简介

我们知道数组可以作为高级算法的载体，所以相关题目特别多，但是链表却很少被作为载体，因此题目数量少很多，类型也相对固定，因此只要将常见题目学完就可以了。

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，没有其他信息，如果遍历链表最终会在访问尾结点之后获得null。

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，比较多的是带头结点的单链表和双向循环链表。 有时候会将多个链表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。

在面试算法中，大多以考察普通单链表为主，我们本章大部分题目都是针对单链表的。其他类型的题目多以设计题为主，难度不算很大，但是要规范、严谨。

【学习目标】

1.理解java构造链表的原理，能够自己构造链表。

2.掌握常见的链表算法问题，并且能从多个角度理解如何解决问题。

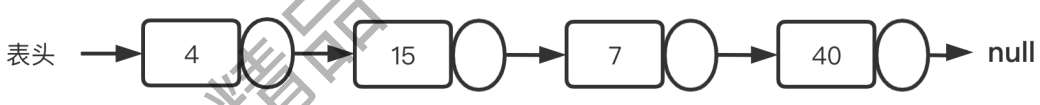
3.如何判断链表中是否有环 ，以及如何找到环的位置。

4.掌握链表反转，以及拓展问题。

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。

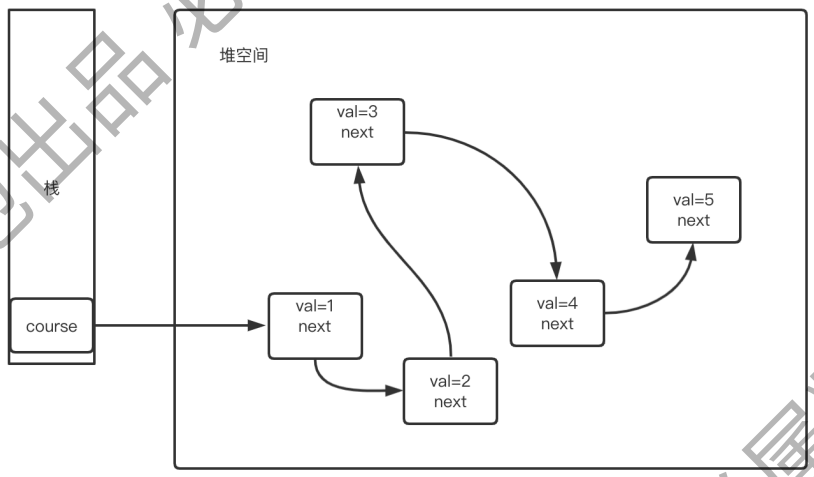
首先理解JVM是怎么构建出链表的，我们知道JVM里有栈区和堆区，栈区主要存引用，也就是一个指向实际对象的地址，而堆区存的才是创建的对象，例如我们定义这样一个类：

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

这里的teacher和student就是指向堆的应用，假如我们这样定义：

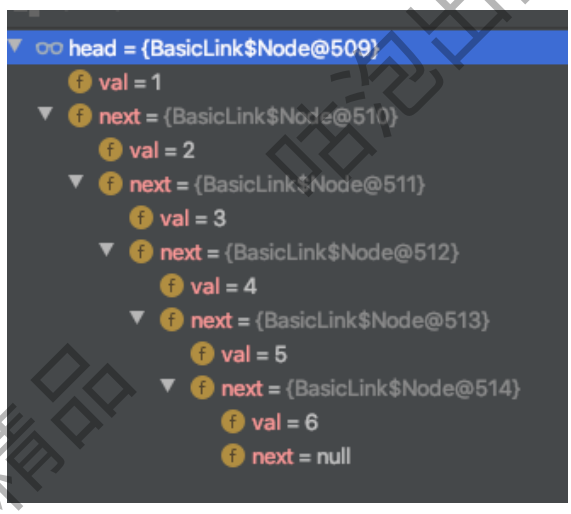
```
public class Course{
 int val;
 Course next;
}
```

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。

在配套代码中BasicLink类，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。

在LeetCode中算法题中经常使用这样的方式来创建链表：

```
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，但是上面的代码更为精简，因此在算法题目中应用广泛。

```
#规范的链表定义
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;
    }
}
```

另外，有时候只看ListNode不知道表示的是链表还是结点，因此在很多工程里会在链表类里定义一个静态内部类专门表示结点，而外面的class表示链表，也就是这样子：

```
public class LinkedListBasicUse {
   .......链表相关内容....
   //内部类
 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：

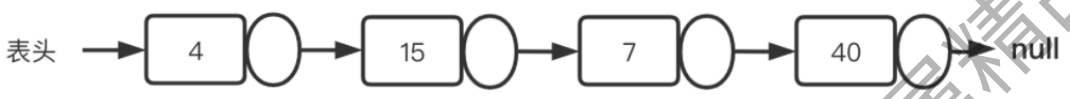
```
   LinkedListBasicUse.Node head = new Node(1);
```

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。




```
    public static int getListLength(Node head) {
        int length = 0;
       Node node = head;
        while (node != null) {
            length++;
            node = node.next;
        }
        return length;
    }
```

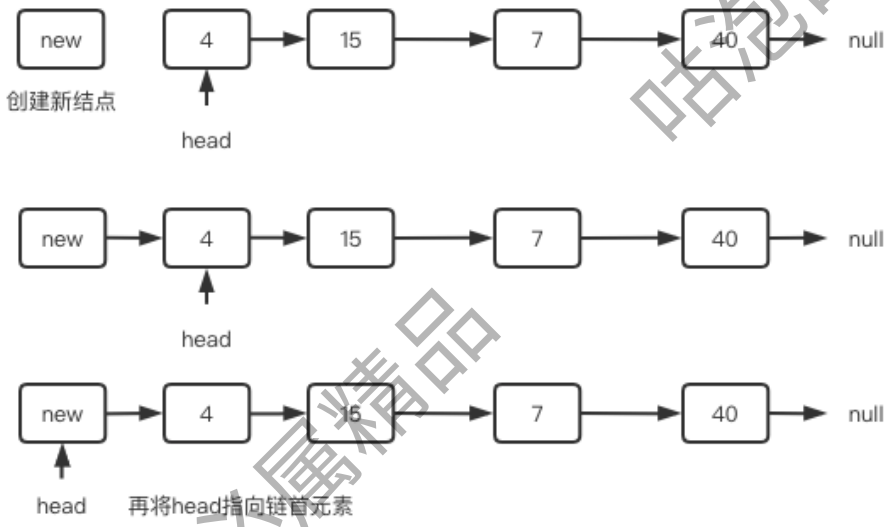
1.3 链表插入

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需要重新指向表头。 我们创建一个新结点newNode，怎么连接到原来的链表上呢？执行newNode.next=head即可。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，所以让head=newNode就行了，如下图：



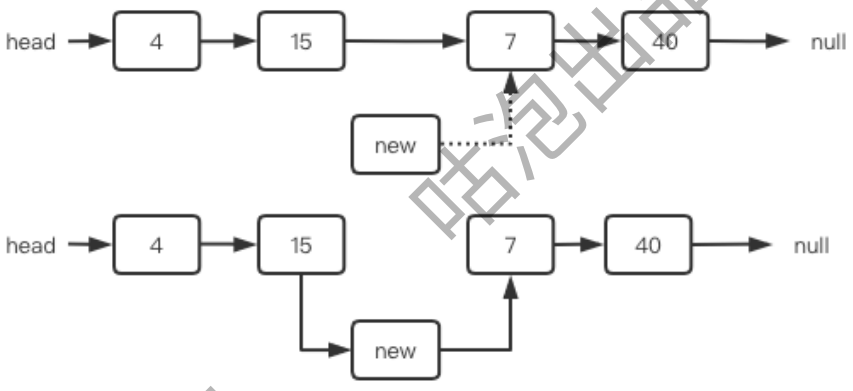
（2）在链表中间插入

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，也就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。

为此，我们要在目标结点的前一个位置停下来，也就是使用cur.next的值而不是cur的值来判断，这是链表最常用的策略。

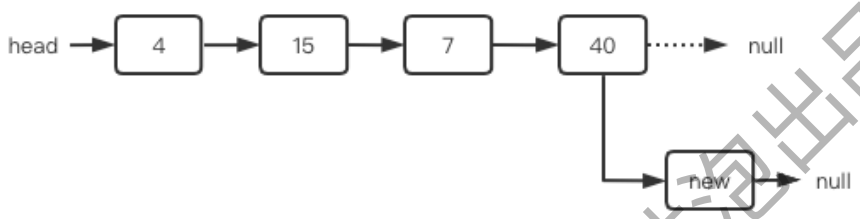
例如下图中，如果要在7的前面插入，当cur.next=node(7)了就应该停下来，此时cur.val=15。然后需要给newNode前后接两根线，此时只能先让new.next=node(15).next(图中虚线)，然后node(15).next=new，而且顺序还不能错。**想一下为什么不能颠倒顺序?**

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，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：

```
/**
     * 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;
    }
```

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，也可以直接抛出不能插入的异常，两种处理都可以，一般来说我们更倾向前者。

