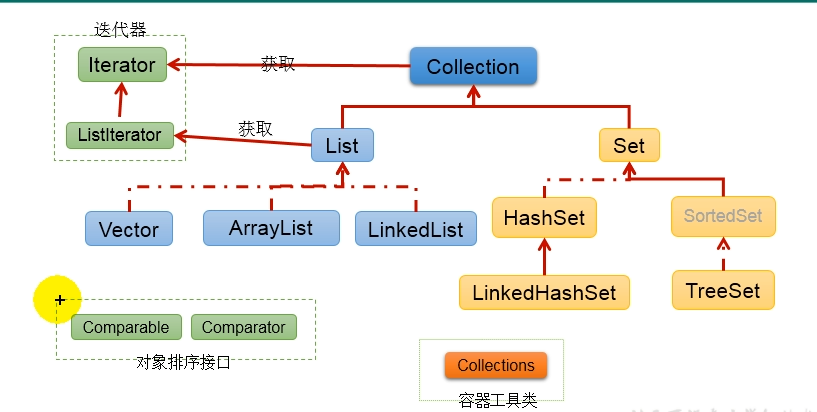
## collection 的结构和常用方法



add(Object obj)   
addAll(Collection coll)  
**int** size()  
**void** clear()  
**boolean** isEmpty()  
**boolean** contains(Object obj)：是通过元素的equals方法来判断是否 是同一个对象  
**boolean** containsAll(Collection c)：也是调用元素的equals方法来比 较的。拿两个集合的元素挨个比较。  
**boolean** remove(Object obj) ：通过元素的equals方法判断是否是 要删除的那个元素。只会删除找到的第一个元素  
**boolean** removeAll(Collection coll)：取当前集合的差集  
**boolean** retainAll(Collection c)：把交集的结果存在当前集合中，不影响c  
**boolean** equals(Object obj)  
Object[] toArray()  
hashCode()  
iterator()

## iterator的常用方法

Collection coll = **new** ArrayList();

Iterator iterator = coll.iterator();

集合元素的遍历操作，使用迭代器Iterator接口  
 1.内部的方法：hasNext() 和 next():①指针下移 ②将下移以后集合 位置上的元素返回  
 2.集合对象每次调用iterator()方法都得到一个全新的迭代器对象，  
 默认游标都在集合的第一个元素之前。  
 3.内部定义了remove(),可以在遍历的时候，删除集合中的元素。此方法不同于集合直接调用remove()

## 增强for 循环的原理

for(集合元素的类型 局部变量 : 集合对象)  
 内部仍然调用了迭代器。

## ArrayList 源码分析

关键词：**elementData、扩容1.5倍、带参构造器**

**jdk 7情况下**  
 ArrayList list = new ArrayList();//***底层创建了长度是10的Object[]数组elementData***  
 list.add(123);//elementData[0] = new Integer(123);  
 list.add(11);//如果此次的添加导致底层elementData数组容量不够，则扩容。 默认情况下，扩容为原来的容量的***1.5***倍，同时需要将原有数组中的数据复制到新的数组中。  
  ***结论：建议开发中使用带参的构造器：***

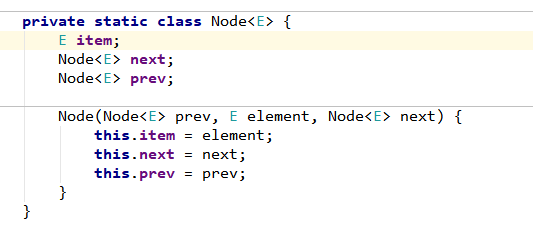
***ArrayList list = new ArrayList(int capacity)***

