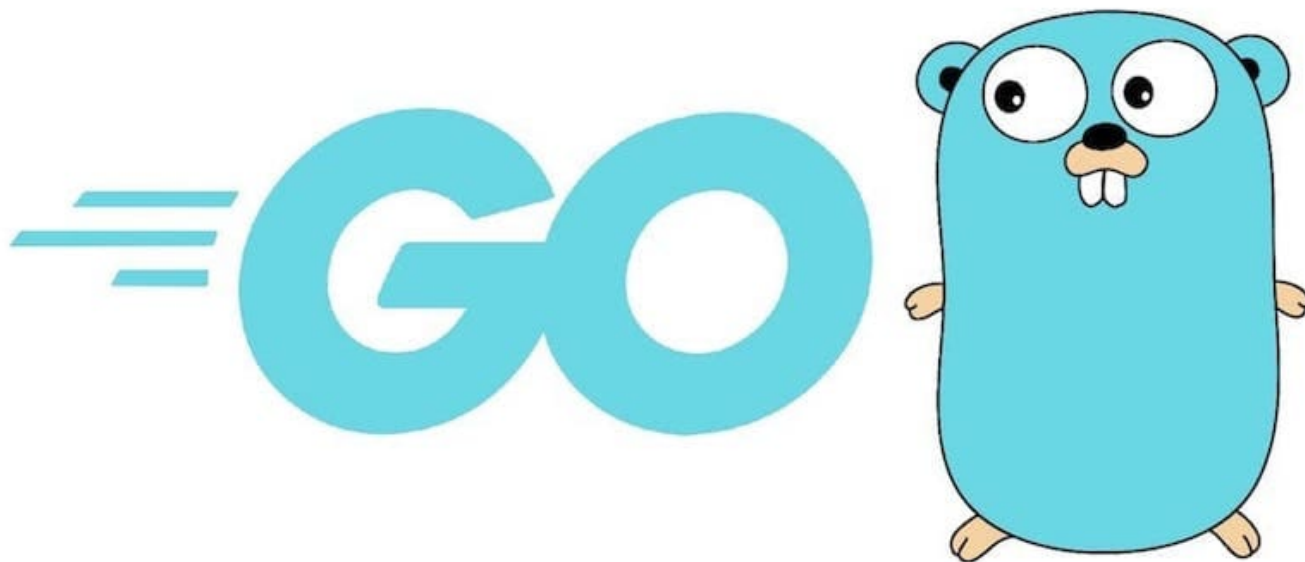


Go 语言简明教程

Go 简明教程系列文章链接:

- [Go 语言简明教程](#) (Aug 6, 2019)
- [Go Gin 简明教程](#) (Aug 7, 2019)
- [Go2 新特性简明教程](#) (Aug 15, 2019)
- [Go Protobuf 简明教程](#) (Jan 11, 2020)
- [Go RPC & TLS 鉴权简明教程](#) (Jan 13, 2020)
- [Go WebAssembly \(Wasm\) 简明教程](#) (Jan 23, 2020)
- [Go Test 单元测试简明教程](#) (Feb 10, 2020)
- [Go Mock \(gomock\)简明教程](#) (Feb 14, 2020)
- [Go Mmap 文件内存映射简明教程](#) (Apr 20, 2020)
- [Go Context 并发编程简明教程](#) (Apr 20, 2020)



Go（又称Golang）是Google开发的一种静态强类型、编译型、并发型，并具有垃圾回收功能的编程语言。—— [Go - wikipedia.org](#)

1 Go 安装

最新版本下载地址官方下载 golang.org，当前是 1.13.6。如无法访问，可以在 studygolang.com/dl 下载

使用 Linux，可以用如下方式快速安装。

```
1  $ wget https://studygolang.com/dl/golang/go1.13.6.linux-amd64.tar.gz
2  $ tar -zxvf go1.13.6.linux-amd64.tar.gz
3  $ sudo mv go /usr/local/
4
5  $ go version
6  go version go1.13.6 linux/amd64
```

