

# 语法基础

因为Go官方建议使用最新稳定版本，很多库也是这样做的。我们教学也采用最新稳定版，本次使用Go 1.20.x版本。

## 注释

- `//` 单行注释
- `/* xxxx */` 编译器忽略该区间，其间都被认为是注释内容。虽然Go支持，但很少使用

```
1 // 这是包注释
2
3 package main
4
5 import "fmt"
6
7 /*
8     x int
9     y int
10    returns: int
11    函数说明
12 */
13 func add(x, y int) int {
14     return x + y
15 }
16
17 // 函数注释也可以这样多行
18 // 写在上面
19 func main() {
20     fmt.Println(add(4, 5)) // 打印
21     // TODO 之后完成某某功能
22 }
```

```
1 // TODO: 将来完成，推荐
2 // NOTE: 请注意
3 // Deprecated: 告知已经过期，建议不要使用。未来某个版本可能移除
```

- 函数、结构体等习惯把注释写在上面
- 包注释会写在package之上

## 行

Go语言把行分隔符作为一条语句的结尾。也就是说，一般情况下，一行结束，敲回车即可。

## 命名规范

- 标识符采用CamelCase驼峰命名法
  - 如果只在包内可用，就采用小驼峰命名
  - 如果要在包外可见，就采用大驼峰命名
- 简单循环变量可以使用i、j、k、v等
- 条件变量、循环变量可以是单个字母或单个单词，Go倾向于使用单个字母。Go建议使用更短小
- 常量驼峰命名即可
  - 在其他语言中，常量多使用全大写加下划线的命名方式，Go语言没有这个要求
  - 对约定俗成的全大写，例如PI
- 函数/方法的参数、返回值应是单个单词或单个字母
- 函数可以是多个单词命名
- 类型可以是多个单词命名
- 方法由于调用时会绑定类型，所以可以考虑使用单个单词
- 包以小写单个单词命名，包名应该和导入路径的最后一段路径保持一致
- 接口优先采用单个单词命名，一般加er后缀。Go语言推荐尽量定义小接口，接口也可以组合

## 关键字

<https://golang.google.cn/ref/spec>

1	<code>break</code>	<code>default</code>	<code>func</code>	<code>interface</code>	<code>select</code>
2	<code>case</code>	<code>defer</code>	<code>go</code>	<code>map</code>	<code>struct</code>
3	<code>chan</code>	<code>else</code>	<code>goto</code>	<code>package</code>	<code>switch</code>
4	<code>const</code>	<code>fallthrough</code>	<code>if</code>	<code>range</code>	<code>type</code>
5	<code>continue</code>	<code>for</code>	<code>import</code>	<code>return</code>	<code>var</code>

## 预定义标识符

[https://golang.google.cn/ref/spec#Predeclared\\_identifiers](https://golang.google.cn/ref/spec#Predeclared_identifiers)

```
1 Types:
2     any bool byte comparable
3     complex64 complex128 error float32 float64
4     int int8 int16 int32 int64 rune string
5     uint uint8 uint16 uint32 uint64 uintptr
6
7 Constants:
8     true false iota
9
10 Zero value:
11     nil
12
13 Functions:
14     append cap close complex copy delete imag len
15     make new panic print println real recover
```

## 标识符

- 一个名字，本质上是字符串，用来指代一个值
- 只能是大小写字母、数字、下划线，也可以是Unicode字符
- 不能以数字开头
- 不能是Go语言的关键字
- 尽量不要使用“预定义标识符”，否则后果难料
- 大小写敏感

标识符建议：

- 不要使用中文
- 非必要不要使用拼音
- 尽量遵守上面的命名规范，或形成一套行之有效的命名规则

## 字面常量

它是值，不是标识符，但本身就是常量，不能被修改。

Go语言中，boolean、rune、integer、float、complex、string都是**字面常量**。其中，rune、integer、float、complex常量被称为数值常量。

```
1 100
2 0x6162 0x61_62_63
3 3.14
4 3.14e2
5 3.14E-2
6
7 '测'
8 '\u6d4b'
9 '\x31'
10 '1'
11 '\n'
12
13 "abc" "\x61b\x63"
14 "测试" "\u6d4b试"
15 "\n"
16
17 true
18 false
19 iota
```

