707.设计链表（中等）

1. 题目描述

   设计链表的实现。您可以选择使用单链表或双链表。单链表中的节点应该具有两个属性：`val` 和 `next`。`val` 是当前节点的值，`next` 是指向下一个节点的指针/引用。如果要使用双向链表，则还需要一个属性 `prev` 以指示链表中的上一个节点。假设链表中的所有节点都是 0-index 的。

   在链表类中实现这些功能：

   - get(index)：获取链表中第 `index` 个节点的值。如果索引无效，则返回`-1`。
   - addAtHead(val)：在链表的第一个元素之前添加一个值为 `val` 的节点。插入后，新节点将成为链表的第一个节点。
   - addAtTail(val)：将值为 `val` 的节点追加到链表的最后一个元素。
   - addAtIndex(index,val)：在链表中的第 `index` 个节点之前添加值为 `val` 的节点。如果 `index` 等于链表的长度，则该节点将附加到链表的末尾。如果 `index` 大于链表长度，则不会插入节点。如果 `index` 小于0，则在头部插入节点。
   - deleteAtIndex(index)：如果索引 `index` 有效，则删除链表中的第 `index` 个节点。

   示例：

   ```
   MyLinkedList linkedList = new MyLinkedList();
   linkedList.addAtHead(1);
   linkedList.addAtTail(3);
   linkedList.addAtIndex(1,2);   //链表变为1-> 2-> 3
   linkedList.get(1);            //返回2
   linkedList.deleteAtIndex(1);  //现在链表是1-> 3
   linkedList.get(1);            //返回3
   ```

   提示：

   - 所有`val`值都在 `[1, 1000]` 之内。
   - 操作次数将在 `[1, 1000]` 之内。
   - 请不要使用内置的 LinkedList 库。

2. 简单实现

   ```
   class MyLinkedList {
   public:
       int val = -1;
       MyLinkedList* next = NULL;
       /** Initialize your data structure here. */
       MyLinkedList() {
           val = -1;
           next = NULL;
       }
       MyLinkedList(int x) {
           val = x;
           next = NULL;
       }
   ```

```
    /** Get the value of the index-th node in the linked list. If the index is
invalid, return -1. */
    int get(int index) {
        MyLinkedList* cur = next;
        while(index-- && cur) cur = cur->next;
        if(!cur) return -1;
        else return cur->val;
    }

    /** Add a node of value val before the first element of the linked list. After
the insertion, the new node will be the first node of the linked list. */
    void addAtHead(int val) {
        MyLinkedList* temp = new MyLinkedList(val);
        temp->next = next;
        next = temp;
    }

    /** Append a node of value val to the last element of the linked list. */
    void addAtTail(int val) {
        if(!next) addAtHead(val);
        MyLinkedList* cur = next;
        while(cur->next) cur = cur->next;
        MyLinkedList* temp = new MyLinkedList(val);
        cur->next = temp;
    }

    /** Add a node of value val before the index-th node in the linked list. If
index equals to the length of linked list, the node will be appended to the end of
linked list. If index is greater than the length, the node will not be inserted. */
    void addAtIndex(int index, int val) {
        if(index <= 0){
            addAtHead(val);
            return;
        };
        MyLinkedList* pre = this;
        MyLinkedList* cur = next;
        while(index && cur){
            pre = cur;
            cur = cur->next;
            index--;
        }
        if(!cur){
            if(index) return;
            else{
                cur = new MyLinkedList(val);
                pre->next = cur;
            }
        }
        else{
            MyLinkedList* temp = new MyLinkedList(val);
            temp->next = cur;
            pre->next = temp;
        }
```

```
    }
    /** Delete the index-th node in the linked list, if the index is valid. */
    void deleteAtIndex(int index) {
        if(index < 0) return;
        MyLinkedList* pre = this;
        MyLinkedList* cur = next;
        while(index && cur){
            pre = cur;
            cur = cur->next;
            index--;
        }
        if(!cur) return;
        else pre->next = cur->next;
    }
};
/**
 * Your MyLinkedList object will be instantiated and called as such:
 * MyLinkedList* obj = new MyLinkedList();
 * int param_1 = obj->get(index);
 * obj->addAtHead(val);
 * obj->addAtTail(val);
 * obj->addAtIndex(index,val);
 * obj->deleteAtIndex(index);
 */
```

