如何实现 LRU 缓存机制

原创 labuladong labuladong 2019-07-06

收录于话题
#手撕力扣高频面试题

59个

预计阅读时间： 8 分钟

一、什么是 LRU 算法

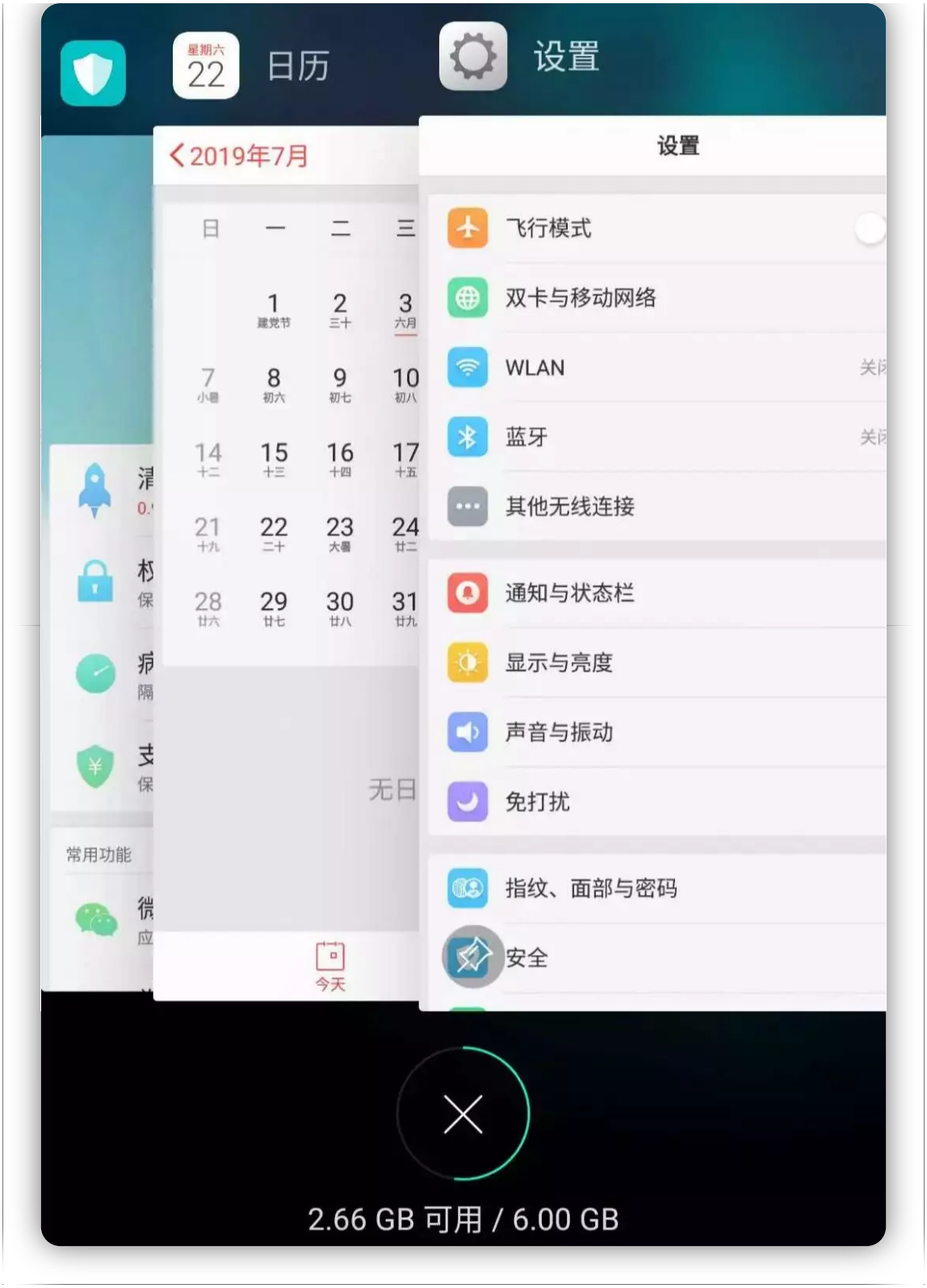
LRU 就是一种缓存淘汰策略。

计算机的缓存容量有限，如果缓存满了就要删除一些内容，给新内容腾位置。但问题是，删除哪些内容呢？我们肯定希望删掉那些没什么用的缓存，而把有用的数据继续留在缓存里，方便之后继续使用。那么，什么样的数据，我们判定为「有用的」的数据呢？

LRU 缓存淘汰算法就是一种常用策略。LRU 的全称是 Least Recently Used，也就是说我们认为最近使用过的数据应该是是「有用的」，很久都没用过的数据应该是无用的，缓存满了就优先删那些很久没用过的数据。

