参考：<http://blog.csdn.net/ns_code/article/details/36421085>

[**【Java集合源码剖析】TreeMap源码剖析**](http://blog.csdn.net/ns_code/article/details/36421085)

**转载请注明出处：**[**http://blog.csdn.net/ns\_code/article/details/36421085**](http://blog.csdn.net/ns_code/article/details/36421085)

**前言**

    本文不打算延续前几篇的风格（对所有的源码加入注释），因为要理解透TreeMap的所有源码，对博主来说，确实需要耗费大量的时间和经历，目前看来不大可能有这么多时间的投入，故这里意在通过阅读源码对TreeMap有个宏观上的把握，并就其中一些方法的实现做比较深入的分析。

**红黑树简介**

    TreeMap是基于红黑树实现的，这里只对红黑树做个简单的介绍，红黑树是一种特殊的二叉排序树，关于二叉排序树，参见：<http://blog.csdn.net/ns_code/article/details/19823463>，红黑树通过一些限制，使其不会出现二叉树排序树中极端的一边倒的情况，相对二叉排序树而言，这自然提高了查询的效率。

    二叉排序树的基本性质如下：

    1、每个节点都只能是红色或者黑色

    2、根节点是黑色

    3、每个叶节点（NIL节点，空节点）是黑色的。

    4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

    5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

    正是这些性质的限制，使得红黑树中任一节点到其子孙叶子节点的最长路径不会长于最短路径的2倍，因此它是一种接近平衡的二叉树。

    说到红黑树，自然不免要和AVL树对比一番。相比较而言，AVL树是严格的平衡二叉树，而红黑树不算严格意义上的平衡二叉树，只是接近平衡，不会让树的高度如BST极端情况那样等于节点的个数。其实能用到红黑树的地方，也都可以用AVL树来实现，但红黑树的应用却非常广泛，而AVL树则很少被使用。在执行插入、删除操作时，AVL树需要调整的次数一般要比红黑树多（红黑树的旋转调整最多只需三次），效率相对较低，且红黑树的统计性能较AVL树要好，当然AVL树在查询效率上可能更胜一筹，但实际上也高不了多少。

    红黑树的插入删除操作很简单，就是单纯的二叉排序树的插入删除操作。红黑树被认为比较变态的地方自然在于插入删除后对红黑树的调整操作（旋转和着色），主要是情况分的很多，限于篇幅及博主的熟悉程度优先，这里不打算详细介绍插入删除后调整红黑树的各种情况及其实现，我们有个宏观上的了解即可，如须详细了解，参见算法导论或一些相关的资料。

**TreeMap源码剖析**

**存储结构**

    TreeMap的排序是基于对key的排序实现的，它的每一个Entry代表红黑树的一个节点，Entry的数据结构如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **static** **final** **class** Entry<K,V> **implements** Map.Entry<K,V> {
2. // 键
3. K key;
4. // 值
5. V value;
6. // 左孩子
7. Entry<K,V> left = **null**;
8. // 右孩子
9. Entry<K,V> right = **null**;
10. // 父节点
11. Entry<K,V> parent;
12. // 当前节点颜色
13. **boolean** color = BLACK;
15. // 构造函数
16. Entry(K key, V value, Entry<K,V> parent) {
17. **this**.key = key;
18. **this**.value = value;
19. **this**.parent = parent;
20. }
22. 。。。。。
23. }

**构造方法**

    先来看下TreeMap的构造方法。TreeMap一共有4个构造方法。

    1、无参构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap() {
2. comparator = **null**;
3. }

    采用无参构造方法，不指定比较器，这时候，排序的实现要依赖key.compareTo()方法，因此key必须实现Comparable接口，并覆写其中的compareTo方法。

    2、带有比较器的构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap(Comparator<? **super** K> comparator) {
2. **this**.comparator = comparator;
3. }

