

# 死磕 java集合之TreeSet源码分析

71个

## 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：

```
package java.util;

// TreeSet实现了NavigableSet接口，所以它是有序的
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
{
    // 元素存储在NavigableMap中
    // 注意它不一定是TreeMap
    private transient NavigableMap<E, Object> m;

    // 虚拟元素，用来作为value存储在map中
    private static final Object PRESENT = new Object();

    // 直接使用传进来的NavigableMap存储元素
    // 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
    // 而且，这个方法不是public的，说明只能给同包使用
    TreeSet(NavigableMap<E, Object> m) {
        this.m = m;
    }
}
```

```
// 使用TreeMap初始化
public TreeSet() {
    this(new TreeMap<E, Object>());
}

// 使用带comparator的TreeMap初始化
public TreeSet(Comparator<? super E> comparator) {
    this(new TreeMap<>(comparator));
}

// 将集合c中的所有元素添加的TreeSet中
public TreeSet(Collection<? extends E> c) {
    this();
    addAll(c);
}

// 将SortedSet中的所有元素添加到TreeSet中
public TreeSet(SortedSet<E> s) {
    this(s.comparator());
    addAll(s);
}

// 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)
    public NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                                E toElement, boolean toInclusive) {
        return new TreeSet<>(m.subMap(fromElement, fromInclusive,
                                       toElement, toInclusive));
    }

    // 头set (NavigableSet中的方法)
    public NavigableSet<E> headSet(E toElement, boolean inclusive) {
        return new TreeSet<>(m.headMap(toElement, inclusive));
    }

    // 尾set (NavigableSet中的方法)
    public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {
        return new TreeSet<>(m.tailMap(fromElement, inclusive));
    }

    // 子set (SortedSet接口中的方法)
    public SortedSet<E> subSet(E fromElement, E toElement) {
        return subSet(fromElement, true, toElement, false);
    }

    // 头set (SortedSet接口中的方法)
    public SortedSet<E> headSet(E toElement) {
        return headSet(toElement, false);
    }

    // 尾set (SortedSet接口中的方法)
    public SortedSet<E> tailSet(E fromElement) {
        return tailSet(fromElement, true);
    }

```

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

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

// 返回最大的元素
public E last() {
    return m.lastKey();
}

// 返回小于e的最大的元素
public E lower(E e) {
    return m.lowerKey(e);
}

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

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

// 返回大于e的最小的元素
public E higher(E e) {
    return m.higherKey(e);
}

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

public E pollLast() {
    Map.Entry<E,?> e = m.pollLastEntry();
    return (e == null) ? null : e.getKey();
}

// 克隆方法
@SuppressWarnings("unchecked")
public Object clone() {
    TreeSet<E> clone;
    try {
        clone = (TreeSet<E>) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }

    clone.m = new TreeMap<>(m);
    return clone;
}
```

```
}

// 序列化写出方法
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out any hidden stuff
    s.defaultWriteObject();

    // Write out Comparator
    s.writeObject(m.comparator());

    // Write out size
    s.writeInt(m.size());

    // Write out all elements in the proper order.
    for (E e : m.keySet())
        s.writeObject(e);
}

// 序列化写入方法
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in any hidden stuff
    s.defaultReadObject();

    // Read in Comparator
    @SuppressWarnings("unchecked")
    Comparator<? super E> c = (Comparator<? super E>) s.readObject();

    // Create backing TreeMap
    TreeMap<E, Object> tm = new TreeMap<>(c);
    m = tm;

    // Read in size
    int size = s.readInt();

