# 第二十五章 Dockerfile基本使用

本节所讲内容：

25.1 Dockerfile基础命令讲解与演示

25.2 实战-制作基于docker的python运行的uwsgi环境+Django小项目

## 25.1 Dockerfile基础命令讲解与演示

在一个空白目录中，建立一个文本文件，并命名为 Dockerfile：

[root@xuegod63 ~]# mkdir xuegod

[root@xuegod63 ~]# cd xuegod/

[root@xuegod63 xuegod]# vim Dockerfile

其内容为：

FROM nginx

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，FROM 和 RUN。

FROM 指定基础镜像（MANITAINER 作者信息）



[root@xuegod63 xuegod]# docker build -t mynginx:v1 . （这个地方有个.）

Step 1/2 : FROM nginx

Trying to pull repository docker.io/library/nginx ...

latest: Pulling from docker.io/library/nginx

1ab2bdfe9778: Pull complete

a17e64cfe253: Pull complete

e1288088c7a8: Pull complete

Digest: sha256:53ddb41e46de3d63376579acf46f9a41a8d7de33645db47a486de9769201fec9

Status: Downloaded newer image for docker.io/nginx:latest

---> 5a3221f0137b

Step 2/2 : RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

---> Running in f0d461c90e14

---> dcc1651132f4

Removing intermediate container f0d461c90e14

Successfully built dcc1651132f4

从网上拉取镜像，并进行配置

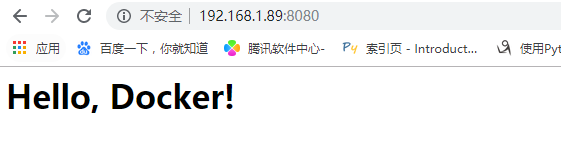
端口映射

[root@xuegod63 xuegod]# docker run -d -p8080:80 mynginx:v1

ed3d6b3991b3a370d471ebd3c93b9c65aec3cc10f5f98744398c5a95704e9e31



打开浏览器访问网络



所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 nginx 镜像的容器，再进行修改一样，基础镜像是必须指定的。

而 FROM 就是指定基础镜像，因此一个 Dockerfile 中 FROM 是必备的指令，并且必须是第一条指令。

注意：from 后面接的镜像，会先在本地寻找，寻找不到后会到网上寻找。

在 [Docker Store](https://store.docker.com/) 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如 [nginx](https://store.docker.com/images/nginx/)、[redis](https://store.docker.com/images/redis/)、[mongo](https://store.docker.com/images/mongo/)、[mysql](https://store.docker.com/images/mysql/)、[httpd](https://store.docker.com/images/httpd/)、[php](https://store.docker.com/images/php/)、[tomcat](https://store.docker.com/images/tomcat/) 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 [node](https://store.docker.com/images/node)、[openjdk](https://store.docker.com/images/openjdk/)、[python](https://store.docker.com/images/python/)、[ruby](https://store.docker.com/images/ruby/)、[golang](https://store.docker.com/images/golang/) 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。（offical）

如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 [ubuntu](https://store.docker.com/images/ubuntu/)、[debian](https://store.docker.com/images/debian/)、[centos](https://store.docker.com/images/centos/)、[fedora](https://store.docker.com/images/fedora/)、[alpine](https://store.docker.com/images/alpine/) 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。

FROM scratch

...

如果你以 scratch 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。

不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，比如 [swarm](https://hub.docker.com/_/swarm/)、[coreos/etcd](https://quay.io/repository/coreos/etcd)。对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 FROM scratch 会让镜像体积更加小巧。使用 [Go 语言](https://golang.org/) 开发的应用很多会使用这种方式来制作镜像，这也是为什么有人认为 Go 是特别适合容器微服务架构的语言的原因之一

