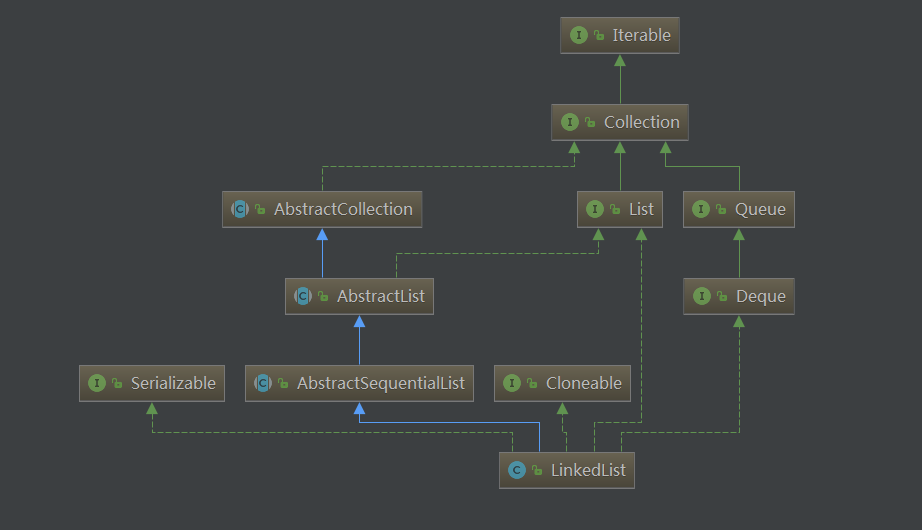
# LinkedList源码分析



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

## Node节点定义

|  |
| --- |
| **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;  }  } |

Node是LinkedList的静态内部类，从节点的结构可知，LinkedList是通过双向链表实现的，是不是双向循环呢？我们等会代码中就会知道了

## 成员变量

|  |
| --- |
| **//链表元素的的长度**  **transient** **int** size = 0;  //链表的头结点  **transient** Node<E> first;  //链表的尾结点  **transient** Node<E> last; |

## 构造方法

|  |
| --- |
| //构造一个空的链表  **public** LinkedList() {  }  //先构造一个空表，然后将集合使用addAll方法添加进链表中  **public** LinkedList(Collection<? **extends** E> c) {  **this**();  addAll(c);  } |

addAll方法放到后面分析。

思考：为什么LinkedList没有提供像ArrayList里面的ArrayList(int initialCapacity)构造指定大小的list的构造方式？

因为ArrayList的这种构造方式，可以知道需要向系统申请多少个连续的空间，从而构造指定空间的数组，这样可以避免空间的浪费；但是LinkedList的空间可连续可不连续，并且动态增加元素很轻松，按照需要add就行了。

## 添加元素

|  |
| --- |
| **//将指定元素连接到表尾，只返回true？？因为只要空间分配足够，就不会添加失败**  **//当空间不够的时候，内存溢出，抛出OOM**  **public** **boolean** add(E e) {  linkLast(e);  **return** **true**;  }  //将e添加到链表的尾部  **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++;  }  Node(Node<E> prev, E element, Node<E> next) {  **this**.item = element;  **this**.next = next;  **this**.prev = prev;  } |

具体过程如下：

①暂存链表旧的尾节点l

②创建一个新节点，prev = l，item=e，next = null；

③新节点赋值为新的尾节点

④判断原来尾结点是否为空（链表是否为空）

4.1如果为空，让头结点指向新节点，即当只有一个节点，头尾结点指向同一个节点

4.2如果不为空，让旧的尾节点next指向新的节点

⑤长度加1

**此时可以知道，新节点的next=null，即新的尾节点的next=null，那么由此可见，LinkedList只是双向链表不是双向循环链表**

|  |
| --- |
| **//上述过程已描述，不再赘述**  **public** **void** addLast(E e) {  linkLast(e);  }  //将元素连接到表头  **public** **void** addFirst(E e) {  linkFirst(e);  }  **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++;  } |

