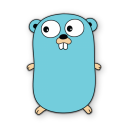
**Go 语言教程**



Go 是一个开源的编程语言，它能让构造简单、可靠且高效的软件变得容易。

Go是从2007年末由Robert Griesemer, Rob Pike, Ken Thompson主持开发，后来还加入了Ian Lance Taylor, Russ Cox等人，并最终于2009年11月开源，在2012年早些时候发布了Go 1稳定版本。现在Go的开发已经是完全开放的，并且拥有一个活跃的社区。

**Go 语言特色**

* 简洁、快速、安全
* 并行、有趣、开源
* 内存管理、v数组安全、编译迅速

**Go 语言用途**

Go 语言被设计成一门应用于搭载 Web 服务器，存储集群或类似用途的巨型中央服务器的系统编程语言。

对于高性能分布式系统领域而言，Go 语言无疑比大多数其它语言有着更高的开发效率。它提供了海量并行的支持，这对于游戏服务端的开发而言是再好不过了。

**第一个 Go 程序**

接下来我们来编写第一个 Go 程序 hello.go（Go 语言源文件的扩展是 .go），代码如下：

**实例**

package main  
import "fmt"  
func main() {  
    fmt.Println("Hello, World!")  
}

执行以上代码输出

$ go run hello.go

Hello, World!

# Go 语言环境安装

Go 语言支持以下系统：

* Linux
* FreeBSD
* Mac OS X（也称为 Darwin）
* Window

安装包下载地址为：<https://golang.org/dl/>。

各个系统对应的包名：

|  |  |
| --- | --- |
| **操作系统** | **包名** |
| Windows | go1.4.windows-amd64.msi |
| Linux | go1.4.linux-amd64.tar.gz |
| Mac | go1.4.darwin-amd64-osx10.8.pkg |
| FreeBSD | go1.4.freebsd-amd64.tar.gz |

## UNIX/Linux/Mac OS X, 和 FreeBSD 安装

以下介绍了在UNIX/Linux/Mac OS X, 和 FreeBSD系统下使用源码安装方法：

1、下载源码包：go1.4.linux-amd64.tar.gz。

2、将下载的源码包解压至 /usr/local目录。

tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz

3、将 /usr/local/go/bin 目录添加至PATH环境变量：

export PATH=$PATH:/usr/local/go/bin

注意：MAC 系统下你可以使用 .pkg 结尾的安装包直接双击来完成安装，安装目录在 /usr/local/go/ 下。

## Windows 系统下安装

Windows 下可以使用 .msi 后缀(在下载列表中可以找到该文件，如go1.4.2.windows-amd64.msi)的安装包来安装。

默认情况下.msi文件会安装在 c:\Go 目录下。你可以将 c:\Go\bin 目录添加到 PATH 环境变量中。添加后你需要重启命令窗口才能生效。

### 安装测试

创建工作目录 C:\>Go\_WorkSpace。

文件名: test.go，代码如下：

package main

import "fmt"

func main() {

fmt.Println("Hello, World!")

}

使用 go 命令执行以上代码输出结果如下：

C:\Go\_WorkSpace>go run test.go

Hello, World!

**Go 语言结构**

在我们开始学习 Go 编程语言的基础构建模块前，让我们先来了解 Go 语言最简单程序的结构。

**Go Hello World 实例**

Go 语言的基础组成有以下几个部分：

* 包声明
* 引入包
* 函数
* 变量
* 语句 & 表达式
* 注释

接下来让我们来看下简单的代码，该代码输出了"Hello World!":

package main

import "fmt"

func main() {

/\* 这是我的第一个简单的程序 \*/

fmt.Println("Hello, World!")

