# LinkedList

## 简介

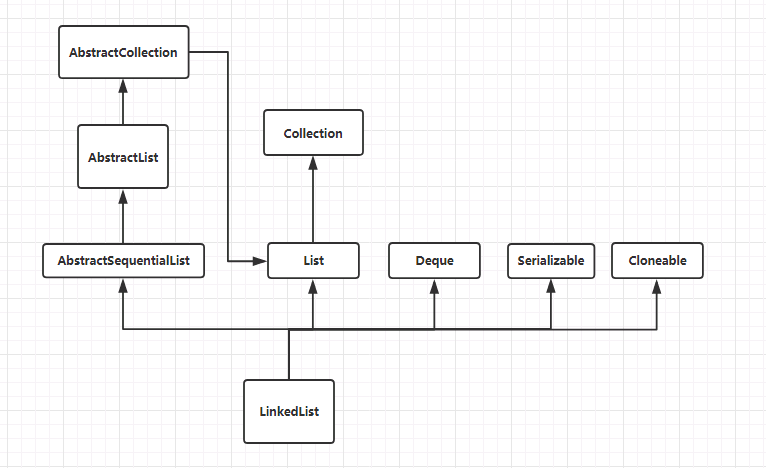
使用链表结构存储数据，适合数据的动态添加或者删除，随机访问或者变量的效率比较低；双向链表，实现list和queue接口,既可以看着是一个顺序容器，也可以看作是一个队列，允许存放null元素；LinkedList元素的添加修改等操作不是线程安全的，要保证链表在多线程环境下的线程安全需要Collections的synchronizedList方法来包装链表返回一个线程安全的链表

LinkedList<String> ll = **new** LinkedList<String>();

List<String> lls = Collections.*synchronizedList*(ll);

源码分析

LinkedList底层是通过双向链表实现的，双向链表的节点用Node表示



## 内部类Node

**private** **static** **class** Node<E> {

E item;存放值

Node<E> next;指向下一个节点

Node<E> prev;指向前一个节点

Node(Node<E> prev, E element, Node<E> next) {

**this**.item = element;

**this**.next = next;

**this**.prev = prev;

}

}

## 变量

**transient** **int** size = 0;节点数

**transient** Node<E> first;指向第一个节点

**transient** Node<E> last; 指向最后一个节点

protected transient int modCount = 0;定义在AbstractList中，用来记录新增删除节点操作的次数

## 构造函数

无参构造函数

**public** LinkedList() {

}

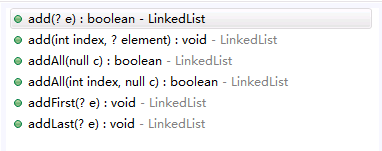
**public** LinkedList(Collection<? **extends** E> c) {

**this**();执行无参构造函数

addAll(c);将集合转换为LinkedList

}

## 添加元素add

****

add(e) :在链表最后添加元素，这个方法等同图addLast(e);只需要修改新节点和last节点的相关引用即可

add(int index,e);在指定位置添加元素；需要先通过线下查找找到指定位置插入元素，然后修改相关引用完成插入操作

addAll(c) 将集合添加到链表中，从末尾位置开始添加

addFirst(e)在起始位置添加

addLast(e)在末尾位置添加

add(e)从链表未部添加节点

**public** **boolean** add(E e) {

linkLast(e);添加是在这个方法中完成的

**return** **true**;

}

**void** linkLast(E e) {

当LinkedList使用无参构造初始化时，first和last都为null

**final** Node<E> l = last; 让l指向last节点，1

**final** Node<E> newNode = **new** Node<>(l, e, **null**);创建新节点，node构造函数中新建的节点的上个节点指向的是l(也就是添加之前的last指向的节点)2

last = newNode;last指向新节点 3

**if** (l == **null**)

first = newNode;初始化时first和last都为null，所以first也指向新节点，这是添加第一个元素

**else**

l.next = newNode;当l节点不为空时，让l的next节点指向新节点5

size++;记录添加的元素的个数

modCount++;

}

add(index,e)这添加操作的逻辑稍显复杂，可以分成2部，

1. 先根据index找到要插入的位置，
2. 修改引用完成插入操作

public void add(int index, E element) {

checkPositionIndex(index);对index进行校验，不在当前范围内就抛出异常

if (index == size)//index==size表示在最后添加，等同于add(e)

linkLast(element);

else

在链表中间位置添加

linkBefore(element, node(index));

}

找到列表中index的原节点

Node<E> node(int index) {

// assert isElementIndex(index);

size >> 1：向右位移，表示除以2，当index小于这个位移结果，那么index位置在上半区，大于的话在下半区

上半区就从first节点开始寻找，下半区从last节点开始寻找（二分查找）

最终找到index位置上的节点

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;

}

}

找到添加位置的节点succ后，将succ节点的prev节点中的next节点执行新节点，新节点的prev节点

执行succ节点的prev节点，新节点的next节点执行succ节点，succ的prev节点执行新节点

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;

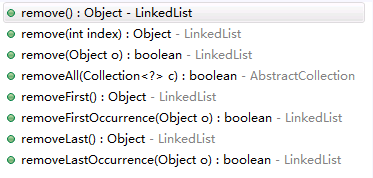
size++;

modCount++;

}

## 删除元素

删除元素，删除节点后返回的是删除节点存储的对象



remove()删除的是first节点，方法返回的就是first节点存放的元素

public E remove() {

return removeFirst();

