203. 移除链表元素

```
//解法 1 特殊处理头节点
func removeElements(head *ListNode, val int) *ListNode {
    if head == nil {return nil}
    prev := head
    for prev.Next != nil {
        if prev.Next.Val == val {
            prev.Next = prev.Next.Next
        } else {
            prev = prev.Next
        }
    }
    if head.Val == val {
        head = head.Next
    }
    return head
}

//解法 2 添加虚拟头节点
func removeElements(head *ListNode, val int) *ListNode {
    if head == nil {return nil}
    newHead := new(ListNode)
    newHead.Next = head
    prev := newHead
    for prev.Next != nil {
        if prev.Next.Val == val {
            prev.Next = prev.Next.Next
        } else {
            prev = prev.Next
        }
    }
    return newHead.Next
}

//解法 3 改变链表的万能写法
func removeElements(head *ListNode, val int) *ListNode {
    if head == nil {return nil}
    newHead := new(ListNode)
    tail := newHead
    p := head
    for p != nil {
        tmp := p.Next
        if p.Val != val {
            p.Next = nil
            tail.Next = p
            tail = p
        }
        p = tmp
    }
    return newHead.Next
}
```

876. 链表的中间结点

```
func middleNode(head *ListNode) *ListNode {
    slow := head
    fast := head
```

```
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    return slow
}
```

83. 删除排序链表中的重复元素

```
func deleteDuplicates(head *ListNode) *ListNode {
    if head == nil {return head}
    newHead := &ListNode{Val:-111, Next:nil} // 虚拟头节点
    tail := newHead
    p := head
    for p != nil {
        tmp := p.Next
        if p.Val != tail.Val {
            tail.Next = p
            tail = p
            p.Next = nil
        }
        p = tmp
    }
    return newHead.Next
}
```

剑指 Offer 25. 合并两个排序的链表

```
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    if l1 == nil {return l2}
    if l2 == nil {return l1}
    p1 := l1
    p2 := l2
    head := new(ListNode) //虚拟头节点
    tail := head
    for p1 != nil && p2 != nil {
        if p1.Val <= p2.Val {
            tail.Next = p1
            tail = p1
            p1 = p1.Next
        } else {
            tail.Next = p2
            tail = p2
            p2 = p2.Next
        }
    }
    //如果 p1 还没处理完，就把剩下的直接接到 tail 后面
    if p1 != nil {tail.Next = p1}
    if p2 != nil {tail.Next = p2}
    return head.Next
}
```

2. 两数相加

```
func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    p1 := l1
    p2 := l2
    dummyHead := new(ListNode) //虚拟头节点
    tail := dummyHead
    carry := 0
    for p1 != nil || p2 != nil {
        sum := 0
        if p1 != nil {
            sum += p1.Val
            p1 = p1.Next
        }
        if p2 != nil {
            sum += p2.Val
            p2 = p2.Next
        }
        if carry != 0 {
            sum += carry
        }
        tail.Next = &ListNode{Val:sum%10}
        carry = sum/10
        tail = tail.Next
    }
    if carry != 0 {
        tail.Next = &ListNode{Val:carry}
    }
    return dummyHead.Next
}
```

206. 反转链表

```
// 1. var newHead = new(ListNode)
// 2. var newHead *ListNode = nil
// 1 和 2 的区别:
// 2 是*ListNode 的零值
// 1 等价于 var newHead = &ListNode{Val:0, Next:nil}
// 本题 newHead 初始值是反转之后的尾节点,所以需要 nil
func reverseList(head *ListNode) *ListNode {
    var newHead *ListNode = nil
    p := head
    for p != nil {
        tmp := p.Next
        p.Next = newHead
        newHead = p
        p = tmp
    }
    return newHead
}
```

234. 回文链表

```
func isPalindrome(head *ListNode) bool {
    if head == nil || head.Next == nil {return true}
    midNode := findMidNode(head)
    rightHalfHead := reverseList(midNode.Next)
```

```
    p := head
    q := rightHalfHead
    for q != nil {
        if p.Val != q.Val {return false}
        p = p.Next
        q = q.Next
    }
    return true
}

func findMidNode(head *ListNode) *ListNode{
    slow := head
    fast := head
    for fast.Next != nil && fast.Next.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
    }
    return slow
}

func reverseList(head *ListNode) *ListNode {
    if head == nil {return nil}
    var newHead *ListNode = nil
    p := head
    for p != nil {
        tmp := p.Next
        p.Next = newHead
        newHead = p
        p = tmp
    }
    return newHead
}
```

