基于Jdk1.7

## ArrayList

### 成员变量

|  |
| --- |
| private static final int DEFAULT\_CAPACITY = 10;//默认容量大小  private static final Object[] EMPTY\_ELEMENTDATA = {};//构造空的数组  private transient Object[] elementData;//ArrayList中的数据  private int size;//包含元素的个数,而不是数组的大小 |

说明：

|  |
| --- |
| 保存数据的elementData为Object[],即底层数据结构是数组；使用了transient修饰此变量，即序列化的时候不会进行序列化此值。 |

### 构造方法

|  |
| --- |
| public ArrayList(int initialCapacity) {  super();  if (initialCapacity < 0)  throw new IllegalArgumentException("Illegal Capacity: "+  initialCapacity);  this.elementData = new Object[initialCapacity];  }  public ArrayList() {  super();  this.elementData = EMPTY\_ELEMENTDATA;//默认空数组,之前版本的jkd中是默认10个大小的  }  public ArrayList(Collection<? extends E> c) {  elementData = c.toArray();  size = elementData.length;  // c.toArray might (incorrectly) not return Object[] (see 6260652)  if (elementData.getClass() != Object[].class)  elementData = Arrays.copyOf(elementData, size, Object[].class);  } |

#### 关于Arrays.copyOf();

|  |
| --- |
| //todo |

### 重要方法：

#### Add(E e);

//在尾部添加一个元素。

|  |
| --- |
| public boolean add(E e) {  ensureCapacityInternal(size + 1); // Increments modCount!!  elementData[size++] = e;  return true;  }  Increments modCount!! //对modCount的值进行修改；  想数组中添加元素，为了保证数组能容纳的下，先确保数组的大小：  private void ensureCapacityInternal(int minCapacity) {  if (elementData == EMPTY\_ELEMENTDATA) {  minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);  }  ensureExplicitCapacity(minCapacity);//确保明确的数组大小  }  private void ensureExplicitCapacity(int minCapacity) {  modCount++;//修改次数+1  // overflow-conscious code  if (minCapacity - elementData.length > 0)//如果添加元素后的大小>数组的大小  grow(minCapacity);//扩容  }  private void grow(int minCapacity) {  // overflow-conscious code  int oldCapacity = elementData.length;  int newCapacity = oldCapacity + (oldCapacity >> 1);//原容量1.5倍  if (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  if (newCapacity - MAX\_ARRAY\_SIZE > 0)// MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;//如果新的容量特别的大,获得int的最大值与数组最大值的最大值  newCapacity = hugeCapacity(minCapacity);  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.copyOf(elementData, newCapacity);  }  private static int hugeCapacity(int minCapacity) {  if (minCapacity < 0) // overflow  throw new OutOfMemoryError();  return (minCapacity > MAX\_ARRAY\_SIZE) ?  Integer.MAX\_VALUE :  MAX\_ARRAY\_SIZE;  } |

#### AddAll(int index,Collection c);

//在指定位置添加插入集合

|  |
| --- |
| public boolean addAll(int index, Collection<? extends E> c) {  rangeCheckForAdd(index);//检查插入的位置是否合法  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;  } |

#### lastIndexOf(Object o);

|  |
| --- |
| 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;  } |

#### Remove(index);

|  |
| --- |
| 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  return oldValue;  }  private void rangeCheck(int index) {  if (index >= size)  throw new IndexOutOfBoundsException(outOfBoundsMsg(index));  } |

#### Iterator();

返回集合的迭代器

|  |
| --- |
| public Iterator<E> iterator() {  return new Itr();  }  private class Itr implements Iterator<E> {  int cursor; // 下个元素的索引位置  int lastRet = -1; // index of last element returned; -1 if no such  //lastRet上一次游标所在的位置，总是比cursor少1  int expectedModCount = modCount;//使得这2个值相等，就不会报’并发修改异常了’  public boolean hasNext() {  return cursor != size;  }  @SuppressWarnings("unchecked")  public E next() {  checkForComodification();  int i = cursor;  if (i >= size)  throw new NoSuchElementException();  Object[] elementData = ArrayList.this.elementData;  if (i >= elementData.length)  throw new ConcurrentModificationException();  cursor = i + 1;//下个位置+1  return (E) elementData[lastRet = i];//取下个位置的数据，elementData是ArrayList的成员变量  }  public void remove() {  if (lastRet < 0)//肯定是在调用完netx()后才有可能调到remove()  throw new IllegalStateException();  checkForComodification();  try {  ArrayList.this.remove(lastRet);//仍然使用的是ArrayList本身的remove，不过已经强制的把expectedModCount = modCount;  cursor = lastRet;  lastRet = -1;  expectedModCount = modCount;  } catch (IndexOutOfBoundsException ex) {  throw new ConcurrentModificationException();  }  }  final void checkForComodification() {  if (modCount != expectedModCount)  throw new ConcurrentModificationException();  }  }  说明：  由源代码我们可知，虽然iterator强制的使expectedModCount = modCount;但多线程环境下，仍然会出问题。 |

