# 代码实践

- leetcode0010

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<sstream>
#include<string>
#include<memory>
#include<assert.h>

using namespace std;

class Solution {
public:
    //递归
    bool isMatch0(string s, string p) {
        if (p.empty()) return s.empty();

        bool first_match = (!s.empty() && (p[0] == s[0] || p[0] == '.'));

        if(p.length() >= 2 && p[1] == '*') {
            return (isMatch0(s, p.substr(2)) || (first_match &&
isMatch0(s.substr(1), p)));
        } else {
            return first_match && isMatch0(s.substr(1), p.substr(1));
        }
    }
    //类似背包
    bool isMatch1(string s, string p) {
        int m = s.length();
        int n = p.length();
        vector<vector<bool>> dp( m + 1, vector<bool> ( n + 1, false ) );
        dp[0][0] = true;
        for( int i = 0; i <= m; i++ )
        {
            for( int j = 1; j <= n; j++ )
            {
                if( p[j - 1] == '*' )
                    dp[i][j] = dp[i][j - 2] || ( i > 0 && ( s[i - 1] == p[j - 2]
|| p[j - 2] == '.' ) && dp[i - 1][j] );
                else
                    dp[i][j] = i > 0 && dp[i - 1][j - 1] && ( s[i - 1] == p[j - 1]
|| p[j - 1] == '.' );
            }
        }
        return dp[m][n];
    }
    //优化了1
```

```
    bool isMatch2(string s, string p) {
        const auto m = s.length()+1;
        const auto n = p.length()+1;
        if(m == 0) {
            return n == 0 || p == ".*";
        } else if(n == 0) {
            return false;
        }
        auto dp = new bool[m*n];
        dp[0*n+0] = true;
        for(int i = 1; i < m; i++)
            dp[i*n+0] = false;
        for(int i = 0; i < m; i++) {
            for(int j = 1; j < n; j++) {
                if(p[j-1] == '*')
                    dp[i*n+j] = j > 1 && (dp[i*n+j-2] || (i > 0 && (s[i-1] == p[j-
2] || p[j-2] == '.') && dp[(i-1)*n+j]));
                else
                    dp[i*n+j] = i > 0 && dp[(i-1)*n+j-1] && (s[i-1] == p[j-1] ||
p[j-1] == '.');
            }
        }
        return dp[m*n-1];
    }
};

string boolToString(bool input) {
    return input ? "True" : "False";
}

int main() {
    for(string s, p; cin >> s >> p;) {
        cout << boolToString(Solution().isMatch0(s, p)) << endl;
        cout << boolToString(Solution().isMatch1(s, p)) << endl;
        cout << boolToString(Solution().isMatch2(s, p)) << endl;
    }
    return 0;
}
```

# 知识整理

▶ issue功能

## issue是什么

- issue的构成：标题、内容、频率、标签、指派者、所属项目看板、里程碑
- issue的典型功能：提问、交流；提bug；提需求

## 理论基础

### 异步沟通

**优点：**

- 跨时区合作
- 个人开发者时间成本低

**挑战：**

- 总体沟通时间成本高
- 需要经历减少沟通的回合数
- 需要尽量在一次沟通中表达尽量多的东西

### 信息沉淀

**优点：**

- 所有信息都沉淀下来，可检索
- 沉淀信息可为后来人提供有效的查询路径

## 实操进阶

### 如何降低总体沟通成本————开源社区是互惠互利，谁也不欠谁

**开发者：**

- 提issue之前先搜索一下相关issue(信息沉淀)
- 提交的内容尽量详实充分
    - 例如bug：说明运行环境(OS、软件版本等)、参数配置
    - 最好可以提供日志或最小可复现仓库
    - vscode目前issue过多，它的机器人就可以做issue的分析和提示(减少重复和检索)，减少总体沟通成本

**维护者：**

- 维护良好的社区沟通环境
- 提供有效的沟通指引
- 使用末班指引开发者提供更全面的信息
- 有效的流程设计是issue可以被更快的响应
    - 制度与流程、自动化
    - sig:标签的使用，用于issue分类
    - 如何将用户转换为贡献者
        - 增加issue响应速度，如尽量能在48小时响应(由mozilla调研)
        - 对用户的首个pr进行快速响应，使其成为长期贡献者

## 管理学的主要目的

### 在现有条件下，通过合理的组织和配置各种资源、提高生产力水平

- 异步沟通-减少同步沟通而减少核心维护者的时间成本
- 信息沉淀-减少沟通而减少开发者搜寻解决方案的成本和维护者反复回答的成本

- 流程清晰-减少决策的心智成本同时降低响应时间从而是开发者有更好的体验
    - 心智成本：决定某些事情的心理成本
- 流程自动化-减少组织的制度成本而使真题项目可以更加顺滑的运转

# 项目参与

## 特训营项目如何参与

- 编写各个文件

▶ 设计模式-开闭原则

在软件开发中，为了提高软件系统的可维护性和可复用性，增加软件的可扩展性和灵活性，程序员要尽量根据 7 条原则来开发程序，从而提高软件开发效率、节约软件开发成本和维护成本。我们将在下面的几节中依次来介绍这 7 条原则，本节首先介绍开闭原则。

# 开闭原则的定义

- 开闭原则（Open Closed Principle，OCP）由勃兰特·梅耶（Bertrand Meyer）提出，他在 1988 年的著作《面向对象软件构造》（Object Oriented Software Construction）中提出：软件实体应当对扩展开放，对修改关闭（Software entities should be open for extension，but closed for modification），这就是开闭原则的经典定义。

这里的软件实体包括以下几个部分：

- 项目中划分出的模块
- 类与接口
- 方法

开闭原则的含义是：当应用的需求改变时，在不修改软件实体的源代码或者二进制代码的前提下，可以扩展模块的功能，使其满足新的需求。

# 开闭原则的作用

开闭原则是面向对象程序设计的终极目标，它使软件实体拥有一定的适应性和灵活性的同时具备稳定性和延续性。具体来说，其作用如下。

1. 对软件测试的影响

- 软件遵守开闭原则的话，软件测试时只需要对扩展的代码进行测试就可以了，因为原有的测试代码仍然能够正常运行。

2. 可以提高代码的可复用性

- 粒度越小，被复用的可能性就越大；在面向对象的程序设计中，根据原子和抽象编程可以提高代码的可复用性。

3. 可以提高软件的可维护性

- 遵守开闭原则的软件，其稳定性高和延续性强，从而易于扩展和维护。

## 开闭原则的实现方法

- 可以通过“抽象约束、封装变化”来实现开闭原则，即通过接口或者抽象类为软件实体定义一个相对稳定的抽象层，而将相同的可变因素封装在相同的具体实现类中。
- 因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节可以从抽象派生来的实现类来进行扩展，当软件需要发生变化时，只需要根据需求重新派生一个实现类来扩展就可以了。

# 工作总结

- 今日新增2位组员，重新拟定项目计划，并简单绘制了项目的界面原型