

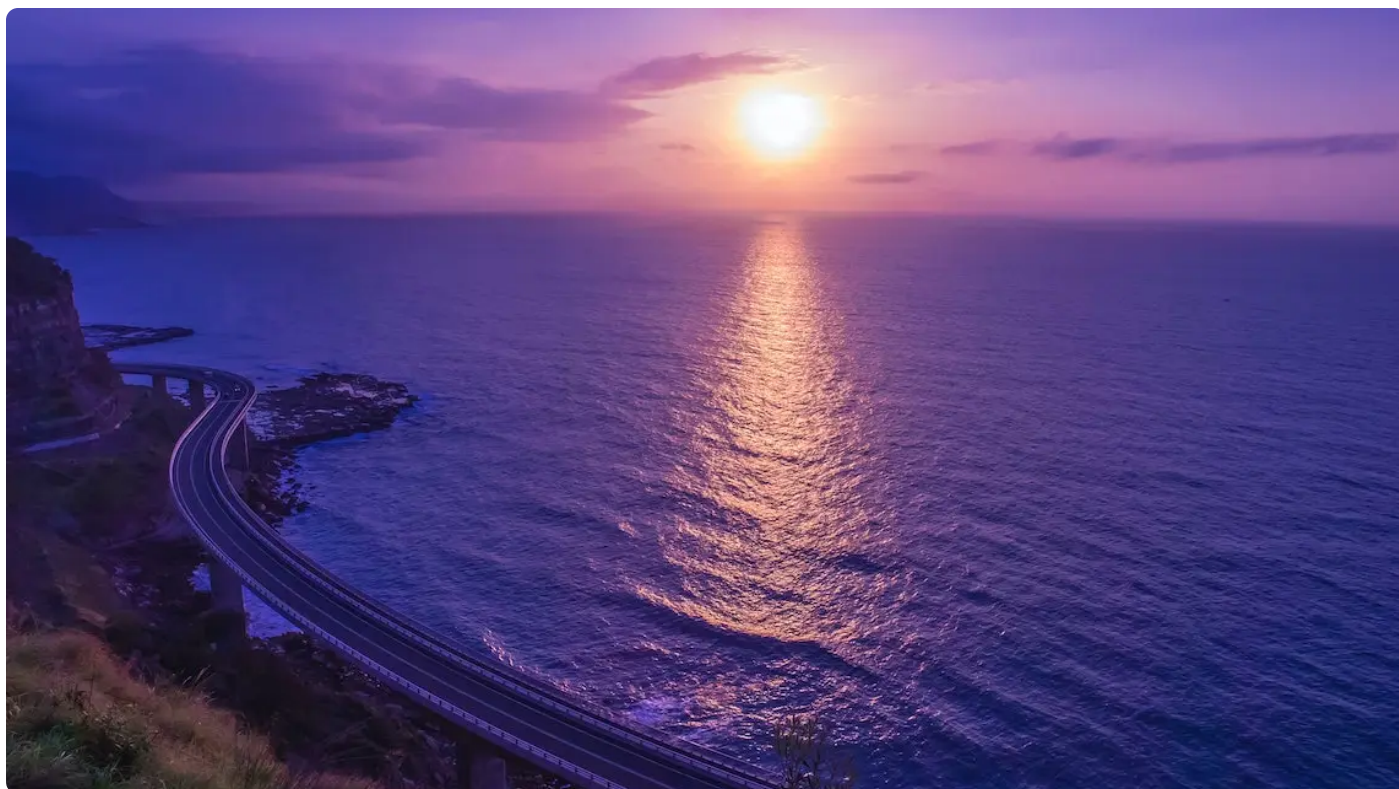
# 13 | 容器化：如何为不同语言快速构建多平台镜像？

2023-01-06 王伟 来自北京



《云原生架构与GitOps实战》

[课程介绍 >](#)



讲述：王伟

时长 17:57 大小 16.41M



你好，我是王伟。从这节课开始，我们来容器化的实践。

在 [第 1 讲](#) 中，我们以 Python Flask 应用为例，展示了如何编写 Dockerfile，如何构建和推送镜像。在编写 Dockerfile 方面，我为你介绍了构建镜像通用的套路。

但对于其他非 Python 语言编写的业务应用来说，如何快速构建镜像呢？

在这节课，我会介绍不同语言构建镜像的例子和模板，包括常用的后端语言 Java、Golang、Node.js 以及前端 Vue 框架编写的业务应用，每一种语言我都会设计一个接近真实的构建容器镜像的例子。此外，考虑构建镜像对多平台的兼容性，我还会介绍一种构建多平台镜像的方法，这样，我们构建出来的镜像就能够兼容多个不同的 CPU 平台了。未来，如果你需要对实际的业务应用进行容器化改造，完全可以参考我们这节课的内容。

在开始之前，你需要确保已经在本地安装了 Docker，并将我提前准备好的示例应用仓库克隆到本地：🔗 <https://github.com/lyzhang1999/gitops.git>。



## Java 应用容器化

我们先来看 Java 应用的容器化。

常见的 Java 应用启动方式有两种，这也就意味着镜像构建方式也有两种。一种是将应用打包成 Jar 包，在镜像内直接启动应用 Jar 包来构建镜像。另一种是在容器里通过 Spring Boot 插件直接启动应用。接下来，我分别介绍这两种镜像构建方式。

### 启动 Jar 包的构建方式

以 Spring Boot 和 Maven 为例，我已经提前创建好了一个 Demo 应用，我们以它为例子介绍如何使用 Jar 包构建镜像。

在将示例应用克隆到本地后，进入 Spring Boot Demo 目录并列出生所有文件。

📄 复制代码

```
1 $ cd gitops/docker/13/spring-boot
2 $ ls -al
3 total 80
4 drwxr-xr-x 12 weiwang staff 384 10 5 11:17 .
5 drwxr-xr-x  4 weiwang staff 128 10 5 11:17 ..
6 -rw-r--r--  1 weiwang staff   6 10 5 10:30 .dockerignore
7 -rw-r--r--  1 weiwang staff 374 10 5 11:05 Dockerfile
8 drwxr-xr-x  4 weiwang staff 128 10 5 11:17 src
9 .....
```


