LRH1993 linkedhashmap 0d60933 on 22 Aug 2017

1 contributor

142 lines (114 sloc) | 5.26 KB

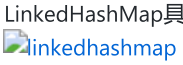
一、 概述

在理解了HashMap后，我们来学习LinkedHashMap的工作原理及实现。首先还是类似的，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：


没错，正如官方文档所说：

> **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**).

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、 三个重点实现的函数

在HashMap中提到了下面的定义：

```
// Callbacks to allow LinkedHashMap post-actions
void afterNodeAccess(Node<K,V> p) { }
void afterNodeInsertion(boolean evict) { }
void afterNodeRemoval(Node<K,V> p) { }
```

LinkedHashMap继承于HashMap，因此也重新实现了这3个函数，顾名思义这三个函数的作用分别是：节点访问后、节点插入后、节点移除后做一些事情。

afterNodeAccess函数

```
void afterNodeAccess(Node<K,V> e) { // move node to last
    LinkedHashMap.Entry<K,V> last;
    // 如果定义了accessOrder，那么就保证最近访问节点放到最后
    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;
    }
}
```

就是说在进行put之后就算是对节点的访问了，那么这个时候就会更新链表，把最近访问的放到最后，保证链表。

afterNodeInsertion函数

```
void afterNodeInsertion(boolean evict) { // possibly remove eldest
    LinkedHashMap.Entry<K,V> first;
    // 如果定义了溢出规则，则执行相应的溢出
    if (evict && (first = head) != null && removeEldestEntry(first)) {
        K key = first.key;
        removeNode(hash(key), key, null, false, true);
    }
}
```

如果用户定义了`removeEldestEntry`的规则，那么便可以执行相应的移除操作。

afterNodeRemoval函数

```
void afterNodeRemoval(Node<K,V> e) { // unlink
    // 从链表中移除节点
    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;
}
```

这个函数是在移除节点后调用的，就是将节点从双向链表中删除。

我们从上面3个函数看出来，基本上都是为了**保证双向链表中的节点次序或者双向链表容量**所做的一些额外的事情，目的就是保持双向链表中节点的顺序要从eldest到youngest。

三、 put和get函数

`put`函数在LinkedHashMap中未重新实现，只是实现了`afterNodeAccess`和`afterNodeInsertion`两个回调函数。`get`函数则重新实现并加入了`afterNodeAccess`来保证访问顺序，下面是`get`函数的具体实现：

```
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;
}
```

值得注意的是，在accessOrder模式下，只要执行get或者put等操作的时候，就会产生`structural modification`。官方文档是这么描述的：

> A structural modification is any operation that adds or deletes one or more mappings or, in the case of access-ordered linked hash maps, affects iteration order. In insertion-ordered linked hash maps, merely changing the value associated with a key that is already contained in the map is not a structural modification. **In access-ordered linked hash maps, merely querying the map with get is a structural modification**.

总之，LinkedHashMap不愧是HashMap的儿子，和老子太像了，当然，青出于蓝而胜于蓝，LinkedHashMap的其他的操作也基本上都是为了维护好那个具有访问顺序的双向链表。