



下载APP



特别放送 | Go Modules依赖包管理全讲

2021-09-09 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 28:11 大小 25.82M



你好，我是孔令飞。今天我们更新一期特别放送作为加餐。

在 Go 项目开发中，依赖包管理是一个非常重要的内容，依赖包处理不好，就会导致编译失败。而且 Go 的依赖包管理有一定的复杂度，所以，我们有必要系统学习下 Go 的依赖包管理工具。

这一讲，我会首先介绍下 Go 依赖包管理工具的历史，并详细介绍下目前官方推荐的依赖包管理方案 Go Modules。Go Modules 主要包括了 go mod 命令行工具、模块下载机制，以及两个核心文件 go.mod 和 go.sum。另外，Go Modules 也提供了一些环境变量，用来控制 Go Modules 的行为。这一讲，我会分别介绍下这些内容。

在正式开始讲解这些内容之前，我们先来对 Go Modules 有个基本的了解。

Go Modules 简介

Go Modules 是 Go 官方推出的一个 Go 包管理方案，基于 vgo 演进而来，具有下面这几个特性：

可以使包的管理更加简单。

支持版本管理。

允许同一个模块多个版本共存。

可以校验依赖包的哈希值，确保包的一致性，增加安全性。

内置在几乎所有的 go 命令中，包括 `go get`、`go build`、`go install`、`go run`、`go test`、`go list` 等命令。

具有 Global Caching 特性，不同项目的相同模块版本，只会在服务器上缓存一份。

在 Go1.14 版本以及之后的版本，Go 官方建议在生产环境中使用 Go Modules。因此，以后的 Go 包管理方案会逐渐统一到 Go Modules。与 Go Modules 相关的概念很多，我在这里把它们总结为“6-2-2-1-1”，这一讲后面还会详细介绍每个概念。

六个环境变量：`GO111MODULE`、`GOPROXY`、`GONOPROXY`、`GOSUMDB`、`GONOSUMDB`、`GOPRIVATE`。

两个概念：Go module proxy 和 Go checksum database。

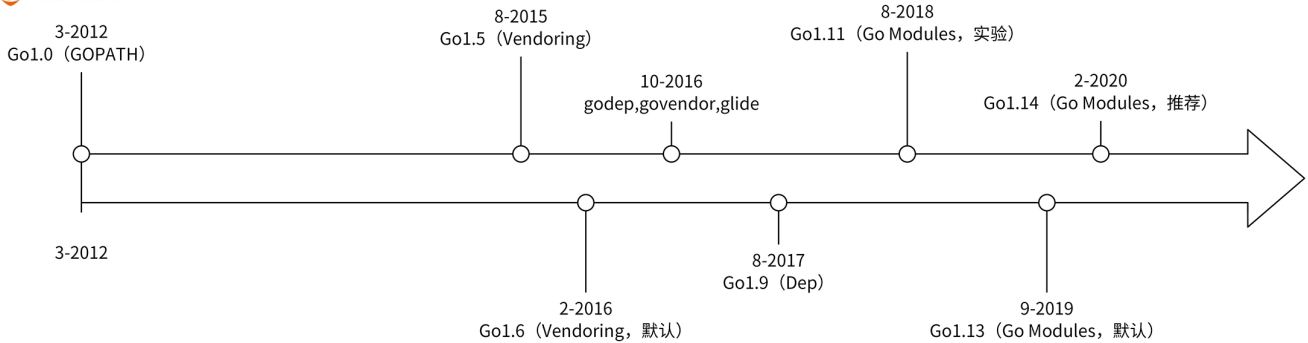
两个主要文件：`go.mod` 和 `go.sum`。

一个主要管理命令：`go mod`。

一个 build flag。

Go 包管理的历史

在具体讲解 Go Modules 之前，我们先看一下 Go 包管理的历史。从 Go 推出之后，因为没有统一的官方方案，所以出现了很多种 Go 包管理方案，比较混乱，也没有彻底解决 Go 包管理的一些问题。Go 包管理的历史如下图所示：



这张图展示了 Go 依赖包管理工具经历的几个发展阶段，接下来我会按时间顺序重点介绍下其中的五个阶段。

Go1.5 版本前：GOPATH

在 Go1.5 版本之前，没有版本控制，所有的依赖包都放在 GOPATH 下。采用这种方式，无法实现包的多版本管理，并且包的位置只能局限在 GOPATH 目录下。如果 A 项目和 B 项目用到了同一个 Go 包的不同版本，这时候只能给每个项目设置一个 GOPATH，将对应版本的包放在各自的 GOPATH 目录下，切换项目目录时也需要切换 GOPATH。这些都增加了开发和实现的复杂度。

Go1.5 版本：Vendoring

Go1.5 推出了 vendor 机制，并在 Go1.6 中默认启用。在这个机制中，每个项目的根目录都可以有一个 vendor 目录，里面存放了该项目的 Go 依赖包。在编译 Go 源码时，Go 优先从项目根目录的 vendor 目录查找依赖；如果没有找到，再去 GOPATH 下的 vendor 目录下找；如果还没有找到，就去 GOPATH 下找。

