敖丙 Lv5
2020年01月08日 阅读 3421

已关注

《吊打面试官》系列-ArrayList

> 你知道的越多，你不知道的越多
>
> 点赞再看，养成习惯
>
> 本文 **GitHub** github.com/JavaFamily 已收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎Star和完善，大家面试可以参照考点复习，希望我们一起有点东西。

前言

作为一个在互联网公司面一次拿一次Offer的面霸，打败了无数竞争对手，每次都只能看到无数落寞的身影失望的离开，略感愧疚（**请允许我使用一下夸张的修辞手法**）。

于是在一个寂寞难耐的夜晚，我痛定思痛，决定开始写互联网技术栈面试相关的文章，希望能帮助各位读者以后面试势如破竹，对面试官进行360°的反击，吊打问你的面试官，让一同面试的同僚瞠目结舌，疯狂收割大厂Offer！

所有文章的名字只是我的噱头，我们应该有一颗谦逊的心，所以希望大家怀着空杯心态好好学，一起进步。

正文

> 一个婀娜多姿，穿着衬衣的小姐姐，拿着一个精致的小笔记本，径直走过来坐在我的面前。
> 看着眼前这个美丽的女人，心想这不会就是Java基础系列的面试官吧，真香。
> 不过看样子这么年轻应该问不出什么深度的吧，嘻嘻。（哦？是么😏）

> 帅丙，上次面试到现在都过去2个星期你才过来，为啥鸽了这么久？

美丽迷人的面试官您好，因为之前得了流感，差点就没了，还有最近年底忙年会和年终review的事情，实在太忙了，不好意思。

还做了一波导演（其实是打杂）去拍摄蘑菇街的年会视频了，实在忙到爆炸，周末也没能怼出文章哈哈。

©

> 好吧那我理解了，下次这种情况记得提前跟我说，对了，多喝热水。

面试官最后的多喝热水，直接触动我内心的防线，居然还有人这么关心我，帅丙的眼角，又湿了……

> ArrayList有用过吗？它是一个什么东西？可以用来干嘛？

有用过，ArrayList就是数组列表，主要用来装载数据，当我们装载的是基本类型的数据int，long，boolean，short，byte...的时候我们只能存储他们对应的包装类，它的主要底层实现是数组Object[] elementData。

与它类似的是LinkedList，和LinkedList相比，它的查找和访问元素的速度较快，但新增，删除的速度较慢。

小结：ArrayList底层是用数组实现的存储。

特点：查询效率高，增删效率低，线程不安全。使用频率很高。

> 为啥线程 不安全还使用他呢？

因为我们正常使用的场景中，都是用来查询，不会涉及太频繁的增删，如果涉及频繁的增删，可以使用LinkedList，如果你需要线程安全就使用Vector，这就是三者的区别了，实际开发过程中还是ArrayList使用最多的。

不存在一个集合工具是查询效率又高，增删效率也高的，还线程安全的，至于为啥大家看代码就知道了，因为数据结构的特性就是优劣共存的，想找个平衡点很难，牺牲了性能，那就安全，牺牲了安全那就快速。

Tip：这里还是强调下大家**不要为了用而用**，我记得我以前最开始工作就有这个毛病。就是不管三七二十一，就是为了用而用，别人用我也用，也不管他的性能啊，是否线程安全啥的，所幸最开始公司接触的，都是线下的系统，而且没啥并发。

现在回想起来还有点后怕，而且我第一次出去面试就被一个算法的大佬给虐得体无完肤，其实他问的我都会用，但是就是说不上来为啥用，为啥这么做。

回想起一年前的第一次面试，我又想到了那时候的点滴，**那时候我身边还有女人**，那时候我还有头发，那时候……我的眼角又湿了。

> 您说它的底层实现是数组，但是数组的大小是定长的，如果我们不断的往里面添加数据的话，不会有问题吗？

ArrayList可以通过构造方法在初始化的时候指定底层数组的大小。

通过无参构造方法的方式ArrayList()初始化，则赋值底层数Object[] elementData为一个默认空数组Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {}所以数组容量为0，只有真正对数据进行添加add时，才分配默认DEFAULT_CAPACITY = 10的初始容量。

