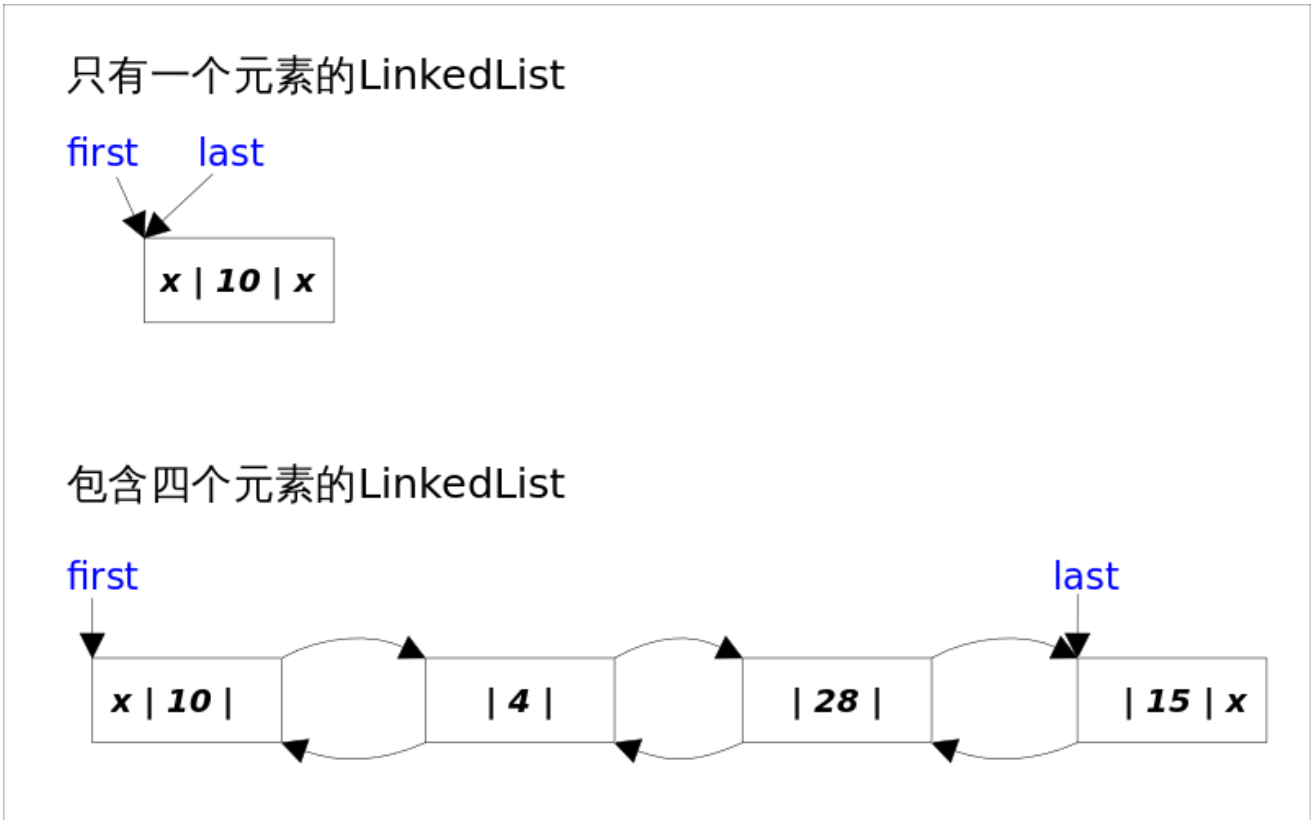
Collection - LinkedList源码解析

概述

*LinkedList*同时实现了*List*接口和*Deque*接口，也就是说它既可以看作一个顺序容器，又可以看作一个队列(*Queue*)，同时又可以看作一个栈(*Stack*)。这样看来，*LinkedList*简直就是个全能冠军。当你需要使用栈或者队列时，可以考虑使用*LinkedList*，一方面是因为Java官方已经声明不建议使用*Stack*类，更遗憾的是，Java里根本没有一个叫做*Queue*的类(它是个接口名字)。关于栈或队列，现在的首选是*ArrayDeque*，它有着比*LinkedList*(当作栈或队列使用时)有着更好的性能。



