

# 双指针技巧秒杀七道链表题目

 Stars 107k  B站 @labuladong 配套PDF和插件  下载 打卡挑战  报名 精品课程  查看




微信搜一搜

labuladong公众号

**通知：**数据结构精品课持续更新中，[详情见这里](#)。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	19. Remove Nth Node From End of List	19. 删除链表的倒数第 N 个结点	
-	21. Merge Two Sorted Lists	21. 合并两个有序链表	
-	23. Merge k Sorted Lists	23. 合并K个升序链表	
-	86. Partition List	86. 分隔链表	
-	141. Linked List Cycle	141. 环形链表	
-	142. Linked List Cycle II	142. 环形链表 II	
-	160. Intersection of Two Linked Lists	160. 相交链表	

牛客	LeetCode	力扣	难度
-	876. Middle of the Linked List	876. 链表的中间结点	

上次在视频号直播，跟大家说到单链表有很多巧妙的操作，本文就总结一下单链表的基本技巧，每个技巧都对应着至少一道算法题：

- 1、合并两个有序链表
- 2、链表的分解
- 3、合并 **k** 个有序链表
- 4、寻找单链表的倒数第 **k** 个节点
- 5、寻找单链表的中点
- 6、判断单链表是否包含环并找出环起点
- 7、判断两个单链表是否相交并找出交点

这些解法都用到了双指针技巧，所以说对于单链表相关的题目，双指针的运用是非常广泛的，下面我们就来一个一个看。

# 合并两个有序链表

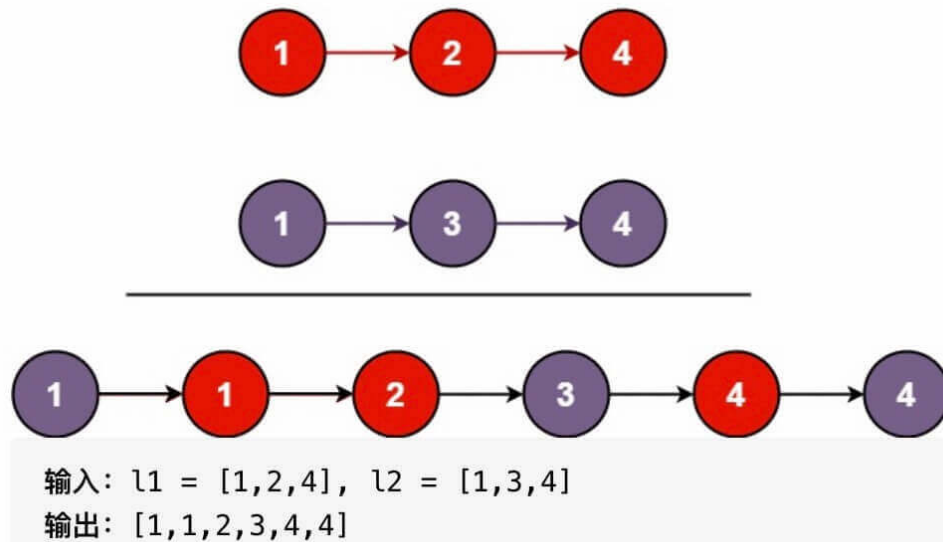
这是最基本的链表技巧，力扣第 21 题「[合并两个有序链表](#)」就是这个问题：

## 21. 合并两个有序链表 labuladong 题解 思路

难度 简单 1846 ☆ 17A 1 1

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。

示例 1:



给你输入两个有序链表，请你把他俩合并成一个新的有序链表，函数签名如下：

```
ListNode mergeTwoLists(ListNode l1, ListNode l2);
```

这题比较简单，我们直接看解法：

```
ListNode mergeTwoLists(ListNode l1, ListNode l2) {  
    // 虚拟头结点  
    ListNode dummy = new ListNode(-1), p = dummy;  
    ListNode p1 = l1, p2 = l2;  
  
    while (p1 != null && p2 != null) {  
        // 比较 p1 和 p2 两个指针
```

```

// 将值较小的节点接到 p 指针
if (p1.val > p2.val) {
    p.next = p2;
    p2 = p2.next;
} else {
    p.next = p1;
    p1 = p1.next;
}
// p 指针不断前进
p = p.next;
}

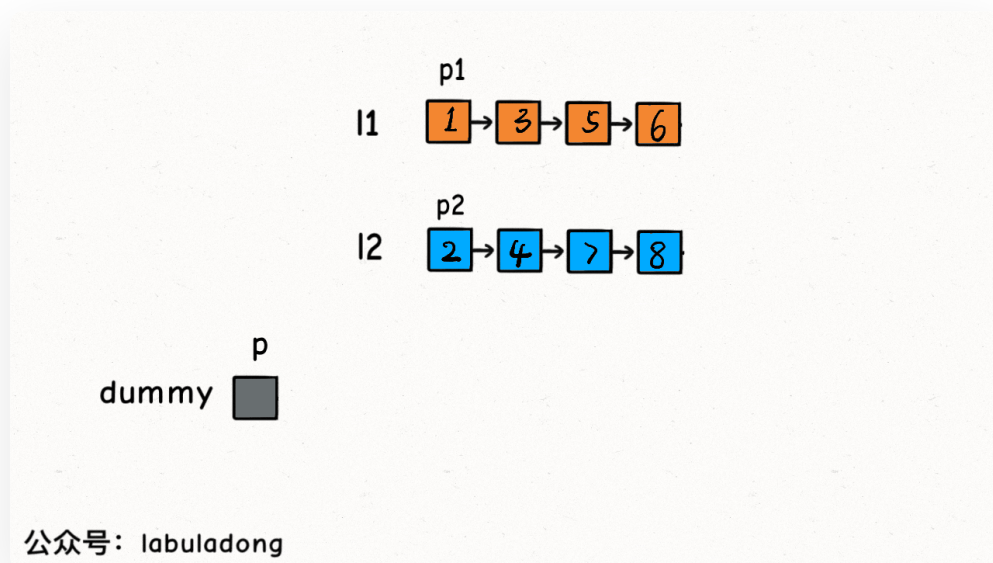
if (p1 != null) {
    p.next = p1;
}

if (p2 != null) {
    p.next = p2;
}

return dummy.next;
}

