例题分析一

LeetCode 第 10 题 ,正则表达式匹配 :给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

- '.' 匹配任意单个字符
- '*' 匹配零个或多个前面的那一个元素

注意：所谓匹配，是要涵盖整个字符串 s 的，而不是部分字符串。

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 1

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

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 3

输入:

s = "ab"

p = ".*"

输出: true

解释: ".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。

示例 4

输入:

s = "aab"

p = "c*a*b"

输出: true

解释: 因为 '*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。

示例 5

输入:

s = "mississippi"

p = "mis*is*p*."

输出: false

解释：'p'与'i'无法匹配。

解题思路

不要害怕，这道题只要求实现正则表达式里的两个小功能。

判断 s 和 p 匹配

举例：给定两个字符串 s 和 p，判断 s 和 p 是否匹配。

解法：s 和 p 必须要相等。定义两个指针 i 和 j，分别指向 s 和 p 的第一个字符，然后逐个去比较，一旦发现不相等，就立即知道 s 和 p 不匹配。

此时，假设 s 字符串的长度为 m，p 字符串的长度为 n，s 和 p 匹配的条件就是 s 和 p 从头到尾一直匹配，即 i == m 同时 j == n 是 s 和 p 匹配的唯一条件。

点匹配符 '.'

'.' 匹配任意单个字符，首先要明确的是，它是一一对应关系，和 '*' 匹配符不一样。举例说明如下。

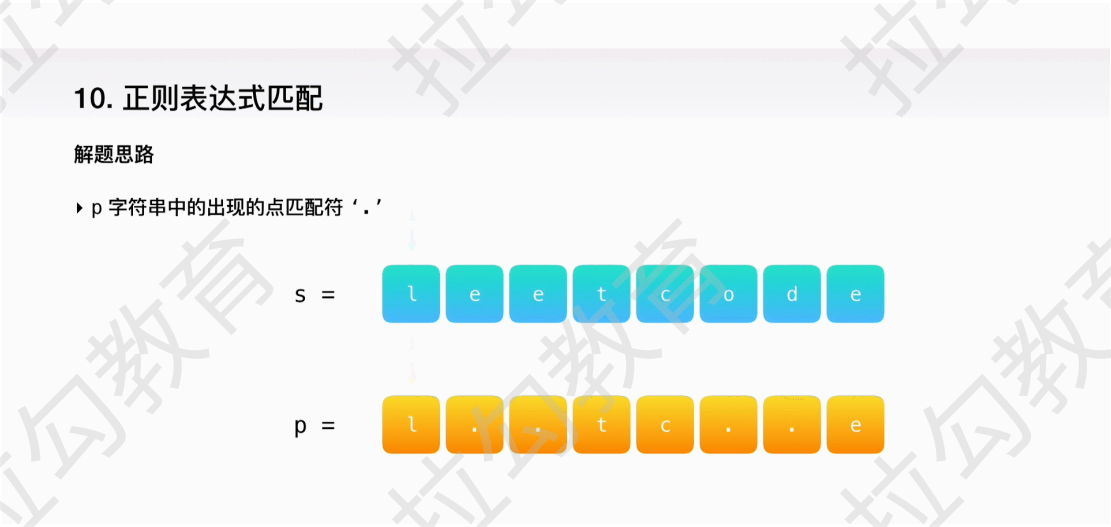
输入:

s = "leetcode"

p = "l..tc..e"