如果链表是单调递增的，一般会让你将元素插入到合适的位置，序列仍然保持单调，你可以尝试写一下该如何实现。

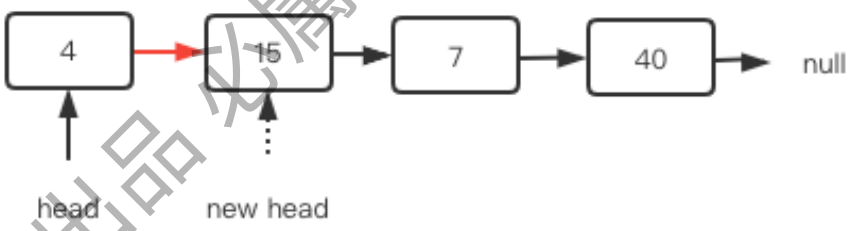
1.4 链表删除

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。

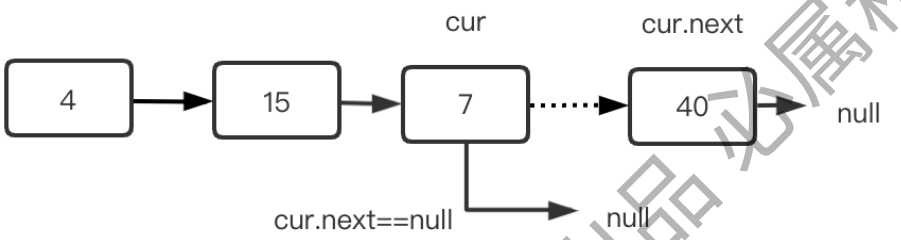
(1)删除表头结点

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。



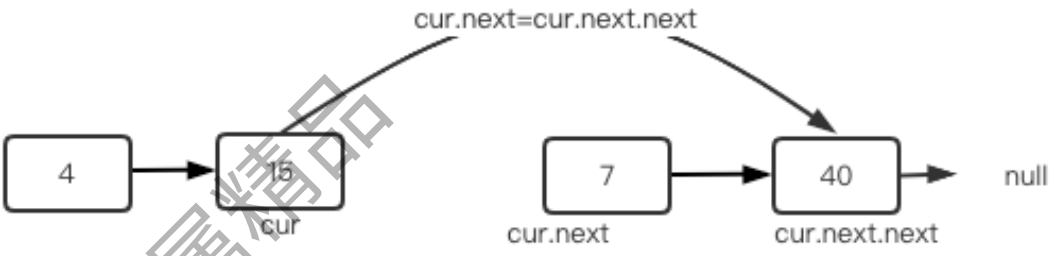
(2)删除最后一个结点

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

删除中间结点时，也会要用cur.next来比较，找到位置后，将cur.next指针的值更新为cur.next.next就可以解决，如下图所示：



完整实现：

```
/**
     * 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;
    }
```

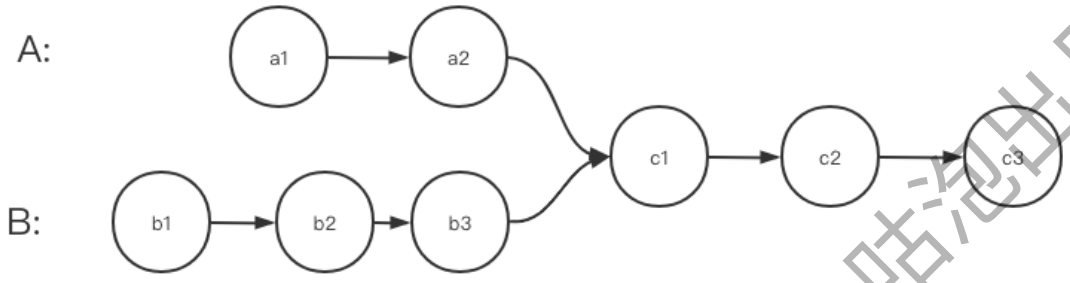
同样，在很多算法中链表是单调的，让你在删除元素之后仍然保持单调，建议写一下试试。

2.高频面试题

链表的算法题比数组少很多，而在回溯贪心动规等高级算法中很少见到链表的影子。我们这里就集中研究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，剑指offer52 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?

单链表中每个节点只能指向唯一的下一个next，但是可以有多个指针指向一个节点。例如上面c1就可以被a2，b3同时指向。该怎么入手呢？如果一时想不到该怎么办呢?

告诉你一个屡试不爽的方法：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。

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。虽然简单，但是时间复杂度高，排除！

再看Hash，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，那就找到了。OK，第二种方法出来了。既然Hash可以，那集合呢？和Hash一样用，也能解决，OK，第三种方法出来了。

队列和栈呢？这里用队列没啥用，但用栈呢？现将两个链表分别压到两个栈里，之后一边同时出栈，一边比较出栈元素是否一致，如果一致则说明存在相交，然后继续找，最晚出栈的那组一致的节点就是要找的位置，于是就有了第四种方法。

这时候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。面试官很明显就会问你，怎么解决?

然后你可以继续说HashMap、集合和栈具体应该怎么解决。

假如你想错了，比如你开始说队列能，但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”就可以了，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。

除此上面的方法，还有两种比较巧妙的方法，我们一个个看：

2.1.1 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 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;
    }
```

2.1.2 集合Set法

这里也可以用集合，思路和上面的一样，直接看代码：

```
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;
}
```

2.1.3 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 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！！

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，我们继续看：

2.1.4 拼接两个字符串

先看下面的链表A和B：

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

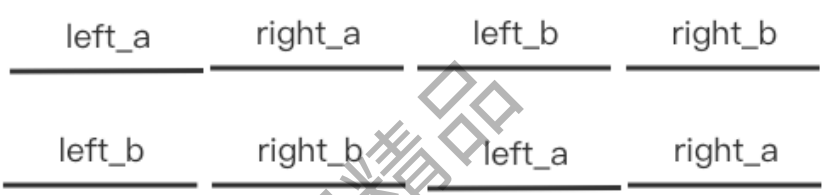
B: a-b-4-5

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

我们发现拼接后从最后的4开始，两个链表是一样的了，自然4就是要找的节点，所以可以通过拼接的方式来寻找交点。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right_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，姑且算作双指针法吧。

2.1.5 差和双指针

我们再看另一个使用差和双指针来解决问题的方法。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;
        //分别统计两个链表的长度
        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;
        //长的先走sub步
       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++;
        }
       }
        //同时遍历两个链表
       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 判断链表是否为回文序列

LeetCode234，这也是一道简单，但是很经典的链表题，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，我们仍然是先将常见的数据结构和算法思想想一遍，看看谁能解决问题。

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。这种方法会被视为逃避链表，面试不能这么干。

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，一边比较两者元素值，只要有一个不相等，那就不是。

方法3：优化方法2，先遍历第一遍，得到总长度。之后一边遍历链表，一边压栈。到达链表长度一半后就不再压栈，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。这样可以节省一半的空间。

方法4：优化方法3：既然要得到长度，那还是要遍历一次链表才可以，那是不是可以一边遍历一边全部压栈，然后第二遍比较的时候，只比较一半的元素呢？也就是只有一半的元素出栈，链表也只遍历一半，当然可以。

方法5：反转链表法， 先创建一个链表newList，将原始链表oldList的元素值逆序保存到newList中，然后重新一边遍历两个链表，一遍比较元素的值，只要有一个位置的元素值不一样，就不是回文链表。

方法6：优化方法5，我们只反转一半的元素就行了。先遍历一遍，得到总长度。然后重新遍历，到达一半的位置后不再反转，就开始比较两个链表。

方法7：优化方法6，我们使用双指针思想里的快慢指针，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8：在遍历的时候使用递归来反转一半链表可以吗？当然可以，再组合一下我们还能想出更多的方法，解决问题的思路不止这些了，此时单纯增加解法数量没啥意义了。

上面这些解法中，各有缺点，实现难度也不一样，有的甚至算不上一个独立的方法，这么想只是为了开拓思路、举一反三。我们选择最佳的两种实现，其他方法请同学自行写一下试试。

2.2.1 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;
    }
    //之后一边出栈，一遍比较
    while (head != null) {
        if (head.val != stack.pop()) {
            return false;
        }
        head = head.next;
    }
    return true;
}
```

2.2.2 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两行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，如果不理解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;
    }
```

2.3 合并有序链表

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。两者有相似的地方，也有不同的地方，你能找到分别是什么吗?

2.3.1 合并两个有序链表

LeetCode21 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。

本题虽然不复杂，但是很多题目的基础，解决思路与数组一样，一般有两种。一种是新建一个链表，然后分别遍历两个链表，每次都选最小的结点接到新链表上，最后排完。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。先看下面这种：

```
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，我们可以将其变得苗条一些：第一个while只处理两个list 都不为空的情况，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 进一步优化代码

进一步分析，我们发现两个继续优化的点，一个是上面第一个大while里有三种情况，我们可以将其合并成两个，如果两个链表存在相同元素，第一次出现时使用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中的代码块。这样就可以将第一个while的代码从三种变成两种，精简了很多。

