**LinkedList的操作主要有：在头/尾添加/删除元素、在任意位置添加/删除元素、得到/修改某位置元素的值、某元素在链表中的位置、链表长度等**

**一、LinkedList实现原理概述**

LinkedList和ArrayList一样，都实现了List接口，但其内部的数据结构有本质的不同。LinkedList是基于链表实现的（通过名字也能区分开来），所以它的插入和删除操作比ArrayList更加高效。但也是由于其为基于链表的，所以随机访问的效率要比ArrayList差。

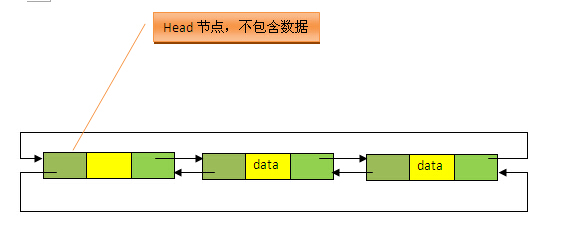
**二、LinkedList类定义**

public class LinkedList<E>

extends AbstractSequentialList<E>

implements List<E>,Deque<E>,Cloneable,java.io.Serializable

1. LinkedList是一个继承于AbstractSequentialList的双向链表（双链表的每个结点都有一个头指针first和尾指针last）。它也可以被当作堆栈、队列或双端队列进行操作。

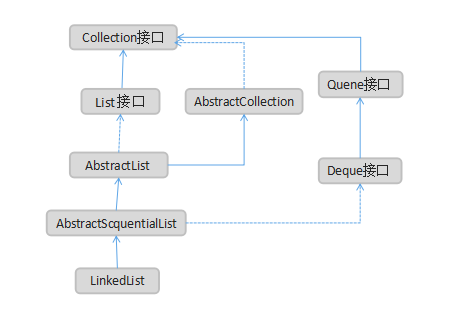


1. LinkedList实现List接口，能对它进行队列操作。
2. LinkedList实现Deque接口，即能将LinkedList当作双端队列使用。
3. LinkedList实现了Cloneable接口，即覆盖了函数clone()，能克隆。
4. LinkedList实现java.io.Serializable接口，这意味着LinkedList支持序列化，能通过序列化去传输。
5. LinkedList是非同步的。

为什么要继承自AbstractSequentialList?

AbstractSequentialList实现了get(int index)、set(int index,E element)、add(int index,E element)和remove(int index)这些骨干性函数。降低了List接口的复杂度。这些接口都是随机访问List的，LinkedList是双向链表；既然它继承于AbstractSequentialList，就相当于已经实现了“get(int index)这些接口”。此外，我们若需要通过AbstractSequentialList自己实现一个列表，只需要扩展此类，并提供listIterator()和size()方法的实现即可。若要实现不可修改的列表，则需要实现列表迭代器的hasNext、next、hasPrevious、previous和index方法即可。