链表中的双指针

让我们从一个经典问题开始：

> 给定一个链表，判断链表中是否有环。

你可能已经使用`哈希表`提出了解决方案。但是，使用`双指针技巧`有一个更有效的解决方案。在阅读接下来的内容之前，试着自己仔细考虑一下。

想象一下，有两个速度不同的跑步者。如果他们在直路上行驶，快跑者将首先到达目的地。但是，如果它们在圆形跑道上跑步，那么快跑者如果继续跑步就会追上慢跑者。

这正是我们在链表中使用两个速度不同的指针时会遇到的情况：

1. `如果没有环，快指针将停在链表的末尾。`
2. `如果有环，快指针最终将与慢指针相遇。`

所以剩下的问题是：

> 这两个指针的适当速度应该是多少？

一个安全的选择是每次移动慢指针`一步`，而移动快指针`两步`。每一次迭代，快速指针将额外移动一步。如果环的长度为 *M*，经过 *M* 次迭代后，快指针肯定会多绕环一周，并赶上慢指针。

141.环形链表（简单）

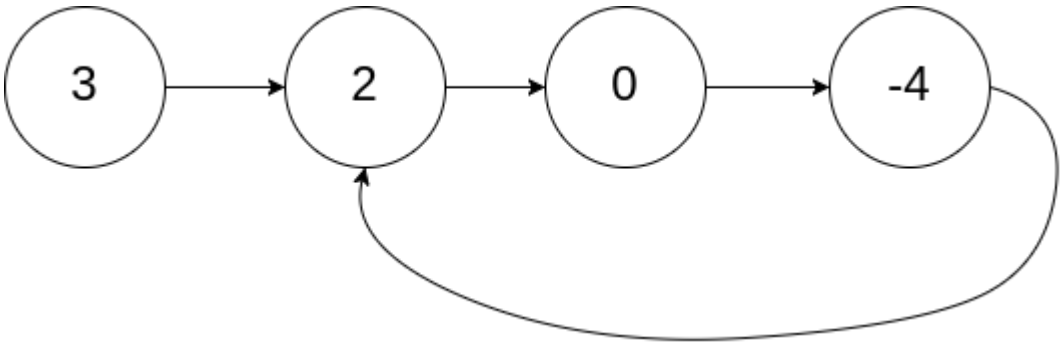
1. 题目描述

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

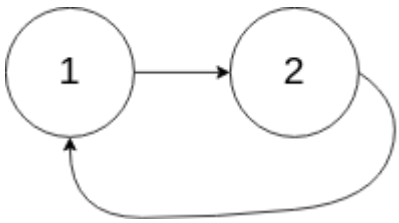
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3：

```
输入：head = [1], pos = -1
输出：false
解释：链表中没有环。
```

进阶：你能用 *O(1)*（即，常量）内存解决此问题吗？

2. 简单实现

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(!head || !head->next)
            return false;
```

```
        ListNode *slow = head;
        ListNode *fast = head->next;
        while(fast && fast != slow){
            if(fast->next && fast->next->next) fast = fast->next->next;
            else return false;
            slow = slow->next;
        }
        if(!fast) return false;
        else return true;
    }
};
```

142.环形链表II（中等）

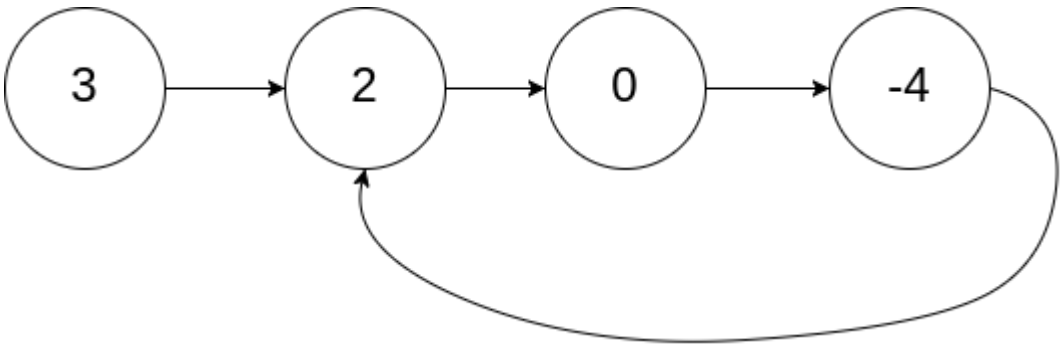
1. 题目描述

给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 `null`。为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

说明：不允许修改给定的链表。

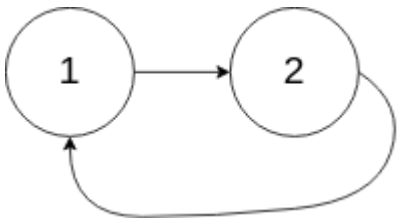
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：tail connects to node index 0
解释：链表中有一个环，其尾部连接到第一个节点。
```

