芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-12-13
JDK

精尽 JDK 源码解析 —— 集合（六）TreeMap

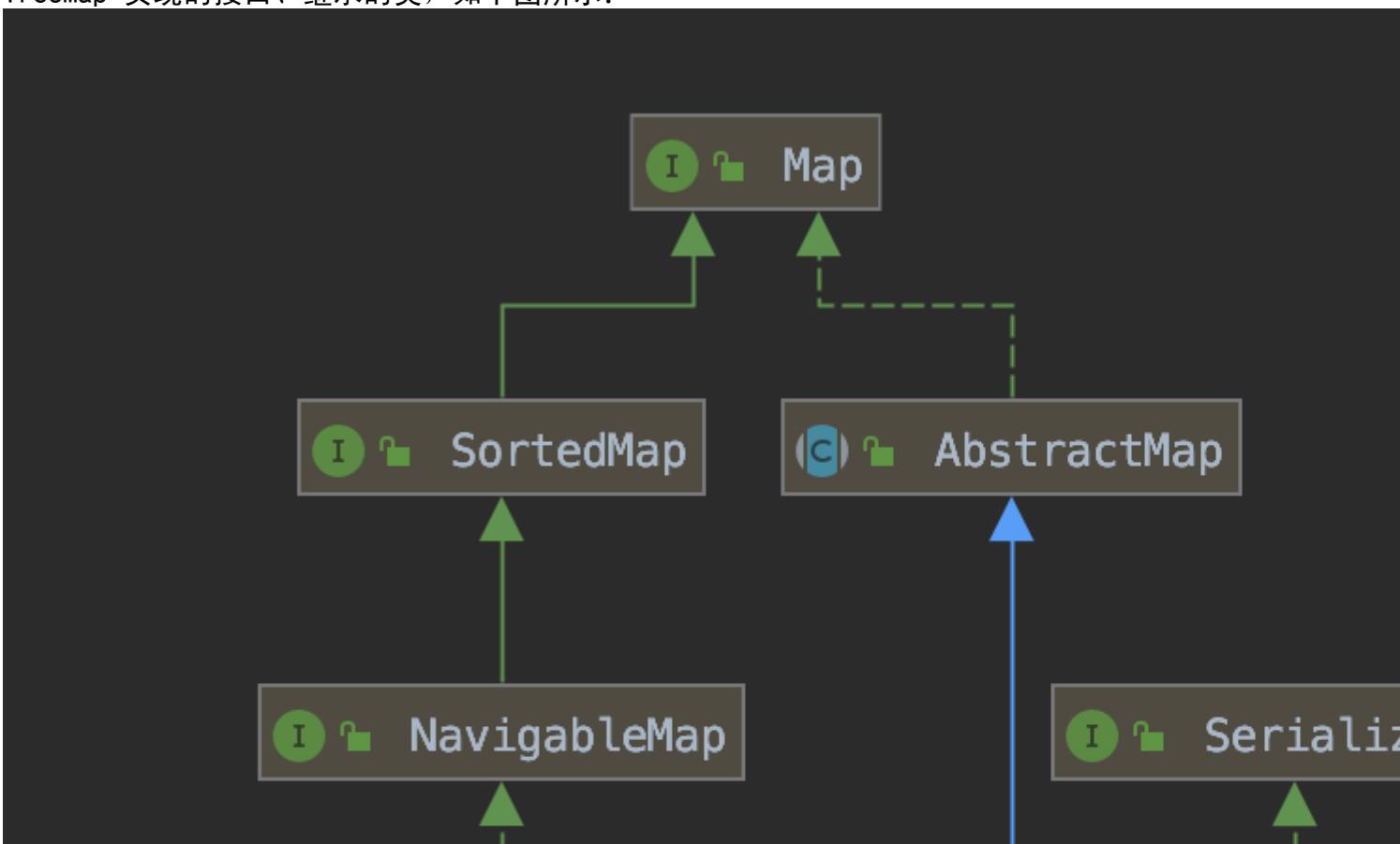
1. 概述

在 《精尽 JDK 源码解析 —— 集合（四）哈希表 LinkedHashMap》 中，我们提到了两种有序 Map 的选择。一种是 LinkedHashMap ，以前在该文进行了详细解析，而本文，我们开始 TreeMap 之旅，按照 key 的顺序的 Map 实现类。

在开始之前，芳芳捉摸了下，什么业务场景下适合使用 TreeMap 呢？发现好像基本没有，嘿嘿。然后，我又翻了自己团队的几个项目，发现唯一在使用的，就是在 签名生成算法 时，要求按照请求参数的 key 排序，然后拼接后加密。如果胖友有 TreeMap 的使用场景，请一定在星球给留言，芳芳可以补充到本文。

2. 类图

TreeMap 实现的接口、继承的类，如下图所示：



实现 java.util.Map 接口，并继承 java.util.AbstractMap 抽像类。
直接实现 java.util.NavigableMap 接口，间接实现 java.util.NavigableMap 接口。关于这两接口的定义的操作，已经添加注释，胖友可以直接点击查看。因为 SortedMap 有很多雷同的寻找最接近 key 的操作，这里简单总结下：
- lower ：小于 ；floor ：小于等于
- higher ：大于；ceiling ：大于等于

实现 java.io.Serializable 接口。
实现 java.lang.Cloneable 接口。

3. 属性

在开始看 TreeMap 的具体属性之前，我们先来简单说说 TreeMap 的实现原理。

在 《精尽 JDK 源码解析 —— 集合（三）哈希表 HashMap》 文章中，我们提到，HashMap 的数组，每个桶的链表在元素过多的情况下，会转换成红黑树。而 TreeMap 也是基于红黑树实现的，并且只是一棵红黑树。所以 TreeMap 可以理解成 HashMap 数组中的一个转换成红黑树的桶。

为什么 TreeMap 会采用红黑树实现呢？我们来看一段红黑树的定义：

> FROM 《维基百科 —— 红黑树》
>
> 红黑树（英语：Red－black tree）是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。
>
> 它在 1972 年由鲁道夫·贝尔发明，被称为”对称二叉B树“，它现代的名字源于 Leo J. Guibas 和 Robert Sedgewick 于 [1978年](https://zh.wikipedia.org/wiki/1978年) 写的一篇论文。红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在 `logN` 时间内完成查找，插入和删除，这里的 `N` 是树中元素的数目。
>
> 有序性：红黑树是一种二叉查找树，父节点的 key 小于左子节点的 key ，大于右子节点的 key 。这样，就完成了 TreeMap 的有序的特性。
> 高性能：红黑树会进行自平衡，避免树的高度过高，导致查找性能下滑。这样，红黑树能够提供 `logN` 的时间复杂度。
>
> 芳芳：绝大多数情况下，包括面试，我们无需了解红黑树是怎么实现的，甚至原理是什么。只要知道，红黑树的上述概念，和时间复杂度即可。
>
> 所以，本文我们不会涉及红黑树的自平衡的内容。如果感兴趣的胖友，可以自己阅读如下的文章：
>
> 《教你初步了解红黑树》
> 《史上最清晰的红黑树讲解（上）》
> 《史上最清晰的红黑树讲解（下）》

下面，让我们来看看 TreeMap 的属性。代码如下：

```
// TreeMap.java

/**
 * key 排序器
 */
private final Comparator<? super K> comparator;
```

```
/**
 * 红黑树的根节点
 */
private transient Entry<K,V> root;

/**
 * key-value 键值对数量
 */
private transient int size = 0;

/**
 * 修改次数
 */
private transient int modCount = 0;
```

comparator 属性，key 排序器。通过该属性，可以自定义 key 的排序规则。如果未设置，则使用 key 类型自己的排序。

root 属性，红黑树的根节点。其中，Entry 是 TreeMap 的内部静态类，代码如下：

```
// TreeMap.java

/**
 * 颜色 - 红色
 */
private static final boolean RED   = false;
/**
 * 颜色 - 黑色
 */
private static final boolean BLACK = true;

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

    /**
     * key 键
     */
    K key;
    /**
     * value 值
     */
    V value;
    /**
     * 左子节点
     */
    Entry<K,V> left;
    /**
     * 右子节点
     */
    Entry<K,V> right;
    /**
     * 父节点
     */
    Entry<K,V> parent;
    /**
     * 颜色
     */
    boolean color = BLACK;
```

```
        // ... 省略一些

    }
```

4. 构造方法

TreeMap 一共有四个构造方法，我们分别来看看。

① #TreeMap()

```
// TreeMap.java

public TreeMap() {
    comparator = null;
}
```

默认构造方法，不使用自定义排序，所以此时 comparator 为空。

② #TreeMap(Comparator<? super K> comparator)

```
// TreeMap.java

public TreeMap(Comparator<? super K> comparator) {
    this.comparator = comparator;
}
```

可传入 comparator 参数，自定义 key 的排序规则。

③ #TreeMap(SortedMap<K, ? extends V> m)

```
// TreeMap.java

public TreeMap(SortedMap<K, ? extends V> m) {
    // <1> 设置 comparator 属性
    comparator = m.comparator();
    try {
        // <2> 使用 m 构造红黑树
        buildFromSorted(m.size(), m.entrySet().iterator(), null, null);
    } catch (java.io.IOException | ClassNotFoundException cannotHappen) {
    }
}
```

传入已经排序的 m ，然后构建出 TreeMap 。
<1> 处，使用 m 的 key 排序器，设置到 comparator 属性。
<2> 处，调用 #buildFromSorted(int size, Iterator<?> it, ObjectInputStream str, V defaultVal) 方法，使用 m 构造红黑树。因为 m 是 SortedMap 类型，所以天然有序，所以可以基于 m 的中间为红黑树的根节点，m 的左边为左子树，m 的右边为右子树。 胖友，发挥下自己的想象力。

#buildFromSorted(int size, Iterator<?> it, ObjectInputStream str, V defaultVal) ，代码如下：

```
// TreeMap.java

private void buildFromSorted(int size, Iterator<?> it,
                             java.io.ObjectInputStream str,
                             V defaultVal)
    throws  java.io.IOException, ClassNotFoundException {
    // <1> 设置 key-value 键值对的数量
    this.size = size;
    // <2> computeRedLevel(size) 方法，计算红黑树的高度
    // <3> 使用 m 构造红黑树，返回根节点
    root = buildFromSorted(0, 0, size - 1, computeRedLevel(size),
                           it, str, defaultVal);
}
```

<1> 处，设置 key-value 键值对的数量到 size 。

<2> 处，调用 #computeRedLevel(int size) 方法，计算红黑树的高度。代码如下：

```
// TreeMap.java

private static int computeRedLevel(int size) {
    return 31 - Integer.numberOfLeadingZeros(size + 1);
}
```

<3> 处，调用 #buildFromSorted(int level, int lo, int hi, int redLevel, Iterator<?> it, ObjectInputStream str, V defaultVal) 方法，使用 m 构造红黑树，返回根节点。

