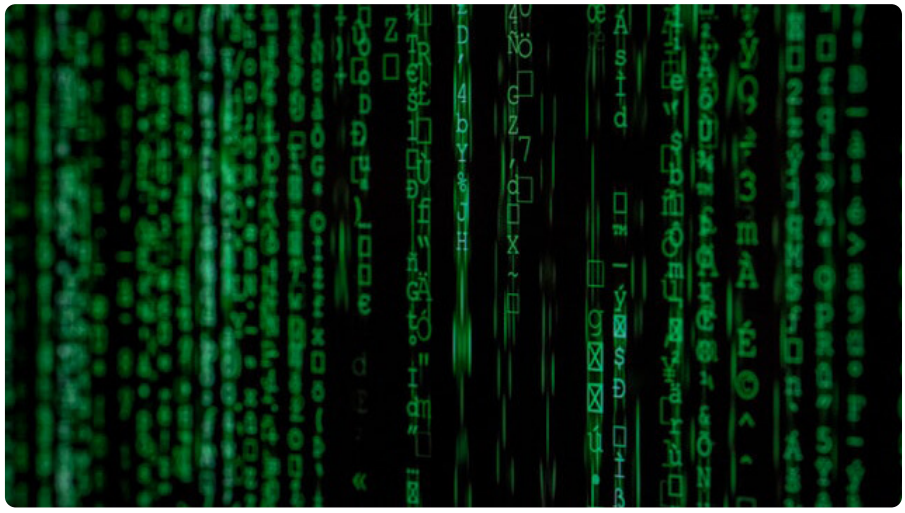
目录

09 TreeMap 和 LinkedHashMap 核心源码解析

更新时间：2019-09-05 10:15:03



> 人的影响短暂而微弱，书的影响则广泛而深远。
>
> ——普希金

引导语

在熟悉 HashMap 之后，本小节我们来看下 TreeMap 和 LinkedHashMap，看看 TreeMap 是如何根据 key 进行排序的，LinkedHashMap 是如何用两种策略进行访问的。

1 知识储备

在了解 TreeMap 之前，我们来看下日常工作中排序的两种方式，作为我们学习的基础储备，两种方式的代码如下：

```
public class TreeMapDemo {

  @Data
  // DTO 为我们排序的对象
  class DTO implements Comparable<DTO> {
    private Integer id;
    public DTO(Integer id) {
      this.id = id;
    }

    @Override
    public int compareTo(DTO o) {
      //默认从小到大排序
      return id - o.getId();
    }
  }

  @Test
  public void testTwoComparable() {
```

```
    list.add(new DTO(i));
  }
  Collections.sort(list);
  log.info(JSON.toJSONString(list));

  // 第二种排序，从大到小排序，利用外部排序器 Comparator 进行排序
  Comparator comparator = (Comparator<DTO>) (o1, o2) -> o2.getId() - o1.getId();
  List<DTO> list2 = new ArrayList<>();
  for (int i = 5; i > 0; i--) {
    list2.add(new DTO(i));
  }
  Collections.sort(list,comparator);
  log.info(JSON.toJSONString(list2));
 }
}
```

第一种排序输出的结果从小到大，结果是：[{“id”:1},{“id”:2},{“id”:3},{“id”:4},{“id”:5}]；

第二种输出的结果恰好相反，结果是：[{“id”:5},{“id”:4},{“id”:3},{“id”:2},{“id”:1}]。

以上两种就是分别通过 Comparable 和 Comparator 两者进行排序的方式，而 TreeMap 利用的也是此原理，从而实现了对 key 的排序，我们一起来看下。

2 TreeMap 整体架构

TreeMap 底层的数据结构就是红黑树，和 HashMap 的红黑树结构一样。

不同的是，TreeMap 利用了红黑树左节点小，右节点大的性质，根据 key 进行排序，使每个元素能够插入到红黑树大小适当的位置，维护了 key 的大小关系，适用于 key 需要排序的场景。

因为底层使用的是平衡红黑树的结构，所以 containsKey、get、put、remove 等方法的时间复杂度都是 log(n)。

2.1 属性

TreeMap 常见的属性有：

```
//比较器，如果外部有传进来 Comparator 比较器，首先用外部的
//如果外部比较器为空，则使用 key 自己实现的 Comparable#compareTo 方法
//比较手段和上面日常工作中的比较 demo 是一致的
private final Comparator<? super K> comparator;

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

//红黑树的已有元素大小
private transient int size = 0;

//树结构变化的版本号，用于迭代过程中的快速失败场景
private transient int modCount = 0;

//红黑树的节点
static final class Entry<K,V> implements Map.Entry<K,V> {}
```

