第五章 Go语言内置容器

本章重点为大家介绍如下的内容：

* 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

5.1 数组(array)

5.1.1 数组简介

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。

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。数组一旦定义后，大小不能更改。

5.1.2 一维数组

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

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）

例：

var lines [5]int

var titles [10]string

数组可以在声明的时候直接赋值，这种操作叫数组初始化。

例：

var lines = [5]int{2,4,6,8,10}

var titles = []string{"编号","部门","教研室","姓名"}

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。

例：

var nums = [...]int{1 , 2 , 3 , 4 ,5 }

该实例没有设置数组的大小，与上面的实例是一样的效果。

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

balance[3] = 4

以上实例为数组第4个元素赋值。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。

如果已知一个数组，希望得到数组的长度，需要使用len()函数获得数组长度。通过将数组作为参数传递给len()函数，得到数组的长度。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。

例：

package main

import (

"fmt"

)

func main() {

var titles = []string{"编号","部门","教研室","姓名"}

var persons = [...]string{"小明","小强","小亮"}

fmt.Print("titles:")

fmt.Println(len(titles))

fmt.Print("persons:")

fmt.Println(len(persons))

}

输出结果为：

titles:4

persons:3

通过循环语句与len()函数，可以实现遍历数组所有数据；通过for range也可以实现同样的功能。

例：

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")

}

}

输出结果：

67.7 89.8 21 78

2 3 5

5.1.3 多维数组

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

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：

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 \*/

}

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：

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。

二维数组可以使用循环嵌套的方式实现数组遍历。

例：

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] )

}

}

}

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。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。

例：

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]

5.2 切片(Slice)

5.2.1 切片简介

Go 语言切片是对数组的抽象。Go语言中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；

5.2.2 切片的语法

1.切片在声明时不需要说明长度，使用一个未指定长度的数组来定义切片如下：

var identifier []type

采用这种声明方式，且未初始化的切片称为空切片。这种切片默认为 nil，长度为 0。

2.使用make()函数来创建切片如下：

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

可以简写为：slice1 := make([]type, len)

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)

}

切片可以在声明的时候直接赋值，这种操作叫切片初始化。

1.直接初始化切片

例：

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

2.通过对数组或切片截取来初始化切片

* s := arr[:]，切片中包含数组所有元素；
* 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；
* s := arr[startIndex:]，缺省endIndex时将表示一直到arr的最后一个元素；
* s := arr[:endIndex]，缺省startIndex时将表示从arr的第一个元素开始。

例：

package main

import "fmt"

func main() {

arr := [5]int {1,2,3,4,5}

s := arr[:]//完全截取

fmt.Println(s)

s = arr[1:2]//截取从索引1到索引2

fmt.Println(s)

s = arr[3:]//从索引3开始截取到结束

fmt.Println(s)

s = arr[:2]//截取到索引2

fmt.Println(s)

}

输出结果：

[1 2 3 4 5]

[2]

[4 5]

[1 2]

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]

5.2.3 len() 和 cap() 函数

切片的长度是切片中元素的数量。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。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。切片实际的是获取数组的某一部分，len切片<=cap切片<=len数组。

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)

}

输出结果：

cap(arr0)= 11 [a b c d e f g h i j k]

[]string

cap(s01)= 9 [c d e f g h]

cap(s02)= 7 [e f g]

截取s01[3:9]后形成s03： [f g h i j k]

截取s02[4:7]后形成s04： [i j k]

[a b c d e f g h x j k] [c d e f g h] [e f g] [f g h x j k] [x j k]

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。数组是值类型，而切片是应用类型。

例：

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：0xc000010300 ， 类型：[4]float64，数值：[67.7 89.8 21 78]，长度：4

变量b —— 地址：0xc0000044a0 ， 类型：[]int，数值：[2 3 5]，长度：3

变量c —— 地址：0xc0000103a0 ， 类型：[4]float64，数值：[67.7 89.8 21 78]，长度：4

变量d —— 地址：0xc000004520 ， 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]

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。

例：

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]

5.2.4 append() 和 copy() 函数

1.函数append()

append()函数实现往切片尾部追加新元素，函数可以向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()

copy()函数实现复制切片元素。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函数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。

