链滴

Golang 入门笔记 -03-Go 语言基本数据类型

作者：zyk

原文链接：https://ld246.com/article/1599961004754

来源网站：链滴

1. 常量

1.1 常量的定义

常量是一种特殊的变量，被初始化之后就无法再改变。

Go 语言中，常量的类型只能是**布尔型，数字型（整型、浮点型和复数）和符串型。**

常量可以使用关键字 const 来定义，定义格式为 const variable [type] = value。

```
const m string = "abc" // 显示声明
const n = "xyz" // 隐式声明
```

常量的值必须在编译时能确定，给常量赋值时可以涉及表达式的计算，但计算值必须能在编译时确定。

```
const m = 2/3 // 正确
const n = getValue() // 错误，编译时自定义函数属于未知，无法用于常量赋值，但可以使用内置函
，如 len()
```

1.2 枚举

常量也可以用于枚举：

Unknown 表示未知性别，Female 表示女性，Male 表示男性。

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

itoa 是一个特殊的常量，itoa 在 const 关键字出现时被重置为 0；const 中每多声明一行常量，itos 会自动加 1（itoa 可理解成常量的**行索引**）。如：

```
const (
    a = itoa // a 的值为 0
    b = itoa // b 的值为 1
    c = itoa // c 的值为 2
)
```

当声明多行常量时，若不指定常量的值和类型，那么该常量的类型和值与**上一个常量相同。**

```
const (
    a = 100
    b // b 的值为 100
)

const (
    a = itoa // a 的值为 0
    b // b 的值为 1
    c // c 的值为 2
```

```
)
```

我们来看一个例子：

```
package main

import "fmt"

func main() {
    const (
        a = iota // 0
        b        // 1
        c        // 2
        d = "ha" // iota += 1
        e        // "ha" iota += 1
        f = 100  // iota +=1
        g        // 100  iota += 1
        h = iota // 7
        i        // 8
    )

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

运行结果为：

```
0 1 2 ha ha 100 100 7 8
```

2. 变量

2.1 定义变量

我们在上一篇文章 Golang 入门笔记-02-Go 语言基本语法和结构 中已阐述了变量的定义方式和注意。

2.2 值类型和引用类型

基本数据类型 int，float，bool，string 以及**数组**和**结构体**都属于**值类型**，**值类型**的变量**直接存储值**内存通常在**栈**中分配。

指针，slice，map 和 chan 等都属于**引用类型**，**引用类型**的变量存储的是**地址**，内存通常在**堆**中分配比栈拥有更大的空间，通过 GC 进行回收。

Go 语言中可以通过 & 来获取变量的内存地址，如获取变量 i 的内存地址：&i。

若一个变量被引用，那么当该变量发生变化时，该变量的引用都会指向被修改后的内容。

我们来看一个例子：

```
package main

import "fmt"
```

```
func main() {
    a := 1
    c := &a // c 的类型为 *int，是变量 a 的引用
    d := &a // d 的类型为 *int，是变量 a 的引用
    fmt.Println("c = ", c, ", d = ", d)
    fmt.Println("*c = ", *c, ", *d = ", *d)

    a = 2 // 修改 a 的值，引用类型变量 c 与 d 会指向修改后的 a
    fmt.Println("c = ", c, ", d = ", d)
    fmt.Println("*c = ", *c, ", *d = ", *d)
}
```

运行结果为：

```
c =  0xc00000a0a0 , d =  0xc00000a0a0
*c =  1 , *d =  1
c =  0xc00000a0a0 , d =  0xc00000a0a0
*c =  2 , *d =  2
```

3. 基本类型和运算符

3.1 基本类型

3.1.1 布尔类型

定义 bool 类型的变量：

```
var b1 bool = true
var b2 = true
b3 := false
```

bool 类型的值只能是 true 或 false。

可以通过运算符等于 == 或不等于 != 得到 bool 类型的值，如：

```
var m = 6

m == 5 // false
m == 6 // true
m != 5 // true
m != 6 // false
```

还可以通过与**逻辑运算符**非 !，与 && ，或 || 结合得到 bool 类型的值。

Go 语言中，&& 和 || 具有**快捷性质**，当 && 和 || 左边的表达式已经能够决定整个表达式结果时（当 & 左边值为 false 或 || 左边值为 true 时），右边的表达式不会被执行。因此，我们应尽量将复杂的达式放在右边，以减少运算量。

```
var T = true
var F = false

!T // false
!F // true
```

```
T && T // true
T && F // false
F && T // false
F && F // false
T || T // true
T || F // true
F || T // true
F || F // false
```

3.1.2 字符串

Go 语言中字符串的字节使用 UTF-8 编码表示 Unicode 文本，所以 Go 语言字符串是**变宽字符序列**这和其他语言（Java，Python）完全不同，后者为**定宽字符序列。**Go 语言这样做，就不需要对 UTF-字符集文本进行编码和解码，节省了内存和空间。

Go 语言支持以下两种形式定义字符串：解释字符串

- 解释字符串用 **双引号**包裹起来，如：

