环形链表的约瑟夫问题：

n个小孩围成圈，丢手帕，从第start个小孩开始丢，每到第step个小孩出局；接着从下一个小孩开始，直到最后一个小孩为止，游戏结束。

/\*\*

\* @author Jiaozl

\* CycLink cyclink = new CycLink();

\* cyclink.setLen(5);

\* cyclink.createLink();

\* cyclink.setStart(2);

\* cyclink.setStep(2);

\* cyclink.show();

\* cyclink.play();

\*/

public class CycLink {

Child firstChild = null; // 指向链表第一个小孩的引用

Child temp=null;

int len = 0; // 表示公有几个小孩

int start = 0;

int step = 0;

/\*\*

\* 设置链表的大小

\*/

public void setLen(int len) {

this.len = len;

}

/\*\*

\* 设置从第几个人开始数数

\*/

public void setStart(int start) {

this.start = start;

}

/\*\*

\* 设置step

\*/

public void setStep(int step) {

this.step = step;

}

/\*\*

\* 开始play

\*/

public void play() {

Child temp = this.firstChild;

// 先找到开始数数的人

for(int i=1; i<=start-1; i++) {

temp = temp.nextChild;

}

while(this.len!=1) {

// 数m下

for (int i = 1; i < step; i++) {

temp = temp.nextChild;

}

Child temp2 = temp;

while (temp2.nextChild!=temp) {

temp2 = temp2.nextChild;

}

// 将疏导step的小孩，退出圈圈

temp2.nextChild = temp.nextChild;

temp = temp.nextChild;

this.len--;

}

System.out.println("最后剩下的小孩:" + temp.no);

}

/\*\*

\* 初始化环形链表

\*/

public void createLink() {

for (int i=1; i<=len; i++) {

if(i==1) {

Child ch = new Child(i);

this.firstChild = ch;

this.temp = ch;

} else {

if(i==len) {

Child ch = new Child(i);

temp.nextChild = ch;

temp = ch;

temp.nextChild = this.firstChild;

} else {

Child ch = new Child(i);

temp.nextChild = ch;

temp = ch;

}

}

}

}

/\*\*

\* 打印环形链表

\*/

public void show() {

Child temp = this.firstChild;

do{

System.out.print(temp.no + " ");

temp = temp.nextChild;

} while(temp!=this.firstChild);

}

}

class Child{

int no;

Child nextChild = null;

public Child(int no) {

this.no = no;

}

}

环形链表

给定一个链表，判断链表中是否有环。

**进阶：**  
你能否不使用额外空间解决此题？

**public boolean** hasCycle(ListNode head) {

**if** (head == **null** || head.**next** == **null**)

**return false**;

ListNode fast = head;

ListNode slow = head;

**while** (fast != **null** && fast.**next** != **null**) {

slow = slow.**next**;

fast = fast.**next**.**next**;

**if** (slow == fast)

**return true**;

}

**return false**;

}

**说明：**

这里有一个进阶的要求，不使用额外的空间，就是说 **只在该链表上操作**

（1）那么，我们先来看看如果**使用额外空间**可以怎么写？

如果链表有环，那么我们遍历的话，会出现无限重复，**不断的重复遍历环内的节点**，那么也就是我们**只要判断遍历是否发生节点重复**，即可**确定**链表是否有环。

说到重复，我们肯定会想到**Set**，set集合里的元素是不会发生重复的，add一个重复的元素，会返回false。

所以我们可以利用set的这个特性，**将遍历的链表节点逐一放进set，如果返回false，即发生重复，即链表有环** //这种解法的代码我就不贴了

**（2）**不使用额外空间的做法，这里我们使用到了[**Floyd判圈算法（龟兔赛跑算法）**](https://blog.csdn.net/xiaoquantouer/article/details/51620657)

这个算法可以**判断是否存在环，以及判断环的起点与长度的算法**。详细请点击上面的链接，自己理解。

**边界值判断：**

**head == null**// 空链表，无环

**head.next == null**// 只有一个节点的链表，无环

**变量注释：**

**fast**// 快指针，初始化为head头结点

**slow**// 慢指针，初始化为head头结点

**解题思路：**

**通过判断快慢指针是否相遇，判断是否有环。//如果有环，那么快指针一定能追上慢指针**

从前往后遍历，循环条件是 快指针能不能继续往下走，,fast != null || fast.next != null

**慢指针每次走一步，快指针每次走两步**

**如果fast == slow ，节点发生重复 ，链表有环 ，返回true**

**直到循环结束都没发生重复，链表有尽头(尾)， 返回false**

**Floyd算法简述**

Floyd判圈算法（Floyd Cycle Detection Algorithm），又称龟兔赛跑算法（Tortoise and Hare Algorithm），是一个可以在有限状态机、迭代函数或者链表上判断是否存在环，以及判断环的起点与长度的算法。