这种方式解决了多 GOPATH 的问题，但是随着项目依赖的增多，vendor 目录会越来越大，造成整个项目仓库越来越大。在 vendor 机制下，一个中型项目的 vendor 目录有几百 M 的大小一点也不奇怪。

“百花齐放”：多种 Go 依赖包管理工具出现

这个阶段，社区也出现了很多 Go 依赖包管理的工具，这里我介绍三个比较有名的。

Godep：解决包依赖的管理工具，Docker、Kubernetes、CoreOS 等 Go 项目都曾用过 godep 来管理其依赖。

Govendor：它的功能比 Godep 多一些，通过 vendor 目录下的 vendor.json 文件来记录依赖包的版本。

Glide：相对完善的包管理工具，通过 glide.yaml 记录依赖信息，通过 glide.lock 追踪每个包的具体修改。

Govendor、Glide 都是在 Go 支持 vendor 之后推出的工具，Godep 在 Go 支持 vendor 之前也可以使用。Go 支持 vendor 之后，Godep 也改用了 vendor 模式。

Go1.9 版本：Dep

对于从 0 构建项目的新用户来说，Glide 功能足够，是个不错的选择。不过，Golang 依赖管理工具混乱的局面最终由官方来终结了：Golang 官方接纳了由社区组织合作开发的 Dep，作为 official experiment。在相当长的一段时间里，Dep 作为标准，成为了事实上的官方包管理工具。

因为 Dep 已经成为了 official experiment 的过去时，现在我们就没有必要再去深究了，让我们直接去了解谁才是未来的 official experiment 吧。

Go1.11 版本之后：Go Modules

Go1.11 版本推出了 Go Modules 机制，Go Modules 基于 vgo 演变而来，是 Golang 官方的包管理工具。在 Go1.13 版本，Go 语言将 Go Modules 设置为默认的 Go 管理工具；在 Go1.14 版本，Go 语言官方正式推荐在生产环境使用 Go Modules，并且鼓励所有用户从其他的依赖管理工具迁移过来。至此，Go 终于有了一个稳定的、官方的 Go 包管理工具。


到这里，我介绍了 Go 依赖包管理工具的历史，下面再来介绍下 Go Modules 的使用方法。

包 (package) 和模块 (module)

Go 程序被组织到 Go 包中，Go 包是同一目录中一起编译的 Go 源文件的集合。在一个源文件中定义的函数、类型、变量和常量，对于同一包中的所有其他源文件可见。

模块是存储在文件树中的 Go 包的集合，并且文件树根目录有 go.mod 文件。go.mod 文件定义了模块的名称及其依赖包，每个依赖包都需要指定导入路径和语义化版本（Semantic Versioning），通过导入路径和语义化版本准确地描述一个依赖。

这里要注意，"module" != "package"，模块和包的关系更像是集合和元素的关系，包属于模块，一个模块是零个或者多个包的集合。下面的代码段，引用了一些包：

 复制代码

```
1 import (  
2     // Go 标准包  
3     "fmt"  
4  
5     // 第三方包  
6     "github.com/spf13/pflag"  
7  
8     // 匿名包  
9     _ "github.com/jinzhu/gorm/dialects/mysql"  
10  
11    // 内部包  
12    "github.com/marmotedu/iam/internal/apiserver"  
13 )
```

这里的fmt、github.com/spf13/pflag和github.com/marmotedu/iam/internal/apiserver都是 Go 包。Go 中有 4 种类型的包，下面我来分别介绍下。

Go 标准包：在 Go 源码目录下，随 Go 一起发布的包。

第三方包：第三方提供的包，比如来自于 github.com 的包。

匿名包：只导入而不使用的包。通常情况下，我们只是想使用导入包产生的副作用，即引用包级别的变量、常量、结构体、接口等，以及执行导入包的init()函数。

内部包：项目内部的包，位于项目目录下。

下面的目录定义了一个模块：

 复制代码

```
1 $ ls hello/  
2 go.mod go.sum hello.go hello_test.go world
```

hello 目录下有一个 go.mod 文件，说明了这是一个模块，该模块包含了 hello 包和一个子包 world。该目录中也包含了一个 go.sum 文件，该文件供 Go 命令在构建时判断依赖包是否合法。这里你先简单了解下，我会在下面讲 go.sum 文件的时候详细介绍。

