第3章 程序流程控制

3.1 代码块与作用域


视频讲解

3.1.1 代码块

从宏观上来说，一个项目程序由多个包构成，每个包由若干个函数构成，函数又由若干个代码块形成，当然也可以把整个函数看成一个代码块。代码块是一个相对独立的执行单元，通常用大括号“{}”括起来(或者由两个 case 关键字分隔)。例如，下述函数体是一个代码块：

```
func sum(a int,b int)int{
     return a + b
}
```

以下循环体也属于一个代码块：

```
var sum = 0
for i := 0;i < 5;i++{
     sum += i
     fmt.Println("sum = ",sum)
}
```

以下条件分支的大括号内也属于一个代码块：

```
if a > b{
    c = a
    a = b
    b = c
}
```

在多分支 switch 及 select 中的两个 case 之间的语句也构成一个代码块，但不含大括号。上述代码块的特点是定义在大括号内，大括号内的所有语句成为一个整体。当然，大括号内的语句块还可以嵌套，包含子语句块，子语句块又成为一个独立的语句块，嵌套的数量不受限制。

可见，代码块就是相对独立的一段程序，有一定的边界。

3.1.2 作用域

一般而言，代码块内部定义的变量具有 private 属性，只在这个代码块内可用，称为在代码块内可见，这种可见性称为变量的作用域。Go 语言规定任何对象只有在其作用域内才可

被访问，或者说它们的名字才可以被使用。

在代码块内定义的变量其作用域为整个代码块内；函数体内定义的变量其作用域为整个函数体内；而主函数体外定义的变量其作用域为整个包，称为全局变量，在整个包内可见。根据 Go 语言的命名规则，如果全局变量或函数名字的首字母为大写，那么这样的变量或函数具有外部可见性，其作用域超出本包，可为其他包所引用。以下例子演示各种变量在不同的作用域内的使用情况：

```
package main
import (
    "fmt"
)
var StuId = 0                   //  作用域为包内外,即包外可见
var a = 10                      //  作用域为包内
func main() {
    a++                         //  取全局变量的值
    StuId++                     //  取全局变量的值
    fmt.Println("函数体内 StuId = ", StuId)
    var x = 1
    for i := 0; i < 3; i++{
        a++                     //  取全局变量的值
        x++                     //  取局部变量的值
        StuId++                 //  取全局变量的值
        fmt.Println("循环体内 StuId = ", StuId)
        fmt.Println("循环体内全局 a = ", a, "局部 x = ", x, "块内 i = ", i)
        var a, x = 10, 20       // 块内变量覆盖块外变量
        fmt.Println("内部声明 a = ", a, "x = ", x)
        x = Add(a, x)
        fmt.Println("求和后 a = ", a, "x = ", x, "i = ", i)
        a++                     //  取内部变量的值
        x++                     //  取内部变量的值
        fmt.Println("循环体内求和自增后 a = ", a, "局部 x = ", x, "块内 i = ", i)
    }
    a++                         //  取全局变量的值
    x++                         //  取局部变量的值
    StuId++                     //  取全局变量的值
    fmt.Println("循环体外 StuId = ", StuId)
    fmt.Println("循环体外全局 a = ", a, "局部 x = ", x)
//  fmt.Println("循环体外 i = ",i)   语法错误,块内定义的变量块外不可见
}
func Add(x int, y int) int {
    return x + y
}
```

输出结果：

```
函数体内 StuId = 1
循环体内 StuId = 2
循环体内全局 a = 12  局部 x = 2  块内 i = 0
内部声明 a = 10 x = 20
求和后 a = 10 x = 30 i = 0
循环体内求和自增后 a = 11  局部 x = 31  块内 i = 0
```

```
循环体内 StuId = 3
循环体内全局 a = 13  局部 x = 3  块内 i = 1
内部声明 a = 10  x = 20
求和后 a = 10 x = 30 i = 1
循环体内求和自增后 a = 11  局部 x = 31  块内 i = 1
循环体内 StuId = 4
循环体内全局 a = 14  局部 x = 4  块内 i = 2
内部声明 a = 10 x = 20
求和后 a = 10 x = 30 i = 2
循环体内求和自增后 a = 11  局部 x = 31  块内 i = 2
循环体外 StuId = 5
循环体外全局 a = 15  局部 x = 5
```

上述例子中，声明了一个公共变量 StuId，为全局变量，包内外均可见，即可以被别的包引用。在函数体内或循环体内均可以修改全局变量的值。

此外，还声明了一个全局变量 a、局部变量 x 以及三个块内变量 i，a，x。i 为块内定义的循环计数变量，在关键字 for 后面用短变量声明操作符“:=”声明并赋值，称为 for 的子句。for 子句仅允许声明一个变量，其作用域仅在循环体内，循环结束即被注销。在循环体内声明的 a，x 为内部变量，与外部变量同名，是合法的，本质上是不同的变量，在内存中处于不同的位置。内部变量的值会屏蔽外部同名变量的值，但不会真实覆盖。循环程序运行时使用的是内部变量的值，直至当前循环结束。整个循环结束后，变量 a，x 又恢复了被覆盖的值。

函数 Add 首字母大写，表明为一个全局性函数，可以被别的包引用。

循环和函数等知识将在后面相关章节中叙述，有兴趣的读者可以提前查阅。

3.1.3 变量的可见性

变量在其作用域内是可见的，一个可见的变量可以被访问及赋值。3.1.2 节的程序例子看起来有点复杂，不细心分析比较，很难看出其表达的意思。这里就变量的可见性总结以下几点：

(1) 全局变量 StuId 在函数体内外，及函数体内嵌的程序块内均可见；

(2) 函数体内局部变量在函数体及其内嵌的程序块内均可见；

(3) 函数体内嵌程序块定义的变量仅在该块内可见；

(4) 程序块定义的变量可与外部变量同名、同类型或不同类型；

(5) 同名的内部变量会屏蔽外部变量，但不会改变外部变量的值；

(6) 在循环体内定义的变量仅在当前循环内起作用，对下一轮次循环无效；

(7) 循环体一次循环就是一个作用域，第二次循环为新的作用域。

变量必须可见才可以操作，使用不可见的变量会引起编译错误。从安全的角度来说，应该尽量缩小变量的可见性，仅为需要的地方提供可见性即可。

因此，谨慎定义全局变量，因为为其可见性覆盖整个包的所有函数，任何一个函数内对该变量的不正确引用都可能带来灾难性后果。


视频讲解

3.2 if 语 句

if 语句称为条件语句，通常用于选择“yes”和“no”的逻辑判断场合。if 语句是 Go 语言基本的条件分支语句，使用的时候必须带有逻辑表达式。

3.2.1 基本语法

if 语句为条件分支语句，用于条件判断后至少有两种可能的程序执行流的情况。在实际编程中有时候需要面对各种不同的情况，针对不同的情况采取不同的措施，这时就要用到条件分支语句。最常用到的条件分支语句是 if…else 语句，该语句有多种语法格式，其典型的语法格式如下：

```
if 条件表达式{
    语句块 1
  }else{
    语句块 2
  }
```

条件表达式为任意的逻辑表达式，其计算结果必须为布尔值。左大括号必须紧接着条件表达式，不得另起一行，否则将导致编译错误。条件表达式可以用小括号“()”括起来，也可以不用。如果条件表达式的值为真 true，则执行语句块 1，否则执行语句块 2。

关键字 else 必须紧接在语句块 1 的右大括号后面与其同行，同时语句块 2 的左大括号必须与 else 同行，不允许另起一行。

语句块 1 和语句块 2 可以有多条语句，也可以没有语句，但是，大括号是必须有的，例如：

```
if a > b{
   fmt.Println("a > b")
}else{}
```

或

```
  if a > b{

  }else{
      fmt.Println("a < b")
}
```

都是合法的。

但是，上述情况中如果语句块 2 为空，则 else 可以省略。

3.2.2 省略 else

如果只有两种情况，也就是说只有二分支，而第二个分支没有需要执行的语句，此时可以忽略 else，例如：

```
if i < 100{
     i++
}
```

