

Map 综述（二）：彻头彻尾理解 LinkedHashMap

标签：[链式哈希表的实现原理](#) [链式哈希表源码分析](#) [hashmap LRU 算法与哈希表](#) [哈希表存取与扩容](#)

摘要：

HashMap 和双向链表合二为一即是 LinkedHashMap。所谓 LinkedHashMap，其落脚点在 HashMap，因此更准确地说，它是一个将所有 Entry 节点链入一个双向链表的 HashMap。由于 LinkedHashMap 是 HashMap 的子类，所以 LinkedHashMap 自然会拥有 HashMap 的所有特性。比如，LinkedHashMap 的元素存取过程基本与 HashMap 基本类似，只是在细节实现上稍有不同。当然，这是由 LinkedHashMap 本身的特性所决定的，因为它额外维护了一个双向链表用于保持迭代顺序。此外，LinkedHashMap 可以很好的支持 LRU 算法，笔者在第七节便在 LinkedHashMap 的基础上实现了一个能够很好支持 LRU 的结构。

友情提示：

本文所有关于 LinkedHashMap 的源码都是基于 **JDK 1.6** 的，不同 JDK 版本之间也许会有些许差异，但不影响我们对 LinkedHashMap 的数据结构、原理等整体的把握和了解。

由于 LinkedHashMap 是 HashMap 的子类，所以其具有 HashMap 的所有特性，这一点在源码共用上体现的尤为突出。因此，读者在阅读本文之前，最好对 HashMap 有一个较为深入的了解和回顾，否则很可能会导致事倍功半。特别地，如果读者需要深入了解 HashMap，请移步我的博文[《Map 综述（一）：彻头彻尾理解 HashMap》](#)。

此外，读者在阅读本文之前，对 LinkedList 的进一步了解和回顾也是十分必要的。关于 LinkedList 的更多介绍，请移步我的博文[《Java Collection Framework: List》](#)。

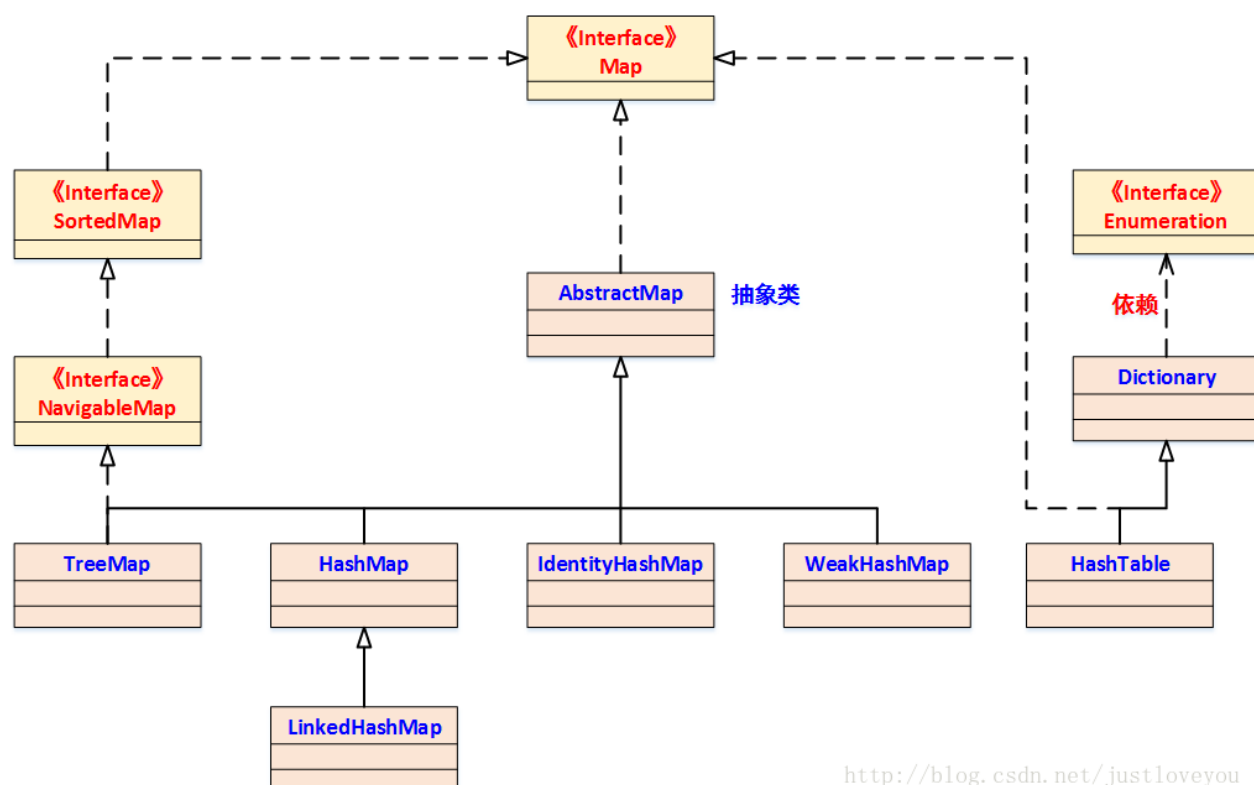
版权声明：

本文原创作者：[书呆子 Rico](#)

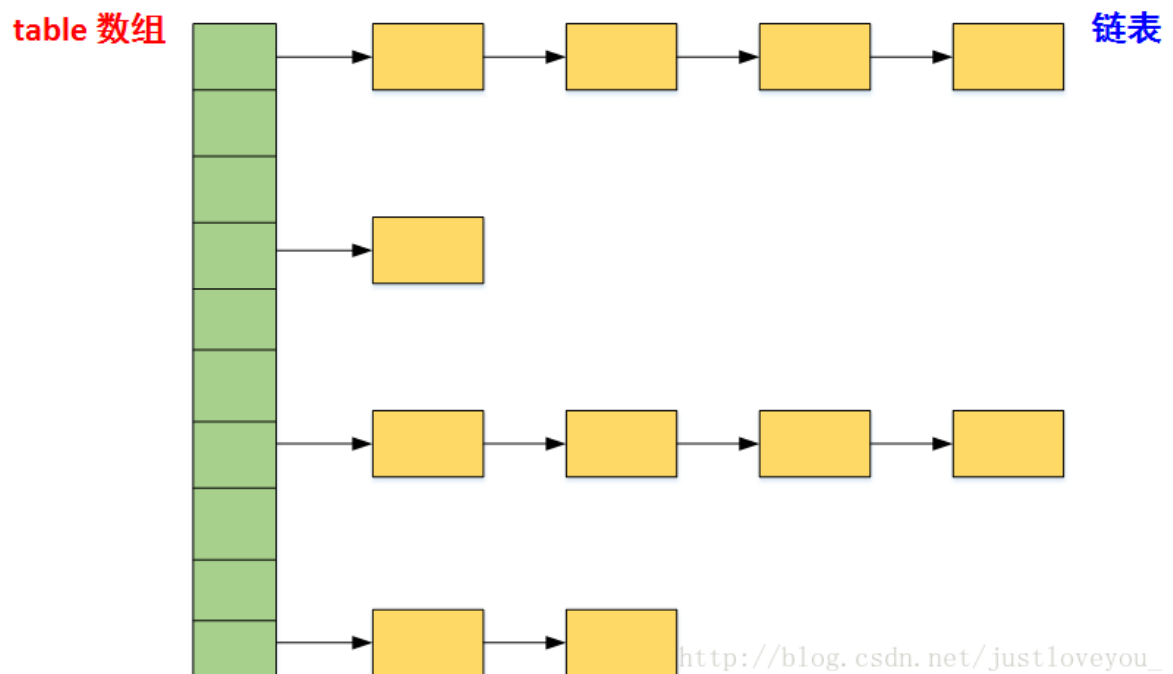
作者博客地址：<http://blog.csdn.net/justloveyou/>

