第5章 函数与错误处理

程序设计中有两种最基本的模式：面向过程编程和面向对象编程。面向过程的编程也叫顺序编程或结构化编程，而面向对象的编程没有严格的顺序关系，以事件驱动为基础，以类为程序单位。面向对象的编程语言主要有 Java，C++，C#，python 等。

严格来说，Go 语言是属于面向过程的编程语言。Go 语言作为一种结构化的编程语言，以模块作为程序单位，每一个模块独立完成特定任务，模块之间相互独立，只有调用与被调用的关系，这样的模块叫作函数。

Go 语言程序就是由函数构成的：由一个到多个函数构成包，由一个到多个包构成一个 Go 语言程序或项目。

Go 语言函数包括标准库函数、内置函数、第三方函数、用户自定义函数等。前面各章节用到的打印函数 fmt. Printf、扫描函数 fmt. Scanf 等都是标准库函数。标准库函数需要用 import 关键字引入相应的标准包才能使用。

而前面章节中用到的 make、append、len 等函数称为内置函数，内置函数不需要引入，可以直接使用。

Go 语言作为开源项目，有大量的第三方包及海量的开源函数可以供下载，下载安装后需要的时候用 import 引入即可。

Go 语言原则上规定，所有函数必须以一条 return 语句结束，return 语句可以带 0 个到多个返回值。

本章主要讲用户自定义函数，下面从函数的声明开始介绍。

5.1 函数的声明


视频讲解

5.1.1 内部函数的声明

内部函数是指函数的作用域是在包内或整个 Go 语言程序内（在各个包内可见）的函数。内部函数与变量一样，必须先声明后使用，函数声明的语法格式如下：

```
func 函数名称(参数列表)(返回值列表){
    函数体
}
```

Go 语言的函数由两大部分组成：函数头和函数体。函数头包括关键字、函数名和函数签名构成；函数体由一对大括号和若干 Go 语句构成。下面分别说明。

func：为函数声明关键字，不能省略，必须为小写，不允许首字母大写。

函数名称：函数名称可以是任何合法的 Go 语言标识符，其命名规则与变量名一样，首字母小写，其作用域为本包内，首字母大写则包外可见，可以被其他包引用。函数名称是可以省略的，没有名称的函数称为匿名函数或闭包。

参数列表：为传入函数的所有参数，称为形参，必须用小括号括起来，参数的数量可以是 0 个到多个，即使没有参数小括号也不能省略。多个参数可以不同类型，参数之间必须用逗号隔开。

返回值列表：为函数调用后的返回值列表。Go 语言函数支持多返回值，因此采用列表的形式。多个返回值需要用逗号隔开，用小括号括起来。函数可以没有返回值，这时小括号可以省略。如果只有一个返回值，也可以省略小括号，仅写上返回值类型即可。

参数列表与返回值列表一起并称为“函数签名”。需要注意的是函数名和函数签名不是同一个概念。

函数体：函数体由一对大括号和若干 Go 语句构成，左大括号必须紧接函数签名，与函数头处于同一行，右大括号必须是函数体的最后一行，建议单独成行。函数体内不允许再声明命名函数，但可以声明匿名函数(称为闭包)。Go 语言函数不支持嵌套、重载及默认参数。

5.1.2 匿名函数的声明

所谓匿名函数就是没有名字的函数，采用以下格式声明：

```
func (参数列表)(返回值列表){
      函数体
}
```

匿名函数的声明除了没有函数名称之外，其他的与命名函数完全相同。对于命名函数，如果有返回值，它可以作为一个变量来参与表达式运算，完成函数调用，将运算结果赋给某一个变量；如果没有返回值，函数可以作为一个语句运行，完成调用。

但是，匿名函数由于没有名字，无法按名字调用，或者作为单独语句运行，必须在定义的时候就赋值给一个变量，然后以这个变量来代替匿名函数被调用；或者在其定义的右大括号外加上一对小括号，表示对该匿名函数的调用。例如：

```
Add := func (x,y int)int{
            return x + y
          }
sum := Add(3,4)     // sum = 7
```

上述 Add 就是一个匿名函数变量，可以替代匿名函数被调用。匿名函数也可以自执行，而不必借助某个变量，例如：

```
Add := func (x,y int)int{
            return x + y
          }(3,4)          // Add = 7
```

这时，Add 不再是匿名函数变量，而是普通变量，其值为 7，而函数变量的值为函数。

5.1.3 外部函数的声明

Go 语言除了可以引用标准库函数，第三方库之外，还可引用一些用其他语言实现的函数，例如汇编语言函数或 C 语言函数等。如果要引用外部定义的函数，需要在 Go 语言程序

内部预先进行声明，声明格式如下：

```
func 函数名(参数列表)回值列类型
```

声明一个在外部实现的函数，只需要给出函数名及函数签名，不需要给出函数体，因而大括号及函数体是不需要的。例如：

```
func copmare(x int,y int)int
```

仅仅需要给出函数头，函数的具体实现在外部完成。

5.1.4 函数类型的声明

在 Go 语言中，函数属于一等值，也就是说函数可以当作变量来使用，可以给一个变量赋值。函数变量不同于基础数据类型变量，基础数据类型变量的类型由声明时指明的数据类型决定，而函数变量的类型就是函数，需要通过 type 关键字来声明，其语法格式如下：

```
type ftn func (参数列表)(返回值列表)
```

可见，函数类型由关键字 type、func 及函数签名构成，不需要函数体。函数类型变量可以像普通变量一样给其他变量赋值，不过其内容为函数，不是某个字面量。函数类型也可以作参数，无论实参还是形参，都可以是函数类型。

不同签名的函数类型不能相互比较，不能相互赋值。因此，函数签名就成了函数类型的关键特征。如果定义了一个函数类型的变量，给它赋值的只能是相同类型的函数，即签名相同的函数。

请认真阅读分析以下例子，体会函数类型的定义与作用。

下述程序的功能是实现求长方形、三角形和圆环的面积，只要输入长方形的两个边长或三角形的底和高以及圆环的外圆半径及内圆半径，就可以计算其面积。

```
package main
import "fmt"
type areatype func(float64, float64) float64     // 定义一个函数类型
func rectangle(l, w float64) float64 {           // 计算长方形面积
    return l * w
}
func triangle(b, h float64) float64 {            // 计算三角形面积
    return b * h / 2.0
}
func ring(R, r float64) float64 {                // 计算圆环面积
    return 3.1416 * (R * R - r * r)
}
func area(x, y float64, z areatype) float64 {    // 计算面积通用方法
    return z(x, y)
}
func main() {
    l, w := 22.3, 37.5
    rect := area(l, w, rectangle)                // 采用通用方法 area 求面积
    fmt.Printf("长方形面积为: %.2f\n", rect)
    b, h := 52.33, 23.5
```

```
    tri := area(b, h, triangle)                    // 采用通用方法 area 求面积
    fmt.Printf("三角形面积为: %.2f\n", tri)
    R, r := 23.63, 13.45
    rin := area(R, r, ring)                        // 采用通用方法 area 求面积
    fmt.Printf("圆环的面积为: %.2f\n", rin)
}
```

运行结果：

```
长方形面积为: 836.25
三角形面积为: 614.88
圆环的面积为: 1185.87
```

程序分析：

上述 area 函数的参数列表里有三个参数：两个浮点数 x,y 以及一个函数类型变量 z。前两个变量用来表示边长等图形参数，后一个函数变量用来赋值不同的函数，计算不同形状的面积。

这就是函数变量的典型应用，也是函数作为参数的一种方式，称为引用传递，后续内容会详细介绍。Go 语言的这种用法极像面向对象程序设计的方法重载。这也就是为什么说面向过程的 Go 语言也具备一定的面向对象的编程特性。

视频讲解

5.2 函数的参数

函数定义时的参数称为形式参数，简称“形参”；而函数调用时传入的参数为实质参数，简称“实参”。形参只在函数体内使用，其作用域为从函数头的形参列表处延伸到函数体结束。如果函数体内又嵌入了匿名函数，则形参在匿名函数内仍然可见。

5.2.1 参数列表的格式

函数的参数列表可以包含 0 个到多个变量。参数必须是变量，不可以是常量及常量表达式，也不可以是变量表达式。例如，NUM 为常量，x,y 为变量，以下格式都是非法的：

```
func fa(x int ,80)int{...}          // 常量不可作参数
func fa(y int ,80 + NUM)int{...}    // 常量表达式不可作参数
func fa(x int ,y + 80)int{...}      // 变量表达式不可作参数
func fa(x + y int )int{...}         // 变量表达式不可作参数
```

参数列表里只允许出现变量名及变量类型，多个变量之间用逗号分隔。所有参数最后必须用小括号括起来，0 个参数也不能省略小括号。

如果多个参数类型相同，就不必每个变量都声明类型，在最后一个变量后面声明即可，如下所示：

```
func fa(x,y,z int,a,b,c float64)int{...}
```

原则上要求所有参数命名，尽管不命名，仅保留类型也是合法的，如下所示：

```
func fa(int, float64)int{...}
```

这种情况下，输入的实参实际上没起作用，因为函数体内没有传入实参，上述函数等同

于无参数函数

```
func fa()int{...}
```

如果需要外部传入实参，则形参一定要命名。尽管参数是命名的，但是传递实参的时候还得按照形参的顺序传递，并不能因为命名了就可以打乱顺序，哪怕实参与形参同名，也必须按顺序传递。这就意味着参数传递只按顺序不按名字，与名字无关，只与位置有关，这就意味着实参和形参可以不同名。

函数的参数可以是基础数据类型、字符串、数组、切片、函数、自定义类型等。其中，基础数据类型、字符串、数组、结构体等类型都是值传递，变量直接将值的副本传入函数，不改变变量的原值，比较简单。而切片、映射、通道、指针等均为引用传递，传入的是地址。Go 语言中数组是按值传递的，传递的是数组的副本。小数组没什么问题，大型数组会消耗大量的内存资源，应避免使用值传递，可以使用地址传递。数组作参数分两种情况：数组元素作参数及整个数组作参数，在定义形参的时候，其类型不同，需谨慎，下面分别说明。

