# 源码分析之\_java集合体系\_基于：jdk1.8\_201

## Jdk源码如何生成本地api文档

通过以下文件将打印出的内容保存在本地package.txt文件中：

**package** com.prosayj.springboot.java数据结构;  
  
**import** java.io.File;  
**import** java.util.Vector;  
  
*/\*\*  
 \** ***@author*** *yangjian  
 \** ***@description TODO*** *\** ***@email*** *ProSayJ@gmail.cn  
 \** ***@creatTime*** *2019/10/27 下午 09:01  
 \** ***@since*** *1.0.0  
 \*/***public class** SniffDir {  
 **private** String **srcDir**;  
 **private** Vector<String> **vs**;  
  
 **public** SniffDir() {  
 **srcDir** = **"D:/src"**;  
 **vs** = **new** Vector<String>();  
 }  
  
 */\*\*  
 \* Sniff函数， 其实就是一个DFS，深度优先搜索  
 \* 用来获取src目录下的满足“当前路径下存在.java文件”这一条件的所有文件夹  
 \*  
 \** ***@param file*** *\*/* **public void** Sniff(File file) {  
 *//File file=new File(curDir);* File[] files = file.listFiles();  
 **int** len = files.**length**;  
 **boolean** ok = **false**;*//表示当前目录下是否有java文件，ok=true表示有并且记录过了。* **for** (**int** i = 0; i < len; i++) {  
 *//System.out.println(files[i].toString());* **if** (files[i].isDirectory()) {  
 Sniff(files[i]);  
 } **else if** (files[i].isFile() && !ok) {*//files[i]为文件（肯定是java文件），并且还没有把当前路径放入向量vs中* ok = **true**;  
 **vs**.addElement(file.toString().substring(**srcDir**.length() + 1));  
 *//System.out.println(files[i].toString());* }  
 }  
 }  
  
 String **sb** = **null**;  
  
 **public void** Print() {  
 **int** len = **vs**.size();  
 **for** (**int** i = 0; i < len; i++) {  
 **sb** = **vs**.get(i);  
 **sb** = **sb**.replace(**'\\'**, **'.'**);  
 System.***out***.println(**sb**);  
 }  
 }  
  
 **public static void** main(String[] args) {  
 SniffDir sd = **new** SniffDir();  
 sd.Sniff(**new** File(sd.**srcDir**));  
 sd.Print();  
 }  
  
}

将package.txt文件放在src目录下。Cmd运行：javadoc javadoc -d api @package.txt 即可。

## ArrayList：

### 继承结构图

### 描述：

*/\*\*  
 \* Resizable-array implementation of the <tt>List</tt> interface. Implements  
 \* all optional list operations, and permits all elements, including  
 \* <tt>null</tt>. In addition to implementing the <tt>List</tt> interface,  
 \* this class provides methods to manipulate the size of the array that is  
 \* used internally to store the list. (This class is roughly equivalent to  
 \* <tt>Vector</tt>, except that it is unsynchronized.)  
 \**List 接口的大小可变数组的实现。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。（此类大致上等同于 Vector 类，除了此类是不同步的。）

*\* <p>The <tt>size</tt>, <tt>isEmpty</tt>, <tt>get</tt>, <tt>set</tt>,  
 \* <tt>iterator</tt>, and <tt>listIterator</tt> operations run in constant  
 \* time. The <tt>add</tt> operation runs in <i>amortized constant time</i>,  
 \* that is, adding n elements requires O(n) time. All of the other operations  
 \* run in linear time (roughly speaking). The constant factor is low compared  
 \* to that for the <tt>LinkedList</tt> implementation.  
 \**size、isEmpty、get、set、iterator 和 listIterator 操作都以固定时间运行。add 操作以*分摊的固定时间* 运行，也就是说，添加 n 个元素需要 O(n) 时间。其他所有操作都以线性时间运行（大体上讲）。与用于 LinkedList 实现的常数因子相比，此实现的常数因子较低。

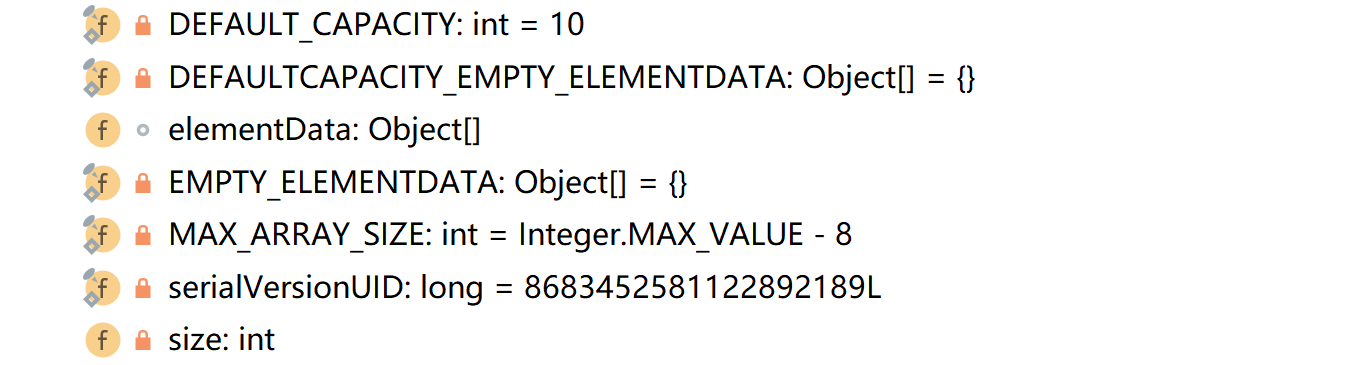
*\* <p>Each <tt>ArrayList</tt> instance has a <i>capacity</i>. The capacity is  
 \* the size of the array used to store the elements in the list. It is always  
 \* at least as large as the list size. As elements are added to an ArrayList,  
 \* its capacity grows automatically. The details of the growth policy are not  
 \* specified beyond the fact that adding an element has constant amortized  
 \* time cost.  
 \**每个 ArrayList 实例都有一个*容量*。该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向 ArrayList 中不断添加元素，其容量也自动增长。并未指定增长策略的细节，因为这不只是添加元素会带来分摊固定时间开销那样简单。

