

【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](#)[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（ \pm / \times ）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, $\frac{3}{4}$, $\frac{7}{8}$, $\frac{90}{7}$ ）
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 \pm / \times ，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++

```
1 #include <iostream>
2 #include <stack>
3 #include <string>
4 #include <cctype>
5 #include <stdexcept>
6
```

```
7 using namespace std;
8 class Fraction {
9 private:
10     int numerator;    // 分子
11     int denominator; // 分母
12
13     // 计算最大公约数的函数
14     int gcd(int a, int b) {
15         return b == 0 ? a : gcd(b, a % b);
16     }
17
18 public:
19     // 构造函数
20     Fraction(int numerator, int denominator) {
21         if (denominator == 0) {
22             throw invalid_argument("Denominator cannot be zero."); // 分母不能为0
23         }
24         int gcdValue = gcd(abs(numerator), abs(denominator)); // 计算最大公约数
25         this->numerator = numerator / gcdValue; // 约分
26         this->denominator = denominator / gcdValue; // 约分
27         if (this->denominator < 0) { // 确保分母为正
28             this->numerator = -this->numerator;
29             this->denominator = -this->denominator;
30         }
31     }
32
33     // 加法运算
34     Fraction add(const Fraction& other) const {
35         return Fraction(numerator * other.denominator + other.numerator * denominator,
36                         denominator * other.denominator);
37     }
38
39     // 减法运算
40     Fraction subtract(const Fraction& other) const {
41         return Fraction(numerator * other.denominator - other.numerator * denominator,
42                         denominator * other.denominator);
43     }
44
45     // 乘法运算
46     Fraction multiply(const Fraction& other) const {
47
```

```
47         return Fraction(numerator * other.numerator, denominator * other.denominator);
48     }
49
50 // 除法运算
51 Fraction divide(const Fraction& other) const {
52     if (other.numerator == 0) {
53         throw invalid_argument("Division by zero."); // 防止除以0
54     }
55     return Fraction(numerator * other.denominator, denominator * other.numerator);
56 }
57
58 // 用于输出的友元函数
59 friend ostream& operator<<(ostream& os, const Fraction& f);
60 };
61
62 ostream& operator<<(ostream& os, const Fraction& f) {
63     if (f.denominator == 1) {
64         os << f.numerator;
65     } else {
66         os << f.numerator << "/" << f.denominator;
67     }
68     return os;
69 }
70
71 // 定义运算符优先级的函数
72 int precedence(char op) {
73     switch (op) {
74         case '+':
75         case '-':
76             return 1;
77         case '*':
78         case '/':
79             return 2;
80         default:
81             return 0;
82     }
83 }
84
85 // 计算函数, 使用栈操作
86 void calculate(stack<Fraction>& numbers, stack<char>& operators) {
87
88 }
```

```
88     Fraction b = numbers.top(); numbers.pop();
89     Fraction a = numbers.top(); numbers.pop();
90     char op = operators.top(); operators.pop();
91
92     switch (op) {
93     case '+':
94         numbers.push(a.add(b));
95         break;
96     case '-':
97         numbers.push(a.subtract(b));
98         break;
99     case '*':
100         numbers.push(a.multiply(b));
101         break;
102     case '/':
103         numbers.push(a.divide(b));
104         break;
105     }
106 }
107
108 // 表达式计算函数
109 Fraction calculateExpression(const string& expression) {
110     stack<Fraction> numbers;
111     stack<char> operators;
112
113     for (size_t i = 0; i < expression.length(); ++i) {
114         char c = expression[i];
115
116         if (isdigit(c)) { // 如果字符是数字
117             size_t j = i;
118             while (j < expression.length() && isdigit(expression[j])) {
119                 j++;
120             }
121             numbers.push(Fraction(stoi(expression.substr(i, j - i)), 1));
122             i = j - 1;
123         } else if (c == '(') {
124             operators.push(c);
125         } else if (c == ')') {
126             while (operators.top() != '(') {
127                 calculate(numbers, operators);
128             }
```

```
129     }
130     operators.pop(); // 弹出 '('
131 } else if (c == '+' || c == '-' || c == '*' || c == '/') {
132     while (!operators.empty() && precedence(c) <= precedence(operators.top())) {
133         calculate(numbers, operators);
134     }
135     operators.push(c);
136 }
137 }
138
139 // 处理剩余的运算
140 while (!operators.empty()) {
141     calculate(numbers, operators);
142 }
143
144 // 返回最终结果
145 return numbers.top();
146 }
147
148 int main() {
149     string expression;
150
151     // 读取一行表达式
152     getline(cin, expression);
153
154     try {
155         // 计算表达式结果
156         Fraction result = calculateExpression(expression);
157         // 输出结果
158         cout << result << endl;
159     } catch (const exception& e) {
160         // 处理异常, 比如除以0
161         cout << "ERROR: " << e.what() << endl;
162     }
163
164     return 0;
165 }
```

