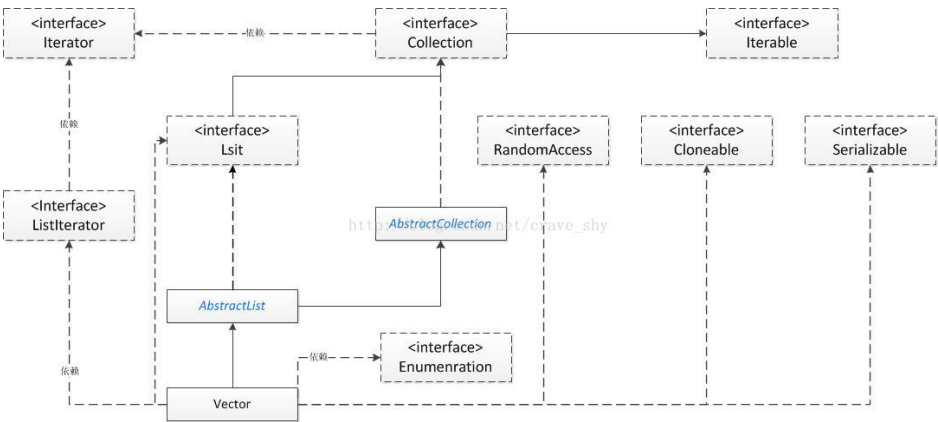


java_集合体系之Vector详解、源码及示例——05

原创 2013年12月23日 14:40:54 2128 0 6

java_集合体系之Vector详解、源码及示例——05

一：Vector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：Vector类简介：

- 1、Vector是内部是以动态数组的形式来存储数据的。
- 2、Vector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Vector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、Vector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Vector只需关心对数组操作的方法的实现、
- 4、Vector实现了RandomAccess接口、此接口只有声明、没有方法体、表示Vector支持随机访问。
- 5、Vector实现了Cloneable接口、此接口只有声明、没有方法体、表示Vector支持克隆。
- 6、Vector实现了Serializable接口、此接口只有声明、没有方法体、表示Vector支持序列化、即可以将Vector以流的形式通过ObjectOutputStream来写入到流中。
- 7、Vector是线程安全的。

三：Vector API

- 1、构造方法



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

+ 关注

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

原创 粉丝 喜欢
182 14 1

- > 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)

```

Vector()           // 默认构造函数

Vector(int capacity)    // capacity是Vector的默认容量大小。当由于增加数据导致容量增加时，每次容量会增加一倍。

Vector(int capacity, int capacityIncrement)    // capacity是Vector的默认容量大小，capacityIncrement是每次Vector容量增加时的增量值。

Vector(Collection<? extends E> collection)    // 创建一个包含collection的Vector

```

2、一般方法

```

synchronized boolean    add(E object)
void                    add(int location, E object)
synchronized boolean    addAll(Collection<? extends E> collection)
synchronized boolean    addAll(int location, Collection<? extends E> collection)
synchronized void        addElement(E object)
synchronized int         capacity()
void                    clear()
synchronized Object      clone()
boolean                 contains(Object object)
synchronized boolean    containsAll(Collection<?> collection)
synchronized void        copyInto(Object[] elements)
synchronized E           elementAt(int location)
Enumeration<E>          elements()
synchronized void        ensureCapacity(int minimumCapacity)
synchronized boolean     equals(Object object)
synchronized E           firstElement()
E                       get(int location)
synchronized int         hashCode()
synchronized int         indexOf(Object object, int location)
int                     indexOf(Object object)
synchronized void        insertElementAt(E object, int location)
synchronized boolean     isEmpty()
synchronized E           lastElement()
synchronized int         lastIndexOf(Object object, int location)
synchronized int         lastIndexOf(Object object)
synchronized E           remove(int location)
boolean                 remove(Object object)
synchronized boolean     removeAll(Collection<?> collection)
synchronized void        removeAllElements()
synchronized boolean     removeElement(Object object)
synchronized void        removeElementAt(int location)
synchronized boolean     retainAll(Collection<?> collection)
synchronized E           set(int location, E object)
synchronized void        setElementAt(E object, int location)
synchronized void        setSize(int length)
synchronized int         size()
synchronized List<E>     subList(int start, int end)
synchronized <T> T[]     toArray(T[] contents)
synchronized Object[]    toArray()
synchronized String      toString()
synchronized void        trimToSize()

