简介

链表与数组同属于线性表，应用广泛，两者都可以根据其特性造出一些题目。我们知道数组可以作为高级算法的载体进行一系列复杂的操作，但是链表的题目很少用来做高级算法的载体，因此题目要少很多，类型也相对固定，只要掌握几种常见情况就不用怕考察了。

链表有普通单链表、循环链表和双向链表三种基本的类型。**普通链表**就是只给你一个指向链表头的指针head，没有链表的其他信息，如果遍历链表最终会在访问尾结点之后获得null。而**循环链表**就是尾结点又指向头结点，整个链表成了一个环，这种场景在算法里、在应用里都很少。应用更多的是**带头结点的链表**，就是给链表增加一个额外的结点记录头、尾、甚至元素个数等信息。**双向链表**就是每个节点有两个指针，一个next指向下一个结点，一个prev指向上一个结点，很明显用这种结构查找、移动元素更方便，但是操作更为复杂。

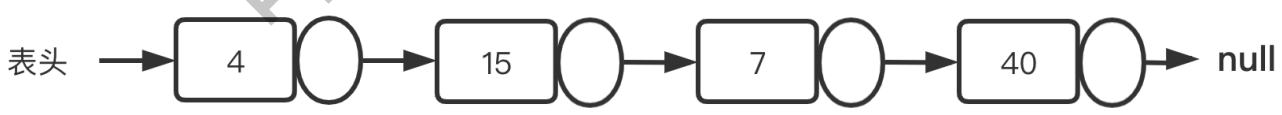
在工程应用，极少见到普通单链表，也很少见到循环链表。应用比较多的是带头结点的单链表和双向循环链表。 有时候会进行组合从而实现更丰富的功能，例如JUC中condition就是双向链表（AQS）+一个带头结点的单链表，而阻塞队列就是一个双向链表+两个带头结点的单链表。

与工程应用不同的是普通单链表在算法中很常见，主要是因为代码非常简洁，这也体现了算法题与工程应用的差异。

1.链表基础

1.1 链表的内部结构

首先看一下什么是链表？单向链表包含多个结点，每个结点有一个指向后继元素的next指针。表中最后一个元素的next指向null。如下图：



我们在数组中就说过任何数据结构的基础都是创建+增删改查，由这几个操作可以构造很多算法题，所以我们也从这五项开始学习链表。

首先理解JVM是怎么构建出链表的，我们知道JVM里有栈区和堆区，栈区主要存引用，也就是一个指向实际对象的地址，而堆区存的才是创建的对象，例如我们定义这样一个类：

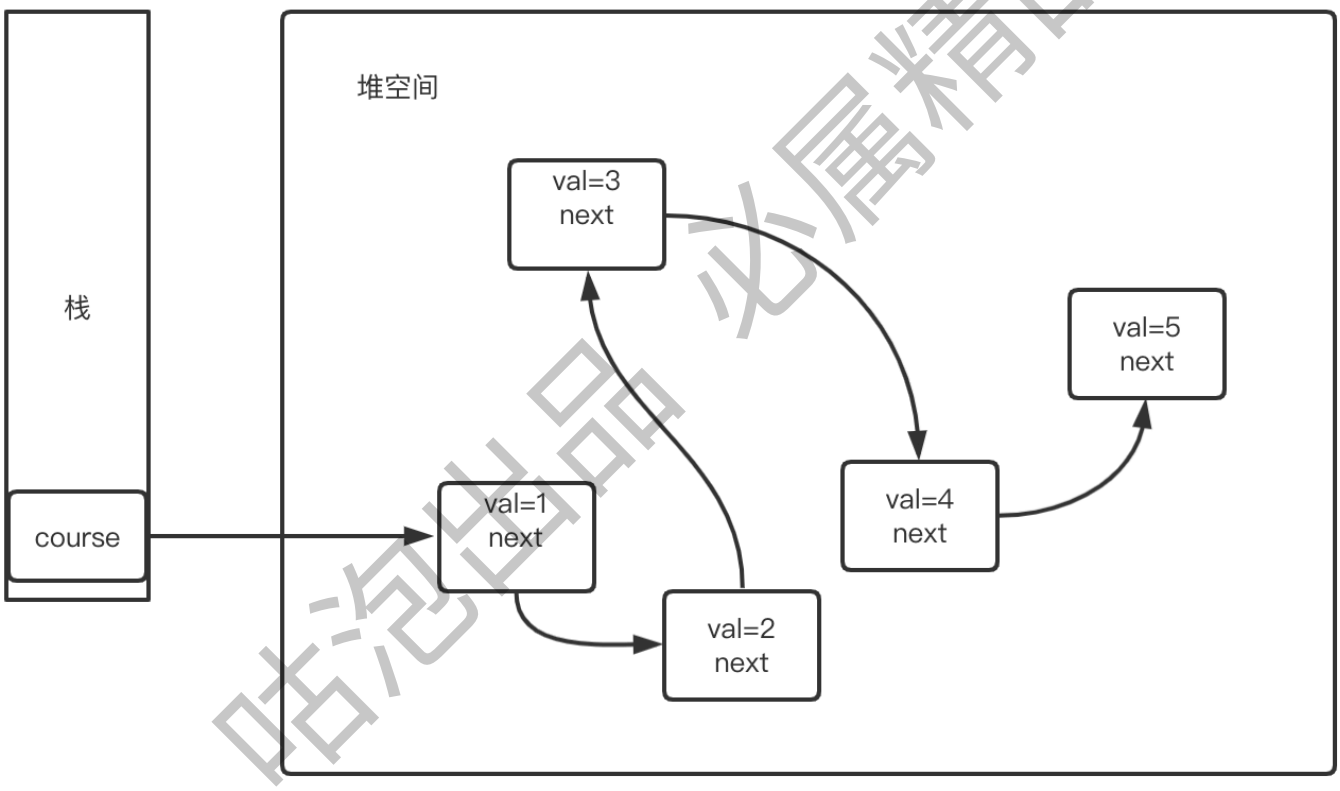
```
public class Course{
 Teacher teacher;
 Student student;
}
```

这里的teacher和student就是指向堆的地址。

假如我们这样定义：

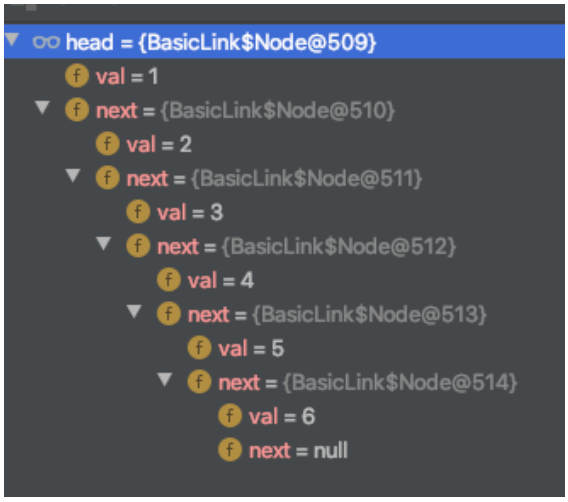
```
public class Course{
 int val;
 Course next;
}
```

这时候next就指向了下一个同为Course类型的对象了，例如：



这里通过栈中的引用（也就是地址）就可以找到val(1)，然后val(1)结点又存了指向val(2)的地址，而val(3)又存了指向val(4)的地址，所以就构造出了一个链条访问结构。

我构造了一个例子**BasicLink**，我们debug一下看一下从head开始next会发现是这样的：



这就是一个简单的线性访问了，所以链表就是从head开始，逐个开始向后访问，而每次所访问对象的类型都是一样的。

创建链表的方式可以很简单，在LeetCode中算法题中经常使用这样的方式：

```
public class ListNode {
    public int val;
    public ListNode next;

    ListNode(int x) {
        val = x;
        //这个一般作用不大，写了会更加规范
        next = null;
    }
}
ListNode listnode=new ListNode(1);
```

这里的val就是当前结点的值，next指向下一个结点。因为两个变量都是public 的，创建对象后能直接使用listnode.val 和listnode.next来操作，虽然违背了面向对象的设计要求，但是代码更为精简，因此在算法题目中应用广泛。

而更加符合面向对象要求的定义是这样的：

```
public class ListNode {
    private int data;
    private ListNode next;
    public ListNode(int data) {
        this.data = data;
    }
    public int getData() {
        return data;
    }
    public void setData(int data) {
        this.data = data;
    }
    public ListNode getNext() {
```

```
        return next;
    }
    public void setNext(ListNode next) {
        this.next = next;
    }
}
```

这样的坏处是你不知道ListNode表示的是链表还是结点，因此在很多地方还会见到在链表类里再定义一个静态内部类来表示结点，也就是这样子:

```
public class LinkedListBasicUse {
 static class Node {
        final int data;
        Node next;
        public Node(int data) {
            this.data = data;
        }
    }
}
```

使用方法如下，一行就完成了很多工作：

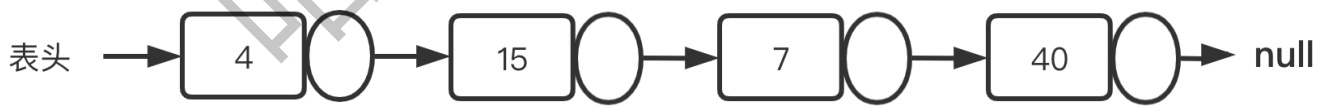
```
   LinkedListBasicUse.Node head = new Node(1);
```

上面介绍的这几种方式都经常见，一般题目会先将使用哪种方式定义的代码告诉你。我们接下来的增删改查操作就使用第二种方式进行。

1.2 遍历链表

完整代码请参考BasicLinkList

对于单链表，不管进行什么操作，一定是从头开始逐个向后访问，所以操作之后是否还能找到表头非常重要。一定要注意"狗熊掰棒子"问题，也就是只顾当前位置而将标记表头的指针丢掉了。




```
    /**
     * 获取链表长度
     * @param head 链表头节点
     * @return 链表长度
     */
    public static int getListLength(Node head) {
        int length = 0;
       Node node = head;
        while (node != null) {
            length++;
```

```
            node = node.next;
        }
        return length;
    }
```

单链表就是如上所示，从head开始沿着指针向前走，当next的值为null时停止。

1.3 链表插入

完整代码请参考BasicLinkList

单链表的插入，和数组的插入一样，过程不复杂，但是在编码时会发现处处是坑。和数组的插入一样，单链表的插入操作同样要考虑三种情况：首部、中部和尾部。

(1) 在链表的表头插入

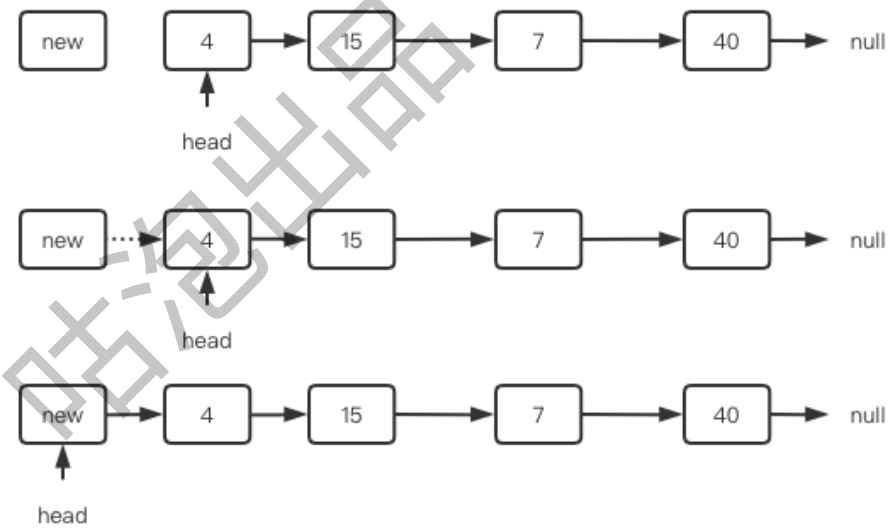
链表表头插入新结点非常简单，容易出错的是经常会忘了head需要重新指向。

我们创建一个新结点newNode，怎么连接到原来的链表上呢？应该是newNode.next=head是吧。之后我们要遍历新链表就要从newNode开始一路next向下了是吧，但是我们还是习惯让head来表示，那怎么办呢？让head=newNode是不是就行了？因此主要分两步：

1.修改结点的next指针，使其指向当前的表头结点head。

2.更新表头指针head，使其指向链表的首位置。

图示如下：



（2）在链表中间插入

在中间位置插入，我们必须先遍历找到要插入的位置，然后将当前位置 接入到前驱结点和后继结点之间，但是到了该位置之后我们却不能获得前驱结点了，因此无法将结点接入进来了。这就好比一边过河一边拆桥，结果自己也回不去了。

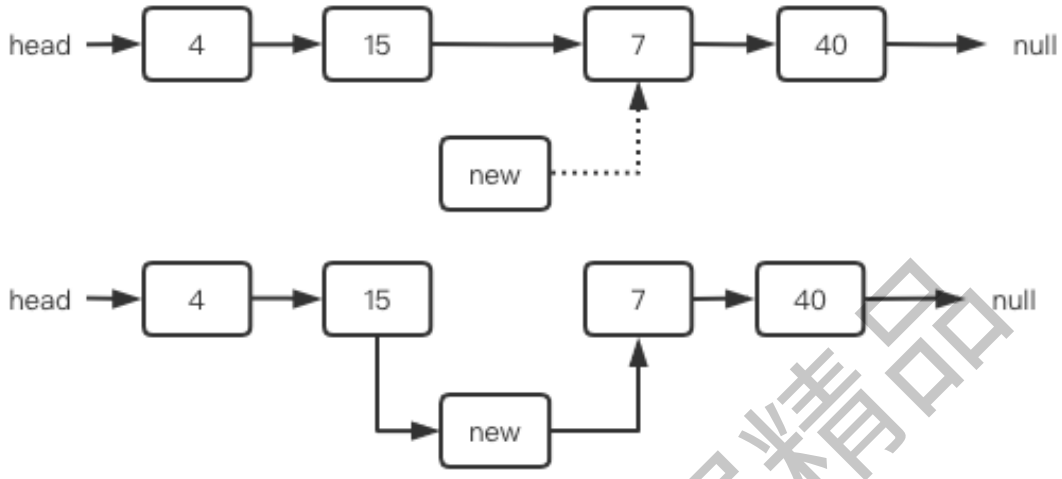
该怎么办呢？我们要在目标结点的前一个位置停下来，也就是判断cur.next是不是我们要插入的位置。

例如下面图示中，如果要在7的前面插入，当cur到达值为15的结点时，判断cur.next=node(7)了就应该停下来。

很显然new结点前后都要接入，也就是node(15).next=new，new.next=node(7)，这时候该先接哪一个呢？

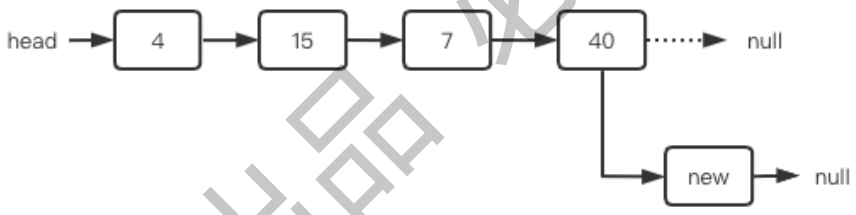
我们是通过node(15).next来定位node(7)的，如果如果先执行node(15).next=new就无法找到node(7)了，因此要先接后面。

