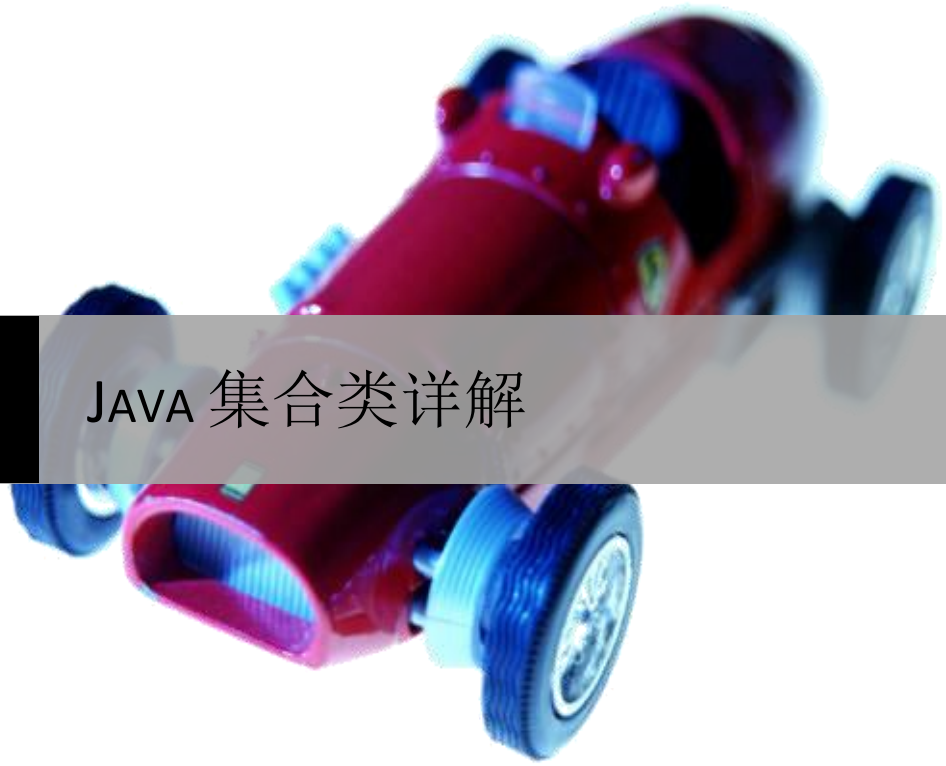
2012/6/19



[键入公司名称]

JAVA 集合类详解

Elvis 整理

集合框架

1.1 容器简介

到目前为止，我们已经学习了如何创建多个不同的对象，定义了这些对象以后，我们就可以利用它们来做一些有意义的事情。

举例来说，假设要存储许多雇员，不同的雇员的区别仅在于雇员的身份证号。我们可以通过身份证号来顺序存储每个雇员，但是在内存中实现呢？是不是要准备足够的内存来存储 1000 个雇员，然后再将这些雇员逐一插入？如果已经插入了 500 条记录，这时需要插入一个身份证号较低的新雇员，该怎么办呢？是在内存中将 500 条记录全部下移后，再从开头插入新的记录？还是创建一个映射来记住每个对象的位置？当决定如何存储对象的集合时，必须考虑如下问题。

对于对象集合，必须执行的操作主要以下三种：

- 添加新的对象
- 删除对象
- 查找对象

我们必须确定如何将新的对象添加到集合中。可以将对象添加到集合的末尾、开头或者中间的某个逻辑位置。

从集合中删除一个对象后，对象集合中现有对象会有什么影响呢？可能必须将内存移来移去，或者就在现有对象所驻留的内存位置下一个“洞”。

在内存中建立对象集合后，必须确定如何定位特定对象。可建立一种机制，利用该机制可根据某些搜索条件（例如身份证号）直接定位到目标对象；否则，便需要遍历集合中的每个对象，直到找到要查找的对象为止。

前面大家已经学习过了数组。数组的作用是可以存取一组数据。但是它却存在一些缺点，使得无法使用它来比较方便快捷的完成上述应用场景的要求。

1. 首先，在很多数情况下面，我们需要能够存储一组数据的容器，这一点虽然数组可以实现，但是如果我们需要存储的数据的个数多少并不确定。比如说：我们需要在容器里面存储某

个应用系统的当前的所有的在线用户信息，而当前的在线用户信息是时刻都可能在变化的。 也就是说，我们需要一种存储数据的容器，它能够自动的改变这个容器的所能存放的数据数量的大小。这一点上，如果使用数组来存储的话，就显得十分的笨拙。

2. 我们再假设这样一种场景：假定一个购物网站，经过一段时间的运行，我们已经存储了一系列的购物清单了，购物清单中有商品信息。如果我们想要知道这段时间里面有多少种商品被销售出去了。那么我们就需要一个容器能够自动的过滤掉购物清单中的关于商品的重复信息。如果使用数组，这也是很难实现的。

3. 最后再想想，我们经常会遇到这种情况，我知道某个人的帐号名称，希望能够进一步了解这个人的其他的一些信息。也就是说，我们在一个地方存放一些用户信息，我们希望能够通过用户的帐号来查找到对应的该用户的其他的一些信息。再举个查字典例子：假设我们希望使用一个容器来存放单词以及对于这个单词的解释，而当我们想要查找某个单词的意思的时候，能够根据提供的单词在这个容器中找到对应的单词的解释。如果使用数组来实现的话，就更加的困难了。

为解决这些问题，Java 里面就设计了容器集合，不同的容器集合以不同的格式保存对象。

数学背景

在常见用法中，集合（collection）和数学上直观的集（set）的概念是相同的。集是一个唯一项组，也就是说组中没有重复项。实际上，"集合框架"包含了一个 Set 接口和许多具体的 Set 类。但正式的集概念却比 Java 技术提前了一个世纪，那时英国数学家 George Boole 按逻辑正式的定义了集的概念。大部分人在小学时通过我们熟悉的维恩图引入的"集的交"和"集的并"学到过一些集的理论。

集的基本属性如下：

- 集内只包含每项的一个实例
- 集可以是有限的，也可以是无限的
- 可以定义抽象概念

集不仅是逻辑学、数学和计算机科学的基础，对于商业和系统的日常应用来说，它也很实用。"连接池"这一概念就是数据库服务器的一个开放连接集。Web 服务器必须管理客户机和连接集。文件描述符提供了操作系统中另一个集的示例。

映射是一种特别的集。它是一种对（pair）集，每个对表示一个元素到另一元素的单向映射。一些映射示例有：

- IP 地址到域名（DNS）的映射
- 关键字到数据库记录的映射
- 字典（词到含义的映射）
- 2 进制到 10 进制转换的映射

就像集一样，映射背后的思想比 Java 编程语言早的多，甚至比计算机科学还早。而 Java 中的 Map 就是映射的一种表现形式。

1.2 容器的分类

既然您已经具备了一些集的理论，您应该能够更轻松的理解"集合框架"。"集合框架"由一组用来操作对象的接口组成。不同接口描述不同类型的组。在很大程度上，一旦您理解了接口，您就理解了框架。虽然您总要创建接口特定的实现，但访问实际集合的方法应该限制在接口方法的使用上；因此，允许您更改基本的数据结构而不必改变其它代码。

Java 容器类类库的用途是“保存对象”，并将其划分为两个不同的概念：

1） Collection 。一组对立的元素，通常这些元素都服从某种规则。List 必须保持元素特定的顺序，而 Set 不能有重复元素。

2） Map 。一组成对的“键值对”对象。初看起来这似乎应该是一个 Collection ，其元素是成对的对象，但是这样的设计实现起

来太笨拙了，于是我们将 Map 明确的提取出来形成一个独立的概念。另一方面，如果使用 Collection 表示 Map 的部分内容，会便于查看此部分内容。因此 Map 一样容易扩展成多维 Map ，无需增加新的概念，只要让 Map 中的键值对的每个“值”也是一个 Map 即可。