```

我们的 while 循环每次比较 `p1` 和 `p2` 的大小，把较小的节点接到结果链表上，看如下 GIF：



形象地理解，这个算法的逻辑类似于拉拉链，`l1`, `l2` 类似于拉链两侧的锯齿，指针 `p` 就好像拉链的拉索，将两个有序链表合并；或者说这个过程像蛋白酶合成蛋白质，`l1`, `l2` 就好比两条氨基

酸，而指针 `p` 就好像蛋白酶，将氨基酸组合成蛋白质。

代码中还用到一个链表的算法题中是很常见的「虚拟头结点」技巧，也就是 `dummy` 节点。你可以试试，如果不使用 `dummy` 虚拟节点，代码会复杂很多，而有了 `dummy` 节点这个占位符，可以避免处理空指针的情况，降低代码的复杂性。

## 单链表的分解

直接看下力扣第 86 题「分隔链表」：

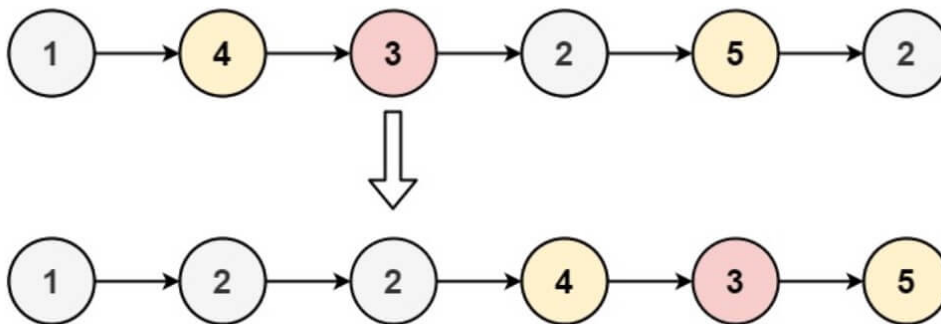
### 86. 分隔链表

难度 中等 569 ☆ 图标 文 A 铃 对话框

给你一个链表的头节点 `head` 和一个特定值 `x`，请你对链表进行分隔，使得所有小于 `x` 的节点都出现在大于或等于 `x` 的节点之前。

你应当保留两个分区中每个节点的初始相对位置。

示例 1：



输入: `head = [1,4,3,2,5,2]`, `x = 3`

输出: `[1,2,2,4,3,5]`

在合并两个有序链表时让你合二为一，而这里需要分解让你把原链表一分为二。具体来说，我们可以把原链表分成两个小链表，一个链表中的元素大小都小于 `x`，另一个链表中的元素都大于等于 `x`，最后再把这两条链表接到一起，就得到了题目想要的结果。

整体逻辑和合并有序链表非常相似，细节直接看代码吧，注意虚拟头结点的运用：

```

ListNode partition(ListNode head, int x) {
    // 存放小于 x 的链表的虚拟头结点
    ListNode dummy1 = new ListNode(-1);
    // 存放大于等于 x 的链表的虚拟头结点
    ListNode dummy2 = new ListNode(-1);
    // p1, p2 指针负责生成结果链表
    ListNode p1 = dummy1, p2 = dummy2;
    // p 负责遍历原链表, 类似合并两个有序链表的逻辑
    // 这里是将一个链表分解成两个链表
    ListNode p = head;
    while (p != null) {
        if (p.val >= x) {
            p2.next = p;
            p2 = p2.next;
        } else {
            p1.next = p;
            p1 = p1.next;
        }
        // 断开原链表中的每个节点的 next 指针
        ListNode temp = p.next;
        p.next = null;
        p = temp;
    }
    // 连接两个链表
    p1.next = dummy2.next;

    return dummy1.next;
}

