题目地址(142. 环形链表 II)

https://leetcode-cn.com/problems/linked-list-cycle-ii/

标签

- 双指针
- 链表

难度

- 中等

入选理由

1. 和昨天题目有点类似，结合起来练习效果比较好
2. 同样也是讲义中的题，考察频率同样很高

题目描述

```
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

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

说明：不允许修改给定的链表。

进阶：

你是否可以使用 O(1) 空间解决此题？
```

哈希法

思路

1. 遍历整个链表,同时将每个节点都插入哈希表。由于题目没有限定每个节点的值均不同，因此我们必须将节点的引用作为哈希表的键。

2. 如果当前节点在哈希表中不存在,继续遍历。

3. 如果存在,那么当前节点就是环的入口节点。

这种做法的正确性不言而喻。这是因为如果没有环，不可能遍历到这个节点之前就已经存在于哈希表。并且**第一次遍历两次的节点一定是环的位置**。

伪代码:

```
data = new Set() // 声明哈希表
while head不为空{
  if 当前节点在哈希表中存在{
    return head // 当前节点就是环的入口节点
  } else {
    将当前节点插入哈希表
  }
  head指针后移
}
return null // 环不存在
```

代码

代码支持：JS，Java，CPP

JS Code:

```
let data = new Set();
while (head) {
  if (data.has(head)) {
    return head;
  } else {
    data.add(head);
  }
  head = head.next;
}
return null;
```

Java Code:

```
public class Solution {
    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;
    }
}
```

C++ Code:

```
ListNode *detectCycle(ListNode *head) {
    set<ListNode*> seen;
    ListNode *cur = head;
    while (cur != NULL) {
        if (seen.find(cur) != seen.end()) return cur;
        seen.insert(cur);
        cur = cur->next;
    }
    return NULL;
}
```

复杂度分析

令 n 为链表中总的节点数。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

快慢指针法

思路

上面的空间复杂度是 $O(n)$，题目的进阶是使用 $O(1)$ 的空间来解决。该如何做呢?

我们可以使用双指针的技巧来完成。关于双指针，后面的专题也会详细地进行介绍。

具体算法：

1. 定义一个 fast 指针,每次**前进两步**,一个 slow 指针,每次**前进一步**

2. 当两个指针相遇时

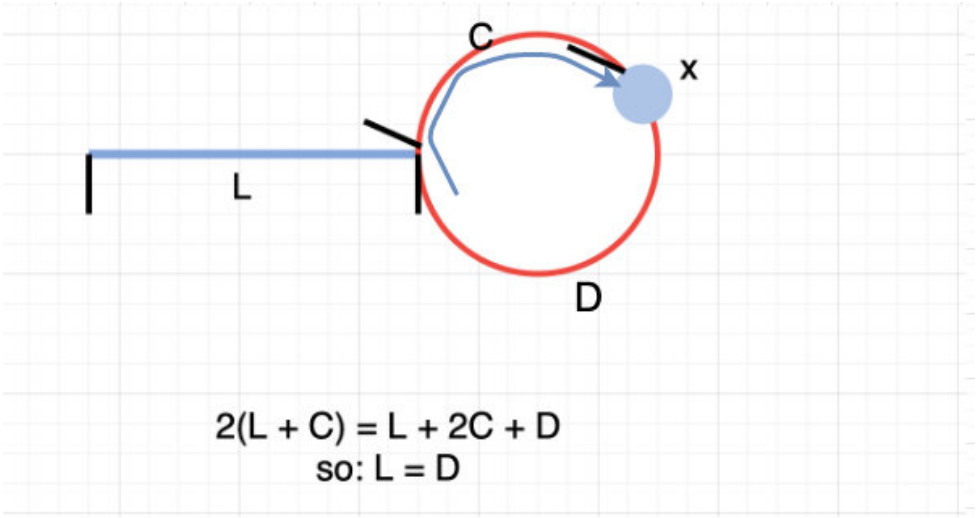
1. 将 fast 指针**重定位**到链表头部,同时 fast 指针每次只**前进一步**

2. slow 指针继续前进,每次**前进一步**

3. 当两个指针再次相遇时,当前节点就是环的入口

下面我们对此方法的正确性进行简单证明:

- x 表示第一次相遇点
- L 是起点到环的入口点的距离
- C 是环的入口点到第一次相遇点的距离
- D 是环的周长减去 C



L + C 是慢指针走的陆离，而快指针走的距离是慢指针的两倍，也就是 2(L + C) , 而快指针走的距离也可以用 L + C + n(C + D) 表示，其中 n 是大于等于 1 的整数，二者结合可以得出 L = (n-1) * (C + D) + D。

因此我们可以在两者第一次相遇后将快指针放回开头，这样二者再次相遇的点一点是环的入口点，此时慢指针**又**走的距离为 D + (n-1) * (C + D)，也就是说慢指针走了 D 加上绕环的 n - 1 圈的距离。

That's all!

伪代码:

```
fast = head
slow = head //快慢指针都指向头部
do {
  快指针向后两步
  慢指针向后一步
} while 快慢指针不相等时
if 指针都为空时{
  return null // 没有环
}
while 快慢指针不相等时{
  快指针向后一步
  慢指针向后一步
}
return fast
```

代码

代码支持：JS，Java，Python3, CPP

JS Code：

```
if (head == null || head.next == null) return null;
let fast = (slow = head);
do {
  if (fast != null && fast.next != null) {
    fast = fast.next.next;
  } else {
    fast = null;
  }
  slow = slow.next;
} while (fast != slow);
if (fast == null) return null;
fast = head;
while (fast != slow) {
  fast = fast.next;
  slow = slow.next;
}
return fast;
```

Java Code:

```
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;
    }
}
```

Python3 Code:

```
class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        slow = fast = head
        x = None

        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next
            if fast == slow:
                x = fast
                break
        if not x:
            return None
        slow = head
        while slow != x:
            slow = slow.next
            x = x.next
        return slow
```

C++ Code:

```
ListNode *detectCycle(ListNode *head) {
    ListNode *slow = head;
    ListNode *fast = head;

    while (fast && fast->next) {
        slow = slow->next;
```

```
        fast = fast->next->next;
        if (slow == fast) {
            fast = head;
            while (slow != fast) {
                slow = slow->next;
                fast = fast->next;
            }
            return slow;
        }
    }
    return NULL;
}
```

复杂度分析

令 n 为链表总的节点数。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

上一页 下一页