

**Go语言学习笔记**

­

**by 邹炜林**

**2017年4月10日**

目录

[二、 初识Go 3](#_Toc480134956)

[1.1 目的 3](#_Toc480134957)

[2.1 3](#_Toc480134958)

[三、 编程基础 3](#_Toc480134959)

[2.1 变量 3](#_Toc480134962)

[2.2 常量 4](#_Toc480134963)

[2.3 类型 4](#_Toc480134964)

[2.4 流程控制 6](#_Toc480134965)

[2.5 函数 6](#_Toc480134966)

[2.6 错误处理 7](#_Toc480134967)

[四、 面向对象编程 7](#_Toc480134968)

[五、 并发编程 7](#_Toc480134969)

[六、 网络编程 7](#_Toc480134970)

# 致谢

|  |  |  |
| --- | --- | --- |
| Go Web编程 | 《Go Web编程》，作者：谢孟军 |  |
|  |  |  |
|  |  |  |

# 初识Go

## 目的

Go playground 中的时间总是从 2009-11-10 23:00:00 UTC 开始

## 开发环境

LiteIDE、JetBrainsGoIDE <https://www.jetbrains.com/go/>

# 编程基础



## 变量

**var 语句定义了一个变量的列表**；跟函数的参数列表一样，类型在后面。就像在这个例子中看到的一样， var 语句可以定义在包或函数级别。

package main

import "fmt"

var c, pythor, java bool

func main(){

var i int

fmt.Println(i, c,pythor,java)

}

打印出0 false false false

**变量初始化**：

变量定义可以包含初始值，每个变量对应一个：

var x1, x2 int = 1 ,2；

如果初始化是使用表达式，则可以省略类型：

var str1 string = "str111"

var str2 = "str222"

变量从初始值中获得类型：

var str3, x3 , flag = "str333", 1, true

package main

import "fmt"

func main() {

fmt.Println(str1,str2,str3,x1,x2,x3,flag)

}

打印出：str111 str222 str333 1 2 1 true

**短声明变量**

在函数中， := 简洁赋值语句在明确类型的地方，可以用于替代 var 定义。

函数中这样写是错误的 x int = 3，必须这样写：var x int = 3或者 x := 3。

注意：函数外的每个语句都必须以关键字开始（ var 、 func 、等等）， := 结构不能使用在函数外。

**零值**

变量在定义时没有明确的初始化时会赋值为零值（和Java相似）。

零值是：

数值类型为 0 ，

布尔类型为 false ，

字符串为 "" （空字符串）。

**类型转换（强转）**

表达式 T(v) 将值 v 转换为类型 T 。

var i int = 42

var f float64 = float64(i)

var u uint = uint(f)

简单写法：

i := 42

f := float64(i)

u := uint(f)

**类型推导**

在定义一个变量却并不显式指定其类型时（使用 := 语法或者 var = 表达式语法）， 变量的类型由（等号）右侧的值推导得出。当右值定义了类型时，新变量的类型与其相同：

var i int

j := i // j 也是一个

但是当右边包含了未指名类型的数字常量时，新的变量就可能是 int 、 float64 或 complex128 。 这取决于常量的精度：

inti := 42 // int

f := 3.142 // float64

g := 0.867 + 0.5i // complex128

## 常量

常量的定义与变量类似，只不过使用 const 关键字。

常量可以是字符、字符串、布尔或数字类型的值。

常量不能使用 := 语法定义。

package main

import "fmt"

const Pi = 3.14

func main() {

const World = "世界"

fmt.Println("Hello", World)

fmt.Println("Happy", Pi, "Day")

const Truth = true

fmt.Println("Go rules?", Truth)

}

## 类型

Go 的基本类型有Basic types

bool

string

int int8 int16（-215~215-1） int32（-231~231-1） int64（-263~263-1）

uint uint8 uint16 uint32（0~232-1） uint64 uintptr

byte // uint8 的别名

rune // int32 的别名

// 代表一个Unicode码

float32 float64

complex64 complex128

int，uint 和 uintptr 类型在32位的系统上一般是32位，而在64位系统上是64位。当你需要使用一个整数类型时，你应该首选 int，仅当有特别的理由才使用定长整数类型或者无符号整数类型。

