Go语言学习笔记

基础知识

包、变量和函数

包

每个Go程序都是由包所构成的

程序从`main`包开始运行

包名与导入路径的最后一个元素一致

```
package main //程序从main包开始执行

import (
    "fmt"
    "math/rand" //包名与导入路径的最后一个元素一致,在rand包下就是package rand
)

func main() {
    fmt.Println("My favorite number is", rand.Intn(100))
}
```

导入

可以通过圆括号导入也可以通过`import`关键字分组导入

```
package main

import (
    "fmt"
    "math"
)
/*
    也可以通过import关键字分组导入，例如:
    import "fmt"
    import "math"
*/
func main() {
    fmt.Printf("Now you have %g problems.\n", math.Sqrt(7))
}
```

导出名

在Go中，如果一个字母以大写字母开头，那就是已经导出的，在导入一个包时，只能引用其中已导出的名字，任何未导出的名字在该包外均无法访问

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println(math.Pi) //不能使用pi，因为pi是未导出的，不能在该包外访问
}
```

函数

函数可以没有参数或接受多个参数，例如下面的`add`函数有两个`int`类型的参数，`main`函数中没有参数。注意类型实在变量名之后的，这种类型后置的写法增加了代码的可阅读性

```go
package main

import "fmt"

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

func main() {
    fmt.Println(add(42, 13))
}
```

在参数列表中，当多个形参类型相同时，除最后一个形参类型以外，其它形参的类型可以省略

```go
package main

import "fmt"

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

func main() {
    fmt.Println(add(42, 13))
}
```

多值返回

函数可以返回任意数量的返回值

```go
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
```

命名返回值

返回值可以被命名，没有参数的`return`语句返回已经命名的返回值

```go
package main

import "fmt"

func split(sum int) (x, y int) { //将返回值命名
    x = sum * 4 / 9
    y = sum - x
    return //返回的就是x, y
}

func main() {
    fmt.Println(split(89))
}
```

变量

`var`用于声明变量，变量类型也是后置的

```go
package main

import "fmt"

var c, python, java bool //声明了三个布尔值

func main() {
    var i int //声明了一个int值
    fmt.Println(i, c, python, java)
}
```

变量初始化

变量可以包含初始值，当存在初始值则可以省略其类型

```go
package main

import "fmt"

var i, j int = 1, 2 //包含初始值

func main() {
    var c, python, java = true, false, "no!" //初始化值存在，则可以省略其类型,但是要声明
    fmt.Println(i, j, c, python, java)
}
```

短变量声明

可以使用`:=`来赋值，这样可以省略var关键字来声明变量，但是这种方式只能在函数内使用，因为在函数外的每个语句都必须以关键字开始

```go
package main

import "fmt"
var s int = 12 //必须以关键字开始
func main() {
    var i, j int = 1, 2
    k := 3 //使用:=赋值，省去了var声明
    c, python, java := true, false, "no!"

    fmt.Println(i, j, k, c, python, java, s)
}
```

基本类型

Go的基本类型有以下几种

- bool //布尔值， true false
- string //底层使用byte数组实现的
- int/int8/int16/int32/int64 //后面的数字代表位宽
- uint/uint8/uint16/uint32/uint64/uintptr //无符号整数，避免了负数，扩大了取值范围
- byte //uint8的别名，1byte = 8bit
- rune //int32的别名，表示一个Unicode的码点，也就是编码表中某个字符对应的代码值
- float32/float64 //代表浮点数类型，位宽越高，精度越高，跟底层小数的实现原理有关
- complex64/complex128 //复数类型，由实部和虚部联合表示

```go
package main

import (
    "fmt"
    "math/cmplx"
)

var (
    ToBe   bool       = false
    MaxInt uint64     = 1<<64 - 1
    z      complex128 = cmplx.Sqrt(-5 + 12i)
```

```
    a rune = 'a'
)

func main() {
    fmt.Printf("Type: %T Value: %v\n", ToBe, ToBe)
    fmt.Printf("Type: %T Value: %v\n", MaxInt, MaxInt)
    fmt.Printf("Type: %T Value: %v\n", z, z)
    fmt.Printf("Type: %T Value: %v\n", a, a)
}
```

控制台：