Collection 和 Map 的区别在于容器中每个位置保存的元素个数。Collection 每个位置只能保存一个元素（对象）。此类容器包括：

List ，它以特定的顺序保存一组元素；

Set 则是元素不能重复。

Map 保存的是“键值对”，就像一个小型数据库。我们可以通过“键”找到该键对应的“值”。

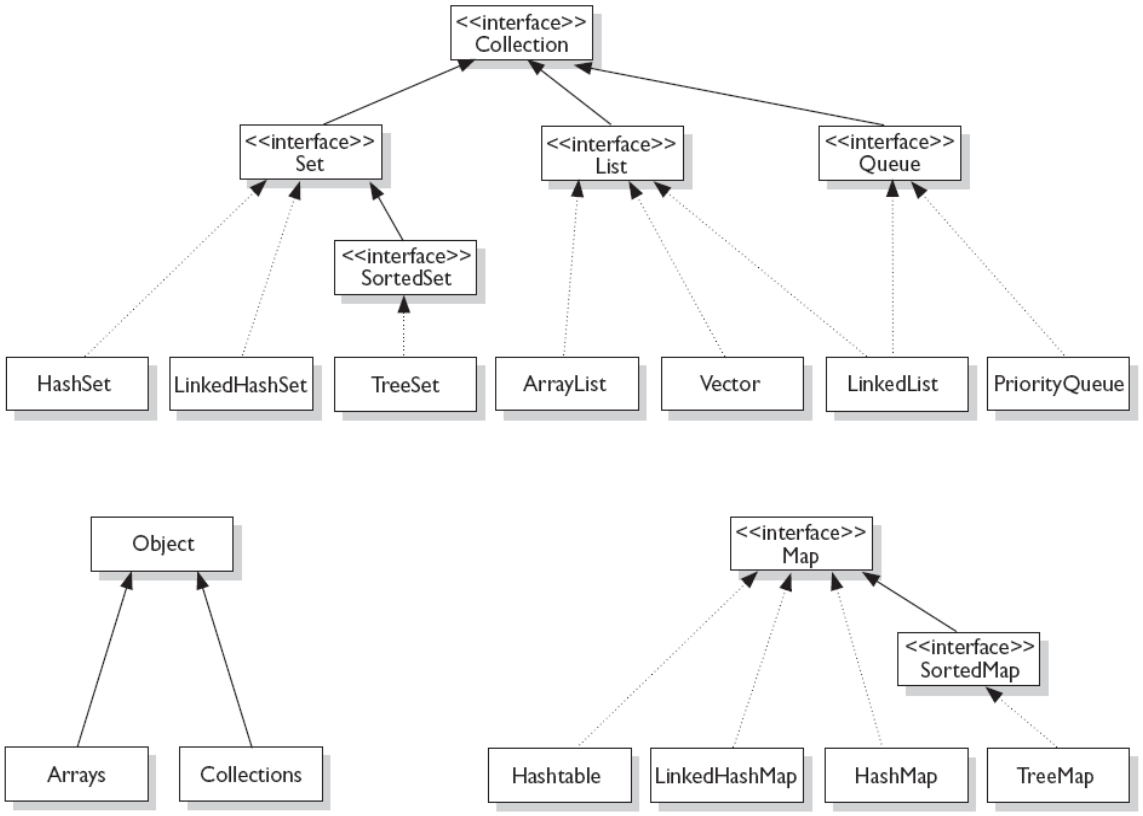
- Set －对象之间没有指定的顺序，不允许重复元素
- List－对象之间有指定的顺序，允许重复元素,并引入位置下标。
- Map－接口用于保存关键字（Key）和数值（Value）的集合，集合中的每个对象加入时都提供数值和关键字。

List、Set、Map 共同的实现基础是 Object 数组

除了四个历史集合类外，Java 2 框架还引入了六个集合实现，如下表所示。

接口	实现	历史集合类
Set	HashSet	
	TreeSet	
List	ArrayList	Vector
	LinkedList	Stack
Map	HashMap	Hashtable
	TreeMap	Properties

集合关系框架图：



Collection

1.2.1 常用方法

Collection 接口用于表示任何对象或元素组。想要尽可能以常规方式处理一组元素时，就使用这一接口。Collection 在前面的大图也可以看出，它是 List 和 Set 的父类。并且它本身也是一个接口。它定义了作为集合所应该拥有的一些方法。如下：

注意：

集合必须只有对象，集合中的元素不能是基本数据类型。

Collection 接口支持如添加和除去等基本操作。设法除去一个元素时，如果这个元素存在，除去的仅仅是集合中此元素的一个实例。

- boolean add(Object element)
- boolean remove(Object element)

Collection 接口还支持查询操作：

- int size()
- boolean isEmpty()
- boolean contains(Object element)

- Iterator iterator()

组操作 ：Collection 接口支持的其它操作，要么是作用于元素组的任务，要么是同时作用于整个集合的任务。

- boolean containsAll(Collection collection)
- boolean addAll(Collection collection)
- void clear()
- void removeAll(Collection collection)
- void retainAll(Collection collection)

containsAll() 方法允许您查找当前集合是否包含了另一个集合的所有元素，即另一个集合是否是当前集合的子集。其余方法是可选的，因为特定的集合可能不支持集合更改。

addAll() 方法确保另一个集合中的所有元素都被添加到当前的集合中，通常称为并。 clear() 方法从当前集合中除去所有元素。 removeAll() 方法类似于 clear() ，但只除去了元素的一个子集。

retainAll() 方法类似于 removeAll() 方法，不过可能感到它所做的与前面正好相反：它从当前集合中除去不属于另一个集合的元素，即交。

我们看一个简单的例子，来了解一下集合类的基本方法的使用：

```
package com.test;
import java.util.ArrayList;
import java.util.Collection;
public class CollectionToArray {
    public static void main(String[] args) {
        Collection collection1=new ArrayList();//创建一个集合对象
        collection1.add("000");//添加对象到Collection集合中
        collection1.add("111");
        collection1.add("222");
        System.out.println("集合collection1的大小：
                           "+collection1.size());
        System.out.println("集合collection1的内容："+collection1);
        collection1.remove("000");//从集合collection1中移除掉
                                  // "000" 这个对象
        System.out.println("集合collection1移除 000 后的内容：
                             "+collection1);
        System.out.println("集合collection1中是否包含000 ：
                          "+collection1.contains("000"));
        System.out.println("集合collection1中是否包含111 ：
                           "+collection1.contains("111"));
```

```
        Collection collection2=new ArrayList();
        //将collection1 集合中的元素全部都加到collection2中

collection2.addAll(collection1);
        System.out.println("集合collection2的内容："+collection2);
        collection2.clear();//清空集合 collection2 中的元素
        System.out.println("集合collection2是否为空 ：
                           "+collection2.isEmpty());
        //将集合collection1转化为数组
        Object s[]= collection1.toArray();
        for(int i=0;i<s.length;i++){
           System.out.println(s[i]);
        }
    }
}
```

运行结果如下：

```
集合collection1的大小：3
集合collection1的内容：[000, 111, 222]
集合collection1移除 000 后的内容：[111, 222]
集合collection1中是否包含000 ：false
集合collection1中是否包含111 ：true
集合collection2的内容：[111, 222]
集合collection2是否为空 ：true
111
222
```

这里需要注意的是，Collection 它仅仅只是一个接口，而我们真正使用的时候，确是创建该接口的一个实现类。做为集合的接口，它定义了所有属于集合的类所都应该具有的一些方法。

而 ArrayList （列表）类是集合类的一种实现方式。

这里需要一提的是，因为 Collection 的实现基础是数组，所以有转换为 Object 数组的方法：

- Object[] toArray()
- Object[] toArray(Object[] a)

其中第二个方法 Object[] toArray(Object[] a) 的参数 a 应该是集合中所有存放的对象的类的父类。

1.2.2 迭代器

任何容器类，都必须有某种方式可以将东西放进去，然后由某种方式将东西取出来。毕竟，存放事物是容器最基本的工作。对于 ArrayList 中 add()是插入对象的方法，而 get()是取出元素的方式之一。ArrayList 很灵活，可以随时选取任意的元素，或使用不同的下标一次选取多个元素。

如果从更高层的角度思考，会发现这里有一个缺点：要使用容器，必须知道其中元素的确切类型。初看起来这没有什么不好的，但是考虑如下情况：如果原本是 ArrayList ，但是后来考虑到容器的特点，你想换用 Set ，应该怎么做？或者你打算写通用的代码，它们只是使用容器，不知道或者说不关心容器的类型，那么如何才能不重写代码就可以应用于不同类型的容器？

所以迭代器(Iterator)的概念，也是出于一种设计模式就是为达成此目的而形成的。所以 Collection 不提供 get()方法。如果要遍历 Collection 中的元素，就必须用 Iterator。

迭代器（Iterator）本身就是一个对象，它的工作就是遍历并选择集合序列中的对象，而客户端的程序员不必知道或关心该序列底层的结构。此外，迭代器通常被称为“轻量级”对象，创建它的代价小。但是，它也有一些限制，例如，某些迭代器只能单向移动。

Collection 接口的 iterator() 方法返回一个 Iterator。Iterator 和您可能已经熟悉的 Enumeration 接口类似。使用 Iterator 接口方法，您可以从头至尾遍历集合，并安全的从底层 Collection 中除去元素。

