Golang

Golang语言函数

Golang语言条件语句与循环语句

Golang语言运算符

Go语言变量与常量

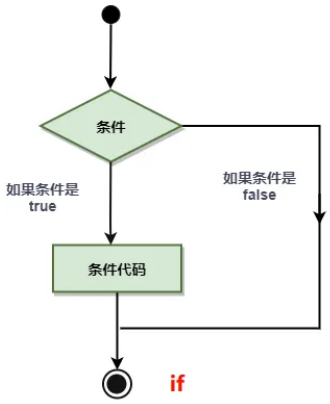
Golang语言的数据类型

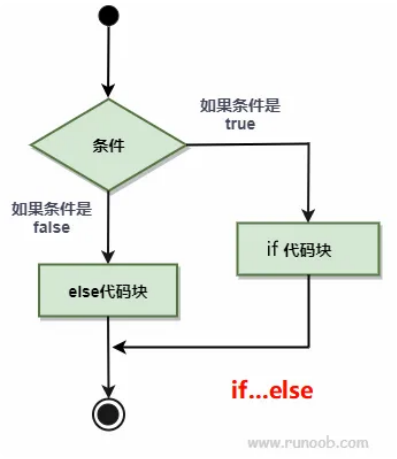
Golang基础语法

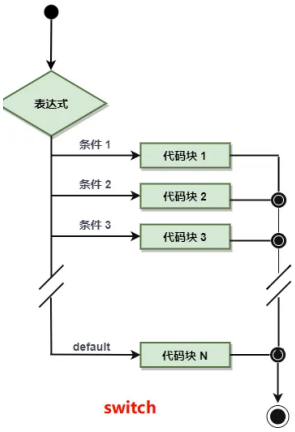
Golang语言及其结构

Golang语言条件语句与循环语句

1. 条件语句







语句	描述
if 语句	**if 语句** 由一个布尔表达式后紧跟一个或多个语句组成。 如果表达式为true,其后紧跟的语句块执行, 否则不执行.
if...else 语句	**if 语句** 后可以使用可选的 **else 语句**, else 语句中的表达式在布尔 表达式为 false 时执行。
if 嵌套语句	你可以在 **if** 或 **else if** 语句中嵌入一个或多个 **if** 或 **else if** 语句。
switch 语句	**switch** 语句用于基于不同条件执行不同动作。
select 语句	**select** 语句类似于 **switch** 语句，但是select会随机执行一个可运 行的case。如果没有case可运行，它将阻塞，直到有case可运 行。

语法 Go

```go
// if
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
}

// if...else
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
} else {
  /* 在布尔表达式为 false 时执行 */
}

// if嵌套
if 布尔表达式 1 {
   /* 在布尔表达式 1 为 true 时执行 */
   if 布尔表达式 2 {
      /* 在布尔表达式 2 为 true 时执行 */
   }
}
```

1.1. switch

- switch 语句用于基于不同条件执行不同动作，每一个 case 分支都是唯一的，**从上至下逐一测试，直到匹配为止。**
- switch默认情况下case后面子带break语句,匹配成功后就不会再执行其他的case,所以后面不需要加break
  - 如果执行了一个case后,想要继续执行后面的case, 可以试用fallthrough
- switch的**条件语句处的变量var可以是任何类型**,不被局限于常量或整数, 但**必须是相同的类型,或者最终结果为相同类型的表达式**
  - 可以**同时测试多个可能符合条件的值**,使用 ","分隔
  - switch可以用于判断**变量的类型**
    - `x.(type)`

1.1.1.1. fallthrough

switch Go

```go
// switch 值
switch var1 {
    case val1:
        ...
    case val2:
        ...
    default:
        ...
}

// switch 类型
switch x.(type){
    case type:
       statement(s);
    case type:
       statement(s);
    /* 你可以定义任意个数的case */
    default: /* 可选 */
       statement(s);
}