*\* <p>An application can increase the capacity of an <tt>ArrayList</tt> instance  
 \* before adding a large number of elements using the <tt>ensureCapacity</tt>  
 \* operation. This may reduce the amount of incremental reallocation.  
 \**在添加大量元素前，应用程序可以使用 ensureCapacity 操作来增加 ArrayList 实例的容量。这可以减少递增式再分配的数量。

*\* <p><strong>Note that this implementation is not synchronized.</strong>  
 \* If multiple threads access an <tt>ArrayList</tt> instance concurrently,  
 \* and at least one of the threads modifies the list structurally, it  
 \* <i>must</i> be synchronized externally. (A structural modification is  
 \* any operation that adds or deletes one or more elements, or explicitly  
 \* resizes the backing array; merely setting the value of an element is not  
 \* a structural modification.) This is typically accomplished by  
 \* synchronizing on some object that naturally encapsulates the list.  
 \****注意，此实现不是同步的。**如果多个线程同时访问一个 ArrayList 实例，而其中至少一个线程从结构上修改了列表，那么它*必须* 保持外部同步。（结构上的修改是指任何添加或删除一个或多个元素的操作，或者显式调整底层数组的大小；仅仅设置元素的值不是结构上的修改。）这一般通过对自然封装该列表的对象进行同步操作来完成。如果不存在这样的对象，则应该使用 [Collections.synchronizedList](mk:@MSITStore:C:\Users\Administrator\Desktop\JDK%201.6%20API.chm::/java/util/Collections.html#synchronizedList(java.util.List)) 方法将该列表“包装”起来。这最好在创建时完成，以防止意外对列表进行不同步的访问：

*\* If no such object exists, the list should be "wrapped" using the  
 \* {****@link*** *Collections#synchronizedList Collections.synchronizedList}  
 \* method. This is best done at creation time, to prevent accidental  
 \* unsynchronized access to the list:<pre>  
 \* List list = Collections.synchronizedList(new ArrayList(...));</pre>  
 \*  
 \* <p><a name="fail-fast">  
 \* The iterators returned by this class's {****@link*** *#iterator() iterator} and  
 \* {****@link*** *#listIterator(int) listIterator} methods are <em>fail-fast</em>:</a>  
 \* if the list is structurally modified at any time after the iterator is  
 \* created, in any way except through the iterator's own  
 \* {****@link*** *ListIterator#remove() remove} or  
 \* {****@link*** *ListIterator#add(Object) add} methods, the iterator will throw a  
 \* {****@link*** *ConcurrentModificationException}. Thus, in the face of  
 \* concurrent modification, the iterator fails quickly and cleanly, rather  
 \* than risking arbitrary, non-deterministic behavior at an undetermined  
 \* time in the future.  
 \*  
 \* <p>Note that the fail-fast behavior of an iterator cannot be guaranteed  
 \* as it is, generally speaking, impossible to make any hard guarantees in the  
 \* presence of unsynchronized concurrent modification. Fail-fast iterators  
 \* throw {****@code*** *ConcurrentModificationException} on a best-effort basis.  
 \* Therefore, it would be wrong to write a program that depended on this  
 \* exception for its correctness: <i>the fail-fast behavior of iterators  
 \* should be used only to detect bugs.</i>  
 \*  
 \* <p>This class is a member of the  
 \* <a href="{****@docRoot****}/../technotes/guides/collections/index.html">  
 \* Java Collections Framework</a>.  
 \*  
 \** ***@author*** *Josh Bloch  
 \** ***@author*** *Neal Gafter  
 \** ***@see*** *Collection  
 \** ***@see*** *List  
 \** ***@see*** *LinkedList  
 \** ***@see*** *Vector  
 \** ***@since*** *1.2  
 \*/*

### 成员变量



#### 1：DEFAULT\_CAPATITY:

*/\*\*  
 \* Default initial capacity.  
 \*/***private static final int *DEFAULT\_CAPACITY*** = 10;

#### 2：DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA

*/\*\*  
 \* Shared empty array instance used for default sized empty instances. We  
 \* distinguish this from EMPTY\_ELEMENTDATA to know how much to inflate when  
 \* first element is added.  
 \*/***private static final** Object[] ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*** = {};

#### 3：elementData

*/\*\*  
 \* The array buffer into which the elements of the ArrayList are stored.  
 \* The capacity of the ArrayList is the length of this array buffer. Any  
 \* empty ArrayList with elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA  
 \* will be expanded to DEFAULT\_CAPACITY when the first element is added.  
 \*/***transient** Object[] **elementData**; *// non-private to simplify nested class access*

***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***，与此同时，当地一个元素被添加的时候，elementDate都被扩大为***DEFAULT\_CAPACITY。***

#### 4：EMPTY\_ELEMENTDATA

*/\*\*  
 \* Shared empty array instance used for empty instances.  
 \*/***private static final** Object[] ***EMPTY\_ELEMENTDATA*** = {};

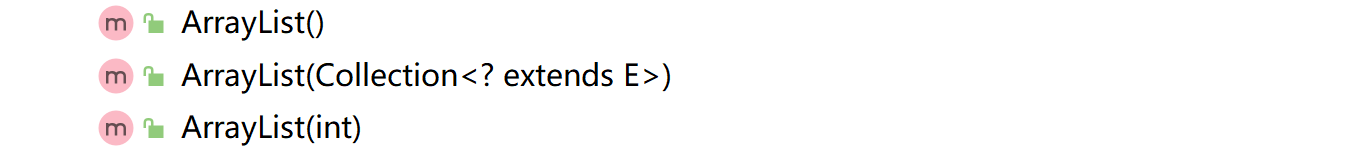
#### 5：MAX\_ARRAY\_SIZE

*/\*\*  
 \* The maximum size of array to allocate.  
 \* Some VMs reserve some header words in an array.  
 \* Attempts to allocate larger arrays may result in  
 \* OutOfMemoryError: Requested array size exceeds VM limit  
 \*/***private static final int *MAX\_ARRAY\_SIZE*** = Integer.***MAX\_VALUE*** - 8;

