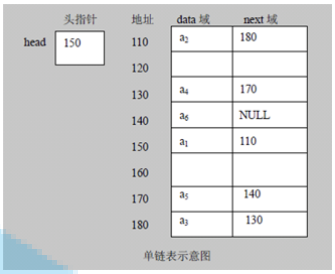
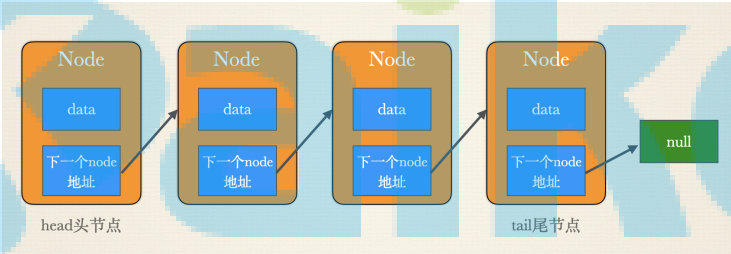
day03–数据结构之链表(外化版)

一、链表介绍

- 动态数组有明显的缺点：可能会造成内存的浪费
- 是否可以用多少申请多少内存：链表可以
- **链表是一种链式存储的线性表，所有元素的内存地址不一定是连续的**





1. 头节点是列表开头的节点
2. 尾节点的特征是其 next 引用为空（null）
3. 当节点的地址为空时，链表终止。

Node节点

链表中的每一个内存块被称为节点Node，Node节点由两部分构成：

- 存储数据(可以是任何类型)
- 还需记录链上下一个节点的地址，即后继指针next，用来存储下一个节点的Node对象

链表的优缺点

数组的插入、删除操作时，为了保持内存数据的连续性，需要做大量的数据搬移，所以时间复杂度是 O(n)

在链表中插入和删除一个数据是非常快速的，我们只需要考虑相邻结点的指针改变，所以对应的时间复杂度是 O(1)。

二、常用链表

最常见的链表结构，它们分别是：单链表、双向链表和循环链表。

1.单链表

1）每个节点只包含一个指针，即后继指针。
2）单链表有两个特殊的节点，即首节点和尾节点。用首节点地址表示整条链表，尾节点的后继指针指向空地址null。
3）性能特点：插入和删除节点的时间复杂度为O（1），查找的时间复杂度为O(n)。

2.循环链表

1）除了尾节点的后继指针指向首节点的地址外均与单链表一致。
2）适用于存储有循环特点的数据，比如约瑟夫问题。

3.双向链表

1）节点除了存储数据外，还有两个指针分别指向前一个节点地址（前驱指针prev）和下一个节点地址（后继指针next）。
2）首节点的前驱指针prev和尾节点的后继指针均指向空地址。
3）性能特点：
和单链表相比，存储相同的数据，需要消耗更多的存储空间。
插入、删除操作比单链表效率更高O(1)级别。

> 以删除操作为例，删除操作分为2种情况：给定数据值删除对应节点和给定节点地址删除节点。
> **第一种情况**：单链表和双向链表都需要从头到尾进行遍历从而找到对应节点进行删除，时间复杂度为O(n)。
> **第二种情况**：要进行删除操作必须找到前驱节点，单链表需要从头到尾进行遍历直到p->next = q，时间复杂度为O(n)，而双向链表可以直接找到前驱节点，时间复杂度为O(1)。
> 对于一个有序链表，双向链表的按值查询效率要比单链表高一些。因为我们可以记录上次查找的位置p，每一次查询时，根据要查找的值与p的大小关系，决定是往前还是往后查找，所以平均只需要查找一半的数据。

4.双向循环链表

- 在双向循环链表中，可见的不只有头指针head，还有尾节点end。这是和单链表的区别。
- 双向循环链表的头指针head的前一个节点指向end，尾节点end的后一个节点指向head。

三、自定义单向链表

设计接口

```
int size(); // 元素的数量
boolean isEmpty(); // 是否为空
int indexOf(E element); // 查看元素的位置
boolean contains(E element); // 是否包含某个元素
E get(int index); // 返回index位置对应的元素
E set(int index, E element); // 设置index位置的元素
void clear(); // 清除所有元素
void add(E element); // 添加元素到最后面
void add(int index, E element); // 往index位置添加元素
E remove(int index); // 删除index位置对应的元素
```

