Go语言学习(四)常用类型介绍

原创 2016年03月26日 14:28:29 标签：go语言 2484

1.布尔类型

var v1 bool

v1 = true;

v2 := (1==2) // v2也会被推导为bool类型

1

2

3

4

2.整型

类 型 长度（字节） 值 范 围

int8 1  -128 ~ 127

uint8（即byte）1 0 ~ 255

int16 2  -32 768 ~ 32 767

uint16 2 0 ~ 65 535

int32 4  -2 147 483 648 ~ 2 147 483 647

uint32 4 0 ~ 4 294 967 295

int64 8  -9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807

uint64 8 0 ~ 18 446 744 073 709 551 615

int 平台相关 平台相关

uint 平台相关 平台相关

uintptr 同指针 在32位平台下为4字节，64位平台下为8字节

var value2 int32

value1 := 64 // value1将会被自动推导为int类型

value2 =value1 //编译失败:cannot use value1 (type int) as type int32 in assignment。

//使用强制类型转换可解决

value2 = int32(value1)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

3.数值运算

常规运算:+、-、\*、/和%

比较运算: > 、 < 、 == 、 >= 、 <= 和 !=

需要注意:两个不同类型的整型数不能直接比较，比如 int8 类型的数和 int类型的数不能直接比较，但各种

类型的整型变量都可以直接与字面常量（literal）进行比较，比如：

1

2

3

4

5

var i int32

var j int64

i, j = 1, 2

if i == j { // 编译错误

fmt.Println("i and j are equal.")

}

if i == 1 || j == 2 { // 编译通过

fmt.Println("i and j are equal.")

}

1

2

3

4

5

6

7

8

9

4.位运算

x << y 左移

x >> y 右移

x ^ y 异或

x & y 与

x | y 或

^x 取反

1

2

3

4

5

6

7

5.浮点类型

Go语言定义了两个类型 float32 和 float64 ，其中 float32 等价于C语言的 float 类型，

float64 等价于C语言的 double 类型。

var fvalue1 float32

fvalue1 = 12

fvalue2 := 12.0 // 如果不加小数点，fvalue2会被推导为整型而不是浮点型

对于以上例子中类型被自动推导的 fvalue2 ，需要注意的是其类型将被自动设为 float64 ，

而不管赋给它的数字是否是用32位长度表示的。因此，对于以上的例子，下面的赋值将导致编译

错误：

fvalue1 = fvalue2

而必须使用这样的强制类型转换：

fvalue1 = float32(fvalue2)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

5.1浮点数比较

因为浮点数不是一种精确的表达方式，所以像整型那样直接用 == 来判断两个浮点数是否相等

是不可行的，这可能会导致不稳定的结果。

下面是一种推荐的替代方案：

1

2

3

4

import "math"

// p为用户自定义的比较精度，比如0.00001

func IsEqual(f1, f2, p float64) bool {

return math.Fdim(f1, f2) < p

}

1

2

3

4

5

6.复数

复数实际上由两个实数（在计算机中用浮点数表示）构成，一个表示实部（real），一个表示

虚部（imag）。如果了解了数学上的复数是怎么回事，那么Go语言的复数就非常容易理解了。

1

2

3

6.1复数表示

var value1 complex64 // 由2个float32构成的复数类型

value1 = 3.2 + 12i

value2 := 3.2 + 12i // value2是complex128类型

value3 := complex(3.2, 12) // value3结果同 value2

1

2

3

4

5

6.2 实部与虚部

对于一个复数 z = complex(x, y) ，就可以通过Go语言内置函数 real(z) 获得该复数的实

部，也就是 x ，通过 imag(z) 获得该复数的虚部，也就是 y 。

1

2

3

7.字符串

Go语言中字符串的声明和初始化非常简单，举例如下：

1

2

var str string // 声明一个字符串变量

str = "Hello world" // 字符串赋值

ch := str[0] // 取字符串的第一个字符

fmt.Printf("The length of \"%s\" is %d \n", str, len(str))

fmt.Printf("The first character of \"%s\" is %c.\n", str, ch)

1

2

3

4

5

输出结果为：

The length of "Hello world" is 11

The first character of "Hello world" is H.