（1）数组元素作为参数。

如果以单个数组元素作为函数的参数，则在函数定义时其形参的数据类型必须与数组的元素类型相同。函数调用时，数组元素必须用元素变量的形式作实参，即用数组名称及方括号下标的形式，其中下标可以是常量，也可以是变量，甚至是表达式。

单个数组元素是不能作为函数形参的，只能以实参传入函数。单个数组元素作参数的应用比较少见，大多数情况下都是以整个数组作参数。

（2）以整个数组作为参数。

若以整个数组作为实参传入函数，则在函数声明时，其形参类型必须为相同的数组类型。作为数组类型，其长度和元素类型是其关键特征。只有长度和元素类型都一致的数组实参才可以被传入。请看以下的例子：

```
package main
import "fmt"
func fa(x [3]int) int {       // 形参 x 为长度等于 3 的 int 型数组
    return x[1] * 8
}
func main() {
    var s = [3]int{1, 2, 3}
    //    var t = [4]int{1, 2, 3, 4}
    h := fa(s)                // 实参 s 为长度等于 3 的 int 型数组
    // h := fa(t)             // 系统运行错误
    fmt.Println("结果 = ", h)
}
```

运行结果：

```
结果 = 16
```

从上述程序可以看出，实参数组必须与形参数组的长度和类型一致，才能被传入的函数接收，以使函数可以正确运行。

如果执行 fa(t)，系统会提示错误，并给出以下错误信息：cannot use t (type [4]int) as type [3]int in argument to fa。

很显然，Go 语言认为 type [3]int 与 type [4]int 不属于同一类型。因此，将数组作为参数的时候，数组是作为一种类型来定义的，也就是该参数类型为数组类型。我们把上述例子改成以下形式，大家可能更容易理解。

```
package main
import "fmt"
type stype [3]int          // 自定义一个类型为[3]int 的数组类型
func fa(x stype) int {     // 变量 x 为 stype 类型的数组
    return x[1] * 8
}
func main() {
    var s = stype{1, 2, 3}// 数组 s 满足 stype 类型的要求
    h := fa(s)
    fmt.Println("结果 h = ", h)
}
```

运行结果：

```
结果 h = 16
```

上述程序中直接定义一个类型 stype 来代替一个数组类型，程序接着就可以用该类型来声明数组变量及参数类型了。

5.2.2 可变参数

函数的参数个数事实上是可变的，由于参数不定，因此最后一个参数后面需要用省略号表示，省略号后面是参数类型。这就意味着所有的可变参数都与最后一个参数同类型。所以，可变参数一定要放在参数列表的最后，如下所示：

```
func fa(x int, y ...float64)int{block}
```

意味着变量 y 及以后所有变量都为 float64 类型。很显然，这就是一个 float64 类型的切片。因此，可变参数实质就是数据类型为 Type 的切片[]Type。既然参数可变，就意味着数量不确定，但函数调用执行的时候必须确定变量的具体数量，怎么做到这点呢？

前文已经指出了可变参数实质就是个切片，那么就可以使用关键字 range 遍历的方式准确获得全部参数。请参看以下的例子。

假设有以下可变参数函数，求所有参数值之和：

```
package main
import "fmt"
func fgh(x int, y ...int) int {
    sum := 0
    for _, value := range y {
        sum += value
    }
    return sum + x
}
func main() {
    var y = []int{1, 2, 3, 4, 5, 6}
    sum1 := fgh(10, y...)
```

```
    sum2 := fgh(1, 3, 6, 9, 12, 15)
    fmt.Println("sum1 = ", sum1)
    fmt.Println("直接输入多参数: sum2 = ", sum2)
}
```

运行结果：

```
sum1 = 31
直接输入多参数: sum2 = 46
```

函数 fgh 为可变参数函数，至少需要一个整型参数，可以有无数多个整型参数。上述程序使用了两种方法调用该函数，第一种采用一个整数切片 y 作为实参（用切片作实参的时候必须在切片名字后面加上省略号）；第二种直接使用多参数调用，参数数量可以多，也可以少，两种调用结果相同。

5.2.3 值传递和引用传递

函数调用的过程实质就是用实参代替形参，执行函数体的过程。实参代替形参有两种方式：值传递和引用传递。

值传递就是用实参值的副本赋给形参，形参以该副本值参与各种表达式运算，运算结果有可能影响形参，即实参的副本，对实参原值却没有影响。

引用传递就是以实参的地址传递给形参，因此要求形参必须是地址类型的变量，或指针变量。形参获得了实参的地址，访问该地址就可以获得实参的值。如果形参所在表达式修改了形参（指针）指向的值，则相当于修改了实参的值。因此，引用传递是有副作用的，对实参的原值会有影响。

Go 语言中默认基础数据类型、数组、字符串、结构体等都为值传递；切片、映射、通道、指针包括函数等为引用传递。Go 语言中字符串即使按值的方式传递，但是编译系统内部实际上还是按地址传递的，这点用户不用理会。因为，就算是地址传递，由于 Go 语言规定字符串是不可以更改的，因此传值和传地址，字符串内容都不变。Go 语言中数组也是按值传递的，如果是小数组，代价还不算大；如果是大数组则代价巨大。幸运的是，Go 语言中数组不常用，大多数情况下都可以用切片来代替，且情况良好。

函数作参数会有两种传递方式：如果是直接调用函数，利用其运行结果作为被调用函数的参数，则是值传递。对于值传递要求函数的返回值的类型和数量要严格满足被调用函数的形参要求，下文会用例子来说明这种应用。如果函数作为形参的类型，则是引用传递，传入的是可执行函数的首地址，如 5.2.2 小节函数类型声明中展示的例子。

结构体与函数类似，可以是按值传递，也可以是按引用传递。如果结构体的字段较少，是可以采用值传递的；如果是字段比较多的大型结构体，建议设计一个指针指向结构体，使用指针作参数，从而间接引用结构体，形成引用传递。这方面的例子等讲到结构体相关章节时再提供。下面举例说明两种参数传递方式的应用。

1. 值传递

（1）整型、浮点型、字符串型的传递。

以下程序声明了三个变量，分别代表整型、浮点型和字符串型。在被调用函数内部也声明同名同类型的三个变量，让实参与形参同名，在函数内对该三个变量进行修改，然后作为

返回值。在主函数处观察调用前后变量值的变化以及与返回值的关系。

```
package main
import "fmt"
func f1(x int, y float64, z string) (int, float64, string) {
    x = x + 10
    y = y + 10
    z = z + "abc"
    fmt.Printf("被调用函数内部变量 x = %d, y = %.2f, z = %s\n", x, y, z)
    return x, y, z
}
func main() {
    x, y, z := 1, 2.1, "student"
    fmt.Printf("函数调用前变量 x = %d, y = %.2f, z = %s\n", x, y, z)
    a, b, c := f1(x, y, z)
    fmt.Printf("函数调用后变量 x = %d, y = %.2f, z = %s\n", x, y, z)
    fmt.Printf("函数调用返回值 x = %d, y = %.2f, z = %s\n", a, b, c)
}
```

运行结果：

```
函数调用前变量 x = 1, y = 2.10, z = student
被调用函数内部变量 x = 11, y = 12.10, z = studentabc
函数调用后变量 x = 1, y = 2.10, z = student
函数调用返回值 x = 11, y = 12.10, z = studentabc
```

程序分析：

可以看出，程序的执行流程是主函数 main 调用子函数 f1。

子函数有三个形参，分别为整型变量 x、浮点型变量 y 及字符串型 z。

主函数中也定义了三个同名的变量并赋值。调用前及调用后主函数 main 定义的三个变量的值都没有发生变化。三个变量的值(副本)传入子函数后，在子函数内均被修改了，并将修改后的值作为返回值传给调用者。

主函数获得的返回值是变量值的副本被修改后的值，其原值保持不变，这就是值传递的本质。

(2) 数组的传递。

数组也是一种类型，其类型特征包含长度及元素类型。因此，形参的类型也必须是数组类型，才能将数组实参传递给数组类型的形参，参看以下程序。

```
package main
import "fmt"
func f1(x [3]int) {
    x[0] = x[0] + 10
    x[1] = x[1] + 10
    x[2] = x[2] + 10
    fmt.Printf("被调用函数内部变量 x = %d\n", x)
    return
}
func main() {
    x := [3]int{1, 2, 3}
```

```
    fmt.Printf("函数调用前变量 x = %d\n", x)
    f1(x)
    fmt.Printf("函数调用后变量 x = %d\n", x)
}
```

运行结果：

```
函数调用前变量 x = [1 2 3]
被调用函数内部变量 x = [11 12 13]
函数调用后变量 x = [1 2 3]
```

程序分析：

从程序的执行结果可以看出，数组也是按值传递的，调用前后并不改变数组元素的原值。

在被调用的子函数内修改的是数组元素值的副本，改变后的副本的值也可以用 return 返回。本例中没有返回值，只是演示子函数程序内修改数组的值，看是否能影响数组元素的原值。

（3）函数的传递。

函数作为值来传递是利用函数的返回值，本质上是以函数调用作为函数的参数。对于这种情况，被调用函数的形参列表与作为参数调用的函数的返回值列表必须完全相同。请参看以下例子：

```
package main
import "fmt"
func f1(x, y, z int) (sum, avg int) {
    sum = x + y + z
    avg = sum / 3
    return
}
func f2(a, b int) int {
    return a + 3 * b
}
func main() {
    x, y, z := 1, 2, 3
    q1 := f2(x, y)
    q2 := f2(f1(x, y, z))
    fmt.Printf("直接值传递调用 q1 = %d\n", q1)
    fmt.Printf("函数调用传递 q2 = %d\n", q2)
}
```