```

总结：相对与ArrayList而言、Vector是线程安全的、即他的有安全隐患的方法都使用了synchronized关键字、Vector中多定义一个构造方法、用于指定当Vector自动扩容时的增量大小、Vector是个很老的类、他其中的许多方法都不是必须的、比如addElement(E e)、setElement(E, int)其实完全可以用add(E e)、set(E, int)取代、所以Vector的API源码显的比较臃肿、基本现在已经不再推荐使用Vector了、可以使用经过处理后的ArrayList来代替多线程环境中的Vector。至于如何处理会在List总结中有说到。

四：Vector源码分析

```

package com.chy.collection.core;

import java.util.Arrays;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.RandomAccess;

/** Vector: 矢量集合、*/
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    /** 保存Vector中元素的数组*/
    protected Object[] elementData;

    /** 保存Vector中元素的数组的容量、即数组的size*/
    protected int elementCount;

    /** 每次Vector自动扩容的增量*/
    protected int capacityIncrement;

    /** 默认版本号*/
    private static final long serialVersionUID = -2767605614048989439L;

    /** 使用指定的Vector容量和每次扩容的增量创建Vector*/
    public Vector(int initialCapacity, int capacityIncrement) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                             initialCapacity);
        this.elementData = new Object[initialCapacity];
        this.capacityIncrement = capacityIncrement;
    }

    /** 使用指定的Vector容量创建Vector*/
    public Vector(int initialCapacity) {
        this(initialCapacity, 0);
    }

    /** 使用默认的Vector容量创建Vector*/
    public Vector() {
        this(10);
    }

    /** 使用指定的Collection创建Vector*/
    public Vector(Collection<? extends E> c) {
        elementData = c.toArray();
        elementCount = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, elementCount, Object[].class);
    }

    /** 将Vector中的元素copy到传入的数组中*/
    public synchronized void copyInto(Object[] anArray) {
        System.arraycopy(elementData, 0, anArray, 0, elementCount);
    }

    /** 将Vector的size与Vector中元素同步*/
    public synchronized void trimToSize() {
        modCount++;
        int oldCapacity = elementData.length;
        if (elementCount < oldCapacity) {
            elementData = Arrays.copyOf(elementData, elementCount);
        }
    }

    /** 确保Vector的capacity最小不小于minCapacity*/
    public synchronized void ensureCapacity(int minCapacity) {
        modCount++;
        ensureCapacityHelper(minCapacity);
    }

    /** 确保Vector的capacity最小不小于minCapacity*/
    private void ensureCapacityHelper(int minCapacity) {
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object[] oldData = elementData;

```

```

        int newCapacity = (capacityIncrement > 0) ?
            (oldCapacity + capacityIncrement) : (oldCapacity * 2);
        if (newCapacity < minCapacity) {
            newCapacity = minCapacity;
        }
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}

/** 修改Vector的size、
 * 1、若传入的newSize > Vector中元素的个数、则将Vector的size修改成newSize、
 * 2、否则将Vector索引从newSize开始后面的元素都设置成null、并且将Vector的size修改成newSize
 */
public synchronized void setSize(int newSize) {
    modCount++;
    if (newSize > elementCount) {
        ensureCapacityHelper(newSize);
    } else {
        for (int i = newSize ; i < elementCount ; i++) {
            elementData[i] = null;
        }
    }
    elementCount = newSize;
}

/** 查看Vector的容量*/
public synchronized int capacity() {
    return elementData.length;
}

/** 查看Vector的size*/
public synchronized int size() {
    return elementCount;
}

/** 查看Vector是否为空*/
public synchronized boolean isEmpty() {
    return elementCount == 0;
}

/** 返回一个包含Vector中所有元素的Enumeration、Enumeration提供用于遍历Vector中所有元素的方法、
 * 相对与Iterator、ListIterator而言他不是fail-fast机制
 */
public Enumeration<E> elements() {
    return new Enumeration<E>() {
        int count = 0;

        public boolean hasMoreElements() {
            return count < elementCount;
        }

        public E nextElement() {
            synchronized (Vector.this) {
                if (count < elementCount) {
                    return (E)elementData[count++];
                }
            }
            throw new NoSuchElementException("Vector Enumeration");
        }
    };
}

/** 查看Vector是否包含o*/
public boolean contains(Object o) {
    return indexOf(o, 0) >= 0;
}

/** 返回o所在的索引*/
public int indexOf(Object o) {
    return indexOf(o, 0);
}

/** 从index处向后搜索o所在的索引值、没有则返回-1*/
public synchronized int indexOf(Object o, int index) {
    if (o == null) {
        for (int i = index ; i < elementCount ; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = index ; i < elementCount ; i++)