```

## 合并 k 个有序链表

看下力扣第 23 题「[合并K个升序链表](#)」：

## 23. 合并K个升序链表

labuladong 题解

思路

难度 困难

👍 1448



给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

输入：lists = [[1,4,5],[1,3,4],[2,6]]

输出：[1,1,2,3,4,4,5,6]

解释：链表数组如下：

```
[  
  1->4->5,  
  1->3->4,  
  2->6  
]
```

将它们合并到一个有序链表中得到。

1->1->2->3->4->4->5->6

函数签名如下：

```
ListNode mergeKLists(ListNode[] lists);
```

合并  $k$  个有序链表的逻辑类似合并两个有序链表，难点在于，如何快速得到  $k$  个节点中的最小节点，接到结果链表上？

这里我们就要用到 优先级队列（二叉堆） 这种数据结构，把链表节点放入一个最小堆，就可以每次获得  $k$  个节点中的最小节点：

```
ListNode mergeKLists(ListNode[] lists) {  
    if (lists.length == 0) return null;  
    // 虚拟头结点
```

```

ListNode dummy = new ListNode(-1);
ListNode p = dummy;
// 优先级队列, 最小堆
PriorityQueue<ListNode> pq = new PriorityQueue<>(
    lists.length, (a, b) -> (a.val - b.val));
// 将 k 个链表的头结点加入最小堆
for (ListNode head : lists) {
    if (head != null)
        pq.add(head);
}

while (!pq.isEmpty()) {
    // 获取最小节点, 接到结果链表中
    ListNode node = pq.poll();
    p.next = node;
    if (node.next != null) {
        pq.add(node.next);
    }
    // p 指针不断前进
    p = p.next;
}
return dummy.next;
}

```

这个算法是面试常考题，它的时间复杂度是多少呢？

优先队列 `pq` 中的元素个数最多是 `k`，所以一次 `poll` 或者 `add` 方法的时间复杂度是  $O(\log k)$ ；所有的链表节点都会被加入和弹出 `pq`，所以算法整体的时间复杂度是  $O(N \log k)$ ，其中 `k` 是链表的条数，`N` 是这些链表的节点总数。

## 单链表的倒数第 `k` 个节点

从前往后寻找单链表的第 `k` 个节点很简单，一个 `for` 循环遍历过去就找到了，但是如何寻找从后往前数的第 `k` 个节点呢？

那你可能说，假设链表有 `n` 个节点，倒数第 `k` 个节点就是正数第 `n - k + 1` 个节点，不也是一个 `for` 循环的事儿吗？

是的，但是算法题一般只给你一个 `ListNode` 头结点代表一条单链表，你不能直接得出这条链表的长度 `n`，而需要先遍历一遍链表算出 `n` 的值，然后再遍历链表计算第 `n - k + 1` 个节点。