过程分析：

①暂存链表旧的头节点f

②创建一个新节点，prev = null，item=e，next = f；

③新节点赋值为新的头节点

④判断原来头结点是否为空（链表是否为空）

4.1如果为空，让尾结点指向新节点，即当只有一个节点，头尾结点指向同一个节点

4.2如果不为空，让旧的头节点prev指向新的节点

⑤长度加1

|  |
| --- |
| **//push方法，对比入栈方法，往头结点添加元素**  **public** **void** push(E e) {  addFirst(e);  }  //添加元素到链表尾部，内部就是调用add（e）  **public** **boolean** offer(E e) {  **return** add(e);  }  //添加元素到链表头部  **public** **boolean** offerFirst(E e) {  addFirst(e);  **return** **true**;  }  //添加元素到末尾  **public** **boolean** offerLast(E e) {  addLast(e);  **return** **true**;  }  //添加指定元素到指定位置  **public** **void** add(**int** index, E element) {  checkPositionIndex(index);//检查是否越界  **if** (index == size)//如果等于size，则直接调用插入表尾方法  linkLast(element);  **else//如果不在表尾，在index对应的节点前插入元素，该方法后面详解**  linkBefore(element, node(index));  }  //如果isPositionIndex为false抛出数组越界异常  **private** **void** checkPositionIndex(**int** index) {  **if** (!isPositionIndex(index))  **throw** **new** IndexOutOfBoundsException(outOfBoundsMsg(index));  }  //检查index是否在0-size中  **private** **boolean** isPositionIndex(**int** index) {  **return** index >= 0 && index <= size;  }  //返回指定index处的非空节点  Node<E> node(**int** index) {  // assert isElementIndex(index);  /\*  这里思想很巧妙，如果index在链表前半部分，从first往后找，否则，从last往前找  \*/  **if** (index < (size >> 1)) {//判断index是否在前半部分  Node<E> x = first;  **for** (**int** i = 0; i < index; i++)//从第一个节点往后找  x = x.next;  **return** x;  } **else** {//后半部分  Node<E> x = last;  **for** (**int** i = size - 1; i > index; i--)从最后一个节点往前找  x = x.prev;  **return** x;  }  }  //在非空节点succ之前插入元素  **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++;  }  **①记录succ的前一个节点**  **②产生新的节点，prev=pred，item=e，next=succ**  **③succ的prev指向新节点**  **④判断succ前一个节点pred是否为空**  **4.1 如果为空，即succ为链表的头结点，first指向新的节点作为头结点**  **4.2 如果不为空，pred的next指向新的节点**  **⑤长度加1** |
| //按照指定集合的迭代器返回的顺序将指定集合中的所有元素追加到此**列表的末尾**。  **public** **boolean** addAll(Collection<? **extends** E> c) {  **return** addAll(size, c);  }  **//**将指定集合中的所有元素插入到此列表中，从指定的位置开始  **public** **boolean** addAll(**int** index, Collection<? **extends** E> c) {  **//①判断是否越界**  checkPositionIndex(index);  **//②将集合转换为数组**  Object[] a = c.toArray();  **//③记录需要插入的集合元素个数**  **int** numNew = a.length;  **//④如果元素个数为0，直接返回false**  **if** (numNew == 0)  **return** **false**;  /\*  **⑤判断一下index与size是否相等**  **5.1相等就插入表尾**  **5.2不等就插入index处**  \*/  Node<E> pred, succ;  **if** (index == size) {  succ = **null**;  pred = last;  } **else** {  //找到index位置的节点，赋值给succ，succ节点之前是pred，  //即在succ与pred之间插入数据  succ = node(index);  pred = succ.prev;  }  **//⑥循环将集合中所有元素连接到pred后面，过程与succ无关，即结束后**  **//新插入的最后一个节点的next=null；还需要进行链接**  **for** (Object o : a) {  @SuppressWarnings("unchecked") E e = (E) o;  Node<E> newNode = **new** Node<>(pred, e, **null**);  //如果pred为空，说明succ为头结点，那么将newNode作为新的头结点  **if** (pred == **null**)  first = newNode;  //如果不为空就让pred的next指向newNode  **else**  pred.next = newNode;  //newNode作为新的pred  pred = newNode;  }  //**⑦此时就是为了链接，判断succ是否为空**  **7.1如果为空，说明最后一个插入的元素（现在是pred）就是尾节点**  **7.2如果不为空，说明只要和succ链接即可**  **if** (succ == **null**) {  last = pred;  } **else** {  pred.next = succ;  succ.prev = pred;  }  //**⑧链表长度加numNew**  size += numNew;  modCount++;  **return** **true**;  } |