```

```

        if (o.equals(elementData[i]))
            return i;
    }
    return -1;
}

/** 从后向前查找o所在索引值*/
public synchronized int lastIndexOf(Object o) {
    return lastIndexOf(o, elementCount-1);
}

/** 从尾部index处向前查找o所在索引值、没有则返回-1*/
public synchronized int lastIndexOf(Object o, int index) {
    if (index >= elementCount)
        throw new IndexOutOfBoundsException(index + " >= " + elementCount);

    if (o == null) {
        for (int i = index; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = index; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

/** 返回index处的元素*/
public synchronized E elementAt(int index) {
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
    }
    return (E)elementData[index];
}

/** 返回第一个元素*/
public synchronized E firstElement() {
    if (elementCount == 0) {
        throw new NoSuchElementException();
    }
    return (E)elementData[0];
}

/** 返回最后一个元素*/
public synchronized E lastElement() {
    if (elementCount == 0) {
        throw new NoSuchElementException();
    }
    return (E)elementData[elementCount - 1];
}

/** 将Vector的index处的元素修改成E*/
public synchronized void setElementAt(E obj, int index) {
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " +
            elementCount);
    }
    elementData[index] = obj;
}

/** 删除Vector的index处元素*/
public synchronized void removeElementAt(int index) {
    modCount++;
    if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException(index + " >= " +
            elementCount);
    }
    else if (index < 0) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    int j = elementCount - index - 1;
    if (j > 0) {
        System.arraycopy(elementData, index + 1, elementData, index, j);
    }
    elementCount--;
    elementData[elementCount] = null; /* to let gc do its work */
}

/** 将obj插入Vector的index处、新增元素的后面的原来的元素后移1位、效率相对LinkedList低的原因*/

```

```

public synchronized void insertElementAt(E obj, int index) {
    modCount++;
    if (index > elementCount) {
        throw new ArrayIndexOutOfBoundsException(index
            + " > " + elementCount);
    }
    ensureCapacityHelper(elementCount + 1);
    System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);
    elementData[index] = obj;
    elementCount++;
}

/** 将obj追加到Vector末尾*/
public synchronized void addElement(E obj) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = obj;
}

/** 删除obj、若成功返回true、失败返回false*/
public synchronized boolean removeElement(Object obj) {
    modCount++;
    int i = indexOf(obj);
    if (i >= 0) {
        removeElementAt(i);
        return true;
    }
    return false;
}

/** 删除Vector所有元素*/
public synchronized void removeAllElements() {
    modCount++;
    // Let gc do its work
    for (int i = 0; i < elementCount; i++)
        elementData[i] = null;
    elementCount = 0;
}

/** 克隆Vector*/
public synchronized Object clone() {
    try {
        Vector<E> v = (Vector<E>) super.clone();
        v.elementData = Arrays.copyOf(elementData, elementCount);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}

/** 将Vector转化成Object[]*/
public synchronized Object[] toArray() {
    return Arrays.copyOf(elementData, elementCount);
}

/** 将Vector转换成T[]、相对与上面方法、对返回数组做了转型*/
public synchronized <T> T[] toArray(T[] a) {
    if (a.length < elementCount)
        return (T[]) Arrays.copyOf(elementData, elementCount, a.getClass());

    System.arraycopy(elementData, 0, a, 0, elementCount);

    if (a.length > elementCount)
        a[elementCount] = null;

    return a;
}

// Positional Access Operations

/** 获取index处元素*/
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return (E)elementData[index];
}