运行结果：

```
直接值传递调用 q1 = 7
函数调用传递 q2 = 12
```

程序分析：

上述程序中，子函数 f2 的形参为两个整型变量，直接使用两个整型实参就可以调用。如果用一个函数的返回值来代替两个整型的实参，则该函数的返回值列表必须是两个整型的返回值。

子函数 f1 的返回值列表正好是两个整型变量，满足 f2 的形参要求。因此，可以用函数

f1 的返回值来作为子函数 f2 的参数，实现一个函数(f1)的返回值作为另一个函数(f2)的参数，这就是函数的值传递方式。

(4) 结构体的传递。

结构体的知识还没讲解到，在这里为了函数内容叙述的完整性，先提供一个简单的例子，若是暂时看不懂也不要紧，待以后熟悉结构体相关知识后可以再回来了解。

```
package main
import "fmt"
type s struct {
    name   string
    height float64
    weight float64
}
func f1(x s) {
    x.name = "李四"
    x.height = 1.75
    x.weight = 78.96
    fmt.Printf("输入张三的参数,在子函数内修改成李四的参数!修改后的参数如下: \n")
    fmt.Printf("姓名: %s\n", x.name)
    fmt.Printf("身高: %.2f\n", x.height)
    fmt.Printf("体重: %.2f\n", x.weight)
    return
}
func main() {
    var q s
    q.name = "张三"
    q.height = 1.82
    q.weight = 85.85
    fmt.Printf("张三的参数原值是: \n")
    fmt.Printf("姓名: %s\n", q.name)
    fmt.Printf("身高: %.2f\n", q.height)
    fmt.Printf("体重: %.2f\n", q.weight)
    f1(q)
    fmt.Printf("张三的参数被调用修改后是: \n")
    fmt.Printf("姓名: %s\n", q.name)
    fmt.Printf("身高: %.2f\n", q.height)
    fmt.Printf("体重: %.2f\n", q.weight)
}
```

运行结果：

```
张三的参数原值是:
姓名: 张三
身高: 1.82
体重: 85.85
输入张三的参数,在子函数内修改成李四的参数!修改后的参数如下:
姓名: 李四
身高: 1.75
体重: 78.96
张三的参数被调用修改后是:
姓名: 张三
身高: 1.82
体重: 85.85
```

程序分析：

上述例子演示了结构体传递是按值传递的，张三的参数传入后被修改成了李四，但是调用返回后张三的参数并没有改变，说明子函数修改的是张三的副本，这就是值传递的原意。

程序中首先定义一个结构体类型 s，包含三个字段：姓名、身高和体重，分别是字符串型和浮点型。接着定义一个函数 f1，其参数 x 为结构体型 s，要调用这个函数，实参也必须是结构体型 s。

子函数的作用就是修改传入结构体的各字段值，用新的值覆盖原来的值，并打印输出，程序没有设计返回值。

主函数 main 里先定义一个结构体 s 类型的变量 q，然后给这个变量赋值。把张三的参数赋给结构体变量 q，然后打印原值，紧接着以结构体变量 q 为实参调用函数 f1。

子函数 f1 修改 q 的参数并打印。

主函数调用 f1 返回后再次打印 q 的参数，观察变化情况。

运行结果证明，q 的原值在调用前后均没发生变化，变化的是 q 的副本。这就是以值传递的结构体，下文中会再次演示以引用的方式传递结构体。

2. 引用传递

所谓引用传递，就是函数的形参类型为地址或指针，实参类型也必须为地址或者指针，这种地址传递就称为引用传递。

(1) 函数的传递。

函数类型变量的传递方式包括值传递和引用传递，值传递是以函数调用后的返回值作为实参，前面已经举例。引用传递是将函数作为实参传递给形参，要求形参的类型为函数类型，且传入的实参也必须是同类型的函数。

函数的引用传递程序示例，如下所示：

```
package main
import "fmt"
type ft func(int, int) int
func f2(a, b int) int {
    return a + b
}
func f1(x, y int, f ft) int {
    return f(x, y)
}
func main() {
    x, y := 1, 2
    q1 := f1(x, y, f2)
    fmt.Printf("函数类型传递  q1 = %d\n", q1)
}
```

运行结果：

```
函数类型传递 q1 = 3
```

程序分析：

上述程序首先定义一个函数类型，这个函数类型的函数签名是：

```
func (int,int)int
```

任何函数的具体实现，只要其函数签名与上述函数签名相同，就是同一类型函数，就可以相互比较或赋值。上述程序定义的函数 f2 就是上述类型的函数。

程序接着定义一个通用函数 f1，其形参包括两个整数型变量 x，y 以及一个函数型变量 f。函数变量用来传递 ft 类型的函数，本例中传入的是 f2 函数。

从运行结果来看，程序运行实现了预定的功能。

（2）结构体的传递。

从上文例子中我们已经看到，结构体是以值传递的，如果要引用传递结构体，则需要用到指针。有关指针的知识将在后续章节中介绍，这里大家先有个感性认识。

下面以前文中已演示的结构体值传递的例子为基础加以改造，来演示结构体的引用传递如何实现。我们先定义一个指针指向结构体，然后以指针为实参传入子函数，则子函数内的形参就被赋值为指针，在子函数中修改指针指向的字段值，并在子函数内显示修改后的结果，返回主函数后继续关注修改的结果是否仍然有效。程序如下所示：

```
package main
import "fmt"
type s struct {
    name   string
    height float64
    weight float64
}
func f1(p *s) {
    p.name = "李四"
    p.height = 1.75
    p.weight = 78.96
    fmt.Printf("输入张三的参数,在子函数内修改成李四的参数!修改后的参数如下: \n")
    fmt.Printf("姓名: %s\n", p.name)
    fmt.Printf("身高: %.2f\n", p.height)
    fmt.Printf("体重: %.2f\n", p.weight)
    return
}
func main() {
    q := s{"张三", 1.82, 85.85}
    p := &q
    fmt.Printf("张三的参数原值是: \n")
    fmt.Printf("姓名: %s\n", p.name)
    fmt.Printf("身高: %.2f\n", p.height)
    fmt.Printf("体重: %.2f\n", p.weight)
    f1(p)
    fmt.Printf("张三的参数被调用修改后是: \n")
    fmt.Printf("姓名: %s\n", p.name)
    fmt.Printf("身高: %.2f\n", p.height)
    fmt.Printf("体重: %.2f\n", p.weight)
}
```

运行结果：

```
张三的参数原值是:
姓名: 张三
```

```
身高: 1.82
体重: 85.85
输入张三的参数,在子函数内修改成李四的参数!修改后的参数如下:
姓名: 李四
身高: 1.75
体重: 78.96
张三的参数被调用修改后是:
姓名: 李四
身高: 1.75
体重: 78.96
```

程序分析:

在上述主函数 main 中先定义一个结构体变量实例 q,并赋初值。然后定义一个指针 p 指向 q,打印 p 指向的字段值,就是 q 的初值。

然后以 p 为实参,调用函数 f1,在 f1 内形参也用 p 表示。修改 p 指向的字段内容,打印显示修改后的内容,然后退出子函数,没有返回值。

回到主函数后再次打印 p 指向的值,发现 q 的内容被改变了。

说明引用传递,不传递具体的值,只传递值的地址,在子函数中利用该值的地址,修改值的内容,退出调用程序后,修改仍然有效。

这表明子函数中修改的不是值的副本,而是值本身,这就是值传递与引用传递的区别。

(3) 指针的传递。

在 Go 语言中数组是值传递的,对于大型数组的传递,由于需要生成副本,会消耗系统大量的资源。所以,C 语言等编程语言都是以数组名为引用来传递参数。

事实上,在 Go 语言里我们也可以使用指向数组的指针来传递数组,以实现类似 C 语言的功能。

我们知道,值传递时数组元素的原值是不变的,子函数中修改的只是数组元素的副本。

以下程序采用数组的引用传递,通过子函数来修改数组的原值,执行完毕后请查看结果,看是否能达到预期的目的。

```
package main
import "fmt"
func f1(p *[10]int) {
    for i, value := range p {
    p[i] = 2 * value
    }
    return
}
func main() {
    x := [10]int{12, 32, 3, 4, 5, 6, 7, 8, 23, 34}
    p := &x
    fmt.Printf("调用前数组的原值 x = %d\n", x)
    f1(p)
    fmt.Printf("调用后数组的现值 x = %d\n", x)
}
```

运行结果：

```
调用前数组的原值 x = [12 32 3 4 5 6 7 8 23 34]
调用后数组的现值 x = [24 64 6 8 10 12 14 16 46 68]
```


程序分析：

上述程序中定义了一个子函数 f1，它的参数是一个长度为 10 的 int 型数组指针，调用它时的实参也必须是指针，而且也必须是指向长度为 10 的 int 型数组，否则会导致类型不匹配。

在子函数中修改数组的值，让其每个元素的值加倍。返回主程序中再查询一下数组是否被改，以检验子程序的修改是否会影响数组原值。

从程序运行结果来看，达到了预期目的，数组的原值被修改成功，证明数组也是可以采用引用传递的。

(4) 切片的传递。

由于切片的灵活性，事实上，Go 语言中很少使用数组，完全可以使用切片来代替。切片原生就是按引用传递的，传递时对内存资源的占用极少。

对切片内容的操作也极为方便，可以使用 range 关键字遍历出全部切片内容及其索引值，然后根据需要进行操作，程序如下所示：

```
package main
import "fmt"
func f1(slice []int) (int, int) {
    sum := 0
    for _, value := range slice {
    sum += value
    }
    aver := sum / len(slice)
    return sum, aver
}
func main() {
    x := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    sum, aver := f1(x)
    fmt.Printf("切片元素之和为 sum = %d\n", sum)
    fmt.Printf("切片元素均值为 aver = %d\n", aver)
}
```