以上字面常量在Go中也被称为无类型常量untyped constant。无类型常量的缺省类型为bool、rune、int、float64、complex128或字符串。

## 常量

常量：使用const定义一个标识符，它所对应的值，不允许被修改。

对常量并不要求全大写加下划线的命名规则。

```

1 // 赋值等式右边用“无类型常量untyped constant”来赋值
2 const a int = 100 // 指定类型定义常量并赋值
3 const (           // 定义常量，等式左边未给出类型，将进行类型推导
4     b = "abc"
5     c = 12.3
6     d = 'T'
7 )

```

```

1 const a           // 错误，const定义常量，必须在定义时赋值，并且之后不能改变
2
3 const c = [2]int{1, 2} // 错误，数组的容器内容会变化，不能在编译期间明确地确定下来，这
    和其它语言不一样

```

**注意：**Go语言的常量定义，必须是能在编译器就要完全确定其值，所以，值只能使用**字面常量**。这和其他语言不同！例如，在其他语言中，可以用常量标识符定义一个数组，因为常量标识符保证数组地址不变，而其内元素可以变化。但是Go根本不允许这样做。

## iota

Go语言提供了一个预定义标识符iota[ar'ou.tə]，非常有趣。

```

1 // 单独写iota从0开始
2 const a = iota // 0
3 const b = iota // 0

```

批量定义写在括号里，以定义星期常量为例

```

1 // 批量写iota从0开始
2 const (
3     SUN = iota // 0
4     MON = iota // 1
5     TUE = iota // 2
6 )
7
8 // 简化
9 const (
10     SUN = iota // 0
11     MON
12     TUE
13 )

```

```

1 // 比较繁琐的写法，仅作测试
2 // 批量写iota从0开始，即使第一行没有写iota，iota也从第一行开始从0开始增加
3 const (
4     a = iota // 0
5     b        // 1
6     c        // 2
7     _        // 按道理是3，但是丢弃了
8     d        // 4
9     e = 10   // 10
10    f        // 10

```

```

11     g = iota // 7
12     h        // 8
13 )
14 // 可以认为Go的const批量定义实现了一种重复上一行机制的能力

```

```

1 // 批量写iota从0开始，智能重复上一行公式
2 const (
3     a = 2 * iota // 0
4     b             // 2
5     c             // 4
6     d             // 6
7 )

```

```

1 const (
2     m = 0
3     n
4     a = 2 * iota
5     b
6     c
7     d
8 )
9 fmt.Println(a, b, c, d, m, n) // 这些常量的值是多少？

```

## 变量

变量：赋值后，可以改变值的标识符。

建议采用驼峰命名法。

```

1 var a        // 错误，无法推测类型
2 var b int    // 正确，只声明，会自动赋为该类型的零值
3 var c, d int // 正确，声明连续的同类型变量，可以一并声明，会自动赋为该类型的零值
4 var b = 200  // 错误，b多次声明，第二行已经声明过了

```

```

1 // 初始化：声明时一并赋初值
2 var a int = 100 // 正确，标准的声明，并初始化
3 var b = 200     // 正确，编译根据等式右值推导左边变量的类型
4 var c = nil     // 错误，非法，nil不允许这样用
5 var d, e int = 11, 22 // 正确，var定义多个变量只能是同类型且类型只能写在最后。详见下面批量赋值

```

```

1 // 用var声明，立即赋值，或之后赋值
2 var b int // 正确，只声明，会自动赋为该类型的零值
3 b = 200
4 b = 300
5 b = "4" // 错误，类型错误

