

18 | 依赖管理：Go Module 用法与原理

2022-11-19 郑建勋 来自北京



天下无鱼

<https://shikey.com/>

课程介绍 >

《Go进阶·分布式爬虫实战》



讲述：郑建勋

时长 20:06 大小 18.36M



你好，我是郑建勋。

由于项目一开始就需要涉及到依赖的管理，因此在下一节课正式书写项目代码前，我们先来看一看和依赖管理的一些重要知识。

我们知道，一个大型程序会引入大量必要的第三方库，这就让这个程序形成了复杂的依赖关系网络。这种复杂性可能引发一系列问题，例如**依赖过多**、**多重依赖**、**依赖冲突**、**依赖回圈**等。因此，需要有专门的工具对程序的依赖进行管理。

Go 语言中的依赖管理经历了长时间的演进过程。在 Go1.5 之前，Go 官方通过 GOPATH 对依赖进行管理。但是由于 GOPATH 存在诸多问题，社区和官方尝试了诸多新的依赖管理工具，中间出现的 Godep、Glide、Vendor 等工具都不如人意，最终笑到最后的是在 Go 1.11 后引入，并在 Go 1.13 之后成为 Go 依赖管理主流的 Go Modules。

让我们先来看看 GOPATH 和它的不足之处。

GOPATH



什么是 GOPATH?

在 Go 1.8 及以上版本中，如果用户不指定 GOPATH，GOPATH 的路径就是默认的。我们可以通过输入 `go env` 或者 `go env GOPATH` 查看 GOPATH 的具体配置：

复制代码

```
1 C:\\Windows\\system32> go env
2 set GOPATH=C:\\Users\\jackson\\go
3 ...
```

GOPATH 的路径在 Mac 和 Linux 操作系统下为 `$HOME/go`，而在 Windows 操作系统下为 `%USERPROFILE%\\go`。我们可以把 GOPATH 可以理解为 Go 语言的工作空间，它内部存储了 `src`、`bin`、`pkg` 三个文件夹：

复制代码

```
1 go/
2 |— bin
3 |— pkg
4 |— src
```

`$GOPATH/bin` 目录下存储了通过 `go install` 安装的二进制文件。操作系统使用 `$PATH` 环境变量来查找不需要完整路径即可执行的二进制应用程序。建议将 `$GOPATH/bin` 目录添加到全局 `$PATH` 变量中。

`$GOPATH/pkg` 目录下会有一个文件夹（文件名根据操作系统的不同而有所不同，例如在 Mac 操作系统下为 `darwin_amd64`）存储预编译的 `obj` 文件，以加快程序的后续编译。大多数开发人员不需要访问此目录。我们在后面还会看到，`pkg` 下的 `mod` 文件会存储 Go Modules 的依赖。

`$GOPATH/src` 目录下存储项目的 Go 代码。通常包含多个由 Git 管理的存储库，每个存储库中都包含一个或多个 `package`，每个 `package` 有多个目录，每个目录下都包含一个或多个 Go 源文件。整个路径看起来是下面的样子：



```
1 go/
2 |— bin
3   |— main.exe
4 |— pkg
5   |— darwin_amd64
6   |— mod
7 |— src
8   |— github.com
9     |— tylfin
10      |— dynatomic
11      |— geospy
12      |— uudashr
13      |— gopkgs
14   |— golang.org
15     |— x
16       |— tools
```

在 Go 的早期版本中，可以使用 `go get` 指令从 GitHub 或其他地方获取 Go 项目代码。这时程序会默认将代码存储到 `$GOPATH/src` 目录下。例如，在拉取 `go get github.com/dreamerjackson/theWayToGolang` 时，目录结构如下所示：

```
1 go/
2 |— bin
3 |— pkg
4 |— src
5   |— github.com
6     |— dreamerjackson
7       |— theWayToGolang
```

当我们使用 `go get -u xxx` 时，会将该项目以及项目所依赖的所有其他项目一并下载到 `$GOPATH/src` 目录下。

在 `GOPATH` 模式下，如果我们在项目中导入了一个第三方包，例如 `import "github.com/gobuffalo/buffalo"`。

那么，实际引用的是 `$GOPATH/src/github.com/gobuffalo/buffalo` 文件中的代码。

GOPATH 的落幕与依赖管理的历史

GOPATH 借鉴了谷歌内部使用的 Blaze 系统。在 Blaze 中，所有项目的源代码共用一个代码仓库。go get 仅仅需要获取 Master 分支上最新的代码，不需要指定依赖的版本。



GOPATH 这种版本管理方式配置简单，容易理解，在谷歌内部不会出现问题。但是在开源领域，一个 GOPATH 走天下的情况就行不通了。由于依赖的第三方包总是在变，而且还没有严格的约束，直接拉取外部包的最新版本时，甚至可能出现一更新依赖代码都编译不过的情况。因此，我们迫切需要新的依赖管理工具。

2015 年，Go1.5 版本首次实验性地加入了 Vendor 机制。项目的所有第三方依赖都可以存放在当前项目的 Vendor 目录下，再也不用为了应用不同版本的依赖对 GOPATH 环境变量“偷梁换柱”了，Go 编译器优先感知和使用 Vendor 目录下缓存的第三方包。

但即便有了 Vendor 的支持，Vendor 内第三方依赖包代码的管理依旧不规范，要么需要手动处理，要么是借助 Godep 这样的第三方包管理工具。在这期间，社区也出现了大量的依赖管理工具，有点乱花渐欲迷人眼的态势。直到 Go Modules 出现，一锤定音。

2018 年初，Go team 的技术负责人 Russ Cox 在 [博客](#) 上连续发表 [七篇文章](#)，系统阐述了新的包依赖管理工具：[vgo](#)。vgo 的主要思路包括：[语义版本控制](#)、[最小版本选择](#)、[引入 Go Modules](#)等。

2018 年 5 月，Russ Cox 的 [提案](#)被接收，此后 vgo 代码被并入了 Go 主干，并正式命名为 Modules。

2018 年 8 月，Go 1.11 发布，Modules 作为官方试验一同发布。

2019 年 9 月，Go1.13 发布，只要目录下有 go.mod 文件，Go 编译器都会默认使用 Modules 来管理依赖。同时，新版本还添加了 GOPROXY、GOSUMDB、GOPRIVATE 等多个与依赖管理有关的环境变量。

