个人资料

Sunday_Vssupermadman

+ 加关注 发私信

访问： 43869次
积分： 762
等级： BLOG 3
排名： 千里之外

原创： 15篇 转载： 108篇
译文： 0篇 评论： 19条

文章搜索

文章分类

Android HTTP (10)
Android 图片 (4)
Java 设计模式 (11)
设计框架 (0)
腾讯SDK (2)
Java基础知识 (18)
Android基础知识 (5)
vlc (0)
Android 重构历程 (1)

文章存档

2016年04月 (1)
2015年11月 (1)
2015年10月 (1)
2015年07月 (1)
2015年05月 (1)

展开

阅读排行

Http 基本认证模式 (1452)
C#中的串口通信 (1140)
Android webkit学习笔记 (1066)
Multipart/form-data POS (879)
Java Socket 网络编程心 (872)
Android HttpClient (768)
adb 协议 (694)

转 Java-Collections Framework学习与总结-ArrayList

2015-03-31 11:53 310人阅读 评论(0) 收藏 举报

分类： Java基础知识（17）

工作中经常会用到Java的集合类，最近不忙了，把相关知识总结一下，便于理解记忆。

打开java.util.ArrayList的源代码，首先映入眼帘的是@author Josh Bloch(相对于源码，本人更喜欢看故事，每次看到一份源码，先看看作者是谁。然后搜索一下作者的百科以及一些八卦。。。以及与作者相关的人的百科和一些八卦。。。以及。。。一上午就过去了)，这家伙是谁？。。。看过Effective Java的人一定会惊呼：这不就是。。。 没错，这位大叔就是玉树凌风，风流倜傥，人称Java之母的Josh Bloch，Collections Framework就是他一手打造的，后来去google做了Java首席架构师。

言归正传，不过话又说回来，有说Collections Framework是参考了Doug Lea大叔的Collections。。。有兴趣的可以去看看相关源码，总之他们都很牛逼就对了。

继续吧，看一个类的时候首先要看看这个类能干什么，有什么特性。这些都可以在这个类实现的接口上体现了(废话。。。)。好，直接从最顶级的接口看吧。首先是接口java.lang.Iterable:

Java代码

```
01. package java.lang;
02. import java.util.Iterator;
03.
04. public interface Iterable<T> {
05.     Iterator<T> iterator();
06. }
```

由于篇幅的关系，去掉了注释。注释上说明，实现了这个接口的类，可以使用"foreach"语法。

接下来是接口java.util.Collection:

Java代码

```
01. package java.util;
02. public interface Collection<E> extends Iterable<E> {
03.     int size();
04.     boolean isEmpty();
05.     boolean contains(Object o);
06.     Iterator<E> iterator();
07.     Object[] toArray();
08.     <T> T[] toArray(T[] a);
09.     boolean add(E e);
10.     boolean remove(Object o);
11.     boolean containsAll(Collection<?> c);
12.     boolean addAll(Collection<? extends E> c);
13.     boolean removeAll(Collection<?> c);
14.     boolean retainAll(Collection<?> c);
15.     void clear();
16.     boolean equals(Object o);
17.     int hashCode();
18. }
```

里面的注释很详细，反正大概意思是说(英语不太好)，这是集合层次的顶级接口，代表了一组对象blablablabla，所有通用的实现应该提供两个"标准"的构造方法:无参的构造方法和拥有一个集合类型参数的构造方法blablablabla，总之这个接口对集合进行了高度滴、抽象滴定义：）

接下来就是java.util.List接口了。

Java代码

```
01. public interface List<E> extends Collection<E> {
02.     //和Collection重复的方法就不贴了,但必须知道有这些方法
03.     boolean addAll(int index, Collection<? extends E> c);
04.     E get(int index);
05.     E set(int index, E element);
06.     void add(int index, E element);
07.     E remove(int index);
08.     int indexOf(Object o);
09.     int lastIndexOf(Object o);
10.     ListIterator<E> listIterator();
11.     ListIterator<E> listIterator(int index);
12.     List<E> subList(int fromIndex, int toIndex);
13. }
```

list代表了一个有序的集合，可以通过index来访问和查找集合内的元素，集合内元素是可重复的，因为index(集合中的位置)不同。

还有一个ArrayList实现的接口是java.util.RandomAccess:

