聊聊缓存淘汰算法 -LRU 实现原理

作者：9526xu
原文链接：https://ld246.com/article/1572314385905
来源网站：链滴

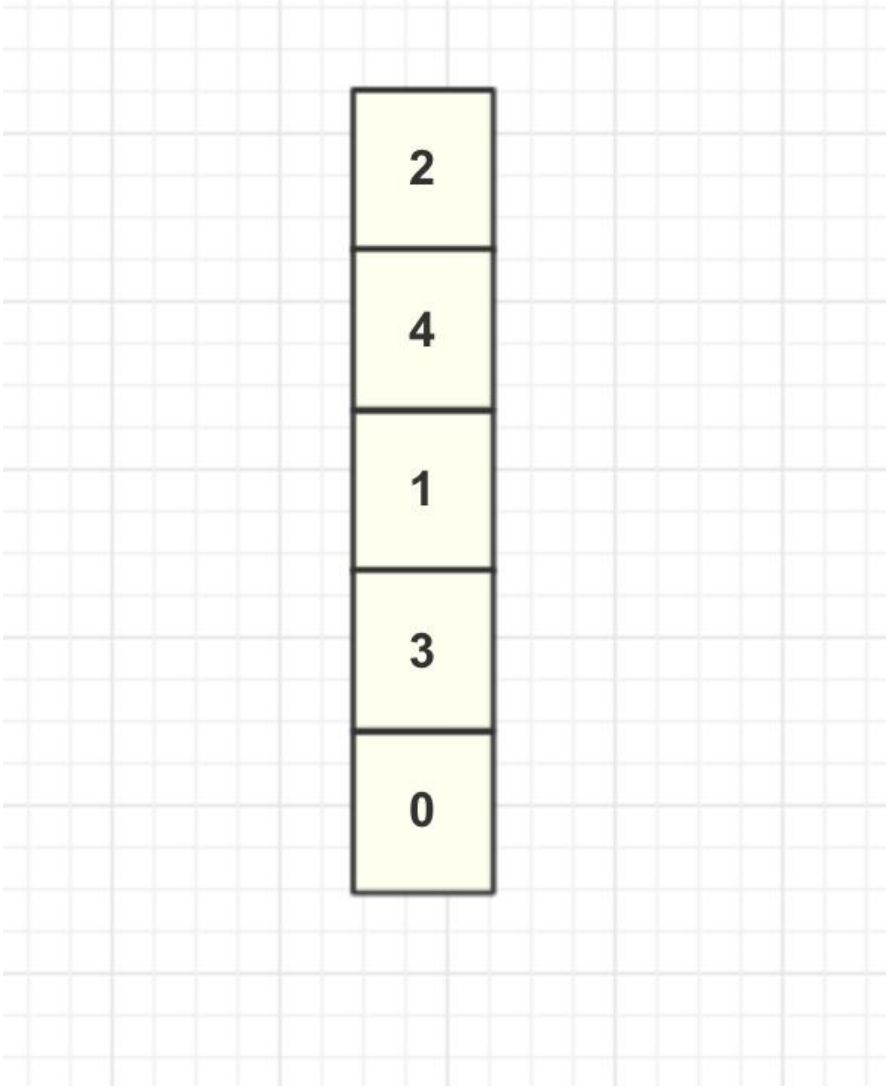
前言

我们常用缓存提升数据查询速度，由于缓存容量有限，当缓存容量到达上限，就需要删除部分数据挪空间，这样新数据才可以添加进来。缓存数据不能随机删除，一般情况下我们需要根据某种算法删除存数据。常用淘汰算法有 LRU,LFU,FIFO,这篇文章我们聊聊 LRU 算法。