RUN 执行命令

RUN 指令是用来执行命令行命令的。由于命令行的强大能力，RUN 指令在定制镜像时是最常用的指令之一。其格式有两种：

shell 格式:RUN <命令>，就像直接在命令行中输入的命令一样。刚才写的 Dockerfile 中的 RUN 指令就是这种格式。

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

exec 格式：RUN ["可执行文件", "参数1", "参数2"]，这更像是函数调用中的格式。

既然 RUN 就像 Shell 脚本一样可以执行命令，那么我们是否就可以像 Shell 脚本一样把每个命令对应一个 RUN 呢？比如这样：

[root@xuegod63 xuegod] # vim dockerfile\_v1

写入内容：

FROM debian:jessie

RUN apt-get update

RUN apt-get install -y gcc libc6-dev make

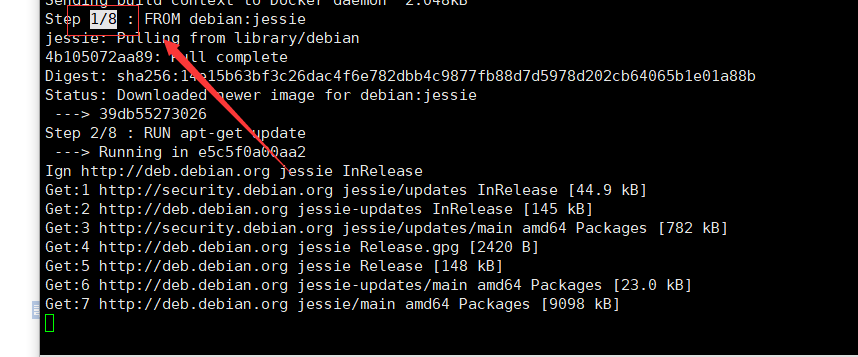
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz"

RUN mkdir -p /usr/src/redis

RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1

RUN make -C /usr/src/redis

RUN make -C /usr/src/redis install



[root@xuegod63 xuegod]# docker build -t myredis:v1 -f dockerfile\_v1 .

注意：-t（tag）：标签 –f（file）:指定文件

FROM debian:jessie

RUN buildDeps='gcc libc6-dev make' \

&& apt-get update \

&& apt-get install -y $buildDeps \

&& wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz" \

&& mkdir -p /usr/src/redis \

&& tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \

&& make -C /usr/src/redis \

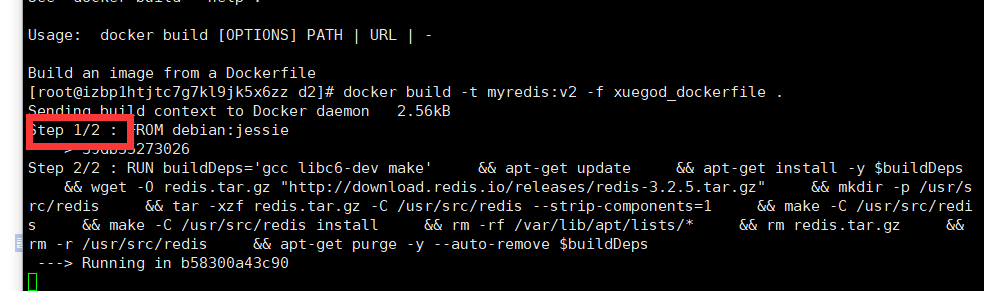
&& make -C /usr/src/redis install \

&& rm -rf /var/lib/apt/lists/\* \

&& rm redis.tar.gz \

&& rm -r /usr/src/redis \

&& apt-get purge -y --auto-remove $buildDeps



[root@xuegod63 xuegod]#docker build -t myredis:v2 -f xuegod\_dockerfile .

COPY 复制文件

格式：

COPY [--chown=<user>:<group>] <源路径（宿主机路径）>... <目标路径(容器内路径)>

COPY [--chown=<user>:<group>] ["<源路径1>",... "<目标路径>"]

<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](https://golang.org/pkg/path/filepath/#Match) 规则，如：

COPY hom\* /mydir/

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。

在使用该指令的时候还可以加上 --chown=<user>:<group> 选项来改变文件的所属用户及所属组。

COPY --chown=55:mygroup files\* /mydir/

COPY --chown=bin files\* /mydir/

COPY --chown=1 files\* /mydir/

COPY --chown=10:11 files\* /mydir/

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。下载后的文件权限自动设置为 600，如果这并不是想要的权限，那么还需要增加额外的一层 RUN进行权限调整，另外，如果下载的是个压缩包，需要解压缩，也一样还需要额外的一层 RUN 指令进行解压缩。所以不如直接使用 RUN 指令，然后使用 wget 或者 curl 工具下载，处理权限、解压缩、然后清理无用文件更合理。因此，这个功能其实并不实用，而且不推荐使用。

如果 <源路径> 为一个 tar 压缩文件的话，压缩格式为 gzip, bzip2 以及 xz 的情况下，ADD 指令将会自动解压缩这个压缩文件到 <目标路径> 去。