运行结果：

```
切片元素之和为 sum = 55
切片元素均值为 aver = 5
```

程序分析：

上述程序中先定义一个子函数 f1，其参数为 int 切片类型。因此，调用它的主函数的实参也必须为切片，而且是同类型的 int 切片。

子函数实现了对切片元素的求和及求平均值。

(5) 映射的传递。

Go 语言默认映射也是按引用传递的。引用传递的最大特征是对内存资源的消耗极少，而且不是值的副本操作，因而对原值有直接影响。

以下例子是映射的引用传递，通过子函数往映射里添加项目，在调用返回后再查看映射，检查添加是否成功。

```
package main
import "fmt"
func f1(mp map[int]string) {
    mp[100] = "张三"
    mp[200] = "李四"
    return
}
func main() {
    mq := map[int]string{1: "teacher", 2: "stdent", 3: "boy", 4: "girl"}
    fmt.Printf("调用前的映射为 %v\n", mq)
    f1(mq)
    fmt.Printf("调用后的映射为 %v\n", mq)
}
```

运行结果：

```
调用前的映射为 map[3: boy 4: girl 1: teacher 2: stdent]
调用后的映射为 map[1: teacher 2: stdent 3: boy 4: girl 100: 张三 200: 李四]
```

程序分析：

从程序执行结果来看，在子函数中添加的两个映射项目在主程序也能查到，说明子函数的添加操作成功，程序达到了预期的目的。

同时也进一步说明，引用传递是可以修改原值的。

Go 语言默认通道也是按引用传递的，有关内容我们放到第 10 章并发编程中介绍。

5.2.4 空接口作为参数

在前文的叙述中，函数的参数必须是有明确类型的，且类型要在参数列表中明确表示。事实上，Go 语言允许使用空接口 interface{}来表示任意类型。回忆一下我们多次使用的打印语句 fmt.Printf，其原型是这样的：

```
func Printf(format string, a ...interface{}) (n int, err error)
```

上式中，format 为字符串类型的格式串，变量 a 为空接口类型，可变参数都为空接口类型。按 Go 语言规定，任何类型都实现了空接口，因此可用空接口代表任何数据类型。也就意味着打印语句的参数可以是任意数据类型。

任何函数的参数都可以使用空接口，尤其是可变参数，这为函数的使用提供了极大的方便。从此，函数调用者再也不必刻意改变或转换实参的类型了，任何类型的实参都可以直接传入函数，函数也能正确执行。

有关空接口方面的知识我们在后续章节中再叙述。以空接口作为参数的例子包括打印语句和扫描语句等，前文已有多处使用，这里不再举例。


视频讲解

5.3 函数的返回值

函数作为一个独立执行单元，其执行结果可以保留在内存中，也可以通过返回变量传递给调用者，其返回值的格式可有多种。

5.3.1 返回值列表的格式

函数的返回值可以有 0 个到多个，形成返回值列表，但不允许有可变参数。返回值列表必须用小括号括起来，不可以用省略方式表示可变返回值。返回值的类型可以是基础数据类型，也可以是复合数据类型，甚至是函数类型。0 个或者只有一个返回值的时候，小括号可以省略(非命名返回值)。而两个以上的返回值必须用小括号括起来。每个返回值必须明确指定数据类型，各个返回值之间用逗号隔开。

返回值可以仅仅给出数据类型，各个数据类型之间用逗号隔开。也可以采用命名返回值，命名的返回值也必须指定类型。多个命名的相同类型的返回值可以连续写，最后写一个数据类型标识符即可。

所有返回值要么全部命名，要么全部不命名，不可以部分命名，部分不命名。命名的返回值位置是任意的，如果不命名，则 return 的返回值类型必须与返回值列表中的类型匹配，而且数量、位置必须一致。

当只有一个返回值的时候可以不命名而只写类型，省略小括号；也可以命名，但是不能省略小括号。

以下的返回值列表写法是合法的：

```
func Name(para_list){block}                          // 无返回值
func Name(para_list)int{block}                       // 一个返回值
func Name(para_list)(sum int){block}                 // 一个命名返回值
func Name(para_list)(x int,y int){block}             // 两个命名返回值
func Name(para_list)(x,y int){block}                 // 同类型,可缩写
func Name(para_list)(int,int,float){block}           // 返回值不命名
```

而下述的返回值列表写法是非法的：

```
func Name(para_list)(){block}                        // 没有返回值,不用小括号
func Name(para_list)sum int{block}                   // 命名返回值需要小括号
func Name(para_list)(sum int,float64){block}         // 不允许部分命名返回值
func Name(para_list)(sum int,x...string ){block}     // 不允许用省略号
func Name(para_list)(x,y,z){block}                   // 缺少类型说明
```

5.3.2 函数作为返回值

函数作返回值用到了匿名函数的概念，以函数作为返回值的函数通常被称为工厂函数，顾名思义，就是生产函数的函数。返回的函数必须是匿名函数，或者称为闭包。

以下程序示例的函数 f1 就称为工厂函数，它可以产生 4 个不同的函数，分别用于“求和”“求差”“求积”及“求均值”。

根据传入的字符串内容不同返回不同的函数。调用工厂函数返回一个匿名函数，并赋值给一个变量，这个变量称为对匿名函数的引用。

匿名函数也必须经由变量的引用才能被执行。

```
package main
import "fmt"
func f1(a string) func(x, y float64) float64 {
```

```
    if a == "求和" {
      return func(x, y float64) float64 {
         return x + y
      }
    } else if a == "求积" {
      return func(x, y float64) float64 {
         return x * y
      }
    } else if a == "求均值" {
      return func(x, y float64) float64 {
         return (x + y) / 2
      }
    }
    return func(x, y float64) float64 {
       return (x - y)
    }
}
func main() {
    add := f1("求和")
    mul := f1("求积")
    ave := f1("求均值")
    dec := f1("求差")
    fmt.Printf("求和: %.2f  求差: %.2f  求积: %.2f  求均值: %.2f\n", add(2, 3),
dec(13, 7), mul(4, 5), ave(6, 7))
}
```

运行结果：

```
求和: 5.00 求差: 6.00 求积: 20.00 求均值: 6.50
```

程序分析：

上述程序中，add，mul，ave 及 dec 都是对闭包的引用，本身也成为闭包，被 fmt. Printf 引用。

函数作为返回值的时候一定要注意函数签名，返回的闭包的签名一定要和函数的返回值函数的签名一致。

例如，上述的 x，y 两个浮点型参数及一个浮点型返回值。如果返回的闭包的签名与返回值函数的签名不一致，会导致系统报错。

5.3.3 多返回值处理

如果被调用函数只有两三个返回值的时候，只需要在返回值列表里列出即可。但是，如果数量比较多的返回值需要处理，那就不能这么做了，例如，对结构体或数组等需修改的数据量比较大的情况。这时，可以采用引用传递的方式，直接操作实参，改变原值。

如果返回值都是同类型的则可以用切片作参数；如果是不同类型数据的返回值，则可以使用结构体，然后使用指向结构体的指针作参数。

多返回值的函数不建议作为语句单独执行，因为其返回值将会被丢弃。最好作为表达式来为变量赋值。

当多返回值函数作表达式赋值给变量的时候，一定要采用平行赋值的方式。赋值符左

边的变量个数必须与返回值个数一致，且变量必须预先定义，否则必须采用短变量赋值操作符“:=”赋值。如下所示：

假设有函数

```
func f1(para_list)(a,b int,c string){block}
```

则必须用以下方式调用该函数

```
var x,y int
var z string
x,y,z = f1(real_list)
```

或者

```
x,y,z := f1(real_list)
```

x,y,z 严格对应 a,b,c 顺序，数量不能少，位置不能错，否则系统将报错。

5.3.4 return 语句

return 语句为函数的最后一个语句，用来指明返回值，原则上要求所有函数都要保留，以明示函数结束。但是，Go 语言为了程序员少敲键盘，允许程序员有条件地省略 return 语句。如果函数不带返回值，则可以省略 return 语句；如果函数带有返回值，则必须以 return 语句结束函数，无论 return 语句后面是否带有表达式，都不可省略。return 语句后面可以带返回值表达式，也可以为空。

如果函数的返回值为匿名函数，则 return 必须返回同类型匿名函数，不得为空。

如果函数的返回值列表里只有变量类型，没有命名变量，则 return 语句必须带返回值表达式，表达式的数量与返回值的类型数量一致，表达式计算结果的类型与返回值列表中的签名类型必须完全一致。

如果 return 语句后面带有多个表达式，每个表达式可以用小括号括起来，也可以没有，各表达式之间用逗号隔开。

如果返回值列表里所有变量都是命名的，那么 return 后面要么什么都不写，要么全写返回值变量，而且数目、顺序必须一致。

视频讲解

5.4 匿名函数与闭包

匿名函数就是没有函数名字的函数，在很多编程语言中都有。在 Go 语言中，匿名函数也称为闭包，在特定场合下很有用。

匿名函数与命名函数的唯一差别是没有名称，其他的都一样。

以下是一个简单的匿名函数声明：

```
func (a,b string)string{
    return a + b
}
```

上述匿名函数用来合并两个字符串，但是，因为没有名字，无法单独执行。

要执行匿名函数有两种方式，一种是在声明的同时加入实参，让其直接执行。方法是在函数体的右大括号外部加入一对小括号，小括号内输入参数，如下所示：

```
package main
import "fmt"
func main() {
    fmt.Printf("%s\n", func(a, b string) string {
     return a + b
    }("Hello, ", "World!"))
}
```

运行结果：

```
Hello, World!
```

另一种方法是将匿名函数赋给一个变量，通过变量的引用来执行函数，如下所示：

```
package main
import "fmt"
func main() {
    q := func(a, b int) int {
                return a + b
        }
    fmt.Printf("q = %d\n", q(2, 3))
}
```

运行结果：

```
q = 5
```

上述程序中变量 q 为匿名函数的引用，为一个函数类型变量。如果匿名函数在定义的时候直接给它赋值执行，则 q 就会成为一个普通的整型变量，为匿名函数的返回值，请看下例：

```
package main
import "fmt"
func main() {
    q := func(a, b int) int {
            return a + b
            }(12, 23)
    fmt.Printf("q = %d\n", q)
}
```