#### 6：size

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

### 构造方法



#### 1：ArrayList( )

*/\*\*  
 \* Constructs an empty list with an initial capacity of ten.  
 \*/***public** ArrayList() {  
 **this**.**elementData** = ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***;  
}

#### 2：ArrayList(int initialCapacity)

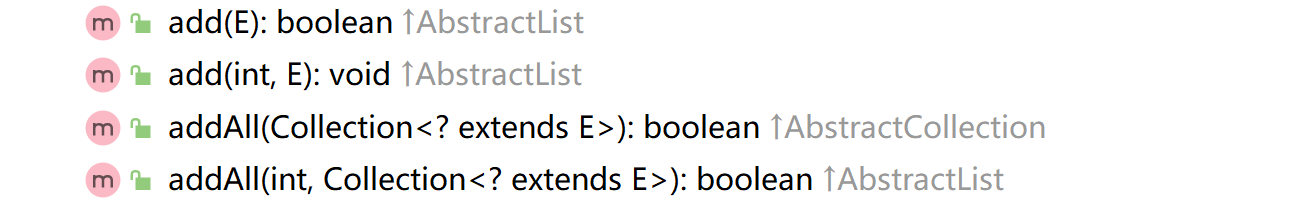
*/\*\*  
 \* Constructs an empty list with the specified initial capacity.  
 \*  
 \** ***@param initialCapacity*** *the initial capacity of the list  
 \** ***@throws*** *IllegalArgumentException if the specified initial capacity  
 \* is negative  
 \*/***public** ArrayList(**int** initialCapacity) {  
 **if** (initialCapacity > 0) {  
 **this**.**elementData** = **new** Object[initialCapacity];  
 } **else if** (initialCapacity == 0) {  
 **this**.**elementData** = ***EMPTY\_ELEMENTDATA***;  
 } **else** {  
 **throw new** IllegalArgumentException(**"Illegal Capacity: "**+initialCapacity);  
 }  
}

#### 3：ArrayList(Collection<? extends E> c

*/\*\*  
 \* 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();  
 **if** ((**size** = **elementData**.**length**) != 0) {  
 *// c.toArray might (incorrectly) not return Object[] (see 6260652)* **if** (**elementData**.getClass() != Object[].**class**)  
 **elementData** = Arrays.*copyOf*(**elementData**, **size**, Object[].**class**);  
 } **else** {  
 *// replace with empty array.* **this**.**elementData** = ***EMPTY\_ELEMENTDATA***;  
 }  
}

### 成员方法

#### 添加：



##### add(E e)

*/\*\*  
 \* Appends the specified element to the end of this list.  
 \*  
 \** ***@param e*** *element to be appended to this list  
 \** ***@return*** *<tt>true</tt> (as specified by {****@link*** *Collection#add})  
 \*/***public boolean** add(E e) {  
 ensureCapacityInternal(**size** + 1); *// Increments modCount!!* **elementData**[**size**++] = e;  
 **return true**;  
}

**private void** ensureCapacityInternal(**int** minCapacity) {  
 ensureExplicitCapacity(*calculateCapacity*(**elementData**, minCapacity));  
}

**private static int** calculateCapacity(Object[] elementData, **int** minCapacity) {  
 **if** (elementData == ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***) {  
 **return** Math.*max*(***DEFAULT\_CAPACITY***, minCapacity);  
 }  
 **return** minCapacity;  
}

**private void** ensureExplicitCapacity(**int** minCapacity) {  
 **modCount**++;  
  
 *// overflow-conscious code* **if** (minCapacity - **elementData**.**length** > 0)  
 grow(minCapacity);  
}

*/\*\*  
 \* The number of times this list has been <i>structurally modified</i>.  
 \* Structural modifications are those that change the size of the  
 \* list, or otherwise perturb it in such a fashion that iterations in  
 \* progress may yield incorrect results.  
 \*  
 \* <p>This field is used by the iterator and list iterator implementation  
 \* returned by the {****@code*** *iterator} and {****@code*** *listIterator} methods.  
 \* If the value of this field changes unexpectedly, the iterator (or list  
 \* iterator) will throw a {****@code*** *ConcurrentModificationException} in  
 \* response to the {****@code*** *next}, {****@code*** *remove}, {****@code*** *previous},  
 \* {****@code*** *set} or {****@code*** *add} operations. This provides  
 \* <i>fail-fast</i> behavior, rather than non-deterministic behavior in  
 \* the face of concurrent modification during iteration.  
 \*  
 \* <p><b>Use of this field by subclasses is optional.</b> If a subclass  
 \* wishes to provide fail-fast iterators (and list iterators), then it  
 \* merely has to increment this field in its {****@code*** *add(int, E)} and  
 \* {****@code*** *remove(int)} methods (and any other methods that it overrides  
 \* that result in structural modifications to the list). A single call to  
 \* {****@code*** *add(int, E)} or {****@code*** *remove(int)} must add no more than  
 \* one to this field, or the iterators (and list iterators) will throw  
 \* bogus {****@code*** *ConcurrentModificationExceptions}. If an implementation  
 \* does not wish to provide fail-fast iterators, this field may be  
 \* ignored.  
 \*/***protected transient int modCount** = 0;

###### private void grow(int minCapacity)

*/\*\*  
 \* Increases the capacity to ensure that it can hold at least the  
 \* number of elements specified by the minimum capacity argument.  
 \*  
 \** ***@param minCapacity*** *the desired minimum capacity  
 \*/***private void** grow(**int** minCapacity) {  
 *// overflow-conscious code* **int** oldCapacity = **elementData**.**length**;  
 **int** newCapacity = oldCapacity + (oldCapacity >> 1);  
 **if** (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;  
 **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  
 newCapacity = *hugeCapacity*(minCapacity);  
 *// minCapacity is usually close to size, so this is a win:* **elementData** = Arrays.*copyOf*(**elementData**, newCapacity);  
}