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链表头接好，后面的自然就接上了，因此循环都不用写，也就是这样：

```
    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，建议先用上面第二种写法写出来，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”，然后再写尝试写第三种，这样不但不用怕翻车，还可以锦上添花。

2.3.2 合并K个链表

合并k个链表，有多种方式，例如堆、归并等等。如果面试遇到，我倾向先将前两个合并，之后再将后面的逐步合并进来，这样的的好处是只要将两个合并的写清楚，合并K个就容易很多，现场写最稳妥：

```
    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：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 中下标从a到b的节点删除，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）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。

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样，花样随便变，以不变应万变就是我们的宗旨。

具体到这个题，按部就班遍历找到链表1保留部分的尾节点和链表2的尾节点，将两链表连接起来就行了。

```
    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;
    }
```

这里需要留意题目中是否有开闭区间的情况，例如如果是从a到b，那就是闭区间[a,b]。还有的会说一个开区间(a,b)，此时是不包括a和b两个元素，只需要处理a和b之间的元素就可以了。比较特殊的是进行分段处理的时候，例如K个一组处理，此时会用到左闭右开区间，也就是这样子[a,b)，此时需要处理a，但是不用处理b，b是在下一个区间处理的。此类题目要非常小心左右边界的问题。

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，但是在面试中出现的频率很高，我们集中看一下。

2.4.1 寻找中间结点

LeetCode876 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，**想一想为什么会这样**。

```
    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。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.
```

这里也可以使用快慢双指针，我们先将fast 向后遍历到第 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 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位置的时候必须判断fast是否为null，这是本题的关键问题之一，最终代码如下：

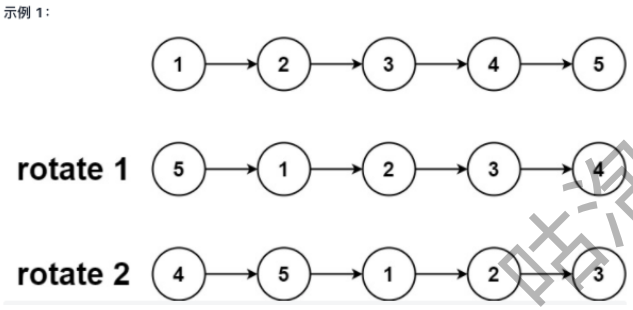
```
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode fast = head;
        ListNode slow = head;

        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

Leetcode61.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：
输入：head = [1,2,3,4,5], k = 2
输出：[4,5,1,2,3]
```

示例 1:



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。**这个题是否在数组里见过类似情况?**

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。具体思路是：

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，如果然后k=k % len，如果k == 0，则不用旋转，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。

```
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;
    }
```

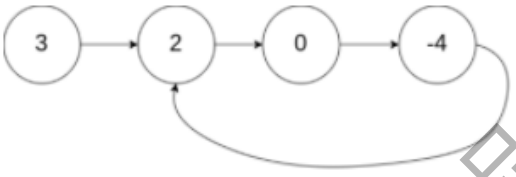
如果使用链表反转怎么做呢？学习完第三章《链表反转以及相关问题》之后，再来看，本题是一道作业题。

2.4.4 链表的环问题

本题同样是链表的经典问题。给定一个链表，判断链表中是否有环，这就是LeetCode141。进一步，假如有环，那环的位置在哪里？这就是LeetCode 142题。

这个问题前一问相对容易一些，后面一问比较难想到。但是，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，所以我们都要会。

```
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;
    }
```

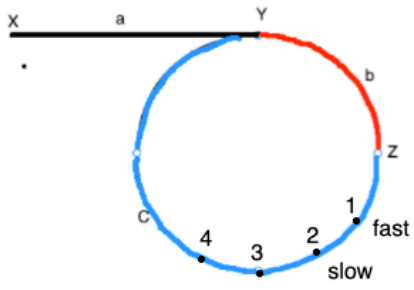
如果只用 O(1)的空间该怎么做呢？在网络上有很多人在解释的“正统”方法，但是都解释得不太清楚，我们这里介绍一种简单易懂的方法“三次使用双指针”，然后再介绍正统的方法。

2.4.4.1 为什么快慢两个指针一定会相遇

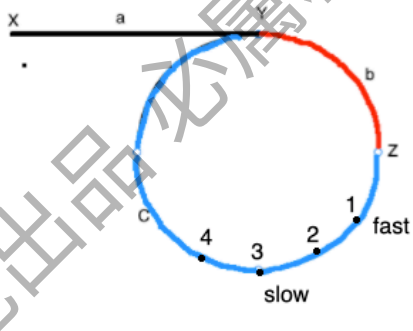
确定是否有环，最有效的方法就是双指针，一个快指针（一次走两步），一个慢指针（一次走一步）。如果快的能到达表尾就不会有环，否则如果存在环，则慢指针一定会在某个位置与快指针相遇。这就像在操场长跑，一个人快一个人慢，只要时间够，快的一定能在某个时候再次追上慢的人(也就是所谓的套圈)。

这里很多人可能会有疑问，因为两者每次走的距离不一样，会不会快的人在追上慢的时候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。

使用双指针思想寻找是否存在环的方法：

```
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;
    }
```

2.4.4.2 三次双指针法确定入口位置

如果还要确定环的入口，例如上面的图，指针从-4指向了2，要输出node(2)，该怎么做呢?

本节和下一节各介绍一种方法，本节的方法好想，但是代码不好写。下一节的问题代码好写但是不好想。

三次双指针的思想是：如果我们确定了环的大小和末尾结点，那该问题就退化成了找倒数第K个结点。我们来分析一下：

问题1：怎么判断环的大小呢？首先我们应该先判断是否存在环，此时可以使用上面说的快慢指针，我们假设fast和slow最后在P点。那接下来只要将一个指针例如slow固定在P位置，另一个fast从P开始遍历，显然，当fast=slow的时候自然就得到环的长度了。

问题2：那如何确定末尾结点呢？在上图中，我们注意到如果入口是node(2)，那我们遍历的时候如果指针p.next=node(2)就说明p就是链表的终点。

到此，这就是三次双指针方法：

第一次使用快慢指针判断是否存在环， fast一次走两步，slow一次走一步来遍历，如果最终相遇说明链表是否存在环。

第二次使用双指针判断环的大小，一个固定在相遇位置不动，另一个从相遇位置开始遍历，当两者再次相等的时候就找到了环的大小，假如为K。

第三次使用找倒数第K个结点的方法来找入口，根据上面2.4.2介绍的方法找倒数第K个元素的方法来找环的入口位置。

理解到这里，你能够将代码写出来？这也是是本章的一道作业题，请你实现一下，详细要求见作业说明https://gitee.com/zh-alg/zongheng_algorithm/tree/master/src/main/java/homework。

2.4.4.3 第二种确定入口的方法

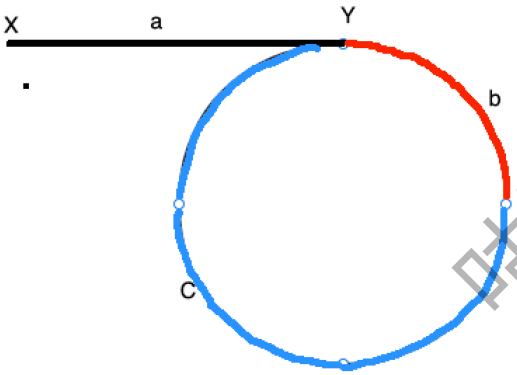
这里的问题是如果知道了一定有入口，那么如何确定入口的位置呢？这里介绍第二种方法，代码比上面小节的少，但是要理解清楚有些难度。

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先看假如一圈就遇到的情况

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 如果多圈之后才相遇

如果是走了多圈之后才遇到会怎么样呢？ 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 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

如果按照LeetCode顺序一道道刷题，会感觉毫无章法而且很慢，但是将相似类型放在一起，瞬间就发现不过就是在改改条件不断造题。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?

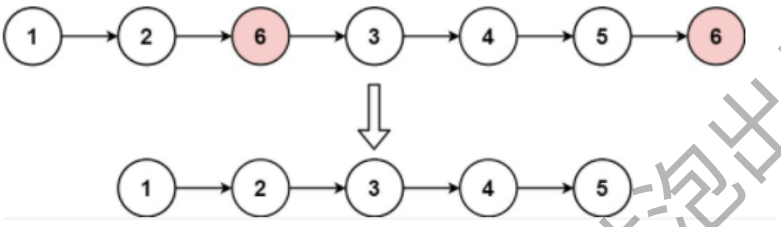
- LeetCode 237：删除某个链表中给定的（非末尾）节点。传入函数的唯一参数为要被删除的节点 。
- LeetCode 203：给你一个链表的头节点 head 和一个整数 val ，请你删除链表中所有满足 Node.val == val 的节点，并返回新的头节点 。
- LeetCode 19. 删除链表的倒数第 N 个节点。
- LeetCode 1474. 删除链表 M 个节点之后的 N 个节点。
- LeetCode 83 存在一个按升序排列的链表，请你删除所有重复的元素，使每个元素只出现一次。
- LeetCode 82 存在一个按升序排列的链表，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中没有重复出现的数字。
- LeetCode 1836. 从未排序链表中删除重复元素。

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。

2.5.1 删除特定结点

先看一个简单的问题，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 。

```
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：

