0025. K 个一组翻转链表

ITCharge 大约 4 分钟

- 标签：递归、链表
- 难度：困难

题目链接

- 0025. K 个一组翻转链表 - 力扣

题目大意

描述：给你链表的头节点 `head` ，再给定一个正整数 `k` ， `k` 的值小于或等于链表的长度。

要求：每 `k` 个节点一组进行翻转，并返回修改后的链表。如果链表节点总数不是 `k` 的整数倍，则将最后剩余的节点保持原有顺序。

说明：

- 不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。
- 假设链表中的节点数目为 `n` 。
- $1 \leq k \leq n \leq 5000$。
- $0 \leq Node.val \leq 1000$。
- 要求设计一个只用 `O(1)` 额外内存空间的算法解决此问题。

示例：

- 示例 1：

```py
输入：head = [1,2,3,4,5], k = 2
输出：[2,1,4,3,5]
```

解题思路

思路 1：迭代

在「0206. 反转链表 」中我们可以通过迭代、递归两种方法将整个链表反转。而这道题要求以 `k` 为单位，对链表的区间进行反转。而区间反转其实就是「0092. 反转链表 II 」这道题的题目要求。

本题中，我们可以以 `k` 为单位对链表进行切分，然后分别对每个区间部分进行反转。最后再返回头节点即可。

但是需要注意一点，如果需要反转的区间包含了链表的第一个节点，那么我们可以事先创建一个哑节点作为链表初始位置开始遍历，这样就能避免找不到需要反转的链表区间的前一个节点。

这道题的具体解题步骤如下：

1. 先使用哑节点 `dummy_head` 构造一个指向 `head` 的指针，避免找不到需要反转的链表区间的前一个节点。使用变量 `index` 记录当前元素的序号。
2. 使用两个指针 `cur` 、 `tail` 分别表示链表中待反转区间的首尾节点。初始 `cur` 赋值为 `dummy_head` ， `tail` 赋值为 `dummy_head.next` ，也就是 `head` 。
3. 将 `tail` 向右移动，每移动一步，就领 `index` 加 `1` 。
    1. 当 `index % k != 0` 时，直接将 `tail` 向右移动，直到移动到当前待反转区间的结尾位置。
    2. 当 `index % k == 0` 时，说明 `tail` 已经移动到了当前待反转区间的结尾位置，此时调用 `cur = self.reverse(cur, tail.next)` ，将待反转区间进行反转，并返回反转后区间的起始节点赋值给当前反转区间的首节点 `cur` 。然后将 `tail` 移动到 `cur` 的下一个节点。
4. 最后返回新的头节点 `dummy_head.next` 。

关于 `def reverse(self, head, tail):` 方法这里也说下具体步骤：

1. `head` 代表当前待反转区间的第一个节点的前一个节点， `tail` 代表当前待反转区间的最后一个节点的后一个节点。
2. 先用 `first` 保存一下待反转区间的第一个节点（反转之后为区间的尾节点），方便反转之后进行连接。
3. 我们使用两个指针 `cur` 和 `pre` 进行迭代。 `pre` 指向 `cur` 前一个节点位置，即 `pre` 指向需要反转节点的前一个节点， `cur` 指向需要反转的节点。初始时， `pre` 指向待反转

区间的第一个节点的前一个节点 `head` ， `cur` 指向待反转区间的第一个节点，即 `pre.next` 。

4. 当当前节点 `cur` 不等于 `tail` 时，将 `pre` 和 `cur` 的前后指针进行交换，指针更替顺序为：
   1. 使用 `next` 指针保存当前节点 `cur` 的后一个节点，即 `next = cur.next` ；
   2. 断开当前节点 `cur` 的后一节点链接，将 `cur` 的 `next` 指针指向前一节点 `pre` ，即 `cur.next = pre` ；
   3. `pre` 向前移动一步，移动到 `cur` 位置，即 `pre = cur` ；
   4. `cur` 向前移动一步，移动到之前 `next` 指针保存的位置，即 `cur = next` 。
5. 继续执行第 `4` 步中的 `1` 、 `2` 、 `3` 、 `4` 步。
6. 最后等到 `cur` 遍历到链表末尾（即 `cur == tail` ）时，令「当前待反转区间的第一个节点的前一个节点」指向「反转区间后的头节点」，即 `head.next = pre` 。令「待反转区间的第一个节点（反转之后为区间的尾节点）」指向「待反转分区间的最后一个节点的后一个节点」，即 `first.next = tail` 。
7. 最后返回新的头节点 `dummy_head.next` 。

思路 1：代码

py

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0,     =None):
#         self.val = val
#         self.next = next
class Solution:
    def reverse(self, head, tail):
        pre = head
        cur = pre.next
        first = cur
        while cur != tail:
            next = cur.next
            cur.next = pre
            pre = cur
            cur = next
        head.next = pre
        first.next = tail
        return first

    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        dummy_head = ListNode(0)
        dummy_head.next = head
        cur = dummy_head
```

```
        tail = dummy_head.next
        index = 0
        while tail:
            index += 1
            if index % k == 0:
                cur = self.reverse(cur, tail.next)
                tail = cur.next
            else:
                tail = tail.next
        return dummy_head.next
```

思路 1：复杂度分析

- **时间复杂度**：$O(n)$。其中 n 为链表的总长度。
- **空间复杂度**：$O(1)$。