List接口

包含共性的方法

```
package com.kkb.day02;
public interface List<E> {
    public int  size();
    public int indexOf(E element); // 查看元素的位置
    public boolean contains(E element); // 是否包含某个元素
    public E get(int index); // 返回index位置对应的元素
    public E set(int index, E element); // 设置index位置的元素
    public void clear(); // 清除所有元素
    public void add(E element); // 添加元素到最后面
    public void add(int index, E element); // 往index位置添加元素
    public E remove(int index); // 删除index位置对应的元素
    public boolean isEmpty(); // 是否为空
}
```

AbstractList类

```
package com.kkb.day02;
public  abstract class AbstractList<E> implements List<E>{
    protected int size;
    protected static final int ELEMENT_NOT_FOUND=-1;
    @Override
    public int size() {
        return size;
    }
    @Override
    public boolean contains(E element) {
        return indexOf(element)!=ELEMENT_NOT_FOUND;
    }
    @Override
    public boolean isEmpty() {
        return size==0;
    }
    @Override
    public void add(E element) {
        add(size,element);
    }
    protected void checkAddIndex(int index) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("索引越界" + "允许范围
size: 0 => " + (size) + " 当前索引: " + index);
        }
    }
    protected void checkIndex(int index){
        if (index < 0 || index >= size) {
```

```
            throw new IndexOutOfBoundsException("索引越界" + "允许范围 size: 0 => " + (size - 1) + " 当前索引: " + index);
        }
    }
}
```

MyLinkedList类

```
package com.kkb.day02;
public class MyLinkedList<E> extends AbstractList<E>{
    private Node<E> first;
    private static class Node<E>{
        Node<E> next;
        E element;
        public Node(Node<E> next, E element) {
            this.next = next;
            this.element = element;
        }
    }
    @Override
    public E get(int index) {
        checkIndex(index);
        return node(index).element;
    }
    private Node<E> node(int index){
        checkIndex(index);
        Node<E> node=first;
        for (int i = 0; i <index; i++) {
            node=node.next;
        }
        return node;
    }
    @Override
    public E set(int index, E element) {
        checkIndex(index);
        Node<E> node=node(index);
        E oldElement=node.element;
        node.element=element;
        return oldElement;
    }
    @Override
    public void clear() {
        size=0;
        first=null;
    }
    @Override
    public int indexOf(E element) {
        Node<E> node=first;
        if (element == null) {
```

```
            //通过遍历找到下一个节点，判断元素是否相等，相等返回i即可
            for (int i = 0; i < size; i++) {//遍历查找
                if (node.element == null) {//找到为null返回其下标
                    return i;
                }
                node=node.next;
            }
        } else {
            for (int i = 0; i < size; i++) {
                if (element.equals(node.element)) {
                    return i;
                }
                node=node.next;
            }
        }
        //没有找到元素返回-1
        return ELEMENT_NOT_FOUND;
    }
    @Override
    public void add(int index, E element) {
        checkAddIndex(index);
        if(index==0){
            first=new Node(first,element);
        }else{
            Node<E> pre=node(index-1);
            Node<E>  next=pre.next;
            pre.next=new Node(next,element);
        }
        size++;
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("size:" + size + "->[");
        Node<E> node=first;
        for (int i = 0; i <size ; i++) {
            if(i!=0){
                sb.append(", ");
            }
            sb.append(node.element);
            node=node.next;
        }
        sb.append("]");//拼接末尾
        return sb.toString();//返回输出结果
    }
    @Override
    public E remove(int index) {
        checkIndex(index);
        Node<E> oldnode=first;
```

```
        if(index==0){
            first=first.next;
        }else{
            Node<E> pre=node(index-1);
            oldnode=pre.next;
            pre.next=oldnode.next;
        }
        size--;
        return oldnode.element;
    }
}
```

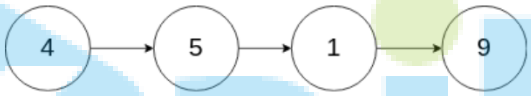
四、力扣

https://leetcode-cn.com/problems/delete-node-in-a-linked-list/

删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为 要被删除的节点 。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:



示例 1：

```
输入：head = [4,5,1,9], node = 5
输出：[4,1,9]
解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2：