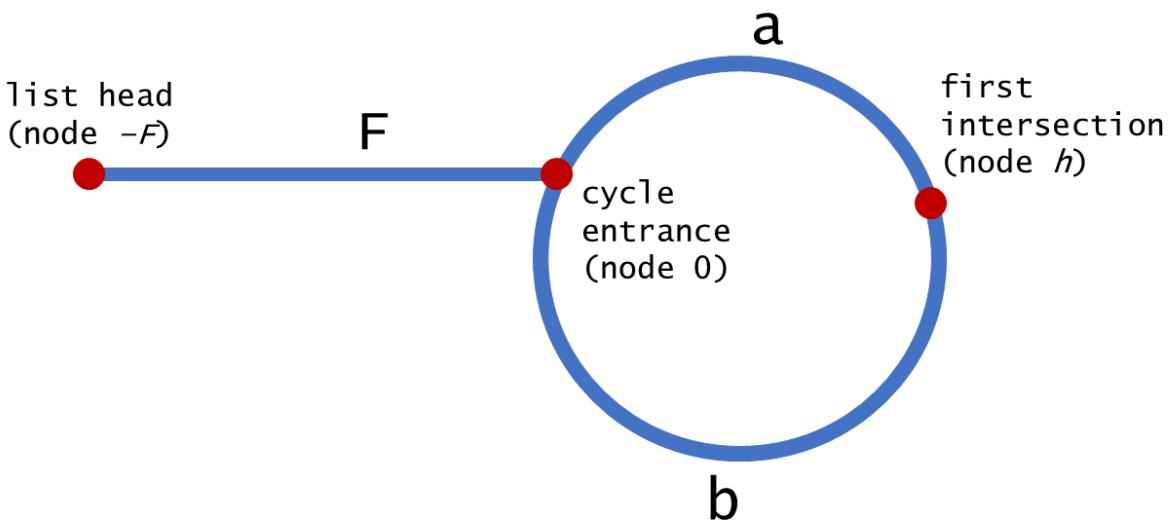


示例 3：

```
输入：head = [1], pos = -1
输出：no cycle
解释：链表中没有环。
```

进阶：你是否可以不用额外空间解决此题？

2. 简单实现



快慢指针相遇时，有：$(F+b)*2 = F+b+a+b$ -> $F=a$

因此，相遇后，head从头开始走，慢指针也继续走，两者相遇时则为入口节点

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if(!head || !head->next)
            return NULL;
        ListNode *slow = head;
        ListNode *fast = head->next;
        int count = 0;
        while(fast && fast != slow){
            if(fast->next && fast->next->next) fast = fast->next->next;
            else return NULL;
            slow = slow->next;
        }
        if(!fast) return NULL;//无环
        else{//相遇
            slow = slow->next;
            while(head != slow){
                head = head->next;
```

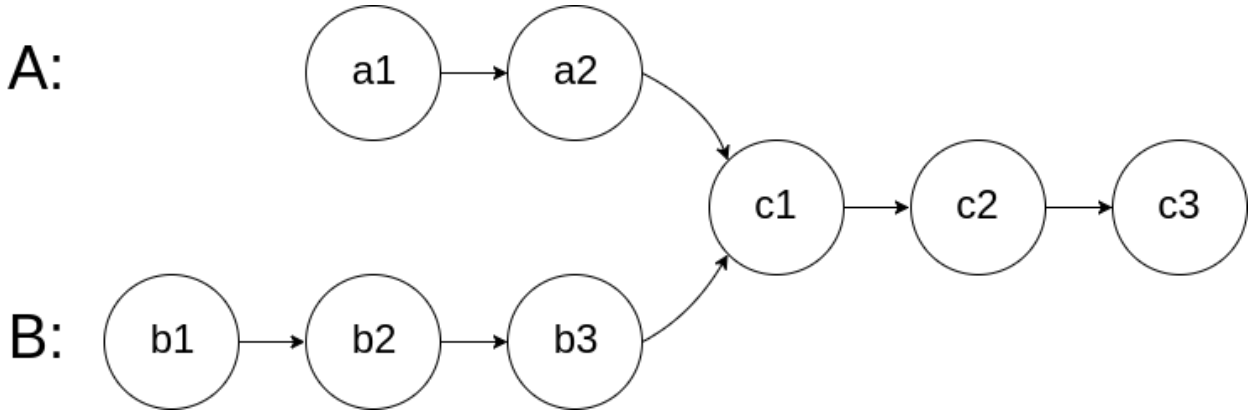
```
                slow = slow->next;
            }
            return head;
        }
    }
};
```

160. 相交链表（简单）

1. 题目描述

   编写一个程序，找到两个单链表相交的起始节点。

   如下面的两个链表：



   在节点 c1 开始相交。

   注意：

   - 如果两个链表没有交点，返回 `null`.
   - 在返回结果后，两个链表仍须保持原有的结构。
   - 可假定整个链表结构中没有循环。
   - 程序尽量满足 O(*n*) 时间复杂度，且仅用 O(*1*) 内存。

2. 简单实现

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if(!headA || !headB) return NULL;
        int lena=0;
        int lenb=0;
        ListNode *cur_A = headA;
        ListNode *cur_B = headB;
        ListNode *result = NULL;
        while(cur_A){
            lena++;
            cur_A=cur_A->next;
        }
        while(cur_B){
            lenb++;
            cur_B=cur_B->next;
        }
```

```
        while(lena>lenb){
            headA = headA->next;
            lena--;
        }
        while(lena<lenb){
            headB = headB->next;
