# Go语言包（package），Golang包（package）

< [示例：二叉树数据结构的应用](http://c.biancheng.net/view/vip_7337.html)[包的基本概念](http://c.biancheng.net/view/5394.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/golang/package/_blank)

Go 语言的源码复用建立在包（package）基础之上。Go 语言的入口 main() 函数所在的包（package）叫 main，main 包想要引用别的代码，必须同样以包的方式进行引用，本章内容将详细讲解如何导出包的内容及如何导入其他包。  
  
Go 语言的包与文件夹一一对应，所有与包相关的操作，必须依赖于工作目录（GOPATH）。

# Go语言包的基本概念

< [Go语言包（package）](http://c.biancheng.net/golang/package/)[Go语言封装简介及实现细节](http://c.biancheng.net/view/5715.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

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。  
  
包可以定义在很深的目录中，包名的定义是不包括目录路径的，但是包在引用时一般使用全路径引用。比如在GOPATH/src/a/b/ 下定义一个包 c。在包 c 的源码中只需声明为package c，而不是声明为package a/b/c，但是在导入 c 包时，需要带上路径，例如import "a/b/c"。  
  
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。

## 包的导入

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，下面以 fmt 包为例来分别演示一下这四种格式。

#### 1) 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。  
  
示例代码如下：

1. package main
2. import "fmt"
3. func main() {
4. fmt.Println("C语言中文网")
5. }

#### 2) 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，如下所示：

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。  
  
示例代码如下：

1. package main
2. import F "fmt"
3. func main() {
4. F.Println("C语言中文网")
5. }

#### 3) 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。  
  
示例代码如下：

1. package main
2. import . "fmt"
3. func main() {
4. //不需要加前缀 fmt.
5. Println("C语言中文网")
6. }

#### 4) 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，如下所示：

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。  
  
示例代码如下：

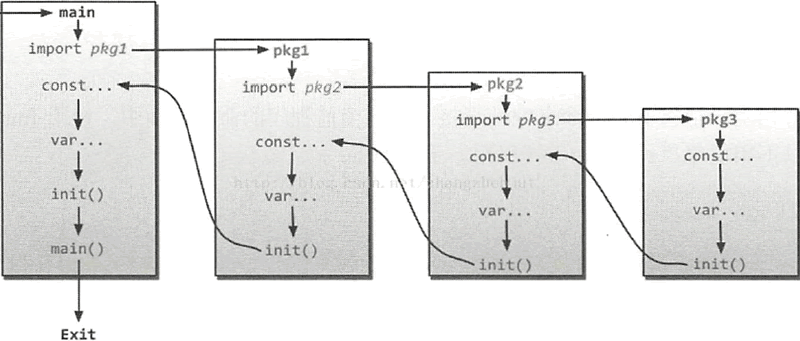
1. package main
2. import (
3. \_ "database/sql"
4. "fmt"
5. )
6. func main() {
7. fmt.Println("C语言中文网")
8. }

注意：

* 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

通过前面一系列的学习相信大家已经大体了解了 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。整个执行的流程如下图所示。

  
图：Go 包的初始化

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。

# Go语言封装简介及实现细节

< [包的基本概念](http://c.biancheng.net/view/5394.html)[Go语言GOPATH](http://c.biancheng.net/view/88.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在Go语言中封装就是把抽象出来的字段和对字段的操作封装在一起，数据被保护在内部，程序的其它包只能通过被授权的方法，才能对字段进行操作。  
  
封装的好处：

* 隐藏实现细节；
* 可以对数据进行验证，保证数据安全合理。

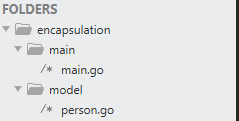
如何体现封装：

* 对结构体中的属性进行封装；
* 通过方法，包，实现封装。

封装的实现步骤：

* 将结构体、字段的首字母小写；
* 给结构体所在的包提供一个工厂模式的函数，首字母大写，类似一个构造函数；
* 提供一个首字母大写的 Set 方法（类似其它语言的 public），用于对属性判断并赋值；
* 提供一个首字母大写的 Get 方法（类似其它语言的 public），用于获取属性的值。

【示例】对于员工，不能随便查看年龄，工资等隐私，并对输入的年龄进行合理的验证。代码结构如下：



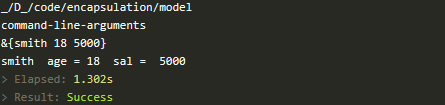
person.go 中的代码如下所示：

1. package model
2. import "fmt"
3. type person struct {
4. Name string
5. age int //其它包不能直接访问..
6. sal float64
7. }
8. //写一个工厂模式的函数，相当于构造函数
9. func NewPerson(name string) \*person {
10. return &person{
11. Name : name,
12. }
13. }
14. //为了访问age 和 sal 我们编写一对SetXxx的方法和GetXxx的方法
15. func (p \*person) SetAge(age int) {
16. if age >0 && age <150 {
17. p.age = age
18. } else {
19. fmt.Println("年龄范围不正确..")
20. //给程序员给一个默认值
21. }
22. }
23. func (p \*person) GetAge() int {
24. return p.age
25. }
26. func (p \*person) SetSal(sal float64) {
27. if sal >= 3000 && sal <= 30000 {
28. p.sal = sal
29. } else {
30. fmt.Println("薪水范围不正确..")
31. }
32. }
33. func (p \*person) GetSal() float64 {
34. return p.sal
35. }

main.go 中的代码如下所示：

1. package main
2. import (
3. "fmt"
4. "../model"
5. )
6. func main() {
7. p := model.NewPerson("smith")
8. p.SetAge(18)
9. p.SetSal(5000)
10. fmt.Println(p)
11. fmt.Println(p.Name, " age =", p.GetAge(), " sal = ", p.GetSal())
12. }

执行效果如下图所示：



# Go语言GOPATH详解（Go语言工作目录）

< [Go语言封装简介及实现细节](http://c.biancheng.net/view/5715.html)[Go语言常用内置包](http://c.biancheng.net/view/4306.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

GOPATH 是 Go语言中使用的一个环境变量，它使用绝对路径提供项目的工作目录。  
  
工作目录是一个工程开发的相对参考目录，好比当你要在公司编写一套服务器代码，你的工位所包含的桌面、计算机及椅子就是你的工作区。工作区的概念与工作目录的概念也是类似的。如果不使用工作目录的概念，在多人开发时，每个人有一套自己的目录结构，读取配置文件的位置不统一，输出的二进制运行文件也不统一，这样会导致开发的标准不统一，影响开发效率。  
  
GOPATH 适合处理大量 Go语言源码、多个包组合而成的复杂工程。

#### 提示

C、[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank)、[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank)、[C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 及其他语言发展到后期，都拥有自己的 IDE（集成开发环境），并且工程（Project）、解决方案（Solution）和工作区（Workspace）等概念将源码和资源组织了起来，方便编译和输出。

## 使用命令行查看GOPATH信息

在《[安装Go语言开发包](http://c.biancheng.net/view/3992.html" \t "http://c.biancheng.net/view/_blank)》一节中我们已经介绍过 Go语言的安装方法。在安装过 Go 开发包的操作系统中，可以使用命令行查看 Go 开发包的环境变量配置信息，这些配置信息里可以查看到当前的 GOPATH 路径设置情况。在命令行中运行go env后，命令行将提示以下信息：

$ go env  
GOARCH="amd64"  
GOBIN=""  
GOEXE=""  
GOHOSTARCH="amd64"  
GOHOSTOS="linux"  
GOOS="linux"  
GOPATH="/home/davy/go"  
GORACE=""  
GOROOT="/usr/local/go"  
GOTOOLDIR="/usr/local/go/pkg/tool/linux\_amd64"  
[GCC](http://c.biancheng.net/gcc/" \t "http://c.biancheng.net/view/_blank)GO="gccgo"  
CC="gcc"  
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0"  
CXX="g++"  
CGO\_ENABLED="1"  
CGO\_CFLAGS="-g -O2"  
CGO\_CPPFLAGS=""  
CGO\_CXXFLAGS="-g -O2"  
CGO\_FFLAGS="-g -O2"  
CGO\_LDFLAGS="-g -O2"  
PKG\_CONFIG="pkg-config"

命令行说明如下：

* 第 1 行，执行 go env 指令，将输出当前 Go 开发包的环境变量状态。
* 第 2 行，GOARCH 表示目标处理器架构。
* 第 3 行，GOBIN 表示编译器和链接器的安装位置。
* 第 7 行，GOOS 表示目标操作系统。
* 第 8 行，GOPATH 表示当前工作目录。
* 第 10 行，GOROOT 表示 Go 开发包的安装目录。

从命令行输出中，可以看到 GOPATH 设定的路径为：/home/davy/go（davy 为笔者的用户名）。  
  
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。

|  |  |  |
| --- | --- | --- |
| GOPATH 在不同平台上的安装路径 | | |
| **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 |

## 使用GOPATH的工程结构

在 GOPATH 指定的工作目录下，代码总是会保存在 $GOPATH/src 目录下。在工程经过 go build、go install 或 go get 等指令后，会将产生的二进制可执行文件放在 $GOPATH/bin 目录下，生成的中间缓存文件会被保存在 $GOPATH/pkg 下。  
  
如果需要将整个源码添加到版本管理工具（Version Control System，VCS）中时，只需要添加 $GOPATH/src 目录的源码即可。bin 和 pkg 目录的内容都可以由 src 目录生成。

## 设置和使用GOPATH

本节以 Linux 为演示平台，为大家演示使用 GOPATH 的方法。

#### 1) 设置当前目录为GOPATH

选择一个目录，在目录中的命令行中执行下面的指令：

export GOPATH=`pwd`

该指令中的 pwd 将输出当前的目录，使用反引号`将 pwd 指令括起来表示命令行替换，也就是说，使用`pwd`将获得 pwd 返回的当前目录的值。例如，假设你的当前目录是“/home/davy/go”，那么使用`pwd`将获得返回值“/home/davy/go”。  
  
使用 export 指令可以将当前目录的值设置到环境变量 GOPATH中。

#### 2) 建立GOPATH中的源码目录

使用下面的指令创建 GOPATH 中的 src 目录，在 src 目录下还有一个 hello 目录，该目录用于保存源码。

mkdir -p src/hello

mkdir 指令的 -p 可以连续创建一个路径。

#### 3) 添加main.go源码文件

使用 Linux 编辑器将下面的源码保存为 main.go 并保存到 $GOPATH/src/hello 目录下。

1. package main
2. import "fmt"
3. func main(){
4. fmt.Println("hello")
5. }

#### 4) 编译源码并运行

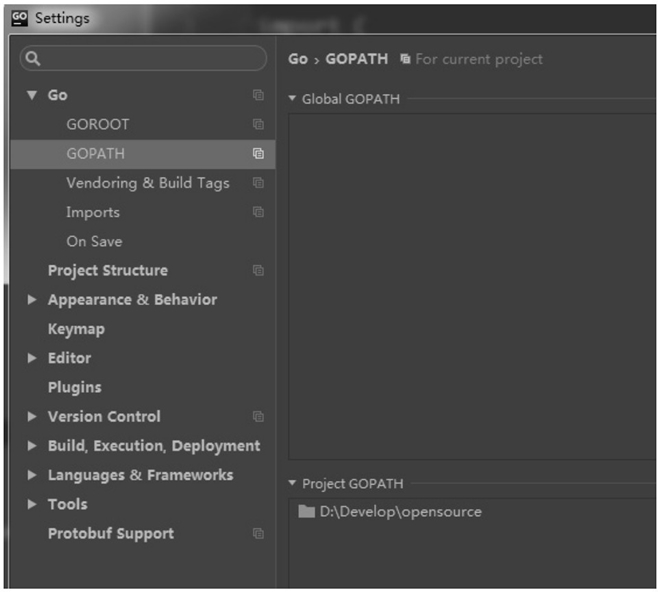
此时我们已经设定了 GOPATH，因此在 Go语言中可以通过 GOPATH 找到工程的位置。  
  
在命令行中执行如下指令编译源码：

go install hello

编译完成的可执行文件会保存在 $GOPATH/bin 目录下。  
  
在 bin 目录中执行 ./hello，命令行输出如下：  
hello world

## 在多项目工程中使用GOPATH

在很多与 Go语言相关的书籍、文章中描述的 GOPATH 都是通过修改系统全局的环境变量来实现的。然而，根据笔者多年的 Go语言使用和实践经验及周边朋友、同事的反馈，这种设置全局 GOPATH 的方法可能会导致当前项目错误引用了其他目录的 Go 源码文件从而造成编译输出错误的版本或编译报出一些无法理解的错误提示。  
  
比如说，将某项目代码保存在 /home/davy/projectA 目录下，将该目录设置为 GOPATH。随着开发进行，需要再次获取一份工程项目的源码，此时源码保存在 /home/davy/projectB 目录下，如果此时需要编译 projectB 目录的项目，但开发者忘记设置 GOPATH 而直接使用命令行编译，则当前的 GOPATH 指向的是 /home/davy/projectA 目录，而不是开发者编译时期望的 projectB 目录。编译完成后，开发者就会将错误的工程版本发布到外网。  
  
因此，建议大家无论是使用命令行或者使用集成开发环境编译 Go 源码时，GOPATH 跟随项目设定。在 Jetbrains 公司的 GoLand 集成开发环境（IDE）中的 GOPATH 设置分为全局 GOPATH 和项目 GOPATH，如下图所示。