### ArrayList整体总结：

|  |
| --- |
| 底层基于Object[]存储数据  数组的优点：(1)随机访问效率高(根据下标查询)，(2)搜索效率较高(可使用折半方法)。数组的缺点：(1)内存连续且固定，存储效率低。(2)插入和删除效率低(可能会进行数组拷贝或扩容)。  线程不安全。  在增加元素时，不停的调整数组的大小。  维护着一个int变量modCout,当集合的数据发生变化时，都会修改此值，使得Itr类中modCount != expectedModCount,因此当在迭代时[增强for也是迭代] ，如果集合发生了变化，都会引起ConcurrentModificationException.  单线程环境下，使用迭代器自带的remove()删除，不会发生异常  多线程环境下，使用CopyOnWriteArrayList代替ArrayList和Vector;或者对迭代器进行同步。  参见：<http://www.cnblogs.com/dolphin0520/p/3933551.html>  http://www.cnblogs.com/hzmark/archive/2012/12/20/ArrayList.html |

## LinkedList

### 成员变量

|  |
| --- |
| transient int size = 0;//元素的个数  transient Node<E> first;//指向第一个节点  transient Node<E> last;//指向最后一个节点  其中Node的数据定义为：  private static class Node<E> {  E item;//节点的元素  Node<E> next;//前一个节点  Node<E> prev;//后一个节点  Node(Node<E> prev, E element, Node<E> next) {// 前，元素，后  this.item = element;  this.next = next;  this.prev = prev;  }  }  说明：  基于双向链表实现 |

### 重要方法

#### Add(E e);

|  |
| --- |
| public boolean add(E e) {  linkLast(e);//说明是向链表的最后插入  return true;  }  void **linkLast**(E e) {  final Node<E> l = last;//暂存之前的最后一个节点  final Node<E> newNode = new Node<>(l, e, null);//构造新的节点  last = newNode;//last执行新的节点  if (l == null)//之前链表为null  first = newNode;//头节点指向新节点  else  l.next = newNode;//之前节点的下个节点指向新的节点  size++;  modCount++;  } |

#### Add(index,E);

|  |
| --- |
| public void add(int index, E element) {  checkPositionIndex(index);//检测插入的位置是否合法  if (index == size)//尾部插入  linkLast(element);  else  linkBefore(element, node(index));  }  void **linkBefore**(E e, Node<E> succ) {//在某个节点前插入  // assert succ != null;  final Node<E> pred = succ.prev;  final Node<E> newNode = new Node<>(pred, e, succ);  succ.prev = newNode;  if (pred == null)  first = newNode;  else  pred.next = newNode;  size++;  modCount++;  } |

#### Remove(obj)

|  |
| --- |
| public boolean remove(Object o) {  if (o == null) {  for (Node<E> x = first; x != null; x = x.next) {  if (x.item == null) {  unlink(x);  return true;  }  }  } else {  for (Node<E> x = first; x != null; x = x.next) {  if (o.equals(x.item)) {  unlink(x);  return true;  }  }  }  return false;  }  关键在于: unlink()  E **unlink**(Node<E> x) {  // assert x != null;  final E element = x.item;  final Node<E> next = x.next;//得到前一个节点  final Node<E> prev = x.prev;//得到后一个节点  if (prev == null) {  first = next;  } else {  prev.next = next;//前一个节点的next指向后一个节点  x.prev = null;//当前节点的prev置空  }  if (next == null) {  last = prev;  } else {  next.prev = prev;//后一个节点的prev指向前一个节点  x.next = null;//当前节点的next置空  }  x.item = null;//将此节点数据置空  size--;  modCount++;  return element;  }  从双向列表上删除一个节点的顺序为：  1取得此节点的前一个节点prev、后一个节点next  2prev的next指向next节点，当前节点的prev置空  3next节点的prev指向prev节点，当前节点的next置空  4当前节点数据置空 |

