Java集合学习手册（4）：Java LinkedHashMap

一、概述

HashMap是无序的，HashMap在put的时候是根据key的hashcode进行hash然后放入对应的地方。所以在按照一定顺序put进HashMap中，然后遍历出HashMap的顺序跟put的顺序不同（除非在put的时候key已经按照hashcode排序好了，这种几率非常小）

JAVA在JDK1.4以后提供了LinkedHashMap来帮助我们实现了有序的HashMap。

LinkedHashMap是HashMap的一个子类，它保留插入的顺序， 如果需要输出的顺序和输入时的相同，那么就选用LinkedHashMap。

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，LinkedHashMap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。

根据链表中元素的顺序可以分为：按插入顺序的链表和按访问顺序(调用get方法)的链表。默认是按插入顺序排序，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表。

我们写一个简单的LinkedHashMap的程序：

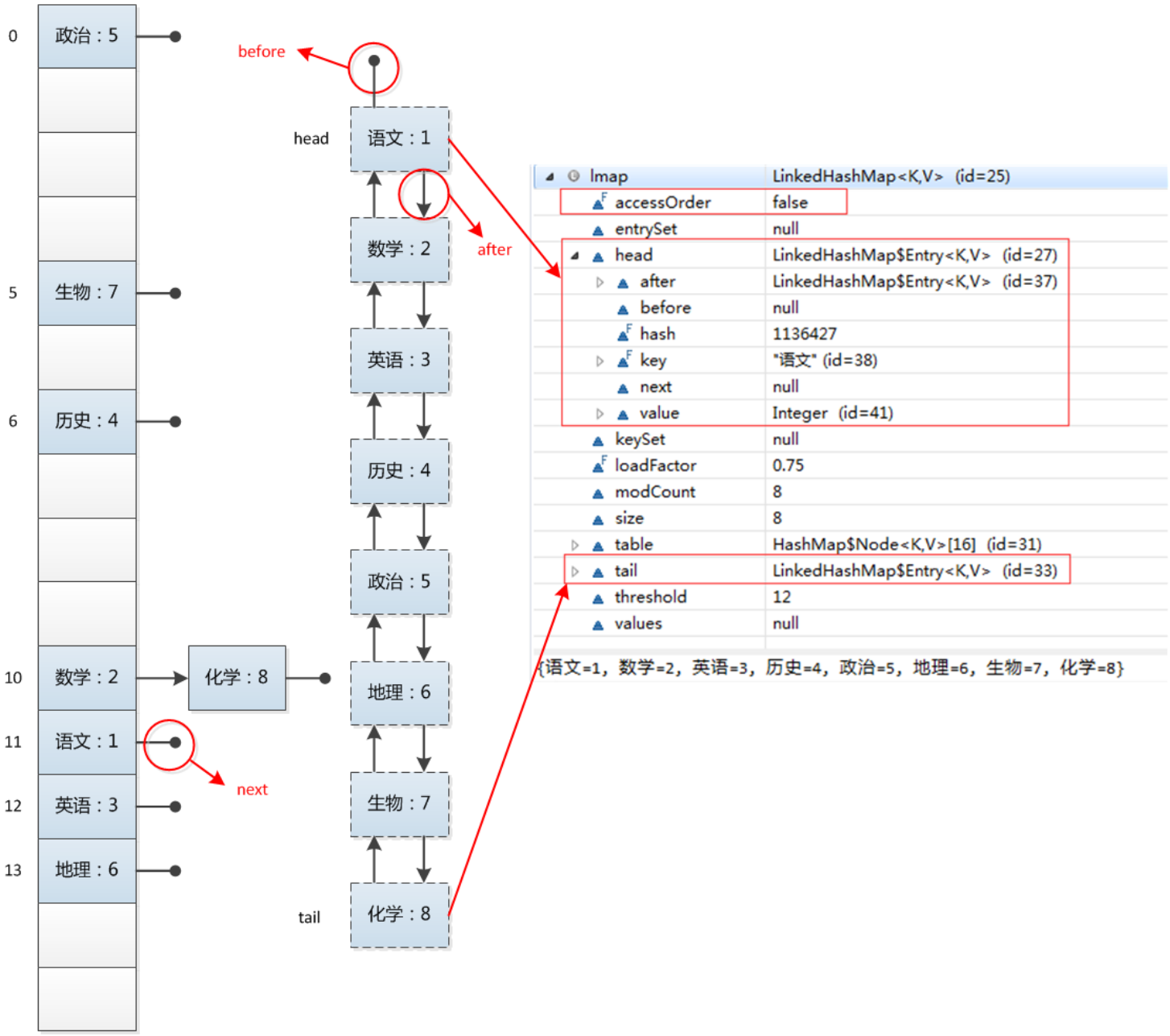
```
LinkedHashMap<String, Integer> lmap = new LinkedHashMap<String, Integer>();
lmap.put("语文", 1);
lmap.put("数学", 2);
lmap.put("英语", 3);
lmap.put("历史", 4);
lmap.put("政治", 5);
lmap.put("地理", 6);
lmap.put("生物", 7);
lmap.put("化学", 8);
```

```
for(Entry<String, Integer> entry : lmap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

运行结果是：

