死磕 java集合之LinkedList源码分析

问题

（1）LinkedList只是一个List吗？

（2）LinkedList还有其它什么特性吗？

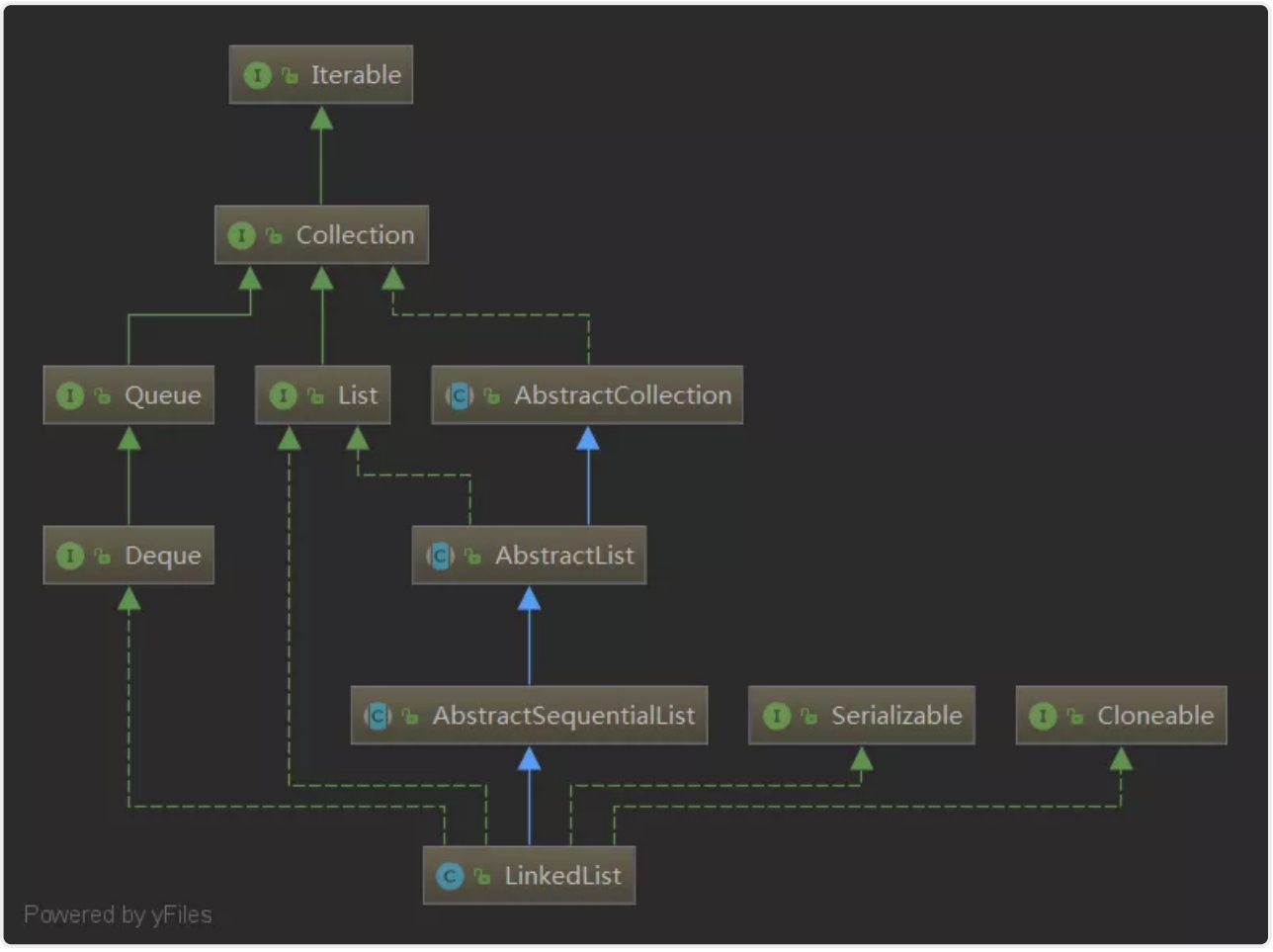
（3）LinkedList为啥经常拿出来跟ArrayList比较？

（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。 java中所有的链表都是双向的

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

LinkedList 实现了List 接口，能对它进行列表操作。
LinkedList 实现了Deque 接口，即能将LinkedList当作双端队列使用。
LinkedList 实现了Cloneable接口，能克隆。
LinkedList 实现了java.io.Serializable接口，这意味着LinkedList支持序列化，能通过序列化去传输。

