

## 19. 一文搞懂 Go Modules 前世今生及入门使用

在以前，Go 语言的包依赖管理一直都被大家所诟病，Go 官方也在一直在努力为开发者提供更方便易用的包管理方案，从最初的 GOPATH 到 GO VENDOR，再到最新的 Go Modules，虽然走了不少的弯路，但最终还是拿出了 Go Modules 这样像样的解决方案。

目前最主流的包依赖管理方式是使用官方推荐的 Go Modules，这不前段时间 Go 1.14 版本发布，官方正式放话，强烈推荐你使用 Go Modules，并且有自信可以用于生产中。

本文会大篇幅的讲解 Go Modules 的使用，但是在那之前，我仍然会简要介绍一下前两个解决方案 GOPATH 和 go vendor 到底是怎么回事？我认为这是有必要的，因为只有了解它的发展历程，才能知道 Go Modules 的到来是有多么的不容易，多么的意义非凡。

### 1. 最古老的 GOPATH

GOPATH 应该很多人都很眼熟了，之前在配置环境的时候，都配置过吧？

你可以将其理解为用户目录，在这个用户目录下，通常有如下的目录结构

```
~ on [?]master! 22:07:47
$ go env GOPATH
/Users/MING/go

~ on [?]master! 22:07:50
$ ll /Users/MING/go
total 0
drwxr-xr-x  18 MING  staff   576B  3  8 22:00 bin
drwxr-xr-x   4 MING  staff  128B  3  8 21:50 pkg
drwxr-xr-x   3 MING  staff   96B  3  5 21:13 src
```

每个目录存放的文件，都不相同

- bin: 存放编译后生成的二进制可执行文件
- pkg: 存放编译后生成的 `.a` 文件
- src: 存放项目的源代码，可以是你自己写的代码，也可以是你 go get 下载的包

将你的包或者别人的包全部放在 `$GOPATH/src` 目录下进行管理的方式，我们称之为 GOPATH 模式。

在这个模式下，使用 go install 时，生成的可执行文件会放在 `$GOPATH/bin` 下

~/go on [?]master! 22:08:09

\$ tree

```
.
├── bin
├── pkg
├── src
│   ├── myapp
│   │   └── main.go
│   └── mypkg
│       └── hello.go
```

5 directories, 2 files

~/go on [?]master! 22:08:33

\$ cat src/myapp/main.go

```
package main
```

```
import "fmt"
```

```
func main(){
    fmt.Println("hello")
}
```

~/go on [?]master! 22:08:37

\$ go install myapp

~/go on [?]master! 22:08:45

\$ tree

```
.
├── bin
│   └── myapp
├── pkg
├── src
│   ├── myapp
│   │   └── main.go
│   └── mypkg
│       └── hello.go
```

5 directories, 3 files

如果你安装的是一个库，则会生成 `.a` 文件到 `$GOPATH/pkg` 下对应的平台目录中（由 `GOOS` 和 `GOARCH` 组合而成），生成 `.a` 为后缀的文件。

```
~/go on [?]master! 22:08:47
```

```
$ go install mypkg
```

```
~/go on [?]master! 22:10:38
```

```
$ tree
```

```
.
├── bin
│   └── myapp
├── pkg
│   └── darwin_amd64
│       └── mypkg.a
└── src
    ├── myapp
    │   └── main.go
    ├── mypkg
    │   └── hello.go
```

6 directories, 4 files

```
~/go on [?]master! 22:10:40
```

```
$ cat src/mypkg/hello.go
```

```
package mypkg
```

```
import "fmt"
```

```
func main(){
    fmt.Println("hello")
}
```

GOOS, 表示的是目标操作系统, 有 darwin (Mac) , linux, windows, android, netbsd, openbsd, solaris, plan9 等

而 GOARCH, 表示目标架构, 常见的有 arm, amd64 等

这两个都是 go env 里的变量, 你可以通过 `go env 变量名` 进行查看

```
~ on [?]master! 13:25:55
```

```
$ go env GOOS GOARCH
```

```
darwin
```

```
amd64
```

至此, 你可能不会觉得上面的方案会产生什么样的问题, 直到你开始真正使用 GOPATH 去开发程序, 就不得不开始面临各种各样的问题, 其中最严重的就是版本管理问题, 因为 GOPATH 根本没有版本的概念。

以下几点是你使用 GOPATH 一定会碰到的问题:

- 你无法在你的项目中, 使用指定版本的包, 因为不同版本的包的导入方法也都一样
- 其他人运行你的开发的程序时, 无法保证他下载的包版本是你所期望的版本, 当对方使用了其他版本, 有可能导致程序无法正常运行
- 在本地, 一个包只能保留一个版本, 意味着你在本地开发的所有项目, 都得用同一个版本的包, 这几乎是不可能的。

## 2. go vendor 模式的过渡

为了解决 GOPATH 方案下不同项目下无法使用多个版本库的问题，Go v1.5 开始支持 vendor 。

以前使用 GOPATH 的时候，所有的项目都共享一个 GOPATH，需要导入依赖的时候，都来这里找，正所谓一山不容二虎，在 GOPATH 模式下只能有一个版本的第三方库。

解决思路就是，在每个项目下都创建一个 vendor 目录，每个项目所需的依赖都只会下载到自己 vendor 目录下，项目之间的依赖包互不影响。在编译时，v1.5 的 Go 在你设置了开启 `GO15VENDOREXPERIMENT=1`（注：这个变量在 v1.6 版本默认为 1，但是在 v1.7 后，已去掉该环境变量，默认开启 `vendor` 特性，无需你手动设置）后，会提升 vendor 目录的依赖包搜索路径的优先级（相较于 GOPATH）。

其搜索包的优先级顺序，由高到低是这样的

- 当前包下的 vendor 目录
- 向上级目录查找，直到找到 src 下的 vendor 目录
- 在 GOROOT 目录下查找
- 在 GOPATH 下面查找依赖包

虽然这个方案解决了一些问题，但是解决得并不完美。

- 如果多个项目用到了同一个包的同一个版本，这个包会存在于该机器上的不同目录下，不仅对磁盘空间是一种浪费，而且没法对第三方包进行集中式的管理（分散在各个角落）。
- 并且如果要分享开源你的项目，你需要将你的所有的依赖包悉数上传，别人使用的时候，除了你的项目源码外，还有所有的依赖包全部下载下来，才能保证别人使用的时候，不会因为版本问题导致项目不能如你预期那样正常运行。

这些看似不是问题的问题，会给我们的开发使用过程变得非常难受，虽然我是初学者，还未使用过 go vendor，但能有很明显的预感，这个方案照样会另我崩溃。

## 3. go mod 的横空出世

go modules 在 v1.11 版本正式推出，在最新发布的 v1.14 版本中，官方正式发话，称其已经足够成熟，可以应用于生产上。

从 v1.11 开始，`go env` 多了个环境变量：`GO111MODULE`，这里的 111，其实就是 v1.11 的象征标志，go 里好像很喜欢这样的命名方式，比如当初 vendor 出现的时候，也多了个 `GO15VENDOREXPERIMENT` 环境变量，其中 15，表示的 vendor 是在 v1.5 时才诞生的。

`GO111MODULE` 是一个开关，通过它可以开启或关闭 go mod 模式。

它有三个可选值：`off`、`on`、`auto`，默认值是 `auto`。

1. `GO111MODULE=off` 禁用模块支持，编译时会从 `GOPATH` 和 `vendor` 文件夹中查找包。
2. `GO111MODULE=on` 启用模块支持，编译时会忽略 `GOPATH` 和 `vendor` 文件夹，只根据 `go.mod` 下载依赖。
3. `GO111MODULE=auto`，当项目在 `$GOPATH/src` 外且项目根目录有 `go.mod` 文件时，自动开启模块支持。

go mod 出现后，`GOPATH`（肯定没人使用了）和 `GOVENDOR` 将会且正在被逐步淘汰，但是若你的项目仍然要使用那些即将过时的包依赖管理方案，请注意将 `GO111MODULE` 置为 `off`。

具体怎么设置呢？可以使用 `go env` 的命令，如我要开启 go mod，就使用这条命令

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

Copy

## 4. go mod 依赖的管理

接下来，来演示一下 go modules 是如何来管理包依赖的。

go mod 不再依靠 `$GOPATH`，使得它可以脱离 `GOPATH` 来创建项目，于是我们在家目录下创建一个 `go_test` 的目录，用来创建我的项目，详细操作如下：

~/go\_test on [?]master! 23:59:41

\$ go env G0111MODULE GOPATH

on

/Users/MING/go

~/go\_test on [?]master! 23:59:47

\$ tree /Users/MING/go

/Users/MING/go

0 directories, 0 files

~/go\_test on [?]master! 23:59:59

\$ tree /Users/MING/go\_test

/Users/MING/go\_test

└─ github.com

└─ BingmingWong

└─ main.go

2 directories, 1 file

~/go\_test on [?]master! 0:00:02

\$ cat github.com/BingmingWong/main.go

package main

import (

log "github.com/sirupsen/logrus"

