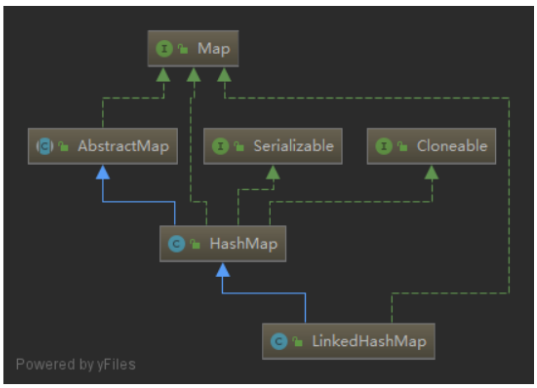
简介

LinkedHashMap内部维护了一个双向链表，能保证元素按插入的顺序访问，也能以访问顺序访问，可以用来实现LRU缓存策略。

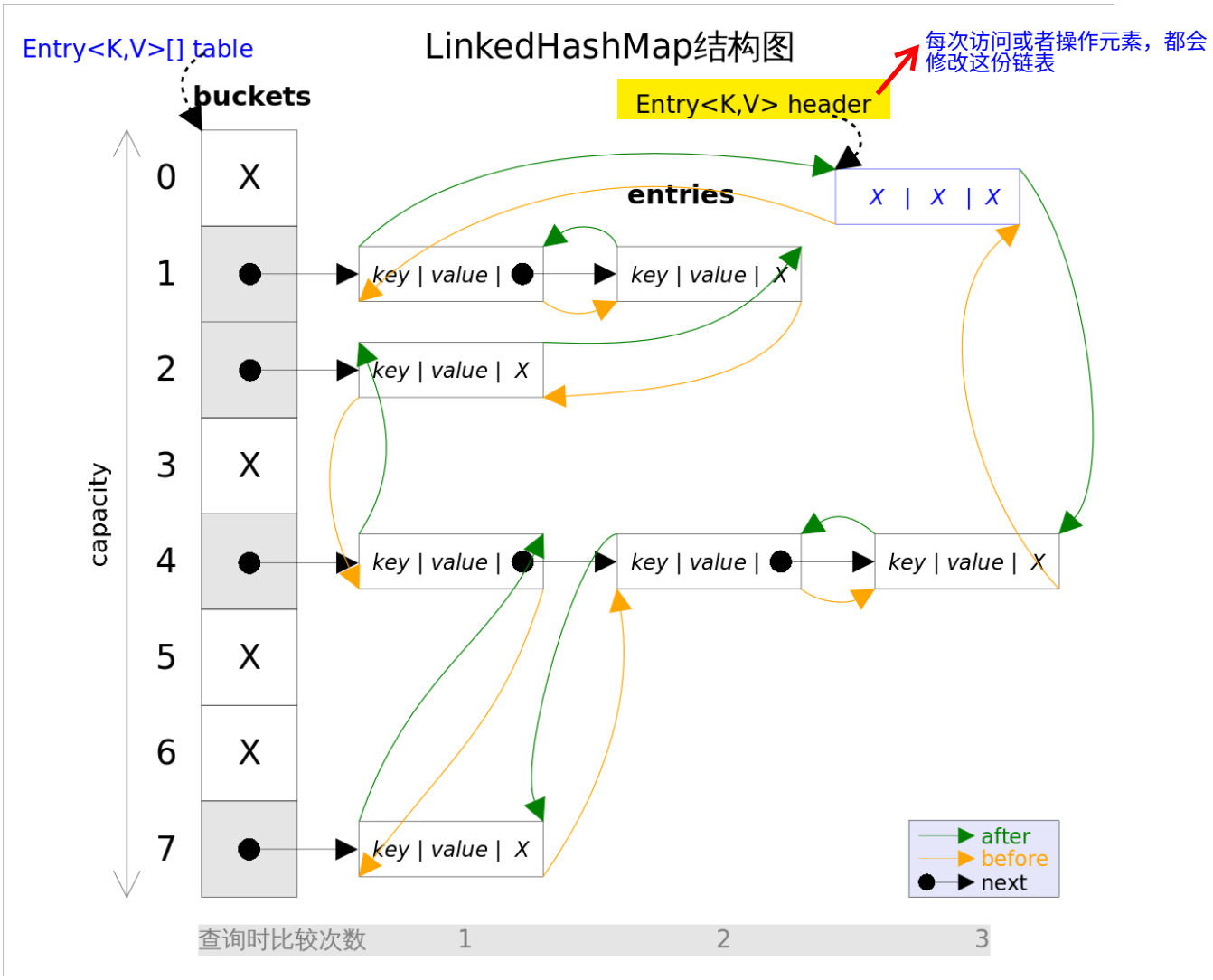
LinkedHashMap可以看成是 LinkedList + HashMap。

继承体系



LinkedHashMap继承HashMap，拥有HashMap的所有特性，并且额外增加的按一定顺序访问的特性。

存储结构



我们知道HashMap使用（数组 + 单链表 + 红黑树）的存储结构，那LinkedHashMap是怎么存储的呢?