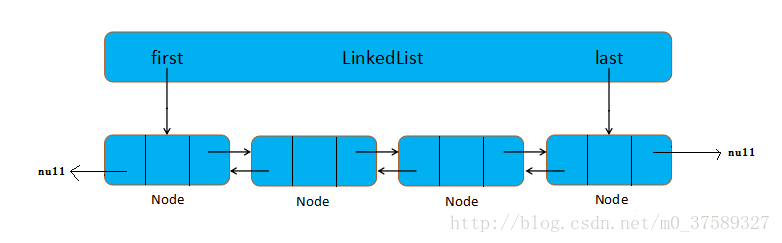
LinkedList的类图关系：



## LinkedList内部原理及特点

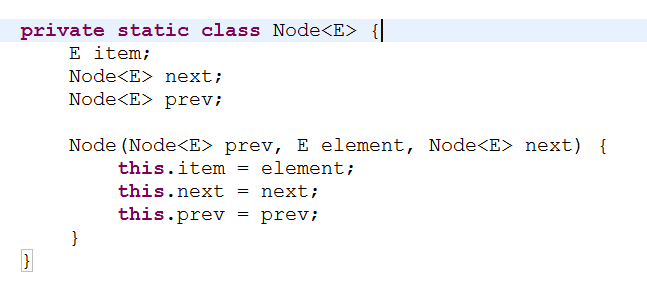
1. 不是线程安全的，没有Synchronized关键字。
2. 允许向其中添加null。
3. 内部是由一个一个的内部类Node前后指针连接组成的链，LinkedList相当于一个Node管理类（如下图所示），有两根指针first和last一直指向Node链的头部和尾部，其优势在于增加和删除元素较快，但不适合查找操作。
4. 其迭代器是快速失败的，在一个LinkedList的迭代器创建之后调用非迭代器中的方法对容器的内部结构进行修改都会抛出

ConcurrentModificationException。



在了解LinkedList的实现原理之前，我们首先需要明白LinkedList中的一个节点是什么以及它的具体数据结构。LinkedList每个节点是一个Node类型的实例，每个Node实例除了保存节点的真实值（即真实数据）外，还保存了这个节点的前一个节点的引用和后一个节点的引用，这样就实现了双线链表的数据结构。Node数据结构如下：

LinkedList中的结点用内部类Node实现：



从代码中我们可以看到，当创建一个Node节点时，我们需要传入三个参数，第一个参数就是当前节点的前驱节点，第二个就是节点的真实数据，第三个就是节点的后继节点。

**源码细节分析**

对双向链表的操作无非就是对几根指针进行玩耍，LinkedList中方法有不少，但是功能相同的很多，只是针对空值或者没有元素的情况做出的反应不同。针对不同的接口，应该使用名字相关的方法，比如将LinkedList当队列来使用时应该使用带有offer，poll，peek等名字的方法，当成栈来使用时应该使用带有push，peek，pop等名字的方法，以增加可读性。

　　LinkedList的所有增删改查方法基本上都和底层这几个方法有关：linkFirst, linkLast, linkBefore, unlinkFirst, unlinkLast, unlink, node。

#### ****linkFirst和linkLast****

/\*头插法\*/

private void linkFirst(E e){

final Node<E> f = first;

//创建一个新结点，其前指针为空，后指针指向first所指结点（可能为null） final Node<E> newNode = new Node<>(null,e,f);

first = newNode;//first指针始终指向Node链的头

if(f==null)//如果LinkedList刚创建，此时新创建的结点既是头结点也是尾结点

last = newNode;

else

f.prev = newNode;//如果头插之前有结点，将其前指针指向新结点

size++;

modCount++;//记录修改次数，迭代时检测快速失败

}

/\*尾插法\*/

void linkLast(E e){//操作和头插相反

final Node<E> l = last;

//创建一个新结点，其后指针为空，前指针指向last所指结点（可能为null） final Node<E> newNode = new Node<>(l,e,null);

last = newNode;//last指针始终指向Node链的尾

if(l==null)//如果LinkedList刚创建，此时新创建的结点既是尾结点也是头结点

first = newNode;

else

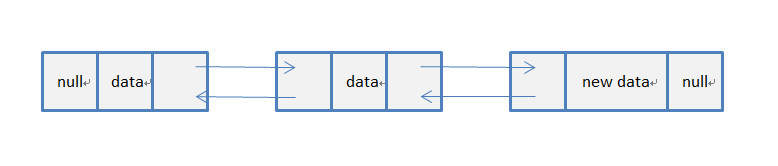
l.next = newNode;//如果尾插之前有结点，将其后指针指向新结点

size++;

modCount++;