}

让我们来看下以上程序的各个部分：

1. 第一行代码 *package main* 定义了包名。你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。
2. 下一行 *import "fmt"* 告诉 Go 编译器这个程序需要使用 fmt 包（的函数，或其他元素），fmt 包实现了格式化 IO（输入/输出）的函数。
3. 下一行 *func main()* 是程序开始执行的函数。main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（如果有 init() 函数则会先执行该函数）。
4. 下一行 /\*...\*/ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。
5. 下一行 *fmt.Println(...)* 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。   
   使用 fmt.Print("hello, world\n") 可以得到相同的结果。   
   Print 和 Println 这两个函数也支持使用变量，如：fmt.Println(arr)。如果没有特别指定，它们会以默认的打印格式将变量 arr 输出到控制台。
6. 当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 protected ）。

**执行 Go 程序**

让我们来看下如何编写 Go 代码并执行它。步骤如下：

1. 打开编辑器如Sublime2，将以上代码添加到编辑器中。
2. 将以上代码保存为 *hello.go*
3. 打开命令行，并进入程序文件保存的目录中。
4. 输入命令 *go run hello.go* 并按回车执行代码。
5. 如果操作正确你将在屏幕上看到 *"Hello World!"* 字样的输出。

$ go run hello.go

Hello, World!

**Go 语言基础语法**

上一章节我们已经了解了 Go 语言的基本组成结构，本章节我们将学习 Go 语言的基础语法。

**Go 标记**

Go 程序可以由多个标记组成，可以是关键字，标识符，常量，字符串，符号。如以下 GO 语句由 6 个标记组成：

fmt.Println("Hello, World!")

6 个标记是(每行一个)：

1. fmt

2. .

3. Println

4. (

5. "Hello, World!"

6. )

**行分隔符**

在 Go 程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号 ; 结尾，因为这些工作都将由 Go 编译器自动完成。

如果你打算将多个语句写在同一行，它们则必须使用 ; 人为区分，但在实际开发中我们并不鼓励这种做法。

以下为两个语句：

fmt.Println("Hello, World!")

fmt.Println("w3cschool菜鸟教程：w3cschool.cc")

**注释**

注释不会被编译，每一个包应该有相关注释。

单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾。如：

// 单行注释

/\*

Author by w3cschool菜鸟教程

我是多行注释

\*/

**标识符**

标识符用来命名变量、类型等程序实体。一个标识符实际上就是一个或是多个字母(A~Z和a~z)数字(0~9)、下划线\_组成的序列，但是第一个字符必须是字母或下划线而不能是数字。

以下是有效的标识符：

mahesh kumar abc move\_name a\_123

myname50 \_temp j a23b9 retVal

以下是无效的标识符：

* 1ab（以数字开头）
* case（Go 语言的关键字）
* a+b（运算符是不允许的）

**关键字**

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| append | bool | byte | cap | close | complex | complex64 | complex128 | uint16 |
| copy | false | float32 | float64 | imag | int | int8 | int16 | uint32 |
| int32 | int64 | iota | len | make | new | nil | panic | uint64 |
| print | println | real | recover | string | true | uint | uint8 | uintptr |

程序一般由关键字、常量、变量、运算符、类型和函数组成。

程序中可能会使用到这些分隔符：括号 ()，中括号 [] 和大括号 {}。

程序中可能会使用到这些标点符号：.、,、;、: 和 …。

**Go 语言的空格**

Go 语言中变量的声明必须使用空格隔开，如：

var age int;

语句中适当使用空格能让程序看易阅读。

无空格：

fruit=apples+oranges;

在变量与运算符间加入空格，程序看起来更加美观，如：

fruit = apples + oranges;

**Go 语言数据类型**

在 Go 编程语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

Go 语言按类别有以下几种数据类型：

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **布尔型** 布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。 |
| 2 | **数字类型** 整型 int 和浮点型 float，Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。 |
| 3 | **字符串类型:** 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。Go语言的字符串的字节使用UTF-8编码标识Unicode文本。 |
| 4 | **派生类型:** 包括：   * (a) 指针类型（Pointer） * (b) 数组类型 * (c) 结构化类型(struct) * (d) Channel 类型 * (e) 函数类型 * (f) 切片类型 * (g) 接口类型（interface） * (h) Map 类型 |

**数字类型**

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

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

浮点型：

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **float32** IEEE-754 32位浮点型数 |
| 2 | **float64** IEEE-754 64位浮点型数 |
| 3 | **complex64** 32 位实数和虚数 |
| 4 | **complex128** 64 位实数和虚数 |

**其他数字类型**

以下列出了其他更多的数字类型：

|  |  |
| --- | --- |
| **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 |

