# 32 K 个一组翻转链表

更新时间：2019-09-20 10:34:43




> 合理安排时间，就等于节约时间。
>
> ——培根

## 刷题内容

> **难度: Hard**
>
> 原题连接:https://leetcode-cn.com/problems/reverse-nodes-in-k-group/

### 内容描述

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

示例 :

给定这个链表：1->2->3->4->5

当 k = 2 时，应当返回: 2->1->4->3->5

当 k = 3 时，应当返回: 3->2->1->4->5

说明 :

你的算法只能使用常数的额外空间。
你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

# 解题方案

## 思路 1：时间复杂度: O(N) 空间复杂度: O(N/K)

这个题目其实就是将24题泛化了，K如果固定成2的话就是24题原封不动，因此我们仍然可以递归操作，  有两种情况：

1. 压根没有k个node，那么我们直接保持这个k-group不动返回head
2. 如果有k个node的话，那么我们先找到第k个node之后的递归结果 node = nxt，然后反转前面k个node，让反转结果的结尾 tail.next = nxt

***Python beats 92.70%***

```python
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        cur = head
        cnt = 0
        while cur and cnt != k: # 往后最多走k步
            cur = cur.next
            cnt += 1
        if cnt == k: # 如果当前 k-group 有 k 个node的话
            # 先找到第k个node之后的递归结果 node = nxt
            # 让反转结果的结尾 tail.next = nxt
            nxt = self.reverseKGroup(cur, k)
            while cnt > 0: # 反转前面k个node
                tmp = head.next
                head.next = nxt
                nxt = head
                head = tmp
                cnt -= 1
            return nxt
        # 当前 k-group 压根没有k个node，那么我们直接保持这个k-group不动返回head
        return head
```



***java beats 100%***

```java
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        int count = 0;
        ListNode cur = head, tmp;
        while (cur != null && count < k) {
            count++;
            cur = cur.next;
        }
        // 当前 k-group 压根没有k个node，那么我们直接保持这个k-group不动返回head
        if (count < k) {
            return head;
        }
        // last是k+1个节点后的链表的翻转结果
        ListNode last = reverseKGroup(cur, k);
        // 从第一个节点开始反转，第一个节点挂在last前面，把last换成第一个节点
        // 第二个节点挂在last前面，继续把last换成第一个节点，直到把k个节点都反转完
        for (count = 0;count < k;count++) {
            tmp = head;
            head = head.next;
            tmp.next = last;
            last = tmp;
        }
        return last;
    }
}
```

***c++ beats 98.11%***

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        int count = 0;
        ListNode *cur = head, *tmp;
        while (cur && count < k) {
            count++;
            cur = cur->next;
        }
        // 当前 k-group 压根没有k个node，那么我们直接保持这个k-group不动返回head
        if (count < k) {
            return head;
        }
        // last是k+1个节点后的链表的翻转结果
        ListNode* last = reverseKGroup(cur, k);
        // 从第一个节点开始反转，第一个节点挂在last前面，把last换成第一个节点
        // 第二个节点挂在last前面，继续把last换成第一个节点，直到把k个节点都反转完
        for (count = 0;count < k;count++) {
            tmp = head;
            head = head->next;
            tmp->next = last;
            last = tmp;
        }
        return last;
    }
};
```

***go beats 97.7%***



```
func reverseKGroup(head *ListNode, k int) *ListNode {
    count := 0
 cur := head
 var tmp *ListNode
 for cur != nil && count < k {
  count++;
  cur = cur.Next;
 }
 // 当前 k-group 压根没有k个node，那么我们直接保持这个k-group不动返回head
 if count < k {
  return head;
 }
 // last是k+1个节点后的链表的翻转结果
 last := reverseKGroup(cur, k);
 // 从第一个节点开始反转，第一个节点挂在last前面，把last换成第一个节点
 // 第二个节点挂在last前面，继续把last换成第一个节点，直到把k个节点都反转完
 for count = 0;count < k;count++ {
  tmp = head;
  head = head.Next;
  tmp.Next = last;
  last = tmp;
 }
 return last;
}
```

之前我们也用过递归的操作，然后发现使用迭代来代替递归通常能够节省空间，接下来让我们试试吧

## 思路 2：时间复杂度: O(N) 空间复杂度: O(1)

思路 2 其实就是思路 1 的迭代版本，理清楚 node 之间的关系之后可以说是 so easy！

***Python beats 98.1%***

```python
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        dummy = dummy2 = ListNode(0)
        dummy2.next = l = r = head

        while True:
            count = 0
            while r and count < k:
                r = r.next
                count += 1
            if count == k:
                pre, cur = r, l
                for _ in range(k): # 反转当前的k-group
                    nxt = cur.next
                    cur.next = pre
                    pre = cur
                    cur = nxt
                dummy2.next = pre
                dummy2 = l
                l = r
            else: # 压根没有k个node，那么我们直接保持这个k-group不动返回head
                return dummy.next
```

***c++ beats 98.1%***

```cpp
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* dummy = new ListNode(-1);
        dummy->next = head;
        ListNode* dummy2 = dummy;
        while (1) {
            ListNode* p = dummy2->next;
            ListNode* start = p;
            int count = 0;
            while (count < k && p != NULL) {
                p = p->next;
                count++;
            }
            //如果少于k个节点，则不需要翻转
            if (count < k) {
                break;
            }
            //k个节点后的那个节点
            ListNode* last = p;
            //一个一个连过去
            p = dummy2->next;
            for (int i = 0;i < k;i++) {
                ListNode* next = p->next;
                p->next = last;
                last = p;
                p = next;
            }
            //翻转后的结果
            dummy2->next = last;
            //当前的第k个数据就是先前的第一个数据
            dummy2 = start;
        }
        return dummy->next;
    }
};
```



***java beats 39.85%***

```java
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode dummy2 = dummy;
        while (true) {
            ListNode p = dummy2.next;
            ListNode start = p;
            int count = 0;
            while (count < k && p != null) {
                p = p.next;
                count++;
            }
            //如果少于k个节点，则不需要翻转
            if (count < k) {
                break;
            }
            //k个节点后的那个节点
            ListNode last = p;
            //一个一个连过去
            p = dummy2.next;
            for (int i = 0;i < k;i++) {
                ListNode next = p.next;
                p.next = last;
                last = p;
                p = next;
            }
            //翻转后的结果
            dummy2.next = last;
            //当前的第k个数据就是先前的第一个数据
            dummy2 = start;
        }
        return dummy.next;
    }
}
```



***go beats 100%***

```
func reverseKGroup(head *ListNode, k int) *ListNode {
 dummy := &ListNode{-1, nil}
 dummy.Next = head
 dummy2 := dummy
 for {
  p := dummy2.Next
  start := p
  count := 0;
  for count < k && p != nil {
   p = p.Next
   count++
  }
  //如果少于k个节点，则不需要翻转
  if count < k {
   break
  }
  //k个节点后的那个节点
  last := p;
  //一个一个连过去
  p = dummy2.Next;
  for i := 0;i < k;i++ {
   next := p.Next
   p.Next = last
   last = p
   p = next
  }
  //翻转后的结果
  dummy2.Next = last
  //当前的第k个数据就是先前的第一个数据
  dummy2 = start
 }
 return dummy.Next
}
```



显然，这里我们的时间复杂度相较于递归没有变化，但是空间上我们得到了优化

## 总结

通常迭代来代替递归通常能够节省空间，但是迭代就会比较难写，我们要经常逼自己一把，因为面试的时候经常会有面试官让你同时写出两种方式的实现

}

31 合并K个排序链表

33 单词接龙