GoLang

学习资料，

- 《刘丹冰8h转Golang》
- 《go官方tour》
- 《go官方文档》

安装

官网，选择对应的OS，

```
go version
```

终端运行

```
go run hello.go
```

```
go build hello.go
./hello
```

常识

- .go源文件执行流程，先进入文件导入的包中执行init函数，再执行当前文件的init函数，最后执行一遍main函数。

```
./init.go
./lib1/lib1.go
./lib2/lib2.go
```

```
package main
import (
    "./lib1"
    "./lib2"
)
func init() { }
func main() {
    lib1.Lib1Test()
    lib2.Lib2Test()
}
```

```
package lib1
import "fmt"
func Lib1Test() {
    fmt.Println("lib1 test function")
}
func init() {
    fmt.Println("lib1 init function")
}
```

```
package lib2
import "fmt"
func Lib2Test() {
    fmt.Println("lib2 test function")
}
func init() {
    fmt.Println("lib2 init function")
}
```

Go Modules

需要go版本在11以上

```
go env
```

例子，能在任何路径下使用自己的包

```
go env -w GO111MODULE=on # 将GO111MODULE设置为on，表示打开GoMo
dule
go mod init test
cat go.mod
```

go.mod内容如下

```
module test

go 1.25.5
```

```
tree
```

```
.
├── go.mod
├── init.go
├── lib1
│   └── lib1.go
└── lib2
    └── lib2.go
```

```
package main

import (
    "test/lib1" // 这个是关键，导入自己的包lib1.go和lib2.go
    "test/lib2"
)
```

```
func init() { }

func main() {
    lib1.Lib1Test()
    lib2.Lib2Test()
}
```

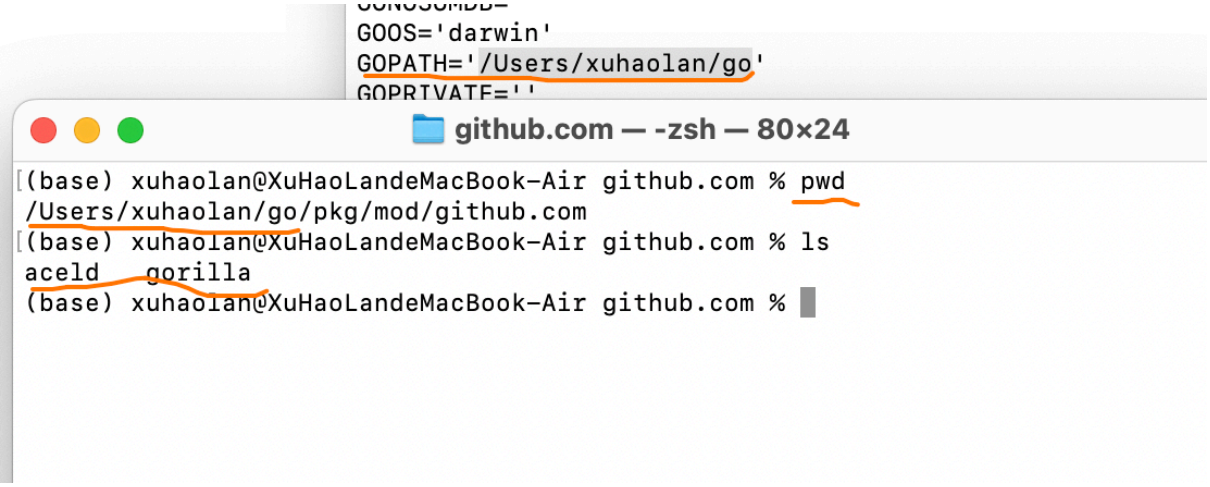
```
package lib1

import "fmt"

func Lib1Test() {
    fmt.Println("lib1 test function")
}

func init() {
    fmt.Println("lib1 init function")
}
```

```
package lib2

import "fmt"

func Lib2Test() {
    fmt.Println("lib2 test function")
}

func init() {
    fmt.Println("lib2 init function")
}
```

```
go run init.go
```

例子，导入github的包，并更改追踪版本，保存在$GOPATH/pkg/mod/github.com路径下

```
go get github.com/aceld/zinx/ziface
```



aceld目录里有我们从Github下载来的包

再查看go.mod文件，显示如下

```
module test

go 1.25.5

require (
    github.com/aceld/zinx v1.2.7 // indirect
    github.com/gorilla/websocket v1.5.0 // indirect
)
```