举个简单的例子，安卓手机都可以把软件放到后台运行，比如我先后打开了「设置」「手机管家」「日历」，那么现在他们在后台排列的顺序是这样的：

但是这时候如果我访问了一下「设置」界面，那么「设置」就会被提前到第一个，变成这样：



假设我的手机只允许我同时开 3 个应用程序，现在已经满了。那么如果我新开了一个应用「时钟」，就必须关闭一个应用为「时钟」腾出一个位置，关那个呢？

按照 LRU 的策略，就关最底下的「手机管家」，因为那是最久未使用的，然后把新开的应用放到最上面：

现在你应该理解 LRU（Least Recently Used）策略了。当然还有其他缓存淘汰策略，比如不要按访问的时序来淘汰，而是按访问频率（LFU 策略）来淘汰等等，各有应用场景。本文讲解 LRU 算法策略。

二、LRU 算法描述

LeetCode 上有一道 LRU 算法设计的题目。让你设计一种数据结构，首先构造函数接收一个 capacity 参数作为缓存的最大容量，然后实现两个 API：

一个是 put(key, val) 方法插入新的或更新已有键值对，如果缓存已满的话，要删除那个最久没用过的键值对以腾出位置插入。

另一个是 get(key) 方法获取 key 对应的 val，如果 key 不存在则返回 -1。

注意哦，get 和 put 方法必须都是 $O(1)$ 的时间复杂度，我们举个具体例子来看看 LRU 算法怎么工作。

```
/* 缓存容量为 2 */
LRUCache cache = new LRUCache(2);
// 你可以把 cache 理解成一个队列
// 假设左边是队头，右边是队尾
// 最近使用的排在队头，久未使用的排在队尾
// 圆括号表示键值对 (key, val)

cache.put(1, 1);
// cache = [(1, 1)]
cache.put(2, 2);
// cache = [(2, 2), (1, 1)]
cache.get(1);       // 返回 1
// cache = [(1, 1), (2, 2)]
// 解释：因为最近访问了键 1，所以提前至队头
// 返回键 1 对应的值 1
cache.put(3, 3);
// cache = [(3, 3), (1, 1)]
// 解释：缓存容量已满，需要删除内容空出位置
// 优先删除久未使用的数据，也就是队尾的数据
// 然后把新的数据插入队头
cache.get(2);       // 返回 -1 (未找到)
// cache = [(3, 3), (1, 1)]
// 解释：cache 中不存在键为 2 的数据
cache.put(1, 4);
// cache = [(1, 4), (3, 3)]
// 解释：键 1 已存在，把原始值 1 覆盖为 4
// 不要忘了也要将键值对提前到队头
```

