# go语言基础

本章给大家讲解Go语言的基本语法，包括变量、常量、数据类型及运算符。学习任何一门编程语言，其基本语法无外乎这几部分。但是值得注意的是，本章包含了其他编程语言所没有的内容。首先是变量的多重赋值，其次是匿名变量，第三点是格式化打印输出的用法，最后就是常量中iota的用法。

本章重点为大家介绍如下的内容：

* 变量声明、初始化及赋值
* 常量声明与初始化
* 数据类型
* 运算符
* 表达式

## 变量

### 变量的概念

1. 变量是计算机语言中储存数据的抽象概念。变量的功能是存储数据。变量通过变量名访问；
2. 变量的本质是计算机分配的一小块内存，专门用于存放指定数据，在程序运行过程中该数值可以发生改变；
3. 变量的存储往往具有瞬时性，或者说是临时存储，当程序运行结束，存放该数据的内存就会释放，而该变量就会消失；
4. Go 语言的变量名由字母、数字、下划线组成，首个字符不能为数字;
5. Go语法规定，定义的局部变量若没有被调用则编译错误。

### 声明变量

● 变量声明有多种形式

1、未初始化的标准格式

● var 变量名 变量类型

2、未初始化的批量格式

● 不用每行都用var申明

var (

a int

b string

c []float32

d func() bool

e struct {

x int

y string

}

)

● 未初始化变量的默认值：

○ 整形和浮点型变量默认值：0

○ 字符串默认值为空字符串

○ 布尔型默认值为false

○ 函数、指针变量初始值为nil

3、初始化变量的标准格式

● var 变量名 类型 = 表达式

4、初始化变量的编译器自动推断类型格式

● var 变量名 = 表达式

5、初始化变量的简短声明格式（短变量声明格式）

● 变量名 := 表达式

● 使用 := 赋值操作符，:= 可以高效地创建一个新的变量，称之为初始化声明。

● 声明语句省略了var 关键字

● 变量类型将由编译器自动推断

● 这是声明变量的首选形式，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

● 该变量名必须是没有定义过的变量，若定义过，将发生编译错误

● 在多个短变量声明和赋值中，至少有一个新声明的变量出现在左侧中，那么即便有其它变量名可能是重复声明的，编译器也不会报错。

### 变量多重赋值（多个变量同时赋值）

● Go语法中，变量初始化和变量赋值是两个不同的概念。Go语言的变量赋值与其他语言一样，但是Go提供了其他程序员期待已久的多重赋值功能，可以实现变量交换。多重赋值让Go语言比其他语言减少了代码量。

### 匿名变量

● Go语言的函数可以返回多个值，而事实上我们并不是对所有的返回值都用得上。那么就可以使用匿名变量，用“\_”下划线替换即可。

● 匿名变量不占用命名空间，不会分配内存。

## 数据类型

● 基本数据类型（原生数据类型）：整型、浮点型、布尔型、字符串、字符（byte、rune）

● 复合数据类型（派生数据类型）：指针（pointer）、数组（array）、切片（slice）、映射（map）、函数（function）、结构体（struct）、通道（channel）

### 整型

1、整型分两大类

* 按长度分：int8、int16、int32、int64、int
* 无符号整型：uint8、uint16、uint32、uint64、uint
* 其中uint8就是byte型，int16对应C语言的short型，int64对应C语言的long型。

表2.1 Go语言中整型

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | uint8 无符号 8 位整型 (0 到 255) 【2的8次方-1】 |
| 2 | uint16 无符号 16 位整型 (0 到 65535) 【2的16次方-1】 |
| 3 | uint32 无符号 32 位整型 (0 到 4294967295) 【2的32次方-1】 |
| 4 | uint64 无符号 64 位整型 (0 到 18446744073709551615) 【2的64次方-1】 |
| 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) |

还有其他数字类型

表2.2 Go语言中其他数字整型

|  |  |
| --- | --- |
| 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 |

int8与int是完全不同的两种数据类型，编译器不能做自动的类型转换。

例:

func main() {

var v int64

v1 := 10 //系统自动推断数据类型为int

v = v1 //此处报错

fmt.Println(v)

}

报错内容如下：

cannot use v1 (type int) as type int64 in assignment

错误提示告诉我们“不能使用int类型作为int64位类型的参数”，所以如果要使之成立就需要使用强制类型转换解决此问题。修改后的代码如下：

例:

func main() {

var v int64

v1 := 10

v = int64(v1)

fmt.Println(v)

}

同样int和uint也是完全不同的两种数据类型，编译器同样不能做自动的类型转换。

在表2.1看到的uint8这个数据类型也就是传说中的byte类型，如下例子的方式是可行的：

例:

func main() {

var v byte

var v1 uint8

v1 = 10

v = v1

fmt.Println(v)

}

既然二者为同一类型，为什么要给它们提供不同的名字呢？

一般情况下我们使用byte类型来说明此变量是一个原始的数据而不是一个小的整型数

uintptr是一种无符号整数类型，没有指定大小，但是足够容纳指针。此类型在底层编程中使用较多，与C语言函数库或者操作系统接口交互的地方会使用，其它情况使用较少。

### 浮点型

● Go语言支持4种浮点型数：float32、float64、complex64（32 位实数和虚数）、complex128（64 位实数和虚数）

● float32的最大范围是3.4e38，用常量定义是：math.MaxFloat32

● float64的最大范围是1.8e308，用常量定义是：math.MaxFloat64

例：

func main() {

fmt.Println(math.MaxFloat32) //输出：3.4028234663852886e+38

fmt.Println(math.MaxFloat64) //输出：1.7976931348623157e+308

}

通常情况下我们建议使用float64，毕竟flaot64的精度相对较大，累计计算的误差相对较小。float32的有效bit位只有23个，其它的用于指数和符号；当整数大于23bit能表达的范围时，float32的表示将出现误差，例如：

例

func main() {

var f float32 = 1 << 24 // 1 << 24

fmt.Println(f == f+1) // "true"!