#buildFromSorted(int level, int lo, int hi, int redLevel, Iterator<?> it, ObjectInputStream str, V defaultVal) 方法，代码如下：

```
// TreeMap.java

private final Entry<K,V> buildFromSorted(int level, int lo, int hi,
                                         int redLevel,
                                         Iterator<?> it,
                                         java.io.ObjectInputStream str,
                                         V defaultVal)
    throws  java.io.IOException, ClassNotFoundException {
    /*
     * Strategy: The root is the middlemost element. To get to it, we
     * have to first recursively construct the entire left subtree,
     * so as to grab all of its elements. We can then proceed with right
     * subtree.
     *
     * The lo and hi arguments are the minimum and maximum
     * indices to pull out of the iterator or stream for current subtree.
     * They are not actually indexed, we just proceed sequentially,
     * ensuring that items are extracted in corresponding order.
     */
    // <1.1> 递归结束
    if (hi < lo) return null;

    // <1.2> 计算中间值
    int mid = (lo + hi) >>> 1;
```

```
    // <2.1> 创建左子树
    Entry<K,V> left  = null;
    if (lo < mid)
        // <2.2> 递归左子树
        left = buildFromSorted(level + 1, lo, mid - 1, redLevel,
                               it, str, defaultVal);

    // extract key and/or value from iterator or stream
    // <3.1> 获得 key-value 键值对
    K key;
    V value;
    if (it != null) { // 使用 it 迭代器，获得下一个值
        if (defaultVal == null) {
            Map.Entry<?,?> entry = (Map.Entry<?,?>)it.next(); // 从 it 获得下一个 Entry 节点
            key = (K) entry.getKey(); // 读取 key
            value = (V) entry.getValue(); // 读取 value
        } else {
            key = (K)it.next();  // 读取 key
            value = defaultVal; // 设置 default 为 value
        }
    } else { // use stream 处理 str 流的情况
        key = (K) str.readObject(); //  从 str 读取 key 值
        value = (defaultVal != null ? defaultVal : (V) str.readObject()); // 从 str 读取 value 值
    }


    // <3.2> 创建中间节点
    Entry<K,V> middle =  new Entry<>(key, value, null);

    // color nodes in non-full bottommost level red
    // <3.3> 如果到树的最大高度，则设置为红节点
    if (level == redLevel)
        middle.color = RED;

    // <3.4> 如果左子树非空，则进行设置
    if (left != null) {
        middle.left = left; // 当前节点，设置左子树
        left.parent = middle; // 左子树，设置父节点为当前节点
    }

    // <4.1> 创建右子树
    if (mid < hi) {
        // <4.2> 递归右子树
        Entry<K,V> right = buildFromSorted(level + 1, mid + 1, hi, redLevel,
                                           it, str, defaultVal);
        // <4.3> 当前节点，设置右子树
        middle.right = right;
        // <4.3> 右子树，设置父节点为当前节点
        right.parent = middle;
    }

    // 返回当前节点
    return middle;
}
```

基于有序的 it 迭代器或者 str 输入流，将其的中间点作为根节点，其左边作为左子树，其右边作为右子树。因为是基于递归实现，所以中间点是基于 lo 和 hi 作为 it 或 str 的“数组”范围。

如果胖友有学习过数据结构与算法，这里代码的实现，就是基于 《五大常用算法之一：分治算法》 。

<1.1> 处，如果 hi 小于 lo ，说明已经超过范围，所以可以结束循环。

<1.2> 处，计算中间位置。

左子树

- <2.1> 处，处理 mid 中间位置的左边，处理左子树。
- <2.2> 处，调用 #buildFromSorted(int level, int lo, int hi, int redLevel, Iterator<?> it, ObjectInputStream str, V defaultVal) 方法，递归处理 it 或 str 的 [lo, mid - 1] 范围，创建左子树，返回该子树的根节点，赋值给 left 。

当前节点（中间节点）

- <3.1> 处，获得 key-value 键值对。分成使用 it 或 str 读取的两种情况。有一点要注意，在 defaultVal 非空的时候，使用它作为 value 。
- <3.2> 处，创建当前节点。
- <3.3> 处，如果到树的最大高度，则设置为红节点。
- <3.4> 处，如果左子树非空，则进行设置。

右子树

- <4.1> 处，处理 mid 中间位置的右边，处理右子树。
- <4.2> 处，调用 #buildFromSorted(int level, int lo, int hi, int redLevel, Iterator<?> it, ObjectInputStream str, V defaultVal) 方法，递归处理 it 或 str 的 [mid + 1, high] 范围，创建右子树，返回该子树的根节点，赋值给 right 。
- <4.3> 处，设置右子树。

返回当前节点。因为是递归，所以递归的第一层，是 TreeMap 红黑树的根节点。

④ #TreeMap(Map<? extends K, ? extends V> m)

```
// TreeMap.java

public TreeMap(Map<? extends K, ? extends V> m) {
    comparator = null;
    // 添加所有元素
    putAll(m);
}
```

传入 m 的是 Map 类型，构建成初始的 TreeMap 。

调用 #putAll(Map<? extends K, ? extends V> map) 方法，添加所有元素。

#putAll(Map<? extends K, ? extends V> map) 方法，代码如下：

```
// TreeMap.java

public void putAll(Map<? extends K, ? extends V> map) {
    // <1> 路径一，满足如下条件，调用 buildFromSorted 方法来优化处理
    int mapSize = map.size();
    if (size == 0 // 如果 TreeMap 的大小为 0
            && mapSize != 0 // map 的大小非 0
            && map instanceof SortedMap) { // 如果是 map 是 SortedMap 类型
        if (Objects.equals(comparator, ((SortedMap<?,?>)map).comparator())) { // 排序规则相同
            // 增加修改次数
            ++modCount;
            // 基于 SortedMap 顺序迭代插入即可
            try {
                buildFromSorted(mapSize, map.entrySet().iterator(),
```

```
                                null, null);
                    } catch (java.io.IOException | ClassNotFoundException cannotHappen) {
                    }
                    return;
                }
            }
            // <2> 路径二，直接遍历 map 来添加
            super.putAll(map);
        }
```

分成 <1> 和 <2> 两种情况。其中，<1> 是作为优化的方式，处理在 TreeMap 为空时，并且 map 为 SortedMap 类型时，可以直接调用 #buildFromSorted(int level, int lo, int hi, int redLevel, Iterator<?> it, ObjectInputStream str, V defaultVal) 方法，可以基于 SortedMap 顺序迭代插入即可，性能更优。

5. 添加单个元素

#put(K key, V value) 方法，添加单个元素。代码如下：

```
// TreeMap.java

public V put(K key, V value) {
    // 记录当前根节点
    Entry<K,V> t = root;
    // <1> 如果无根节点，则直接使用 key-value 键值对，创建根节点
    if (t == null) {
        // <1.1> 校验 key 类型。
        compare(key, key); // type (and possibly null) check

        // <1.2> 创建 Entry 节点
        root = new Entry<>(key, value, null);
        // <1.3> 设置 key-value 键值对的数量
        size = 1;
        // <1.4> 增加修改次数
        modCount++;
        return null;
    }
    // <2> 遍历红黑树
    int cmp; // key 比父节点小还是大
    Entry<K,V> parent; // 父节点
    // split comparator and comparable paths
    Comparator<? super K> cpr = comparator;
    if (cpr != null) { // 如果有自定义 comparator ，则使用它来比较
        do {
            // <2.1> 记录新的父节点
            parent = t;
            // <2.2> 比较 key
            cmp = cpr.compare(key, t.key);
            // <2.3> 比 key 小，说明要遍历左子树
            if (cmp < 0)
                t = t.left;
            // <2.4> 比 key 大，说明要遍历右子树
            else if (cmp > 0)
                t = t.right;
            // <2.5> 说明，相等，说明要找到的 t 就是 key 对应的节点，直接设置 value 即可。
```

```
            else
                return t.setValue(value);
        } while (t != null);  // <2.6>
    } else { // 如果没有自定义 comparator ，则使用 key 自身比较器来比较
        if (key == null) // 如果 key 为空，则抛出异常
            throw new NullPointerException();
        @SuppressWarnings("unchecked")
        Comparable<? super K> k = (Comparable<? super K>) key;
        do {
            // <2.1> 记录新的父节点
            parent = t;
            // <2.2> 比较 key
            cmp = k.compareTo(t.key);
            // <2.3> 比 key 小，说明要遍历左子树
            if (cmp < 0)
                t = t.left;
            // <2.4> 比 key 大，说明要遍历右子树
            else if (cmp > 0)
                t = t.right;
            // <2.5> 说明，相等，说明要找到的 t 就是 key 对应的节点，直接设置 value 即可。
            else
                return t.setValue(value);
        } while (t != null); // <2.6>
    }
    // <3> 创建 key-value 的 Entry 节点
    Entry<K,V> e = new Entry<>(key, value, parent);
    // 设置左右子树
    if (cmp < 0) // <3.1>
        parent.left = e;
    else // <3.2>
        parent.right = e;
    // <3.3> 插入后，进行自平衡
    fixAfterInsertion(e);
    // <3.4> 设置 key-value 键值对的数量
    size++;
    // <3.5> 增加修改次数
    modCount++;
    return null;
}
```

虽然比较长，逻辑还是相对清晰的。因为红黑树是二叉查找树，所以我们可以使用二分查找的方式遍历红黑树。循环遍历红黑树的节点，根据不同的结果，进行处理：

- 如果当前节点比 key 小，则遍历左子树。
- 如果当前节点比 key 大，则遍历右子树。
- 如果当前节点比 key 相等，则直接设置该节点的 value 即可。
- 如果遍历到叶子节点，无法满足上述情况，则说明我们需要给 key-value 键值对，创建 Entry 节点。如果比叶子节点小，则作为左子树；如果比叶子节点大，则作为右子树。

<1> 处，如果无根节点，则直接使用 key-value 键值对，创建根节点。

- <1.1> 处，调用 #compare(Object k1, Object k2) 方法， 比较 key 。代码如下：

```
// TreeMap.java

final int compare(Object k1, Object k2) {
    return comparator == null ?
            ((Comparable<? super K>)k1).compareTo((K)k2) // 如果没有自定义 comparator 比较器，则使用 key
```

```
                    : comparator.compare((K)k1, (K)k2); // 如果有自定义 comparator 比较器，则使用它来比较。
            }
```

        - 根据是否有自定义的 comparator 比较器，进行 key 的比较。
    - <1.2> 处，创建 key-value 键值对的 Entry 节点，并赋值给 root 节点。

<2> 处，遍历红黑树。会分成是否有自定义的 comparator 作为遍历左右节点的比较器，逻辑是相同的。所以，我们只看 cpr != null 的部分先。

- <2.1> 处，记录新的父节点。目的是，如果遍历到叶子节点 t 时，无法继续遍历时，此时 parent 作为被插入的父节点。
- <2.2> 处，比较 key 。
- <2.3> 处， 比 key 小，说明要遍历左子树。
- <2.4> 处，比 key 大，说明要遍历右子树。
- <2.5> 处，相等，说明要找到的 t 就是 key 对应的节点，直接设置 value 即可。
- <2.6> 处，通过 while(t != null) 来不断遍历，而 t 作为当前遍历到的节点。如果遍历到 t 为空时，说明二分查找不到 key 对应的节点，此时只能创建 key-value 的节点，根据 key 大小作为 parent 的左右节点。

<3> 处，创建 key-value 的 Entry 节点。

- <3.1> 处，如果 key 比 parent 节点的 key 小，作为 parent 的左子节点。
- <3.2> 处，如果 key 比 parent 节点的 key 大，作为 parent 的右子节点。
- <3.3> 处，调用 fixAfterInsertion(Entry<K,V> x) 方法，插入后，进行自平衡。关于这块，我们就先不进行深入了。

另外，因为 TreeMap 是基于树的结构实现，所以无需考虑扩容问题。

6. 获得单个元素

#get(Object key) 方法，获得 key 对应的 value 值。代码如下：

```
// TreeMap.java

public V get(Object key) {
    // 获得 key 对应的 Entry 节点
    Entry<K,V> p = getEntry(key);
    // 返回 value 值
    return (p == null ? null : p.value);
}

final Entry<K,V> getEntry(Object key) { // 不使用 comparator 查找
    // Offload comparator-based version for sake of performance
    // 如果自定义了 comparator 比较器，则基于 comparator 比较来查找
    if (comparator != null)
        return getEntryUsingComparator(key);
    // 如果 key 为空，抛出异常
    if (key == null)
        throw new NullPointerException();
    @SuppressWarnings("unchecked")
    Comparable<? super K> k = (Comparable<? super K>) key;
    // 遍历红黑树
    Entry<K,V> p = root;
    while (p != null) {
        // 比较值
```

```
            int cmp = k.compareTo(p.key);
            // 如果 key 小于当前节点，则遍历左子树
            if (cmp < 0)
                p = p.left;
            // 如果 key 大于当前节点，则遍历右子树
            else if (cmp > 0)
                p = p.right;
            // 如果 key 相等，则返回该节点
            else
                return p;
        }
        // 查找不到，返回 null
        return null;
    }

    final Entry<K,V> getEntryUsingComparator(Object key) { // 使用 comparator 查找
        @SuppressWarnings("unchecked")
        K k = (K) key;
        Comparator<? super K> cpr = comparator;
        if (cpr != null) {
            // 遍历红黑树
            Entry<K,V> p = root;
            while (p != null) {
                // 比较值
                int cmp = cpr.compare(k, p.key);
                // 如果 key 小于当前节点，则遍历左子树
                if (cmp < 0)
                    p = p.left;
                // 如果 key 大于当前节点，则遍历右子树
                else if (cmp > 0)
                    p = p.right;
                // 如果 key 相等，则返回该节点
                else
                    return p;
            }
        }
        // 查找不到，返回 null
        return null;
    }
```

