2017-05-14

如何编写最佳的Dockerfile

译者按: Dockerfile的语法非常简单，然而如何加快镜像构建速度，如何减少Docker镜像的大小却不是那么直观，需要积累实践经验。这篇博客可以帮助你快速掌握编写Dockerfile的技巧。

原文: How to write excellent Dockerfiles

译者: Fundebug

为了保证可读性，本文采用意译而非直译。另外，本文版权归原作者所有，翻译仅用于学习。

我已经使用Docker有一段时间了，其中编写Dockerfile是非常重要的一部分工作。在这篇博客中，我打算分享一些建议，帮助大家编写更好的Dockerfile。

目标:

- 更快的构建速度
- 更小的Docker镜像大小
- 更少的Docker镜像层
- 充分利用镜像缓存
- 增加Dockerfile可读性
- 让Docker容器使用起来更简单

总结

- 编写.dockerignore文件
- 容器只运行单个应用
- 将多个RUN指令合并为一个
- 基础镜像的标签不要用latest
- 每个RUN指令后删除多余文件
- 选择合适的基础镜像(alpine版本最好)
- 设置WORKDIR和CMD
- 使用ENTRYPOINT (可选)
- 在entrypoint脚本中使用exec
- COPY与ADD优先使用前者

- 合理调整COPY与RUN的顺序
- 设置默认的环境变量，映射端口和数据卷
- 使用LABEL设置镜像元数据
- 添加HEALTHCHECK

Docker运行一个Node.js应用，下面就是它的Dockerfile(CMD指令太复杂了，所以我简化了，它是错误的，仅供参考)。


```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y nodejs ssh mysql
RUN cd /app && npm install

# this should start three processes, mysql and ssh
# in the background and node app in foreground
# isn't it beautifully terrible? <3
CMD mysql & sshd & npm start
```

构建镜像:

```
docker build -t wtf .
```

1. 编写.dockerignore文件

构建镜像时，Docker需要先准备 context ，将所有需要的文件收集到进程中。默认的 context 包含Dockerfile目录中的所有文件，但是实际上，**我们并不需要.git目录，node_modules目录等内容。** .dockerignore 的作用和语法类似于 .gitignore ，可以忽略一些不需要的文件，这样可以有效加快镜像构建时间，同时减少Docker镜像的大小。示例如下:

```
.git/
node_modules/
```

2. 容器只运行单个应用

从技术角度讲，你可以在Docker容器中运行多个进程。你可以将数据库，前端，后端，ssh，supervisor都运行在同一个Docker容器中。但是，这会让你非常痛苦:

- 非常长的构建时间(修改前端之后，整个后端也需要重新构建)
- 非常大的镜像大小
- 多个应用的日志难以处理(不能直接使用stdout，否则多个应用的日志会混合到一起)
- 横向扩展时非常浪费资源(不同的应用需要运行的容器数并不相同)

器。

现在，我从Dockerfile中删除一些不需要的安装包，另外，SSH可以用docker exec替代。示例如下：

```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y

# we should remove ssh and mysql, and use
# separate container for database
RUN apt-get install -y nodejs  # ssh mysql
RUN cd /app && npm install

CMD npm start
```

3. 将多个RUN指令合并为一个

Docker镜像是分层的，下面这些知识点非常重要:

- Dockerfile中的每个指令都会创建一个新的镜像层。
- 镜像层将被缓存和复用
- 当Dockerfile的指令修改了，复制的文件变化了，或者构建镜像时指定的变量不同了，对应的镜像层缓存就会失效
- 某一层的镜像缓存失效之后，它之后的镜像层缓存都会失效
- 镜像层是不可变的，如果我们再某一层中添加一个文件，然后在下一层中删除它，则镜像中依然会包含该文件(只是这个文件在Docker容器中不可见了)。

Docker镜像类似于洋葱。它们都有很多层。为了修改内层，则需要将外面的层都删掉。记住这一点的话，其他内容就很好理解了。

现在，我们**将所有的RUN指令合并为一个**。同时把 apt-get upgrade 删除，因为它会使得镜像构建非常不确定(我们只需要依赖基础镜像的更新就好了)

```
FROM ubuntu

ADD . /app

RUN apt-get update \
    && apt-get install -y nodejs \
```

```
CMD npm start
```


记住一点，我们只能将变化频率一样的指令合并在一起。将node.js安装与npm模块安装放在一起的话，则每次修改源代码，都需要重新安装node.js，这显然不合适。因此，正确的写法是这样的:

```
FROM ubuntu

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```

4. 基础镜像的标签不要用latest

当镜像没有指定标签时，将默认使用 latest 标签。因此， FROM ubuntu 指令等同于 FROM ubuntu:latest 。当时，当镜像更新时，latest标签会指向不同的镜像，这时构建镜像有可能失败。如果你的确需要使用最新版的基础镜像，可以使用latest标签，否则的话，最好指定确定的镜像标签。

示例Dockerfile应该使用 16.04 作为标签。

```
FROM ubuntu:16.04  # it's that easy!

