39 专题1：LRU Cache 最近最少使用算法

更新时间：2019-10-14 09:25:20

> 梦想只要能持久，就能成为现实。我们不就是生活在梦想中的吗？
>
> ——丁尼生

什么是 LRU Cache

LRU Cache 算法是 Least Recently Used，也就是最近最少使用算法。

对于一个操作系统来说，我们的缓存是有限的，所以有的时候我们必须要舍弃掉一些 object 来增加当前程序的运行效率。LRU Cache 算法的概念是：当缓存空间满了的时候，将最近最少使用的数据从缓存空间中删除以增加可用的缓存空间来缓存新的数据。这个算法的核心是一个缓存列表，当我们在缓存中的数据被访问的时候，这个数据就会被提到列表的头部，这样的话列表尾部的数据就是不会经常被访问的数据。当缓存空间不足的时候，就会删除列表尾部的数据来提升程序运行效率。

设计一个 LRU Cache

大多数注重算法能力的公司都会考相关的数据结构设计，LRU Cache 这个经典问题首当其冲，今天我就带着大家来实现一下它：

LRU Cache 的核心是什么？

LRU cache 相当于要维护一个跟时间顺序相关的数据结构，那么能找到最早更新元素的数据结构有 queue、heap 和 LinkedList 这几种：

- queue ：队列是先进先出的数据结构，我们确实知道哪一个元素是先进去的，哪一个元素是后进去的。
- heap ：我们可以通过维护一个丢入时间 time 来确定哪一个元素是先进去的，哪一个元素是后进去的。

- LinkedList ：链表有头部和尾部，我们自然而然可以利用头部来始终存放最新的元素（即最近使用的那个元素）

选择哪一种？

- 我们知道一旦某个中间进去的元素突然被调用后，我们就应该将它的位置更新到头部。频繁更新位置对于 queue 来说时间消耗太多，因此排除；
- 有时候我们要删除某一个很久不用的元素。如果用 heap 的话我们可能需要遍历所有的元素才能找到 time 为最早的那个节点，除非我们同时维护一个最小堆。但是每次更新元素位置还是需要 O(lgN) 的时间来调整，远不如 LinkedList 快；
- LinkedList 对于插入删除元素只需要 O(1) 的时间，但是我们还需要能够快速地访问到指定需要被更新的元素。这一点 LinkedList 要用 O(n) 遍历，但是我们可以通过一个字典来对应 key 和 node 的信息，这样就可以用 O(1) 的时间来找到对应元素了；
- 再进一步，由于要随时删除指定的 node，我们还需要将该 node 前的 node 与其后方的 node 相连，所以会有一个找指定 node 前面一个 node 的需求。双向链表 Doubly LinkedList 就可以用 O(1) 时间来实现这个需求，所以我们确定使用**双向链表** Doubly LinkedList **来完成一个** LRU Cache 。

确定了要使用的数据结构之后，我们来捋一捋接下来的思路：

1、 LRU Cache **整体设计：**

LRU Cache 里面维护一个 cache 字典对应 key 和 node 的信息。一个 cap 表示最大容量，一个双向链表，其中 head.next 是 most recently （最近使用）的 node，tail.prev 是 least recently（最近最少使用） 的 node（即 LRU 容量满了会被删除的那个 node）。

2、对于 get **方法：**

- 如果 key 在 cache 字典中，说明 node 在链表中
  - 根据 key 从 cache 字典中拿到对应的 node，从双向链表中删除这个 node，再向双向链表中重新插入这个 node（插入逻辑包含了更新到最新的位置）
- 如果不在直接返回 -1

3、对于 put **方法：**

- 如果 key 在 cache 字典中，说明 node 在链表中
  - 根据 key 从 cache 字典中拿到对应的 node，从双向链表中删除这个 node，再向双向链表中重新插入这个 node（插入逻辑包含了更新到最新的位置）
- 如果 key 不在 cache 字典中，说明是一个新的 node
  - 如果此时容量还没满的话：
    - 生成新 node，插入双向链表中，同时放入 cache 中
  - 如果此时容量满了的话：
    - 从双向链表中删除 tail.prev，即 least recently 的 node
    - 从 cache 中删除这个 node 的信息

