算法Project1 318104584 毛雨帆

Part1：Leetcode 题目

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。(Hard)

'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素
所谓匹配，是要涵盖整个字符串s的，而不是部分字符串。

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

提示：

0 <= s.length <= 20
0 <= p.length <= 30
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 *。
保证每次出现字符 * 时，前面都匹配到有效的字符

Part2：题目分析与算法设计

题目的要求，即实现一个最简单的正则表达式，即`.`与`*`的匹配。而从给出的例子中，可以看出，题目要求字符串s与字符模式p完全匹配才能算是通过，而不只是在s中找到一个p能匹配的子字符串。

法一：递归回溯

递归回溯的方法就是把遍历所有可能的情况。

eg：*s=‘aaaa’,p='a*a',如果单纯让a*继续下去，那么p中a*就可以把s全匹配完，导致的结果是，算法以为p的字符多了个a，最终s和p没匹配上。当然全部的情况也包括：a*匹配aaa，让p的最后一个a和s的最后一个a匹配，这时的结果就是匹配上了。所以递归回溯把所有情况都试一遍，有一种成功匹配那就成功了。

本题目我们实现函数isMatch（string s,string p）:通过不停的剪去s和p相同的首部，直到某一个或两个都被剪空，就可以得到结论（当然需要结合所有剪空的情况）。

首先我们从没有‘*’的最简单情况来看，这时是不是只需要扫一遍s和p，从首部开始比较对应的元素是否相同即可，如果相同就可以剪去，比较下一个即：

// 第i个下标位置元素的比较 '.'代表任何元素

s[i] == p[i] || p[i] == '.'

那么现在添加‘*’，需要注意题目要求，此时‘*’前的元素可以出现0次或者多次，那么注意在检测到p中第i个元素的下一个元素为‘*’时，就会有两种情况：

1. p的第i个元素在s中出现0次；此时，我们保持s不变，将p剪2个元素，继续调用isMatch。*如：*s=‘bb’,p='a*bb'，将p的首部2个元素剪去，得到p=‘bb’,继续比较；
2. p的第i个元在s中出现一次或者多次；此时，比较i元素与s的首元素，如果相同，剪去s的首元素，保持p不变继续调用isMatch。如：s=‘aabb’,p='a*bb',那么比较首元素相同，再剪去s首元素得s=‘abb’，在调用isMatch比较。

当然会出现这种情况：s='abb',p='a*abb';*按1来说，p剪去两个元素，s='abb',p='abb'，成功；按2来说，s='bb',p='a*abb',失败；

所以，这会把所有情况全试一遍，就得到结果了。

方法二：

用指针的方法：
1.当*s和p*都空时，返回true
2.当*s不空，*p空时，返回false
3.当*s空，*p非空时，不一定，eg: p = "a*a*a",最后一个*可以表示a出现0次*
4."是关键因素，所以我们这里分*(p + 1) == "和*(p + 1) !=''
5.(p + 1) != ''，这种情况直接匹配当前字符，如果匹配成果，继续匹配下一个，匹配失败则返回false，所谓的匹配成功（1）相同字符（2）p = '.'和s != '\0'
6.(p + 1) == '',"可以表示0个及0个以上的字符：
匹配：（1）s不动，p后移两位（2）s后移一位，p不动
不匹配：s不动，p后移两位，跳过符号'*'

方法三：

动态规划的思想也很简单，即找到最优子结构。

> 设动态`dp(i,j)`表示字符串`s`中的前`i`个字符和`p`的前`j`个字符是否匹配，值为`ture`或者`false`。

接下来就是把dp这个矩阵给填满，从初始dp[0][0]=true开始，到得到dp(s.size(),p.szie())为止。

具体流程如下：

1.状态定义：
记s第i个字符为 s[m]==s[i-1]； p中第 j 个字符记为 p[n]==p[j-1]。

记s和p的长度分别为ls，lp。

2.初始状态
初始化第一列 dp[0][2]-dp[0][lp]: dp[0][j]=dp[0][j-2] and p[j-1]=='*';

