[java集合框架之LinkedList 深度解析（二）](http://www.spring4all.com/article/1071)

**摘要：**

LinkedList 和 ArrayList 一样，都实现了 List 接口，但其内部的数据结构有本质的不同。LinkedList 是基于链表实现的（通过名字也能区分开来），所以它的插入和删除操作比 ArrayList 更加高效。但也是由于其为基于链表的，所以随机访问的效率要比 ArrayList 差。

**方法剖析**

**add(E e)**

*/\*\**

*\* Appends the specified element to the end of this list.*

*\**

*\* &lt;p>This method is equivalent to {@link #addLast}.*

*\**

*\* @param e element to be appended to this list*

*\* @return {@code true} (as specified by {@link Collection#add})*

*\*/*

public boolean add(E e) {

linkLast(e);

return true;

}

该方法在LinkedList的末尾插入元素，因为有last指向链表末尾，在末尾插入元素的花费是常数时间。只需要简单修改几个相关引用即可；  
内部其调用了自己的方法linkLast(E e)。

*/\*\**

*\* Links e as last element.*

*\*/*

void linkLast(E e) {

final Node&lt;E> l = last;

final Node&lt;E> newNode = new Node&lt;>(l, e, null);

last = newNode;

if (l == null)

*//如果是null说明这个链表为空，加入的节点指向头节点，同时也指向尾节点*

first = newNode;

else

l.next = newNode;

size++;

modCount++;

}

该方法首先将 last 的 Node 引用指向了一个新的 Node(l)，然后根据l新建了一个 newNode，其中的元素就为要添加的 e；而后，我们让 last 指向了 newNode。接下来是自身进行维护该链表。

**add(int index, E element)**

*/\*\**

*\* Inserts the specified element at the specified position in this list.*

*\* Shifts the element currently at that position (if any) and any*

*\* subsequent elements to the right (adds one to their indices).*

*\**

*\* @param index index at which the specified element is to be inserted*

*\* @param element element to be inserted*

*\* @throws IndexOutOfBoundsException {@inheritDoc}*

*\*/*

public void add(int index, E element) {

checkPositionIndex(index);*//检查index是否越界*

if (index == size)

linkLast(element);*//在链表尾节点添加元素*

else

linkBefore(element, node(index));*//利用二分法查找添加位置*

}

该方法是在指定 index 位置插入元素。如果 index 位置正好等于 size，则调用 linkLast(element) 将其插入末尾；否则调用 linkBefore(element, node(index))方法进行插入。

private void checkPositionIndex(int index) {

if (!isPositionIndex(index))

throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

}

*/\*\**

*\* Tells if the argument is the index of a valid position for an*

*\* iterator or an add operation.*

*\*/*

private boolean isPositionIndex(int index) {

return index >= 0 && index &lt;= size;

}

优化查找链表的指定位置，利用二分法进行查找，方便更快的找到要添加元素的位置

*/\*\**

*\* Returns the (non-null) Node at the specified element index.*

*\*/*

Node&lt;E> node(int index) {

*// assert isElementIndex(index);*

*//size>>1等于size/2，就是链表的一半长度*

if (index < (size >> 1)) {

Node&lt;E> x = first;

for (int i = 0; i < index; i++)

x = x.next;

return x;

} else {

Node&lt;E> x = last;

for (int i = size - 1; i > index; i--)

x = x.prev;

return x;

}

}

上面代码中的node(int index)函数有一点小小的trick，因为链表双向的，可以从开始往后找，也可以从结尾往前找，具体朝那个方向找取决于条件index < (size >> 1)，也即是index是靠近前端还是后端。

*/\*\**

*\* Inserts element e before non-null Node succ.*

*\*/*

void linkBefore(E e, Node&lt;E> succ) {

*// assert succ != null;*

final Node&lt;E> pred = succ.prev;

final Node&lt;E> newNode = new Node<>(pred, e, succ);

succ.prev = newNode;

if (pred == null)

first = newNode;

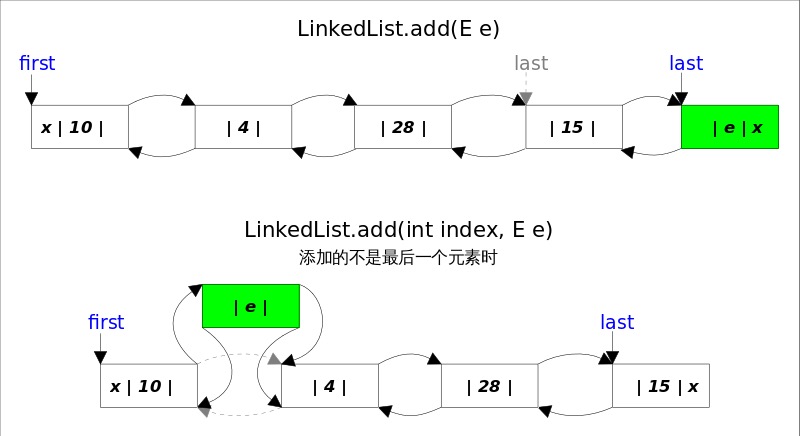
else

pred.next = newNode;

size++;

modCount++;

