# 中缀表达式转换为后缀表达式

首先需要说明一下，什么是中缀表达式和后缀表达式。以一个例子为

- 1+((2+3)*4)-5，这是中缀表达式，就是运算符在两个运算数中间
- 1 2 3 + 4 * + 5 -，这是后缀表达式，就是运算符在两个运算符后面。

为什么要转成后缀表达式呢？是因为后缀表达式对于计算机来说，更容易计算。

中缀表达式转换为后缀表达式的计算步骤如下：

- 1，初始化两个栈，运算符栈s1和中间结果的栈s2；
- 2，从左往右扫描中缀表达式；
- 3，遇到操作数时，将其压入s2；
- 4，遇到运算符是，比较其与s1栈顶运算符的优先级：
    - 4.1 如果s1为空，或栈顶运算符为左括号“（”，则直接将此运算符入栈；
    - 4.2 否则，如果优先级比栈顶运算符的高，也将运算符压入s1；
    - 4.3 否则，将s1栈顶的运算符弹出并压入到s2中，再次转到4.1中与新的栈顶运算符相比较。
- 5，遇到括号时
    - 5.1 如果是左括号“（”，则直接压入s1；
    - 5.2 如果是右括号“）”，则依次弹出s1栈顶的运算符，并压入s2，直到遇到左括号位置，此时，将这一对括号丢弃
- 6，重复步骤2至5，知道表达式的最右边
- 7，将s1中剩余的运算符依次弹出并压入s2
- 8，依次弹出s2中的运算符并输出，结果的逆序即为中缀表达式对应的后缀表达式。

举例说明：

将中缀表达式“1+（（2 + 3）* 4）- 5转换为后缀表达式的过程如下：

因此结果为：1 2 3 + 4 * + 5 -，手动推导的步骤如下：

- 第一次扫描到的元素是1，根据步骤3，是操作数，压入s2栈
    - s2： 1
    - s1 ：空
- 第2次扫描到的元素是+ 根据步骤4.1，符号栈s1为空，将+压入符号栈
    - s2:1
    - s1:+
- 第3次扫描到的元素是（ ，根据步骤5.1，直接将左括号压入s1栈；
    - s2:1
    - s1:+ (
- 第4次扫描到的元素是（ ，根据步骤5.1，直接将左括号压入s1栈；
    - s2:1
    - s2:+ （ （
- 第5次扫描到的元素是2，根据步骤3，将2压入s2栈；
    - s2:1 2
    - s1： + （ （
- 第6次扫描到的元素是+ ，根据4.1，因为栈顶元素是左括号(，直接将+压入s1栈；
    - s2:1 2
    - s1: + ( ( +

- 第7次扫描到的元素是3，根据步骤3，将3压入s2栈；
    - s2:1 2 3
    - s1: + ( ( +
- 第8次扫描到的元素是 ），根据5.2，依次弹出s1栈顶的运算符，并压入到s2中，直到遇到左括号为止，此时将这一对括号丢弃。所以，将+从s1弹出，压入s2，丢弃最后入栈的左括号。
    - s2:1 2 3 +
    - s1: + (
- 第9次扫描到的元素是* ，根据步骤4.1，栈顶元素是左括号，直接将* 压入s1栈；
    - s2: 1 2 3 +
    - s1： + （ *
- 第10次扫描到的元素是4，根据步骤3，直接将4压入s2栈；
    - s2 : 1 2 3 + 4
    - s1: + ( *
- 第11次扫描到的元素是 ），根据5.2，依次弹出栈顶的运算符，并压入s2中，直到遇到左括号位置，此时将这一对括号丢弃，所以，将* 从s1弹出，压入s2，丢弃最后入栈的左括号。
    - s2 ： 1 2 3 + 4 *
    - s1: +
- 第12次扫描到的元素是-，根据4.3，因为-和+的优先级相同，将s1的栈顶元素弹出，压入s2中，此时s1为空，根据4.1，将-压入到s1中。
    - s2: 1 2 3 + 4 * +
    - s1: -
- 第13次扫描的元素是5，根据步骤3，直接压入到s2中
    - s2: 1 2 3 + 4 * + 5
    - s1： -
- 已经扫描完毕，根据步骤7，将-从s1弹出，并压入到s2中；
    - s2: 1 2 3 + 4 * + 5 -
- 根据步骤8，依次弹出s2中的所有元素，逆序的结果就是对应的后缀表达式
    - 弹出结果：- 5 + * 4 + 3 2 1
    - 逆序结果： 1 2 3 + 4 * + 5 -