```

1.2. select

- 类似于switch语句, 但只能用于通道操作, **每个case必须是一个通道操作**,要么是发送要么是接受
- select语句会**监听所有指定的通道上的操作**,一旦其中的一**个通道准备好就会执行相应的代码块**
  - 如果**多个**通道都**准备好**,那么select语句会**随机选择一个**通道**执行**,如果所有通道**都没有准备好**, 那么执行**default**块中的代码

select Go

```go
// select
select {
  case <- channel1:
    // 执行的代码
  case value := <- channel2:
    // 执行的代码
  case channel3 <- value:
    // 执行的代码

    // 你可以定义任意数量的 case

  default:
    // 所有通道都没有准备好，执行的代码
}
```

- 注:
  - 每个case都必须是一个通道
  - 所有channel表达式都会被求值
  - 所有被发送的表达式都会被求值
  - 如果任意通道可以进行,他就会执行,其他被忽略
  - 如果有多个case都可以进行,select会随机公平的选出一个执行,其他不会执行
    - 否则
      - 如果有default子句,则执行该子句
      - 如果没有default子句,select将阻塞,知道某个通道可以运行,Go不会重新对channel或值进行求值

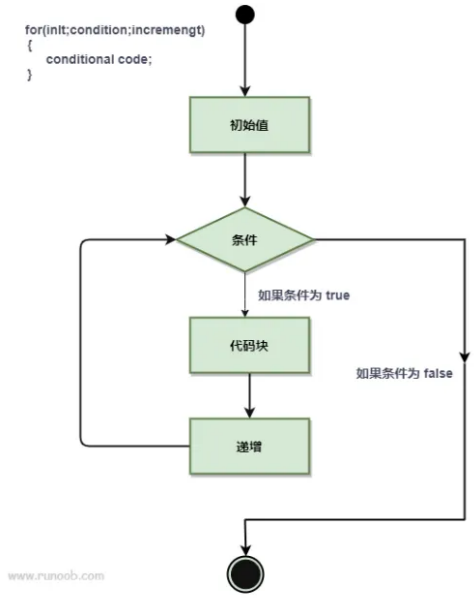
2. 循环语句

2.1. 循环处理语句

2.1.1. for循环

循环类型	描述
for 循环	重复执行语句块

循环嵌套	在 for 循环中嵌套一个或多个 for 循环



i. `for 初始值; 终止条件; 步长 { 循环体 }` 与C中的for一致

ii. `for 终止条件 { 循环体 }` 与C中的while一致

iii. `for { 循环体 }` 与C中的for(;;)一致

iv. for循环的range格式可以对slice, map, 数组, 字符串等进行迭代循环

for循环的range格式 Go

```go
for key, value := range oldMap {
    newMap[key] = value
}

//key和value可以省略
```

- 可以**只读取其中的key**: `for key := range oldMap`
- 也可以**只读取其中的value**: `for _, value := range oldMap`

for-each range循环实例 Go

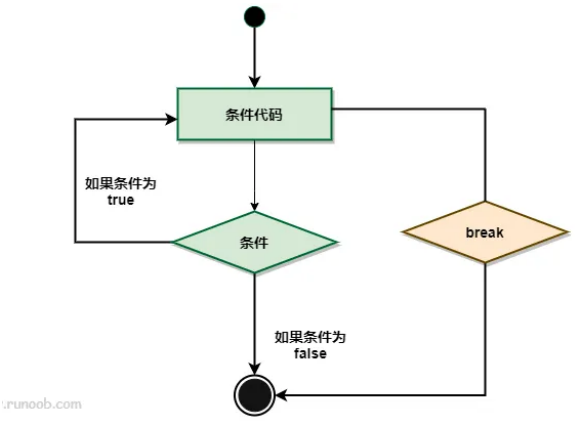
```go
// 正常写法
package main
import "fmt"

func main() {
   strings := []string{"google", "runoob"}
   for i, s := range strings {
      fmt.Println(i, s)
   }


   numbers := [6]int{1, 2, 3, 5}
   for i,x:= range numbers {
      fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)
   }
}

// 省略key和value
package main
import "fmt"

func main() {
    map1 := make(map[int]float32)
    map1[1] = 1.0
    map1[2] = 2.0
    map1[3] = 3.0
    map1[4] = 4.0

    // 读取 key 和 value
    for key, value := range map1 {
      fmt.Printf("key is: %d - value is: %f\n", key, value)
    }

    // 读取 key
    for key := range map1 {
      fmt.Printf("key is: %d\n", key)
    }

    // 读取 value
    for _, value := range map1 {
      fmt.Printf("value is: %f\n", value)
    }
}
```