p第j个字符记为'*'且dp[0][j-2]=true。

3.转移方程

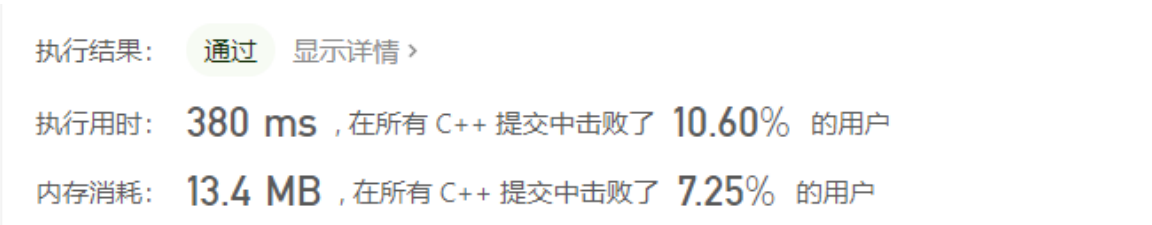
1. 当第 p[n] 为'.' 时:
    1. 当 p[n-1] 为'.' 或 s[m]==p[n-1] 时 : dp[i][j]=dp[i-1][j];
       此两种情况代表 s[m] 和 p[n-1] 可以匹配，等价于无 s[m] 的状态 dp[i-1][j]。
    2. 否则 : dp=dp :
       " Tips: 此情况代表 s[m] 和 p[n-1] 无法匹配 , p[n-1] p[n] 的组合必须出现 0 次，等价于没有 p[n-1] p[n] 时的状态 dp[i][j-2]。
2. 否则 , 当 p[n] 为'.' 或 s[m]==p[n] 时 : dp[i][j]=dp[i-1][j-1];
   此情况代表 s[m] 和 p[n] 直接匹配，当前状态等价于未匹配此两字符前的状态 d p[i-1][j-1]。

4.返回值：字符串s中前ls个字符和p中前lp个字符是否匹配 , 即 : dp[ls][lp]。

Part3：代码实现(C++)&实验结果

法一：递归回溯

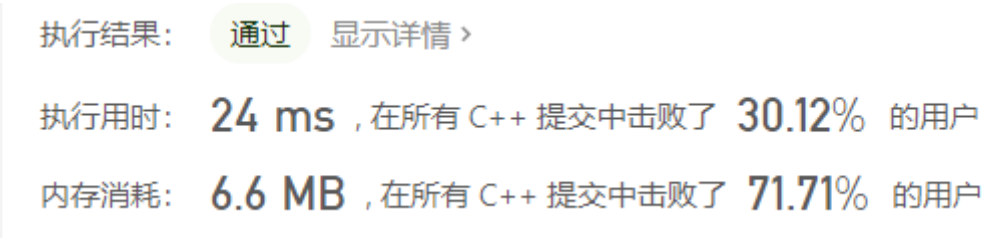
```
class Solution {
public:
    bool isMatch(string s, string p) {
        if (p.empty())
            return s.empty();
        if (p.size() > 1 && p[1] == '*')
            return isMatch(s, p.substr(2)) || (!s.empty() && (s[0] == p[0] ||
p[0] == '.') && isMatch(s.substr(1), p));
        else
            return !s.empty() && (s[0] == p[0] || p[0] == '.') &&
isMatch(s.substr(1), p.substr(1));
    }
};
```



法二：指针实现

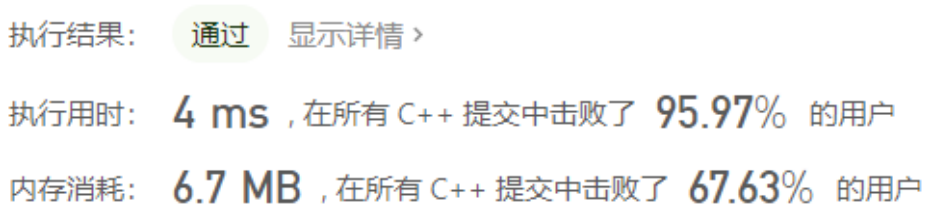
```
class Solution {
public:
    bool isMatch(string s, string p) {
        return match(s.data(), p.data());
    }
    bool match(char* s, char* p) {
        if (!*p) return !*s;
        if (*(p + 1) != '*')
```

```
            return *s == *p || (*p == '.' && *s != '\0') ? match(s + 1, p + 1) :
false;
        else
            return *s == *p || (*p == '.' && *s != '\0') ? match(s, p + 2) ||
match(s + 1, p) : match(s, p + 2);
            //或者return (*s == *p || (*p == '.' && *s != '\0')) && match(s + 1,
p) || match(s, p + 2);
    }
};
```