Go Modules 命令

Go Modules 的管理命令为 go mod，go mod 有很多子命令，你可以通过 go help mod 来获取所有的命令。下面我来具体介绍下这些命令。

download：下载 go.mod 文件中记录的所有依赖包。

edit：编辑 go.mod 文件。

graph：查看现有的依赖结构。

init：把当前目录初始化为一个新模块。

tidy：添加丢失的模块，并移除无用的模块。默认情况下，Go 不会移除 go.mod 文件中的无用依赖。当依赖包不再使用了，可以使用 go mod tidy 命令来清除它。

vendor：将所有依赖包存到当前目录下的 vendor 目录下。

verify：检查当前模块的依赖是否已经存储在本地下载的源代码缓存中，以及检查下载后是否有修改。

why：查看为什么需要依赖某模块。

Go Modules 开关

如果要使用 Go Modules，在 Go1.14 中仍然需要确保 Go Modules 特性处在打开状态。你可以通过环境变量 GO111MODULE 来打开或者关闭。GO111MODULE 有 3 个值，我来分别介绍下。

auto：在 Go1.14 版本中是默认值，在 \$GOPATH/src 下，且没有包含 go.mod 时则关闭 Go Modules，其他情况下都开启 Go Modules。

on：启用 Go Modules，Go1.14 版本推荐打开，未来版本会设为默认值。

off：关闭 Go Modules，不推荐。

所以，如果要打开 Go Modules，可以设置环境变量`export GO111MODULE=on`或者`export GO111MODULE=auto`，建议直接设置`export GO111MODULE=on`。

Go Modules 使用语义化的版本号，我们开发的模块在发布版本打 tag 的时候，要注意遵循语义化的版本要求，不遵循语义化版本规范的版本号都是无法拉取的。

模块下载

在执行 `go get` 等命令时，会自动下载模块。接下来，我会介绍下 `go` 命令是如何下载模块的。主要有三种下载方式：

通过代理下载；

指定版本号下载；

按最小版本下载。

通过代理来下载模块

默认情况下，Go 命令从 VCS (Version Control System，版本控制系统) 直接下载模块，例如 GitHub、Bitbucket、Bazaar、Mercurial 或者 SVN。

在 Go 1.13 版本，引入了一个新的环境变量 `GOPROXY`，用于设置 Go 模块代理 (Go module proxy)。模块代理可以使 Go 命令直接从代理服务器下载模块。`GOPROXY` 默认值为`https://proxy.golang.org,direct`，代理服务器可以指定多个，中间用逗号隔开，例如`GOPROXY=https://proxy.golang.org,https://goproxy.cn,direct`。当下载模块时，会优先从指定的代理服务器上下载。如果下载失败，比如代理服务器不可访问，或者 HTTP 返回码为 404 或 410，Go 命令会尝试从下一个代理服务器下载。

`direct` 是一个特殊指示符，用来指示 Go 回源到模块的源地址 (比如 GitHub 等) 去抓取，当值列表中上一个 Go module proxy 返回 404 或 410，Go 会自动尝试列表中的下一个，遇见 `direct` 时回源，遇见 EOF 时终止，并抛出类似 `invalid version: unknown revision...` 的错误。如果 `GOPROXY=off`，则 Go 命令不会尝试从代理服务器下载模块。

引入 Go module proxy 会带来很多好处，比如：

国内开发者无法访问像 `golang.org`、`gopkg.in`、`go.uber.org` 这类域名，可以设置 `GOPROXY` 为国内可以访问的代理服务器，解决依赖包下载失败的问题。

Go 模块代理会永久缓存和存储所有的依赖，并且这些依赖一经缓存，不可更改，这也意味着我们不需要再维护一个 `vendor` 目录，也可以避免因为维护 `vendor` 目录所带来的存储空间占用。

因为依赖永久存在于代理服务器，这样即使模块从互联网上被删除，也仍然可以通过代理服务器获取到。

一旦将 Go 模块存储在 Go 代理服务器中，就无法覆盖或删除它，这可以保护开发者免受可能注入相同版本恶意代码所带来的攻击。

我们不再需要 VCS 工具来下载依赖，因为所有的依赖都是通过 HTTP 的方式从代理服务器下载。

因为 Go 代理通过 HTTP 独立提供了源代码（.zip 存档）和 `go.mod`，所以下载和构建 Go 模块的速度更快。因为可以独立获取 `go.mod`（而之前必须获取整个仓库），所以解决依赖也更快。

当然，开发者也可以设置自己的 Go 模块代理，这样开发者可以对依赖包有更多的控制，并可以预防 VCS 停机所带来的下载失败。

在实际开发中，我们的很多模块可能需要从私有仓库拉取，通过代理服务器访问会报错，这时候我们需要将这些模块添加到环境变量 `GONOPROXY` 中，这些私有模块的哈希值也不会存在 `checksum database` 中，需要将这些模块添加到 `GONOSUMDB` 中。一般来说，我建议直接设置 `GOPRIVATE` 环境变量，它的值将作为 `GONOPROXY` 和 `GONOSUMDB` 的默认值。

`GONOPROXY`、`GONOSUMDB` 和 `GOPRIVATE` 都支持通配符，多个域名用逗号隔开，例如 `*.example.com,github.com`。

对于国内的 Go 开发者来说，目前有 3 个常用的 `GOPROXY` 可供选择，分别是官方、七牛和阿里云。

官方的 `GOPROXY`，国内用户可能访问不到，所以我更推荐使用七牛的 `goproxy.cn`，`goproxy.cn` 是七牛云推出的非营利性项目，它的目标是为中国和世界上其他地方的 Go 开发者提供一个免费、可靠、持续在线，且经过 CDN 加速的模块代理。