#### indexOf(obj);

其实就是从头到尾遍历

|  |
| --- |
| public int indexOf(Object o) {  int index = 0;  if (o == null) {  for (Node<E> x = first; x != null; x = x.next) {  if (x.item == null)  return index;  index++;  }  } else {  for (Node<E> x = first; x != null; x = x.next) {  if (o.equals(x.item))  return index;  index++;  }  }  return -1;  } |

#### listIterator(index);

每个list接口类都有的方法，实现对list的迭代

|  |
| --- |
| public ListIterator<E> listIterator(int index) {  checkPositionIndex(index);  return new ListItr(index);  }  private class ListItr implements ListIterator<E> {  private Node<E> lastReturned = null;//最后返回的节点,看remove()  private Node<E> next;  private int nextIndex;  private int expectedModCount = modCount;  ListItr(int index) {  // assert isPositionIndex(index);  next = (index == size) ? null : node(index);  nextIndex = index;  }  public boolean hasNext() {  return nextIndex < size;  }  public E next() {  checkForComodification();  if (!hasNext())  throw new NoSuchElementException();  lastReturned = next;  next = next.next;  nextIndex++;  return lastReturned.item;  }  public boolean hasPrevious() {  return nextIndex > 0;  }  public E previous() {  checkForComodification();  if (!hasPrevious())  throw new NoSuchElementException();  lastReturned = next = (next == null) ? last : next.prev;  nextIndex--;  return lastReturned.item;  }  public void remove() {//删除刚刚迭代的节点  checkForComodification();  if (lastReturned == null)//没有进行迭代就删除，报错  throw new IllegalStateException();  Node<E> lastNext = lastReturned.next;  unlink(lastReturned);//使用通用的unlink()删除节点  if (next == lastReturned)  next = lastNext;  else  nextIndex--;  lastReturned = null;  expectedModCount++;  }    public void add(E e) {  checkForComodification();  lastReturned = null;  if (next == null)  linkLast(e);  else  linkBefore(e, next);  nextIndex++;  expectedModCount++;  }  final void checkForComodification() {  if (modCount != expectedModCount)  throw new ConcurrentModificationException();  }  } |

#### Get(index)

|  |
| --- |
| 链表是不支持随机访问的，它是如何通过索引来获取元素的呢？  public E get(int index) {  checkElementIndex(index);  return node(index).item;  }  Node<E> node(int index) {//某个索引位置的node  // assert isElementIndex(index);  if (index < (size >> 1)) {//在前一半进行搜索  Node<E> x = first;//初始时是头节点  for (int i = 0; i < index; i++)  x = x.next;//使下个元素作为临时头节点  return x;  } else {//后一半进行搜索  Node<E> x = last;  for (int i = size - 1; i > index; i--)  x = x.prev;  return x;  }  } |