Java代码

```
01. package java.util;
02. public interface RandomAccess {
03. }
```

一看没有方法的接口，就知道这大概是个标记接口喽。实现这个接口的集合类会在随机访问元素上提供很好的性能。比如说，ArrayList实现了RandomAccess而LinkedList没有，那么当我们拿到一个集合时(假设这个集合不是ArrayList就是LinkedList)，如果这个集合时ArrayList，那么我们遍历集合元素可以使用下标遍历，如果是LinkedList，就使用迭代器遍历。比如集合工具类Collections中提供的对集合中元素进行二分查找的方法，代码片段如下：

Java代码

```
01. public static <T>
02.     int binarySearch(List<? extends Comparable<? super T>> list, T key) {
03.         if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
04.             return Collections.indexedBinarySearch(list, key);
05.         else
06.             return Collections.iteratorBinarySearch(list, key);
07.     }
```

还有java.lang.Cloneable和java.io.Serializable两个标记接口，表示ArrayList是可克隆的、可序列化的。

还要看一下AbstractCollection和AbstractList，这两个抽象类里实现了一部分公共的功能，代码都还比较容易看懂。

需要注意的地方是AbstractList中的一个属性modCount。这个属性主要由集合的迭代器来使用，对于List来说，可以调用iterator()和listIterator()等方法来生成一个迭代器，这个迭代器在生成时会将List的modCount保存起来(迭代器实现为List的内部类)，在迭代过程中会去检查当前list的modCount是否发生了变化(和自己保存的进行比较)，如果发生变化，那么马上抛出java.util.ConcurrentModificationException异常，这种行为就是fail-fast。

好了，终于可以看ArrayList了，在看之前可以先思考一下，如果我们自己来实现ArrayList，要怎么实现。我们都会把ArrayList简单的看作动态数组，说明我们使用它的时候大都是当做数组来使用的，只不过它的长度是可改变的。那我们的问题就变成了一个实现长度可变化的数组的问题了。那么首先，我们会用一个数组来做底层数据存储的结构，新建的时候会提供一个默认的数组长度。当我们添加或删除元素的时候，会改变数据的长度(当默认长度不够或者其他情况下)，这里涉及到数据的拷贝，我们可以使用系统提供的System.arraycopy来进行数组拷贝。如果我们每次添加或者删除元素时都会因为改变数组长度而拷贝数组，那也太2了，可以在当数组长度不够的时候，扩展一定的空间，比如可以扩展到原来的2倍，这样会提高一点性能(如果不是在数组末尾添加或删除元素的话，还是会进行数组拷贝)，但同时浪费了一点空间。再看看上面的接口，比如我们要实现size方法，那么就不能直接返回数组的长度了(由于上面的原因)，也没办法去遍历数组得到长度(因为可能存在null元素)，我们会想到利用一个私有变量来

保存长度，在添加删除等方法里面去相应修改这个变量(这里不考虑线程安全问题，因为ArrayList本身就不是线程安全的)。

带着这些问题，来看一下ArrayList的代码。

首先，真正存储元素的是一个数组。

```java
/**
  * The array buffer into which the elements of the ArrayList are stored.
  * The capacity of the ArrayList is the length of this array buffer.
  */
 private transient Object[] elementData;
```

那么这个数组该怎么初始化呢？数组的长度该怎么确定呢？看一下构造方法。

```java
/**
  * Constructs an empty list with the specified initial capacity.
  *
  * @param   initialCapacity   the initial capacity of the list
  * @exception IllegalArgumentException if the specified initial capacity
  *            is negative
  */
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    this.elementData = new Object[initialCapacity];
}

/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
    this(10);
}
```

可以看到，我们可以选择使用一个参数作为ArrayList内部数组的容量来构造一个ArrayList；或者使用无参的构造方法，这样会默认使用10作为数组容量。我们在实际应用中可以根据具体情况来选择使用哪个构造方法，例如在某种特定逻辑下，我们需要存储200个元素(固定值)到一个ArrayList里，那我们就可以使用带一个int型参数的构造方法，这样就避免了在数组长度不够，进行扩容时，内部数组的拷贝。

