



搜索



(/)

文章目录

[1. 概述](#)

[2. 原理](#)

[3. 源码分析](#)

[3.1 Entry 的继承体系](#)

[3.1 链表的建立过程](#)

[3.2 链表节点的删除过程](#)

[3.3 访问顺序的维护过程](#)

[3.4 基于 LinkedHashMap 实现缓存](#)

[4. 总结](#)

[附录：映射类文章列表](#)

LinkedHashMap 源码详细分析（JDK1.8）

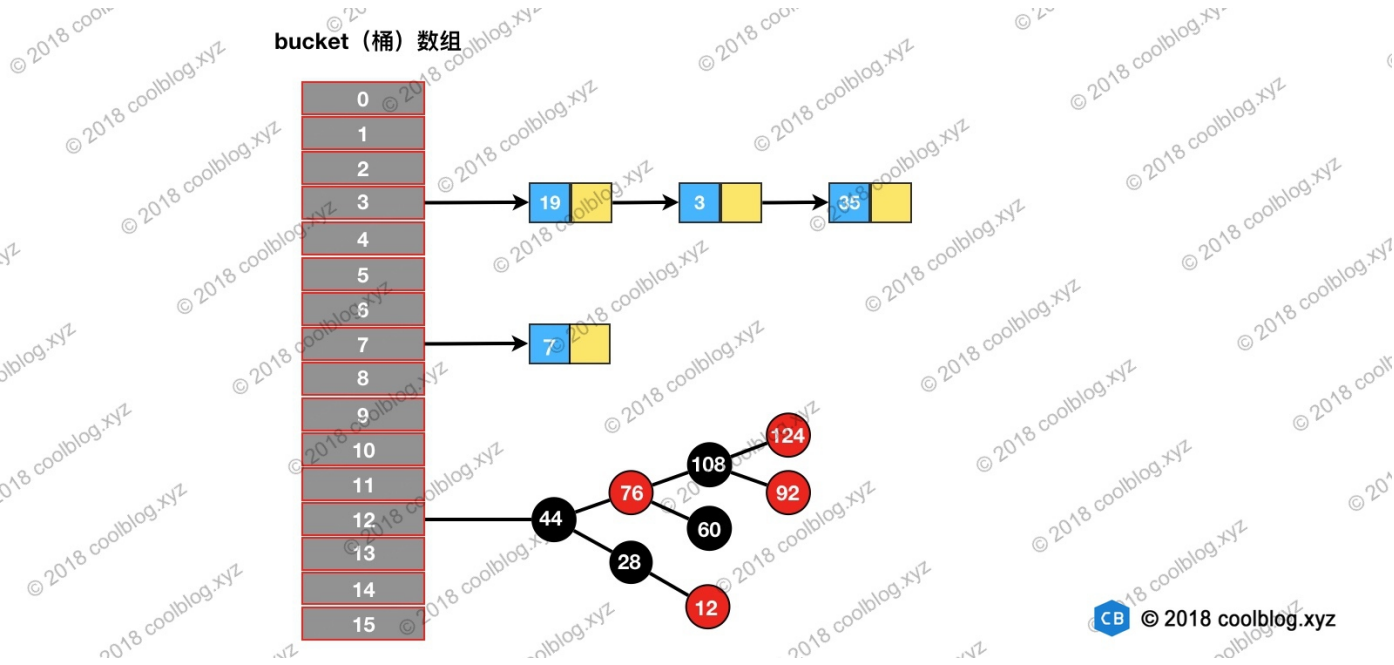
📅 2018-01-24 ([/2018/01/24/LinkedHashMap-源码详细分析（JDK1-8）/](#)) · 📁 [Java基础 \(/categories/foundation-of-java/\)](#) ▶ [集合框架 \(/categories/foundation-of-java/collection/\)](#) · 🏷️ [Java \(/tags/Java/\)](#), [LinkedHashMap \(/tags/LinkedHashMap/\)](#), [集合框架 \(/tags/集合框架/\)](#) · 💬 [评论 \(/2018/01/24/LinkedHashMap-源码详细分析（JDK1-8）/#comments\)](#)

1. 概述

LinkedHashMap 继承自 HashMap，在 HashMap 基础上，通过维护一条双向链表，解决了 HashMap 不能随时保持遍历顺序和插入顺序一致的问题。除此之外，LinkedHashMap 对访问顺序也提供了相关支持。在一些场景下，该特性很有用，比如缓存。在实现上，LinkedHashMap 很多方法直接继承自 HashMap，仅为维护双向链表覆写了部分方法。所以，要看懂 LinkedHashMap 的源码，需要先看懂 HashMap 的源码。关于 HashMap 的源码分析，本文并不打算展开讲了。大家可以参考我之前的一篇文章“HashMap 源码详细分析（JDK1.8）(<http://www.coolblog.xyz/2018/01/18/HashMap-%E6%BA%90%E7%A0%81%E8%AF%A6%E7%BB%86%E5%88%86%E6%9E%90-JDK1-8/>)”。在那篇文章中，我配了十多张图帮助大家学习 HashMap 源码。