***可以省去扩容的操作，增加性能。***

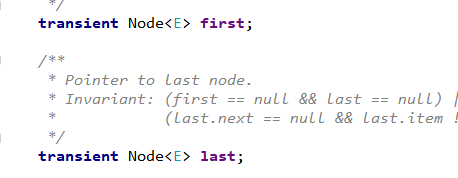
**jdk 8中ArrayList的变化**  
 ArrayList list = new ArrayList();//***底层Object[] elementData初始化为{}.并没有创建长度为10的数组***  
 list.add(123);//***第一次调用add()时，底层才创建了长度10的数组，并将数据123添加到elementData[0]***  
 后续的添加和扩容操作与jdk 7 无异。  
 2.3 小结：jdk7中的ArrayList的对象的创建类似于单例的饿汉式，而jdk8中的ArrayList的对象的创建类似于单例的懒汉式，延迟了数组的创建，节省内存。

## LinkedList 源码分析

1、内部使用***Node*** 对象存储数据，Node对象有三个属性

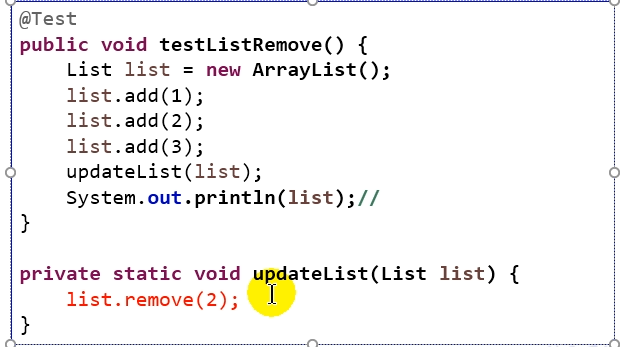


1. LinkedList内部声明了Node类型的first和last属性，默认值为null，记住第一个和最后一个Node



## list 的remove 方法问题

删除对象还是删除索引



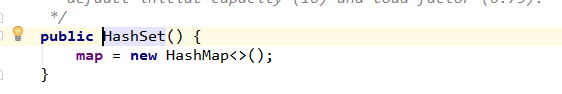
## HashSet特性及元素的存储

### · 特性

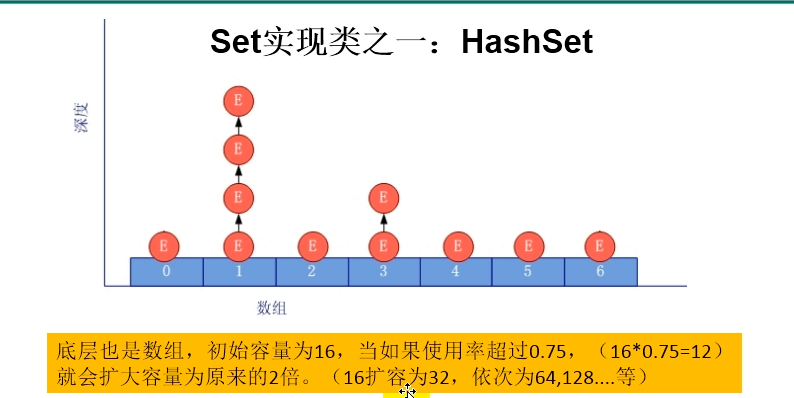
1. 无序性：不等于随机性。存储的数据在***底层数组***中并非按照数组索引的顺序添加，而是根据数据的***哈希值***决定的。  
  
