Go语言包（package）

Go语言是使用包来组织源代码的，包（package）是多个 Go 源码的集合，是一种高级的代码复用方案。Go语言中为我们提供了很多内置包，如 fmt、os、io 等。

任何源代码文件必须属于某个包，同时源码文件的第一行有效代码必须是`package pacakgeName`语句，通过该语句声明自己所在的包。

包的基本概念

Go语言的包借助了目录树的组织形式，一般包的名称就是其源文件所在目录的名称，虽然Go语言没有强制要求包名必须和其所在的目录名同名，但还是建议包名和所在目录同名，这样结构更清晰。

包可以区分命令空间（一个文件夹中不能有两个同名文件），也可以更好的管理项目。Go中创建一个包，一般是创建一个文件夹，在该文件夹里面的Go文件中，使用package关键字声明包名称，通常，文件夹名称和包名称相同。并且，同一个文件下面只有一个包。

包的习惯用法：

- 包名一般是小写的，使用一个简短且有意义的名称。
- 包名一般要和所在的目录同名，也可以不同，包名中不能包含`-`等特殊符号。
- 包一般使用域名作为目录名称，这样能保证包名的唯一性，比如 GitHub 项目的包一般会放到`GOPATH/src/github.com/userName/projectName`目录下。
- 包名为 main 的包为应用程序的入口包，编译不包含 main 包的源码文件时不会得到可执行文件。
- 一个文件夹下的所有源码文件只能属于同一个包，同样属于同一个包的源码文件不能放在多个文件夹下。

创建包

1. 创建一个名为baozi的文件夹
2. 创建一个baozi.go文件
3. 在该文件中声明包

示例：

```
package baozi

import "fmt"

func PkgTest() {
    fmt.Println("test package")
}
```

- 一个文件夹下只能有一个package
- 一个package的文件不能在多个文件夹下
- 如果多个文件夹下有重名的package，它们其实是彼此无关的package。
- 如果一个go文件需要同时使用不同目录下的同名package，需要在import这些目录时为每个目录指定一个package的别名。

导入包

要在代码中引用其他包的内容，需要使用 import 关键字导入使用的包。具体语法如下：

```
import "包的路径"
```

注意事项：

- import 导入语句通常放在源码文件开头包声明语句的下面；
- 导入的包名需要使用双引号包裹起来；
- 包名是从`GOPATH/src/`后开始计算的，使用`/`进行路径分隔。

包的导入有两种写法，分别是单行导入和多行导入。

单行导入

单行导入的格式如下：

```
import "包 1 的路径"
import "包 2 的路径"
```

多行导入

多行导入的格式如下：

```
import (
  "包 1 的路径"
  "包 2 的路径"
)
```

包的导入路径

包的引用路径有两种写法，分别是全路径导入和相对路径导入。

全路径导入

包的绝对路径就是`GOROOT/src/`或`GOPATH/src/`后面包的存放路径，如下所示：

```
import "lab/test"
import "database/sql/driver"
import "database/sql"
```

上面代码的含义如下：

- test 包是自定义的包，其源码位于`GOPATH/src/lab/test`目录下；
- driver 包的源码位于`GOROOT/src/database/sql/driver`目录下；
- sql 包的源码位于`GOROOT/src/database/sql`目录下。

相对路径导入

相对路径只能用于导入`GOPATH`下的包，标准包的导入只能使用全路径导入。

例如包 a 的所在路径是`GOPATH/src/lab/a`，包 b 的所在路径为`GOPATH/src/lab/b`，如果在包 b 中导入包 a ，则可以使用相对路径导入方式。示例如下：

```
// 相对路径导入
import "../a"
```

当然了，也可以使用上面的全路径导入，如下所示：

```
// 全路径导入
import "lab/a"
```

包的引用格式

包的引用有四种格式，标准引用格式、自定义别名引用格式、省略引用格式、匿名引用格式。

标准引用格式

```
import "fmt"
```

此时可以用`fmt.`作为前缀来使用 fmt 包中的方法，这是常用的一种方式。

```
package main
import "fmt"
func main() {
    fmt.Println("包子真帅")
}
```