```
输入：head = [4,5,1,9], node = 1
输出：[4,5,9]
解释：给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

提示：

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。

题解：

```
class Solution {

    public void deleteNode(ListNode node) {

        node.val = node.next.val;

        node.next = node.next.next;

    }

}
```

五、LinkedList源码解析（基于jdk1.8）

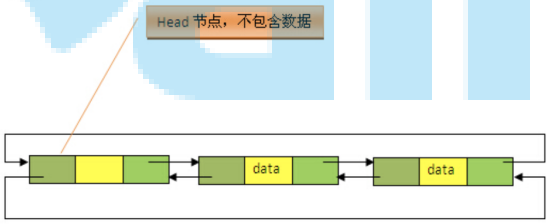
- LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。
- LinkedList 实现 List 接口，能对它进行队列操作。
- LinkedList 实现 Deque 接口，即能将LinkedList当作双端队列使用。
- LinkedList 实现了Cloneable接口，即覆盖了函数clone()，能克隆。
- LinkedList 实现java.io.Serializable接口，这意味着LinkedList支持序列化，能通过序列化去传输。

为什么要继承自AbstractSequentialList？

AbstractSequentialList 实现了get(int index)、set(int index, E element)、add(int index, E element) 和 remove(int index)这些骨干性函数。降低了List接口的复杂度。**这些接口都是随机访问List的**，LinkedList是双向链表；既然它继承于AbstractSequentialList，就相当于已经实现了“get(int index)这些接口”。

此外，我们若需要通过AbstractSequentialList自己实现一个列表，只需要扩展此类，并提供 listIterator() 和 size() 方法的实现即可。若要实现不可修改的列表，则需要实现列表迭代器的 hasNext、next、hasPrevious、previous 和 index 方法即可。

LinkedList底层的数据结构是基于双向循环链表的，且头结点中不存放数据,如下：



类成员

LinkedList内部有两个引用，一个 `first`，一个 `last`，分别用于指向链表的头和尾，另外有一个 `size`，用于标识这个链表的长度，而它的接的引用类型是 `Node`,这是他的一个内部类： `item` 用于保存数据，而 `prve` 用于指向当前节点的前一个节点， `next` 用于指向当前节点的下一个节点。

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    transient int size = 0;
    transient Node<E> first;
    transient Node<E> last;
}
```

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

构造方法

LinkedList提供了两个构造器，ArrayList比它多提供了一个通过设置初始化容量来初始化类。LinkedList不提供该方法的原因：因为LinkedList底层是通过链表实现的，每当有新元素添加进来的时候，都是通过链接新的节点实现的，也就是说它的容量是随着元素的个数的变化而动态变化的。而ArrayList底层是通过数组来存储新添加的元素的，所以我们可以为ArrayList设置初始容量（实际设置的数组的大小）。

空构造器

```
    public LinkedList() {
    }
```

有参构造

传入一个集合（Collection）作为参数初始化LinkedList。

```
// 首先调用一下空的构造器。
//然后调用addAll(c)方法。
  public LinkedList(Collection<? extends E> c) {
        this();
        addAll(c);
  }
//通过调用addAll(int index, Collection<? extends E> c) 完成集合的添加。
  public boolean addAll(Collection<? extends E> c) {
        return addAll(size, c);
  }
```

addAll(int index, Collection c)

```
public boolean addAll(int index, Collection<? extends E> c) {
    //在指定位置添加或者删除或修改操作都需要判断传进来的参数是否合法
    checkPositionIndex(index);
    //先把集合转化为数组，然后为该数组添加一个新的引用（Objext[] a）。
    Object[] a = c.toArray();
```