指定版本号下载

通常，我们通过`go get`来下载模块，下载命令格式为`go get <package[@version]>`，如下表所示：

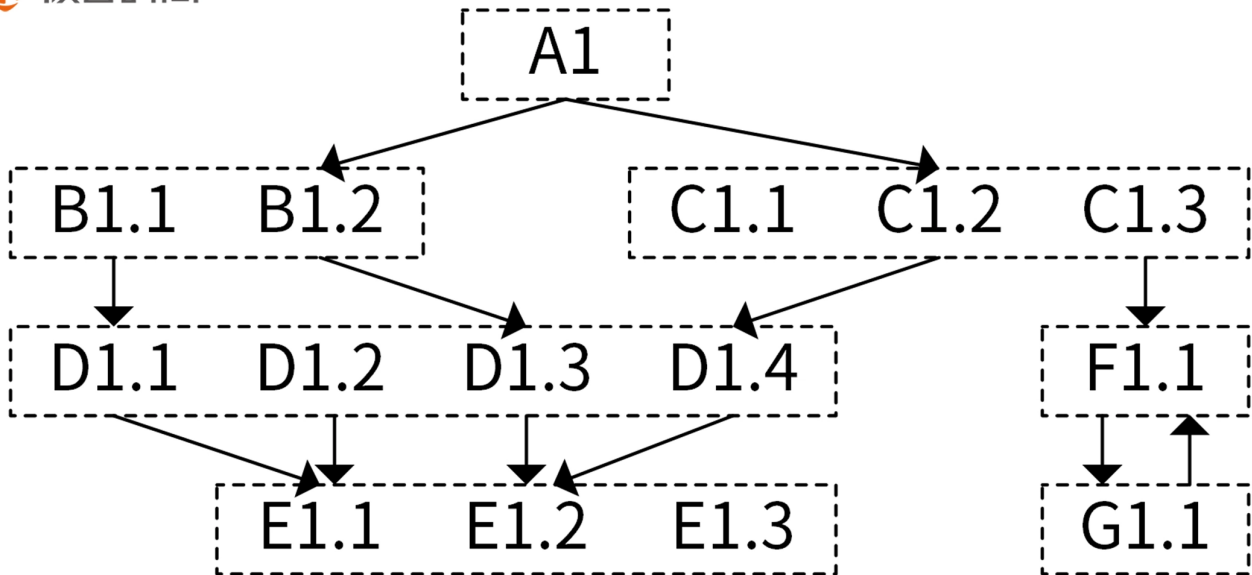


命令	作用
<code>go get golang.org/x/text@latest</code>	若模块有tag，会优先选择最新的稳定的版本，比如v0.4.5。如果没有稳定的版本，则选择最新的预发布版本，比如v0.0.1-pre1。若模块没有tag，Go命令会选择最新的commit
<code>go get golang.org/x/text</code>	效果等同于 <code>go get golang.org/x/text@latest</code>
<code>go get golang.org/x/text@v0.3.2</code>	下载tag为v0.3.2的版本
<code>go get golang.org/x/text@v0</code>	下载前缀是v0的最新版本
<code>go get "golang.org/x/text@<v0.3.2"</code>	进行版本比较（@<0.3.2或@>=0.3.1），将匹配最接近目标的可用标签版本。<为小于该版本的最新版本，>为大于该版本的最旧版本
<code>go get golang.org/x/text@master</code>	拉取master分支的最新commit
<code>go get golang.org/x/text@342b2e</code>	拉取hash为342b231的commit，如果该commit有对应的tag，则会被转换为对应的tag，没有则不转换

你可以使用`go get -u`更新 package 到 latest 版本，也可以使用`go get -u=patch`只更新小版本，例如从v1.2.4到v1.2.5。