运行结果：

```
q = 35
```

从上述程序可以看出，变量 q 已不再是匿名函数的引用，而是匿名函数的返回值，只是个普通变量，打印的时候不能再以引用的方式 q(12,23)，而只能以普通的变量方式 q。

匿名函数在很多地方都很有用，例如定义函数类型就需要用匿名函数，以函数作为返回值也需要匿名函数，有些局部应用也常常使用闭包的形式。

闭包作为函数的返回值时，这样的函数也叫工厂函数。下例就是一个工厂函数的典型例子。

```
package main
import "fmt"
type fit func(int, int, int) int        /* 定义一个函数类型 */
func sum(x, y, z int) int {             // 求和
   return x + y + z
}
func mul(x, y, z int) int {             // 求乘积
   return x * y * z
}
func aver(x, y, z int) int {            // 求平均值
   return (x + y + z) / 3
}
func max(x, y, z int) int {             // 求最大值
   switch {
   case x >= y && x >= z:
      return x
   case y >= x && y >= z:
      return y
   case z >= y && z >= x:
      return z
   }
   return 0
}
func min(x, y, z int) int {             // 求最小值
   switch {
   case x <= y && x <= z:
      return x
   case y <= x && y <= z:
      return y
   case z <= y && z <= x:
      return z
   }
   return 0
}
func normal(s string, f []fit) fit {    // 工厂函数
    var x fit
    switch s {
    case "sum":
        x = f[0]
    case "mul":
        x = f[1]
    case "aver":
        x = f[2]
    case "max":
        x = f[3]
    case "min":
        x = f[4]
    }
```

```
    return x
}
func main() {
    var a, b, c int
    f := make([]fit, 5)              //  函数仓库
    f[0] = sum
    f[1] = mul
    f[2] = aver
    f[3] = max
    f[4] = min
    fmt.Println("请输入三个整数: ")
    fmt.Scanf("%d,%d,%d", &a, &b, &c)
    result := normal("sum", f)       // 采用工厂函数返回所需算法函数
    fmt.Printf("三个数之和为: %d\n", result(a, b, c))
    result = normal("mul", f)
    fmt.Printf("三个数之积为: %d\n", result(a, b, c))
    result = normal("aver", f)
    fmt.Printf("三个数平均值为: %d\n", result(a, b, c))
    result = normal("max", f)
    fmt.Printf("三个数之最大值为: %d\n", result(a, b, c))
    result = normal("min", f)
    fmt.Printf("三个数之最小值为: %d\n", result(a, b, c))
}
```

运行结果：

```
请输入三个整数:
32,53,78
三个数之和为: 163
三个数之积为: 132288
三个数平均值为: 54
三个数之最大值为: 78
三个数之最小值为: 32
```

程序分析：

要实现工厂函数，需要遵循几个步骤：

(1) 要先定义一个通用的函数类型，工厂生产出来的函数保持类型一致；

(2) 定义好类型相同的函数 n 个；

(3) 定义一个函数类型的切片，将上述定义好的 n 个函数全部装入切片中，构成函数库；

(4) 再定义一个工厂函数，该函数的参数为一个函数选择变量及一个函数类型切片。选择变量告诉工厂函数，应该产出什么函数，函数类型切片保存有 n 个用户自定义的函数，工厂函数根据用户的选择指令，从切片中选择一个函数返回；

(5) 主程序以选择指令及切片函数库作为实参调用工厂函数，返回所需的子函数赋给一个变量；

(6) 主函数中根据用户输入的参数作实参，调用该参数所代表的函数。

工厂函数有点类似面向对象的函数的多态。

5.5 init 函数和 main 函数

5.5.1 init 函数


视频讲解

init 函数为 Go 语言保留的用于包级别的初始化函数，该函数不能接收任何参数，也没有返回值。init 函数由用户实现，但用户不能显式调用它，只能由操作系统调用执行。每个包内可以有 0 到多个 init 函数，多个 init 函数的执行顺序默认按其出现的顺序。各个包内的 init 函数的执行顺序严格按照引入的顺序执行。一个包在不同的包中被引入多次，实际只有第一次会被执行，以后再次出现引入相同的包名会被编译系统忽略。每个被引入包内的 init 函数总是先于引入它的包被执行。每个包内的执行顺序总是先创建常量和变量，接着执行 init 函数，然后再返回引入它的上层包。Go 语言程序的具体执行顺序流程图如图 5-1 所示。

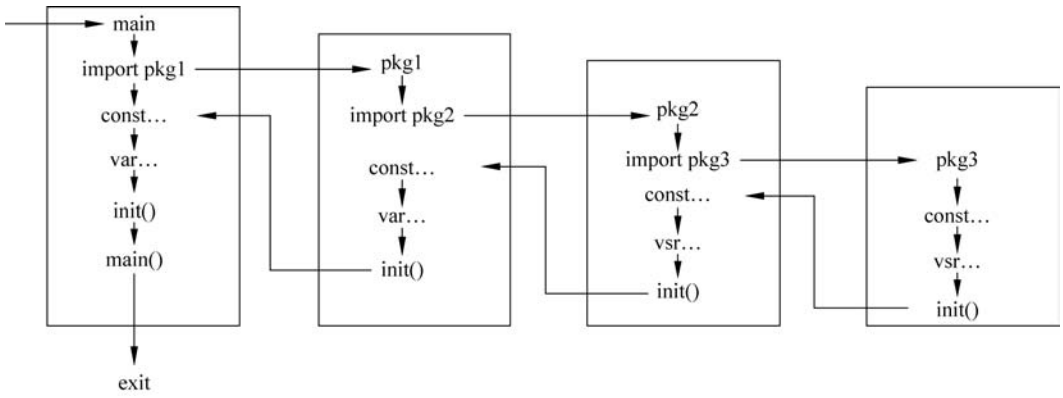

图 5-1 init 函数执行顺序图

5.5.2 main 函数

Go 语言规定 Go 程序必须有一个 main 包，main 包中必须有一个 main 函数，用作程序的入口。尽管有 init 函数的存在，先于 main 函数被执行，但是，init 函数仅仅是包级别的初始化，多用于环境变量、注册等应用，只有 main 函数才是用户程序的真正开始。main 函数没有参数，也没有返回值，不使用 return 语句。Go 语言在执行完所有包的 init 函数及常量、变量定义后开始执行 main 函数，main 函数执行完毕，将退出程序，返回操作系统。

5.6 错误与恢复机制


视频讲解

错误是不可避免的，无论是在编译期间还是在运行期间。在编译期间出现的问题都是小问题，好处理，易于解决。运行期间出现的错误往往导致程序崩溃，是大问题，需要提前于编程期间就做好预案。对于错误的处理，Go 语言提供了一些机制，比如提供了 error 错误类型接口，还有 errors 标准包、defer 语句、内置函数 panic 和 recover 等，下面分别说明。

5.6.1 错误信息提示

程序运行过程中都不可避免地会出现错误，例如我们使用浏览器的时候，有时候打不开网页；发送电子邮件的时候，有时发送失败；登录账号的时候输错密码；输入身份证号码的时候数字位数不足，或使用了非法字符等，诸如此类的错误，都会导致程序无法正常执行，甚至崩溃死锁等。当出现程序执行中断的时候，用户或者编程人员都想知道是什么原因导致程序执行失败。这时候如果程序能返回错误信息提示，将非常有利于用户及编程人员排查错误。Go 语言的内置函数 panic 就是用来抛出这种错误信息的。在 Go 语言标准库中有大量的函数都带有错误返回信息，请看我们经常使用的打印语句的格式：

```
func Printf(format string, a ...interface{}) (n int, err error)
```

上述打印语句有两个返回值：n 表示输出字符数，err 表示错误信息。这种错误信息就不是由内置函数 panic 抛出的，而是由程序员根据错误情况提前预置的。如果我们错误地使用了打印语句，系统就会根据错误类型，返回不同的错误信息，提示用户检查。如果是在运行时出现的未知错误，错误信息会由运行时系统使用 panic 函数抛出。根据错误提示，顺藤摸瓜，就很容易发现问题的根源。

5.6.2 defer 语句

defer 语句在 Go 语言中称为滞后执行语句，主要用于函数返回前清理现场使用。其语法格式是：

```
defer  函数或方法
```

defer 语句后边必须是已定义的函数或方法，表示由 defer 语句执行对函数或方法的调用。因而，defer 后不能是表达式，但可以是匿名函数或闭包。defer 语句可以放在函数体内任何位置，且一个函数体内允许有多条 defer 语句，其执行顺序采用堆栈机制，先出现后执行。defer 语句还可以嵌套，即 defer 语句执行的函数体内还可以包含其他的 defer 语句。

以下这段代码常常被用来说明不使用 defer 存在的隐患：

```
func CopyFile(dstName, srcName string) (written int64, err error) {
src, err := os.Open(srcName)
if err != nil {
    return
}
dst, err := os.Create(dstName)
if err != nil {
    return
}
written, err = io.Copy(dst, src)
dst.Close()
src.Close()
return
}
```

上述程序代码用来复制一份文件，如果源文件能正常打开，以及目标文件名能正常创

建，则复制过程大致顺利。如果其中的创建目标文件出现错误，程序直接返回，则后续的文件关闭工作将被忽略，有可能导致已经打开的源文件受损。

为此，可以使用 defer 语句对上述程序进行修改，如下所示：

```
func CopyFile1(dstName, srcName string) (written int64, err error) {
src, err := os.Open(srcName)
if err != nil {
    return
}
defer src.Close()
dst, err := os.Create(dstName)
if err != nil {
    return
}
defer dst.Close()
return io.Copy(dst, src)
  }
```

