深入 Java 集合学习系列：ArrayList 的实现原理

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

```
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

```
1. public ArrayList() {
2.     this(10);
3. }
4.
5. public ArrayList(int initialCapacity) {
6.     super();
7.     if (initialCapacity < 0)
8.         throw new IllegalArgumentException("Illegal Capacity: "+ initi
   alCapacity);
9.     this.elementData = new Object[initialCapacity];
10.}
11.
12.public ArrayList(Collection<? extends E> c) {
13.    elementData = c.toArray();
14.    size = elementData.length;
```

```
15.     // c.toArray might (incorrectly) not return Object[] (see 626065
    2)
16.     if (elementData.getClass() != Object[].class)
17.         elementData = Arrays.copyOf(elementData, size, Object[].clas
    s);
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

```
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);
4.
5.      E oldValue = (E) elementData[index];
6.      elementData[index] = element;
7.      return oldValue;
8.  }
```

Java 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

```
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, siz
    e - index);
12.     elementData[index] = element;
```

```
13.     size++;
14. }
```

Java 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

```
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);
6.
7.     Object[] a = c.toArray();
8.     int numNew = a.length;
9.     ensureCapacity(size + numNew);  // Increments modCount
10.
11.    int numMoved = size - index;
12.    if (numMoved > 0)
13.        System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
14.
15.    System.arraycopy(a, 0, elementData, index, numNew);
16.    size += numNew;
17.    return numNew != 0;
18. }
```

4) 读取：

Java 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);
4.
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

```java
// 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;
}
```

Java 代码

```java
// 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;
}
```

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

```
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

```
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 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。

原文地址

http://zhangshixi.iteye.com/blog/674856