输出: true

因为 '.' 可以匹配任何字符，即，一旦遇上了 '.' 匹配符，可以让 i 指针和 j 指针同时跳到下一个位置。

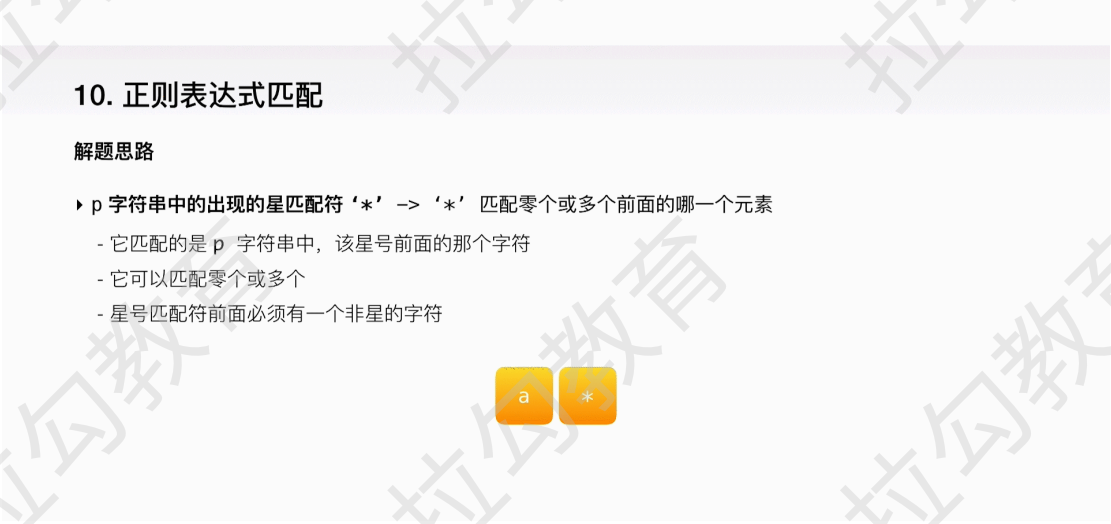


星匹配符 '*'

'*' 匹配符较难，先要理解这个星匹配符的定义。题目 “ '*' 匹配零个或多个前面的那一个元素” 中包含三个重要的信息：

1. 它匹配的是 p 字符串中，该 '*' 前面的那个字符。
2. 它可以匹配零个或多个。
3. '*' 匹配符前面必须有一个非星的字符。

因此，在分析 '*' 匹配符的时候，一定要把 '*' 以及它前面的一个字符看作为一个整体， '*' 不能单独作为一个个体来看（例如点匹配符）。例如，p 字符串是 a *，则把（a*）当作一个整体来看。



对 p 字符串说明如下。

- p 可以表示空字符串，因为 '*' 可以匹配 0 个前面的字符，即当有 0 个 a 的时候，为空字符串。
- a* 还能匹配一个 a，两个 a，三个 a，一直到无穷个 a。