**二、基本思路**

在某种关系下，顶点 i 到 k 拓扑有序，顶点 k 到 j 也是相同的顺序，那么 i 和 j 也存在这个顺序。要是某一个顶点出现了自己到自己的环，那么图中就有环，但是这种方法复杂度高一些，没有检测顶点出度或者DFS的方法快，但是非常简单。

**三、问题**

如何检测一个链表是否有环，如果有，那么如何确定环的起点和环的长度。

**四、解决方案**

**（1）判断是否有环？**

龟兔解法的基本思想可以用我们跑步的例子来解释，如果两个人同时出发，如果赛道有环，那么快的一方总能追上慢的一方。进一步想，追上时快的一方肯定比慢的一方多跑了几圈，即多跑的路的长度是圈的长度的倍数。

基于上面的想法，Floyd用两个指针，一个慢指针（龟）每次前进一步，快指针（兔）指针每次前进两步（两步或多步效果时等价的，只要一个比另一个快就行）。如果两者在链表头以外的某一点相遇（即相等）了，那么说明链表有环，否则，如果（快指针）到达了链表的结尾，那么说明没坏。

**（2）求环的长度**

相遇的时候，一定已经在环上了，然后两个人只要再次在环上接着跑，再次相遇的时候（也就是所谓的套圈），跑的快的那个人就比跑的慢的人整整多跑了一圈，所以环的长度也就出来了。

**（3）如何确定环的起点**

环的检测用上述原理，接下来我们来看一下如何确定环的起点，这也是Floyd解法的第二部分。方法是将其中一个指针移到链表起点，两者同时移动，每次移动一步，那么两者相遇的地方就是环的起点。

解析：

首先假设第一次相遇的时候慢指针走过的节点个数为i，设链表头部到环的起点的长度为m，环的长度为n，相遇的位置与起点位置距离为k。

于是有：

i = m + a \* n + k

其中a为慢指针走的圈数。

因为快指针的速度是慢指针的2倍，于是又可以得到另一个式子：

2 \* i = m + b \* n + k

其中b为快指针走的圈数。

两式相减得：

i = ( b - a ) \* n

也就是说i是圈长的整数倍。

这是将其中一个节点放在起点，然后同时向前走m步时，此时从头部走的指针在m位置。而从相遇位置开始走的指针应该在距离起点i+m，i为圈长整数倍，则该指针也应该在距离起点为m的位置，即环的起点。

第1部分 LinkedList介绍

LinkedList简介

LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。  
LinkedList 实现 List 接口，能对它进行队列操作。  
LinkedList 实现 Deque 接口，即能将LinkedList当作双端队列使用。  
LinkedList 实现了Cloneable接口，即覆盖了函数clone()，能克隆。  
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。  
LinkedList 是非同步的。

 LinkedList构造函数

// 默认构造函数

LinkedList()

// 创建一个LinkedList，保护Collection中的全部元素。

LinkedList(Collection<? extends E> collection)

LinkedList的API

[复制代码](javascript:void(0);)

LinkedList的API

boolean add(E object)

void add(int location, E object)

boolean addAll(Collection<? extends E> collection)

boolean addAll(int location, Collection<? extends E> collection)

void addFirst(E object)

void addLast(E object)

void clear()

Object clone()

boolean contains(Object object)

Iterator<E> descendingIterator()

E element()

E get(int location)

E getFirst()

E getLast()

int indexOf(Object object)

int lastIndexOf(Object object)

ListIterator<E> listIterator(int location)

boolean offer(E o)

boolean offerFirst(E e)

boolean offerLast(E e)

E peek()

E peekFirst()

E peekLast()

E poll()

E pollFirst()

E pollLast()

E pop()

void push(E e)

E remove()

E remove(int location)

boolean remove(Object object)

E removeFirst()

boolean removeFirstOccurrence(Object o)

E removeLast()

boolean removeLastOccurrence(Object o)

E set(int location, E object)

int size()

<T> T[] toArray(T[] contents)

Object[] toArray()

[复制代码](javascript:void(0);)