**private static int** hugeCapacity(**int** minCapacity) {  
 **if** (minCapacity < 0) *// overflow* **throw new** OutOfMemoryError();  
 **return** (minCapacity > ***MAX\_ARRAY\_SIZE***) ? Integer.***MAX\_VALUE*** : ***MAX\_ARRAY\_SIZE***;  
}

*/\*\*  
 \* Copies the specified array, truncating or padding with nulls (if necessary)  
 \* so the copy has the specified length. For all indices that are  
 \* valid in both the original array and the copy, the two arrays will  
 \* contain identical values. For any indices that are valid in the  
 \* copy but not the original, the copy will contain <tt>null</tt>.  
 \* Such indices will exist if and only if the specified length  
 \* is greater than that of the original array.  
 \* The resulting array is of exactly the same class as the original array.  
 \*  
 \** ***@param <T>*** *the class of the objects in the array  
 \** ***@param original*** *the array to be copied  
 \** ***@param newLength*** *the length of the copy to be returned  
 \** ***@return*** *a copy of the original array, truncated or padded with nulls  
 \* to obtain the specified length  
 \** ***@throws*** *NegativeArraySizeException if <tt>newLength</tt> is negative  
 \** ***@throws*** *NullPointerException if <tt>original</tt> is null  
 \** ***@since*** *1.6  
 \*/*@SuppressWarnings(**"unchecked"**)  
**public static** <T> T[] copyOf(T[] original, **int** newLength) {  
 **return** (T[]) *copyOf*(original, newLength, original.getClass());  
}

##### add(int index,E element)

*/\*\*  
 \* Inserts the specified element at the specified position in this  
 \* list. Shifts the element currently at that position (if any) and  
 \* any subsequent elements to the right (adds one to their indices).  
 \*  
 \** ***@param index*** *index at which the specified element is to be inserted  
 \** ***@param element*** *element to be inserted  
 \** ***@throws*** *IndexOutOfBoundsException {****@inheritDoc****}  
 \*/***public void** add(**int** index, E element) {  
 rangeCheckForAdd(index);  
  
 ensureCapacityInternal(**size** + 1); *// Increments modCount!!* System.*arraycopy*(**elementData**, index, **elementData**, index + 1, **size** - index);  
 **elementData**[index] = element;  
 **size**++;  
}

*/\*\*  
 \* A version of rangeCheck used by add and addAll.  
 \*/***private void** rangeCheckForAdd(**int** index) {  
 **if** (index > **size** || index < 0)  
 **throw new** IndexOutOfBoundsException(outOfBoundsMsg(index));  
}

**private void** ensureCapacityInternal(**int** minCapacity) {  
 ensureExplicitCapacity(*calculateCapacity*(**elementData**, minCapacity));  
}

**private static int** calculateCapacity(Object[] elementData, **int** minCapacity) {  
 **if** (elementData == ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***) {  
 **return** Math.*max*(***DEFAULT\_CAPACITY***, minCapacity);  
 }  
 **return** minCapacity;  
}

**private void** ensureExplicitCapacity(**int** minCapacity) {  
 **modCount**++;  
  
 *// overflow-conscious code* **if** (minCapacity - **elementData**.**length** > 0)  
 [grow(minCapacity);](#_private_void_grow(int)  
}

##### allAll(Collection<? extends E> c)

*/\*\*  
 \* Appends all of the elements in the specified collection to the end of  
 \* this list, in the order that they are returned by the  
 \* specified collection's Iterator. The behavior of this operation is  
 \* undefined if the specified collection is modified while the operation  
 \* is in progress. (This implies that the behavior of this call is  
 \* undefined if the specified collection is this list, and this  
 \* list is nonempty.)  
 \*  
 \** ***@param c*** *collection containing elements to be added to this list  
 \** ***@return*** *<tt>true</tt> if this list changed as a result of the call  
 \** ***@throws*** *NullPointerException if the specified collection is null  
 \*/***public boolean** addAll(Collection<? **extends** E> c) {  
 Object[] a = c.toArray();  
 **int** numNew = a.**length**;  
 ensureCapacityInternal(**size** + numNew); *// Increments modCount* System.*arraycopy*(a, 0, **elementData**, **size**, numNew);  
 **size** += numNew;  
 **return** numNew != 0;  
}

##### set(int index, E element)

*/\*\*  
 \* Replaces the element at the specified position in this list with  
 \* the specified element.  
 \*  
 \** ***@param index*** *index of the element to replace  
 \** ***@param element*** *element to be stored at the specified position  
 \** ***@return*** *the element previously at the specified position  
 \** ***@throws*** *IndexOutOfBoundsException {****@inheritDoc****}  
 \*/***public** E set(**int** index, E element) {  
 rangeCheck(index);  
  
 E oldValue = elementData(index);  
 **elementData**[index] = element;  
 **return** oldValue;  
}

#### 删除：

##### remove(int index)

*/\*\*  
 \* Removes the element at the specified position in this list.  
 \* 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 = elementData(index);  
  
 **int** numMoved = **size** - index - 1;  
 **if** (numMoved > 0)  
 System.*arraycopy*(**elementData**, index+1, **elementData**, index,numMoved);  
 **elementData**[--**size**] = **null**; *// clear to let GC do its work* **return** oldValue;  
}

##### remove(Object o)

*/\*\*  
 \* 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&nbsp;?&nbsp;get(i)==null&nbsp;:&nbsp;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**;  
}