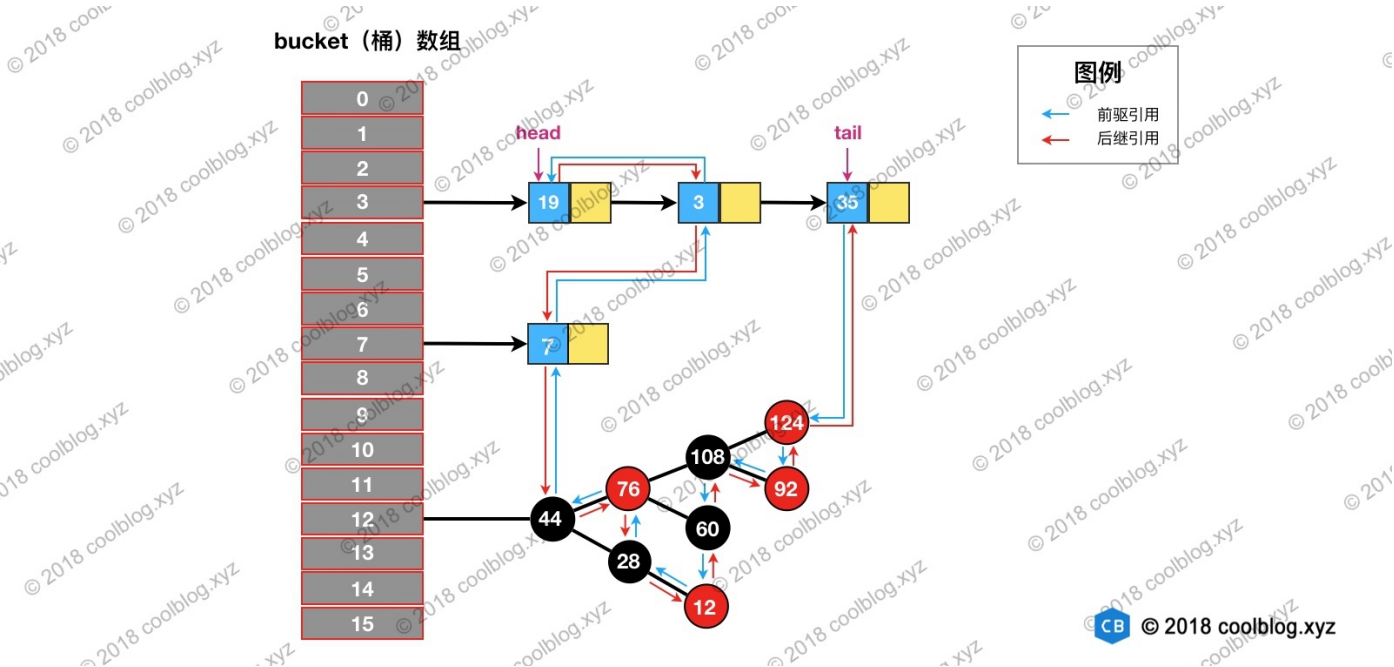
本篇文章的结构与我之前两篇关于 Java 集合类（集合框架 (<http://www.coolblog.xyz/categories/foundation-of-java/collection/>)）的源码分析文章不同，本文将不再分析集合类的基本操作（查找、遍历、插入、删除），而是把重点放在双向链表的维护上。包括链表的建立过程，删除节点的过程，以及访问顺序维护的过程等。好了，接下来开始分析吧。

上(1)章说了 LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构。该结构由数组和链表或红黑树组成，结构示意图大致如下：



(<http://www.coolblog.xyz>)

LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。其结构可能如下图：



(<http://www.coolblog.xyz>)

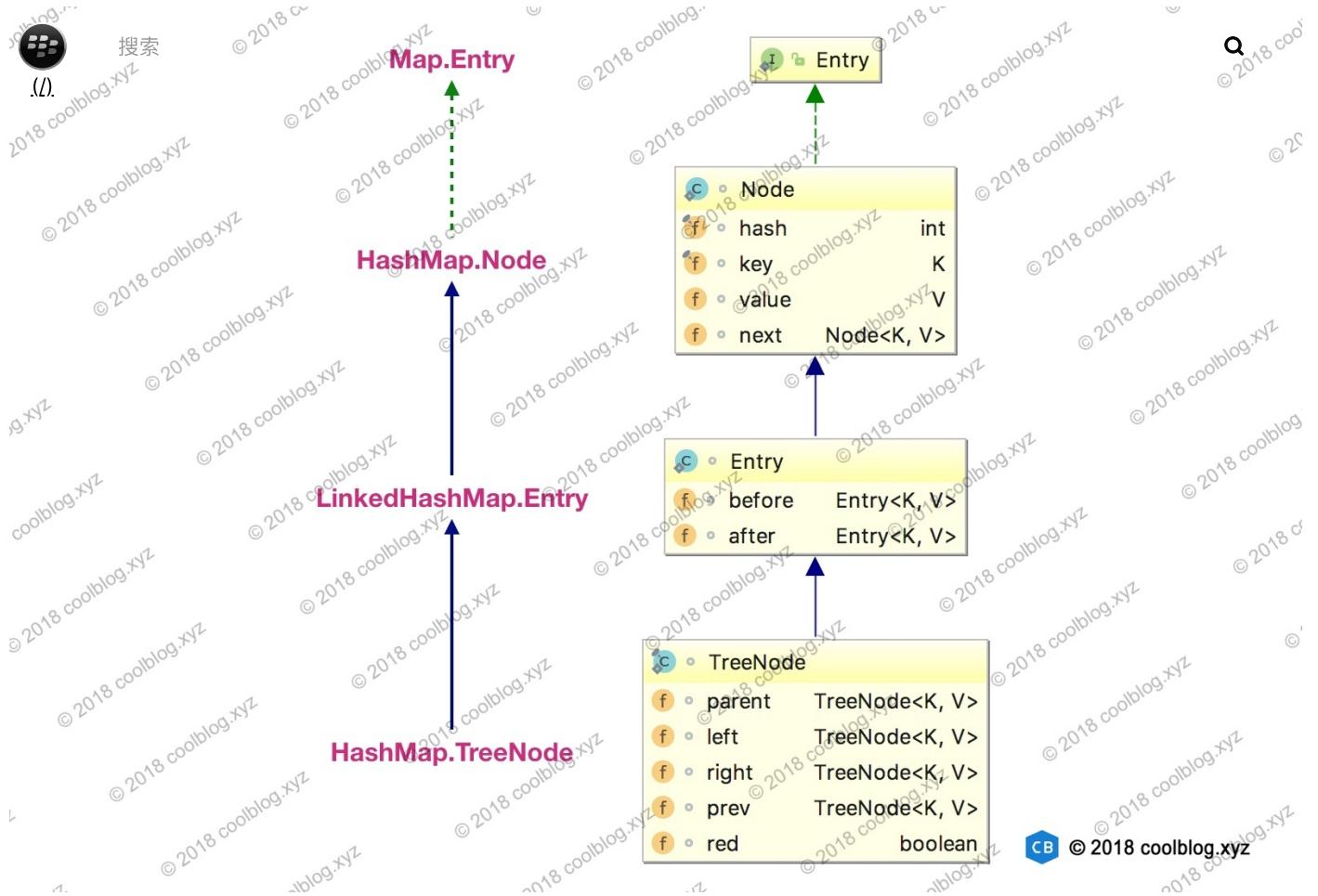
上图中，淡蓝色的箭头表示前驱引用，红色箭头表示后继引用。每当有新键值对节点插入，新节点最终会接在 tail 引用指向的节点后面。而 tail 引用则会移动到新的节点上，这样一个双向链表就建立起来了。