*LinkedList*的实现方式决定了所有跟下标相关的操作都是线性时间，而在首段或者末尾删除元素只需要常数时间。为追求效率*LinkedList*没有实现同步(synchronized)，如果需要多个线程并发访问，可以先采用`Collections.synchronizedList()`方法对其进行包装。

LinkedLists实现

底层数据结构

*LinkedList*底层**通过双向链表实现**，本节将着重讲解插入和删除元素时双向链表的维护过程，也即是之间解跟*List*接口相关的函数，而将*Queue*和*Stack*以及*Deque*相关的知识放在下一节讲。双向链表的每个节点用内部类*Node*表示。*LinkedList*通过`first`和`last`引用分别指向链表的第一个和最后一个元素。注意这里没有所谓的哑元，当链表为空的时候`first`和`last`都指向`null`。

```
    transient int size = 0;

    /**
     * Pointer to first node.
     * Invariant: (first == null && last == null) ||
     *            (first.prev == null && first.item != null)
     */
    transient Node<E> first;

    /**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     *            (last.next == null && last.item != null)
     */
    transient Node<E> last;
```

其中Node是私有的内部类:

```
    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;
        }
    }
```

构造函数

```
    /**
     * Constructs an empty list.
     */
    public LinkedList() {
    }

    /**
     * Constructs a list containing the elements of the specified
     * collection, in the order they are returned by the collection's
     * iterator.
     *
     * @param  c the collection whose elements are to be placed into this list
     * @throws NullPointerException if the specified collection is null
     */
    public LinkedList(Collection<? extends E> c) {
        this();
```

```
        addAll(c);
    }
```

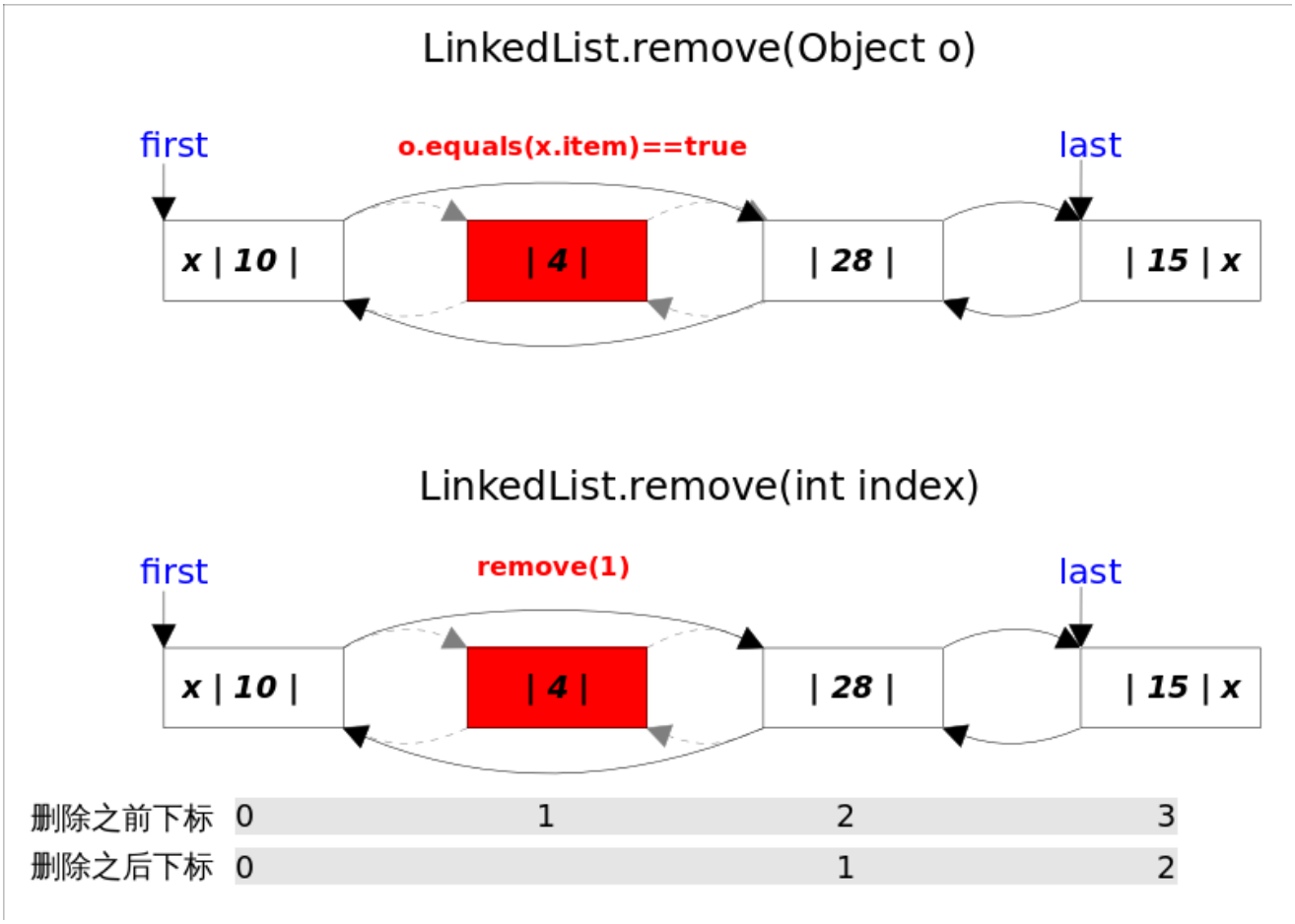
getFirst(), getLast()