##### removeAll(Collection<?> c)

*/\*\*  
 \* Removes from this list all of its elements that are contained in the  
 \* specified collection.  
 \*  
 \** ***@param c*** *collection containing elements to be removed from this list  
 \** ***@return*** *{****@code*** *true} if this list changed as a result of the call  
 \** ***@throws*** *ClassCastException if the class of an element of this list  
 \* is incompatible with the specified collection  
 \* (<a href="Collection.html#optional-restrictions">optional</a>)  
 \** ***@throws*** *NullPointerException if this list contains a null element and the  
 \* specified collection does not permit null elements  
 \* (<a href="Collection.html#optional-restrictions">optional</a>),  
 \* or if the specified collection is null  
 \** ***@see*** *Collection#contains(Object)  
 \*/***public boolean** removeAll(Collection<?> c) {  
 Objects.*requireNonNull*(c);  
 **return** batchRemove(c, **false**);  
}

**private boolean** batchRemove(Collection<?> c, **boolean** complement) {  
 **final** Object[] elementData = **this**.**elementData**;  
 **int** r = 0, w = 0;  
 **boolean** modified = **false**;  
 **try** {  
 **for** (; r < **size**; r++)  
 **if** (c.contains(elementData[r]) == complement)  
 elementData[w++] = elementData[r];  
 } **finally** {  
 *// Preserve behavioral compatibility with AbstractCollection,  
 // even if c.contains() throws.* **if** (r != **size**) {  
 System.*arraycopy*(elementData, r,  
 elementData, w,  
 **size** - r);  
 w += **size** - r;  
 }  
 **if** (w != **size**) {  
 *// clear to let GC do its work* **for** (**int** i = w; i < **size**; i++)  
 elementData[i] = **null**;  
 **modCount** += **size** - w;  
 **size** = w;  
 modified = **true**;  
 }  
 }  
 **return** modified;  
}

##### removeIf(Predicate<? Super E> filter)

@Override  
**public boolean** removeIf(Predicate<? **super** E> filter) {  
 Objects.*requireNonNull*(filter);  
 *// figure out which elements are to be removed  
 // any exception thrown from the filter predicate at this stage  
 // will leave the collection unmodified* **int** removeCount = 0;  
 **final** BitSet removeSet = **new** BitSet(**size**);  
 **final int** expectedModCount = **modCount**;  
 **final int** size = **this**.**size**;  
 **for** (**int** i=0; **modCount** == expectedModCount && i < size; i++) {  
 @SuppressWarnings(**"unchecked"**)  
 **final** E element = (E) **elementData**[i];  
 **if** (filter.test(element)) {  
 removeSet.set(i);  
 removeCount++;  
 }  
 }  
 **if** (**modCount** != expectedModCount) {  
 **throw new** ConcurrentModificationException();  
 }  
  
 *// shift surviving elements left over the spaces left by removed elements* **final boolean** anyToRemove = removeCount > 0;  
 **if** (anyToRemove) {  
 **final int** newSize = size - removeCount;  
 **for** (**int** i=0, j=0; (i < size) && (j < newSize); i++, j++) {  
 i = removeSet.nextClearBit(i);  
 **elementData**[j] = **elementData**[i];  
 }  
 **for** (**int** k=newSize; k < size; k++) {  
 **elementData**[k] = **null**; *// Let gc do its work* }  
 **this**.**size** = newSize;  
 **if** (**modCount** != expectedModCount) {  
 **throw new** ConcurrentModificationException();  
 }  
 **modCount**++;  
 }  
  
 **return** anyToRemove;  
}

#### 查找：

##### get(int index)

*/\*\*  
 \* Returns the element at the specified position in this list.  
 \*  
 \** ***@param index*** *index of the element to return  
 \** ***@return*** *the element at the specified position in this list  
 \** ***@throws*** *IndexOutOfBoundsException {****@inheritDoc****}  
 \*/***public** E get(**int** index) {  
 rangeCheck(index);  
  