  
图：全局和项目GOPATH

图中的 Global GOPATH 代表全局 GOPATH，一般来源于系统环境变量中的 GOPATH；Project GOPATH 代表项目所使用的 GOPATH，该设置会被保存在工作目录的 .idea 目录下，不会被设置到环境变量的 GOPATH 中，但会在编译时使用到这个目录。建议在开发时只填写项目 GOPATH，每一个项目尽量只设置一个 GOPATH，不使用多个 GOPATH 和全局的 GOPATH。

#### 提示

Visual Studio 早期在设计时，允许 C++ 语言在全局拥有一个包含路径。当一个工程多个版本的编译，或者两个项目混杂有不同的共享全局包含时，会发生难以察觉的错误。在新版本 Visual Studio 中已经废除了这种全局包含的路径设计，并建议开发者将包含目录与项目关联。  
  
Go语言中的 GOPATH 也是一种类似全局包含的设计，因此鉴于 Visual Studio 在设计上的失误，建议开发者不要设置全局的 GOPATH，而是随项目设置 GOPATH。

# Go语言常用内置包简介

< [Go语言GOPATH](http://c.biancheng.net/view/88.html)[Go语言自定义包](http://c.biancheng.net/view/5123.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

标准的Go语言代码库中包含了大量的包，并且在安装 Go 的时候多数会自动安装到系统中。我们可以在 $GOROOT/src/pkg 目录中查看这些包。下面简单介绍一些我们开发中常用的包。

#### 1) fmt

fmt 包实现了格式化的标准输入输出，这与C语言中的 printf 和 scanf 类似。其中的 fmt.Printf() 和 fmt.Println() 是开发者使用最为频繁的函数。  
  
格式化短语派生于C语言，一些短语（%- 序列）是这样使用：

* %v：默认格式的值。当打印结构时，加号（%+v）会增加字段名；
* %#v：Go样式的值表达；
* %T：带有类型的 Go 样式的值表达。

#### 2) io

这个包提供了原始的 I/O 操作界面。它主要的任务是对 os 包这样的原始的 I/O 进行封装，增加一些其他相关，使其具有抽象功能用在公共的接口上。

#### 3) bufio

bufio 包通过对 io 包的封装，提供了数据缓冲功能，能够一定程度减少大块数据读写带来的开销。  
  
在 bufio 各个组件内部都维护了一个缓冲区，数据读写操作都直接通过缓存区进行。当发起一次读写操作时，会首先尝试从缓冲区获取数据，只有当缓冲区没有数据时，才会从数据源获取数据更新缓冲。

#### 4) sort

sort 包提供了用于对切片和用户定义的集合进行排序的功能。

#### 5) strconv

strconv 包提供了将字符串转换成基本数据类型，或者从基本数据类型转换为字符串的功能。

#### 6) os

os 包提供了不依赖平台的操作系统函数接口，设计像 Unix 风格，但错误处理是 go 风格，当 os 包使用时，如果失败后返回错误类型而不是错误数量。

#### 7) sync

sync 包实现多线程中锁机制以及其他同步互斥机制。

#### 8) flag

flag 包提供命令行参数的规则定义和传入参数解析的功能。绝大部分的命令行程序都需要用到这个包。

#### 9) encoding/json

JSON 目前广泛用做网络程序中的通信格式。encoding/json 包提供了对 JSON 的基本支持，比如从一个对象序列化为 JSON 字符串，或者从 JSON 字符串反序列化出一个具体的对象等。

#### 10) html/template

主要实现了 web 开发中生成 html 的 template 的一些函数。

#### 11) net/http

net/http 包提供 HTTP 相关服务，主要包括 http 请求、响应和 URL 的解析，以及基本的 http 客户端和扩展的 http 服务。  
  
通过 net/http 包，只需要数行代码，即可实现一个爬虫或者一个 Web 服务器，这在传统语言中是无法想象的。

#### 12) reflect

reflect 包实现了运行时反射，允许程序通过抽象类型操作对象。通常用于处理静态类型 interface{} 的值，并且通过 Typeof 解析出其动态类型信息，通常会返回一个有接口类型 Type 的对象。

#### 13) os/exec

os/exec 包提供了执行自定义 linux 命令的相关实现。

#### 14) strings

strings 包主要是处理字符串的一些函数集合，包括合并、查找、分割、比较、后缀检查、索引、大小写处理等等。  
  
strings 包与 bytes 包的函数接口功能基本一致。

#### 15) bytes

bytes 包提供了对字节切片进行读写操作的一系列函数。字节切片处理的函数比较多，分为基本处理函数、比较函数、后缀检查函数、索引函数、分割函数、大小写处理函数和子切片处理函数等。

#### 16) log

log 包主要用于在程序中输出日志。  
  
log 包中提供了三类日志输出接口，Print、Fatal 和 Panic。

* Print 是普通输出；
* Fatal 是在执行完 Print 后，执行 os.Exit(1)；
* Panic 是在执行完 Print 后调用 panic() 方法。

# Go语言自定义包

< [Go语言常用内置包](http://c.biancheng.net/view/4306.html)[Go语言package](http://c.biancheng.net/view/89.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

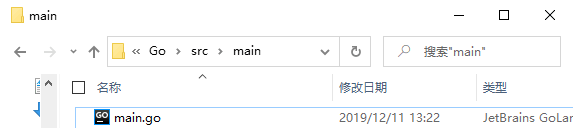
包是Go语言中代码组成和代码编译的主要方式。关于包的基本信息我们已经在前面介绍过了，本节我们主要来介绍一下如何自定义一个包并使用它。  
  
到目前为止，我们所使用的例子都是以一个包的形式存在的，比如 main 包。在Go语言里，允许我们将同一个包的代码分隔成多个独立的源码文件来单独保存，只需要将这些文件放在同一个目录下即可。  
  
我们创建的自定义的包需要将其放在 GOPATH 的 src 目录下（也可以是 src 目录下的某个子目录），而且两个不同的包不能放在同一目录下，这样会引起编译错误。  
  
一个包中可以有任意多个文件，文件的名字也没有任何规定（但后缀必须是 .go），这里我们假设包名就是 .go 的文件名（如果一个包有多个 .go 文件，则其中会有一个 .go 文件的文件名和包名相同）。  
  
下面通过示例来演示一下如何创建一个名为 demo 的自定义包，并在 main 包中使用自定义包 demo 中的方法。  
  
首先，在 GOPATH 下的 src 目录中新建一个 demo 文件夹 ，并在 demo 文件夹下创建 demo.go 文件，如下所示：

  
图：demo 文件夹

demo.go 文件的代码如下所示：

1. package demo
2. import (
3. "fmt"
4. )
5. func PrintStr() {
6. fmt.Println("C语言中文网")
7. }

然后，在 GOPATH 下的 src 目录中新建一个 main 文件夹，并在 main 文件夹下创建 mian.go 文件，如下所示：

  
图：main 文件夹

main.go 文件的代码如下所示：

1. package main
2. import (
3. "demo"
4. )
5. func main() {
6. demo.PrintStr()
7. }

运行结果如下所示：

go run main.go  
C语言中文网

对引用自定义包需要注意以下几点：

* 如果项目的目录不在 GOPATH 环境变量中，则需要把项目移到 GOPATH 所在的目录中，或者将项目所在的目录设置到 GOPATH 环境变量中，否则无法完成编译；
* 使用 import 语句导入包时，使用的是包所属文件夹的名称；
* 包中的函数名第一个字母要大写，否则无法在外部调用；
* 自定义包的包名不必与其所在文件夹的名称保持一致，但为了便于维护，建议保持一致；
* 调用自定义包时使用 包名 . 函数名 的方式，如上例：demo.PrintStr()。

# Go语言package（创建包）

< [Go语言自定义包](http://c.biancheng.net/view/5123.html)[Go语言导出包中的标识符](http://c.biancheng.net/view/90.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

包（package）是多个 Go 源码的集合，是一种高级的代码复用方案，像 fmt、os、io 等这样具有常用功能的内置包在 Go语言中有 150 个以上，它们被称为标准库，大部分（一些底层的除外）内置于 Go 本身。  
  
包要求在同一个目录下的所有文件的第一行添加如下代码，以标记该文件归属的包：

package 包名

包的特性如下：

* 一个目录下的同级文件归属一个包。
* 包名可以与其目录不同名。
* 包名为 main 的包为应用程序的入口包，编译源码没有 main 包时，将无法编译输出可执行的文件。

任何包系统设计的目的都是为了简化大型程序的设计和维护工作，通过将一组相关的特性放进一个独立的单元以便于理解和更新，在每个单元更新的同时保持和程序中其它单元的相对独立性。这种模块化的特性允许每个包可以被其它的不同项目共享和重用，在项目范围内、甚至全球范围统一的分发和复用。  
  
每个包一般都定义了一个不同的名字空间用于它内部的每个标识符的访问。每个名字空间关联到一个特定的包，让我们给类型、函数等选择简短明了的名字，这样可以避免在我们使用它们的时候减少和其它部分名字的冲突。  
  
每个包还通过控制包内名字的可见性和是否导出来实现封装特性。通过限制包成员的可见性并隐藏包 API 的具体实现，将允许包的维护者在不影响外部包用户的前提下调整包的内部实现。通过限制包内变量的可见性，还可以强制用户通过某些特定函数来访问和更新内部变量，这样可以保证内部变量的一致性和并发时的互斥约束。  
  
当我们修改了一个源文件，我们必须重新编译该源文件对应的包和所有依赖该包的其他包。即使是从头构建，Go语言编译器的编译速度也明显快于其它编译语言。Go语言的闪电般的编译速度主要得益于三个语言特性。

* 第一点，所有导入的包必须在每个文件的开头显式声明，这样的话编译器就没有必要读取和分析整个源文件来判断包的依赖关系。
* 第二点，禁止包的环状依赖，因为没有循环依赖，包的依赖关系形成一个有向无环图，每个包可以被独立编译，而且很可能是被并发编译。
* 第三点，编译后包的目标文件不仅仅记录包本身的导出信息，目标文件同时还记录了包的依赖关系。因此，在编译一个包的时候，编译器只需要读取每个直接导入包的目标文件，而不需要遍历所有依赖的的文件。

# Go语言导出包中的标识符——让外部访问包的类型和值

< [Go语言package](http://c.biancheng.net/view/89.html)[Go语言import导入包](http://c.biancheng.net/view/91.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在 Go语言中，如果想在一个包里引用另外一个包里的标识符（如类型、变量、常量等）时，必须首先将被引用的标识符导出，将要导出的标识符的首字母大写就可以让引用者可以访问这些标识符了。

## 导出包内标识符

下面代码中包含一系列未导出标识符，它们的首字母都为小写，这些标识符可以在包内自由使用，但是包外无法访问它们，代码如下：

1. package mypkg
2. var myVar = 100
3. const myConst = "hello"
4. type myStruct struct {
5. }

将 myStruct 和 myConst 首字母大写，导出这些标识符，修改后代码如下：

1. package mypkg
2. var myVar = 100
3. const MyConst = "hello"
4. type MyStruct struct {
5. }

此时，MyConst 和 MyStruct 可以被外部访问，而 myVar 由于首字母是小写，因此只能在 mypkg 包内使用，不能被外部包引用。

## 导出结构体及接口成员

在被导出的结构体或接口中，如果它们的字段或方法首字母是大写，外部可以访问这些字段和方法，代码如下：

1. type MyStruct struct {
2. // 包外可以访问的字段
3. ExportedField int
4. // 仅限包内访问的字段
5. privateField int
6. }
7. type MyInterface interface {
8. // 包外可以访问的方法
9. ExportedMethod()
10. // 仅限包内访问的方法
11. privateMethod()
12. }

在代码中，MyStruct 的 ExportedField 和 MyInterface 的 ExportedMethod() 可以被包外访问。

# Go语言import导入包——在代码中使用其他的代码

< [Go语言导出包中的标识符](http://c.biancheng.net/view/90.html)[Go语言工厂模式自动注册](http://c.biancheng.net/view/vip_7338.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

可以在一个 Go语言源文件包声明语句之后，其它非导入声明语句之前，包含零到多个导入包声明语句。每个导入声明可以单独指定一个导入路径，也可以通过圆括号同时导入多个导入路径。要引用其他包的标识符，可以使用 import 关键字，导入的包名使用双引号包围，包名是从 GOPATH 开始计算的路径，使用/进行路径分隔。

## 默认导入的写法

导入有两种基本格式，即单行导入和多行导入，两种导入方法的导入代码效果是一致的。

#### 1) 单行导入

单行导入格式如下：

import "包1"  
import "包2"

#### 2) 多行导入

当多行导入时，包名在 import 中的顺序不影响导入效果，格式如下：

import(  
    "包1"  
    "包2"  
    …  
)

参考代码 8-1 的例子来理解 import 的机制。

本套教程所有源码下载地址：[https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ](https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ" \t "http://c.biancheng.net/view/_blank)    提取密码：hfyf

代码 8-1 的目录层次如下：

.  
└── src  
    └── chapter08  
        └── importadd  
            ├── main.go  
            └── mylib  
                └── add.go

代码8-1　加函数（具体文件：…/chapter08/importadd/mylib/add.go）

1. package mylib
2. func Add(a, b int) int {
3. return a + b
4. }

第 3 行中的 Add() 函数以大写 A 开头，表示将 Add() 函数导出供包外使用。当首字母小写时，为包内使用，包外无法引用到。  
  
add.go 在 mylib 文件夹下，习惯上将文件夹的命名与包名一致，命名为 mylib 包。  
  
代码8-2　导入包（具体文件：…/chapter08/importadd/main.go）

1. package main
2. import (
3. "chapter08/importadd/mylib"
4. "fmt"
5. )
6. func main() {
7. fmt.Println(mylib.Add(1, 2))
8. }