```

```

1 // 批量赋值
2 var a int, b string // 错误，批量不能这么写
3 var ( // 正确，a、b类型知道了，可以用零值
4     a int
5     b string
6 )
7 var ( // 错误，变量必须有类型，但没有给类型，也不能使用值来推导类型
8     a
9     b
10 )
11
12 var a int, b string = 111, "abc" // 错误，多种类型不能这么写，语法不对
13 var (
14     a int    = 111
15     b string = "abc"
16 ) // 正确，建议批量常量、变量都这么写
17 var (
18     a = 111
19     b = "abc"
20 ) // 正确，类型推导

```

```

1 // 短格式 short variable declarations
2 // _ 空白标识符，或称为匿名变量
3 a := 100
4
5 b, c := 200, "xyz"
6
7 // 交换
8 b, c = c, b
9
10 d, _, f := func() (int, string, bool) { return 300, "ok", true }()

```

`_`下划线 是空白标识符 (Blank identifier) ,

- [https://golang.google.cn/ref/spec#Declarations\\_and\\_scope](https://golang.google.cn/ref/spec#Declarations_and_scope)
- [https://golang.google.cn/ref/spec#Blank\\_identifier](https://golang.google.cn/ref/spec#Blank_identifier)
- 下划线和其他标识符使用方式一样，但它不会分配内存，不占名词空间
- 为匿名变量赋值，其值会被抛弃，因此，后续代码中不能使用匿名变量的值，也不能使用匿名变量为其他变量赋值

## 短格式

- 使用 `:=` 定义变量并立即初始化
- 只能用在函数中，**不能用来定义全局变量**
- 不能提供数据类型，由编译器来推断

## 零值

变量已经被声明，但是未被显式初始化，这是变量将会被设置为零值。其它语言中，只声明未初始化的变量误用非常危险，但是，Go语言却坚持“**零值可用**”理念。在Go语言中合理利用零值确实带来不小的便利，这在后面的课程中大家可以慢慢体会。

- int为0
- float为0.0
- bool为false
- string为空串""（注意是双引号）
- 指针类型为nil
- 其它类型数据零值，学到再说

## 标识符本质

每一个标识符对应一个具有数据结构的值，但是这个值不方便直接访问，程序员就可以通过其对应的标识符来访问数据，标识符就是一个指代。一句话，标识符是给程序员编程使用的。

## 变量可见性

### 1、包级标识符

在Go语言中，在.go文件中的顶层代码中，定义的标识符称为**包级标识符**。如果首字母大写，可在包外可见。如果首字母小写，则包内可见。

```
1 // 无类型常量定义
2 var a = 20 // int
3 var b = 3.14 // float64
4 // 指定类型
5 var a int32 = 20
6 var b float32 = 3.14
```

```
1 // 延迟初始化需要指定类型，用零值先初始化。因为若不给出类型，不知道用什么类型的零值
2 // 有相同关系的声明可以使用同一批定义
3 var (
4     name string
5     age int
6 )
```

### 使用建议

- 顶层代码中定义包级标识符
  - 首字母大写作为包导出标识符，首字母小写作为包内可见标识符
  - const定义包级常量，必须在声明时初始化
  - var定义包级变量
    - 可以指定类型，也可以使用无类型常量定义
    - 延迟赋值必须指定类型，不然没法确定零值
- 有相关关系的，可以批量定义在一起
- 一般声明时，还是考虑“就近原则”，尽量靠近第一次使用的地方声明

- 不能使用短格式定义

## 2、局部标识符

定义在函数中，包括main函数，这些标识符就是**局部标识符**。

使用建议

- 在函数中定义的标识符
- const定义局部常量
- var定义局部变量
  - 可以指定类型，也可以使用无类型常量定义
  - 延迟赋值必须指定类型，不然没法确定零值
- 有相关关系的，可以批量定义在一起
- 在函数内，直接赋值的变量多采用短格式定义

## 布尔型

类型bool，定义了2个预定义常量，分别是true、false。

## 数值型

[https://golang.google.cn/ref/spec#Numeric\\_types](https://golang.google.cn/ref/spec#Numeric_types)