```
语文: 1
数学: 2
英语: 3
历史: 4
政治: 5
地理: 6
生物: 7
化学: 8
```

我们可以观察到，和HashMap的运行结果不同，LinkedHashMap的迭代输出的结果保持了插入顺序。是什么样的结构使得LinkedHashMap具有如此特性呢？我们还是一样的看看LinkedHashMap的内部结构，对它有一个感性的认识：



> 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).

没错，正如官方文档所说：LinkedHashMap是Hash表和链表的实现，并且依靠着双向链表保证了迭代顺序是插入的顺序。

二、插入顺序、访问顺序的演示

先做几个demo来演示一下LinkedHashMap的使用。看懂了其效果，然后再来研究其原理。

2.1 HashMap

看下面这个代码：

```
public static void main(String[] args) {
    Map<String, String> map = new HashMap<String, String>();
    map.put("apple", "苹果");
    map.put("watermelon", "西瓜");
    map.put("banana", "香蕉");
    map.put("peach", "桃子");

    Iterator iter = map.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry entry = (Map.Entry) iter.next();
        System.out.println(entry.getKey() + "=" + entry.getValue());
    }
}
```

一个比较简单的测试HashMap的代码，通过控制台的输出，我们可以看到HashMap是没有顺序的。

```
banana=香蕉
apple=苹果
peach=桃子
watermelon=西瓜
```

2.2 LinkedHashMap

我们现在将map的实现换成LinkedHashMap，其他代码不变：`Map<String, String> map = new LinkedHashMap<String, String>();`
看一下控制台的输出：

```
apple=苹果
watermelon=西瓜
banana=香蕉
peach=桃子
```

我们可以看到，其输出顺序是完成按照插入顺序的！也就是我们上面所说的保留了插入的顺序。我们不是在上面还提到过其可以按照访问顺序进行排序么？好的，我们还是通过一个例子来验证一下：

```
public static void main(String[] args) {
    Map<String, String> map = new LinkedHashMap<String, String>(16,0.75f,true);
```

```
        map.put("apple", "苹果");
        map.put("watermelon", "西瓜");
        map.put("banana", "香蕉");
        map.put("peach", "桃子");

        map.get("banana");
        map.get("apple");

        Iterator iter = map.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            System.out.println(entry.getKey() + "=" + entry.getValue());
        }
    }
```

代码与之前的都差不多，但我们多了两行代码，并且初始化LinkedHashMap的时候，用的构造函数也不相同，看一下控制台的输出结果：

```
watermelon=西瓜
peach=桃子
banana=香蕉
apple=苹果
```

这也就是我们之前提到过的，LinkedHashMap可以选择按照访问顺序进行排序。

三、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：

3.1 成员变量

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：

```
/**
* The iteration ordering method for this linked hash map: <tt>true</tt>
* for access-order, <tt>false</tt> for insertion-order.
* 如果为true，则按照访问顺序；如果为false，则按照插入顺序。
*/
```

```
private final boolean accessOrder;
/**
* 双向链表的表头元素。
 */
private transient Entry<K,V> header;

