如何优化 node 项目的 docker 镜像（像老板压榨员工一样压榨镜像）

https://juejin.cn/post/6991689670027542564

2021年08月02日 阅读 4708 关注 Fri Aug 13 21:57

本文将以 Node 程序展示如何优化 Docker 镜像（优化思想是通用的，不分程序），主要解决镜像大小过大、CI/CD 构建镜像速度，本文演示如何一步步优化 Dockerfile 文件，绝对的干货，建议先赞再看，看完不是干货再取消赞也不是不行😄。优化的结果如下：

1. **大小从 1.06G 到 73.4M**
2. **构建速度从 29.6 秒到 1.3 秒**（对比的是第二次构建的速度）

Node 项目

简单写了一个自己用的 wechat-bot ，接下来就以这个项目演示怎么去优化 Docker 镜像

以下是我没有仔细研究 Docker 刚开始写的 Dockerfile 文件

```
FROM node:14.17.3

# 设置环境变量
ENV NODE_ENV=production
ENV APP_PATH=/node/app

# 设置工作目录
WORKDIR $APP_PATH

# 把当前目录下的所有文件拷贝到镜像的工作目录下 .dockerignore 指定的文件不会拷贝
COPY . $APP_PATH

# 安装依赖
RUN yarn

# 暴露端口
EXPOSE 4300

CMD yarn start
复制代码
```

build 之后，如下图，我这个简单的 Node 程序镜像竟然有 **1G** 多，接下来我们将逐步去优化减少这个大小

```
iamc@ruanhongchaodeMacBook-Pro wechat-bot % docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
wechat-bot   latest    080b2a6fc68b   13 seconds ago   1.06GB
```

优化前言

在优化之前，有些东西我们必须了解，解决问题的第一步就是先找出导致问题的原因

1. Dockerfile 文件，其内包含了一条条的指令，每一条指令构建一层，因此每一条指令的内容，就是描述该层如何构建

2. Docker 镜像并非只是一个文件，而是由一堆文件组成，最主要的文件是 **层**（Layers）

    - 镜像构建时，会一层层构建，前一层是后一层的基础

      每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像

    - 镜像层将会被缓存和复用（这也是从第二次开始构建镜像时，速度会快的原因，优化镜像构建速度的原理也是利用缓存原理来做）

    - 当 Dockerfile 的指令修改了，操作的文件变化了，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

      docker build 的缓存机制，docker 是怎么知道文件变化的呢?

      Docker 采取的策略是：获取 Dockerfile 下内容（包括文件的部分 inode 信息），计算出一个唯一的 hash 值，若 hash 值未发生变化，则可以认为文件内容没有发生变化，可以使用缓存机制，反之亦然。

    - 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

    - 镜像的每一层只记录文件变更，在容器启动时，Docker 会将镜像的各个层进行计算，最后生成一个文件系统

      当我知道这点时，我恍然大悟，我们使用的操作系统，比如安卓、ios、win、mac 等，其实就是一个文件系统，我们的软件界面交互等，其实就是在读写文件，我们网页写个弹框，操作 dom，就是在读写本地文件或者是读写内存里的数据，个人的一些见解不知道对不对，本人非科班出身的前端 coder 😄

    参考资料：[docker 镜像分层原理]

ok，我们已经知道镜像是由多层文件系统组成，想要优化它的大小，就需要去减少层数、每一层尽量只包含该层需要的东西，任何额外的东西应该在该层构建结束前清理掉，下面开始正文

优化 Dockerfile

优化第一层 `FROM node:14.17.3`

方案一：使用 node 的 Alpine 版本

这也是绝多数人知道的优化镜像手段，Alpine 是一个很小的 Linux 发行版，只要选择 Node 的 Alpine 版本，就会有很大改进，我们把这一句改成指令改成 `FROM node:14.17.4-alpine`（可以去 dockerhub 查看 node 有哪些版本标签），build 后镜像大小如下图，瞬间**从 1.06G 降到 238M**，可以说是效果显著

```
iamc@ruanhongchaodeMacBook-Pro wechat-bot % docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
wechat-bot   latest    93c25b23a3b0   8 seconds ago    238MB
```

还可以使用其它的基础小镜像，比如 mhart/alpine-node，这个还能再小，改成 `FROM mhart/alpine-node:14.17.3` 再试试，可以看到又小了 **5M** 😂，虽然不多，但是秉着能压榨一点是一点的“老板原则”，积少成多，极致压榨

```
iamc@ruanhongchaodeMacBook-Pro wechat-bot % docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
wechat-bot   latest    6daabab4a1dc   9 seconds ago    233MB
```

