

基础类型

变量的命名

Go语言中的函数名、变量名、常量名、类型名、语句标号和包名等所有的命名，都遵循一个简单的命名规则：一个名字必须以一个字母（Unicode字母）或下划线开头，后面可以跟任意数量的字母、数字或下划线。大写字母和小写字母是不同的：**heapSort**和**Heapsort**是两个不同的名字。

25个关键字

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			

30多个预定义的名字，比如int和true等，主要对应内建的常量、类型和函数。

内建常量：

```
true false iota nil
```

内建类型：

```
int int8 int16 int32 int64  
uint uint8 uint16 uint32 uint64 uintptr  
float32 float64 complex128 complex64  
bool byte rune string error
```

内建函数：

```
make len cap new append copy close delete  
complex real imag  
panic recover
```

变量

变量的声明

Go语言引入了关键字var

```
var v1 int
```

```
var v2 int
```

```
//一次定义多个变量
```

```
var v3, v4 int
```

```
var (
```

```

    v5 int

    v6 int

)

```

变量初始化

对于声明变量时需要进行初始化的场景，`var`关键字可以保留，但不再是必要的元素，如下：

```

var v1 int = 10 // 方式1
var v2 = 10     // 方式2，编译器自动推导出v2的类
型
v3 := 10        // 方式3，编译器自动推导出v3的
类型
fmt.Println("v3 type is ", reflect.TypeOf(v3))
//v3 type is  int

//出现在 := 左侧的变量不应该是已经被声明过，:=定义时
必须初始化
var v4 int
v4 := 2 //err

```

变量赋值

```

var v1 int
v1 = 123

```

```
var v2, v3, v4 int
v2, v3, v4 = 1, 2, 3    //多重赋值

i := 10
j := 20
i, j = j, i    //多重赋值
```

匿名变量

`_`（下划线）是个特殊的变量名，任何赋予它的值都会被丢弃：

```
_, i, _, j := 1, 2, 3, 4

func test() (int, string) {
    return 250, "sb"
}

_, str := test()
```

常量

Go语言中，常量是指编译期间就已知且不可改变的值。常量可以是数值类型（包括整型、浮点型和复数类型）、布尔类型、字符串类型等。

字面常量(常量值)

所谓字面常量 (literal)，是指程序中硬编码的常量，如：

```
123
```

```
3.1415 // 浮点类型的常量
```

```
3.2+12i // 复数类型的常量
```

```
true // 布尔类型的常量
```

```
"foo" // 字符串常量
```

常量定义

```
const Pi float64 = 3.14
```

```
const zero = 0.0 // 浮点常量，自动推导类型
```

```
const ( // 常量组
```

```
    size int64 = 1024
```

```
    eof      = -1 // 整型常量，自动推导类型
```

```
)
```

```
const u, v float32 = 0, 3 // u = 0.0, v = 3.0,
```

常量的多重赋值

```
const a, b, c = 3, 4, "foo"
```

```
// a = 3, b = 4, c = "foo" //err, 常量不能修
```

改

iota枚举

常量声明可以使用iota常量生成器初始化，它用于生成一组以相似规则初始化的常量，但是不用每行都写一遍初始化表达式。

在一个const声明语句中，在第一个声明的常量所在的行，iota将会被置为0，然后在每一个有常量声明的行加一。

```
const (
    x = iota // x == 0
    y = iota // y == 1
    z = iota // z == 2
    w // 这里隐式地说w = iota，因此w == 3。其实上面y和z可同样不用"= iota"
)
```

`const v = iota` // 每遇到一个const关键字，iota就会重置，此时v == 0

```
const (
    h, i, j = iota, iota, iota //h=0,i=0,j=0
iota在同一行值相同
)
```

```
const (
    a          = iota //a=0
    b          = "B"
    c          = iota          //c=2
    d, e, f    = iota, iota, iota //d=3,e=3,f=3
)
```

```

    g      = iota      //g = 4
)

const (
    x1 = iota * 10 // x1 == 0
    y1 = iota * 10 // y1 == 10
    z1 = iota * 10 // z1 == 20
)

```

基础数据类型

分类

Go语言内置以下这些基础类型：

类型	名称	长度 (byte)	零值	说明
bool	布尔类型	1	false	其值不为真即为假，不可以用数字代表true或false
byte	字节型	1	0	uint8别名
rune	字符类型	4	0	专用于存储unicode编码，等价于uint32
int, uint	整型	4或8	0	32位或64位
int8, uint8	整型	1	0	-128 ~ 127, 0 ~ 255