Go Modules

在好的依赖管理出现之前，有一些问题长期困扰 Go 语言的开发人员：

- 能不能让 Go 工程代码脱离 GOPATH？
- 能否处理版本依赖问题并且自动选择最兼容的依赖版本？

- 能否在本地管理依赖项，自定义依赖项？

Go Modules 巧妙解决了上面这些问题。



import 路径问题

在 GOPATH 中，“导入路径”与“项目在文件系统中的目录结构和名称”必须是匹配的。但是，如果我们希望项目的实际路径和导入路径不同，例如 import 路径为 `github.com/gobuffalo/buffalo`，我们希望项目的实际路径在另一个任意的文件目录下 (例如，`/users/gobuffalo/buffalo`)，这个期待能否实现呢？

答案是肯定的。Go Modules 可以通过在 `go.mod` 文件中指定模块名来解决这一问题。`go.mod` 文件如下所示：

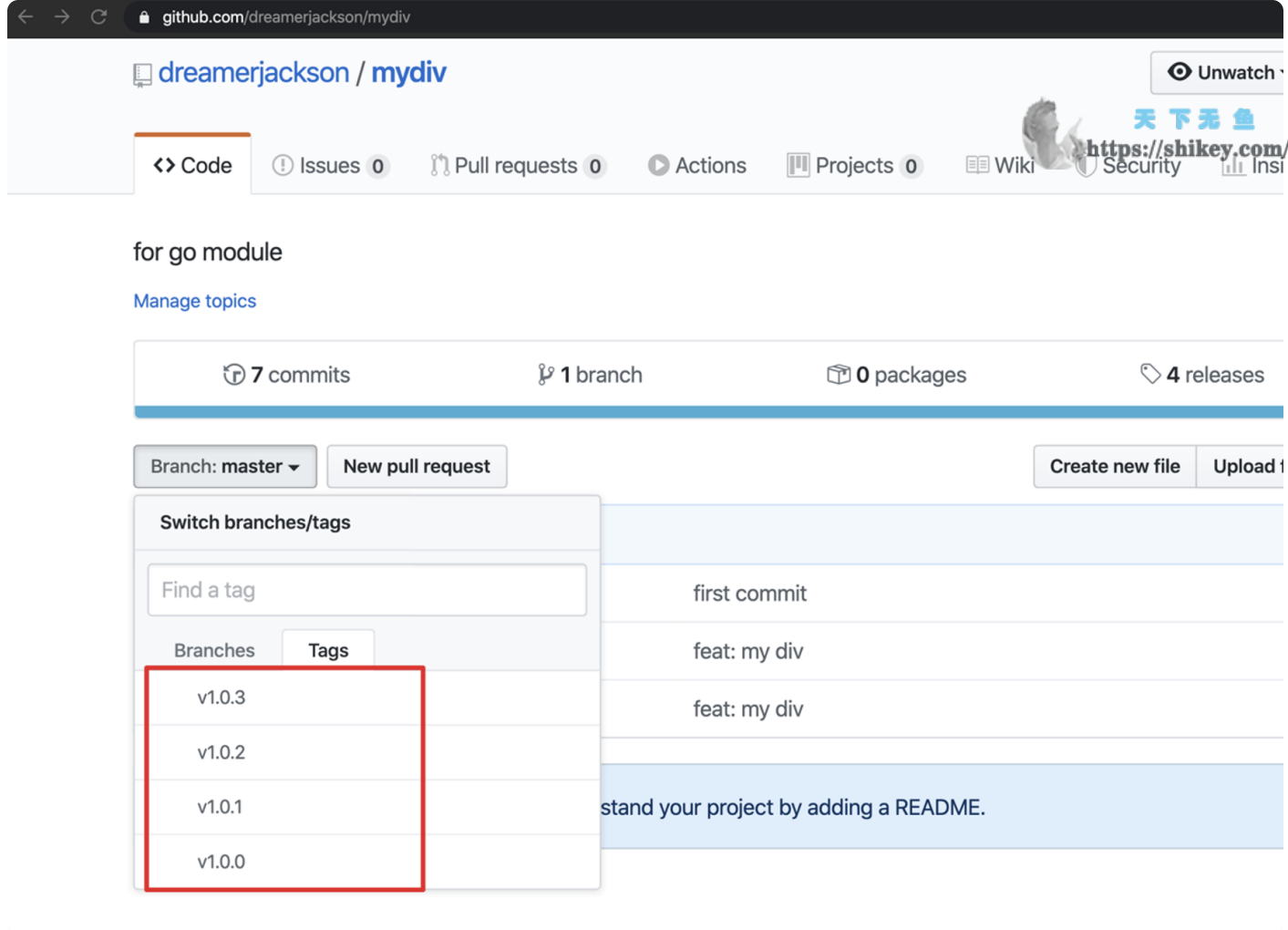
```
1  ## go.mod
2  01 module github.com/gobuffalo/buffalo
3  02
4  ...
5  06
```

复制代码

`go.mod` 文件的第一行指定了模块名，开发人员可以用模块名引用当前项目中任何 `package` 的路径名。这样，无论当前项目路径在什么位置，都可以使用模块名来解析代码内部的 `import`。

代码捆绑和版本控制问题

对于任何版本控制（VCS，version control system）工具，我们都能在任何提交的 `commit` 处打上 `tag` 标记。



开发人员可以使用 **VCS** 工具引用特定标签，将软件包的任何指定版本克隆到本地。当我们引用一个第三方包时，出于测试等不同的目的，可能并不总是希望应用项目最新的代码，而是想要应用某一个特定的，与当前项目兼容的代码。

对于特定第三方库来说，维护者可能并没有意识到有人在使用他们的代码，或者代码库由于某种原因进行了巨大的不兼容更新。因此，我们希望能够明确使用的第三方包的版本，这样才能完成可重复的构建，并且希望能够自动下载、管理依赖包。一个依赖管理工具至少需要考虑下面几个问题。

- 如何查找并把所有的依赖包下载下来？
- 某一个包下载失败怎么办？
- 所有项目之间如何进行依赖的传导？
- 如何选择一个最兼容的包？
- 如何解决包的冲突？
- 如何在项目中同时引用第三方包的两个不同版本？

因此，只通过 **GOPATH** 维护单一的 **Master** 包是远远不够的，Go 官方的 **Go Modules** 提供了一种可以在文件中同时维护直接和间接依赖项的集成解决方案。一个特定版本的依赖项也被叫做一个模块（module），一个模块是一系列指定版本的 **package** 的集合。



为了加快构建程序的速度，快速切换、获取项目中依赖项的更新，Go 维护了下载到本地计算机上的所有模块的缓存，缓存目前默认位于 **\$GOPATH/pkg** 目录下：

