本笔记参考神奕博客

设计一个LRU Cache

1: 什么是Cache

1. 概念
   Cache也就是高速缓存，它是介于CPU和内存之间的一种存储结构，在计算机系统的多级存储结构中，它的访问速度仅次于CPU寄存器。在计算机系统中，CPU的处理速度很快，但是系统对内存的访问以及数据的存取速度远远达不到CPU处理速度，因此高速缓存存在的目的就是减少计算机对内存的访问，使得数据访问速度尽可能的达到CPU处理速度。
2. 原理
   计算机系统对Cache的访问步骤如下：
   * 首先计算机CPU会发出对内存的访问请求；
   * 然后会先查看Cache内是否存在缓存数据，如果有，称为命中，则直接返回数据；
   * 如果Cache命中失败，称为miss，则去访问内存，然后先把内存中的数据存入缓存，再将数据返回处理；
   再上面的步骤中，有一个很重要就是当访问miss的时候，从内存取回的数据要存放在Cache中，如果Cache已经占满，那就需要进行数据替换。这就牵扯到Cache的核心——数据替换策略。
3. Cache背后的计算机原理
   为什么Cache能够实现对数据对快速访问，加大计算机的处理速度。其原因包括：

   - 其一：Cache的空间小，因而访问速度快；
   - 其二：程序执行和数据访问的**局部性原理:**
     - 时间局部性：如果程序中某条指令被执行，那么很可能不久后该指令还会执行；如果某数据被访问，那么不久后改数据还会被访问；
     - 空间局部性：程序一旦访问了内存中某个存储单元，那么在不久之后，该存储单元附近的单元也会被访问；也就是说在某一段时间内，程序访问的内存地址集中在一定的范围之内，这是因为在计算机中程序指令和数据存放通常是按照顺序存放的；

4. 替换策略
   因为Cache的内存有限，因此当内存占满，且访问miss之后，就需要对Cache中的数据进行替换。替换一般有以下几种策略：

   - 随机算法(Rand): 由系统产生一个随机数用来确定被替换块。这种方法简单，容易实现，但命中率低；
   - 先进先出(FIFO)：先进先出算法总是被替换掉的是首先进来对数据块，而不管它是否被使用。因而这种算法不符合局部性原理；
   - 最久未使用算法(LRU, Least Recently Used)： 这种算法根据Cache中各数据块的使用情况，总是选择最长时间没被使用的数据块替换。这种算法较好的反应了句不行原理；
   - 最不经常使用(LFU, Least Frequently Used)：将最近一段时间内，访问次数最少的块替换出去；

2: Cache的实现

Design an LRU Cache with all the operations to be done in O(1).

对一个Cache的操作一般来说就是三种：插入、查找、替换。
因为我们需要实现的是LRU，因此需要维护一个双向链表以保证插入的高效；同时维护链表的表头到表尾是从最近访问到最旧访问顺序。

- 插入：因为是双向链表，插入的时候往表头插入，时间O(1);
- 替换：当Cache已经占满，删除表尾数据，然后往表头插入新数据；时间复杂度同样为O(1);
- 查找：将每次查找到的数据项放到链表头；
  由于查找需要遍历双向链表，显然违背时间复杂度要求，因此可以同时可以使用一个hash map来实现查找。

总结一下：
要实现一个LRU,要用到两种数据结构；

- 双向链表：保证插入和替换的时间复杂度
- hashmap：保证查找的时间复杂度

看代码：

```
//先定义双向链表节点
public class DoubleNode{
    int val;//当前链表存储的数据
    int key;//当前node的key
    DoubleNode prev;
    DoubleNode next;
    public DoubleNode() {
        this.val = 0;
        this.key = 0;
        this.prev = null;
        this.next = null;
    }
}

//定义双向链表
public class DoubleLinkedList {
    private HashMap<Integer, DoubleNode> map;
    private int capacity;
    private int count;
    private DoubleNode head;
    private DoubleNode tail;

    public DoubleLinkedList(int capacity) {
        this.capacity = capacity;
        this.count = 0;
        map = new HashMap<>();
        head = new DoubleNode();// 两个空节点，单纯的为访问
        tail = new DoubleNode();
        head.next = tail;
        head.prev = null;
        tail.next = null;
        tail.prev = head;
    }

    //首先实现查找方法
    public int get(int key) {
        if (map.get(key) == null) {
            return -1;
        } else {
            DoubleNode cur = map.get(key);
            detach(cur);//剥离这个节点
            insertFront(cur);//从表头插入当前查找节点
            return cur.val;
        }
    }

    //接下来实现访问并设置算法，未满插入，占满替换。
    public void set(int key, int value) {
        if (map.get(key) == null) {
            DoubleNode cur = new DoubleNode();
            if (count == capacity) {
                deleteNode();
            }
            node.key = key;
            node.value = value;
            map.put(key, cur);
            insertFront(cur);
            count++;
        } else {
            DoubleNode cur = map.get(key);
            detach(cur);
            cur.value = value;
            inserFront(cur);
        }
    }

    //实现删除尾部节点，头部插入节点，以及剥离节点方法
    private void detach(DoubleNode cur) {
        cur.prev.next = cur.next;
        cur.next.prev = cur.prev;
        cur.next = null;
        cur.prev = null;
    }

    private void insertFront(DoubleNode cur) {
        cur.next = head.next;
        cur.prev = head;
        head.next = cur;
        cur.next.prev = cur;
    }

    private void deleteNode() {
        DoubleNode cur = tail.prev;
        detach(cur);
        map.put(cur.key, null);
        count--;
    }
}
```