获取第一个元素， 和获取最后一个元素:

```
    /**
     * Returns the first element in this list.
     *
     * @return the first element in this list
     * @throws NoSuchElementException if this list is empty
     */
    public E getFirst() {
        final Node<E> f = first;
        if (f == null)
            throw new NoSuchElementException();
        return f.item;
    }

    /**
     * Returns the last element in this list.
     *
     * @return the last element in this list
     * @throws NoSuchElementException if this list is empty
     */
    public E getLast() {
        final Node<E> l = last;
        if (l == null)
            throw new NoSuchElementException();
        return l.item;
    }
```

removeFirest(), removeLast(), remove(e), remove(index)

`remove()`方法也有两个版本，一个是删除跟指定元素相等的第一个元素`remove(Object o)`，另一个是删除指定下标处的元素`remove(int index)`。



删除元素 - 指的是删除第一次出现的这个元素, 如果没有这个元素，则返回false；判读的依据是equals方法， 如果equals，则直接unlink这个node；由于LinkedList可存放null元素，故也可以删除第一次出现null的元素；

```
    /**
     * 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
     * <tt>(o==null ? get(i)==null : o.equals(get(i)))</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<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;
    }

    /**
     * Unlinks non-null node x.
     */
    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) {// 第一个元素
            first = next;
        } else {
            prev.next = next;
            x.prev = null;
        }

        if (next == null) {// 最后一个元素
            last = prev;
        } else {
            next.prev = prev;
            x.next = null;
        }

        x.item = null; // GC
        size--;
        modCount++;
        return element;
    }
```

`remove(int index)`使用的是下标计数， 只需要判断该index是否有元素即可，如果有则直接unlink这个node。

```
    /**
     * 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);
        return unlink(node(index));
    }
```

删除head元素:

```
    /**
     * Removes and returns the first element from this list.
     *
     * @return the first element from this list
     * @throws NoSuchElementException if this list is empty
     */
```

