ArrayList 源码解析

作者：huihui

原文链接：https://ld246.com/article/1514101642917

来源网站：链滴

```
package java.util;

import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.UnaryOperator;

public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
private static final long serialVersionUID = 8683452581122892189L;


//默认容量
private static final int DEFAULT_CAPACITY = 10;

//静态的一个属性，所有实例共享属性，当初始化容量为0的时候，就使用这个属性作为实例底层数组
private static final Object[] EMPTY_ELEMENTDATA = {};

/*根据注释，这个大概意思就是构造一个空的对象数组，用来与EMPTY_ELEMENTDATA 这个数组进
对比
来确定当第一次向ArrayList中添加数据时，应该如果进行扩容，就是增加多大的容量。*/
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

//实际上真正保存数据的数组，从此出可以看出ArrayList使用Object数组来保存数据
transient Object[] elementData; // non-private to simplify nested class access

//实际包含元素的个数
private int size;

/*
传递一个初始化容量的构造函数，会判断传递的参数与0的关系
如果大于0，会在ArrayList内部构建一个长度为initalCapacity的数组
如果等于0，会将上述的静态EMPTY_ELEMENTDATA属性赋值给elementData，也不会产生新的数
。如果小于0，则抛出异常
 */
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];//注意此处并没有将initialCapacity赋值给siz

    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}

/*
无参的构造函数，在该构造函数中，会将上述的静态的DEFAULTCAPACITY_EMPTY_ELEMENTDAT
属性，赋值给elementData属性
也即我们用这种方法构造ArraList的时候，并不会真正产生实例化的数组，而是引用一个静态的空数组
 */
```

```
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}

/*
传递一个集合给ArrayList，它首先会将集合转换成数组赋值给elementData
之后判断数组长度，如果等于0，则将elementData赋值为EMPTY_ELEMENTDATA
如果不等于0，还需要判断接受过来的数组(现在是elementData)是否是Object[]类型的
如果不是的化，将它转换成Object[]类型(根据注释，toArray方法有可能得到的不是Object[]类型)
 */
public ArrayList(Collection<? extends E> c) {
    elementData = c.toArray();
    if ((size = elementData.length) != 0) {
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    } else {
        // replace with empty array.
        this.elementData = EMPTY_ELEMENTDATA;
    }
}

/*
本质上是将数组的尾部删除掉形成新数组
新数组的length与size一致，节约空间
 */
public void trimToSize() {
    modCount++;
    if (size < elementData.length) {
        elementData = (size == 0)
          ? EMPTY_ELEMENTDATA
          : Arrays.copyOf(elementData, size);
    }
}

/*
增加这个ArrayList实例的能力，如果有必要，以确保它至少能容纳的最小容量参数指定元素个数。
提供给外界的方法，是的使用者可以通过这个方法自己去扩容
 */
public void ensureCapacity(int minCapacity) {
    int minExpand = (elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
        // any size if not default element table
        ? 0
        // larger than default for default empty table. It's already
        // supposed to be at default size.
        : DEFAULT_CAPACITY;//elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA意味
elementData可能不是一个length为0的数组

    if (minCapacity > minExpand) {
        ensureExplicitCapacity(minCapacity);
    }
}

/*
```

```
一个私有方法，确保minCapacity在容量范围内
如果elementData等于DEFAULTCAPACITY_EMPTY_ELEMENTDATA，则minCapacity会取DEFAUL
_CAPACITY, minCapacity中比较大的那个
也即如果minCapacity小于10，则取10，如果大于10，则去minCapacity
随后要执行ensureExplicitCapacity方法
 */
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}

/*
ensureExplicitCapacity要接受一个int类型的参数，意味着最少需要容量为minCapacity
首先会对modCount+1,modCount是AbstractList类中的一个成员变量,该值表示对List的修改次数，
要是为了服务快速失败功能的
随后如果minCapacity要大于现有数组elementData的长度的化，那么就执行grow方法，grow是扩
的方法
 */
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}


/*
数组所能开辟的最大长度
因为有些虚拟机保留了一些header words在数组中
尝试要开辟更大的长度的数组，可能会出现OOM异常(在一些虚拟机实现中)
 */
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

/*
ArrayList的扩容，接收一个int类型参数，表示至少需要多少容量
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;//得到目前的容量
    //oldCapacity>>1表示除2取整数，该式子最终表示意思为newCapacity大于为oldCapacity的1.
