

# 链表

链表和数组、切片一样，都是容器，也就是它就是个盒子，盒子里面装着各种元素的节点。

所以在设计时，需要设计容器和元素。

```
1 // 单向链表
2 package main
3
4 import (
5     "fmt"
6     "strconv"
7 )
8
9 type Node struct {
10     item int
11     next *Node
12     // prev *Node
13 }
14
15 func (n *Node) String() string {
16     var next string
17     if n.next == nil {
18         next = "null"
19     } else {
20         next = strconv.Itoa(n.next.item)
21     }
22     return fmt.Sprintf(
23         "%d ==> %s",
24         n.item,
25         next,
26     )
27 }
28
29 type LinkedList struct {
30     head *Node
31     tail *Node
32     len int
33 }
34
35 func New() *LinkedList {
36     return &LinkedList{}
37 }
38
39 func (l *LinkedList) Len() int {
40     return l.len
41 }
42
43 func (l *LinkedList) Append(item int) {
44     node := new(Node)
45     node.item = item
46     // node.next = nil 默认
47 }
```

```

48     if l.head == nil {
49         // 没有元素，头尾都是nil，但凡有一个元素，头尾都不是nil
50         l.head = node
51         l.tail = node
52     } else {
53         // 哪怕是有有一个元素，append都是动尾巴
54         l.tail.next = node // 手拉手1，修改当前尾巴的下一个
55         l.tail = node      // 修改尾巴
56     }
57
58     l.len++ // 新增成功长度加1
59 }
60
61 func (l *LinkedList) IterNodes() {
62     p := l.head
63     for p != nil {
64         fmt.Println(p)
65         p = p.next
66     }
67 }
68
69 func main() {
70     // 构造一个链表
71     ll := NewList()
72     ll.Append(2)
73     ll.Append(3)
74     fmt.Println(ll, ll.Len)
75     fmt.Println("~~~~~")
76
77     ll.IterNodes()
78 }
79

```

有了上面代码，修改为双向链表

```

1  package main
2
3  import (
4      "fmt"
5      "strconv"
6  )
7
8  type Node struct {
9      item int
10     next *Node
11     prev *Node
12 }
13
14 func (n *Node) String() string {
15     var next, prev string
16     if n.next == nil {
17         next = "null"
18     } else {
19         next = strconv.Itoa(n.next.item)
20     }
21

```

```

21     if n.prev == nil {
22         prev = "null"
23     } else {
24         prev = fmt.Sprintf("%d", n.prev.item)
25     }
26     return fmt.Sprintf(
27         "%s <== %d ==> %s",
28         prev,
29         n.item,
30         next,
31     )
32 }
33
34 type LinkedList struct {
35     head *Node
36     tail *Node
37     len  int
38 }
39
40 func NewList() *LinkedList {
41     return &LinkedList{}
42 }
43
44 func (l *LinkedList) Len() int {
45     return l.len
46 }
47
48 func (l *LinkedList) Append(item int) {
49     node := new(Node)
50     node.item = item
51     // node.prev = nil 默认
52     // node.next = nil 默认
53
54     if l.head == nil {
55         // 没有元素，头尾都是nil，但凡有一个元素，头尾都不是nil
56         l.head = node
57         l.tail = node
58     } else {
59         // 哪怕是有一个元素，append都是动尾巴
60         l.tail.next = node // 手拉手1，修改当前尾巴的下一个
61         node.prev = l.tail // 手拉手2，新节点拉尾巴
62         l.tail = node      // 修改尾巴
63     }
64     l.len++ // 新增成功长度加1
65 }
66
67 func (l *LinkedList) IterNodes() {
68     p := l.head
69     for p != nil {
70         fmt.Println(p)
71         p = p.next
72     }
73 }
74
75 func main() {

```

```

76 // 构造一个链表
77 ll := NewList()
78 ll.Append(2)
79 ll.Append(3)
80 fmt.Println(ll, ll.Len)
81 fmt.Println("~~~~~")
82
83 ll.IterNodes()
84
85 }

```

开始增加xxxx方法

```

1  package main
2
3  import (
4      "errors"
5      "fmt"
6      "strconv"
7  )
8
9  type Node struct {
10     item int
11     next *Node
12     prev *Node
13 }
14
15 func (n *Node) String() string {
16     var next, prev string
17     if n.next == nil {
18         next = "null"
19     } else {
20         next = strconv.Itoa(n.next.item)
21     }
22     if n.prev == nil {
23         prev = "null"
24     } else {
25         prev = fmt.Sprintf("%d", n.prev.item)
26     }
27     return fmt.Sprintf(
28         "%s <== %d ==> %s",
29         prev,
30         n.item,
31         next,
32     )
33 }
34
35 type LinkedList struct {
36     head *Node
37     tail *Node
38     len  int
39 }
40
41 func NewList() *LinkedList {
42     return &LinkedList{}