利用 defer 的滞后执行特性，在函数 return 返回之前，执行 defer 语句，把文件关闭，就可避免出现安全隐患。

不管是正常的 return 返回，还是由于错误发生导致的返回都会触发 defer 的执行。

在一个函数中 defer 语句可以有多个，但是执行的时候是按逆序执行的，类似堆栈，先进后出，先出现的 defer 语句最后执行。

defer 的执行时间是在函数的返回值计算完毕之后马上返回到调用者之前执行，即 return 语句后的表达式计算完毕之后。

在程序中每遇到一个 defer 语句，会即时解析 defer 语句后函数的参数，并会暂存实参的当前值，但不执行该函数体。

defer 语句调用的函数或方法的作用域与 defer 语句所在函数的作用域相同，这就意味着 defer 语句所在函数的返回值可以被 defer 语句中调用的函数或方法修改。请仔细观察以下例子：

```
package main
import (
    "fmt"
)
func main() {
    var x [5]struct{}
    for i := range x {
    defer func(y int) { fmt.Println(y) }(i)
    }
}
```

运行结果：

```
4
3
2
1
0
```

从上述结果可以看出，defer 语句的打印语句是按逆序打印的，而且每个打印语句保留了 i 的值。再继续观察：

```
package main
import (
    "fmt"
)
func main() {
    var x [5]struct{}
    for i := range x {
    defer func() { fmt.Println(i) }()
    }
}
```

运行结果：

```
4
4
4
4
4
```

我们将上述程序进行了修改，将带参数的闭包改成不带参数，执行结果有点出乎预料。上述程序说明两个问题：

第一，Go 语言解析 defer 语句时仅保留当前传入函数的实参，不考虑函数体内的变量值，即不解析函数体；

第二，如果把 defer 语句后的函数称为子函数，defer 语句所在的函数称为母函数的话，那么子函数与母函数的作用域相同，子函数执行的时候能够读到母函数变量的当前值。因此，上述程序最后执行 defer 语句的时候能读到 i 的当前值为 4。在遍历 x 的时候，由于闭包不带参数，defer 后的函数不带参数，不需要传入实参，所以并没有暂存解析 defer 时 i 的当前值，只有执行到函数体的时候才会去读取 i 值，这时 i 值已经为 4，导致执行每一个 defer 输出都是 4。关于作用域的问题还可以参看以下例子：

```
package main
import (
    "fmt"
)
func F(a, b int) (s int) {
    defer func() { s = 2 * s }()
    return a + b
}
func main() {
    s := F(1, 2)
    fmt.Printf("s = %d\n", s)
}
```

运行结果：

```
s = 6
```

从上述程序可以看出，defer 语句后的函数能够读取母函数的命名返回值，并加以修

改。当然,这种用法不太合乎常理,一般 return 的返回值就是函数的执行结果,不应该再被修改。上述用法只是用来证明 defer 执行函数的作用域与 defer 所在函数的作用域相同。

defer 语句最常见的使用是配合 recover 函数,恢复被 panic 函数中断的程序,下面两小节将详细说明。

5.6.3 panic 函数

panic 的中文意思叫“恐慌”,程序运行过程中出现恐慌那就意味着出现了异常,有可能导致程序崩溃。运行时 panic 由 runtime 系统触发,一旦触发将中断程序的运行,并给出相关错误信息。panic 也是 Go 语言的内置函数,可以由用户主动显式调用。panic 函数的格式为:

```
panic(v interface{})
```

由于 panic 函数的参数为空接口,意味着 panic 函数接受任意类型的参数,实际编程中主要是传入错误信息,多为字符串类型。

程序员在编程过程中也可以人为地插入一些 panic 函数,返回特定信息,以便于程序调试。显式调用 panic 函数举例如下:

```
package main
import (
    "fmt"
)
func main() {
    var w int
    fmt.Printf("panic()函数显式调用测试!请输入一个正数: \n")
    fmt.Scanf("%d\n", &w)
    if w <= 0 {
    panic("输入非法,程序异常退出!")
    }
    fmt.Printf("w = %d\n", w)
    fmt.Printf("程序结束,正常退出!\n")
}
```

运行结果:

```
panic()函数显式调用测试!请输入一个正数:
-9
panic: 输入非法,程序异常退出!

goroutine 1 [running]:
main.main()
C:/Users/Administrator/q1/q1.go: 12 +0x186
```

由于输入不满足要求,触发了 panic 函数的执行,程序异常退出,并打印 panic 函数的参数内容,以及层层退出堆栈的内容。

程序中断退出,无法正常执行后续的 w 参数打印输出。

请看下面的正常输出:

运行结果：

```
panic()函数显式调用测试!请输入一个正数:
9
w = 9
程序结束,正常退出!
```

事实上，如果程序员能够预料到的错误，一般不用显式调用 panic 函数，因为该函数会导致程序非正常退出。通常的做法都是返回一个错误值，程序员会根据不同的错误值来处理异常。例如，上述程序可更改如下：

```
package main
import (
    "errors"
    "fmt"
)
func getint() (w int, err error) {
    fmt.Printf("请输入一个正数: \n")
    fmt.Scanf("%d\n", &w)
    if w <= 0 {
        err = errors.New("输入非法,请按要求输入!")
        return
    }
    return
}
func main() {
    w, err := getint()
    fmt.Printf("w = %d  err = %v\n", w, err)
    fmt.Printf("程序结束,正常退出!\n")
}
```

运行结果：

```
请输入一个正数:
-9
w = -9    err = 输入非法,请按要求输入!
程序结束,正常退出!
```

用户输入了一个非法数字，程序正常退出，返回输入值及一个错误值。而且不影响调用者，调用者程序流程可以继续执行。这种程序设计思路更加合理，不会异常中断程序。

如果输入正确的数字，结果如下：

运行结果：

```
请输入一个正数:
6
w = 6 err = <nil>
程序结束,正常退出!
```

因此，在程序中尽可能地不用 panic，如果是意料之外由 runtime 触发的 panic，程序员无法避免。但是，还是有办法不让程序直接退出的，这就需要配合 recover 函数来使用。

5.6.4 recover 函数

recover 函数是 Go 语言提供的专门用于“拦截”运行时 panic 信息的内建函数，它可以

使当前的程序从运行时 panic 中断状态中恢复并重新获得流程控制权。recover 函数的格式如下：

```
func recover() interface{}
```

recover 函数不接收任何参数，但是能拦截 panic 抛出的错误信息，并以接口类型返回。recover 函数用来恢复被 panic 中断的程序。recover 函数只能在 defer 语句后的函数中被显式调用，在其他地方被调用不会产生任何效果，仅返回一个 nil 值。

由于运行时 panic 异常一旦被触发就会导致程序崩溃，因此，为免服务中断，一般都在函数中配备 defer 语句，让 defer 执行 recover 来捕获 panic 信息，根据捕获的信息执行相关的操作，以恢复被 panic 中断的程序。

如果 defer 语句所在的函数发生了 panic 异常，而 defer 语句又调用了内置函数 recover，则 recover 函数会使程序从 panic 中恢复，并返回一个 panic value，根据该 value 就可以判断错误类型，执行相关操作。

导致 panic 异常的函数不会继续运行，但能正常返回调用者，返回前会执行 defer 语句。在未发生 panic 时调用 recover 函数，recover 函数会返回 nil。

下面的示例说明不加 recover 函数的执行结果：程序层层退出至操作系统。

```
package main
import "fmt"
func Test1() {
    fmt.Println("recover 函数功能测试.Test1")
    return
}
func Test2() {
    panic("Test2 函数异常: panic")
    return
}
func Test3() {
    fmt.Println("recover 函数功能测试.Test3")
    return
}
func main() {
    Test1()
    Test2()
    Test3()
}
```

运行结果：

```
recover 函数功能测试.Test1
panic: Test2 函数异常: panic

goroutine 1 [running]:
main.Test2(...)
D:/Go 语言/src/ttqqq.go: 18
main.main()
D:/Go 语言/src/ttqqq.go: 29 + 0x45
```

执行 Test1 函数的时候，屏幕打印输出正常。

执行 Test2 函数的时候，由于显式调用了 panic 函数，导致程序中断，直接退出 Test2 函数，接着直接退出 main 函数，于是 main 函数中的 Test3 函数没得到执行。

如果使用了 recover 函数，则 Test3 函数就可以正常执行，如下所示：

```
package main
import "fmt"
func Test1() {
fmt.Println("recover 函数功能测试.Test1")
    return
}
func Test2() (err error) {
    defer func() {
    if p := recover(); p != nil {
         err = fmt.Errorf("发生异常： %v", p)
    }
    }()
    panic("Test2 函数异常：panic")
    fmt.Println("发生 panic 后，测试是否还正常执行本打印项目.")
    return
}
func Test3() {
    fmt.Println("recover 函数功能测试.Test3")
    return
}
func main() {
    Test1()
    err := Test2()
    fmt.Println(err)
    Test3()
}
```

运行结果：

```
recover()函数功能测试.Test1()
发生异常：Test2()函数异常：panic
recover()函数功能测试.Test3()
```

由于 Test2 函数中设置了 defer 语句，其中的闭包调用了 recover 函数，使得被 panic 中断掉的 Test2 函数能正常返回，主函数中调用 Test2 的返回值 err 有效，且其后的打印语句及 Test3 函数都可以正常执行。

这就是 recover 函数所起的作用。

细心的读者可能会发现，Test2 函数中的“fmt.Println"语句并没有被执行，而是直接执行 return 返回了。这就说明 recover 函数不是从 panic 引起的中断处恢复，而是直接从 return 处恢复。同时也说明任何情况下退出函数之前都会执行 defer 语句。

defer 语句后的函数执行过程中可能还会触发 panic，当有多个 panic 出现的时候，recover 函数捕获的是最后一个 panic 携带的信息；同理，当有多个 recover 函数执行的时候，只有第一个 recover 函数能够捕获 panic 的信息。参看以下例子：

```
package main
import "fmt"
func test1() {
    defer func() {
        fmt.Println("1: recover = ", recover())
    }()
    defer func() {
        fmt.Println("2: recover = ", recover())
    }()
    defer func() {
          panic("1: defer panic")
    }()
    test2()
    panic("2: test panic")
}
func test2() {
    defer func() {
        fmt.Println("3: recover = ", recover())
    }()
    defer func() {
        fmt.Println("4: recover = ", recover())
    }()
    defer func() {
        panic("3: defer panic")
    }()
    panic("4: test panic")
}
func main() {
    test1()
}
```

