【华为OD机考 统一考试机试C卷】符号运算（C++ Java JavaScript Python）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

2023年11月份，华为官方已经将 华为OD机考：OD统一考试（A卷 / B卷）切换到 OD统一考试（C卷）和 OD统一考试（D卷） 。根据考友反馈：目前抽到的试卷为B卷或C卷/D卷，其中C卷居多 ，按照之前的经验C卷D卷部分考题会复用A卷/B卷题，博主正积极从考过的同学收集C卷和D卷真题，可以查看下面的真题目录。

真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境 华为OD机考B卷C卷华为OD机考华为OD机考B卷华为OD机试B卷华为OD机试C卷华为OD机考C卷华为OD机考D卷题目华为OD机考C卷/D卷答案华为OD机考C卷/D卷解析华为OD机考C卷和D卷真题华为OD机考C卷和D卷题解

题目描述

给定一个表达式，求其分数计算结果。

表达式的限制如下：

1. 所有的输入数字皆为正整数（包括0）
2. 仅支持四则运算（±*/）和括号
3. 结果为整数或分数，分数必须化为最简格式（比如6，3/4，7/8，90/7）
4. 除数可能为0，如果遇到这种情况，直接输出"ERROR"
5. 输入和最终计算结果中的数字都不会超出整型范围

用例输入一定合法，不会出现括号匹配的情况

输入描述

字符串格式的表达式，仅支持±*/，数字可能超过两位，可能带有空格，没有负数

长度小于200个字符

输出描述

表达式结果，以最简格式表达

- 如果结果为整数，那么直接输出整数
- 如果结果为负数，那么分子分母不可再约分，可以为假分数，不可表达为带分数
- 结果可能是负数，符号放在前面

用例

输入	1 + 5 * 7 / 8
输出	43/8
说明	无

输入	1 / (0 - 5)
输出	-1/5
说明	符号需要提到最前面

输入	1 * (3*4/(8-(7+0)))
输出	12
说明	注意括号可以多重嵌套

解题思路

C++

```cpp
#include <iostream>
#include <stack>
#include <string>
#include <cctype>
#include <stdexcept>

```

```
using namespace std;
class Fraction {
private:
    int numerator;   // 分子
    int denominator; // 分母

    // 计算最大公约数的函数
    int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

public:
    // 构造函数
    Fraction(int numerator, int denominator) {
        if (denominator == 0) {
            throw invalid_argument("Denominator cannot be zero."); // 分母不能为0
        }
        int gcdValue = gcd(abs(numerator), abs(denominator)); // 计算最大公约数
        this->numerator = numerator / gcdValue;  // 约分
        this->denominator = denominator / gcdValue; // 约分
        if (this->denominator < 0) { // 确保分母为正
            this->numerator = -this->numerator;
            this->denominator = -this->denominator;
        }
    }

    // 加法运算
    Fraction add(const Fraction& other) const {
        return Fraction(numerator * other.denominator + other.numerator * denominator,
                        denominator * other.denominator);
    }

    // 减法运算
    Fraction subtract(const Fraction& other) const {
        return Fraction(numerator * other.denominator - other.numerator * denominator,
                        denominator * other.denominator);
    }

    // 乘法运算
    Fraction multiply(const Fraction& other) const {
```

```
        return Fraction(numerator * other.numerator, denominator * other.denominator);
    }

    // 除法运算
    Fraction divide(const Fraction& other) const {
        if (other.numerator == 0) {
            throw invalid_argument("Division by zero."); // 防止除以0
        }
        return Fraction(numerator * other.denominator, denominator * other.numerator);
    }

    // 用于输出的友元函数
    friend ostream& operator<<(ostream& os, const Fraction& f);
};

ostream& operator<<(ostream& os, const Fraction& f) {
    if (f.denominator == 1) {
        os << f.numerator;
    } else {
        os << f.numerator << "/" << f.denominator;
    }
    return os;
}

// 定义运算符优先级的函数
int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0;
    }
}

// 计算函数，使用栈操作
void calculate(stack<Fraction>& numbers, stack<char>& operators) {
```

```
    Fraction b = numbers.top(); numbers.pop();
    Fraction a = numbers.top(); numbers.pop();
    char op = operators.top(); operators.pop();

    switch (op) {
        case '+':
            numbers.push(a.add(b));
            break;
        case '-':
            numbers.push(a.subtract(b));
            break;
        case '*':
            numbers.push(a.multiply(b));
            break;
        case '/':
            numbers.push(a.divide(b));
            break;
    }
}

// 表达式计算函数
Fraction calculateExpression(const string& expression) {
    stack<Fraction> numbers;
    stack<char> operators;

    for (size_t i = 0; i < expression.length(); ++i) {
        char c = expression[i];

        if (isdigit(c)) { // 如果字符是数字
            size_t j = i;
            while (j < expression.length() && isdigit(expression[j])) {
                j++;
            }
            numbers.push(Fraction(stoi(expression.substr(i, j - i)), 1));
            i = j - 1;
        } else if (c == '(') {
            operators.push(c);
        } else if (c == ')') {
            while (operators.top() != '(') {
                calculate(numbers, operators);
```

```
            }
            operators.pop(); // 弹出'('
        } else if (c == '+' || c == '-' || c == '*' || c == '/') {
            while (!operators.empty() && precedence(c) <= precedence(operators.top())) {
                calculate(numbers, operators);
            }
            operators.push            (c);
        }
    }

    // 处理剩余的运算
    while (!operators.empty()) {
        calculate(numbers, operators);
    }

    // 返回最终结果
    return numbers.top();
}

int main() {
    string expression;

    // 读取一行表达式
    getline(cin, expression);

    try {
        // 计算表达式结果
        Fraction result = calculateExpression(expression);
        // 输出结果
        cout << result << endl;
    } catch (const exception& e) {
        // 处理异常，比如除以0
        cout << "ERROR: " << e.what() << endl;
    }

    return 0;
}
```