```
Type: bool Value: false
Type: uint64 Value: 18446744073709551615
Type: complex128 Value: (2+3i)
Type: int32 Value: 97
```

零值

没有明确初始值的变量声明会被赋予他们的零值

类型	零值
bool	false
string	""
int/int8/int16/int32/int64	0
uint/uint8/uint16/uint32/uint64/uintptr	0
byte	0
rune	0
float32/float64	0.0
complex64/complex128	0.0 + 0.01

```
package main

import "fmt"

func main() {
    var i int
    var f float64
    var b bool
    var s string
    fmt.Printf("%v %v %v %q\n", i, f, b, s)
}
```

控制台：

```
0 0 false ""
```

类型转换

表达式`Type(value)`可以将值value转换为类型Type

例如：

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)

或者直接用 := 来置换
var i := 42
var f := float64(i)
var u := uint(f)
```

在不同类型赋值之间必须要用显示转换

```
package main

import (
    "fmt"
    "math"
)

func main() {
    var x, y int = 3, 4
    var f float64 = math.Sqrt(float64(x*x + y*y))
    var z uint = uint(f)
    fmt.Println(x, y, z)
}
```

类型推导

下面举例两种情况：

- 传递类型

```
var i int //声明i是int类型
j := i //j也是int类型
```

- 取决于精度

```
i := 42 //i是int类型
f := 3.14 //f是float64类型
g := 0.867 + 0.5i //g是complex128类型
```

```go
package main

import "fmt"

func main() {
    v := 3.14
    fmt.Printf("v is of type %T\n", v)
}
```

控制台：

```
v is of type float64
```

常量

常量使用const声明

常量可以是字符、字符串、布尔值或数值

常量不能用`:=`声明

```go
package main

import "fmt"

const Pi = 3.14 //大写是为了导出，不大写也可以是常量

func main() {
    const World = "世界"
    fmt.Println("Hello", World)
    fmt.Println("Happy", Pi, "Day")

    const Truth = true
    fmt.Println("Go rules?", Truth)
}
```

数值常量

未指定类型的常量通常由上下文来决定类型

```go
package main

import "fmt"

const (
    // 将 1 左移 100 位来创建一个非常大的数字
    // 即这个数的二进制是 1 后面跟着 100 个 0
    Big = 1 << 100
    // 再往右移 99 位，即 Small = 1 << 1，或者说 Small = 2
    Small = Big >> 99
)

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

```
func main() {
    fmt.Println(needInt(Small))
    fmt.Println(needFloat(Small))
    fmt.Println(needFloat(Big))
}

控制台：
21
0.2
1.2676506002282295e+29
```

控制流程语句

for

Go只有一种循环结构：`for`循环

for循环由三部分组成，他们分别用分号隔开：

`for 初始化语句：在第一次迭代前执行 ； 条件表达式：在每次迭代的进行前进行判断求值 ； 后置语句：迭代结尾执行`

当条件表达式的布尔值为false，则终止循环迭代

Go中的for循环没有小括号，但是大括号是必须的

```
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum)
}
```

for的初始化语句和后置语句是可选的，也就是可以不写

```
package main

import "fmt"

func main() {
    sum := 1
    for ; sum < 1000; {
        sum += sum
    }
    fmt.Println(sum)
}
```

因此当去掉分号的时候，for就可以当作"while"了

```go
package main

import "fmt"

func main() {
    sum := 1
    for sum < 1000 {
        sum += sum
    }
    fmt.Println(sum)
}
```

当省略掉循环条件就会无限循环

```go
package main

func main() {
    for {
    }
}
```

if

Go的`if`语句与`for`循环类似，表达式外不需要小括号，大括号是必须的

```go
package main

import (
    "fmt"
    "math"
)

func sqrt(x float64) string {
    if x < 0 {
        return sqrt(-x) + "i"
    }
    return fmt.Sprint(math.Sqrt(x))
}

func main() {
    fmt.Println(sqrt(2), sqrt(-4))
}
```

if也同样可以在条件表达式前执行一个简单的语句

```go
package main

import (
    "fmt"
    "math"
)

func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim { //可以在if后面进行声明，用分号与条件表达式分隔开
```

