Go语言 标准库 container包

Go标准库的`container`包提供了3个包，分别是`heap`，`list`，`ring`，分别实现了堆，链表和环形链表。

- List：Go中对链表的实现，其中List：双向链表，Element：链表中的元素。
- Ring：实现的是一个循环链表，也就是我们俗称的环。
- Heap：Go中对堆的实现。

heap

Go中堆使用的数据结构是最小二叉树，即根节点比左边子树和右边子树的所有值都小。

heap包提供了对任意类型（实现了heap.Interface接口）的堆操作。（最小）堆是具有“每个节点都是以其为根的子树中最小值”属性的树。

树的最小元素为其根元素，索引0的位置。

heap是常用的实现优先队列的方法。要创建一个优先队列，实现一个具有使用（负的）优先级作为比较的依据的Less方法的Heap接口，如此一来可用Push添加项目而用Pop取出队列最高优先级的项目。

堆概念

堆是一种经过排序的完全二叉树，其中任一非终端节点的数据值均不大于（或不小于）其左孩子和右孩子节点的值。

最大堆和最小堆是二叉堆的两种形式。

最大堆：**根**结点的键值是所有堆结点键值中最大者。

最小堆：**根**结点的键值是所有堆结点键值中最小者。

heap

树的最小元素在根部，为index 0。

heap包对任意实现了heap接口的类型提供堆操作。

heap是常用的实现优先队列的方法。要创建一个优先队列，实现一个具有使用（负的）优先级作为比较的依据的Less方法的Heap接口，如此一来可用Push添加项目而用Pop取出队列最高优先级的项目。

类型接口

heap包中核心是heap.Interface接口， 堆的基础存储是一个树形结构，可以用数组或是链表实现，通过heap的函数，可以建立堆并在堆上进行操作。

heap.Interface接口

```
type Interface interface {
  sort.Interface
  Push(x any) // add x as element Len()
  Pop() any   // remove and return element Len() - 1.
}
```

根据注释我们发现`Push`的实现是要求在末尾插入，`Pop`的实现要求删除末尾元素。`sort.Interface`是另一个接口，定义了一个可排序的容器。

sort.Interface接口

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less reports whether the element with
    // index i should sort before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

接口中`Less`方法的实现方式决定了排序方式，是增序还是降序。`sort`包提供了3种默认实现：`IntSlice`，`Float64Slice`和`StringSlice`。它们默认都是增序的，如果需要降序排序，可以使用`Reverse`函数快速得到一个降序数组。

在实现了这些接口之后，就可以被heap包提供的各个函数进行操作，从而实现一个堆。

根据上面interface的定义，可以看出这个堆结构继承自sort.Interface，而sort.Interface，需要实现三个方法：Len()，Less()，Swap()。

同事还需要实现堆接口定义的两个方法：Push(x interface{}) / Pop() interface{}， 所以我们要想使用heap定义一个堆， 只需要定义实现了这五个方法结构就可以了。

> 接口的Push和Pop方法是供heap包调用的，请使用heap.Push和heap.Pop来向一个堆添加或者删除元素。

成员函数

heap包中提供了几个最基本的堆操作函数，包括Init，Fix，Push，Pop和Remove （其中up, down函数为非导出函数）。这些函数都通过调用前面实现接口里的方法，对堆进行操作。

Init

一个堆在使用任何堆操作之前应先初始化。接受参数为实现了heap.Interface接口的对象。

```
func Init(h Interface)
```

Fix

在修改第i个元素后，调用本函数修复堆，比删除第i个元素后插入新元素更有效率。

```
func Fix(h Interface, i int)
```

Push&Pop

Push和Pop是一对标准堆操作，Push向堆添加一个新元素，Pop弹出并返回堆顶元素，而在push和pop操作不会破坏堆的结构。

Remove

删除堆中的第i个元素，并保持堆的约束性。

up&down

维护堆的函数有两个，`up`和`donw`。`up`将小元素"上浮"，`down`让大元素"下沉"。

`up`在将元素上浮时，并不是挨个比较上浮，而是向当前位置往前一半的地方上浮，所以堆的底层数组并不是有序的。使用`up`维护堆的目的是不是让堆有序，而是让最小的元素始终是数组的第一个元素。

```
func up(h heap.Interface, j int)
```

down在将大元素下沉的时候也是按当前位置2倍的距离下沉，同样也不一定能让数组有序。参数i0表示要下沉的元素位置，n表示下沉的界限，超过该位置便不再下沉，注意这个位置是达不到的。down的返回值表示有没有下沉，因为如果当前元素没有被下沉，就说明当前元素比后面的元素小，那么就需要考虑将当前元素上浮，具体可以看heap.Remove的实现。

```
func down(h sort.Interface, i0, n int) bool
```

