。只需在每个新命令 FROM 之前，记录提交上次的镜像 ID。
- tag 或 digest 是可选的，如果不使用这两个值时，会使用 latest 版本的基础镜像

2 RUN 执行命令

在镜像的构建过程中执行特定的命令，并生成一个中间镜像。格式:

```
1. #shell格式
2. RUN <command>
3. #exec格式
4. RUN ["executable", "param1", "param2"]
```

- RUN 命令将在当前 image 中执行任意合法命令并提交执行结果。命令执行提交后，就会自动执行 Dockerfile 中的下一个指令。

- 层级 RUN 指令和生成提交是符合 Docker 核心理念的做法。它允许像版本控制那样，在任意一个点，对 image 镜像进行定制化构建。
- RUN 指令创建的中间镜像会被缓存，并会在下次构建中使用。如果不想使用这些缓存镜像，可以在构建时指定 `--no-cache` 参数，如： `docker build --no-cache`。

3 COPY 复制文件

格式：

```
1. COPY <源路径>... <目标路径>
2. COPY ["<源路径1>",... "<目标路径>"]
```

和 RUN 指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。COPY 指令将从构建上下文目录中 <源路径> 的文件/目录复制到新的一层的镜像内的 `<目标路径>`位置。比如：

```
1. COPY package.json /usr/src/app/
```

`<源路径>`可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的 filepath.Match 规则，如：

```
1. COPY hom* /mydir/
2. COPY hom?.txt /mydir/
```

`<目标路径>`可以是容器内的绝对路径，也可以是相对于工作目录的相对路径（工作目录可以用 WORKDIR 指令来指定）。目标路径不需要事先创建，如果目录不存在会在复制文件前先行创建缺失目录。

此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限、文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git 进行管理的时候。

4 ADD 更高级的复制文件

ADD 指令和 COPY 的格式和性质基本一致。但是在 COPY 基础上增加了一些功能。比如 `<源路径>`可以是一个 URL，这种情况下，Docker 引擎会试图去下载这个链接的文件放到 `<目标路径>`去。

在构建镜像时，复制上下文中的文件到镜像内，格式：

```
1. ADD <源路径>... <目标路径>
2. ADD ["<源路径>",... "<目标路径>"]
```

注意

如果 docker 发现文件内容被改变，则接下来的指令都不会再使用缓存。关于复制文件时需要处理的/，基本跟正常的 copy 一致

5 ENV 设置环境变量

格式有两种：

```
1. ENV <key> <value>
2. ENV <key1>=<value1> <key2>=<value2>...
```

这个指令很简单，就是设置环境变量而已，无论是后面的其它指令，如 RUN，还是运行时的应用，都可以直接使用这里定义的环境变量。

```
1. ENV VERSION=1.0 DEBUG=on \
2.  NAME="Happy Feet"
```

这个例子中演示了如何换行，以及对含有空格的值用双引号括起来的办法，这和 Shell 下的行为是一致的。

6 EXPOSE

为构建的镜像设置监听端口，使容器在运行时监听。格式：

```
1. EXPOSE <port> [<port>...]
```

EXPOSE 指令并不会让容器监听 host 的端口，如果需要，需要在 docker run 时使用 `-p`、 `-P` 参数来发布容器端口到 host 的某个端口上。

7 VOLUME 定义匿名卷

VOLUME用于创建挂载点，即向基于所构建镜像创始的容器添加卷：

```
1. VOLUME ["/data"]
```

一个卷可以存在于一个或多个容器的指定目录，该目录可以绕过联合文件系统，并具有以下功能：

- 卷可以容器间共享和重用
- 容器并不一定要和其它容器共享卷
- 修改卷后会立即生效
- 对卷的修改不会对镜像产生影响
- 卷会一直存在，直到没有任何容器在使用它

