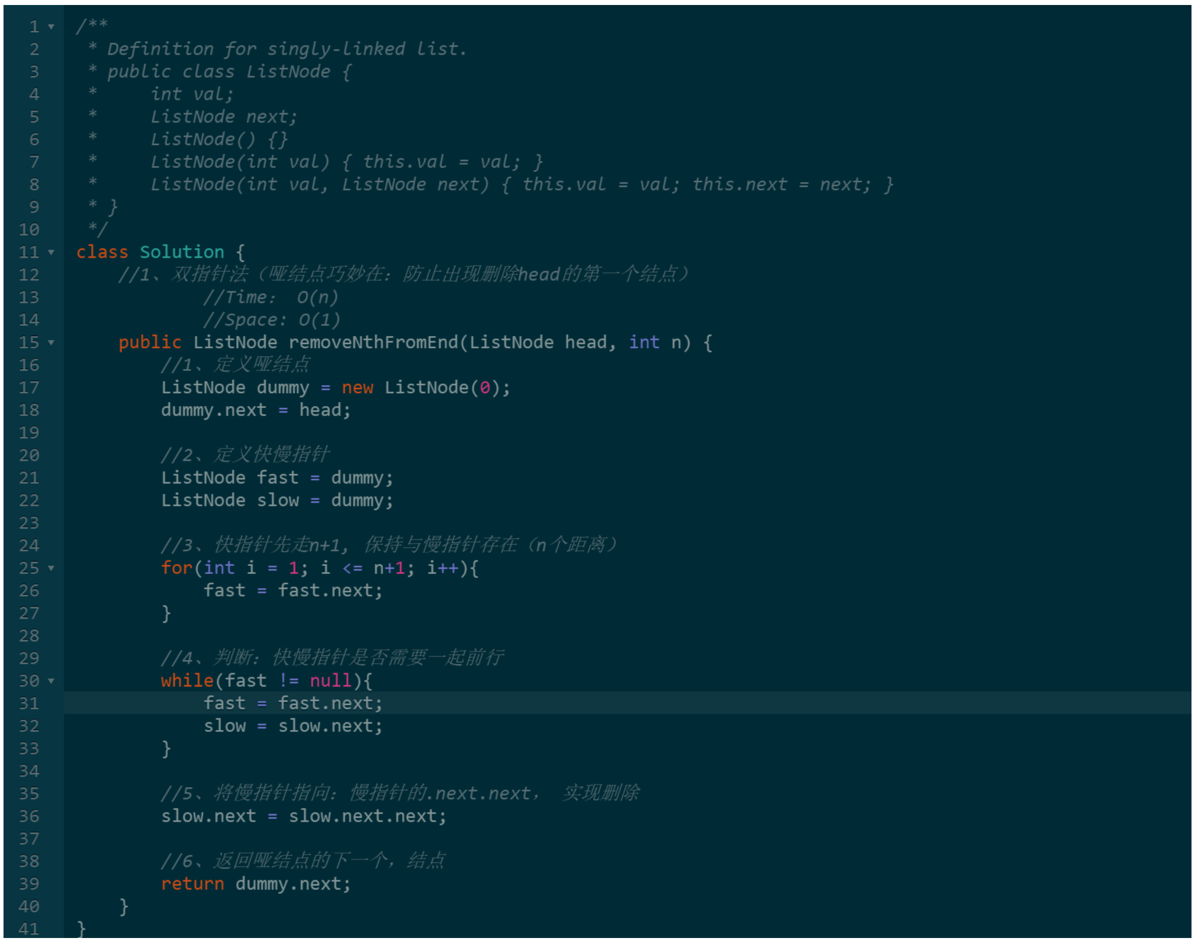
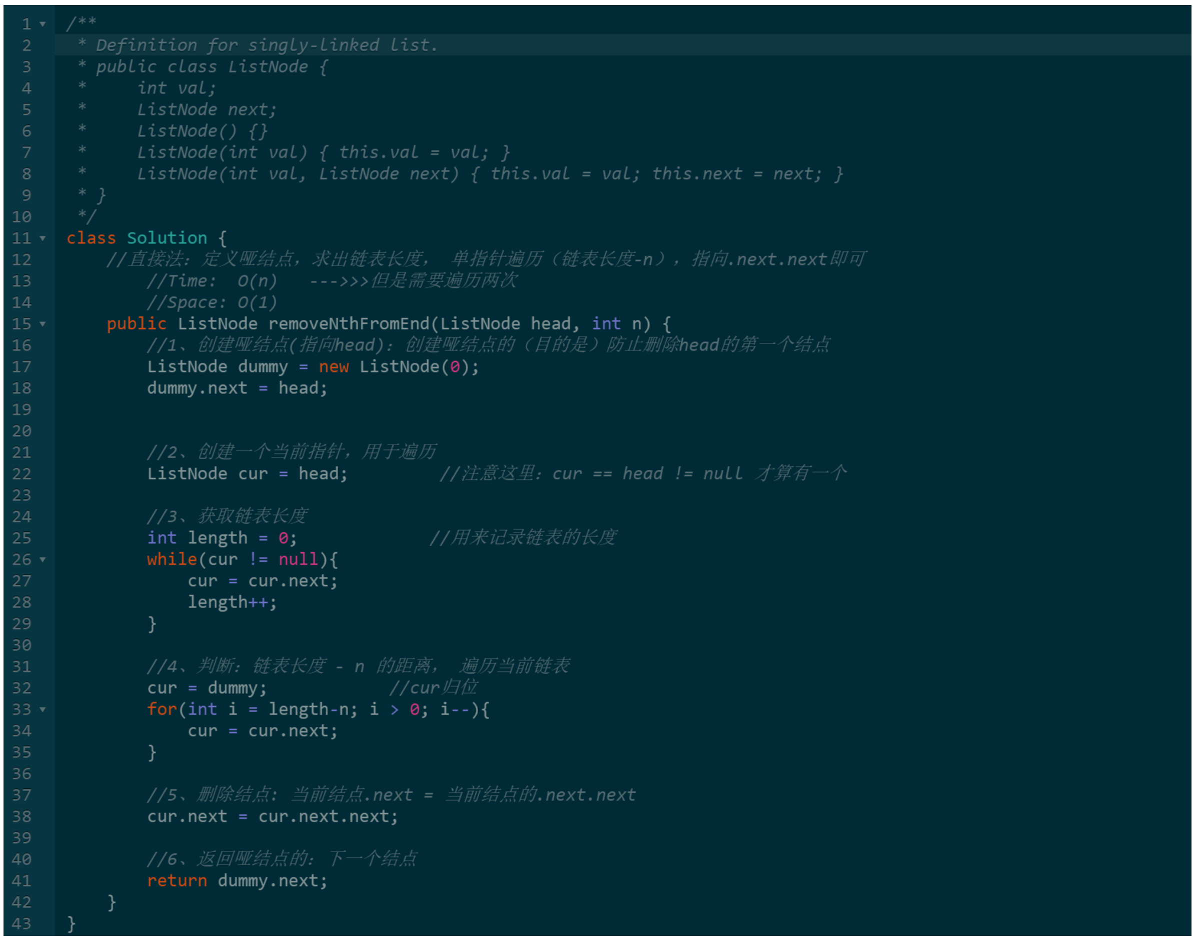
19、Remove Nth Node From End of List

①、（双指针法：哑结点）



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //1、双指针法（哑结点巧妙在：防止出现删除head的第一个结点）
            //Time:  O(n)
            //Space: O(1)
    public ListNode removeNthFromEnd(ListNode head, int n) {
        //1、定义哑结点
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        //2、定义快慢指针
        ListNode fast = dummy;
        ListNode slow = dummy;

        //3、快指针先走n+1，保持与慢指针存在（n个距离）
        for(int i = 1; i <= n+1; i++){
            fast = fast.next;
        }

        //4、判断：快慢指针是否需要一起前行
        while(fast != null){
            fast = fast.next;
            slow = slow.next;
        }

        //5、将慢指针指向：慢指针的.next.next，  实现删除
        slow.next = slow.next.next;