自定义别名引用格式

在导入包的时候，我们还可以为导入的包设置别名。

```
import F "fmt"
```

其中 F 就是 fmt 包的别名，使用时我们可以使用`F.`来代替标准引用格式的`fmt.`来作为前缀使用 fmt 包中的方法。

```
package main
import F "fmt"
func main() {
    F.Println("包子好帅")
}
```

省略引用格式

```
import . "fmt"
```

这种格式相当于把 fmt 包直接合并到当前程序中，在使用 fmt 包内的方法是可以不用加前缀`fmt.`，直接引用。

```
package main
import . "fmt"
func main() {
    //不需要加前缀 fmt.
    Println("C语言中文网")
}
```

匿名引用格式

在引用某个包时，如果只是希望执行包初始化的 init 函数，而不使用包内部的数据时，可以使用匿名引用格式。

```
import _ "fmt"
```

匿名导入的包与其他方式导入的包一样都会被编译到可执行文件中。

使用标准格式引用包，但是代码中却没有使用包，编译器会报错。如果包中有 init 初始化函数，则通过`import _ "包的路径"`这种方式引用包，仅执行包的初始化函数，即使包没有 init 初始化函数，也不会引发编译器报错。

```
package main
import (
    _ "database/sql"
    "fmt"
)
func main() {
    fmt.Println("C语言中文网")
}
```

注意事项

- 一个包可以有多个 init 函数，包加载时会执行全部的 init 函数，但并不能保证执行顺序，所以不建议在一个包中放入多个 init 函数，将需要初始化的逻辑放到一个 init 函数里面。
- 包不能出现环形引用的情况，比如包 a 引用了包 b，包 b 引用了包 c，如果包 c 又引用了包 a，则编译不能通过。
- 包的重复引用是允许的，比如包 a 引用了包 b 和包 c，包 b 和包 c 都引用了包 d。这种场景相当于重复引用了 d，这种情况是允许的，并且 Go 编译器保证包 d 的 init 函数只会执行一次。

包加载

Go 程序的启动和加载过程，在执行 main 包的 mian 函数之前， Go 引导程序会先对整个程序的包进行初始化。

Go语言包的初始化有如下特点：

- 包初始化程序从 main 函数引用的包开始，逐级查找包的引用，直到找到没有引用其他包的包，最终生成一个包引用的有向无环图。
- Go 编译器会将有向无环图转换为一棵树，然后从树的叶子节点开始逐层向上对包进行初始化。
- 单个包的初始化过程如上图所示，先初始化常量，然后是全局变量，最后执行包的 init 函数。

包管理工具

GOPATH

1. 在 1.8 版本前必须设置这个环境变量。
2. 1.8 版本后（含 1.8）如果没有设置使用用默认值。
    - 在 Unix 上默认为 $HOME/go , 在 Windows 上默认为 %USERPROFILE%/go
    - 在 Mac 上 GOPATH 可以通过修改 ~/.bash_profile 来设置

在 go mod 出现之前，所有的 Go 项目都需要放在同一个工作空间：`$GOPATH/src` 内，比如：

```
src/
    github.com/golang/example/
        .git/                       # Git repository metadata
    outyet/
        main.go                # command source
        main_test.go           # test source
    stringutil/
        reverse.go             # package source
        reverse_test.go        # test source
```

相比其他语言，这个限制有些无法理解。其实，这和 Go 的一设计理念紧密相关：**包管理应该是去中心化的。**

所以 Go 里面没有 maven/npm 之类的包管理工具，只有一个 go get，支持从公共的代码托管平台（Bitbucket/GitHub..）下载依赖，当然也支持自己托管，具体可参考官方文档：Remote import paths（自行搜索下，这里只做了解）。

