# Go内置容器——数组和切片

目录：



## 一、概述

1、基本数据类型（原生数据类型）： 整型、浮点型、布尔型、字符串、字符（byte、rune）

2、复合数据类型（派生数据类型）： 函数与指针、数组、切片、map、list、结构体、通道

3、本章讲解的核心知识点：

1. 数组的用法
2. 切片slice的用法
3. 冒泡排序
4. strings字符串处理包（string类型可以看成是一种特殊的slice类型）
5. strconv包
6. map集合的用法（声明、创建和遍历、map元素删除、查找）
7. list列表的用法
8. 深拷贝与浅拷贝(值类型和引用类型的区别)
9. 随机数及键盘输入
10. time包及math包
11. 利用所学知识练习封装函数

## 二、数组(array)

（一）、什么是数组

1、Go 语言提供了数组类型的数据结构。 数组是具有相同类型的一组长度固定的数据序列，这种类型可以是任意的基本数据类型或复合数据类型及自定义类

型。

2、数组元素可以通过索引下标（位置）来读取或者修改元素数据。索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。数组的下标取值范围是从0开始，到长度减1。

3、数组一旦定义后，大小不能更改。

（二）、数组的语法

1、声明数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

- var 变量名 [数组长度] 数据类型
- 以上为一维数组的定义方式
- 数组长度必须是整数且大于 0
- 未初始化的数组不是nil，也就是说没有空数组（与切片不同）

2、初始化数组

- var nums = [5]int{1 , 2 , 3 , 4 ,5 }
  - 初始化数组中 {} 中的元素个数不能大于 [] 中的数字。
- 如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小。
  - 可以忽略声明中数组的长度并将其替换为...。编译器会自动计算长度。
  - var nums = [...]int{1 , 2 , 3 , 4 ,5 }
  - 该实例与上面的实例是一样的，虽然没有设置数组的大小。
- balance[3] = 4
  - 以上实例读取数组第五个元素。数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。

（三）、数组的长度

1、通过将数组作为参数传递给**len()**函数，可以获得数组的长度。

- 忽略声明中数组的长度并将其替换为...，编译器可以找到长度。

（四）、遍历数组：

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

```
func main() {
  a := [4]float64{67.7, 89.8, 21, 78}
  b := [...]int{2, 3, 5}

  //遍历数组方式1
  for i := 0; i < len(a); i++ {
    fmt.Print(a[i], "\t")
  }

  fmt.Println()
  // 遍历数组方式2
  for _, value := range b {
    fmt.Print(value, "\t")
  }
}
```

（五）、多维数组

1、Go 语言支持多维数组，以下为常用的多维数组声明方式：

- var variable_name [SIZE1][SIZE2]...[SIZEn] variable_type
- 以下实例声明了三维的整型数组： var threedim [5][10][4]int

2、二维数组

- 二维数组是最简单的多维数组，二维数组本质上是由一维数组组成的。二维数组定义方式如下：
- var arrayName [ x ][ y ] variable_type
- 例如： a = [3][4]**int**{

```
    {0, 1, 2, 3} ,  /*  第一行索引为 0 */
    {4, 5, 6, 7} ,  /*  第二行索引为 1 */
    {8, 9, 10, 11}  /*  第三行索引为 2 */
}
```

3、访问二维数组

- 二维数组通过指定坐标来访问。如数组中的行索引与列索引。
- 例如： int val = a[2][3]

- 以上实例访问了二维数组 val 第三行的第四个元素。

4、二维数组可以使用循环嵌套来输出元素：

```
package main
import "fmt"

func main() {
  /* 数组 - 5 行 2 列*/
  var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}

  fmt.Println(len(a))
  fmt.Println(len(a[0]))

  /* 输出数组元素 */
  for  i := 0; i < len(a); i++ {
    for j := 0; j < len(a[0]); j++ {
      fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )
    }
  }
}
```

（六）、数组是值类型

1、数组是值类型 Go中的数组是值类型，而不是引用类型。这意味着当它们被分配给一个新变量时，将把原始数组的副本分配给新变量。如果对新变量进行了更改，则不会在原始数组中反映。

2、当将数组传递给函数作为参数时，它们将通过值传递，而原始数组将保持不变。

3、示例代码：

```
package main
import "fmt"
func main() {
  a := [...]string{"USA", "China", "India", "Germany", "France"}
  b := a // a copy of a is assigned to b
  b[0] = "Singapore"
```

```
    fmt.Println("a： ", a)
    fmt.Println("b： ", b)
}
```