VOLUME 让我们可以将源代码、数据或其它内容添加到镜像中，而又不并提交到镜像中，并使我们可以多个容器间共享这些内容。

8 WORKDIR 指定工作目录

WORKDIR用于在容器内设置一个工作目录：

```
1. WORKDIR  /path/to/workdir
```

通过WORKDIR设置工作目录后，Dockerfile 中其后的命令 RUN、CMD、ENTRYPOINT、ADD、COPY 等命令都会在该目录下执行。 如，使用WORKDIR设置工作目录：

```
1. WORKDIR  /a
2. WORKDIR b
3. WORKDIR c
4. RUN pwd
```

在以上示例中，pwd 最终将会在 `/a/b/c` 目录中执行。在使用 docker run 运行容器时，可以通过 `-w`参数覆盖构建时所设置的工作目录。

9 USER 指定当前用户

USER 用于指定运行镜像所使用的用户：

```
1. USER daemon
```

使用USER指定用户时，可以使用用户名、UID 或 GID，或是两者的组合。以下都是合法的指定试：

```
1. USER user
2. USER user:group
3. USER uid
4. USER uid:gid
5. USER user:gid
6. USER uid:group
```

使用USER指定用户后，Dockerfile 中其后的命令 RUN、CMD、ENTRYPOINT 都将使用该用户。镜像构建完成后，通过 docker run 运行容器时，可以通过 `-u` 参数来覆盖所指定的用户。

10 CMD

CMD用于指定在容器启动时所要执行的命令。CMD 有以下三种格式：

```
1. CMD  ["executable","param1","param2"]
2. CMD  ["param1","param2"]
3. CMD command param1 param2
```

省略可执行文件的 exec 格式，这种写法使 CMD 中的参数当做 ENTRYPOINT 的默认参数，此时 ENTRYPOINT 也应该是 exec 格式，具体与 ENTRYPOINT 的组合使用，参考 ENTRYPOINT。

注意

与 RUN 指令的区别：RUN 在构建的时候执行，并生成一个新的镜像，CMD 在容器运行的时候执行，在构建时不进行任何操作。

11 ENTRYPOINT

ENTRYPOINT 用于给容器配置一个可执行程序。也就是说，每次使用镜像创建容器时，通过 ENTRYPOINT 指定的程序都会被设置为默认程序。ENTRYPOINT 有以下两种形式：

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

ENTRYPOINT 与 CMD 非常类似，不同的是通过 `docker run`执行的命令不会覆盖 ENTRYPOINT，而 `docker run`命令中指定的任何参数，都会被当做参数再次传递给 ENTRYPOINT。Dockerfile 中只允许有一个 ENTRYPOINT 命令，多指定时会覆盖前面的设置，而只执行最后的 ENTRYPOINT 指令。

`docker run`运行容器时指定的参数都会被传递给 ENTRYPOINT ，且会覆盖 CMD 命令指定的参数。如，执行 `docker run <image> -d`时，-d 参数将被传递给入口点。

也可以通过 `docker run --entrypoint`重写 ENTRYPOINT 入口点。如：可以像下面这样指定一个容器执行程序：

```
1. ENTRYPOINT ["/usr/bin/nginx"]
```

完整构建代码：

```
1. # Version: 0.0.3
2. FROM ubuntu:16.04
3. MAINTAINER 何民三 "cn.liuht@gmail.com"
4. RUN apt-get update
5. RUN apt-get install -y nginx
6. RUN echo 'Hello World, 我是个容器' \
7.   > /var/www/html/index.html
8. ENTRYPOINT ["/usr/sbin/nginx"]
9. EXPOSE 80
```

使用docker build构建镜像，并将镜像指定为 itbilu/test：

```
1. docker build -t="itbilu/test" .
```

构建完成后，使用itbilu/test启动一个容器：

```
1. docker run -i -t itbilu/test -g "daemon off;"
```

在运行容器时，我们使用了 `-g "daemon off;"`，这个参数将会被传递给 ENTRYPOINT，最终在容器中执行的命令为 `/usr/sbin/nginx -g "daemon off;"`。

12 LABEL

LABEL用于为镜像添加元数据，元数以键值对的形式指定：

```
1. LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

使用LABEL指定元数据时，一条LABEL指定可以指定一或多条元数据，指定多条元数据时不同元数据之间通过空格分隔。推荐将所有的元数据通过一条LABEL指令指定，以免生成过多的中间镜像。 如，通过LABEL指定一些元数据：