从 Go 1.11 版本开始，Go 提供了 **Go Modules** 的机制，推荐设置以下环境变量，第三方包的下载将通过国内镜像，避免出现官方网址被屏蔽的问题。

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

或在 `~/.profile` 中设置环境变量

```
1  export GOPROXY=https://goproxy.cn
```

2 Hello World

新建一个文件 `main.go`，写入

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello World!")
7  }
```

执行 `go run main.go` 或 `go run .`，将会输出

```
1  $ go run .
2  Hello World!
```

如果强制启用了 Go Modules 机制，即环境变量中设置了 `GO111MODULE=on`，则需要先初始化模块
`go mod init hello`

否则会报错误：`go: cannot find main module; see 'go help modules'`

我们的第一个 Go 程序就完成了，接下来我们逐行来解读这个程序：

- `package main`: 声明了 `main.go` 所在的包, Go 语言中使用包来组织代码。一般一个文件夹即一个包, 包内可以暴露类型或方法供其他包使用。
- `import "fmt"`: `fmt` 是 Go 语言的一个标准库/包, 用来处理标准输入输出。
- `func main`: `main` 函数是整个程序的入口, `main` 函数所在的包名也必须为 `main`。
- `fmt.Println("Hello World!")`: 调用 `fmt` 包的 `Println` 方法, 打印出 "Hello World!"

`go run main.go`, 其实是 2 步:

- `go build main.go`: 编译成二进制可执行程序
- `./main`: 执行该程序

3 变量与内置数据类型

3.1 变量(Variable)

Go 语言是静态类型的, 变量声明时必须明确变量的类型。Go 语言与其他语言显著不同的一个地方在于, Go 语言的类型在变量后面。比如 java 中, 声明一个整体一般写成 `int a = 1`, 在 Go 语言中, 需要这么写:

```
1  var a int // 如果没有赋值, 默认为0
2  var a int = 1 // 声明时赋值
3  var a = 1 // 声明时赋值
```

`var a = 1`, 因为 1 是 `int` 类型的, 所以赋值时, `a` 自动被确定为 `int` 类型, 所以类型名可以省略不写, 这种方式还有一种更简单的表达:

```
1  a := 1
2  msg := "Hello World!"
```

3.2 简单类型

空值: `nil`

整型类型: `int`(取决于操作系统), `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, ...

浮点数类型: `float32`, `float64`

字节类型: `byte` (等价于 `uint8`)

字符串类型: `string`

布尔值类型: `boolean`, (`true` 或 `false`)

```
1  var a int8 = 10
2  var c1 byte = 'a'
3  var b float32 = 12.2
```

```
4  var msg = "Hello World"
5  ok := false
```

3.3 字符串

在 Go 语言中，字符串使用 UTF8 编码，UTF8 的好处在于，如果基本是英文，每个字符占 1 byte，和 ASCII 编码是一样的，非常节省空间，如果是中文，一般占3字节。包含中文的字符串的处理方式与纯 ASCII 码构成的字符串有点区别。

我们看下面的例子：

```
1  package main
2
3  import (
4      "fmt"
5      "reflect"
6  )
7  func main() {
8      str1 := "Golang"
9      str2 := "Go语言"
10     fmt.Println(reflect.TypeOf(str2[2]).Kind()) // uint8
11     fmt.Println(str1[2], string(str1[2]))       // 108 l
12     fmt.Printf("%d %c\n", str2[2], str2[2])     // 232 è
13     fmt.Println("len(str2): ", len(str2))       // len(str2):  8
14 }
```

- `reflect.TypeOf().Kind()` 可以知道某个变量的类型，我们可以看到，字符串是以 byte 数组形式保存的，类型是 `uint8`，占1个 byte，打印时需要用 `string` 进行类型转换，否则打印的是编码值。
- 因为字符串是以 byte 数组的形式存储的，所以，`str2[2]` 的值并不等于 语。`str2` 的长度 `len(str2)` 也不是 4，而是 8（Go 占 2 byte，语言占 6 byte）。

正确的处理方式是将 `string` 转为 `rune` 数组