CMD 容器启动命令

CMD 指令的格式和 RUN 相似，也是两种格式：

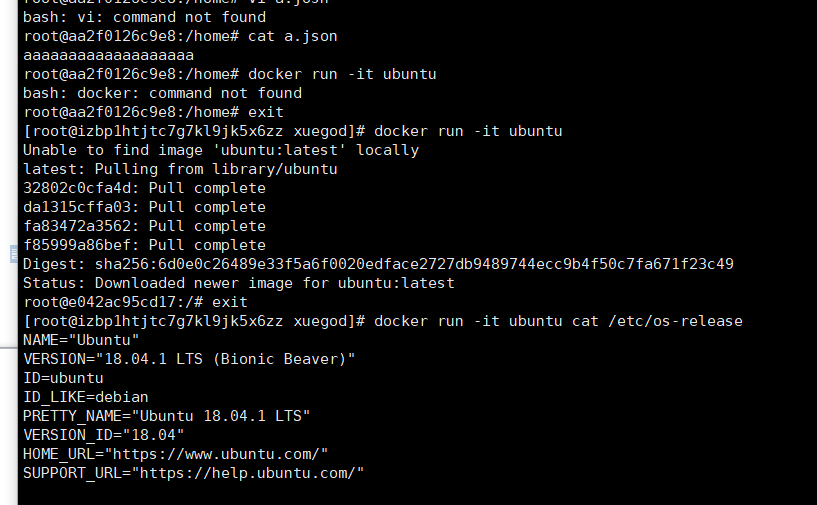
* shell 格式：CMD <命令>
* exec 格式：CMD ["可执行文件", "参数1", "参数2"...]

|  |
| --- |
| CMD ["nginx", "-g", "daemon off;"] |

之前介绍容器的时候曾经说过，Docker 不是虚拟机，容器就是进程。既然是进程，那么在启动容器的时候，需要指定所运行的程序及参数。CMD 指令就是用于指定默认的容器主进程的启动命令的。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令，比如，ubuntu 镜像默认的 CMD 是 /bin/bash，如果我们直接 docker run -it ubuntu 的话，会直接进入 bash。我们也可以在运行时指定运行别的命令，如 docker run -it ubuntu cat /etc/os-release。这就是用 cat /etc/os-release 命令替换了默认的 /bin/bash 命令了，输出了系统版本信息。

在指令格式上，一般推荐使用 exec 格式，这类格式在解析时会被解析为 JSON 数组，因此一定要使用双引号 "，而不要使用单引号。



CMD echo $HOME

在实际执行中，会将其变更为：

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

一些初学者将 CMD 写为：

CMD service nginx start

然后发现容器执行后就立即退出了。甚至在容器内去使用 systemctl 命令结果却发现根本执行不了。这就是因为没有搞明白前台、后台的概念，没有区分容器和虚拟机的差异，依旧在以传统虚拟机的角度去理解容器。

对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其它辅助进程不是它需要关心的东西。

而使用 service nginx start 命令，则是希望 upstart 来以后台守护进程形式启动 nginx 服务。而刚才说了 CMD service nginx start 会被理解为 CMD [ "sh", "-c", "service nginx start"]，因此主进程实际上是 sh。那么当 service nginx start 命令结束后，sh 也就结束了，sh 作为主进程退出了，自然就会令容器退出。

正确的做法是直接执行 nginx 可执行文件，并且要求以前台形式运行。比如：

CMD ["nginx", "-g（前台运行）", "daemon off;（关闭守护进程）"]

ENTRYPOINT 入口点

ENTRYPOINT 的格式和 RUN 指令格式一样，分为 exec 格式和 shell 格式。

ENTRYPOINT 的目的和 CMD 一样，都是在指定容器启动程序及参数。ENTRYPOINT 在运行时也可以替代，不过比 CMD 要略显繁琐，需要通过 docker run 的参数 --entrypoint 来指定。

当指定了 ENTRYPOINT 后，CMD 的含义就发生了改变，不再是直接的运行其命令，而是将 CMD 的内容作为参数传给 ENTRYPOINT 指令，换句话说实际执行时，将变为：

<ENTRYPOINT> "<CMD>"

那么有了 CMD 后，为什么还要有 ENTRYPOINT 呢？这种 <ENTRYPOINT> "<CMD>" 有什么好处么？让我们来看几个场景。

场景一：让镜像变成像命令一样使用

假设我们需要一个得知自己当前公网 IP 的镜像，那么可以先用 CMD 来实现：

FROM ubuntu:18.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

CMD [ "curl", "-s", "http://ifconfig.me" ]

假如我们使用 docker build -t myip . 来构建镜像的话，如果我们需要查询当前公网 IP，只需要执行：

$ docker run myip

嗯，这么看起来好像可以直接把镜像当做命令使用了，不过命令总有参数，如果我们希望加参数呢？比如从上面的 CMD 中可以看到实质的命令是 curl，那么如果我们希望显示 HTTP 头信息，就需要加上 -i 参数。那么我们可以直接加 -i 参数给 docker run myip 么？

$ docker run myip -i

docker: Error response from daemon: invalid header field value "oci runtime error: container\_linux.go:247: starting container process caused \"exec: \\\"-i\\\": executable file not found in $PATH\"\n".