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **布尔型：**布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。 |
| 2 | **数字类型** 整型 int 和浮点型 float，Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。 |
| 3 | **字符串类型:** 字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。Go语言的字符串的字节使用UTF-8编码标识Unicode文本。 |
| 4 | **派生类型：**  (a) 指针类型（Pointer）  (b) 数组类型  (c) 结构化类型(struct)  (d) Channel 类型  (e) 函数类型  (f) 切片类型  (g) 接口类型（interface）  (h) Map 类型 |

**数字类型**

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **uint8** 无符号 8 位整型 (0 到 255) |
| 2 | **uint16** 无符号 16 位整型 (0 到 65535) |
| 3 | **uint32** 无符号 32 位整型 (0 到 4294967295) |
| 4 | **uint64** 无符号 64 位整型 (0 到 18446744073709551615) |
| 5 | **int8** 有符号 8 位整型 (-128 到 127) |
| 6 | **int16** 有符号 16 位整型 (-32768 到 32767) |
| 7 | **int32** 有符号 32 位整型 (-2147483648 到 2147483647) |
| 8 | **int64** 有符号 64 位整型 (-9223372036854775808 到 9223372036854775807) |

**浮点型**：

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **float32** IEEE-754 32位浮点型数 |
| 2 | **float64** IEEE-754 64位浮点型数 |
| 3 | **complex64** 32 位实数和虚数 |
| 4 | **complex128** 64 位实数和虚数 |

**其他数字类型**

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **byte** 类似 uint8 |
| 2 | **rune** 类似 int32 |
| 3 | **uint** 32 或 64 位 |
| 4 | **int** 与 uint 一样大小 |
| 5 | **uintptr** 无符号整型，用于存放一个指针 |

## 流程控制

for、if、else、switch和defer（延迟）

**for**

Go 只有一种循环结构—— for 循环。

基本的 for 循环包含三个由分号分开的组成部分：

初始化语句：在第一次循环执行前被执行

循环条件表达式：每轮迭代开始前被求值

后置语句：每轮迭代后被执行

初始化语句一般是一个短变量声明，这里声明的变量仅在整个 for 循环语句可见。

package main

import "fmt"

func main() {

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

fmt.Print(i, "\t")

}

}

打印结果是：0 1 2 3 4 5 6 7 8 9 999

**循环初始化语句和后置语句都是可选的**：

var i2 int = 1

for ; i2 < 10; { //（for 是 Go 的 “while”，所以 ; i2 < 10; 可写成i2 < 10 ）

i2 += i2

}

fmt.Println(i2) 打印结果是16

**死循环**

for{}

func main() {

var i int = 1

for {

i += i

if i > 10 {

break

}

}

fmt.Println(i)

}

**if**

就像 for 循环一样，Go 的 if 语句也不要求用 ( ) 将条件括起来，同时， { } 还是必须有的。

和for一样，if语句可以在条件之前执行一个简单语句。由这个语句定义的变量v的作用域仅在 if 范围之内。

func pow(x, n, lim float64) float64 {

if *v := math.Pow(x, n);* v < lim {

return v

}

return lim//如果此处返回v，将报错

}

同样的，v变量也可以在if{} else{}中的else里使用

switch

你可能已经知道 switch 语句会长什么样了。除非以 fallthrough 语句结束，否则分支会自动终止。

package main

import (

"fmt"

"runtime"

)

func main() {

fmt.Print("Go runs on ")

switch os := runtime.GOOS; os {

case "darwin":

fmt.Println("OS X.")

case "linux":

fmt.Println("Linux.")

default:

// freebsd, openbsd,

// plan9, windows...

fmt.Printf("%s.", os)

}

}打印的结果是Go run on windows.

**switch 的条件从上到下的执行，当匹配成功的时候停止。**（例如，当 i==0 时不会调用 f 。）

switch i {

case 0:

case f():

}

**没有条件的 switch**

没有条件的 switch 同 switch true 一样。

package main

import (

"fmt"

"time"

)

func main() {

t := time.Now()

switch {

case t.Hour() < 12:

fmt.Println("Good morning!")

case t.Hour() < 17:

fmt.Println("Good afternoon.")

default:

fmt.Println("Good evening.")