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


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

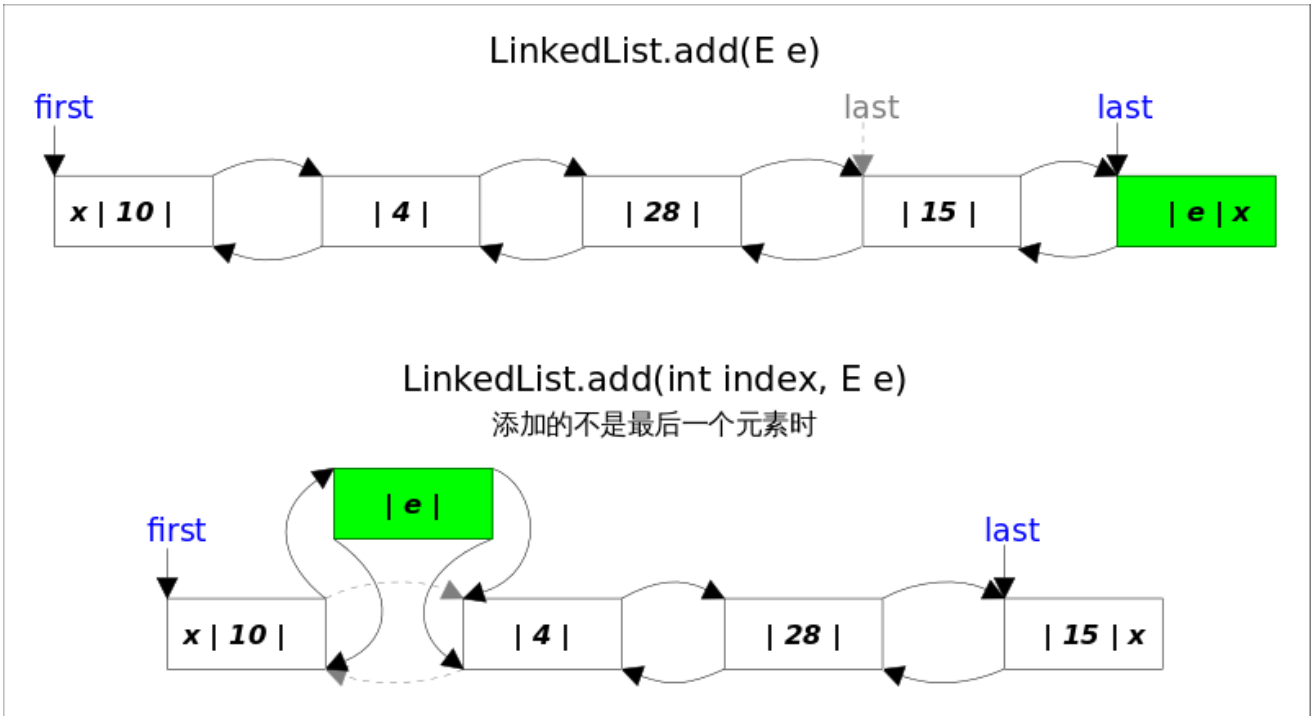
删除last元素:

```
/**
 * Removes and returns the last element from this list.
 *
 * @return the last element from this list
 * @throws NoSuchElementException if this list is empty
 */
public E removeLast() {
    final Node<E> l = last;
    if (l == null)
        throw new NoSuchElementException();
    return unlinkLast(l);
}

/**
 * Unlinks non-null last node l.
 */
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;
}
```

add()

*add()*方法有两个版本，一个是*add(E e)*，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；另一个是add(int index, E element)，该方法是在指定下表处插入元素，需要先通过线性查找找到具体位置，然后修改相关引用完成插入操作。

```
/**
 * Appends the specified element to the end of this list.
 *
 * <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;
}

/**
 * Links e as last element.
 */
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++;
}
```



add(int index, E element), 当index==size时，等同于add(E e); 如果不是，则分两步: 1.先根据index找到要插入的位置,即node(index)方法；2.修改引用，完成插入操作。

```
    /**
     * 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);

        if (index == size)
            linkLast(element);
        else
            linkBefore(element, node(index));
    }
```

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。从这里也可以看出，linkedList通过index检索元素的效率没有arrayList高。

```
    /**
     * Returns the (non-null) Node at the specified element index.
     */
    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;
        }
    }
```

addAll()

addAll(index, c) 实现方式并不是直接调用add(index,e)来实现，主要是因为效率的问题，另一个是fail-fast中modCount只会增加1次；

```
    /**
     * Appends all of the elements in the specified collection to the end of
     * this list, in the order that they are returned by the specified
     * collection's iterator.  The behavior of this operation is undefined if
     * the specified collection is modified while the operation is in
     * progress.  (Note that this will occur if the specified collection is
```

```
     * this list, and it's nonempty.)
     *
     * @param c collection containing elements to be added to this list
     * @return {@code true} if this list changed as a result of the call
     * @throws NullPointerException if the specified collection is null
     */
    public boolean addAll(Collection<? extends E> c) {
        return addAll(size, c);
    }

    /**
     * Inserts all of the elements in the specified collection into this
     * list, starting at the specified position.  Shifts the element
     * currently at that position (if any) and any subsequent elements to
     * the right (increases their indices).  The new elements will appear
     * in the list in the order that they are returned by the
     * specified collection's iterator.
     *
     * @param index index at which to insert the first element
     *              from the specified collection
     * @param c collection containing elements to be added to this list
     * @return {@code true} if this list changed as a result of the call
     * @throws IndexOutOfBoundsException {@inheritDoc}
     * @throws NullPointerException if the specified collection is null
     */
    public boolean addAll(int index, Collection<? extends E> c) {
        checkPositionIndex(index);

        Object[] a = c.toArray();
        int numNew = a.length;
        if (numNew == 0)
            return false;

        Node<E> pred, succ;
        if (index == size) {
            succ = null;
            pred = last;
        } else {
            succ = node(index);
            pred = succ.prev;
        }

        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;
        }

        if (succ == null) {
            last = pred;
        } else {
            pred.next = succ;
            succ.prev = pred;
        }

        size += numNew;
        modCount++;
```

```
        return true;
    }
```