这时就成了简单的 if 语句，在实际编程中很常见。实际上，有多种情况需要判断也可以使用简单的 if 语句，例如学生成绩等级划分就可以使用简单的 if 语句。假如学生成绩为 x，等级划分程序如下所示：

```
package main
import "fmt"
func main() {
    var x float64
    fmt.Println("请输入学生的成绩: ")
    fmt.Scanf("%f", &x)
    if x >= 90 {
        fmt.Println("学生成绩等级为: 优秀")
    }
    if x >= 80 && x < 90 {
        fmt.Println("学生成绩等级为: 良好")
    }
    if x >= 70 && x < 80 {
        fmt.Println("学生成绩等级为: 中等")
    }
    if x >= 60 && x < 70 {
        fmt.Println("学生成绩等级为: 及格")
    }
    if x < 60 {
        fmt.Println("学生成绩等级为: 不及格")
    }
}
```

运行结果：

```
请输入学生的成绩:
45
学生成绩等级为: 不及格

//再次运行:
请输入学生的成绩:
85
学生成绩等级为: 良好

//再次运行:
请输入学生的成绩:
95
学生成绩等级为: 优秀
```

上述程序中，每一个 if 语句只考虑自己的条件，满足就执行语句块，不满足就跳过。多种条件分别用不同的 if 语句判断，这样使用简单的 if 语句可以使程序的逻辑性变得清晰，让人易读易懂。

3.2.3 带子句的 if

Go 语言允许在 if 关键字之后，条件表达式之前插入一个简单语句，称为 if 语句的子句，与条件表达式之间用分号隔开。if 子句是可选的，一旦有，必须为简单语句(区别于由多条语句构成的复合语句)。带子句的 if 语句其语法格式如下：

```
if  可选子句 ; 条件表达式{
     语句块
}
```

以下子句写法都是正确的：

```
if i := 0; x < 5{
    i++
}
```

或

```
var x,y int
x = 10
if y = 10 + x; x < 5{
    x++
}
```

显然，上述子句都是简单语句，包括表达式、通道操作、增减值语句、赋值语句或短变量声明等语句均可。

上述变量 i 定义在 if 的子句里面，属于 if 语句块的局部变量，其可见性仅在当前 if 语句块及其块内嵌套的其他块内可见，结束 if 语句块后 i 不可用。如果有多层 if 嵌套，i 的作用域包括所有 else if 及 else 分支。但变量 x，y 在 if 语句块外定义，结束 if 语句块后仍然有效。

可选子句的执行可为其后续的逻辑表达式提前准备条件，如变量初始化赋值等。当然，可选子句也可以与条件表达式无关，仅仅用来声明及初始化一个变量，而且仅能声明一个变量，为后续程序块做准备。可选子句也可以是函数调用，如下所示：

```
package main
import "fmt"
func main() {
    UU := 1
    vv := 2
        if i := add(UU, vv); i < 8 {
        fmt.Println("UU = ", UU)
        var UU int = 10
        UU++
        fmt.Println("UU = ", UU, "i = ", i)
    }
    fmt.Println("UU = ", UU, "vv = ", vv)
}
func add(x int, y int) int {
    return x + y
}
```

运行结果：

```
UU = 1
UU = 11 i = 3
UU = 1 vv = 2
```

程序分析：

上述程序的 if 子句是一个函数调用，先执行函数计算，获得其计算结果的返回值赋给 i，然后 i 作为条件表达式的参数，计算条件表达式的值。如果该值为 true，则执行 if 语句块；如果该值为 false，则跳过 if 语句块。add 为一个独立的加法子函数，可被主函数调用。

上述程序中主函数 main 中有一个变量 UU,在 if 语句块中又声明了一个变量 UU。主函数内的 UU 在 if 语句块内可见,因此,第一个打印语句的输出是 main 函数内 UU 的值。第二个打印语句打印的 UU 是 if 语句块内自己定义的 UU,它会覆盖外部 UU 的值,因此,打印结果是内部 UU 的值,同时,i 是调用函数的返回值。第三个打印语句已经退出了 if 语句块,这时打印的 UU 又恢复了外部 UU 的值。显然,内部 UU 只是遮挡了外部 UU,并不是真正地改变了外部 UU 的值。

3.2.4 if 语句的嵌套

if…else 语句还允许嵌套,在其语句块 1 和语句块 2 内部还可以有 if…else 语句,形成语句块 3 和语句块 4;语句块 3 和语句块 4 还可以嵌套 if…else 形成语句块 5 和语句块 6。这样一层套一层,层数没有限制,只需要保持大括号配对即可。实际编程中,最好不要超过三层,例如:

```
package main
import "fmt"
func main() {
    var a, b, c, d, e, f = 11, 2, 1, 4, 15, 6
    if a > b {
        if c < d {
            if e > f {
            fmt.Println("a > b and c < d and e > f")
        } else {
            fmt.Println("a > b and c < d and e < f")
        }
        } else {
        fmt.Println("a > b and c > d")
        }
    } else {
    fmt.Println("a < b ")
    }
}
```

运行结果:

```
a > b and c < d and e > f
```

注意:if 嵌套会占用大量系统资源,极大地影响系统性能,应尽量避免使用。

上述主要采用两分支的情况进行嵌套判断,很容易导致逻辑复杂混乱,程序也难以看懂。如果分支不多,可以采用 3.2.5 节介绍的 if…else if 语句,逻辑会更清晰一些。

3.2.5 if…else if 语句

对于多分支的情况,如果采用 if…else 嵌套,则很容易导致逻辑复杂、混乱且易出错,程序难懂,这时可以考虑用 if…else if 语句替代。该语句格式是专门用来处理 if 语句多分支的情况。例如,上述为学生成绩分档的示例,可以用 if…else if 格式改写如下:

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

```
func main() {
    var x float64
    fmt.Println("请输入学生的成绩: ")
    fmt.Scanf("%f%", &x)
    if x >= 90 {
        fmt.Println("学生成绩等级为: 优秀")
    } else if x >= 80 {
        fmt.Println("学生成绩等级为: 良好")
    } else if x >= 70 {
        fmt.Println("学生成绩等级为: 中等")
    } else if x >= 60 {
        fmt.Println("学生成绩等级为: 及格")
    } else {
        fmt.Println("学生成绩等级为: 不及格")
    }
}
```

运行结果：

```
请输入学生的成绩:
56
学生成绩等级为: 不及格

// 再次运行:
请输入学生的成绩:
78
学生成绩等级为: 中等

// 再次运行:
请输入学生的成绩:
98
学生成绩等级为: 优秀
```

程序分析：

通过上述 if…else if 改写后，条件表达式变得更加简洁，程序更容易为人读懂，逻辑思路也清晰。因此，对于多分支的 if 语句建议采用这种处理方式。但是，这种方式也有不足，当分支过多的时候，大括号的配对比较麻烦，容易出错。其实，对于多分支条件表达式有更好的语句，这就是 3.3 节要介绍的 switch…case 语句。

视频讲解

3.3 switch 语句

switch 语句称为开关语句，是 Go 语言专门为多分支选择执行而设计的语句。switch 作为多分支开关，后接一个开关表达式，而用 case 表示多分支出口。每个 case 后面也要带同类型的表达式，case 后边还可以带多个表达式。case 表达式的计算顺序是从上到下，从左到右，计算结果会与 switch 表达式的计算结果相比较，结果相同的 case 分支将得到执行。如果所有 case 表达式的值都与 switch 表达式的值不同，则执行 default case 分支；如果没有该分支，则直接结束 switch 语句。

Go 语言提供的 switch 分为两种类型：表达式 switch(expression switch)和类型 switch

(type switch)。表达式 switch 就是常用的多分支选择语句,类型 switch 主要用于类型判断,使用类型断言表达式,根据不同的数据类型执行不同的动作。

3.3.1 表达式 switch

表达式 switch 是常用的格式,适用于大多数的多分支选择情况,其语法格式如下:

```
switch 可选子句; 可选表达式 {
case 表达式列表 1: 语句块 1
case 表达式列表 2: 语句块 2
...
case 表达式列表 n: 语句块 n
default: 语句块 d
}
```

