

如何使用 docker buildx 构建跨平台 Go 镜像

📅 2021.9.2 📁 Golang ✎ 4706 ⌚ 10 分钟 👁 5605

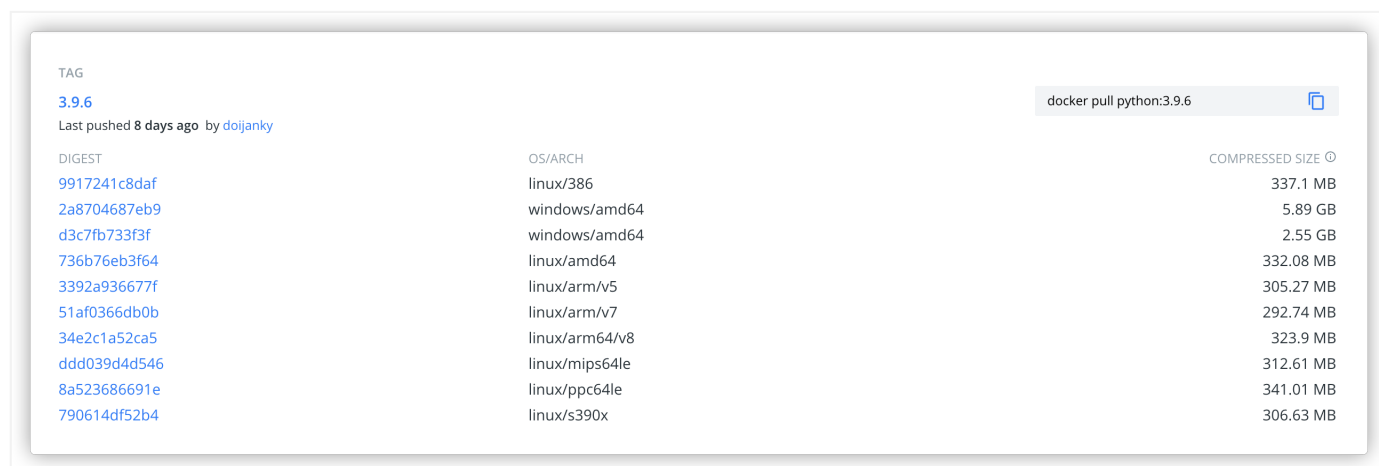
目录

1. 前提
2. docker buildx
 - 2.1 启用 Buildx
 - 2.2 builder 实例
 - 2.3 构建驱动
 - 2.4 如何解决 ERROR merging manifest list
3. buildx 的跨平台构建策略
4. 一次构建多个架构 Go 镜像实践
 - 4.1 源代码和 Dockerfile
 - 4.2 执行跨平台构建
 - 4.3 验证构建结果
5. 如何交叉编译 Golang 的 CGO 项目
 - 5.1 准备交叉编译环境和依赖
 - 5.2 交叉编译 CGO 示例
6. 总结
7. 参考链接

随着国产化和信创的推进，为应用适配多个操作系统和处理器架构的需求越来越普遍。常见做法是为不同平台单独构建一个版本，当用来开发的平台与部署的目标平台不同时，实现这一目标并不容易。例如在 x86 架构上开发一个应用程序并将其部署到 ARM 平台的机器上，通常需要准备 ARM 平台的基础设施用于开发和编译。

前提

大部分镜像托管平台支持多平台镜像，这意味着镜像仓库中单个标签可以包含不同平台的多个镜像，以 docker hub 的 python 镜像仓库为例，3.9.6 这个标签就包含了 10 个不同系统和架构的镜像（平台 = 系统 + 架构）：



The screenshot shows the Docker Hub interface for the python:3.9.6 image. It displays a table of manifests for different OS/ARCH combinations. The table has three columns: TAG, DIGEST, and OS/ARCH. The TAG column shows the version 3.9.6 and the user dojanky. The DIGEST column lists the unique identifiers for each manifest. The OS/ARCH column lists the operating system and architecture for each manifest. The COMPRESSED SIZE column shows the size of each manifest in MB.

