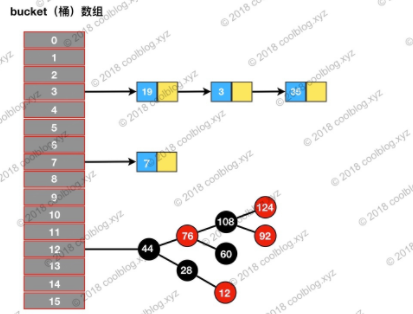
# LinkedHashMap源码分析

**1.1 概述**

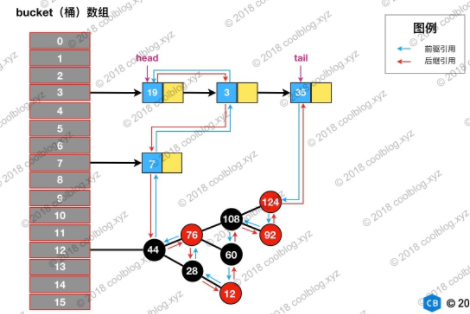
LinkedHashMap 继承自 HashMap，在 HashMap 基础上，通过维护一条双向链表，解决了 HashMap 不能随时保持遍历顺序和插入顺序一致的问题。除此之外，LinkedHashMap 对访问顺序也提供了相关支持。在一些场景下，该特性很有用，比如缓存。在实现上，LinkedHashMap 很多方法直接继承自 HashMap，仅为维护双向链表覆写了部分方法。

**1.2 原理**

**HashMap的底层结构**



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：

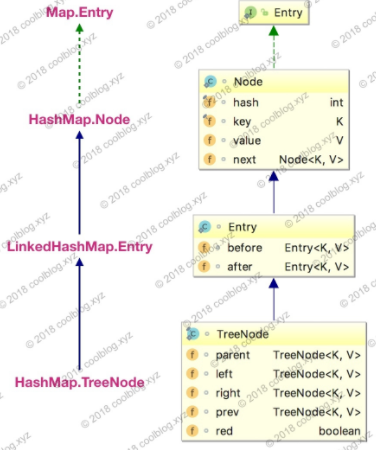


淡蓝色表示前驱，红色箭头表示后继

**1.3 源码分析**

**1.3.1 Entry的继承体系及特性分析**

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



TreeNode不继承它的一个内部类Node，却继承LinkedHashMap的内部类Entry，个人认为是TreeNode在维持红黑树的时候，还会继续维持链表的结构，而且是双向链表，通过prev和next。

①LinkedHashMap和TreeMap都实现了entry的排序，有什么区别：

--TreeMap按照key排序，而LinkedHashMap按照entry插入或者访问顺序排序

--LinkedHashMap保持entry有序方式是调整链表的before，after指针，而treeMap保持entry有序的方式是对tree结构的调整，因此显然LinkedHashMap代价小

②特殊的构造函数LinkedHashMap(int, float,Boolean)

