

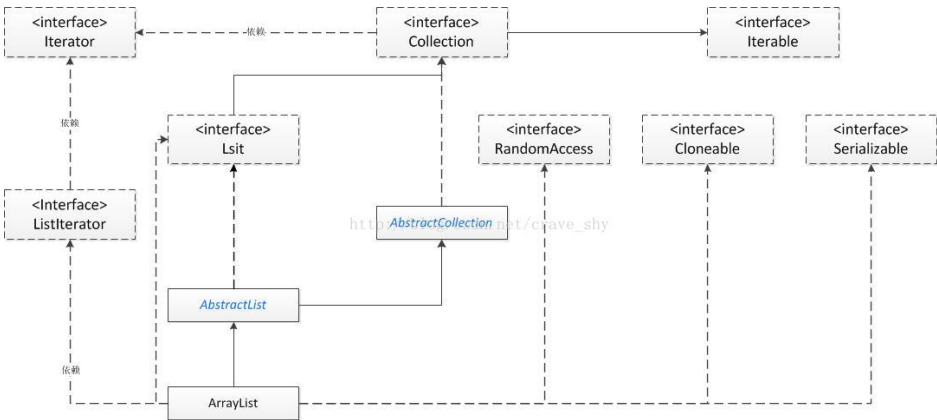
java_集合体系之ArrayList详解、源码及示例——03

原创 2013年12月20日 10:56:11

7063 2 6 收藏 编辑 (<http://write.blog.csdn.net/postedit/{fileName}>) 删除

java_集合体系之ArrayList详解、源码及示例——03

一：ArrayList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：ArrayList类简介：

- 1、ArrayList是内部是以动态数组的形式来存储数据的、知道数组的可能会疑惑：数组不是定长的吗？这里的动态数组不是意味着去改变原有内部生成的数组的长度、而是保留原有数组的引用、将其指向新生成的数组对象、这样会造成数组的长度可变的假象。
- 2、ArrayList具有数组所具有的特性、通过索引支持随机访问、所以通过随机访问ArrayList中的元素效率非常高、但是执行插入、删除时效率比较地下、具体原因后面有分析。
- 3、ArrayList实现了AbstractList抽象类、List接口、所以其更具有了AbstractList和List的功能、前面我们知道AbstractList内部已经实现了获取Iterator和ListIterator的方法、所以ArrayList只需关心对数组操作的方法的实现、
- 4、ArrayList实现了RandomAccess接口、此接口只有声明、没有方法体、表示ArrayList支持随机访问。
- 5、ArrayList实现了Cloneable接口、此接口只有声明、没有方法体、表示ArrayList支持克隆。
- 6、ArrayList实现了Serializable接口、此接口只有声明、没有方法体、表示ArrayList支持序列化、即将ArrayList以流的形式通过ObjectInputStream/ObjectOutputStream来写/读。



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

+ 关注

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

原创 182 粉丝 14 喜欢 2

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

【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/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)

三：ArrayList API

```
// Collection中定义的API
boolean      add(E object)
boolean      addAll(Collection<? extends E> collection)
void         clear()
boolean      contains(Object object)
boolean      containsAll(Collection<?> collection)
boolean      equals(Object object)
int          hashCode()
boolean      isEmpty()
Iterator<E>  iterator()
boolean      remove(Object object)
boolean      removeAll(Collection<?> collection)
boolean      retainAll(Collection<?> collection)
int          size()
<T> T[]      toArray(T[] array)
Object[]     toArray()
// AbstractList中定义的API
void         add(int location, E object)
boolean      addAll(int location, Collection<? extends E> collection)
E            get(int location)
int          indexOf(Object object)
int          lastIndexOf(Object object)
ListIterator<E> listIterator(int location)
ListIterator<E> listIterator()
E            remove(int location)
E            set(int location, E object)
List<E>      sublist(int start, int end)
// ArrayList新增的API
Object       clone()
void         ensureCapacity(int minimumCapacity)
void         trimToSize()
void         removeRange(int fromIndex, int toIndex)
```