TAG	DIGEST	OS/ARCH	COMPRESSED SIZE
3.9.6	9917241c8daf	linux/386	337.1 MB
	2a8704687eb9	windows/amd64	5.89 GB
	d3c7fb733f3f	windows/amd64	2.55 GB
	736b76eb3f64	linux/amd64	332.08 MB
	3392a936677f	linux/arm/v5	305.27 MB
	51af0366db0b	linux/arm/v7	292.74 MB
	34e2c1a52ca5	linux/arm64/v8	323.9 MB
	ddd039d4d546	linux/mips64le	312.61 MB
	8a523686691e	linux/ppc64le	341.01 MB
	790614df52b4	linux/s390x	306.63 MB

通过 docker pull 或 docker run 拉取一个支持跨平台的镜像时，docker 会自动选择与当前运行平台相匹配的镜像。由于该特性的存在，在进行镜像的跨平台分发时，我们不需要对镜像的消费做任何处理，只需要关心镜像的生产，即如何构建跨平台的镜像。

docker buildx

默认的 docker build 命令无法完成跨平台构建任务，我们需要为 docker 命令行安装 buildx 插件扩展其功能。buildx 能够使用由 Moby BuildKit 提供的构建镜像额外特性，它能够创建多个 builder 实例，在多个节点并行地执行构建任务，以及跨平台构建。

启用 Buildx

macOS 或 Windows 系统的 Docker Desktop，以及 Linux 发行版通过 deb 或者 rpm 包所安装的 docker 内置了 buildx，不需要另行安装。

如果你的 docker 没有 buildx 命令，可以下载二进制包进行安装：

1. 首先从 [Docker buildx](#) 项目的 release 页面找到适合自己平台的二进制文件。
2. 下载二进制文件到本地并重命名为 docker-buildx，移动到 docker 的插件目录 `~/.docker/cli-plugins`。
3. 向二进制文件授予可执行权限。

Shall We Code?

```
1 $ DOCKER_BUILDKIT=1 docker build --platform=local -o . "https://github.com/docker/bui
2 $ mkdir -p ~/.docker/cli-plugins
3 $ mv buildx ~/.docker/cli-plugins/docker-buildx
```

使用 buildx 进行构建的方法如下：

```
1 docker buildx build .
```

buildx 和 docker build 命令的使用体验基本一致，还支持 build 常用的选项如 -t、-f 等。

builder 实例

docker buildx 通过 builder 实例对象来管理构建配置和节点，命令行将构建任务发送至 builder 实例，再由 builder 指派给符合条件的节点执行。我们可以基于同一个 docker 服务程序创建多个 builder 实例，提供给不同的项目使用以隔离各个项目的配置，也可以为一组远程 docker 节点创建一个 builder 实例组成构建阵列，并在不同阵列之间快速切换。

使用 docker buildx create 命令可以创建 builder 实例，这将以当前使用的 docker 服务为节点创建一个新的 builder 实例。要使用一个远程节点，可以在创建示例时通过 DOCKER_HOST 环境变量指定远程端口或提前切换到远程节点的 docker context。

下面首先以当前节点创建一个新的 builder 实例，并通过命令行选项指定实例名称、驱动以及当前节点的目标平台：

```
$ docker buildx create --driver docker-container --platform linux/amd64 --name multi-builder
multi-builder
```

创建 docker-container 驱动实例时可通过 --driver-opt image=moby/buildkit:v0.10.5 选项配置所使用的 buildkit 镜像版本。

实例创建之后可以添加新的节点，通过 docker buildx create 命令的 --append 选项可将 --node <node> 节点加入到 --name <builder> 选项指定的 builder 实例。如下将把一个远程节点加入 builder 实例：

```
1 $ export DOCKER_HOST=tcp://10.10.150.66:2375
2 $ docker buildx create --name multi-builder --append --node remote-builder
```

