

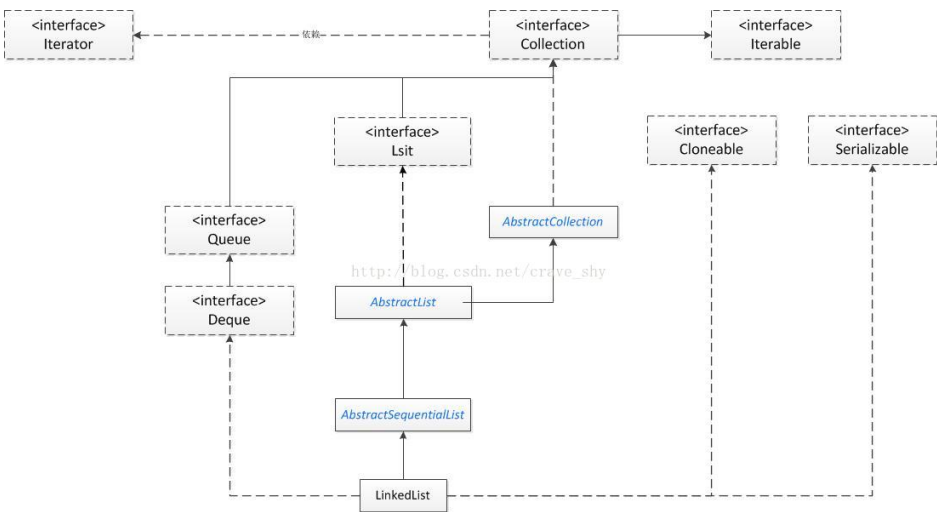
## java\_集合体系之:LinkedList详解、源码及示例——04

原创    2013年12月20日 15:11:00

6348    1    9    编辑 (<http://write.blog.csdn.net/postedit/{fileName}>)    删除

## java\_集合体系之:LinkedList详解、源码及示例——04

### 一：LinkedList结构图



简单说明：

- 1、上图中虚线且无依赖字样、说明是直接实现的接口
- 2、虚线但是有依赖字样、说明此类依赖与接口、但不是直接实现接口
- 3、实线是继承关系、类继承类、接口继承接口

### 二：LinkedList类简介

因为LinkedList内部是通过链表这种数据结构来实现数据存储和操作的、所以类简介分为两部分、第一部分是关于链表的简介（知道的可以无视）、第二部分是LinkedList类的简介。

#### 1、链表简介：

链表是常用的一种数据结构、关于链表这里只给出简单介绍、只是基础的概念。

##### a) 概念：

如果一个节点包含指向另一个节点的数据值，那么多个节点可以连接成一串，只通过一个变量访问整个节点序列，这样的节点序列称为链表（linked list）

##### b) 单向链表：

如果每个节点仅包含其指向后继节点的引用，这样的链表称为单向链表。

##### c) 双向链表：

链表节点，包含两个引用，一个指向前驱节点，一个指向后继节点，也就是——双向链表。

#### 2、LinkedList类简介：



Oscar Chen (<http://blog....>)

+ 关注

(<http://blog.csdn.net/chenghuaying>)

原创    粉丝    喜欢

182    14    0

- > CentOS 集群机器之间ssh免密  
(/crave\_shy/article/details/72964997)
- > JVM-内存管理-运行时数据区域  
(/crave\_shy/article/details/56675052)
- > JVM-Blog目录  
(/crave\_shy/article/details/56675032)
- > JVM-为什么要学JVM  
(/crave\_shy/article/details/56673439)

更多文章

(<http://blog.csdn.net/chenghuaying>)

在线课程



([http://edu.csdn.net/huiyiCourse/series\\_detail?utm\\_source=blog7](http://edu.csdn.net/huiyiCourse/series_detail?utm_source=blog7))

【直播】机器学习&数据挖掘7周实训--韦玮

([http://edu.csdn.net/huiyiCourse/series\\_detail/54?utm\\_source=blog7](http://edu.csdn.net/huiyiCourse/series_detail/54?utm_source=blog7))



([http://edu.csdn.net/combo/detail/471?utm\\_source=blog7](http://edu.csdn.net/combo/detail/471?utm_source=blog7))

【套餐】系统集成项目管理工程师顺利通关--徐朋

([http://edu.csdn.net/combo/detail/471?utm\\_source=blog7](http://edu.csdn.net/combo/detail/471?utm_source=blog7))

01 ) LinkedList以链表的形式存储数据、对增删元素有很高的效率、查询效率较低、尤其是随机访问、效率不忍直视、

02 ) LinkedList继承AbstractSequentialList ( 其继承与AbstractList、所以要求其子类要实现通过索引操作元素 )、使得LinkedList支持使用索引的“增删改查”操作、