运行结果：

```
4: recover = 3: defer panic
3: recover = <nil>
2: recover = 1: defer panic
1: recover = <nil>
```

由上述函数的执行结果可以看出，两个 test 函数内的 panic 信息总是捕获不到，defer 后的 panic 信息总是能捕获到。说明后出现的 panic 信息会覆盖之前的 panic 信息，因而 recover 函数捕获的总是最后一个 panic 的信息。

一旦 panic 的信息被捕获成功后，panic 信息字段将被清空，因而后续的 recover 函数将捕获不到信息，只能返回 nil 值。


视频讲解

5.7 递归函数

在函数定义中，如果出现了直接或间接调用自身，这样的函数称为递归函数，这样的调用称为递归调用。

(1) 直接递归调用。

直接递归调用形式如下所示：

```
func recu(x, y int)int{
   a := x + y
   b := x - y
   c := recu(a,b)
  …
   return m
}
```

显然，直接调用是在函数定义中自己调用自己，这是比较纯粹的递归调用。

(2) 间接递归调用。

我们再来看看如下形式的函数调用：

```
func sum(x,y int)int{
     ...
   s := add (y,x)
   return s
}
func add(x,y int)int{
   c := sum(x,y)
    ...
   return c
}
```

这种调用方式是在被调用的函数中调用自己，这种调用称为间接调用。无论是直接还是间接调用自己，都称为递归调用。

递归调用表面上看来像死循环，但是，只要合理地设置结束条件，是可以避免死循环的。

递归函数算法在实际生活中也是很有意义的，如求 n 的阶乘，实现斐波那契数列，实现快速排序算法等。下面我们用递归函数来求 n 的阶乘，n 由键盘输入。

```
package main
import "fmt"
func multn(n int) (result int) {
    if n > 0 {
        result = n * multn(n - 1)
    return
    }
    return 1
}
var result int
func main() {
    var x int
    fmt.Println("请输入一个大于 1 的整数: ")
    for {
        fmt.Scanf("%d\n", &x)
           if x > 1 {
              break
        }
```

```
        fmt.Println("输入参数不合格,请重新输入.")
        }
        result = multn(x)
        fmt.Printf("%d 的阶乘是: %d\n", x, result)
}
```

输出：

```
请输入一个大于 1 的整数:
7
7 的阶乘是: 5040
```


视频讲解

5.8 内置函数简介

Go 语言预定义了一些常用的函数，方便用户编程时直接引用。Go 语言的内置函数不用声明，可直接引用。各个内置函数简要介绍如下：

- append——切片追加函数，返回值为切片 slice。
- close——关闭函数，用于关闭 channel 或者 file。
- delete——从 map 中删除 key 对应的 value。
- panic——停止常规的 goroutine，抛出异常。
- recover——异常处理和恢复。
- real——返回复数 complex 的实部。
- imag——返回复数 complex 的虚部。
- make——创建并初始化 slice，map，channel。
- new——返回指向 Type 的指针。
- cap——返回切片或映射的容量 capacity。
- copy——复制 slice，返回复制的数目。
- len——返回切片、数组、映射的长度 length。

上 机 训 练

一、训练内容

1. 编程实现由键盘输入任意个整数，将其中的奇数和偶数构成两个切片输出。

2. 编程实现从键盘输入三个整数，用子函数方式求三个数的和及均值(浮点型)，并打印输出。

3. 编程实现从键盘输入若干整数，找出其中的素数，以切片的形式输出，并统计所有素数之和。

二、参考程序

1. 编程实现由键盘输入任意个整数，将其中的奇数和偶数构成两个切片输出。

参考程序

```
package main
import (
    "fmt"
)
func getoddeven(slice []int) (sl1, sl2 []int) {
    for _, value := range slice {
        if value % 2 != 0 {
            sl1 = append(sl1, value)
        } else {
            sl2 = append(sl2, value)
        }
    }
    return
}
func main() {
    var (
    input, i  int
    slice    []int
    )
    slice1 := make([]int, 100)
    fmt.Println("请输入任意数目的整数,用逗号分隔,输入负数结束.")
    for input >= 0 {
        fmt.Scanf("%d,", &input)
        if input >= 0 {
            slice1[i] = input
            i++
        }
    }
    for _, value := range slice1 {
        if value == 0 {
            break
        }
        slice = append(slice, value)
    }
    fmt.Printf("输入的整数为 %v\n", slice)
    slice2, slice3 := getoddeven(slice)
    fmt.Printf("输入中的奇数为 %v\n", slice2)
    fmt.Printf("输入中的偶数为 %v\n", slice3)
}
```

运行结果

```
请输入任意数目的整数,用逗号分隔,输入负数结束.
12,43,54,23,45,65,78,76,45,32,97,57,342,4567,432,35,45,73,-9
输入的整数为 [12 43 54 23 45 65 78 76 45 32 97 57 342 4567 432 35 45 73]
输入中的奇数为 [43 23 45 65 45 97 57 4567 35 45 73]
输入中的偶数为 [12 54 78 76 32 342 432]
```

2. 编程实现从键盘输入三个整数,用子函数方式求三个数的和及均值(浮点型),并打印输出。

参考程序

```
package main
import "fmt"
func sm(x, y, z int) (int, float64) {
    sum := x + y + z
    ave := float64(sum) / 3.0
    return sum, ave
}
func main() {
    var a, b, c int
    fmt.Println("请输入三个整数: ")
    fmt.Scanf("%d,%d,%d", &a, &b, &c)
    sum, ave := sm(a, b, c)
    fmt.Printf("sum = %d   ave = %.2f\n", sum, ave)
}
```

运行结果

```
请输入三个整数:
32,15,61
sum = 108   ave = 36.00
```

3. 编程实现从键盘输入若干整数,找出其中的素数,以切片的形式输出,并统计所有素数之和。

参考程序

```
package main
import (
    "fmt"
)
func getprime(slice []int) (sl1 []int) {
    for _, value := range slice {
        i := 2
        for i < value {
          if value % i == 0 {
              break
          }
          i++
        }
        if i == value {
            sl1 = append(sl1, value)
        }
    }
    return
}
func main() {
    var (
    input, i int
    slice    []int
    )
    slice1 := make([]int, 100)
    fmt.Println("请输入任意数目的整数,用逗号分隔,输入负数结束.")
```

续

参考程序

```
        for input >= 0 {
        fmt.Scanf("%d,", &input)
        if input >= 0 {
            slice1[i] = input
            i++
        }
    }
    for _, value := range slice1 {
        if value == 0 {
            continue
        }
        slice = append(slice, value)
    }
    fmt.Printf("输入的整数为 %v\n", slice)
    sprime := getprime(slice)
    sum := 0
    for _, value := range sprime {
        sum += value
    }
    fmt.Printf("输入其中的素数为 %v\n", sprime)
    fmt.Printf("输入的素数之和为 %v\n", sum)
}
```

运行结果

```
请输入任意数目的整数,用逗号分隔,输入负数结束.
123,23,34,45,56,67,78,809,87,67,5,45,45,34,2,323,,3445,456,65,723,23,23,45,-8
输入的整数为 [123 23 34 45 56 67 78 809 87 67 5 45 45 34 2 323 323 3445 456 65 723 23 23 45]
输入其中的素数为 [23 67 809 67 5 2 23 23]
输入的素数之和为 1019
```

习　　题

一、单项选择题

1. 以下关键字用来声明函数的是(　　)。

   A. var　　B. func　　C. type　　D. new

2. 以下关键字用来声明函数类型的是(　　)。

   A. make　　B. func　　C. type　　D. close

3. 假设函数声明为 func f(a,b int)(int,int){block},以下 return 正确的是(　　)。

   A. return b　　B. return a int　　C. return a+b　　D. return a,b

4. 以下函数声明的参数列表正确的是(　　)。

   A. func f(a,b)(int,int){block}　　B. func f(a int,b)(int,int){block}

   C. func f(a,b int)(int,int){block}　　D. func f(int a,b)(int,int){block}

5. 以下函数声明的返回值列表正确的是(　　)。

A. func f(a,b int)(x int,int){block}　　B. func f(a,b int)(int x,y int){block}

C. func f(a,b int)(int x, y){block}　　D. func f(a,b int)(x,y int){block}

6. 下列数据类型为引用型的是(　　)。

A. string　　B. [12]int　　C. []bype　　D. float64

7. 如果要实现函数参数可变,下列声明方式正确的是(　　)。

A. func f(a,b int...)(int,int){block}　　B. func f(a,b ...int)(int,int){block}

C. func f(a int,...b)(int,int){block}　　D. func f(...a, b int)(int,int){block}

8. 下列(　　)数据类型为值类型。

A. 映射　　B. 通道　　C. 数组　　D. 切片

9. Go 语言执行程序的入口是(　　)函数。

A. new()　　B. init()　　C. main()　　D. append()

10. 属于 Go 语言的内置函数的是(　　)。

A. new()　　B. init()　　C. main()　　D. app()

11. 获取数组长度的内置函数的是(　　)。

A. new()　　B. len()　　C. main()　　D. make()

12. 可以用来将切片扩容的函数是(　　)。

A. new()　　B. len()　　C. cap()　　D. append()

13. 以下(　　)是滞后执行语句。

A. delay　　B. defer　　C. late　　D. after

14. 以下(　　)函数可以用来中断程序执行并抛出异常。

A. close()　　B. delete()　　C. panic()　　D. recover()

15. 以下(　　)函数可以用来捕获 panic 抛出的信息。

A. recover()　　B. panic()　　C. defer()　　D. append()

16. defer 语句只能执行(　　)。

A. 算术表达式　　B. 逻辑表达式　　C. 函数　　D. 字符串表达式

二、判断题

1. 函数是独立可执行的一个程序块。(　　)
2. 函数只有被调用后才会执行。(　　)
3. 函数可以没有任何参数。(　　)
4. 函数可以没有任何返回值。(　　)
5. init()函数是一个不接收任何参数也没有返回值的函数。(　　)
6. Go 语言程序的入口为 init()函数。(　　)
7. Go 语言的 main()函数可以放入任何包中。(　　)
8. 用户可以在任何位置调用 init()函数。(　　)
9. 用户可以在任何位置调用 main()函数。(　　)
10. 函数的参数列表可以仅包含类型,不命名变量。(　　)
11. 函数的返回值列表可以仅包含类型,不命名变量。(　　)