- Go语言**允许用户在循环内使用循环**

2.2. 循环控制语句

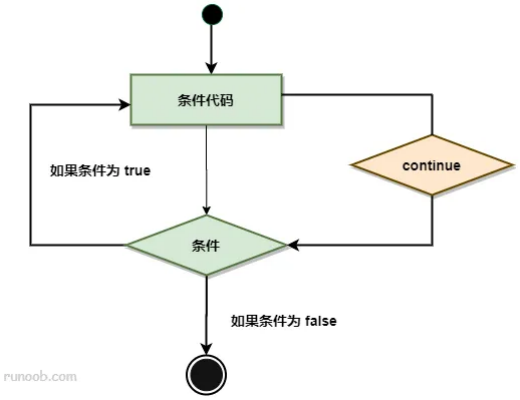
控制语句	描述
break 语句	经常用于中断当前 for 循环或跳出 switch 语句
continue 语句	跳过当前循环的剩余语句，然后继续进行下一轮循环。
goto 语句	将控制转移到被标记的语句。

2.2.1. break



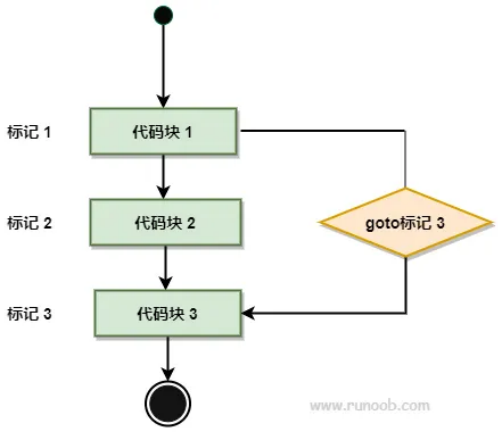
- break语句用于**终止当前循环或者switch语句的执行**,并**跳出该循环或者switch语句的代码块**
- 用途
  - 循环语句中的**跳出循环, 并开始执行循环之后的语句**
  - break在**switch语句中在执行一条case后跳出语句**的作用
  - break**可应用在select语句中**
    - break语句并**不能直接用于跳出select语句本身**,因为他**是非阻塞的**,会**一直等待所有的通信操作都准备就绪**.如果需要提前结束select语句的执行,可以试用return或goto语句来达到相同的效果
      - 注: 使用return语句会立即终止当前的函数执行,所以请根据实际需求来决定在select中使用何种方式提前结束
  - 在**多重循环中**,可以**用标号label标出想break的循环**
- 语法: `break`

2.2.2. continue



- Go中的continue语句有点像break语句,但continue不是跳出循环, 而是跳过当前循环执行下一次循环
- for循环中,执行continue语句会触发for增量语句的执行
  - 在多重循环中,可以使用标号label标出想continue的循环
- 语法: `continue;`

2.2.3. goto



- Go语言中的goto语句可以无条件地转移到过程中指定的行
- goto语句通常与条件语句配合使用,可以用来实现条件转移,构成循环,跳出循环体等功能
  - **避免使用goto语句,避免程序流程混乱.**

goto语句用法 Go

```
goto label;
..
.
label: statement;

```

2.3. 无限循环

条件语句永远不为false的循环语句

可以通过for循环语句中值设置一个条件表达式来执行无限循环

无限循环例子 Go

```go
package main

import "fmt"

func main() {
    for true  {
        fmt.Printf("这是无限循环。\n");
    }
}
```