/aceld下的zinx版本是v1.2.7，现在有一个需求，我们不用v1.2.7，要v1.2.6

从Github下载对应的.zip包，解压缩并发动/aceld下，可能需要改名，如下图所示

```
(base) xuhaolan@XuHaoLandeMacBook-Air aceld % pwd
/Users/xuhaolan/go/pkg/mod/github.com/aceld
(base) xuhaolan@XuHaoLandeMacBook-Air aceld % ls
zinx@v1.2.6     zinx@v1.2.7
(base) xuhaolan@XuHaoLandeMacBook-Air aceld %
```

```
go mod edit -replace=zinx@v1.2.7=zinx@v1.2.6
```

```
module test

go 1.25.5

require (
    github.com/aceld/zinx v1.2.7 // indirect
    github.com/gorilla/websocket v1.5.0 // indirect
)

replace zinx v1.2.7 ⇒ zinx v1.2.6
```

> replace zinx v1.2.7 ⇒ zinx v1.2.6，这行代码表示zinx包使用的v1.2.6版本

```
package main

import (
  "fmt"
  _ "github.com/aceld/zinx/ziface"
)


func main() {
  fmt.Printf("hhh\n")
}
```

```
go run main.go
```

Grammar

Packages

在Go中只要名字首字母大写，都可以被其他的包调用

```
import "fmt"
```

```
import (
    formatPrint "fmt" // 为fmt起一个别名formatPrint
    _ "time"
)
```

Variables

关键字是 `var`

有两个level，分别是package和function

```
package main
import "fmt"
var c, python, java bool // package level

func main() {
    var i int // function level
    fmt.Println(i, c, python, java)
}
```

```
//--------方式1--------// 不给初始值，默认为0
var a int

var b int = 10

var x, y int = 1, 3

//--------方式2--------// 不指定变量的数据类型
var c = "abc"

var u, v = 3.1, "abc"

var (
  One = "one"
  Two = "two"
```

```
  Three = "three"
)

fmt.Printf("%T\n", c) // 打印变量的类型

//--------方式3--------// 不使用var关键字，也不指定变量的数据类型。只能是function level
c := true
i, j, k := 1, "ABC", 3.12
```

Basic Types

type conversion

不同类型之间赋值需要显式转换explicit conversion

`T(v)` 将值 `v` 转换成 `T` 类型

type inference

意思是，当声明变量时，不指明变量的类型，可通过之后赋予的值，来推断变量的类型

bool

string

int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

`uintptr` 配合 `unsafe` 包使用，用于计算内存偏移量。一般不使用

byte

uint8的别名alias

rune

int32的别名，Unicode组织给世界上的每一个字符都分配了一个唯一的数字编码Code Point，rune变量就是用来存储这个Code Point的

float32 float64

complex64 complex128

需要导入 `math/cmplx` 包

Constants

const-ioto

将关键字var改为关键字const

在声明常量时，不能使用 `:=`

```
const (
  BEIJING = 2 * ioto
  SHANGHAI
  CHENGDU
  SHENZHEN
)
```

numeric constants

不显示指定类型，数值常量可以有很高的精度，如至少256位。且只有当它真正被使用时，才会根据上下文变成具体的类型

```
package main

import "fmt"

const (
    Big = 1 << 100 // 2^100
)

func needInt(x int) int { return x*10 + 1 }
func needFloat(x float64) float64 {
    return x * 0.1
}

func main() {
    fmt.Println(needFloat(Big)) // 可行，因为float64存储得了2^100
```

```
    // fmt.Println(needInt(Big)) 报错，因为int存储不了2^100，会溢出
}
```

Zero Values

在变量声明时不给定初值，将会被赋为zero

- `0` 数，包括整型和浮点型
- `false` 布尔
- `""` 字符串
- `0 + 0i` 复数
- `nil` 指针、动态数组Slice、映射Map

Functions

```
func test(a string, b int) (int, string) {

    fmt.Println(a)
    fmt.Println(b)

    c := 10

    return c, "China"
}
```

```
func add(x, y int) int {
    return x + y
}
```

```
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

Function Values

函数可以被当成value来使用，可被当作函数的入参和返回值