我们可以看到可执行文件找不到的报错，executable file not found。之前我们说过，跟在镜像名后面的是 command，运行时会替换 CMD 的默认值。因此这里的 -i 替换了原来的 CMD，而不是添加在原来的 curl -s http://ip.cn 后面。而 -i 根本不是命令，所以自然找不到。

那么如果我们希望加入 -i 这参数，我们就必须重新完整的输入这个命令：

docker run myip curl -s http://ip.cn -i（替换原来的cmd）

这显然不是很好的解决方案，而使用 ENTRYPOINT 就可以解决这个问题。现在我们重新用 ENTRYPOINT 来实现这个镜像：

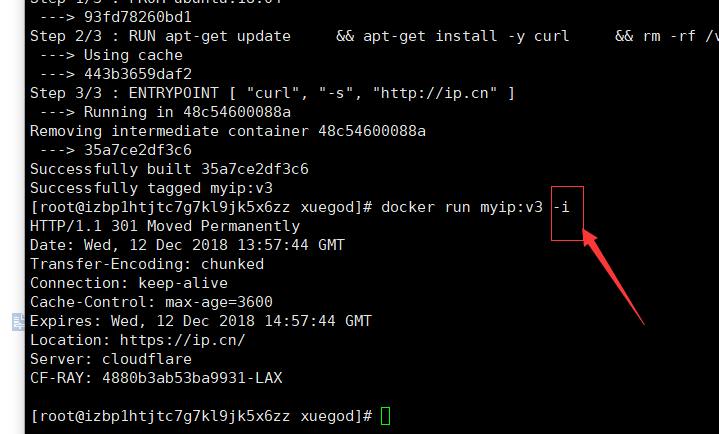
FROM ubuntu:18.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

ENTRYPOINT [ "curl", "-s", "http://ifconfig.me" ]（加-i视为追加命令）



ENV 设置环境变量

格式有两种：

* ENV <key> <value>
* 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。

ENV VERSION=1.0 DEBUG=on \

NAME="Happy Feet"

定义了环境变量，那么在后续的指令中，就可以使用这个环境变量。比如在官方 node 镜像 Dockerfile中，就有类似这样的代码：

ENV NODE\_VERSION 7.2.0

RUN curl -SLO "https://nodejs.org/dist/v$NODE\_VERSION/node-v$NODE\_VERSION-linux-x64.tar.xz" \

&& curl -SLO "https://nodejs.org/dist/v$NODE\_VERSION/SHASUMS256.txt.asc" \

&& gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \

&& grep " node-v$NODE\_VERSION-linux-x64.tar.xz\$" SHASUMS256.txt | sha256sum -c - \

&& tar -xJf "node-v$NODE\_VERSION-linux-x64.tar.xz" -C /usr/local --strip-components=1 \

&& rm "node-v$NODE\_VERSION-linux-x64.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt \

&& ln -s /usr/local/bin/node /usr/local/bin/nodejs

VOLUME 定义匿名卷

* VOLUME ["<路径1>", "<路径2>"...]
* VOLUME <路径>

之前我们说过，容器运行时应该尽量保持容器存储层不发生写操作，对于数据库类需要保存动态数据的应用，其数据库文件应该保存于卷(volume)中，后面的章节我们会进一步介绍 Docker 卷的概念。为了防止运行时用户忘记将动态文件所保存目录挂载为卷，在 Dockerfile 中，我们可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

VOLUME /data

这里的 /data 目录就会在运行时自动挂载为匿名卷，任何向 /data 中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。当然，运行时可以覆盖这个挂载设置。比如：

docker run -d -v mydata:/data xxxx

在这行命令中，就使用了 mydata 这个命名卷挂载到了 /data 这个位置，替代了 Dockerfile 中定义的匿名卷的挂载配置。

EXPOSE 声明端口

格式为 EXPOSE <端口1> [<端口2>...]。

EXPOSE 指令是声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是 docker run -P时，会自动随机映射 EXPOSE 的端口。

要将 EXPOSE 和在运行时使用 -p <宿主端口>:<容器端口> 区分开来。-p，是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而 EXPOSE 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

WORKDIR 指定工作目录

格式为 WORKDIR <工作目录路径>。

使用 WORKDIR 指令可以来指定工作目录（或者称为当前目录），以后各层的当前目录就被改为指定的目录，如该目录不存在，WORKDIR 会帮你建立目录。

之前提到一些初学者常犯的错误是把 Dockerfile 等同于 Shell 脚本来书写，这种错误的理解还可能会导致出现下面这样的错误：

RUN cd /app(千万不用这个方法切换)

RUN echo "hello" > world.txt（错误的）

WORKDIR /var/log（正确）

Run..........