## 删除元素

|  |
| --- |
| //移除链表的第一个节点  **public** E remove() {  **return** removeFirst();  }  //移除链表第一个节点  **public** E removeFirst() {  **final** Node<E> f = first;  //注意：如果是空链表，抛异常没有该元素  **if** (f == **null**)  **throw** **new** NoSuchElementException();  **return** unlinkFirst(f);  }  //将第一个节点删掉，f就是first  **private** E unlinkFirst(Node<E> f) {  // assert f == first && f != null;  **//①记录第一个节点的数据值**  **final** E element = f.item;  **//②记录下一个节点**  **final** Node<E> next = f.next;  **//③将第一个节点置空 帮助回收GC**  f.item = **null**;  f.next = **null**; // help GC  **//④将下一个节点作为新的头结点**  first = next;  **//⑤如果下一个为空，表示first就是唯一一个元素，删除后成空表**  **//如果不为空，就将下一个节点的prev=null**  **if** (next == **null**)  last = **null**;  **else**  next.prev = **null**;  **//⑥链表长度-1**  size--;  modCount++;  **return** element;  }  //移除指定位置元素  **public** E remove(**int** index) {  checkElementIndex(index);  //node(index)找到index处的节点  **return** unlink(node(index));  }  //移除节点x  E unlink(Node<E> x) {  // assert x != null;  //①记录该节点数据值，前一个节点，后一个节点  **final** E element = x.item;  **final** Node<E> next = x.next;  **final** Node<E> prev = x.prev;  //②前一个节点是否为空  2.1如果为空，x为头结点，将x.next作为新的头结点  2.2如果不为空，前一个节点的next指针指向x的下一个节点  **if** (prev == **null**) {  first = next;  } **else** {  prev.next = next;  x.prev = **null**;  }  //③判断x后一个节点是否为空  3.1如果为空，说明x节点是尾节点，让x的prev称为新的尾节点  3.2如果不为空，将x的next的prev指向x的prev  **if** (next == **null**) {  last = prev;  } **else** {  next.prev = prev;  x.next = **null**;  }  //④x节点数据值清空  x.item = **null**;  //⑤链表长度-1  size--;  modCount++;  **return** element;  }  //从链表中删除第一次出现的o对象，不管o是否为null  //①判断o是否为空  //1.1如果为空，找到第一个数据值为null的节点，进行删除  //1.2如果非空找到第一个与o的数据值相等的节点  **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**;  }  //从链表中删除第一次出现的指定元素o，内部调用remove(object)  **public** **boolean** removeFirstOccurrence(Object o) {  **return** remove(o);  }  //删除链表的最后一个元素并返回  **public** E removeLast() {  **final** Node<E> l = last;  **if** (l == **null**)  **throw** **new** NoSuchElementException();  **return** unlinkLast(l);  }  //移除链表的最后一个元素，l为last  **private** E unlinkLast(Node<E> l) {  // assert l == last && l != null;  //①记录尾节点的数据值、以及前一个节点  **final** E element = l.item;  **final** Node<E> prev = l.prev;  //②将尾节点置为空 方便GC  l.item = **null**;  l.prev = **null**; // help GC  //③指定新的尾节点，为l的prev  last = prev;  //④判断prev是否为null  //4.1为空说明该表只有一个节点last，删除之后，为空表  //4.2不为空，就将l的前一个节点的next置为null  **if** (prev == **null**)  first = **null**;  **else**  prev.next = **null**;  //⑤链表长度-1  size--;  modCount++;  //⑦返回之前的尾节点的数据  **return** element;  } |
| //从链表中删除最后一次出现o的指定元素o  **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**;  }  //获取第一个元素的同时删除第一个元素，当链表无节点时，**不会报错**  **public** E poll() {  **final** Node<E> f = first;  **return** (f == **null**) ? **null** : unlinkFirst(f);  }  //获取第一个元素的同时删除第一个元素，当链表无节点时，**会报错**  **public** E pop() {  **return** removeFirst();  } |

## 修改元素

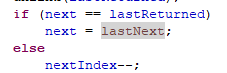
|  |
| --- |
| **public** E set(**int** index, E element) {  //①入参检测  checkElementIndex(index);  //②找到index处节点  Node<E> x = node(index);  //③保存当前节点旧值  E oldVal = x.item;  //④替换新值  x.item = element;  **return** oldVal;  } |
|  |
|  |

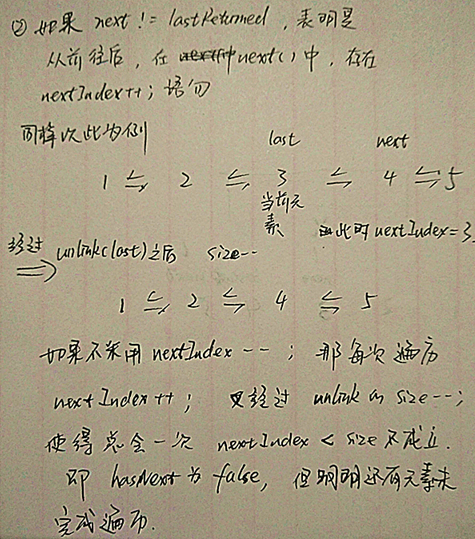
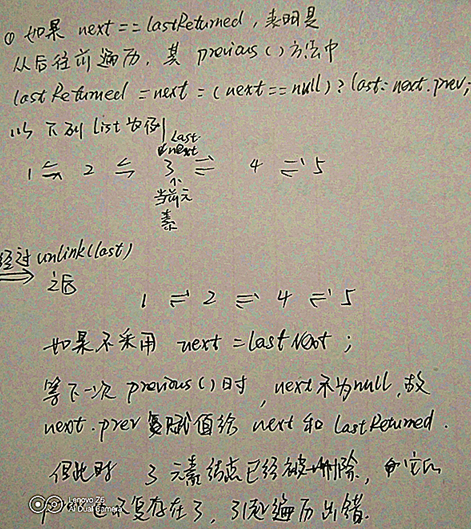
## 查询元素