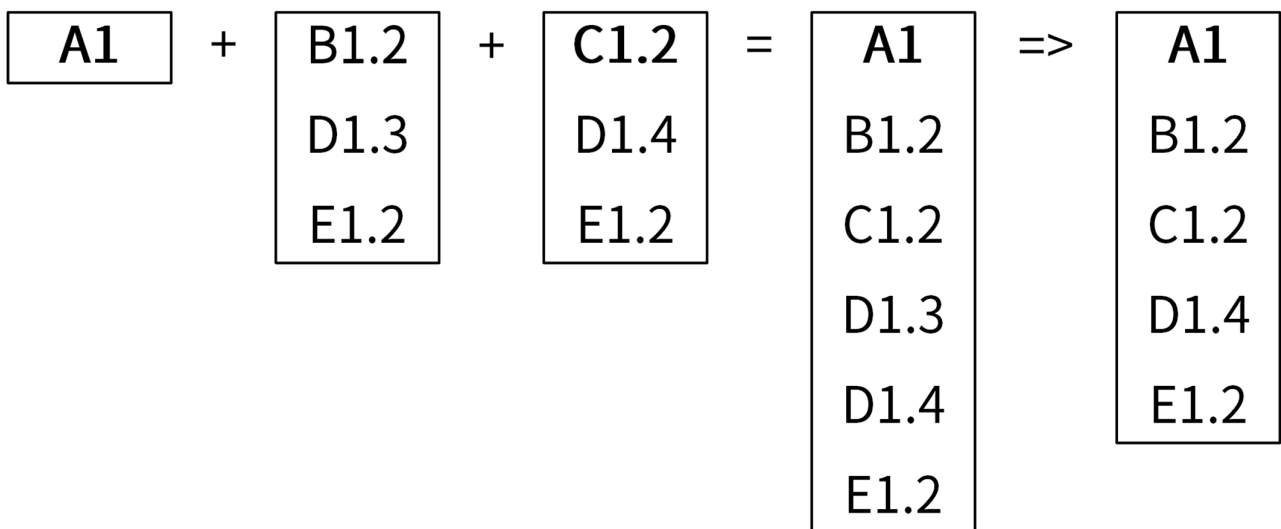
按最小版本下载

一个模块往往会依赖许多其他模块，并且不同的模块也可能会依赖同一个模块的不同版本，如下图所示：



在上述依赖中，模块 A 依赖了模块 B 和模块 C，模块 B 依赖了模块 D，模块 C 依赖了模块 D 和模块 F，模块 D 又依赖了模块 E。并且，同模块的不同版本还依赖了对应模块的不同版本。

那么 Go Modules 是如何选择版本的呢？Go Modules 会把每个模块的依赖版本清单都整理出来，最终得到一个构建清单，如下图所示：



上图中，rough list 和 final list 的区别在于重复引用的模块 D (v1.3、v1.4)，最终清单选用了 D 的v1.4版本。

这样做的主要原因有两个。第一个是语义化版本的控制。因为模块 D 的 v1.3 和 v1.4 版本变更都属于次版本号的变更，而在语义化版本的约束下，v1.4 必须要向下兼容 v1.3，因此我们要选择高版本的 v1.4。

第二个是模块导入路径的规范。主版本号不同，模块的导入路径就不一样。所以，如果出现不兼容的情况，主版本号会改变，例如从 v1 变为 v2，模块的导入路径也就改变了，因此不会影响 v1 版本。

go.mod 和 go.sum 介绍

在 Go Modules 中，go.mod 和 go.sum 是两个非常重要的文件，下面我就来详细介绍这两个文件。

go.mod 文件介绍

go.mod 文件是 Go Modules 的核心文件。下面是一个 go.mod 文件示例：

[复制代码](#)

```
1 module github.com/marmotedu/iam
2
3 go 1.14
4
5 require (
6     github.com/AlekSi/pointer v1.1.0
7     github.com/appleboy/gin-jwt/v2 v2.6.3
8     github.com/asaskevich/govalidator v0.0.0-20200428143746-21a406dcc535
9     github.com/gin-gonic/gin v1.6.3
10    github.com/golangci/golangci-lint v1.30.0 // indirect
11    github.com/google/uuid v1.0.0
12    github.com/blang/semver v3.5.0+incompatible
13    golang.org/x/text v0.3.2
14 )
15
16 replace (
17     github.com/gin-gonic/gin => /home/colin/gin
18     golang.org/x/text v0.3.2 => github.com/golang/text v0.3.2
19 )
20
21 exclude (
22     github.com/google/uuid v1.1.0
23 )
```

接下来，我会从 go.mod 语句、go.mod 版本号、go.mod 文件修改方法三个方面来介绍 go.mod。

1. go.mod 语句

go.mod 文件中包含了 4 个语句，分别是 module、require、replace 和 exclude。下面我来介绍下它们的功能。

module：用来定义当前项目的模块路径。

go：用来设置预期的 Go 版本，目前只是起标识作用。

require：用来设置一个特定的模块版本，格式为<导入包路径> <版本> [// indirect]。

exclude：用来从使用中排除一个特定的模块版本，如果我们知道模块的某个版本有严重的问题，就可以使用 exclude 将该版本排除掉。

replace：用来将一个模块版本替换为另外一个模块版本。格式为 \$module => \$newmodule，\$newmodule 可以是本地磁盘的相对路径，例如github.com/gin-gonic/gin => ./gin。也可以是本地磁盘的绝对路径，例如github.com/gin-gonic/gin => /home/lk/gin。还可以是网络路径，例如golang.org/x/text v0.3.2 => github.com/golang/text v0.3.2。

这里需要注意，虽然我们用 *newmodule* 替换了 module，但是在代码中的导入路径仍然为 \$module。replace 在实际开发中经常用到，下面的场景可能需要用到 replace：

在开启 Go Modules 后，缓存的依赖包是只读的，但在日常开发调试中，我们可能需要修改依赖包的代码来进行调试，这时可以将依赖包另存到一个新的位置，并在 go.mod 中替换这个包。

如果一些依赖包在 Go 命令运行时无法下载，就可以通过其他途径下载该依赖包，上传到开发构建机，并在 go.mod 中替换为这个包。

在项目开发初期，A 项目依赖 B 项目的包，但 B 项目因为种种原因没有 push 到仓库，这时也可以在 go.mod 中把依赖包替换为 B 项目的本地磁盘路径。