在这里，我们重点关注 **src 目录**、**Dockerfile 文件** 和 **.dockerignore 文件**。

首先，src 目录下的 `src/main/java/com/example/demo/DemoApplication.java` 文件的内容是 Demo 应用的主体文件，它包含一个 `/hello` 接口，使用 `Get` 请求访问后会返回 “Hello World”。

📄 复制代码

```
1
2
3 package com.example.demo;
4 import org.springframework.boot.SpringApplication;
```

```
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.bind.annotation.RestController;
9
10 @SpringBootApplication
11 @RestController
12 public class DemoApplication {
13     public static void main(String[] args) {
14         SpringApplication.run(DemoApplication.class, args);
15     }
16
17     @GetMapping("/hello")
18     public String hello(@RequestParam(value = "name", defaultValue = "World") String
19         return String.format("Hello %s!", name);
20     }
21 }
```



天下无鱼  
<https://shikey.com/>

Demo 应用的主体内容虽然很简单，但它代表了 **Spring Boot + Maven** 的典型组合，只要符合这两种技术选型，你都可以直接参考这里的例子来容器化你的业务应用。

接下来，是构建镜像的核心内容 Dockerfile 文件。

 复制代码

```
1 # syntax=docker/dockerfile:1
2
3 FROM eclipse-temurin:17-jdk-jammy as builder
4 WORKDIR /opt/app
5 COPY .mvn/ .mvn
6 COPY mvnw pom.xml ./
7 RUN ./mvnw dependency:go-offline
8 COPY ./src ./src
9 RUN ./mvnw clean install
10
11
12 FROM eclipse-temurin:17-jre-jammy
13 WORKDIR /opt/app
14 EXPOSE 8080
15 COPY --from=builder /opt/app/target/*.jar /opt/app/*.jar
16 CMD ["java", "-jar", "/opt/app/*.jar"]
```

刚开始学习 Dockerfile 的同学可能会感到疑惑，为什么这里有两个 FROM 语句呢？

实际上，这里使用了多阶段构建的方式，你可以理解为，第一个阶段的构建产物可以作为下一个阶段的输入，这里你只需要先知道这种用法就好，后面的课程我们还会有更详细的介绍。



我们来看第一阶段的构建，也就是从第 3 行到第 9 行。第 3 行 FROM 表示把 eclipse-temurin:17-jdk-jammy 作为 build 阶段的基础镜像，然后使用 WORKDIR 关键字指定了工作目录为 /opt/app，后续的文件操作都会在这个工作目录下展开。

接下来，第 5 和第 6 行通过 COPY 关键字将 .mvn 目录和 mvnw、pom.xml 文件复制到了工作目录下，第 7 行通过 RUN 关键字运行 ./mvnw dependency:go-offline 来安装依赖。然后，第 8 行将 src 目录复制到了镜像中，第 9 行使用 RUN 关键字执行 ./mvnw clean install 进行编译。

第 12 行到 16 行是第二个构建阶段。第 12 行表示使用 eclipse-temurin:17-jre-jammy 作为基础镜像，第 13 行同样指定了工作目录为 /opt/app，第 14 行的 EXPOSE 关键字之前我们有提到过，它是一个备注功能，并不是要暴露真实容器端口的意思。

第 15 行的 COPY 语句比较复杂，它指的是从 builder 阶段也就是将第一个阶段位于 /opt/app/target/ 目录下所有的 .jar 文件都拷贝到当前构建阶段镜像的 /opt/app/ 目录下。第 16 行使用 CMD 关键字定义了启动命令，也就是通过 java -jar 的方式启动应用。

最后，.dockerignore 的功能和我们熟悉的 .gitignore 文件功能类似，它指的是在构建过程中需要忽略的文件或目录，合理的文件忽略策略将有助于提高构建镜像的速度。在这个例子中，因为我们要在容器里重新编译应用，所以我们忽略了本地的 target 目录。

接下来，我们就可以使用 docker build 命令来构建镜像了。

 复制代码

```
1 $ docker build -t spring-boot .
```

当镜像构建完成后，我们要使用 docker run 命令启动镜像，并通过 --publish 暴露端口。

 复制代码

```
1 $ docker run --publish 8080:8080 spring-boot
2 .....
3 2022-10-05 03:59:48.746 INFO 1 --- [main] com.example.demo.DemoApp
```

```
4 2022-10-05 03:59:48.748 INFO 1 --- [main] com.example.demo.DemoApp
5 2022-10-05 03:59:49.643 INFO 1 --- [main] o.s.b.w.embedded.tomcat.T
6 2022-10-05 03:59:49.655 INFO 1 --- [main] o.apache.catalina.core.St
7 2022-10-05 03:59:49.656 INFO 1 --- [main] org.apache.catalina.core
8 2022-10-05 03:59:49.754 INFO 1 --- [main] o.a.c.c.C.[Tomcat].[local
9 2022-10-05 03:59:49.755 INFO 1 --- [main] w.s.c.ServletWebServerApp
10 2022-10-05 03:59:50.105 INFO 1 --- [main] o.s.b.w.embedded.tomcat.T
11 2022-10-05 03:59:50.117 INFO 1 --- [main] com.example.demo.DemoAppl
```

打开一个新的命令行终端，并使用 `curl` 访问 `hello` 接口验证返回内容。

 复制代码

```
1 $ curl localhost:8080/hello
2 Hello World!
```

如果要终止 `spring-boot` 应用，你可以回到执行 `docker run` 的命令行终端，并使用 `ctrl+c` 来停止容器。

如果你跟着我操作到了这里，说明你也已经成功以 `Jar` 包的方式将 `Spring Boot` 应用构建为 `Docker` 镜像了。

## Spring Boot 插件的构建方式

除了使用 `Jar` 包，`Spring Boot` 应用还可以通过 `./mvnw spring-boot:run` 的方式启动，这意味着我们也可以把它作为镜像的启动命令。

