链表

基本概述

链表的特点

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。 **链表**需要动态开辟空间，在执行删除或插入操作时效率高。

主要分类

- 单链表
- 双向链表
- 循环链表

应用场景：不需要预先知道数据规模；适应于频繁的删除插入操作。

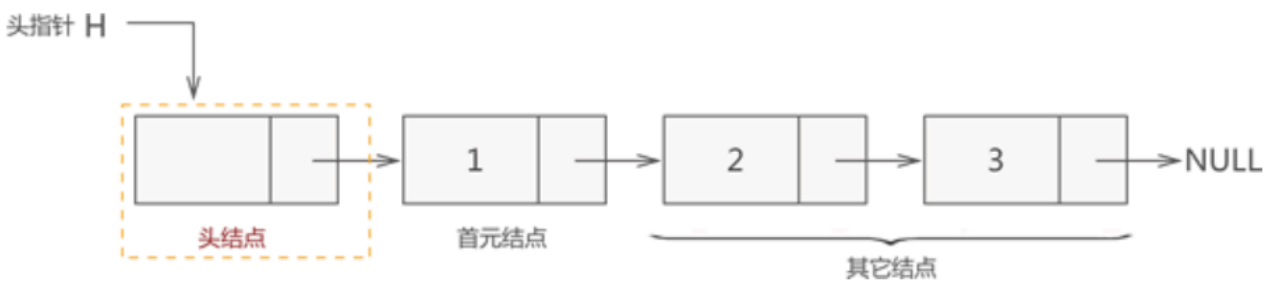
注意 其他的数据结构：栈、队列、集合、哈希表和图都要依赖于链表

三种链表的介绍

单链表

简称链表，由各个数据成员通过一个指针彼此链接起来而组成。每个元素包括两部分：数据成员和一个称为next的指针。只能以一个方向进行遍历，从头到尾。
链表中存储各数据元素的结构：