- 当 p 等于 '.*' 的时候，可以表示一个空字符串，也可以表示一个点，两个点，三个点，一直到无穷个点。即它可以表示任何长度的一段字符串，包括空串。

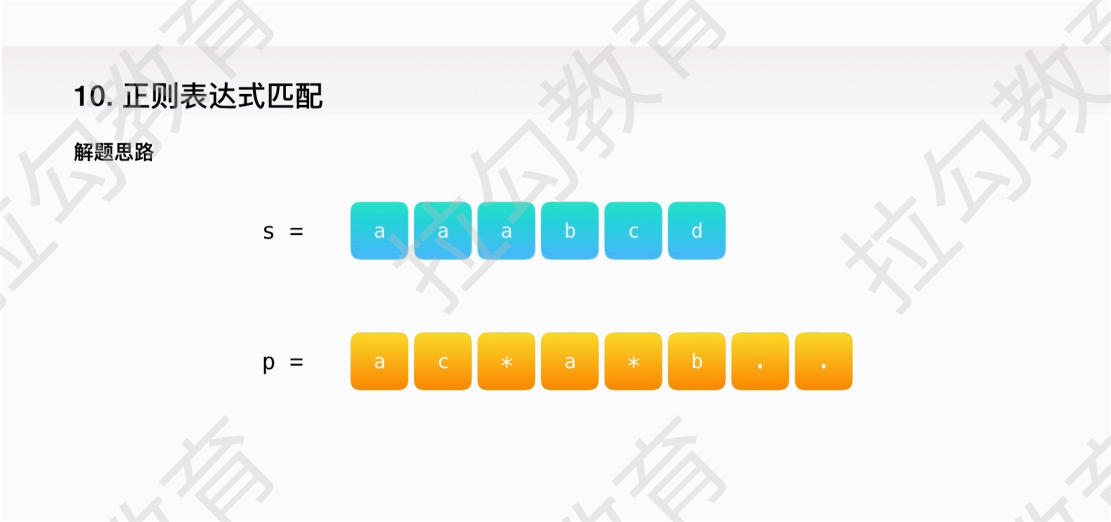
举例说明

输入：

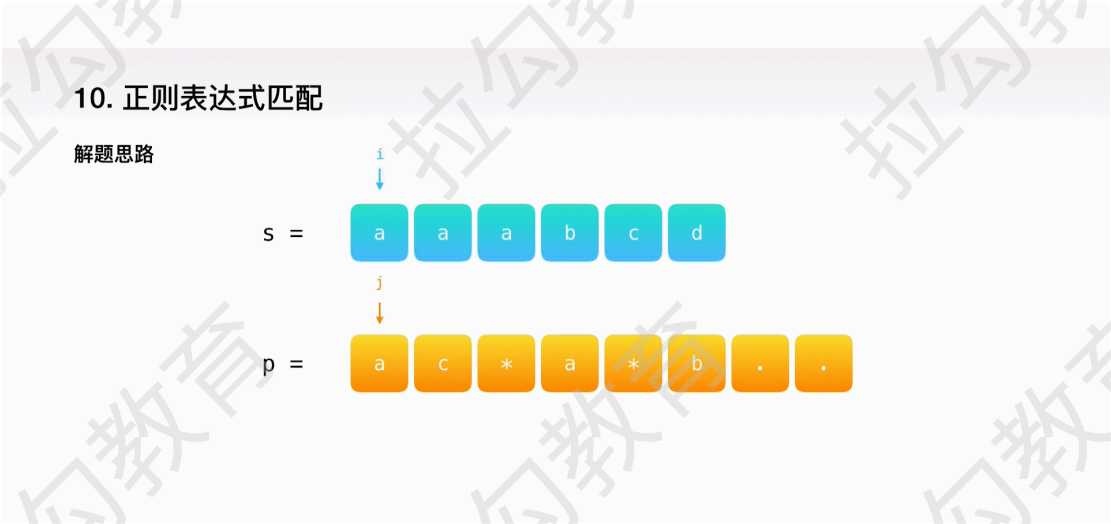
s = "aaabcd"

p = "ac*a*b.."

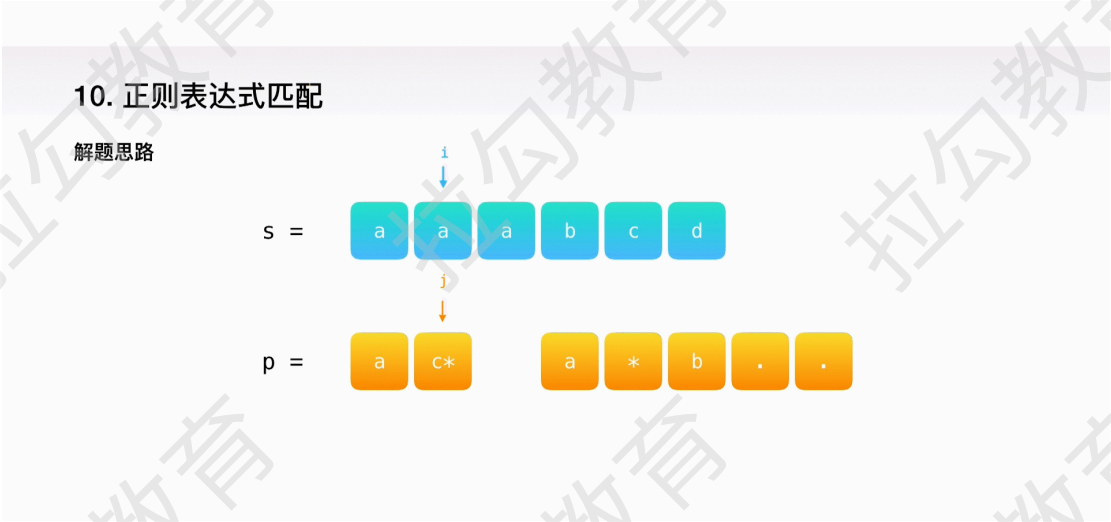
1. 用两个指针 i 和 j 分别指向 s 和 p 的开头。



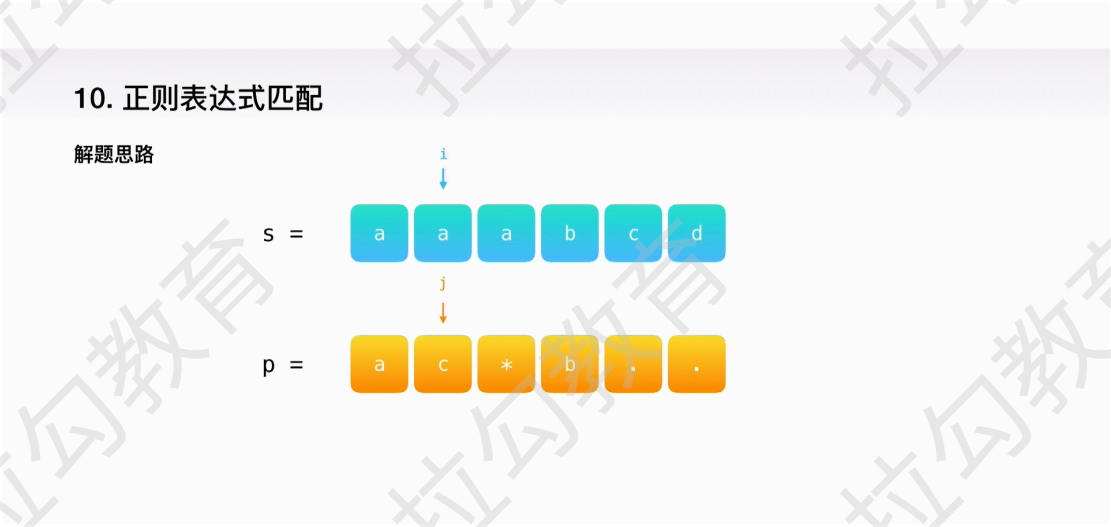
2. 在 p 字符串里，a 的下一个字符是 c，不是 '*'，比较 s[i] 和 p[j]。因为它们都是字符 a，所以这个位置匹配正确，i 和 j 同时指向下一个。此时 j 的下一个字符是 '*'，要将 c* 当作一个整体去看待。