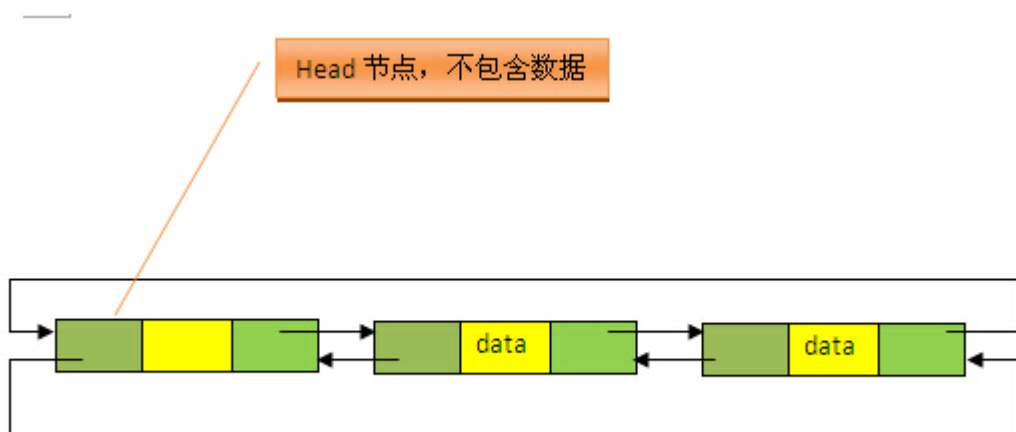
一. LinkedHashMap 概述

笔者曾在[《Map 综述\(一\): 彻头彻尾理解 HashMap》](#)一文中提到, HashMap 是 Java Collection Framework 的重要成员, 也是 Map 族(如下图所示)中我们最为常用的一种。不过遗憾的是, **HashMap 是无序的**, 也就是说, 迭代 **HashMap** 所得到的元素顺序并不是它们最初放置到 **HashMap** 的顺序。**HashMap** 的这一缺点往往会造成诸多不便, 因为在有些场景中, 我们确需要用到一个可以保持插入顺序的 **Map**。庆幸的是, JDK 为我们解决了这个问题, 它为 **HashMap** 提供了一个子类 —— **LinkedHashMap**。虽然 **LinkedHashMap** 增加了时间和空间上的开销, 但是它通过维护一个额外的双向链表保证了迭代顺序。特别地, **该迭代顺序**可以是插入顺序, 也可以是访问顺序。因此, 根据链表中元素的顺序可以将 **LinkedHashMap** 分为: **保持插入顺序的 LinkedHashMap** 和 **保持访问顺序的 LinkedHashMap**, 其中 **LinkedHashMap** 的默认实现是按插入顺序排序的。

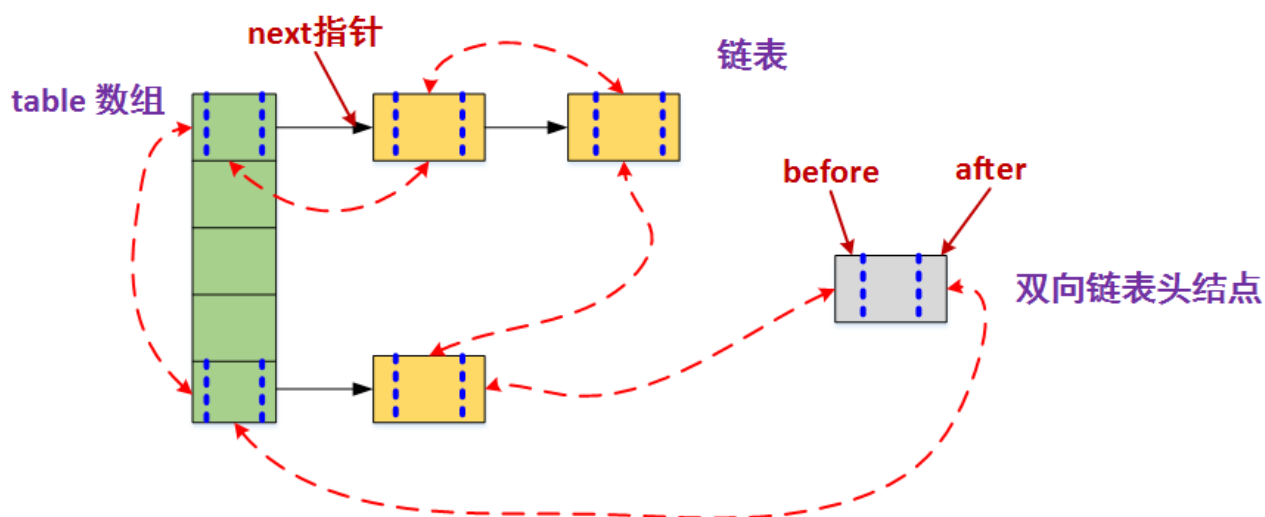


本质上，HashMap 和双向链表合二为一即是 LinkedHashMap。所谓 LinkedHashMap，其落脚点在 HashMap，因此更准确地说，它是一个将所有 Entry 节点链入一个双向链表双向链表的 HashMap。在 LinkedHashMapMap 中，所有 put 进来的 Entry 都保存在如下面第一个图所示的哈希表中，但由于它又额外定义了一个以 head 为头结点的双向链表(如下面第二个图所示)，因此对于每次 put 进来 Entry，除了将其保存到哈希表中对应的位置上之外，还会将其插入到双向链表的尾部。





更直观地，下图很好地还原了 LinkedHashMap 的原貌：HashMap 和双向链表的密切配合和分工合作造就了 LinkedHashMap。特别需要注意的是，next 用于维护 HashMap 各个桶中的 Entry 链，before、after 用于维护 LinkedHashMap 的双向链表，虽然它们的作用对象都是 Entry，但是各自分离，是两码事儿。

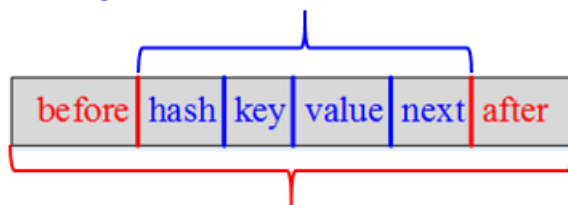


本示意图中，LinkedHashMap一共包含五个节点。除去红色双向虚线来看，其就是一个正宗的HashMap。在这里，用额外的红色虚线串起来的节点就是一个双向链表了。由此可见，LinkedHashMap就是一个标准的HashMap与LinkedList的融合体。

<http://blog.csdn.net/justloveyou>

其中，HashMap 与 LinkedHashMap 的 Entry 结构示意图如下图所示：

HashMap中的Entry的结构，next用于维护每个桶中的单链表



LinkedHashMap中的Entry的结构，before/after用于维护整个双向链表

特别地，由于 LinkedHashMap 是 HashMap 的子类，所以 LinkedHashMap 自然会拥有 HashMap 的所有特性。比如，LinkedHashMap 也最多只允许一条 Entry 的键为 Null(多条会覆盖)，但允许多条 Entry 的值为 Null。此外，LinkedHashMap 也是 Map 的一个非同步的实现。此外，LinkedHashMap 还可以用来实现 LRU (Least recently used, 最近最少使用)算法，这个问题会在下文的特别谈到。

