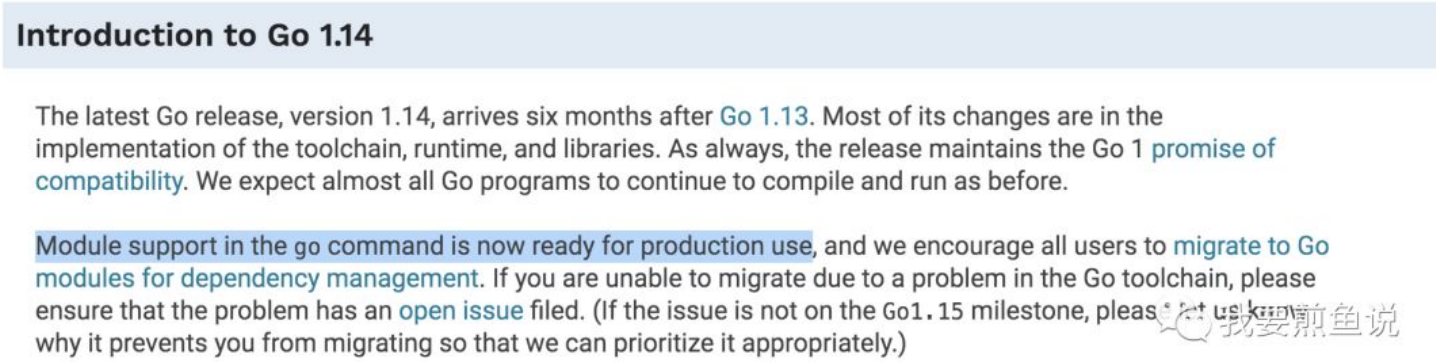
煎鱼
6.3k

Go Modules 终极入门

Go modules 是 Go 语言中正式官宣的项目依赖解决方案，Go modules（前身为vgo）于 Go1.11 正式发布，在 Go1.14 已经准备好，并且可以用在生产上（ready for production）了，Go官方也鼓励所有用户从其他依赖项管理工具迁移到 Go modules。

而 Go1.14，在近期也终于正式发布，Go 官方亲自 “喊” 你来用：



因此在今天这篇文章中，我将给大家带来 Go modules 的 “终极入门”，欢迎大家一起共同探讨。

Go modules 是 Go 语言中正式官宣的项目依赖管理工具，Go modules（前身为vgo）于 Go1.11 正式发布，在 Go1.14 已经准备好，并且可以用在生产上（ready for production）了，鼓励所有用户从其他依赖项管理工具迁移到 Go modules。

什么是Go Modules

Go modules 是 Go 语言的依赖解决方案，发布于 Go1.11，成长于 Go1.12，丰富于 Go1.13，正式于 Go1.14 推荐在生产上使用。

Go moudles 目前集成在 Go 的工具链中，只要安装了 Go，自然而然也就可以使用 Go moudles 了，而 Go modules 的出现也解决了在 Go1.11 前的几个常见争议问题：

1. Go 语言长久以来的依赖管理问题。
2. “淘汰”现有的 GOPATH 的使用模式。
3. 统一社区中的其它的依赖管理工具（提供迁移功能）。

GOPATH的那些点点滴滴

我们有提到 Go modules 的解决的问题之一就是“淘汰”掉 GOPATH，但是 GOPATH 又是什么呢，为什么在 Go1.11 前就使用 GOPATH，而 Go1.11 后就开始逐步建议使用 Go modules，不再推荐 GOPATH 的模式了呢?

GOPATH是什么

我们先看看第一个问题，GOPATH 是什么，我们可以输入如下命令查看：

```
$ go env
GOPATH="/Users/eddycjy/go"
...
```

我们输入`go env`命令行后可以查看到 GOPATH 变量的结果，我们进入到该目录下进行查看，如下：

```
go
├── bin
├── pkg
└── src
    ├── github.com
    ├── golang.org
    ├── google.golang.org
    ├── gopkg.in
    ....
```

GOPATH目录下一共包含了三个子目录，分别是：

- bin：存储所编译生成的二进制文件。
- pkg：存储预编译的目标文件，以加快程序的后续编译速度。
- src：存储所有.go文件或源代码。在编写 Go 应用程序，程序包和库时，一般会以$GOPATH/src/github.com/foo/bar的路径进行存放。

因此在使用 GOPATH 模式下，我们需要将应用代码存放在固定的$GOPATH/src目录下，并且如果执行go get来拉取外部依赖会自动下载并安装到$GOPATH目录下。

为什么弃用GOPATH模式

在 GOPATH 的 $GOPATH/src 下进行 .go 文件或源代码的存储，我们可以称其为 GOPATH 的模式，这个模式，看起来好像没有什么问题，那么为什么我们要弃用呢，参见如下原因：

- GOPATH 模式下没有版本控制的概念，具有致命的缺陷，至少会造成以下问题：
  - 在执行go get的时候，你无法传达任何的版本信息的期望，也就是说你也无法知道自己当前更新的是哪一个版本，也无法通过指定来拉取自己所期望的具体版本。
  - 在运行Go应用程序的时候，你无法保证其它人与你所期望依赖的第三方库是相同的版本，也就是说在项目依赖库的管理上，你无法保证所有人的依赖版本都一致。
  - 你没办法处理 v1、v2、v3 等等不同版本的引用问题，因为 GOPATH 模式下的导入路径都是一样的，都是 github.com/foo/bar。
- Go 语言官方从 Go1.11 起开始推进 Go modules（前身vgo），Go1.13 起不再推荐使用 GOPATH 的使用模式，Go modules 也渐趋稳定，因此新项目也没有必要继续使用GOPATH模式。

在GOPATH模式下的产物

Go1 在 2012 年 03 月 28 日发布，而 Go1.11 是在 2018 年 08 月 25 日才正式发布（数据来源：Github Tag），在这个空档的时间内，并没有 Go modules 这一个东西，最早期可能还好说，因为刚发布，用的人不多，所以没有明显暴露，但是后期 Go 语言使用的人越来越多了，那怎么办?