)

func main() {

log.WithFields(log.Fields{

"animal": "walrus",

}).Info("A walrus appears")

}

GOPATH下没有任何文件

在 GOPATH 外新建一个项目  
main.go 里引用了 logrus 包

接下来，进入项目目录，执行如下命令进行 go modules 的初始化



```
~/go_test on [?]master! 0:00:21
```

```
$ cd github.com/BingmingWong
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:05:54
```

```
$ go mod init github.com/BingmingWong/module-main
```

```
go: creating new go.mod: module github.com/BingmingWong/module-main
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:06:52
```

```
$ ls -l
```

```
total 16
```

```
-rw-r--r-- 1 MING staff 52 3 14 00:06 go.mod
```

```
-rw-r--r-- 1 MING staff 170 3 12 22:41 main.go
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:06:58
```

```
$ go install
```

```
go: finding module for package github.com/sirupsen/logrus
```

```
go: downloading github.com/sirupsen/logrus v1.4.2
```

```
go: found github.com/sirupsen/logrus in github.com/sirupsen/logrus v1.4.2
```

```
go: downloading golang.org/x/sys v0.0.0-20190422165155-953cdadca894
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:07:21
```

```
$ ls -l
```

```
total 24
```

```
-rw-r--r-- 1 MING staff 95 3 14 00:07 go.mod
```

```
-rw-r--r-- 1 MING staff 849 3 14 00:07 go.sum
```

```
-rw-r--r-- 1 MING staff 170 3 12 22:41 main.go
```

初始化, 多出了go.mod

执行安装, 会下载依赖包  
并且完善 go.mod 文件  
生成 go.sum 文件

接下来很重要的一点, 我们要看看 go install 把下载的包安装到哪里了?

```
~/go_test/github.com/BingmingWong on [?]master! 0:12:00
```

```
$ tree ~/go_test/  
/Users/MING/go_test/  
└─ github.com  
   └─ BingmingWong  
      ├── go.mod  
      ├── go.sum  
      └─ main.go
```

在当前项目下，并没有发现 logrus 依赖包

2 directories, 3 files

```
~/go_test/github.com/BingmingWong on [?]master! 0:12:02
```

```
$ tree ~/go/ -L 5  
/Users/MING/go/  
└─ bin  
   └─ module-main  
└─ pkg
```

但是在 GOPATH 下，发现 logrus 的踪迹，原来 go mod 会将所有的依赖包安装在 \$GOPATH/pkg 下呀

同时，可以发现 go install 为我们生成的可执行文件 module-main 也在 \$GOPATH/bin 下

```
└─ mod  
   ├── cache  
   │   ├── download  
   │   │   ├── github.com  
   │   │   ├── golang.org  
   │   │   └─ sumdb  
   ├── lock  
   ├── github.com  
   │   ├── sirupsen  
   │   │   └─ logrus@v1.4.2  
   ├── golang.org  
   │   └─ x  
   │       └─ sys@v0.0.0-20190422165155-953cdadca894  
   └─ sumdb  
       ├── sum.golang.org  
       └─ latest
```

16 directories, 3 files

上面我们观察到，在使用 go modules 模式后，项目目录下会多生成两个文件也就是 `go.mod` 和 `go.sum`。

这两个文件是 go modules 的核心所在，这里不得不好好介绍一下。

```
~/go_test/github.com/BingmingWong on [?]master! 0:12:04
```

```
$ cat go.mod  
module github.com/BingmingWong/module-main  
  
go 1.14  
  
require github.com/sirupsen/logrus v1.4.2
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:16:38
```

```
$ cat go.sum  
github.com/davecgh/go-spew v1.1.1/go.mod h1:J7Y8YcW2NihsgmVo/mv3lAwl/sk0N4iLHjSsi+c5H38=  
github.com/konsorten/go-windows-terminal-sequences v1.0.1/go.mod h1:T0+1ngSBFLxvqU3pZ+m/2kptfBszLMUkC4ZK/EgS/cQ=  
github.com/pmezard/go-difflib v1.0.0/go.mod h1:iKH77koFhYxTK1pcRnkKkqfTogsbg7gZNVY4sRDYZ/4=  
github.com/sirupsen/logrus v1.4.2 h1:SPIRiBHv4MatM3XXN02BJeFLZwZ2LvZgfQ5+UNI2im4=  
github.com/sirupsen/logrus v1.4.2/go.mod h1:tLMuLIddtU9McNUSpp0xgXVQah82FyeX6MwdIuYE2rE=  
github.com/stretchr/testify v1.2.2/go.mod h1:801J144z43v533Zj35dWtU39p381U1wWpK9CtYVY=  
golang.org/x/sys v0.0.0-20190422165155-953cdadca894 h1:Cz4ceDQGxuKRnVBDS23GTn/pU50E2C0WrNT0YK1Uuc=  
golang.org/x/sys v0.0.0-20190422165155-953cdadca894/go.mod h1:h1NjWce9XRLGQEsW7wpKNCjG9DtN1C1VuFLEZdDNBES=
```

