Dockerfile 优化...

> 这是本专栏的第三部分：镜像篇，共 8 篇。前四篇我分别为你介绍了如何对 Docker 镜像进行生命周期的管理，如何使用 Dockerfile 进行镜像的构建和分发以及 Docker 的构建系统和下一代构建系统——BuildKit。下面我们一起进入本篇的学习。

在前期的课程中，我们已经知道使用 Dockerfile 构建镜像，是当前最为普遍，也最为标准的方式。

随着业务容器化进度的推进，以及 Kubernetes 等云原生技术的普及，不可避免的技术之一便是构建镜像。

Dockerfile 的语法并不算多，之前的内容中也有所介绍。同样的需求，不同的人写出来的 Dockerfile 可能相差不多，但 Docker 有不少的“默认行为”，不同的 Dockerfile 构建镜像的效率和后期的维护性也相差甚远。

使用 Dockerfile

虽然本文的标题是 **Dockerfile 的优化和最佳实践**，但此处还是要再次重申下，**请尽量使用 Dockerfile 描述镜像的构建过程。**主要原因如下：

- 过程可追溯：根据之前课程的内容，想必大家也已经了解了 Dockerfile 是在描述镜像的构建过程，大家均可通过 Dockerfile 看到镜像构建时需要执行的步骤，或者其需要安装的依赖等。
- 变更可管理：Dockerfile 是个纯文本文件，配合 Git 做版本控制，可以很清晰的查找到每个版本之间的变更。
- 基于上述两个原因，使用 Dockerfile 来描述镜像的构建过程，从可维护性上来看，也是首选。
- 易于优化：最后一个主要原因，也是本篇内容的重点，使用 Dockerfile 描述出来的过程，可以直接通过修改 Dockerfile 来修改构建过程，并对其进行优化，包括构建效率，最终镜像体积等。

从一个示例程序开始

在接下来的内容中，我会以一个 Spring Boot 的示例项目来为你介绍如何对 Dockerfile 进行优化。

在后续内容开始前，这里需要解释下为何选择了 Spring Boot 作为示例项目：

- Java 系的项目，需要有运行时环境支持（需要装 Java 环境）；
- 有安装依赖的过程；
- 最终产物与所安装的依赖无关（最后的产物是 jar 包）。

上述的三点是基于项目做功能演示方面的考虑；另一个考虑是 Spring Boot 当前应用还算比较广泛，使用此项目举例，也还比较直观。

尽可能使用 Docker 官方镜像

在需要为此项目写 Dockerfile 构建 Docker image 的时候，可能会有人将 Dockerfile 写成这样：

```
FROM debian:latest

# 安装运行时
RUN apt-get update
RUN apt-get install -y openjdk-8-jdk

# 拷贝文件到镜像
COPY . /app
WORKDIR /app
RUN ./mvnw package

CMD ["java", "-jar", "/app/target/gs-spring-boot-0.1.0.jar"]
```

我们来看看这个 Dockerfile 中的内容。

首先使用了 `debian:latest` 作为基础镜像；其次由于要跑 Java 项目，安装了 openjdk-8-jdk；接下来将项目代码 COPY 到 /app 目录，并将默认的工作目录也设置到了这里。

然后就使用 MVN 进行打包，最后在 CMD 命令上启动服务。

在 Docker Hub 上 我们可以找到很多官方镜像，如果使用对应的官方镜像作基础镜像的话，则可省略掉开头安装 openjdk-8-jdk 的步骤，以此减少构建过程中的耗时。

修改后的 Dockerfile 如下：

```
FROM maven:3.6.1-jdk-8-alpine

# 拷贝文件到镜像
COPY . /app
WORKDIR /app
RUN mvn package

CMD ["java", "-jar", "/app/target/gs-spring-boot-0.1.0.jar"]
```

利用缓存

Docker 的构建系统中，内置了对缓存的支持，在构建时，会检查当前要构建的内容是否已经被缓存，如果被缓存则直接使用，否则重新构建，并且后续的缓存也将失效。

比如在我刚才构建镜像成功后，我再次执行刚才的命令，则可看到如下内容的输出：

复制

```
(MoeLove) ➜  complete git:(master) ✗ docker build -t local:v1 .
[+] Building 0.1s (11/11) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 96B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/debian:latest
 => [1/6] FROM docker.io/library/debian:latest
 => [internal] load build context
 => => transferring context: 6.16kB
 => CACHED [2/6] RUN apt-get update
 => CACHED [3/6] RUN apt-get install -y openjdk-8-jdk
 => CACHED [4/6] COPY . /app
 => CACHED [5/6] WORKDIR /app
 => CACHED [6/6] RUN ./mvnw package
 => exporting to image
 => => exporting layers
 => => writing image sha256:806f060ba3be3ec26326e06983ac9216981a1bc5b3cd1b1d61a4ac
 => => naming to docker.io/library/local:v1
```

