

双指针算法基本原理和实践 - huansky - 博客园

 [cnblogs.com/huansky/p/13508533.html](https://www.cnblogs.com/huansky/p/13508533.html)

双指针算法基本原理和实践

什么是双指针

双指针，指的是在遍历对象的过程中，不是普通的使用单个指针进行访问，而是使用两个相同方向（*快慢指针*）或者相反方向（*对撞指针*）的指针进行扫描，从而达到相应的目的。

换言之，双指针法充分使用了数组有序这一特征，从而在某些情况下能够简化一些运算。

在 **LeetCode** 题库中，关于双指针的问题还是挺多的。双指针

截图来之 LeetCode 中文官网

🔖 标签分类



对撞指针

对撞指针是指在数组中，将指向最左侧的索引定义为 **左指针(left)**，最右侧的定义为 **右指针(right)**，然后从两头向中间进行数组遍历。

对撞数组适用于连续数组和字符串，也就是说当你遇到题目给定连续数组和字符床时，应该第一时间想到用对撞指针解题。

伪代码大致如下：



```
public void find (int[] list) {  
    var left = 0;  
    var right = list.length - 1;  
  
    //遍历数组  
    while (left <= right) {  
        left++;  
        // 一些条件判断 和处理  
        ... ..  
        right--;  
    }  
}
```



算法实例

344. 反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 `char[]` 的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 $O(1)$ 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例 1:

输入: ["h","e","l","l","o"]
输出: ["o","l","l","e","h"]

示例 2:

输入: ["H","a","n","n","a","h"]
输出: ["h","a","n","n","a","H"]

解答

可以套用前面的伪代码:



```
class Solution {
    public void reverseString(char[] s) {
        if (s.length == 0 || s.length == 1) return ;
        int left = 0;
        int right = s.length-1;
        while (left < right) {
            char temp = s[left];
            s[left++] = s[right];
            s[right--] = temp;
        }
        return ;
    }
}
```



209. 长度最小的子数组

给定一个含有 n 个正整数的数组和一个正整数 s ，找出该数组中满足其和 $\geq s$ 的长度最小的连续子数组，并返回其长度。如果不存在符合条件的子数组，返回 0。

示例：

输入：s = 7, nums = [2,3,1,2,4,3]

输出：2

解释：子数组 [4,3] 是该条件下的长度最小的子数组。

解答



```
class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        int right = 0;
        int left = 0;
        int sum = 0;
        int len = Integer.MAX_VALUE;
        while(right < nums.length) {
            sum += nums[right];
            while (sum >= s) {
                len = Math.min(right - left + 1, len);
                sum -= nums[left];
                left++;
            }
            right++;
        }
        if (len == Integer.MAX_VALUE) return 0;
        return len;
    }
}
```



虽然这道题目也是用的双指针，但是实际上采用滑动窗口的算法思想，具体可以看文章：[滑动窗口算法基本原理与实践](#)。

快慢指针

快慢指针也是双指针，但是两个指针从同一侧开始遍历数组，将这两个指针分别定义为 **快指针 (fast)** 和 **慢指针 (slow)**，两个指针以不同的策略移动，直到两个指针的值相等（或其他特殊条件）为止，如 fast 每次增长两个，slow 每次增长一个。

以LeetCode 141.环形链表为例，判断给定链表中是否存在环，可以定义快慢两个指针，快指针每次增长一个，而慢指针每次增长两个，最后两个指针指向节点的值相等，则说明有环。就好像一个环形跑道上有一快一慢两个运动员赛跑，如果时间足够长，跑地快的运动员一定会赶上慢的运动员。

算法示例

快慢指针一般都初始化指向链表的头结点 head，前进时快指针 fast 在前，慢指针 slow 在后，巧妙解决一些链表中的问题。

1、判定链表中是否含有环

这应该属于链表最基本的操作了，如果读者已经知道这个技巧，可以跳过。

单链表的特点是每个节点只知道下一个节点，所以一个指针的话无法判断链表中是否含有环的。

如果链表中不包含环，那么这个指针最终会遇到空指针 null 表示链表到头了，这还好说，可以判断该链表不含环。

```
boolean hasCycle(ListNode head) {  
    while (head != null)  
        head = head.next;  
    return false;  
}
```

