# Go语言内置容器

本章。

本章重点为大家介绍如下的内容：

* Go内置容器概述
* 数组的用法（数组声明、元素访问、值类型、多维数组）
* 切片的用法（创建切片、元素遍历、动态增减元素、内容复制）
* 冒泡排序什么是共识算法
* map（概念、声明、初始化和赋值、遍历map、键值对是否存在、map元素删除、清空map）
* 列表list（概念、声明、初始化、遍历list、插入元素、从list中删除元素）

## 内置容器概述

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

## 数组(array)

### 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。

### 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。

### 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。

### 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")

}

}

### 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] )

}

}

}

### 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]

## 切片(Slice)

### 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

### 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]

### 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)

}

### 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]

### 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

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：

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]

## 冒泡排序

### 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”。

### 冒泡排序算法的原理如下：

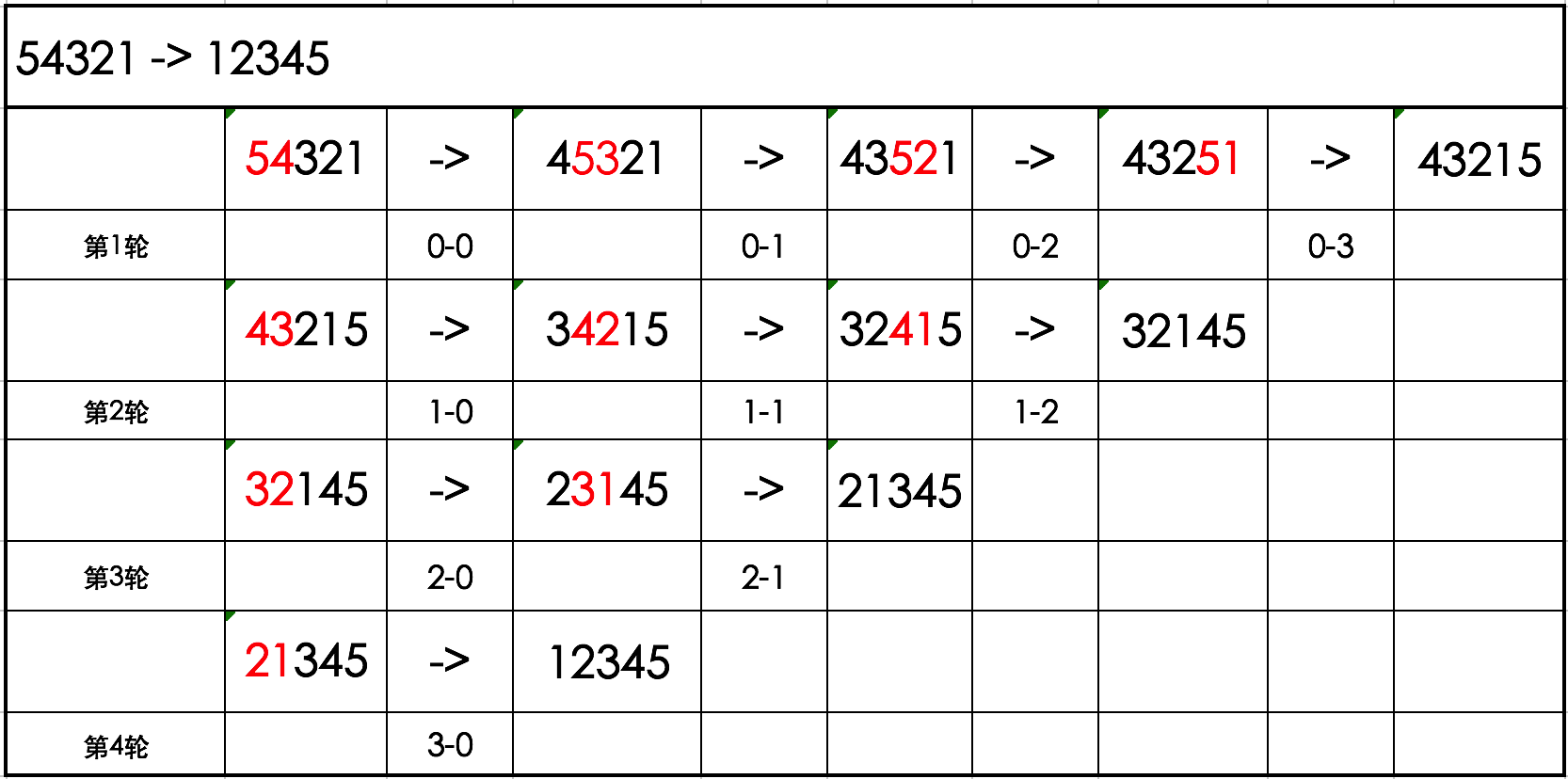
1、比较相邻的元素。如果第一个比第二个大，就交换他们两个。

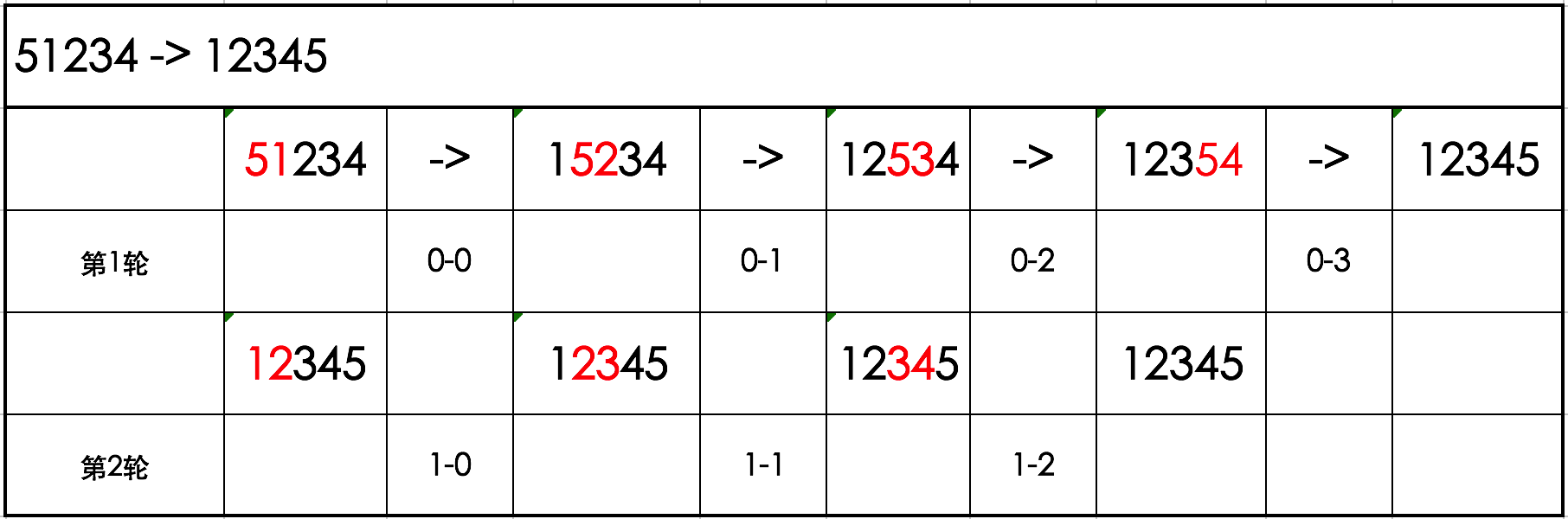
2、对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