二. LinkedHashMap 在 JDK 中的定义

1、类结构定义

LinkedHashMap 继承于 HashMap，其在 JDK 中的定义为：

```
public class LinkedHashMap<K,V>

    extends HashMap<K,V>

    implements Map<K,V> {
```

```
...  
}
```

2、成员变量定义

与 `HashMap` 相比, `LinkedHashMap` 增加了两个属性用于保证迭代顺序, 分别是 **双向链表头结点 `header`** 和 **标志位 `accessOrder`** (值为 `true` 时, 表示按照访问顺序迭代; 值为 `false` 时, 表示按照插入顺序迭代)。

```
/**  
  
 * The head of the doubly linked list.  
  
 */  
  
private transient Entry<K,V> header; // 双向链表的表头元素  
  
/**  
  
 * The iteration ordering method for this linked hash map: <tt>true</tt>  
 * for access-order, <tt>>false</tt> for insertion-order.  
  
 *  
 * @serial  
  
 */  
  
private final boolean accessOrder; //true 表示按照访问顺序迭代, false  
时表示按照插入顺序
```

3、成员方法定义

从下图我们可以看出，LinkedHashMap 中并没有增加额外方法。也就是说，LinkedHashMap 与 HashMap 在操作上大致相同，只是在实现细节上略有不同罢了。

java.util
LinkedHashMap<K, V>
serialVersionUID : long
header : Entry<K, V>
accessOrder : boolean
LinkedHashMap(int, float)
LinkedHashMap(int)
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V>)
LinkedHashMap(int, float, boolean)
init() : void
transfer(Entry[]) : void
containsValue(Object) : boolean
get(Object) : V
clear() : void
Entry<K, V>
LinkedHashIterator<T>
KeyIterator
ValueIterator
EntryIterator
newKeyIterator() : Iterator<K>
newValueIterator() : Iterator<V>
newEntryIterator() : Iterator<Entry<K, V>>
addEntry(int, K, V, int) : void
createEntry(int, K, V, int) : void
removeEldestEntry(Entry<K, V>) : boolean

LinkedHashMap特有的两个属性：双向列表头结点和迭代顺序

基于对应父类HashMap的五个构造方法

重写父类中的方法

4、基本元素 Entry

LinkedHashMap 采用的 hash 算法和 HashMap 相同，但是它重新定义了 Entry。LinkedHashMap 中的 Entry 增加了两个指针 **before** 和 **after**，它们分别用于维护双向链接列表。特别需要注意的是，**next** 用于维护 HashMap 各个桶中 Entry 的连接顺序，**before**、**after** 用于维护 Entry 插入的先后顺序的，源代码如下：

```
private static class Entry<K,V> extends HashMap.Entry<K,V> {  
  
    // These fields comprise the doubly linked list used for iteration.  
  
    Entry<K,V> before, after;  
}
```

```

Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {

    super(hash, key, value, next);

}

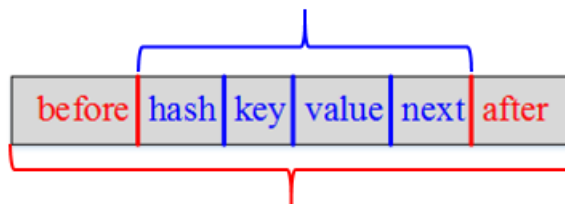
...

}

```

形象地，HashMap 与 LinkedHashMap 的 Entry 结构示意图如下图所示：

HashMap中的Entry的结构，next用于维护每个桶中的单链表



LinkedHashMap中的Entry的结构，before/after用于维护整个双向链表

三. LinkedHashMap 的构造函数

LinkedHashMap 一共提供了五个构造函数，它们都是在 HashMap 的构造函数的基础上实现的，分别如下：

1、LinkedHashMap()

该构造函数意在构造一个具有 默认初始容量 (16)和默认负载因子(0.75)的空 LinkedHashMap，是 Java Collection Framework 规范推荐提供的，其源码如下：


```
/**
 * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance
 * with the default initial capacity (16) and load factor (0.75).
 */
public LinkedHashMap() {
    super(); // 调用 HashMap 对应的构造函数
    accessOrder = false; // 迭代顺序的默认值
}
```

2、LinkedHashMap(int initialCapacity, float loadFactor)

该构造函数意在构造一个指定初始容量和指定负载因子的空 LinkedHashMap，其源码如下：

```
/**
 * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance
 * with the specified initial capacity and load factor.
 *
 * @param initialCapacity the initial capacity
 * @param loadFactor      the load factor
 *
 * @throws IllegalArgumentException if the initial capacity is negative
```

```

        *           or the load factor is nonpositive

        */

    public LinkedHashMap(int initialCapacity, float loadFactor) {

        super(initialCapacity, loadFactor);    // 调用 HashMap 对应的构造
函数

        accessOrder = false;                // 迭代顺序的默认值

    }

```

3、LinkedHashMap(int initialCapacity)

该构造函数意在构造一个指定初始容量和默认负载因子 (0.75) 的空 LinkedHashMap，其源码如下：

```

/**

    * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> inst
    ance

    * with the specified initial capacity and a default load factor (0.7
    5).

    *

    * @param  initialCapacity the initial capacity

    * @throws IllegalArgumentException if the initial capacity is negati
    ve

    */

    public LinkedHashMap(int initialCapacity) {

        super(initialCapacity); // 调用 HashMap 对应的构造函数

```

```
        accessOrder = false;    // 迭代顺序的默认值

    }
```

4、LinkedHashMap(Map<? extends K, ? extends V> m)

该构造函数意在构造一个与指定 Map 具有相同映射的 LinkedHashMap，其初始容量不小于 16 (具体依赖于指定 Map 的大小)，负载因子是 0.75，是 Java Collection Framework 规范推荐提供的，其源码如下：

```
/**
 * Constructs an insertion-ordered <tt>LinkedHashMap</tt> instance with
 * the same mappings as the specified map. The <tt>LinkedHashMap</tt>
 * instance is created with a default load factor (0.75) and an initial
 * capacity sufficient to hold the mappings in the specified map.
 *
 * @param m the map whose mappings are to be placed in this map
 * @throws NullPointerException if the specified map is null
 */
public LinkedHashMap(Map<? extends K, ? extends V> m) {

    super(m);    // 调用 HashMap 对应的构造函数

    accessOrder = false;    // 迭代顺序的默认值

}
```

5 、 LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

该构造函数意在构造一个指定初始容量和指定负载因子的具有指定迭代顺序的 LinkedHashMap，其源码如下：

```
/**
 * Constructs an empty <tt>LinkedHashMap</tt> instance with the
 * specified initial capacity, load factor and ordering mode.
 *
 * @param initialCapacity the initial capacity
 * @param loadFactor      the load factor
 * @param accessOrder     the ordering mode - <tt>true</tt> for
 *                        access-order, <tt>false</tt> for insertion-order
 * @throws IllegalArgumentException if the initial capacity is negative
 *                        or the load factor is nonpositive
 */
public LinkedHashMap(int initialCapacity,
                    float loadFactor,
                    boolean accessOrder) {
    super(initialCapacity, loadFactor); // 调用 HashMap 对应的构造函数
}
```

```
        this.accessOrder = accessOrder;    // 迭代顺序的默认值
    }
```