下面，我们看一个对于迭代器的简单使用：

```
package com.test;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class IteratorDemo {
   public static void main(String[] args) {
      Collection collection = new ArrayList();
      for (int i = 0; i < 5; i++) {
         collection.add("collection"+i);
      }
      Iterator iterator = collection.iterator();//得到一个迭代器
      while (iterator.hasNext()) {//遍历集合，检查是否存在下一个
```

```
            Object element = iterator.next();
            System.out.println("iterator = " + element);
        }
        if(collection.isEmpty()){
            System.out.println("collection is Empty!");
        }else{
            System.out.println("collection is not Empty!
                                size="+collection.size());
        }
        Iterator iterator2 = collection.iterator();
        while (iterator2.hasNext()) {//移除元素
            Object element = iterator2.next();
            System.out.println("remove之前: "+element);
            iterator2.remove();
        }
        Iterator iterator3 = collection.iterator();
        if (!iterator3.hasNext()) {//察看是否还有元素
            System.out.println("已经没有元素了");
        }
        if(collection.isEmpty())
            System.out.println("collection is Empty!");
        //使用collection.isEmpty()方法来判断
    }
}
```

运行结果如下：

```
iterator = collection0
iterator = collection1
iterator = collection2
iterator = collection3
iterator = collection4
collection is not Empty! size=5
remove之前: collection0
remove之前: collection1
remove之前: collection2
remove之前: collection3
remove之前: collection4
已经没有元素了
collection is Empty!
```

可以出：

1) 使用方法 iterator() 要求容器返回一个 Iterator .第一次调用 Iterator 的 next() 方法时，它返回集合序列的第一个元素。

2) 使用 next() 获得集合序列的中的下一个元素。

3) 使用 hasNext()检查序列中是否元素。

4) 使用 remove()将迭代器新返回的元素删除。

需要注意的是：方法删除由 next 方法返回的最后一个元素，在每次调用 next 时，remove 方法只能被调用一次 。

Java 实现的这个迭代器的使用就是如此的简单。Iterator（跌代器）虽然功能简单，但仍然可以帮助我们解决许多问题，同时针对 List 还有一个更复杂更高级的 ListIterator。

List

1.1 概述

前面我们讲述的 Collection 接口实际上并没有直接的实现类。而 List 是容器的一种，表示列表的意思。当我们不知道存储的数据有多少的情况，我们就可以使用 List 来完成存储数据的工作。例如我们想要在保存一个应用系统当前的在线用户的信息。我们就可以使用一个 List 来存储。因为 List 的最大的特点就是能够自动的根据插入的数据量来动态改变容器的大小。下面我们先看看 List 接口的一些常用方法。

1.2 常用方法

List 就是列表的意思，它是 Collection 的一种，即继承了 Collection 接口，以定义一个允许重复项的有序集合。该接口不但能够对列表的一部分进行处理，还添加了面向位置的操作。List 是按对象的进入顺序进行保存对象，而不做排序或编辑操作。它除了拥有父类 Collection 接口的所有的方法外还定义了一些其他的方法。

面向位置的操作包括插入某个元素或 Collection 的功能，还包括获取、除去或更改元素的功能。在 List 中搜索元素可以从列表的头部或尾部开始，如果找到元素，还将报告元素所在的位置。

- void add(int index, Object element) ：添加对象 element 到位置 index 上
- boolean addAll(int index, Collection collection) ：在 index

位置后添
加容器 collection
中
所有的元素

- Object get(int index) ：取出下标为 index 的位置的元素
- int indexOf(Object element) ：查找对象 element 在 List 中第一次出
现的位置
- int lastIndexOf(Object element) ：查找对象 element 在 List 中最后
出现的位置
- Object remove(int index) ：删除 index 位置上的元素
- Object set(int index, Object element) ：将 index 位置上的对象替换为
element 并返回老的元素。

在"集合框架"中有两种常规的 List 实现：ArrayList 和 LinkedList。使用两种 List 实现的哪一种取决于您特定的需要。如果要支持随机访问，而不必在除尾部的任何位置插入或除去元素，那么，ArrayList 提供了可选的集合。但如果，您要频繁的从列表的中间位置添加和除去元素，而只要顺序的访问列表元素，那么，LinkedList 实现更好。

以 ArrayList 为例，我们可以把 12 个月份存放到 ArrayList 中，然后用一个循环，并使用 get()方法将列表中的对象都取出来。

而 LinkedList 添加了一些处理列表两端元素的方法，使用这些方法，您就可以轻松的把 LinkedList 当作一个堆栈、队列或其它面向端点的数据结构。

如下例子：

```
package com.test;
import java.util.LinkedList;
public class LinkedListDemo {
   public static void main(String[] args) {
       LinkedList queue = new LinkedList();
          queue.addFirst("Head1");
          queue.addFirst("Head2");
```

```
            queue.addFirst("Head3");
            queue.addFirst("Head4");
            queue.addLast("Head5");
            queue.addFirst("Head6");
            System.out.println(queue);
           //移除最后一个
            queue.removeLast();
            //移除第一个
            queue.removeFirst();
            System.out.println(queue);
    }
}
```

运行程序产生了以下输出。

```
[Head6, Head4, Head3, Head2, Head1, Head5]
[Head4, Head3, Head2, Head1]
```

1.3 实现原理

前面已经提了一下 Collection 的实现基础都是基于数组的。下面我们就已 ArrayList 为例，简单分析一下 ArrayList 列表的实现方式。首先，先看下它的构造函数。

下列表格是在 SUN 提供的 API 中的描述：

ArrayList()	构造一个初始容量为 10 的空列表。
ArrayList(Collection c)	构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。
ArrayList(int initialCapacity)	构造一个具有指定初始容量的空列表。

其中第一个构造函数 **ArrayList**()和第二构造函数 **ArrayList**(Collection c) 是按照 Collection 接口文档所述，所应该提供两个构造函数，一个无参数，一个接受另一个 Collection。

第 3 个构造函数：

ArrayList(int initialCapacity) 是 ArrayList 实现的比较重要的构造函数，虽然，我们不常用它，但是某认的构造函数正是调用的该带参数：initialCapacity 的构造函数来实现的。 其中参数：initialCapacity 表示我们构造的这个 ArrayList 列表的初始化容量是多大。如果调用默认的构造函数，则表示默认调用该参数为 initialCapacity =10 的方式,来进行构建一个 ArrayList 列表对象。

为了更好的理解这个 initialCapacity 参数的概念，我们先看看 ArrayList 在 Sun 提供的源码中的实现方式。先看一下它的属性有哪些：

ArrayList 继承了 AbstractList 我们主要看看 ArrayList 中的属性就可以了。

ArrayList 中主要包含 2 个属性：

- private transient Object elementData[];
- private int size;

其中数组:: elementData[] 是列表的实现核心属性：数组。 我们使用该数组来进行存放集合中的数据。而我们的初始化参数就是该数组构建时候的长度，即该数组的 length 属性就是 initialCapacity 参数。

Keys：**transient** 表示被修饰的属性不是对象持久状态的一部分，不会自动的序列化。

第 2 个属性：size 表示列表中真实数据的存放个数。

我们再来看一下 ArrayList 的构造函数,加深一下 ArrayList 是基于数组的理解。

从源码中可以看到默认的构造函数调用的就是带参数的构造函数：

```
public ArrayList(int initialCapacity)
```

不过参数 initialCapacity＝10 。

我们主要看 ArrayList(int initialCapacity) 这个构造函数。可以看到：

```
this.elementData = new Object[initialCapacity];
```

我们就是使用的 **initialCapacity** 这个参数来创建一个 Object 数组。而我们所有的往该集合对象中存放的数据，就是存放到了这个 Object 数组中去了。

我们在看看另外一个构造函数的源码：

这里，我们先看 size() 方法的实现形式。它的作用即是返回 size 属性值的大小。然后我们再看另外一个构造函数 public ArrayList(Collection c) ，该构造函数的作用是把另外一个容器对象中的元素存放到当前的 List 对象中。

可以看到，首先，我们是通过调用另外一个容器对象 c 的方法 size()来设置当前的 List 对象的 size 属性的长度大小。

接下来，就是对 **elementData** 数组进行初始化，初始化的大小为原先容器大小的 1.1 倍。最后，就是通过使用容器接口中的 Object[] toArray(Object[] a) 方法来把当前容器中的对象都存放到新的数组 **elementData** 中。这样就完成了一个 ArrayList 的建立。

可能大家会存在一个问题，那就是，我们建立的这个 ArrayList 是使用数组来实现的，但是数组的长度一旦被定下来，就不能改变了。而我们在给 ArrayList 对象中添加元素的时候，却没有长度限制。这

个时候，ArrayList 中的 **elementData** 属性就必须存在一个需要动态的扩充容量的机制。我们看下面的代码，它描述了这个扩充机制：

```
  public void ensureCapacity(int minCapacity) {
   modCount++;//父类中的属性记录集合变化次数
   int oldCapacity = elementData.length;
   if (minCapacity > oldCapacity) {
       Object oldData[] = elementData;
       int newCapacity = (oldCapacity * 3)/2 + 1;
          if (newCapacity < minCapacity)
       newCapacity = minCapacity;
        elementData = Arrays.copyOf(elementData, newCapacity);
   }
}
```

