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。

```go
// 单向链表
package main

import (
    "fmt"
    "strconv"
)

type Node struct {
    item int
    next *Node
    // prev *Node
}

func (n *Node) String() string {
    var next string
    if n.next == nil {
        next = "null"
    } else {
        next = strconv.Itoa(n.next.item)
    }
    return fmt.Sprintf(
        "%d ==> %s",
        n.item,
        next,
    )
}

type LinkedList struct {
    head *Node
    tail *Node
    len  int
}

func New() *LinkedList {
    return &LinkedList{}
}

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

func (l *LinkedList) Append(item int) {
    node := new(Node)
    node.item = item
    // node.next = nil 默认

```

```
    if l.head == nil {
        // 没有元素，头尾都是nil，但凡有一个元素，头尾都不是nil
        l.head = node
        l.tail = node
    } else {
        // 哪怕是有一个元素，append都是动尾巴
        l.tail.next = node // 手拉手1，修改当前尾巴的下一个
        l.tail = node      // 修改尾巴
    }

    l.len++ // 新增成功长度加1
}

func (l *LinkedList) IterNodes() {
    p := l.head
    for p != nil {
        fmt.Println(p)
        p = p.next
    }
}

func main() {
    // 构造一个链表
    ll := NewList()
    ll.Append(2)
    ll.Append(3)
    fmt.Println(ll, ll.Len)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~")

    ll.IterNodes()
}

```

有了上面代码，修改为双向链表

```
package main

import (
    "fmt"
    "strconv"
)

type Node struct {
    item int
    next *Node
    prev *Node
}

func (n *Node) String() string {
    var next, prev string
    if n.next == nil {
        next = "null"
    } else {
        next = strconv.Itoa(n.next.item)
    }
```

```go
    if n.prev == nil {
        prev = "null"
    } else {
        prev = fmt.Sprintf("%d", n.prev.item)
    }
    return fmt.Sprintf(
        "%s <== %d ==> %s",
        prev,
        n.item,
        next,
    )
}

type LinkedList struct {
    head *Node
    tail *Node
    len  int
}

func NewList() *LinkedList {
    return &LinkedList{}
}

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

func (l *LinkedList) Append(item int) {
    node := new(Node)
    node.item = item
    // node.prev = nil 默认
    // node.next = nil 默认

    if l.head == nil {
        // 没有元素，头尾都是nil，但凡有一个元素，头尾都不是nil
        l.head = node
        l.tail = node
    } else {
        // 哪怕是有一个元素，append都是动尾巴
        l.tail.next = node // 手拉手1，修改当前尾巴的下一个
        node.prev = l.tail // 手拉手2，新节点拉尾巴
        l.tail = node      // 修改尾巴
    }
    l.len++ // 新增成功长度加1
}

func (l *LinkedList) IterNodes() {
    p := l.head
    for p != nil {
        fmt.Println(p)
        p = p.next
    }
}

func main() {
```

```
    // 构造一个链表
    ll := NewList()
    ll.Append(2)
    ll.Append(3)
    fmt.Println(ll, ll.len)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")

    ll.IterNodes()

}
```

