LRU原理和Redis实现——一个今日头条的面试题

文西
字节跳动 架构师

745 人赞同了该文章

很久前参加过今日头条的面试，遇到一个题，目前半部分是如何实现 LRU，后半部分是 Redis 中如何实现 LRU。

我的第一反应是操作系统课程里学过，应该是内存不够的场景下，淘汰旧内容的策略。LRU ... Least Recent Used，淘汰掉最不经常使用的。可以稍微多补充两句，因为计算机体系结构中，最大的最可靠的存储是硬盘，它容量很大，并且内容可以固化，但是访问速度很慢，所以需要把使用的内容载入内存中；内存速度很快，但是容量有限，并且断电后内容会丢失，并且为了进一步提升性能，还有CPU内部的 L1 Cache，L2 Cache等概念。因为速度越快的地方，它的单位成本越高，容量越小，新的内容不断被载入，旧的内容肯定要被淘汰，所以就有这样的使用背景。

LRU原理

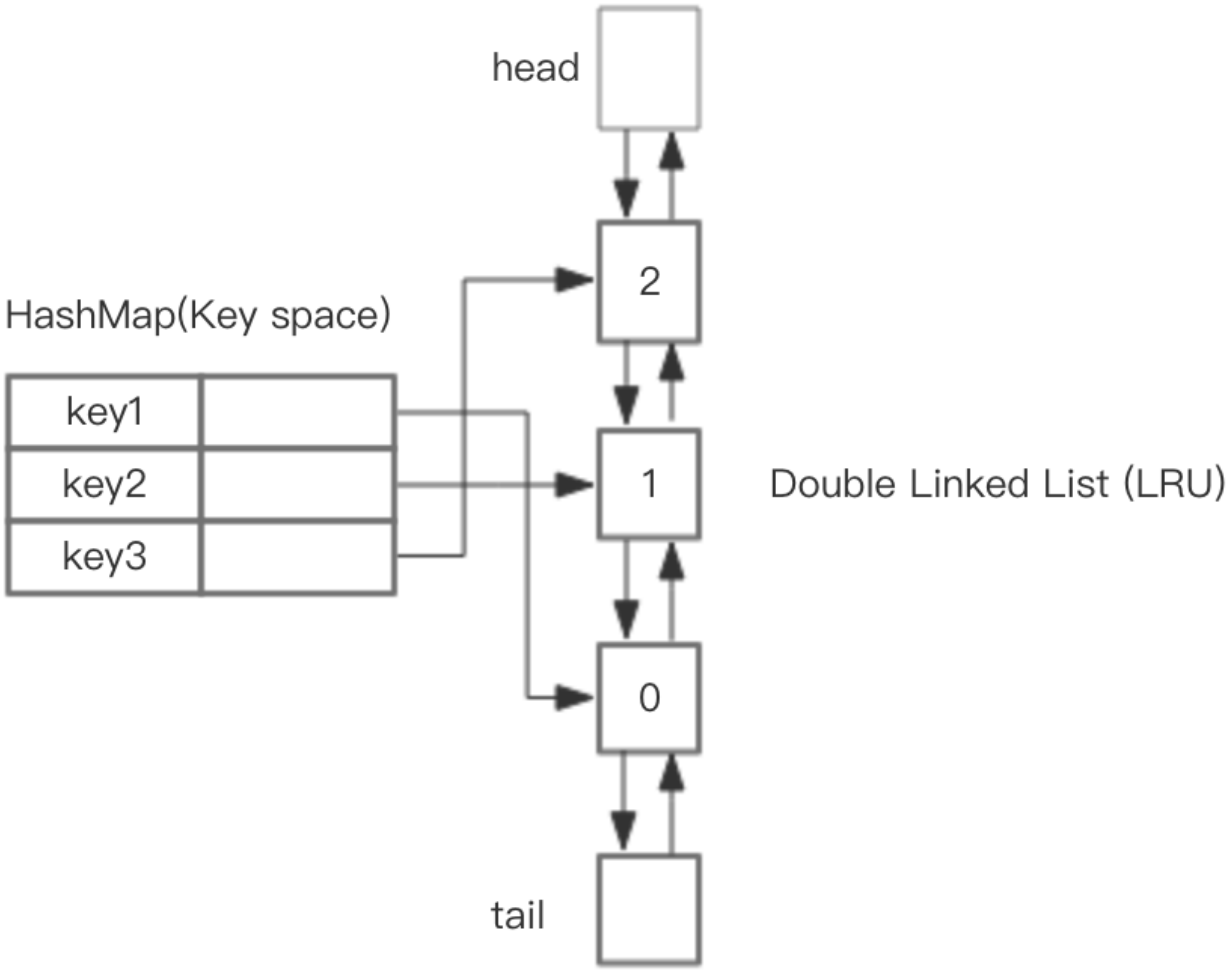
在一般标准的操作系统教材里，会用下面的方式来演示 LRU 原理，假设内存只能容纳3个页大小，按照 7 0 1 2 0 3 0 4 的次序访问页。假设内存按照栈的方式来描述访问时间，在上面的，是最近访问的，在下面的是，最远时间访问的，LRU就是这样工作的。