代码说明如下：

* 第 4 行，导入 chapter08/importadd/mylib 包。
* 第 9 行，使用 mylib 作为包名，并引用 Add() 函数调用。

在命令行中运行下面代码：

export GOPATH=/home/davy/golangbook/code  
go install chapter08/importadd  
$GOPATH/bin/importadd

命令说明如下：

* 第 1 行，根据你的 GOPATH 不同，设置 GOPATH。
* 第 2 行，使用 go install 指令编译并安装 chapter08/code8-1 到 GOPATH 的 bin 目录下。
* 第 3 行，执行 GOPATH 的 bin 目录下的可执行文件 code8-1。

运行代码，输出结果如下：

3

导入的包之间可以通过添加空行来分组；通常将来自不同组织的包独自分组。包的导入顺序无关紧要，但是在每个分组中一般会根据字符串顺序排列。（gofmt 和 goimports 工具都可以将不同分组导入的包独立排序。）

1. import (
2. "fmt"
3. "html/template"
4. "os"
5. "golang.org/x/net/html"
6. "golang.org/x/net/ipv4"
7. )

## 导入包后自定义引用的包名

如果我们想同时导入两个有着名字相同的包，例如 math/rand 包和 crypto/rand 包，那么导入声明必须至少为一个同名包指定一个新的包名以避免冲突。这叫做导入包的重命名。

1. import (
2. "crypto/rand"
3. mrand "math/rand" // 将名称替换为mrand避免冲突
4. )

导入包的重命名只影响当前的源文件。其它的源文件如果导入了相同的包，可以用导入包原本默认的名字或重命名为另一个完全不同的名字。  
  
导入包重命名是一个有用的特性，它不仅仅只是为了解决名字冲突。如果导入的一个包名很笨重，特别是在一些自动生成的代码中，这时候用一个简短名称会更方便。选择用简短名称重命名导入包时候最好统一，以避免包名混乱。选择另一个包名称还可以帮助避免和本地普通变量名产生冲突。例如，如果文件中已经有了一个名为 path 的变量，那么我们可以将"path"标准包重命名为 pathpkg。  
  
每个导入声明语句都明确指定了当前包和被导入包之间的依赖关系。如果遇到包循环导入的情况，Go语言的构建工具将报告错误。

## 匿名导入包——只导入包但不使用包内类型和数值

如果只希望导入包，而不使用任何包内的结构和类型，也不调用包内的任何函数时，可以使用匿名导入包，格式如下：

1. import (
2. \_ "path/to/package"
3. )

其中，path/to/package 表示要导入的包名，下画线\_表示匿名导入包。  
  
匿名导入的包与其他方式导入包一样会让导入包编译到可执行文件中，同时，导入包也会触发 init() 函数调用。

## 包在程序启动前的初始化入口：init

在某些需求的设计上需要在程序启动时统一调用程序引用到的所有包的初始化函数，如果需要通过开发者手动调用这些初始化函数，那么这个过程可能会发生错误或者遗漏。我们希望在被引用的包内部，由包的编写者获得代码启动的通知，在程序启动时做一些自己包内代码的初始化工作。  
  
例如，为了提高数学库计算三角函数的执行效率，可以在程序启动时，将三角函数的值提前在内存中建成索引表，外部程序通过查表的方式迅速获得三角函数的值。但是三角函数索引表的初始化函数的调用不希望由每一个外部使用三角函数的开发者调用，如果在三角函数的包内有一个机制可以告诉三角函数包程序何时启动，那么就可以解决初始化的问题。  
  
Go 语言为以上问题提供了一个非常方便的特性：init() 函数。  
  
init() 函数的特性如下：

* 每个源码可以使用 1 个 init() 函数。
* init() 函数会在程序执行前（main() 函数执行前）被自动调用。
* 调用顺序为 main() 中引用的包，以深度优先顺序初始化。

例如，假设有这样的包引用关系：main→A→B→C，那么这些包的 init() 函数调用顺序为：

C.init→B.init→A.init→main

说明：

* 同一个包中的多个 init() 函数的调用顺序不可预期。
* init() 函数不能被其他函数调用。

## 理解包导入后的init()函数初始化顺序

Go 语言包会从 main 包开始检查其引用的所有包，每个包也可能包含其他的包。Go 编译器由此构建出一个树状的包引用关系，再根据引用顺序决定编译顺序，依次编译这些包的代码。  
  
在运行时，被最后导入的包会最先初始化并调用 init() 函数。  
  
通过下面的代码理解包的初始化顺序。  
  
代码8-3　包导入初始化顺序入口（…/chapter08/pkginit/main.go）

1. package main
2. import "chapter08/code8-2/pkg1"
3. func main() {
4. pkg1.ExecPkg1()
5. }

代码说明如下：

* 第 3 行，导入 pkg1 包。
* 第 7 行，调用 pkg1 包的 ExecPkg1() 函数。

代码8-4　包导入初始化顺序pkg1（…/chapter08/pkginit/pkg1/pkg1.go）

1. package pkg1
2. import (
3. "chapter08/code8-2/pkg2"
4. "fmt"
5. )
6. func ExecPkg1() {
7. fmt.Println("ExecPkg1")
8. pkg2.ExecPkg2()
9. }
10. func init() {
11. fmt.Println("pkg1 init")
12. }

代码说明如下：

* 第 4 行，导入 pkg2 包。
* 第 8 行，声明 ExecPkg1() 函数。
* 第 12 行，调用 pkg2 包的 ExecPkg2() 函数。
* 第 15 行，在 pkg1 包初始化时，打印 pkg1 init。

代码8-5　包导入初始化顺序pkg2（…/chapter08/pkginit/pkg2/pkg2.go）

1. package pkg2
2. import "fmt"
3. func ExecPkg2() {
4. fmt.Println("ExecPkg2")
5. }
6. func init() {
7. fmt.Println("pkg2 init")
8. }

代码说明如下：

* 第 5 行，声明 ExecPkg2() 函数。
* 第 10 行，在 pkg2 包初始化时，打印 pkg2 init。

执行代码，输出如下：

pkg2 init  
pkg1 init  
ExecPkg1  
ExecPkg2

# Go语言工厂模式自动注册——管理多个包的结构体

< [Go语言import导入包](http://c.biancheng.net/view/91.html)[Go语言单例模式](http://c.biancheng.net/view/vip_7339.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

本例利用包的 init 特性，将 cls1 和 cls2 两个包注册到工厂，使用字符串创建这两个注册好的结构实例。  
  
完整代码的结构如下：

.  
└── src  
    └── chapter08  
        └── clsfactory  
            ├── main.go  
            └── base  
                └── factory.go  
            └── cls1  
                └── reg.go  
            └── cls2  
                └── reg.go

本套教程所有源码下载地址：[https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ](https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ" \t "http://c.biancheng.net/view/_blank)    提取密码：hfyf

类工厂（具体文件：…/chapter08/clsfactory/base/factory.go）

1. package base
2. // 类接口
3. type Class interface {
4. Do()
5. }
6. var (
7. // 保存注册好的工厂信息
8. factoryByName = make(map[string]func() Class)
9. )
10. // 注册一个类生成工厂
11. func Register(name string, factory func() Class) {
12. factoryByName[name] = factory
13. }
14. // 根据名称创建对应的类
15. func Create(name string) Class {
16. if f, ok := factoryByName[name]; ok {
17. return f()
18. } else {
19. panic("name not found")
20. }
21. }

这个包叫base，负责处理注册和使用工厂的基础代码，该包不会引用任何外部的包。  
  
以下是对代码的说明：

* 第 4 行定义了“产品”：类。
* 第 10 行使用了一个 map 保存注册的工厂信息。
* 第 14 行提供给工厂方注册使用，所谓的“工厂”，就是一个定义为func() Class的普通函数，调用此函数，创建一个类实例，实现的工厂内部结构体会实现 Class 接口。
* 第 19 行定义通过名字创建类实例的函数，该函数会在注册好后调用。
* 第 20 行在已经注册的信息中查找名字对应的工厂函数，找到后，在第 21 行调用并返回接口。
* 第 23 行是如果创建的名字没有找到时，报错。

类1及注册代码（具体文件：…/chapter08/clsfactory/cls1/reg.go）

1. package cls1
2. import (
3. "chapter08/clsfactory/base"
4. "fmt"
5. )
6. // 定义类1
7. type Class1 struct {
8. }
9. // 实现Class接口
10. func (c \*Class1) Do() {
11. fmt.Println("Class1")
12. }
13. func init() {
14. // 在启动时注册类1工厂
15. base.Register("Class1", func() base.Class {
16. return new(Class1)
17. })
18. }

上面的代码展示了Class1的工厂及产品定义过程。

* 第 9～15 行定义 Class1 结构，该结构实现了 base 中的 Class 接口。
* 第 20 行，Class1 结构的实例化过程叫 Class1 的工厂，使用 base.Register() 函数在 init() 函数被调用时与一个字符串关联，这样，方便以后通过名字重新调用该函数并创建实例。

类2及注册代码（具体文件：…/chapter08/clsfactory/cls2/reg.go）

1. package cls2
2. import (
3. "chapter08/clsfactory/base"
4. "fmt"
5. )
6. // 定义类2
7. type Class2 struct {
8. }
9. // 实现Class接口
10. func (c \*Class2) Do() {
11. fmt.Println("Class2")
12. }
13. func init() {
14. // 在启动时注册类2工厂
15. base.Register("Class2", func() base.Class {
16. return new(Class2)
17. })
18. }

Class2 的注册与 Class1 的定义和注册过程类似。  
  
类工程主流程（具体文件：…/chapter08/clsfactory/main.go）

1. package main
2. import (
3. "chapter08/clsfactory/base"
4. \_ "chapter08/clsfactory/cls1" // 匿名引用cls1包, 自动注册
5. \_ "chapter08/clsfactory/cls2" // 匿名引用cls2包, 自动注册
6. )
7. func main() {
8. // 根据字符串动态创建一个Class1实例
9. c1 := base.Create("Class1")
10. c1.Do()
11. // 根据字符串动态创建一个Class2实例
12. c2 := base.Create("Class2")
13. c2.Do()
14. }

下面是对代码的说明：

* 第 5 和第 6 行使用匿名引用方法导入了 cls1 和 cls2 两个包。在 main() 函数调用前，这两个包的 init() 函数会被自动调用，从而自动注册 Class1 和 Class2。
* 第 12 和第 16 行，通过 base.Create() 方法查找字符串对应的类注册信息，调用工厂方法进行实例创建。
* 第 13 和第 17 行，调用类的方法。

执行下面的指令进行编译：

export GOPATH=/home/davy/golangbook/code  
go install chapter08/clsfactory  
$GOPATH/bin/clsfactory

代码输出如下：

Class1  
Class2

# Go语言单例模式简述

< [Go语言工厂模式自动注册](http://c.biancheng.net/view/vip_7338.html)[Go语言sync包与锁](http://c.biancheng.net/view/4307.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

设计模式的重要性不用多说，也是面试时常常会被问到的问题。对于设计模式，更多的则是仁者见仁智者见智，要在实际工作中不断的积累，再进行深度的思考，才能逐渐形成的一种思维。  
  
单例模式也叫单子模式，是常用的模式之一，在它的核心结构中只包含一个被称为单例的特殊类，能够保证系统运行中一个类只创建一个实例，本节我们就来介绍一下Go语言中的单例模式。

## 单例模式实现

Go语言实现单例模式的有四种方式，分别是懒汉式、饿汉式、双重检查和 sync.Once。  
  
懒汉式就是创建对象时比较懒，先不急着创建对象，在需要加载配置文件的时候再去创建；饿汉式则是在系统初始化的时候就已经把对象创建好了，需要用的时候直接拿过来用就好了。  
  
不管那种模式最终目的只有一个，就是只实例化一次，只允许一个实例存在。  
  
下面就来分别介绍一下这四种实现方式：

#### 1) 懒汉式——非线程安全

非线程安全，指的是在多线程下可能会创建多次对象。

1. //使用结构体代替类
2. type Tool struct {
3. values int
4. }
5. //建立私有变量
6. var instance \*Tool
7. //获取单例对象的方法，引用传递返回
8. func GetInstance() \*Tool {
9. if instance == nil {
10. instance = new(Tool)
11. }
12. return instance
13. }

在非线程安全的基本上，利用 Sync.Mutex 进行加锁保证线程安全，但由于每次调用该方法都进行了加锁操作，在性能上不是很高效。

1. //锁对象
2. var lock sync.Mutex
3. //加锁保证线程安全
4. func GetInstance() \*Tool {
5. lock.Lock()
6. defer lock.Unlock()
7. if instance == nil {
8. instance = new(Tool)
9. }
10. return instance
11. }

#### 2) 饿汉式

直接创建好对象，不需要判断为空，同时也是线程安全，唯一的缺点是在导入包的同时会创建该对象，并持续占有在内存中。  
  
Go语言饿汉式可以使用 init 函数，也可以使用全局变量。

1. type cfg struct {
2. }
3. var cfg \*config
4. func init() {
5. cfg = new(config)
6. }
7. // NewConfig 提供获取实例的方法
8. func NewConfig() \*config {
9. return cfg
10. }
11. type config struct {
12. }
13. //全局变量
14. var cfg \*config = new(config)
15. // NewConfig 提供获取实例的方法
16. func NewConfig() \*config {
17. return cfg
18. }

#### 3) 双重检查

在懒汉式（线程安全）的基础上再进行优化，减少加锁的操作，保证线程安全的同时不影响性能。

1. //锁对象
2. var lock sync.Mutex
3. //第一次判断不加锁，第二次加锁保证线程安全，一旦对象建立后，获取对象就不用加锁了。
4. func GetInstance() \*Tool {
5. if instance == nil {
6. lock.Lock()
7. if instance == nil {
8. instance = new(Tool)
9. }
10. lock.Unlock()
11. }
12. return instance
13. }

