Go内置容器——map和list

目录：

一、map

（一）、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。

（二）、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；

（三）、 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！")
}
```

（四）、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))
}
```

（五）、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。

（六）、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]

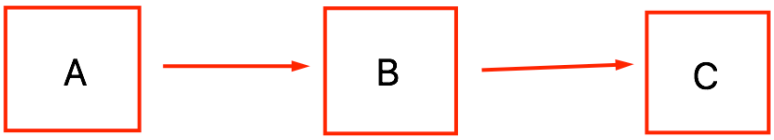
personSalary受影响情况： map[Steven:18000 Daniel:8000 Josh:20000]

二、list

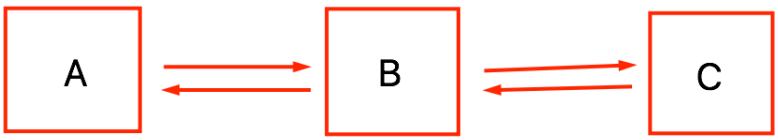
（一）、概述

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；



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。



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{}

（二）、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。

（三）、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。

（四）、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。

（五）、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, " ")
}
```

（六）、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。