}

1. 代码中使用了变量first和last，这两个变量分别保存了当前链表第一个节点和最后一个节点的引用。
2. 在新增一个节点之前，首先把指向最后一个节点的引用（即变量 last）保存起来（即变量 l）。
3. 然后新建一个节点，指定前驱节点是原来链表的最后一个节点，然后把指向最后一个节点的引用（last）指向新建的节点。
4. 紧接着就设置新建节点的前驱节点的后继节点指向新增的节点（考虑链表开始为空和不为空的情况）。
5. 最后把整个链表的总数加1，完成了新增一个节点的操作。完成新增后的链表结构类似如下：



#### ****linkBefore****

/\*在指定结点前插入\*/

void linkBefore(E e,Node<E>succ){

final Node<E> pred = succ.prev;

//创建一个新结点，其前指针指向指定结点的前驱，后指针指向指定结点的后继

final Node<E> newNode = new Node<>(pred,e,succ);

succ.prev = newNode;//指定结点的前指针连上新结点

if(pred == null)//LinkedList刚创建

first = newNode;

else

pred.next = newNode;//指定结点的前驱结点的后指针连上新结点

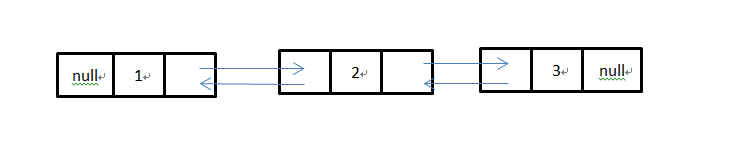
size++;

modCount++;

}

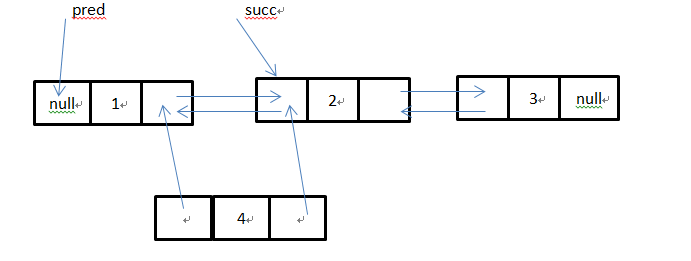
从linkBefore方法我们可以看到，在创建新节点时，同时指定了前驱节点和后继节点，这里后继节点就是我们传进来的第二个参数，所以验证了我们之前所说的第二个参数即为新增节点的后继节点的说法。在创建节点之前，我们把后继节点的前驱节点的引用先保存到一个pred变量中，然后创建一个新的节点，并指定前驱节点和后继节点；接着把后继节点的前驱节点引用指向了新的节点，这样新的节点就可以找到后继节点，接下来根据前驱节点如果为空，则更改first指向新建的节点，否则将前驱节点指向后继节点的引用指向新建的节点，最后把LinkedList的size加1。下面通过图感受一下linkBefore里具体的过程：

现在有一个有三个节点的LinkedList：

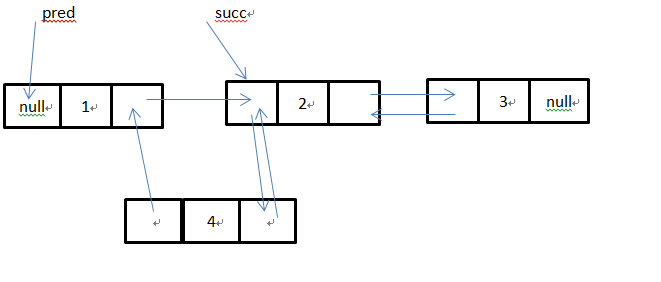


然后我调用add(1,”4”)方法，即新增的节点取代了二号位节点（数据为2）的那个节点的位置，后面的节点将往后移：

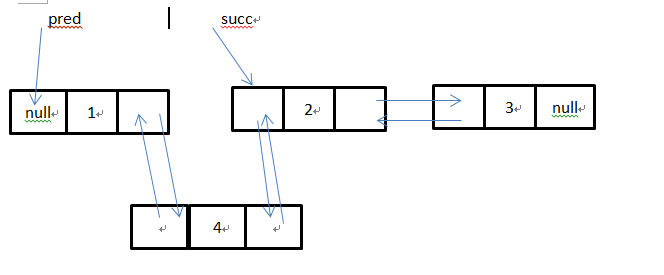
当执行了final Node pred=succ.prev;final Node newNode=new Node<>(pred,e,succ);这两句代码，LinkedList的结构变成这样：