和我们在 「5. 添加单个元素」 中看到的，也是基于红黑树进行二分查找，逻辑是一致的。
如果未自定义 `comparator` 比较器，则调用 `#getEntry(Object key)` 方法，使用 key 自身的排序，进行比较二分查找。
如果有自定义 `comparator` 比较器，则调用 `#getEntryUsingComparator(Object key)` 方法，使用 `comparator` 的排序，进行比较二分查找。

`#containsKey(Object key)` 方法，判断是否存在指定 key 。代码如下：

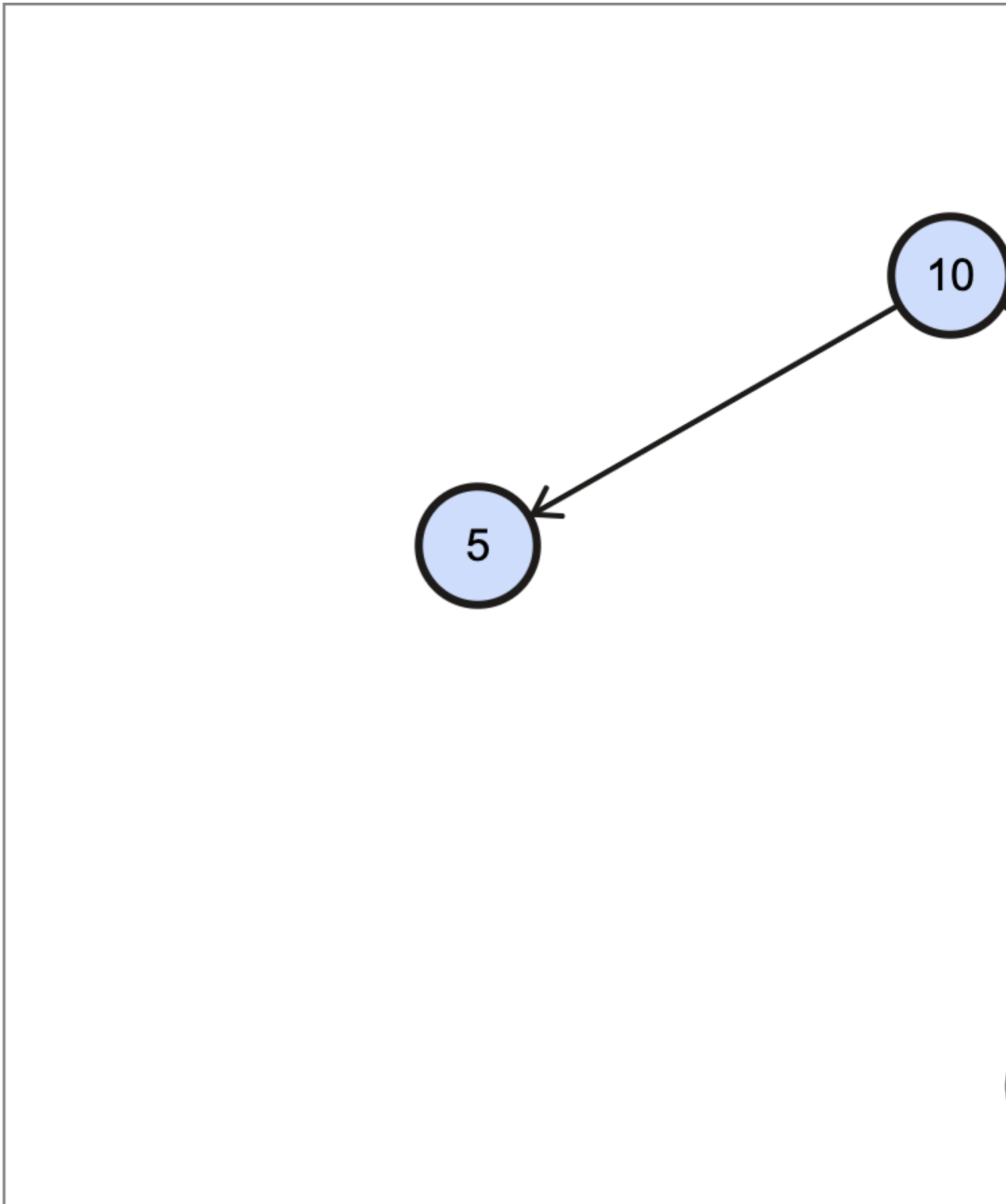
```
    // TreeMap.java

    public boolean containsKey(Object key) {
        return getEntry(key) != null;
    }
```

基于 `#getEntry(key)` 方法来实现。

7. 删除单个元素

相比 「5. 添加单个元素」 来说，删除会更加复杂一些。所以呢，我们先看删除的四种情况。为了让案例更加复杂，我们会使用一颗二叉查找树来举例子。因为，在去掉自平衡的逻辑的情况下，红黑树的删除和二叉查查找树的删除逻辑是一致的。

对于二叉查找树的删除，需要保证删除节点后，能够继续满足二叉和查找的特性。



该图通过 http://btv.melezinek.cz/binary-search-tree.html 绘制，胖友可以通过使用它，辅助理解这个过程。

情况一，无子节点。
直接删除父节点对其的指向即可。
例如说，叶子节点 5、11、14、18 。

情况二，只有左子节点。
将删除节点的父节点，指向删除节点的左子节点。
例如说，节点 20 。可以通过将节点 15 的右子节点指向节点 19 。

情况三，只有右子节点。
和情况二的处理方式一致。将删除节点的父节点，指向删除节点的右子节点。
图中暂无示例，胖友自己脑补下，嘿嘿。

情况四，有左子节点 + 右子节点。
这种情况，相对会比较复杂，因为无法使用子节点替换掉删除的节点。所以此时有一个巧妙的思路。我们结合删除节点 15 来举例。

1、先查找节点 15 的右子树的最小值，找到是节点 17 。
2、将节点 17 设置到节点 15 上。因为节点 17 是右子树的最小值，能够满足比节点 15 的左子树都大，右子树都小。这样，问题就可以变成删除节点 17 。
3、删除节点 17 的过程，满足情况三。将节点 19 的左子节点指向节点 18 即可。

理解完这四种情况后，我们来看看代码。#remove(Object key) 方法，移除 key 对应的 Entry 节点。代码如下：

```
// TreeMap.java

public V remove(Object key) {
    // <1> 获得 key 对应的 Entry 节点
    Entry<K,V> p = getEntry(key);
    // <2> 如果不存在，则返回 null ，无需删除
    if (p == null)
        return null;

    V oldValue = p.value;
    // <3> 删除节点
    deleteEntry(p);
    return oldValue;
}
```

<1> 处，调用 #getEntry(Object key) 方法，获得 key 对应的 Entry 节点。
<2> 处，如果不存在，则返回 null ，无需删除。
<3> 处，调用 #deleteEntry(Entry<K,V> p) 方法，删除该节点。

#deleteEntry(Entry<K,V> p) 方法，代码如下：

```
// TreeMap.java

private void deleteEntry(Entry<K,V> p) {
    // 增加修改次数
    modCount++;
    // 减少 key-value 键值对数
```

```
size--;

// If strictly internal, copy successor's element to p and then make p
// point to successor.
// <1> 如果删除的节点 p 既有左子节点，又有右子节点，
if (p.left != null && p.right != null) {
    // <1.1> 获得右子树的最小值
    Entry<K,V> s = successor(p);
    // <1.2> 修改 p 的 key-value 为 s 的 key-value 键值对
    p.key = s.key;
    p.value = s.value;
    // <1.3> 设置 p 指向 s 。此时，就变成删除 s 节点了。
    p = s;
} // p has 2 children

// Start fixup at replacement node, if it exists.
// <2> 获得替换节点
Entry<K,V> replacement = (p.left != null ? p.left : p.right);
// <3> 有子节点的情况
if (replacement != null) {
    // Link replacement to parent
    // <3.1> 替换节点的父节点，指向 p 的父节点
    replacement.parent = p.parent;
    // <3.2.1> 如果 p 的父节点为空，则说明 p 是根节点，直接 root 设置为替换节点
    if (p.parent == null)
        root = replacement;
    // <3.2.2> 如果 p 是父节点的左子节点，则 p 的父子节的左子节指向替换节点
    else if (p == p.parent.left)
        p.parent.left  = replacement;
    // <3.2.3> 如果 p 是父节点的右子节点，则 p 的父子节的右子节指向替换节点
    else
        p.parent.right = replacement;

    // Null out links so they are OK to use by fixAfterDeletion.
    // <3.3> 置空 p 的所有指向
    p.left = p.right = p.parent = null;

    // Fix replacement
    // <3.4> 如果 p 的颜色是黑色，则执行自平衡
    if (p.color == BLACK)
        fixAfterDeletion(replacement);
// <4> 如果 p 没有父节点，说明删除的是根节点，直接置空 root 即可
} else if (p.parent == null) { // return if we are the only node.
    root = null;
// <5> 如果删除的没有左子树，又没有右子树
} else { //  No children. Use self as phantom replacement and unlink.
    // <5.1> 如果 p 的颜色是黑色，则执行自平衡
    if (p.color == BLACK)
        fixAfterDeletion(p);

    // <5.2> 删除 p 和其父节点的相互指向
    if (p.parent != null) {
        // 如果 p 是父节点的左子节点，则置空父节点的左子节点
        if (p == p.parent.left)
            p.parent.left = null;
        // 如果 p 是父节点的右子节点，则置空父节点的右子节点
        else if (p == p.parent.right)
            p.parent.right = null;
        // 置空 p 对父节点的指向
        p.parent = null;
```

```
        }
    }
}
```

<1> 处，如果删除的节点 p 既有左子节点，又有右子节点，则符合我们提到的情况四。在这里，我们需要将其转换成情况三。
- <1.1> 处，调用 #successor(Entry<K,V> t) 方法，获得右子树的最小值。这里，我们先不深究 #successor(Entry<K,V> t) 方法的具体代码，知道在这里的用途即可。
- <1.2> 处，修改 p 的 key-value 为 s 的 key-value 键值对。这样，我们就完成 s 对 p 的替换。
- <1.3> 处，设置 p 指向 s 。此时，就变成删除 s 节点了。此时，情况四就转换成了情况三了。

<2> 处，获得替换节点。此时对于 p 来说，至多有一个子节点，要么左子节点，要么右子节点，要么没有子节点。

<3> 处，有左子节点，或者右子节点的情况：
- <3.1> 处，替换节点的父节点，指向 p 的父节点。
- <3.2.1> + <3.2.2> + <3.2.3> 处，将 p 的父节点的子节点，指向替换节点。
- <3.3> 处， 置空 p 的所有指向。
- <3.4> 处，如果 p 的颜色是黑色，则调用 #fixAfterDeletion(Entry<K,V> x) 方法，执行自平衡。

<4> 处，如果 p 没有父节点，说明删除的是根节点，直接置空 root 即可。

<5> 处，既没有左子树，又没有右子树的情况：
- <5.1> 处，如果 p 的颜色是黑色，则调用 #fixAfterDeletion(Entry<K,V> x) 方法，执行自平衡。
- <5.2> 处，删除 p 和其父节点的相互指向。

这样一看，其实删除节点的逻辑，也并不是怎么复杂噢。感兴趣的胖友，可以去 LeetCode 找树的题目刷一刷，哈哈。

在前面，我们漏了一个 #successor(Entry<K,V> t) 静态方法，没有详细来看。获得 t 节点的后继节点，代码如下：

```
// TreeMap.java

static <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {
    // <1> 如果 t 为空，则返回 null
    if (t == null)
        return null;
    // <2> 如果 t 的右子树非空，则取右子树的最小值
    else if (t.right != null) {
        // 先取右子树的根节点
        Entry<K,V> p = t.right;
        // 再取该根节点的做子树的最小值，即不断遍历左节点
        while (p.left != null)
            p = p.left;
        // 返回
        return p;
    // <3> 如果 t 的右子树为空
    } else {
        // 先获得 t 的父节点
        Entry<K,V> p = t.parent;
        // 不断向上遍历父节点，直到子节点 ch 不是父节点 p 的右子节点
        Entry<K,V> ch = t;
        while (p != null // 还有父节点
```

```
                && ch == p.right) { // 继续遍历的条件，必须是子节点 ch 是父节点 p 的右子节点
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
```