可选子句:为 Go 语言的简单语句,与 if 子句相同。如果有可选子句出现,则必须用分号与后面的可选表达式隔开,即使后面的可选表达式没有,分号也不能省略。如果变量是在可选子句里用短变量声明的,则其作用域为从声明处到 switch 结尾,包括所有 case 分支和 default 分支。如果各个 case 包含了嵌套的子语句块,则在该子语句块也可见。

可选表达式:可以是算术表达式,也可以是逻辑表达式,甚至可以没有。如果没有可选表达式,则编译系统会默认为逻辑表达式,其值为 true,这种情况下要求所有 case 表达式为逻辑表达式。

case:后面必须有表达式列表,其类型必须与 switch 表达式匹配,表达式数量不限,必须用逗号隔开。多个表达式对应同一个分支,执行相同动作。

default:为缺省分支,不是必须的,如果有的话,位置不限,可放置在任何地方。若没有任何 case 匹配,流程控制将执行 default 分支,执行结束退出 switch 语句。如果没有任何 case 匹配,也没有 default 分支,则直接退出 switch 语句。

语句块:可以含有 0 到多条语句,甚至还可以嵌套其他的语句块,但是不用大括号括起来,两个 case 之间就等同于大括号。语句块内的语句顺序执行,执行完毕自动退出 switch 语句。

下面,仍然以上述成绩等级划分为例,看看用 switch 多分支语句,如何改写该程序。

```
package main
import "fmt"
func main() {
    var x float64
    fmt.Println("请输入学生的成绩: ")
    fmt.Scanf("%f%", &x)
    switch int(x / 10) {
    case 9, 10:
        fmt.Println("学生成绩等级为: 优秀")
        var x, y = 10, 20 // x,y 为局部变量,会屏蔽外部变量 x
        x = x + y
        fmt.Println("x + y = ", x)
    case 8:
        fmt.Println("学生成绩等级为: 良好")
```

```
        case 7:
            fmt.Println("学生成绩等级为: 中等")
        case 6:
            fmt.Println("学生成绩等级为: 及格")
        default:
            fmt.Println("学生成绩等级为: 不及格")
        }
        fmt.Println("x + y = ", x) // y 变量不可用,x 为 switch 外部变量
    }
```

运行结果：

```
请输入学生的成绩:
95
学生成绩等级为: 优秀
x + y = 30
x + y = 95
```

程序分析：

从上面的例子可以看出，case 表达式的值类型必须与 switch 表达式的值类型相匹配。一个 case 后可以有多个值，每个值匹配成功后均执行相同的语句块。语句块为一个独立的作用域，其内部定义的变量（如 x，y）只在该块内有效，退出 switch 后该变量不可用。

如果忽略 switch 表达式，则其默认为 true，其后的所有 case 表达式必须为逻辑表达式，其值为布尔型，这样才能匹配 switch。可将上述成绩分级程序改写如下：

```
package main
import "fmt"
func main() {
    var x float64
    fmt.Println("请输入学生的成绩: ")
    fmt.Scanf("%f%", &x)
    var y = int(x / 10)
    switch {
    case y >= 9:
        fmt.Println("学生成绩等级为: 优秀")
    case y >= 8:
        fmt.Println("学生成绩等级为: 良好")
    case y >= 7:
        fmt.Println("学生成绩等级为: 中等")
    case y >= 6:
        fmt.Println("学生成绩等级为: 及格")
    default:
        fmt.Println("学生成绩等级为: 不及格")
    }
}
```

运行结果：

```
请输入学生的成绩:
52
学生成绩等级为: 不及格
```

可见，两种形式的 switch 本质上没什么不同，完全可以互换使用。第一种形式是经典形式，与其他编程语言一样。第二种形式逻辑更清晰，程序更容易读懂，感觉更符合人类的思维习惯，实际使用时可以依据个人习惯来选择。

3.3.2 类型 switch

类型 switch 本质上与表达式 switch 差不多，都是 switch 后面带表达式，根据表达式的计算结果来匹配每个 case。不同的地方是，类型 switch 的表达式为类型断言表达式，用关键字“type”代替实际类型，其表达式的计算结果值是变量的类型(Type)字面量。而其每个 case 表达式是各种数据类型的字面量，为常数。语法格式如下：

```
switch 可选子句; 类型开关守护 {
case 类型列表 1: 语句块 1
case 类型列表 2: 语句块 2
...
case 类型列表 n: 语句块 n
default: 语句块 d
}
```

可选子句与表达式 switch 语句及 if 语句中可选子句一样，为简单语句或短变量声明等。

类型开关守护(type switch guard)是一个结果为类型的表达式 x.(type)，如果其采用短变量声明的方式赋值给一个变量，则变量的值即为该表达式的值。该变量的类型将与所有 case 子句的类型表达式匹配，匹配成功将执行该分支的语句块。default 分支也不是必须的，有的话其位置是任意的。

参看以下的例子：

```
package main
import "fmt"
func main() {
    var x interface{}
    var (
    ch  rune      = 'a'
    ct  int       = 12
    cf  float32   = 12.34
    cff float64   = 34.56
    cb  bool      = true
    cs  string    = "asdfgh"
    i             = 0
    )
    fmt.Println("请输入数字 1 -- 6!")
    fmt.Scanln(&i)
    switch i {
    case 1:
        x = ch
    case 2:
        x = ct
    case 3:
```

```
            x = cf
        case 4:
            x = cff
        case 5:
            x = cb
        case 6:
            x = cs
        default:
            x = i
        }
        switch z := x.(type) {
        case rune:
            fmt.Printf("你输入字符 a(ASCII 97)! %T, %v\n", z, z)
        case int,int8,int16,int64:
            fmt.Printf("你输入的是整型数字! %T, %v\n", z, z)
        case float32:
            fmt.Printf("你输入的是浮点数! %T, %v\n", z, z)
        case float64:
            fmt.Printf("你输入的是浮点数! %T, %v\n", z, z)
        case string:
            fmt.Printf("你输入的是字符串! %T, %v\n", z, z)
        case bool:
            fmt.Printf("你输入的是布尔型! %T, %v\n", z, z)
        default:
            fmt.Printf("未知类型! %T, %v\n", z, z)
        }
    }
```

运行结果：

```
请输入数字 1 -- 6!
6
你输入的是字符串!string,asdfgh
```

上述例子中，首先定义了一个空接口类型的变量 x，它可以接受任意类型的赋值，类型守护开关就是常用接口类型。然后声明了 6 种数据类型（rune，int，float32，float64，bool，string）的 6 个变量，rune 是 int32 的别名。要求用户从键盘上读入一个 1～6 的数字，从而确定让哪一个变量赋给 x，然后执行类型断言表达式 x.(type)获得接口变量的类型及值，由 switch…case 来判断输出该类型的名字。

实际工作中，很少用到 type switch，只有牵涉外部数据源需要判断数据类型的时候才用到 type switch。

3.3.3 switch 的嵌套

switch 语句和 if 语句一样，允许嵌套使用。在每个 case 后面的语句块中如果又包含了多种可能条件需要判断执行，就可以继续使用 switch…case 进行多分支选择，从而形成了 switch 的嵌套。例如：

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

```
func main() {
    var (
    cff float64
    i   int
    ch  rune
    )
    fmt.Println("请输入参数: ")
    fmt.Scanln(&cff)
    i = int(cff / 10)
    ch = 'c'
    switch i {
    case 0:
        switch ch {
        case 'a':
            fmt.Printf("i = 0,ch = a\n")
        case 'b':
            fmt.Printf("i = 0,ch = b\n")
        case 'c':
            fmt.Printf("i = 0,ch = c\n")
        default:
            fmt.Printf("i = 0,ch = _\n")
        }
    case 1:
        switch ch {
        case 'd':
            fmt.Printf("i = 1,ch = d\n")
        case 'e':
            fmt.Printf("i = 1,ch = e\n")
        case 'f':
            fmt.Printf("i = 1,ch = f\n")
        default:
            fmt.Printf("i = 1,ch = _\n")
        }
    case 2:
        switch ch {
        case 'g':
            fmt.Printf("i = 2,ch = g\n")
        case 'h':
            fmt.Printf("i = 2,ch = h\n")
        case 'i':
            fmt.Printf("i = 2,ch = i\n")
        default:
            fmt.Printf("i = 2,ch = _\n")
        }
    default:
        fmt.Printf("输入超范围!\n")
    }
}
```

