# LinkedHashMap

继承HashMap,实现Map接口；是由哈希表和链表组合而成的；具有hashMap的所有操作，和hashMap不同的是，它也是可以维护节点顺序的双向链表，LinkedHashMap节点Entry内部除了继承HashMap的Node属性，还有before 和 after用于标识前置节点和后置节点。可以实现按插入的顺序或访问顺序排序。节点顺序默认是插入的顺序；如果再次插入相同key的节点，此时顺序是不变的；LinkedHashMap操作的效率略低于HashMap，因为有维护链表的额外操作；

LinkedhashMap允许key和value都为null；

LinkedHashMap的收集视图的迭代需要与地图*大小*成比例的时间，无论其容量如何。 HashMap的迭代可能更昂贵，需要与其*容量*成比例的时间。

LinkedHashMap不是线程安全的，多线程环境小操作并有一个线程在结构上修改了结构，那么就不行进行同步；

使用Collections.synchronizeMap来保证LinkedHashMap的线程安全，最好是在创建时完成

LinkedHashMap map=new Collections.synchronizedMap(new LinkedHashMap())

## 构造函数

无参构造函数，调用hashMap的无参构造函数，默认数组长度为16，加载因子为0.75

**public** LinkedHashMap() {

**super**();

accessOrder = **false**;

}

指定初始容量，加载因子为0.75

**public** LinkedHashMap(**int** initialCapacity) {

**super**(initialCapacity);

accessOrder = **false**;

}

指定初始容量和加载因子

**public** LinkedHashMap(**int** initialCapacity, **float** loadFactor) {

**super**(initialCapacity, loadFactor);

accessOrder = **false**;

}

**public** LinkedHashMap(**int** initialCapacity, **float** loadFactor, **boolean** accessOrder) {

**super**(initialCapacity, loadFactor);

**this**.accessOrder = accessOrder;

}

**public** LinkedHashMap(Map<? **extends** K, ? **extends** V> m) {

**super**();

accessOrder = **false**;

putMapEntries(m, **false**);

}

## 内部节点结构

**static** **class** Entry<K,V> **extends** HashMap.Node<K,V> {

Entry<K,V> before, after;

Entry(**int** hash, K key, V value, Node<K,V> next) {

**super**(hash, key, value, next);

}

}

继承HashMap的Node内部类的Entry，新增了before和after2个用来指向上一个节点和后一个节点的引用，用来维护双向链表的顺序

## 变量

**transient** LinkedHashMap.Entry<K,V> head;指向链表的添加时间最久的结点

**transient** LinkedHashMap.Entry<K,V> tail;指向链表的添加时间最近的节点

final boolean accessOrder;控制访问后的节点顺序，设置为true，每次访问一个元素（get方法，put方法是当key已经存在时修改节点的value，修改后将节点位移到链表最后），将将该元素的节点

调整为最后一个节点，改变了LinkedHashMap的节点顺序；设置为false，访问不会改变存储顺序

## 常见操作

### add

添加add方法，调用HashMap的方法，添加流程和HashMap基本一致，只是在添加完成后LinkedHashMap

重写了HashMap的方法afterNodeAccess和linkNodeLast，这些方法是为了维护链表的顺序

afterNodeAccess是在添加的key已经存在时执行的方法

linkNodeLast新增节点的时候执行的（在创建链表节点newNode()方法中执行，LinkedHashMap重写了HashMap的newNode方法，创建节点的同时维护添加顺序；当创建TreeNode节点时LinkedHashMap也重写了newTreeNode方法，创建树节点的同时维护了添加顺序）

1. 添加的key已经存在（accessOrder为true）

参数node是访问的节点，

当accessOrder设置为true，并且访问的节点不是指向链表尾节点才会调整当前节点为尾节点

调整过程

1.访问节点是head节点，将head指向当前节点的after节点（**if** (b == **null**) head = a;）,然后将tail节点的after节点指向当前节点，让当前节点的before节点指向tail节点，最后将tail引用指向当前节点（tail = p）；

2.访问节点不是head节点，将当前节点的before节点指向当前节点的after节点（b.after = a），after节点的before节点指向当前节点的before节点（a.before = b;）

将tail节点的after节点指向当前节点，当前节点的before节点指向tail节点，然后让引用tail指向当前节点，将当前节点变为tail节点（p.before = last; last.after = p;tail = p）

**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;

}

}

添加节点时直接将tail节点的next指向新增的节点，新增节点的before节点设置为tail节点，然后将tail节点指向新增的节点

**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;

}

}

### remove

删除节点：指向HashMap的remove(key)方法,指向过程基本一致，LinkedHashMap重写了afterNodeRemoval()方法

用来维护删除后的链表顺序；（remove（key）方法是清除了节点在单向链表或者树型结构上的关系，并没有清除双向链表上的关系，afterNodeRemoval方法就是用来清除删除节点在双向链表上的关联关系），

**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;

}

get（key）方法,重写了HashMap的get方法，获取节点方法还是沿用的HashMap的getNode方法

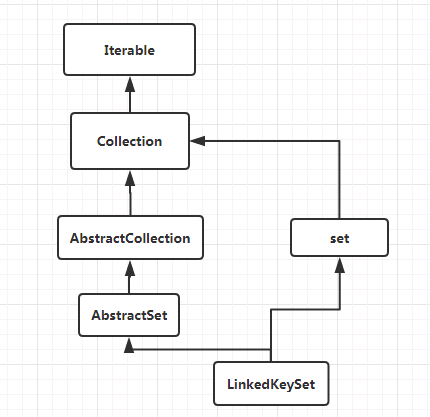
增加了顺序的维护（当accessOrder为true时，修改访问节点为tail）

**if** (accessOrder)

afterNodeAccess(e);

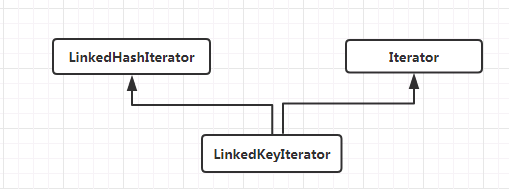
维护访问后的方法afterNodeAccess，将访问节点修改为尾节点，访问节点的before节点的after指向访问节点的after，访问节点的after节点的before节点指向访问节点的before

获取Key的集合：实现原来和HashMap类似



### keySet()

定义内部类LinkedKeySet实现Iterable接口，LinkedKeySet重写iterator方法，返回实现了Iterator接口的迭代对象



通过keySet（）方法创建LinkedKeySet对象，循环遍历时，通过while循环访问迭代对象的next方法返回key

**final** **class** LinkedKeyIterator **extends** LinkedHashIterator

**implements** Iterator<K> {

**public** **final** K next() { **return** nextNode().getKey(); }

}

获取value的集合:实现和key类似

LinkedValues实现Iterable接口

LinkedValueIterator迭代对象

获取Node集合