 **return** elementData(index);  
}

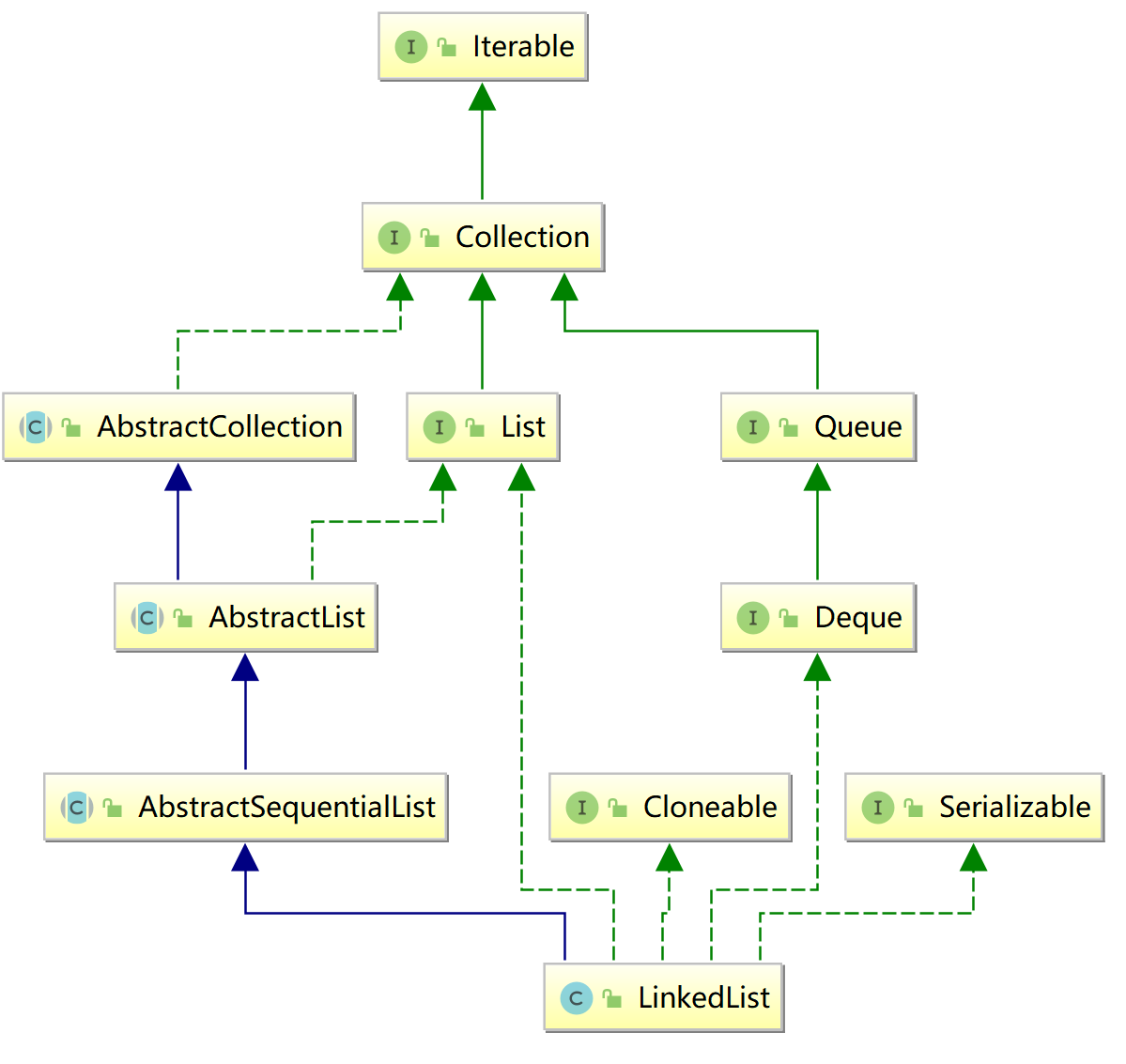
### 私有方法

### 内部类



## LinkedList

### 继承结构图

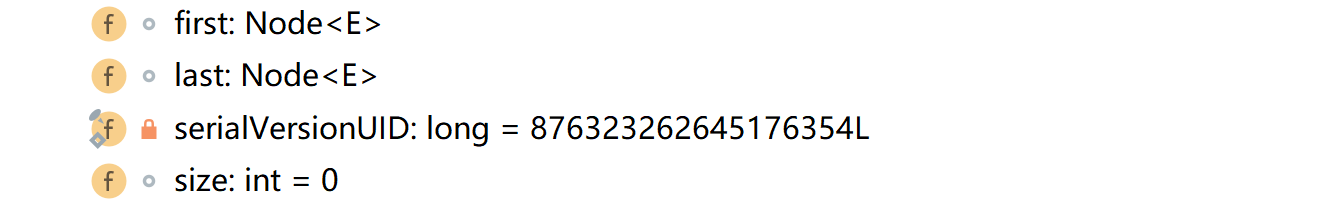


### 描述：

*/\*\*  
 \* Doubly-linked list implementation of the {****@code*** *List} and {****@code*** *Deque} interfaces. Implements all \*optional list operations, and permits all elements (including {****@code*** *null}).  
 \*双链表实现了列表和Deque接口。实现所有可选的列表操作，并允许所有元素(包括null)。*

*\* <p>All of the operations perform as could be expected for a doubly-linked  
 \* list. Operations that index into the list will traverse the list from  
 \* the beginning or the end, whichever is closer to the specified index.  
 \*  
 \* <p><strong>Note that this implementation is not synchronized.</strong>  
 \* If multiple threads access a linked list concurrently, and at least  
 \* one of the threads modifies the list structurally, it <i>must</i> be  
 \* synchronized externally. (A structural modification is any operation  
 \* that adds or deletes one or more elements; merely setting the value of  
 \* an element is not a structural modification.) This is typically  
 \* accomplished by synchronizing on some object that naturally  
 \* encapsulates the list.  
 \*  
 \* If no such object exists, the list should be "wrapped" using the  
 \* {****@link*** *Collections#synchronizedList Collections.synchronizedList}  
 \* method. This is best done at creation time, to prevent accidental  
 \* unsynchronized access to the list:<pre>  
 \* List list = Collections.synchronizedList(new LinkedList(...));</pre>  
 \*  
 \* <p>The iterators returned by this class's {****@code*** *iterator} and  
 \* {****@code*** *listIterator} methods are <i>fail-fast</i>: if the list is  
 \* structurally modified at any time after the iterator is created, in  
 \* any way except through the Iterator's own {****@code*** *remove} or  
 \* {****@code*** *add} methods, the iterator will throw a {****@link*** *\* ConcurrentModificationException}. Thus, in the face of concurrent  
 \* modification, the iterator fails quickly and cleanly, rather than  
 \* risking arbitrary, non-deterministic behavior at an undetermined  
 \* time in the future.  
 \*  
 \* <p>Note that the fail-fast behavior of an iterator cannot be guaranteed  
 \* as it is, generally speaking, impossible to make any hard guarantees in the  
 \* presence of unsynchronized concurrent modification. Fail-fast iterators  
 \* throw {****@code*** *ConcurrentModificationException} on a best-effort basis.  
 \* Therefore, it would be wrong to write a program that depended on this  
 \* exception for its correctness: <i>the fail-fast behavior of iterators  
 \* should be used only to detect bugs.</i>  
 \*  
 \* <p>This class is a member of the  
 \* <a href="{****@docRoot****}/../technotes/guides/collections/index.html">  
 \* Java Collections Framework</a>.  
 \*  
 \** ***@author*** *Josh Bloch  
 \** ***@see*** *List  
 \** ***@see*** *ArrayList  
 \** ***@since*** *1.2  
 \** ***@param <E>*** *the type of elements held in this collection  
 \*/*

### 成员变量



#### 1：Node<E> first

*/\*\*  
 \* Pointer to first node.  
 \* Invariant: (first == null && last == null) ||  
 \* (first.prev == null && first.item != null)  
 \*/***transient** Node<E> **first**;

