深入 java8 的集合 4：LinkedHashMap 的实现原理

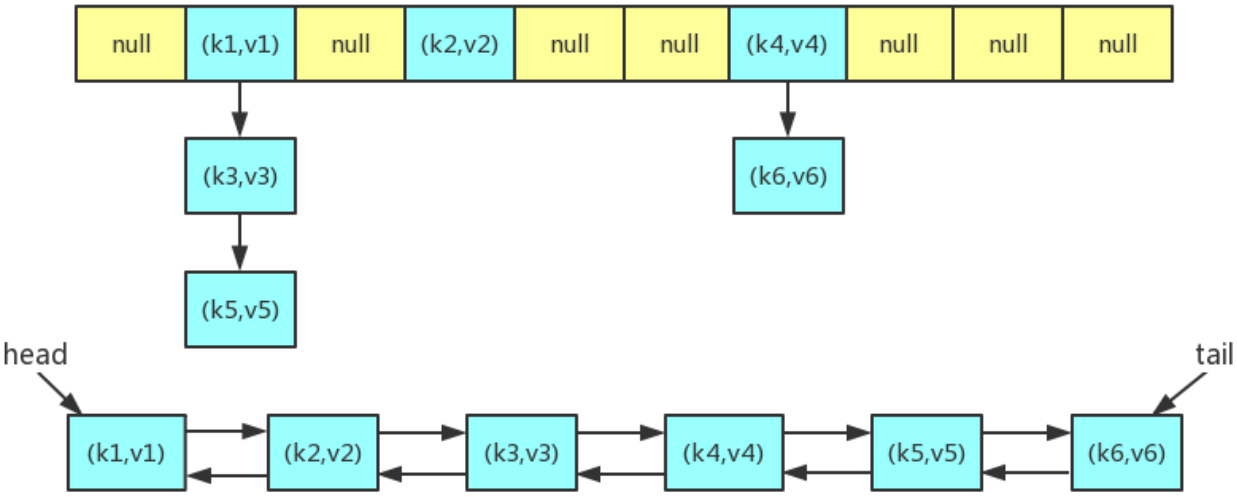
一、概述

按照惯例，先看一下源码里的第一段注释：

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map. (A key k is reinserted into a map m if m.put(k, v) is invoked when m.containsKey(k) would return true immediately prior to the invocation.)
从注释中，我们可以先了解到 LinkedHashMap 是通过哈希表和链表实现的，它通过维护一个链表来保证对哈希表迭代时的有序性，而这个有序是指键值对插入的顺序。另外，当向哈希表中重复插入某个键的时候，不会影响到原来的有序性。也就是说，假设你插入的键的顺序为 1、2、3、4，后来再次插入 2，迭代时的顺序还是 1、2、3、4，而不会因为后来插入的 2 变成 1、3、4、2。（但其实我们可以改变它的规则，使它变成 1、3、4、2）

LinkedHashMap 的实现主要分两部分，一部分是哈希表，另外一部分是链表。哈希表部分继承了 HashMap，拥有了 HashMap 那一套高效的操作，所以我们要看的就是 LinkedHashMap 中链表的部分，了解它是如何来维护有序性的。

LinkedHashMap 的大致实现如下图所示，当然链表和哈希表中相同的键值对都是指向同一个对象，这里把它们分开来画只是为了呈现出比较清晰的结构。



二、属性

在看属性之前，我们先来看一下 LinkedHashMap 的声明：

```
public class LinkedHashMap<K,V> extends HashMap<K,V> imple
ments Map<K,V>
```

从上面的声明中，我们可以看见 LinkedHashMap 是继承自 HashMap 的，所以它已经从 HashMap 那里继承了与哈希表相关的操作了，那么在 LinkedHashMap 中，它可以专注于链表实现的那部分，所以与链表实现相关的属性如下。

```
//LinkedHashMap 的链表节点继承了 HashMap 的节点，而且每个节点都包含了前指针和后指针，所以这里可以看出它是一个双向链表
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);
    }
}

//头指针
transient LinkedHashMap.Entry<K,V> head;

//尾指针
transient LinkedHashMap.Entry<K,V> tail;
```

```
//默认为 false。当为 true 时，表示链表中键值对的顺序与每个键的插入顺序一致，也就是说重复插入键，也会更新顺序

//简单来说，为 false 时，就是上面所指的 1、2、3、4 的情况；为 true 时，就是 1、3、4、2 的情况

final boolean accessOrder;
```

