layout: post
title: 004-Go基础之三 流程控制
category: golang架构师之路
tags: golang,tour,function,variables,package

keywords: package,tour,function,variables

流程控制语句: `if,else,switch,for and defer`

学习如何使用条件、开关分支switch、循环、推迟语句控制代码流程

for循环

for基本组成

Go只有一种循环结构，即for循环，没有while等关键字

基本的for循环由分号分隔成三部分组成:
`for init[optional];conditin;after[optional]; {}`

- 初始化部分:第一次迭代之前执行
- 条件表达式部分:每次迭代之前计算求值
- 后置部分:每次迭代之后执行

初始化部分通常是简短的变量声明，并且被声明的变量只能在for语句块范围内可见。

当条件表达式值为false时，循环就会终止。

注意：Go的for循环和其他语言比如C，Java，JavaScript不一样，没有且并不需要括号()括起来其组成的三部分，并且循环体{}不能少。

```
package main

import "fmt"

func sum() int {
    sum := 0
    for i := 0; i <= 10; i++ {
```

```
        sum += i
        fmt.Printf("i=%v, sum=%v\n", i, sum)
    }
    return sum
}

func main() {
    fmt.Printf("sum:%v\n", sum())
}

// i=0, sum=0
// i=1, sum=1
// i=2, sum=3
// i=3, sum=6
// i=4, sum=10
// i=5, sum=15
// i=6, sum=21
// i=7, sum=28
// i=8, sum=36
// i=9, sum=45
// i=10, sum=55
// sum:55
```

注意: 如果条件表达式中含有表达式len(a)最好保存在变量中，否则在循环次数较多或计算耗时时影响应能。

for组成的可选部分

初始部分和后置部分都是可选的。

```
package main

import "fmt"

func sum() int {
    sum := 1
    for ; sum < 1000; {
        sum += sum
        println(sum)
    }
    return sum
}

func main() {
    fmt.Println(sum())
}
```

```
// 1024
// 2
// 4
// 8
// 16
// 32
// 64
// 128
// 256
// 512
// 1024
```

for是Go语言的"while"

基于此你可以省略分号：Go语言的`for`等同于C语言的`while`，也就是说前置短声明变量与后置语句的省略，再去除分号，**可以只保留条件表达式部分，等价于while循环**
`for condition {}`

```
func sum() int{
    sum := 1
    for sum<1000 {
        sum += sum
    }
    return sum
}
```

可以使用while循环重写高斯算和的案例

```
package main

import "fmt"

func sum() int {
    sum := 0
    i := 0
    for i <= 10 {
        sum += i
        fmt.Printf("i=%v, sum=%v\n", i, sum)
        i++
    }
    return sum
}

func main() {
```

```
    fmt.Printf("sum:%v\n", sum())
}

// i=0, sum=0
// i=1, sum=1
// i=2, sum=3
// i=3, sum=6
// i=4, sum=10
// i=5, sum=15
// i=6, sum=21
// i=7, sum=28
// i=8, sum=36
// i=9, sum=45
// i=10, sum=55
// sum:55
```

死循环

如果省略表达式条件将会永远循环下去，所以更简单直接的表达就是死循环。
`for {}`

```
func main() {
    for{} // infinite for loop
    //for ; ;  {}
}
```

if语句

Go的`if`语句块与它的`for`循环类似；表达式不需要被括号括起来，但是需要`{}`括起`if`的执行体。

用法:
`if condition {}`

```
package main

import (
    "fmt"
    "math"
)

func sqrt(f float64) string {
```

```
    fmt.Printf("%v\n", f)
    if f < 0 {
        return sqrt(-f)
    }
    return fmt.Sprint(math.Sqrt(f),9999)
}

func main() {
    fmt.Printf("%v, %v", sqrt(2), sqrt(-4))
}

// 2
// -4
// 4
// 1.4142135623730951 9999, 2 9999
```

