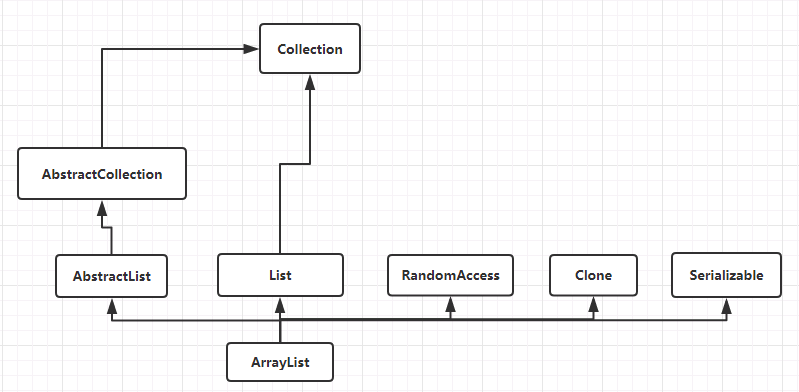
# ArrayList

## 简介

动态数组，底层是一个数组，主要使用来装载数据，但是不能装载基本数据类型，只能装载基本数据类型对应的包装类，对ArrayList的操作都是在操作数组；它的查找和访问元素的速度较快，但新增，删除消息可能较低，（**当数组到达最大容量时，新增就会先进行数组扩容，也就是将原数据复制到新的数组上；删除时，如果不是删除最后下标的数据，那么就会将删除为之后的数据向前移动1位。**）提供动态增加和删除元素，实现list接口；线程是不安全的



源码分析

java.util.AbstractCollection<E>抽象类，提供了Collection接口的骨干实现，最大限度的减少实现此接口所需的工作

如果要实现一个不可修改的collection，只需要扩展此类，并实现iterator和size方法即可，

如果要实现一个可修改的collection，还需要重写此类的add方法，因为这个add方法是直接抛出UnsupportedOperationException异常

java.util.AbstractList<E> 提供list接口的骨干实现，从而实现最大限度的减少了实现由随机访问数据存储（比如数组）支持的接口需要的工作，对于连续的访问数据（如链表），应优先使用[AbstractSequentialList](https://baike.baidu.com/item/AbstractSequentialList" \t "_blank)，而非此类。LikedList就是继承AbstractSequentialList

要实现不可修改的列表，只需要扩展此类，并提供get(index)和size()方法即可

要实现可修改的列表，那么就需要重写 add,remove set方法 这些方法在AbstractList中都是直接抛出UnsupportedOperationException异常（不支持操作异常）

## ArrayList成员变量

**private** **static** **final** **int** *DEFAULT\_CAPACITY* = 10;容器默认初始容量

**private** **static** **final** Object[] *EMPTY\_ELEMENTDATA* = {};当调用非无参构造函数时，指定的初始化长度为0时将这个对象赋值给elementData原数组

**private** **static** **final** Object[] *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA* = {};调用无参构造函数时将这个对象赋值给elementData原数组

**transient** Object[] elementData; 用来存储数据的数组元素数组

**private** **int** size;数组中存放数据的个数，包括添加null的个数

private static final int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;数组的最大容量 如果数组长度超过这个值，那么可能抛出outofmemoryerror(内存溢出异常)

protected transient int modCount = 0;定义在AbstractList中，用来记录新增删除节点操作的次数

## 构造函数

无参构造函数

**public** ArrayList() {

调用无参构造函数时将*DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*这个对象赋值给elementData原数组

**this**.elementData = *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*;

}

带参构造函数，参数是int类型的，用来指定初始化数组长度

**public** ArrayList(**int** initialCapacity) {

**if** (initialCapacity > 0) {

**this**.elementData = **new** Object[initialCapacity];

} **else** **if** (initialCapacity == 0) {

参数为0，数组长度为0，

**this**.elementData = *EMPTY\_ELEMENTDATA*;

} **else** { 参数为负，抛出异常

**throw** **new** IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

}

}

带参构造函数，参数是一个collection集合

**public** ArrayList(Collection<? **extends** E> c) {

集合对象调用toArray()方法将集合转换为数组并且赋值给元数组

elementData = c.toArray();

**if** ((size = elementData.length) != 0) {

// c.toArray might (incorrectly) not return Object[] (see 6260652)

**if** (elementData.getClass() != Object[].**class**)当不是正确的数组时

elementData = Arrays.*copyOf*(elementData, size, Object[].**class**);

} **else** {

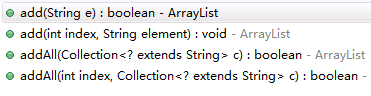
// replace with empty array.

