Go Modules介绍

1 背景

- Go长期一以来，没有一个官方包管理工具，社区推出了各种包管理工具(以dep最为知名)，整体较为混乱。
- Go的代码组织结构/管理跟GOPATH强关联，饱受社区争议。
- 为了解决如上两个问题，Go的掌托人Russ Cox推出使用Go Modules来解决。
- 在Go1.11和Go1.12初步完成了对Go Modules的支持，在Go1.13时官方推荐把Go Modules作为包管理系统。
- Go Modules作为未来Go官方的包管理系统，有必要一探究竟。

2 介绍

随着Go Modules的推出，引入了一些新的属性和概念。

2.1 环境变量

2.1.1 开关变量

GO111MODULE： Go Modules功能的开关，可选值：

- auto: 只要项目内包含了go.mod文件，就会启用Go Modules功能（Go1.13中的默认值）
- on： 总是启用Go Modules功能
- off：总是关闭Go Modules功能

注：由于Go Modules功能几乎内置到go的所有命令(如：go build、go install、go test），所以Go Modules功能的启用与否，将影响go各个子命令的行为。

2.1.2 辅助变量

- GOPROXY
  - 通过该变量指定的代理地址去下载模块（下载的是指定版本的ZIP包，减少网络传输，加快模块构建）
  - 默认值为：https://proxy.golang.org,direct
  - ”direct“为特殊标识符，表示直接从模块的原代码地址下载
- GOSUMDB
  - 用来验证指定版本包的checksum，确保包的内容没有变化
  - 默认值为：https://sum.golang.org
  - 当设置为off时，表示不检查
- GONOPROXY、GONOSUMDB、GOPRIVATE

- 私有仓库的包无法通过GOPROXY来下载，这个时候可通过设置如上3个变量来解决
- GOPRIVATE的值将作为GONOPROXY和GONOSUMDB的默认值，所以一般只要设置GOPRIVATE即可
- 假定GOPRIVATE设置为"*.example.com"，所有以example.com的子域名(如：abc.example.com)为前缀的模块下载都不通过GOPROXY指定的代理，但不包括example.com本身

2.1.3 变量管理

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

$ go env -w GOPROXY = "https://mirrors.aliyun.com/goproxy/,direct"
$ go env -w GOSUMDB = "https://sum.golang.google.cn/"

$ go env -w GOSUMDB = "off"
```

注：如上方式不会覆盖OS的环境变量，即：OS设置的环境变量GOPROXY与`go env`中GOPROXY变量的值不一样，go会采用OS中环境变量GOPROXY的值，因此，最好不要通过OS环境变量方式设置go变量。

2.2 记录文件

Go Modules的包版本管理功能的相关信息会记录到go.mod和go.sum两个文件。

go.mod

```
module example.com/hello

go 1.13

require (
    example.com/aaa v1.1.1
    example.com/bbb v1.1.1                                  <===
    example.com/bbb/v2 v2.2.2                               <=== tagv2v2
    example.com/ccc v0.0.0-20200124185754-1b0db40df49a      <=== tagGo Modulespseudo-version(https://golang.org
/cmd/go/#hdr-Pseudo_versions)
    example.com/ddd v1.0.0 //indirect                       <===
)

exclude example.com/aaa v1.1.2
replace example.com/bbb v1.1.1 => example.com/3b v1.1.1
replace example.com/ccc => ../my-project-dir/ccc            <===
```

启用Go Modules功能会生成一个go.mod文件，它描述了项目的元数据，每行都以一个动词开头：

- module： 用于定义当前项目的模块路径
- go： 用于设置预期的 Go 版本
- 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

注：go.mod文件默认只记录其直接依赖的包，只有当主动升级其非直接依赖的包才会记录到go.mod文件，其以”// indirect"作为后缀进行标识。

go.sum