fmt.Println(f) // 1.6777216e+0.7

fmt.Println(f + 1) // 1.6777216e+0.7

}

上面的例子是显示指数形式的浮点数，我们平常在使用的时候使用的是类似3.1415926这样的浮点数，那我们需要如何输出呢？

例

func main() {

var f float32 = 123453.1415926

fmt.Printf("%8.2f", f) // 123453.14

fmt.Printf("%3.1f", f) // 123453.1

fmt.Printf("%3.4f", f) // 123453.1406

}

注意，如果定义的时候不加小数点，会被推导为整型。

### 布尔型

● 声明方式：var flag bool

● 布尔型无法参与数值运算，也无法与其他类型进行转换。

例

func main() {

var v1 bool

fmt.Println(v1) //false

v1 = true

fmt.Println(v1) //true

v1 = (1 == 2)

fmt.Println(v1) //false

v2 := (2 == 2)

fmt.Println(v2) //true

var a, b int

a = 2

b = 2

v3 := (a == b)

fmt.Println(v3) //true

}

由例子可见，定义了v1变量后，系统默认赋值为false，同时我们在v3变量可以看到，我们将一个逻辑判断表达式赋值给v3，v3会被自动推导为bool类型。

### 字符串

● 字符串在Go语言中是以基本数据类型出现的，使用字符串就像使用其他原生基本数据类型int、float32、float64、bool一样。

例3.8

func main() {

var s string

s1 := "this is String"

ch := s1[0]

fmt.Println(s) //这里什么也不会显示

fmt.Println(s1) //这里输出this is String

fmt.Println(reflect.TypeOf(ch)) //这里输出uint8，也就是byte类型

fmt.Println(ch) //这里输出ascii码116

fmt.Printf("%c", ch) //这里输出字母t

}

从上面的例子我们可以看到，只定义一个字符串，在不赋值的情况下，输出的是空内容；我们可以使用s1[N]的方式，来获取第N个字符（程序员的世界数字从0开始哦^\_^）；s1[N]方式取出来的值类型是uint8类型，也就是我们之前说过的byte类型，其内容为字符串的ascii码，我们使用printf的“%c”模式，将其转换成字符串，就可以得到我们获取到的字符串了。

如果我们可以使用s1[N]的方式得到字符串的第N个字符，那么我们是不是可以通过s1[N] = “a”的方式来赋值呢？我们看下面的例子：

例

func main() {

var s = "this is String"

ch := s[1]

fmt.Printf("%c", ch)

s[1] = "k" //cannot assign to s[1]

}

这个例子会报一个编译错误，告诉我们不能给s[1]赋值。

cannot assign to s[1]

● 字符串中可以使用转移符

○ \r 回车符return，返回行首

○ \n 换行符new line，直接跳到下一行的同列位置

○ \t 制表符TAB

○ \' 单引号

○ \" 双引号

○ \\ 反斜杠

● 定义多行字符串

○ 双引号书写字符串被称为字符串字面量（string literal），这种字面量不能跨行；

○ 多行字符串需要使用“`”反引号，多用于内嵌源码和内嵌数据;

○ 在反引号中的所有代码不会被编译器识别，而只是作为字符串的一部分。

### 字符

字符串中的每一个元素叫做“字符”，定义字符时使用单引号。Go语言的字符有两种：

1、byte型：其实是uint8的别名。代表了一个ASCII码的一个字符

2、rune型：其实就是int32。代表一个UTF-8字符。当需要处理中文等unicode字符集时需要用到rune类型。

○ var a byte = 'a'

○ var b rune = '一'

例：

func main() {

var s1 rune

var s2 rune

var s3 rune

s1 = 'a'

s2 = "a"//此处报错cannot use "a" (type string) as type rune is assignment

s3 = 'abc'//此处报错 invalid character literal(more than one character)

fmt.Println(s1) //97

fmt.Printf("%c", s1) //a

}

上例中，我们给s2赋值为”a”的时候报错为不能使用字符串为字符类型赋值，而我们给s3赋值为‘abc’的时候，编译错误为无效的字符，超过了一个字符。所以我们可以得出结论rune类型在使用及赋值的时候，必须使用单引号包起来，并且只能是一个字符，不允许赋值多个。

我们给s1赋值为’a’,实际我们刚才说过rune是int32的等价类型，所以直接输出我们得到的是字符的ascii码97。

### 数组

数组是一个由固定长度的特定类型元素组成的序列，一个数组可以由N（N >= 0）个元素组成。数组的长度必须是固定的不可变的，所以很少会直接使用数组。和数组对应的两种引用类型切片类型与字典类型是可以增长和收缩的动态序列，它们的使用也更灵活，所以这两种数据类型也是未来我们最常用的数据类型。但是要理解这两种数据类型，我们需要先从理解数组类型开始。

数组的每个元素都可以通过索引下标来访问，还记得我们之前说过，程序员的世界从0开始。所以索引下标也是从0开始的。

大家可以回归到4.1.3章节看一下在字符串中，我们是如何使用下标获取字符串中单个字符的。数组的基础操作，也和字符串中获取字符的方式类似。我们通过例子看一下如何构建数组，如何对数组进行基本操作。

例：

func main() {

var a [8]int //定义长度为8整型数组

//定义长度为5的数组并初始化

var b [5]int = [5]int{0, 1, 2, 3, 4}

fmt.Println(a[0]) //输出：0

fmt.Println(b[len(b)-1]) //输出:4

fmt.Println(b) //输出：[0 1 2 3 4]

b[0] = 100 //修改b数组第0个元素的值

fmt.Println(b[0]) //输出：100

}

数组遍历的方法有两种：

例

func main() {

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

//以下为第一种遍历方式

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

fmt.Println(b[i])

//换行输出：

//0

//1

//2

//3

//4

}

//以下为第二种遍历方式

for key, value := range b {

fmt.Printf("下标是：%d 值是：%d\n", key, value)

//换行输出：

//下标是：0 值是：0

//下标是：1 值是：1

//下标是：2 值是：2

//下标是：3 值是：3

//下标是：4 值是：4

}

}

数组的长度必须是固定的，所以我们在给数组赋值的时候要注意a[8]int与b[5]int是完全不同的两种类型。

例：

func main() {

var a [8]int //定义长度为8整型数组

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

a = b //编译错误：cannot use b(type [5]int) as type [8]int in assignment

}

当我们尝试将长度为5的数组赋值给长度为8的数组时，编译报错，告诉我们不允许这样的操作。

### 结构体

结构体是一种聚合的数据类型，由N（N >=0）个任意类型的值聚合成的实体，每个值被称为结构体的成员或属性。

结构体的基本定义与使用方法如下：

例:

type Object struct {

Hash string

length int

}

func main() {

var obj Object

obj.length = 10

obj.Hash = "45454afasf4asdf1a"

fmt.Println(obj.length) //输出：10

}

我们使用type关键字定义了一个Object的结构体（struct），然后将成员用“{”、“}”包含。任何数据类型都可以成为结构体的成员的类型，接下来的内容中，我们都将使用属性来替代成员这个词，大家只要看到属性，就一定要想到，这是和结构体有关。