03 ) LinkedList直接实现了List接口、使其可以内部存储元素有序并且为每个元素提供索引值、

04 ) LinkedList直接实现了Deque接口、Deque接口继承了Queue、使其可以作为双向链表这种数据结构来使用、操作元素、

05 ) LinkedList直接实现了Cloneable接口、使其可以复制其中的全部元素

06 ) 在使用ObjectOutputStream/ObjectInputStream流时、会先讲LinkedList的capacity读取/写入到流中、然后将元素——读取/写入。

## 三：LinkedList API

### 1、构造方法：

```
// 默认构造函数
LinkedList()
// 使用传入的Collection创建LinkedList。
LinkedList(Collection<? extends E> collection)
```

### 2、一般方法：

```
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()
```

## 四：LinkedList源码分析

```

package com.chy.collection.core;

import java.util.AbstractSequentialList;
import java.util.Collections;
import java.util.ConcurrentModificationException;
import java.util.Deque;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Queue;
import java.util.Vector;

/**
 * LinkedList实际上是通过双向链表去实现的、整个链表是同过Entry实体类来存储数据的
 */

public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    //链表的表头、表头不包含任何数据、
    //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。
    private transient Entry<E> header = new Entry<E>(null, null, null);

    // LinkedList中元素个数
    private transient int size = 0;

    /**
     * 构造一个空的LinkedList、只含有表头
     */
    public LinkedList() {
        header.next = header.previous = header;
    }

    /**
     * 创建一个包含c的LinkedList、先创建默认空、然后将c中所有元素添加到LinkedList中
     */
    public LinkedList(Collection<? extends E> c) {
        this();
        addAll(c);
    }

    /** 获取链表第一个元素、*/
    public E getFirst() {
        if (size==0)
            throw new NoSuchElementException();
        //因其是双向链表、这里的header可视为顺序的第一个不含元素的表头、所以第一个是此header的下一个元素
        return header.next.element;
    }

    /** 获取链表最后一个元素*/
    public E getLast() {
        if (size==0)
            throw new NoSuchElementException();
        //因其是双向链表、这里的header可视为逆序的第一个不含元素的表头、所以最后一个是此header的上一个元素
        return header.previous.element;
    }

    /** 删除LinkedList的第一个元素*/
    public E removeFirst() {
        return remove(header.next);
    }

    /** 删除LinkedList的最后一个元素*/
    public E removeLast() {
        return remove(header.previous);
    }

    /** 将元素e添加到LinkedList的起始位置*/
    public void addFirst(E e) {
        addBefore(e, header.next);
    }

    /** 将元素e添加到LinkedList的结束位置*/
    public void addLast(E e) {
        addBefore(e, header);
    }
}

```

```

/** 判断是否包含Object o*/
public boolean contains(Object o) {
    return indexOf(o) != -1;
}

/** 返回LinkedList的大小*/
public int size() {
    return size;
}

/** 将元素(E)添加到LinkedList中、添加到末尾*/
public boolean add(E e) {
    addBefore(e, header);
    return true;
}

/** 从LinkedList中删除o、如果存在则删除第一查找到的o并返回true、若不存在则返回false*/
public boolean remove(Object o) {
    if (o==null) {
        for (Entry<E> e = header.next; e != header; e = e.next) {
            if (e.element==null) {
                remove(e);
                return true;
            }
        }
    } else {
        for (Entry<E> e = header.next; e != header; e = e.next) {
            if (o.equals(e.element)) {
                remove(e);
                return true;
            }
        }
    }
    return false;
}

/** 将c中元素添加到双向链表LinkedList中、从尾部开始添加*/
public boolean addAll(Collection<? extends E> c) {
    return addAll(size, c);
}

/** 将c中元素添加到双向链表LinkedList中、所有元素添加到index与index+1表示的元素中间*/
public boolean addAll(int index, Collection<? extends E> c) {
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException("Index: "+index+ ", Size: "+size);
    //将c转换成数组、方便遍历元素和获取元素个数
    Object[] a = c.toArray();
    int numNew = a.length;
    if (numNew==0)
        return false;
    modCount++;

    //设置当前要插入节点的下一个节点
    Entry<E> successor = (index==size ? header : entry(index));
    //设置当前要插入节点的上一个节点
    Entry<E> predecessor = successor.previous;
    //将c中元素插入到LinkedList中
    for (int i=0; i<numNew; i++) {
        Entry<E> e = new Entry<E>((E)a[i], successor, predecessor);
        predecessor.next = e;
        predecessor = e;
    }
    successor.previous = predecessor;
    size += numNew;
    return true;
}

/** 删除LinkedList中所有元素*/
public void clear() {
    Entry<E> e = header.next;
    while (e != header) {
        Entry<E> next = e.next;
        e.next = e.previous = null;
        e.element = null;
        e = next;
    }
    header.next = header.previous = header;
    size = 0;
    modCount++;
}