通过上面的继承体系，我们知道它继承了Map，所以它的内部也有这三种结构，但是它还额外添加了一种“双向链表”的结构存储所有元素的顺序。

添加删除元素的时候需要同时维护在HashMap中的存储，也要维护在LinkedList中的存储，所以性能上来说会比HashMap稍慢。

源码解析

属性

```
    /**
    * 双向链表头节点
    */
    transient LinkedHashMap.Entry<K,V> head;

    /**
    * 双向链表尾节点
    */
    transient LinkedHashMap.Entry<K,V> tail;

    /**
    * 是否按访问顺序排序
    */
    final boolean accessOrder;

```

（1）head

双向链表的头节点，旧数据存在头节点。

（2）tail

双向链表的尾节点，新数据存在尾节点。

（3）accessOrder 默认false，按插入顺序

是否需要按访问顺序排序，如果为false则按插入顺序存储元素，如果是true则按访问顺序存储元素。

内部类

节点，比hashMap多了before、after两个引用，双向链表就是靠这个来维护的

```
    // 位于LinkedHashMap中
    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);
        }
    }

    // 位于HashMap中
    static class Node<K, V> implements Map.Entry<K, V> {
        final int hash;
        final K key;
        V value;
        Node<K, V> next;
    }

```

存储节点，继承自HashMap的Node类，next用于单链表存储于桶中，before和after用于双向链表存储所有元素。

构造方法

```
    public LinkedHashMap(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor);
        accessOrder = false;
    }

    public LinkedHashMap(int initialCapacity) {
        super(initialCapacity);
        accessOrder = false;
    }

    public LinkedHashMap() {
        super();
        accessOrder = false;
    }

    public LinkedHashMap(Map<? extends K, ? extends V> m) {
        super();
        accessOrder = false;
        putMapEntries(m, false);
    }

    public LinkedHashMap(int initialCapacity,
                         float loadFactor,
                         boolean accessOrder) {
        super(initialCapacity, loadFactor);
        this.accessOrder = accessOrder;
    }
```

前四个构造方法accessOrder都等于false，说明双向链表是按插入顺序存储元素。

最后一个构造方法accessOrder从构造方法参数传入，如果传入true，则就实现了按访问顺序存储元素，这也是实现LRU缓存策略的关键。

afterNodeInsertion(boolean evict)方法

在节点插入之后做些什么，在HashMap中的putVal()方法中被调用，可以看到HashMap中这个方法的实现为空。

true时才removeNode最老节点（双向链表头结点）

```
    void afterNodeInsertion(boolean evict) { // possibly remove eldest
        LinkedHashMap.Entry<K,V> first;
        if (evict && (first = head) != null && removeEldestEntry(first)) {
            K key = first.key;
            removeNode(hash(key), key, null, false, true);
        }
    }

    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return false;
    }
```

```
12.
```

evict，驱逐的意思。

（1）如果evict为true，且头节点不为空，且确定移除最老的元素，那么就调用HashMap.removeNode()把头节点移除（这里的头节点是双向链表的头节点，而不是某个桶中的第一个元素）；

（2）HashMap.removeNode()从HashMap中把这个节点移除之后，会调用afterNodeRemoval()方法；

（3）afterNodeRemoval()方法在LinkedHashMap中也有实现，用来在移除元素后修改双向链表，见下文；

（4）默认removeEldestEntry()方法返回false，也就是不删除元素。

afterNodeAccess(Node<K,V> e)方法

在节点访问之后被调用，主要在put()已经存在的元素或get()时被调用，如果accessOrder为true，调用这个方法把访问到的节点移动到双向链表的末尾。

```
1.     void afterNodeAccess(Node<K,V> e) { // move node to last
2.         LinkedHashMap.Entry<K,V> last;
3.         // 如果accessOrder为true，并且访问的节点不是尾节点
4.         if (accessOrder && (last = tail) != e) {
5.             LinkedHashMap.Entry<K,V> p =
6.                     (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
7.             // 把p节点从双向链表中移除
8.             p.after = null;
9.             if (b == null)
10.                head = a;
11.            else
12.                b.after = a;
13.
14.            if (a != null)
15.                a.before = b;
16.            else
17.                last = b;
18.
19.            // 把p节点放到双向链表的末尾
20.            if (last == null)
21.                head = p;
22.            else {
23.                p.before = last;
24.                last.after = p;
25.            }
26.            // 尾节点等于p
27.            tail = p;
28.            ++modCount;
29.        }
30.    }
31.
```