例：

package main

import "fmt"

func main() {

numbers := make([]int , 0 , 20)

numbers = append(numbers, 2, 3, 4, 5, 6, 7)

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)

}

输出结果：

拷贝个数： 6

numbers1: addr:0xc000108000 len=6 cap=40 slice=[2 3 4 5 6 7]

numbers1: addr:0xc000108000 len=6 cap=40 slice=[100 3 4 5 6 7]

numbers: addr:0xc000106000 len=6 cap=20 slice=[2 3 4 5 6 99]

5.2.5 切片高级操作

1.删除切片中的元素。

Go语言中并没有为切片删除元素及提供相应的函数，如果需要实现删除切片中的某个元素，可以利用切片截取及append()函数实现删除元素

删除第一个元素使用numbers = numbers[1:]；

删除最后一个元素使用numbers = numbers[:len(numbers)-1]；

删除任意元素n使用numbers = append(numbers[:n] , numbers[n+1:]...)

例：

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)

}

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:0xc000106000 len=0 cap=20 slice=[]

numbers: addr:0xc000106000 len=1 cap=20 slice=[0]

numbers: addr:0xc000106000 len=2 cap=20 slice=[0 1]

numbers: addr:0xc000106000 len=8 cap=20 slice=[0 1 2 3 4 5 6 7]

2、------------------

numbers: addr:0xc000106000 len=15 cap=20 slice=[0 1 2 3 4 5 6 7 100 200 300 400 500 600 700]

3、------------------

numbers: addr:0xc000106008 len=14 cap=19 slice=[1 2 3 4 5 6 7 100 200 300 400 500 600 700]

numbers: addr:0xc000106008 len=13 cap=19 slice=[1 2 3 4 5 6 7 100 200 300 400 500 600]

中间数： 6

numbers: addr:0xc000106008 len=12 cap=19 slice=[1 2 3 4 5 6 100 200 300 400 500 600]