我还是以 `Spring Boot` 示例应用为例，我在示例应用 `Dockerfile` 文件同级目录下已经提前准备好了 `Dockerfile-Boot` 文件，下面是该文件的内容。

 复制代码

```
1 # syntax=docker/dockerfile:1
2
3 FROM eclipse-temurin:17-jdk-jammy
4
5 WORKDIR /app
6
7 COPY .mvn/ .mvn
8 COPY mvnw pom.xml ./
9 RUN ./mvnw dependency:resolve
10
11 COPY src ./src
```

相比较 Jar 的启动方式，Spring Boot 插件的启动方式显得更加简单。在构建过程中，我们实际上还用了一个小技巧：第 7 和第 8 行代表单独复制了依赖清单文件 pom.xml 而不是复制整个根目录，目的是在依赖不变的情况下充分利用 Docker 构建缓存。

在这个 Dockerfile 文件中有两条关键的命令，一个 mvnw dependency:resolve 用于安装依赖，另一个 mvnw spring-boot:run 命令用来启动应用。

接下来，我们使用 docker build 命令构建镜像，这里要注意增加 -f 参数指定新的 Dockerfile 文件。

[复制代码](#)

```
1 $ docker build -t spring-boot . -f Dockerfile-Boot
```

镜像构建成功后，使用 docker run 命令启动镜像。

[复制代码](#)

```
1 $ docker run --publish 8080:8080 spring-boot
```

最后，你可以尝试用 curl 访问 localhost:8080/hello 接口，将会得到 Hello World 返回结果。

Spring Boot 插件的启动方式虽然比较简单，但它将构建过程延迟到了启动阶段，并且依赖镜像的 JDK 工具，对于生产环境来说这些都不是必要的。如果你通过 docker images 命令仔细对比两次构建镜像占用的空间大小，你会发现，第一种方式构建生成的镜像大概在 280M 左右，而第二种构建方式生成的镜像在 500M 左右。在实际的生产环境中，我更推荐你使用第一种方式来构建 Java 镜像。

## Golang 应用容器化

下面我们继续来看 Go 应用的容器化。

以 Echo 框架为例，我提前编写好了一个简单的示例应用。在将示例应用克隆到本地后，你可以进入 docker/13/golang 目录并查看。



```
1 $ cd gitops/docker/13/golang
2 $ ls -al
3 -rw-r--r-- 1 weiwang staff 292 10 5 14:16 Dockerfile
4 -rw-r--r-- 1 weiwang staff 599 10 5 14:12 go.mod
5 -rw-r--r-- 1 weiwang staff 2825 10 5 14:12 go.sum
6 -rw-r--r-- 1 weiwang staff 235 10 5 14:13 main.go
```



main.go 文件是应用的主体文件，包含一个 /hello 接口，通过 Get 方法请求后，将返回 Hello World 字符串。

```
1 package main
2
3 import (
4     "net/http"
5     "github.com/labstack/echo/v4"
6 )
7
8 func main() {
9     e := echo.New()
10    e.GET("/hello", func(c echo.Context) error {
11        return c.String(http.StatusOK, "Hello World Golang")
12    })
13    e.Logger.Fatal(e.Start(":8080"))
14 }
```

接下来，我们来看 Dockerfile 的内容。

```
1 # syntax=docker/dockerfile:1
2 FROM golang:1.17 as builder
3 WORKDIR /opt/app
4 COPY . .
5 RUN go build -o example
6
7 FROM ubuntu:latest
8 WORKDIR /opt/app
9 COPY --from=builder /opt/app/example /opt/app/example
10 EXPOSE 8080
11 CMD ["/opt/app/example"]
```

同样地，这个 **Dockerfile** 包含了两个构建阶段，第一个构建阶段是以 **golang:1.17** 为基础镜像，然后我们执行 **go build** 命令编译并输出可执行文件，将其命名为 **example**。



第二个构建阶段是以 **ubuntu:latest** 为基础镜像，第 9 行通过 **COPY** 关键字将第一个阶段构建的 **example** 可执行文件复制到镜像的 **/opt/app/** 目录下，最后，使用 **CMD** 来运行 **example** 启动应用。

现在，我们可以通过 **docker build** 来构建镜像。

复制代码

```
1 $ docker build -t golang .
```

接下来，使用 **docker run** 来启动镜像。

复制代码

```
1 $ docker run --publish 8080:8080 golang
2      ____  _
3     / __/___/ /   ___/
4    / _// __/ _  \ / _ \
5   /___/\___/\___/\___/ v4.9.0
6   High performance, minimalist Go web framework
7   https://echo.labstack.com
8   _____0/_____
9                                     0\
10  => http server started on [::]:8080
```

如果你还没有终止之前运行的 **Spring Boot** 示例，在运行 **Golang** 示例时，你可能会得到“**Bind for 0.0.0.0:8080 failed: port is already allocated**”的错误。你可以通过 **docker ps** 命令来查看 **Spring Demo** 的容器 ID，并通过 **docker stop [Container ID]** 来终止它。这时候再运行 **Golang** 示例就能够正常启动了。

现在，你可以使用 **curl** 命令来访问 **localhost:8080/hello** 接口，查看是否返回了预期的 **Hello World Golang** 字符串。