刚创建的 builder 处于 inactive 状态，可以在 create 或 inspect 子命令中添加 --bootstrap 选项立即启动实例（可验证节点是否可用）：

Shall We Code?

```
3  => [remote-builder internal] booting buildkit
4  => => pulling image moby/buildkit:buildx-stable-1
5  => => creating container buildx_buildkit_remote-builder
6  => [multi-builder0 internal] booting buildkit
7  => => pulling image moby/buildkit:buildx-stable-1
8  => => creating container buildx_buildkit_multi-builder0
9  Name:      multi-builder
10 Driver:    docker-container
11
12 Nodes:
13 Name:      multi-builder0
14 Endpoint:  unix:///var/run/docker.sock
15 Status:    running
16 Buildkit:  v0.10.5
17 Platforms: linux/amd64*, linux/386
18
19 Name:      remote-builder
20 Endpoint:  tcp://10.10.88.20:2375
21 Status:    running
22 Buildkit:  v0.10.5
23 Platforms: linux/arm64
```

`docker buildx ls` 将列出所有可用的 builder 实例和实例中的节点:

```
1 $ docker buildx ls
2 NAME/NODE          DRIVER/ENDPOINT          STATUS   PLATFORMS
3 multi-builder      docker-container
4 multi-builder0     unix:///var/run/docker.sock running v0.10.5 linux/amd64*, linux/
5 remote-builder     tcp://10.10.88.20:2375   running v0.10.5 linux/arm64
6 default *         docker
7 default           default                  running linux/amd64, linux/386
```

如上就创建了一个支持多平台架构的 builder 实例，执行 `docker buildx use <builder>` 将切换到所指定的 builder 实例。

`docker buildx inspect`、`docker buildx stop` 和 `docker buildx rm` 命令用于管理一个实例的生命周期。

构建驱动

buildx 实例通过两种方式来执行构建任务，两种执行方式被称为使用不同的「驱动」：

- `docker` 驱动：使用 Docker 服务程序中集成的 BuildKit 库执行构建。
- `docker-container` 驱动：启动一个包含 BuildKit 的容器并在容器中执行构建。

Shall We Code?

到 docker 的镜像目录（通常是 `/var/lib/overlay2`），之后执行 `docker images` 命令可以列出所输出的镜像；而 `docker container` 则需要通过 `--output` 选项指定输出格式为镜像或其他格式。

为了一次性构建多个平台的镜像，本文使用 `docker container` 作为默认的 builder 实例驱动。

如何解决 ERROR merging manifest list

如果你使用了较新版本的 `buildx(>=0.10)` 和 `buildkit(>=0.11)` 以及较旧版本的镜像仓库，在构建跨平台镜像时可能会遇到构建成功但推送失败的问题，报错如下：

```
⇒ ⇒ pushing layers
⇒ ⇒ pushing manifest for registry.xxx.com/xxx/postgres-vb
⇒ [auth] xxx/postgres-vb:pull,push token for registry.xxx.com
⇒ ERROR merging manifest list registry.xxx.com/xxx/postgres-vb:v1.2.0
```

这是因为新版本默认开启了 `Build attestations` 功能以增强供应链安全，但同时带来了兼容性问题。如果你并不需要，可添加如下 `build` 选项禁用它以解决该问题：

```
docker buildx build --sbom=false --provenance=false
```

buildx 的跨平台构建策略

根据构建节点和目标程序语言不同，`buildx` 支持以下三种跨平台构建策略：

1. 通过 QEMU 的用户态模式创建轻量级的虚拟机，在虚拟机系统中构建镜像。
2. 在一个 builder 实例中加入多个不同目标平台的节点，通过原生节点构建对应平台镜像。
3. 分阶段构建并且交叉编译到不同的目标架构。