#### 5) sync.Once

通过 sync.Once 来确保创建对象的方法只执行一次

1. var once sync.Once
2. func GetInstance() \*Tool {
3. once.Do(func() {
4. instance = new(Tool)
5. })
6. return instance
7. }

sync.Once 内部本质上也是双重检查的方式，但在写法上会比自己写双重检查更简洁，以下是 Once 的源码

[纯文本复制](http://c.biancheng.net/view/vip_7339.html)

1. func (o \*Once) Do(f func()) {
2. //判断是否执行过该方法，如果执行过则不执行
3. if atomic.LoadUint32(&o.done) == 1 {
4. return
5. }
6. // Slow-path.
7. o.m.Lock()
8. defer o.m.Unlock()
9. //进行加锁，再做一次判断，如果没有执行，则进行标志已经扫行并调用该方法
10. if o.done == 0 {
11. defer atomic.StoreUint32(&o.done, 1)
12. f()
13. }
14. }

# Go语言sync包与锁：限制线程对变量的访问

< [Go语言单例模式](http://c.biancheng.net/view/vip_7339.html)[Go语言big包](http://c.biancheng.net/view/4308.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中 sync 包里提供了互斥锁 Mutex 和读写锁 RWMutex 用于处理并发过程中可能出现同时两个或多个协程（或线程）读或写同一个变量的情况。

## 为什么需要锁

锁是 sync 包中的核心，它主要有两个方法，分别是加锁（Lock）和解锁（Unlock）。  
  
在并发的情况下，多个线程或协程同时其修改一个变量，使用锁能保证在某一时间内，只有一个协程或线程修改这一变量。  
  
不使用锁时，在并发的情况下可能无法得到想要的结果，如下所示：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. var a = 0
8. for i := 0; i < 1000; i++ {
9. go func(idx int) {
10. a += 1
11. fmt.Println(a)
12. }(i)
13. }
14. time.Sleep(time.Second)
15. }

从理论上来说，上面的程序会将 a 的值依次递增输出，然而实际结果却是下面这样子的。

537  
995  
996  
997  
538  
999  
1000

通过运行结果可以看出 a 的值并不是按顺序递增输出的，这是为什么呢？  
  
协程的执行顺序大致如下所示：

* 从寄存器读取 a 的值；
* 然后做加法运算；
* 最后写到寄存器。

按照上面的顺序，假如有一个协程取得 a 的值为 3，然后执行加法运算，此时又有一个协程对 a 进行取值，得到的值同样是 3，最终两个协程的返回结果是相同的。  
  
而锁的概念就是，当一个协程正在处理 a 时将 a 锁定，其它协程需要等待该协程处理完成并将 a 解锁后才能再进行操作，也就是说同时处理 a 的协程只能有一个，从而避免上面示例中的情况出现。

## 互斥锁 Mutex

上面的示例中出现的问题怎么解决呢？加一个互斥锁 Mutex 就可以了。那什么是互斥锁呢 ？互斥锁中其有两个方法可以调用，如下所示：

func (m \*Mutex) Lock()  
func (m \*Mutex) Unlock()

将上面的代码略作修改，如下所示：

1. package main
2. import (
3. "fmt"
4. "sync"
5. "time"
6. )
7. func main() {
8. var a = 0
9. var lock sync.Mutex
10. for i := 0; i < 1000; i++ {
11. go func(idx int) {
12. lock.Lock()
13. defer lock.Unlock()
14. a += 1
15. fmt.Printf("goroutine %d, a=%d\n", idx, a)
16. }(i)
17. }
18. // 等待 1s 结束主程序
19. // 确保所有协程执行完
20. time.Sleep(time.Second)
21. }

运行结果如下：

goroutine 995, a=996  
goroutine 996, a=997  
goroutine 997, a=998  
goroutine 998, a=999  
goroutine 999, a=1000

需要注意的是一个互斥锁只能同时被一个 goroutine 锁定，其它 goroutine 将阻塞直到互斥锁被解锁（重新争抢对互斥锁的锁定），示例代码如下：

1. package main
2. import (
3. "fmt"
4. "sync"
5. "time"
6. )
7. func main() {
8. ch := make(chan struct{}, 2)
9. var l sync.Mutex
10. go func() {
11. l.Lock()
12. defer l.Unlock()
13. fmt.Println("goroutine1: 我会锁定大概 2s")
14. time.Sleep(time.Second \* 2)
15. fmt.Println("goroutine1: 我解锁了，你们去抢吧")
16. ch <- struct{}{}
17. }()
18. go func() {
19. fmt.Println("goroutine2: 等待解锁")
20. l.Lock()
21. defer l.Unlock()
22. fmt.Println("goroutine2: 欧耶，我也解锁了")
23. ch <- struct{}{}
24. }()
25. // 等待 goroutine 执行结束
26. for i := 0; i < 2; i++ {
27. <-ch
28. }
29. }

上面的代码运行结果如下：

goroutine1: 我会锁定大概 2s  
goroutine2: 等待解锁  
goroutine1: 我解锁了，你们去抢吧  
goroutine2: 欧耶，我也解锁了

## 读写锁

读写锁有如下四个方法：

* 写操作的锁定和解锁分别是func (\*RWMutex) Lock和func (\*RWMutex) Unlock；
* 读操作的锁定和解锁分别是func (\*RWMutex) Rlock和func (\*RWMutex) RUnlock。

读写锁的区别在于：

* 当有一个 goroutine 获得写锁定，其它无论是读锁定还是写锁定都将阻塞直到写解锁；
* 当有一个 goroutine 获得读锁定，其它读锁定仍然可以继续；
* 当有一个或任意多个读锁定，写锁定将等待所有读锁定解锁之后才能够进行写锁定。

所以说这里的读锁定（RLock）目的其实是告诉写锁定，有很多协程或者进程正在读取数据，写操作需要等它们读（读解锁）完才能进行写（写锁定）。  
  
我们可以将其总结为如下三条：

* 同时只能有一个 goroutine 能够获得写锁定；
* 同时可以有任意多个 gorouinte 获得读锁定；
* 同时只能存在写锁定或读锁定（读和写互斥）。

示例代码如下所示：

1. package main
2. import (
3. "fmt"
4. "math/rand"
5. "sync"
6. )
7. var count int
8. var rw sync.RWMutex
9. func main() {
10. ch := make(chan struct{}, 10)
11. for i := 0; i < 5; i++ {
12. go read(i, ch)
13. }
14. for i := 0; i < 5; i++ {
15. go write(i, ch)
16. }
17. for i := 0; i < 10; i++ {
18. <-ch
19. }
20. }
21. func read(n int, ch chan struct{}) {
22. rw.RLock()
23. fmt.Printf("goroutine %d 进入读操作...\n", n)
24. v := count
25. fmt.Printf("goroutine %d 读取结束，值为：%d\n", n, v)
26. rw.RUnlock()
27. ch <- struct{}{}
28. }
29. func write(n int, ch chan struct{}) {
30. rw.Lock()
31. fmt.Printf("goroutine %d 进入写操作...\n", n)
32. v := rand.Intn(1000)
33. count = v
34. fmt.Printf("goroutine %d 写入结束，新值为：%d\n", n, v)
35. rw.Unlock()
36. ch <- struct{}{}
37. }

其执行结果如下：

goroutine 0 进入读操作...  
goroutine 0 读取结束，值为：0  
goroutine 3 进入读操作...  
goroutine 1 进入读操作...  
goroutine 3 读取结束，值为：0  
goroutine 1 读取结束，值为：0  
goroutine 4 进入写操作...  
goroutine 4 写入结束，新值为：81  
goroutine 4 进入读操作...  
goroutine 4 读取结束，值为：81  
goroutine 2 进入读操作...  
goroutine 2 读取结束，值为：81  
goroutine 0 进入写操作...  
goroutine 0 写入结束，新值为：887  
goroutine 1 进入写操作...  
goroutine 1 写入结束，新值为：847  
goroutine 2 进入写操作...  
goroutine 2 写入结束，新值为：59  
goroutine 3 进入写操作...  
goroutine 3 写入结束，新值为：81

下面再来看两个示例。  
  
【示例 1】多个读操作同时读取一个变量时，虽然加了锁，但是读操作是不受影响的。（读和写是互斥的，读和读不互斥）

1. package main
2. import (
3. "sync"
4. "time"
5. )
6. var m \*sync.RWMutex
7. func main() {
8. m = new(sync.RWMutex)
9. // 多个同时读
10. go read(1)
11. go read(2)
12. time.Sleep(2\*time.Second)
13. }
14. func read(i int) {
15. println(i,"read start")
16. m.RLock()
17. println(i,"reading")
18. time.Sleep(1\*time.Second)
19. m.RUnlock()
20. println(i,"read over")
21. }

运行结果如下：

1 read start  
1 reading  
2 read start  
2 reading  
1 read over  
2 read over

【示例 2】由于读写互斥，所以写操作开始的时候，读操作必须要等写操作进行完才能继续，不然读操作只能继续等待。

1. package main
2. import (
3. "sync"
4. "time"
5. )
6. var m \*sync.RWMutex
7. func main() {
8. m = new(sync.RWMutex)
9. // 写的时候啥也不能干
10. go write(1)
11. go read(2)
12. go write(3)
13. time.Sleep(2\*time.Second)
14. }
15. func read(i int) {
16. println(i,"read start")
17. m.RLock()
18. println(i,"reading")
19. time.Sleep(1\*time.Second)
20. m.RUnlock()
21. println(i,"read over")
22. }
23. func write(i int) {
24. println(i,"write start")
25. m.Lock()
26. println(i,"writing")
27. time.Sleep(1\*time.Second)
28. m.Unlock()
29. println(i,"write over")
30. }

运行结果如下：

1 write start  
3 write start  
1 writing  
2 read start  
1 write over  
2 reading

# Go语言big包：对整数的高精度计算

< [Go语言sync包与锁](http://c.biancheng.net/view/4307.html)[示例：使用图像包制作GIF动画](http://c.biancheng.net/view/vip_7340.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

实际开发中，对于超出 int64 或者 uint64 类型的大数进行计算时，如果对精度没有要求，使用 float32 或者 float64 就可以胜任，但如果对精度有严格要求的时候，我们就不能使用浮点数了，因为浮点数在内存中只能被近似的表示。  
  
Go语言中 math/big 包实现了大数字的多精度计算，支持 Int（有符号整数）、Rat（有理数）和 Float（浮点数）等数字类型。  
  
这些类型可以实现任意位数的数字，只要内存足够大，但缺点是需要更大的内存和处理开销，这使得它们使用起来要比内置的数字类型慢很多。  
  
在 math/big 包中，Int 类型定义如下所示：

1. // An Int represents a signed multi-precision integer.
2. // The zero value for an Int represents the value 0.
3. type Int struct {
4. neg bool // sign
5. abs nat // absolute value of the integer
6. }

生成 Int 类型的方法为 NewInt()，如下所示：

1. // NewInt allocates and returns a new Int set to x.
2. func NewInt(x int64) \*Int {
3. return new(Int).SetInt64(x)
4. }

注意：NewInt() 函数只对 int64 有效，其他类型必须先转成 int64 才行。

Go语言中还提供了许多 Set 函数，可以方便的把其他类型的整形存入 Int ，因此，我们可以先 new(int) 然后再调用 Set 函数，Set 函数有如下几种：

1. // SetInt64 函数将 z 转换为 x 并返回 z。
2. func (z \*Int) SetInt64(x int64) \*Int {
3. neg := false
4. if x < 0 {
5. neg = true
6. x = -x
7. }
8. z.abs = z.abs.setUint64(uint64(x))
9. z.neg = neg
10. return z
11. }
12. ​
13. // SetUint64 函数将 z 转换为 x 并返回 z。
14. func (z \*Int) SetUint64(x uint64) \*Int {
15. z.abs = z.abs.setUint64(x)
16. z.neg = false
17. return z
18. }
19. ​
20. // Set 函数将 z 转换为 x 并返回 z。
21. func (z \*Int) Set(x \*Int) \*Int {
22. if z != x {
23. z.abs = z.abs.set(x.abs)
24. z.neg = x.neg
25. }
26. return z
27. }

示例代码如下所示：

1. package main
2. import (
3. "fmt"
4. "math/big"
5. )
6. func main() {
7. big1 := new(big.Int).SetUint64(uint64(1000))
8. fmt.Println("big1 is: ", big1)
9. big2 := big1.Uint64()
10. fmt.Println("big2 is: ", big2)
11. }

运行结果如下：

big1 is:  1000  
big2 is:  1000

除了上述的 Set 函数，math/big 包中还提供了一个 SetString() 函数，可以指定进制数，比如二进制、十进制或者十六进制等！

1. // SetString sets z to the value of s, interpreted in the given base,
2. // and returns z and a boolean indicating success. The entire string
3. // (not just a prefix) must be valid for success. If SetString fails,
4. // the value of z is undefined but the returned value is nil.
5. //
6. // The base argument must be 0 or a value between 2 and MaxBase. If the base
7. // is 0, the string prefix determines the actual conversion base. A prefix of
8. // ``0x'' or ``0X'' selects base 16; the ``0'' prefix selects base 8, and a
9. // ``0b'' or ``0B'' prefix selects base 2. Otherwise the selected base is 10.
10. //
11. func (z \*Int) SetString(s string, base int) (\*Int, bool) {
12. r := strings.NewReader(s)
13. if \_, \_, err := z.scan(r, base); err != nil {
14. return nil, false
15. }
16. // entire string must have been consumed
17. if \_, err := r.ReadByte(); err != io.EOF {
18. return nil, false
19. }
20. return z, true // err == io.EOF => scan consumed all of s
21. }

