## 问题

（1）TreeSet真的是使用TreeMap来存储元素的吗?

（2）TreeSet是有序的吗?

（3）TreeSet和LinkedHashSet有何不同?

## 简介

TreeSet底层是采用TreeMap实现的一种Set，所以它是有序的，同样也是非线程安全的。

## 源码分析

经过前面我们学习HashSet和LinkedHashSet，基本上已经掌握了Set实现的套路了。

所以，也不废话了，直接上源码：

```
1.  package java.util;
2.
3.  // TreeSet实现了NavigableSet接口，所以它是有序的
4.  public class TreeSet<E> extends AbstractSet<E>
5.      implements NavigableSet<E>, Cloneable, java.io.Serializable
6.  {
7.      // 元素存储在NavigableMap中
8.      // 注意它不一定就是TreeMap
9.      private transient NavigableMap<E,Object> m;
10.
11.     // 虚拟元素，用来作为value存储在map中
12.     private static final Object PRESENT = new Object();
13.
14.     // 直接使用传进来的NavigableMap存储元素
15.     // 这里不是深拷贝,如果外面的map有增删元素也会反映到这里
16.     // 而且，这个方法不是public的，说明只能给同包使用
17.     TreeSet(NavigableMap<E,Object> m) {
18.         this.m = m;
19.     }
20.
21.     // 使用TreeMap初始化
22.     public TreeSet() {
23.         this(new TreeMap<E,Object>());
24.     }
25.
26.     // 使用带comparator的TreeMap初始化
27.     public TreeSet(Comparator<? super E> comparator) {
28.         this(new TreeMap<>(comparator));
29.     }
30.
31.     // 将集合c中的所有元素添加的TreeSet中
32.     public TreeSet(Collection<? extends E> c) {
33.         this();
34.         addAll(c);
35.     }
36.
37.     // 将SortedSet中的所有元素添加到TreeSet中
38.     public TreeSet(SortedSet<E> s) {
39.         this(s.comparator());
40.         addAll(s);
41.     }
42.
```

```
        // 迭代器
        public Iterator<E> iterator() {
            return m.navigableKeySet().iterator();
        }

        // 逆序迭代器
        public Iterator<E> descendingIterator() {
            return m.descendingKeySet().iterator();
        }

        // 以逆序返回一个新的TreeSet
        public NavigableSet<E> descendingSet() {
            return new TreeSet<>(m.descendingMap());
        }

        // 元素个数
        public int size() {
            return m.size();
        }

        // 判断是否为空
        public boolean isEmpty() {
            return m.isEmpty();
        }

        // 判断是否包含某元素
        public boolean contains(Object o) {
            return m.containsKey(o);
        }

        // 添加元素，调用map的put()方法，value为PRESENT
        public boolean add(E e) {
            return m.put(e, PRESENT)==null;
        }

        // 删除元素
        public boolean remove(Object o) {
            return m.remove(o)==PRESENT;
        }

        // 清空所有元素
        public void clear() {
            m.clear();
        }

        // 添加集合c中的所有元素
        public  boolean addAll(Collection<? extends E> c) {
            // 满足一定条件时直接调用TreeMap的addAllForTreeSet()方法添加元素
            if (m.size()==0 && c.size() > 0 &&
                c instanceof SortedSet &&
                m instanceof TreeMap) {
                SortedSet<? extends E> set = (SortedSet<? extends E>) c;
                TreeMap<E,Object> map = (TreeMap<E, Object>) m;
                Comparator<?> cc = set.comparator();
                Comparator<? super E> mc = map.comparator();
                if (cc==mc || (cc != null && cc.equals(mc))) {
                    map.addAllForTreeSet(set, PRESENT);
                    return true;
                }
            }
            // 不满足上述条件，调用父类的addAll()通过遍历的方式一个一个地添加元素
            return super.addAll(c);
        }

        // 子set（NavigableSet中的方法）
```