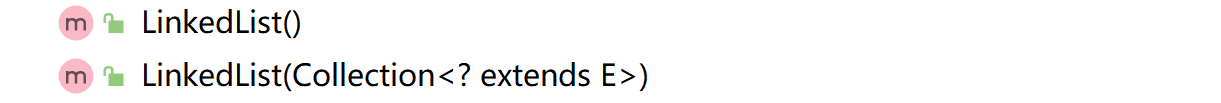
#### 2：Node<E> last

*/\*\*  
 \* Pointer to last node.  
 \* Invariant: (first == null && last == null) ||  
 \* (last.next == null && last.item != null)  
 \*/***transient** Node<E> **last**;

#### 3：size

**transient int size** = 0;

### 构造方法



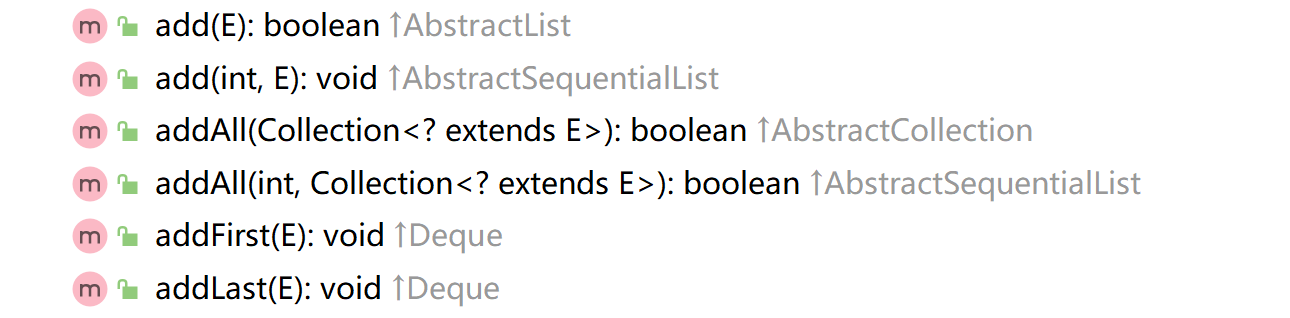
#### LinkedList():

*/\*\*  
 \* Constructs an empty list.  
 \*/***public** LinkedList() {  
}

#### LinkedList(Collection<? Extends E>)

### 成员方法

#### 添加：



##### add(E e)

*/\*\*  
 \* Appends the specified element to the end of this list.  
 \*  
 \* <p>This method is equivalent to {****@link*** *#addLast}.  
 \*  
 \** ***@param e*** *element to be appended to this list  
 \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Collection#add})  
 \*/***public boolean** add(E e) {  
 linkLast(e);  
 **return true**;  
}

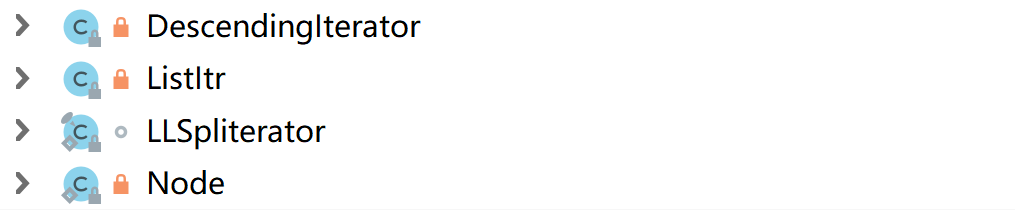
*/\*\*  
 \* Links e as last element.  
 \*/***void** linkLast(E e) {  
 **final** Node<E> l = **last**;  
 **final** Node<E> newNode = **new** Node<>(l, e, **null**);  
 **last** = newNode;  
 **if** (l == **null**)  
 **first** = newNode;  
 **else** l.**next** = newNode;  
 **size**++;  
 **modCount**++;  
}

#### 删除：

#### 查找：

### 私有方法

### 内部类



#### DescendingIterator

*/\*\*  
 \* Adapter to provide descending iterators via ListItr.previous  
 \*/***private class** DescendingIterator **implements** Iterator<E> {  
 **private final** ListItr **itr** = **new** ListItr(size());  
 **public boolean** hasNext() {  
 **return itr**.hasPrevious();  
 }  
 **public** E next() {  
 **return itr**.previous();  
 }  
 **public void** remove() {  
 **itr**.remove();  
 }  
}

#### ListItr

**private class** ListItr **implements** ListIterator<E> {  
 **private** Node<E> **lastReturned**;  
 **private** Node<E> **next**;  
 **private int nextIndex**;  
 **private int expectedModCount** = **modCount**;  
  
 ListItr(**int** index) {  
 *// assert isPositionIndex(index);* **next** = (index == **size**) ? **null** : node(index);  
 **nextIndex** = index;  
 }  
  
 **public boolean** hasNext() {  
 **return nextIndex** < **size**;  
 }  
  
 **public** E next() {  
 checkForComodification();  
 **if** (!hasNext())  
 **throw new** NoSuchElementException();  
  
 **lastReturned** = **next**;  
 **next** = **next**.**next**;  
 **nextIndex**++;  
 **return lastReturned**.**item**;  
 }  
  
 **public boolean** hasPrevious() {  
 **return nextIndex** > 0;  
 }  
  
 **public** E previous() {  
 checkForComodification();  
 **if** (!hasPrevious())  
 **throw new** NoSuchElementException();  
  
 **lastReturned** = **next** = (**next** == **null**) ? **last** : **next**.**prev**;  
 **nextIndex**--;  
 **return lastReturned**.**item**;  
 }  
  
 **public int** nextIndex() {  
 **return nextIndex**;  
 }  
  
 **public int** previousIndex() {  
 **return nextIndex** - 1;  
 }  
  
 **public void** remove() {  
 checkForComodification();  
 **if** (**lastReturned** == **null**)  
 **throw new** IllegalStateException();  
  
