文档：https://docs.docker.com/engine/reference/builder/

dockerfile介绍

dockerfile是什么

Dockerfile是一个创建镜像所有命令的文本文件, 包含了一条条指令和说明, 每条指令构建一层, 通过docker build命令,根据Dockerfile的内容构建镜像,因此每一条指令的内容, 就是描述该层如何构建.有了Dockefile, 就可以制定自己的docker镜像规则,只需要在Dockerfile上添加或者修改指令, 就可生成docker 镜像.

dockerfile解决了什么问题

Dockerfile 包含了镜像制作的完整操作流程，其他开发者可以通过 Dockerfile 了解并复现制作过程
Dockerfile 中的每一条指令都会创建新的镜像层，这些镜像可以被 Docker Daemon 缓存。再次制作镜像时，Docker 会尽量复用缓存的镜像层（using cache），而不是重新逐层构建，这样可以节省时间和磁盘空间
Dockerfile 的操作流程可以通过docker image history [镜像名称]查询，方便开发者查看变更记录

docker build 构建流程

docker build命令会读取Dockerfile的内容，并将Dockerfile的内容发送给 Docker 引擎，最终 Docker 引擎会解析Dockerfile中的每一条指令，构建出需要的镜像。
第一步，docker build会将 context 中的文件打包传给 Docker daemon。如果 context 中有.dockerignore文件，则会从上传列表中删除满足.dockerignore规则的文件。**注意：如果上下文中有相当多的文件，可以明显感受到整个文件发送过程**
这里有个例外，如果.dockerignore文件中有.dockerignore或者Dockerfile，docker build命令在排除文件时会忽略掉这两个文件。如果指定了镜像的 tag，还会对 repository 和 tag 进行验证。
第二步，docker build命令向 Docker server 发送 HTTP 请求，请求 Docker server 构建镜像，请求中包含了需要的 context 信息。
第三步，Docker server 接收到构建请求之后，会执行以下流程来构建镜像：

1. 创建一个临时目录，并将 context 中的文件解压到该目录下。
2. 读取并解析 Dockerfile，遍历其中的指令，根据命令类型分发到不同的模块去执行。
3. Docker 构建引擎为每一条指令创建一个临时容器，在临时容器中执行指令，然后 commit 容器，生成一个新的镜像层。
4. 最后，将所有指令构建出的镜像层合并，形成 build 的最后结果。最后一次 commit 生成的镜像 ID 就是最终的镜像 ID。

为了提高构建效率，docker build默认会缓存已有的镜像层。如果构建镜像时发现某个镜像层已经被缓存，就会直接使用该缓存镜像，而不用重新构建。如果不希望使用缓存的镜像，可以在执行docker build命令时，指定--no-cache=true参数。
Docker 匹配缓存镜像的规则为：遍历缓存中的基础镜像及其子镜像，检查这些镜像的构建指令是否和当前指令完全一致，如果不一样，则说明缓存不匹配。对于ADD、COPY指令，还会根据文件的校验和（checksum）来判断添加到镜像中的文件是否相同，如果不相同，则说明缓存不匹配。
这里要注意，缓存匹配检查不会检查容器中的文件。比如，当使用RUN apt-get -y update命令更新了容器中的文件时，缓存策略并不会检查这些文件，来判断缓存是否匹配。
最后，可以通过docker history命令来查看镜像的构建历史

关键字

```
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步
CMD 设置容器的启动命令
LABEL 设置镜像标签
EXPOSE 设置镜像暴露的端口
ENV 设置容器的环境变量
ADD 编译镜像时复制上下文中文件到镜像中
COPY 编译镜像时复制上下文中文件到镜像中
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
```

dockerfile 实践

基本语法实践

```
mkdir example1
cd example1
```

```
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
# 仅指定镜像元数据内容
LABEL hello 1.0.0
RUN git clone https://gitee.com/nickdemo/helloworld.git
WORKDIR helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

```
docker build  -t hello:1.0.0 -f Dockerfile .

docker run -p 80:80 -d --name hello hello:1.0.0
```

docker build上下文

1. 素材准备

```
# 创建一个目录，案例需要
mkdir example2
cd example2
# 下载nginx源码包，作为案例素材
curl https://nginx.org/download/nginx-1.21.6.tar.gz > ./nginx-1.21.6.tar.gz
# 下载app代码
git clone https://gitee.com/nickdemo/helloworld
```

```
# ./nginx*
# helloworld
```

2. 没有什么作用的上下文

```
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
# 仅指定镜像元数据内容
LABEL hello 1.0.0
RUN git clone https://gitee.com/nickdemo/helloworld.git
WORKDIR helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

```
# 调整为不同的上下文，查看不同的效果
docker build  -t hello:1.0.0 -f Dockerfile .
```

3. 上下文的真正作用

```
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
COPY ./helloworld /go/src/helloworld
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