当属性名的首字母大写时，我们就可以在任何位置上访问这个属性；如果属性首字母是小写，那这个属性就是私有的，只能在结构体声明的包下进行访问或赋值。上例中，结构体Object是在main名声明的，所以可以在main函数中直接访问操作。如果在别的包中，就不能访问length属性，只能访问Hash属性。

### 指针

Go语言中的指针相对于C语言中的指针简单了很多，对于开发者来说我们可以不必担心因为指针的错误使用而造成的各种各样奇奇怪怪的问题。

指针是一种指向变量内存地址的数据类型。一个指针变量可以指向任何一个变量的内存地址。

在Go语言中取地址的运算符为“&”，我将通过一个短例子查看地址与指针基本的样子。

例：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

\*p = 100

fmt.Println(a) //输出：100

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：100

fmt.Println(\*p == a) //输出：true

}

上面例子中我们使用指针p指向了变量a的地址（&a），通过输出我们发现指针p的值就是变量a的地址，而\*p的值是变量a的值；当我们改变了\*p的值的时候，变量a的值也随之改变，因此我们可以认为在这种情况下，\*p等价于变量a。

我们刚才说到，每个变量都有一个地址，指针本身也是一个变量，那么指针是不是也有地址呢？指针是不是也可以被另一个指针指向呢？我们看下面的例子。

例：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

fmt.Println(&p) //输出：0xc42000c028

//定义一个指向指针的指针

var ptr \*\*int

//将指针ptr指向指针p的地址

ptr = &p

fmt.Println(ptr) //输出：指针ptr的地址为0xc42000c028

fmt.Println(\*ptr) //输出：0xc420014100

fmt.Println(\*\*ptr) //输出：10

}

上例中我们定义了一个指向指针的指针ptr，并使其指向指针p的地址。

我们在输出的时候发现

ptr的值为指针p的地址；

\*ptr的值为指针p的值，也就是变量a的地址；

\*\*ptr的值为指针\*p的值，也就是变量a的值10。

### 切片

切片也叫数组切片，通过上面对数组的介绍，我们知道数组的长度一 旦确定就不可以更改了。这种情况下是无法满足我们的真实需求的。

Go语言为我们提供了数组切片（slice）来补充数组的不足。

切片是一个可以随时动态扩充的存储空间，用于存放相同类型的数据；并且切片可以随意传递而不会导致所管理的元素被重复复制。

本书第5章将重点讲解切片的使用。

### 映射（map）

map是一种键/值（key/value）对形式的数据结构，它是一种未排序的集合，所有的key都是唯一的，通过key可以精准定位数据的位置，并对其进行相关操作。

在Go语言中，一个map就是一个哈希表的引用，所有的key的类型必须相同，value的类型也必须相同。要注意所有的的key都必须使用支持”==”比较运算符的类型。

需要注意的是浮点型最好不要使用在key中，一些误差会导致判断失败，影响程序的正常运行。value的类型则没有任何限制。

例：

func main() {

//创建一个map,key为string类型，value为string类型

m := make(map[string]string)

//给map增加值

m["username"] = "admin"

m["sex"] = "man"

m["age"] = "20"

fmt.Println(m) //输出：map[username:admin sex:man age:20]

//删除键值

delete(m, "age")

fmt.Println(m) //输出：map[username:admin sex:man]

//查询键值是否存在

value, ok := m["username"]

if ok {

fmt.Println(value) //输出：value的结果，此处为admin

} else {

fmt.Println("nil") //输出：如果ok返回false，则输出nil

}

}

通过上面的例子我们可以了解到如何创建map，如何给map增加数据、修改数据、删除数据以及查询数据是否存在 。这一个示例，已经基本上包含了map的大部分操作。

### 函数

在Go语言中，函数也作为一种数据类型。既然是数据类型那么函数也就可以当作一个值来传递和使用。

例：

func main() {

//定义一个匿名函数，赋值给fun变量

fun := func(a, b int) int {

if a > b {

return a

}

return b

}

//调用匿名函数并保存返回值

fmt.Println(fun(5, 10)) //输出：10

fmt.Println(reflect.TypeOf(fun)) //输出:func()int,int)int

}

在上面例子中，我们定义了一个匿名函数，将其赋值给变量fun，我们在程序结尾输出类型的时候，看到类型是func类型，也就是函数类型；函数类型的调用 方式也与常规函数的方法一样，关于函数详细的使用方式，我们将在第六章函数中进行详细的解读。

### 通道

通道(channel)是Go语言中为我们提供的用于在进程间通信的一种方式，我们使用channel在多个不同的进程间传递消息。

channel本身有类型的限制，一个channel只能传递一种类型的数据，在声明的时候就必须指定好要传递的类型。本书第12章并发编程中将会对channel的使用进行详细的解释说明，需要提前了解的读者请移步第12章