## 解决代码

```
import java.util.Stack;
/*
 * 1，初始化两个栈，运算符栈s1和中间结果的栈s2；
 * 2，从左往右扫描中缀表达式；
 * 3，遇到操作数时，将其压入s2；
 * 4，遇到运算符时，比较其与s1栈顶运算符的优先级：
 * 4.1 如果s1为空，或栈顶运算符为左括号“（”，则直接将此运算符入栈；
 * 4.2 否则，如果优先级比栈顶运算符的高，也将运算符压入s1；
 * 4.3 否则，将s1栈顶的运算符弹出并压入到s2中，再次转到4.1中与新的栈顶运算符相比较。
 * 5，遇到括号时
 * 5.1 如果是左括号“（”，则直接压入s1；
 * 5.2 如果是右括号“）”，则依次弹出s1栈顶的运算符，并压入s2，直到遇到左括号位置，此时，将这一对括号丢弃
 * 6，重复步骤2至5，知道表达式的最右边
 * 7，将s1中剩余的运算符依次弹出并压入s2
 * 8，依次弹出s2中的运算符并输出，结果的逆序即为中缀表达式对应的后缀表达式。

 */
public class houzhui {
    public  static  void  main(String[] args) {
        String str = "1+((2+3)*4)-5";
```

```
        System.out.println(parseSuffiexExpressionList(str));
    }
    public static String parseSuffiexExpressionList(String str) {
//          * 1，初始化两个栈，运算符栈s1和中间结果的栈s2；
        Stack<Character> s1 = new Stack<Character>();
        Stack<Character> s2 = new Stack<Character>();
//        2，从左往右扫描中缀表达式；
        for(int i=0;i<str.length();i++){
            char ch = str.charAt(i);
//            如果是数字，直接加入s2
            if(ch>='0' && ch<='9'){
                s2.push(ch);
//                如果是左括号，直接加入s2
            }else if(ch=='('){
                s1.push(ch);
//                如果是右括号，将元素从s1中弹出，压入s2，直至s1的栈顶元素是左括号
            }else if(ch==')'){
                while(s1.peek()!='('){
                    s2.push(s1.pop());
                }
//                弹出s1的栈顶元素即左括号
                s1.pop();
            }else{
//                如果ch的优先级小于s1栈顶元素优先级，将s1栈顶的运算符弹出并添加到s2中，再次转到4.1与新的
栈顶元素比较
                while(!s1.isEmpty()&&(priority(s1.peek())>=priority(ch))){
                    s2.push(s1.pop());
                }
//                还需要将ch压入栈
                s1.push(ch);
            }
        }
//将s1的剩余元素依次弹出压入s2
        while(!s1.isEmpty()){
            s2.add(s1.pop());
        }
//        s2逆序一次
        while(!s2.isEmpty()){
            s1.push(s2.pop());
        }
//        将s2逆袭结果依次输出，组成字符串
        StringBuilder sbu = new StringBuilder();
        while(!s1.isEmpty()){
            sbu.append(s1.pop());
        }
        return sbu.toString();
    }


    public  static int priority(char ch){
        if(ch=='*'||ch=='/'){
            return 1;
        }
        if(ch=='+'||ch=='-'){
            return 0;
        }
        return  -1;
    }
}
```

# 根据后缀表达式计算表达式的值

前面提到，后缀表达式易于计算，这里对后缀表达的计算过程进行说明。

思路如下（以逆波兰表达式3 4 + 5 * 6 -为例，该表达式对应的中缀表达式是（3+4）* 5-6，易知计算结果是29：

- 1,从左到右扫描，将3和4压入栈
- 2，遇到+运算符，弹出4和3，计算出来3+4，得到7压入栈
- 将5入栈
- 接下来是* 运算符，弹出5和7，计算出来5*7，得到35压入栈
- 将6入栈，
- 最后是-运算符，计算出来35-6，得到29，就是最终结果