LRU 简介

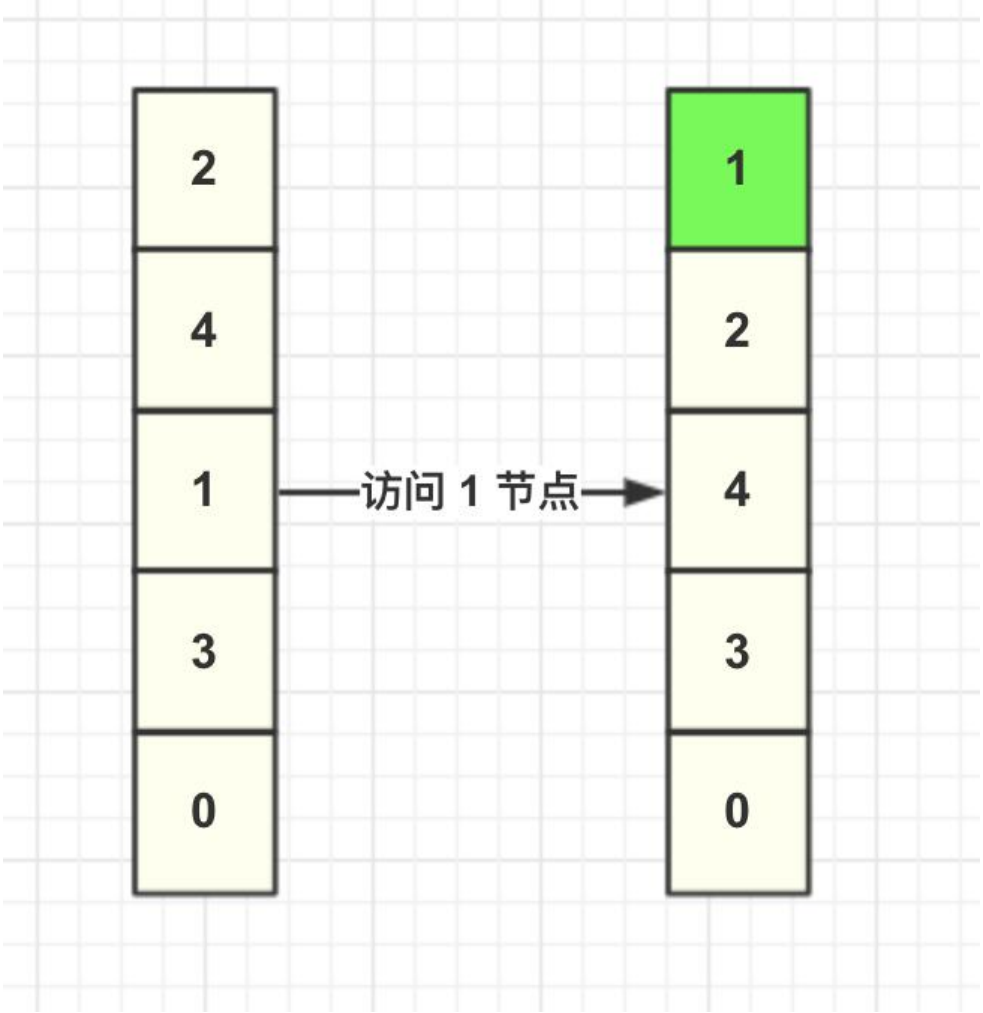
LRU 是 Least Recently Used 的缩写，这种算法认为最近使用的数据是热门数据，下一次很大概率会再次被使用。而最近很少被使用的数据，很大概率下一次不再用到。当缓存容量的满时候，优先淘最近很少使用的数据。