5.3 冒泡排序

5.3.1 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”。

5.3.2 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。

5.3.3 冒泡排序分析：

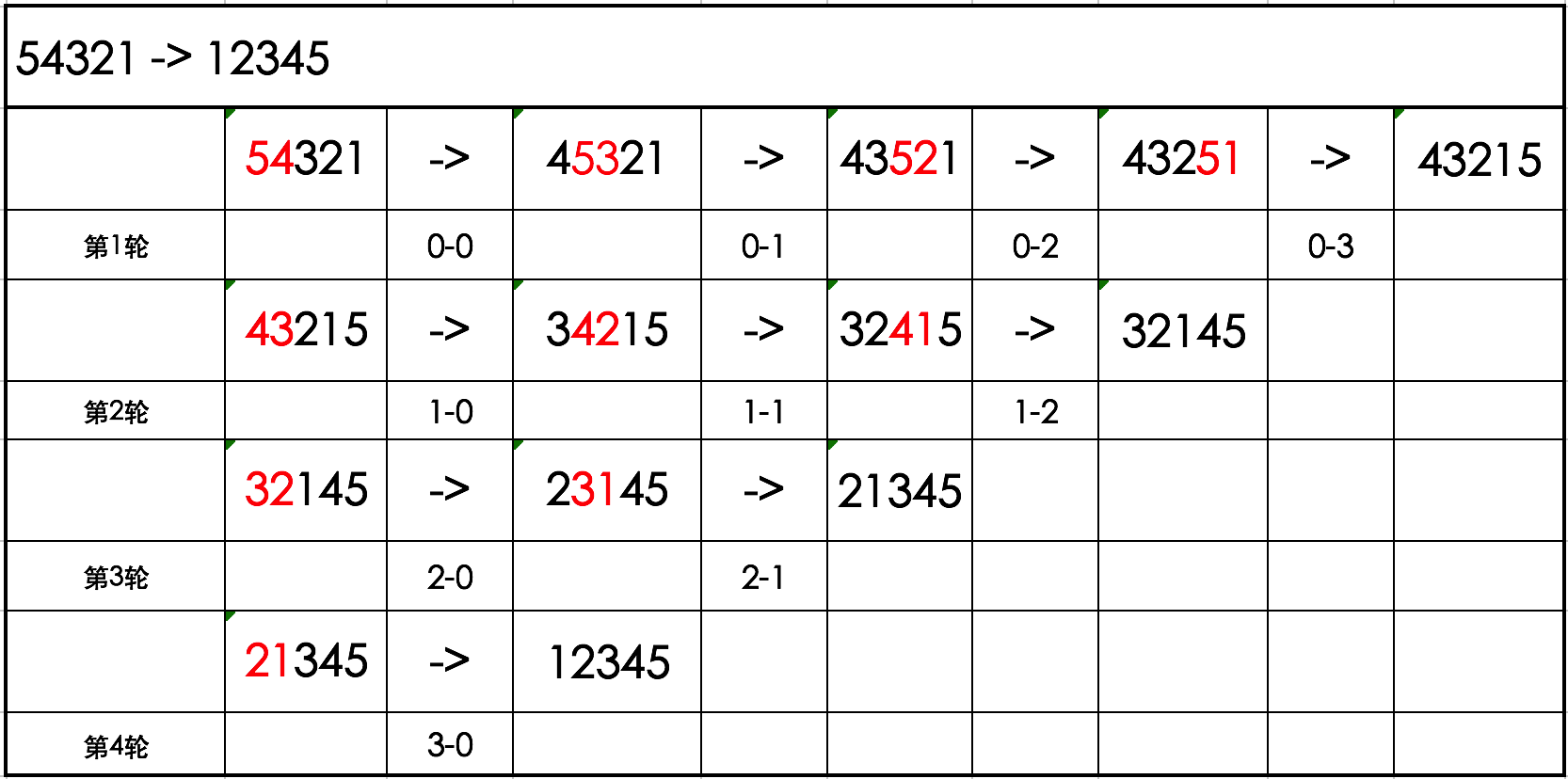


图5.1

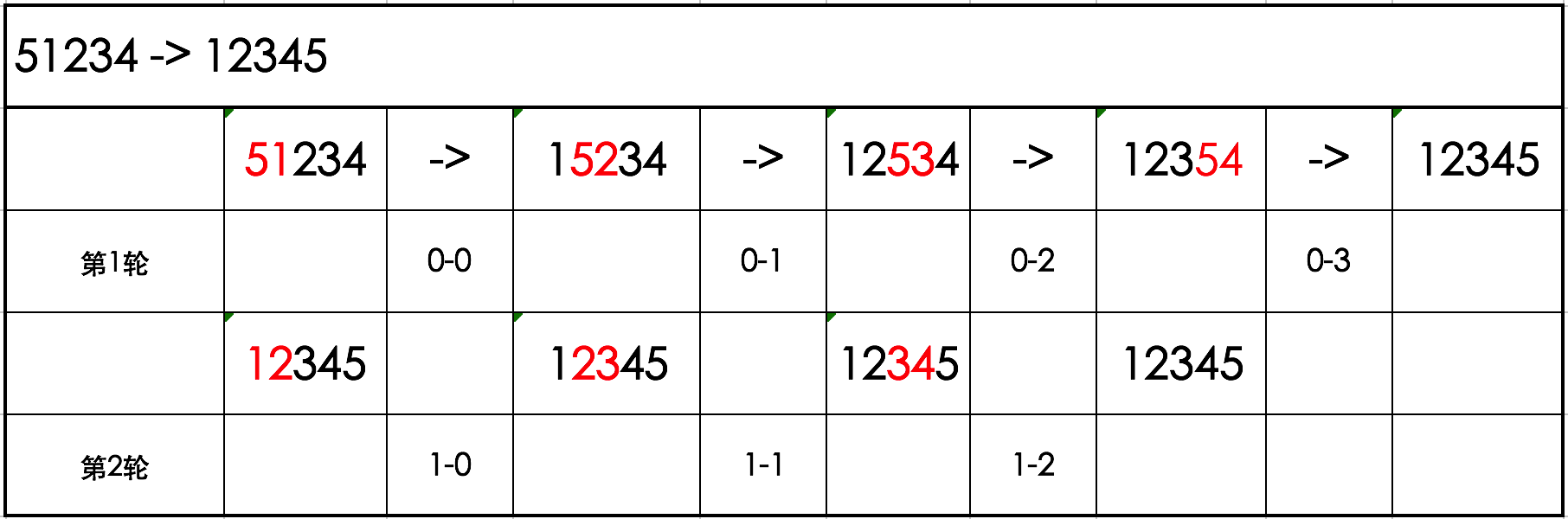


图5.2

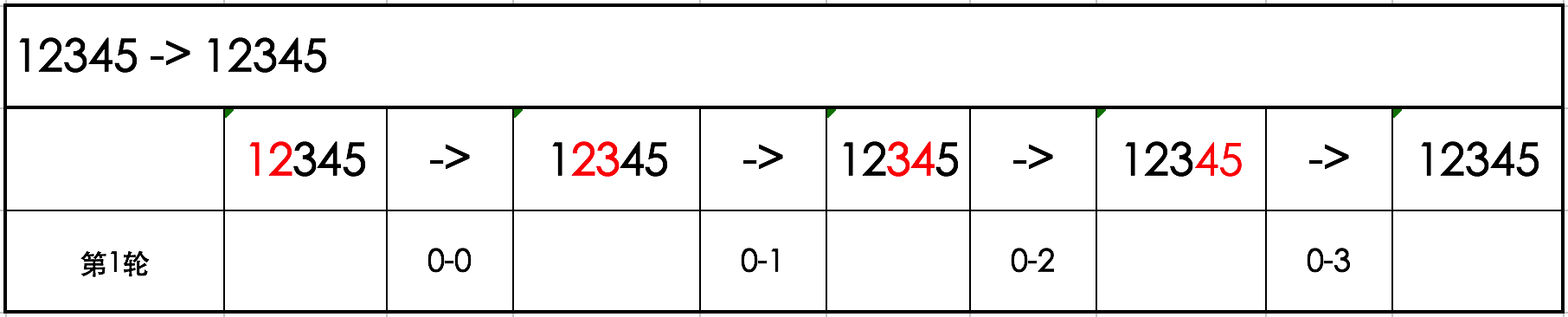


图5.3

5.3.4 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)

}

5.4 map

5.4.1 map简介

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。

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；

5.4.2 map的语法

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。

例：

声明时初始化数值

var country = map[string]string{

"China": "Beijing",

"France": "Paris",

"Italy": "Rome",

"Japan": "Tokyo",

//"Japan": "Tokyo2",//key名不能重复

}

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

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

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。

例：

定义时初始化

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

fmt.Println(rating)

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

由于map是无序的数据集合，遍历时只能使用迭代遍历的方式，使用for range的方式迭代map集合。

例：

package main

import "fmt"

func main() {

countryCapitalMap := make(map[string]string)

countryCapitalMap["China"] = "Beijing"

countryCapitalMap["France"] = "Paris"

countryCapitalMap["Italy"] = "Rome"

countryCapitalMap["Japan"] = "Tokyo"

countryCapitalMap["India"] = "New Delhi"

for k, v := range countryCapitalMap {

fmt.Println("国家是：", k, "首都：", v)

}

}

输出结果：

国家是： China 首都： Beijing

国家是： France 首都： Paris

国家是： Italy 首都： Rome

国家是： Japan 首都： Tokyo

国家是： India 首都： New Delhi

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。

例：

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！")

}

5.4.4 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。

例：

package main

import "fmt"

func main() {

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

fmt.Println("删除前：", map1)

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

delete(map1, "background")

}

fmt.Println("删除后：", map1)

map1 = make(map[string]string)

fmt.Println("清空后：", map1 , len(map1))

}

输出结果：

删除前： map[background:none border:solid color:red element:div height:200px width:100px]

删除后： map[border:solid color:red element:div height:200px width:100px]

清空后： map[] 0

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。

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。

例：

package main

import "fmt"

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[Daniel:5000 Josh:20000 Steven:18000]