```
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：

LeetCode 19. 删除链表的倒数第 N 个节点

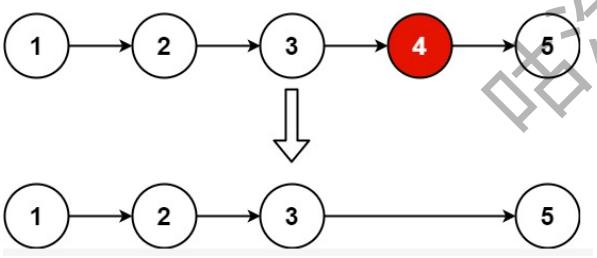
LeetCode 1474. 删除链表 M 个节点之后的 N 个节点。

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，前面已经介绍过，而这里不过是找到之后再将其删除。

2.5.3 删除倒数第n个结点

LeetCode19题要求：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]



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。

上面三种方法，第一种比较常规，第二种方法需要开辟一个O(n)的空间，还要考虑栈与链表的操作等，不中看也不中用。第三种方法一次遍历就行，用双指针也有逼格。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

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。代码如下：

```
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。代码如下：

```
public ListNode removeNthFromEnd(ListNode head, int n) {
      ListNode dummy = new ListNode(0);
       dummy.next=head;
       ListNode first = 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

我们继续看关于结点删除的题：

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。

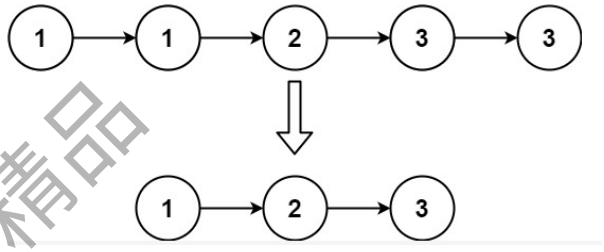
LeetCode 82 存在一个按升序排列的链表，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中没有重复出现的数字。

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，处理起来略有差别。
LeetCode 1836是在82的基础上将链表改成无序的了，难度要增加不少，感兴趣的同学请自己研究一下。

2.5.3.1 重复元素保留一个

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。

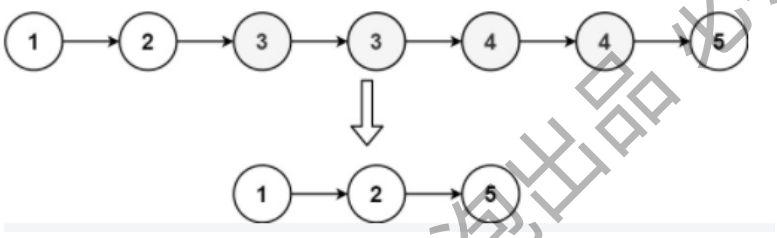
另外要注意的是 当我们遍历到链表的最后一个节点时，cur.next 为空节点，此时要加以判断，上代码：

```
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;
    }
```

2.5.3.2 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]
```



当一个都不要时，链表只要直接对cur.next 以及 cur.next.next 两个node进行比较就行了，这里要注意两个node可能为空，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;
    }
```

如果链表是未排序的该怎么办呢？如果先排序再操作代价太高了，感兴趣的同学可以继续研究一下

3 链表反转以及相关问题

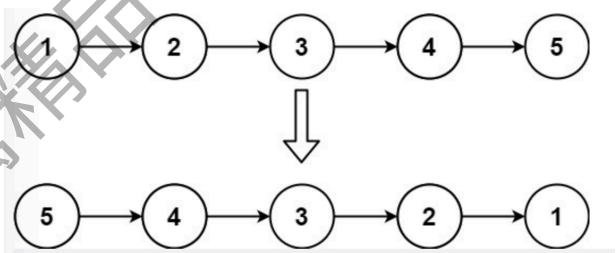
链表反转是一个出现频率特别高的算法题，笔者过去这些年面试，至少遇到过七八次。其中更夸张的是曾经两天写了三次，上午YY，下午金山云，第二天快手。链表反转在各大高频题排名网站也长期占领前三。比如牛客网上这个No1 好像已经很久了。所以链表反转是我们学习链表最重要的问题，没有之一。

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，能非常有效考察思维能力和代码驾驭能力。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。因为太重要，所以我们用一章专门研究这个题目。

3.1 反转链表

LeetCode206 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。

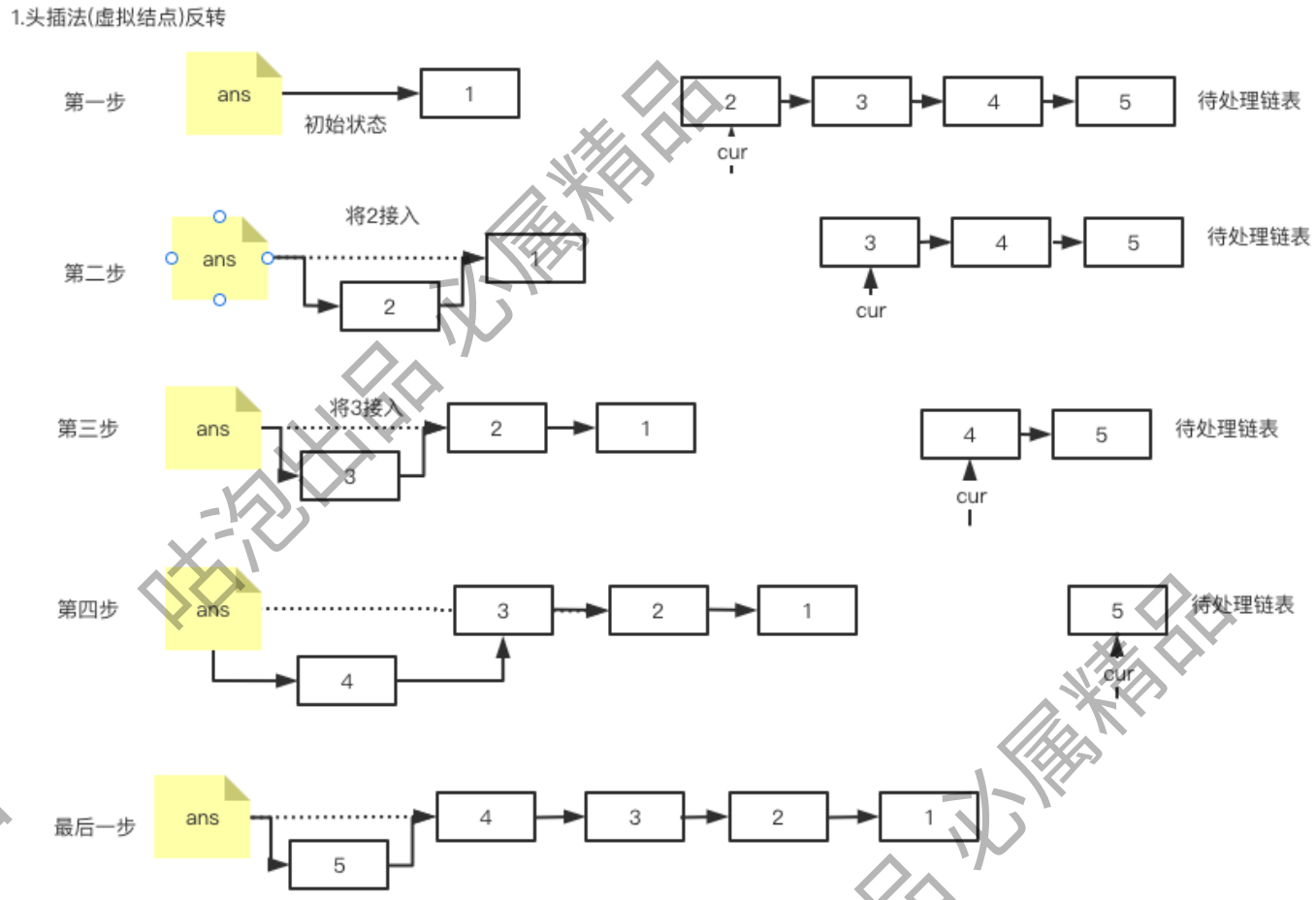
```
示例1:
输入: head = [1,2,3,4,5]
输出: [5,4,3,2,1]
```



本题有两种方法，带头结点和不带头结点，我们都应该会，因为这两种方式都很重要，如果搞清楚，很多链表的算法题就不用做了。

3.1.1 建立虚拟头结点辅助反转

前面分析链表插入元素的时候，会发现如何处理头结点是个比较麻烦的问题，为此可以先建立一个虚拟的结点ans，并且令ans.next=head，这样可以很好的简化我们的操作。如下图所示，如果我们将链表{1->2->3->4->5}进行反转，我们首先建立虚拟结点ans，并令ans.next=node(1)，接下来我们每次从旧的链表拆下来一个结点接到ans后面，然后将其他线调整好就可以了。



