第 5 章
函数与方法

函数是执行计算的命名语句序列。将一段代码封装为函数并在需要的位置进行调用，不仅可以实现代码的重复利用，更重要的是可以保证代码完全一致。

5.1 函数的定义与使用

5.1.1 函数的定义

在 Go 语言中，定义函数的语法格式如下：

```
func funcName( [arg argType] ) [returnType] {
    function body
}
```

其中，funcName 是函数名，arg 是形式参数，argType 是形式参数的类型，returnType 是返回值的类型。Go 语言使用关键字 func 定义函数(Function)。func 后面有一个空格，然后是函数名，接下来是一对小括号，小括号里是可选的形式参数及其对应的类型，小括号后面是可选的返回值及其类型，最后是一对大括号括起来的函数体。函数名、形式参数类型和返回值类型三者共同构成了函数签名。形参名以及返回值的变量名不属于函数签名。定义函数时还需要注意如下几个问题：

(1) 一个函数即使不需要接收任何参数，也必须保留一对小括号；

(2) 如果一个函数没有返回值，则返回值可省略(包括小括号)。

```
func f1() { }                                   //没有形参,没有返回值
func f2(a int) {  }                             //一个形参,没有返回值
//两个形参,一个返回值,返回一个整数
func f3(a int, b int) int { return 1 }
func f4(x float64) (y int, z float64) {         //一个形参,两个返回值
    return 1, 2
}
```

举例：自定义函数 bigger()。

```
func main() {
    var a, b int = 10, 20                       //定义局部变量 a 和 b
    var result int
    result = bigger(a, b)
```

```
    fmt.Printf("较大值: %d\n", result)
}
/* 函数的返回值是两个形式参数(Formal Parameter)x 和 y 中的较大者 */
func bigger(x, y int) int {                    //函数签名 bigger(int, int) int
    if x > y {
        return x
    }
    return y
}
```

上述代码的输出结果：

```
较大值: 20
```

当一组形参或返回值的数据类型相同时，可不必逐个为它们标明数据类型。下面两个函数声明是等价的，即它们的函数签名相同。

```
func f(a, b int, m, n string) { … }
func f(a int, b int, m string, n string) { … }
```

举例：

```
func main() {
    //输出数据类型使用格式控制符%T,Type
    fmt.Printf("%T\n", add)
    fmt.Printf("%T\n", sub)
    fmt.Printf("%T\n", first)
    fmt.Printf("%T\n", zero)
}
func add(x int, y int) int   { return x + y }
func sub(x, y int) (z int)   { z = x - y; return }
func first(x int, _ int) int { return x }
func zero(int, int) int      { return 0 }
```

上述代码的输出结果：

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

5.1.2 函数的调用

函数定义完毕并不能自动运行，只有被调用(Call)时才能运行。下面的代码用整数 10 和 20 调用 bigger()函数，该函数的返回值被赋值给变量 result。

```
result = bigger(10, 20)
```

上述调用 bigger()函数时使用的整数 10 和 20 是实际参数(Actual Parameter)，简称实参。形参没有具体的值，形参的值来自实参。调用函数时必须按照声明的顺序为所有的形参赋值。Go 语言的形参不能带有默认值，这点与 Python 语言等不同。

实参通过值传递的方式为形参赋值，形参是实参的一个副本，对形参执行操作通常不会影响实参的值。但是，如果实参是引用类型，如指针、切片 slice、投影 map，则可以通过形参修改实参的值。

举例：

```
func modify(z * int) {                          //形参 z 是指针类型
    * z = 20
}
func main() {
    var x int = 10
    fmt.Printf("调用函数前 x 的值 = %d\n", x)
    modify(&x)                                  //实参 &x 是引用类型
    fmt.Printf("调用函数后 x 的值 = %d\n", x)
}
```