```
k8s.io/klog v1.0.0 h1:Pt+yjF5aB1xDSVbau4VsWe+dQNzA0qv1LlXdC2dF6Q8=
k8s.io/klog v1.0.0/go.mod h1:4Bi6QPql/J/LkTDqv7R/cd3hPo4k2DG6Ptcz060Ez5I=
k8s.io/kube-openapi v0.0.0-20190816220812-743ec37842bf h1:EYm5AW/UUDbnmnI+gK0TJDVK9qPLhM+sRHYanNKw0EQ=
k8s.io/kube-openapi v0.0.0-20190816220812-743ec37842bf/go.mod h1:1TqjTSzOxsLGIKfj0lK8EeCP7K1iUG65v09OM0/WG5E=
```

说明：

- 所有直接和间接依赖模块都会记录到go.sum文件，每个模块会有两行记录
- 第一行记录为模块打包为zip文件后的hash值
- 第二行记录为包中go.mod文件的hash值

2.3 命令

Go Modules功能提供`go mod`管理命令，有如下子命令：

子命令	说明
download	download modules to local cache（下载依赖模块到本地缓存）
edit	edit go.mod from tools or scripts（编辑go.mod文件）
graph	print module requirement graph（打印模块依赖图）
init	initialize new module in current directory（初始化模块）
tidy	add missing and remove unused modules（拉取缺少的模块以及删除不需要的模块）
vendor	make vendored copy of dependencies （将依赖包复制到vendor目录）
verify	verify dependencies have expected content（验证模块依赖是否正确）
why	explain why packages or modules are needed（解释为什么某个模块/包被需要）

3 基本使用

打开Go Modules功能, 运行命令： `go env -w GO111MODULE = "on"`

3.1 创建module

- 创建一个hello空目录(任意目录下，跟是否在GOPATH目录下无关)

```
$ mkdir hello
$ cd hello
```

- 初始化模块名

```
$ go mod init example.com/hello
go: creating new go.mod: module example.com/hello
$ cat go.mod
module example.com/hello

go 1.13
$
```

- 添加hello.go和hello_test.go文件

```
$ ls
go.mod hello.go  hello_test.go
$ cat hello.go
package hello

func Hello() string {
        return "Hello World"
}
$ cat hello_test.go
package hello

import "testing"

func TestHello(t *testing.T) {
        want := "Hello World"
        if got := Hello(); want != got {
                t.Errorf("want: %s, got: %s", want, got)
        }
}
$
```

- 运行测试

```
$ go test
PASS
ok          example.com/hello        0.005s
$
```

3.2 增加module

- 修改hello.go，导入go-module-example模块：

```
$ cat hello.go
package hello

import "github.com/kulong0105/go-modules-example"

func Hello() string {
        return "Hello World"
}

func Info(s string) string {
        return show.Name(s)
}
$
```

- 修改hello_test.go，添加Info函数测试：

```
$ cat hello_test.go
package hello

import "testing"

func TestHello(t *testing.T) {
        want := "Hello World"
        if got := Hello(); want != got {
                t.Errorf("want: %s, got: %s", want, got)
        }
}

func TestInfo(t *testing.T) {
        name := "kulong0105"
        want := "My Name: kulong0105."
        if got := Info(name); want != got {
                t.Errorf("want: %s, got: %s", want, got)
        }
}
$
```

- 运行测试

```
$ go test -v
go: finding github.com/kulong0105/go-modules-example v1.2.0
go: downloading github.com/kulong0105/go-modules-example v1.2.0
go: extracting github.com/kulong0105/go-modules-example v1.2.0
=== RUN   TestHello
--- PASS: TestHello (0.00s)
=== RUN   TestInfo
--- PASS: TestInfo (0.00s)
PASS
ok      example.com/hello       0.005s
$
```

- 查看go.mod

```
$ cat go.mod
module example.com/hello

go 1.13

require github.com/kulong0105/go-modules-example v1.2.0
$
```

说明:

- 同一个模块版本的数据只缓存一份，所有其他模块共享使用
- 所有模块版本数据均缓存在 `$GOPATH/pkg/mod`和 `$GOPATH/pkg/sum` 目录
- 可使用命令`go clean -modcache`清空所有缓存的module cache

3.3 更新module

3.3.1 查看可用版本

更新之前，先查看模块的可用版本:

```
$ go list -m
example.com/hello
$ go list -m all
example.com/hello
github.com/kulong0105/go-modules-example v1.2.0
$ go list -m -versions github.com/kulong0105/go-modules-example
github.com/kulong0105/go-modules-example v1.0.0 v1.1.0 v1.2.0
$ go list -m -versions github.com/kulong0105/go-modules-example/v2
go: finding github.com/kulong0105/go-modules-example/v2 v2.0.0
github.com/kulong0105/go-modules-example/v2 v2.0.0
$ go list -m -versions github.com/kulong0105/go-modules-example/v3
go: finding github.com/kulong0105/go-modules-example/v3 v3.0.0
github.com/kulong0105/go-modules-example/v3 v3.0.0
$
```

说明：

- 模块的版本命名需要遵守 semantic version 规则，如：模块版本为：vX.Y.Z，X表示主版本，Y表示次版本，Z表示修复版本。
- 主版本大于1的tag(如: v2.0.0和v3.0.0)都需要通过添加后缀才能查询到。

3.3.2 更新方式

通过go get可更新依赖，其支持多种方式更新依赖模块：

- go get xxx@latest （默认方式）
- go get xxx@v1.2.3
- go get xxx/v2@v2.3.4
- go get xxx/v3@master
- go get xxx@e6c2b0d
- go get -u ./...

3.3.3 非主版本更新

1）高 → 低

当前使用“go-modules-example”模块的版本是v1.2.0，更新到使用v1.0.0版本：