```
        return v
    }
    return lim
}

func main() {
    fmt.Println(
        pow(3, 2, 10),
        pow(3, 3, 20),
    )
}
```

else关键字在Go中仍然有效，if中声明的变量同样可以使用在else块内

```
package main

import (
    "fmt"
    "math"
)

func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    } else {
        fmt.Printf("%g >= %g\n", v, lim)
    }
    return lim
}

func main() {
    fmt.Println(
        pow(3, 3, 20), // v应当为27， lim应当为20
    )
}
```

控制台：

```
27 >= 20
20
```

switch

switch可以说是一连串的`if-else`的简便方法，switch关键字后的语句运行后，返回与表达式值相匹配的case语句所对应的语句的执行结果

Go中的switch不需要break就会自动终止，除非以fallthrough语句结束

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("Go runs on ")
```

```
    switch os := runtime.GOOS; os { //runtime.GOOS，获取当前go运行的操作系统
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan9, windows...
        fmt.Printf("%s.\n", os)
    }
}

控制台：
Go runs on Linux.
```

switch的case语句从上到下顺次执行，直到匹配成功为止

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("When's Saturday?")
    today := time.Now().Weekday()
    switch time.Saturday {
    case today + 0:
        fmt.Println("Today.")
    case today + 1:
        fmt.Println("Tomorrow.")
    case today + 2:
        fmt.Println("In two days.")
    default:
        fmt.Println("Too far away.")
    }
}

控制台：
When's Saturday?
Too far away.
```

没有条件的switch和switch true的结果一样

```
package main

import "fmt"

func main() {
    switch {
    case false:
        fmt.Println("Good morning!")
    case true:
        fmt.Println("Good afternoon.")
    default:
```

```
        fmt.Println("Good evening.")
    }
}

控制台:
Good afternoon.
```

defer

`defer`语句会将函数推迟到外层函数返回之后执行

推迟调用的函数其参数会立即求值，但直到外层函数返回前该函数都不会被调用

```
package main

import "fmt"

func main() {
    x := 1
    defer fmt.Println(x)
    x += x
    fmt.Println(x)
}

控制台:
2
1
```

被defer推迟的函数会压入一个栈结构中，因此被推迟的函数会按照后进先出的顺序被调用

```
package main

import "fmt"

func main() {
    fmt.Println("counting")

    for i := 0; i < 10; i++ {
        defer fmt.Println(i)
    }

    fmt.Println("done")
}

控制台:
counting
done
9
8
7
6
5
4
3
2
1
0
```

struct、slice 和映射

指针

Go拥有指针，指针保存了所对应值的内存地址。

一个类型T，*T表示指向T类型的值的指针，零值为nil

例如`var p *int`表示指向int类型值的指针p

&为取地址操作符，会生成一个指向其操作数的指针

```
i := 42 //i为int类型，值为42
p = &i //p指向i所对应的值的内存地址
```

*操作符表示指针所指向的底层值，也就是指针所指向的内存地址中的数据

```
fmt.Println(*p) //p指针指向i所对应的数值，也就是通过指针p来读取i
*p = 21 //通过指针p来设置i
```

与C不同，Go没有指针运算

```
package main

import "fmt"

func main() {
    i, j := 42, 2701

    p := &i          // 指向 i
    fmt.Println(*p) // 通过指针读取 i 的值
    *p = 21          // 通过指针设置 i 的值
    fmt.Println(i)  // 查看 i 的值

    p = &j         // 指向 j
    *p = *p / 37   // 通过指针对 j 进行除法运算
    fmt.Println(j) // 查看 j 的值
}

控制台：
42
21
73
```

结构体

一个结构体`struct`就是一组字段(属性)，type的作用类似于取乳名、别名，也就是为结构体取一个名字

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
```

```
}

func main() {
    fmt.Println(Vertex{3, 2})
}

控制台
{3 2}
```

结构体中的属性可以通过.`来访问

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    v.X = 4
    fmt.Println(v.X)
}

控制台:
4
```

结构体指针

结构体的字段可以通过结构体指针来访问

如果我们有一个指向结构体的指针p，那么可以通过(*p).X来访问其字段X，但这么写太罗嗦了，因此Go也允许我们使用隐式间接引用，也就是直接写成`p.X`就可以

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    p := &v
    p.X = 1e9
    fmt.Println(v)
    (*p).X = 22
    fmt.Println(v)
}

控制台:
{1000000000 2}
```

```
{22 2}
```

