问题

（1）ConcurrentSkipListSet的底层是ConcurrentSkipListMap吗?

（2）ConcurrentSkipListSet是线程安全的吗?

（3）ConcurrentSkipListSet是有序的吗?

（4）ConcurrentSkipListSet和之前讲的Set有何不同?

简介

ConcurrentSkipListSet底层是通过ConcurrentNavigableMap来实现的，它是一个有序的线程安全的集合。

源码分析

它的源码比较简单，跟通过Map实现的Set基本是一致，只是多了一些取最近的元素的方法。

为了保持专栏的完整性，我还是贴一下源码，最后会对Set的整个家族作一个对比，有兴趣的可以直接拉到最下面。

```
// 实现了NavigableSet接口，并没有所谓的ConcurrentNavigableSet接口
public class ConcurrentSkipListSet<E>
    extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable {

    private static final long serialVersionUID = -2479143111061671589L;

    // 存储使用的map
    private final ConcurrentNavigableMap<E,Object> m;

    // 初始化
    public ConcurrentSkipListSet() {
        m = new ConcurrentSkipListMap<E,Object>();
    }

    // 传入比较器
    public ConcurrentSkipListSet(Comparator<? super E> comparator) {
        m = new ConcurrentSkipListMap<E,Object>(comparator);
    }

    // 使用ConcurrentSkipListMap初始化map
    // 并将集合c中所有元素放入到map中
    public ConcurrentSkipListSet(Collection<? extends E> c) {
        m = new ConcurrentSkipListMap<E,Object>();
        addAll(c);
    }

    // 使用ConcurrentSkipListMap初始化map
    // 并将有序Set中所有元素放入到map中
    public ConcurrentSkipListSet(SortedSet<E> s) {
        m = new ConcurrentSkipListMap<E,Object>(s.comparator());
        addAll(s);
    }

    // ConcurrentSkipListSet类内部返回子set时使用的
    ConcurrentSkipListSet(ConcurrentNavigableMap<E,Object> m) {
        this.m = m;
    }

    // 克隆方法
```

```
41.         public ConcurrentSkipListSet<E> clone() {
42.             try {
43.                 @SuppressWarnings("unchecked")
44.                 ConcurrentSkipListSet<E> clone =
45.                     (ConcurrentSkipListSet<E>) super.clone();
46.                 clone.setMap(new ConcurrentSkipListMap<E,Object>(m));
47.                 return clone;
48.             } catch (CloneNotSupportedException e) {
49.                 throw new InternalError();
50.             }
51.         }
52.
53.         /* ---------------- Set operations -------------- */
54.         // 返回元素个数
55.         public int size() {
56.             return m.size();
57.         }
58.
59.         // 检查是否为空
60.         public boolean isEmpty() {
61.             return m.isEmpty();
62.         }
63.
64.         // 检查是否包含某个元素
65.         public boolean contains(Object o) {
66.             return m.containsKey(o);
67.         }
68.
69.         // 添加一个元素
70.         // 调用map的putIfAbsent()方法
71.         public boolean add(E e) {
72.             return m.putIfAbsent(e, Boolean.TRUE) == null;
73.         }
```

```
74.
75.         // 移除一个元素
76.         public boolean remove(Object o) {
77.             return m.remove(o, Boolean.TRUE);
78.         }
79.
80.         // 清空所有元素
81.         public void clear() {
82.             m.clear();
83.         }
84.
85.         // 迭代器
86.         public Iterator<E> iterator() {
87.             return m.navigableKeySet().iterator();
88.         }
89.
90.         // 降序迭代器
91.         public Iterator<E> descendingIterator() {
92.             return m.descendingKeySet().iterator();
93.         }
94.
95.
96.         /* ---------------- AbstractSet Overrides -------------- */
97.         // 比较相等方法
98.         public boolean equals(Object o) {
99.             // Override AbstractSet version to avoid calling size()
100.            if (o == this)
101.                return true;
102.            if (!(o instanceof Set))
103.                return false;
104.            Collection<?> c = (Collection<?>) o;
105.            try {
```

```
                // 这里是通过两次两层for循环来比较
                // 这里是有很大优化空间的，参考上篇文章CopyOnWriteArraySet中的彩蛋
                return containsAll(c) && c.containsAll(this);
            } catch (ClassCastException unused) {
                return false;
            } catch (NullPointerException unused) {
                return false;
            }
        }

        // 移除集合c中所有元素
        public boolean removeAll(Collection<?> c) {
            // Override AbstractSet version to avoid unnecessary call to size()
            boolean modified = false;
            for (Object e : c)
                if (remove(e))
                    modified = true;
            return modified;
        }

        /* ---------------- Relational operations -------------- */

        // 小于e的最大元素
        public E lower(E e) {
            return m.lowerKey(e);
        }

        // 小于等于e的最大元素
        public E floor(E e) {
            return m.floorKey(e);
        }

        // 大于等于e的最小元素
        public E ceiling(E e) {
            return m.ceilingKey(e);
        }

        // 大于e的最小元素
        public E higher(E e) {
            return m.higherKey(e);
        }

        // 弹出最小的元素
        public E pollFirst() {
            Map.Entry<E,Object> e = m.pollFirstEntry();
            return (e == null) ? null : e.getKey();
        }

        // 弹出最大的元素
        public E pollLast() {
            Map.Entry<E,Object> e = m.pollLastEntry();
            return (e == null) ? null : e.getKey();
        }


        /* ---------------- SortedSet operations -------------- */

        // 取比较器
        public Comparator<? super E> comparator() {
            return m.comparator();
        }

        // 最小的元素
        public E first() {
            return m.firstKey();
```

```
171.         }
172.
173.         // 最大的元素
174.         public E last() {
175.             return m.lastKey();
176.         }
177.
178.         // 取两个元素之间的子set
179.         public NavigableSet<E> subSet(E fromElement,
180.                                       boolean fromInclusive,
181.                                       E toElement,
182.                                       boolean toInclusive) {
183.             return new ConcurrentSkipListSet<E>
184.                 (m.subMap(fromElement, fromInclusive,
185.                           toElement,   toInclusive));
186.         }
187.
188.         // 取头子set
189.         public NavigableSet<E> headSet(E toElement, boolean inclusive) {
190.             return new ConcurrentSkipListSet<E>(m.headMap(toElement, inclusive));
191.         }
192.
193.         // 取尾子set
194.         public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {
195.             return new ConcurrentSkipListSet<E>(m.tailMap(fromElement, inclusive));
196.         }
197.
198.         // 取子set，包含from，不包含to
199.         public NavigableSet<E> subSet(E fromElement, E toElement) {
200.             return subSet(fromElement, true, toElement, false);
201.         }
202.
203.         // 取头子set，不包含to
```

```
204.         public NavigableSet<E> headSet(E toElement) {
205.             return headSet(toElement, false);
206.         }
207.
208.         // 取尾子set，包含from
209.         public NavigableSet<E> tailSet(E fromElement) {
210.             return tailSet(fromElement, true);
211.         }
212.
213.         // 降序set
214.         public NavigableSet<E> descendingSet() {
215.             return new ConcurrentSkipListSet<E>(m.descendingMap());
216.         }
217.
218.         // 可分割的迭代器
219.         @SuppressWarnings("unchecked")
220.         public Spliterator<E> spliterator() {
221.             if (m instanceof ConcurrentSkipListMap)
222.                 return ((ConcurrentSkipListMap<E,?>)m).keySpliterator();
223.             else
224.                 return (Spliterator<E>)((ConcurrentSkipListMap.SubMap<E,?>)m).keyIterator();
225.         }
226.
227.         // 原子更新map，给clone方法使用
228.         private void setMap(ConcurrentNavigableMap<E,Object> map) {
229.             UNSAFE.putObjectVolatile(this, mapOffset, map);
230.         }
231.
232.         // 原子操作相关内容
233.         private static final sun.misc.Unsafe UNSAFE;
234.         private static final long mapOffset;
235.         static {
```

```
            try {
                UNSAFE = sun.misc.Unsafe.getUnsafe();
                Class<?> k = ConcurrentSkipListSet.class;
                mapOffset = UNSAFE.objectFieldOffset
                    (k.getDeclaredField("m"));
            } catch (Exception e) {
                throw new Error(e);
            }
        }
    }