```
    //新建一个变量存储数组的长度。
    int numNew = a.length;
    //如果待添加的集合为空，直接返回，无需进行后面的步骤。后面都是用来把集合中的元素添加到LinkedList中。
    if (numNew == 0)
        return false;
    //Node<E> succ：指代待添加节点的位置。
    //Node<E> pred：指代待添加节点的前一个节点。
    Node<E> pred, succ;
    //如果index==size;说明此时需要添加LinkedList中的集合中的每一个元素都是在LinkedList最后面。所以把succ设置为空，pred指向尾节点。
    if (index == size) {
        succ = null;
        pred = last;
    //否则的话succ指向插入待插入位置的节点。
    } else {
        succ = node(index);//方法返回对应索引位置上的Node（节点）
        //pred指向succ节点的前一个节点
        pred = succ.prev;
    }
//接着遍历数组中的每个元素。在每次遍历的时候，都新建一个节点，该节点的值存储数组a中遍历的值，该节点的prev用来存储pred节点，next设置为空。接着判断一下该节点的前一个节点是否为空，如果为空的话，则把当前节点设置为头节点。否则的话就把当前节点的前一个节点的next值设置为当前节点。最后把pred指向当前节点，以便后续新节点的添加。
    for (Object o : a) {
        @SuppressWarnings("unchecked") E e = (E) o;
        Node<E> newNode = new Node<>(pred, e, null);
        if (pred == null)
            first = newNode;
        else
            pred.next = newNode;
        pred = newNode;
    }
//当succ==null（也就是新添加的节点位于LinkedList集合的最后一个元素的后面），通过遍历上面的a的所有元素，此时pred指向的是LinkedList中的最后一个元素，所以把last指向pred指向的节点。当不为空的时候，表明在LinkedList集合中添加的元素，需要把pred的next指向succ上，succ的prev指向pred。最后把集合的大小设置为新的大小。modCount（修改的次数）自增。
    if (succ == null) {
        last = pred;
    } else {
        pred.next = succ;
```

```
        succ.prev = pred;
    }
    size += numNew;
    modCount++;
    return true;
}
```

linkFirst(E e)

把参数中的元素作为链表的第一个元素。

```
//因为我们需要把该元素设置为头节点，所以需要新建一个变量把头节点存储起来。然后新建一个节点，把next指向f，然后自身设置为头结点。再判断一下f是否为空，如果为空的话，说明原来的LinkedList为空，所以同时也需要把新节点设置为尾节点。否则就把f的prev设置为newNode。size和modCount自增。
    private void linkFirst(E e) {
        final Node<E> f = first;
        final Node<E> newNode = new Node<>(null, e, f);
        first = newNode;
        if (f == null)
            last = newNode;
        else
            f.prev = newNode;
        size++;
        modCount++;
    }
```

linkLast(E e)

```
//因为我们需要把该元素设置为尾节点，所以需要新建一个变量把尾节点存储起来。然后新建一个节点，把last指向l，然后自身设置为尾结点。再判断一下l是否为空，如果为空的话，说明原来的LinkedList为空，所以同时也需要把新节点设置为头节点。否则就把l的next设置为newNode。size和modCount自增。
    void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

linkBefore(E e, Node<E> succ)

首先我们需要新建一个变量指向succ节点的前一个节点，因为我们要在succ前面插入一个节点。接着新建一个节点，它的prev设置为我们刚才新建的变量，后置节点设置为succ。然后修改succ的prev为新节点。接着判断一个succ的前一个节点是否为空，如果为空的话，需要把新节点设置为为头结点。如果不为空，则把succ的前一个节点的next设置为新节点。

```
void linkBefore(E e, Node<E> succ) {
    // assert succ != null;
    final Node<E> pred = succ.prev;
    final Node<E> newNode = new Node<>(pred, e, succ);
    succ.prev = newNode;
    if (pred == null)
        first = newNode;
    else
        pred.next = newNode;
    size++;
    modCount++;
}
```

unlinkFirst(Node<E> f)

删除LinkedList中第一个节点。（该节点不为空）（并且返回删除的节点的值）

官方文档的代码中也给出了注释：使用该方法的前提是参数f是头节点，而且f不能为空。它是私有方法，我们也没有权限使用。

```
  private E unlinkFirst(Node<E> f) {
//定义一个变量element指向待删除节点的值，接着定义一个变量next指向待删除节点的下一个节点。（因为我们需要设置f节点的下一个节点为头结点，而且需要把f节点的值设置为空）接着把f的值和它的next设置为空，把它的下一个节点设置为头节点。接着判断一个它的下一个节点是否为空，如果为空的话，则需要把last设置为空。否则的话，需要把next的prev设置为空，因为next现在指代头节点。
        final E element = f.item;
        final Node<E> next = f.next;
        f.item = null;
        f.next = null;
        first = next;
        if (next == null)
            last = null;
        else
            next.prev = null;
        size--;
        modCount++;
        return element;
```

```
    }
```

unlinkLast(Node<E> l)

