# Golang

目标明确

方法可行

流程合理

激励到位

考核有效

## 编程资源

### 入门教程

<http://www.runoob.com/go/go-program-structure.html>

### 官方文档

Golang基础文章搜索：

<https://studygolang.com/search>

程序组织结构：

<https://tonybai.com/2014/10/21/organize-golang-code/>

#### API文档函数库

<https://studygolang.com/pkgdoc>

#### 编程风格

<https://golang.org/ref/spec>

#### 文档生成

#### 日志模块

### 编程环境

<https://tonybai.com/2012/08/17/hello-go/>

#### Golang标准命令详解

<https://github.com/hyper0x/go_command_tutorial>

<https://blog.csdn.net/u012210379/article/details/50443654>

### 程序结构

Go 语言的基础组成有以下几个部分：

* 包声明
* 引入包
* 函数
* 变量
* 语句 & 表达式
* 注释

package main

import "fmt"

func main() {

/\* 这是我的第一个简单的程序 \*/

fmt.Println("Hello, World!")

}

让我们来看下以上程序的各个部分：

1. 第一行代码 *package main* 定义了包名。你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。
2. 下一行 *import "fmt"* 告诉 Go 编译器这个程序需要使用 fmt 包（的函数，或其他元素），fmt 包实现了格式化 IO（输入/输出）的函数。
3. 下一行 *func main()* 是程序开始执行的函数。main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（如果有 init() 函数则会先执行该函数）。
4. 下一行 /\*...\*/ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。
5. 下一行 *fmt.Println(...)* 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。   
   使用 fmt.Print("hello, world\n") 可以得到相同的结果。   
   Print 和 Println 这两个函数也支持使用变量，如：fmt.Println(arr)。如果没有特别指定，它们会以默认的打印格式将变量 arr 输出到控制台。
6. 当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 protected ）。

当前的调试部分可以使用 **go run filename.go** 来执行。

可以生成一个 **build.sh** 脚本，用于在指定位置产生已编译好的 可执文件:

#!/usr/bin/env bash

CURRENT\_DIR=`pwd`

OLD\_GO\_PATH="$GOPATH" #例如: /usr/local/go

OLD\_GO\_BIN="$GOBIN" #例如: /usr/local/go/bin

export GOPATH="$CURRENT\_DIR"

export GOBIN="$CURRENT\_DIR/bin"

#指定并整理当前的源码路径

gofmt -w src

go install test\_hello

export GOPATH="$OLD\_GO\_PATH"

export GOBIN="$OLD\_GO\_BIN"

#### 大型程序组织结构

## 基础语法

<https://www.cnblogs.com/tianyajuanke/p/5207996.html>

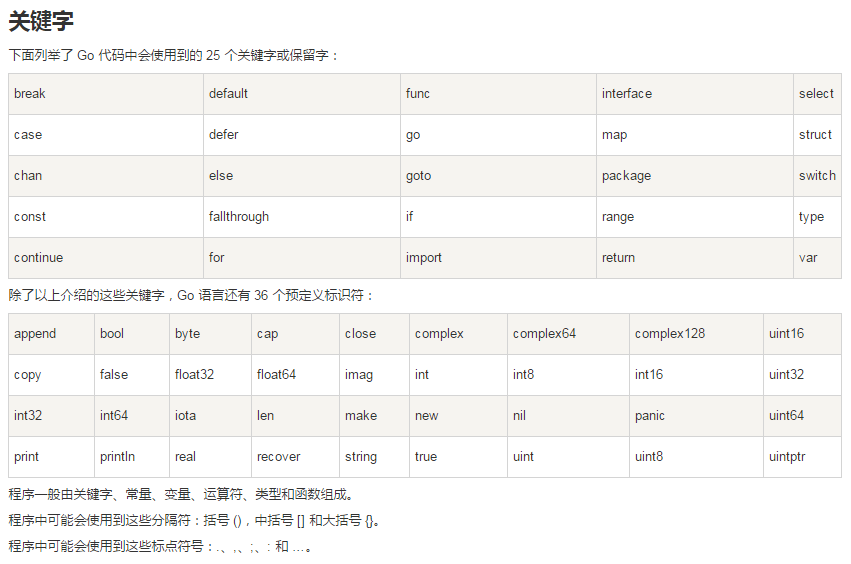
### 关键字

**25个关键字**

　　程序声明：import, package

　　程序实体声明和定义：chan, const, func, interface, map, struct, type, var

　　程序流程控制：go, select, break, case, continue, default, defer, else, fallthrough, for, goto, if, range, return



#### 特殊的空标识符

下划线 \_ 作为一个特殊的标识符，可以用于 import 语句中，仅执行导入包中的 init 方法。

也可以作为赋值语句的左边，表示该变量并不关心且不使用。

此外，标识符首字母的大小写，在GO语言中被用来控制变量或函数的访问权限，类似于其它语言的 public\private。

##### ★函数首字母的大小写，控制 变量或函数的访问权限（public/private）

### 数据类型

类型

　　18个基本类型：bool, string, rune, byte, int, uint, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float32, float64, complex64, complex128

　　7个复合类型：array, struct, function, interface, slice, map, channel

　　其中，切片、字典、通道类型都是引用类型

　　类型的声明一般以 type 关键字开始，然后是自定义的标识符名称，然后是基本类型的名称或复合类型的定义。

　　Unicode字符rune类型是和int32等价的类型，通常用于表示一个Unicode码点。这两个名称可以互换使用。同样byte也是uint8类型的等价类型，byte类型一般用于强调数值是一个原始的数据而不是一个小的整数。

　　最后，还有一种无符号的整数类型uintptr，没有指定具体的bit大小但是足以容纳指针。uintptr类型只有在底层编程是才需要，特别是Go语言和C语言函数库或操作系统接口相交互的地方。

　　一个float32类型的浮点数可以提供大约6个十进制数的精度，而float64则可以提供约15个十进制数的精度；通常应该优先使用float64类型，因为float32类型的累计计算误差很容易扩散，并且float32能精确表示的正整数并不是很大。