Golang语言运算符

1. 运算符分类

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位移运算符
- 赋值运算符
- 其他运算符

2. 算术运算符

运算符	描述	实例
+	相加	A + B
–	相减	A – B
*	相乘	A * B
/	相除	B / A
%	求余	B % A
++	自增	A++
––	自减	A––

- **注**
  - 如果没有浮点数参与运算, "/" 运算的结果会舍去小数部分, 不管将其赋给什么类型的值
  - "%" 取余运算的结果与被除数的符号一致
  - 自增(++)自减(––) 只能放在变量的后面,**不能放在变量的前边**
  - 自增(++)自减(––) 只能单独使用,**不能赋给其他变量**

3. 关系运算符

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)
!=	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)
>	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)
<	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)
>=	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)
<=	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)

4. 逻辑运算符

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

5. 位运算符

5.1. 与,或,异或运算规则

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

5.2. 位运算

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 0 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 00 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 0 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 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 1111

6. 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

7. 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

8. 运算符优先级

优先级	运算符
5	* / % << >> & &^

4	+ – | ^
3	== != < <= > >=
2	&&
1	||

- **注**: 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**

Go语言变量与常量

1. 变量

变量名由数字, 字母, 下划线组成,且首字符不能为数字

1.1. 声明

- 使用 `var` 关键字　`var 变量名 类型`

变量声明 Go

```go
// 单个变量
var a string = "abc"

// 多个变量
var b, c int = 1, 2
```

a. 指定变量类型，如果没有初始化，则变量默认为**零值**(变量没有初始化时系统默认设置的值)

- `var 变量名 类型`　　`变量名 = 值`

```go
package main
import "fmt"
func main() {

    // 声明一个变量并初始化
    var a = "RUNOOB"
    fmt.Println(a)

    // 没有初始化就为零值
    var b int
    fmt.Println(b)

    // bool 零值为 false
    var c bool
    fmt.Println(c)
}

/*
结果
RUNOOB
0
false
*/
```

b. 根据值自行判断变量的类型

- `var 变量名 = 值`

```go
package main
import "fmt"
func main() {
    var d = true
    fmt.Println(d)
}

/*
结果
true
*/
```

c. 使用 `:=` 相当于使用 `var` 定义并使用 "=" 赋值

- `var x int = 23` 相当于 `x := 23`

    - 注: 如果变量已经使用var声明过了, 再使用 := 声明变量, 就会产生编译错误
- 上述三种方式都可以进行多变量声明
    - 同一类型的多个变量可以声明在同一行
    - 多变量可以在同一行进行赋值(并行/同时赋值)

多变量声明 Go

```go
//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
)

```

- 其他使用方式
    - 交换两个变量的值
        - `a, b = b, a` 要求两个变量的类型是相同的
        - 空白标识符"_" 也被用于抛弃值: 5在 `_, b=5, 7` 中被抛弃
            - "_" 是一个只写变量,得不到值
                - **Go语言中必须使用所有被声明的变量,但有时并不需要使用从一个函数中得到的所有返回值**

2. 值类型和引用类型

- 所有相int, float, bool, string 这些基本类型都属于值类型, 这些类型的变量直接指向所在内存中的值.
    - 值类型之间使用 "=" 赋值时, 如j=i ,实际上是在**内存中将**i的**值进行拷贝**
- 可以通过 `&i` 来获取变量 i的内存地址, 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,这个内存地址成为**指针**

- 引用类型之间使用 "=" 赋值时, 如 r1=r2 ,只有引用(地址)被复制, 如果后续r1 的值被改变, 那么这个值得所有引用都会被指向修改后的内容

3. 报错的形式

a. 如果已经定义过的变量被再次使用 `:=` ,会提示 `no new variables on left side of :=`

b. 如果在定义一个变量之前就使用它,会提示 `undefined: 变量名`

c. 如果定义了一个**局部变量**却没有在相同的代码块中使用它,同样会报错: **`a declared but not used`**

i. **全局变量允许定义而不使用.**

4. 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, 复数)和字符串型**

4.1. 常量的定义与应用

- `const 常量名 [类型] = 值`
  - 类型可以省略, 编译器可以根据变量的值来推断其类型
  - 多个相同类型的常量可以同时声明
  - 常量可以多重赋值