## Node.js 应用容器化



以 Express.js 框架为例，我已经提前编写好了一个简单示例，在将示例应用克隆到本地后，你可以进入 `docker/13/node` 目录并查看。



复制代码

```
1 $ cd gitops/docker/13/node
2 $ ls -al
3 -rw-r--r--    1 weiwang  staff    12 10   5 16:45 .dockerignore
4 -rw-r--r--    1 weiwang  staff   589 10   5 16:39 Dockerfile
5 -rw-r--r--    1 weiwang  staff   230 10   5 16:44 app.js
6 drwxr-xr-x   60 weiwang  staff  1920 10   5 16:26 node_modules
7 -rw-r--r--    1 weiwang  staff 39326 10   5 16:26 package-lock.json
8 -rw-r--r--    1 weiwang  staff   251 10   5 16:26 package.json
```

`app.js` 是示例应用的主体文件，包含一个 `/hello` 接口。当我们通过 `Get` 请求访问时，会返回“Hello World Node.js”字符串。

复制代码

```
1 const express = require('express')
2 const app = express()
3 const port = 3000
4
5 app.get('/hello', (req, res) => {
6   res.send('Hello World Node.js')
7 })
8
9 app.listen(port, () => {
10   console.log(`Example app listening on port ${port}`)
11 })
```

`.dockerignore` 是构建镜像时的忽略文件，在这个例子中，忽略了 `node_modules` 目录。

复制代码

```
1 $ cat .dockerignore
2 node_modules
```

接着我们来看一下 `Dockerfile` 文件的内容。

复制代码

```
1 # syntax=docker/dockerfile:1
2 FROM node:latest AS build
```

```
3 RUN sed -i "s@http://\|(deb|security\).debian.org@https://mirrors.aliyun.com@g"
4 RUN apt-get update && apt-get install -y dumb-init
5 WORKDIR /usr/src/app
6 COPY package*.json ./
7 RUN npm ci --only=production
8
9
10 FROM node:16.17.0-bullseye-slim
11 ENV NODE_ENV production
12 COPY --from=build /usr/bin/dumb-init /usr/bin/dumb-init
13 USER node
14 WORKDIR /usr/src/app
15 COPY --chown=node:node --from=build /usr/src/app/node_modules /usr/src/app/node
16 COPY --chown=node:node . /usr/src/app
17 CMD ["dumb-init", "node", "app.js"]
```



这是一个由两个阶段组成的镜像构建方法。第一个阶段使用 `node:latest` 作为 `build` 阶段的基础镜像，同时安装了 `dumb-init` 组件。此外，这种构建方法还将 `package.json` 和 `package-lock.json` 复制到镜像内，并通过 `npm ci --only=production` 命令安装依赖。

从第 10 行开始是第二个构建阶段，这里使用了 `node:16.17.0-bullseye-slim` 作为基础镜像，此外，我们还为 `Express` 配置了 `NODE_ENV=production` 的环境变量，代表在生产环境中使用。这会改变 `Express.js` 框架的默认配置，如日志等级、缓存处理策略等。然后，我们还要将 `build` 阶段安装的 `dumb-init` 组件、依赖以及源码复制到第二个阶段的镜像中，修改源码和依赖目录的用户组。最后，通过 `CMD` 命令使用 `node` 启动 `app.js`。

在将 `NodeJS` 容器化的过程中，有一个需要特别注意的细节，由于 `NodeJS` 并不是设计以 `PID=1` 的进程运行的，所以常规的启动方式并不能让 `NodeJS` 程序在容器内接收到 `Kill` 信号，这会导致 `Node` 进程不能被优雅终止（例如更新时突然中断），所以我们可以通过 `dumb-init` 组件来启动 `Node` 进程。

现在，我们可以通过 `docker build` 来构建镜像。

复制代码

```
1 $ docker build -t nodejs .
```

接下来，使用 `docker run` 来启动镜像。

```
1 $ docker run --publish 3000:3000 nodejs
2 Example app listening on port 3000
```

[复制代码](#)

天下无鱼

<https://shikey.com/>

进行到这里，你可以使用 `curl` 命令来访问 `localhost:3000/hello` 接口，查看是否返回了预期的 `Hello World Node.js` 字符串。

## Vue 应用容器化

常见的 Vue 应用容器化方案有两种，第一种是将 `http-server` 组件作为代理服务器来构建镜像，第二种是让 `Nginx` 作为代理服务器来构建镜像。接下来，我会分别介绍这两种镜像构建方式。

### Http-server 构建方式

先看 `http-server` 的构建方式。以 `Vue` 框架为例，我已经提前将项目进行了初始化，接下来你需要将示例应用克隆到本地，然后进入 `docker/13/vue/example` 目录并查看。

```
1 $ cd gitops/docker/13/vue/example
2 $ ls -al
3 -rw-r--r--  1 weiwang  staff    12 10  5 17:26 .dockerignore
4 -rw-r--r--  1 weiwang  staff   172 10  5 17:27 Dockerfile
5 -rw-r--r--  1 weiwang  staff     0 10  5 17:34 Dockerfile-Nginx
6 -rw-r--r--  1 weiwang  staff   631 10  5 17:23 README.md
7 -rw-r--r--  1 weiwang  staff   337 10  5 17:23 index.html
8 .....
```

[复制代码](#)

在这个例子中，`.dockerignore` 文件的内容和 `Node.js` 应用一样，都是忽略 `node_modules` 目录，以便加速镜像的构建速度。

接下来，我们重点关注 `Dockerfile` 文件内容。

```
1 # syntax=docker/dockerfile:1
2
3 FROM node:lts-alpine
4 RUN npm install -g http-server
5 WORKDIR /app
6 COPY package*.json ./
```

[复制代码](#)

```
7 RUN npm install
8 COPY . .
9 RUN npm run build
10
11 EXPOSE 8080
12 CMD [ "http-server", "dist" ]
```



简单分析一下上面 **Dockerfile** 的内容。首先使用 **node:lts-alpine** 作为基础镜像，然后安装 **http-server** 作为代理服务器，第 6 行代表的含义是，将 **package.json** 和 **package-lock.json** 复制到镜像内，并使用 **npm install** 安装依赖。这里让依赖安装和源码安装解耦的目的是尽量使用 **Docker** 镜像构建缓存，只要在 **package.json** 文件内容不变的情况下，即便是源码改变，都可以使用已经下载好的 **npm** 依赖缓存。

