基于LinkedHashMap的LRU算法实现及LinkedHashMap介绍

# 尾插法与头插法：

新元素放入到链表尾部，叫做尾插法；最近最少使用的元素位于头部；

新元素放入到链表头部，叫做头插法；最近最少使用的元素位于尾部。

# LinkedHashMap

<https://www.jianshu.com/p/b8b00da28a49?utm_campaign=haruki&utm_content=note&utm_medium=reader_share&utm_source=qq>

一提到LRU，我们就应该想到**LinkedHashMap**。LRU是通过**双向链表**来实现的。当某个位置的数据被命中，通过调整该数据的位置，将其移动至**尾部**。**新插入的元素也是直接放入尾部(尾插法)**。这样一来，最近被命中的元素就向尾部移动，那么**链表的头部就是最近最少使用的元素所在的位置**。

HashMap的**afterNodeAccess**()、**afterNodeInsertion**()、**afterNodeRemoval**()方法都是空实现，留着**LinkedHashMap**去重写。**LinkedHashMap靠重写这3个方法就完成了核心功能的实现。**不得不感叹，LinkedHashMap设计之妙。

## HashMap的三个空方法：

// Callbacks to allow **LinkedHashMap** post-actions后操作

void **afterNodeAccess**(Node<K,V> p) { }//访问access之后

void **afterNodeInsertion**(boolean evict) { }//插入insertion之后

void **afterNodeRemoval**(Node<K,V> p) { }//删除removal之后

## LinkedHashMap的实现

### afterNodeRemoval

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;

}

final Node<K,V> **removeNode**(int hash, Object key, Object value,

boolean matchValue, boolean movable) {

Node<K,V>[] tab; Node<K,V> p; int n, index;

if ((tab = table) != null && (n = tab.length) > 0 &&

(p = tab[index = (n - 1) & hash]) != null) {

Node<K,V> node = null, e; K k; V v;

if (p.hash == hash &&

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

node = p;

else if ((e = p.next) != null) {

if (p instanceof TreeNode)

node = ((TreeNode<K,V>)p).getTreeNode(hash, key);

else {

do {

if (e.hash == hash &&

((k = e.key) == key ||

(key != null && key.equals(k)))) {

node = e;

break;

}

p = e;

} while ((e = e.next) != null);

}

}

if (node != null && (!matchValue || (v = node.value) == value ||

(value != null && value.equals(v)))) {

if (node instanceof TreeNode)

((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);

else if (node == p)

tab[index] = node.next;

else

p.next = node.next;

++modCount;

--size;

**afterNodeRemoval(node);**

return node;

}

}

return null;

}

### afterNodeInsertion

void **afterNodeInsertion**(boolean evict) { // possibly remove eldest

LinkedHashMap.Entry<K,V> first;

if (evict && (first = head) != null && **removeEldestEntry(first)**) {

K key = first.key;

**removeNode(hash(key), key, null, false, true);**

}

}

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

return **false**;

}

我们可以看到插入数据的时候，如果**removeEldestEntry(first)**返回true，按照LRU策略，那么会删除头节点。

**因此需要继承覆写removeEldestEntry(Map.Entry<K,V> eldest)方法，指定什么情况下返回true，即什么情况下删除头节点。**

### afterNodeAccess

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;

}

}

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;

}

**在LinkedHashMap的get()方法**中，我们每次获取元素的时候，**都要调用afterNodeAccess(e)都要将元素移动到尾部**。(尾插法)

accessOrder 访问顺序，默认为true就是FIFO，若设置为true，就是按照访问顺序。

我们想要**LinkedHashMap实现LRU功能，accessOrder必须为true**。如果accessOrder为false，那就是FIFO了。

## 如何使用LinkedHashMap实现LRU

LinkedHashMap大体的LRU架子都为我们搭好了。那我们怎么去基于LinkedHashMap实现LRU呢。先别慌，我们先看看MyBatis中的LruCache是怎么实现的。

public class **LruCache** implements **Cache** {

private final **Cache delegate;**

private Map<Object, Object> keyMap;

private Object eldestKey;

public **LruCache**(Cache delegate) {

**this.delegate = delegate;**

setSize(1024);

}

@Override

public String getId() {

return delegate.getId();

}

@Override

public int getSize() {

return delegate.getSize();

}

public void setSize(final int size) {

**keyMap = new LinkedHashMap<Object, Object>(size, .75F, true) {**

**private static final long serialVersionUID = 4267176411845948333L;**

**@Override**

**protected boolean removeEldestEntry(Map.Entry<Object, Object> eldest) {**

**boolean tooBig = size() > size;**

**if (tooBig) {**

**eldestKey = eldest.getKey();**

**}**

**return tooBig;**

**}**

**};**

}

@Override

public void putObject(Object key, Object value) {

delegate.putObject(key, value);

cycleKeyList(key);