复制代码

```
1 go/
2 |— bin
3 |— pkg
4   |— darwin_amd64
5   |— mod
6 |— src
7
```

在 **mod** 目录下，我们能够看到模块名路径中的第一部分即为顶级文件夹，如下所示：

复制代码

```
1 ~/go/pkg/mod » ls -l
2 drwxr-xr-x   6 jackson  staff   192  1 15 20:50 cache
3 drwxr-xr-x   7 jackson  staff   224  2 20 17:50 cloud.google.com
4 drwxr-xr-x   3 jackson  staff    96  2 18 12:03 git.apache.org
5 drwxr-xr-x 327 jackson  staff 10464  2 28 00:02 github.com
6 drwxr-xr-x   8 jackson  staff   256  2 20 17:27 gitlab.followme.com
7 drwxr-xr-x   6 jackson  staff   192  2 19 22:05 go.etcd.io
8 ...
```

当我们打开一个实际的模块时（例如 **github.com/nats-io**），会看到许多与 **NATS** 库有关的模块及其版本：

复制代码

```
1 ~/go/pkg/mod » ls -l github.com/nats-io
2 total 0
3 dr-x----- 24 jackson  staff   768  1 17 10:27 gnatsd@v1.4.1
4 dr-x----- 15 jackson  staff   480  2 17 22:22 go-nats-streaming@v0.4.0
5 dr-x----- 26 jackson  staff   832  2 19 22:05 go-nats@v1.7.0
6 dr-x----- 26 jackson  staff   832  1 17 10:27 go-nats@v1.7.2
7 ...
```


为了拥有一个干净的工作环境，我们可以用下面的指令清空缓存区。但是要注意，在正常的工作流程中，是不需要执行这段代码的：



 复制代码

```
1 $ go clean -modcache
```

Go Modules 实践

我们从 GOPATH 外开始一个新的项目讲解 **go modules** 的使用方法和原理。首先新建一个文件夹和一个 main.go 文件：

 复制代码

```
1 $ cd $HOME
2 $ mkdir mathlib
3 $ cd mathlib
4 $ touch main.go
```

接着在当前目录下，执行下面这段指令，初始化 module：

 复制代码

```
1 ~/mathlib » go mod init github.com/dreamerjackson/mathlib
```

go mod init 指令的功能很简单，即自动生成一个 go.mod 文件，后面紧跟的路径就是自定义的模块名。习惯上，我们会以托管代码仓库的 URL 作为模块名。go.mod 文件位于项目的根目录，内容如下所示，第一行就是模块名（此例中，代码放置在 [这个](#)目录下）。

 复制代码

```
1 module github.com/dreamerjackson/mathlib
2
3 go 1.13
```

go.mod 文件第三行指定了当前模块中使用的 Go 的版本，不同的 Go 版本可能对应不同的依赖管理行为。例如，Go1.17 之后，才会有通过 Go Modules 分支修剪来加速依赖拉取和编译的特性。如果你想深入了解不同的 Go 版本在依赖管理上的差异，可以阅读 [这篇文章](#)。

- 引入第三方模块

接下来，我们要书写初始化的代码片段。



复制代码

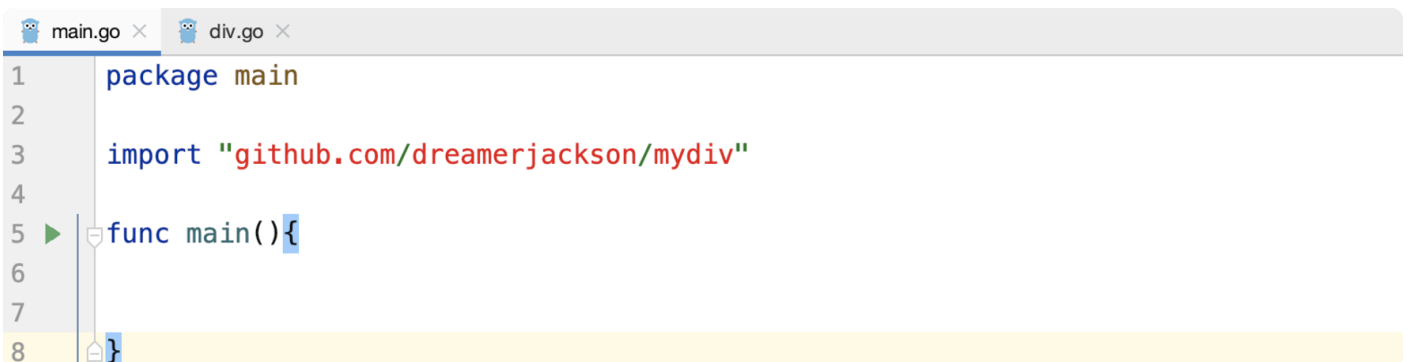
```
1 package main
2
3 import "github.com/dreamerjackson/mydiv"
4
5 func main(){
6
7 }
```

我在代码片段中导入了为了讲解 Go Modules 而特地引入的 `github.com/dreamerjackson/mydiv`，它的作用是进行简单的除法操作，同时我还引入了另一个包（`github.com/pkg/errors`）用于包装错误，代码如下：

复制代码

```
1 package mydiv
2
3 import "github.com/pkg/errors"
4
5 func Div(a int,b int) (int,error){
6     if b==0{
7         return 0,errors.Errorf("new error b can't = 0")
8     }
9     return a/b,nil
10 }
```

在 GoLand 中，我们可以看到导入的 `package` 是红色的，这是因为这个时候在 Go Modules 的缓存中找不到这个 `package`。



- 下载第三方模块

为了能够将项目依赖的 `package` 下载到本地，我们可以使用 `go mod tidy` 指令。



天下无鱼

<https://shikey.com/>

复制代码

```
1 $ go mod tidy
2 go: finding github.com/dreamerjackson/mydiv latest
3 go: downloading github.com/dreamerjackson/mydiv v0.0.0-20200305082807-fdd187670161
4 go: extracting github.com/dreamerjackson/mydiv v0.0.0-20200305082807-fdd187670161
```

执行完毕后，在 `go.mod` 文件中增加了一行，指定了引用的依赖库和版本。

复制代码

```
1 module github.com/dreamerjackson/mathlib
2
3 go 1.13
4
5 require github.com/dreamerjackson/mydiv v0.0.0-20200305082807-fdd187670161
```

注意，这里间接的依赖（即 `github.com/dreamerjackson/mydiv` 依赖的 `github.com/pkg/errors`）没有在 `go.mod` 文件中展示出来，而是在一个自动生成的新文件 `go.sum` 中进行了指定。