上述代码的输出结果：

```
调用函数前 x 的值 = 10
调用函数后 x 的值 = 20
```

5.1.3　函数的返回值

通常，定义一个函数是希望它能够返回一个或多个计算结果，这在 Go 语言中是通过关键字 return 实现的。无论 return 语句出现在函数的什么位置，一旦被执行，它都会立即结束函数的执行过程。Go 语言支持函数返回多个值。

```
func twoRetValues() (int, int) {                //返回值必须与 return 语句相匹配
    return 1, 2                                 //不能写作(1, 2)
}
func main() {
    a, b := twoRetValues()
    fmt.Println(a, b)
}
```

上述代码的执行结果：

```
1 2
```

与形参名一样，Go 语言支持对返回值进行命名，此时需要在函数体中显式地使用 return 语句返回。

```
func namedRetValues() (x, y int) {              //两个返回值被命名为 x 和 y
    x = 2
    y = 1
    return                                      //return 语句可为空，等价于 return x, y
}
func main() {
    a, b := namedRetValues()
    fmt.Println(a, b)
}
```

上述代码的输出结果：

```
2 1
```

main()函数是一种特殊类型的函数，它不接收任何参数，也没有返回值。main()函数是一个可执行程序的入口。Go 编译器会自动调用 main()函数，因此不需要显式地调用它。main 包是一个特殊的包。一个可执行程序必须拥有一个 main 包，而且在该包中必须有且只能有一个 main()函数。

与 main()函数类似，init()函数不接收任何参数，也没有返回值。每个包中可以包含一个或多个 init()函数。init()函数按照出现的先后顺序依次执行。init()函数也不需要显式地调用，Go 编译器会自动调用它。init()函数在 main()函数之前执行，它的主要用途是初始化全局变量。如果发生多个包的嵌套引用，则最后导入的包会最先执行其包含的 init()函数。

5.2 lambda 函数

lambda 函数又称为匿名函数。匿名函数没有函数名，只有函数体。定义匿名函数的语法格式如下：

```
func ( [arg argType] ) [returnType] {
    function body
}
```

(1) 在定义的同时调用匿名函数

```
func main() {
    func(x int) {                          //函数嵌套
        fmt.Println(x)
    }(10)
}
```

上述代码的输出结果：

```
10
```

(2) 将匿名函数赋值给一个变量

```
func main() {
    f := func(x int) {                     //将匿名函数赋值给变量 f
        fmt.Println(x)
    }
    f(10)                                  //使用 f()调用匿名函数
}
```

上述代码的输出结果：

```
10
```

再举一个例子：

```
func main() {
    add := func(x int) int {
        return x + 1
    }
    fmt.Println(add(10))
}
```

上述代码的输出结果：

```
11
```

5.3　闭包

可以将闭包(Closure)理解为定义在一个函数内部的匿名函数。本质上，闭包是连接匿名函数内部与外部的桥梁。闭包对外部环境中变量的引用过程称为“捕获”。闭包可以用一个简单的公式表示如下：

闭包 = 匿名函数 + 引用的外部环境

举例：

```
func main() {
    i := 42                                    //声明一个整型变量 i
    f := func() {                              //将匿名函数赋值给变量 f
        j := i / 2                             //访问外部变量 i
        fmt.Println(j)
    }
    f()                                        //输出 21
}
```

上述代码创建了包含整型变量 i 的匿名函数 f 的闭包。函数 f 可以直接访问变量 i，这是闭包的属性。

```
func f() func() int {
    i := 0
    return func() int {                        //函数 f()的返回值是一个匿名函数
        i += 1
        return i
    }
}
func main() {
    a := f()                                   //闭包 a
    b := f()                                   //闭包 b
    fmt.Println(a())                           //输出 1
    fmt.Println(b())                           //输出 1
    b()
    fmt.Println(a())                           //输出 2
    fmt.Println(b())                           //输出 3
}
```