总结：相对与AbstractCollection而言、多实现了List中新增的通过索引操作元素的方法。

四：ArrayList源码分析

```

package com.chy.collection.core;

import java.util.Arrays;
import java.util.ConcurrentModificationException;
import java.util.RandomAccess;
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    private static final long serialVersionUID = 8683452581122892189L;

    /** 保存ArrayList中元素的数组*/
    private transient Object[] elementData;

    /** 保存ArrayList中元素的数组的容量、即数组的size*/
    private int size;

    /** 使用指定的大小创建ArrayList*/
    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
        this.elementData = new Object[initialCapacity];
    }

    /** 使用默认的大小创建ArrayList*/
    public ArrayList() {
        this(10);
    }

    /**
     * 使用指定的Collection构造ArrayList、构造之后的ArrayList中包含Collection中的元素、
     * 这些元素的排序方式是按照ArrayList的Iterator返回他们时候的顺序排序的
     */
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        size = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    }

    /**
     * 将此 ArrayList 实例的容量调整为列表的当前大小
     */
    public void trimToSize() {
        //此集合总共被修改的次数
        modCount++;
        int oldCapacity = elementData.length;
        if (size < oldCapacity) {
            elementData = Arrays.copyOf(elementData, size);
        }
    }

    /**
     * 确保此ArrayList的最小容量能容纳下参数minCapacity指定的容量、
     * 1、minCapacity大于原来容量、则将原来的容量增加(oldCapacity * 3)/2 + 1;
     * 2、若minCapacity仍然大于增加后的容量、则使用minCapacity作为ArrayList容量
     * 3、若minCapacity不大于增加后的容量、则使用增加后的容量。
     */
    public void ensureCapacity(int minCapacity) {
        modCount++;
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            int newCapacity = (oldCapacity * 3)/2 + 1;
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
            // minCapacity is usually close to size, so this is a win:
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }

    /** 返回此列表中的元素的个数*/
    public int size() {
        return size;
    }

    /** 如果此列表中没有元素，则返回 true*/
    public boolean isEmpty() {
        return size == 0;
    }
}

```

```

    }

    /** 如果此列表中包含指定的元素，则返回 true。*/
    public boolean contains(Object o) {
        return indexOf(o) >= 0;
    }

    /** 返回指定对象在ArrayList中存放的第一个位置索引、注意空值的处理和Object.equals(? extends Object o)的返回值、不存在的话返回-1*/
    public int indexOf(Object o) {
        if (o == null) {
            for (int i = 0; i < size; i++)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = 0; i < size; i++)
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }

    /** 返回指定对象在ArrayList中存放最后一个位置的索引、注意空值的处理和Object.equals(? extends Object o)的返回值、不存在的话返回-1*/
    public int lastIndexOf(Object o) {
        if (o == null) {
            for (int i = size-1; i >= 0; i--)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = size-1; i >= 0; i--)
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }

    /** 返回一个当前集合的浅clone对象*/
    public Object clone() {
        try {
            ArrayList<E> v = (ArrayList<E>) super.clone();
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError();
        }
    }

    /** 将当前ArrayList转换成Object数组、注意操作使用此方法转换后的数组有可能抛异常*/
    public Object[] toArray() {
        return Arrays.copyOf(elementData, size);
    }

    /**
     * 将当前ArrayList转换成与传入的T类型相同的数组、当传入的a的length小于ArrayList的size的时候、方法内部会生成一个新的T[]返回
     * 如果传入的T[]的length大于ArrayList的size、则T[]从下标size开始到最后的元素都自动用null填充。
     */
    public <T> T[] toArray(T[] a) {
        if (a.length < size)
            // Make a new array of a's runtime type, but my contents:
            return (T[]) Arrays.copyOf(elementData, size, a.getClass());
        System.arraycopy(elementData, 0, a, 0, size);
        if (a.length > size)
            a[size] = null;
        return a;
    }

    // Positional Access Operations

    /** 获取ArrayList中索引为index位置的元素*/
    public E get(int index) {
        RangeCheck(index);

        return (E) elementData[index];
    }

    /** 将ArrayList的索引为index处的元素使用指定的E元素替换、返回被替换的原来的元素值*/