            lenb--;
        }
        while(headA && headB && headA != headB){
            headA = headA->next;
            headB = headB->next;
        }
        return headA;
    }
};
```

19.删除链表的倒数第N个节点（中等）

1. 题目描述

   给定一个链表，删除链表的倒数第 *n* 个节点，并且返回链表的头结点。

   示例：

   ```
   给定一个链表: 1->2->3->4->5, 和 n = 2.

   当删除了倒数第二个节点后，链表变为 1->2->3->5.
   ```

   说明：给定的 *n* 保证是有效的。

   进阶：你能尝试使用一趟扫描实现吗？

2. 简单实现

   使快指针比慢指针领先n个节点即可

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* h = new ListNode(0);
        h->next = head;
        ListNode* p = head;
        ListNode* q = h;
        while(n--){
            p = p->next;
        }
        while(p){
            p = p->next;
            q = q->next;
        }
        q->next = q->next->next;
        return h->next;
    }
```

```
};
```

关于链表的几个经典问题

206.反转链表（简单）

1. 题目描述

   反转一个单链表。

   示例:

   ```
   输入: 1->2->3->4->5->NULL
   输出: 5->4->3->2->1->NULL
   ```

   进阶:你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

2. 简单实现

```
class Solution {
public:
    //迭代
    ListNode* reverseList(ListNode* head) {
        ListNode* cur = NULL;
        ListNode* next = head;
        while(next){
            ListNode* temp = next->next;
            next->next = cur;
            cur = next;
            next = temp;
        }
        return cur;
    }
    //递归
    ListNode* reverseList(ListNode* head) {
        if(!head) return NULL;
        else if(!head->next) return head;
        ListNode* done = reverseList(head->next);
        head->next->next = head;
        head->next = NULL;
        return done;
    }
};
```

303.移除链表元素（简单）

1. 题目描述

   删除链表中等于给定值 **val** 的所有节点。

   示例:

```
输入: 1->2->6->3->4->5->6, val = 6
输出: 1->2->3->4->5
```

2. 简单实现

添加头节点可以简化问题；注意尾指针；虽然不是必须的，但是把删除的节点真的delete掉是个好习惯

```
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        ListNode* newhead = new ListNode(-1);
        newhead->next = head;
        ListNode* pre = newhead;
        ListNode* cur = head;
        while(cur){
            if(cur->val == val){
                pre->next = cur->next;
                delete cur;
                cur = pre->next;
            }
            else{
                pre = cur;
                cur = cur->next;
            }
        }
        return newhead->next;
    }
};
```

328.奇偶链表（中等）

1. 题目描述

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

示例 1:

```
输入: 1->2->3->4->5->NULL
输出: 1->3->5->2->4->NULL
```

示例 2:

```
输入: 2->1->3->5->6->4->7->NULL
输出: 2->3->6->7->1->5->4->NULL
```

说明:

- 应当保持奇数节点和偶数节点的相对顺序。

- 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

2. 简单实现

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(!head)
            return head;
        ListNode* evenhead = head->next;//偶数链表的头
        ListNode* odd = head;//奇数当前节点
        ListNode* even = evenhead;//偶数当前节点
        while(even){
            if(even->next){
                //其实就是隔一个插一个，这里一次处理两个，一个给odd，一个给even
                odd->next = even->next;
                even->next = even->next->next;
                odd = odd->next;
                even = even->next;
            }
            else{
                odd->next = evenhead;
                return head;
            }
        }
        odd->next = evenhead;
        return head;
    }
};
```

234.回文链表（简单）

1. 题目描述

   请判断一个链表是否为回文链表。

   示例 1:

   ```
   输入: 1->2
   输出: false
   ```

   示例 2:

   ```
   输入: 1->2->2->1
   输出: true
   ```

   进阶：你能否用 O(n) 时间复杂度和 O(1) 空间复杂度解决此题?

2. 简单实现

   用栈可以很简单的实现，此处省略代码

   对于进阶问题，可以：

1. 遍历一遍链表求得长度
2. 将链表后半段进行反转
3. 判断前半段和后半段是否相同

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if(!head) return true;
        int len = 0;
        ListNode* cur = head;
        while(cur){
            len++;
            cur = cur->next;
        }
        //计算长度
        len /= 2;
        cur = head;
        while(len--)
            cur = cur->next;
        //反转
        ListNode* pre = NULL;
        ListNode* next = cur->next;
        while(next){
            cur->next = pre;
            pre = cur;
            cur = next;
            next = cur->next;
        }
        cur->next = pre;
        //检验
        while(head && cur){
            if(head->val != cur->val)
                return false;
            head = head->next;
            cur = cur->next;
        }
        return true;
    }
};
```

3. 更优解法

   看题解之后再结合我自己的想法，觉得可以：

   1. 快慢指针法找中点
   2. 慢指针在移动的同时对链表进行反转
   3. 再检验

双链表

707.设计链表（中等）

1. 题目描述（同前面单链表的设计链表题目）
2. 简单实现

```
class MyLinkedList {
public:
    int val = -1;
    MyLinkedList* prev = NULL;
    MyLinkedList* next = NULL;
    /** Initialize your data structure here. */
    MyLinkedList() {
        val = -1;
        prev = NULL;
        next = NULL;
    }
    MyLinkedList(int x) {
        val = x;
        prev = NULL;
        next = NULL;
    }
    /** Get the value of the index-th node in the linked list. If the index is
invalid, return -1. */
    int get(int index) {
        MyLinkedList* cur = next;
        while(index-- && cur) cur = cur->next;
        if(!cur) return -1;
        else return cur->val;
    }

