p[i+1][j-1]值一定已经计算过。这就使判断一个子串是否为回文串变得极为方便。

4．最终返回 dp[0]的值，过程结束。全部过程请参看如下代码中的 minCut 方法。

```
public int minCut(String str) {
        if (str == null || str.equals("")) {
                return 0;
        }
        char[] chas = str.toCharArray();
        int len = chas.length;
        int[] dp = new int[len + 1];
        dp[len] = -1;
        boolean[][] p = new boolean[len][len];
        for (int i = len - 1; i >= 0; i--) {
           dp[i] = Integer.MAX_VALUE;
           for (int j = i; j < len; j++) {
               if (chas[i] == chas[j] && (j - i < 2 || p[i + 1][j - 1])) {
                       p[i][j] = true;
                       dp[i] = Math.min(dp[i], dp[j + 1] + 1);
               }
           }
        }
        return dp[0];
}
```

# 字符串匹配问题

## 【题目】

给定字符串 str，其中绝对不含有字符'.'和'*'。再给定字符串 exp，其中可以含有'.'或'*'，'*'字符不能是 exp 的首字符，并且任意两个'*'字符不相邻。exp 中的'.'代表任何一个字符，exp 中的'*'表示'*'的前一个字符可以有 0 个或者多个。请写一个函数，判断 str 是否能被 exp 匹配。

## 【举例】

str="abc"，exp="abc"，返回 true。

str="abc"，exp="a.c"，exp 中单个'.'可以代表任意字符，所以返回 true。

str="abcd"，exp=".*"。exp 中'*'的前一个字符是'.'，所以可表示任意数量的'.'字符，当 exp 是"...."时与"abcd"匹配，返回 true。

str=""，exp="..*"。exp 中'*'的前一个字符是'.'，可表示任意数量的'.'字符，但是".*"之前还有一个'.'字符，该字符不受'*'的影响，所以 str 起码有一个字符才能被 exp 匹配。所以返

回 false。

## 【难度】

校 ★★★☆

## 【解答】

首先解决 str 和 exp 有效性的问题。根据描述，str 中不能含有'.'和'*'，exp 中'*'字符不能是首字符，并且任意两个'*'字符不相邻。具体请参看如下代码中的 isValid 方法。

```
public boolean isValid(char[] s, char[] e) {
        for (int i = 0; i < s.length; i++) {
                if (s[i] == '*' || s[i] == '.') {
                        return false;
                }
        }
        for (int i = 0; i < e.length; i++) {
                if (e[i] == '*' && (i == 0 || e[i - 1] == '*')) {
                        return false;
                }
        }
        return true;
}
```

接下来看如何用递归方法来解这道题，如下代码中的 isMatch 方法是递归解法的主函数，process 方法是递归的主要过程，先列出代码，然后详细解释过程。

```
public boolean isMatch(String str, String exp) {
        if (str == null || exp == null) {
                return false;
        }
        char[] s = str.toCharArray();
        char[] e = exp.toCharArray();
        return isValid(s, e) ? process(s, e, 0, 0) : false;
}

public boolean process(char[] s, char[] e, int si, int ei) {
        if (ei == e.length) {
                return si == s.length;
        }
        if (ei + 1 == e.length || e[ei + 1] != '*') {
                return si != s.length && (e[ei] == s[si] || e[ei] == '.')
                                && process(s, e, si + 1, ei + 1);
        }
        while (si != s.length && (e[ei] == s[si] || e[ei] == '.')) {
                if (process(s, e, si, ei + 2)) {
                        return true;
```

```
                }
                si++;
            }
            return process(s, e, si, ei + 2);
        }
```

下面解释一下递归过程，process 函数的意义是，从 str 的 si 位置开始，一直到 str 结束位置的子串，即 str[si...slen]，是否能被从 exp 的 ei 位置开始一直到 exp 结束位置的子串（即 exp[ei...elen]）匹配，所以 process(s,e,0,0)就是最终返回的结果。

那么在递归过程中如何判断 str[si...slen]是否能被 exp[ei...elen]匹配呢？

假设当前判断到 str 的 si 位置和 exp 的 ei 位置，即 process(s,e,si,ei)。

1．如果 ei 为 exp 的结束位置(ei==elen)，si 也是 str 的结束位置，返回 true，因为“”可以匹配“”。如果 si 不是 str 的结束位置，返回 false，这是显而易见的。