截图(Alt + A)

```
108.         public NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
109.                                       E toElement,   boolean toInclusive) {
110.             return new TreeSet<>(m.subMap(fromElement, fromInclusive,
111.                                          toElement,   toInclusive));
112.         }
113.
114.         // 头set（NavigableSet中的方法）
115.         public NavigableSet<E> headSet(E toElement, boolean inclusive) {
116.             return new TreeSet<>(m.headMap(toElement, inclusive));
117.         }
118.
119.         // 尾set（NavigableSet中的方法）
120.         public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {
121.             return new TreeSet<>(m.tailMap(fromElement, inclusive));
122.         }
123.
124.         // 子set（SortedSet接口中的方法）
125.         public SortedSet<E> subSet(E fromElement, E toElement) {
126.             return subSet(fromElement, true, toElement, false);
127.         }
128.
129.         // 头set（SortedSet接口中的方法）
130.         public SortedSet<E> headSet(E toElement) {
131.             return headSet(toElement, false);
132.         }
133.
134.         // 尾set（SortedSet接口中的方法）
135.         public SortedSet<E> tailSet(E fromElement) {
136.             return tailSet(fromElement, true);
137.         }
138.
139.         // 比较器
140.         public Comparator<? super E> comparator() {
```

```
141.             return m.comparator();
142.         }
143.
144.         // 返回最小的元素
145.         public E first() {
146.             return m.firstKey();
147.         }
148.
149.         // 返回最大的元素
150.         public E last() {
151.             return m.lastKey();
152.         }
153.
154.         // 返回小于e的最大的元素
155.         public E lower(E e) {
156.             return m.lowerKey(e);
157.         }
158.
159.         // 返回小于等于e的最大的元素
160.         public E floor(E e) {
161.             return m.floorKey(e);
162.         }
163.
164.         // 返回大于等于e的最小的元素
165.         public E ceiling(E e) {
166.             return m.ceilingKey(e);
167.         }
168.
169.         // 返回大于e的最小的元素
170.         public E higher(E e) {
171.             return m.higherKey(e);
172.         }
```

```
173.
174.        // 弹出最小的元素
175.        public E pollFirst() {
176.            Map.Entry<E,?> e = m.pollFirstEntry();
177.            return (e == null) ? null : e.getKey();
178.        }
179.
180.        public E pollLast() {
181.            Map.Entry<E,?> e = m.pollLastEntry();
182.            return (e == null) ? null : e.getKey();
183.        }
184.
185.        // 克隆方法
186.        @SuppressWarnings("unchecked")
187.        public Object clone() {
188.            TreeSet<E> clone;
189.            try {
190.                clone = (TreeSet<E>) super.clone();
191.            } catch (CloneNotSupportedException e) {
192.                throw new InternalError(e);
193.            }
194.
195.            clone.m = new TreeMap<>(m);
196.            return clone;
197.        }
198.
199.        // 序列化写出方法
200.        private void writeObject(java.io.ObjectOutputStream s)
201.            throws java.io.IOException {
202.            // Write out any hidden stuff
203.            s.defaultWriteObject();
204.
```

```
205.            // Write out Comparator
206.            s.writeObject(m.comparator());
207.
208.            // Write out size
209.            s.writeInt(m.size());
210.
211.            // Write out all elements in the proper order.
212.            for (E e : m.keySet())
213.                s.writeObject(e);
214.        }
215.
216.        // 序列化写入方法
217.        private void readObject(java.io.ObjectInputStream s)
218.            throws java.io.IOException, ClassNotFoundException {
219.            // Read in any hidden stuff
220.            s.defaultReadObject();
221.
222.            // Read in Comparator
223.            @SuppressWarnings("unchecked")
224.                Comparator<? super E> c = (Comparator<? super E>) s.readObject();
225.
226.            // Create backing TreeMap
227.            TreeMap<E,Object> tm = new TreeMap<>(c);
228.            m = tm;
229.
230.            // Read in size
231.            int size = s.readInt();
232.
233.            tm.readTreeSet(size, s, PRESENT);
234.        }
235.
236.        // 可分割的迭代器
237.        public Spliterator<E> spliterator() {
```

```
            return TreeMap.keySpliteratorFor(m);
        }

        // 序列化id
        private static final long serialVersionUID = -2479143000061671589L;
    }

