函数与指针

目录：

一、函数

（一）、什么是函数

1、函数是组织好的、可重复使用的执行特定任务的代码块。它可以提高应用程序的模块性和代码的重复利用率。

2、Go语言支持普通函数、匿名函数和闭包，从设计上对函数进行了优化和改进，让函数使用起来更加方便。

3、Go语言的函数属于一等公民（first-class）：

- 函数本身可以作为值进行传递；
- 支持匿名函数和闭包（closure）；
- 函数可以满足接口。

（二）、声明函数

1、函数声明的作用

- 普通函数需要先声明才能调用，一个函数的声明包括参数和函数名等。编

译器通过声明才能了解函数应该怎样在调用代码和函数体之间传递参数和返回参数。

2、语法格式：

```
func 函数名（参数列表）（返回参数列表）{
    //函数体
}
```

```
func  funcName (parametername type1, parametername type2...) (output1 type1, output2 type2...) {
    //逻辑代码
    //返回多个值
    return value1, value2...
 }
```

3、函数定义解析：

- func：函数关键字。
  - 函数由 func 开始声明
- funcName：函数名。
  - 函数名和参数列表一起构成了函数签名。
  - 函数名由字母、数字和下划线组成。函数名的第一个字母不能为数字。在同一个包内，函数名称不能重名。
- parametername type：参数列表。
  - 参数就像一个占位符，定义函数时的参数叫做形式参数，形参变量是函数的局部变量；当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。
  - 参数列表指定的是参数类型、顺序、及参数个数。
  - 参数是可选的，也就是说函数也可以不包含参数。
  - 参数类型的简写
    - 在参数列表中，如果有多个参数变量，则以逗号分隔；如果相邻变量是同类型，则可以将类型省略。
    - 例如：func add （a , b int） {}
  - Go语言的函数支持可变参数。接受变参的函数是有着不定数量的参数的。

```
func myfunc(arg ...int) {}
```

`arg ...int`告诉Go这个函数接受不定数量的参数。注意，这些参数的类型全部是int。在函数体中，变量arg是一个int的slice。

- output1 type1, output2 type2：返回值列表。
  - 返回值返回函数的结果，结束函数的执行。
  - Go语言的函数可以返回多个值。
  - 返回值可以是：返回数据的数据类型，或者是：变量名+变量类型的组合。
  - 函数声明时有返回值，必须在函数体中使用return语句提供返回值列表。
  - 如果只有一个返回值且不声明返回值变量，那么可以省略包括返回值的括号。
  - return后的数据，要保持和声明的返回值类型、数量、顺序一致。
  - 如果函数没有声明返回值，函数中也可以使用return关键字，用于强制结束函数。
- 函数体：函数定义的代码集合，是能够被重复调用的代码片段。

（三）、变量作用域

1、概述

- 作用域是变量、常量、类型、函数的作用范围。
- Go 语言中变量可以在三个地方声明：
  - 函数内定义的变量称为局部变量
  - 函数外定义的变量称为全局变量
  - 函数中定义的参数称为形式参数

2、局部变量

- 在函数体内声明的变量称之为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

3、全局变量

- 在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。

全局变量可以在任何函数中使用。Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑。

4、形式参数

- 形式参数会作为函数的局部变量来使用。

5、案例分析

```
package main
import "fmt"
/* 声明全局变量 */
var a1 int = 7
var b1 int = 9

func main() {
   /* main 函数中声明局部变量 */
   a1, b1, c1 := 10, 20, 0

   fmt.Printf("main()函数中 a1 = %d\n", a1) //10
   fmt.Printf("main()函数中 b1 = %d\n", b1) //20
   fmt.Printf("main()函数中 c1 = %d\n", c1) //0
   c1 = sum(a1, b1)
   fmt.Printf("main()函数中 c1 = %d\n", c1) //33
}

/* 函数定义-两数相加 */
func sum(a1, b1 int) (c1 int) {
   a1++
   b1 += 2
   c1 = a1 + b1
   fmt.Printf("sum() 函数中 a1 = %d\n", a1) //11
   fmt.Printf("sum() 函数中 b1 = %d\n", b1) //22
   fmt.Printf("sum() 函数中 c1 = %d\n", c1) //33
   return c1
}
```

输出结果：

```
main()函数中 a1 = 10
main()函数中 b1 = 20
```

main()函数中 c1 = 0
sum() 函数中 a1 = 11
sum() 函数中 b1 = 22
sum() 函数中 c1 = 33
main()函数中 c1 = 33

