ArrayList是Java List类型的集合类中最常使用的，本文基于Java1.8，对于ArrayList的实现原理做一下详细讲解。

（Java1.8源码：<http://docs.oracle.com/javase/8/docs/api/>）

**一、ArrayList实现原理总结**

ArrayList的实现原理总结如下：

①数据存储是基于**数组**实现的，默认初始容量为10；

②添加数据时，首先需要检查元素个数是否超过数组容量，如果超过了则需要对数组进行扩容；插入数据时，需要将插入点k开始到数组末尾的数据全部向后移动一位。

③数组的扩容是新建一个大容量（原始数组大小+扩充容量）的数组，然后将原始数组数据拷贝到新数组，然后将新数组作为扩容之后的数组。数组扩容的操作代价很高，我们应该尽量减少这种操作。

④删除数据时，需要将删除点+1位置开始到数组末尾的数据全部向前移动一位。

⑤获取数据很快，根据数组下表可以直接获取。

二、ArrayList的实现原理详解

（转自：http://zhangshixi.iteye.com/blog/674856）

1. ArrayList概述：

   ArrayList是List接口的可变数组的实现。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。  
   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。自动增长会带来数据向新数组的重新拷贝，因此，如果可预知数据量的多少，可在构造ArrayList时指定其容量。在添加大量元素前，应用程序也可以使用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。  
   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。

2. ArrayList的实现：

   对于ArrayList而言，它实现List接口、底层使用数组保存所有元素。其操作基本上是对数组的操作。下面我们来分析ArrayList的源代码：

   1) 底层使用数组实现：

Java代码  收藏代码

1. **private** **transient** Object[] elementData;

    2) 构造方法：  
   ArrayList提供了三种方式的构造器，可以构造一个默认初始容量为10的空列表、构造一个指定初始容量的空列表以及构造一个包含指定collection的元素的列表，这些元素按照该collection的迭代器返回它们的顺序排列的。

Java代码  收藏代码

1. **public** ArrayList() {
2. **this**(10);
3. }
5. **public** ArrayList(**int** initialCapacity) {
6. **super**();
7. **if** (initialCapacity < 0)
8. **throw** **new** IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
9. **this**.elementData = **new** Object[initialCapacity];
10. }
12. **public** ArrayList(Collection<? **extends** E> c) {
13. elementData = c.toArray();
14. size = elementData.length;
15. // c.toArray might (incorrectly) not return Object[] (see 6260652)
16. **if** (elementData.getClass() != Object[].**class**)
17. elementData = Arrays.copyOf(elementData, size, Object[].**class**);
18. }

    3) 存储：  
   ArrayList提供了set(int index, E element)、add(E e)、add(int index, E element)、addAll(Collection<? extends E> c)、addAll(int index, Collection<? extends E> c)这些添加元素的方法。下面我们一一讲解：

Java代码  收藏代码

1. // 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。
2. **public** E set(**int** index, E element) {
3. RangeCheck(index);
5. E oldValue = (E) elementData[index];
6. elementData[index] = element;
7. **return** oldValue;
8. }

Java代码  收藏代码

1. // 将指定的元素添加到此列表的尾部。
2. **public** **boolean** add(E e) {
3. ensureCapacity(size + 1);
4. elementData[size++] = e;
5. **return** **true**;
6. }

Java代码  收藏代码

1. // 将指定的元素插入此列表中的指定位置。
2. // 如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（将其索引加1）。
3. **public** **void** add(**int** index, E element) {
4. **if** (index > size || index < 0)
5. **throw** **new** IndexOutOfBoundsException("Index: "+index+", Size: "+size);
6. // 如果数组长度不足，将进行扩容。
7. ensureCapacity(size+1);  // Increments modCount!!
8. // 将 elementData中从Index位置开始、长度为size-index的元素，
9. // 拷贝到从下标为index+1位置开始的新的elementData数组中。
10. // 即将当前位于该位置的元素以及所有后续元素右移一个位置。
11. System.arraycopy(elementData, index, elementData, index + 1, size - index);
12. elementData[index] = element;
13. size++;
14. }