大家可以分别看下他的无参构造器和有参构造器，无参就是默认大小，有参会判断参数。

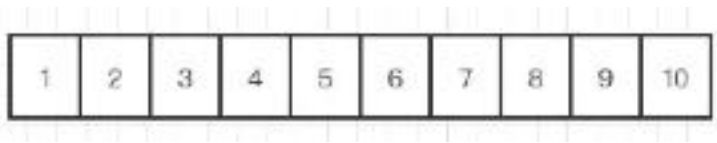
©

```java
/**
 * 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);
    }
}

/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() { this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA; }
```

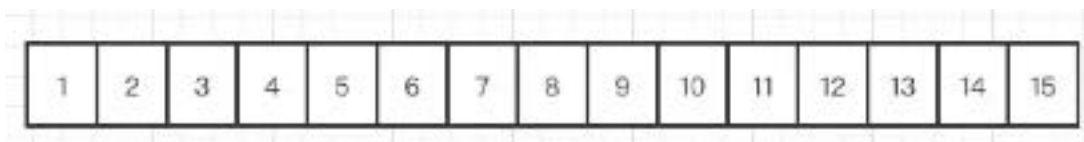
> 数组的长度是有限制的，而ArrayList是可以存放任意数量对象，长度不受限制，那么他是怎么实现的呢？

其实实现方式比较简单，他就是通过数组扩容的方式去实现的。

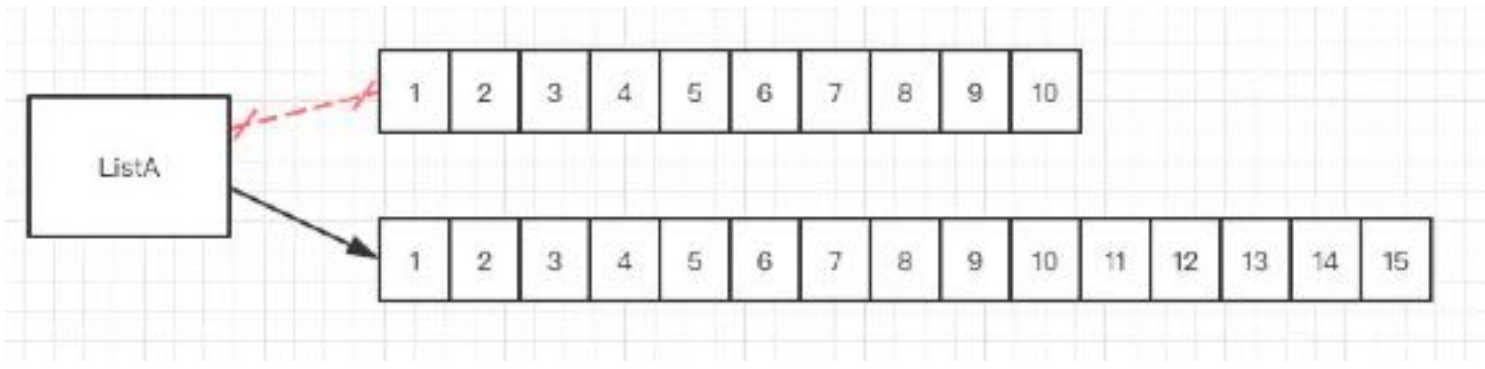
就比如我们现在有一个长度为10的数组，现在我们要新增一个元素，发现已经满了，那ArrayList会怎么做呢？



第一步他会重新定义一个长度为10+10/2的数组也就是新增一个长度为15的数组。



然后把原数组的数据，原封不动的复制到新数组中，这个时候再把指向原数的地址换到新数组，ArrayList就这样完成了一次改头换面。



Tip：很多小伙伴说，你举例干嘛用10这么长的数组举例，这样画都要多画一点格子了，帅丙我会做无意义的事情？因为我们在使用ArrayList的时候一般不会设置初始值的大小，那ArrayList默认的大小就刚好是10。

```java
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;
```