3. 将 c* 看成是空字符，p 如下所示。



4. 若匹配中不一致即 c* 不能当作空字符串，则当作一个 c 字符，此时 p 如下。



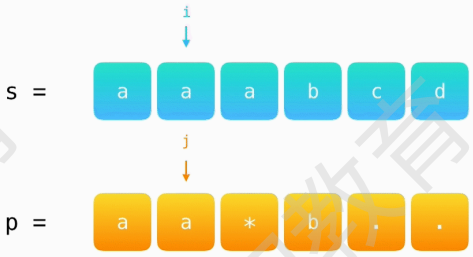
5. 若不行，则看作两个 c。

以此类推，应用了回溯的思想。

对于将 c* 作为空字符串的情况。每一次，都要看看当前 j 指向的字符的下一个是不是 '*'。如果是 '*'，就要作为整体考虑。很明显，a 的下一个字符是 '*'。

10. 正则表达式匹配

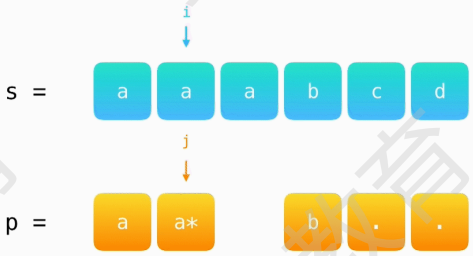
解题思路



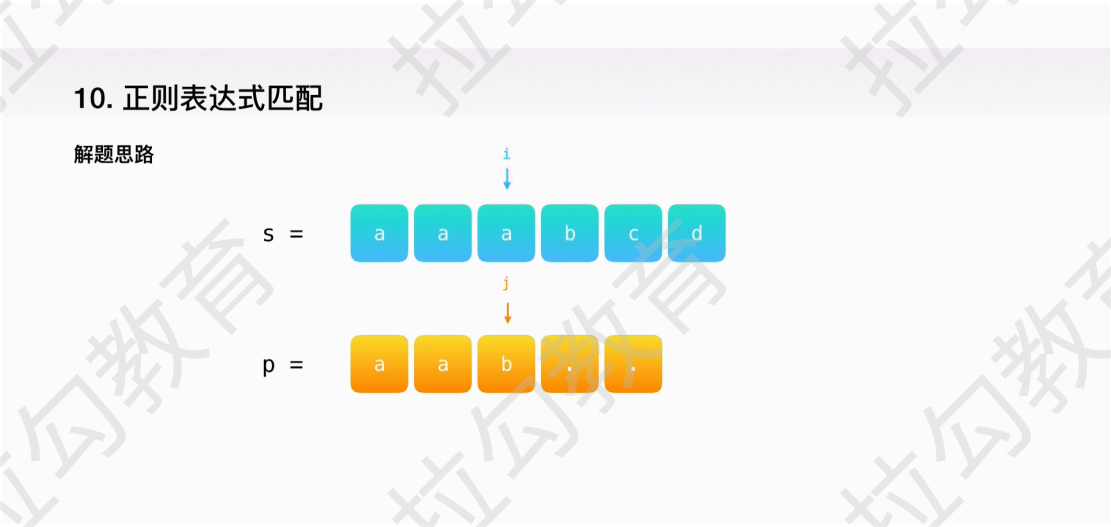
同样，先将 a* 作为空字符串看待。此时，a != b，两个字符串不匹配，因此回溯.现在将 a* 看成是一个 a，此时 a = a，两个位置的字符匹配。

