0010. 正则表达式匹配

ITCharge 大约 5 分钟

- 标签：递归、字符串、动态规划
- 难度：困难

题目链接

- 0010. 正则表达式匹配 - 力扣

题目大意

描述：给定一个字符串 `s` 和一个字符模式串 `p` 。

要求：实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。两个字符串完全匹配才算匹配成功。如果匹配成功，则返回 `True` ，否则返回 `False` 。

- `'.'` 匹配任意单个字符。
- `'*'` 匹配零个或多个前面的那一个元素。

说明：

- $1 \leq s.length \leq 20$。
- $1 \leq p.length \leq 30$。
- `s` 只包含从 `a` ~ `z` 的小写字母。
- `p` 只包含从 `a` ~ `z` 的小写字母，以及字符 `.` 和 `*` 。
- 保证每次出现字符 `*` 时，前面都匹配到有效的字符。

示例：

- 示例 1：

```py
输入：s = "aa", p = "a*"
输出：True
解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。
```

- 示例 2：

```py
输入：s = "aa", p = "a"
输出：False
解释："a" 无法匹配 "aa" 整个字符串。
```

解题思路

思路 1：动态规划

1. 划分阶段

按照两个字符串的结尾位置进行阶段划分。

2. 定义状态

定义状态 `dp[i][j]` 表示为：字符串 `s` 的前 `i` 个字符与字符串 `p` 的前 `j` 个字符是否匹配。

3. 状态转移方程

- 如果 `s[i - 1] == p[j - 1]`，则字符串 `s` 的第 `i` 个字符与字符串 `p` 的第 `j` 个字符是匹配的。此时「字符串 `s` 的前 `i` 个字符与字符串 `p` 的前 `j` 个字符是否匹配」取决于「字符串 `s` 的前 `i - 1` 个字符与字符串 `p` 的前 `j - 1` 个字符是否匹配」。即 `dp[i][j] = dp[i - 1][j - 1]` 。
- 如果 `p[j - 1] == '.'`，则字符串 `s` 的第 `i` 个字符与字符串 `p` 的第 `j` 个字符是匹配的（同上）。此时 `dp[i][j] = dp[i - 1][j - 1]` 。
- 如果 `p[j - 1] == '*'`，则我们可以对字符 `p[j - 2]` 进行 `0` ~ 若干次数的匹配。
  - 如果 `s[i - 1] != p[j - 2]` 并且 `p[j - 2] != '.'`，则说明当前星号匹配不上，只能匹配 `0` 次（即匹配空字符串），则「字符串 `s` 的前 `i` 个字符与字符串 `p` 的前 `j` 个字符是否匹配」取决于「字符串 `s` 的前 `i` 个字符与字符串 `p` 的前 `j - 2` 个字符是否匹配」，即 `dp[i][j] = dp[i][j - 2]` 。
  - 如果 `s[i - 1] == p[j - 2]` 或者 `p[j - 2] == '.'`，则说明当前星号前面的字符 `p[j - 2]` 可以匹配 `s[i - 1]` 。
    - 如果匹配 `0` 个，则「字符串 `s` 的前 `i` 个字符与字符串 `p` 的前 `j` 个字符是否匹配」取决于「字符串 `s` 的前 `i` 个字符与字符串 `p` 的前 `j - 2` 个字符是否匹配」。即 `dp[i][j] = dp[i][j - 2]` 。

- 如果匹配 1 个，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 i 个字符与字符串 p 的前 j - 1 个字符是否匹配」。即 dp[i][j] = dp[i][j - 1] 。
- 如果匹配多个，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 i - 1 个字符与字符串 p 的前 j 个字符是否匹配」。即 dp[i][j] = dp[i - 1][j] 。

4. 初始条件

- 默认状态下，两个空字符串是匹配的，即 dp[0][0] = True 。
- 当字符串 s 为空，字符串 p 右端有 * 时，想要匹配，则如果「空字符串」与「去掉字符串 p 右端的 * 和 * 之前的字符之后的字符串」匹配的话，则空字符串与字符串 p 匹配。也就是说如果 p[j - 1] == '*' ，则 dp[0][j] = dp[0][j - 2] 。

5. 最终结果

根据我们之前定义的状态， dp[i][j] 表示为：字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配。则最终结果为 dp[size_s][size_p] ，其实 size_s 是字符串 s 的长度， size_p 是字符串 p 的长度。

思路 1：动态规划代码

py

```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        size_s, size_p = len(s), len(p)
        dp = [[False for _ in range(size_p + 1)] for _ in range(size_s + 1)]

        dp[0][0] = True
        for j in range(1, size_p + 1):
            if p[j - 1] == '*':
                dp[0][j] = dp[0][j - 2]

        for i in range(1, size_s + 1):
            for j in range(1, size_p + 1):
                if s[i - 1] == p[j - 1] or p[j - 1] == '.':
                    dp[i][j] = dp[i - 1][j - 1]
                elif p[j - 1] == '*':
                    if s[i - 1] != p[j - 2] and p[j - 2] != '.':
```

```
                    dp[i][j] = dp[i][j - 2]
                else:
                    dp[i][j] = dp[i][j - 1] or dp[i][j - 2] or dp[i - 1][j]

        return dp[size_s][size_p]
```

思路 1：复杂度分析

- **时间复杂度**：$O(mn)$，其中 m 是字符串 `s` 的长度，n 是字符串 `p` 的长度。使用了两重循环，外层循环遍历的时间复杂度是 $O(m)$，内层循环遍历的时间复杂度是 $O(n)$，所以总体的时间复杂度为 $O(mn)$。
- **空间复杂度**：$O(mn)$，其中 m 是字符串 `s` 的长度，n 是字符串 `p` 的长度。使用了二维数组保存状态，且第一维的空间复杂度为 $O(m)$，第二位的空间复杂度为 $O(n)$，所以总体的空间复杂度为 $O(mn)$。