AbstractSequentialList简介

在介绍LinkedList的源码之前，先介绍一下AbstractSequentialList。毕竟，LinkedList是AbstractSequentialList的子类。

AbstractSequentialList 实现了get(int index)、set(int index, E element)、add(int index, E element) 和 remove(int index)这些函数。**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。

第2部分 LinkedList数据结构

LinkedList的继承关系

[复制代码](javascript:void(0);)

java.lang.Object

↳ java.util.AbstractCollection<E> 抽象类

↳ java.util.AbstractList<E> 抽象类

↳ java.util.AbstractSequentialList<E> 抽象类

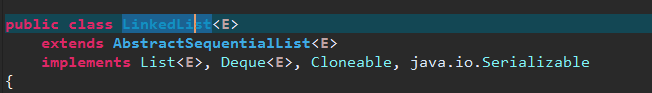
↳ java.util.LinkedList<E>

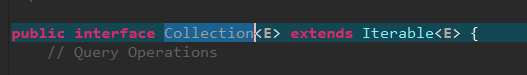
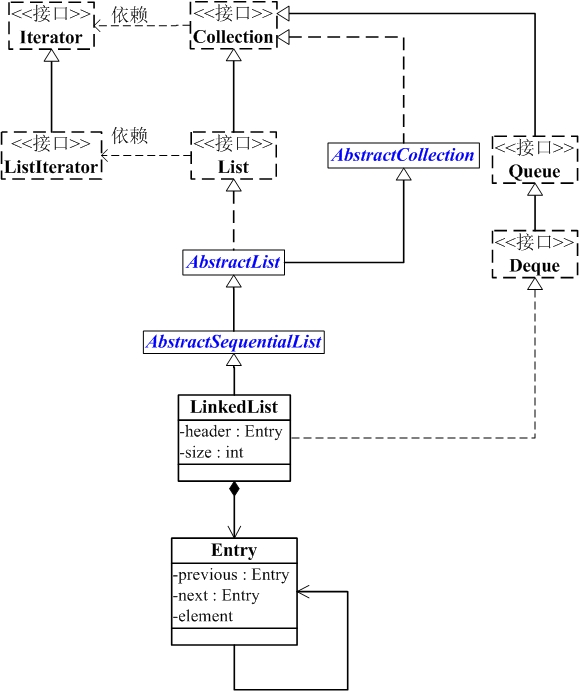
public class LinkedList<E>

extends AbstractSequentialList<E>

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

[复制代码](javascript:void(0);)

 **LinkedList与Collection关系如下图：**

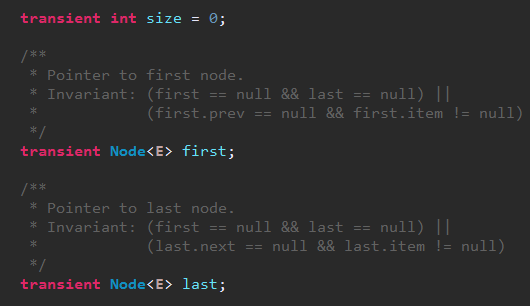
 [](https://images0.cnblogs.com/blog/497634/201401/272345393446232.jpg)

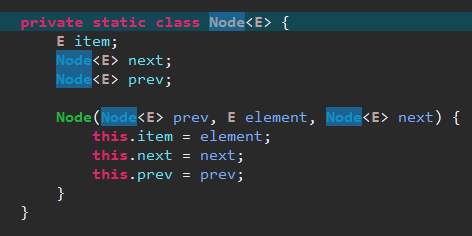
附注：iteratable **迭代出来的元素都是原来集合元素的拷贝**

**Java集合中保存的元素实质是对象的引用(可以理解为C中的指针)，而非对象本身。**

**迭代出的元素也就都是 引用的拷贝，结果还是引用**。那么，如果集合中保存的元素是可变类型的，我们就可以通过迭代出的元素修改原集合中的对象。

LinkedList的本质是双向链表。  
(01) LinkedList继承于AbstractSequentialList，并且实现了Dequeue接口。   
(02) LinkedList包含两个重要的成员：header （已经没有header了，为first）和 size。  
　　header是双向链表的表头，它是双向链表节点所对应的类Entry的实例。Entry中包含成员变量： previous, next, element。其中，previous是该节点的上一个节点，next是该节点的下一个节点，element是该节点所包含的值。   
　　size是双向链表中节点的个数。





第3部分 LinkedList源码解析(基于JDK1.8)