## 打印格式化

### 通用

● %v 值的默认格式表示 value

● %+v 类似%v，但输出结构体时会添加字段名

● %#v 值的Go语法表示

● %T 值的类型的Go语法表示 type

### 布尔值

● %t 单词true或false true

### 整数

● %b 表示为二进制 binary

● %c 该值对应的unicode码值 char

● %d 表示为十进制 digital

● %8d 表示该整型长度是8，不足8则在数值前补空格。如果超出8，则以实际为准。

● %08d 数字长度是8，不足8位的，在数字前补0。如果超出8，则以实际为准。

● %o 表示为八进制 octal

● %q 该值对应的单引号括起来的go语法字符字面值，必要时会采用安全的转义表示 quotation

● %x 表示为十六进制，使用a-f hex

● %X 表示为十六进制，使用A-F

● %U 表示为Unicode格式：U+1234，等价于"U+%04X" unicode

### 浮点数与复数的两个组分

● %b 无小数部分、二进制指数的科学计数法，如-123456p-78；参见strconv.FormatFloat

● %e （=%.6e）有6位小数部分的科学计数法，如-1234.456e+78

● %E 科学计数法，如-1234.456E+78

● %f （=%.6f）有6位小数部分，如123.456123 float

● %F 等价于%f

● %g 根据实际情况采用%e或%f格式（以获得更简洁、准确的输出）

● %G 根据实际情况采用%E或%F格式（以获得更简洁、准确的输出）

### 字符串和[]byte

● %s 直接输出字符串或者[]byte string

● %q 该值对应的双引号括起来的go语法字符串字面值，必要时会采用安全的转义表示

● %x 每个字节用两字符十六进制数表示（使用a-f）

● %X 每个字节用两字符十六进制数表示（使用A-F）

### 指针

● %p 表示为十六进制，并加上前导的0x pointer

● 没有%u。整数如果是无符号类型自然输出也是无符号的。类似的，也没有必要指定操作数的尺寸（int8，int64）。

● 宽度通过一个紧跟在百分号后面的十进制数指定，如果未指定宽度，则表示值时除必需之外不作填充。精度通过（可选的）宽度后跟点号后跟的十进制数指定。如果未指定精度，会使用默认精度；如果点号后没有跟数字，表示精度为0。举例如下：

● %f: 默认宽度，默认精度

● %9f 宽度9，默认精度

● %.2f 默认宽度，精度2

● %9.2f 宽度9，精度2

● %9.f 宽度9，精度0

### 其它flag

● '+' 总是输出数值的正负号；对%q（%+q）会生成全部是ASCII字符的输出（通过转义）；

● ' ' 对数值，正数前加空格而负数前加负号；

● '-' 在输出右边填充空白而不是默认的左边（即从默认的右对齐切换为左对齐）；

● '#' 切换格式：

● 八进制数前加0（%#o），十六进制数前加0x（%#x）或0X（%#X），指针去掉前面的0x（%#p）；

● 对%q（%#q），如果strconv.CanBackquote返回真会输出反引号括起来的未转义字符串；

● 对%U（%#U），输出Unicode格式后，如字符可打印，还会输出空格和单引号括起来的go字面值；

● 对字符串采用%x或%X时（% x或% X）会给各打印的字节之间加空格；

● '0' 使用0而不是空格填充，对于数值类型会把填充的0放在正负号后面；

## 数据类型转换

### T(表达式)

● 采用数据类型前置加括号的方式进行类型转换。T表示要转换的类型；表达式包括变量、数值、函数返回值等。

● 类型转换时，需要考虑两种类型之间的关系和范围，是否会发生数值截断。

● 布尔型无法与其他类型进行转换。

### float与int之间转换

● 需要注意float转int时精度的损失

### int转string

● 其实相当于是byte或rune转string。

● 该int数值是ASCII码的编号或Unicode字符集的编号。转成string就是将根据字符集，将对应编号的字符查找出来。

● 当该数值超出Unicode编号范围，则转成的字符串显示为乱码。

● 例如19968转string，就是“一”。

【备注：】

■ ASCII字符集中数字的10进制范围是[30 - 39]

■ ASCII字符集中大写字母的10进制范围是[65 - 90]

■ ASCII字符集中小写字母的10进制范围是[97 - 122]

■ Unicode字符集中汉字的范围是[4e00-9fa5]，10进制范围是[19968 - 40869]