先让new.next=node(15).next就将图中后面的虚线建立起来了，然后node(15).next=new就将新结点接入进来了。由于每个节点都只有一个next，因此执行了node(15).next=new之后，结点15和7之间的连线就自动断开了，如下图所示：



(3)在单链表的结尾插入结点

表尾插入就比较容易了，但是思想和上面中间位置是一样的。我们只要将尾结点指向新结点就行了。



综上，我们写出链表插入的方法如下所示，

这里需要再补充一点head = null的时候该执行什么操作呢？如果是null的话，你要插入的结点就是链表的头结点。你也可以认为链表是null所以直接抛出不能插入的异常，这两种都可以，也没有对错之分，一般来说我们更倾向前者。

下面的例子是根据指定的位置插入元素：

```
/**
     * 链表插入
     * @param head        链表头节点
     * @param nodeInsert 待插入节点
     * @param position    待插入位置，从1开始
     * @return 插入后得到的链表头节点
     */
    public static Node insertNode(Node head, Node nodeInsert, int position) {
        if (head == null) {
        //这里可以认为待插入的结点就是链表的头结点，也可以抛出不能插入的异常
            return nodeInsert;
        }
```

```
        //已经存放的元素个数
        int size = getLength(head);
        if (position > size+1  || position < 1) {
            System.out.println("位置参数越界");
            return head;
        }

        //表头插入
        if (position == 1) {
            nodeInsert.next = head;
            // 这里可以直接 return nodeInsert;还可以这么写:
            head = nodeInsert;
            return head;
        }

        Node pNode = head;
        int count = 1;
        //这里position被上面的size被限制住了，不用考虑pNode=null
        while (count < position - 1) {
            pNode = pNode.next;
            count++;
        }
        nodeInsert.next = pNode.next;
        pNode.next = nodeInsert;

        return head;
    }
```

在很多算法中，如果链表是单调的，一般会让你将元素插入后仍然保持单调。你可以尝试写一下该如何实现。

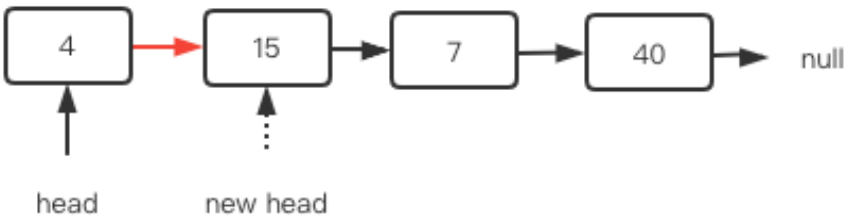
1.4 链表删除

完整代码请参考BasicLinkList

删除同样分为在删除头部元素，删除中间元素和删除尾部元素。

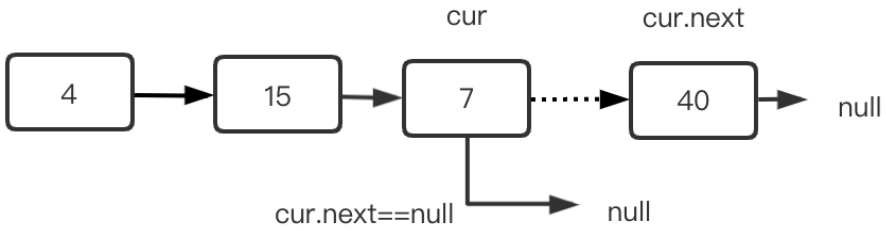
(1)删除表头结点

删除表头元素还是比较简单的，一般只要执行head=head.next就行了。如下图，将head向前移动一次之后，原来的结点不可达，会被JVM在某个时刻回收掉。



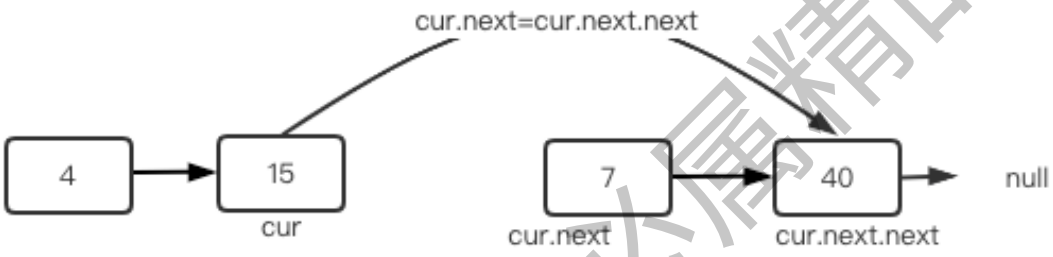
(2)删除最后一个结点

删除的过程不算复杂，也是找到要删除的结点的前驱结点，这里同样要在提前一个位置判断，例如下图中删除40，其前驱结点为7。遍历的时候需要判断cur.next是否为40，如果是，则只要执行cur.next=null即可，此时结点40变得不可达，最终会被JVM回收掉。



(3)删除中间结点

与尾结点类似，这里也要注意保存前驱结点。一旦找到要被删除的结点，将前驱结点next指针的值更新为被删除结点的next值。如下图所示：



我们可以写一个删除的方法了：

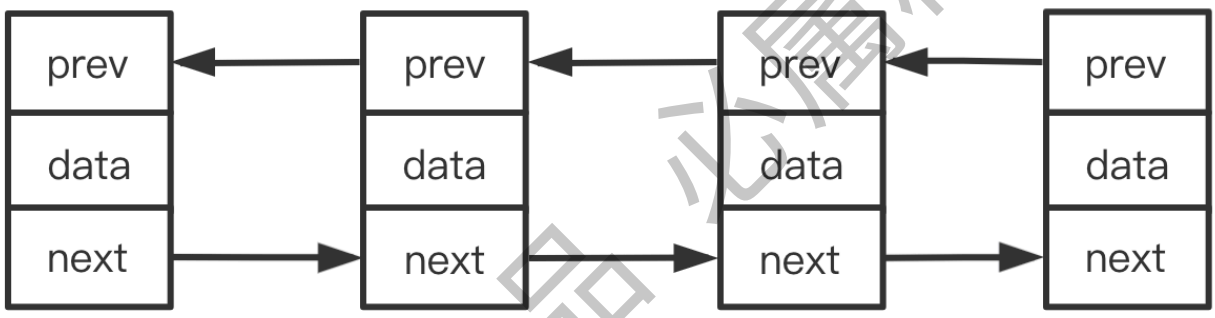
```
/**
     * 删除节点
     *
     * @param head     链表头节点
     * @param position 删除节点位置，取值从1开始
     * @return 删除后的链表头节点
     */
    public static Node deleteNode(Node head, int position) {
        if (head == null) {
            return null;
        }
        int size = getListLength(head);
        //思考一下，这里为什么是size，而不是size+1
        if (position > size || position <1) {
            System.out.println("输入的参数有误");
            return head;
        }
        if (position == 1) {
            //curNode就是链表的新head
            return head.next;
        } else {
            Node preNode = head;
```

```
            int count = 1;
            while (count < position - 1) {
                preNode = preNode.next;
                count++;
            }
            Node curNode = preNode.next;
            preNode.next = curNode.next;
        }
        return head;
    }
```

同样，在很多算法中链表是单调的，让你在删除元素之后仍然保持单调，也是一个算法问题，建议写一下试试。

1.5 双向链表简介

双向链表顾名思义就是既可以向前，也可以向后，这是与单向链表最大的区别。有两个指针的好处自然是在指针在中间位置的时候，操作更方便。该结构我们在LRU设计等问题中会遇到，所以这里简单看一下。

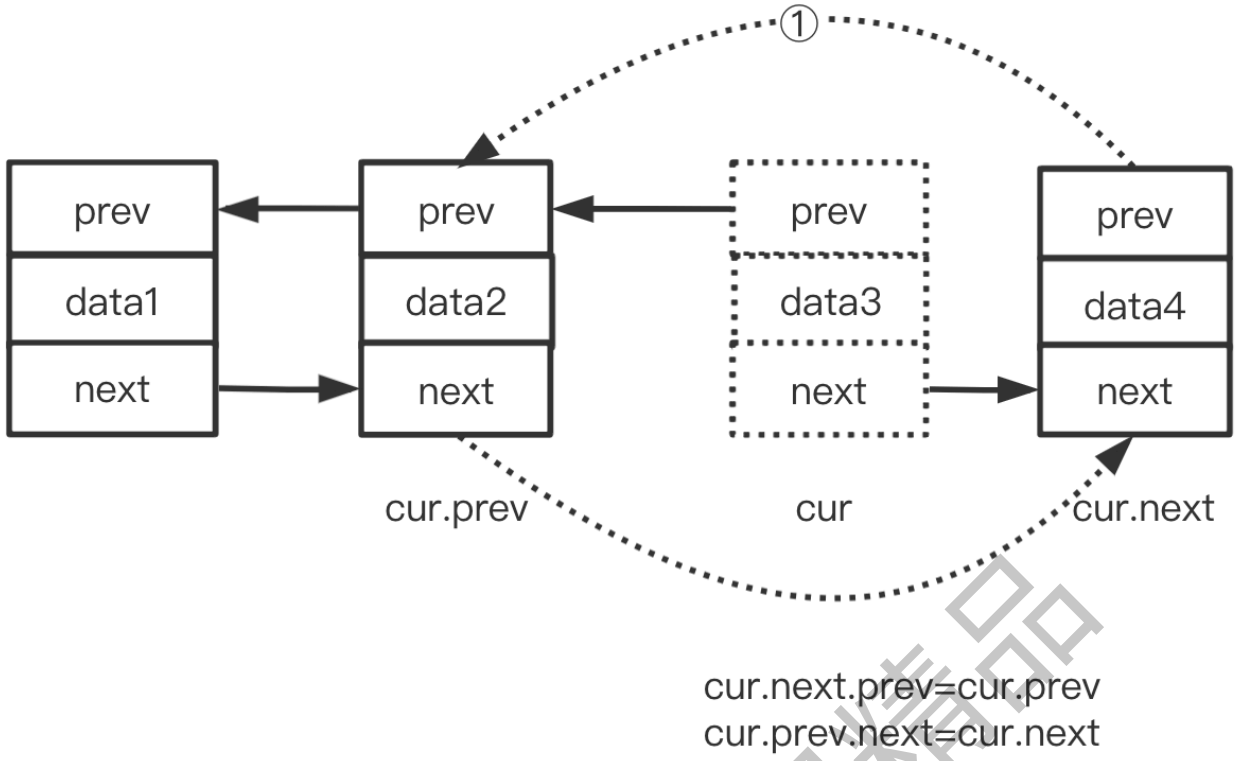



```
public class Node {
    public int data;    //数据域
    public Node next;    //指向下一个结点
    public Node prev;

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

当然坏处就是增删改的时候，需要修改的指针多了，操作更麻烦了。我们看一下插入和删除的大致过程:

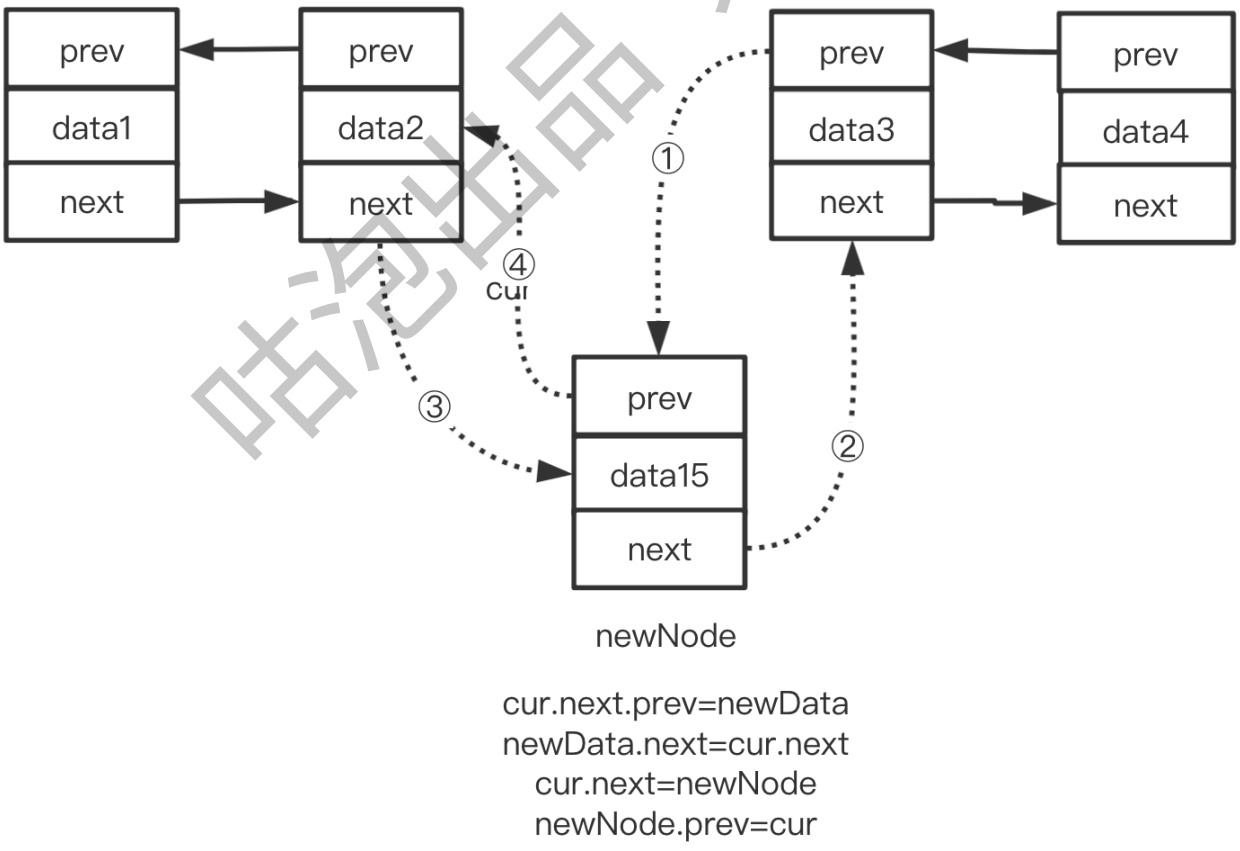
由于在算法测试中，双向链表不是重点，我们只看一下在中间位置删除的操作过程。

该操作与单向链表相似，首先标记出几个关键结点的位置。也就是图中的cur，cur.next和cur.next.next结点。由于在双向链表中，可以走回头路，所以我们使用cur，cur.next和cur.prev任意一个位置都能实现删除。假如我们就删除cur，则图示是这样的：



由于头结点和尾结点的指针情况不一样，所以即使使用了虚拟指针，还是要单独处理一下头部和尾部删除的情况，这个我们不再细谈。

我们再看一下在中间位置插入的情况：



虽然分析不麻烦，但是看上图的图还是比较繁琐的，而要手动实现也更为麻烦。我在alg工程中提供了一个完整实现的例子DoublyLinkList，感兴趣的可以调试一下试试。

与单链表相比，双向链表的一个重要功能是可以方便地进行结点的调整，这在AQS的优先级问题等场景中非常适用。不过单纯的双向链表排序实现非常复杂，而且效率也不高，我们一般会根据实际需要进行简化或者直接使用jdk封装过的类来实现。

2.高频面试题

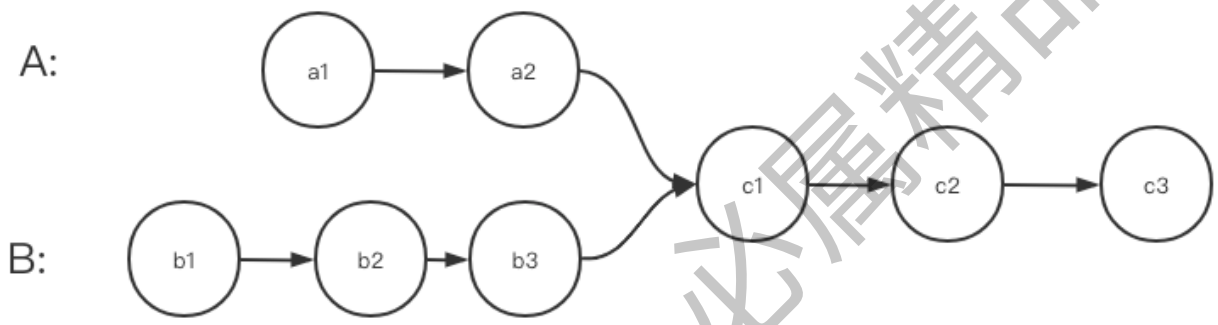
链表部分面试题很多都非常典型，但是与数组相比，题目要少很多，而在回溯贪心动规等高级算法中很少见到链表的影子。我们这里就按照专题学习一些出现频率特别高的算法题。

2.1 五种方法解决两个链表第一个公共子节点

这是一道经典的链表问题，先看一下题目。

输入两个链表，找出它们的第一个公共节点。

例如下面的两个链表：



两个链表的头结点都是已知的，相交之后成为一个单链表，但是相交的位置未知，并且相交之前的结点数也是未知的，请设计算法找到两个链表的合并点。

没有思路时该怎么解题

这种问题该怎么入手呢？如果一时想不到该怎么办呢？这时候我们可以将常用数据结构和常用算法思想都想一遍，看看哪些能解决问题。

常用的数据结构有数组、链表、队、栈、Hash、集合、树、堆。常用的算法思想有查找、排序、双指针、递归、迭代、分治、贪心、回溯和动态规划等等。

我们先在脑子里快速过一下谁有可能解决问题。首先想到的是蛮力法，类似于冒泡排序的方式，将第一个链表中的每一个结点依次与第二个链表的进行比较，当出现相等的结点指针时，即为相交结点，但是这种方法时间复杂度高，而且有可能只是部分匹配上，所以还有要处理复杂的情况。排除！

其次Hash呢？模模糊糊感觉行的。再想一下，先将第一个链表元素全部存到Map里，然后一边遍历第二个链表，一边检测当前元素是否在Hash中，如果两个链表有交点，那么一定能找到。

既然Hash可以，那集合呢？和Hash一样用，目测也能解决，OK，第二种方法。

队列和栈呢？貌似队列没啥用，但是栈能解决问题，现将两个链表分别保存到两个栈里，之后一边同时出栈，一边比较出栈元素是否一直，如果一致则说明存在相交，然后继续找，最晚一致的那组就是交点了，于是就有了第三种方法。

其他的几种结构或者算法呢？貌似都不太好用。这时候我们可以直接和面试官说，应该可以用HashMap做，另外集合和栈应该也能解决问题。面试官很明显就问了，怎么解决？

那这时候你可以继续说HashMap、集合和栈具体应该怎么解决。假如错了，比如你开始说队列也行，但是后面发现根本解决不了，这时候直接对面试官说“队列不行，我想想其他方法”，一般对方就不会再细究了。

算法面试本身也是一个相互交流的过程，如果有些地方你不清楚，他甚至会提醒你一下，所以不用紧张，也不用怕他盯着你写代码，努力去做就行了。

言归正传。

(1)HashMap法

先将一个链表元素全部存到Map里，然后一边遍历第二个链表，一边检测Hash中是否存在当前结点，如果有交点，那么一定能检测出来。 如果面试官点头，就可以写了：