    采用带比较器的构造方法，这时候，排序依赖该比较器，key可以不用实现Comparable接口。

    3、带Map的构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap(Map<? **extends** K, ? **extends** V> m) {
2. comparator = **null**;
3. putAll(m);
4. }

    该构造方法同样不指定比较器，调用putAll方法将Map中的所有元素加入到TreeMap中。putAll的源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. // 将map中的全部节点添加到TreeMap中
2. **public** **void** putAll(Map<? **extends** K, ? **extends** V> map) {
3. // 获取map的大小
4. **int** mapSize = map.size();
5. // 如果TreeMap的大小是0,且map的大小不是0,且map是已排序的“key-value对”
6. **if** (size==0 && mapSize!=0 && map **instanceof** SortedMap) {
7. Comparator c = ((SortedMap)map).comparator();
8. // 如果TreeMap和map的比较器相等；
9. // 则将map的元素全部拷贝到TreeMap中，然后返回！
10. **if** (c == comparator || (c != **null** && c.equals(comparator))) {
11. ++modCount;
12. **try** {
13. buildFromSorted(mapSize, map.entrySet().iterator(),
14. **null**, **null**);
15. } **catch** (java.io.IOException cannotHappen) {
16. } **catch** (ClassNotFoundException cannotHappen) {
17. }
18. **return**;
19. }
20. }
21. // 调用AbstractMap中的putAll();
22. // AbstractMap中的putAll()又会调用到TreeMap的put()
23. **super**.putAll(map);
24. }

    显然，如果Map里的元素是排好序的，就调用buildFromSorted方法来拷贝Map中的元素，这在下一个构造方法中会重点提及，而如果Map中的元素不是排好序的，就调用AbstractMap的putAll(map)方法，该方法源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** **void** putAll(Map<? **extends** K, ? **extends** V> m) {
2. **for** (Map.Entry<? **extends** K, ? **extends** V> e : m.entrySet())
3. put(e.getKey(), e.getValue());
4. }

    很明显它是将Map中的元素一个个put（插入）到TreeMap中的，主要因为Map中的元素是无序存放的，因此要一个个插入到红黑树中，使其有序存放，并满足红黑树的性质。

    4、带有SortedMap的构造方法

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** TreeMap(SortedMap<K, ? **extends** V> m) {
2. comparator = m.comparator();
3. **try** {
4. buildFromSorted(m.size(), m.entrySet().iterator(), **null**, **null**);
5. } **catch** (java.io.IOException cannotHappen) {
6. } **catch** (ClassNotFoundException cannotHappen) {
7. }
8. }

    首先将比较器指定为m的比较器，这取决于生成m时调用构造方法是否传入了指定的构造器，而后调用buildFromSorted方法，将SortedMap中的元素插入到TreeMap中，由于SortedMap中的元素师有序的，实际上它是根据SortedMap创建的TreeMap，将SortedMap中对应的元素添加到TreeMap中。