方案二：使用纯净 Alpine 镜像手动装 Node

既然 Alpine 是最小的 Linux，那我们试下用纯净的 Alpine 镜像，自己再装 Node 试试

```
FROM alpine:latest

# 使用 apk 命令安装 nodejs 和 yarn，如果使用 npm 启动，就不需要装 yarn
RUN apk add --no-cache --update nodejs=14.17.4-r0 yarn=1.22.10-r0

# ... 后面的步骤不变
复制代码
```

build 后看下图，只有 **174M** 了，又小了不少

```
iamc@ruanhongchaodeMacBook-Pro wechat-bot % docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
wechat-bot   latest    60973a1eb1ab   16 seconds ago   174MB
```

结论就是不嫌麻烦追求极致就用方案二，**从 1.06G 减少到 174M**

减少层数、不经常变动的层提到前面去

- `ENV` 指令是可以一次性设置多个环境变量，能一次指令执行完，就不用两次，多一个指令就多一层
- `EXPOSE` 指令是暴露端口，其实也可以不用写这个指令，在启动容器的时候自己映射端口，如果写了这个指令的话，因为端口不经常变，所以把这个指令提前，写上这个指令有两个好处：
    1. 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
    2. 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

至于写还是不写，看个人吧，我个人一般不写，因为我在项目启动命令会指定项目端口，启动容器的时候映射出来就好，这样我就要维护一个地方，Dockerfile 也写了的话，项目端口变了，这里也要修改，多了点维护成本，当然也有办法让两边端口变量取自配置文件，只要改配置文件即可

下面是改写后的 Dockerfile

```
FROM alpine:latest

# 使用 apk 命令安装 nodejs 和 yarn，如果使用 npm 启动，就不需要装 yarn
RUN apk add --no-cache --update nodejs=14.17.4-r0 yarn=1.22.10-r0

# 暴露端口
EXPOSE 4300

# 设置环境变量
ENV NODE_ENV=production \
    APP_PATH=/node/app

# 设置工作目录
WORKDIR $APP_PATH

# 把当前目录下的所有文件拷贝到镜像的工作目录下 .dockerignore 指定的文件不会拷贝
COPY . $APP_PATH

# 安装依赖
RUN yarn

# 启动命令
CMD yarn start
复制代码
```

这一步的优化，无论从镜像大小还是构建镜像速度都看不到明显的差别，因为改动的层内容少（体现不出来），但是可以查看到镜像的层是变少了的，可以自行试试查看镜像的层试试

减少镜像层数是“好老板”的传统优良习惯，不让“员工”浪费资源

package.json 提前提高编译速度

从下图可以看到每次我们 build 的时候最耗时的就是在执行 `yarn` 命令装依赖的时候，大部分时候我们只是改代码，依赖不变，这时候如果可以让这一步缓存起来，依赖没有变化的时候，就不需要重新装依赖，就可以大大改进编译速度

前面我们说了镜像构建时，是一层层构建，前一层是后一层的基础，既然是这样的话，我们就把 package.json 文件单独提前拷贝到镜像，然后下一步装依赖，执行命令装依赖这层的前一层是拷贝 package.json 文件，因为安装依赖命令不会变化，所以只要 package.json 文件没变化，就不会重新执行 `yarn` 安装依赖，它会复用之前安装好的依赖，原理讲清楚了，下面我们看效果

改变后的 Dockerfile 文件

```
FROM alpine:latest

# 使用 apk 命令安装 nodejs 和 yarn，如果使用 npm 启动，就不需要装 yarn
RUN apk add --no-cache --update nodejs=14.17.4-r0 yarn=1.22.10-r0

# 暴露端口
EXPOSE 4300

# 设置环境变量
ENV NODE_ENV=production \
    APP_PATH=/node/app

# 设置工作目录
WORKDIR $APP_PATH

# 拷贝 package.json 到工作跟目录下
COPY package.json .

# 安装依赖
RUN yarn

# 把当前目录下的所有文件拷贝到镜像的工作目录下 .dockerignore 指定的文件不会拷贝
COPY . .

# 启动命令
CMD yarn start
```
复制代码