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```

5. 每个RUN指令后删除多余文件

假设我们更新了apt-get源，下载，解压并安装了一些软件包，它们都保存在 /var/lib/apt/lists/ 目录中。但是，运行应用时Docker镜像中并不需要这些文件。我们最好将它们删除，因为它会使Docker镜像变大。

示例Dockerfile中，我们可以删除 /var/lib/apt/lists/ 目录中的文件(它们是由apt-get update生成的)。

```
FROM ubuntu:16.04

RUN apt-get update \
```

```
ADD . /app
RUN cd /app && npm install

CMD npm start
```

6. 选择合适的基础镜像(alpine版本最好)

在示例中，我们选择了 ubuntu 作为基础镜像。但是我们只需要运行node程序，有必要使用一个通用的基础镜像吗？ node 镜像应该是更好的选择。

```
FROM node

ADD . /app
# we don't need to install node
# anymore and use apt-get
RUN cd /app && npm install

CMD npm start
```

更好的选择是alpine版本的 node 镜像。alpine是一个极小化的Linux发行版，只有4MB，这让它非常适合作为基础镜像。

```
FROM node:7-alpine

ADD . /app
RUN cd /app && npm install

CMD npm start
```

apk是Alpine的包管理工具。它与 apt-get 有些不同，但是非常容易上手。另外，它还有一些非常有用的特性，比如 no-cache 和 --virtual 选项，它们都可以帮助我们减少镜像的大小。

7. 设置WORKDIR和 CMD

WORKDIR指令可以设置默认目录，也就是运行 RUN / CMD / ENTRYPOINT 指令的地方。

CMD指令可以设置容器创建是执行的默认命令。另外，你应该讲命令写在一个数组中，数组中每个元素为命令的每个单词(参考官方文档)。

```
ADD . /app
RUN npm install

CMD ["npm", "start"]
```


8. 使用ENTRYPOINT (可选)

ENTRYPOINT指令并不是必须的，因为它会增加复杂度。 ENTRYPOINT 是一个脚本，它会默认执行，并且将指定的命令错误其参数。它通常用于构建可执行的Docker镜像。entrypoint.sh如下:

```
#!/usr/bin/env sh
# $0 is a script name,
# $1, $2, $3 etc are passed arguments
# $1 is our command
CMD=$1

case "$CMD" in
  "dev" )
    npm install
    export NODE_ENV=development
    exec npm run dev
    ;;

  "start" )
    # we can modify files here, using ENV variables passed in
    # "docker create" command. It can't be done during build process.
    echo "db: $DATABASE_ADDRESS" >> /app/config.yml
    export NODE_ENV=production
    exec npm start
    ;;

  * )
    # Run custom command. Thanks to this line we can still use
    # "docker run our_image /bin/bash" and it will work
    exec $CMD ${@:2}
    ;;
esac
```

示例Dockerfile:

```
FROM node:7-alpine
```

```
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```


可以使用如下命令运行该镜像:

```
# 运行开发版本
docker run our-app dev

# 运行生产版本
docker run our-app start

# 运行bash
docker run -it our-app /bin/bash
```

9. 在entrypoint脚本中使用exec

在前文的entrypoint脚本中，我使用了 exec 命令运行node应用。不使用 exec 的话，我们则不能顺利地关闭容器，因为SIGTERM信号会被bash脚本进程吞没。 exec 命令启动的进程可以取代脚本进程，因此所有的信号都会正常工作。

10. COPY与ADD优先使用前者

COPY指令非常简单，仅用于将文件拷贝到镜像中。ADD相对来讲复杂一些，可以用于下载远程文件以及解压压缩包(参考官方文档)。

```
FROM node:7-alpine

WORKDIR /app

COPY . /app
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

11. 合理调整COPY与RUN的顺序

我们应该**把变化最少的部分放在Dockerfile的前面**，这样可以充分利用镜像缓存。

```
FROM node:7-alpine

WORKDIR /app

COPY package.json /app
RUN npm install
COPY . /app

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

12. 设置默认的环境变量，映射端口和数据卷

运行Docker容器时很可能需要一些环境变量。在Dockerfile设置默认的环境变量是一种很好的方式。另外，我们应该在Dockerfile中设置映射端口和数据卷。示例如下:

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
    NODE_ENV=production \
    APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

ENV指令指定的环境变量在容器中可以使用。如果你只是需要指定构建镜像时的变量，你可以使用ARG指令。

13. 使用LABEL设置镜像元数据

```
FROM node:7-alpine
LABEL maintainer "jakub.skalecki@example.com"
...
```

14. 添加HEALTHCHECK

运行容器时，可以指定 --restart always 选项。这样的话，容器崩溃时，Docker守护进程(docker daemon)会重启容器。对于需要长时间运行的容器，这个选项非常有用。但是，如果容器的确在运行，但是不可(陷入死循环，配置错误)用怎么办？使用HEALTHCHECK指令可以让Docker周期性的检查容器的健康状况。我们只需要指定一个命令，如果一切正常的话返回0，否则返回1。对HEALTHCHECK感兴趣的话，可以参考这篇博客。示例如下:

```
FROM node:7-alpine
LABEL maintainer "jakub.skalecki@example.com"

ENV PROJECT_DIR=/app
WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
    NODE_ENV=production \
    APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT
HEALTHCHECK CMD curl --fail http://localhost:$APP_PORT || exit 1

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

当请求失败时， curl --fail 命令返回非0状态。

欢迎加入我们Fundebug的**Docker技术交流群: 305097057。**

关于Fundebug

Fundebug是全栈JavaScript实时错误监测平台，能够及时发现您的应用错误，助您提升用户体验。

标签

Docker JavaScript 小程序 BUG Source Map Node.js 新闻 翻译 Linux Java Swarm 客户故事 Dockerfile

最新博客

热门博客