这个方法的作用就是用来判断当前的数组是否需要扩容，应该扩容多少。其中属性：modCount 是继承自父类，它表示当前的对象对 **elementData** 数组进行了多少次扩容，清空，移除等操作。该属性相当于是一个对于当前 List 对象的一个操作记录日志号。

1. 首先得到当前 **elementData** 属性的长度 oldCapacity。
2. 然后通过判断 oldCapacity 和 minCapacity 参数谁大来决定是否需要扩容
   - 如果 minCapacity 大于 oldCapacity，那么我们就对当前的 List 对象进行扩容。扩容的的策略为：取(oldCapacity * 3)/2 + 1 和 minCapacity 之间更大的那个。然后使用数组拷贝的方法，把以前存放的数据转移到新的数组对象中
   - 如果 minCapacity 不大于 oldCapacity 那么就不进行扩容。

下面我们看看 ensureCapacity 方法的是如何使用的：

```
public boolean add(E e) {
   ensureCapacity(size + 1);  // Increments modCount!!
   elementData[size++] = e;
```

```
    return true;
    }
    public void add(int index, E element) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(
        "Index: "+index+", Size: "+size);

    ensureCapacity(size+1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
            size - index);
    elementData[index] = element;
    size++;
    }
```

这两个 add()都是往 List 中添加元素。每次在添加元素的时候，我们就需要判断一下，是否需要对于当前的数组进行扩容。

我们主要看看 **public boolean add(Object o)**方法，可以发现在添加一个元素到容器中的时候，首先我们会判断是否需要扩容。因为只增加一个元素，所以扩容的大小判断也就为当前的 size+1 来进行判断。然后，就把新添加的元素放到数组 **elementData** 中。

方法 **public boolean addAll(Collection c)**也是同样的原理。将新的元素放到 **elementData** 数组之后。同时改变当前 List 对象的 size 属性。

类似的 List 中的其他的方法也都是基于数组进行操作的。大家有兴趣可以看看源码中的更多的实现方式。

最后我们再看看如何判断在集合中是否已经存在某一个对象的：

```
public boolean contains(Object o) {
    return indexOf(o) >= 0;
    }
```

由源码中我们可以看到，public boolean contains(Object elem)方法是通过调用 public int indexOf(Object elem)方法来判断是否在集合中存在某个对象 elem。我们看看 indexOf 方法的具体实现。

```
public int indexOf(Object o) {
   if (o == null) {
       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;
       }
```

- 首先我们判断一下 elem 对象是否为 **null** ，如果为 null 的话，那么遍历数组 **elementData** 把第一个出现 null 的位置返回。
- 如果 elem 不为 **null** 的话，我们也是遍历数组 **elementData** ，并通过调用 elem 对象的 equals()方法来得到第一个相等的元素的位置。

这里我们可以发现，ArrayList 中用来判断是否包含一个对象，调用的是各个对象自己实现的 equals()方法。在前面的高级特性里面，我们可以知道：如果要判断一个类的一个实例对象是否等于另外一个对象，那么我们就需要自己覆写 Object 类的 public boolean equals(Object obj) 方法。如果不覆写该方法的话，那么就会调用 Object 的 equals()方法来进行判断。这就相当于比较两个对象的内存应用地址是否相等了。

在集合框架中，不仅仅是 List，所有的集合类，如果需要判断里面是否存放了的某个对象，都是调用该对象的 equals()方法来进行处理的。

Map

1.1 概述

数学中的映射关系在 Java 中就是通过 Map 来实现的。它表示，里面存储的元素是一个对（pair）,我们通过一个对象，可以在这个映射关系中找到另外一个和这个对象相关的东西。

前面提到的我们对于根据帐号名得到对应的人员的信息，就属于这种情况的应用。我们讲一个人员的帐户名和这人员的信息作了一个映射关系，也就是说，我们把帐户名和人员信息当成了一个“键值对”,“键”就是帐户名,“值”就是人员信息。下面我们先看看 Map 接口的常用方法。

1.2 常用方法

Map 接口不是 Collection 接口的继承。而是从自己的用于维护键-值关联的接口层次结构入手。按定义，该接口描述了从不重复的键到值的映射。

我们可以把这个接口方法分成三组操作：改变、查询和提供可选视图。

改变操作允许您从映射中添加和除去键-值对。键和值都可以为 null。但是，您不能把 Map 作为一个键或值添加给自身。

- Object put(Object key,Object value)：

  用来存放一个键-值对 Map 中

- Object remove(Object key)：

  根据 key(键)，移除一个键-值对，并将值返回

- void putAll(Map mapping) ：

  将另外一个 Map 中的元素存入当前的 Map 中

◆ void clear() ：清空当前 Map 中的元素

查询操作允许您检查映射内容：

◆ Object get(Object key) ：根据 key(键)取得对应的值

◆ boolean containsKey(Object key) ：

判断 Map 中是否存在某键（key）

◆ boolean containsValue(Object value)：

判断 Map 中是否存在某值(value)

◆int size()：返回 Map 中 键-值对的个数

◆ boolean isEmpty() ：判断当前 Map 是否为空

最后一组方法允许您把键或值的组作为集合来处理。

◆public Set keySet() ：返回所有的键（key），并使用 Set 容器存放

◆ public Collection values() ：返回所有的值（Value），并使用 Collection 存放

◆ public Set entrySet() ：返回一个实现 Map.Entry 接口的元素 Set

因为映射中键的集合必须是唯一的，就使用 Set 来支持。因为映射中值的集合可能不唯一，就使用 Collection 来支持。最后一个方法返回一个实现 Map.Entry 接口的元素 Set。

我们看看 Map 的常用实现类的比较：

Map ：保存键值对成员，基于键找值操作，使用 compareTo 或 compare 方法对键进行排序

HashMap ：能满足用户对 Map 的通用需求,成员要求：键成员可为任意 Object 子类的对象，但如果覆盖了 equals 方法，同时注意修改 hashCode 方法。

TreeMap: 支持对键有序地遍历,使用时建议先用 HashMap 增加和删除成员，最后从 HashMap 生成 TreeMap； 附加实现了 SortedMap 接口，支持子 Map 等要求顺序的操作，成员要求：键成员要求实现 `Comparable` 接口，或者使用 Comparator 构造 TreeMap 键成员一般为同一类型。

下面我们看一个简单的例子：

```
package com.test;
import java.util.HashMap;
import java.util.Map;
public class MapTest {
public static void main(String[] args) {
      Map map1 = new HashMap();
      Map map2 = new HashMap();
      map1.put("1","a1");
      map1.put("2","b1");
      map2.put("10","a2");
      map2.put("11","b2");
//根据键 "1" 取得值：  "a1"
      System.out.println("map1.get(\"1\")="+map1.get("1"));
// 根据键 "1" 移除键值对"1"-"a1"
   System.out.println("map1.remove(\"1\")="+map1.remove("1"));
      System.out.println("map1.get(\"1\")="+map1.get("1"));
      map1.putAll(map2);//将map2全部元素放入map1中
      map2.clear();//清空map2
      System.out.println("map1 IsEmpty?="+map1.isEmpty());
      System.out.println("map2 IsEmpty?="+map2.isEmpty());
      System.out.println("map1 中的键值对的个数size =
                          "+map1.size());
      System.out.println("KeySet="+map1.keySet());//set
      System.out.println("values="+map1.values());//Collection
      System.out.println("entrySet="+map1.entrySet());
      System.out.println("map1 是否包含键：11 =
                            "+map1.containsKey("11"));
      System.out.println("map1 是否包含值：a1 =
                            "+map1.containsValue("a1"));
   }
   }
```

运行输出结果为：

```
      map1.get("1")=a1
      map1.remove("1")=a1
```

```
map1.get("1")=null
map1 IsEmpty?=false
map2 IsEmpty?=true
map1 中的键值对的个数size = 3
KeySet=[2, 10, 11]
values=[b1, a2, b2]
entrySet=[2=b1, 10=a2, 11=b2]
map1 是否包含键：11 = true
map1 是否包含值：a1 = false
```

在该例子中，我们创建一个 HashMap，并使用了一下 Map 接口中的各个方法。

其中 Map 中的 **entrySet**()方法先提一下，该方法返回一个实现 Map.Entry 接口的对象集合。集合中每个对象都是底层 Map 中一个特定的键-值对。

Map.Entry 接口是 Map 接口中的一个内部接口，该内部接口的实现类存放的是键值对。在下面的实现原理中，我们会对这方面再作介绍，现在我们先不管这个它的具体实现。