build 看下图，编译时间从 29.6s 到 1.3s，使用了缓存的层前面会有个 CACHED 字眼，仔细看下图可以看到

充分利用 docker 缓存特性是优化构建速度的利器

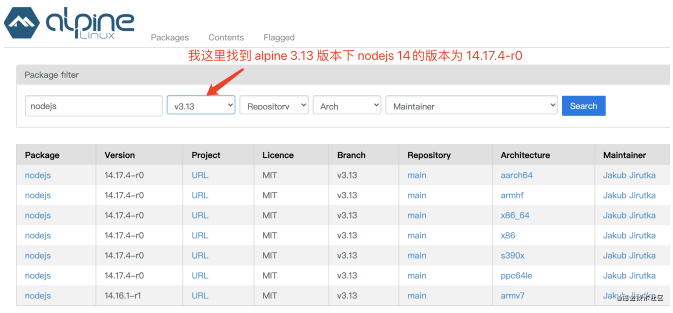
使用多阶段构建再次压榨镜像大小

多阶段构建这里不多说了，不了解的可以先搜相关资料了解

因为我们运行 node 程序是只需要生产的依赖和最终 node 可以运行的文件，就是说我们运行项目只需要 package.js 文件里 dependencies 里的依赖，devDependencies 依赖只是编译阶段用的，比如 eslint 等这些工具在项目运行时是用不到的，再比如我们项目是用 typescript 写的，node 是不能直接运行 ts 文件，ts 文件需要编译成 js 文件，运行项目我们只需要编译后的文件和 dependencies 里的依赖就可以运行，也就是说最终镜像只需要我们需要的东西，任何其他东西都可以删掉，下面我们使用多阶段改写 Dockerfile

```
# 构建基础镜像
    FROM alpine:3.14 AS base

    # 设置环境变量
    ENV NODE_ENV=production \
        APP_PATH=/node/app

    # 设置工作目录
    WORKDIR $APP_PATH

    # 安装 nodejs 和 yarn
    RUN apk add --no-cache --update nodejs=14.17.4-r0 yarn=1.22.10-r0

# 使用基础镜像 装依赖阶段
    FROM base AS install

    # 拷贝 package.json 到工作跟目录下
    COPY package.json ./

    # 安装依赖
    RUN yarn

# 最终阶段，也就是输出的镜像是这个阶段构建的，前面的阶段都是为这个阶段做铺垫
    FROM base

    # 拷贝 装依赖阶段 生成的 node_modules 文件夹到工作目录下
    COPY --from=install $APP_PATH/node_modules ./node_modules

    # 将当前目录下的所有文件（除了.dockerignore排除的路径），都拷贝进入镜像的工作目录下
    COPY . .

    # 启动
    CMD yarn start
复制代码
```

细心的朋友会发现我这里有指定 alpine 版本，而上面都是用的 latest 版本，因为就在刚刚发现有个坑需要注意下，就是我们选择 alpine 版本的时候，最好不要选择 latest 版本，因为后面要装的软件版本可能会在 alpine 的 latest 版本没有对应软件的版本号，就会安装错误，我刚刚就翻车了，[点击查看 alpine 版本下的包信息]



build 后，我们看看镜像大小，上次的是 174M 再次降到 73.4M，极致压榨。镜像：”放过我把，我真的没有了“😂

讲解：

我把这个构建分成了三个阶段：

1. 第一阶段：构建基础镜像

   安装依赖、编译、运行等等阶段，就是所有阶段共用的东西都在第一阶段封到一个基础镜像里供其它阶段使用，比如设置环境变量、设置工作目录、安装 nodejs、yarn 等等

2. 第二阶段：装依赖阶段

   在这个阶段，装依赖，如果项目需要编译，可以在这个阶段装依赖编译好

   这里在说下装依赖的小细节，就是执行 `yarn --production` 加个 production 参数或者环境变量 `NODE_ENV` 为 `production`，yarn 将不会安装 devDependencies 中列出的任何软件包，[点我查看官方文档说明]，因为我设置了环境变量所以就没加这个参数

3. 第三阶段：最终使用镜像

   拷贝第二阶段安装的好的依赖文件夹，然后在拷贝代码文件到工作目录，执行启动命令，第二阶段装依赖多出的一些垃圾我们不需要，我们就只拷贝我们要用的东西，大大减少镜像的大小

   如果项目需要编译，在拷贝编译后的文件夹，不需要拷贝编译前的代码，有编译后的代码和依赖就可以跑起项目

多阶段构建，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。

最终优化成果：

1. **大小从 1.06G 到 73.4M**
2. **构建速度从 29.6 秒到 1.3 秒**（对比的是第二次构建的速度）

至此，压榨镜像手段就完了，如果各位老板还有压榨手段可以分享分享

镜像内心独白：”你礼貌吗？还来“

github 的 actions 构建镜像问题

github 提供的 actions，每次都是一个干净的实例，什么意思，就是每次执行，都是干净的机器，这会导致一个问题，会导致 docker 没法使用缓存，那有没有解决办法呢，我想到了两种解决办法：

1. docker 官方提供的 action 缓存方案

   我用的是 Github cache 方案

2. 自托管 actions 运行机器

   相当于 gitlab 的 runner 一样，自己提供运行器，自己提供的就不会每次都是干净的机器，详情看 actions 官方文档

3. 先构建一个已经安装好依赖包的镜像，然后基于此镜像再次构建，相当于多阶段构建，把前两个阶段构建的镜像产物推送到镜像仓库，再以这个镜像为基础去构建后续部分。借助镜像仓库存储基础镜像从而达到缓存的效果（此方案来源于评论里的大佬）

```
# 以这个镜像为基础去构建，这个镜像是已经装好项目依赖的镜像并推送到镜像仓库里，这里从镜像仓库拉下来
FROM project-base-image:latest

COPY . .

CMD yarn start
复制代码
```

参考资料：

- 在 GitHub Actions 上使用 Docker 层缓存构建镜像

最后

项目仓库地址 wechat-bot

文章有错误的地方欢迎指正，避免误人子弟


等我写完了代码，
我也要和你们一起玩