# Go基础

[Go基础 1](#_Toc152965066)

[1. 变量 2](#_Toc152965067)

[1.1. 概念 2](#_Toc152965068)

[1.2. 整数类型、浮点类型、字符类型、布尔类型 3](#_Toc152965069)

[1.2.1整数类型 3](#_Toc152965070)

[1.2.2 浮点类型 4](#_Toc152965071)

[1.2.3 字符类型 9](#_Toc152965072)

[1.2.4 布尔类型 11](#_Toc152965073)

[1.2.5 字符串类型 11](#_Toc152965074)

[1.3 基本数据类型的默认值以及转换 12](#_Toc152965075)

[1.3.1 基本数据类型默认值 12](#_Toc152965076)

[1.3.2 类型转换 12](#_Toc152965077)

[1.4 基本数据类型和string相互转换 13](#_Toc152965078)

[1.4.1 基本数据类型转换string类型 13](#_Toc152965079)

[1.5 指针、值类型与引用类型 16](#_Toc152965080)

[1.5.1 指针 16](#_Toc152965081)

[1.5.2 值类型和引用类型 17](#_Toc152965082)

[1.6 标识符的命名规范 18](#_Toc152965083)

[2. 运算符 19](#_Toc152965084)

[2.1算数运算符 19](#_Toc152965085)

[2.2.关系运算符 20](#_Toc152965086)

[2.3.逻辑运算符 21](#_Toc152965087)

[2.4.位运算符 22](#_Toc152965088)

[2.4.1 进制 22](#_Toc152965089)

[2.4.2 位运算符、原码、反码、补码 25](#_Toc152965090)

[2.5.赋值运算符 26](#_Toc152965091)

[2.7.运算符优先级 27](#_Toc152965092)

[3. 流程控制 29](#_Toc152965093)

[4. 函数、包、错误处理 29](#_Toc152965094)

[5. 数组和切片 29](#_Toc152965095)

[6. 排序和查找 29](#_Toc152965096)

[7. map 29](#_Toc152965097)

[8. 面向对象编程 29](#_Toc152965098)

[9. 小项目案例 29](#_Toc152965099)

[10. 文件操作 29](#_Toc152965100)

[11. 单元测试 29](#_Toc152965101)

[12. 协程和管道 29](#_Toc152965102)

[13. 反射和常量 29](#_Toc152965103)

[14. TCP编程 29](#_Toc152965104)

[15. redis使用 29](#_Toc152965105)

[16. 数据结构 29](#_Toc152965106)

[17. Web开发基础 29](#_Toc152965107)

# 变量

## 概念

变量相当于内存中一个数据存储空间的表示，通过变量名可以访问到变量值。

变量使用的步骤：

1. 声明变量（定义变量）
2. 赋值
3. 使用

|  |
| --- |
| package main  import "fmt"  func main() {  // 定义变量/声明变量  var i int  // 给变量赋值  i = 10  // 使用变量  fmt.Println(i) } |

变量使用注意事项

1）变量表示内存中一个存储区域

2）该区域有自己的名称（变量名）和类型（数据类型）

3）go变量使用有三种方式

（1）指定变量类型，声明后若不赋值，则使用默认零值

（2）根据值自行判断变量类型（类型推导）

（3）省略 var，注意 := 左侧的变量不应该是已经声明过的，且这种方式仅适用于局部变量

4）多变量声明

在编程中,有时候我们需要一次性声明多个变量,go也提供这样的语法

|  |
| --- |
| package main  import "fmt"  func main() {  // 1、指定变量类型，声明后若不赋值则使用默认零值  var i int  fmt.Println(i) // 0  // 2、根据值自行判断变量类型（类型推导）  var num = 10.1  fmt.Println(num)  // 3、省略 var，注意 := 左侧的变量不应该是已经声明过的，且这种方式仅适用于局部变量  num1 := 10.2  fmt.Println(num1)  // 4、一次性声明多个变量  n1, n2, n3 := 10, "abc", 10.2  fmt.Println(n1, n2, n3) } |

5）全局变量定义

|  |
| --- |
| var a = 10 var b = 20 var c = 30  var (  n1 = 10  n2 = 20  n3 = 30 ) |

6）变量在同一个作用于内（同一个函数或者代码块中）不能重名

7）变量 = 变量名+变量值+变量类型

8）go变量如果没有赋初值,编译器会使用默认值,比如:int默认值0,string默认值空串

9）声明的变量必须使用

## 整数类型、浮点类型、字符类型、布尔类型

### 1.2.1整数类型

简单的来说,就是用于存放整数值的,比如:0,-1,2345等简单的来说,就是用于存放整数值的,比如:0,-1,2345等

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 有无符号 | 占用存储空间，通过unsafe.Sizeof(变量)  可以查看 | 表数范围 | 备注 |
| int | 有 | 32位操作系统4个字节  64位操作系统8个字节 | -2^31~2^31-1  -2^63~2^63-1 |  |
| uint | 无 | 32位操作系统4个字节  64位操作系统8个字节 | 0~2^32-1  0~2^64-1 |  |
| rune | 有 | 与int32相同 | -2^31~2^31-1 | 等价int32，表示一个unicode码，也就是可以表示汉字 |
| byte | 无 | 与uint8相同 | 0~2^8-1 | 存储字符时选择byte |
| int8 | 有 | 1个字节 | -2^7~2^7 -1即: -128~127 |  |
| int16 | 有 | 2个字节 | -2^15~2^15-1 |  |
| int32 | 有 | 4个字节 | -2^31~2^31-1 |  |
| int64 | 有 | 8个字节 | -2^63~2^63-1 |  |
| uint8 | 无 | 1个字节 | 0~2^8-1即：0～255 |  |
| uint16 | 无 | 2个字节 | 0~2^16-1 |  |
| uint32 | 无 | 4个字节 | 0~2^32-1 |  |
| uint64 | 无 | 8个字节 | 0~2^64-1 |  |

整数使用细节：

1).go各个整数类型分:有符号和无符号, int uint 的大小和操作系统有关

2).go的整型默认声明为int型

3).go程序中整型变量在使用时,遵守保小不保大原则,即:在保证程序正确运行下,尽量使用占用空间小的数据类型(如:年龄使用uint8类型)

4).bit:计算机中的最小存储单位.byte:计算机中基本存储单元.1byte = 8 bit

### 1.2.2 浮点类型

go的浮点类型可以表示一个小数比如1.2，0.1等

|  |  |  |
| --- | --- | --- |
| 类型 | 占用存储空间 | 表数范围 |
| 单精度float32 | 4个字节 | -3.403E38~3.403E38 |
| 双精度float64 | 8个字节 | -1.798E308~1.798E308 |