```
package main

import "fmt"

func main() {
    var s = "abc\nefg"

    fmt.Println("s: ", s)
}
```

运行结果为：

```
s:  abc
efg
```

同时，以下这些字符串将被转义：

- \n：换行符
- \r：回车符
- \t：tab 键
- \u 或\U：Unicode 字符
- \\：反斜杠

- 非解释字符串

非解释字符串用**反引号**包裹起来，与解释字符串不同，非解释字符串会按照用户的输入形式保存起来且 \n，\r 等字符串不会被转义， 如：

```
package main

import "fmt"

func main() {
  var s = `
        Hello World!\n
```

```
        I am a gopher!
    `
    fmt.Println("s: ", s)
}
```

运行结果为：

```
s:
        Hello World!\n
            I am a gopher!
```

Go 语言中的字符串和 C/C++ 不一样，不以 \0 表示结尾，而是以长度限定。

长度为 0 的字符串是空字符串 ""。

可以通过 len() 来获取字符串的长度，如：

```
s := "abc"
length := len(s) // 3
```

可以通过数组下标获取字符串中的字符，如：

```
s := "abc"
s1 := s[0] // 'a'
s2 := s[len(s) - 1] // 'c'
```

可以通过 + 进行字符串拼接：

```
s1 := "Hello "
s2 := "World"
s3 := s1 + s2

s4 := "He" + "llo "
s4 += "World" // 等同于 s4 = s4 + "World"

s5 := "Hello " +  // 多行拼接 +  必须放在上一行
"World"
```

用 + 来拼接字符串效率并不是太高，后续我们会讲到使用字节缓冲 bytes.Buffer 来进行字符串拼接。

3.1.3 整型

Go 语言中比较常见的整型有 int，uint，uintptr，这三个类型的长度和计算机架构有关。

- int 和uint 在 32 位操作系统上，长度为 32 位（4 个字节）；64 位操作系统上，长度为 64 位（8 字节）。
- uintptr 的长度可存放一个指针。

还有与计算机架构无关的整型，从命名上就可以看出：

- 整型
    - int8（-128 ~ 127）
    - int16（-32768 ~ 32767）

- int32（-2,147,483,648 ~ 2,147,483,647）
  - int64（-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807）
- 无符号整型
  - uint8（0 ~ 255）
  - uint16（0 ~ 65,535）
  - uint32（0 ~ 4,294,967,295）
  - uint64（0 ~ 18,446,744,073,709,551,615）

可以通过前缀 0 来表示**八进制数**：0666；通过前缀 0x 表示**十六进制数**：0xEF；通过 e 来表示 10 的次方：1e2：100，3.14e20：3.14 x (10^20)。

注意：

- int 和 int64 **不是同类型**，以下代码是**错误**的：

```
var a int = 1
var b int64 = 2
a = b // a 与 b 属于不同类型变量，无法赋值
```

- 声明整型变量时，若不指定类型，默认为 int：

```
var a = 1 // int
b := 1 // int
```

- 短变量声明中可以通过 **指定类型**来定义变量：

```
b := uint64(2) // uint64
```

3.1.4 浮点型

Go 语言中没有 float 类型，仅存在两种浮点类型（遵循 IEEE-754 标准）：

- float32（+- 1e-45 ~ +- 3.4 * 1e38）
- float64（+- 5 * 1e-324 ~ 107 * 1e308）

浮点型默认值为 0.0。

float32 精确到小数点后 7 位，float64 精确到小数点后 15 位。

对于精确度要求较高的运算，应使用 float64 类型。

3.1.5 复数

Go 语言中，复数有两种类型：

- complex64（32 位实数和虚数）
- complex128（64 位实数和虚数）

我们可以通过 a + bi 的方式来定义复数，a 表示**实部**，b 表示**虚部**，如：

```
var c complex64 = 1 + 2i
```

可以通过 real(c) 和 imag(c) 函数来获取复数变量 c 的**实部**和**虚部**。

复数和其他数据类型一样，支持 == 和 != 的比较，比较时需注意复数**精确度**。若对性能没有太高要，建议使用精确度更高的 complex128 。

3.2 运算符

3.2.1 位运算

注意：位运算是对给定数所对应的**二进制数**进行运算。

- 二元运算符
  - 按位与 &

当两位都为 1 时，值才为 1，否则值为 0：

```
1 & 1 // 1
1 & 0 // 0
0 & 1 // 0
0 & 0 // 0
```

  - 按位或 |

两位中有一个为 1，值即为 1,否则，值为 0：

```
1 | 1 // 1
1 | 0 // 1
0 | 1 // 1
0 | 0 // 0
```

  - 按位异或 ^

若两位的值相同，值为 0，若不相同，值为 1：

```
1 ^ 1 // 0
1 ^ 0 // 1
0 ^ 1 // 1
0 ^ 0 // 0
```

  - 按位置零 &^

p &^ q：当 q 为 0 时，则结果为 p；否则，结果为 0：