类型	名称	长度 (byte)	零值	说明
int16, uint16	整型	2	0	-32768 ~ 32767, 0 ~ 65535
int32, uint32	整型	4	0	-21亿 ~ 21 亿, 0 ~ 42 亿
int64, uint64	整型	8	0	
float32	浮点 型	4	0.0	小数位精确到7位
float64	浮点 型	8	0.0	小数位精确到15位
		8		
complex128	复数 类型	16		
uintptr	整型	4或8		足以存储指针的 uint32或uint64整 数
string	字符 串		""	utf-8字符串

布尔类型

```
var v1 bool
v1 = true
v2 := (1 == 2) // v2也会被推导为bool类型
```

//布尔类型不能接受其他类型的赋值，不支持自动或强制的类型转换

```
var b bool  
b = 1 // err, 编译错误  
b = bool(1) // err, 编译错误
```

整形

```
var v1 int32  
v1 = 123  
v2 := 64 // v1将会被自动推导为int类型
```

浮点型

```
var f1 float32  
f1 = 12  
f2 := 12.0 // 如果不加小数点， fvalue2会被推导为整  
型而不是浮点型， float64
```

字符类型

Go语言中支持两个字符类型，一个是byte（实际上是uint8的别名），代表utf-8字符串的单个字节的值；另一个是rune，代表单个unicode字符。

```
package main  
  
import (  
    "fmt"
```

```

)

func main(){
    var ch1, ch2, ch3 byte
    ch1 = 'a'    //字符赋值
    ch2 = 97     //字符的ascii码赋值
    ch3 = '\n'   //转义字符
    fmt.Printf("ch1 = %c, ch2 = %c, %c", ch1, ch2,
ch3)
}

```

字符串

Go语言中，字符串也是一种基本类型：

```

var str string
// 声明一个字符串变量

str = "abc"
// 字符串赋值

ch := str[0]
// 取字符串的第一个字符

fmt.Printf("str = %s, len = %d\n", str,
len(str)) //内置的函数len()来取字符串的长度

fmt.Printf("str[0] = %c, ch = %c\n", str[0],
ch)

```

//` (反引号)括起的字符串为Raw字符串，即字符串在代码中的形式就是打印时的形式，它没有字符转义，换行也将原样输出。

```

str2 := `hello
mike \n \r测试
`

fmt.Println("str2 = ", str2)

/*
    str2 =  hello
           mike \n \r测试
*/

```

复数类型

复数实际上由两个实数（在计算机中用浮点数表示）构成，一个表示实部（real），一个表示虚部（imag）。

```

var v1 complex64 // 由2个float32构成的复数类型
v1 = 3.2 + 12i
v2 := 3.2 + 12i      // v2是complex128类型
v3 := complex(3.2, 12) // v3结果同v2

fmt.Println(v1, v2, v3)
//内置函数real(v1)获得该复数的实部
//通过imag(v1)获得该复数的虚部
fmt.Println(real(v1), imag(v1))

```

fmt包的格式化输出输入

格式说明

格式	含义
%%	一个%字面量
%b	一个二进制整数值(基数为2)，或者是一个(高级的)用科学计数法表示的指数为2的浮点数
%c	字符型。可以把输入的数字按照ASCII码相应转换为对应的字符
%d	一个十进制数值(基数为10)
%e	以科学记数法e表示的浮点数或者复数值
%E	以科学记数法E表示的浮点数或者复数值
%f	以标准记数法表示的浮点数或者复数值
%g	以%e或者%f表示的浮点数或者复数，任何一个都以最为紧凑的方式输出
%G	以%E或者%f表示的浮点数或者复数，任何一个都以最为紧凑的方式输出
%o	一个以八进制表示的数字(基数为8)
%p	以十六进制(基数为16)表示的一个值的地址，前缀为0x,字母使用小写的a-f表示
%q	使用Go语法以及必须时使用转义，以双引号括起来的字符串或者字节切片[]byte，或者是以单引号括起来的数字
%s	字符串。输出字符串中的字符直至字符串中的空字符（字符串以'\0'结尾，这个'\0'即空字符）
%t	以true或者false输出的布尔值
%T	使用Go语法输出的值的类型
%U	一个用Unicode表示法表示的整型码点，默认值为4个数字字符