正如我们在[《Map 综述（一）：彻头彻尾理解 HashMap》](#)一文中提到的那样，初始容量 和负载因子是影响 HashMap 性能的两个重要参数。同样地，它们也是影响 LinkedHashMap 性能的两个重要参数。此外，LinkedHashMap 增加了双向链表头结点 header 和标志位 accessOrder 两个属性用于保证迭代顺序。

6、init 方法

从上面的五种构造函数我们可以看出，无论采用何种方式创建 LinkedHashMap，其都会调用 HashMap 相应的构造函数。事实上，**不管调用 HashMap 的哪个构造函数，HashMap 的构造函数都会在最后调用一个 init() 方法进行初始化，只不过这个方法在 HashMap 中是一个空实现，而在 LinkedHashMap 中重写了它用于初始化它所维护的双向链表。**例如，HashMap 的参数为空的构造函数以及 init 方法的源码如下：

```
/**
 * Constructs an empty <tt>HashMap</tt> with the default initial capacity
 * (16) and the default load factor (0.75).
 */
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR;

    threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
}
```

```

        table = new Entry[DEFAULT_INITIAL_CAPACITY];

        init();

    }

    /**
     * Initialization hook for subclasses. This method is called
     * in all constructors and pseudo-constructors (clone, readObject)
     * after HashMap has been initialized but before any entries have
     * been inserted. (In the absence of this method, readObject would
     * require explicit knowledge of subclasses.)
     */
    void init() {

    }

```

在 `LinkedHashMap` 中，它重写了 `init` 方法以便初始化双向列表，源码如下：

```

    /**
     * Called by superclass constructors and pseudoconstructors (clone,
     * readObject) before any entries are inserted into the map. Initial
    izes
     * the chain.
     */
    void init() {

```

```

header = new Entry<K,V>(-1, null, null, null);

header.before = header.after = header;

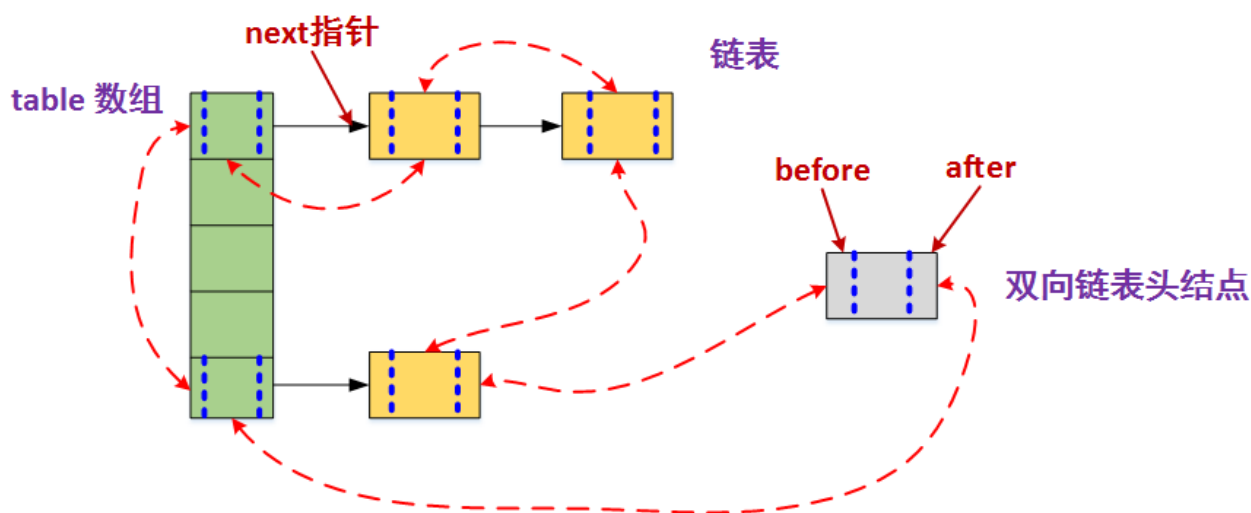
}

```

因此，我们在创建 LinkedHashMap 的同时就会不知不觉地对双向链表进行初始化。

四. LinkedHashMap 的数据结构

本质上，LinkedHashMap = HashMap + 双向链表，也就是说，HashMap 和双向链表合二为一即是 LinkedHashMap。也可以这样理解，**LinkedHashMap** 在不对 **HashMap** 做任何改变的基础上，给 **HashMap** 的任意两个节点间加了两条连线(**before** 指针和 **after** 指针)，使这些节点形成一个双向链表。在 LinkedHashMapMap 中，所有 put 进来的 Entry 都保存在 HashMap 中，但由于它又额外定义了一个以 head 为头结点的空的双向链表，因此对于每次 put 进来 Entry 还会将其插入到双向链表的尾部。



本示意图中，LinkedHashMap 一共包含五个节点。除去红色双向虚线来看，其就是一个正宗的 HashMap。在这里，用额外的红色虚线串起来的节点就是一个双向链表了。由此可见，LinkedHashMap 就是一个标准的 HashMap 与 LinkedList 的融合体。

<http://blog.csdn.net/justloveyou>

五. LinkedHashMap 的快速存取

我们知道，在 HashMap 中最常用的两个操作就是：put(Key,Value) 和 get(Key)。同样地，在 LinkedHashMap 中最常用的也是这两个操作。对于 put(Key,Value) 方法而言，LinkedHashMap 完全继承了 HashMap 的 put(Key,Value) 方法，只是对 put(Key,Value)方法所调用的 recordAccess 方法和 addEntry 方法进行了重写；对于 get(Key)方法而言，LinkedHashMap 则直接对它进行了重写。下面我们结合 JDK 源码看 LinkedHashMap 的存取实现。

1、LinkedHashMap 的存储实现 : put(key, vlaue)

上面谈到，LinkedHashMap 没有对 put(key,vlaue) 方法进行任何直接的修改，完全继承了 HashMap 的 put(Key,Value) 方法，其源码如下：

```
/**
 * Associates the specified value with the specified key in this map.
 *
 * If the map previously contained a mapping for the key, the old
 * value is replaced.
 *
 * @param key key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 *
 * @return the previous value associated with key, or null if there was
 * no mapping for key.
 *
 * Note that a null return can also indicate that the map previously
 * associated null with key.
```



```
*/
```

```
public V put(K key, V value) {
```

//当 key 为 null 时，调用 putForNullKey 方法，并将该键值对保存到 table 的第一个位置

```
if (key == null)
```

```
    return putForNullKey(value);
```

//根据 key 的 hashCode 计算 hash 值

```
int hash = hash(key.hashCode());
```

//计算该键值对在数组中的存储位置（哪个桶）

```
int i = indexFor(hash, table.length);
```

//在 table 的第 i 个桶上进行迭代，寻找 key 保存的位置

```
for (Entry<K,V> e = table[i]; e != null; e = e.next) {
```

```
    Object k;
```

//判断该条链上是否存在 hash 值相同且 key 值相等的映射，若存在，则直接覆盖 value，并返回旧 value

```
if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
```

```
    V oldValue = e.value;
```

```

        e.value = value;

        e.recordAccess(this); // LinkedHashMap 重写了 Entry 中的 recordAccess 方法--- (1)

        return oldValue;    // 返回旧值

    }

}

modCount++; //修改次数增加 1，快速失败机制

//原 Map 中无该映射，将该添加至该链的链头

addEntry(hash, key, value, i); // LinkedHashMap 重写了 HashMap 中的 createEntry 方法 ---- (2)

return null;

}

```

上述源码反映了 LinkedHashMap 与 HashMap 保存数据的过程。特别地，在 LinkedHashMap 中，它对 addEntry 方法和 Entry 的 recordAccess 方法进行了重写。下面我们对比地看一下 LinkedHashMap 和 HashMap 的 addEntry 方法的具体实现：

```

/**

    * This override alters behavior of superclass put method. It causes
    newly

    * allocated entry to get inserted at the end of the linked list and

    * removes the eldest entry if appropriate.

