# 题目

给你一个字符串表达式 s ，请你实现一个基本计算器来计算并返回它的值。

注意:不允许使用任何将字符串作为数学表达式计算的内置函数，比如 eval() 。

示例 1：

输入：s = "1 + 1"

输出：2

示例 2：

输入：s = " 2-1 + 2 "

输出：3

示例 3：

输入：s = "(1+(4+5+2)-3)+(6+8)"

输出：23

提示：

1 <= s.length <= 3 \* 105

s 由数字、'+'、'-'、'('、')'、和 ' ' 组成

s 表示一个有效的表达式

'+' 不能用作一元运算(例如， "+1" 和 "+(2 + 3)" 无效)

'-' 可以用作一元运算(即 "-1" 和 "-(2 + 3)" 是有效的)

输入中不存在两个连续的操作符

每个数字和运行的计算将适合于一个有符号的 32位 整数

# 分析

要解决“实现基本计算器计算字符串表达式的值”的问题，核心思路是利用栈处理括号和运算符的优先级，通过维护当前计算结果和符号，逐步解析表达式并处理括号内的子表达式。

解题思路

1、处理空格：首先过滤掉表达式中的所有空格，简化解析过程。

2、维护当前值和符号：

- 使用current\_num存储当前解析的数字（如“123”需逐步拼接为123）。

- 使用sign记录当前运算的符号（+1表示正，-1表示负）。

3、栈的作用：当遇到左括号`(`时，将当前的计算结果和符号入栈，然后重置当前结果和符号，开始计算括号内的子表达式；当遇到右括号`)`时，弹出栈顶的符号和结果，与当前子表达式的结果合并（`栈顶结果 + 栈顶符号 × 当前子表达式结果`）。

4、运算符处理：遇到`+`或`-`时，先将current\_num按当前sign累加到result中，再更新sign为新的符号，并重置current\_num。

5、结束处理：表达式解析结束后，需将最后一个current\_num累加到result中，得到最终结果。

代码：

class Solution {

public:

int calculate(string s) {

stack<int> st; // 存储括号前的结果和符号

int result = 0; // 当前计算结果

int current\_num = 0; // 当前解析的数字

int sign = 1; // 当前符号：1为正，-1为负

for (char c : s) {

if (c == ' ') {

continue; // 过滤空格

} else if (isdigit(c)) {

// 拼接多位数（如"123" → 1×100 + 2×10 + 3）

current\_num = current\_num \* 10 + (c - '0');

} else if (c == '+') {

// 累加当前数字到结果，更新符号为正

result += sign \* current\_num;

current\_num = 0;

sign = 1;

} else if (c == '-') {

// 累加当前数字到结果，更新符号为负

result += sign \* current\_num;

current\_num = 0;

sign = -1;

} else if (c == '(') {

// 遇到左括号：将当前结果和符号入栈，重置结果和符号

st.push(result);

st.push(sign);

result = 0;

sign = 1;

} else if (c == ')') {

// 遇到右括号：累加当前数字，与栈中结果合并

result += sign \* current\_num;

current\_num = 0;

// 弹出栈顶符号（括号前的符号）

result \*= st.top();

st.pop();

// 弹出栈顶结果（括号前的结果），累加得到新结果

result += st.top();

st.pop();

}

}

// 处理最后一个数字（表达式末尾无运算符的情况）

result += sign \* current\_num;

return result;

}

};

代码解释：

1、过滤空格：直接跳过所有空格，不影响计算逻辑。

2、数字解析：对于连续数字（如“123”），通过current\_num = current\_num \* 10 + (c - '0')逐步拼接，将字符转换为整数。

3、运算符`+`和`-`：

- 遇到运算符时，先将current\_num按当前sign累加到result中（例如result += sign \* current\_num）。

- 重置current\_num为0，更新`sign`为新运算符对应的符号（`+`对应1，`-`对应-1）。

4、左括号`(`：

- 将当前的result和sign入栈（保存括号外的计算状态）。

- 重置result为0、`sign`为1，开始计算括号内的子表达式。

5、右括号`)`：

- 先将括号内的最后一个数字累加到result中（完成子表达式计算）。

- 弹出栈顶的sign（括号前的符号），将子表达式结果乘以该符号。

- 弹出栈顶的result（括号前的结果），与上一步结果累加，得到包含括号的新结果。

6、最终结果：表达式解析结束后，需将最后一个未处理的current\_num累加到result中，得到最终答案。

复杂度分析

- 时间复杂度：O(n)，其中n是表达式的长度。每个字符仅被遍历一次，栈的操作（入栈、出栈）均为O(1)，整体为线性时间。

- 空间复杂度：O(n)。最坏情况下，表达式中全是嵌套括号（如`((((1+2)))))`），栈的深度为O(n)，因此空间复杂度为O(n)。