倍数
    int newCapacity = oldCapacity + (oldCapacity >> 1);
      //判断newCapacity是否溢出
    if (newCapacity - minCapacity < 0)
        //溢出：newCapacity等于minCapacity
        newCapacity = minCapacity;
    //判断newCapacity是否超过了MAX_ARRAY_SIZE，超过了，则计算最大容量；具体原因是因为
同虚拟机的实现不同
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    //执行Arrays.copyOf方法，传递原数组与新数组长度，由Arrays内部创建数组返回并接受给eleme
tData
    elementData = Arrays.copyOf(elementData, newCapacity);
}

/*
求出最大的容量值，首先判断minCapacity是否已经溢出了，溢出了就直接抛出OOM
否则就去判断minCapacity 是否大于 MAX_ARRAY_SIZE
  大于返回 Integer.MAX_VALUE ， 不大于 返回MAX_ARRAY_SIZE
 */
private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}

//得到size，size是真正的保存的元素的数量
public int size() {
    return size;
}

//判断容器是否为空(指是不包含元素)
public boolean isEmpty() {
    return size == 0;
}

//判断容器是否包含某个元素
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

//indexOf是来获得o元素(包括null)在容器中的位置的,位置从0开始到size-1结束,如果返回-1表示不
含
//对于重复的元素，只获取第一个所在的位置
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

//与indexOf功能一样，但是确实获得重复元素的最后一个位置
public int lastIndexOf(Object o) {
    if (o == null) {
        for (int i = size-1; i >= 0; i--)
            if (elementData[i]==null)
```

```
                return i;
    } else {
        for (int i = size-1; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

//重写了Object中的clone方法，用于赋值容器，浅复制
public Object clone() {
    try {
        ArrayList<?> v = (ArrayList<?>) super.clone();
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {//看来大神也感觉异常不好处理了...不可能发生异常
地方，却还是要处理...
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
}

//得到数组的副本
public Object[] toArray() {
    return Arrays.copyOf(elementData, size);
}

/*
给定一个指定数组，返回指定数组大小，类型的副本
 */
@SuppressWarnings("unchecked")
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);//此处是size == a.length
    if (a.length > size)
        a[size] = null;//如果a.length>size，则截取size的长度，但是如果a本身就是有数据的，可能会
现a[size+?]有数据，而a[size]为null
    return a;
}

// Positional Access Operations
//不需要检查index的快速访问元素，但是是包权限，只允许内部使用
@SuppressWarnings("unchecked")
E elementData(int index) {
    return (E) elementData[index];
}

/*
判断一下是否index是否越界
然后通过快速访问来返回元素
 */
```

```
public E get(int index) {
    rangeCheck(index);

    return elementData(index);
}

/*
判断一下是否越界
然后得到处于index位置的原元素，随后将index位置置入新元素
返回原来的元素
要求 index<size
 */
public E set(int index, E element) {
    rangeCheck(index);

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

/*
集合中新增一个元素，首先要确保在承受能力范围内
之后将新加入进来的元素赋值到数组的第size的位置上
随后size+1
新增的元素，插入到数组的末尾
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}

/*
插入一个元素element到指定index位置，原位置的元素依次向后移动一位
改方法效率要低一些，如果并不是特定必须要塞入哪个位置的话，最好不要用
 */
public void add(int index, E element) {
    //首先会去检查一下index是否可以使用
    rangeCheckForAdd(index);
    //确保数组可容纳
    ensureCapacityInternal(size + 1);  // Increments modCount!! 会修改modCount的值，modC
unt+1
    //随后调用System.arraycopy方法，将elementData的index位置元素依次向后移动，为接下来的
入预留空间
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;//真正的插入操作
    size++;//size+1
}

/*
删除指定位置的元素，如果index>size的话，会出现数组越界
 */
public E remove(int index) {
    rangeCheck(index);//index>size throw IndexOutOfBoundsException

    modCount++;
    E oldValue = elementData(index);//得到原来elementData中的元素

    int numMoved = size - index - 1;//计算删除之后需要移动元素的数量
    if (numMoved > 0)//移动元素
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);//移动的时候，就会覆盖原来的元素
    //清除最后一个元素的引用，因为原来的元素以及被删除了
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;//返回被删除的元素
}

/*
删除某一个元素，传入要被删除的元素
 */
public boolean remove(Object o) {
    if (o == null) {//删除null元素
        for (int index = 0; index < size; index++)//迭代ArrayList
            if (elementData[index] == null) {//如果在size之前的位置有存在空元素
                fastRemove(index);//则快速删除(所谓快速删除，就是不去做越界检查以及不返回结果，
全给本类自己使用的private方法)
                return true;
            }