**this**.elementData = *EMPTY\_ELEMENTDATA*;

}

}

## 添加元素方法



add(e):在数组的最后添加对象

add(index,e)在数组的指定位置添加对象

addAll(collection)将collection集合添加到当前数组中，从最后位置开始

addAll(index,collection)，从指定位置添加集合，并且之后的对象向后为集合长度位

添加过程中如果添加的对象个数超过了数组的长度，那么首先要对数组进行扩容

### add(e)方法

**public** **boolean** add(E e) {

ensureCapacityInternal方法判断是否对数组进行扩容，如果添加后数组的容量超过了当前数组的最大容量，那么就要对数组进行扩容，参数表示数组的对象容量

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;在size+1位置上赋值

**return** **true**;

}

ensureCapacityInternal方法判断是否对数组进行扩容

minCapacity表示数组的最小容量

**private** **void** ensureCapacityInternal(**int** minCapacity) {

**if** (elementData == *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*) {

表示使用无参构造函数初始化ArrayList对象，那么要设置源数组的初始容量

minCapacity = Math.*max*(*DEFAULT\_CAPACITY*, minCapacity);

}

ensureExplicitCapacity(minCapacity);

}

minCapacity表示数组的最小容量

**private** **void** ensureExplicitCapacity(**int** minCapacity) {

modCount++;记录操作次数

// overflow-conscious code

**if** (minCapacity - elementData.length > 0)

如果数组的容量大于了当前数组的最大容量那么就要对数组进行扩容，扩容在grow方法中进行

grow(minCapacity);

}

**private** **void** grow(**int** minCapacity) {

// overflow-conscious code

**int** oldCapacity = elementData.length;

新容量原容量的1.5倍

**int** newCapacity = oldCapacity + (oldCapacity >> 1);

**if** (newCapacity - minCapacity < 0)

newCapacity = minCapacity;当数组的最小容量大于扩容后的容量，就将数组最小容量值赋值给新容量

**if** (newCapacity - *MAX\_ARRAY\_SIZE* > 0)当新容量大于最大数组长度时，那数组的长度就为int的最大值

newCapacity = *hugeCapacity*(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.*copyOf*(elementData, newCapacity);将当前数组拷贝到扩容后的数组上，将扩容后的数组赋值给元数组

}

### add(index,e)方法

add(index,e)在链表上的指定位置插入数据

**public** **void** add(**int** index, E element) {

校验index，只能在已经添加过数据的位置上插入

rangeCheckForAdd(index);

判断是否要扩容

ensureCapacityInternal(size + 1); // Increments modCount!!

首先将插入位及其之后的数组向后移动一位，然后将插入位的值赋值为指定的值

System.*arraycopy*(elementData, index, elementData, index + 1,

size - index);

elementData[index] = element;

size++;

}

### addAll(c)方法

addAll(c):添加集合对象

**public** **boolean** addAll(Collection<? **extends** E> c) {

Object[] a = c.toArray();将集合对象转换为数组，然后获取这个数组的长度

**int** numNew = a.length;获取长度

集合数组的长度加上当前elementData数组实际容量就是数组最新的容量

判断这个容量是否大于elementData数组的长度，大于的话就进行扩容

ensureCapacityInternal(size + numNew); // Increments modCount

然后将集合数组从0位开始向elementData数组的第size为开始拷贝数据，拷贝的长度时集合数组的长度

System.*arraycopy*(a, 0, elementData, size, numNew);

size += numNew;数组最新的实际容量

**return** numNew != 0;

}

## 删除元素

list.remove(index);删除指定位置的数据

list.remove(Object);删除指定的数据

list.removeAll(Collection)删除指定的集合

remove(index);删除指定为的数据，方法返回删除的数据

**public** E remove(**int** index) {

rangeCheck(index);校验index，index应该小于size；不在这个范围抛出异常

modCount++;操作次数+1

E oldValue = elementData(index);获取index位置的数据

**int** numMoved = size - index - 1;判断index之后的元素个数，如果为0，表示在最后删除

**if** (numMoved > 0)

个数大于0，不在最末尾删除，那么就将index后的元素全部向前移动一位

System.*arraycopy*(elementData, index+1, elementData, index,

numMoved);

size减1，并且将最后一位设置为null

elementData[--size] = **null**; // clear to let GC do its work

**return** oldValue;//返回删除位置的数据

}

remove(Object)删除指定的数据，删除成功返回true，没有找到要删除的数据返回false

**public** **boolean** remove(Object o) {

**if** (o == **null**) {

o为null遍历elementData数组，遍历从0到size-1位 ，如果找到值为null的节点，就执行删除那个节点

并且返回true，

**for** (**int** index = 0; index < size; index++)

**if** (elementData[index] == **null**) {

fastRemove(index);

