# **Go 语言过程式编程**

## **实验简介**

这节讲解Go语言的各种语句，控制结构，以及如何使用这些语句进行过程式编程，课程的内容比较多，但是都比较重要，在编写Go程序的过程中会频繁使用到。

## **一、实验说明**

本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境:

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

## **二. Go语言语句基础**

之所以先学习过程式编程，是因为在Go语言中面向对象编程也是建立在面向过程的基础上的。形式上讲，Go语言需要使用分号(;) 来作为上下文语句的分隔结束符。实际上在前面的代码中我们可以看到在Go语言中很少使用分号，那是因为编译器会自动在需要分号的地方加上分号。但是有两个地方必须使用分号，第一个是需要在一个行中放入一条或多条语句时，或者是在使用原始的 for 循环时。Go语言也支持多重赋值，如a, b = b, a。另外在之前的课程中我们提到过快速声明操作符:=，它的作用是同时在一个语句中声明和赋值一个变量。当:=操作符用于多个逗号分隔的变量时，如果该变量已经存在，则只是简单的修改它的值。但是当:=操作符位于作用域的起始处时，Go语言会创建一个新的变量，不管该变量之前是否存在，如在if或者for语句中。下面有一个例子可以说明:

a, b, c := 2, 3, 5 for a := 7; a < 8; a++ {

fmt.Println(a)

}

以上代码中，先使用:=声明并赋值了三个变量，Go会自动推导出变量的类型。然后再for语句处又一次使用:=操作符声明了变量a。需要注意的地方是，for语句代表了一个新的作用域，所以:=在这里新声明创建了一个变量a，这个变量和之前的变量a是完全不同的两个变量（内存地址不一样），所以是一个影子变量，因为覆盖了外部的同名变量。这是需要注意的一个地方。

### **1. 类型转换**

Go语言提供了一种在不同但相互兼容的类型之间相互转换的方式，这种转换非常有用并且是安全的。但是需要注意的是在数值之间进行转换可能造成其他问题，如精度丢失或者错误的结果。以下是类型转换的语法:

* resultOfType := Type(expression)

几个例子：

x := int16(2345) // 声明一个类型为int16的整数，其值为2345

y := int32(x) // 将int16类型的整数转换为int32类型

a := uint16(6500) // 声明一个类型为uint16类型的整数

b := int16(x) // 转换为int16类型，虽然能转换成功，但是由于6500超过in16类型的范围，会导致结果错误，b的值为 -536

另外在Go语言中可以通过type关键字声明类型，如type StringsSlice []string 将[]string（string类型的切片）声明为StringSlice类型。

### **2. 类型断言**

说到类型断言就需要先了解下Go语言中的接口。在Go语言中接口是一个自定义类型。它声明了一个或者多个方法。任何实现了这些方法的对象（类型）都满足这个接口。接口是完全抽象的，不能实例化。interface{}类型表示一个空接口，任何类型都满足空接口。也就是说interface{}类型的值可以用于表示任意Go语言类型的值。这里的空接口有点类似于python语言中的object实例。既然interface{} 可以用于表示任意类型，那有的时候我们需要将interface{}类型转换为我们需要的类型，这个操作类型断言。****一般情况下只有我们希望表达式是某种特定类型的值时才使用类型断言****。Go语言中可以使用以下语法：

* resultOfType, boolean := expression.(Type) // 安全的类型断言
* resultOfType := expression.(Type) // 非安全的类型断言，失败时程序会产生异常

使用VIM创建源文件type\_t.go，输入以下源文件：

package main

import (

"fmt")

func main() {

x := uint16(65000)

y := int16(x) // 将 x转换为int16类型

fmt.Printf("type and value of x is: %T and %d\n", x, x) // %T 格式化指令的作用是输出变量的类型

fmt.Printf("type and value of y is: %T and %d\n", y, y)

var i interface{} = 99 // 创建一个interface{}类型，其值为99

var s interface{} = []string{"left", "right"}

j := i.(int) // 我们假设i是兼容int类型，并使用类型断言将其转换为int类型

fmt.Printf("type and value of j is: %T and %d\n", j, j)

if s, ok := s.([]string); ok { // 创建了影子变量，if的作用域中覆盖了外部的变量s

fmt.Printf("%T -> %q\n", s, s)

}

}

运行程序：

$ go run type\_t.go

type and value of x is: uint16 and 65000

type and value of y is: int16 and -536

type and value of j is: int and 99

[]string -> ["left" "right"]