```
1  str2 := "Go语言"
2  runeArr := []rune(str2)
3  fmt.Println(reflect.TypeOf(runeArr[2]).Kind()) // int32
4  fmt.Println(runeArr[2], string(runeArr[2]))    // 35821 语
5  fmt.Println("len(runeArr): ", len(runeArr))    // len(runeArr):  4
```

转换成 `[]rune` 类型后，字符串中的每个字符，无论占多少个字节都用 `int32` 来表示，因而可以正确处理中文。

3.4 数组(array)与切片(slice)

声明数组

```
1  var arr [5]int      // 一维
2  var arr2 [5][5]int  // 二维
```

声明时初始化

```
1  var arr = [5]int{1, 2, 3, 4, 5}
2  // 或 arr := [5]int{1, 2, 3, 4, 5}
```

使用 [] 索引/修改数组

```
1  arr := [5]int{1, 2, 3, 4, 5}
2  for i := 0; i < len(arr); i++ {
3      arr[i] += 100
4  }
5  fmt.Println(arr) // [101 102 103 104 105]
```

数组的长度不能改变，如果想拼接2个数组，或是获取子数组，需要使用切片。切片是数组的抽象。切片使用数组作为底层结构。切片包含三个组件：容量，长度和指向底层数组的指针,切片可以随时进行扩展

声明切片：

```
1  slice1 := make([]float32, 0) // 长度为0的切片
2  slice2 := make([]float32, 3, 5) // [0 0 0] 长度为3容量为5的切片
3  fmt.Println(len(slice2), cap(slice2)) // 3 5
4
```

使用切片：

```
1  // 添加元素，切片容量可以根据需要自动扩展
2  slice2 = append(slice2, 1, 2, 3, 4) // [0, 0, 0, 1, 2, 3, 4]
3  fmt.Println(len(slice2), cap(slice2)) // 7 12
4  // 子切片 [start, end)
5  sub1 := slice2[3:] // [1 2 3 4]
6  sub2 := slice2[:3] // [0 0 0]
7  sub3 := slice2[1:4] // [0 0 1]
8  // 合并切片
9  combined := append(sub1, sub2...) // [1, 2, 3, 4, 0, 0, 0]
```

- 声明切片时可以为切片设置容量大小，为切片预分配空间。在实际使用的过程中，如果容量不够，切片容量会自动扩展。
- `sub2...` 是切片解构的写法，将切片解构为 N 个独立的元素。

3.5 字典(键值对, map)

map 类似于 java 的 HashMap, Python 的字典(dict), 是一种存储键值对(Key-Value)的数据解构。使用方式和其他语言几乎没有区别。

```
1 // 仅声明
2 m1 := make(map[string]int)
3 // 声明时初始化
4 m2 := map[string]string{
5     "Sam": "Male",
6     "Alice": "Female",
7 }
8 // 赋值/修改
9 m1["Tom"] = 18
```

3.6 指针(pointer)

指针即某个值的地址，类型定义时使用符号 `*`，对一个已经存在的变量，使用 `&` 获取该变量的地址。

```
1 str := "Golang"
2 var p *string = &str // p 是指向 str 的指针
3 *p = "Hello"
4 fmt.Println(str) // Hello 修改了 p, str 的值也发生了改变
```

一般来说，指针通常在函数传递参数，或者给某个类型定义新的方法时使用。Go 语言中，参数是按值传递的，如果不使用指针，函数内部将会拷贝一份参数的副本，对参数的修改并不会影响到外部变量的值。如果参数使用指针，对参数的传递将会影响到外部变量。

例如：

```
1 func add(num int) {
2     num += 1
3 }
4
5 func realAdd(num *int) {
6     *num += 1
7 }
8
9 func main() {
10     num := 100
```

```
11     add(num)
12     fmt.Println(num) // 100, num 没有变化
13
14     realAdd(&num)
15     fmt.Println(num) // 101, 指针传递, num 被修改
16 }
```

4 流程控制(if, for, switch)

4.1 条件语句 if else