```
docker build  -t hello:1.0.0 -f Dockerfile .
```

操作：

1. 尝试在.dockerignore 文件中忽略不同的内容，观察发送到守护进程的数据
2. 尝试 采用不同的上下文路径来构建镜像
3. 尝试 copy上下文以外的内容来构建镜像

多阶段构建以及ADD与COPY

多阶段构建

Docker 17.05版本以后，新增了Dockerfile多阶段构建。所谓多阶段构建，实际上是允许一个Dockerfile中出现多个 FROM 指令。这样做有什么意义呢?
多个 FROM 指令的意义
多个 FROM 指令并不是为了生成多根的层关系，最后生成的镜像，仍以最后一条 FROM 为准，之前的FROM 会被抛弃，那么之前的FROM 又有什么意义呢?
每一条 FROM 指令都是一个构建阶段，多条 FROM 就是多阶段构建，虽然最后生成的镜像只能是最后一个阶段的结果，但是，能够将前置阶段中的文件拷贝到后边的阶段中，这就是多阶段构建的最大意义。
最大的使用场景是将编译环境和运行环境分离，比如，之前我们需要构建一个Go语言程序，那么就需要用到go命令等编译环境

1. 素材准备

```
# 创建一个目录，案例需要
mkdir example3
cd example3
# 下载nginx源码包，作为案例素材
curl https://nginx.org/download/nginx-1.21.6.tar.gz > ./nginx-1.21.6.tar.gz
# 下载app代码
git clone https://gitee.com/nickdemo/helloworld
```

2. 一个最基本的dockerfile

```
FROM golang:1.18
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
COPY ./helloworld /go/src/helloworld
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

3. 多阶段构建dockerfile

```
FROM golang:1.18
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
```

```
COPY --from=0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

4. 通过as关键词，为构建阶段指定别名，可以提高可读性

```
FROM golang:1.18 as stage0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

ADD 与 COPY

1. ADD 与 COPY 不能拷贝**上下文**以外的文件
2. COPY 命令语法格式

```
COPY <src> <dest> //将上下文中源文件，拷贝到目标文件

COPY prefix* /destDir/  //将所有prefix 开头的文件拷贝到 destDir 目录下

COPY prefix?.log /destDir/  //支持单个占位符，例如 ： prefix1.log、prefix2.log 等
```

3. 对于目录而言，COPY 和 ADD 命令具有相同的特点：**只复制目录中的内容而不包含目录自身**

```
COPY srcDir /destDir/  //只会将源文件夹srcDir下的文件拷贝到 destDir 目录下
```

4. COPY 区别于ADD在于Dockerfile中使用multi-stage

```
FROM golang:1.18 as stage0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
```

```
CMD ["./app","--param1=p1","--param2=p2"]
```

5. ADD 命令语法

```
ADD <src> <dest>
```

ADD 命令除了不能用在 multistage 的场景下，ADD 命令可以完成 COPY 命令的所有功能，并且还可以完成两类的功能：

- 解压压缩文件并把它们添加到镜像中，对于宿主机本地压缩文件，ADD命令会自动解压并添加到镜像
- 从 url 拷贝文件到镜像中，需要注意：url 所在文件如果是压缩包，ADD 命令不会自动解压缩

6. ADD 案例

```
# 下载nginx tar.gz包作为素材
curl https://nginx.org/download/nginx-1.21.6.tar.gz > ./nginx-1.21.6.tar.gz
```

```
FROM golang:1.18 as stage0
# ADD ./helloworld /go/src/helloworld/
COPY ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
ADD https://nginx.org/download/nginx-1.21.6.tar.gz /soft/
COPY nginx-1.21.6.tar.gz /soft/copy/
ADD nginx-1.21.6.tar.gz /soft/add/
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

```
docker build -t hello:1.0.0 -f Dockerfile .
```

```
# 运行容器
docker run -d --name hello hello:1.0.0
# 查看add和copy的文件
docker exec -it hello /bin/sh
```

注意：目标文件位置要注意路径后面是否带 "/" ，带斜杠表示目录，不带斜杠表示文件名
文件名里带有空格，需要再 ADD（或COPY）指令里用双引号的形式标明：

```
ADD "space file.txt" "/tmp/space file.txt"
```

CMD 与 ENTRYPOINT

CMD

CMD 指令有三种格式

```
# shell 格式
CMD <command>
# exec格式，推荐格式
CMD ["executable","param1","param2"]
# 为ENTRYPOINT 指令提供参数
CMD ["param1","param2"]
```

1. CMD 指令提供容器运行时的默认值，这些默认值可以是一条指令，也可以是一些参数。
2. 一个dockerfile中可以有多条CMD指令，但只有最后一条CMD指令有效。
3. **CMD参数格式**是在CMD指令与ENTRYPOINT指令配合时使用，CMD指令中的参数会添加到ENTRYPOINT指令中。
4. 使用**shell 和exec 格式时**，命令在容器中的运行方式与RUN 指令相同。不同在于，RUN指令在构建镜像时执行命令，并生成新的镜像。
5. CMD指令在构建镜像时并不执行任何命令，而是在容器启动时默认将CMD指令作为第一条执行的命令。如果在命令行界面运行docker run 命令时指定命令参数，则会覆盖CMD指令中的命令。