我们再看看排序的 Map 是如何使用：

```
package com.test;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Random;
import java.util.TreeMap;
public class MapSortDemo {
     public static void main(String args[]) {
          Map map1 = new HashMap();
          Map map2 = new LinkedHashMap();
          Random random =new Random();
          for(int i=1;i<=5;i++){
             int  s=random.nextInt(10)+1;
//产生一个随机数，并将其放入Map中
              map1.put(s,"第 "+i+" 个放入的元素："+s+"\n");
              map2.put(s,"第 "+i+" 个放入的元素："+s+"\n");
          }
          System.out.println("未排序前HashMap："+map1);
          System.out.println("未排序前LinkedHashMap："+map2);
          //使用TreeMap来对另外的Map进行重构和排序
```

```
            Map sortedMap = new TreeMap(map1);
            System.out.println("排序后: "+sortedMap);
            System.out.println("排序后: "+new TreeMap(map2));
        }
}
```

该程序的一次运行结果为：

```
未排序前HashMap: {1=第 5 个放入的元素: 1
, 6=第 4 个放入的元素: 6
, 7=第 3 个放入的元素: 7
, 9=第 1 个放入的元素: 9
}
未排序前LinkedHashMap: {9=第 1 个放入的元素: 9
, 6=第 4 个放入的元素: 6
, 7=第 3 个放入的元素: 7
, 1=第 5 个放入的元素: 1
}
排序后: {1=第 5 个放入的元素: 1
, 6=第 4 个放入的元素: 6
, 7=第 3 个放入的元素: 7
, 9=第 1 个放入的元素: 9
}
排序后: {1=第 5 个放入的元素: 1
, 6=第 4 个放入的元素: 6
, 7=第 3 个放入的元素: 7
, 9=第 1 个放入的元素: 9
}
```

从运行结果，我们可以看出，HashMap 的存入顺序和输出顺序无关。而 LinkedHashMap 则保留了键值对的存入顺序。TreeMap 则是对 Map 中的元素进行排序。在实际的使用中我们也经常这样做：使用 HashMap 或者 LinkedHashMap 来存放元素，当所有的元素都存放完成后，如果使用则是需要一个经过排序的 Map 的话，我们再使用 TreeMap 来重构原来的 Map 对象。这样做的好处是：因为 HashMap 和 LinkedHashMap 存储数据的速度比直接使用 TreeMap 要快，存取效率要高。当完成了所有的元素的存放后，我们再对整个的 Map 中的元素进行排序。这样可以提高整个程序的运行的效率，缩短执行时间。

这里需要注意的是，TreeMap 中是根据键（Key）进行排序的。而如果我们要使用 TreeMap 来进行正常的排序的话，Key 中存放的对象必须实现 **Comparable** 接口。

我们简单介绍一下这个接口：

1．3　　Comparable 接口

在 java.lang 包中，此接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的*自然排序*，类的 compareTo 方法被称为它的*自然比较方法*。

它只有一个方法：int compareTo(T o)；比较此对象与指定对象的顺序。

用来比较当前实例和作为参数传入的元素。如果排序过程中当前实例出现在参数前（当前实例比参数大），就返回某个负值。如果当前实例出现在参数后（当前实例比参数小），则返回正值。否则，返回零。如果这里不要求零返回值表示元素相等。零返回值可以只是表示两个对象在排序的时候排在同一个位置。

int 的包装类：Integer 就实现了该接口。我们可以看一下这个类的源码：

```
public int compareTo(Integer anotherInteger) {
    int thisVal = this.value;
    int anotherVal = anotherInteger.value;
    return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 :
1));

        }
```

可以看到 compareTo 方法里面通过判断当前的 Integer 对象的值是否大于传入的参数的值来得到返回值的。

在 Java 2 SDK，版本 1.2 中有十四个类实现 Comparable 接口。下表展示了它们的自然排序。虽然一些类共享同一种自然排序，但只有相互可比的类才能排序。

类	排序
BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short	按数字大小排序
Character	按 Unicode 值的数字大小排序
CollationKey	按语言环境敏感的字符串排序
Date	按年代排序
File	按系统特定的路径名的全限定字符的 Unicode 值排序
ObjectStreamField	按名字中字符的 Unicode 值排序
String	按字符串中字符 Unicode 值排序

1.4 Map 实现原理

有的人可能会认为 Map 会继承 Collection。在数学中，映射只是对（pair）的集合。但是，在“集合框架”中，接口 Map 和 Collection 在层次结构没有任何亲缘关系，它们是截然不同的。这种差别的原因与 Set 和 Map 在 Java 库中使用的方法有关。Map 的典型应用是访问按关键字存储的值。它支持一系列集合操作的全部，但操作的是键-值对，而不是单个独立的元素。因此 Map 需要支持 get() 和 put() 的基本操作，而 Set 不需要。此外，还有返回 Map 对象的 Set 视图的方法：

```
Set set = aMap.keySet();
```

下面我们以 HashMap 为例，对 Map 的实现机制作一下更加深入一点的理解。

因为 HashMap 里面使用 Hash 算法，所以在理解 HashMap 之前，我们需要先了解一下 Hash 算法和 Hash 表。

Hash，一般翻译做“散列”，也有直接音译为"哈希"的，就是把任意长度的输入（又叫做 预映射， pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能 会散列成相同的输出，而不可能从散列值来唯一的确定输入值。

说的通俗一点，Hash 算法的意义在于提供了一种快速存取数据的方法,它用一种算法建立键值与真实值之间的对应关系,(每一个真实值只能有一个键值,但是一个键值可以对应多个真实值),这样可以快速在数组等里面存取数据。

我们建立一个 HashTable（哈希表），该表的长度为 N，然后我们分别在该表中的格子中存放不同的元素。每个格子下面存放的元素又是以链表的方式存放元素。

- 当添加一个新的元素 Entry 的时候，首先我们通过一个 Hash 函数计算出这个 Entry 元素的 Hash 值 hashcode。通过该 hashcode 值，就可以直接定位出我们应该把这个 Entry 元素存入到 Hash 表的哪个格子中，如果该格子中已经存在元素了，那么只要把新的 Entry 元存放到这个链表中即可。
- 如果要查找一个元素 Entry 的时候，也同样的方式，通过 Hash 函数计算出这个 Entry 元素的 Hash 值 hashcode。然后通过该 hashcode 值，就可以直接找到这个 Entry 是存放到哪个格子中的。接下来就对该格子存放的链表元素进行逐个的比较查找就可以了。

举一个比较简单的例子来说明这个算法的运算方式：

假定我们有一个长度为 8 的 Hash 表（可以理解为一个长度为 8 的数组）。在这个 Hash 表中存放数字：如下表

0	1	2	3	4	5	6	7

假定我们的 Hash 函数为：

Hashcode = X%8 -------- 对 8 取余数

其中 X 就是我们需要放入 Hash 表中的数字，而这个函数返回的 Hashcode 就是 Hash 码。

假定我们有下面 10 个数字需要依次存入到这个 Hash 表中：

11 ， 23 ， 44 ， 9 ， 6 ， 32 ， 12 ， 45 ， 57 ， 89

通过上面的 Hash 函数，我们可以得到分别对应的 Hash 码：

11－－3 ； 23－－7 ； 44－－4 ； 9－－1； 6－－6； 32－－0； 12－－4； 45－－5； 57－－1； 89－－1；

计算出来的 Hash 码分别代表，该数字应该存放到 Hash 表中的哪个对应数字的格子中。如果改格子中已经有数字存在了，那么就以链表的方式将数字依次存放在该格子中，如下表：

0	1	2	3	4	5	6	7
32	9		11	44	45	6	23
	57			12			
	89						

Hash 表和 Hash 算法的特点就是它的存取速度比数组差一些，但是比起单纯的链表，在查找和存储方面却要好很多。同时数组也不利于数据的重构而排序等方面的要求。

简单的了解了一下 Hash 算法后，我们就来看看 HashMap 的属性有哪些：
里面最重要的 3 个属性：

transient Entry[] table: 用来存放键值对的对象 Entry 数组，也就是 Hash 表
transient int size：当前 Map 中存放的键值对的个数
final float loadFactor：负载因子，用来决定什么情况下应该对 Entry 进行扩容

我们 Entry 对象是 Map 接口中的一个内部接口。即是使用它来保存我们的键值对的。
我们看看这个 Entry 内部接口在 HashMap 中的实现：

```
static class Entry<K,V> implements Map.Entry<K,V> {}
```

可以查看源码，我们可以看到 Entry 类有点类似一个单向链表。其中：

```
final K key;
V value;
Entry<K,V> next;
final int hash;
```

key 和 value ;存放的就是我们放入 Map 中的键值对。
而属性 Entry next;表示当前键值对的下一个键值对是哪个 Entry。
接下来，我们看看 HashMap 的主要的构造函数：
我们主要看看 **public** HashMap(**int** initialCapacity, **float** loadFactor)

```
public HashMap(int initialCapacity, float loadFactor) {
        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);
      // Find a power of 2 >= initialCapacity
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;
        this.loadFactor = loadFactor;
        threshold = (int)(capacity * loadFactor);
        table = new Entry[capacity];
        init();
```