我们来看下 TreeMap 新增节点的步骤：

1. 判断红黑树的节点是否为空，为空的话，新增的节点直接作为根节点，代码如下：

```
Entry<K,V> t = root;
//红黑树根节点为空，直接新建
if (t == null) {
    // compare 方法限制了 key 不能为 null
    compare(key, key); // type (and possibly null) check
    // 成为根节点
    root = new Entry<>(key, value, null);
    size = 1;
    modCount++;
    return null;
}
```

2. 根据红黑树左小右大的特性，进行判断，找到应该新增节点的父节点，代码如下：

```
Comparator<? super K> cpr = comparator;
if (cpr != null) {
    //自旋找到 key 应该新增的位置，就是应该挂载那个节点的头上
    do {
        //一次循环结束时，parent 就是上次比过的对象
        parent = t;
        // 通过 compare 来比较 key 的大小
        cmp = cpr.compare(key, t.key);
        //key 小于 t，把 t 左边的值赋予 t，因为红黑树左边的值比较小，循环再比
        if (cmp < 0)
            t = t.left;
        //key 大于 t，把 t 右边的值赋予 t，因为红黑树右边的值比较大，循环再比
        else if (cmp > 0)
            t = t.right;
        //如果相等的话，直接覆盖原值
        else
            return t.setValue(value);
        // t 为空，说明已经到叶子节点了
    } while (t != null);
}
```

3. 在父节点的左边或右边插入新增节点，代码如下：

```
//cmp 代表最后一次对比的大小，小于 0 ，代表 e 在上一节点的左边
if (cmp < 0)
    parent.left = e;
//cmp 代表最后一次对比的大小，大于 0 ，代表 e 在上一节点的右边，相等的情况第二步已经
else
    parent.right = e;
```

4. 着色旋转，达到平衡，结束。

从源码中，我们可以看到：

1. 新增节点时，就是利用了红黑树左小右大的特性，从根节点不断往下查找，直到找到节点是 null 为止，节点为 null 说明到达了叶子结点；
2. 查找过程中，发现 key 值已经存在，直接覆盖；

类似的，TreeMap 查找也是类似的原理，有兴趣的同学可以去 github 上面去查看源码。

2.3 小结

TreeMap 相对来说比较简单，红黑树和 HashMap 比较类似，比较关键的是通过 compare 来比较 key 的大小，然后利用红黑树左小右大的特性，为每个 key 找到自己的位置，从而维护了 key 的大小排序顺序。

3 LinkedHashMap 整体架构

HashMap 是无序的，TreeMap 可以按照 key 进行排序，那有木有 Map 是可以维护插入的顺序的呢？接下来我们一起来看下 LinkedHashMap。

LinkedHashMap 本身是继承 HashMap 的，所以它拥有 HashMap 的所有特性，再此基础上，还提供了两大特性：

- 按照插入顺序进行访问；
- 实现了访问最少最先删除功能，其目的是把很久都没有访问的 key 自动删除。

接着我们来看下上述两大特性。

3.1 按照插入顺序访问

3.1.1 LinkedHashMap 链表结构

我们看下 LinkedHashMap 新增了哪些属性，以达到了链表结构的：

```
// 链表头
transient LinkedHashMap.Entry<K,V> head;

// 链表尾
transient LinkedHashMap.Entry<K,V> tail;

// 继承 Node，为数组的每个元素增加了 before 和 after 属性
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}

// 控制两种访问模式的字段，默认 false
// true 按照访问顺序，会把经常访问的 key 放到队尾
// false 按照插入顺序提供访问
final boolean accessOrder;
```

从上述 Map 新增的属性可以看到，LinkedHashMap 的数据结构很像是把 LinkedList 的每个元素换成了 HashMap 的 Node，像是两者的结合体，也正是因为增加了这些结构，从而能把 Map 的元素都串联起来，形成一个链表，而链表就可以保证顺序了，就可以维护元素插入进来的顺序。

3.1.2 如何按照顺序新增

LinkedHashMap 初始化时，默认 accessOrder 为 false，就是会按照插入顺序提供访问，插入方法使用的是父类 HashMap 的 put 方法，不过覆写了 put 方法执行中调用的

newNode/newTreeNode 方法，控制新增节点追加到链表的尾部，这样每次新节点都追加到尾部，即可保证插入顺序了，我们以 newNode 源码为例：

```
// 新增节点，并追加到链表的尾部
Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {
    // 新增节点
    LinkedHashMap.Entry<K,V> p =
        new LinkedHashMap.Entry<K,V>(hash, key, value, e);
    // 追加到链表的尾部
    linkNodeLast(p);
    return p;
}
// link at the end of list
private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {
    LinkedHashMap.Entry<K,V> last = tail;
    // 新增节点等于位节点
    tail = p;
    // last 为空，说明链表为空，首尾节点相等
    if (last == null)
        head = p;
    // 链表有数据，直接建立新增节点和上个尾节点之间的前后关系即可
    else {
        p.before = last;
        last.after = p;
    }
}
```