Java代码  收藏代码

1. // 按照指定collection的迭代器所返回的元素顺序，将该collection中的所有元素添加到此列表的尾部。
2. **public** **boolean** addAll(Collection<? **extends** E> c) {
3. Object[] a = c.toArray();
4. **int** numNew = a.length;
5. ensureCapacity(size + numNew);  // Increments modCount
6. System.arraycopy(a, 0, elementData, size, numNew);
7. size += numNew;
8. **return** numNew != 0;
9. }

Java代码  收藏代码

1. // 从指定的位置开始，将指定collection中的所有元素插入到此列表中。
2. **public** **boolean** addAll(**int** index, Collection<? **extends** E> c) {
3. **if** (index > size || index < 0)
4. **throw** **new** IndexOutOfBoundsException(
5. "Index: " + index + ", Size: " + size);
7. Object[] a = c.toArray();
8. **int** numNew = a.length;
9. ensureCapacity(size + numNew);  // Increments modCount
11. **int** numMoved = size - index;
12. **if** (numMoved > 0)
13. System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
15. System.arraycopy(a, 0, elementData, index, numNew);
16. size += numNew;
17. **return** numNew != 0;
18. }

    4) 读取：

Java代码  收藏代码

1. // 返回此列表中指定位置上的元素。
2. **public** E get(**int** index) {
3. RangeCheck(index);
5. **return** (E) elementData[index];
6. }

    5) 删除：  
   ArrayList提供了根据下标或者指定对象两种方式的删除功能。如下：

Java代码  收藏代码

1. // 移除此列表中指定位置上的元素。
2. **public** E remove(**int** index) {
3. RangeCheck(index);
5. modCount++;
6. E oldValue = (E) elementData[index];
8. **int** numMoved = size - index - 1;
9. **if** (numMoved > 0)
10. System.arraycopy(elementData, index+1, elementData, index, numMoved);
11. elementData[--size] = **null**; // Let gc do its work
13. **return** oldValue;
14. }

Java代码  收藏代码

1. // 移除此列表中首次出现的指定元素（如果存在）。这是应为ArrayList中允许存放重复的元素。
2. **public** **boolean** remove(Object o) {
3. // 由于ArrayList中允许存放null，因此下面通过两种情况来分别处理。
4. **if** (o == **null**) {
5. **for** (**int** index = 0; index < size; index++)
6. **if** (elementData[index] == **null**) {
7. // 类似remove(int index)，移除列表中指定位置上的元素。
8. fastRemove(index);
9. **return** **true**;
10. }
11. } **else** {
12. **for** (**int** index = 0; index < size; index++)
13. **if** (o.equals(elementData[index])) {
14. fastRemove(index);
15. **return** **true**;
16. }
17. }
18. **return** **false**;
19. }

    注意：从数组中移除元素的操作，也会导致被移除的元素以后的所有元素的向左移动一个位置。  
   6) 调整数组容量：  
   从上面介绍的向ArrayList中存储元素的代码中，我们看到，每当向数组中添加元素时，都要去检查添加后元素的个数是否会超出当前数组的长度，如果超出，数组将会进行扩容，以满足添加数据的需求。数组扩容通过一个公开的方法ensureCapacity(int minCapacity)来实现。在实际添加大量元素前，我也可以使用ensureCapacity来手动增加ArrayList实例的容量，以减少递增式再分配的数量。

Java代码  收藏代码

1. **public** **void** ensureCapacity(**int** minCapacity) {
2. modCount++;
3. **int** oldCapacity = elementData.length;
4. **if** (minCapacity > oldCapacity) {
5. Object oldData[] = elementData;
6. **int** newCapacity = (oldCapacity \* 3)/2 + 1;
7. **if** (newCapacity < minCapacity)
8. newCapacity = minCapacity;
9. // minCapacity is usually close to size, so this is a win:
10. elementData = Arrays.copyOf(elementData, newCapacity);
11. }
12. }

