判断一个链表是否为回文结构

【题目】

给定一个链表的头节点 head，请判断该链表是否为回文结构。

例如：

1->2->1，返回 true。

1->2->2->1，返回 true。

15->6->15，返回 true。

1->2->3，返回 false。

进阶：

如果链表长度为 N，时间复杂度达到 $O(N)$，额外空间复杂度达到 $O(1)$。

【难度】

普通解法 士 ★☆☆☆

进阶解法 尉 ★★☆☆

【解答】

方法一：

方法一是最容易实现的方法，利用栈结构即可。从左到右遍历链表，遍历的过程中把每个节点依次压入栈中。因为栈是先进后出的，所以在遍历完成后，从栈顶到栈底的节点值出现顺序会与原链表从左到右的值出现顺序反过来。那么，如果一个链表是回文结构，逆序之后，值出现的次序还是一样的，如果不是回文结构，顺序就肯定对不上。

例如：

链表 1->2->3->4，从左到右依次压栈之后，从栈顶到栈底的节点值顺序为 4，3，2，1。两者顺序对不上，所以这个链表不是回文结构。

链表 1->2->2->1，从左到右依次压栈之后，从栈顶到栈底的节点值顺序为 1，2，2，1。两者顺序一样，所以这个链表是回文结构。

方法一需要一个额外的栈结构，并且需要把所有的节点都压入栈中，所以这个额外的栈结构需要 $O(N)$的空间。具体过程请参看如下代码中的 isPalindrome1 方法。

```
public class Node {
        public int value;
        public Node next;

        public Node(int data) {
                this.value = data;
        }
}

public boolean isPalindrome1(Node head) {
        Stack<Node> stack = new Stack<Node>();
        Node cur = head;
        while (cur != null) {
                stack.push(cur);
                cur = cur.next;
        }
        while (head != null) {
                if (head.value != stack.pop().value) {
                        return false;
                }
                head = head.next;
        }
        return true;
}
```

方法二：

方法二对方法一进行了优化，虽然也是利用栈结构，但其实并不需要将所有的节点都压入栈中，只用压入一半的节点即可。首先假设链表的长度为 N，如果 N 是偶数，前 $N/2$ 的节点叫作左半区，后 $N/2$ 的节点叫作右半区。如果 N 是奇数，忽略处于最中间的节点，还是前 $N/2$ 的节点叫作左半区，后 $N/2$ 的节点叫作右半区。

例如：

链表 1->2->2->1，左半区为：1，2；右半区为：2，1。

链表 1->2->3->2->1，左半区为：1，2；右半区为：2，1。

方法二就是把整个链表的右半部分压入栈中，压入完成后，再检查栈顶到栈底值出现的顺序是否和链表左半部分的值相对应。

例如：

链表 1->2->2->1，链表的右半部分压入栈中后，从栈顶到栈底为 1，2。链表的左半部分也是 1，2。所以这个链表是回文结构。

链表 1->2->3->2->1，链表的右半部分压入栈中后，从栈顶到栈底为 1，2。链表的左半部分也是 1，2。所以这个链表是回文结构。

链表 1->2->3->3->1，链表的右半部分压入栈中后，从栈顶到栈底为 1，3。链表的左半

部分也是 1，2。所以这个链表不是回文结构。

方法二可以直观地理解为将链表的右半部分“折过去”，然后让它和左半部分比较，如图 2-1 所示。

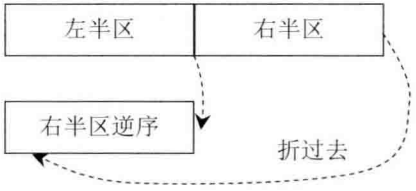

图 2-1

方法二的具体过程请参看如下代码中的 isPalindrome2 方法。

```
public boolean isPalindrome2(Node head) {
        if (head == null || head.next == null) {
                return true;
        }
        Node right = head.next;
        Node cur = head;
        while (cur.next != null && cur.next.next != null) {
                right = right.next;
                cur = cur.next.next;
        }
        Stack<Node> stack = new Stack<Node>();
        while (right != null) {
                stack.push(right);
                right = right.next;
        }
        while (!stack.isEmpty()) {
                if (head.value != stack.pop().value) {
                        return false;
                }
                head = head.next;
        }
        return true;
}
```