    } else {//删除非空元素，与删除null元素逻辑相同
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {//此处使用equals方法来进行比较，所以在使用remove
Object o)的时候，要考虑是否重写了equals方法
                fastRemove(index);//fastRemove也是会移动数组的，如果有删除重复元素的时候，效率
低
                return true;
            }
    }
    return false;
}

/*
快速删除
不做index检查，只允许内部使用
 */
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
}

/*
清除数组，所有元素置为null
 */
```

```
public void clear() {
    modCount++;

    // clear to let GC do its work
    for (int i = 0; i < size; i++)
        elementData[i] = null;

    size = 0;
}

/*
添加一次性add多个元素，接受参数为集合类型
 */
public boolean addAll(Collection<? extends E> c) {
    Object[] a = c.toArray();
    int numNew = a.length;//可能会产生空指针错误
    ensureCapacityInternal(size + numNew);  // Increments modCount
    //将a数组插入到elementData的size位置
    System.arraycopy(a, 0, elementData, size, numNew);
    size += numNew;
    return numNew != 0;
}

/*
指定index位置插入多个元素，原来位置的元素依次向后移动
index不能大于size,如果大于size会产生数组越界
 */
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);

    Object[] a = c.toArray();
    int numNew = a.length;//可能会产生空指针错误
    ensureCapacityInternal(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;
}

/*
范围删除，删除从fromIndex~toIndex,包含fromIndex，不包含toIndex
 */
protected void removeRange(int fromIndex, int toIndex) {
    modCount++;
    int numMoved = size - toIndex;
    System.arraycopy(elementData, toIndex, elementData, fromIndex,
                     numMoved);

    // clear to let GC do its work
    int newSize = size - (toIndex-fromIndex);
    for (int i = newSize; i < size; i++) {
        elementData[i] = null;
    }
    size = newSize;
}

//index检查判断，专门封装起来是因为很多地方使用
private void rangeCheck(int index) {
    if (index >= size)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

//专门为add方法封装的rangeCheck方法
private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

//为IndexOutOfBoundsException提供信息的方法，告诉哪个位置出现了数组越界
private String outOfBoundsMsg(int index) {
    return "Index: "+index+", Size: "+size;
}

//一次性删除多个元素
public boolean removeAll(Collection<?> c) {
    Objects.requireNonNull(c);//判断c是否为空，为空抛出异常
    return batchRemove(c, false);//批量删除
}

//保留当前容器与c的并集，并返回
public boolean retainAll(Collection<?> c) {
    Objects.requireNonNull(c);
    return batchRemove(c, true);
}

//批量删除方法，complement为true表示求交集，如果为false表示在elementData中保留原有的非
的集合
//也即true: a属于elementData同时a属于c; false: a属于elementData同时a不属于c
private boolean batchRemove(Collection<?> c, boolean complement) {
    final Object[] elementData = this.elementData;
    int r = 0, w = 0;//一个读的index,一个是写的index
    boolean modified = false;
    try {
        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) {//只移动一次数组，比单独remove效果要好
            System.arraycopy(elementData, r,
                             elementData, w,
                             size - r);
            w += size - r;
        }
        if (w != size) {//清理数组中不需要的引用
            // clear to let GC do its work
            for (int i = w; i < size; i++)
                elementData[i] = null;
            modCount += size - w;//记录修改次数
            size = w;//重新定义size
            modified = true;
        }
    }
    return modified;
}

//保存数组实例的状态到一个流（即它序列化）
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out size as capacity for behavioural compatibility with clone()
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}

//从一个流中读出数组实例的状态
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    elementData = EMPTY_ELEMENTDATA;

    // Read in size, and any hidden stuff
    s.defaultReadObject();

    // Read in capacity
    s.readInt(); // ignored

    if (size > 0) {
        // be like clone(), allocate array based upon size not capacity
        ensureCapacityInternal(size);

        Object[] a = elementData;
        // Read in all elements in the proper order.