```
import java.util.HashMap;
public class Solution {
    public ListNode findFirstCommonNodeByMap(ListNode pHead1, ListNode pHead2) {
     if(pHead1==null || pHead2==null){
             return null;
         }
        ListNode current1=pHead1;
        ListNode current2=pHead2;

        HashMap<ListNode,Integer>hashMap=new HashMap<>();
        while(current1!=null){
            hashMap.put(current1,null);
            current1=current1.next;
        }

        while(current2!=null){
            if(hashMap.containsKey(current2))
                return current2;
            current2=current2.next;
        }
        return null;
    }
}
```

(2) 集合Set法

能用Hash，那能不能用Set呢？其实思路和上面的一样，之间看代码

```
public ListNode findFirstCommonNodeBySet(ListNode headA, ListNode headB) {
    Set<ListNode> set = new HashSet<>();
    while (headA != null) {
        set.add(headA);
        headA = headA.next;
    }

    while (headB != null) {
```

```
        if (set.contains(headB))
            return headB;
        headB = headB.next;
    }
    return null;
}
```

(3) 使用栈

这里需要使用两个栈，分别将两个链表的结点入两个栈，然后分别出栈，如果相等就继续出栈，一直找到最晚出栈的那一组。这种方式需要两个O(n)的空间，所以在面试时不占优势，但是能够很好锻炼我们的基础能力，所以花十分钟写一个吧：

```
import java.util.Stack;
public class Solution {
    public ListNode findFirstCommonNodeByStack(ListNode headA, ListNode headB) {
        Stack<ListNode> stackA=new Stack();
        Stack<ListNode> stackB=new Stack();
        while(headA!=null){
            stackA.push(headA);
            headA=headA.next;
        }
        while(headB!=null){
            stackB.push(headB);
            headB=headB.next;
        }

      ListNode preNode=null;
      while(stackB.size()>0 && stackA.size()>0){
          if(stackA.peek()==stackB.peek()){
            preNode=stackA.pop();
             stackB.pop();
          }else{
              break;
          }
      }
        return preNode;
    }
}
```


看到了吗，从一开始没啥思路到最后搞出三种方法，熟练掌握数据结构是多么重要！！

(4)拼接两个字符串

如果你想到了这三种方法中的两个，并且顺利手写并运行出一个来，面试基本就过了，至少面试官对你的基本功是满意的。但是对方可能会再来一句：还有其他方式吗？或者说，有没有申请空间大小是O(1)的方法。方法是有的，但是需要一些技巧，而这种技巧普适性并不强，我们还是看一下吧。

先看下面的链表A和B：

A: 0-1-2-3-4-5

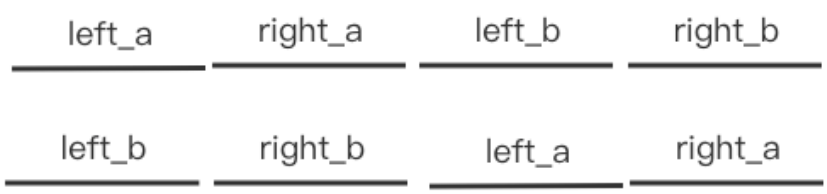
B: a-b-4-5

如果分别拼接成AB和BA会怎么样呢?

AB：0-1-2-3-4-5-a-b-4-5

BA：a-b-4-5-0-1-2-3-4-5

我们发现最后从4开始的就是公共子节点，所以可以通过拼接的方式来寻找交点，按这么做的道理是什么呢？我们可以从几何的角度来分析。我们假定A和B有相交的位置，以交点为中心，可以将两个链表分别分为left_a和right_a，left_b和left_b这样四个部分，并且right_a和right_b是一样的，这时候我们拼接AB和BA就是这样的结构：



我们说right_a和right_b是一样的，那这时候分别遍历AB和BA是不是从某个位置开始恰好就找到了相交的点了?

这里还可以进一步优化，如果建立新的链表太浪费空间了，我们只要在每个链表访问完了之后，调整到一下链表的表头继续遍历就行了，于是代码就出来了：

```
public ListNode findFirstCommonNode(ListNode pHead1, ListNode pHead2) {
         if(pHead1==null || pHead2==null){
             return null;
         }
        ListNode p1=pHead1;
        ListNode p2=pHead2;
        while(p1!=p2){
            p1=p1.next;
            p2=p2.next;
            if(p1!=p2){
            //一个链表访问完了就跳到另外一个链表继续访问
                if(p1==null){
                    p1=pHead2;
                }
                if(p2==null){
                    p2=pHead1;
                }
            }
        }
        return p1;
    }
```

这种方式使用了两个指针，姑且算作双指针吧，我们再看一个使用差来解决问题的方法。

(5) 差和双指针

假如公共子节点一定存在第一轮遍历，假设La长度为L1，Lb长度为L2.则｜L2-L1｜就是两个的差值。第二轮遍历，长的先走｜L2-L1｜,然后两个链表同时向前走，结点一样的时候就是公共结点了。

```
public ListNode findFirstCommonNode(ListNode pHead1, ListNode pHead2) {
     if(pHead1==null || pHead2==null){
             return null;
         }
        ListNode current1=pHead1;
        ListNode current2=pHead2;
        int l1=0,l2=0;
        while(current1!=null){
            current1=current1.next;
            l1++;
        }

         while(current2!=null){
            current2=current2.next;
            l2++;
        }
        current1=pHead1;
        current2=pHead2;

        int sub=l1>l2?l1-l2:l2-l1;

       if(l1>l2){
           int a=0;
           while(a<sub){
            current1=current1.next;
            a++;
        }
       }

       if(l1<l2){
           int a=0;
           while(a<sub){
            current2=current2.next;
            a++;
        }
       }

       while(current2!=current1){
          current2=current2.next;
          current1=current1.next;
       }

        return current1;
    }
```

一个普通的算法，我们整出来5种方法， 就相当于做了五道题，但是思路比单纯做5道题更加开阔， 下个题我们继续练习这种思路。

2.2 六种判断链表是否为回文序列

这也是一道不难但是很经典的链表题，请判断一个链表是否为回文链表。

```
示例1:
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?
```

看到这个题你有几种思路解决，经过前面这些题目的蹂躏，我现在看到这个题瞬间就想到6种解法。虽然有几个是相同的，但是仍然可以是新的方法。

方法1：将链表元素都赋值到数组中，然后可以从数组两端向中间对比。

方法2：将链表元素全部压栈，然后一边出栈，一边重新遍历链表，一边比较，只要有一个不相等，那就不是回文链表了。

方法3：上面方法的改造，先遍历第一遍，得到总长度。之后一边历链表，一边压栈。当到达链表长度一半的位置之后，就不再压栈，而是一边出栈，一边遍历，一边比较，只要有一个不相等，就不是回文链表。

方法4：反转链表法， 先创建一个链表newList，然后原始链表oldList的元素值逆序保存到newList中，然后重新遍历newList和oldList，同时比较元素的值，只要有一个位置的元素值不一样，就不是回文链表。

方法5：将4进行优化，我们其实只反转一半的元素就行了。先遍历一遍链表，得到长度，然后重新遍历链表，一边遍历，一边将链表反转。到达一半的位置后，就不再反转，而是比较两个链表，只要有一个元素不一样，就不是回文链表。

方法6：还是对4的改进，我们使用快慢指针 ，fast一次走两步，slow一次走一步。当fast到达表尾的时候，slow正好到达一半的位置，那么接下来可以从头开始逆序一半的元素，或者从slow开始逆序一半的元素，都可以。

方法7：假如使用递归法等，我们还能想出更多的方法，解决问题的思路不止7种，但是单纯增加数量没啥意义了。

上面这些解法中，方法1即使你说了，面试官一般也不会让你写，因为就是想考你链表，偷懒不行。方法2和3，如果能写出来，算良好。使用反转虽然能解决问题，但是反转本身就是一个很重量级的操作，面试的时候如果面试官不要求，就不要写（别给自己找麻烦）。

(1) 使用栈:全部压栈

将链表元素全部压栈，然后一边出栈，一边重新遍历链表，一边比较，只要有一个不相等，那就不是回文链表了。代码：

```
public boolean isPalindrome(ListNode head) {
    ListNode temp = head;
    Stack<Integer> stack = new Stack();
```

```
    //把链表节点的值存放到栈中
    while (temp != null) {
        stack.push(temp.val);
        temp = temp.next;
    }
    //然后再出栈
    while (head != null) {
        if (head.val != stack.pop()) {
            return false;
        }
        head = head.next;
    }
    return true;
}
```

(2) 使用栈:部分压栈

改造上面的方法，先遍历第一遍，得到总长度。之后一边遍历链表，一边压栈。当到达链表长度一半的位置之后，就不再压栈，而是一边出栈，一边遍历，一边比较，只要有一个不相等，就不是回文链表。代码就是这样：

```
public boolean isPalindrome(ListNode head) {
    if (head == null)
        return true;
    ListNode temp = head;
    Stack<Integer> stack = new Stack();
    //链表的长度
    int len = 0;
    //把链表节点的值存放到栈中
    while (temp != null) {
        stack.push(temp.val);
        temp = temp.next;
        len++;
    }
    //len长度除以2
    len >>= 1;
    //然后再出栈
    while (len-- >= 0) {
        if (head.val != stack.pop())
            return false;
        head = head.next;
    }
    return true;
}
```

(3)拓展1 快慢指针+一半反转法

这个实现略有难度，主要是在while循环中pre.next = prepre;和prepre = pre;实现了一边遍历一边将访问过的链表给反转了，所以理解起来有些难度，可以在学完链表反转之后再看这个问题。

```
    public boolean isPalindrome(ListNode head) {
        if(head == null || head.next == null) {
            return true;
        }
        ListNode slow = head, fast = head;
        ListNode pre = head, prepre = null;
        while(fast != null && fast.next != null) {
            pre = slow;
            slow = slow.next;
            fast = fast.next.next;
            //将前半部分链表反转
            pre.next = prepre;
            prepre = pre;
        }
        if(fast != null) {
            slow = slow.next;
        }
        while(pre != null && slow != null) {
            if(pre.val != slow.val) {
                return false;
            }
            pre = pre.next;
            slow = slow.next;
        }
        return true;
    }
```

(3)拓展2 递归法

递归的过程我们后面有一章单独解释，这里先简单看一下。考虑一个问题，如果用递归能否对链表逆序打印呢？有的，而且这个本身就是一个可以考察的算法题，可以这样写：