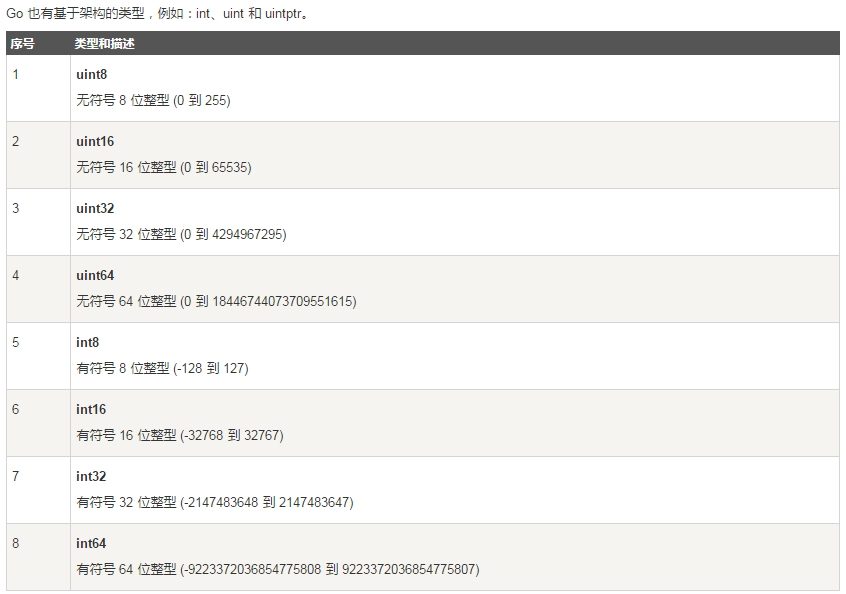
在 Go 编程语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

Go 语言按类别有以下几种数据类型：



#### 数字类型



#### 其他数字类型



#### 浮点型



#### ★数组类型

array: 声明一个长度为 n 、元素类型为 T 的数组为： [n]T, 元素类型可以为基本类型也可以为复合类型，也可以不指定 n ，由推导得出，如： [...]string{"a","b"} , 数组长度 n = len([...]string{"a","b"})，另外如果指定了数组长度，但定义的数组长度小于声明的长度，则以声明长度为准，不足的元素补默认值。同一元素类型，但数组长度不同，则视为不同类型。

##### 数组的四种声明方法

//var <数组名称> [<数组长度>]<数组元素>

//var <数组名称> = [<数组长度>]<数组元素>{元素1,元素2,...}

//var <数组名称> [<数组长度>]<数组元素> = [...]<元素类型>{元素1,元素2,...}

//var <数组名称> [<数组长度>]<数组元素> = [...]<元素类型>{索引1:元素1,索引2:元素2,...}



func main() {

var a [4]int //元素自动初始化为零[0 0 0 0]

b := [4]int{2, 5} //未提供初始化值得元素自动初始化为0 [2 5 0 0]

c := [4]int{5, 3: 10} //可指定索引位置初始化 [5 0 0 10]

d := [...]int{1, 2, 3} //编译器按初始化值数量确定数组长度 [1 2 3]

e := [...]int{10, 3: 100} //支持索引初始化，但注意数组长度与此有关 [10 0 0 100]

fmt.Println(a, b, c, d, e)

}

func main() {

type user struct {

name string

age byte

}

d := [...]user{

{"tom", 20},// 可省略元素类型。

{"lee", 18},// 别忘了最后一行的逗号。

}

fmt.Printf("%#v\n", d)

}

/\*output

[2]main.user{main.user{name:"tom", age:0x14}, main.user{name:"lee", age:0x12}}

\*/

c := [...][2][2]int{ //三维数组

{

{1, 2},

{3, 4},

},

{

{10, 20},

{30, 40},

},

}

#### ★切片类型

<http://www.runoob.com/go/go-slice.html>

<https://studygolang.com/articles/7415>

　　slice: 切片类型的声明为 []T数组，切片类型里没有关于长度的规定，其它跟数组一样，切片类型的零值是 nil。切片总是对应于一个数组，其对应的数组称为底级数组。切片和其底层数组的关系是引用关系，如果有修改都会影响到对方。

切片的数据结构包含了指向其底层数组的指针、切片长度和切片容量。切片的长度很容易理解，切片的容量是什么呢，它是切片第一个元素到底层数组最后一个元素的长度。

Go语言中，切片是长度可变、容量固定的相同的元素序列。Go语言的切片本质是一个数组。容量固定是因为数组的长度是固定的，切片的容量即隐藏数组的长度。长度可变指的是在数组长度的范围内可变。

**（1）切片的创建。**

切片的创建有4种方式：

1）make ( []Type ,length, capacity )

2)  make ( []Type, length)

3) []Type{}

4) []Type{value1 , value2 , ... , valueN }

从3)、4)可见，创建切片跟创建数组唯一的区别在于 Type 前的“ [] ”中是否有数字，为空，则代表切片，否则则代表数组。因为切片是长度可变的。如下是创建切片的示例：

复制代码 代码如下:

func test8() {  
    slice1 := make([]int32, 5, 8)  
    slice2 := make([]int32, 9)  
    slice3 := []int32{}  
    slice4 := []int32{1, 2, 3, 4, 5}  
    fmt.Println(slice1)  
    fmt.Println(slice2)  
    fmt.Println(slice3)  
    fmt.Println(slice4)  
}

输出为：

[0 0 0 0 0]  
[0 0 0 0 0 0 0 0 0]  
[]  
[1 2 3 4 5]

如上，创造了4个切片，3个空切片，一个有值的切片。

**（4）、追加、复制切片：**

复制代码 代码如下:

func test11() {  
    slice := []int32{}  
    fmt.Printf("slice的长度为：%d,slice为：%v\n", len(slice), slice)  
    slice = append(slice, 12, 11, 10, 9)  
    fmt.Printf("追加后，slice的长度为：%d,slice为：%v\n", len(slice), slice)  
    slicecp := make([]int32, (len(slice)))  
    fmt.Printf("slicecp的长度为：%d,slicecp为：%v\n", len(slicecp), slicecp)  
    copy(slicecp, slice)  
    fmt.Printf("复制赋值后，slicecp的长度为：%d,slicecp为：%v\n", len(slicecp), slicecp)  
}

追加、复制切片，用的是内置函数append和copy，copy函数返回的是最后所复制的元素的数量。

**（5）、内置函数append**

内置函数append可以向一个切片后追加一个或多个同类型的其他值。如果追加的元素数量超过了原切片容量，那么最后返回的是一个全新数组中的全新切片。如果没有超过，那么最后返回的是原数组中的全新切片。无论如何，append对原切片无任何影响。如下示例：

复制代码 代码如下:

func test12() {  
    slice := []int32{1, 2, 3, 4, 5, 6}  
    slice2 := slice[:2]  
    \_ = append(slice2, 50, 60, 70, 80, 90)  
    fmt.Printf("slice为：%v\n", slice)  
    fmt.Printf("操作的切片：%v\n", slice2)  
    \_ = append(slice2, 50, 60)  
    fmt.Printf("slice为：%v\n", slice)  
    fmt.Printf("操作的切片：%v\n", slice2)  
}

如上，append方法用了2次，结果返回的结果完全不同，原因是第二次append方法追加的元素数量没有超过 slice 的容量。而无论怎样，原切片slice2都无影响。结果：

slice为：[1 2 3 4 5 6]  
操作的切片：[1 2]  
slice为：[1 2 50 60 5 6]  
操作的切片：[1 2]