# Go 语言变量

变量来源于数学，是计算机语言中能储存计算结果或能表示值抽象概念。变量可以通过变量名访问。

Go 语言变量名由字母、数字、下划线组成，其中首个字母不能为数字。

声明变量的一般形式是使用 var 关键字：

var identifier type

### 变量声明

第一种，指定变量类型，声明后若不赋值，使用默认值。

var v\_name v\_type

v\_name = value

第二种，根据值自行判定变量类型。

var v\_name = value

第三种，省略var, 注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误。

v\_name := value

// 例如

var a int = 10

var b = 10

c : = 10

实例如下：

package main

var a = "菜鸟教程"

var b string = "runoob.com"

var c bool

func main(){

println(a, b, c)

}

以上实例执行结果为：

菜鸟教程 runoob.com false

### 多变量声明

//类型相同多个变量, 非全局变量

var vname1, vname2, vname3 type

vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 //和python很像,不需要显示声明类型，自动推断

vname1, vname2, vname3 := v1, v2, v3 //出现在:=左侧的变量不应该是已经被声明过的，否则会导致编译错误

// 这种因式分解关键字的写法一般用于声明全局变量

var (

vname1 v\_type1

vname2 v\_type2

)

实例如下：

package main

var x, y int

var ( // 这种因式分解关键字的写法一般用于声明全局变量

a int

b bool

)

var c, d int = 1, 2

var e, f = 123, "hello"

//这种不带声明格式的只能在函数体中出现

//g, h := 123, "hello"

func main(){

g, h := 123, "hello"

println(x, y, a, b, c, d, e, f, g, h)

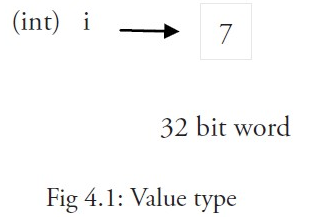
}

以上实例执行结果为：

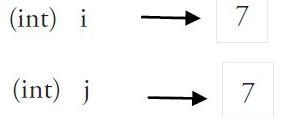
0 0 0 false 1 2 123 hello 123 hello

## 值类型和引用类型

所有像 int、float、bool 和 string 这些基本类型都属于值类型，使用这些类型的变量直接指向存在内存中的值：



当使用等号 = 将一个变量的值赋值给另一个变量时，如：j = i，实际上是在内存中将 i 的值进行了拷贝：

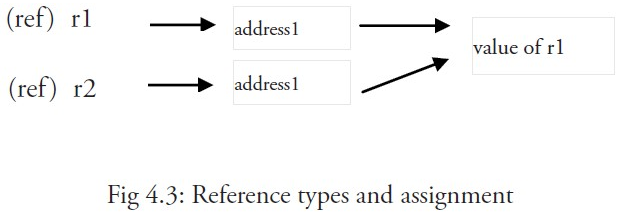


你可以通过 &i 来获取变量 i 的内存地址，例如：0xf840000040（每次的地址都可能不一样）。值类型的变量的值存储在栈中。

内存地址会根据机器的不同而有所不同，甚至相同的程序在不同的机器上执行后也会有不同的内存地址。因为每台机器可能有不同的存储器布局，并且位置分配也可能不同。

更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。

一个引用类型的变量 r1 存储的是 r1 的值所在的内存地址（数字），或内存地址中第一个字所在的位置。



这个内存地址为称之为指针，这个指针实际上也被存在另外的某一个字中。

同一个引用类型的指针指向的多个字可以是在连续的内存地址中（内存布局是连续的），这也是计算效率最高的一种存储形式；也可以将这些字分散存放在内存中，每个字都指示了下一个字所在的内存地址。

当使用赋值语句 r2 = r1 时，只有引用（地址）被复制。

如果 r1 的值被改变了，那么这个值的所有引用都会指向被修改后的内容，在这个例子中，r2 也会受到影响。

## 简短形式，使用 := 赋值操作符

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，声明语句写上 var 关键字其实是显得有些多余了，因此我们可以将它们简写为 a := 50 或 b := false。

a 和 b 的类型（int 和 bool）将由编译器自动推断。

这是使用变量的首选形式，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。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