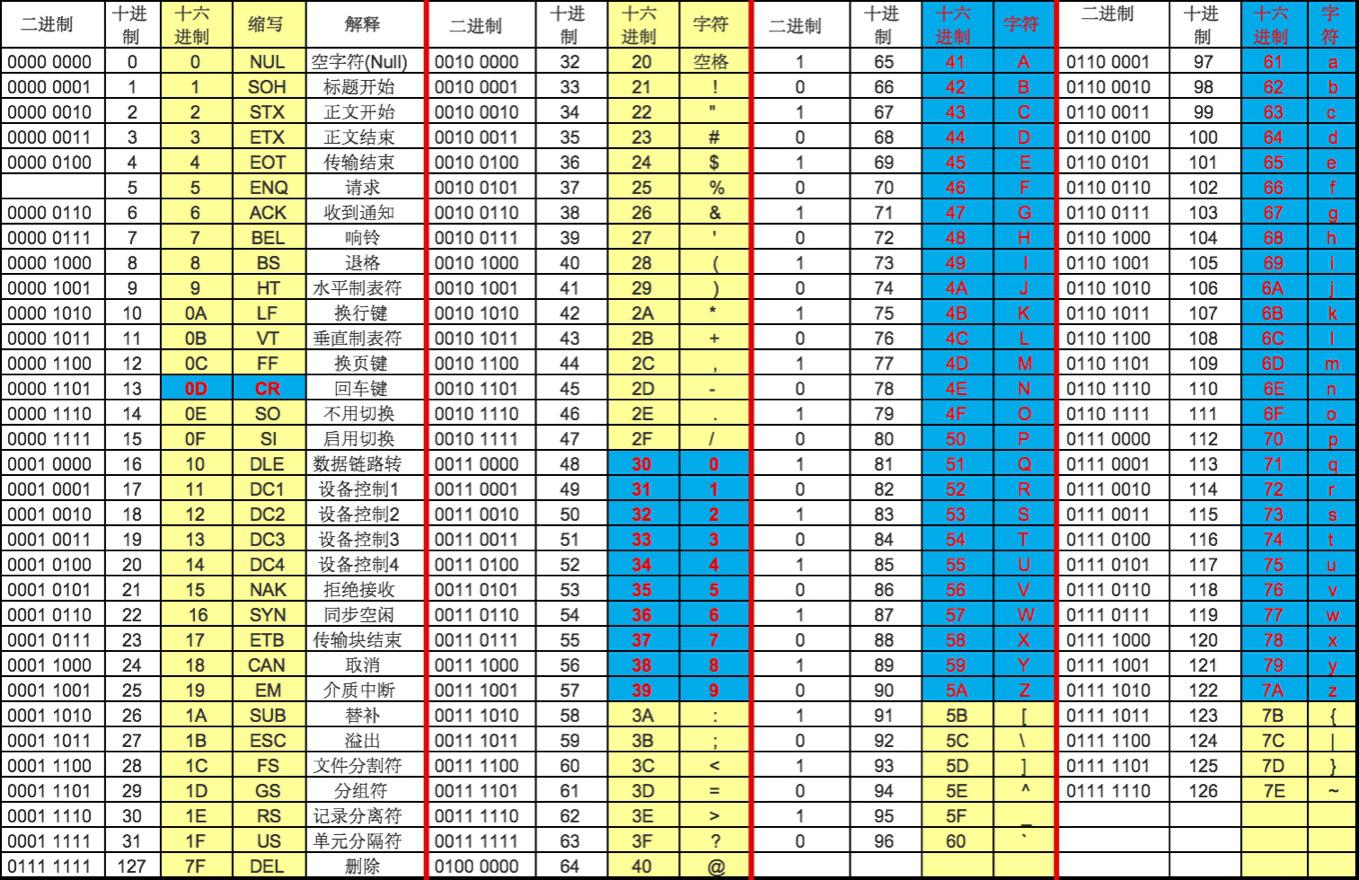


图 1

### string转int

● 不允许字符串转int（cannot convert 变量 (type string) to type int）

## 常量

### 声明方式

1、相对于变量，常量是恒定不变的值，例如圆周率。

● 常量是一个简单值的标识符，在程序运行时，不会被修改。

2、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。

3、常量的定义格式：

● const 标识符 [类型] = 值

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

○ 显式类型定义： const B string = "Steven"

○ 隐式类型定义： const C = "Steven"

4、多个相同类型的声明可以简写为：

● const WIDTH , HEIGHT = value1, value2

5、常量定义后未被使用，不会在编译时出错。

### 常量用于枚举（常量组）

● 例如以下格式：

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。

● 常量组中如果不指定类型和初始值，则与上一行非空常量的值相同。

const (

a = 10

b

c

)

打印a、b、c，输出：10 10 10

### iota

1、iota，特殊常量值，是一个系统定义的可以被编译器修改的常量值。iota只能用在常量赋值中。

2、在每一个const关键字出现时，被重置为0，然后每出现一个常量，iota所代表的数值会自动增加1。iota可以理解成常量组中常量的计数器，不论该常量的值是什么，只要有一个常量，那么iota就加1。

3、iota 可以被用作枚举值：

const (

a = iota

b = iota

c = iota

)

println(a, b, c)

● 打印输出：0 1 2

● 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

4、常量组中如果不指定类型和初始值，则与上一行非空常量的值相同。所以上述的枚举可以简写为如下形式：

const (

a = iota

b

c

)

println(a, b, c)

打印输出：0 1 2

5、示例一

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)

}

● 打印输出结果：

○ i= 1

○ j= 6

○ k= 12

○ l= 24

6、示例二

const (

a1 = '一'

b1

c1 = iota

d1

)

func main() {

fmt.Println(a1, b1, c1, d1)

}

● 打印输出结果：

○ 19968 19968 2 3

## 类型别名（Type Alias）

### 概要

类型别名是Go1.9版本添加的新功能。主要用于代码升级、迁移中类型的兼容性问题。

在Go1.9版本前内建类型定义的代码是：

● type byte uint8

● type rune int32

而在Go1.9版本之后变更为：

● type byte = uint8

● type rune = int32

### 类型别名与类型定义

1、类型别名的语法格式：

● type 类型别名 = 类型

2、定义类型的语法格式：

● type 新的类型名 类型

例如：

● type NewString string

该语句是将NewString定义为string类型。通过type关键字，NewString会形成一种新的类型。NewString本身依然具备string的特性。

● type StringAlias = string

该语句是将StringAlias定义为string的一个别名。使用StringAlias与string等效。别名类型只会在代码中存在，编译完成时，不会有别名类型。

备注：

出于性能考虑的最佳实践和建议

1. 尽可能的使用 := 去初始化声明一个变量（在函数内部）；

2. 尽可能的使用字符代替字符串；

3. 尽可能的使用切片代替数组；

4. 尽可能的使用数组和切片代替map；

5. 如果只想获取切片中某项值，不需要值的索引，尽可能的使用for range去遍历切片，这比必须查询切片中的每个元素要快一些；

6. 当数组元素是稀疏的（例如有很多0值或者空值nil），使用map会降低内存消耗；

7. 初始化map时指定其容量；

8. 当定义一个方法时，使用指针类型作为方法的接收者；

9. 在代码中使用常量或者标志提取常量的值；

10. 尽可能在需要分配大量内存时使用缓存；

11. 使用缓存模板。

## Go 语言运算符

运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 （Arithmetic operator）

下表列出了所有Go语言的算术运算符。假定 A 值为 10，B 值为 20。

表2.3 Go语言中算术运算符

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| + | 相加 | A + B 输出结果 30 |
| - | 相减 | A - B 输出结果 -10 |
| \* | 相乘 | A \* B 输出结果 200 |
| / | 相除 | B / A 输出结果 2 |
| % | 求余 | B % A 输出结果 0 |
| ++ | 自增 | A++ 输出结果 11 |
| -- | 自减 | A-- 输出结果 9 |

以下实例演示了各个算术运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var b int = 10

var c int

c = a + b

fmt.Printf("第一行 - c 的值为 %d\n", c )

c = a - b

fmt.Printf("第二行 - c 的值为 %d\n", c )

c = a \* b

fmt.Printf("第三行 - c 的值为 %d\n", c )

c = a / b

fmt.Printf("第四行 - c 的值为 %d\n", c )

c = a % b

fmt.Printf("第五行 - c 的值为 %d\n", c )

a++

fmt.Printf("第六行 - a 的值为 %d\n", a )

a=21 // 为了方便测试，a 这里重新赋值为 21

a--

fmt.Printf("第七行 - a 的值为 %d\n", a )

}

以上实例运行结果：

第一行 - c 的值为 31

第二行 - c 的值为 11

第三行 - c 的值为 210

第四行 - c 的值为 2

第五行 - c 的值为 1

第六行 - a 的值为 22

第七行 - a 的值为 20

### 关系运算符（Relational operator）

下表列出了所有Go语言的关系运算符。假定 A 值为 10，B 值为 20。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| == | 检查两个值是否相等，如果相等返回 True 否则返回 False。 | (A == B) 为 False |
| != | 检查两个值是否不相等，如果不相等返回 True 否则返回 False。 | (A != B) 为 True |
| > | 检查左边值是否大于右边值，如果是返回 True 否则返回 False。 | (A > B) 为 False |
| < | 检查左边值是否小于右边值，如果是返回 True 否则返回 False。 | (A < B) 为 True |
| >= | 检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。 | (A >= B) 为 False |
| <= | 检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。 | (A <= B) 为 True |