然后你们也可以看到，他的构造方法，如果你传入了初始值大小，那就使用你传入的参数，如果没，那就使用默认的，一切都是有迹可循的。

> 能具体说下1.7和1.8版本初始化的时候的区别么？

arrayList1.7开始变化有点大，一个是初始化的时候，1.7以前会调用this(10)才是真正的容量为10，1.7即本身以后是默认走了空数组，只有第一次add的时候容量会变成10。

> ArrayList的默认数组大小为什么是10？

其实我也没找到具体原因。

据说是因为sun的程序员对一系列广泛使用的程序代码进行了调研，结果就是10这个长度的数组是最常用的最有效率的。也有说就是随便起的一个数字，8个12个都没什么区别，只是因为10这个数组比较的圆满而已。

> 我记得你说到了，他增删很慢，你能说一下ArrayList在增删的时候是怎么做的么？主要说一下他为啥慢。

诶卧*，这个我想一下，大学看的有点忘记了，我想想。

嗯嗯好的，我分别说一下他的新增的逻辑吧。

他有指定index新增，也有直接新增的，在这之前他会有一步校验长度的判断**ensureCapacityInternal**，就是说如果长度不够，是需要扩容的。

```java
/**
 * 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( minCapacity: size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

©

在扩容的时候，老版本的jdk和8以后的版本是有区别的，8之后的效率更高了，采用了位运算，**右移**一位，其实就是除以2这个操作。

1.7的时候3/2+1 ，1.8直接就是3/2。

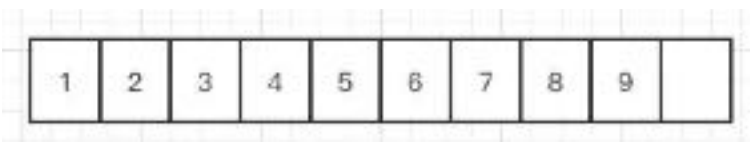
```
/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    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);
}
```