    tm.readTreeSet(size, s, PRESENT);
}

// 可分割的迭代器
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。

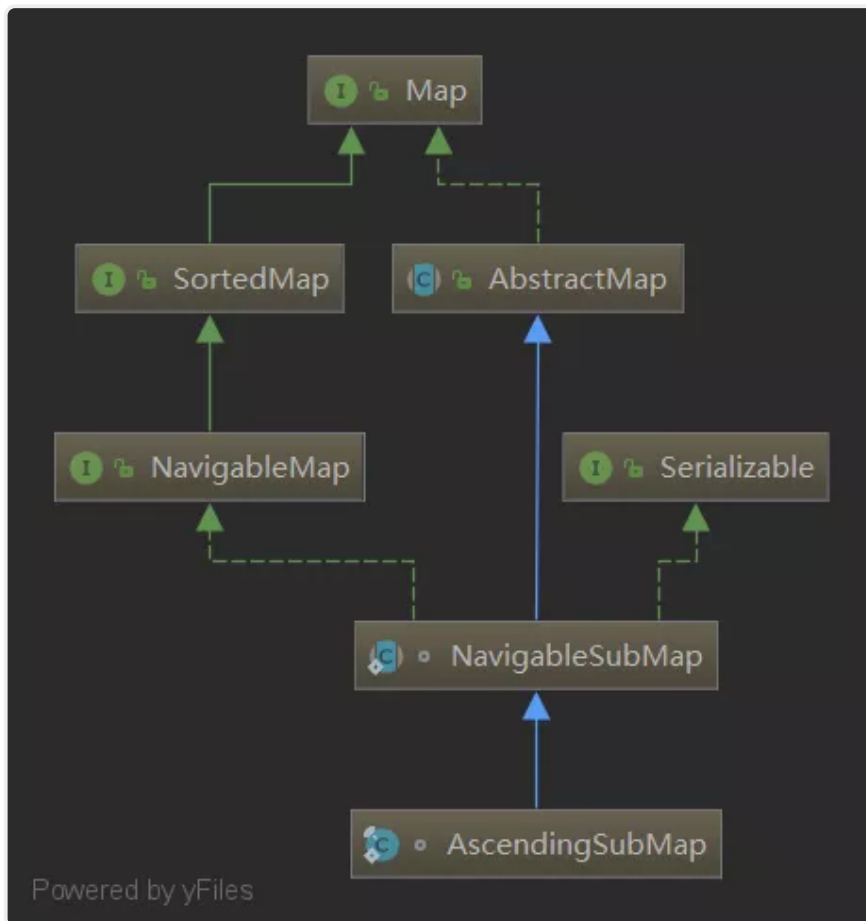
因为有一个构造方法是 `TreeSet ( NavigableMap < E , Object > m )`，而且这是一个非public方法，通过调用关系我们可以发现这个构造方法都是在自己类中使用的，比如下面这个：

```
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。

## TreeSet的基本使用

### 1、添加到TreeSet的数据必须是同一个类型的数据

```
Set set=new TreeSet();
set.add(123);
set.add("name");
直接抛异常 java.lang.Integer cannot be cast to java.lang.String
```

### 2、默认遍历元素

```
Set set=new TreeSet();
set.add(123);
set.add(-23);
set.add(466);
Iterator iterator = set.iterator();
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
-23
123
466
```

如果添加的对象元素没有重写Comparable则报错

```
Set set=new TreeSet();
set.add(new Person(12,"tom"));
set.add(new Person(22,"jerry"));
set.add(new Person(1,"bob"));
Iterator iterator = set.iterator();
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
抛异常 Person cannot be cast to java.lang.Comparable
```

**第一种排序方式：Set内对象Person自然排序：Person对象实现Comparable接口**

```
class Person implements Comparable{
    //先按照姓名从小到大排序后按照年龄从小到大排序。二级排序
    @Override
    public int compareTo(Object o) {
        if (o instanceof Person){
            Person p=(Person)o;
            int res= this.name.compareTo(p.name);
            if (res!=0) { //第一级排序按照姓名从小到大
                return res;
            } else { //二级排序，姓名相同按照年龄排序
                return Integer.compare(this.age,((Person) o).age);
            }
        } else {
            throw new RuntimeException("类型错误");
        }
    }
}
```

Comparable接口中只有一个方法就是compareTo方法，比较两个对象大小，

**返回：**

负整数、零或正整数，根据此对象是小于、等于还是大于指定对象

**例如Integer中的compare方法**

```
public static int compare(int x, int y) {
    return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
```

TreeSet中添加元素是否成功不再是按照equals方法判断两个元素是否相同，而是根据compareTo方法判断，如果compareTo返回0表示添加元素相同，则添加失败。



第二种排序方式：定制排序

TreeSet的构造器中传入Comparator比较器