对于树来说，会存在前序遍历，中序遍历，后续遍历。对于二叉查找树来说，中序遍历恰好满足 key 顺序递增。所以，这个方法是基于中序遍历的方式，寻找传入 t 节点的后续节点，也是下一个比 t 大的节点。

<1> 处，如果 t 为空，则返回 null 。

<2> 处，如果 t 有右子树，则右子树的最小值，肯定是它的后继节点。胖友可以自己看下芳芳在代码里写的注释。在 #deleteEntry(Entry<K,V> p) 方法的 <1.1> 处，就走了这块代码分支逻辑。

<3> 处，如果 t 没有右子树，则需要向上遍历父节点。胖友可以自己看下芳芳在代码里写的注释，结合 图 来理解。

- 简单来说，寻找第一个祖先节点 p 是其父节点的左子节点。因为是中序遍历，该节点的左子树肯定已经遍历完，在没有右子节点的情况下，需要找到其所在的“大子树”，成为左子树的情况。
- 例如说，节点 14 来说，需要按照 14 -> 13 -> 15 的路径，从而找到节点 15 是其后继节点。

8. 查找接近的元素

在 NavigableMap 中，定义了四个查找接近的元素：

#lowerEntry(K key) 方法，小于 key 的节点
#floorEntry(K key) 方法，小于等于 key 的节点
#higherEntry(K key) 方法，大于 key 的节点
#ceilingEntry(K key) 方法，大于等于 key 的节点

我们逐个来看看哈。

#ceilingEntry(K key) 方法，大于等于 key 的节点。代码如下：

```
// TreeMap.java

public Map.Entry<K,V> ceilingEntry(K key) {
    // <1>
    // <2>
    return exportEntry(getCeilingEntry(key));
}

static <K,V> Map.Entry<K,V> exportEntry(TreeMap.Entry<K,V> e) {
    return (e == null) ? null :
        new AbstractMap.SimpleImmutableEntry<>(e);
}

final Entry<K,V> getCeilingEntry(K key) {
    Entry<K,V> p = root;
    // <3> 循环二叉查找遍历红黑树
    while (p != null) {
        // <3.1> 比较 key
```

```
            int cmp = compare(key, p.key);
            // <3.2> 当前节点比 key 大，则遍历左子树，这样缩小节点的值
            if (cmp < 0) {
                // <3.2.1> 如果有左子树，则遍历左子树
                if (p.left != null)
                    p = p.left;
                // <3.2.2.> 如果没有，则直接返回该节点
                else
                    return p;
            // <3.3> 当前节点比 key 小，则遍历右子树，这样放大节点的值
            } else if (cmp > 0) {
                // <3.3.1> 如果有右子树，则遍历右子树
                if (p.right != null) {
                    p = p.right;
                } else {
                    // <3.3.2> 找到当前的后继节点
                    Entry<K,V> parent = p.parent;
                    Entry<K,V> ch = p;
                    while (parent != null && ch == parent.right) {
                        ch = parent;
                        parent = parent.parent;
                    }
                    return parent;
                }
            // <3.4> 如果相等，则返回该节点即可
            } else
                return p;
        }
        // <3.5>
        return null;
    }
```

<1> 处，调用 `#getCeilingEntry(K key)` 方法，查找满足大于等于 `key` 的 Entry 节点。

<2> 处，调用 `#exportEntry(TreeMap.Entry<K,V> e)` 方法，创建不可变的 SimpleImmutableEntry 节点。这样，避免使用者直接修改节点，例如说修改 `key` 导致破坏红黑树。

本质上，`#getCeilingEntry(K key)` 方法，是加强版的二叉树查找，在找不到 `key` 的情况下，找到比 `key` 大且最接近的节点。

<3> 处，循环二叉查找遍历红黑树，每一轮都会在 <3.1> 处，通过调用 `#compare(Object k1, Object k2)` 方法，比较当前节点的 key 与 `key` 的大小。

<3.4> 处，当前节点和 `key` 相等，则返回该节点。此时，我们找到了和 `key` 相等的节点。

<3.2> 处，当前节点比 `key` 大，则遍历左子树，这样缩小节点的值。

- <3.2.1> 处，如果有左子树，则遍历左子树。
- <3.2.2> 处，如果没有，则直接返回该节点。此时，我们找到的是比 `key` 大且最接近的节点。

<3.3> 处，当前节点比 `key` 小，则遍历右子树，这样放大节点的值。

- <3.3.1> 处，如果有右子树，则遍历右子树。
- <3.3.2> 处，找到当前的后继节点。这小块的代码，和我们在 「7. 删除单个元素」 的 `#successor(Entry<K,V> t)` 」 方法的 <3> 处的代码是一致的。

<3.5> 处，极端情况下，找不到，返回 `null` 。

对于 <3.3.2> 的逻辑，可能胖友理解起来会有一点懵逼。我们来看一个示例。如下图：

假设查找节点 60 时，遍历路径为 20 -> 30 -> 40 -> 50 ，此时没有右子树，查找后继节点为不存在，返回 null。

假设查找节点 19 时，遍历路径为 20 -> 10 -> 15 -> 18 ，此时没有右子树，查找后继节点为节点 20 ，返回节点 20 。

> 芳芳：有点不造怎么特别理论的描述，为什么这样 <3.3.2> 的逻辑是成立的。
>
> 从直观感受上来说，对于没有右子树的节点，其后继节点一定大于它。
>
> 并且，以节点 10 举例子。在我们因为 key 比节点 20 小时，遍历其左子树 leftTree 。在找不到匹配的节点时，此时 leftTree 的根节点 20 ，肯定是满足比 key 大且最接近的节点。恰好，根节点 20 就是节点 18 的后继节点。
>
> 等后面我在想想怎么能够描述的更清楚。如果胖友有更好的解释，可以星球给芳芳留言。
>
> 目前的话，可以多画图理解。

#higherEntry(K key) 方法，大于 key 的节点。代码如下：

```
// TreeMap.java

public Map.Entry<K,V> higherEntry(K key) {
    return exportEntry(getHigherEntry(key));
}

final Entry<K,V> getHigherEntry(K key) {
    Entry<K,V> p = root;
    // 循环二叉查找遍历红黑树
    while (p != null) {
        // 比较 key
        int cmp = compare(key, p.key);
        // 当前节点比 key 大，则遍历左子树，这样缩小节点的值
        if (cmp < 0) {
            // 如果有左子树，则遍历左子树
            if (p.left != null)
                p = p.left;
            // 如果没有，则直接返回该节点
            else
                return p;
        // 当前节点比 key 小，则遍历右子树，这样放大节点的值
        } else {
            // 如果有右子树，则遍历右子树
            if (p.right != null) {
                p = p.right;
            } else {
                // 找到当前的后继节点
                Entry<K,V> parent = p.parent;
                Entry<K,V> ch = p;
                while (parent != null && ch == parent.right) {
                    ch = parent;
                    parent = parent.parent;
                }
                return parent;
            }
        }
        // <X> 此处，相等的情况下，不返回
```

```
        }
        // 查找不到，返回 null
        return null;
    }
```

和 #ceilingEntry(K key) 逻辑的差异，在于 <X> 处，相等的情况下，不返回该节点。

#ceilingEntry(K key) 方法，小于等于 key 的节点。代码如下：

```
    // TreeMap.java

    public Map.Entry<K,V> floorEntry(K key) {
        return exportEntry(getFloorEntry(key));
    }

    final Entry<K,V> getFloorEntry(K key) {
        Entry<K,V> p = root;
        // 循环二叉查找遍历红黑树
        while (p != null) {
            // 比较 key
            int cmp = compare(key, p.key);
            if (cmp > 0) {
                // 如果有右子树，则遍历右子树
                if (p.right != null)
                    p = p.right;
                // 如果没有，则直接返回该节点
                else
                    return p;
            // 当前节点比 key 小，则遍历右子树，这样放大节点的值
            } else if (cmp < 0) {
                // 如果有左子树，则遍历左子树
                if (p.left != null) {
                    p = p.left;
                } else {
                    // 找到当前节点的前继节点
                    Entry<K,V> parent = p.parent;
                    Entry<K,V> ch = p;
                    while (parent != null && ch == parent.left) {
                        ch = parent;
                        parent = parent.parent;
                    }
                    return parent;
                }
            // 如果相等，则返回该节点即可
            } else
                return p;

        }
        // 查找不到，返回 null
        return null;
    }
```

思路是一致的，胖友自己看下注释噢。

#getLowerEntry(K key) 方法，小于 key 的节点。代码如下：

```
// TreeMap.java

public Map.Entry<K,V> lowerEntry(K key) {
    return exportEntry(getLowerEntry(key));
}

final Entry<K,V> getLowerEntry(K key) {
    Entry<K,V> p = root;
    // 循环二叉查找遍历红黑树
    while (p != null) {
        // 比较 key
        int cmp = compare(key, p.key);
        // 当前节点比 key 小，则遍历右子树，这样放大节点的值
        if (cmp > 0) {
            // 如果有右子树，则遍历右子树
            if (p.right != null)
                p = p.right;
            // 如果没有，则直接返回该节点
            else
                return p;
        // 当前节点比 key 大，则遍历左子树，这样缩小节点的值
        } else {
            // 如果有左子树，则遍历左子树
            if (p.left != null) {
                p = p.left;
            } else {
                // 找到当前节点的前继节点
                Entry<K,V> parent = p.parent;
                Entry<K,V> ch = p;
                while (parent != null && ch == parent.left) {
                    ch = parent;
                    parent = parent.parent;
                }
                return parent;
            }
        }
        // 此处，相等的情况下，不返回
    }
    // 查找不到，返回 null
    return null;
}
```

思路是一致的，胖友自己看下注释噢。

在一些场景下，我们并不需要返回 Entry 节点，只需要返回符合条件的 key 即可。所以有了对应的如下四个方法：

```
// TreeMap.java

public K lowerKey(K key) {
    return keyOrNull(getLowerEntry(key));
}

public K floorKey(K key) {
    return keyOrNull(getFloorEntry(key));
}

public K ceilingKey(K key) {
```

```
        return keyOrNull(getCeilingEntry(key));
    }

    public K higherKey(K key) {
        return keyOrNull(getHigherEntry(key));
    }

    static <K,V> K keyOrNull(TreeMap.Entry<K,V> e) {
        return (e == null) ? null : e.key;
    }
```

9. 获得首尾的元素

#firstEntry() 方法，获得首个 Entry 节点。代码如下：

```
    // TreeMap.java

    public Map.Entry<K,V> firstEntry() {
        return exportEntry(getFirstEntry());
    }

    final Entry<K,V> getFirstEntry() {
        Entry<K,V> p = root;
        if (p != null)
            // 循环，不断遍历到左子节点，直到没有左子节点
            while (p.left != null)
                p = p.left;
        return p;
    }
```

通过不断遍历到左子节点，直到没有左子节点。

在 #getFirstEntry() 方法的基础上，还提供了另外两个方法：

```
    // TreeMap.java

    public Map.Entry<K,V> pollFirstEntry() { // 获得并移除首个 Entry 节点
        // 获得首个 Entry 节点
        Entry<K,V> p = getFirstEntry();
        Map.Entry<K,V> result = exportEntry(p);
        // 如果存在，则进行删除。
        if (p != null)
            deleteEntry(p);
        return result;
    }

    public K firstKey() {
        return key(getFirstEntry());
    }
    static <K> K key(Entry<K,?> e) {
        if (e == null) // 如果不存在 e 元素，则抛出 NoSuchElementException 异常
            throw new NoSuchElementException();
        return e.key;
    }
```

#lastEntry() 方法，获得尾部 Entry 节点。代码如下：

```
// TreeMap.java

public Map.Entry<K,V> lastEntry() {
    return exportEntry(getLastEntry());
}

final Entry<K,V> getLastEntry() {
    Entry<K,V> p = root;
    if (p != null)
        // 循环，不断遍历到右子节点，直到没有右子节点
        while (p.right != null)
            p = p.right;
    return p;
}
```