### LinkedList整体总结：

|  |  |  |
| --- | --- | --- |
| 基于双向链表实现，增加、删除操作效率很高。  链表的优点：(1)不要求连续内存，内存利用率高，(2)插入和删除效率高(只需要改变指针指向)。链表的缺点：(1)不支持随机访问，(2)搜索效率低(需要遍历)。  线程不安全。  可以依靠其提供的方法实现堆栈、队列、双端队列操作。  堆栈方法(push()压栈 pop()弹栈 peek()只取栈顶的数据不弹栈)  队列方法:addLast(e) offerLast(e) removeFirst() pollFirst()  每个节点的数据类型为：   |  | | --- | | private static class Node<E> {  E item;//元素  Node<E> next;//后  Node<E> prev;//前  Node(Node<E> prev, E element, Node<E> next) {//前，元素，后  this.item = element;  this.next = next;  this.prev = prev;  }  } |   对数据结构的操作方法为：   |  | | --- | | void **linkLast**(E e) {//在尾部插入  final Node<E> l = last;//暂存之前的最后一个节点  final Node<E> newNode = new Node<>(l, e, null);//构造新的节点  last = newNode;//last执行新的节点  if (l == null)//之前链表为null  first = newNode;//头节点指向新节点  else  l.next = newNode;//之前节点的下个节点指向新的节点  size++;  modCount++;  }  void **linkBefore**(E e, Node<E> succ) {//在某个节点前插入  // assert succ != null;  final Node<E> pred = succ.prev;  final Node<E> newNode = new Node<>(pred, e, succ);  succ.prev = newNode;  if (pred == null)  first = newNode;  else  pred.next = newNode;  size++;  modCount++;  }  private void **linkFirst**(E e) {//在头部插入  final Node<E> f = first;  final Node<E> newNode = new Node<>(null, e, f);  first = newNode;  if (f == null)  last = newNode;  else  f.prev = newNode;  size++;  modCount++;  }  E **unlink**(Node<E> x) {//删除某个节点  // assert x != null;  final E element = x.item;  final Node<E> next = x.next;//得到前一个节点  final Node<E> prev = x.prev;//得到后一个节点  if (prev == null) {  first = next;  } else {  prev.next = next;//前一个节点的next指向后一个节点  x.prev = null;//当前节点的prev置空  }  if (next == null) {  last = prev;  } else {  next.prev = prev;//后一个节点的prev指向前一个节点  x.next = null;//当前节点的next置空  }  x.item = null;//将此节点数据置空  size--;  modCount++;  return element;  }  private E unlinkFirst(Node<E> f) {//删除头节点  // assert f == first && f != null;  final E element = f.item;  final Node<E> next = f.next;  f.item = null;  f.next = null; // help GC  first = next;  if (next == null)  last = null;  else  next.prev = null;  size--;  modCount++;  return element;  }  private E unlinkLast(Node<E> l) {//删除尾节点  // assert l == last && l != null;  final E element = l.item;  final Node<E> prev = l.prev;  l.item = null;  l.prev = null; // help GC  last = prev;  if (prev == null)  first = null;  else  prev.next = null;  size--;  modCount++;  return element;  } | |

## HashMap

### 重要的成员变量

|  |
| --- |
| static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4;//默认容量16  static final int MAXIMUM\_CAPACITY = 1 << 30;//对打容量  static final float DEFAULT\_LOAD\_FACTOR = 0.75f;//负载因子  static final Entry<?,?>[] EMPTY\_TABLE = {};//空的Entry[],说明其底层结构有数组  transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY\_TABLE;//用于存储键值对的  int threshold;// 实际能容纳的容量capacity \* load factor，size超过此数会进行再hash  总结：  Hasmap中的元素是用名称为table的Entry[]保存的，默认大小是16 |

### 存储结构

|  |
| --- |
| 说明：  Entry是一个链表结构，不仅报刊key,value,还包含指向下一个节点的next  图中的每一行都是一个**桶(bucket)**  static class Entry<K,V> implements Map.Entry<K,V> {  final K key;  V value;  Entry<K,V> next;//指向下一个Entry  int hash;  /\*\*  \* Creates new entry.  \*/  Entry(int h, K k, V v, Entry<K,V> n) {  value = v;  next = n;  key = k;  hash = h;  }  ………………. } |

### 重要方法

#### Put(k,v);

|  |
| --- |
| public V put(K key, V value) {  if (table == EMPTY\_TABLE) {  inflateTable(threshold);  }  if (key == null)  return putForNullKey(value);//对null key的处理  int hash = hash(key);//得到key的hash值  int i = indexFor(hash, table.length);//此hash值在table中的索引位置  for (Entry<K,V> e = table[i]; e != null; e = e.next) {//遍历此索引位置的链表  Object k;  if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  V oldValue = e.value;  e.value = value;  e.recordAccess(this);  return oldValue;  }  }  modCount++;  addEntry(hash, key, value, i);  return null;  }  private V putForNullKey(V value) {  //null key 只存在于table[0]上  for (Entry<K,V> e = table[0]; e != null; e = e.next) {  if (e.key == null) {  V oldValue = e.value;  e.value = value;  e.recordAccess(this);  return oldValue;  }  }  modCount++;  addEntry(0, null, value, 0);//添加Entry对  return null;  }  void addEntry(int hash, K key, V value, int bucketIndex) {  if ((size >= threshold) && (null != table[bucketIndex])) {  resize(2 \* table.length);//对数组进行扩容2倍,包括调整桶的位置等  hash = (null != key) ? hash(key) : 0;  bucketIndex = indexFor(hash, table.length);//计算桶索引  }  createEntry(hash, key, value, bucketIndex);//将新加入的元素放到首位  }  void createEntry(int hash, K key, V value, int bucketIndex) {  Entry<K,V> e = table[bucketIndex];  table[bucketIndex] = new Entry<>(hash, key, value, e);  size++;  }  总结来讲：  Null key 时，会将其放在table[0]的位置上。  先对key进行hash取值，string类型的key单独对待取hash值；再根据hash得到索引位置，然后遍历此索引上的各个链表节点，将值放到第一个的位置上。  期间会经历resize等 |

