Golang 入门笔记 -10- 函数高级特性

作者：zyk
原文链接：https://ld246.com/article/1604497659285
来源网站：链滴

传递变长参数

如果函数最后一个参数采用 ...type 的形式，那么这个函数就可以处理一个变长参数（长度可以为 0）这样的函数被称为变参函数，如：

```
func myFunc(a int, args ...int)
```

如果参数存储在切片 arr 中，可以用 arr... 来传递参数，如：

```
package main

import "fmt"

func main() {
    x := Min(1, 3, 2, 0)
    fmt.Printf("The minimum is: %d\n", x)

    arr := []int{7, 9, 3, 5, 1}
    x = Min(arr...)
    fmt.Printf("The minimum in the arr is: %d", x)
}

func Min(a ...int) int {
    if len(a) == 0 {
        return 0
    }

    min := a[0]
    for _, v := range a {
        if v < min {
          min = v
        }
    }

    return min
}
```

上述代码运行结果为：

```
The minimum is: 0
The minimum in the arr is: 1
```

如果一个变长参数的类型未知，我们可以使用空接口 interface{} 来接收，然后再对参数的类型进行断：

```
func typeCheck(values ...interface{}) {
    for _, value := range values {
        switch v := value.(type) {
            case int:
                // ...
            case string:
                // ...
            case bool:
```

```
            // ...
        default:
            // ...
    }
  }
}
```

defer 和追踪

defer 函数允许我们在函数返回之前（return 语句执行之后）能执行某些语句或函数，有点类似于 Jav 的 finally 语句，一般用于释放一些资源，比如关闭文件等。

多个 defer 语句的执行顺序

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("defer begin")
    // 将 defer 放入延迟调用栈
    defer fmt.Println(1)
    defer fmt.Println(2)
    // 最后一个放入, 位于栈顶, 最先调用
    defer fmt.Println(3)
    fmt.Println("defer end")
}
```

上述代码运行结果为：

```
defer begin
defer end
3
2
1
```

结果分析：

- defer 是延迟调用，defer 后面的语句在正常语句执行完之后才会执行。
- 多个 defer 语句执行顺序和代码顺序**相反**（后进先出）。

使用 defer 语句释放资源

使用 defer 并发解锁

我们来看一个不使用 defer 来解决 map 在高并发下线程不安全的例子：

由于 map 默认不是并发安全的，所以需要一个 sync.Mutex 互斥量来保护 map 的访问。

```
var (
    // 一个演示用的字典
```

```
    valueByKey = make(map[string]int)
    // 保证使用字典时的并发安全的互斥锁
    valueByKeyGuard sync.Mutex
)
// 根据键读取值
func readValue(key string) int {
    // 对共享资源加锁
    valueByKeyGuard.Lock()
    // 取值
    v := valueByKey[key]
    // 对共享资源解锁
    valueByKeyGuard.Unlock()
    // 返回值
    return v
}
```

可以使用 defer 对上述代码进行优化：

```
var (
    valueByKey = make(map[string]int)
    valueByKeyGuard sync.Mutex
)
// 根据键读取值
func readValue(key string) int {
    valueByKeyGuard.Lock()

    // defer 后面的语句不会马上调用, 而是延迟到函数结束时调用
    defer valueByKeyGuard.Unlock()

    return valueByKey[key]
}
```

使用 defer 释放文件句柄

文件操作需要经过打开文件、获取和操作资源、关闭资源几个过程。若在操作完毕后不关闭资源，进将一直无法释放文件资源。以下的例子，会实现打开文件、获取文件和关闭文件等操作：

```
// 根据文件名查询其大小
func fileSize(filename string) int64 {
    // 根据文件名打开文件, 返回文件句柄和错误
    f, err := os.Open(filename)
    // 如果打开时发生错误, 返回文件大小为 0
    if err != nil {
        return 0
    }
    // 取文件状态信息
    info, err := f.Stat()

    // 如果获取信息时发生错误, 关闭文件并返回文件大小为 0
    if err != nil {
        f.Close()
        return 0
    }
    // 取文件大小
```

```
    size := info.Size()
    // 关闭文件
    f.Close()

    // 返回文件大小
    return size
}
```

可以用 defer 对代码进行简化：

```
func fileSize(filename string) int64 {
    f, err := os.Open(filename)
    if err != nil {
        return 0
    }

    //  延迟调用 Close, 此时 Close 不会被调用
    defer f.Close()

    info, err := f.Stat()
    if err != nil {
    // defer 机制触发, 调用 Close 关闭文件
        return 0
    }

    size := info.Size()

    // defer 机制触发, 调用 Close 关闭文件
    return size
}
```

函数作为参数

函数可以作为参数进行传递，被其他函数调用，我们来看一个例子：

