24 两两交换链表中的节点

更新时间：2019-09-06 09:46:10

> 不经一翻彻骨寒，怎得梅花扑鼻香。
>
> ——宋帆

刷题内容

> 难度: Medium
>
> 原题链接：https://leetcode-cn.com/problems/swap-nodes-in-pairs/

内容描述

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

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。

示例:

给定 1->2->3->4, 你应该返回 2->1->4->3.

题目详解

题目给我们一个链表，两两交换相邻的节点，并且是实际的交换，不仅仅是交换值，那么我们应该有几个问题产生：

1. 如果链表长度为0呢；

2. 链表长度为奇数该怎么办，最后一个落单的节点怎么处理呢？

解题方案

思路 1：时间复杂度: O(N) 空间复杂度: O(N)

我们想了想，不管那么多，先直接处理一下最基本的case，那么就是链表长度小于2的情况，我们都直接返回head就行了。然后其它情况呢？说明我们至少拥有两个节点了，我们直接交换前两个节点，再递归地去处理后面所有的节点即可。

因为递归深度为链表长度/2，所以空间复杂度为O(N)，时间复杂度同理。

Python beats 100%

```python
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        if not head or not head.next: # 链表长度小于2，我们都直接返回head就行
            return head
        nxt = head.next
        head.next = self.swapPairs(head.next.next) # 递归去处理后面所有的节点
        nxt.next = head # 交换前两个节点
        return nxt
```

c++ beats 100%

```cpp
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        // 链表长度小于2，我们都直接返回head就行
        if (head == NULL || head->next == NULL) {
            return head;
        }
        ListNode* next = head->next;
        // 递归处理后面所有的节点
        head->next = swapPairs(next->next);
        // 交换前两个节点
        next->next = head;
        return next;
    }
};
```

go beats 100%

```go
func swapPairs(head *ListNode) *ListNode {
    // 链表长度小于2，我们都直接返回head就行
    if head == nil || head.Next == nil {
  return head
 }
 next := head.Next
    // 递归处理后面所有的节点
 head.Next = swapPairs(next.Next)
    // 交换前两个节点
 next.Next = head
 return next
}
```

java beats 100%

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        // 链表长度小于2，我们都直接返回head就行
        if (head == null || head.next == null) {
            return head;
        }
        ListNode next = head.next;
        // 递归处理后面所有的节点
        head.next = swapPairs(next.next);
        // 交换前两个节点
        next.next = head;
        return next;
    }
}
```

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)

我们也可以用迭代的方法来做，这样可以将空间复杂度减小到O(1)。

Python

```
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        if not head or not head.next:
            return head

        cur = dummy = ListNode(-1)
        cur.next = head

        while cur.next and cur.next.next: # # 迭代所有的节点
            next_one, next_two, next_three = cur.next, cur.next.next, cur.next.next.next
            cur.next = next_two # 交换前两个节点
            next_two.next = next_one
            next_one.next = next_three
            cur = next_one
        return dummy.next
```

c++ beats 80.93 %

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        //开一个新的节点，这也是为什么这个方法比上面一个方法慢
        ListNode* dummy = new ListNode(-1);
        dummy->next = head;
        ListNode* p = dummy;
        while (1) {
         //迭代所有节点
            if (p->next == NULL || p->next->next == NULL) {
                break;
            }
            ListNode* first = p->next;
            ListNode* second = first->next;
            //交换前两个节点
            first->next = second->next;
            second->next = first;
            p->next = second;
            p = p->next->next;
        }
        return dummy->next;
    }
};
```

go

```
func swapPairs(head *ListNode) *ListNode {
 dummy := &ListNode{-1, nil}
 dummy.Next = head
 p := dummy
 for {
        //迭代所有节点
  if p.Next == nil || p.Next.Next == nil {
   break
  }
  first := p.Next
  second := first.Next
        //交换前两个节点
  first.Next = second.Next
  second.Next = first
  p.Next = second
  p = p.Next.Next
 }
 return dummy.Next
}
```

java beats 100%

```java
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode p = dummy;
        while (true) {
         //迭代所有节点
            if (p.next == null || p.next.next == null) {
                break;
            }
            ListNode first = p.next;
            ListNode second = first.next;
            //交换前两个节点
            first.next = second.next;
            second.next = first;
            p.next = second;
            p = p.next.next;
        }
        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。

小结

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。

}

23 括号生成

25 盛水最多的容器