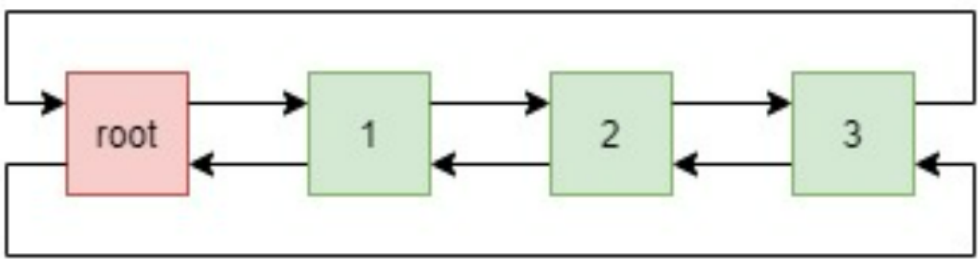
示例：**包含int的最小堆**

```
// This example demonstrates an integer heap built using the heap interface.
package heap_test

import (
    "container/heap"
    "fmt"
)

// An IntHeap is a min-heap of ints.
type IntHeap []int

func (h IntHeap) Len() int           { return len(h) }
func (h IntHeap) Less(i, j int) bool { return h[i] < h[j] }
func (h IntHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }

func (h *IntHeap) Push(x interface{}) {
    // Push and Pop use pointer receivers because they modify the slice's
length,
    // not just its contents.
    *h = append(*h, x.(int))
}

func (h *IntHeap) Pop() interface{} {
    old := *h
    n := len(old)
    x := old[n-1]
    *h = old[0 : n-1]
    return x
}

// This example inserts several ints into an IntHeap, checks the minimum,
// and removes them in order of priority.
```

```
func Example_intHeap() {
    h := &IntHeap{2, 1, 5}
    heap.Init(h)
    heap.Push(h, 3)
    fmt.Printf("minimum: %d\n", (*h)[0])
    for h.Len() > 0 {
        fmt.Printf("%d ", heap.Pop(h))
    }
    // Output:
    // minimum: 1
    // 1 2 3 5
}
```

list

标准库的`list`是一个双向环链表。



这个代码包中有两个结构体——List和Element，List表示一个双向链表，而 Element 则代表了链表中元素的结构。

结构体

Element

```
type Element struct {
  // Next and previous pointers in the doubly-linked list of elements.
  // To simplify the implementation, internally a list l is implemented
  // as a ring, such that &l.root is both the next element of the last
  // list element (l.Back()) and the previous element of the first list
  // element (l.Front()).
  next, prev *Element

  // The list to which this element belongs.
  list *List

  // The value stored with this element.
  Value any
}
```

List

```
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
}
```

`List`可以开箱即用，也可以通过`list.New()`返回一个初始化过的链表。

示例：

```
package main

import (
  "container/list"
  "fmt"
)

func main() {
  var l1 list.List
  l1.PushBack(1)
  fmt.Println(l1)

  l2 := list.New()
  l2.PushFront("a")
  fmt.Println(l2)
}
```

成员函数

List有三个基础方法 New，Init，Len。

New

返回一个初始化链表。

```
func New() *List {
    return new(List).Init()
}
```

Init

初始化链表，清空链表。

```
func (l *List) Init() *List {
    l.root.next = &l.root
    l.root.prev = &l.root
    l.len = 0
    return l
}
```

Len

返回链表长度。

```
func (l *List) Len() int {
    return l.len
}
```

lazyInit

延迟初始化

```
func (l *List) lazyInit() {
    if l.root.next == nil {
        l.Init()
    }
}
```

Front

返回list头节点元素

```
func (l *List) Front() *Element {}
```

Back

返回list尾节点元素

```
func (l *List) Back() *Element {}
```

PushFront

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

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

PushFrontList

创建链表other的拷贝，并将拷贝的最后一个位置连接到list的第一个位置

```
func (l *List) PushFrontList(other *List) {}
```