希望本文所述对大家的GO语言程序设计有所帮助。

#### ★字典类型：Map(集合)

<http://www.runoob.com/go/go-map.html>

map: 定义一个哈希表的格式为 map[K]V，其中 K 表示键的类型，V表示值的类型，

如: map[string]bool{"IsOK":true, "IsError":false}

##### 定义Map

可以使用内建函数 make 也可以使用 map 关键字来定义 Map:

/\* 声明变量，默认 map 是 nil \*/

var map\_variable map[key\_data\_type]value\_data\_type

/\* 使用 make 函数 \*/

map\_variable := make(map[key\_data\_type]value\_data\_type)

##### map 类似python语言中的dict（字典类型），以key-value形式存储数据

* Key必须是支持==或！=比较运算的类型，不可以是函数，map或slice
* map查找比线性搜素快很多，但比使用索引访问数据的类型慢100倍
* map使用make()函数创建，支持 :=赋值的简单写法

//基本操作

func main() {

var m map[int]string //定义一个map变量

m = map[int]string{} //第一种方式

m = make(map[int]string) //第二种方式：使用make

m := make(map[int]string) //第三种方式make直接初始化

m[1] = "ok" //赋值一对key-value

a := m[1] //获取一个key对应的value值

delete(m, 1) //删除map中的某一个key

fmt.Println(m)

fmt.Println(m)

fmt.Println(a)

fmt.Println(m)

}

/\*output

map\_init----> map[]

map\_add----> map[1:ok]

map\_get----> ok

map\_del----> map[]

\*/

##### ****map，间接排序，通过slice，对key进行排序。****

//间接排序,对key进行排序

package main

import "fmt"

import "sort" //排序包

func main() {

m := map[int]string{1: "a", 2: "b", 3: "c", 4: "d"} //这是一个map类型

s := make([]int, len(m)) //这是一个slice类型

i := 0

for k := range m {

s[i] = k

i ++

}

sort.Ints(s)

fmt.Println(s)

}

/\*输出

[1 2 3 4]

\*/

func main() {

13 yinzhengjie := make(map[int]string)

14 letter := []string{"a","b","c","d","e","f","g","h"}

15 for k,v := range letter{

16 //fmt.Println(k,v)

17 yinzhengjie[k] = v

18 }

19 fmt.Println(yinzhengjie) //注意，字典是无序的哟！

20

21 for i:=0;i<4 ;i++ {

22 delete(yinzhengjie,i) //删除字典中key所对应的value.1

23 }

24 fmt.Println(yinzhengjie)

25

26 for k := range yinzhengjie{

27 delete(yinzhengjie,k) //删除整个字典的数据

28 }

29 fmt.Println(yinzhengjie)

30 fmt.Println(yinzhengjie==nil) //字典的“壳子”还在，空字典也是有地址的。所以返回值是false！

31 }

##### for 迭代操作，类似python，dict操作，for k,v in dic.

//for 迭代操作slice

package main

import "fmt"

func main() {

sum := make([]map[int]string, 5) //初始化一个slice,元素是map类型

for i := range sum {

sum[i] = make(map[int]string, 1)

sum[i][1] = "ok"

fmt.Println(sum[i])

}

fmt.Println(sum)

}

/\*输出

i=0---> map[1:ok]

i=1---> map[1:ok]

i=2---> map[1:ok]

i=3---> map[1:ok]

i=4---> map[1:ok]

sum---> [map[1:ok] map[1:ok] map[1:ok] map[1:ok] map[1:ok]]

\*/

#### ★接口类型

<http://www.runoob.com/go/go-interfaces.html>

　　定义了一组方法声明，接口中可以包含接口

　　GO语言对接口类型的实现是非侵入式的（备注：侵入式是指用户代码与框架代码有依赖，而非侵入式则没有依赖，或者说耦合），只要一个类型定义了某个接口中声明的所有方法，就认为它实现了该接口。

　　一个特殊的接口： interface{} 是一个空接口，不包含任何方法声明，所以GO语言所有的类型都可以看成是它的实现类型，我们就可以使用它实现类似其它语言中的公共基类的功能。比如声明一个字典，键是字符串，值是不确定类型，就可以使用 map[string]interface{}

判断一个类型是否实现了一个接口，可以通过类型断言来确定： \_, ok := interface{}(MyType{}).(MyInterface)

Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。

实例

/\* 定义接口 \*/

type interface\_name interface {

method\_name1 [return\_type]

method\_name2 [return\_type]

method\_name3 [return\_type]

...

method\_namen [return\_type]

}

/\* 定义结构体 \*/

type struct\_name struct {

/\* variables \*/

}

/\* 实现接口方法 \*/

func (struct\_name\_variable struct\_name) method\_name1() [return\_type] {

/\* 方法实现 \*/

}

...

func (struct\_name\_variable struct\_name) method\_namen() [return\_type] {

/\* 方法实现\*/

}

实例

package main

import (

"fmt"

)

type Phone interface {

call()

}

type NokiaPhone struct {

}

func (nokiaPhone NokiaPhone) call() {

fmt.Println("I am Nokia, I can call you!")

}

type IPhone struct {

}

func (iPhone IPhone) call() {

fmt.Println("I am iPhone, I can call you!")

}

func main() {

var phone Phone

phone = new(NokiaPhone)

phone.call()

phone = new(IPhone)

phone.call()

}

在上面的例子中，我们定义了一个接口Phone，接口里面有一个方法call()。然后我们在main函数里面定义了一个Phone类型变量，并分别为之赋值为NokiaPhone和IPhone。然后调用call()方法，输出结果如下：

I am Nokia, I can call you!

I am iPhone, I can call you!

##### 实现接口

func Interface() {

car := MyCar{id: 1}

var icar ICar = car

icar.beep()

icar.drive("yitian")

}

type ICar interface {

beep()

drive(driver string)

}

type MyCar struct {

id int

}

func (car MyCar) beep() {

fmt.Printf("car %v beeps\n", car.id)

}

func (car MyCar) drive(driver string) {

fmt.Printf("%v drives car %v\n", driver, car.id)

}

##### 空接口

什么方法都没定义的接口就是空接口。根据Golang的概念，空接口被任何类型隐式实现，所以空接口可以容纳任何类型。

//空接口可以作为任何类型使用

type Everything interface {

}

var e Everything = "123"

fmt.Println(e)

##### 类型细化

定义和实现接口是一个类型泛化的过程，在这个过程中，我们抹消掉了类型特有的部分，让类型公有的部分能够统一利用。不过有时候需要反过来，将一个接口对象转换为原始的具体类，让我们能够获取更具体的行为。

现在来看看在Golang中，这件事情应该怎么做。再次使用上面定义的类型。可以看到和C系语言的括号强转方式不同，在Golang中是.(T)类型的语法。

//特化类型

myCar := icar.(MyCar)

//myCar是MyCar类型变量

myCar.beep()