运行结果：

a： [USA China India Germany France]

b： [Singapore China India Germany France]

# 三、切片(Slice)

（一）、什么是切片

1、Go 语言切片是对数组的抽象。

- Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组")；
- 与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。
- 切片本身没有任何数据，它们只是对现有数组的引用。
- 切片与数组相比，不需要设定长度，在[]中不用设定值，相对来说比较自由
- 从概念上面来说slice像一个结构体，这个结构体包含了三个元素：
  - 指针，指向数组中slice指定的开始位置
  - 长度，即slice的长度
  - 最大长度，也就是slice开始位置到数组的最后位置的长度

（二）、切片的语法

1、声明切片

（1）、声明一个未指定长度的数组来定义切片

- var identifier []type
- 切片不需要说明长度。
- 该声明方式，且未初始化的切片为空切片。该切片默认为 nil，长度为 0。

（2）、使用make()函数来创建切片：

a)、var slice1 []type = make([]type, len)

b）、可以简写为： slice1 := make([]type, len)

c）、可以指定容量，其中capacity为可选参数：make([]T, length, capacity)

```
package main
import "fmt"
func main() {
  var numbers = make([]int,3,5)
  fmt.Printf("%T\n" , numbers)
  fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

2、初始化

（1）、直接初始化切片：

- s :=[] int {1,2,3 }

（2）、通过数组截取来初始化切片：

- 数组： arr := [5]int {1,2,3,4,5}

a)、s := arr[:]

- 切片中包含数组所有元素

b)、s := arr[startIndex:endIndex]

- 将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片（前闭后开），长度为endIndex-startIndex

c)、s := arr[startIndex:]

- 缺省endIndex时将表示一直到arr的最后一个元素

d)、s := arr[:endIndex]

- 缺省startIndex时将表示从arr的第一个元素开始

（3）、通过切片截取来初始化切片：

- 可以通过设置下限及上限来设置截取切片 *[lower-bound:upper-bound]*

```
package main
import "fmt"
func main() {
  /* 创建切片 */
```

```
    numbers := []int{0,1,2,3,4,5,6,7,8}
    printSlice(numbers)

    /* 打印原始切片 */
    fmt.Println("numbers ==", numbers)//[0 1 2 3 4 5 6 7 8]

    /* 打印子切片从索引1(包含) 到索引4(不包含)*/
    fmt.Println("numbers[1:4] ==", numbers[1:4])//[1 2 3 ]

    /* 默认下限为 0*/
    fmt.Println("numbers[:3] ==", numbers[:3])//[0 1 2]

    /* 默认上限为 len(s)*/
    fmt.Println("numbers[4:] ==", numbers[4:])//[4 5 6 7 8]

    /* 打印子切片从索引  0(包含) 到索引 2(不包含) */
    number2 := numbers[:2]
    printSlice(number2) //[0 1]

    /* 打印子切片从索引 2(包含) 到索引 5(不包含) */
    number3 := numbers[2:5]
    printSlice(number3) // [2 3 4]
}
func printSlice(x []int){
    fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

运行结果：

```
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=2 cap=9 slice=[0 1]
```

len=3 cap=7 slice=[2 3 4]

（三）、len() 和 cap() 函数

1、切片的长度是切片中元素的数量。

2、切片的容量是从创建切片的索引开始的底层数组中元素的数量。

3、切片是可索引的，并且可以由 len() 方法获取长度， 切片提供了计算容量的方法 cap()， 可以测量切片最长可以达到多少。[数组计算cap()结果与len()相同]

4、切片实际的是获取数组的某一部分，len切片<=cap切片<=len数组

5、cap()的结果决定了切片截取的注意细节

```
package main
import "fmt"
func main() {
  sliceCap()
}

func sliceCap() {
  arr0 := [...]string{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k"}
  fmt.Println("cap(arr0)=", cap(arr0), arr0) //[a b c d e f g h i j k]

  //截取数组，形成切片
  s01 := arr0[2:8]
  fmt.Printf("%T \n", s01)  //[]string
  fmt.Println("cap(s01)=", cap(s01), s01) //9，[c d e f g h]

  s02 := arr0[4:7]
  fmt.Println("cap(s02)=", cap(s02), s02) //7，[e f g]

  //截取切片，形成切片
  s03 := s01[3:9]
  fmt.Println("截取s01[3:9]后形成s03：", s03) //[f g h i j k]

  s04 := s02[4:7]
  fmt.Println("截取s02[4:7]后形成s04：", s04) //[i j k]
```

```
  //切片是引用类型
  s04[0] = "x"

  fmt.Print(arr0, s01, s02, s03, s04)
}
```