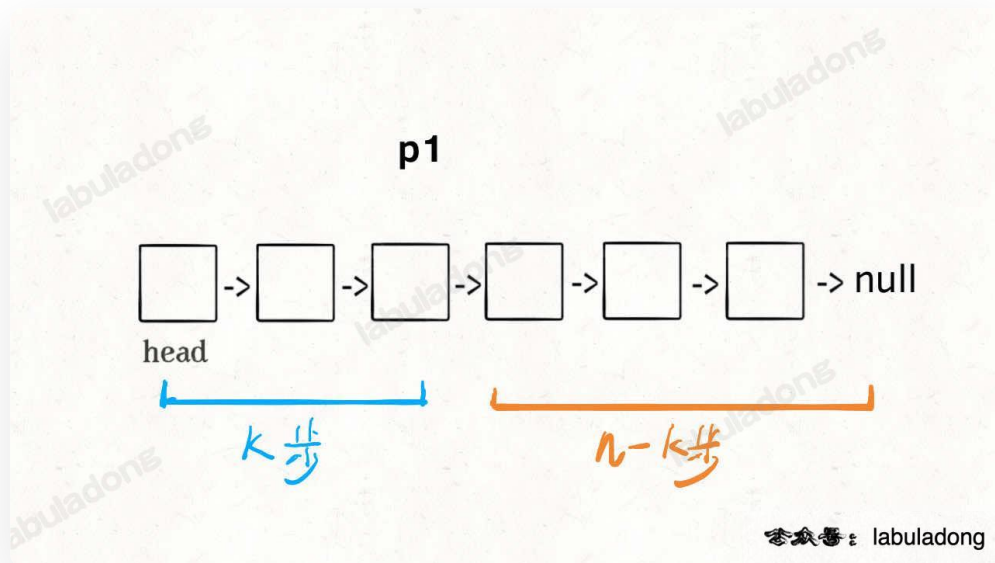


也就是说，这个解法需要遍历两次链表才能得到倒数第  $k$  个节点。

那么，我们能不能**只遍历一次链表**，就算出倒数第  $k$  个节点？可以做到的，如果是面试问到这道题，面试官肯定也是希望你给出只需遍历一次链表的解法。

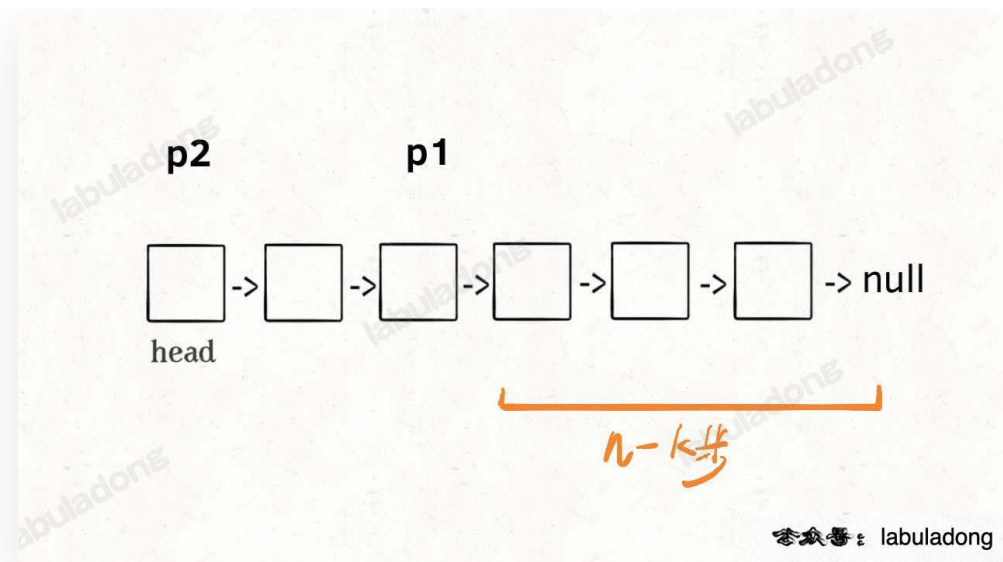
这个解法就比较巧妙了，假设  $k = 2$ ，思路如下：

首先，我们先让一个指针  $p1$  指向链表的头节点  $head$ ，然后走  $k$  步：

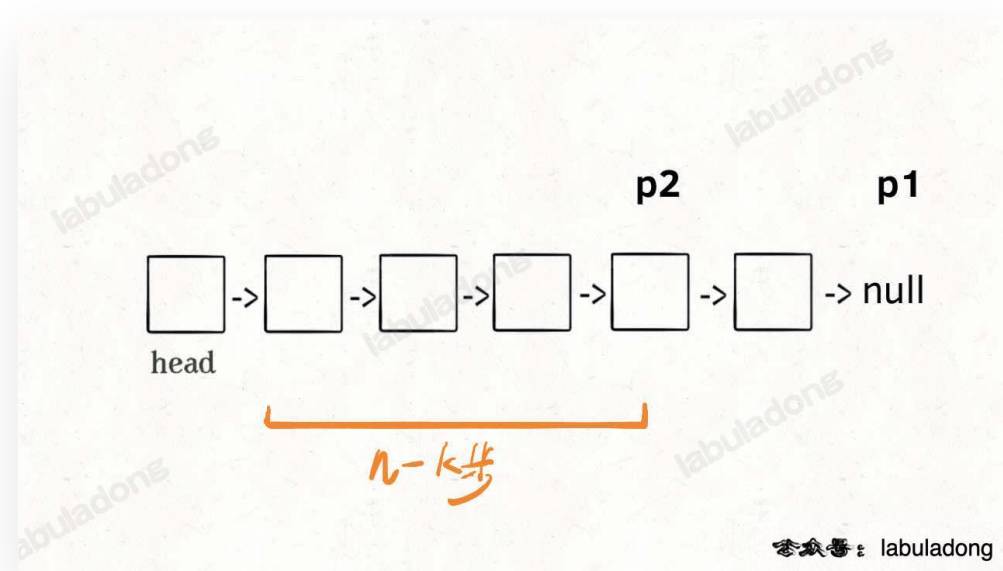


现在的  $p1$ ，只要再走  $n - k$  步，就能走到链表末尾的空指针了对吧？

趁这个时候，再用一个指针  $p2$  指向链表头节点  $head$ ：



接下来就很显然了，让 **p1** 和 **p2** 同时向前走，**p1** 走到链表末尾的空指针时前进了  $n - k$  步，**p2** 也从 **head** 开始前进了  $n - k$  步，停留在第  $n - k + 1$  个节点上，即恰好停链表的倒数第  $k$  个节点上：



这样，只遍历了一次链表，就获得了倒数第  $k$  个节点 **p2**。

上述逻辑的代码如下：

```
// 返回链表的倒数第  $k$  个节点
```

```

ListNode findFromEnd(ListNode head, int k) {
    ListNode p1 = head;
    // p1 先走 k 步
    for (int i = 0; i < k; i++) {
        p1 = p1.next;
    }
    ListNode p2 = head;
    // p1 和 p2 同时走 n - k 步
    while (p1 != null) {
        p2 = p2.next;
        p1 = p1.next;
    }
    // p2 现在指向第 n - k 个节点
    return p2;
}