指定位置新增的时候，在校验之后的操作很简单，就是数组的copy，大家可以看下代码。

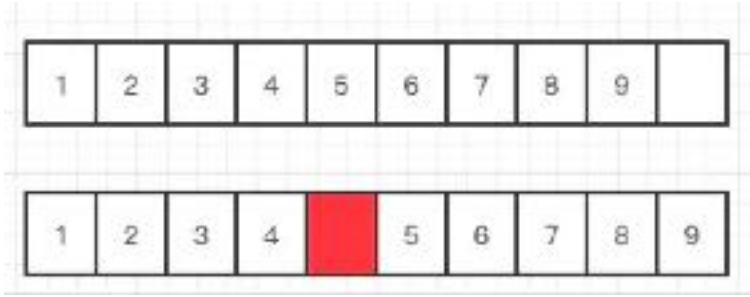
```
/**
 * Inserts the specified element at the specified position in this
 * list. Shifts the element currently at that position (if any) and
 * any subsequent elements to the right (adds one to their indices).
 *
 * @param index index at which the specified element is to be inserted
 * @param element element to be inserted
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal( minCapacity: size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData,  destPos: index + 1,
                     length: size - index);
    elementData[index] = element;
    size++;
}
```

不知道大家看懂**arraycopy**的代码没有，我画个图解释下，你可能就明白一点：

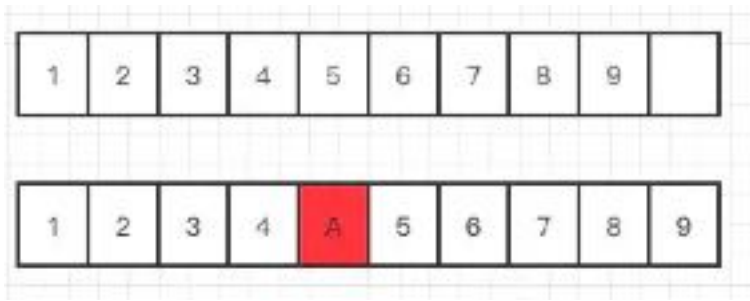
比如有下面这样一个数组我需要在index 5的位置去新增一个元素A



那从代码里面我们可以看到，他复制了一个数组，是从index 5的位置开始的，然后把它放在了index 5+1的位置



给我们要新增的元素腾出了位置，然后在index的位置放入元素A就完成了新增的操作了



至于为啥说他效率低，我想我不说你也应该知道了，我这只是在一个这么小的List里面操作，要是我去一个几百几千几万大小的List新增一个元素，那就需要后面所有的元素都复制，然后如果再涉及到扩容啥的就更慢了不是嘛。

