欢迎大家参与本项目，贡献其他语言版本的代码，拥抱开源，让更多学习算法的小伙伴们收益！

> 这不仅仅是一道好题，也展现出计算机的思考方式

150. 逆波兰表达式求值

https://leetcode-cn.com/problems/evaluate-reverse-polish-notation/

根据 逆波兰表示法，求表达式的值。

有效的运算符包括 +，-，*，/。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

整数除法只保留整数部分。 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1： 输入: ["2", "1", "+", "3", " * "] 输出: 9 解释: 该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9

示例 2： 输入: ["4", "13", "5", "/", "+"] 输出: 6 解释: 该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6

示例 3： 输入: ["10", "6", "9", "3", "+", "-11", " * ", "/", " * ", "17", "+", "5", "+"] 输出: 22 解释: 该算式转化为常见的中缀算术表达式为： ((10 * (6 / ((9 + 3) * -11))) + 17) + 5 = ((10 * (6 / (12 * -11))) + 17) + 5 = ((10 * (6 / -132)) + 17) + 5 = ((10 * 0) + 17) + 5 = (0 + 17) + 5 = 17 + 5 = 22

逆波兰表达式：是一种后缀表达式，所谓后缀就是指算符写在后面。

平常使用的算式则是一种中缀表达式，如（1 + 2）*（3 + 4）。

该算式的逆波兰表达式写法为（（1 2 +）（3 4 +）*）。

逆波兰表达式主要有以下两个优点：

- 去掉括号后表达式无歧义，上式即便写成1 2 + 3 4 + * 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

思路

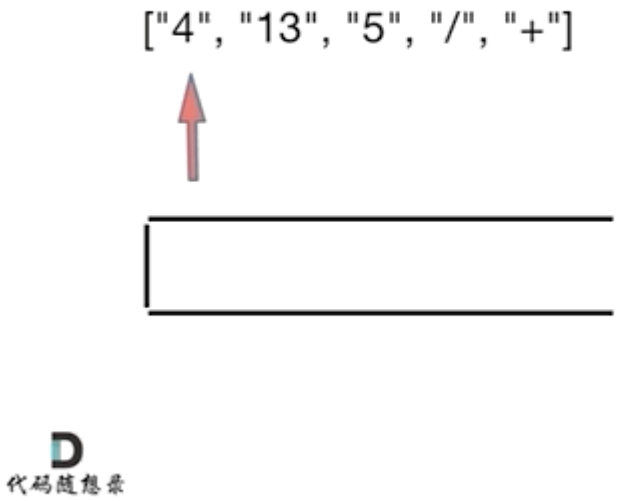
在上一篇文章中栈与队列：匹配问题都是栈的强项提到了 递归就是用栈来实现的。

所以**栈与递归之间在某种程度上是可以转换的！**这一点我们在后续讲解二叉树的时候，会更详细的讲解到。

那么来看一下本题，**其实逆波兰表达式相当于是二叉树中的后序遍历**。 大家可以把运算符作为中间节点，按照后序遍历的规则画出一个二叉树。

但我们没有必要从二叉树的角度去解决这个问题，只要知道逆波兰表达式是用后续遍历的方式把二叉树序列化了，就可以了。

在进一步看，本题中每一个子表达式要得出一个结果，然后拿这个结果再进行运算，那么**这岂不就是一个相邻字符串消除的过程，和栈与队列：匹配问题都是栈的强项中的对对碰游戏是不是就非常像了。**



如动画所示：

相信看完动画大家应该知道，这和1047. 删除字符串中的所有相邻重复项是差不错的，只不过本题不要相邻元素做消除了，而是做运算！

C++代码如下：

```cpp
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> st;
        for (int i = 0; i < tokens.size(); i++) {
            if (tokens[i] == "+" || tokens[i] == "-" || tokens[i] == "*" || tokens[i] == "/
                int num1 = st.top();
                st.pop();
                int num2 = st.top();
                st.pop();
                if (tokens[i] == "+") st.push(num2 + num1);
                if (tokens[i] == "-") st.push(num2 - num1);
                if (tokens[i] == "*") st.push(num2 * num1);
                if (tokens[i] == "/") st.push(num2 / num1);
            } else {
                st.push(stoi(tokens[i]));
            }
        }
        int result = st.top();
        st.pop(); // 把栈里最后一个元素弹出（其实不弹出也没事）
        return result;
    }
};
```

function template

std::stoi <sti

```
int stoi (const string&  str, size_t* idx = 0, int base = 10);
int stoi (const wstring& str, size_t* idx = 0, int base = 10);
```

Convert string to integer

Parses *str* interpreting its content as an integral number of the specified *base*, which is returned as an `int` value.