--boolean = true;迭代器顺序遵循LRU原则，最近最少访问的entry会被最先遍历到，这种map结构非常适合构建LRU缓存

③removeEldestEntry（map.entry）

--通过覆写，可以实现：当添加新的映射到map中时，强制自动移除过期的映射。

--过期数据：

----双链表按插入entry排序，则为最早插入双链表的entry

----双链表按访问entry排序，则为最近最少访问的entry

④和hashmap的比较

--增删改查性能比hashmap要差一些，因为要维护双向链表

--迭代器执行时间长短

----LinkedHashMap和size成比例，HashMap和capacity成比例，因此hashmap相对比较费时，以为size<=capacity

⑤三个特殊回调方法

--afterNodeRemoval，删除节点后，双向链表中unlink

--afterNodeInsertion，插入节点后，是否删除eldest节点

--afterNodeAccess，访问节点后，是否调整当前访问节点的顺序

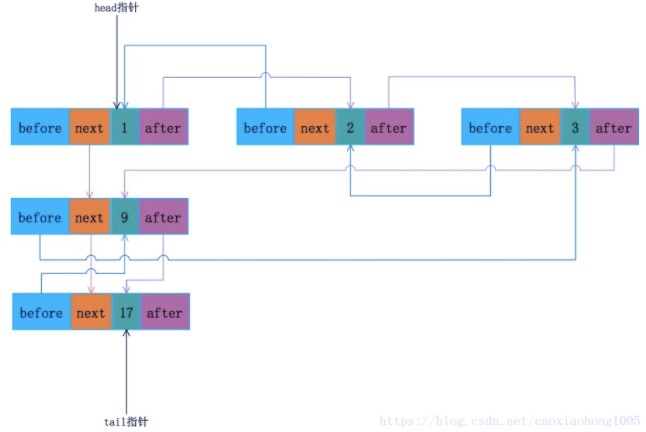
--这三个方法保证了双向链表的有序性，在hashmap中方法体为空，此处进行覆写

⑥为了清晰理解LHM插入节点后的结构，给出一个例子

--hash函数为：h(key)=key%8

--依次插入元素：(k,v)对依次为：(1,11)，(2,12)，(3,13)，(9,19)，(17,27)

--给出结构图：(图中node节点未写出value，只写了key)



**1.3.2 属性**

|  |
| --- |
| **保存头指针和尾指针**  **transient** LinkedHashMap.Entry<K,V> head;  **transient** LinkedHashMap.Entry<K,V> tail;  通过accessOrder来决定双向链表的排序  **final** **boolean** accessOrder;  false：构造函数的默认值，表示按照entry的插入顺序进行排序 ，故每插入一个新的entry则添加到双向链表的尾部。(注意：如果插入entry的key之前就存在双向链表中，则此次插入操作只会更改value，不会更改原双向链表各个entry的顺序)  true：表示按entry的访问顺序进行排序，根据LRU原则，最新访问的entry排列在双链表的尾部 |

**1.3.3 构造函数**

|  |
| --- |
| 指定初始化容量和扩容负载因子，默认按插入顺序  **public** LinkedHashMap(**int** initialCapacity, **float** loadFactor) {  **super**(initialCapacity, loadFactor);  accessOrder = **false**;  }  指定初始化容量，默认按插入顺序  **public** LinkedHashMap(**int** initialCapacity) {  **super**(initialCapacity);  accessOrder = **false**;  }  调用父类的无参构造器  **public** LinkedHashMap() {  **super**();  accessOrder = **false**;  }  调用父类插入集合的方法putmapentries  **public** LinkedHashMap(Map<? **extends** K, ? **extends** V> m) {  **super**();  accessOrder = **false**;  putMapEntries(m, **false**);  }  初始化容量负载因子，和迭代顺序，false按插入，true按访问  **public** LinkedHashMap(**int** initialCapacity,  **float** loadFactor,  **boolean** accessOrder) {  **super**(initialCapacity, loadFactor);  **this**.accessOrder = accessOrder;  } |

**1.3.4增加元素**

LinkedHashMap并没有**重写任何put**方法，但是重写了构建新节点的**newNode方法**。newNode方法会在**hashMap中的putVal**中被调用，**putVal方法**会在批量插入数据**putMapEntries**（Map<? Extends k, ? extends v>, Boolean evict）或者插入单个数据public V **put** （K key, V value）时候被调用