方法三：

方法三不需要栈和其他数据结构，只用有限几个变量，其额外空间复杂度为 $O(1)$，就可以在时间复杂度为 $O(N)$内完成所有的过程，也就是满足进阶的要求。具体过程如下：

1．首先改变链表右半区的结构，使整个右半区反转，最后指向中间节点。

例如：

链表 1->2->3->2->1，通过这一步将其调整之后的结构如图 2-2 所示。

链表 1->2->3->3->2->1，将其调整之后的结构如图 2-3 所示。

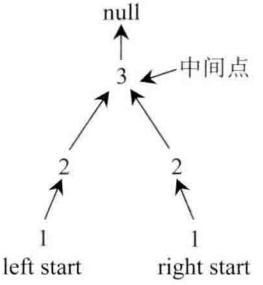

图 2-2

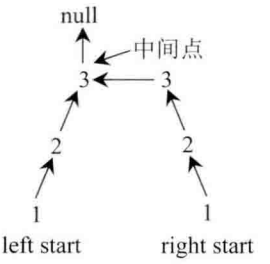

图 2-3

我们将左半区的第一个节点（也就是原链表的头节点）记为 leftStart，右半区反转之后最右边的节点（也就是原链表的最后一个节点）记为 rightStart。

2．leftStart 和 rightStart 同时向中间点移动，移动每一步都比较 leftStart 和 rightStart 节点的值，看是否一样。如果都一样，说明链表为回文结构，否则不是回文结构。

3．不管最后返回的是 true 还是 false，在返回前都应该把链表恢复成原来的样子。

4．链表恢复成原来的结构之后，返回检查结果。

粗看起来，虽然方法三的整个过程也没有多少难度，但要想用有限几个变量完成以上所有的操作，在实现上还是比较考查代码实现能力的。方法三的全部过程请参看如下代码中的 isPalindrome3 方法，该方法只申请了三个 Node 类型的变量。

```
public boolean isPalindrome3(Node head) {
        if (head == null || head.next == null) {
                return true;
        }
        Node n1 = head;
        Node n2 = head;
        while (n2.next != null && n2.next.next != null) { // 查找中间节点
                n1 = n1.next; // n1 -> 中部
                n2 = n2.next.next; // n2 -> 结尾
        }
        n2 = n1.next; // n2 -> 右部分第一个节点
        n1.next = null; // mid.next -> null
        Node n3 = null;
        while (n2 != null) { // 右半区反转
                n3 = n2.next; // n3 -> 保存下一个节点
                n2.next = n1; // 下一个反转节点
                n1 = n2; // n1 移动
                n2 = n3; // n2 移动
        }
```

```
        n3 = n1; // n3 -> 保存最后一个节点
        n2 = head;// n2 -> 左边第一个节点
        boolean res = true;
        while (n1 != null && n2 != null) { // 检查回文
                if (n1.value != n2.value) {
                        res = false;
                        break;
                }
                n1 = n1.next; // 从左到中部
                n2 = n2.next; // 从右到中部
        }
        n1 = n3.next;
        n3.next = null;
        while (n1 != null) { // 恢复列表
                n2 = n1.next;
                n1.next = n3;
                n3 = n1;
                n1 = n2;
        }
        return res;
}
```

将单向链表按某值划分成左边小、中间相等、右边大的形式

【题目】

给定一个单向链表的头节点 head，节点的值类型是整型，再给定一个整数 pivot。实现一个调整链表的函数，将链表调整为左部分都是值小于 pivot 的节点，中间部分都是值等于 pivot 的节点，右部分都是值大于 pivot 的节点。除这个要求外，对调整后的节点顺序没有更多的要求。

例如：链表 9->0->4->5->1，pivot=3。

调整后链表可以是 1->0->4->9->5，也可以是 0->1->9->5->4。总之，满足左部分都是小于 3 的节点，中间部分都是等于 3 的节点（本例中这个部分为空），右部分都是大于 3 的节点即可。对某部分内部的节点顺序不做要求。

进阶：

在原问题的要求之上再增加如下两个要求。

- 在左、中、右三个部分的内部也做顺序要求，要求每部分里的节点从左到右的顺序与原链表中节点的先后次序一致。

例如：链表 9->0->4->5->1，pivot=3。调整后的链表是 0->1->9->4->5。在满足原问题要求的同时，左部分节点从左到右为 0、1。在原链表中也是先出现 0，后出现 1；中间部