法三：动态规划(自顶向下)

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size();
        int n = p.size();
        bool dp[m + 1][n + 1];
        // 初始化
        for (auto &i : dp)
            for (auto &j : i)
                j = false;
        dp[0][0] = true;
        for (auto j = 0; j < n; ++ j) {
            if (p[j] == '*')
                dp[0][j + 1] = dp[0][j - 1];
        }
        for (auto i = 0; i < m; ++ i) {
            for (auto j = 0; j < n; ++ j) {
                if (s[i] == p[j] || p[j] == '.')
                    dp[i + 1][j + 1] = dp[i][j];
                else if (p[j] == '*') {
                    if (s[i] != p[j - 1] && p[j - 1] != '.')
                        dp[i + 1][j + 1] = dp[i + 1][j - 1];
                    else
                         dp[i + 1][j + 1] = dp[i][j + 1] || dp[i + 1][j] || dp[i
+ 1][j - 1];
                }
            }
        }
        return dp[m][n];
    }
};
```



Part4：分析总结

根据前面的实验结果，不难发现，从空间效率（内存消耗）和时间效率（执行用时）来看，动态规划法均是最优的，优于递归法，这也符合算法设计的初衷和原理。动态规划与递归法最大的不同便在于动态规划会存储中间的计算状态，以减少重复计算。

自顶向下的方法，跟递归法几乎没有差异，只是多使用了一个二维数组。而用自底向上的方法来解决，从最后的字符开始匹配，将多次递归调用转为在一个循环体中完成。虽然比自顶向下方法多计算了不少值，但是减少了方法调用次数，省去了多次递归调用方法的开销，而且每次计算的过程相当简单，所以并不能说它的效率比自顶向下的方法低，要视具体情况而定。

最后总结一下动态规划的步骤：

1. 抽象问题。将问题分解为多个子问题，子问题的解一旦求出就会被保存。
2. 确定状态。确认我们要求解的子问题的状态空间，并设置初始状态。
3. 确定状态转移方程。这一步是最难也是最重要的一步。

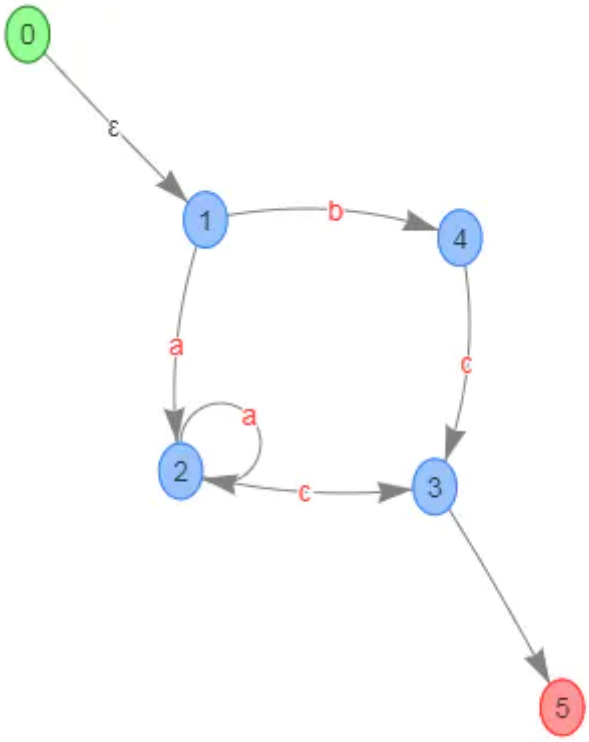
Part5：拓展

利用有限状态机实现正则表达式匹配（参考Leetcode题解）

有限状态机介绍：NFA 是指 Nondeterministic Finite Automaton，非确定有限状态自动机。有限状态机是一种用来进行对象行为建模的工具，其作用主要是描述对象在它的生命周期内所经历的状态序列，以及如何响应来自外界的各种事件。状态机可归纳为4个要素，即现态、条件、动作、次态。“现态”和“条件”是因，“动作”和“次态”是果。详解如下：
①现态：是指当前所处的状态。
②条件：又称为“事件”。当一个条件被满足，将会触发一个动作，或者执行一次状态的迁移。
③动作：条件满足后执行的动作。动作执行完毕后，可以迁移到新的状态，也可以仍旧保持原状态。动作不是必需的，当条件满足后，也可以不执行任何动作，直接迁移到新状态。
④次态：条件满足后要迁往的新状态。“次态”是相对于“现态”而言的，“次态”一旦被激活，就转变成新的“现态”了。

算法描述：

有限状态机可以用来描述字符串集合，同样是正则表达式所描述的集合，用有限状态机来表示，可以是这样的：



NFA - (a+|b)c

并且，有限状态机是可以“**执行**”的，给出如上的状态机之后，就可以用来对输入的字符串进行检测。如果最终匹配，也就意味着输入的字符串和正则表达式 `(a+|b)c` 匹配。

所以，编程语言中的正则表达式的匹配可以通过有限状态机来实现。

正则表达式匹配字符串的过程，可以分解为：1. 正则表达式转换为等价的有限状态机；2. 有限状态机输入字符串执行。

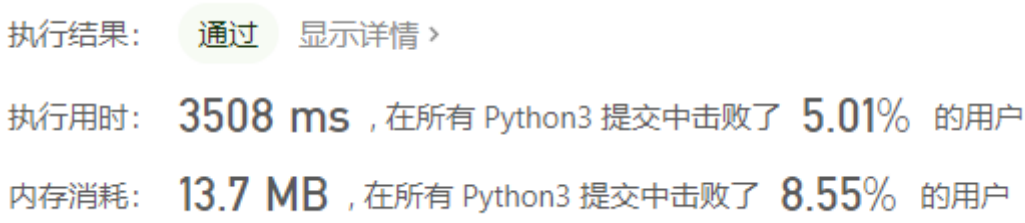
代码实现

1.Python语言实现