主要属性

```
// 元素个数
transient int size = 0;
// 链表首节点
transient Node<E> first;
// 链表尾节点
transient Node<E> last;
```

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

```
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

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。

```
// 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;
    // 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？

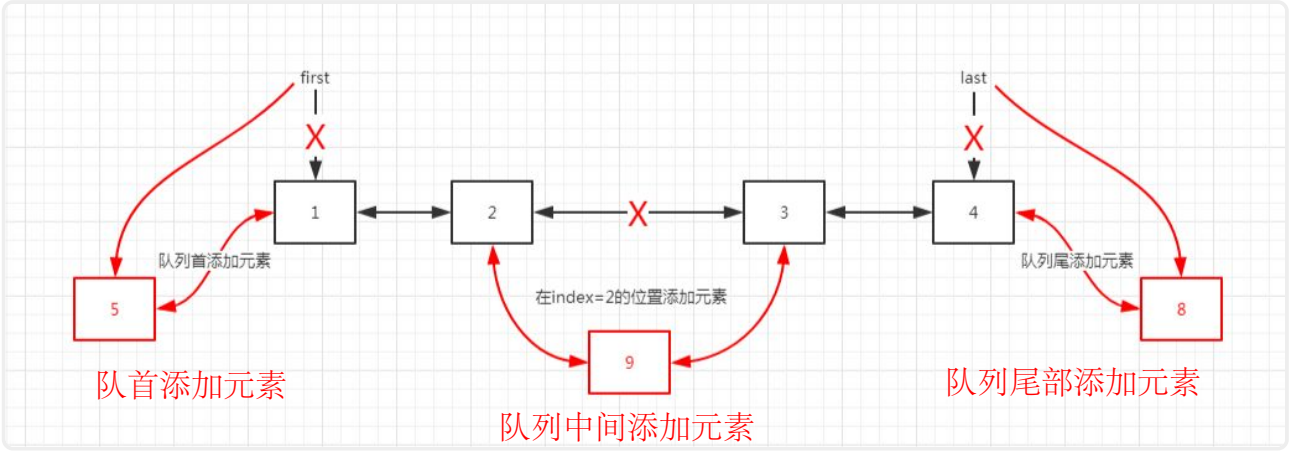
作为List，是要支持在中间添加元素的，主要是通过下面这个方法实现的。

```
// 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) { list的 list.get(index)调用
    // 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      提高遍历效率
    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));
}
```

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。

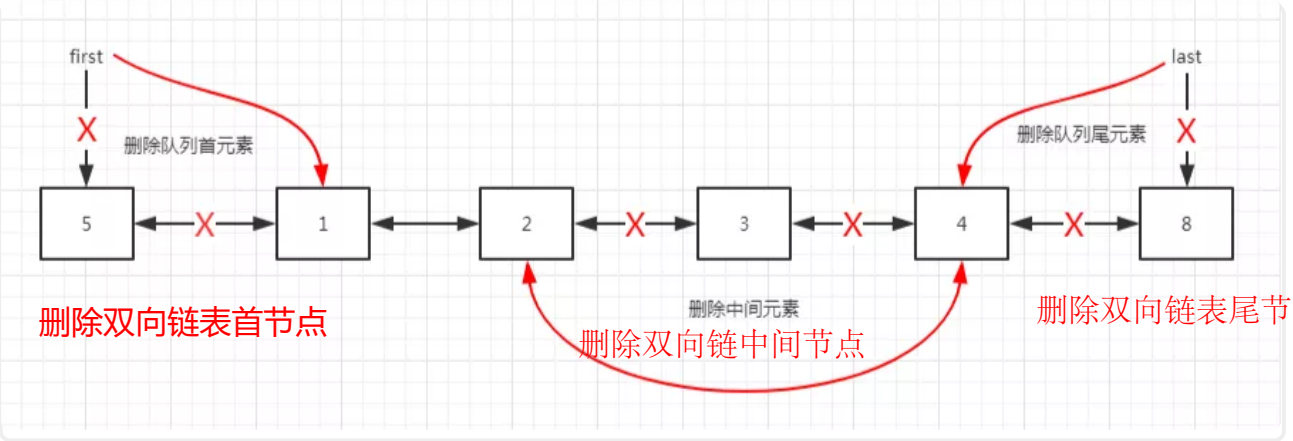
作为List，又要支持中间删除元素，所以删除元素一个有三个方法，分别如下。

```
// 删除首节点
private E unlinkFirst(Node<E> f) {  removeFirst()调用unlinkFirst
    // 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) { removeLast()调用unlinkLast
    // 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() { 检索并删除此列表的head first元素
    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() {检索并删除此列表的最后一个元素
    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?