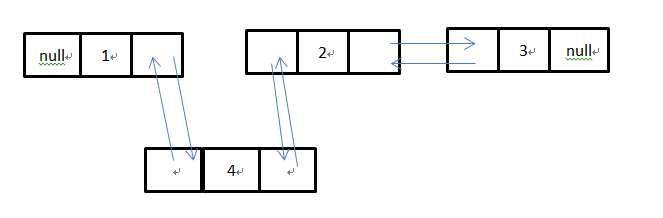
接着执行：succ.prev=newNode;即把三号位的节点指向前驱节点的引用指向新的节点：



这里pred很明显不为空，所以会执行pred.next=newNode;这时候LinkedList结构：



屏蔽掉pred和succ，我们将看得更直观：



执行到这里，我们发现新增的节点已经正确的插入到了指定的位置，最后只需要把size加1即可。

#### ****unlinkFirst, unlinkLast和unlink****

/\*删除头结点\*/

private E unlinkFirst(Node<E>f){//这个f必须是头结点

final E element = f.item;

final Node<E> next = f.next;

f.item = null;

f.next = null;//标记一下有利于垃圾回收

first =next;//first指针始终指向头结点

if(next==null)//LinkedList刚创建

last = null;

else

next.prev = null;//next此时为头结点

size--;

modCount++;

return element;

}

/\*删除尾结点\*/

private E unlinkLast(Node<E>l){//l必须是尾结点

final E element =l.item;//和上面一样

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;

}

/\*删除某个结点\*/

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){//LinkedList刚创建时

first = next;

}else{

prev.next = next;//前驱连接上后继

x.prev = null;//删除

}

if(next==null){//LinkedList刚创建时

last = prev;

}else{

next.prev = prev;//后继连接上前驱

x.next = null;//删除

}

x.item = null;

size--;

modCount++;

return element;

}

#### ****node****

/\* 查找双向链表从头结点往尾结点数的第index个结点 \*/

Node<E>node(int index){

if(index<(size>>1)){//如果index小于链表总长度的一半，则从头往尾找

Node<E> x = first;

for(int i=0;i<index;i++)

x = x.next;

return x;

}else{//如果index大于链表总长度的一半，则从尾往头找

Node<E> x = last;

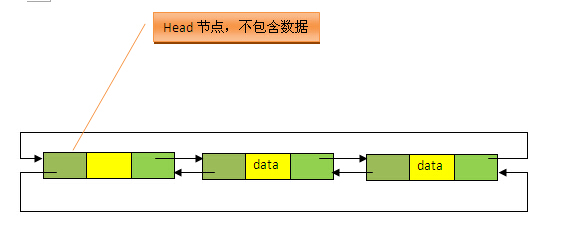
for(int i = size-1;i>index;i--)

x = x.prev;

return x;

}

}

LinkedList底层的数据结构是基于双向循环链表的，且头结点中不存放数据,如下：  


既然是双向链表，那么必定存在一种数据结构——我们可以称之为节点，节点实例保存业务数据，前一个节点的位置信息和后一个节点位置信息，如下图所示：



**四、私有属性**

LinkedList中之定义了三个属性：

**transient** **int** size = 0;

**transient** Node<E> first;

**transient** Node<E> last;

first是指向第一个节点的指针，last是指向最后一个节点的指针，它们是双向链表节点所对应的类Node的实例。Node中包含成员变量：prev,next,item。其中，prev是该节点的上一个节点，next是该节点的下一个节点，item是该节点所包含的值。

size是双向链表中节点实例的个数。

**五、构造方法**

LinkedList提供了两个构造方法。

public LinkedList(){

//构造一个空列表

}

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

/\*第二个构造方法接收一个Collection参数c，

调用第一个构造方法构造一个空的链表，

之后通过addAll将c中的元素全部添加到链表中。

\*/

this();

addAll(c);

