一、基础概念

1. 基本介绍

(1) LinkedHashMap是HashMap的子类，但内部还有一个双向链表维护键值对的顺序，每个键值对既位于哈希表中，也位于这个双向链表中。

(2) LinkedHashMap支持两种顺序，一种是插入顺序，另外一种是访问顺序。

插入顺序：先添加的在前面，后添加的在后面，修改操作不影响顺序。

访问顺序：所谓访问是指get/put操作，对一个键执行get/put操作后，**其对应的键值对会移到链表末尾**，所以，最末尾的是最近访问的，最开始的最久没被访问的，这种顺序就是访问顺序。

2. 基本用法

(1) 构造方法分类

① 插入顺序的构造方法

public LinkedHashMap()

public LinkedHashMap(int initialCapacity)

public LinkedHashMap(int initialCapacity, float loadFactor)

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

② 访问顺序的构造方法

public LinkedHashMap(int initialCapacity,

float loadFactor,

boolean accessOrder)

③ 参数accessOrder就是用来指定是否按访问顺序，如果为true，就是访问顺序。

(2) 插入顺序的例子（保持先进先出，类似于队列）

① 程序代码：//需要通过视图来取出

Map<String,Integer> seqMap = new LinkedHashMap<>();

seqMap.put("c", 100);

seqMap.put("d", 200);

seqMap.put("a", 500);

seqMap.put("d", 300); //按照c，d，a顺序插入

for(Entry<String,Integer> entry : seqMap.entrySet()){

System.out.println(entry.getKey()+" "+entry.getValue());

}

② 输出：c 100 d 300 a 500

③ 应用背景：

Map经常用来处理一些数据，其处理模式是，接受一些键值对作为输入，处理，然后输出，输出时希望保持原来的顺序。比如一个配置文件，其中有一些键值对形式的配置项，但其中有一些键是重复的，希望保留最后一个值，但还是按原来的键顺序输出，LinkedHashMap就是一个合适的数据结构。

(3) 访问顺序（先进后出，类似于栈）

① 程序代码：

Map<String,Integer> accessMap = new LinkedHashMap<>(16, 0.75f, true);

accessMap.put("c", 100);

accessMap.put("d", 200);

accessMap.put("a", 500);

accessMap.get("c");

accessMap.put("d", 300);

for(Entry<String,Integer> entry : accessMap.entrySet()){

System.out.println(entry.getKey()+" "+entry.getValue());

}

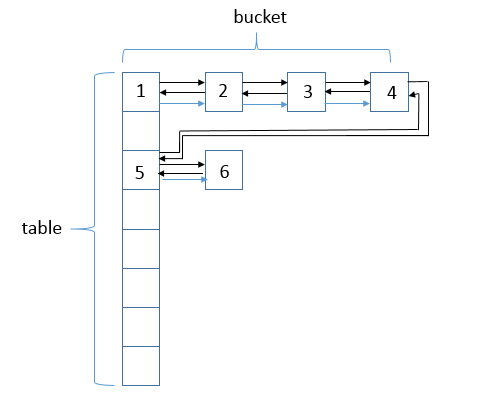
//由于d是最后执行put操作所以放到最后

② 输出：a 500 c 100 d 300

③ 应用背景：（常用于LRU缓存）

一般而言，缓存容量有限，不能无限存储所有数据，如果缓存满了，当需要存储新数据时，就需要一定的策略将一些老的数据清理出去，这个策略一般称为替换算法。LRU是一种流行的替换算法，它的全称是Least Recently Used，最近最少使用，它的思路是，**最近刚被使用的很快再次被用的可能性最高，而最久没被访问的很快再次被用的可能性最低，所以被优先清理。**（所以刚被使用的放到最底层，防止被清理。）

二、实现原理



LikedHashMap其结构在HashMap结构上增加了链表结构。数据结构为（数组 + 单链表 + 红黑树 + 双链表）

1. 主要属性

transient LinkedHashMap.Entry<K,V> head; //双向链表的头部

transient LinkedHashMap.Entry<K,V> tail; //双向链表的尾部

final boolean accessOrder; //true为插入顺序，false为访问顺序

2. 内部类Entry

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

//继承与hashMap中的node类，带有双向链表的指针before和after

Entry<K,V> before, after;