10. 正则表达式匹配

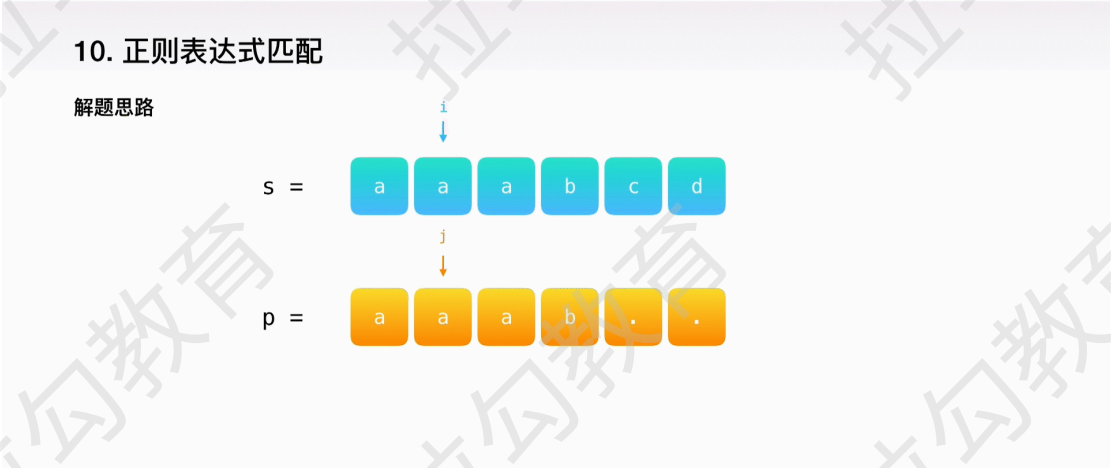
解题思路



j 的下一个字符不是 '*'，而是点号，比较 s[i] 和 p[j]，发现 a != b。于是再次回溯，将 a* 看成是两个 a，回到刚才的位置。



最后遇到了两个点号，由于点号可以匹配任何字符，因此可以直接忽略。i 和 j 同时往前一步，再次遇到了点号。i 和 j 继续往前一步。



此时，i 和 j 都已经同时结束了各自的遍历，表明 s 和 p 是匹配的。

提示：重点是把这种回溯的思想掌握好。对于这道题，可以采用递归的写法，也可以采用动态规划的写法。

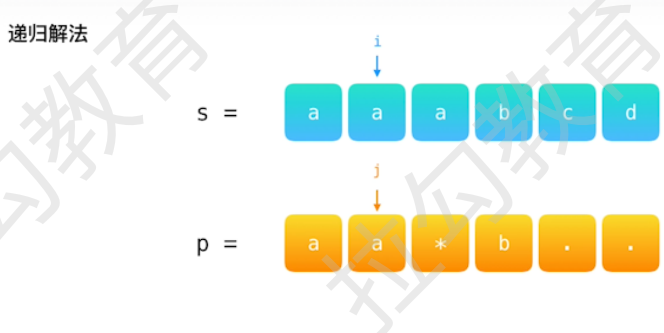
递归法一

一开始，用两个指针 i 和 j 分别指向字符串 s 和 p 的第一个字符，当我们发现它们指向的字符相同时，我们同时往前一步移动指针 i 和 j。

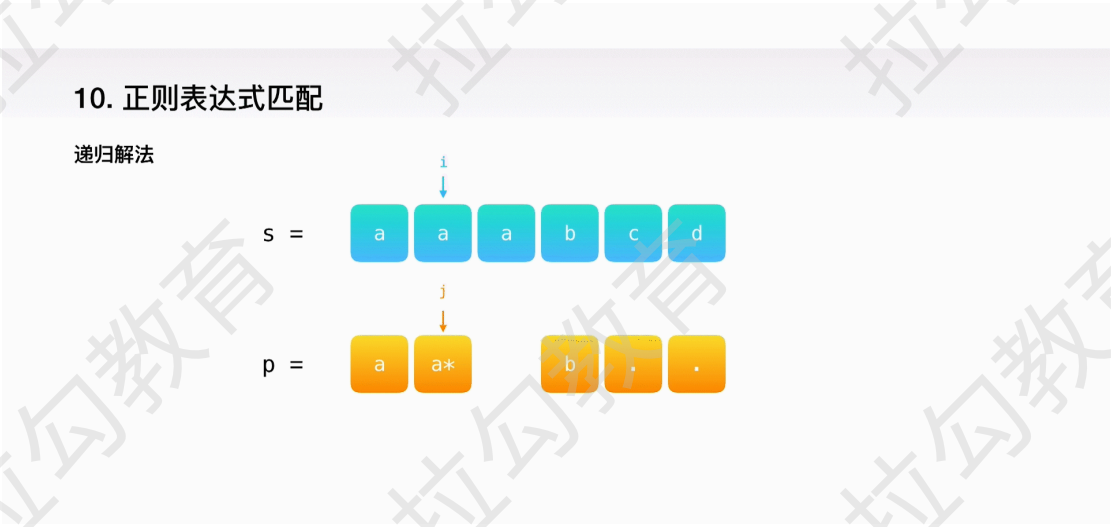
接下来重复进行相同的操作，即，若将函数定义为 isMatch(String s, int i, String p, int j) 的话，通过传递 i 和 j，就能实现重复利用匹配逻辑的效果。

