# 概览

容器主要包含两种：Collection和Map。Collection存储对象的集合，Map存储键值对的映射。

## Collection



### Set

* TreeSet：基于红黑树实现，支持有序性操作，例如根据一个范围查找元素的操作。查找效率不如HashSet，HashSet查找时间复杂度O(1)，TreeSet查找时间复杂度O(logN)。
* HashSet：基于哈希表实现，支持快速查找，但是不支持有序性操作。并且失去了元素的插入顺序信息，也就是说Iterator遍历HashSet得到的结果是不确定的。
* LinkedHashSet：具有HashSet的查找效率，并且内部使用双向链表维护插入的顺序。

### List

* ArrayList：基于动态数组实现，支持随机访问。
* Vector：和ArrayList类似，但是是线程安全的。
* LinkedList：基于双向链表实现，只能顺序访问，但是可以快速地在链表中插入和删除元素。不仅如此，LinkedList还可以作栈、队列和双向队列。

### Queue

* LinkedList：可以用来实现双向队列；
* PriorityQueue：基于堆结构实现，可以用来实现优先队列。

## Map



* TreeMap：基于红黑树实现。
* HashMap：基于哈希表实现。
* HashTable：与HashMap类似，但是是线程安全的。这意味着同一时刻多个线程可以同时写入HashTable并且不会导致数据不一致，它是遗留类，不应该去使用它。现在可以使用ConcurrentHashMap来支持线程安全，ConcurrentHashMap效率更高，因为引入了分段锁。
* LinkedHashMap：使用双向链表维护元素的顺序，顺序为插入顺序或者最近最少使用顺序（LRU）。

# 容器中的设计模式

## 迭代器模式



Collection继承了Iterable接口，通过其中的Iterator()方法产生一个Iterator对象，通过对象迭代遍历Collection中的元素。

Java1.5之后可以使用foreach方法来遍历实现了Iterator接口的对象。

|  |
| --- |
| List<String> list = new ArrayList<>();  list.add("a");  list.add("b");  for (String item : list) {  System.out.println(item);  } |

## 适配器模式

Java.util.Arrays#asList()可以把数组类型转化为List类型。

|  |
| --- |
| @SafeVarargs  public static <T> List<T> asList(T... a) |

应该注意的是 asList() 的参数为泛型的变长参数，不能使用基本类型数组作为参数，只能使用相应的包装类型数组。例如：

|  |
| --- |
| Integer[] arr = {1, 2, 3};  List list = Arrays.asList(arr); |

也可以使用以下方式调用 asList()：

|  |
| --- |
| List list = Arrays.asList(1, 2, 3); |

# 源码分析

如果没有特殊说明，以下源码分析基于Java1.8。

## ArrayList

### 概览

因为 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。

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

数组的默认大小为 10。

|  |
| --- |
| private static final int DEFAULT\_CAPACITY = 10; |

### 扩容

添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，如果不够时，需要使用 grow() 方法进行扩容，新容量的大小为 oldCapacity + (oldCapacity >> 1)，也就是旧容量的 1.5 倍。

扩容操作需要调用 Arrays.copyOf() 把原数组整个复制到新数组中，这个操作代价很高，因此最好在创建 ArrayList 对象时就指定大概的容量大小，减少扩容操作的次数。

|  |
| --- |
| public boolean add(E e) {  ensureCapacityInternal(size + 1); // Increments modCount!!  elementData[size++] = e;  return true;  }  private void ensureCapacityInternal(int minCapacity) {  if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {  minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);  }  ensureExplicitCapacity(minCapacity);  }  private void ensureExplicitCapacity(int minCapacity) {  modCount++;  // overflow-conscious code  if (minCapacity - elementData.length > 0)  grow(minCapacity);  }  private void grow(int minCapacity) {  // overflow-conscious code  int oldCapacity = elementData.length;  int newCapacity = oldCapacity + (oldCapacity >> 1);  if (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  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);  } |