```
//--------Function Values被当作其他函数的入参--------//
func compute(fn func(float64, float64) float64) float64 {
    return fn(3, 4)
}

hypot := func(x, y float64) float64 {
    return math.Sqrt(x*x + y*y)
}

compute(hypot)
```

Function Closures

```
//--------Function Values被被当作函数的返回值--------//
func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
} // 记住变量sum的值

pos := adder()

for i := 0; i < 10; i++ {
        fmt.Println(
            pos(i),
        )
    } // 打印 0 1 3 6 10 15 21 28 36 45
```

Flow Control

For

```
for i := 0; i < 10; i++ {...}
```

组成，

- init statement，通常是short variable declaration，可省略
- condition expression，每次迭代都要判断，为true才继续，这个不写将会出现死循环
- post statement，可省略

If

```
if x < 0 {...}
```

```
if v := math.Pow(x, n); v < limit {
    // ...
    // 能使用variable v
} else {
    // ...
    // 能使用variable v
}
```

Switch

效果和If statement类似

```
switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("macOS.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan9, windows...
        fmt.Printf("%s.\n", os)
    }
```

Go的Switch采用自上而下的顺序对case进行求值，一旦某个case符合，则立刻终止，后续的case一定不会被执行，具有短路特性

```
switch os := runtime.GOOS; {
   case os == "darwin": // 这里必须写成布尔表达式
      fmt.Println("macOS.")
   case os == "linux":
      fmt.Println("Linux.")
   }
}
```

condition expression可以缺省，则功能相当于If-then-else语句

Type Switches

```
func do(i interface{}) {
   switch v := i.(type) {
   case int:
      fmt.Printf("Twice %v is %v\n", v, v*2)
   case string:
      fmt.Printf("%q is %v bytes long\n", v, len(v))
   default:
      fmt.Printf("I don't know about type %T!\n", v)
   }
}
```

Pointers

说明，`*` 可以表示指针类型，也可以表示取指针存储的地址中的值；`&` 表示取地址。

```
var aP *int
var a int = 10
aP = &a
*ap = 5
```

Defer

```
package main

import "fmt"
func deferPrint() int {
    fmt.Println("defer Print")
    return 1
}

func returnPrint() int {
    fmt.Println("return Print")
    return 2
}

func test() int {
    defer deferPrint()
    return returnPrint()
}

func main() {
    test()
}
```

Arrays

它的长度不可更改，赋值操作是传递指针

```
//--------方式1--------//
var myArrOne [5]int

//--------方式2--------//
myArrTwo := [9]int {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
func arrFivePrint(arr [5]int) {...} // 值传递
```

Slices

它的长度是动态可变的，赋值操作是传递指针

```
//--------方式1--------//
 := []int {1, 2, 3}

 //--------方式2--------// 此时sliceNone == nil
var sliceNone []int

//--------方式3--------// 4 == len(slice)，4个元素的缺省为0，5 == cap(slic
e)
slice := make([]int, 4, 5)
```

```
arr = append(arr, 1, 2)
```

```
func slicePrint(slice []int) {...} // 传递指针
```

Arrays和Slices共有操作

```
arrTwo[0 : 2]
```

```
// slice切片超级示例
package main

import "fmt"

func main() {
    s := []int{2, 3, 5, 7, 11, 13}
    printSlice(s)

    s = s[:0]
    printSlice(s)

    s = s[:4]
    printSlice(s)

    s = s[2:]
    printSlice(s)
```

```
    s = s[:2]
    printSlice(s)
}

func printSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
} // len() 表示包含的元素个数；cap() 表示容量大小
```

```
arrOne := arrTwo[0：2] // 浅拷贝/传递指针
copy(arrOne, arrTwo) // 深拷贝/值传递
```

Maps

本质上就是key-value pair

```
//--------方式1--------//
var mapOne map[string]int
mapOne = make(map[string]int, 4)

//--------方式2--------//
mapTwo := make(map[string]int, 5)

//--------方式3--------//
mapThree := map[string]int {
  "five"：5,
  "six"：6, // 此,不能被省略掉
}
```

```
mapThree["five"] = 555
mapOne["two"] = 2
```

```
value, ok = mapThree["seven"]
```

```
delete(mapOne, "two")
```

```
for key, value := range mapOne {...}
```

```
func test(mapMy map[string]int) {...} // 传递指针
```

Range

在for loop中迭代slice或map