## **三. 分支和for语句**

Go语言提供了3种分支，即if、switch、select, 其中select用于监听channel（通道）在讲解通道的时候再详细介绍。

### **1. if 分支**

语法:

if optionalStatement1; booleanExpression1 {

block1

} else if optionalStatement2; booleanExpression2 {

block2

} else {

block3

}

其中optionalStatement是可选的表达式，真正决定分支走向的是booleanExpression1的值。

### **2. switch分支**

Go语言中switch分支既可用于常用的分支就象C语言中的switch一样，也可以用于类型开关，所谓类型开关就是用于判断变量属于什么类型。但是需要注意的是Go语言的switch语句不会自动贯穿，相反，如果想要贯穿需要添加fallthrough语句。表达式开关switch的语法如下：

switch optionalStatement; optionalExpression {

case expression1: block1

...

case expressionN: blockN

default: blockD

}

下面是个例子：

switch { // 没有表达式，默认为True值，匹配分支中值为True的分支

case value < minimum:

return minimum

case value > maximum:

return maximum

default:

return value

}

在上面的例子中，switch后面没有默认的表达式，这个时候Go语言默认其值为True。 在前面我们提到过类型断言，如果我们知道变量的类型就可以使用类型断言，但是当我们知道类型可能是许多类型中的一种时候，我们就可以使用类型开关。其语法如下：

switch optionalStatement; typeSwitchGuard {

case type1: block1

...

case typeN: blockN

default: blockD

}

### **3. for循环语句**

在前面的代码中我们已经遇到很多遍for语句了，它可以遍历数组，切片，映射等类型，也可以用于无限循环。以下是其语法：

for { // 无限循环

block

}

for booleanExpression { // while循环，在Go语言中没有while关键字

}

for index, char := range aString { // 迭代字符串

}

for item := range aChannel { // 迭代通道

}

说了这么多，让我们进行下练习，创建源文件switch\_t.go，输入以下代码：

package main

import (

"fmt")

func classchecker(items ...interface{}) { // 创建一个函数，该函数可以接受任意多的任意类型的参数

for i, x := range items {

switch x := x.(type) { // 创建了影子变量

case bool:

fmt.Printf("param #%d is a bool, value: %t\n", i, x)

case float64:

fmt.Printf("param #%d is a float64, value: %f\n", i, x)

case int, int8, int16, int32, int64:

fmt.Printf("param #%d is a int, value: %d\n", i, x)

case uint, uint8, uint16, uint32, uint64:

fmt.Printf("param #%d is a uint, value: %d\n", i, x)

case nil:

fmt.Printf("param #%d is a nil\n", i)

case string:

fmt.Printf("param #%d is a string, value: %s\n", i, x)

default:

fmt.Printf("param #%d's type is unknow\n", i)

}

}

}

func main() {

classchecker(5, -17.98, "AIDEN", nil, true, complex(1, 1))

}

以上代码中我们首先创建了一个接收任意数量任意类型参数的函数，然后使用for ... range aSlice的语法迭代了每一个在切片items中的元素，接着使用了switch类型开关判断了每一个参数的类型，并打印了其值和类型。程序运行输出如下：

$ go run switch\_t.go

param #0 is a int, value: 5

param #1 is a float64, value: -17.980000

param #2 is a string, value: AIDEN

param #3 is a nil

param #4 is a bool, value: true

param #5's type is unknow

## **四. 函数**

Go语言可以很方便的自定义函数，其中有特殊的函数main函数。main函数必须出现在main包里，且只能出现一次。当Go程序运行时候会自动调用main函数开始整个程序的执行。main函数不可接收任何参数，也不返回任何结果。 Go语言中函数的创建使用以下语法：

func functionName(optionalParameters) optionalReturnType {

block // func 函数名(参数列表) 单个返回值类型

}

func functionName(optionalParameters) (optionalReturnValues) {

block // func 函数名(参数列表) (返回值1 类型，返回值2 类型)

}

函数可以有任意多个参数，也可以有任意多个返回值，返回值可以是命名的。具体的看以下例子：

func func1(first int, rest ...int) int {

return first // func1 函数可以接收任多的int类型参数，并且返回一个int类型的值

}

func func2(first int, second string) (int, string) {

return first, second // func2 接收两个，函数，并且返回一个int和string类型的值

}

func func3(first int, second string) (a, b int) {

a, b := 1, 2 // func3 接收两个参数，并返回两个int类型的值，因为返回值是命名的，所以这里可以缩写

return

}

## **五. 通信(channel)和并发(goroutine)语句**

在本课程开篇中我们就介绍过Go语言强大的并发功能，这些功能都是建立在通信和并发语句上的。所谓goroutine是程序中与其他goroutine完全独立而并发执行的函数或者方法调用。每一个Go程序都至少有一个goroutine，其中main()函数所在的goroutine是主goroutine。goroutine很像轻量级的线程，它们可以被大批量的创建。那goroutine之间怎么进行通信呢？Go语言中推荐的做法是使用channel(通道)。channel是一个双向的或者单向的通信管道，可以用于两个或者多个goroutine之间进行通信（即接收和发送）数据。