clear()

为了让GC更快可以回收放置的元素，需要将node之间的引用关系赋空。

```
    /**
     * Removes all of the elements from this list.
     * The list will be empty after this call returns.
     */
    public void clear() {
        // Clearing all of the links between nodes is "unnecessary", but:
        // - helps a generational GC if the discarded nodes inhabit
        //   more than one generation
        // - is sure to free memory even if there is a reachable Iterator
        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++;
    }
```

Positional Access 方法

通过index获取元素

```
    /**
     * Returns the element at the specified position in this list.
     *
     * @param index index of the element to return
     * @return the element at the specified position in this list
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public E get(int index) {
        checkElementIndex(index);
        return node(index).item;
    }
```

将某个位置的元素重新赋值:

```
    /**
     * Replaces the element at the specified position in this list with the
     * specified element.
     *
     * @param index index of the element to replace
     * @param element element to be stored at the specified position
```

```
     * @return the element previously at the specified position
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public E set(int index, E element) {
        checkElementIndex(index);
        Node<E> x = node(index);
        E oldVal = x.item;
        x.item = element;
        return oldVal;
    }
```

将元素插入到指定index位置:

```
    /**
     * 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);

        if (index == size)
            linkLast(element);
        else
            linkBefore(element, node(index));
    }
```

删除指定位置的元素:

```
    /**
     * 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);
        return unlink(node(index));
    }
```

其它位置的方法:

```
    /**
     * Tells if the argument is the index of an existing element.
     */
    private boolean isElementIndex(int index) {
        return index >= 0 && index < size;
    }

    /**
     * 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 <= size;
    }

    /**
     * Constructs an IndexOutOfBoundsException detail message.
     * Of the many possible refactorings of the error handling code,
     * this "outlining" performs best with both server and client VMs.
     */
    private String outOfBoundsMsg(int index) {
        return "Index: "+index+", Size: "+size;
    }

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

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

查找操作

查找操作的本质是查找元素的下标:

查找第一次出现的index, 如果找不到返回-1;

```
    /**
     * Returns the index of the first occurrence of the specified element
     * in this list, or -1 if this list does not contain the element.
     * More formally, returns the lowest index {@code i} such that
     * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
     * or -1 if there is no such index.
     *
     * @param o element to search for
     * @return the index of the first occurrence of the specified element in
     *         this list, or -1 if this list does not contain the element
     */
    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;
    }
```

查找最后一次出现的index, 如果找不到返回-1；

```
    /**
     * Returns the index of the last occurrence of the specified element
     * in this list, or -1 if this list does not contain the element.
     * More formally, returns the highest index {@code i} such that
     * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
     * or -1 if there is no such index.
     *
     * @param o element to search for
     * @return the index of the last occurrence of the specified element in
     *         this list, or -1 if this list does not contain the element
     */
    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;
    }
```

Queue 方法

```
    /**
     * Retrieves, but does not remove, the head (first element) of this list.
     *
     * @return the head of this list, or {@code null} if this list is empty
     * @since 1.5
     */
    public E peek() {
        final Node<E> f = first;
        return (f == null) ? null : f.item;
    }

    /**
     * Retrieves, but does not remove, the head (first element) of this list.
     *
     * @return the head of this list
     * @throws NoSuchElementException if this list is empty
     * @since 1.5
     */
    public E element() {
        return getFirst();
```