```

```

// Positional Access Operations

/** 获取index处的元素*/
public E get(int index) {
    return entry(index).element;
}

/** 设置index处的元素、并将old元素返回*/
public E set(int index, E element) {
    Entry<E> e = entry(index);
    E oldVal = e.element;
    e.element = element;
    return oldVal;
}

/** 在index前添加节点, 且节点的值为element*/
public void add(int index, E element) {
    addBefore(element, (index==size ? header : entry(index)));
}

/** 删除index位置的节点*/
public E remove(int index) {
    return remove(entry(index));
}

/** 获取双向链表LinkedList中指定位置的节点、是LinkedList实现List中通过index操作元素的关键*/
private Entry<E> entry(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException("Index: "+index+ ", Size: "+size);
    Entry<E> e = header;
    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;
}

// Search Operations

/** 查询o所在LinkedList中的位置的索引、从前向后、不存在返回-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;
}

/** 查询o所在LinkedList中的位置的索引、从后向前、不存在返回-1*/
public int lastIndexOf(Object o) {
    int index = size;
    if (o==null) {
        for (Entry e = header.previous; e != header; e = e.previous) {
            index--;
            if (e.element==null)
                return index;
        }
    } else {
        for (Entry e = header.previous; e != header; e = e.previous) {
            index--;
            if (o.equals(e.element))
                return index;
        }
    }
}

```

```

        return -1;
    }

    // Queue operations.

    /** 返回第一个节点、若size为0则返回null*/
    public E peek() {
        if (size==0)
            return null;
        return getFirst();
    }

    /** 返回第一个节点、若size为0则抛异常NoSuchElementException*/
    public E element() {
        return getFirst();
    }

    /** 删除并返回第一个节点、若LinkedList的大小为0,则返回null*/
    public E poll() {
        if (size==0)
            return null;
        return removeFirst();
    }

    /** 删除第一个元素、若LinkedList的大小为0,则抛异常*/
    public E remove() {
        return removeFirst();
    }

    /** 将e添加双向链表末尾*/
    public boolean offer(E e) {
        return add(e);
    }

    // Deque operations
    /** 将e添加双向链表开头*/
    public boolean offerFirst(E e) {
        addFirst(e);
        return true;
    }

    /** 将e添加双向链表末尾*/
    public boolean offerLast(E e) {
        addLast(e);
        return true;
    }

    /**返回第一个节点、若LinkedList的大小为0,则返回null*/
    public E peekFirst() {
        if (size==0)
            return null;
        return getFirst();
    }

    /**返回最后一个节点、若LinkedList的大小为0,则返回null*/
    public E peekLast() {
        if (size==0)
            return null;
        return getLast();
    }

    /** 删除并返回第一个、若LinkedList的大小为0,则返回null*/
    public E pollFirst() {
        if (size==0)
            return null;
        return removeFirst();
    }

    /** 删除并返回最后一个、若LinkedList的大小为0,则返回null*/
    public E pollLast() {
        if (size==0)
            return null;
        return removeLast();
    }

    /** 将e插入到双向链表开头*/
    public void push(E e) {
        addFirst(e);
    }

```

```

/** 删除并返回第一个节点*/
public E pop() {
    return removeFirst();
}

/** 从LinkedList中删除o、如果存在则删除第一查找到的o并返回true、若不存在则返回false*/
public boolean removeFirstOccurrence(Object o) {
    return remove(o);
}

/** 从LinkedList末尾向前查找，删除第一个值为元素(o)的节点*/
public boolean removeLastOccurrence(Object o) {
    if (o==null) {
        for (Entry<E> e = header.previous; e != header; e = e.previous) {
            if (e.element==null) {
                remove(e);
                return true;
            }
        }
    } else {
        for (Entry<E> e = header.previous; e != header; e = e.previous) {
            if (o.equals(e.element)) {
                remove(e);
                return true;
            }
        }
    }
    return false;
}

/** 返回“index到末尾的全部节点”对应的ListIterator对象(List迭代器)*/
public ListIterator<E> listIterator(int index) {
    return new ListItr(index);
}

private class ListItr implements ListIterator<E> {
    // 上一次返回的节点
    private Entry<E> lastReturned = header;
    // 下一个节点
    private Entry<E> next;
    // 下一个节点对应的索引值
    private int nextIndex;
    // 期望的改变计数。用来实现fail-fast机制。
    private int expectedModCount = modCount;

    //构造函数、从index位置开始进行迭代
    ListItr(int index) {
        if (index < 0 || index > size)
            throw new IndexOutOfBoundsException("Index: "+index+ ", Size: "+size);
        /*
         * 若 “index 小于 ‘双向链表长度的一半’”，则从第一个元素开始往后查找；
         * 否则，从最后一个元素往前查找。
         */
        if (index < (size >> 1)) {
            next = header.next;
            for (nextIndex=0; nextIndex<index; nextIndex++)
                next = next.next;
        } else {
            next = header;
            for (nextIndex=size; nextIndex>index; nextIndex--)
                next = next.previous;
        }
    }
    // 是否存在下一个元素
    public boolean hasNext() {
        return nextIndex != size;
    }
    // 获取下一个元素
    public E next() {
        checkForComodification();
        if (nextIndex == size)
            throw new NoSuchElementException();

        lastReturned = next;
        next = next.next;
        nextIndex++;
        return lastReturned.element;
    }

    // 是否存在上一个元素