    /** Add a node of value val before the first element of the linked list. After
the insertion, the new node will be the first node of the linked list. */
    void addAtHead(int val) {
        MyLinkedList* temp = new MyLinkedList(val);
        temp->next = next;
        if(temp->next) temp->next->prev = temp;
        next = temp;
    }

    /** Append a node of value val to the last element of the linked list. */
    void addAtTail(int val) {
        if(!next) addAtHead(val);
        MyLinkedList* cur = next;
        while(cur->next) cur = cur->next;
        MyLinkedList* temp = new MyLinkedList(val);
        cur->next = temp;
        temp->prev = cur;
    }

    /** Add a node of value val before the index-th node in the linked list. If
index equals to the length of linked list, the node will be appended to the end of
linked list. If index is greater than the length, the node will not be inserted. */
    void addAtIndex(int index, int val) {
        if(index <= 0){
            addAtHead(val);
```

```
            return;
        };
        MyLinkedList* pre = this;
        MyLinkedList* cur = next;
        while(index && cur){
            pre = cur;
            cur = cur->next;
            index--;
        }
        if(!cur){
            if(index) return;
            else{
                cur = new MyLinkedList(val);
                pre->next = cur;
                cur->prev = pre;
            }
        }
        else{
            MyLinkedList* temp = new MyLinkedList(val);
            temp->prev = pre;
            temp->next = cur;
            pre->next = temp;
            cur->prev = temp;
        }
    }

    /** Delete the index-th node in the linked list, if the index is valid. */
    void deleteAtIndex(int index) {
        if(index < 0) return;
        MyLinkedList* pre = this;
        MyLinkedList* cur = next;
        while(index && cur){
            pre = cur;
            cur = cur->next;
            index--;
        }
        if(!cur) return;
        else{
            pre->next = cur->next;
            if(cur->next)
                cur->next->prev = pre;
            delete cur;
        }
    }
};
```

小结

这里我们提供链表和其他数据结构（包括数组，队列和栈）之间`时间复杂度`的比较：

		Array	Singly Linked List	Doubly Linked List	Queue	Stack
Access	by index	O(1)	O(N)	O(N)	O(N)	O(N)
Add	before first node	O(N)	O(1)	O(1)	O(N)	O(N)
	after given node	O(N)	O(1)	O(1)	O(N)	O(N)
	after last node	O(1)	O(1)	O(1)	O(1)	O(1)
Delete	the first node	O(N)	O(1)	O(1)	O(1)	O(N)
	a given node	O(N)	O(N)	O(1)	O(N)	O(N)
	the last node	O(1)	O(N)	O(1)	O(N)	O(1)
Search	a given value	O(N)	O(N)	O(N)	O(N)	O(N)

经过这次比较，我们不难得出结论：

- 如果你需要经常添加或删除结点，链表可能是一个不错的选择。
- 如果你需要经常按索引访问元素，数组可能是比链表更好的选择。

21.合并两个有序链表（简单）

1. 题目描述

   将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

   示例：

   ```
   输入：1->2->4, 1->3->4
   输出：1->1->2->3->4->4
   ```

2. 简单实现

   和有序数组的合并没什么太大的区别

   ```
   class Solution {
   public:
       ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
           if(!l1) return l2;
           if(!l2) return l1;
           ListNode* head = new ListNode(-1);
           ListNode* temp = head;
           while(l1 && l2){
               if(l1->val <= l2->val){
                   temp->next = l1;
                   temp = l1;
                   l1 = l1->next;
               }
               else{
                   temp->next = l2;
   ```

```
                temp = l2;
                l2 = l2->next;
            }
        }
        if(l1) temp->next = l1;
        if(l2) temp->next = l2;
        return head->next;
    }
};
```

2.两数相加（中等）

1. 题目描述

   给出两个 **非空** 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 **逆序** 的方式存储的，并且它们的每个节点只能存储 **一位** 数字。

   如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

   您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

   示例:

```
输入: (2 -> 4 -> 3) + (5 -> 6 -> 4)
输出: 7 -> 0 -> 8
原因: 342 + 465 = 807
```

2. 简单实现

   和字符串的两数相加差不多

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* head = new ListNode(-1);
        ListNode* cur = head;
        int c = 0;
        while(l1 && l2){
            int sum = l1->val + l2->val + c;
            if(sum > 9){
                sum -= 10;
                c = 1;
            }
            else
                c = 0;
            cur->next = new ListNode(sum);
            cur = cur->next;
            l1 = l1->next;
            l2 = l2->next;
        }
        if(l2) l1 = l2;
        while(l1){
            cout << l1->val << ' ' << c << endl;
```

```
            int sum = l1->val + c;
            l1 = l1->next;
            if(sum > 9){
                sum -= 10;
                c = 1;
                cur->next = new ListNode(sum);
                cur = cur->next;
            }
            else{//无需继续遍历进位
                cur->next = new ListNode(sum);
                cur->next->next = l1;
                c = 0;
                break;
            }
        }
        if(c)
            cur->next = new ListNode(1);
        return head->next;
    }
};
```

430.扁平化多级双向链表（中等）

1. 题目描述

   您将获得一个双向链表，除了下一个和前一个指针之外，它还有一个子指针，可能指向单独的双向链表。这些子列表可能有一个或多个自己的子项，依此类推，生成多级数据结构，如下面的示例所示。扁平化列表，使所有结点出现在单级双链表中。您将获得列表第一级的头部。

   示例:

```
输入:
 1---2---3---4---5---6--NULL
         |
         7---8---9---10--NULL
             |
             11--12--NULL

输出:
1-2-3-7-8-11-12-9-10-4-5-6-NULL
```

2. 简单实现

   每次遇到有child的节点，用栈保存其next，然后进入到child级；每次遇到空节点，则需要出栈；**注意有child的节点的next是NULL的特殊情况**

```
/*// Definition for a Node.
class Node {
public:
    int val;
    Node* prev;
    Node* next;
```

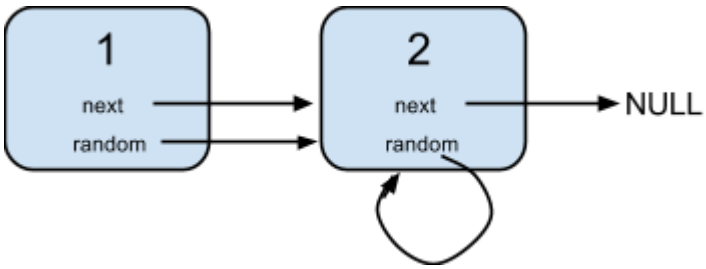
```
    Node* child;
    Node() {}
    Node(int _val, Node* _prev, Node* _next, Node* _child) {
        val = _val;
        prev = _prev;
        next = _next;
        child = _child;
    }
};*/
class Solution {
public:
    Node* flatten(Node* head) {
        if(!head) return NULL;
        Node* ans = new Node(-1,NULL,NULL,NULL);
        Node* cur = ans;
        stack<Node*> s;
        Node* temp = head;
        while(1){
            if(!temp){//某一级遍历结束
                if(s.empty()) break;//整个链表遍历完毕
                else{//返回上一级剩余部分
                    temp = s.top();
                    s.pop();
                }
            }
            if(!temp) continue;//上一级有child的节点next为NULL
            if(!temp->child){//没child，继续向下
                cur->next = temp;
                temp->prev = cur;
                cur = temp;
                temp = temp->next;
            }
            else{//有child，next入栈，temp进入下一级
                cur->next = temp;
                temp->prev = cur;
                s.push(temp->next);
                cur = temp;
                temp = temp->child;
                cur->child = NULL;
            }
        }
        ans = ans->next;
        ans->prev = NULL;//头节点的prev也要设为NULL
        return ans;
    }
};
```

138.复制带随机指针的链表（中等）

1. 题目描述

给定一个链表，每个节点包含一个额外增加的随机指针，该指针可以指向链表中的任何节点或空节点。 要求返回这个链表的**深拷贝**。

示例：




```
输入：
{"$id":"1","next":{"$id":"2","next":null,"random":{"$ref":"2"},"val":2},"random":
{"$ref":"2"},"val":1}

解释：
节点 1 的值是 1，它的下一个指针和随机指针都指向节点 2 。
节点 2 的值是 2，它的下一个指针指向 null，随机指针指向它自己。
```