因为上面已经根据实例进行了说明，这里不再一步一步进行说明。

# 解决代码

```
public static String calculate(String str) {
        Stack<String> stack = new Stack<String>();
        for (int i = 0; i < str.length(); i++) {
            String ch = str.charAt(i) + "";
            if (ch.matches("\\d+")) {
                stack.push(ch);
            } else {
                int num1 = Integer.parseInt(stack.pop());
                int num2 = Integer.parseInt(stack.pop());
                int res = cal(num1, num2, ch);
                stack.push(res + "");
            }
        }
        return stack.pop();
    }
//        System.out.println(parseSuffiexExpressionList(str));

    public static int cal(int num1,int num2,String op){
        int res;
        if(op.equals("+")){
            res = num1+num2;
        }else if(op.equals("-")){
            res = num2-num1;
        }else if(op.equals("*")){
            res = num1*num2;
        }else{
            res = num2/num1;
        }
        return res;
    }
```

# 完整的计算代码如下：

```
import java.util.List;
import java.util.Stack;
/*
 * 1，初始化两个栈，运算符栈s1和中间结果的栈s2；
 * 2，从左往右扫描中缀表达式；
 * 3，遇到操作数时，将其压入s2；
 * 4，遇到运算符时，比较其与s1栈顶运算符的优先级：
 * 4.1 如果s1为空，或栈顶运算符为左括号“（”，则直接将此运算符入栈；
```

```
 * 4.2 否则，如果优先级比栈顶运算符的高，也将运算符压入s1；
 * 4.3 否则，将s1栈顶的运算符弹出并压入到s2中，再次转到4.1中与新的栈顶运算符相比较。
 * 5，遇到括号时
 * 5.1 如果是左括号“（”，则直接压入s1；
 * 5.2 如果是右括号“）”，则依次弹出s1栈顶的运算符，并压入s2，直到遇到左括号位置，此时，将这一对括号丢弃
 * 6，重复步骤2至5，知道表达式的最右边
 * 7，将s1中剩余的运算符依次弹出并压入s2
 * 8，依次弹出s2中的运算符并输出，结果的逆序即为中缀表达式对应的后缀表达式。

 */
public class houzhui {
    public  static  void  main(String[] args) {
        String str = "1+((2+3)*4)-5";
        String str1 = parseSuffiexExpressionList(str);
        String str2 = calculate(str1);
        System.out.println(str);
        System.out.println(str1);
        System.out.println(str2);
    }

    public static String calculate(String str) {
        Stack<String> stack = new Stack<String>();
        for (int i = 0; i < str.length(); i++) {
            String ch = str.charAt(i) + "";
            if (ch.matches("\\d+")) {
                stack.push(ch);
            } else {
                int num1 = Integer.parseInt(stack.pop());
                int num2 = Integer.parseInt(stack.pop());
                int res = cal(num1, num2, ch);
                stack.push(res + "");
            }
        }
        return stack.pop();
    }
//        System.out.println(parseSuffiexExpressionList(str));

    public static int cal(int num1,int num2,String op){
        int res;
        if(op.equals("+")){
            res = num1+num2;
        }else if(op.equals("-")){
            res = num2-num1;
        }else if(op.equals("*")){
            res = num1*num2;
        }else{
            res = num2/num1;
        }
        return res;
    }
    public static String parseSuffiexExpressionList(String str) {
//          * 1，初始化两个栈，运算符栈s1和中间结果的栈s2；
        Stack<Character> s1 = new Stack<Character>();
        Stack<Character> s2 = new Stack<Character>();
//        2，从左往右扫描中缀表达式；
        for(int i=0;i<str.length();i++){
            char ch = str.charAt(i);
//            如果是数字，直接加入s2
            if(ch>='0' && ch<='9'){
                s2.push(ch);
```

```
//                  如果是左括号，直接加入s2
            }else if(ch=='('){
                s1.push(ch);
//                  如果是右括号，将元素从s1中弹出，压入s2，直至s1的栈顶元素是左括号
            }else if(ch==')'){
                while(s1.peek()!='('){
                    s2.push(s1.pop());
                }
//                  弹出s1的栈顶元素即左括号
                s1.pop();
            }else{
//                  如果ch的优先级小于s1栈顶元素优先级，将s1栈顶的运算符弹出并添加到s2中，再次转到4.1与新的
栈顶元素比较
                while(!s1.isEmpty()&&(priority(s1.peek())>=priority(ch))){
                    s2.push(s1.pop());
                }
//                  还需要将ch压入栈
                s1.push(ch);
            }
        }
//将s1的剩余元素依次弹出压入s2
        while(!s1.isEmpty()){
            s2.add(s1.pop());
        }
//          s2逆序一次
        while(!s2.isEmpty()){
            s1.push(s2.pop());
        }
//          将s2逆袭结果依次输出，组成字符串
        StringBuilder sbu = new StringBuilder();
        while(!s1.isEmpty()){
            sbu.append(s1.pop());
        }
        return sbu.toString();
    }


    public  static int priority(char ch){
        if(ch=='*'||ch=='/'){
            return 1;
        }
        if(ch=='+'||ch=='-'){
            return 0;
        }
        return  -1;
    }
}
```