依赖安装完毕后，第 8 行，我们要将项目源码复制到镜像内，并且通过 **npm run build** 来构建 **dist** 目录，最后，第 12 行，使用 **http-server** 来启动 **dist** 目录的静态文件。

现在，我们可以通过 **docker build** 来构建镜像了。

```
1 $ docker build -t vue .
```

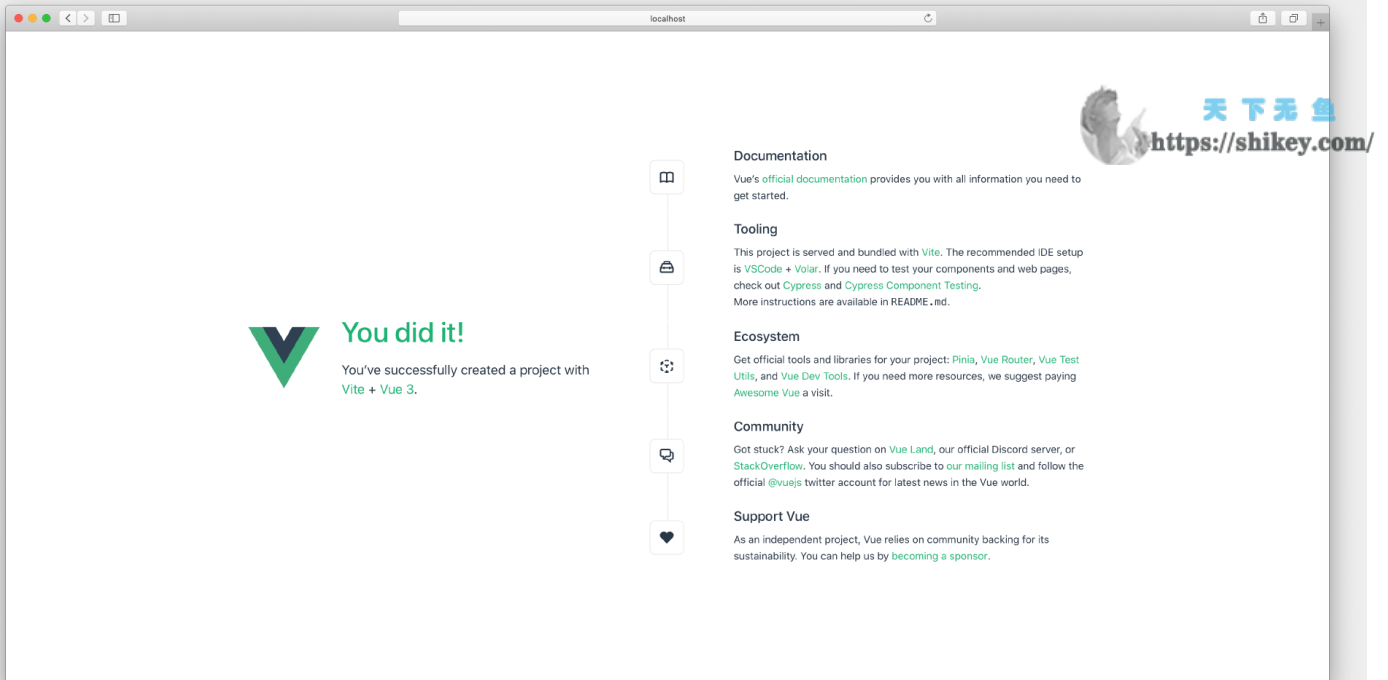
 复制代码

接下来，使用 **docker run** 启动镜像。

```
1 $ docker run --publish 8080:8080 vue
2 Starting up http-server, serving dist
3
4 http-server version: 14.1.1
5
6 http-server settings:
7 CORS: disabled
8 .....
```

 复制代码

到这里，你可以打开浏览器访问 <http://localhost:8080>，如果出现 **Vue** 示例应用的项目，说明镜像构建完成，如下图所示。



## Nginx 构建方式

在上面的例子中，我们使用 `http-server` 来对外提供服务，这在开发和测试场景，或者是在小型的使用场景中是完全可以的。不过，在正式的生产环境中，我推荐你把 **Nginx** 作为反向代理服务器来对外提供服务，它也是性能最好、使用最广泛和稳定性最高的一种方案。

在 **Vue** 示例项目的同级目录下，我已经创建好了名为 `Dockerfile-Nginx` 文件。

复制代码

```
1 # syntax=docker/dockerfile:1
2
3 FROM node:lts-alpine as build-stage
4 WORKDIR /app
5 COPY package*.json ./
6 RUN npm install
7 COPY . .
8 RUN npm run build
9
10 FROM nginx:stable-alpine as production-stage
11 COPY --from=build-stage /app/dist /usr/share/nginx/html
12 EXPOSE 80
13 CMD ["nginx", "-g", "daemon off;"]
```

这个 **Dockerfile** 定义了两个构建阶段，第一个阶段是第 3 行到第 8 行的内容，其他的是第二阶段的内容。

第一阶段的构建过程和我们在上面提到的 `http-server` 的构建方式非常类似，它是以 `node:its-alpine` 为基础镜像，同时复制 `package.json` 和 `package-lock.json` 并安装依赖，然后再复制项目源码并且执行 `npm run build` 来构建项目，生成 `dist` 目录。



第二个阶段的构建过程则是引入了一个新的 `nginx:stable-alpine` 镜像作为运行镜像，还将第一阶段构建的 `dist` 目录复制到了第二阶段的 `/usr/share/nginx/html` 目录中。这个目录是 Nginx 默认的网页目录，默认情况下，Nginx 将使用该目录的内容作为静态资源。最后第 13 行以前台的方式启动 Nginx。

