利用HashMap与双向链表实现LRU\_(重要)\_Leetcode\_146\_LRUCache\_LRU缓存\_Hard

# Leetcode\_146\_LRUCache\_LRU缓存\_Hard

## 问题描述

\* 题目难度：Hard

\* <https://leetcode.com/problems/lru-cache/description/>

\* Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and put.

\* get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

\* put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

\* Follow up:

\* Could you do both operations in O(1) time complexity?

\* Example:

\* LRUCache cache = new LRUCache( 2 );//capacity

\* cache.put(1,1);

\* cache.put(2,2);

\* cache.get(1); // returns 1

\* cache.put(3,3); // evicts key 2

\* cache.get(2); // returns -1 (not found)

\* cache.put(4,4); // evicts key 1

\* cache.get(1); // returns -1 (not found)

\* cache.get(3); // returns 3

\* cache.get(4); // returns 4

## 思路分析

\* 思路分析：利用HashMap+双端链表实现；可以利用HashMap+LinkedList实现。

\* HashMap中存放key和链表节点的引用，双端链表将节点连在一起。

\* 对于双端链表需要定义两个端点节点head和tail，因为在容量满的时候，

\* 需要直接删除最后一节点，在get到或put到某节点，需要把该节点调整到

\* 最前端，必须定义head和tail，便于操作。

\* 构造方法：设定容量，HashMap、head、tail。

\* 对于get操作：没有找到该key，直接返回-1，若存在key，

\*需要将该节点移动到双端队列的头节点：包括两步：删除当前节点，并在头节点插入。

\* 对于put操作：若存在，则修改，并删除当前节点移动到头节点；

\* 若不存在，则直接添加到头节点。

## Java代码

class LRUCache {

//双向链表数据结构

class DListNode {

int key;

int val;

DListNode prev;

DListNode next;

DListNode(int key, int val, DListNode prev, DListNode next) {

this.key = key;

this.val = val;

this.prev = prev;

this.next = next;

}

}

int capacity;//缓存队列容量

DListNode head;

DListNode tail;

HashMap<Integer,DListNode> hashMap;

//构造方法

public **LRUCache**(int capacity) {

this.capacity = capacity;

this.hashMap = new HashMap<Integer, DListNode>();

//初始化链表

head = new DListNode(0, 0, null, null);

tail = new DListNode(0, 0, null, null);

head.next = tail;

tail.prev = head;

}

//get方法

public int **get**(int key) {

if (hashMap.get(key) == null) return -1;//若不存在，返回-1

DListNode node = hashMap.get(key);

**deleteCurrentNode(node);//删除当前节点**

**moveToHead(node);//移动到头节点**

return node.val;

}

//put方法

public void **put**(int key, int value) {

if (hashMap.size() == this.capacity && hashMap.get(key) == null)//达到容量了

deleteTailNode();

DListNode node = hashMap.get(key);//从hashMap中获取节点引用

if (node != null){//若已经存在，更新并删除当前节点

node.val = value;

//删除当前节点

deleteCurrentNode(node);

}else{//创建新的节点

node = new DListNode(key, value, null, null);

hashMap.put(key,node);

}

moveToHead(node);//插入到头部

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*辅助方法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//删除当前节点

public void **deleteCurrentNode**(DListNode node){

//删除当前节点

node.prev.next = node.next;

node.next.prev = node.prev;

}

//删除尾节点

public void deleteTailNode(){

//删除尾节点

DListNode delete = tail.prev;

hashMap.remove(delete.key);//从hashMap中删除该节点

delete.prev.next = tail;

tail.prev = delete.prev;

}

//调整到头部

public void moveToHead(DListNode node){

head.next.prev = node;

node.next = head.next;

head.next = node;

node.prev = head;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//测试使用打印方法

public void **printLinkedList**() {

DListNode node = head;

StringBuilder sb = new StringBuilder();

while(node != null){

sb.append("<").append(node.key).append(",").append(node.val).append(">==");

node = node.next;

}

if(sb.length()>1) sb.delete(sb.length()-2,sb.length());

System.out.println("sb.toString() = " + sb.toString());