```

```

public E set(int index, E element) {
    RangeCheck(index);

    E oldValue = (E) elementData[index];
    elementData[index] = element;
    return oldValue;
}

/** 将指定元素E添加到ArrayList的结尾处*/
public boolean add(E e) {
    //确保ArrayList的容量能够添加新的元素
    ensureCapacity(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}

/** 将指定元素添加到指定的索引处 、
 * 注意：
 * 1、如果指定的index大于Object[] 的size或者小于0、则抛IndexOutOfBoundsException
 * 2、检测Object[]是否需要扩容
 * 3、将从index开始到最后的元素后移一个位置、
 * 4、将新添加的元素添加到index去。
 */
public void add(int index, E element) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(
            "Index: "+index+", Size: "+size);

    ensureCapacity(size+1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
        size - index);
    elementData[index] = element;
    size++;
}

/** 与add类似、
 * 1、将指定index处的元素删除、
 * 2、将index之后的所有元素前一个位置、最后一个
 * 3、将最后一个元素设置为null、--size
 * 返回被删除的元素。
 */
public E remove(int index) {
    RangeCheck(index);

    modCount++;
    E oldValue = (E) elementData[index];

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
            numMoved);
    elementData[--size] = null; // Let gc do its work

    return oldValue;
}

/** 删除Object[]中指定的元素Object 类似与contains方法与remove的结合体、只不过这里使用的是fastRemove方法去移除指定元素、移除成功则返回true*/
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
    }
    return false;
}

/* 删除指定索引处的元素、不返回被删除的元素*/
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;

```

```

        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                               numMoved);
        elementData[--size] = null; // Let gc do its work
    }

    /** 清空ArrayList*/
    public void clear() {
        modCount++;

        // Let gc do its work
        for (int i = 0; i < size; i++)
            elementData[i] = null;
        size = 0;
    }

    /** 将指定集合中的所有元素追加到ArrayList中（从最后开始追加）*/
    public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacity(size + numNew); // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
        return numNew != 0;
    }

    /** 将指定集合中的所有元素插入到index开始的后面位置处、原有的元素往后排*/
    public boolean addAll(int index, Collection<? extends E> c) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(
                "Index: " + index + ", Size: " + size);

        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacity(size + numNew); // Increments modCount

        int numMoved = size - index;
        if (numMoved > 0)
            System.arraycopy(elementData, index, elementData, index + numNew,
                               numMoved);

        System.arraycopy(a, 0, elementData, index, numNew);
        size += numNew;
        return numNew != 0;
    }

    /** 移除列表中索引在 fromIndex（包括）和 toIndex（不包括）之间的所有元素。
     * 1、将Object[] 从toIndex开始之后的元素（包括toIndex处的元素）移到Object[]下标从fromIndex开始
     之后的位置
     * 2、若有Object[]尾部要有剩余的位置则用null填充
     */
    protected void removeRange(int fromIndex, int toIndex) {
        modCount++;
        int numMoved = size - toIndex;
        System.arraycopy(elementData, toIndex, elementData, fromIndex,
                           numMoved);

        // Let gc do its work
        int newSize = size - (toIndex-fromIndex);
        while (size != newSize)
            elementData[--size] = null;
    }

    /** 检测下标是否越界*/
    private void RangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(
                "Index: "+index+", Size: "+size);
    }

    /** 将此ArrayList写入到ObjectOutputStream流中、先写ArrayList存放元素的Object[]长度、再将Object
    []中的每个元素写入到ObjectOutputStream流中*/
    private void writeObject(java.io.ObjectOutputStream s) throws java.io.IOException{
        // Write out element count, and any hidden stuff
        int expectedModCount = modCount;
        s.defaultWriteObject();

        // Write out array length
        s.writeInt(elementData.length);
    }

```

```

        // Write out all elements in the proper order.
        for (int i=0; i<size; i++)
            s.writeObject(elementData[i]);

        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }

    /** 从ObjectInputStream中读取ArrayList、先读取ArrayList中Object[]的长度、再读取每个元素放入Object
    []中对应的位置*/
    private void readObject(java.io.ObjectInputStream s) throws java.io.IOException, ClassNotFoundException {
        // Read in size, and any hidden stuff
        s.defaultReadObject();

        // Read in array length and allocate array
        int arrayLength = s.readInt();
        Object[] a = elementData = new Object[arrayLength];

        // Read in all elements in the proper order.
        for (int i=0; i<size; i++)
            a[i] = s.readObject();
    }
}