        //6、返回哑结点的下一个，结点
        return dummy.next;
    }
}
```

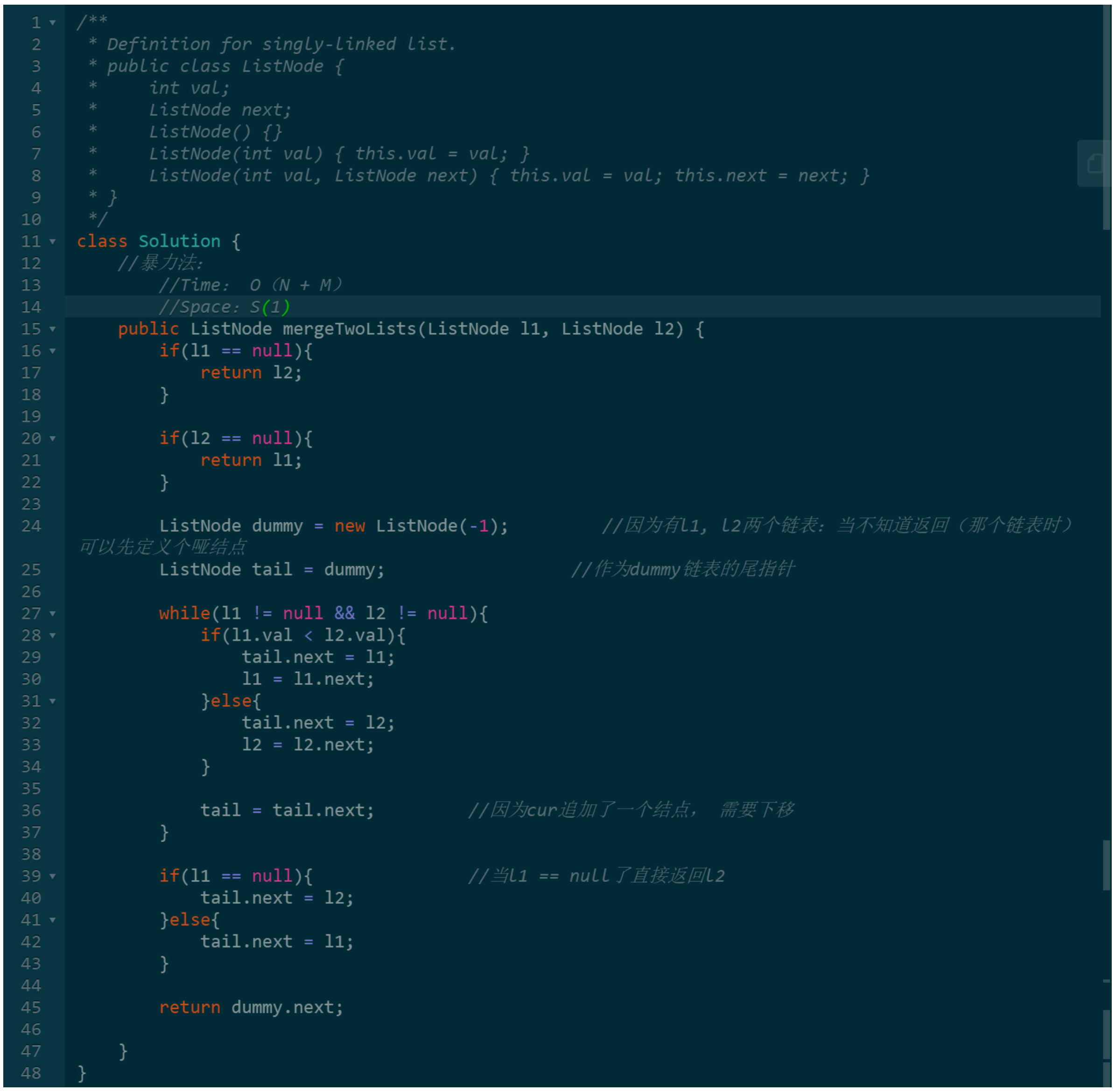
②、直接法（但是需要遍历两次）



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //直接法：定义哑结点，求出链表长度， 单指针遍历（链表长度-n），指向.next.next即可
        //Time:  O(n)   --->>>但是需要遍历两次
        //Space: O(1)
    public ListNode removeNthFromEnd(ListNode head, int n) {
        //1、创建哑结点(指向head)：创建哑结点的（目的是）防止删除head的第一个结点
        ListNode dummy = new ListNode(0);
        dummy.next = head;


        //2、创建一个当前指针，用于遍历
        ListNode cur = head;          //注意这里：cur == head != null 才算有一个

        //3、获取链表长度
        int length = 0;               //用来记录链表的长度
        while(cur != null){
            cur = cur.next;
            length++;
        }

        //4、判断：链表长度 - n 的距离， 遍历当前链表
        cur = dummy;              //cur归位
        for(int i = length-n; i > 0; i--){
            cur = cur.next;
        }

        //5、删除结点： 当前结点.next = 当前结点的.next.next
        cur.next = cur.next.next;

        //6、返回哑结点的: 下一个结点
        return dummy.next;
    }
}
```

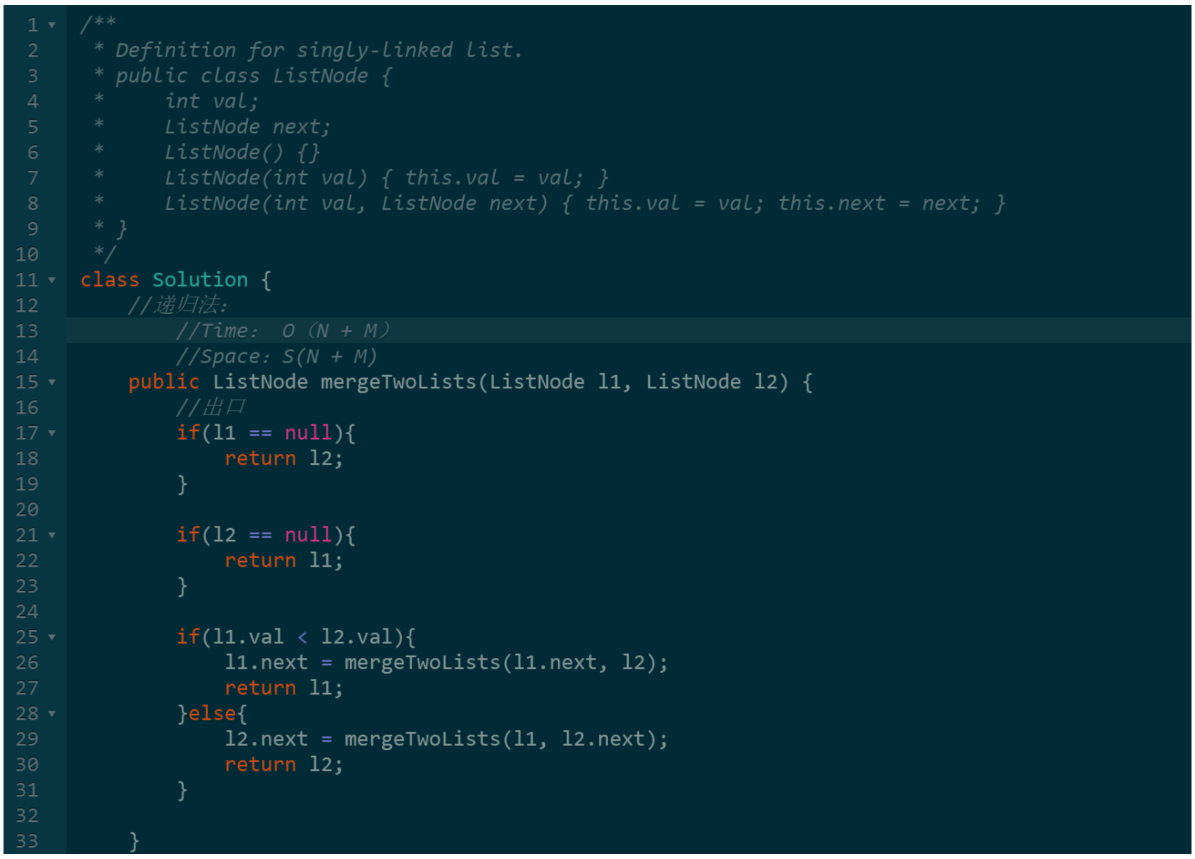
21、Merge Two Sorted Lists

①、暴力法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //暴力法:
        //Time:  O（N + M）
        //Space: S(1)
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null){
            return l2;
        }

        if(l2 == null){
            return l1;
        }

        ListNode dummy = new ListNode(-1);         //因为有l1, l2两个链表: 当不知道返回（那个链表时）可以先定义个哑结点
        ListNode tail = dummy;                     //作为dummy链表的尾指针

        while(l1 != null && l2 != null){
            if(l1.val < l2.val){
                tail.next = l1;
                l1 = l1.next;
            }else{
                tail.next = l2;
                l2 = l2.next;
            }

            tail = tail.next;         //因为cur追加了一个结点,  需要下移
        }

        if(l1 == null){               //当l1 == null 了直接返回l2
            tail.next = l2;
        }else{
            tail.next = l1;
        }

        return dummy.next;

    }
}
```

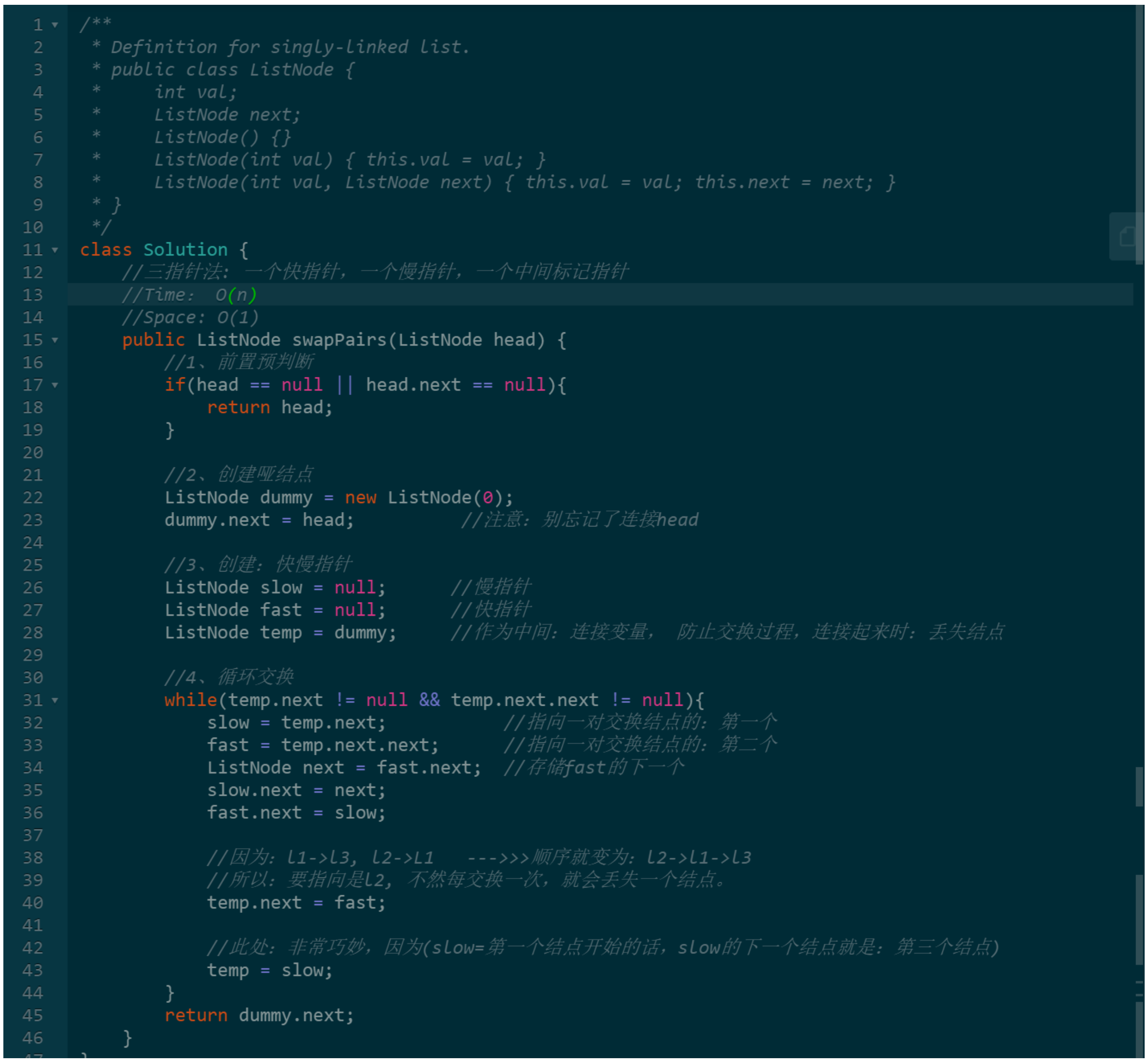
②、递归法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //递归法:
        //Time:  O（N + M）
        //Space: S(N + M)
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        //出口
        if(l1 == null){
            return l2;
        }

        if(l2 == null){
            return l1;
        }

        if(l1.val < l2.val){
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        }else{
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }

    }
```