}



**remove(Object o)**

*/\*\**

*\* Removes the first occurrence of the specified element from this list,*

*\* if it is present. If this list does not contain the element, it is*

*\* unchanged. More formally, removes the element with the lowest index*

*\* {@code i} such that*

*\* &lt;tt>(o==null&nbsp;?&nbsp;get(i)==null&nbsp;:&nbsp;o.equals(get(i)))&lt;/tt>*

*\* (if such an element exists). Returns {@code true} if this list*

*\* contained the specified element (or equivalently, if this list*

*\* changed as a result of the call).*

*\**

*\* @param o element to be removed from this list, if present*

*\* @return {@code true} if this list contained the specified element*

*\*/*

public boolean remove(Object o) {

if (o == null) {

for (Node&lt;E> x = first; x != null; x = x.next) {

if (x.item == null) {

unlink(x);*//删除节点操作*

return true;

}

}

} else {

for (Node&lt;E> x = first; x != null; x = x.next) {

if (o.equals(x.item)) {

unlink(x);

return true;

}

}

}

return false;

}

以下代码是删除节点的源码，由于逻辑有点儿混淆，不好解释，请大家仔细思考，其实很简单的。

*/\*\**

*\* Unlinks non-null node x.*

*\*/*

E unlink(Node&lt;E> x) {

*// assert x != null;*

final E element = x.item;

final Node&lt;E> next = x.next;

final Node&lt;E> prev = x.prev;

if (prev == null) {*//删除的是第一个元素*

first = next;

} else {

prev.next = next;

x.prev = null;

}

if (next == null) {*//删除的是最后一个元素*

last = prev;

} else {

next.prev = prev;

x.next = null;

}

x.item = null;*//通知垃圾回收器回收*

size--;

modCount++;

return element;

}

**E remove(int index)**

*/\*\**

*\* Removes the element at the specified position in this list. Shifts any*

*\* subsequent elements to the left (subtracts one from their indices).*

*\* Returns the element that was removed from the list.*

*\**

*\* @param index the index of the element to be removed*

*\* @return the element previously at the specified position*

*\* @throws IndexOutOfBoundsException {@inheritDoc}*

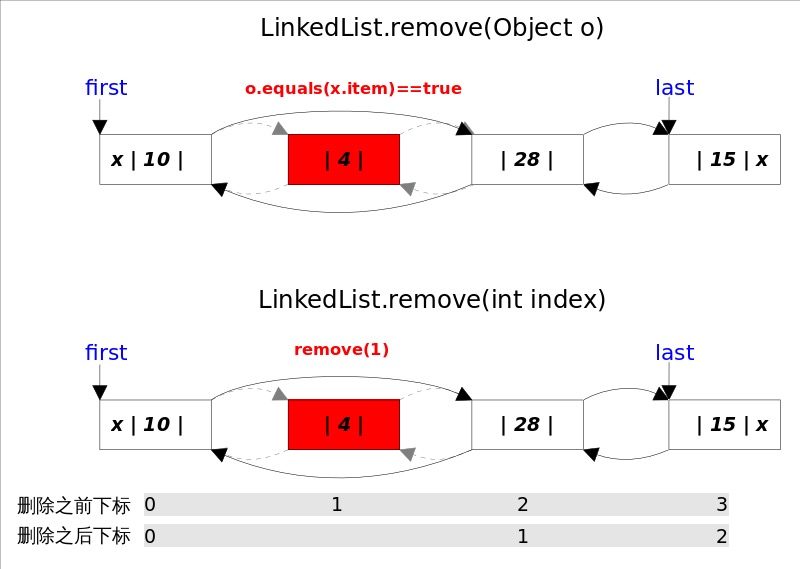
*\*/*

public E remove(int index) {

checkElementIndex(index);*//检查是否越界*

return unlink(node(index));通过索引找到节点，最后删除节点

}

两个删除操作都要1.先找到要删除元素的引用，2.修改相关引用，完成删除操作。在寻找被删元素引用的时候remove(Object o)调用的是元素的equals方法，而remove(int index)使用的是下标计数，两种方式都是线性时间复杂度。在步骤2中，两个revome()方法都是通过unlink(Node<E> x)方法完成的。这里需要考虑删除元素是第一个或者最后一个时的边界情况。  


**总结：**

该篇文章主要讲解了LinkedList的添加元素和删除元素的方法，通过源码分析了底层是如何实现的，同时也借助了部分图表的形式，或许有些部分没有解释得很清楚，希望大家能够理解，而且这个部分有一些是很抽象的，需要大家自己好好想明白才行，后续文章会持续讲解LinkedList的其余部分方法。

[java集合框架之LinkedList 深度解析（一）](http://www.spring4all.com/article/1067)  
[java集合框架之LinkedList 深度解析（二）](http://www.spring4all.com/article/1071)