}

}

defer 延迟函数

defer 语句会延迟函数的执行直到上层函数返回。延迟调用的参数会立刻生成，但是在上层函数返回前函数都不会被调用。

package main

import "fmt"

func main() {

defer fmt.Print("world")

fmt.Print("hello ")

}//打印出 hello world

defer 栈

延迟函数的调用被压入一个栈中，且遵循后进先出原则

func main() {

defer fmt.Print("1")

defer fmt.Print("2")

}//打印出 21

## 函数

函数可以没有参数或接受多个参数。在这个例子中， add 接受两个 int 类型的参数。注意类型在变量名之后。

package main

import "fmt"

func add(x int, y int) int {

return x + y

}

func main() {

fmt.Println(add(42, 13))

}

x int, y int 和x, y int这两种写法是等价的

函数的返回是可以多值返回的。

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)

}

结果打印出：world hello

命名返回值

// 命名返回值，return没有参数，将返回各个变量的当前值

func split(sum int) (x,y int) {

x = sum\*4/2

y = x-4

return

}

fmt.Println(split(10))打印出20 16

## 指针

我们都知道，变量是一种使用方便的占位符，用于引用计算机内存地址。Go 语言的取地址符是 &，放到一个变量前使用就会返回相应变量的内存地址。

package main

import "fmt"

func main() {

var a int = 10

fmt.Printf("变量的地址: %x\n", &a )

}// 打印出 变量的地址:20818a220

指针是一个变量，一个指针变量可以指向任何一个值的内存地址~~它指向那个值的内存地址~~。类似于变量和常量，在使用指针前你需要声明指针。指针声明格式如下：

var var\_name \*var-type

var-type 为指针类型，var\_name 为指针变量名，\* 号用于指定变量是作为一个指针。以下是有效的指针声明：

var ip \*int /\* 指向整型\*/

var fp \*float32 /\* 指向浮点型 \*/

**指针使用流程：**

1、定义指针变量；2、为指针变量赋值；3、访问指针变量中指向地址的值；4、在指针类型前面加上 \* 号（前缀）来获取指针所指向的内容。

package main

import "fmt"

func main() {

var a int= 20 /\* 声明实际变量 \*/

var ip \*int /\* 声明指针变量 \*/

ip = &a /\* 指针变量的存储地址 \*/

fmt.Printf("a 变量的地址是: %x\n", &a )

/\* 指针变量的存储地址 \*/

fmt.Printf("ip 变量储存的指针地址: %x\n", ip )

/\* 使用指针访问值 \*/

fmt.Printf("\*ip 变量的值: %d\n", \*ip )

}// 打印如下

*a 变量的地址是: 20818a220*

*ip 变量储存的指针地址: 20818a220*

*\*ip 变量的值: 20*

**空指针**

当一个指针被定义却没有被赋值时，称之为空指针，用nil表示。nil概念上和其他语言的null、None、NULL一样，都指代零值或空值。

一个指针变量通常用ptr表示

package main

import "fmt"

func main() {

var ptr \*int

fmt.Printf("ptr 的值为 : %x\n", ptr )

fmt.Println("ptr:",ptr)

}//打印如下

ptr:0

ptr: <nil>

**关于指针更多内容：**

[指针数组](http://www.runoob.com/go/go-array-of-pointers.html)：const MAX int = 3，var ptr [MAX]\*int;

[指针的指针](http://www.runoob.com/go/go-pointer-to-pointer.html)：如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。声明格式：var ptr \*\*int;

[向函数传递指针](http://www.runoob.com/go/go-passing-pointers-to-functions.html)