```
private void printListNode(ListNode head) {
    if (head == null)
        return;
    printListNode(head.next);
    System.out.println(head.val);
}
```

这里的过程就是先递归走到链表最后，再逐个打印就行，递归帮助我们实现了逆序的功能。看到这里是不是有灵感了，我们来对上面的对面进行改造一下：

```
ListNode temp;
public boolean isPalindrome(ListNode head) {
    temp = head;
    return check(head);
}
private boolean check(ListNode head) {
    if (head == null)
        return true;
    boolean res = check(head.next) && (temp.val == head.val);
    temp = temp.next;
    return res;
}
```

上面的temp就是反转之后的链表。除此之外，还可以单独写个链表反转等方法，但是有点大材小用了，我们就不写了。

2.3 合并有序链表

学完这个专题之后，你会更加相信LeetCode就是在造题。数组中我们研究过合并的问题，链表同样可以造出两个或者多个链表合并的问题。

2.3.1 合并两个有序链表

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

这种题一般有两种要求，一种是新建一个链表，另外一个就是将一个链表结点拆下来，逐个合并到另外一个对应位置上去。这个过程本身就是链表插入和删除操作的拓展，难度不算大，这时候代码是否优美就比较重要了。先看这个中规中矩的实现方式：

```
public ListNode mergeTwoLists (ListNode list1, ListNode list2) {
     ListNode newHead=new ListNode(-1);
     ListNode res=newHead;
     while(list1!=null||list2!=null){
         //情况1：都不为空的情况
         if(list1!=null&&list2!=null){
             if(list1.val<list2.val){
                 newHead.next=list1;
                 list1=list1.next;
             }else if(list1.val>list2.val){
                 newHead.next=list2;
                 list2=list2.next;
             }else{ //相等的情况，分别接两个链
                 newHead.next=list2;
                 list2=list2.next;
                 newHead=newHead.next;
                 newHead.next=list1;
                 list1=list1.next;
             }
             newHead=newHead.next;
```

```
            //情况2：假如还有链表一个不为空
         }else if(list1!=null&&list2==null){
             newHead.next=list1;
             list1=list1.next;
             newHead=newHead.next;
         }else if(list1==null&&list2!=null){
             newHead.next=list2;
             list2=list2.next;
             newHead=newHead.next;
         }
     }
     return res.next;
 }
```

上面这种方式能完成基本的功能，但是所有的处理都在一个大while循环里，代码过于臃肿，我们可以将其拆开，第一个while只处理两个list 都不为空。之后单独写while分别处理list1或者list2不为null的情况。也就是这样：

```
public  ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode newHead = new ListNode(-1);
        ListNode res = newHead;
        while (list1 != null && list2 != null) {

            if (list1.val < list2.val) {
                newHead.next = list1;
                list1 = list1.next;
            } else if (list1.val > list2.val) {
                newHead.next = list2;
                list2 = list2.next;
            } else { //相等的情况，分别接两个链
                newHead.next = list2;
                list2 = list2.next;
                newHead = newHead.next;
                newHead.next = list1;
                list1 = list1.next;
            }
            newHead = newHead.next;

        }
        //下面的两个while最多只有一个会执行
        while (list1 != null) {
            newHead.next = list1;
            list1 = list1.next;
            newHead = newHead.next;
        }
        while (list2 != null) {
            newHead.next = list2;
            list2 = list2.next;
            newHead = newHead.next;
        }
```

```
        return res.next;
}
```

拓展1 进一步精简代码

进一步分析，我们发现两个继续精简的点，一个是list和list2的合并也可以简化，如果两个链表存在相同元素，第一次出现时使用下面的if (l1.val <= l2.val)来处理，后面一次则会被else处理掉，什么意思呢？我们看一个序列。

假如list1为{1, 5, 8, 12}，list2为{2, 5, 9, 13}，此时都有一个node(5)。当两个链表都到5的位置时，出现了list1.val == list2.val，此时list1中的node(5)会被合并进来。

然后list1继续向前走到了node(8)，此时list2还是node(5)，因此就会执行else中的代码快。

这样就可以将第一个while的代码进行精简。

第二个优化是后面两个小的while循环，这两个while最多只有一个会执行，而且由于链表只要将头接好，后面的自然就接上了，因此直接将剩下的不为空的链表接到新链表的后面就行了，循环都不用写，也就是这样：

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode prehead = new ListNode(-1);
        ListNode prev = prehead;
        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
                prev.next = list1;
                list1 = list1.next;
            } else {
                prev.next = list2;
                list2 = list2.next;
            }
            prev = prev.next;
        }
        // 最多只有一个还未被合并完，直接接上去就行了,这是链表合并比数组合并方便的地方
        prev.next = list1 == null ? list2 : list1;
        return prehead.next;
    }
}
```

这种方式很明显更高级，但是面试时很难考虑周全，如果面试的时候遇到了，建议先用上面第一种写法写出来，面试官满意之后，接着说“我还可以用更精简的方式来解决这个问题”。然后再写第二种，这样不用怕翻车，还可以锦上添花。、

拓展2:能够给面试官下马威的写法

递归我们后面还会分析，这里也是先简单看一下思想。大部分面试官可能也只知道前面的写法，不见得一下子明白如何通过递归来做，我们继续分析。

当我们将list1或者list2中拿到最小的那个结点之后，剩下的不管是list1还是list2都还是完整的链表，而且执行一样的判断逻辑，这非常符合递归的两个条件：处理方式一样，问题规模在缩小，所以我们就可以通过递归的方式来继续处理：

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }
         if (l2 == null) {
            return l1;
        }
        else if (l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        }
        else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}
```

当然这种方式有些难度的，只有熟练之后在面试时才能写出来，否则还是按照前面基本的方式比较稳妥。

2.3.2 合并K个链表

合并k个链表，有多种方式，如果面试，我倾向的方式是先将前两个合并，之后再将后面的逐步合并进来，因为这样的话，我只要将两个合并的写清楚就行了，也就是直接调用二中的方法：

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        ListNode res = null;
        for (ListNode list: lists) {
            res = mergeTwoLists(res, list);
        }
        return res;
    }
}
```

2.3.3 一道很无聊的好题

LeetCode1669题：给你两个链表 list1 和 list2 ，它们包含的元素分别为 n 个和 m 个。请你将 list1 中第 a 个节点到第 b 个节点删除，并将list2 接在被删除节点的位置。

1669题的意思就是将list1中的[a,b)区间的删掉，然后将list2接进去，你觉得难吗？如果这也是算法的话，我至少可以造出七八道题，例如：

（1）定义list1的[a,b)区间为list3，将list3和list2按照升序合并成一个链表。
（2）将list2也将区间[a,b)的元素删掉，然后将list1和list2合并成一个链表。
（3）定义list2的[a,b)区间为list4，将list2和list4合并成有序链表。

注[a,b)就是左闭右开区间，意思是左侧的a属于这个区间，而b是属于下一个区间的意思。

看到了吗？掌握基础是多么重要，我们自己都能造出题目来。
这也是为什么算法会越刷越少，因为到后面会发现套路就这么写，花样随便换，以不变应万变就是我们的宗旨。

不仅仅造题，链表合并我们还有五六种方法解决，最后拓展部分我们再介绍

遍历找到链表1保留部分的尾节点和链表2的尾节点，将两链表连接起来。

```
class Solution {
    public ListNode mergeInBetween(ListNode list1, int a, int b, ListNode list2) {
        ListNode pre1 = list1, post1 = list1, post2 = list2;
        int i = 0, j = 0;
        while(pre1 != null && post1 != null && j < b){
            if(i != a - 1){
                pre1 = pre1.next;
                i++;
            }
            if(j != b){
                post1 = post1.next;
                j++;
            }
        }
        post1 = post1.next;
        //寻找list2的尾节点
        while(post2.next != null){
            post2 = post2.next;
        }
        //链1尾接链2头，链2尾接链1后半部分的头
        pre1.next = list2;
        post2.next = post1;
        return list1;
    }
}
```


2.4 双指针专题

在数组里我们介绍过双指针的思想， 可以简单有效的解决很多问题，而所谓的双指针只不过是两个变量而已。在链表中同样可以使用双指针来轻松解决一部分算法问题。这类题目的整体难度不大，除了在链表中找环的入口问题。

2.4.1 寻找中间结点

```
题目要求：
给定一个头结点为 head 的非空单链表，返回链表的中间结点。
如果有两个中间结点，则返回第二个中间结点。

示例1
输入：[1,2,3,4,5]
输出：此列表中的结点 3
示例2：
输入：[1,2,3,4,5,6]
输出：此列表中的结点 4
```

这个问题用经典的快慢指针可以轻松搞定，用两个指针 `slow` 与 `fast` 一起遍历链表。`slow` 一次走一步，`fast` 一次走两步。那么当 `fast` 到达链表的末尾时，`slow` 必然位于中间。

这里还有个问题，就是偶数的时候该返回哪个，例如上面示例2返回的是4， 而3貌似也可以，那该使用哪个呢？如果我们使用标准的快慢指针就是后面的4，而在很多数组问题中会是前面的3，所以这里知道有这么回事就行，没必要较真。

```
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

2.4.2寻找倒数第K个元素

这也是经典的快慢双指针问题，先看要求：

```
输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。

示例
给定一个链表: 1->2->3->4->5, 和 k = 2.
返回链表 4->5.
```

使用快慢双指针，我们先将fast 向后遍历到达第 k+1 个节点， slow仍然指向链表的第一个节点，此时指针fast 与 slow 二者之间刚好间隔 k 个节点。之后两个指针同步向后走，当第一个指针 fast 走到链表的尾部空节点时，slow 指针刚好指向链表的倒数第k个节点。

这里需要特别注意的是**链表的长度可能小于k**，寻找k位置的时候必须同时判断fast是否为null了，这是本题的关键。所以最终的代码如下：

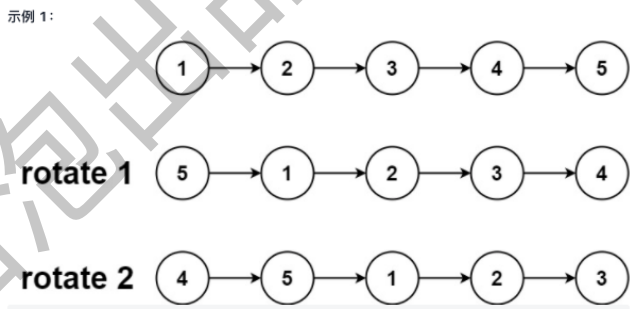
```
class Solution {
```

```
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode fast = head;
        ListNode slow = head;

        while (fast != null && k > 0) {
            fast = fast.next;
            k--;
        }
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }

        return slow;
    }
}
```

2.4.3 旋转链表

先看题目要求：

```
给你一个链表的头节点 head ，旋转链表，将链表每个节点向右移动 k 个位置。

示例1，示意图如下
输入：head = [1,2,3,4,5], k = 2
输出：[4,5,1,2,3]
```



这个题有多种解决思路，首先想到的是根据题目要求硬写，但是这样比较麻烦，也容易错。

观察链表调整前后的结构，我们可以发现从旋转位置开始，链表被分成了两条，例如上面的{1,2,3}和{4,5}，这里我们可以参考上一题的倒数K的思路，找到这个位置，然后将两个链表调整一下重新接起来就行了。具体怎么调整呢？脑子里瞬间想到两种思路：

第一种是将整个链表反转变成{5,4,3,2,1}，然后再将前K和N-K两个部分分别反转，也就是分别变成了{4,5}和{1,2,3}，这样就轻松解决了。这个在后面学习了链表反转之后，请读者自行解决。

第二种思路就是先用双指针策略找到倒数K的位置，也就是{1,2,3}和{4,5}两个序列，之后再将两个链表拼接成{5,4,3,2,1}就行了，

具体可以这么做：

因为k有可能大于链表长度，所以首先获取一下链表长度len。如果k % len == 0，等于不用旋转，直接返回头结点。否则：

1.快指针先走k步。

2.慢指针和快指针一起走。

3.快指针走到链表尾部时，慢指针所在位置刚好是要断开的地方。把快指针指向的节点连到原链表头部，慢指针指向的节点断开和下一节点的联系。

4.返回结束时慢指针指向节点的下一节点。

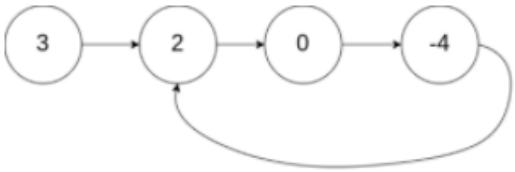
```
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if(head == null || k == 0){
            return head;
        }
        ListNode temp = head;
        ListNode fast = head;
        ListNode slow = head;
        int len = 0;
        while(head != null){
            head = head.next;
            len++;
        }
        if(k % len == 0){
            return temp;
        }
        while((k % len) > 0){
            k--;
            fast = fast.next;
        }
        while(fast.next != null){
            fast = fast.next;
            slow = slow.next;
        }
        ListNode res = slow.next;
        slow.next = null;
        fast.next = temp;
        return res;
    }
}
```


2.4.4 判断链表中是否存在环

这个题目同样特别经典：给定一个链表，判断链表中是否有环。进一步，假如有环，那环的位置在哪里？

这个问题前一问相对容易一些，后面一问比较难想到。但是吧，假如面试遇到第一问了，面试官很可能会问第二个，因为谁都知道有这个一个进阶问题。就像你和女孩子表白成功后，你会忍不住进阶一下——“亲一个呗”，一样的道理，所以我们需要都会。

```
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。
```



判断是否有环，最容易的方法是使用Hash，遍历的时候将元素放入到map中，如果有环一定会发生碰撞。发生碰撞的位置也就是入口的位置，因此这个题so easy。如果在工程中，我们这么做就OK了。

代码如下：

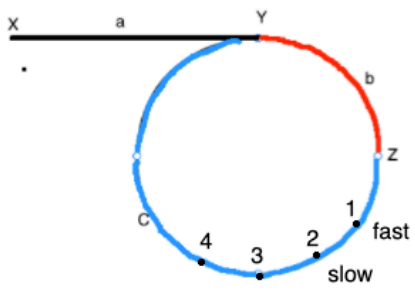
```
public ListNode detectCycle(ListNode head) {
        ListNode pos = head;
        Set<ListNode> visited = new HashSet<ListNode>();
        while (pos != null) {
            if (visited.contains(pos)) {
                return pos;
            } else {
                visited.add(pos);
            }
            pos = pos.next;
        }
        return null;
    }
```

但是如果只有 O(1)的空间该怎么做呢？我们必须逐步讨论了。

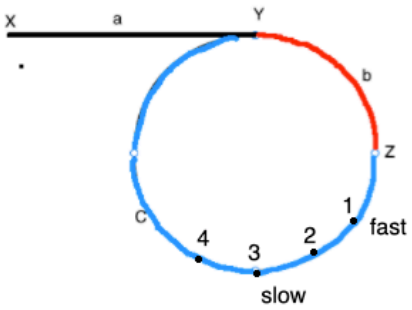
首先看如何确定是否有环，最有效的方法就是双指针，一个快指针（一次走两步），一个慢指针（一次走一步）。如果快的能到达表尾就不会有环，否则如果存在圈，则慢指针一定会在某个位置与快指针相遇。这就像在操场长跑，一个人快一个人慢，只要时间够，快的一定能在某个时候再次追上慢的人(也就是所谓的套圈)。

这里还有人可能会有疑问，因为两者每次走的距离不一样，会不会快的人在追上慢的人之后跳过去了导致两者不会相遇呢?

不会！如下图所示，当fast快要追上slow的时候，fast一定距离slow还有一个空格，或者两个空格，不会有其他情况。



情况1：两者相距一个空格



情况2：两者相距两个空格

假如有一个空格，如上图情况1所示，fast和slow下一步都到了3号位置，因此就相遇了。

假如有两个空格，如上图情况2所示，fast下一步到达3，而slow下一步到达4，这就变成了情况1了，因此只要有环，一定会相遇。

代码：