**return** **true**;

}

} **else** {

和上面类似遍历elementData数组，找到和o相等的元素进行删除，然后返回true

**for** (**int** index = 0; index < size; index++)

**if** (o.equals(elementData[index])) {

fastRemove(index);

**return** **true**;

}

}

如果没有找到要删除的元素，那么就remove(object)方法就放回false

**return** **false**;

}

删除方法，和remove(int)删除指定位置的元素方法 类似，只是这是个void方法

**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**; // clear to let GC do its work

}

removeAll(Collection),删除在集合中存在的元素

**public** **boolean** removeAll(Collection<?> c) {

Objects.*requireNonNull*(c);集合为空抛出异常

**return** batchRemove(c, **false**);具体实现删除的过程

}

Complement值为false

**private** **boolean** batchRemove(Collection<?> c, **boolean** complement) {

**final** Object[] elementData = **this**.elementData;原数组

**int** r = 0, w = 0;

**boolean** modified = **false**;

**try** {

遍历原数组，找到原数组不在集合中的元素，将这些元素从0为开始保存在原数组上（会覆盖一些位置的数据被覆盖的就是要删除的），将不需要删除的元素全部转移到数组从0到w-1位置，然后将w之后的全部设置为null就完成删除

**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) {在判断集合中是否存在数组中的数据时出现异常就这些这段代码，一般不会出现

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;删除的个数

size = w;

modified = **true**;

}

}

**return** modified;如果有删除就返回true，没有就返回false

}

获取指定位置的元素get(index)

index在size范围内，就直接获取

elementData[index]

修改指定位置的元素set(index,e)

index在size范围内，直接给index赋值e即可

elementData[index]=e

## sort方法

对ArrayList进行排序，sort方法，要自定义一个实现java.util.Comparator<? super E>的比较器

作为参数传入sort

**public** **void** sort(Comparator<? **super** E> c) {

**final** **int** expectedModCount = modCount;

Arrays.*sort*((E[]) elementData, 0, size, c);

**if** (modCount != expectedModCount) {

**throw** **new** ConcurrentModificationException();

}

modCount++;

}

## 数组扩容实现

System类的arraycopy方法实现数组复制扩容

从源数组的srcPos位置开发向目标数组dest的destPos位置开发复制length位值，

public static native void arraycopy(

Object src, 原数组

int srcPos,原数组起始位置

Object dest, 目标数组

int destPos,目标数组的起始位置

int length 拷贝的长度

);

Arrays类封装了System.arraycopy方法，

传入源数组，和拷贝的长度

public static <T> T[] copyOf(T[] original, int newLength) {

源数组的class对象作为创建目标数组的class对象

return (T[]) copyOf(original, newLength, original.getClass());

}

传入源数组，和拷贝的长度，以及目标数组的class类型

public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {

更具传入的class类型创建目标数组，然后实现数组拷贝

@SuppressWarnings("unchecked")

T[] copy = ((Object)newType == (Object)Object[].class)

? (T[]) new Object[newLength]

: (T[]) Array.newInstance(newType.getComponentType(), newLength);

System.arraycopy(original, 0, copy, 0,

Math.min(original.length, newLength));

return copy;

}

## 迭代器

ArrayList实现了接口java.util.Iterable接口，重写iterator方法这个方法返回实现Iterator的迭代实例对象

ArrayList的内部类Itr，实现Iterator接口，ArrayList对象通过iterator（）方法创建这个类的对象

**private** **class** Itr **implements** Iterator<E>

**int** cursor;指针，指向下一个返回数据的位置索引，初始化的时候默认值为0，索引位置从0开始

**int** lastRet = -1; 最后一个返回数据的位置，当没有赋值时，值为-1

**int** expectedModCount = modCount;记录elementData数组添加或者删除节点的次数

获取当前节点数据，指向下一个节点

**public** E next() {

checkForComodification();比较期望的修改节点次数和当前的修改次数，如果不相等抛出异常ConcurrentModificationException