```

```

/** 将index处元素设置成element、返回oldElement*/
public synchronized E set(int index, E element) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

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

/** 将e追加到Vector末尾处*/
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}

/** 删除o*/
public boolean remove(Object o) {
    return removeElement(o);
}

/** 将element添加到index处、后面的所有元素后移一位*/
public void add(int index, E element) {
    insertElementAt(element, index);
}

/** 删除index处元素、返回oldElement*/
public synchronized E remove(int index) {
    modCount++;
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    Object oldValue = elementData[index];

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

    return (E)oldValue;
}

/** 删除所有元素*/
public void clear() {
    removeAllElements();
}

// Bulk Operations

/** 是否包含Collection c?*/
public synchronized boolean containsAll(Collection<?> c) {
    return super.containsAll(c);
}

/** 将Collection c所有元素追加到Vector末尾*/
public synchronized boolean addAll(Collection<? extends E> c) {
    modCount++;
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityHelper(elementCount + numNew);
    System.arraycopy(a, 0, elementData, elementCount, numNew);
    elementCount += numNew;
    return numNew != 0;
}

/** 将Vector中所有与Collection c相同的元素删除*/
public synchronized boolean removeAll(Collection<?> c) {
    return super.removeAll(c);
}

/** 求Vector与传入的Collection的元素的交集*/
public synchronized boolean retainAll(Collection<?> c) {
    return super.retainAll(c);
}

/** 将Collection所有元素追加到Vector从index处开始的位置、后面的原来是元素后移c.size()个位置*/
public synchronized boolean addAll(int index, Collection<? extends E> c) {
    modCount++;

```

```

        if (index < 0 || index > elementCount)
            throw new ArrayIndexOutOfBoundsException(index);

        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityHelper(elementCount + numNew);

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

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

    /** 判断Vector中是否包含o*/
    public synchronized boolean equals(Object o) {
        return super.equals(o);
    }

    /** 返回hash值*/
    public synchronized int hashCode() {
        return super.hashCode();
    }

    public synchronized String toString() {
        return super.toString();
    }

    /** 返回Vector子集*/
    public synchronized List<E> subList(int fromIndex, int toIndex) {
        return Collections.synchronizedList(super.subList(fromIndex, toIndex),
                                                this);
    }

    /** 删除部分元素*/
    protected synchronized void removeRange(int fromIndex, int toIndex) {
        modCount++;
        int numMoved = elementCount - toIndex;
        System.arraycopy(elementData, toIndex, elementData, fromIndex,
                           numMoved);

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

    /** 将Vector写入到ObjectOutputStream流中、注意：没有对应的ObjectInputStream来读取*/
    private synchronized void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException
    {
        s.defaultWriteObject();
    }
}

```

总结：从Vector源码可以看出、Vector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Vector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(intminCapacity)方法中会确保数组的最小size、当不够时会原来的容量变为oldCapacity *2、之后那个值与传入的最小容量进行比较、若还小于传入的最小容量值、则使用传入的最小容量值。

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、有些方法完全没有必要、比如对元素的增删改查的、后缀为Element的完全可以使用从List中继承的增删改查来替代。

7、对于Vector得到的Iterator、ListIterator是fail-fast机制、针对此现象、Vector提供了自己特有的遍历方式Enumeration、此迭代不是fail-fast机制的。用于并发线程的环境中。

8、在使用ObjectOutputStream时、会先将Vector的capacity写入到流中、他与ArrayList不同的是：Vector没有ObjectInputStream用于读取写入的Vector。

五：Vector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Vector

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

02) 使用ListIterator遍历Vector

```
ListIterator<String> it = v.listIterator();
while(it.hasNext()){
    String s = it.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Vector

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

04) 使用增强for循环遍历Vector

```
for(String str : v){
    String s = str;
}
```

05) 使用Enumeration迭代Vector

```
Enumeration<String> e = v.elements();
while(e.hasMoreElements()){
    String s = e.nextElement();
}
```

06) 示例