```
1. LABEL version="1.0" description="这是一个Web服务器" by="IT笔录"
```

指定后可以通过docker inspect查看：

```
1. docker inspect itbilu/test
2. "Labels": {
3.   "version": "1.0",
4.   "description": "这是一个Web服务器",
5.   "by": "IT笔录"
6. },
```

13 ARG

ARG用于指定传递给构建运行时的变量：

```
1. ARG <name>[=<default value>]
```

如，通过ARG指定两个变量：

```
1. ARG site
2. ARG build_user=IT笔录
```

以上我们指定了 site 和 build*user 两个变量，其中 buila*user 指定了默认值。在使用 docker build 构建镜像时，可以通过 `--build-arg <varname>=<value>` 参数来指定或重设置这些变量的值。

```
1. docker build --build-arg site=itiblu.com -t itbilu/test .
```

这样我们构建了 itbilu/test 镜像，其中site会被设置为 itbilu.com，由于没有指定 build_user，其值将是默认值 IT 笔录。

14 ONBUILD

ONBUILD用于设置镜像触发器：

```
1. ONBUILD [INSTRUCTION]
```

当所构建的镜像被用做其它镜像的基础镜像，该镜像中的触发器将会被钥触发。 如，当镜像被使用时，可能需要做一些处理：

```
1. [...]
2. ONBUILD ADD . /app/src
3. ONBUILD RUN /usr/local/bin/python-build --dir /app/src
```

```
4. [...]
```

15 STOPSIGNAL

STOPSIGNAL用于设置停止容器所要发送的系统调用信号：

```
1. STOPSIGNAL signal
```

所使用的信号必须是内核系统调用表中的合法的值，如：SIGKILL。

16 SHELL

SHELL用于设置执行命令（shell式）所使用的的默认 shell 类型：

```
1. SHELL ["executable", "parameters"]
```

SHELL在Windows环境下比较有用，Windows 下通常会有 cmd 和 powershell 两种 shell，可能还会有 sh。这时就可以通过 SHELL 来指定所使用的 shell 类型：

```
1. FROM microsoft/windowsservercore
2. # Executed as cmd /S /C echo default
3. RUN echo default
4. # Executed as cmd /S /C powershell -command Write-Host default
5. RUN powershell -command Write-Host default
6. # Executed as powershell -command Write-Host hello
7. SHELL ["powershell", "-command"]
8. RUN Write-Host hello
9. # Executed as cmd /S /C echo hello
10. SHELL ["cmd", "/S"", "/C"]
11. RUN echo hello
```

Dockerfile 使用经验

Dockerfile 示例

构建Nginx运行环境

```
1. # 指定基础镜像
2. FROM sameersbn/ubuntu:14.04.20161014
3. # 维护者信息
4. MAINTAINER sameer@damagehead.com
5. # 设置环境
6. ENV RTMP_VERSION=1.1.10 \
7.   NPS_VERSION=1.11.33.4 \
8.   LIBAV_VERSION=11.8 \
9.   NGINX_VERSION=1.10.1 \
10.   NGINX_USER=www-data \
11.   NGINX_SITECONF_DIR=/etc/nginx/sites-enabled \
12.   NGINX_LOG_DIR=/var/log/nginx \
13.   NGINX_TEMP_DIR=/var/lib/nginx \
14.   NGINX_SETUP_DIR=/var/cache/nginx
```

```
# 设置构建时变量，镜像建立完成后就失效
ARG BUILD_LIBAV=false
ARG WITH_DEBUG=false
ARG WITH_PAGESPEED=true
ARG WITH_RTMP=true
# 复制本地文件到容器目录中
COPY setup/  ${NGINX_SETUP_DIR}/
RUN bash ${NGINX_SETUP_DIR}/install.sh
# 复制本地配置文件到容器目录中
COPY nginx.conf  /etc/nginx/nginx.conf
COPY entrypoint.sh  /sbin/entrypoint.sh
# 运行指令
RUN chmod  755  /sbin/entrypoint.sh
# 允许指定的端口
EXPOSE  80/tcp  443/tcp  1935/tcp
# 指定网站目录挂载点
VOLUME  ["${NGINX_SITECONF_DIR}"]
ENTRYPOINT  ["/sbin/entrypoint.sh"]
CMD  ["/usr/sbin/nginx"]
```

构建tomcat 环境

Dockerfile文件