- 生成新 `node`，插入双向链表中，放入 `cache` 中

其中 `逻辑3` 中如果 `key` 不在 `cache` 字典中的这一段代码可以优化，

生成新 `node`，插入链表中，放入 `cache` 中这一步是重复的。

> 其中 `逻辑3` 中如果 `key` 不在 `cache` 字典中的这一段代码可以优化。
>
> 优化的原因：生成新 `node`，插入链表中，放入 `cache` 中这一步是重复的。
> 优化的方法：定义一个insert方法，当要执行这一步操作的时候直接调用即可。

具体代码实现

下面是用 `Python`、 `Java` 和 `Go` 实现的代码，直接提交到 LeetCode 对应题目也可以 `Accept` 哟:

Python beats 95.66%

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



class LRUCache(object):
    def __init__(self, capacity):
        """
        :type capacity: int
        """
        self.cache = {}
        self.cap = capacity
        self.head = Node(None, None)
        self.tail = Node(None, None)
        self.head.next = self.tail
        self.tail.prev = self.head

    # 从双向链表中删除这个 node
    def remove(self, node):
        n = node.next
        p = node.prev
        p.next = n
        n.prev = p
        node.next = None
        node.prev = None

    # 向双向链表中插入 node，并放到最新位置
    def insert(self, node):
        n = self.head.next
        self.head.next = node
        node.next = n
        n.prev = node
        node.prev = self.head


    def get(self, key):
        """
        :type key: int
        :rtype: int
        """
```

```
        # 如果 key 在 cache 字典中，说明 node在链表中
        if key in self.cache:
            # 根据 key 从 cache 字典中拿到对应的 node
            node = self.cache[key]
            # 从双向链表中删除这个node
            self.remove(node)
            # 再向双向链表中重新插入这个 node（插入逻辑包含了更新到最新的位置）
            self.insert(node)
            return node.val
        else: # 如果不在直接返回 -1
            return -1


    def put(self, key, value):
        """
        :type key: int
        :type value: int
        :rtype: void
        """
    # 如果 key 在 cache 字典中，说明 node 在链表中
        if key in self.cache:
            # 根据 key 从 cache 字典中拿到对应的 node
            node = self.cache[key]
            # 更新 node 的 val
            node.val = value
            # 从双向链表中删除这个 `node`
            self.remove(node)
            # 再向双向链表中重新插入这个 node（插入逻辑包含了更新到最新的位置）
            self.insert(node)
        else: # 如果 key 不在 cache 字典中，说明是一个新的 node
            # 如果此时容量满了的话，我们需要先删除 least recently 的 node
            if len(self.cache) == self.cap:
                delete_node = self.tail.prev
                del self.cache[delete_node.key]
                self.remove(delete_node)
            # 生成新 node
            node = Node(key, value)
            # 插入双向链表中
            self.insert(node)
            # 同时放入 cache 中
            self.cache[key] = node


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


Java beats 85.25%

```
import java.util.HashMap;
/**
 * 维护一个类似双向链表的数据结构，包含前驱、后继以及 K-V 结构
 */
class Node {
    Node next;
    Node prev;
    int key;
    int val;

    public Node(int key, int val) {
        this.key = key;
        this.val = val;
    }

    public Node() {
    }
}
```

```
class LRUCache {
    // HashMap 保证插入结点和查找结点的时间复杂度为 O(1)
    HashMap<Integer, Node> map;
    // head.next 是 most recently 的结点
    Node head;
    // tail.prev 是 least recently 的结点
    Node tail;
    // capacity 表示最大容量
    int capacity;
    // count 表示当前已有结点数
    int count;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<>();
        head = new Node();
        tail = new Node();
        head.next = tail;
        tail.next = head;
    }

    public int get(int key) {
        // 如果 key 在 HashMap 中，先拿到该结点，删除结点，再插入结点。
        if (map.containsKey(key)) {
            Node node = map.get(key);
            remove(node);
            insert(node);
            return map.get(key).val;
        }
        // 如果不在就返回 -1
        return -1;
    }

    public void put(int key, int value) {
        // 如果 key 在 HashMap 中，和 get 类似，也是先拿到该结点，删除结点，再插入结点。
        if (map.containsKey(key)) {
            Node node = map.get(key);
            node.val = value;
            remove(node);
            insert(node);
        } else {
            // 如果 key 不在 HashMap 中，那么是一个新的结点，直接插入即可。
            Node node = new Node(key, value);
            insert(node);
        }
    }

    public void remove(Node node) {
        if (count > 0) {
            // 在 map 中移除结点前，先将双向链表的指针指向进行修改
            Node prev = node.prev;
            Node next = node.next;
            node.prev = null;
            node.next = null;
            next.prev = prev;
            prev.next = next;
            map.remove(node.key);
            count--;
        }
    }

    public void insert(Node node) {
        Node next = head.next;
        head.next = node;
        node.next = next;
        next.prev = node;
        node.prev = head;
        map.put(node.key, node);
```

```
            count++;
            // 如果结点数超过可允许容量，将 least recently 的结点移除
            if (count > capacity) {
                remove(tail.prev);
            }
        }
    }
```

