题目地址(146. LRU 缓存机制)

https://leetcode-cn.com/problems/lru-cache/

标签

- 哈希表
- 链表

难度

- 困难

入选理由

1. 书写难度较大，当压轴题
2. 设计题是最考察数据结构知识的，希望通过这道题让大家感受链表

题目描述

```
运用你所掌握的数据结构，设计和实现一个  LRU（最近最少使用）缓存机制 。
实现 LRUCache 类：

LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在


进阶：你是否可以在 O(1) 时间复杂度内完成这两种操作?



示例：

输入

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]
解释
```

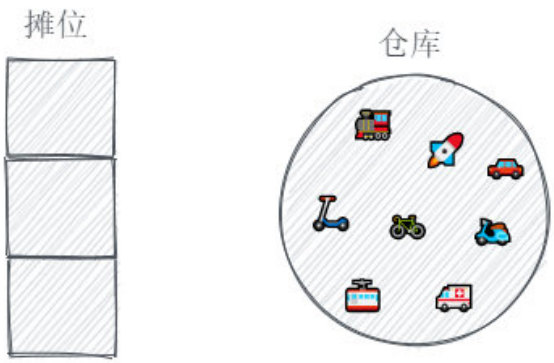
```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

哈希表+双向链表

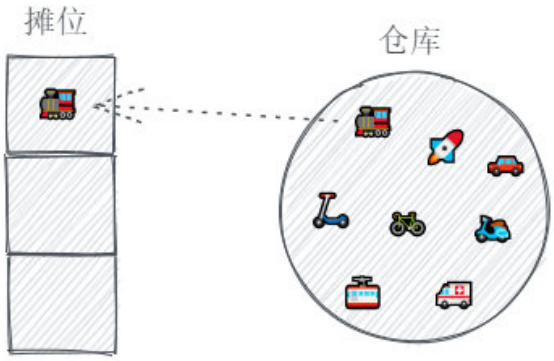
思路

首先简单介绍下什么是 LRU 缓存，熟悉的可以跳过。

假设我们有一个玩具摊位，用于向顾客展示小玩具。玩具很多，摊位大小有限，不能一次性展示所有玩具，于是大部分玩具放在了仓库里。

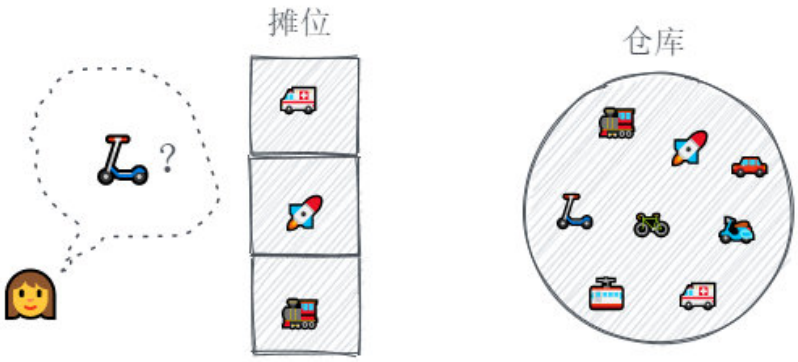


如果有顾客来咨询某个玩具，我们就去仓库把该玩具拿出来，摆在摊位上。



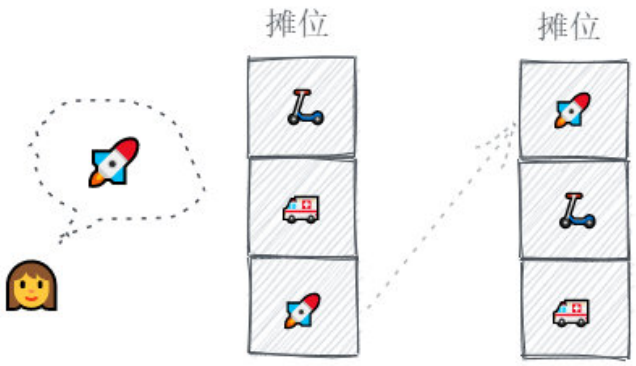
因为摊位最上面的位置最显眼，所以我们总是把最新拿出来的玩具放在那。

不过由于摊位大小有限，很快就摆满了，这时如果又有顾客想看新玩具。



我们只能把摊位最下面的玩具拿回仓库(因为最下面的位置相对没那么受欢迎)，然后其他玩具往下移，腾出最上面的位置来放新玩具。

如果顾客想看的玩具就摆在摊位上，我们就可以把这个玩具直接移到摊位最上面的位置，其他的玩具就要往下挪挪位置了。还记得我们的规则吧，最近有人询问的玩具要摆在最上面显眼的位置。



这就是对 LRU 缓存的一个简单解释。回到计算机问题上面来，玩具摊位代表的就是缓存空间，我们首先需要考虑的问题是使用哪种数据结构来表示玩具摊位，以达到题目要求的时间复杂度。

数组?

如果选择数组，因为玩具在摊位上的位置会挪来挪去，这个操作的时间复杂度是$O(N)$，不符合题意，pass。

链表?

- 如果选择链表，在给定节点后新增节点，或者移除给定节点的时间复杂度是 $O(1)$。但是，链表查找节点的时间复杂度是$O(N)$，同样不符合题意，不过还有办法补救。
- 在玩具摊位的例子中，我们是手动移动玩具，人类只需要看一眼就知道要找的玩具在哪个位置上，但计算机没那么聪明，因此还需要给它一个脑子(哈希表)来记录什么玩具在什么位置上，也就是要用一个哈希表来记录每个 key 对应的链表节点

引用。这样查找链表节点的时间复杂度就降到了 $O(1)$，不过代价是空间复杂度增加到了 $O(N)$。

- 另外，由于移除链表节点后还需要把该节点前后的两个节点连起来，因此我们需要的是双向链表而不是单向链表。

伪代码：