1

2

3

4

字符串的内容可以用类似于数组下标的方式获取，但与数组不同，字符串的内容不能在初始

化后被修改，比如以下的例子：

str := "Hello world" // 字符串也支持声明时进行初始化的做法

str[0] = 'X' // 编译错误

编译器会报类似如下的错误：

cannot assign to str[0]

如果想修改字符串的内容,可以通过如下的方式修改:

1

2

3

4

5

6

7

8

s := "hello"

c := []rune(s) //将字符串s转成rune数组

c[0] = 'x' //修改rune数组中的第一个元素为x

s2 := string(c) //根据rune数组创建一个新的字符串

fmt.Println("s2=",s2)

1

2

3

4

5

运行结果:

s2= xello

1

2

3

7.1字符串操作

x + y 字符串连接 "Hello" + "123" // 结果为Hello123

len(s) 字符串长度 len("Hello") // 结果为5

s[i] 取字符 "Hello" [1] // 结果为'e'

1

2

3

4

7.2多行字符串

使用多行字符串需要小心,例如:

s := "Hello "

+"world"

编译时会被自动转换成:

s := "Hello ";

+"world";

此时会编译失败,报错:invalid operation: + untyped string

正确的多行写法是这样的:

s := "Hello " +

"world"

Go 就不会在错误的地方插入分号。另一种写法是使用反引号 ` 作为原始字符串符号：

s := `Hello

world`

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

7.3字符串遍历

Go语言支持两种方式遍历字符串。一种是以字节数组的方式遍历：

1

2

str := "abc中国"

n := len(str)

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

ch := str[i] // 依据下标取字符串中的字符，类型为byte

fmt.Printf("i=%v\n", ch)

}

1

2

3

4

5

6

输出结果:

i=97 //a

i=98 //b

i=99 //c

i=228 //中字的第一字节,每个中文字符在UTF-8中占3个字节

i=184 //中字的第二个字节

i=173 //中字的第三个字节

i=229 //国字的第一个字节

i=155 //国字的第二个字节

i=189 //国字的第三个字节

1

2

3

4

5

6

7

8

9

10

11

另一种是以Unicode字符遍历,以Unicode字符方式遍历时，每个字符的类型是 rune

（早期的Go语言用 int 类型表示Unicode字符），而不是byte.

1

2

3

str := "abc中国"

for i, ch := range str{ //range关键字在数组中有介绍

fmt.Println(i,ch) //ch的类型为rune

}

1

2

3

4

//输出结果:

0 97

1 98

2 99

3 20013

6 22269

1

2

3

4

5

6

7

8.字符类型

在Go语言中支持两个字符类型，一个是 byte （实际上是 uint8 的别名），代表UTF-8字符串的单个字节的值；

另一个是 rune ，代表单个Unicode字符。出于简化语言的考虑，Go语言的多数API都假设字符串为UTF-8编码。

尽管Unicode字符在标准库中有支持，但实际上较少使用。

1

2

3

4

9.数组

以下为一些常规的数组声明方法：

[32]byte // 长度为32的数组，每个元素为一个字节

[2\*N] struct { x, y int32 } // 复杂类型数组

[1000]\*float64 // 指针数组

[3][5]int // 二维数组

[2][2][2]float64 // 等同于[2]([2]([2]float64))

在Go语言中，数组长度在定义后就不可更改，在声明时长度可以为一个常量或

者一个常量表达式（常量表达式是指在编译期即可计算结果的表达式）。

1

2

3

4

5

6

7

8

9

10

9.1元素的访问

可以使用数组下标来访问数组中的元素。与C语言相同，数组下标从0到len(array)-1,

下面的示例遍历整型数组并逐个打印元素内容：

1

2

3

//通过for循环遍历

for i := 0; i < len(array); i++ {

fmt.Println("Element", i, "of array is", array[i])

}

//Go语言还提供了一个关键字range,用于便捷地遍历容器中的元素。

//当然，数组也是range的支持范围。上面的遍历过程可以简化为如下的写法：

for k, v := range array { //k相当于索引,v相当于值

fmt.Println("Array element[", k, "]=", v)

}

1

2

3

4

5

6

7

8

9

9.2值操作

需要特别注意的是，在Go语言中数组是一个值类型（value type）。所有的值类型变量在赋值

和作为参数传递时都将产生一次复制动作。如果将数组作为函数的参数类型，则在函数调用时该

参数将发生数据复制。因此，在函数体中无法修改传入的数组的内容，因为函数内操作的只是所

传入数组的一个副本。下面用例子来说明这一特点：

1

2

3

4

5

func main(){

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

modify(array)

fmt.Println("In main(),array values:",array)

}

func modify(array [5] int){

array[0] = 10

fmt.Println("In modify(),array values:",array)

}

1

2

3

4

5

6

7

8

9

10

//输出结果:

In modify(), array values: [10 2 3 4 5]

In main(), array values: [1 2 3 4 5]

1

2

3

4

从执行结果可以看出，函数 modify() 内操作的那个数组跟 main() 中传入的数组是两个不同的实

例。那么，如何才能在函数内操作外部的数据结构呢？这个就要用到后接下来要讲的数组切片.