**int** i = cursor;

**if** (i >= size)指针索引大于size,超过了当前数组的实际容量，抛出异常

**throw** **new** NoSuchElementException();

Object[] elementData = ArrayList.**this**.elementData;

**if** (i >= elementData.length) 指针索引超过了当前数组的最大容量，抛出异常

**throw** **new** ConcurrentModificationException();

cursor = i + 1;修改指针，表示指向下一个位置

**return** (E) elementData[lastRet = i];获取当前位置的数据

}

校验modCount和expectedModCount

**final** **void** checkForComodification() {

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

}

Itr的remove方法

**public** **void** remove() {

当Itr初始化时lastRet的值为-1，直接调用remove方法后抛出异常

**if** (lastRet < 0)

**throw** **new** IllegalStateException();

checkForComodification();比较期望的修改节点次数和当前的修改次数，如果不相等抛出异常ConcurrentModificationException

**try** {

调用ArrayList的remove(index)方法（根据位置删除节点的方法）

ArrayList.**this**.remove(lastRet);

cursor = lastRet; lastRet赋值给cusor，所以指针cursor的值一直是0

lastRet = -1;lastRet赋值为-1，所有值迭代对象的循环体中直接执行remove方法都会抛出异常，在执行删除方法之前必须调用next方法，让lastRet指向当前指针位置

expectedModCount = modCount;

} **catch** (IndexOutOfBoundsException ex) {

**throw** **new** ConcurrentModificationException();

}

}

ArrayList的另一个内部类ListItr继承了Itr,实现了ListIterator接口

ListItr提供了一个带参构造函数，参数是int类型

ListItr(**int** index) {

**super**();

cursor = index;用来给Itr的cursor赋值，这样就可以指定指针指向的初始位置

}

ListItr还提供了add,set等方法，

ArrayList提供了2个方法来创建这个类的对象：listIterator()，listIterator(index)

重载listIterator方法，带参数的指定cursor

**public** ListIterator<E> listIterator() {

**return** **new** ListItr(0);

}

**public** ListIterator<E> listIterator(**int** index) {

**if** (index < 0 || index > size)

**throw** **new** IndexOutOfBoundsException("Index: "+index);

**return** **new** ListItr(index);

}

ArrayList实现了Iterable接口，那么就可以使用foreach循环体遍历数据，ArrayList对象使用foreach循环体时，内部会调用iterator方法创建Iterator的实现类Itr类的对象，每循环一次就调用一次Itr的next（）方法，依次返回数组每个位置上的对象

**for** (String string : list) string就是Itr类对象调用next方法返回的

{

System.*out*.println(string);

}

for循环遍历ArrayList比用foreach遍历ArrayList快，for循环遍历，使用get(index)方法获取数组节点对象，直接通过数组的下标就可以返回对象

foreach通过Itr对象的next()方法来获取对象，在方法内部要进行指针判断，指针修改等操作，所有效率相对而言会低一些

在foreach循环中add或者remove，会抛出java.util.ConcurrentModificationException

在使用foreach循环时每次都会调用next（）方法获取对象，在next方法中会验证操作次数和期望的是否相等，期望的在初始化话后没有修改那么就不会改变，执行异常remove后，实际修改次数已经变了，所以在下次执行next方法时就会抛出异常，add方法同理，所以不要在ArrayList的foreach中执行add或者remove方法；

可以使用迭代对象的remove方法

**while** (iterator.hasNext())

{

iterator.next();必须要先调用next方法给lastRet赋值，当Itr初始化时lastRet的值为-1，直接调用remove方法后抛出异常

iterator.remove();

}

循环遍历ArrayList集合时，for和foreach的耗时时间差不多，多次测试可以发现，for循环效率比foreach的效率还要高一些；

循环遍历LinkedList集合时，for循环的耗时明显高于foreach循环的耗时。

在不确定长度或者计算长度有损性能的时候用foreach比较方便；当遍历链表结构的集合时一定不要用for循环。

RandomAccess接口是一个标记接口，实现这个接口的list集合支持快速随机访问官网还特意说明了，如果是实现了这个接口的 **List**，那么使用for循环的方式获取数据会优于用迭代器获取数据