上面的结构并不是很难理解，虽然引入了红黑树，导致结构看起来略为复杂了一些。但大家完全可以忽略红黑树，而只关注链表结构本身。好了，接下来进入细节分析吧。

3. 源码分析

3.1 Entry 的继承体系

在对核心内容展开分析之前，这里先插队分析一下键值对节点的继承体系。先来看看继承体系结构图：



(<http://www.coolblog.xyz>)

上面的继承体系乍一看还是有点复杂的，同时也有点让人迷惑。HashMap 的内部类 `TreeNode` 不继承它的父类 `Node`，却继承自 `Node` 的子类 `LinkedHashMap` 内部类 `Entry`。这里这样做是有一定原因的，这里先不说。先来简单说明一下上面的继承体系。`LinkedHashMap` 内部类 `Entry` 继承自 `HashMap` 内部类 `Node`，并新增了两个引用，分别是 `before` 和 `after`。这两个引用的用途不难理解，也就是用于维护双向链表。同时，`TreeNode` 继承 `LinkedHashMap` 的内部类 `Entry` 后，就具备了和其他 `Entry` 一起组成链表的能力。但是这里需要大家考虑一个问题。当我们使用 `HashMap` 时，`TreeNode` 并不需要具备组成链表能力。如果继承 `LinkedHashMap` 内部类 `Entry`，`TreeNode` 就多了两个用不到的引用，这样做不是会浪费空间吗？简单说明一下这个问题（水平有限，不保证完全正确），这里这么做确实会浪费空间，但与 `TreeNode` 通过继承获取的组成链表的能力相比，这点浪费是值得的。在 `HashMap` 的设计思路注释中，有这样一段话：

Because TreeNodes are about twice the size of regular nodes, we use them only when bins contain enough nodes to warrant use (see `TREEIFY_THRESHOLD`). And when they become too small (due to removal or resizing) they are converted back to plain bins. In usages with well-distributed user hashCodes, tree bins are rarely used.

大致的意思是 `TreeNode` 对象的大小约是普通 `Node` 对象的2倍，我们仅在桶（bin）中包含足够多的节点时再使用。当桶中的节点数量变少时（取决于删除和扩容），`TreeNode` 会被转成 `Node`。当用户实现的 `hashCode` 方法具有良好分布性时，树类型的桶将会很少被使用。

通过上面的注释，我们可以了解到。一般情况下，只要 `hashCode` 的实现不糟糕，`Node` 组成的链表很少会被转成由 `TreeNode` 组成的红黑树。也就是说 `TreeNode` 使用的并不多，浪费那点空间是可接受的。假如 `TreeNode` 机制继承自 `Node` 类，那么它要想具备组成链表的能力，就需要 `Node` 去继承 `LinkedHashMap` 的内部类 `Entry`。这个时候就得不偿失了，浪费很多空间去获取不一定用得上的能力。

说到这里，大家应该能明白节点类型的继承体系了。这里单独拿出来说一下，为下面的分析做铺垫。叙述略为啰嗦，见谅。

3.1 链表的建立过程

链表的建立过程是在插入键值对节点时开始的，初始情况下，让 `LinkedHashMap` 的 `head` 和 `tail` 引用同时指向新节点，链表就算建立起来了。随后不断有新节点插入，通过将新节点接在 `tail` 引用指向节点的后面，即可实现链表的更新。

`Map` 类型的集合类是通过 `put(K,V)` 方法插入键值对，`LinkedHashMap` 本身并没有覆写父类的 `put` 方法，而是直接使用了父类的实现。但在 `HashMap` 中，`put` 方法插入的是 `HashMap` 内部类 `Node` 类型的节点，该类型的节点并不具备与 `LinkedHashMap` 内部类 `Entry` 及其子类型节点组成链表的能力。那么，`LinkedHashMap` 是怎样建立链表的呢？在展开说明之前，我们先看一下 `LinkedHashMap` 插入操作相关的代码：