（四）、切片是引用类型

1、slice没有自己的任何数据。它只是底层数组的一个引用。对slice所做的任何修改都将反映在底层数组中。

2、数组是值类型，而切片是应用类型

3、两者区别的示例代码：

```
package main
import "fmt"
func main() {
  a := [4]float64{67.7, 89.8, 21, 78}
  b := []int{2, 3, 5}
  fmt.Printf("变量a —— 地址：%p ， 类型：%T，数值：%v，长度：%d
\n", &a, a, a, len(a))
  fmt.Printf("变量b —— 地址：%p ， 类型：%T，数值：%v，长度：%d
\n", &b, b, b, len(b))
  c := a
  d := b
  fmt.Printf("变量c —— 地址：%p ， 类型：%T，数值：%v，长度：%d
\n", &c, c, c, len(c))
  fmt.Printf("变量d —— 地址：%p ， 类型：%T，数值：%v，长度：%d
\n", &d, d, d, len(d))
  a[1] = 200
  fmt.Println("a=", a, "c=", c)
  d[0] = 100
  fmt.Println("b=", b, "d=", d)
}
```

运行结果：

```
变量a —— 地址：0xc4200180c0 ， 类型：[4]float64，数值：[67.7 89.8 21
```

78]，长度：4
变量b —— 地址：0xc42000a060 ， 类型：[]int，数值：[2 3 5]，长度：3
变量c —— 地址：0xc420018160 ， 类型：[4]float64，数值：[67.7 89.8 21
78]，长度：4
变量d —— 地址：0xc42000a0c0 ， 类型：[]int，数值：[2 3 5]，长度：3
a= [67.7 200 21 78] c= [67.7 89.8 21 78]
b= [100 3 5] d= [100 3 5]

4、修改切片数值：

- 当多个片共享相同的底层数组时，每个元素所做的更改将在数组中反映出来。

示例代码：

```
package main
import  "fmt"
func main() {
  //定义数组
  arr := [3]int{1, 2, 3}

  //根据数组截取切片
  nums1 := arr[:]
  nums2 := arr[:]
  fmt.Println("arr=", arr)//[1 2 3]

  nums1[0] = 100
  fmt.Println("arr=", arr)//[100 2 3]

  nums2[1] = 200
  fmt.Println("arr=", arr)//[100 200 3]
}
```

运行结果：

arr= [1 2 3]
arr= [100 2 3]
arr= [100 200 3]

（五）、append() 和 copy() 函数

1、函数append():

- 往切片中追加新元素
- 可以向slice里面追加一个或者多个元素，也可以追加一个切片。
- append函数会改变slice所引用的数组的内容，从而影响到引用同一数组的其它slice。
- 当使用append追加元素到切片时，如果容量不够（也就是(cap-len) == 0），Go就会创建一个新的内存地址来储存元素。

2、函数copy:

- 复制切片元素
- 将源切片中的元素复制到目标切片中，返回复制的元素的个数
- copy方法是不会建立源切片与目标切片之间的联系。也就是两个切片不存在联系，一个修改不影响另一个。

【备注：】以上两个方法不适应于数组。

3、利用切片截取及append()函数实现slice删除元素

1）、删除第一个元素

```
numbers = numbers[1:]
printSlices("numbers:", numbers)
```

2）、删除最后一个元素

```
numbers = numbers[:len(numbers)-1]
printSlices("numbers:", numbers)
```

3）、删除中间元素

```
a := int(len(numbers)/2)
fmt.Println(a)
numbers = append(numbers[:a] , numbers[a+1:]...)
printSlices("numbers:", numbers)
```

