

15 | 容器化：如何选择最适合业务的基础镜像？

2023-01-11 王炜 来自北京



天下无鱼

<https://shikey.com/>

《云原生架构与GitOps实战》

课程介绍 >



讲述：王炜

时长 09:44 大小 8.90M



你好，我是王炜。

在上一节课，我们介绍了缩小镜像体积的有效方法，其中包括更换基础镜像和多阶段构建。在编写多阶段构建的 `Dockerfile` 时，我推荐你使用 `ubuntu:latest` 镜像作为第二阶段的基础镜像，不推荐你在生产环境下使用 `Alpine` 或者 `Scratch` 镜像。

这是为什么呢？

归根结底，**Docker** 镜像包含了运行环境和业务应用。我们在镜像构建过程中，需要从程序依赖、可调试（开发）、安全性和镜像大小方面综合考虑。由于镜像大小的优先级并不是最高的，因此我们首先要保证的是程序的运行及其依赖的完整性，没必要一味地追求大小的极限。

但是，在业务容器化的过程中，基础镜像的数量非常多，每一种编程语言的实践都有所不同，我们又该如何选择基础镜像呢？如何避免选择错误的镜像埋下隐患？

这节课，我们就延伸一下上节课的内容，看看镜像选择的实践。

通用镜像



我们知道，镜像并不是从头开始构建的，我们会基于一个基础镜像来构建，这个基础镜像包含编译工具和运行环境，它负责构建和运行我们的业务代码。

例如，在上节课多阶段构建的第二个阶段，我们用了 `ubuntu:latest` 为 `Golang` 应用提供运行环境，也就是在 `Dockerfile` 中通过 `FROM` 关键字引用 `ubuntu:latest` 为基础镜像。

复制代码

```
1 FROM ubuntu:latest
```

要注意的是，在使用 `FROM` 引用镜像时，除非给定完整的镜像地址，否则 `Docker` 默认会从 [官网](#) 下载镜像，你也可以在这个网站里搜索到你所需要的基础镜像。

现在，我们回忆一下 [第 14 讲](#) 提到的多阶段构建 `Dockerfile` 的内容。

复制代码

```
1 FROM golang:1.17 as builder
2 # 第一阶段
3
4 FROM ubuntu:latest
5 # 第二节阶段
```

你有没有想过，为什么我们在第一个阶段使用 `golang:1.17` 而不是 `ubuntu:latest` 呢？这是因为 `golang:1.17` 包含 `Golang` 的构建工具，这是 `ubuntu:latest` 所没有的。为了不需要手动在 `Dockerfile` 中安装 `Golang`，我们引用了语言的专用镜像。

为了更好地帮助你理解镜像的类别，**我将镜像类型归纳为了通用镜像和专用镜像**。在这个例子中，`ubuntu:latest` 是一种通用镜像，`golang:1.17` 则是一种专用镜像。

我们先说通用镜像。

你可以把通用镜像理解成 Linux 发行版，他们是一个全新安装的 Linux 系统，除了系统工具以外，不包含任何特定语言的编译和运行环境。一般来说，通用镜像有下面两种使用场景。



- 1. 使用通用镜像构建符合业务特定要求的镜像，例如特定的工具链和依赖，特定的构建环境 和安全工具等。
- 2. 单纯作为业务的运行环境，常见于多阶段构建。

像第一种场景，我们就可以使用 ubuntu:latest 作为基础镜像，然后手动安装 Golang 编译环境。这时 Dockerfile 可以像下面这么写。

复制代码

```
1 # syntax=docker/dockerfile:1
2 FROM ubuntu:latest
3 COPY . .
4 RUN apt-get update && apt install -y golang-go
5 RUN go build
```

当然，除了 ubuntu:latest 以外，你还可以选择其他的通用镜像。我给你画了一张表格，列出在工作中常用的通用镜像版本及其压缩后的大小（以 Linux amd64 平台为例）。