- 使用第三方模块

一切就绪之后，现在，我们可以愉快地调用第三方代码了。

复制代码

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/dreamerjackson/mydiv"
6 )
7
8 func main(){
9     res,_ :=mydiv.Div(4,2)
10    fmt.Println(res)
11 }
```

运行 `go run` 命令后，我们可以得到除法结果。



- 手动更新第三方模块

假设我们依赖的第三方包出现了更新怎么办？如何将依赖代码更新到最新的版本呢？

有多种方式可以实现依赖模块的更新，我们需要在 `go.mod` 文件中修改版本号为：

```
1 require github.com/dreamerjackson/mydiv latest
```

复制代码

或者：

```
1 require github.com/dreamerjackson/mydiv master
```

复制代码

或者将指定 `commit id` 复制到末尾：

```
1 require github.com/dreamerjackson/mydiv c9a7ffa8112626ba6c85619d7fd98122dd49f8
```

复制代码

还有一种办法是在终端的当前项目中，运行 `go get github.com/dreamerjackson/mydiv`。

使用上面任一方式保存文件后，再次运行 `go mod tidy`，版本即会进行更新。这个时候如果我们再打开 `go.sum` 文件会发现，`go.sum` 中不仅存储了直接和间接的依赖，还存储了过去的版本信息。

```
1 github.com/dreamerjackson/mydiv v0.0.0-20200305082807-fdd187670161 h1:QR1fJ05yj
2 github.com/dreamerjackson/mydiv v0.0.0-20200305082807-fdd187670161/go.mod h1:h7
3 github.com/dreamerjackson/mydiv v0.0.0-20200305090126-c9a7ffa81126/go.mod h1:h7
4 github.com/pkg/errors v0.9.1 h1:FEbLx1zS214owpjy7qsBeixbURkuhQAwRk5UwLGTwt4=
5 github.com/pkg/errors v0.9.1/go.mod h1:bwawxfHBFNV+L2hUp1rHADufV3IMtnDRdf1r5NIN
```

复制代码

- **replace 指令**

Go Modules 中还提供了其他的功能，除了 **require**，还包括了 **replace**、**exclude**、**retract** 等指令。**replace** 指令可以将依赖的模块替换为另一个模块，例如由公共库替换为内部私有仓库，如下所示。**replace** 还可以用于本地调试场景，这时我们可以将依赖的第三方库替换为本地代码，便于进行本地调试。

 复制代码

```
1 replace golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5
2
3 replace (
4     golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5
5     golang.org/x/net => example.com/fork/net v1.4.5
6     golang.org/x/net v1.2.3 => ./fork/net
7     golang.org/x/net => ./fork/net
8 )
```

- **exclude 指令**

有时我们希望排除某一模块特定的版本，这时就需要用到 **exclude** 指令了。如果当前项目中，**exclude** 指令与 **require** 指令对应的版本相同，那么 **go get** 或 **go mod tidy** 指令将查找高一级的版本。

 复制代码

```
1 exclude golang.org/x/net v1.2.3
2
3 exclude (
4     golang.org/x/crypto v1.4.5
5     golang.org/x/text v1.6.7
6 )
```

- **retract 指令**

retract 撤回指令表示不依赖指定模块的版本或版本范围。当版本发布得太早，或者版本发布之后发现严重问题时，撤回指令就很有用了。例如，对于模块 **example.com/m**，假设我们错误地发布了 **v1.0.0** 版本后想要撤销。这时，我们就需要发布一个新的版本，tag 为 **v1.0.1**。

```
1 retract (  
2     v1.0.0  
3     v1.0.1  
4 )
```

[复制代码](#)[天下无鱼](#)<https://shikey.com/>

然后，我们要执行 `go get example.com/m@latest`，这样，依赖管理工具读到最新的版本 `v1.0.1` 是撤回指令，而且发现 `v1.0.0` 和 `v1.0.1` 都被撤回了，`go` 命令就会降级到下一个最合适的版本，比如 `v0.9.5` 之类的。除此之外，`retract` 指令还可以指定范围，更灵活地撤回版本。

