数组的核心就是一段连续的内存，在Java中数组是特殊的类，所以还会有一个对象头。

Java中把内存分为了堆内存和栈内存，当初始化数组时，堆内存分配相应大小的连续的内存块，并将第一个内存块的地址放入栈内存中存储。这样读取数据的时候取第0个就是首地址的内存中的数据，第1个就是首地址+1的内存块中数据。其余删除与写入操作与读取类似。

#### 数组的复制

*/\*\**

*\* 复制数组方法一：System类中的方法  
 \*  
 \** ***@param*** *a  
 \** ***@return*** *\*/*public int[] copy1(int[] a) {  
 int[] cpA = new int[a.length];  
 System.*arraycopy*(a, 0, cpA, 0, a.length);  
 return cpA;  
}  
  
*/\*\*  
 \* 复制数组方法二：使用数组工具类中的方法  
 \*  
 \** ***@param*** *a  
 \** ***@return*** *\*/*public int[] copy2(int[] a) {  
 return Arrays.*copyOf*(a, a.length);  
}  
  
*/\*\*  
 \* 复制数组方法三：通过for循环来复制数组  
 \*  
 \** ***@param*** *a  
 \** ***@return*** *\*/*public int[] copy3(int[] a) {  
 int[] cpA = new int[a.length];  
 for (int i = 0; i < a.length; i++) {  
 cpA[i] = a[i];  
 }  
 return cpA;  
}

#### 静态数组

增删改查

* 首先创建一个简单的int集合来表示数组，默认大小为10，添加一些常规的方法之后，再对重要的方法进行单独书写。
* add方法：在指定索引位置插入数字，就相当于把指定索引处后的值往后移一位，即把赋值给下一个索引，一层for循环就可以，同时不要忘记size++，然后再在指定索引处赋值。
* remove方法：在指定索引位置删除数字，就相当于把指定索引处后的值往前移一位，即把值赋给上一个索引，一层for循环就可以，同时不要忘记size–。这里有所不同，即int[size]的值需不需要删除，其实这里删不删除并不影响数组的逻辑和功能实现，但本着严谨的态度，将int[size]赋值为null，节省一点内存空间。

public class StaticArray {

*/\*\*  
 \* 定义一个数组  
 \*/* private int[] data;  
 */\*\*  
 \* 定义数据有效长度  
 \*/* private int size;  
  
 */\*\*  
 \* 构造函数，传入数组的容量capacity构造Array  
 \*  
 \** ***@param*** *capacity  
 \*/* public StaticArray(int capacity) {  
 data = new int[capacity];  
 size = 0;  
 }  
  
 */\*\*  
 \* 无参数的构造函数，默认数组的容量capacity=10  
 \*/* public StaticArray() {  
 this(10);  
 }  
  
 */\*\*  
 \* 获取数组中的元素个数  
 \*  
 \** ***@return*** *\*/* public int getSize() {  
 return size;  
 }  
  
 */\*\*  
 \* 获取数组的容量  
 \*  
 \** ***@return*** *\*/* public int getCapacity() {  
 return data.length;  
 }  
  
 */\*\*  
 \* 判断数组是否为空  
 \*  
 \** ***@return*** *\*/* public boolean isEmpty() {  
 return size == 0;  
 }  
  
 */\*\*  
 \* 在第index位置插入一个新元素e  
 \*  
 \** ***@param*** *index  
 \** ***@param*** *e  
 \*/* public void add(int index, int e) {  
 //有效长度等于数组的长度，抛出非法参数异常  
 if (size == data.length) {  
 throw new IllegalArgumentException("Add failed.Array is full.");  
 }  
 //索引小于0或者大于有效长度，抛出数组越界异常  
 if (index < 0 || index > size) {  
 throw new ArrayIndexOutOfBoundsException("Add failed.Require index >=0 is full.");  
 }  
 //复制数组  
 for (int i = size - 1; i >= index; i--) {  
 data[i + 1] = data[i];  
 }  
 //可优化为System.arraycopy(data, index, data, index + 1, size - index);  
 data[index] = e;  
 size++;  
 }  
  