Java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 创建扫描器读取输入
        Scanner scanner = new Scanner(System.in);
        // 读取一行表达式
        String expression = scanner.nextLine();
        // 关闭扫描器
        scanner.close();

        try {
            // 尝试计算表达式结果
            Fraction result = calculate(expression);
            // 输出计算结果
            System.out.println(result);
        } catch (ArithmeticException e) {
            // 捕获并处理算术异常，比如除以0
            System.out.println("ERROR");
        }
    }

    private static Fraction calculate(String expression) {
        // 创建两个栈，一个用于存储数字，一个用于存储操作符
        Stack<Fraction> numbers = new Stack<>();
        Stack<Character> operators = new Stack<>();

        // 遍历表达式的每个字符
        for (int i = 0; i < expression.length(); i++) {
            char c = expression.charAt(i);

            // 如果当前字符是数字
            if (Character.isDigit(c)) {
                int j = i;
                // 继续向后读取直到非数字字符
                while (j < expression.length() && Character.isDigit(expression.charAt(j))) {
                    j++;
                }
                // 将读取到的数字字符串转换为Fraction对象并入栈
                Fraction number = new Fraction(Integer.parseInt(expression.substring(i, j)), 1);
```

```
                numbers.push(number);
                i = j - 1;
            } else if (c == '(') {
                // 如果是左括号，直接入操作符栈
                operators.push(c);
            } else if (c == ')') {
                // 如果是右括号，计算到最近一个左括号为止
                while (operators.peek() != '(') {
                    calculate(numbers, operators);
                }
                // 弹出左括号
                operators.pop();
            } else if (c == '+' || c == '-' || c == '*' || c == '/') {
                // 如果是运算符，处理优先级
                while (!operators.isEmpty() && precedence(c) <= precedence(operators.peek())) {
                    calculate(numbers, operators);
                }
                // 当前运算符入栈
                operators.push(c);
            }
        }

        // 处理剩余的运算
        while (!operators.isEmpty()) {
            calculate(numbers, operators);
        }

        // 返回计算结果
        return numbers.pop();
    }

    private static void calculate(Stack<Fraction> numbers, Stack<Character> operators) {
        // 从数字栈中弹出两个数字
        Fraction b = numbers.pop();
        Fraction a = numbers.pop();
        // 从操作符栈中弹出操作符
        char operator = operators.pop();