这个语法还有一个携带一个成功标志的版本 t, ok := i.(T)。当成功标志为真时，表示成功将接口转换为具体类型，否则表示该接口不是具体类型的实例。

如果要进行多次判断，可以利用switch语句。下面是一个例子。

func testType(i interface{}) {

switch i.(type) {

case string:

fmt.Printf("%v is string\n", i)

case int:

fmt.Printf("%v is int\n", i)

default:

fmt.Printf("%v is interface{}\n", i)

}

}

对这个方法调用多次，可以看到针对不同的类型，方法会返回不同结果。

//类型检测

testType("abc")

testType(123)

testType(nil)

#### ★结构体

<http://www.runoob.com/go/go-structures.html>

可以包含字段，也可以包含匿名字段，一般匿名字段是接口类型，这样就相当于该结构体包含了该接口中的方法，另外也可以在结构里重写隐藏接口中的方法。

<https://studygolang.com/articles/7293>

结构体定义的一般方式如下：

type identifier struct {

field1 type1

field2 type2

...

}

type T struct {a, b int} 也是合法的语法，它更适用于简单的结构体。

var t \*T

t = new(T)

变量 t 是一个指向 T的指针，此时结构体字段的值是它们所属类型的零值，使用 new 函数给一个新的结构体变量分配内存，它返回指向已分配内存的指针。

**无论变量是一个结构体类型还是一个结构体类型指针，都使用同样的 选择器符（selector-notation） 来引用结构体的字段，即：**

type myStruct struct { i int }

var v myStruct // v是结构体类型变量

var p \*myStruct // p是指向一个结构体类型变量的指针

v.i

p.i

##### struct实例：

package main

import (

"fmt"

"strings"

)

type Person struct {

firstName string

lastName string

}

func upPerson(p \*Person) {

p.firstName = strings.ToUpper(p.firstName)

p.lastName = strings.ToUpper(p.lastName)

}

func main() {

// 1-struct as a value type:

var pers1 Person

pers1.firstName = "Chris"

pers1.lastName = "Woodward"

upPerson(&pers1)

fmt.Printf("The name of the person is %s %s\n", pers1.firstName, pers1.lastName)

// 2—struct as a pointer:

pers2 := new(Person)

pers2.firstName = "Chris"

pers2.lastName = "Woodward"

(\*pers2).lastName = "Woodward" // 这是合法的

upPerson(pers2)

fmt.Printf("The name of the person is %s %s\n", pers2.firstName, pers2.lastName)

// 3—struct as a literal:

pers3 := &Person{"Chris", "Woodward"}

upPerson(pers3)

fmt.Printf("The name of the person is %s %s\n", pers3.firstName, pers3.lastName)

}

##### 使用反射访问struct的tag标签：

package main

import (

"fmt"

"reflect"

)

type Person struct {

firstName string `An important answer`

lastName string `The name of the thing`

}

func main() {

tt := Person{"first", "last"}

for i := 0; i < 2; i++ {

refTag(tt, i)

}

}

程序输出：

Tag:An important answer Name:firstName

Tag:The name of the thing Name:lastName

在值和指针上调用方法：

可以有连接到类型的方法，也可以有连接到类型指针的方法。

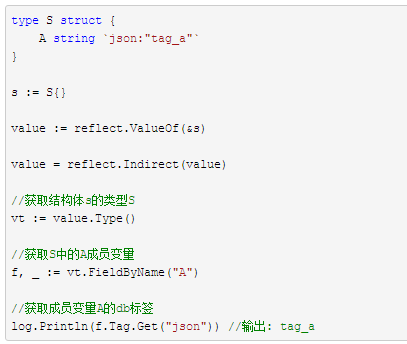
但是这没关系：对于类型 T，如果在 \*T 上存在方法 Meth()，并且 t 是这个类型的变量，那么 t.Meth() 会被自动转换为(&t).Meth()。

指针方法和值方法都可以在指针或非指针上被调用，收者类型是 \*TwoInts 的方法 AddThem()，它能在类型 TwoInts 的值上被调用，这是自动间接发生的。

因此 two2.AddThem 可以替代 (&two2).AddThem()。

##### 获取结构体成员变量的tag信息

由于golang变量大小写和公有私有息息相关，所以码农门很难按照自己的意愿来定义变量名。于是golang提供了tag机制，来给变量提供一个标签，这个标签可以作为一个别名，来给一些存储结构来获取结构体变量名字使用。下面是一个获取结构体成员变量tag信息的例子：



package main

import (

"fmt"

"reflect" // 这里引入reflect模块

)

type User struct {

Name string "user name" //这引号里面的就是tag

Passwd string "user passsword"

}

func main() {

user := &User{"chronos", "pass"}

s := reflect.TypeOf(user).Elem() //通过反射获取type定义

for i := 0; i < s.NumField(); i++ {

fmt.Println(s.Field(i).Tag) //将tag输出出来

}

}

#### ★指针

<http://www.runoob.com/go/go-pointers.html>

　　有一个专门用于存储内存地址的类型 unitptr，它与 int/unit 一样属于数值类型，存储32/64位无符号整数。

可以在一个类型前面使用\*号，来表示对应的指针类型，也可以在可寻址的变量前面使用&，来获取其地址。

#### ★函数类型

<https://studygolang.com/articles/2841>

package main

import "fmt"

type A func(int, int)

func (f A)Serve() {

fmt.Println("serve2")

}

func serve(int,int) {

fmt.Println("serve1")

}

func main() {

a := A(serve)

a(1,2)

a.Serve()

}

type functinTyoe func(int) bool // 声明了一个函数类型

func isOdd(integer int) bool {

if integer%2 == 0 {

return false

}

return true

}

func isEven(integer int) bool {

if integer%2 == 0 {

return true

}

return false

}

// 声明的函数类型在这个地方当做了一个参数

func filter(slice []int, f functinTyoe) []int {

var result []int

for \_, value := range slice {

if f(value) {

result = append(result, value)

}

}

return result

｝

func test(){

slice := []int {1, 2, 3, 4, 5, 7}

fmt.Println("slice = ", slice)

odd := filter(slice, isOdd) // 函数当做值来传递了

fmt.Println("Odd elements of slice are: ", odd)

even := filter(slice, isEven) // 函数当做值来传递了

fmt.Println("Even elements of slice are: ", even)

}

#### 类型断言

比较特殊的表达式有类型断言，如果判断一个表达式 element 的类型是 T 的话，表达式为 element.(T)，意思是 element 不为 nil 且存储在其中的值是T类型。这里有两种情况，如果 T 不是一个接口类型，则 element 必须要为接口类型的值，比如判断 100 是 int 型，不能使用 100.(int)，而要用 interface{}(100).(int) ； 如果T 是一个接口类型，表示断言 element 实现了 T 这个接口。如果函数的参数是一个空接口，则必须断言传入的接口实现类型，才能使用其对应的方法。