带有简短语句块的if

像`for`循环一样，`if`语句可以在条件表达式之前先执行一个简短的语句。

在这个简短的表达式块中声明的变量生命周期只在`if`块中。

```
func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    }
    return lim
}

func main() {
    fmt.Printf("%v\n", pow(2, 3, 10))// 8
    fmt.Printf("%v", pow(3, 2, 6))// 6
}
```

if else

变量声明可以在if的简短语句中也可以在else语句块中。

```
package main

import "fmt"
```

```
func pow(x, n, lim float64) float64 {
    if fmt.Printf("%q\n", "if,else执行前都要先执行"); 1 < lim {
        return 1
    } else {
        fmt.Printf("%g >= %g\n", 1, lim)
    }
    return lim
}

func main() {
    fmt.Printf("%v\n", pow(2, 3, 10))
    fmt.Printf("%v", pow(3, 2, -1))
}

// "if,else执行前都要先执行"
// 1
// "if,else执行前都要先执行"
// %!g(int=1) >= -1
// -1
```

在 main方法 的 `fmt.Println` 调用开始前，两次对 `pow` 的调用均已执行并返回。

```
package main

import "fmt"

func pow(x, n, lim float64) float64 {
    if 1 < lim {
        return 1
    } else if fmt.Printf("%q\n", "if,else执行前都要先执行"); 1>0 {
        fmt.Printf("%g >= %g\n", 1, lim)
    }
    return lim
}

func main() {
    fmt.Printf("%v\n", pow(2, 3, 10))
    fmt.Printf("%v", pow(3, 2, -1))
}

// 1
// "if,else执行前都要先执行"
// %!g(int=1) >= -1
// -1
```

```go
package main

import "fmt"

func main() {

    b := true

    if food := "Chocolate"; b {
        fmt.Println(food)
    }

    if fmt.Println("hello"); !b {
        fmt.Println("world")
    } else if fmt.Println("else if");!b {
        fmt.Println("else")
    }

}
```

if else if 短路情况就知道了，未必都会执行，只要if成立，elseif的输出就不会走。假设if不成立，但if也会执行，elseif 成立当然执行。

```go
package main

import (
    "runtime"
    "fmt"
)

var prompt = "Enter a digit, e.g. 3 " + "or %s to quit."

func init() {
    if runtime.GOOS == "windows" {
        prompt = fmt.Sprintf(prompt, "Ctrl+Z, Enter")
    } else { //Unix-like
        prompt = fmt.Sprintf(prompt, "Ctrl+D")
    }
}

func main() {
    fmt.Printf("%s", prompt)
}
```

省略else

当if结构内含有break，return，goto，continue时，go代码的常见写法是省略else。无论哪个条件返回x或y时，一般使用如下写法:

```
if condition {
    return x
}
return y
```

多返回值惯用方法

```
value, err := pack1.Function1(param1)
if err != nil {
    fmt.Printf("An error occured in pack1.Function1 with parameter %v", para
m1)
    return err
}
// 未发生错误，继续执行：
```

测试 err 变量是否包含一个真正的错误（if err != nil）的习惯用法。如果确实存在错误，则会打印相应的错误信息然后通过 return 提前结束函数的执行。我们还可以使用携带返回值的 return 形式，例如 return err。这样一来，函数的调用者就可以检查函数执行过程中是否存在错误了。

```
if err != nil {
    fmt.Printf("Program stopping with error %v", err)
    os.Exit(1)
}
```

使用 `os` 包的 `exit` 函数，**退出代码 `1` 可以被外部脚本获取到**

```
f, err := os.Open(name)
if err != nil {
    return err
}
dosomething(f)
do...
```

if语句习惯总结

1. Go语言的习惯是在**if中处理错误然后直接返回**,这样可以确保正常执行的语句不需要代码缩进。

```
package main

import (
    "os"
    "log"
    "fmt"
    "io/ioutil"
)

func main() {

    file, err := os.Open("/usr/local/a.txt")
    if err != nil {
        log.Fatal("error")
        return
    }
    content, err := ioutil.ReadAll(file)
    if err != nil {
        log.Fatal("err read")
        return
    }
    fmt.Println(string(content))

    // go中并不推荐下面的写法，而是更建议上面的写法。Go语言的习惯是在if中处理错误然后直接返回,这样可以确保 正常执行的语句不需要代码缩进。
    if file, err := os.Open("/usr/local/a.txt"); err != nil {
        log.Fatal("error")
        return
    } else {
        content, err := ioutil.ReadAll(file)
        if err != nil {
            log.Fatal("err read")
            return
        }
        fmt.Println(string(content))

    }
}
```