```
FROM golang:1.18 as stage0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
CMD ["./app","--param1=p1","--param2=p2"]
```

```
docker build -t hello:1.0.0 .
# 指定启动命令和参数
docker run -d -p 80:80 hello:1.0.0 ./app --param1=1 --param2=2
# 指定启动命令为sh
docker run -dit -p 80:80 hello:1.0.0 sh
```

```
 curl http://localhost/print/startup
```

ENTRYPOINT

ENTRYPOINT指令有两种格式

```
# shell 格式
ENTRYPOINT <command>
# exec 格式，推荐格式
ENTRYPOINT ["executable","param1","param2"]
```

1. ENTRYPOINT指令和CMD指令类似，都可以让容器每次启动时执行相同的命令，但它们之间又有不同。一个Dockerfile中可以有多条ENTRYPOINT指令，但只有最后一条ENTRYPOINT指令有效。
2. 当使用shell格式时，ENTRYPOINT指令会忽略任何CMD指令和docker run 命令的参数，并且会运行在bin/sh -c中。
3. 推荐使用exec格式，使用此格式，docker run 传入的命令参数将会覆盖CMD指令的内容并且附加到ENTRYPOINT指令的参数中。
4. **CMD可以是参数，也可以是指令，ENTRYPOINT只能是命令；docker run 命令提供的运行命令参数可以覆盖CMD，但不能覆盖ENTRYPOINT。**
5. **可以通过docker run --**entrypoint 替换容器的入口程序

```
FROM golang:1.18 as stage0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app/
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
ENTRYPOINT ["./app","--param1=p1","--param2=p2"]
```

```
docker build -t hello:1.0.0 .
# 指定启动参数
docker run -d -p 80:80 hello:1.0.0  --param1=1 --param2=2
# 指定入口程序为sh
docker run -dit -p 80:80 --entrypoint sh  hello:1.0.0
```

```
 curl http://localhost/print/startup
```

build arg

1. dockerfile 预定义参数：HTTP_PROXY，http_proxy，HTTPS_PROXY，https_proxy，FTP_PROXY，ftp_proxy，NO_PROXY，no_proxy，ALL_PROXY，all_proxy。预定义参数，可直接在dockerfile中使用，无需使用arg来声明

```
FROM golang:1.18 as s0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
# RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN go env -w GOPROXY=$http_proxy
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
```

```
FROM alpine:latest as s1
ARG wd label tag
RUN echo $wd,$label,$tag
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL $label $tag
WORKDIR $wd
COPY --from=stage0 /go/src/helloworld/app ./
EXPOSE 80
ENTRYPOINT ["./app","--param1=p1","--param2=p2"]
```

```
docker build -t hello:1.0.0 --build-arg
"http_proxy=https://proxy.golang.com.cn,https://goproxy.cn,direct" --build-arg
"wd=/home/app/" --build-arg "label=myhello" --build-arg "tag=1.0.0" --no-cache .
```

target与cache-from

对于多阶段构建，可以通过--target指定需要重新构建的阶段。--cache-from 可以指定一个镜像作为缓存源，当构建过程中dockerfile指令与缓存镜像源指令匹配，则直接使用缓存镜像中的镜像层，从而加快构建进程。可以将缓存镜像推送到远程注册中心，提供给不同的构建过程使用，在使用前需要先pull到本地。

```
FROM golang:1.18 as s0
ADD ./helloworld /go/src/helloworld/
WORKDIR /go/src/helloworld
RUN go env -w GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .

FROM alpine:latest as s1
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app
COPY --from=s0 /go/src/helloworld/app ./
EXPOSE 80
ENTRYPOINT ["./app","--param1=p1","--param2=p2"]
```

```
docker build -t prehello:1.0.0 --target s0 .
```

```
docker build -t hello:1.0.0 --cache-from prehello:1.0.0 --target s1 .
```

onbuild

onbuild指令将指令添加到镜像中，当镜像作为另一个构建的基础镜像时，将触发这些指令执行。触发指令将在下游构建的上下文中执行。注意：onbuild指令不会影响当前构建，但会影响下游构建

1. 生成自定义基础镜像

```
FROM golang:1.18
ONBUILD ADD ./helloworld /go/src/helloworld/
ONBUILD WORKDIR /go/src/helloworld
ONBUILD RUN go env -w
GOPROXY=https://proxy.golang.com.cn,https://goproxy.cn,direct
ONBUILD RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app .
```

```
docker build -t mygolang:1.0.0  .
```

2. 通过自定义基础镜像构建新的镜像

```
FROM mygolang:1.0.0 as s0

FROM alpine:latest as s1
ENV env1=env1value
ENV env2=env2value
MAINTAINER nick
LABEL hello 1.0.0
WORKDIR /app
COPY --from=s0 /go/src/helloworld/app ./
EXPOSE 80
ENTRYPOINT ["./app","--param1=p1","--param2=p2"]
```

```
docker build -t hello:1.0.0 .
```