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

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

5.5 list

5.5.1 概述

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。

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；

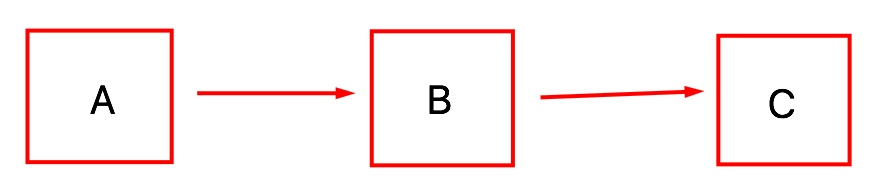


图 5.4

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。

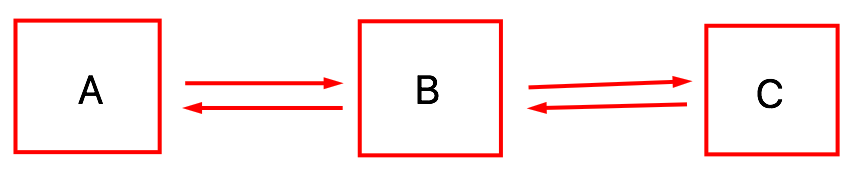


图 5.5

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。

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；

5.5.2 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。

5.5.3 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。

5.5.4 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。

5.5.5 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, " ")

}

5.5.6 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

例：

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----------")

}

输出结果：

刚声明的list1：----------

\*list.List:&{{0xc00005e330 0xc00005e330 <nil> <nil>} 0} ， 长度为：0

----------

赋值后的list1----------

\*list.List:&{{0xc00005e420 0xc00005e3f0 <nil> <nil>} 4} ， 长度为：4

----------

1:first 2:one 3:2 4:three

----------

刚拷贝的list2----------

\*list.List:&{{0xc00005e420 0xc00005e3f0 <nil> <nil>} 4} ， 长度为：4

----------

1:first 2:one 3:2 4:three

----------

修改后的list2----------

\*list.List:&{{0xc00005e420 0xc00005e510 <nil> <nil>} 7} ， 长度为：7

----------

1:first 2:one 3:2 4:three 5:250 6:350 7:450

----------

修改list2的list1----------

\*list.List:&{{0xc00005e420 0xc00005e510 <nil> <nil>} 7} ， 长度为：7

----------

1:first 2:one 3:2 4:three 5:250 6:350 7:450

----------

Process finished with exit code 0

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。

[第五章 Go语言内置容器 1](#_Toc20301)

[5.1 内置容器概述 1](#_Toc22945)

[5.2 数组(array) 1](#_Toc30014)

[5.2.1 什么是数组 1](#_Toc13832)

[5.2.2 数组的语法 2](#_Toc11894)

[5.2.3 数组的长度 2](#_Toc29628)

[5.2.4 遍历数组 2](#_Toc15211)

[5.2.5 多维数组 3](#_Toc29768)

[5.2.6 数组是值类型 4](#_Toc22186)

[5.3 切片(Slice) 5](#_Toc4742)

[5.3.1 什么是切片 5](#_Toc22555)

[5.3.2 切片的语法 5](#_Toc30037)

[5.3.3 len() 和 cap() 函数 7](#_Toc1898)

[5.3.4 切片是引用类型 9](#_Toc14540)

[5.3.5 append() 和 copy() 函数 10](#_Toc16111)

[5.4 冒泡排序 15](#_Toc10782)

[5.4.1 概念： 15](#_Toc19069)

[5.4.2 冒泡排序算法的原理如下： 15](#_Toc29582)

[5.4.3 冒泡排序分析： 16](#_Toc2561)

[5.4.4 冒泡排序核心代码 16](#_Toc16679)

[5.5 map 18](#_Toc24034)

[5.5.1 什么是map 18](#_Toc27672)

[5.5.2 使用map的注意细节： 18](#_Toc21366)

[5.5.3 map的用法 18](#_Toc27090)

[5.5.4 delete() 函数 20](#_Toc31241)

[5.5.5 清空map中所有元素 21](#_Toc25966)

[5.5.6 map是引用类型的 21](#_Toc19794)

[5.6 list 21](#_Toc23774)

[5.6.1 概述 21](#_Toc18499)

[5.6.2 声明list 23](#_Toc10477)

[5.6.3 element常用方法 23](#_Toc2960)

[5.6.4 list常用方法 23](#_Toc20013)

[5.6.5 遍历list 24](#_Toc7542)

[5.6.6 list是值类型还是引用类型 24](#_Toc9776)