这时候社区中逐渐的涌现出了大量的依赖解决方案，百花齐放，让人难以挑选，其中包括我们所熟知的 vendor 目录的模式，以及曾经一度被认为是“官宣”的 dep 的这类依赖管理工具。

但为什么 dep 没有正在成为官宣呢，其实是因为随着 Russ Cox 与 Go 团队中的其他成员不断深入地讨论，发现dep 的一些细节似乎越来越不适合 Go，因此官方采取了另起 proposal 的方式来推进，其方案的结果一开始先是释出 vgo（Go modules的前身，知道即可，不需要深入了解），最终演变为我们现在所见到的 Go modules，也在 Go1.11 正式进入了 Go 的工具链。

因此与其说是 “在GOPATH模式下的产物”，不如说是历史为当前提供了重要的教训，因此出现了 Go modules。

Go Modules基本使用

在初步了解了 Go modules 的前世今生后，我们正式进入到 Go modules 的使用，首先我们将从头开始创建一个 Go modules 的项目（原则上所创建的目录应该不要放在 GOPATH 之中）。

所提供的命令

在 Go modules 中，我们能够使用如下命令进行操作：

命令	作用
go mod init	生成 go.mod 文件
go mod download	下载 go.mod 文件中指明的所有依赖
go mod tidy	整理现有的依赖
go mod graph	查看现有的依赖结构
go mod edit	编辑 go.mod 文件
go mod vendor	导出项目所有的依赖到vendor目录
go mod verify	校验一个模块是否被篡改过
go mod why	查看为什么需要依赖某模块

所提供的环境变量

在 Go modules 中有如下常用环境变量，我们可以通过 `go env` 命令来进行查看，如下：

```
$ go env
GO111MODULE="auto"
GOPROXY="https://proxy.golang.org,direct"
GONOPROXY=""
GOSUMDB="sum.golang.org"
GONOSUMDB=""
GOPRIVATE=""
...
```

GO111MODULE

Go语言提供了 GO111MODULE 这个环境变量来作为 Go modules 的开关，其允许设置以下参数：

- auto：只要项目包含了 go.mod 文件的话启用 Go modules，目前在 Go1.11 至 Go1.14 中仍然是默认值。
- on：启用 Go modules，推荐设置，将会是未来版本中的默认值。
- off：禁用 Go modules，不推荐设置。

GO111MODULE的小历史

你可能会留意到 GO111MODULE 这个名字比较“奇特”，实际上在 Go 语言中经常会有这类阶段性的变量， GO111MODULE 这个命名代表着Go语言在 1.11 版本添加的，针对 Module 的变量。

像是在 Go1.5 版本的时候，也发布了一个系统环境变量 GO15VENDOREXPERIMENT，作用是用于开启 vendor 目录的支持，当时其默认值也不是开启，仅仅作为 experimental。其随后在 Go1.6 版本时也将默认值改为了开启，并且最后作为了official，GO15VENDOREXPERIMENT 系统变量就退出了历史舞台。

而未来 GO111MODULE 这一个系统环境变量也会面临这个问题，也会先调整为默认值为 on（曾经在Go1.13想想改为 on，并且已经合并了 PR，但最后因为种种原因改回了 auto），然后再把 GO111MODULE 的支持给去掉，我们猜测应该会在 Go2 将 GO111MODULE 给去掉，因为如果直接去掉 GO111MODULE 的支持，会存在兼容性问题。

GOPROXY

这个环境变量主要是用于设置 Go 模块代理（Go module proxy），其作用是用于使 Go 在后续拉取模块版本时能够脱离传统的 VCS 方式，直接通过镜像站点来快速拉取。

GOPROXY 的默认值是：`https://proxy.golang.org,direct`，这有一个很严重的问题，就是 `proxy.golang.org` 在国内是无法访问的，因此这会直接卡住你的第一步，所以你必须在开启 Go modules 的时，同时设置国内的 Go 模块代理，执行如下命令：

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

GOPROXY的值是一个以英文逗号 “,” 分割的 Go 模块代理列表，允许设置多个模块代理，假设你不想使用，也可以将其设置为 “off” ，这将会禁止 Go 在后续操作中使用任何 Go 模块代理。

direct是什么

而在刚刚设置的值中，我们可以发现值列表中有 “direct” 标识，它又有什么作用呢?