镜像名称	大小
ubuntu:latest	29.2M
alpine:latest	2.68M
centos:latest	79.65M
debian:latest	52.5M
debian:bullseye-slim	29.96M

极客时间

在生产环境下，综合镜像的通用性、可调式、安全和镜像大小等各方面因素，我最推荐你使用这三个通用镜像：ubuntu:latest、debian:slim 和 alpine:latest。其中，前两个镜像在本质上其

实是同一个类型，而 **Alpine** 则是一个特殊的 Linux 发行版，接下来我们对这两个类型的镜像做简单介绍。



Ubuntu/Debian

Ubuntu 和 Debian 是综合能力非常强的 Linux 发行版，非常适合作为通用镜像使用，它们主要的优点如下。

- 支持的软件包众多。
- 镜像体积较小。
- 用户数量大，社区活跃，容易及时发现和修复安全问题。
- 相比较 Alpine 具有更通用的 C 语言标准库 glibc。
- 文档和教程丰富。

当然，它们也有一定的缺点，由于内置了更多的系统级工具，所以相比较 Alpine 更容易受到安全攻击。不过，如果你比较关注调试的便利性和通用性，那么它们仍然是生产环境下的首选镜像。

Alpine

我们再来看另一种通用镜像 Alpine。在很长的时间里，Alpine 发行版并没有受到太多的关注。直到 Docker 时代，大家为了追求更小的镜像体积才开始大量使用 Alpine 镜像。相比较 Debian，Alpine 有下面这些优点。

- 快速的包安装体验。
- 极小的镜像体积。
- 只包含少量的系统级程序，安全性更高。
- 更轻量的初始化系统 OpenRC。

相比较普通的 Linux 发行版，Alpine 也有非常明显的差异，其中最大的差异在于它使用 musl libc 而不是标准的 glibc，对编译型语言例如 C 和 Golang，这可能会导致一些编译方面的问题，需要额外注意。

专用镜像

专用镜像提供了特定语言的编译和运行环境，绝大多数语言都有 Docker 官方维护的专用镜像。在实际工作中，专用镜像一般有以下两种使用场景。



<https://shikey.com/>

- 作为解释型语言的运行镜像使用，例如 `python:latest`、`php:8.1-fpm-buster` 等。
- 作为编译型语言多阶段构建中编译阶段的基础镜像使用，例如 `golang:latest`。

下面我们来看看不同语言如何选择专用镜像。

Golang

对于编译型语言，我推荐你使用多阶段构建的方法将构建环境和运行镜像区分开，包括 C 语言。当构建 Golang 业务镜像时，我们就需要选择合适的第一阶段镜像（构建镜像）和第二阶段镜像（运行镜像）。

在 [第 13 讲](#)，我介绍了如何通过多阶段构建来打包 Docker 镜像，我使用了 `golang:1.17` 作为构建镜像，`ubuntu:latest` 作为运行镜像。

 复制代码

```
1 FROM golang:1.17 as builder
2 # 第一阶段
3 .....
4 RUN go build -o example
5
6 FROM ubuntu:latest
7 # 第二阶段
8 .....
9 COPY --from=builder /opt/app/example /opt/app/example
```

在这个例子中，我们最终构建的镜像大小约为 76M。

实际上，在选择第一阶段和第二阶段的镜像时，还有其他的组合。例如，你还可以使用 Alpine 镜像。

 复制代码

```
1 FROM golang:alpine
2 # 第一阶段
```

```
3 RUN go build -o example
4
5 FROM alpine:latest
6 # 第二阶段
```



这种组合最终构建的镜像大小为 12.4M。

还记得我在上节课留的思考题吗？我提到了如果你尝试删除 `CGO_ENABLED=0` 之后，下面这种镜像组合将会导致 **Golang** 程序无法运行。

复制代码

```
1 FROM golang:1.17
2 # 第一阶段
3 RUN go build -o example
4
5 FROM alpine:latest
6 # 第二阶段
```