2.

```
package main

import (
    "fmt"
    "os"
    "log"
)

var pwd string

func main() {
    pwd, err := os.Getwd() //compile error unused variable pwd. pwd declared
 and not used
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
    //log.Printf("Working directory = %s", pwd) 加上这行代码，虽然用到了main函数
中的pwd,但是我们想要的是保存在全局中定义的pwd变量，这并不符本意。
}
```

解决方案，可以采取以下方式:不使用 `:=` 变量声明语句，而是 `=` 赋值操作。

```
go
var err error
pwd, err = os.Getwd()
if err != nil {
log.Fatalf("os.Getwd failed: %v", err)
}
```

练习：循环与函数

我们来简单练习一下函数和循环：用牛顿法实现平方根函数。
在本例中，牛顿法是通过选择一个起点 z 然后重复以下过程来求 Sqrt(x) 的近似值：

```
z = z - (z*z - x) / (2*z)
```

为此只需重复计算 10 次，并且观察不同的值（1，2，3，……）是如何逐步逼近结果的。 然后，修改循环条件，使得当值停止改变（或改变非常小）的时候退出循环。观察迭代次数是否变化。结果与 math.Sqrt 接近吗?
提示：用类型转换或浮点数语法来声明并初始化一个浮点数值：

```
z := float64(1)
```

```
z := 1.0
```

```
package main

import (
    "fmt"
)

func Sqrt(x float64) float64 {
    z := float64(x)
    for i := 0; i < 10; i++ {
        z = z - (z*z-x)/(2*x)
        fmt.Printf("%v\n", z)
    }
    return z
}

func main() {
    Sqrt(2)
}
```

精度控制版本:

```
package main

import (
    "fmt"
    "math"
)

func Sqrt(x float64) float64 {
    z := float64(x)
    y := 0.0
    //y := float64(0)
    for {
        z = z - (z*z-x)/(2*x)
        if math.Abs(y - z) < 1e-10 {
            break
        }
        y = z
        fmt.Printf("%v\n", y)
    }
    return y
}
```

```
func main() {
    Sqrt(2)
}
```

swtich语句

- switch 条件语句可以为任意类型或表达式
- 除非以 `fallthrough` 语句结束(全部判断一遍)，否则分支会自动终止(**匹配则自动break**)。
- switch语句在表达式之前也可以有语句执行，也就是初始化语句，类似if和for初始语句

switch几种简单形式

1. `swtich condition {case a: ..}`

```
package main

import "fmt"

func main() {
    a := 1
    switch a {
    case 0:
        fmt.Println("a=0")
    case 1:
        fmt.Println("a=1")
    default:
        fmt.Println("none")
    }

}
```

1. `switch {case a:....}`

```
package main

import "fmt"

func main() {
    a := 1
    switch  {
    case a >= 0:
        fmt.Println("a=0")
        fallthrough //想要继续判断下一个条件
```

```
    case a >= 1:
        fmt.Println("a=1")
    default:
        fmt.Println("none")
    }

}
```

1. `switch init;{case a: ..}` 初始化语句

```
package main

import "fmt"

func main() {

    switch a := 1; { // 注意初始化语句必须加`;`，即使只有初始化语句。否则syntax erro
r: a := 1 used as value
    case a >= 0:
        fmt.Println("a=0")
        fallthrough //想要继续判断下一个条件
    case a >= 1:
        fmt.Println("a=1")
    default:
        fmt.Println("none")
    }
}
```

```
package main

import (
    "runtime"
    "fmt"
)

func main() {
    switch os := runtime.GOOS; os {
    case "linux":
        println("case linux")
        fmt.Printf("%q", os)
    case "darwin":
        println("case darwin")
        fmt.Printf("%q", os)
```

```
    default:
        println("case default")
        fmt.Printf("%q", os)
    }
}
```

switch计算求值顺序

switch的case语句自上而下顺序执行，直到匹配成功终止。

例如:

```
go
switch i {
case 0 :
case f():
}
```

如果 `i==0`，则不会再调用 `f()`

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("when is saturday?")
    today := time.Now().Weekday()
    switch time.Saturday {
    case today + 0:
        fmt.Printf("today is saturday %q", today)
    case today + 1:
        fmt.Printf("tomorrday is saturday %q", today)
    case today + 2:
        fmt.Printf("in two days %q", today)
    default:
        fmt.Printf("too far away %q\n", today)
        fmt.Printf("too far away %v", today)
    }
    /*
    when is saturday?
    too far away "Wednesday"
    too far away Wednesday
```