通过不断遍历到右子节点，直到没有右子节点。

在 #getLastEntry() 方法的基础上，还提供了另外两个方法：

```
// TreeMap.java

public Map.Entry<K,V> pollLastEntry() { // 获得并移除尾部 Entry 节点
    // 获得尾部 Entry 节点
    Entry<K,V> p = getLastEntry();
    Map.Entry<K,V> result = exportEntry(p);
    // 如果存在，则进行删除。
    if (p != null)
        deleteEntry(p);
    return result;
}

public K lastKey() {
    return key(getLastEntry());
}
```

在这里，补充一个 #containsValue(Object value) 方法，通过中序遍历的方式，遍历查找值为 value 的节点是否存在。代码如下：

```
// TreeMap.java

public boolean containsValue(Object value) {
    for (Entry<K,V> e = getFirstEntry(); // 获得首个 Entry 节点
         e != null;  // 遍历到没有下一个节点
         e = successor(e)) { // 通过中序遍历，获得下一个节点
        if (valEquals(value, e.value)) // 判断值是否相等
            return true;
    }
    return false;
}

static final boolean valEquals(Object o1, Object o2) {
    return (o1==null ? o2==null : o1.equals(o2));
}
```

10. 清空

#clear() 方法，清空。代码如下：

```
// TreeMap.java

public void clear() {
    // 增加修改次数
    modCount++;
    // key-value 数量置为 0
    size = 0;
    // 设置根节点为 null
    root = null;
}
```

11. 克隆

#clone() 方法，克隆 TreeMap 。代码如下：

```
// TreeMap.java

public Object clone() {
    // 克隆创建 TreeMap 对象
    TreeMap<?,?> clone;
    try {
        clone = (TreeMap<?,?>) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }

    // Put clone into "virgin" state (except for comparator)
    // 重置 clone 对象的属性
    clone.root = null;
    clone.size = 0;
    clone.modCount = 0;
    clone.entrySet = null;
    clone.navigableKeySet = null;
    clone.descendingMap = null;

    // Initialize clone with our mappings
    // 使用自己，构造 clone 对象的红黑树
    try {
        clone.buildFromSorted(size, entrySet().iterator(), null, null);
    } catch (java.io.IOException | ClassNotFoundException cannotHappen) {
    }

    return clone;
}
```

12. 序列化

#writeObject(ObjectOutputStream s) 方法，序列化 TreeMap 对象。代码如下：

```
// TreeMap.java

@java.io.Serial
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out the Comparator and any hidden stuff
    // 写入非静态属性、非 transient 属性
    s.defaultWriteObject();

    // Write out size (number of Mappings)
    // 写入 key-value 键值对数量
    s.writeInt(size);

    // Write out keys and values (alternating)
    // 写入具体的 key-value 键值对
    for (Map.Entry<K, V> e : entrySet()) {
        s.writeObject(e.getKey());
        s.writeObject(e.getValue());
    }
}
```

比较简单，胖友自己瞅瞅即可。

13. 反序列化

#readObject(ObjectInputStream s) 方法，反序列化成 TreeMap 对象。代码如下：

```
// TreeMap.java

@java.io.Serial
private void readObject(final java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in the Comparator and any hidden stuff
    // 读取非静态属性、非 transient 属性
    s.defaultReadObject();

    // Read in size
    // 读取 key-value 键值对数量 size
    int size = s.readInt();

    // 使用输入流，构建红黑树。
    // 因为序列化时，已经是顺序的，所以输入流也是顺序的
    buildFromSorted(size, null, s, null); // 注意，此时传入的是 s 参数，输入流
}
```

14. 获得迭代器

芮芮：本小节，可以选择性看，或者不看。

#keyIterator() 方法，获得 key 的正序迭代器。代码如下：

```
// TreeMap.java
```

```
Iterator<K> keyIterator() {
    return new KeyIterator(getFirstEntry()); // 获得的是首个元素
}
```

创建的是 KeyIterator 迭代器。在 「14.2 KeyIterator」 详细解析。

#descendingKeyIterator() 方法，获得 key 的倒序迭代器。代码如下：

```
// TreeMap.java

Iterator<K> descendingKeyIterator() {
    return new DescendingKeyIterator(getLastEntry()); // 获得的是尾部元素
}
```

创建的是 DescendingKeyIterator 迭代器。在 「14.3 DescendingKeyIterator」 详细解析。

不过上述两个方法，都不是 public 方法，只提供给 TreeMap 内部使用。

14.1 PrivateEntryIterator

PrivateEntryIterator ，实现 Iterator 接口，提供了 TreeMap 的通用实现 Iterator 的抽象类。代码如下：

```
// TreeMap.java

abstract class PrivateEntryIterator<T> implements Iterator<T> {

    /**
     * 下一个节点
     */
    Entry<K,V> next;
    /**
     * 最后返回的节点
     */
    Entry<K,V> lastReturned;
    /**
     * 当前的修改次数
     */
    int expectedModCount;

    PrivateEntryIterator(Entry<K,V> first) {
        expectedModCount = modCount;
        lastReturned = null;
        next = first;
    }

    public final boolean hasNext() {
        return next != null;
    }

    final Entry<K,V> nextEntry() { // 获得下一个 Entry 节点
        // 记录当前节点
```

```
            Entry<K,V> e = next;
            // 如果没有下一个，抛出 NoSuchElementException 异常
            if (e == null)
                throw new NoSuchElementException();
            // 如果发生了修改，抛出 ConcurrentModificationException 异常
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            // 获得 e 的后继节点，赋值给 next
            next = successor(e);
            // 记录最后返回的节点
            lastReturned = e;
            // 返回当前节点
            return e;
        }

        final Entry<K,V> prevEntry() { // 获得前一个 Entry 节点
            // 记录当前节点
            Entry<K,V> e = next;
            // 如果没有下一个，抛出 NoSuchElementException 异常
            if (e == null)
                throw new NoSuchElementException();
            // 如果发生了修改，抛出 ConcurrentModificationException 异常
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            // 获得 e 的前继节点，赋值给 next
            next = predecessor(e);
            // 记录最后返回的节点
            lastReturned = e;
            // 返回当前节点
            return e;
        }

        public void remove() { // 删除节点
            // 如果当前返回的节点不存在，则抛出 IllegalStateException 异常
            if (lastReturned == null)
                throw new IllegalStateException();
            // 如果发生了修改，抛出 ConcurrentModificationException 异常
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            // deleted entries are replaced by their successors
            // 在 lastReturned 左右节点都存在的时候，实际在 deleteEntry 方法中，是将后继节点替换到 lastReturned 中
            // 因此，next 需要指向 lastReturned
            if (lastReturned.left != null && lastReturned.right != null)
                next = lastReturned;
            // 删除节点
            deleteEntry(lastReturned);
            // 记录新的修改次数
            expectedModCount = modCount;
            // 置空 lastReturned
            lastReturned = null;
        }

    }
```

整体代码比较简单，胖友自己看看艿艿写的注释噢。

在上述代码中，我们会看到 `#predecessor(Entry<K,V> t)` 静态方法，我们来看看。获得 `t` 节点的前继节点，代码如下：

```
// TreeMap.java

static <K,V> Entry<K,V> predecessor(Entry<K,V> t) {
    // 如果 t 为空，则返回 null
    if (t == null)
        return null;
    // 如果 t 的左子树非空，则取左子树的最大值
    else if (t.left != null) {
        Entry<K,V> p = t.left;
        while (p.right != null)
            p = p.right;
        return p;
    // 如果 t 的左子树为空
    } else {
        // 先获得 t 的父节点
        Entry<K,V> p = t.parent;
        // 不断向上遍历父节点，直到子节点 ch 不是父节点 p 的左子节点
        Entry<K,V> ch = t;
        while (p != null // 还有父节点
                && ch == p.left) { // 继续遍历的条件，必须是子节点 ch 是父节点 p 的左子节点
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
```

和 #successor(Entry<K,V> t) 方法，是一样的思路。所以，胖友跟着注释，自己再理解下。

14.2 KeyIterator

KeyIterator ，继承 PrivateEntryIterator 抽象类，key 的正序迭代器。代码如下：

```
// TreeMap.java

final class KeyIterator extends PrivateEntryIterator<K> {

    KeyIterator(Entry<K,V> first) {
        super(first);
    }

    // 实现 next 方法，实现正序
    public K next() {
        return nextEntry().key;
    }

}
```

14.3 DescendingKeyIterator

DescendingKeyIterator ，继承 PrivateEntryIterator 抽象类，key 的倒序迭代器。代码如下：

```
// TreeMap.java
```

```
final class DescendingKeyIterator extends PrivateEntryIterator<K> {

    DescendingKeyIterator(Entry<K,V> first) {
        super(first);
    }

    // 实现 next 方法，实现倒序
    public K next() {
        return prevEntry().key;
    }

    // 重写 remove 方法，因为在 deleteEntry 方法中，在 lastReturned 左右节点都存在的时候，是将后继节点替换到 lastRetu
    // 而这个逻辑，对于倒序遍历，没有影响。
    public void remove() {
        // 如果当前返回的节点不存在，则抛出 IllegalStateException 异常
        if (lastReturned == null)
            throw new IllegalStateException();
        // 如果发生了修改，抛出 ConcurrentModificationException 异常
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        // 删除节点
        deleteEntry(lastReturned);
        // 置空 lastReturned
        lastReturned = null;
        // 记录新的修改次数
        expectedModCount = modCount;
    }

}
```

14.4 EntryIterator

EntryIterator ，继承 PrivateEntryIterator 抽象类，Entry 的正序迭代器。代码如下：

```
// TreeMap.java

final class EntryIterator extends PrivateEntryIterator<Map.Entry<K,V>> {

    EntryIterator(Entry<K,V> first) {
        super(first);
    }

    // 实现 next 方法，实现正序
    public Map.Entry<K,V> next() {
        return nextEntry();
    }

}
```

14.5 ValueIterator

ValueIterator ，继承 PrivateEntryIterator 抽象类，value 的正序迭代器。代码如下：

```
// TreeMap.java

final class ValueIterator extends PrivateEntryIterator<V> {

    ValueIterator(Entry<K,V> first) {
        super(first);
    }

    // 实现 next 方法，实现正序
    public V next() {
        return nextEntry().value;
    }

}
```

15. 转换成 Set/Collection

芮芮：本小节，可以选择性看，或者不看。

15.1 keySet

#keySet() 方法，获得正序的 key Set 。代码如下：

```
// TreeMap.java

/**
 * 正序的 KeySet 缓存对象
 */
private transient KeySet<K> navigableKeySet;

public Set<K> keySet() {
    return navigableKeySet();
}

public NavigableSet<K> navigableKeySet() {
    KeySet<K> nks = navigableKeySet;
    return (nks != null) ? nks : (navigableKeySet = new KeySet<>(this));
}
```

创建的 KeySet 类。它实现 NavigableSet 接口，继承了 java.util.AbstractSet 抽像类，是 TreeMap 的内部类。比较简单，就不哔哔了。
KeySet 使用的迭代器，就是 「14.2 KeyIterator」 。

15.2 descendingKeySet

#descendingKeySet() 方法，获得倒序的 key Set 。代码如下：

```
// TreeMap.java

/**
 * 倒序的 NavigableMap 缓存对象
```

```
     */
    private transient NavigableMap<K,V> descendingMap;

    public NavigableSet<K> descendingKeySet() {
        return descendingMap().navigableKeySet();
    }

    public NavigableMap<K, V> descendingMap() {
        NavigableMap<K, V> km = descendingMap;
        return (km != null) ? km :
            (descendingMap = new DescendingSubMap<>(this,
                                                    true, null, true,
                                                    true, null, true));
    }
```

首先，调用 #descendingMap() 方法，返回倒序访问当前 TreeMap 的 DescendingSubMap 对象。
然后，调用 #navigableKeySet() 方法，返回 DescendingSubMap 对象的正序的 key Set 。
关于 DescendingSubMap 类，我们在 TODO 来详细解析。