/**
* LinkedHashMap的Entry元素。
* 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。
 */
private static class Entry<K,V> extends HashMap.Entry<K,V> {
    Entry<K,V> before, after;
    ……
}
```

LinkedHashMap中的Entry集成与HashMap的Entry，但是其增加了before和after的引用，指的是上一个元素和下一个元素的引用。

3.2 初始化

通过源代码可以看出，在LinkedHashMap的构造方法中，实际调用了父类HashMap的相关构造方法来构造一个底层存放的table数组，但额外可以增加accessOrder这个参数，如果不设置，默认为false，代表按照插入顺序进行迭代；当然可以显式设置为true，代表以访问顺序进行迭代。如：

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

HashMap中的相关构造方法：

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);

    // Find a power of 2 >= initialCapacity
```

```
    int capacity = 1;
    while (capacity < initialCapacity)
        capacity <<= 1;

    this.loadFactor = loadFactor;
    threshold = (int)(capacity * loadFactor);
    table = new Entry[capacity];
    init();
}
```

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。

```
/**
* Called by superclass constructors and pseudoconstructors (clone,
* readObject) before any entries are inserted into the map.  Initializes
* the chain.
*/
@Override
void init() {
  header = new Entry<>(-1, null, null, null);
  header.before = header.after = header;
}
```

3.3 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 recordAccess(HashMap m) ， 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。我们在之前的文章中已经讲解了HashMap的put方法，我们在这里重新贴一下HashMap的put方法的源代码：

```
public V put(K key, V value) {
        if (key == null)
            return putForNullKey(value);
        int hash = hash(key);
        int i = indexFor(hash, table.length);
```

```
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

        modCount++;
        addEntry(hash, key, value, i);
        return null;
}
```

重写方法：

```
void recordAccess(HashMap<K,V> m) {
    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    if (lm.accessOrder) {
        lm.modCount++;
        remove();
        addBefore(lm.header);
        }
}

void addEntry(int hash, K key, V value, int bucketIndex) {
    // 调用create方法，将新元素以双向链表的的形式加入到映射中。
    createEntry(hash, key, value, bucketIndex);

    // 删除最近最少使用元素的策略定义
    Entry<K,V> eldest = header.after;
    if (removeEldestEntry(eldest)) {
        removeEntryForKey(eldest.key);
    } else {
        if (size >= threshold)
            resize(2 * table.length);
    }
}

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

```
}

private void addBefore(Entry<K,V> existingEntry) {
    after  = existingEntry;
    before = existingEntry.before;
    before.after = this;
    after.before = this;
}
```

3.4 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。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失。

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

void recordAccess(HashMap<K,V> m) {
    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    // 如果定义了LinkedHashMap的迭代顺序为访问顺序，
    // 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。
    if (lm.accessOrder) {
        lm.modCount++;
        remove();
        addBefore(lm.header);
    }
}

/**
* Removes this entry from the linked list.
*/
private void remove() {
    before.after = after;
    after.before = before;
}
```

```
/**clear链表，设置header为初始状态*/
public void clear() {
 super.clear();
 header.before = header.after = header;
}
```

LinkedHashMap的这些额外操作基本上都是为了维护好那个具有访问顺序的双向链表，目的就是保持双向链表中节点的顺序要从eldest到youngest。

3.4 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。

```
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：

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

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：

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

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 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。

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

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进行自我修改。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。

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

四、总结

其实LinkedHashMap几乎和HashMap一样：从技术上来说，不同的是它定义了一个Entry header，这个header不是放在Table里，它是额外独立出来的。LinkedHashMap通过继承hashMap中的Entry,并添加两个属性Entry before,after,和header结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。

在写关于LinkedHashMap的过程中，记起来之前面试的过程中遇到的一个问题，也是问我Map的哪种实现可以做到按照插入顺序进行迭代？当时脑子是突然短路的，但现在想想，也只能怪自己对这个知识点还是掌握的不够扎实，所以又从头认真的把代码看了一遍。

不过，我的建议是，大家首先首先需要记住的是：LinkedHashMap能够做到按照插入顺序或者访问顺序进行迭代，这样在我们以后的开发中遇到相似的问题，才能想到用LinkedHashMap来解决，否则就算对其内部结构非常了解，不去使用也是没有什么用的。我们学习的目的是为了更好的应用。