结构体文法

结构体文法通过直接在大括号中列出来字段的值来分配一个新结构体

使用`Name:`的语法可以仅列出部分字段（看下面的例子）

使用`&`来操作结构体返回的是一个指向结构体的指针

```
package main

import "fmt"

type Vertex struct {
    X, Y int
}

var (
    v1 = Vertex{1, 2}  // 创建一个 Vertex 类型的结构体
    v2 = Vertex{X: 1}  // Y:0 被隐式地赋予 使用了Name: 语法
    v3 = Vertex{}      // X:0 Y:0
    p  = &Vertex{1, 2} // 创建一个 *Vertex 类型的结构体（指针）
)

func main() {
    fmt.Println(v1, p, v2, v3)
}

控制台：
{1 2} &{1 2} {1 0} {0 0}
```

数组

类型`[n]T`表示拥有n个T类型的值的数组

表达式`var a [10]int`

会将变量a声明为拥有10个整数的数组

数组的长度是其类型的一部分，因此数组不能改变大小，这看起来是一个限制，不过没有关系，Go提供了更加便利的方式来使用数组

```
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "Hello"
    a[1] = "World"
    fmt.Println(a[0], a[1])
    fmt.Println(a)

    primes := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primes)
}
```

```
控制台:
Hello World
[Hello World]
[2 3 5 7 11 13]
```

切片

每个数组的大小都是固定的，而切片可以为数组的元素提供动态大小的、灵活的视角。在实践中，切片比数组要更常用

类型`[]T`表示一个元素类型为T的切片

切片通过两个下标来界定，即一个上界和一个下界，二者用冒号来分隔:`a[low: high]`

它会选择一个左闭右开的区间，也就是包含low指向的下标的元素，但不包含high指向的下标的元素

以下表达式创建了一个切片，它包含了a中下标从1到3的元素

`a[1:4]`

```
package main

import "fmt"

func main() {
    primes := [6]int{2, 3, 5, 7, 11, 13}

    var s []int = primes[1:4]
    fmt.Println(s)
}

控制台:
[3 5 7]
```

切片就像是数组的引用，它实际还是指向被切片的数组的，也就是切片不会存储任何数据，它只是描述了底层数组的一段

更改切片的元素就会修改其底层数组中对应的元素

也就是说，底层数组相同的切片，都会观测到这些修改

```
package main

import "fmt"

func main() {
    names := [4]string{
        "John",
        "Paul",
        "George",
        "Ringo",
    }
    fmt.Println(names)

    a := names[0:2]
    b := names[1:3]
    fmt.Println(a, b)
```

```
    b[0] = "XXX" //因为切片b指向的是names[1:3]，也就是说b[0]就是name[1]，也是a[1]
    fmt.Println(a, b)
    fmt.Println(names)
}

控制台：
[John Paul George Ringo]
[John Paul] [Paul George]
[John XXX] [XXX George]
[John XXX George Ringo]
```

切片文法类似于没有长度的数组文法

```
package main

import "fmt"

func main() {
    q := []int{2, 3, 5, 7, 11, 13}
    fmt.Println(q)

    r := []bool{true, false, true, true, false, true}
    fmt.Println(r)

    s := []struct {
        i int
        b bool
    }{
        {2, true},
        {3, false},
        {5, true},
        {7, true},
        {11, false},
        {13, true},
    }
    fmt.Println(s)
}

控制台：
[2 3 5 7 11 13]
[true false true true false true]
[{2 true} {3 false} {5 true} {7 true} {11 false} {13 true}]
```

进行切面时，可以利用切片的默认行为来忽略上下界，切片的下界默认是0，上界默认是该切片的长度

对于数组

`var a[10]int`

以下切片都是等价的:

```
a[0:10]
a[:10]
a[0:]
a[:]
```

```go
package main

import "fmt"

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

    s = s[1:4]
    fmt.Println(s)
    fmt.Println(len(s))
    s = s[:2]
    fmt.Println(s)
    fmt.Println(len(s))
    s = s[1:]
    fmt.Println(s)
    fmt.Println(len(s))
}
```