示例代码如下所示：

1. package main
2. import (
3. "fmt"
4. "math/big"
5. )
6. func main() {
7. big1, \_ := new(big.Int).SetString("1000", 10)
8. fmt.Println("big1 is: ", big1)
9. big2 := big1.Uint64()
10. fmt.Println("big2 is: ", big2)
11. }

运行结果如下：

big1 is:  1000  
big2 is:  1000

因为Go语言不支持运算符重载，所以所有大数字类型都有像是 Add() 和 Mul() 这样的方法。  
  
Add 方法的定义如下所示：

func (z \*Int) Add(x, y \*Int) \*Int

该方法会将 z 转换为 x + y 并返回 z。  
  
【示例】计算第 1000 位的斐波那契数列。

1. package main
2. import (
3. "fmt"
4. "math/big"
5. "time"
6. )
7. const LIM = 1000 //求第1000位的斐波那契数列
8. var fibs [LIM]\*big.Int //使用数组保存计算出来的数列的指针
9. func main() {
10. result := big.NewInt(0)
11. start := time.Now()
12. for i := 0; i < LIM; i++ {
13. result = fibonacci(i)
14. fmt.Printf("数列第 %d 位: %d\n", i+1, result)
15. }
16. end := time.Now()
17. delta := end.Sub(start)
18. fmt.Printf("执行完成，所耗时间为: %s\n", delta)
19. }
20. func fibonacci(n int) (res \*big.Int) {
21. if n <= 1 {
22. res = big.NewInt(1)
23. } else {
24. temp := new(big.Int)
25. res = temp.Add(fibs[n-1], fibs[n-2])
26. }
27. fibs[n] = res
28. return
29. }

运行结果如下：

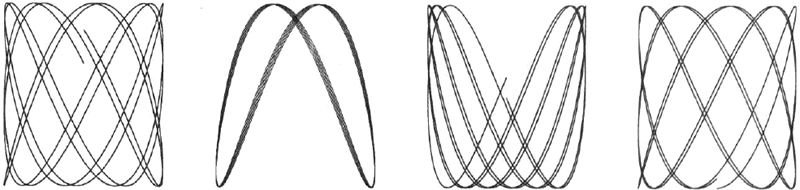
数列第 1 位: 1  
数列第 2 位: 1  
数列第 3 位: 2  
数列第 4 位: 3  
数列第 5 位: 5  
...  
数列第 997 位: 10261062362033262336604926729245222132668558120602124277764622905699407982546711488272859468887457959  
08773311924256407785074365766118082732679853917775891982813511440749936979646564952426675539110499009  
9120377  
数列第 998 位: 16602747662452097049541800472897701834948051198384828062358553091918573717701170201065510185595898605  
10409473691887927846223301598102952299783631123261876053919903676539979992673143323971886037334508837  
5054249  
数列第 999 位: 26863810024485359386146727202142923967616609318986952340123175997617981700247881689338369654483356564  
19182785616144335631297667364221035032463485041037768036733415117289916972319708276398561576445007847  
4174626  
数列第 1000 位: 4346655768693745643568852767504062580256466051737178040248172908953655541794905189040387984007925516  
92959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668  
49228875  
执行完成，所耗时间为: 6.945ms

# Go语言使用图像包制作GIF动画

< [Go语言big包](http://c.biancheng.net/view/4308.html)[Go语言正则表达式：regexp包](http://c.biancheng.net/view/5124.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

本节将通过示例来介绍一下Go语言标准库中图像包的使用，并使用这些图像包来创建一系列的位图图像，然后将位图序列编码为 GIF 动画。  
  
下面的图像叫作利萨茹图形，它类似于 20 世纪 60 年代科幻片中出现的一种纤维状视觉效果，利萨茹图形是由在互相垂直的方向上的两个频率成简单整数比的简谐振动所合成的规则的、稳定的曲线。

  
图：四种利萨茹图形

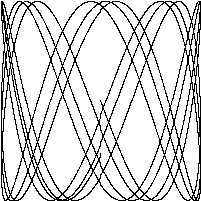
上图中所示的图像看似复杂，利用Go语言标准库中的图像包可以很轻松的实现，代码如下所示：

1. package main
2. import (
3. "image"
4. "image/color"
5. "image/gif"
6. "io"
7. "math"
8. "math/rand"
9. "os"
10. "time"
11. )
12. var palette = []color.Color{color.White, color.Black} //调色板
13. const (
14. whiteIndex = 0
15. blackIndex = 1
16. )
17. func main() {
18. rand.Seed(time.Now().UTC().UnixNano())
19. lissajous(os.Stdout)
20. }
21. func lissajous(out io.Writer) {
22. const (
23. cycles  = 5     //完整的x振荡器变化的个数
24. res     = 0.001 //角度分辨率
25. size    = 100   //图像画布包含[-size. .+size]
26. nframes = 64    //动画中的帧数
27. delay   = 8     //以10ms为单位的帧间延迟
28. )
29. freq := rand.Float64() \* 3.0 //y 振荡器的相对频率
30. anim := gif.GIF{LoopCount: nframes}
31. phase := 0.0
32. for i := 0; i < nframes; i++ {
33. rect := image.Rect(0, 0, 2\*size+1, 2\*size+1)
34. img := image.NewPaletted(rect, palette)
35. for t := 0.0; t < cycles\*2\*math.Pi; t += res {
36. x := math.Sin(t)
37. y := math.Sin(t\*freq + phase)
38. img.SetColorIndex(size+int(x\*size+0.5), size+int(y\*size+0.5), blackIndex)
39. }
40. phase += 0.1
41. anim.Delay = append(anim.Delay, delay)
42. anim.Image = append(anim.Image, img)
43. }
44. gif.EncodeAll(out, &anim)
45. }

其中，lissajous 函数内部包含两层嵌套的 for 循环，外层循环会循环 64 次，每一次都会生成一个单独的动画帧，即生成一个只有黑色和白色，且大小为 201\*201 的图片。  
  
而 lissajous 函数的内层循环则设置了两个偏振值 x 和 y，x 轴偏振使用 sin 函数，y 轴偏振也是正弦波，但其相对 x 轴的偏振是一个 0~3 的随机值。初始偏振值是一个零值，随着动画的每一帧逐渐增加，循环会一直跑到 x 轴完成五次完整的循环，每一步它都会调用 SetColorIndex 来把 (x, y) 点来染成黑色。  
  
想要成功的生成 gif 图像需要我们先使用go build 命令将代码编译为可执行文件，然后在 CMD（命令行工具）中运行所生成的可执行文件，如下所示：

go build main.go  
main.exe >1.gif

其中，>在这里用来指定生成图像的文件名，1.gif 为文件名。  
  
执行成功后就可以在当前目录下找到生成的 gif 文件了，如下图所示：

  
图：利萨如图形

# Go语言正则表达式：regexp包

< [示例：使用图像包制作GIF动画](http://c.biancheng.net/view/vip_7340.html)[Go语言time包：时间和日期](http://c.biancheng.net/view/5392.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

正则表达式是一种进行模式匹配和文本操纵的复杂而又强大的工具。虽然正则表达式比纯粹的文本匹配效率低，但是它却更灵活，按照它的语法规则，根据需求构造出的正则表达式能够从原始文本中筛选出几乎任何你想要得到的字符组合。  
  
Go语言通过 regexp 包为正则表达式提供了官方支持，其采用 RE2 语法，除了\c、\C外，Go语言和 Perl、[Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank) 等语言的正则基本一致。

## 正则表达式语法规则

正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为"元字符"）构成的文字序列，可以是单个的字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。  
  
下面的表格中列举了构成正则表达式的一些语法规则及其含义。

#### 1) 字符

|  |  |  |  |
| --- | --- | --- | --- |
| **语法** | **说明** | **表达式示例** | **匹配结果** |
| 一般字符 | 匹配自身 | abc | abc |
| . | 匹配任意除换行符"\n"外的字符， 在 DOTALL 模式中也能匹配换行符 | a.c | abc |
| \ | 转义字符，使后一个字符改变原来的意思； 如果字符串中有字符 \* 需要匹配，可以使用 \\* 或者字符集［\*]。 | a\.c a\\c | a.c a\c |
| [...] | 字符集（字符类），对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如 [abc] 或 [a-c]， 第一个字符如果是 ^ 则表示取反，如 [^abc] 表示除了abc之外的其他字符。 | a[bcd]e | abe 或 ace 或 ade |
| \d | 数字：[0-9] | a\dc | a1c |
| \D | 非数字：[^\d] | a\Dc | abc |
| \s | 空白字符：[<空格>\t\r\n\f\v] | a\sc | a c |
| \S | 非空白字符：[^\s] | a\Sc | abc |
| \w | 单词字符：[A-Za-z0-9] | a\wc | abc |
| \W | 非单词字符：[^\w] | a\Wc | a c |

#### 2) 数量词（用在字符或 (...) 之后）

|  |  |  |  |
| --- | --- | --- | --- |
| **语法** | **说明** | **表达式示例** | **匹配结果** |
| \* | 匹配前一个字符 0 或无限次 | abc\* | ab 或 abccc |
| + | 匹配前一个字符 1 次或无限次 | abc+ | abc 或 abccc |
| ? | 匹配前一个字符 0 次或 1 次 | abc? | ab 或 abc |
| {m} | 匹配前一个字符 m 次 | ab{2}c | abbc |
| {m,n} | 匹配前一个字符 m 至 n 次，m 和 n 可以省略，若省略 m，则匹配 0 至 n 次； 若省略 n，则匹配 m 至无限次 | ab{1,2}c | abc 或 abbc |

#### 3) 边界匹配

|  |  |  |  |
| --- | --- | --- | --- |
| **语法** | **说明** | **表达式示例** | **匹配结果** |
| ^ | 匹配字符串开头，在多行模式中匹配每一行的开头 | ^abc | abc |
| $ | 匹配字符串末尾，在多行模式中匹配每一行的末尾 | abc$ | abc |
| \A | 仅匹配字符串开头 | \Aabc | abc |
| \Z | 仅匹配字符串末尾 | abc\Z | abc |
| \b | 匹配 \w 和 \W 之间 | a\b!bc | a!bc |
| \B | [^\b] | a\Bbc | abc |

#### 4) 逻辑、分组

|  |  |  |  |
| --- | --- | --- | --- |
| **语法** | **说明** | **表达式示例** | **匹配结果** |
| | | | 代表左右表达式任意匹配一个，优先匹配左边的表达式 | abc|def | abc 或 def |
| (...) | 括起来的表达式将作为分组，分组将作为一个整体，可以后接数量词 | (abc){2} | abcabc |
| (?P<name>...) | 分组，功能与 (...) 相同，但会指定一个额外的别名 | (?P<id>abc){2} | abcabc |
| \<number> | 引用编号为 <number> 的分组匹配到的字符串 | (\d)abc\1 | 1abe1 或 5abc5 |
| (?P=name) | 引用别名为 <name> 的分组匹配到的字符串 | (?P<id>\d)abc(?P=id) | 1abe1 或 5abc5 |

#### 5) 特殊构造（不作为分组）

|  |  |  |  |
| --- | --- | --- | --- |
| **语法** | **说明** | **表达式示例** | **匹配结果** |
| (?:...) | (…) 的不分组版本，用于使用 "|" 或后接数量词 | (?:abc){2} | abcabc |
| (?iLmsux) | iLmsux 中的每个字符代表一种匹配模式，只能用在正则表达式的开头，可选多个 | (?i)abc | AbC |
| (?#...) | # 后的内容将作为注释被忽略。 | abc(?#comment)123 | abc123 |
| (?=...) | 之后的字符串内容需要匹配表达式才能成功匹配 | a(?=\d) | 后面是数字的 a |
| (?!...) | 之后的字符串内容需要不匹配表达式才能成功匹配 | a(?!\d) | 后面不是数字的 a |
| (?<=...) | 之前的字符串内容需要匹配表达式才能成功匹配 | (?<=\d)a | 前面是数字的a |
| (?<!...) | 之前的字符串内容需要不匹配表达式才能成功匹配 | (?<!\d)a | 前面不是数字的a |

## Regexp 包的使用

下面通过几个示例来演示一下 regexp 包的使用。  
  
【示例 1】匹配指定类型的字符串。

1. package main
2. import (
3. "fmt"
4. "regexp"
5. )
6. func main() {
7. buf := "abc azc a7c aac 888 a9c tac"
8. //解析正则表达式，如果成功返回解释器
9. reg1 := regexp.MustCompile(`a.c`)
10. if reg1 == nil {
11. fmt.Println("regexp err")
12. return
13. }
14. //根据规则提取关键信息
15. result1 := reg1.FindAllStringSubmatch(buf, -1)
16. fmt.Println("result1 = ", result1)
17. }

运行结果如下：

result1 =  [[abc] [azc] [a7c] [aac] [a9c]]

【示例 2】匹配 a 和 c 中间包含一个数字的字符串。

1. package main
2. import (
3. "fmt"
4. "regexp"
5. )
6. func main() {
7. buf := "abc azc a7c aac 888 a9c tac"
8. //解析正则表达式，如果成功返回解释器
9. reg1 := regexp.MustCompile(`a[0-9]c`)
10. if reg1 == nil { //解释失败，返回nil
11. fmt.Println("regexp err")
12. return
13. }
14. //根据规则提取关键信息
15. result1 := reg1.FindAllStringSubmatch(buf, -1)
16. fmt.Println("result1 = ", result1)
17. }