搜索



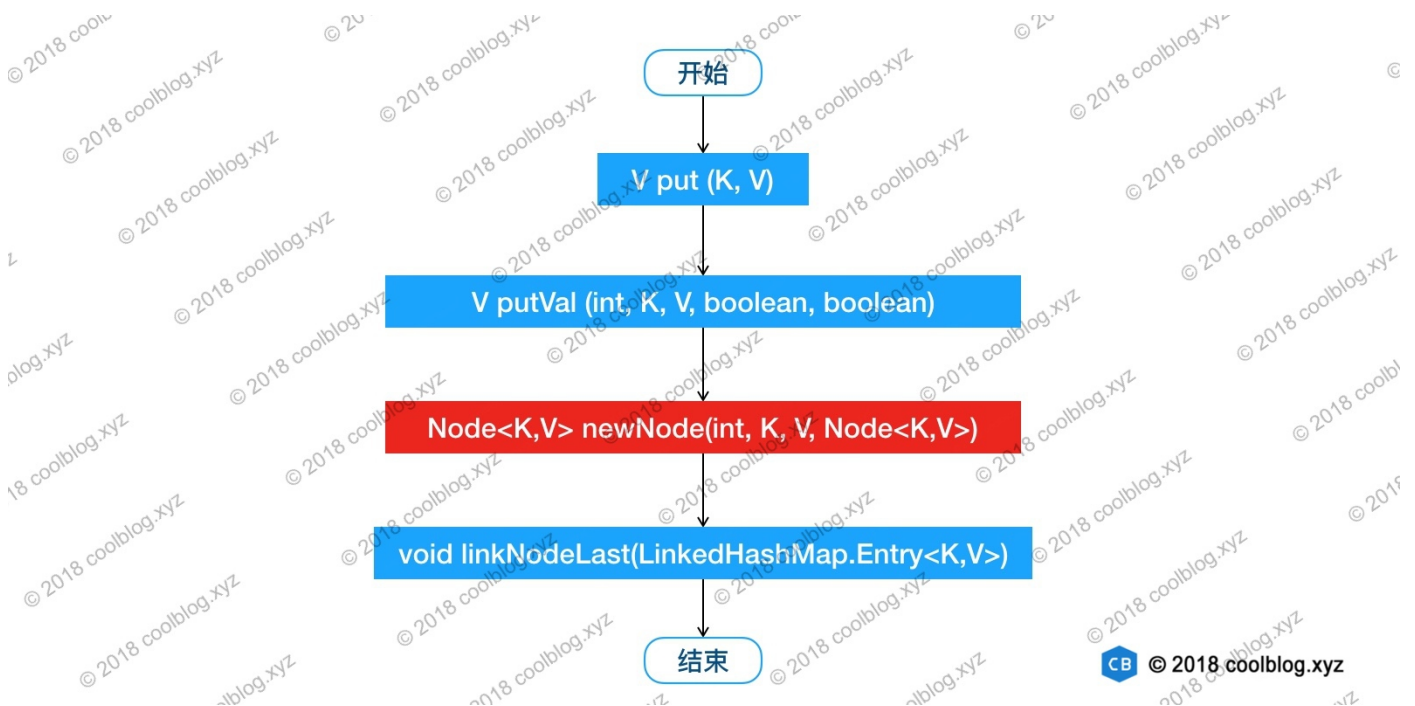
```

// HashMap 中实现
1 public V put(K key, V value) {
2     return putVal(hash(key), key, value, false, true);
3 }
4
5
6 // HashMap 中实现
7 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
8                 boolean evict) {
9     Node<K,V>[] tab; Node<K,V> p; int n, i;
10    if ((tab = table) == null || (n = tab.length) == 0) {...}
11    // 通过节点 hash 定位节点所在的桶位置, 并检测桶中是否包含节点引用
12    if ((p = tab[i = (n - 1) & hash]) == null) {...}
13    else {
14        Node<K,V> e; K k;
15        if (p.hash == hash &&
16            ((k = p.key) == key || (key != null && key.equals(k))))
17            e = p;
18        else if (p instanceof TreeNode) {...}
19        else {
20            // 遍历链表, 并统计链表长度
21            for (int binCount = 0; ; ++binCount) {
22                // 未在单链表中找到要插入的节点, 将新节点接在单链表的后面
23                if ((e = p.next) == null) {
24                    p.next = newNode(hash, key, value, null);
25                    if (binCount >= TREEIFY_THRESHOLD - 1) {...}
26                    break;
27                }
28                // 插入的节点已经存在于单链表中
29                if (e.hash == hash &&
30                    ((k = e.key) == key || (key != null && key.equals(k))))
31                    break;
32                p = e;
33            }
34        }
35        if (e != null) { // existing mapping for key
36            V oldValue = e.value;
37            if (!onlyIfAbsent || oldValue == null) {...}
38            afterNodeAccess(e); // 回调方法, 后续说明
39            return oldValue;
40        }
41    }
42    ++modCount;
43    if (++size > threshold) {...}
44    afterNodeInsertion(evict); // 回调方法, 后续说明
45    return null;
46 }
47
48 // HashMap 中实现
49 Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {
50     return new Node<>(hash, key, value, next);
51 }
52

```

```
52 // LinkedHashMap 中覆写
53 Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {
54     ()
55     LinkedHashMap.Entry<K,V> p =
56         new LinkedHashMap.Entry<K,V>(hash, key, value, e);
57     // 将 Entry 接在双向链表的尾部
58     linkNodeLast(p);
59     return p;
60 }
61
62 // LinkedHashMap 中实现
63 private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {
64     LinkedHashMap.Entry<K,V> last = tail;
65     tail = p;
66     // last 为 null, 表明链表还未建立
67     if (last == null)
68         head = p;
69     else {
70         // 将新节点 p 接在链表尾部
71         p.before = last;
72         last.after = p;
73     }
74 }
```

上面就是 LinkedHashMap 插入相关的源码，这里省略了部分非关键的代码。我根据上面的代码，可以知道 LinkedHashMap 插入操作的调用过程。如下：



(<http://www.coolblog.xyz>)

我把 `newNode` 方法红色背景标注了出来，这一步比较关键。LinkedHashMap 覆写了该方法。在这个方法中，LinkedHashMap 创建了 Entry，并通过 `linkNodeLast` 方法将 Entry 接在双向链表的尾部，实现了双向链表的建立。双向链表建立之后，我们就可以按照插入顺序去遍历 LinkedHashMap，大家可以自己写点测试代码验证一下插入顺序。

以上就是 LinkedHashMap 维护插入顺序的相关分析。本节的最后，再额外补充一些东西。大家如果仔细看上面的代码的话，会发现有两个以 `after` 开头方法，在上文中没有被提及。在 JDK 1.8 HashMap 的源码中，相关的方法有3个：



Callbacks to allow LinkedHashMap post-actions



```

1 void afterNodeAccess(Node<K,V> p) { }
2
3 void afterNodeInsertion(boolean evict) { }
4
5 void afterNodeRemoval(Node<K,V> p) { }