#### 类型转换

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。Go 语言类型转换基本格式如下：

type\_name(expression)

type\_name 为类型，expression 为表达式。

以下实例中将整型转化为浮点型，并计算结果，将结果赋值给浮点型变量：

package main

import "fmt"

func main() {

var sum int = 17

var count int = 5

var mean float32

mean = float32(sum)/float32(count)

fmt.Printf("mean 的值为: %f\n",mean)

}

以上实例执行输出结果为：

mean 的值为: 3.400000

### 变量

#### 变量声明

声明变量的一般形式是使用 var 关键字：

var identifier type

##### 第一种，指定变量类型，声明后若不赋值，使用默认值。

var v\_name v\_type

v\_name = value

##### 第二种，根据值自行判定变量类型。

var v\_name = value

##### 第三种，省略var, 注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误。

v\_name := value

// 例如

var a int = 10

var b = 10

c := 10

实例如下：

package main

var a = "菜鸟教程"

var b string = "runoob.com"

var c bool

func main(){

println(a, b, c)

}

##### 多变量声明

//类型相同多个变量, 非全局变量

var vname1, vname2, vname3 type

vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 //和python很像,不需要显示声明类型，自动推断

vname1, vname2, vname3 := v1, v2, v3 //出现在:=左侧的变量不应该是已经被声明过的，否则会导致编译错误

// 这种因式分解关键字的写法一般用于声明全局变量

var (

vname1 v\_type1

vname2 v\_type2

)

实例如下：

package main

var x, y int

var ( // 这种因式分解关键字的写法一般用于声明全局变量

a int

b bool

)

var c, d int = 1, 2

var e, f = 123, "hello"

//这种不带声明格式的只能在函数体中出现

//g, h := 123, "hello"

func main(){

g, h := 123, "hello"

println(x, y, a, b, c, d, e, f, g, h)

}

##### 简短形式，使用 := 赋值操作符

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，声明语句写上 var 关键字其实是显得有些多余了，因此我们可以将它们简写为 a := 50 或 b := false。

a 和 b 的类型（int 和 bool）将由编译器自动推断。

这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

#### 变量：声明定义

　　变量的声明语句以 var 开始，声明多个变量时，和声明多个const的方法相同。

var x string = "df"

var x = "df"

x := "df" //此为简写形式。

#### 数据初始化

　　GO语言的数据初始化有两种方法，一是使用 new ，一是使用 make ，其中 new 是为某个类型分配内存，并设置零值后返回地址，如 new(T) 返回的就是指向T类型值指针值，即\*T。如 new([3]int) 其实相当于 [3]int{0,0,0}，所以它是一种很干净的内存分配策略。

　　make 只用于创建切片类型、字典类型和通道类型（注意这三个类型的特点，都是引用类型），它对这些类型创建之后，还会进行必要的初始化，与 new 不同，它返回的就是指T类型的值，而不是指针值。

定义常量的方式是使用 const ，如 const PI = 3.14，如果定义多个常量可以使用

编译器会自动选择在栈上还是在堆上分配局部变量的存储空间，但可能令人惊讶的是，这个选择并不是由用var还是new声明变量的方式决定的。 如果一个函数里声明一个局部变量，但是将其指针赋给一个全局变量，那么则不能将此局部变量放在栈中，而只能放在堆中，我们可以称之为该局部变量逃逸了，所以关于变量是分配在栈上还是堆上，是由编译器根据情况来选择的。

#### ★合并书写：type

和 var/const 类似，多个 type 定义可合并成组，如下：

type (

Person struct {

Name string

Age int32

}

myfunc func(string) bool

)

 尤其是在函数内部定义一堆临时类型时，可集中书写，可读性更好。

自增/自减运算符不同于其它语言，不能用作表达式，只能当独立的语句使用，且不能前置，只有后置形式，以尽可能避免该运算符带的复杂性问题。

type (

    optionValue struct {

        Value interface{}

        Type optionType

    }

    // Option HTTP option

    Option func(map[string]optionValue) error

)

#### unsafe.Pointer 与 uintptr 的区别：

前者类似 C 语言中的 void\* 万能指针，能安全持有对象，但后者不行，后者只是一种特殊整型，并不引用目标对象，无法阻止垃圾回收器回收对象内存。

#### 语言变量作用域

<http://www.runoob.com/go/go-scope-rules.html>

作用域为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。

Go 语言中变量可以在三个地方声明：

* 函数内定义的变量称为局部变量
* 函数外定义的变量称为全局变量
* 函数定义中的变量称为形式参数

接下来让我们具体了解局部变量、全局变量和形式参数。

### 常量

常量

　　定义常量和多个常量如下：

const PP = iota //0

const QQ = iota //0

const (

A = 1

B = 2

C

D = iota

E

F

)

log.Print(A, B, C, D, E, F)

//输出是： 1 2 2 3 4 5

　　注意，iota 只能在 const 里使用，它是 const 的行数索引器，其值是所在const组中的索引值，单常量定义时 iota 就等于 0。另外，const组中某一项如果不赋值，则默认和上一个值一样，但如果前面是 iota ，则为上一个值+1。使用 iota 可以实现其它语言中的枚举。

常量是一个简单值的标识符，在程序运行时，不会被修改的量。

常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：

const identifier [type] = value

你可以省略类型说明符 [type]，因为编译器可以根据变量的值来推断其类型。

* 显式类型定义： const b string = "abc"
* 隐式类型定义： const b = "abc"

多个相同类型的声明可以简写为：

const c\_name1, c\_name2 = value1, value2

以下实例演示了常量的应用：

package main

import "fmt"

func main() {

const LENGTH int = 10

const WIDTH int = 5

var area int

const a, b, c = 1, false, "str" //多重赋值

area = LENGTH \* WIDTH

fmt.Printf("面积为 : %d", area)

println()

println(a, b, c)

}

以上实例运行结果为：

面积为 : 50

1 false str

#### 枚举：常量还可以用作枚举

const (

Unknown = 0

Female = 1

Male = 2

)

数字 0、1 和 2 分别代表未知性别、女性和男性。

常量可以用len(), cap(), unsafe.Sizeof()函数计算表达式的值。常量表达式中，函数必须是内置函数，否则编译不过：

package main

import "unsafe"

const (

a = "abc"

b = len(a)

c = unsafe.Sizeof(a)

)

func main(){

println(a, b, c)

}

以上实例运行结果为：

abc 3 16

#### iota

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

在每一个const关键字出现时，被重置为0，然后再下一个const出现之前，每出现一次iota，其所代表的数字会自动增加1。

iota 可以被用作枚举值：

const (

a = iota

b = iota

c = iota

)

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2 可以简写为如下形式：

const (

a = iota

b

c

)