```
# -*- coding: utf-8 -*-

# 空转移
epsilon = "__epsilon__"


class Node:
    def __init__(self):
        self.fan_out = {}
        # 是否是一个终点
        self.is_end = False

    def link_to(self, char: str, node: 'Node'):
        """
        连接到另一个节点
        :param char:
        :param node:
        :return:
        """
        t = self.fan_out.get(char, [])
        t.append(node)
```

```
        self.fan_out[char] = t

    def get_transfer(self, char: str):
        """
        取得符合条件的转移
        :param char:
        :return:
        """
        return set(self.fan_out.get(char, [])) | set(self.fan_out.get('.', []))

    def get_epsilon(self):
        return set(self.fan_out.get(epsilon, []))


class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        start = self.build_NFA(p)
        return self.search(s, start, 0)

    def search(self, s: str, it: Node, index: int):
        """
        递归搜索状态.
        用递归写可以节约代码，还能节约一个队列。
        因为长度小于30，所以不用考虑效率
        :param s: 要匹配的字符串
        :param it: 指向当前状态
        :param index: 指向当前字符
        :return:
        """
        if index == len(s) and it.is_end:
            return True
        # 即使到了字符串末尾，依然应该先做空转移
        for node in it.get_epsilon():
            if self.search(s, node, index):
                return True
        # 到了末尾，做了空转移，不是接受状态
        if index == len(s):
            return False
        # 对于正常转移和epsilon转移，唯一不同之处在于要不要消耗一个字符
        for node in it.get_transfer(s[index]):
            if self.search(s, node, index + 1):
                return True

        return False

    @staticmethod
    def build_NFA(p: str) -> Node:
        """
        根据模式创建一个自动机
        :param p: 模式
        :return: 自动机开始节点
        """
        start = Node()
        it = start
        i = 0
        while i < len(p):
            new_node = Node()
            if i < len(p) - 1 and p[i + 1] == "*":
```