当需要创建一个对状态进行封装的函数时，就需要使用闭包。闭包可以实现很多高级

的功能，如创建一个生成器，限于篇幅，本书不再讲述。

5.4 defer 语句

defer 将其后的语句进行延迟处理。在 defer 语句所属的函数返回之前，将延迟处理语句按照后进先出(Last In First Out，LIFO)的顺序执行。

举例：

```
func main() {
    defer fmt.Printf("%s\n", "first")
    defer fmt.Printf("%s ", "second")
    defer fmt.Printf("%s ", "third")          //最后进入，最先出来
    fmt.Println("exit")
}
```

上述代码的输出结果：

```
exit
third second first
```

defer 语句一般用于释放某些已分配的系统资源，如关闭文件。下面定义函数 fileSize() 用于打开并获取一个文件的大小。

举例：

```
func fileSize(fileName string) int64 {
    f, err := os.Open(fileName)
    if err != nil {
        return 0
    }
    //延迟调用 Close()函数，此刻不会调用它
    defer f.Close()
    info, err := f.Stat()                     //统计 Statistics
    if err != nil {
        return 0                              //函数返回前，会调用 Close()函数关闭文件
    }
    size := info.Size()
    return size                               //函数返回前，会调用 Close()函数关闭文件
}
```

在 main()主函数中调用 fileSize()函数。

```
func main() {
    size := fileSize("test.txt")
    fmt.Printf("File size = %d", size)
}
```

上述代码的输出结果：

```
File size = 25
```

在上述例子中，如果没有 defer 语句，则在 fileSize()函数中后两个 return 语句的前面都

必须添加语句 f.Close()关闭文件，以释放系统资源。读者是否能通过上述例子，总结出 defer 语句的优点？

5.5　递归函数

什么是算法？简单地说，算法是解决问题的方法与步骤。递归算法(Recursive Algorithm)的核心思想是分治策略。分治是“分而治之”(Divide and Conquer)的意思。分治策略将一个复杂的问题反复分解为两个或更多个相同的或相似的子问题，直至这些子问题可以直接求解，最后将子问题的解合并起来，就能得到原问题的解，如图 5-1 所示。

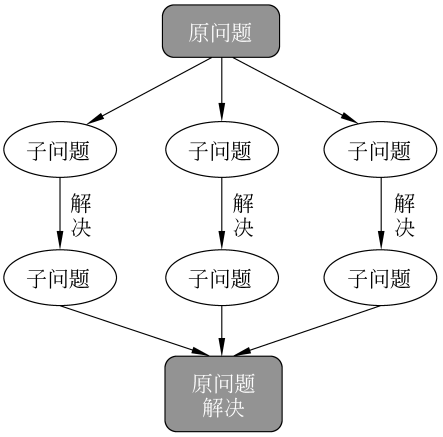

图 5-1　分治策略

一个函数在其函数体内部调用它自身，这种函数叫作递归函数。递归函数使用了分治策略，其由终止条件和递归条件两部分组成。下面定义一个计算阶乘的函数 factorial(n)：

```
func factorial(n int) int {                    //阶乘 factorial
    if n <= 1 {                                //终止条件
        return 1
    }
    return n * factorial(n-1)                  //递归条件
}
func main() {
    var n int = 5
    fmt.Println(factorial(n))                  //输出 120
}
```

调用上述定义的 factorial(n)函数计算 5 的阶乘，其执行流程如下所示：

```
factorial(5)  =  5 * factorial(4)
              =  5 * 4 * factorial(3)
              =  5 * 4 * 3 * factorial(2)
              =  5 * 4 * 3 * 2 * factorial(1)
              =  5 * 4 * 3 * 2 * 1
              =  120
```

下面定义一个计算斐波那契数列(Fibonacci)的递归函数 fib(n)：

```
func fib(n int) int {
    if n <= 1 {                                  //终止条件
        return n
    }
    return fib(n-1) + fib(n-2)                   //递归条件
}
func main() {
    var n int = 10
    for i := 0; i <= n; i++ {
        fmt.Printf("%d ", fib(i))
    }
    fmt.Println()
}
```