开始增加xxxx方法

```
package main

import (
    "errors"
    "fmt"
    "strconv"
)

type Node struct {
    item int
    next *Node
    prev *Node
}

func (n *Node) String() string {
    var next, prev string
    if n.next == nil {
        next = "null"
    } else {
        next = strconv.Itoa(n.next.item)
    }
    if n.prev == nil {
        prev = "null"
    } else {
        prev = fmt.Sprintf("%d", n.prev.item)
    }
    return fmt.Sprintf(
        "%s <== %d ==> %s",
        prev,
        n.item,
        next,
    )
}

type LinkedList struct {
    head *Node
    tail *Node
    len  int
}

func NewList() *LinkedList {
    return &LinkedList{}
```

```go
}

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

func (l *LinkedList) Append(item int) {
    node := new(Node)
    node.item = item
    // node.prev = nil 默认
    // node.next = nil 默认

    if l.head == nil {
        // 没有元素，头尾都是nil，但凡有一个元素，头尾都不是nil
        l.head = node
        l.tail = node
    } else {
        // 哪怕是有一个元素，append都是动尾巴
        l.tail.next = node // 手拉手1，修改当前尾巴的下一个
        node.prev = l.tail // 手拉手2，新节点拉尾巴
        l.tail = node      // 修改尾巴
    }
    l.len++ // 新增成功长度加1
}

func (l *LinkedList) Pop() error {
    // 尾部弹出
    if l.tail == nil { // 空链表
        return errors.New("Empty")
    } else if l.head == l.tail {
        // 仅有一个元素
        l.head = nil
        l.tail = nil
    } else {
        tail := l.tail    // 当前尾巴
        prev := tail.prev // 当前倒数第二个
        prev.next = nil
        l.tail = prev
    }

    l.len-- // 弹出一个，长度减1
    return nil
}

func (l *LinkedList) Insert(index int, value int) error {
    // 使用索引插入。同学们自行实现按照值插入
    if index < 0 {
        return errors.New("Not negative")
    }
    var current *Node
    var flag bool
    for i, v := range l.IterNodes(false) {
        if i == index {
            current = v
            flag = true
```

```
            break
        }
    }
    if !flag {
        // 遍历完了没找到，则追加
        // 思考一下，如果一个元素都没有，insert会怎么样
        l.Append(value)
        return nil
    }
    // 走到这里说明一定有元素，把这个元素往后挤
    node := new(Node)
    node.item = value

    prev := current.prev
    if index == 0 { // 开头插入，换头
        l.head = node
    } else {
        prev.next = node // 手拉手
        node.prev = prev // 手拉手
    }
    node.next = current // 手拉手1
    current.prev = node // 手拉手2

    l.len++
    return nil
}

func (l *LinkedList) Remove(index int) error {
    if l.tail == nil { // 空链表
        return errors.New("Empty")
    }
    // 使用索引移除。同学们自行实现按照值移除
    if index < 0 {
        return errors.New("Not negative")
    }
    var current *Node
    var flag bool
    for i, v := range l.IterNodes(false) {
        if i == index {
            current = v
            flag = true
            break
        }
    }
    if !flag {
        // 遍历完了没找到，则报错
        return errors.New("Out")
    }
    // 走到这里说明一定有元素且找到了这个元素
    prev := current.prev
    next := current.next

    // 4种情况
    if prev == nil && next == nil {
        // 仅仅只有一个元素，想想还有什么等价的条件？
```

```
        l.head = nil
        l.tail = nil
    } else if prev == nil { // 是开头，且多于一个元素
        l.head = next
        next.prev = nil
    } else if next == nil { // 是尾巴，且多于一个元素
        prev.next = nil
        l.tail = prev
    } else { // 既不是开头也不是尾巴，且多于一个元素
        prev.next = next
        next.prev = prev
    }
    l.len--
    return nil
}

func (l *LinkedList) IterNodes(reversed bool) []*Node {
    var p *Node
    r := make([]*Node, 0, l.len)
    if reversed {
        p = l.tail
    } else {
        p = l.head
    }

    for p != nil {
        // fmt.Println(p)
        r = append(r, p)
        if reversed {
            p = p.prev
        } else {
            p = p.next
        }
    }
    return r
}

func main() {
    // 构造一个链表
    ll := NewList()
    ll.Append(2)
    ll.Append(3)
    ll.Insert(0, 0)
    ll.Insert(1, 1)
    ll.Append(5)
    ll.Insert(4, 4)

    fmt.Println(ll.IterNodes(false))
    ll.Remove(1)
    fmt.Println(ll.IterNodes(false))
    ll.Pop()
    fmt.Println(ll.IterNodes(false))
}
```

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。

```go
package main

import (
    "container/list"
    "fmt"
)

func main() {
    stack := list.New() // 创建链表
    stack.PushBack(1)   // 尾部追加，返回元素对象
    stack.PushBack(2)
    stack.PushBack(3)
    stack.PushBack(4)
    fmt.Println(stack.Front())
    fmt.Println(stack.Back())
    fmt.Println(stack.Len())

    for r := stack.Front(); r != nil; r = r.Next() {
        fmt.Println(r.Value)
    }

    stack.Remove(stack.Back()) // 移除尾巴
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~")
    for r := stack.Front(); r != nil; r = r.Next() {
        fmt.Println(r.Value)
    }
}
```