反转链表

https://leetcode.cn/problems/reverse-linked-list/solutions/551596/fan-zhuan-lian-biao-by-leetcode-solution-d1k2

方法一：迭代
假设链表为 1→2→3→∅，我们想要把它改成 ∅←1←2←3。

在遍历链表时，将当前节点的 next 指针改为指向前一个节点。由于节点没有引用其前一个节点，因此必须事先存储其前一个节点。在更改引用之前，还需要存储后一个节点。最后返回新的头引用。

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表的长度。需要遍历链表一次。
- 空间复杂度：$O(1)$。

方法二：递归

递归版本稍微复杂一些，其关键在于反向工作。假设链表的其余部分已经被反转，现在应该如何反转它前面的部分？

假设链表为：

$$n_1 \to \ldots \to n_{k-1} \to n_k \to n_{k+1} \to \ldots \to n_m \to \varnothing$$

若从节点 n_{k+1} 到 n_m 已经被反转，而我们正处于 n_k。

$$n_1 \to \ldots \to n_{k-1} \to n_k \to n_{k+1} \leftarrow \ldots \leftarrow n_m$$

我们希望 n_{k+1} 的下一个节点指向 n_k。

所以，$n_k.next.next = n_k$。

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

复杂度分析

时间复杂度：O(n)，其中 n 是链表的长度。需要对链表的每个节点进行反转操作。

空间复杂度：O(n)，其中 n 是链表的长度。空间复杂度主要取决于递归调用的栈空间，最多为 n 层。