当我们运行镜像时，会得到错误信息。

复制代码

```
1 exec /opt/app/example: no such file or directory
```

这是因为 **golang:1.17** 镜像的 C 语言标准库是 **glibc**，而 **alpine:latest** 使用的是 **musl**。在第一阶段 **Golang** 的构建过程中，虽然我们编译生成了二进制可执行文件，但这个二进制文件并不是“纯静态”的。在默认条件下，**CGO** 将被开启，这意味着当程序内包含底层是由 C 语言库实现的功能时，二进制可执行文件仍然需要依赖外部的动态链接库。所以，在第二阶段的 **Alpine** 镜像中，自然找不到第一阶段的外部动态链接库，这就会导致抛出“no such file or directory”的异常。

此时，我们有两种方法解决这个问题。

- 禁用 **CGO**，也就是在多阶段构建的编译过程中指定 `CGO_ENABLED=0`，达到“纯静态”编译的效果，然后，你就可以使用任意的通用镜像来运行了。
- 在第一阶段和第二阶段同时使用 **Alpine** 版本的镜像，例如第一阶段使用 **golang:alpine**，第二阶段使用 **alpine:latest** 镜像。

不过，Alpine musl 和 glibc 还是有一些差异的，你可以通过 [这个链接](#)来进一步学习。

对于初学者而言，在不熟悉这些差异的情况下，我并不推荐你为了缩小镜像尺寸而盲目使用 Alpine 镜像。

Java

对于 Java 来说，我同样推荐你使用多阶段构建来将编译和运行镜像区分开。和 Golang 不同的是，Java 程序需要 JVM 的支持才能运行。所以，在编译阶段，我们需要 JDK 工具提供完整的编译环境；而在运行阶段，我们只需要 JRE 即可。

下面我列出了几种镜像组合，你可以根据实际情况进行选择，镜像大小以 Linux/amd64 为例。

多阶段构建编译阶段	多阶段构建运行阶段	备注
eclipse-temurin:11-jdk(229.84M)	eclipse-temurin:11-jre(85.23M)	可选择其他 Java 版本
eclipse-temurin:11-jdk-alpine(198.66M)	eclipse-temurin:11-jre-alpine(55.12M)	可选择其他 Java 版本
openjdk:11-jdk(317.68M)	openjdk:11.0-jre(116.79M)	官方已弃用，谨慎选择



其他解释型语言

常见的解释型语言有 Python、Node、PHP 和 Ruby，和编译型语言不同，它不需要编译阶段，所以多阶段构建自然也就失去了价值。那么，我们要如何选择解释型语言的镜像呢？

在一些情况下，如果你确信你的业务程序不依赖于外部的 C 代码库，那么你可以考虑使用专用镜像的 Alpine 版本作为基础镜像，例如 python:alpine 镜像，它只有 18.15M。但是，在大多数情况下，我们是很难确定的。所以，对于解释型语言，我并不推荐你使用专用镜像的 Alpine 版本作为基础镜像。

实际上，除了 Alpine 版本以外，大多数专用镜像都会提供 Slim 版本，Slim 版本大都基于 Ubuntu、Debian 或 CentOS 等标准 Linux 发行版构建，并且删除了一些不必要的系统应用，体积也相对较小，非常适合作为首选镜像。下面我列出了不同语言专用镜像的 Slim 版本及其 Linux/amd64 版本的大小，供你参考。

镜像	大小
python:slim	45.71M
node:lts-slim	62.29M
ruby:slim	70.45M
php:8.1-fpm-buster	135.11M
python:slim-bullseye	45.71M
rust:slim-bullseye	229.38M



总结

好了，这节课就讲到这里。这节课，我为你总结了两类镜像，分别是通用镜像和专用镜像。对于通用镜像，我总结了两种使用场景，第一种是基于通用镜像构建适合业务的专属镜像，另一种场景是在多阶段构建，单纯将其作为编译型语言的运行镜像来使用。