```

当然，如果用 big O 表示法来计算时间复杂度，无论遍历一次链表和遍历两次链表的时间复杂度都是  $O(N)$ ，但上述这个算法更有技巧性。

很多链表相关的算法题都会用到这个技巧，比如说力扣第 19 题「删除链表的倒数第 N 个结点」：

19. 删除链表的倒数第 N 个结点
labuladong 题解
思路

难度 中等
1506
收藏
分享
切换为英文
接收动态
反馈

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：

```

graph LR
    1((1)) --> 2((2))
    2 --> 3((3))
    3 --> 4((4))
    4 --> 5((5))
    style 4 fill:#ff0000
    4 --> 5
    4 --> 3
    3 --> 5
    
```

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

我们直接看解法代码：

```
// 主函数
public ListNode removeNthFromEnd(ListNode head, int n) {
    // 虚拟头结点
    ListNode dummy = new ListNode(-1);
    dummy.next = head;
    // 删除倒数第 n 个，要先找倒数第 n + 1 个节点
    ListNode x = findFromEnd(dummy, n + 1);
    // 删掉倒数第 n 个节点
    x.next = x.next.next;
    return dummy.next;
}

private ListNode findFromEnd(ListNode head, int k) {
    // 代码见上文
}
```

这个逻辑就很简单了，要删除倒数第  $n$  个节点，就得获得倒数第  $n + 1$  个节点的引用，可以用我们实现的 `findFromEnd` 来操作。

不过注意我们又使用了虚拟头结点的技巧，也是为了防止出现空指针的情况，比如说链表总共有 5 个节点，题目就让你删除倒数第 5 个节点，也就是第一个节点，那按照算法逻辑，应该首先找到倒数第 6 个节点。但第一个节点前面已经没有节点了，这就会出错。

但有了我们虚拟节点 `dummy` 的存在，就避免了这个问题，能够对这种情况进行正确的删除。

## 单链表的中点

力扣第 876 题「[链表的中间结点](#)」就是这个题目，问题的关键也在于我们无法直接得到单链表的长度  $n$ ，常规方法也是先遍历链表计算  $n$ ，再遍历一次得到第  $n / 2$  个节点，也就是中间节点。

如果想一次遍历就得到中间节点，也需要耍点小聪明，使用「快慢指针」的技巧：

我们让两个指针 `slow` 和 `fast` 分别指向链表头结点 `head`。

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步，这样，当 `fast` 走到链表末尾时，`slow` 就指向了链表 midpoint。

上述思路的代码实现如下：

```
ListNode middleNode(ListNode head) {  
    // 快慢指针初始化指向 head  
    ListNode slow = head, fast = head;  
    // 快指针走到末尾时停止  
    while (fast != null && fast.next != null) {  
        // 慢指针走一步，快指针走两步  
        slow = slow.next;  
        fast = fast.next.next;  
    }  
    // 慢指针指向中点  
    return slow;  
}
```

需要注意的是，如果链表长度为偶数，也就是说中点有两个的时候，我们这个解法返回的节点是靠后的那个节点。

另外，这段代码稍加修改就可以直接用到判断链表成环的算法题上。

## 判断链表是否包含环

判断链表是否包含环属于经典问题了，解决方案也是用快慢指针：

每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步。

如果 `fast` 最终遇到空指针，说明链表中没有环；如果 `fast` 最终和 `slow` 相遇，那肯定是 `fast` 超过了 `slow` 一圈，说明链表中含有环。

只需要把寻找链表中点的代码稍加修改就行了：

```
boolean hasCycle(ListNode head) {  
    // 快慢指针初始化指向 head  
    ListNode slow = head, fast = head;  
    // 快指针走到末尾时停止  
    while (fast != null && fast.next != null) {  
        // 慢指针走一步，快指针走两步
```

```

        slow = slow.next;
        fast = fast.next.next;
        // 快慢指针相遇, 说明含有环
        if (slow == fast) {
            return true;
        }
    }
    // 不包含环
    return false;
}

