ArrayList 实现原理

作者：dreamertn9527
原文链接：https://ld246.com/article/1514043359369
来源网站：链滴

ArrayList实现原理

一、 ArrayList概述

 ArrayList是基于数组实现的，是一个动态数组，其容量能自动增长，类似于C语言中的动态申请存，动态增长内存。ArrayList不是线程安全的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发下的CopyOnWriteArrayList类。

 ArrayList实现了Serializable接口，因此它支持序列化，能够通过序列化传输，实现了RandomAccess接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了Cloneable接口，能克隆。

 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构上修改了列表，那么它必须保持外部同步。

二、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。面我们来分析ArrayList的源代码：

私有属性

ArrayList定义只定义类两个私有属性：

```
    // 声明数组
    private transient Object[] elementData;

    // ArrayList容量
    private int size;
```

elementData存储ArrayList内的元素，size表示它包含的元素的数量

 有个关键字需要解释：transient。 Java的serialization提供了一种持久化对象实例的机制。当持化对象时，可能有一个特殊的对象数据成员，我们不想用serialization机制来保存它。为了在一个特对象的一个域上关闭serialization，可以在这个域前加上关键字transient。如下：

```
public class UserInfo implements Serializable {
    private static final long serialVersionUID = 996890129747019948L;
    private String name;
    private transient String psw;

    public UserInfo(String name, String psw) {
        this.name = name;
        this.psw = psw;
    }
```

```
    public String toString() {
        return "name=" + name + ", psw=" + psw;
    }
}

public class TestTransient {
    public static void main(String[] args) {
        UserInfo userInfo = new UserInfo("张三", "123456");
        System.out.println(userInfo);
        try {
            // 序列化，被设置为transient的属性没有被序列化
            ObjectOutputStream o = new ObjectOutputStream(new FileOutputStream(
                    "UserInfo.out"));
            o.writeObject(userInfo);
            o.close();
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
        try {
            // 重新读取内容
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(
                    "UserInfo.out"));
            UserInfo readUserInfo = (UserInfo) in.readObject();
            //读取后psw的内容为null
            System.out.println(readUserInfo.toString());
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }
}
```

被标记为transient的属性在对象被序列化的时候不会被保存

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的迭代器返回它的顺序排列的:

```
// ArrayList带容量大小的构造函数。
    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        // 新建一个数组
        this.elementData = new Object[initialCapacity];
    }

    // ArrayList无参构造函数。默认容量是10。
    public ArrayList() {
        this(10);
```

```
    }

    // 创建一个包含collection的ArrayList
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        size = elementData.length;
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    }
```

元素存储

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。下面我一一讲解：

```
// 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。
public E set(int index, E element) {
    RangeCheck(index);

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

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

// 将指定的元素插入此列表中的指定位置。
// 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）。
public void add(int index, E element) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);
    // 如果数组长度不足，将进行扩容。
    ensureCapacity(size+1);  // Increments modCount!!
    // 将 elementData中从Index位置开始、长度为size-index的元素，
    // 拷贝到从下标为index+1位置开始的新的elementData数组中。
    // 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。
    System.arraycopy(elementData, index, elementData, index + 1, size - index);
    elementData[index] = element;
    size++;
}


// 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
。
public boolean addAll(Collection<? extends E> c) {
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacity(size + numNew);  // Increments modCount
```

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


// 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。
public boolean addAll(int index, Collection<? extends E> c) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(
            "Index: " + index + ", Size: " + size);

    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacity(size + numNew);  // Increments modCount

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

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

元素读取

```
// 返回此列表中指定位置上的元素。
 public E get(int index) {
    RangeCheck(index);

    return (E) elementData[index];
  }
```

元素删除

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：

```
// 移除此列表中指定位置上的元素。
 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;
 }
```

首先是检查范围，修改modCount，保留将要被移除的元素，将移除位置之后的元素向前挪动一个位，将list末尾元素置空（null），返回被移除的元素。

```
// 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。
 public boolean remove(Object o) {
   // 由于ArrayList中允许存放null，因此下面通过两种情况来分别处理。
   if (o == null) {
      for (int index = 0; index < size; index++)
         if (elementData[index] == null) {
            // 类似remove(int index)，移除列表中指定位置上的元素。
            fastRemove(index);
            return true;
         }
   } else {
      for (int index = 0; index < size; index++)
         if (o.equals(elementData[index])) {
            fastRemove(index);
            return true;
         }
      }
      return false;
   }
}
```

首先通过代码可以看到，当移除成功后返回true，否则返回false。remove(Object o)中通过遍历element寻找是否存在传入对象，一旦找到就调用fastRemove移除对象。为什么找到了元素就知道了index，不通过remove(index)来移除元素呢？因为fastRemove跳过了判断边界的处理，因为找到元素就当于确定了index不会超过边界，而且fastRemove并不返回被移除的元素。下面是fastRemove的代，基本和remove(index)一致。

```
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
 }

protected void removeRange(int fromIndex, int toIndex) {
    modCount++;
    int numMoved = size - toIndex;
       System.arraycopy(elementData, toIndex, elementData, fromIndex,
                        numMoved);

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

执行过程是将elementData从toIndex位置开始的元素向前移动到fromIndex，然后将toIndex位置之的元素全部置空顺便修改size。

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。

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。