链表结构：

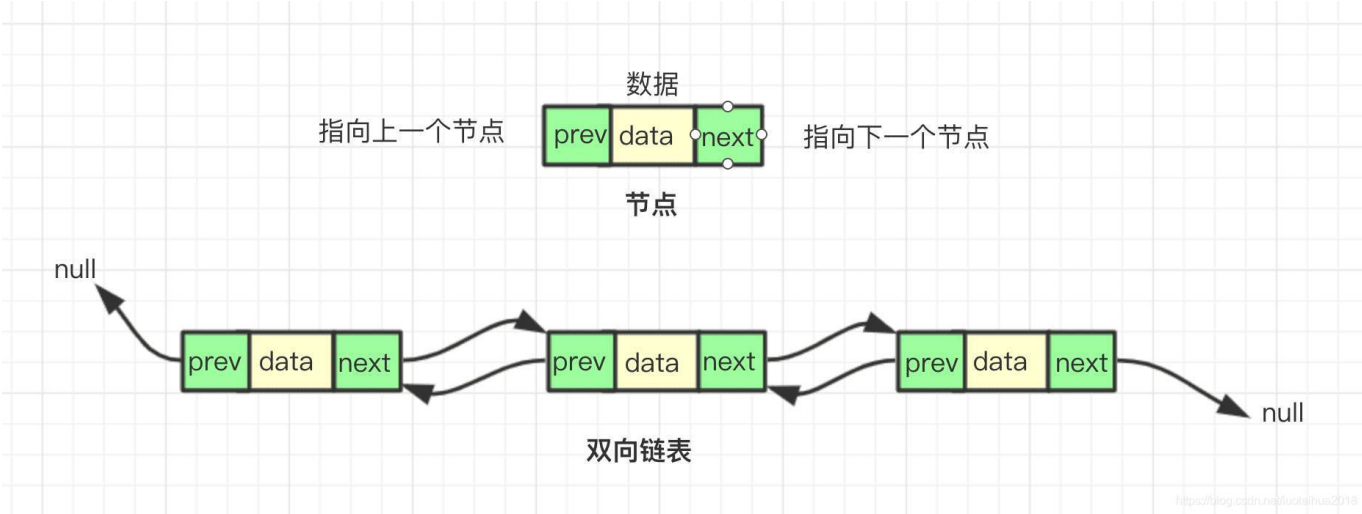


1. 头指针：一个普通的指针，它的特点是永远指向链表第一个节点的位置。很明显，头指针用于指明链表的位置，便于后期找到链表并使用表中的数据；

2. 节点：链表中的节点又细分为头节点、首元节点和其他节点；
    - 头节点：其实就是一个不存任何数据的空节点，通常作为链表的第一个节点。对于链表来说，头节点不是必须的，它的作用只是为了方便解决某些实际问题；
    - 首元节点：由于头节点（也就是空节点）的缘故，链表中称第一个存有数据的节点为首元节点。首元节点只是对链表中第一个存有数据节点的一个称谓，没有实际意义；
    - 其他节点：链表中其他的节点；
      注意：链表中有头节点时，头指针指向头节点；反之，若链表中没有头节点，则头指针指向首元节点。

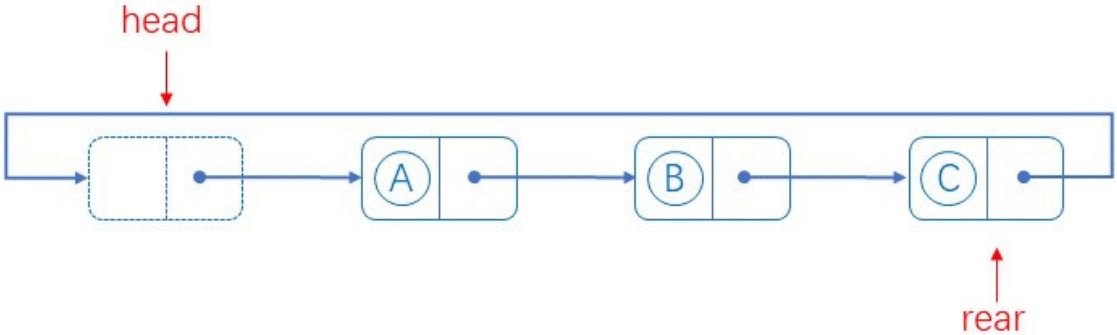
双向链表

链表元素由两个指针链接。每个元素有三部分组成：除了数据成员和next指针外，还有包含一个指向前一个元素的指针prev。为了标识头和尾，将第一个元素的prev指针和最后一个元素的next指针置为NULL。它可以从头到尾，也可以从尾到头遍历整个链表。
循坏列表结构：



循坏链表

循环链表可以是单向的也可以是双向的。区分一个链表是不是循环链表只要看它有没有尾部元素即可。
在循环链表中，最后一个元素指向头元素，而不是NULL；如果是双向的循环链表，它的头元素的prev要指向最后一个元素，而不是NULL。
单向循坏列表结构



注意：插入元素的时候会遇到从表头插入、表尾插入和指定角标处插入，这些操作基本上和单链表的元素插入类似，但是需要注意的是从表头和表尾插入的时候，要考虑头指针和尾指针的移动，还有尾结点要指向头结点形成新的循环链表。