```

```

*

* LinkedHashMap 中的 addEntry 方法

*/

void addEntry(int hash, K key, V value, int bucketIndex) {

    //创建新的 Entry，并插入到 LinkedHashMap 中

    createEntry(hash, key, value, bucketIndex); // 重写了 HashMap 中的
createEntry 方法


    //双向链表的第一个有效节点（header 后的那个节点）为最近最少使用的节
点，这是用来支持 LRU 算法的

    Entry<K,V> eldest = header.after;

    //如果有必要，则删除掉该近期最少使用的节点，

    //这要看对 removeEldestEntry 的覆写,由于默认为 false，因此默认是不做任
何处理的。

    if (removeEldestEntry(eldest)) {

        removeEntryForKey(eldest.key);

    } else {

        //扩容到原来的 2 倍

        if (size >= threshold)

            resize(2 * table.length);

    }
}

```

```
}
```

-----我是分割线-----

```
/**  
  
 * Adds a new entry with the specified key, value and hash code to  
  
 * the specified bucket. It is the responsibility of this  
  
 * method to resize the table if appropriate.  
  
 *  
 * Subclass overrides this to alter the behavior of put method.  
  
 *  
 * HashMap 中的 addEntry 方法  
  
 */  
  
void addEntry(int hash, K key, V value, int bucketIndex) {  
  
    //获取 bucketIndex 处的 Entry  
  
    Entry<K,V> e = table[bucketIndex];  
  
    //将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry  
  
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
```

```

        //若 HashMap 中元素的个数超过极限了，则容量扩大两倍

        if (size++ >= threshold)

            resize(2 * table.length);

    }

```

由于 **LinkedHashMap** 本身维护了插入的先后顺序，因此其可以用来做缓存，14~19 行的操作就是用来支持 **LRU** 算法的，这里暂时不用去关心它。此外，在 **LinkedHashMap** 的 **addEntry** 方法中，它重写了 **HashMap** 中的 **createEntry** 方法，我们接着看一下 **createEntry** 方法：

```

void createEntry(int hash, K key, V value, int bucketIndex) {

    // 向哈希表中插入 Entry，这点与 HashMap 中相同

    //创建新的 Entry 并将其链入到数组对应桶的链表的头结点处，

    HashMap.Entry<K,V> old = table[bucketIndex];

    Entry<K,V> e = new Entry<K,V>(hash, key, value, old);

    table[bucketIndex] = e;

    //在每次向哈希表插入 Entry 的同时，都会将其插入到双向链表的尾部，

    //这样就按照 Entry 插入 LinkedHashMap 的先后顺序来迭代元素(LinkedHash
    Map 根据双向链表重写了迭代器)

    //同时，新 put 进来的 Entry 是最近访问的 Entry，把其放在链表末尾，也符
    合 LRU 算法的实现

    e.addBefore(header);

    size++;
}

```

```
}
```

由以上源码我们可以知道，在 `LinkedHashMap` 中向哈希表中插入新 `Entry` 的同时，还会通过 `Entry` 的 `addBefore` 方法将其链入到双向链表中。其中，`addBefore` 方法本质上是一个双向链表的插入操作，其源码如下：

```
//在双向链表中，将当前的 Entry 插入到 existingEntry(header)的前面
```

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

到此为止，我们分析了在 `LinkedHashMap` 中 `put` 一条键值对的完整过程。总的来说，**相比 `HashMap` 而言，`LinkedHashMap` 在向哈希表添加一个键值对的同时，也会将其链入到它所维护的双向链表中，以便设定迭代顺序。**

2、`LinkedHashMap` 的扩容操作：`resize()`

在 `HashMap` 中，我们知道随着 `HashMap` 中元素的数量越来越多，发生碰撞的概率将越来越大，所产生的子链长度就会越来越长，这样势必会影响 `HashMap` 的存取速度。为了保证 `HashMap` 的效率，系统必须要在某个临界点进行扩容处理，该临界点就是 `HashMap` 中元素的数量在数值上等于 `threshold`（`table` 数组长度*加载因子）。但是，不得不说，扩容是一个非常耗时的过程，因为它需要重新计算这些元素在新 `table` 数组中的位置并进行复制处理。所以，如果我们能够提前预知 `HashMap` 中元素的个数，那么在构造 `HashMap` 时预设元素的个数能够有效的提高 `HashMap` 的性能。

同样的问题也存在于 `LinkedHashMap` 中，因为 `LinkedHashMap` 本来就是

一个 `HashMap`，只是它还将所有 `Entry` 节点链入到了一个双向链表中。`LinkedHashMap` 完全继承了 `HashMap` 的 `resize()` 方法，只是对它所调用的 `transfer` 方法进行了重写。我们先看 `resize()` 方法源码：

```
/**
 * Rehashes the contents of this map into a new array with a
 *
 * larger capacity. This method is called automatically when the
 *
 * number of keys in this map reaches its threshold.
 *
 *
 * If current capacity is MAXIMUM_CAPACITY, this method does not
 *
 * resize the map, but sets threshold to Integer.MAX_VALUE.
 *
 * This has the effect of preventing future calls.
 *
 *
 * @param newCapacity the new capacity, MUST be a power of two;
 *
 *         must be greater than current capacity unless current
 *
 *         capacity is MAXIMUM_CAPACITY (in which case value
 *
 *         is irrelevant).
 */
void resize(int newCapacity) {
    Entry[] oldTable = table;

    int oldCapacity = oldTable.length;
```

```

        // 若 oldCapacity 已达到最大值，直接将 threshold 设为 Integer.MAX_V
        ALUE

        if (oldCapacity == MAXIMUM_CAPACITY) {

            threshold = Integer.MAX_VALUE;

            return;          // 直接返回

        }

        // 否则，创建一个更大的数组

        Entry[] newTable = new Entry[newCapacity];

        //将每条 Entry 重新哈希到新的数组中

        transfer(newTable); //LinkedHashMap 对它所调用的 transfer 方法进行
        了重写

        table = newTable;

        threshold = (int)(newCapacity * loadFactor); // 重新设定 threshol
        d

    }

```

从上面代码中我们可以看出，**Map 扩容操作的核心在于重哈希**。所谓重哈希是指重新计算原 HashMap 中的元素在新 table 数组中的位置并进行复制处理的过程。鉴于性能和 LinkedHashMap 自身特点的考量，LinkedHashMap 对重哈希过程(transfer 方法)进行了重写，源码如下：

```

/**

```



```

    * Transfers all entries to new table array. This method is called
    * by superclass resize. It is overridden for performance, as it is
    * faster to iterate using our linked list.

    */

    void transfer(HashMap.Entry[] newTable) {

        int newCapacity = newTable.length;

        // 与 HashMap 相比，借助于双向链表的特点进行重哈希使得代码更加简洁

        for (Entry<K,V> e = header.after; e != header; e = e.after) {

            int index = indexFor(e.hash, newCapacity);    // 计算每个 Entry
所在的桶

            // 将其链入桶中的链表

            e.next = newTable[index];

            newTable[index] = e;

        }

    }

```

如上述源码所示，LinkedHashMap 借助于自身维护的双向链表轻松地实现了重哈希操作。若读者想要进一步了解 HashMap 的重哈希过程，请移步我的博文 [《Map 综述（一）：彻头彻尾理解 HashMap》](#) 进行深入了解，此不赘述。