## go.mod 文件

go.mod 的内容比较容易理解

- 第一行：模块的引用路径
- 第二行：项目使用的 go 版本
- 第三行：项目所需的直接依赖包及其版本

在实际应用上，你会看见更复杂的 go.mod 文件，比如下面这样

```
module github.com/BingmingWong/module-test

go 1.14

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 (
    golang.org/x/crypto v0.0.0-20180820150726-614d502a4dac => github.com/golang/crypto v0.0.0-20180820150726-614d502a4dac
    golang.org/x/net v0.0.0-20180821023952-922f4815f713 => github.com/golang/net v0.0.0-20180826012351-8a410e7b
    golang.org/x/text v0.3.0 => github.com/golang/text v0.3.0
)
```

主要是多出了两个 flag：

- `exclude`：忽略指定版本的依赖包
- `replace`：由于在国内访问golang.org/x的各个包都需要f.q，你可以在go.mod中使用replace替换成github上对应的库。

## go.sum 文件

反观 go.sum 文件，就比较复杂了，密密麻麻的。

可以看到，内容虽然多，但是也不难理解

每一行都是由 `模块路径`，`模块版本`，`哈希检验值` 组成，其中哈希检验值是用来保证当前缓存的模块不会被篡改。hash 是以 `h1:` 开头的字符串，表示生成checksum的算法是第一版的hash算法（sha256）。

值得注意的是，为什么有的包只有一行

```
<module> <version>/go.mod <hash>
```

而有的包却有两行呢

```
<module> <version> <hash>
<module> <version>/go.mod <hash>
```

那些有两行的包，区别就在于 hash 值不一行，一个是 `h1:hash`，一个是 `go.mod h1:hash`

而 `h1:hash` 和 `go.mod h1:hash` 两者，要不就是同时存在，要不就是只存在 `go.mod h1: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 h1:hash` 的情况。  
[引用自 3]

go.mod 和 go.sum 是 go modules 版本管理的指导性文件，因此 go.mod 和 go.sum 文件都应该提交到你的 Git 仓库中去，避免其他人使用你写项目时，重新生成的go.mod 和 go.sum 与你开发的基准版本的不一致。



## 5. go mod 命令的使用

- `go mod init` : 初始化go mod, 生成go.mod文件, 后可接参数指定 module 名, 上面已经演示过。
- `go mod download` : 手动触发下载依赖包到本地cache (默认为 `$GOPATH/pkg/mod` 目录)
- `go mod graph` : 打印项目的模块依赖结构

```
~/go_test/github.com/BingmingWong on [?]master! 0:28:15
$ go mod graph
github.com/BingmingWong/module-main github.com/sirupsen/logrus@v1.4.2
github.com/sirupsen/logrus@v1.4.2 github.com/davecgh/go-spew@v1.1.1
github.com/sirupsen/logrus@v1.4.2 github.com/konsorten/go-windows-terminal-sequences@v1.0.1
github.com/sirupsen/logrus@v1.4.2 github.com/pmezard/go-difflib@v1.0.0
github.com/sirupsen/logrus@v1.4.2 github.com/stretchr/testify@v1.2.2
github.com/sirupsen/logrus@v1.4.2 golang.org/x/sys@v0.0.0-20190422165155-953cdadca894
```

- `go mod tidy` : 添加缺少的包, 且删除无用的包
- `go mod verify` : 校验模块是否被篡改过
- `go mod why` : 查看为什么需要依赖
- `go mod vendor` : 导出项目所有依赖到vendor下

```
~/go_test/github.com/BingmingWong on [?]master! 0:38:40
$ tree
.
├── go.mod
├── go.sum
└── main.go
```