}

**（一）get方法包括：**

getFirst()得到链表头元素，会判断链表是否为空，getLast()得到链表末尾元素，会判断链表是否为空；

**（二）remove方法类似**

有removeFirst()，removeLast()；

public E removeFirst(){

final Node<E>f=first;

if(f==null)

throw new NoSuchElementException();

return unlinkFirst(f);

}

public E removeLast(){

final Node<E>l=last;

if(l==null)

throw new NoSuchElementException();

return unlinkLast(l);

}

**（三）add方法有**

addFirst()和addLast()；

public void addFirst(E e){

linkFirst(e);

}

public void addLast(E e){

linkLast(e);

}

**（四）size方法**

得到链表现在的长度；

**以上的方法均能在常数时间完成对链表的操作；**

**（五）clear方法**

依次删除链表中的元素，直至链表为空，执行clear方法后链表会被回收；

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++;

}

**（六）链表的定位方法：**

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;

}

}

返回某位置上的元素；

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;

}

返回某元素第一次出现的位置，同理有返回元素最后一次出现的位置，就不再赘述；

这两个查询方法是线性时间复杂度的，配合前面的方法就可以完成在任意位置的插入，删除，查询和修改了；

public E get(int index) {

checkElementIndex(index);

return node(index).item;

}

public E set(int index, E element) {

checkElementIndex(index);

Node<E> x = node(index);

E oldVal = x.item;

x.item = element;

return oldVal;

}

public void add(int index, E element) {

checkPositionIndex(index);

if (index == size)

linkLast(element);

else

linkBefore(element, node(index));

}

public E remove(int index) {

checkElementIndex(index);

return unlink(node(index));

}

checkElementIndex方法和checkPositionIndex方法用于判断index是不是一个有效值；

**（七）LinkedList中封装了栈与队列的一些方法**

如：peek方法，pop方法，push方法等，都是由上文提到的基本方法实现的；

**（八）LinkedList中有两个Iterator**

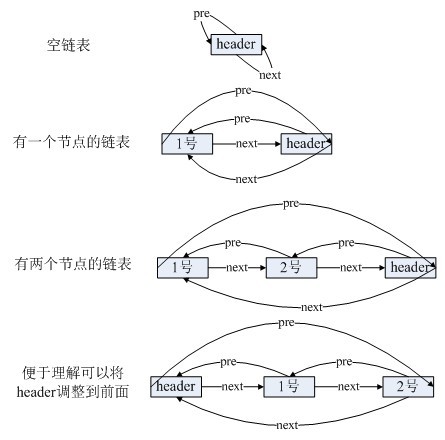
分别负责正序遍历和反序遍历；

private class ListItr implements ListIterator<E>//正序遍历链表；

private class DescendingIterator implements Iterator<E>//反序遍历链表；

Iterator内部实现了next方法，hasnext等方法，对于用户而言，可以直接使用；

对集合的遍历最好采用Iterator；



public void addFirst(E e){

addBefore(e,header.next);

}

public void addLast(E e){

addBefore(e,header);

}

看上面的示意图，结合addBefore(E e,Entry entry)方法，很容易理解addFrist(E e)只需实现在header元素的下一个元素之前插入，即示意图中的一号之前。addLast(E e)只需在实现在header节点前（因为是循环链表，所以header的前一个节点就是链表的最后一个节点）插入节点（插入后在2号节点之后）。

七、删除数据remove()

public E remove(int index){

Entry e=get(index);

remove(e);

return e.element;