> 我问你个真实的场景，这个问题很少人知道，你可要好好回答哟！
> ArrayList（int initialCapacity）会不会初始化数组大小？

这是什么问题？卧槽问个ArrayList还能问到知识盲区？

不要慌，我记得丙丙说过：无论我们遇到什么困难都不要害怕，战胜恐惧最好的办法就是面对他！！！奥利给！！！...

会初始化数组大小！但是List的大小没有变，因为list的大小是返回size的。

而且将构造函数与initialCapacity结合使用，然后使用set（）会抛出异常，尽管该数组已创建，但是大小设置不正确。

使用sureCapacity（）也不起作用，因为它基于elementData数组而不是大小。

还有其他副作用，这是因为带有sureCapacity（）的静态DEFAULT_CAPACITY。

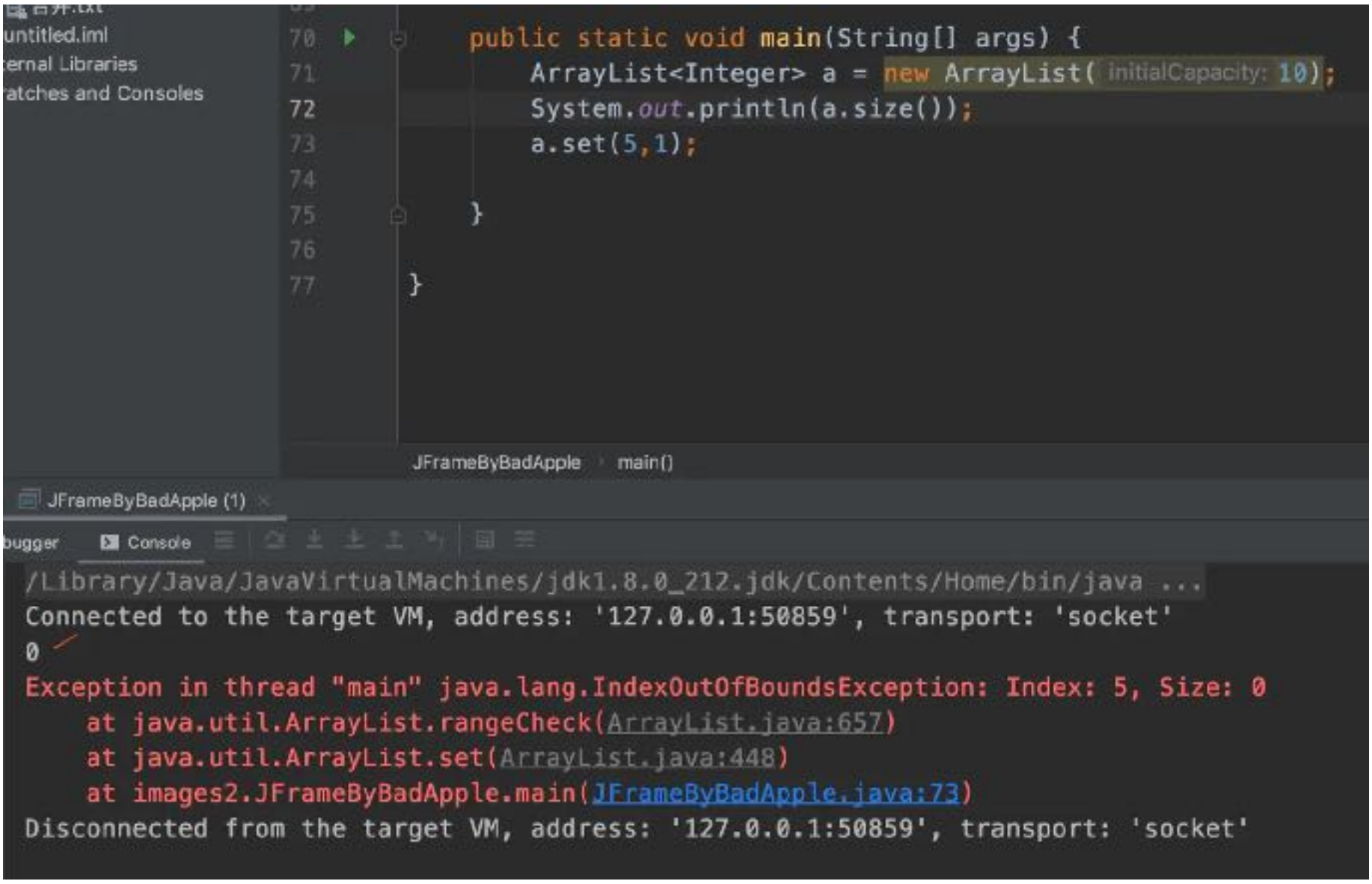
进行此工作的唯一方法是在使用构造函数后，根据需要使用add（）多次。

大家可能有点懵，我直接操作一下代码，大家会发现我们虽然对ArrayList设置了初始大小，但是我们打印List大小的时候还是0，我们操作下标set值的时候也会报错，数组下标越界。

其实数组是初始化了，但是List没，那size就没变，set下标是和size比较的那就报错了。

再结合源码，大家仔细品读一下，这是Java Bug里面的一个经典问题了，还是很有意思的，大家平时可能也不会注意这个点。

©



> ArrayList插入删除一定慢么？

取决于你删除的元素离数组末端有多远，ArrayList拿来作为堆栈来用还是挺合适的，push和pop操作完全不涉及数据移动操作。

> 那他的删除怎么实现的呢？

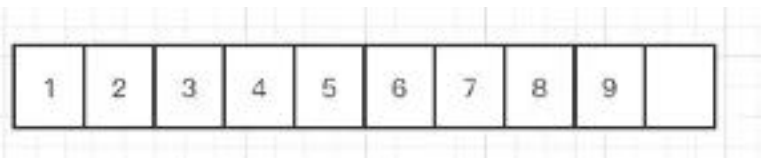
删除其实跟新增是一样的，不过叫是叫删除，但是在代码里面我们发现，他还是在copy一个数组。

> 为啥是copy数组呢？

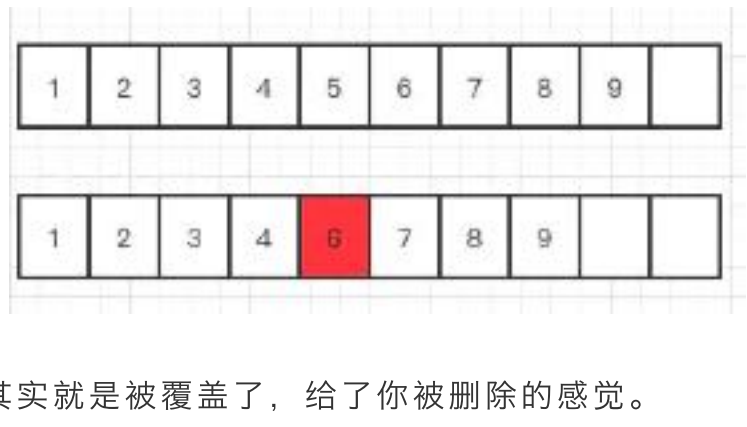
©

