问题

(1) LinkedList只是一个List吗? //双向链表、队列、栈

(2) LinkedList还有其它什么特性吗? //无界队列

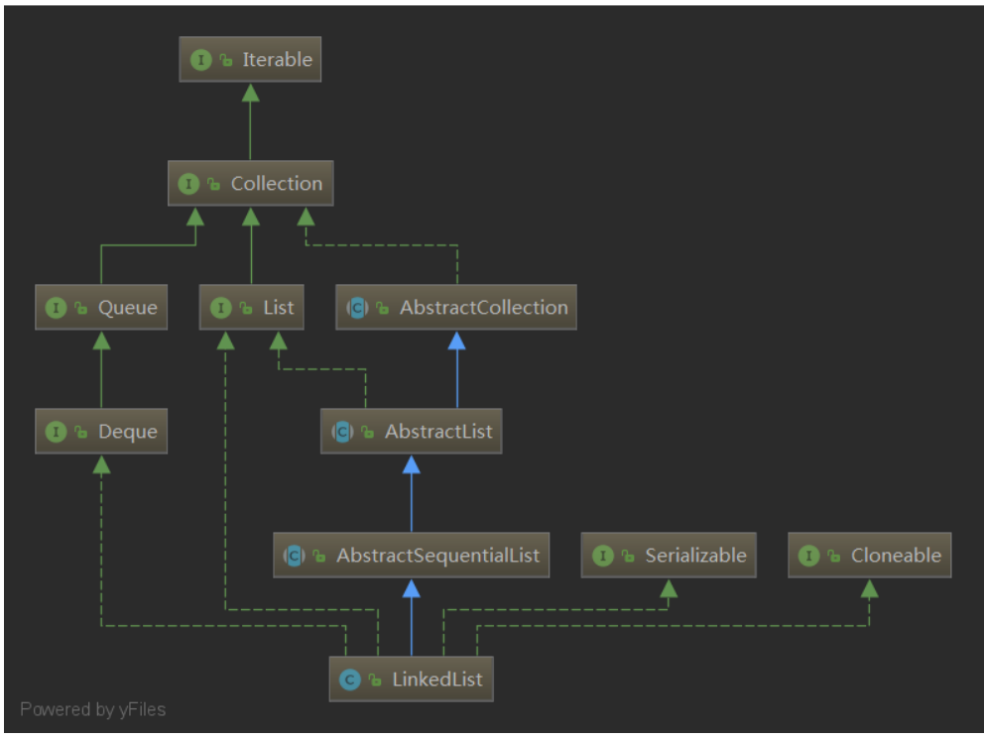
(3) LinkedList为啥经常拿出来跟ArrayList比较?

(4) 我为什么把LinkedList放在最后一章来讲?

简介

LinkedList是一个以双向链表实现的List，它除了作为List使用，还可以作为队列或者栈来使用，它是怎么实现的呢？让我们一起来学习吧。

继承体系



通过继承体系，我们可以看到LinkedList不仅实现了List接口，还实现了Queue和Deque接口，所以它既能作为List使用，也能作为双端队列使用，当然也可以作为栈使用。

源码分析

主要属性

```
1.    // 元素个数
2.    transient int size = 0;
3.    // 链表首节点
4.    transient Node<E> first;
5.    // 链表尾节点
6.    transient Node<E> last;
7.
```

属性很简单，定义了元素个数size和链表的首尾节点。

主要内部类

典型的双链表结构。

```
    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;
        }
    }

```

主要构造方法 //内部是链表存储，所以也就没有ArrayList的扩容、初始化大小了

```
    public LinkedList() {
    }

    public LinkedList(Collection<? extends E> c) {
        this();
        addAll(c);
    }