（四）、函数变量（函数作为值）

- 在Go语言中，函数也是一种类型，可以和其它类型一样被保存在变量中。
- 可以通过type来定义一个自定义类型。函数的参数完全相同（包括：参数类型、个数、顺序），函数返回值相同。

1、案例代码一：

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

func main() {
  result := StringToLower("AbcdefGHijklMNOPqrstUVWxyz", processCase)
  fmt.Println(result)
  result = StringToLower2("AbcdefGHijklMNOPqrstUVWxyz", processCase)
  fmt.Println(result)
}

//处理字符串，奇数偶数依次显示为大小写
func processCase(str string) string {
  result := ""
  for i, value := range str {
    if i%2 == 0 {
      result += strings.ToUpper(string(value))
    } else {
      result += strings.ToLower(string(value))
    }
```

```
    }
    return result
}

func StringToLower(str string, f func(string) string) string {
    fmt.Printf("%T \n", f)
    return f(str)
}

type caseFunc func(string) string // 声明了一个函数类型。通过type关键字,
caseFunc会形成一种新的类型。
func StringToLower2(str string, f caseFunc) string {
    fmt.Printf("%T \n", f)
    return f(str)
}
```

2、案例代码二：

```
package main
import "fmt"
type processFunc func(int) bool // 声明了一个函数类型
func main() {
    slice := []int{1, 2, 3, 4, 5, 7}
    fmt.Println("slice = ", slice)
    odd := filter(slice, isOdd) // 函数当做值来传递
    fmt.Println("奇数元素: ", odd)
    even := filter(slice, isEven) // 函数当做值来传递
    fmt.Println("偶数元素: ", even)
}
//判断元素是否是偶数
func isEven(integer int) bool {
    if integer%2 == 0 {
        return true
    }
    return false
```

```
}
//判断元素是否是奇数
func isOdd(integer int) bool {
  if integer%2 == 0 {
    return false
  }
  return true
}
//根据函数来处理切片，根据元素奇数偶数分组，返回新的切片
func filter(slice []int, f processFunc) []int {
  var result []int
  for _, value := range slice {
    if f(value) {
      result = append(result, value)
    }
  }
  return result
}
```

3、函数变量的使用步骤及意义：

1. 定义一个函数类型
2. 实现定义的函数类型
3. 作为参数调用

- 函数变量的用法类似接口的用法。
- 函数当做值和类型在写一些通用接口的时候非常有用，通过上面例子可以看到processFunc这个类型是一个函数类型，然后两个filter函数的参数和返回值与processFunc类型是一样的。用户可以实现很多种的逻辑，这样使得程序变得非常的灵活。

（五）、匿名函数

1、概念

- Go语言支持匿名函数，即在需要使用函数时，再定义函数，匿名函数没有函数名，只有函数体，函数可以被作为一种类型被赋值给变量，匿名函数也往往以变量方式被传递。

- 匿名函数经常被用于实现回调函数、闭包等。

2、定义格式

```
func(参数列表）（返回参数列表） {
    //函数体
}
```

3、定义匿名函数

(1)、在定义时调用匿名函数

```
package main
import "fmt"
func main() {
  func(data int) {
    fmt.Println("hello" , data)
  }(100)
}
```

(2)、将匿名函数赋值给变量

```
package main
import "fmt"
func main() {
  f:= func(data string) {
    fmt.Println(data)
  }
  f("欢迎学习Go语言！")
}
```

4、匿名函数的用法——作回调函数

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

func main() {
```

```
    //调用函数，对每个元素进行求平方根操作
    arr := []float64{1, 9, 16, 25, 30}
    visit(arr, func(v float64) {
        v = math.Sqrt(v)
        fmt.Printf("%.2f \n", v)
    })

    //调用函数，对每个元素进行求平方操作
    visit(arr, func(v float64) {
        v = math.Pow(v , 2)
        fmt.Printf("%.0f \n", v)
    })
}

//定义一个函数，遍历切片元素，对每个元素进行处理
func visit(list []float64, f func(float64)) {
    for _, value := range list {
        f(value)
    }
}
```