### **1. 语法**

goroutine使用以下的go语句进行创建:

* ****go**** function(arguments)
* ****go**** func(parameters) { block } (arguments)

第二种方式中，我们是创建了一个临时的匿名函数，并马上在goroutine中执行。 当调用用go关键字执行函数时，函数会在另一个goroutine上马上执行，并且当前的goroutine的执行会从下一条语句马上恢复。因此执行一个go语句之后，当前程序中至少有两个goroutine在运行。

在大多数情况下，goroutine之间需要相互协作，最好的方式是通过channel来交换数据。使用下面语法创建channel（通道）：

* make(chan Type)
* make(chan Type, capacity)

语法中的Type指明了通道能发送的数据类型。其中第一种语法中创建了一个同步的通道，一次只能发送一项数据，它会阻塞直到发送者准备好发送和接收者准备好接收。如果给定了capacity也就是缓冲区容量，在缓冲区容量未满之前通道都是异步无阻塞的。通道支持的操作如下:

| **语法** | **含义** |
| --- | --- |
| channel <- value | 发送value到通道中，有可能阻塞 |
| <-channel | 从通道中接收数据 |
| x := <-channel | 接收数据并赋值给x |
| x, ok := <-channel | 功能同上，同时检查通道是否已关闭或者是否为空 |

### **2. select语句**

在前面的课程中我们提到过select语句，用于监听通道。其语法如下：

select {

case sendOrReceviae1: block1

...

case sendOrReceiveN: blockN

default: blockD

}

Go语言会从头至尾的判断每一个case中的发送和接收语句。如果其中任何一条语句可以执行（即没有被阻塞），那就从那些可执行的语句中任意选择一条来使用。如果所有的通道都被阻塞，那可能有两种情况。第一种，如果有default语句，那就会执行default 语句，同时程序的执行会从select语句恢复。第二种，如果没有default语句，则select语句会一直阻塞，直到有一个通道可用

下面让我们使用以上的相关知识进行下练习，使用VIM创建源文件goroutine\_channel\_t.go，输入如下源代码：

package main

import (

"fmt"

"math/rand")

func main() {

channels := make([]chan bool, 6) // 创建一个类型为chan bool的切片，每一项是能发送bool值的通道

for i := range channels { // 通过`range`初始化切片

channels[i] = make(chan bool)

}

go func() { // 在其他gouroutine中执行匿名函数

for {

channels[rand.Intn(6)] <- true // rand.Intn(n int)的用途是产生一个不大于n的随机数

} // 发送数据到随机出现的通道

}()

for i := 0; i < 36; i++ {

var x int

select { // select 语句当监听到哪个分支的同道未阻塞时就跳转到哪个分支

case <-channels[0]:

x = 1

case <-channels[1]:

x = 2

case <-channels[2]:

x = 3

case <-channels[3]:

x = 4

case <-channels[4]:

x = 5

case <-channels[5]:

x = 6

}

fmt.Printf("%d ", x)

}

fmt.Println()

}

通过以上注释可以很清晰的看到整个代码的执行流程，下面我们执行代码:

$ go run goroutine\_channel\_t.go6 4 6 6 2 1 2 3 5 1 3 2 1 6 5 3 4 6 6 3 6 1 3 5 4 2 2 5 1 4 2 1 6 6 4 3

## **六. defer, panic和recover**

### **1. defer**

开发程序时，有的时候忘记关闭打开的文件导致程序执行失败，在python中可以很方便的使用with语句对这些资源进行自动管理。在Go中我们可以使用defer语句完成这项任务。defer语句用于延迟执行一个函数或者方法或者是当前创建的匿名函数，它会在外部函数或者方法返回之前但是其返回值计算之后执行。这样就可能在一个延迟执行的函数中修改函数的命名返回值。如果一个函数中又多个defer语句，它们会以后进先出的顺序执行。defer最常用的地方就是保证一个使用完成后的文件正常关闭。如下例子:

var file \*os.Filevar err errorif file, err = os.Open(filename); err != ni {

do\_something(file)

return

}

defer file.Close()