如上图所示，我们完成最后一步之后，只要返回ans.next就得到反转的链表了，代码如下：

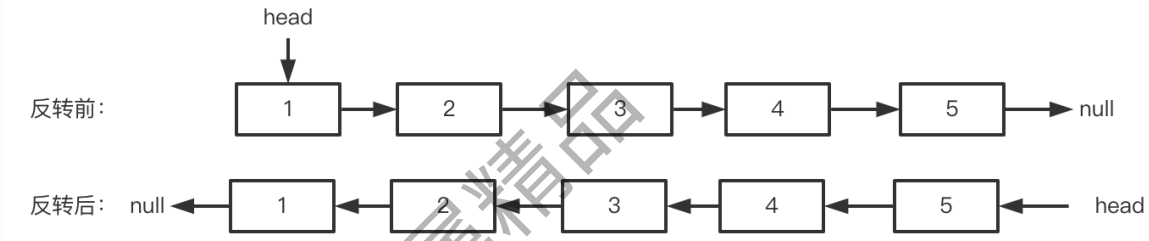
```
// 方法1：虚拟结点法
  public static ListNode reverseList(ListNode head) {
      ListNode ans = new ListNode(-1);
      ListNode cur = head;
      while (cur != null) {
          ListNode next = cur.next;
          cur.next = ans.next;
          ans.next = cur;
          cur = next;
      }
      return ans.next;
  }
```

建立虚拟结点是处理链表的经典方法之一，虚拟结点在很多工具的源码里都有使用，用来处理链表反转也比较好理解，因此我们必须掌握好。

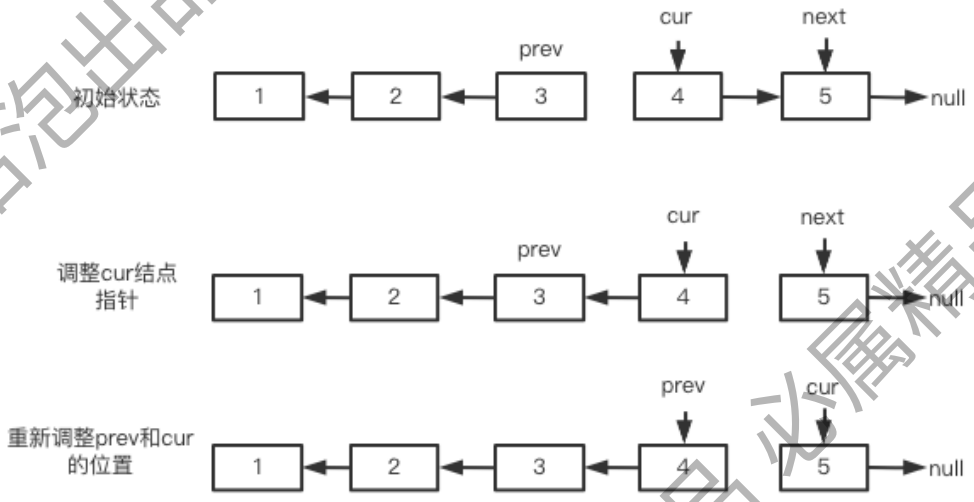
3.1.2 直接操作链表实现反转

上面的方式虽然好理解应用也广，但是可能会被面试官禁止，为啥？原因是不借助虚拟结点的方式更难，更能考察面试者的能力。

我们观察一下反转前后的结构和指针位置：



我们再看一下执行期间的过程示意图，在图中，cur本来指向旧链表的首结点，pre表示已经调整好的新链表的表头，next是下一个要调整的。注意图中箭头方向，cur和pre是两个表的表头，移动过程中cur经过一次中间状态之后，又重新变成了两个链表的表头。



理解这个图就够了，直接看代码：

```
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

3.1.3 小结

上面我们讲解了链表反转的两种方法，带虚拟头结点方法是很多底层源码使用的，而不使用带头结点的方法是面试经常要考的，所以两种方式我们都要好好掌握。

另外这种带与不带头结点的方式，在接下来的指定区间、K个一组反转也采用了，只不过为了便于理解，我们将其改成了“头插法”和“穿针引线法”。

拓展 通过递归来实现反转，链表反转还有第三种常见的方式，使用递归来实现，这里不做重点，感兴趣的同学可以研究一下：

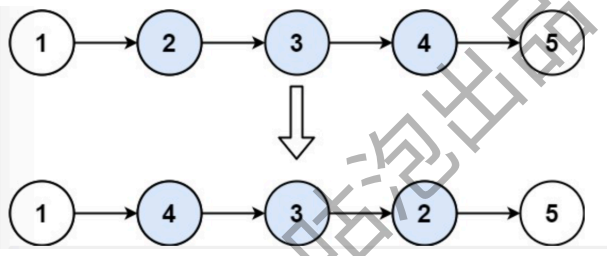
```
public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }
```

3.2 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]
```

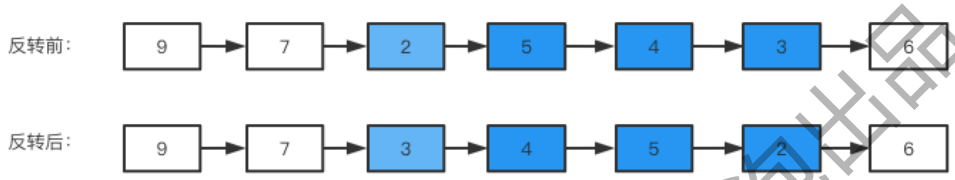
图示：



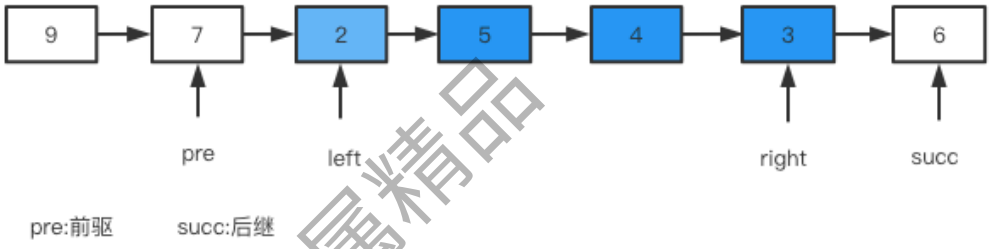
这里的处理方式也有多种，甚至给个名字都有点困难，干脆就分别叫穿针引线法和头插法吧。穿针引线本质上就是不带有节点的方式来实现反转，而头插法本质上就是带头结点的反转。

3.2.1 穿针引线法

我们以反转下图中蓝色区域的链表反转为例：

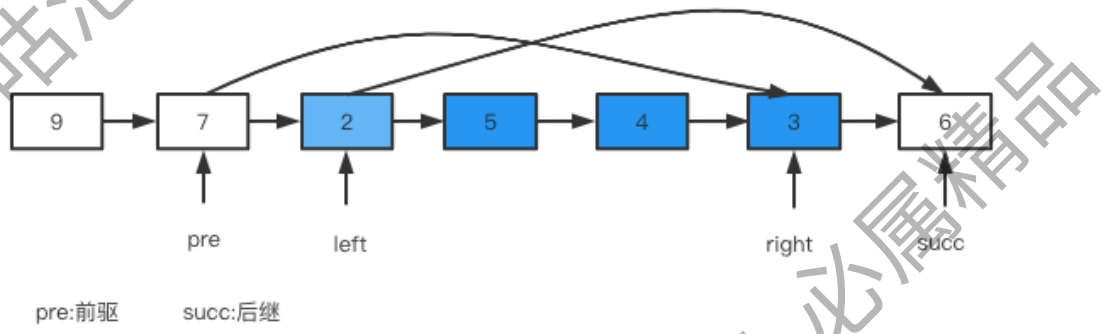


我们可以这么做：先确定好需要反转的部分，也就是下图的 `left` 到 `right` 之间，然后再将三段链表拼接起来。这种方式类似裁缝一样，找准位置减下来，再缝回去。这样问题就变成了如何标记下图四个位置，以及如何反转left到right之间的链表。



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。



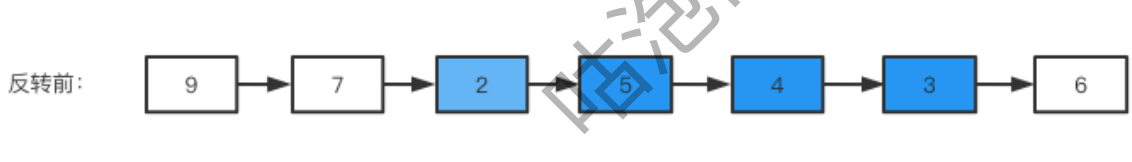
编码细节我们直接看下方代码。思路想明白以后，编码不是一件很难的事情。这里要提醒大家的是，链接什么时候切断，什么时候补上去，先后顺序一定要想清楚，如果想不清楚，可以在纸上模拟，思路清晰。

```
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：切出一个子链表
        ListNode leftNode = pre.next;
        ListNode succ = 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 = succ;
        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;
        }
    }
```