```
for index, value := range arr {...} // 迭代arr元素的下标index和值value
```

Type

```
type myint int
```

```
type User struct {
    Name string
    Age  int
}

users := []User{
    {Name: "张三", Age: 18},
    {Name: "李四", Age: 20},
}
```

Structs-Methods(和Type紧密关联)

```
type Person struct {
    nick string
    age int
}
```

```
//--------方式1--------//
var haolanxu Person
haolanxu.nick = "lanlan"
```

```
haolanxu.age = 24

//--------方式2--------//
personOne := Person{nick : "haolanxu", age : 24}

//--------方式3--------//
personPoint := &personOne // personPoint指针也用dot operator操作结构体中的field
personPoint.nick = "xu haolan"

//--------方式4--------//
var (
    person1 = Person{"xhs", 22}
    person2 = Person{nick: "xhl"}
    person3 = Person{}
    person4  = &Person{"cx", 18}
)
```

```
func changeInfo(people Person) {...} // 值传递
func changeInfo(people *Person) {...} // 指针传递
```

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}
```

```
func (p Person) sayHello() {
    fmt.Println(p.nick, "say hell0!!!")
}

func (this * Person) setInfo(nick string, age int) {
  this.nick = nick
```

```
  this.age = age
} // 推荐使用这个，能直接修改值，不用copy所以效率更高

func (this Person) printfInfo() {
  fmt.Println(this.nick)
  fmt.Println(this.age)
}
```

封装

struct名、属性成员名、方法名的首字母大写，表示外部可调用；

继承

在子struct中对父struct的属性增强（添加额外属性成员）以及重写或添加一些方法

```
type Child struct {
  Person

  peeBedFrequency int
}
```

```
func (this Child) printfInfo() {
  fmt.Println(this.name)
  fmt.Println(this.age)
  fmt.Println(this.peeBedFrequency)
}
```

```
child := Child{Person{name : "litte boy", age : 1}, 10} // 实例子struct
child.printfInfo() // 子struct方法
child.setInfo("baby", 2) // 父struct方法
child.printfInfo() //子struct方法
```

多态-Interface关键字

```
//--------built-in interface--------// fmt包中
type Stringer interface {
```

```
    String() string
}

//--------built-in interface--------//
type error interface {
    Error() string
}
/*
    在调用function时，经常返回error类型的值
    通常值为nil表示成功，为non-nil表示失败
*/

//--------built-in interface--------//
type Reader interface {
    Read(p []byte) (n int, err error)
}
r := strings.NewReader("Hello, Reader!")
b := make([]byte, 8)
n, err := r.Read(b)

//--------built-in interface--------// 包image
type Image interface {
    ColorModel() color.Model
    Bounds() Rectangle
    At(x, y int) color.Color
}
m := image.NewRGBA(image.Rect(0, 0, 100, 100))
m.Bounds()
m.At(0, 0).RGBA()
```

例子：

```
type Object interface {
    Color()
    Size() string
    Weight() string
}
```

```
type Table struct {
    color string
    size string
    weight string
}
//----------------//
type Desk struct {
    color string
    size string
    weight string
}
```

```
func (this Table) Color() {
    fmt.Println("Table color is ...")
}
func (this Table) Size() string {
    fmt.Println("Table size is ...")
    return this.size
}
func (this Table) Weight() string {
    fmt.Println("Table weight is ...")
    return this.weight
}
//----------------//
func (this Desk) Color() {
    fmt.Println("Desk color is ...")
}
func (this Desk) Size() string {
    fmt.Println("Desk size is ...")
    return this.size
}
func (this Desk) Weight() string {
    fmt.Println("Desk weight is ...")
    return this.weight
}
```

```
func ShowObjectInfo(object Object) {
    object.Color()
    fmt.Println(object.Size())
    fmt.Println(object.Weight())
}
```

```
table := Table{"yellow", "medium", "heavy"}
ShowObjectInfo(&table)

fmt.Println("---------------")

desk := Desk{"green", "small", "light"}
ShowObjectInfo(&desk)
```

Empty Interface-Type Assertion

```
func test(arg interface{}) {
    fmt.Printf("%v is %T type\n", arg, arg)
    value, ok := arg.(string) // 断言变量arg是否是string类型
    if !ok {
        fmt.Println("not string type")
    } else {
        fmt.Println("is string type", value)
    }
}
```