}

}

# 尾插法与头插法

LinkedHashMap中采用的是尾插法。

上述Leetcode的题目采用的是头插法。

# 模拟LinkedHash实现

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

用**LinkedHashMap**里面搭好的LRU架子来实现LRU的。现在我们脱离LinkedHashMap这个容器，手动去维护链表中元素的关系，也就是**仿照LinkedHashMap里面的LRU实现写出属于自己的afterNodeRemoval()、afterNodeInsertion()、afterNodeAccess()方法**。其实也是照着葫芦画瓢，只不过这一次难度升了几颗星。

话外音：**HashMap的查询、插入、修改、删除平均时间复杂度都是O(1)**。

**最坏的情况是所有的key都散列到一个Entry中，时间复杂度会退化成O(N)**。这就是为什么Java8的HashMap引入了红黑树的原因。当Entry中的链表长度超过8，链表会进化成红黑树。**红黑树是一个自平衡二叉查找树，它的查询/插入/修改/删除的平均时间复杂度为O(log(N))**。

## 尾插法

### 首先定义双向链表数据结构

static final class Entry<K, V> {

private Entry before;

private Entry after;

private K key;

private V value;

}

### 代码

首先我们采用的是**尾插法**，也就是新插入的元素或者命中的元素往尾部移动，头部的元素即是最近最少使用。

public class MyLru01<K, V> {

private int maxSize;

private Map<K, Entry<K, V>> map;

private Entry head;

private Entry tail;

public MyLru01(int maxSize) {

this.maxSize = maxSize;

map = new HashMap<>();

}

public void put(K key, V value) {

Entry<K, V> entry = new Entry<>();

entry.key = key;

entry.value = value;

afterEntryInsertion(entry);

map.put(key, entry);

if (map.size() > maxSize) {

map.remove(head.key);

afterEntryRemoval(head);

}

}

private void afterEntryInsertion(Entry<K, V> entry) {

if (entry != null) {

if (head == null) {

head = entry;

tail = head;

return;

}

if (tail != entry) {

Entry<K, V> pred = tail;

entry.before = pred;

tail = entry;

pred.after = entry;

}

}

}

private void afterEntryAccess(Entry<K, V> entry) {

Entry<K, V> last;

if ((last = tail) != entry) {

Entry<K, V> p = entry, b = p.before, a = p .after;

p.before = p.after = null;

if (b == null) {

head = a;

} else {

b.after = a;

}

if (a == null) {

last = b;

} else {

a.before = b;

}

if (last == null) {

head = p;

} else {

p.before = last;

last.after = p;

}

tail = p;

}

}

private Entry<K, V> getEntry(K key) {

return map.get(key);

}

public V get(K key) {

Entry<K, V> entry = this.getEntry(key);

if (entry == null) {

return null;

}

afterEntryAccess(entry);

return entry.value;

}

public void remove(K key) {

Entry<K, V> entry = this.getEntry(key);

afterEntryRemoval(entry);

}

private void afterEntryRemoval(Entry<K, V> entry) {

if (entry != null) {

Entry<K, V> p = entry, 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;

}

}

}

@Override

public String toString() {

StringBuffer sb = new StringBuffer();

Entry<K, V> entry = head;

while (entry != null) {

sb.append(String.format("%s:%s", entry.key, entry.value));

sb.append(" ");

entry = entry.after;

}

return sb.toString();

}

static final class Entry<K, V> {

private Entry before;

private Entry after;

private K key;

private V value;

}

public static void main(String[] args) {

MyLru01<String, String> map = new MyLru01<>(5);

map.put("1", "1");

map.put("2", "2");

map.put("3", "3");

map.put("4", "4");

map.put("5", "5");

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

map.put("6", "6");

map.get("2");

map.put("7", "7");

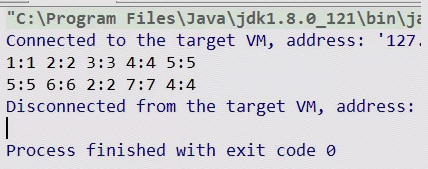
map.get("4");