```
    }

    /**
     * Retrieves and removes the head (first element) of this list.
     *
     * @return the head of this list, or {@code null} if this list is empty
     * @since 1.5
     */
    public E poll() {
        final Node<E> f = first;
        return (f == null) ? null : unlinkFirst(f);
    }

    /**
     * Retrieves and removes the head (first element) of this list.
     *
     * @return the head of this list
     * @throws NoSuchElementException if this list is empty
     * @since 1.5
     */
    public E remove() {
        return removeFirst();
    }

    /**
     * Adds the specified element as the tail (last element) of this list.
     *
     * @param e the element to add
     * @return {@code true} (as specified by {@link Queue#offer})
     * @since 1.5
     */
    public boolean offer(E e) {
        return add(e);
    }
```

Deque 方法

```
    /**
     * Inserts the specified element at the front of this list.
     *
     * @param e the element to insert
     * @return {@code true} (as specified by {@link Deque#offerFirst})
     * @since 1.6
     */
    public boolean offerFirst(E e) {
        addFirst(e);
        return true;
    }

    /**
     * Inserts the specified element at the end of this list.
     *
     * @param e the element to insert
     * @return {@code true} (as specified by {@link Deque#offerLast})
     * @since 1.6
     */
    public boolean offerLast(E e) {
```

```
    addLast(e);
    return true;
}

/**
 * Retrieves, but does not remove, the first element of this list,
 * or returns {@code null} if this list is empty.
 *
 * @return the first element of this list, or {@code null}
 *         if this list is empty
 * @since 1.6
 */
public E peekFirst() {
    final Node<E> f = first;
    return (f == null) ? null : f.item;
 }

/**
 * Retrieves, but does not remove, the last element of this list,
 * or returns {@code null} if this list is empty.
 *
 * @return the last element of this list, or {@code null}
 *         if this list is empty
 * @since 1.6
 */
public E peekLast() {
    final Node<E> l = last;
    return (l == null) ? null : l.item;
}

/**
 * Retrieves and removes the first element of this list,
 * or returns {@code null} if this list is empty.
 *
 * @return the first element of this list, or {@code null} if
 *     this list is empty
 * @since 1.6
 */
public E pollFirst() {
    final Node<E> f = first;
    return (f == null) ? null : unlinkFirst(f);
}

/**
 * Retrieves and removes the last element of this list,
 * or returns {@code null} if this list is empty.
 *
 * @return the last element of this list, or {@code null} if
 *     this list is empty
 * @since 1.6
 */
public E pollLast() {
    final Node<E> l = last;
    return (l == null) ? null : unlinkLast(l);
}

/**
 * Pushes an element onto the stack represented by this list.  In other
 * words, inserts the element at the front of this list.
 *
 * <p>This method is equivalent to {@link #addFirst}.
```

```
     *
     * @param e the element to push
     * @since 1.6
     */
    public void push(E e) {
        addFirst(e);
    }

    /**
     * Pops an element from the stack represented by this list.  In other
     * words, removes and returns the first element of this list.
     *
     * <p>This method is equivalent to {@link #removeFirst()}.
     *
     * @return the element at the front of this list (which is the top
     *         of the stack represented by this list)
     * @throws NoSuchElementException if this list is empty
     * @since 1.6
     */
    public E pop() {
        return removeFirst();
    }

    /**
     * Removes the first occurrence of the specified element in this
     * list (when traversing the list from head to tail).  If the list
     * does not contain the element, it is unchanged.
     *
     * @param o element to be removed from this list, if present
     * @return {@code true} if the list contained the specified element
     * @since 1.6
     */
    public boolean removeFirstOccurrence(Object o) {
        return remove(o);
    }

    /**
     * Removes the last occurrence of the specified element in this
     * list (when traversing the list from head to tail).  If the list
     * does not contain the element, it is unchanged.
     *
     * @param o element to be removed from this list, if present
     * @return {@code true} if the list contained the specified element
     * @since 1.6
     */
    public boolean removeLastOccurrence(Object o) {
        if (o == null) {
            for (Node<E> x = last; x != null; x = x.prev) {
                if (x.item == null) {
                    unlink(x);
                    return true;
                }
            }
        } else {
            for (Node<E> x = last; x != null; x = x.prev) {
                if (o.equals(x.item)) {
                    unlink(x);
                    return true;
                }
            }
        }
```

```
        return false;
    }
```