运行结果如下：

result1 =  [[a7c] [a9c]]

【示例 3】使用 \d 来匹配 a 和 c 中间包含一个数字的字符串。

1. package main
2. import (
3. "fmt"
4. "regexp"
5. )
6. func main() {
7. buf := "abc azc a7c aac 888 a9c tac"
8. //解析正则表达式，如果成功返回解释器
9. reg1 := regexp.MustCompile(`a\dc`)
10. if reg1 == nil { //解释失败，返回nil
11. fmt.Println("regexp err")
12. return
13. }
14. //根据规则提取关键信息
15. result1 := reg1.FindAllStringSubmatch(buf, -1)
16. fmt.Println("result1 = ", result1)
17. }

运行结果如下：

result1 =  [[a7c] [a9c]]

【示例 4】匹配字符串中的小数。

1. package main
2. import (
3. "fmt"
4. "regexp"
5. )
6. func main() {
7. buf := "43.14 567 agsdg 1.23 7. 8.9 1sdljgl 6.66 7.8 "
8. //解释正则表达式
9. reg := regexp.MustCompile(`\d+\.\d+`)
10. if reg == nil {
11. fmt.Println("MustCompile err")
12. return
13. }
14. //提取关键信息
15. //result := reg.FindAllString(buf, -1)
16. result := reg.FindAllStringSubmatch(buf, -1)
17. fmt.Println("result = ", result)
18. }

运行结果如下：

result =  [[43.14] [1.23] [8.9] [6.66] [7.8]]

【示例 5】匹配 div 标签中的内容。

1. package main
2. import (
3. "fmt"
4. "regexp"
5. )
6. func main() {
7. // 原生字符串
8. buf := `
10. <!DOCTYPE html>
11. <html lang="zh-CN">
12. <head>
13. <title>C语言中文网 | Go语言入门教程</title>
14. </head>
15. <body>
16. <div>Go语言简介</div>
17. <div>Go语言基本语法
18. Go语言变量的声明
19. Go语言教程简明版
20. </div>
21. <div>Go语言容器</div>
22. <div>Go语言函数</div>
23. </body>
24. </html>
25. `
26. //解释正则表达式
27. reg := regexp.MustCompile(`<div>(?s:(.\*?))</div>`)
28. if reg == nil {
29. fmt.Println("MustCompile err")
30. return
31. }
32. //提取关键信息
33. result := reg.FindAllStringSubmatch(buf, -1)
34. //过滤<></>
35. for \_, text := range result {
36. fmt.Println("text[1] = ", text[1])
37. }
38. }

运行结果如下：

text[1] =  Go语言简介  
text[1] =  Go语言基本语法  
    Go语言变量的声明  
    Go语言教程简明版  
     
text[1] =  Go语言容器  
text[1] =  Go语言函数

【示例 6】通过 Compile 方法返回一个 Regexp 对象，实现匹配，查找，替换相关的功能。

1. package main
2. import (
3. "fmt"
4. "regexp"
5. "strconv"
6. )
7. func main() {
8. //目标字符串
9. searchIn := "John: 2578.34 William: 4567.23 Steve: 5632.18"
10. pat := "[0-9]+.[0-9]+" //正则
11. f := func(s string) string{
12. v, \_ := strconv.ParseFloat(s, 32)
13. return strconv.FormatFloat(v \* 2, 'f', 2, 32)
14. }
15. if ok, \_ := regexp.Match(pat, []byte(searchIn)); ok {
16. fmt.Println("Match Found!")
17. }
18. re, \_ := regexp.Compile(pat)
19. //将匹配到的部分替换为 "##.#"
20. str := re.ReplaceAllString(searchIn, "##.#")
21. fmt.Println(str)
22. //参数为函数时
23. str2 := re.ReplaceAllStringFunc(searchIn, f)
24. fmt.Println(str2)
25. }

输出结果：

Match Found!  
John: ##.# William: ##.# Steve: ##.#  
John: 5156.68 William: 9134.46 Steve: 11264.36

上面代码中 Compile 方法可以解析并返回一个正则表达式，如果成功返回，则说明该正则表达式正确可用于匹配文本。  
  
另外我们也可以使用 MustCompile 方法，它也可以像 Compile 方法一样检验正则的有效性，但是当正则不合法时程序将 panic。

# Go语言time包：时间和日期

< [Go语言正则表达式：regexp包](http://c.biancheng.net/view/5124.html)[Go语言os包用法简述](http://c.biancheng.net/view/5572.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

时间和日期是我们开发中经常会用到的，Go语言中的 time 包提供了时间显示和测量等所用的函数，本节我们就来介绍一下 time 包的基本用法。

## time 包简介

时间一般包含时间值和时区，可以从Go语言中 time 包的源码中看出：

1. type Time struct {
2. // wall and ext encode the wall time seconds, wall time nanoseconds,
3. // and optional monotonic clock reading in nanoseconds.
4. //
5. // From high to low bit position, wall encodes a 1-bit flag (hasMonotonic),
6. // a 33-bit seconds field, and a 30-bit wall time nanoseconds field.
7. // The nanoseconds field is in the range [0, 999999999].
8. // If the hasMonotonic bit is 0, then the 33-bit field must be zero
9. // and the full signed 64-bit wall seconds since Jan 1 year 1 is stored in ext.
10. // If the hasMonotonic bit is 1, then the 33-bit field holds a 33-bit
11. // unsigned wall seconds since Jan 1 year 1885, and ext holds a
12. // signed 64-bit monotonic clock reading, nanoseconds since process start.
13. wall uint64
14. ext int64
15. // loc specifies the Location that should be used to
16. // determine the minute, hour, month, day, and year
17. // that correspond to this Time.
18. // The nil location means UTC.
19. // All UTC times are represented with loc==nil, never loc==&utcLoc.
20. loc \*Location
21. }

上面代码中：

* wall：表示距离公元 1 年 1 月 1 日 00:00:00UTC 的秒数；
* ext：表示纳秒；
* loc：代表时区，主要处理偏移量，不同的时区，对应的时间不一样。

#### 如何正确表示时间呢？

公认最准确的计算应该是使用“原子震荡周期”所计算的物理时钟了（Atomic Clock, 也被称为原子钟），这也被定义为标准时间（International Atomic Time）。  
  
而我们常常看见的 UTC（Universal Time Coordinated，世界协调时间）就是利用这种 Atomic Clock 为基准所定义出来的正确时间。UTC 标准时间是以 GMT（Greenwich Mean Time，格林尼治时间）这个时区为主，所以本地时间与 UTC 时间的时差就是本地时间与 GMT 时间的时差。

UTC + 时区差 ＝ 本地时间

国内一般使用的是北京时间，与 UTC 的时间关系如下：

UTC + 8 个小时 = 北京时间

在Go语言的 time 包里面有两个时区变量，如下：

* time.UTC：UTC 时间
* time.Local：本地时间

同时，Go语言还提供了 LoadLocation 方法和 FixedZone 方法来获取时区变量，如下：

FixedZone(name string, offset int) \*Location

其中，name 为时区名称，offset 是与 UTC 之前的时差。

LoadLocation(name string) (\*Location, error)

其中，name 为时区的名字。

## 时间的获取

#### 1) 获取当前时间

我们可以通过 time.Now() 函数来获取当前的时间对象，然后通过事件对象来获取当前的时间信息。示例代码如下：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. now := time.Now() //获取当前时间
8. fmt.Printf("current time:%v\n", now)
9. year := now.Year() //年
10. month := now.Month() //月
11. day := now.Day() //日
12. hour := now.Hour() //小时
13. minute := now.Minute() //分钟
14. second := now.Second() //秒
15. fmt.Printf("%d-%02d-%02d %02d:%02d:%02d\n", year, month, day, hour, minute, second)
16. }

运行结果如下：

current time:2019-12-12 12:33:19.4712277 +0800 CST m=+0.006980401  
2019-12-12 12:33:19

#### 2) 获取时间戳

时间戳是自 1970 年 1 月 1 日（08:00:00GMT）至当前时间的总毫秒数，它也被称为 Unix 时间戳（UnixTimestamp）。  
  
基于时间对象获取时间戳的示例代码如下：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. now := time.Now() //获取当前时间
8. timestamp1 := now.Unix() //时间戳
9. timestamp2 := now.UnixNano() //纳秒时间戳
10. fmt.Printf("现在的时间戳：%v\n", timestamp1)
11. fmt.Printf("现在的纳秒时间戳：%v\n", timestamp2)
12. }

运行结果如下：

现在的时间戳：1576127858  
现在的纳秒时间戳：1576127858829900100

使用 time.Unix() 函数可以将时间戳转为时间格式，示例代码如下：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. now := time.Now() //获取当前时间
8. timestamp := now.Unix() //时间戳
9. timeObj := time.Unix(timestamp, 0) //将时间戳转为时间格式
10. fmt.Println(timeObj)
11. year := timeObj.Year() //年
12. month := timeObj.Month() //月
13. day := timeObj.Day() //日
14. hour := timeObj.Hour() //小时
15. minute := timeObj.Minute() //分钟
16. second := timeObj.Second() //秒
17. fmt.Printf("%d-%02d-%02d %02d:%02d:%02d\n", year, month, day, hour, minute, second)
18. }

运行结果如下：

2019-12-12 13:24:09 +0800 CST  
2019-12-12 13:24:09

#### 3) 获取当前是星期几

time 包中的 Weekday 函数能够返回某个时间点所对应是一周中的周几，示例代码如下：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. //时间戳
8. t := time.Now()
9. fmt.Println(t.Weekday().String())
10. }

运行结果如下：

Thursday

## 时间操作函数

#### 1) Add

我们在日常的开发过程中可能会遇到要求某个时间 + 时间间隔之类的需求，Go语言中的 Add 方法如下：

func (t Time) Add(d Duration) Time

Add 函数可以返回时间点 t + 时间间隔 d 的值。  
  
【示例】求一个小时之后的时间：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. now := time.Now()
8. later := now.Add(time.Hour) // 当前时间加1小时后的时间
9. fmt.Println(later)
10. }

运行结果如下：

2019-12-12 16:00:29.9866943 +0800 CST m=+3600.007978201

#### 2) Sub

求两个时间之间的差值：

func (t Time) Sub(u Time) Duration

返回一个时间段 t - u 的值。如果结果超出了 Duration 可以表示的最大值或最小值，将返回最大值或最小值，要获取时间点 t - d（d 为 Duration），可以使用 t.Add(-d)。

#### 3) Equal

判断两个时间是否相同：

func (t Time) Equal(u Time) bool

Equal 函数会考虑时区的影响，因此不同时区标准的时间也可以正确比较，Equal 方法和用 t==u 不同，Equal 方法还会比较地点和时区信息。

#### 4) Before

判断一个时间点是否在另一个时间点之前：

func (t Time) Before(u Time) bool

如果 t 代表的时间点在 u 之前，则返回真，否则返回假。

#### 5) After

判断一个时间点是否在另一个时间点之后：

func (t Time) After(u Time) bool

如果 t 代表的时间点在 u 之后，则返回真，否则返回假。

## 定时器

使用 time.Tick(时间间隔) 可以设置定时器，定时器的本质上是一个通道（channel），示例代码如下：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. ticker := time.Tick(time.Second) //定义一个1秒间隔的定时器
8. for i := range ticker {
9. fmt.Println(i) //每秒都会执行的任务
10. }
11. }

运行结果如下：

2019-12-12 15:14:26.4158067 +0800 CST m=+16.007460701  
2019-12-12 15:14:27.4159467 +0800 CST m=+17.007600701  
2019-12-12 15:14:28.4144689 +0800 CST m=+18.006122901  
2019-12-12 15:14:29.4159581 +0800 CST m=+19.007612101  
2019-12-12 15:14:30.4144337 +0800 CST m=+20.006087701  
...

