第 5 章

CHAPTER 5

流 程 控 制

流程控制是 Go 语言中必不可少的一部分，也是整个编程基础的重要一环。Go 语言的流程控制语句和其他编程语言的流程控制语句有些不同，主要体现在 Go 语言没有 while 和 do-while 语句。

Go 语言常用的流程控制包括 if 语句、switch 语句、for 语句及 goto 语句等，switch 语句和 goto 语句主要是为了简化代码、降低代码重复率，属于扩展类的流程控制语句。

5.1 条件判断

在 Go 语言中，if 语句主要用于条件判断。if 语句还有两个分支结构：if-else 语句和 else-if 语句。

5.1.1 if 单分支

在 Go 语言中，关键字 if 是用于判断某个条件(布尔型或逻辑型)的语句，如果该条件成立，则会执行 if 后由花括号“{}”括起来的代码块，否则就忽略该代码块继续执行后续的代码，伪代码如下：

```
if 条件表达式 {
    代码块
}
```

if 语句的流程图如图 5-1 所示。

例如使用 if 语句判断一个变量的大小：

```
var a int = 20
if a < 30 {
    fmt.Printf("a 小于 30\n")
}
fmt.Printf("a 的值为 %d\n", a)
```

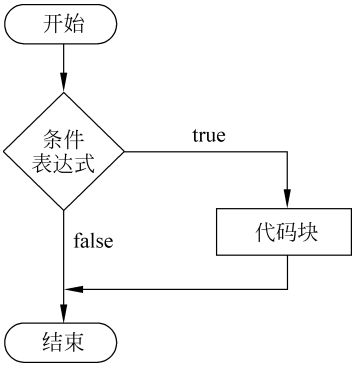


图 5-1　if 语句的流程图

这时可以使用输入函数动态地进行判断，代码如下：

```
//unit5/if 语句/1.if 单分支.go
package main

import "fmt"

func main() {
    var a int

    fmt.Printf("输入一个数字:")
    _, err := fmt.Scanf("%d", &a)
    if err != nil {
        fmt.Println("输入的数字错误")
        return
    }

    fmt.Printf("你输入的值是 %d\n", a)

    if a < 30 {
        fmt.Println("a 小于 30")
    }
}
```

5.1.2　if-else 双分支

if 语句后可以使用可选的 else 语句，else 语句中的表达式在条件表达式为 false 时执行，if 和 else 后的两个代码块是相互独立的分支，只能执行其中的一个，伪代码如下：

```
if 条件表达式 {
    代码块 1
} else {
    代码块 2
}
```

if-else 双分支的流程图如图 5-2 所示。

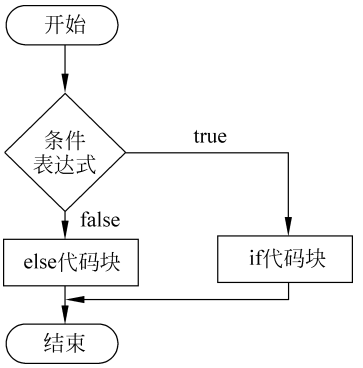


图 5-2　if-else 双分支的流程图

示例代码如下：

```
//unit5/if 语句/2.if - else 双分支.go
package main

import "fmt"

func main() {
    var day int

    fmt.Printf("几天周几:")
    _, err := fmt.Scanf("%d", &day)
    if err != nil {
        fmt.Println("输入的数字错误")
        return
    }

    if day >= 1 && day <= 5 {
        fmt.Println("今天是工作日")
    }else {
        fmt.Println("今天是休息日")
    }
}
```

5.1.3　if-else-if 多分支

除了可以直接使用 if-else 双分支语句，还可以在 else 后面继续增加 if 条件判断，语法如下：

```
if 条件表达式 1 {
   //当条件表达式 1 为 true 时执行
} else if 条件表达式 2 {
    //当条件表达式 2 为 true 时执行
```

```
} else {
    //当条件表达式 1 和 2 都不为 true 时执行
}
```