1

2

3

10.数组切片

在前一节里我们已经提过数组的特点：数组的长度在定义之后无法再次修改；数组是值类型，

每次传递都将产生一份副本。显然这种数据结构无法完全满足开发者的真实需求。

Go语言提供了数组切片（slice）这个非常酷的功能来弥补数组的不足,可以随时动态扩充存放空

间，并且可以被随意传递而不会导致所管理的元素被重复复制。有点类似于java中的集合.

数组切片的数据结构可以抽象为以下3个变量：

一个指向原生数组的指针；

数组切片中的元素个数；

数组切片已分配的存储空间。

1

2

3

4

5

6

7

8

9

10

10.1创建数组切片

(1)基于已存在数组创建,数组切片可以只使用数组的一部分元素或者整个数组来创建，

甚至可以创建一个比所基于的数组还要大的数组切片。

1

2

3

var myArray [10]int = [10]int{1,2,3,4,5,6,7,8,9,10}

// 基于数组前5个元素创建一个数组切片

var mySlice []int = myArray[:5]

fmt.Println("Elements of myArray:")

for \_, v := range myArray{

fmt.Print(v," ")

}

fmt.Println("\nElements of mySlice:")

for \_, v:= range mySlice{

fmt.Print(v," ")

}

1

2

3

4

5

6

7

8

9

10

11

12

13

//输出结果:

Elements of myArray:

1 2 3 4 5 6 7 8 9 10

Elements of mySlice:

1 2 3 4 5

1

2

3

4

5

6

Go语言支持用 myArray[first:last] 这样的方式来基于数组生成一

个数组切片，而且这个用法还很灵活，比如下面几种都是合法的。

a.基于 myArray 的所有元素创建数组切片：

mySlice = myArray[:]

b.基于 myArray 的前5个元素创建数组切片：

mySlice = myArray[:5]

c.基于从第5个元素开始的所有元素创建数组切片：

mySlice = myArray[5:]

(2)通过make()函数直接创建数组切片

创建一个初始元素个数为5的数组切片,元素初始值为0:

mySlice1 := make([]int,5)

创建一个初始元素个数为5的数组切片,元素初始值为0,并预留10个元素的存储空间:

mySlice2 := make([]int,5,10)

通过cap()函数可以获取切片的分配空间,len()函数可以知道当前已存储的元素个数.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

fmt.Println("len(mySlice2):", len(mySlice2))

fmt.Println("cap(mySlice2):", cap(mySlice2))

1

2

//输出结果:

len(mySlice2): 5

cap(mySlice2): 10

1

2

3

4

当然，事实上还会有一个匿名数组被创建出来，只是不需要我们来操心而已。

(3)直接在声明的时候并初始化

mySlice3 := []int{8,9,10}

(4)基于数组切片创建数组切片

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

newSlice := oldSlice[:3] //基于oldSlice的前3个元素创建

1

2

3

4

5

6

7

8

9

10.2数组切片的元素遍历

操作数组的所有方法都适合数组切片,通过上面的代码也可以看得出来共有2种方式:

for循环遍历,或者for循环结合range来操作

1

2

3

10.3数组切片元素的动态增减

与数组相比,数组切片多了一个存储能力的概念,即元素个数和分配的空间可以是两个不同的值,

合理地设置存储能力的值,可以大幅降低数组切片内部重新分配内存和搬送内存块的频率，

从而大大提高程序性能。通过append()函数可以在切片后面追加元素,例如:

mySlice = append(mySlice, 8, 9, 10)

append()函数的第二个参数是个不定的参数,有点类似java中的可变参数,甚至可以追加1个数组切片

mySlice2 := []int{8,9,10}

mySlice = append(mySlice,mySlice2...) //注意:这3个...点不能省略,否则编译失败

1

2

3

4

5

6

7

8

9

10.4内容复制

通过copy()函数,可以将一个数组切片复制到另一个数组切片中,

如果2个的数组切片不一样大,则会按其中较小的那个数组切片的元素个数进行复制.