#### Get(key);

|  |
| --- |
| public V get(Object key) {  if (key == null)  return getForNullKey();//在table[0]上取值  Entry<K,V> entry = getEntry(key);//通过key获取到Entry  return null == entry ? null : entry.getValue();  }  final Entry<K,V> getEntry(Object key) {  if (size == 0) {  return null;  }  int hash = (key == null) ? 0 : hash(key);//取得key的hash  for (Entry<K,V> e = table[indexFor(hash, table.length)];  e != null;  e = e.next) {//在这个桶上找数据  Object k;  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k))))  return e;  }  return null;  } |

### 整体总结：

|  |
| --- |
| HashMap底层基于Entry[]和链表，默认大小16，负载因子0.75.  Hash表的优点：(1)搜索效率高，(2)插入和删除效率较高，Hash表的缺点：(1)内存利用率低(基于数组)，(2)存在散列冲突。  允许null key null value  线程不安全。  也会发生并发修改异常，也有迭代器。  取key的hash值，然后确定在数组中的位置，然后遍历操作。  2个对象的equals返回false，但其hashcode有可能一样，但最好要规避这种现象，所以一个好的hash算法很关键.  HashMap存值的时候，遵循的是LRU（Least Recently Used）算法，就是最近最少使用，意思就是系统认为最近使用的值可能会被系统经常用到，而添加的早的值用到的几率会小。  参见：https://segmentfault.com/a/1190000003617333 |

### HashSet

#### 理解

|  |
| --- |
| private transient HashMap<E,Object> map;//基于map存储  构造函数：  public HashSet() {  map = new HashMap<>();  }  Add方法：  public boolean add(E e) {  return map.put(e, PRESENT)==null;  } |

总结

|  |
| --- |
| 基于hashMap实现，利用了HashMap 的key不重复的特点。  无序，不重复，允许空  线程不安全 |

## TreeMap

|  |
| --- |
| http://www.cnblogs.com/tstd/p/5081237.html |

## Iterator迭代器模式

|  |
| --- |
| 实现了Collection接口的类，都实现了Iterable接口，比如ArrayList,linkedList,Vector,hashSet等，实现了迭代的功能。 |

### Iterable接口

|  |
| --- |
| public interface Iterable<T> {  Iterator<T> iterator();  }  //实现了此接口的类，均可使用foreach.  public interface Iterator<E> {  boolean hasNext();  E next();  void remove();  }  //有了上述这3个方法，我们就可以进行迭代了。  参考：http://www.cnblogs.com/tstd/p/5049338.html |

## Comparator比较器

|  |
| --- |
| 此接口有个比较的方法：  int compare(T o1, T o2);  实现此接口需实现此方法。 |

### 常见用法

|  |
| --- |
| System.out.println(list);//[3, 1, 5, 10, 7]  Collections.sort(list, new Comparator<Integer>() {  @Override  public int compare(Integer o1, Integer o2) {  // return o1.compareTo(o2);//升序 [1, 3, 5, 7, 10]  // return o2.compareTo(o1);//降序 [10, 7, 5, 3, 1]  //这里，compareTo是Integer中实现的Comparable接口的方法  if(o1<o2){  return -1;  }else if(o1>02){  return 1;  }else if(o1 == o2){  return 0;  }  return 0;  }  //Collections.sort(list) 如果不指定比较器，则是升序操作，底层使用Arrys.sort(),二分排序  //默认情况下，如果o1>02 则返回 1 ，是升序；如果非要自己定义规则：o1<02 返回1，那就是降序了  }); |

关于升序降序的理解

|  |
| --- |
| 简单理解：  默认情况下：如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数，这样是默认的升序操作；如果要使之为降序，则把规则反过来即可。 |