```
1  age := 18
2  if age < 18 {
3      fmt.Printf("Kid")
4  } else {
5      fmt.Printf("Adult")
6  }
7
8  // 可以简写为:
9  if age := 18; age < 18 {
10     fmt.Printf("Kid")
11 } else {
12     fmt.Printf("Adult")
13 }
```

4.2 switch

```
1  type Gender int8
2  const (
3      MALE  Gender = 1
4      FEMALE Gender = 2
5  )
6
7  gender := MALE
8
9  switch gender {
10 case FEMALE:
11     fmt.Println("female")
12 case MALE:
13     fmt.Println("male")
14 default:
15     fmt.Println("unknown")
```

```
16 }
17 // male
```

- 在这里，使用了 `type` 关键字定义了一个新的类型 `Gender`。
- 使用 `const` 定义了 `MALE` 和 `FEMALE` 2 个常量，Go 语言中没有枚举(enum)的概念，一般可以用常量的方式来模拟枚举。
- 和其他语言不同的地方在于，Go 语言的 `switch` 不需要 `break`，匹配到某个 `case`，执行完该 `case` 定义的行为后，默认不会继续往下执行。如果需要继续往下执行，需要使用 `fallthrough`，例如：

```
1  switch gender {
2  case FEMALE:
3      fmt.Println("female")
4      fallthrough
5  case MALE:
6      fmt.Println("male")
7      fallthrough
8  default:
9      fmt.Println("unknown")
10 }
11 // 输出结果
12 // male
13 // unknown
```

4.3 for 循环

一个简单的累加的例子，`break` 和 `continue` 的用法与其他语言没有区别。

```
1  sum := 0
2  for i := 0; i < 10; i++ {
3      if sum > 50 {
4          break
5      }
6      sum += i
7  }
```

对数组(arr)、切片(slice)、字典(map) 使用 `for range` 遍历：

```
1  nums := []int{10, 20, 30, 40}
2  for i, num := range nums {
3      fmt.Println(i, num)
4  }
5  // 0 10
```



```
6 // 1 20
7 // 2 30
8 // 3 40
9 m2 := map[string]string{
10     "Sam": "Male",
11     "Alice": "Female",
12 }
13
14 for key, value := range m2 {
15     fmt.Println(key, value)
16 }
17 // Sam Male
18 // Alice Female
```

5 函数(functions)

5.1 参数与返回值

一个典型的函数定义如下，使用关键字 `func`，参数可以有多个，返回值也支持有多个。特别地，`package main` 中的 `func main()` 约定为可执行程序入口。

```
1 func funcName(param1 Type1, param2 Type2, ...) (return1 Type3, ...) {
2     // body
3 }
```

例如，实现2个数的加法（一个返回值）和除法（多个返回值）：

```
1
2 func add(num1 int, num2 int) int {
3     return num1 + num2
4 }
5
6 func div(num1 int, num2 int) (int, int) {
7     return num1 / num2, num1 % num2
8 }
9 func main() {
10     quo, rem := div(100, 17)
11     fmt.Println(quo, rem) // 5 15
12     fmt.Println(add(100, 17)) // 117
13 }
```

也可以给返回值命名，简化 `return`，例如 `add` 函数可以改写为

```
1 func add(num1 int, num2 int) (ans int) {
2     ans = num1 + num2
3     return
4 }
```

5.2 错误处理(error handling)

如果函数实现过程中，如果出现不能处理的错误，可以返回给调用者处理。比如我们调用标准库函数 `os.Open` 读取文件，`os.Open` 有2个返回值，第一个是 `*File`，第二个是 `error`，如果调用成功，`error` 的值是 `nil`，如果调用失败，例如文件不存在，我们可以通过 `error` 知道具体的错误信息。