上述代码的输出结果：

```
0 1 1 2 3 5 8 13 21 34 55
```

5.6 可变长度参数

与 Python 语言类似，Go 语言也支持可变长度参数。也就是说，一个函数可以接收任意数量的实际参数。

举例：

```
func multiply(args …int) int {               //形参类型是 int 型
    z := 1
    for _, arg := range args {
        z *= arg
    }
    return z
}
func main() {
    fmt.Println(multiply(2, 3, 4))           //输出 24
    fmt.Println(multiply(4, 5))              //输出 20
    fmt.Println(multiply(10, 9))             //输出 90
}
```

…int 本质上是一个切片，也就是[]int，因此可以将其用在 for 循环中。如果想传入任意类型的数据，则需要将 int 修改为空接口 interface{}。

举例：

```
func what(args …interface{}) {
    for _, arg := range args {
        switch arg.(type) {
        case int:
            fmt.Println(arg, "int")
        case int64:
```

```
            fmt.Println(arg, "int64")
        case string:
            fmt.Println(arg, "string")
        default:
            fmt.Println(arg, "unknown")
        }
    }
}
func main() {
    var val1 int = 1
    var val2 int64 = 2
    var val3 string = "good"
    var val4 float32 = 1.5
    what(val1, val2, val3, val4)
}
```

上述代码的输出结果：

```
1 int
2 int64
good string
1.5 unknown
```

5.7　方法

函数和方法分别是面向过程和面向对象编程范畴的概念。从某种角度说，Go 是将两种编程理念融为一体的语言。那么，在 Go 语言中怎样定义方法呢？定义方法的语法格式如下：

```
func (receiver receiverType) methodName ( [arg argType] )( [returnType] ){
    method body
}
```

其中，receiver 是接受者的名称，receiverType 是接受者的类型，methodName 是方法名，arg 是形式参数，argType 是形式参数的类型，returnType 是返回值的类型。显然，函数与方法的区别之一是方法有接受者。在 Go 语言中，方法的接受者可以是结构体。

举例：

```
type User struct {                          //定义结构体 User
    name string
    email string
}
func (u User) userInfo() string {           //方法的接受者是 User 类型
    return fmt.Sprintf("User name: %s and email: %s\n", u.name, u.email)
}
func main() {
    user1 := User{name: "Hui", email: "whui2008@tust.edu.cn"}
    fmt.Print(user1.userInfo())
}
```

上述代码的输出结果：

```
User name: Hui and email: whui2008@tust.edu.cn
```

在上述代码中，u 是接受者的名称，User 是接受者的类型(结构体)，userInfo 是方法名，形式参数及其类型为空，返回值的类型是 string。

方法的接受者也可以是非结构体类型，如 int。Go 语言要求方法 methodName()与接受者类型 receiverType 必须定义在同一个包内。

举例：

```
type myNumber int                                    //定义整数类型 myNumber
func (num myNumber) square() myNumber {
    if num <= 1 {
        return 1
    }
    return num * num
}
func main() {
    var n myNumber = 15
    result := n.square()
    fmt.Printf("The square of %d is %d.\n", n, result)
}
```

上述代码的输出结果：

```
The square of 15 is 225.
```

如果读者删除代码行 type myNumber int，则编译器会给出错误信息“cannot define new methods on non-local type int”。

在 Go 语言中，函数也是一种数据类型，与其他数据类型一样可以将其赋值给变量。

举例：

```
func demo() {
    fmt.Println("in demo()")
}
func main() {
    //声明变量 f 为 func()函数类型，此时 f = nil
    var f func()
    f = demo
    f()
}
```

上述代码的输出结果：

```
in demo()
```

5.8 小结

函数是执行计算的命名语句序列，而方法则是有接受者的函数。定义函数，使用关键字 func。函数定义完毕后并不能自动运行，只有被调用时才能运行。参数分为形参和实参，形