```
public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;  //增加50%+1
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
      // minCapacity is usually close to size, so this is a win:
      elementData = Arrays.copyOf(elementData, newCapacity);
    }
 }
```

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实的容量。

Object oldData[] = elementData;//为什么要用到oldData[]

乍一看来后面并没有用到关于oldData， 这句话显得多此一举！但是这是一个牵涉到内存管理的类，所以要了解内部的问题。 而且为什么这一句还在if的内部，这跟elementData = Arrays.copyOf(elemntData, newCapacity); 这句是有关系的， 下面这句Arrays.copyOf的实现时新创建了newCapacity小的内存，然后把老的elementData放入。好像也没有用到oldData，有什么问题呢。问题就在于旧内存的引用是elementData， elementData指向了新的内存块，如果有一个局部变量oldData变量引旧的内存块的话，在copy的过程中就会比较安全，因为这样证明这块老的内存依然有引用，分配内存时候就不会被侵占掉，然后copy完成后这个局部变量的生命期也过去了，然后释放才是安全的。不然copy的的时候万一新的内存或其他线程的分配内存侵占了这块老的内存，而copy还没有结束，这将个严重的事情。

关于ArrayList和Vector区别如下：

- ArrayList在内存不够时默认是扩展50% + 1个，Vector是默认扩展1倍。
- Vector提供indexOf(obj, start)接口，ArrayList没有。
- Vector属于线程安全级别的，但是大多数情况下不使用Vector，因为线程安全需要更大的系统开销。

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trimToSize方法来实现。代码如下：

```
public void trimToSize() {
  modCount++;
  int oldCapacity = elementData.length;
  if (size < oldCapacity) {
      elementData = Arrays.copyOf(elementData, size);
  }
   }
```

由于elementData的长度会被拓展，size标记的是其中包含的元素的个数。所以会出现size很小但elementData.length很大的情况，将出现空间的浪费。trimToSize将返回一个新的数组给elementData，素内容保持不变，length和size相同，节省空间。

转为静态数组toArray

注意ArrayList的两个转化为静态数组的toArray方法。

 第一个， 调用Arrays.copyOf将返回一个数组，数组内容是size个elementData的元素，即拷贝elementData从0至size-1位置的元素到新数组并返回。

```
public Object[] toArray() {
        return Arrays.copyOf(elementData, size);
 }
```

第二个，如果传入数组的长度小于size，返回一个新的数组，大小为size，类型与传入数组相同。所入数组长度与size相等，则将elementData复制到传入数组中并返回传入的数组。若传入数组长度大size，除了复制elementData外，还将把返回数组的第size个元素置为空。

```
public <T> T[] toArray(T[] a) {
        if (a.length < size)
            // Make a new array of a's runtime type, but my contents:
            return (T[]) Arrays.copyOf(elementData, size, a.getClass());
    System.arraycopy(elementData, 0, a, 0, size);
        if (a.length > size)
            a[size] = null;
        return a;
    }
```

三、总结

关于ArrayList的源码，给出几点比较重要的总结：

 1、注意其三个不同的构造方法。无参构造方法构造的ArrayList的容量默认为10，带有Collection数的构造方法，将Collection转化为数组赋给ArrayList的实现数组elementData。

 2、注意扩充容量的方法ensureCapacity。ArrayList在每次增加元素（可能是1个，也可能是一组时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧容量的1.5倍加1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量，而后用Arrays.copyof()方法将元素拷贝到新的数组（详见下面的第3点）。从中可以看出，当容量够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此建议在事先能定元素数量的情况下，才使用ArrayList，否则建议使用LinkedList。

 3、ArrayList的实现中大量地调用了Arrays.copyof()和System.arraycopy()方法。我们有必要对两个方法的实现做下深入的了解。

 首先来看Arrays.copyof()方法。它有很多个重载的方法，但实现思路都是一样的，我们来看泛型本的源码：

```
public static <T> T[] copyOf(T[] original, int newLength) {
    return (T[]) copyOf(original, newLength, original.getClass());
}
```

很明显调用了另一个copyof方法，该方法有三个参数，最后一个参数指明要转换的数据的类型，其源如下：

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

这里可以很明显地看出，该方法实际上是在其内部又创建了一个长度为newlength的数组，调用System.arraycopy()方法，将原来数组中的元素复制到了新的数组中。

 下面来看System.arraycopy()方法。该方法被标记了native，调用了系统的C/C++代码，在JDK是看不到的，但在openJDK中可以看到其源码。该函数实际上最终调用了C语言的memmove()函数因此它可以保证同一个数组内元素的正确复制和移动，比一般的复制方法的实现效率要高很多，很适用来批量处理数组。Java强烈推荐在复制大量数组元素时用该方法，以取得更高的效率。

4、ArrayList基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每插入或删除元素，就要大量地移动元素，插入删除元素的效率低。

5、在查找给定元素索引值等的方法中，源码都将该元素的值分为null和不为null两种情况处理，ArrayList中允许元素为null。