Java

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         // 创建扫描器读取输入
6         Scanner scanner = new Scanner(System.in);
7         // 读取一行表达式
8         String expression = scanner.nextLine();
9         // 关闭扫描器
10        scanner.close();
11
12        try {
13            // 尝试计算表达式结果
14            Fraction result = calculate(expression);
15            // 输出计算结果
16            System.out.println(result);
17        } catch (ArithmeticException e) {
18            // 捕获并处理算术异常, 比如除以0
19            System.out.println("ERROR");
20        }
21    }
22
23    private static Fraction calculate(String expression) {
24        // 创建两个栈, 一个用于存储数字, 一个用于存储操作符
25        Stack<Fraction> numbers = new Stack<>();
26        Stack<Character> operators = new Stack<>();
27
28        // 遍历表达式的每个字符
29        for (int i = 0; i < expression.length(); i++) {
30            char c = expression.charAt(i);
31
32            // 如果当前字符是数字
33            if (Character.isDigit(c)) {
34                int j = i;
35                // 继续向后读取直到非数字字符
36                while (j < expression.length() && Character.isDigit(expression.charAt(j))) {
37                    j++;
38                }
39                // 将读取到的数字字符串转换为Fraction对象并入栈
40                Fraction number = new Fraction(Integer.parseInt(expression.substring(i, j)), 1);
41            }
```



```
41         numbers.push(number);
42         i = j - 1;
43     } else if (c == '(') {
44         // 如果是左括号, 直接入操作符栈
45         operators.push(c);
46     } else if (c == ')') {
47         // 如果是右括号, 计算到最近一个左括号为止
48         while (operators.peek() != '(') {
49             calculate(numbers, operators);
50         }
51         // 弹出左括号
52         operators.pop();
53     } else if (c == '+' || c == '-' || c == '*' || c == '/') {
54         // 如果是运算符, 处理优先级
55         while (!operators.isEmpty() && precedence(c) <= precedence(operators.peek())) {
56             calculate(numbers, operators);
57         }
58         // 当前运算符入栈
59         operators.push(c);
60     }
61 }
62
63 // 处理剩余的运算
64 while (!operators.isEmpty()) {
65     calculate(numbers, operators);
66 }
67
68 // 返回计算结果
69 return numbers.pop();
70 }
71
72 private static void calculate(Stack<Fraction> numbers, Stack<Character> operators) {
73     // 从数字栈中弹出两个数字
74     Fraction b = numbers.pop();
75     Fraction a = numbers.pop();
76     // 从操作符栈中弹出操作符
77     char operator = operators.pop();
78
79     // 根据操作符计算结果并入数字栈
80     switch (operator) {
81         case '+':
```

```
82         case '+':
83             numbers.push(a.add(b));
84             break;
85         case '-':
86             numbers.push(a.subtract(b));
87             break;
88         case '*':
89             numbers.push(a.multiply(b));
90             break;
91             case '/':
92                 // 注意这里可能会抛出除以0的异常
93                 numbers.push(a.divide(b));
94                 break;
95     }
96 }
97
98 private static int precedence(char operator) {
99     // 定义运算符的优先级
100     switch (operator) {
101         case '+':
102         case '-':
103             return 1;
104         case '*':
105         case '/':
106             return 2;
107         default:
108             return 0;
109     }
110 }
111 }
112
113 class Fraction {
114     private int numerator; // 分子
115     private int denominator; // 分母
116
117     public Fraction(int numerator, int denominator) {
118         // 分母不能为0
119         if (denominator == 0) {
120             throw new ArithmeticException("ERROR");
121         }
122     }
123 }
```

```
123 // 计算最大公约数
124 int gcd = gcd(Math.abs(numerator), Math.abs(denominator));
125 // 约分
126 this.numerator = numerator / gcd;
127 this.denominator = denominator / gcd;
128 // 确保分母为正
129 if (this.denominator < 0) {
130     this.numerator = -this.numerator;
131     this.denominator = -this.denominator;
132 }
133 }
134
135 // 加法运算
136 public Fraction add(Fraction other) {
137     return new Fraction(numerator * other.denominator + other.numerator * denominator, denominator * other.denominator);
138 }
139
140 // 减法运算
141 public Fraction subtract(Fraction other) {
142     return new Fraction(numerator * other.denominator - other.numerator * denominator, denominator * other.denominator);
143 }
144
145 // 乘法运算
146 public Fraction multiply(Fraction other) {
147     return new Fraction(numerator * other.numerator, denominator * other.denominator);
148 }
149
150 // 除法运算
151 public Fraction divide(Fraction other) {
152     return new Fraction(numerator * other.denominator, denominator * other.numerator);
153 }
154
155 // 计算最大公约数
156 private int gcd(int a, int b) {
157     return b == 0 ? a : gcd(b, a % b);
158 }
159
160 // 重写toString方法, 用于输出
161 @Override
162 public String toString() {
163
```