复数：complex64、complex128

## 整型

- 长度不同：int8、int16（C语言short）、int32、int64（C语言long）
  - 最高位是符号位
  - rune类型本质上就是int32
- 长度不同无符号：uint8、uint16、uint32、uint64
  - u即unsigned，最高位不是符号位
  - byte类型，它是uint8的别名
- 自动匹配平台：int、uint
  - int类型它至少占用32位，但一定注意它不等同于int32，不是int32的别名。要看CPU，32位就是4字节，64位就是8字节。但是，也不是说int是8字节64位，就等同于int64，它们依然是**不同类型**！

## 扩展知识

计算机内部是使用二进制的，也就是只有0和1，那如何表达出正负数的区别呢？（为了让初学者便于理解，不引入补码表示法）

为了方便以单字节为例，1个字节是8位，那么可以表示256种状态。

如何表示十进制2呢？它是00000010。那十进制-2，如何表示？

那就把最高位单独解释，最高位定为符号位，0表示正号，1表示负号。 -2用二进制表示为10000010。



## 进制表示

- 十六进制: 0x10、0X10
- 八进制: 0o10、0O10。010也行, 但不推荐
- 二进制: 0b10、0B10

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var a = 20
7     b := 30
8     var c int = 40
9     fmt.Printf("%T, %T, %T, %d\n", a, b, c, a+b+c)
10    var d int64 = 0x32 // 50
11    fmt.Printf("%T, %d\n", d, d)
12    fmt.Println(a + d) // 错误, int和int64类型不同不能操作
13    fmt.Println(a + int(d)) // 显示强制类型转换才行
14 }
```

与其他语言不同, 即使同是整型这个大类中, 在Go中, 也不能跨类型计算。如有必要, 请强制类型转换。

**强制类型转换:** 把一个值从一个类型强制转换到另一种类型, 有可能转换失败。

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var d int64 = 50
7     fmt.Printf("%T, %d\n", d, d)
8     fmt.Printf("%T, %s; %T, %d; %T, %f; %T %d\n", string(d), string(d),
9     rune(d), rune(d), float32(d), float32(d), byte(d), byte(d))
10 }
11
12 输出如下
13 int64, 50
14 string, 2; int32, 50; float32, 50.000000
```

## 字符和整数

字符表达, 必须使用**单引号**引住一个字符。

```
1 type rune = int32 // rune是int32的别名, 4个字节, 可以是Unicode字符
2 type byte = uint8 // byte是uint8的别名, 1个字节
```

```

1 var c rune = '中' // 字符用单引号。想一想，计算机中有中文吗？
2 fmt.Printf("%T, %c, %d\n", c, c, c) // int32, 中, 20013
3 c = 'a'
4 fmt.Printf("%T, %c, %d\n", c, c, c) // int32, a, 97
5
6 var d byte = '中' // 错误，超出byte范围
7
8 var d byte = '\x61'
9 fmt.Printf("%T, %c, %d\n", d, d, d)
10
11 var e rune = 20013
12 fmt.Printf("%T, %c, %d\n", e, e, e)

```

特别注意：字符串是若干个字符组成，在内存中使用utf-8编码。而rune只能保存一个字符，输出是unicode。

## 浮点数

- float32：最大范围约为3.4e38，通过math.MaxFloat32查看
- float64：最大范围约为1.8e308，通过math.MaxFloat64查看
- 打印格式化符常用%f

```

1 // fmt的格式化，参考包帮助 https://pkg.go.dev/fmt
2 f := 12.15
3 fmt.Printf("%T, %f\n", f, f) // 默认精度6
4 fmt.Printf("%.3f\n", f)      // 小数点后3位
5 fmt.Printf("[%3.2f]\n", f)    // 宽度撑爆了，中括号加上没有特殊含义，只是为了看清楚占的
                               // 打印宽度
6 fmt.Printf("[%6.2f]\n", f)    // 宽度为6
7 fmt.Printf("[% -6.2f]\n", f)  // 左对齐