单向循坏列表插入演示：

双向循坏列表结构

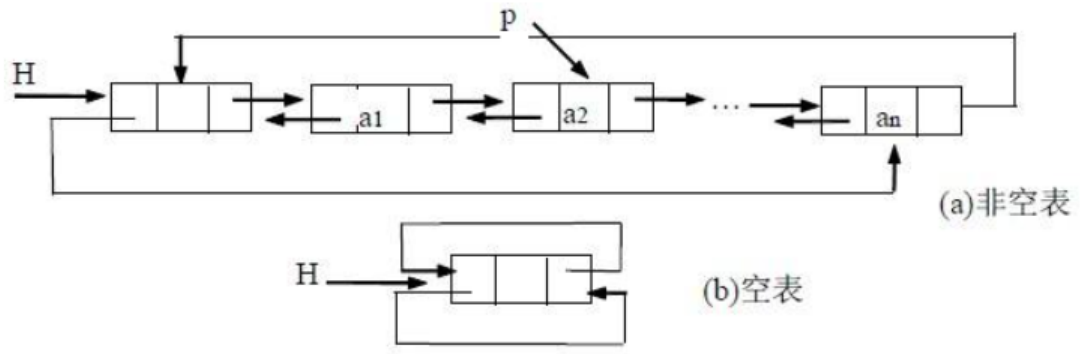


图2.19 带头结点的双循环链表

链表的基本操作(以go的标准库中container/list所实现的双向链表讲解)

Go 语言的链表实现在其标准库的container/list代码包中。这个包包含了两个公开的程序实体：List和Element. 前者实现了一个双向链表（以下简称链表），而后者则代表了链表中元素的结构。

List Element结构如下：

```
type Element struct {
    next, prev *Element  // 上一个元素和下一个元素
    list *List  // 元素所在链表
    Value interface{}  // 元素
}

type List struct {
    root Element  // 链表的根元素
    len  int      // 链表的长度
}
```

list方法如下：

```
type Element
    func (e *Element) Next() *Element
    func (e *Element) Prev() *Element
type List
    func New() *List
    func (l *List) Back() *Element   // 最后一个元素
    func (l *List) Front() *Element  // 第一个元素
    func (l *List) Init() *List  // 链表初始化
    func (l *List) InsertAfter(v interface{}, mark *Element) *Element // 在某个元素后插入
    func (l *List) InsertBefore(v interface{}, mark *Element) *Element  // 在某个元素前插入
    func (l *List) Len() int // 在链表长度
    func (l *List) MoveAfter(e, mark *Element)  // 把 e 元素移动到 mark 之后
    func (l *List) MoveBefore(e, mark *Element)  // 把 e 元素移动到 mark 之前
    func (l *List) MoveToBack(e *Element) // 把 e 元素移动到队列最后
    func (l *List) MoveToFront(e *Element) // 把 e 元素移动到队列最头部
    func (l *List) PushBack(v interface{}) *Element  // 在队列最后插入元素
    func (l *List) PushBackList(other *List)  // 在队列最后插入接上新队列
    func (l *List) PushFront(v interface{}) *Element  // 在队列头部插入元素
    func (l *List) PushFrontList(other *List) // 在队列头部插入接上新队列
    func (l *List) Remove(e *Element) interface{} // 删除某个元素
```

链表方法讲解

---

1. `func New() *List`

```
//初始化一个列表，返回类型为一个list对象
 var a = list.New()
 b := list.New()
 fmt.Printf("%+v\n",l)
```

2. `func (l *List) PushBack(v interface{}) *Element{}` 和 `func (l *List) PushFront(v interface{}) *Element{}`

```
//Push（Back/Front）在列表l的（后/前）面插入一个值为v的新元素e并返回e。
  a1 := l.PushBack("a")
  a2 := l.PushBack(1)
  a3 := l.PushBack("c")
  fmt.Printf("%+v\n",l)
  fmt.Printf("%+v\n",a1)
  fmt.Printf("%+v\n",a2)
  fmt.Printf("%+v\n",a3)
  a0 := l.PushFront("开始")
  fmt.Printf("%+v\n",a0)
```

3. `func (l *List) Back() *Element{}` 和 `func (l *List) Front() *Element{}`

```
//返回链表的最后一个元素，返回类型元列表中的元素。
  x1 := l.Back()
  fmt.Println(strings.Repeat("-----------", 10))
  fmt.Println(x1)
//返回链表的第一个元素，返回类型元列表中的元素。
  x2 := l.Front()
  fmt.Println(strings.Repeat("-----------", 10))
  fmt.Println(x2)
```