在国内访问 golang.org/x 的各个包都需要翻墙，可以在 go.mod 中使用 replace，替换成 GitHub 上对应的库，例如golang.org/x/text v0.3.0 =>

`github.com/golang/text v0.3.0`。

有一点要注意，`exclude` 和 `replace` 只作用于当前主模块，不影响主模块所依赖的其他模块。

2. go.mod 版本号

`go.mod` 文件中有很多版本号格式，我知道在平时使用中，有很多开发者对此感到困惑。这里，我来详细说明一下。

如果模块具有符合语义化版本格式的 `tag`，会直接展示 `tag` 的值，例如 `github.com/AlekSi/pointer v1.1.0`。

除了 `v0` 和 `v1` 外，主版本号必须显式地出现在模块路径的尾部，例如 `github.com/appleboy/gin-jwt/v2 v2.6.3`。

对于没有 `tag` 的模块，`Go` 命令会选择 `master` 分支上最新的 `commit`，并根据 `commit` 时间和哈希值生成一个符合语义化版本的版本号，例如

`github.com/asaskevich/govalidator v0.0.0-20200428143746-21a406dcc535`。

如果模块名字跟版本不符合规范，例如模块的名字为 `github.com/blang/semver`，但是版本为 `v3.5.0`（正常应该是 `github.com/blang/semver/v3`），`go` 会在 `go.mod` 的版本号后加 `+incompatible` 表示。

如果 `go.mod` 中的包是间接依赖，则会添加 `// indirect` 注释，例如 `github.com/golangci/golangci-lint v1.30.0 // indirect`。

这里要注意，`Go Modules` 要求模块的版本号格式为 `v<major>.<minor>.<patch>`，如果 `<major>` 版本号大于 1，它的版本号还要体现在模块名字中，例如模块 `github.com/blang/semver` 版本号增长到 `v3.x.x`，则模块名应为 `github.com/blang/semver/v3`。

这里再详细介绍下出现 `// indirect` 的情况。原则上 `go.mod` 中出现的都是直接依赖，但是下面的两种情况只要出现一种，就会在 `go.mod` 中添加间接依赖。

直接依赖未启用 Go Modules：如果模块 A 依赖模块 B，模块 B 依赖 B1 和 B2，但是 B 没有 go.mod 文件，则 B1 和 B2 会记录到 A 的 go.mod 文件中，并在最后加上 `// indirect`。

直接依赖 go.mod 文件中缺失部分依赖：如果模块 A 依赖模块 B，模块 B 依赖 B1 和 B2，B 有 go.mod 文件，但是只有 B1 被记录在 B 的 go.mod 文件中，这时候 B2 会被记录到 A 的 go.mod 文件中，并在最后加上 `// indirect`。

3. go.mod 文件修改方法


要修改 go.mod 文件，我们可以采用下面这三种方法：

Go 命令在运行时自动修改。

手动编辑 go.mod 文件，编辑之后可以执行 `go mod edit -fmt` 格式化 go.mod 文件。

执行 go mod 子命令修改。

在实际使用中，我建议你采用第三种修改方法，和其他两种相比不太容易出错。使用方式如下：

 复制代码

```
1 go mod edit -fmt # go.mod 格式化
2 go mod edit -require=golang.org/x/text@v0.3.3 # 添加一个依赖
3 go mod edit -droprequire=golang.org/x/text # require的反向操作，移除一个依赖
4 go mod edit -replace=github.com/gin-gonic/gin=/home/colin/gin # 替换模块版本
5 go mod edit -dropreplace=github.com/gin-gonic/gin # replace的反向操作
6 go mod edit -exclude=golang.org/x/text@v0.3.1 # 排除一个特定的模块版本
7 go mod edit -dropexclude=golang.org/x/text@v0.3.1 # exclude的反向操作
```

go.sum 文件介绍

Go 会根据 go.mod 文件中记载的依赖包及其版本下载包源码，但是下载的包可能被篡改，缓存在本地的包也可能被篡改。单单一个 go.mod 文件，不能保证包的一致性。为了解决这个潜在的安全问题，Go Modules 引入了 go.sum 文件。


go.sum 文件用来记录每个依赖包的 hash 值，在构建时，如果本地的依赖包 hash 值与 go.sum 文件中记录的不一致，则会拒绝构建。go.sum 中记录的依赖包是所有的依赖包，包括间接和直接的依赖包。