（六）、闭包

1、概念：

- 闭包并不是什么新奇的概念，它早在高级语言开始发展的年代就产生了。闭包（Closure）是词法闭包（Lexical Closure）的简称。对闭包的具体定义有很多种说法，大体可以分为两类：
    - 一种说法认为闭包是符合一定条件的函数，比如这样定义闭包：闭包是在其词法上下文中引用了自由变量的函数。
    - 另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。比如这样的定义：在实现深约束时，需要创建一个能显式表示引用环境的东西，并将它与相关的子程序捆绑在一起，这样捆绑起来的整体被称为闭包。函数 + 引用环境 = 闭包
- 上面的定义，一个认为闭包是函数，另一个认为闭包是函数和引用环境组

成的整体。显然第二种说法更确切。闭包只是在形式和表现上像函数，但实际上不是函数。

- 函数是一些可执行的代码，这些代码在函数被定义后就确定了，不会在执行时发生变化，所以一个函数只有一个实例。
- 闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

- 闭包在某些编程语言中被称为Lambda表达式。
- 函数本身不存储任何信息，只有与引用环境结合后形成的闭包才具有“记忆性”。函数是编译器静态的概念，而闭包是运行期动态的概念。
- 对象是附有行为的数据，而闭包是附有数据的行为。

2、闭包的价值

（1）、加强模块化

- 闭包有益于模块化编程，它能以简单的方式开发较小的模块，从而提高开发速度和程序的可复用性。和没有使用闭包的程序相比，使用闭包可将模块划分得更小。
- 比如我们要计算一个数组中所有数字的和，这只需要循环遍历数组，把遍历到的数字加起来就行了。如果现在要计算所有元素的积呢？要打印所有的元素呢？解决这些问题都要对数组进行遍历，如果是在不支持闭包的语言中，我们不得不一次又一次重复地写循环语句。而这在支持闭包的语言中是不必要的。这种处理方法多少有点像回调函数，不过要比回调函数写法更简单，功能更强大。

（2）、抽象

- 闭包是数据和行为的组合，这使得闭包具有较好抽象能力。

（3）、简化代码

3、一个编程语言需要哪些特性来支持闭包呢?

- 函数是一阶值（First-class value，一等公民），即函数可以作为另一个函数的返回值或参数，还可以作为一个变量的值。
- 函数可以嵌套定义，即在一个函数内部可以定义另一个函数。
- 允许定义匿名函数。
- 可以捕获引用环境，并把引用环境和函数代码组成一个可调用的实体；

4、案例代码一

（1）、没有使用闭包进行计数的代码

```
package main
import "fmt"
func main() {
  for i := 0; i < 5; i++ {
    fmt.Printf("i=%d \t", i)
    fmt.Println(add2(i))
  }
}
func add2(x int) int {
  sum := 0
  sum += x
  return sum
}
```

运行结果：

```
i=0   0
i=1   1
i=2   2
i=3   3
i=4   4
```

for循环每执行一次，sum都会清零，没有实现sum累加计数。

（2）、使用闭包函数实现计数器：

```
package main
import "fmt"
func main() {
  pos := adder()
  for i := 0; i < 10; i++ {
    fmt.Printf("i=%d \t", i)
    fmt.Println(pos(i))
  }
  fmt.Println("----------------------")
  for i := 0; i < 10; i++ {
```

```
        fmt.Printf("i=%d \t", i)
        fmt.Println(pos(i))
    }
}
func adder() func(int) int {
    sum := 0
    return func(x int) int {
        fmt.Printf("sum1=%d \t", sum)
        sum += x
        fmt.Printf("sum2=%d \t", sum)
        return sum
    }
}
```

运行结果为：

```
i=0   sum1=0   sum2=0  0
i=1   sum1=0   sum2=1  1
i=2   sum1=1   sum2=3  3
i=3   sum1=3   sum2=6  6
i=4   sum1=6   sum2=10       10
---------------------
i=0   sum1=10        sum2=10       10
i=1   sum1=10        sum2=11       11
i=2   sum1=11        sum2=13       13
i=3   sum1=13        sum2=16       16
i=4   sum1=16        sum2=20       20
```

5、案例代码二

```
package main
import "fmt"
func main() {
    myfunc := Counter()
    //fmt.Printf("%T\n", myfunc)
    fmt.Println("myfunc", myfunc)
```

```
    /* 调用 myfunc 函数，i 变量自增 1 并返回 */
    fmt.Println(myfunc())
    fmt.Println(myfunc())
    fmt.Println(myfunc())

    /* 创建新的函数 nextNumber1，并查看结果 */
    myfunc1 := Counter()
    fmt.Println("myfunc1", myfunc1)
    fmt.Println(myfunc1())
    fmt.Println(myfunc1())
}

//计数器.闭包函数
func Counter() func() int {
    i := 0
    res := func() int {
        i += 1
        return i
    }
    //fmt.Printf("%T , %v \n" , res , res）//func() int , 0x1095af0
    fmt.Println("Counter中的内部函数:", res) //0x1095af0
    return res
}
```