以下实例演示了关系运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var b int = 10

if( a == b ) {

fmt.Printf("第一行 - a 等于 b\n" )

} else {

fmt.Printf("第一行 - a 不等于 b\n" )

}

if ( a < b ) {

fmt.Printf("第二行 - a 小于 b\n" )

} else {

fmt.Printf("第二行 - a 不小于 b\n" )

}

if ( a > b ) {

fmt.Printf("第三行 - a 大于 b\n" )

} else {

fmt.Printf("第三行 - a 不大于 b\n" )

}

/\* Lets change value of a and b \*/

a = 5

b = 20

if ( a <= b ) {

fmt.Printf("第四行 - a 小于等于 b\n" )

}

if ( b >= a ) {

fmt.Printf("第五行 - b 大于等于 a\n" )

}

}

以上实例运行结果：

第一行 - a 不等于 b

第二行 - a 不小于 b

第三行 - a 大于 b

第四行 - a 小于等于 b

第五行 - b 大于等于 a

### 逻辑运算符（Logical operator）

下表列出了所有Go语言的逻辑运算符。假定 A 值为 True，B 值为 False。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| && | 逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。 | (A && B) 为 False |
| || | 逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。 | (A || B) 为 True |
| ! | 逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。 | !(A && B) 为 True |

以下实例演示了逻辑运算符的用法：

package main

import "fmt"

func main() {

var a bool = true

var b bool = false

if ( a && b ) {

fmt.Printf("第一行 - 条件为 true\n" )

}

if ( a || b ) {

fmt.Printf("第二行 - 条件为 true\n" )

}

/\* 修改 a 和 b 的值 \*/

a = false

b = true

if ( a && b ) {

fmt.Printf("第三行 - 条件为 true\n" )

} else {

fmt.Printf("第三行 - 条件为 false\n" )

}

if ( !(a && b) ) {

fmt.Printf("第四行 - 条件为 true\n" )

}

}

以上实例运行结果：

第二行 - 条件为 true

第三行 - 条件为 false

第四行 - 条件为 true

### 位运算符（Bitwise operator）

位运算符对整数在内存中的二进制位进行操作。

位运算符比一般的算术运算符速度要快，而且可以实现一些算术运算符不能实现的功能。如果要开发高效率程序，位运算符是必不可少的。位运算符用来对二进制位进行操作，包括：按位与（&）、按位或（|）、按位异或（^）、按位左移（<<）、按位右移（>>）。

假定 A = 60; B = 13; 其二进制数转换为：

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

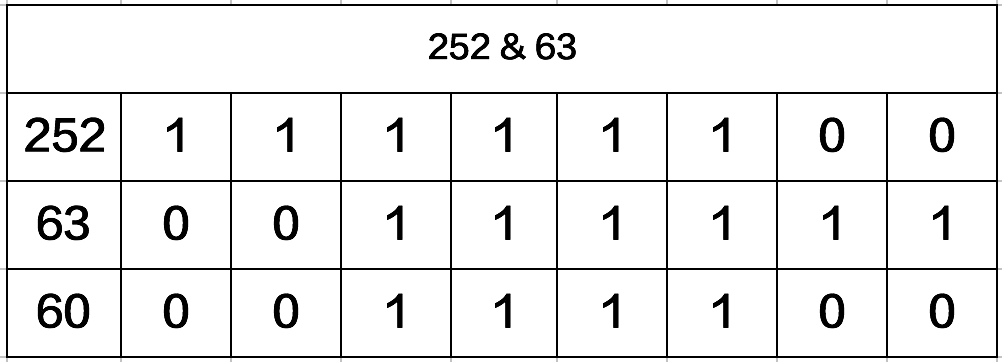
A^B = 0011 0001

Go 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| & | 按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。 | (A & B) 结果为 12, 二进制为 0000 1100 |
| | | 按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或 | (A | B) 结果为 61, 二进制为 0011 1101 |
| ^ | 按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。 | (A ^ B) 结果为 49, 二进制为 0011 0001 |
| << | 左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 | A << 2 结果为 240 ，二进制为 1111 0000 |
| >> | 右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。 | A >> 2 结果为 15 ，二进制为 0000 1111 |

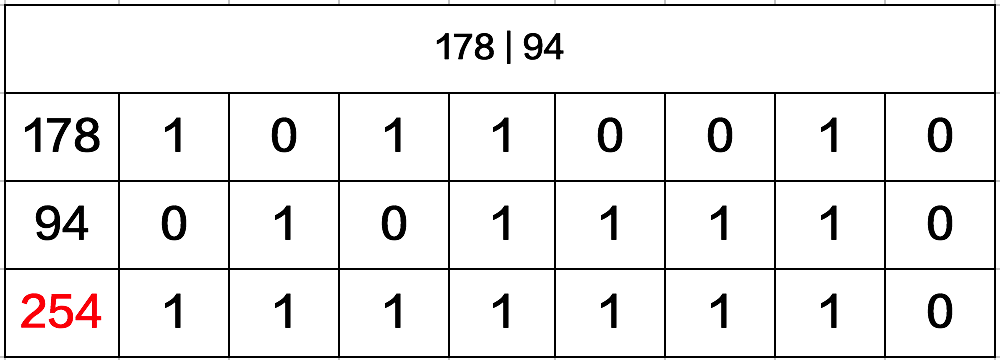
1、按位与

● 按位与（&）：对两个数进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为1 时才为1。简单说就是：同一位同时为1则为1。