2．如果 ei 位置的下一个字符(e[ei+1])不为'*'。那么就必须关注 str[si]字符能否和 exp[ei]字符匹配。如果 str[si]与 exp[ei]能匹配(e[ei] == s[si] || e[ei] == '.')，还要关注 str 后续的部分能否被 exp 后续的部分匹配，即 process(s,e,si+1,ei+1)的返回值。如果 str[si]与 exp[ei]不能匹配，当前字符都匹配，当然不用计算后续的，直接返回 false。

3．如果当前 ei 位置的下一个字符(e[ei+1])为'*'字符。

1）如果 str[si]与 exp[ei]不能匹配，那么只能让 exp[ei..ei+1]这个部分为""，也就是 exp[ei+1]=='*'字符的前一个字符 exp[ei]的数量为 0 才行，然后考查 process(s,e,si,ei+2)的返回值。举个例子，str[si..slen]为"bXXX"，"XXX"代指字符'b'之后的字符串。exp[ei..elen]为"a*YYY"，"YYY"代指字符'*'之后的字符串。当前无法匹配('a'!='b')，所以让"a*"为""，然后考查 str[si..slen]（即"bXXX"）能否被 exp[ei+2..elen]（即"YYY"）匹配。

2）如果 str[si]与 exp[ei]能匹配，这种情况下举例说明。

str[si...slen]为"aaaaaXXX"，"XXX"指不再连续出现'a'字符的后续字符串。exp[ei...elen])为"a*YYY"，"YYY"指字符'*'之后的后续字符串。

如果令"a"和"a*"匹配，且有"aaaaXXX"和"YYY"匹配，可以返回 true。

如果令"aa"和"a*"匹配，且有"aaaXXX"和"YYY"匹配，可以返回 true。

如果令"aaa"和"a*"匹配，且有"aaXXX"和"YYY"匹配，可以返回 true。

如果令"aaaa"和"a*"匹配，且有"aXXX"和"YYY"匹配，可以返回 true。

如果令"aaaaa"和"a*"匹配，且有"XXX"和"YYY"匹配，可以返回 true。

也就是说，exp[ei..ei+1]（即"a*"）的部分如果能匹配 str 后续很多位置的时候，只要有一个返回 true，就可以直接返回 true。

整体递归过程结束。

在分析完如上递归过程之后，来看递归函数的结构。我们很容易发现递归函数process(s,e,si,ei)在每次调用的时候，有两个参数是始终不变的(s 和 e)，所以代表 process 函数状态的就是 si 和 ei 值的组合。所以，如果把递归函数 $p$ 在所有不同参数(si 和 ei)的情况下的所有返回值看作一个范围，这个范围就是一个(slen+1)*(elen+1)的二维数组，并且 p(si,ei)在整个递归过程中，依赖的总是 p(si+1,ei+1)或者 p(si+k(k>=0),ei+2)，假设二维数组 dp[i][j]代表 $p(i,j)$的返回值，dp[i][j]就只是依赖 dp[i+1][j+1]或者 dp[i+k(k>=0)][j+2]的值。进一步可以看出，想要求 dp[i][j]的值，只需要$(i,j)$位置右下方的某些值。所以只要从二维数组的右下角开始，从右到左、再从下到上地计算出二维数组 dp 中每个位置的值就可以，dp[0][0]就是最终的结果。$p(i,j)$的递归过程如何，dp[i][j]的值就怎样去计算。这种方法实际上就是动态规划的方法，省去了递归过程中很多重复计算的过程。

先从右到左计算 dp[slen][...]，也就是二维数组 dp 中的最后一行，dp[slen][elen]值的含义是 str 已经结束，剩下的字符串为""，exp 也已经结束，剩下的字符串为""，所以此时 exp 可以匹配 str，dp[slen][elen]=true。对于 dp[slen][0..elen-1]的部分，dp[slen][i]的含义是 str 已经结束，剩下的字符串为""，exp 却没有结束，剩下的字符串为 exp[i..elen-1]，什么情况下 exp[i..elen-1]可以匹配""？只能是不停地重复出现"X*"这种方式。比如，exp[i..elen-1]为"*"，这种情况下，exp[i+1..elen-1]根本不合法，匹配不了""。如果 exp[i..elen-1]="A*"，可以匹配""。如果 exp[i..elen-1]="A*B*"，也能匹配""。也就是说，在从右向左计算 dp[slen][0..elen-1]的过程中，看 exp 是不是从右往左重复出现"X*"，如果是重复出现，那么如果 exp[i]='X'，exp[i+1]='*'，令 dp[slen][i]=true，如果 exp[i]='*'，exp[i+1]='X'，令 dp[slen][i]=false。如果不是重复出现，最后一行后面的部分（即 dp[slen][0..i]），全都是 false。这样就搞定了 dp[][]最后一行的值。