(七)、可变参数

1、如果一个函数的参数，类型一致，但个数不定，可以使用函数的可变参数。

2、语法格式:

```
func 函数名(参数名 ...类型) [(返回值列表)] {
    //函数体
}
```

- 该语法格式定义了一个接受任何数目、任何类型参数的函数。这里特殊的语法是三个点“...”，在一个变量后面加上三个点后，表示从该处开始接受不定参数。

- 当要传递若干个值到不定参数函数中得时候，可以手动书写每个参数，也可以将一个slice传递给该函数，通过"..."可以将slice中的参数对应的传递给函数。

3、案例代码：计算学员考试总成绩及平均成绩

```
package main
import (
  "fmt"
)
func main() {
  //1、传进n个参数
  sum, avg, count := GetScore(90, 82.5, 73, 64.8)
  fmt.Printf("学员共有%d门成绩，总成绩为：%.2f，平均成绩为：%.2f", count, sum, avg)
  fmt.Println()

  // 2、传切片作为参数
  scores := []float64{92, 72.5, 93, 74.5, 89, 87, 74}
  sum, avg, count = GetScore(scores...)
  fmt.Printf("学员共有%d门成绩，总成绩为：%.2f，平均成绩为：%.2f", count, sum, avg)
}

//累加求和，参数个数不定，参数个数从0-n
func GetScore(scores ...float64) (sum, avg float64, count int) {
  for _, value := range scores {
    sum += value
    count++
  }
  avg = sum / float64(count)
  return
}
```

4、可变参数注意细节：

- 一个函数最多只能有一个可变参数
- 参数列表中还有其它类型参数，则可变参数写在所有参数的最后

（八）、递归函数

1、在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

- 递归函数必须满足以下两个条件：

1） 在每一次调用自己时，必须是（在某种意义上）更接近于解；

2） 必须有一个终止处理或计算的准则。

2、案例代码：求阶乘

- 计算阶乘n! = 1 x 2 x 3 x ... x n，用函数fact(n)表示，可以看出：fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n。所以，fact(n)可以表示为n x fact(n-1)，只有n=1时需要特殊处理。

```
package main
import "fmt"
func main() {
  fmt.Println(factorial(5))
}
//通过递归实现阶乘
func factorial(n int) int {
  if n == 0 {
    return 1
  }
  return n * factorial(n-1)
}

//通过循环实现阶乘
func getMultiple(num int) (result int) {
  result = 1
  for i:=1; i<= num; i++ {
    result *= i
  }
```

```
  return
}
```

3、使用递归的注意事项

- 递归的计算过程

```
===> 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)))
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120
```

- 递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以用循环的方式实现，但循环的逻辑不如递归清晰。
- 使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈，每当函数返回，栈就会减一层。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。
- 使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

二、指针

（一）、指针的概述

1、指针是存储另一个变量的内存地址的变量。

- 变量是一种使用方便的占位符，变量都指向计算机的内存地址。
- 一个指针变量可以指向任何一个值的内存地址。
- 如下图：变量b的值为156，存储在内存地址0x1040a124。变量a持有b的地址，则a被认为指向b。

2、获取变量的地址

- Go 语言的**取地址符&**，一个变量前使用&，会返回该变量的内存地址。

```
func main() {
  a := 10
  fmt.Printf("变量的地址: %x \n", &a)
}
```

运行结果：

变量的地址： c420014050

3、Go语言指针的特点：

- Go语言指针的最大特点是：指针不能运算（不同于C语言）。
- 在Go语言中如果对指针进行运算会报错：nvalid operation: p++ (non-numeric type *int)