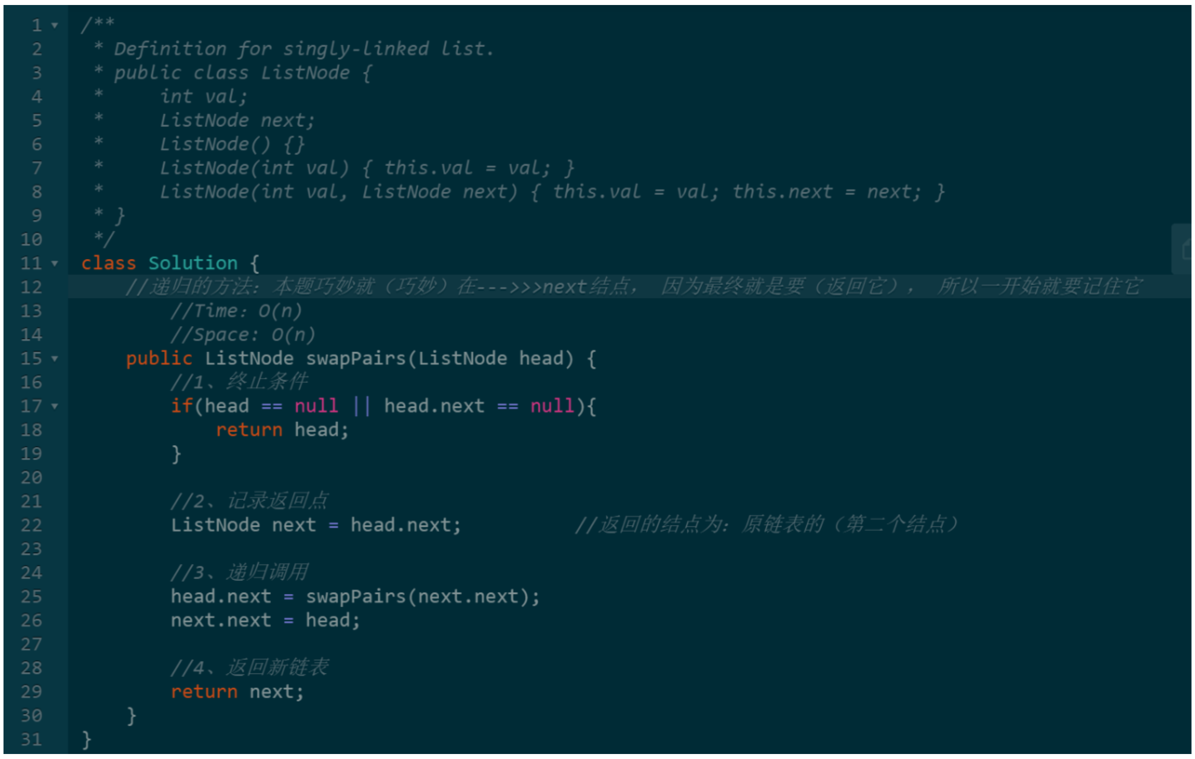
24、Swap Nodes in Pairs (Medium)

①、三指针法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //三指针法： 一个快指针， 一个慢指针， 一个中间标记指针
    //Time:  O(n)
    //Space: O(1)
    public ListNode swapPairs(ListNode head) {
        //1、前置预判断
        if(head == null || head.next == null){
            return head;
        }

        //2、创建哑结点
        ListNode dummy = new ListNode(0);
        dummy.next = head;          //注意: 别忘记了连接head

        //3、创建: 快慢指针
        ListNode slow = null;       //慢指针
        ListNode fast = null;       //快指针
        ListNode temp = dummy;      //作为中间: 连接变量， 防止交换过程，连接起来时: 丢失结点

        //4、循环交换
        while(temp.next != null && temp.next.next != null){
            slow = temp.next;           //指向一对交换结点的: 第一个
            fast = temp.next.next;      //指向一对交换结点的: 第二个
            ListNode next = fast.next;  //存储fast的下一个
            slow.next = next;
            fast.next = slow;

            //因为: l1->l3, l2->L1   --->>>顺序就变为: l2->l1->l3
            //所以: 要指向是l2, 不然每交换一次，就会丢失一个结点。
            temp.next = fast;

            //此处: 非常巧妙，因为(slow=第一个结点开始的话，slow的下一个结点就是: 第三个结点)
            temp = slow;
        }
        return dummy.next;
    }
```

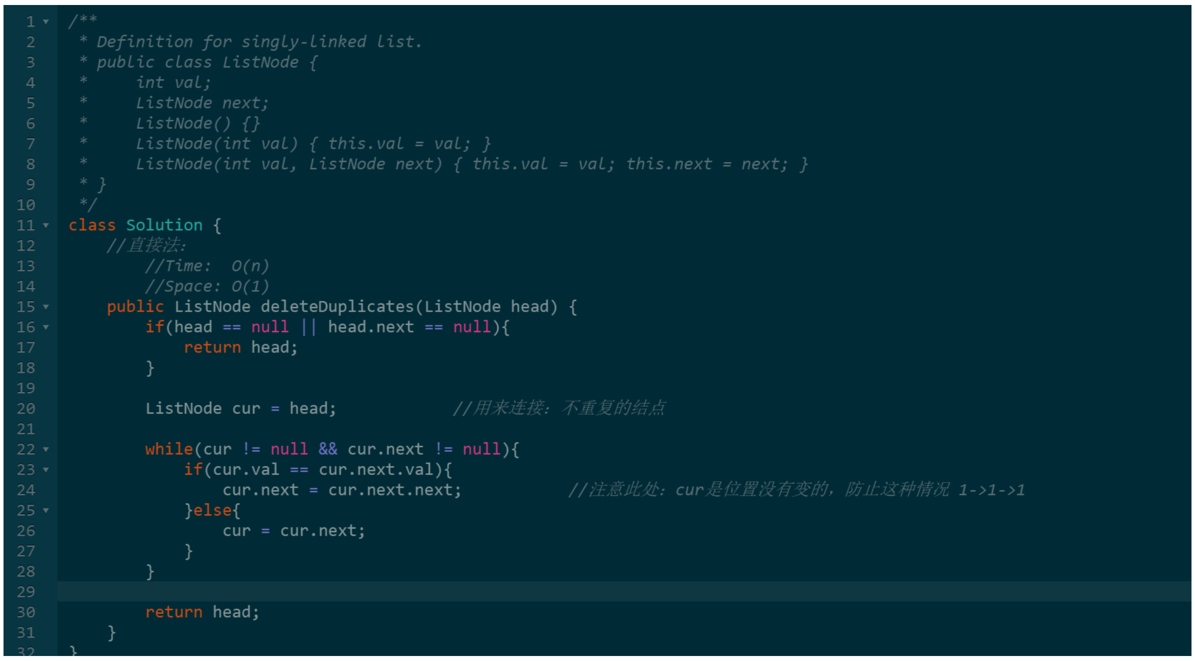
②、递归法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //递归的方法: 本题巧妙就（巧妙）在--->>>next结点， 因为最终就是要（返回它）， 所以一开始就要记住它
        //Time: O(n)
        //Space: O(n)
    public ListNode swapPairs(ListNode head) {
        //1、终止条件
        if(head == null || head.next == null){
            return head;
        }

        //2、记录返回点
        ListNode next = head.next;          //返回的结点为: 原链表的（第二个结点）

        //3、递归调用
        head.next = swapPairs(next.next);
        next.next = head;

        //4、返回新链表
        return next;
    }
}
```

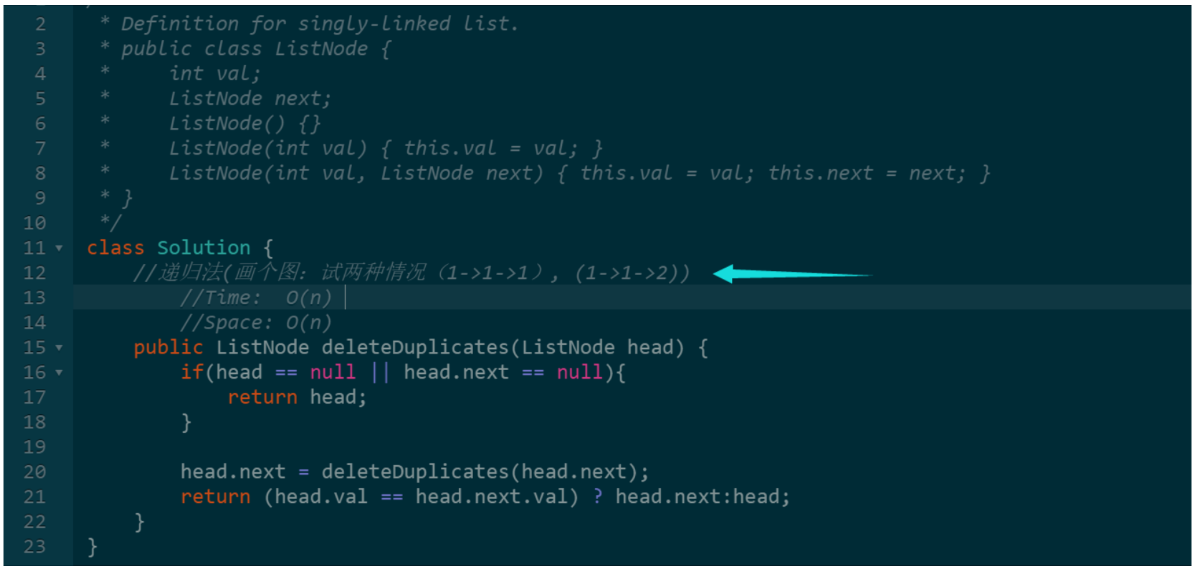
83、 Remove Duplicates from Sorted List

①、直接法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //直接法:
        //Time:  O(n)
        //Space: O(1)
    public ListNode deleteDuplicates(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }

        ListNode cur = head;            //用来连接: 不重复的结点

        while(cur != null && cur.next != null){
            if(cur.val == cur.next.val){
                cur.next = cur.next.next;           //注意此处: cur是位置没有变的, 防止这种情况 1->1->1
            }else{
                cur = cur.next;
            }
        }

        return head;
    }
}
```

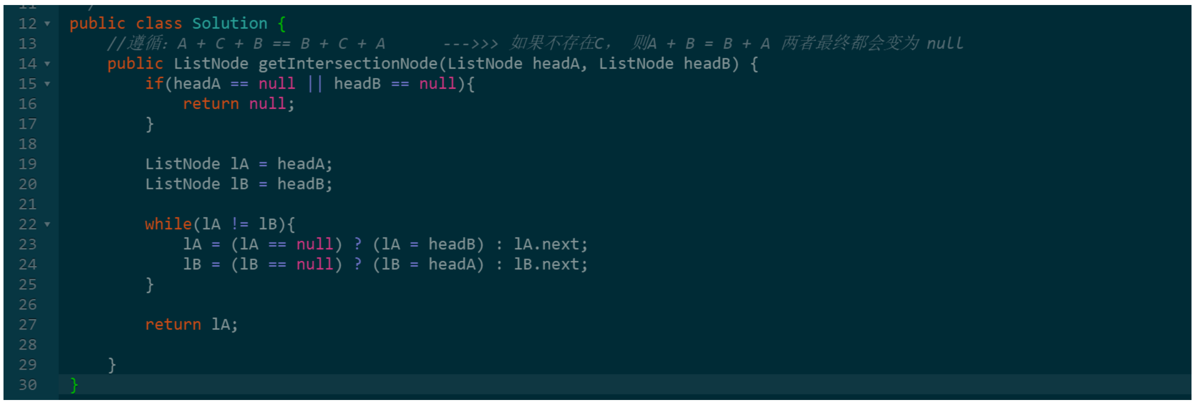
②、递归法



```java
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //递归法(画个图: 试两种情况（1->1->1）, (1->1->2))
        //Time:  O(n)
        //Space: O(n)
    public ListNode deleteDuplicates(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }

        head.next = deleteDuplicates(head.next);
        return (head.val == head.next.val) ? head.next:head;
    }
}
```

160、Intersection of Two Linked Lists

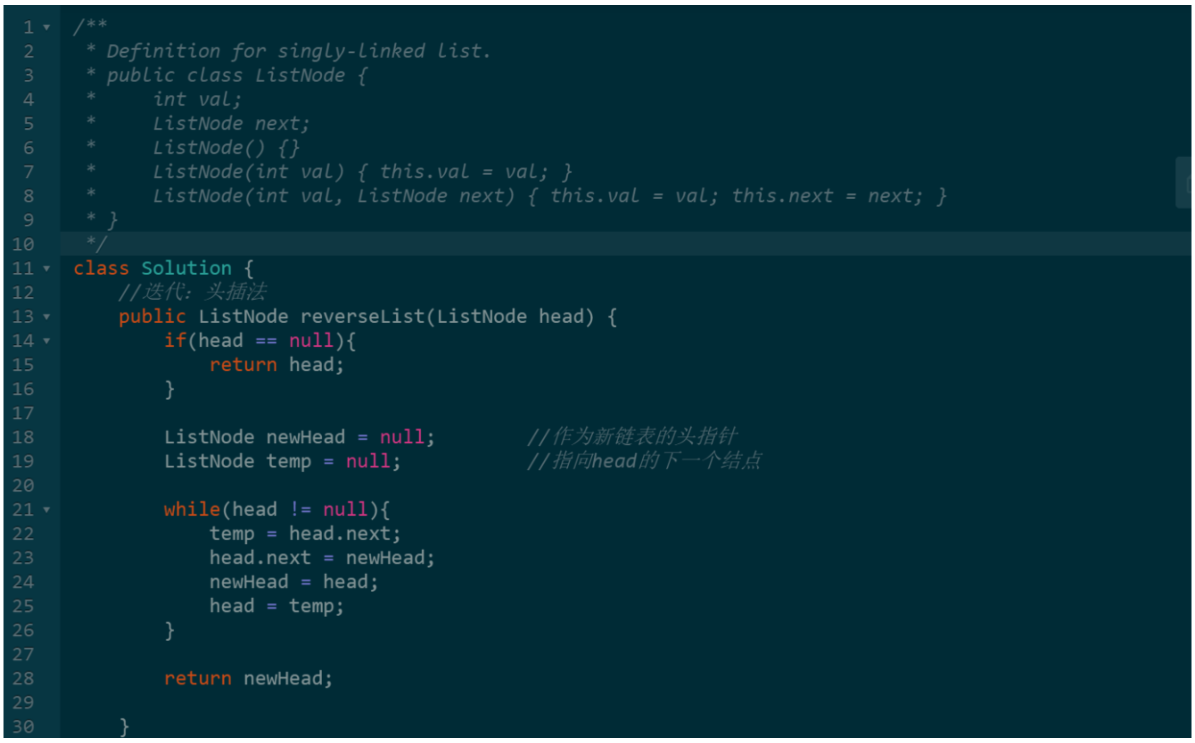
①、GitHub双指针法（完美）



```java
public class Solution {
    //遵循: A + C + B == B + C + A      --->>> 如果不存在C,  则A + B = B + A 两者最终都会变为 null
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA == null || headB == null){
            return null;
        }

        ListNode lA = headA;
        ListNode lB = headB;

        while(lA != lB){
            lA = (lA == null) ? (lA = headB) : lA.next;
            lB = (lB == null) ? (lB = headA) : lB.next;
        }

        return lA;

    }
}
```

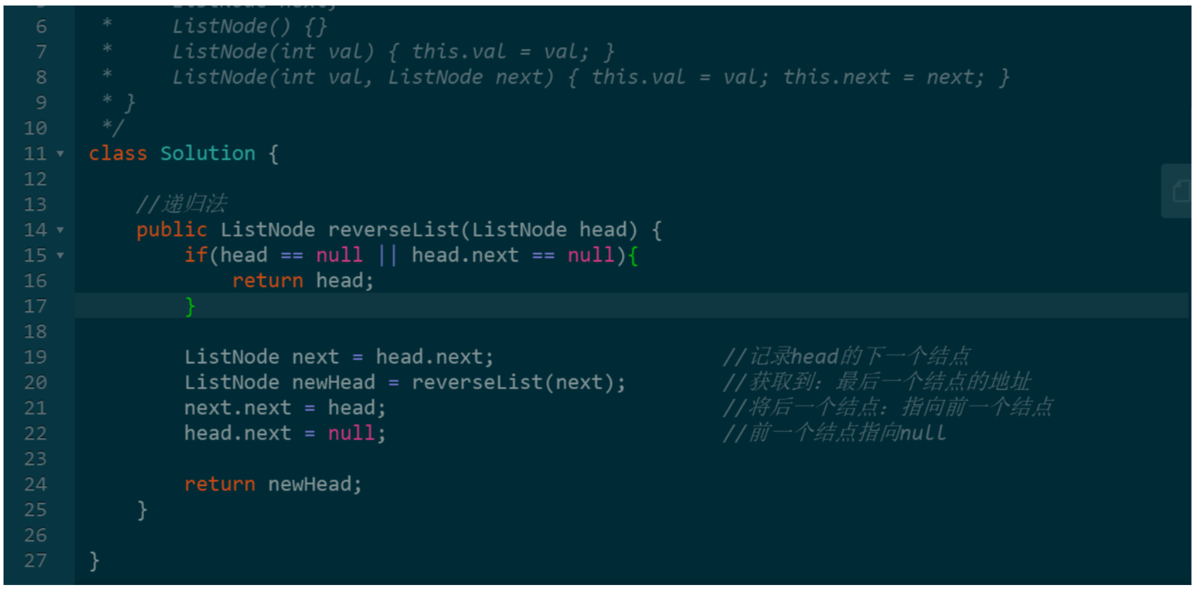
206、Reverse Linked List (Easy)

①、迭代法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //迭代: 头插法
    public ListNode reverseList(ListNode head) {
        if(head == null){
            return head;
        }

        ListNode newHead = null;        //作为新链表的头指针
        ListNode temp = null;           //指向head的下一个结点

        while(head != null){
            temp = head.next;
            head.next = newHead;
            newHead = head;
            head = temp;
        }

        return newHead;

    }
```

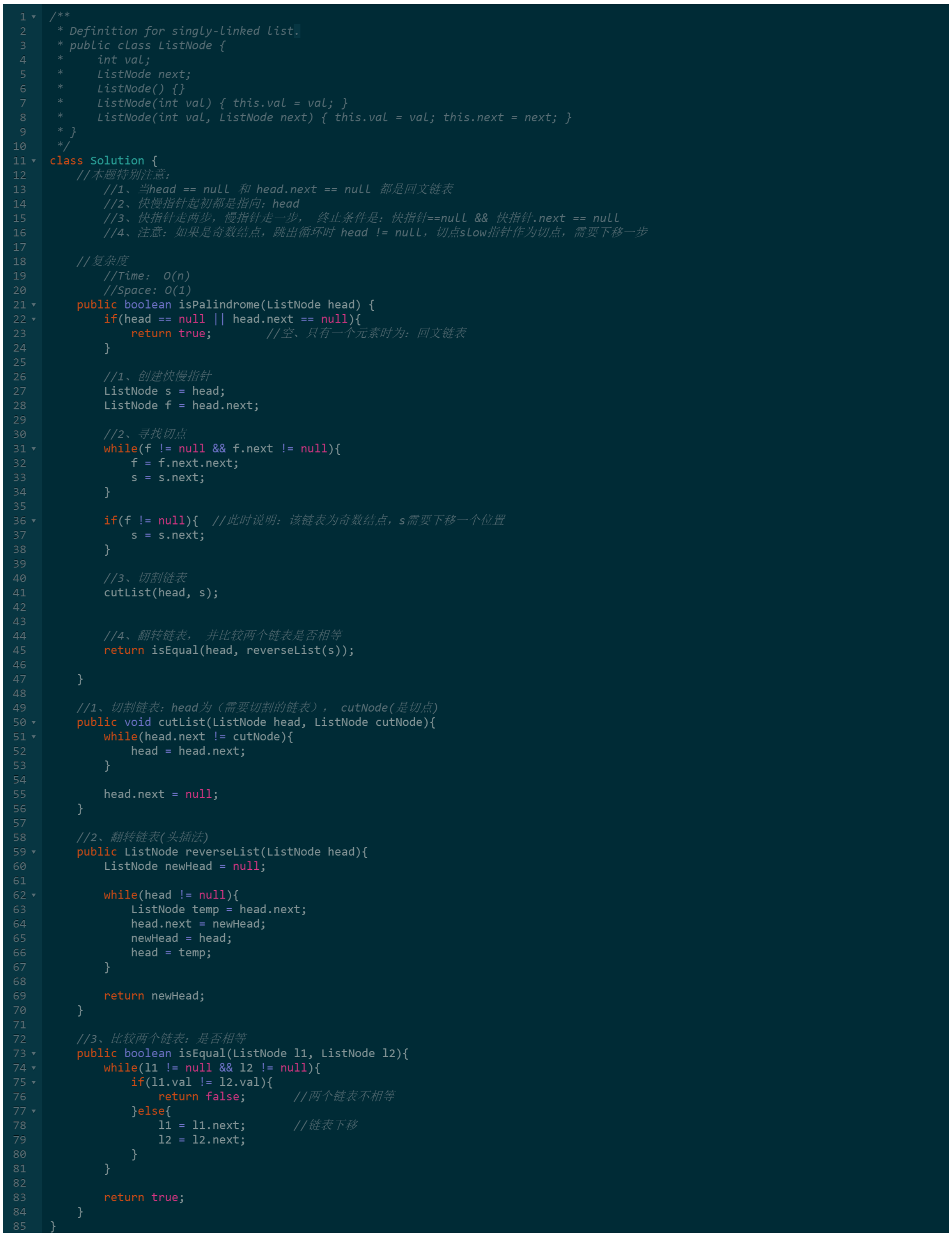
②、递归法



```java
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {

    //递归法
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }

        ListNode next = head.next;                  //记录head的下一个结点
        ListNode newHead = reverseList(next);       //获取到：最后一个结点的地址
        next.next = head;                           //将后一个结点：指向前一个结点
        head.next = null;                           //前一个结点指向null

        return newHead;
    }

}
```