GoLang中断言机制的机理，变量/对象的类型构造是type-value pair。

Generics

泛型

- Type parameters

泛型函数，使函数可以处理多种类型，在函数参数之前的方括号中，使用预定义标识符Type Constraint，如comparable要求泛型必须支持==和!=比较

```
func Index[T comparable](s []T, x T) int {
    for i, v := range s {
        if v == x {
            return i
        }
    }
    return -1
}
```

- **Generic types**

泛型类型

```
type List[T any] struct {
    next *List[T]
    val  T
}
```

反射

官方文档

```
import "reflect"
```

反射基础数据类型

```
//--------这是一个非常简单的例子--------//
func test(arg interface{}) {
    fmt.Println("value is ", reflect.ValueOf(arg))
    fmt.Println("type is ", reflect.TypeOf(arg))
}
```

反射struct

```
//--------这是一个有些复杂的例子--------//
type Person struct {
    Name string
```

```
    Age int
}

func (this Person) PrintInfo() {
    fmt.Println("her/his name is", this.Name, "she/he is ", this.Age)
}

func test(arg interface{}) {

    argType := reflect.TypeOf(arg)
    argValue := reflect.ValueOf(arg)

    fmt.Println("type", argType)
    fmt.Println("value", argValue)


    for i := 0; i < argType.NumField(); i ++ { // struct的属性成员
        field := argType.Field(i)
        value := argValue.Field(i)
        fmt.Printf("%s：%v = %v\n", field.Name, field.Type, value)
    }

    for i := 0; i < argType.NumMethod(); i ++ { // struct的方法
        method := argType.Method(i)
        fmt.Printf("%s：%v\n", method.Name, method.Type)
    }
}
```

反射struct中的标签

为什么要为struct中属性成员绑定标签？当ta人导入此结构体时，只要查看标签就知道这个属性成员需不需要。

```
package main

import (
    "reflect"
    "fmt"
```

```
)

type Animals struct { // A 写的结构体
    name string `info:"name" classify:"nick"`
    special string `info:"kind"`
}

func test(arg interface{}) { // B 通过test的来使用 A 写的结构体中的属性成员
    argType := reflect.TypeOf(arg).Elem()

    for i := 0; i < argType.NumField(); i ++ {
        info := argType.Field(i).Tag.Get("info") // B 能提前从 A 得知有什么tag，
如info
        classify := argType.Field(i).Tag.Get("classify") // B 能提前从 A 得知有什
么tag，如classify
        fmt.Println("info: ", info, "classify", classify)
    }
}

func main() {
    var animal Animals
    test(&animal)
}
```

JSON与struct属性成员value互相编码

```
package main

import (
    "fmt"
    "encoding/json"
)

type Movie struct {
    Name string `json:"name"`
    Year int `json:"year"`
    Actors []string `json:"actors"`
```

```
}

func main() {
    movie := Movie{"xijuzhiwang", 2007, []string{"haolanxu" , "xiaolongli"}}

    jsonStr, err := json.Marshal(movie) // 成功返回 nil 给 err
    if err != nil {
        fmt.Println("json marshal error", err)
        return
    }
    fmt.Printf("%s\n", jsonStr)
    /* 打包好的 JSON 文件的 format
    {"name":"xijuzhiwang","year":2007,"actors":["haolanxu","xiaolongli"]}
    */

    film := Movie{}
    err = json.Unmarshal(jsonStr, &film)
    if err != nil { // 成功返回 nil 给 err
        fmt.Println("json unmarshal error", err)
    }
    fmt.Printf("%v\n", film)
    /* 解码 JSON 文件 得到的 format
    {xijuzhiwang 2007 [haolanxu xiaolongli]}
    */
}
```

Concurrency

并发，包含goroutines和channels

Goroutines

线程thread，由操作系统内核Kernel直接管理

协程coroutine，用户态的轻量级线程，操作系统并不知道协程的存在

Go通过M:N（N个协程复用M个线程）GMP模型来调度创建和CPU核心数相等线程M，调度员P把成千上万个任务G分配给这几个线程M去执行

```
package main

import (
    "fmt"
    "time"
)

func testGroutine() {
    i := 0
    for {
        i ++
        fmt.Printf("test goroutine : i = %d\n", i)
        time.Sleep(1 * time.Second)
    }
}

func main() {
    go testGroutine() // 在main协程中开启一个新的协程
    j := 0
    for {
        j ++
        fmt.Printf("main goroutine : j = %d\n", j)
        time.Sleep(1 * time.Second)
    }
}
```

```
package main

import (
    "fmt"
    "time"
)

func testGroutine() {
    i := 0
    for {
        i ++
```

```
        fmt.Printf("test goroutine : i = %d\n", i)
        time.Sleep(1 * time.Second)
    }
}

func main() {
    go testGroutine()

    time.Sleep(10 * time.Second)
}
```

