ArrayList简介： ArrayList底层是数组队列，相当于动态数组，在操作过程中可以使用ensureCapacity操作来增加ArrayList实例的容量。

ArrayList源码： package java.util;

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

//继承了AbstractList类，实现了List接口，RandomAccess，Cloneable,Serializable接口
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    /*
    serialVersionUID适用于java序列化机制。简单来说，JAVA序列化的机制是通过判断类的serialVersionUID来验证的版本一致的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID于本地相应实体类的serialVersionUID进行比较。如果相同说明是一致的，可以进行反序列化，否则会出现反序列化版本一致的异常
    */
    private static final long serialVersionUID = 8683452581122892189L;

    /**
    * 默认初始容量大小
    */
    private static final int DEFAULT_CAPACITY = 10;

    /**
    * 空数组（用于空实例）
    */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    //用于默认大小空实例的共享空数组实例。
    //我们把它从EMPTY_ELEMENTDATA数组中区分出来，以知道在添加第一个元素时容量需要增加多少。
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    /**
    * 保存ArrayList数据的数组
    */
    transient Object[] elementData; // non-private to simplify nested class access

    /**
    * ArrayList 所包含的元素个数
    */
    private int size;

    /**
    * 带初始容量参数的构造函数。（用户自己指定容量）
    */
```

```
    //自己设置容量的构造器
    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            //创建initialCapacity大小的数组
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            //创建空数组
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }
    /**
    *默认构造函数，DEFAULTCAPACITY_EMPTY_ELEMENTDATA 为0.初始化为10，也就是说初始其实是空数组 当添加第一个元素的时候数组容量才变成10
    */
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    /**
    * 构造一个包含指定集合的元素的列表，按照它们由集合的迭代器返回的顺序。
    */
    public ArrayList(Collection<? extends E> c) {
        //
        elementData = c.toArray();
        //如果指定集合元素个数不为0
        if ((size = elementData.length) != 0) {
            // c.toArray 可能返回的不是Object类型的数组所以加上下面的语句用于判断，
            //这里用到了反射里面的getClass()方法
            if (elementData.getClass() != Object[].class)
                elementData = Arrays.copyOf(elementData, size, Object[].class);
        } else {
            // 用空数组代替
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }

    /**
    * 修改这个ArrayList实例的容量是列表的当前大小。 应用程序可以使用此操作来最小化ArrayList实例的存储。
    */
    public void trimToSize() {
        modCount++;
        if (size < elementData.length) {
            elementData = (size == 0)
            ? EMPTY_ELEMENTDATA
            : Arrays.copyOf(elementData, size);
        }
    }
```

```
//下面是ArrayList的扩容机制
//ArrayList的扩容机制提高了性能，如果每次只扩充一个，
//那么频繁的插入会导致频繁的拷贝，降低性能，而ArrayList的扩容机制避免了这种情况。
    /**
    * 如有必要，增加此ArrayList实例的容量，以确保它至少能容纳元素的数量
    * @param   minCapacity   所需的最小容量
    */
    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;

        if (minCapacity > minExpand) {
            ensureExplicitCapacity(minCapacity);
        }
    }
//得到最小扩容量
    private void ensureCapacityInternal(int minCapacity) {
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            // 获取默认的容量和传入参数的较大值
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }

        ensureExplicitCapacity(minCapacity);
    }
//判断是否需要扩容
    private void ensureExplicitCapacity(int minCapacity) {
        modCount++;

        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            //调用grow方法进行扩容，调用此方法代表已经开始扩容了
            grow(minCapacity);
    }

    /**
    * 要分配的最大数组大小
    */
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

    /**
    * ArrayList扩容的核心方法。
    */
    private void grow(int minCapacity) {
        // oldCapacity为旧容量，newCapacity为新容量
        int oldCapacity = elementData.length;
        //将oldCapacity 右移一位，其效果相当于oldCapacity /2，
        //我们知道位运算的速度远远快于整除运算，整句运算式的结果就是将新容量更新为旧容量
```