假设现在缓存内部数据如图所示：

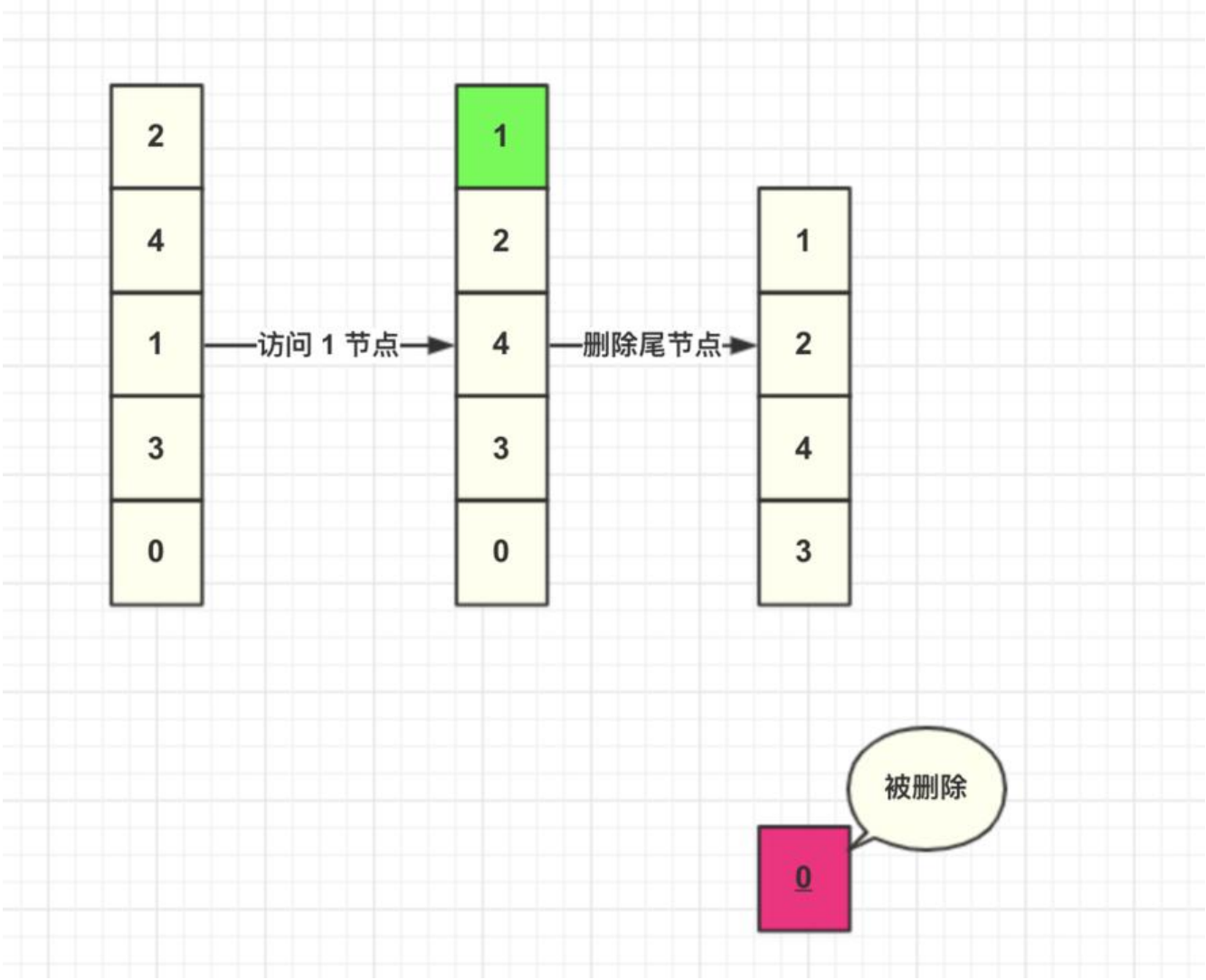


这里我们将列表第一个节点称为头结点，最后一个节点为尾结点。

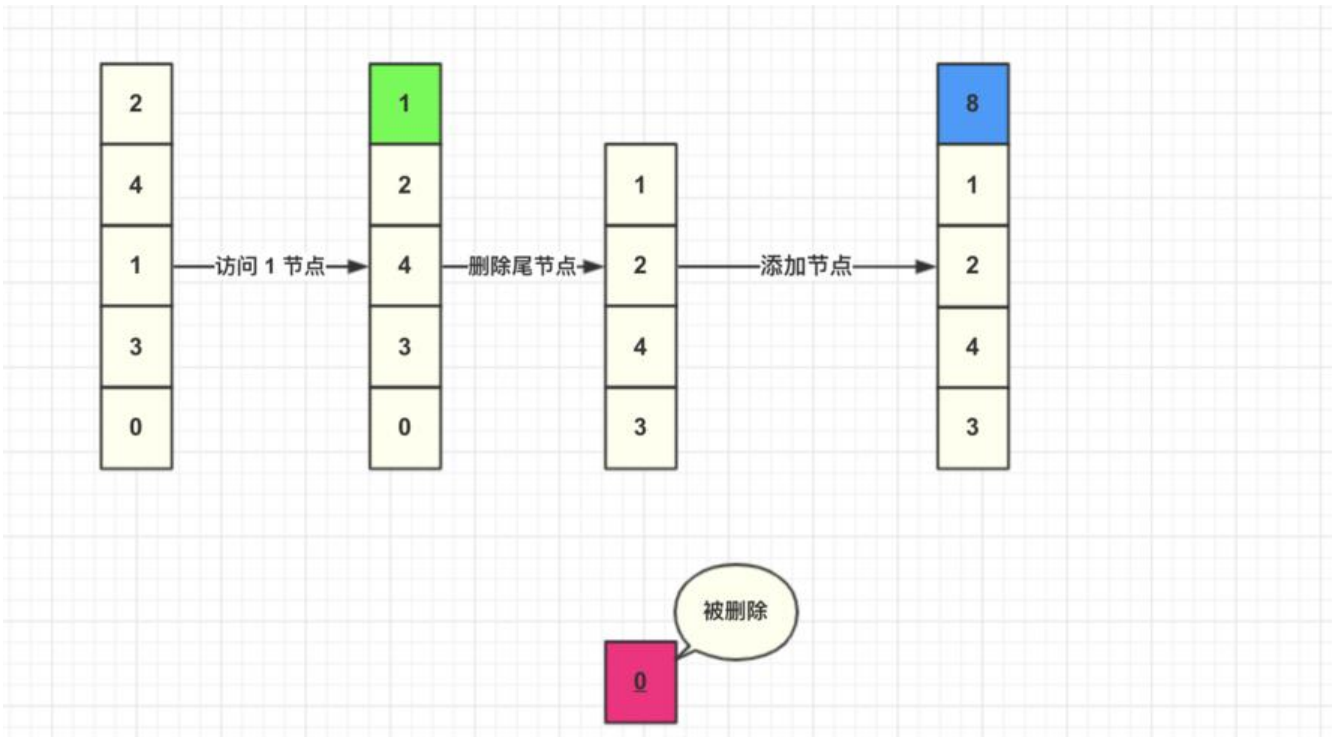
当调用缓存获取 key=1 的数据，LRU 算法需要将 1 这个节点移动到头结点，其余节点不变，如图所。



然后我们插入一个 key=8 节点，此时缓存容量到达上限，所以加入之前需要先删除数据。由于每次询都会将数据移动到头结点，未被查询的数据就将会下沉到尾部节点，尾部的数据就可以认为是最少访问的数据，所以删除尾结点的数据。



然后我们直接将数据添加到头结点。



原文链接：聊聊缓存淘汰算法 -LRU 实现原理

这里总结一下 LRU 算法具体步骤：

- 新数据直接插入到列表头部
- 缓存数据被命中，将数据移动到列表头部
- 缓存已满的时候，移除列表尾部数据。

LRU 算法实现

上面例子中可以看到，LRU 算法需要添加头节点，删除尾结点。而链表添加节点/删除节点时间复杂度 O(1)，非常适合当做存储缓存数据容器。但是不能使用普通的单向链表，单向链表有几点劣势:

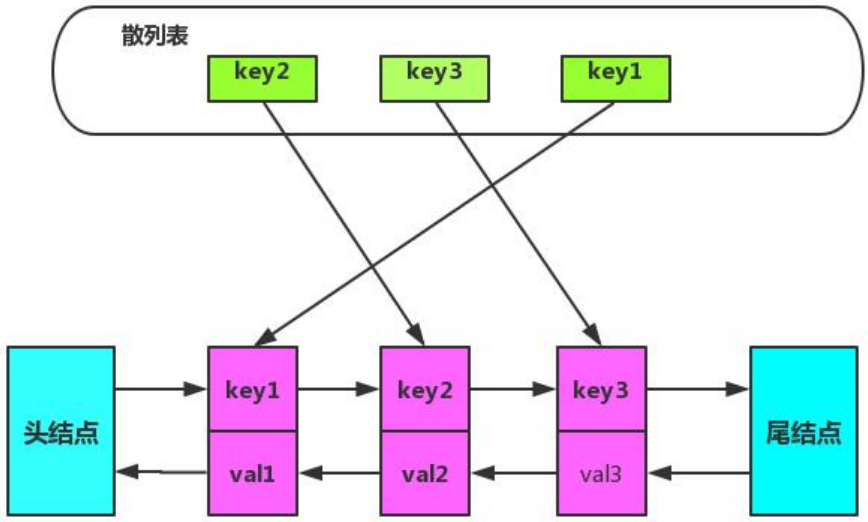
1. 每次获取任意节点数据，都需要从头结点遍历下去，这就导致获取节点复杂度为 O(N)。

2. 移动中间节点到头结点，我们需要知道中间节点前一个节点的信息，单向链表就不得不再次遍历获信息。

针对以上问题，可以结合其他数据结构解决。

使用散列表存储节点，获取节点的复杂度将会降低为 O(1)。节点移动问题可以在节点中再增加前驱指，记录上一个节点信息，这样链表就从单向链表变成了双向链表。

综上使用双向链表加散列表结合体，数据结构如图所示:



在双向链表中特意增加两个『哨兵』节点，不用来存储任何数据。使用哨兵节点，增加/删除节点的候就可以不用考虑边界节点不存在情况，简化编程难度，降低代码复杂度。

LRU 算法实现代码如下，为了简化 key ， val 都认为 int 类型。

```
public class LRUCache {

    Entry head, tail;
    int capacity;
    int size;
    Map<Integer, Entry> cache;
```

```
public LRUCache(int capacity) {
    this.capacity = capacity;
    // 初始化链表
    initLinkedList();
    size = 0;
    cache = new HashMap<>(capacity + 2);
}

/**
 * 如果节点不存在，返回 -1.如果存在，将节点移动到头结点，并返回节点的数据。
 *
 * @param key
 * @return
 */
public int get(int key) {
    Entry node = cache.get(key);
    if (node == null) {
        return -1;
    }
    // 存在移动节点
    moveToHead(node);
    return node.value;
}

/**
 * 将节点加入到头结点，如果容量已满，将会删除尾结点
 *
 * @param key
 * @param value
 */
public void put(int key, int value) {
    Entry node = cache.get(key);
    if (node != null) {
        node.value = value;
        moveToHead(node);
        return;
    }
    // 不存在。先加进去，再移除尾结点
    // 此时容量已满 删除尾结点
    if (size == capacity) {
        Entry lastNode = tail.pre;
        deleteNode(lastNode);
        cache.remove(lastNode.key);
        size--;
    }
    // 加入头结点

    Entry newNode = new Entry();
    newNode.key = key;
    newNode.value = value;
    addNode(newNode);
    cache.put(key, newNode);
    size++;
```

```
}

private void moveToHead(Entry node) {
    // 首先删除原来节点的关系
    deleteNode(node);
    addNode(node);
}

private void addNode(Entry node) {
    head.next.pre = node;
    node.next = head.next;

    node.pre = head;
    head.next = node;
}

private void deleteNode(Entry node) {
    node.pre.next = node.next;
    node.next.pre = node.pre;
}


public static class Entry {
    public Entry pre;
    public Entry next;
    public int key;
    public int value;

    public Entry(int key, int value) {
        this.key = key;
        this.value = value;
    }

    public Entry() {
    }
}

private void initLinkedList() {
    head = new Entry();
    tail = new Entry();

    head.next = tail;
    tail.pre = head;

}

public static void main(String[] args) {

    LRUCache cache = new LRUCache(2);

    cache.put(1, 1);
    cache.put(2, 2);
    System.out.println(cache.get(1));
```

```
        cache.put(3, 3);
        System.out.println(cache.get(2));

    }
}
```