   从上述代码中可以看出，数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长大约是其原容量的1.5倍。这种操作的代价是很高的，因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。  
   ArrayList还给我们提供了将底层数组的容量调整为当前列表保存的实际元素的大小的功能。它可以通过trimToSize方法来实现。代码如下：

Java代码  收藏代码

1. **public** **void** trimToSize() {
2. modCount++;
3. **int** oldCapacity = elementData.length;
4. **if** (size < oldCapacity) {
5. elementData = Arrays.copyOf(elementData, size);
6. }
7. }

   7) Fail-Fast机制：  
ArrayList也采用了快速失败的机制，通过记录modCount参数来实现。在面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。具体介绍请参考我之前的文章[深入Java集合学习系列：HashMap的实现原理](http://zhangshixi.iteye.com/blog/672697)中的Fail-Fast机制。  
   8) 关于其他的一些方法的实现都很简单易懂，读者可参照API文档和源代码，一看便知，这里就不再多说。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ArrayList是List里面使用率最高的。

　　package collection.lession7;

|  |  |
| --- | --- |
|  | import java.util.ArrayList;  import java.util.Arrays;  import java.util.Collection;  import java.util.Iterator;  import java.util.List;    public class Lession7 {    　public static void main(String[] args) {  　　testNormal();  　　testSpecial();  　　// 一个最常见的错误  　　testForProblem();  　}    　public static void testNormal() {  　　// -------------------------------------------------------  　　// 声明一个列表  　　// 允许放入任何数据  　　// -------------------------------------------------------  　　ArrayList list = new ArrayList();  　　// 放入整数  　　// 当然你用 new Integer(1)也可以  　　list.add(1);  　　// 放入字符串  　　list.add("abc");  　　// 放入浮点数  　　list.add(new Float(1.11));  　　// add会将数据保存到列表的尾部  　　showList(list); // 1, abc, 1.11]    　　// 下面我们在列表的头部增加数据  　　list.add(0, 2);  　　list.add(0, "bcd");  　　list.add(0, new Double(2.34));  　　// 列表可以指定插入的位置  　　// 0 是头部第一个位置，所以数据都逐个放到最前面了  　　showList(list); // [2.34, bcd, 2, 1, abc, 1.11]    　　// 下面我们插入到我们希望的任何位置  　　// 当然不能越界，(0 到 list.size()-1)范围内才可以  　　list.add(1, 3);  　　list.add(4, "xyz");  　　// 数据被放到了正确的位置  　　showList(list); // [2.34, 3, bcd, 2, xyz, 1, abc, 1.11]    　　// -------------------------------------------------------  　　// 我们有了数据，我们来测试读取数据  　　// -------------------------------------------------------  　　// 我们可以通过指定索引的位置，来拿到我们希望的数据  　　System.out.println(list.get(0)); // 2.34  　　System.out.println(list.get(4)); // xyz    　　// -------------------------------------------------------  　　// 测试是否存在某个数据  　　// -------------------------------------------------------  　　System.out.println(list.contains("xyz")); // true    　　// 测试是否包含一组数据  　　Collection c = new ArrayList();  　　c.add(1);  　　c.add(2);  　　System.out.println(list.containsAll(c)); // true  　　c.add(3);  　　c.add(4);  　　// containsAll\_1234=false  　　System.out.println(list.containsAll(c)); // false    　　// -------------------------------------------------------  　　// 查找某个数据所在的索引位置  　　// 如果不存在，返回-1  　　// -------------------------------------------------------  　　System.out.println(list.indexOf(3)); // 1  　　System.out.println(list.indexOf("xyz")); // 4  　　System.out.println(list.indexOf("abcd")); // -1    　　// -------------------------------------------------------  　　// 测试删除数据  　　// 请注意，  　　// 如果你使用整数(int)数字，则默认调用的是remove(int index);  　　// 如果你用 long,则会调用 remove(Object obj);  　　// 所以如果你要删除整数，请使用 