**指针练习：**

func main() {

i, j := 42, 99

var p \*int= &i // point to i // 把i的内存地址给p

// 另一种写法 p := &i

fmt.Println("p=",p," &i=",&i)//打印出p= 0x10b92140 &i= 0x10b92140

fmt.Println("\*p=",\*p) // read i through the pointer // 读取p指针指向底层的值

\*p = 21 // set i through the pointer // 把21赋给p指针指向底层的值

fmt.Println("i=",i) // see the new value of i // 打印i的值

p = &j // point to j //把j的指针赋给p

fmt.Println("p=",p," &j=",&j)

\*p = \*p / 33 // divide j through the pointer // 获取指针p指向底层的值，然后除以33，把结果赋给底层的值

fmt.Println("\*p=",\*p)

fmt.Println("j=",j," &j=",&j) // see the new value of j // 因为底层的值被修改了，所以j的指针指向底层的值为3

var a int = 1

var b \*int = &a

var c \*\*int = &b

var x int = \*b

fmt.Println("a = ",a)

fmt.Println("&a = ",&a)

fmt.Println("\*&a = ",\*&a)

fmt.Println("b = ",b)

fmt.Println("&b = ",&b)

fmt.Println("\*&b = ",\*&b)

fmt.Println("\*b = ",\*b)

fmt.Println("c = ",c)

fmt.Println("\*c = ",\*c)

fmt.Println("&c = ",&c)

fmt.Println("\*&c = ",\*&c)

fmt.Println("\*\*c = ",\*\*c)

fmt.Println("\*\*\*&\*&\*&\*&c = ",\*\*\*&\*&\*&\*&\*&c)

fmt.Println("x = ",x)

var ptr \*int

fmt.Printf("ptr:%x\n",ptr)

fmt.Println("ptr:",ptr)

}

## 结构体

Go语言中数组存放同一类型的数据，而结构体可以存放类型相同或者不同的数据。一个结构体（ struct ）就是一个字段的集合。结构体的定义必须有type 和 struct 语句（也可以用Java去理解为一个静态的类，内含静态变量，同时也含有和继承相似的功能，因为在Go里没有继承），下面是简单示例：

import "fmt"

// 定义一个结构体，命名为Parent

type Parent struct {

mother string// 字段一为string类型

father string// 字段二为string类型

}

// 定义一个结构体，命名为Me

type Me struct {

age int // 字段一为int类型

name string // 字段二为string类型

gender string // 字段三为string类型

parents Parent // 字段四为结构体Parent类型，类似“继承了Parent”的属性

}

func main() {

var me Me

me.age = 20

me.name = "小李"

me.gender ="男"

me.parents.father = "李雷"

me.parents.mother="韩梅梅"

fmt.Printf("age:%d \tname:%s \tgender:%s \tfather:%s \tmother:%s",me.age,me.name,me.gender,me.parents.father,me.parents.mother)

fmt.Println()

// 当然，你也可以这样赋值

me2 := Me{38,"李雷","男",Parent{"牛爱花","李刚"}}

fmt.Printf("age:%d \tname:%s \tgender:%s \tfather:%s \tmother:%s",me2.age,me2.name,me2.gender,me2.parents.father,me2.parents.mother)

}

// 打印结果如下：

age:20 name:小李 gender:男 father:李雷 mother:韩梅梅

age:38 name:李雷 gender:男 father:李刚 mother:牛爱花

**结构体可以作为函数的参数**

**结构体也有指针，使用方法如下：**

var me\_ptr \*Me // 什么结构体指针

me\_ptr = &me // 把me的指针传递给me\_ptr

printMe(me\_ptr) // 通过结构体指针获取到指针对应的结构体的字段

func printMe(me \*Me) {

fmt.Println("我是：",me.name)

fmt.Printf("爹是：%s\t娘是：%s\n",me.parents.father,me.parents.mother)

}

// 打印结果如下：

我是： 小李

爹是：李雷 娘是：韩梅梅

**结构体文法**

结构体文法表示通过结构体字段的值作为列表来新分配一个结构体。使用 Name: 语法可以仅列出部分字段。（字段名的顺序无关。）特殊的前缀 & 返回一个指向结构体的指针。

import "fmt"

type Vertex struct {

X, Y int

}

var (

v1 = Vertex{1, 2} // 类型为 Vertex

v2 = Vertex{X: 1} // Y:0 被省略

v3 = Vertex{} // X:0 和 Y:0

v4 = Vertex{Y: 2, X: 4}

p = &Vertex{1, 2} // 类型为 \*Vertex

)

func main() {

fmt.Println(v1, p, v2, v3, v4)