}

因为，另外两个构造函数实行也是同样的方式进行构建一个 HashMap 的。

该构造函数：

1. 首先是判断参数 **int** initialCapacity 和 **float** loadFactor 是否合法
2. 然后确定 Hash 表的初始化长度。确定的策略是：通过传进来的参数 initialCapacity 来找出第一个大于它的 2 的次方的数。比如说我们传了 18 这样的一个 initialCapacity 参数，那么真实的 table 数组的长度为 2 的 5 次方，即 32。之所以采用这种策略来构建 Hash 表的长度，是因为 2 的次方的运算对于现代的处理器来说，可以通过一些方法得到更加好的执行效率。
3. 接下来就是得到重构因子（threshold）了，这个属性也是 HashMap 中的一个比较重要的属性，它表示，当 Hash 表中的元素被存放了多少个之后，我们就需要对该 Hash 表进行重构。
4. 最后就是使用得到的初始化参数 capacity 来构建 Hash 表：Entry[] table。

下面我们看看一个键值对是如何添加到 HashMap 中的。

```
public V put(K key, V value) {
        if (key == null)
            return putForNullKey(value);
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            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;
        }
```

该 put 方法是用来添加一个键值对（key-value）到 Map 中，如果 Map 中已经存在相同的键的键值对的话，那么就把新的值覆盖老的值，并把老的值返回给方法的调用者。如果不存在一样的键，那么就返回 **null** 。我们看看方法的具体实现：

1. 首先我们判断如果 key 为 null 则使用一个常量来代替该 key 值，该行为在方法 maskNull()终将 key 替换为一个非 **null** 的对象 k。
2. 计算 key 值的 Hash 码：hash
3. 通过使用 Hash 码来定位，我们应该把当前的键值对存放到 Hash 表中的哪个格子中。indexFor()方法计算出的结果：i 就是 Hash 表（table）中的下标。
4. 然后遍历当前的 Hash 表中 table[i]格中的链表。从中判断已否已经存在一样的键(Key)的键值对。如果存在一样的 key 的键，那么就用新的 value 覆写老的 value，并把老的 value 返回
5. 如果遍历后发现没有存在同样的键值对，那么就增加当前键值对到 Hash 表中的第 i 个格子中的链表中。并返回 **null** 。

最后我们看看一个键值对是如何添加到各个格子中的链表中的：

我们先看 `void addEntry(int hash, Object key, Object value, int bucketIndex)` 方法：

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K,V> e = table[bucketIndex];
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        if (size++ >= threshold)
            resize(2 * table.length);
        }
```

该方法的作用就用来添加一个键值对到 Hash 表的第 bucketIndex 个格子中的链表中去。这个方法作的工作就是：

1. 创建一个 Entry 对象用来存放键值对。
2. 添加该键值对 ---- Entry 对象到链表中
3. 最后在 size 属性加一，并判断是否需要对当前的 Hash 表进行重构。如果需要就在 `void resize(int newCapacity)` 方法中进行重构。

之所以需要重构，也是基于性能考虑。大家可以考虑这样一种情况，假定我们的 Hash 表只有 4 个格子，那么我们所有的数据都是放到这 4 个格子中。如果存储的数据量比较大的话，例如 100。这个时候，我们就会发现，在这个 Hash 表中的 4 个格子存放的 4 个长长的链表。而我们每次查找元素的时候，其实相当于就是遍历链表了。这种情况下，我们用这个 Hash 表来存取数据的性能实际上和使用链表差不多了。

但是如果我们对这个 Hash 表进行重构，换为使用 Hash 表长度为 200 的表来存储这 100 个数据，那么平均 2 个格子里面才会存放一个数据。这个时候我们查找的数据的速度就会非常的快。因为基本上每个格子中存放的链表都不会很长，所以我们遍历链表的次数也就很少，这样也就加快了查找速度。但是这个时候又存在了另外的一个问题。我们使用了至少 200 个数据的空间来存放 100 个数据，这样就造成至少 100 个数据空间的浪费。 在速度和空间上面，我们需要找到一个适合自己的中间值。在 HashMap 中我们通过负载因子（loadFactor）来决定应该什么时候应该重构我们的 Hash 表，以达到比较好的性能状态。

我们再看看重构 Hash 表的方法：**void** resize(**int** newCapacity)是如何实现的：

```
void resize(int newCapacity) {
        Entry[] oldTable = table;
        int oldCapacity = oldTable.length;
        if (oldCapacity == MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return;
        }

        Entry[] newTable = new Entry[newCapacity];
        transfer(newTable);
        table = newTable;
        threshold = (int)(newCapacity * loadFactor);
    }
```

它的实现方式比较简单：

1.首先判断如果 Hash 表的长度已经达到最大值，那么就不进行重构了。因为这个时候 Hash 表的长度已经达到上限，已经没有必要重构了。

2.然后就是构建新的 Hash 表

3.把老的 Hash 表中的对象数据全部转移到新的 Hash 表 newTable 中，并设置新的重构因子 threshold

对于 HashMap 中的实现原理，我们就分析到这里。大家可能会发现，HashCode 的计算，是用来定位我们的键值对应该放到 Hash 表中哪个格子中的关键属性。而这个 HashCode 的计算方法是调用的各个对象自己的实现的 hashCode()方法。而这个方法是在 Object 对象中定义的，所以我们自己定义的类如果要在集合中使用的话，就需要正确的覆写 hashCode() 方法。下面就介绍一下应该如何正确覆写 hashCode()方法。

1.5 覆写 hashCode()

在明白了 HashMap 具有哪些功能，以及实现原理后，了解如何写一个 hashCode()方法就更有意义了。当然，在 HashMap 中存取一个键值对涉及到的另外一个方法为 equals ()。

设计 hashCode()时最重要的因素就是：无论何时，对同一个对象调用 hashCode()都应该生成同样的值。如果在将一个对象用 put()方法添加进 HashMap 时产生一个 hashCode()值，而用 get()取出时却产生了另外一个 hashCode()值，那么就无法重新取得该对象了。所以，如果你的 hashCode()方法依赖于对象中易变的数据，那用户就要

小心了，因为此数据发生变化时，hashCode()就会产生一个不同的 hash 码，相当于产生了一个不同的“键”。

此外，也不应该使 hashCode()依赖于具有唯一性的对象信息，尤其是使用 this 的值，这只能产生很糟糕的 hashCode()。因为这样做无法生成一个新的“键”，使之与 put()种原始的“键值对”中的“键”相同。例如，如果我们不覆写 Object 的 hashCode()方法，那么调用该方法的时候，就会调用 Object 的 hashCode()方法的默认实现。Object 的 hashCode（）方法，返回的是当前对象的内存地址。下次如果我们需要取一个一样的“键”对应的键值对的时候，我们就无法得到一样的 hashCode 值了。因为我们后来创建的“键”对象已经不是存入 HashMap 中的那个内存地址的对象了。

我们看一个简单的例子，就能更加清楚的理解上面的意思。假定我们写了一个类：Person （人），我们判断一个对象“人”是否指向同一个人，只要知道这个人的身份证号一直就可以了。

先看我们没有实现 hashCode 的情况：

```
package com.test;
import java.util.HashMap;
public class HashCodeDemo {
    public static void main(String[] args) {
        HashMap map=new HashMap();
        Person p1=new Person("张三",new Code(123));
        Person p2=new Person("李四",new Code(456));
        map.put(p1.id,p1);//我们根据身份证来作为key值存放到Map中
        map.put(p2.id,p2);
        System.out.println("HashMap 中存放的人员信息:\n"+map);
        // 张三 改名为：张山 但是还是同一个人。
        Person p3=new Person("张山",new Code(123));
        map.put(p3.id,p3);
        System.out.println("张三改名后 HashMap 中存放的人员信
息:\n"+map);
        //查找身份证为：123 的人员信息
        System.out.println("查找身份证为：123 的人员信
```

```
息:"+map.get(new Code(123)));
    }
}
//身份证类
class Code{
    public final int id;//身份证号码已经确认,不能改变
    public Code(int i){
        this.id=i;
    }
    //身份号号码相同，则身份证相同
    public boolean equals(Object anObject) {
        if (anObject instanceof Code){
            Code other=(Code) anObject;
            return this.id==other.id;
        }
        return false;
    }
    public String toString() {
        return "身份证:"+id;
    }
}
//人员信息类
class Person {
    public Code id;// 身份证
    public String name;// 姓名
    public Person(String name, Code id) {
        this.id=id;
        this.name=name;
    }
    //如果身份证号相同，就表示两个人是同一个人
    public boolean equals(Object anObject) {
        if (anObject instanceof Person){
            Person other=(Person) anObject;
            return this.id.equals(other.id);
        }
        return false;
    }
    public String toString() {
        return "姓名:"+name+" 身份证:"+id.id+"\n";
    }
    }
```

运行结果为：

HashMap 中存放的人员信息：

