Dockerfile 基础...

这是本专栏的第三部分：镜像篇，共 8 篇。前两篇我为你介绍了 Docker 镜像生命周期的管理，以及镜像的构建和分发方式。本篇，我来为你介绍 Dockerfile 带你理解 Dockerfile 中的重点知识，方便你自己定义构建镜像的行为。下面我们一起进入本篇的学习。

Dockerfile 基本格式

在前面的内容中，我有写过几个 Dockerfile，但是并没有对它做过多介绍。这里，我们来先看看 Dockerfile 它最基本的格式。如下：

复制

```
# 注释
INSTRUCTION arguments
```

看起来很简单对吧，一般情况下以 # 开头的内容是注释，其他内容以指令开头，后面跟着参数所使用的指令。

指令实际不区分大小写，但是**约定使用大写。**

注意：为什么说是"一般情况下以 # 开头的是注释"呢？这是因为目前有两种特殊情况。分别是：

- 以 `# escape=` 格式开头的 Dockerfile
- 以 `# syntax=` 格式开头的 Dockerfile

使用 escape 主要的需求是转义 Windows 镜像的特殊字符；而使用 syntax 的场景目前比较少，主要是使用构建的高级特性，在后续章节中会介绍。

Dockerfile 常用指令介绍

FROM

指定构建镜像所用的基础镜像，通常情况下我们会使用 Docker 官方镜像，可以在 Docker Hub 上找到。

注意：**每个 Dockerfile 都必须有 FROM 指令**，如果没有指定 FROM，则 Docker 在解析 Dockerfile 时会报错。

如果不想使用任何基础镜像，则需要使用 `FROM scratch`，通常构建基础系统镜像，或者独立的纯二进制文件的镜像时会使用这种方式。

RUN

指定构建过程中需要执行的操作。根据我们在前面章节的介绍，也许你已经意识到了，Docker 镜像是层级结构的（比如我们之前对 Docker 镜像做解压操作时，可以看到其配置文件中包含了各层的信息等），每个 RUN 指令都会一个新层中执行，并将其结果提交为一个新的层，并用于后续的 Dockerfile 的操作。

我们常用的 RUN 指令的形式是 shell 形式的，在 Linux 上通常是指 `/bin/sh -c` 的形式。

当然它还支持另外一种 exec 形式的，需要用 `[]` 括起来，例如：

复制

```
RUN ["/usr/bin/echo", "using exec form"]
```

EXPOSE

EXPOSE 指令表示容器运行时要监听的端口，例如 `EXPOSE 6379` 则表示要暴露 6379 端口。

我们来看个实际例子。

在 Docker 官方 Redis 镜像的 Dockerfile 中写了 EXPOSE 6379，我们启动一个容器看看：

复制

```
(MoeLove) ➜  ~ docker run -d --rm redis
aef837559549907e650f9935d694af776b3a855e2ba97b024f972027d9d21d27
(MoeLove) ➜  ~ docker ps -l
CONTAINER ID        IMAGE               COMMAND                  CREATED
aef837559549        redis               "docker-entrypoint.s…"   4 seconds ago
```

可以通过 `docker ps` 看到 PORTS 那一列中有容器所暴露的端口。同时，EXPOSE 支持的完整格式是 `EXPOSE port/protocol`，目前支持 TCP 和 UDP 协议。

当然，暴露端口除了在 Dockerfile 中直接声明以外，还可以在 `docker run` 启动容器时，通过 `-p` 参数进行指定。

Dockerfile 重点知识说明

Dockerfile 还有一些其他的指令，常会被人混淆，下面我来为你详细介绍这几个容易被混淆的指令。

COPY vs ADD

在正式介绍之前，我们使用如下两个 Dockerfile 进行验证：

复制

```
(MoeLove) ➜  build echo 'file in container' > note

# 使用 COPY 指令
(MoeLove) ➜  build cat << EOF > Dockerfile.copy
FROM alpine
COPY note /note
EOF


# 使用 ADD 指令
(MoeLove) ➜  build cat << EOF > Dockerfile.add
FROM alpine
ADD note /note
EOF
```

构建镜像并运行：

复制

```
(MoeLove) ➜  build docker build --no-cache -t build:copy -f Dockerfile.copy .
[+] Building 0.4s (7/7) FINISHED
 => [internal] load build definition from Dockerfile.copy
 => => transferring dockerfile: 100B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/alpine:latest
 => [internal] load build context
 => => transferring context: 107B
 => CACHED [1/2] FROM docker.io/library/alpine
 => [2/2] COPY note /note
 => exporting to image
 => => exporting layers
 => => writing image sha256:994666e664c4e8929fba395cb3c115575991c1447bece09cb6c70a
 => => naming to docker.io/library/build:copy
(MoeLove) ➜  build docker run --rm build:copy cat /note
file in container
```

