程序控制

- 顺序
  - 按照先后顺序一条条执行
  - 例如，先洗手，再吃饭，再洗碗
- 分支
  - 根据不同的情况判断，条件满足执行某条件下的语句
  - 例如，先洗手，如果饭没有做好，玩游戏；如果饭做好了，就吃饭；如果饭都没有做，叫外卖
- 循环
  - 条件满足就反复执行，不满足就不执行或不再执行
  - 例如，先洗手，看饭好了没有，没有好，一会来看一次是否好了，一会儿来看一次，直到饭好了，才可是吃饭。这里循环的条件是饭没有好，饭没有好，就循环的来看饭好了没有

单分支

```
if condition {
    代码块
}

if 5 > 2 {
    fmt.Println("5 greater than 2")
}
```

注意：Go语言中，花括号一定要跟着if、for、func等行的最后，否则语法出错。这其实就是为了解决C风格、Java风格之争。

- condition**必须**是一个**bool类型**，在Go中，**不能**使用其他类型等效为布尔值。`if 1 {}`是错误的
- 语句块中可以写其他代码
- 如果condition为true，才能执行其后代码块

多分支

```
if condition1 {
    代码块1
} else if condition2 {
    代码块2
} else if condition3 {
    代码块3
} ... {
    ...
} else if conditionN {
    代码块N
} else {
    代码块
}

```

```
15
16 a := 6
17 if a < 0 {
18     fmt.Println("negative")
19 } else if a > 0 { // 走到这里一定 a 不小于 0
20     fmt.Println("positive")
21 } else { // 走到这里一定 a 不大于、也不小于 0
22     fmt.Println("zero")
23 }
```

- 多分支结构，从上向下依次判断分支条件，只要一个分支条件成立，其后语句块将被执行，那么其他条件都不会被执行
- **前一个分支条件被测试过，下一个条件相当于隐含着这个条件**
- 一定要考虑一下else分支是有必要写，以防逻辑漏洞

```
1  // 嵌套
2  a := 6
3  if a == 0 {
4      fmt.Println("zero")
5  } else {
6      if a > 0 {
7          fmt.Println("negative")
8      } else if a >= 0 { // 走到这里一定 a 不小于 0
9          fmt.Println("positive")
10     }
11 }
```

循环也可以互相嵌套，形成多层循环。循环嵌套不易过深。

switch分支

特别注意：Go语言的switch有别于C语言的switch，case是独立代码块，不能穿透。

```
1  a := 20
2  switch a { // 待比较的是a
3  case 10:
4      fmt.Println("ten")
5  case 20:
6      fmt.Println("twenty")
7  case 30, 40, 50: // 或关系
8      fmt.Println(">=30 and <=50")
9  default:
10     fmt.Println("other")
11 }
12 或写成
13 switch a:=20;a { // 待比较的是a
14 case 10:
15     fmt.Println("ten")
16 case 20:
17     fmt.Println("twenty")
18 case 30, 40, 50: // 或关系
19     fmt.Println(">=30 and <=50")
```

```
default:
    fmt.Println("other")
}
```

```
a := 20
switch { // 没有待比较变量，意味着表达式是true，是布尔型
case a > 0:
    fmt.Println("positive")
case a < 0:
    fmt.Println("negative")
default:
    fmt.Println("zero")
}
或写成
switch a := 20; { // 没有待比较变量，意味着表达式是true，是布尔型
case a > 0: // 如果待比较值是true，a > 0如果返回true，就进入
    fmt.Println("positive")
    // fallthrough // 穿透
case a < 0: // 如果待比较值是true，a < 0如果返回true，就进入
    fmt.Println("negative")
default:
    fmt.Println("zero")
}
```

C语言的switch有穿透效果，如果想在Go语言中实现穿透效果，使用fallthrough穿透当前case语句块。但是，大家使用C语言的时候，一般都不想要使用这种穿透效果，所以，如非必要，不要使用fallthrough

特殊if

switch可以写成`switch a:=20;a`这种形式，也就是可以在表达式a之前写一个语句后接一个分号。if也可以这样

```
if score, line := 99, 90; score > line {
    fmt.Println("perfect")
} else {
    fmt.Println("good")
} // score, line作用域只能是当前if语句
```