这里提示下，为了避免已缓存的模块被更改，\$GOPATH/pkg/mod 下缓存的包是只读的，不允许修改。

接下来我从 go.sum 文件内容、go.sum 文件生成、校验三个方面来介绍 go.sum。

1. go.sum 文件内容


下面是一个 go.sum 文件的内容：

 复制代码

```
1 golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c h1:qg0Y6WgZ0aTkIIMiVjBQcw
2 golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c/go.mod h1:NqM8EUOU14njkJ3
3 rsc.io/quote v1.5.2 h1:w5fcysjrx7yqtD/a0+QwRjYZ0KnaM9Uh2b40tElTs3Y=
4 rsc.io/quote v1.5.2/go.mod h1:LzX7hefJvL54yjefDEdHNONDjII0t9xZLPXsUe+TKr0=
5 rsc.io/sampler v1.3.0 h1:7uVkiFmeBqHfdjD+gZwtXXI+RODJ2Wc407MPEh/QiW4=
6 rsc.io/sampler v1.3.0/go.mod h1:T1hPZKmBbMNahiBKfy5HrXp6adAjACjK9JXDnKaTXpA=
```

go.sum 文件中，每行记录由模块名、版本、哈希算法和哈希值组成，如<module><version>[/go.mod] <algorithm>:<hash>。目前，从 Go1.11 到 Go1.14 版本，只有一个算法 SHA-256，用 h1 表示。

正常情况下，每个依赖包会包含两条记录，分别是依赖包所有文件的哈希值和该依赖包 go.mod 的哈希值，例如：

 复制代码

```
1 rsc.io/quote v1.5.2 h1:w5fcysjrx7yqtD/a0+QwRjYZ0KnaM9Uh2b40tElTs3Y=
2 rsc.io/quote v1.5.2/go.mod h1:LzX7hefJvL54yjefDEdHNONDjII0t9xZLPXsUe+TKr0=
```

但是，如果一个依赖包没有 go.mod 文件，就只记录依赖包所有文件的哈希值，也就是只有第一条记录。额外记录 go.mod 的哈希值，主要是为了在计算依赖树时不必下载完整的依赖包版本，只根据 go.mod 即可计算依赖树。

2. go.sum 文件生成

在 Go Modules 开启时，如果我们的项目需要引入一个新的包，通常会执行 `go get` 命令，例如：

```
1 $ go get rsc.io/quote
```

[复制代码](#)

当执行 `go get rsc.io/quote` 命令后，`go get` 命令会先将依赖包下载到 `$GOPATH/pkg/mod/cache/download`，下载的依赖包文件名格式为 `$version.zip`，例如 `v1.5.2.zip`。

下载完成之后，`go get` 会对该 `zip` 包做哈希运算，并将结果存在 `$version.ziphash` 文件中，例如 `v1.5.2.ziphash`。如果在项目根目录下执行 `go get` 命令，则 `go get` 会同时更新 `go.mod` 和 `go.sum` 文件。例如，`go.mod` 新增一行 `require rsc.io/quote v1.5.2`，`go.sum` 新增两行：

```
1 rsc.io/quote v1.5.2 h1:w5fcysjrx7yqtD/a0+QwRjYZ0KnaM9Uh2b40tElTs3Y=  
2 rsc.io/quote v1.5.2/go.mod h1:LzX7hefJvL54yjeFDEdHNDjII0t9xZLPxsUe+TKr0=
```

[复制代码](#)

3. 校验

在我们执行构建时，`go` 命令会从本地缓存中查找所有的依赖包，并计算这些依赖包的哈希值，然后与 `go.sum` 中记录的哈希值进行对比。如果哈希值不一致，则校验失败，停止构建。

校验失败可能是因为本地指定版本的依赖包被修改过，也可能是 `go.sum` 中记录的哈希值是错误的。但是 `Go` 命令倾向于相信依赖包被修改过，因为当我们在 `go get` 依赖包时，包的哈希值会经过校验和数据库（checksum database）进行校验，校验通过才会被加入到 `go.sum` 文件中。也就是说，`go.sum` 文件中记录的哈希值是可信的。

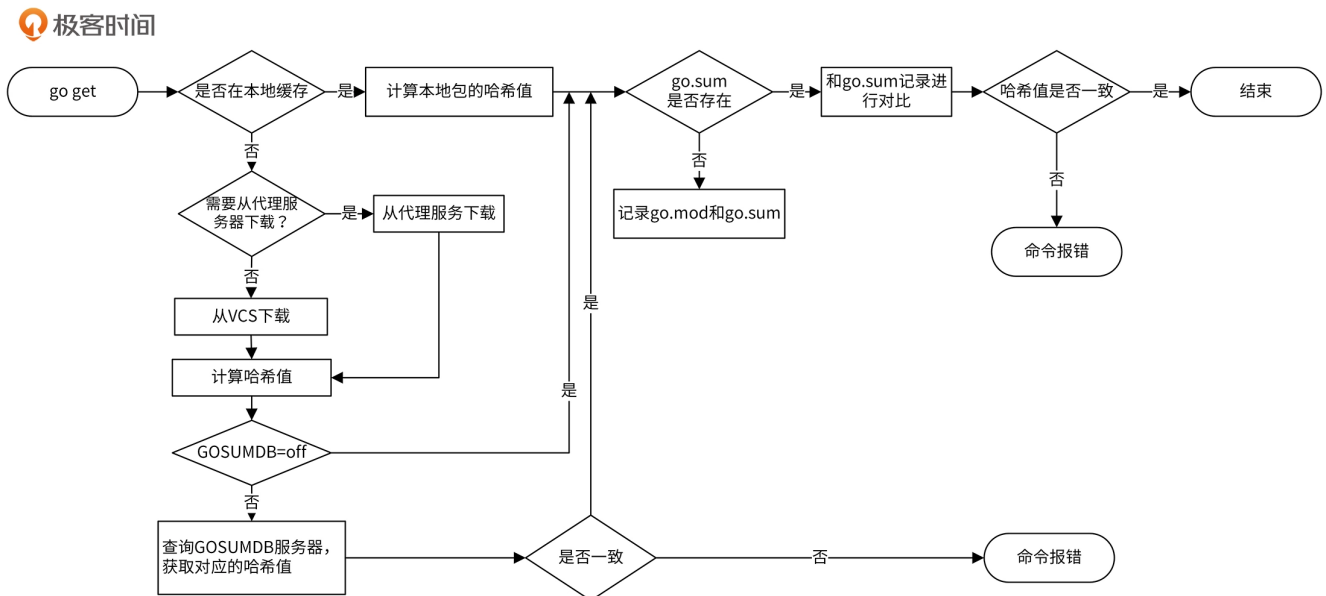
校验和数据库可以通过环境变量GOSUMDB指定，GOSUMDB的值是一个 web 服务器，默认值是sum.golang.org。该服务可以用来查询依赖包指定版本的哈希值，保证拉取到的模块版本数据没有经过篡改。