由于没有中央仓库，所以 Go 项目位置决定了其 import path，同时为了与 go get 保持一致，所以一般来说我们的项目名称都是 github.com/user/repo 的形式。
当然也可以不是这种形式，只是不方便别人引用而已，后面会讲到如何在 go mod 中实现这种效果。

vendor、dep

使用 go get 下载依赖的方式简单暴力，伴随了 Go 七年之久，直到 1.6（2016/02/17）才正式支持了 vendor，可以把所有依赖下载到当前项目中，解决可重复构建（reproducible builds）的问题，但是无法管理依赖版本。社区出现了各式各样的包管理工具，来方便开发者固化依赖版本，由于不同管理工具采用不同的元信息格式（比如：godep 的 Godeps.json、Glide 的 glide.yaml），不利于社区发展，所以 Go 官方推出了 dep。

dep 的定位是实验、探索如何管理版本，并不会直接集成到 Go 工具链，Go 核心团队会吸取 dep 使用经验与社区反馈，开发下一代包管理工具 modules，并于 2019/09/03 发布的 1.13 正式支持，并随之发布 Module Mirror, Index, Checksum，用于解决软件分发、中间人攻击等问题。

Go modules（重点）

Go modules是golang 1.11新加的特性，用来管理模块中包的依赖关系。

GO111MODULE有三个值：auto、on和off，默认值为auto。

GO111MODULE的值会直接影响Go compiler的"依赖管理"模式的选择（是GOPATH mode还是module-aware mode），我们详细来看一下：

- 当GO111MODULE的值为off时，go compiler显然会始终使用GOPATH mode，即无论要构建的源码目录是否在GOPATH路径下，go compiler都会在传统的GOPATH和vendor目录(仅支持在gopath目录下的package)下搜索目标程序依赖的go package。

- 当GO111MODULE的值为on时（export GO111MODULE=on），与off相反，go compiler会始终使用module-aware mode，即无论要构建的源码目录是否在GOPATH路径下，go compiler都不会在传统的GOPATH和vendor目录下搜索目标程序依赖的go package，而是在go mod命令的缓存目录($GOPATH/pkg/mod）下搜索对应版本的依赖package。
- 当GO111MODULE的值为auto时(不显式设置即为auto)，也就是我们在上面的例子中所展现的那样：使用GOPATH mode还是module-aware mode，取决于要构建的源码目录所在位置以及是否包含go.mod文件。
    - 如果要构建的源码目录不在以GOPATH/src为根的目录体系下，且包含go.mod文件(两个条件缺一不可)，那么使用module-aware mode；
- 否则使用传统的GOPATH mode。

Module 文件

执行命令 `go build && go mod tidy`，下载依赖并整理。

项目根目录下会生成两个文件（需要加入到 git 中）：

- 文件 `go.mod`：指示模块名称、go 的版本、该模块的依赖信息（依赖名称）。
- 文件 `go.sum`：该模块的所有依赖的校验和。

Module 是多个 package 的集合，版本管理的基本单元，使用 go.mod 文件记录依赖的 module。

go.mod 位于项目的根目录，支持 4 条命令：module、require、replace、exclude。

示例：

```
module github.com/my/repo

require (
    github.com/some/dependency v1.2.3
    github.com/another/dependency/v4 v4.0.0
)
```

- module 声明 module path，一个 module 内所有 package 的 import path 都以它为前缀。
- require 声明所依赖的 module，版本信息使用形如 v(major).(minor).(patch) 的语义化版本 。
- replace/exclude 用于替换、排查指定 module path。