3.2.2 头插法

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，我们仍然以方法一的序列为例进行说明。



反转的整体思想是，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

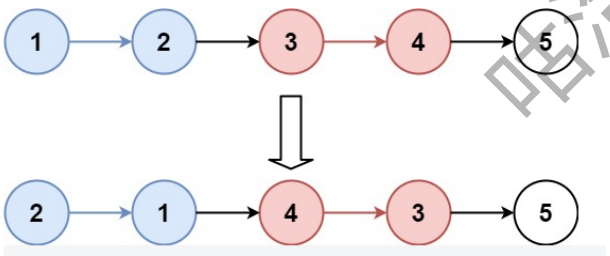
这个过程就是前面的带虚拟结点的插入操作，每走一步都要考虑各种指针怎么指，既要将结点摘下来接到对应的位置上，还要保证后续结点能够找到，请读者务必画图看一看，想一想到底该怎么调整。代码如下：

```
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;
}
```

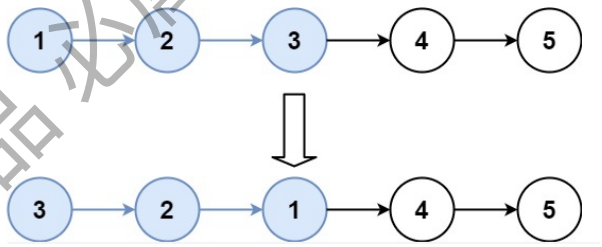
3.3 K个一组反转链表

可以说是链表中最难的一个问题了，LeetCode25.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]



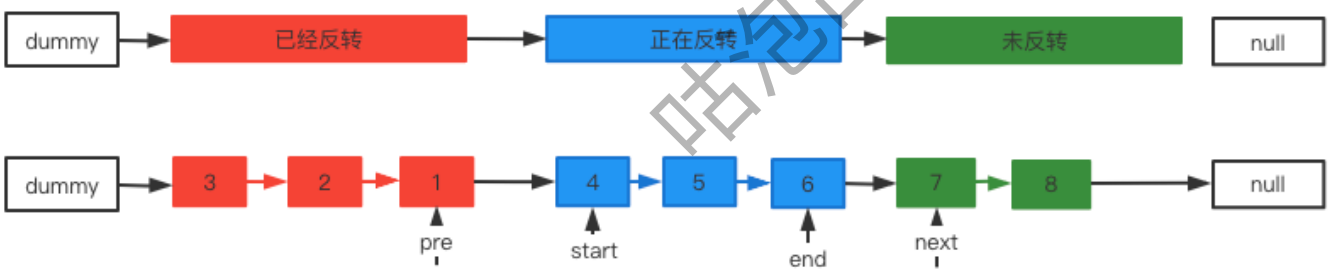
这个问题难点不在思路，而是具体实现，特别是每个段的首尾需要自动确定还要能接到移位。本题的思路就两种，要么是穿针引线法，要么是头插法。 这两种画出来的图示都非常复杂，还不如读者自己一边想一边画。这里只给出简洁的文字描述和实现代码。

3.3.1 穿针引线法

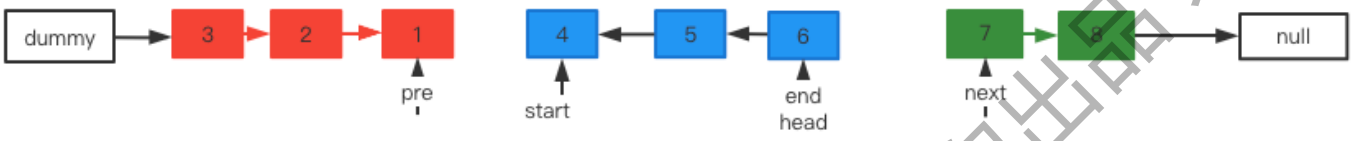
这种思路与上面的穿针引线类似，我们根据图示一点点看：

首先，因为要分组反转，我们就一组一组的处理，将其分成已经反转、正在反转和未反转三个部分，同时为了好处理头结点，我们新建一个虚拟头结点。

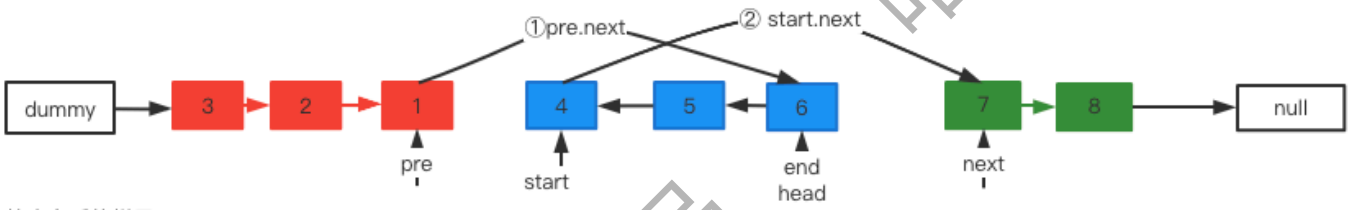
之后我们直接遍历，根据是否为K个找到四个关键位置，并用变量pre、start、end和next标记，如下：



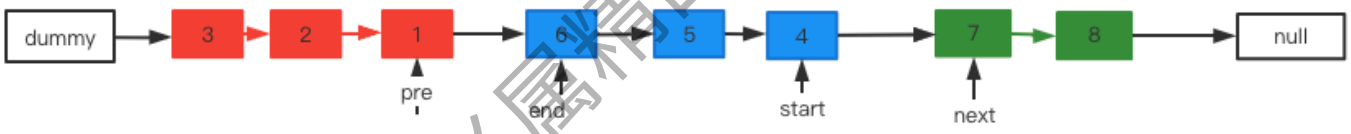
接着我们就可以对上图中颜色部分进行反转了，我们如果将end.next=null，那颜色部分可以直接复用前面3.1.2小节中的实现来完成反转。注意上图中指针的指向和几个变量的位置，head表示传给反转方法的参数，结构如下所示：



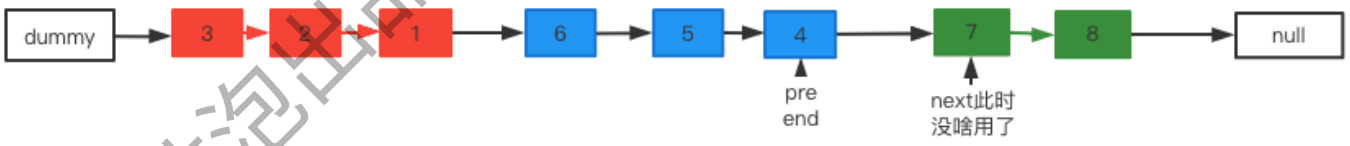
完成之后，我们要将裁下来的部分再缝到原始链表上，这就需要调整指针指向了，同样注意指针的指向



拉直之后的样子：



最后调整一下几个变量的位置，为下一次做准备：



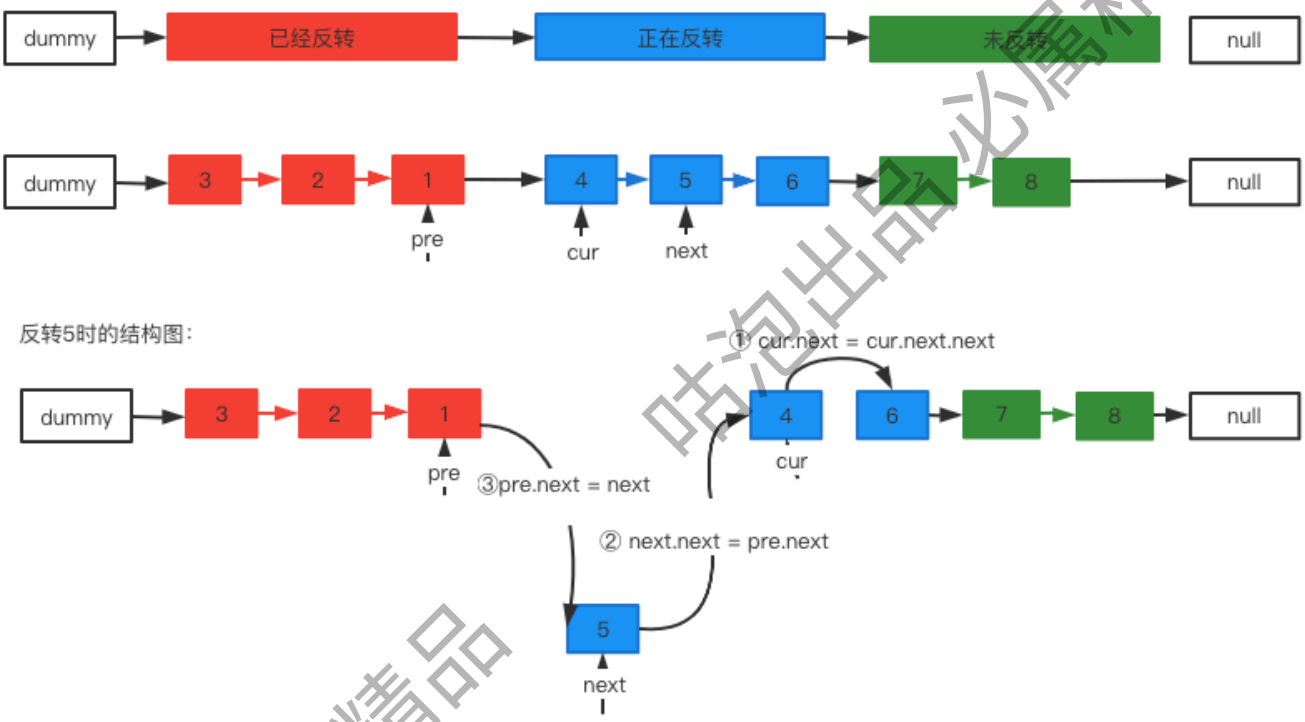
如果上述代码看懂了，整个过程就非常简单了， 实现代码：

```
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) {
    //找到要处理的区间的末尾
        for (int i = 0; i < k && end != null; i++){
           end = end.next;
        }
        if (end == null){
          break;
        }
        //将要处理的区间裁剪下来
        ListNode start = pre.next;
        ListNode next = end.next;
        end.next = null;
        //执行反转
        pre.next = reverse(start);
       // 上面pre.next和下面start.next两个指针是为了将反转的区间缝补回去
```

```
            start.next = next;
            //调整指针，为下一组做准备
            pre = start;
            end = pre;
        }
        return dummy.next;
    }

    //复用前面小节中的实现
    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;
    }
```