```

两个构造方法也很简单，可以看出是一个无界的队列。

添加元素 linkFirst、linkLast、addFirst、addLast、offerFirst、offerLast

作为一个双端队列，添加元素主要有两种，一种是在队列尾部添加元素，一种是在队列首部添加元素，这两种形式在LinkedList中主要是通过下面两个方法来实现的。//first和last分别指向首节点和尾结点，假如没有元素，则两个都是null，假如只有一个元素，则都指向那个元素

```
    // 从队列首添加元素
    private void linkFirst(E e) {
        // 首节点
        final Node<E> f = first;
        // 创建新节点，新节点的next是首节点
        final Node<E> newNode = new Node<>(null, e, f);
        // 让新节点作为新的首节点
        first = newNode;
        // 判断是不是第一个添加的元素
        // 如果是就把last也置为新节点
        // 否则把原首节点的prev指针置为新节点
        if (f == null)
            last = newNode;
        else
            f.prev = newNode; //因为是双向链表
        // 元素个数加1
        size++;
        // 修改次数加1，说明这是一个支持fail-fast的集合
        modCount++;
    }

    // 从队列尾添加元素
    void linkLast(E e) {
        // 队列尾节点
        final Node<E> l = last;
        // 创建新节点，新节点的prev是尾节点
```

```
        final Node<E> newNode = new Node<>(l, e, null);
        // 让新节点成为新的尾节点
        last = newNode;
        // 判断是不是第一个添加的元素
        // 如果是就把first也置为新节点
        // 否则把原尾节点的next指针置为新节点
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        // 元素个数加1
        size++;
        // 修改次数加1
        modCount++;
    }

    public void addFirst(E e) {
        linkFirst(e);
    }

    public void addLast(E e) {
        linkLast(e);
    }

    // 作为无界队列，添加元素总是会成功的
    public boolean offerFirst(E e) {
        addFirst(e);
        return true;
    }

    public boolean offerLast(E e) {
        addLast(e);
        return true;
    }

```

典型的双链表在首尾添加元素的方法，代码比较简单，这里不作详细描述了。

上面是作为双端队列来看，它的添加元素分为首尾添加元素，那么，作为List呢?

作为List，是要支持在中间添加元素的，主要是通过下面这个方法实现的。

```
    // 在节点succ之前添加元素
    void linkBefore(E e, Node<E> succ) {
        // succ是待添加节点的后继节点
        // 找到待添加节点的前置节点
        final Node<E> pred = succ.prev;
        // 在其前置节点和后继节点之间创建一个新节点
        final Node<E> newNode = new Node<>(pred, e, succ);
        // 修改后继节点的前置指针指向新节点
        succ.prev = newNode;
        // 判断前置节点是否为空
        // 如果为空，说明是第一个添加的元素，修改first指针
        // 否则修改前置节点的next为新节点
        if (pred == null)
            first = newNode;
        else
            pred.next = newNode;
        // 修改元素个数
        size++;
        // 修改次数加1
        modCount++;
    }

```

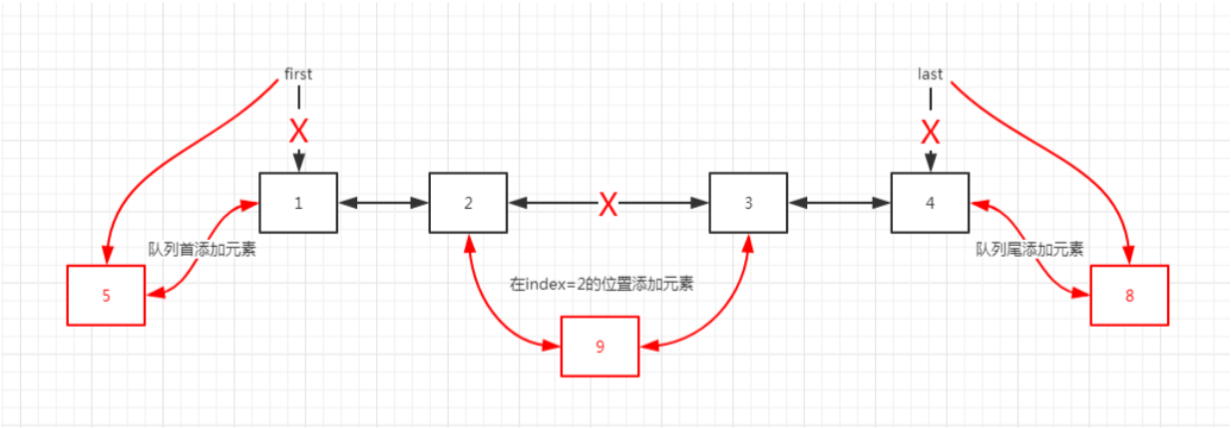
```
    // 寻找index位置的节点
    Node<E> node(int index) {
        // 因为是双链表
        // 所以根据index是在前半段还是后半段决定从前遍历还是从后遍历
        // 这样index在后半段的时候可以少遍历一半的元素
        if (index < (size >> 1)) {
            // 如果是在前半段
            // 就从前遍历
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            // 如果是在后半段
            // 就从后遍历
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }

    // 在指定index位置处添加元素
    public void add(int index, E element) {
        // 判断是否越界
        checkPositionIndex(index);
        // 如果index是在队列尾节点之后的一个位置
        // 把新节点直接添加到尾节点之后
        // 否则调用linkBefore()方法在中间添加节点
        if (index == size)
            linkLast(element);
        else
            linkBefore(element, node(index));
    }
```