示例代码如下：

```
//unit5/if 语句/3.if - else - if 多分支.go
package main

import "fmt"

func main() {
    score := 80
    if score >= 90 {
        fmt.Println("优秀")
    } else if score >= 80 {
        fmt.Println("良好")
    } else {
        fmt.Println("一般")
    }
}
```

if-else-if 多分支的流程图如图 5-3 所示。

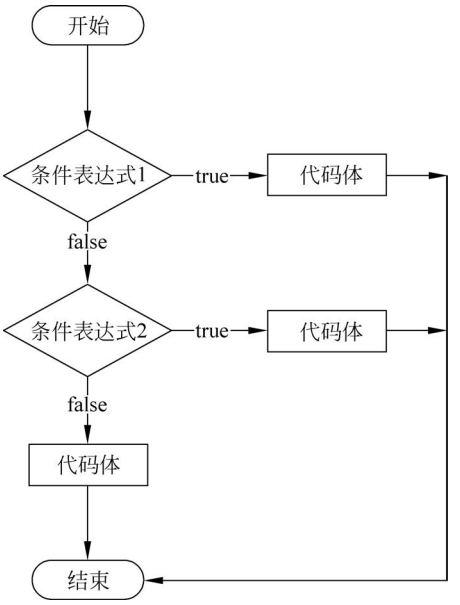


图 5-3　if-else-if 多分支的流程图

5.1.4　if 嵌套

if 嵌套表示可以在 if 语句块中继续添加 if 判断，语法如下：

```
if 条件表达式 1 {
    if 条件表达式 2 {
        //当条件表达式 2 为 true 时执行
    } else {
        //当条件表达式 2 为 false 时执行
    }
} else {
    //当条件表达式 1 为 false 时执行
}
```

示例代码如下：

```
//unit5/if 语句/4.if 嵌套.go
package main

import "fmt"

func main() {
    score := 90
    if score >= 90 {
        if score >= 95 {
            fmt.Println("非常优秀")
        } else {
            fmt.Println("优秀")
        }
    } else {
        fmt.Println("一般")
    }
}
```

if 嵌套也可以无限制地嵌套，但是在实际开发中同样不建议嵌套太多层。

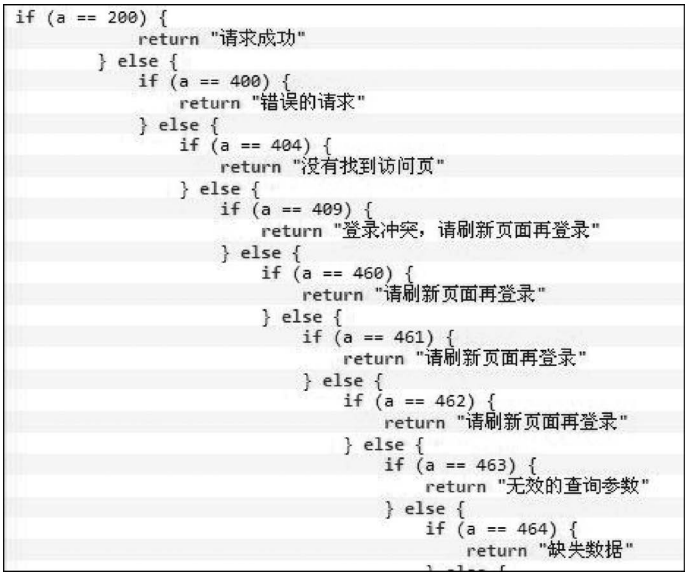
5.1.5 知识扩展——卫语句

万能的 if-else 语句，仿佛任何需求都能够满足，也是写代码常用的一种语句，有很多复杂的业务嵌套了无数个 if-else 语句，例如图 5-4 中让人头疼的 if 语句，应该如何优化呢？

卫语句就是把复杂的条件表达式拆分成多个条件表达式，减少嵌套。将嵌套了好几层的 if-else 语句，转换为多个 if 语句，实现它的逻辑，这多条的 if 语句就是卫语句。

卫语句将某些关键条件优先判断，简化程序流程走向。卫语句往往用于对 if 条件嵌套代码的优化。

在《阿里巴巴 Java 开发手册》中强制规定：超过 3 层的 if-else 的逻辑判断代码可以使用卫语句、策略模式、状态模式等来实现，其中卫语句即代码逻辑先考虑失败、异常、中断、退出等直接返回的情况，以方法多个出口的方式，解决代码中判断分支嵌套的问题，这是逆向思维的体现。



```
if (a == 200) {
            return "请求成功"
        } else {
            if (a == 400) {
                return "错误的请求"
            } else {
                if (a == 404) {
                    return "没有找到访问页"
                } else {
                    if (a == 409) {
                        return "登录冲突，请刷新页面再登录"
                    } else {
                        if (a == 460) {
                            return "请刷新页面再登录"
                        } else {
                            if (a == 461) {
                                return "请刷新页面再登录"
                            } else {
                                if (a == 462) {
                                    return "请刷新页面再登录"
                                } else {
                                    if (a == 463) {
                                        return "无效的查询参数"
                                    } else {
                                        if (a == 464) {
                                            return "缺失数据"
```