再看看 dp[][]除右下角的值之外，最后一列其他位置的值，即 dp[0..slen-1][elen]。这表示如果 exp 已经结束，而 str 还没结束，显然，exp 为""匹配不了任何非空字符串，所以 dp[0..slen-1][elen]都为 false。

接着看 dp[][]倒数第二列的值，即 dp[0..slen-1][elen-1]。这表示如果 exp 还剩一个字符即（exp[elen-1]），而 str 还剩 1 个字符或多个字符。很明显，str 还剩多个字符的情况下，exp 匹配不了。str 还剩 1 个字符的情况下（即 str[slen-1]），如果和 exp[elen-1]相等，则可以匹配，或者 exp[elen-1]=='.'的情况下可以匹配。

因为 dp[i][j]只依赖 dp[i+1][j+1]或者 dp[i+k][j+2](k>=0)的值，所以在单独计算完最后一行、最后一列与倒数第二列之后，剩下的位置在从右到左、再从下到上计算 dp 值的时候，

所有依赖的值都被计算出来，直接拿过来用即可。如果 str 的长度为 $N$，exp 的长度为 $M$，因为有枚举的过程，所以时间复杂度为 $O(N^2 \times M)$，额外空间复杂度为 $O(N \times M)$。具体请参看如下代码中的 isMatchDP 方法。

```
public boolean isMatchDP(String str, String exp) {
        if (str == null || exp == null) {
                return false;
        }
        char[] s = str.toCharArray();
        char[] e = exp.toCharArray();
        if (!isValid(s, e)) {
                return false;
        }
        boolean[][] dp = initDPMap(s, e);
        for (int i = s.length - 1; i > -1; i--) {
           for (int j = e.length - 2; j > -1; j--) {
                if (e[j + 1] != '*') {
                   dp[i][j] = (s[i] == e[j] || e[j] == '.')
                                   && dp[i + 1][j + 1];
                } else {
                   int si = i;
                   while (si != s.length && (s[si] == e[j] || e[j] == '.')) {
                        if (dp[si][j + 2]) {
                                dp[i][j] = true;
                                break;
                        }
                        si++;
                   }
                   if (dp[i][j] != true) {
                        dp[i][j] = dp[si][j + 2];
                   }
                }
           }
        }
        return dp[0][0];
}

public boolean[][] initDPMap(char[] s, char[] e) {
        int slen = s.length;
        int elen = e.length;
        boolean[][] dp = new boolean[slen + 1][elen + 1];
        dp[slen][elen] = true;
        for (int j = elen - 2; j > -1; j = j - 2) {
                if (e[j] != '*' && e[j + 1] == '*') {
                        dp[slen][j] = true;
                } else {
                        break;
                }
        }
        if (slen > 0 && elen > 0) {
```

```
                if ((e[elen - 1] == '.' || s[slen - 1] == e[elen - 1])) {
                    dp[slen - 1][elen - 1] = true;
                }
            }
        }
        return dp;
    }
```

# 字典树（前缀树）的实现

## 【题目】

字典树又称为前缀树或 Trie 树，是处理字符串常见的数据结构。假设组成所有单词的字符仅是“a”～“z”，请实现字典树结构，并包含以下四个主要功能。

- void insert(String word)：添加 word，可重复添加。
- void delete(String word)：删除 word，如果 word 添加过多次，仅删除一个。
- boolean search(String word)：查询 word 是否在字典树中。
- int prefixNumber(String pre)：返回以字符串 pre 为前缀的单词数量。

## 【难度】

尉 ★★☆☆

## 【解答】

字典树的介绍。字典树是一种树形结构，优点是利用字符串的公共前缀来节约存储空间，比如加入"abc"、"abcd"、"abd"、"b"、"bcd"、"efg"、"hik"之后，字典树如图 5-1 所示。

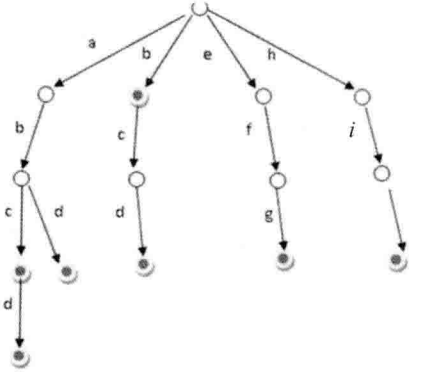


图 5-1