```

```

    public boolean hasPrevious() {
        return nextIndex != 0;
    }
    // 获取上一个元素
    public E previous() {
        if (nextIndex == 0)
            throw new NoSuchElementException();

        lastReturned = next = next.previous;
        nextIndex--;
        checkForComodification();
        return lastReturned.element;
    }

    // 获取下一个元素的索引
    public int nextIndex() {
        return nextIndex;
    }

    // 获取上一个元素的索引
    public int previousIndex() {
        return nextIndex-1;
    }
    // 删除双向链表中的当前节点
    public void remove() {
        checkForComodification();
        Entry<E> lastNext = lastReturned.next;
        try {
            LinkedList.this.remove(lastReturned);
        } catch (NoSuchElementException e) {
            throw new IllegalStateException();
        }
        if (next==lastReturned)
            next = lastNext;
        else
            nextIndex--;
        lastReturned = header;
        expectedModCount++;
    }
    // 设置当前节点为e
    public void set(E e) {
        if (lastReturned == header)
            throw new IllegalStateException();
        checkForComodification();
        lastReturned.element = e;
    }
    // 将e添加到当前节点的前面
    public void add(E e) {
        checkForComodification();
        lastReturned = header;
        addBefore(e, next);
        nextIndex++;
        expectedModCount++;
    }
    // 判断 “modCount和expectedModCount是否相等”，以此来实现fail-fast机制。
    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}

/**
 * 内部静态类、是双向链表的节点所对应的数据结构、
 * 此数据结构包含三部分：上一节点、下一节点、当前节点值
 */
private static class Entry<E> {
    //当前节点值
    E element;
    //下一节点
    Entry<E> next;
    //上一节点
    Entry<E> previous;

    /**
     * 链表节点构造函数
     * @param element    节点值
     * @param next       下一节点
     * @param previous    上一节点
     */
    Entry(E element, Entry<E> next, Entry<E> previous) {

```



```

        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}

//新建节点、节点值是e、将新建的节点添加到entry之前
private Entry<E> addBefore(E e, Entry<E> entry) {
    //觉得难理解的可以先花个几分钟看一下链式结构资料、最好是图片形式的
    //新建节点实体
    Entry<E> newEntry = new Entry<E>(e, entry, entry.previous);
    //将参照节点原来的上一个节点（即插在谁前面的）的下一个节点设置成newEntry
    newEntry.previous.next = newEntry;
    //将参照节点（即插在谁前面的）的前一个节点设置成newEntry
    newEntry.next.previous = newEntry;
    size++;
    modCount++;
    return newEntry;
}

//将节点从链表中删除、返回被删除的节点的内容
private E remove(Entry<E> e) {
    //如果是表头、抛异常
    if (e == header)
        throw new NoSuchElementException();

    E result = e.element;
    //下面实际上就是、将e拿掉、然后将e的上下两个节点连接起来
    e.previous.next = e.next;
    e.next.previous = e.previous;
    e.next = e.previous = null;
    e.element = null;
    size--;
    modCount++;
    return result;
}

/**
 * 反向迭代器
 * @since 1.6
 */
public Iterator<E> descendingIterator() {
    return new DescendingIterator();
}

/** 反向迭代器实现类 */
private class DescendingIterator implements Iterator {
    final ListItr itr = new ListItr(size());
    public boolean hasNext() {
        return itr.hasPrevious();
    }
    public E next() {
        return itr.previous();
    }
    public void remove() {
        itr.remove();
    }
}

/** 返回LinkedList的克隆对象*/
public Object clone() {
    LinkedList<E> clone = null;
    try {
        clone = (LinkedList<E>) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }

    // Put clone into "virgin" state
    clone.header = new Entry<E>(null, null, null);
    clone.header.next = clone.header.previous = clone.header;
    clone.size = 0;
    clone.modCount = 0;

    // Initialize clone with our elements
    for (Entry<E> e = header.next; e != header; e = e.next)
        clone.add(e.element);

    return clone;
}

```

```

}

/** 将LinkedList中的所有元素转换成Object[]中*/
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;
}

/** 将LinkedList中的所有元素转换成Object[]中、并且完成类型转换*/
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;
}

private static final long serialVersionUID = 876323262645176354L;

/** 将LinkedList的“容量，所有的元素值”都写入到输出流中
 * 1、将LinkedList的容量写入进去
 * 2、将LinkedList中的所有元素写入进去
 */
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out any hidden serialization magic
    s.defaultWriteObject();

    // Write out size
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (Entry e = header.next; e != header; e = e.next)
        s.writeObject(e.element);
}

/**
 * 将写入的LinkedList读取出来
 * 1、读取写入的LinkedList的容量
 * 2、读取写入的元素
 */
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();

    // Initialize header
    header = new Entry<E>(null, null, null);
    header.next = header.previous = header;

    // Read in all elements in the proper order.
    for (int i=0; i<size; i++)
        addBefore((E)s.readObject(), header);
}
}