QEMU 通常用于模拟完整的操作系统，它还可以通过用户态模式运行：以 `binfmt_misc` 在宿主机系统中注册一个二进制转换处理程序，并在程序运行时动态翻译二进制文件，根据需要将系统调用从目标 CPU 架构转换为当前系统的 CPU 架构。最终的效果就像在一个虚拟机中运行目标 CPU 架构的二进制文件。Docker Desktop 内置了 QEMU 支持，其他满足运行要求的平台可通过以下方式安装：

```
1 $ docker run --privileged --rm tonistiigi/binfmt --install all
```

这种方式不需要对已有的 `Dockerfile` 做任何修改，实现的成本很低，但显而易见效率并不高。

将不同系统架构的原生节点添加到 builder 实例中可以为跨平台编译带来更好的支持，而且效率更高，但需要有足够的基础设施支持。

Shall We Code?

复制到目标架构的单一镜像中。下文会使用 GO 实现一个具体的示例。这种方式不需要额外的硬件，也能得到较好的性能，但只有特定编程语言能够实现。

一次构建多个架构 Go 镜像实践

源代码和 Dockerfile

下面将以一个简单的 Go 项目作为示例，假设示例程序文件 `main.go` 内容如下：

```
1 package main
2
3 import (
4     "fmt"
5     "runtime"
6 )
7
8 func main() {
9     fmt.Println("Hello world!")
10    fmt.Printf("Running in [%s] architecture.\n", runtime.GOARCH)
11 }
```

定义构建过程的 Dockerfile 如下：

```
1 FROM --platform=$BUILDPLATFORM golang:1.14 as builder
2
3 ARG TARGETARCH
4
5 WORKDIR /app
6 COPY main.go /app/main.go
7 RUN GOOS=linux GOARCH=$TARGETARCH go build -a -o output/main main.go
8
9 FROM alpine:latest
10 WORKDIR /root
11 COPY --from=builder /app/output/main .
12 CMD /root/main
```

构建过程分为两个阶段：

- 在一阶段中，我们将拉取一个和当前构建节点相同平台的 `golang` 镜像，并使用 Go 的交叉编译特性将其编译为目标架构的二进制文件。
- 然后拉取目标平台的 `alpine` 镜像，并将上一阶段的编译结果拷贝到镜像中。

执行跨平台构建

如果没有设置 `--platform` 标志，就会以目标平台拉取基础镜像，最终生成的镜像也将属于目标平台。此外 `Dockerfile` 中可通过 `BUILDPLATFORM`、`TARGETPLATFORM`、`BUILDARCH` 和 `TARGETARCH` 等参数使用该选项的值。当使用 `docker-container` 驱动时，这个选项可以接受用逗号分隔的多个值作为输入以同时指定多个目标平台，所有平台的构建结果将合并为一个整体的镜像列表作为输出，因此无法直接输出为本地的 `docker images` 镜像。

`docker buildx build` 支持丰富的输出行为，通过 `--output=[PATH,-,type=TYPE[,KEY=VALUE]]` 选项可以指定构建结果的输出类型和路径等，常用的输出类型有以下几种：

- `local`：构建结果将以文件系统格式写入 `dest` 指定的本地路径，如 `--output type=local,dest=./output`。
- `tar`：构建结果将在打包后写入 `dest` 指定的本地路径。
- `oci`：构建结果以 OCI 标准镜像格式写入 `dest` 指定的本地路径。
- `docker`：构建结果以 Docker 标准镜像格式写入 `dest` 指定的本地路径或加载到 `docker` 的镜像库中。同时指定多个目标平台时无法使用该选项。
- `image`：以镜像或者镜像列表输出，并支持 `push=true` 选项直接推送到远程仓库，同时指定多个目标平台时可使用该选项。
- `registry`：`type=image,push=true` 的精简表示。