0 directories, 3 files

```
~/go_test/github.com/BingmingWong on [?]master! 0:38:42
$ go mod vendor
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:38:47
$ tree -L 2
.
├── go.mod
├── go.sum
├── main.go
└── vendor
    ├── github.com
    ├── golang.org
    └── modules.txt
```

3 directories, 4 files

- `go mod edit` : 编辑go.mod文件, 接 `-fmt` 参数格式化 go.mod 文件, 接 `-require=golang.org/x/text` 添加依赖, 接 `-droprequire=golang.org/x/text` 删除依赖, 详情可参考 `go help mod edit`

~/go\_test/github.com/BingmingWong on [?]master! 0:45:53

\$ cat go.mod

module github.com/BingmingWong/module-main

go 1.14

require github.com/sirupsen/logrus v1.4.2

~/go\_test/github.com/BingmingWong on [?]master! 0:45:58

\$ go mod edit -replace=github.com/sirupsen/logrus@1.4.2=github.com/sirupsen/logrus@1.4.1

~/go\_test/github.com/BingmingWong on [?]master! 0:46:00

\$ cat go.mod

module github.com/BingmingWong/module-main

go 1.14

require github.com/sirupsen/logrus v1.4.2

replace github.com/sirupsen/logrus 1.4.2 => github.com/sirupsen/logrus 1.4.1

- `go list -m -json all` : 以 json 的方式打印依赖详情

~/go\_test/github.com/BingmingWong on [?]master! 0:58:35

\$ go list -m -json all

```
{
  "Path": "github.com/BingmingWong/module-main",
  "Main": true,
  "Dir": "/Users/MING/go_test/github.com/BingmingWong",
  "GoMod": "/Users/MING/go_test/github.com/BingmingWong/go.mod",
  "GoVersion": "1.14"
}
{
  "Path": "github.com/davecgh/go-spew",
  "Version": "v1.1.1",
  "Time": "2018-02-21T23:26:28Z",
  "Indirect": true,
  "Dir": "/Users/MING/go/pkg/mod/github.com/davecgh/go-spew@v1.1.1",
  "GoMod": "/Users/MING/go/pkg/mod/cache/download/github.com/davecgh/go-spew/@v/v1.1.1.mod"
}
```

如何给项目添加依赖（写进 go.mod）呢？

有两种方法：

- 你只要在项目中有 import，然后 go build 就会 go module 就会自动下载并添加。
- 自己手工使用 go get 下载安装后，会自动写入 go.mod。



```
~/go_test/github.com/BingmingWong on [?]master! 0:50:01
```

```
$ cat go.mod
```

```
module github.com/BingmingWong/module-main
```

```
go 1.14
```

```
require github.com/sirupsen/logrus v1.4.2
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:50:06
```

```
$ go get gopkg.in/gin-gonic/gin.v1
```

```
go: downloading gopkg.in/gin-gonic/gin.v1 v1.3.0
```

```
go: gopkg.in/gin-gonic/gin.v1 upgrade => v1.3.0
```

```
go: finding module for package github.com/gin-gonic/gin/json
```

```
go: finding module for package github.com/gin-contrib/sse
```

```
go: finding module for package github.com/mattn/go-isatty
```

```
go: finding module for package github.com/gin-gonic/gin/binding
```

```
go: finding module for package github.com/gin-gonic/gin/render
```

```
go: downloading github.com/gin-gonic/gin v1.5.0
```

```
go: downloading github.com/gin-contrib/sse v0.1.0
```

```
go: downloading github.com/mattn/go-isatty v0.0.12
```

```
go: found github.com/gin-contrib/sse in github.com/gin-contrib/sse v0.1.0
```

```
go: found github.com/gin-gonic/gin/binding in github.com/gin-gonic/gin v1.5.0
```

```
go: found github.com/mattn/go-isatty in github.com/mattn/go-isatty v0.0.12
```

```
go: finding module for package github.com/gin-gonic/gin/json
```

```
go: downloading golang.org/x/sys v0.0.0-20200116001909-b77594299b42
```

```
go: downloading gopkg.in/go-playground/validator.v9 v9.29.1
```

```
go: downloading github.com/ugorji/go v1.1.7
```

```
go: downloading gopkg.in/yaml.v2 v2.2.2
```

```
go: downloading github.com/json-iterator/go v1.1.7
```

```
go: downloading github.com/golang/protobuf v1.3.2
```

```
go: downloading github.com/ugorji/go/codec v1.1.7
```

```
go: downloading github.com/go-playground/universal-translator v0.16.0
```

```
go: downloading github.com/leodido/go-urn v1.1.0
```

```
go: downloading github.com/go-playground/locales v0.12.1
```

```
go: downloading github.com/modern-go/reflect2 v0.0.0-20180701023420-4b7aa43c6742
```

```
go: downloading github.com/modern-go/concurrent v0.0.0-20180228061459-e0a39a4cb421
```

```
go: finding module for package github.com/gin-gonic/gin/json
```

```
go: finding module for package github.com/gin-gonic/gin/json
```

```
../../../../go/pkg/mod/gopkg.in/gin-gonic/gin.v1@v1.3.0/errors.go:12:2: module github.com/gin-gonic/gin@1.5.0: missing go.sum entry for module providing package github.com/gin-gonic/gin/json
```

```
~/go_test/github.com/BingmingWong on [?]master! 0:50:20
```

```
$ cat go.mod
```

```
module github.com/BingmingWong/module-main
```

```
go 1.14
```

```
require (
    github.com/gin-gonic/gin v1.5.0 // indirect
    github.com/mattn/go-isatty v0.0.12 // indirect
    github.com/sirupsen/logrus v1.4.2
    gopkg.in/gin-gonic/gin.v1 v1.3.0 // indirect
)
```