2. 不可重复性：保证添加的元素按照***hashCode()***和***equals()***判断时，不能返回true.即：相同的元素只能添加一个。

### · 底层数据结构

底层实现



数据结构

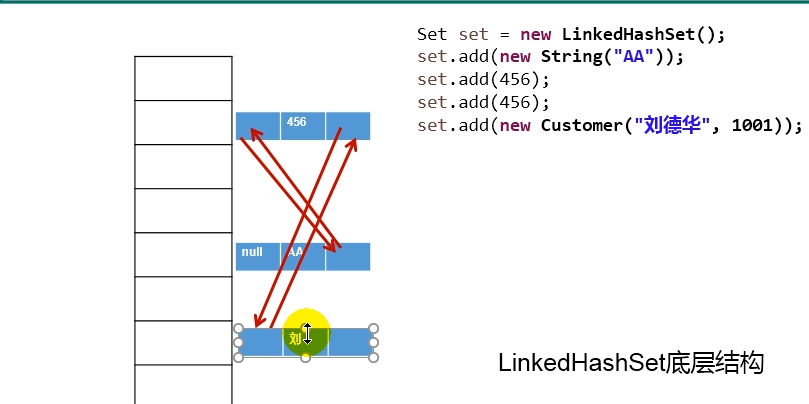


我们向HashSet中添加元素a,首先调用元素a所在类的hashCode()方法，计算元素a的哈希值，  
此哈希值接着通过散列算法计算出在HashSet底层数组中的存放位置（即为：索引位置），判断  
数组此位置上是否已经有元素：  
 如果此位置上没有其他元素，则元素a添加成功。 --->情况1  
 如果此位置上有其他元素b(或以链表形式存在的多个元素），则比较元素a与元素b的hash值：  
 如果hash值不相同，则元素a添加成功。--->情况2  
 如果hash值相同，进而需要调用元素a所在类的equals()方法：  
 equals()返回true,元素a添加失败  
 equals()返回false,则元素a添加成功。--->情况2  
  
对于添加成功的情况2和情况3而言：元素a 与已经存在指定索引位置上数据以链表的方式存储。  
jdk 7 :元素a放到数组中，指向原来的元素。  
jdk 8 :原来的元素在数组中，指向元素a  
总结：七上八下  
  
HashSet底层：***数组+链表***的结构。

### · 要求

***要求***：向Set(主要指：HashSet、LinkedHashSet)中添加的数据，其所在的类一定要重写hashCode()和equals()  
 重写的hashCode()和equals()尽可能保持一致性：***相等的对象必须具有相等的散列码***  
 重写两个方法的小技巧：对象中用作 equals() 方法比较的 Field，都应该用来计算 hashCode 值。

## LinkedHashSet特性及数据结构



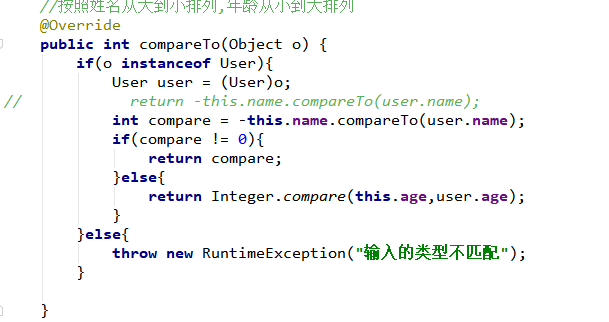
LinkedHashSet作为HashSet的子类，在添加数据的同时，每个数据还维护了***两个引用***，记录此数据前一个数据和后一个数据。  
优点：对于频繁的遍历操作，LinkedHashSet效率高于HashSet

## TreeSet特性

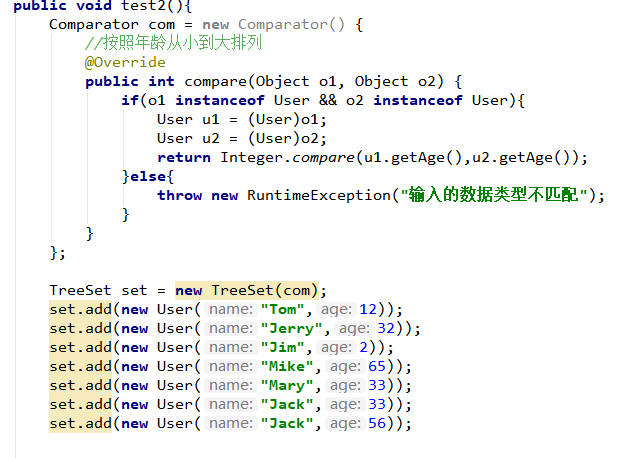
1. 向TreeSet中添加的数据，要求是相同类的对象。

2.底层使用的是***红黑树***  
3.两种排序方式：自然排序（实现Comparable接口） 和 定制排序（Comparator）  
自然排序中，比较两个对象是否相同的标准为：***compareTo()返回0.不再是equals().***