运行结果：

```
请输入参数 i:
21
i = 2,ch = _
```

程序分析：

上述外层 switch 根据变量 i 的取值来选择分支，而内层 switch 根据字符变量 ch 的取值来选择分支。switch 嵌套层数是没有限制的，建议不要嵌套太深，以免造成阅读、调试困难。

3.3.4 break 语句

与 C 语言不同，Go 语言不需要显式地在每一个 case 分支最后加一个 break 语句。因为系统默认在每一个 case 分支最后隐含一个 break。如果要在 case 分支语句块中提前退出程序的执行，则必须人为地插入一个 break 语句，退出当前的 switch 语句。

例如：

```
var x,a,b,c = 0,1,2,3
switch x{
case 0:
      if a == 0{
         break
      }
      c = b/a
case 1:
      if b == 0{
         break
      }
      c = a/b
}
```

在任何一个 case 语句块中，如果不需要程序继续执行，就可以插入一个 break，退出当前 switch 语句。需要注意的是，break 仅中断当前所在的 switch 语句。如果 break 是在子语句块的 switch 语句里面，则仅退出子语句块所在的 switch，其外层 switch 的语句仍在执行。

break 语句也可以一次退出多层 switch，只需在 break 后面加一个标签即可。要退出哪一层 switch，就跳转到该 switch 前面的标签处。标签是一个带有冒号结束的标识符，只可以放置在 for、switch 和 select 前面，如下所示：

```
package main
import "fmt"
func main() {
    var (
    x       = 0
    a, b, c = 1, 2, 3.14
    )
    NEXT:
    switch x {
```

```
    case 0:
        if a > 0 {
       switch y := 0; y {
       case 0:
           c = float64(b) / float64(a)
           fmt.Println("y = 0")
           break               // No.1
       case 1:
           c = float64(5 * b) / float64(a)
           fmt.Println("y = 1")
           break NEXT          // No.2
       }
        }
        a++
    case 1:
        if b == 0 {
            break              // No.3
        }
        c = float64(a) / float64(b)
        fmt.Printf("c = %f\n", c)
    default:
        break                  // No.4
    }
    fmt.Printf("a = %d,c = %f\n", a, c)
}
```

运行结果：(我们改变 x,y 的值观察不同的运行结果)

```
x = 0,y = 0 的输出: y = 0
                a = 2,c = 2.000000

x = 0,y = 1 的输出: y = 1
                a = 1,c = 10.000000

x = 1,y = 1 的输出: c = 0.500000
                a = 1,c = 0.500000
```

程序分析：

上述程序中，在最外层 switch 前面加了一个标签 NEXT，在内层 switch 中有两个 break：第一个 break 不带标签，执行后将退出内层子 switch，外层 switch 仍在运行；第二个 break 带了标签 NEXT，一旦执行该 break 语句，程序流程将直接跳转到 NEXT 标签处。由于该标签在外层 switch 的前面，意味着退出该层 switch，执行该 switch 语句下面一条语句。

第三条和第四条 break 语句处于外层 switch，如果执行的话直接退出外层 switch。

3.3.5 fallthrough 语句

switch 语句中在每个 case 语句块最后隐含了一条 break 语句，语句块执行结束后直接退出 switch。而不是像 C 语言那样，会穿越下一个 case 继续执行后续分支的语句块。这对大多数实际编程应用来说是有利的，减少了程序员敲键盘的数量，减轻了工作量。

但是，在某些情况下，还是有需要穿越 case，执行后续的语句。Go 语言专门提供了一条

语句：fallthrough，用来穿越 case，去执行下一个 case 分支的语句块。一条 fallthrough 语句，仅穿越一个 case，如果需要连续穿越多个 case，则需要多条 fallthrough 语句，在每个需要被穿越的 case 前面放置一条 fallthrough。fallthrough 语句穿越 case 后直接执行该 case 后第一条语句。如果没有语句，即该 case 语句块为空，则直接退出 switch。

需要注意的是，fallthrough 仅适用于表达式 switch，对类型 switch 及 select 语句无效，且非法。

举个例子，假如需要对某类果汁饮料浓度进行分等级，原汁含量超过 80%的为上等，50%～80%的为中等，低于 50%的为下等，则可以编制以下程序实现。

```
package main
import "fmt"
func main() {
    var (
        cff float64
        i   int
    )
    fmt.Println("请输入原汁含量百分比: ")
    fmt.Scanln(&cff)
    i = int(cff / 10)
    switch i {
    case 0:
        fallthrough
    case 1:
        fallthrough
    case 2:
        fallthrough
    case 3:
        fallthrough
    case 4:
        fmt.Printf("该饮料品质等级为: 下等\n")
    case 5:
        fallthrough
    case 6:
        fallthrough
    case 7:
        fmt.Printf("该饮料品质等级为: 中等\n")
    default:
        fmt.Printf("该饮料品质等级为: 上等\n")
    }
}
```

运行结果：

```
请输入原汁含量百分比:
75
该饮料品质等级为: 中等
```

程序分析：

从上述例子可以看出，需要在每个被穿越的 case 前面加一个 fallthrough 语句，这样重

复劳动太多，显得烦琐。Go 语言允许一个 case 有多个值，可以省略写多个 case 的情况。例如上述程序可以改写如下：

```
func main() {
    var cff float64
    fmt.Println("请输入原汁含量百分比：")
    fmt.Scanln(&cff)
    switch int(cff / 10) {
    case 0,1,2,3,4:
        fmt.Printf("该饮料品质等级为：下等\n")
    case 5,6,7:
        fmt.Printf("该饮料品质等级为：中等\n")
    default:
        fmt.Printf("该饮料品质等级为：上等\n")
    }
}
```

一个 case 含多个值，等同于多个 case 的串联，同时也隐含了多个 fallthrough，就像本例一样，把多个值写在一个 case 后面，就省略了给每个 case 写 fallthrough，这样程序就显得简洁多了。

对于需要共同执行同一动作的多种情况，使用这种方式最适宜。


视频讲解

3.4 select 语句

Go 语言是原生支持并发编程的语言，Go 语言的 goroutine 是其专有的执行单元，是非常轻量级的线程（也称为协程），程序执行期间可以创建成千上万个 goroutine。所有 goroutine 共享相同进程的内存空间，而各 goroutine 之间使用 channel 进行同步与通信。select 语句就是用来随机选择一个 channel 通信的。有关并发及通信的相关知识将在第 10 章介绍，这里仅仅说明 select 语句的一些语法知识。

从形式上来说，select 语句与 switch 语句类似，都是根据表达式的值来匹配多分支 case，匹配成功则执行该 case 分支；匹配不成功则执行 default 分支。但是，它们计算 case 表达式的顺序是不同的，switch 采用从上到下，从左到右的顺序，一旦匹配成功马上执行该 case 分支。而 select 不同，select 语句的 case 表达式只有通道操作语句，select 语句从上到下，从左到右扫描计算所有 case，寻找所有非阻塞的 case，然后随机选择一个非阻塞的 case 执行。下面通过一个例子来看看 select 语句的执行情况。

```
package main
import (
    "fmt"
    "math/rand"
)
func main() {
    a := make(chan int, 10)
    b := make(chan int, 10)
    c := make(chan int, 10)
    d := make(chan int, 10)
```

```
        for i := 0; i < 10; i++{
        q := rand.Intn(6)
        fmt.Printf("第 %d 次读取,rand = %d ", i, q)
        switch q {
        case 0:
            a <- 1
        case 1:
            b <- 1
        case 2:
            c <- 1
        case 3:
            d <- 1
        default:
            a <- 1
        }
        select {
        case <- a:
            fmt.Printf("通道为: %c\n", 'a')
        case <- b:
            fmt.Printf("通道为: %c\n", 'b')
        case <- c:
            fmt.Printf("通道为: %c\n", 'c')
        case <- d:
            fmt.Printf("通道为: %c\n", 'd')
        default:
            <- a
            fmt.Printf("通道为: %c\n", 'a')
        }
        }
}
```