```
public void push(E e) {
    addFirst(e);
}

public E pop() {
    return removeFirst();
}
```

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

（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；

（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。

还记得我们一共分析了哪些类吗？

下一章，笔者将对整个java集合做一个总结，并提出一些阅读源码过程中的问题，敬请期待^^

LinkedList常用API

常用方法

增

public boolean add(E e)，链表末尾添加元素，返回是否成功；
public void add(int index, E element)，向指定位置插入元素；
public boolean addAll(Collection<? extends E> c)，将一个集合的所有元素添加到链表后面，返回是否成功；
public boolean addAll(int index, Collection<? extends E> c)，将一个集合的所有元素添加到链表的指定位置后面，返回是否成功；
public void addFirst(E e)，添加到第一个元素；
public void addLast(E e)，添加到最后一个元素；
public boolean offer(E e)，向链表末尾添加元素，返回是否成功；
public boolean offerFirst(E e)，头部插入元素，返回是否成功；
public boolean offerLast(E e)，尾部插入元素，返回是否成功；

删

public void clear()，清空链表；
public E removeFirst()，删除并返回第一个元素；
public E removeLast()，删除并返回最后一个元素；
public boolean remove(Object o)，删除某一元素，返回是否成功；
public E remove(int index)，删除指定位置的元素；
public E poll()，删除并返回第一个元素；
public E remove()，删除并返回第一个元素

改

public E set(int index, E element)，设置指定位置的元素

查
public boolean contains(Object o)，判断是否含有某一元素；
public E get(int index)，返回指定位置的元素；
public E getFirst(), 返回第一个元素；
public E getLast()，返回最后一个元素；
public int indexOf(Object o)，查找指定元素从前往后第一次出现的索引；
public int lastIndexOf(Object o)，查找指定元素最后一次出现的索引；
public E peek()，返回第一个元素；
public E element()，返回第一个元素；
public E peekFirst()，返回头部元素；
public E peekLast()，返回尾部元素

其他
public Object clone()，克隆该列表；
public Iterator<E> descendingIterator()，返回倒序迭代器；
public int size()，返回链表元素个数；
public ListIterator<E> listIterator(int index)，返回从指定位置开始到末尾的迭代器；
public Object[] toArray()，返回一个由链表元素组成的数组；
public <T> T[] toArray(T[] a)，返回一个由链表元素转换类型而成的数组；

LinkedList的迭代器实现

```
LinkedList linkedList=new LinkedList();
    linkedList.add("a");
    linkedList.add("b");
    linkedList.add("d");
    linkedList.add("c");
    Iterator iterator = linkedList.iterator();
    while (iterator.hasNext()){
      System.out.println(iterator.next());
}
```

LinkedList的内部类ListItr(内部类实现Iterator迭代器接口)

```
private class ListItr implements ListIterator<E> {
    private Node<E> lastReturned;上一次next返回的节点
    private Node<E> next;下一次调用next()返回的Node
    private int nextIndex;下一次调用next()返回的索引
    private int expectedModCount = modCount;这里其实是调用this$0获得的modCount
```

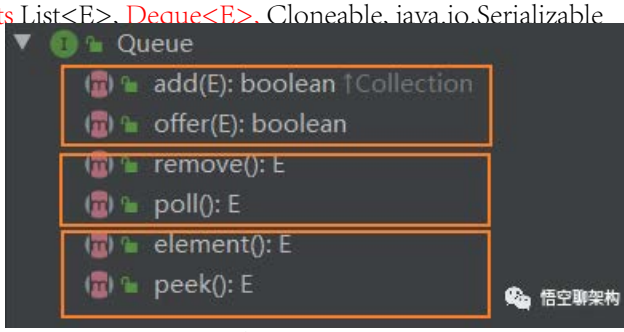
final synthetic LinkedList this$0=外部类对象LinkedList的引用（内部类对象持有外部类对象的引用）