实际上 “direct” 是一个特殊指示符，用于指示 Go 回源到模块版本的源地址去抓取（比如 GitHub 等），场景如下：当值列表中上一个 Go 模块代理返回 404 或 410 错误时，Go 自动尝试列表中的下一个，遇见 “direct” 时回源，也就是回到源地址去抓取，而遇见 EOF 时终止并抛出类似 “invalid version: unknown revision...” 的错误。

GOSUMDB

它的值是一个 Go checksum database，用于在拉取模块版本时（无论是从源站拉取还是通过 Go module proxy 拉取）保证拉取到的模块版本数据未经过篡改，若发现不一致，也就是可能存在篡改，将会立即中止。

GOSUMDB的默认值为：`sum.golang.org`，在国内也是无法访问的，但是 GOSUMDB 可以被 Go 模块代理所代理（详见：Proxying a Checksum Database）。

因此我们可以通过设置 GOPROXY 来解决，而先前我们所设置的模块代理 `goproxy.cn` 就能支持代理 `sum.golang.org`，所以这一个问题在设置 GOPROXY 后，你可以不需要过度关心。

另外若对 GOSUMDB 的值有自定义需求，其支持如下格式：

- 格式 1：`<SUMDB_NAME>+<PUBLIC_KEY>`。
- 格式 2：`<SUMDB_NAME>+<PUBLIC_KEY> <SUMDB_URL>`。

也可以将其设置为“off”，也就是禁止 Go 在后续操作中校验模块版本。

GONOPROXY/GONOSUMDB/GOPRIVATE

这三个环境变量都是用在当前项目依赖了私有模块，例如像是你公司的私有 git 仓库，又或是 github 中的私有库，都是属于私有模块，都是要进行设置的，否则会拉取失败。

更细致来讲，就是依赖了由 GOPROXY 指定的 Go 模块代理或由 GOSUMDB 指定 Go checksum database 都无法访问到的模块时的场景。

而一般建议直接设置 GOPRIVATE，它的值将作为 GONOPROXY 和 GONOSUMDB 的默认值，所以建议的最佳姿势是直接使用 GOPRIVATE。

并且它们的值都是一个以英文逗号 “,” 分割的模块路径前缀，也就是可以设置多个，例如：

```
$ go env -w GOPRIVATE="git.example.com,github.com/eddycjy/mquote"
```

设置后，前缀为 git.xxx.com 和 github.com/eddycjy/mquote 的模块都会被认为是私有模块。

如果不想每次都重新设置，我们也可以利用通配符，例如：

```
$ go env -w GOPRIVATE="*.example.com"
```

这样子设置的话，所有模块路径为 example.com 的子域名（例如：git.example.com）都将不经过 Go module proxy 和 Go checksum database，需要注意的是不包括 example.com 本身。

开启Go Modules

目前Go modules并不是默认开启，因此Go语言提供了GO111MODULE这个环境变量来作为Go modules的开关，其允许设置以下参数：

- auto：只要项目包含了go.mod文件的话启用 Go modules，目前在Go1.11至Go1.14中仍然是默认值。
- on：启用 Go modules，推荐设置，将会是未来版本中的默认值。
- off：禁用 Go modules，不推荐设置。

如果你不确定你当前的值是什么，可以执行`go env`命令，查看结果：

```
$ go env
GO111MODULE="off"
...
```

如果需要对GO111MODULE的值进行变更，推荐通过`go env`命令进行设置：

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

但是需要注意的是如果对应的系统环境变量有值了（进行过设置），会出现如下警告信息：`warning: go env -w GO111MODULE=... does not override conflicting OS environment variable`。

又或是可以通过直接设置系统环境变量（写入对应的.bash_profile文件亦可）来实现这个目的：

```
$ export GO111MODULE=on
```

初始化项目

在完成 Go modules 的开启后，我们需要创建一个示例项目来进行演示，执行如下命令：

```
$ mkdir -p $HOME/eddycjy/module-repo
$ cd $HOME/eddycjy/module-repo
```

然后进行Go modules的初始化，如下：

```
$ go mod init github.com/eddycjy/module-repo
go: creating new go.mod: module github.com/eddycjy/module-repo
```

在执行 `go mod init` 命令时，我们指定了模块导入路径为 `github.com/eddycjy/module-repo`。接下来我们在该项目根目录下创建 main.go 文件，如下：

```
package main

import (
    "fmt"
    "github.com/eddycjy/mquote"
)

func main() {
    fmt.Println(mquote.GetHello())
}
```

然后在项目根目录执行 `go get github.com/eddycjy/mquote` 命令，如下：

```
$ go get github.com/eddycjy/mquote
go: finding github.com/eddycjy/mquote latest
go: downloading github.com/eddycjy/mquote v0.0.0-20200220041913-e066a990ce6f
go: extracting github.com/eddycjy/mquote v0.0.0-20200220041913-e066a990ce6f
```

查看go.mod 文件

在初始化项目时，会生成一个 go.mod 文件，是启用了 Go modules 项目所必须的最重要的标识，同时也是GO111MODULE 值为 auto 时的识别标识，它描述了当前项目（也就是当前模块）的元信息，每一行都以一个动词开头。

在我们刚刚进行了初始化和简单拉取后，我们再次查看go.mod文件，基本内容如下：

```
module github.com/eddycjy/module-repo

go 1.13

require (
    github.com/eddycjy/mquote v0.0.0-20200220041913-e066a990ce6f
)
```