15.3 values

#values() 方法，获得 value 集合。代码如下：

```
    // TreeMap.java

    public Collection<V> values() {
        Collection<V> vs = values;
        if (vs == null) {
            vs = new Values();
            values = vs; // values 缓存，来自 AbstractMap 的属性
        }
        return vs;
    }
```

创建的 Values 类。它继承了 java.util.AbstractCollection 抽像类，是 TreeMap 的内部类。比较简单，就不哔哔了。
Values 使用的迭代器，就是 「14.4 ValueIterator」 。

15.4 entrySet

#entrySet() 方法，获得 Entry 集合。代码如下：

```
    // TreeMap.java

    /**
     * Entry 缓存集合
     */
    private transient EntrySet entrySet;

    public Set<Map.Entry<K,V>> entrySet() {
        EntrySet es = entrySet;
        return (es != null) ? es : (entrySet = new EntrySet());
```

```
}
```

创建的 EntrySet 类。它继承了 java.util.AbstractSet 抽像类，是 TreeMap 的内部类。比较简单，就不哔哔了。
EntrySet 使用的迭代器，就是 「14.3 EntryIterator」 。

16. 查找范围的元素

芳芳：本小节，可以选择性看，或者不看。

这部分，内容有点长。

在 SortedMap 接口中，定义了按照 key 查找范围，返回子 SortedMap 结果的方法：

```
#subMap(K fromKey, K toKey)
#headMap(K toKey)
#tailMap(K fromKey)
```

在 NavigableMap 中，定义了按照 key 查找范围，返回子 NavigableMap 结果的方法：

```
#subMap(K fromKey, K toKey)
#subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)
#headMap(K toKey)
#headMap(K toKey, boolean inclusive)
#tailMap(K fromKey)
#tailMap(K fromKey, boolean inclusive)
```

TreeMap 对上述接口，实现如下方法：

```
// TreeMap.java

// subMap 组
public SortedMap<K,V> subMap(K fromKey, K toKey) {
    return subMap(fromKey, true, toKey, false);
}
public NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,
                                K toKey,   boolean toInclusive) {
    return new AscendingSubMap<>(this,
                                 false, fromKey, fromInclusive,
                                 false, toKey,   toInclusive);
}

// headMap 组
public SortedMap<K,V> headMap(K toKey) {
    return headMap(toKey, false);
}
public NavigableMap<K,V> headMap(K toKey, boolean inclusive) {
    return new AscendingSubMap<>(this,
                                 true,  null,  true,
                                 false, toKey, inclusive);
}

// tailMap 组
```

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

返回的都是 AscendingSubMap 对象。所以，我们在 「XX. AscendingSubMap」 来看。

16.1 NavigableSubMap

NavigableSubMap ，实现 NavigableMap、Serializable 接口，继承 AbstractMap 抽象类，子 NavigableMap 的抽象类。

后续，我们会看到 NavigableSubMap 的两个子类：

AscendingSubMap ，正序的 子 NavigableMap 的实现类。
DescendingSubMap ，倒序的 子 NavigableMap 的实现类。

16.1.1 构造方法

NavigableSubMap 仅有一个构造方法，代码如下：

```
// TreeMap.java#NavigableSubMap.java

/**
 * The backing map.
 */
final TreeMap<K,V> m;

/**
 * lo - 开始位置
 * hi - 结束位置
 */
final K lo, hi;
/**
 * fromStart - 是否从 TreeMap 开头开始。如果是的话，{@link #lo} 可以不传
 * toEnd - 是否从 TreeMap 结尾结束。如果是的话，{@link #hi} 可以不传
 */
final boolean fromStart, toEnd;
/**
 * loInclusive - 是否包含 key 为 {@link #lo}  的元素
 * hiInclusive - 是否包含 key 为 {@link #hi} 的元素
 */
final boolean loInclusive, hiInclusive;

NavigableSubMap(TreeMap<K,V> m,
                boolean fromStart, K lo, boolean loInclusive,
                boolean toEnd,     K hi, boolean hiInclusive) {
    // 如果既不从开头开始，又不从结尾结束，那么就要校验 lo 小于 hi ，否则抛出 IllegalArgumentException 异常
    if (!fromStart && !toEnd) {
        if (m.compare(lo, hi) > 0)
            throw new IllegalArgumentException("fromKey > toKey");
```

```
        } else {
            // 如果不从开头开始，则进行 lo 的类型校验
            if (!fromStart) // type check
                m.compare(lo, lo);
            // 如果不从结尾结束，则进行 hi 的类型校验
            if (!toEnd)
                m.compare(hi, hi);
        }

        // 赋值属性
        this.m = m;
        this.fromStart = fromStart;
        this.lo = lo;
        this.loInclusive = loInclusive;
        this.toEnd = toEnd;
        this.hi = hi;
        this.hiInclusive = hiInclusive;
    }
```

每个属性，胖友自己看代码上的注释。

16.1.2 范围校验

因为 NavigableSubMap 是 TreeMap 的子 NavigableMap ，所以其所有的操作，不能超过其子范围，既我们在创建 NavigableSubMap 时，锁设置的开始和结束的 key 位置。

#inRange(Object key) 方法，校验传入的 key 是否在子范围中。代码如下：

```
// TreeMap.java#NavigableSubMap.java

final boolean inRange(Object key) {
    return !tooLow(key)
            && !tooHigh(key);
}
```

调用 #tooLow(Object key) 方法，判断 key 是否小于 NavigableSubMap 的开始位置的 key 。代码如下：

```
// TreeMap.java#NavigableSubMap.java

final boolean tooLow(Object key) {
    if (!fromStart) {
        // 比较 key
        int c = m.compare(key, lo);
        if (c < 0 // 如果小于，则肯定过小
                || (c == 0 && !loInclusive)) // 如果相等，则进一步判断是否 !loInclusive ，不包含 lo 的情况
            return true;
    }
    return false;
}
```

调用 #tooHigh(Object key) 方法，判断 key 是否大于 NavigableSubMap 的结束位置的 key 。代码如下：

```
        // TreeMap.java#NavigableSubMap.java

        final boolean tooHigh(Object key) {
            if (!toEnd) {
                // 比较 key
                int c = m.compare(key, hi);
                if (c > 0 // 如果大于，则肯定过大
                        || (c == 0 && !hiInclusive)) // 如果相等，则进一步判断是否 !hiInclusive ，不包含 high 的情况
                    return true;
            }
            return false;
        }
```

通过这样两个判断，不过大，且不过小，那么就在范围之内了。

#inClosedRange(Object key) 方法，判断是否在闭合的范围内。代码如下：

```
        // TreeMap.java#NavigableSubMap.java

        final boolean inClosedRange(Object key) {
            return (fromStart || m.compare(key, lo) >= 0)
                && (toEnd || m.compare(hi, key) >= 0);
        }
```

也就是说，不包含包含边界的情况。

#inRange(Object key, boolean inclusive) 方法，根据传入的 inclusive 参数，调用上述的两种范围判断的方法。代码如下：

```
        // TreeMap.java#NavigableSubMap.java

        final boolean inRange(Object key, boolean inclusive) {
            return inclusive ? inRange(key) : inClosedRange(key);
        }
```

16.1.3 添加单个元素

#put(K key, V value) 方法，添加单个元素。代码如下：

```
        // TreeMap.java#NavigableSubMap.java

        public final V put(K key, V value) {
            // 校验 key 的范围，如果不在，则抛出 IllegalArgumentException 异常
            if (!inRange(key))
                throw new IllegalArgumentException("key out of range");
            // 执行添加单个元素
            return m.put(key, value);
        }
```

16.1.4 获得单个元素

#get(Object key) 方法，获得 key 对应的 value 值。代码如下：

```
// TreeMap.java#NavigableSubMap.java

public final V get(Object key) {
    return !inRange(key) // 校验 key 的范围
            ? null : // 如果不在，则返回 null
            m.get(key); // 执行获得单个元素
}
```

#containsKey(Object key) 方法，判断是否存在指定 key 。代码如下：

```
// TreeMap.java

public final boolean containsKey(Object key) {
    return inRange(key)
            && m.containsKey(key);
}
```

基于 #getEntry(key) 方法来实现。

16.1.5 删除单个元素

#remove(Object key) 方法，移除 key 对应的 Entry 节点。代码如下：

```
// TreeMap.java#NavigableSubMap.java

public final V remove(Object key) {
    return !inRange(key) // 校验 key 的范围
            ? null : // 如果不在，则返回 null
            m.remove(key); // 执行移除单个元素
}
```

16.1.6 查找接近的元素

```
// TreeMap.java#NavigableSubMap.java

public final Map.Entry<K,V> ceilingEntry(K key) {
    return exportEntry(subCeiling(key));
}
public final K ceilingKey(K key) {
    return keyOrNull(subCeiling(key));
}

public final Map.Entry<K,V> higherEntry(K key) {
    return exportEntry(subHigher(key));
}
public final K higherKey(K key) {
    return keyOrNull(subHigher(key));
}

public final Map.Entry<K,V> floorEntry(K key) {
    return exportEntry(subFloor(key));
}
public final K floorKey(K key) {
```

```
    return keyOrNull(subFloor(key));
}

public final Map.Entry<K,V> lowerEntry(K key) {
    return exportEntry(subLower(key));
}
public final K lowerKey(K key) {
    return keyOrNull(subLower(key));
}
```

因为子类的排序规则不同，所以 NavigableSubMap 定义了如下抽象方法，交给子类实现。代码如下：

```
// TreeMap.java#NavigableSubMap.java

abstract TreeMap.Entry<K,V> subLowest();
abstract TreeMap.Entry<K,V> subHighest();
abstract TreeMap.Entry<K,V> subCeiling(K key);
abstract TreeMap.Entry<K,V> subHigher(K key);
abstract TreeMap.Entry<K,V> subFloor(K key);
abstract TreeMap.Entry<K,V> subLower(K key);
```