```
    ListItr(int index) {
      // assert isPositionIndex(index);
      next = (index == size) ? null : node(index);
      nextIndex = index;
      this$0=外部类对象的引用/
    }

    public boolean hasNext() {
      return nextIndex < size;下一个返回的元素索引小于List的size
    }

    public E next() {
      checkForComodification();
      if (!hasNext())
        throw new NoSuchElementException();

      lastReturned = next;
      next = next.next;
      nextIndex++;
      return lastReturned.item;返回刚刚执行next返回的节点Node.item
    }
```

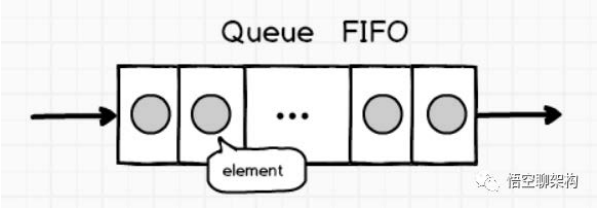
总结：LinkedList的迭代器是使用内部类实现的，内部类持有外部类LinkedList对象的引用，通过内部类的next指针和nextIndex索引完成双向链表的遍历

LinkedList实现的双端队列

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, java.io.Serializable



队列Queue接口:队列是一种FIFO的结构（一般禁止添加null元素）



add （入队尾）增加一个元素 如果队列已满，则抛出一个IIIegaISlabEepeplian异常
remove （出队头）移除并返回队列头部的元素 如果队列为空，则抛出一个NoSuchElementException异常
element 返回队列头部的元素，但是队头不出队列。 如果队列为空，则抛出一个NoSuchElementException异常
offer 添加一个元素并返回true 如果队列已满，则返回false
poll 移除并返问队列头部的元素 如果队列为空，则返回null
peek 返回队列头部的元素 如果队列为空，则返回null
put 添加一个元素 如果队列满，则阻塞
take 移除并返回队列头部的元素 如果队列为空，则阻塞

1、队尾入队方法 add(E), offer(E) 在尾部添加:

boolean add(E e);
boolean offer(E e);
相同点：实现类禁止添加 null 元素，否则会报空指针 NullPointerException；
不同点： add() 方法在添加失败（比如队列已满）时会报 一些运行时错误 错；而 offer() 方法即使在添加失败时也不会奔溃，只会返回 false

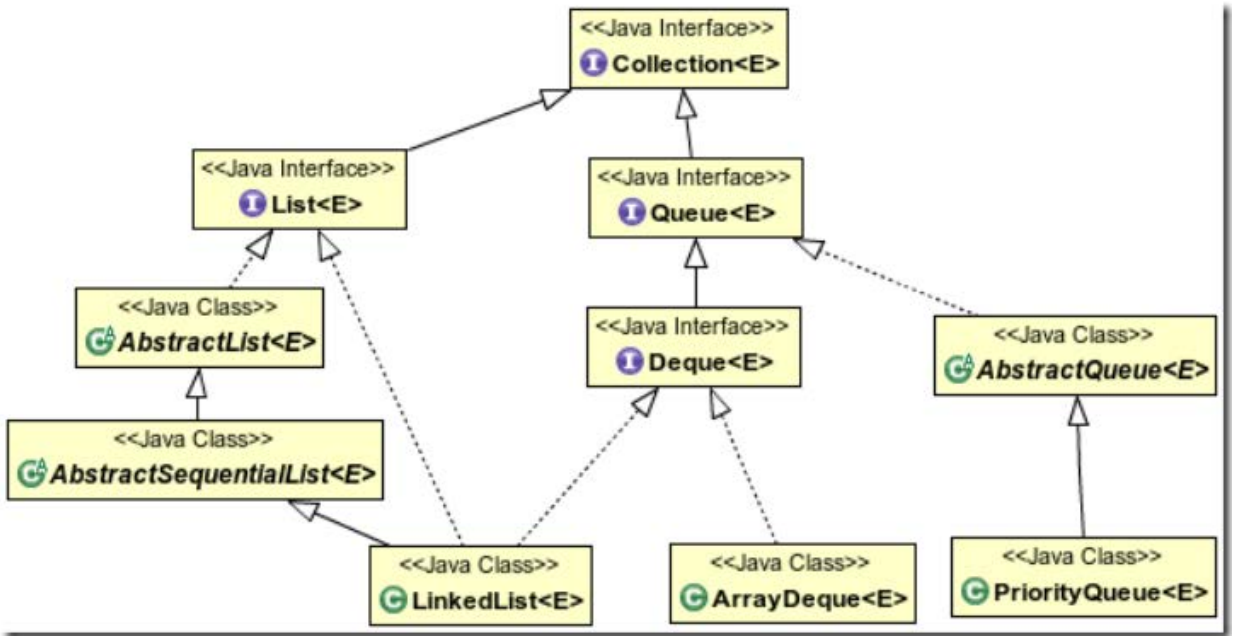
2、队首出队方法remove(), poll() 删除并返回头部

E remove();
E poll();
当队列为空时 remove() 方法会报 NoSuchElementException 错; 而 poll() 不会奔溃，只会返回 null。

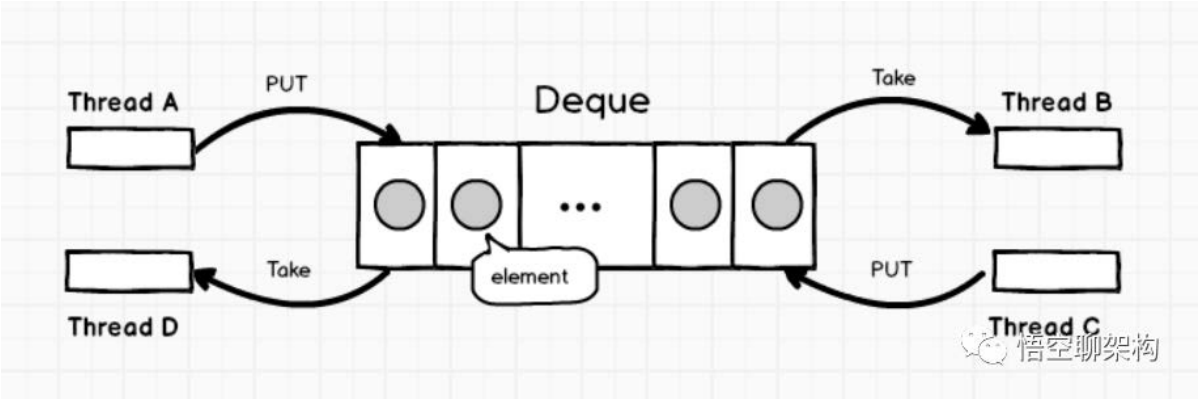
3、获取队首元素，但不删除element(), peek()

E element();
E peek();
当队列为空时 element() 抛出异常；peek() 不会奔溃，只会返回 null。

Queue接口常见实现类



双端队列Deque接口



双端队列Deque

（1）Deque概念：支持两端元素插入和移除的线性集合。名称deque是双端队列的缩写，通常发音为deck。大多数实现Deque的类，对它们包含的元素的数量没有固定的限制的，支持有界和无界。

	First Element (Head)	First Element (Head)	Last Element (Tail)	Last Element (Tail)
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Deque方法

说明：

该列表包含包含访问deque两端元素的方法，提供了插入，移除和检查元素的方法。

这些方法种的每一种都存在两种形式：如果操作失败，则会抛出异常，另一种方法返回一个特殊值（null或false，取决于具体操作）。

插入操作的后一种形式专门设计用于容量限制的Deque实现，大多数实现中，插入操作不能失败，所以可以用插入操作的后一种形式。

Deque接口扩展了Queue接口，当使用deque作为队列时，作为FIFO。元素将添加到deque的末尾，并从头开始删除。

作为FIFO时等价于Queue的方法如下表所示：即是用双端队列Deque实现单端队列Queue的FIFO功能

Queue 方法	Deque 等价方法
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

Deque实现Stack的LIFO功能

Deque也可以用作LIFO（后进先出）栈，这个接口优于传统的Stack类。当作为栈使用时，元素被push到deque队列的头，而pop也是从队列的头pop出来。

Stack（栈）的方法正好等同于Deque的如下方法

Stack 方法	Deque 等价方法
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

LinkedList实现了Deque双端队列接口

Demo:LinkedList是一个双向链表