运行结果：

```
第 0 次读取,rand = 5 通道为: a
第 1 次读取,rand = 3 通道为: d
第 2 次读取,rand = 5 通道为: a
第 3 次读取,rand = 5 通道为: a
第 4 次读取,rand = 1 通道为: b
第 5 次读取,rand = 0 通道为: a
第 6 次读取,rand = 1 通道为: b
第 7 次读取,rand = 2 通道为: c
第 8 次读取,rand = 4 通道为: a
第 9 次读取,rand = 0 通道为: a
```

程序分析：

上述程序首先定义了 4 个通道，然后通过一个随机数来控制通道赋值(往通道里写入 1)，接着让 select 语句随机选择一个有数据的通道进行读取，并打印通道号。如果不用随机给通道赋值，可以每次循环都给全部通道赋值。尽管每个 case 都有数据，但 select 语句仍然是每次随机选择一个有数据的通道读取。

同样，select 也支持 break 语句，在任意 case 分支上，如果提前结束程序流程，可以插入

一条 break 语句。如果有多层嵌套的 select 语句，需要一次性退出多层 select，可以加标签，将程序流程直接退出标签标注的 select 语句。参看以下例子：

```
package main
import "fmt"
func main() {
    a := make(chan int, 10)
    b := make(chan int, 10)
    c := make(chan int, 10)
    d := make(chan int, 10)
    for i := 0; i < 10; i++{
        fmt.Printf("第 %d 次读取,", i)
        a <- 1
        b <- 1
        c <- 1
        d <- 1
             out:
        select {
        case <- a:
            fmt.Printf("通道为: %c\n", 'a')
        case <- b:
            fmt.Printf("外层退出,放弃 b!\n")
        break
            fmt.Printf("通道为: %c\n", 'b')
        case <- c:
            fmt.Printf("通道为: %c\n", 'c')
        case <- d:
            select {
            case <- a:
                fmt.Printf("内层退出至外层,放弃 a:\n")
                break out     // out 指向外层 select 语句
                fmt.Printf("通道为: %c\n", 'a')
            case <- b:
                fmt.Printf("内层通道为: %c\n", 'b')
            }
        default:
            fmt.Printf("退出通道!\n")
        }
    }
}
```

运行结果：

```
第 0 次读取,通道为: c
第 1 次读取,内层退出至外层,放弃 a:
第 2 次读取,外层退出,放弃 b!
第 3 次读取,外层退出,放弃 b!
第 4 次读取,通道为: c
第 5 次读取,通道为: a
第 6 次读取,外层退出,放弃 b!
第 7 次读取,内层退出至外层,放弃 a:
第 8 次读取,通道为: a
第 9 次读取,通道为: c
```

程序分析：

仔细分析上述程序的运行结果，就能看清楚什么时候执行的是外层 break，什么时候执行的是内层 break。执行内层 break 的时候才能退出标签所在的 switch。

但是，fallthrough 语句不适用于 select 语句，加入了该语句，编译系统会报错（错误提示：fallthrough statement out of place）。


视频讲解

3.5 for 语 句

循环是编程实践中使用最多的语法结构，重复执行的动作都需要用循环完成。Go 语言不同于其他语言，它仅有 for 循环，而没有 while 和 do…while 循环。

3.5.1 基本语法

for 循环的基本语法格式如下：

```
for 可选前置子句; 布尔表达式; 可选后置子句{
    循环体
}
```

for 关键字后面由三部分构成：可选前置子句、布尔表达式、可选后置子句。各部分之间必须用分号“;”隔开。事实上，可选子句及表达式都是可以省略的，省略子句后分隔符“;”也要跟着省略。

如果可选子句及布尔表达式均省略，编译系统默认为逻辑真 true，则 for 循环就成了以下格式：

```
for{
      循环体
}
```

这是一个无限循环，如果需要退出循环，必须在循环体内设置退出条件，比如加上一条 break 语句。

大多数情况下的循环语句是以下格式的：

```
for 布尔表达式{
    循环体
}
```

当布尔表达式的值为 true 时，执行循环体。循环体执行结束之后程序流程将返回布尔表达式，再次计算布尔表达式的值，如果其值为 false，则退出循环；为真就继续执行循环体。

前置子句：必须为 Go 语言的简单语句，多用于变量的初始化，通常是短变量声明语句。需要注意的是，用短变量操作符“:=”声明的变量，其作用域仅为循环体内，循环体外不可见。大多数情况下，子句中声明的变量都是作为循环控制变量，用来控制循环次数。

布尔表达式：也称逻辑表达式，是循环的条件，可省略。省略的逻辑表达式其结果默认为 true。

后置子句：通常用来控制循环的次数，通过修改循环控制变量的值来影响逻辑表达式

的值，目标是使得循环控制变量逐渐脱离循环条件，最终结束循环。

后置子句是可选的，如果没有，则控制循环的表达式放在循环体内，在循环体内控制循环结束的条件。

for 循环程序的执行顺序是：

① 执行前置子句；

② 执行条件表达式，如果为假，则直接退出循环；为真，则执行循环体；

③ 执行循环体；

④ 执行后置子句；

⑤ 返回②继续执行条件表达式。

以下程序利用循环实现输出 20 以内的奇数。

```
package main
import (
    "fmt"
)
func main() {
    fmt.Printf("20 以内的奇数是: \n")
    for i := 1; i <= 20; i++{
        if i % 2 != 0 {
           fmt.Printf(" % 3d   ", i)
        }
    }
    fmt.Printf("\n")
}
```

运行结果：

```
20 以内的奇数是:
1 3 5 7 9 11 13 15 17 19
```

程序分析：

上述程序中利用模 2 运算，也就是求余运算，从 1 开始，能被 2 整除的数就是偶数，不能被 2 整除的数就是奇数。因此，通过 2 的求余运算，只要余数为 0 就抛弃，不为 0 就打印输出。奇数打印的时候是连续打印，不换行（不加转义字符\n），待全部打印完毕后打印一个“\n”，表示换行。

3.5.2 for 子句

for 子句主要是指前置子句和后置子句，它们都是可选的。所有子句必须为简单语句。前置子句多用来声明循环变量，后置子句多用来修改循环变量。仅当需要明确控制变量的作用域时，才把控制循环次数的变量放在前置子句中定义，一旦循环结束，该变量将不可见。这样做既可以节省内存，还可以提高变量使用的安全性。3.5.1 节例子就是在前置子句中声明的循环变量。当然也可以把变量声明及初始化表达式放在循环语句前面，变成以下形式：

```
func main() {
    fmt.Printf("20 以内的奇数是: \n")
```

```
    i := 1
    for ; i <= 20; i = i + 1{
        if i % 2 != 0 {
            fmt.Printf("%d  ", i)
        }
    }
    fmt.Println()
}
```

上述 fmt.Println()语句起到换行的作用。

如果有必要，还可以把后置子句省略掉，在循环体内修改循环条件，如下所示：

```
func main() {
    fmt.Printf("20 以内的奇数是: \n")
    i := 1
    for i <= 20{
        if i % 2 != 0 {
            fmt.Printf("%d  ", i)
        }
        i++
    }
    fmt.Printf("\n")
}
```

效果是一样的，这种方式更符合一般的逻辑。

将变量声明及初始化放在 for 前面，把循环控制语句放在循环体内最后一句，逻辑更加清晰，程序也更容易让人读懂。

for 关键字后省略掉前置和后置两个子句后，实际上变成以下形式：

```
for ; i <= 20; {
   循环体
}
```

可选子句省略了，但保留了两个分号，不算错，编译系统会自动把它去掉。但是位置不能出错，如果是下面的形式：

```
for ; ; i <= 20{
    循环体
}
```

或者

```
for i <= 20 ; ; {
  循环体
}
```

则编译系统会报错，程序无法执行。如果只是省略前置子句或后置子句其中一个，则分号不能省略，必须是如下形式：

```
for ; i <= 20 ; i++{
    循环体
}
```