```

## 进制及转换

常见进制有二进制、八进制、十进制、十六进制。应该重点掌握二进制、十六进制。

十进制逢十进一；十六进制逢十六进一；二进制逢二进一

```

1 每8位（bit）为1个字节（byte）。
2
3 一个字节能够表示的整数的范围：
4 无符号数0~0xFF，即0到255，256种状态
5 有符号数，依然是256种状态，去掉最高位还剩7位，能够描述的最大正整数为127，那么负数最大就
   为-128。也就是说负数有128个，正整数有127个，加上0，共256种。

```

```

1 转为十进制——按位乘以权累加求和
2 ^ 在这里表示求幂，2^3表示2的3次方
3
4 0b1110 计算为 1 * (2^3) + 1 * (2^2) + 1 * (2^1) + 0 * (2^0) = 14
5 0o664 计算为 6 * (8^2) + 6 * (8^1) + 4 * (8^0) = 436
6 0x41 计算为 4 * 16 + 1 * 1 = 65

```

- 1 十六进制中每4位断开转换
- 2 1000 0000 二进制  $2^7 = 128$
- 3 8 0 十六进制  $8 * (16^1) = 128$
- 4 八进制每3位断开转换
- 5 10 000 000 二进制  $2^7 = 128$
- 6 2 0 0 八进制  $2 * (8^2) + 0 + 0 = 128$

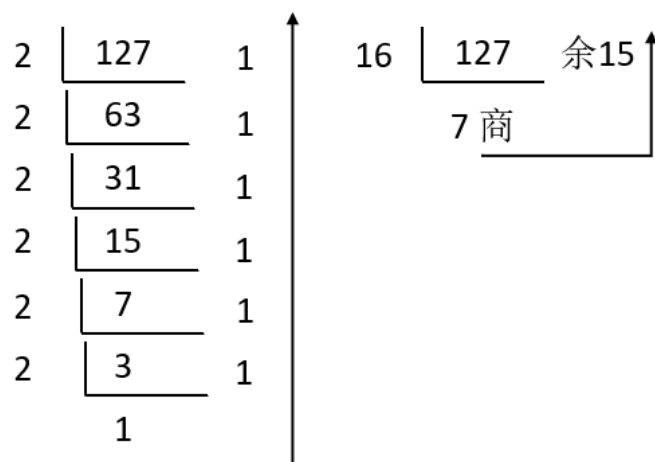
特殊二进制数0b	十进制	十六进制0x
1	1	1
11	3	3
111	7	7
1111	15	F
11111	31	1F
111111	63	3F
1111111	127	7F
11111111	255	FF
100000000	256	100

- 二进制中最低位为1，一定是奇数；最低位为0，一定是偶数

特殊十六进制值0x	十进制
9	9
A	10
D	13
20	32
30	48
31	49
41	65
61	97
7F	127
FF	255
FE	254

- 1 十六进制转为二进制
- 2 **0xF8** 按位展开即可，得到 **0b1111 1000**
- 3
- 4 八进制转为二进制
- 5 **0o664** 按位展开即可，得到**0b 110 110 100**

- 1 十进制转二进制
- 2 **127** 除以基数**2**，直到商为**0**为止，反向提取余数
- 3 尝试将十进制**5**、**12**转换为二进制
- 4
- 5 转为十六进制
- 6 **127** 除以基数**16**，直到商为**0**为止，反向提取余数



## 转义字符

每一个都是一个字符，`rune`类型。可以作为单独字符使用，也可以作为字符串中的一个字符。

- 1 `\a` U+0007 alert or bell
- 2 `\b` U+0008 backspace
- 3 `\f` U+000C form feed
- 4 `\n` U+000A line feed or newline
- 5 `\r` U+000D carriage return
- 6 `\t` U+0009 horizontal tab
- 7 `\v` U+000B vertical tab
- 8 `\\` U+005C backslash
- 9 `\'` U+0027 single quote (valid escape only within rune literals)
- 10 `\"` U+0022 double quote (valid escape only within string literals)