}

//打印出 {1 2} &{1 2} {1 0} {0 0} {4 2}

## 数组

数组是具有相同唯一类型的一组已编号且长度固定的数据项序列，这种类型可以是任意的原始类型例如整形、字符串或者自定义类型。

结构：var variable\_name [SIZE] variable\_type

使用：var a [10]int，定义变量 a 是一个有十个整数的数组。

数组的长度是其类型的一部分，因此数组不能改变大小。 这看起来是一个制约，但是请不要担心； Go 提供了更加便利的方式来使用数组。

import "fmt"

func main() {

var a [2]string

a[0] = "Hello"

a[1] = "World"

fmt.Println(a[0], a[1])

fmt.Println(a)

}

// 打印出

Hello World

[Hello World]

**初始化数组**

var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}，初始化数组中 {} 中的元素个数不能大于 [] 中的数字。

如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小，如下：

var balance = [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}，“...”可以不写，此处默认等于5。

示例：

import "fmt"

func main() {

var array [4]int // 定义一个长度为4的int类型数组

fmt.Println(array) // 打印结果[0 0 0 0]

array[0] = 1 // 给第数组索引为1的元素赋值为1

fmt.Println(array[0]) // 打印结果为1

var flag = []bool{true,false,true} // 定义一个长度为3的bool类型数组

fmt.Println(flag) // 打印结果为[true false true]

}

## Slice（片）名词

Go语言中，Slice是对数组的抽象。Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组"),与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

声明一个切片：var identifier []type，或使用var slice1 []type = make([]type, len)创建切片，亦或slice1 := make([]type, len)，这里的len 是切片的初识长度

func main(){

s1 := []int{2,4,6,8} //声明一个有限int类型的数组，长度为4,

s2 := []int{1,2,3,4,5,6,7,8,9,0} //声明一个有限int类型数组，长度为10

fmt.Println(s1) // 内容为：[2 4 6 8]

// 切片结构，type表示切片类型，length表示切片当前长度，capacity表示容量（可选参数），即使设定，也会在切片(动词)后被变更

// 结构式：var var\_name []type = make([]type, length, capacity)

var ss []int = make([]int,3,100) // 声明一个长度为3，容量为100的切片

// len(切片)表示切片的长度，cap(切片)表示切片最长可达到的容量

fmt.Printf("len:%d\tcap:%d\tcontent:%v\n",len(ss),cap(ss),ss) // len:3 cap:100 content:[0 0 0]

ss = s1[3:4] // 从s1中截取索引3~4（不含）的元素创建为一个新的切片

fmt.Printf("len:%d\tcap:%d\tcontent:%v\n",len(ss),cap(ss),ss) // len:1 cap:1 content:[8]

ss = s1[1:3] // 从s1中截取索引1~3（不含）的元素创建为一个新的切片

fmt.Printf("len:%d\tcap:%d\tcontent:%v\n",len(ss),cap(ss),ss) // len:2 cap:3 content:[4 6]

ss = s2[1:3] // 从s2中截取索引1~3（不含）的元素穿件为一个新的切片

fmt.Printf("len:%d\tcap:%d\tcontent:%v\n",len(ss),cap(ss),ss) // len:2 cap:9 content:[2 3]

// 由此可见，切片ss是一个长度可变，最大容量根据数组源的长度而改变的“动态数组”,切cap由源数组len-startIndex

}

slice的零值是nil，slice的长度和容量是0。

**向切片添加元素**

import "fmt"

func main() {

var a []int

printSlice("a", a)

a = append(a, 0)// append works on nil slices.

printSlice("a", a)

a = append(a, 1)// the slice grows as needed.

printSlice("a", a)

a = append(a, 2,3,4)// we can add more than one element at a time.

printSlice("a", a)

}

func printSlice(s string, x []int) {

fmt.Printf("%s len=%d cap=%d %v\n",

s, len(x), cap(x), x)