```

当然，这个问题还有进阶版：如果链表中含有环，如何计算这个环的起点？

这里简单提一下解法：

```

ListNode detectCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
        if (fast == slow) break;💡
    }
    // 上面的代码类似 hasCycle 函数
    if (fast == null || fast.next == null) {
        // fast 遇到空指针说明没有环
        return null;
    }

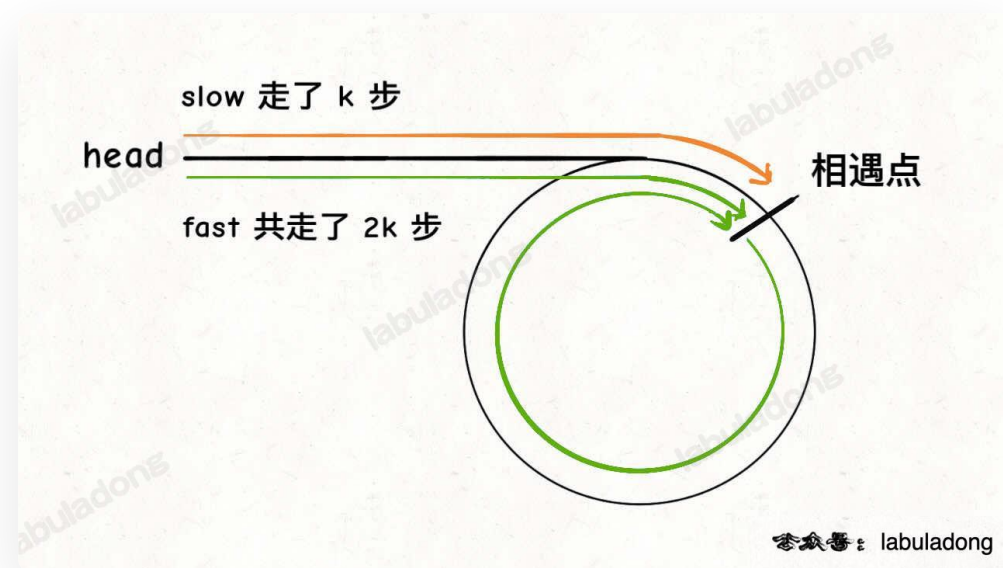
    // 重新指向头结点
    slow = head;💡
    // 快慢指针同步前进, 相交点就是环起点
    while (slow != fast) {
        fast = fast.next;
        slow = slow.next;
    }
    return slow;
}

```

可以看到，当快慢指针相遇时，让其中任一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。

为什么要这样呢？这里简单说一下其中的原理。

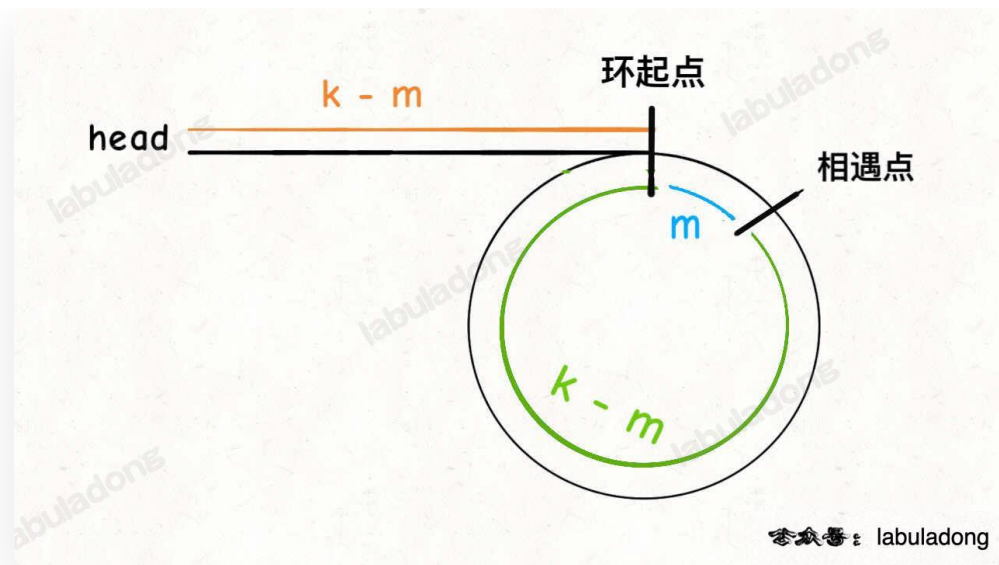
我们假设快慢指针相遇时，慢指针 `slow` 走了  $k$  步，那么快指针 `fast` 一定走了  $2k$  步：



`fast` 一定比 `slow` 多走了  $k$  步，这多走的  $k$  步其实就是 `fast` 指针在环里转圈圈，所以  $k$  的值就是环长度的「整数倍」。

假设相遇点距环的起点的距离为  $m$ ，那么结合上图的 `slow` 指针，环的起点距头结点 `head` 的距离为  $k - m$ ，也就是说如果从 `head` 前进  $k - m$  步就能到达环起点。

巧的是，如果从相遇点继续前进  $k - m$  步，也恰好到达环起点。因为结合上图的 `fast` 指针，从相遇点开始走  $k$  步可以转回到相遇点，那走  $k - m$  步肯定就走到环起点了：



所以，只要我们把快慢指针中的任一个重新指向 `head`，然后两个指针同速前进，`k - m` 步后一定会相遇，相遇之处就是环的起点了。

## 两个链表是否相交

这个问题有意思，也是力扣第 160 题「[相交链表](#)」函数签名如下：

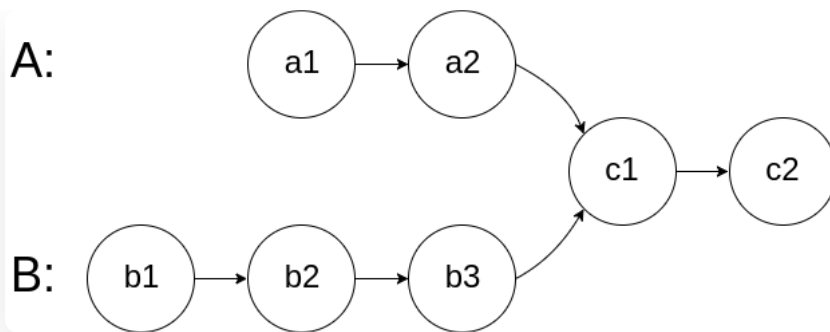
```
ListNode getIntersectionNode(ListNode headA, ListNode headB);
```