或者

```
for i:= 0; i <= 20 ; {
    循环体
}
```

这是合法的，可以正常编译通过。

3.5.3 range 子句

通过循环来遍历一棵树的全部元素，或者一个数组的全部元素等，类似的应用在编程实践中经常遇到，这种一遍一遍的重复动作称为迭代。Go 语言专门提供了一个迭代关键字 range，用来完成迭代操作。range 配合 for 在循环中使用，其语法格式如下：

```
for 变量 1,变量 2 := range 迭代对象{
    循环体
}
```

迭代对象：可以是数组(array)、切片(slice)、字符串(string)、映射(map)及通道(channel)。不同的迭代对象，其返回的值也不同，具体见表 3-1。

表 3-1 range 返回值

变量 1(key)	变量 2(value)	迭代对象(object)
index	arrayName[index]	array
index	sliceName[index]	slice
index	stringName[index]	string
key	mapName[key]	map
element		chan

循环次数取决于迭代对象的内容长度，其长度可用内部函数 len 求得，如遇到空的字符串等会直接退出，不会遍历。

从表中可以看出，数组、切片、字符串、映射等都有两个返回值，变量 1 为索引值，其值从 0 开始，直至其长度 len－1；变量 2 为迭代对象的内容值。映射没有索引值，返回的是映射项中的键，第二个返回映射项中的值。而通道仅返回一个值，就是通道里的元素值。

变量 1 和变量 2 不需要预先定义，可以在 for 子句里采用短变量声明操作符声明并赋值。当然，这也不是必须的，在 for 子句里给已声明过的变量赋值也是允许的。关于数组、切片、字符串等知识尚未讲到，后续章节中会陆续说明，目前我们暂时用字符串来说明 range 子句的用法。以下例子是从键盘输入一个字符串，然后遍历每一个字符并输出。

```
package main
import (
    "fmt"
)
func main() {
    fmt.Printf("请输入一个给定的字符串: \n")
    var str string
    fmt.Scanf("%s", &str)
```

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

运行结果：

```
请输入一个给定的字符串:
abcde
i = 0 ch = a
i = 1 ch = b
i = 2 ch = c
i = 3 ch = d
i = 4 ch = e
```

程序分析：

使用 range 子句遍历字符串的时候，其遍历的第一个变量是其索引值，为 int 型，序号从 0 开始，是其 UTF-8 字节编码的下标。遍历的第二个变量为 rune 类型的字符的 Unicode 码值。如果遇到非法的 UTF-8 顺序，则第二个遍历的值为 0xFFED，并从下一个字节再次遍历。

上述变量 i 和 ch 在 for 子句里声明，其作用域仅在循环体内，遍历的结果也仅在循环体内有效，结束循环后 i 及 ch 均不可见。为了保存遍历结果，可以定义一个外部的字符串数组，将遍历出来的每一个字符存放数组里。

如果只是提取字符串中的每一个字符，也可以使用标准库函数来获取每一个字符，如下所示。

```
package main
import (
    "fmt"
    "unicode/utf8"
)
func main() {
    fmt.Printf("给定字符串: xαy zγ \n")
    var str string = "xαy zγ"
    for i := range str {
        char, size := utf8.DecodeRuneInString(str[i:])
        fmt.Printf("i = %d   char = %c   size = %d \n", i, char, size)
    }
    fmt.Printf("\n")
}
```

运行结果：

```
给定字符串: xαy zγ
i = 0  char = x   size = 1
i = 1  char = α   size = 2
i = 3  char = y   size = 1
i = 4  char =     size = 2
i = 6  char = z   size = 1
i = 7  char = γ   size = 2
```

程序分析：

上述程序使用了库函数，没有前一种方法方便直接，因此，推荐使用前一种方法。

需要注意的是，range 迭代字符串给出的 index 是字节偏移值，如果是纯 ASCII 码字符串，如“abcde”，则 index 的值为 0，1，2，3，4。如果是 UTF-8 字符串，如“xαyβzγ”这样的字符串，则 index 的值为 0，1，3，4，6，7。UTF-8 字符使用 Unicode 码点值，占两个字节。所以，遍历字符串字符用 UTF8 包的库函数比较安全可靠。

从上述程序中可以看出，在 range 中迭代出索引值，然后在循环体里根据索引获得元素值，这种用法在数组、切片的编程实践中经常使用。

在实际编程实践中，变量 1 和变量 2 有时可能只需要其中一个，这个时候可以用一个空字符“_”代替，作用是忽略该变量，如下所示。

```
_, ch = range str
```

或

```
i, _ = range str
```

甚至一个变量都不要，也是允许的，如下所示。

```
package main
import (
    "fmt"
)
func main() {
    var str = "astring"
    i := 1
    for range str {
        fmt.Printf("str[ %d] = %c \n", i, str[i])
        if i++; i == len(str) {
            break
        }
    }
}
```

运行结果：

```
str[1] = s
str[2] = t
str[3] = r
str[4] = i
str[5] = n
str[6] = g
```

程序分析：

上述索引 i 从 1 开始，因此 str[0]的字符 a 并没有输出，这进一步说明了，用下标方式索引字符串、数组、切片等，下标值一定要从 0 开始。

如果赋值号左边只写一个变量，默认是变量 1(index)，如果需要获取数组或切片元素的值及其索引，则必须有第二个变量。

for …range 遍历语句中的 for 后面不再支持加入子句，如果加入了子句会编译出错，如

下所示。

```
func main() {
    var y = []int{1,2,3,4,5,6}
    for sum := 0; _, value := range y {
        sum += value
    }
    fmt.Println("sum = ", sum)
}
```

上述程序中把变量 sum 的声明及初始化放在了 for 后面的子句里，导致编译出错，syntax error: unexpected range, expecting expression。

如果把初始化语句提到 for 前面一行，编译就通过了，如下所示。

```
func main() {
    var y = []int{1, 2, 3, 4, 5, 6}
    sum := 0
    for _, value := range y {
        sum += value
    }
    fmt.Println("sum = ", sum)
}
```

运行结果：

```
sum = 21
```

3.5.4 break 语句

for 循环的结束由其逻辑表达式的值决定，当出现逻辑假时，循环结束。如果有必要，可以人为地结束循环，这就是 break 语句的使用。上述例子中就是使用 break 结束循环的。break 结束的是离它所在位置最近的最内层循环，如果需要退出到最外层循环，或者是某一层循环，则可以在 break 后面加标签，指定退出的是哪一层循环。想要结束哪一层循环，就在该层循环前面加一个标签，如下述例子所示。

假设在一个有 i 行 j 列的二维矩阵 Matrix 中搜索一个特定值 x，搜索成功就将标志符 flag 置 1，失败就将 flag 清零，程序实现如下。

```
package main
import (

    "fmt"
)
func main() {
    var Matrix = [][]int{{12, 25, 14, 29}, {32, 58, 41, 42}, {24, 31, 56, 63}}
    flag := 0
    x := 42
    Found:
    for i := range Matrix {
    for j := range Matrix[i] {
        if Matrix[i][j] == x {
```

```
                flag = 1
                fmt.Printf("\n搜索成功!flag = %d\n", flag)
                fmt.Printf("Matrix[%d][%d] = %d \n", i, j, Matrix[i][j])
                break Found
            }
            }
        }
    }
```

运行结果：

```
搜索成功!flag = 1
Matrix[1][3] = 42
```

程序分析：

上述例子中首先定义了一个 3 行 4 列的数组切片，假设一个被搜索数据 42，通过两层循环来扫描所有数据。

搜索成功后置 flag 为 1，并打印标志位及该矩阵位置的值。break 后加了标签 Found，指明是外层循环，break 直接退出外层循环。如果不加标签，则退出的是内层循环，内层循环之外还得加一个 break 才能退出外层循环。

3.5.5 continue 语句

在循环编程实践中，常常会碰到这种情况：循环体程序运行到某一处，已经获得了期望的结果，不需要继续执行后续的语句了，这时该怎么做呢？这就要用到 continue 语句了。

continue 语句的执行逻辑是中断本轮循环，将程序控制流程返回到 for 后置子句，重新计算循环次数，接着计算逻辑表达式的值。如果为 true，则进入下一轮循环；如果为 false，则直接退出循环，等同执行 break 语句。