优化，提高速度

在中间添加元素的方法也很简单，典型的双链表在中间添加元素的方法。

添加元素的三种方式大致如下图所示：



在队列首尾添加元素很高效，时间复杂度为O(1)。

在中间添加元素比较低效，首先要先找到插入位置的节点，再修改前后节点的指针，时间复杂度为O(n)。

删除元素

作为双端队列，删除元素也有两种方式，一种是队列首删除元素，一种是队列尾删除元素。

作为List，又要支持中间删除元素，所以删除元素一个有三个方法，分别如下。

```
    // 删除首节点
    private E unlinkFirst(Node<E> f) {
        // 首节点的元素值
        final E element = f.item;
        // 首节点的next指针
        final Node<E> next = f.next;
        // 添加首节点的内容，协助GC
        f.item = null;
        f.next = null; // help GC
        // 把首节点的next作为新的首节点
        first = next;
        // 如果只有一个元素，删除了，把last也置为空
        // 否则把next的前置指针置为空
        if (next == null)
            last = null;
        else
            next.prev = null;
        // 元素个数减1
        size--;
        // 修改次数加1
        modCount++;
        // 返回删除的元素
        return element;
    }
    // 删除尾节点
    private E unlinkLast(Node<E> l) {
        // 尾节点的元素值
        final E element = l.item;
        // 尾节点的前置指针
        final Node<E> prev = l.prev;
        // 清空尾节点的内容，协助GC
        l.item = null;
        l.prev = null; // help GC
        // 让前置节点成为新的尾节点
        last = prev;
        // 如果只有一个元素，删除了把first置为空
        // 否则把前置节点的next置为空
        if (prev == null)
            first = null;
        else
            prev.next = null;
        // 元素个数减1
        size--;
        // 修改次数加1
        modCount++;
        // 返回删除的元素
        return element;
    }
    // 删除指定节点x
    E unlink(Node<E> x) {
        // x的元素值
        final E element = x.item;
        // x的前置节点
        final Node<E> next = x.next;
        // x的后置节点
        final Node<E> prev = x.prev;

        // 如果前置节点为空
```