}

private void remove(E e){

if(e==header)

throw new NoSuchElementException();

//将前一节点的next引用赋值为e的下一节点

e.previous.next=e.next;

//将e的下一节点的previous赋值为e的上一节点

e.next.previous=e.previous;

//上面两条语句的执行已经导致了无法在链表中访问到e节点，而下面解除了e节点对前后节点的引用

e.next=e.previous=null;

//将被移除的节点的内容设为null

e.element=null;

//修改size大小

size--;

}

由于删除了某一节点因此调整相应节点的前后指针信息，如下：

e.previous.next=e.next;//预删除节点的前一节点的后指针指向预删除节点的后一个节点。

e.next.previous=e.previous;//预删除节点的后一节点的前指针指向预删除节点的前一个节点。

清空预删除节点：

e.next=e.previous=null;

e.element=null;

交给gc完成资源回收，删除操作结束。

与ArrayList比较而言，LinkedList的删除动作不需要“移动”很多数据，从而效率更高。

八、数据获取get()

Get(int)方法的实现在remove(int)中已经涉及过了。首先判断位置信息是否合法（大于等于0，小于当前LinkedList实例的Size），然后遍历到具体位置，获得节点的业务数据（element）并返回。

注意：为了提高效率，需要根据获取的位置判断是从头还是从尾开始遍历。

//获取双向链表中指定位置的节点

private Entry<E>entry(int index){

if(index<0||index>=size)

throw new IndexOutOfBoundsException("Index:"+index+

",Size:"+size);

Entry<E>e=header;

//获取index处的节点。

//若index<双向链表长度的1/2,则从前先后查找;

//否则，从后向前查找。

if(index<(size>>1)){

for(int i=0;i<=index;i++)

e=e.next;

}else{

for(int i=size;i>index;i--)

e=e.previous;

}

return e;

}

注意细节：位运算与直接做除法的区别。先将index与长度size的一半比较，如果index

九、清除数据clear()

public void clear(){

Entry<E>e=header.next;

//e可以理解为一个移动的“指针”，因为是循环链表，所以回到header的时候说明已经没有节点了

while(e!=header){

//保留e的下一个节点的引用

Entry<E>next=e.next;

//解除节点e对前后节点的引用

e.next=e.previous=null;

//将节点e的内容置空

e.element=null;

//将e移动到下一个节点

e=next;

}

//将header构造成一个循环链表，同构造方法构造一个空的LinkedList

header.next=header.previous=header;

//修改size

size=0;

modCount++;

}

十、数据包含contains(Object o)

public boolean contains(Object o){

return indexOf(o)!=-1;

}

//从前向后查找，返回“值为对象(o)的节点对应的索引”不存在就返回-1

public int indexOf(Object o){

int index=0;

if(o==null){

for(Entry e=header.next;e!=header;e=e.next){

if(e.element==null)

return index;

index++;

}

}else{

for(Entry e=header.next;e!=header;e=e.next){

if(o.equals(e.element))

return index;

index++;

}

}

return-1;

}

indexOf(Object o)判断o链表中是否存在节点的element和o相等，若相等则返回该节点在链表中的索引位置，若不存在则放回-1。

contains(Object o)方法通过判断indexOf(Object o)方法返回的值是否是-1来判断链表中是否包含对象o。

十一、数据复制clone()与toArray()

clone()

public Object clone(){

LinkedList<E>clone=null;

try{

clone=(LinkedList<E>)super.clone();

}catch(CloneNotSupportedException e){

throw new InternalError();

}

clone.header=new Entry<E>(null,null,null);

clone.header.next=clone.header.previous=clone.header;

clone.size=0;

clone.modCount=0;

for(Entry<E>e=header.next;e!=header;e=e.next)

clone.add(e.element);

return clone;