```
                # 如果是克林闭包
                it.link_to(p[i], it)
                it.link_to(epsilon, new_node)
                # 额外+1，跳过后面的星号
                i += 1

            else:
                # 星号被跳过了，而且不可能有连续星号
                assert p[i] != "*"
                # 如果不是闭包
                it.link_to(p[i], new_node)
            it = new_node
            i += 1
        it.is_end = True
        return start
```



2.Go语言实现：

```
func debug(v ...interface{}) {
    log.Println(v...)
}

func toString(i interface{}) string {
    switch i.(type) {
    case int:
        return fmt.Sprintf("%v", i)
    case string:
        return fmt.Sprintf("%v", i)
    case bool:
        return fmt.Sprintf("%v", i)
    default:
        return fmt.Sprintf("%p", i)
    }
}

func isMatch(s string, p string) bool {
    begin := new(Node)
    begin.C = '>'
    begin.Size = generatePattern(begin, p, 0)
    debug(begin.String())
    return check(begin, s, 0)
}

type Node struct {
    C        byte
    Parent   *Node
    Children map[byte][]*Node
    End      bool
    Size     int
}
```

```
func (n *Node) String() string {
    return n.StringLevel(0, make(map[*Node]bool))
}

func (n *Node) StringLevel(level int, finishNodes map[*Node]bool) string {
    r := make([]string, 0)
    if n.End {
        r = append(r, fmt.Sprintf("  id%s{%v};", toString(n), string(n.C)))
    } else {
        r = append(r, fmt.Sprintf("  id%s(%v);", toString(n), string(n.C)))
    }
    finishNodes[n] = true
    for k, v := range n.Children {
        for _, c := range v {
            if _, ok := finishNodes[c]; !ok {
                r = append(r, c.StringLevel(level+1, finishNodes))
            }
            r = append(r, fmt.Sprintf("  id%s -- %s --> id%s;", toString(n),
string(k), toString(c)))
        }
    }
    return strings.Join(r, "\n")
}

func (n *Node) Append(c byte, child *Node) {
    m := n.Children
    if m == nil {
        m = make(map[byte][]*Node)
        n.Children = m
    }
    list := m[c]
    if list == nil {
        list = make([]*Node, 0)
    }
    for _, v := range list {
        if v == child {
            m[c] = list
            return
        }
    }
    list = append(list, child)
    m[c] = list
}

func generatePattern(now *Node, str string, idx int) int {
    if len(str) <= idx {
        now.End = true
        return now.Size
    }
    vnow := now
    switch str[idx] {
    case '*':
        now.Size = 0
        now.Append(now.C, now)
    default:
        node := new(Node)
        node.C = str[idx]
```

```
            now.Append(str[idx], node)
            node.Parent = now
            node.Size = 1
            vnow = node
        }
        ret := generatePattern(vnow, str, idx+1)
        if ret == 0 {
            now.End = true
        }
        addParent := now
        for addParent.Parent != nil {
            if addParent.Size == 0 {
                debug(toString(vnow), " -> ", toString(addParent.Parent))
                addParent.Parent.Append(vnow.C, vnow)
                addParent = addParent.Parent
            } else {
                break
            }
        }
        return now.Size + ret
}

func check(now *Node, str string, idx int) bool {
        if len(str) <= idx {
            return now.End
        }
        list := now.Children['.']
        for _, v := range now.Children[str[idx]] {
            list = append(list, v)
        }
        for _, v := range list {
            r := check(v, str, idx+1)
            if r {
                return true
            }
        }
        return false
}
```

执行结果： 通过 显示详情 >

执行用时： 128 ms，在所有 Go 提交中击败了 5.16% 的用户

内存消耗： 7.3 MB，在所有 Go 提交中击败了 5.11% 的用户