##### iota 用法

package main

import "fmt"

func main() {

const (

a = iota //0

b //1

c //2

d = "ha" //独立值，iota += 1

e //"ha" iota += 1

f = 100 //iota +=1

g //100 iota +=1

h = iota //7,恢复计数

i //8

)

fmt.Println(a,b,c,d,e,f,g,h,i)

}

以上实例运行结果为：

0 1 2 ha ha 100 100 7 8

再看个有趣的的 iota 实例：

package main

import "fmt"

const (

i=1<<iota

j=3<<iota

k

l

)

func main() {

    fmt.Println("i=",i)

    fmt.Println("j=",j)

    fmt.Println("k=",k)

    fmt.Println("l=",l)

}

以上实例运行结果为：

i= 1

j= 6

k= 12

l= 24

iota 表示从 0 开始自动加 1，所以 **i=1<<0**, **j=3<<1**（**<<** 表示左移的意思），即：i=1, j=6，这没问题，关键在 k 和 l，从输出结果看 **k=3<<2**，**l=3<<3**。

简单表述:

**i=1**：左移 0 位,不变仍为 1;

* **j=3**：左移 1 位,变为二进制 110, 即 6;
* **k=3**：左移 2 位,变为二进制 1100, 即 12;
* **l=3**：左移 3 位,变为二进制 11000,即 24。

a = "hello"

unsafe.Sizeof(a)

输出结果为：16

字符串类型在 go 里是个结构, 包含指向底层数组的指针和长度,这两部分每部分都是 8 个字节，所以字符串类型大小为 16 个字节。

### 运算符

算符用于在程序运行时执行数学或逻辑运算。

Go 语言内置的运算符有：

* 算术运算符
* 关系运算符
* 逻辑运算符
* 位运算符
* 赋值运算符
* 其他运算符

#### 算术运算符



#### 关系运算符

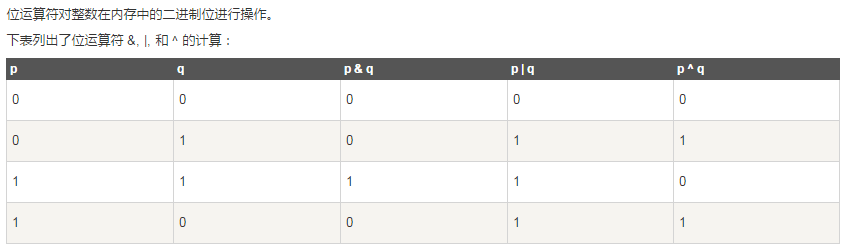


#### 逻辑运算符



#### 位操作符





　　列举一些特殊的操作符，注意下面的位操作符

& 位运算 AND

| 位运算 OR

^ 位运算 XOR

&^ 位清空 (AND NOT)

<< 左移

>> 右移

#### 赋值运算符



#### 其他运算符



以下实例演示了其他运算符的用法：

package main

import "fmt"

func main() {

var a int = 4

var b int32

var c float32

var ptr \*int

/\* 运算符实例 \*/

fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );

fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );

fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

/\* & 和 \* 运算符实例 \*/

ptr = &a    /\* 'ptr' 包含了 'a' 变量的地址 \*/

fmt.Printf("a 的值为 %d\n", a);

fmt.Printf("\*ptr 为 %d\n", \*ptr);

}

以上实例运行结果：

第 1 行 - a 变量类型为 = int

第 2 行 - b 变量类型为 = int32

第 3 行 - c 变量类型为 = float32

a 的值为 4

\*ptr 为 4

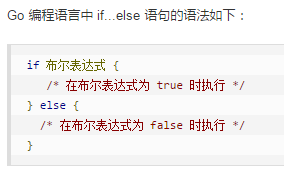
#### 运算符优先级



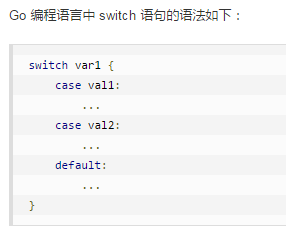
### 条件判断



#### if-else语句



#### switch语句



变量 var1 可以是任何类型，而 val1 和 val2 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同的类型；或者最终结果为相同类型的表达式。

您可以同时测试多个可能符合条件的值，使用逗号分割它们，例如：case val1, val2, val3。

1. 支持多条件匹配

switch{

    case 1,2,3,4:

    default:

}

2. 不同的 case 之间不使用 break 分隔，默认只会执行一个 case。

3. 如果想要执行多个 case，需要使用 fallthrough 关键字，也可用 break 终止。

switch{

    case 1:

    ...

    if(...) {

        break

    }

    fallthrough // 此时switch(1)会执行case1和case2，但是如果满足if条件，则只执行case1

    case 2:

    ...

    case 3:

}

#### select语句

select是Go中的一个控制结构，类似于用于通信的switch语句。每个case必须是一个通信操作，要么是发送要么是接收。

select随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。一个默认的子句应该总是可运行的。



以下描述了 select 语句的语法：

* 每个case都必须是一个通信
* 所有channel表达式都会被求值
* 所有被发送的表达式都会被求值
* 如果任意某个通信可以进行，它就执行；其他被忽略。
* 如果有多个case都可以运行，Select会随机公平地选出一个执行。其他不会执行。   
  否则：
  1. 如果有default子句，则执行该语句。
  2. 如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。
* package main
* import "fmt"
* func main() {
* var c1, c2, c3 chan int
* var i1, i2 int
* select {
* case i1 = <-c1:
* fmt.Printf("received ", i1, " from c1\n")
* case c2 <- i2:
* fmt.Printf("sent ", i2, " to c2\n")
* case i3, ok := (<-c3): // same as: i3, ok := <-c3
* if ok {
* fmt.Printf("received ", i3, " from c3\n")
* } else {
* fmt.Printf("c3 is closed\n")
* }
* default:
* fmt.Printf("no communication\n")
* }
* }
* 以上代码执行结果为：
* no communication

### 循环语句



#### for循环

Go语言的For循环有3中形式，只有其中的一种使用分号。

和 C 语言的 for 一样：

for init; condition; post { }

和 C 的 while 一样：

for condition { }

和 C 的 for(;;) 一样：

for { }

* init： 一般为赋值表达式，给控制变量赋初值；
* condition： 关系表达式或逻辑表达式，循环控制条件；
* post： 一般为赋值表达式，给控制变量增量或减量。

for语句执行过程如下：

* ①先对表达式1赋初值；
* ②判别赋值表达式 init 是否满足给定条件，若其值为真，满足循环条件，则执行循环体内语句，然后执行 post，进入第二次循环，再判别 condition；否则判断 condition 的值为假，不满足条件，就终止for循环，执行循环体外语句。

