链表

虚拟头节点

当你需要创造一条新链表的时候，可以使用虚拟头结点简化边界情况的处理。

```
ListNode* dummy = new ListNode(-1)
```

总的来说，如果我们需要把原链表的节点接到新链表上，而不是 new 新节点来组成新链表的话，那么断开节点和原链表之间的链接可能是必要的。那其实我们可以养成一个好习惯，但凡遇到这种情况，就把原链表的节点断开，这样就不会出错了。

数组

前缀和

```
NumArray(vector<int>& nums) {
    vector<int> preSum;
    // preSum[0] = 0，便于计算累加和
    preSum.resize(nums.size() + 1);
    // 计算 nums 的累加和
    for (int i = 1; i < preSum.size(); i++) {
        preSum[i] = preSum[i - 1] + nums[i - 1];
    }
}
```

如果我们想求区间nums[i..j]的累加和，只要计算`preSum[j+1] - preSum[i]`即可。

差分数组

```
// 差分数组工具类
class Difference {
    // 差分数组
    private:
        int* diff;

    /* 输入一个初始数组，区间操作将在这个数组上进行 */
    public:
        Difference(int* nums, int length) {
            assert(length > 0);
            diff = new int[length]();
            diff[0] = nums[0];
            for (int i = 1; i < length; i++) {
                diff[i] = nums[i] - nums[i - 1];
            }
        }

        /* 给闭区间 [i, j] 增加 val（可以是负数）*/
        void increment(int i, int j, int val) {
            diff[i] += val;
            if (j + 1 < sizeof(diff) / sizeof(diff[0])) {
                diff[j + 1] -= val;
```

```
            }
        }

        /* 返回结果数组 */
        int* result() {
            int* res = new int[sizeof(diff) / sizeof(diff[0])]();
            res[0] = diff[0];
            for (int i = 1; i < sizeof(diff) / sizeof(diff[0]); i++) {
                res[i] = res[i - 1] + diff[i];
            }
            return res;
        }
};
```

双向链表+哈希表实现LFU

```
struct Node {
    int key;
    int val;
    int freq;
    Node* prev;
    Node* next;

    Node () : key(-1), val(-1), freq(0), prev(nullptr), next(nullptr) {}

    Node (int _k, int _v) : key(_k), val(_v), freq(1), prev(nullptr),
next(nullptr) {}
};
struct FreqList {
    int freq;
    Node* vhead;
    Node* vtail;

    FreqList (int _f) : freq(_f), vhead(new Node()), vtail(new Node()) {
        vhead->next = vtail;
        vtail->prev = vhead;
    }
};
class LFUCache {
private:
    unordered_map<int, Node*> occ;
    unordered_map<int, FreqList*> freq_map;
    int sz;
    int min_freq;
public:
    LFUCache (int capacity) : sz(capacity) {}

    bool empty(FreqList* l) {
        return l->vhead->next == l->vtail ? true : false;
    }

    void deleteNode (Node* t) {
        t->prev->next = t->next;
        t->next->prev = t->prev;
    }

    void addHead (Node* t) {
```

```
        int freq = t->freq;
        if (freq_map.find(freq) == freq_map.end()) {
            freq_map[freq] = new FreqList(freq);
        }
        FreqList* l = freq_map[freq];
        t->next = l->vhead->next;
        l->vhead->next->prev = t;
        t->prev = l->vhead;
        l->vhead->next = t;
    }

    void popTail () {
        Node* t = freq_map[min_freq]->vtail->prev;
        deleteNode(t);
        occ.erase(t->key);
    }

    int get (int key) {
        int res = -1;
        if (occ.find(key) != occ.end()) {
            Node* t = occ[key];
            res = t->val;
            deleteNode(t);
            t->freq++;
            if (empty(freq_map[min_freq])) min_freq++;
            addHead(t);
        }
        return res;
    }

    void put (int key, int value) {
        if (sz == 0) return;
        if (get(key) != -1) {
            occ[key]->val = value;
        }
        else {
            if (occ.size() == sz) {
                popTail();
            }
            Node* t = new Node(key, value);
            occ[key] = t;
            min_freq = 1;//新插入的 频率一定最少, 为1
            addHead(t);
        }
    }
};
```