现在，我们可以通过 `docker build` 来构建镜像。

复制代码

```
1 $ docker build -t vue-nginx -f Dockerfile-Nginx .
```

接下来，使用 `docker run` 启动镜像。

复制代码

```
1 $ docker run --publish 8080:80 vue-nginx
2 /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perf
3 /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
4 /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-defa
5 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.
6 10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/con
7 /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.
8 .....
```

最后，打开浏览器访问 <http://localhost:8080> 验证一下，如果出现和前面提到的 `http-server` 构建方式一样的 Vue 示例应用界面，就说明镜像构建成功了。

## 构建多平台镜像

好了，上面的案例，我们都是通过在本地执行 `docker build` 命令来构建镜像，然后在本地通过 `docker run` 命令来执行的。实际上，在构建镜像时，Docker 会默认构建本机对应平台的镜像，例如常见的 AMD64 平台，这在大多数情况是适用的。

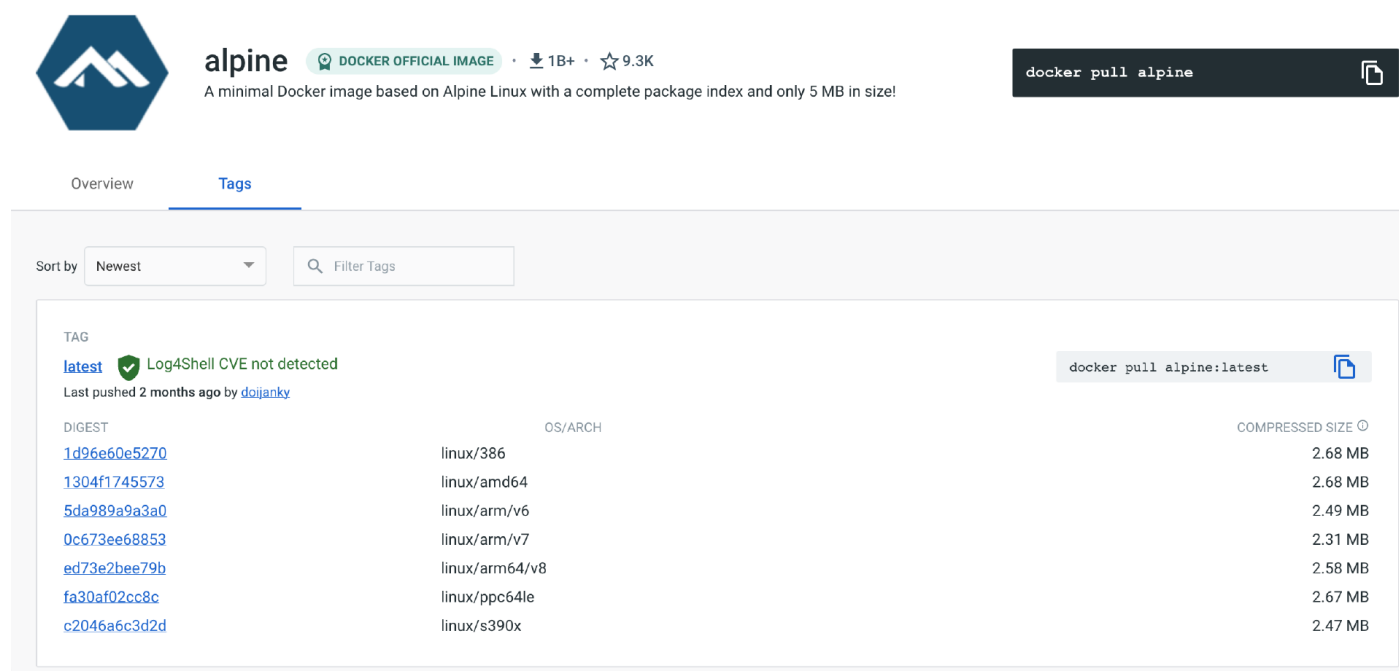
但是，当我们使用不同平台的设备尝试启动这个镜像时，可能会遇到下面的问题。

```
1 WARNING: The requested image's platform (linux/arm64/v8) does not match the det
```



产生这个问题的原因是，构建和运行设备的 CPU 平台存在差异。在实际项目中，最典型的例子是构建镜像的计算机是 AMD64 架构，但运行镜像的机器是 ARM64。

要查看镜像适用于什么平台，你可以找到 DockerHub 镜像详情页。例如，Alpine 镜像适用的平台就可以在这个 [链接](#) 查看，详情页截图如下。



TAG	DIGEST	OS/ARCH	COMPRESSED SIZE
latest			
1d96e60e5270	linux/386	2.68 MB	
1304f1745573	linux/amd64	2.68 MB	
5da989a9a3a0	linux/arm/v6	2.49 MB	
0c673ee68853	linux/arm/v7	2.31 MB	
ed73e2bee79b	linux/arm64/v8	2.58 MB	
fa30af02cc8c	linux/ppc64le	2.67 MB	
c2046a6c3d2d	linux/s390x	2.47 MB	

从这个页面我们可以看出，Alpine 镜像适用的平台非常多，例如 Linux/386、Linux/amd64 等等。一般情况下，在构建镜像时，我们只会构建本机平台的镜像，但是当拉取镜像时，Docker 会自动拉取符合当前平台的镜像版本。

那么，怎么才能真正实现跨平台的“一次构建，到处运行”目标呢？Docker 为我们提供了构建多平台镜像的方法：buildx。

## 初始化

要使用 Buildx，首先需要创建构建器，你可以使用 `docker buildx create` 命令来创建它，并将其命名为 mybuilder。



```
1 $ $ docker buildx create --name builder
2 builder
```