```
1 retract v1.0.0  
2 retract [v1.0.0, v1.9.9]  
3 retract (  
4     v1.0.0  
5     [v1.0.0, v1.9.9]  
6 )
```

[复制代码](#)

• 依赖移除

当我们不想使用此第三方包时，可以直接在代码中删除无用的代码，接着执行 `go mod tidy`，会发现 `go.mod` 和 `go.sum` 又空空如也了。

Go Modules 最小版本选择原理

明白了 Go Modules 的使用方法，接下来我们来看一看 Go Modules 在复杂情况下的版本选择原理。

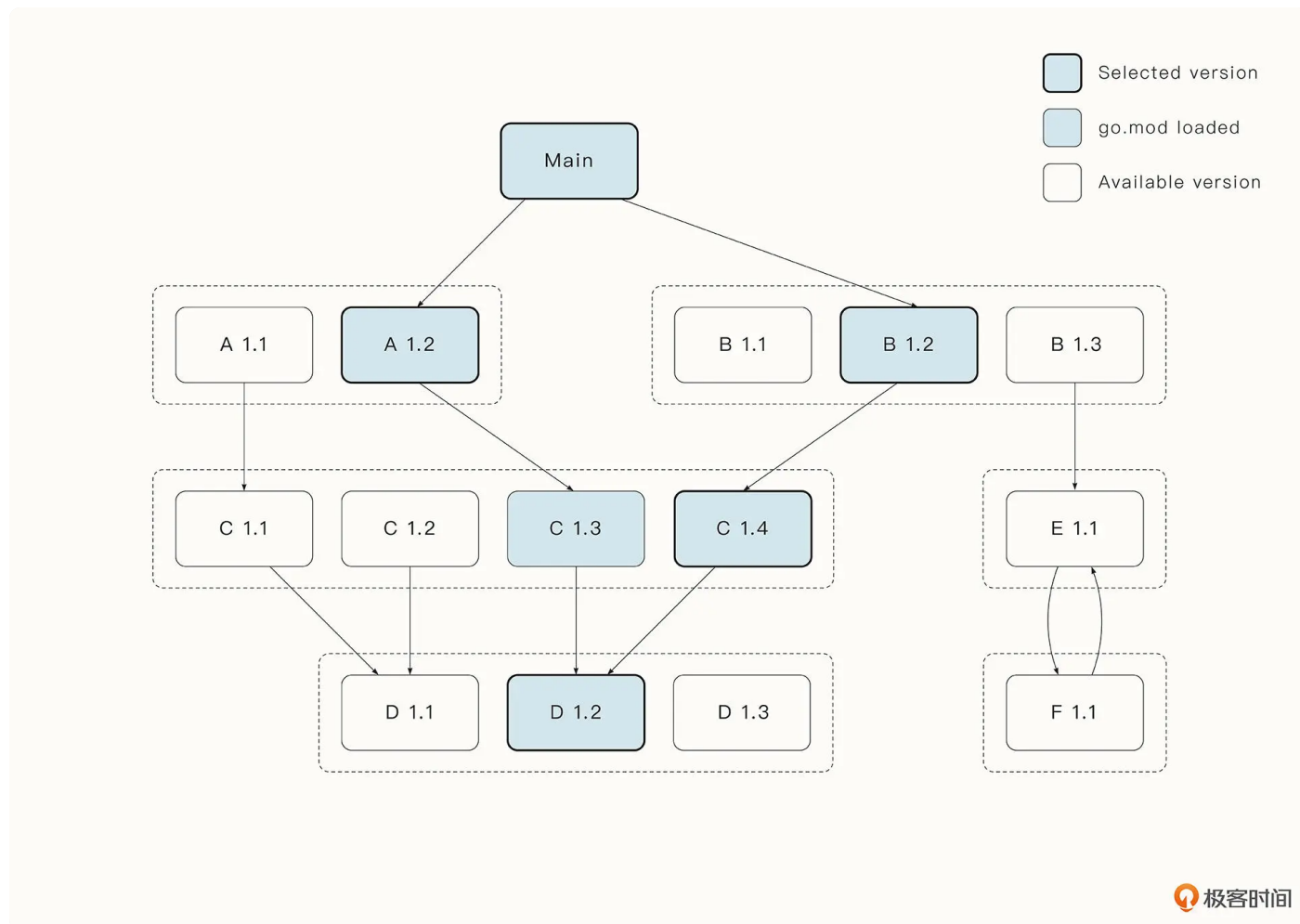
每个依赖管理解决方案都必须解决选择哪个依赖版本的问题。当前许多版本选择算法都倾向于选择依赖的最新版本。如果人们能够正确应用语义版本控制并且遵守约定，那么这是有道理的。在这些情况下，依赖项的最新版本应该是最稳定和最安全的，而且，它还应该和较早版本有很好的兼容性。

但是，Go 语言采用了其他方法，Go Team 技术负责人 Russ Cox 花费了大量时间和精力撰写和谈论 Go 的版本选择算法，即 [最小版本选择](#)（Minimal Version Selection, MVS）。Go 团队相信 MVS 可以更好地为 Go 程序提供兼容性和可重复性。

那么什么是最小版本选择原理呢？

Go 最小版本选择指的是，在选择依赖的版本时，优先选择项目中最合适的最低版本。当然，并不是说 **MVS** 不能选择最新的版本，而是说如果项目中任何依赖都用不到最新的版本，那么我们本质上不需要它。

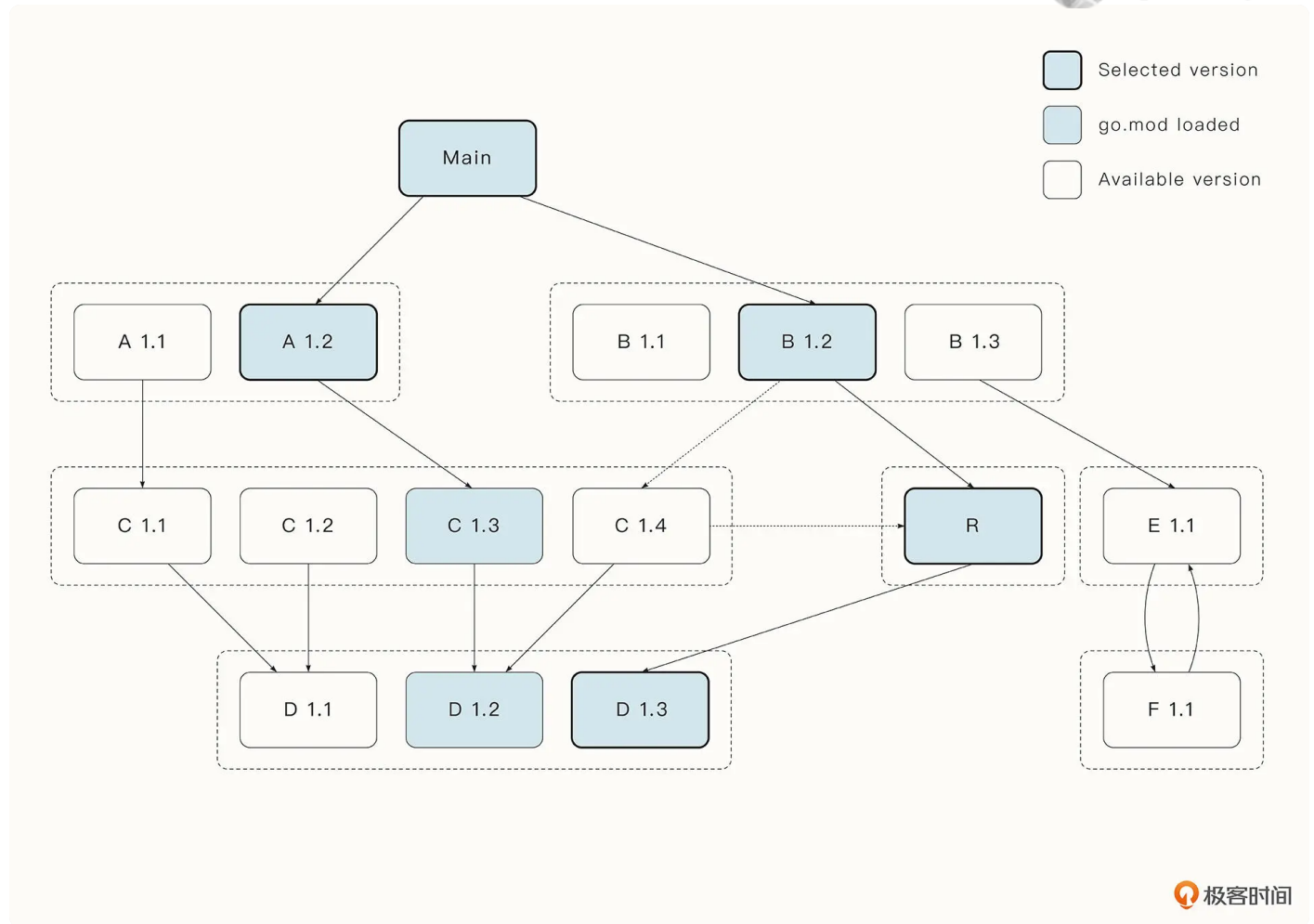
举个例子，项目 **Main** 依赖 **A 1.2** 版本以及 **B 1.2** 版本，而 **A 1.2** 版本依赖 **C1.3** 版本，**B1.2** 版本依赖 **C1.4** 版本，**C1.3** 与 **C1.4** 版本共同依赖了 **D1.2** 版本。



最终，我们选择的版本是项目导入的可以使用的最小版本，即 **A 1.2**、**B 1.2**、**C 1.4**、**D 1.2**。在这个例子中，虽然 **C 1.3**、**C 1.4** 分别被 **A**、**B** 两个包导入了，但是现在 **Go Modules** 认为最好的版本是这两个版本中最大的版本 **C 1.4**，因为 **C 1.4** 相对于 **C 1.3** 增加了接口等操作，如果选择 **C 1.3** 版本，可能出现编译都不通过的情况。而从语义版本控制的角度，默认 **C 1.4** 版本是向后兼容的。

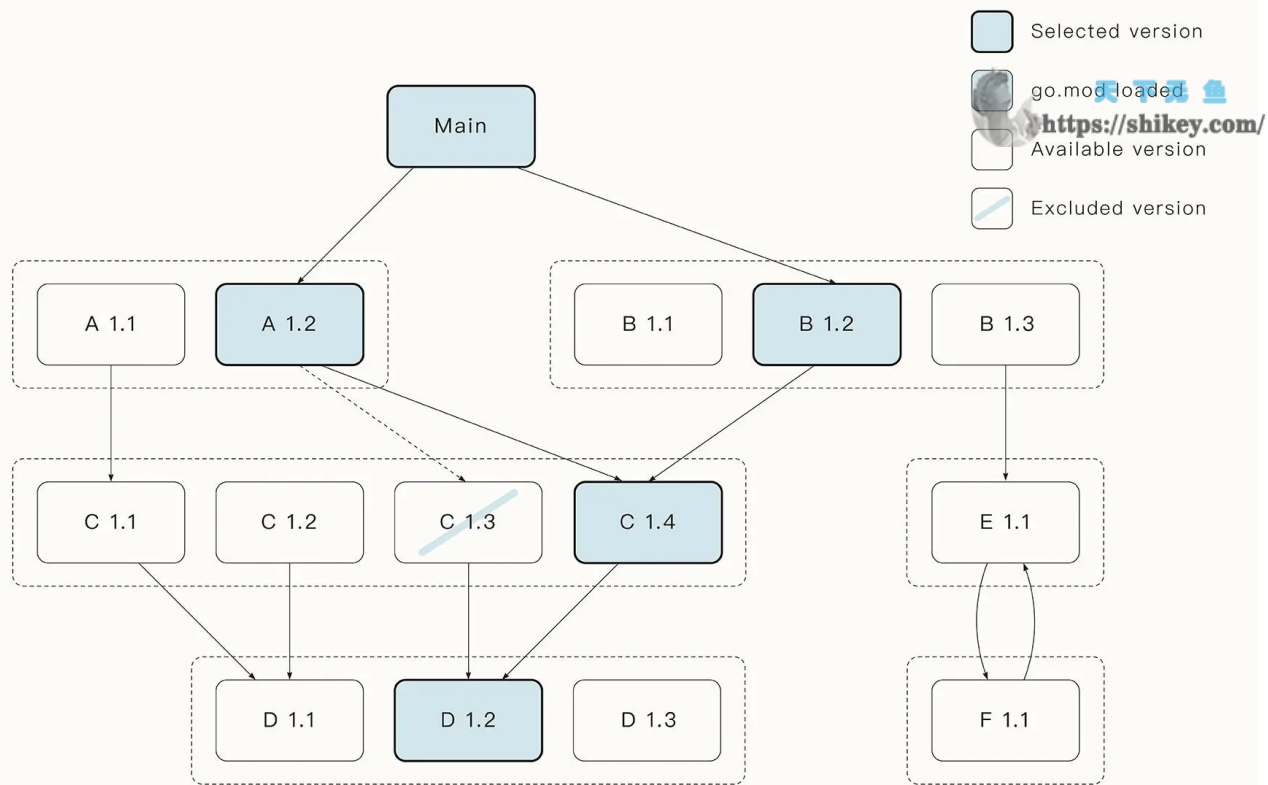
- **replace 指令与最小版本选择**

当项目中使用了`replace`指令，如下图所示，B1.2 依赖的 C1.4 `replace` 为了 R 模块，R 依赖 D1.3，最终最小版本算法选择的版本为 R、D 1.3 版本。



• Exclusion 指令与最小版本选择

当项目中使用了 `Exclusion` 指令时，如下图所示，当 C 1.3 被排除，A1.2 将表现为直接依赖比 C1.3 版本更高的 C1.4。



下面我举一个简单的例子来验证最小版本选择原理。

假设现在项目 D（即 github.com/dreamerjackson/mydiv 的最新版本）为 **v1.0.3**，模块可用的版本号可以通过下面这个指令查看：