234.、Palindrome Linked List

①、快慢指针法



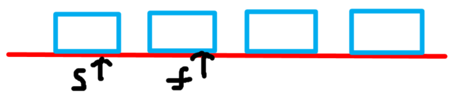
```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //本题特别注意:
        //1、当head == null 和 head.next == null 都是回文链表
        //2、快慢指针起初都是指向: head
        //3、快指针走两步, 慢指针走一步, 终止条件是: 快指针==null && 快指针.next == null
        //4、注意: 如果是奇数结点, 跳出循环时 head != null, 切点slow指针作为切点, 需要下移一步

    //复杂度
        //Time:  O(n)
        //Space: O(1)
    public boolean isPalindrome(ListNode head) {
        if(head == null || head.next == null){
            return true;        //空、只有一个元素时为: 回文链表
        }

        //1、创建快慢指针
        ListNode s = head;
        ListNode f = head.next;

        //2、寻找切点
        while(f != null && f.next != null){
            f = f.next.next;
            s = s.next;
        }

        if(f != null){  //此时说明: 该链表为奇数结点, s需要下移一个位置
            s = s.next;
        }

        //3、切割链表
        cutList(head, s);


        //4、翻转链表,  并比较两个链表是否相等
        return isEqual(head, reverseList(s));

    }

    //1、切割链表: head为(需要切割的链表),  cutNode(是切点)
    public void cutList(ListNode head, ListNode cutNode){
        while(head.next != cutNode){
            head = head.next;
        }

        head.next = null;
    }

    //2、翻转链表(头插法)
    public ListNode reverseList(ListNode head){
        ListNode newHead = null;

        while(head != null){
            ListNode temp = head.next;
            head.next = newHead;
            newHead = head;
            head = temp;
        }

        return newHead;
    }

    //3、比较两个链表: 是否相等
    public boolean isEqual(ListNode l1, ListNode l2){
        while(l1 != null && l2 != null){
            if(l1.val != l2.val){
                return false;       //两个链表不相等
            }else{
                l1 = l1.next;       //链表下移
                l2 = l2.next;
            }
        }

        return true;
    }
}
```

题目要求：判断 2->1->1->2 是不是回文链表？　　　要求：空间复杂度O(1)

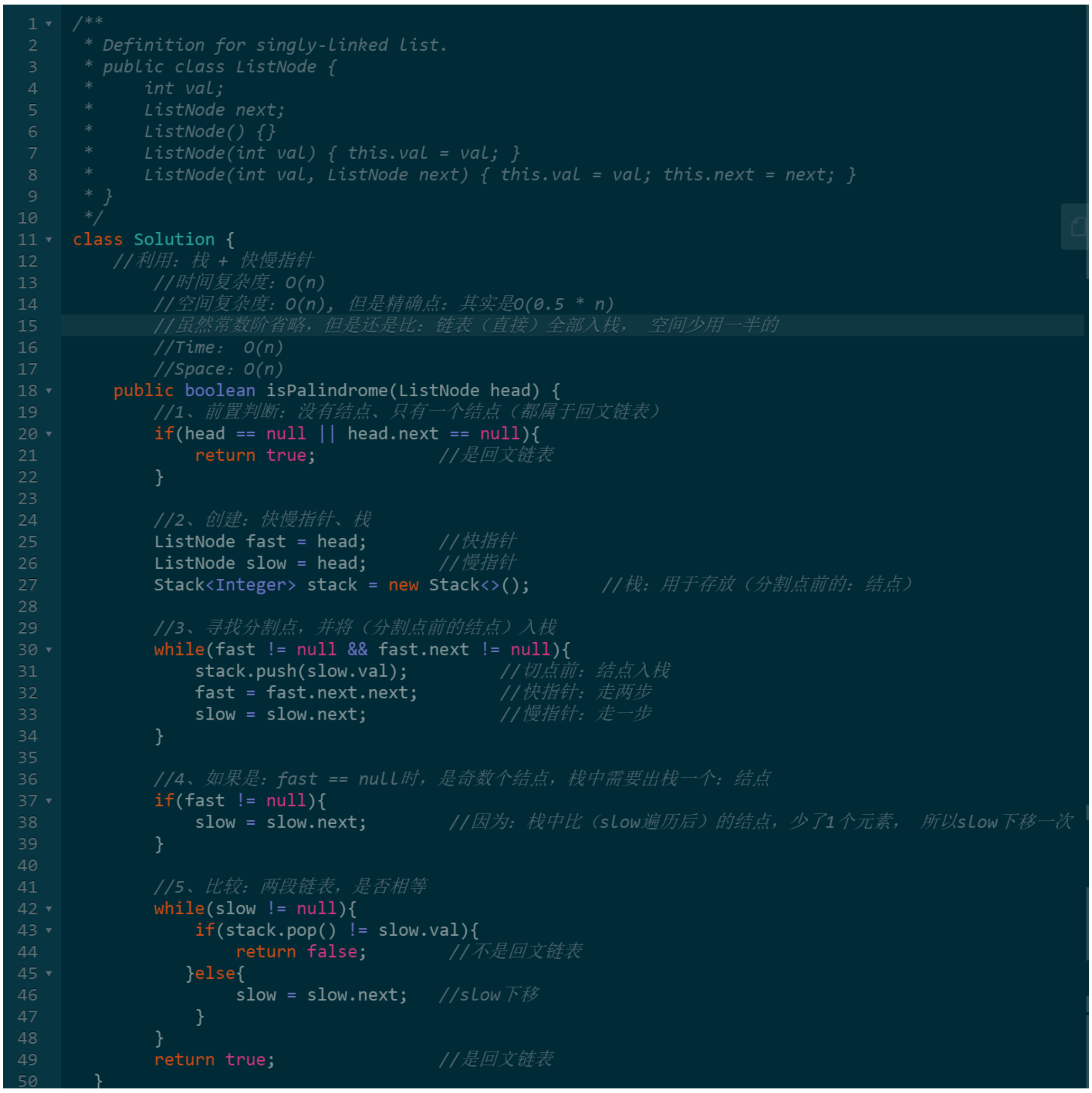
解决方法： 用快、慢指针

方法切割
- public boolean isPalindrome(ListNode head)　　判断：回文主入口
- private void cutLink(ListNode head, ListNode cutNode)　　切割：链表
- private ListNode reverseLink(ListNode head)　　翻转：链表
- private boolean isEqual(ListNode l1, ListNode l2) 判断：分割后的链表，值是否相同

注意要点
- 1、切点为：slow.next = cutNode　　其中：slow = head, fast = head.next; slow走一步, fast走两步 ,遍历条件为 fast != null && fast.next != null
- 2、（奇数个）结点时，slow不用下移；（偶数个）结点时，slow需要下移 原因是：奇数.next == cutNode了
- 3、翻转链表时，使用的是：头插法