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

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

PushBackList

创建链表other的拷贝，并将拷贝的第一个位置连接到list的最后一个位置

```
func (l *List) PushBackList(other *List) {}
```

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

```
func (l *List) InsertBefore(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

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

```
func (l *List) MoveToFront(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

```
func (l *List) MoveToBack(e *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

```
func (l *List) MoveBefore(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

```
func (l *List) MoveAfter(e, mark *Element) {}
```

Remove

删除链表中的元素e，并返回e.Value

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

container/list 标准库中代码实现巧妙，包括延迟初始化、封装设计等技巧，对小白(like me)有很好的借鉴意义。

示例：

```
package main

import (
    "container/list"
    "fmt"
```

```
)

func main() {
    l := list.New()
    l.PushFront(1)                // 1
    l.PushBack(2)                 // 1->2
    fmt.Println(l.Front().Value)  // 1
    fmt.Println(l.Back().Value)   // 2
    other := list.New()
    other.PushFront(3)
    other.PushBack(4)
    l.PushFrontList(other)        // 3->4->1->2
    fmt.Println(l.Front().Value)
    fmt.Println(l.Back().Value)
    fmt.Println(l.Len())
    l.Remove(l.Front().Next())    // 3->1->2
    fmt.Println(l.Len())
    for v := l.Front(); v != nil; v = v.Next() {
        fmt.Printf("%d ", v.Value)
    }
    fmt.Printf("\n")
}
```

ring

Go中提供的ring是一个双向的循环链表，与list的区别在于没有表头和表尾，ring表头和表尾相连，构成一个环。

ring实现了环形链表的操作。环的尾部就是头部，所以每个元素实际上就可以代表自身的这个环。不需要像list一样保持list和element两个结构，只需要保持一个结构就可以。

数据结构

type Ring

环形链表没有头尾；指向环形链表任一元素的指针都可以作为整个环形链表看待。Ring零值是具有一个（Value字段为nil的）元素的链表。

```
type Ring struct {
    next, prev *Ring
    Value      interface{} // for use by client; untouched by this library
}
```

成员函数

使用`Ring`建议通过`ring.New`函数创建环形链表，并指定容量。

ring包中除了init初始化，其他都是对外调用的包。

init

初始化ring。

```
func (r *Ring) init() *Ring
```

Next

返回下一节点。r的后一个元素，r不能为空。

```
func (r *Ring) Next() *Ring
```

Prev

返回上一个节点。r的前一个元素，r不能为空。

```
func (r *Ring) Prev() *Ring
```

Move

移动n步，负数向前移动，正数向后移动。返回r移动n%r.Len()个位置（n>=0向前移动，n<0向后移动）后的元素，r不能为空。

```
func (r *Ring) Move(n int) *Ring
```

New

创建一个具有n个元素的环形链表。

```
func New(n int) *Ring
```

Link

将一个环形链表连接进来。Link连接r和s，并返回r原本的后继元素r.Next()。r不能为空。

如果r和s指向同一个环形链表，则会删除掉r和s之间的元素，删掉的元素构成一个子链表，返回指向该子链表的指针（r的原后继元素）；如果没有删除元素，则仍然返回r的原后继元素，而不是nil。

如果r和s指向不同的链表，将创建一个单独的链表，将s指向的链表插入r后面，返回s原最后一个元素后面的元素（即r的原后继元素）。

```
func (r *Ring) Link(s *Ring) *Ring
```

Unlink

从环形链表中分离出一个n节点的环形链表。删除链表中n % r.Len()个元素，从r.Next()开始删除。如果n % r.Len() == 0，不修改r。返回删除的元素构成的链表，r不能为空。

```
func (r *Ring) Unlink(n int) *Ring
```

Len

返回环形链表中的元素个数，复杂度O(n)。

```
func (r *Ring) Len() int
```

Do

接受一个函数，遍历环形链表并应用该函数。对链表的每一个元素都执行f（正向顺序） 注意如果f改变了*r，Do的行为是未定义的。

```
func (r *Ring) Do(f func(interface{}))
```

示例：

```
package main

import (
  "container/ring"
  "fmt"
)

func main() {
  show := func(x any) {
    fmt.Printf("%v ", x)
  }
  r := ring.New(2)
  r.Value = 1
  r.Next().Value = 2
  r.Do(show)
  fmt.Println()
  r = r.Move(-1)
  r.Do(show)
  fmt.Println()
}
/* 输出
1 2
2 1
*/
```