If *idx* is not a null pointer, the function also sets the value of *idx* to the position of the first character in *str* after the number.

The function uses strtol (or wcstol) to perform the conversion (see strtol for more details on the process).

题外话

我们习惯看到的表达式都是中缀表达式，因为符合我们的习惯，但是中缀表达式对于计算机来说就不是很友好了。

例如：4 + 13 / 5，这就是中缀表达式，计算机从左到右去扫描的话，扫到13，还要判断13后面是什么运算法，还要比较一下优先级，然后13还和后面的5做运算，做完运算之后，还要向前回退到 4 的位置，继续做加法，你说麻不麻烦！

那么将中缀表达式，转化为后缀表达式之后：["4", "13", "5", "/", "+"] ，就不一样了，计算机可以利用栈里顺序处理，不需要考虑优先级了。也不用回退了， **所以后缀表达式对计算机来说是非常友好的。**

可以说本题不仅仅是一道好题，也展现出计算机的思考方式。

在1970年代和1980年代，惠普在其所有台式和手持式计算器中都使用了RPN（后缀表达式），直到2020年代仍在某些模型中使用了RPN。

参考维基百科如下：

> During the 1970s and 1980s, Hewlett-Packard used RPN in all of their desktop and hand-held calculators, and continued to use it in some models into the 2020s.

其他语言版本

java:

```java
public class EvalRPN {

    public int evalRPN(String[] tokens) {
        Deque<Integer> stack = new LinkedList();
        for (String token : tokens) {
            char c = token.charAt(0);
            if (!isOpe(token)) {
                stack.addFirst(stoi(token));
            } else if (c == '+') {
                stack.push(stack.pop() + stack.pop());
            } else if (c == '-') {
                stack.push(- stack.pop() + stack.pop());
            } else if (c == '*') {
                stack.push( stack.pop() * stack.pop());
            } else {
                int num1 = stack.pop();
                int num2 = stack.pop();
                stack.push( num2/num1);
            }
        }
        return stack.pop();
    }


    private boolean isOpe(String s) {
        return s.length() == 1 && s.charAt(0) <'0' || s.charAt(0) >'9';
    }

    private int stoi(String s) {
        return Integer.valueOf(s);
    }


    public static void main(String[] args) {
        new EvalRPN().evalRPN(new String[] {"10","6","9","3","+","-11","*","/","*","17","+"
    }

}
```

Go:

```go
func evalRPN(tokens []string) int {
        stack := []int{}
        for _, token := range tokens {
                val, err := strconv.Atoi(token)
                if err == nil {
                        stack = append(stack, val)
                } else {
                        num1, num2 := stack[len(stack)-2], stack[(len(stack))-1]
                        stack = stack[:len(stack)-2]
                        switch token {
                        case "+":
                                stack = append(stack, num1+num2)
                        case "-":
                                stack = append(stack, num1-num2)
```

```
                        case "*":
                                stack = append(stack, num1*num2)
                        case "/":
                                stack = append(stack, num1/num2)
                        }
                }
        }
        return stack[0]
}
```

javaScript:

```
/**
 * @param {string[]} tokens
 * @return {number}
 */
var evalRPN = function(tokens) {
    const s = new Map([
        ["+", (a, b) => a * 1  + b * 1],
        ["-", (a, b) => b - a],
        ["*", (a, b) => b * a],
        ["/", (a, b) => (b / a) | 0]
    ]);
    const stack = [];
    for (const i of tokens) {
        if(!s.has(i)) {
            stack.push(i);
            continue;
        }
        stack.push(s.get(i)(stack.pop(),stack.pop()))
    }
    return stack.pop();
};
```

python3

```
def evalRPN(tokens) -> int:
    stack = list()
    for i in range(len(tokens)):
        if tokens[i] not in ["+", "-", "*", "/"]:
            stack.append(tokens[i])
        else:
            tmp1 = stack.pop()
            tmp2 = stack.pop()
            res = eval(tmp2+tokens[i]+tmp1)
            stack.append(str(int(res)))
    return stack[-1]
```