构建另一个镜像并运行：

复制

```
(MoeLove) ➜  build docker build --no-cache -t build:add -f Dockerfile.add .
[+] Building 0.4s (7/7) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile.add
 => => transferring dockerfile: 132B
 => [internal] load metadata for docker.io/library/alpine:latest
 => [internal] load build context
 => => transferring context: 83B
 => CACHED [1/2] FROM docker.io/library/alpine
 => [2/2] ADD note /note
 => exporting to image
 => => exporting layers
 => => writing image sha256:d7468aaa330be5e6ca6d060149a4fa09e96933a4459156223ed299
 => => naming to docker.io/library/build:add
(MoeLove) ➜  build docker run --rm build:add cat /note
file in container
```

可以看到，两者均可在镜像构建时，为镜像添加内容。那它们的区别是什么呢？

- ADD 除可用于正常拷贝文件外，还可添加 URL 形式的远程内容。
- ADD 可添加本地的 tar 归档文件或压缩文件（支持的格式为 gzip、bzip2 或 xz 等），并且会被解压。如果资源是来自远程的内容，则 **不会进行解压**。
- COPY 还可用于多阶段构建中，通过传递 `--from=` 的参数，可以从之前的阶段中拷贝内容到新的构建阶段中。

关键的区别主要就是上面这几条，在具体使用时，我建议如果 COPY 能满足需求的话，就尽量使用 COPY，除了因为 COPY 的行为更加清楚透明外，更关键考虑有两点：

1. 构建缓存
2. 减小镜像体积

当然这两点在下下节中会更深入探讨。

ARG vs ENV

ARG 和 ENV 均可用于在构建镜像过程中预定义变量。

但两者的主要区别如下：

- 生命周期不同：ARG 定义的变量只影响镜像构建阶段，但是 ENV 定义的变量会存在于镜像的整个声明周期，包括使用镜像创建容器，该变量仍然可用。
- ARG 在构建时，可通过 `--build-arg` 进行修改和指定，但是 ENV 指定的变量在构建时不可修改。

- 优先级不同：如果 ARG 和 ENV 定义的变量相同，且 ARG 在 ENV 之前，则 ENV 所定义的变量会覆盖 ARG 所定义的变量。
- 使用范围不同：ARG 可先于 FROM 使用，但 ENV 不可以。

总结来说，当你在使用时，如果需要在构建过程中修改变量的值，则使用 ARG 指令，如果是想要将值保留至镜像中，甚至是之后容器中使用的话，那使用 ENV 更为合适。另外，为了避免行为混淆，尽量避免 ARG 和 ENV 指定相同名称的变量，除非你已经很理解它们之间的行为。

ENTRYPOINT vs CMD

ENTRYPOINT 和 CMD 都定义了当容器运行时，需要执行的命令。 它们的主要区别如下：

- ENTRYPOINT 定义的行为，在启动容器时，需要指定 `--entrypoint` 才能覆盖，而 CMD 定义的行为是在 `docker run [OPTIONS] IMAGE [COMMAND] [ARG...]` 的 `[COMMAND] [ARG...]` 处进行覆盖。
- 当定义 ENTRYPOINT 后，如果 CMD 使用的是 exec 格式，即：`CMD ["aa", "bb"]` 形式的话，则其内容会直接作为 ENTRYPOINT 的参数。但如果 CMD 使用的是 shell 格式，即：`CMD aa bb` 的话，最终的连接形式为 `ENTRYPOINT sh -c aa bb`。

很多人在构建镜像或使用镜像时，常会被这两者搞混，导致镜像启动失败之类的问题。

我个人建议，在你构建镜像时，如果镜像是工具类镜像，即启动容器时类似直接执行该工具的形式，那么可直接将该命令使用 ENTRYPOINT 进行定义，这样用户在使用时，直接传递参数即可。

如果是想要组合使用 ENTRYPOINT 和 CMD 时，没有特殊需求的情况下，建议 ENTRYPOINT 和 CMD 均使用 exec 格式。

多阶段构建

在构建镜像时，为了能让最终产生只保留我们所需的内容，我们可以使用多阶段构建的方式。

具体而言就是 Dockerfile 中存在多个 FROM 指令，表示不同的阶段，后续阶段可以使用之前阶段的产物，或镜像中原本具备的内容。 比如下面的例子：

复制

```
FROM golang:1.11.1 AS builder

WORKDIR /go/src/be
COPY . /go/src/be
RUN go get -u github.com/golang/dep/cmd/dep \
        && dep ensure \
        && go build

FROM debian:stretch-slim
COPY --from=builder /go/src/be/be /usr/bin/be
ENTRYPOINT ["/usr/bin/be"]
EXPOSE 8080
```

给第一个 FROM 的阶段使用 AS 指令给一个别名，在后续阶段中则可以使用 `COPY --from` 通过别名来使用其中的内容。

当然，如果不给它别名的话，默认是从 0 开始给它一个索引，也就是说，上述的 `COPY --from=builder /go/src/be/be /usr/bin/be` 等价于 `COPY --from=0 /go/src/be/be /usr/bin/be`。

小结

本节我为你介绍了 Dockerfile 中的基础指令及一些经常被搞混的问题，以及简单地介绍了下多阶段构建。后续内容中，我们会继续深入到镜像的构建过程，并对其原理进行探究。并且会介绍镜像构建的最佳实践，到时，会以更为详尽的例子为你解释本篇内容中这些指令是如何影响我们的构建效率的。