4、案例代码一：

```
package main
```

```go
import "fmt"
func main() {
  fmt.Println("1、------------------")
  //var numbers []int
  numbers := make([]int , 0 , 20)
  printSlices("numbers:", numbers)

  numbers = append(numbers, 0) //[0]
  printSlices("numbers:", numbers)

  /* 向切片添加一个元素 */
  numbers = append(numbers, 1) //[0 1]
  printSlices("numbers:", numbers)

  /* 同时添加多个元素 */
  numbers = append(numbers, 2, 3, 4, 5, 6, 7) //[0 1 2 3 4 5 6 7]
  printSlices("numbers:", numbers)

  fmt.Println("2、------------------")
  //追加一个切片
  s1 := []int{100, 200, 300, 400, 500, 600, 700}
  numbers = append(numbers, s1...)
  printSlices("numbers:", numbers)

  fmt.Println("3、------------------")
  //切片删除元素
  //删除第一个元素
  numbers = numbers[1:]
  printSlices("numbers:", numbers)

  //删除最后一个元素
  numbers = numbers[:len(numbers)-1]
  printSlices("numbers:", numbers)
```

```
    //删除中间一个元素
    a := int(len(numbers)/2)
    fmt.Println("中间数： " , a)
    numbers = append(numbers[:a] , numbers[a+1:]...)
    printSlices("numbers:", numbers)

    fmt.Println("4、========================")
    /* 创建切片 numbers1 是之前切片的两倍容量*/
    //numbers1 := make([]int, 0, (cap(numbers))*2)
    numbers1 := make([]int, len(numbers), (cap(numbers))*2)

    /* 拷贝 numbers 的内容到 numbers1 */
    count := copy(numbers1, numbers)
    fmt.Println("拷贝个数： ", count)
    printSlices("numbers1:", numbers1)
    numbers[len(numbers)-1] = 99
    numbers1[0] = 100

    /*numbers1与numbers两者不存在联系，numbers发生变化时，
    numbers1是不会随着变化的。也就是说copy方法是不会建立两个切片的
联系的
     */
    printSlices("numbers1:", numbers1)
    printSlices("numbers:", numbers)
}

func printSlices(name string, x []int) {
    fmt.Print(name, "\t")
    fmt.Printf("addr:%p \t len=%d \t cap=%d \t slice=%v\n", x , len(x),
cap(x), x)
}
```

运行结果：

```
1、------------------
numbers:        addr:0xc42008a000       len=0       cap=20   slice=[]
```

```
numbers:    addr:0xc42008a000    len=1    cap=20    slice=[0]
numbers:    addr:0xc42008a000    len=2    cap=20    slice=[0 1]
numbers:    addr:0xc42008a000    len=8    cap=20    slice=[0 1 2 3 4 5 6
7]
2、------------------
numbers:    addr:0xc42008a000    len=15    cap=20    slice=[0 1 2 3 4 5 6
7 100 200 300 400 500 600 700]
3、------------------
numbers:    addr:0xc42008a008    len=14    cap=19    slice=[1 2 3 4 5 6 7
100 200 300 400 500 600 700]
numbers:    addr:0xc42008a008    len=13    cap=19    slice=[1 2 3 4 5 6 7
100 200 300 400 500 600]
中间数： 6
numbers:    addr:0xc42008a008    len=12    cap=19    slice=[1 2 3 4 5 6
100 200 300 400 500 600]
4、========================
拷贝个数： 12
numbers1:    addr:0xc42008e000    len=12    cap=38    slice=[1 2 3 4 5 6
100 200 300 400 500 600]
numbers1:    addr:0xc42008e000    len=12    cap=38    slice=[100 2 3 4 5 6
100 200 300 400 500 600]
numbers:    addr:0xc42008a008    len=12    cap=19    slice=[1 2 3 4 5 6
100 200 300 400 500 99]
```

5、案例代码二：

```go
package main
import (
  "fmt"
  "strconv"
)

func main() {
  //思考：使用哪种初始化切片的方式更高效?
```

```
    var sa []string
    //sa := make([]string , 0 , 20)
    printSliceMsg(sa)

    //当使用append追加元素到切片时，如果容量不够，go就会创建一个新的切片变量来储存元素。
    for i := 0; i < 15; i++ {
        sa = append(sa, strconv.Itoa(i))
        printSliceMsg(sa)
    }
    printSliceMsg(sa)
}

//打印输出格式化信息
func printSliceMsg(sa []string) {
    fmt.Printf("addr:%p \t len:%v \t cap:%d  \t  value:%v\n", sa, len(sa), cap(sa), sa)
}
```