```

根据这三个方法的注释可以看出，这些方法的用途是在增删查等操作后，通过回调的方式，让 LinkedHashMap 有机会做一些后置操作。上述三个方法的具体实现在 LinkedHashMap 中，本节先不分析这些实现，相关分析会在后续章节中进行。

3.2 链表节点的删除过程

与插入操作一样，LinkedHashMap 删除操作相关的代码也是直接用父类的实现。在删除节点时，父类的删除逻辑并不会修复 LinkedHashMap 所维护的双向链表，这不是它的职责。那么删除及节点后，被删除的节点该如何从双链表中移除呢？当然，办法还算是有的。上一节最后提到 HashMap 中三个回调方法运行 LinkedHashMap 对一些操作做出响应。所以，在删除及节点后，回调方法 afterNodeRemoval 会被调用。LinkedHashMap 覆写该方法，并在该方法中完成了移除被删除节点的操作。相关源码如下：

```

1 // HashMap 中实现
2 public V remove(Object key) {
3     Node<K,V> e;
4     return (e = removeNode(hash(key), key, null, false, true)) == null ?
5         null : e.value;
6 }
7
8 // HashMap 中实现
9 final Node<K,V> removeNode(int hash, Object key, Object value,
10                             boolean matchValue, boolean movable) {
11     Node<K,V>[] tab; Node<K,V> p; int n, index;
12     if ((tab = table) != null && (n = tab.length) > 0 &&
13         (p = tab[index = (n - 1) & hash]) != null) {
14         Node<K,V> node = null, e; K k; V v;
15         if (p.hash == hash &&
16             ((k = p.key) == key || (key != null && key.equals(k))))
17             node = p;
18         else if ((e = p.next) != null) {
19             if (p instanceof TreeNode) {...}
20             else {
21                 // 遍历单链表，寻找要删除的节点，并赋值给 node 变量
22                 do {
23                     if (e.hash == hash &&
24                         ((k = e.key) == key ||
25                         (key != null && key.equals(k)))) {
26                         node = e;
27                         break;
28                     }
29                     p = e;
30                 } while ((e = e.next) != null);
31             }
32         }
33         if (node != null && (!matchValue || (v = node.value) == value ||
34             (value != null && value.equals(v)))) {
35             if (node instanceof TreeNode) {...}
36             // 将要删除的节点从单链表中移除
37             else if (node == p)
38                 tab[index] = node.next;

```



搜索

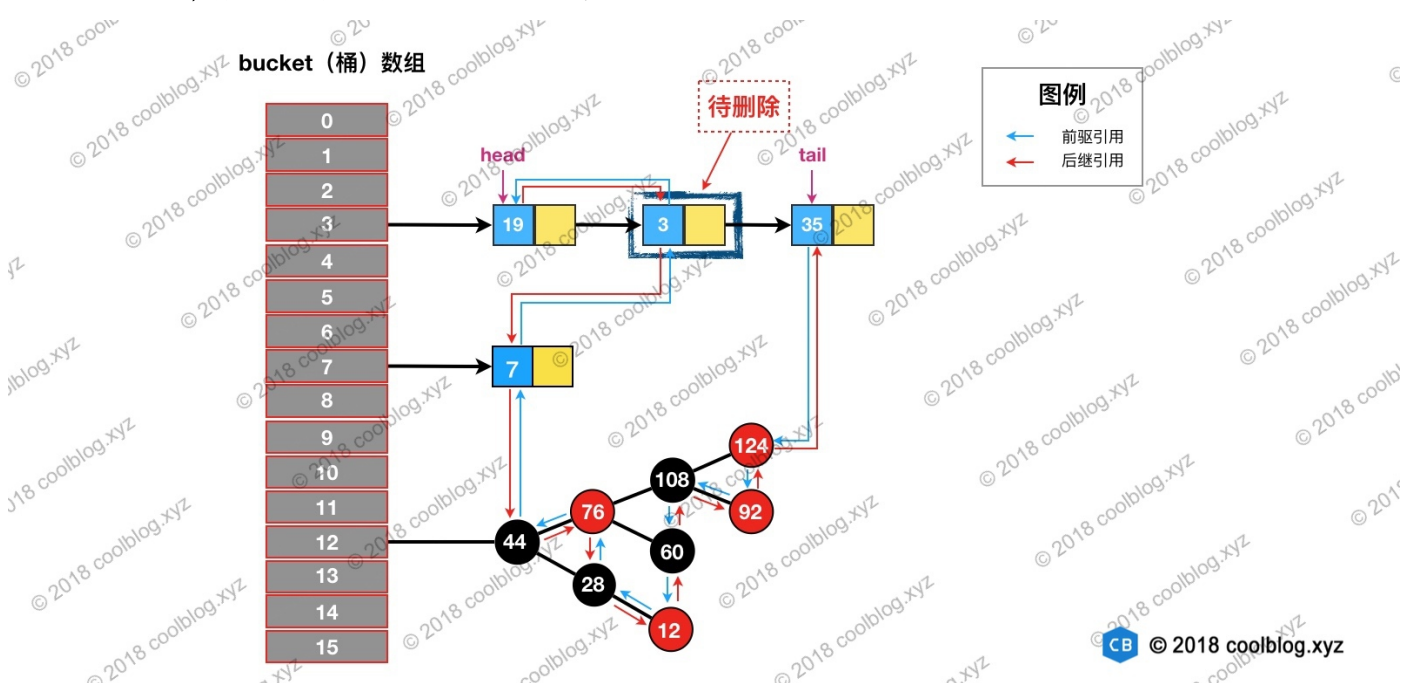


```
26
    else
41      p.next = node.next;
42      ++modCount;
43      --size;
44      afterNodeRemoval(node);    // 调用删除回调方法进行后续操作
45      return node;
46  }
47  return null;
48 }
49
50 // LinkedHashMap 中覆写
51 void afterNodeRemoval(Node<K,V> e) { // unlink
52     LinkedHashMap.Entry<K,V> p =
53         (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
54     // 将 p 节点的前驱后继引用置空
55     p.before = p.after = null;
56     // b 为 null, 表明 p 是头节点
57     if (b == null)
58         head = a;
59     else
60         b.after = a;
61     // a 为 null, 表明 p 是尾节点
62     if (a == null)
63         tail = b;
64     else
65         a.before = b;
66 }
```