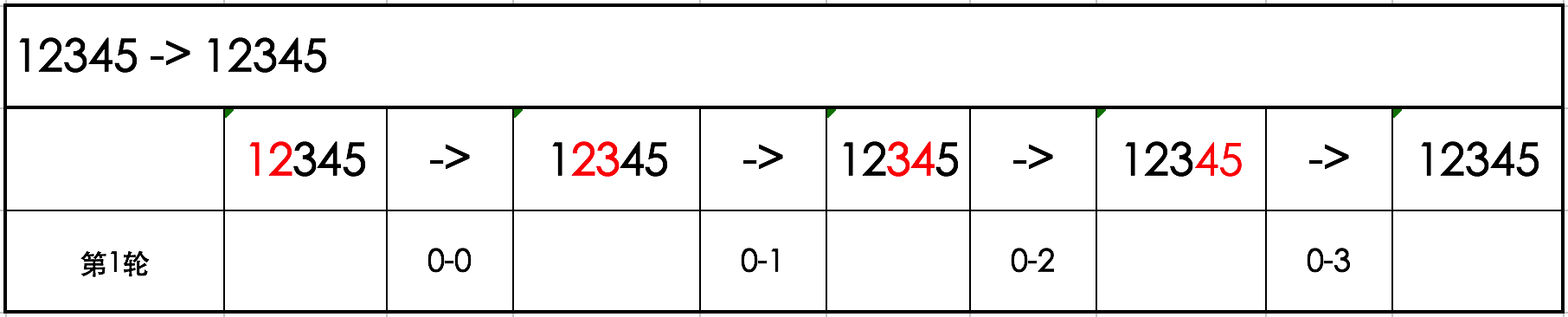
3、针对所有的元素重复以上的步骤，除了最后一个。

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。

### 冒泡排序分析：







### 冒泡排序核心代码

//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)

}

## map

### 什么是map

1、map是Go中的内置类型，它将一个值与一个键关联起来。可以使用相应的键检索值。

● 有资料翻译成地图、映射或字典。但是大多数习惯上翻译成集合。

● map 是一种无序的键值对（key-value pair ）的集合。map通过 key 来快速检索数据，key 类似于索引，指向相应的value值。

● map 是一种集合，所以可以像遍历数组或切片那样去遍历它，不过map 是无序的，所以无法决定它的展示顺序，这是因为 map 是使用 hash 表来实现的。

### 使用map的注意细节：

● map是无序的，每次打印出来的map都会不一样，它不能通过index获取，而必须通过key获取；

● map的长度是不固定的，和slice一样可以扩展。内置的len()函数同样适用于map，返回map拥有的键值对的数量。但是map不能通过cap()函数计算容量（或者说cap()函数的参数不可以是map）；

● 同一个map中key必须保证唯一。key的数据类型必须是可参与比较运算的类型，也就是支持==或!=操作的类型。如布尔型、整数型、浮点型、字符串型、数组。对于切片、函数等引用类型则不能作为键的类型；(Invalid map key type: must not be must not be a function , map or slice)

● map的value可以是任何数据类型。

● 和slice一样，map也是一种引用类型；

### map的用法

1、map的声明

● 可以使用var map 关键字来定义 map，也可以使用内建函数 make 。

（1）、使用map关键字定义map

● var 变量名 map[key类型]value类型

● var声明变量，默认 map 是 nil 。nil map 不能用来存放键值对。

● var声明后，要么声明时初始化，要么再使用make()函数分配到内存空间，这样才能在其中存放键值对。

（2）、使用 make 函数

● 变量名 := make(map[key类型]value类型)

● 该声明方式，如果不初始化 map，那么map也不等于nil。

2、往map中存放键值对（key-value pair ）

（1）、声明时初始化数值

● 示例代码：

var country = map[string]string{

"China": "Beijing",

"France": "Paris",

"Italy": "Rome",

"Japan": "Tokyo",

//"Japan": "Tokyo2",//key名不能重复

}

● 示例代码：

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

● 示例代码：rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }

（2）、先声明再插入数值。

● 示例代码：

//1、定义时初始化

rating := map[string]float32{"C": 5, "Go": 4.5, "Python": 4.5, "C++": 2}

fmt.Println(rating)

var country = map[string]string{

"China": "Beijing",

"France": "Paris",

"Italy": "Rome",

"Japan": "Tokyo",

//"Japan": "Tokyo2",//key名不能重复

}

fmt.Println(country)