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

}

}

2.运行结果也是5,6,2,7,4,与之前用LinkedHashMap实现的LRU运行结果一致。后面会分析写代码的思路。



### 解析

把key，value包装成Entry节点。调用afterEntryInsertion(entry)方法，把Entry节点移动到双向链表尾部。然后将key,Entry放入到HashMap中。如果map中元素的数量大于maxSize，则删除双向链表中的头结点(头结点所在的元素就是最近最少使用的元素)。首先在map中删除head.key对应着的元素，然后调用 afterEntryRemoval(head)，在双向链表中删除头节点。

public void put(K key, V value) {

Entry<K, V> entry = new Entry<>();

entry.key = key;

entry.value = value;

afterEntryInsertion(entry);

map.put(key, entry);

if (map.size() > maxSize) {

map.remove(head.key);

afterEntryRemoval(head);

}

}

### 双向链表为空

如果双向链表head节点为空的话，证明双向链表为空。**那么我们把新插入的元素置为head节点和tail节点**。否则我们把插入当前节点至尾部。这里是怎么插入呢？tail节点之前是尾部节点，现在突然要插入一个节点(entry节点)。那么tail节点再也不能占据尾部的位置，我们把置它为pre节点。pre节点也就是新的tail节点(也就是entry节点)的前一个节点。entry的先驱节点指向pre，pre节点的后继节点指向entry，这样就完成了尾插入。

private void afterEntryInsertion(Entry<K, V> entry) {

if (entry != null) {

if (head == null) {

head = entry;

tail = head;

return;

}

if (tail != entry) {

Entry<K, V> pred = tail;

entry.before = pred;

tail = entry;

pred.after = entry;

}

}

}

### 删除一个节点

我们是怎么在双向链表中删除一个节点呢？现在要删除的节点是entry节点。我们首先获取它的先驱节点b和后继节点a。如果b等于null，那么删除entry节点后，head节点应该为a。如果b不等于null，b的后继节点应该指向a。同样如果a等于null，那么删除entry节点后，tail节点应该为b。如果a不等于null，a的先驱节点应该指向b。这样就完成删除操作，如果还没明白的话，自己拿个笔画张图就差不多了。

public void afterEntryRemoval(Entry<K, V> entry) {

if (entry != null) {

Entry<K, V> p = entry, 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;

}

}

}

### 移动至尾部

我们通过get()方法命中了entry节点。那么我们怎么把entry节点移动至双向链表中的尾部呢？如果当前节点已位于尾部，那么我们什么也不做。如果当前节点不在尾部，和上面操作一样首先获取它的先驱节点b和后继节点a。然后把先驱节点和后继节点都置为null，方便后续操作。

如果b节点等于null，那么移动entry节点至尾部后，head节点应该为a节点。

如果b节点不等于null，那么b的后继节点应该指向a。

如果a节点等于null，那么新的尾部节点的前一个节点应该为b。

如果a节点不等于null，那么a的先驱节点应该指向b。

如果last节点(也就是新尾部节点的前一个节点)等于null的话，说明head节点应该为p节点。

如果last节点不等于null的话，我们把p的先驱节点指向last，last的后继节点指向p。最后新的尾部节点就是p。

过程有点绕，如果不明白的话，可以动手画图。

private void afterEntryAccess(Entry<K, V> entry) {

Entry<K, V> last;

if ((last = tail) != entry) {

Entry<K, V> p = entry, b = p.before, a = p .after;

p.before = p.after = null;

if (b == null) {

head = a;

} else {

b.after = a;

}

if (a == null) {

last = b;

} else {

a.before = b;

}

if (last == null) {

head = p;

} else {

p.before = last;

last.after = p;

}

tail = p;