```
1 > go list -m -versions github.com/dreamerjackson/mydiv
2 github.com/dreamerjackson/mydiv v1.0.0 v1.0.1 v1.0.2 v1.0.3
```

复制代码

现在有两个模块 A、B，它们都依赖模块 D，其中模块 A 引用了 D **v1.0.1** 版本，模块 B 引用了 D **v1.0.2** 版本。

如果我们当前的项目只依赖模块 A，这个时候 Go Modules 会如何选择版本呢？

像 dep 这样的依赖工具将选择 **v1.0.3**，即选择最新的语义版本。但是在 Go Modules 中，最小版本选择原理将选择 A 模块声明的版本，即 **v1.0.1**。这里就有下面两个问题了。

如果当前项目在之后又引入了模块 B 的新代码该怎么办呢？答案是，将模块 B 导入项目后，Go 会将项目中模块 D 的版本从 v1.0.1 升级到 v1.0.2，这符合最小版本选择原理。



那如果删除刚刚为模块 B 添加的代码，又会发生什么呢？Go 会将项目锁定到模块 D 的版本 v1.0.2 中。这是因为降级到版本 v1.0.1 将是一个更大的更改，而 Go 知道版本 v1.0.2 可以正常运行并且稳定，因此版本 v1.0.2 仍然是“最新版本”。

这是在不同情境下，我们预测最小版本选择原理会产生结果。到底对不对呢？我们来验证一下。我们以 github.com/dreamerjackson/mydiv 为例，其 v1.0.1 与 v1.0.2 版本的代码如下：

复制代码

```
1  ## v1.0.1
2  package mydiv
3  import "github.com/pkg/errors"
4  func Div(a int,b int) (int,error){
5      if b==0{
6          return 0,errors.Errorf("new error b can't = 0")
7      }
8      return a/b,nil
9  }
10
11 ## v1.0.2
12 package mydiv
13 import "github.com/pkg/errors"
14 func Div(a int,b int) (int,error){
15     if b==0{
16         return 0,errors.Errorf("new error b can't = 0")
17     }
18     return a/b,nil
19 }
```

接着模块 B（github.com/dreamerjackson/minidiv）引用了模块 D（github.com/dreamerjackson/mydiv）v1.0.1 版本。

复制代码