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

super(hash, key, value, next);

}

}

3. 构造方法

(1) 默认方法

public LinkedHashMap() {

super();

accessOrder = false;

}

(2) 传入初始化容量

public LinkedHashMap(int initialCapacity) {

super(initialCapacity);

accessOrder = false;

}

(3) 传入初始化容量和加载因子

public LinkedHashMap(int initialCapacity, float loadFactor) {

super(initialCapacity, loadFactor);

accessOrder = false;

}

(4) 传入初始化容量和加载因子，声明为访问顺序

public LinkedHashMap(int initialCapacity,

float loadFactor,

boolean accessOrder) {

super(initialCapacity, loadFactor);

this.accessOrder = accessOrder;

}

4. 添加元素put()

由于LikedHashMap继承与HashMap，所以可以直接调用HashMap的put方法，只需重写关键方法即可，就是方法newNode()

(1) 源码：

① Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {

//当哈希桶中节点为node时，调用这个方法；当为treeNode，调用newTreeNode

//方法

LinkedHashMap.Entry<K,V> p =

new LinkedHashMap.Entry<K,V>(hash, key, value, e); // 生成Node结点

// 将该结点插入双链表末尾

linkNodeLast(p);

return p;

}

② private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {

LinkedHashMap.Entry<K,V> last = tail; //取得当前尾部的对象

tail = p; //将尾部设为p

if (last == null) //当尾部为null，说明此时数组无元素

head = p;

else {

p.before = last;//在p和last之间建立双向链

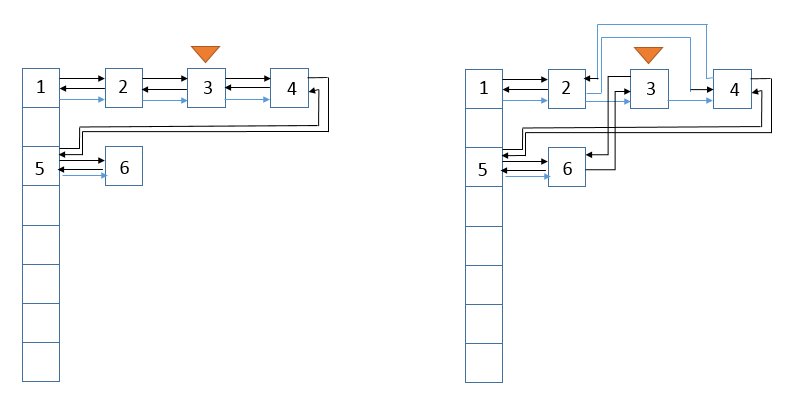
last.after = p;

}

}

③ put执行成功后的回调函数（作用：当设为访问顺序时，需要将刚被访问的元素移动到链表的最后一个位置）

void afterNodeAccess(Node<K,V> e)



5. 获取元素

public V get(Object key) {

Node<K,V> e;

if ((e = getNode(hash(key), key)) == null)

return null; //调用HashMap中的getNode方法

if (accessOrder)

afterNodeAccess(e); //访问成功后的回调函数，当设为访问顺序时才会执行

return e.value;

}

三、总结

1. LinkedHashMap内部**维护了一个单独的双向链表，每个节点即位于哈希表中，也位于双向链表中**，在链表中的顺序默认是插入顺序，也可以配置为访问顺序，LinkedHashMap及其节点类LinkedHashMap.Entry重写了若干方法以维护这种关系。

2. Map接口的实现类都有一个对应的Set接口的实现类，比如HashMap有HashSet，TreeMap有TreeSet，LinkedHashMap也不例外，它也有一个对应的Set接口的实现类LinkedHashSet。LinkedHashSet是HashSet的子类，但它内部的Map的实现类是LinkedHashMap，所以它也可以保持插入顺序，