remove(new Integer(int));  　　// -------------------------------------------------------  　　// 删除索引为1的数据  　　list.remove(1);  　　// 索引为1的数据被干掉了  　　showList(list); // [2.34, bcd, 2, xyz, 1, abc, 1.11]    　　// 删除数字1 和字符串 abc  　　list.remove(new Integer(1));  　　list.remove("xyz");  　　showList(list); // [2.34, bcd, 2, abc, 1.11]    　　// -------------------------------------------------------  　　// 迭代器的使用  　　// -------------------------------------------------------  　　Iterator it = list.iterator();  　　while (it.hasNext()) {  　　　System.out.print(it.next() + " "); // 2.34 bcd 2 abc 1.11  　　}  　　System.out.println();    　　// -------------------------------------------------------  　　// 转化为数组  　　// -------------------------------------------------------  　　Object[] objs = list.toArray();  　　for (Object obj : objs) {  　　　System.out.print(obj + " "); // 2.34 bcd 2 abc 1.11  　　}  　　System.out.println();  　}    　public static void testSpecial() {  　　// -------------------------------------------------------  　　// 测试重复和null  　　// -------------------------------------------------------  　　//  　　List<Integer> list = new ArrayList<Integer>();  　　list.add(123);  　　list.add(456);  　　list.add(123);  　　list.add(456);  　　// 数据允许重复  　　showList(list); // [123, 456, 123, 456]    　　list.add(null);  　　list.add(789);  　　list.add(null);  　　list.add(999);  　　// 允许放入多个null  　　showList(list); // [123, 456, 123, 456, null, 789, null, 999]    　　// -------------------------------------------------------  　　// 测试一下查找最后一次出现的位置  　　// -------------------------------------------------------  　　System.out.println(list.indexOf(123)); // 0  　　System.out.println(list.lastIndexOf(123)); // 2    　　// -------------------------------------------------------  　　// 转化为数组  　　// 记得要转化为Inerger.  　　// -------------------------------------------------------  　　Integer[] nums = (Integer[]) list.toArray(new Integer[0]);  　　// 注意数据里面有null,所以循环变量不要用int 要用Integer  　　for (Integer num : nums) {  　　　System.out.print(num + " "); // 123 456 123 456 null 789 null 999  　　}  　　System.out.println();    　}    　public static void testForProblem() {  　　// 一些朋友在向循环里向列表增加对象的时候  　　// 经常忘记初始化，造成最终加入的都是同一个对象  　　List<MyObject> list = new ArrayList<MyObject>();  　　MyObject obj = new MyObject();  　　for (int i = 1; i <= 5; i++) {  　　　obj.setName("Name" + i);  　　　list.add(obj);  　　}  　　// 里面的数据都是最后一个  　　showList(list); // [Name5, Name5, Name5, Name5, Name5]    　　// 正确的做法  　　List<MyObject> list2 = new ArrayList<MyObject>();  　　MyObject obj2 = null;  　　for (int i = 1; i <= 5; i++) {  　　　obj2 = new MyObject();  　　　obj2.setName("Name" + i);  　　　list2.add(obj2);  　　}  　　// 里面的数据都是最后一个  　　showList(list2); // [Name1, Name2, Name3, Name4, Name5]  　}    　/\*\*  　 \* 显示List里面的数据。  　 \*  　 \* @param list  　 \*/  　private static void showList(List list) {  　　System.out.println(Arrays.toString(list.toArray()));  　}  }    class MyObject {  　private String name;    　public String getName() {  　　return name;  　}    　public void setName(String name) {  　　this.name = name;  　}    　/\*\*  　 \* 重写toString方法，输出name  　 \*/  　public String toString() {  　　return name;  　}  } |