```
1 import (
2     "fmt"
3     "os"
4 )
5
6 func main() {
7     _, err := os.Open("filename.txt")
8     if err != nil {
9         fmt.Println(err)
10    }
11 }
12
13 // open filename.txt: no such file or directory
```

可以通过 `errors.New` 返回自定义的错误

```
1 import (
2     "errors"
3     "fmt"
4 )
5
6 func hello(name string) error {
7     if len(name) == 0 {
8         return errors.New("error: name is null")
9     }
10    fmt.Println("Hello,", name)
11    return nil
12 }
13
14 func main() {
15     if err := hello(""); err != nil {
16         fmt.Println(err)
17     }
18 }
```

```
17     }
18 }
19 // error: name is null
```

error 往往是能预知的错误，但是也可能出现一些不可预知的错误，例如数组越界，这种错误可能会导致程序非正常退出，在 Go 语言中称之为 panic。

```
1 func get(index int) int {
2     arr := [3]int{2, 3, 4}
3     return arr[index]
4 }
5
6 func main() {
7     fmt.Println(get(5))
8     fmt.Println("finished")
9 }
```

```
1 $ go run .
2 panic: runtime error: index out of range [5] with length 3
3 goroutine 1 [running]:
4 exit status 2
```

在 Python、Java 等语言中有 try...catch 机制，在 try 中捕获各种类型的异常，在 catch 中定义异常处理的行为。Go 语言也提供了类似的机制 defer 和 recover。

```
1 func get(index int) (ret int) {
2     defer func() {
3         if r := recover(); r != nil {
4             fmt.Println("Some error happened!", r)
5             ret = -1
6         }
7     }()
8     arr := [3]int{2, 3, 4}
9     return arr[index]
10 }
11
12 func main() {
13     fmt.Println(get(5))
14     fmt.Println("finished")
15 }
```

```
1  $ go run .
2  Some error happened! runtime error: index out of range [5] with length 3
3  -1
4  finished
```

- 在 `get` 函数中，使用 `defer` 定义了异常处理的函数，在协程退出前，会执行完 `defer` 挂载的任务。因此如果触发了 `panic`，控制权就交给了 `defer`。
- 在 `defer` 的处理逻辑中，使用 `recover`，使程序恢复正常，并且将返回值设置为 `-1`，在这里也可以不处理返回值，如果不处理返回值，返回值将被置为默认值 `0`。

6 结构体，方法和接口

6.1 结构体(struct) 和方法(methods)

结构体类似于其他语言中的 `class`，可以在结构体中定义多个字段，为结构体实现方法，实例化等。接下来我们定义一个结构体 `Student`，并为 `Student` 添加 `name`，`age` 字段，并实现 `hello()` 方法。

```
1  type Student struct {
2      name string
3      age  int
4  }
5
6  func (stu *Student) hello(person string) string {
7      return fmt.Sprintf("hello %s, I am %s", person, stu.name)
8  }
9
10 func main() {
11     stu := &Student{
12         name: "Tom",
13     }
14     msg := stu.hello("Jack")
15     fmt.Println(msg) // hello Jack, I am Tom
16 }
```

- 使用 `Student{field: value, ...}` 的形式创建 `Student` 的实例，字段不需要每个都赋值，没有显性赋值的变量将被赋予默认值，例如 `age` 将被赋予默认值 `0`。
- 实现方法与实现函数的区别在于，`func` 和函数名 `hello` 之间，加上该方法对应的实例名 `stu` 及其类型 `*Student`，可以通过实例名访问该实例的字段 `name` 和其他方法了。
- 调用方法通过 `实例名.方法名(参数)` 的方式。

除此之外，还可以使用 `new` 实例化：

```
1  func main() {
2      stu2 := new(Student)
```

```
3     fmt.Println(stu2.hello("Alice")) // hello Alice, I am , name 被赋予默认值""
4 }
```

6.2 接口(interfaces)

一般而言，接口定义了一组方法的集合，接口不能被实例化，一个类型可以实现多个接口。

举一个简单的例子，定义一个接口 `Person` 和对应的方法 `getName()` 和 `getAge()`：