```

总结：从源码中可以看出List是通过一个很关键的实体类Entry来实现链表式的存储的。

1、Entry、该类是一个实体类、用来表示链表中的一个节点、他包括连接上一个节点的引用、连接下一个节点的引用、和节点的属性。

2、LinkedList实现了List中要求的使用索引操作的一些方法、LinkedList中是通过private Entry<E> entry(int index)方法将LinkedList中的元素与索引关联起来的、他对于传进来的每个index都会与size进行比较、若index >= size/2 那么就后开始向前依次遍历LinkedList中元素、直到LinkedList中第index个元素被查找到、若inde < size/2、则从LinkedList开头依次遍历所有元素直到第index个元素被查找到并返回、这也是通过随机访问LinkedList效率非常低的原因。

3、由LinkedList的数据结构的特性、决定其方法很多、但是也是有规律可循、其中一部分的方法对相同结果有两种处理、一种是返回null、一种是抛出异常、这些方法如下

返回值:	异常	null	异常	null
	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
	removeFirst()	pollFirst()	removeLast()	pollLast()
	getFirst()	peekFirst()	getLast()	peekLast()

4、当LinkedList作为先进先出（FIFO）队列时、下列方法等效

队列方法	等效方法
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

5、当LinkedList作为先进后出（FILO）栈时、下列方法等效

栈方法	等效方法
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

## 五：LinkedList示例

因为使用集合、我们最关心的就是使用不同集合的不同方法的效率问题、而在这些中、最能体现效率问题的关键点是对集合的遍历、所以对于示例、分为两部分：第一部分是关于集合的不同的遍历方法的耗时示例、第二部分是集合的API的使用示例。

1、遍历方法：

01 ) 使用Iterator遍历LinkedList

```
Iterator<String> it = list.iterator();
while(it.hasNext()){
    it.next();
}
```

02 ) 使用ListIterator遍历LinkedList

```
ListIterator<String> listIt = list.listIterator();
while(listIt.hasNext()){
    listIt.next();
}
```

03 ) 使用随机访问（即for(int i=0; i<xxx; i++)这种形式称为随机访问）遍历LinkedList

```
for (int i = 0; i < list.size(); i++) {
    list.get(i);
}
```

04 ) 使用增强for循环遍历LinkedList

```
for(String i : list);
```

05 ) 使用removeFirst遍历

```
try {
    while(list.removeFirst() != null);
} catch (Exception e) {}
```

06 ) 使用removeLast遍历

```
try {
    while(list.removeLast() != null);
} catch (Exception e) {}
```

07 ) 使用pollFirst遍历

```
while(list2.pollFirst() != null);
```

08 ) 使用pollLast遍历

```
while(list3.pollLast() != null);
```

示例

```

package com.chy.collection.example;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.ListIterator;

public class ErgodicLinkedList {
    /**
     * 测试LinkedList不同遍历方式以及效率
     */
    public static void testErgodicLinkedList(){
        //初始化一个较大的list
        LinkedList<String> list = getBigLinkedList();

        //零: Iterator迭代器
        long start0 = currentTime();
        Iterator<String> it = list.iterator();
        while(it.hasNext()){
            it.next();
        }
        long end0 = currentTime();
        System.out.println("iterator use time : " + (end0-start0) + " ms");

        //一: ListIterator迭代器
        long start = currentTime();
        ListIterator<String> listIt = list.listIterator();
        while(listIt.hasNext()){
            listIt.next();
        }
        long end = currentTime();
        System.out.println("listIterator use time : " + (end-start) + " ms");

        //二: 随机访问
        long start1 = currentTime();
        for (int i = 0; i < list.size(); i++) {
            list.get(i);
        }
        long end1 = currentTime();
        System.out.println("RandomAccess use time : " + (end1-start1) + " ms"); //r
result: RandomAccess use time : 43891 ms

        //三: 使用增强for循环
        long start2 = currentTime();
        for(String i : list);
        long end2 = currentTime();
        System.out.println("Foreach use time : " + (end2-start2) + " ms");

        //四: 使用removeFirst
        long start3 = currentTime();
        try {
            while(list.removeFirst() != null);
        } catch (Exception e) {}
        long end3 = currentTime();
        System.out.println("removeFirst use time : " + (end3-start3) + " ms");

        //五: 使用removeLast
        LinkedList<String> list1 = getBigLinkedList();
        long start4 = currentTime();
        try {
            while(list1.removeLast() != null);
        } catch (Exception e) {}
        long end4 = currentTime();
        System.out.println("removeLast use time : " + (end4-start4) + " ms");

        //六: 使用pollFirst
        LinkedList<String> list2 = getBigLinkedList();
        long start5 = currentTime();
        while(list2.pollFirst() != null);
        long end5 = currentTime();
        System.out.println("pollFirst use time : " + (end5-start5) + " ms");

        //七: 使用pollLast
        LinkedList<String> list3 = getBigLinkedList();
        long start6 = currentTime();
        while(list3.pollLast() != null);
        long end6 = currentTime();
        System.out.println("pollLast use time : " + (end6-start6) + " ms");
    }
}

