

## 深入 Java 集合学习系列(四): LinkedHashMap 的实现原理

### 1. LinkedHashMap 概述

LinkedHashMap 是 Map 接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

LinkedHashMap 实现与 HashMap 的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。


注意，此实现不是同步的。如果多个线程同时访问链接的哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

### 2. LinkedHashMap 的实现

对于 LinkedHashMap 而言，它继承于 HashMap、底层使用哈希表与双向链表来保存所有元素。其基本操作与父类 HashMap 相似，它通过重写父类相关的方法，来实现自己的链接列表特性。下面我们来分析 LinkedHashMap 的源代码：

#### 1) Entry 元素：

LinkedHashMap 采用的 hash 算法和 HashMap 相同，但是它重新定义了数组中保存的元素 Entry，该 Entry 除了保存当前对象的引用外，还保存了其上一个元素 before 和下一个元素 after 的引用，从而在哈希表的基础上又构成了双向链接列表。看源代码：

Java 代码 

```
1. /**
2.  * 双向链表的表头元素。
3.  */
4. private transient Entry<K,V> header;
5.
6. /**
7.  * LinkedHashMap 的 Entry 元素。
8.  * 继承 HashMap 的 Entry 元素，又保存了其上一个元素 before 和下一个元素 after 的引用。
9.  */
```

```
10. private static class Entry<K,V> extends HashMap.Entry<K,V> {  
11.     Entry<K,V> before, after;  
12.     .....  
13. }
```

## 2) 初始化:

通过源代码可以看出, 在 `LinkedHashMap` 的构造器中, 实际调用了父类 `HashMap` 的相关构造器来构造一个底层存放的 `table` 数组。如:

Java 代码

```
1. public LinkedHashMap(int initialCapacity, float loadFactor) {  
2.     super(initialCapacity, loadFactor);  
3.     accessOrder = false;  
4. }
```

`HashMap` 中的相关构造方法:

Java 代码

```
1. public HashMap(int initialCapacity, float loadFactor) {  
2.     if (initialCapacity < 0)  
3.         throw new IllegalArgumentException("Illegal initial capacity: " +  
4.             initialCapacity);  
5.     if (initialCapacity > MAXIMUM_CAPACITY)  
6.         initialCapacity = MAXIMUM_CAPACITY;  
7.     if (loadFactor <= 0 || Float.isNaN(loadFactor))  
8.         throw new IllegalArgumentException("Illegal load factor: " +  
9.             loadFactor);  
10.  
11.     // Find a power of 2 >= initialCapacity  
12.     int capacity = 1;  
13.     while (capacity < initialCapacity)  
14.         capacity <= 1;  
15.  
16.     this.loadFactor = loadFactor;  
17.     threshold = (int)(capacity * loadFactor);  
18.     table = new Entry[capacity];  
19.     init();  
20. }
```

我们已经知道 `LinkedHashMap` 的 `Entry` 元素继承 `HashMap` 的 `Entry`, 提供了双向链表的功能。在上述 `HashMap` 的构造器

中，最后会调用 `init()` 方法，进行相关的初始化，这个方法在 `HashMap` 的实现中并无意义，只是提供给子类实现相关的初始化调用。

`LinkedHashMap` 重写了 `init()` 方法，在调用父类的构造方法完成构造后，进一步实现了对其元素 `Entry` 的初始化操作。

Java 代码

```
1. void init() {
2.     header = new Entry<K,V>(-1, null, null, null);
3.     header.before = header.after = header;
4. }
```

### 3) 存储:

`LinkedHashMap` 并未重写父类 `HashMap` 的 `put` 方法，而是重写了父类 `HashMap` 的 `put` 方法调用的子方法 `void addEntry(int hash, K key, V value, int bucketIndex)` 和 `void createEntry(int hash, K key, V value, int bucketIndex)`，提供了自己特有的双向链接列表的实现。