3.3.2 头插法

如果虚拟结点理解了，会发现头插法比上面的穿针引线法更好理解，主要思路还是将其分成已经反转、正在反转和未反转三个部分。为了方便循环，我们可以先遍历一遍数组，统计一下元素数量len，确定要分几组n=len/k，然后再采用与3.1.1小节一样的思路进行反转就可以了。下图表示的是反转结点5时的过程图：



实现代码：

```
public ListNode reverseKGroup(ListNode head, int k) {
     ListNode dummyNode = new ListNode(0);
       dummyNode.next = head;
       ListNode cur = head;
       int len = 0;//先计算出链表的长度
       while(cur != null) {
           len++;
           cur = cur.next;
       }
       int n = len/k;//计算出有几组
       ListNode pre = dummyNode;
       cur = head;
       for(int i=0;i<n;i++) {
           for(int j=0;j<k-1;j++) {
               ListNode next = cur.next;
               cur.next = cur.next.next;
               next.next = pre.next;
               pre.next = next;
           }
           pre = cur;
           cur = cur.next;
       }
       return dummyNode.next;
   }
```

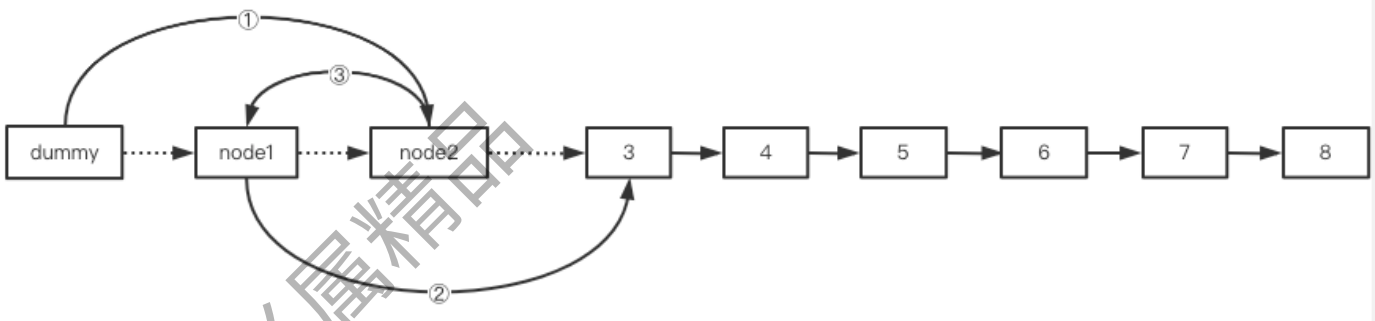
3.4 两两交换链表中的节点

LeetCode24.给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

如果要解决该问题，将上面小节的K换成2不就是这个题吗？道理确实如此，但是如果K为2的时候，可以不需要像K个一样需要先遍历找到区间的两端，而是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。

如果原始顺序是 dummy -> node1 -> node2，交换后面两个节点关系要变成 dummy -> node2 -> node1，事实上我们只要多执行一次next就可以拿到后面的元素，也就是类似node2 = temp.next.next这样的操作。

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。指针的调整可以参考如下图示：



完整代码是：

```
    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;
    }
```

3.5 链表反转的应用

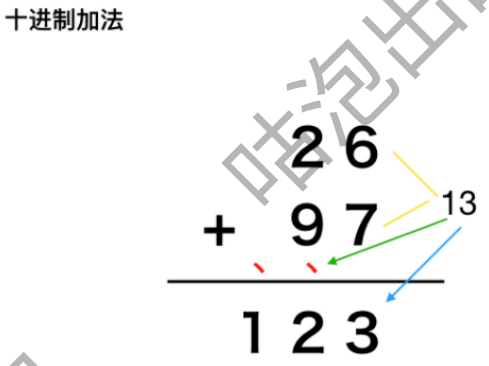
链表的反转我们研究了很多种情况，这几种都非常重要，但是这还不足征服链表反转，我们再看两个应用链表反转的例子。

3.5.1 单链表加1

LeetCode369.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]
```

我们先看一下加法的计算过程：



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。

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 。

```
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;
  }
```

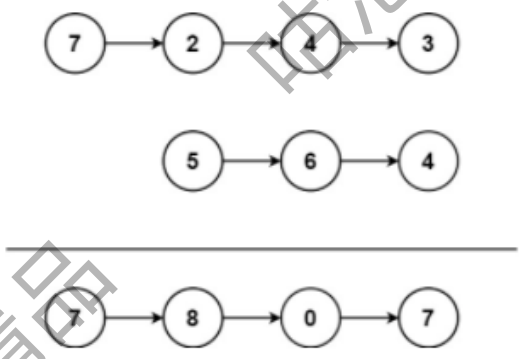
基于链表反转实现 如果这里不使用栈，使用链表反转来实现该怎么做呢？很显然，我们先将原始链表反转，这方面完成加1和进位等处理，完成之后再次反转。

本实现作为一个作业，请读者完成。要求是先将链表反转，得到结果之后再反转过来。反转方法要求使用3.1中介绍的反转方法。

3.5.2 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。

LeetCode445题，给你两个非空链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。你可以假设除了数字 0 之外，这两个数字都不会以零开头。示例：



```
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。

怎么反转呢？栈和链表反转都可以，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。

```
public static ListNode addInListByStack(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

如果使用链表反转，先将两个链表分别反转，最后计算完之后再将结果反转，一共有三次反转操作，所以必然将反转抽取出一个方法比较好，代码如下：

```
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。

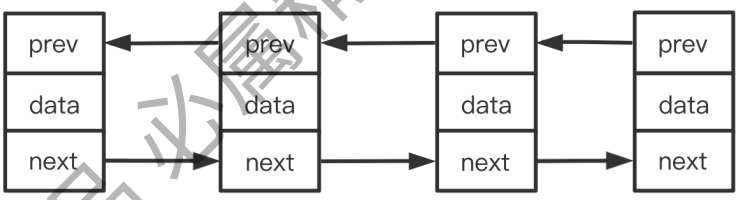
既然加法可以，那如果是减法呢？读者可以自己想想该怎么处理。

4 双向链表设计

参考实现：DoublyLinkList

4.1 基本设计

双向链表顾名思义就是既可以向前，也可以向后。有两个指针的好处自然是移动元素更方便。该结构我们在设计LRU缓存时也会遇到，所以这里简单看一下。



双向链表的结点：

```
    class DoubleNode {
        public int data;    //数据域
        public DoubleNode next;    //指向下一个结点
        public DoubleNode prev;
        public DoubleNode(int data) {
            this.data = data;
        }
        //打印结点的数据域
        public void displayNode() {
            System.out.print("{" + data + "} ");
        }
    }
```

我们再定义一下双向链表的结构和遍历的方法：

```
public class DoublyLinkList {
    private DoubleNode first;
    private DoubleNode last;
    public DoublyLinkList() {
        first = null;
        last = first;
    }
    //从头部开始打印
    public void displayForward() {
        System.out.print("List(first--->last): ");
```

```
        DoubleNode current = first;
        while (current != null) {
            current.displayNode();
            current = current.next;
        }
        System.out.println();
    }

    //从尾部开始演绎
    public void displayBackward() {
        System.out.print("List(last--->first): ");
        DoubleNode current = last;
        while (current != null) {
            current.displayNode();
            current = current.prev;
        }
        System.out.println();
    }
  }
```