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func main() {
    go func () {
        defer fmt.Println("first layer defer")

        func () { // 匿名函数
            defer fmt.Println("second layer defer")
            runtime.Goexit()
            fmt.Println("SECOND")
        }()
        fmt.Println("FIRST")
    }()

    i := 1
    for {
        i ++
        fmt.Println("i = ", i)
        time.Sleep(1 * time.Second)
```

```
    }
}
```

Channels

带有类型的通道。Channel操作符←，用于接收或发送值，箭头的方向就是数据流的方向。在使用前需要用make关键字创建。默认情况下，发送和接收操作在另一端准备好之前都会阻塞。

```
package main

import (
    "fmt"
)

func main() {
    c := make(chan int) // 创建

    go func() {
        defer fmt.Println("goroutine over")

        fmt.Println("goruntine running...")

        c ← 666 // 向channel中写
    }()

    num := ← c // 从channel中读，会等待666写入到c中，即发生了阻塞

    fmt.Println("num = ", num)
    fmt.Println("main goroutine over")
}
```

```
package main

import (
    "fmt"
    "time"
)
```

```
func main() {
    c := make(chan int, 3) // 缓冲长度为3
    // len()当前通道元素个数，cap()通道容量
    fmt.Println("len of channel c is ", len(c), ";capacity of channel c is ", cap
(c))

    go func() {
        defer fmt.Println("goroutine over")
        for i := 0; i < 4; i ++ {
            c ← i
            fmt.Println("goroutine is running, sending value = ", i, "len(c) is ", len
(c), "cap(c) is ", cap(c))
        }
    } ()

    time.Sleep(2 * time.Second)

    for i := 0; i < 4; i ++ {
        num := ← c
        fmt.Println("num = ", num)
    }

    time.Sleep(3 * time.Second)
}
```

```
package main

import "fmt"

func main() {
    c := make(chan int)

    go func () {
        for i := 0; i < 5; i ++ {
            c ← i
        }
```

```
        close(c)
    }()

    for {
        if data, ok := ← c; ok {
            fmt.Println(data)
        } else {
            break
        }
    }

    fmt.Println("Main Finished...")
}
```

```
package main

import "fmt"

func main() {
    c := make(chan int)

    go func () {
        for i := 0; i < 5; i ++ {
            c ← i
        }

        close(c)
    }()

    for data := range c {
        fmt.Println(data)
    }

    fmt.Println("Main Finished...")
}
```

```
package main

import "fmt"

func test(c, quit chan int) {
    x, y := 1, 1
    for {
        select {
        case c ← x: // 监控channel c 要发送
            x = y
            y = x + y
        case ← quit: // 监控channel quit 要接收
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)

    go func() {
        for i := 0; i < 6; i ++ {
            fmt.Println(←c) // 表示channel c 要发送
        }

        quit ← 0 // // 表示channel quit 要接收
    }()
    test(c, quit)
}
```

```
package main

import (
    "fmt"
    "sync"
```

```
    "time"
)

// SafeCounter 是并发安全的
type SafeCounter struct {
    mu sync.Mutex
    v  map[string]int
}

// Inc 对给定键的计数加一
func (c *SafeCounter) Inc(key string) {
    c.mu.Lock() // 锁定使得一次只有一个 Go 协程可以访问映射 c.v。
    c.v[key]++
    c.mu.Unlock() // 在函数结束前解锁
}

// Value 返回给定键的计数的当前值。
func (c *SafeCounter) Value(key string) int {
    c.mu.Lock() // 锁定使得一次只有一个 Go 协程可以访问映射 c.v。
    defer c.mu.Unlock() // defer实现解锁
    return c.v[key]
}

func main() {
    c := SafeCounter{v: make(map[string]int)}
    for i := 0; i < 1000; i++ {
        go c.Inc("somekey")
    }

    time.Sleep(time.Second)
    fmt.Println(c.Value("somekey"))
}
```