3、LinkedHashMap 的读取实现：get(Object key)

相对于 LinkedHashMap 的存储而言，读取就显得比较简单了。LinkedHashMap 中重写了 HashMap 中的 get 方法，源码如下：

```

/**
 * Returns the value to which the specified key is mapped,
 * or {@code null} if this map contains no mapping for the key.
 *
 * <p>More formally, if this map contains a mapping from a key
 * {@code k} to a value {@code v} such that {@code (key==null ? k==nu
11 :
 * key.equals(k))}, then this method returns {@code v}; otherwise
 * it returns {@code null}. (There can be at most one such mapping.)
 *
 * <p>A return value of {@code null} does not <i>necessarily</i>
 * indicate that the map contains no mapping for the key; it's also
 * possible that the map explicitly maps the key to {@code null}.
 * The {@link #containsKey containsKey} operation may be used to
 * distinguish these two cases.
 */
public V get(Object key) {

    // 根据 key 获取对应的 Entry，若没有这样的 Entry，则返回 null

    Entry<K,V> e = (Entry<K,V>)getEntry(key);

    if (e == null)        // 若不存在这样的 Entry，直接返回

        return null;

```

```

        e.recordAccess(this);

        return e.value;
    }

/**
 * Returns the entry associated with the specified key in the
 *
 * HashMap. Returns null if the HashMap contains no mapping
 *
 * for the key.
 *
 *
 * HashMap 中的方法
 *
 */
final Entry<K,V> getEntry(Object key) {

    if (size == 0) {

        return null;

    }

    int hash = (key == null) ? 0 : hash(key);

    for (Entry<K,V> e = table[indexFor(hash, table.length)];

        e != null;

```

```
        e = e.next) {

        Object k;

        if (e.hash == hash &&

            ((k = e.key) == key || (key != null && key.equals(k))))

            return e;

        }

        return null;

    }
}
```

在 LinkedHashMap 的 get 方法中，通过 HashMap 中的 getEntry 方法获取 Entry 对象。注意这里的 recordAccess 方法，如果链表中元素的排序规则是按照插入的先后顺序排序的话，该方法什么也不做；如果链表中元素的排序规则是按照访问的先后顺序排序的话，则将 e 移到链表的末尾处，笔者会在后文专门阐述这个问题。

另外，同样地，调用 LinkedHashMap 的 get(Object key)方法后，若返回值是 NULL，则也存在如下两种可能：

- 该 key 对应的值就是 null;
- HashMap 中不存在该 key。

4、LinkedHashMap 存取小结

LinkedHashMap 的存取过程基本与 HashMap 基本类似，只是在细节实现上稍有不同，这是由 LinkedHashMap 本身的特性所决定的，因为它要额外维护一个双向链表用于保持迭代顺序。在 put 操作上，虽然 LinkedHashMap 完全继承了 HashMap 的 put 操作，但是在细节上还是做了一定的调整，比如，在 LinkedHashMap 中向哈希表中插入新 Entry 的同时，还会通过 Entry 的

`addBefore` 方法将其链入到双向链表中。在扩容操作上，虽然 `LinkedHashMap` 完全继承了 `HashMap` 的 `resize` 操作，但是鉴于性能和 `LinkedHashMap` 自身特点的考量，`LinkedHashMap` 对其中的重哈希过程(`transfer` 方法)进行了重写。在读取操作上，`LinkedHashMap` 中重写了 `HashMap` 中的 `get` 方法，通过 `HashMap` 中的 `getEntry` 方法获取 `Entry` 对象。在此基础上，进一步获取指定键对应的值。

六. `LinkedHashMap` 与 LRU(Least recently used, 最近最少使用)算法

到此为止，我们已经分析完了 `LinkedHashMap` 的存取实现，这与 `HashMap` 大体相同。**`LinkedHashMap` 区别于 `HashMap` 最大的一个不同点是，前者是有序的，而后者是无序的。为此，`LinkedHashMap` 增加了两个属性用于保证顺序，分别是双向链表头结点 `header` 和标志位 `accessOrder`。**我们知道，`header` 是 `LinkedHashMap` 所维护的双向链表的头结点，而 `accessOrder` 用于决定具体的迭代顺序。实际上，`accessOrder` 标志位的作用可不像我们描述的这么简单，我们接下来仔细分析一波~

我们知道，当 `accessOrder` 标志位为 `true` 时，表示双向链表中的元素按照访问的先后顺序排列，可以看到，虽然 `Entry` 插入链表的顺序依然是按照其 `put` 到 `LinkedHashMap` 中的顺序，但 `put` 和 `get` 方法均有调用 `recordAccess` 方法（`put` 方法在 `key` 相同时会调用）。`recordAccess` 方法判断 `accessOrder` 是否为 `true`，如果是，则将当前访问的 `Entry`（`put` 进来的 `Entry` 或 `get` 出来的 `Entry`）移到双向链表的尾部（`key` 不相同时，`put` 新 `Entry` 时，会调用 `addEntry`，它会调用 `createEntry`，该方法同样将新插入的元素放入到双向链表的尾部，既符合插入的先后顺序，又符合访问的先后顺序，因为这时该 `Entry` 也被访问了）；当标志位 `accessOrder` 的值为 `false` 时，表示双向链表中的元素按照 `Entry` 插入 `LinkedHashMap` 到中的先后顺序排序，即每次 `put` 到 `LinkedHashMap` 中的 `Entry` 都放在双向链表的尾部，这样遍历双向链表时，`Entry` 的输出顺序便和插入的顺序一致，这也是默认的双向链表的存储顺序。因此，当标志位 `accessOrder` 的值为 `false` 时，虽然也会调用 `recordAccess` 方法，但不做任何操作。

注意到我们在前面介绍的 `LinkedHashMap` 的五种构造方法，前四个构造方法都将 `accessOrder` 设为 `false`，说明默认是按照插入顺序排序的；而第五个构

造方法可以自定义传入的 `accessOrder` 的值，因此可以指定双向循环链表中元素的排序规则。特别地，**当我们要用 `LinkedHashMap` 实现 LRU 算法时，就需要调用该构造方法并将 `accessOrder` 置为 `true`。**

1、put 操作与标志位 `accessOrder`

/ 将 key/value 添加到 LinkedHashMap 中

```
public V put(K key, V value) {

    // 若 key 为 null，则将该键值对添加到 table[0] 中。

    if (key == null)

        return putForNullKey(value);

    // 若 key 不为 null，则计算该 key 的哈希值，然后将其添加到该哈希值对应的链表中。

    int hash = hash(key.hashCode());

    int i = indexFor(hash, table.length);

    for (Entry<K,V> e = table[i]; e != null; e = e.next) {

        Object k;

        // 若 key 对已经存在，则用新的 value 取代旧的 value

        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

            V oldValue = e.value;

            e.value = value;

            e.recordAccess(this);
```

```

        return oldValue;

    }

}

// 若 key 不存在，则将 key/value 键值对添加到 table 中

modCount++;

//将 key/value 键值对添加到 table[i]处

addEntry(hash, key, value, i);

return null;

}

```

从上述源码我们可以看到，当要 put 进来的 Entry 的 key 在哈希表中已经存在时，会调用 Entry 的 recordAccess 方法；当该 key 不存在时，则会调用 addEntry 方法将新的 Entry 插入到对应桶的单链表的头部。我们先来看 recordAccess 方法：

```

/**

 * This method is invoked by the superclass whenever the value

 * of a pre-existing entry is read by Map.get or modified by Map.set.

 * If the enclosing Map is access-ordered, it moves the entry

 * to the end of the list; otherwise, it does nothing.

 */

void recordAccess(HashMap<K,V> m) {

    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;

