Go语言list（列表）

列表是一种非连续的存储容器，由多个节点组成，节点通过一些变量记录彼此之间的关系，列表有多种实现方法，如单链表、双链表等。

列表的原理可以这样理解：假设 A、B、C 三个人都有电话号码，如果 A 把号码告诉给 B，B 把号码告诉给 C，这个过程就建立了一个单链表结构。

如果在这个基础上，再从 C 开始将自己的号码告诉给自己所知道号码的主人，这样就形成了双链表结构。

那么如果需要获得所有人的号码，只需要从 A 或者 C 开始，要求他们将自己的号码发出来，然后再通知下一个人如此循环，这样就构成了一个列表遍历的过程。

如果 B 换号码了，他需要通知 A 和 C，将自己的号码移除，这个过程就是列表元素的删除操作。

在Go语言中，列表使用 list 包来实现，内部的实现原理是双链表，列表能够高效地进行任意位置的元素插入和删除操作。

初始化列表

list 的初始化有两种方法：分别是使用 New() 函数和 var 关键字声明，两种方法的初始化效果都是一致的。

```
变量名 := list.New()
//或
var 变量名 list.List
```

列表与切片和 map 不同的是，列表并没有具体元素类型的限制，因此，列表的元素可以是任意类型，这既带来了便利，也引来一些问题，例如给列表中放入了一个 interface{} 类型的值，取出值后，如果要将 interface{} 转换为其他类型将会发生宕机。

示例：

```
package main

import (
    "container/list"
)

func main() {
    var lists list.List
    lists1 := list.New()
}
```

在列表中插入元素

双链表支持从队列前方或后方插入元素，分别对应的方法是 PushFront 和 PushBack。这两个方法都会返回一个 *list.Element 结构，如果在以后的使用中需要删除插入的元素，则只能通过 *list.Element 配合 Remove() 方法进行删除，这种方法可以让删除更加效率化，同时也是双链表特性之一。

`PushFront()`：将元素插入到列表前方。

`PushBack()`：将元素插入到列表后方。

`InsertBefore()`：调用`insertValue()`在元素`mark`前插入。要求元素`mark`属于调用的链表对象且不为`nil`。

`InsertAfter()`：与`InsertBefore()`类似，在元素`mark`后插入。

示例：

```
package main

import (
    "container/list"
)

func main() {
    var lists list.List
    e := lists.PushFront("4")
    e2 := lists.PushBack("1")
    lists.InsertBefore(3, e)
    lists.InsertAfter(2, e2)
}
```

基础方法

- `Next()`：返回**当前节点**的下一个节点或者nil。从实现可以看出，当`list`为空或者**下一个节点为哨兵节点**时，返回`nil`。
- `Prev()`：返回**当前节点**的上一个节点或者nil，跟上面类似。
- `New()`：返回一个初始化完成的链表。
- `Len()`：回链表中的元素个数，时间复杂度为O(1)。
- `Front()`：返回链表的第一个元素。当链表为空时，返回nil。
- `Back()`：返回链表的最后一个元素。当链表为空时，返回nil。

```
package main

import (
    "container/list"
    "fmt"
)

func main() {
    lists := list.New()
    e := lists.PushFront("4")
    e2 := lists.PushBack("1")
    e3 := lists.InsertBefore(3, e)
    e4 := lists.InsertAfter(2, e2)
    fmt.Printf("e3.Next().Value: %v\n", e3.Next().Value)
    fmt.Printf("e4.Prev().Value: %v\n", e4.Prev().Value)
    fmt.Printf("lists.Back().Value: %v\n", lists.Back().Value)
    fmt.Printf("lists.Front().Value: %v\n", lists.Front().Value)
    // 遍历输出链表内容
    for e := lists.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
```

```
}
#结果
e3.Next().Value: 4
e4.Prev().Value: 1
lists.Back().Value: 2
lists.Front().Value: 3
3
4
1
2
```

从列表中删除元素

列表插入函数的返回值会提供一个 *list.Element 结构，这个结构记录着列表元素的值以及与其他节点之间的关系等信息，从列表中删除元素时，需要用到这个结构进行快速删除。