|  |
| --- |
| LinkedHashMap重写了newNode方法，每次**构建新节点**时，通过linkNodeLast(p);将新节点链接在内部双向链表的尾部，创建了一个以null 为节点的entry    **在构建新节点时，构建的是`LinkedHashMap.Entry` 不再是`Node`**  Node<K,V> newNode(**int** hash, K key, V value, Node<K,V> e) {  LinkedHashMap.Entry<K,V> p =  **new** LinkedHashMap.Entry<K,V>(hash, key, value, e);  linkNodeLast(p);  **return** p;  }  **将新增的节点，连接在链表的尾部**  **private** **void** linkNodeLast(LinkedHashMap.Entry<K,V> p) {  LinkedHashMap.Entry<K,V> last = tail;  tail = p;  **if** (last == **null**)  head = p;  **else** {  p.before = last;  last.after = p;  }  }  以及HashMap专门预留给LinkedHashMap的afterNodeAccess() afterNodeInsertion() afterNodeRemoval() 方法。  当accessOrder为true时，将节点移动到最后  **void** afterNodeAccess(Node<K,V> e) { // move node to last  LinkedHashMap.Entry<K,V> last;  **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;  }  }  如果accessOrder为true且当前节点不是tail节点  当前节点设为p，并得到p的before和after节点，分别赋值为b，a  为了将p移到最后，将p的after设为null，如果p的before为null，说明p就是head节点，将head指向p的after；如果before不为null，将b的after指向a；如果a为null，说明p就是last节点，将last指向b，因为if以外会有重新设置tail，此处个人认为写不写无所谓；如果a不为null，将a的before指向b；如果last为null，说明p是新插入的节点，并且链表为空，因此将head指向p，如果last不为null；将p放在last后面，设置新的tail指向p；  -----------------------------------------------------------  **Evict为false表示哈希表处于创建模式，只有在使用map集合作为构造器创建linkedHashMap或者HashMap时才会为false，**  **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**);  }  }  需要下列三个条件才能进入if语句  1.evict为true，只要不是构造方法中插入map集合，evict为true，否则为false  2.first！=null 表明表不为空，基本满足  3.removeEldestEntry（）返回true，该方法定义删除最老节点的规则  --------------------------------------------------------------  **protected** **boolean** removeEldestEntry(Map.Entry<K,V> eldest) {  **return** **false**;  }  而LinkedHashMap的removeEldestEntry方法默认返回false，因此不会进入if语句；  removeEldestEntry该方法用于定义删除最老元素的规则，一旦需要删除最老节点，那么将会调用removeNode删除节点。例如重写removeEldestEntry方法，如果一个个链表只能维持100元素，那么插入第101个元素时，  public boolean removeEldestEntry(Map.Entry<K,V> eldest){  return size()>100;  }  如果有101个元素，removeEldestEntry返回true，进入if语句，删除第一个元素first；第一个元素表示最近最少使用的元素，因为LinkedHashMap的 afterNodeAccess方法将最近访问的放到了表尾  -----------------------------------------------------------------  主要是将节点从双向链表中移除  **void** afterNodeRemoval(Node<K,V> e) { // unlink  LinkedHashMap.Entry<K,V> p =  (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;  p.before = p.after = **null**;  **if** (b == **null**)  head = a;  **else**  b.after = a;  **if** (a == **null**)  tail = b;  **else**  a.before = b;  } |

**1.3.5 查找**

LinkedHashMap重写了get和getOrDefault

|  |
| --- |
| **-----------------------------------------**  **对比hashmap中的实现，linkedHashMap只是增加了accessOrder为true的情况，要去回调**afterNodeAccess，将e节点移到表尾。  **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;  }  **public** V getOrDefault(Object key, V defaultValue) {  Node<K,V> e;  **if** ((e = getNode(*hash*(key), key)) == **null**)  **return** defaultValue;  **if** (accessOrder)  afterNodeAccess(e);  **return** e.value;  } |

**1.3.6 containsValue**

|  |
| --- |
| **public** **boolean** containsValue(Object value) {  Node<K,V>[] tab; V v;  **if** ((tab = table) != **null** && size > 0) {  **for** (**int** i = 0; i < tab.length; ++i) {  **for** (Node<K,V> e = tab[i]; e != **null**; e = e.next) {  **if** ((v = e.value) == value ||  (value != **null** && value.equals(v)))  **return** **true**;  }  }  }  **return** **false**;  }  ---------------------------------------------------------------------  **上述是HashMap的containsValue，通过先找数组，再找链表；而LinkedHashMap直接通过head和after来寻找元素，会比hashMap更高效**  **public** **boolean** containsValue(Object value) {  **for** (LinkedHashMap.Entry<K,V> e = head; e != **null**; e = e.after) {  V v = e.value;  **if** (v == value || (value != **null** && value.equals(v)))  **return** **true**;  }  **return** **false**;  } |