常量定义及应用 Go

```go
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)
}
```

- 常量可以用作枚举

常量用作枚举 Go

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

```

- 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

```go
package main

import "unsafe"
const (
    a = "abc"
    b = len(a)
    c = unsafe.Sizeof(a)
)

func main(){
    println(a, b, c)
}

/*
结果
abc 3 16

*/
```

4.2. 特殊常量 iota

iota, 特殊常量, 可以认为是一个可以被编译器修改的常量

> iota 在 const关键字出现时将被重置为 0(const 内部的第一行之前)，const 中每新增一行常量声明将使 iota 计数一次(iota 可理解为 const 语句块中的行索引)。

- iota可以被用作枚举

```go
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

①

```go
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)
}

/*
结果
0 1 2 ha ha 100 100 7 8
*/
```

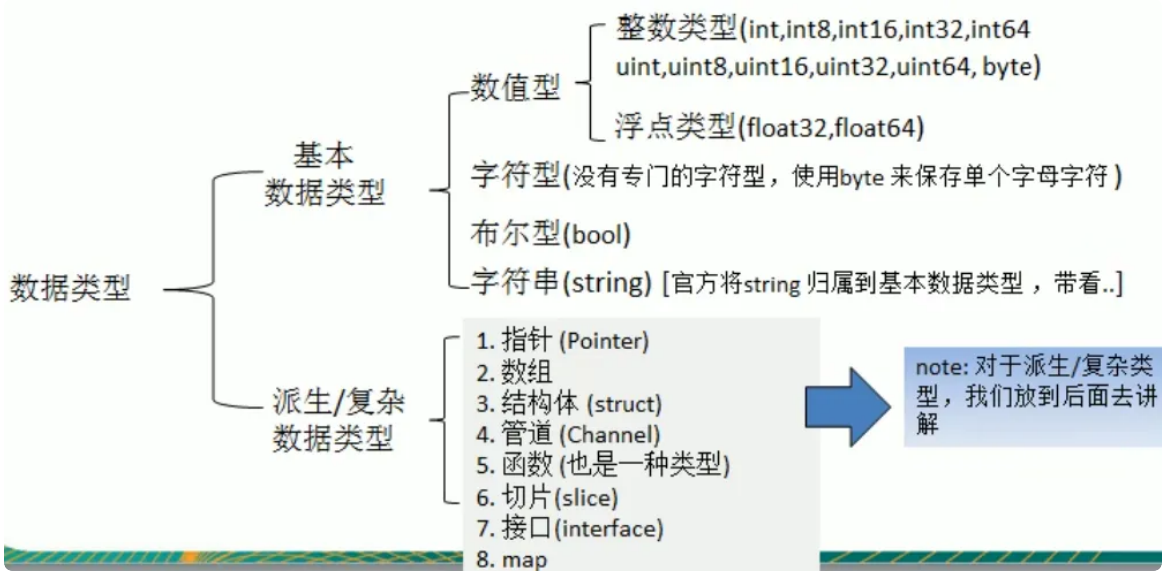
②

Go

```go
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)
}

/*
结果
i= 1
j= 6
k= 12
l= 24
*/
/*
解释
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。

简单表述:

i=1：左移 0 位，不变仍为 1。
j=3：左移 1 位，变为二进制 110，即 6。
k=3：左移 2 位，变为二进制 1100，即 12。
l=3：左移 3 位，变为二进制 11000，即 24。
注：<<n==*(2^n)。
*/
```