```
1 &^ 0 // 1
1 &^ 1 // 0
```

- 一元运算符
  - 位左移 <<

用法：a << n，将 a 左移 n 位，右侧部分用 0 填充。相当于 a 乘以 2 的 n 次方。

```
1 << 10 // 2^10 -> 1KB
1 << 20 // 2^20 -> 1MB
1 << 30 // 2^30 -> 1GB
```

- 位右移 >>

用法：a >> n，将 a 右移 n 位，左侧部分用 0 填充。相当于 a 除以 2 的 n 次方。

```
8 >> 1 // 8/(2^1) -> 4
1024 >> 6 // 1024/(2^6) -> 16
```

3.2.2 逻辑运算符

- 逻辑非 !

! 运算符表示取反，当 T 为 true 时，!T 值为 false，反之则为 true。

- 逻辑与 &&

&& 运算符表示逻辑与，当 && 两边表达式都为 true 时，值才为 true，否则为 false。

- 逻辑或 ||

|| 运算符表示逻辑或，当 || 有一边表达式为 true 时，值就为 true，仅当两边表达式都为 false，值才 false。

3.2.3 算术运算符

- + 加法

```
a := 1
b := 2
c := a + b // 3
```

- - 减法

```
a := 2
b := 1
c := a - b // 1
```

- * 乘法

```
a := 1
b := 2
c := a * b // 2
```

- / 除法

```
a := 2
b := 1
c := a / b // 2
```

- % 求余

```
a := 6
b := 5
c := a % b // 1
```

- ++ 自增

变量自增 1。

```
a := 1
```

```
a++ // 2，相当于 a = a + 1
```

- -- 自减

变量自减 1。

```
a := 2
a-- // 1，相当于 a = a - 1
```

注意：

- Go 语言中， ++ 和 -- 仅仅是**语句**，不能作为**表达式**，以下写法是**错误**的：

```
a := 1
b := a++ // 错误，++ 不能作为表达式
```

- 运算符的优先级

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

4. 指针

Go 语言为我们提供了指针功能，但不能进行指针运算。Go 语言允许我们控制特定数据结构，分配数和内存访问，有利于构建强大的网络应用。

指针在 Go 语言中被拆分为两个概念：

- 类型指针：允许对这个指针类型的数据进行修改，传递参数可以使用指针，无需拷贝数据。
- 切片：由指向起始元素的原始指针、元素容量和容量构成。

4.1 指针地址和指针类型

一个**指针变量**可以指向**任意**一个值的**地址**，它所指向的值的内存地址在 32 位和 64 位计算机上分别用 4 个字节和 8 个字节。（占用字节大小与值大小无关）

当一个指针被定义后，若没有被分配变量，则它的默认值为 nil（相当于于 C 语言中的 null）。

每一个变量都有一个地址，Go 语言中可以通过取址符号 & 来获取一个变量的内存地址：

```
a := 1
p := &a // &a 表示取 a 的地址
```

我们通过一个例子来了解地址：

```
package main

import "fmt"
```

```
func main() {
    a := 1 // 定义 int 型变量 a
    s := "abc" // 定义 string 型变量 s

    fmt.Printf("%p %p", &a, &s) // 打印 a 和 s 的内存地址
}
```

运行结果为：

```
0xc00000a0a0 0xc00003c1f0
```

任意变量都有地址，指针变量保存的就是地址。

4.2 获取指针指向的值

可以对指针使用 * 操作符来获取指针指向的值，例如我们定义了指针变量 p，可以通过 *p 来获取指针 p 指向的值：

```
package main

import "fmt"

func main() {
    a := 1  // 定义 int 型变量 a
    p := &a // 定义指针变量 p，p 保存 a 的地址

    fmt.Printf("p -> type %T\n", p)    // p 的类型
    fmt.Printf("p -> value %p\n", p)   // p 的值
    fmt.Printf("*p -> type %T\n", *p)   // *p 的类型
    fmt.Printf("*p ->    value %v\n", *p) // *p 的值
}
```

运行结果为：

```
p -> type *int
p -> value 0xc00000a0a0
*p - > type int
*p -> value 1
```

4.3 通过指针修改值

指针也可以修改值：

```
package main

import "fmt"

func swap(a, b *int) {
    t := *a // t 保存 a 指向的变量的值

    *a = *b // 将 b 指向的变量的值赋值给 a 指向的变量
```

```
    *b = t // 将 t 赋值给 b 指向的变量
}

func main() {
    m, n := 1, 2

    swap(&m, &n)

    fmt.Println(m, n)
}
```

运行结果为：

```
2 1
```

* 指针操作符作为右值时，如 t := *a ，表示取指针指向的变量的值；作为左值时，如 *b = t，表示针指向的变量。

我们还可以通过 new 函数来创建指针变量，如：

```
s := new(string)
*s = "Hello World"
```

new() 函数可以创建一个对应类型的指针，创建时会分配内存，创建完成后指针指向默认值。

注意：

● 取址符号 & 和指针操作符* 是一对**互逆操作**，通过取址符号& 来获取变量的地址，通过指针操作符* 来获取指针指向的变量的值。