## 7. 总结写在最后

如果让我用一段话来评价 GOPATH 和 go vendor, 我会说

GOPATH 做为 Golang 的第一个包管理模式, 只能保证你能用, 但不保证好用, 而 go vendor 解决了 GOPATH 忽视包版本的管理, 保证好用, 但是还不够好用, 直到 go mod 的推出后, 才使 Golang 包的依赖管理有了一个能让 Gopher 都统一比较满意的方案, 达到了能用且好用的标准。

如果是刚开始学习 Golang，那么 GOPATH 和 go vendor 可以做适当了解，不必深入研究，除非你要接手的项目由于一些历史原因仍然在使用 go vender 管理，除此之外，任何 Gopher 应该从此刻就投入 go modules 的怀抱。

以上是我在这几天的学习总结，希望对还未入门阶段的你，有所帮助。另外，本篇文章如有写得不对的，请后台批评指正，以免误导其他朋友，非常感谢。

## 8. 推荐参考文章

- [Go语言之依赖管理](#)
- [Go 包依赖管理工具 —— govendor](#)
- [Go Modules 终极入门](#)
- [何处安放我们的 Go 代码](#)

### 系列导读

#### 01. 开发环境的搭建 (Goland & VS Code)

#### 02. 学习五种变量创建的方法

#### 03. 详解数据类型：\*\*\*\*整形与浮点型

#### 04. 详解数据类型：byte、rune与string

#### 05. 详解数据类型：数组与切片

#### 06. 详解数据类型：字典与布尔类型

#### 07. 详解数据类型：指针

#### 08. 面向对象编程：结构体与继承

#### 09. 一篇文章理解 Go 里的函数

#### 10. Go语言流程控制：if-else 条件语句

#### 11. Go语言流程控制：switch-case 选择语句

#### 12. Go语言流程控制：for 循环语句

#### 13. Go语言流程控制：goto 无条件跳转

#### 14. Go语言流程控制：defer 延迟调用

#### 15. 面向对象编程：接口与多态

#### 16. 关键字：make 和 new 的区别？

#### 17. 一篇文章理解 Go 里的语句块与作用域

#### 18. 学习 Go 协程：goroutine

#### 19. 学习 Go 协程：详解信道/通道

#### 20. 几个信道死锁经典错误案例详解

#### 21. 学习 Go 协程：WaitGroup

#### 22. 学习 Go 协程：互斥锁和读写锁

#### 23. Go 里的异常处理：panic 和 recover

#### 24. 超详细解读 Go Modules 前世今生及入门使用

#### 25. Go 语言中关于包导入必学的 8 个知识点

#### 26. 如何开源自己写的模块给别人用？



## 27. 说说 Go 语言中的类型断言?

## 28. 这五点带你理解Go语言的select用法



作者：MING - Python编程时光

出处：<https://www.cnblogs.com/wongbingming/p/12941021.html>

本站使用「署名 4.0 国际」创作共享协议，转载请在文章明显位置注明作者及出处。

posted @ 2020-05-23 08:56 王小白 阅读(3420) 评论(1) 编辑 收藏

### 评论列表

Copyright © 2021 王小白  
Powered by .NET 5.0 on Kubernetes  
Powered By Cnblogs | Theme Silence v1.1.2  
Powered By Cnblogs | Theme Silence v1.1.2