```
1  type Person interface {
2      getName() string
3  }
4
5  type Student struct {
6      name string
7      age  int
8  }
9
10 func (stu *Student) getName() string {
11     return stu.name
12 }
13
14 type Worker struct {
15     name  string
16     gender string
17 }
18
19 func (w *Worker) getName() string {
20     return w.name
21 }
22
23 func main() {
24     var p Person = &Student{
25         name: "Tom",
26         age:  18,
27     }
28
29     fmt.Println(p.getName()) // Tom
30 }
```

- Go 语言中，并不需要显式地声明实现了哪一个接口，只需要直接实现该接口对应的方法即可。
- 实例化 `Student` 后，强制类型转换为接口类型 `Person`。

在上面的例子中，我们在 main 函数中尝试将 Student 实例类型转换为 Person，如果 Student 没有完全实现 Person 的方法，比如我们将 (*Student).getName() 删掉，编译时会出现如下报错信息。

```
1 *Student does not implement Person (missing getName method)
```

但是删除 (*Worker).getName() 程序并不会报错，因为我们并没有在 main 函数中使用。这种情况下我们如何确保某个类型实现了某个接口的所有方法呢？一般可以使用下面的方法进行检测，如果实现不完整，编译期将会报错。

```
1 var _ Person = (*Student)(nil)
2 var _ Person = (*Worker)(nil)
```

- 将空值 nil 转换为 *Student 类型，再转换为 Person 接口，如果转换失败，说明 Student 并没有实现 Person 接口的所有方法。
- Worker 同上。

实例可以强制类型转换为接口，接口也可以强制类型转换为实例。

```
1 func main() {
2     var p Person = &Student{
3         name: "Tom",
4         age: 18,
5     }
6
7     stu := p.(*Student) // 接口转为实例
8     fmt.Println(stu.getAge())
9 }
```

6.3 空接口

如果定义了一个没有任何方法的空接口，那么这个接口可以表示任意类型。例如

```
1 func main() {
2     m := make(map[string]interface{})
3     m["name"] = "Tom"
4     m["age"] = 18
5     m["scores"] = [3]int{98, 99, 85}
6     fmt.Println(m) // map[age:18 name:Tom scores:[98 99 85]]
7 }
```

7 并发编程(goroutine)

7.1 sync

Go 语言提供了 sync 和 channel 两种方式支持协程(goroutine)的并发。

例如我们希望并发下载 N 个资源，多个并发协程之间不需要通信，那么就可以使用 sync.WaitGroup，等待所有并发协程执行结束。

```
1  import (
2      "fmt"
3      "sync"
4      "time"
5  )
6
7  var wg sync.WaitGroup
8
9  func download(url string) {
10     fmt.Println("start to download", url)
11     time.Sleep(time.Second) // 模拟耗时操作
12     wg.Done()
13 }
14
15 func main() {
16     for i := 0; i < 3; i++ {
17         wg.Add(1)
18         go download("a.com/" + string(i+'0'))
19     }
20     wg.Wait()
21     fmt.Println("Done!")
22 }
```

- wg.Add(1): 为 wg 添加一个计数，wg.Done(), 减去一个计数。
- go download(): 启动新的协程并发执行 download 函数。
- wg.Wait(): 等待所有的协程执行结束。

```
1  $ time go run .
2  start to download a.com/2
3  start to download a.com/0
4  start to download a.com/1
5  Done!
6
7  real    0m1.563s
```

可以看到串行需要 3s 的下载操作，并发后，只需要 1s。

7.2 channel

```
1  var ch = make(chan string, 10) // 创建大小为 10 的缓冲信道
2
3  func download(url string) {
4      fmt.Println("start to download", url)
5      time.Sleep(time.Second)
6      ch <- url // 将 url 发送给信道
7  }
8
9  func main() {
10     for i := 0; i < 3; i++ {
11         go download("a.com/" + string(i+'0'))
12     }
13     for i := 0; i < 3; i++ {
14         msg := <-ch // 等待信道返回消息。
15         fmt.Println("finish", msg)
16     }
17     fmt.Println("Done!")
18 }
```

使用 channel 信道，可以在协程之间传递消息。阻塞等待并发协程返回消息。