删除LinkedList的最后一个节点。（该节点不为空）（并且返回删除节点对应的值）

和unlinkFirst（）方法思路差不多。

```
  private E unlinkLast(Node<E> l) {
        // assert l == last && l != null;
        final E element = l.item;
        final Node<E> prev = l.prev;
        l.item = null;
        l.prev = null; // help GC
        last = prev;
        if (prev == null)
            first = null;
        else
            prev.next = null;
        size--;
        modCount++;
        return element;
    }
```

unlink(Node<E> x)

删除一个节点（该节点不为空）

删除LinkedList中的一个节点，都需要把该节点的前、后节点重新链接起来，不能让链表断开，所以需要我们新建几个变量来保存他们的状态。

所以，下面新建了三个变量，第一个变量用来存储当前被删除节点的值，因为我们最后需要把这个返回回去。第二个变量用来存储待删除节点的前一个节点，第三个变量用来存储待删除节点的后一个节点。

接下来判断一下，待删除节点的前一个节点是否为空，如果为空的话，表明待删除的节点是我们的头结点。则需要把待删除节点的后一个节点设置为头结点。如果不为空，就需要把待删除的节点的前、后节点链接起来，此刻只需要链接一部分，通过pre.next=next；x.prev= null设置，接着判断一下待删除的节点是否为空，如果为空的话，则表明待删除节点是尾节点，所以需要我们把待删除节点的前一个节点设置为尾节点。如果不为空的，则把next.prev=prev，x.next=null。最后把待删除节点的值设置为空。切记不要忘了把size和modCount自减，最后返回被删除节点的值。

```
    E unlink(Node<E> x) {
        // assert x != null;
        final E element = x.item;
```

```
        final Node<E> next = x.next;
        final Node<E> prev = x.prev;
        if (prev == null) {
            first = next;
        } else {
            prev.next = next;
            x.prev = null;
        }
        if (next == null) {
            last = prev;
        } else {
            next.prev = prev;
            x.next = null;
        }
        x.item = null;
        size--;
        modCount++;
        return element;
    }
```

node(int index)

计算指定索引上的节点（返回Node）

LinkedList还对整个做了优化，不是盲目地直接从头进行遍历，而是先比较一下index更靠近链表（LinkedList）的头节点还是尾节点。然后进行遍历，获取相应的节点。

```
Node<E> node(int index) {
        // assert isElementIndex(index);
        if (index < (size >> 1)) {
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }
```

removeFirst()

提供给用户使用的删除头结点，并返回删除的值。

直接调用了上面的工具方法unlinkFirst（Node f）

```
public E removeFirst() {
        final Node<E> f = first;
        if (f == null)
            throw new NoSuchElementException();
        return unlinkFirst(f);
}
```

removeLast()

删除链表中的最后一个节点，并返回被删除节点的值。
和上面一样调用了unlinkLast（Node last）方法。

```
public E removeLast() {
        final Node<E> l = last;
        if (l == null)
            throw new NoSuchElementException();
        return unlinkLast(l);
    }
```

addFirst(E e)

在LinkedList头部添加一个新的元素、尾部添加一个新元素。
都是调用了私有方法。

```
public void addFirst(E e) {
        linkFirst(e);
}
public void addLast(E e) {
        linkLast(e);
}
```

contains(Object o)

判断LinkedList是否包含某一个元素。
底层通过调用indexof()。该方法主要用于计算元素在LinkedList中的位置。
其实indexof （）方法也非常简单：
首先依据obejct是否为空，分为两种情况：
然后通过在每种情况下，从头节点开始遍历LinkedList，判断是否有与object相等的元素，如果有，则返回对应的位置index，如果找不到，则返回-1。

```
public boolean contains(Object o) {
        return indexOf(o) != -1;
    }
public int indexOf(Object o) {
        int index = 0;
        if (o == null) {
            for (Node<E> x = first; x != null; x = x.next) {
                if (x.item == null)
                    return index;
```

```
                index++;
            }
        } else {
            for (Node<E> x = first; x != null; x = x.next) {
                if (o.equals(x.item))
                    return index;
                index++;
            }
        }
        return -1;
    }
```

add(E e)方法

添加一个新元素。直接在最后面添加，调用了linkLast（）方法。

```
public boolean add(E e) {
        linkLast(e);
        return true;
}
```

remove(Object o)