        // 根据操作符计算结果并入数字栈
        switch (operator) {
```

```
            case '+':
                numbers.push(a.add(b));
                break;
            case '-':
                numbers.push(a.subtract(b));
                break;
            case '*':
                numbers.push(a.multiply(b));
                break;
                    case '/':
                // 注意这里可能会抛出除以0的异常
                numbers.push(a.divide(b));
                break;
        }
    }

    private static int precedence(char operator) {
        // 定义运算符的优先级
        switch (operator) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
            default:
                return 0;
        }
    }
}

class Fraction {
    private int numerator; // 分子
    private int denominator; // 分母

    public Fraction(int numerator, int denominator) {
        // 分母不能为0
        if (denominator == 0) {
            throw new ArithmeticException("ERROR");
        }

```

```
        // 计算最大公约数
        int gcd = gcd(Math.abs(numerator), Math.abs(denominator));
        // 约分
        this.numerator = numerator / gcd;
        this.denominator = denominator / gcd;
        // 确保分母为正
        if (this.denominator < 0) {
            this.numerator = -this.numerator;
            this.denominator = -this.denominator;
        }
    }

    // 加法运算
    public Fraction add(Fraction other) {
        return new Fraction(numerator * other.denominator + other.numerator * denominator, denominator * other.denominator);
    }

    // 减法运算
    public Fraction subtract(Fraction other) {
        return new Fraction(numerator * other.denominator - other.numerator * denominator, denominator * other.denominator);
    }

    // 乘法运算
    public Fraction multiply(Fraction other) {
        return new Fraction(numerator * other.numerator, denominator * other.denominator);
    }

    // 除法运算
    public Fraction divide(Fraction other) {
        return new Fraction(numerator * other.denominator, denominator * other.numerator);
    }

    // 计算最大公约数
    private int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    // 重写toString方法，用于输出
    @Override
    public String toString() {

```

```
        if (denominator == 1) {
            return String.valueOf(numerator);
        } else {
            return numerator + "/" + denominator;
        }
    }
}
```

javaScript

```javascript
const readline = require('readline');

class Fraction {
    constructor(numerator, denominator) {
        // 分母不能为0
        if (denominator === 0) {
            throw new Error("ERROR");
        }
        // 计算最大公约数
        let gcd = this.gcd(Math.abs(numerator), Math.abs(denominator));
        // 约分
        this.numerator = numerator / gcd;
        this.denominator = denominator / gcd;
        // 确保分母为正
        if (this.denominator < 0) {
            this.numerator = -this.numerator;
            this.denominator = -this.denominator;
        }
    }

    // 加法运算
    add(other) {
        return new Fraction(this.numerator * other.denominator + other.numerator * this.denominator, this.denominator * other.denominator);
    }

    // 减法运算
    subtract(other) {
        return new Fraction(this.numerator * other.denominator - other.numerator * this.denominator, this.denominator * other.denominator);
    }

```

```
    // 乘法运算
    multiply(other) {
        return new Fraction(this.numerator * other.numerator, this.denominator * other.denominator);
    }

    // 除法运算
    divide(other) {
        return new Fraction(this.numerator * other.denominator, this.denominator * other.numerator);
    }

    // 计算最大公约数
    gcd(a, b) {
        return b === 0 ? a : this.gcd(b, a % b);
    }

    // 重写toString方法，用于输出
    toString() {
        if (this.denominator === 1) {
            return String(this.numerator);
        } else {
            return this.numerator + "/" + this.denominator;
        }
    }
}

function calculate(expression) {
    // 创建两个栈，一个用于存储数字，一个用于存储操作符
    let numbers = [];
    let operators = [];

    // 遍历表达式的每个字符
    for (let i = 0; i < expression.length; i++) {
        let c = expression.charAt(i);

        // 如果当前字符是数字
        if (!isNaN(c)) {
            let j = i;
            // 继续向后读取直到非数字字符
            while (j < expression.length && !isNaN(expression.charAt(j))) {
                j++;
```