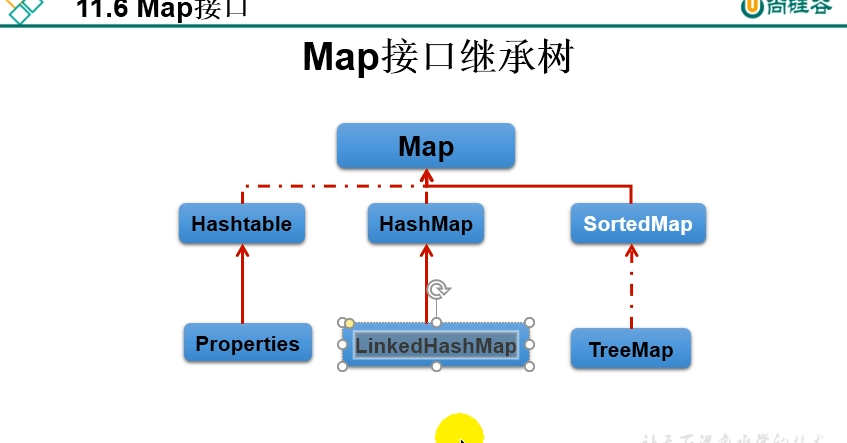




4.定制排序中，比较两个对象是否相同的标准为：compare()返回0.不再是equals().



## Map



**HashMap**:作为Map的主要实现类；线程不安全的，效率高；存储null的key和value  
**LinkedHashMap**保证在遍历map元素时，可以按照添加的顺序实现遍历。  
 原因：在原有的HashMap底层结构基础上，添加了一对指针，指向前一个和后一个元素。对于频繁的遍历操作，此类执行效率高于HashMap。  
**TreeMap**:保证按照添加的key-value对进行排序，实现排序遍历。此时考虑key的自然排序或定制排序底层使用红黑树  
**Hashtable**:作为古老的实现类；线程安全的，效率低；不能存储null的key和value  
**Properties**:常用来处理配置文件。key和value都是String类型

## 十一、HashMap 和 Hashtable的区别

1、HashMap 和 Hashtable 好比 ArrayList 和Vector 后者都是被弃用并且线程安全。

2、HashMap 可以存null的 key， hashtable 不行。

## 十二、HashMap 底层实现原理

HashMap map = new HashMap():  
 在实例化以后，底层创建了长度是16的一维数组**Entry[] table**。  
 ...可能已经执行过多次put...  
 map.put(key1,value1):  
 首先，调用key1所在类的hashCode()计算key1哈希值，此哈希值经过某种算法计算以后，得到在Entry数组中的存放位置。  
 如果此位置上的数据为空，此时的key1-value1添加成功。 ----情况1  
 如果此位置上的数据不为空，(意味着此位置上存在一个或多个数据(以链表形式存在)),比较key1和已经存在的一个或多个数据的哈希值：  
 如果key1的哈希值与已经存在的数据的哈希值都不相同，此时key1-value1添加成功。----情况2  
 如果key1的哈希值和已经存在的某一个数据(key2-value2)的哈希值相同，继续比较：调用key1所在类的equals(key2)方法，比较：  
 如果equals()返回false:此时key1-value1添加成功。----情况3  
 如果equals()返回true:使用value1替换value2。  
  
 补充：关于情况2和情况3：此时key1-value1和原来的数据以链表的方式存储。  
  
 在不断的添加过程中，会涉及到扩容问题，当超出临界值(且要存放的位置非空)时，扩容。默认的扩容方式：扩容为原来容量的2倍，并将原有的数据复制过来。

### ·jdk8 相较于jdk7在底层实现方面的不同

1. new HashMap():底层没有创建一个长度为16的数组

2. jdk 8底层的数组是：**Node[]**,而非Entry[]

3. 首次调用put()方法时，底层创建长度为16的数组

4. jdk7底层结构只有：数组+链表。jdk8中底层结构：***数组+链表+红黑***树。

4.1 形成链表时，七上八下（jdk7:新的元素指向旧的元素。jdk8：旧的元素指向新的元素）

4.2 当数组的某一个索引位置上的元素以链表形式存在的数据个数 > ***8*** 且当前数组的长度 > ***64***时，此时此索引位置上的所数据改为使用红黑树存储。