- `Remove()`：当元素 `e` 属于被调用链表的元素时，调用 `remove()` 函数，返回 `e.Value`。

```
package main

import (
    "container/list"
    "fmt"
)

func main() {
    var lists list.List
    e := lists.PushFront("4")
    e2 := lists.PushBack("1")
    e3 := lists.InsertBefore(3, e)
    lists.InsertAfter(2, e2)
    // 遍历输出链表内容
    for e := lists.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
    lists.Remove(e3) //删除
    fmt.Println("--------------------")
    // 遍历输出链表内容对比之前
    for e := lists.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
}
#结果
3
4
1
2
--------------------
4
1
2
```

移动元素

- `MoveToFront()`：将元素`e`移至表头。要求元素`e`属于被调用链表的元素。
- `MoveToBack()`：将元素`e`移至表尾。要求元素`e`属于被调用链表的元素。
- `MoveBefore()`：将元素`e`移至元素`mark`的前面。要求元素`e`属于被调用链表的元素且不为元素`mark`。
- `MoveAfter()`：与`MoveBefore()`类似，移至其后面。

```
package main

import (
    "container/list"
    "fmt"
)

func main() {
    var lists list.List
    e := lists.PushFront("4")
    e2 := lists.PushBack("1")
    e3 := lists.InsertBefore(3, e)
    e4 := lists.InsertAfter(2, e2)

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

    lists.MoveAfter(e3, e4) // 将e3移动到e4后
    lists.MoveToBack(e)     // 将e移动到列表最后

    fmt.Println("-----------")
    // 遍历输出链表内容对比
    for e := lists.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
}
#结果
3
4
1
2
-----------
1
2
3
4
```

合并链表

- `PushBackList()`：将另一个链表**浅拷贝**到当前链表的末尾。要求两个链表均不为`nil`。
- `PushFronList()`：将另一个链表**浅拷贝**到当前链表的开头。要求两个链表均不为`nil`。

```
package main
```

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

func main() {
    var lists list.List
    lists.PushBack("a")
    lists.PushBack("b")
    lists.PushBack("c")
    lists.PushBack("d")

    stack := list.New()
    stack.PushBack(1)
    stack.PushBack(2)
    stack.PushBack(3)
    stack.PushBack(4)

    lists.PushBackList(stack)//将stack列表合并到lists列表末尾

    // 遍历输出链表内容
    for e := lists.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
}
#结果:
a
b
c
d
1
2
3
4
```

遍历列表——访问列表的每一个元素

遍历双链表需要配合 Front() 函数获取头元素，遍历时只要元素不为空就可以继续进行，每一次遍历都会调用元素的 Next() 函数。

```
package main

import (
    "container/list"
    "fmt"
)

func main() {
    var lists list.List
    lists.PushBack("a")
    lists.PushBack("b")
    lists.PushBack("c")
    lists.PushBack("d")
```

```
    // 遍历输出链表内容
    for e := lists.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
}
#结果
a
b
c
d
```

栈

我们知道栈有三个基本操作：`Push()`、`Pop()`和`Top()`，分别是向栈顶压入，弹出，和查看栈顶操作。我们可以用`list`的方法来模拟这三个基本操作：`s.PushBack()`、`s.Back().Value`和`s.Remove(s.Back())`。

```
package main

import (
    "container/list"
    "fmt"
)

func main() {
    stack := list.New()
    stack.PushBack(1)
    stack.PushBack(2)
    stack.PushBack(3)
    stack.PushBack(4)
    for stack.Len() > 0 {
        fmt.Println(stack.Remove(stack.Back()))
    }
}
#结果
4
3
2
1
```

队列

同样，我们还可以用`q.PushBack()`与`q.Remove(q.Front())`来模拟队列的进度与出队操作。

```
package main

import (
    "container/list"
    "fmt"
)

func main() {
    queue := list.New()
    queue.PushBack(1)
```

```
    queue.PushBack(2)
    queue.PushBack(3)
    queue.PushBack(4)
    for queue.Len() > 0 {
        fmt.Println(queue.Remove(queue.Front()))
    }
}
#结果
1
2
3
4
```