```

总结：从ArrayList源码可以看出、ArrayList内部是通过动态数组来存储数据、从中我们也可以很容易的找到ArrayList的几个特性：

- 1、有序：如果不指定元素存放位置、则元素将依次从Object数组的第一个位置开始放、如果指定插入位置、则会将元素插入指定位置、后面的所有元素都后移
- 2、可重复：从源码中没有看到对存放的元素的校验
- 3、随机访问效率高：可以直接通过索引定位到我们要找的元素
- 4、自动扩容：ensureCapacity(int minCapacity)方法中会确保数组的最小size、当不够时会原来的容量扩增到： $(oldCapacity * 3) / 2 + 1$ 。
- 5、变动数组元素个数（即添加、删除数组元素）效率低、在增删的操作中我们常见的一个函数：System.arraycopy()、他是将删除、或者添加之后、原有的元素进行移位、这是需要较大代价的。
- 6、ArrayList不是线程安全的、即当使用多线程操作ArrayList时会有可能出错、后面总结会有。

五：ArrayList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ArrayList

```

for(Iterator<Integer> iter = list.iterator(); iter.hasNext(); ) {
    iter.next();
}

```

02) 使用ListIterator遍历ArrayList

```

for(Iterator<Integer> iter = list.listIterator(); iter.hasNext(); ) {
    iter.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ArrayList

```

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

```

04) 使用增强for循环遍历ArrayList

```
for(@SuppressWarnings("unused") int i : list);
```

05) 示例

```
package com.chy.collection.example;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;

public class EragodicArrayList {

    /**
     * 测试不同遍历方式的效率
     */
    public static void testObtainAllElements(){
        //初始化一个较大的ArrayList
        ArrayList<Integer> list = new ArrayList<Integer>();
        for(int i=0; i<2000000; i++){
            list.add(i);
        }
        //零：使用Iterator
        long start = startTime();
        for(Iterator<Integer> iter = list.iterator(); iter.hasNext(); ) {
            iter.next();
        }
        endTime(start);          //result: 63ms

        //一：使用Iterator
        long start0 = startTime();
        for(Iterator<Integer> iter = list.listIterator(); iter.hasNext(); ) {
            iter.next();
        }
        endTime(start0);          //result: 78ms

        //二：使用随机访问、通过索引
        long start1 = startTime();
        for (int i = 0; i < list.size(); i++){
            list.get(i);
        }
        endTime(start1);          //result: 16ms

        //三：使用增强for循环
        long start2 = startTime();
        for(@SuppressWarnings("unused") int i : list);
        endTime(start2);          //result:62ms

        //四：使用ListIterator
        long start3 = startTime();
        ListIterator<Integer> li = list.listIterator(0);
        while(li.hasNext()){
            li.next();
        }
        endTime(start3);          //result: 63ms
    }

    private static void endTime(long start) {
        long end = startTime();
        System.out.println(end - start + " ms");
    }

    private static long startTime() {
        long start = System.currentTimeMillis();
        return start;
    }