```
1  ## 模块B
2  package div
3
4  import (
5      "github.com/dreamerjackson/mydiv"
6  )
7
8  func Div(a int,b int) (int,error){
```

```
9     return mydiv.Div(a,b)
10 }
```



天下无鱼

<https://shikey.com/>

最后，我们把当前的项目叫做模块 **Now**，直接依赖模块 **D v1.0.2**，同时依赖模块 **B**，其代码如下：

复制代码

```
1 package main
2
3 import (
4     "fmt"
5     div "github.com/dreamerjackson/minidiv"
6     "github.com/dreamerjackson/mydiv"
7 )
8
9 func main(){
10     _,err1:= mydiv.Div(4,0)
11     _,err2 := div.Div(4,0)
12     fmt.Println(err1,err2)
13 }
14
```

当前的依赖关系如下：

复制代码

```
1 当前模块 --> 模块D v1.0.2
2 当前模块 --> 模块B --> 模块D v1.0.1
```

关键时刻到了，我们需要验证当前项目是不是如我们所料选择了模块 **D v1.0.2**。

验证方式有两种。第一种是直接运行，查看项目采用了哪一个版本的代码。

复制代码

```
1 $ go run main.go
2 v1.0.2 b can't = 0 v1.0.2 b can't = 0
```

通过上面这段代码可以看出，输出的结果全部是我们在模块 **D v1.0.2** 中定义的代码。

第二种方式是使用 `go list` 指令，得到的也是一样的结果：



```
1 ~/mathlib » go list -m all | grep mydiv
2 github.com/dreamerjackson/mydiv v1.0.2
```

第三种方式，我们可以使用“`go list -m -u all`”指令查看直接和间接模块的当前和最新版本。

复制代码

```
1 ~/mathlib » go list -m -u all | column -t
2 go: finding github.com/dreamerjackson/minidiv latest
3 github.com/dreamerjackson/mathlib
4 github.com/dreamerjackson/minidiv    v0.0.0-20200305104752-fcd15cf402bb
5 github.com/dreamerjackson/mydiv      v1.0.2                                [v1.0.3]
6 github.com/pkg/errors                v0.9.1
```

• 更新模块

更新模块可以使用 `go get` 指令，`go get` 指令有不少的参数可供选择。使用下面的命令，可以更新项目中所有的依赖模块为最新版本。（注意，除非你了解项目的所有细节，否则不要直接将所有模块调整到最新版本。）

复制代码

```
1 ~/mathlib » go get -u -t -v ./...
2 go: finding github.com/dreamerjackson/minidiv latest
3 go: downloading github.com/dreamerjackson/mydiv v1.0.3
4 go: extracting github.com/dreamerjackson/mydiv v1.0.3
```

解释一下，这里的 `-v` 表示提供详细输出，`-t` 表示会连带着下载指定模块中的测试包，`-u` 表示将指定模块更新到最新版本，`./...` 表明在整个源代码树中执行这些操作。

我们再次查看当前引用的版本，会发现模块 `github.com/dreamerjackson/mydiv` 已经被强制更新到了最新的 `v1.0.3` 版本。

复制代码

```
1 ~/mathlib » go list -m all | grep mydiv
2 github.com/dreamerjackson/mydiv v1.0.3
```

- 重置依赖关系

如果不满意所选的模块和版本，我们可以通过删除 `go.mod`、`go.sum` 中的依赖关系并再次运行 `go mod tidy` 来重置版本。当项目还不太成熟时，这是一种选择。

 复制代码

```
1 $ rm go.*
2 $ go mod init <module name>
3 $ go mod tidy
```

语义版本（Semantic Versioning）

为了标识一个模块的快照，Go Modules 使用了语义版本来管理模块。每个语义版本都采用 `VMAJOR.MINOR.PATCH` 的形式，我们简单介绍下各字段的含义。

- **v**：所有版本号都以 **v** 开头。
- **MAJOR**：主版本号。意味着有大的版本更新，一般会导致 **API** 和之前版本不兼容。
- **MINOR**：次版本号，添加了新的特性但是向后兼容。
- **PATCH**：修订版本号，用户做了向后兼容的 **bug** 修复。

如果两个版本具有相同的主版本号，那么预期更高版本则可以向后兼容较低版本。但是，如果两个版本的主版本号不同，那么它们之间就没有预期的兼容关系。因此，我们在上面的实例中可以看到，Go 判断 `v1.0.3` 与 `v1.0.1` 是兼容的，是因为他们有相同的主版本号 **1**。如果我们将版本升级到了 `v2.0.0`，则会被看作出现了重大更新，兼容关系不再成立。

下面这张图展示了 Go 处理版本更新的方法。`my/thing/v2` 标识的是特定模块的语义主版本 **2**。版本 **1** 是 `my/thing`，模块路径中没有明确的版本。但是，当用户引入主要版本 **2** 或更大版本时，必须在模块名称后添加版本，以区别于版本 **1** 和其他主要版本，所以版本 **2** 为 `my/thing/v2`，版本 **3** 为 `my/thing/v3`，依此类推。

Major version: increment for backwards-incompatible changes.

Minor version: increment for new features.

Patch version: increment for bug fixes.

v2.3.4

import "my/thing/v2/sub/pkg"

天下无鱼
<https://shikey.com/>

极客时间

v2 版本

假设模块 A 引入了模块 B 和模块 C，模块 B 引入了模块 D v1.0.0，模块 C 引入了模块 D v2.0.0。如下所示：

复制代码

```
1 A --> 模块B --> 模块D v1.0.0
2 A --> 模块C --> 模块D v2.0.0
```

由于 v1 和 v2 模块的路径不相同，所以他们是互不干扰的两个模块，可以共存。

下面我用实例来验证一下。首先给 github.com/dreamerjackson/mydiv 打一个 v2.0.0 的 tag，其代码如下，在 v2.0.0 中简单修改了返回的错误文字。

复制代码

```
1 package mydiv
2 import "github.com/pkg/errors"
3
4 func Div(a int, b int) (int, error) {
5     if b == 0 {
6         return 0, errors.Errorf("v2.0.0 b can't = 0")
7     }
8     return a/b, nil
9 }
```

同时，我们需要修改 v2 模块的路径名：



```
1 module github.com/dreamerjackson/mydiv/v2
```

接着在 mathlib 模块中书写代码如下：

复制代码