**DEFAULT\_INITIAL\_CAPACITY : HashMap的默认容量，16**

**DEFAULT\_LOAD\_FACTOR：HashMap的默认加载因子：0.75**

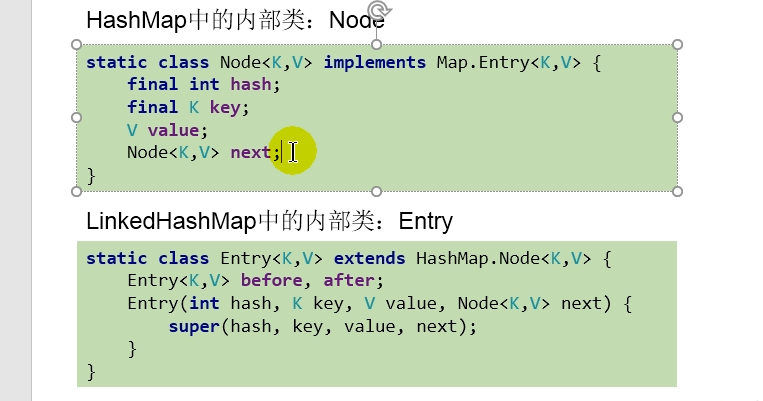
**threshold：扩容的临界值，=容量\*填充因子：16 \* 0.75 => 12**

TREEIFY\_THRESHOLD：***桶***中链表长度大于该默认值，转化为红黑树:8

MIN\_TREEIFY\_CAPACITY：桶中的Node被树化时最小的***hash表***容量:64

所以如果我们已经预知HashMap中元素的个数， 那么预设元素的个数能够有效的提高HashMap的性能

## LinkedHashMap底层实现

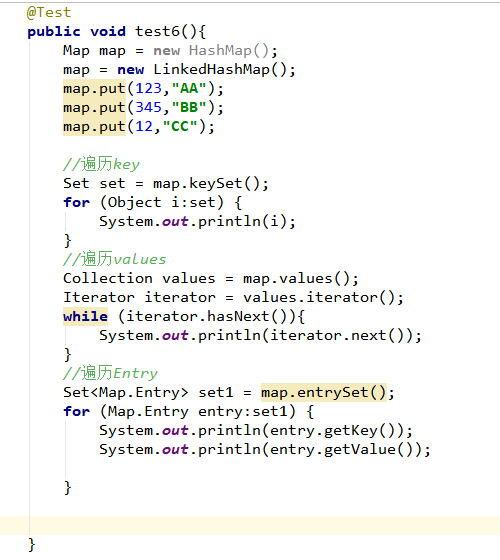


有多次遍历的诉求使用LinkedHashSet

## TreeMap排序



## map 的遍历



## Collections工具类

Collections 类中提供了多个 **synchronizedXxx()** 方法，  
该方法可使将指定集合包装成线程同步的集合，从而可以解决  
多线程并发访问集合时的线程安全问题

**reverse(List)**：反转 List 中元素的顺序  
**shuffle(List)**：对 List 集合元素进行随机排序  
**sort(List)**：根据元素的自然顺序对指定 List 集合元素按升序排序  
**sort(List，Comparator)**：根据指定的 Comparator 产生的顺序对 List 集合元素进行排序  
**swap(List，int， int)**：将指定 list 集合中的 i 处元素和 j 处元素进行交换  
  
Object max(Collection)：根据元素的自然顺序，返回给定集合中的最大元素  
Object max(Collection，Comparator)：根据 Comparator 指定的顺序，返回给定集合中的最大元素  
Object min(Collection)  
Object min(Collection，Comparator)  
int frequency(Collection，Object)：返回指定集合中指定元素的出现次数  
void copy(List dest,List src)：将src中的内容复制到dest中  
boolean replaceAll(List list，Object oldVal，Object newVal)：使用新值替换 List 对象的所有旧值