        for (int i=0; i<size; i++) {
            a[i] = s.readObject();
        }
    }
}

//返回一个list迭代器，链表迭代器，可以双向迭代，并且还具有add方法，但是只有在list类型中才
以使用，别的集合类没有
//接受一个Index，确定迭代器初始的位置
public ListIterator<E> listIterator(int index) {
    if (index < 0 || index > size)//先判断index是否合法
        throw new IndexOutOfBoundsException("Index: "+index);
    return new ListItr(index);
}

/**
 * Returns a list iterator over the elements in this list (in proper
 * sequence).
 *
 * <p>The returned list iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
 *
 * @see #listIterator(int)
 */
public ListIterator<E> listIterator() {
    return new ListItr(0);
}

/**
 * Returns an iterator over the elements in this list in proper sequence.
 *
 * <p>The returned iterator is <a href="#fail-fast"><i>fail-fast</i></a>.
 *
 * @return an iterator over the elements in this list in proper sequence
 */
public Iterator<E> iterator() {
    return new Itr();
}

/**
 * An optimized version of AbstractList.Itr
 * AbstractList.Itr的优化版本迭代器
 */
private class Itr implements Iterator<E> {
    int cursor;       // 下一个要被返回元素的下标
    int lastRet = -1; // 上一个被返回的元素的下标，如果没有的话默认为-1
    int expectedModCount = modCount;

    //判断是否还有下一个元素
    public boolean hasNext() {
        return cursor != size;
    }

    //返回下一个元素，默认一开始的next是第一个元素
    @SuppressWarnings("unchecked")
    public E next() {
        checkForComodification();//快速失败
        int i = cursor;
        if (i >= size)//会判断一次位置是否合法，因为cursor只是盲目的+1
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;//cursor设置为下一个要被返回的元素下标
        return (E) elementData[lastRet = i];//将lastRet设置为被返回的元素下标
    }

    //删除上一个元素，也即最近被next()出来的元素
    public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();

        try {
            ArrayList.this.remove(lastRet);//删除的是下标为lastRet元素
            cursor = lastRet;//回退
            lastRet = -1;//设置成为-1，也即不能连续的删除，该类不能够往回走，只能继续前进，因为
续删除，会抛出IllegalStateException异常
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }

    /*
    遍历余下的元素
     */
    @Override
    @SuppressWarnings("unchecked")
    public void forEachRemaining(Consumer<? super E> consumer) {
        Objects.requireNonNull(consumer);//判断consumer不能为null
        final int size = ArrayList.this.size;
        int i = cursor;//余下的体现在这..
        if (i >= size) {
            return;
        }
        final Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length) {
            throw new ConcurrentModificationException();
        }
        while (i != size && modCount == expectedModCount) {
            consumer.accept((E) elementData[i++]);//此处接受elementData元素，执行consumer中
方法，可能会去改变elementData元素
        }
        // update once at end of iteration to reduce heap write traffic
        cursor = i;
        lastRet = i - 1;
        checkForComodification();
    }

    final void checkForComodification() {
        if (modCount != expectedModCount)
```

```
            throw new ConcurrentModificationException();
        }
    }

    /**
     * An optimized version of AbstractList.ListItr
     * 一个对AbstractList.ListItr的优化版本链表迭代器
     */
    private class ListItr extends Itr implements ListIterator<E> {
        ListItr(int index) {
            super();
            cursor = index;
        }

        public boolean hasPrevious() {
            return cursor != 0;
        }

        public int nextIndex() {
            return cursor;
        }

        public int previousIndex() {
            return cursor - 1;
        }

        //返回上一个元素
        @SuppressWarnings("unchecked")
        public E previous() {
            checkForComodification();
            int i = cursor - 1;
            if (i < 0)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i;
            return (E) elementData[lastRet = i];
        }

        //更新上一个位置的元素，将其置换成e
        public void set(E e) {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                ArrayList.this.set(lastRet, e);
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }

        //新增一个元素，处在上一个元素之后，下一个元素之前，会移动数组
        public void add(E e) {
            checkForComodification();

            try {
                int i = cursor;
                ArrayList.this.add(i, e);
                cursor = i + 1;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }
    }

    //得到子列表 从fromIndex~toIndex位置
    public List<E> subList(int fromIndex, int toIndex) {
        subListRangeCheck(fromIndex, toIndex, size);
        return new SubList(this, 0, fromIndex, toIndex);
    }

    //判断Index是否合法