提示：必须返回**给定头的拷贝**作为对克隆列表的引用。

2. 简单实现

先遍历一遍链表构建出所有节点，同时用map记录旧节点对应的新节点；然后再遍历旧节点的random值，利用Map构建新链表各节点的random值

```
/*Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;
    Node() {}
    Node(int _val, Node* _next, Node* _random) {
        val = _val;
        next = _next;
        random = _random;
    }
};*/
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if(!head) return NULL;
        map<Node*, Node*> m;//old->new
        //复制所有节点，构建map
        Node* ans = new Node(head->val, NULL, NULL);
        Node* old = head;
        Node* cur = ans;
        m[old] = cur;
        while(old->next){
            old = old->next;
            cur->next = new Node(old->val, NULL, NULL);
```

```
            cur = cur->next;
            m[old] = cur;
        }
        //构建random
        old = head;
        cur = ans;
        while(old){
            cur->random = m[old->random];
            old = old->next;
            cur = cur->next;
        }
        return ans;
    }
};
```

3. 最优解法：O(1)额外空间

    1. 复制节点，同时将复制节点链接到原节点后面，如A->B->C 变为 A->A'->B->B'->C->C'。
    2. 设置复制节点的random值：如node=、指向A时，`node->next->random = node->random->next;`
    3. 将复制链表从原链表分离。

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (head == nullptr) return head;
        Node *node = head;
        //1. 将复制节点添加到原节点后面
        while (node != nullptr) {
            Node *copy = new Node(node->val, nullptr, nullptr);
            copy->next = node->next;
            node->next = copy;
            node = copy->next;
        }
        //2. 复制random节点
        node = head;
        while (node != nullptr) {
            if (node->random != nullptr)
                node->next->random = node->random->next;
            node = node->next->next;
        }
        //3. 分离链表
        node = head;
        Node *newHead = head->next;
        Node *newNode = newHead;
        while (node != nullptr) {
            node->next = node->next->next;
            if (newNode->next != nullptr)
                newNode->next = newNode->next->next;
            node = node->next;
            newNode = newNode->next;
        }
        return newHead;
    }
```

```
};
```

61.旋转链表（中等）

1. 题目描述

   给定一个链表，旋转链表，将链表每个节点向右移动 *k* 个位置，其中 *k* 是非负数。

   示例 1:

```
输入: 1->2->3->4->5->NULL, k = 2
输出: 4->5->1->2->3->NULL
解释:
向右旋转 1 步: 5->1->2->3->4->NULL
向右旋转 2 步: 4->5->1->2->3->NULL
```

   示例 2:

```
输入: 0->1->2->NULL, k = 4
输出: 2->0->1->NULL
解释:
向右旋转 1 步: 2->0->1->NULL
向右旋转 2 步: 1->2->0->NULL
向右旋转 3 步: 0->1->2->NULL
向右旋转 4 步: 2->0->1->NULL
```

2. 简单实现

   - 关键在于找到旋转点x，添加头节点ans，最终的旋转点x变为ans->next，即新链表的第一个节点
   - 旋转点可以用快慢指针找到，快指针比慢指针快k步
   - 由于需要将旋转点x之前的节点next设为NULL，且旧链表尾节点的next要连到旧链表的head上，因此可以使快慢指针都前瞻一步，即停止时慢节点指向x之前的节点，快节点指向尾部节点而非NULL
   - 另外需要处理k大于等于链表长度的情况

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(k == 0) return head;
        ListNode* ans = new ListNode(-1);
        //快指针前移k步
        ListNode* fast = head;
        int len = 1;
        while(len < k && fast->next){
            fast = fast->next;
            len++;
        }
        if(!fast->next){//k值大于等于链表长度
            k %= len;
            return rotateRight(head, k);
        }
        else{//找到旋转点
```

```
            ListNode* slow = head;
            fast = fast->next;
            while(fast->next){
                slow = slow->next;
                fast = fast->next;
            }
            //旋转链表
            ans->next = slow->next;
            slow->next = NULL;
            fast->next = head;
        }
        return ans->next;
    }
};
```