12. 函数的返回值列表可以部分命名变量。(　　)
13. 函数的返回值列表可以全部命名变量,不使用类型。(　　)
14. 闭包可以通过赋值一个变量来被引用。(　　)
15. 闭包不赋值变量也可以直接执行。(　　)
16. 函数也可以作为函数的参数。(　　)
17. 函数也可以作为函数的返回值。(　　)
18. 函数可以嵌套调用。(　　)
19. 函数可以嵌套声明(匿名函数除外)。(　　)
20. 函数作为实参时是按值传递的。(　　)
21. 数组作为参数时是按引用传递的。(　　)
22. 映射作为参数时是按值传递的。(　　)
23. 结构体作为参数时是按引用传递的。(　　)
24. 切片作为参数时是按值传递的。
25. 以函数作为返回值的函数可以称为工厂函数。(　　)
26. 函数的可变参数实质是一个同类型的切片。(　　)
27. 函数的参数列表为空时小括号可以省略。(　　)
28. 函数的返回值列表为空时小括号可以省略。(　　)
29. 函数的返回值列表中各变量之间用分号隔开。(　　)
30. 声明类型函数时必须带函数名称标识符。(　　)
31. 滞后执行语句 defer 必须放置在函数的最后紧接着 return 语句。(　　)
32. recover 函数可以从 panic 引起的断点处恢复继续执行程序。(　　)
33. 在程序中任何位置调用 recover 函数效果是一样的。(　　)
34. 用户显式调用 panic 仅影响其所在函数,对调用方没有影响。(　　)
35. 程序中有多个 defer 语句的话,其执行顺序是按其出现的先后顺序执行。(　　)
36. defer 语句执行的只能是函数或方法。(　　)
37. 运行时 runtime 触发的 panic 是不可以恢复的。(　　)
38. 连续多个 panic 的话,recover 函数只能捕获到最后一个 panic 的信息。(　　)

三、分析题

1. 请找出以下函数声明中的错误并改正。

A. func f(a,b)(int,int){block}
B. func f(int a,int b)(int,int){block}
C. func f(a,b)(x,y){block}
D. func f(a,b int)(a int,int){block}
E. func f(a,b int)(x,y){block}
F. func f(a[2],b int)(int,int){block}
G. func f(a,b struct)(int,int[3]){block}
H. func f(a,b []type)x int{block}
I. func f(a,b int)y {block}
J. func f(a,b int[])(int,int){block}

2. 请找出以下函数调用中的错误并改正。

A. fun(x int)

B. fun(int)

C. fun(x int; y int)

D. fun([]int)

3. 请找出以下函数定义中的错误。

A.
```
func f(x int,y int){
    x :=10
    …
    return x
}
```

B.
```
func f(x int,y int)(a,b int){
    return x+y
}
```

C.
```
func f(x int,y int)(a,b int){
    a=x+y-b
    return x-y
}
```

D.
```
func f(x int,y int)int{
    z :=x+y
    return
}
```

4. 请找出以下函数类型定义中的错误。

A. func ft(int,int)(int,int)

B. type func ft(int,int)(int,int)

C. func type ft(int)(int,int)

D. type ft func(int)(int,int){block}

E. type ft func(int)(x)

5. 请分析以下程序功能并给出执行结果。

```
package main
import "fmt"
func multn(n int) (result int) {
    if n > 0 {
        result = n * multn(n - 1)
        return
    }
    return 1
}
func main() {
    var x int
    fmt.Println("请输入一个大于 1 的整数: ")
    for {
        fmt.Scanf("%d\n", &x)
        if x > 1 {
            break
        }
        fmt.Println("输入参数不合格,请重新输入.")
    }
    result := multn(x)
    fmt.Printf("%d 的阶乘是: %d\n", x, result)
}
```

6. 请分析以下程序实现的功能并给出执行结果。

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

```
func sm(x, y, z int) (int, float64) {
    sum := (x + y + z)
    ave := float64(sum) / 3.0
    return sum, ave
}
func main() {
    var a, b, c int
    fmt.Println("请输入三个整数: ")
    fmt.Scanf("%d, %d, %d", &a, &b, &c)
    sum, ave := sm(a, b, c)
    fmt.Printf("sum = %d   ave = %.2f\n", sum, ave)
}
```

7. 请分析以下程序段的执行结果。

```
package main
import "fmt"
func main() {
    fg()
    fmt.Println("main over")
}
func fg() {
    defer func() {
        err := recover()
        if err != nil {
        fmt.Println("recover!  ", err)
        }
      }()
  defer func() {
    fmt.Println("defer end!")
  }()
  panic("internal error!")
  fmt.Println("fg over")
}
```

8. 请分析以下程序段的执行结果。

```
package main
import (
    "errors"
    "fmt"
)
func Divide(a, b float64) (result float64, err error) {
    if b == 0 {
    result = 0.0
    err = errors.New("runtime error: divide by zero")
    return
    }
    result = a / b
    err = nil
    return
}
```

```
func main() {
    r, err := Divide(10.0, 8) // 请输入不同值测试程序运行结果
    if err != nil {
    fmt.Println(err)
    } else {
    fmt.Println("计算结果: ", r)
    }
}
```

9. 请分析以下程序段,并写出运行结果。

```
package main
import (
    "fmt"
    "reflect"
)

func main() {
    defer func() {                    // 第一个 defer 后执行,
    err := recover()                  // recover 捕获的是后一个 No.2 panic 的信息——函数
    if err != nil {                   // err 为接口类型,其值为函数
        fmt.Println("err 的类型为: ", reflect.TypeOf(err))
        v, ok := err.(func() string) //   利用断言表达式求取接口类型的值
        if ok {
        fmt.Println("err 的值为: ", err)
        fmt.Println("v 函数的返回值为: ",v())
        fmt.Println("err 的类型名为: ", reflect.TypeOf(err).Kind().String())
        } else {
        fmt.Println("类型断言失败!")
        }
    } else {
        fmt.Println("fatal")          //  recover 未能捕获 panic 信息,输出"fatal"
        }
    }()
    defer func() {                    //  第二个 defer 先执行
        panic(func() string {         /*  No.2 panic  抛出一个函数(返回值为字符串),覆
                                          盖了 No.1 panic 抛出的信息"main panic"。 */
        return "defer2  panic"
        })
    }()
    panic("main panic")               //  No.1 panic 先抛出错误信息"main panic"
}
```

四、简答题

1. 什么是面向过程的编程模式?
2. 什么是面向对象的编程模式?
3. 什么叫函数?
4. Go 语言函数的构成包括哪几个部分?
5. 函数名称的命名有什么要求?

6. 函数的参数列表格式有什么要求？
7. 函数的返回值列表格式有什么要求？
8. 什么叫函数签名？
9. 什么叫闭包(匿名函数)？
10. 匿名函数如何调用？
11. 什么叫内部函数？
12. 什么叫外部函数？
13. 外部函数如何声明？
14. 什么叫函数类型？如何声明？
15. 什么叫形参？有什么要求？
16. 什么叫实参？有什么要求？
17. Go 语言默认的值传递有哪些数据类型？
18. Go 语言默认的引用传递有哪些数据类型？
19. 函数的可变参数有什么要求？
20. 什么叫值传递？有什么特点？
21. 什么叫引用传递？有什么特点？
22. 空接口作参数有什么好处？
23. 函数有多返回值时该如何获取？
24. 当函数作为返回值时有什么要求？
25. return 语句的使用规则是什么？
26. init 函数的作用及执行流程是什么样的？
27. main 函数的作用及要求是什么？
28. defer 语句的作用是什么？有什么要求？
29. panic 函数的作用是什么？
30. recover 函数的作用是什么？
31. panic 函数的参数有什么要求？
32. recover 函数是如何捕获错误信息的？
33. recover 函数是如何使用的？
34. 什么叫递归函数？
35. 递归函数有哪些类型？
36. 内置函数 make 有什么作用？

五、编程题

1. 编程实现求梯形的面积并输出，要求梯形的边长及高由键盘输入。
2. 请编程实现输出一个 5×5 的矩阵，要求两条对角线的值为 1，其余的值为 0。
3. 编程实现一个统计函数，计算学生语文、数学、外语、政治等四门课程总成绩及平均分。从键盘输入 4 个学生的成绩，计算及输出结果。
4. 编程实现从键盘上输入一个字符串，统计输出各个字母出现的频度。
5. 编程实现三个子程序：第一个子程序读入任意 10 个整数；第二个子程序实现对这

10 个整数从大到小的排序；第三个子程序实现对这 10 个整数从小到大的排序，并将两种排序结果输出。

6. 编写两个函数计算，输入 n 为偶数时，调用函数求 1/2＋1/4＋…＋1/n，当输入 n 为奇数时，调用函数 1/1＋1/3＋…＋1/n。

7. 编程实现从键盘上输入任意一个字符串，要求把重复的字符全部删除。

8. 编程实现一个工厂函数，该工厂函数能生产出计算三个多边形面积的函数（包括长方形、三角形、圆形），并从键盘输入三个多边形的数据加以验证。