为了更进一步的讲解，我们模拟引用如下：

```
module github.com/eddycjy/module-repo

go 1.13

require (
    example.com/apple v0.1.2
    example.com/banana v1.2.3
    example.com/banana/v2 v2.3.4
    example.com/pear // indirect
    example.com/strawberry // incompatible
)

exclude example.com/banana v1.2.4
replace example.com/apple v0.1.2 => example.com/fried v0.1.0
replace example.com/banana => example.com/fish
```

- module：用于定义当前项目的模块路径。
- go：用于标识当前模块的 Go 语言版本，值为初始化模块时的版本，目前来看还只是个标识作用。
- require：用于设置一个特定的模块版本。
- exclude：用于从使用中排除一个特定的模块版本。
- replace：用于将一个模块版本替换为另外一个模块版本。

另外你会发现 example.com/pear 的后面会有一个 indirect 标识，indirect 标识表示该模块为间接依赖，也就是在当前应用程序中的 import 语句中，并没有发现这个模块的明确引用，有可能是你先手动 go get 拉取下来的，也有可能是你所依赖的模块所依赖的，情况有好几种。

查看go.sum文件

在第一次拉取模块依赖后，会发现多出了一个 go.sum 文件，其详细罗列了当前项目直接或间接依赖的所有模块版本，并写明了那些模块版本的 SHA-256 哈希值以备 Go 在今后的操作中保证项目所依赖的那些模块版本不会被篡改。

```
github.com/eddycjy/mquote v0.0.1 h1:4QHXKo7J8a6J/k8UA6CiHhswJQs0sm2foAQQUq8GFHM=
github.com/eddycjy/mquote v0.0.1/go.mod h1:ZtlkDs7Mriynl7wsDQ4cU23okEtVYqHwl7F1eDh4qPg=
github.com/eddycjy/mquote/module/tour v0.0.1 h1:cc+pgV0LnR8Fhou0zNHughT7IbSnLvfUZ+X3fvshrv8=
github.com/eddycjy/mquote/module/tour v0.0.1/go.mod h1:8uL1FOiQJZ4/1hzqQ5mv4Sm7nJcwYu41F3nZmkiWx5I=
...
```

我们可以看到一个模块路径可能有如下两种：

```
github.com/eddycjy/mquote v0.0.1 h1:4QHXKo7J8a6J/k8UA6CiHhswJQs0sm2foAQQUq8GFHM=
github.com/eddycjy/mquote v0.0.1/go.mod h1:ZtlkDs7Mriynl7wsDQ4cU23okEtVYqHwl7F1eDh4qPg=
```

h1 hash 是 Go modules 将目标模块版本的 zip 文件开包后，针对所有包内文件依次进行 hash，然后再把它们的 hash 结果按照固定格式和算法组成总的 hash 值。

而 h1 hash 和 go.mod hash 两者，要不就是同时存在，要不就是只存在 go.mod hash。那什么情况下会不存在 h1 hash 呢，就是当 Go 认为肯定用不到某个模块版本的时候就会省略它的 h1 hash，就会出现不存在 h1 hash，只存在 go.mod hash 的情况。

查看全局缓存

我们刚刚成功的将 github.com/eddycjy/mquote 模块拉取了下来，其拉取的结果缓存在 $GOPATH/pkg/mod和 $GOPATH/pkg/sumdb 目录下，而在mod目录下会以 github.com/foo/bar 的格式进行存放，如下：

```
mod
├── cache
├── github.com
├── golang.org
├── google.golang.org
├── gopkg.in
...
```

需要注意的是同一个模块版本的数据只缓存一份，所有其它模块共享使用。如果你希望清理所有已缓存的模块版本数据，可以执行 go clean -modcache 命令。

Go Modules下的go get行为

在拉取项目依赖时，你会发现拉取的过程总共分为了三大步，分别是 finding（发现）、downloading（下载）以及 extracting（提取）， 并且在拉取信息上一共分为了三段内容：

需要注意的是，所拉取版本的 commit 时间是以UTC时区为准，而并非本地时区，同时我们会发现我们 `go get` 命令所拉取到的版本是 v0.0.0，这是因为我们是直接执行 `go get -u` 获取的，并没有指定任何的版本信息，由 Go modules 自行按照内部规则进行选择。

go get的拉取行为

刚刚我们用 `go get` 命令拉取了新的依赖，那么 `go get` 又提供了哪些功能呢，常用的拉取命令如下：

命令	作用
go get	拉取依赖，会进行指定性拉取（更新），并不会更新所依赖的其它模块。
go get -u	更新现有的依赖，会强制更新它所依赖的其它全部模块，不包括自身。
go get -u -t ./...	更新所有直接依赖和间接依赖的模块版本，包括单元测试中用到的。

那么我想选择具体版本应当如何执行呢，如下：

命令	作用
go get golang.org/x/text@latest	拉取最新的版本，若存在tag，则优先使用。
go get golang.org/x/text@master	拉取 master 分支的最新 commit。
go get golang.org/x/text@v0.3.2	拉取 tag 为 v0.3.2 的 commit。
go get golang.org/x/text@342b2e	拉取 hash 为 342b231 的 commit，最终会被转换为 v0.3.2。