LinkedHashMap 通过新增头节点、尾节点，给每个节点增加 before、after 属性，每次新增时，都把节点追加到尾节点等手段，在新增的时候，就已经维护了按照插入顺序的链表结构了。

3.1.3 按照顺序访问

LinkedHashMap 只提供了单向访问，即按照插入的顺序从头到尾进行访问，不能像 LinkedList 那样可以双向访问。

我们主要通过迭代器进行访问，迭代器初始化的时候，默认从头节点开始访问，在迭代的过程中，不断访问当前节点的 after 节点即可。

Map 对 key、value 和 entity（节点）都提供出了迭代的方法，假设我们需要迭代 entity，就可使用 LinkedHashMap.entrySet().iterator() 这种写法直接返回 LinkedHashIterator，LinkedHashIterator 是迭代器，我们调用迭代器的 nextNode 方法就可以得到下一个节点，迭代器的源码如下：

```
// 初始化时，默认从头节点开始访问
LinkedHashIterator() {
    // 头节点作为第一个访问的节点
    next = head;
    expectedModCount = modCount;
    current = null;
}

final LinkedHashMap.Entry<K,V> nextNode() {
    LinkedHashMap.Entry<K,V> e = next;
    if (modCount != expectedModCount)// 校验
        throw new ConcurrentModificationException();
    if (e == null)
        throw new NoSuchElementException();
```

```
}
```

在新增节点时，我们就已经维护了元素之间的插入顺序了，所以迭代访问时非常简单，只需要不断的访问当前节点的下一个节点即可。

3.2 访问最少删除策略

3.2.1 demo

这种策略也叫做 LRU（Least recently used,最近最少使用），大概的意思就是经常访问的元素会被追加到队尾，这样不经常访问的数据自然就靠近队头，然后我们可以通过设置删除策略，比如当 Map 元素个数大于多少时，把头节点删除，我们写个 demo 方便大家理解。demo 如下，完整代码可到 github 上查看：

```
public void testAccessOrder() {
  // 新建 LinkedHashMap
  LinkedHashMap<Integer, Integer> map = new LinkedHashMap<Integer, Integer>(4,0.75f,tru
    {
      put(10, 10);
      put(9, 9);
      put(20, 20);
      put(1, 1);
    }

    @Override
    // 覆写了删除策略的方法，我们设定当节点个数大于 3 时，就开始删除头节点
    protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {
      return size() > 3;
    }
  };

  log.info("初始化：{}",JSON.toJSONString(map));
  Assert.assertNotNull(map.get(9));
  log.info("map.get(9)：{}",JSON.toJSONString(map));
  Assert.assertNotNull(map.get(20));
  log.info("map.get(20)：{}",JSON.toJSONString(map));

}
```

打印出来的结果如下：

```
初始化：{9:9,20:20,1:1}
map.get(9)：{20:20,1:1,9:9}
map.get(20)：{1:1,9:9,20:20}
```

可以看到，map 初始化的时候，我们放进去四个元素，但结果只有三个元素，10 不见了，这个主要是因为我们覆写了 removeEldestEntry 方法，我们实现了如果 map 中元素个数大于 3 时，我们就把队头的元素删除，当 put(1, 1) 执行的时候，正好把队头的 10 删除，这个体现了达到我们设定的删除策略时，会自动的删除头节点。

当我们调用 map.get(9) 方法时，元素 9 移动到队尾，调用 map.get(20) 方法时， 元素 20 被移动到队尾，这个体现了经常被访问的节点会被移动到队尾。

这个例子就很好的说明了访问最少删除策略，接下来我们看下原理。

我们先来看下为什么 get 的，元素会被移动到队尾：

```
public V get(Object key) {
    Node<K,V> e;
    // 调用 HashMap  get 方法
    if ((e = getNode(hash(key), key)) == null)
        return null;
    // 如果设置了 LRU 策略
    if (accessOrder)
    // 这个方法把当前 key 移动到队尾
        afterNodeAccess(e);
    return e.value;
}
```

从上述源码中，可以看到，通过 afterNodeAccess 方法把当前访问节点移动到了队尾，其实不仅仅是 get 方法，执行 getOrDefault、compute、computeIfAbsent、computeIfPresent、merge 方法时，也会这么做，通过不断的把经常访问的节点移动到队尾，那么靠近队头的节点，自然就是很少被访问的元素了。