当然，根据接口java.util.Collection的规范(非强制)，ArrayList也提供用一个集合做为参数的构造方法。

```java
/**
  * Constructs a list containing the elements of the specified
  * collection, in the order they are returned by the collection's
  * iterator.
  *
  * @param c the collection whose elements are to be placed into this list
  * @throws NullPointerException if the specified collection is null
  */
 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);
 }
```

上面说到说考虑用一个私有变量来存储size(集合中实际原始的个数)。在ArrayList也有这样一个变量。

```java
```

```java
/**
 * The size of the ArrayList (the number of elements it contains).
 *
 * @serial
 */
private int size;
```

有了这个变量，一些方法实现起来就非常简单了。

Java代码

```java
/**
 * Returns the number of elements in this list.
 *
 * @return the number of elements in this list
 */
public int size() {
    return size;
}

/**
 * Returns <tt>true</tt> if this list contains no elements.
 *
 * @return <tt>true</tt> if this list contains no elements
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * Returns <tt>true</tt> if this list contains the specified element.
 * More formally, returns <tt>true</tt> if and only if this list contains
 * at least one element <tt>e</tt> such that
 * <tt>(o==null ? e==null : o.equals(e))</tt>.
 *
 * @param o element whose presence in this list is to be tested
 * @return <tt>true</tt> if this list contains the specified element
 */
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

/**
 * Returns the index of the first occurrence of the specified element
 * in this list, or -1 if this list does not contain the element.
 * More formally, returns the lowest index <tt>i</tt> such that
 * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
 * or -1 if there is no such index.
 */
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;
}

/**
 * Returns the index of the last occurrence of the specified element
 * in this list, or -1 if this list does not contain the element.
 * More formally, returns the highest index <tt>i</tt> such that
 * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>,
 * or -1 if there is no such index.
 */
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--)
```

```
66.      if (o.equals(elementData[i]))
67.          return i;
68.      }
69.      return -1;
70.  }
```

由于内部数组的存在，一些基于下标的方法实现起来也非常简单。

Java代码

```
01.  /**
02.      * Returns the element at the specified position in this list.
03.      *
04.      * @param  index index of the element to return
05.      * @return the element at the specified position in this list
06.      * @throws IndexOutOfBoundsException {@inheritDoc}
07.      */
08.     public E get(int index) {
09.         RangeCheck(index);
10.
11.         return (E) elementData[index];
12.     }
13.
14.     /**
15.      * Replaces the element at the specified position in this list with
16.      * the specified element.
17.      *
18.      * @param index index of the element to replace
19.      * @param element element to be stored at the specified position
20.      * @return the element previously at the specified position
21.      * @throws IndexOutOfBoundsException {@inheritDoc}
22.      */
23.     public E set(int index, E element) {
24.         RangeCheck(index);
25.
26.         E oldValue = (E) elementData[index];
27.         elementData[index] = element;
28.         return oldValue;
29.     }
30.
31.     /**
32.      * Appends the specified element to the end of this list.
33.      *
34.      * @param e element to be appended to this list
35.      * @return <tt>true</tt> (as specified by {@link Collection#add})
36.      */
37.     public boolean add(E e) {
38.         ensureCapacity(size + 1);  // Increments modCount!!
39.         elementData[size++] = e;
40.         return true;
41.     }
42.
43.     /**
44.      * Inserts the specified element at the specified position in this
45.      * list. Shifts the element currently at that position (if any) and
46.      * any subsequent elements to the right (adds one to their indices).
47.      *
48.      * @param index index at which the specified element is to be inserted
49.      * @param element element to be inserted
50.      * @throws IndexOutOfBoundsException {@inheritDoc}
51.      */
52.     public void add(int index, E element) {
53.         if (index > size || index < 0)
54.             throw new IndexOutOfBoundsException(
55.              "Index: "+index+", Size: "+size);
56.
57.         ensureCapacity(size+1);  // Increments modCount!!
58.         System.arraycopy(elementData, index, elementData, index + 1,
59.                          size - index);
60.         elementData[index] = element;
61.         size++;
62.     }
63.
64.     /**
65.      * Removes the element at the specified position in this list.
66.      * Shifts any subsequent elements to the left (subtracts one from their
```

```
     * indices).
     *
     * @param index the index of the element to be removed
     * @return the element that was removed from the list
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    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;
    }
```