//2、创建集合后再赋值

countryCapitalMap := make(map[string]string)

/\* map 插入 key-value 对，各个国家对应的首都 \*/

countryCapitalMap["China"] = "Beijing"

countryCapitalMap["France"] = "Paris"

countryCapitalMap["Italy"] = "Rome"

countryCapitalMap["Japan"] = "Tokyo"

countryCapitalMap["India"] = "New Delhi"

3、map数值遍历

● 示例代码：

（1）、遍历key与value

for k, v := range countryCapitalMap {

fmt.Println("国家是：", k, "首都：", v)

}

（2）、只遍历value

for \_, v := range countryCapitalMap {

fmt.Println("国家是：?", "首都：", v)

}

（3）、只遍历key

for k := range countryCapitalMap {

fmt.Println("国家是：", k, "首都：", countryCapitalMap[k])

}

4、查看元素在集合中是否存在

● 我们可以通过key获取map中对应的value值。语法为：map[key]

● 但是当key如果不存在的时候，会得到该value值类型的默认值，比如string类型得到空字符串，int类型得到0。但是程序不会报错。

● 所以可以通过 value, ok := map[key] ，获取key/value是否存在。ok是bool型，如果 ok 是 true, 则该键值对存在，否则不存在。

● 示例代码：

captial, ok := countryCapitalMap["United States"]

/\* 如果 ok 是 true, 则存在，否则不存在 \*/

if ok {

fmt.Println("首都是：", captial)

} else {

fmt.Println("该国家的首都没有列出！")

}

### delete() 函数

● delete(map, key) 函数用于删除集合的某个元素, 参数为 map 和其对应的 key。删除函数不返回任何值。

● 示例代码：

func main() {

//1、创建并初始化map

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

//2、根据key删除map中的元素

fmt.Println("删除前：", map1)

if \_, ok := map1["background"]; ok {

delete(map1, "background")

}

fmt.Println("删除后：", map1)

//3、清空map

//map1 = map[string]string{}

map1 = make(map[string]string)

fmt.Println("清空后：", map1 , len(map1))

}

### 清空map中所有元素

● Go语言没有为map提供任何清空所有元素的函数；

● 清空map的唯一办法是重新make一个新的map；

● 不用担心垃圾回收的效率，Go语言的垃圾回收比写一个清空函数更高效。

### map是引用类型的

● 与切片相似，map是引用类型。当将map分配给一个新变量时，它们都指向相同的内部数据结构。因此，一个的变化会反映另一个。

● 示例代码：

func main() {

personSalary := map[string]int{

"Steven": 18000,

"Daniel": 5000,

"Josh": 20000,

}

fmt.Println("原始薪资：", personSalary)

newPersonSalary := personSalary

newPersonSalary["Daniel"] = 8000

fmt.Println("修改后newPersonSalary：", newPersonSalary)

fmt.Println("personSalary受影响情况：", personSalary)

}

运行结果：

原始薪资： map[Steven:18000 Daniel:5000 Josh:20000]

修改后newPersonSalary： map[Steven:18000 Daniel:8000 Josh:20000]

personSalary受影响情况： map[Steven:18000 Daniel:8000 Josh:20000]