## 时间格式化

时间类型有一个自带的 Format 方法进行格式化，需要注意的是Go语言中格式化时间模板不是常见的Y-m-d H:M:S 而是使用Go语言的诞生时间 2006 年 1 月 2 号 15 点 04 分 05 秒。

提示：如果想将时间格式化为 12 小时格式，需指定 PM。

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. now := time.Now()
8. // 格式化的模板为Go的出生时间2006年1月2号15点04分 Mon Jan
9. // 24小时制
10. fmt.Println(now.Format("2006-01-02 15:04:05.000 Mon Jan"))
11. // 12小时制
12. fmt.Println(now.Format("2006-01-02 03:04:05.000 PM Mon Jan"))
13. fmt.Println(now.Format("2006/01/02 15:04"))
14. fmt.Println(now.Format("15:04 2006/01/02"))
15. fmt.Println(now.Format("2006/01/02"))
16. }

运行结果如下：

2019-12-12 15:20:52.037 Thu Dec  
2019-12-12 03:20:52.037 PM Thu Dec  
2019/12/12 15:20  
15:20 2019/12/12  
2019/12/12

## 解析字符串格式的时间

Parse 函数可以解析一个格式化的时间字符串并返回它代表的时间。

func Parse(layout, value string) (Time, error)

与 Parse 函数类似的还有 ParseInLocation 函数。

func ParseInLocation(layout, value string, loc \*Location) (Time, error)

ParseInLocation 与 Parse 函数类似，但有两个重要的不同之处：

* 第一，当缺少时区信息时，Parse 将时间解释为 UTC 时间，而 ParseInLocation 将返回值的 Location 设置为 loc；
* 第二，当时间字符串提供了时区偏移量信息时，Parse 会尝试去匹配本地时区，而 ParseInLocation 会去匹配 loc。

示例代码如下：

1. package main
2. import (
3. "fmt"
4. "time"
5. )
6. func main() {
7. var layout string = "2006-01-02 15:04:05"
8. var timeStr string = "2019-12-12 15:22:12"
9. timeObj1, \_ := time.Parse(layout, timeStr)
10. fmt.Println(timeObj1)
11. timeObj2, \_ := time.ParseInLocation(layout, timeStr, time.Local)
12. fmt.Println(timeObj2)
13. }

运行结果如下：

2019-12-12 15:22:12 +0000 UTC  
2019-12-12 15:22:12 +0800 CST

# Go语言os包用法简述

< [Go语言time包：时间和日期](http://c.biancheng.net/view/5392.html)[Go语言flag包：命令行参数解析](http://c.biancheng.net/view/5573.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言的 os 包中提供了操作系统函数的接口，是一个比较重要的包。顾名思义，os 包的作用主要是在服务器上进行系统的基本操作，如文件操作、目录操作、执行命令、信号与中断、进程、系统状态等等。

## os 包中的常用函数

#### 1) Hostname

函数定义:

func Hostname() (name string, err error)

Hostname 函数会返回内核提供的主机名。

#### 2) Environ

函数定义:

func Environ() []string

Environ 函数会返回所有的环境变量，返回值格式为“key=value”的字符串的切片拷贝。

#### 3) Getenv

函数定义:

func Getenv(key string) string

Getenv 函数会检索并返回名为 key 的环境变量的值。如果不存在该环境变量则会返回空字符串。

#### 4) Setenv

函数定义:

func Setenv(key, value string) error

Setenv 函数可以设置名为 key 的环境变量，如果出错会返回该错误。

#### 5) Exit

函数定义:

func Exit(code int)

Exit 函数可以让当前程序以给出的状态码 code 退出。一般来说，状态码 0 表示成功，非 0 表示出错。程序会立刻终止，并且 defer 的函数不会被执行。

#### 6) Getuid

函数定义:

func Getuid() int

Getuid 函数可以返回调用者的用户 ID。

#### 7) Getgid

函数定义:

func Getgid() int

Getgid 函数可以返回调用者的组 ID。

#### 8) Getpid

函数定义:

func Getpid() int

Getpid 函数可以返回调用者所在进程的进程 ID。

#### 9) Getwd

函数定义:

func Getwd() (dir string, err error)

Getwd 函数可以返回一个对应当前工作目录的根路径。如果当前目录可以经过多条路径抵达（因为硬链接），Getwd 会返回其中一个。

#### 10) Mkdir

函数定义:

func Mkdir(name string, perm FileMode) error

Mkdir 函数可以使用指定的权限和名称创建一个目录。如果出错，会返回 \*PathError 底层类型的错误。

#### 11) MkdirAll

函数定义:

func MkdirAll(path string, perm FileMode) error

MkdirAll 函数可以使用指定的权限和名称创建一个目录，包括任何必要的上级目录，并返回 nil，否则返回错误。权限位 perm 会应用在每一个被该函数创建的目录上。如果 path 指定了一个已经存在的目录，MkdirAll 不做任何操作并返回 nil。

#### 12) Remove

函数定义:

func Remove(name string) error

Remove 函数会删除 name 指定的文件或目录。如果出错，会返回 \*PathError 底层类型的错误。  
  
RemoveAll 函数跟 Remove 用法一样，区别是会递归的删除所有子目录和文件。  
  
在 os 包下，有 exec，signal，user 三个子包，下面来分别介绍一下。

## os/exec 执行外部命令

exec 包可以执行外部命令，它包装了 os.StartProcess 函数以便更容易的修正输入和输出，使用管道连接 I/O，以及作其它的一些调整。

func LookPath(file string) (string, error)

在环境变量 PATH 指定的目录中搜索可执行文件，如果 file 中有斜杠，则只在当前目录搜索。返回完整路径或者相对于当前目录的一个相对路径。  
  
示例代码如下：

1. package main
2. import (
3. "fmt"
4. "os/exec"
5. )
6. func main() {
7. f, err := exec.LookPath("main")
8. if err != nil {
9. fmt.Println(err)
10. }
11. fmt.Println(f)
12. }

运行结果如下：

main.exe

## os/user 获取当前用户信息

可以通过 os/user 包中的 Current() 函数来获取当前用户信息，该函数会返回一个 User 结构体，结构体中的 Username、Uid、HomeDir、Gid 分别表示当前用户的名称、用户 id、用户主目录和用户所属组 id，函数原型如下：

func Current() (\*User, error)

示例代码如下：

1. package main
2. import (
3. "log"
4. "os/user"
5. )
6. func main() {
7. u, \_ := user.Current()
8. log.Println("用户名：", u.Username)
9. log.Println("用户id", u.Uid)
10. log.Println("用户主目录：", u.HomeDir)
11. log.Println("主组id：", u.Gid)
12. // 用户所在的所有的组的id
13. s, \_ := u.GroupIds()
14. log.Println("用户所在的所有组：", s)
15. }

运行结果如下：

2019/12/13 15:12:14 用户名： LENOVO-PC\Administrator  
2019/12/13 15:12:14 用户id S-1-5-21-711400000-2334436127-1750000211-000  
2019/12/13 15:12:14 用户主目录： C:\Users\Administrator  
2019/12/13 15:12:14 主组id： S-1-5-22-766000000-2300000100-1050000262-000  
2019/12/13 15:12:14 用户所在的所有组： [S-1-5-32-544 S-1-5-22-000 S-1-5-21-777400999-2344436111-1750000262-003]

## os/signal 信号处理

一个运行良好的程序在退出（正常退出或者强制退出，如 Ctrl+C，kill 等）时是可以执行一段清理代码的，将收尾工作做完后再真正退出。一般采用系统 Signal 来通知系统退出，如 kill pid，在程序中针对一些系统信号设置了处理函数，当收到信号后，会执行相关清理程序或通知各个子进程做自清理。  
  
Go语言中对信号的处理主要使用 os/signal 包中的两个方法，一个是 Notify 方法用来监听收到的信号，一个是 stop 方法用来取消监听。

func Notify(c chan<- os.Signal, sig ...os.Signal)

其中，第一个参数表示接收信号的 channel，第二个及后面的参数表示设置要监听的信号，如果不设置表示监听所有的信号。  
  
【示例 1】使用 Notify 方法来监听收到的信号：

1. package main
2. import (
3. "fmt"
4. "os"
5. "os/signal"
6. )
7. func main() {
8. c := make(chan os.Signal, 0)
9. signal.Notify(c)
10. // Block until a signal is received.
11. s := <-c
12. fmt.Println("Got signal:", s)
13. }

运行该程序，然后在 CMD 窗口中通过 Ctrl+C 来结束该程序，便会得到输出结果：

Got signal: interrupt

【示例 2】使用 stop 方法来取消监听：

1. package main
2. import (
3. "fmt"
4. "os"
5. "os/signal"
6. )
7. func main() {
8. c := make(chan os.Signal, 0)
9. signal.Notify(c)
10. signal.Stop(c) //不允许继续往c中存入内容
11. s := <-c //c无内容，此处阻塞，所以不会执行下面的语句，也就没有输出
12. fmt.Println("Got signal:", s)
13. }

因为使用 Stop 方法取消了 Notify 方法的监听，所以运行程序没有输出结果。

# Go语言flag包：命令行参数解析

< [Go语言os包用法简述](http://c.biancheng.net/view/5572.html)[Go语言go mod包依赖管理工具](http://c.biancheng.net/view/5712.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在编写命令行程序（工具、server）时，我们有时需要对命令参数进行解析，各种编程语言一般都会提供解析命令行参数的方法或库，以方便程序员使用。在Go语言中的 flag 包中，提供了命令行参数解析的功能。  
  
下面我们就来看一下 flag 包可以做什么，它具有什么样的能力。  
  
这里介绍几个概念：

* 命令行参数（或参数）：是指运行程序时提供的参数；
* 已定义命令行参数：是指程序中通过 flag.Type 这种形式定义了的参数；
* 非 flag（non-flag）命令行参数（或保留的命令行参数）：可以简单理解为 flag 包不能解析的参数。

## flag 包概述

Go语言内置的 flag 包实现了命令行参数的解析，flag 包使得开发命令行工具更为简单。若要使用 flag 包，首先需要使用 import 关键字导入 flag 包，如下所示：

import "flag"

## flag 参数类型

flag 包支持的命令行参数类型有 bool、int、int64、uint、uint64、float、float64、string、duration，如下表所示：

|  |  |
| --- | --- |
| **flag 参数** | **有效值** |
| 字符串 flag | 合法字符串 |
| 整数 flag | 1234、0664、0x1234 等类型，也可以是负数 |
| 浮点数 flag | 合法浮点数 |
| bool 类型 flag | 1、0、t、f、T、F、true、false、TRUE、FALSE、True、False |
| 时间段 flag | 任何合法的时间段字符串，如“300ms”、“-1.5h”、“2h45m”， 合法的单位有“ns”、“us”、“µs”、“ms”、“s”、“m”、“h” |

## flag 包基本使用

有以下两种常用的定义命令行 flag 参数的方法：

#### 1) flag.Type()

基本格式如下：

flag.Type(flag 名, 默认值, 帮助信息) \*Type

Type 可以是 Int、String、Bool 等，返回值为一个相应类型的指针，例如我们要定义姓名、年龄、婚否三个命令行参数，我们可以按如下方式定义：

1. name := flag.String("name", "张三", "姓名")
2. age := flag.Int("age", 18, "年龄")
3. married := flag.Bool("married", false, "婚否")
4. delay := flag.Duration("d", 0, "时间间隔")

需要注意的是，此时 name、age、married、delay 均为对应类型的指针。

#### 2) flag.TypeVar()

基本格式如下：

flag.TypeVar(Type 指针, flag 名, 默认值, 帮助信息)

TypeVar 可以是 IntVar、StringVar、BoolVar 等，其功能为将 flag 绑定到一个变量上，例如我们要定义姓名、年龄、婚否三个命令行参数，我们可以按如下方式定义：

1. var name string
2. var age int
3. var married bool
4. var delay time.Duration
5. flag.StringVar(&name, "name", "张三", "姓名")
6. flag.IntVar(&age, "age", 18, "年龄")
7. flag.BoolVar(&married, "married", false, "婚否")
8. flag.DurationVar(&delay, "d", 0, "时间间隔")

#### flag.Parse()

通过以上两种方法定义好命令行 flag 参数后，需要通过调用 flag.Parse() 来对命令行参数进行解析。  
  
支持的命令行参数格式有以下几种：

* -flag：只支持 bool 类型；
* -flag=x；
* -flag x：只支持非 bool 类型。

其中，布尔类型的参数必须使用等号的方式指定。  
  
flag 包的其他函数：

flag.Args()  //返回命令行参数后的其他参数，以 []string 类型  
flag.NArg()  //返回命令行参数后的其他参数个数  
flag.NFlag() //返回使用的命令行参 数个数

结合上面的介绍知识，我们来看一个实例，代码如下：

1. package main
2. import (
3. "flag"
4. "fmt"
5. )
6. var Input\_pstrName = flag.String("name", "gerry", "input ur name")
7. var Input\_piAge = flag.Int("age", 20, "input ur age")
8. var Input\_flagvar int
9. func Init() {
10. flag.IntVar(&Input\_flagvar, "flagname", 1234, "help message for flagname")
11. }
12. func main() {
13. Init()
14. flag.Parse()
15. // After parsing, the arguments after the flag are available as the slice flag.Args() or individually as flag.Arg(i). The arguments are indexed from 0 through flag.NArg()-1
16. // Args returns the non-flag command-line arguments
17. // NArg is the number of arguments remaining after flags have been processed
18. fmt.Printf("args=%s, num=%d\n", flag.Args(), flag.NArg())
19. for i := 0; i != flag.NArg(); i++ {
20. fmt.Printf("arg[%d]=%s\n", i, flag.Arg(i))
21. }
22. fmt.Println("name=", \*Input\_pstrName)
23. fmt.Println("age=", \*Input\_piAge)
24. fmt.Println("flagname=", Input\_flagvar)
25. }

运行结果如下：

go run main.go -name "aaa" -age=123 -flagname=999  
args=[], num=0  
name= aaa  
age= 123  
flagname= 999

#### 自定义 Value

另外，我们还可以创建自定义 flag，只要实现 flag.Value 接口即可（要求 receiver 是指针类型），这时候可以通过如下方式定义该 flag：

flag.Var(&flagVal, "name", "help message for flagname")

【示例】解析喜欢的编程语言，并直接解析到 slice 中，我们可以定义如下 sliceValue 类型，然后实现 Value 接口：

1. package main
2. import (
3. "flag"
4. "fmt"
5. "strings"
6. )
7. //定义一个类型，用于增加该类型方法
8. type sliceValue []string
9. //new一个存放命令行参数值的slice
10. func newSliceValue(vals []string, p \*[]string) \*sliceValue {
11. \*p = vals
12. return (\*sliceValue)(p)
13. }
14. /\*
15. Value接口：
16. type Value interface {
17. String() string
18. Set(string) error
19. }
20. 实现flag包中的Value接口，将命令行接收到的值用,分隔存到slice里
21. \*/
22. func (s \*sliceValue) Set(val string) error {
23. \*s = sliceValue(strings.Split(val, ","))
24. return nil
25. }
26. //flag为slice的默认值default is me,和return返回值没有关系
27. func (s \*sliceValue) String() string {
28. \*s = sliceValue(strings.Split("default is me", ","))
29. return "It's none of my business"
30. }
31. /\*
32. 可执行文件名 -slice="java,go" 最后将输出[java,go]
33. 可执行文件名 最后将输出[default is me]
34. \*/
35. func main(){
36. var languages []string
37. flag.Var(newSliceValue([]string{}, &languages), "slice", "I like programming `languages`")
38. flag.Parse()
39. //打印结果slice接收到的值
40. fmt.Println(languages)
41. }