对本示例我们执行如下 `docker buildx build` 命令：

```
1 $ docker buildx build --platform linux/amd64,linux/arm64,linux/arm -t registry.cn-har
```

该命令将在当前目录同时构建 `linux/amd64`、`linux/arm64` 和 `linux/arm` 三种平台的镜像，并将输出结果直接推送到远程的阿里云镜像仓库中。

构建过程可拆解如下：

1. `docker` 将构建上下文传输给 `builder` 实例。
2. `builder` 为命令行 `--platform` 选项指定的每一个目标平台构建镜像，包括拉取基础镜像和执行构建步骤。
3. 导出构建结果，镜像文件层被推送到远程仓库。
4. 生成一个清单 JSON 文件，并将其作为镜像标签推送给远程仓库。

验证构建结果

运行结束后可以通过 `docker buildx imagetools` 探查已推送到远程仓库的镜像：

```
1 $ docker buildx imagetools inspect registry.cn-hangzhou.aliyuncs.com/waynerv/arch-demo:latest
2 Name:          registry.cn-hangzhou.aliyuncs.com/waynerv/arch-demo:latest
3 MediaType: application/vnd.docker.distribution.manifest.list.v2+json
```



```
7   Name:      registry.cn-hangzhou.aliyuncs.com/waynerv/arch-demo:latest@sha256:cb6a7
8   MediaType: application/vnd.docker.distribution.manifest.v2+json
9   Platform:  linux/amd64
10
11  Name:      registry.cn-hangzhou.aliyuncs.com/waynerv/arch-demo:latest@sha256:034aa
12  MediaType: application/vnd.docker.distribution.manifest.v2+json
13  Platform:  linux/arm64
14
15  Name:      registry.cn-hangzhou.aliyuncs.com/waynerv/arch-demo:latest@sha256:db0ee
16  MediaType: application/vnd.docker.distribution.manifest.v2+json
17  Platform:  linux/arm/v7
```

最后在不同的平台以 `latest` 标签拉取并运行镜像，验证构建结果是否正确。使用 Docker Desktop 时，其本身集成的虚拟化功能可以运行不同平台的镜像，可以直接以 `sha256` 值拉取镜像：

```
1 $ docker run --rm registry.cn-hangzhou.aliyuncs.com/waynerv/arch-demo:latest@sha256:c
2 Hello world!
3 Running in [amd64] architecture.
```

```
1 $ docker run --rm registry.cn-hangzhou.aliyuncs.com/waynerv/arch-demo:latest@sha256:0
2 Hello world!
3 Running in [arm64] architecture.
```

如何交叉编译 Golang 的 CGO 项目

支持交叉编译到常见的操作系统和 CPU 架构是 Golang 的一大优势，但以上示例中的解决方案只适用于纯 Go 代码，如果项目中通过 `cgo` 调用了 C 代码，情况会变得更加复杂。

准备交叉编译环境和依赖

为了能够顺利编译 C 代码到目标平台，首先需要在编译环境中安装目标平台的 C 交叉编译器（通常基于 `gcc`），常用的 Linux 发行版会提供大部分平台的交叉编译器安装包，可以直接通过包管理器安装。

其次还需要安装目标平台的 C 标准库（通常标准库会作为交叉编译器的安装依赖，不需要单独安装），另外取决于你所调用的 C 代码的依赖关系，可能还需要安装一些额外的 C 依赖库（如 `libopus-dev` 之类）。

我们将使用 `amd64` 架构的 `golang:1.14` 官方镜像作为基础镜像执行编译，其使用的 Linux 发行版为 Debian。假设交叉编译的目标平台是 `linux/arm64`，则需要准备的交叉编译器为 `gcc-aarch64-`