 */\*\*  
 \* 从数组中删除index位置的元素，返回删除的元素  
 \*  
 \** ***@param*** *index  
 \** ***@return*** *\*/* public int remove(int index) {  
 if (index < 0 || index > size) {  
 throw new IllegalArgumentException("Remove failed.Require index is illegal.");  
 }  
 int ret = data[index];  
 for (int i = index + 1; i < size; i++) {  
 data[i - 1] = data[i];  
 }  
 size--;  
 return ret;  
 }  
  
 */\*\*  
 \* 获取index所有位置的元素  
 \*  
 \** ***@param*** *index  
 \** ***@return*** *\*/* public int get(int index) {  
 if (index < 0 || index > size) {  
 throw new IllegalArgumentException("Get failed.Require index is illegal.");  
 }  
 return data[index];  
 }  
  
 */\*\*  
 \* 修改index索引位置的元素为e  
 \*  
 \** ***@param*** *index  
 \** ***@param*** *e  
 \*/* public void set(int index, int e) {  
 if (index < 0 || index > size) {  
 throw new IllegalArgumentException("Set failed.Require index is illegal.");  
 }  
 data[index] = e;  
 }  
  
 */\*\*  
 \* 查找数据中是否有元素e  
 \*  
 \** ***@param*** *e  
 \** ***@return*** *\*/* public boolean contains(int e) {  
 for (int i = 0; i < size; i++) {  
 if (data[i] == e) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 */\*\*  
 \* 查找数组中元素e所在的索引，如果不存在返回-1  
 \*  
 \** ***@param*** *e  
 \** ***@return*** *\*/* public int find(int e) {  
 for (int i = 0; i < size; i++) {  
 if (data[i] == e) {  
 return i;  
 }  
 }  
 return -1;  
 }  
  
 */\*\*  
 \* 向所有元素前添加一个新元素e  
 \*  
 \** ***@param*** *e  
 \*/* public void addFirst(int e) {  
 add(0, e);  
 }  
  
 */\*\*  
 \* 向所有元素后添加一个新元素e  
 \*  
 \** ***@param*** *e  
 \*/* public void addLast(int e) {  
// if (size == data.length) {  
// throw new IllegalArgumentException("AddLast failed.Array is full.");  
// }  
// data[size] = e;  
// size++;  
 add(size, e);  
 }  
  
 */\*\*  
 \* 删除头部元素，返回删除的元素  
 \*  
 \** ***@return*** *\*/* public int removeFirst() {  
 return remove(0);  
 }  
  
 */\*\*  
 \* 删除尾部元素，返回删除的元素  
 \*  
 \** ***@return*** *\*/* public int removeLast() {  
 return remove(size - 1);  
 }  
  
 */\*\*  
 \* 删除元素e  
 \*  
 \** ***@param*** *e  
 \*/* public void removeElement(int e) {  
 int index = find(e);  
 if (index != -1) {  
 remove(index);  
 }  
 }  
}

#### 泛型和动态数组的实现

* 泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？

就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用传入具体的类型。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体现在的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

* 因为不能直接定义泛型，所有可以通过定义Object的方式来强转为泛型
* 定义了泛型之后就可以接收的多种类型，因为泛型把具体的参数泛型化，模糊化，使得传参可以有更多的选择。

动态数组的实现即实现了数组的扩容。扩容的过程是；

* 是否超出了默认的数组长度，超出则执行扩容方法；
* 扩容是新建一个数组，新建数组的长度为原数组的2倍（ArrayList为1.5倍,通过位运算来计算）,接着把原数组的值通过for循环来赋值到新数组，再接着把原数组的引用指向新数组。
* 在增加和删除的时候进行扩容和缩容(新建一个为原数组长度1/2的新数组)
* 在数组缩容的问题上，为了防止复杂度的震荡，数组在长度边缘疯狂的试探，频繁的进行扩容和缩容。（即数组长度为10，增加一个元素扩大为20，之后紧接着又删除了一个元素，有缩容为10）.

所以缩容的时候判断数组的实际长度为1/4时才进行缩容，就可以减少疯狂的试探，从而防止方法的时间复杂度一直为O(n)。