**插入删除**

    插入操作即对应TreeMap的put方法，put操作实际上只需按照二叉排序树的插入步骤来操作即可，插入到指定位置后，再做调整，使其保持红黑树的特性。put源码的实现：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. **public** V put(K key, V value) {
2. Entry<K,V> t = root;
3. // 若红黑树为空，则插入根节点
4. **if** (t == **null**) {
5. // TBD:
6. // 5045147: (coll) Adding null to an empty TreeSet should
7. // throw NullPointerException
8. //
9. // compare(key, key); // type check
10. root = **new** Entry<K,V>(key, value, **null**);
11. size = 1;
12. modCount++;
13. **return** **null**;
14. }
15. **int** cmp;
16. Entry<K,V> parent;
17. // split comparator and comparable paths
18. Comparator<? **super** K> cpr = comparator;
19. // 找出(key, value)在二叉排序树中的插入位置。
20. // 红黑树是以key来进行排序的，所以这里以key来进行查找。
21. **if** (cpr != **null**) {
22. **do** {
23. parent = t;
24. cmp = cpr.compare(key, t.key);
25. **if** (cmp < 0)
26. t = t.left;
27. **else** **if** (cmp > 0)
28. t = t.right;
29. **else**
30. **return** t.setValue(value);
31. } **while** (t != **null**);
32. }
33. **else** {
34. **if** (key == **null**)
35. **throw** **new** NullPointerException();
36. Comparable<? **super** K> k = (Comparable<? **super** K>) key;
37. **do** {
38. parent = t;
39. cmp = k.compareTo(t.key);
40. **if** (cmp < 0)
41. t = t.left;
42. **else** **if** (cmp > 0)
43. t = t.right;
44. **else**
45. **return** t.setValue(value);
46. } **while** (t != **null**);
47. }
48. // 为（key-value）新建节点
49. Entry<K,V> e = **new** Entry<K,V>(key, value, parent);
50. **if** (cmp < 0)
51. parent.left = e;
52. **else**
53. parent.right = e;
54. // 插入新的节点后，调用fixAfterInsertion调整红黑树。
55. fixAfterInsertion(e);
56. size++;
57. modCount++;
58. **return** **null**;
59. }

    这里的fixAfterInsertion便是节点插入后对树进行调整的方法，这里不做介绍。  
    删除操作及对应TreeMap的deleteEntry方法，deleteEntry方法同样也只需按照二叉排序树的操作步骤实现即可，删除指定节点后，再对树进行调整即可。deleteEntry方法的实现源码如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/36421085) [copy](http://blog.csdn.net/ns_code/article/details/36421085)

1. // 删除“红黑树的节点p”
2. **private** **void** deleteEntry(Entry<K,V> p) {
3. modCount++;
4. size--;
6. **if** (p.left != **null** && p.right != **null**) {
7. Entry<K,V> s = successor (p);
8. p.key = s.key;
9. p.value = s.value;
10. p = s;
11. }
13. Entry<K,V> replacement = (p.left != **null** ? p.left : p.right);
15. **if** (replacement != **null**) {
16. replacement.parent = p.parent;
17. **if** (p.parent == **null**)
18. root = replacement;
19. **else** **if** (p == p.parent.left)
20. p.parent.left  = replacement;
21. **else**
22. p.parent.right = replacement;
24. p.left = p.right = p.parent = **null**;
26. **if** (p.color == BLACK)
27. fixAfterDeletion(replacement);
28. } **else** **if** (p.parent == **null**) {
29. root = **null**;
30. } **else** {
31. **if** (p.color == BLACK)
32. fixAfterDeletion(p);
34. **if** (p.parent != **null**) {
35. **if** (p == p.parent.left)
36. p.parent.left = **null**;
37. **else** **if** (p == p.parent.right)
38. p.parent.right = **null**;
39. p.parent = **null**;
40. }
41. }
42. }

    后面的fixAfterDeletion方法便是节点删除后对树进行调整的方法，这里不做介绍。

    其他很多方法这里不再一一介绍。

**几点总结**

    本文对TreeMap的分析较前几篇文章有些浅尝辄止，TreeMap用的没有HashMap那么多，我们有个宏观上的把我和比较即可。

    1、TreeMap是根据key进行排序的，它的排序和定位需要依赖比较器或覆写Comparable接口，也因此不需要key覆写hashCode方法和equals方法，就可以排除掉重复的key，而HashMap的key则需要通过覆写hashCode方法和equals方法来确保没有重复的key。

    2、TreeMap的查询、插入、删除效率均没有HashMap高，一般只有要对key排序时才使用TreeMap。

    3、TreeMap的key不能为null，而HashMap的key可以为null。

    注：对TreeSet和HashSet的源码不再进行剖析，二者分别是基于TreeMap和HashMap实现的，只是对应的节点中只有key，而没有value，因此对TreeMap和HashMap比较了解的话，对TreeSet和HashSet的理解就会非常容易。