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

结果及说明：

63 ms

78 ms

15 ms

63 ms

62 ms

从上面可以看出：使用随机访问效率最高、其他的差不多。

2、API演示

```

package com.chy.collection.example;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.ListIterator;

import com.chy.collection.bean.Student;

public class ArrayListTest {

    /**
     * 测试ArrayList的添加元素方法、以及与size有关的方法
     */
    public static void testArrayListSize(){
        //use default object array's size 10
        ArrayList<String> list = new ArrayList<String>();
        list.add("a");
        list.add(1, "b");

        // use specified size 4
        ArrayList<String> list2 = new ArrayList<String>(4);
        list2.add("c");
        list2.add("d");
        list2.add("e");
        list2.add("f");

        //use specified size 5
        ArrayList<String> list3 = new ArrayList<String>(5);
        list3.add("g");
        list3.add("h");
        list3.add("i");
        list3.add("j");
        list3.add("k");
        list.addAll(list2);//从list末尾开始追加
        System.out.println(list.size());// result: 6
        list.addAll(6, list3);//从list索引6开始添加
        System.out.println(list.size());// result: 11
        //see AbstractCollection.toString();
        System.out.println(list);          //result: [a, b, c, d, e, f, g, h, i, j, k]

        // 对于ArrayList的大小、我们可以使用三个方法来操作
        list.add(null);
        list.add(null);
        System.out.println(list.size());
        list.trimToSize();//将list的大小设置成与其包含的元素相同、null也算是list中的元素、
并且可以重复出现
        System.out.println(list.size());
        list.ensureCapacity(1);//确保list的大小不小于传入的参数值。
        System.out.println(list.size());
        System.out.println(list.size());

    }

    /**
     * 测试ArrayList的包含、删除方法
     */
    public static void testArrayListContainsRemove(){

        //初始化包含学号从1到10的十个学生的ArrayList
        ArrayList<Student> list1 = new ArrayList<Student>();
        Student s1 = new Student(1,"chy1");
        Student s2 = new Student(2,"chy2");
        Student s3 = new Student(3,"chy3");
        Student s4 = new Student(4,"chy4");
        list1.add(s1);
        list1.add(s2);
        list1.add(s3);
        list1.add(s4);
        for (int i = 5; i < 11; i++) {
            list1.add(new Student(i, "chy" + i));
        }
        System.out.println(list1);

        //初始化包含学号从1到4的四个学生的ArrayList
        ArrayList<Student> list2 = new ArrayList<Student>();
        list2.add(s1);
        list2.add(s2);
        list2.add(s3);
        list2.add(s4);
    }
}

```

```

        //查看list1中是否包含学号为1的学生（ 这里要注意、ArrayList中存放的都是对象的引用、
        而非堆内存中的对象）
        System.out.println(list1.contains(s1));

        //查看list1中是否包含list2
        System.out.println(list1.containsAll(list2));

        //从新构造一个指向学号为1的student、查看list2是否包含、不包含就添加进去、在判断list
1是否包含list2
        Student newS1 = new Student(1, "newchy1");
        System.out.println("list2 contains newS1 ? " + list2.contains(newS1));

        if(!list2.contains(newS1)){
            list2.add(newS1);
        }
        System.out.println("list2 members : " + list2.toString());
        System.out.println("list1 contains list2 ? " + list1.containsAll(list2));

        //删除list1中索引为0的学生
        System.out.println(list1.remove(0));
        //如果学号为1的学生存在则删除、不存在删除学号为2的学生
        if(!list1.remove(s1)){
            System.out.println(list1.remove(s2));
        }
        //删除list2中的学生
        list1.removeAll(list2);
        System.out.println(list1);

        //清空list1
        list1.clear();

        //求list1与list2中元素的交集
        list1.retainAll(list2);
        System.out.println(list1);
    }

    /**
     * 测试ArrayList的获取元素方法、
     */
    public static void testObtainArrayListElements(){
        //将字符串数组转化成ArrayList
        String[] strArray = {"a", "b", "c",
"d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"};

        /*
        使用时会抛异常、是由于Arrays.asList(strArray)返回的是一个Object[]、不能强转成ArrayL
ist<String>类型
        ArrayList<String> list2 = (ArrayList<String>)Arrays.asList(strArray);
        System.out.println(list2);
        */

        //一般情况下使用下面这种转换方式、他会自动的将数组转换之后的类型设置为runtime时的类
型
        ArrayList<String> list1 = new ArrayList<String>(Arrays.asList(strArray));
        System.out.println(list1);

        //获取某个索引处的元素
        System.out.println("str " + list1.get(0) + " size: " + list1.size());

        //将最后一个元素设置成"a"、打印被替换的元素
        System.out.println("old element : " + list1.set(list1.size()-1, list1.get(0)) + " li
st elements: " + list1);
        System.out.println();

        //返回第一个、最后一个"a"、“w”、“z”的索引、不存在则返回-1、内部是根据ListIterator来
返回索引的
        System.out.println("first index of a : " + list1.indexOf("a") + " last index of a
:" + list1.lastIndexOf("a"));
        System.out.println("first index of w : " + list1.indexOf("w") + " last index of w
:" + list1.lastIndexOf("w"));
        System.out.println("first index of z : " + list1.indexOf("z") + " last index of z
:" + list1.lastIndexOf("z"));
    }

    /**
     * 对ListIterator方法的测试
     */

