题目地址（30. 串联所有单词的子串）

https://leetcode-cn.com/problems/substring-with-concatenation-of-all-words

入选理由

1. 今天和明天都是困难难度，作为压轴。
2. 这道题哈希是如何优化我们的算法的呢?

题目描述

给定一个字符串 s 和一些长度相同的单词 words。找出 s 中恰好可以由 words 中所有单词串联形成的子串的起始位置。

注意子串要与 words 中的单词完全匹配，中间不能有其他字符，但不需要考虑 words 中单词串联的顺序。

示例 1:
输入:
s = "barfoothefoobarman",
words = ["foo","bar"]
输出: [0,9]
解释:
从索引 0 和 9 开始的子串分别是 "barfoo" 和 "foobar" 。
输出的顺序不重要, [9,0] 也是有效答案。
示例 2:

输入:
s = "wordgoodgoodgoodbestword",
words = ["word","good","best","word"]
输出: []

标签

- 字符串
- 双指针
- 哈希表

难度

- 困难

前置知识

- 哈希表
- 双指针

思路

还是从题意暴力入手。大体上会有两个想法：

1. 从 words 入手，words 所有单词排列生成字符串 X, 通过字符串匹配查看 X 在 s 中的出现位置
2. 从 s 串入手，遍历 s 串中所有长度为 (words[0].length * words.length) 的子串 Y，查看 Y 是否可以由 words 数组构造生成

先看第一种思路: 构造 X 的时间开销是 (words.length)! / (words 中单词重复次数相乘), 时间复杂度为 O(m!), m 为 words 长度。阶乘的时间复杂度基本不可能通过。

下面看第二种思路: 仅考虑遍历过程， 遍历 s 串的时间复杂度为 $O(n - m + 1)$, 其中 n 为 s 字符串长度, m 为 words[0].length * words.length，也就是 words 的字符总数。问题关键在于如何判断 s 的子串 Y 是否可以由 words 数组的构成，由于 words 中单词长度固定，我们可以将 Y 拆分成对应 words[0]长度的一个个子串 parts, 只需要判断 words 和 parts 中的单词是否一一匹配即可，这里用两个哈希表表比对出现次数即可。一旦一个对应不上，意味着此种分割方法不正确，继续尝试下一种即可。

代码

```
class Solution {

    public List<Integer> findSubstring(String s, String[] words) {

        List<Integer> res = new ArrayList<>();

        Map<String, Integer> map = new HashMap<>();

        if (words == null || words.length == 0)
            return res;

        for (String word : words)
            map.put(word, map.getOrDefault(word, 0) + 1);

        int sLen = s.length(), wordLen = words[0].length(), count = words.length;

        int match = 0;

        for (int i = 0; i < sLen - wordLen * count + 1; i++) {
```

```
            //得到当前窗口字符串
            String cur = s.substring(i, i + wordLen * count);
            Map<String, Integer> temp = new HashMap<>();
            int j = 0;

            for (; j < cur.length(); j += wordLen) {

                String word = cur.substring(j, j + wordLen);
                // 剪枝
                if (!map.containsKey(word))
                    break;

                temp.put(word, temp.getOrDefault(word, 0) + 1);
                // 剪枝
                if (temp.get(word) > map.get(word))
                    break;
            }

            if (j == cur.length())
                res.add(i);
        }

        return res;
    }
}
```

复杂度分析

令 n 为字符串 S 长度, m 为 words 数组元素个数, k 为单个 word 字串长度。

- 时间复杂度: 本质上我们的算法是将 s 划分为若干了段，这些段的长度为 $m * k$，对于每一段我们最多需要检查 $n - m * k$ 次，因此时间复杂度为 $O(n * m * k)$。
- 空间复杂度: temp 在下一次循环会覆盖上一次的 temp，因此 temp 的空间在任意时刻都不大于 $O(m)$, 因此空间复杂度为 $O(m)$。

上一页 下一页