```

```

private static LinkedList<String> getBigLinkedList() {
    LinkedList<String> list = new LinkedList<String>();
    for (int i = 0; i < 100000; i++) {
        list.add("a");
    }
    return list;
}
private static long currentTime() {
    return System.currentTimeMillis();
}

public static void main(String[] args) {
    testErgodicLinkedList();
}
}

```

结果说明：

iterator use time : 16 ms

listIterator use time : 16 ms

RandomAccess use time : 7500 ms

Foreach use time : 15 ms

removeFirst use time : 16 ms

removeLast use time : 16 ms

pollFirst use time : 16 ms

pollLast use time : 16 ms

不能使用随机访问去遍历LinkedList！效率相差不可以道理记！后面四种虽然也可以遍历LinkedList、但是会清空LinkedList中数据、所以常用的可以使用迭代器和增强for循环来迭代LinkedList。

2、API演示

```

package com.chy.collection.example;

import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;

@SuppressWarnings("all")
public class LinkedListTest {

    //LinkedList中方法很多、但是没有一个是自己特有的、都是从抽象类或者接口中继承的、所以测试的
    时候就根据其继承或者实现的类、接口做为分类依据

    /**
     * 测试从AbstractSequentialList中继承的方法
     */
    private static void testLinkedListFromASL(){
        System.out.println("Test methods: add(), addAll(Collection<?> extends E c), get(int
index), set(int index, E e), remove(int index), clear(), clone());

        //将"a", "b"添加到list中
        LinkedList<String> list = new LinkedList<String>();
        list.add("a");
        list.add("b");
        printList(list);

        //将含有"c", "d"的集合list2添加到list中
        LinkedList<String> list2 = new LinkedList<String>();
        list2.add("c");
        list2.add("d");
        list.addAll(list.size(), list2);
        printList(list);

        //将list中的最后一个元素设置成第一个元素
        list.set(list.size()-1, list.get(0)); //list.set(index, e)、list.get(index)都不推
荐使用！

        printList(list);

        //将第一个"a"元素从list中移除
        list.remove(list.indexOf("a"));
        printList(list);

        //将list克隆一份
        LinkedList<String> cloneList = (LinkedList<String>) list.clone();
        printList(cloneList);

        //将list中元素清除
        list.clear();
        System.out.println("list elements : " + list + " list size : " + list.size());
    }

    /**
     * 测试从Deque中继承的方法、双向队列Deque又是从Queue中继承的、所以方法比较多
     * 但是有规律可循、LinkedList作为Deque的实现类、既可以作为先进先出的队列、也可以作为先进后
    出的堆栈

    */

    /**
     * 测试LinkedList以堆栈（后进先出、LIFO）形式使用的方法、

    */
    private static void testLinkedListASStack(){
        System.out.println("Test methods: push(E e), poll(), peek()");
        LinkedList<String> stack = new LinkedList<String>();
        stack.push("a");
        stack.push("b");
        stack.push("c");
        stack.push("d");
        printList(stack);

        /**
         * pop(), poll()都是取出stack栈顶元素 、区别就是当stack中没有元素时、stack.pop
        ()抛异常、stack.poll()返回null
         */
        printStr("取出stack栈顶元素 str : " + stack.pop());
        printStr("取出stack栈顶元素 str : " + stack.poll());
        printStr("查看stack栈顶元素 str : " + stack.peek());
        printList(stack);
    }

    /**

```

```

    * 测试LinkedList以队列的形式（先进先出FIFO）形式使用的方法、
    */
    private static void testLinkedListASQueue(){
        System.out.println("Test methods: add(e), offer(e), remove(), poll(), element(), peek()" );

        LinkedList<String> queue = new LinkedList<String>();
        //add(e) 与方法off(e)等效
        queue.add("a");
        queue.add("b");
        queue.add("c");
        queue.offer("d");
        queue.offer("e");
        printList(queue);

        //他移除的是队列最前端的元素、最先进去的元素
        printStr(queue.remove());
        printStr(queue.poll());

        /*
        * 下面两个方法都是查看第一个元素、区别就是当queue中没有元素时、queue.element()抛异常、queue.peek()返回null
        */
        printStr(queue.element());
        printList(queue);
        printStr(queue.peek());
        printList(queue);

        queue.clear();
        printList(queue);
        printStr("the result of peek : " + queue.peek()); //result: the result of peek : null
        printStr("the result of element : " + queue.element()); //result: java.util.NoSuchElementException
    }

    /**
    * 测试LinkedList以双向链表的使用方法
    */
    private static void testLinkedListDeque(){

        System.out.println("Test methods: addFirst(object), addLast(object), offerFirst(), offerLast(), getFirst(object), getLast(object), pollFirst(), pollLast(), peekFirst(), peekLast(), removeFirst(), removeLast()");

        //将“abcdefg”初始化到LinkedList中
        LinkedList<String> list = new LinkedList<String>();

        //addFirst/addLast 没有返回值、offerFirst/offerLast返回true、内部都是调用Entry.addBefore()来添加的
        list.addFirst("c");
        list.offerFirst("b");
        list.offerFirst("a");
        list.addLast("d");
        list.addLast("e");
        list.offerLast("f");
        printList(list);

        /*
        * 相同：
        *
        * 不同：
        *
        *
        */
        printStr("list getFirst() obtains element: " + list.getFirst());
        printStr("list peekFirst() obtains element: " + list.peekFirst());
        printStr("list getLast() obtains element: " + list.getLast());
        printStr("list peekLast() obtains element: " + list.peekLast());
        printList(list);

        /*
        * 相同：
        *
        * 不同：
        *
        *
        */
        printStr(list.pollFirst());