```
164     if (denominator == 1) {
165         return String.valueOf(numerator);
166     } else {
167         return numerator + "/" + denominator;
168     }
169 }
170 }
```

javaScript

```
1  const readline = require('readline');
2
3  class Fraction {
4      constructor(numerator, denominator) {
5          // 分母不能为0
6          if (denominator === 0) {
7              throw new Error("ERROR");
8          }
9          // 计算最大公约数
10         let gcd = this.gcd(Math.abs(numerator), Math.abs(denominator));
11         // 约分
12         this.numerator = numerator / gcd;
13         this.denominator = denominator / gcd;
14         // 确保分母为正
15         if (this.denominator < 0) {
16             this.numerator = -this.numerator;
17             this.denominator = -this.denominator;
18         }
19     }
20
21     // 加法运算
22     add(other) {
23         return new Fraction(this.numerator * other.denominator + other.numerator * this.denominator, this.denominator * other.denominator);
24     }
25
26     // 减法运算
27     subtract(other) {
28         return new Fraction(this.numerator * other.denominator - other.numerator * this.denominator, this.denominator * other.denominator);
29     }
30
31 }
```

```
31 // 乘法运算
32 multiply(other) {
33     return new Fraction(this.numerator * other.numerator, this.denominator * other.denominator);
34 }
35
36 // 除法运算
37 divide(other) {
38     return new Fraction(this.numerator * other.denominator, this.denominator * other.numerator);
39 }
40
41 // 计算最大公约数
42 gcd(a, b) {
43     return b === 0 ? a : this.gcd(b, a % b);
44 }
45
46 // 重写toString方法, 用于输出
47 toString() {
48     if (this.denominator === 1) {
49         return String(this.numerator);
50     } else {
51         return this.numerator + "/" + this.denominator;
52     }
53 }
54 }
55
56 function calculate(expression) {
57     // 创建两个栈, 一个用于存储数字, 一个用于存储操作符
58     let numbers = [];
59     let operators = [];
60
61     // 遍历表达式的每个字符
62     for (let i = 0; i < expression.length; i++) {
63         let c = expression.charAt(i);
64
65         // 如果当前字符是数字
66         if (!isNaN(c)) {
67             let j = i;
68             // 继续向后读取直到非数字字符
69             while (j < expression.length && !isNaN(expression.charAt(j))) {
70                 j++;
71             }
72         }
73     }
74 }
```

```
72     }
73     // 将读取到的数字字符串转换为Fraction对象并入栈
74     let number = new Fraction(parseInt(expression.substring(i, j)), 1);
75     numbers.push(number);
76     i = j - 1;
77 } else if (c === '(') {
78     // 如果是左括号, 直接入操作符栈
79     operators.push(c);
80 } else if (c === ')') {
81     // 如果是右括号, 计算到最近一个左括号为止
82     while (operators[operators.length - 1] !== '(') {
83         performCalculation(numbers, operators);
84     }
85     // 弹出左括号
86     operators.pop();
87 } else if (c === '+' || c === '-' || c === '*' || c === '/') {
88     // 如果是运算符, 处理优先级
89     while (operators.length > 0 && precedence(c) <= precedence(operators[operators.length - 1])) {
90         performCalculation(numbers, operators);
91     }
92     // 当前运算符入栈
93     operators.push(c);
94 }
95 }
96
97 // 处理剩余的运算
98 while (operators.length > 0) {
99     performCalculation(numbers, operators);
100 }
101
102 // 返回计算结果
103 return numbers.pop();
104 }
105
106 function performCalculation(numbers, operators) {
107     // 从数字栈中弹出两个数字
108     let b = numbers.pop();
109     let a = numbers.pop();
110     // 从操作符栈中弹出操作符
111     let operator = operators.pop();
112 }
```