```
$ go get github.com/kulong0105/go-modules-example@v1.0.0
go: finding github.com/kulong0105/go-modules-example v1.0.0
go: downloading github.com/kulong0105/go-modules-example v1.0.0
go: extracting github.com/kulong0105/go-modules-example v1.0.0
$ cat go.mod
module example.com/hello

go 1.13

require github.com/kulong0105/go-modules-example v1.0.0
$
```

说明：可以看到go.mod文件针对go-modules-example模块的依赖版本发生了变化， 从v1.2.0变为v1.0.0

2）低 → 高

当前使用“go-modules-example”模块的版本是v1.0.0，更新到使用v1.1.0版本：

```
$ go get -u github.com/kulong0105/go-modules-example
$ cat go.mod
module example.com/hello

go 1.13

require github.com/kulong0105/go-modules-example v1.2.0
$ go get -u github.com/kulong0105/go-modules-example@v1.1.0
go: finding github.com/kulong0105/go-modules-example v1.1.0
go: downloading github.com/kulong0105/go-modules-example v1.1.0
go: extracting github.com/kulong0105/go-modules-example v1.1.0
$ cat go.mod
module example.com/hello

go 1.13

require github.com/kulong0105/go-modules-example v1.1.0
$
```

说明：`go get -u`不会更新”主版本“(即不会更新到v2.0.0)，只会更新到当前主版本的最新稳定版本(即v1.2.0)。要想更新到当前主版本的非最新稳定版本，必须指定版本号。

3.3.4 主版本更新

当前使用“go-modules-example”模块的版本是v1.1.0，更新到使用v2.0.0版本：

1) 更新hello.go的import路径：

```
diff --git a/hello.go b/hello.go
index 045e67f..f00ccd1 100644
--- a/hello.go
+++ b/hello.go
@@ -1,6 +1,6 @@
 package hello

-import "github.com/kulong0105/go-modules-example"
+import "github.com/kulong0105/go-modules-example/v2"
```

注：由于Go Modules采用 semantic import versioning 机制，一定要添加v2作为导入后缀。

2) 更新到v2.0.0

```
$ GOPROXY=”direct“ go get  github.com/kulong0105/go-modules-example/v2@v2.0.0
go: finding github.com/kulong0105/go-modules-example v2.0.0
go: finding github.com/kulong0105/go-modules-example/v2 v2.0.0
go: downloading github.com/kulong0105/go-modules-example/v2 v2.0.0
go: extracting github.com/kulong0105/go-modules-example/v2 v2.0.0
$ cat go.mod
module example.com/hello

go 1.13

require (
        github.com/kulong0105/go-modules-example v1.2.0
        github.com/kulong0105/go-modules-example/v2 v2.0.0 // indirect
)
$ go mod tidy
$ cat go.mod
module example.com/hello

go 1.13

require github.com/kulong0105/go-modules-example/v2 v2.0.0
$
```

注：获取非v0和v1主版本的模块，必须在URL中后缀版本，如：” github.com/kulong0105/go-modules-example/v2 “

3）运行测试

```
$ go test
# example.com/hello [example.com/hello.test]
./hello.go:10:9: undefined: show.Name
FAIL        example.com/hello [build failed]
$ go doc github.com/kulong0105/go-modules-example/v2


package show // import "github.com/kulong0105/go-modules-example/v2"

func NameV2(name string) string
$ vim hello.go
$ git diff hello.go
diff --git a/hello.go b/hello.go
index 045e67f..44f4445 100644
--- a/hello.go
+++ b/hello.go
@@ -7,5 +7,5 @@ func Hello() string {
 }