|  |
| --- |
| **//得到链表的第一个元素**  **public** E element() {  **return** getFirst();  }  **public** E getFirst() {  **final** Node<E> f = first;  **if** (f == **null**)  **throw** **new** NoSuchElementException();  **return** f.item;  }  //得到指定索引的元素  **public** E get(**int** index) {  //①入参检验  checkElementIndex(index);  //②获取指定索引处节点数据值  **return** node(index).item;  }  //得到第一个元素  **public** E getFirst() {  **final** Node<E> f = first;  **if** (f == **null**)  **throw** **new** NoSuchElementException();  **return** f.item;  }  //得到最后一个元素  **public** E getLast() {  **final** Node<E> l = last;  **if** (l == **null**)  **throw** **new** NoSuchElementException();  **return** l.item;  } |

## 遍历

|  |
| --- |
| **//获得listIterator迭代器**  **public** ListIterator<E> listIterator(**int** index) {  checkPositionIndex(index);  **return** **new** ListItr(index);  }  //实现了ListIterator接口  **private** **class** ListItr **implements** ListIterator<E> {  //上一次返回的节点  **private** Node<E> lastReturned;  //下一个节点  **private** Node<E> next;  //下一个节点的索引  **private** **int** nextIndex;  **private** **int** expectedModCount = modCount;  //构造指定索引的迭代器  ListItr(**int** index) {  // assert isPositionIndex(index);  /\*  如果是最后一个节点，那么next=null；  如果不是最后一个节点，那么找到该index索引处的节点  \*/  next = (index == size) ? **null** : node(index);  nextIndex = index;  }  **public** **boolean** hasNext() {  //判断是否还有下一个元素  **return** nextIndex < size;  }    //获取下一个元素  **public** E next() {  checkForComodification();  //①如果没有下一个抛出异常  **if** (!hasNext())  **throw** **new** NoSuchElementException();  //②将本次访问到的next节点赋值为lastReturned  lastReturned = next;  //③next后移  next = next.next;  //④索引++；  nextIndex++;  //⑤返回本次的值  **return** lastReturned.item;  }    //是否还有前一个元素，nextIndex是否大于0  **public** **boolean** hasPrevious() {  **return** nextIndex > 0;  }    //获取前一个元素  **public** E previous() {  checkForComodification();  //①如果没有前一个元素，则抛异常  **if** (!hasPrevious())  **throw** **new** NoSuchElementException();  //②next==null表明当前节点就是尾节点的next，  //故把next设置为last，  //如果next不是null，那就指向将next指向next之前  lastReturned = next = (next == **null**) ? last : next.prev;  //③索引值-1，返回当前last对应的元素  nextIndex--;  **return** lastReturned.item;  }  **public** **int** nextIndex() {  **return** nextIndex;  }  **public** **int** previousIndex() {  **return** nextIndex - 1;  }  //移除当前遍历到的元素  **public** **void** remove() {  checkForComodification();  //①移除当前遍历到的元素，如果为空，直接抛异常  **if** (lastReturned == **null**)  **throw** **new** IllegalStateException();    //②记录当前节点的下一个节点  Node<E> lastNext = lastReturned.next;  //③移除当前节点  unlink(lastReturned);    **if** (next == lastReturned)  next = lastNext;  **else**  nextIndex--;  lastReturned = **null**;  expectedModCount++;  } |

分析为什么



|  |
| --- |
| **//设定当前正在遍历的值**  **public** **void** set(E e) {  **if** (lastReturned == **null**)  **throw** **new** IllegalStateException();  checkForComodification();  lastReturned.item = e;  }  //添加一个值  **public** **void** add(E e) {  checkForComodification();  lastReturned = **null**;  **if** (next == **null**)  linkLast(e);  **else**  linkBefore(e, next);  nextIndex++;  expectedModCount++;  }  //循环，往后遍历，每遍历一个节点就回调当前节点数据值  **public** **void** forEachRemaining(Consumer<? **super** E> action) {  Objects.*requireNonNull*(action);  **while** (modCount == expectedModCount && nextIndex < size) {  action.accept(next.item);  lastReturned = next;  next = next.next;  nextIndex++;  }  checkForComodification();  } |

## 总结

①底层是双向链表的存储数据，并记录了头结点和尾节点

②添加元素非常快，如果添加头部和尾部更快，因为已经有了头尾节点，如果添加道中间，需要先找到索引处的元素

③遍历的时候，建议采用forEach遍历，这样可以在每次获取下一个元素时都非常轻松，如果是通过for和get进行遍历，效率比较低