}

调用父类的clone()方法初始化对象链表clone，将clone构造成一个空的双向循环链表，之后将header的下一个节点开始将逐个节点添加到clone中。最后返回克隆的clone对象。

toArray()

public Object[]toArray(){

Object[]result=new Object[size];

int i=0;

for(Entry<E>e=header.next;e!=header;e=e.next)

result[i++]=e.element;

return result;

}

创建大小和LinkedList相等的数组result，遍历链表，将每个节点的元素element复制到数组中，返回数组。

toArray(T[]a)

public<T>T[]toArray(T[]a){

if(a.length<size)

a=(T[])java.lang.reflect.Array.newInstance(

a.getClass().getComponentType(),size);

int i=0;

Object[]result=a;

for(Entry<E>e=header.next;e!=header;e=e.next)

result[i++]=e.element;

if(a.length>size)

a[size]=null;

return a;

}

先判断出入的数组a的大小是否足够，若大小不够则拓展。这里用到了发射的方法，重新实例化了一个大小为size的数组。之后将数组a赋值给数组result，遍历链表向result中添加的元素。最后判断数组a的长度是否大于size，若大于则将size位置的内容设置为null。返回a。

从代码中可以看出，数组a的length小于等于size时，a中所有元素被覆盖，被拓展来的空间存储的内容都是null；若数组a的length的length大于size，则0至size-1位置的内容被覆盖，size位置的元素被设置为null，size之后的元素不变。

为什么不直接对数组a进行操作，要将a赋值给result数组之后对result数组进行操作？

十二、遍历数据：Iterator()

LinkedList的Iterator

除了Entry，LinkedList还有一个内部类：ListItr。

ListItr实现了ListIterator接口，可知它是一个迭代器，通过它可以遍历修改LinkedList。

在LinkedList中提供了获取ListItr对象的方法：listIterator(int index)。

public ListIterator<E>listIterator(int index){

return new ListItr(index);

}

该方法只是简单的返回了一个ListItr对象。

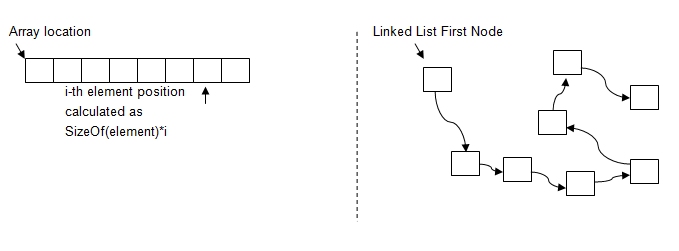
LinkedList中还有通过集成获得的listIterator()方法，该方法只是调用了listIterator(int index)并且传入0。

在讨论链表(linked-list)之前，需要明确几个概念：线性表（顺序表,list,linear list）,数组(array)，链表(linked-list)。

线性表：在中文里，线性表也叫作顺序表。在英文中，它称为list,linear list等。它是最基础、最简单、最常用的一种基本数据结构，线性表总存储的每个数据称为一个元素，各个元素及其索引是一一对应的关系。线性表有两种存储方式：顺序存储方式和链式存储方式。

数组(array)：数组就是线性表的顺序存储方式。数组的内存是连续分配的，并且是静态分配的，即在使用数组之前需要分配固定大小的空间。可以通过索引直接得到数组中的而元素，即获取数组中元素的时间复杂度为O(1)。

链表(linked-list)：链表就是线性表的链式存储方式。链表的内存是不连续的，前一个元素存储地址的下一个地址中存储的不一定是下一个元素。链表通过一个指向下一个元素地址的引用将链表中的元素串起来。



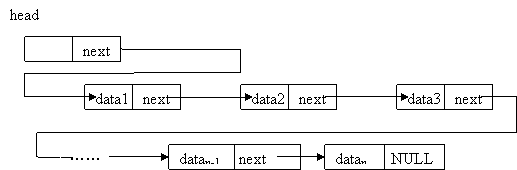
其实为了简便，通常我们也直接将list看做是链表。但是也不必太过纠结这种名称定义，更重要的还是关注数据结构的实现。