```

```

public static void testListIterator(){
    String[] strArray = {"a", "b", "c", "d", "e"};
    ArrayList<String> list = new ArrayList<String>(Arrays.asList(strArray));

    //倒序遍历list
    ListIterator<String> li = list.listIterator(list.size());
    while(li.hasPrevious()){
        System.out.println(li.previous());
    }
    System.out.println("=====");

    //以获取index方式、正序遍历list
    ListIterator<String> li1 = list.listIterator(0);
    while(li1.hasNext()){
        //System.out.println(li1.nextIndex());          会造成死循环、具体可以看源
        //System.out.println(li1.previousIndex());        同样会造成死循环、
        String s = li1.next();

        if("d".equals(s)){
            li1.set("a");
        }

        if("e".equals(s)){
            li1.add("f");
        }

        if("b".equals(s)){
            li1.remove();
        }
    }
    System.out.println(list);

    //对于在遍历过程中想获取index、要注意死循环、和字节想要获取的方式、具体可以自己动手
    ListIterator<String> li2 = list.listIterator();
    while(li2.hasNext()){
        li2.next();
        System.out.println(li2.nextIndex() + "=====" + li2.previousIndex());
    }
}

/**
 * 测试ArrayList转换成Array时注意事项、附Array转换成List
 */
public static void testArrayList2Array(){
    //关于Array转换成ArrayList上面已经有过介绍、现在再补充一点特殊情况
    int[] intArray = new int[10];
    for (int i = 0; i < intArray.length; i++) {
        intArray[i] = i;
    }

    //将上面的数组转化成ArrayList
    //ArrayList<int> list = Arrays.asList(intArray); 这种写法编译就会报错、因为集合的定义中、只能存放对象（其实是对象的引用）、所以我们要使用包装类型Integer

    //要先将上面的数组转换成Integer类型数组、只能手动转、不能强制或者自动转换、若有的话
    Integer[] integerArray = new Integer[intArray.length];
    for (int i = 0; i < intArray.length; i++) {
        integerArray[i] = intArray[i];
    }

    //ArrayList<Integer> list = (ArrayList<Integer>)Arrays.asList(integerArray);
    //System.out.println(list.get(0)); 会报错、原因上面有

    //通常使用下面的转换方式
    ArrayList<Integer> normalList = new ArrayList<Integer>(Arrays.asList(integerArra
y));

    System.out.println(normalList.get(0));

    //第一种
    /**
     * 会报强制转换错误、
     */
    //ArrayList转换成Array
    Integer[] itg = (Integer[])normalList.toArray();
    System.out.println(itg[0]);
    */

```

码

具体可以看源码

试试

定义中、只能存放对象（其实是对象的引用）、所以我们要使用包装类型Integer

望贴出来啊

```

        //第二种
        Integer[] ia = new Integer[normallist.size()];
        normallist.toArray(ia);
        System.out.println(ia[0]);

        //第三种、应该使用这种形式的定义、传入的参数本质是供toArray内部调用其类型、对其size简单处理一下、如果size大于list的size、则后面的补null、如果小于、则使用新的数组替换传入的、并作为结果返回

        Integer[] ia2 = normallist.toArray(new Integer[11]);
        System.out.println(ia2[10]);

    }

    /**
     * 测试fail-fast机制
     */
    public static void testFailFast(){
        String[] s = {"a", "b", "c", "d", "e"};
        ArrayList<String> strList = new ArrayList<String>(Arrays.asList(s));
        Iterator<String> it = strList.iterator();
        while(it.hasNext()){
            String str = it.next();
            System.out.println(str);
            //这里本来是多线程动了ArrayList中的元素造成的、现在仅仅是模拟一种情况、就是在迭代的过程中、另一个线程向ArrayList中添加一个元素造成的fail-fast
            //异常信息: java.util.ConcurrentModificationException
            if("d".equals(str)){
                strList.add(str);
            }
        }
    }

    public static void main(String[] args) {
        // testArrayListSize();
        // testArrayListContainsRemove();
        // testObtainArrayListElements();
        // testArrayList2Array();
        // testFailFast();
        // testListIterator();
    }
}