Java 代码

```
1. void addEntry(int hash, K key, V value, int bucketIndex) {
2.     // 调用 create 方法，将新元素以双向链表的的形式加入到映射中。
3.     createEntry(hash, key, value, bucketIndex);
4.
5.     // 删除最近最少使用元素的策略定义
6.     Entry<K,V> eldest = header.after;
7.     if (removeEldestEntry(eldest)) {
8.         removeEntryForKey(eldest.key);
9.     } else {
10.        if (size >= threshold)
11.            resize(2 * table.length);
12.    }
13. }
```

Java 代码

```
1. void createEntry(int hash, K key, V value, int bucketIndex) {
2.     HashMap.Entry<K,V> old = table[bucketIndex];
3.     Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
4.     table[bucketIndex] = e;
5.     // 调用元素的 addBefore 方法，将元素加入到哈希、双向链接列表。
6.     e.addBefore(header);
7.     size++;
}
```

```
8. }
```

Java 代码

```
1. private void addBefore(Entry<K,V> existingEntry) {  
2.     after = existingEntry;  
3.     before = existingEntry.before;  
4.     before.after = this;  
5.     after.before = this;  
6. }
```

## 4) 读取:

LinkedHashMap 重写了父类 HashMap 的 get 方法，实际在调用父类 getEntry() 方法取得查找的元素后，再判断当排序模式 accessOrder 为 true 时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于链表的增加、删除操作是常量级的，故并不会带来性能的损失。

Java 代码

```
1. public V get(Object key) {  
2.     // 调用父类 HashMap 的 getEntry() 方法，取得要查找的元素。  
3.     Entry<K,V> e = (Entry<K,V>)getEntry(key);  
4.     if (e == null)  
5.         return null;  
6.     // 记录访问顺序。  
7.     e.recordAccess(this);  
8.     return e.value;  
9. }
```

Java 代码

```
1. void recordAccess(HashMap<K,V> m) {  
2.     LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;  
3.     // 如果定义了 LinkedHashMap 的迭代顺序为访问顺序，  
4.     // 则删除以前位置上的元素，并将最新访问的元素添加到链表表头。  
5.     if (lm.accessOrder) {  
6.         lm.modCount++;  
7.         remove();  
8.         addBefore(lm.header);  
9.     }  
10. }
```

## 5) 排序模式:

LinkedHashMap 定义了排序模式 `accessOrder`，该属性为 `boolean` 型变量，对于访问顺序，为 `true`；对于插入顺序，则为 `false`。

Java 代码

```
1. private final boolean accessOrder;
```

一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。看 LinkedHashMap 的构造方法，如：

Java 代码

```
1. public LinkedHashMap(int initialCapacity, float loadFactor) {  
2.     super(initialCapacity, loadFactor);  
3.     accessOrder = false;  
4. }
```

这些构造方法都会默认指定排序模式为插入顺序。如果你想构造一个 LinkedHashMap，并打算按从近期访问最少到近期访问最多的顺序（即访问顺序）来保存元素，那么请使用下面的构造方法构造 LinkedHashMap：

Java 代码

```
1. public LinkedHashMap(int initialCapacity,  
2.     float loadFactor,  
3.     boolean accessOrder) {  
4.     super(initialCapacity, loadFactor);  
5.     this.accessOrder = accessOrder;  
6. }
```

该哈希映射的迭代顺序就是最后访问其条目的顺序，这种映射很适合构建 LRU 缓存。LinkedHashMap 提供了 `removeEldestEntry(Map.Entry<K,V> eldest)` 方法，在将新条目插入到映射后，`put` 和 `putAll` 将调用此方法。该方法可以提供在每次添加新条目时移除最旧条目的实现程序，默认返回 `false`，这样，此映射的行为将类似于正常映射，即永远不能移除最旧的元素。

Java 代码

```
1. protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {  
2.     return false;  
3. }
```

此方法通常不以任何方式修改映射，相反允许映射在其返回值的指引下自我修改。如果用此映射构建 LRU 缓存，则非常方便，它允许映射通过删除旧条目来减少内存损耗。

例如：重写此方法，维持此映射只保存 100 个条目的稳定状态，在每次添加新条目时删除最旧的条目。

## Java 代码

```
1. private static final int MAX_ENTRIES = 100;  
2. protected boolean removeEldestEntry(Map.Entry eldest) {  
3.     return size() > MAX_ENTRIES;  
4. }
```