```

package com.chy.collection.example;

import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;

@SuppressWarnings("unused")
public class EragodicVector {
    private static Vector<String> v ;
    //静态块初始化一个较大的Vector
    static{
        v = new Vector<String>();
        for (int i = 0; i < 300000; i++) {
            v.add("a");
        }
    }

    /**
     * 使用Iterator迭代
     */
    private static void testIterator(){
        long start = currentTime();
        Iterator<String> it = v.iterator();
        while(it.hasNext()){
            String s = it.next();
        }
        long end = currentTime();
        System.out.println("iterator time : " + (end - start) + "ms");
    }

    /**
     * 使用ListIterator迭代
     */
    private static void testListIterator(){
        long start = currentTime();
        ListIterator<String> it = v.listIterator();
        while(it.hasNext()){
            String s = it.next();
        }
        long end = currentTime();
        System.out.println("ListIterator time : " + (end - start) + "ms");
    }

    /**
     * 使用foreach循环
     */
    private static void testForeache(){
        long start = currentTime();
        for(String str : v){
            String s = str;
        }
        long end = currentTime();
        System.out.println("ListIterator time : " + (end - start) + "ms");
    }

    /**
     * 使用Enumeration迭代
     */
    private static void testEnumeration(){
        long start = currentTime();
        Enumeration<String> e = v.elements();
        while(e.hasMoreElements()){
            String s = e.nextElement();
        }
        long end = currentTime();
        System.out.println("Enumeration time : " + (end - start) + "ms");
    }

    /**
     * 使用随机访问迭代
     */
    private static void testRandomAccess(){
        long start = currentTime();
        for (int i = 0; i < v.size(); i++) {
            String s = v.get(i);
        }
        long end = currentTime();
        System.out.println("RandomAccess time : " + (end - start) + "ms");
    }
}

```

```
private static long currentTime() {  
    return System.currentTimeMillis();  
}  
  
public static void main(String[] args) {  
    testIterator();  
    testListIterator();  
    testRandomAccess();  
    testEnumeration();  
    testForeache();  
}  
}
```

结果及说明：

```
iterator time : 32ms  
ListIterator time : 31ms  
RandomAccess time : 16ms  
Enumeration time : 31ms  
ListIterator time : 31ms
```

与ArrayList类似、使用随机访问效率最高、其他的都基本相似。

2、API演示

```

package com.chy.collection.example;

import java.util.Arrays;
import java.util.Vector;

import com.chy.collection.bean.Student;

@SuppressWarnings("unused")
public class VectorTest {

    /**
     * 测试Vector添加元素方法、与size有关的方法
     */
    private static void testAddSizeElements(){
        //初始化含有字符串“abcdefg”的Vector
        Vector<String> v = new Vector<String>();
        v.add("a");
        v.add(v.size(), "b");
        v.addElement("c");
        String[] strArray = {"d", "e"};
        Vector<String> v1 = new Vector<String>(Arrays.asList(strArray));
        v.addAll(v.size(), v1);
        //在结尾处插入一个元素“f”
        v.insertElementAt("f", v.size());
        v.addElement("g");
        println(v.toString());

        //查看当前Vector的size和容量
        println("v size : " + v.size() + " v capacity : " + v.capacity());
        //确保当前Vector的容量不小于10
        v.ensureCapacity(10);
        println("v size : " + v.size() + " v capacity : " + v.capacity());
        //确保当前Vector的size与容量同步
        v.trimToSize();
        println("v size : " + v.size() + " v capacity : " + v.capacity());
        //设置Vector的size
        v.setSize(15);
        println("v size : " + v.size() + " v capacity : " + v.capacity());
        println(v.toString());
        //确保当前Vector的size与容量同步
        v.trimToSize();
        println("v size : " + v.size() + " v capacity : " + v.capacity());

        /*
         * 结果说明:
         * 1、size指的是Vector中所具有的元素的个数、包括值为null的元素
         * 2、capacity指的是Vector所能容纳的最大的元素个数、
         * 3、ensureCapacity(int minCapacity)方法是确保Vector最小的容纳元素个数不小于传入的

