# 题目

实现一个基本的计算器来计算简单的表达式字符串。

表达式字符串只包含非负整数，算符 +、-、\*、/ ，左括号 ( 和右括号 ) 。整数除法需要 向下截断 。

你可以假定给定的表达式总是有效的。所有的中间结果的范围均满足 [-231, 231 - 1] 。

注意：你不能使用任何将字符串作为表达式求值的内置函数，比如 eval() 。

示例 1：

输入：s = "1+1"

输出：2

示例 2：

输入：s = "6-4/2"

输出：4

示例 3：

输入：s = "2\*(5+5\*2)/3+(6/2+8)"

输出：21

提示：

1 <= s <= 104

s 由整数、'+'、'-'、'\*'、'/'、'(' 和 ')' 组成

s 是一个 有效的 表达式

# 分析

要解决“计算包含加减乘除和括号的表达式”问题，核心思路是利用栈处理运算符优先级和括号，通过“先乘除后加减”的规则分阶段计算，并借助栈暂存中间结果和运算符，以处理括号带来的优先级提升。

解题思路

1、运算符优先级与括号处理：

- 乘除（`\*`、`/`）优先级高于加减（`+`、`-`）。

- 括号会提升内部表达式的优先级，需优先计算括号内的结果。

- 栈的作用：暂存未计算的操作数和运算符，当遇到高优先级运算符或括号时，通过入栈/出栈调整计算顺序。

2、核心步骤：

- 预处理：过滤空格，简化解析。

- 数字解析：拼接多位数（如“123”解析为123）。

- 运算符处理：

- 遇到`+`、`-`：若栈顶有`\*`、`/`，先计算栈顶的乘除，再将当前运算符入栈。

- 遇到`\*`、`/`：直接入栈（因其优先级最高，后续遇到数字时立即计算）。

- 括号处理：

- 遇到`(`：将当前计算的中间结果和运算符栈入栈，重置中间结果和运算符栈，开始计算括号内表达式。

- 遇到`)`：计算括号内的最终结果，弹出栈中保存的外层结果和运算符栈，将括号内结果作为新操作数继续计算。

- 最终计算：遍历结束后，处理栈中剩余的运算符，得到最终结果。

代码：

class Solution {

public:

int calculate(string s) {

// 用于存储操作数的栈（主栈）

vector<int> nums;

// 用于存储运算符的栈

vector<char> ops;

int n = s.size();

int i = 0;

while (i < n) {

if (s[i] == ' ') {

// 过滤空格

i++;

continue;

} else if (isdigit(s[i])) {

// 解析多位数

int num = 0;

while (i < n && isdigit(s[i])) {

num = num \* 10 + (s[i] - '0');

i++;

}

nums.push\_back(num);

} else if (s[i] == '(') {

// 遇到左括号：将当前nums和ops入栈保存，重置当前nums和ops

ops.push\_back('(');

i++;

} else if (s[i] == ')') {

// 遇到右括号：计算括号内的结果，直到匹配到左括号

while (ops.back() != '(') {

calculateTop(nums, ops);

}

ops.pop\_back(); // 弹出左括号

i++;

} else {

// 遇到运算符：处理栈中优先级更高或相等的运算符（乘除）

while (!ops.empty() && ops.back() != '(' && hasHigherPriority(ops.back(), s[i])) {

calculateTop(nums, ops);

}

ops.push\_back(s[i]);

i++;

}

}

// 处理剩余的运算符

while (!ops.empty()) {

calculateTop(nums, ops);

}

return nums.back();

}

private:

// 判断op1的优先级是否高于或等于op2

// 乘除（\*、/）优先级高于加减（+、-）

bool hasHigherPriority(char op1, char op2) {

if ((op1 == '\*' || op1 == '/') && (op2 == '+' || op2 == '-')) {

return true;

}

// 同优先级（如+和-，\*和/）也需要先计算（按从左到右顺序）

if ((op1 == '+' || op1 == '-') && (op2 == '+' || op2 == '-')) {

return true;

}

if ((op1 == '\*' || op1 == '/') && (op2 == '\*' || op2 == '/')) {

return true;

}

return false;

}

// 计算栈顶的一个运算符和对应的两个操作数

void calculateTop(vector<int>& nums, vector<char>& ops) {

if (nums.size() < 2 || ops.empty()) {

return;

}

int b = nums.back(); nums.pop\_back(); // 第二个操作数（右）

int a = nums.back(); nums.pop\_back(); // 第一个操作数（左）

char op = ops.back(); ops.pop\_back();

int res;

switch (op) {

case '+': res = a + b; break;

case '-': res = a - b; break;

case '\*': res = a \* b; break;

case '/': res = a / b; break; // 整数除法向下截断（题目保证有效）

}

nums.push\_back(res);

}

};

代码解释

1、预处理与数字解析：

过滤空格，通过循环拼接连续数字字符（如“123”→123），存入nums栈。

2、运算符优先级判断：

hasHigherPriority函数定义优先级规则：乘除优先级高于加减，同优先级运算符（如`+`与`-`、`\*`与`/`）按从左到右顺序计算（返回true表示栈顶运算符需先计算）。

3、栈顶计算：

calculateTop函数弹出栈顶的两个操作数和一个运算符，计算结果后将结果压回nums栈（如nums = [3,4]，ops = ['\*'] → 计算3\*4=12，nums变为[12]）。

4、括号处理：

- 左括号`(`直接入`ops`栈，标记括号开始。

- 右括号`)`触发括号内计算：反复调用calculateTop，直到弹出左括号`(`，此时nums栈顶为括号内表达式的结果。

5、最终计算：

遍历结束后，栈中可能剩余未计算的运算符（如1+2+3中最后一个`+`），通过calculateTop处理剩余运算符，得到最终结果。

复杂度分析

- 时间复杂度：O(n)，其中n是表达式长度。每个字符仅被遍历一次，栈的入栈、出栈和计算操作均为O(1)（每个元素最多入栈和出栈一次）。

- 空间复杂度：O(n)。最坏情况下（如全是括号嵌套或运算符），nums和ops栈的空间开销为O(n)。