# List

## ArrayList

### 特点：

ArrayList 是基于数组实现的，所以支持快速随机访问。RandomAccess 接口标识着该类支持快速随机访问。

ArrayList 基于数组实现，并且具有动态扩容特性，因此保存元素的数组不一定都会被使用，那么就没必要全部进行序列化。

ArrayList每次扩容请求其大小的1.5 倍。

可以使用 `Collections.synchronizedList();` 得到一个线程安全的 ArrayList。也可以使用 concurrent 并发包下的 CopyOnWriteArrayList 类。

### 类的实现：

public class ArrayList<E> extends AbstractList<E>  
 implements List<E>, RandomAccess, Cloneable, java.io.Serializable

//RandomAccess for循环更快，反之迭代器更快（ArrayList和LinkedList）

### 基本结构：

transient Object[] elementData;//数据(自定义了序列化方式)

private int size;//大小

protected transient int modCount = 0;//改动次数用于避免多线程

### 构造方法：

public ArrayList() {//无参构造空的Object[]  
 this.elementData = *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*;  
}

public ArrayList(int initialCapacity) {//定义初始化大小  
 this.elementData = new Object[initialCapacity];  
}

public ArrayList(Collection<? extends E> c) {//转为ArrayList  
 elementData = c.toArray();  
 if ((size = elementData.length) != 0) {  
 if (elementData.getClass() != Object[].class)  
 elementData = Arrays.*copyOf*(elementData, size, Object[].class);  
 } else {  
 this.elementData = *EMPTY\_ELEMENTDATA*;  
 }  
}

### 其他：

public void ensureCapacity(int minCapacity) {//扩充容量  
 int minExpand = (elementData != *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*)  
 ? 0: *DEFAULT\_CAPACITY*;   
 if (minCapacity > minExpand) {  
 ensureExplicitCapacity(minCapacity);  
 }  
}

public boolean contains(Object o) {//判断包含某元素=查找第一次出现的位置  
 return indexOf(o) >= 0;  
}

public int indexOf(Object o) {//循环找到元素的位置  
 if (o == null) {//允许有null元素  
 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;  
}

public E get(int index) {//得到索引的第几个  
 rangeCheck(index);   
 return elementData(index);  
}

private void rangeCheck(int index) {//检查当前索引是否正确  
 if (index >= size)  
 throw new IndexOutOfBoundsException(outOfBoundsMsg(index));  
}

public boolean retainAll(Collection<?> c) {//清除包含集合c元素的  
 Objects.*requireNonNull*(c);  
 return batchRemove(c, true);//相等的是否清除  
}

## Vector

### 特点：

它的实现与 ArrayList 类似，但是使用了 synchronized 进行同步。

Vector 是同步的，因此开销就比 ArrayList 要大，访问速度更慢。最好使用 ArrayList 而不是 Vector，因为同步操作完全可以由程序员自己来控制；

Vector 每次扩容请求其大小的 2 倍空间，而 ArrayList 是 1.5 倍。

### 类的实现：

public class Vector<E> extends AbstractList<E>  
 implements List<E>, RandomAccess, Cloneable, java.io.Serializable

### 基本结构：

protected Object[] elementData;//数据

protected int elementCount;//集合大小

protected int capacityIncrement;//集合增量大小

### 构造方法：

public Vector() {  
 this(10); //初始化大小  
}

public Vector(int initialCapacity) {  
 this(initialCapacity, 0);  
}

public Vector(int initialCapacity, int capacityIncrement) { //增量大小  
 super();  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException("Illegal Capacity: "+  
 initialCapacity);  
 this.elementData = new Object[initialCapacity];  
 this.capacityIncrement = capacityIncrement;  
}

public Vector(Collection<? extends E> c) {  
 elementData = c.toArray();  
 elementCount = elementData.length;  
 // c.toArray might (incorrectly) not return Object[] (see 6260652)  
 if (elementData.getClass() != Object[].class)  
 elementData = Arrays.*copyOf*(elementData, elementCount, Object[].class);  
}

### 其他

public synchronized void trimToSize() {}//变成真实容量

public synchronized void ensureCapacity(int minCapacity) {}//扩充容量到

## LinkedList

### 特点：

基于双向链表实现，使用Node存储链表节点信息。每个链表存储了 first 和 last 指针。

ArrayList 基于动态数组实现，LinkedList 基于双向链表实现；

ArrayList 支持随机访问，LinkedList 不支持；

LinkedList 在任意位置添加删除元素更快。

### 类的实现：

public class LinkedList<E> extends AbstractSequentialList<E>  
 implements List<E>, Deque<E>, Cloneable, java.io.Serializable

### 基本结构：

transient int size = 0;

transient Node<E> first;

transient Node<E> last;