```
Deque queue=new LinkedList();
// insert:双端队列首尾插入
queue.addFirst("a");
queue.offerFirst("b");
queue.addLast("y");
queue.offerLast("z");
//remove双端队列首尾删除
queue.removeFirst();
queue.pollFirst();
queue.removeLast();
```

因此：LinkedList可以实现列表List,FIFO队列Queue,双向队列Deque,LIFO的stack

LinkedList的自定义序列化规则：不序列化Node,序列化Node内部的item对象，节省内存

```
private void writeObject(java.io.ObjectOutputStream s)//序列化内容为list的size数字和所有的有序item
    throws java.io.IOException {
    // Write out any hidden serialization magic
    s.defaultWriteObject();// 写出非transient非static属性

    // Write out size写出元素个数
    s.writeInt(size);

    // Write out all elements in the proper order.// 有序写出元素，序列化出Node对象内部指向的item对象
    for (Node<E> x = first; x != null; x = x.next)
        s.writeObject(x.item);
}

private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in any hidden serialization magic
    s.defaultReadObject();

    // Read in size 读取元素个数
    int size = s.readInt();

    // Read in all elements in the proper order.
    for (int i = 0; i < size; i++) 有序读出元素item，调用linklast方法，把iterm封装成Node添加到LinkedList
        linkLast((E)s.readObject());
}
```