```
package main

import (
    "fmt"
)

func main() {
    callback(1, Add)
}

func Add(a, b int) {
    fmt.Printf("The sum of %d and %d is: %d\n", a, b, a+b)
}

func callback(y int, f func(int, int)) {
    f(y, 2) // this becomes Add(1, 2)
}
```

输出结果为：

```
The sum of 1 and 2 is: 3
```

递归函数

一个函数在其函数体内调用自身就是递归，最经典的例子就是斐波那列数列（每个数均为前两个数之）：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...
```

我们用 Go 语言来实现该算法：

```
package main

import "fmt"

func fibonacci(n int) int {
    if n <= 2 {
        return 1
    }
    return fibonacci(n-1) + fibonacci(n-2)
}

func main() {
    for i := 0; i <= 10; i++ {
        fmt.Printf("fibonacci(%d) is: %d\n", i, fibonacci(i))
    }
}
```

上述代码运行结果为：

```
fibonacci(1) is: 1
fibonacci(2) is: 1
fibonacci(3) is: 2
fibonacci(4) is: 3
fibonacci(5) is: 5
fibonacci(6) is: 8
fibonacci(7) is: 13
fibonacci(8) is: 21
fibonacci(9) is: 34
fibonacci(10) is: 55
```

闭包

闭包其实就是匿名函数。匿名函数，顾名思义就是没有名称的函数，通常定义格式为：

```
func(参数列表)(返回参数列表){
    函数体
}
```

我们来看一个例子，如何构建一个匿名函数并调用：

```
package main
```

```
import "fmt"

func main() {
    f()
}

func f() {
    g := func(i int) { // 创建一个匿名函数，并赋值给变量 g
        fmt.Printf("%d ", i)
    }

    for i := 0; i < 4; i++ {
        g(i) // 调用匿名函数
        fmt.Printf(" - g is of type %T and has value %v\n", g, g)
    }
}
```

上述代码运行结果为：

```
0  - g is of type func(int) and has value 0x1056b20
1  - g is of type func(int) and has value 0x1056b20
2  - g is of type func(int) and has value 0x1056b20
3  - g is of type func(int) and has value 0x1056b20
```

通过以上示例我们可以了解到 g 的类型是 func(int)，它的值是内存地址。

现在有两个函数 Add2() 和 Adder()，它们都返回类型为 func(b int) int 的函数：

```
func Add2() (func(b int) int)
func Adder(a int) (func(b int) int)
```

Add2() 不接受参数，而 Adder 接收一个 int 类型的参数。

我们通过闭包来调用这两个函数：

```
package main

import "fmt"

func main() {
    // make an Add2 function, give it a name p2, and call it:
    p2 := Add2()
    fmt.Printf("Call Add2 for 3 gives: %v\n", p2(3))
    // make a special Adder function, a gets value 3:
    TwoAdder := Adder(2)
    fmt.Printf("The result is: %v\n", TwoAdder(3))
}

func Add2() func(b int) int {
    return func(b int) int {
        return b + 2
    }
}

func Adder(a int) func(b int) int {
```

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

上述代码运行结果为：

```
Call Add2 for 3 gives: 5
The result is: 5
```

我们再来看一个例子：

```
package main

import "fmt"

func main() {
    var f = Adder()
    fmt.Print(f(1), " - ")
    fmt.Print(f(20), " - ")
    fmt.Print(f(300))
}

func Adder() func(int) int {
    var x int
    return func(delta int) int {
        x += delta
        return x
    }
}
```

上述代码运行结果为：

```
1 - 21 - 321
```

我们发现在多次调用 f() 函数时，该函数内部的变量 x 的值被保留了，x 的初始值为 0，第一次调用：0 + 1，第二次调用 1 + 20，第三次 21 + 300。可以得出结论：闭包函数会保存并积累其中变量的值不管外部函数是否退出）。

将上述的例子稍加改动：

```
package main

import "fmt"

func main() {
    f1 := Adder()
    f2 := Adder()

    fmt.Print(f1(1), " - ")
    fmt.Print(f1(20), " - ")
    fmt.Print(f1(300), "\n")

    fmt.Print(f2(1), " - ")
    fmt.Print(f2(10), " - ")
```

```
    fmt.Print(f2(100))
}

func Adder() func(int) int {
    var x int
    return func(delta int) int {
        x += delta
        return x
    }
}
```

上述代码运行结果为：

```
1 - 21 - 321
1 - 11 - 111
```

我们发现创建的两个变量 f1 和 f2 是相互**隔离**的，调用 f1 函数只会在 f1 自身环境下保留变量，f2 也同理，可见 f1 和 f2 引用了两个不同环境，互不干扰。

注意：

- 闭包=函数 + 引用环境。
- 闭包可以缩小变量作用域，减少对全局变量的污染。