    static void subListRangeCheck(int fromIndex, int toIndex, int size) {
        if (fromIndex < 0)
            throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
        if (toIndex > size)
            throw new IndexOutOfBoundsException("toIndex = " + toIndex);
        if (fromIndex > toIndex)
            throw new IllegalArgumentException("fromIndex(" + fromIndex +
                                               ") > toIndex(" + toIndex + ")");
    }

    //继承与AbstractList的SubList类，其实这个类，只是去封装了几个属性，实际上用的还是原来ArrayL
    st类的数组，外观模式
    private class SubList extends AbstractList<E> implements RandomAccess {
        private final AbstractList<E> parent;
        private final int parentOffset;
        private final int offset;
        int size;

        //参数:
        //parent 父类型
        //offset 父类型的偏移量
        //fromIndex 子列表的开始元素，位于父列表的位置
        //toIndex 子列表的结束元素，位于父列表的位置
        SubList(AbstractList<E> parent,
                int offset, int fromIndex, int toIndex) {
            this.parent = parent;
            this.parentOffset = fromIndex;
            this.offset = offset + fromIndex;
            this.size = toIndex - fromIndex;
            this.modCount = ArrayList.this.modCount;
        }
```

```
public E set(int index, E e) {
    rangeCheck(index);
    checkForComodification();
    E oldValue = ArrayList.this.elementData(offset + index);
    ArrayList.this.elementData[offset + index] = e;
    return oldValue;
}

public E get(int index) {
    rangeCheck(index);
    checkForComodification();
    return ArrayList.this.elementData(offset + index);
}

public int size() {
    checkForComodification();
    return this.size;
}

public void add(int index, E e) {
    rangeCheckForAdd(index);
    checkForComodification();
    parent.add(parentOffset + index, e);
    this.modCount = parent.modCount;
    this.size++;
}

public E remove(int index) {
    rangeCheck(index);
    checkForComodification();
    E result = parent.remove(parentOffset + index);
    this.modCount = parent.modCount;
    this.size--;
    return result;
}

protected void removeRange(int fromIndex, int toIndex) {
    checkForComodification();
    parent.removeRange(parentOffset + fromIndex,
                       parentOffset + toIndex);
    this.modCount = parent.modCount;
    this.size -= toIndex - fromIndex;
}

public boolean addAll(Collection<? extends E> c) {
    return addAll(this.size, c);
}

public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);
    int cSize = c.size();
    if (cSize==0)
        return false;
        checkForComodification();
        parent.addAll(parentOffset + index, c);
        this.modCount = parent.modCount;
        this.size += cSize;
        return true;
    }

    public Iterator<E> iterator() {
        return listIterator();
    }

    public ListIterator<E> listIterator(final int index) {
        checkForComodification();
        rangeCheckForAdd(index);
        final int offset = this.offset;

        return new ListIterator<E>() {
            int cursor = index;
            int lastRet = -1;
            int expectedModCount = ArrayList.this.modCount;

            public boolean hasNext() {
                return cursor != SubList.this.size;
            }

            @SuppressWarnings("unchecked")
            public E next() {
                checkForComodification();
                int i = cursor;
                if (i >= SubList.this.size)
                    throw new NoSuchElementException();
                Object[] elementData = ArrayList.this.elementData;
                if (offset + i >= elementData.length)
                    throw new ConcurrentModificationException();
                cursor = i + 1;
                return (E) elementData[offset + (lastRet = i)];
            }

            public boolean hasPrevious() {
                return cursor != 0;
            }

            @SuppressWarnings("unchecked")
            public E previous() {
                checkForComodification();
                int i = cursor - 1;
                if (i < 0)
                    throw new NoSuchElementException();
                Object[] elementData = ArrayList.this.elementData;
                if (offset + i >= elementData.length)
                    throw new ConcurrentModificationException();
                cursor = i;
                return (E) elementData[offset + (lastRet = i)];
            }
```

```
@SuppressWarnings("unchecked")
public void forEachRemaining(Consumer<? super E> consumer) {
    Objects.requireNonNull(consumer);