```
    */
}
```

不带条件的switch

- 没有条件的switch等同于 `switch true` 。这种形式能够将一长串 `if-then-else` 写得更清晰。
- 也可以理解为**条件表达式要么在 switch关键字后面，要么在case后面**

```
package main

import (
    "time"
    "fmt"
)

func main() {
    t := time.Now()

    switch  {
    case t.Hour() < 12:
        fmt.Printf("Good morning! %v", t.Hour())
    case t.Hour() < 17:
        fmt.Printf("Good afternoon! %v", t.Hour())
    default:
        fmt.Printf("Good evening! %v", t.Hour())
    }
}
```

案例2

```
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.Now()
    hour := t.Hour()

    if hour < 12 {
        fmt.Printf("morning: %v", hour)
```

```
    }

    if hour < 18 {
        fmt.Printf("afternoon: %v", hour)
    }

    if hour < 24 {
        fmt.Printf("evening: %v", hour)
    }

    switch {
    case hour < 12:
        fmt.Printf("morning: %v", hour)
    case hour < 18:
        fmt.Printf("afternoon: %v", hour)
    case hour < 24:
        fmt.Printf("evening: %v", hour)
    }
}
```

switch true中需要case的表达式为布尔表达式。

多个值switch计算

```
package main

import "fmt"

func main() {
    switch "Jenny" {
    case "Tim", "Jenny":
        fmt.Println("Wassup Tim, or, err, Jenny")
    case "Marcus", "Medhi":
        fmt.Println("Both of your names start with M")
    case "Julian", "Sushant":
        fmt.Println("Wassup Julian / Sushant")
    }
}
```

没有表达式

```
package main
```

```
import "fmt"

func main() {

    myFriendsName := "Mar"

    switch {
    case len(myFriendsName) == 2:
        fmt.Println("Wassup my friend with name of length 2")
    case myFriendsName == "Tim":
        fmt.Println("Wassup Tim")
    case myFriendsName == "Jenny":
        fmt.Println("Wassup Jenny")
    case myFriendsName == "Marcus", myFriendsName == "Medhi":
        fmt.Println("Your name is either Marcus or Medhi")
    case myFriendsName == "Julian":
        fmt.Println("Wassup Julian")
    case myFriendsName == "Sushant":
        fmt.Println("Wassup Sushant")
    default:
        fmt.Println("nothing matched; this is the default")
    }
}

/*
  expression not needed
  -- if no expression provided, go checks for the first case that evals to t
rue
  -- makes the switch operate like if/if else/else
  cases can be expressions
*/
```

注意该种场景，也是完全可以使用`,`表示一种情况下多个值的比较。

switch 类型断言

```
package main

import "fmt"

//  switch on types
//  -- normally we switch on value of variable
//  -- go allows you to switch on type of variable

type contact struct {
```

```
    greeting string
    name     string
}

// SwitchOnType works with interfaces
// we'll learn more about interfaces later
func SwitchOnType(x interface{}) {
    switch x.(type) { // this is an assert; asserting, "x is of this type"
    case int:
        fmt.Println("int")
    case string:
        fmt.Println("string")
    case contact:
        fmt.Println("contact")
    default:
        fmt.Println("unknown")

    }
}

func main() {
    SwitchOnType(7)
    SwitchOnType("McLeod")
    var t = contact{"Good to see you,", "Tim"}
    SwitchOnType(t)
    SwitchOnType(t.greeting)
    SwitchOnType(t.name)
}
```

switch type类型自推断

可以先了解下 go-控制结构中的 `ok partner` 的类型判断。

实际上我们使用 `ok partner` 的方式只适用于少量的分支判断，但是如果想本例中传入的是广泛的空接口，这种想法将很麻烦。switch结构更加适合，配合Go的类型推断和 `switch type` 结构。

```
func stop(printer empty) {
    if hp, ok := printer.(HP_Printer); ok {
        fmt.Println(hp.name, "stop()")
        return
    }
    fmt.Print("Unknown printer.")
```