```
public boolean hasCycle(ListNode head) {
       if(head==null || head.next==null){
           return false;
       }
        ListNode fast=head;
        ListNode slow=head;
        while(fast!=null && fast.next!=null){
            fast=fast.next.next;
            slow=slow.next;
            if(fast==slow)
                return true;
        }
        return false;
    }
```

这里的问题是如果知道了一定有入口，那么如何确定入口的位置呢?

如果不使用双指针，而直接使用hash，则分分钟搞定这个题目。然而这样的话题目就没有思维含量了，所以如果工程中遇到这个问题就用Hash，但是算法面试中却不行。

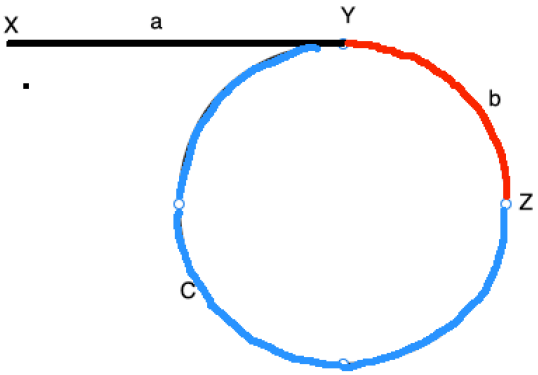
先说结论

先按照快慢方式寻找到相遇的位置（假如为下图中Z），然后将两指针分别放在链表头（X）和相遇位置（Z），并改为相同速度推进，则两指针在环开始位置相遇（Y）。

结论很简单，但这是为什么呢?

①先看一个简单的场景

为了便于理解，我们首先假定快指针在第二次进入环的时候就相遇了：



此时的过程是：

1.找环中相汇点。分别用fast、slow表示快慢指针，slow每次走一步，fast就走两步，直到在环中的某个位置相会，假如是图中的Z。

2.第一次相遇：

那么我们可以知道fast指针走了a+b+c+b步，

slow指针走了a+b步

那么：

2*(a+b) = a+b+c+b

所以a = c

因此此时让slow从Z继续向前走，fast回到起点，两个同时开始走（两个每次都走一步），一次走一步那么它们最终会相遇在y点，正是环的起始点。
② 普适场景

如果是普通场景会怎么样呢?

设链表中环外部分的长度为 a。slow 指针进入环后，又走了 b 的距离与 fast 相遇。此时，fast 指针已经走完了环的 n 圈，因此它走过的总距离为

Fast: a+n(b+c)+b=a+(n+1)b+nc

根据题意，任意时刻，fast 指针走过的距离都为 slow 指针的 2 倍。因此，我们有

a+(n+1)b+nc=2(a+b)

也就是：a=c+(n−1)LEN

由于b+c就是环的长度，假如为LEN，则：

a=c+(n-1)LEN

这说明什么呢？说明相遇的时候快指针在环了已经转了(n-1)LEN圈，如果n-1就退化成了我们上面说的一圈的场景。假如n是2，3，4，...呢，这只是说明当一个指针p1重新开始从head走的时候，另一个指针p2从Z点开始，两者恰好在入口处相遇，只不过p2要先在环中转n-1圈。

当然上面的p1和p2要以相同速度，我们发现slow和fast指针在找到位置Z之后就没有作用了，因此完全可以用slow和fast来代表p1和p2。因此代码如下：

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if (head == null) {
            return null;
        }
        ListNode slow = head, fast = head;
        while (fast != null) {
            slow = slow.next;
            if (fast.next != null) {
                fast = fast.next.next;
            } else {
                return null;
            }
            if (fast == slow) {
                ListNode ptr = head;
                while (ptr != slow) {
                    ptr = ptr.next;
                    slow = slow.next;
                }
                return ptr;
            }
        }
        return null;
    }
}
```

2.5 删除链表元素专题

2.5.1 又开始造题了

如果按照LeetCode顺序一道道刷题，会感觉毫无章法而且很慢，但是将相关类型放在一起，你瞬间发现这不就是在改改条件不断造题吗？我们前面已经多次见证这个情况，现在集中看一下与链表删除相关的问题。如果在链表中删除元素搞清楚了，一下子就搞定8道题，是不是很爽？一下子看到这么多题目你可能会蒙，但是没关系，我们后面会有具体的解析。

【1】LeetCode 237：删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为要被删除的节点 。

【2】LeetCode 203：给你一个链表的头节点 head 和一个整数 val ，请你删除链表中所有满足 Node.val == val 的节点，并返回新的头节点 。

【3】LeetCode 19. 删除链表的倒数第 N 个节点

【4】LeetCode 1474. 删除链表 M 个节点之后的 N 个节点。

【5】LeetCode 83 存在一个按升序排列的链表，请你删除所有重复的元素，使每个元素只出现一次。

【6】LeetCode 82 存在一个按升序排列的链表，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中没有重复出现的数字。

【7】LeetCode 1836. 从未排序链表中删除重复元素。

下面这个题目比较特殊，我们在后面分析nSum问题的时候统一来看：

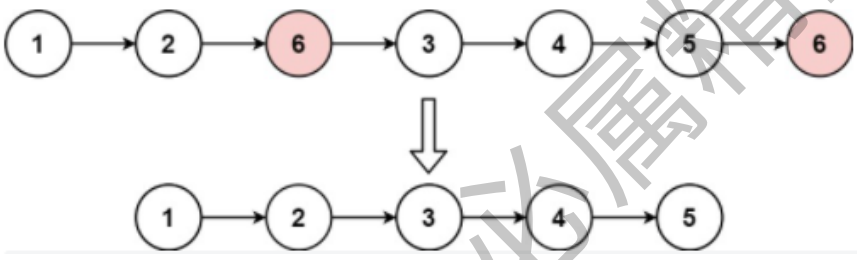
【8】LeetCode 1171 请你编写代码，反复删去链表中由总和值为 0 的连续节点组成的序列，直到不存在这样的序列为止。

我们在链表基本操作部分介绍了删除的方法，至少需要考虑删除头部，删除尾部和中间位置三种情况的处理。而上面这些题目就是这个删除操作的进一步拓展。

2.5.2 删除特定结点

先看一个简单的问题：

```
给你一个链表的头节点 head 和一个整数 val ，请你删除链表中所有满足 Node.val == val 的节点，并返回新的头节点
示例1：
输入：head = [1,2,6,3,4,5,6], val = 6
输出：[1,2,3,4,5]
```



我们前面说过，我们删除节点cur时，必须知道其前驱pre节点和后继next节点，然后让pre.next=next。这时候cur就脱离链表了，cur节点会在某个时刻被gc回收掉。

对于删除，我们注意到首元素的处理方式与后面的不一样。为此，我们可以先创建一个虚拟节点 dummyHead，使其指向head，也就是dummyHead.next=head，这样就不用单独处理首节点了。

完整的步骤是：

- 1.我们创建一个虚拟链表头dummyHead，使其next指向head。
- 2.开始循环链表寻找目标元素，注意这里是通过cur.next.val来判断的。
- 3.如果找到目标元素，就使用cur.next = cur.next.next;来删除。
- 4.注意最后返回的时候要用dummyHead.next，而不是dummyHead。

代码实现过程：

```
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummyHead = new ListNode(0);
        dummyHead.next = head;
        ListNode cur = dummyHead;
        while (cur.next != null) {
            if (cur.next.val == val) {
                cur.next = cur.next.next;
            } else {
                cur = cur.next;
            }
```

```
        }
        return dummyHead.next;
    }
}
```

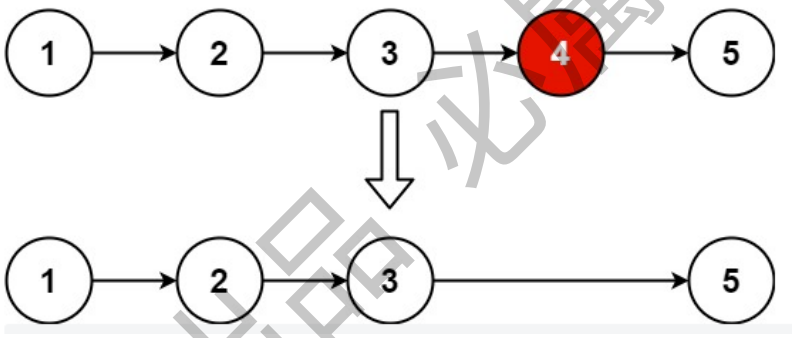
我们继续看下面这两个题，其实就是一个题对不?

LeetCode 19. 删除链表的倒数第 N 个节点

LeetCode 1474. 删除链表 M 个节点之后的 N 个节点。

既然要删除倒数第N个节点，那一定要先找到倒数第N个节点，这本身就是一道算法题，我们前面已经介绍过，而这里的删除不过是找到位置之后将其删除。我们重点看一下19题，1474题我们就不看了。LeetCode19题具体题目要求：

```
给你一个链表，删除链表的倒数第n个结点，并且返回链表的头结点。
进阶：你能尝试使用一趟扫描实现吗?
示例1：
输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]
```



分析

我们前面说过，遇到一个题目可以先在脑子里快速过一下常用的数据结构和算法思想，看看哪些看上去能解决问题。为了开拓思维，我们看看能怎么做：

第一种方法：先遍历一遍链表，找到链表总长度L，然后重新遍历，位置L-N+1的元素就是我们要删的。

第二种方法：貌似栈可以，先将元素全部压栈，然后弹出第N个的时候就是我们要的是不？OK，又搞定一种方法。

第三种方法：我们前面提到可以使用双指针 来寻找倒数第K，那这里同样可以用来寻找要删除的问题。

到此为止，我们的方案就有三种了，可以看到第一种方法比较常规，但是需要遍历两次链表，第二种方法需要开辟一个O(n)的空间，而且还要考虑栈顺序与链表顺序的关系，不中看也不中用，所以不管了。而第三种方法一次遍历就行，所以我们一般倾向第三种。接下来我们详细看一下第一和三两种。

方法1：计算链表长度

虽然这种方式不是最优的，但是面试的时候心一慌，手一凉，还是最不费脑的方式最靠谱，所以我们具体看一下如何实现。

首先从头节点开始对链表进行一次遍历，得到链表的长度 L。随后我们再从头节点开始对链表进行一次遍历，当遍历到第L−n+1 个节点时，它就是我们需要删除的节点。

确实能解决问题，在很多场景下也能用，所以我们先练一下，至少是个及格分

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next=head;
        int length = getLength(head);
        ListNode cur = dummy;
        for (int i = 1; i < length - n + 1; ++i) {
            cur = cur.next;
        }
        cur.next = cur.next.next;
        ListNode ans = dummy.next;
        return ans;
    }

    public int getLength(ListNode head) {
        int length = 0;
        while (head != null) {
            ++length;
            head = head.next;
        }
        return length;
    }
}
```

方法二： 双指针

我们定义first和second两个指针，first先走N步，然后second再开始走，当first走到队尾的时候，second就是我们要的节点对不?

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
       ListNode dummy = new ListNode(0);
        dummy.next=head;
        ListNode second = dummy;
        for (int i = 0; i < n; ++i) {
            first = first.next;
        }
        while (first != null) {
            first = first.next;
            second = second.next;
        }
        second.next = second.next.next;
        ListNode ans = dummy.next;
        return ans;
    }
}
```

2.5.3 删除重复元素的三道题

我们继续看第三组关于结点删除的题：

【5】LeetCode 83 存在一个按升序排列的链表，请你删除所有重复的元素，使每个元素只出现一次。

【6】LeetCode 82 存在一个按升序排列的链表，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中没有重复出现的数字。

【7】LeetCode 1836. 从未排序链表中删除重复元素。

LeetCode82和83 这两个题其实是一个，区别就是一个要将出现重复的保留一个，一个是只要重复都不要了，这种细微的差别我们处理起来并不是难事。这也再次用事实说明LeetCode不需要全部刷完，高质量刷三四百道就足够了。这里为了完整，我们还是都看一下。

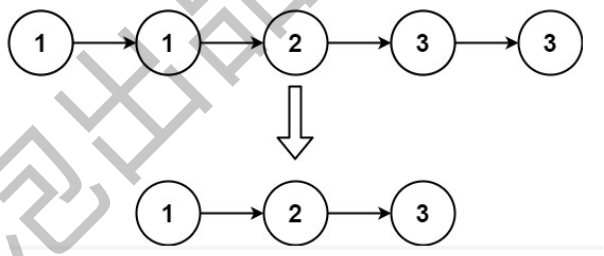
LeetCode 1836虽然也是在82的基础上改了一下条件，将链表改成无序的了，这个难度要增加不少，接下来也一起分析。

【1】重复元素保留一个

我们还是先看题目要求：

LeetCode83 存在一个按升序排列的链表，给你这个链表的头节点 head ，请你删除所有重复的元素，使每个元素只出现一次 。返回同样按升序排列的结果链表。

```
示例1：
输入：head = [1,1,2,3,3]
输出：[1,2,3]
```



由于给定的链表是排好序的，因此重复的元素在链表中出现的位置是连续的，因此我们只需要对链表进行一次遍历，就可以删除重复的元素。

具体地，我们从指针 cur 指向链表的头节点，随后开始对链表进行遍历。如果当前 cur 与cur.next 对应的元素相同，那么我们就将cur.next 从链表中移除；否则说明链表中已经不存在其它与cur 对应的元素相同的节点，因此可以将 cur 指向 cur.next。当遍历完整个链表之后，我们返回链表的头节点即可。

另外要注意的是 当我们遍历到链表的最后一个节点时，cur.next 为空节点，如果不加以判断，访问 cur.next 对应的元素会产生运行错误。因此我们只需要遍历到链表的最后一个节点，而不需要遍历完整个链表。

上代码：

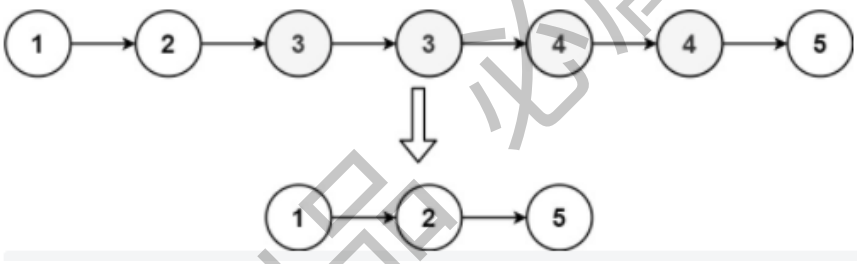
```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) {
            return head;
```

```
        }
        ListNode cur = head;
        while (cur.next != null) {
            if (cur.val == cur.next.val) {
                cur.next = cur.next.next;
            } else {
                cur = cur.next;
            }
        }
        return head;
    }
}
```