控制台：

```
[3 5 7]
3
[3 5]
2
[5]
1
```

切片的长度与容量

切片拥有**长度**和**容量**

切片的长度就是它所包含的元素个数

切片容量是从它第一个元素开始数，到其底层元素末尾的个数

切片`s`的长度和容量可以通过表达式`len(s)`和`cap(s)`来获取

```go
import "fmt"

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

    // 截取切片使其长度为 0
    s = s[:0]
    printSlice(s)

    // 拓展其长度
    s = s[:4]
    printSlice(s)

    // 舍弃前两个值
    s = s[2:]
    printSlice(s)
}

func printSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

```
控制台：
len=6 cap=6 [2 3 5 7 11 13]
len=0 cap=6 []
len=4 cap=6 [2 3 5 7]
len=2 cap=4 [5 7]
```

切片的零值是nil
切片可以用内建函数make来创建，这也是创建动态数组的方式

make函数会分配一个元素为零值的数组并返回一个引用了它的切片：

```
a := make([]int, 5)  // len(a)=5
```

要指定它的容量，需向 make传入第三个参数：

```
b := make([]int, 0, 5) // len(b)=0, cap(b)=5

b = b[:cap(b)] // len(b)=5, cap(b)=5
b = b[1:]      // len(b)=4, cap(b)=4
```

```
package main

import "fmt"

func main() {
    a := make([]int, 5)
    printSlice("a", a)

    b := make([]int, 0, 5)
    printSlice("b", b)

    c := b[:2]
    printSlice("c", c)

    d := c[2:5]
    printSlice("d", d)
}

func printSlice(s string, x []int) {
    fmt.Printf("%s len=%d cap=%d %v\n",
        s, len(x), cap(x), x)
}

控制台：
a len=5 cap=5 [0 0 0 0 0]
b len=0 cap=5 []
c len=2 cap=5 [0 0]
d len=3 cap=3 [0 0 0]
```

切片可以包含任何类型，甚至包括切片的切片

```
package main

import (
```

```
    "fmt"
    "strings"
)

func main() {
    // 创建一个井字板（经典游戏）
    board := [][]string{
        []string{"_", "_", "_"},
        []string{"_", "_", "_"},
        []string{"_", "_", "_"},
    }

    // 两个玩家轮流打上 X 和 O
    board[0][0] = "X"
    board[2][2] = "O"
    board[1][2] = "X"
    board[1][0] = "O"
    board[0][2] = "X"

    for i := 0; i < len(board); i++ {
        fmt.Printf("%s\n", strings.Join(board[i], " "))
    }
}
```

控制台：

```
X _ X
O _ X
_ _ O
```

向切片追加元素

为切片追加新的元素是种常用的操作，为此Go提供了内建的`append`函数，内建函数的文档对此函数有详细的介绍

```
func append(s []T, vs ...T) []T
```

`append` 的第一个参数 `s` 是一个元素类型为 `T` 的切片，其余类型为 `T` 的值将会追加到该切片的末尾

`append` 的结果是一个包含原切片所有元素加上新添加元素的切片

当 `s` 的底层数组太小，不足以容纳所有给定的值时，它就会分配一个更大的数组。返回的切片会指向这个新分配的数组

```
package main

import "fmt"

func main() {
    var s []int
    printSlice(s)

    temp := &s
    // 添加一个空切片
    s = append(s, 0)
    printSlice(s)
    if &s == temp {
        fmt.Println("相同")
```

```
    }else {
        fmt.Println("不同")
    }

    // 这个切片会按需增长
    s = append(s, 1)
    printSlice(s)
    if &s == temp {
        fmt.Println("相同")
    }else {
        fmt.Println("不同")
    }

    // 可以一次性添加多个元素
    s = append(s, 2, 3, 4)
    printSlice(s)
}

func printSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

控制台：

```
len=0 cap=0 []
len=1 cap=1 [0]
相同
len=2 cap=2 [0 1]
相同
len=5 cap=6 [0 1 2 3 4]
```