```
Comparator comparator=new Comparator() {  
    @Override  
    public int compare(Object o1, Object o2) {  
        if (o1 instanceof Person && o2 instanceof Person){  
            Person p1=(Person)o1;  
            Person p2=(Person)o2;  
            //按照年龄从小到大排序  
            return Integer.compare(p1.getAge(),p2.getAge());  
        }else {  
            throw new RuntimeException("类型错误");  
        }  
    }  
};  
Set set=new TreeSet(comparator);  
set.add(new Person(12,"tom"));  
set.add(new Person(22,"jerry"));  
set.add(new Person(1,"bob"));  
Iterator iterator = set.iterator();  
while (iterator.hasNext()){  
    System.out.println(iterator.next());  
}
```

TreeSet元素排序总结：

自然排序中：比较两个对象是否相等的标准：compareTo()返回0，不再是equals()

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



当前位置: HollisChuang's Blog (<https://www.hollischuang.com>) > Java (<https://www.hollischuang.com/archives/category/java>) > 正文

## 简单介绍Java中Comparable和Comparator (<https://www.hollischuang.com/archives/1292>)

2016-03-15 来源: Comparable vs. Comparator in Java (<http://www.programcreek.com/2011/12/examples-to-demonstrate-comparable-vs-comparator-in-java/>) 分类: Java (<https://www.hollischuang.com/archives/category/java>) 阅读(11994) 评论(3)

[GitHub 19k Star 的Java工程师成神之路，不来了解一下吗！](https://github.com/hollischuang/toBeTopJavaer)  
(<https://github.com/hollischuang/toBeTopJavaer>)

Comparable 和 Comparator 是Java核心API提供的两个接口，从它们的名字中，我们大致可以猜到它们用来做对象之间的比较的。但它们到底怎么用，它们之间又有哪些差别呢？下面有两个例子可以很好的回答这个问题。下面的例子用来比较HDTV的大小。看完下面的代码，相信对于如何使用 Comparable 和 Comparator 会有一个更加清晰的认识。

### Comparable

一个实现了 Comparable 接口的类，可以让其自身的对象和其他对象进行比较。也就是说，同一个类的两个对象之间要想比较，**对应的类就要实现 Comparable 接口**，并实现 compareTo() 方法，代码如下：

```
class HDTV implements Comparable<HDTV> {  
    private int size;  
    private String brand;  
  
    public HDTV(int size, String brand) {  
        this.size = size;  
        this.brand = brand;  
    }  
  
    public int getSize() {  
        return size;  
    }  
  
    public void setSize(int size) {  
        this.size = size;  
    }  
  
    public String getBrand() {  
        return brand;  
    }  
  
    public void setBrand(String brand) {  
        this.brand = brand;  
    }  
  
    @Override  
    public int compareTo(HDTV tv) {  
  
        if (this.getSize() > tv.getSize())  
            return 1;  
    }  
}
```




输出结果：

Sony is better.

## Comparator

在一些情况下，你不希望修改一个原有的类，但是你还想让他可以比较，Comparator 接口可以实现这样的功能。通过使用 Comparator 接口，你可以针对其中特定的属性/字段来进行比较。比如，当我们要比较两个人的时候，我可能通过年龄比较、也可能通过身高比较。这种情况使用 Comparable 就无法实现（因为要实现 Comparable 接口，其中的 compareTo 方法只能有一个，无法实现多种比较）。却可以定义多个实现Comparator的子类，每个子类定义一种Person对象的比较方法，第一个子类按照年龄比较，第二个子类按照身高比较。如果Person实现Comparable方法，则只能一种比较方法

通过实现 Comparator 接口同样要重写一个方法：compare()。接下来的例子就通过这种方式来比较HDTV的大小。其实 Comparator 通常用于排序。Java中的 Collections 和 Arrays 中都包含排序的 sort 方法，该方法可以接收一个 Comparator 的实例（比较器）来进行排序。