　　输出结果

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | [1, abc, 1.11]  [2.34, bcd, 2, 1, abc, 1.11]  [2.34, 3, bcd, 2, xyz, 1, abc, 1.11]  2.34  xyz  true  true  false  1  4  -1  [2.34, bcd, 2, xyz, 1, abc, 1.11]  [2.34, bcd, 2, abc, 1.11]  2.34 bcd 2 abc 1.11  2.34 bcd 2 abc 1.11  [123, 456, 123, 456]  [123, 456, 123, 456, null, 789, null, 999]  0  2  123 456 123 456 null 789 null 999  [Name5, Name5, Name5, Name5, Name5]  [Name1, Name2, Name3, Name4, Name5] |

### [Java中ArrayList和LinkedList区别](http://pengcqu.iteye.com/blog/502676)

一般大家都知道ArrayList和LinkedList的大致区别：   
     1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。   
     2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。   
     3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。   
  
ArrayList和LinkedList是两个集合类，用于存储一系列的对象引用(references)。例如我们可以用ArrayList来存储一系列的String或者Integer。那么ArrayList和LinkedList在性能上有什么差别呢？什么时候应该用ArrayList什么时候又该用LinkedList呢？

**一．时间复杂度**   
首先一点关键的是，ArrayList的内部实现是基于基础的对象数组的，因此，它使用get方法访问列表中的任意一个元素时(random access)，它的速度要比LinkedList快。LinkedList中的get方法是按照顺序从列表的一端开始检查，直到另外一端。对LinkedList而言，访问列表中的某个指定元素没有更快的方法了。   
假设我们有一个很大的列表，它里面的元素已经排好序了，这个列表可能是ArrayList类型的也可能是LinkedList类型的，现在我们对这个列表来进行二分查找(binary search)，比较列表是ArrayList和LinkedList时的查询速度，看下面的程序：

**Java代码  [收藏代码](javascript:void())**

1. **package** com.mangocity.test;
2. **import** java.util.LinkedList;
3. **import** java.util.List;
4. **import** java.util.Random;
5. **import** java.util.ArrayList;
6. **import** java.util.Arrays;
7. **import** java.util.Collections;
8. **public** **class** TestList ...{
9. **public** **static** **final** **int** N=50000;
11. **public** **static** List values;
13. **static**...{
14. Integer vals[]=**new** Integer[N];
16. Random r=**new** Random();
18. **for**(**int** i=0,currval=0;i<N;i++)...{
19. vals=**new** Integer(currval);
20. currval+=r.nextInt(100)+1;
21. }
23. values=Arrays.asList(vals);
24. }
26. **static** **long** timeList(List lst)...{
27. **long** start=System.currentTimeMillis();
28. **for**(**int** i=0;i<N;i++)...{
29. **int** index=Collections.binarySearch(lst, values.get(i));
30. **if**(index!=i)
31. System.out.println("\*\*\*错误\*\*\*");
32. }
33. **return** System.currentTimeMillis()-start;
34. }
35. **public** **static** **void** main(String args[])...{
36. System.out.println("ArrayList消耗时间："+timeList(**new** ArrayList(values)));
37. System.out.println("LinkedList消耗时间："+timeList(**new** LinkedList(values)));
38. }
39. }

我得到的输出是：ArrayList消耗时间：15   
                 LinkedList消耗时间：2596   
这个结果不是固定的，但是基本上ArrayList的时间要明显小于LinkedList的时间。因此在这种情况下不宜用LinkedList。二分查找法使用的随机访问(random access)策略，而LinkedList是不支持快速的随机访问的。对一个LinkedList做随机访问所消耗的时间与这个list的大小是成比例的。而相应的，在ArrayList中进行随机访问所消耗的时间是固定的。   
这是否表明ArrayList总是比LinkedList性能要好呢？这并不一定，在某些情况下LinkedList的表现要优于ArrayList，有些算法在LinkedList中实现时效率更高。比方说，利用Collections.reverse方法对列表进行反转时，其性能就要好些。   
看这样一个例子，加入我们有一个列表，要对其进行大量的插入和删除操作，在这种情况下LinkedList就是一个较好的选择。请看如下一个极端的例子，我们重复的在一个列表的开端插入一个元素：