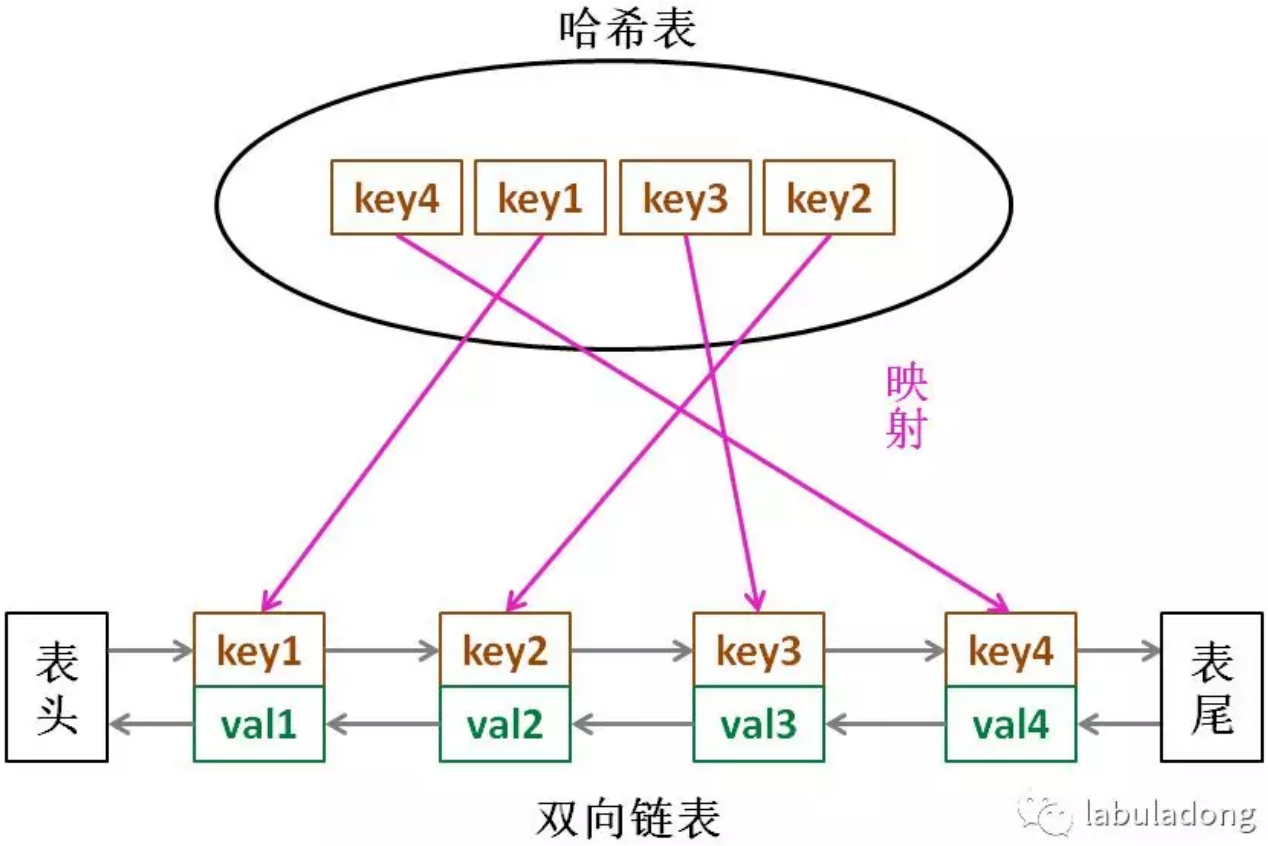
三、LRU 算法设计

分析上面的操作过程，要让 put 和 get 方法的时间复杂度为 $O(1)$，我们可以总结出 cache 这个数据结构必要的条件：查找快，插入快，删除快，有顺序之分。

因为显然 cache 必须有顺序之分，以区分最近使用的和久未使用的数据；而且我们要在 cache 中查找键是否已存在；如果容量满了要删除最后一个数据；每次访问还要把数据插入到队头。

那么，什么数据结构同时符合上述条件呢？哈希表查找快，但是数据无固定顺序；链表有顺序之分，插入删除快，但是查找慢。所以结合一下，形成一种新的数据结构：**哈希链表**。

LRU 缓存算法的核心数据结构就是哈希链表，双向链表和哈希表的结合体。这个数据结构长这样：



思想很简单，就是借助哈希表赋予了链表快速查找的特性嘛：可以快速查找某个 key 是否存在缓存（链表）中，同时可以快速删除、添加节点。回想刚才的例子，这种数据结构是不是完美解决了 LRU 缓存的需求？

也许读者会问，为什么要是双向链表，单链表行不行？另外，既然哈希表中已经存了 key，为什么链表中还要存键值对呢，只存值不就行了？

想的时候都是问题，只有做的时候才有答案。这样设计的原因，必须等我们亲自实现 LRU 算法之后才能理解，所以我们开始看代码吧～

四、代码实现

很多编程语言都有内置的哈希链表或者类似 LRU 功能的库函数，但是为了帮大家理解算法的细节，我们用 Java 自己造轮子实现一遍 LRU 算法。

首先，我们把双链表的节点类写出来，为了简化，key 和 val 都认为是 int 类型：

```
class Node {
    public int key, val;
    public Node next, prev;
    public Node(int k, int v) {
        this.key = k;
        this.val = v;
    }
}
```

然后依靠我们的 Node 类型构建一个双链表，实现几个要用到的 API，这些操作的时间复杂度均为 $O(1)$：

```
class DoubleList {
    // 在链表头部添加节点 x
    public void addFirst(Node x);

    // 删除链表中的 x 节点（x 一定存在）
    public void remove(Node x);

    // 删除链表中最后一个节点，并返回该节点
    public Node removeLast();

    // 返回链表长度
    public int size();
}
```

PS：这就是普通双向链表的实现，为了让读者集中精力理解 LRU 算法的逻辑，就省略链表的具体代码。有疑问的朋友可以点击文末「阅读原文」查看完整代码。

到这里就能回答刚才“为什么必须要用双向链表”的问题了，因为我们需要删除操作。删除一个链表节点不光要得到该节点本身的指针，也需要操作其前驱节点的指针，而双向链表才能支持直接查找前驱，保证操作的时间复杂度 $O(1)$。