（二）、声明指针

1、声明指针，*T是指针变量的类型，它指向T类型的值。

var 指针变量名 ***指针类型**

- * 号用于指定变量是一个指针。
- var ip *int　　//指向整型的指针
- var fp *float32　//指向浮点型的指针

2、如何使用指针？（指针使用流程）

- 定义指针变量。
- 为指针变量赋值。
- 访问指针变量中指向地址的值。
- 获取指针的值：在指针类型的变量前加上 * 号（前缀）来获取指针所指向

的内容。

获取一个指针意味着访问指针指向的变量的值。语法是：*a

3、示例代码一：

```
func main() {
  //声明实际变量
  var a int = 120

  //声明指针变量
  var ip *int

  //给指针变量赋值，将变量a的地址赋值给ip
  ip = &a

  //打印a的类型和值
  fmt.Printf("a 的类型是%T，值是%v \n", a, a)

  //打印&a的类型和值
  fmt.Printf("&a 的类型是%T，值是%v \n", &a, &a)

  //打印ip的类型和值
  fmt.Printf("ip 的类型是%T，值是%v \n", ip, ip)

  //打印变量*ip的类型和值
  fmt.Printf("*ip 变量的类型是%T，值是%v \n", *ip, *ip)

  //打印变量*&a的类型和值
  fmt.Printf("*&a 变量的类型是%T，值是%v \n", *&a, *&a)

  fmt.Println(a, &a, *&a)
  fmt.Println(ip, &ip, *ip, *(&ip), &(*ip))
}
```

运行结果：

- a 的类型是int，值是120

- &a 的类型是*int，值是0xc420014050
- ip 的类型是*int，值是0xc420014050
- *ip 变量的类型是int，值是120
- *&a 变量的类型是int，值是120
- 120 0xc420014050 120
- 0xc420014050 0xc42000c028 120 0xc420014050 0xc420014050

4、示例代码二：

```
package main
import "fmt"
type Student struct {
  name    string
  age     int
  married bool
  sex     int8
}

func main() {
  var s1 = Student{"Steven", 35, true, 1}
  var s2 = Student{"Sunny", 20, false, 0}

  var a *Student = &s1 //将s1的内存地址赋值给Student指针变量a
  var b *Student = &s2 //将s2的内存地址赋值给Student指针变量b

  fmt.Println("\n--------------------")
  fmt.Printf("s1类型为%T，值为%v \n", s1, s1)
  fmt.Printf("s2类型为%T，值为%v \n", s2, s2)

  fmt.Println("\n--------------------")
  fmt.Printf("a类型为%T，值为%v \n", a, a)
  fmt.Printf("b类型为%T，值为%v \n", b, b)

  fmt.Println("\n--------------------")
  fmt.Printf("*a类型为%T，值为%v \n", *a, *a)
```

```
    fmt.Printf("*b类型为%T，值为%v \n", *b, *b)

    fmt.Println("\n---------------------")
    fmt.Println(s1.name, s1.age, s1.married, s1.sex)
    fmt.Println(a.name, a.age, a.married, a.sex)

    fmt.Println("\n---------------------")
    fmt.Println(s2.name, s2.age, s2.married, s2.sex)
    fmt.Println(b.name, b.age, b.married, b.sex)

    fmt.Println("\n---------------------")
    fmt.Println((*a).name, (*a).age, (*a).married, (*a).sex)
    fmt.Println((*b).name, (*b).age, (*b).married, (*b).sex)

    fmt.Println("\n---------------------")
    fmt.Printf("&a类型为%T，值为%v\n", &a, &a)
    fmt.Printf("&b类型为%T，值为%v\n", &b, &b)

    fmt.Println("\n---------------------")
    fmt.Println(&a.name, &a.age, &a.married, &a.sex)
    fmt.Println(&b.name, &b.age, &b.married, &b.sex)
}
```

运行结果：

```
---------------------
s1类型为main.Student，值为{Steven 35 true 1}
s2类型为main.Student，值为{Sunny 20 false 0}

---------------------
a类型为*main.Student，值为&{Steven 35 true 1}
b类型为*main.Student，值为&{Sunny 20 false 0}

---------------------
*a类型为main.Student，值为{Steven 35 true 1}
```

*b类型为main.Student，值为{Sunny 20 false 0}

Steven 35 true 1

Steven 35 true 1

Sunny 20 false 0

Sunny 20 false 0

Steven 35 true 1

Sunny 20 false 0

&a类型为**main.Student，值为0xc42000c028

&b类型为**main.Student，值为0xc42000c030

0xc42000a060 0xc42000a070 0xc42000a078 0xc42000a079

0xc42000a080 0xc42000a090 0xc42000a098 0xc42000a099

（四）、空指针

1、Go 空指针

- 当一个指针被定义后没有分配到任何变量时，它的值为 nil。
- nil 指针也称为空指针。
- nil在概念上和其它语言的null、None、NULL一样，都指代零值或空值。
- 一个指针变量通常缩写为 ptr。