如果设置GOSUMDB为off，或者使用go get的时候启用了-insecure参数，Go 就不会去对下载的依赖包做安全校验，这存在一定的安全隐患，所以我建议你开启校验和数据库。如果对安全性要求很高，同时又访问不了sum.golang.org，你也可以搭建自己的校验和数据库。

值得注意的是，Go checksum database 可以被 Go module proxy 代理，所以当我们设置了GOPROXY后，通常情况下不用再设置GOSUMDB。还要注意的，go.sum 文件也应该提交到你的 Git 仓库中去。

模块下载流程

上面，我介绍了模块下载的整体流程，还介绍了 go.mod 和 go.sum 这两个文件。因为内容比较多，这里用一张图片来做个总结：



最后还想介绍下 Go modules 的全局缓存。Go modules 中，相同版本的模块只会缓存一份，其他所有模块公用。目前，所有模块版本数据都缓存在 \$GOPATH/pkg/mod 和 \$GOPATH/pkg/sum 下，未来有可能移到 \$GOCACHE/mod 和 \$GOCACHE/sum 下，我认为这可能发生在 GOPATH 被淘汰后。你可以使用 go clean -modcache 清除所有的缓存。

总结

Go 依赖包管理是 Go 语言中一个重点的功能。在 Go1.11 版本之前，并没有官方的依赖包管理工具，业界虽然存在多个 Go 依赖包管理方案，但效果都不理想。直到 Go1.11 版本，Go 才推出了官方的依赖包管理工具，Go Modules。这也是我建议你在进行 Go 项目开发时选择的依赖包管理工具。

Go Modules 提供了 `go mod` 命令，来管理 Go 的依赖包。`go mod` 有很多子命令，这些子命令可以完成不同的功能。例如，初始化当前目录为一个新模块，添加丢失的模块，移除无用的模块，等等。

在 Go Modules 中，有两个非常重要的文件：`go.mod` 和 `go.sum`。`go.mod` 文件是 Go Modules 的核心文件，Go 会根据 `go.mod` 文件中记载的依赖包及其版本下载包源码。`go.sum` 文件用来记录每个依赖包的 hash 值，在构建时，如果本地的依赖包 hash 值与 `go.sum` 文件中记录的不一致，就会拒绝构建。

Go 在下载依赖包时，可以通过代理来下载，也可以指定版本号下载。如果不指定版本号，Go Modules 会根据自定义的规则，选择最小版本来下载。

课后练习

1. 思考下，如果不提交 `go.sum`，会有什么风险？
2. 找一个没有使用 Go Modules 管理依赖包的 Go 项目，把它的依赖包管理方式切换为 Go Modules。

欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

 赞 0  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。

更多学习推荐

175 道 Go 工程师 大厂常考面试题

限量免费领取 

精选留言 (3)

 写留言**josephzxy** 置顶

2021-09-09

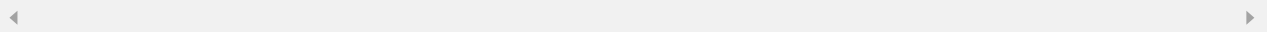
“思考下，如果不提交 go.sum，会有什么风险？”
如果go get时，GOSUMDB=off，就没有办法校验下载的包是否被篡改。

推荐两篇博文可做辅助阅读

[https://zaracooper.github.io/blog/posts/go-module-tidbits/...](https://zaracooper.github.io/blog/posts/go-module-tidbits/)

展开 ∨

作者回复: 老哥，回答给满分~



4

**helloworld**

2021-09-10

“如果不指定版本号，Go Modules 会根据自定义的规则，选择最小版本来下载。”，这里说的最小版本指的是latest版本吗

展开 ∨

1





helloworld

2021-09-10

分析的真细致，厉害👍

展开▼