说明：

1).浮点数在机器中存放形式:浮点数 = 符号位 + 指数位 + 尾数位

说明:浮点数都是有符号的

整数部分换算二进制：处2取余，倒叙排列

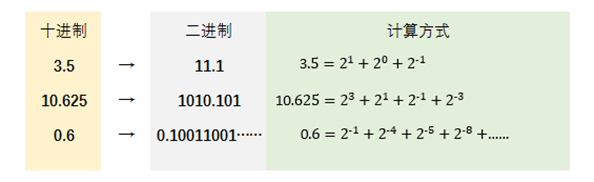
小数部分换算二进制：乘2取整，顺序排列

引用连接：https://cloud.tencent.com/developer/article/1473541

要表示浮点数的第一步，就是让小数也能使用二进制来表示。我们知道二进制表示整数时，最低位代表2的0次方，往高位依次是2的1次方，2次方，3次方……那么对应的，二进制数小数点后面，最高位则是2的-1次方，-2次方，-3次方……如下图所示：

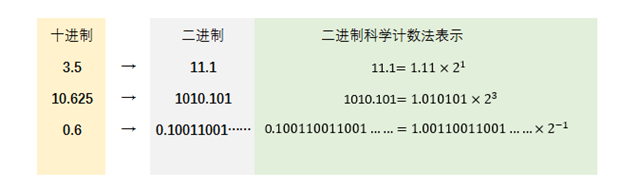


下面举几个例子：

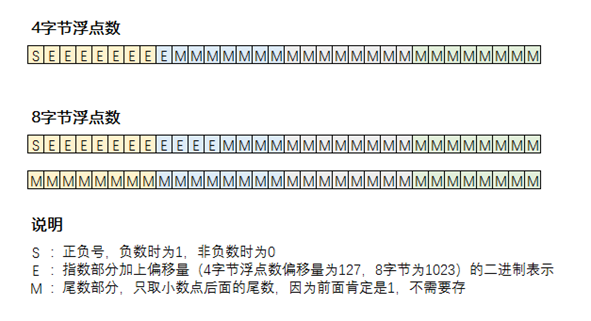


十进制数字10.625，用二进制表示为1010.101 。其实这种二进制表示小数的方法，造成了一个隐含的问题：一些本来不是无限循环的十进制小数，表示成二进制之后成了无限循环小数。比如上图中的十进制数字0.6，表示成二进制之后成了循环体为1001的无限循环小数。这就是“浮点数有精度问题”的根源之一，你在代码中声明一个变量double a = 0.6;时，计算机底层其实是无法精确存储那个无限循环二进制数的，只能存一个四舍五入（准确说应该是零舍一入，毕竟是二进制）后的近似值。

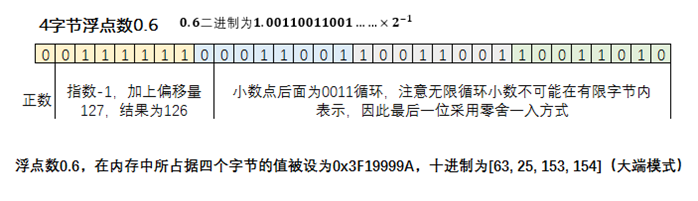
下一步，将二进制表示为以2为底的科学计数法，如图：



对于任何数字表示成二进制科学计数法以后，一定是1点几（尾数）乘以2的多少次方（指数）。对于小于零的负数来说，就是负1点几（尾数）乘以2的多少次方（指数）。所以要存这个数，需要存储三个部分：正负号，尾数，指数。



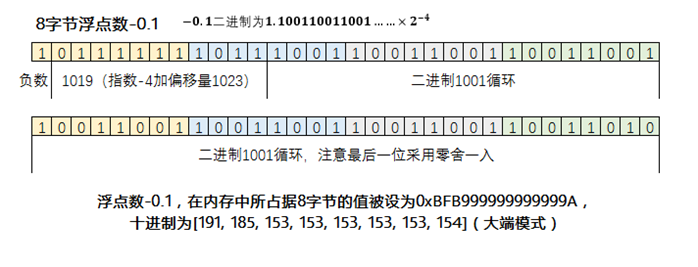
具体存储方式如上图所示。最高位有1bit存储正负号，然后指数部分占据8bits（4字节）或11bits（8字节），其余部分全都用来存储尾数部分。对于指数部分，这里存储的结果是实际的指数加上偏移量之后的结果。这里设置偏移量，是为了让指数部分不出现负数，全都为大于等于0的正整数。尾数部分的存储，因为二进制的科学计数法，小数点前一定是1开头，因此我们尾数只需要存储小数点后面的部分即可。接下来依然是举例说明：



如果你在程序中声明float a = 0.6，那么实际上a变量在内存中占据的4个字节的值为0x3F19999A。其实如果你再声明一个 uint32 b = 1058642330，其实b变量所占据的4个字节的值也是0x3F19999A，因为整数在内存中就是直接按照二进制值来存储，刚好a和b两个变量在内存中的值一模一样，只不过他们的数据类型不同。

另外值得注意的是，虽然float a=0.6在内存中被存为了数字0x3F19999A，但是如果我们把4个字节看作是长度为4的byte数组，不同的计算机对这个数组有不同的存储方式。在大端模式下，这个数组为[0x3F, 0x19, 0x99, 0x9A]，即数字的高位在数组靠前位置；在小端模式下，这个数组为[0x9A, 0x99, 0x19, 0x3F]，即数字的高位在数组靠后位置，与我们正常的阅读习惯刚好反过来。正常的x86\_64位机器是小端存储模式，ARM是大端存储模式

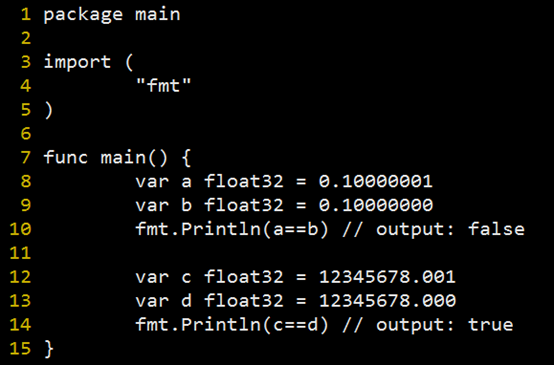
再来看一个8字节浮点数的例子：



8字节-0.1的存储方式

8字节数字-0.1，可以看到最高位为1，表示负数。后面逻辑和前文的4字节浮点数类似，只是偏移量略有区别。

浮点数的这种表示法，其实对于绝对值比较大的数来说，小数点后面的精度会比较差。对于绝对值接近0的比较小的数来说，小数点后面的精度反而会非常高。我们用一段简单的golang代码来说明一下（非常简单，非golang开发也能看懂）。



我们可以看到，变量a和b的差距只有0.00000001，但是他们在内存中所存储的值依然是不同的，a和b比较会返回false。但是对于c和d来说，他们值只差了0.001，小数点后的差距比a和b的差距要大很多，c和d的判断结果依然是相等。这是由于c和d整数部分占据了4字节太多位置，导致小数部分的数值差距，在4字节内已经体现不出来了。c和d在内存中存的值是完全一样的。前文所说的零舍一入机制，加上浮点数在内存中本身的存储机制，导致了我们编程中经常被提醒的：“浮点数有精度问题”。

以上知识主要的应用场景是浮点数的序列化，在各种通信协议中经常会用到，如protobuf的序列化算法。

浮点型常量有两种表示形式：

(1).十进制数形式: 5.12(必须有小数点)

(2).科学计数法形式: 5.1234e2 = 5.12 \* 10的2次方; 5.12E-2 = 5.12/10的2次方

|  |
| --- |
| package main  import "fmt"  func main() {  // 科学计数法表示浮点型，默认是float64  f1 := 5.123e10  fmt.Println(f1) } |

### 1.2.3 字符类型

go中并没有专门的字符类型,如果要单独存储单个字符(字母),一般使用byte来保存.

