# Go基本语法

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
* 数据类型
* 打印格式化
* 数据类型转换
* 常量与枚举
* Go语言运算符
* 运算符优先级

## 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】 |
| 2 | uint16 无符号 16 位整型 (0 到 65535) 【2的16次方】 |
| 3 | uint32 无符号 32 位整型 (0 到 4294967295) 【2的32次方】 |
| 4 | uint64 无符号 64 位整型 (0 到 18446744073709551615) 【2的64次方】 |
| 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 |

### 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

### 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。

### 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。

● 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'

## 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]

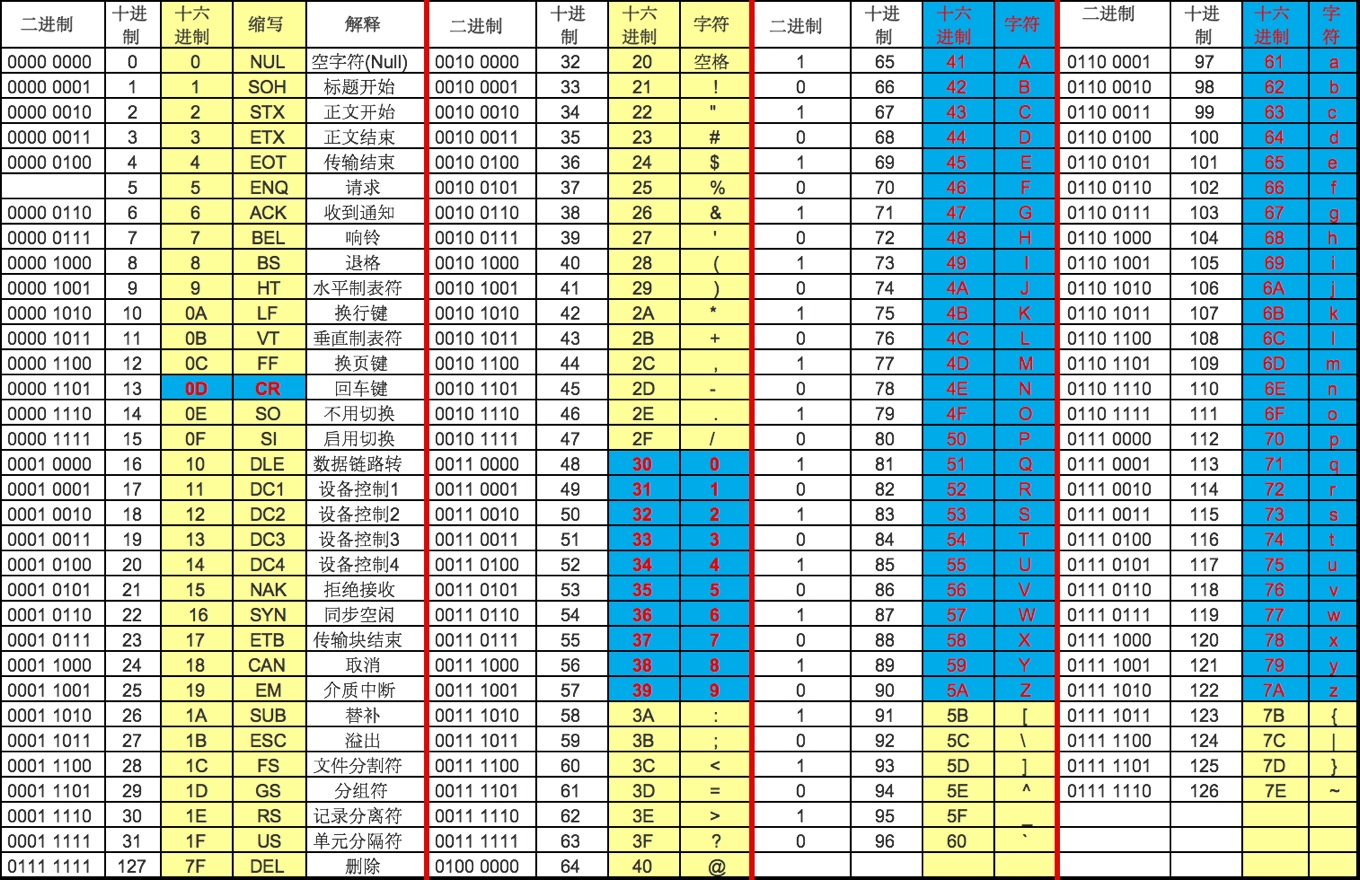


图 1

### 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

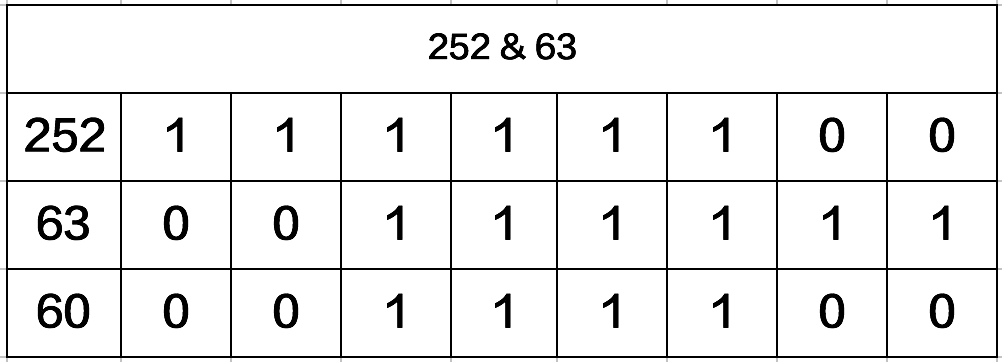
A^B = 0011 0001

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 |

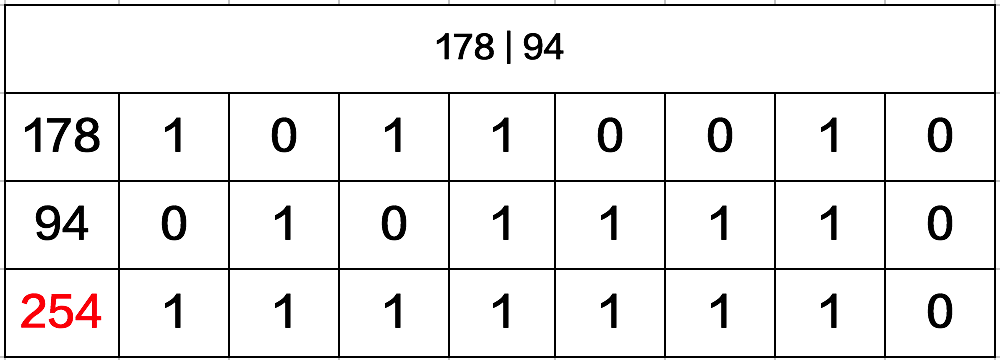
1、按位与

● 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。



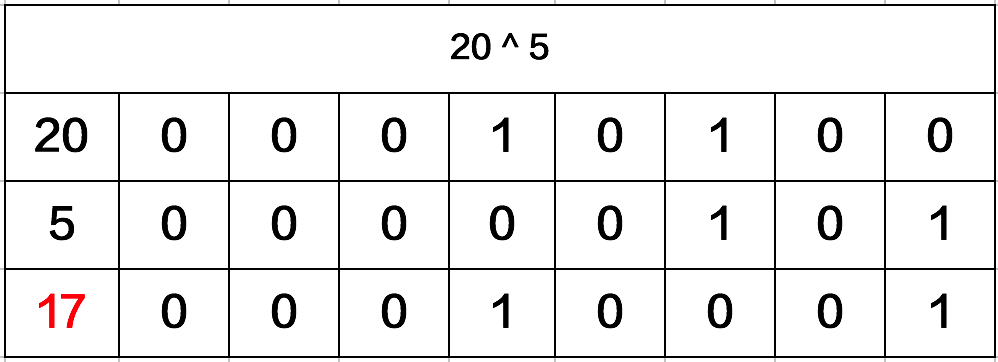
2、按位或

● 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

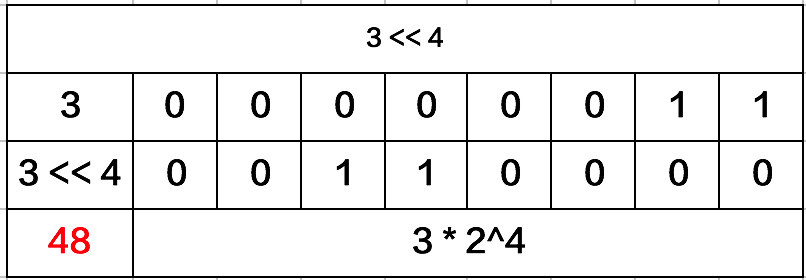
（2）、计算过程：

● 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

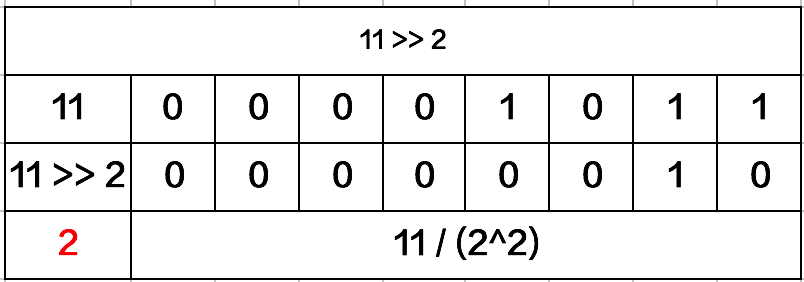
● 例如11 >> 2，则是将数字11右移2位

（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