图 5-4　多层 if 嵌套语句

例如分数得分示例，不使用卫语句如下：

```
//unit5/if 语句/5.卫语句.go
package main

import "fmt"

func main() {
    var scope int

    fmt.Printf("输入你的分数:")
    _, err := fmt.Scanf("%d", &scope)
    if err != nil {
        fmt.Println("输入的分数错误")
        return
    }

    if scope >= 80 {
        if scope >= 90 {
            fmt.Println("A")
        }else {
            fmt.Println("B")
        }
    }else {
        if scope >= 60 {
            fmt.Println("C")
        }else {
```

```
            fmt.Println("D")
        }
    }

}
```

使用卫语句如下：

```
//unit5/if 语句/5.卫语句.go
package main

import "fmt"

func main() {
    var scope int

    fmt.Printf("输入你的分数:")
    _, err := fmt.Scanf("%d", &scope)
    if err != nil {
        fmt.Println("输入的分数错误")
        return
    }

    if scope >= 90 {
        fmt.Println("A")
        return
    }
    if scope >= 80 {
        fmt.Println("B")
        return
    }
    if scope >= 60 {
        fmt.Println("C")
        return
    }
    fmt.Println("D")
}
```

5.2 switch 语句

switch 语句是根据变量的值执行不同的 case，在执行的过程中会从第 1 个 case 开始判断，直到碰到一个符合条件的 case 为止，然后执行该 case 中的语句，语法如下：

```
switch 变量 {
    case 变量 1:
        //当变量和变量 1 相等时执行
    case 变量 2:
```

```
        //当变量和变量 2 相等时执行
    default:
        //当没有符合的 case 时执行
}
```

示例代码如下：

```
//unit5/switch 语句/6.switch 语句.go
package main

import "fmt"

func main() {
    var day int

    fmt.Printf("几天周几:")
    _, err := fmt.Scanf("%d", &day)
    if err != nil {
        fmt.Println("输入的数字错误")
        return
    }

    switch day {
    case 1,2,3,4,5:
        fmt.Println("工作日")
    case 6,7:
        fmt.Println("周末")
    default:
        fmt.Println("错误的时间")
    }

}
```

switch 语句的主要特点如下：

(1) switch 和 if 语句一样，switch 后面可以带一个可选的简单的初始化语句。

(2) switch 后面的表达式也是可选的，如果没有表达式，则 case 子句是一个布尔表达式，而不是一个值，此时就相当于多重 if-else 语句。

(3) switch 条件表达式的值不像 C 语言那样必须限制为整数，可以是任意支持相等比较运算的类型变量。

(4) 通过 fallthough 语句来强制执行下一个 case 子句(不再判断下一个 case 子句的条件是否满足)。

(5) switch 支持 default 语句，当所有的 case 分支都不符合时，执行 default 语句，并且 default 语句可以放到任意位置，并不影响 switch 的判断逻辑。

(6) switch 和.(type)结合可以进行类型的查询。

在默认情况下，匹配到一个 case 之后整个 switch 语句就会结束，其他的 case 条件就不会参与下次匹配，如果需要执行后面的 case，则可以使用 fallthrough 关键字，使用 fallthrough

之后 switch 便不会退出,代码如下:

```
//unit5/switch 语句/7.fallthrough.go
package main

import "fmt"

func main() {
    a := 11
    //没有 fallthrough
    switch {
    case a > 5:
        fmt.Println("a 大于 5")
    case a > 10:
        fmt.Println("a 大于 10")
    }
    fmt.Println("=======")
    //有 fallthrough
    switch {
    case a > 5:
        fmt.Println("a 大于 5")
        fallthrough
    case a > 10:
        fmt.Println("a 大于 10")
    }
}
```


7min

5.3 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。

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 4 种模式,分别是标准 for 循环模式、while 模式、do-while 模式、for range 模式。


5min

5.3.1 标准 for 循环

for 循环的语法如下:

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

其中,init 表示初始化条件,condition 表示条件,post 表示赋值表达式。

例如,以从 1 加到 5 为例,使用标准 for 循环实现,代码如下:

```
//unit5/for 循环/标准 for 循环.go
package main
```

```
import "fmt"

func main() {
    var sum = 0                    //定义一个用于求和的变量
    for i := 0; i <= 5; i++{
        sum += i
    }
    fmt.Println(sum)               //15
}
```

以下是详细执行过程分析：

(1) 定义一个 sum 变量，用于把每次循环的结果相加。

(2) 定义一个变量 i 并赋值为 0，进入循环将 i 的值追加到 sum 变量中，然后 i 自增 1。

(3) 此时 i 等于 1，判断 i 的值是否小于或等于 5，如果条件成立，则继续进入循环，循环往复直到 i 等于 6；如果条件不成立，则结束循环，输出 sum 变量。

5.3.2　while 模式的 for 循环

while 模式下的 for 循环，只需一个条件，如果条件成立，则执行循环体；如果条件不成立，则结束循环，语法如下：

```
for condition { }
```

以 1 加到 100 为例，使用 while 模式的 for 循环，代码如下：

```
//unit5/for 循环/while 型 for 循环.go
package main

import "fmt"

func main() {
    var sum = 0                    //定义一个用于求和的变量
    var i = 0
    for i <= 100 {
        sum += i
        i++
    }
    fmt.Println(sum)               //5050
}
```

5.3.3　do-while 模式的 for 循环

do-while 模式下的 for 循环没有条件，结束循环的条件在循环体中编写，语法如下：

```
for { }
```

以 1 加到 100 为例，使用 do-while 模式的 for 循环，代码如下：

```
//unit5/for 循环/do while 型 for 循环.go
package main

import "fmt"

func main() {
    var sum = 0          //定义一个用于求和的变量
    var i = 0
    for {
        if i > 100 {
            //当 i 大于 100 时结束循环
            break
        }
        sum += i
        i++
    }
    fmt.Println(sum)     //5050
}
```

当循环体内没有条件结束循环时程序就会不停地执行循环体中的代码,在计算机中称为“死循环”。

下例是不断打印当前时间的死循环,代码如下:

```
//unit5/for 循环/死循环.go
package main

import (
    "fmt"
    "time"
)

func main() {
    for {
        time.Sleep(1 * time.Second)                              //程序等待 1s
        fmt.Println(time.Now().Format("2006-01-02 15:04:05"))    //年月日时分秒格式的时间
    }
}
```

5.3.4 for range 模式的 for 循环

用于遍历 slice(切片)、map、数组、字符串等数据类型,格式如下:

```
for key, value := range iteration {}
```

当循环 slice、数组、字符串等数据类型时,key 为当前元素的索引,代码如下:

```
var slice = []string{"A", "B", "C"}
for index, item := range slice {
```

```
    fmt.Printf("元素值 %s, 索引:%d\n", item, index)
}
```

当循环 map 类型时,代码如下:

```
var maps = map[string]any{"name": "枫枫", "age": 25}
for key, value := range maps {
    fmt.Printf("元素值 %v, key:%s\n", value, key)
}
```

5.3.5　break 语句

在 Go 语言中,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。

例如循环 10 个数字,但是只需前 3 个数,代码如下:

```
//unit5/for 循环/break.go
package main

import "fmt"

func main() {
    for i := 1; i < 11; i++{
        fmt.Println(i)
        if i == 3 {
            break
        }
    }
}
```

5.3.6　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条循环语句。

例如循环 10 个数字,只需其中的偶数,代码如下:

```
//unit5/for 循环/continue.go
package main

import "fmt"

func main() {
    for i := 1; i < 11; i++{
        //任何数对 2 取余
        //如果结果为 0,则表示是 2 的倍数
        if i%2 != 0 {
```

```
            continue
        }
        fmt.Println(i)
    }
}
```

5.3.7 多重循环

Go 语言允许用户在循环内使用循环,需要注意的是,break 只能跳出当前 for 循环,不能跳出多个 for 循环。

例如打印九九乘法表,代码如下:

```
//unit5/for 循环/九九乘法表.go
package main

import "fmt"

func main() {
    for m := 1; m < 10; m++{
        for n := 1; n <= m; n++{
            fmt.Printf("%dx%d=%d ", n, m, m*n)
        }
        fmt.Println()
    }
}
```

输出结果如图 5-5 所示。

```
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
```

图 5-5 九九乘法表