### **2. panic和recover**

panic类似于其他程序中的异常，而recover 则用于恢复异常。当panic()函数被调用时，外围函数或者方法的执行会立即终止。然后任何延迟执行的函数都会被调用。这个过程一直在调用栈中层层发生，最后到达main函数，这个时候整个程序会终止，最终将最初的调用栈信息输出到stderr。但是当延迟执行函数中包含recover语句时，recover会捕捉到panic引发的异常，并停止panic的传播，这个时候我们能够以任何我们想用的方式处理panic。

Go语言将错误和异常两者区分对待。错误是指有可能出错的东西，程序中已经包含处理这些错误的优雅逻辑。而异常则是指不可能发生的事情。例如，一个永远为true的条件在实际环境中却是false。Go语言推荐使用错误，而不使用异常。通常情况下，我们可以在recover中阻止panic的传播，并将recover()的返回值转换成错误。

使用VIM创建源文件panic\_t.go, 输入以下代码：

package main

import (

"fmt"

"math"

)

func ContvertIntToInt16(x int) int16 {

if math.MinInt16 <= x && x <= math.MaxInt16 {

return int16(x)

}

panic(fmt.Sprintf("%d is out of int16 range", x)) // 手动触发panic

}

func main() {

i := ContvertIntToInt16(655567)

fmt.Printf("%d", i)

}

上面代码中为了演示panic，代码中手动促发了panic()的执行，但是我们没有使用recover进行捕捉，这会导致整个程序执行失败，下面执行程序验证下：

$ go run panic\_t.go

panic: 655567 is out of int16 range

goroutine 16 [running]:

runtime.panic(0x96bc0, 0x208178180)

/usr/local/go/src/pkg/runtime/panic.c:279 +0xf5

main.ContvertIntToInt16(0xa00cf, 0x3ec8f)

/Users/aiden/Project/golang/panic\_t.go:13 +0x10f

main.main()

/Users/aiden/Project/golang/panic\_t.go:17 +0x26

goroutine 17 [runnable]:

runtime.MHeap\_Scavenger()

/usr/local/go/src/pkg/runtime/mheap.c:507

runtime.goexit()

/usr/local/go/src/pkg/runtime/proc.c:1445

goroutine 18 [runnable]:

bgsweep()

/usr/local/go/src/pkg/runtime/mgc0.c:1976

runtime.goexit()

/usr/local/go/src/pkg/runtime/proc.c:1445

goroutine 19 [runnable]:

runfinq()

/usr/local/go/src/pkg/runtime/mgc0.c:2606

runtime.goexit()

/usr/local/go/src/pkg/runtime/proc.c:1445exit status 2

可以看到没有捕捉panic时，整个程序退出，并且打印出了调用栈的异常信息。 下面我们使用Go语言推荐的做法捕捉panic并将panic转换为error, 创建源文件panic\_t1.go，输入以下代码：

package main

import (

"fmt"

"math"

)

func ContvertIntToInt16(x int) int16 {

if math.MinInt16 <= x && x <= math.MaxInt16 {

return int16(x)

}

panic(fmt.Sprintf("%d is out of int16 range", x)) // 手动触发panic

}

func Int16FromInt(x int) (i int16, err error) {

defer func() { // 延迟执行匿名函数，并使用recover捕捉了panic，并将panic转换为了error

if e := recover(); e != nil {

err = fmt.Errorf("%v", e)

}

}()

i = ContvertIntToInt16(x)

return i, nil

}

func main() {

if \_, e := Int16FromInt(655567); e != nil {

fmt.Printf("%v\n", e)

} else {

fmt.Printf("no errors\n")

}

}

以上代码中，我们通过recover捕捉了异常，现在程序将异常转换成了错误，所以程序不会异常退出，执行验证如下：

$ go run panic\_t1.go655567 is out of int16 range

值得注意的地方是，在以上代码中的Int16FromInt(x int) (i int16, err error)函数中，我们在defer语句的匿名函数中修改了命名的返回值err。该函数在被调用时，Go语言会自动的将其返回值设置为对应类型的零值，在Int16FromInt函数中，i被初始化为0，err被初始化为nil。当在defer语句中匿名函数执行时候，recover如果捕捉到异常，然后修改了命名返回值err，并保持i的值（零值 )不变。如果没有捕捉到异常，则程序正常返回i和nil。

## **作业**

请使用本节所讲知识编写一个函数，实现输入一个正整数n，输出2-n之间的所有素数。