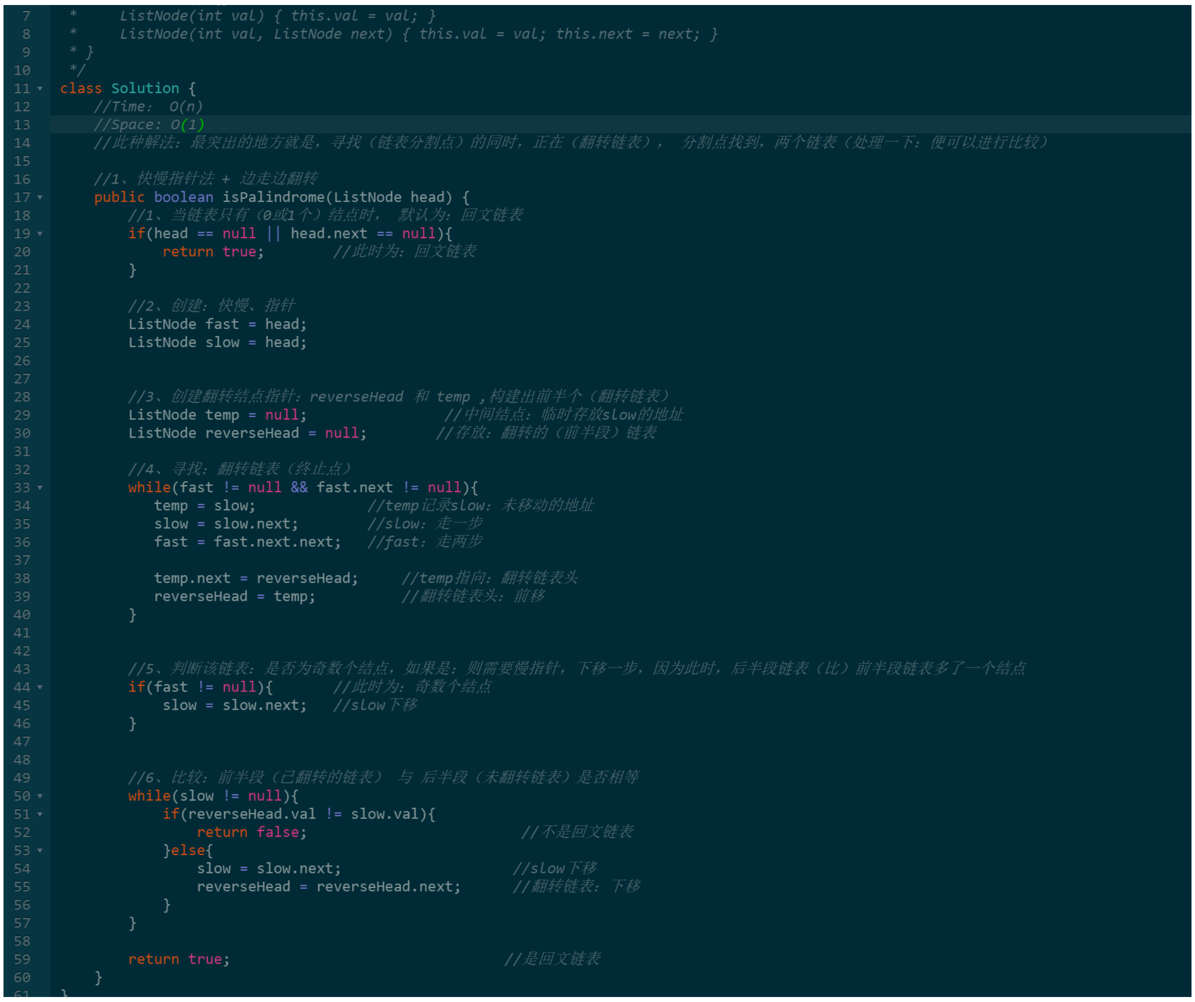


②、栈+快慢指针法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //利用：栈 + 快慢指针
        //时间复杂度：O(n)
        //空间复杂度：O(n)，但是精确点：其实是O(0.5 * n)
        //虽然常数阶省略，但是还是比：链表（直接）全部入栈， 空间少用一半的
        //Time:  O(n)
        //Space: O(n)
    public boolean isPalindrome(ListNode head) {
        //1、前置判断：没有结点、只有一个结点（都属于回文链表）
        if(head == null || head.next == null){
            return true;            //是回文链表
        }

        //2、创建：快慢指针、栈
        ListNode fast = head;       //快指针
        ListNode slow = head;       //慢指针
        Stack<Integer> stack = new Stack<>();       //栈：用于存放（分割点前的：结点）

        //3、寻找分割点，并将（分割点前的结点）入栈
        while(fast != null && fast.next != null){
            stack.push(slow.val);         //切点前：结点入栈
            fast = fast.next.next;        //快指针：走两步
            slow = slow.next;             //慢指针：走一步
        }

        //4、如果是：fast == null时，是奇数个结点，栈中需要出栈一个：结点
        if(fast != null){
            slow = slow.next;        //因为：栈中比（slow遍历后）的结点，少了1个元素，  所以slow下移一次
        }

        //5、比较：两段链表，是否相等
        while(slow != null){
            if(stack.pop() != slow.val){
                return false;          //不是回文链表
            }else{
                slow = slow.next;   //slow下移
            }
        }
        return true;                //是回文链表
    }
```

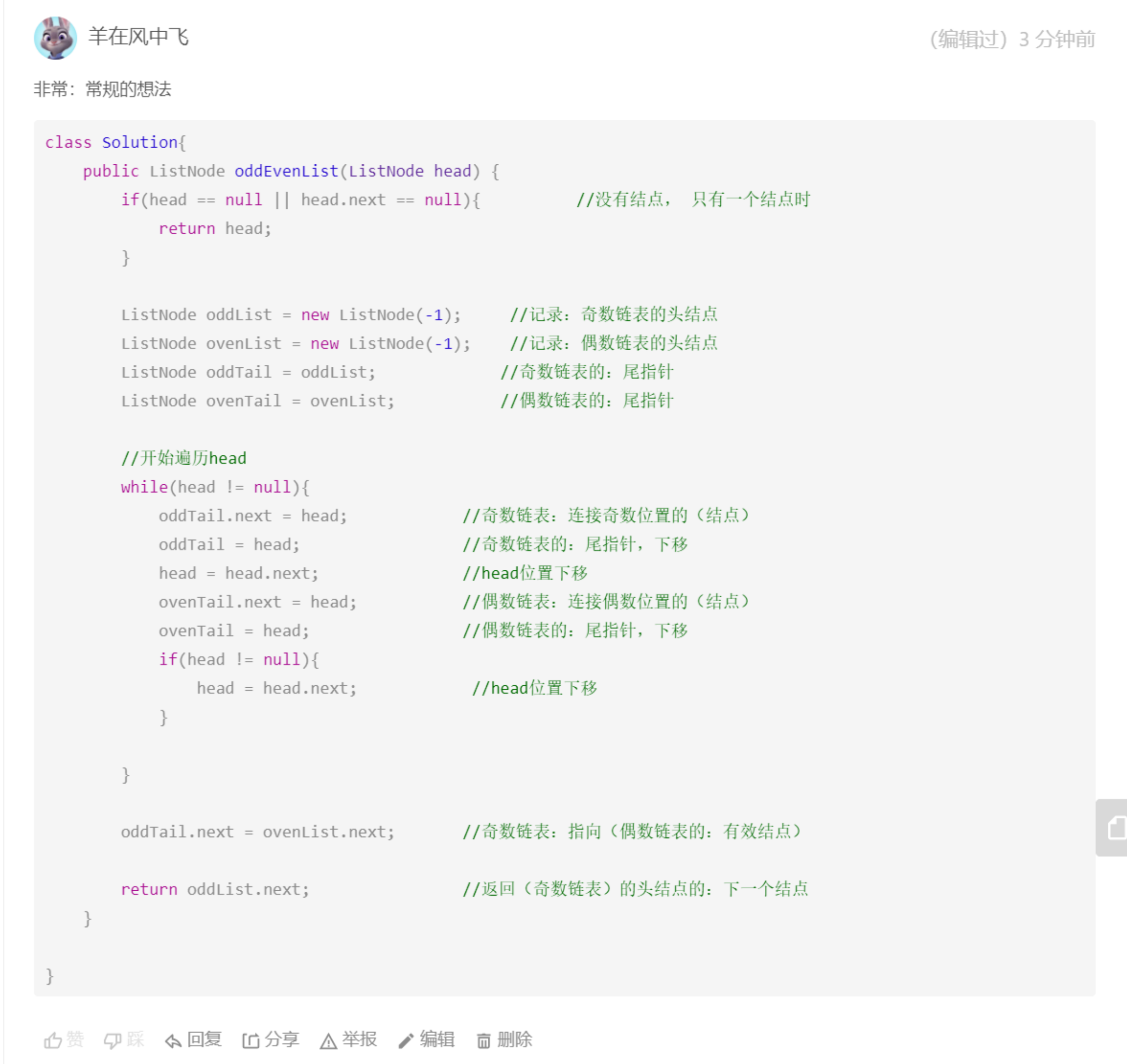
③、快慢指针+边走边翻转



```
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //Time:  O(n)
    //Space: O(1)
    //此种解法：最突出的地方就是，寻找（链表分割点）的同时，正在（翻转链表），  分割点找到，两个链表（处理一下：便可以进行比较）

    //1、快慢指针法 + 边走边翻转
    public boolean isPalindrome(ListNode head) {
        //1、当链表只有（0或1个）结点时，  默认为：回文链表
        if(head == null || head.next == null){
            return true;        //此时为：回文链表
        }

        //2、创建：快慢、指针
        ListNode fast = head;
        ListNode slow = head;


        //3、创建翻转结点指针：reverseHead 和 temp ,构建出前半个（翻转链表）
        ListNode temp = null;                //中间结点：临时存放slow的地址
        ListNode reverseHead = null;        //存放：翻转的（前半段）链表

        //4、寻找：翻转链表（终止点）
        while(fast != null && fast.next != null){
           temp = slow;             //temp记录slow：未移动的地址
           slow = slow.next;        //slow：走一步
           fast = fast.next.next;   //fast：走两步

           temp.next = reverseHead;     //temp指向：翻转链表头
           reverseHead = temp;          //翻转链表头：前移
        }


        //5、判断该链表：是否为奇数个结点，如果是：则需要慢指针，下移一步，因为此时，后半段链表（比）前半段链表多了一个结点
        if(fast != null){       //此时为：奇数个结点
            slow = slow.next;   //slow下移
        }


        //6、比较：前半段（已翻转的链表） 与 后半段（未翻转链表）是否相等
        while(slow != null){
            if(reverseHead.val != slow.val){
                return false;                         //不是回文链表
            }else{
                slow = slow.next;                    //slow下移
                reverseHead = reverseHead.next;      //翻转链表：下移
            }
        }