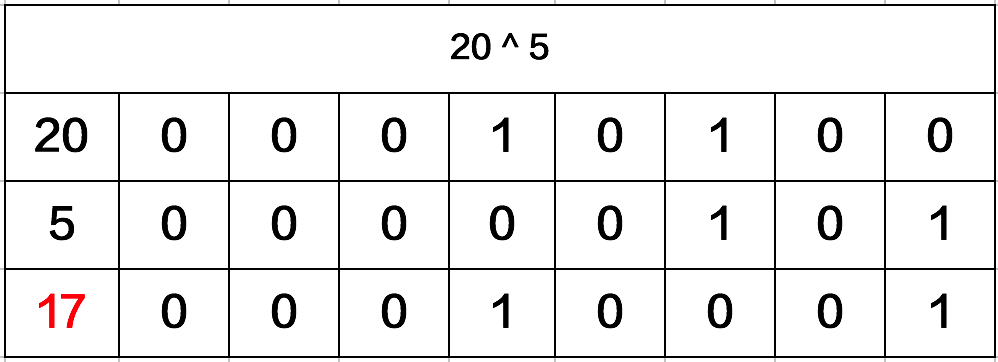
2、按位或

● 按位与（|）：比较两个数，然后返回一个新的数，这个数的每一位设置1的条件是任意一个数的同一位为1 则为1。简单说就是：同一位其中一个为1则为1。



3、按位异或

● 按位异或（^）：比较两个数，然后返回一个数，这个数的每个位设为1 的条件是两个输入数的同一位不同则为1，如果相同就设为0。简单说就是同一位不相同则为1。



4、左移运算符（<<）

● 按二进制形式把所有的数字向左移动对应的位数，高位移出(舍弃)，低位的空位补零。

（1）、语法格式:

● 需要移位的数字 << 移位的次数

● 例如： 3 << 4，则是将数字3左移4位

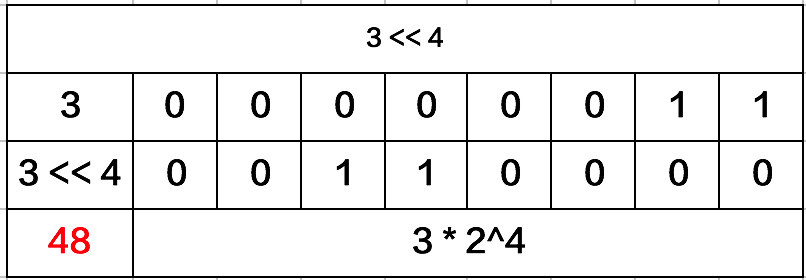
（2）、计算过程：

● 3 << 4

● 首先把3转换为二进制数字0000 0000 0000 0000 0000 0000 0000 0011，然后把该数字高位(左侧)的两个零移出，其他的数字都朝左平移4位，最后在低位(右侧)的两个空位补零。则得到的最终结果是0000 0000 0000 0000 0000 0000 0011 0000，则转换为十进制是48。

（3）、数学意义：

● 在数字没有溢出的前提下，对于正数和负数，左移一位都相当于乘以2的1次方，左移n位就相当于乘以2的n次方。



5、右移运算符（>>）

● 按二进制形式把所有的数字向右移动对应位移位数，低位移出(舍弃)，高位的空位补符号位，即正数补零，负数补1。

（1）、语法格式：

● 需要移位的数字 >> 移位的次数

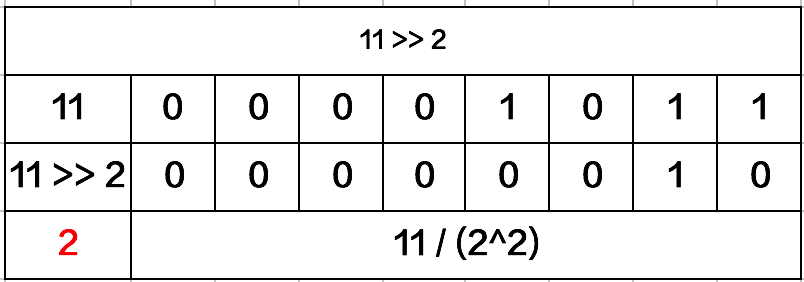
● 例如11 >> 2，则是将数字11右移2位

（2）、计算过程：

● 11的二进制形式为：0000 0000 0000 0000 0000 0000 0000 1011，然后把低位的最后两个数字移出，因为该数字是正数，所以在高位补零。则得到的最终结果是0000 0000 0000 0000 0000 0000 0000 0010。转换为十进制是2。

（3）、数学意义：

● 右移一位相当于除2，右移n位相当于除以2的n次方。



以下实例演示了位运算符的用法：

package main

import "fmt"

func main() {

var a uint = 60 /\* 60 = 0011 1100 \*/

var b uint = 13 /\* 13 = 0000 1101 \*/

var c uint = 0

c = a & b /\* 12 = 0000 1100 \*/

fmt.Printf("第一行 - c 的值为 %d\n", c )

c = a | b /\* 61 = 0011 1101 \*/

fmt.Printf("第二行 - c 的值为 %d\n", c )

c = a ^ b /\* 49 = 0011 0001 \*/

fmt.Printf("第三行 - c 的值为 %d\n", c )

c = a << 2 /\* 240 = 1111 0000 \*/

fmt.Printf("第四行 - c 的值为 %d\n", c )

c = a >> 2 /\* 15 = 0000 1111 \*/

fmt.Printf("第五行 - c 的值为 %d\n", c )

}

以上实例运行结果：

第一行 - c 的值为 12

第二行 - c 的值为 61

第三行 - c 的值为 49

第四行 - c 的值为 240

第五行 - c 的值为 15

### 赋值运算符（Assignment operator）

下表列出了所有Go语言的赋值运算符。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| = | 简单的赋值运算符，将一个表达式的值赋给一个左值 | C = A + B 将 A + B 表达式结果赋值给 C |
| += | 相加后再赋值 | C += A 等于 C = C + A |
| -= | 相减后再赋值 | C -= A 等于 C = C - A |
| \*= | 相乘后再赋值 | C \*= A 等于 C = C \* A |
| /= | 相除后再赋值 | C /= A 等于 C = C / A |
| %= | 求余后再赋值 | C %= A 等于 C = C % A |
| <<= | 左移后赋值 | C <<= 2 等于 C = C << 2 |
| >>= | 右移后赋值 | C >>= 2 等于 C = C >> 2 |
| &= | 按位与后赋值 | C &= 2 等于 C = C & 2 |
| ^= | 按位异或后赋值 | C ^= 2 等于 C = C ^ 2 |
| |= | 按位或后赋值 | C |= 2 等于 C = C | 2 |