}

@Override

public Object getObject(Object key) {

keyMap.get(key); //touch

return delegate.getObject(key);

}

@Override

public Object removeObject(Object key) {

return delegate.removeObject(key);

}

@Override

public void clear() {

delegate.clear();

keyMap.clear();

}

@Override

public ReadWriteLock getReadWriteLock() {

return null;

}

private void cycleKeyList(Object key) {

keyMap.put(key, key);

if (eldestKey != null) {

delegate.removeObject(eldestKey);

eldestKey = null;

}

}

}

## 基于LinkedHashMap实现LRU

可以照葫芦画瓢，来手写LRU。其实我们只要把**accessOrder设置为true**，重写**removeEldestEntry(eldest)即可**。我们在removeEldestEntry(eldest)加上什么时候执行LRU操作的逻辑，比如**map里面的元素数量超过指定的大小，开始删除最近最少使用的元素，为后续新增的元素腾出位置来**。

我们来看看自己手写的LRU例子。

### 思路分析

1.首先往map里面添加了5个元素，使用的是尾插法，顺序应该是1,2,3,4,5。

2.调用了map.put("6", "6")，通过尾插法插入元素6，此时的顺序是1,2,3,4,5,6，然后 LinkedHashMap调用removeEldestEntry()，map里面的元素数量是6，大于指定的size，返回true。LinkedHashMap会删除头节点的元素，此时顺序应该是2,3,4,5,6。

3.调用了map.get("2")，元素2被命中，元素2需要移动到链表尾部，此时的顺序是3,4,5,6,2

4.调用了map.put("7", "7")，和步骤2一样的操作。此时的顺序是4,5,6,2,7

5.调用了map.get("4"),和步骤3一样的操作。此时的顺序是5,6,2,7,4

### 代码：

/\*\*

\* 基于LinkedHashMap实现LRU算法

\* 两点：1、accessOrder必须为true；2. 覆盖removeEldestEntry方法；

\*

\* 注意：LinkedHashMap默认采用的是尾插法：

\* 即新元素放在尾部，满时删除头元素。

\*/

//测试

@Test

public void test(){

final int size = 4;

LinkedHashMap<Integer,Integer> myLRU = new LinkedHashMap<Integer, Integer>(size,0.75F,true){

@Override

protected boolean removeEldestEntry(HashMap.Entry<Integer, Integer> eldest) {

boolean tooBig = size() > size;

if(tooBig) System.out.println("最近最少使用的key = "+eldest.getKey());

return tooBig;

}

};

myLRU.put(1,10);

System.out.println(myLRU.toString());

myLRU.put(2,14);

System.out.println(myLRU.toString());

myLRU.put(3,15);

System.out.println(myLRU.toString());

myLRU.put(4,18);

System.out.println(myLRU.toString());

myLRU.put(5,12);

System.out.println(myLRU.toString());

myLRU.put(6,99);

System.out.println(myLRU.toString());

myLRU.get(4);

System.out.println(myLRU.toString());

myLRU.get(5);

System.out.println(myLRU.toString());

}

结果：

{1=10}

{1=10, 2=14}

{1=10, 2=14, 3=15}

{1=10, 2=14, 3=15, 4=18}

最近最少使用的key = 1

{2=14, 3=15, 4=18, 5=12}

最近最少使用的key = 2

{3=15, 4=18, 5=12, 6=99}

{3=15, 5=12, 6=99, 4=18}

{3=15, 6=99, 4=18, 5=12}

# 继续讲述LinkedHashMap

HashMap与LinkedHashMap的区别：

HashMap在遍历的时候，获取元素是无序的，即与插入顺序不一致；

而LinkedHashMap在遍历的时候保证了顺序性。

其中LinkedHashMap有个参数accessOrder，指定了两种顺序：

<accessOrder字面翻译为访问顺序>。

当**accessOrder=false**默认值，是按照**插入顺序**；

当**accessOrder=true**时，是按照访问顺序，即用于LRU。MyBatis中的缓存就会基于LinkedHashMap实现的。

实现原理：

**当accessOrder=true时**，主要是通过在get方法操作末尾执行afterNodeAccess方法，将该节点移动到尾部(尾插法)。

### LinkedHashMap的构造方法

**默认初始容量为16，loadFactor为0.75F，accessOrder为fasle。**

public **LinkedHashMap**(int initialCapacity,

float loadFactor,

boolean **accessOrder**) {

super(initialCapacity, loadFactor);

**this.accessOrder = accessOrder;**

}

默认**accessOrder为false**：

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

super(initialCapacity, loadFactor);

accessOrder = false;

}

默认**loadFactor为0.75F**

public LinkedHashMap(int initialCapacity) {

super(initialCapacity);

accessOrder = false;

}

**默认初始容量为16**

public **LinkedHashMap**() {

super();

accessOrder = false;

}

### LinkedHashMap实现插入顺序的原理

/\*\*

\* HashMap.Node subclass for normal LinkedHashMap entries.