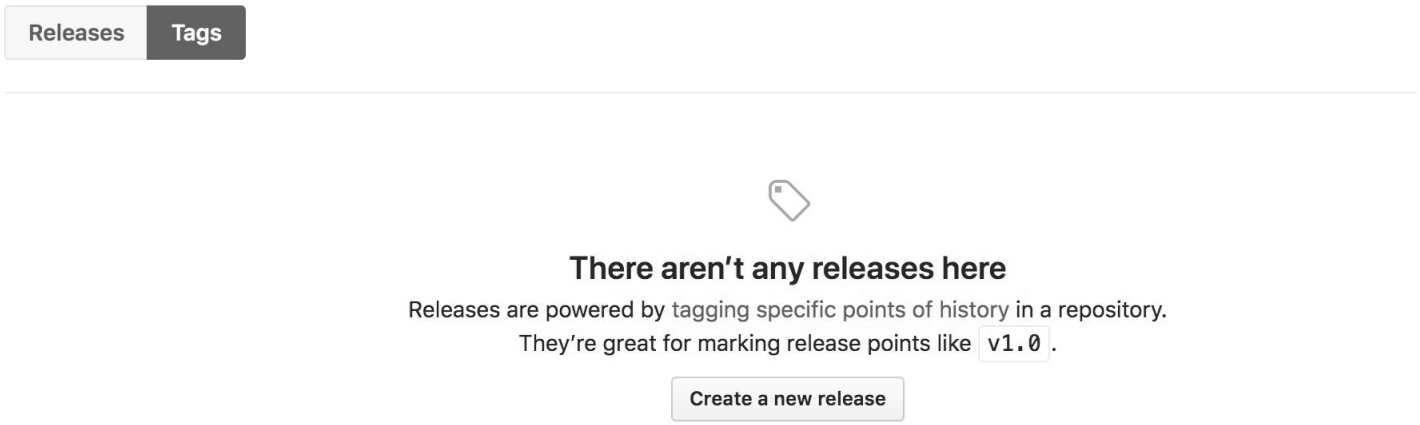
go get的版本选择

我们回顾一下我们拉取的 `go get github.com/eddycjy/mquote`，其结果是 `v0.0.0-20200220041913-e066a990ce6f`，对照着上面所提到的 `go get` 行为来看，你可能还会有一些疑惑，那就是在 `go get` 没有指定任何版本的情况下，它的版本选择规则是怎么样的，也就是为什么 `go get` 拉取的是 `v0.0.0`，它什么时候会拉取正常带版本号的 tags 呢。实际上这需要区分两种情况，如下：

1. 所拉取的模块有发布 tags：
    - 如果只有单个模块，那么就取主版本号最大的那个tag。
    - 如果有多个模块，则推算相应的模块路径，取主版本号最大的那个tag（子模块的tag的模块路径会有前缀要求）
2. 所拉取的模块没有发布过 tags：
    - 默认取主分支最新一次 commit 的 commithash。

没有发布过 tags

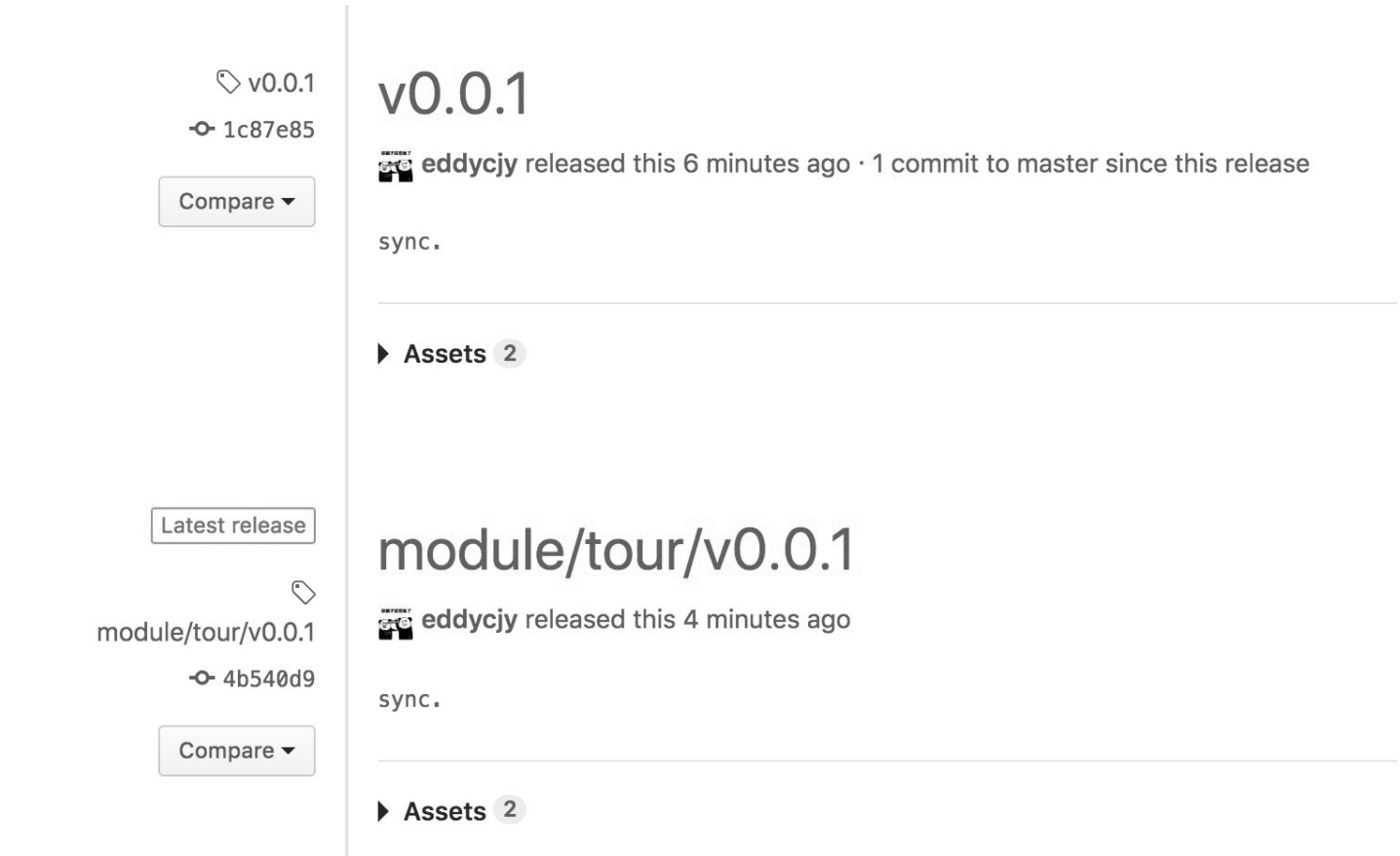
那么为什么会拉取的是 `v0.0.0` 呢，是因为 `github.com/eddycjy/mquote` 没有发布任何的tag，如下：