Golang语言的数据类型

1. 数据类型分类



- 基本数据类型
  - 布尔型(bool)
    - 值值可以是**常量true或false**
  - 数字类型
    - 整型int和浮点型float32, float64
    - Go支持整型和浮点型数字, 并且支持附属, 其中位的运算采用补码
  - 字符串类型(String)
    - 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
- 派生类型
  - 指针类型(Pointer)
  - 数组类型
  - 结构化类型(struct)
  - 管道(Channel)
  - 函数类型
  - 切片类型(slice)

- 接口类型(interface)
- Map类型

2. 数字类型

- Go有基于框架的类型:**int, uint, uintptr**
  - int是有符号的整数类型, 大小取决于具体的平台,(在32位平台上， `int` 通常是32位的；在64位平台上， `int` 通常是64位的。)
  - uint是无符号的证书类型,大小也取决于平台
    - 用于**表示非负整数**,因为不需要存储符号位,可以**比相同大小的int类型表示更大的正整数**
  - uintptr是一种无符号整数类型,大小足以存储任意指针的位模式, 通常用于底层编程和与操作系统交互的场景中
    - 主要用于将指针转换为整数类型,以便进行底层操作, 还可以作为与C语言交互时的中介类型
  - **三者可以进行隐式和显示转换**, 但由于大小和表示范围可能不同,在进行类型转换时需谨慎处理
- Golang中没有字符型, **使用byte保存单个字母字符(不能存汉字(UTF-8))**
- **常用**
  - 整型
    - uint

类型		值	默认值
uint8	无符号8位整型	0 到 255	0
uint16	无符号16位整型	0 到 65535	0
uint32	无符号32位整型	0 到 4294967295	0
uint64	无符号64位整型	0 到 18446744073709551615	0

    - int

类型		值	默认值
int8	有符号8位整型	-128 到 127	0
int16	有符号16位整型	-32768 到 32767	0

int32	有符号32位整型	-2147483648 到 2147483647	0
int64	有符号64位整型	-9223372036854775808 到 9223372036854775807	0

- 浮点型

类型	值	默认值
float32	IEEE-754 32位浮点型数	0.0
float64	IEEE-754 64位浮点型数	0.0
complex64	32 位实数和虚数	0+0i
complex128	64 位实数和虚数	0+0i

- 其他

类型	值	默认值
byte	类似 uint8　0到255	0
rune	类似 int32　-2147483648 到 2147483647 可以说是int32的别名,但rune表示一个Unicode码	0
uint	32 或 64 位	0
int	32或64位	0
uintptr	无符号整型，用于存放一个指针	0
bool	true或false	false
string		""

- string
  - 注意事项
    - Go语言的字符串使用UTF-8编码标识Unicode文本, 不会有乱码问题
    - 在Go中,字符串是不可变的　　一旦赋值就不可修改
    - 字符串的两种表示形式
      1. **双引号""**,会识别转义字符

2. **反引号``**,以字符串的原生形式输出,包括换行和特殊字符,可以实现**防止攻击**, 输出源代码等效果

- 字符串拼接使用 "+" , 也可以使用 "+="
  - 当需要拼接的字符串很多时,可以分行写
    - 注意,此时 "+" 要写在上一行

字符串很长或很多拼接时

```
//此时是不会报错的
var str="hello" + "world" + "hello" + "world" +
"hello" + "world" +"hello" + "world" +
"hello" + "world" +"hello" + "world" +
```

类型	默认值
指针	nil
数组	nil
结构体	nil
管道	nil
接口	nil
切片	nil
map	nil

?

