Go 语言循环语句

在不少实际问题中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。

Go 语言提供了以下几种类型循环处理语句：

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
循环语句range	类似迭代器操作，返回 (索引, 值) 或 (键, 值)

Go 语言 for 循环

for 循环是一个循环控制结构，可以执行指定次数的循环。

Go 语言的 For 循环有 3 种形式，只有其中的一种使用分号。

```
for init; condition; post { }
```

```
for condition { }
```

```
for { }
```

- init： 一般为赋值表达式，给控制变量赋初值；
- condition： 关系表达式或逻辑表达式，循环控制条件；
- post： 一般为赋值表达式，给控制变量增量或减量。

for语句执行过程如下：

- 1、先对表达式 1 赋初值；
- 2、判别赋值表达式 init 是否满足给定条件，若其值为真，满足循环条件，则执行循环体内语句，然后执行 post，进入第二次循环，再判别 condition；否则判断 condition 的值为假，不满足条件，就终止for循环，执行循环体外语句。

示例：计算 1 到 10 的数字之和

```
package main

import "fmt"

func main() {
   sum := 0
      for i := 0; i <= 10; i++ {
         sum += i
      }
   fmt.Println(sum)
}
#结果
55
```

init 和 post 参数是可选的，我们可以直接省略它，类似 While 语句。

以下实例在 sum 小于 10 的时候计算 sum 自相加后的值：

```
package main

import (
    "fmt"
)

func main() {
    sum := 1
    for sum <= 10 {
        sum += sum
    }
    fmt.Println(sum)
}
#结果
16
```

无限循环：

```
package main

import "fmt"

func main() {
   sum := 0
   for {
      sum++ // 无限循环下去
   }
   fmt.Println(sum) // 无法输出
}
```

Go 语言循环嵌套

Go 语言允许用户在循环内使用循环。

以下为 Go 语言嵌套循环的格式：

```
for [condition |  ( init; condition; increment ) | Range]
{
   for [condition |  ( init; condition; increment ) | Range]
   {
      statement(s);
   }
   statement(s);
}
```

示例：使用循环嵌套来输出 2 到 100 间的素数

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var i, j int

   for i=2; i < 100; i++ {
      for j=2; j <= (i/j); j++ {
         if(i%j==0) {
            break; // 如果发现因子，则不是素数
         }
      }
      if(j > (i/j)) {
         fmt.Printf("%d  是素数\n", i);
      }
   }
}
#结果
2  是素数
3  是素数
5  是素数
7  是素数
11  是素数
13  是素数
17  是素数
19  是素数
23  是素数
29  是素数
31  是素数
37  是素数
41  是素数
43  是素数
47  是素数
53  是素数
59  是素数
61  是素数
67  是素数
71  是素数
73  是素数
79  是素数
83  是素数
```

```
89  是素数
97  是素数
```

示例：九九乘法表

```
package main

import "fmt"

func main() {
    for m := 1; m < 10; m++ {
    /*    fmt.Printf("第%d次：\n",m) */
        for n := 1; n <= m; n++ {
            fmt.Printf("%dx%d=%d ",n,m,m*n)
        }
        fmt.Println("")
    }
}
#结果
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
1x4=4 2x4=8 3x4=12 4x4=16
1x5=5 2x5=10 3x5=15 4x5=20 5x5=25
1x6=6 2x6=12 3x6=18 4x6=24 5x6=30 6x6=36
1x7=7 2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49
1x8=8 2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64
1x9=9 2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

循环语句range

range类似迭代器操作，返回 (索引, 值) 或 (键, 值)。

for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

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

以上代码中的 key 和 value 是可以省略。可忽略不想要的返回值，或 `"_"` 这个特殊变量。

如果只想读取 key，格式如下：

```
for key := range oldMap
```

或者这样：

```
for key, _ := range oldMap
```

如果只想读取 value，格式如下：

```
for _, value := range oldMap
```

示例：

```
package main
import "fmt"

func main() {
   strings := []string{"hello", "world"}
   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)
   }
}
#结果
0 hello
1 world
第 0 位 x 的值 = 1
第 1 位 x 的值 = 2
第 2 位 x 的值 = 3
第 3 位 x 的值 = 5
第 4 位 x 的值 = 0
第 5 位 x 的值 = 0
```

示例：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)
    }
}

#结果
```

```
key is: 2 - value is: 2.000000
key is: 3 - value is: 3.000000
key is: 4 - value is: 4.000000
key is: 1 - value is: 1.000000
key is: 1
key is: 2
key is: 3
key is: 4
value is: 2.000000
value is: 3.000000
value is: 4.000000
value is: 1.000000
```

*注意，range 会复制对象。

```
package main

import "fmt"

func main() {
    a := [3]int{0, 1, 2}

    for i, v := range a { // index、value 都是从复制品中取出。
        if i == 0 { // 在修改前，我们先修改原数组。
            a[1], a[2] = 999, 999
            fmt.Println(a) // 确认修改有效，输出 [0, 999, 999]。
        }

        a[i] = v + 100 // 使用复制品中取出的 value 修改原数组。
    }

    fmt.Println(a) // 输出 [100, 101, 102]。
}
#结果
 [0 999 999]
 [100 101 102]
```

建议改用引用类型，其底层数据不会被复制。

```
package main

func main() {
    s := []int{1, 2, 3, 4, 5}
    for i, v := range s { // 复制 struct slice { pointer, len, cap }。
        if i == 0 {
            s = s[:3]  // 对 slice 的修改，不会影响 range。
            s[2] = 100 // 对底层数据的修改。
        }
        println(i, v)
    }
}
#结果
0 1
1 2
```