三、方法

如果你有仔细看过 HashMap 源码的话，你会发现 HashMap 中有如下三个方法：

```
// Callbacks to allow LinkedHashMap post-actions

void afterNodeAccess(Node<K,V> p) { }

void afterNodeInsertion(boolean evict) { }

void afterNodeRemoval(Node<K,V> p) { }
```

如果你没有注意到注释的解释的话，你可能会很奇怪为什么会有三个空方法，而且有不少地方还调用过它们。其实这三个方法表示的是在访问、插入、删除某个节点之后，进行一些处理，它们在 LinkedHashMap 都有各自的实现。LinkedHashMap 正是通过重写这三个方法来保证链表的插入、删除的有序性。

1、afterNodeAccess 方法

```
void afterNodeAccess(Node<K,V> e) { // move node to last

    LinkedHashMap.Entry<K,V> last;

    //当 accessOrder 的值为 true，且 e 不是尾节点
```

```
    if (accessOrder && (last = tail) != e) {
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a =
p.after;
        p.after = null;
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a != null)
            a.before = b;
        else
            last = b;
        if (last == null)
            head = p;
        else {
            p.before = last;
            last.after = p;
        }
        tail = p;
        ++modCount;
    }
}
```

这段代码的意思简洁明了，就是把当前节点 e 移至链表的尾部。因为使用的是双向链表，所以在尾部插入可以以 O（1）的时间复杂度来完成。并且只有当 accessOrder

设置为 true 时，才会执行这个操作。在 HashMap 的 putVal 方法中，就调用了这个方法。

2、afterNodeInsertion 方法

```
void afterNodeInsertion(boolean evict) { // possibly remov
e eldest

    LinkedHashMap.Entry<K,V> first;

    if (evict && (first = head) != null && removeEldestEntr
y(first)) {

        K key = first.key;

        removeNode(hash(key), key, null, false, true);

    }

}
```

afterNodeInsertion 方法是在哈希表中插入了一个新节点时调用的，它会把链表的头节点删除掉，删除的方式是通过调用 HashMap 的 removeNode 方法。想一想，通过 afterNodeInsertion 方法和 afterNodeAccess 方法，是不是就可以简单的实现一个基于最近最少使用（LRU）的淘汰策略了？当然，我们还要重写 removeEldestEntry 方法，因为它默认返回的是 false。

3、afterNodeRemoval 方法

```
void afterNodeRemoval(Node<K,V> e) { // unlink

    LinkedHashMap.Entry<K,V> p =

        (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.aft
er;

    p.before = p.after = null;
```

```
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a == null)
            tail = b;
        else
            a.before = b;
    }
```

这个方法是当 HashMap 删除一个键值对时调用的，它会把在 HashMap 中删除的那个键值对一并从链表中删除，保证了哈希表和链表的一致性。

4、get 方法

```
public V get(Object key) {
    Node<K,V> e;
    if ((e = getNode(hash(key), key)) == null)
        return null;
    if (accessOrder)
        afterNodeAccess(e);
    return e.value;
}
```

你没看错，LinkedHashMap 的 get 方法就是这么简单，因为它调用的是 HashMap 的 getNode 方法来获取结果的。并且，如果你把 accessOrder 设置为 true，那么在获取

到值之后，还会调用 afterNodeAccess 方法。这样是不是就能保证一个 LRU 的算法了？

5、put 方法和 remove 方法

我在 LinkedHashMap 的源码中没有找到 put 方法，这就说明了它并没有重写 put 方法，所以我们调用的 put 方法其实是 HashMap 的 put 方法。因为 HashMap 的 put 方法中调用了 afterNodeAccess 方法和 afterNodeInsertion 方法，已经足够保证链表的有序性了，所以它也就没有重写 put 方法了。remove 方法也是如此。