（1）如果accessOrder为true，并且访问的节点不是尾节点；

（2）从双向链表中移除访问的节点；

（3）把访问的节点加到双向链表的末尾；（末尾为最新访问的元素）

afterNodeRemoval(Node<K,V> e)方法

在节点被删除之后调用的方法。

```
    void afterNodeRemoval(Node<K,V> e) { // unlink
        LinkedHashMap.Entry<K,V> p =
                (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        // 把节点p从双向链表中删除。
        p.before = p.after = null;
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a == null)
            tail = b;
        else
            a.before = b;
    }
```

经典的把节点从双向链表中删除的方法。

get(Object key)方法

获取元素。

```
    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;
    }
```

如果查找到了元素，且accessOrder为true，则调用afterNodeAccess()方法把访问的节点移到双向链表的末尾。

总结

（1）LinkedHashMap继承自HashMap，具有HashMap的所有特性；

（2）LinkedHashMap内部维护了一个双向链表存储所有的元素；

（3）如果accessOrder为false，则可以按插入元素的顺序遍历元素；

（4）如果accessOrder为true，则可以按访问元素的顺序遍历元素；

（5）LinkedHashMap的实现非常精妙，很多方法都是在HashMap中留的钩子（Hook），直接实现这些Hook就可以实现对应的功能了，并不需要再重写put()等方法；

（6）默认的LinkedHashMap并不会移除旧元素，如果需要移除旧元素，则需要重写removeEldestEntry()方法设定移除策略；

afterNodeAccess(Node<K,V> p) ：访问元素之后维护

afterNodeInsertion(boolean evict) ：插入元素之后维护

afterNodeRemoval(Node<K,V> p) ：删除元素之后维护

removeEldestEntry：是否删除双向链表头节点

LinkedHashMap有一个 removeEldestEntry(Map.Entry eldest)方法，通过覆盖这个方法，加入一定的条件，满足条件返回true。当put进新的值方法返回true时，便移除该map中最老的键和值。

（7）LinkedHashMap可以用来实现LRU缓存淘汰策略；

彩蛋

LinkedHashMap如何实现LRU缓存淘汰策略呢?

首先，我们先来看看LRU是个什么鬼。LRU，Least Recently Used，最近最少使用，也就是优先淘汰最近最少使用的元素。

如果使用LinkedHashMap，我们把accessOrder设置为true是不是就差不多能实现这个策略了呢？答案是肯定的。请看下面的代码：

```
1.  package com.coolcoding.code;
2.
3.  import java.util.LinkedHashMap;
4.  import java.util.Map;
5.
6.  /**
7.   * @author: tangtong
8.   * @date: 2019/3/18
9.   */
10. public class LRUTest {
11.     public static void main(String[] args) {
12.         // 创建一个只有5个元素的缓存
13.         LRU<Integer, Integer> lru = new LRU<>(5, 0.75f);
14.         lru.put(1, 1);
15.         lru.put(2, 2);
16.         lru.put(3, 3);
17.         lru.put(4, 4);
18.         lru.put(5, 5);
19.         lru.put(6, 6);
20.         lru.put(7, 7);
21.
22.         System.out.println(lru.get(4));
23.
24.         lru.put(6, 666);
25.
26.         // 输出: {3=3, 5=5, 7=7, 4=4, 6=666}
27.         // 可以看到最旧的元素被删除了
28.         // 且最近访问的4被移到了后面
29.         System.out.println(lru);
30.     }
31. }
32.
33. class LRU<K, V> extends LinkedHashMap<K, V> {
34.
35.     // 保存缓存的容量
36.     private int capacity;
37.
38.     public LRU(int capacity, float loadFactor) {
39.         super(capacity, loadFactor, true);
40.         this.capacity = capacity;
41.     }
42.
43.     /**
44.      * 重写removeEldestEntry()方法设置何时移除旧元素
45.      * @param eldest
46.      * @return
47.      */
48.     @Override
49.     protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
50.         // 当元素个数大于了缓存的容量, 就移除元素
51.         return size() > this.capacity;
52.     }
53. }
54.
55.
```

按访问来，所以就是没人访问的删除
如果按插入来，就是最早的删除

重写这方法，put时符合条件的话就会删除掉双向链表的头节点
所以实现了LRU功能