3.2.3 删除策略

上述 demo 我们在执行 put 方法时，发现队头元素被删除了，LinkedHashMap 本身是没有 put 方法实现的，调用的是 HashMap 的 put 方法，但 LinkedHashMap 实现了 put 方法中的调用 afterNodeInsertion 方法，这个方式实现了删除，我们看下源码：

```
// 删除很少被访问的元素，被 HashMap 的 put 方法所调用
void afterNodeInsertion(boolean evict) {
    // 得到元素头节点
    LinkedHashMap.Entry<K,V> first;
    // removeEldestEntry 来控制删除策略，如果队列不为空，并且删除策略允许删除的情况下，删除
    if (evict && (first = head) != null && removeEldestEntry(first)) {
        K key = first.key;
        // removeNode 删除头节点
        removeNode(hash(key), key, null, false, true);
    }
}
```

3.3 小结

LinkedHashMap 提供了两个很有意思的功能：按照插入顺序访问和删除最少访问元素策略，简单地通过链表的结构就实现了，设计得非常巧妙。

总结

本小节主要说了 TreeMap 和 LinkedHashMap 的的数据结构，分析了两者的核心内容源码，我们发现两者充分利用了底层数据结构的特性，TreeMap 利用了红黑树左小右大的特性进行排序，LinkedHashMap 在 HashMap 的基础上简单地加了链表结构，就形成了节点的顺序，非常巧妙，很有意思，大家可以在看源码的过程中，可以多想想设计思路，说不定会有不一样的感悟。

← 08 HashMap 源码解析 10 Map源码会问哪些面试题 →

目录

欢迎在这里发表留言，作者筛选后可公开显示

风舞炫动

3.1.2 LinkedHashMap.Entry last = tail; // 新增节点等于位节点 tail = p; 这块应该是尾结点等于新增节点才对吧

0 回复 2019-11-05

文贺 回复 **风舞炫动**

同学你好，tail 是尾结点，p 是新增节点，tail = p，所以是新增节点等于尾结点，意思是新增节点给尾结点赋值的意思哈。

回复 2019-11-05 20:43:07

licly

问题：LinkedHashMap中afterNodeInsertion方法中 if (evict && (first = head) != null && removeEldestEntry(first)) 这个条件里面removeEldestEntry(first)始终返回的false，所以afterNodeInsertion这个方法调用了也是没有效果的，那该方法的意义是什么呢？

0 回复 2019-10-03

文贺 回复 **licly**

同学你好，你看下3.2.1，是留给子类去实现的。

回复 2019-10-08 19:13:11

慕码人6169125

感觉TreeMap的整体架构那边写的有点问题，“不同的是TreeMap利用了红黑树左节点小，右节点大的性质”，其实HashMap在putValTree方法里面也有用到这个性质。其实这两个结构最大的区别应该是TreeMap真的是一棵完整的树不存在数组，而HashMap还是存在数组结构

0 回复 2019-09-17

文贺 回复 **慕码人6169125**

主要想说明一下 TreeMap 底层是如何利用红黑树实现 key 的排序的，并把这种能力直接通过 API 暴露了出来，HashMap 虽然也有红黑树，但却无法提供这种能力。你说的没错，TreeMap 是没有数组，TreeMap 和 HashMap 相比，差异的太多了，没有数组，也没有链表，各个方法都不同，这样是说不完的，我们主要想基于 TreeMap 已有的能力，来对比下两个 API 利用红黑树之后的结果是啥，给使用者带来什么不同的感受。

回复 2019-09-18 21:22:46

慕码人6169125 回复 **文贺**

哦哦，理解老师要表达的意思啦

回复 2019-09-19 10:19:34

大LOVE辉 回复 **文贺**

我找了半天，想找到数组有没有，就看到这了。。谢谢

回复 2019-11-22 09:37:56

为什么在初始化时就直接删除了 初始化：{} {"9":9,"20":20,"1":1} 9 map.get(9)：{}{"20":20,"1":1,"9":9} 20 map.get(20)：{}{"1":1,"9":9,"20":20} null map.get(10)：{}{"1":1,"9":9,"20":20}

1 回复 2019-09-10

文贺 回复 **Amy楠**

文中有说哈，removeEldestEntry 设置了删除策略

回复 2019-09-10 14:01:13

Megetood

请问一下什么时候能更新完

0 回复 2019-09-05

初一 回复 **Megetood**

同学你好 本专栏是每周二、周四进行更新，感谢支持 ^ ^

回复 2019-09-05 14:36:41

千学不如一看，千看不如一练