         * 4、setSize()是改变当前Vector中元素个数
         * 5、trimToSize() 都是取size的值
         */
    }

    /**
     * 测试包含、删除方法
     */
    private static void testContainsRomve(){
        //初始化包含学号从1到10的十个学生的ArrayList
        Vector<Student> v1 = new Vector<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");
        v1.add(s1);
        v1.add(s2);
        v1.add(s3);
        v1.add(s4);
        for (int i = 5; i < 11; i++) {
            v1.add(new Student(i, "chy" + i));
        }
        System.out.println(v1);

        //初始化包含学号从1到4的四个学生的ArrayList
        Vector<Student> v2 = new Vector<Student>();
        v2.add(s1);
        v2.add(s2);
        v2.add(s3);
        v2.add(s4);
        //查看v1中是否包含学号为1的学生
    }
}

```

参数

```

        println(v1.contains(s1));
        //查看v1中是否包含学号为5的学生、因为下面学号为5的学生是新创建的对象、所以不包含
        //从这里可以看出、v1中保存的是对象的引用
        println(v1.contains(new Student(5, "chy5")));
        //查看v1中是否包含集合v2
        println(v1.containsAll(v2));
        //修改v2中第一元素的值
        v2.set(0, new Student(10, "chy10"));
        //查看v1是否包含v2
        println(v1.containsAll(v2));

        //删除当前Vector第一个元素
        println(v1.remove(0));
        //删除当前Vector第一个元素
        println(v1);

        //如果s3存在、则删除s3
        if(v1.contains(s3)){
            println(v1.remove(s3));
        }
        //删除v1中所包含的v2的元素
        v1.removeAll(v2);
        println(v1);

        //求v1与v2的交集
        v1.retainAll(v2);
        println(v1);
        //删除v1中所有元素
        v1.removeAllElements();
        //v1.clear();作用相同
        println(v1);
    }

    /**
     * 测试Vector查找、修改元素方法
     */
    private static void testGetSet(){
        //初始化包含"abcde"的Vector
        Vector<String> v1 = new Vector<String>();
        v1.add("a");
        v1.add("b");
        v1.add("c");
        v1.add("d");
        v1.add("e");
        //获取"a"元素的索引
        println("从前向后找 第一个 a 元素索引   : " + v1.indexOf("a") + "从后向前找: " + v1.
lastIndexOf("a"));
        //获取第一个、最后一个元素
        println("first element : " + v1.firstElement() + " last element : " + v1.lastElemen
t());

        //从前或者后面指定的索引开始查找 "a"的索引值
        println("from start: " + v1.indexOf("a", 1) + " from end: " + v1.lastIndexOf("a", v
1.size() - 1));

        //将Vector中“b” 修改成“a”
        if(v1.indexOf("b") != -1){
            v1.set(v1.indexOf("b"), "a");
        }
        println(v1);
    }

    /**
     * 测试数组集合之间的转换
     */
    private static void testConvertBetweenArrayAndVector(){
        //Array2Vector
        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"};
        Vector<String> v = (Vector<String>)Arrays.asList(strArray);//作为String[]使用时会报错、因为此方法的返回值是Object、将一个Object强转成Vector<String>会报异常
        Vector<String> v1 = new Vector<String>(Arrays.asList(strArray));//可正常使用、在Vec
tor构造方法中会根据strArray的类型返回对应类型的Vector。

        //Vector2Array
        String[] strArray1 = (String[])v1.toArray();//作为String[]使用时会报错、因为v1.toAr
ray()返回的是Object[]、强转会出错
        String[] strArray2 = v1.toArray(new String[0]);//可正常使用、v1.toArray(new String
[0])会根据传入的参数类型、将返回结果转换成对应类型
    }

```