Go beats 50.00%

```
type Node struct {
    key, val   int
    next, prev *Node
}

type LRUCache struct {
    capa  int
    cache map[int]*Node
    head  *Node
    tail  *Node
}

func remove(node *Node) { // 从双向链表中删除这个 node
    n := node.next
    p := node.prev
    p.next = n
    n.prev = p
    node.next = nil
    node.prev = nil
}

func insert(this *LRUCache, node *Node) { // 向双向链表中插入 node，并放到最新位置
    n := this.head.next
    this.head.next = node

    node.next = n
    n.prev = node
    node.prev = this.head
}

func Constructor(capacity int) LRUCache {
    this := LRUCache{
        capa:  capacity,
        cache: map[int]*Node{},
        head:  &Node{},
        tail:  &Node{},
    }
    this.head.next = this.tail
    this.tail.prev = this.head
    return this
}

func (this *LRUCache) Get(key int) int {
    if node, ok := this.cache[key]; ok { // 如果 `key` 在 `cache` 字典中，说明 `node` 在链表中
        remove(node) // 从双向链表中删除这个 `node`
        insert(this, node) // 再向双向链表中重新插入这个 `node`（插入逻辑包含了更新到最新的位置）
        return node.val
    } else {
        return -1 // 如果不在直接返回 `-1`
    }
}

func (this *LRUCache) Put(key int, value int) {
    if node, ok := this.cache[key]; ok { // 如果 `key` 在 `cache` 字典中，说明 `node` 在链表中
        node.val = value // 更新 node 的 val
        remove(node) // 从双向链表中删除这个 `node`
        insert(this, node) // 再向双向链表中重新插入这个 `node`（插入逻辑包含了更新到最新的位置）
```

```
 } else { // 如果 `key` 不在 `cache` 字典中，说明是一个新的 `node`
  if len(this.cache) == this.capa {
   deleteNode := this.tail.prev
   delete(this.cache, deleteNode.key)
   remove(deleteNode)
  }
  node := &Node{ // 生成新 `node`
   key: key,
   val: value,
  }
  insert(this, node) // 插入双向链表中
  this.cache[key] = node // 同时放入 `cache` 中
 }
}
```

总结

1. 看到一个问题我们要先问自己哪个地方是需要我们突破的，比如今天的难点就在于，如何通过我们想要实现的数据结构的所有特点来选取所需要的基础数据结构，这种能力也称为 未知变已知 的能力；
2. 运用我们知道的最基本的数据结构的优劣势来思考该如何设计一个新的数据结构；
3. 先不要着手于代码编写，可以在草稿纸上先画一下图，写一下之后的逻辑；
4. 运用好已有的一些语言包来快速且准确地实现之前确定的逻辑；
5. 此时代码可以 work，看看有没有值得优化的点，必要的话可以进行重构（参照代码可读性、代码可维护性、代码可扩展性、代码的效率），不要怕麻烦，要知道哪怕只是运行速度提高了一秒，对于自己也是一个巨大的进步；
6. 做到这一步就已经 over average 了，一定是个大大的 hire 在等着你！

}