```
2 100
3 4
4 5
```

另外两种引用类型 map、channel 是指针包装，而不像 slice 是 struct。

for 和 for range有什么区别?

主要是使用场景不同

for可以遍历array和slice，遍历key为整型递增的map，遍历string。

for range可以完成所有for可以做的事情，却能做到for不能做的，包括遍历key为string类型的map并同时获取key和value，遍历channel。

循环控制语句

循环控制语句可以控制循环体内语句的执行过程。

GO 语言支持以下几种循环控制语句：

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。

标签（label）形式的标识符使用，即某一行第一个以冒号（`:`）结尾的单词（gofmt 会将后续代码自动移至下一行）。标签的名称是大小写敏感的，为了提升可读性，一般建议使用全部大写字母

> 1.三个语句都可以配合标签(label)使用
> 2.标签名区分大小写，定以后若不使用会造成编译错误
> 3.continue、break配合标签(label)可用于多层循环跳出
> 4.goto是调整执行位置，与continue、break配合标签(label)的结果并不相同

Go 语言 break 语句

Go 语言中 break 语句用于以下两方面：

- 用于循环语句中跳出循环，并开始执行循环之后的语句。
- break 在 switch（开关语句）中在执行一条 case 后跳出语句的作用。
- 在多重循环中，可以用标号 label 标出想 break 的循环。

break 语法格式如下：

```
break;
```

示例：在变量 a 大于 15 的时候跳出循环

```
package main

import "fmt"

func main() {
```

```
   /* 定义局部变量 */
   var a int = 10

   /* for 循环 */
   for a < 20 {
      fmt.Printf("a 的值为 : %d\n", a);
      a++;
      if a > 15 {
         /* 使用 break 语句跳出循环 */
         break;
      }
   }
}
#结果
a 的值为 : 10
a 的值为 : 11
a 的值为 : 12
a 的值为 : 13
a 的值为 : 14
a 的值为 : 15
```

示例：有多重循环，演示了使用标记和不使用标记的区别

```
package main

import "fmt"

func main() {
   // 不使用标记
   fmt.Println("---- break ----")
   for i := 1; i <= 3; i++ {
      fmt.Printf("i: %d\n", i)
      for i2 := 11; i2 <= 13; i2++ {
         fmt.Printf("i2: %d\n", i2)
         break
      }
   }

   // 使用标记
   fmt.Println("---- break label ----")
   re:
      for i := 1; i <= 3; i++ {
         fmt.Printf("i: %d\n", i)
         for i2 := 11; i2 <= 13; i2++ {
         fmt.Printf("i2: %d\n", i2)
         break re
      }
   }
}
#结果
---- break ----
i: 1
i2: 11
i: 2
i2: 11
```

```
i: 3
i2: 11
---- break label ----
i: 1
i2: 11
```

Go 语言 continue 语句

Go 语言的 continue 语句 有点像 break 语句。但是 continue 不是跳出循环，而是跳过当前循环执行下一次循环语句。

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，可以用标号 label 标出想 continue 的循环。

continue 语法格式如下：

```
continue;
```

示例：在变量 a 等于 15 的时候跳过本次循环执行下一次循环：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* for 循环 */
   for a < 20 {
      if a == 15 {
         /* 跳过此次循环 */
         a = a + 1;
         continue;
      }
      fmt.Printf("a 的值为 : %d\n", a);
      a++;
   }
}
#结果
a 的值为 : 10
a 的值为 : 11
a 的值为 : 12
a 的值为 : 13
a 的值为 : 14
a 的值为 : 16
a 的值为 : 17
a 的值为 : 18
a 的值为 : 19
```

示例：有多重循环，演示了使用标记的区别

```
package main
```

```
import "fmt"

func main() {
    // 不使用标记
    fmt.Println("---- continue ---- ")
    for i := 1; i <= 3; i++ {
        fmt.Printf("i: %d\n", i)
            for i2 := 11; i2 <= 13; i2++ {
                fmt.Printf("i2: %d\n", i2)
                continue
            }
    }

    // 使用标记
    fmt.Println("---- continue label ----")
    re:
        for i := 1; i <= 3; i++ {
            fmt.Printf("i: %d\n", i)
                for i2 := 11; i2 <= 13; i2++ {
                    fmt.Printf("i2: %d\n", i2)
                    continue re
                }
        }
}
#结果
---- continue ----
i: 1
i2: 11
i2: 12
i2: 13
i: 2
i2: 11
i2: 12
i2: 13
i: 3
i2: 11
i2: 12
i2: 13
---- continue label ----
i: 1
i2: 11
i: 2
i2: 11
i: 3
i2: 11
```

Go 语言 goto 语句

Go 语言的 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。可用来实现条件转移， 构成循环，跳出循环体等功能。

但是，在结构化程序设计中一般不主张使用 goto 语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。

goto 语法格式如下：

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

示例：在变量 a 等于 15 的时候跳过本次循环并回到循环的开始语句 LOOP 处：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* 循环 */
   LOOP: for a < 20 {
      if a == 15 {
         /* 跳过迭代 */
         a = a + 1
         goto LOOP
      }
      fmt.Printf("a的值为 : %d\n", a)
      a++
   }
}
#结果
a的值为 : 10
a的值为 : 11
a的值为 : 12
a的值为 : 13
a的值为 : 14
a的值为 : 16
a的值为 : 17
a的值为 : 18
a的值为 : 19
```