 func Info(s string) string {
-       return show.Name(s)
+       return show.NameV2(s)
 }
$ go test -v
=== RUN   TestHello
--- PASS: TestHello (0.00s)
=== RUN   TestInfo
--- PASS: TestInfo (0.00s)
PASS
ok          example.com/hello       0.004s
$
```

说明：通过测试可以发现，v2版本的接口名字从Name改为NameV2，修改接口调用，重新测试通过。

4 语义化版本

Go Modules针对module的版本管理有”强制“要求，引入了语义化版本(semantic version)和语义化导入版本(sematic import versioning)

4.1 semantic version

Go Modules要求go.mod文件中的每个module都需要满足semantic version，其格式为： `vMAJOR.MINOR.PATCH(v2.2.1)`:

- MAJOR：主版本号，在对外暴露的API接口发生向后不兼容的变更，这种场景才需要更新主版本号
- MINOR： 次版本号，在对外暴露的API接口发生向后兼容的更改或增加新功能，这种场景只需要更新次版本号
- PATCH：修复版本号，在对外暴露的API接口没有变化的变更(如：Bug修复)，这种场景只需要更新修复版本号

注：

- 主版本号为0的版本，如：v0.1.0 → v0.2.0，次版本变更了，但由于v0版本本身表示的就是一个非稳定版本，因此可以不满足向后兼容性。
- 主版本号大于等于1的版本，如：1.2.0 → 1.3.0，次版本变更了，根据semantic version要求，需要满足向后兼容性。

4.2 semantic import versioning

Go Modules通过"假定"module的版本都满足semantic version机制，即满足模块兼容性策略，于是针对go.mod文件中的依赖模块提出了” sematic import versioning “机制：

- 同一主版本号模块只能存在一个，即不可配置同时依赖hello模块的v1.2.0和v1.3.0两个版本，所以，go.mod文件中不可能存在如下配置:

```
require example.com/hello v1.2.0
require example.com/hello v1.3.0
```

- 不同主版本号模块可以并存，即可配置同时依赖hello模块的v1.2.0和2.0.0两个版本，所以，go.mod文件如下配置是正常的：

```
require example.com/hello v1.2.0
require example.com/hello/v2 v2.0.0
```

- 主版本号大于1的模块，其模块名必须以主版本号作为后缀，如： "example.com/hello/v2”表示hello模块的版本号为v2.X.Y

4.3 发布版本

由于主版本号大于1的模块，其模块名必须以主版本号作为后缀，可以两种不同方式创建主版本号模块：

- 不新建主版本目录：仓库简洁，同名文件不需要存在多份
- 新建主版本目录: 官方推荐，提供更好兼容性，使用GOPATH模式的用户可以以文件路径方式导入模块，即 `import "github.com/kulong0105/go-modules-example/v3"`

需要注意的是：

在使用Go Modules功能时，导入包的路径并不表示远程仓库的文件路径，如导入go-modules-example模块的v2版本：`import "github.com/kulong0105/go-modules-example/v2"`，不是表示远程仓库一定要有v2目录，而是表示远程仓库的go.mod文件(可能在根目录，也可能在v2目录下)记录该模块的名字为”github.com/kulong0105/go-modules-example/v2"。

4.3.1 不新建主版本目录

```
$ cd hello
$ ls
go.mod  go.sum  hello.go  hello_test.go
$
...
Do something to change API
...
$ go mod edit -module example.com/hello/v2 go.mod
$ git diff go.mod
diff --git a/go.mod b/go.mod
index 688f32a..d2e7e78 100644
--- a/go.mod
+++ b/go.mod
@@ -1,4 +1,4 @@
-module example.com/hello
+module example.com/hello/v2

 go 1.13
$ git add --all
$ git commit -m "release v2.0.0"
$ git tag v2.0.0 HEAD
$
```

4.3.2 新建主版本目录

```
$ cd hello
$ ls
go.mod  go.sum  hello.go  hello_test.go
$ mkdir v3 $$ cd v3
$ cp -a ../{hello.go,hello_test.go}  ./
...
Do something to change API
...
$ go mod init example.com/hello/v3
go: creating new go.mod: module example.com/hello/v3
$ cat go.mod
module example.com/hello/v3

go 1.13
$ git add --all
$ git commit -m "release v3.0.0"
$ git tag 3.0.0 HEAD
$
```

5 Tips

在使用Go Modules管理包时，有如下tips：

- go get 默认总是获取最新稳定版本，如：一个模块有两个版本 v1.0.0 和 v1.1.0-rc，go get默认会获取v1.0.0版本。
- 不要篡改/覆盖/删除仓库中的tag，如：发现某个tag版本存在bug， 对bug进行修复后不要覆盖此tag，而是打个新tag。
- 由于模块镜像和检查仓库会存储模块内容和hash值，像覆盖tag的操作，不会使得新tag内容同步到镜像仓库，这样会导致无法从镜像仓库获取覆盖tag后的模块。
- 代理需要时间，刚提交到仓库的新tag，可能无法通过代理获取到，需要等一段时间，或者临时采用direct方式来获取，如：`GOPROXY=“direct” go get github.com/kulong0105/go-modules-example/v3"。
- 当无法在https://pkg.go.dev/ 网站搜索到你在github上的模块时，可运行一次'go get github.com/name/module'获取模块，之后即可在https://pkg.go.dev/ 查找到。

6 参考

https://blog.golang.org/using-go-modules