Go mod使用方法

Go语言提供了 `go mod`命令来管理包。

- 初始化模块

```
Go mod init <项目模块名称>
```

- 依赖关系处理，根据go.mod文件

```
Go mod tidy
```

- 将依赖包复制到项目的vendor目录

```
Go mod vendor
```

如果报被屏蔽（墙），可以使用这个命令，随后使用go build -mod=vendor编译

- 显示依赖关系

```
Go list -m all
```

- 显示详细依赖关系

```
Go list -m -json all
```

- 下载依赖

```
Go mod download [path@version]
```

- 其他：

命令	说明
download	download modules to local cache(下载依赖包)
edit	edit go.mod from tools or scripts（编辑go.mod）
graph	print module requirement graph (打印模块依赖图)
init	initialize new module in current directory（在当前目录初始化mod）
tidy	add missing and remove unused modules(拉取缺少的模块，移除不用的模块)
vendor	make vendored copy of dependencies(将依赖复制到vendor下)
verify	verify dependencies have expected content (验证依赖是否正确)
why	explain why packages or modules are needed(解释为什么需要依赖)

Go modules使用步骤：

1. 首先将你的版本更新到最新的Go版本(>=1.11)。
2. 通过go命令行，进入到你当前的工程目录下，**在命令行设置环境变量**：

```
#方式1：临时设置

注意

# Windows环境用set
# linux环境用export

#########Windows#########
# 开启
set GO111MODULE=on
# 1.13 之后才支持多个地址，之前版本只支持一个
set GOPROXY=https://goproxy.cn,https://mirrors.aliyun.com/goproxy,direct
# 1.13 开始支持，配置私有 module，不去校验 checksum
set GOPRIVATE=*.corp.example.com,rsc.io/private
```

```
#########Linux#########
# 开启
export GO111MODULE=on
# 1.13 之后才支持多个地址，之前版本只支持一个
export GOPROXY=https://goproxy.cn,https://mirrors.aliyun.com/goproxy,direct
# 1.13 开始支持，配置私有 module，不去校验 checksum
export GOPRIVATE=*.corp.example.com,rsc.io/private

#方式2：全局设置
# 设置全局开启 go mod Go1.16版本默认为on，可跳过这一步
go env -w GO111MODULE=on
# 设置全局代理地址
go env -w
GOPROXY=https://goproxy.cn,https://mirrors.aliyun.com/goproxy,direct
```

3. 执行命令go mod init在当前目录下生成一个go.mod文件，执行这条命令时，当前目录不能存在go.mod文件。如果之前生成过，要先删除。
4. 如果你工程中存在一些不能确定版本的包，那么生成的go.mod文件可能就不完整，因此继续执行下面的命令；
5. 执行go mod tidy命令，它会添加缺失的模块以及移除不需要的模块。执行后会生成go.sum文件(模块下载条目)。添加参数-v，例如go mod tidy -v可以将执行的信息，即删除和添加的包打印到命令行；
6. 执行命令go mod verify来检查当前模块的依赖是否全部下载下来，是否下载下来被修改过。如果所有的模块都没有被修改过，那么执行这条命令之后，会打印all modules verified。
7. 执行命令go mod vendor生成vendor文件夹，该文件夹下将会放置你go.mod文件描述的依赖包，文件夹下同时还有一个文件modules.txt，它是你整个工程的所有模块。在执行这条命令之前，如果你工程之前有vendor目录，应该先进行删除。同理go mod vendor -v会将添加到vendor中的模块打印出来；

Go module的文件下载后位置：

```
存储下载的依赖包,具体位置在$GOPATH/pkg/mod
```

在 Go 1.8 版本之前，GOPATH 环境变量默认是空的。从 Go 1.8 版本开始，Go 开发包在安装完成后，将 GOPATH 赋予了一个默认的目录，参见下表。

平 台	GOPATH 默认值	举 例
Windows 平台	%USERPROFILE%/go	C:\Users\用户名\go
Unix 平台	$HOME/go	/home/用户名/go

代理地址

```
https://goproxy.cn  //七牛云赞助支持的开源代理
https://mirrors.aliyun.com/goproxy  //阿里云官方维护的go代理
https://goproxy.io //也是一个开源的go代理
```