```
HashMap 中存放的人员信息:
{身份证:456=姓名:李四 身份证:456
, 身份证:123=姓名:张三 身份证:123
}
张三改名后 HashMap 中存放的人员信息:
{身份证:123=姓名:张山 身份证:123
, 身份证:456=姓名:李四 身份证:456
, 身份证:123=姓名:张三 身份证:123
}
查找身份证为：123 的人员信息:null
```

上面的例子的演示的是，我们在一个 HashMap 中存放了一些人员的信息。并以这些人员的身份证最为人员的“键”。当有的人员的姓名修改了的情况下，我们需要更新这个 HashMap。同时假如我们知道某个身份证号，想了解这个身份证号对应的人员信息如何，我们也可以根据这个身份证号在 HashMap 中得到对应的信息。

而例子的输出结果表示，我们所做的更新和查找操作都失败了。失败的原因就是我们的身份证类：Code 没有覆写 hashCode()方法。这个时候，当查找一样的身份证号码的键值对的时候，使用的是默认的对象的内存地址来进行定位。这样，后面的所有的身份证号对象 new Code(123) 产生的 hashCode()值都是不一样的。所以导致操作失败。

下面，我们给 Code 类加上 hashCode（）方法，然后再运行一下程序看看：

```
package com.test;
import java.util.HashMap;
public class HashCodeDemo {
    public static void main(String[] args) {
        HashMap map=new HashMap();
        Person p1=new Person("张三",new Code(123));
        Person p2=new Person("李四",new Code(456));
        map.put(p1.id,p1);//我们根据身份证来作为key值存放到Map中
        map.put(p2.id,p2);
        System.out.println("HashMap 中存放的人员信息:\n"+map);
        // 张三 改名为：张山 但是还是同一个人。
        Person p3=new Person("张山",new Code(123));
        map.put(p3.id,p3);
        System.out.println("张三改名后 HashMap 中存放的人员信
息:\n"+map);
```

```
        //查找身份证为：123 的人员信息
        System.out.println("查找身份证为：123 的人员信
息:"+map.get(new Code(123)));
    }
}
//身份证类
class Code{
    public final int id;//身份证号码已经确认,不能改变
    public Code(int i){
        this.id=i;
    }
    //身份号号码相同，则身份证相同
    public boolean equals(Object anObject) {
        if (anObject instanceof Code){
            Code other=(Code) anObject;
            return this.id==other.id;
        }
        return false;
    }
    public String toString() {
        return "身份证:"+id;
    }
    //覆写hashCode方法，并使用身份证号作为hash值
        public int hashCode(){
            return id;
        }

}
//人员信息类
class Person {
    public Code id;// 身份证
    public String name;// 姓名
    public Person(String name, Code id) {
        this.id=id;
        this.name=name;
    }
    //如果身份证号相同，就表示两个人是同一个人
    public boolean equals(Object anObject) {
        if (anObject instanceof Person){
            Person other=(Person) anObject;
            return this.id.equals(other.id);
        }
        return false;
    }
```

```
public String toString() {
    return "姓名:"+name+" 身份证:"+id.id+"\n";
}
 }
```

再次执行上面的 HashCodeDemo 的结果就为：

```
HashMap 中存放的人员信息:
{身份证:456=姓名:李四 身份证:456
, 身份证:123=姓名:张三 身份证:123
}
张三改名后 HashMap 中存放的人员信息:
{身份证:456=姓名:李四 身份证:456
, 身份证:123=姓名:张山 身份证:123
}
查找身份证为：123 的人员信息:姓名:张山 身份证:123
```

查找身份证为：123 的人员信息:姓名:张山 身份证:123

这个时候，我们发现。我们想要做的更新和查找操作都成功了。

对于 Map 部分的使用和实现，主要就是需要注意存放“键值对”中的对象的 equals()方法和 hashCode()方法的覆写。如果需要使用到排序的话，那么还需要实现 Comparable 接口中的 compareTo() 方法。我们需要注意 Map 中的“键”是不能重复的，而是否重复的判断，是通过调用“键”对象的 equals（）方法来决定的。而在 HashMap 中查找和存取“键值对”是同时使用 hashCode（）方法和 equals（）方法来决定的。

Set

1.1 概述

Java 中的 Set 和正好和数学上直观的集（set）的概念是相同的。Set 最大的特性就是不允许在其中存放的元素是重复的。根据这个特点，我们就可以使用 Set 这个接口来实现前面提到的关于商品种类的存储需求。Set 可以被用来过滤在其他集合中存放的元素，从而得到一个没有包含重复新的集合。

1.5.2 常用方法

按照定义，Set 接口继承 Collection 接口，而且它不允许集合中存在重复项。所有原始方法都是现成的，没有引入新方法。具体的 Set 实现类依赖添加的对象的 equals() 方法来检查等同性。

我们简单的描述一下各个方法的作用：

- public int size() ：返回 set 中元素的数目，如果 set 包含的元素数大于 Integer.MAX_VALUE，返回 Integer.MAX_VALUE
- public boolean isEmpty() ：如果 set 中不含元素，返回 true
- public boolean contains(Object o) ：如果 set 包含指定元素，返回 true
- public Iterator iterator()
  - 返回 set 中元素的迭代器
  - 元素返回没有特定的顺序，除非 set 是提高了该保证的某些类的实例
- public Object[] toArray() ：返回包含 set 中所有元素的数组
- public Object[] toArray(Object[] a) ：返回包含 set 中所有元素的数组，返回数组的运行时类型是指定数组的运行时类型
- public boolean add(Object o) ：如果 set 中不存在指定元素，则向 set 加入
- public boolean remove(Object o) ：如果 set 中存在指定元素，则从 set 中删除
- public boolean removeAll(Collection c) ：如果 set 包含指定集合，则从 set 中删除指定集合的所有元素
- public boolean containsAll(Collection c) ：如果 set 包含指定集合的所有元素，返回 true。如果指定集合也是一个 set，只有是当前 set 的子集时，方法返回 true
- public boolean addAll(Collection c) ：如果 set 中中不存在指定集合的元素，则向 set 中加入所有元素
- public boolean retainAll(Collection c) ：只保留 set 中所含的指定集合的元素（可选操作）。换言之，从 set 中删除所有指定集合不包含的元素。 如果指定集合也是一个 set，那么该操作修改

set 的效果是使它的值为两个 set 的交集

- public boolean removeAll(Collection c) ：如果 set 包含指定集合，则从 set 中删除指定集合的所有元素
- public void clear() ：从 set 中删除所有元素

“集合框架”支持 Set 接口两种普通的实现：HashSet 和 TreeSet 以及 LinkedHashSet。下表中是 Set 的常用实现类的描述：

	简述	实现	操作特性	成员要求
Set	成员不能重复	HashSet	外部无序地遍历成员。	成员可为任意 Object 子类的对象，但如果覆盖了 equals 方法，同时注意修改 hashCode 方法。
		TreeSet	外部有序地遍历成员；附加实现了 SortedSet，支持子集等要求顺序的操作	成员要求实现 Comparable 接口，或者使用 Comparator 构造 TreeSet。成员一般为同一类型。

在更多情况下，您会使用 HashSet 存储重复自由的集合。同时 HashSet 中也是采用了 Hash 算法的方式进行存取对象元素的。所以添加到 HashSet 的对象对应的类也需要采用恰当方式来实现 hashCode() 方法。虽然大多数系统类覆盖了 Object 中缺省的 hashCode() 实现，但创建您自己的要添加到 HashSet 的类时，别忘了覆盖 hashCode()。

对于 Set 的使用，我们先以一个简单的例子来说明：

```
package com.test;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class HashSetDemo {
   public static void main(String[] args) {
```

```
        Set set1 = new HashSet();
        if (set1.add("a")) {//添加成功
            System.out.println("1 add true");
        }
        if (!set1.add("a")) {//添加失败
            System.out.println("2 add false");
        }
        set1.add("000");//添加对象到Set集合中
        set1.add("111");
        set1.add("22");
        System.out.println("集合set1的大小: "+set1.size());
        System.out.println("集合set1的内容: "+set1);
        set1.remove("000");//从集合set1中移除掉 "000" 这个对象
        System.out.println("集合set1移除 000 后的内容: "+set1);
        System.out.println("集合set1中是否包含000 :
"+set1.contains("000"));
        System.out.println("集合set1中是否包含111 :
"+set1.contains("111"));
        Set set2=new HashSet();
        set2.add("111");
        set2.addAll(set1);//将set1 集合中的元素全部都加到set2中
        System.out.println("集合set2的内容: "+set2);
        set2.clear();//清空集合 set1 中的元素
        System.out.println("集合set2是否为空 : "+set2.isEmpty());
        Iterator iterator = set1.iterator();//得到一个迭代器
        while (iterator.hasNext()) {//遍历
            Object element = iterator.next();
            System.out.println("iterator = " + element);
        }
        //将集合set1转化为数组
        Object s[]= set1.toArray();
        for(int i=0;i<s.length;i++){
            System.out.println(s[i]);
        }
    }
}
```

程序执行的结果为：