```java
/**
 * 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, srcPos: index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
```

继续打个比方，我们现在要删除下面这个数组中的index5这个位置



那代码他就复制一个index5+1开始到最后的数组，然后把它放到index开始的位置



index5的位置就成功被”删除“了其实就是被覆盖了，给了你被删除的感觉。

同理他的效率也低，因为数组如果很大的话，一样需要复制和移动的位置就大了。

> ArrayList是线程安全的么？

当然不是，线程安全版本的数组容器是Vector。

Vector的实现很简单，就是把所有的方法统统加上synchronized就完事了。

你也可以不使用Vector，用Collections.synchronizedList把一个普通ArrayList包装成一个线程安全版本的数组容器也可以，原理同Vector是一样的，就是给所有的方法套上一层synchronized。

> ArrayList用来做队列合适么？

队列一般是FIFO（先入先出）的，如果用ArrayList做队列，就需要在数组尾部追加数据，数组头部删除数组，反过来也可以。

但是无论如何总会有一个操作会涉及到数组的数据搬迁，这个是比较耗费性能的。

结论：ArrayList不适合做队列。

> 那数组适合用来做队列么？

这个女人是魔鬼么？不过还是得微笑面对！

数组是非常合适的。

比如ArrayBlockingQueue内部实现就是一个环形队列，它是一个定长队列，内部是用一个定长数组来实现的。

另外著名的Disruptor开源Library也是用环形数组来实现的超高性能队列，具体原理不做解释，比较复杂。

简单点说就是使用两个偏移量来标记数组的读位置和写位置，如果超过长度就折回到数组开头，前提是它们是定长数组。

> ArrayList的遍历和LinkedList遍历性能比较如何？

论遍历ArrayList要比LinkedList快得多，ArrayList遍历最大的优势在于内存的连续性，CPU的内部缓存结构会缓存连续的内存片段，可以大幅降低读取内存的性能开销。

> 能跟我聊一下LinkedList相关的东西么？

可以呀，不然今天天色已晚，不然我们下次再聊？

> 好吧，每次你都找借口，下次可以集合最后章节了，我们好好聊聊，你好好准备吧。

总结

ArrayList就是动态数组，用MSDN中的说法，就是Array的复杂版本，它提供了动态的增加和减少元素，实现了ICollection和IList接口，灵活的设置数组的大小等好处。

面试里面问的时候没HashMap，ConcurrentHashMap啥的这么常问，但是也有一定概率问到的，还是那句话，**不打没把握的仗**。

我们在源码阅读过程中，不需要全部都读懂，需要做的就是读懂核心的源码，加深自己对概念的理解就好了，用的时候不至于啥都不知道，不要为了用而用就好了。

ArrayList常用的方法总结

- boolean add(E e)

将指定的元素添加到此列表的尾部。

- void add(int index, E element)

将指定的元素插入此列表中的指定位置。

- boolean addAll(Collection c)

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。

- boolean addAll(int index, Collection c)

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。

- void clear()

移除此列表中的所有元素。

- Object clone()

返回此 ArrayList 实例的浅表副本。

- boolean contains(Object o)

如果此列表中包含指定的元素，则返回 true。

- void ensureCapacity(int minCapacity)

如有必要，增加此 ArrayList 实例的容量，以确保它至少能够容纳最小容量参数所指定的元素数。

- E get(int index)

返回此列表中指定位置上的元素。

- int indexOf(Object o)

返回此列表中首次出现的指定元素的索引，或如果此列表不包含元素，则返回 -1。

- boolean isEmpty()

如果此列表中没有元素，则返回 true

- int lastIndexOf(Object o)

返回此列表中最后一次出现的指定元素的索引，或如果此列表不包含索引，则返回 -1。

- E remove(int index)

移除此列表中指定位置上的元素。

- boolean remove(Object o)

移除此列表中首次出现的指定元素（如果存在）。

- protected void removeRange(int fromIndex, int toIndex)

移除列表中索引在 fromIndex（包括）和 toIndex（不包括）之间的所有元素。

- E set(int index, E element)

用指定的元素替代此列表中指定位置上的元素。

- int size()

返回此列表中的元素数。

- Object[] toArray()

按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组。

- T[] toArray(T[] a)

按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。

- void trimToSize()

将此 ArrayList 实例的容量调整为列表的当前大小。

点关注，不迷路

好了各位，以上就是这篇文章的全部内容了，能看到这里的人呀，都是**人才**。

©

我后面会每周都更新几篇一线互联网大厂面试和常用技术栈相关的文章，非常感谢**人才**们能看到这里，如果这个文章写得还不错，觉得「敖丙」我**有点东西**的话 **求点赞👍** **求关注❤️** **求分享👥** 对暖男我来说真的 **非常有用**！！！

白嫖不好，创作不易，各位的支持和认可，就是我创作的最大动力，我们下篇文章见！

敖丙 | 文 【原创】

如果本篇博客有任何错误，请批评指教，不胜感激 ！