```

```

43 }
44
45 func (l *LinkedList) Len() int {
46     return l.len
47 }
48
49 func (l *LinkedList) Append(item int) {
50     node := new(Node)
51     node.item = item
52     // node.prev = nil 默认
53     // node.next = nil 默认
54
55     if l.head == nil {
56         // 没有元素，头尾都是nil，但凡有一个元素，头尾都不是nil
57         l.head = node
58         l.tail = node
59     } else {
60         // 哪怕是有一个元素，append都是动尾巴
61         l.tail.next = node // 手拉手1，修改当前尾巴的下一个
62         node.prev = l.tail // 手拉手2，新节点拉尾巴
63         l.tail = node      // 修改尾巴
64     }
65     l.len++ // 新增成功长度加1
66 }
67
68 func (l *LinkedList) Pop() error {
69     // 尾部弹出
70     if l.tail == nil { // 空链表
71         return errors.New("Empty")
72     } else if l.head == l.tail {
73         // 仅有一个元素
74         l.head = nil
75         l.tail = nil
76     } else {
77         tail := l.tail // 当前尾巴
78         prev := tail.prev // 当前倒数第二个
79         prev.next = nil
80         l.tail = prev
81     }
82
83     l.len-- // 弹出一个，长度减1
84     return nil
85 }
86
87 func (l *LinkedList) Insert(index int, value int) error {
88     // 使用索引插入。同学们自行实现按照值插入
89     if index < 0 {
90         return errors.New("Not negative")
91     }
92     var current *Node
93     var flag bool
94     for i, v := range l.IterNodes(false) {
95         if i == index {
96             current = v
97             flag = true

```

```

98         break
99     }
100 }
101 if !flag {
102     // 遍历完了没找到，则追加
103     // 思考一下，如果一个元素都没有，insert会怎么样
104     l.Append(value)
105     return nil
106 }
107 // 走到这里说明一定有元素，把这个元素往后挤
108 node := new(Node)
109 node.item = value
110
111 prev := current.prev
112 if index == 0 { // 开头插入，换头
113     l.head = node
114 } else {
115     prev.next = node // 手拉手
116     node.prev = prev // 手拉手
117 }
118 node.next = current // 手拉手1
119 current.prev = node // 手拉手2
120
121 l.len++
122 return nil
123 }
124
125 func (l *LinkedList) Remove(index int) error {
126     if l.tail == nil { // 空链表
127         return errors.New("Empty")
128     }
129     // 使用索引移除。同学们自行实现按照值移除
130     if index < 0 {
131         return errors.New("Not negative")
132     }
133     var current *Node
134     var flag bool
135     for i, v := range l.IterNodes(false) {
136         if i == index {
137             current = v
138             flag = true
139             break
140         }
141     }
142     if !flag {
143         // 遍历完了没找到，则报错
144         return errors.New("Out")
145     }
146     // 走到这里说明一定有元素且找到了这个元素
147     prev := current.prev
148     next := current.next
149
150     // 4种情况
151     if prev == nil && next == nil {
152         // 仅仅只有一个元素，想想还有什么等价的条件？

```

```

153     l.head = nil
154     l.tail = nil
155 } else if prev == nil { // 是开头，且多于一个元素
156     l.head = next
157     next.prev = nil
158 } else if next == nil { // 是尾巴，且多于一个元素
159     prev.next = nil
160     l.tail = prev
161 } else { // 既不是开头也不是尾巴，且多于一个元素
162     prev.next = next
163     next.prev = prev
164 }
165 l.len--
166 return nil
167 }
168
169 func (l *LinkedList) IterNodes(reversed bool) []*Node {
170     var p *Node
171     r := make([]*Node, 0, l.len)
172     if reversed {
173         p = l.tail
174     } else {
175         p = l.head
176     }
177
178     for p != nil {
179         // fmt.Println(p)
180         r = append(r, p)
181         if reversed {
182             p = p.prev
183         } else {
184             p = p.next
185         }
186     }
187     return r
188 }
189
190 func main() {
191     // 构造一个链表
192     ll := NewList()
193     ll.Append(2)
194     ll.Append(3)
195     ll.Insert(0, 0)
196     ll.Insert(1, 1)
197     ll.Append(5)
198     ll.Insert(4, 4)
199
200     fmt.Println(ll.IterNodes(false))
201     ll.Remove(1)
202     fmt.Println(ll.IterNodes(false))
203     ll.Pop()
204     fmt.Println(ll.IterNodes(false))
205 }

```

上面代码实现的主要目的是让大家学透这种非常重要的数据结构链接表。

Go语言标准库"container/list"中的List实现了双向链表。

## 栈

栈stack，一种仅在表尾进行插入、删除的线性表。先压入的数据被压在栈底，最后进入的数据压在栈顶。弹出数据时，要从栈顶弹出数据。插入数据称为进栈、压栈、入栈，弹出数据称为退栈、出栈。

栈的特点就是后进先出LIFO。

我们在线性表的时候已经分析过栈，请同学们思考一下，栈如何实现？使用顺序表还是链接表？

"container/list"中的List实现了双向链表，可以用链表作为底层数据结构来实现实现栈。

```
1 package main
2
3 import (
4     "container/list"
5     "fmt"
6 )
7
8 func main() {
9     stack := list.New() // 创建链表
10    stack.PushBack(1)    // 尾部追加，返回元素对象
11    stack.PushBack(2)
12    stack.PushBack(3)
13    stack.PushBack(4)
14    fmt.Println(stack.Front())
15    fmt.Println(stack.Back())
16    fmt.Println(stack.Len())
17
18    for r := stack.Front(); r != nil; r = r.Next() {
19        fmt.Println(r.Value)
20    }
21
22    stack.Remove(stack.Back()) // 移除尾巴
23    fmt.Println("~~~~~")
24    for r := stack.Front(); r != nil; r = r.Next() {
25        fmt.Println(r.Value)
26    }
27 }
```