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,也就是说对于传统的字符串是由字符组成的,而go的字符串不同,它是由字节组成(官方将string归属到基本数据类型)

|  |
| --- |
| package main  import "fmt"  func main() {  var c1 byte = 'a'  var c2 byte = '0'  //var c3 byte = '北'. overflows  var c3 int = '北'  fmt.Println(c1, c2, c3)  fmt.Printf("c1 = %c, c2 = %c, c3 = %c", c1, c2, c3)  /\*  97 48 21271  c1 = a, c2 = 0, c3 = 北  \*/ }  上面的代码，默认输出的是码点值 |

1).如果我们保存的字符在ASCII表的,比如:[0-9,a-z,A-Z]直接可以保存到byte

2).如果我们保存的字符对应码值大于255,这时我们可以考虑使用int类型保存

3).如果我们需要按照字符的方式输出,这时我们需要格式化输出,即fmt.Printf("%c",c1)

type byte = uint8 // 类型别名使用 %T 格式化输出为 uint8 类型，和后面的自定义类型区分开

1）go中允许使用转义字符'\'来将后面的字符转变为特殊字符型常量,

例如：var c1 char = '\n',其中 '\n'表示换行符

|  |
| --- |
| \a 匹配响铃符 （相当于 \x07）  注意：正则表达式中不能使用 \b 匹配退格符，因为 \b 被用来匹配单词边界，  可以使用 \x08 表示退格符。  \f 匹配换页符 （相当于 \x0C）  \t 匹配横向制表符（相当于 \x09）  \n 匹配换行符 （相当于 \x0A）  \r 匹配回车符 （相当于 \x0D）  \v 匹配纵向制表符（相当于 \x0B）  \123 匹配 8 進制编码所代表的字符（必须是 3 位数字）  \x7F 匹配 16 進制编码所代表的字符（必须是 3 位数字）  \x{10FFFF} 匹配 16 進制编码所代表的字符（最大值 10FFFF ）  \Q...\E 匹配 \Q 和 \E 之间的文本，忽略文本中的正则语法    \\ 匹配字符 \  \^ 匹配字符 ^  \$ 匹配字符 $  \. 匹配字符 .  \\* 匹配字符 \*  \+ 匹配字符 +  \? 匹配字符 ?  \{ 匹配字符 {  \} 匹配字符 }  \( 匹配字符 (  \) 匹配字符 )  \[ 匹配字符 [  \] 匹配字符 ]  \| 匹配字符 | |

2）go语言的字符使用UTF-8编码,英文字母1个字节,汉字3个字节

3）在go中,字符的本质是一个整数,直接输出时,是该字符对应的UTF-8编码的码值

4）可以直接给某个变量赋一个数字然后按格式化输出时%c,会输出该数字对应的unicode字符

5）字符类型是可以进行运算的,相当于一个整数,因为它都对应有unicode码

package main  
  
import "fmt"  
  
func main() {  
 var c1 byte = 'a'  
 fmt.Println(10 + c1) // 相当于 10 + 97 等于107  
}

字符类型的本质：

1）字符型存储到计算机中,需要将字符对应的码值(整数)找出来存储,读取

2）字符和码值的对应关系是通过字符编码表决定的(规定好的)

3）go语言的编码都统一成UTF-8,非常方便

### 1.2.4 布尔类型

基本介绍:

1).布尔类型也叫bool类型,bool类型只允许取值true和false

2).bool类型占一个字节

3).bool类型适用于逻辑运算,一般用于程序流程控制

        if条件控制语句

        for循环控制语句

### 1.2.5 字符串类型

字符串就是一串固定长度的字符连接起来组成的字符序列.

go的字符串是由单个字符连接起来的

go的字符串的字节使用UTF-8编码标识Unicode文本

注意事项和使用细节：

1） go的字符串的字节使用UTF-8编码标识Unicode文本,这样go统一使用UTF-8编码,乱码问题不会再困扰程序员了

2）字符串一旦赋值了,字符串就不能修改了,在go语言中字符串是不可变的

package main  
  
import "fmt"  
  
func main() {  
 var str string = "hello"  
 str[0] = 'y' // cannot assign to str[0] (value of type byte)  
 fmt.Println(str)  
}

3）字符串的两种表示方式

(1).双引号,会识别转义字符

(2).反引号,以字符串的原生形式输出,包括换行和特殊字符,可以实现防止攻击,输出源代码等效果

package main  
  
import "fmt"  
  
func main() {  
 // 反引号表示  
 str1 := `  
package main  
import (  
 "fmt"  
 "unsafe"  
)`  
 fmt.Println(str1)  
 /\*  
 package main  
 import (  
 "fmt"  
 "unsafe"  
 )  
  
 \*/  
}

1. 字符串拼接

package main  
  
import "fmt"  
  
func main() {  
 str1 := "hello" + "world" +  
 "test1" + "test2" +  
 "test3"  
 fmt.Println(str1)  
}

字符串拼使用 + 号，必须留在上一行

## 1.3 基本数据类型的默认值以及转换

### 1.3.1 基本数据类型默认值

|  |  |
| --- | --- |
| 数据类型 | 默认值 |
| 整形 | 0 |
| 浮点型 | 0 |
| 字符串 | “” |
| 布尔类型 | false |

### 1.3.2 类型转换

go语言和java/c不同,go在不同类型的变量之间赋值时需要显式转换,也就是说go中数据类型不能自动转换

1） 基本语法

表达式T(v)  将值v转换为类型T

T:数据类型,比如:int32,int64,float32等等

v:需要转换的变量

package main  
  
import "fmt"  
  
func main() {  
 var i int32 = 10  
 var f1 float32 = float32(i)  
 var n2 int8 = int8(i)  
 var n3 int64 = int64(i)  
 fmt.Printf("i=%v, f1=%v, n2=%v, n3=%v", i, f1, n2, n3)  
}

细节说明:

1).在go中,数据类型的转换可以是从 表示范围小->表示范围大, 也可以 范围大->范围小

2).被转换的是变量存储的数据(即值),变量本身的数据类型并没有发生变化

3).在转换中,比如将int16转换成int8,编译时不会报错,只是转换的结果可能会按溢出处理,和我们希望的结果不一样

package main  
  
import "fmt"  
  
func main() {  
 var i int32 = 300  
 var n1 int8 = int8(i)  
 fmt.Println(n1) // 44 益处  
}

## 1.4 基本数据类型和string相互转换

在程序开发中,我们经常需要将基本[数据类型转换](https://so.csdn.net/so/search?q=%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B%E8%BD%AC%E6%8D%A2&spm=1001.2101.3001.7020" \t "_blank)成string类型,或将string类型转换成基本数据类型

### 1.4.1 基本数据类型转换string类型

方式1:

fmt.Sprintf("%参数", 表达式)

func Sprintf(format string, a ...any) string

1).参数需要和表达式的数据类型相匹配

2).fmt.Sprintf()会返回转换后的字符串

package main  
  
import "fmt"  
  
func main() {  
 var n1 int = 100  
 var n2 float32 = 10.1  
 var b bool = true  
 var b1 byte = 'a'  
 var str1 string  
  
 str1 = fmt.Sprintf("%d", n1)  
 str1 = fmt.Sprintf("%f", n2)  
 str1 = fmt.Sprintf("%t", b)  
 str1 = fmt.Sprintf("%c", b1)  
 fmt.Printf("%T\n", str1) // string  
}

方式2:

使用strconv包的函数

package main  
  
import (  
 "fmt"  
 "strconv"  
)  
  
func main() {  
 var n1 int = 100  
 var n2 float32 = 10.1  
 var b bool = true  
 var str1 string  
  
 // FormatInt(int64(n1), 10):n1 要转换的数, 10: 10进制  
 str1 = strconv.FormatInt(int64(n1), 10)  
 fmt.Printf("%T\n", str1)  
 /\*\*  
 \* strconv.FormatFloat(float64(n2),, 'f', 10, 64):  
 \* f 代表格式,  
 \* 10: 表示小数位保留10位,  
 \* 64: 表示这个小数是float64  
 \*/  
 str1 = strconv.FormatFloat(float64(n2), 'f', 10, 64)  
 fmt.Printf("%T\n", str1)  
 str1 = strconv.FormatBool(b)  
 fmt.Printf("%T\n", str1)  
}  
  
/\*  
string  
string  
string  
  
 \*/

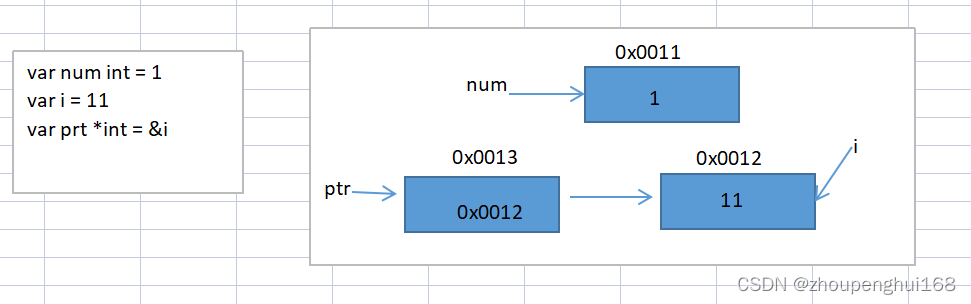
1.4.2 string类型转换为基本数据类型

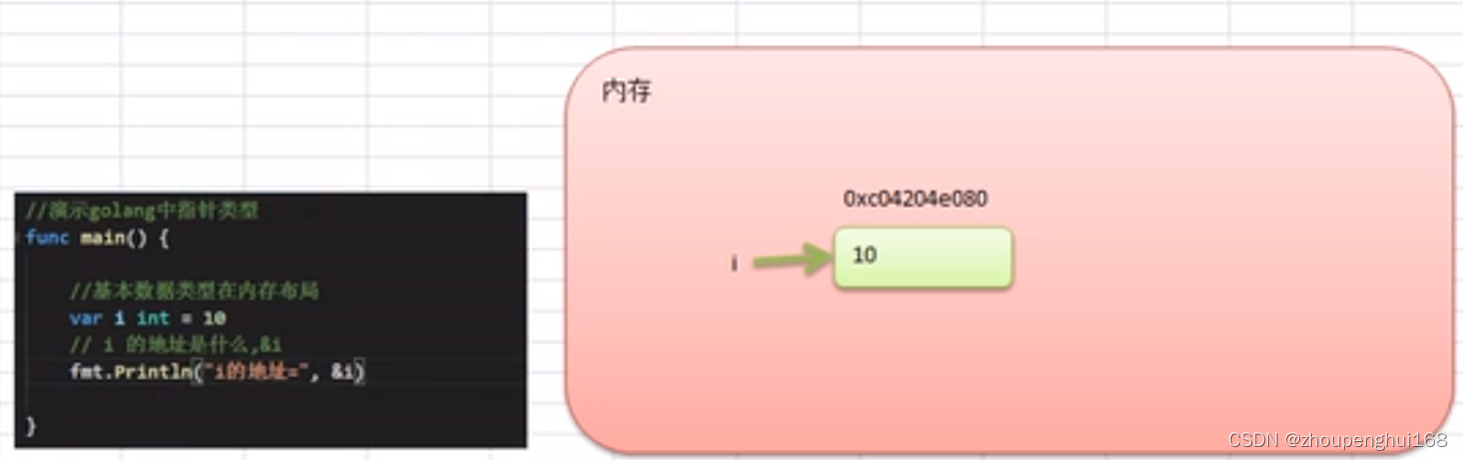
package main  
  
import (  
 "fmt"  
 "strconv"  
)  
  
func main() {  
 var str1 string = "true"  
 var b1 bool  
 //b1,\_ = strconv.ParseBool(str1)  
 //说明:  
 // 1. strconv.ParseBool(str1)函数会返回两个值(value bool,err error)  
 // 2.因为我们只想获取value bool,不想获取err,所以使用\_忽略  
 b1, \_ = strconv.ParseBool(str1)  
 fmt.Printf("b1 数据类型%T, b1=%t\n", b1, b1)  
 /\*  
 \*返回字符串表示的整数值，接受正负号。  
 \*base指定进制（2到36），如果base为0，则会从字符串前置判断，"0x"是16进制，"0"是8进制，否则是10进制；  
 \*bitSize指定结果必须能无溢出赋值的整数类型，0、8、16、32、64 分别代表 int、int8、int16、int32、int64；  
 \*返回的err是\*NumErr类型的，如果语法有误，err.Error = ErrSyntax；如果结果超出类型范围err.Error = ErrRange  
 \*/  
 var str2 string = "12345"  
 var num1 int64  
 num1, \_ = strconv.ParseInt(str2, 10, 64)  
 fmt.Printf("num1 数据类型%T, num1=%d\n", num1, num1)  
 // 因为返回的是int64或float64,如果希望得到int32,float32等,做如下处理:  
 var num2 int  
 num2 = int(num1)  
 fmt.Printf("num2 的数据类型:%T,num2=%d\n", num2, num2)  
  
 var str3 string = "10.23242"  
 var fl64 float64  
 fl64, \_ = strconv.ParseFloat(str3, 64)  
 fmt.Printf("fl64 的数据类型:%T,fl64=%f\n", fl64, fl64)  
}