slice1 := []int{1,2,3,4,5}

slice2 := []int{5,4,3}

copy(slice2,slice1) //只会复制slice1的前3个元素到slice2中,slice2中的值被自动替换

copy(slice1,slice2) //只会复制slice2的3个元素到slice1的前3个位置,该3个位置的值自动替换,其他的不变

1

2

3

4

5

6

7

11.map

在C++/Java中，map一般都以库(导包)的方式提供,在C++/Java中，

而在Go中，使用map 不需要引入任何库，并且用起来也更加方便。看下面的例子:

1

2

3

type PersonInfo struct{

ID string

Name string

Address string

}

func main(){

var personDB map[string] PersonInfo //声明一个key=string,value=PersonInfo的map

personDB = make(map[string] PersonInfo) //通过make初始化

//往这个map里插入几条数据

personDB["0"] = PersonInfo{"12345","Tom","Room 203,..."}

personDB["1"] = PersonInfo{"1","Jack","Room 101,..."}

//....

//从这个map查找键为12345的信息

person, ok := personDB["0"] //如果查找到,则ok=true,否则=false

if ok{

fmt.Println("Found person",person.Name,"with ID",person.ID)

}else{

fmt.Println("Did not find person")

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

//运行结果:

Found person Tom with ID 12345

1

2

3

11.1map的声明

上面的例子如果看不懂的话没关系,下面就介绍下map的使用

map的声明基本上没有多余的元素,形如:

var 变量名 map[key的类型] value的类型

1

2

3

4

11.2map的创建

(1)通过make()函数创建

1

2

var myMap map[int] string

myMap = make(map[int]string) //也可以简写成一条语句: myMap := make(map[int]string)

myMap[0] = "1" //赋值

myMap[1] = "2"

result,\_ := myMap[0] //这里用到了匿名变量

fmt.Println("myMap[0]=",result)

fmt.Println("myMap[1]=",myMap[1])

1

2

3

4

5

6

7

//运行结果:

myMap[0]= 1

myMap[1]= 2

1

2

3

4

通过make创建的时候,还可以指定其初始容量

myMap2 := make(map[int]string,100)

(2)方式2,通过{}赋值的方式创建

1

2

3

4

5

myMap3 := map[int]string{

0:"张三",

1:"李四",

}

fmt.Println("myMap3[0]=",myMap3[0])

fmt.Println("myMap3[1]=",myMap3[1])

1

2

3

4

5

6

//运行结果:

myMap3[0]= 张三

myMap3[1]= 李四

1

2

3

4

11.3元素的删除

通过delete()函数完成,形如:

delete(map变量名,map的key)

如果要删除的元素没有对应的key,则什么都不发生,但是如果传入的map变量的值是 nil ，

该调用将导致程序抛出异常（panic）。

1

2

3

4

5

11.4元素的查找

map通过key查找,可以返回2个变量,第一个为对应key的值,第二个为是否查找成功的bool,形如:

1

2

value,ok := map[key]

if ok{

//找到了,处理value

}else{

//未找到

}

1

2

3

4

5

6

当然,如果你很确定该key一定能够找到对应的值的话,那就可以直接使用一个变量来接收,例如:

value := map[key]

或者,第二个参数用匿名参数来接收,例如:

value,\_ := map[key]

1

2

3

4

5

11.5元素的遍历

(1)使用for循环遍历

1

2

myMap := make(map[int]int)

myMap[0] = 100

myMap[1] = 101

myMap[2] = 102

for i:=0;i<len(myMap);i++ {

fmt.Println("key=",i," value=",myMap[i])

}

1

2

3

4

5

6

7

运行结果:

key= 0 value= 100

key= 1 value= 101

key= 2 value= 102

1

2

3

4

5

(2)使用for循环结合range关键字遍历

1

2

myMap := map[int]int{

0 : 10,

1 : 20,

2 : 30,

}

for k,v := range myMap{

fmt.Println("key=",k," value=",v)

}

1

2

3

4

5

6

7

8

运行结果:

key= 0 value= 10

key= 1 value= 20

key= 2 value= 30