# 题目

编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

**示例 1：**



输入：intersectVal=8, listA=[4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3

输出：Reference of the node with value = 8

输入解释：相交节点的值为8（注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表A为 [4,1,8,4,5]，链表B为[5,0,1,8,4,5]。在 A 中，相交节点前有2个节点；在B中，相交节点前有3个节点。

**示例 2：**



输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

输出：Reference of the node with value = 2

输入解释：相交节点的值为2（注意，如果两个链表相交则不能为0）。从各自的表头开始算起，链表A为[0,9,1,2,4]，链表B为[3,2,4]。在A中，相交节点前有3个节点；在B中，相交节点前有1个节点。

**示例 3：**



输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

输出：null

输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。

解释：这两个链表不相交，因此返回 null。

**注意：**

如果两个链表没有交点，返回 null.

在返回结果后，两个链表仍须保持原有的结构。

可假定整个链表结构中没有循环。

程序尽量满足O(n)时间复杂度，且仅用 O(1) 内存。

**类似题目：**

面试题52

LeetCode 141

# 分析

## 方法一：暴力法

逐个考虑list1中的节点，并把list2里的节点轮流与该节点比对。如果相同，那么说明二者汇聚于此。若找不到相同的节点，则继续考虑list1中的下一个节点。

时间复杂度：O(mn)。

空间复杂度：O(1)。

## 方法二：哈希表

算法：

1. 选出节点数较少的那个链表（如果不知道二者的长度，那就随机选择一个），将其中的各个节点指针都纳入哈希表。
2. 遍历另一个链表。针对它的每个节点指针都去检查哈希表里面，是不是已经有了相同的指针。
3. 如果有，那就说明二者肯定汇聚于该节点所指的那个节点。

时间复杂度：创建哈希表所耗的时间+搜索第二份链表所耗的时间：O(m)+O(n)。

空间复杂度：O(m)或O(n)。

### unordered\_set

**思路：**

遍历链表A并将每个结点的地址/引用存储在哈希表中。然后检查链表B中的每一个结点bi是否在哈希表中。若在，则bi为相交结点。

说明：唯一的相交点，可以考虑使用数据结构set。  
 **代码：**

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

std::unordered\_set<ListNode\*> set;

ListNode\* cur\_a = headA;

while (cur\_a)

{

set.insert(cur\_a);

cur\_a = cur\_a->next;

}

ListNode\* cur\_b = headB;

while (cur\_b)

{

if(set.find(cur\_b) != set.end()) //找到了

{

return cur\_b;

}

cur\_b = cur\_b->next;

}

return nullptr;

}

};

**复杂度分析：**

时间复杂度：O(m+n)。

空间复杂度：O(m)或O(n)。

### unordered\_mp

也可以采用unordered\_mp实现：

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode(int x) : val(x), next(NULL) {}

 \* };

 \*/

class Solution {

public:

    ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

        unordered\_map<ListNode\*,int> mp;

        while(headA)

        {

            mp[headA]++;

            headA = headA->next;

        }

        while(headB)

        {

            mp[headB]++;

            if(mp[headB]>1)

                return headB;

            headB = headB->next;

        }

        return nullptr;

    }

};

## 方法三：双指针/快慢指针

### 解法一

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

private:

int getListLength(ListNode \*node)

{

int len = 0;

while(node)

{

len++;

node = node->next;

}

return len;

}

//两链表尾节点对齐

ListNode \*forwardLongList(int long\_len,int short\_len,ListNode \*head)

{

int len = long\_len - short\_len;

while(head && len)

{

head = head->next;

len--;

}

return head;

}

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

int listALen = getListLength(headA);

int listBLen = getListLength(headB);

if(listALen > listBLen)

{

headA = forwardLongList(listALen,listBLen,headA);

}

else

{

headB = forwardLongList(listBLen,listALen,headB);

}

while(headA && headB)

{

if(headA == headB)

return headA;

headA = headA->next;

headB = headB->next;

}

return NULL;

}

};

或：

1. 确定两个链表的长度，将其分别记为L1和L2。这样做的时间复杂度为O(n)+O(m)=O(max(m,n))。
2. 求出链表长度的差值d。这样做的复杂度为O(1)。
3. 对于较长的那份链表来说，跳过开头的d个节点，这样做的时间复杂度为O(d)。
4. 分别比较两份链表的对应节点，如果不相同，那就继续比较二者的下一个节点。这样做的时间复杂度为O(min(m,n))。

综上所述，总的时间复杂度为O(max(m,n))。

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

int L1=0,L2=0,diff=0;

ListNode \*head1 = headA;

ListNode \*head2 = headB;

while(head1)

{

L1++;

head1 = head1->next;

}

while(head2)

{

L2++;

head2 = head2->next;

}

if(L1<L2)

{

head1 = headB; //head1指向较长的链表头指针

head2 = headA;

diff = L2-L1;

}

else

{

head1 = headA;

head2 = headB;

diff = L1-L2;

}

while(diff--)

head1 = head1->next; //较长的链表跳过前面的k个节点

while(head1 && head2)

{

if(head1 == head2)

return head1;

head1 = head1->next;

head2 = head2->next;

}

return nullptr;

}

};

### 解法二

思路：

使用双指针的方法，可以将空间复杂度降至 O(1)。

只有当链表headA和headB都不为空时，两个链表才可能相交。因此首先判断链表headA和headB是否为空，如果其中至少有一个链表为空，则两个链表一定不相交，返回null。

当链表 headA和headB都不为空时，创建两个指针pA和pB，初始时分别指向两个链表的头节点headA和headB，然后将两个指针依次遍历两个链表的每个节点。具体做法如下：

每步操作需要同时更新指针pA和pB。

如果指针pA不为空，则将指针pA移到下一个节点；如果指针pB不为空，则将指针pB移到下一个节点。

如果指针pA为空，则将指针pA移到链表headB的头节点；如果指针pB为空，则将指针pB移到链表headA 的头节点。

当指针pA 和pB指向同一个节点或者都为空时，返回它们指向的节点或者null。

代码：

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

if (headA == nullptr || headB == nullptr) {

return nullptr;

}

ListNode \*pA = headA, \*pB = headB;

while (pA != pB) {

pA = pA == nullptr ? headB : pA->next;

pB = pB == nullptr ? headA : pB->next;

}

return pA;

}

};

另一种写法：

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

if(nullptr==headA || nullptr==headB)

return nullptr;

ListNode \*pA = headA;

ListNode \*pB = headB;

while(pA != pB)

{

if(nullptr == pA)

{

pA = headB;

}

else

{

pA = pA->next;

}

if(nullptr == pB)

{

pB = headA;

}

else

{

pB = pB->next;

}

}

return pA;

}

};

**复杂度分析：**

时间复杂度：O(m+n)，其中m和n是分别是链表headA和headB的长度。两个指针同时遍历两个链表，每个指针遍历两个链表各一次。

空间复杂度：O(1)。