### 注意事项

如果在相同的代码块中，我们不可以再次对于相同名称的变量使用初始化声明，例如：a := 20 就是不被允许的，编译器会提示错误 no new variables on left side of :=，但是 a = 20 是可以的，因为这是给相同的变量赋予一个新的值。

如果你在定义变量 a 之前使用它，则会得到编译错误 undefined: a。

如果你声明了一个局部变量却没有在相同的代码块中使用它，同样会得到编译错误，例如下面这个例子当中的变量 a：

package main

import "fmt"

func main() {

var a string = "abc"

fmt.Println("hello, world")

}

尝试编译这段代码将得到错误 **a declared and not used**。

此外，单纯地给 a 赋值也是不够的，这个值必须被使用，所以使用

fmt.Println("hello, world", a)

会移除错误。

但是全局变量是允许声明但不使用。 同一类型的多个变量可以声明在同一行，如：

var a, b, c int

多变量可以在同一行进行赋值，如：

a, b, c = 5, 7, "abc"

上面这行假设了变量 a，b 和 c 都已经被声明，否则的话应该这样使用：

a, b, c := 5, 7, "abc"

右边的这些值以相同的顺序赋值给左边的变量，所以 a 的值是 5， b 的值是 7，c 的值是 "abc"。

这被称为 并行 或 同时 赋值。

如果你想要交换两个变量的值，则可以简单地使用 a, b = b, a。

空白标识符 \_ 也被用于抛弃值，如值 5 在：\_, b = 5, 7 中被抛弃。

\_ 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 val 和错误 err 是通过调用 Func1 函数同时得到：val, err = Func1(var1)。

# Go 语言常量

常量是一个简单值的标识符，在程序运行时，不会被修改的量。

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。

常量的定义格式：

const identifier [type] = value

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

* 显式类型定义： const b string = "abc"
* 隐式类型定义： const b = "abc"

多个相同类型的声明可以简写为：

const c\_name1, c\_name2 = value1, value2

以下实例演示了常量的应用：

package main

import "fmt"

func main() {

const LENGTH int = 10

const WIDTH int = 5

var area int

const a, b, c = 1, false, "str" //多重赋值

area = LENGTH \* WIDTH

fmt.Printf("面积为 : %d", area)

println()

println(a, b, c)

}

以上实例运行结果为：

面积为 : 50

1 false str

常量还可以用作枚举：

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。

常量可以用len(), cap(), unsafe.Sizeof()函数计算表达式的值。常量表达式中，函数必须是内置函数，否则编译不过：

package main

import "unsafe"

const (

a = "abc"

b = len(a)

c = unsafe.Sizeof(a)

)

func main(){

println(a, b, c)

}

以上实例运行结果为：

abc 3 16

## iota

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

在每一个const关键字出现时，被重置为0，然后再下一个const出现之前，每出现一次iota，其所代表的数字会自动增加1。

iota 可以被用作枚举值：

const (

a = iota

b = iota

c = iota

)

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 可以简写为如下形式：

const (

a = iota

b

c

)

### iota 用法

package main

import "fmt"

func main() {

const (

a = iota //0

b //1

c //2

d = "ha" //独立值，iota += 1

e //"ha" iota += 1

f = 100 //iota +=1

g //100 iota +=1

h = iota //7,恢复计数

i //8

)

fmt.Println(a,b,c,d,e,f,g,h,i)

}

以上实例运行结果为：

0 1 2 ha ha 100 100 7 8

再看个有趣的的 iota 实例：

package main

import "fmt"

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)

}

以上实例运行结果为：

i= 1

j= 6

k= 12

l= 24

iota表示从0开始自动加1，所以i=1<<0,j=3<<1（<<表示左移的意思），即：i=1,j=6，这没问题，关键在k和l，从输出结果看，k=3<<2，l=3<<3。

**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

接下来让我们来详细看看各个运算符的介绍。

**算术运算符**

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。

|  |  |  |
| --- | --- | --- |
| **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 - c 的值为 %d\n", a )

a--

fmt.Printf("第七行 - c 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 - c 的值为 22

第七行 - c 的值为 21

**关系运算符**

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**

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**

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** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

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 |

以下实例演示了逻辑运算符的用法：

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**

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