当遇到点匹配符的时候，方法类似。

来看看当遇到星匹配符的情况，举例说明如下。要不断地用 a* 去表示一个空字符串，一个 a，两个 a，一直到多个 a……

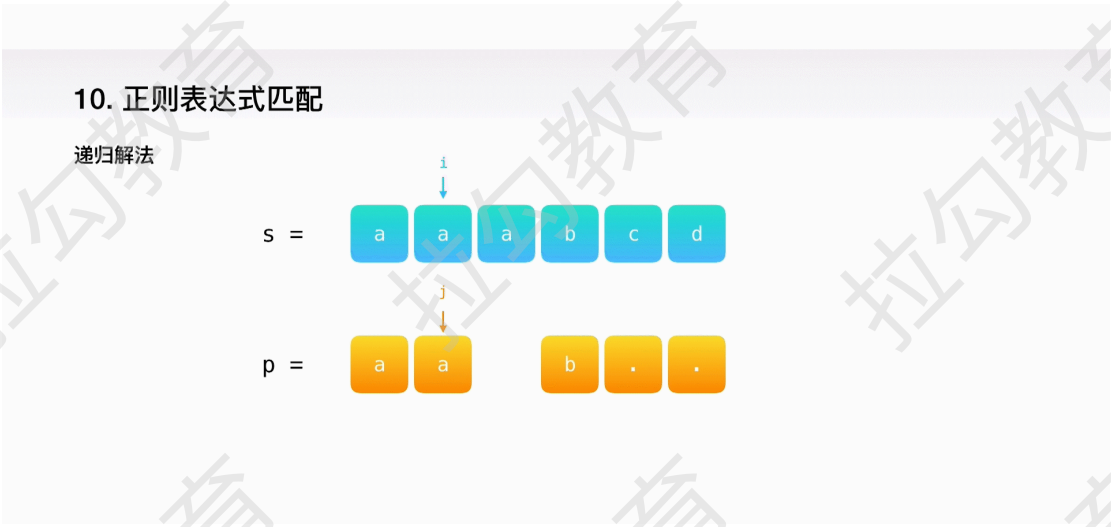


当 a* 表示空字符串的时候，i 和 j 应该如何调整呢？此时 i 保持不变，j+2，递归调用函数的时候，变成 isMatch(s, i, p, j + 2)。

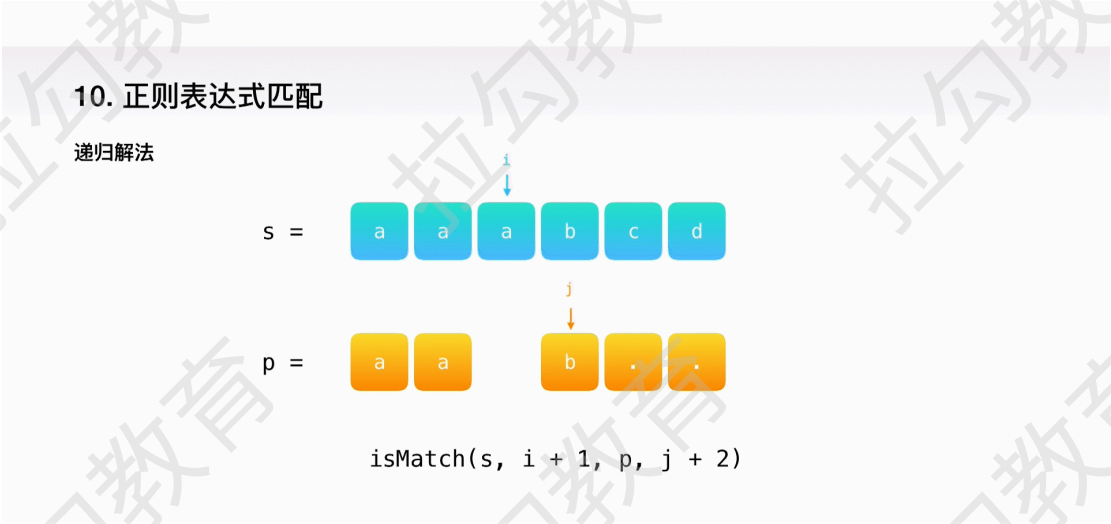


此时，指向的字符和 j 指向的字符不匹配，于是回溯，回到原来的位置。11:57

用 a* 去表示一个 a，i 指向的字符与 a 匹配，那么 i+1。指针 j，已经完成了用 a* 去表示一个 a 的任务，接下来要指向 b，调用的时候应该是 isMatch(s, i + 1, p, j + 2)。



i 指向的字符和 j 指向的字符不匹配，又进行回溯，但是不用回到最开始的位置。已知用 a* 去表示空字符串不行，表示一个 a 也不行，那么尝试两个 a 字符，于是，i 再往前一步，用 a* 去匹配两个 a，i 就到了 b 的位置上，调用的时候就是 isMatch(s, i + 2, p, j + 2)。