需要注意的是，如果 for 语句省略了后置子句，而将循环控制变量的计算放在循环体末尾，这时执行 continue 中断循环，并不会执行循环控制变量的运算，而是直接返回执行逻辑表达式计算。由于循环变量没改变，逻辑表达式的值也就不会改变，从而又重复执行本次循环，反复执行同一轮循环将导致程序死锁。

另外，continue 语句只适合于 for 循环，不适合于 switch 和 select 语句。

以下程序实现求一个最小的四位整数，要求该整数的个位为 2，十位是 4，且减去 3 后能被 3 整除，减去 7 后能被 7 整除。

```
package main
import (
    "fmt"
)
func main() {
    for i := 1000; i <= 9999; i++{
        if i%10 != 2 {
            continue
        } else if i/10%10 != 4 {
            continue
        } else if (i-3)%3 != 0 {
```

```
            continue
        } else if (i - 7) % 7 != 0 {
            fmt.Printf("\n满足条件的数是: %d\n", i)
            break
        }
    }
}
```

运行结果：

```
满足条件的数是: 1242
```

程序分析：

上述程序多次使用 continue 中断当前循环轮次，进入下一轮次循环，当所有条件都满足之后，由 break 退出循环体。

上述程序的执行逻辑是：只要第一个条件不满足，则其余条件就不用检查了，直接用 continue 中断当前循环轮次，进入下一轮次循环，继续比对下一个数据。

3.5.6 goto 语句

goto 语句是程序流程跳转语句，改变程序流程，直接跳转到同一函数内某个标签所在的位置。goto 语句不允许跨越函数，否则编译系统会报错。

在很多编程语言中 goto 语句已经被取消，人为控制程序流程极容易导致程序逻辑混乱，程序难以移植，难以读懂，不易维护。Go 语言保留了 goto 语句，主要是为了编程的灵活性，适应某些需求。但是，编程实践中建议尽量少使用 goto 语句。goto 语句的使用非常简单，在函数体内有需要的地方插入标签，在 goto 语句后边加上标签即可。

参看下例，寻找 10 以内减 3 且能被 3 整除的数：

```
package main
import (
    "fmt"
)
func main() {
    i := 10
    for i > 0 {
        i--
            if i == 0 {
        break
        }
        if (i - 3) % 3 == 0 {
            goto cont
        } else {
            continue
        }
        cont:
        fmt.Printf("10 以内减 3 且能被 3 整除的数是: %d\n", i)
    }
}
```

运行结果：

```
10 以内减 3 且能被 3 整除的数是: 9
10 以内减 3 且能被 3 整除的数是: 6
10 以内减 3 且能被 3 整除的数是: 3
```

程序分析：

上述程序算法比较简单，仅用求余运算就可以解决问题。程序中加入了 goto 语句，goto 语句后必须带标签，表示跳转的目的地。

上 机 训 练

一、训练内容

1. 请用循环语句实现以下圣诞树。

```
          *
         ***
         *****
        *******
        *********
         *****
        *******
        *********
      ***********
      *************
        *********
      ***********
      *************
    ***************
    *****************
         *****
         *****
         *****
```

2. 请编程实现从键盘输入一门课程成绩，为其分级，等级为优秀、良好、中等、及格和不及格，要求用两种方式实现：if…else if 和 switch case。

二、参考程序

1. 圣诞树参考程序。

参考程序

```
package main

import (
    "fmt"
)

func main() {
    mid := 30                                    // 控制树顶的打印位置
```

续

参考程序

```
    for a := 0; a < 5; a += 2 {                    // 控制圣诞树层数
        // 打印树叶
        for i := 0; i < 5; i++{                    // 打印行数
            for j := 0; j < mid - i - a; j++{      // 打印空格数
                fmt.Printf(" ")
            }
            for j := -(i + a); j <= i + a; j++{    // 打印 * 数量
                fmt.Printf("*")
            }
            fmt.Printf("\n")
        }
    }
    for i := 0; i < 3; i++{                        // 打印树干
        for j := 0; j <= mid - 3; j++{
            fmt.Printf(" ")
        }
        fmt.Printf("*****\n")
    }
}
```

运行结果

```
                    *
                   ***
                  *****
                 *******
                *********
                  *****
                 *******
                *********
               ***********
              *************
                *********
               ***********
              *************
             ***************
            *****************
                  *****
                  *****
                  *****
```

2. 成绩等级划分参考程序。

(1) if…else if 方式。

参考程序

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("请输入一门课程的成绩:")
    var score int
```

参考程序

```
    fmt.Scanf("%d", &score)
    if score >= 90 && score < 100 {
        fmt.Println("成绩等级为：优秀")
    } else if score >= 80 && score < 90 {
        fmt.Println("成绩等级为：良好")
    } else if score >= 70 && score < 80 {
        fmt.Println("成绩等级为：中等")
    } else if score >= 60 && score < 70 {
        fmt.Println("成绩等级为：及格")
    } else if score >= 0 && score < 60 {
        fmt.Println("成绩等级为：不及格")
    }
}
```

运行结果

```
请输入一门课程的成绩：
89
成绩等级为：良好
```

（2）switch case 方式。

参考程序

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("请输入一门课程的成绩：")
    var score int
    fmt.Scanf("%d", &score)
    switch score / 10 {
    case 9, 10:
        fmt.Println("成绩等级为：优秀")
    case 8:
        fmt.Println("成绩等级为：良好")
    case 7:
        fmt.Println("成绩等级为：中等")
    case 6:
        fmt.Println("成绩等级为：及格")
    default:
        fmt.Println("成绩等级为：不及格")
    }
}
```