```

```
//如果链表中元素按照访问顺序排序，则将当前访问的 Entry 移到双向循环链表的尾部，
```

```
//如果是按照插入的先后顺序排序，则不做任何事情。
```

```
if (lm.accessOrder) {
```

```
    lm.modCount++;
```

```
    //移除当前访问的 Entry
```

```
    remove();
```

```
    //将当前访问的 Entry 插入到链表的尾部
```

```
    addBefore(lm.header);
```

```
}
```

```
}
```

LinkedHashMap 重写了 HashMap 中的 recordAccess 方法（HashMap 中该方法为空），当调用父类的 put 方法时，在发现 key 已经存在时，会调用该方法；当调用自己的 get 方法时，也会调用到该方法。该方法提供了 LRU 算法的实现，它将最近使用的 Entry 放到双向循环链表的尾部。也就是说，当 accessOrder 为 true 时，get 方法和 put 方法都会调用 recordAccess 方法使得最近使用的 Entry 移到双向链表的末尾；当 accessOrder 为默认值 false 时，从源码中可以看出 recordAccess 方法什么也不会做。我们反过头来，再看一下 addEntry 方法：

```
/**
```

```
 * This override alters behavior of superclass put method. It causes newly
```

```
 * allocated entry to get inserted at the end of the linked list and
```

```
 * removes the eldest entry if appropriate.
```



```

*

* LinkedHashMap 中的 addEntry 方法

*/

void addEntry(int hash, K key, V value, int bucketIndex) {

    //创建新的 Entry，并插入到 LinkedHashMap 中

    createEntry(hash, key, value, bucketIndex); // 重写了 HashMap 中的
createEntry 方法


    //双向链表的第一个有效节点（header 后的那个节点）为最近最少使用的节
点，这是用来支持 LRU 算法的

    Entry<K,V> eldest = header.after;

    //如果有必要，则删除掉该近期最少使用的节点，

    //这要看对 removeEldestEntry 的覆写,由于默认为 false，因此默认是不做任
何处理的。

    if (removeEldestEntry(eldest)) {

        removeEntryForKey(eldest.key);

    } else {

        //扩容到原来的 2 倍

        if (size >= threshold)

            resize(2 * table.length);

    }
}

```

```

    }

    void createEntry(int hash, K key, V value, int bucketIndex) {

        // 向哈希表中插入 Entry，这点与 HashMap 中相同

        //创建新的 Entry 并将其链入到数组对应桶的链表的头结点处，

        HashMap.Entry<K,V> old = table[bucketIndex];

        Entry<K,V> e = new Entry<K,V>(hash, key, value, old);

        table[bucketIndex] = e;


        //在每次向哈希表插入 Entry 的同时，都会将其插入到双向链表的尾部，

        //这样就按照 Entry 插入 LinkedHashMap 的先后顺序来迭代元素(LinkedHash
        Map 根据双向链表重写了迭代器)

        //同时，新 put 进来的 Entry 是最近访问的 Entry，把其放在链表末尾，也符
        合 LRU 算法的实现

        e.addBefore(header);

        size++;

    }

```

同样是将新的 Entry 链入到 table 中对应桶中的单链表中，但可以在 createEntry 方法中看出，同时也会把新 put 进来的 Entry 插入到了双向链表的尾部。从插入顺序的层面来说，新的 Entry 插入到双向链表的尾部可以实现按照插入的先后顺序来迭代 Entry，而从访问顺序的层面来说，新 put 进来的 Entry 又是最近访问的 Entry，也应该将其放在双向链表的尾部。在上面的 addEntry 方法中还调用了 removeEldestEntry 方法，该方法源码如下：

```
/**
```

* Returns `true` if this map should remove its eldest entry.

* This method is invoked by `put` and `putAll` after

* inserting a new entry into the map. It provides the implementor

* with the opportunity to remove the eldest entry each time a new one

* is added. This is useful if the map represents a cache: it allows

* the map to reduce memory consumption by deleting stale entries.

*

* `Sample use: this override will allow the map to grow up to 100`

* `entries and then delete the eldest entry each time a new entry is`

* `added, maintaining a steady state of 100 entries.`

* `<pre>`

```
*     private static final int MAX_ENTRIES = 100;
```

*

```
*     protected boolean removeEldestEntry(Map.Entry eldest) {
```

```
*         return size() > MAX_ENTRIES;
```

```
*     }
```

* `</pre>`

*

* `This method typically does not modify the map in any way,`

* `instead allowing the map to modify itself as directed by its`

```

* return value. It <i>is</i> permitted for this method to modify
* the map directly, but if it does so, it <i>must</i> return
* <tt>>false</tt> (indicating that the map should not attempt any
* further modification). The effects of returning <tt>>true</tt>
* after modifying the map from within this method are unspecified.
*
* <p>This implementation merely returns <tt>>false</tt> (so that this
* map acts like a normal map - the eldest element is never removed).
*
* @param   eldest The least recently inserted entry in the map, or
if         this is an access-ordered map, the least recently accesse
d          ed
*         entry. This is the entry that will be removed if this
*         method returns <tt>>true</tt>. If the map was empty prior
*         to the <tt>put</tt> or <tt>putAll</tt> invocation result
ing        ing
*         in this invocation, this will be the entry that was just
*         inserted; in other words, if the map contains a single
*         entry, the eldest entry is also the newest.
*
* @return  <tt>>true</tt> if the eldest entry should be removed
*         from the map; <tt>>false</tt> if it should be retained.

```

```

    */

    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {

        return false;

    }
}

```

该方法是用来被重写的，一般地，如果用 `LinkedHashMap` 实现 LRU 算法，就要重写该方法。比如可以将该方法覆写为如果设定的内存已满，则返回 `true`，这样当再次向 `LinkedHashMap` 中 `putEntry` 时，在调用的 `addEntry` 方法中便会将近期最少使用的节点删除掉（`header` 后的那个节点）。在第七节，笔者便重写了该方法并实现了一个名副其实的 LRU 结构。

2、get 操作与标志位 `accessOrder`

```

/**

 * Returns the value to which the specified key is mapped,

 * or {@code null} if this map contains no mapping for the key.

 *

 * <p>More formally, if this map contains a mapping from a key

 * {@code k} to a value {@code v} such that (key==null ? k==null :

 11 : key.equals(k)), then this method returns v; otherwise

 * it returns null. (There can be at most one such mapping.)