**链表分类**

链表分为单向链表(Singly linked list)、双向链表(Doubly linked list)、循环链表(Circular Linked list)。

**单向链表(Singly linked list)**

单向链表是最简单的链表形式。我们将链表中最基本的数据称为节点(node)，每一个节点包含了数据块和指向下一个节点的指针。



typedef struct node

{

int val;

struct node\*next;

}Node;

## 数据结构

LinkedList是基于链表结构实现，所以在LinkedList类中包含了first和last两个指针（类型为Node）。Node中包含了对prev节点、next节点的引用，这样就构成了双向的链表。

## 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;

## }

## }

## 存储

### **add(E e)方法**

该方法首先声明一个新Node l，将链表的最后一个Node赋值给l，然后将新的Node即newNode覆盖last（或者说让last指向newNode），最后将l的next指针指向newNode。

//Appends the specified element to the end of this list.

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++;

}

### 2.add(int index,E element)

该方法是在指定的index位置插入新元素。如果index位置正好等于size，则调用linkLast(element)将其插入到末尾；否则调用linkBefore(element,node(index))方法进行插入。

linkBefore中的node(index)是返回指定位置的元素。

public void add(int index,E element){

checkPositionIndex(index);

if(index==size)

linkLast(element);

else

linkBefore(element,node(index));

}

/\*\*

\*Returns the(non-null)Node at the specified element index.

\*/

Node<E>node(int index){

//assert isElementIndex(index);

//如果要取元素的位置是整个list一半的左半边，那么从list的头开始向后遍历，遍历至要取元素的位置

if(index<(size>>1)){

Node<E>x=first;

for(int i=0;i<index;i++)

x=x.next;

return x;

}

//否则从list一半的右半边开始寻找，也就是从尾部开始向前遍历，遍历至要取元素的位置

else{

Node<E>x=last;

for(int i=size-1;i>index;i--)

x=x.prev;

return x;

}

}

## 删除

### 1.remove(int index)

删除列表中指定位置的元素，首先检测该位置是否在该列表中存在，然后解除该元素前、后指向的元素。

public E remove(int index){

checkElementIndex(index);

return unlink(node(index));

}

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){

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;

}

## 获取

### 1.get(int index)

获取指定位置元素的值。同样首先判断传入位置是否越界，如果超过list的size，抛出IndexOutBoundsException异常，然后node()方法进行左半边或右半边遍历获取，add(int index,E element)中有提到，不再赘述。

public E get(int index){

checkElementIndex(index);

return node(index).item;

}

### 2.getFirst()

### 3.getLast()

first、last是LinkedList的两个属性，判空后直接返回。

/\*\*

\*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;

}

**六、元素添加**

下面说明双向链表添加元素add()的原理:

//首先传入一个将要添加的目标类型的元素（形参E e）

public boolean add(E e){

linkLast(e);

return true;

}

void linkLast(E e){

//1、设定一个常量Node类型l,并将当前表的最后一个元素付给l

final Node<E> l=last;

//2、创建一个newNode为将要添加元素，并将前后指针分别指向了l

//（未添加时链表最后一个元素）、null元素。

final Node<E> newNode=new Node<>(l,e,null);

//3、把newNode改变为链表末尾元素，原来的末尾元素成为前一个元素。

last=newNode;

//4、判断未添加是当前链表末尾元素，原来的末尾元素成为前一个元素

if(l==null)

first=newNode;

else

l.next=newNode;

size++;

modCount++;

}

add原理就是：

1. 首先获取链表最后一个节点。
2. 把新节点插入到最后一个节点之后。
3. linkedList的last属性重新指向最后一个节点。
4. 如果这个节点是第一个节点，之前没有节点，那么将linkedList的first的属 性指向新节点；如果不是，则将上一个节点的next属性指向该节点。