LRU 算法分析

缓存命中率是缓存系统的非常重要指标，如果缓存系统的缓存命中率过低，将会导致查询回流到数据，导致数据库的压力升高。

结合以上分析 LRU 算法优缺点。

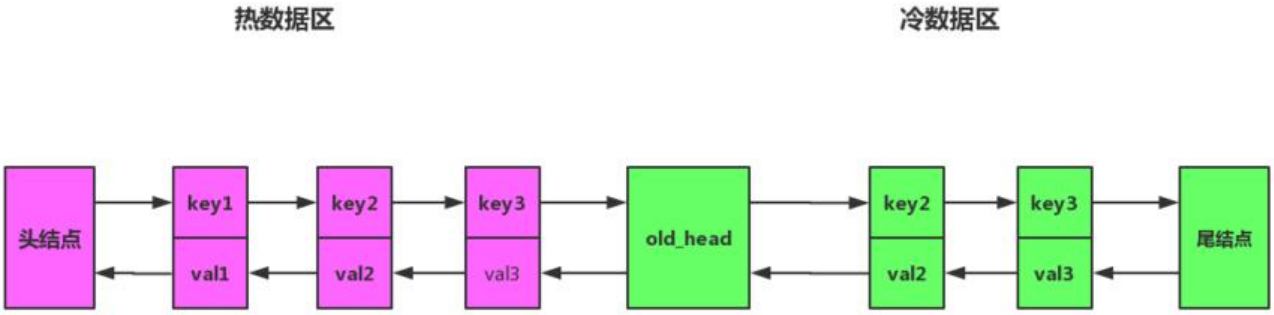
LRU 算法优势在于算法实现难度不大，对于对于热点数据， LRU 效率会很好。

LRU 算法劣势在于对于偶发的批量操作，比如说批量查询历史数据，就有可能使缓存中热门数据被这历史数据替换，造成缓存污染，导致缓存命中率下降，减慢了正常数据查询。

LRU 算法改进方案

以下方案来源与 MySQL InnoDB LRU 改进算法

将链表拆分成两部分，分为热数据区，与冷数据区，如图所示。



改进之后算法流程将会变成下面一样:

1. 访问数据如果位于热数据区，与之前 LRU 算法一样，移动到热数据区的头结点。
2. 插入数据时，若缓存已满，淘汰尾结点的数据。然后将数据 **插入冷数据区**的头结点。
3. 处于冷数据区的数据每次被访问需要做如下判断:
    - 若该数据已在缓存中超过指定时间，比如说 1 s，则移动到热数据区的头结点。
    - 若该数据存在在时间小于指定的时间，则位置保持不变。

对于偶发的批量查询，数据仅仅只会落入冷数据区，然后很快就会被淘汰出去。热门数据区的数据将会受到影响，这样就解决了 LRU 算法缓存命中率下降的问题。

其他改进方法还有 LRU-K，2Q,LIRS 算法，感兴趣同学可以自行查阅。