```

可以看到，ConcurrentSkipListSet基本上都是使用ConcurrentSkipListMap实现的，虽然取子set部分是使用ConcurrentSkipListMap中的内部类，但是这些内部类其实也是和ConcurrentSkipListMap相关的，它们返回ConcurrentSkipListMap的一部分数据。

另外，这里的equals()方法实现的相当敷衍，有很大的优化空间，作者这样实现，应该也是知道几乎没有人来调用equals()方法吧。

总结

（1）ConcurrentSkipListSet底层是使用ConcurrentNavigableMap实现的；

（2）ConcurrentSkipListSet有序的，基于元素的自然排序或者通过比较器确定的顺序；

（3）ConcurrentSkipListSet是线程安全的；

彩蛋

Set大汇总：

Set	有序性	线程安全	底层实现	关键接口	特点
HashSet	无	否	HashMap	无	简单
LinkedHashSet	有	否	LinkedHashMap	无	插入顺序
TreeSet	有	否	NavigableMap	NavigableSet	自然顺序
CopyOnWriteArraySet	有	是	CopyOnWriteArrayList	无	插入顺序，读写分离
ConcurrentSkipListSet	有	是	ConcurrentNavigableMap	NavigableSet	自然顺序

从中我们可以发现一些规律：

（1）除了HashSet其它Set都是有序的；

（2）实现了NavigableSet或者SortedSet接口的都是自然顺序的；

（3）使用并发安全的集合实现的Set也是并发安全的；

（4）TreeSet虽然不是全部都是使用的TreeMap实现的，但其实都是跟TreeMap相关的（TreeMap的子Map中组合了TreeMap）；

（5）ConcurrentSkipListSet虽然不是全部都是使用的ConcurrentSkipListMap实现的，但其实都是跟ConcurrentSkipListMap相关的（ConcurrentSkipListeMap的子Map中组合了ConcurrentSkipListMap）；