}

// 打印结果

a len=0 cap=0 []

a len=1 cap=2 [0]

a len=2 cap=2 [0 1]

a len=5 cap=8 [0 1 2 3 4]

**range**

for循环的range格式可以slice或map进行迭代循环。当使用 for 循环遍历一个 slice 时，每次迭代 range 将返回两个值。 第一个是当前下标（序号），第二个是该下标所对应元素的一个拷贝。

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)

}

}

// 打印结果

2\*\*0 = 1

2\*\*1 = 2

2\*\*2 = 4

2\*\*3 = 8

2\*\*4 = 16

2\*\*5 = 32

2\*\*6 = 64

2\*\*7 = 128

for index := range pow { // 循环打印出切片的索引

fmt.Print(index)

}

for \_, value := range pow { // 循环打印出切片索引所对应的值

fmt.Print("\t", value)

}

// 打印结果是：

关于切片更多的知识：01234567 1 2 4 8 16 32 64 128

<https://blog.go-zh.org/go-slices-usage-and-internals>

## Map

Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的。

**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的类型]value的类型

例如 var m map[string]int ，此时m 是nil，如果赋值 m[“test”]=1，会直接奔溃掉，报错panic: assignment to entry in nil map

**初始化**一个可使用的map：两种方式

1. var m = map[string]int{}（快速赋值var m = map[string]int{“test1”:1, “test2”:2} ）
2. 先声明var m map[string]int 然后使用make()函数初始化：m = make(map[string]int)

之后赋值 m[“test”]=1就不会报错了

**map数据操作**：

map的**增改**：m[“test”]=1

map的**删**：delete(m,”test”)

map的**查**：a := m[“test”]

**map键值**

map[key]value的key和value都适用构造体

type KEY struct {

p1 int

}

type VALUE struct {

value1 int

value2 string

}

var m1 = map[KEY]VALUE{KEY{1}: VALUE{1,"content"}}

func main(){

fmt.Print(m1)

}

// 打印结果是 map[{1}:{1 content}]

**遍历map**

m4 := map[string]int{"test1": 1, "test4": 4, "test2": 2}

for key := range m4 {

fmt.Println("key=", key, "\tvalue=", m4[key])

}

// 打印的结果是无序的

key= test2 value= 2

key= test1 value= 1

key= test4 value= 4

**查看不存在的元素**

test3Value,ok := m4["test3"]

fmt.Println(test3Value, ok)

// 打印结果：0 false

test3Value是map的value的零值（零值：0，nil，null，NULL等）

## 函数“值”

函数也可以作为值来进行传递、赋值。函数值称为匿名函数

var a int // 变量a，变量值为0

func myFunc()

函数的闭包

## 方法

Go 没有类。然而，仍然可以在结构体类型上定义方法。**方法接收者**出现在 func 关键字和方法名之间的参数中。

type Bird struct {

name string

color string

}

func (b \*Bird) fly() {

// 这里的方法接收者是Bird

fmt.Println(b.color+"的"+ b.name + "在飞")

}

func main() {

var bird = new(Bird)

bird.name="燕子"

bird.color="黑色"

bird.fly()

bird2 := &Bird{"喜鹊","灰色"}//通过指针

bird2.fly()

}

// 打印出

黑色的燕子在飞

灰色的喜鹊在飞

**接收者为指针的方法**

方法可以与命名类型或命名类型的指针关联。

func (n Num) Add() {

fmt.Println(n.a + n.b)

}

// 【方法接收者】为类型指针的方法

func (n \*Num) ChangeNum() {

// 计算后，n.a的值改变

n.a = n.a \* n.a

n.b = n.b \* n.b

}

// 【方法接收者】为类型的方法

func (n Num) ChangeNum2() {

// 计算后，n.a的值不会改变

n.a = n.a \* n.a

n.b = n.b \* n.b

}

func main() {

n := Num{2, 4}

n.Add()

fmt.Println(n)

n.ChangeNum()

fmt.Println(n)

n.ChangeNum2()

fmt.Println(n)