 *

```

```

* <p>A return value of {@code null} does not <i>not necessarily</i>
* indicate that the map contains no mapping for the key; it's also
* possible that the map explicitly maps the key to {@code null}.
* The {@link #containsKey containsKey} operation may be used to
* distinguish these two cases.
*/

public V get(Object key) {

    // 根据 key 获取对应的 Entry，若没有这样的 Entry，则返回 null

    Entry<K,V> e = (Entry<K,V>)getEntry(key);

    if (e == null)        // 若不存在这样的 Entry，直接返回

        return null;

    e.recordAccess(this);

    return e.value;

}

```

在 LinkedHashMap 中进行读取操作时，一样也会调用 recordAccess 方法。上面笔者已经表述的很清楚了，此不赘述。

3、LinkedListMap 与 LRU 小结

使用 **LinkedHashMap** 实现 **LRU** 的必要前提是将 **accessOrder** 标志位设为 **true** 以便开启按访问顺序排序的模式。我们可以看到，无论是 put 方法还是 get 方法，都会导致目标 Entry 成为最近访问的 Entry，因此就把该 Entry 加入到了双向链表的末尾：get 方法通过调用 recordAccess 方法来实现；put 方法在覆

盖已有 key 的情况下，也是通过调用 recordAccess 方法来实现，在插入新的 Entry 时，则是通过 createEntry 中的 addBefore 方法来实现。这样，我们便把最近使用的 Entry 放入到了双向链表的后面。多次操作后，双向链表前面的 Entry 便是最近没有使用的，这样当节点个数满的时候，删除最前面的 Entry(head 后面的那个 Entry)即可，因为它就是最近最少使用的 Entry。

七. 使用 LinkedHashMap 实现 LRU 算法

如下所示，笔者使用 LinkedHashMap 实现一个符合 LRU 算法的数据结构，该结构最多可以缓存 6 个元素，但元素多余六个时，会自动删除最近最久没有被使用的元素，如下所示：

```
/**
 * Title: 使用 LinkedHashMap 实现 LRU 算法
 * Description:
 * @author rico
 * @created 2017 年 5 月 12 日 上午 11:32:10
 */

public class LRU<K,V> extends LinkedHashMap<K, V> implements Map<K, V>{

    private static final long serialVersionUID = 1L;

    public LRU(int initialCapacity,

               float loadFactor,
```

```

        boolean accessOrder) {

    super(initialCapacity, loadFactor, accessOrder);

}

/**
 * @description 重写 LinkedHashMap 中的 removeEldestEntry 方法，当 LRU 中
元素多余 6 个时，
 *
 *      删除最不经常使用的元素
 *
 * @author rico
 *
 * @created 2017 年 5 月 12 日 上午 11:32:51
 *
 * @param eldest
 *
 * @return
 *
 * @see java.util.LinkedHashMap#removeEldestEntry(java.util.Map.Entr
y)
 */
@Override

protected boolean removeEldestEntry(java.util.Map.Entry<K, V> eldest) {

    // TODO Auto-generated method stub

    if(size() > 6){

        return true;

    }
}

```



```
        return false;

    }

    public static void main(String[] args) {

        LRU<Character, Integer> lru = new LRU<Character, Integer>(

            16, 0.75f, true);

        String s = "abcdefghijkl";

        for (int i = 0; i < s.length(); i++) {

            lru.put(s.charAt(i), i);

        }

        System.out.println("LRU 中 key 为 h 的 Entry 的值为: " + lru.get('h'

        ));

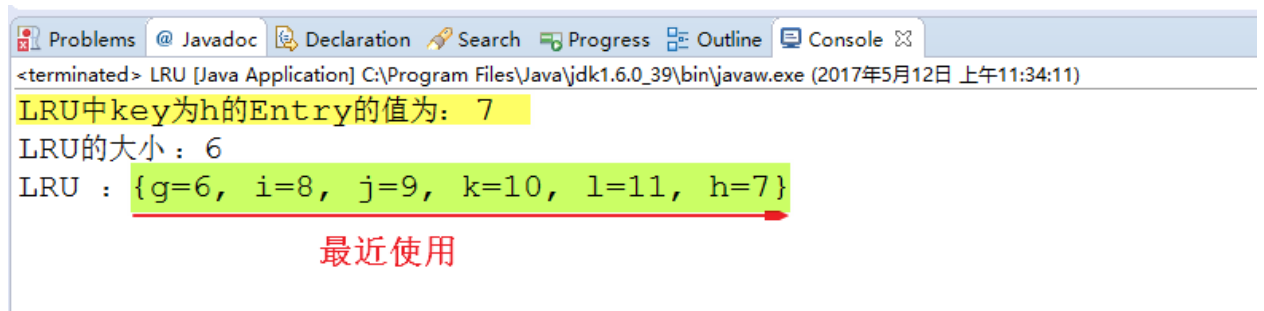
        System.out.println("LRU 的大小 : " + lru.size());

        System.out.println("LRU : " + lru);

    }

}
```

下图是程序的运行结果：



The screenshot shows an IDE console window with the following content:

```
<terminated> LRU [Java Application] C:\Program Files\Java\jdk1.6.0_39\bin\javaw.exe (2017年5月12日 上午11:34:11)
LRU中key为h的Entry的值为: 7
LRU的大小: 6
LRU : {g=6, i=8, j=9, k=10, l=11, h=7}
```

A red arrow points to the entry 'h=7' in the LRU set, with the text '最近使用' (Recently Used) written below it.

八. LinkedHashMap 有序性原理分析

如前文所述，LinkedHashMap 增加了双向链表头结点 `header` 和 标志位 `accessOrder` 两个属性用于保证迭代顺序。但是要想真正实现其有序性，还差临门一脚，那就是重写 `HashMap` 的迭代器，其源码实现如下：

```
private abstract class LinkedHashMapIterator<T> implements Iterator<T> {

    Entry<K,V> nextEntry    = header.after;

    Entry<K,V> lastReturned = null;

    /**
     * The modCount value that the iterator believes that the backing
     * List should have.  If this expectation is violated, the iterator
     * has detected concurrent modification.
     */
    int expectedModCount = modCount;
```

```

public boolean hasNext() {           // 根据双向列表判断

    return nextEntry != header;

}

public void remove() {

    if (lastReturned == null)

        throw new IllegalStateException();

    if (modCount != expectedModCount)

        throw new ConcurrentModificationException();

    LinkedHashMap.this.remove(lastReturned.key);

    lastReturned = null;

    expectedModCount = modCount;

}

Entry<K,V> nextEntry() {           // 迭代输出双向链表各节点

    if (modCount != expectedModCount)

        throw new ConcurrentModificationException();

    if (nextEntry == header)

```

```

        throw new NoSuchElementException();

        Entry<K,V> e = lastReturned = nextEntry;

        nextEntry = e.after;

        return e;
    }
}

// Key 迭代器, KeySet

private class KeyIterator extends LinkedHashMapIterator<K> {

    public K next() { return nextEntry().getKey(); }

}

// Value 迭代器, Values(Collection)

private class ValueIterator extends LinkedHashMapIterator<V> {

    public V next() { return nextEntry().value; }

}

// Entry 迭代器, EntrySet

private class EntryIterator extends LinkedHashMapIterator<Map.Entry<K,V>> {

```

```
public Map.Entry<K,V> next() { return nextEntry(); }  
}
```

从上述代码中我们可以知道，**LinkedHashMap** 重写了 **HashMap** 的迭代器，它使用其维护的双向链表进行迭代输出。

九. 更多

如果读者需要深入了解 **HashMap**，请移步我的另一篇博文[《Map 综述\(一\): 彻头彻尾理解 HashMap》](#)。

更多关于 **LinkedList** 的介绍，请移步我的博文[《Java Collection Framework : List》](#)。

更多关于哈希(Hash)和 **equals** 方法的介绍，请移步我的博文[《Java 中的 ==, equals 与 hashCode 的区别与联系》](#)。

更多关于 **Java SE** 进阶 方面的内容,请关注我的专栏[《Java SE 进阶之路》](#)。本专栏主要研究 **Java** 基础知识、**Java** 源码和设计模式，从初级到高级不断总结、剖析各知识点的内在逻辑，贯穿、覆盖整个 **Java** 知识面，在一步步完善、提高把自己的同时，把对 **Java** 的所学所思分享给大家。万丈高楼平地起，基础决定你的上限，让我们携手一起勇攀 **Java** 之巅...