 Node<E> lastNext = **lastReturned**.**next**;  
 unlink(**lastReturned**);  
 **if** (**next** == **lastReturned**)  
 **next** = lastNext;  
 **else  
 nextIndex**--;  
 **lastReturned** = **null**;  
 **expectedModCount**++;  
 }  
  
 **public void** set(E e) {  
 **if** (**lastReturned** == **null**)  
 **throw new** IllegalStateException();  
 checkForComodification();  
 **lastReturned**.**item** = e;  
 }  
  
 **public void** add(E e) {  
 checkForComodification();  
 **lastReturned** = **null**;  
 **if** (**next** == **null**)  
 linkLast(e);  
 **else** linkBefore(e, **next**);  
 **nextIndex**++;  
 **expectedModCount**++;  
 }  
  
 **public void** forEachRemaining(Consumer<? **super** E> action) {  
 Objects.*requireNonNull*(action);  
 **while** (**modCount** == **expectedModCount** && **nextIndex** < **size**) {  
 action.accept(**next**.**item**);  
 **lastReturned** = **next**;  
 **next** = **next**.**next**;  
 **nextIndex**++;  
 }  
 checkForComodification();  
 }  
  
 **final void** checkForComodification() {  
 **if** (**modCount** != **expectedModCount**)  
 **throw new** ConcurrentModificationException();  
 }  
}

#### LLSpliterator

*/\*\* A customized variant of Spliterators.IteratorSpliterator \*/***static final class** LLSpliterator<E> **implements** Spliterator<E> {  
 **static final int *BATCH\_UNIT*** = 1 << 10; *// batch array size increment* **static final int *MAX\_BATCH*** = 1 << 25; *// max batch array size;* **final** LinkedList<E> **list**; *// null OK unless traversed* Node<E> **current**; *// current node; null until initialized* **int est**; *// size estimate; -1 until first needed* **int expectedModCount**; *// initialized when est set* **int batch**; *// batch size for splits* LLSpliterator(LinkedList<E> list, **int** est, **int** expectedModCount) {  
 **this**.**list** = list;  
 **this**.**est** = est;  
 **this**.**expectedModCount** = expectedModCount;  
 }  
  
 **final int** getEst() {  
 **int** s; *// force initialization* **final** LinkedList<E> lst;  
 **if** ((s = **est**) < 0) {  
 **if** ((lst = **list**) == **null**)  
 s = **est** = 0;  
 **else** {  
 **expectedModCount** = lst.**modCount**;  
 **current** = lst.**first**;  
 s = **est** = lst.**size**;  
 }  
 }  
 **return** s;  
 }  
  
 **public long** estimateSize() { **return** (**long**) getEst(); }  
  
 **public** Spliterator<E> trySplit() {  
 Node<E> p;  
 **int** s = getEst();  
 **if** (s > 1 && (p = **current**) != **null**) {  
 **int** n = **batch** + ***BATCH\_UNIT***;  
 **if** (n > s)  
 n = s;  
 **if** (n > ***MAX\_BATCH***)  
 n = ***MAX\_BATCH***;  
 Object[] a = **new** Object[n];  
 **int** j = 0;  
 **do** { a[j++] = p.**item**; } **while** ((p = p.**next**) != **null** && j < n);  
 **current** = p;  
 **batch** = j;  
 **est** = s - j;  
 **return** Spliterators.*spliterator*(a, 0, j, Spliterator.***ORDERED***);  
 }  
 **return null**;  
 }  
  
 **public void** forEachRemaining(Consumer<? **super** E> action) {  
 Node<E> p; **int** n;  
 **if** (action == **null**) **throw new** NullPointerException();  
 **if** ((n = getEst()) > 0 && (p = **current**) != **null**) {  
 **current** = **null**;  
 **est** = 0;  
 **do** {  
 E e = p.**item**;  
 p = p.**next**;  
 action.accept(e);  
 } **while** (p != **null** && --n > 0);  
 }  
 **if** (**list**.**modCount** != **expectedModCount**)  
 **throw new** ConcurrentModificationException();  
 }  
  
 **public boolean** tryAdvance(Consumer<? **super** E> action) {  
 Node<E> p;  
 **if** (action == **null**) **throw new** NullPointerException();  
 **if** (getEst() > 0 && (p = **current**) != **null**) {  
 --**est**;  
 E e = p.**item**;  
 **current** = p.**next**;  
 action.accept(e);  
 **if** (**list**.**modCount** != **expectedModCount**)  
 **throw new** ConcurrentModificationException();  
 **return true**;  
 }  
 **return false**;  
 }  
  
 **public int** characteristics() {  
 **return** Spliterator.***ORDERED*** | Spliterator.***SIZED*** | Spliterator.***SUBSIZED***;  
 }  
}

#### Node

**private static class** Node<E> {  
 E **item**;  
 Node<E> **next**;  
 Node<E> **prev**;  
  
 Node(Node<E> prev, E element, Node<E> next) {  
 **this**.**item** = element;  
 **this**.**next** = next;  
 **this**.**prev** = prev;  
 }  
}

## 1111

### 继承结构图

### 成员变量

### 构造方法

### 成员方法

#### 添加：

#### 删除：

#### 查找：

### 私有方法

### 内部类

## 1111

### 继承结构图

### 成员变量

### 构造方法

### 成员方法

#### 添加：

#### 删除：

#### 查找：

### 私有方法

### 内部类

## 1111

### 继承结构图

### 成员变量

### 构造方法

### 成员方法

#### 添加：

#### 删除：

#### 查找：

### 私有方法

### 内部类

## 1111

### 继承结构图

### 成员变量

### 构造方法

### 成员方法

#### 添加：

#### 删除：

#### 查找：

### 私有方法

### 内部类

## 1111

### 继承结构图

### 成员变量

### 构造方法

### 成员方法

#### 添加：

#### 删除：

#### 查找：

### 私有方法

### 内部类

## 1111

### 继承结构图

### 成员变量

### 构造方法

### 成员方法

#### 添加：

#### 删除：

#### 查找：

### 私有方法

### 内部类