不断地这样操作，一旦遇到了 '*' 组合，就不断地尝试，直到最后满足匹配；或者尝试过所有的可能还是不行则表示 s 和 p 无法匹配。

代码实现

根据上面的思路，一起来写递归的实现。主体函数如下。

```
boolean isMatch(String s, String p) {
    if (s == null || p == null) {
        return false;
    }

    return isMatch(s, 0, p, 0);
}
```

主体函数非常简单，一开始做简单的判断，只要 s 和 p 有一个为 null，就表示不匹配。

注意：面试的时候，一定要注意对这些基本情况的考量，千万不要认为输入的值都是有效的。

接下来实现递归函数。

```
boolean isMatch(String s, int i, String p, int j) {
    int m = s.length();
    int n = p.length();

    // 看看 pattern 和字符串是否都扫描完毕
        if (j == n) {
        return i == m;
    }

    // next char is not '*': 必须满足当前字符并递归到下一层
        if (j == n - 1 || p.charAt(j + 1) != '*') {
        return (i < m) &&

                        (p.charAt(j) == '.' || s.charAt(i) == p
.charAt(j)) &&
                        isMatch(s, i + 1, p, j + 1);
    }

    // next char is '*', 如果有连续的 s[i]出现并且都等于 p[j]，一直尝试下去
        if (j < n - 1 && p.charAt(j + 1) == '*') {
        while ((i < m) && (p.charAt(j) == '.' || s.charAt(i)
 == p.charAt(j))) {
            if (isMatch(s, i, p, j + 2)) {
                return true;
            }
            i++;
        }
    }
```

```
        //  接着继续下去
        return  isMatch(s,  i,  p,  j  +  2);
}
```

1. 函数接受四个输入参数，s 字符串，p 字符串，i 指针，j 指针。
2. 开始时计算 s 字符串和 p 字符串的长度，分别记为 m 和 n。
3. 当 j 指针遍历完了 p 字符串后，可以跳出递归，而 i 也刚好遍历完，说明 s 和 p 完全匹配。
4. 判断 j 字符的下一个是不是 '*'，不是，则递归地调用 isMatch 函数，i + 1，j + 1。
5. 若是，则不断地将它和 '*' 作为一个整体，分别去表示空字符串，一个字符，两个字符，依此类推。如果其中一种情况能出现 s 和 p 的匹配，就返回 true。
6. while 循环是整个递归算法的核心，前提条件如下。

a. i 指向的字符必须要能和 j 指向的字符匹配，其中 j 指向的可能是点匹配符。

b. 若无法匹配，i++，即用 '*' 组合去匹配更长的一段字符串。

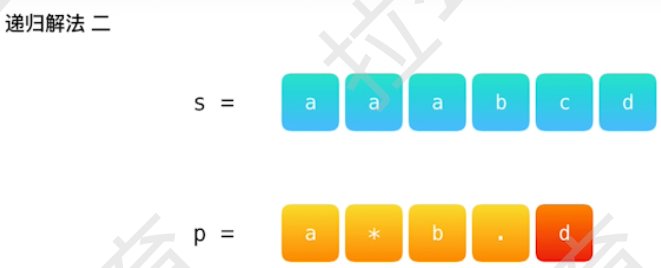
7. 当 i 字符和 j 字符不相同，或者 i 已经遍历完了 s 字符串，同时 j 字符后面跟着一个 '*'的情况，只能用 '*'组合去表示一个空字符串，然后递归下去。

递归法二

上法是从前往后进行递归地调用，现在从后往前地分析这个问题。例如：

```
s = "aaabcd"
```

```
p = "a*b.d"
```



实现过程如下所示。

1. p 字符串的最后一个字符 d 必须要和 s 字符串的最后一个字符相同，才能使 p 有可能与 s 匹配，那么当它们都相同的时候，问题规模也缩小。

2. p 字符串的最后一个字符不是 '*'，而是点号。它可以匹配 s 字符串里的任意一个字符，且它是最后一个，所以对应的就是 s 字符串里的 c，很明显互相匹配，继续缩小问题规模。
3. 同样，b 不是 '*'，比较它与 s 字符串的最后一个字符是否相同，是，则继续缩小问题规模。
4. 遇到 '*'，'*'可以表示一个空字符串，与前一个字符表示空字符串的时候，将问题变成了判断两个字符串是否匹配，其中，s 等于 aaa，而 p 是空字符串，很明显不能匹配。
5. 用 a* 去表示一个 a。
6. p 的最后一个还是 '*'，用同样的策略。
7. 继续用 a* 去表示一个 a。
8. 用 a* 去表示空字符串。
9. 最后 s 和 p 都变成了空字符串，互相匹配。