```
}  
}  
  
class SizeComparator implements Comparator<HDTV> {  
    @Override  
    public int compare(HDTV tv1, HDTV tv2) {  
        int tv1Size = tv1.getSize();  
        int tv2Size = tv2.getSize();  
  
        if (tv1Size > tv2Size) {  
            return 1;  
        } else if (tv1Size < tv2Size) {  
            return -1;  
        } else {  
            return 0;  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        HDTV tv1 = new HDTV(55, "Samsung");  
        HDTV tv2 = new HDTV(60, "Sony");  
        HDTV tv3 = new HDTV(42, "Panasonic");  
  
        ArrayList<HDTV> al = new ArrayList<HDTV>();  
        al.add(tv1);  
        al.add(tv2);  
        al.add(tv3);  
  
        Collections.sort(al, new SizeComparator());  
    }  
}
```

输出结果：

```
Panasonic  
Samsung  
Sony
```

以上代码就实现了通过自定义一个比较器（Comparator）来实现对一个列表进行排序。

我们也经常会使用 `Collections.reverseOrder()` 来获取一个倒序的 Comparator。例如：

```
ArrayList<Integer> al = new ArrayList<Integer>();  
al.add(3);  
al.add(1);  
al.add(2);  
System.out.println(al);  
Collections.sort(al);  
System.out.println(al);  
  
Comparator<Integer> comparator = Collections.reverseOrder();  
Collections.sort(al,comparator);  
System.out.println(al);
```



输出结果：

```
[3,1,2]  
[1,2,3]  
[3,2,1]
```

## 如何选择

简单来说，一个类如果实现 Comparable 接口，那么他就具有了可比较性，意思就是说它的实例之间相互直接可以进行比较。

通常在两种情况下会定义一个实现 Comparator 类。

- 1、如上面的例子一样，可以把一个 Comparator 的子类传递给 Collections.sort() 、 Arrays.sort() 等方法，用于自定义排序规则。
- 2、用于初始化特定的数据结构。常见的有可排序的Set（TreeSet）和可排序的Map（TreeMap）

下面通过这两种方式分别创建 TreeSet 。

## 使用 Comparator 创建 TreeSet





```
class Dog {  
    int size;  
  
    Dog(int s) {  
        size = s;  
    }  
}  
  
class SizeComparator implements Comparator<Dog> {  
    @Override  
    public int compare(Dog d1, Dog d2) {  
        return d1.size - d2.size;  
    }  
}  
  
public class ImpComparable {  
    public static void main(String[] args) {  
        TreeSet<Dog> d = new TreeSet<Dog>(new SizeComparator()); // pass comparator  
        d.add(new Dog(1));  
        d.add(new Dog(2));  
        d.add(new Dog(1));  
    }  
}
```

这里使用的就是 Comparator 的第二种用法，定义一个 Comparator 的子类，重写 compare 方法。然后在定义 HashSet 的时候，把这个类的实例传递给其构造函数。这样，再使用 add 方法向 HashSet 中增加元素的时候，就会按照刚刚定义的那个比较器的逻辑进行排序。

## 使用 Comparable 创建 TreeSet

```
class Dog implements Comparable<Dog>{  
    int size;  
  
    Dog(int s) {  
        size = s;  
    }  
  
    @Override  
    public int compareTo(Dog o) {  
        return o.size - this.size;  
    }  
}  
  
public class ImpComparable {  
    public static void main(String[] args) {  
        TreeSet<Dog> d = new TreeSet<Dog>();  
        d.add(new Dog(1));  
        d.add(new Dog(2));  
        d.add(new Dog(1));  
    }  
}
```



这里，定义 `TreeSet` 的时候并没有传入一个比较器。但是使用 `add` 方法向 `HashSet` 中增加的对象是一个实现了 `Comparable` 的类的实例。所以，也能实现排序功能。

## 两种比较器区别

**1. Comparable** 是能被比较的对象(Person)自己实现 `Comparable` 接口的 `compareTo` 方法，并且 Person 对象只能有一种比较规则

**2. Comparator** 比较器需要手动实现子类，并且重写 `compar(Obj1, Obj2)`，可以不破坏被比较对象的源码(Person)，并且可以创建多个 `Comparator` 子类，每个子类定义一种比较规则，第一个比较器按照年龄排序，第二个子类按照身高排序。