### 删除元素

需要调用 System.arraycopy() 将 index+1 后面的元素都复制到 index 位置上，该操作的时间复杂度为 O(N)，可以看出 ArrayList 删除元素的代价是非常高的。

|  |
| --- |
| 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;  } |

### Fail-Fast

modCount 用来记录 ArrayList 结构发生变化的次数。结构发生变化是指添加或者删除至少一个元素的所有操作，或者是调整内部数组的大小，仅仅只是设置元素的值不算结构发生变化。

在进行序列化或者迭代等操作时，需要比较操作前后 modCount 是否改变，如果改变了需要抛出 ConcurrentModificationException。

|  |
| --- |
| private void writeObject(java.io.ObjectOutputStream s)  throws java.io.IOException{  // Write out element count, and any hidden stuff  int expectedModCount = modCount;  s.defaultWriteObject();  // Write out size as capacity for behavioural compatibility with clone()  s.writeInt(size);  // Write out all elements in the proper order.  for (int i=0; i<size; i++) {  s.writeObject(elementData[i]);  }  if (modCount != expectedModCount) {  throw new ConcurrentModificationException();  }  } |

### 序列化

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。

保存元素的数组 elementData 使用 transient 修饰，该关键字声明数组默认不会被序列化。

|  |
| --- |
| transient Object[] elementData; // non-private to simplify nested class access |

ArrayList 实现了 writeObject() 和 readObject() 来控制只序列化数组中有元素填充那部分内容。

|  |
| --- |
| private void readObject(java.io.ObjectInputStream s)  throws java.io.IOException, ClassNotFoundException {  elementData = EMPTY\_ELEMENTDATA;  // Read in size, and any hidden stuff  s.defaultReadObject();  // Read in capacity  s.readInt(); // ignored  if (size > 0) {  // be like clone(), allocate array based upon size not capacity  ensureCapacityInternal(size);  Object[] a = elementData;  // Read in all elements in the proper order.  for (int i=0; i<size; i++) {  a[i] = s.readObject();  }  }  } |

|  |
| --- |
| private void writeObject(java.io.ObjectOutputStream s)  throws java.io.IOException{  // Write out element count, and any hidden stuff  int expectedModCount = modCount;  s.defaultWriteObject();  // Write out size as capacity for behavioural compatibility with clone()  s.writeInt(size);  // Write out all elements in the proper order.  for (int i=0; i<size; i++) {  s.writeObject(elementData[i]);  }  if (modCount != expectedModCount) {  throw new ConcurrentModificationException();  }  } |

序列化时需要使用 ObjectOutputStream 的 writeObject() 将对象转换为字节流并输出。而 writeObject() 方法在传入的对象存在 writeObject() 的时候会去反射调用该对象的 writeObject() 来实现序列化。反序列化使用的是 ObjectInputStream 的 readObject() 方法，原理类似。

## Vector

### 同步

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

|  |
| --- |
| public synchronized boolean add(E e) {  modCount++;  ensureCapacityHelper(elementCount + 1);  elementData[elementCount++] = e;  return true;  }  public synchronized E get(int index) {  if (index >= elementCount)  throw new ArrayIndexOutOfBoundsException(index);  return elementData(index);  } |

### 与ArrayList比较

* Vector是同步的，因此开销就会比ArrayList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控制。
* Vector每次扩容请求其大小2倍的空间，而ArrayList为1.5倍。

### 替代方案

可以使用 Collections.synchronizedList(); 得到一个线程安全的 ArrayList。

|  |
| --- |
| List<String> list = new ArrayList<>();  List<String> synList = Collections.synchronizedList(list); |

也可以使用 concurrent 并发包下的 CopyOnWriteArrayList 类。

|  |
| --- |
| List<String> list = new CopyOnWriteArrayList<>(); |

## CopyOnWriteArrayList

### 读写分离

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。

|  |
| --- |
| public boolean add(E e) {  final ReentrantLock lock = this.lock;  lock.lock();  try {  Object[] elements = getArray();  int len = elements.length;  Object[] newElements = Arrays.copyOf(elements, len + 1);  newElements[len] = e;  setArray(newElements);  return true;  } finally {  lock.unlock();  }  }  final void setArray(Object[] a) {  array = a;  } |

|  |
| --- |
| @SuppressWarnings("unchecked")  private E get(Object[] a, int index) {  return (E) a[index];  } |

### 适用场景

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。

### 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。

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