    final int size = SubList.this.size;
    int i = cursor;
    if (i >= size) {
        return;
    }
    final Object[] elementData = ArrayList.this.elementData;
    if (offset + i >= elementData.length) {
        throw new ConcurrentModificationException();
    }
    while (i != size && modCount == expectedModCount) {
        consumer.accept((E) elementData[offset + (i++)]);
    }
    // update once at end of iteration to reduce heap write traffic
    lastRet = cursor = i;
    checkForComodification();
}

public int nextIndex() {
    return cursor;
}

public int previousIndex() {
    return cursor - 1;
}

public void remove() {
    if (lastRet < 0)
        throw new IllegalStateException();
    checkForComodification();

    try {
        SubList.this.remove(lastRet);
        cursor = lastRet;
        lastRet = -1;
        expectedModCount = ArrayList.this.modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}

public void set(E e) {
    if (lastRet < 0)
        throw new IllegalStateException();
    checkForComodification();

    try {
        ArrayList.this.set(offset + lastRet, e);
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
```

```
            }

            public void add(E e) {
                checkForComodification();

                try {
                    int i = cursor;
                    SubList.this.add(i, e);
                    cursor = i + 1;
                    lastRet = -1;
                    expectedModCount = ArrayList.this.modCount;
                } catch (IndexOutOfBoundsException ex) {
                    throw new ConcurrentModificationException();
                }
            }

            final void checkForComodification() {
                if (expectedModCount != ArrayList.this.modCount)
                    throw new ConcurrentModificationException();
            }
        };
    }

    public List<E> subList(int fromIndex, int toIndex) {
        subListRangeCheck(fromIndex, toIndex, size);
        return new SubList(this, offset, fromIndex, toIndex);
    }

    private void rangeCheck(int index) {
        if (index < 0 || index >= this.size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

    private void rangeCheckForAdd(int index) {
        if (index < 0 || index > this.size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

    private String outOfBoundsMsg(int index) {
        return "Index: "+index+", Size: "+this.size;
    }

    private void checkForComodification() {
        if (ArrayList.this.modCount != this.modCount)
            throw new ConcurrentModificationException();
    }

    public Spliterator<E> spliterator() {
        checkForComodification();
        return new ArrayListSpliterator<E>(ArrayList.this, offset,
                                           offset + this.size, this.modCount);
    }
}
```

```
//与forEachRemaining很像，一个是迭代所有，一个是迭代剩余，都会去执行Consumer中定义的
法，可能会改变元素的值
@Override
public void forEach(Consumer<? super E> action) {
    Objects.requireNonNull(action);
    final int expectedModCount = modCount;
    @SuppressWarnings("unchecked")
    final E[] elementData = (E[]) this.elementData;
    final int size = this.size;
    for (int i=0; modCount == expectedModCount && i < size; i++) {
        action.accept(elementData[i]);
    }
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}

//返回spliterator,用于并行计算中，splitable iterator可分割迭代器
@Override
public Spliterator<E> spliterator() {
    return new ArrayListSpliterator<>(this, 0, -1, 0);
}

static final class ArrayListSpliterator<E> implements Spliterator<E> {

    private final ArrayList<E> list;//原数组
    private int index; // current index, modified on advance/split
    private int fence; // -1 until used; then one past last index
    private int expectedModCount; // initialized when fence set

    /** Create new spliterator covering the given  range */
    ArrayListSpliterator(ArrayList<E> list, int origin, int fence,
                         int expectedModCount) {
        this.list = list; // OK if null unless traversed
        this.index = origin;
        this.fence = fence;
        this.expectedModCount = expectedModCount;
    }

    private int getFence() { // 第一次使用时，初始化fence大小
        int hi; // (a specialized variant appears in method forEach)
        ArrayList<E> lst;
        if ((hi = fence) < 0) { //-1表示初始化的值
            if ((lst = list) == null)
                hi = fence = 0;
            else {
                expectedModCount = lst.modCount;
                hi = fence = lst.size;