但是如果让我们自己设计一个基于 LRU 的缓存，这样设计可能问题很多，这段内存按照访问时间进行了排序，会有大量的内存拷贝操作，所以性能肯定是不能接受的。

那么如何设计一个LRU缓存，使得放入和移除都是 O(1) 的，我们需要把访问次序维护起来，但是不能通过内存中的真实排序来反应，有一种方案就是使用双向链表。

基于 HashMap 和 双向链表实现 LRU 的

整体的设计思路是，可以使用 HashMap 存储 key，这样可以做到 save 和 get key的时间都是 O(1)，而 HashMap 的 Value 指向双向链表实现的 LRU 的 Node 节点，如图所示。



LRU 存储是基于双向链表实现的，下面的图演示了它的原理。其中 head 代表双向链表的表头，tail 代表尾部。首先预先设置 LRU 的容量，如果存储满了，可以通过 O(1) 的时间淘汰掉双向链表的尾部，每次新增和访问数据，都可以通过 O(1)的效率把新的节点增加到对头，或者把已经存在的节点移动到队头。

下面展示了，预设大小是 3 的，LRU存储的在存储和访问过程中的变化。为了简化图复杂度，图中没有展示 HashMap部分的变化，仅仅演示了上图 LRU 双向链表的变化。我们对这个LRU缓存的操作序列如下：

save("key1", 7)

save("key2", 0)

save("key3", 1)

save("key4", 2)

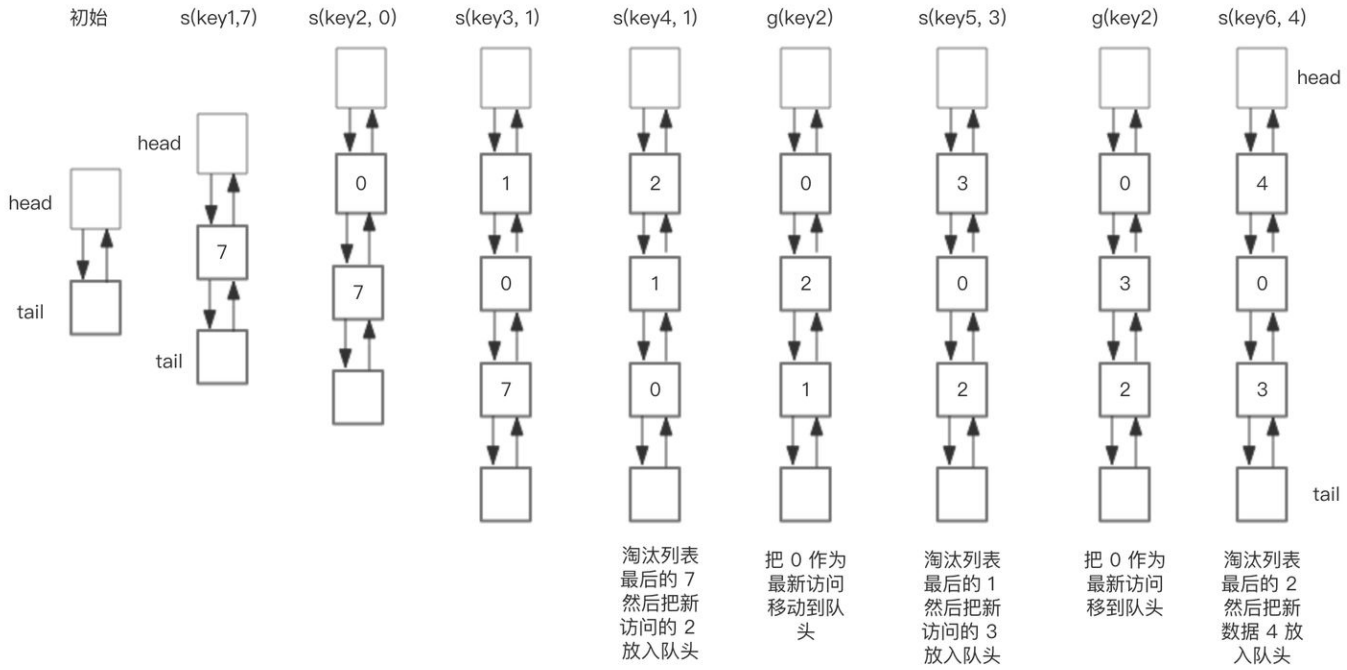
get("key2")