4. `func (l *List) Init() *List{}`

```
//初始化此链表，将元素全部清空
  l.Init()
  fmt.Printf("%+v\n",l)
```

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

```
//返回链表的长度，返回值类型为int
  fmt.Println(l.Len())
```

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

```
//如果e是列表l的元素，则从l中删除e。它返回元素值e.Value。元素不能为零。
```

7. `func (l *List) InsertAfter(v interface{}, mark *Element) *Element{}` 和 `func (l *List) InsertBefore(v interface{}, mark *Element) *Element{}`

```
  //Insert（After/Before）在mark之（后/前）立即插入一个值为v的新元素e并返回e。如果mark不是l的元素，则不修改列表。mark不得为零。
```

```
    l := list.New()
    x1 := l.PushBack("b")
    l.InsertAfter("c",x1)
    fmt.Println(l.Back())
    l.InsertBefore("a",x1)
    fmt.Println(l.Front())
```

8. `func (l *List) MoveAfter(e, mark *Element){}` 和 `func (l *List) MoveBefore(e, mark *Element){}`

```
    //Move(After/Before)将标记(后/前)的元素e移动到新位置。如果e或mark不是l或e == mark的元素，则不修改列表。元素和标记不得为零。
    l := list.New()
    x1 := l.PushBack("b")
    x2 := l.PushBack("a")
    l.MoveAfter(x1,x2)
    fmt.Printf("%+v\n",l.Back())
    l.MoveBefore(x1,x2)
    fmt.Printf("%+v\n",l.Back())
```

9. `func (l *List) MoveToBack(e *Element){}` 和 `func (l *List) MoveToFront(e *Element){}`

```
    //MoveTo（Back/Front）将标记（后/前）的元素e移动(前面或者后面)。如果e或mark不是l或e == mark的元素，则不修改列表。元素和标记不得为零。
    l := list.New()
    x1 :=  l.PushBack("b")
    l.PushBack("a")
    l.MoveToBack(x1)
    fmt.Println(l.Back())
    l.MoveToFront(x1)
    fmt.Println(l.Back())
```

10. `func (l *List) PushBackList(other *List){}` 和 `func (l *List) PushFrontList(other *List){}`

```
    //Push（Back/Front）List在列表l的（后/前）面插入另一个列表的副本。列表l和其他可以是相同的。他们一定不能为零。
    l := list.New()
    l.PushBack("a")
    l.PushBack("b")
    l.PushBack("c")
    l0 := list.New()
    l0.PushBack("d")
    l0.PushBack("e")
    l.PushBackList(l0)
    fmt.Println(l.Back())
```

```
l.PushFrontList(l0)
fmt.Println(l.Front())
```

链表元素方法讲解

1. `func (e *Element) Next() *Element{}`

```
//Next返回下一个list元素或nil
l := list.New()
x1 := l.PushBack("a")
l.PushBack("b")
l.PushBack("c")
fmt.Println(x1.Next())
```

2. `func (e *Element) Prev() *Element{}`

```
//Prev返回下一个list元素或nil
l := list.New()
l.PushBack("a")
x1 := l.PushBack("b")
l.PushBack("c")
fmt.Println(x1.Prev())
```

3. 取出链表中元素的值

```
//Prev返回下一个list元素或nil
l := list.New()
x1 := l.PushBack("a")
fmt.Println(x1.Value)
```

链表遍历

```
//Prev返回下一个list元素或nil
l := list.New()
for i := 1;i <= 6;i++{
    l.PushBack(i)
}
for e := l.Front(); e != nil; e = e.Next() {
    fmt.Println(e.Value)
}
```

对于链表与切片的总结。

链表适合于频繁存取的数据结构中。
切片适合于频繁查询的数据结构中

作业