```
1  $ time go run .
2  start to download a.com/2
3  start to download a.com/0
4  start to download a.com/1
5  finish a.com/2
6  finish a.com/1
7  finish a.com/0
8  Done!
9
10 real    0m1.528s
```

8 单元测试(unit test)

假设我们希望测试 package main 下 calc.go 中的函数，要只需要新建 calc_test.go 文件，在 calc_test.go 中新建测试用例即可。

```
1  // calc.go
```



```
2  package main
3
4  func add(num1 int, num2 int) int {
5      return num1 + num2
6  }
```

```
1  // calc_test.go
2  package main
3
4  import "testing"
5
6  func TestAdd(t *testing.T) {
7      if ans := add(1, 2); ans != 3 {
8          t.Error("add(1, 2) should be equal to 3")
9      }
10 }
```

运行 `go test` , 将自动运行当前 package 下的所有测试用例, 如果需要查看详细的信息, 可以添加 `-v` 参数。

```
1  $ go test -v
2  === RUN   TestAdd
3  --- PASS: TestAdd (0.00s)
4  PASS
5  ok      example 0.040s
```

9 包(Package)和模块(Modules)

9.1 Package

一般来说, 一个文件夹可以作为 package, 同一个 package 内部变量、类型、方法等定义可以相互看到。

比如我们新建一个文件 `calc.go` , `main.go` 平级, 分别定义 `add` 和 `main` 方法。

```
1  // calc.go
2  package main
3
4  func add(num1 int, num2 int) int {
5      return num1 + num2
6  }
```

```
1  // main.go
```

```
2  package main
3
4  import "fmt"
5
6  func main() {
7      fmt.Println(add(3, 5)) // 8
8  }
```

运行 `go run main.go` , 会报错, `add` 未定义:

```
1  ./main.go:6:14: undefined: add
```

因为 `go run main.go` 仅编译 `main.go` 一个文件, 所以命令需要换成

```
1  $ go run main.go calc.go
2  8
```

或

```
1  $ go run .
2  8
```

Go 语言也有 `Public` 和 `Private` 的概念, 粒度是包。如果类型/接口/方法/函数/字段的首字母大写, 则是 `Public` 的, 对其他 `package` 可见, 如果首字母小写, 则是 `Private` 的, 对其他 `package` 不可见。

9.2 Modules

`Go Modules` 是 Go 1.11 版本之后引入的, Go 1.11 之前使用 `$GOPATH` 机制。Go Modules 可以算是较为完善的包管理工具。同时支持代理, 国内也能享受高速的第三方包镜像服务。接下来简单介绍 `go mod` 的使用。Go Modules 在 1.13 版本仍是可选使用的, 环境变量 `GO111MODULE` 的值默认为 `AUTO`, 强制使用 Go Modules 进行依赖管理, 可以将 `GO111MODULE` 设置为 `ON`。

在一个空文件夹下, 初始化一个 Module

```
1  $ go mod init example
2  go: creating new go.mod: module example
```

此时, 在当前文件夹下生成了 `go.mod` , 这个文件记录当前模块的模块名以及所有依赖包的版本。

接着, 我们在当前目录下新建文件 `main.go` , 添加如下代码:

```
1  package main
```

```
2
3  import (
4      "fmt"
5
6      "rsc.io/quote"
7  )
8
9  func main() {
10      fmt.Println(quote.Hello()) // Ahoy, world!
11  }
```

运行 `go run .`，将会自动触发第三方包 `rsc.io/quote` 的下载，具体的版本信息也记录在了 `go.mod` 中：

```
1  module example
2
3  go 1.13
4
5  require rsc.io/quote v3.1.0+incompatible
```

我们在当前目录，添加一个子 package `calc`，代码目录如下：

```
1  demo/
2      |--calc/
3          |--calc.go
4          |--main.go
```

在 `calc.go` 中写入