```
的1.5倍，
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        //然后检查新容量是否大于最小需要容量，若还是小于最小需要容量，那么就把最小需要容
量当作数组的新容量，
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        //再检查新容量是否超出了ArrayList所定义的最大容量，
        //若超出了，则调用hugeCapacity()来比较minCapacity和 MAX_ARRAY_SIZE，
        //如果minCapacity大于MAX_ARRAY_SIZE，则新容量则为Interger.MAX_VALUE，否则，新
容量大小则为 MAX_ARRAY_SIZE。
        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);
    }
    //比较minCapacity和 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;
    }

    /**
    *返回此列表中的元素数。
    */
    public int size() {
        return size;
    }

    /**
    * 如果此列表不包含元素，则返回 true 。
    */
    public boolean isEmpty() {
        //注意=和==的区别
        return size == 0;
    }

    /**
    * 如果此列表包含指定的元素，则返回true 。
    */
    public boolean contains(Object o) {
        //indexOf()方法：返回此列表中指定元素的首次出现的索引，如果此列表不包含此元素，
则为-1
        return indexOf(o) >= 0;
    }

    /**
    *返回此列表中指定元素的首次出现的索引，如果此列表不包含此元素，则为-1
    */
```

```
    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++)
                //equals()方法比较
                if (o.equals(elementData[i]))
                    return i;
        }
        return -1;
    }

    /**
    * 返回此列表中指定元素的最后一次出现的索引，如果此列表不包含元素，则返回-1。.
    */
    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;
    }

    /**
    * 返回此ArrayList实例的浅拷贝。 （元素本身不被复制。）
    */
    public Object clone() {
        try {
            ArrayList<?> v = (ArrayList<?>) super.clone();
            //Arrays.copyOf功能是实现数组的复制，返回复制后的数组。参数是被复制的数组和
复制的长度
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            // 这不应该发生，因为我们是可以克隆的
            throw new InternalError(e);
        }
    }

    /**
    *以正确的顺序（从第一个到最后一个元素）返回一个包含此列表中所有元素的数组。
    *返回的数组将是“安全的”，因为该列表不保留对它的引用。 （换句话说，这个方法必须分配一
个新的数组）。
```

```
    *因此，调用者可以自由地修改返回的数组。 此方法充当基于阵列和基于集合的API之间的桥梁。
    */
    public Object[] toArray() {
        return Arrays.copyOf(elementData, size);
    }

    /**
    * 以正确的顺序返回一个包含此列表中所有元素的数组（从第一个到最后一个元素）；
    *返回的数组的运行时类型是指定数组的运行时类型。 如果列表适合指定的数组，则返回其中。
    *否则，将为指定数组的运行时类型和此列表的大小分配一个新数组。
    *如果列表适用于指定的数组，其余空间（即数组的列表数量多于此元素），则紧跟在集合结束后的数组中的元素设置为null 。
    *（这仅在调用者知道列表不包含任何空元素的情况下才能确定列表的长度。）
    */
    @SuppressWarnings("unchecked")
    public <T> T[] toArray(T[] a) {
        if (a.length < size)
            // 新建一个运行时类型的数组，但是ArrayList数组的内容
            return (T[]) Arrays.copyOf(elementData, size, a.getClass());
            //调用System提供的arraycopy()方法实现数组之间的复制
        System.arraycopy(elementData, 0, a, 0, size);
        if (a.length > size)
            a[size] = null;
        return a;
    }

    // Positional Access Operations

    @SuppressWarnings("unchecked")
    E elementData(int index) {
        return (E) elementData[index];
    }

    /**
    * 返回此列表中指定位置的元素。
    */
    public E get(int index) {
        rangeCheck(index);

        return elementData(index);
    }

    /**
    * 用指定的元素替换此列表中指定位置的元素。
    */
    public E set(int index, E element) {
        //对index进行界限检查
        rangeCheck(index);

        E oldValue = elementData(index);
        elementData[index] = element;
```

```
        //返回原来在这个位置的元素
        return oldValue;
    }

    /**
    * 将指定的元素追加到此列表的末尾。
    */
    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        //这里看到ArrayList添加元素的实质就相当于为数组赋值
        elementData[size++] = e;
        return true;
    }