但是如果链表中含有环，那么这个指针就会陷入死循环，因为环形数组中没有 null 指针作为尾部节点。

经典解法就是用两个指针，一个每次前进两步，一个每次前进一步。如果不含有环，跑得快的那个指针最终会遇到 null，说明链表不含环；如果含有环，快指针最终会和慢指针相遇，说明链表含有环。

就好像一个环形跑道上有一快一慢两个运动员赛跑，如果时间足够长，跑地快的运动员一定会赶上慢的运动员。



```
boolean hasCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while(fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
        if (fast == slow)
            return true;
    }
    return false;
}
```



2、已知链表中含有环，返回这个环的起始位置

这个问题其实不困难，有点类似脑筋急转弯，先直接看代码：



```
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;
    }

    slow = head;
    while (slow != fast) {
        fast = fast.next;
        slow = slow.next;
    }
    return slow;
}
```



可以看到，当快慢指针相遇时，让其中任一个指针重新指向头节点，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。

3、寻找链表的中点

类似上面的思路，我们还可以让快指针一次前进两步，慢指针一次前进一步，当快指针到达链表尽头时，慢指针就处于链表的中间位置。



```
ListNode slow, fast;
slow = fast = head;
while (fast != null && fast.next != null) {
    fast = fast.next.next;
    slow = slow.next;
}
// slow 就在中间位置
return slow;
```



当链表的长度是奇数时，slow 恰巧停在中点位置；如果长度是偶数，slow 最终的位置是中间偏右：

寻找链表中点的一个重要作用是对链表进行归并排序。

回想数组的归并排序：求中点索引递归地把数组二分，最后合并两个有序数组。对于链表，合并两个有序链表是很简单的，难点就在于二分。

但是现在你学会了找到链表的中点，就能实现链表的二分了。关于归并排序的具体内容本文就不具体展开了。具体可看文章

4、寻找链表的倒数第 k 个元素

我们的思路还是使用快慢指针，让快指针先走 k 步，然后快慢指针开始同速前进。这样当快指针走到链表末尾 null 时，慢指针所在的位置就是倒数第 k 个链表节点（为了简化，假设 k 不会超过链表长度）：



```
ListNode slow, fast;
slow = fast = head;
while (k-- > 0)
    fast = fast.next;

while (fast != null) {
    slow = slow.next;
    fast = fast.next;
}
return slow;
```



滑动窗口算法

这也许是双指针技巧的最高境界了，如果掌握了此算法，可以解决一大类子字符串匹配的问题，不过「滑动窗口」算法比上述的这些算法稍微复杂些。

具体原理和实践可以详见文章：滑动窗口算法基本原理与实践

参考文章：

<https://www.cnblogs.com/kyoner/p/11087755.html>

<https://zhuanlan.zhihu.com/p/71643340>

树林美丽、幽暗而深邃，但我有诺言尚待实现，还要奔行百里方可沉睡。 -- 罗伯特·弗罗斯特

标签: 数据结构和算法





« 上一篇: 滑动窗口算法基本原理与实践

» 下一篇: PriorityQueue 源码分析

刷新评论刷新页面返回顶部

发表评论

编辑 预览

B    

退出 订阅评论

[Ctrl+Enter快捷键提交]

【推荐】 超50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库

【推荐】 7天蜕变！阿里云专家免费授课，名额有限！

【推荐】 828企业上云节，亿元上云补贴，华为云更懂企业

【推荐】 未知数的距离，毫秒间的传递，声网与你实时互动

【推荐】 了不起的开发者，挡不住的华为，园子里的品牌专区

【推荐】 精品问答: Java 技术 1000 问

Copyright © 2020 huansky

Powered by .NET Core on Kubernetes