当然，NavigableSubMap 为了子类实现更方便，提供了如下方法：

```
// TreeMap.java#NavigableSubMap.java

final TreeMap.Entry<K,V> absLowest() { // 获得 NavigableSubMap 开始位置的 Entry 节点
    TreeMap.Entry<K,V> e =
        (fromStart ?  m.getFirstEntry() : // 如果从 TreeMap 开始，则获得 TreeMap 的首个 Entry 节点
         (loInclusive ? m.getCeilingEntry(lo) : // 如果 key 从 lo 开始（包含），则获得 TreeMap 从 lo 开始（>=）最接近
                        m.getHigherEntry(lo))); // 如果 key 从 lo 开始（不包含），则获得 TreeMap 从 lo 开始(>)最接近的
    return (e == null || tooHigh(e.key)) /** 超过 key 过大 **/ ? null : e;
}

final TreeMap.Entry<K,V> absHighest() { // 获得 NavigableSubMap 结束位置的 Entry 节点
    TreeMap.Entry<K,V> e =
        (toEnd ?  m.getLastEntry() : // 如果从 TreeMap 开始，则获得 TreeMap 的尾部 Entry 节点
         (hiInclusive ?  m.getFloorEntry(hi) : // 如果 key 从 hi 开始（包含），则获得 TreeMap 从 hi 开始（<=）最接近的
                         m.getLowerEntry(hi))); // 如果 key 从 lo 开始（不包含），则获得 TreeMap 从 lo 开始(<)最接近的
    return (e == null || tooLow(e.key)) /** 超过 key 过小 **/ ? null : e;
}

final TreeMap.Entry<K,V> absCeiling(K key) { // 获得 NavigableSubMap >= key 最接近的 Entry 节点
    // 如果 key 过小，则只能通过 `#absLowest()` 方法，获得 NavigableSubMap 开始位置的 Entry 节点
    if (tooLow(key))
        return absLowest();
    // 获得 NavigableSubMap >= key 最接近的 Entry 节点
    TreeMap.Entry<K,V> e = m.getCeilingEntry(key);
    return (e == null || tooHigh(e.key)) /** 超过 key 过大 **/ ? null : e;
}

final TreeMap.Entry<K,V> absHigher(K key) { // 获得 NavigableSubMap > key 最接近的 Entry 节点
    // 如果 key 过小，则只能通过 `#absLowest()` 方法，获得 NavigableSubMap 开始位置的 Entry 节点
    if (tooLow(key))
        return absLowest();
    // 获得 NavigableSubMap > key 最接近的 Entry 节点
```

```
        TreeMap.Entry<K,V> e = m.getHigherEntry(key);
        return (e == null || tooHigh(e.key)) /** 超过 key 过大 **/ ? null : e;
    }

    final TreeMap.Entry<K,V> absFloor(K key) { // 获得 NavigableSubMap <= key 最接近的 Entry 节点
        // 如果 key 过大，则只能通过 `#absHighest()` 方法，获得 NavigableSubMap 结束位置的 Entry 节点
        if (tooHigh(key))
            return absHighest();
        // 获得 NavigableSubMap <= key 最接近的 Entry 节点
        TreeMap.Entry<K,V> e = m.getFloorEntry(key);
        return (e == null || tooLow(e.key)) /** 超过 key 过小 **/ ? null : e;
    }

    final TreeMap.Entry<K,V> absLower(K key) { // 获得 NavigableSubMap < key 最接近的 Entry 节点
        // 如果 key 过大，则只能通过 `#absHighest()` 方法，获得 NavigableSubMap 结束位置的 Entry 节点
        if (tooHigh(key))
            return absHighest();
        // 获得 NavigableSubMap < key 最接近的 Entry 节点
        TreeMap.Entry<K,V> e = m.getLowerEntry(key);
        return (e == null || tooLow(e.key)) /** 超过 key 过小 **/ ? null : e;
    }

    /** Returns the absolute high fence for ascending traversal */
    final TreeMap.Entry<K,V> absHighFence() { // 获得 TreeMap 最大 key 的 Entry 节点，用于升序遍历的时候。注意，是 TreeMa
        // toEnd 为真时，意味着无限大，所以返回 null
        return (toEnd ? null : (hiInclusive ?
                                m.getHigherEntry(hi) : // 获得 TreeMap > hi 最接近的 Entry 节点。
                                m.getCeilingEntry(hi))); // 获得 TreeMap => hi 最接近的 Entry 节点。
    }

    /** Return the absolute low fence for descending traversal  */
    final TreeMap.Entry<K,V> absLowFence() { // 获得 TreeMap 最小 key 的 Entry 节点，用于降序遍历的时候。注意，是 TreeMap
        return (fromStart ? null : (loInclusive ?
                                    m.getLowerEntry(lo) :  // 获得 TreeMap < lo 最接近的 Entry 节点。
                                    m.getFloorEntry(lo))); // 获得 TreeMap <= lo 最接近的 Entry 节点。
    }
```

方法有点点小多，不过基本是雷同的。耐心如我~

16.1.7 获得首尾的元素

`#firstEntry()` 方法，获得首个 Entry 节点。代码如下：

```
    // TreeMap.java#NavigableSubMap.java

    public final Map.Entry<K,V> firstEntry() {
        return exportEntry(subLowest());
    }

    public final K firstKey() {
        return key(subLowest());
    }

    public final Map.Entry<K,V> pollFirstEntry() {
        // 获得 NavigableSubMap 的首个 Entry 节点
        TreeMap.Entry<K,V> e = subLowest();
        Map.Entry<K,V> result = exportEntry(e);
```

```
            // 如果存在，则进行删除。
            if (e != null)
                m.deleteEntry(e);
            return result;
        }
```

#lastEntry() 方法，获得尾部 Entry 节点。代码如下：

```
        // TreeMap.java#NavigableSubMap.java

        public final Map.Entry<K,V> lastEntry() {
            return exportEntry(subHighest());
        }

        public final K lastKey() {
            return key(subHighest());
        }

        public final Map.Entry<K,V> pollLastEntry() {
            // 获得 NavigableSubMap 的尾部 Entry 节点
            TreeMap.Entry<K,V> e = subHighest();
            Map.Entry<K,V> result = exportEntry(e);
            // 如果存在，则进行删除。
            if (e != null)
                m.deleteEntry(e);
            return result;
        }
```

16.1.8 清空

直接使用继承自 AbstractMap 的 #clear() 方法，仅清空自己范围内的数据。代码如下：

```
        // AbstractMap.java

        public void clear() {
            entrySet().clear();
        }
```

而 #entrySet() 方法，NavigableSubMap 的子类在实现时，会重写该方法。这样，能够保证仅清空自己范围内的数据。

16.1.9 获得迭代器

SubMapIterator ，实现 Iterator 接口，提供了 NavigableSubMap 的通用实现 Iterator 的抽象类。代码如下：

```
        // TreeMap.java#NavigableSubMap.java

        abstract class SubMapIterator<T> implements Iterator<T> {

            /**
             * 最后返回的节点
```

```
 */
TreeMap.Entry<K,V> lastReturned;
/**
 * 下一个节点
 */
TreeMap.Entry<K,V> next;
/**
 * 遍历的上限 key 。
 *
 * 如果遍历到该 key ，说明已经超过范围了
 */
final Object fenceKey;
/**
 * 当前的修改次数
 */
int expectedModCount;

SubMapIterator(TreeMap.Entry<K,V> first,
               TreeMap.Entry<K,V> fence) {
    expectedModCount = m.modCount;
    lastReturned = null;
    next = first;
    fenceKey = fence == null ? UNBOUNDED /** 无界限 **/ : fence.key;
}

public final boolean hasNext() { // 是否还有下一个节点
    return next != null && next.key != fenceKey;
}

final TreeMap.Entry<K,V> nextEntry() { // 获得下一个 Entry 节点
    // 记录当前节点
    TreeMap.Entry<K,V> e = next;
    // 如果没有下一个，抛出 NoSuchElementException 异常
    if (e == null || e.key == fenceKey)
        throw new NoSuchElementException();
    // 如果发生了修改，抛出 ConcurrentModificationException 异常
    if (m.modCount != expectedModCount)
        throw new ConcurrentModificationException();
    // 获得 e 的后继节点，赋值给 next
    next = successor(e);
    // 记录最后返回的节点
    lastReturned = e;
    // 返回当前节点
    return e;
}

final TreeMap.Entry<K,V> prevEntry() { // 获得前一个 Entry 节点
    // 记录当前节点
    TreeMap.Entry<K,V> e = next;
    // 如果没有下一个，抛出 NoSuchElementException 异常
    if (e == null || e.key == fenceKey)
        throw new NoSuchElementException();
    // 如果发生了修改，抛出 ConcurrentModificationException 异常
    if (m.modCount != expectedModCount)
        throw new ConcurrentModificationException();
    // 获得 e 的前继节点，赋值给 next
    next = predecessor(e);
    // 记录最后返回的节点
    lastReturned = e;
    // 返回当前节点
```

```
            return e;
        }

        final void removeAscending() { // 删除节点（顺序遍历的情况下）
            // 如果当前返回的节点不存在，则抛出 IllegalStateException 异常
            if (lastReturned == null)
                throw new IllegalStateException();
            // 如果发生了修改，抛出 ConcurrentModificationException 异常
            if (m.modCount != expectedModCount)
                throw new ConcurrentModificationException();
            // deleted entries are replaced by their successors
            // 在 lastReturned 左右节点都存在的时候，实际在 deleteEntry 方法中，是将后继节点替换到 lastReturned 中
            // 因此，next 需要指向 lastReturned
            if (lastReturned.left != null && lastReturned.right != null)
                next = lastReturned;
            // 删除节点
            m.deleteEntry(lastReturned);
            // 置空 lastReturned
            lastReturned = null;
            // 记录新的修改次数
            expectedModCount = m.modCount;
        }

        final void removeDescending() { // 删除节点倒序遍历的情况下）
            // 如果当前返回的节点不存在，则抛出 IllegalStateException 异常
            if (lastReturned == null)
                throw new IllegalStateException();
            // 如果发生了修改，抛出 ConcurrentModificationException 异常
            if (m.modCount != expectedModCount)
                throw new ConcurrentModificationException();
            // 删除节点
            m.deleteEntry(lastReturned);
            // 置空 lastReturned
            lastReturned = null;
            // 记录新的修改次数
            expectedModCount = m.modCount;
        }

    }
```

整体代码比较简单，胖友自己看看艿艿写的注释噢。

16.1.9.1 SubMapKeyIterator

SubMapKeyIterator ，继承 SubMapIterator 抽象类，key 的正序迭代器。代码如下：

```
    // TreeMap.java#NavigableSubMap.java

    final class SubMapKeyIterator extends SubMapIterator<K>
        implements Spliterator<K> {

        SubMapKeyIterator(TreeMap.Entry<K,V> first,
                          TreeMap.Entry<K,V> fence) {
            super(first, fence);
        }

        // 实现 next 方法，实现正序
```

```
        public K next() {
            return nextEntry().key;
        }

        // 实现 remove 方法，实现正序的移除方法
        public void remove() {
            removeAscending();
        }

        public Spliterator<K> trySplit() {
            return null;
        }
        public void forEachRemaining(Consumer<? super K> action) {
            while (hasNext())
                action.accept(next());
        }
        public boolean tryAdvance(Consumer<? super K> action) {
            if (hasNext()) {
                action.accept(next());
                return true;
            }
            return false;
        }
        public long estimateSize() {
            return Long.MAX_VALUE;
        }
        public int characteristics() {
            return Spliterator.DISTINCT | Spliterator.ORDERED |
                Spliterator.SORTED;
        }
        public final Comparator<? super K>  getComparator() {
            return NavigableSubMap.this.comparator();
        }
    }
```

16.1.9.2 DescendingSubMapKeyIterator

DescendingSubMapKeyIterator ，继承 SubMapIterator 抽象类，key 的倒序迭代器。代码如下：

```
    // TreeMap.java#NavigableSubMap.java

    final class DescendingSubMapKeyIterator extends SubMapIterator<K>
        implements Spliterator<K> {

        DescendingSubMapKeyIterator(TreeMap.Entry<K,V> last,
                                    TreeMap.Entry<K,V> fence) {
            super(last, fence);
        }

        // 实现 next 方法，实现倒序
        public K next() {
            return prevEntry().key;
        }

        // 实现 remove 方法，实现倒序的移除方法
        public void remove() {
            removeDescending();
```

```
        }

        public Spliterator<K> trySplit() {
            return null;
        }
        public void forEachRemaining(Consumer<? super K> action) {
            while (hasNext())
                action.accept(next());
        }
        public boolean tryAdvance(Consumer<? super K> action) {
            if (hasNext()) {
                action.accept(next());
                return true;
            }
            return false;
        }
        public long estimateSize() {
            return Long.MAX_VALUE;
        }
        public int characteristics() {
            return Spliterator.DISTINCT | Spliterator.ORDERED;
        }
    }
```

16.1.9.3 SubMapEntryIterator

SubMapEntryIterator ，继承 SubMapIterator 抽象类，Entry 的正序迭代器。代码如下：

```
    // TreeMap.java#NavigableSubMap.java

    final class SubMapEntryIterator extends SubMapIterator<Map.Entry<K,V>> {

        SubMapEntryIterator(TreeMap.Entry<K,V> first,
                            TreeMap.Entry<K,V> fence) {
            super(first, fence);
        }

        // 实现 next 方法，实现正序
        public Map.Entry<K,V> next() {
            return nextEntry();
        }

        // 实现 remove 方法，实现正序的移除方法
        public void remove() {
            removeAscending();
        }

    }
```

16.1.9.4 DescendingSubMapEntryIterator

DescendingSubMapEntryIterator ，继承 SubMapIterator 抽象类，Entry 的倒序迭代器。代码如下：

```
// TreeMap.java#NavigableSubMap.java

final class DescendingSubMapEntryIterator extends SubMapIterator<Map.Entry<K,V>> {

    DescendingSubMapEntryIterator(TreeMap.Entry<K,V> last,
                                  TreeMap.Entry<K,V> fence) {
        super(last, fence);
    }

    // 实现 next 方法，实现倒序
    public Map.Entry<K,V> next() {
        return prevEntry();
    }

    // 实现 remove 方法，实现倒序的移除方法
    public void remove() {
        removeDescending();
    }
}
```