格式	含义
%v	使用默认格式输出的内置或者自定义类型的值，或者是使用其类型的String()方式输出的自定义值，如果该方法存在的话
%x	以十六进制表示的整型值(基数为十六)，数字a-f使用小写表示
%X	以十六进制表示的整型值(基数为十六)，数字A-F使用小写表示

输出

//整型

```
a := 15
```

```
fmt.Printf("a = %b\n", a) //a = 1111
```

```
fmt.Printf("%%\n") //只输出一个%
```

//字符

```
ch := 'a'
```

```
fmt.Printf("ch = %c, %c\n", ch, 97) //a, a
```

//浮点型

```
f := 3.14
```

```
fmt.Printf("f = %f, %g\n", f, f) //f =
3.140000, 3.14
```

```
fmt.Printf("f type = %T\n", f) //f type =
float64
```

//复数类型

```

v := complex(3.2, 12)

fmt.Printf("v = %f, %g\n", v, v) //v =
(3.200000+12.000000i), (3.2+12i)

fmt.Printf("v type = %T\n", v)    //v type =
complex128


//布尔类型

fmt.Printf("%t, %t\n", true, false) //true,
false


//字符串

str := "hello go"

fmt.Printf("str = %s\n", str) //str = hello go

```

输入

```

var v int

fmt.Println("请输入一个整型: ")

fmt.Scanf("%d", &v)

//fmt.Scan(&v)

fmt.Println("v = ", v)

```

Scan和Scanf的区别

Scan和Scanf

Scanf分隔符

类型转换

Go语言中不允许隐式转换，所有类型转换必须显式声明，而且转换只能发生在两种相互兼容的类型之间。

```
var ch byte = 97
//var a int = ch //err, cannot use ch (type
byte) as type int in assignment
var a int = int(ch)
```

类型别名

```
type bigint int64 //int64类型改名为bigint
var x bigint = 100

type (
    myint int    //int改名为myint
    mystr string //string改名为mystr
)
```

运算符

算数运算符

运算符	术语	示例	结果
+	加	10 + 5	15
-	减	10 - 5	5
*	乘	10 * 5	50
/	除	10 / 5	2

运算符	术语	示例	结果
%	取模(取余)	10 % 3	1
++	后自增，没有前自增	a=0; a++	a=1
--	后自减，没有前自减	a=2; a--	a=1

关系运算符

运算符	术语	示例	结果
==	相等于	4 == 3	false
!=	不等于	4 != 3	true
<	小于	4 < 3	false
>	大于	4 > 3	true
<=	小于等于	4 <= 3	false
>=	大于等于	4 >= 1	true

逻辑运算符

运算符	术语	示例	结果
!	非	!a	如果a为假，则!a为真； 如果a为真，则!a为假。
&&	与	a && b	如果a和b都为真，则结果为真，否则为假。
	或	a b	如果a和b有一个为真，则结果为真，二者都为假时，结果为假。

位运算符

运算符	术语	说明	示例
&	按位与	参与运算的两数各对应的二进制位相与	60 & 13 结果为12
	按位或	参与运算的两数各对应的二进制位相或	60 13 结果为61
^	异或	参与运算的两数各对应的二进制位相异或，当两对应的二进制位相异时，结果为1	60 ^ 13 结果为240
<<	左移	左移n位就是乘以2的n次方。左边丢弃，右边补0。	4 << 2 结果为16
>>	右移	右移n位就是除以2的n次方。右边丢弃，左边补位。	4 >> 2 结果为1

赋值运算符

运算符	说明	示例
=	普通赋值	$c = a + b$ 将 $a + b$ 表达式结果赋值给 c
+=	相加后再赋值	$c += a$ 等价于 $c = c + a$
-=	相减后再赋值	$c -= a$ 等价于 $c = c - a$
*=	相乘后再赋值	$c = a$ 等价于 $c = c * a$
/=	相除后再赋值	$c /= a$ 等价于 $c = c / a$
%=	求余后再赋值	$c \% = a$ 等价于 $c = c \% a$
<<=	左移后赋值	$c <<= 2$ 等价于 $c = c << 2$
>>=	右移后赋值	$c >>= 2$ 等价于 $c = c >> 2$
&=	按位与后赋值	$c \&= 2$ 等价于 $c = c \& 2$