```

```

        printStr(list.removeFirst());
        printList(list);

        printStr(list.pollLast());
        printStr(list.removeLast());
        printList(list);
    }

    /**
     * 测试LinkedList与Array之间的转换
     */
    private static void testLinkedListOthersAPI(){
        String[] strArray = {"a", "b", "c", "d", "e"};
        LinkedList<String> strList = new LinkedList<String>(Arrays.asList(strArray));
        printList(strList);

        String[] array = strList.toArray(new String[0]);
        System.out.println(Arrays.toString(array));
    }

    private static void printList(LinkedList<String> list) {
        System.out.println(list);
    }
    private static void printStr(String str) {
        System.out.println(str);
    }
    public static void main(String[] args) {
        // testLinkedListFromASL();
        // testLinkedListASStack();
        // testLinkedListASQueue();
        // testLinkedListDeque();
        // testLinkedListOthersAPI();
        // testLinkedListOthersAPI();
    }
}

```

## 总结：

对于LinkedList、掌握其本质——LinkedList是以链表的数据结构形式来存储元素的、但是又因LinkedList实现了List接口、所以他要实现将索引与其内部元素关联起来、但是这些索引不是存储元素就生成的、而是根据传入的索引值来通过二分法依次查找得出的、对于每次调用哪些通过索引操作元素的方法、LinkedList都要从新查找指定索引处的元素、所以当使用随机访问时、效率非常低下！关于LinkedList与ArrayList区别（附带与数组的区别）、会在List结尾做总结。


更多内容：[java\\_集合体系之总体目录——00](http://blog.csdn.net/crave_shy/article/details/1741679)  
([http://blog.csdn.net/crave\\_shy/article/details/1741679](http://blog.csdn.net/crave_shy/article/details/1741679))

版权声明：本文为博主原创文章，未经博主允许不得转载。



标签：LinkedList (<http://so.csdn.net/so/search/s.do?q=LinkedList&t=blog>) /  
java集合 (<http://so.csdn.net/so/search/s.do?q=java集合&t=blog>) /  
迭代器 (<http://so.csdn.net/so/search/s.do?q=迭代器&t=blog>) /  
java (<http://so.csdn.net/so/search/s.do?q=java&t=blog>) /  
数据结构 (<http://so.csdn.net/so/search/s.do?q=数据结构&t=blog>) /

1条评论

 qq\_36596145 ([http://my.csdn.net/qq\\_36596145](http://my.csdn.net/qq_36596145))  
([http://my.csdn.net/qq\\_36596145](http://my.csdn.net/qq_36596145))



发表评论





u010850027 (/u010850027) 2016-12-07 11:28  
mark  
(/u010850027)  
回复

1楼

更多评论

## 相关文章推荐

### 【数据结构】LinkedList原理及实现学习总结 (/jianyuerensheng/article/details/51204598)

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 ...



jianyuerensheng 2016-04-20 22:06 6054

### 数据结构和算法-----LinkedList的实现 (/sinat\_33661267/article/details/53149766)

```
package com.example;import java.util.ConcurrentModificationException; import java.util.Iterator; imp...
```



sinat\_33661267 2016-11-13 16:36 524

### Fedora 25 - Linux 下 Qt5.9 的安装和配置 (/zhangk9509/article/details/73480491)

Qt安装Qt的安装有两种方法，一种是编译源码，一种是使用安装包。本着少折腾的原则，我选用后一种安装方式。在网上下载Qt的安装包，有online安装和offline安装。online的太慢了所以选择...



ZhangK9509 2017-06-20 09:45 613

### poj 1791 Heavy Transportation (/gakki\_wpt/article/details/76775572)

Background Hugo Heavy is happy. After the breakdown of the Cargolifter project he can now expand b...