然后，将 **mybuilder** 设置为默认的构建器。

```
1 $ docker buildx use builder
```

复制代码

接下来，初始化构建器，这一步主要是启动 **buildkit** 容器。

复制代码

```
1 $ $ docker buildx inspect --bootstrap
2 [+] Building 19.1s (1/1) FINISHED
3 => [internal] booting buildkit
4 => => pulling image moby/buildkit:buildx-stable-1
5 => => creating container buildx_buildkit_mybuilder0
6 Name:      builder
7 Driver:    docker-container
8
9 Nodes:
10 Name:      mybuilder0
11 Endpoint:  unix:///var/run/docker.sock
12 Status:    running
13 Buildkit:  v0.10.4
14 Platforms: linux/amd64, linux/amd64/v2, linux/amd64/v3, linux/arm64, linux/risc
```

初始化完成后，我们可以从返回结果中看到支持的平台，例如 **Linux/amd64**、**Linux/arm64** 等。

## 构建多平台镜像

这时候，我们就可以尝试使用 **buildx** 来构建多平台镜像了。我已经提前编写好了一个简单示例，在将示例应用克隆到本地后，你可以进入 **docker/13/multi-arch** 目录并查看。

复制代码

```
1 $ cd gitops/docker/13/multi-arch
2 $ ls -al
3 -rw-r--r--  1 weiwang  staff   439  10   5 23:49 Dockerfile
4 -rw-r--r--  1 weiwang  staff  1075  10   5 18:34 go.mod
5 -rw-r--r--  1 weiwang  staff   696  10   5 18:34 go.sum
```

main.go 是示例应用的主体文件，我们启动一个 HTTP 服务器，访问根路径可以返回 Runtime 包的一些内置变量。

[复制代码](#)

```

1 package main
2 import (
3     "net/http"
4     "runtime"
5     "github.com/gin-gonic/gin"
6 )
7 var (
8     r = gin.Default()
9 )
10 func main() {
11     r.GET("/", indexHandler)
12     r.Run(":8080")
13 }
14 func indexHandler(c *gin.Context) {
15     var osinfo = map[string]string{
16         "arch":    runtime.GOARCH,
17         "os":      runtime.GOOS,
18         "version": runtime.Version(),
19     }
20     c.JSON(http.StatusOK, osinfo)
21 }

```

相比较单一平台的构建方法，在构建多平台镜像的时候，我们可以在 Dockerfile 内使用一些内置变量，例如 BUILDPLATFORM、TARGETOS 和 TARGETARCH，他们分别对应构建平台（例如 Linux/amd64）、系统（例如 Linux）和架构（例如 AMD64）。

[复制代码](#)

```

1 # syntax=docker/dockerfile:1
2 FROM --platform=$BUILDPLATFORM golang:1.18 as build
3 ARG TARGETOS TARGETARCH
4 WORKDIR /opt/app
5 COPY go.* ./
6 RUN go mod download
7 COPY . .
8 RUN --mount=type=cache,target=/root/.cache/go-build \
9 GOOS=$TARGETOS GOARCH=$TARGETARCH go build -o /opt/app/example .
10
11 FROM ubuntu:latest
12 WORKDIR /opt/app

```

```
13 COPY --from=build /opt/app/example ./example
14 CMD ["/opt/app/example"]
```



天下无鱼

<https://shikey.com/>

这个 Dockerfile 包含两个构建阶段，第一个构建阶段是从第 2 行至第 9 行，第二个构建阶段是从第 11 行到第 14 行。

我们先看第一个构建阶段。

第 2 行 FROM 基础镜像增加了一个 `--platform=$BUILDPLATFORM` 参数，它代表“强制使用不同平台的基础镜像”，例如 `Linux/amd64`。在没有该参数配置的情况下，Docker 默认会使用构建平台（本机）对应架构的基础镜像。

第 3 行 ARG 声明了使用两个内置变量 TARGETOS 和 TARGETARCH，TARGETOS 代表系统，例如 `Linux`，TARGETARCH 则代表平台，例如 `Amd64`。这两个参数将会在 Golang 交叉编译时生成对应平台的二进制文件。

第 4 行 WORKDIR 声明了工作目录。

第 5 行的意思是通过 COPY 将 `go.mod` 和 `go.sum` 拷贝到镜像中，并在第 6 行使用 RUN 来运行 `go mod download` 下载依赖。这样，在这两个文件不变的前提下，Docker 将使用构建缓存来加快构建速度。

在下载完依赖之后，我们通过第 7 行把所有源码文件复制到镜像内。

第 8 行有两个含义，首先，`--mount=type=cache,target=/root/.cache/go-build` 的目的是告诉 Docker 使用 Golang 构建缓存，加快镜像构建的速度。接下来，`GOOS=TARGETOSGOARCH=TARGETARCH go build -o /opt/app/example .` 代表的含义是 Golang 交叉编译。注意，TARGETOS 和 TARGETARCH 是我们提到的内置变量，在具体构建镜像的时候，Docker 会帮我们填充进去。

第二个构建阶段比较简单，主要是使用 `ubuntu:latest` 基础镜像，将第一个构建阶段生成的二进制文件复制到镜像内，然后指定镜像的启动命令。

接下来，我们就可以开始构建多平台镜像了。

在开始构建之前，先执行 `docker login` 登录到 DockerHub。



```
1 $ docker login
2 Username:
3 Password:
4 Login Succeeded
```

接下来，使用 `docker buildx build` 一次性构建多平台镜像。


复制代码

```
1 $ docker buildx build --platform linux/amd64,linux/arm64 -t lyzhang1999/multi-a
```

在这个命令中，我们使用 `--platform` 参数指定了两个平台：Linux/amd64 和 Linux/arm64，同时 `-t` 参数指定了镜像的 Tag，而 `--push` 参数则代表构建完成后直接将镜像推送到 DockerHub。

还记得我们在 Dockerfile 第 2 行增加的 `--platform=$BUILDPLATFORM` 参数吗？当执行这条命令时，Docker 会分别使用 Amd64 和 Arm64 两个平台的 `golang:1.18` 镜像，并且在对应的镜像内执行编译过程。

执行完命令后，镜像会上传到 DockerHub 平台。进入这个镜像详情页我们就会发现它同时兼容了 Amd64 和 Arm64 两个平台。这样，多平台镜像就构建完成了。

 lyzhang1999 / multi-arch

Description

*This repository does not have a description*

Last pushed: 3 hours ago

Docker commands



To push a new tag to this repository,

`docker push lyzhang1999/multi-arch:tagname`

Tags and scans

VULNERABILITY SCANNING - DISABLED

This repository contains 3 tag(s).