这种写法中定义的变量作用域只能是当前if或switch。

for循环

注意：Go语言没有提供while关键字，可以用for方便的替代

C风格for

```
for [初始操作];[循环条件];[循环后操作] {
    循环体
}
```

- 初始操作：第一次进入循环前执行，语句只能执行一次，之后不再执行
- 循环条件：要求布尔值，每次进入循环体前进行判断。如果每次条件满足，就进入循环执行一次循环体；否则，循环结束
- 循环后操作：每次循环体执行完，在执行下一趟循环条件判断之前，执行该操作一次

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
} // 初始操作中的短格式定义的i的作用域只能在for中
```

```
// 特殊写法
for i := 5; i < 10; {}
for i := 5; ; {} // 没条件就相当于true

for i < 10 {} // for condition {}，condition就是循环条件

for ;; {} // 死循环
// 死循环简写如下
for {} // 死循环 相对于 for true {}
```


continue

中止当前这一趟循环体的执行，直接执行“循环后操作”后，进入下一趟循环的条件判断。

```
for i := 0; i < 10; i++ {
    if i%2 == 0 {
        continue
    }
    fmt.Println(i)
}
```

break

终止当前循环的执行，结束了。

```go
for i := 0; ; i++ {
    if i%2 == 0 {
        continue
    }
    fmt.Println(i)
    if i >= 10 {
        break
    }
} // 请问执行结果是什么？
```

除了break，函数的return结束函数执行，当然也能把函数中的循环打断。

goto和label

这是一个被很多语言尘封或者废弃的关键字，它会破坏结构化编程，但是确实能做到便利的无条件跳转。

- 跳出多重循环使用，但是问题是为什么要用多重循环？
- 到同一处标签处统一处理，例如统一错误处理。问题是，写个函数也可以实现。

有时候也能简化一些代码，但是它是双刃剑，**不要轻易使用。**

goto需要配合标签label使用，label就像代码中的锚点，goto将无条件跳到那里开始向后执行代码。

```go
for i := 0; ; i++ {
    if i%2 == 0 {
        continue
    }
    fmt.Println(i)
    if i > 10 {
        goto condition
    }
}
condition:
fmt.Println("done")
```

continue、break也可以指定label，方便某些循环使用。但是，建议不要这么写，弄不好就成了毛线团。

for range

类型	变量	Range expression	第一个值	第二个值
array or slice	a	[n]E, *[n]E []E	index i int	a[i] E
string	s	"abcd"	index i int utf-8字节偏移	unicode值 rune
map	m	map[K]V	key k K	m[k] V

类型	变量	Range expression	第一个值	第二个值
channel	chan E, <-chan E 通道或只读通道		element E	无

```
"测试" utf-8 编码为 "\xe6\xb5\x8b\xe8\xaf\x95"
```

```
for i, v := range "abcd测试" {
    fmt.Printf("%d, %[2]d, %[2]c, %#[2]x\n", i, v)
}
fmt.Println("\xe6\xb5\x8b\xe8\xaf\x95")

运行结果如下
0, 97, a, 0x61
1, 98, b, 0x62
2, 99, c, 0x63
3, 100, d, 0x64
4, 27979, 测, 0x6d4b
7, 35797, 试, 0x8bd5
测试

可以看出，索引就是字节偏移量，中文是utf-8编码，占3个字节。
%d 打印的是unicode值
%c 打印的是字符
```

```
arr := [5]int{1, 3, 5, 7, 9}
for i, v := range arr {
    fmt.Println(i, v, arr[i])
}
for i := range arr {
    fmt.Println(i, arr[i])
}
for _, v := range arr {
    fmt.Println(v)
}
```

数组、切片、映射、通道遍历后面讲。

随机数

标准库"math/rand"

我们使用的是伪随机数，是内部写好的公式计算出来的。这个公式运行提供一个种子，有这个种子作为起始值开始计算。

- src := rand.NewSource(100)，使用种子100创建一个随机数源
- rand.New(rand.NewSource(time.Now().UnixNano()))，利用当前时间的纳秒值做种子
- r10 := rand.New(src)，使用源创建随机数生成器
- r10.Intn(5)，返回[0, 5)的随机整数

```
package main

import (
```

```
    "fmt"
    "math/rand"
)

func main() {
    src := rand.NewSource(10)
    r10 := rand.New(src)
    r1 := rand.New(rand.NewSource(1))
    for i := 0; i < 10; i++ {
        fmt.Printf("%d, %d, %d\n", rand.Intn(5), r1.Intn(5), r10.Intn(5))
    }
}
```

全局随机数生成器globalRand

- 它的种子默认为1
- 如果要改变globalRand的种子，就需要使用rand.Seed(2)修改种子
- rand.Intn(5)就是使用它生成随机数