\*/

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);

}

}

/\*\*

\* The head (eldest) of the doubly linked list.

\*/

**transient LinkedHashMap.Entry<K,V> head;**

/\*\*

\* The tail (youngest) of the doubly linked **list**.

\*/

**transient LinkedHashMap.Entry<K,V> tail;**

### 插入尾节点

// link at the end of list

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;

}

}

//尾插法，将新Node置于尾部，即tail

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

LinkedHashMap.Entry<K,V> p =

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

**linkNodeLast(p);**

return p;

}

# HashMap与LinkedHashMap的区别

HashMap是无序的；LinkedHashMap是有序的，可以实现两种顺序，一是插入顺序，一是访问顺序，通过accessOrder指定。

//测试

@Test

public void test(){

//HashMap是无序的

HashMap<Integer,Integer> hashMap = new HashMap<Integer, Integer>();

hashMap.put(122,1);

hashMap.put(333,9);

hashMap.put(21,6);

hashMap.put(52,10);

for (Integer key : hashMap.keySet()){

System.out.println("key = "+key+"; value = " + hashMap.get(key));

}

//LinkedHashMap是有序的

LinkedHashMap<Integer,Integer> linkedHashMap = new LinkedHashMap<Integer, Integer>();

linkedHashMap.put(122,1);

linkedHashMap.put(333,9);

linkedHashMap.put(21,6);

linkedHashMap.put(52,10);

for (Integer key : linkedHashMap.keySet()){

System.out.println("key = "+key+"; value = " + linkedHashMap.get(key));

}

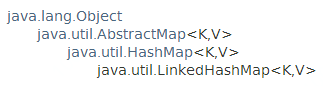
}

# LinkedHashMap基本介绍

## 继承关系介绍

public class **LinkedHashMap<K,V>**

extends **HashMap**<K,V> implements Map<K,V>



## 功能介绍

A special constructor is provided to create **a linked hash map** whose order of iteration is the order in which its entries were **last accessed**, **from least-recently accessed to most-recently (access-order)**. This kind of map is well-suited to building **LRU caches**. Invoking the **put, putIfAbsent, get, getOrDefault, compute, computeIfAbsent, computeIfPresent, or merge methods** results in an access to the corresponding entry (assuming it exists after the invocation completes). The **replace** methods only result in an access of the entry if the value is replaced. The **putAll** method generates one entry access for each mapping in the specified map, in the order that key-value mappings are provided by the specified map's entry set iterator. No other methods generate entry accesses. In particular, operations on collection-views do not affect the order of iteration of the backing map.

## 构造方法

三个配置参数：

**int initialCapacity, float loadFactor, boolean accessOrder**

**accessOrder 指定存储模式**：添加顺序存储和访问顺序顺序，默认情况下是添加顺序存储，当指定accessOrder为true时，采用**访问顺序存储**。

LinkedHashMap()

Constructs **an empty insertion-ordered LinkedHashMap instance** with the default initial capacity (16) and load factor (0.75).

LinkedHashMap(int initialCapacity)

Constructs **an empty insertion-ordered LinkedHashMap instance** with the specified initial capacity and a default load factor (0.75).

LinkedHashMap(int initialCapacity, float loadFactor)

Constructs **an empty insertion-ordered LinkedHashMap instance** with the specified initial capacity and load factor.

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

Constructs an empty LinkedHashMap instance with the specified initial capacity, load factor and **ordering mode**.

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

Constructs an **insertion-ordered LinkedHashMap** instance with the same mappings as the specified map.

## 方法介绍

### get方法

### V get(Object key)

Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

### V getOrDefault(Object key, V defaultValue)

Returns the value to which the specified key is mapped, or **defaultValue** if this map contains no mapping for the key.

### void clear()

Removes all of the mappings from this map.

### boolean containsValue(Object value)

Returns true if this map maps one or more keys to the specified value.

### Set<Map.Entry<K,V>> entrySet()

Returns a Set view of the mappings contained in this map.

### void forEach(BiConsumer<? super K,? super V> action)

Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.

### Set<K> keySet()

Returns a Set view of the keys contained in this map.

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

Returns true if this map should remove its eldest entry.

### void replaceAll(BiFunction<? super K,? super V,? extends V> function)

Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

### Collection<V> values()

Returns a Collection view of the values contained in this map.