}

// 打印结果是：

6

{2 4}

{4 16}

{4 16}

## 接口

import "fmt"

// 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。

// 1、定义一个人的接口，人的行为共性会说话

type IPerson interface {

speak()

//walk()

}

// 2、定义一个人，李雷

type LiLei struct {

}

// 3、写一个speak方法，方法接收者是type LiLei，说明LiLei实现了这个IPerson接口

func (l LiLei) speak() {

fmt.Println("Hi, I`m LiLei")

}

/\*func (l LiLei) walk() {

fmt.Println("I can walk")

}\*/

func main() {

var person IPerson // a、声明一个接口

person = new(LiLei) // b、new一个李雷来实现person接口，如果我们没有写3这一步，我们是不能这样写的

person.speak() // c、执行者其实是我们的LiLei，这很像Java中的多态

// 打印结果：Hi! I`m LiLei

// 4、此时在IPersion接口内再建立一个walk()方法，b处马上出现提醒：

// Cannot use new(LiLei) (type \*LiLei) as type IPerson in assignment：不能通过new(LiLei)作为IPerson类型

// 5、因为我们的type LiLei没有实现walk()方法，所以Go认为，LiLei没有实现IPerson接口

// 6、只要我们补上一个方法，接收者为LiLei就可以了

//person.walk()

}

**隐式接口**

## Stringers

一个普遍存在的接口是 fmt 包中定义的 Stringer。

type Stringer interface {

String() string

}

Stringer 是一个可以用字符串描述自己的类型。`fmt`包 （还有许多其他包）使用这个来进行输出。

type Dog struct {

name string

age int

}

func (d Dog) String() string {

return fmt.Sprintf("这条狗的名字是：%s，今年%d岁了。", d.name, d.age)

}

func main() {

dog1 := Dog{"大黄", 8}

var dog2 = new(Dog)

dog2.name = "二狗子"

dog2.age = 2

fmt.Println(dog1,dog2)

}

// 打印结果：

这条狗的名字是：大黄，今年8岁了。 这条狗的名字是：二狗子，今年2岁了。

**例子**：让 IPAddr 类型实现 fmt.Stringer 以便用点分格式输出地址。例如，IPAddr{1, 2, 3, 4} 应当输出 "1.2.3.4"。

import "fmt"

type IPAddr [4]byte

// TODO: Add a "String() string" method to IPAddr.

func (a IPAddr) String() string{

return fmt.Sprintf("%d.%d.%d.%d",a[0],a[1],a[2],a[3])

}

func main() {

addrs := map[string]IPAddr{

"loopback": {127, 0, 0, 1},

"googleDNS": {8, 8, 8, 8},

}

for n, a := range addrs {

fmt.Printf("%v: %v\n", n, a)

}

}

## 错误处理

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

}

**例子：**

import "fmt"

/\*\*

error

Go 已经定义了一个error接口，其有一个Error()方法，我们只需要实现Error()这个方法即可

\*/

type DivideError struct {

a, b int

}

func (div DivideError) division() (result int, errorMsg string) {

if div.b == 0 {

errorMsg = div.Error()

} else {

result = div.a / div.b

}

return // 这里默认返回变量最后的值，如果变量没有被赋值则默认为空值，string空值为""

}

// DivideError实现了error接口

func (div DivideError) Error() string {

return fmt.Sprintf("b:%d不能为零。", div.b)

}

func main() {

d := DivideError{10, 5}

if result, errorMsg := d.division(); errorMsg == "" {

fmt.Println(result)

}

d2 := DivideError{10,0}

if \_,errorMsg :=d2.division();errorMsg!="" {

// 因为division()是多返回值函数，但进入到此肯定发生了错误，所以我不在乎返回的result，使用“\_”表示不关心结果

fmt.Println(errorMsg)

}

}

// 打印结果：

2

b:0不能为零。

# 输入输出

## 格式化IO

fmt的使用

## 基本IO接口

从IO接口中读取信息，字符串读取简单例子：

**import** (

    "fmt"

    "io"

    "os"

    "strings"

)