运行结果：

```
addr:0x0        len:0       cap:0       value:[]
addr:0xc42000e1d0       len:1       cap:1       value:[0]
addr:0xc42000a0a0       len:2       cap:2       value:[0 1]
addr:0xc4200540c0       len:3       cap:4       value:[0 1 2]
addr:0xc4200540c0       len:4       cap:4       value:[0 1 2 3]
addr:0xc42008a000       len:5       cap:8       value:[0 1 2 3 4]
addr:0xc42008a000       len:6       cap:8       value:[0 1 2 3 4 5]
addr:0xc42008a000       len:7       cap:8       value:[0 1 2 3 4 5 6]
addr:0xc42008a000       len:8       cap:8       value:[0 1 2 3 4 5 6 7]
addr:0xc42008c000       len:9       cap:16      value:[0 1 2 3 4 5 6 7 8]
addr:0xc42008c000       len:10      cap:16      value:[0 1 2 3 4 5 6 7 8 9]
addr:0xc42008c000       len:11      cap:16      value:[0 1 2 3 4 5 6 7 8 9 10]
addr:0xc42008c000       len:12      cap:16      value:[0 1 2 3 4 5 6 7 8 9 10 11]
addr:0xc42008c000       len:13      cap:16      value:[0 1 2 3 4 5 6 7 8 9 10 11 12]
```

addr:0xc42008c000 len:14 cap:16 value:[0 1 2 3 4 5 6 7 8 9 10 11 12 13]
addr:0xc42008c000 len:15 cap:16 value:[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]
addr:0xc42008c000 len:15 cap:16 value:[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]

# 四、冒泡排序

(一)、概念：

- 冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法。
- 它重复地遍历要排序的数组元素，一次比较两个元素，如果他们的顺序错误就把他们交换过来。重复地进行直到没有再需要交换，也就是说该数组已经排序完成。
- 这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端，故名“冒泡排序”。

（二）、冒泡排序算法的原理如下：

1、比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2、对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3、针对所有的元素重复以上的步骤，除了最后一个。
4、持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

（三）、冒泡排序分析：

| 54321 -> 12345 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 54321 | -> | 45321 | -> | 43521 | -> | 43251 | -> | 43215 |
| 第1轮 | | 0-0 | | 0-1 | | 0-2 | | 0-3 | |
| | 43215 | -> | 34215 | -> | 32415 | -> | 32145 | | |
| 第2轮 | | 1-0 | | 1-1 | | 1-2 | | | |
| | 32145 | -> | 23145 | -> | 21345 | | | | |
| 第3轮 | | 2-0 | | 2-1 | | | | | |
| | 21345 | -> | 12345 | | | | | | |
| 第4轮 | | 3-0 | | | | | | | |

| 51234 -> 12345 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 51234 | -> | 15234 | -> | 12534 | -> | 12354 | -> | 12345 |
| 第1轮 | | 0-0 | | 0-1 | | 0-2 | | 0-3 | |
| | 12345 | | 12345 | | 12345 | | 12345 | | |
| 第2轮 | | 1-0 | | 1-1 | | 1-2 | | | |

| 12345 -> 12345 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 12345 | -> | 12345 | -> | 12345 | -> | 12345 | -> | 12345 |
| 第1轮 | | 0-0 | | 0-1 | | 0-2 | | 0-3 | |

（四）、核心代码：

```
//从小到大排列
func BubbleSort(arr []int) {
   iCount := 0 //记录内循环次数
   jCount := 0 //记录外循环的次数

   //双层for循环
```

```
  for i := 0; i < len(arr)-1; i++ {
    //定义一个标记，判断本轮是否有两两换位。如果没有换位，那说明排序结束，可以跳出循环。
    //例如：12345，当执行第一轮循环后，所有相邻的两个数值都无需换位，那说明排序正常，无需排序。不用执行后续的循环。
    flag := true

    for j := 0; j < len(arr)-1-i; j++ {
      //判断相邻两个数据的大小
      if arr[j] > arr[j+1] {
        //如果前者比后者大，则执行数据交换
        arr[j], arr[j+1] = arr[j+1], arr[j]
        //如果出现换位，说明排序还没有结束，需要继续循环执行。
        flag = false
      }
      iCount++
    }
    jCount++

    //如果本轮没有换位，那说明排序结束，则跳出循环
    if (flag) {
      break
    }
  }
  fmt.Println("i循环次数=", jCount)
  fmt.Println("j循环次数=", iCount)
}
```