```

源码比较简单，基本都是调用map相应的方法。

## 总结

（1）TreeSet底层使用NavigableMap存储元素；

（2）TreeSet是有序的；

（3）TreeSet是非线程安全的；

（4）TreeSet实现了NavigableSet接口，而NavigableSet继承自SortedSet接口；

（5）TreeSet实现了SortedSet接口；（彤哥年轻的时候面试被问过TreeSet和SortedSet的区别^^）

## 彩蛋

（1）通过之前的学习，我们知道TreeSet和LinkedHashSet都是有序的，那它们有何不同？

LinkedHashSet并没有实现SortedSet接口，它的有序性主要依赖于LinkedHashMap的有序性，所以它的有序性是指按照插入顺序保证的有序性；

而TreeSet实现了SortedSet接口，它的有序性主要依赖于NavigableMap的有序性，而NavigableMap又继承自SortedMap，这个接口的有序性是指按照key的自然排序保证的有序性，而key的自然排序又有两种实现方式，一种是key实现Comparable接口，一种是构造方法传入Comparator比较器。

（2）TreeSet里面真的是使用TreeMap来存储元素的吗？

通过源码分析我们知道TreeSet里面实际上是使用的NavigableMap来存储元素，虽然大部分时候这个map确实是TreeMap，但不是所有时候都是TreeMap。

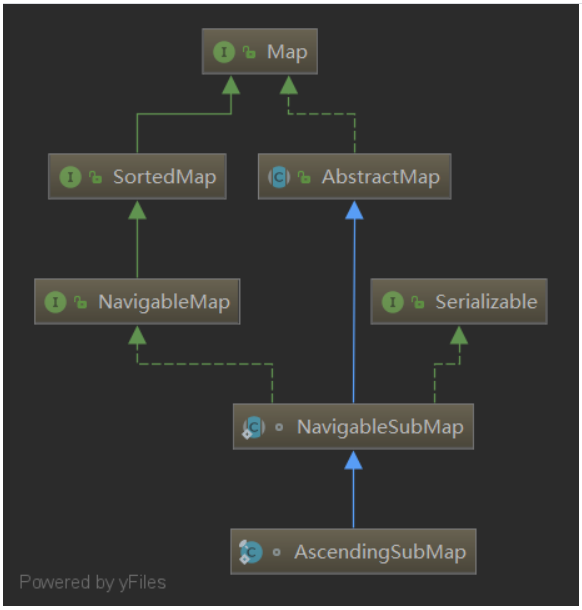
因为有一个构造方法是 `TreeSet(NavigableMap<E,Object> m)` ，而且这是一个非public方法，通过调用关系我们可以发现这个构造方法都是在自己类中使用的，比如下面这个：

```
        public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {
            return new TreeSet<>(m.tailMap(fromElement, inclusive));
        }

```

而这个m我们姑且认为它是TreeMap，也就是调用TreeMap的tailMap()方法：

```
        public NavigableMap<K,V> tailMap(K fromKey, boolean inclusive) {
            return new AscendingSubMap<>(this,
                                         false, fromKey, inclusive,
                                         true,  null,    true);
        }

```

可以看到，返回的是AscendingSubMap对象，这个类的继承链是怎么样的呢？




可以看到，这个类并没有继承TreeMap，不过通过源码分析也可以看出来这个类是组合了TreeMap，也算和TreeMap有点关系，只是不是继承关系。

所以，TreeSet的底层不完全是使用TreeMap来实现的，更准确地说，应该是NavigableMap。