Tag	OS	Pulled	Pushed
latest		3 hours ago	3 hours ago
arm64	Operating system: linux Architecture: amd64, arm64	---	2 days ago
amd64		---	2 days ago

[See all](#)

[Go to Advanced Image Management](#)


Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions.

[Upgrade](#) [Learn more](#)

## 总结

在这节课，我为你介绍了主流语言镜像构建的案例，包括后端语言 Golang、Java、Node.js 以及前端 Vue 框架。 <https://shikey.com/>

在这些案例中，我尽量按照真实的生产环境来编写 Dockerfile，我使用到了一些 Dockerfile 的高级用法，例如多阶段构建、使用缓存和使用 .dockerignore 等，这些用法可以帮助我们加速构建镜像和缩小镜像大小。当你需要将实际的业务进行容器化改造时，可以直接参考我编写的案例。

此外，我还介绍了如何使用 buildx 构建多平台镜像。在这一部分，我通过一个真实的例子介绍了如何构建 Golang 的多平台镜像。一般情况下，多平台镜像并不常用，但如果你构建的镜像需要兼容不同的 CPU 平台，那就可以通过这种方法来实现。

这节课我还提到了“多阶段构建”，但并没有对它做深入讲解，这是我们下节课的重点。

## 思考题


最后，给你留一道思考题吧。

结合相关资料，请你简单分享一下为什么我们能构建非本地平台的镜像呢？（提示：Docker QEMU。）

欢迎你给我留言交流讨论，你也可以把这节课分享给更多的朋友一起阅读。我们下节课见。

分享给需要的人，Ta 购买本课程，你将得 18 元

 生成海报并分享

 赞 5  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言 (11)



争光 Alan

2023-01-10 来自广东

多平台构建在dockerfile写架构信息感觉很麻烦

我们的做法是from 的基础镜像做成多架构，然后dockerfile正常写即可，构建的时候指定架构就行了

作者回复: 也是一种不错的实践方式。



1



徐曙辉

2023-01-29 来自湖南

为什么需要两次构建，只要第一次构建可以吗？



Adam

2023-01-18 来自广东

老师，请教个问题，除了docker buildx这种工具外，还有没有其它工具也可以用来进行多平台构建。感觉docker in docker的方式不太灵活，kaniko又满足不了。



郑海成

2023-01-18 来自北京

思考题：查了一些关于qemu的原理，尝试理解一下。docker buildx 通过qemu的用户模式（binfmt\_misc）注册一个转换程序动态转换不同cpu架构之间的命令🤔

作者回复: 很易懂的原理解释。



GAC·DU

2023-01-09 来自广东

老师，执行./mvnw dependency:go-offline一直卡着不动是什么情况？

作者回复: 应该是网络问题, 可以尝试参考第16讲的内容, 用 github action 来构建, 这样就可以解决网络问题了。



天下无鱼

<https://shikey.com/>



不瘦二十斤  
不改头像

**jeffery**

2023-01-08 来自广东

buildx 需要安装

<https://github.com/docker/buildx#linux-packages>

```
mkdir -pv ~/.docker/cli-plugins/
```

```
wget -O ~/.docker/cli-plugins/docker-buildx \
```

```
https://github.com/docker/buildx/releases/download/v0.9.1/buildx-v0.9.1.linux-amd64
```

```
chmod a+x ~/.docker/cli-plugins/docker-buildx
```

然后命令

```
$ docker buildx create --name builder 才能生效
```

作者回复: macOS 和 Windows Docker desktop 是包含 buildx 的, Linux 需要按照你的方法手动安装。



不瘦二十斤  
不改头像

**jeffery**

2023-01-08 来自广东

```
1. git clone https://github.com/lyzhang1999/gitops.git
```

```
fatal: unable to access 'https://github.com/.../.g
```

解决方案:

```
~# git config --global http.proxy
```

```
~# git config --unset http.proxy
```

```
2. https://github.com/lyzhang1999/gitops/blob/main/docker/13/golang/Dockerfile#L6
```

之前添加goproxy

```
RUN go env -w GO111MODULE=on
```

```
RUN go env -w GOPROXY=https://goproxy.cn,direct
```

遇到错误:

```
Step 6/14 : RUN go mod download
```

```
---> Running in bd45f51f1f37
```

```
go mod download: github.com/labstack/echo/v4@v4.10.0: Get "https://proxy.golang.org/github.com/labstack/echo/v4/@v/v4.10.0.info": dial tcp 172.217.163.49:443: connect: connection refused
```



...

The command '/bin/sh -c go mod download' returned a non-zero code: 1



作者回复: 看起来是代理或者网络的问题, 配置 Go proxy 能解决吗?

共 3 条评论 >



一步

2023-01-07 来自广东

最后多平台构建

FROM ubuntu 镜像的时候 也要加上 --platform=\$BUILDPLATFORM 参数



êwě n

2023-01-06 来自广东

请问下, ubuntu 这个镜像是能支持不同架构的吧?

作者回复: 支持的, 常见的 arm64, amd64, ppc64le 等都支持。



无名无姓

2023-01-06 来自广东

多平台构建一般都是什么样的环境下面才使用呢

作者回复: 有异构的 K8s 集群就可能会需要, 例如运行在 ARM64 平台的 K8s。



橙汁

2023-01-06 来自北京

原来nodejs应用启动后默认pid不是1, 用了1年多都没注意 一会赶紧看看