328. 奇偶链表

```
func oddEvenList(head *ListNode) *ListNode {
    if head == nil {return nil}
    oddHead := new(ListNode)
    oddTail := oddHead
    evenHead := new(ListNode)
    evenTail := evenHead
    p := head
    count := 1
    for p != nil {
        tmp := p.Next
        if count % 2 == 1 { // 奇数
            p.Next = nil
            oddTail.Next = p
            oddTail = p
        } else { // 偶数
            p.Next = nil
            evenTail.Next = p
            evenTail = p
        }
        count++
        p = tmp
    }
    oddTail.Next = evenHead.Next
    return oddHead.Next
}
```

```go
func oddEvenList(head *ListNode) *ListNode {
    if head == nil {return nil}
    odd := head
    even := head.Next
    podd := odd
    peven := even
    for podd.Next != nil && podd.Next.Next != nil {
        podd.Next = podd.Next.Next
        podd = podd.Next
        peven.Next = peven.Next.Next
        peven = peven.Next
    }
    podd.Next = even
    return odd
}
```

25. K 个一组翻转链表

```go
func reverseKGroup(head *ListNode, k int) *ListNode {
    dummyHead := new(ListNode)
    tail := dummyHead
    p := head
    for p != nil {
        count := 0
        q := p
        for q != nil {
            count++
            if count == k {
                break
            }
            q = q.Next
        }
        if q == nil {
            tail.Next = p
            return dummyHead.Next
        } else {
            tmp := q.Next
            nodes := reverse(p, q)
            tail.Next = nodes[0]
            tail = nodes[1]
            p = tmp
        }
    }
    return dummyHead.Next
}

func reverse(head *ListNode, tail *ListNode) []*ListNode {
    var newHead *ListNode = nil
    p := head
    for p != tail {
        tmp := p.Next
        p.Next = newHead
        newHead = p
        p = tmp
    }
    tail.Next = newHead
    newHead = tail
    return []*ListNode{tail, head}
}
```

剑指 Offer 22. 链表中倒数第 k 个节点

```
func getKthFromEnd(head *ListNode, k int) *ListNode {
    //遍历 1
    fast := head
    count := 0
    for fast != nil {
        count++
        if count == k {break}
        fast = fast.Next
    }
    if fast == nil { // 链表节点不够 k
        return nil
    }
    //遍历 2
    slow := head
    for fast.Next != nil {
        slow = slow.Next
        fast = fast.Next
    }
    return slow
}
```

19. 删除链表的倒数第 N 个结点

```
func removeNthFromEnd(head *ListNode, n int) *ListNode {
    //遍历 1：fast 先到第 n 个节点
    fast := head
    count := 0
    for fast != nil  {
        count++
        if count == n {
            break
        }
        fast = fast.Next
    }
    if fast == nil { //不够 k 个
        return head
    }
    //遍历 2
    slow := head
    var pre *ListNode = nil
    for fast.Next != nil {
        fast = fast.Next
        pre = slow // 加了这一行
        slow = slow.Next
    }
    // 删除倒数第 n 个节点
    if pre == nil { // 头节点是倒数第 n 个节点
        head = head.Next
    } else {
        pre.Next = slow.Next
    }
    return head
}
```

160. 相交链表

```
func getIntersectionNode(headA, headB *ListNode) *ListNode {
    //  求链表 A 的长度 na
    na := 0
    pA := headA
    for pA != nil {
        na++
        pA = pA.Next
    }
    //求链表 B 的长度 nb
    nb := 0
    pB := headB
    for pB != nil {
        nb++
        pB = pB.Next
    }

    //先让指向长链表的指针多走 na-nb 或 nb-na 步
    pA = headA
    pB = headB
    if na >= nb {
        for i := 0; i < na - nb; i++ {
            pA = pA.Next
        }
    } else {
        for i := 0; i < nb-na; i++ {
            pB = pB.Next
        }
    }
    //让 pA 和 pB 同步前进
    for pA != nil && pB != nil && pA != pB {
        pA = pA.Next
        pB = pB.Next
    }
    if pA == nil && pB == nil {
        return nil
    } else {
        return pA
    }
}
```

141. 环形链表

```
func hasCycle(head *ListNode) bool {
    if head == nil {return false}
    slow := head
    fast := head.Next
    for fast != nil && fast.Next != nil && slow != fast {
        slow = slow.Next
        fast = fast.Next.Next
    }
    if slow == fast {return true}
    return false
}
```