Range

for循环的range形式可遍历切片或映射

当使用 `for` 循环遍历切片时，每次迭代都会返回两个值。第一个值为当前元素的下标，第二个值为该下标所对应元素的一份副本

```
package main

import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

控制台：

```
2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
```

```
2**7 = 128
```

可以将下标或值赋予`_`来忽略掉它

```
for i, _ := range pow
for _, value := range pow
```

若只需要索引，忽略第二个变量就可以

```
for i := range pow
```

```
package main

import "fmt"

func main() {
    pow := make([]int, 10)
    for i := range pow {
        pow[i] = 1 << uint(i) // == 2**i
    }
    for _, value := range pow {
        fmt.Printf("%d\n", value)
    }
}
```

控制台：

```
0
4
8
12
16
20
24
28
32
36
```

映射

映射将键映射到值

`map[键的类型]值的类型`

映射的零值为nil，nil映射既没有键，也不能添加键

`make`函数会返回给定类型的映射，并将其初始化备用

```
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m map[string]Vertex
```

```
func main() {
    m = make(map[string]Vertex)
    m["Bell Labs"] = Vertex{
        40.68433, -74.39967,
    }
    m["Bai"] = Vertex{
        20.22, -12.2222,
    }
    fmt.Println(m["Bell Labs"])
    fmt.Println(m["Bai"])
}

控制台：
{40.68433 -74.39967}
{20.22 -12.2222}
```

映射的文法

映射的文法与结构体相似，不过必须有键名

```
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex{
    "Bell Labs": Vertex{
        40.68433, -74.39967,
    },
    "Google": Vertex{
        37.42202, -122.08408,
    },
}

func main() {
    fmt.Println(m)
}

控制台：
map[Bell Labs:{40.68433 -74.39967} Google:{37.42202 -122.08408}]
```

若顶级类型只是一个类型名，你可以在文法的元素中省略它

```
package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex{
```

```
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}

func main() {
    fmt.Println(m)
}

控制台：
map[Bell Labs:{40.68433 -74.39967} Google:{37.42202 -122.08408}]
```

修改映射

在映射 `m` 中插入或修改元素：

```
m[key] = elem
```

获取元素：

```
elem = m[key]
```

删除元素：

```
delete(m, key)
```

通过双赋值检测某个键是否存在：

```
elem, ok := m[key]
```

若 `key` 在 `m` 中，`ok` 为 `true`；否则，`ok` 为 `false`。

若 `key` 不在映射中，那么 `elem` 是该映射元素类型的零值。

同样的，当从映射中读取某个不存在的键时，结果是映射的元素类型的零值。

```
func main() {
    m := make(map[string]int)

    m["Answer"] = 42
    fmt.Println("The value:", m["Answer"])

    m["Answer"] = 48
    fmt.Println("The value:", m["Answer"])

    delete(m, "Answer")
    fmt.Println("The value:", m["Answer"])

    v, ok := m["Answer"]
    fmt.Println("The value:", v, "Present?", ok)
}

控制台：
The value: 42
The value: 48
The value: 0
```

```
The value: 0 Present? false
```

函数值

函数也可以像值一样传递

函数值可以用作函数的参数或返回值

```
package main

import (
    "fmt"
    "math"
)

func compute(fn func(float64, float64) float64) float64 {//fn有两个参数，参数类型是
float,返回值是float，compute返
  回值也是float
    return fn(3, 4)
}

func main() {
    hypot := func(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
    }
    fmt.Println(hypot(5, 12))

    fmt.Println(compute(hypot))
    fmt.Println(compute(math.Pow))
}

控制台：
13
5
81
```

函数的闭包

Go 函数可以是一个闭包。闭包是一个函数值，它引用了其函数体之外的变量。该函数可以访问并赋予其引用的变量的值，换句话说，该函数被这些变量“绑定”在一起。

例如，函数 `adder` 返回一个闭包。每个闭包都被绑定在其各自的 `sum` 变量上。

```
package main

import "fmt"

func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos, neg := adder(), adder()
```

```
    for i := 0; i < 10; i++ {
        fmt.Println(
            pos(i),
            neg(-2*i),
        )
    }
}
```

控制台：

```
0 0
1 -2
3 -6
6 -12
10 -20
15 -30
21 -42
28 -56
36 -72
45 -90
```