```
// put

if key 存在:
    更新节点值
    把节点移到链表头部

else:
    if 缓存满了:
        移除最后一个节点
        删除它在哈希表中的映射

    新建一个节点
    把节点加到链表头部
    在哈希表中增加映射


// get

if key 存在:
    返回节点值
    把节点移到链表头部
else:
    返回 -1
```

代码

代码支持：JS， Java，CPP

JavaScript Code

```
class DoubleLinkedListNode {
    constructor(key, value) {
        this.key = key
        this.value = value
        this.prev = null
        this.next = null
    }
}

class LRUCache {
    constructor(capacity) {
        this.capacity = capacity
        this.usedSpace = 0
```

```
    // Mappings of key->node.
    this.hashmap = {}
    this.dummyHead = new DoubleLinkedListNode(null, null)
    this.dummyTail = new DoubleLinkedListNode(null, null)
    this.dummyHead.next = this.dummyTail
    this.dummyTail.prev = this.dummyHead
}

_isFull() {
    return this.usedSpace === this.capacity
}

_removeNode(node) {
    node.prev.next = node.next
    node.next.prev = node.prev
    node.prev = null
    node.next = null
    return node
}

_addToHead(node) {
    const head = this.dummyHead.next
    node.next = head
    head.prev = node
    node.prev = this.dummyHead
    this.dummyHead.next = node
}

get(key) {
    if (key in this.hashmap) {
        const node = this.hashmap[key]
        this._addToHead(this._removeNode(node))
        return node.value
    }
    else {
        return -1
    }
}

put(key, value) {
    if (key in this.hashmap) {
        // If key exists, update the corresponding node and move it to the head.
        const node = this.hashmap[key]
        node.value = value
        this._addToHead(this._removeNode(node))
    }
    else {
    // If it's a new key.
        if (this._isFull()) {
            // If the cache is full, remove the tail node.
            const node = this.dummyTail.prev
            delete this.hashmap[node.key]
```

```
                this._removeNode(node)
                this.usedSpace--
            }
            // Create a new node and add it to the head.
            const node = new DoubleLinkedListNode(key, value)
            this.hashmap[key] = node
            this._addToHead(node)
            this.usedSpace++
        }
    }
}


/**
 * Your LRUCache object will be instantiated and called as such:
 * var obj = new LRUCache(capacity)
 * var param_1 = obj.get(key)
 * obj.put(key,value)
 */
```

Java Code：