从LinkedList中删除指定元素。（且只删除第一次出现的指定的元素）（如果指定的元素在集合中不存在，则返回false，否则返回true）

该方法也是通过object是否为空分为两种情况去和LinkedList中的每一个元素比较，如果找到了，就删掉，返回true即可。如果找不到，则返回false。

```
 public boolean remove(Object o) {
        if (o == null) {
            for (Node<E> x = first; x != null; x = x.next) {
                if (x.item == null) {
                    unlink(x);
                    return true;
                }
            }
        } else {
            for (Node<E> x = first; x != null; x = x.next) {
                if (o.equals(x.item)) {
                    unlink(x);
                    return true;
                }
            }
        }
```

```
        return false;
    }
```

clear()

清空LinkedList中的所有元素，该方法也简单，直接遍历整个LinkedList，然后把每个节点都置空。最后要把头节点和尾节点设置为空，size也设置为空，但是modCount仍然自增。

```
    public void clear() {

        for (Node<E> x = first; x != null; ) {
            Node<E> next = x.next;
            x.item = null;
            x.next = null;
            x.prev = null;
            x = next;
        }
        first = last = null;
        size = 0;
        modCount++;
    }
```

get(int index)

获取对应index的节点的值。

通过node（）方法返回其值。（node（）方法依据索引的值返回其对应的节点。）

```
    public E get(int index) {
        checkElementIndex(index);
        return node(index).item;
    }
```

set(int index, E element)

设置对应index的节点的值。

首先检查一下索引是否合法，然后通过node（）方法求出旧值，然后设置新值。最后把旧值返回回去。

```
public E set(int index, E element) {
        checkElementIndex(index);
        Node<E> x = node(index);
        E oldVal = x.item;
        x.item = element;
        return oldVal;
```

```
    }
```

add(int index, E element)

在指定的位置上添加新的元素。

在方法中先判断新添加的元素是否是位于LinkedList的最后，然后是，则直接调用linkLast（）方法添加即可。否则的话，调用linkBefore（）添加即可。

```
public void add(int index, E element) {
        checkPositionIndex(index);
        if (index == size)
            linkLast(element);
        else
            linkBefore(element, node(index));
    }
```

remove(int index)

移除指定位置上的元素

还是调用上面的方法。

```
    public E remove(int index) {
        checkElementIndex(index);
        return unlink(node(index));
    }
```

isElementIndex(int index)

判断参数index是否是元素的索引

这个方法也是辅助方法，它用来判断index是否在LinkedList索引范围内，所以index

```
    private boolean isElementIndex(int index) {
        return index >= 0 && index < size;
    }
```

isPositionIndex(int index)

LinkedList的这个方法用于判断当我们新添加元素的时候，传进来的index是否合法，而且我们新添加的元素可能在LinkedList最后一个元素的后面，所以这里允许index<=size。

```
    private boolean isPositionIndex(int index) {
        return index >= 0 && index <= size;
    }
```

outOfBoundsMsg(int index)

返回越界的信息：也是一个辅助方法。

```
  private String outOfBoundsMsg(int index) {
        return "Index: "+index+", Size: "+size;
    }
```

checkElementIndex(int index)

判断参数index是否是元素的索引（如果不是则抛出异常）

```
private void checkElementIndex(int index) {
        if (!isElementIndex(index))
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
```

checkPositionIndex(int index)

LinkedList的这个方法用于判断当我们新添加元素的时候，传进来的index是否合法，（调用的是isPositionIndex(index)方法）而且我们新添加的元素可能在LinkedList最后一个元素的后面，所以这里允许index<=size。如果不合法，则抛出异常。

```
    private void checkPositionIndex(int index) {
        if (!isPositionIndex(index))
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
```

lastIndexOf(Object o)

在LinkedList中查找object在LinkedList中的位置。（从后向前遍历，只返回第一出线的元素的索引，如果没找到，则返回-1）

```
  public int lastIndexOf(Object o) {
        int index = size;
        if (o == null) {
            for (Node<E> x = last; x != null; x = x.prev) {
                index--;
                if (x.item == null)
                    return index;
            }
        } else {
            for (Node<E> x = last; x != null; x = x.prev) {
```

```
                index--;
                if (o.equals(x.item))
                    return index;
            }
        }
        return -1;
    }
```