摘要

本文适用于初学者。它涉及了以下几个概念：链表，哈希表和双指针。

解决方案

方法一：哈希表

**思路**

我们可以通过检查一个结点此前是否被访问过来判断链表是否为环形链表。常用的方法是使用哈希表。

**算法**

我们遍历所有结点并在哈希表中存储每个结点的引用（或内存地址）。如果当前结点为空结点 null（即已检测到链表尾部的下一个结点），那么我们已经遍历完整个链表，并且该链表不是环形链表。如果当前结点的引用已经存在于哈希表中，那么返回 true（即该链表为环形链表）。

public boolean hasCycle(ListNode head) {

Set<ListNode> nodesSeen = new HashSet<>();

while (head != null) {

if (nodesSeen.contains(head)) {

return true;

} else {

nodesSeen.add(head);

}

head = head.next;

}

return false;

}

**复杂度分析**

* 时间复杂度：*O*(*n*)， 对于含有 *n* 个元素的链表，我们访问每个元素最多一次。添加一个结点到哈希表中只需要花费 *O*(1) 的时间。
* 空间复杂度：*O*(*n*)， 空间取决于添加到哈希表中的元素数目，最多可以添加 *n* 个元素。

方法二：双指针

**思路**

想象一下，两名运动员以不同的速度在环形赛道上跑步会发生什么？

**算法**

通过使用具有 **不同速度** 的快、慢两个指针遍历链表，空间复杂度可以被降低至 *O*(1)。慢指针每次移动一步，而快指针每次移动两步。

如果列表中不存在环，最终快指针将会最先到达尾部，此时我们可以返回 false。

现在考虑一个环形链表，把慢指针和快指针想象成两个在环形赛道上跑步的运动员（分别称之为慢跑者与快跑者）。而快跑者最终一定会追上慢跑者。这是为什么呢？考虑下面这种情况（记作情况 A） - 假如快跑者只落后慢跑者一步，在下一次迭代中，它们就会分别跑了一步或两步并相遇。

其他情况又会怎样呢？例如，我们没有考虑快跑者在慢跑者之后两步或三步的情况。但其实不难想到，因为在下一次或者下下次迭代后，又会变成上面提到的情况 A。

public boolean hasCycle(ListNode head) {

if (head == null || head.next == null) {

return false;

}

ListNode slow = head;

ListNode fast = head.next;

while (slow != fast) {

if (fast == null || fast.next == null) {

return false;

}

slow = slow.next;

fast = fast.next.next;

}

return true;

}

**复杂度分析**

* 时间复杂度：*O*(*n*)， 让我们将 *n* 设为链表中结点的总数。为了分析时间复杂度，我们分别考虑下面两种情况。
  + ***链表中不存在环：***  
    快指针将会首先到达尾部，其时间取决于列表的长度，也就是 *O*(*n*)。
  + ***链表中存在环：***  
    我们将慢指针的移动过程划分为两个阶段：非环部分与环形部分：
    1. 慢指针在走完非环部分阶段后将进入环形部分：此时，快指针已经进入环中 迭代次数=非环部分长度=*N*
    2. 两个指针都在环形区域中：考虑两个在环形赛道上的运动员 - 快跑者每次移动两步而慢跑者每次只移动一步。其速度的差值为1，因此需要经过 \dfrac{\text{二者之间距离}}{\text{速度差值}}速度差值二者之间距离​ 次循环后，快跑者可以追上慢跑者。这个距离几乎就是 "\text{环形部分长度 K}环形部分长度 K" 且速度差值为 1，我们得出这样的结论 \text{迭代次数} = \text{近似于}迭代次数=近似于 "\text{环形部分长度 K}环形部分长度 K".

因此，在最糟糕的情形下，时间复杂度为 *O*(*N*+*K*)，也就是 *O*(*n*)。

* 空间复杂度：*O*(1)， 我们只使用了慢指针和快指针两个结点，所以空间复杂度为 *O*(1)。