删除的过程并不复杂, 上面这么多代码其实就做了三件事:

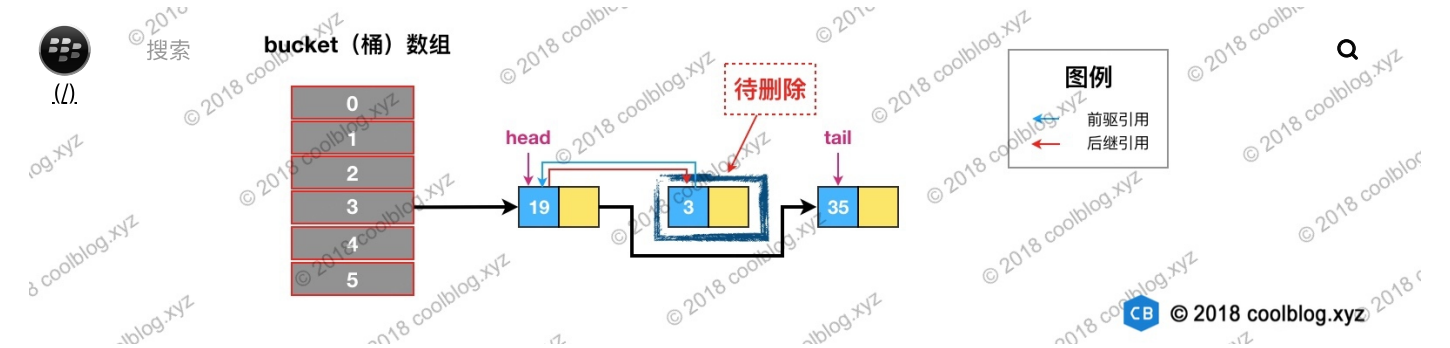
1. 根据 hash 定位到桶位置
2. 遍历链表或调用红黑树相关的删除方法
3. 从 LinkedHashMap 维护的双链表中移除要删除的节点

举个例子说明一下, 假如我们要删除下图键值为 3 的节点。



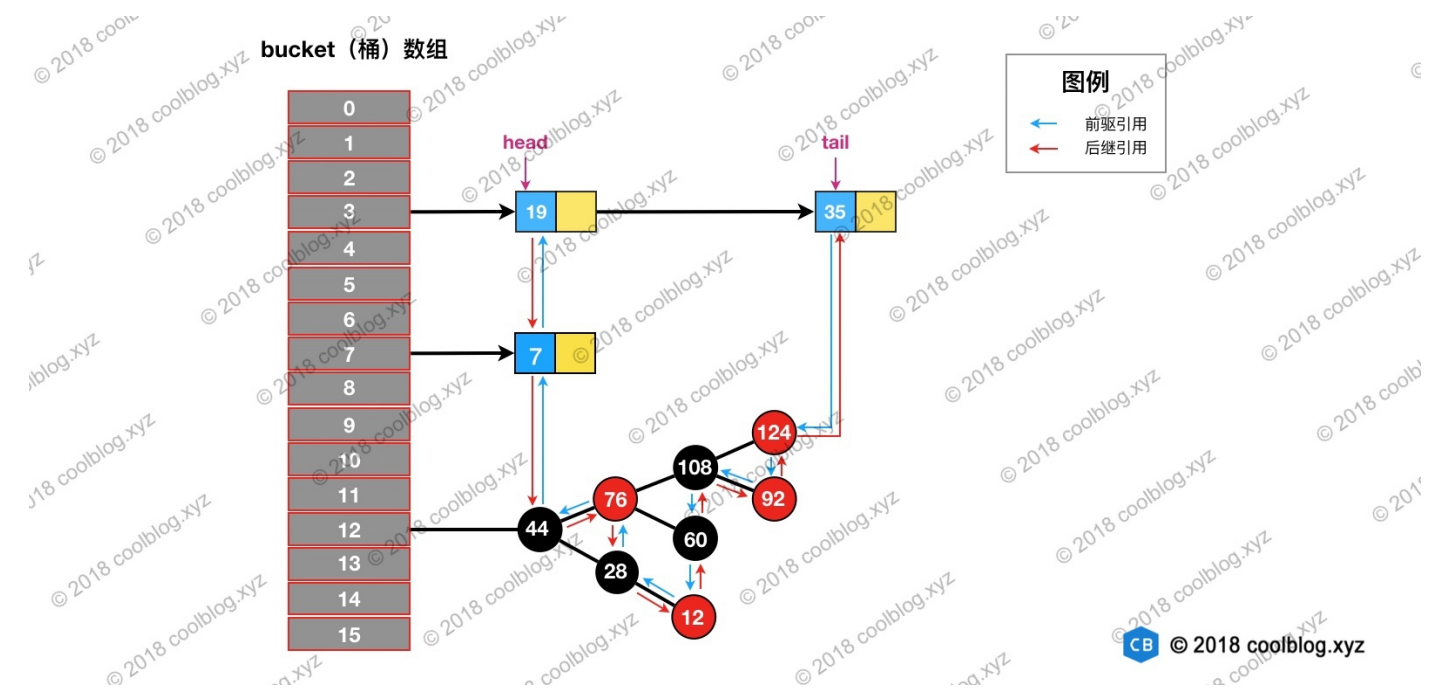
(<http://www.coolblog.xyz>)

根据 hash 定位到该节点属于3号桶, 然后在对3号桶保存的单链表进行遍历。找到要删除的节点后, 先从单链表中移除该节点。如下:



(http://www.coolblog.xyz)

然后再双向链表中移除该节点：



(http://www.coolblog.xyz)