## 字符串

使用双引号或反引号引起来的任意个字符。它是字面常量。

```

1  "abc测试"      // 不能换行，换行需要借助\n
2  "abc\n测试"    // 换行
3
4  `abc
5      测试`      // 等价下面的字符串
6  "abc\n\t测试"
7
8  `json:"name"`  // 字符串里面如果有双引号，使用反引号定义方便
9  "json:\`"name\`" // 和上一行等价
10
11 "abc" + "xyz"  // 拼接

```

注意，反引号内不支持转义字符

## 字符串格式化

格式符参考fmt包帮助 <https://pkg.go.dev/fmt>

- %v 适合所有类型数据，调用数据的缺省打印格式
  - %+v 对于结构体，会多打印出字段名
- %#v 对于结构体，有更加详细的输出
- %T 打印值的类型
- %% 打印百分号本身

### 整数

- %b 二进制；%o 八进制；%O 八进制带0o前缀；%x 十六进制小写；%X 十六进制大写
- %U 把一个整数用Unicode格式打印。例如 `fmt.Printf("%U, %x, %c\n", 27979, 27979, 27979)` 输出 `U+6D4B, 6d4b, 测`
- %c 把rune、byte的整型值用字符形式打印
- %q 把一个整型当做Unicode字符输出，类似%c，不过在字符外面多了单引号。q的意思就是quote

### 浮点数

- %e、%E 科学计数法
- %f、%F 小数表示法，最常用
- %g 内部选择使用%e还是%f以简洁输出；%G 选择%E或%F

### 字符串或字节切片

- %s 字符串输出。如果是rune切片，需要string强制类型转换
- %q 类似%s，外部加上双引号。q的意思就是quote

### 指针

- %p 十六进制地址

类型	说明	缺省格式符	常用格式符
bool	布尔型	%t	%t
int/int8/int16/int32/int64/	整型	%d	%d、%b、%x
uint/unit8/uint16/uint32/uint64	无符号整型	%d，如果使用%#v就等同%#x	%d、%b、%x
float32/float64	浮点型	%g	%f、%e
complex64/complex128	复数	%g	
byte	字节型	%c	%c、%d
rune	字符型	%c	%c、%d
string/[]byte	字符串	%s	%s
uintptr	指针	%p	%p
map slice channel error	引用	%v	
slice	索引0元素地址	%p	

## 特殊格式符写法

```
1 a, b, c, d := 100, 200, 300, 400
2 fmt.Printf("%d, %[2]v, %[1]d, %d", a, b, c, d)
```

可以认为中括号内写的是索引，是 Printf 的索引，索引0是格式字符串本身，1开始才是参数。如果写了[n]，之后默认就是n+1。

## 输出函数

输出到标准输出

- Print：使用缺省格式输出，空格分割
- Println：使用缺省格式输出，空格分割，最后追加换行
- Printf：按照指定的格式符输出

输出到字符串，经常用来拼接字符串用

- Sprint：相当于Print，不过输出为string
- Sprintln：相当于Println，不过输出为string
- Sprintf：相当于Printf，不过输出为string

## 操作符

参考 [https://golang.google.cn/ref/spec#Operators\\_and\\_punctuation](https://golang.google.cn/ref/spec#Operators_and_punctuation)

### 逻辑运算真值表

与逻辑			或逻辑			非逻辑	
<i>A</i>	<i>B</i>	<i>F</i>	<i>A</i>	<i>B</i>	<i>F</i>	<i>A</i>	<i>F</i>
0	0	0	0	0	0	0	1
0	1	0	0	1	1		
1	0	0	1	0	1		
1	1	1	1	1	1	1	0