有了双向链表的实现，我们只需要在 LRU 算法中把它和哈希表结合起来即可。我们先把逻辑理清楚：

```
// key 映射到 Node(key, val)
HashMap<Integer, Node> map;
// Node(k1, v1) <-> Node(k2, v2)...
DoubleList cache;

int get(int key) {
    if (key 不存在) {
        return -1;
    } else {
        将数据 (key, val) 提到开头；
        return val;
    }
}

void put(int key, int val) {
    Node x = new Node(key, val);
    if (key 已存在) {
        把旧的数据删除；
        将新节点 x 插入到开头；
    } else {
        if (cache 已满) {
            删除链表的最后一个数据腾位置；
            删除 map 中映射到该数据的键；
        }
        将新节点 x 插入到开头；
        map 中新建 key 对新节点 x 的映射；
    }
}
```

如果能够看懂上述逻辑，翻译成代码就很容易理解了：

```
class LRUCache {
    // key -> Node(key, val)
    private HashMap<Integer, Node> map;
    // Node(k1, v1) <-> Node(k2, v2)...
    private DoubleList cache;
    // 最大容量
    private int cap;

    public LRUCache(int capacity) {
        this.cap = capacity;
        map = new HashMap<>();
        cache = new DoubleList();
    }

    public int get(int key) {
        if (!map.containsKey(key))
            return -1;
        int val = map.get(key).val;
        // 利用 put 方法把该数据提前
        put(key, val);
        return val;
    }

    public void put(int key, int val) {
        // 先把新节点 x 做出来
        Node x = new Node(key, val);

        if (map.containsKey(key)) {
            // 删除旧的节点，新的插到头部
            cache.remove(map.get(key));
            cache.addFirst(x);
            // 更新 map 中对应的数据
            map.put(key, x);
```

这里就能回答之前的问题“为什么要在链表中同时存储 key 和 val，而不是只存储 val”，注意这段代码：

```
if (cap == cache.size()) {
    // 删除链表最后一个数据
    Node last = cache.removeLast();
    map.remove(last.key);
}
```

当缓存容量已满，我们不仅仅要删除最后一个 Node 节点，还要把 map 中映射到该节点的 key 同时删除，而这个 key 只能由 Node 得到。如果 Node 结构中只存储 val，那么我们就无法得知 key 是什么，就无法删除 map 中的键，造成错误。

至此，你应该已经掌握 LRU 算法的思想和实现了，很容易犯错的一点是：处理链表节点的同时不要忘了更新哈希表中对节点的映射。

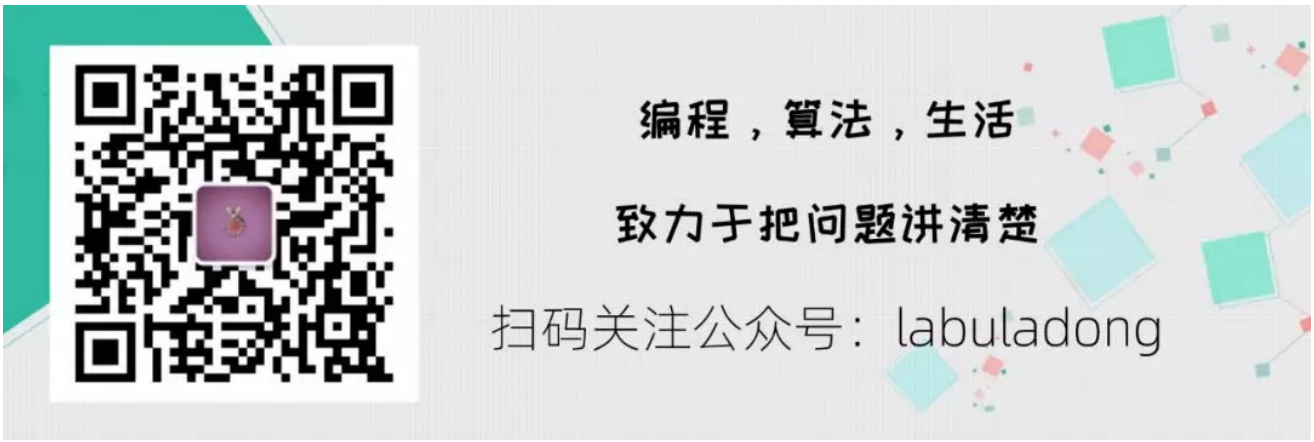
最后，如果读者觉得这类设计问题有意思的话，点个在看分个享鼓励一下我，以后多写类似的文章～

猜你喜欢

滑动窗口算法解决子串问题

单调队列解决滑动窗口问题

点击这里进入留言板



收录于话题 #手撕力扣高频面试题 59

上一篇
设计 Twitter：合并 k 个有序链表和面向对象设计

下一篇
三个反直觉的概率问题

阅读原文