```

总结：

对于ArrayList、在记住其特性、有序可重复、便与查找、不便于增删的同时最好是能知道为什么他会有这些特性、其实源码就是最好的说明书、平常所接触的东西都是别人在源码的基础上分析得出的结论、只有自己的才是最适合自己的、别人总结的再好、看过、受教了、但是还是希望自己能动手总结一份、再差也是自己总结的、慢慢改进、只有自己的东西才是最适合自己的！


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)

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：ArrayList (<http://so.csdn.net/so/search/s.do?q=ArrayList&t=blog>) /
java集合框架图 (<http://so.csdn.net/so/search/s.do?q=java集合框架图&t=blog>) /
源码 (<http://so.csdn.net/so/search/s.do?q=源码&t=blog>) /
Iterator (<http://so.csdn.net/so/search/s.do?q=Iterator&t=blog>) /
ListIterator (<http://so.csdn.net/so/search/s.do?q=ListIterator&t=blog>) /

2条评论

 qq_36596145 (http://my.csdn.net/qq_36596145)
(http://my.csdn.net/qq_36596145)

[发表评论](#)

erjinzhilwd (/erjinzhilwd) 2017-02-13 16:04

2楼

ListIterator继承Iterator，不是依赖。~~感谢楼主
(/erjinzhilwd)
回复



ragrok (/ragrok) 2016-09-28 19:49

1楼

写的很好，集合部分是Josh Bloch 经典之作。
(/ragrok)
回复

[更多评论](#)

相关文章推荐

数据结构学习笔记之一:链表 (/fenglibing/article/details/669362)

原贴作者BOLG:http://blog.csdn.net/woolceo/许多人都知道链表(C语言)都是借助指针来实现链接的,同样许多人也知道java语言是没有指针的,所以很多人可能很理所当然的认为...



fenglibing 2006-04-19 15:43 12766

Java链表 (/xiangsuixinsheng/article/details/6537505)

使用Java实现链表，首先定义链表的数据结构，也就是定义一个类，LinkedListNode。这个定义了链表的节点，链表节点分两部分，数据info和链接link。public class Linked...



xiangsuixinsheng 2011-06-11 00:36 5665

java链表ListNode (/qq_17525769/article/details/53915042)

/** * 描述：删除链表中等于给定值val的所有节点。样例：给出链表 1->2->3->3->4->5->3, 和 val = 3, 你需要返回删除3之后的链表：1->2->4->5。分析：1....



qq_17525769 2016-12-28 22:52 4720

java自定义List链表 (/jdhanhua/article/details/6596395)

第一步：定义一个List接口，规定一些基本操作0001 package my.list; 0002 0003 public interface MyList extends Object { 000...



jdhanhua 2011-07-10 20:45 6499

双向链表listnode (/zclongembedded/article/details/8922185)

在init源代码中双向链表listnode被使用地很多。android源代码中定义了结构体listnode，奇怪的是，这个结构体只有用于链接节点的prev和next指针，却没有任何和“数据”有关的成员...



zclongembedded 2013-05-13 19:35 3276

Delete Node in a Linked List Java LeetCode (/zuoyexingchennn/article/details/47168331)

Write a function to delete a node (except the tail) in a singly linked list, given only access to th...



zuoyexingchennn 2015-07-31 12:21 776

java 把一个list中的数据按照树结构排序 (/liuxiao723846/article/details/41862495)

import java.util.ArrayList; import java.util.List; public class HList { static List list = new Arr...



liuxiao723846 2014-12-11 09:36 1328

NodeJS 对于 Java 开发者而言是什么? (/robertsong2004/article/details/53967409)

我们都知道Node.js现在得到了所有的关注。每个人都对学习Node.js感兴趣，并希望可以工作于Node.js。在开始工作之前了解技术背后的概念总是不会错的。但对初学者来说，可能会因为不同的人使用的...



robertsong2004 2017-01-01 18:07 749

Java集合类ArrayList实现细节 (/studyfordream2015/article/details/70461365)

第1部分 ArrayList介绍 ArrayList简介 ArrayList 是一个数组队列，相当于 动态数组。与Java中的数组相比，它的容量能动态增长。它继承于AbstractList，实现了...