### 构造方法：

public LinkedList() {  
}

public LinkedList(Collection<? extends E> c) {  
 this();  
 addAll(c);  
}

### 其他：

public E peek() {//得到第一个元素  
 final Node<E> f = first;  
 return (f == null) ? null : f.item;  
}

public E poll() {//删除第一个元素  
 final Node<E> f = first;  
 return (f == null) ? null : unlinkFirst(f);  
}

public boolean offer(E e) {//添加元素  
 return add(e);  
}

public void push(E e) {//队列添加元素  
 addFirst(e);  
}

public E pop() {//队列移除元素  
 return removeFirst();  
}

## CopyOnWriteArrayList

### 特点：

写操作在一个复制的数组上进行，读操作还是在原始数组中进行，读写分离，互不影响。

写操作需要加锁，防止并发写入时导致写入数据丢失。

写操作结束之后需要把原始数组指向新的复制数组。

CopyOnWriteArrayList 在写操作的同时允许读操作，大大提高了读操作的性能，因此很适合读多写少的应用场景。

但是 CopyOnWriteArrayList 有其缺陷：

内存占用：在写操作时需要复制一个新的数组，使得内存占用为原来的两倍左右；

数据不一致：读操作不能读取实时性的数据，因为部分写操作的数据还未同步到读数组中。

所以 CopyOnWriteArrayList 不适合内存敏感以及对实时性要求很高的场景。

### 类的实现：

### 基本结构：

### 构造方法：

### 其他：

# Set

## TreeSet

### 特点：

能够对元素按照某种规则进行排序。同时也保证元素的唯一。

两种排序方式：自然排序、比较器排序。使用元素的自然排序对元素进行排序，还是根据创建集合时提供的Comparator比较器进行排序，这取决于使用的构造方法。

指定比较的规则，需要在自定义类中实现Comparable接口，并重写接口中的compareTo方法。

### 类的实现：

public class TreeSet<E> extends AbstractSet<E>  
 implements NavigableSet<E>, Cloneable, java.io.Serializable

### 基本结构：

private transient NavigableMap<E,Object> m;

private static final Object *PRESENT* = new Object();

### 构造方法：

TreeSet(NavigableMap<E,Object> m) {  
 this.m = m;  
}

public TreeSet() {//自然顺序排序  
 this(new TreeMap<E,Object>());  
}

public TreeSet(Comparator<? super E> comparator) {//按照比较器排序  
 this(new TreeMap<>(comparator));  
}

public TreeSet(Collection<? extends E> c) {//自然顺序排序  
 this();  
 addAll(c);  
}

public TreeSet(SortedSet<E> s) {//指定排序集相同的顺序排序  
 this(s.comparator());  
 addAll(s);  
}

### 其他：

public E ceiling(E e) {}//当前元素的后一个或者是自身

public E floor(E e) {}//当前元素的前一个或者是自身

public E higher(E e) {}//当前元素的后一个

public E lower(E e) {}//当前元素的前一个

public E first() {}//得到第一个元素

public E last() {}//得到最后一个元素

public E pollFirst() {//移出第一个  
 Map.Entry<E,?> e = m.pollFirstEntry();  
 return (e == null) ? null : e.getKey();  
}

public E pollLast() {//移出最后一个  
 Map.Entry<E,?> e = m.pollLastEntry();  
 return (e == null) ? null : e.getKey();  
}

public SortedSet<E> subSet(E fromElement, E toElement) {//范围内  
 return subSet(fromElement, true, toElement, false);  
}

public NavigableSet<E> subSet(E fromElement, boolean fromInclusive,  
 E toElement, boolean toInclusive) {  
 return new TreeSet<>(m.subMap(fromElement, fromInclusive,  
 toElement, toInclusive));  
}

public SortedSet<E> headSet(E toElement) {//当前元素之前  
 return headSet(toElement, false);  
}

public NavigableSet<E> headSet(E toElement, boolean inclusive) {  
 return new TreeSet<>(m.headMap(toElement, inclusive));  
}

public SortedSet<E> tailSet(E fromElement) {//当前元素之后  
 return tailSet(fromElement, true);  
}

public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {  
 return new TreeSet<>(m.tailMap(fromElement, inclusive));  
}

# Queue

# Map

## HashMap

### 特点：

### 类的实现：

### 基本结构：

### 构造方法：

### 其他：

## ConcurrentHashMap

### 特点：

### 类的实现：

### 基本结构：

### 构造方法：

### 其他：

## LinkedHashMap

### 特点：

### 类的实现：

### 基本结构：

### 构造方法：

### 其他：

## WeakHashMap

### 特点：

### 类的实现：

### 基本结构：

### 构造方法：

### 其他：