对于通用镜像的选择并没有一套黄金规则。不过，对于初学者而言，我建议你选择标准的 Linux 发行版，例如 Debian、Ubuntu 和 CentOS。Alpine 镜像虽然足够小，但如果你不熟悉它的内部环境，可能会带来一些程序编译和运行上的问题，所以我不推荐你在生产环境使用它。

而对于特定语言的专用镜像呢，首先你需要根据自己的业务情况区分编译型语言和解释型语言，然后再在这个基础上选择专用镜像。对于编译型语言，我推荐你使用多阶段构建把编译和运行镜像区分开，不过，在使用多阶段构建时，你需要特别注意保持编译和运行阶段对应镜像 C 语言标准库的一致性，否则可能导致编译后的二进制文件无法运行。

对于解释型语言，虽然在做镜像选择时会更容易，但一般情况下我们都会基于专用镜像安装其他的一些工具，配置起来也并不容易。在大多数情况下，选择一个具有丰富安装包资源的专用

镜像可以减少工作量，比如选择基于 **Debian** 构建的专用镜像。在我为你列出的专用镜像中，**php:8.1-fpm-buster** 和 **python:slim-bullseye** 都是基于 **Debian** 构建的专用镜像。



总的来说，基础镜像的选择并没有一套固定规则，你需要从多方面，例如业务程序的依赖、可调试性、安全性、体积大小和社区维护等角度来考虑。


思考题

最后，给你留一道思考题吧。

在构建业务镜像的过程中，你有哪些额外的经验可以和我们分享呢？

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 3  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。

上一篇 14 | 容器化：如何将镜像体积缩减 90%？

下一篇 16 | 自动构建：如何使用 GitHub Action 构建镜像？

精选留言 (5)

 写留言



Jich

2023-01-13 来自上海

老师，有一个问题，多阶段构建，在对应多环境的话，是不是会存在发布效率上的问题。我们现在是在**DEV**环境编译完之后，直接可以用这个镜像发布**test**或生产。如果使用多阶段构建的话，是不是意味着我在每个环境都会要执行编译的阶段。

作者回复: 使用多阶段构建不影响原有的镜像发布过程, 只是可以帮助你减小镜像大小, 构建完成之后仍然走老的流程。



天下无鱼

<https://shikey.com/>



不瘦二十斤
不改头像

jeffery

2023-01-11 来自陕西

优化镜像大小

FROM 适合业务基础镜像

减少镜像层

清理无用数据

使用.dockerignore忽略构建镜像时非必需的文件

多阶段构建

作者回复: 很完整的总结~



Wisdom

2023-01-11 来自中国香港

总结的挺好的, 不难理解。老师可以出品快一点吗? :)

作者回复: 感谢你的认可, 固定每周一三五更新, 敬请期待~



includestdio.h

2023-01-11 来自陕西

以前整理的关于 dockerfile 的最佳实践:

1. 不安装无效软件包
2. 理想状态下, 每个镜像应该只有一个进程
3. 无法避免多进程运行时, 应选择合适的初始化进程【1.需要捕获 SIGTERM 信号并完成子进程的优雅终止; 2.负责清理退出的子进程以避免僵尸进程 --init 参数】
4. 最小层级数: 1.多条 RUN 命令可通过连接符连接成一条指令集以减少层数; 2.通过多段构建减少镜像层数
5. 多行参数按字母排序, 可以减少可能出现的重复参数, 并且提供可读性
6. 把变更频率低的编译指令放在镜像底层, 有效利用 build cache【考虑 build cache 下层失效, 上层同步失效的机制】
7. 复制文件时, 每个文件应独立复制, 确保某个文件变更时, 只影响该文件对应的缓存

作者回复: 很棒的总结 , 例如提到的多阶段构建和复制文件都是在实际使用场景用的很多的。



👍 1



天下无鱼

<https://shikey.com/>



一步

2023-01-11 来自广东

了解每一种镜像的优缺点和使用场景 才能更好的进行选择