【2】重复元素都不要

LeetCode82：这个题目的要求与83的区别仅仅是重复的元素都不要了。例如：

```
示例1：
输入：head = [1,2,3,3,4,4,5]
输出：[1,2,5]
```



如果换成数组，解决起来比链表要麻烦，因为数组必须考虑元素指向和后序移动问题，而对于链表则直接对 cur.next 以及 cur.next.next 两个node进行比较就行了。但是这里要注意两个node可能为空节点，稍加判断就行了。

```
public ListNode deleteDuplicates(ListNode head) {
        if (head == null) {
            return head;
        }

        ListNode dummy = new ListNode(0, head);

        ListNode cur = dummy;
        while (cur.next != null && cur.next.next != null) {
            if (cur.next.val == cur.next.next.val) {
                int x = cur.next.val;
                while (cur.next != null && cur.next.val == x) {
                    cur.next = cur.next.next;
                }
            } else {
                cur = cur.next;
```

```
            }
        }
        return dummy.next;
    }
```

【3】从未排序链表中删除重复元素

如果将82中的排序链表改成无序的该怎么删呢?

```
示例1:
输入: head = [3,2,2,1,3,2,4]
输出: [1,4]
```

分析这个题目之前，我们先造一个题：无序链表中，重复元素只保留一个，该怎么做？可以借助一个hash，遍历的时候一边访问元素，一边写到hashMap中，如果发生碰撞，则只保留一个。这时候Hashmap里有的就是所有不存在重复元素的情况，接下来就好办了。

但是这个题要求重复的都不要，那一次遍历就不行了。我们需要扫描两遍+一个HashMap，HashMap 记录每个不同 node.val 的出现次数。第一遍扫描的时候记录每个不同 node.val 的出现次数。第二遍扫描的时候，创建一个虚拟 dummy 节点，用dummy.next去试探下一个节点是否是需要删除的节点，如果是，就直接跳过即可。

```
public ListNode deleteDuplicatesUnsorted(ListNode head) {
        if (head == null || head.next == null) {
           return head;
        }
        HashMap<Integer, Integer> map = new HashMap<>();
        ListNode cur = head;
        ListNode dummy = new ListNode(), pos = dummy; //创建一个虚拟头

        //将链表中的每个值都放入map中并计数;
        while (cur != null){
            map.put(cur.val, map.getOrDefault(cur.val, 0)+1);
            cur = cur.next;
        }

        cur = head; //指针回拨到头结点
        while (cur != null){
            if (map.get(cur.val) == 1){
                pos.next = cur;
                cur = cur.next;
                pos = pos.next;
            }else{
              //倒数第一个如果出现多次，则pos.next 应该直接设定为null
                if (cur.next == null){
                    pos.next = null;
                }
                cur = cur.next;
            }
        }
```

```
        return dummy.next;
    }
```

2.5.4一个特殊的结点删除问题

LeetCode 237：删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为要被删除的节点 。

```
示例1：
输入：head = [4,5,1,9], node = 5
输出：[4,1,9]
```

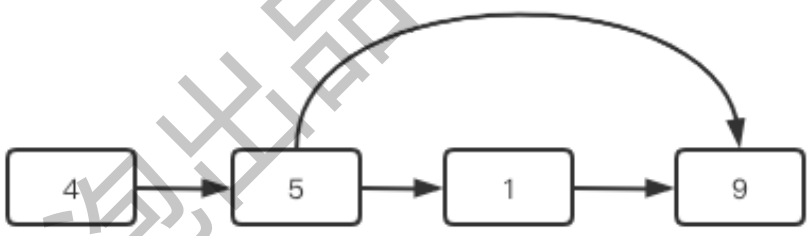
解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.

```
示例2：
输入：head = [4,5,1,9], node = 1
输出：[4,5,9]
解释：给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

这里的意思是给你的node节点不是链表的头节点，而是直接要删除的节点。例如上面给的node=5，这个node是直接要删除的，而不是首节点，那就不用使用前面的方式先遍历找前驱，再删除了。

那该怎么删呢， 其实也不难，我们可以采用数组移动的思想，将node后面的元素值逐个覆盖其前面的元素就行了。

例如下面这个图：



我们要删除的node=3，那我们就用node.next的值4来覆盖3。然后后面的5覆盖4就可以了。这就是数组删除元素的套路嘛（又是套路）

所以，代码就可以这么写：

```
class Solution {
    public void deleteNode(ListNode node) {
         node.val = node.next.val;
         node.next = node.next.next;
    }
}
```

你是不是觉得写错了，为啥只有两行？但是确实想清楚之后，两行就解决了。

上面这种移动元素值的情况遇到的很少，一般的算法题，如果我们想移动值，或者将其先保存到数组里，会被面试官毙掉，所以我们不到万不得已，不能用这种方法。

2.6 重点+热点：链表反转以及5道变形题

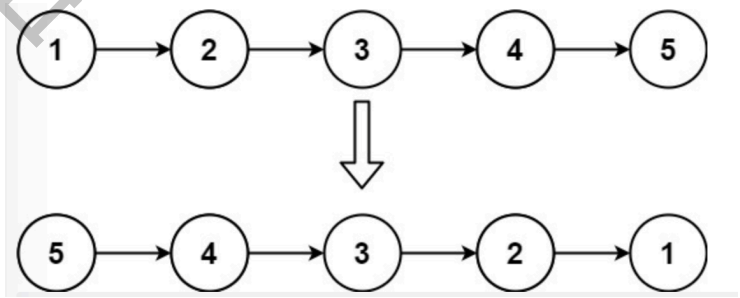
链表反转是一个出现频率特别高的算法题，笔者过去这些年面试，至少遇到过七八次。其中更夸张的是曾经两天写了三次，上午YY，下午金山云，第二天快手。链表反转在各大高频题排名网站也长期占领前三。比如牛客网上这个No 1 好像已经很久了。所以链表反转是我们学习链表最重要的问题，没有之一。

那为什么反转这么重要呢？因为反转链表涉及结点的增加、删除等多种操作，能非常有效考察对指针的驾驭能力和思维能力。

另外很多题目也都要用它来做基础， 例如指定区间反转、链表K个一组翻转。还有一些在内部的某个过程用到了反转，例如两个链表生成相加链表。还有一种是链表排序的，也是需要移动元素之间的指针，难度与此差不多。接下来我们就具体看一下每个题目。

2.6.1 反转一个链表

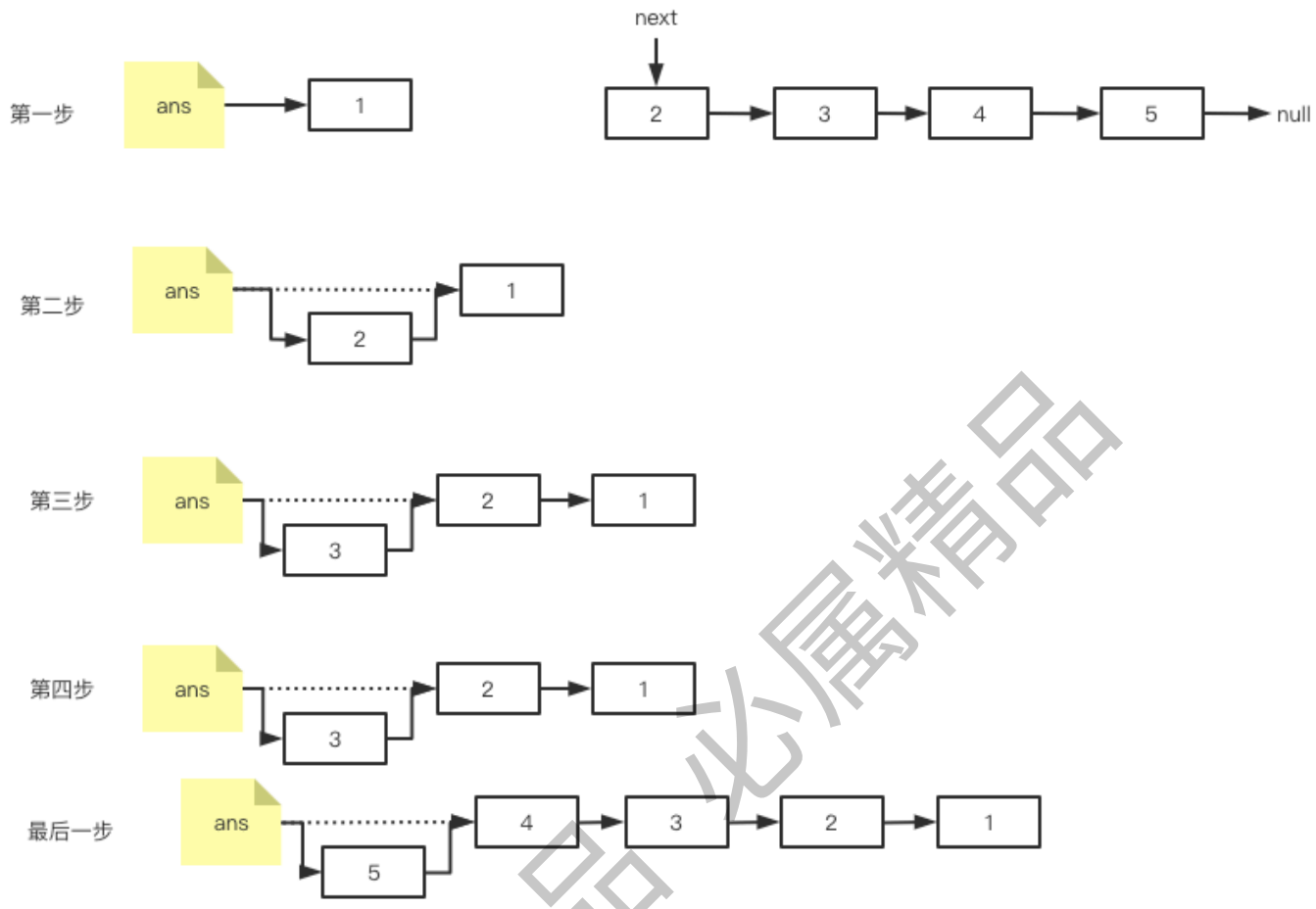
给你单链表的头节点 head，请你反转链表，并返回反转后的链表。
示例1：
输入：head = [1,2,3,4,5]
输出：[5,4,3,2,1]



这个题有两种方法，带头结点和不带头结点。我们都应该会，因为链表反转只有这两种方式，都很重要。

【1】建立虚拟头结点辅助反转

对于链表问题，如何处理头结点是个比较麻烦的问题。很多场景下可以先建立一个虚拟的结点ans，使得ans.next=head，这样可以很好的简化我们的操作。如下图所示。



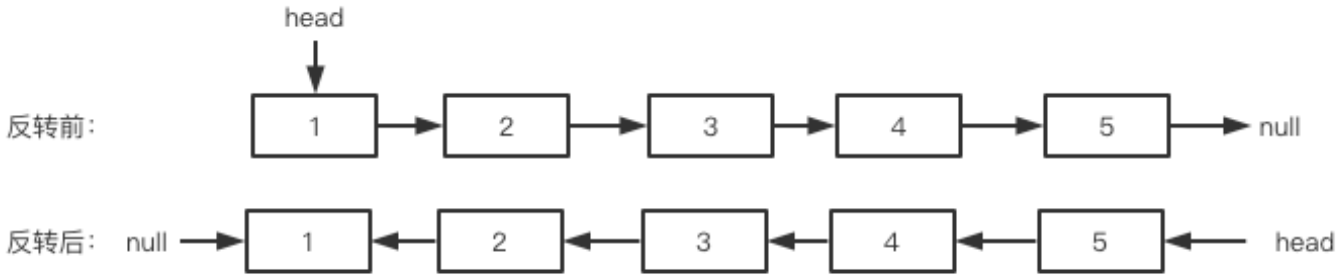
首先我们可以将1接到ans的后面之后，后面每个元素，例如2，3，4，5，我们都将其接到ans后面，这样已经组成链的1 2 3 4 将被逐渐甩到后面去了，所以当5成功插入到ans之后，整个链表的反转就完成了。这时候只要返回ans.next就得到反转的链表了。

当我们插入元素的时候，可以创建新的结点然后接到ans后面，也可以复用已有的结点，只是调整指针，相对来说，前面一种思维难度稍微低一些，但是往往会被面试官禁止，我们提倡使用后者。直接复用已有结点，只是调整指针的代码：

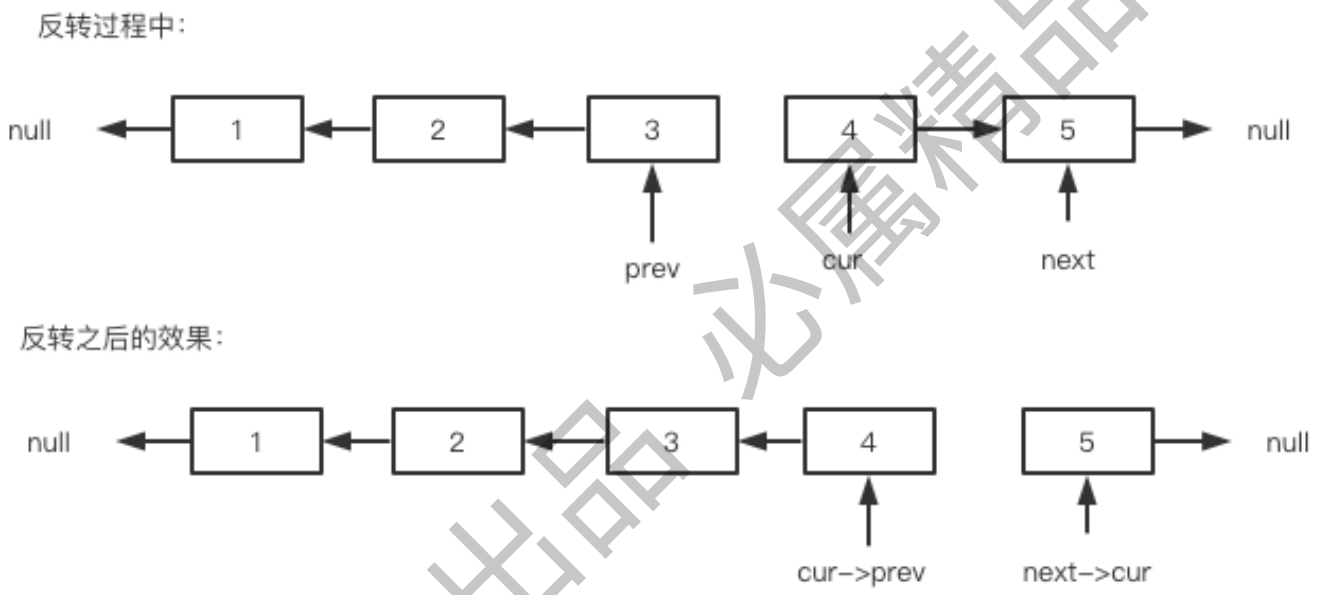
```
// 方法1：虚拟结点，,并复用已有的结点
  public static ListNode reverseList(ListNode head) {
      ListNode ans = new ListNode(-1);
      ListNode cur = head;
      while (cur != null) {
          ListNode next = cur.next;
          cur.next = ans.next;
          ans.next = cur;
          cur = next;
      }
      return ans.next;
  }
```

【2】直接操作链表实现反转

如果不使用虚拟结点，同样可以选择创建新结点或者只调整指针，但是如果再定义一个新的会浪费空间，所以我们只看如何将每个结点的指向都反过来的方法：



那这里的问题就是如何准确的记录并调整指针，我们看执行期间的过程示意图：



在上图中，我们用cur来表示旧链表被访问的位置，也就是本轮要调整的结点，pre表示已经调整好的新链表的表头，next是先一个要调整的。注意图中箭头方向，cur和pre都是两个表的表头，每移动完一个结点之后，我们必须准确知道两个链表的表头。

cur是需要接到pre的，那该怎么知道其下一个结点5呢?

代码也不算很复杂：