```
1 package main
2
3 import (
4     "fmt"
5     div "github.com/dreamerjackson/minidiv"
6     mydiv "github.com/dreamerjackson/mydiv/v2"
7 )
8
9 func main(){
10     _,err1:= mydiv.Div(4,0)
11     _,err2 := div.Div(4,0)
12     fmt.Println(err1,err2)
13 }
```

现在的依赖路径可以表示为：

- mathlib --> 依赖 mydiv v2
- mathlib --> 依赖 minidiv --> 依赖 mydiv v1

运行代码，可以看到两段代码是共存的。

复制代码

```
1 v2.0.0 b can't = 0 : : v1.0.1 b can't = 0
```

最后，我们接着执行 go list，进一步确认模块 v1 与 v2 是共存的，验证成功～

复制代码

```
1 ~/mathlib(master*) » go list -m all | grep mydiv
2 github.com/dreamerjackson/mydiv v1.0.1
3 github.com/dreamerjackson/mydiv/v2 v2.0.1
```


当我们引入了一个没有用语义版本管理的模块，或者我们希望用某一个特殊的 commit 快照进行测试时，导入模块的版本会是伪版本，例如：`v0.0.0-20191109021931-daa7c04131f5` 就是一个伪版本。伪版本包含了 3 个部分：

- 基本版本前缀：通常为 `vX.0.0` 或 `vX.Y.Z-0`。`vX.Y.Z-0` 表明该 commit 快照派生自某一个语义版本，`vX.0.0` 表明该 commit 快照找不到派生的语义版本；
- 时间戳：格式为“`yyyymmddhhmmss`”，它是创建 commit 的 UTC 时间；
- 最后是长度为 12 的 commit 号。

我们在 Go 代码库的设计中，应该严格遵守语义版本的规范，尽可能保证代码具有向后兼容性。我在实践中看到过很多依赖库由于代码不兼容带来了许多问题，`v1` 与 `v2` 版本如果同时存在也会增加理解的成本，这些都是我们要尽量避免的。

总结

我们这节课就讲到这里。

这节课，我们简单梳理了一下 Go 语言的依赖管理进程。在 Go1.5 之前，Go 官方主要使用 GOPATH 管理依赖，但是由于 GOPATH 存在很多问题，在后续的版本中，GOPATH 逐渐式微。Go1.13 之后，Go Modules 成为了 Go 项目依赖管理的主流方式。

Go Modules 提供了脱离 GOPATH 管理 Go 代码的方式。同时提供了代码捆绑、版本控制、依赖管理的功能，供全球开发人员使用、构建、下载、授权、验证、获取、缓存和重用模块。Go Modules 让世界各地的开发者能够始终得到完全相同的代码，而不管它被构建了多少次，从何处获取及由谁获取。

写在最后

最近，我们收到了一些同学对于一门实战课程还未看到代码的困惑，这里我统一做一个回复。

我们的这个专栏，内容比较庞大。想要交付给大家的不只是一个实战项目，更会介绍为什么要这样去写代码。所以在专栏的前期，我做了一些铺垫。回顾一下目前为止的知识：

- 1-3 讲，我们回顾了 Go 的基础知识，也看到了一条完整的进阶学习路线；
- 4-5 讲，我们介绍了整个大型项目的流程，看到了大型企业项目流程的全貌；
- 6-7 讲，我们介绍了爬虫的内涵，还聊了一个重要的话题，为什么 Go 适合爬虫网络项目？
- 8-13 讲，我介绍了系统设计的三个主要问题，高性能、微服务、分布式，以此推导出了 14 讲的内容，也就是项目的功能、设计架构以及为什么要这样去设计；
- 紧接着，我们来到了 Worker 开发篇，在 15 讲，我介绍了项目的编程规范，这是我们从会写代码到写好代码，从单兵作战到团队协作需要遵守的原则与规范；
- 16-17 讲，我们看到了爬虫的核心，一次网络爬取的流程。如果你把 7、16、17 结合起来看，你就能回答一个重要的问题，知道 Go 语言的 HTTP 请求流程是怎样的，以及 Go 为什么适合开发网络服务，同时，我们后面的开发也依赖于这样的知识；
- 18 讲，也就是这一讲，我详细论述了依赖管理，因为一个项目一开始就需要它。
- 下一讲，我会开始书写项目的第一行代码。

天下无鱼
<https://shikey.com/>

可以看到，我想打造的这个专栏拥有一条比较全面、完整的逻辑链，前面这些篇幅都是围绕着整个项目展开的。我的教育理念和课程设计可能并不能满足每个人的需求，但是相信很快你会看到自己想要看到的内容。这些前期的知识，后面回过头来看，也许别有一番滋味。

在未来，我们也会在不违背课程设计初衷的前提下做更多实战侧的调整。也期待过段时间，还能够看到你们的反馈。


课后题

最后，我也给你留一道思考题。

你认为应该如何解决 Go 项目常常遇到的循环依赖问题？

欢迎你在留言区与我交流讨论，我们下节课再见！

分享给需要的人，Ta 购买本课程，你将得 20 元

 生成海报并分享

上一篇 17 | 巨人的肩膀：HTTP协议与Go标准库原理

下一篇 19 | 从正则表达式到CSS选择器：4种网页文本处理手段

精选留言 (6)

💬 写留言



风铃

2022-11-20 来自浙江

主要是现在大家都习惯了快速上手，突然间讲了这么多理论还没有开始编码，大家就不太理解了，老师最好的有一个大纲展示。



👍 1



shuff1e

2022-11-20 来自上海

能不能上github的项目代码链接？

共 2 条评论 >

👍 1



徐曙辉

2022-11-21 来自湖南

我觉得最关键的是分清楚每一层的功能和依赖关系，在拆分微服务的时候也是这样，循环依赖都是因为每层的作用没有梳理清楚，如果实在要依赖，就把依赖部分代码拆的更细，不调用业务方法，而是调用底层的存储或者第三方包，少量的重复代码没关系



Geek_crazydaddy

2022-11-21 来自江苏

好菜不怕晚，老师还是按自己的节奏来啊，道与术都很重要啊，甚至有些时候不知道“道”，“术”理解起来就很难，而且知识很难一次就理解透彻，前面说了原理后面代码遇到不懂的地方回头对着看肯定比自己百度谷歌有效率啊，至于循环依赖，个人感觉大部分是由于项目结构或者代码只能不够明确导致的，小面积循环可以复制一份，大面积的就要考虑下项目结构了



一步



2022-11-19 来自广东

Go 项目中如果存在循环依赖，编译器是不允许通过的



天下无鱼

<https://shikey.com/>



Jack

2022-11-19 来自广东

打卡