##### for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

for key, value := range oldMap {

newMap[key] = value

}

#### 实例

package main

import "fmt"

func main() {

var b int = 15

var a int

numbers := [6]int{1, 2, 3, 5}

/\* for 循环 \*/

for a := 0; a < 10; a++ {

fmt.Printf("a 的值为: %d\n", a)

}

for a < b {

a++

fmt.Printf("a 的值为: %d\n", a)

}

for i,x:= range numbers {

fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)

}

}

#### 无限循环

如过循环中条件语句永远不为 false 则会进行无限循环，我们可以通过 for 循环语句中只设置一个条件表达式来执行无限循环：

package main

import "fmt"

func main() {

for true {

fmt.Printf("这是无限循环。\n");

}

}

#### 范围(Range)

<http://www.runoob.com/go/go-range.html>

Go 语言中 range 关键字用于 for 循环中迭代数组(array)、切片(slice)、通道(channel)或集合(map)的元素。在数组和切片中它返回元素的索引和索引对应的值，在集合中返回 key-value 对的 key 值。

#### 实例

package main

import "fmt"

func main() {

//这是我们使用range去求一个slice的和。使用数组跟这个很类似

nums := []int{2, 3, 4}

sum := 0

for \_, num := range nums {

sum += num

}

fmt.Println("sum:", sum)

//在数组上使用range将传入index和值两个变量。上面那个例子我们不需要使用该元素的序号，所以我们使用空白符"\_"省略了。有时侯我们确实需要知道它的索引。

for i, num := range nums {

if num == 3 {

fmt.Println("index:", i)

}

}

//range也可以用在map的键值对上。

kvs := map[string]string{"a": "apple", "b": "banana"}

for k, v := range kvs {

fmt.Printf("%s -> %s\n", k, v)

}

//range也可以用来枚举Unicode字符串。第一个参数是字符的索引，第二个是字符（Unicode的值）本身。

for i, c := range "go" {

fmt.Println(i, c)

}

}

以上实例运行输出结果为：

sum: 9

index: 1

a -> apple

b -> banana

0 103

1 111

Go Range 简单循环：

package main

import "fmt"

func main(){

nums := []int{1,2,3,4};

length := 0;

for range nums {

length++;

}

fmt.Println( length);

}

### 函数

<http://www.runoob.com/go/go-functions.html>

#### 函数定义

Go 语言函数定义格式如下：

func function\_name( [parameter list] ) [return\_types] {

函数体

}

函数定义解析：

* func：函数由 func 开始声明
* function\_name：函数名称，函数名和参数列表一起构成了函数签名。
* parameter list：参数列表，参数就像一个占位符，当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。参数列表指定的是参数类型、顺序、及参数个数。参数是可选的，也就是说函数也可以不包含参数。
* return\_types：返回类型，函数返回一列值。return\_types 是该列值的数据类型。有些功能不需要返回值，这种情况下 return\_types 不是必须的。
* 函数体：函数定义的代码集合。

##### 实例

以下实例为 max() 函数的代码，该函数传入两个整型参数 num1 和 num2，并返回这两个参数的最大值：

/\* 函数返回两个数的最大值 \*/

func max(num1, num2 int) int {

/\* 声明局部变量 \*/

var result int

if (num1 > num2) {

result = num1

} else {

result = num2

}

return result

}

#### 函数返回多个值

Go 函数可以返回多个值，例如：

package main

import "fmt"

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

return y, x

}

func main() {

a, b := swap("Mahesh", "Kumar")

fmt.Println(a, b)

}

#### 函数参数



<http://www.runoob.com/go/go-function-call-by-reference.html>

/\* 定义交换值函数\*/

func swap(x \*int, y \*int) {

var temp int

temp = \*x /\* 保持 x 地址上的值 \*/

\*x = \*y /\* 将 y 值赋给 x \*/

\*y = temp /\* 将 temp 值赋给 y \*/

}

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

var a int = 100

var b int= 200

fmt.Printf("交换前，a 的值 : %d\n", a )

fmt.Printf("交换前，b 的值 : %d\n", b )

/\* 调用 swap() 函数

\* &a 指向 a 指针，a 变量的地址

\* &b 指向 b 指针，b 变量的地址

\*/

swap(&a, &b)

fmt.Printf("交换后，a 的值 : %d\n", a )

fmt.Printf("交换后，b 的值 : %d\n", b )

}

func swap(x \*int, y \*int) {

var temp int

temp = \*x /\* 保存 x 地址上的值 \*/

\*x = \*y /\* 将 y 值赋给 x \*/

\*y = temp /\* 将 temp 值赋给 y \*/

}

#### 函数用法



##### 函数作为值

<http://www.runoob.com/go/go-function-as-values.html>

Go 语言可以很灵活的创建函数，并作为值使用。以下实例中我们在定义的函数中初始化一个变量，该函数仅仅是为了使用内置函数 math.sqrt() ，实例为：

package main

import (

"fmt"

"math"

)

func main(){

/\* 声明函数变量 \*/

getSquareRoot := func(x float64) float64 {

return math.Sqrt(x)

}

/\* 使用函数 \*/

fmt.Println(getSquareRoot(9))

}

以上代码执行结果为：

3

##### 闭包

Go 语言支持匿名函数，可作为闭包。匿名函数是一个"内联"语句或表达式。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

以下实例中，我们创建了函数 getSequence() ，返回另外一个函数。该函数的目的是在闭包中递增 i 变量，代码如下：

package main

import "fmt"

func getSequence() func() int {

i:=0

return func() int {

i+=1

return i

}

}

func main(){

/\* nextNumber 为一个函数，函数 i 为 0 \*/

nextNumber := getSequence()

/\* 调用 nextNumber 函数，i 变量自增 1 并返回 \*/

fmt.Println(nextNumber())

fmt.Println(nextNumber())

fmt.Println(nextNumber())

/\* 创建新的函数 nextNumber1，并查看结果 \*/

nextNumber1 := getSequence()

fmt.Println(nextNumber1())

fmt.Println(nextNumber1())

}

以上代码执行结果为：

1

2

3

1

2

###### 带参数的闭包函数调用:

package main

import "fmt"

func main() {

add\_func := add(1,2)

fmt.Println(add\_func())

fmt.Println(add\_func())

fmt.Println(add\_func())

}

// 闭包使用方法

func add(x1, x2 int) func()(int,int) {

i := 0

return func() (int,int){

i++

return i,x1+x2

}

}

###### 闭包带参数补充:

package main

import "fmt"

func main() {

add\_func := add(1,2)

fmt.Println(add\_func(1,1))

fmt.Println(add\_func(0,0))

fmt.Println(add\_func(2,2))

}

// 闭包使用方法

func add(x1, x2 int) func(x3 int,x4 int)(int,int,int) {

i := 0

return func(x3 int,x4 int) (int,int,int){

i++

return i,x1+x2,x3+x4

}

}

##### 函数方法

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。语法格式如下：

func (variable\_name variable\_data\_type) function\_name() [return\_type]{

/\* 函数体\*/

}

下面定义一个结构体类型和该类型的一个方法：

package main

import (

"fmt"