```
# 指定基于的基础镜像
FROM ubuntu:13.10
# 维护者信息
MAINTAINER zhangjiayang  "zhangjiayang@sczq.com.cn"
# 镜像的指令操作
# 获取APT更新的资源列表
RUN echo  "deb http://archive.ubuntu.com/ubuntu precise main universe"> /etc/apt/sources.list
# 更新软件
RUN apt-get  update
# Install curl
RUN apt-get  -y install curl
# Install JDK 7
RUN cd  /tmp  &&  curl  -L  'http://download.oracle.com/otn-pub/java/jdk/7u65-b17/jdk-7u65-linux-x64.tar.gz'  -H  'Cookie: oraclelicense=accept-securebackup-cookie; gpw_e24=Dockerfile'  |  tar  -xz
RUN mkdir  -p  /usr/lib/jvm
RUN mv  /tmp/jdk1.7.0_65/ /usr/lib/jvm/java-7-oracle/
# Set Oracle JDK 7 as default Java
RUN update-alternatives  --install  /usr/bin/java java  /usr/lib/jvm/java-7-oracle/bin/java  300
RUN update-alternatives  --install  /usr/bin/javac javac  /usr/lib/jvm/java-7-oracle/bin/javac  300
# 设置系统环境
ENV JAVA_HOME  /usr/lib/jvm/java-7-oracle/
```

```
# Install tomcat7
RUN cd /tmp && curl -L 'http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.8/bin/apache-tomcat-7.0.8.tar.gz' | tar -xz
RUN mv /tmp/apache-tomcat-7.0.8/ /opt/tomcat7/
ENV CATALINA_HOME /opt/tomcat7
ENV PATH $PATH:$CATALINA_HOME/bin
# 复件tomcat7.sh到容器中的目录
ADD tomcat7.sh /etc/init.d/tomcat7
RUN chmod 755 /etc/init.d/tomcat7
# Expose ports. 指定暴露的端口
EXPOSE 8080
# Define default command.
ENTRYPOINT service tomcat7 start && tail -f /opt/tomcat7/logs/catalina.out
```

`tomcat7.sh`命令文件

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle/
export TOMCAT_HOME=/opt/tomcat7
case $1 in
start)
  sh $TOMCAT_HOME/bin/startup.sh
;;
stop)
  sh $TOMCAT_HOME/bin/shutdown.sh
;;
restart)
  sh $TOMCAT_HOME/bin/shutdown.sh
  sh $TOMCAT_HOME/bin/startup.sh
;;
esac
exit 0
```

原则与建议

- 容器轻量化。从镜像中产生的容器应该尽量轻量化，能在足够短的时间内停止、销毁、重新生成并替换原来的容器。
- 使用 `.gitignore`。在大部分情况下，Dockerfile 会和构建所需的文件放在同一个目录中，为了提高构建的性能，应该使用 `.gitignore` 来过滤掉不需要的文件和目录。
- 为了减少镜像的大小，减少依赖，仅安装需要的软件包。
- 一个容器只做一件事。解耦复杂的应用，分成多个容器，而不是所有东西都放在一个容器内运行。如一个 Python Web 应用，可能需

要 Server、DB、Cache、MQ、Log 等几个容器。一个更加极端的说法：One process per container。

- 减少镜像的图层。不要多个 Label、ENV 等标签。
- 对续行的参数按照字母表排序，特别是使用 `apt-get install -y` 安装包的时候。
- 使用构建缓存。如果不想使用缓存，可以在构建的时候使用参数 `--no-cache=true` 来强制重新生成中间镜像。