StudyForDream2015 2017-04-22 16:40 118

Java集合ArrayList实现类的总结 (/wenzhi20102321/article/details/52490738)

本文对java集合的ArrayList的实现类做了详细描述，ArrayList的使用方法，ArrayList的增删改查操作。泛型的使用，迭代器Iterator的使用等等知识点



wenzhi20102321 2016-09-09 21:44 630

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_集合体系之总体目录——00 (http://810364804.iteye.com/blog/1992787)

java_集合体系之总体目录——00 java_集合体系之总体框架——01 <a target="_blank" href="http://blog.csdn.net/crave_shy



810364804 2013-12-19 15:41 66

java_集合体系之:LinkedList详解、源码及示例——04 (/crave_shy/article/details/17440835)



摘要： 本文通过对LinkedList内部存储数据的结构、LinkedList的结构图、示例、源码、多方面深入分析LinkedList的特性和使用方法。



chenghuaying 2013-12-20 15:11 6348

文章收录1 (<http://444878909.iteye.com/blog/1951392>)

3.Hive Metastore 代码简析 <td width="760" class=



 444878909 2013-08-02 15:52  987

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 — : ArrayList结构图 <img



src="http://img.blog.csdn.net/20131220102938781?

watermark/2/text/aHR0cDovL2Jsbn2cuY3Nkbi5uZXQvY3JhdmVfc2h5/font/5a6L5L2T/fontsize/400/ fill/10JBQkFCMA==/dissolve/

 810364804 2013-12-20 10:56  163

java_集合体系之总体目录——00 ([/crave_shy/article/details/17416791](http://crave_shy/article/details/17416791))

摘要： java集合系列目录、不断更新中、、、水平有限、总有不足、误解的地方、请多包涵、也希望多提意见、多多讨论 ^_^

 chenghuaying 2013-12-19 15:41  3210



Android界面特殊全汇总 (<http://yuanlanjun.iteye.com/blog/1616453>)

(一) Activity 页面切换的效果 Android 2.0 之后有了 overridePendingTransition () , 其中里面两个参数 , 一个是前一个 activity 的退出两一个 activity 的进入 , Java 代码 1. @Override public void onCreate(Bundle savedInstanceState) { 2. super.onCreate(savedInstanceState); 3. 4

 yuanlanjun 2012-04-04 11:12  1483

java_集合体系之Vector详解、源码及示例——05 ([/crave_shy/article/details/17504279](http://crave_shy/article/details/17504279))

摘要： 本文通过对Vector的结构图中涉及到的类、接口来说明Vector的特性、通过源码来深入了解Vector各种功能的实现原理、通过示例加深对Vector的理解。

 chenghuaying 2013-12-23 14:40  2128

Java_io体系之OutputStreamWriter、InputStreamReader简介、走进源码及示例——17 (<http://810364804.iteye.com/blog/1992797>)



Java_io体系之OutputStreamWriter、InputStreamReader简介、走进源码及示例——17 — :

OutputStreamWriter 1、类功能简介： 输入字符转换流、是输入字节流转向输入字符流的桥梁、用于将输入字节流转换成输入

 810364804 2013-12-10 09:51  60

Java线程池 ([/nanmuling/article/details/37881089](http://nanmuling/article/details/37881089))

Java线程池 线程池编程 java.util.concurrent多线程框架---线程池编程 (一) 一般的服务器都需要线程池, 比如 Web、FTP等服务器, 不过它们一般都自己实现了线程池, 比如以...

 nanmuling 2014-07-16 16:44  2850