**1.3.7 遍历**

|  |
| --- |
| **返回一个**LinkedEntrySet()  **public** Set<Map.Entry<K,V>> entrySet() {  Set<Map.Entry<K,V>> es;  **return** (es = entrySet) == **null** ? (entrySet = **new** LinkedEntrySet()) : es;  }  final class LinkedEntrySet extends AbstractSet<Map.Entry<K,V>> {  **public** **final** Iterator<Map.Entry<K,V>> iterator() {  **return** **new** LinkedEntryIterator();  }  }  通过LinkedEntryIterator实现entry的遍历，这里就是迭代器里的nextNode方法代表next（）方法  **final** **class** LinkedEntryIterator **extends** LinkedHashIterator  **implements** Iterator<Map.Entry<K,V>> {  **public** **final** Map.Entry<K,V> next() { **return** nextNode(); }  }  **abstract** **class** LinkedHashIterator {  LinkedHashMap.Entry<K,V> next;  LinkedHashMap.Entry<K,V> current;  **int** expectedModCount;  LinkedHashIterator() {  初始化时，next为LinkedHashMap内部维护的双向链表的表头  next = head;  记录当前的modcount，以满足fail-fast  expectedModCount = modCount;  current = **null**;  }  **public** **final** **boolean** hasNext() {  **return** next != **null**;  }  通过双向链表的after进行迭代  **final** LinkedHashMap.Entry<K,V> nextNode() {  LinkedHashMap.Entry<K,V> e = next;  **if** (modCount != expectedModCount)  **throw** **new** ConcurrentModificationException();  **if** (e == **null**)  **throw** **new** NoSuchElementException();  current = e;  next = e.after;  **return** e;  }  底层还是调用hashmap的删除方法  **public** **final** **void** remove() {  Node<K,V> p = current;  **if** (p == **null**)  **throw** **new** IllegalStateException();  **if** (modCount != expectedModCount)  **throw** **new** ConcurrentModificationException();  current = **null**;  K key = p.key;  removeNode(*hash*(key), key, **null**, **false**, **false**);  expectedModCount = modCount;  }  }  **final** **class** LinkedKeyIterator **extends** LinkedHashIterator  **implements** Iterator<K> {  **public** **final** K next() { **return** nextNode().getKey(); }  }  **final** **class** LinkedValueIterator **extends** LinkedHashIterator  **implements** Iterator<V> {  **public** **final** V next() { **return** nextNode().value; }  }  **final** **class** LinkedEntryIterator **extends** LinkedHashIterator  **implements** Iterator<Map.Entry<K,V>> {  **public** **final** Map.Entry<K,V> next() { **return** nextNode(); }  }  } |

1.3.8 LRU通过LinkedHashMap实现

|  |
| --- |
| class LRULinkedHashMap<K, V> extends LinkedHashMap<K, V> {  // 缓存大小  private int capacity;  public LRULinkedHashMap(int capacity) {  // **构造时指定accessOrder为true,按get()时间排序**  super(16, 0.75, true);  this.capacity = capacity;  }  @Override  public boolean removeEldestEntry(Map.Entry<K, V> eldest) {  // **如果添加缓存，即put()后size > capacity，就会移除链表队头，即最近最少使用的缓存项**  return size() > capacity;  }  } |

# LinkedHashSet

|  |
| --- |
| public class LinkedHashSet<E>  extends HashSet<E>  implements Set<E>, Cloneable, java.io.Serializable {  ......  // LinkedHashSet里面有一个LinkedHashMap  public LinkedHashSet(int initialCapacity, float loadFactor) {  map = new LinkedHashMap<>(initialCapacity, loadFactor);  }  ......  public boolean add(E e) {//简单的方法转换  return map.put(e, PRESENT)==null;  }  ......  } |