因此它默认取的是主分支最新一次 commit 的 commit 时间和 commithash，也就是 `20200220041913-e066a990ce6f`，属于第二种情况。

有发布 tags

在项目有发布 tags 的情况下，还存在着多种模式，也就是只有单个模块和多个模块，我们统一以多个模块来进行展示，因为多个模块的情况下就已经包含了单个模块的使用了，如下图：



在这个项目中，我们一共打了两个tag，分别是：v0.0.1 和 module/tour/v0.0.1。这时候你可能会奇怪，为什么要打 `module/tour/v0.0.1` 这么“奇怪”的tag，这有什么用意吗？

其实是 Go modules 在同一个项目下多个模块的tag表现方式，其主要目录结构为：

```
mquote
├── go.mod
├── module
│   └── tour
│       ├── go.mod
│       └── tour.go
└── quote.go
```

可以看到在 `mquote` 这个项目的根目录有一个 go.mod 文件，而在 `module/tour` 目录下也有一个 go.mod 文件，其模块导入和版本信息的对应关系如下：

tag	模块导入路径	含义
v0.0.1	github.com/eddycjy/mquote	mquote 项目的v 0.0.1 版本
module/tour/v0.01	github.com/eddycjy/mquote/module/tour	mquote 项目下的子模块 module/tour 的 v0.0.1 版本

导入主模块和子模块

结合上述内容，拉取主模块的话，还是照旧执行如下命令：

```
$ go get github.com/eddycjy/mquote@v0.0.1
go: finding github.com/eddycjy/mquote v0.0.1
go: downloading github.com/eddycjy/mquote v0.0.1
go: extracting github.com/eddycjy/mquote v0.0.1
```

如果是想拉取子模块，执行如下命令：

```
$ go get github.com/eddycjy/mquote/module/tour@v0.0.1
go: finding github.com/eddycjy/mquote/module v0.0.1
go: finding github.com/eddycjy/mquote/module/tour v0.0.1
go: downloading github.com/eddycjy/mquote/module/tour v0.0.1
go: extracting github.com/eddycjy/mquote/module/tour v0.0.1
```

我们将主模块和子模块的拉取进行对比，你会发现子模块的拉取会多出一步，它会先发现 `github.com/eddycjy/mquote/module`，再继续推算，最终拉取到 `module/tour`。

Go Modules的导入路径说明

不同版本的导入路径

在前面的模块拉取和引用中，你会发现我们的模块导入路径就是 `github.com/eddycjy/mquote` 和 `github.com/eddycjy/mquote/module/tour`，似乎并没有什么特殊的。

其实不然，实际上 Go modules 在主版本号为 v0 和 v1 的情况下省略了版本号，而在主版本号为v2及以上则需要明确指定出主版本号，否则会出现冲突，其tag与模块导入路径的大致对应关系如下：

tag	模块导入路径
v0.0.0	github.com/eddycjy/mquote
v1.0.0	github.com/eddycjy/mquote

tag	模块导入路径
v2.0.0	github.com/eddycjy/mquote/v2
v3.0.0	github.com/eddycjy/mquote/v3

简单来讲，就是主版本号为 v0 和 v1 时，不需要在模块导入路径包含主版本的信息，而在 v1 版本以后，也就是 v2 起，必须要在模块的导入路径末尾加上主版本号，引用时就需要调整为如下格式：

```
import (
    "github.com/eddycjy/mquote/v2/example"
)
```