### 算数运算符

+, -, \*, /, %, ++, --

- 5 / 2、-5 / 2
- +、-还可以当做正负用，就不是算数运算符了，例如-s。
- 类C语言语法没有Python `//` 的除法符号，因为它是注释

++, --只能是i++, i--, 且是语句，不是表达式。也就是说，语句不能放到等式、函数参数等地方。例如，`fmt.Println(a++)` 是语法错误。

没有++i、--i。

### 常量计算问题

常量分为typed类型化常量和untyped常量。

注意下面的常见错误

```
1 var a int = 1
2 var b float32 = 2.3
3 fmt.Println(a * b) // 错误，int和float32类型不同，无法计算，除非强制类型转
4
5 var a = 1 // int
6 var b = 2.3 // float64
7 fmt.Println(a * b) // 错误，int和float64类型不同，无法计算，除非强制类型转换
8
9 fmt.Println(1 * 2.3) // 报错吗？
```

上面的常量被赋给了变量，这些变量就确定了类型，虽然他们指向的值是字面常量，但是计算使用变量，但变量的类型不一致，报错。

再看下面的例子

```
1 var a = 1 * 2.3 // 不报错
2 fmt.Printf("%T %v\n", s, s) // float64 2.3
```

等号右边使用的就是不同类型的值计算为什么可以？

因为右边使用的都是字面常量，而字面常量都是无类型常量untyped constant，它会在上下文中隐式转换。Go为了方便，不能过于死板，增加程序员转换类型的负担，在无类型常量上做了一些贴心操作。

```
1 An untyped constant has a default type which is the type to which the
  constant is implicitly converted in contexts where a typed value is required,
  for instance, in a short variable declaration such as i := 0 where there is
  no explicit type.
2
3 摘自 https://golang.google.cn/ref/spec#Constants
```

## 位运算符

&位与、|位或、^异或、&^位清空、<<、>>

```
1 fmt.Println(2&1, 2&^1, 3&1, 3&^1) // 0 2 1 2
2 fmt.Println(2|1, 3^3, 1<<3, 16>>3, 2^1) // 3 0 8 2 3
```

x&y，位与本质就是按照y有1的位把x对应位的值保留下来。

x&^y，位清空本质就是先把y按位取反后的值，再和x位与，也就是y有1的位的值不能保留，被清空，原来是0的位被保留。换句话说，就是**按照y有1的位清空x对应位**。

思考一下 15 & 5 和 15 &^ 5 分别是什么？

## 比较运算符

比较运算符组成的表达式，返回bool类型值。成立返回True，不成立返回False

==、!=、>、<、>=、<=

## 逻辑运算符

&&、||、!

由于Go语言对类型的要求，逻辑运算符操作的只能是bool类型数据，那么结果也只能是bool型。

```
1 // 短路
2 fmt.Println(false && true, true && true && false)
3 fmt.Println(false || true, true || false || true)
```



## 赋值运算符

`=`、`+=`、`-=`、`*=`、`/=`、`%=`、`>>=`、`<<=`、`&=`、`&^=`、`^=`、`|=`

`:=` 短格式赋值。

## 三元运算符

Go 中没有三元运算符。

没有 `?:` 的原因是，语言的设计者看到这个操作经常被用来创建难以理解的复杂表达式。在替代方案上，`if-else` 形式虽然较长，但无疑是更清晰的。一门语言只需要一个条件控制流结构。

## 指针操作

数据是放在内存中，内存是线性编址的。任何数据在内存中都可以通过一个地址来找到它。

`&` 取地址

`*` 指针变量，表示通过指针取值

```
1  a := 123
2  b := &a // &取地址
3  c := *b
4  fmt.Printf("%d, %p, %d\n", a, b, c)
5  // 请问，下面相等吗？
6  fmt.Println(a == c, b == &c, &c)
7
8  var d = a
9  fmt.Println(a == d, &a, &d) // &a == &d吗？
```

## 优先级

Category	Operator	Associativity
Postfix后缀	() [] -> . ++ --	Left to right
Unary单目	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative乘除	* / %	Left to right
Additive加减	+ -	Left to right
Shift移位	<< >>	Left to right
Relational关系	< <= > >=	Left to right
Equality相等	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Assignment赋值	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma逗号运算符	,	Left to right

规则：

- 表中优先级由高到低
- 单目 > 双目
- 算数 > 移位 > 比较 > 逻辑 > 赋值
- 搞不清，用括号，避免产生歧义