Gakki\_wpt 2017-08-06 16:12 38

### 构造函数访问权限和基类构造函数，派生类的构造函数调用顺序 (/wmaiynij/article/details/75729435)

```
#include #include using namespace std; //1.访问权限 class B { public: void Fun1(int b1,int b2,int b3) ...
```



wMaiYniJ 2017-07-22 12:53 37

### BufferedReader的用法 (/j\_jorey/article/details/75258096)

public class BufferdReader extends Reader 从字符输入流中读取文本，缓冲各个字符，从而提供字符、数组和行的高效读取。通常，Reader 所作的每个读取...



J\_Jorey 2017-07-17 17:33 79

### 在Docker环境下部署Kafka (/snowcity1231/article/details/54946857)

1、下载镜像 这里使用了wurstmeister/kafka和wurstmeister/zookeeper这两个版本的镜像 docker pull wurstmeister/zookeeperdock...



snowcity1231 2017-02-09 14:50 2914

### navicat 提示“1045 access denied for user 'root'@'localhost' ”解决方法 (/liufuwu1/article/details/76794695)

切换到MySQL bin目录下, mysqld --skip\_grant\_tables; mysql -u root -p without password; denglu  
mysql>...



liufuwu1 2017-08-06 21:38 41

### java 企业网站源码模版 有前后台 springmvc SSM 生成静态化 (/q207055576q/article/details/56009142)

java 企业网站源码 前后台都有 静态模版引擎, 代码生成器大大提高开发效率 系统介绍: 1.网站后台采用主流的 SSM 框架 jsp JSTL, 网站后台采用freemaker静态化模版...



q207055576q 2017-02-20 10:48 696

### Java多态的应用 (/baqi\_zxh/article/details/53896166)

1.使用多态实现为宠物喂食需求说明: 实现如果所示的当宠物饿了(健康值介于0至100之间时), 主人需要为宠物喂食(健康值加3, 但上限不超过100), 为宠物增加健康值的输出效果。2.使用多态为...



BaQi\_zxh 2016-12-27 15:51 241

### java\_集合体系之Hashtable详解、源码及示例——10 (/crave\_shy/article/details/17583001)

摘要: 本文通过Hashtable的结构图来说明Hashtable的结构、以及所具有的功能。根据源码给出Hashtable所具有的特性、结合源码对其特性深入理解、给出示例体会使用方式。



chenghuaying 2013-12-26 15:29 1840

### Java\_io体系之PipedWriter、PipedReader简介、走进源码及示例——14 (http://810364804.iteye.com/blog/1992802)

Java\_io体系之PipedWriter、PipedReader简介、走进源码及示例——14 ——管道字符输出流、必须建立在管道输入流之上、所以先介绍管道字符输出流。可以先看示例或者总结、总结写的有点Q、不喜可无视、有误的地方指出则不胜感激。一: PipedWriter 1、类功能简介: 管道字符输出流、用于将当前线程的指定字符写入到与此线程对应的管道字符输入流



810364804 2013-12-08 18:50 57

### Java\_io体系之FilterWriter、FilterReader简介、走进源码及示例——15 (http://810364804.iteye.com/blog/1992801)

Java\_io体系之FilterWriter、FilterReader简介、走进源码及示例——15 一: FilterWriter 1、类功能简介: 字符过滤输出流、与FilterOutputStream功能一样、只是简单重写了父类的方法、目的是为所有装饰类提供标准和基本的方法、要求子类必须实现核心方法、和拥有自己的特色。这里FilterWriter没有子类、可能其意义只是提供一个接口、留着以后的扩展。。。本身是一个抽象类、如同Wr



810364804 2013-12-08 20:50 62

### java\_集合体系之HashMap详解、源码及示例——09 (/crave\_shy/article/details/17552679)



摘要: 本文通过HashMap的结构图分析HashMap所具有的特性、通过源码深入了解HashMap实现原理、使用方法、通过实例加深对HashMap的应用的理解。篇幅较长、慎入!



chenghuaying 2013-12-25 14:54 2541

### java\_集合体系之ArrayList详解、源码及示例——03 (<http://810364804.iteye.com/blog/1992789>)



java\_集合体系之ArrayList详解、源码及示例——03 一：ArrayList结构图 

 liuxinglanyue 2010-11-14 19:36  656

---

### Java\_io体系之PipedWriter、PipedReader简介、走进源码及示例——14 (/crave\_shy/article/details/17202977)



Java\_io体系之PipedWriter、PipedReader简介、走进源码及示例——14 ——管道字符输出流、必须建立在管道输入流之上、所以先介绍管道字符输出流。可...

 chenghuaying 2013-12-08 18:50  1473

---

### Java API——RMIIO入门教程（2）远程流传输示例之RMIIO服务源码 (http://xiaoruanjian.iteye.com/blog/1548554)

1. RMI Service Interface package com.sinosuperman.rmiio2; import java.rmi.Remote; import java.rmi.RemoteException; import com.healthmarketscience.rmiio.RemoteInputStream; public interface RmiiService2 extends Remote { pub

 wapysun 2011-12-01 17:16  330