            }
        }
        return hi;
    }

    //这就是为Spliterator专门设计的方法，区分与普通的Iterator，该方法会把当前元素划分一部分
```

```
去创建一个新的Spliterator作为返回，
    //两个Spliterator变会并行执行，如果元素个数小到无法划分则返回null
    public ArrayListSpliterator<E> trySplit() {
        int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;//由于lo + hi都是整数，>>>相当于
2
        return (lo >= mid) ? null : // divide range in half unless too small
            new ArrayListSpliterator<E>(list, lo, index = mid,//注意index=min
                                        expectedModCount);
    }

    //tryAdvance就是顺序处理每个元素，类似Iterator，如果还有元素要处理，则返回true，否则返回
alse
    public boolean tryAdvance(Consumer<? super E> action) {
        if (action == null)
            throw new NullPointerException();
        int hi = getFence(), i = index;
        if (i < hi) {
            index = i + 1;
            @SuppressWarnings("unchecked") E e = (E)list.elementData[i];
            action.accept(e);
            if (list.modCount != expectedModCount)
                throw new ConcurrentModificationException();
            return true;
        }
        return false;
    }

    public void forEachRemaining(Consumer<? super E> action) {
        int i, hi, mc; // hoist accesses and checks from loop
        ArrayList<E> lst; Object[] a;
        if (action == null)
            throw new NullPointerException();
        if ((lst = list) != null && (a = lst.elementData) != null) {
            if ((hi = fence) < 0) {
                mc = lst.modCount;
                hi = lst.size;
            }
            else
                mc = expectedModCount;
            if ((i = index) >= 0 && (index = hi) <= a.length) {
                for (; i < hi; ++i) {
                    @SuppressWarnings("unchecked") E e = (E) a[i];
                    action.accept(e);
                }
                if (lst.modCount == mc)
                    return;
            }
        }
        throw new ConcurrentModificationException();
    }

    //该方法用于估算还剩下多少个元素需要遍历
    public long estimateSize() {
        return (long) (getFence() - index);
    }

    //其实就是表示该Spliterator有哪些特性，用于可以更好控制和优化Spliterator的使用
    public int characteristics() {
        return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
    }
}

//删除，增加过滤功能
@Override
public boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);//判断过滤器是否为空
    // figure out which elements are to be removed
    // any exception thrown from the filter predicate at this stage
    // will leave the collection unmodified
    int removeCount = 0;//要删除元素的个数
    final BitSet removeSet = new BitSet(size);//使用BitSet类来保存要被删除的Set，BitSet是使用
图来保存数据，节省很大内存
    final int expectedModCount = modCount;//预期的modCount
    final int size = this.size;
    for (int i=0; modCount == expectedModCount && i < size; i++) {
        @SuppressWarnings("unchecked")
        final E element = (E) elementData[i];
        if (filter.test(element)) {//如果element匹配filter中的过滤条件的话，则会返回true
            removeSet.set(i);//使用位图来保存要被删除的index
            removeCount++;
        }
    }
    if (modCount != expectedModCount) {//快速失败机制，在多线程情况下，去报错，引起程序
注意
        throw new ConcurrentModificationException();
    }

    // shift surviving elements left over the spaces left by removed elements
    final boolean anyToRemove = removeCount > 0;//用于记录是否需要删除
    if (anyToRemove) {
        //需要删除...
        final int newSize = size - removeCount;//计算剩余的长度，也即新数组的长度
        for (int i=0, j=0; (i < size) && (j < newSize); i++, j++) {//(i < size) && (j < newSize)会节
一些效率
            i = removeSet.nextClearBit(i);//得到没有被拦截的index
            elementData[j] = elementData[i];
        }
        for (int k=newSize; k < size; k++) {//清除数组后面的多余引用，GC
            elementData[k] = null;  // Let gc do its work
        }
        this.size = newSize;
        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
        modCount++;//用于记录本条数据也改变了数组结构，从这个地方可以看出来，快速失败机制
不能完全确保一定会提醒到程序员，只是有可能
    }
    return anyToRemove;
}

//替换所有
@Override
@SuppressWarnings("unchecked")
public void replaceAll(UnaryOperator<E> operator) {
    Objects.requireNonNull(operator);
    final int expectedModCount = modCount;
    final int size = this.size;
    for (int i=0; modCount == expectedModCount && i < size; i++) {
        elementData[i] = operator.apply((E) elementData[i]);
    }
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
    modCount++;
}

@Override
@SuppressWarnings("unchecked")
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++;
}

}
```

原文链接：ArrayList 源码解析