Golang基础语法

1. 标记

- Golang程序可以有多个标记组成,可以是
  - 关键字
  - 标识符
  - 常量
  - 字符串
  - 符号

例子 Go

```go
fmt.Println("Hello, World!")

// 每一行是一个
1. fmt
2. .
3. Println
4. (
5. "Hello, World!"
6. )

```

2. 行分隔符

- 在Go程序中,**一行代表一个语句结束**,不需要以分号";"结束
- 如果**多个语句写在同一行,必须使用分号";"人为区分　　不推荐**

3. 注释

- 单行注释:　//
- 多行注释:　/**/

注释 Go

```go
//这是一个单行注释

/*
这是
一个
多行注释
或者说块注释
*/
```

4. 标识符

- 用途:**为变量,类型等程序实体命名**
- **组成**
  - 字母(A-Z,a-z)
  - 数字(0-9)
  - 下划线("_")
    - 下划线在Go中是一个特殊的标识符,称为**空标识符**,可以**用作占位符,用于丢弃值或忽略某个返回值**,赋给下划线的任何值都会被忽略
- **注意**
  - Go语言**区分大小写**
  - **首字符**必须是字母或下划线,**不能是数字**
  - **不能是关键字**
  - **不能包含运算符,不能包含空格**
  - **使用前必须先声明**
  - **同一作用域内必须唯一**
  - 包名通常与其所在的目录名保持一致,并应避免与标准库冲突,**带有main函数的包必须将其package定义为main**
  - 在引用自定义包时Go语言会自动补充 `$GOPATH/src/` 路径(其中GOPATH为环境变量),并将路径中的文件夹用 "/"分割

5. 字符串

5.1. 字符串拼接

- 在Go中,字符串的拼接使用 "+" 实现

字符串拼接 Go

```go
package main
import "fmt"
func main() {
    fmt.Println("Google" + "Runoob")
}

/* 结果
GoogleRunoob
*/

```

5.2. 格式化字符串

- Go语言中使用 `fmt.Sprintf` 或 `fmt.Printf` **格式化字符串**并赋值给新串
  - Sprintf根据格式化参数生成格式化的字符串并返回该字符串
  - Printf根据格式化参数生成格式化的字符串并写入标准输出

Sprintf实例 Go

```go
package main

import (
    "fmt"
)

func main() {
   // %d 表示整型数字, %s 表示字符串
    var stockcode=123
    var enddate="2020-12-31"
    var url="Code=%d&endDate=%s"
    var target_url=fmt.Sprintf(url,stockcode,enddate)
    fmt.Println(target_url)
}

/*
结果
Code=123&endDate=2020-12-31
*/
```

Printf实例 Go

```go
package main

import (
    "fmt"
)

func main() {
   // %d 表示整型数字, %s 表示字符串
    var stockcode=123
    var enddate="2020-12-31"
    var url="Code=%d&endDate=%s"
    fmt.Printf(url,stockcode,enddate)
}

/*
Code=123&endDate=2020-12-31
*/
```

- **fmt.Printf和fmt.Println之间的区别**
  - **格式化输出**
    - Printf允许使用格式化字符串来指定输出格式,可以试用格式化动词(如:%d, %s,

%v等)来控制输出数据的格式

- Println不支持格式化字符串,直接将给定的参数转换为字符串(使用默认格式),并在每个参数之间添加一个空格,最后添加一个换行符

格式化输出 Go

```go
// Printf
fmt.Printf("Hello, %s! You have %d new messages.\n", "Alice", 5)

/*
输出
Hello, Alice! You have 5 new messages.
*/

// Println
fmt.Println("Hello,", "Alice!", "You have", 5, "new messages.")

/*
输出
Hello, Alice! You have 5 new messages.
*/
```

- 换行符
  - Printf不会在输出的末尾添加换行符,如果需要输出后换行,需要在格式化字符串中显示的添加"\n"
  - Println会自动地在每个输出的末尾添加换行符
- 用途
  - Printf通常用于需要精确控制输出格式的场景
  - Println用于简单的输出需求,不需要特定的格式,只是需要快速的在控制台打印信息

6. 关键字

- 常用**关键字,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

- **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**,**变量**,**常量**,**运算符**,**类型**和**函数组成**