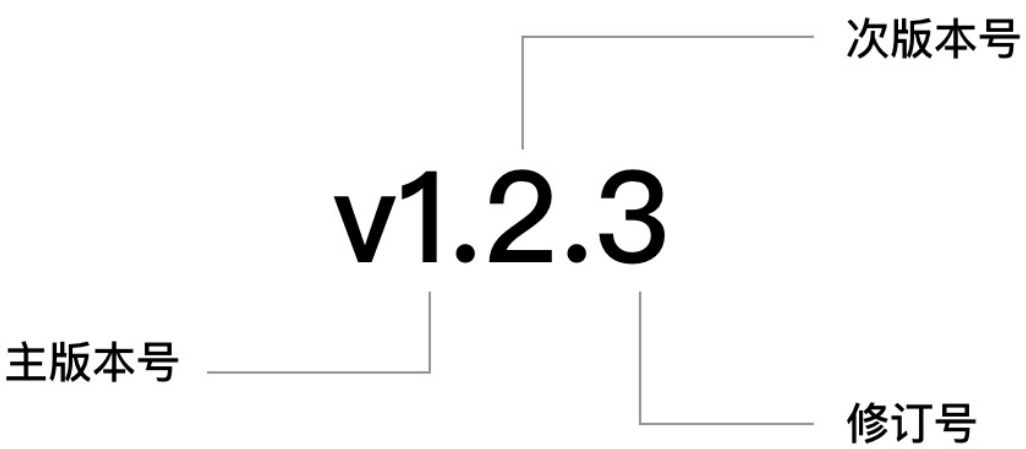
另外忽略主版本号 v0 和 v1 是强制性的（不是可选项），因此每个软件包只有一个明确且规范的导入路径。

为什么忽略v0和v1的主版本号

1. 导入路径中忽略 v1 版本的原因是：考虑到许多开发人员创建一旦到达 v1 版本便永不改变的软件包，这是官方所鼓励的，不认为所有这些开发人员在无意发布 v2 版时都应被迫拥有明确的 v1 版本尾缀，这将导致 v1 版本变成“噪音”且无意义。
2. 导入路径中忽略了 v0 版本的原因是：根据语义化版本规范，v0的这些版本完全没有兼容性保证。需要一个显式的 v0 版本的标识对确保兼容性没有多大帮助。

Go Modules的语义化版本控制

我们不断地在 Go Modules 的使用中提到版本号，其实质上被称为“语义化版本”，假设我们的版本号是 v1.2.3，如下：



其版本格式为“主版本号.次版本号.修订号”，版本号的递增规则如下：

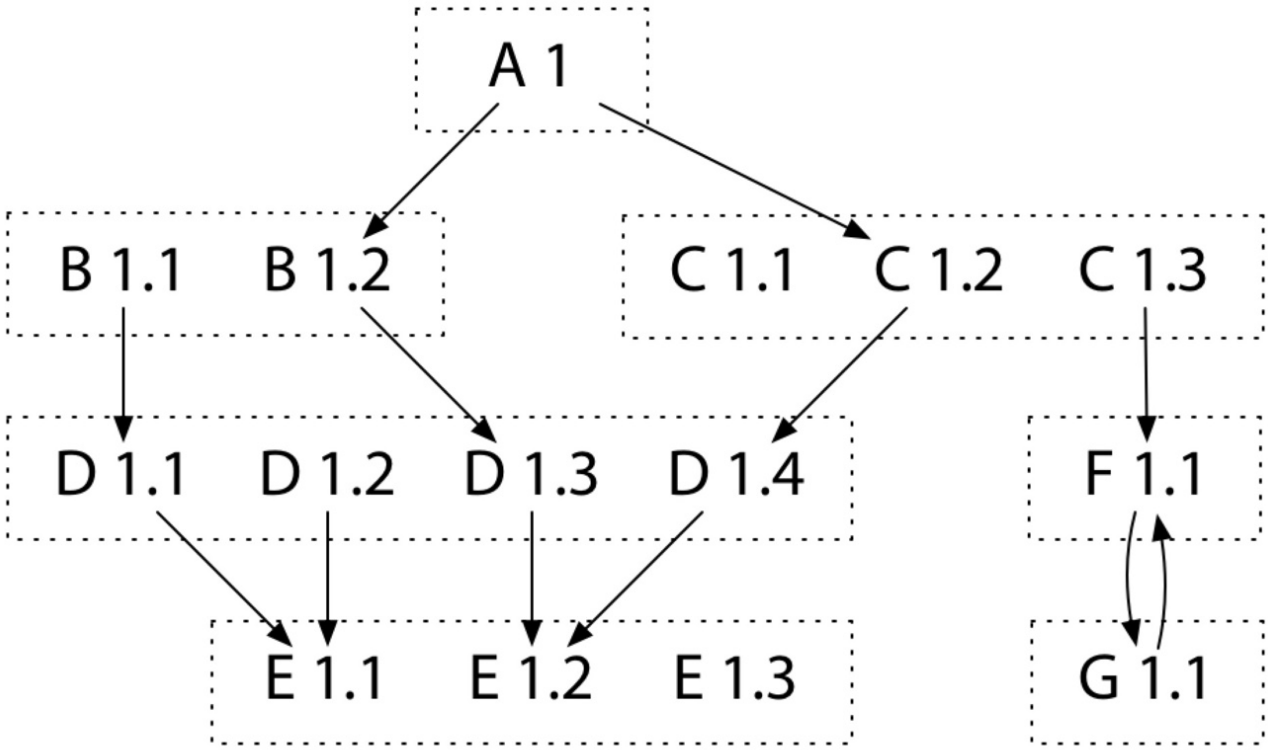
1. 主版本号：当你做了不兼容的 API 修改。
2. 次版本号：当你做了向下兼容的功能性新增。
3. 修订号：当你做了向下兼容的问题修正。

假设你是先行版本号或特殊情况，可以将版本信息追加到“主版本号.次版本号.修订号”的后面，作为延伸，如下：

至此我们介绍了 Go modules 所支持的两类版本号方式，在我们发布新版本打 tag 的时候，需要注意遵循，否则不遵循语义化版本规则的版本号都是无法进行拉取的。