删除及相关修复过程并不复杂，结合上面的图片，大家应该很容易就能理解，这里就不多说了。

3.3 访问顺序的维护过程

前面说了插入顺序的实现，本节来讲讲访问顺序。默认情况下，LinkedListHashMap 是按插入顺序维护链表。不过我们可以在初始化 LinkedListHashMap，指定 accessOrder 参数为 true，即可让它按访问顺序维护链表。访问顺序的原理上并不复杂，当我们调用 get/getOrDefault/replace 等方法时，只需要将这些方法访问的节点移动到链表的尾部即可。相应的源码如下：



// LinkedHashMap 中覆写

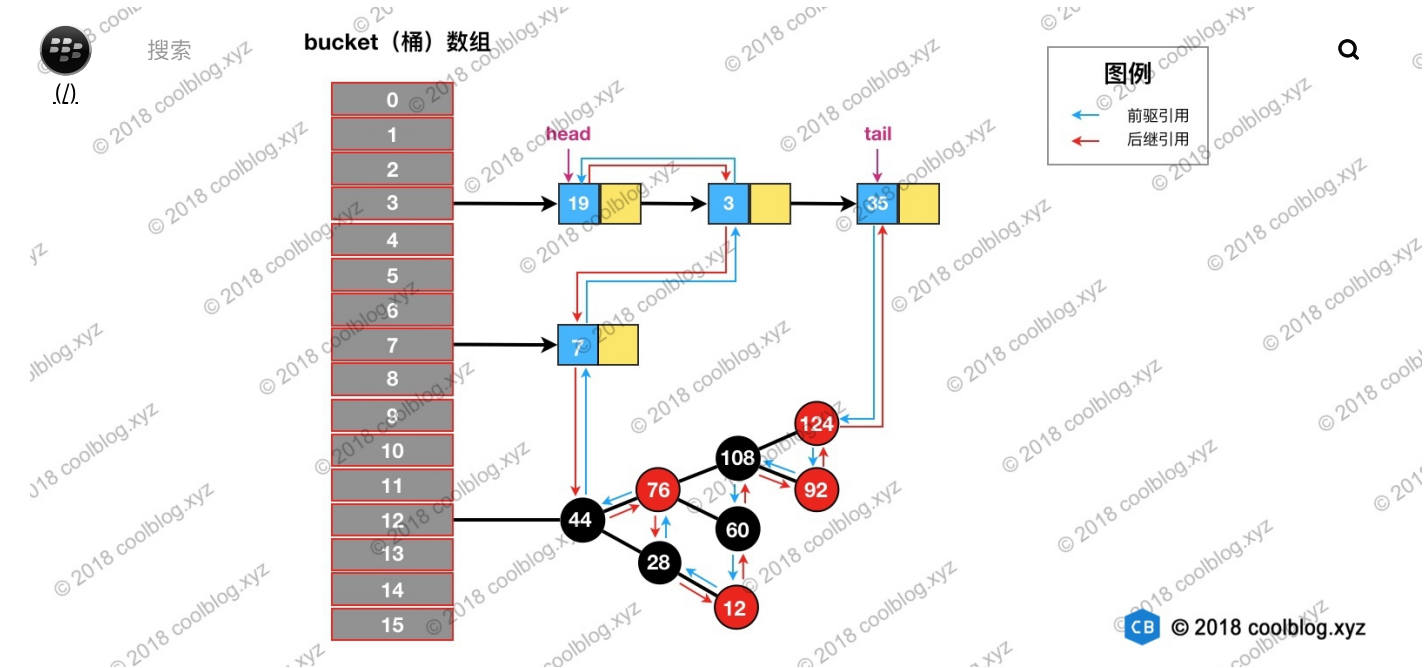
Q

```

(2) public V get(Object key) {
    3     Node<K,V> e;
    4     if ((e = getNode(hash(key), key)) == null)
    5         return null;
    6     // 如果 accessOrder 为 true, 则调用 afterNodeAccess 将被访问节点移动到链表最后
    7     if (accessOrder)
    8         afterNodeAccess(e);
    9     return e.value;
10 }
11
12 // LinkedHashMap 中覆写
13 void afterNodeAccess(Node<K,V> e) { // move node to last
14     LinkedHashMap.Entry<K,V> last;
15     if (accessOrder && (last = tail) != e) {
16         LinkedHashMap.Entry<K,V> p =
17             (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
18         p.after = null;
19         // 如果 b 为 null, 表明 p 为头节点
20         if (b == null)
21             head = a;
22         else
23             b.after = a;
24
25         if (a != null)
26             a.before = b;
27         /*
28          * 这里存疑, 父条件分支已经确保节点 e 不会是尾节点,
29          * 那么 e.after 必然不会为 null, 不知道 else 分支有什么作用
30          */
31         else
32             last = b;
33
34         if (last == null)
35             head = p;
36         else {
37             // 将 p 接在链表的最后
38             p.before = last;
39             last.after = p;
40         }
41         tail = p;
42         ++modCount;
43     }
44 }

```

上面就是访问顺序的实现代码, 并不复杂。下面举例演示一下, 帮助大家理解。假设我们访问下图键值为3的节点, 访问前结构为:





搜索



```
void afterNodeInsertion(boolean evict) { // possibly remove eldest
(2)    LinkedHashMap.Entry<K,V> first;
3        // 根据条件判断是否移除最近最少被访问的节点
4        if (evict && (first = head) != null && removeEldestEntry(first)) {
5            K key = first.key;
6            removeNode(hash(key), key, null, false, true);
7        }
8    }
9
10 // 移除最近最少被访问条件之一，通过覆盖此方法可实现不同策略的缓存
11 protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
12     return false;
13 }
```

上面的源码的核心逻辑在一般情况下都不会被执行，所以之前并没有进行分析。上面的代码做的事情比较简单，就是通过一些条件，判断是否移除最近最少被访问的节点。看到这里，大家应该知道上面两个方法的用途了。当我们基于 LinkedHashMap 实现缓存时，通过覆写 `removeEldestEntry` 方法可以实现自定义策略的 LRU 缓存。比如我们可以根据节点数量判断是否移除最近最少被访问的节点，或者根据节点的存活时间判断是否移除该节点等。本节所实现的缓存是基于判断节点数量是否超限的策略。在构造缓存对象时，传入最大节点数。当插入的节点数超过最大节点数时，移除最近最少被访问的节点。实现代码如下：