## list

### 概述

1、list是一种非连续存储的容器，由多个节点组成，节点通过一些变量记录彼此之间的关系。list有多种实现方法，如单向链表、双向链表等。

2、假设A、B、C三个都有电话号码，如果A把号码告诉B，B把号码告诉C。这个过程就建立了一个单向链表结构；

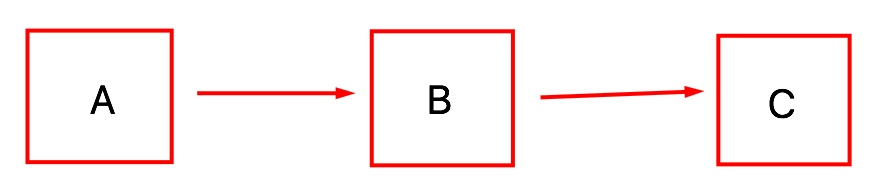


图 1

3、如果在单链表的基础上，再从C开始，将自己的号码给告诉自己号码的人，这样就形成了双向链表结构。

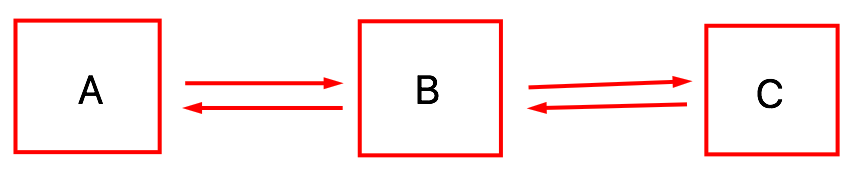


图 2

4、Go语言中list的实现原理是双向链表。list能高效地进行任意位置的元素插入和删除操作。

5、Golang的标准库提供了高级的数据结构List。具体在包container/list。

● container/list包里主要有两个数据结构类型：“Element”、“List”；

● Element类型代表双向链表中的一个元素，相当于C++里面的"iterator"；

● List代表一个双向链表。List零值为一个空的、可用的链表。

● Element有Prev和Next方法用于得到前一个或者下一个Element，Element可以直接调用Value属性；

● list的用法，请查看go语言学习文档：https://www.studygolang.com/pkgdoc

6、list包中两种类型：Element及List的核心方法

（1）、type Element

● func (e \*Element) Next() \*Element

● func (e \*Element) Prev() \*Element

（2）、type List

● func New() \*List

● func (l \*List) Init() \*List

● func (l \*List) Len() int

● func (l \*List) Front() \*Element

● func (l \*List) Back() \*Element

● func (l \*List) PushFront(v interface{}) \*Element

● func (l \*List) PushFrontList(other \*List)

● func (l \*List) PushBack(v interface{}) \*Element

● func (l \*List) PushBackList(other \*List)

● func (l \*List) InsertBefore(v interface{}, mark \*Element) \*Element

● func (l \*List) InsertAfter(v interface{}, mark \*Element) \*Element

● func (l \*List) MoveToFront(e \*Element)

● func (l \*List) MoveToBack(e \*Element)

● func (l \*List) MoveBefore(e, mark \*Element)

● func (l \*List) MoveAfter(e, mark \*Element)

● func (l \*List) Remove(e \*Element) interface{}

### 声明list

● list的声明有两种方法：New和List声明。

1、通过container/list包的New方法声明list

● 变量名 := list.New()

2、通过var声明list

● var 变量名 list.List

● list与切片和map不同，没有具体元素类型的限制。list中的元素可以是任意类型。

● 在CPP里面，list的成员必须是同一个数据类型，但是Go语言中却允许list中插入任意类型的成员。

● 建议使用New()实现声明list。

### element常用方法

1、func (e \*Element) Next() \*Element

Next返回链表的后一个元素或者nil。

2、func (e \*Element) Prev() \*Element

Prev返回链表的前一个元素或者nil。

### list常用方法

1、func New() \*List

New创建一个链表。

2、func (l \*List) Init() \*List

Init清空链表。

3、func (l \*List) Len() int

Len返回链表中元素的个数，复杂度O(1)。

4、func (l \*List) Front() \*Element

Front返回链表第一个元素或nil。

5、func (l \*List) Back() \*Element

Back返回链表最后一个元素或nil。

6、func (l \*List) PushFront(v interface{}) \*Element

PushBack将一个值为v的新元素插入链表的第一个位置，返回生成的新元素。

7、func (l \*List) PushFrontList(other \*List)