2、空指针判断：

- `if(ptr != nil)     // ptr 不是空指针`
- `if(ptr == nil)    // ptr 是空指针`

（五）、操作指针改变变量的数值

1、示例代码：

```
package main
import (
  "fmt"
)
func main() {
  b := 3158
  a := &b
  fmt.Println("b 的地址：", a) //0xc420014050
  fmt.Println("*a 的值：", *a) //3158
  *a++
  fmt.Println("b 的新值：", b)//3159
}
```

运行结果

```
b 的地址： 0xc420014050
*a 的值：3158
b 的新值： 3159
```

（六）、使用指针作为函数的参数

1、示例代码一（基本数据类型指针作为函数参数）

```
package main
import (
  "fmt"
)
func main() {
  a := 58
  fmt.Println("函数调用之前a的值：", a)
  fmt.Printf("%T \n", a)
  fmt.Printf("%x \n", &a)
  //b := &a
  var b *int = &a

  change(b)
```

```
  fmt.Println("函数调用之后的a的值：", a)
}
func change(val *int) {
  *val = 15
}
```

运行结果

- 函数调用之前a的值： 58
- int
- c420014050
- 函数调用之后的a的值： 15

2、示例代码二：

```
package main
import "fmt"
func main() {
  /* 定义局部变量 */
  a := 100
  b := 200

  //返回值的写法
  a, b = swap0(a, b)

  //指针作为参数的写法
  swap(&a, &b)

  fmt.Printf("交换后 a 的值 : %d\n", a)
  fmt.Printf("交换后 b 的值 : %d\n", b)
}

 //具有返回值的惯用写法
func swap0(x, y int) (int , int) {
  return  y, x
}
```

```
//指针作为参数的写法
func swap(x *int, y *int) {
  *x, *y = *y, *x
}
```

3、示例代码二：（将一个指向切片的指针传递给函数）

- 虽然将指针传递给一个切片作为函数的参数，可以实现对该切片中元素的修改，但这并不是实现这一目标的惯用方法。惯用做法是使用切片。

```
package main
import  "fmt"

func main() {
  a := [3]int{89, 90, 91}
  modify(&a)
  fmt.Println(a)
}

func modify(arr *[3]int) {
  (*arr)[0] = 189
}
```

运行结果

```
[189 90 91]
```

【备注：】建议做法是使用切片来实现更改元素数值，代码如下：

```
package main
import "fmt"
func main() {
  a := []int{89, 90, 91}
  modify(a[:])
  fmt.Println(a)
}

func modify(sls []int) {
  sls[0] = 190
```

}

（七）、指针数组

1、指针数组：就是元素为指针类型的数组。

- 定义一个指针数组，例如：var ptr [3]*string ；
- 有一个元素个数相同的数组，将该数组中每个元素的地址赋值给该指针数组。也就是说该指针数组与某一个数组完全对应。
- 可以通过*指针变量获取到该地址所对应的数值。

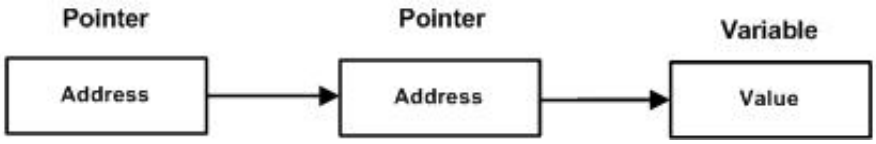
2、案例代码：

```
package main

import "fmt"

const COUNT int = 4

func main() {
  a := [COUNT]string{"abc", "ABC", "123", "一二三"}
  i := 0

  //定义指针数组
  var ptr [COUNT]*string

  fmt.Printf("%T , %v \n", ptr, ptr)

  for i = 0; i < COUNT; i++ {
    //将数组中每个元素的地址赋值给指针数组
    ptr[i] = &a[i]
  }
  fmt.Printf("%T , %v \n", ptr, ptr)

  //获取指针数组中第一个值，其实就是一个地址
  fmt.Println(ptr[0])
```

```
  //根据数组元素的每个地址获取该地址所指向的元素的数值
  for i = 0; i < COUNT; i++ {
    fmt.Printf("a[%d] = %s \n", i, *ptr[i])
  }
}
```

（八）、指针的指针

1、如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。
当定义一个指向指针的指针变量时，第一个指针存放第二个指针的地址，第二个指针存放变量的地址：