```
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;
    }
}
```

将上面这段代码在理解的基础上背下来，是的，因为这个算法太重要

【3】拓展 通过递归来实现

这个问题其实还有个递归方式反转，我们在讲解递归的时候会再来看该部分，这里只做了解。

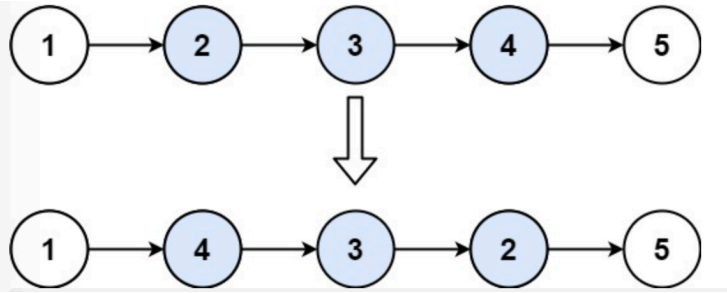
```
public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }
```

除了上面的基础方式，还有三个典型的拓展题也是面试经常见到的，这三个拓展题就是指定区间反转、K个一组反转和两两反转，每一个问题的实现思路都是两种，一种是头插法的变形，一种是不用虚拟结点的变形，我们一个个来看。

2.6.2 指定区间反转

```
LeetCode92 ：给你单链表的头指针 head 和两个整数 left 和 right ，其中 left <= right 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表 。
示例 1：
输入：head = [1,2,3,4,5], left = 2, right = 4
输出：[1,4,3,2,5]
```

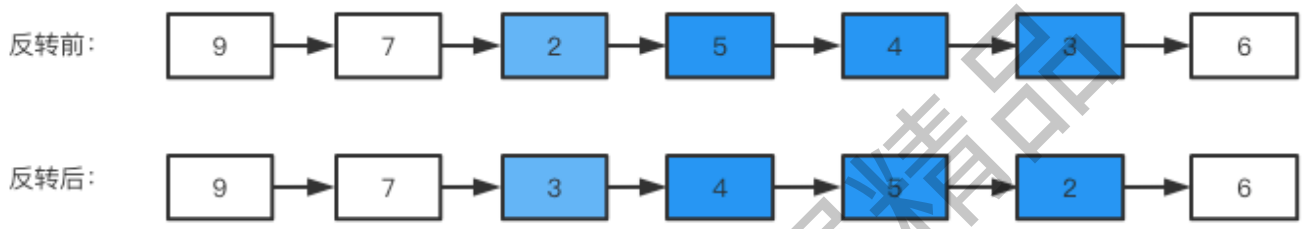
图示：



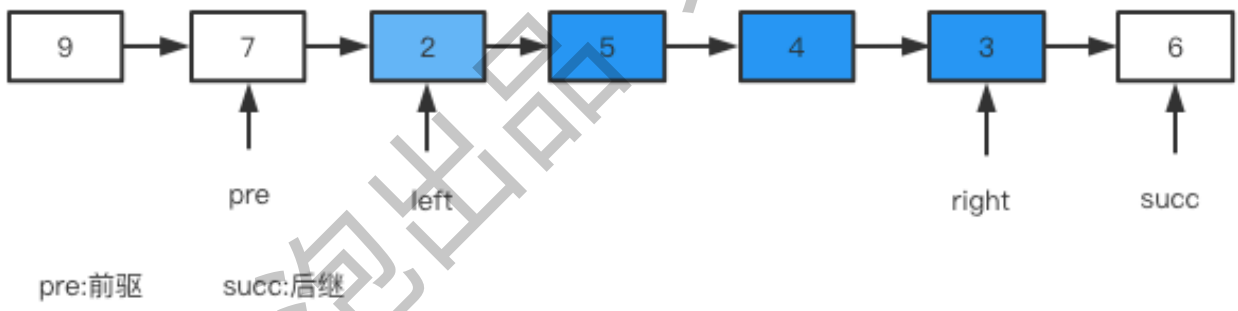
这里的处理方式也有多种，甚至给个名字都有点困难，干脆就分别叫穿针引线法和头插法吧。

【1】穿针引线法

我们以反转下图中蓝色区域的链表反转为例。

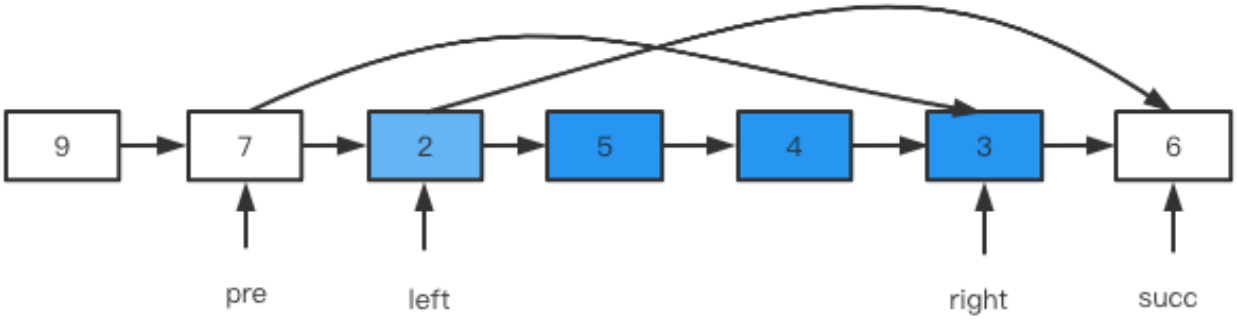


我们可以这么做：先确定好需要反转的部分，也就是下图的 `left` 到 `right` 之间，然后再将三段链表拼接起来。这种方式类似裁缝一样，找准位置减下来，再缝回去。这样问题就变成了如何标记下图四个位置，以及如何反转left到right之间的链表。



算法步骤：

- 第 1 步：先将待反转的区域反转；
- 第 2 步：把 pre 的 next 指针指向反转以后的链表头节点，把反转以后的链表的尾节点的 next 指针指向 succ。



pre:前驱　　succ:后继

编码细节我们直接看下方代码。思路想明白以后，编码不是一件很难的事情。这里要提醒大家的是，链接什么时候切断，什么时候补上去，先后顺序一定要想清楚，如果想不清楚，可以在纸上模拟，让思路清晰。

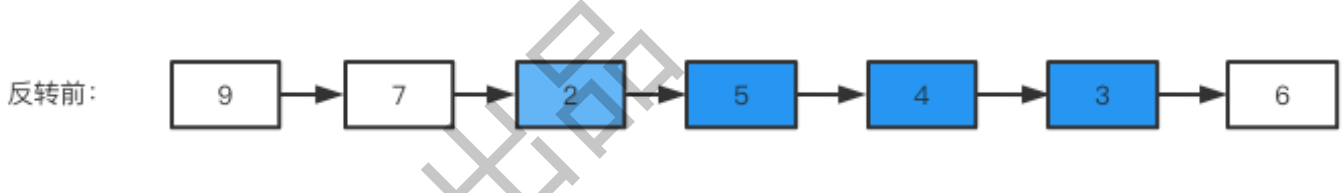
```
class Solution {
    public ListNode reverseBetween(ListNode head, int left, int right) {
        // 因为头节点有可能发生变化，使用虚拟头节点可以避免复杂的分类讨论
        ListNode dummyNode = new ListNode(-1);
        dummyNode.next = head;

        ListNode pre = dummyNode;
        // 第 1 步：从虚拟头节点走 left - 1 步，来到 left 节点的前一个节点
        // 建议写在 for 循环里，语义清晰
        for (int i = 0; i < left - 1; i++) {
            pre = pre.next;
        }

        // 第 2 步：从 pre 再走 right - left + 1 步，来到 right 节点
        ListNode rightNode = pre;
        for (int i = 0; i < right - left + 1; i++) {
            rightNode = rightNode.next;
        }

        // 第 3 步：切断出一个子链表（截取链表）
        ListNode leftNode = pre.next;
        ListNode curr = rightNode.next;

        // 思考一下，如果这里不设置next为null会怎么样
        pre.next = null;
        rightNode.next = null;

        // 第 4 步：同第 206 题，反转链表的子区间
        reverseLinkedList(leftNode);

        // 第 5 步：接回到原来的链表中
        //想一下，这里为什么可以用rightNode
```

```
            pre.next = rightNode;
            leftNode.next = curr;
            return dummyNode.next;
        }

        private void reverseLinkedList(ListNode head) {
            // 也可以使用递归反转一个链表
            ListNode pre = null;
            ListNode cur = head;
            while (cur != null) {
                ListNode next = cur.next;
                cur.next = pre;
                pre = cur;
                cur = next;
            }
        }
    }
```

【2】头插法（虚拟）

方法一的缺点是：如果 left 和 right 的区域很大，恰好是链表的头节点和尾节点时，找到 left 和 right 需要遍历一次，反转它们之间的链表还需要遍历一次，虽然总的时间复杂度为 O(N)，但遍历了链表 2次，可不可以只遍历一次呢？答案是可以的。我们依然画图进行说明。

我们依然以方法一的示例为例进行说明。



整体思想是：在需要反转的区间里，每遍历到一个节点，让这个新节点来到反转部分的起始位置。下面的图展示了整个流程。

第一步：将结点5插入到结点2的前面

第二步：将结点4插入到结点5的前面

第二步：将结点3插入到结点4的前面

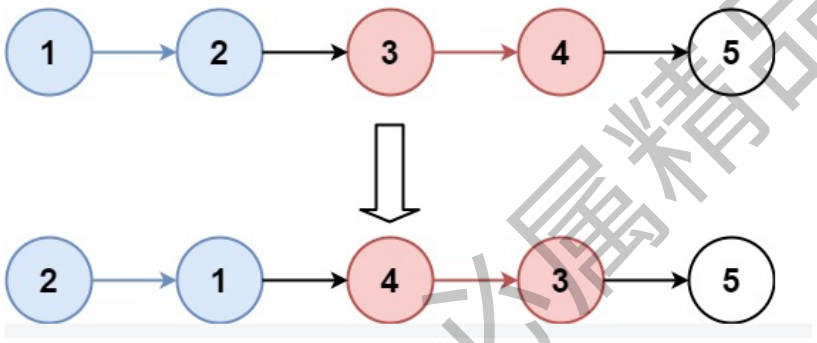
这个过程就是前面的带虚拟结点的插入操作，每走一步都要考虑各种指针怎么指，既要将结点摘下来接到对应的位置上，还要保证后续结点能够找到，请读者务必画图看一看，想一想，到底该怎么调整。代码如下：

```
class Solution {
    public ListNode reverseBetween(ListNode head, int left, int right) {
        // 设置 dummyNode 是这一类问题的一般做法
        ListNode dummyNode = new ListNode(-1);
        dummyNode.next = head;
        ListNode pre = dummyNode;
        for (int i = 0; i < left - 1; i++) {
            pre = pre.next;
        }
        ListNode cur = pre.next;
        ListNode next;
        for (int i = 0; i < right - left; i++) {
            next = cur.next;
            cur.next = next.next;
            next.next = pre.next;
            pre.next = next;
        }
        return dummyNode.next;
    }
}
```

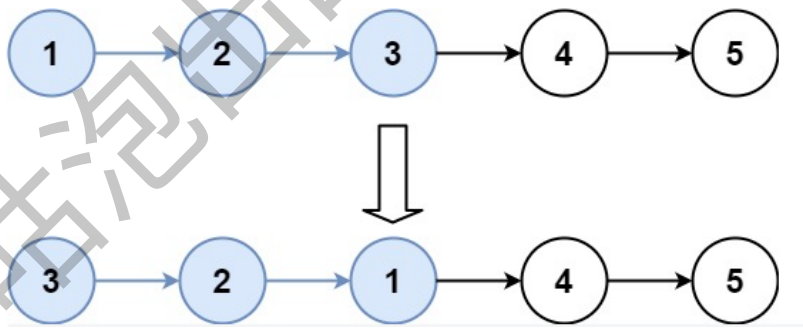
2.6.3 K个一组反转链表

K个一组反转可以说是链表中最难的一个问题了，我们看一下题意

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。k 是一个正整数，它的值小于或等于链表的长度。
如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。
进阶：
你可以设计一个只使用常数额外空间的算法来解决此问题吗?
你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。
示例1：
输入：head = [1,2,3,4,5], k = 2
输出：[2,1,4,3,5]



示例2：
输入：head = [1,2,3,4,5], k = 3
输出：[3,2,1,4,5]



分析

这个问题难点不在思路，而是具体实现，特别是每个段的首尾需要自动确定还要能接到移位。

参考网络上各路神仙的解决思路，基本思路就两种，要么是穿针引线法，要么是头插法。前者就是先将目标位置裁剪下来，然后再将两端补上，要么就是类似虚拟结点插入的头插法。 这两种画出来的图示都非常复杂，还不如读者自己一边想一边画。这里只给出简洁的文字描述和实现代码。

【1】穿针引线法

这种思路与上面的穿针引线类似，图示比较复杂，先看文字表述：

- 链表分区为已翻转部分+待翻转部分+未翻转部分
- 每次翻转前，要确定翻转链表的范围，这个必须通过 k 此循环来确定
- 需记录翻转链表前驱和后继，方便翻转完成后把已翻转部分和未翻转部分连接起来
- 初始需要两个变量 pre 和 end，pre 代表待翻转链表的前驱，end 代表待翻转链表的末尾
- 经过k此循环，end 到达末尾，记录待翻转链表的后继 next = end.next
- 翻转链表，然后将三部分链表连接起来，然后重置 pre 和 end 指针，然后进入下一次循环
- 特殊情况，当翻转部分长度不足 k 时，在定位 end 完成后，end==null，已经到达末尾，说明题目已完成，直接返回即可
- 时间复杂度为 O(n*K) 最好的情况为 O(n)最差的情况未 O(n^2)，空间复杂度为 O(1) 除了几个必须的节点指针外，我们并没有占用其他空间。

实现代码：