7. 符号

- 程序中可能会使用到的分隔符:
  - 括号"()"
  - 中括号"[]"
  - 大括号"{}"
  - 程序中可能会使用到的标点符号
    - "."、","、";"、":"、"..."
  - 空格
    - Go语言中,空格常用于分割标识符,关键字,运算符和表达式,以提高代码的可读性
    - **变量的声明**必须使用空格隔开
    - **运算符和操作数之间**要使用空格能让程序更易阅读
    - 在**变量与运算符间**加入空格，程序看起来更加美观
    - 在**关键字和表达式之间**要使用空格
    - 在函数调用时，**函数名和左边等号之间**要使用空格，**参数之间**也要使用空格

Golang语言及其结构

1. Golang

1.1. 特点

- 开源的、结构简单、可靠且高效
- 简洁、快速、安全
- 并行、内存管理、数组安全、编译迅速

1.2. 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。

1.3. 简单Go程序

hello.go文件 Go

```go
package main    //每个Go程序都应包含一个名为main的包
import "fmt"
func main(){
    fmt.Println("hello!")
}
```

- **执行**
  - 使用 `go build` 命令进行**编译**生成二进制.exe文件
  - 使用 `go run` 命令**直接执行**.go源代码文件

命令窗口中执行 Go

```
// go run命令
$ go run hello.go
hello!  //执行结果

// go build命令
$ go build hello.go
$ ls    //显示有那些文件
hello   hello.go    //显示结果
$ ./hello
hello!  //执行hello.exe文件
```

2. 语言结构

2.1. 组成

- Golang语言包含
  - 包声明(package)
  - 引入包(import)
  - 函数(func)
  - 变量
  - 语句&表达式
  - **注释**(和Java相差不多)
    - // **单行注释**
    - /**/ **多行注释**(块注释)

2.2. 部分说明

示例代码 Go

```go
package main

import "fmt"

func main() {
   /* 这是我的第一个简单的程序 */
   fmt.Println("Hello, World!")
}
```

i. 包声明(package)

- 包声明(package)必须位于非注释的第一行
- `package main` 表示一个**独立的可执行程序**,每个Go程序都包含一个名为main的包

ii. **引入包**(import)

- 单包引入: `import "包名"`
- 多宝引入:

引入多个包 Go

```go
import (
    "包名1"
    "包名2"
    ...
)
```

  - 多个包同时引入时使用"()",同时**每一个包单独一行**
  - 可以为每个引入的包起别名(在包名之前起别名)

起别名 Go

```go
import (
    "fmt"
    alias "path/to/some/other/package"
)

func main() {
    fmt.Println("Hello, World!")
    alias.SomeFunction()
}
```

- 局部引入(仅在文件中使用)
  - 在包名前为其起别名 "_"

- 此种方法可以用在想要**引入一个包,但不使用其中的任何内容,只执行init函数**

- 路径引入:可以使用路径来标识报的位置,可以是相对路径(相对于当前模块或工作区)或绝对路径(通常是模块化的路径)

  路径引入例子 Go

  ```go
  import (
      "github.com/user/repo/subpkg"
  )
  ```

- 使用 "." (不建议)
  - 在包名前添加"." ,意味着**直接使用包中导出的名称而不需要使用前缀**
    - 可能会出现冲突问题

    使用 "." Go

    ```go
    package main

    import . "fmt"

    func main() {
        Println("你好") // 注意这里没有前缀 fmt.
    }
    ```

iii. 函数(func)

- func main()是程序开始时执行函数,main函数是每一个可执行程序所必须包含的,一般在启动后第一个执行(如果有init()函数,则会先执行init()函数)

iv. 标识符

- 标识符包含:常量,变量,类型,函数名,结构字段等
- 以一个大写字母开头
  - 可以被外部包的代码所使用(需要先导入所在的包),被称为**导出**(类似面向对象语言中的public)
- 以小写字母开头
  - 对包外是不可见的,但在所在的整个包的内部是可见且可用的(类似面向对象语言中的protected)

- **fmt.Println(...)**
  - 可以**将字符串输出到控制台**,并**在最后直通添加换行字符"\n"**
  - 与fmt.Print(...)可以得到相同的结果,但fmt.Print不会在最后添加换行字符"\n"