```
class LRUCache {
    class DLinkedNode {
        int key, value;
        DLinkedNode prev, next;
        public DLinkedNode() {}
        public DLinkedNode(int _key, int _value) {
            key = _key;
            value = _value;
        }
    }

    private Map<Integer, DLinkedNode> cache = new HashMap<Integer, DLinkedNode>();
    private int size, cap;
    private DLinkedNode head, tail;

    public LRUCache(int capacity) {
        size = 0;
        cap = capacity;
        //add dummy head and dummyTail
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        DLinkedNode node = cache.get(key);
        if (node == null) return -1;
```

```
        //if key exist, move it to head by using its location store in Hashmap
        moveToHead(node);
        return node.value;
    }

    public void put(int key, int value) {
        DLinkedNode node = cache.get(key);
        if (node == null) {
            //made a newNode if it does not exist
            DLinkedNode newNode = new DLinkedNode(key, value);
            cache.put(key, newNode);
            addToHead(newNode);
            ++size;
            if (size > cap) {
                DLinkedNode removedTail = removeTail();
                cache.remove(removedTail.key);
                --size;
            }
        } else {
            node.value = value;
            moveToHead(node);
        }
    }

    private void addToHead(DLinkedNode node){
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(DLinkedNode node){
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void moveToHead(DLinkedNode node){
        removeNode(node);
        addToHead(node);
    }

    private DLinkedNode removeTail(){
        DLinkedNode res = tail.prev;
        removeNode(res);
        return res;
    }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

C++ Code

```
class DLinkedListNode {
public:
    int key;
    int value;
    DLinkedListNode *prev;
    DLinkedListNode *next;
    DLinkedListNode() : key(0), value(0), prev(NULL), next(NULL) {};
    DLinkedListNode(int k, int val) : key(k), value(val), prev(NULL), next(NULL) {};
};

class LRUCache {
public:
    LRUCache(int capacity) : capacity_(capacity) {
        // 创建两个 dummy 节点来简化操作，这样就不用特殊对待头尾节点了
        dummy_head_ = new DLinkedListNode();
        dummy_tail_ = new DLinkedListNode();
        dummy_head_->next = dummy_tail_;
        dummy_tail_->prev = dummy_head_;
    }

    int get(int key) {
        if (!key_exists_(key)) {
            return -1;
        }
        // 1. 通过哈希表找到 key 对应的节点
        // 2. 将节点移到链表头部
        // 3. 返回节点值
        DLinkedListNode *node = key_node_map_[key];
        move_to_head_(node);
        return node->value;
    }

    void put(int key, int value) {
        if (key_exists_(key)) {
            // key 存在的情况
            DLinkedListNode *node = key_node_map_[key];
            node->value = value;
            move_to_head_(node);
        } else {
            // key 不存在的情况：
            // 1. 如果缓存空间满了，先删除尾节点，再新建节点
            // 2. 否则直接新建节点
            if (is_full_()) {
                DLinkedListNode *tail = dummy_tail_->prev;
                remove_node_(tail);
                key_node_map_.erase(tail->key);
            }
```

```
            DLinkedListNode *new_node = new DLinkedListNode(key, value);
            add_to_head_(new_node);
            key_node_map_[key] = new_node;
        }
    }
private:
    unordered_map<int, DLinkedListNode*> key_node_map_;
    DLinkedListNode *dummy_head_;
    DLinkedListNode *dummy_tail_;
    int capacity_;

    void move_to_head_(DLinkedListNode *node) {
        remove_node_(node);
        add_to_head_(node);
    };

    void add_to_head_(DLinkedListNode *node) {
        DLinkedListNode *prev_head = dummy_head_->next;

        dummy_head_->next = node;
        node->prev = dummy_head_;

        node->next = prev_head;
        prev_head->prev = node;
    };

    void remove_node_(DLinkedListNode *node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
        node->prev = node->next = NULL;
    };

    bool key_exists_(int key) {
        return key_node_map_.count(key) > 0;
    };

    bool is_full_() {
        return key_node_map_.size() == capacity_;
    };
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

复杂度分析

- 时间复杂度：各种操作平均都是 $O(1)$。
- 空间复杂度：链表占用空间 $O(N)$，哈希表占用空间也是 $O(N)$，因此总的空间复杂度为 $O(N)$，其中 N 为容量大小，也就是题目中的 capacity。

上一页 下一页