为了更了解LinkedList的原理，**下面对LinkedList源码代码作出分析**。

在阅读源码之前，我们先对LinkedList的整体实现进行大致说明：  
    LinkedList实际上是通过双向链表去实现的。既然是双向链表，那么它的**顺序访问会非常高效，而随机访问效率比较低**。  
    既然LinkedList是通过双向链表的，但是它也实现了List接口{*也就是说，它实现了get(int location)、remove(int location)等“根据****索引值****来获取、删除节点的函数”*}。LinkedList是如何实现List的这些接口的，如何将“**双向链表和索引值联系起来的**”？  
    实际原理非常简单，它就是通过一个**计数索引值**来实现的；**说明：在根据索引查找结点时，会有一个小优化，结点在前半段则从头开始遍历，在后半段则从尾开始遍历，这样就保证了只需要遍历最多一半结点就可以找到指定索引的结点。  
   这就是“双向链表和索引值联系起来”的方法。代码如下：**

**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;**

**}**

**}**

核心：有参构造函数，先调用无参函数，然后使用addall函数：

public boolean addAll(int index, Collection<? extends E> c) {

checkPositionIndex(index);

Object[] a = c.toArray();

int numNew = a.length;

if (numNew == 0)

return false;

Node<E> pred, succ;

if (index == size) {

succ = null;

pred = last;

} else {

succ = node(index);

pred = succ.prev;

}

for (Object o : a) {

@SuppressWarnings("unchecked") E e = (E) o;

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

if (pred == null)

first = newNode;

else

pred.next = newNode;

pred = newNode;

}

if (succ == null) {

last = pred;

} else {

pred.next = succ;

succ.prev = pred;

}

size += numNew;

modCount++;

return true;

}

对addAll函数的思考

　　在addAll函数中，传入一个集合参数和插入位置，然后将集合转化为数组，然后再遍历数组，挨个添加数组的元素，但是问题来了，为什么要先转化为数组再进行遍历，而不是直接遍历集合呢？从效果上两者是完全等价的，都可以达到遍历的效果。关于为什么要转化为数组的问题，我的思考如下：1. 如果直接遍历集合的话，那么在遍历过程中需要插入元素，在堆上分配内存空间，修改指针域，这个过程中就会一直占用着这个集合，考虑正确同步的话，其他线程只能一直等待。2. 如果转化为数组，只需要遍历集合，而遍历集合过程中不需要额外的操作，所以占用的时间相对是较短的，这样就利于其他线程尽快的使用这个集合。说白了，就是有利于提高多线程访问该集合的效率，尽可能短时间的阻塞。

**总结**：  
(01) LinkedList 实际上是通过双向链表去实现的。  
        它包含一个非常重要的内部类：**Entry**。Entry是**双向链表节点所对应的数据结构**，它包括的属性有：**当前节点所包含的值**，**上一个节点**，**下一个节点**。  
(02) 从LinkedList的实现方式中可以发现，它不存在LinkedList容量不足的问题。  
(03) LinkedList的克隆函数，即是将全部元素克隆到一个新的LinkedList对象中。

public Object clone() {

LinkedList<E> clone = superClone();

// Put clone into "virgin" state

clone.first = clone.last = null;

clone.size = 0;

clone.modCount = 0;

// Initialize clone with our elements

for (Node<E> x = first; x != null; x = x.next)

clone.add(x.item);

return clone;

}  
(04) LinkedList实现java.io.Serializable。当写入到输出流时，先写入“容量”，再依次写入“每一个节点保护的值”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

// Read in any hidden serialization magic

s.defaultReadObject();

// Read in size

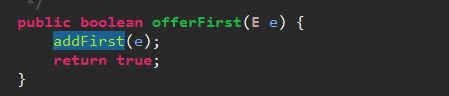
int size = s.readInt();

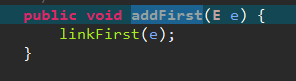
// Read in all elements in the proper order.