```
113
114 // 根据操作符计算结果并入数字栈
115 switch (operator) {
116     case '+':
117         numbers.push(a.add(b));
118         break;
119     case '-':
120         numbers.push(a.subtract(b));
121         break;
122     case '*':
123         numbers.push(a.multiply(b));
124         break;
125     case '/':
126         // 注意这里可能会抛出除以0的异常
127         numbers.push(a.divide(b));
128         break;
129 }
130 }
131
132 function precedence(operator) {
133     // 定义运算符的优先级
134     switch (operator) {
135         case '+':
136         case '-':
137             return 1;
138         case '*':
139         case '/':
140             return 2;
141         default:
142             return 0;
143     }
144 }
145
146 const rl = readline.createInterface({
147     input: process.stdin,
148     output: process.stdout
149 });
150
151 rl.on("line", (expression) => {
152     try {
153
```

```
154     // 尝试计算表达式结果
155     let result = calculate(expression);
156     // 输出计算结果
157     console.log(result.toString());
158 } catch (e) {
159     // 捕获并处理算术异常, 比如除以0
160     console.log("ERROR");
161 }
162 rl.close();
163 });
```

Python

```
1 import fractions
2
3 def main():
4     # 输入一行表达式
5     expression = input("")
6
7     try:
8         # 计算表达式结果
9         result = calculate(expression)
10        # 输出计算结果
11        print(result)
12    except ArithmeticException:
13        # 捕获并处理算术异常, 比如除以0
14        print("ERROR")
15
16 def calculate(expression):
17     # 创建两个栈, 一个用于存储数字, 一个用于存储操作符
18     numbers = []
19     operators = []
20
21     i = 0
22     while i < len(expression):
23         c = expression[i]
24
25         # 如果当前字符是数字
26         if c.isdigit():
27             j = i
28             while j < len(expression) and expression[j].isdigit():
29                 num = int(expression[i:j+1])
30                 numbers.append(num)
31                 j += 1
32             i = j
33         elif c in '+-*/':
34             operators.append(c)
35         elif c == '(':
36             operators.append(c)
37         elif c == ')':
38             while operators[-1] != '(':
39                 operators.pop()
40             operators.pop()
41         i += 1
42
43     while len(operators) > 0:
44         op = operators.pop()
45         if op == '+':
46             numbers.append(numbers[-1] + numbers[-2])
47         elif op == '-':
48             numbers.append(numbers[-1] - numbers[-2])
49         elif op == '*':
50             numbers.append(numbers[-1] * numbers[-2])
51         elif op == '/':
52             numbers.append(numbers[-1] / numbers[-2])
53
54     return numbers[-1]
```



```
28     # 继续向后读取直到非数字字符
29     while j < len(expression) and expression[j].isdigit():
30         j += 1
31     # 将读取到的数字字符串转换为Fraction对象并入栈
32     number = fractions.Fraction(int(expression[i:j]))
33     numbers.append(number)
34     i = j
35 elif c == '(':
36     # 如果是左括号, 直接入操作符栈
37     operators.append(c)
38     i += 1
39 elif c == ')':
40     # 如果是右括号, 计算到最近一个左括号为止
41     while operators[-1] != '(':
42         perform_calculation(numbers, operators)
43     # 弹出左括号
44     operators.pop()
45     i += 1
46 elif c in '+-*/':
47     # 如果是运算符, 处理优先级
48     while operators and precedence(c) <= precedence(operators[-1]):
49         perform_calculation(numbers, operators)
50     # 当前运算符入栈
51     operators.append(c)
52     i += 1
53 else:
54     # 忽略非法字符
55     i += 1
56
57 # 处理剩余的运算
58 while operators:
59     perform_calculation(numbers, operators)
60
61 # 返回计算结果
62 return numbers.pop()
63
64 def perform_calculation(numbers, operators):
65     # 从数字栈中弹出两个数字
66     b = numbers.pop()
67     a = numbers.pop()
68     --
```

```
69     # 从操作符栈中弹出操作符
70     operator = operators.pop()
71
72     # 根据操作符计算结果并入数字栈
73     if operator == '+':
74         numbers.append(a + b)
75     elif operator == '-':
76         numbers.append(a - b)
77     elif operator == '*':
78         numbers.append(a * b)
79     elif operator == '/':
80         # 注意这里可能会抛出除以0的异常
81         numbers.append(a / b)
82
83 def precedence(operator):
84     # 定义运算符的优先级
85     if operator in '+-':
86         return 1
87     elif operator in '*/':
88         return 2
89     else:
90         return 0
91
92 if __name__ == "__main__":
93     main()
```

文章目录

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

[题目描述](#)

[输入描述](#)

[输出描述](#)

[用例](#)

[解题思路](#)

[C++](#)

[Java](#)

[javaScript](#)

[Python](#)