        return true;                                //是回文链表
    }
}
```

328、Odd Even Linked List

①、自己的解法

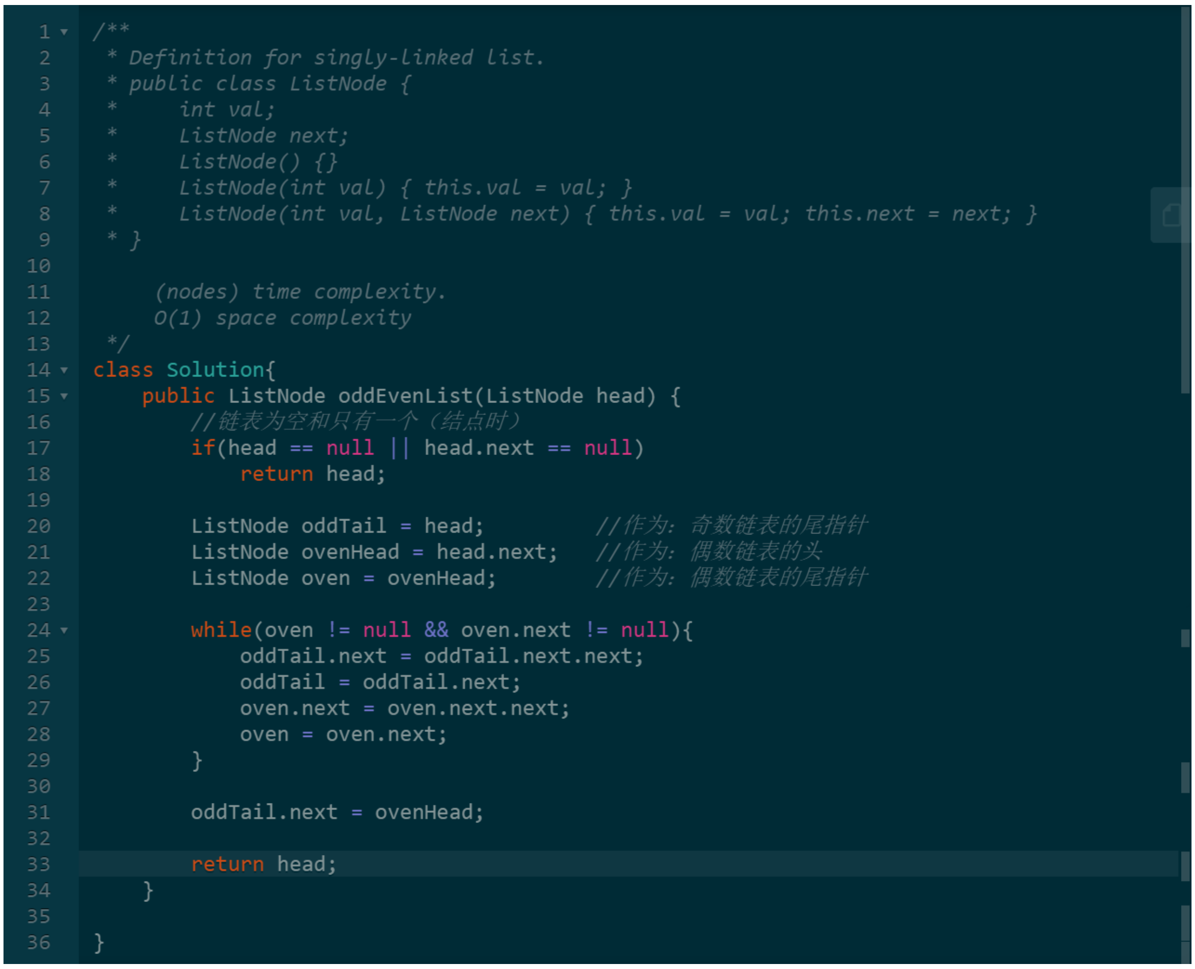


羊在风中飞 （编辑过）3 分钟前

非常：常规的想法

```java
class Solution{
    public ListNode oddEvenList(ListNode head) {
        if(head == null || head.next == null){          //没有结点， 只有一个结点时
            return head;
        }

        ListNode oddList = new ListNode(-1);     //记录：奇数链表的头结点
        ListNode ovenList = new ListNode(-1);    //记录：偶数链表的头结点
        ListNode oddTail = oddList;             //奇数链表的：尾指针
        ListNode ovenTail = ovenList;           //偶数链表的：尾指针

        //开始遍历head
        while(head != null){
            oddTail.next = head;            //奇数链表：连接奇数位置的（结点）
            oddTail = head;                 //奇数链表的：尾指针，下移
            head = head.next;               //head位置下移
            ovenTail.next = head;           //偶数链表：连接偶数位置的（结点）
            ovenTail = head;                //偶数链表的：尾指针，下移
            if(head != null){
                head = head.next;            //head位置下移
            }

        }

        oddTail.next = ovenList.next;       //奇数链表：指向（偶数链表的：有效结点）

        return oddList.next;                //返回（奇数链表）的头结点的：下一个结点
    }

}
```

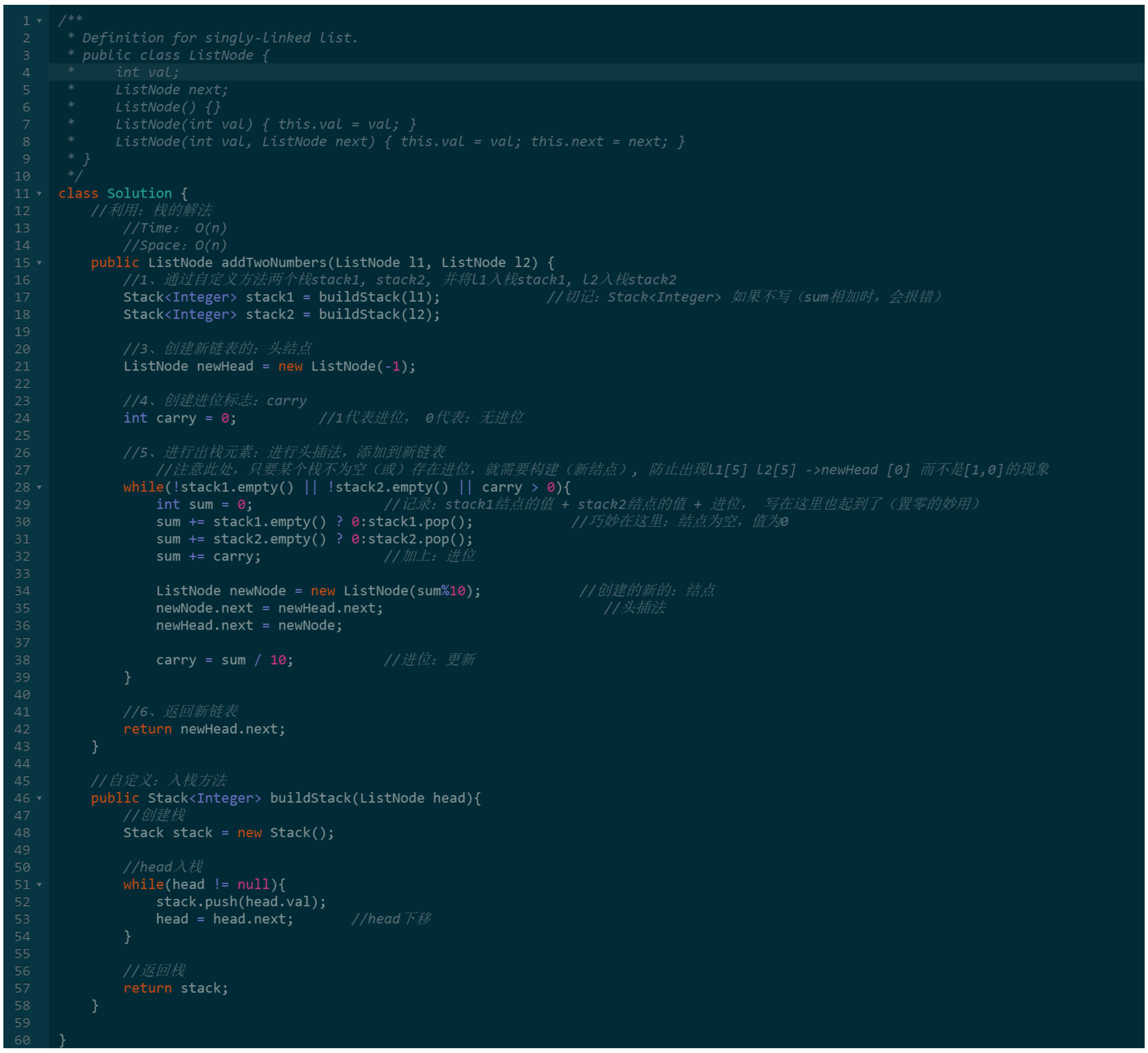
②、更简洁的解法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }

     (nodes) time complexity.
     O(1) space complexity
 */
class Solution{
    public ListNode oddEvenList(ListNode head) {
        //链表为空和只有一个（结点时）
        if(head == null || head.next == null)
            return head;

        ListNode oddTail = head;         //作为: 奇数链表的尾指针
        ListNode ovenHead = head.next;   //作为: 偶数链表的头
        ListNode oven = ovenHead;        //作为: 偶数链表的尾指针

        while(oven != null && oven.next != null){
            oddTail.next = oddTail.next.next;
            oddTail = oddTail.next;
            oven.next = oven.next.next;
            oven = oven.next;
        }

        oddTail.next = ovenHead;

        return head;
    }

}
```

445、Add Two Numbers II (Medium)