运行结果

```
请输入一门课程的成绩：
76
成绩等级为：中等
```

习　　题

一、单项选择题

1. 以下全局变量定义正确的是(　　)。

A. var X int=10　　B. X :=10　　C. int X=10　　D. var X=10

2. 以下全局可导出变量的定义正确的是(　　)。

A. set Name　　B. int Name　　C. var Name int　　D. var name int

3. 以下局部变量定义正确的是(　　)。

A. int x :=10　　B. var x :=10　　C. x :=10　　D. int x=10

4. 已知：a=10,b=3.14,c=20,以下表达式正确的是(　　)。

A. a=100 * b-c　　B. a+=b　　C. c=10%3+a　　D. c=10 * a/b

5. 已知：a=1,b=3.2,c=20,以下表达式正确的是(　　)。

A. a=++b-c　　B. a++=b

C. c=10/3+a++　　D. c=10 * a/c

6. 已定义 c 为字符型变量,则下列语句中正确的是(　　)。

A. c='97'　　B. c="97"　　C. c=97　　D. c="a"

7. 以下 if 语句格式正确的是(　　)。

A. if (x>y && y+1) {block}　　B. if x+y {block}

C. if x>0 && y>1 {block}　　D. if x=5 {block}

8. 以下 if 语句格式正确的是(　　)。

A. if x:=0>y {block}　　B. if x=x+y; x>y {block}

C. if x>0||y=1 {block}　　D. if !(x=5){block}

9. 以下 if 语句格式正确的是(　　)。

A. if x:=0; y<=10 {block}　　B. if x>y; x=x-1 {block}

C. if x>0;; {block}　　D. if x>=5; x=x-1 {block}

10. 以下 switch 语句格式正确的是(　　)。

A.
```
switch x :=0; y<=10{
    case 0: block0
    case 1: block1
    default: blockd
}
```

B.
```
switch x :=0; y=10 {
    case 0: block0
    case 1: block1
    default: blockd
}
```

C.
```
switch y <=10 {
    case 0: block0
    case 1: block1
    default: blockd
}
```

D.
```
switch x :=0; y {
    case 0: block0
    case 1: block1
    default: blockd
}
```

二、判断题

1. if 语句的表达式必须为算术表达式。（　　）
2. if 语句的表达式必须为逻辑表达式。（　　）
3. if 语句中的表达式子句是必须的。（　　）
4. if 语句中的表达式是可选的。（　　）
5. if 语句中的前置子句是可选的。（　　）
6. for 语句中的后置子句是可选的。（　　）
7. 当 for 语句中的前置及后置子句其中之一缺省时，缺省子句前面或后面的分号不可省略。（　　）
8. 在外层 if 前置子句声明的变量在嵌入的内层 if 表达式中是可见的。（　　）
9. 当 switch 语句不带表达式时，其 case 子句的表达式必须为逻辑表达式。（　　）
10. 当 switch 语句带逻辑表达式时，其 case 子句的表达式必须为逻辑表达式。（　　）
11. 当 switch 语句带逻辑表达式时，其 case 子句的表达式可以是算术表达式。（　　）
12. 当 switch 语句带算术表达式时，其 case 子句的表达式必须为逻辑表达式。（　　）
13. 当 switch 语句带算术表达式时，其 case 子句的表达式必须为算术表达式。（　　）
14. fallthrough 语句可以穿越“表达式 switch”的 case 分支。（　　）
15. fallthrough 语句可以穿越“类型 switch”的 case 分支。（　　）
16. fallthrough 语句可以穿越 select 语句的 case 分支。（　　）
17. break 语句可以退出 for 循环语句。（　　）
18. break 语句可以退出 switch…case 语句。（　　）
19. break 语句可以退出 if…else 语句。（　　）
20. continue 语句可以用于 switch…case 语句。（　　）
21. continue 语句只能用于 for 循环语句。（　　）
22. 标签 lable 只能用于 break 语句或 goto 语句。（　　）
23. 标签 lable 也可以用于 continue 语句或 fallthrough 语句。（　　）
24. break 语句一次性退出多层循环必须使用标签 lable。（　　）

三、分析题

1. 分析以下程序，写出运行结果，并分析其实现的功能。

```
package main
import (
        "fmt"
)
func main() {
    UU := 0
    for i := 0; i < 2; i++{
    fmt.Printf("这是外部变量 UU 的值: %d     循环变量 i = %d\n", UU, i)
    var UU int = 1
        UU++
        fmt.Printf("这是内部变量 UU 的值: %d     循环变量 i = %d\n", UU, i)
```

```
}
fmt.Println("这是退出循环后外部变量 UU 的值: ", UU)
}
```


2. 分析以下程序,写出运行结果,并分析其实现的功能。

```
package main
import (
    "fmt"
)
func main() {
    var i = 10
    switch i++; {
        case i >= 10:
            i++
        case i >= 11:
            i++
        case i >= 12:
            i++
        default:
            i++
    }
    fmt.Println("i = ", i)
}
```

3. 分析以下程序,写出运行结果。

```
package main
import "fmt"
func main() {
    var a, b = 'a', 'b'
    a++
    fmt.Printf("%c,", a)
    a = b + 10
    b = a
    fmt.Printf("%c\n", b)
}
```

4. 分析以下程序,写出运行结果。

```
package main
import "fmt"
func main() {
    var i int
    for i = 0; i < 3; i = i + 1 {
        switch i {
           case 1:
          fmt.Printf("%d,", i)
           case 2:
          fmt.Printf("%d\n", i)
        }
    }
}
```

5. 分析以下程序,写出运行结果。

```
package main
import "fmt"
func main() {
    var ch = 'W'
    if ch > 'A' && ch < 'Z' {
       ch = ch + 32
    } else {
       ch = ch - 32
    }
    fmt.Printf("%c\n", ch)
}
```

6. 分析以下程序,写出运行结果。

```
package main
import "fmt"
func main() {
    var a, c, b, d, x = 2, 6, 5, 4, 10
    if a < b {
        if c < d {
            x = 1
        } else if a < c {
            if b < d {
                x = 2
            } else {
                x = 3
            }
        } else {
            x = 6
        }
    } else {
        x = 7
    }
    fmt.Printf("%d\n", x)
}
```

7. 分析以下程序,写出运行结果。

```
package main
import "fmt"
func main() {
    var a, c, b, x = 5, 2, 3, 35
    x--
    if !(a < b) && ^c > a {
        if c < b {
            x = 1
        } else if a > c {
            x = 2
        } else {
            x = 3
```

```
        }
    } else {
        x = 6
    }
    fmt.Printf("%d\n", x)
}
```

8. 分析以下程序,写出运行结果。

```
package main
import "fmt"
func main() {
    var x, y, z = 16, 21, 0
    switch x % 3 {
    case 0:
        z++
    case 1:
         z++
        switch y % 2 {
        case 0:
            z++
            break
             default:
            z++
        }
    }
    fmt.Printf("%d\n", z)
}
```

9. 若 x=-4,y=8,z=1,m=-13,计算下列各表达式的值。

A. x+y<z+m%4　　B. x+3*y+z>=2*x-m

C. 4*y/x-m>y%2-z　　D. !(x>2*y)||(z+m)<x

E. x<3*m/4&&y%5<z

10. 下列代码执行后,x,y,z 的值各是多少?

```
package main
import "fmt"
func main() {
    var x, y, z = 0, 0, 1
    switch x {
    case 0:
        x = 2
        y = 3
    case 1:
        x = 4
    default:
        x = 1
        y = 3
    }
    fmt.Printf("x = %d, y = %d, z = %d\n", x, y, z)
}
```

11. 下列代码执行后，x，y，z 的值各是多少？

```
package main
import "fmt"
func main() {
    var x, y, z = 2, 1, 1
    switch x % 2 {
    case 0:
        x = 2
        y = 3
    case 1:
        x = 4
    default:
        x = 1
        y = 3
    }
    fmt.Printf("x = %d, y = %d, z = %d\n", x, y, z)
}
```

12. 下列代码执行后，x，y，z 的值各是多少？

```
package main
import "fmt"
func main() {
    var x, y, z = 1, 3, 0
    switch x + y % 2 {
    case 0:
        x = 5
        y = 7
    case 1:
        x = 4
    default:
        x = 9
        y = 8
    }
    fmt.Printf("x = %d, y = %d, z = %d\n", x, y, z)
}
```

四、简答题

1. Go 语言流程控制语句主要包括哪些？
2. if 语句的子句有什么限制条件？
3. if 语句的语法格式有什么样的规定？
4. 什么情况下可以省略 else 子句？
5. if 语句中子句声明的变量其作用域到哪里结束？
6. 什么叫代码块？
7. 什么是变量的可见性？
8. 什么叫变量的作用域？
9. 什么是全局变量？

10. 什么叫局部变量？
11. 内部变量对外部变量的覆盖是什么意思？
12. 如何快速退出嵌套的 switch 语句？
13. 什么是标签？有什么作用？
14. switch 有几种类型？各有什么特点？
15. 表达式 switch…case 和 select…case 有什么区别？
16. switch 语句的执行流程是什么样的？
17. select 语句的执行流程是什么样的？
18. switch 子句声明的变量作用域在哪里？
19. break 语句的适用范围是什么？
20. break 语句的标签应该放在程序中什么位置？
21. fallthrough 语句的功能和适用范围是什么？
22. for 的前置子句省略后，其后面的分号也要省略吗？
23. range 子句迭代数组输出两个变量，分别是什么？
24. continue 语句的适用范围是什么？

五、编程题

1. 请编程实现从键盘输入一个任意位数的整数，按逆序打印输出每一位数据。
2. 请编程实现从键盘输入三个整数，按从小到大顺序打印输出。
3. 判断 100～200 有多少个素数，并输出所有素数。
4. 编程实现从键盘输入两个正整数 m 和 n，求其最大公约数和最小公倍数。
5. 请编程从键盘输入一行字符，分别统计出其中大写英文字母、小写英文字母、数字和其他字符的个数。
6. 请编程实现从键盘输入一个任意整数，求该数的阶乘并输出。
7. 请编程实现求 1＋2＋3＋…＋100 的和并打印输出。
8. 请编程实现从键盘输入四门课程的成绩，求其最高分、最低分及平均分并打印输出。
9. 请编程输出 9×9 乘法口诀表(分别用两种形式输出：正方形和三角形)。
10. 一球从 100 米高度自由落下，每次落地后反跳回原高度的一半；再落下，求它在第 10 次落地时，共经过多少米？第 10 次反弹多高？