```
    switch v:=printer.(type) {
    case HP_Printer:
        fmt.Println(v.name,"stop()")
    default:
        fmt.Println("Unknown printer.")
    }
}
```

switch fallthrough break

```
func customHTTPErrorHandler(err error, c echo.Context) {
    code := http.StatusInternalServerError
    if he, ok := err.(*echo.HTTPError); ok {
        code = he.Code
    }
    fmt.Println("code==>", code)

    var errorPage string

    switch code {
    case 404:
        fallthrough  //switch人第一个expr为true的case开始执行，如果case带有fallthrough，程序会继续执行下一条case,不会再判断下一条case的expr,如果之后的case都有fallthrough,default出会被执行。
        //默认是break，但如果使用falthrough，就会强制执行下一个case的语句块，同时注意并不需要下一case表达式是成立的。
    case 500:
        errorPage = fmt.Sprintf("static/html/exception/%d.html", code) //采用static相对路径，而非/static绝对路径要从当前路径开始
    default:
        errorPage = fmt.Sprintf("static/html/exception/unknown.html") //其余的都做不识别的请求处理
    }
    if err := c.File(errorPage); err != nil {
        c.Logger().Error(err)
    }
    c.Logger().Error(err)
}
```

golang switch fallthrough

跳转语句 goto break continue

- goto 让程序调到某一标签处执行

- break 终止该层级的循环或终止指定标签处的循环
- continue 跳过该次循环或跳过指定标签处（适合有限循环)
- break和continue配合标签可用于多层循环的跳出
- goto是调整程序执行位置，与其他2个语句配合标签的结果不同

```
package main

func main() {
LABEL:
    for {
        for i := 0; i < 10; i++ {
            if i >= 3 {
                break LABEL
            }
            println(i)
        }
    }
}
```

break变为goto则为死循环

```
package main

func main() {
LABEL:
    for {
        for i := 0; i < 10; i++ {
            if i >= 3 {
                goto LABEL
            }
            println(i)
        }
    }
}
```

将LABEL的顺序移动到无限for循环后面，则可以避免死循环。

```
package main

func main() {
    for {
        for i := 0; i < 10; i++ {
```

```
            if i >= 3 {
                goto LABEL
            }
            println(i)
        }
    }
LABEL:
}
```

continue的使用

```
package main

func main() {
LABEL:
    for {
        for i := 0; i < 10; i++ {
            if i >= 3 {
                continue LABEL
            }
            println(i)
        }
    }
}
```

案例
```go
package main

func main() {
LABEL:
for i := 0; i < 10; i++ {
for {
println(i)
continue LABEL //执行指定标签层级的下次循环
//goto LABEL
}
}
}

// continue 和 goto的执行结果一样吗?
```

goto 死循环打印0的状态
而continue则是打印0-9后退出

递增、递减语句

go中的++、--，与其他语言不同的是，递增或递减语句在go中**只是作为语句，而非表达式。不能将递增或递减语句放在等号右边**。避免出现容易混淆易错的执行结果。使得go的递增或递减很纯粹，只做一件事情。

`a := b++ // 编译不通过` ×
`a++ //只能单独作为一行语句` √

go异常处理

- Go没有异常机制 try catch，但有panic/revover模式来处理错误
- Panic可以在任何地方引发，但recover只有在defer调用的函数才有效
- 发生panic，程序会立即停止

```
package main

import "fmt"

func main() {
    A()
    B()
    C()
}

func A() {
    fmt.Println("func A()")
}

func B() {
    panic("panic in B()")
}

func C() {
    fmt.Println("func C()")
}
/*
func A()
panic: panic in B()
```

```
goroutine 1 [running]:
main.B()
    /Users/fqc/work/src/run.go:16 +0x64
main.main()
    /Users/fqc/work/src/run.go:7 +
*/
```

```go
package main

import "fmt"

func main() {
    A()
    B()
    C()
}

func A() {
    fmt.Println("func A()")
}

func B() {
    defer func() {
        if err := recover(); err != nil {
            fmt.Println("recover in B()")
        }
    }()
    panic("panic in B()") // 需要在panic之前注册defer
}

func C() {
    fmt.Println("func C()")
}

// func A()
// recover in B()
// func C()
```

panic是来自被调用函数的信息，表示发生了某个已知的bug，一个设计处理良好的函数是永远不应该发生panic异常。