添加另一个列表到开头。PushFrontList创建链表other的拷贝，并将拷贝的最后一个位置连接到链表l的第一个位置。

8、func (l \*List) PushBack(v interface{}) \*Element

PushBack将一个值为v的新元素插入链表的最后一个位置，返回生成的新元素。

9、func (l \*List) PushBackList(other \*List)

追加另一个列表到末尾。PushBack创建链表other的拷贝，并将链表l的最后一个位置连接到拷贝的第一个位置。

10、func (l \*List) InsertBefore(v interface{}, mark \*Element) \*Element

InsertBefore将一个值为v的新元素插入到mark前面，并返回生成的新元素。如果mark不是l的元素，l不会被修改。

11、func (l \*List) InsertAfter(v interface{}, mark \*Element) \*Element

InsertAfter将一个值为v的新元素插入到mark后面，并返回新生成的元素。如果mark不是l的元素，l不会被修改。

12、func (l \*List) MoveToFront(e \*Element)

MoveToFront将元素e移动到链表的第一个位置，如果e不是l的元素，l不会被修改。

13、func (l \*List) MoveToBack(e \*Element)

MoveToBack将元素e移动到链表的最后一个位置，如果e不是l的元素，l不会被修改。

14、func (l \*List) MoveBefore(e, mark \*Element)

MoveBefore将元素e移动到mark的前面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

15、func (l \*List) MoveAfter(e, mark \*Element)

MoveAfter将元素e移动到mark的后面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

16、func (l \*List) Remove(e \*Element) interface{}

Remove删除链表中的元素e，并返回e.Value。

### 遍历list

1、顺序遍历

for e := l.Front(); e != nil; e = e.Next() {

fmt.Print(e.Value, " ")

}

2、逆序遍历

for e := l.Back(); e != nil; e = e.Prev() {

fmt.Print(e.Value, " ")

}

### list是值类型还是引用类型

1、list本质是什么？

type List struct {

root Element // sentinel list element, only &root, root.prev, and root.next are used

len int // current list length excluding (this) sentinel element

}

type Element struct {

next, prev \*Element

// The list to which this element belongs.

list \*List

// The value stored with this element.

Value interface{}

}

2、struct是值类型

3、示例代码：

package main

import (

"container/list"

"fmt"

)

func main() {

copyList()

}

//list是值类型，不过采用New()方法声明的是一个指针。所以在拷贝操作和参数传递时具有引用类型的特征。

func copyList() {

//声明list1

list1 := list.New()

printListInfo2("刚声明的list1：", list1)

//给list1赋值

list1.PushBack("one")

list1.PushBack(2)

list1.PushBack("three")

list1.PushFront("first")

printListInfo2("赋值后的list1", list1)

iterateList2(list1)

//将list1拷贝给list2。其实拷贝的是地址

list2 := list1

printListInfo2("刚拷贝的list2", list2)

iterateList2(list2)

//list2修改后

list2.PushBack(250)

list2.PushBack(350)

list2.PushBack(450)

printListInfo2("修改后的list2", list2)

iterateList2(list2)

//list2的修改是否影响到list1？

printListInfo2("修改list2的list1", list1)

iterateList2(list1)

}

func printListInfo2(info string, l \*list.List) {

fmt.Println(info + "----------")

fmt.Printf("%T:%v \t ， 长度为：%d \n", l, l, l.Len())

fmt.Println("----------")

}

func iterateList2(l \*list.List) {

i := 0

for e := l.Front(); e != nil; e = e.Next() {

i++

fmt.Printf("%d:%v \t", i, e.Value)

}

fmt.Println("\n----------")

}

4、结论：

● list是值类型，不过采用list的New()方法声明的是一个指针变量。所以在拷贝操作和参数传递时具有引用类型的特征。