2、指向指针的指针变量声明格式如下：

- var ptr **int
- 以上指向指针的指针变量为整型。
- 访问指向指针的指针变量值需要使用两个 * 号。

3、案例代码

```
package main
import "fmt"
func main() {
  var a int
  var ptr *int
  var pptr **int
  a = 1234
  /* 指针 ptr 地址 */
  ptr = &a
  fmt.Println("ptr" , ptr)
  /* 指向指针 ptr 地址 */
  pptr = &ptr
  fmt.Println("pptr" , ptr)
```

```
    /* 获取 pptr 的值 */
    fmt.Printf("变量 a = %d\n", a)
    fmt.Printf("指针变量 *ptr = %d\n", *ptr)
    fmt.Printf("指向指针的指针变量 **pptr = %d\n", **pptr)
}
```

运行结果：

- ptr 0xc420014050
- pptr 0xc420014050
- 变量 a = 1234
- 指针变量 *ptr = 1234
- 指向指针的指针变量 **pptr = 1234

三、 函数的参数传递

- 函数如果使用参数，该参数变量称为函数的形参。形参就像定义在函数体内的局部变量。调用函数，可以通过两种方式来传递参数。即：值传递和引用传递，或者叫做传值和传引用。

（一）、值传递（传值）

1、概念：

- 值传递：是指在调用函数时将实际**参数复制**一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到原内容数据。
- 默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到原内容数据。
- 每次调用函数，都将实参拷贝一份再传递到函数中。每次拷贝一份，性能是不是就下降了呢？其实Go语言中使用指针和值传递配合就避免了性能降低问题，也就是通过传指针参数来解决实参拷贝的问题。

（二）、引用传递（传引用）

1、概念：

- 引用传递：是指在调用函数时将实际参数的**地址传递**到函数中，那么在函数中对参数所进行的修改，将影响到原内容数据。
- 严格来说Go语言只有值传递一种传参方式，Go语言是没有引用传递的。
- Go语言中可以借助传指针来实现引用传递的效果。函数参数使用指针参

数，传参时其实是在拷贝一份指针参数，也就是拷贝了一份变量地址。

- 函数的参数如果是指针，当函数调用时，虽然参数仍然是按拷贝传递的，但是此时仅仅只是拷贝一个指针，也就是一个内存地址，这样就不用担心实参拷贝造成的内存浪费、时间开销、性能降低的情况。

2、引用传递的作用：

- 传指针使得多个函数能操作同一个对象。
- 传指针更轻量级 (8bytes)，只需要传内存地址。如果参数是非指针参数，那么值传递的过程中，每次在拷贝上面就会花费相对较多的系统开销（内存和时间）。所以当要传递大的结构体的时候，用指针是一个明智的选择。
- Go语言中slice、map、chan类型的实现机制都是类似指针，所以可以直接传递，而不必取地址后传递指针。

3、示例代码一：函数传传int型参数

```
package main
import "fmt"
/*
  ● 值传递：是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，不会影响到实际参数。
  ● 引用传递：是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。
  ● 严格来说Go语言只有值传递一种传参方式，Go语言是没有引用传递的。
  ● Go语言中可以借助传指针来实现引用传递的效果。函数参数使用指针参数，传参时其实是在拷贝一份指针参数，也就是拷贝了一份变量地址。
 */

func main() {
  a := 10
  fmt.Printf("1、变量a的内存地址：%p ， 值为：%v \n\n", &a, a) //10
  fmt.Printf("=========int型变量a的内存地址：%p \n\n", a)  //? ? ? %!p

  //传值
  changeIntVal(a)
  fmt.Printf("2、changeIntVal函数调用之后：变量a的内存地址：%p ， 值为：
```

```
%v \n\n", &a, a) //10

  //传引用
  changeIntPtr(&a)
  fmt.Printf("3、changeIntPtr函数调用之后：变量a的内存地址：%p ， 值为：
%v \n\n", &a, a) //50
}

func changeIntVal(a int) {
  fmt.Printf("--------changeIntVal函数内：值参数a的内存地址：%p ， 值为：
%v \n", &a, a) //10
  a = 90
}

func changeIntPtr(a *int) {
  fmt.Printf("--------changeIntPtr函数内：指针参数a的内存地址：%p ， 值
为：%v \n", &a, a) //地址
  *a = 50
}
```