WORKDIR /var/log/nginx

USER 指定当前用户

格式：USER <用户名>[:<用户组>]

## 25.2 实战-制作基于docker的python运行的uwsgi环境+Django小项目

制作python运行的uwsgi环境+Django。老师现场演示，大家跟着操作，跟不上的先听思路。

# Django开发环境python3版

postgresql-dev \ alpine集成pg

alpine-sdk \ alpine的pandas支持

## 启动

```

# make run

```

## 停止

```

# make stop

```

# 目录

```

.

├── Makefile #启动器

├── README.md

├── app #后台

│   ├── Dockerfile

│   ├── requirements.txt

│   ├── src

│   │   ├── xuegod

│   │   ├── test.log

│   └── uwsgi.ini

│   └── run.sh

├── docker-compose.yml

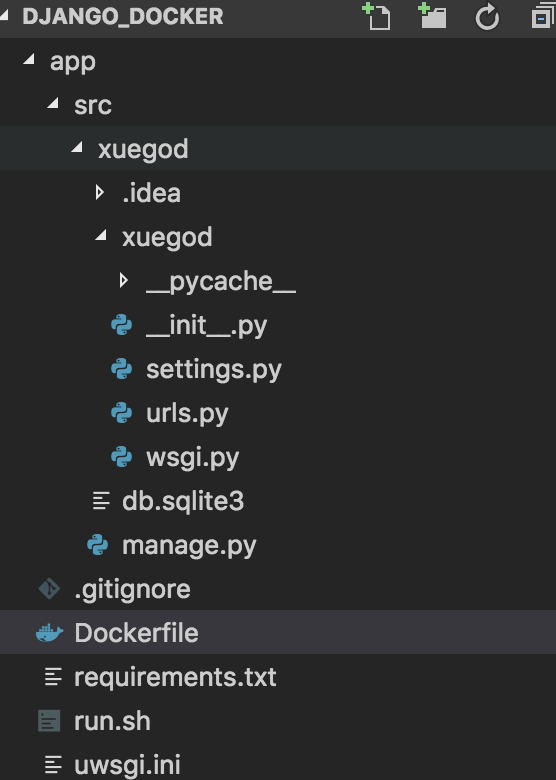
└── nginx #前端

├── Dockerfile

└── nginx.conf

```

步骤一：制作djangoapp

1. 首先使用命令 django-admin.py startproject xuegod (就是个文件)
2. 将xuegod拷贝到app目录下的src目录下
3. 在src目录下创建uwsgi文件
4. 编写依赖包requirements .txt
5. 编写Dockerfile
6. 

# 基础镜像

FROM python:3.6-alpine3.8

RUN mkdir /var/www

# 指定工作路径

WORKDIR /var/www

#设置时区和源

ENV TIME\_ZONE Asia/Shanghai

#使用清华源

RUN echo "https://mirror.tuna.tsinghua.edu.cn/alpine/v3.8/main/" > /etc/apk/repositories

RUN apk add --no-cache -U tzdata \

&& ln -sf /usr/share/zoneinfo/${TIME\_ZONE} /etc/localtime \

&& echo "${TIME\_ZONE}" > /etc/timezone

# 拷贝安装python包清单

COPY . ./

# 安装项目依赖包

RUN apk update && apk add \

bash \

libuuid \

pcre \

mailcap \

gcc \

libc-dev \

linux-headers \

pcre-dev \

alpine-sdk \

&& python -m pip install --upgrade --force pip \

&& pip install setuptools \

&& pip install --no-cache-dir -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple \

&& apk del \

gcc \

libc-dev \

linux-headers \

&& rm -rf /tmp/\*

#切换工作目录

WORKDIR /var/www/src

6.docker build -t xuegod/django .

7.docker run -itd -p 8888:8888 xuegod/django

8．docker ps -a

总结：

25.1 Dockerfile基础命令讲解与演示

25.2 实战-制作基于docker的python运行的uwsgi环境+Django小项目