给你输入两个链表的头结点 `headA` 和 `headB`，这两个链表可能存在相交。

如果相交，你的算法应该返回相交的那个节点；如果没相交，则返回 `null`。

比如题目给我们举的例子，如果输入的两个链表如下图：



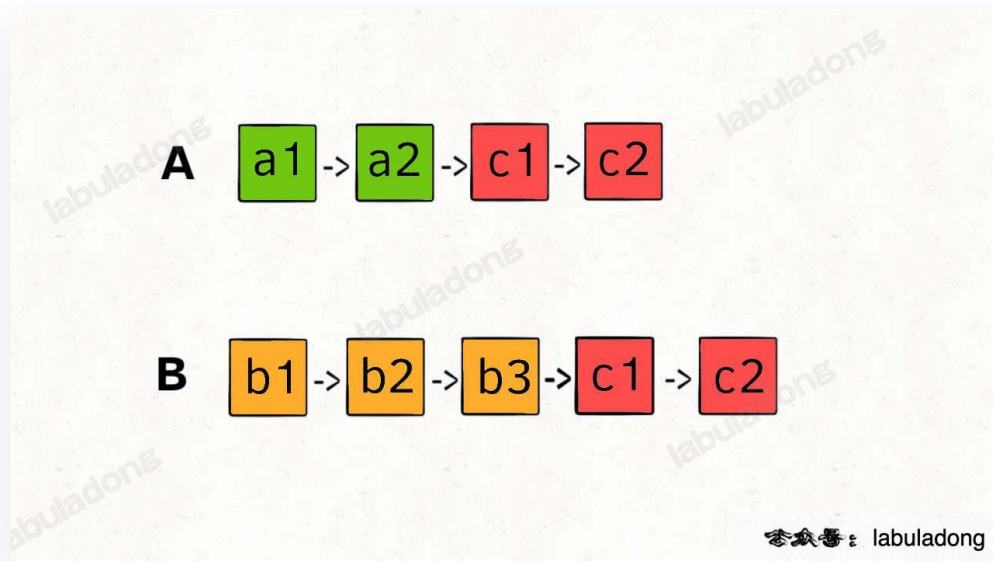


那么我们的算法应该返回 `c1` 这个节点。

这个题直接的想法可能是用 `HashSet` 记录一个链表的所有节点，然后和另一条链表对比，但这就需要额外的空间。

如果不用额外的空间，只使用两个指针，你如何做呢？

难点在于，由于两条链表的长度可能不同，两条链表之间的节点无法对应：

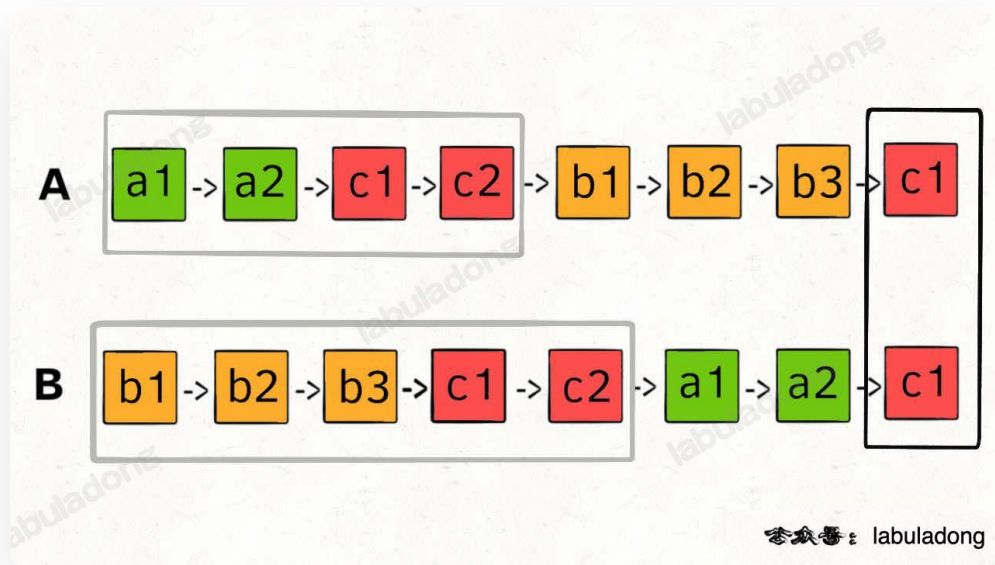


如果用两个指针 `p1` 和 `p2` 分别在两条链表上前进，并不能同时走到公共节点，也就无法得到相交节点 `c1`。

解决这个问题的关键是，通过某些方式，让 `p1` 和 `p2` 能够同时到达相交节点 `c1`。

所以，我们可以让 `p1` 遍历完链表 A 之后开始遍历链表 B，让 `p2` 遍历完链表 B 之后开始遍历链表 A，这样相当于「逻辑上」两条链表接在了一起。

如果这样进行拼接，就可以让 `p1` 和 `p2` 同时进入公共部分，也就是同时到达相交节点 `c1`：



那你可能会问，如果说两个链表没有相交点，是否能够正确的返回 `null` 呢？

这个逻辑可以覆盖这种情况的，相当于 `c1` 节点是 `null` 空指针嘛，可以正确返回 `null`。

按照这个思路，可以写出如下代码：

```
ListNode getIntersectionNode(ListNode headA, ListNode headB) {  
    // p1 指向 A 链表头结点, p2 指向 B 链表头结点  
    ListNode p1 = headA, p2 = headB;  
    while (p1 != p2) {  
        // p1 走一步，如果走到 A 链表末尾，转到 B 链表  
        if (p1 == null) p1 = headB;  
        else p1 = p1.next;  
        // p2 走一步，如果走到 B 链表末尾，转到 A 链表  
        if (p2 == null) p2 = headA;  
        else p2 = p2.next;  
    }  
    return p1;  
}
```

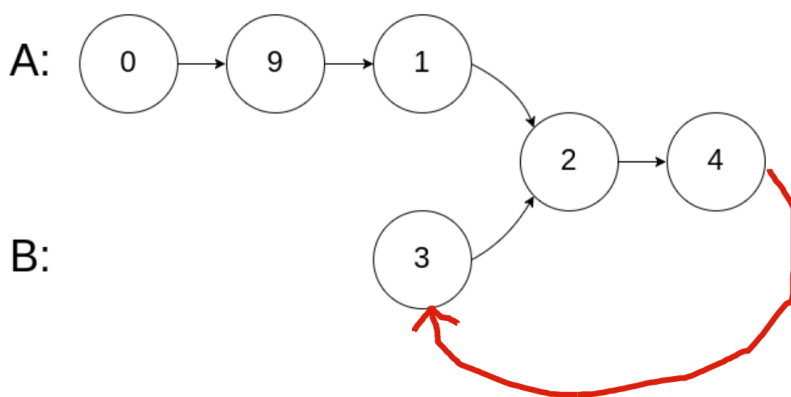
这样，这道题就解决了，空间复杂度为 `O(1)`，时间复杂度为 `O(N)`。

以上就是单链表的所有技巧，希望对你有启发。

## 2022/1/24 更新：

评论区有不少优秀读者对最后一题「寻找两条链表的交点」提出了一些其他思路，也补充到这里。