4、示例代码二：函数传值和传引用_传slice型参数

```
package main
import "fmt"
func main() {
  a := []int{1, 2, 3, 4}
  fmt.Printf("1、变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//[1,2,3,4]
  fmt.Printf("切片型变量a内存地址是：%p \n\n", a)//可以获取到地址，类似：
0xc420018080

  //传值
  changeSliceVal(a)
  fmt.Printf("2、changeSliceVal函数调用后：变量a的内存地址是：%p ， 值
为：%v \n\n", &a, a)//[1,2,3,4]
```

```
  //传引用
  changeSlicePtr(&a)
  fmt.Printf("3、changeSlicePtr函数调用后：变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//[250,2,3,4]
}

func changeSliceVal(a []int) {
  fmt.Printf("----------changeSliceVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //[1,2,3,4]
  fmt.Printf("----------changeSlicePtr函数内：值参数a的内存地址是：%p \n", a)
  a[0] = 99
}

func changeSlicePtr(a *[]int) {
  fmt.Printf("----------changeSlicePtr函数内：指针参数a的内存地址是：%p ， 值为：%v \n", &a, a) //&[1,2,3,4]
  (*a)[1] = 250
}
```

5、示例代码三：函数传值和传引用_传数组

```
package main
import "fmt"
func main() {
  a := [4]int{1, 2, 3, 4}
  fmt.Printf("1、变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//[1,2,3,4]
  fmt.Printf("数组型变量a内存地址是：%p \n\n", a)//可以获取到地址？ ✘

  //传值
  changeArrayVal(a)
  fmt.Printf("2、changeArrayVal函数调用后：变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//[99,2,3,4] ✘

  //传引用
```

```
  changeArrayPtr(&a)
  fmt.Printf("3、changeArrayPtr函数调用后：变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//[99,250,3,4] ❌
}

func changeArrayVal(a [4]int) {
  fmt.Printf("----------changeArrayVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //[1,2,3,4]
  fmt.Printf("----------changeArrayPtr函数内：值参数a的内存地址是：%p \n", a) //获取不到地址
  a[0] = 99
}

func changeArrayPtr(a *[4]int) {
  fmt.Printf("----------changeArrayPtr函数内：指针参数a的内存地址是：%p ， 值为：%v \n", &a, a) //&[1,2,3,4]
  (*a)[1] = 250
}
```

6、示例代码三：函数传值和传引用_传struct结构体

```
package main
import "fmt"
type Teacher struct {
  name    string
  age     int
  married bool
  sex     int8
}

func main() {
  a := Teacher{"Steven", 35, true, 1}
  fmt.Printf("1、变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//{Steven 35 true 1}
  fmt.Printf("struct型变量a内存地址是：%p \n\n", a)//可以获取到地址?
```

```
  //传值
  changeStructVal(a)
  fmt.Printf("2、changeArrayVal函数调用后：变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//{Steven 35 true 1}

  //传引用
  changeStructPtr(&a)
  fmt.Printf("3、changeArrayPtr函数调用后：变量a的内存地址是：%p ， 值为：%v \n\n", &a, a)//
}

func changeStructVal(a Teacher) {
  fmt.Printf("----------changeArrayVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //
  fmt.Printf("----------changeArrayPtr函数内：值参数a的内存地址是：%p \n", a) //获取不到地址？
  a.name = "Josh"
  a.age = 29
  a.married = false
}

func changeStructPtr(a *Teacher) {
  fmt.Printf("----------changeArrayPtr函数内：指针参数a的内存地址是：%p ， 值为：%v \n", &a, a) //{Daniel 20 false 1}
  (*a).name = "Daniel"
  (*a).age = 20
  (*a).married = false
}
```

（三）、值传递和引用传递的注意细节【重要】

1、Go语言中所有的传参都是值传递（传值），都是一个副本，一个拷贝。

- 拷贝的内容有时候是值类型（int、string、bool、数组、struct属于值类

型），这样就在函数中就无法修改原内容数据；

- 有的是引用类型（指针、slice、map、chan属于引用类型），这样就可以修改原内容数据。

2、是否可以修改原内容数据，和传值、传引用没有必然的关系。在C++中，传引用肯定是可以修改原内容数据的，在Go语言里，虽然只有传值，但是我们也可以修改原内容数据，因为参数可以是引用类型。

3、传引用和引用类型是两个概念。虽然Go语言只有传值一种方式，但是可以通过传引用类型变量达到跟传引用一样的效果。