0044. 通配符匹配

ITCharge 大约 4 分钟

- 标签：贪心、递归、字符串、动态规划
- 难度：困难

题目链接

- 0044. 通配符匹配 - 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 `'*'` 的通配符匹配。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` 。

- `'?'` 可以匹配任何单个字符。
- `'*'` 可以匹配任意字符串（包括空字符串）。

说明：

- `s` 可能为空，且只包含从 `a` ~ `z` 的小写字母。
- `p` 可能为空，且只包含从 `a` ~ `z` 的小写字母，以及字符 `'?'` 和 `'*'` 。

示例：

- 示例 1：

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

- 示例 2：

```py
输入：s = "aa"  p = "*"
输出：True
```

解释：'*' 可以匹配任意字符串。

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] ，或者 p[j - 1] == '?' ，则表示字符串 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 p 的第 j 个字符可以对应字符串 s 中 0 ~ 若干个字符。则：
  - 如果当前星号没有匹配当前第 i 个字符，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] 。
  - 如果当前星号匹配了当前第 i 个字符，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] 。
  - 这两种情况只需匹配一种，就视为匹配，所以 dp[i][j] = dp[i - 1][j] or dp[i][j - 1] 。

则动态转移方程为：

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & s[i-1] == p[j-1] \backslash \text{or} p[j-1] == '?' \\ dp[i-1][j] or dp[i][j-1] & p[j-1] == '*' \end{cases}$$

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` 开始字符为若干个 `*` 时，两个字符串是匹配的，即 `p[j - 1] == '*'` 时， `dp[0][j] = True` 。

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] != '*':
                break
            dp[0][j] = True

        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] == '*':
                    dp[i][j] = dp[i - 1][j] or dp[i][j - 1]

        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 $O(mn)$。