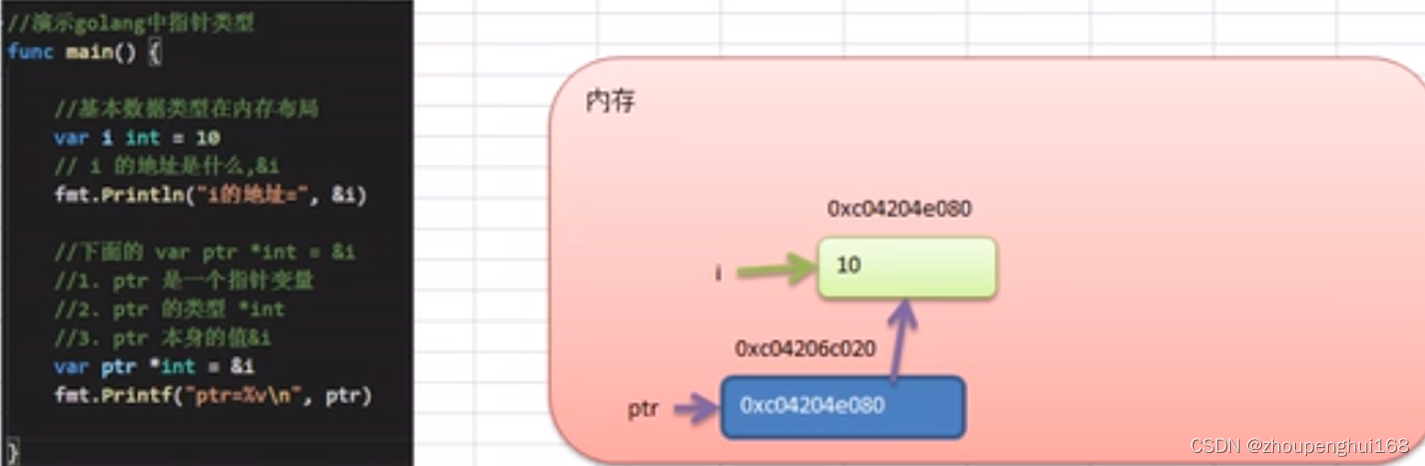
在将string类型转换成基本数据类型时,要确保string类型能够转换成有效的数据,如:我们可把"123"转换成一个整数,但不能把"hello"转换成一个整数,如果这样做,go会直接将其转换为0,其他类型也是一样的道理,float->0,bool->false

## 1.5 指针、值类型与引用类型

### 1.5.1 指针







package main  
  
import "fmt"  
  
func main() {  
 var i int = 10  
 fmt.Println("i 的地址是：", &i)  
  
 var ptr \*int = &i  
 fmt.Printf("ptr 的值:%v\n", ptr)  
 fmt.Printf("ptr 的类型:%T\n", ptr)  
 fmt.Printf("ptr 的地址:%v\n", &ptr)  
 fmt.Printf("ptr 指针对应的值:%v\n", \*ptr)  
}  
  
/\*  
i 的地址是： 0xc00001e068  
ptr 的值:0xc00001e068  
ptr 的类型:\*int  
ptr 的地址:0xc00000e030  
ptr 指针对应的值:10  
  
\*/

指针细节说明：

1).值类型,都有对应的指针类型,形式为 \*数据类型,eg: int的对应指针就是\*int,float32对应的指针类型为\*float32...

2).值类型包括:基本数据类型int系列,float系列,bool,string,数组和结构体struct

### 1.5.2 值类型和引用类型

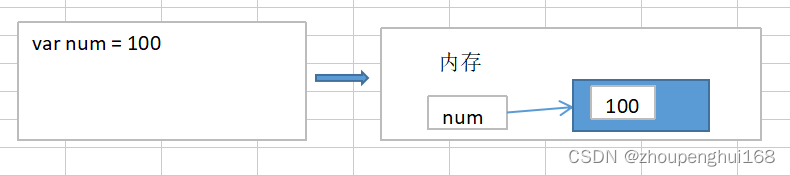
常见的值类型和引用类型

1).值类型:基本数据类型int系列,float系列,bool,string,数据和结构体struct

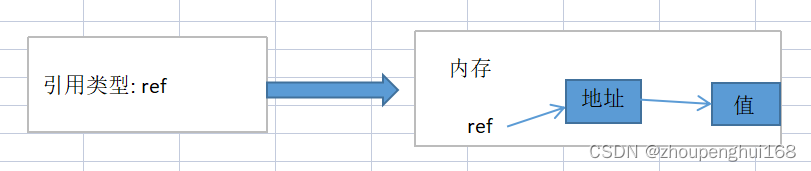
2).引用类型:指针,slice切片,map,channel管道,interface等都是引用类型

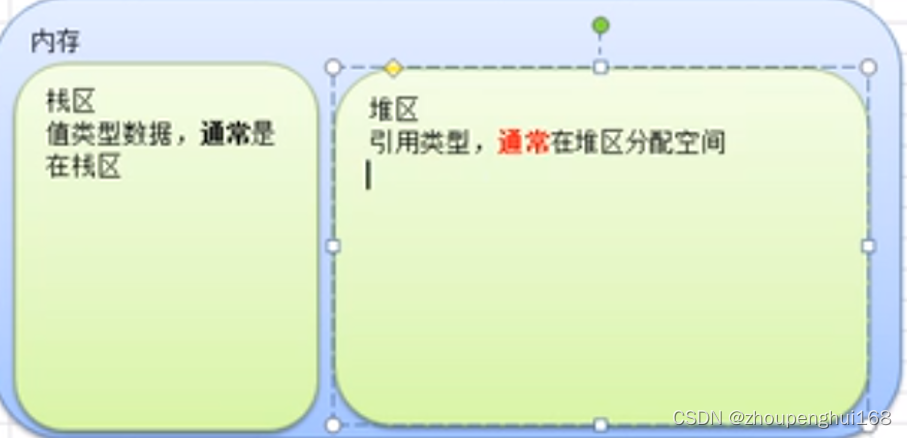
值类型和引用类型的使用特点：

1).值类型:变量直接存储值,内存通常在栈中分配



2).引用类型:变量存储的是一个地址,这个地址对应的空间才真正存储数据(值),内存通常在堆上分配,当没有任何变量引用这个地址时,该地址对应的数据空间就成为一个垃圾,会有GC来回收