以下实例演示了赋值运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var c int

c = a

fmt.Printf("第 1 行 - = 运算符实例，c 值为 = %d\n", c )

c += a

fmt.Printf("第 2 行 - += 运算符实例，c 值为 = %d\n", c )

c -= a

fmt.Printf("第 3 行 - -= 运算符实例，c 值为 = %d\n", c )

c \*= a

fmt.Printf("第 4 行 - \*= 运算符实例，c 值为 = %d\n", c )

c /= a

fmt.Printf("第 5 行 - /= 运算符实例，c 值为 = %d\n", c )

c = 200;

c <<= 2

fmt.Printf("第 6行 - <<= 运算符实例，c 值为 = %d\n", c )

c >>= 2

fmt.Printf("第 7 行 - >>= 运算符实例，c 值为 = %d\n", c )

c &= 2

fmt.Printf("第 8 行 - &= 运算符实例，c 值为 = %d\n", c )

c ^= 2

fmt.Printf("第 9 行 - ^= 运算符实例，c 值为 = %d\n", c )

c |= 2

fmt.Printf("第 10 行 - |= 运算符实例，c 值为 = %d\n", c )

}

以上实例运行结果：

第 1 行 - = 运算符实例，c 值为 = 21

第 2 行 - += 运算符实例，c 值为 = 42

第 3 行 - -= 运算符实例，c 值为 = 21

第 4 行 - \*= 运算符实例，c 值为 = 441

第 5 行 - /= 运算符实例，c 值为 = 21

第 6行 - <<= 运算符实例，c 值为 = 800

第 7 行 - >>= 运算符实例，c 值为 = 200

第 8 行 - &= 运算符实例，c 值为 = 0

第 9 行 - ^= 运算符实例，c 值为 = 2

第 10 行 - |= 运算符实例，c 值为 = 2

### 其他运算符

下表列出了Go语言的其他运算符。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| & | 返回变量存储地址 | &a; 将给出变量的实际地址。 |
| \* | 指针变量。 | \*a; 是一个指针变量 |

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

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

|  |  |
| --- | --- |
| 优先级 | 运算符 |
| 7 | ^ ! |
| 6 | \* / % << >> & &^ |
| 5 | + - | ^ |
| 4 | == != < <= >= > |
| 3 | <- |
| 2 | && |
| 1 | || |

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

以上实例运行结果：

package main

import "fmt"

func main() {

var a int = 20

var b int = 10

var c int = 15

var d int = 5

var e int;

e = (a + b) \* c / d; // ( 30 \* 15 ) / 5

fmt.Printf("(a + b) \* c / d 的值为 : %d\n", e );

e = ((a + b) \* c) / d; // (30 \* 15 ) / 5

fmt.Printf("((a + b) \* c) / d 的值为 : %d\n" , e );

e = (a + b) \* (c / d); // (30) \* (15/5)

fmt.Printf("(a + b) \* (c / d) 的值为 : %d\n", e );

e = a + (b \* c) / d; // 20 + (150/5)

fmt.Printf("a + (b \* c) / d 的值为 : %d\n" , e );

}

以上实例运行结果：

(a + b) \* c / d 的值为 : 90

((a + b) \* c) / d 的值为 : 90

(a + b) \* (c / d) 的值为 : 90

a + (b \* c) / d 的值为 : 50

[第2章 go语言基础 1](#_Toc28642464)

[2.1 变量 1](#_Toc357555520)

[2.1.1 变量的概念 1](#_Toc776380334)

[2.1.2 声明变量 1](#_Toc513634366)

[2.1.3 变量多重赋值（多个变量同时赋值） 2](#_Toc1916012069)

[2.1.4 匿名变量 3](#_Toc897556918)

[2.2 数据类型 3](#_Toc1313984298)

[2.2.1 整型 3](#_Toc1559754385)

[2.2.2 浮点型 5](#_Toc459069766)

[2.2.3 布尔型 6](#_Toc1824297138)

[2.2.4 字符串 6](#_Toc1337970147)

[2.2.5 字符 7](#_Toc962992892)

[2.2.6 数组 8](#_Toc1584772052)

[2.2.7 结构体 9](#_Toc24204223)

[2.2.8 指针 10](#_Toc925966678)

[2.2.9 切片 11](#_Toc2055450984)

[2.2.10 映射（map） 11](#_Toc1542742446)

[2.2.11 函数 12](#_Toc154736044)

[2.2.12 通道 13](#_Toc45994991)

[2.3 打印格式化 13](#_Toc2091184464)

[2.3.1 通用 13](#_Toc819919646)

[2.3.2 布尔值 13](#_Toc2134411170)

[2.3.3 整数 13](#_Toc1481694702)

[2.3.4 浮点数与复数的两个组分 14](#_Toc622485902)

[2.3.5 字符串和[]byte 14](#_Toc1727710377)

[2.3.6 指针 14](#_Toc1501915152)

[2.3.7 其它flag 14](#_Toc1165172826)

[2.4 数据类型转换 15](#_Toc156309589)

[2.4.1 T(表达式) 15](#_Toc722762042)

[2.4.2 float与int之间转换 15](#_Toc1294132462)

[2.4.3 int转string 15](#_Toc769912018)

[2.4.4 string转int 16](#_Toc1322313351)

[2.5 常量 16](#_Toc1959711101)

[2.5.1 声明方式 16](#_Toc907780468)

[2.5.2 常量用于枚举（常量组） 17](#_Toc1342497388)

[2.5.3 iota 17](#_Toc1890404734)

[2.6 类型别名（Type Alias） 18](#_Toc11806973)

[2.6.1 概要 18](#_Toc871299687)

[2.6.2 类型别名与类型定义 19](#_Toc242850516)

[2.7 Go 语言运算符 20](#_Toc1369693112)

[2.7.1 算术运算符 （Arithmetic operator） 20](#_Toc1554921191)

[2.7.2 关系运算符（Relational operator） 21](#_Toc831956794)

[2.7.3 逻辑运算符（Logical operator） 22](#_Toc431811141)

[2.7.4 位运算符（Bitwise operator） 23](#_Toc1102603574)

[2.7.5 赋值运算符（Assignment operator） 27](#_Toc821878255)

[2.7.6 其他运算符 29](#_Toc693014281)

[2.8 运算符优先级 30](#_Toc1687203086)