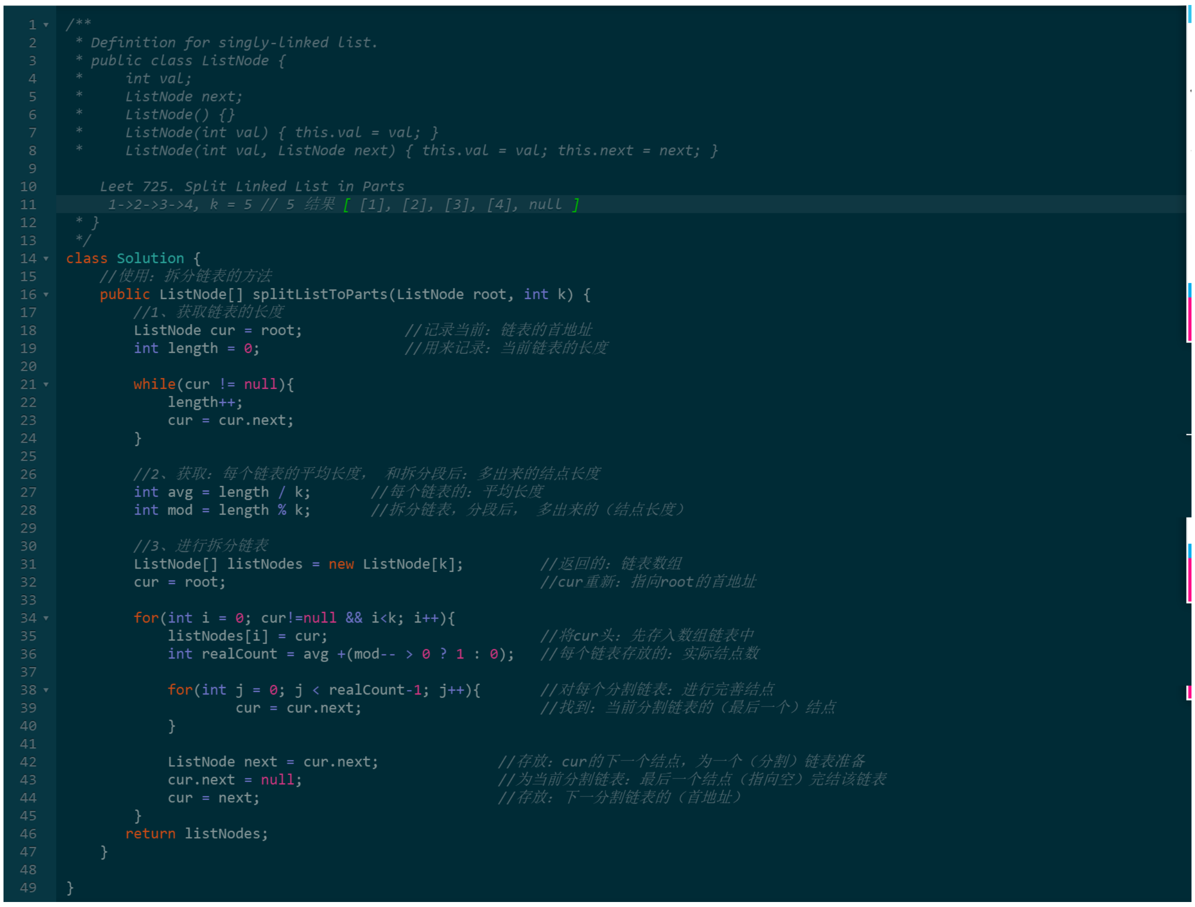
①、栈的解决方法



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    //利用：栈的解法
        //Time:  O(n)
        //Space: O(n)
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        //1、通过自定义方法两个栈stack1, stack2, 并将l1入栈stack1, l2入栈stack2
        Stack<Integer> stack1 = buildStack(l1);             //切记：Stack<Integer> 如果不写（sum相加时，会报错）
        Stack<Integer> stack2 = buildStack(l2);

        //3、创建新链表的：头结点
        ListNode newHead = new ListNode(-1);

        //4、创建进位标志：carry
        int carry = 0;          //1代表进位， 0代表：无进位

        //5、进行出栈元素：进行头插法，添加到新链表
            //注意此处，只要某个栈不为空（或）存在进位，就需要构建（新结点），防止出现l1[5] l2[5] ->newHead [0] 而不是[1,0]的现象
        while(!stack1.empty() || !stack2.empty() || carry > 0){
            int sum = 0;                //记录：stack1结点的值 + stack2结点的值 + 进位， 写在这里也起到了（置零的妙用）
            sum += stack1.empty() ? 0:stack1.pop();            //巧妙在这里：结点为空，值为0
            sum += stack2.empty() ? 0:stack2.pop();
            sum += carry;               //加上：进位

            ListNode newNode = new ListNode(sum%10);            //创建的新的：结点
            newNode.next = newHead.next;                          //头插法
            newHead.next = newNode;

            carry = sum / 10;           //进位：更新
        }

        //6、返回新链表
        return newHead.next;
    }

    //自定义：入栈方法
    public Stack<Integer> buildStack(ListNode head){
        //创建栈
        Stack stack = new Stack();

        //head入栈
        while(head != null){
            stack.push(head.val);
            head = head.next;       //head下移
        }

        //返回栈
        return stack;
    }

}
```

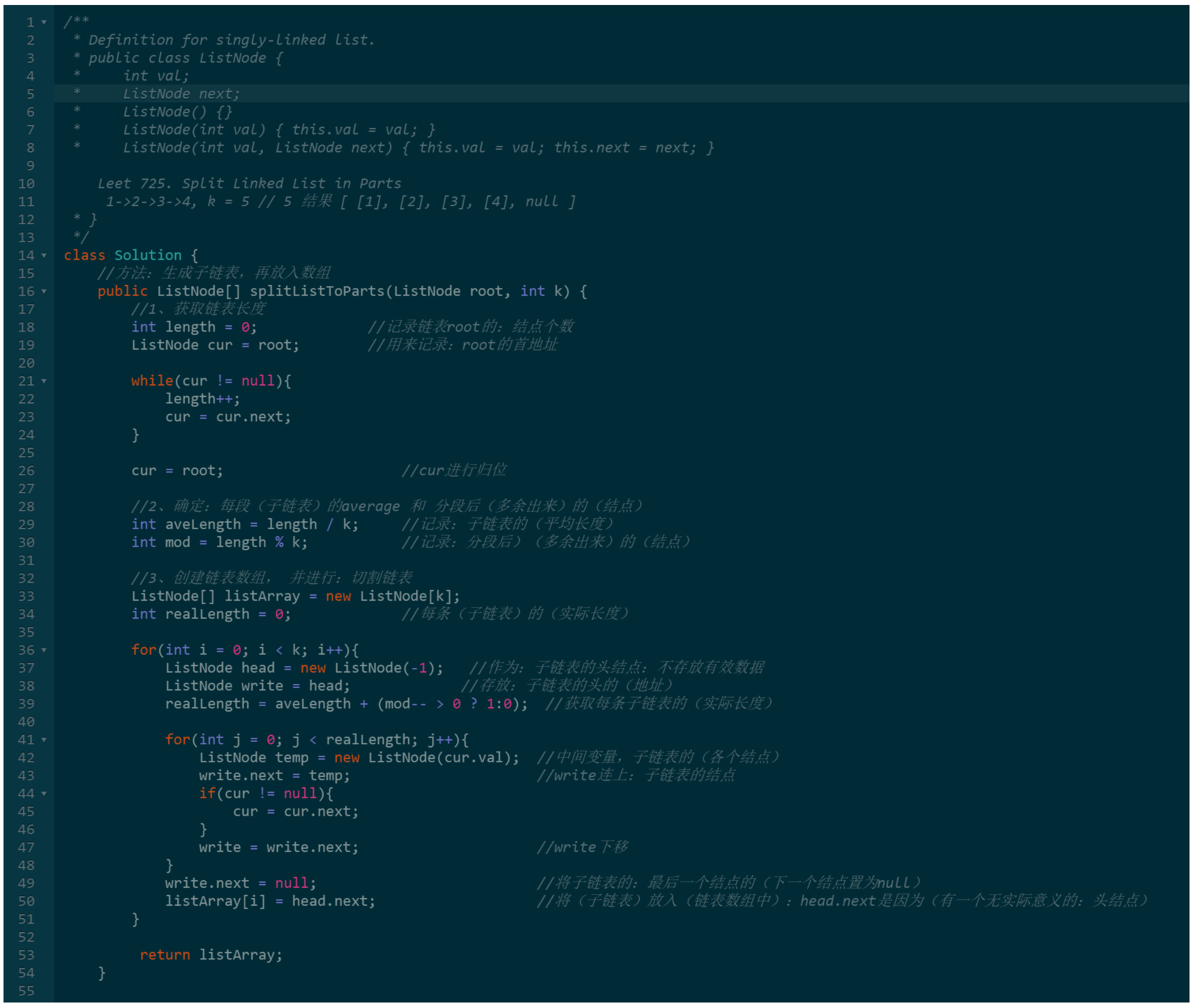
725、 Split Linked List in Parts

①、拆分链表（直接串上数组链表）



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }

    Leet 725. Split Linked List in Parts
     1->2->3->4, k = 5 // 5 结果 [ [1], [2], [3], [4], null ]
 * }
 */
class Solution {
    //使用：拆分链表的方法
    public ListNode[] splitListToParts(ListNode root, int k) {
        //1、获取链表的长度
        ListNode cur = root;            //记录当前：链表的首地址
        int length = 0;                 //用来记录：当前链表的长度

        while(cur != null){
            length++;
            cur = cur.next;
        }

        //2、获取：每个链表的平均长度， 和拆分段后：多出来的结点长度
        int avg = length / k;       //每个链表的：平均长度
        int mod = length % k;       //拆分链表，分段后， 多出来的（结点长度）

        //3、进行拆分链表
        ListNode[] listNodes = new ListNode[k];         //返回的：链表数组
        cur = root;                                     //cur重新：指向root的首地址

        for(int i = 0; cur!=null && i<k; i++){
            listNodes[i] = cur;                         //将cur头：先存入数组链表中
            int realCount = avg +(mod-- > 0 ? 1 : 0);   //每个链表存放的：实际结点数

            for(int j = 0; j < realCount-1; j++){       //对每个分割链表：进行完善结点
                    cur = cur.next;                     //找到：当前分割链表的（最后一个）结点
            }

            ListNode next = cur.next;              //存放：cur的下一个结点，为一个（分割）链表准备
            cur.next = null;                       //为当前分割链表：最后一个结点（指向空）完结该链表
            cur = next;                            //存放：下一分割链表的（首地址）
        }
       return listNodes;
    }

}
```

②、生成（子链表）再放入数组



```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }

    Leet 725. Split Linked List in Parts
     1->2->3->4, k = 5 // 5 结果 [ [1], [2], [3], [4], null ]
 * }
 */
class Solution {
    //方法：生成子链表，再放入数组
    public ListNode[] splitListToParts(ListNode root, int k) {
        //1、获取链表长度
        int length = 0;             //记录链表root的：结点个数
        ListNode cur = root;        //用来记录：root的首地址

        while(cur != null){
            length++;
            cur = cur.next;
        }

        cur = root;                     //cur进行归位

        //2、确定：每段（子链表）的average 和 分段后（多余出来）的（结点）
        int aveLength = length / k;     //记录：子链表的（平均长度）
        int mod = length % k;           //记录：分段后）（多余出来）的（结点）

        //3、创建链表数组， 并进行：切割链表
        ListNode[] listArray = new ListNode[k];
        int realLength = 0;             //每条（子链表）的（实际长度）

        for(int i = 0; i < k; i++){
            ListNode head = new ListNode(-1);   //作为：子链表的头结点：不存放有效数据
            ListNode write = head;             //存放：子链表的头的（地址）
            realLength = aveLength + (mod-- > 0 ? 1:0);  //获取每条子链表的（实际长度）

            for(int j = 0; j < realLength; j++){
                ListNode temp = new ListNode(cur.val);  //中间变量，子链表的（各个结点）
                write.next = temp;                      //write连上：子链表的结点
                if(cur != null){
                    cur = cur.next;
                }
                write = write.next;                     //write下移
            }
            write.next = null;                          //将子链表的：最后一个结点的（下一个结点置为null）
            listArray[i] = head.next;                   //将（子链表）放入（链表数组中）：head.next是因为（有一个无实际意义的：头结点）
        }

         return listArray;
    }

```