16.1.10 转换成 Set/Collection

16.1.10.1 keySet

#keySet() 方法，获得正序的 key Set 。代码如下：

```
// TreeMap.java#NavigableSubMap.java

/**
 * 正序的 KeySet 缓存对象
 */
transient KeySet<K> navigableKeySetView;

public final Set<K> keySet() {
    return navigableKeySet();
}

public final NavigableSet<K> navigableKeySet() {
    KeySet<K> nksv = navigableKeySetView;
    return (nksv != null) ? nksv :
        (navigableKeySetView = new TreeMap.KeySet<>(this));
}
```

KeySet 使用的迭代器，就是 「16.1.9.1 SubMapKeyIterator」 。

16.1.10.2 navigableKeySet

#navigableKeySet() 方法，获得倒序的 key Set 。代码如下：

```
// TreeMap.java#NavigableSubMap.java

/**
 * 倒序的 KeySet 缓存对象
 */
```

```
transient NavigableMap<K,V> descendingMapView;

public NavigableSet<K> descendingKeySet() {
    return descendingMap().navigableKeySet();
}
```

其中，#descendingMap() 方法，子类自己实现。

16.1.10.3 values

直接使用继承自 AbstractMap 的 #values() 方法，代码如下：

```
// AbstractMap.java

transient Collection<V> values;

public Collection<V> values() {
    Collection<V> vals = values;
    if (vals == null) {
        vals = new AbstractCollection<V>() {
            public Iterator<V> iterator() {
                return new Iterator<V>() {
                    private Iterator<Entry<K,V>> i = entrySet().iterator();

                    public boolean hasNext() {
                        return i.hasNext();
                    }

                    public V next() {
                        return i.next().getValue();
                    }

                    public void remove() {
                        i.remove();
                    }
                };
            }

            public int size() {
                return AbstractMap.this.size();
            }

            public boolean isEmpty() {
                return AbstractMap.this.isEmpty();
            }

            public void clear() {
                AbstractMap.this.clear();
            }

            public boolean contains(Object v) {
                return AbstractMap.this.containsValue(v);
            }
        };
        values = vals;
    }
    return vals;
```

```
}
```

也是基于 #entrySet() 方法，来实现。

16.1.10.4 entrySet

NavigableSubMap 未提供 #entrySet() 方法的实现，不过提供了 EntrySetView 抽象类，它实现了 java.util.AbstractSet 抽像类，是 NavigableSubMap 的内部类。比较简单，就不哔哔了。

16.1.11 查找范围的元素

```
// TreeMap.java#NavigableSubMap.java

public final SortedMap<K,V> subMap(K fromKey, K toKey) {
    return subMap(fromKey, true, toKey, false);
}

public final SortedMap<K,V> headMap(K toKey) {
    return headMap(toKey, false);
}

public final SortedMap<K,V> tailMap(K fromKey) {
    return tailMap(fromKey, true);
}
```

每个方法内部调用的方法，都是子类来实现。

16.2 AscendingSubMap

AscendingSubMap ，继承 NavigableSubMap 抽象类，正序的 子 NavigableMap 的实现类。

16.2.1 查找接近的元素

```
// TreeMap.java#AscendingSubMap.java

TreeMap.Entry<K,V> subLowest()       { return absLowest(); }
TreeMap.Entry<K,V> subHighest()      { return absHighest(); }
TreeMap.Entry<K,V> subCeiling(K key) { return absCeiling(key); }
TreeMap.Entry<K,V> subHigher(K key)  { return absHigher(key); }
TreeMap.Entry<K,V> subFloor(K key)   { return absFloor(key); }
TreeMap.Entry<K,V> subLower(K key)   { return absLower(key); }
```

16.2.2 获得迭代器

```
// TreeMap.java#AscendingSubMap.java

Iterator<K> keyIterator() {
    return new SubMapKeyIterator(absLowest(), absHighFence());
}
```

```
        Iterator<K> descendingKeyIterator() {
            return new DescendingSubMapKeyIterator(absHighest(), absLowFence());
        }
```

16.2.3 转换成 Set/Collection

16.2.3.1 descendingMap

#descendingMap() 方法，获得倒序 descendingMap。代码如下：

```
        // TreeMap.java#AscendingSubMap.java

        public NavigableMap<K,V> descendingMap() {
            NavigableMap<K,V> mv = descendingMapView;
            return (mv != null) ? mv :
                (descendingMapView =
                 new DescendingSubMap<>(m,
                                        fromStart, lo, loInclusive,
                                        toEnd,     hi, hiInclusive));
        }
```

该方法，会被 「16.1.9.2 DescendingSubMapKeyIterator」 调用。

16.2.3.2 entrySet

#entrySet() 方法，获得 Entry 集合。代码如下：

```
        // TreeMap.java#AscendingSubMap.java

        public Set<Map.Entry<K,V>> entrySet() {
            EntrySetView es = entrySetView;
            return (es != null) ? es : (entrySetView = new AscendingEntrySetView());
        }

        final class AscendingEntrySetView extends EntrySetView {
            public Iterator<Map.Entry<K,V>> iterator() {
                return new SubMapEntryIterator(absLowest(), absHighFence());
            }
        }
```

AscendingEntrySetView 使用的迭代器，就是 「16.1.9.3 SubMapEntryIterator」 。

16.2.4 查找范围的元素

```
        // TreeMap.java#AscendingSubMap.java

        public NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,
                                        K toKey,   boolean toInclusive) {
            // 如果不在范围，抛出 IllegalArgumentException 异常
            if (!inRange(fromKey, fromInclusive))
```

```
            throw new IllegalArgumentException("fromKey out of range");
        // 如果不在范围，抛出 IllegalArgumentException 异常
        if (!inRange(toKey, toInclusive))
            throw new IllegalArgumentException("toKey out of range");
        // 创建 AscendingSubMap 对象
        return new AscendingSubMap<>(m,
                                     false, fromKey, fromInclusive,
                                     false, toKey,   toInclusive);
    }

    public NavigableMap<K,V> headMap(K toKey, boolean inclusive) {
        // 如果不在范围，抛出 IllegalArgumentException 异常
        if (!inRange(toKey, inclusive))
            throw new IllegalArgumentException("toKey out of range");
        // 创建 AscendingSubMap 对象
        return new AscendingSubMap<>(m,
                                     fromStart, lo,    loInclusive,
                                     false,     toKey, inclusive);
    }

    public NavigableMap<K,V> tailMap(K fromKey, boolean inclusive) {
        // 如果不在范围，抛出 IllegalArgumentException 异常
        if (!inRange(fromKey, inclusive))
            throw new IllegalArgumentException("fromKey out of range");
        // 创建 AscendingSubMap 对象
        return new AscendingSubMap<>(m,
                                     false, fromKey, inclusive,
                                     toEnd, hi,      hiInclusive);
    }
```

16.2.5 获得排序器

#comparator() 方法，代码如下：

```
    // TreeMap.java#AscendingSubMap.java

    // 排序器，使用传入的 TreeMap
    public Comparator<? super K> comparator() {
        return m.comparator();
    }
```

16.3 DescendingSubMap

DescendingSubMap ，继承 NavigableSubMap 抽象类，倒序的 子 NavigableMap 的实现类。

16.3.1 查找接近的元素

```
    // TreeMap.java#DescendingSubMap.java

    TreeMap.Entry<K,V> subLowest()       { return absHighest(); }
    TreeMap.Entry<K,V> subHighest()      { return absLowest(); }
    TreeMap.Entry<K,V> subCeiling(K key) { return absFloor(key); }
    TreeMap.Entry<K,V> subHigher(K key)  { return absLower(key); }
```

```
TreeMap.Entry<K,V> subFloor(K key)    { return absCeiling(key); }
TreeMap.Entry<K,V> subLower(K key)    { return absHigher(key); }
```

恰好是反过来。

16.3.2 获得迭代器

```
// TreeMap.java#DescendingSubMap.java

Iterator<K> keyIterator() {
    return new DescendingSubMapKeyIterator(absHighest(), absLowFence());
}

Iterator<K> descendingKeyIterator() {
    return new SubMapKeyIterator(absLowest(), absHighFence());
}
```

恰好是反过来。

16.3.3 转换成 Set/Collection

16.3.3.1 descendingMap

#descendingMap() 方法，获得倒序 descendingMap。代码如下：

负负得正，其实返回的是正序的。

```
// TreeMap.java#DescendingSubMap.java

public NavigableMap<K,V> descendingMap() {
    NavigableMap<K,V> mv = descendingMapView;
    return (mv != null) ? mv :
        (descendingMapView =
         new AscendingSubMap<>(m,
                               fromStart, lo, loInclusive,
                               toEnd,     hi, hiInclusive));
}
```

该方法，会被 「16.1.9.2 DescendingSubMapKeyIterator」 调用。

16.3.3.2 entrySet

#entrySet() 方法，获得 Entry 集合。代码如下：

```
// TreeMap.java#DescendingSubMap.java

public Set<Map.Entry<K,V>> entrySet() {
    EntrySetView es = entrySetView;
    return (es != null) ? es : (entrySetView = new DescendingEntrySetView());
}
```

```
final class DescendingEntrySetView extends EntrySetView {
    public Iterator<Map.Entry<K,V>> iterator() {
        return new DescendingSubMapEntryIterator(absHighest(), absLowFence());
    }
}
```

AscendingEntrySetView 使用的迭代器，就是 「16.1.9.4 DescendingSubMapEntryIterator」 。

16.3.4 查找范围的元素

```
// TreeMap.java#DescendingSubMap.java

public NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,
                                K toKey,   boolean toInclusive) {
    // 如果不在范围，抛出 IllegalArgumentException 异常
    if (!inRange(fromKey, fromInclusive))
        throw new IllegalArgumentException("fromKey out of range");
    // 如果不在范围，抛出 IllegalArgumentException 异常
    if (!inRange(toKey, toInclusive))
        throw new IllegalArgumentException("toKey out of range");
    // 创建 DescendingSubMap 对象
    return new DescendingSubMap<>(m,
                                  false, toKey,   toInclusive,
                                  false, fromKey, fromInclusive);
}

public NavigableMap<K,V> headMap(K toKey, boolean inclusive) {
    // 如果不在范围，抛出 IllegalArgumentException 异常
    if (!inRange(toKey, inclusive))
        throw new IllegalArgumentException("toKey out of range");
    // 创建 DescendingSubMap 对象
    return new DescendingSubMap<>(m,
                                  false, toKey, inclusive,
                                  toEnd, hi,    hiInclusive);
}

public NavigableMap<K,V> tailMap(K fromKey, boolean inclusive) {
    // 如果不在范围，抛出 IllegalArgumentException 异常
    if (!inRange(fromKey, inclusive))
        throw new IllegalArgumentException("fromKey out of range");
    // 创建 DescendingSubMap 对象
    return new DescendingSubMap<>(m,
                                  fromStart, lo, loInclusive,
                                  false, fromKey, inclusive);
}
```

16.3.5 获得排序器

#comparator() 方法，代码如下：

```
// TreeMap.java#DescendingSubMap.java

/**
```

```
     * 倒序排序器
     */
    private final Comparator<? super K> reverseComparator =
        Collections.reverseOrder(m.comparator);

    public Comparator<? super K> comparator() {
        return reverseComparator;
    }
```

666. 彩蛋

抛开红黑树的自平衡的逻辑来说，TreeMap 的实现代码，实际是略简单于 HashMap 的。当然，因为 TreeMap 提供的方法较多，所以导致本文会比 HashMap 写的长一些。

下面，我们来对 TreeMap 做一个简单的小结：

> TreeMap 按照 key 的顺序的 Map 实现类，底层采用红黑树来实现存储。
> TreeMap 因为采用树结构，所以无需初始考虑像 HashMap 考虑容量问题，也不存在扩容问题。
>
> TreeMap 的 key 不允许为空（ null ），可能是因为红黑树是一颗二叉查找树，需要对 key 进行排序。
>
> > 看了下 《为什么 TreeMap 中不允许使用 null 键？》 文章，也是这个观点。
>
> TreeMap 的查找、添加、删除 key-value 键值对的平均时间复杂度为 O(logN) 。原因是，TreeMap 采用红黑树，操作都需要经过二分查找，而二分查找的时间复杂度是 O(logN) 。
>
> 相比 HashMap 来说，TreeMap 不仅仅支持指定 key 的查找，也支持 key 范围的查找。当然，这也得益于 TreeMap 数据结构能够提供的有序特性。

文章目录

回到首页