```
public ListNode reverseKGroup(ListNode head, int k) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;

    ListNode pre = dummy;
    ListNode end = dummy;

    while (end.next != null) {
        for (int i = 0; i < k && end != null; i++)
            end = end.next;

        if (end == null) break;
        ListNode start = pre.next;
        ListNode next = end.next;
        end.next = null;
        pre.next = reverse(start);
        start.next = next;
        pre = start;

        end = pre;
    }
    return dummy.next;
}

private ListNode reverse(ListNode head) {
    ListNode pre = null;
    ListNode curr = head;
    while (curr != null) {
        ListNode next = curr.next;
        curr.next = pre;
        pre = curr;
        curr = next;
```

```
    }
    return pre;
}
```

【2】头插法

与上一题的头插法，大致过程为：

- 1、找到待翻转的k个节点（注意：若剩余数量小于 k 的话，则不需要反转，因此直接返回待翻转部分的头结点即可）。
- 2、对其进行翻转。并返回翻转后的头结点（注意：翻转为左闭又开区间，所以本轮操作的尾结点其实就是下一轮操作的头结点）。
- 3、对下一轮 k 个节点也进行翻转操作。
- 4、将上一轮翻转后的尾结点指向下一轮翻转后的头节点，即将每一轮翻转的k的节点连接起来。

实现代码：

```
    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode tail = head;
        for (int i = 0; i < k; i++) {
            //剩余数量小于k的话，则不需要反转。
            if (tail == null) {
                return head;
            }
            tail = tail.next;
        }
        // 反转前 k 个元素
        ListNode newHead = reverse(head, tail);
        //下一轮的开始的地方就是tail
        head.next = reverseKGroup(tail, k);
        return newHead;
    }
    /*
    左闭又开区间
     */
    private ListNode reverse(ListNode head, ListNode tail) {
        ListNode pre = null;
        ListNode next = null;
        while (head != tail) {
            next = head.next;
            head.next = pre;
            pre = head;
            head = next;
        }
        return pre;

    }
```

2.6.4 两两交换链表中的节点

这个题的要求是给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

分析

这里为什么会有 这个加粗的部分的补充要求呢？其实我们进行调整有一种方法就是先将元素值保存到数组里，然后调整好之后再写回到链表，也就是只改链表的结点值，而不修改节点。这种方式降低了反转的难度，加粗部分就是要毙掉这种方式😄。

看到这个题是否感觉又是换了条件瞎搞的题？将上面的K换成2不就是这个题吗？如果将K设置尾3，4，5，那式是不是又可以造题了？

道理确实如此，但是如果K为2的时候，可不需要像K个一组一样需要一个遍历过程，直接取前后两个就行了，因此基于相邻结点的特性重新设计和实现就行，不需要上面这么复杂的操作，所以我们单独看一下。

我们创建虚拟结点 dummyHead，令 dummyHead.next = head。令 temp 表示当前到达的节点，初始时 temp = dummyHead。每次需要交换 temp 后面的两个节点。

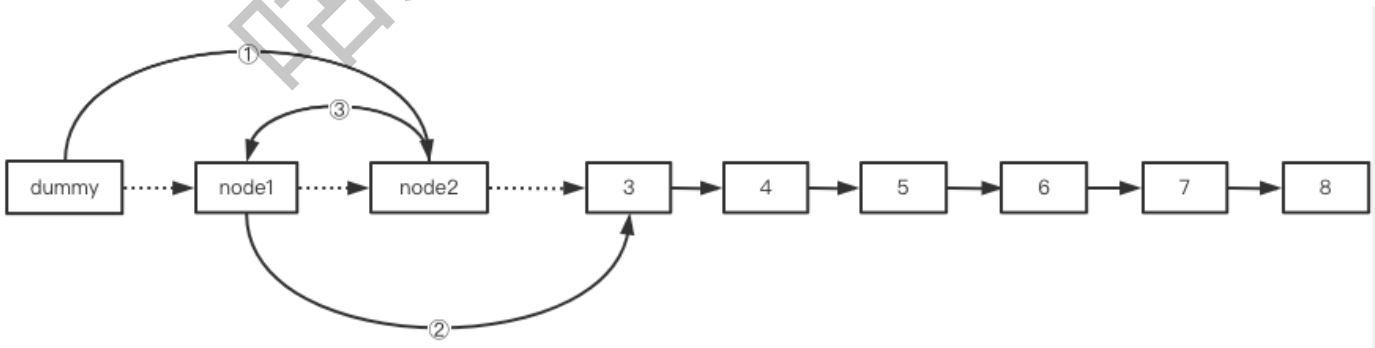
如果 temp 的后面没有节点或者只有一个节点，则没有更多的节点需要交换，因此结束交换。否则，获得 temp 后面的两个节点 node1 和 node2，通过更新节点的指针关系实现两两交换节点。

具体而言，交换之前的节点关系是 temp -> node1 -> node2，交换之后的节点关系要变成 temp -> node2 -> node1，因此需要进行如下操作。

```
temp.next = node2
node1.next = node2.next
node2.next = node1
```

完成上述操作之后，节点关系即变成 temp -> node2 -> node1。再令 temp = node1，对链表中的其余节点进行两两交换，直到全部节点都被两两交换。

两两交换链表中的节点之后，新的链表的头节点是 dummyHead.next，返回新的链表的头节点即可。



完整的代码是：

```
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummyHead = new ListNode(0);
        dummyHead.next = head;
```

```
        ListNode temp = dummyHead;
        while (temp.next != null && temp.next.next != null) {
            ListNode node1 = temp.next;
            ListNode node2 = temp.next.next;
            temp.next = node2;
            node1.next = node2.next;
            node2.next = node1;
            temp = node1;
        }
        return dummyHead.next;
    }
}
```

2.7 链表反转的应用

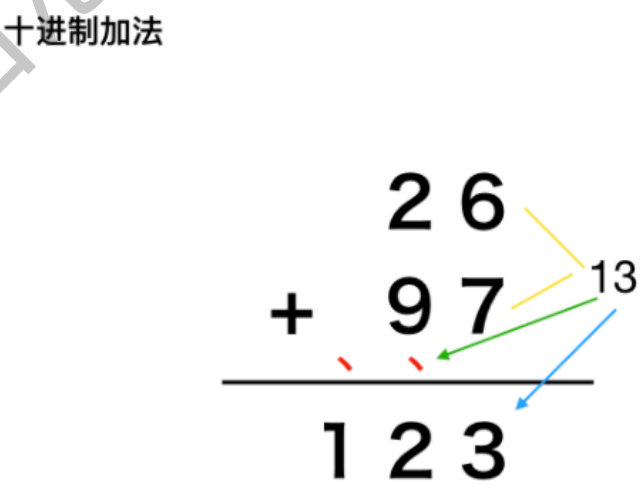
链表的反转我们研究了很多种情况，这几种都非常重要，但是这还不足征服链表反转，我们再看两个应用链表反转的例子。

2.7.1 单链表加1

用一个非空单链表来表示一个非负整数，然后将这个整数加一。
你可以假设这个整数除了 0 本身，没有任何前导的 0。
这个整数的各个数位按照 高位在链表头部、低位在链表尾部 的顺序排列。

示例：
输入：[1,2,3]
输出：[1,2,4]

在数组部分我们处理过几种加法和进制的算法，这里换成的链表又要搞一遍，你说算法是不是就是换换条件继续折腾。我们看一下加法的过程：



计算是从低位开始的，而链表是从高位开始的，所以要处理就必须反转过来，此时可以使用栈，也可以使用链表反转来实现。

【1】基于栈实现

基于栈实现的思路不算复杂，先把题目给出的链表遍历放到栈中，然后从栈中弹出栈顶数字 digit，加的时候再考虑一下进位的情况就ok了，加完之后根据是否大于0决定视为下一次要进位 。

```
class Solution {
    public ListNode plusOne(ListNode head) {
        Stack<Integer> st = new Stack();
        while (head != null) {
            st.push(head.val);
            head = head.next;
        }
        int carry = 0;
        ListNode dummy = new ListNode(0);
        int adder = 1;
        while (!st.empty() || adder != 0 || carry > 0) {
            int digit = st.empty() ? 0 : st.pop();
            int sum = digit + adder + carry;
            carry = sum >= 10 ? 1 : 0;
            sum = sum >= 10 ? sum - 10 : sum;
            ListNode cur = new ListNode(sum);
            cur.next = dummy.next;
            dummy.next = cur;
            adder = 0;
        }
        return dummy.next;
    }
}
```

【2】基于链表反转实现

如果这里不使用栈，要求你使用链表反转来实现，该怎么做呢？很显然，我们先将原始链表反转，这方面完成加1和进位等处理，完成之后再次反转。

这里请读者根据前面介绍的链表反转过程自行实现。

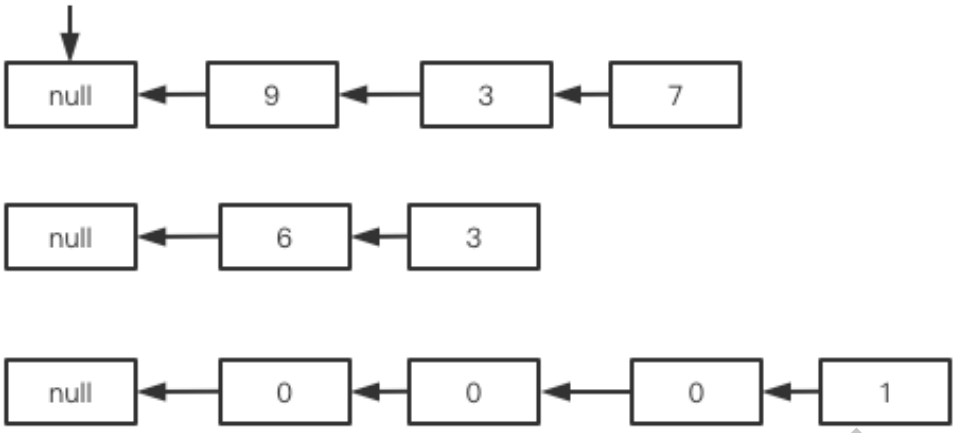
2.7.2 链表加法

相加相链表是基于链表构造的一种特殊题，反转只是其中的一部分。这个题还存在进位等的问题，因此看似简单，但是手写成功并不容易，这个题目在LeetCode中我没找到原题，但是在很多材料里有，而且笔者也确实曾经遇到过，所以我们就来研究一下。

题目要求是这样的：

```
假设链表中每一个节点的值都在 0 - 9 之间，那么链表整体就可以代表一个整数。
给定两个这种链表，请生成代表两个整数相加值的结果链表。这些数位是正向存放的，正向的意思如下所示
示例：
输入：(6 -> 1 -> 7) + (2 -> 9 -> 5)，即617 + 295
输出：9 -> 1 -> 2，即912
```

这个题目的难点在于存放是从最高位向最低位开始的，但是因为低位会产生进位的问题，计算的时候必须从最低位开始。所以我们必须想办法将链表节点的元素反转过来，如下图所示：



怎么反转呢？首先想到可以先用栈来将两个链表分别反转，然后再计算。当你将思路说出来之后，面试官通过会说：“栈要开辟O(n)的空间，你有只需要O(1)空间的方法吗？”。其实言外之意是栈太简单了，那这时候只能用链表反转来做了。

不过呢，为了充分思考，两种方式我们都看一下。

【1】使用栈实现

思路是先将两个链表的元素分别压栈，然后再一起出栈，将两个结果分别计算。之后对计算结果取模，模数保存到新的链表中，进位保存到下一轮。

完成之后再进行一次反转就行了。

我们知道在链表插入有头插法和尾插法两种。头插法就是每次都将新的结点插入到head之前。而尾插法就是将新结点都插入到链表的表尾。两者的区别是尾插法的顺序与原始链表是一致的，而头插法与原始链表是逆序的，所以上面最后一步如果不想进行反转，可以将新结点以头插法。

```
import java.util.*;
/*
 * public class ListNode {
 *   int val;
 *   ListNode next = null;
 * }
 */
public class Solution {
    public ListNode addInList (ListNode head1, ListNode head2) {
          Stack<ListNode> st1 = new Stack<ListNode>();
          Stack<ListNode> st2 = new Stack<ListNode>();
        while(head1!=null){
            st1.push(head1);
            head1=head1.next;
        }
        while(head2!=null){
            st2.push(head2);
            head2=head2.next;
        }
```

```
        ListNode newHead=new ListNode(-1);
        int carry=0;
 //这里设置carry!=0,是因为当st1,st2都遍历完时，如果carry=0,就不需要进入循环了
while(!st1.empty()||!st2.empty()||carry!=0)
{
    ListNode a=new ListNode(0);
     ListNode b=new ListNode(0);
    if(!st1.empty())
    {
        a=st1.pop();
    }
    if(!st2.empty())
    {
        b=st2.pop();
    }
    //每次的和应该是对应位相加再加上进位
    int get_sum=a.val+b.val+carry;
    //对累加的结果取余
    int ans=get_sum%10;
    //如果大于0，就进位
    carry=get_sum/10;
    ListNode cur=new ListNode(ans);
    cur.next=newHead.next;
    //每次把最新得到的节点更新到neHead.next中
    newHead.next=cur;
}
   return newHead.next;
   }
}
```

【2】使用链表反转实现

如果不用栈，那只能用链表反转了，先将两个链表分别反转，最后计算完之后再将结果反转，一共需要三次。进位等的处理与上面差不多。

```
public class Solution {
    public ListNode addInList (ListNode head1, ListNode head2) {
        head1 = reverse(head1);
        head2 = reverse(head2);
        ListNode head = new ListNode(-1);
        ListNode cur = head;
        int carry = 0;
        while(head1 != null || head2 != null) {
            int val = carry;
            if (head1 != null) {
                val += head1.val;
                head1 = head1.next;
            }
            if (head2 != null) {
```

```
                val += head2.val;
                head2 = head2.next;
            }
            cur.next = new ListNode(val % 10);
            carry = val / 10;
            cur = cur.next;
        }
        if (carry > 0) {
            cur.next = new ListNode(carry);
        }
        return reverse(head.next);
    }

    private ListNode reverse(ListNode head) {
        ListNode cur = head;
        ListNode pre = null;
        while(cur != null) {
            ListNode temp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = temp;
        }
        return pre;
    }
}
```

上面我们直接调用了反转函数，这样代码写起来就容易很多，如果你没手写过反转，所有功能都是在一个方法里，那复杂度要高好几个数量级，甚至自己都搞不清楚了。

既然加法可以，那如果是减法呢？读者可以自己想想该怎么处理。

3.大厂冲刺题

4.总结

通过上面的这些题目，我们能感受到链表的题目真是不少，但是大部分常规题目，都是从增删改查变换或者组合而来的。这些题目大部分一看就知道该怎么做，但是要写出来甚至运行成功，难度还是很大的，所以，我们需要耐住寂寞，认真练习，只有练会了才可能在考场上应对自如，这就是所谓的思维能力了。

在上面这些题目中，需要特别强调的就是反转相关的几个问题必须都要会，因为这几个问题的考察频率非常高，而且对链表的能力要求也不低，必须好好掌握。