    /**
    * 在此列表中的指定位置插入指定的元素。
    *先调用 rangeCheckForAdd 对index进行界限检查；然后调用 ensureCapacityInternal 方法保证capacity足够大；
    *再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。
    */
    public void add(int index, E element) {
        rangeCheckForAdd(index);

        ensureCapacityInternal(size + 1);  // Increments modCount!!
        //arraycopy()这个实现数组之间复制的方法一定要看一下，下面就用到了arraycopy()方法实现数组自己复制自己
        System.arraycopy(elementData, index, elementData, index + 1,
                         size - index);
        elementData[index] = element;
        size++;
    }

    /**
    * 删除该列表中指定位置的元素。 将任何后续元素移动到左侧（从其索引中减去一个元素）。
    */
    public E remove(int index) {
        rangeCheck(index);

        modCount++;
        E oldValue = elementData(index);

        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
    //从列表中删除的元素
        return oldValue;
    }

    /**
```

```
    * 从列表中删除指定元素的第一个出现（如果存在）。 如果列表不包含该元素，则它不会更改。
    *返回true，如果此列表包含指定的元素
    */
    public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }

    /*
    * Private remove method that skips bounds checking and does not
    * return the value removed.
    */
    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
    }

    /**
    * 从列表中删除所有元素。
    */
    public void clear() {
        modCount++;

        // 把数组中所有的元素的值设为null
        for (int i = 0; i < size; i++)
            elementData[i] = null;

        size = 0;
    }

    /**
    * 按指定集合的Iterator返回的顺序将指定集合中的所有元素追加到此列表的末尾。
    */
    public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
```

```
        int numNew = a.length;
        ensureCapacityInternal(size + numNew);  // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
        return numNew != 0;
    }

    /**
    * 将指定集合中的所有元素插入到此列表中，从指定的位置开始。
    */
    public boolean addAll(int index, Collection<? extends E> c) {
        rangeCheckForAdd(index);

        Object[] a = c.toArray();
        int numNew = a.length;
        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;
    }

    /**
    * 从此列表中删除所有索引为fromIndex （含）和toIndex之间的元素。
    *将任何后续元素移动到左侧（减少其索引）。
    */
    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;
    }

    /**
    * 检查给定的索引是否在范围内。
    */
    private void rangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
```

```
    }

    /**
    * add和addAll使用的rangeCheck的一个版本
    */
    private void rangeCheckForAdd(int index) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

    /**
    * 返回IndexOutOfBoundsException细节信息
    */
    private String outOfBoundsMsg(int index) {
        return "Index: "+index+", Size: "+size;
    }

    /**
    * 从此列表中删除指定集合中包含的所有元素。
    */
    public boolean removeAll(Collection<?> c) {
        Objects.requireNonNull(c);
        //如果此列表被修改则返回true
        return batchRemove(c, false);
    }

    /**
    * 仅保留此列表中包含在指定集合中的元素。
    *换句话说，从此列表中删除其中不包含在指定集合中的所有元素。
    */
    public boolean retainAll(Collection<?> c) {
        Objects.requireNonNull(c);
        return batchRemove(c, true);
    }


    /**
    * 从列表中的指定位置开始，返回列表中的元素（按正确顺序）的列表迭代器。
    *指定的索引表示初始调用将返回的第一个元素为next 。 初始调用previous将返回指定索引减1
的元素。
    *返回的列表迭代器是fail-fast 。
    */
    public ListIterator<E> listIterator(int index) {
        if (index < 0 || index > size)
            throw new IndexOutOfBoundsException("Index: "+index);
        return new ListItr(index);
    }

    /**
    *返回列表中的列表迭代器（按适当的顺序）。
    *返回的列表迭代器是fail-fast 。
```

```
    */
    public ListIterator<E> listIterator() {
        return new ListItr(0);
    }

    /**
    *以正确的顺序返回该列表中的元素的迭代器。
    *返回的迭代器是fail-fast 。
    */
    public Iterator<E> iterator() {
        return new Itr();
    }
```