```
1  package calc
2
3  func Add(num1 int, num2 int) int {
4      return num1 + num2
5  }
```

在 package `main` 中如何使用 package `calc` 中的 `Add` 函数呢？`import` 模块名/子目录名 即可，修改后的 `main` 函数如下：

```
1  package main
2
3  import (
4      "fmt"
5      "example/calc"
```

```
6
7     "rsc.io/quote"
8 )
9
10 func main() {
11     fmt.Println(quote.Hello())
12     fmt.Println(calc.Add(10, 3))
13 }
```

```
1 $ go run .
2 Ahoy, world!
3 13
```

附 参考

- [golang 官方文档 - golang.org](https://golang.org)
- [goproxy.cn 文档 - github.com](https://goproxy.cn)
- [Go Modules - github.com](https://github.com)

专题: [Go 简明教程](#)

本文发表于 2019-08-06, 最后修改于 2021-02-06。

本站永久域名「geektutu.com」, 也可搜索「极客兔兔」找到我。

[上一篇](#) « [机器学习笔试面试题 11-20](#)

[下一篇](#) » [Go Gin 简明教程](#)

赞赏支持



推荐阅读

Go 语言笔试面试题(基础语法)

发表于2020-09-04, 阅读约22分钟

动手写分布式缓存 - GeeCache第五天 分布式节点

发表于2020-02-16, 阅读约32分钟

TensorFlow入门(三) - mnist手写数字识别(可视化训练)

发表于2018-03-29， 阅读约18分钟

#关于我 (9) #Go (48) #百宝箱 (2) #Cheat Sheet (1) #Go语言高性能编程 (20) #友链 (1) #Pandas (3)

#机器学习 (9) #TensorFlow (9) #mnist (5) #Python (10) #强化学习 (3) #OpenAI gym (4) #DQN (1)

#Q-Learning (1) #CNN (1) #TensorFlow 2 (10) #官方文档 (10) #Rust (1)

6 条评论

未登录用户 ▾



说点什么

📘 支持 Markdown 语法

使用 GitHub 登录

预览



SimonYen 发表于 大约 1 年前

写的不错



shiftj18 发表于 11 个月前

通俗易懂！ ps: 4.3节实例的输出结果有点问题 0 10, 1 20, 2 30, 3 40



anbenqishi 发表于 11 个月前

最后一个例子，引用的包名应该是“calc/calc”吧



geektutu 发表于 11 个月前

通俗易懂！ ps: 4.3节实例的输出结果有点问题 0 10, 1 20, 2 30, 3 40

@shiftj18 感谢指出问题，下次更新时修正。



geektutu 发表于 11 个月前

最后一个例子，引用的包名应该是“calc/calc”吧

@anbenqishi go module 模式下，引用方式是 <mod-name>/<dir-name>，这个地方是OK的。





lianfulei 发表于 5 个月前



已经看完了，不错


减小 Go 代码编译后的二进制体积

2 评论 • 15天前

 **bestgopher** —— 使用的测试工程如下，该程序启动了一个 RPC 服务，引用了 log、net/http 和 net/log 三个 package。

如何退出协程 goroutine (超时场景)

2 评论 • 7天前

 **zeromake** —— 还有一个技巧对于chan只需要做一次信号传递，且没有数据传输可以用 ``make(chan struct{})`` 传递时可以用


Go语言动手写Web框架 - Gee第三天 路由 Router

30 评论 • 2天前

 **GaloisZhou** —— 很棒的学习资料！有一个问题 `get /a/:b get /a/c /a/x` 也是去到 `/a/c` -

动手写分布式缓存 - GeeCache第二天 单机并发缓存

32 评论 • 12小时前

 **feixintianxia** —— @geektutu @yangzhuangqiu @walkmiao 保存时复制

Gitalk Plus

© 2021 - 极客兔兔 - 沪ICP备18001798号-1

Powered by [Hexo](#) | Theme [Geektutu](#) 

👁 996632 🗨 33441