39. 模拟实现 LRU 缓存

1. 核心概念 (Core Concept)

LRU (Least Recently Used) 缓存是一种基于**“最近最少使用”**原则的缓存淘汰策略。当缓存容量达到上限，需要淘汰旧数据时，LRU 算法会优先移除那些最近最久未被访问的数据项。

2. 为什么需要它？ (The "Why")

1. **提高数据访问效率:** 将常用数据保存在快速访问的缓存中，避免频繁从慢速存储（如磁盘、网络）中读取，显著提升系统性能。
2. **控制内存使用:** 在有限的内存空间内，通过淘汰不常用的数据，确保缓存不会无限增长，避免内存溢出或过度占用。
3. **命中率优化:** LRU 策略基于局部性原理，认为最近访问的数据在未来很可能再次被访问，从而提高缓存命中率。

3. API 与用法 (API & Usage)

模拟实现 LRU 缓存通常需要支持以下两个核心操作：

- **`put(key, value)`**：将键值对添加到缓存中。如果 `key` 已存在，则更新其对应的值，并将其标记为“最新使用”。如果 `key` 不存在且缓存已满，则淘汰最近最少使用的数据项，然后添加新的键值对。
- **`get(key)`**：获取指定 `key` 对应的值。如果 `key` 存在，则返回对应值，并将其标记为“最新使用”；如果 `key` 不存在，则返回特殊值（如 -1 或 null）。

为了高效地实现这两个操作，通常会结合使用**哈希表（用于快速查找）**和**双向链表（用于维护访问顺序）**。哈希表的键是缓存的 key，值是指向双向链表中对应节点的指针。双向链表则按照访问的新旧顺序存储数据，链表头部是最近使用的，尾部是最近最少使用的。

以下是一个使用 JavaScript 模拟实现的经典 LRU 缓存结构示例：

```
class LRUCache {
    constructor(capacity) {
        this.capacity = capacity;
        this.cache = new Map(); // 哈希表：用于存储 key 到链表节点的映射
        // 双向链表，用于维护访问顺序
        this.head = new Node(); // 哨兵节点：链表头部，表示最新使用
        this.tail = new Node(); // 哨兵节点：链表尾部，表示最近最少使用
        this.head.next = this.tail;
        this.tail.prev = this.head;
    }

    // 双向链表节点结构
```

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

// 将节点移动到链表头部（表示最新使用）
_moveToHead(node) {
    this._removeNode(node);
    this._addNode(node);
}

// 在链表头部添加节点
_addNode(node) {
    node.prev = this.head;
    node.next = this.head.next;
    this.head.next.prev = node;
    this.head.next = node;
}

// 移除指定节点
_removeNode(node) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

// 移除链表尾部节点（最近最少使用）
_removeTail() {
    const nodeToRemove = this.tail.prev;
    this._removeNode(nodeToRemove);
    return nodeToRemove;
}

get(key) {
    if (this.cache.has(key)) {
        const node = this.cache.get(key);
        this._moveToHead(node); // 访问后移动到头部
        return node.value;
    }
    return -1; // 或 null，表示未找到
}

put(key, value) {
    if (this.cache.has(key)) {
        const node = this.cache.get(key);
        node.value = value; // 更新值
        this._moveToHead(node); // 更新后移动到头部
```

```
        } else {
            // 新建节点
            const newNode = new Node(key, value);
            this.cache.set(key, newNode);
            this._addNode(newNode); // 添加到链表头部

            // 检查容量，如果超出则淘汰尾部节点
            if (this.cache.size > this.capacity) {
                const tailNode = this._removeTail();
                this.cache.delete(tailNode.key); // 从哈希表中删除
            }
        }
    }
}

// 示例用法
const lruCache = new LRUCache(2);
lruCache.put(1, 1); // 缓存: {1=Node(1,1)}，链表: head <-> Node(1,1) <->
tail
lruCache.put(2, 2); // 缓存: {1=Node(1,1), 2=Node(2,2)}，链表: head <->
Node(2,2) <-> Node(1,1) <-> tail
console.log(lruCache.get(1)); // 输出 1，访问 1，1 移动到头部
// 缓存: {1=Node(1,1), 2=Node(2,2)}，链表: head <-> Node(1,1) <-> Node(2,2)
<-> tail
lruCache.put(3, 3); // 容量已满，淘汰最近最少使用的 2
// 缓存: {1=Node(1,1), 3=Node(3,3)}，链表: head <-> Node(3,3) <-> Node(1,1)
<-> tail
console.log(lruCache.get(2)); // 输出 -1，2 已被淘汰
lruCache.put(4, 4); // 容量已满，淘汰最近最少使用的 1
// 缓存: {4=Node(4,4), 3=Node(3,3)}，链表: head <-> Node(4,4) <-> Node(3,3)
<-> tail
console.log(lruCache.get(1)); // 输出 -1
console.log(lruCache.get(3)); // 输出 3，访问 3，3 移动到头部
// 缓存: {4=Node(4,4), 3=Node(3,3)}，链表: head <-> Node(3,3) <-> Node(4,4)
<-> tail
```

4. 关键注意事项 (Key Considerations)

1. **数据结构选择:** LRU 的高效实现依赖于哈希表 O(1) 的查找速度以及双向链表 O(1) 的节点插入和删除操作。单链表或仅使用数组是不行的。
2. **链表头尾:** 双向链表的头部通常表示最近使用的元素，尾部表示最近最少使用的元素（准备被淘汰）。哨兵节点（Dummy Head/Tail）可以简化边界情况的处理。
3. **哈希表存储链表节点引用:** 哈希表的值不是数据本身，而是指向双向链表中对应数据节点的引用。这样通过 key 找到节点后，可以直接操作链表来更新访问顺序。
4. **操作复杂度:** 采用哈希表和双向链表结合的实现，`get` 和 `put` 操作的时间复杂度均为 O(1)。

5. 参考资料 (References)

- LeetCode 146. LRU 缓存 - LeetCode 上的经典题目，提供多种语言的实现思路和解法。
- 如何实现一个 LRU 缓存 - 知乎 - 一篇关于 LRU 原理和实现的讲解，有助于理解其背后的思想。
- 各种高级编程语言（如 Java, Python, 等）的标准库中可能有 LinkedList 和 HashMap/Dict 的实现，可作为参考底层数据结构选择的依据。