}

public E removeFirst() {

final Node<E> f = first;

if (f == null)

throw new NoSuchElementException();

return unlinkFirst(f);执行删除的方法

}

private E unlinkFirst(Node<E> f) {

// assert f == first && f != null;

final E element = f.item;取出f节点的值

final Node<E> next = f.next;取出f节点的next节点

f.item = null; f节点的item引用指向null

f.next = null; // help GC f的next指向null，那么f引用就可以被gc回收

first = next;让first指向first的next节点

if (next == null)

last = null;next为null，那么删除的几点就是last节点

else

next.prev = null;next不为空，那么next就称为了first，将prev节点设置为null

size--;

modCount++;

return element;

}

删除指定位置的节点

public E remove(int index) {

checkElementIndex(index);

return unlink(node(index));首先找到指定位置的节点，然后删除

}

修改节点的prev节点和next节点的指向，返回删除节点的存放的值

E unlink(Node<E> x) {

// assert x != null;

final E element = x.item;取出x节点的值

final Node<E> next = x.next;取出x节点的next节点

final Node<E> prev = x.prev;取出x节点的prev节点

if (prev == null) { prev == null表示x为first节点，就然first指向next

first = next;

} else {

prev.next = next;

x.prev = null;

}

if (next == null) {

last = prev;

} else {

next.prev = prev;

x.next = null;

}

x.item = null;

size--;

modCount++;

return element;

}

修改元素set将指定下标出的元素修改成指定值，先通过node方法找到index位置的元素，然后将Node的item值修改成新的值

**public** E set(**int** index, E element) {

checkElementIndex(index);

Node<E> x = node(index);

E oldVal = x.item;

x.item = element;

**return** oldVal;

}

获取元素

获取指定位置的节点

**public** E get(**int** index) {

checkElementIndex(index);

**return** node(index).item;

}

获取第一个节点的元素

public E getFirst() {

final Node<E> f = first;

if (f == null)

throw new NoSuchElementException();

return f.item;

}

获取最后一个节点的元素

public E getLast() {

final Node<E> l = last;

if (l == null)

throw new NoSuchElementException();

return l.item;

}

## Deque接口

LinkedList实现了Deque接口，它也可以看做是一个队列，实现的Deque的方法

队列的方法

Peek();返回第一个节点的元素

poll()：删除第一个节点并返回节点元素

迭代

内部类

ListItr 实现java.util.ListIterator<E>接口，ListIterator接口继承java.util.Iterator<E>接口

变量

**private** Node<E> lastReturned;指向返回的节点

**private** Node<E> next;指向当前对象初始化时指定位置的节点

**private** **int** nextIndex; next节点的位置索引

**private** **int** expectedModCount = modCount;记录创建对象时修改链表数量的操作次数，初始化后不能修改

构造函数

ListItr(**int** index) {

next = (index == size) ? **null** : node(index);获取index位置的节点

nextIndex = index;

}

Next()方法

**public** E next() {

checkForComodification();//校验操作数

**if** (!hasNext())

**throw** **new** NoSuchElementException();

lastReturned = next;

next = next.next;next节点直线它的next节点

nextIndex++;

**return** lastReturned.item;返回节点的元素

}

**final** **void** checkForComodification() {

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

}

创建迭代对象

listIterator();ListItr构造默认index为0

listIterator(index);指定index

iterator();ListItr构造默认index为0

## for each

**使用for each循环对LinkedList对象进行添加或者删除操作时**

LinkedList的继承体系中实现了Iterable接口，那么表示它可以使用foreach循环遍历集合，LinkedList自身没有重写Iterable接口的iterator方法，但是父类AbstractSequentialList已经重写了这个方法；所以使用foreach标签时会调用iterator方法创建一个实现Iterator接口的迭代对象ListItr，然后每循环一次就会调用一次迭代对象的next()方法来获取节点数据；在next方法中会调用checkForComodification方法来校验expectedModCount和modCount是否相等，不相等抛出java.util.ConcurrentModificationException异常；（expectedModCount将要执行操作的操作次数，modCount已经执行的操作次数）；

在循环体中对链表进行了删除或者添加操作，那么modCount就会改变，此时ListItr的expectedModCount值没有改变

那么在下次循环时调用next方法执行checkForComodification方法校验expectedModCount和modCount就不相等，那么就抛出异常

所以在foreach循环体中最多只能执行异常修改链表数量的操作,最好不要再foreach循环体执行修改链表数量的操作

**LinkedList for循环遍历比较慢的原因是**

for循环遍历通过循环get方法来获取的，get内部又是循环遍历链表的，虽然通过二分查找做了优化，但是LinkedList在get任何一个位置的数据的时候，都会把前面的数据走一遍，越往后效率就会越低，数据量越大效率会越来越低，所以最好不要使用for循环遍历

遍历LinkedList使用foreach循环或者迭代器最好，这种方式是通过内存地址去找数据，能大大提升访问效率

LinkedList:添加删除的效率比ArrayList高，它不需要位移节点；

foreach循环，在循环时直接调用next()方法获取下一个节点的元素

在java.util.AbstractList<E>中重写了equals和hashCode方法

**public** **boolean** equals(Object o) {

**if** (o == **this**)

**return** **true**;

**if** (!(o **instanceof** List))

**return** **false**;

获取迭代对象

ListIterator<E> e1 = listIterator();

ListIterator<?> e2 = ((List<?>) o).listIterator();

遍历迭代对象，

**while** (e1.hasNext() && e2.hasNext()) {

E o1 = e1.next();取出节点元素

Object o2 = e2.next();取出节点元素

判断节点元素是否相等，通过equals方法判断，

**if** (!(o1==**null** ? o2==**null** : o1.equals(o2)))

**return** **false**;

}

**return** !(e1.hasNext() || e2.hasNext());

}

链表中只有一个节点first和last都指向这个节点

添加

删除

查找：get(i)内部优化为二分查找，但是回去需要的结果之前，要把之前的节点都会访问一遍