## 1.6 标识符的命名规范

1).由26个英文字母大小写,0-9,\_组成

2).不能以数字开头

3).go语言中严格区分大小写

4).标识符不能包含空格

5).下划线" "本身在go中是一个特殊的标识符,称为空标识符.可以代表任何其他的标识符,但是它对应的值会被忽略(eg:忽略某个返回值),所以仅能被作为占位符使用

6).不能以系统保留关键字作为标识符,eg:break,if等等

注意事项：

1）包名:保持package的名字和目录保持一致,尽量采取有意义的的包名,简短,有意义,不要和标准库发生冲突

2）.变量名,函数名,常量名:采用驼峰法

3）.如果 变量名,函数名,常量名 首字母大写,则可以被其他包访问,如果首字母小写,则只能在本包中使用(可简单理解成:首字母大写是共公有的,首字母小写是私有的)

# 运算符

## 2.1算数运算符

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 运算 | 范例 | 结果 |
| + | 正号 | +3 | 3 |
| - | 负号 | -3 | -3 |
| + | 加号 | 5+5 | 10 |
| - | 减号 | 5-4 | 1 |
| \* | 乘号 | 2\*3 | 6 |
| / | 除号 | 5/2 | 2（取商去掉余数） |
| % | 取模（取余） | 5%2 | 1 |
| ++ | 自增 | a=2 a++ | 3 |
| -- | 自减 | a=2 a-- | 1 |
| + | 字符串相加 | “he” + “llo” | “hello” |

package main  
  
import "fmt"  
  
func main() {  
 // 1. 除法：如果运算数都是整数，那么除后，去掉小数部分，保留整数部分  
 fmt.Println(5 / 2) // 2  
  
 var f1 float64 = 5 / 2  
 fmt.Println(f1) // 2  
  
 // 如果希望保留小数部分，则需要有浮点数参与运算  
 fmt.Println(5.0 / 2) // 2.5  
  
 // 2. %  
 fmt.Println(5 % 2) // 1  
  
 // 3. ++ 和 -- 使用  
 n1 := 10  
 n1++  
 fmt.Println(n1) // 11  
 n1--  
 fmt.Print(n1) // 10  
}

细节说明

1).对于除号"/",它的整数除和小数除是有区别的:整数之间做除法时,只保留整数部分而舍弃小数部分,eg: x := 15/4,结果是3

2).当对一个数取模时,可以等价 a%b = a - a / b \* b, 这样我们可以看到取模的一个本质运算

3).go的自增自减只能当做一个独立的语言来使用是,不能这样使用: b := a++ 或者 b := a--

4).go的++,--只能写在变量后面,不能写在变量前面,即:只有a++,a--,没有++a,--a

5).go的设计者去掉java/c中的自增自减的容易混淆的写法,让go更加简洁统一

## 2.2.关系运算符

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

1).关系运算符的结果都是bool型,要么true,要么false

2).关系表达式经常使用在if结构的条件中或循环结构的条件中

细节说明:

1).关系运算符的结果都是bool型, 要么true,要么false

2).关系运算符组成的表达式,称之为关系表达式: a > b

3).比较运算符 "==" 不能写成 "="

package main  
  
import "fmt"  
  
func main() {  
 a := 1  
 b := 2  
 flag := a > b  
  
 fmt.Println(flag) // false  
 fmt.Println(int(flag)) // error cannot convert flag (variable of type bool) to type int  
}

## 2.3.逻辑运算符

下表列出了所有Go语言的[逻辑运算符](https://so.csdn.net/so/search?q=%E9%80%BB%E8%BE%91%E8%BF%90%E7%AE%97%E7%AC%A6&spm=1001.2101.3001.7020" \t "_blank)。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 |

注意事项

1).&&也叫短路与,如果第一个条件为false,则第二个条件不会判断,最终结果为false

2).||也叫短路或,如果第一个添条件为true,则第二个条件不会判断,最终结果为true

//案例演示  
package main  
  
import "fmt"  
  
//声明一个函数  
func test() bool {  
 fmt.Println("test...")  
 return true  
}  
  
func main() {  
 var i int = 10  
 //短路与  
 //说明: i < 9 为 false,故后面的test()就不会执行  
 if i < 9 && test() {  
 fmt.Println("ok...")  
 }  
 //短路或  
 //说明: i > 9 为 true,故后面的test()就不会执行  
 if i > 9 || test() {  
 fmt.Println("ok...")  
 }  
}

## 2.4.位运算符

位运算符对整数在内存中的二进制位进行操作

下表列出了位运算符 &, |, 和 ^ 的计算：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| p | q | p&q 与，只有两边都为1，结果为1 | p|q 或，有一边为1，结果为1 | p^q 异或，相同为0，不同为1 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

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 |

### 2.4.1 进制

对于整数而言,有四种表示方式

1).二进制: 0,1.逢2进1

2).八进制:0-7,逢8进1,以数字0开头表示

3).十进制:0-9,逢10进1

4).十六进制:0-9以及A-F,逢16进1,以0x或者0X开头表示,eg:0x21AF + 1 = 0X21B0

package main  
  
import "fmt"  
  
func main() {  
  
 var i int = 5  
 fmt.Printf("%b\n", i)  
  
 var j int = 011  
 fmt.Println(j)  
  