有几个要注意的地方，首先是RangeCheck方法，顾名思义是做边界检查的，看看方法实现。

Java代码

```
/**
  * Checks if the given index is in range.  If not, throws an appropriate
  * runtime exception.  This method does *not* check if the index is
  * negative: It is always used immediately prior to an array access,
  * which throws an ArrayIndexOutOfBoundsException if index is negative.
  */
 private void RangeCheck(int index) {
     if (index >= size)
         throw new IndexOutOfBoundsException(
      "Index: "+index+", Size: "+size);
 }
```

异常信息是不是很眼熟，呵呵。

还有就是在"添加"方法中的ensureCapacity方法，上面也考虑到扩容的问题，这个方法其实就干了这件事情。

Java代码

```
/**
 * Increases the capacity of this <tt>ArrayList</tt> instance, if
 * necessary, to ensure that it can hold at least the number of elements
 * specified by the minimum capacity argument.
 *
 * @param   minCapacity   the desired minimum capacity
 */
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);
    }
}
```

注意当数组长度不够的时候，会进行数组的扩展，扩展到原来长度的1.5倍(注意整型的除法)加1，比如原来是2，会扩展到4，原来是30，会扩展到46。当数组的长度大到算出的新容量超出了整型最大范围，那么新容量就等于传入的"最小容量"。

最后再看一下另一个删除方法。

Java代码

```java
/**
 * Removes the first occurrence of the specified element from this list,
 * if it is present.  If the list does not contain the element, it is
 * unchanged.  More formally, removes the element with the lowest index
 * <tt>i</tt> such that
 * <tt>(o==null ? get(i)==null : o.equals(get(i)))</tt>
 * (if such an element exists).  Returns <tt>true</tt> if this list
 * contained the specified element (or equivalently, if this list
 * changed as a result of the call).
 *
 * @param o element to be removed from this list, if present
 * @return <tt>true</tt> if this list contained the specified element
 */
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;
}

/*
 * Private remove method that skips bounds checking and does not
 * return the value removed.
 */
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
}
```

可以看到，像size、isEmpty、get、set这样的方法时间复杂度为O(1)，而像indexOf、add、remove等方法，最坏的情况下(如添加元素到第一个位置，删除第一个位置的元素，找最后一个元素的下标等)时间复杂度为O(n)。

一般情况下内部数组的长度总是大于集合中元素总个数的，ArrayList也提供了一个释放多余空间的方法，我们可以适时调用此方法来减少内存占用。

Java代码

```java
/**
 * Trims the capacity of this <tt>ArrayList</tt> instance to be the
 * list's current size.  An application can use this operation to minimize
 * the storage of an <tt>ArrayList</tt> instance.
 */
public void trimToSize() {
    modCount++;
    int oldCapacity = elementData.length;
    if (size < oldCapacity) {
                elementData = Arrays.copyOf(elementData, size);
    }
}
```

基本上ArrayList的内容总结的差不多啦。最后，别忘了它是线程不安全的。

上一篇 Java-Collections Framework学习与总结-IdentityHashMap
下一篇 Java-Collections Framework学习与总结-LinkedList

快速回复
我要收藏
返回顶部

相关文章推荐

- framework调用第三方的Jar包
- 【直播】计算机视觉原理及实战—屈教授
- JAVA FRAMEWORK
- 【套餐】Spark+Scala课程包--陈超
- android framework中添加私有资源文件
- 【套餐】Linux应用和网络编程实战套餐--朱有鹏
- Java Web Framework综述
- 【直播】机器学习&数据挖掘7周实训--韦玮
- JavaFramework 3.0 的框架思想
- 【套餐】从0蜕变为自动化测试工程师--李晓鹏
- Java-Collections Framework学习与总结-HashMap
- 【直播】广义线性模型及其应用——李科
- 我的Java开发学习之旅------>Java经典面试题
- Java-Collections Framework学习与总结-LinkedH…
- java学习总结
- 通过C#实现集合类纵览.NET Collections及相关技术

查看评论

暂无评论

发表评论

用 户 名： qq_36596145

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

关闭