代码实现

主函数代码如下。

```
boolean isMatch(String s, String p) {
    if (s == null || p == null) return false;

        return isMatch(s, s.length(), p, p.length());
}
```

在主函数里，进行一些简单基础的判断，如果 s 和 p 有一个是 null，则返回 false。

递归函数代码如下。

```
boolean isMatch(String s, int i, String p, int j) {
    if (j == 0) return i == 0;
        if (i == 0) {
        return j > 1 && p.charAt(j - 1) == '*' && isMatch(
s, i, p, j - 2);
    }

    if (p.charAt(j - 1) != '*') {
        return isMatch(s.charAt(i - 1), p.charAt(j - 1)) &&
                            isMatch(s, i - 1, p, j - 1);
    }

    return    isMatch(s, i, p, j - 2) || isMatch(s, i - 1,
  p, j) &&
            isMatch(s.charAt(i - 1), p.charAt(j - 2));
}

boolean isMatch(char a, char b) {
    return a == b || b == '.';
}
```

1. 递归函数的输入参数有四个，分别是字符串 s，当前字符串 s 的下标，字符串 p，以及字符串 p 的当前下标。由主函数可以看到，两个字符串的下标都是从最后一位开始。
2. 若 p 字符串为空，并且如果 s 字符串也为空，就表示匹配。
3. 当 p 字符串不为空，而 s 字符串为空，如上例所示，当 s 为空字符串，而 p 等于 a*，此时只要 p 总是由 '*'组合构成，一定能满足匹配，否则不行。
4. 若 p 的当前字符不是 '*'，判断当前的两个字符是否相等，如果相等，就递归地看前面的字符。
5. 否则，如果 p 的当前字符是 '*'：

a. 用 '*' 组合表示空字符串，看看是否能匹配；

b. 用 '*' 组合表示一个字符，看看能否匹配。

动态规划法

递归的方法比较好理解，但是容易造成重叠计算。为了避免重叠计算，可以用动态规划，自底向上地实现刚才的策略。

代码实现

a.

```
//  分别用  m  和  n  表示  s  字符串和  p  字符串的长度
boolean  isMatch(String  s,  String  p)  {
    int  m  =  s.length(),  n  =  p.length();

    //  定义一个二维布尔矩阵  dp
    boolean[][]  dp  =  new  boolean[m  +  1][n  +  1];

    //  当两个字符串的长度都为  0，也就是空字符串的时候，它们互相匹配
    dp[0][0]  =  true;

    for  (int  j  =  1;  j  <=  n;  j++)  {
        dp[0][j]  =  j  >  1  &&  p.charAt(j  -  1)  ==  '*'  &&  dp[0][j  -  2];
    }

    for  (int  i  =  1;  i  <=  m;  i++)  {
        for  (int  j  =  1;  j  <=  n;  j++)  {
            //  p  的当前字符不是  '*'，判断当前的两个字符是否相等，如果相等，就看  dp[i-1][j-1]  的值，因为它保存了前一个匹配的结果
            if  (p.charAt(j  -  1)  !=  '*')  {
                dp[i][j]  =  dp[i  -  1][j  -  1]  &&
                        isMatch(s.charAt(i  -  1),  p.charAt(j  -  1));
            }  else  {
                dp[i][j]  =  dp[i][j  -  2]  ||  dp[i  -  1][j]  &&
```

```
                                isMatch(s.charAt(i - 1), p.c
harAt(j - 2));
                }
            }
        }

        return dp[m][n];
    }

    boolean isMatch(char a, char b) {
        return a == b || b == '.';

    }
```

注意：

- 初始化二维矩阵第一行的所有值时，当 s 字符串为空，对于 p 字符串的任何一个位置，要使到这个位置的子串能和空字符串匹配，要求该子串都是由一系列的 '*' 组合构成。
- 对二维矩阵填表，运用到的逻辑跟递归一摸一样。
- p 的当前字符不是 '*'，判断当前的两个字符是否相等。如果相等，就看 dp[i-1][j-1] 的值，因为它保存了前一个匹配的结果。
- 如果 p 的当前字符是 '*'：
  - 用 '*' 组合表示空字符串，能否匹配，也就是 dp[i][j - 2]；
  - 用 '*' 组合表示一个字符，能否匹配，也就是 dp[i - 1][j]。

复杂度分析

运用动态规划，把时间复杂度控制在 $O(n^2)$，而空间复杂度也是 $O(n^2)$。