

## LinkedList 使用

### 1、概述

- 与 ArrayList 一样，LinkedList 也按照索引位置排序，但它的元素之间是双向链接的
- 适合快速地插入和删除元素
- LinkedList 实现 List 和 Queue 两个接口

### 2、构造方法

方法名	说明
LinkedList()	构造一个空列表
LinkedList(Collection<? extends E> c)	构造一个包含指定 collection 中的元素的列表，这些元素按其 collection 的迭代器返回的顺序排列

### 3、常用方法

方法名	说明
boolean add(E e)	将指定元素添加到此列表的结尾
void add(int index, E element)	在此列表中指定的位置插入指定的元素
boolean addAll(Collection<? extends E> c)	添加指定 collection 中的所有元素到此列表的结尾
boolean addAll(int index, Collection<? extends E> c)	将指定 collection 中的所有元素从指定位置开始插入此列表
void addFirst(E e)	将指定元素插入此列表的开头
void addLast(E e)	将指定元素添加到此列表的结尾
void clear()	从此列表中移除所有元素

boolean contains(Object o)	如果此列表包含指定元素，则返回 true
E get(int index)	返回此列表中指定位置处的元素
E getFirst()	返回此列表的第一个元素
E getLast()	返回此列表的最后一个元素
int indexOf(Object o)	返回此列表中首次出现的指定元素的索引， 如果此列表中不包含该元素，则返回 -1
int lastIndexOf(Object o)	返回此列表中最后出现的指定元素的索引， 如果此列表中不包含该元素，则返回 -1
E peek()	获取但不移除此列表的头（第一个元素）
E peekFirst()	获取但不移除此列表的第一个元素；如果此列表为空，则返回 null
E peekLast()	获取但不移除此列表的最后一个元素；如果此列表为空，则返回 null
E poll()	获取并移除此列表的头（第一个元素）
E pollFirst()	获取并移除此列表的第一个元素；如果此列表为空，则返回 null
E pollLast()	获取并移除此列表的最后一个元素；如果此列表为空，则返回 null
E pop()	从此列表所表示的堆栈处弹出一个元素
void push(E e)	将元素推入此列表所表示的堆栈
E remove()	获取并移除此列表的头（第一个元素）
E remove(int index)	移除此列表中指定位置处的元素

boolean remove(Object o)	从此列表中移除首次出现的指定元素（如果存在）
E removeFirst()	移除并返回此列表的第一个元素
E set(int index, E element)	将此列表中指定位置的元素替换为指定的元素
int size()	返回此列表的元素数
Object[] toArray()	返回以适当顺序（从第一个元素到最后一个元素）包含此列表中所有元素的数组

#### 4、案例

（1）案例一：使用 LinkedList 对字符串进行管理

```
import java.util.LinkedList;

public class LindedListDemo1 {

    public static void main(String[] args) {

        LinkedList<String> list=new LinkedList<String>();

        //向链表添加数据
        list.add("apple");
        list.add("pear");

        //将数据添加到链表的开始
        list.addFirst("banana");

        //将数据添加到链表的末尾
        list.addLast("grape");
```

---

```
//在指定位置处添加数据，第一个参数为index值，从0开始

list.add(2, "orange");

//显示链表中的所有数据

System.out.println(list);

//判断列表中是否包含指定的元素，并输出相应的结果

boolean flag=list.contains("grape");

if(flag){

    System.out.println("grape找到了！");

}else{

    System.out.println("grape没找到！");

}

//返回index值为3的数据并输出

System.out.println("index值为3的数据为："+list.get(3));

//返回第一个元素

System.out.println("第一个元素为："+list.getFirst());

//返回最后一个元素

System.out.println("最后一个元素为："+list.getLast());

}

}
```

运行结果为：

---

```
[banana, apple, orange, pear, grape]
grape找到了！
index值为3的数据为：pear
第一个元素为：banana
最后一个元素为：grape
```