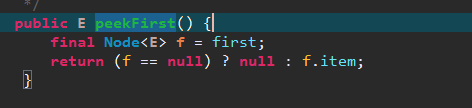
for (int i = 0; i < size; i++)

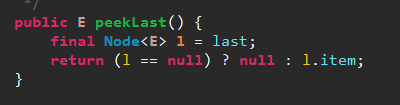
linkLast((E)s.readObject());

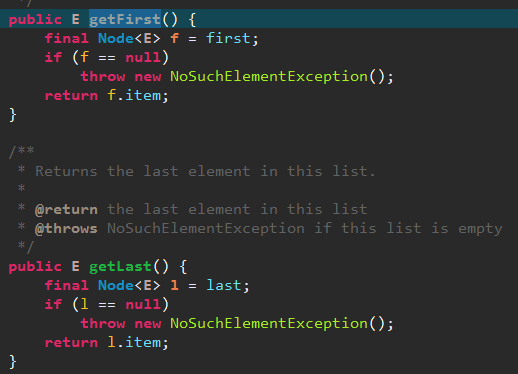
}  
(05) 由于LinkedList实现了Deque，而Deque接口定义了在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。每种方法都存在两种形式：一种形式在操作失败时抛出异常，另一种形式返回一个特殊值（具体取决于操作）。

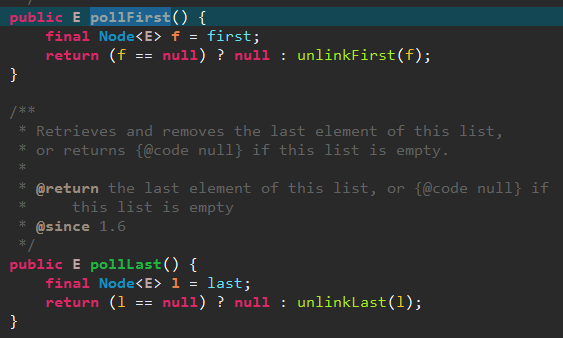


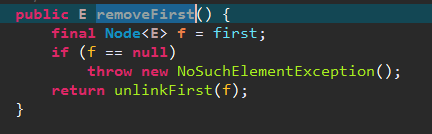


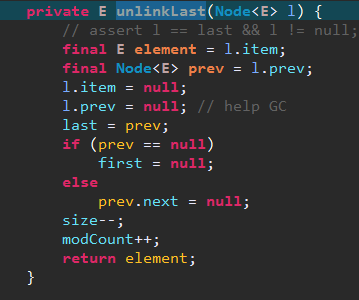












总结起来如下表格：

**第一个元素（头部） 最后一个元素（尾部）**

抛出异常 特殊值 抛出异常 特殊值

**插入** addFirst(e) offerFirst(e) addLast(e) offerLast(e)

**移除** removeFirst() pollFirst() removeLast() pollLast()

**检查** getFirst() peekFirst() getLast() peekLast()

(06) LinkedList可以作为**FIFO**(先进先出)的队列，作为FIFO的队列时，下表的方法等价：

[复制代码](javascript:void(0);)

**队列方法** **等效方法**

add(e) addLast(e)

offer(e) offerLast(e)

remove() removeFirst()

poll() pollFirst()

element() getFirst()

peek() peekFirst()

[复制代码](javascript:void(0);)

(07) LinkedList可以作为**LIFO**(后进先出)的栈，作为LIFO的栈时，下表的方法等价：

**栈方法 等效方法**

push(e) addFirst(e)

pop() removeFirst()

peek() peekFirst()

第4部分 LinkedList遍历方式

**LinkedList遍历方式**

LinkedList支持多种遍历方式。建议不要采用随机访问的方式去遍历LinkedList，而采用逐个遍历的方式。  
(01) 第一种，通过**迭代器**遍历。即通过Iterator去遍历。

for(Iterator iter = list.iterator(); iter.hasNext();)

iter.next();

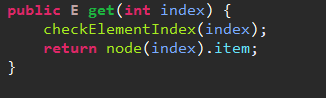
(02) 通过**快速随机**访问遍历LinkedList

int size = list.size();

for (int i=0; i<size; i++) {

list.get(i);

}



(03) 通过**另外一种for循环**来遍历LinkedList

for (Integer integ:list)

;

(04) 通过**pollFirst()**来遍历LinkedList

while(list.pollFirst() != null)

;

(05) 通过**pollLast()**来遍历LinkedList

while(list.pollLast() != null)

;

(06) 通过**removeFirst()**来遍历LinkedList

try {

while(list.removeFirst() != null)

;

} catch (NoSuchElementException e) {

}

(07) 通过**removeLast()**来遍历LinkedList

try {

while(list.removeLast() != null)

;

} catch (NoSuchElementException e) {

}

由此可见，遍历LinkedList时，使用removeFist()或removeLast()效率最高。但用它们遍历时，会删除原始数据；若单纯只读取，而不删除，应该使用第3种遍历方式。  
**无论如何，千万不要通过随机访问去遍历LinkedList！**