)

/\* 定义结构体 \*/

type Circle struct {

radius float64

}

func main() {

var c1 Circle

c1.radius = 10.00

fmt.Println("Area of Circle(c1) = ", c1.getArea())

}

//该 method 属于 Circle 类型对象中的方法

func (c Circle) getArea() float64 {

//c.radius 即为 Circle 类型对象中的属性

return 3.14 \* c.radius \* c.radius

}

以上代码执行结果为：

Area of Circle(c1) = 314

#### ★内建函数

　　close 只接受通道类型的值

　　len函数，可以应用于字符串、切片、数组、字典、通道类型

　　cap函数，可以应用于切片、数组、通道类型

　　new函数和make函数

　　append函数和copy函数，应用于切片

　　delete函数，根据字典的键删除某一项

　　complex\real\imag函数，复数相关

　　panic 函数，异常相关，它的参数一般是某个error接口的某个实现类型；recover 函数，不接受任何参数，返回 interface{} 类型，也就是意味着，它可以返回任意类型。recover返回的内容是与panic相关的，就是panic的参数内容。

print\println 函数，这两个函数支持基本类型参数，且是可变参数。但输出格式是固定的，且GO语言不保证以后会保留这两个函数，所以知道就好，不推荐使用。可以使用 fmt.Print 和 fmt.Println 来代替，效果更佳。

##### make()

内建函数 make 用来为 slice，map 或 chan 类型分配内存和初始化一个对象(注意：只能用在这三种类型上)，跟 new 类似，第一个参数也是一个类型而不是一个值，跟 new 不同的是，make 返回类型的引用而不是指针，而返回值也依赖于具体传入的类型

make()函数在golang的代码如下：

func make(t Type,size IntegerType) Type

使用make来创建slice，map，chanel说明如下：

* slice:

var slice\_ []int = make([]int,5,10)

fmt.Println(slice\_)

var slice\_1 []int = make([]int,5)

fmt.Println(slice\_1)

var slice\_2 []int = []int{1,2}

fmt.Println(slice\_2)

打印结果：

[0 0 0 0 0]

[0 0 0 0 0]

[1,2]

在创建slice时第一个参数用于确定初始化该slice的大小该slice中的值为零值，第三个参数用于确定该slice的长度

如果我们创建该slice时就能确定该slice最多只会装多大的数据量时就可以使用第三个参数固定该slice的长度，这样做的就会免去如果slice的数据超出内部所指向的array的最大长度时对array数据的转移，可以提高效率

* map：

var m\_ map[string]int = make(map[string]int)

m\_["one"] = 1

fmt.Println(m\_)

var m map[string]int = map[string]int{"1":1}

m["2"] = 2

fmt.Println(m)

打印结果：

map[one:1]

map[1:1 2:2]

根据 size 大小来初始化分配内存，不过分配后的 map 长度为 0，如果 size 被忽略了，那么会在初始化分配内存时分配一个小尺寸的内存

* chanel:

管道缓冲区依据缓冲区容量被初始化。如果容量为 0 或者忽略容量，管道是没有缓冲区的

#### ★可变参函数

　　最后一个参数为 ...T 的形式的函数即为可变参函数，意味着可变参数都是 T 类型（或实现了T的类型）如：func CallFunction(first string, t ...string)，GO语言会在调用可变参函数时，创建一个切片，然后将这些可变参数放入切片中，但如果传入的可变参部分就是一个元素类型为T的切片的话，则直接把传入切片赋值给创建的切片，且在调用写法上也有区别，为： CallFunction("hello", []string{"x","y"}...)

#### 函数与方法

　　GO语言中，函数跟方法是有区别的，函数就是我们通常理解的函数，而方法是附属于某个自定义数据类型的函数，即存在某个接收者。

　　func (self MyType) Len() int {}     这里的 (self MyInterface) 是表示方法接收者。

　　值方法和指针方法，值方法是指接收者是一个对象，而指针方法是指接收者是一个对象指针。两者的区别是，值方法中对接收者的改变是副本级别的，而指针方法中对接收者的改变是地址级别的。所以其实一般都推荐使用指针方法，因为大多数情况下我们在方法内部修改接收者，都是为了真实的改变它，而不是改变一个副本。但是，对于引用类型的接收者来说，两者并无区别。

　　匿名函数由函数字面量表示，函数是作为值存在的，可以赋给函数类型的变量。如：

var myfunc func(int, int) int

myfunc = func(x, y int) (result int) {

result = x + y

return

}

log.Println(myfunc(3, 4))

　　一个方法的类型是一个函数类型，即把接收者放到函数的第一个参数位置即可。

　　非常遗憾，GO语言不支持函数和方法重载。

### 错误异常处理

<http://www.runoob.com/go/go-error-handling.html>

Go 语言通过内置的错误接口提供了非常简单的错误处理机制。

error类型是一个接口类型，这是它的定义：

type error interface {

Error() string

}

我们可以在编码中通过实现 error 接口类型来生成错误信息。

函数通常在最后的返回值中返回错误信息。使用errors.New 可返回一个错误信息：

func Sqrt(f float64) (float64, error) {

if f < 0 {

return 0, errors.New("math: square root of negative number")

}

// 实现

}

在下面的例子中，我们在调用Sqrt的时候传递的一个负数，然后就得到了non-nil的error对象，将此对象与nil比较，结果为true，所以fmt.Println(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

result, err:= Sqrt(-1)

if err != nil {

fmt.Println(err)

}

实例

package main

import (

    "fmt"

)

// 定义一个 DivideError 结构

type DivideError struct {

    dividee int

    divider int

}

// 实现 `error` 接口

func (de \*DivideError) Error() string {

    strFormat := `

    Cannot proceed, the divider is zero.

    dividee: %d

    divider: 0

`

    return fmt.Sprintf(strFormat, de.dividee)

}

// 定义 `int` 类型除法运算的函数

func Divide(varDividee int, varDivider int) (result int, errorMsg string) {

    if varDivider == 0 {

        dData := DivideError{

            dividee: varDividee,

            divider: varDivider,

        }

        errorMsg = dData.Error()

        return

    } else {

        return varDividee / varDivider, ""

    }

}

func main() {

    // 正常情况

    if result, errorMsg := Divide(100, 10); errorMsg == "" {

        fmt.Println("100/10 = ", result)

    }

    // 当被除数为零的时候会返回错误信息

    if \_, errorMsg := Divide(100, 0); errorMsg != "" {

        fmt.Println("errorMsg is: ", errorMsg)

    }

}

执行以上程序，输出结果为：

100/10 = 10

errorMsg is:

    Cannot proceed, the divider is zero.

    dividee: 100

    divider: 0

### 日志