搜索

Q

```
public class SimpleCache<K, V> extends LinkedHashMap<K, V> {  
(2)  
3     private static final int MAX_NODE_NUM = 100;  
4  
5     private int limit;  
6  
7     public SimpleCache() {  
8         this(MAX_NODE_NUM);  
9     }  
10  
11    public SimpleCache(int limit) {  
12        super(limit, 0.75f, true);  
13        this.limit = limit;  
14    }  
15  
16    public V save(K key, V val) {  
17        return put(key, val);  
18    }  
19  
20    public V getOne(K key) {  
21        return get(key);  
22    }  
23  
24    public boolean exists(K key) {  
25        return containsKey(key);  
26    }  
27  
28    /**  
29     * 判断节点数是否超限  
30     * @param eldest  
31     * @return 超限返回 true, 否则返回 false  
32     */  
33    @Override  
34    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {  
35        return size() > limit;  
36    }  
37 }
```

测试代码如下:

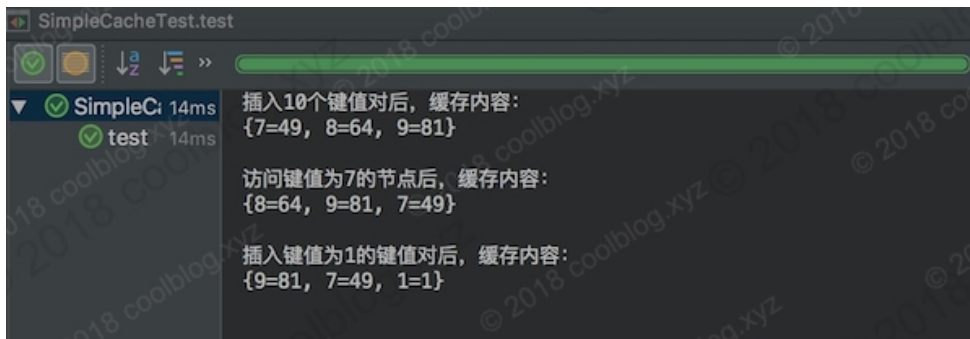


搜索



```
public class SimpleCacheTest {  
(2)  
3     @Test  
4     public void test() throws Exception {  
5         SimpleCache<Integer, Integer> cache = new SimpleCache<>(3);  
6  
7         for (int i = 0; i < 10; i++) {  
8             cache.save(i, i * i);  
9         }  
10  
11        System.out.println("插入10个键值对后, 缓存内容: ");  
12        System.out.println(cache + "\n");  
13  
14        System.out.println("访问键值为7的节点后, 缓存内容: ");  
15        cache.getOne(7);  
16        System.out.println(cache + "\n");  
17  
18        System.out.println("插入键值为1的键值对后, 缓存内容: ");  
19        cache.save(1, 1);  
20        System.out.println(cache);  
21    }  
22 }
```

测试结果如下:



(<http://www.coolblog.xyz>)

在测试代码中, 设定缓存大小为3。在向缓存中插入10个键值对后, 只有最后3个被保存下来了, 其他的都被移除了。然后通过访问键值为7的节点, 使得该节点被移到双向链表的最后位置。当我们再次插入一个键值对时, 键值为7的节点就不会被移除。

本节作为对前面内容的补充, 简单介绍了 LinkedHashMap 在其他方面的应用。本节内容及相关代码并不难理解, 这里就不赘述了。


4. 总结

本文从 LinkedHashMap 维护双向链表的角度对 LinkedHashMap 的源码进行了分析, 并在文章的结尾基于 LinkedHashMap 实现了一个简单的 Cache。在日常开发中, LinkedHashMap 的使用频率虽不及 HashMap, 但它也是个重要的实现。在 Java 集合框架中, HashMap、LinkedHashMap 和 TreeMap 三个映射类基于不同的数据结构, 并实现了不同的功能。HashMap 底层基于拉链式的散列结构, 并在 JDK 1.8 中引入红黑树优化过长链表的问题。基于这样结构, HashMap 可提供高效的增删改查操作。LinkedHashMap 在其之上, 通过维护一条双向链表, 实现了散列数据结构的有序遍历。TreeMap 底层基于红黑树实现, 利用红黑树的性质, 实现了键值对排序功能。我在前面几篇文章中, 对 HashMap 和 TreeMap 以及它们均使用到的红黑树进行了详细的分析, 有兴趣的朋友可以去看看。

到此, 本篇文章就写完了, 感谢大家的阅读!

附录: 映射类文章列表


- 红黑树详细分析 (<http://www.coolblog.xyz/2018/01/11/%E7%BA%A2%E9%BB%91%E6%A0%91%E8%AF%A6%E7%BB%86%E5%88%86%E6%9E%90/>)
- TreeMap源码分析 (<http://www.coolblog.xyz/2018/01/11/TreeMap%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90/>)
- HashMap 源码详细分析(JDK1.8) (<http://www.coolblog.xyz/2018/01/18/HashMap-%E6%BA%90%E7%A0%81%E8%AF%A6%E7%BB%86%E5%88%86%E6%9E%90-JDK1-8/>)



声明：本博客所有文章除特别声明外，均采用 [知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议](http://creativecommons.org/licenses/by-nc-nd/4.0/) (<http://creativecommons.org/licenses/by-nc-nd/4.0/>) 许可协议。转载请注明出处！

搜索

Q



田小波 [_ \(https://github.com/code4wt\)](https://github.com/code4wt)
勿在浮沙筑高台

< [上一篇 \(/\(2018/01/28/ArrayList源码分析/\)\)](#) [下一篇](#) > [/\(2018/01/20/基于-Zookeeper-的分布式锁实现/\)\)](#)