旋转链表\_LeetCode\_61\_RotateList\_Medium

# 旋转链表\_LeetCode\_61\_RotateList\_Medium

## 题目介绍

/\*\*

\* LeetCode\_61\_RotateList\_Medium\_旋转数组

\* https://leetcode.com/problems/rotate-list/description/

\* 难度:Medium

\* DateTime:2018-10-10 22:46

\* <p>

\* 题目介绍：

\* Given a linked list, rotate the list to the right by k places, where k is non-negative.

\* <p>

\* Example 1:

\* Input: 1->2->3->4->5->NULL, k = 2

\* Output: 4->5->1->2->3->NULL

\* Explanation:

\* rotate 1 steps to the right: 5->1->2->3->4->NULL

\* rotate 2 steps to the right: 4->5->1->2->3->NULL

\* <p>

\* Example 2:

\* Input: 0->1->2->NULL, k = 4

\* Output: 2->0->1->NULL

\* Explanation:

\* rotate 1 steps to the right: 2->0->1->NULL

\* rotate 2 steps to the right: 1->2->0->NULL

\* rotate 3 steps to the right: 0->1->2->NULL

\* rotate 4 steps to the right: 2->0->1->NULL

\* <p>

## 思路分析

\* 思路分析：

\* \* 注意：测试用例中，有这么几个特殊的：

\* \* 1. k特别大，因此需要适当处理k；

\* \* 必须求出链表长度，然后k%=len;

\* \* 2. 边界情况：k=0;或者head==null;

\* \* 3. 边界情况：链表长度为1，k=1的情况。

\* 方法1：前后指针+求余法。

\* S1. front指针先走k步;若k>=N,对k进行求余处理;

\* S2. front、behind双指针同时前进至front到达最后一个节点;

\* S3. 得到的两段链表，处理连接关系。

\* 方法2: 遍历+求余+寻找。

\* S1:首先遍历一遍链表，得到链表长度和链表的尾节点,并首尾相连;

\* S2:处理k,利用for循环寻找旋转后链表的尾节点;

\* S3:处理尾节点和旋转后头节点。

\* 方法3: 栈+求余+向前寻找

\* S1:利用栈得出链表的长度;

\* S2:获取旋转后的链表的头节点;

\* S3:处理旋转链表的尾节点

## Java代码

/\*\*

\* 方法1:前后指针+求余法

\*/

public ListNode rotateRight(ListNode head, int k) {

if (head == null || k <= 0) return head;

ListNode front = head, behind = head;

int len = 0;//记录链表长度

//S1. 初始化front指针

//两种情况：1. k小于链表长度时，直接前进k步;

//2. k大于链表长度时，首先遍历一边链表，求出链表长度len;然后k%=len;

//同样再走剩下的k步;最大复杂度是走2N-1。

//若k<=N;走k步;若N<k<2N-1,走k步;

//若k>=2N;走k%N+N步;

while (k-- > 0) {

front = front.next;//前进一步

len++;

if (front == null) {

k %= len;//降低k是N好几倍情况的复杂度

front = head;//从头开始走

}

}

//S2. front和behind同时前进;front走到最后一个节点;behind走到前一段的最后一个节点

if (front == head) return head;//k正好是N的倍数，不需要再次遍历一遍;排除了front和behind重叠的情况;

//front和behind重叠的话后续很容易出错

while (front.next != null) {

front = front.next;

behind = behind.next;

}

//S3. 连接关系处理

front.next = head;

ListNode newHead = behind.next;//旋转之后的头节点

behind.next = null;

return newHead;

}

/\*\*

\* 方法2:遍历+求余+寻找

\*/

public ListNode rotateRight2(ListNode head, int k) {

if (head == null || k <= 0) return head;

ListNode currNode = head;

//S1:统计链表长度,并寻找最后一个节点

int len = 1;

while (currNode.next != null) {

currNode = currNode.next;

len++;

}

currNode.next = head;//首尾相连

//S2:寻找旋转后链表的尾节点和头节点

currNode = head;

k %= len;

k = len - k - 1; //k是倒数的,因此从前往后需走len-k-1步到newHeadPre

for (int i = 0; i < k; i++) {

currNode = currNode.next;

}

//S3:处理尾节点和旋转链表节点

ListNode lastNode = currNode.next;

currNode.next = null;

return lastNode;

}

/\*\*

\* 方法3:利用栈的方法

\*/

public ListNode rotateRight3(ListNode head, int k) {

if (head == null || k <= 0) return head;

Deque<ListNode> stack = new ArrayDeque<ListNode>();

ListNode currNode = head;

ListNode newHead, lastNode;

//S1:压栈并统计链表长度,处理k,并获取最后一个节点

int len = 0;

while (currNode != null) {

stack.push(currNode);

currNode = currNode.next;

len++;

}

//处理k,获取最后一个节点

k %= len;//处理k

if (k == 0) return head;//k是链表长度的整数倍

//获取最后一个节点

k--;

lastNode = stack.pop();

//S2:获取旋转后的链表的头节点

newHead = lastNode;//若k=0,此时newHead与lastNode重合

while (k-- > 0) newHead = stack.pop();

//S3:处理旋转链表的尾节点

stack.pop().next = null;//尾节点next指向null

lastNode.next = head;

return newHead;

}