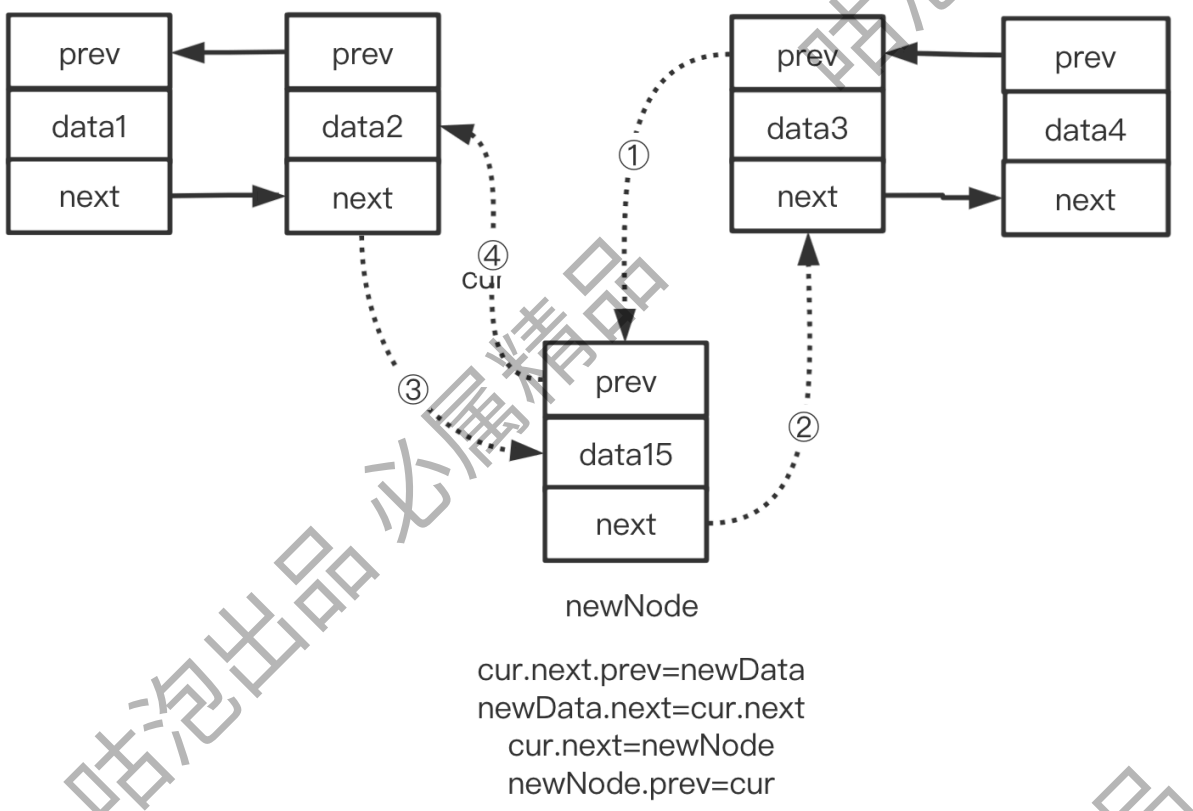
4.2 插入元素

操作双向链表的方法稍微复杂一些，而且头部、尾部和中间位置的操作有较大的区别，我们分别看。我们先插入，在头部和尾部插入的方式稍微简单，直接上代码：

```
//头部插入
public void insertFirst(int data) {
    DoubleNode newDoubleNode = new DoubleNode(data);
    if (isEmpty()) {
        last = newDoubleNode;
    } else {//如果不是第一个结点的情况
      //将还没插入新结点之前链表的第一个结点的previous指向newNode
        first.prev = newDoubleNode;
    }
    newDoubleNode.next = first;
    //将新结点赋给first（链接）成为第一个结点
    first = newDoubleNode;
}

//尾部插入
public void insertLast(int data) {
    DoubleNode newDoubleNode = new DoubleNode(data);
    if (isEmpty()) {
        first = newDoubleNode;
    } else {
        newDoubleNode.prev = last;
        last.next = newDoubleNode;
    }
    //由于插入了一个新的结点，又因为是尾部插入，所以将last指向newNode
```

```
    last = newDoubleNode;
}
```

我们再看一下在中间位置插入的情况：



找到插入的位置之后，我们需要给newNode接四根线，假如上图中data2是当前结点，你能分析出连线的顺序吗？（提示：不止一种情况）

```
public void insertAfter(int key, int data) {
    DoubleNode newDoubleNode = new DoubleNode(data);
    DoubleNode current = first;
    while ((current != null) && (current.data != key)) {
        current = current.next;
    }
    //若当前结点current为空
    if (current == null) {
        if (isEmpty()) {
            first = newDoubleNode;
            last = newDoubleNode;
        } else {
         //2、找不到key值，则在链表尾部插入一个新的结点
            last.next = newDoubleNode;
            newDoubleNode.prev = last;
            last = newDoubleNode;
        }
    } else {//第3种情况，找到了key值，分两种情况
        if (current == last) {
          //1、key值与最后结点的data相等
```

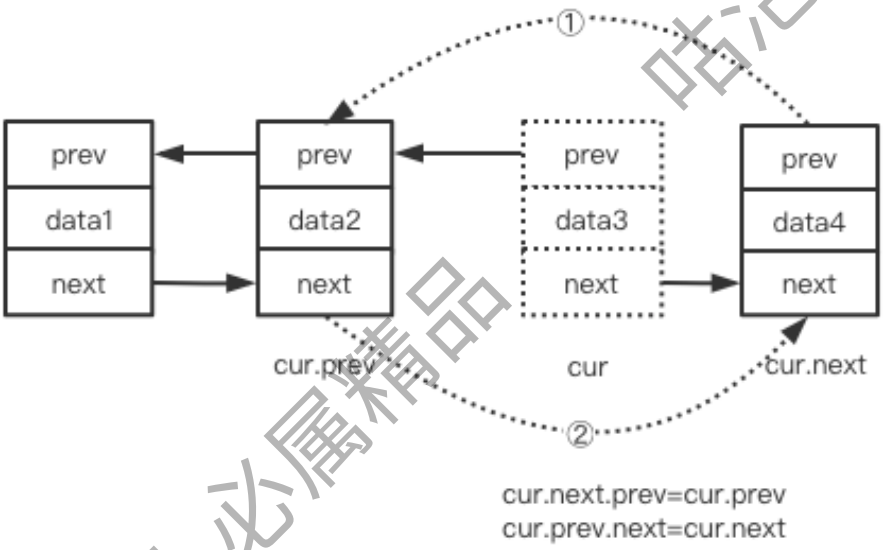
```
            newDoubleNode.next = null;
            last = newDoubleNode;
        } else {
           //2、两结点中间插入
            newDoubleNode.next = current.next;

            current.next.prev = newDoubleNode;
        }
        current.next = newDoubleNode;
        newDoubleNode.prev = current;
    }
}
```

4.3 删除元素

双向链表的不足就是增删改的时候，需要修改的指针多了，操作更麻烦了。由于双向链表在算法中不是很重要，我们先看一下删除的大致过程。首尾元素的删除还比较简单，直接上代码：

```
//删除首元素
public DoubleNode deleteFirst() {
    DoubleNode temp = first;
    //若链表只有一个结点，删除后链表为空，将last指向null
    if (first.next == null) {
        last = null;
    } else {
    //若链表有两个及以上的结点 ，因为是头部插入，则first.next将变成第一个结点，其previous将变成null
        first.next.prev = null;
    }
    //将first.next赋给first
    first = first.next;
    //返回删除的结点
    return temp;
}

//从尾部删除结点
public DoubleNode deleteLast() {
    DoubleNode temp = last;
    //如果链表只有一个结点，则删除以后为空表,last指向null
    if (first.next == null) {
        first = null;
    } else {
    //将上一个结点的next域指向null
        last.prev.next = null;
    }
    //上一个结点称为最后一个结点，last指向它
    last = last.prev;
    //返回删除的结点
    return temp;
}
```

我们再看删除中间元素的情况，要标记出几个关键结点的位置，也就是图中的cur，cur.next和cur.prev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，图示是这样的：



我们只需要调整两个指针，一个是cur.next的prev指向cur.prev，第二个是cur.prev的next指向cur.next。此时cur结点没有结点访问了，根据垃圾回收算法，此时cur就变得不可达，最终被回收掉，所以这样就完成了删除cur的操作。**想一下，这里调整两条线的代码是否可以换顺序?

```
public DoubleNode deleteKey(int key) {
    DoubleNode current = first;
    //遍历链表寻找该值所在的结点
    while (current != null && current.data != key) {
        current = current.next;
    }
    //若当前结点指向null则返回null,
    if (current == null) {
        return null;
    } else {
      //如果current是第一个结点
        if (current == first) {
        //则将first指向它，将该结点的previous指向null,其余不变
            first = current.next;
            current.next.prev = null;
        } else if (current == last) {
          //如果current是最后一个结点
            last = current.prev;
            current.prev.next = null;
        } else {
           //当前结点的上一个结点的next域应指向当前的下一个结点
            current.prev.next = current.next;
            //当前结点的下一个结点的previous域应指向当前结点的上一个结点
            current.next.prev = current.prev;
        }
```

```
    }
    return current;         //返回
}
```

5.大厂算法实战

通过上面的这些题目，我们能感受到链表的题目都是从增删改查变换或者组合而来的。而且不管是大厂还是小厂，基本都是这些。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。

上面的题目虽然比较大，但是精华题目是这三道：

LeetCode707 设计链表：设计一个链表，将链表的基本问题搞明白。

LeetCode141 判断链表中是否有环，这个是双指针思想在链表中的典型应用。

LeetCode25 K个一组反转链表，反转是链表的大重点，而K个一组是最难的反转。

在上面这些题目中，需要再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。