(2) 案例二：使用 LinkedList 对自定义类进行管理

Student 类：

```
package com.linkedlist;
```

```
public class Student {
```

```
    private String stuNum;
```

```
    private String stuName;
```

```
    private int age;
```

```
    public Student(String stuNum,String stuName,int age){
```

```
        this.stuNum=stuNum;
```

```
        this.stuName=stuName;
```

```
        this.age=age;
```

```
    }
```

```
    public String getStuNum() {
```

```
        return stuNum;
```

```
    }
```

```
    public void setStuNum(String stuNum) {
```

```
        this.stuNum = stuNum;
```

---

```
}

    public String getStuName() {

        return stuName;

    }

    public void setStuName(String stuName) {

        this.stuName = stuName;

    }

    public int getAge() {

        return age;

    }

    public void setAge(int age) {

        this.age = age;

    }

    @Override

    public String toString() {

        return "[学号：" + stuNum + ", 姓名：" + stuName + ", 年龄："

+ age + "]";

    }

}
```

LinkedListDemo2 类：

```
import java.util.LinkedList;

public class LinkedListDemo2 {
```

---

```
public static void main(String[] args) {

    LinkedList<Student> stuList=new LinkedList<Student>();

    Student stu1=new Student("001","Mike",18);

    Student stu2=new Student("002","Jack",20);

    Student stu3=new Student("003","Lucy",19);

    //将学生添加到链表，使用push完成

    //LinkedList实现List接口的同时，也实现了Queue接口

    //push和pop就是针对Queue进行添加和取出数据的操作的

    stuList.push(stu1);

    stuList.push(stu2);

    stuList.push(stu3);

    System.out.println("链表为：" + stuList);

    //弹出一个元素，这里可以把链表看成一个容器，先加入到链表的数据

    后弹出，

    //依据的原则是先进后出

    System.out.println("弹出的数据为：" + stuList.pop());

    System.out.println("调用pop()方法后的链表为：" + stuList);

    //peek()方法获取并不移除元素

    System.out.println("调用peek()方法的数据为：" + stuList.peek());

    System.out.println("调用peek()方法后的链表为：" + stuList);

    //再次调用pop()方法，发现调用pop()方法后数据从链表中移除了，而

    peek()方法不会
```

```

System.out.println("再次调用pop()方法"+stuList.pop());

System.out.println("再次调用pop()方法后的链表为 :\n"+stuList);

//在链表中再重新添加元素

stuList.push(stu2);

stuList.push(stu3);

System.out.println("再次添加元素后的链表为 :\n"+stuList);

//调用poll()方法

System.out.println("调用poll()方法输出元素"+stuList.poll());

//调用poll()方法将获取元素的同时从链表中删除了元素

System.out.println("调用poll()方法后的链表为 :\n"+stuList);

}

}

```

运行结果为：

```

链表为：[[学号：003, 姓名：Lucy, 年龄：19], [学号：002, 姓名：Jack, 年龄：20], [学号：001, 姓名：Mike, 年龄：18]]
弹出的数据为：[学号：003, 姓名：Lucy, 年龄：19]
调用pop()方法后的链表为：
[[学号：002, 姓名：Jack, 年龄：20], [学号：001, 姓名：Mike, 年龄：18]]
调用peek()方法的数据为：[学号：002, 姓名：Jack, 年龄：20]
调用peek()方法后的链表为：
[[学号：002, 姓名：Jack, 年龄：20], [学号：001, 姓名：Mike, 年龄：18]]
再次调用pop()方法 [学号：002, 姓名：Jack, 年龄：20]
再次调用pop()方法后的链表为：
[[学号：001, 姓名：Mike, 年龄：18]]
再次添加元素后的链表为：
[[学号：003, 姓名：Lucy, 年龄：19], [学号：002, 姓名：Jack, 年龄：20], [学号：001, 姓名：Mike, 年龄：18]]
调用poll()方法输出元素 [学号：003, 姓名：Lucy, 年龄：19]
调用poll()方法后的链表为：
[[学号：002, 姓名：Jack, 年龄：20], [学号：001, 姓名：Mike, 年龄：18]]

```