```
            }
            // 将读取到的数字字符串转换为Fraction对象并入栈
            let number = new Fraction(parseInt(expression.substring(i, j)), 1);
            numbers.push(number);
            i = j - 1;
        } else if (c === '(') {
            // 如果是左括号，直接入操作符栈
            operators.push(c);
        } else if (c === ')') {
            // 如果是右括号，计算到最近一个左括号为止
            while (operators[operators.length - 1] !== '(') {
                performCalculation(numbers, operators);
            }
            // 弹出左括号
            operators.pop();
        } else if (c === '+' || c === '-' || c === '*' || c === '/') {
            // 如果是运算符，处理优先级
            while (operators.length > 0 && precedence(c) <= precedence(operators[operators.length - 1])) {
                performCalculation(numbers, operators);
            }
            // 当前运算符入栈
            operators.push(c);
        }
    }

    // 处理剩余的运算
    while (operators.length > 0) {
        performCalculation(numbers, operators);
    }

    // 返回计算结果
    return numbers.pop();
}

function performCalculation(numbers, operators) {
    // 从数字栈中弹出两个数字
    let b = numbers.pop();
    let a = numbers.pop();
    // 从操作符栈中弹出操作符
    let operator = operators.pop();

```

```
    // 根据操作符计算结果并入数字栈
    switch (operator) {
        case '+':
            numbers.push(a.add(b));
            break;
        case '-':
            numbers.push(a.subtract(b));
            break;
        case '*':
            numbers.push(a.multiply(b));
            break;
        case '/':
            // 注意这里可能会抛出除以0的异常
            numbers.push(a.divide(b));
            break;
    }
}

function precedence(operator) {
    // 定义运算符的优先级
    switch (operator) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0;
    }
}

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.on("line", (expression) => {
     try {

```

```
        // 尝试计算表达式结果
        let result = calculate(expression);
        // 输出计算结果
        console.log(result.toString());
    } catch (e) {
        // 捕获并处理算术异常，比如除以0
        console.log("ERROR");
    }
    rl.close();
});
```

Python

```
import fractions

def main():
    # 输入一行表达式
    expression = input("")

    try:
        # 计算表达式结果
        result = calculate(expression)
        # 输出计算结果
        print(result)
    except ArithmeticException:
        # 捕获并处理算术异常，比如除以0
        print("ERROR")

def calculate(expression):
    # 创建两个栈，一个用于存储数字，一个用于存储操作符
    numbers = []
    operators = []

    i = 0
    while i < len(expression):
        c = expression[i]

        # 如果当前字符是数字
        if c.isdigit():
            j = i
```

```
            # 继续向后读取直到非数字字符
            while j < len(expression) and expression[j].isdigit():
                j += 1
            # 将读取到的数字字符串转换为Fraction对象并入栈
            number = fractions.Fraction(int(expression[i:j]))
            numbers.append(number)
            i = j
        elif c == '(':
            # 如果是左括号，直接入操作符栈
            operators.append(c)
            i += 1
        elif c == ')':
            # 如果是右括号，计算到最近一个左括号为止
            while operators[-1] != '(':
                perform_calculation(numbers, operators)
            # 弹出左括号
            operators.pop()
            i += 1
        elif c in '+-*/':
            # 如果是运算符，处理优先级
            while operators and precedence(c) <= precedence(operators[-1]):
                perform_calculation(numbers, operators)
            # 当前运算符入栈
            operators.append(c)
            i += 1
        else:
            # 忽略非法字符
            i += 1

    # 处理剩余的运算
    while operators:
        perform_calculation(numbers, operators)

    # 返回计算结果
    return numbers.pop()

def perform_calculation(numbers, operators):
    # 从数字栈中弹出两个数字
    b = numbers.pop()
    a = numbers.pop()
```

```
    # 从操作符栈中弹出操作符
    operator = operators.pop()

    # 根据操作符计算结果并入数字栈
    if operator == '+':
        numbers.append(a + b)
    elif operator == '-':
        numbers.append(a - b)
    elif operator == '*':
        numbers.append(a * b)
    elif operator == '/':
        # 注意这里可能会抛出除以0的异常
        numbers.append(a / b)

def precedence(operator):
    # 定义运算符的优先级
    if operator in '+-':
        return 1
    elif operator in '*/':
        return 2
    else:
        return 0

if __name__ == "__main__":
    main()
```

文章目录


机考真题
华为OD
No No No
CSDN @算法大师