# 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

## 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));  
}

## HashSet

### 特点：

特点利用hashcode，查询速度，引用了hashmap，所以存的值都是key，统一存了一常量Object。

### 类的实现：

public class HashSet<E> extends AbstractSet<E>  
 implements Set<E>, Cloneable, java.io.Serializable

### 基本结构：

private transient HashMap<E,Object> map;//原形毕露  
private static final Object *PRESENT* = new Object();//static的重要性,只有一份

### 构造方法：

public HashSet() {  
 map = new HashMap<>();  
}

public HashSet(Collection<? extends E> c) {  
 map = new HashMap<>(Math.*max*((int) (c.size()/.75f) + 1, 16));  
 addAll(c);  
}

public HashSet(int initialCapacity, float loadFactor) {

//这俩参数很重要,可以先看看下hashmap,分别是初始容量和扩容标准  
 map = new HashMap<>(initialCapacity, loadFactor);  
}

public HashSet(int initialCapacity) {  
 map = new HashMap<>(initialCapacity);  
}

### 其他：

调用自然都是hashmap的方法

public boolean isEmpty()

public boolean contains(Object o)

public boolean add(E e)

public boolean remove(Object o)

public void clear()

public Object clone()

# Queue

## 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();  
}

# Map

## HashMap

### 特点：

JDK1.8对HashMap底层的实现进行了优化，例如引入红黑树的数据结构和扩容的优化.它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度。(键和值允许都空)

### 类的实现：

public class HashMap<K,V> extends AbstractMap<K,V>  
 implements Map<K,V>, Cloneable, Serializable

### 基本结构：

transient Node<K,V>[] table;

static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 V value;  
 Node<K,V> next;

}

### 构造方法：

public HashMap(int initialCapacity, float loadFactor) {//初始容量，加载因子  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException("Illegal initial capacity: " +  
 initialCapacity);  
 if (initialCapacity > *MAXIMUM\_CAPACITY*)  
 initialCapacity = *MAXIMUM\_CAPACITY*;  
 if (loadFactor <= 0 || Float.*isNaN*(loadFactor))  
 throw new IllegalArgumentException("Illegal load factor: " +  
 loadFactor);  
 this.loadFactor = loadFactor;  
 this.threshold = *tableSizeFor*(initialCapacity);  
}

public HashMap(int initialCapacity) {  
 this(initialCapacity, *DEFAULT\_LOAD\_FACTOR*);  
}

public HashMap() {  
 this.loadFactor = *DEFAULT\_LOAD\_FACTOR*; // all other fields defaulted  
}

public HashMap(Map<? extends K, ? extends V> m) {  
 this.loadFactor = *DEFAULT\_LOAD\_FACTOR*;  
 putMapEntries(m, false);  
}

### 其他：(重要)

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

HashMap.Node<K,V>[] tab; HashMap.Node<K,V> p; int n, i;

// 1.如果table为空或者长度为0，即没有元素，那么使用resize()方法扩容

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

// 2.计算插入存储的数组索引i，此处计算方法同 1.7 中的indexFor()方法

// 如果数组为空，即不存在Hash冲突，则直接插入数组

if ((p = tab[i = (n - 1) & hash]) == null)

tab[i] = newNode(hash, key, value, null);

// 3.插入时，如果发生Hash冲突，则依次往下判断

else {

HashMap.Node<K,V> e; K k;

// a.判断table[i]的元素的key是否与需要插入的key一样，若相同则直接用新的value覆盖掉旧的value

// 判断原则equals() - 所以需要当key的对象重写该方法

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

e = p;

// b.继续判断：需要插入的数据结构是红黑树还是链表

// 如果是红黑树，则直接在树中插入 or 更新键值对

else if (p instanceof HashMap.TreeNode)

e = ((HashMap.TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

// 如果是链表，则在链表中插入 or 更新键值对

else {

// i .遍历table[i]，判断key是否已存在：采用equals对比当前遍历结点的key与需要插入数据的key

// 如果存在相同的，则直接覆盖

// ii.遍历完毕后任务发现上述情况，则直接在链表尾部插入数据

// 插入完成后判断链表长度是否 > 8：若是，则把链表转换成红黑树

for (int binCount = 0; ; ++binCount) {

if ((e = p.next) == null) {

p.next = newNode(hash, key, value, null);

if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

break;

}

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

break;

p = e;

}

}

// 对于i 情况的后续操作：发现key已存在，直接用新value覆盖旧value&返回旧value

if (e != null) { // existing mapping for key

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null)

e.value = value;

afterNodeAccess(e);

return oldValue;

}

}

++modCount;

// 插入成功后，判断实际存在的键值对数量size > 最大容量

// 如果大于则进行扩容

if (++size > threshold)

resize();

// 插入成功时会调用的方法（默认实现为空）

afterNodeInsertion(evict);

return null;

}

/\*\*

\* 该函数有2中使用情况：1.初始化哈希表；2.当前数组容量过小，需要扩容

\*/

final Node<K,V>[] resize() {

Node<K,V>[] oldTab = table;// 扩容前的数组（当前数组）

int oldCap = (oldTab == null) ? 0 : oldTab.length;// 扩容前的数组容量（数组长度）

int oldThr = threshold;// 扩容前数组的阈值

int newCap, newThr = 0;

if (oldCap > 0) {

// 针对情况2：若扩容前的数组容量超过最大值，则不再扩容

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

}

// 针对情况2：若没有超过最大值，就扩容为原来的2倍（左移1位）

else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

}

// 针对情况1：初始化哈希表（采用指定或者使用默认值的方式）

else if (oldThr > 0) // initial capacity was placed in threshold

newCap = oldThr;

else { // zero initial threshold signifies using defaults

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

}

// 计算新的resize上限

if (newThr == 0) {

float ft = (float)newCap \* loadFactor;

newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?

(int)ft : Integer.MAX\_VALUE);

}

threshold = newThr;

@SuppressWarnings({"rawtypes","unchecked"})

Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];

table = newTab;

if (oldTab != null) {

// 把每一个bucket都移动到新的bucket中去

for (int j = 0; j < oldCap; ++j) {

Node<K,V> e;

if ((e = oldTab[j]) != null) {

oldTab[j] = null;

if (e.next == null)

newTab[e.hash & (newCap - 1)] = e;

else if (e instanceof TreeNode)

((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

else { // preserve order

Node<K,V> loHead = null, loTail = null;

Node<K,V> hiHead = null, hiTail = null;

Node<K,V> next;

do {

next = e.next;

if ((e.hash & oldCap) == 0) {

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

}

else {

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

} while ((e = next) != null);

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead;

}

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;

}

}

}

}

}

return newTab;

}

## ConcurrentHashMap

### 特点：

ConcurrentHashMap 结合了 HashMap 和 HashTable 二者的优势。HashMap 没有考虑同步，HashTable 考虑了同步的问题。但是 HashTable 在每次同步执行时都要锁住整个结构。 ConcurrentHashMap 锁的方式是稍微细粒度的。(键和值不允许空)

### 类的实现：

public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>  
 implements ConcurrentMap<K,V>, Serializable

### 基本结构：

transient Node<K,V>[] table;

static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 V value;  
 Node<K,V> next;

}

### 构造方法：

(和hashmap一样的用法,这里介绍和hashmap,这里介绍几个)

### 其他：

## HashTabale

### 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：

## TreeMap

### 特点：

### 类的实现：

### 基本结构：

### 构造方法：

### 其他：