运算符	说明	示例
$\wedge=$	按位异或后赋值	$c \wedge= 2$ 等价于 $c = c \wedge 2$
$ =$	按位或后赋值	$c = 2$ 等价于 $c = c 2$

其他运算符

运算符	术语	示例	说明
$\&$	取地址运算符	$\&a$	变量a的地址
$*$	取值运算符	$*a$	指针变量a所指向内存的值

运算符优先级

Go语言中，一元运算符拥有最高的优先级，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	\wedge $!$
6	$*$ $/$ $\%$ $<<$ $>>$ $\&$ $\&\wedge$
5	$+$ $-$ $ $ \wedge
4	$==$ $!=$ $<$ $<=$ $>=$ $>$
3	$<-$
2	$\&\&$
1	$ $

流程控制

选择结构

if语句

if

```
var a int = 3
if a == 3 { //条件表达式没有括号
    fmt.Println("a==3")
}
```

//支持一个初始化表达式，初始化语句和条件表达式直接需要用分号分隔

```
if b := 3; b == 3 {
    fmt.Println("b==3")
}
```

if..else

```
if a := 3; a == 4 {
    fmt.Println("a==4")
} else { //左大括号必须和条件语句或else在同一行
    fmt.Println("a!=4")
}
```

if ... else if ... else

```
if a := 3; a > 3 {  
    fmt.Println("a>3")  
} else if a < 3 {  
    fmt.Println("a<3")  
} else if a == 3 {  
    fmt.Println("a==3")  
} else {  
    fmt.Println("error")  
}
```

switch 语句

Go里面switch默认相当于每个case最后带有break，匹配成功后不会自动向下执行其他case，而是跳出整个switch，但是可以使用fallthrough强制执行后面的case代码：

```
var score int = 90  
  
switch score {  
case 90:  
    fmt.Println("优秀")  
    //fallthrough  
case 80:  
    fmt.Println("良好")  
    //fallthrough  
case 50, 60, 70:  
    fmt.Println("一般")  
}
```

```

        //fallthrough
    default:
        fmt.Println("差")
    }

```

可以使用任何类型或表达式作为条件语句：

```

//1
switch s1 := 90; s1 { //初始化语句;条件
case 90:
    fmt.Println("优秀")
case 80:
    fmt.Println("良好")
default:
    fmt.Println("一般")
}

//2
var s2 int = 90
switch { //这里没有写条件
case s2 >= 90: //这里写判断语句
    fmt.Println("优秀")
case s2 >= 80:
    fmt.Println("良好")
default:
    fmt.Println("一般")
}

```

```
//3
switch s3 := 90; { //只有初始化语句，没有条件
case s3 >= 90: //这里写判断语句
    fmt.Println("优秀")
case s3 >= 80:
    fmt.Println("良好")
default:
    fmt.Println("一般")
}
```

循环语句

for

```
var i, sum int

for i = 1; i <= 100; i++ {
    sum += i
}

fmt.Println("sum = ", sum)
```

range

关键字 range 会返回两个值，第一个返回值是元素的数组下标，第二个返回值是元素的值：

```

s := "abc"

for i := range s { //支持
string/array/slice/map。
    fmt.Printf("%c\n", s[i])
}

for _, c := range s { // 忽略 index
    fmt.Printf("%c\n", c)
}

for i, c := range s {
    fmt.Printf("%d, %c\n", i, c)
}

```

跳转语句

break和continue

在循环里面有两个关键操作break和continue，break操作是跳出当前循环，continue是跳过本次循环。

```

for i := 0; i < 5; i++ {
    if 2 == i {
        //break      //break操作是跳出当前循环
        continue //continue是跳过本次循环
    }
    fmt.Println(i)
}

```

```
}
```

注意：break可用于for、switch、select，而continue仅能用于for循环。

goto

用goto跳转到必须在当前函数内定义的标签：

```
func main() {  
    for i := 0; i < 5; i++ {  
        for {  
            fmt.Println(i)  
            goto LABEL //跳转到标签LABEL，从标签处，执  
行代码  
        }  
    }  
  
    fmt.Println("this is test")  
  
LABEL:  
    fmt.Println("it is over")  
}
```