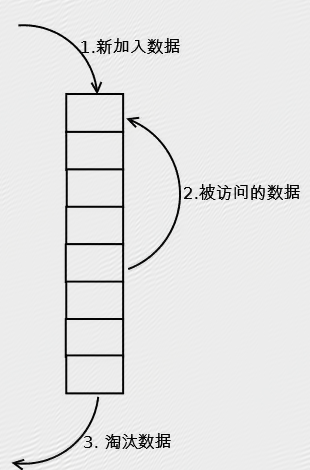
}

}

## 头插法

### 示意图

头插法其实和尾插法大同小异，**区别就是新插入的节点或者是命中的节点都移动至双向链表的头部，那么双向链表的尾部节点中所在的元素就是最近最少使用的元素**。



头插法的代码实现和尾插法基本一致，只是afterEntryInsertion()和afterEntryAccess()方法有所改动。改动的地方其实可以用上面的文字概括了！

再来说说下面例子中元素位置变化的过程吧

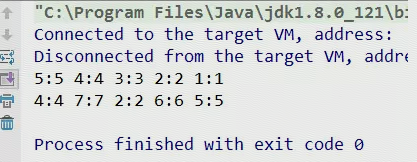
1.因为头插入法，5个元素插入完毕后。顺序应该是5,4,3,2,1

2.执行map.put("6", "6")后，把元素6插入到头部，并删除掉尾部元素1，顺序是6,5,4,3,2。

3.执行map.get("2")后，将元素2移动到头部，顺序是2,6,5,4,3

4.执行map.put("7", "7")后，把元素7插入到头部，并删除掉尾部元素,3，顺序是7,2,6,5,4

5.执行map.get("4")后，把元素4移动到头部，最后的顺序是4,7,2,6,5



### 代码

/\*\*

\* @author cmazxiaoma

\* @version V1.0

\* @Description: TODO

\* @date 2018/9/3 9:19

\*/

public class MyLru02<K, V> {

private int maxSize;

private Map<K, Entry<K, V>> map;

private Entry<K, V> head;

private Entry<K, V> tail;

public MyLru02(int maxSize) {

this.maxSize = maxSize;

map = new HashMap<>();

}

public void put(K key, V value) {

Entry<K, V> entry = new Entry<>();

entry.key = key;

entry.value = value;

afterEntryInsertion(entry);

map.put(key, entry);

if (map.size() > maxSize) {

map.remove(tail.key);

afterEntryRemoval(tail);

}

}

public void afterEntryInsertion(Entry<K, V> entry) {

if (entry != null) {

if (head == null) {

head = entry;

tail = head;

return;

}

// if entry is not head

if (head != entry) {

entry.after = head;

entry.before = null;

head.before = entry;

head = entry;

}

}

}

public void afterEntryRemoval(Entry<K, V> entry) {

if (entry != null) {

Entry<K, V> p = entry, 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;

}

}

}

public void afterEntryAccess(Entry<K, V> entry) {

Entry<K, V> first;

if ((first = head) != entry) {

Entry<K, V> p = entry, b = p.before, a = p.after;

p.before = p.after = null;

if (b == null) {

first = a;

} else {

b.after = a;

}

if (a == null) {

tail = b;

} else {

a.before = b;

}

if (first == null) {

tail = p;

} else {

p.after = first;

first.before = p;

}

head = p;

}

}

public void remove(K key) {

Entry<K, V> entry = this.getEntry(key);

afterEntryRemoval(entry);

}

public V get(K key) {

Entry<K, V> entry = this.getEntry(key);

if (entry == null) {

return null;

}

afterEntryAccess(entry);

return entry.value;

}

private Entry<K, V> getEntry(K key) {

Entry<K, V> entry = map.get(key);

if (entry == null) {

return null;

}

return entry;

}

@Override

public String toString() {

Entry<K, V> p = head;

StringBuffer sb = new StringBuffer();

while(p != null) {

sb.append(String.format("%s:%s", p.key, p.value));

sb.append(" ");

p = p.after;

}

return sb.toString();

}

static final class Entry<K, V> {

private Entry<K, V> before;

private Entry<K, V> after;

private K key;

private V value;

}

public static void main(String[] args) {

MyLru02<String, String> map = new MyLru02<>(5);

map.put("1", "1");

map.put("2", "2");

map.put("3", "3");

map.put("4", "4");

map.put("5", "5");

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

map.put("6", "6");

map.get("2");

map.put("7", "7");

map.get("4");

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

}

}