 var k int = 0x11  
 fmt.Println(k)  
}  
  
/\*  
101  
9  
17  
  
\*/

1).二进制转十进制

规则: 从最低位开始(右边的),将每个位上的数提取处出来,乘以2的(位数-1)次方,然后求和

eg:1011 = 1 \* 1 + 1 \* 2 + 0 \* 2 \* 2 + 1 \* 2 \* 2 \* 2 = 1 + 2 + 0 + 8 = 11

2).八进制转十进制

规则: 从最低位开始(右边的),将每个位上的数提取处出来,乘以8的(位数-1)次方,然后求和

eg:0123 = 3 \* 1 + 2 \* 8 + 1 \* 8 \* 8 = 3 + 16 + 64 = 83

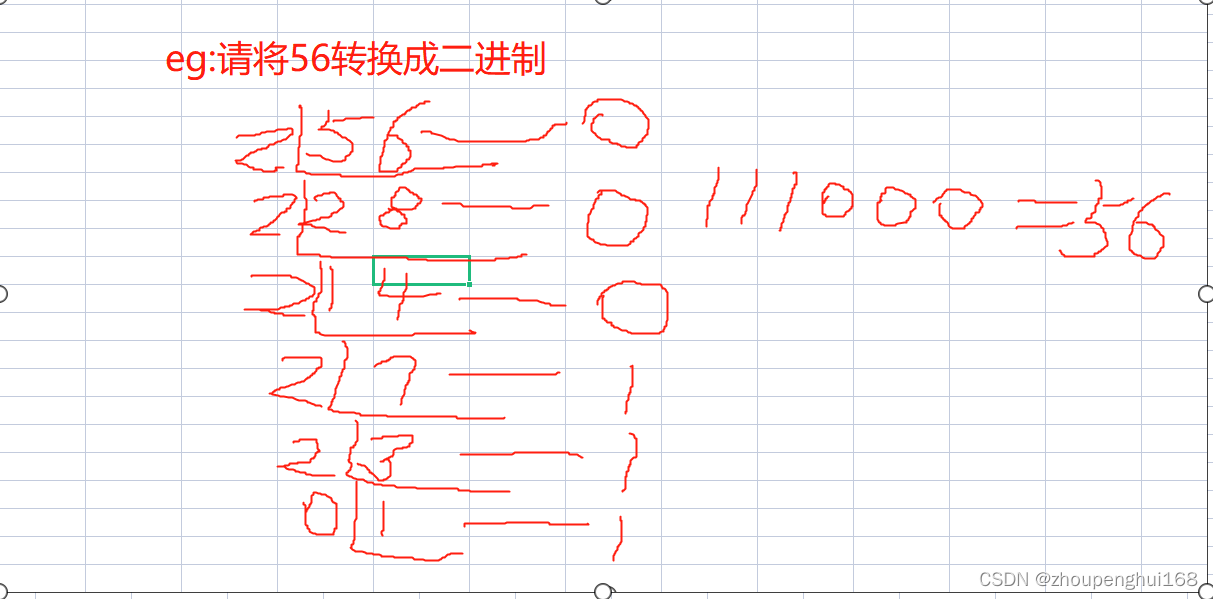
3).十六进制转十进制

规则: 从最低位开始(右边的),将每个位上的数提取处出来,乘以16的(位数-1)次方,然后求和

eg:0x34A = 10 \* 1 + 4 \* 16+ 3 \* 16 \* 16 = 10 + 64 + 768 = 842

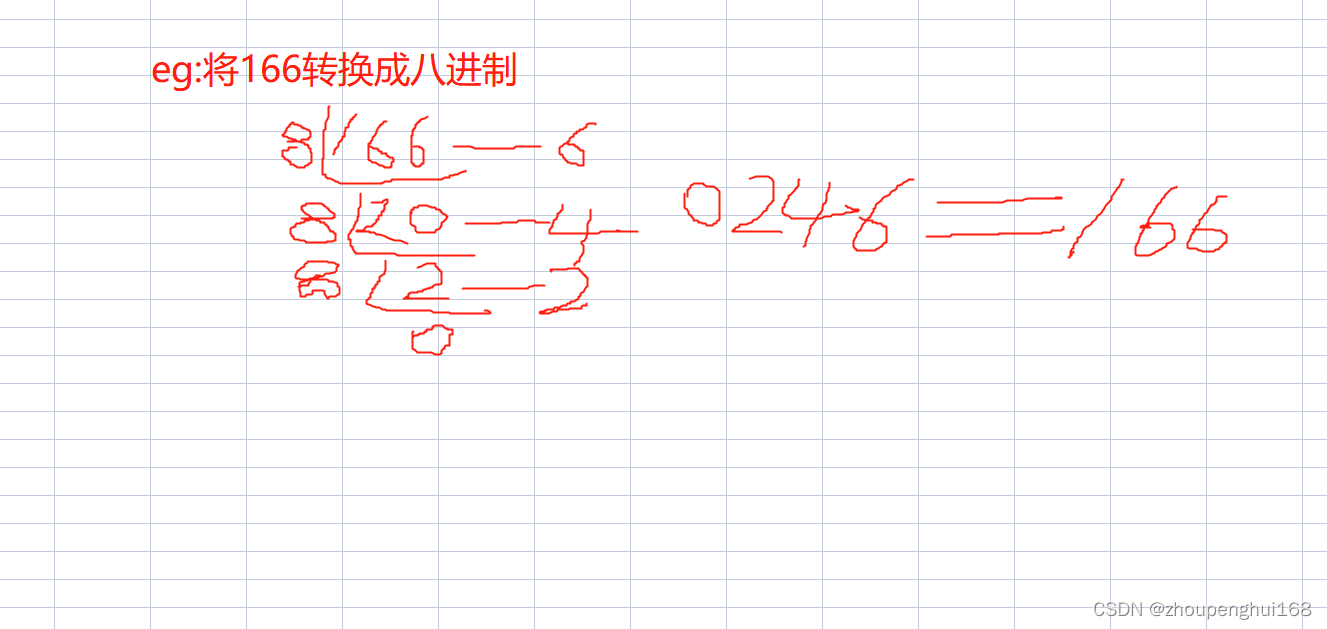
1.十进制转换成二进制:

        规则:将该数不断除以2,直到商为0为止,然后将每步得到的余数倒过来,就是对应的二进制



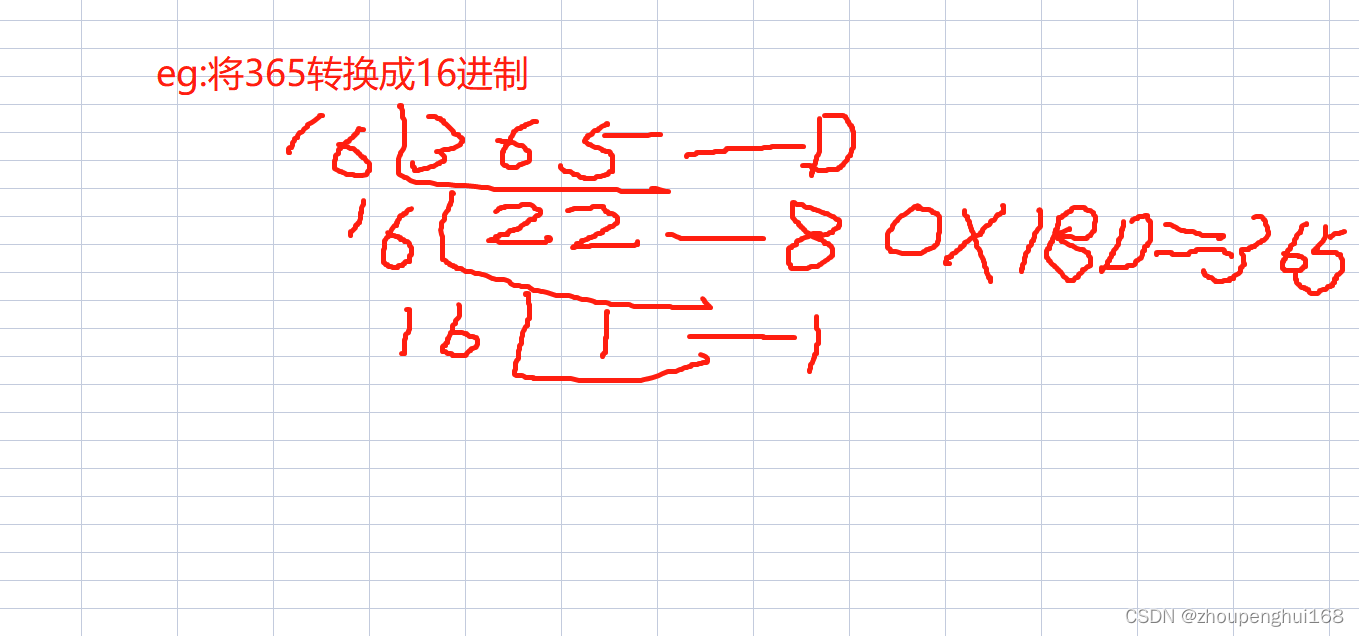
2.十进制转换成八进制

 规则:将该数不断除以8,直到商为0为止,然后将每步得到的余数倒过来,就是对应的八进制



3.十进制转换成十六进制

 规则:将该数不断除以16,直到商为0为止,然后将每步得到的余数倒过来,就是对应的十六进制



4. 二进制转换成八进制,十六进制

1）二进制转换成八进制

规则:将二进制每三位数一组(从低位开始组合),转成对应的八进制数即可

eg:11010101 = 0325, 11100101 = 345

2）二进制转换成十六进制

规则:将二进制每四位数一组(从低位开始组合),转成对应的十六进制数即可

eg:11010101 = 0xD5, 1110010110 = 196

5. 八进制,十六进制转换成二进制

1）八进制转换成二进制

规则:将八进制数每1位,转换成对应的一个3位的二进制即可

eg:0237 = 10011111

2）十六进制转换成二进制

规则:将十六进制数每1位,转换成对应的一个4位的二进制即可

eg:0x237 = 1000110111

### 2.4.2 位运算符、原码、反码、补码

对于有符号的而言

1).二进制的最高位是符号位:0表示正数,1表示负数

1 ===> [00000001] -1 ===> [10000001]

2).正数的原码,反码,补码都一样

1 ===> 原码[0000001],反码[00000001],补码[00000001]

3).负数的反码 = 它的原码符号位不变,其它位取反(0 ==> 1, 1 ==> 0)

负数的补码 = 负数的反码 + 1

-1 ===>原码[10000001],反码[11111110],补码[11111111]

4).0的反码,补码都是0

5).在计算机运算的时候,都是以补码的方式来运算的

1 - 1 = 1 + (-1)

位运算符：

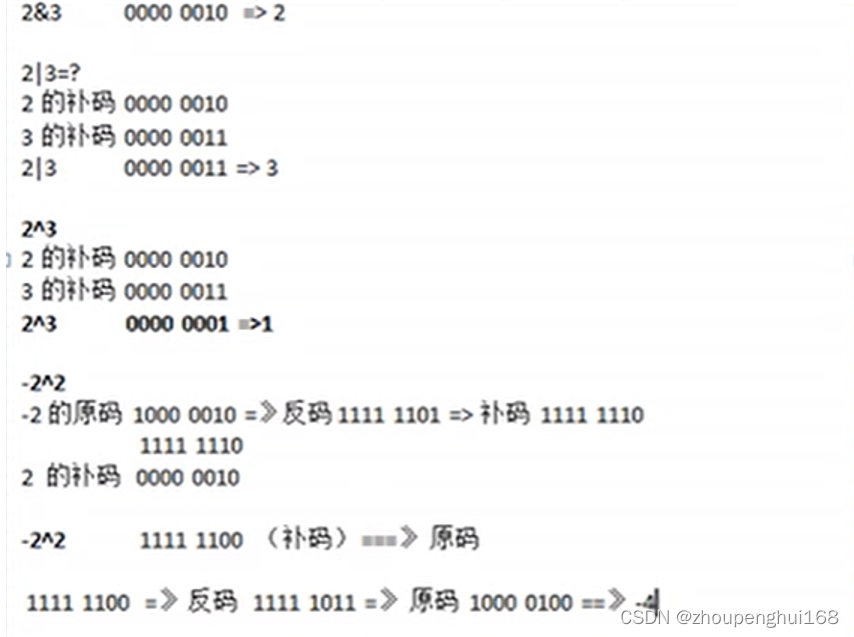
go中有三个位运算符,分别是 按位与&,按位或|,按位异或^, 它们的运算规则是:

按位与&: 两位全为1,结果为1,否则为0

按位或|:两位有一个为1,结果为1,否则为0

按位异或^:两位一个为0,一个为1,结果为1,否则为0(相异为1)

eg: 2&3=? 2|3=? 2^3=?



>>右移,<<左移,运算规则:

右移运算符>>:低位溢出,符号位不变,并用符号位补溢出的高位

左移运算符<<:符号位不变,低位补0

a := 1 >>2 ==> 0000 0001 ==> 0000 0000 ==> 0

b := 1 << 2 ==> 0000 0001 ==> 0000 0100 ==> 4

## 2.5.赋值运算符

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 |

赋值运算符的特点

1).运算顺序从左至右

2).赋值运算符的左边只能是变量,右边可以是变量,表达式,常量值

3).复合赋值运算符等价于下面效果,eg: a += 1 ==> a = a + 1

4). c = a + 3 //赋值运算的顺序是从右到左

**2.6.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 |

## 2.7.运算符优先级

所谓优先级，就是当多个运算符出现在同一个表达式中时，先执行哪个运算符，比如：a + b \* c，先计算乘法后计算加法，说明乘法运算符的优先级比加法运算符的优先级高，注意：下表中，优先级值（数值越大例如14号优先级最高）越大，表示优先级越高

|  |  |  |  |
| --- | --- | --- | --- |
| 优先级 | 分类 | 运算符 | 结合性 |
| 1 | 逗号运算符 | , | 从左到右 |
| 2 | 赋值运算符 | =、+=、-=、\*=、/=、 %=、 >=、 <<=、&=、^=、|= | 从右到左 |
| 3 | 逻辑或 | || | 从左到右 |
| 4 | 逻辑与 | && | 从左到右 |
| 5 | 按位或 | | | 从左到右 |
| 6 | 按位异或 | ^ | 从左到右 |
| 7 | 按位与 | & | 从左到右 |
| 8 | 相等/不等 | ==、!= | 从左到右 |
| 9 | 关系运算符 | <、<=、>、>= | 从左到右 |
| 10 | 位移运算符 | <<、>> | 从左到右 |
| 11 | 加法/减法 | +、- | 从左到右 |
| 12 | 乘法/除法/取余 | \*（乘号）、/、% | 从左到右 |
| 13 | 单目运算符 | !、\*（指针）、& 、++、--、+（正号）、-（负号） | 从右到左 |
| 14 | 后缀运算符 | ( )、[ ]、-> | 从左到右 |

总结优先级：

单目运算符 > 算术运算符 > 位移运算符 > 关系运算符 > 位运算符 > 逻辑运算符 > 赋值运算符，其中，将 “位运算符” 大分支 分为了 位移运算符 <<、>> 和 位运算符 |、^、&

# 流程控制

# 函数、包、错误处理

# 数组和切片

# 排序和查找

# map

# 面向对象编程

# 小项目案例

# 文件操作

# 单元测试

# 协程和管道

# 反射和常量

# TCP编程

# redis使用

# 数据结构

# Web开发基础