H1 146 缓存机制

H2 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`。它应该支持以下操作： 获取数据`get` 和 写入数据 `put` 。

获取数据 `get(key)` - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。 写入数据 `put(key, value)` - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

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:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

H2 代码

```python
class Node:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None

class DLL:
    def __init__(self):
        self.head = Node(None, None)
        self.tail = Node(None, None)
        self.head.next = self.tail
        self.tail.prev = self.head

    def insert(self, node):
        node.prev, self.tail.prev.next = self.tail.prev, node
        node.next, self.tail.prev = self.tail, node
```

```
    def remove_at_head(self):
        node = self.head.next
        node.next.prev = self.head
        self.head.next = self.head.next.next
        key = node.key
        del node
        return key

    def update(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev
        self.insert(node)

class LRUCache:

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.queue = DLL()
        self.mapping = {}

    def get(self, key: int) -> int:
        if key not in self.mapping:
            return -1
        node = self.mapping[key]
        self.queue.update(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        if key in self.mapping:
            node = self.mapping[key]
            node.val = value
            self.queue.update(node)
            return

        node = Node(key, value)
        self.mapping[key] = node
        self.queue.insert(node)

        if self.capacity == 0:
            removed_key = self.queue.remove_at_head()
            del self.mapping[removed_key]
        else:
            self.capacity -= 1


# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
```

```
# obj.put(key,value)
```

成功　显示详情 ›

执行用时 : 172 ms, 在LRU Cache的Python3提交中击败了67.29% 的用户

内存消耗 : 22 MB, 在LRU Cache的Python3提交中击败了25.13% 的用户

进行下一个挑战：

LFU缓存　设计内存文件系统　迭代压缩字符串

炫耀一下:

提交时间	状态	执行用时	内存消耗	语言
几秒前	通过	172 ms	22 MB	python3