save("key5", 3)

get("key2")

save("key6", 4)

相应的 LRU 双向链表部分变化如下：



s = save, g = get

总结一下核心操作的步骤:

1. save(key, value)，首先在 HashMap 找到 Key 对应的节点，如果节点存在，更新节点的值，并把这个节点移动队头。如果不存在，需要构造新的节点，并且尝试把节点塞到队头，如果LRU空间不足，则通过 tail 淘汰掉队尾的节点，同时在 HashMap 中移除 Key。
2. get(key)，通过 HashMap 找到 LRU 链表节点，因为根据LRU 原理，这个节点是最新访问的，所以要把节点插入到队头，然后返回缓存的值。

完整基于 Java 的代码参考如下

```
class DLinkedNode {
        String key;
        int value;
        DLinkedNode pre;
        DLinkedNode post;
}
```

LRU Cache

```
public class LRUCache {

    private Hashtable<Integer, DLinkedNode>
            cache = new Hashtable<Integer, DLinkedNode>();
    private int count;
    private int capacity;
    private DLinkedNode head, tail;

    public LRUCache(int capacity) {
        this.count = 0;
        this.capacity = capacity;

        head = new DLinkedNode();
        head.pre = null;

        tail = new DLinkedNode();
        tail.post = null;

        head.post = tail;
        tail.pre = head;
    }

    public int get(String key) {
```

```
    DLinkedNode node = cache.get(key);
    if(node == null){
        return -1; // should raise exception here.
    }

    // move the accessed node to the head;
    this.moveToHead(node);

    return node.value;
}


public void set(String key, int value) {
    DLinkedNode node = cache.get(key);

    if(node == null){

        DLinkedNode newNode = new DLinkedNode();
        newNode.key = key;
        newNode.value = value;

        this.cache.put(key, newNode);
        this.addNode(newNode);

        ++count;

        if(count > capacity){
            // pop the tail
            DLinkedNode tail = this.popTail();
            this.cache.remove(tail.key);
            --count;
        }
    }else{
        // update the value.
        node.value = value;
        this.moveToHead(node);
    }
}
/**
 * Always add the new node right after head;
 */
```

```
    private void addNode(DLinkedNode node){
        node.pre = head;
        node.post = head.post;

        head.post.pre = node;
        head.post = node;
    }

    /**
     * Remove an existing node from the linked list.
     */
    private void removeNode(DLinkedNode node){
        DLinkedNode pre = node.pre;
        DLinkedNode post = node.post;

        pre.post = post;
        post.pre = pre;
    }

    /**
     * Move certain node in between to the head.
     */
    private void moveToHead(DLinkedNode node){
        this.removeNode(node);
        this.addNode(node);
    }

    // pop the current tail.
    private DLinkedNode popTail(){
        DLinkedNode res = tail.pre;
        this.removeNode(res);
        return res;
    }
}
```