通过-slice go,php 这样的形式传递参数，languages 得到的就是 [go, php]，如果不加-slice 参数则打印默认值[default is me]，如下所示：

go run main.go -slice go,php,java  
[go php java]

flag 中对 Duration 这种非基本类型的支持，使用的就是类似这样的方式，即同样实现了 Value 接口。

# Go语言go mod包依赖管理工具使用详解

< [Go语言flag包：命令行参数解析](http://c.biancheng.net/view/5573.html)[示例：使用Go语言生成二维码](http://c.biancheng.net/view/vip_7341.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

最早的时候，Go语言所依赖的所有的第三方库都放在 GOPATH 这个目录下面，这就导致了同一个库只能保存一个版本的代码。如果不同的项目依赖同一个第三方的库的不同版本，应该怎么解决？  
  
go module 是Go语言从 1.11 版本之后官方推出的版本管理工具，并且从 Go1.13 版本开始，go module 成为了Go语言默认的依赖管理工具。  
  
Modules 官方定义为：

Modules 是相关 Go 包的集合，是源代码交换和版本控制的单元。Go语言命令直接支持使用 Modules，包括记录和解析对其他模块的依赖性，Modules 替换旧的基于 GOPATH 的方法，来指定使用哪些源文件。

## 如何使用 Modules？

1) 首先需要把 golang 升级到 1.11 版本以上（现在 1.13 已经发布了，建议使用 1.13）。  
  
2) 设置 GO111MODULE。

#### GO111MODULE

在Go语言 1.12 版本之前，要启用 go module 工具首先要设置环境变量 GO111MODULE，不过在Go语言 1.13 及以后的版本则不再需要设置环境变量。通过 GO111MODULE 可以开启或关闭 go module 工具。

* GO111MODULE=off 禁用 go module，编译时会从 GOPATH 和 vendor 文件夹中查找包；
* GO111MODULE=on 启用 go module，编译时会忽略 GOPATH 和 vendor 文件夹，只根据 go.mod下载依赖；
* GO111MODULE=auto（默认值），当项目在 GOPATH/src 目录之外，并且项目根目录有 go.mod 文件时，开启 go module。

Windows 下开启 GO111MODULE 的命令为：

set GO111MODULE=on 或者 set GO111MODULE=auto

MacOS 或者 Linux 下开启 GO111MODULE 的命令为：

export GO111MODULE=on 或者 export GO111MODULE=auto

在开启 GO111MODULE 之后就可以使用 go module 工具了，也就是说在以后的开发中就没有必要在 GOPATH 中创建项目了，并且还能够很好的管理项目依赖的第三方包信息。  
  
常用的go mod命令如下表所示：

|  |  |
| --- | --- |
| **命令** | **作用** |
| go mod download | 下载依赖包到本地（默认为 GOPATH/pkg/mod 目录） |
| go mod edit | 编辑 go.mod 文件 |
| go mod graph | 打印模块依赖图 |
| go mod init | 初始化当前文件夹，创建 go.mod 文件 |
| go mod tidy | 增加缺少的包，删除无用的包 |
| go mod vendor | 将依赖复制到 vendor 目录下 |
| go mod verify | 校验依赖 |
| go mod why | 解释为什么需要依赖 |

#### GOPROXY

proxy 顾名思义就是代理服务器的意思。大家都知道，国内的网络有防火墙的存在，这导致有些Go语言的第三方包我们无法直接通过go get命令获取。GOPROXY 是Go语言官方提供的一种通过中间代理商来为用户提供包下载服务的方式。要使用 GOPROXY 只需要设置环境变量 GOPROXY 即可。  
  
目前公开的代理服务器的地址有：

* goproxy.io；
* goproxy.cn：（推荐）由国内的七牛云提供。

Windows 下设置 GOPROXY 的命令为：

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

MacOS 或 Linux 下设置 GOPROXY 的命令为：

export GOPROXY=https://goproxy.cn

Go语言在 1.13 版本之后 GOPROXY 默认值为 https://proxy.golang.org，在国内可能会存在下载慢或者无法访问的情况，所以十分建议大家将 GOPROXY 设置为国内的 goproxy.cn。

#### 使用go get命令下载指定版本的依赖包

执行go get 命令，在下载依赖包的同时还可以指定依赖包的版本。

* 运行go get -u命令会将项目中的包升级到最新的次要版本或者修订版本；
* 运行go get -u=patch命令会将项目中的包升级到最新的修订版本；
* 运行go get [包名]@[版本号]命令会下载对应包的指定版本或者将对应包升级到指定的版本。

提示：go get [包名]@[版本号]命令中版本号可以是 x.y.z 的形式，例如 go get foo@v1.2.3，也可以是 git 上的分支或 tag，例如 go get foo@master，还可以是 git 提交时的哈希值，例如 go get foo@e3702bed2。

## 如何在项目中使用

【示例 1】创建一个新项目：  
  
1) 在 GOPATH 目录之外新建一个目录，并使用go mod init初始化生成 go.mod 文件。

go mod init hello  
go: creating new go.mod: module hello

go.mod 文件一旦创建后，它的内容将会被 go toolchain 全面掌控，go toolchain 会在各类命令执行时，比如go get、go build、go mod等修改和维护 go.mod 文件。  
  
go.mod 提供了 module、require、replace 和 exclude 四个命令：

* module 语句指定包的名字（路径）；
* require 语句指定的依赖项模块；
* replace 语句可以替换依赖项模块；
* exclude 语句可以忽略依赖项模块。

初始化生成的 go.mod 文件如下所示：

module hello  
  
go 1.13

2) 添加依赖。  
  
新建一个 main.go 文件，写入以下代码：

1. package main
2. import (
3. "net/http"
4. "github.com/labstack/echo"
5. )
6. func main() {
7. e := echo.New()
8. e.GET("/", func(c echo.Context) error {
9. return c.String(http.StatusOK, "Hello, World!")
10. })
11. e.Logger.Fatal(e.Start(":1323"))
12. }

执行go run main.go运行代码会发现 go mod 会自动查找依赖自动下载：

go run main.go  
go: finding github.com/labstack/echo v3.3.10+incompatible  
go: downloading github.com/labstack/echo v3.3.10+incompatible  
go: extracting github.com/labstack/echo v3.3.10+incompatible  
go: finding github.com/labstack/gommon v0.3.0  
......  
go: finding golang.org/x/text v0.3.0  
  
   \_\_\_\_    \_\_  
  / \_\_/\_\_\_/ /  \_\_\_  
/ \_// \_\_/ \_ \/ \_ \  
/\_\_\_/\\_\_/\_//\_/\\_\_\_/ v3.3.10-dev  
High performance, minimalist Go web framework  
https://echo.labstack.com  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_O/\_\_\_\_\_\_\_  
                                                      O\  
⇨ http server started on [::]:1323

现在查看 go.mod 内容：

module hello  
  
go 1.13  
  
require (  
    github.com/labstack/echo v3.3.10+incompatible // indirect  
    github.com/labstack/gommon v0.3.0 // indirect  
    golang.org/x/crypto v0.0.0-20191206172530-e9b2fee46413 // indirect  
)

go module 安装 package 的原则是先拉取最新的 release tag，若无 tag 则拉取最新的 commit，详见[Modules 官方](https://github.com/golang/go/wiki/Modules" \t "http://c.biancheng.net/view/_blank)介绍。  
  
go 会自动生成一个 go.sum 文件来记录 dependency tree：

github.com/davecgh/go-spew v1.1.0/go.mod h1:J7Y8YcW2NihsgmVo/mv3lAwl/skON4iLHjSsI+c5H38=  
github.com/labstack/echo v3.3.10+incompatible h1:pGRcYk231ExFAyoAjAfD85kQzRJCRI8bbnE7CX5OEgg=  
github.com/labstack/echo v3.3.10+incompatible/go.mod h1:0INS7j/VjnFxD4E2wkz67b8cVwCLbBmJyDaka6Cmk1s=  
github.com/labstack/gommon v0.3.0 h1:JEeO0bvc78PKdyHxloTKiF8BD5iGrH8T6MSeGvSgob0=  
github.com/labstack/gommon v0.3.0/go.mod h1:MULnywXg0yavhxWKc+lOruYdAhDwPK9wf0OL7NoOu+k=  
github.com/mattn/go-colorable v0.1.2 h1:/bC9yWikZXAL9uJdulbSfyVNIR3n3trXl+v8+1sx8mU=  
... 省略很多行

再次执行脚本go run main.go发现跳过了检查并安装依赖的步骤。  
  
可以使用命令go list -m -u all来检查可以升级的 package，使用go get -u need-upgrade-package升级后会将新的依赖版本更新到 go.mod \* 也可以使用go get -u升级所有依赖。  
  
【示例 2】改造现有项目。  
  
项目目录结构为：

├─ main.go  
│  
└─ api  
      └─ apis.go

main.go 源码为：

1. package main
2. import (
3. api "./api" // 这里使用的是相对路径
4. "github.com/labstack/echo"
5. )
6. func main() {
7. e := echo.New()
8. e.GET("/", api.HelloWorld)
9. e.Logger.Fatal(e.Start(":1323"))
10. }

api/apis.go 源码为：

1. package api
2. import (
3. "net/http"
4. "github.com/labstack/echo"
5. )
6. func HelloWorld(c echo.Context) error {
7. return c.JSON(http.StatusOK, "hello world")
8. }

1) 使用 go mod init \*\*\* 初始化 go.mod。

go mod init hello  
go: creating new go.mod: module hello

2) 运行go run main.go。

go run main.go  
go: finding golang.org/x/crypto latest  
build \_/D\_/code/src/api: cannot find module for path \_/D\_/code/src/api

首先还是会查找并下载安装依赖，然后运行脚本 main.go，这里会抛出一个错误：

build \_/D\_/code/src/api: cannot find module for path \_/D\_/code/src/api

但是 go.mod 已经更新：

module hello  
  
go 1.13  
  
require (  
    github.com/labstack/echo v3.3.10+incompatible // indirect  
    github.com/labstack/gommon v0.3.0 // indirect  
    golang.org/x/crypto v0.0.0-20191206172530-e9b2fee46413 // indirect  
)

那为什么会抛出这个错误呢？  
  
这是因为 main.go 中使用 internal package 的方法跟以前已经不同了，由于 go.mod 会扫描同工作目录下所有 package 并且变更引入方法，必须将 hello 当成路径的前缀，也就是需要写成 import hello/api，以往 GOPATH/dep 模式允许的 import ./api 已经失效。  
  
3) 更新旧的 package import 方式。  
  
所以 main.go 需要改写成：

1. package main
2. import (
3. api "hello/api" // 这里使用的是相对路径
4. "github.com/labstack/echo"
5. )
6. func main() {
7. e := echo.New()
8. e.GET("/", api.HelloWorld)
9. e.Logger.Fatal(e.Start(":1323"))
10. }

提示：在 Go语言 1.11 版本下使用 go mod 时可能会遇到 go build github.com/valyala/fasttemplate: module requires go 1.12 这种错误，遇到类似这种需要升级到 1.12 的问题，直接升级到Go语言1.12 版本以上就好了。

4) 到这里就和新创建一个项目没什么区别了。

## 使用 replace 替换无法直接获取的 package

由于某些已知的原因，并不是所有的 package 都能成功下载，比如：golang.org 下的包。  
  
modules 可以通过在 go.mod 文件中使用 replace 指令替换成 github 上对应的库，比如：

replace (  
    golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a => github.com/golang/crypto v0.0.0-20190313024323-a1f597ede03a  
)

或者

replace golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a => github.com/golang/crypto v0.0.0-20190313024323-a1f597ede03a

# Go语言生成二维码

< [Go语言go mod包依赖管理工具](http://c.biancheng.net/view/5712.html)[Go语言Context（上下文）](http://c.biancheng.net/view/vip_7342.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

二维码作为一种快速的输入手段越来越流行，支付，添加好友，买东西，扫个二维码即可，可以说二维码已经深深地融入了我们的生活中。那么二维码是如何制作生成的呢？我们如何制作自己的二维码呢？

## 什么是二维码？

二维条码是指在一维条码的基础上扩展出另一维具有可读性的条码，使用黑白矩形图案表示二进制数据，被设备扫描后可获取其中所包含的信息。一维条码的宽度记载着数据，但其长度没有记载数据，而二维条码的长度、宽度均记载着数据。  
  
二维条码有一维条码没有的“定位点”和“容错机制”，容错机制在即使没有辨识到全部的条码、或是说条码有污损时，也可以正确地还原条码上的信息。

## 使用Go语言生成二维码图片

使用Go语言编程时，生成任意内容的二维码是非常方便的，因为我们有 go-qrcode 这个库。该库的源代码托管在 github 上，大家可以从 github 上（[https://github.com/skip2/go-qrcode](https://github.com/skip2/go-qrcode" \t "http://c.biancheng.net/view/_blank)）下载并使用这个库。  
  
go-qrcode 的使用很简单，假如要为我们的官网 [http://c.biancheng.net/](http://c.biancheng.net/" \t "http://c.biancheng.net/view/_blank) 生成一张 256\*256 的图片，可以使用如下代码：

[纯文本复制](http://c.biancheng.net/view/vip_7341.html)

1. package main
2. import "github.com/skip2/go-qrcode"
3. func main() {
4. qrcode.WriteFile("http://c.biancheng.net/",qrcode.Medium,256,"./golang\_qrcode.png")
5. }