```

    }

    private static void println(Object str) {
        System.out.println(str.toString());
    }

    public static void main(String[] args) {
        //      testAddSizeElements();
        //      testContainsRomve();
        //      testGetSet();
    }
}

```


总结：

对于Vector、是一个比较古老的类、相对于ArrayList而言、它通过将许多方法使用synchronized修饰来保证线程安全性、但是保证线程安全是要代价的、这也使得他的效率并没有ArrayList高、所以在单线程环境中不推荐使用Vector、即使在并发情况也也不推荐使用Vector、而是使用被包装后的ArrayList ！

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。





 目录
 标签：Vector (<http://so.csdn.net/so/search/s.do?q=Vector&t=blog>) /
 Vector框架图 (<http://so.csdn.net/so/search/s.do?q=Vector框架图&t=blog>) /
 Collection (<http://so.csdn.net/so/search/s.do?q=Collection&t=blog>) /
 Iterator (<http://so.csdn.net/so/search/s.do?q=Iterator&t=blog>) /
 Enumeration (<http://so.csdn.net/so/search/s.do?q=Enumeration&t=blog>) /

0条评论



收藏

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



发表评论

暂无评论

相关文章推荐

Java集合之Vector (/qq924862077/article/details/48039567)

Vector是矢量队列，它继承了AbstractList，实现了List、RandomAccess、Cloneable、java.io.Serializable接口。Vector接口依赖图：...



qq924862077 2015-08-28 00:18 6470

Java 集合深入理解（12）：古老的 Vector (/u011240877/article/details/52900893)

点击查看 Java 集合框架深入理解 系列，-(° - °)つ口 乾杯~ 今天刮台风，躲屋里看看 Vector ！都说 Vector 是线程安全的 Arra...



u011240877 2016-10-23 12:03 1936

Java集合06--Vector源码详解 (/wangxiaotongfan/article/details/51332193)

概要 学完ArrayList和LinkedList之后，我们接着学习Vector。学习方式还是和之前一样，先对Vector有个整体认识，然后再学习它的源码；最后再通过实例来学会使用它。 第1部分 Ve...



wangxiaotongfan 2016-05-06 15:49 285

java类vector的详细用法整理 (/lskyne/article/details/8769147)

Vector v=new Vector(); E可以是泛型类，如String,可以自定义，感觉E很像但链表中的节点定义，Vector则是数组 具体用法如下 ArrayList会比Vect...



lskyne 2013-04-07 19:19 8833

java集合系列——List集合之Vector介绍（四） (/u010648555/article/details/59199840)

Vector 类可以实现可增长的对象数组。与数组一样，它包含可以使用整数索引进行访问的组件。但是，Vector 的大小可以根据需要增大或缩小，以适应创建 Vector 后进行添加或移除项的操作。Vec...



u010648555 2017-03-01 23:37 200

java.util.vector中的vector的详细用法及与list的区别 (/lv18092081172/article/details/51516694)

转载自：http://www.cnblogs.com/strivers/archive/2010/12/28/1918877.html ArrayList会比Vector快，他是非同步的，如果设计...



lv18092081172 2016-05-27 14:52 2941

c++的vector赋值方法汇总 (/educast/article/details/12966379)

```
#include #include using namespace std; void main() { vec...
```



educast 2013-10-23 08:33 3887

Java 集合ArrayList与Vector的详解 (/qq_33642117/article/details/51998866)

--| Iterable ----| Collection -----| List -----| ArrayList 底层采用数组实现，默认10。每次增长 ...



qq_33642117 2016-07-22 20:56 560

java集合——Vector的用法 (/tingzhiyi/article/details/51885926)

/* 功能：Vector的用法 */ package Application; import java.util.*; public class jihe { public sta...



tingzhiyi 2016-07-12 09:06 418

java 中创建Vector二维数组添加一维Vector的问题 (/abyss521/article/details/8843598)

//创建对工作簿文件的引用 HSSFWorkbook workbook = new HSSFWorkbook(new FileInputStream(file)); // 创建对工作表的引用 HSS...



abyss521 2013-04-24 11:19 1480

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_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 6349

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 <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)

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

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

网址 (<http://zhangziyueup.iteye.com/blog/1325894>)

Soft<http://www.donews.net/eb国内破解论坛> ★万花筒极酷大论坛 <a class="li



zhangziyueup 2004-09-22 13:49 👁 450

Java线程池 (/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