```
1 add true
2 add false
集合set1的大小: 4
集合set1的内容: [111, a, 22, 000]
```

```
集合set1移除 000 后的内容：[111, a, 22]
集合set1中是否包含000 ： false
集合set1中是否包含111 ： true
集合set2的内容：[a, 111, 22]
集合set2是否为空 ： true
iterator = 111
iterator = a
iterator = 22
111
a
22
```

从上面的这个简单的例子中，我们可以发现，Set 中的方法与直接使用 Collection 中的方法一样。唯一需要注意的就是 Set 中存放的元素不能重复。

我们再看一个例子，来了解一下其它的 Set 的实现类的特性：

```
package c08;
import java.util.*;
public class SetSortExample {
  public static void main(String args[]) {
    Set set1 = new HashSet();
    Set set2 = new LinkedHashSet();
    for(int i=0;i<5;i++){
     //产生一个随机数，并将其放入 Set 中
     int s=(int) (Math.random()*100);
      set1.add(new Integer( s));
      set2.add(new Integer( s));
      System.out.println("第 "+i+" 次随机数产生为： "
+s);
    }
    System.out.println("未排序前 HashSet: "+set1);
    System.out.println("未排序前 LinkedHashSet: "+se
t2);
    //使用 TreeSet 来对另外的 Set 进行重构和排序
    Set sortedSet = new TreeSet(set1);
```

```
        System.out.println("排序后 TreeSet : "+sortedSet);
      }
    }
```

该程序的一次执行结果为：

```
第 0 次随机数产生为：96
第 1 次随机数产生为：64
第 2 次随机数产生为：14
第 3 次随机数产生为：95
第 4 次随机数产生为：57
未排序前 HashSet：[64, 96, 95, 57, 14]
未排序前 LinkedHashSet：[96, 64, 14, 95, 57]
排序后 TreeSet : [14, 57, 64, 95, 96]
```

从这个例子中，我们可以知道 HashSet 的元素存放顺序和我们添加进去时候的顺序没有任何关系，而 LinkedHashSet 则保持元素的添加顺序。TreeSet 则是对我们的 Set 中的元素进行排序存放。

一般来说，当您要从集合中以有序的方式抽取元素时，TreeSet 实现就会有用处。为了能顺利进行，添加到 TreeSet 的元素必须是可排序的。 而您同样需要对添加到 TreeSet 中的类对象实现 Comparable 接口的支持。对于 Comparable 接口的实现，在前一小节的 Map 中已经简单的介绍了一下。我们暂且假定一棵树知道如何保持 java.lang 包装程序器类元素的有序状态。一般说来，先把元素添加到 Ha

shSet，再把集合转换为 TreeSet 来进行有序遍历会更快。这点和 HashMap 的使用非常的类似。

其实 Set 的实现原理是基于 Map 上面的。通过下面我们对 Set 的进一步分析大家就能更加清楚的了解这点了。

1.3 实现原理

Java 中 Set 的概念和数学中的集合(set)一致，都表示一个集内可以存放的元素是不能重复的。

前面我们会发现，Set 中很多实现类和 Map 中的一些实现类的使用上非常的相似。而且前面再讲解 Map 的时候，我们也提到：Map 中的“键值对”，其中的“键”是不能重复的。这个和 Set 中的元素不能重复一致。我们以 HashSet 为例来分析一下，会发现其实 Set 利用的就是 Map 中“键”不能重复的特性来实现的。

先看看 HashSet 中的有哪些属性：

```
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();
```

再结合构造函数来看看：

```
public HashSet() {
    map = new HashMap<E,Object>();
 }
```

通过这些方法，我们可以发现，其实 HashSet 的实现，全部的操作都是基于 HashMap 来进行的。我们看看是如何通过 HashMap 来保证我们的 HashSet 的元素不重复性的：

看到这个操作我们可以发现 HashSet 的巧妙实现：就是建立一个“键值对”，“键”就是我们要存入的对象，“值”则是一个常量。这样可以确保，我们所需要的存储的信息之是“键”。而“键”在 Map 中是不能重复的，这就保证了我们存入 Set 中的所有的元素都不重复。而判断是否添加元素成功，则是通过判断我们向 Map 中存入的“键值对”是否已经存在，如果存在的话，那么返回值肯定是常量：PRESENT ，表示添加失败。如果不存在，返回值就为 **null** 表示添加成功。

了解了这些后，我们就不难理解，为什么 HashMap 中需要注意的地方，在 HashSet 中也同样的需要注意。其他的 Set 的实现类也是差不多的原理。

总结:集合框架中常用类比较

用"集合框架"设计软件时，记住该框架四个基本接口的下列层次结构关系会有用处：

- Collection 接口是一组允许重复的对象。
- Set 接口继承 Collection，但不允许重复。
- List 接口继承 Collection，允许重复，并引入位置下标。
- Map 接口既不继承 Set 也不继承 Collection，存取的是键值对

我们以下面这个图表来描述一下常用的集合的实现类之间的区别：

Collection/Map	接口	成员重复性	元素存放顺序（Ordered/Sorted）	元素中被调用的方法	基于那中数据结构来实现的
HashSet	Set	Unique elements	No order	equals() hashCode()	Hash 表
LinkedHashSet	Set	Unique elements	Insertion order	equals() hashCode()	Hash 表和双向链表
TreeSet	Sorted Set	Unique elements	Sorted	equals() compareTo()	平衡树（Balanced tree）
ArrayList	List	Allowed	Insertion order	equals()	数组
LinkedList	List	Allowed	Insertion order	equals()	链表
Vector	List	Allowed	Insertion order	equals()	数组
HashMap	Map	Unique keys	No order	equals() hashCode()	Hash 表
LinkedHashMap	Map	Unique keys	Key insertion order/Access order of entries	equals() hashCode()	Hash 表和双向链表

Collection/Map	接口	成员重复性	元素存放顺序（Ordered/Sorted）	元素中被调用的方法	基于那中数据结构来实现的
Hashtable	Map	Unique keys	No order	equals() hashCode()	Hash 表
TreeMap	Sorted Map	Unique keys	Sorted in key order	equals() compareTo()	平衡树（Balanced tree）

2 练习

撰写一个 Person 类，表示一个人员的信息。令该类具备多辆 Car 的信息，表示一个人可以拥有的车子的数据，以及：

Certificate code： 身份证对象

name： 姓名

cash： 现金

List car; 拥有的汽车,其中存放的是 Car 对象

boolean buycar(car); 买车子

boolean sellcar(Person p);//把自己全部的车子卖给别人

boolean buyCar(Car car,Person p);//自动查找卖车的人 p 是否有买主想要买的车 car，如果有就买,并返回 true ,否则返回 false

viod addCar(car);//把某辆车送给方法的调用者。

String toString();//得到人的信息

并撰写第二个 Car 类 具备的属性：

String ID; //ID 车牌号

cost //价格

color //颜色

Person owner; //车子的拥有者

toString();//得到汽车的信息

equals();//比较车子是否同一俩汽车，ID 相同则认为相同

在另外一个 Market 类里面，进行车子的买卖。并保留所有交易人员的的信息到一个 HashMap 中，我们可以通过身份证号来查找到对应的人员的信息。同时所有的车子种类都在市场中进行注册，即车子的信息使用一个 Set 来保存

属性：

HashMap people;//存放交易人员的信息。Key 为身份证号，value 为 Person 对象

方法：

static boolean sellCar(Person p1 ,Car car1, Person p2);

//p1 将 car1 卖给 p2 。并在该方法中记录效益人的信息到 people 中。

撰写类 Certificate 表示身份证：

属性：
Id;//号码
方法：
equals();//比较两个身份证是否同一个，ID 相同则认为相同
hashCode();//正确编写 hashCode 方法

场景：
一个叫 Bob 的人：身份证：310　　现金：30000。
有一辆车子：ID:001,红色，价格：50000 的车子;
一个叫 Tom 的人：身份证：210 现金：70000,
有一辆车子: 颜色：白色,ID:003, 价格：25000。
一个叫 King 的人：身份证：245 现金：60000,
有 2 辆车子: 颜色：白色,ID:005, 价格：18000。
颜色：红色,ID:045, 价格：58000。

Tom 买了 Bob 的车子.他就拥有了 2 辆汽车
King 把 ID=005 的车子买给了 Bob
最后各人的信息如何？