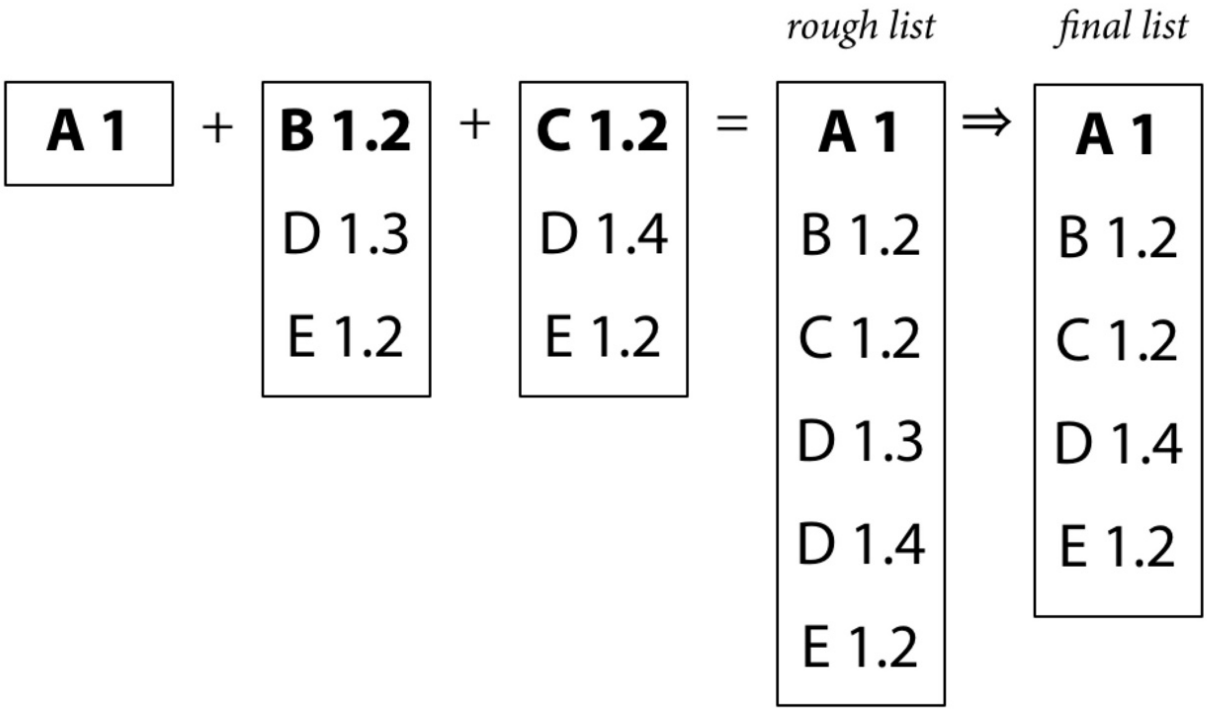
Go Modules的最小版本选择

现在我们已经有一个模块，也有发布的 tag，但是一个模块往往依赖着许多其它许许多多的模块，并且不同的模块在依赖时很有可能会出现依赖同一个模块的不同版本，如下图（来自Russ Cox）：



在上述依赖中，模块 A 依赖了模块 B 和模块 C，而模块 B 依赖了模块 D，模块 C 依赖了模块 D 和 F，模块 D 又依赖了模块 E，而且同模块的不同版本还依赖了对应模块的不同版本。那么这个时候 Go modules 怎么选择版本，选择的是哪一个版本呢?

我们根据 proposal 可得知，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（来自Russ Cox）：



我们看到 rough list 和 final list，两者的区别在于重复引用的模块 D（v1.3、v1.4），其最终清单选用了模块 D 的 v1.4 版本，主要原因：

1. 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 必须是要向下兼容 v1.3 版本，因此认为不存在破坏性变更，也就是兼容的。
2. 模块导入路径的规范：主版本号不同，模块的导入路径不一样，因此若出现不兼容的情况，其主版本号会改变，模块的导入路径自然也就改变了，因此不会与第一点的基础相冲突。

go.sum文件要不要提交

理论上 go.mod 和 go.sum 文件都应该提交到你的 Git 仓库中去。

假设我们不上传 go.sum 文件，就会造成每个人执行 Go modules 相关命令，又会生成新的一份 go.sum，也就是会重新到上游拉取，再拉取时有可能就是被篡改过的了，会有很大的安全隐患，失去了与基准版本（第一个所提交的人，所期望的版本）的校验内容，因此 go.sum文件是需要提交。

总结

至此我们介绍了 Go modules 的前世今生、基本使用和在 Go modules 模式下 `go get` 命令的行为转换，同时我们对常见的多版本导入路径、语义化版本控制以及多模块的最小版本选择规则进行了大致的介绍。

Go modules 的成长和发展经历了一定的过程，如果你是刚接触的读者，直接基于 Go modules 的项目开始即可，如果既有老项目，那么是时候考虑切换过来了，Go1.14起已经准备就绪，并推荐你使用。

我的公众号

参考

- wiki/Modules
- wiki/vgo
- proposal
- 干货满满的 Go Modules 和 goproxy.cn

阅读 2k 更新于 6月7日