/\*

Go的IO接口一些简单读取函数

\*/

**func** ReadFrom(reader io.Reader, num int) ([]byte, error) {

    p := make([]byte, num) // 缓冲buffer，生成一个切片

    n, err := reader.Read(p) // 把数据读到切片中，n表示读了多少数据，err表示可能存在的错误

**if** n > 0 {

**return** p[:n], nil

    }

**return** p, err // 为什么读取到n呢，因为buffer的长度可能大于我们给的字符的长度，所以通过切片的特性只返回我们读取到的位置

}

// 从字符串读取

**func** sampleReadFromString() {

    // 通过给定的字符串，建立一个读取器

    data, \_ := ReadFrom(strings.NewReader("from string"), 12)

    fmt.Println(data)

}

func main() {

sampleReadFromString()

}

// 打印出[102 114 111 109 32 115 116 114 105 110 103]，byte数组的切片

**例子2**

// 从控制台输入读取字符串,使用IDE是无法打印出结果的，必须通过命令行来编译出xxx.exe文件，然后通过命令行执行，方式：在该go文件所在文件件，按住shift+鼠标右键，选择在此处打开命令窗口，输入go build 【文件名】.go，编译生成【文件名】.exe文件，把文件拖到命令窗口运行，然后输入字符即可看到我们截取的字符串了。

**func** sampleReadFromStdin() {

    data, \_ := ReadFrom(os.Stdin, 12)

    fmt.Println(data) // 读取到的字节

    fmt.Println(string(data)) // 字节强转为字符查看我们到底读取到了什么

}

func main(){

sampleReadFromStdin()

}

// 编译build，然后用cmd打开该可执行文件，输入字符串即可看到从终端读取到的数据

// D:\Go\workplace\dev\src\basiceio>basiceio.exe

// 12345678901234567890num // 在平台输入的字符

// [49 50 51 52 53 54 55 56 57 48 49 50] // 这是切片读取到的字节

// 123456789012 //这是把切片字节强转为字符的结果，也就是从1读取到了012

**例子3**

// 从文件读取字符串

**func** sampleReadFromFile(longth int) {

// 这里如果你用的是JetBrains Gogland IDE，可能文件读不出来，可以使用LiteIDE进行测试

    file, \_ := os.Open("main.go") // 打开我们的源码

**defer** file.Close() // 读取完毕后关闭文件流

    data, \_ := ReadFrom(file, longth) // 读取我们的文件

    fmt.Println(data) // 打印数组切片

    fmt.Println(string(data)) // 打印数组切片强转字符串结果

}

**func** main() {

    sampleReadFromFile(9)

    sampleReadFromFile(10)

}

// 打印结果

[112 97 99 107 97 103 101 32 109]

package m

[112 97 99 107 97 103 101 32 109 97]

package ma

以上就是Go语言中比较简单读取不同数据源字符串的IO操作。以上几个IO读取例子来源于：<http://www.imooc.com/learn/492> 的视频教程

# 面向对象编程

# 并发编程

## goroutine

goroutine 是由 Go 运行时环境管理的轻量级线程。

var a =0

func say(s string) {

for i := 0; i < 5; i++ {

a += 1

time.Sleep(100 \* time.Millisecond)

fmt.Println(s, "\t a=", a)

}

}

func main() {

go say("world")

say("hello")

}

## channel

通道

缓冲channel：channel 可以是带缓冲的。为 make 提供第二个参数作为缓冲长度来初始化一个缓冲 channel：

c := make(chan string,2)，生成一个缓冲区为2条的通道

func main() {

c := make(chan string,2)

c <- "我是1通道"

c <- "我是2通道"

//c <- "我是3通道" // 打开的话就发生all goroutines are asleep - deadlock!

fmt.Println(<-c)

fmt.Println(<-c)

}

# 网络编程

IP地址和域名解析

在net包中，IP地址类型被定义成一个byte数组，即若干个8位组

# 扩展

常见的格式化打印符号：fmt.Printf(“打印了一个int%d”,2)，打印结果“打印了一个2”

%d表示输入整数型

%f 表示浮点型

%s表示输入字符串

%v表示一种类型，自动识别类型