**Java代码  [收藏代码](javascript:void())**

1. **package** com.mangocity.test;
3. **import** java.util.\*;
4. **public** **class** ListDemo {
5. **static** **final** **int** N=50000;
6. **static** **long** timeList(List list){
7. **long** start=System.currentTimeMillis();
8. Object o = **new** Object();
9. **for**(**int** i=0;i<N;i++)
10. list.add(0, o);
11. **return** System.currentTimeMillis()-start;
12. }
13. **public** **static** **void** main(String[] args) {
14. System.out.println("ArrayList耗时："+timeList(**new** ArrayList()));
15. System.out.println("LinkedList耗时："+timeList(**new** LinkedList()));
16. }
17. }

 这时我的输出结果是：ArrayList耗时：2463

                           LinkedList耗时：15   
这和前面一个例子的结果截然相反，当一个元素被加到ArrayList的最开端时，所有已经存在的元素都会后移，这就意味着数据移动和复制上的开销。相反的，将一个元素加到LinkedList的最开端只是简单的未这个元素分配一个记录，然后调整两个连接。在LinkedList的开端增加一个元素的开销是固定的，而在ArrayList的开端增加一个元素的开销是与ArrayList的大小成比例的。

**二．空间复杂度**   
在LinkedList中有一个私有的内部类，定义如下：

**Java代码  [收藏代码](javascript:void())**

1. **private** **static** **class** Entry {
2. Object element;
3. Entry next;
4. Entry previous;
5. }

每个Entry对象reference列表中的一个元素，同时还有在LinkedList中它的上一个元素和下一个元素。一个有1000个元素的LinkedList对象将有1000个链接在一起的Entry对象，每个对象都对应于列表中的一个元素。这样的话，在一个LinkedList结构中将有一个很大的空间开销，因为它要存储这1000个Entity对象的相关信息。   
ArrayList使用一个内置的数组来存储元素，这个数组的起始容量是10.当数组需要增长时，新的容量按如下公式获得：新容量=(旧容量\*3)/2+1，也就是说每一次容量大概会增长50%。这就意味着，如果你有一个包含大量元素的ArrayList对象，那么最终将有很大的空间会被浪费掉，这个浪费是由ArrayList的工作方式本身造成的。如果没有足够的空间来存放新的元素，数组将不得不被重新进行分配以便能够增加新的元素。对数组进行重新分配，将会导致性能急剧下降。如果我们知道一个ArrayList将会有多少个元素，我们可以通过构造方法来指定容量。我们还可以通过trimToSize方法在ArrayList分配完毕之后去掉浪费掉的空间。

三．总结   
ArrayList和LinkedList在性能上各有优缺点，都有各自所适用的地方，总的说来可以描述如下：   
1．对ArrayList和LinkedList而言，在列表末尾增加一个元素所花的开销都是固定的。对ArrayList而言，主要是在内部数组中增加一项，指向所添加的元素，偶尔可能会导致对数组重新进行分配；而对LinkedList而言，这个开销是统一的，分配一个内部Entry对象。

2．在ArrayList的中间插入或删除一个元素意味着这个列表中剩余的元素都会被移动；而在LinkedList的中间插入或删除一个元素的开销是固定的。

3．LinkedList不支持高效的随机元素访问。

4．ArrayList的空间浪费主要体现在在list列表的结尾预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗相当的空间

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用ArrayList会提供比较好的性能；当你的操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用LinkedList了。