首先有读者提到，如果把两条链表首尾相连，那么「寻找两条链表的交点」的问题转换成了前面讲的「寻找环起点」的问题：



说实话我没有想到这种思路，不得不说这是一个很巧妙的转换！不过需要注意的是，这道题说不让你改变原始链表的结构，所以你把题目输入的链表转化成环形链表求解之后记得还要改回来，否则无法通过。

另外，还有读者提到，既然「寻找两条链表的交点」的核心在于让 `p1` 和 `p2` 两个指针能够同时到达相交节点 `c1`，那么可以通过预先计算两条链表的长度来做到这一点，具体代码如下：

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {  
    int lenA = 0, lenB = 0;  
    // 计算两条链表的长度  
    for (ListNode p1 = headA; p1 != null; p1 = p1.next) {  
        lenA++;  
    }  
    for (ListNode p2 = headB; p2 != null; p2 = p2.next) {  
        lenB++;  
    }
```

```

    }
    // 让 p1 和 p2 到达尾部的距离相同
    ListNode p1 = headA, p2 = headB;
    if (lenA > lenB) {
        for (int i = 0; i < lenA - lenB; i++) {
            p1 = p1.next;
        }
    } else {
        for (int i = 0; i < lenB - lenA; i++) {
            p2 = p2.next;
        }
    }
    // 看两个指针是否会相同, p1 == p2 时有两种情况:
    // 1、要么是两条链表不相交, 他俩同时走到尾部空指针
    // 2、要么是两条链表相交, 他俩走到两条链表的相交点
    while (p1 != p2) {
        p1 = p1.next;
        p2 = p2.next;
    }
    return p1;
}

```

虽然代码多一些，但是时间复杂度是还是  $O(N)$ ，而且会更容易理解一些。

总之，我的解法代码并不一定就是最优或者最正确的，鼓励大家在评论区多多提出自己的疑问和思考，我也很高兴和大家探讨更多的解题思路~

到这里，链表相关的双指针技巧就全部讲完了，这些技巧的更多扩展延伸见 [更多双指针经典高频题](#)。

---

## ► 引用本文的题目

---

## ► 引用本文的文章

---

-----

《labuladong 的算法小抄》已经出版，关注公众号查看详情；后台回复关键词「进群」可加入算法群；回复「PDF」可获取精华文章 PDF：