题目地址(24. 两两交换链表中的节点)

https://leetcode-cn.com/problems/swap-nodes-in-pairs/

入选理由

1. 链表常规操作就是改变指针，这次其实就是两两反转再拼接，因此比昨天的题多了操作，那么你还会么？

2. 练习虚拟节点的使用，这个技巧在头结点可能会发生变化的时候都可以用。

标签

- 链表

前置知识

- 链表的基本知识

难度

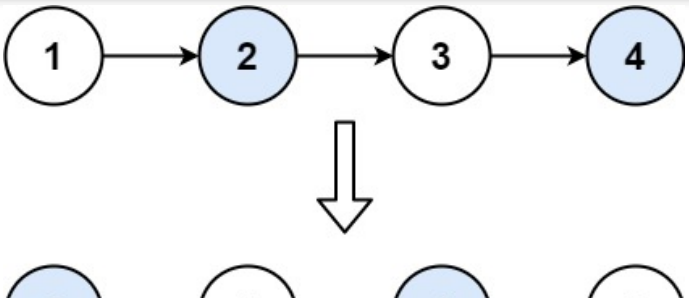
- 中等

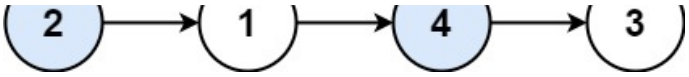
题目描述

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 1:





image

```
输入：head = [1,2,3,4]
输出：[2,1,4,3]
示例 2：

输入：head = []
输出：[]
示例 3：

输入：head = [1]
输出：[1]


提示：

链表中节点的数目在范围 [0, 100] 内
0 <= Node.val <= 100
```

迭代

思路

这道题其实考察的内容就是链表节点的指针的指向。

由于所有的两两交换逻辑都是一样的，因此我们只要关注某一个两两交换如何实现就可以了。

因为要修改的是二个一组的链表节点，所以需要操作 4 个节点。例如：将链表 A -> B 进行逆转，我们需要得到 A,B 以及 A 的前置节点 preA,以及 B 的后置节点 nextB

原始链表为 preA -> A -> B -> nextB，我们需要改为 preA -> B -> A -> nextB，接下来用同样的逻辑交换 nextB 以及 nextB 的下一个元素。

那么修改指针的顺序为：

1. A 节点的 next 指向 nextB:

> preA -> A -> nextB
> B -> nextB

2. B 节点的 next 指向 A

> preA -> A -> nextB
> B -> A

3. preA 节点的 next 指向 B

> preA -> B -> A -> nextB

伪代码：

```
A.next = next.B;
B.next = A;
preA.next = B;
```

我们可以创建一个空节点 preHead，让其 next 指针指向 A(充当 preA 的角色)，这样是我们专注于算法逻辑，避免判断边界条件。**这涉及到链表指针修改的时候头节点可能发生变化的时候非常常用。**

例如当前链表为： A -> B -> C -> D，使用虚拟节点后为 preHead -> A -> B -> C -> D

按照上诉步骤修改指针的 3 步后，链表为

> preHead -> B -> A -> C -> D

这时让 preHead 指向 A，继续上述步骤逆转 C -> D,循环此步骤直到整个链表被逆转

伪代码：

```
if 为空表或者只有一个节点{
  return head
}
let 前置指针 = new 链表节点
前置指针.next = head
第一个节点 = head
返回的结果 = 第一个节点
while(第一个节点存在 && 第一个节点.next不为空){
  第二个节点 = 第一个节点.next
  后置指针 = 第二个节点.next

  // 对链表进行逆转
  第一个节点.next = 后置指针
  第二个节点.next = 第一个节点
  前置指针.next = 第二个节点

  // 修改指针位置，进行下一轮逆转
  前置指针 = 第一个节点
  第一个节点 = 后置指针
}
return 返回的结果
```

代码

代码支持： JS，Java，Python3, C++

JS Code：

```
var swapPairs = function (head) {
  if (!head || !head.next) return head;
  let res = head.next;
  let now = head;
  let preNode = new ListNode();
  preNode.next = head;
  while (now && now.next) {
    let nextNode = now.next;
    let nnNode = nextNode.next;
    now.next = nnNode;
    nextNode.next = now;
    preNode.next = nextNode;
    preNode = now;
    now = nnNode;
  }
  return res;
};
```

Java Code：

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode preNode = new ListNode(-1, head), res;
        preNode.next = head;
        res = head.next;
        ListNode firstNode = head, secondNode, nextNode;
        while(firstNode != null && firstNode.next != null){
            secondNode = firstNode.next;
            nextNode = secondNode.next;

            firstNode.next = nextNode;
            secondNode.next = firstNode;
            preNode.next = secondNode;

            preNode = firstNode;
            firstNode = nextNode;
        }
        return res;
    }
}
```

Python3 Code:

```
if not head or not head.next: return head
    ans = ListNode()
    ans.next = head.next
    pre = ans
    while head and head.next:
        next = head.next
        n_next = next.next

        next.next = head
        pre.next = next
        head.next = n_next
        # 更新指针
        pre = head
        head = n_next
    return ans.next
```

C++ Code:

```
ListNode* swapPairs(ListNode* head) {
    if (head == nullptr || head->next == nullptr) return head;

    ListNode* dummy = new ListNode(-1, head);
    ListNode* prev = dummy;
    ListNode* cur = prev->next;

    while (cur != nullptr && cur->next != nullptr) {
        ListNode* next = cur->next;
        cur->next = next->next;
        next->next = cur;
        prev->next = next;

        prev = cur;
        cur = cur->next;
    }
    return dummy->next;
}
```

复杂度分析

- 时间复杂度：所有节点只遍历一遍，时间复杂度为$O(N)$
- 空间复杂度：未使用额外的空间，空间复杂度$O(1)$

递归

思路

1. 关注最小子结构，即将两个节点进行逆转。

2. 将逆转后的尾节点.next 指向下一次递归的返回值

3. 返回逆转后的链表头节点（ps:逆转前的第二个节点）

如果看不懂，建议看下我写的链表专题，里面详细讲述了这个技巧。

伪代码：

```
function run(head)
  if 为空表或者只有一个节点{
   return head
 }
 后一个节点 = head.next
 head.next = run(后一个节点.next)
 后一个节点.next = head
 return 后一个节点
}
```

代码

代码支持：JS，Java，Python，C++

JS Code:

```
var swapPairs = function (head) {
  if (!head || !head.next) return head;
  let nextNode = head.next;
  head.next = swapPairs(nextNode.next);
  nextNode.next = head;
  return nextNode;
};
```

Java Code:

```
class Solution {
   public ListNode swapPairs(ListNode head) {
       if(head == null || head.next == null) return head;
       ListNode nextNode = head.next;
       head.next = swapPairs(nextNode.next);
       nextNode.next = head;
       return nextNode;
   }
}
```

Python3 Code:

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

        next = head.next
        head.next = self.swapPairs(next.next)
        next.next = head

        return next
```

C++ Code:

```cpp
ListNode* swapPairs(ListNode* head) {
    if (head == nullptr || head->next == nullptr) return head;

    ListNode* first = head;
    ListNode* second = first->next;

    ListNode* head_of_next_group = swapPairs(second->next);

    first->next = head_of_next_group;
    second->next = first;

    return second;
}
```

复杂度分析

- 时间复杂度：所有节点只遍历一遍，时间复杂度为$O(N)$
- 空间复杂度：未使用额外的空间(递归造成的函数栈除外)，空间复杂度$O(1)$

上一页 下一页