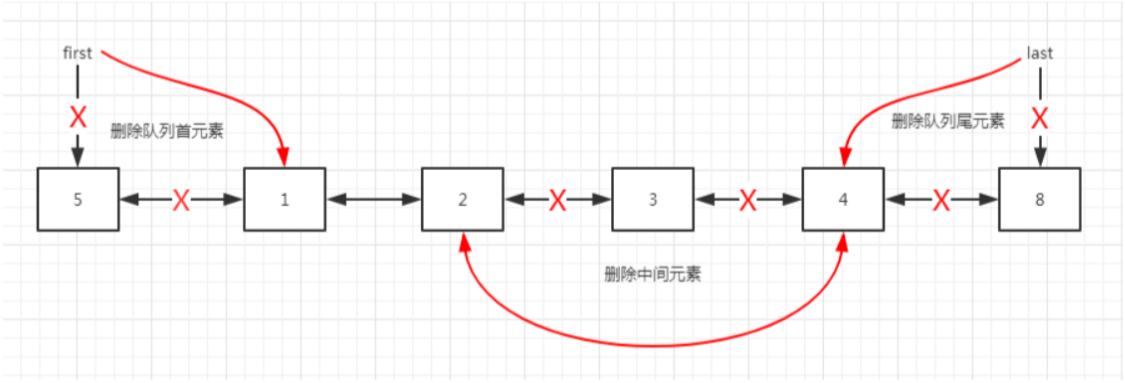
```
        // 说明是首节点，让first指向x的后置节点
        // 否则修改前置节点的next为x的后置节点
        if (prev == null) {
            first = next;
        } else {
            prev.next = next;
            x.prev = null;
        }

        // 如果后置节点为空
        // 说明是尾节点，让last指向x的前置节点
        // 否则修改后置节点的prev为x的前置节点
        if (next == null) {
            last = prev;
        } else {
            next.prev = prev;
            x.next = null;
        }

        // 清空x的元素值，协助GC
        x.item = null;
        // 元素个数减1
        size--;
        // 修改次数加1
        modCount++;
        // 返回删除的元素
        return element;
    }
    // remove的时候如果没有元素抛出异常
    public E removeFirst() {
        final Node<E> f = first;
        if (f == null)
            throw new NoSuchElementException();
        return unlinkFirst(f);
    }
    // remove的时候如果没有元素抛出异常
    public E removeLast() {
        final Node<E> l = last;
        if (l == null)
            throw new NoSuchElementException();
        return unlinkLast(l);
    }
    // poll的时候如果没有元素返回null
    public E pollFirst() {
        final Node<E> f = first;
        return (f == null) ? null : unlinkFirst(f);
    }
    // poll的时候如果没有元素返回null
    public E pollLast() {
        final Node<E> l = last;
        return (l == null) ? null : unlinkLast(l);
    }
    // 删除中间节点
    public E remove(int index) {
        // 检查是否越界
        checkElementIndex(index);
        // 删除指定index位置的节点
        return unlink(node(index));
    }

```

删除元素的三种方法都是典型的双链表删除元素的方法，大致流程如下图所示。



在队列首尾删除元素很高效，时间复杂度为O(1)。

在中间删除元素比较低效，首先要找到删除位置的节点，再修改前后指针，时间复杂度为O(n)。

栈

前面我们说了，LinkedList是双端队列，还记得双端队列可以作为栈使用吗?

```
1.    public void push(E e) {
2.        addFirst(e);
3.    }
4.
5.    public E pop() {
6.        return removeFirst();
7.    }
8.
```

栈的特性是LIFO(Last In First Out)，所以作为栈使用也很简单，添加删除元素都只操作队列首节点即可。

总结

protected transient int modCount = 0;
是AbstractList中的属性
fast-fail

（1）LinkedList是一个以双链表实现的List；

（2）LinkedList还是一个双端队列，具有队列、双端队列、栈的特性；

（3）LinkedList在队列首尾添加、删除元素非常高效，时间复杂度为O(1)；

（4）LinkedList在中间添加、删除元素比较低效，时间复杂度为O(n)；

（5）LinkedList不支持随机访问，所以访问非队列首尾的元素比较低效； //RandomAccess，随机访问，数组可以随机访问，链表得遍历

（6）LinkedList在功能上等于ArrayList + ArrayDeque；

彩蛋

java集合部分的源码分析全部完结，整个专题以ArrayList开头，以LinkedList结尾，我觉得非常合适，因为ArrayList代表了List的典型实现，LInkedList代表了Deque的典型实现，同时LinkedList也实现了List，通过这两个类一首一尾正好可以把整个集合贯穿起来。