Shall We Code?

```
1 $ apt-get update
2 $ apt-get install gcc-aarch64-linux-gnu
```

libc6-dev-arm64-cross 会同时被安装。

得益于 Debian 包管理器 dpkg 提供的多架构安装能力，假如我们的代码依赖 libopus-dev 等非标准库，可通过 <library>:<architecture> 的方式安装其 arm64 架构的安装包：

```
1 $ dpkg --add-architecture arm64
2 $ apt-get update
3 $ apt-get install -y libopus-dev:arm64
```

交叉编译 CGO 示例

假设有如下 cgo 的示例代码：

```
1 package main
2
3 /*
4 #include <stdlib.h>
5 */
6 import "C"
7 import "fmt"
8
9 func Random() int {
10     return int(C.random())
11 }
12
13 func Seed(i int) {
14     C.srandom(C.uint(i))
15 }
16
17 func main() {
18     rand := Random()
19     fmt.Printf("Hello %d\n", rand)
20 }
```

将使用的 Dockerfile 如下：

```
1 FROM --platform=$BUILDPLATFORM golang:1.14 as builder
2
3 ARG TARGETARCH
4 RUN apt-get update && apt-get install -y gcc-aarch64-linux-gnu
5
6 WORKDIR /app
```

```
10 CGO_ENABLED=1 GOOS=linux GOARCH=$TARGETARCH CC=$CC CC_FOR_TARGET=$CC_FOR_TARGET go
```

Dockerfile 中通过 `apt-get` 安装了 `gcc-aarch64-linux-gnu` 作为交叉编译器，示例程序较为简单因此不需要额外的依赖库。在执行 `go build` 进行编译时，需要通过 `CC` 和 `CC_FOR_TARGET` 环境变量指定所使用的交叉编译器。

为了基于同一份 Dockerfile 执行多个目标平台的编译（假设目标架构只有 `amd64 / arm64`），最下方的 `RUN` 指令使用了一个小技巧，通过 Bash 的条件判断语法来执行不同的编译命令：

- 假如构建任务的目标平台是 `arm64`，则指定 `CC` 和 `CC_FOR_TARGET` 环境变量为已安装的交叉编译器（注意它们的值有所不同）。
- 假如构建任务的目标平台是 `amd64`，则不指定交叉编译器相关的变量，此时将使用默认的 `gcc` 作为编译器。

最后使用 `buildx` 执行构建的命令如下：

```
1 $ docker buildx build --platform linux/amd64,linux/arm64 -t registry.cn-hangzhou.aliy
```

总结

有了 `Buildx` 插件的帮助，在缺少基础设施的情况下，我们也能使用 `docker` 方便地构建跨平台的应用镜像。

但默认通过 `QEMU` 虚拟化目标平台指令的方式有明显地性能瓶颈，如果编写应用的语言支持交叉编译，我们可以通过结合 `buildx` 和交叉编译获得更高的效率。

本文最后介绍了一种进阶场景的解决方案：如何对使用了 `CGO` 的 `Golang` 项目进行交叉编译，并给出了编译到 `linux/arm64` 平台的示例。

参考链接

- [Leverage multi-CPU architecture support](#)
- [Docker Buildx](#)
- [docker buildx build](#)
- [C? Go? Cgo! - go.dev](#)
- [Cross-Compiling Golang \(CGO\) Projects](#)

链接: <https://waynerv.com/posts/building-multi-architecture-images-with-docker-buildx/>

许可: CC BY-NC-SA 4.0

updated 2023-06-06



相关文章:



容器技术原理(一): 从根本上认识容器镜像

Kubernetes 故障排查指南

如何优化 docker 镜像体积

为 Kubernetes 集群启用 Pod 安全策略

💎 云原生 💎 Docker

< 制作开箱即用的 Ubuntu qcow2 镜像

如何手动安装 Nginx 二进制文件 >

1 个表情




1 条评论 - 由 *giscus* 提供支持

最早

最新

Shall We Code?

© 2020–2023  Waynerv
Powered by Hugo | Theme is MemE
CC BY-NC-SA 4.0