其中的步骤，由于没有做任何变更，所以都还在缓存中。

对于一个正常的项目而言，源代码的更新是最为频繁的。所以看上面的 Dockerfile 你会发现 `COPY . /app` 这一行，很容易就**会让缓存失效，从而导致后面的缓存也都失效。**

所以我们对它做下改进：

复制

```
FROM maven:3.6.1-jdk-8-alpine

# 设置工作目录
WORKDIR /app

# 拷贝 pom.xml 文件到镜像
COPY pom.xml /app/pom.xml

# 保存依赖的
RUN mvn dependency:go-offline

COPY . /app

RUN mvn -e -B package

CMD ["java", "-jar", "/app/target/gs-spring-boot-0.1.0.jar"]
```

这样修改后，构建镜像时，也可缓解因为项目代码变更导致的构建缓存失效。

小结：这里的内容总体而言就是，**为了更有效的利用构建缓存，将更新最频繁的步骤放在最后面 。**

忽略无关内容

上一篇中，我们在介绍 Docker 构建系统的时候，有稍微介绍过 Docker build context，具体而言就是执行 docker build 的上下文。默认情况下，通过 context 所指定的地址中的全部文件都会被发送到 docker。

但很多时候 context 所指定的目录，可能不全是我们真正需要的内容，并且当项目增大时，通过类似 `COPY . /app` 的命令，会把所有内容拷贝至镜像中，导致最终镜像的体积变大了。

对于这种情况，我们通常使用两种方式来解决：

- 增加 `.dockerignore` 文件。将不需要的内容都写入，这样子 docker CLI 在与 Docker Daemon 交互的时候，就会把不需要的内容排除在外了。
- 继续对 Dockerfile 进行修改，只 COPY 需要的内容即可。

我们一起看下，具体到我们的示例项目中修改的结果：

复制

```
FROM maven:3.6.1-jdk-8-alpine

# 设置工作目录
WORKDIR /app

# 拷贝 pom.xml 文件到镜像
COPY pom.xml /app/pom.xml

# 保存依赖的
RUN mvn dependency:go-offline

COPY src /app

RUN mvn -e -B package

CMD  ["java", "-jar", "/app/target/gs-spring-boot-0.1.0.jar"]
```

这里需要注意的点是，如果我们直接通过 .dockerignore 设置要过滤的文件/目录时，可有效的减少 Docker Daemon 的压力。

小结

前面我以一个示例项目，为你介绍了几个 Dockerfile 的优化和实践的经验。总体来说主要是以下几个点：

- 将更新越频繁的内容，就写到 Dockerfile 越下面的位置，这是为了能更好的利用缓存（当某处缓存失效后，其后的全部缓存都会失效）；
- 尽量选择 Docker 官方镜像，一方面是因为质量比较靠谱，另一方面是官方镜像的体积都在尽可能的减小了，使用官方镜像，也有利于减小镜像的体积；
- 通过 .dockerignore 和在 Dockerfile 中指定要拷贝至镜像的具体内容，可有效的减少镜像体积。

防止包缓存过期

除去上面具体的项目的例子外，我在再为你介绍一点通用的实践经验。

上面我们已经提到了，docker build 可以利用缓存，但你有没有考虑到，如果你机器上需要构建多个不同项目的镜像，但它们的 Dockerfile 有相同的内容或者是需要安装的依赖发生变化的时候，缓存可能就不是我们想要的了。

所以，**将包管理器的缓存生成与安装包的命令写到一起可防止包缓存过期**，例如：

复制

```
FROM debian

RUN apt update && apt install -y vim
```

但是也请谨慎的使用包管理器，因为包管理器也有很多默认行为。比如 apt-get 在安装包的时候，如果不指定 `--no-install-recommends` 的话，就会默认把一些推荐的包也给装进来，造成镜像体积变大。

清理包管理器缓存

系统的包管理器，除了会有安装推荐的一些包以外，也会在镜像中保留自己的一些缓存。推荐在使用时，要在安装包结束后就立刻清理。示例如下：

复制

```
FROM debian

RUN apt update && apt install -y --no-install-recommends vim \
        && rm -rf /var/lib/apt/lists/*
```

总结

本篇，我为你介绍了 Dockerfile 的优化和一些实践经验，但这也不是全部的内容。 如果在你可选择 Docker 版本时，我推荐你选择最新的版本，并且在构建镜像的时候，开启 BuildKit。

Docker 官方文档中有一篇最佳实践的内容，推荐你也看看。

在本篇中提到的一些内容，比如缓存之类的，在后续内容，我们深入原理时，就可以把它的行为给你介绍清楚了。

现在先将本篇的内容当作一个解决方案来使用即可。

下一篇，我将为你介绍 Docker 镜像构建的原理，有了前面几篇内容的铺垫，相信你会很容易掌握。

下一章