# Java ArrayList的自动扩容机制

**注意：**

不同的JDK版本的扩容机制可能有差异   
实验环境：JDK1.8

**扩容机制**：

当向ArrayList中添加元素的时候，ArrayList如果要满足新元素的存储超过ArrayList存储新元素前的存储能力，ArrayList会增强自身的存储能力，已达到存储新元素的要求

ArrayList:**本质通过内部维护的数组对象进行数据存储**

**①：分析ArrayList的add(E)方法**

public boolean add(E e) {

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

elementData[size++] = e;

return true;

}

* 1
* 2
* 3
* 4
* 5

分析：add方法首先通过ensureCapacityInternal()方法确保当前ArrayList维护的数组具有存储新元素的能力，经过处理之后将元素存储在数组elementData的尾部   
elementData：ArrayList真正用于存储元素的数组

**②：分析ensureCapacityInternal方法**

private void ensureCapacityInternal(int minCapacity) {

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

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

}

ensureExplicitCapacity(minCapacity);

}

* 1
* 2
* 3
* 4
* 5
* 6

分析：ensureCapacityInternal判断ArrayList默认的元素存储数据是否为空，为空则设置最小要求的存储能力为必要存储的元素和默认存储元素个数的两个数据之间的最大值，然后调用ensureExplicitCapacity方法实现这种最低要求的存储能力

注意：ArrayList的存储空间并不是需要一个创建一个，而是分阶段性的创建，一般会预留存储空间。   
例如，如果ArrayList需要存储10个元素，恰好ArrayList只能存储6个元素，剩余4个元素无法存储，ArrayList可能会一次性扩展10个元素，这种ArrayList就有20个元素的存储能力，在存储能力范围内，下次再存放元素，就不需要再次扩容

**③：分析ensureExplicitCapacity方法：**

private void ensureExplicitCapacity(int minCapacity) {

modCount++;

// overflow-conscious code

if (minCapacity - elementData.length > 0)

grow(minCapacity);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

分析：如果最低要求的存储能力>ArrayList已有的存储能力，这就表示ArrayList的存储能力不足，因此需要调用 grow();方法进行扩容   
④：分析grow()方法

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);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

分析：当ArrayList扩容的时候，首先会设置新的存储能力为原来的1.5倍

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

* 1

如果扩容之后还是不能满足要求则MAX\_ARRAY\_SIZE比较，求取最大值，   
如果MAX\_ARRAY\_SIZE大小的能力还是不能满足则通过hugeCapacity()方法获取ArrayList能允许的最大值：

private static int hugeCapacity(int minCapacity) {

if (minCapacity < 0) // overflow

throw new OutOfMemoryError();

return (minCapacity > MAX\_ARRAY\_SIZE) ?

Integer.MAX\_VALUE :

MAX\_ARRAY\_SIZE;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

从hugeCapacity方法看出，ArrayList最大的存储能力：存储元素的个数为整型的范围。   
确定ArrayList扩容之后最新的可存储元素个数时，调用   
elementData = Arrays.copyOf(elementData, newCapacity);   
实现elementData数组的扩容，整个流程就是ArrayList的自动扩容机制工作流程

扩展：   
ArrayList的自动扩容机制底层借助于System实现

public static native void arraycopy

(Object src, int srcPos,

Object dest, int destPos,

int length);

* 1
* 2
* 3
* 4

arraycopy标识为native意味JDK的本地库，不可避免的会进行IO操作，如果频繁的对ArrayList进行扩容，毫不疑问会降低ArrayList的使用性能，因此当我们确定添加元素的个数的时候，我们可以事先知道并指定ArrayList的可存储元素的个数，这样当我们向ArrayList中加入元素的时候，就可以避免ArrayList的自动扩容，从而提高ArrayList的性能

ArrayList含参构造函数:初始化时指定存储元素的能力：

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);

}

}