## 练习题

## leetcode 224 基本计算器

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式可以包含左括号 （ ，右括号 ），加号 + ，减号 -，非负整数和空格 。

示例 1:

```
输入: "1 + 1"
输出: 2
```

## 解决代码：

```
import java.util.Stack;
class Solution {

    public int calculate(String string){
        StringBuilder sbu = new StringBuilder();
        for(int i=0;i<string.length();i++){
            if(string.charAt(i)!=' '){
                sbu.append(string.charAt(i));
            }
        }
        String str1 = parseSuffiexExpressionList(sbu.toString());
        String str2 = calculateByhouzhui(str1);
        return Integer.parseInt(str2);
    }

    public static String calculateByhouzhui(String str) {
        Stack<String> stack = new Stack<String>();
        for (int i = 0; i < str.length(); i++) {
            String ch = str.charAt(i) + "";
            if (ch.matches("\\d+")) {
                stack.push(ch);
            } else {
                int num1 = Integer.parseInt(stack.pop());
                int num2 = Integer.parseInt(stack.pop());
                int res = cal(num1, num2, ch);
                stack.push(res + "");
            }
        }
        return stack.pop();
    }

    public static int cal(int num1,int num2,String op){
        int res;
        if(op.equals("+")){
            res = num1+num2;
        }else if(op.equals("-")){
            res = num2-num1;
        }else if(op.equals("*")){
            res = num1*num2;
        }else{
            res = num2/num1;
        }
        return res;
    }
    public static String parseSuffiexExpressionList(String str) {
//          * 1，初始化两个栈，运算符栈s1和中间结果的栈s2；
        Stack<Character> s1 = new Stack<Character>();
        Stack<Character> s2 = new Stack<Character>();
//        2，从左往右扫描中缀表达式；
```

```
        for(int i=0;i<str.length();i++){
            char ch = str.charAt(i);
//              如果是数字，直接加入s2
            if(ch>='0' && ch<='9'){
                s2.push(ch);
//                  如果是左括号，直接加入s2
            }else if(ch=='('){
                s1.push(ch);
//                  如果是右括号，将元素从s1中弹出，压入s2，直至s1的栈顶元素是左括号
            }else if(ch==')'){
                while(s1.peek()!='('){
                    s2.push(s1.pop());
                }
//                  弹出s1的栈顶元素即左括号
                s1.pop();
            }else{
//                  如果ch的优先级小于s1栈顶元素优先级，将s1栈顶的运算符弹出并添加到s2中，再次转到4.1与新的
栈顶元素比较
                while(!s1.isEmpty()&&(priority(s1.peek())>=priority(ch))){
                    s2.push(s1.pop());
                }
//                  还需要将ch压入栈
                s1.push(ch);
            }
        }
//将s1的剩余元素依次弹出压入s2
        while(!s1.isEmpty()){
            s2.add(s1.pop());
        }
//          s2逆序一次
        while(!s2.isEmpty()){
            s1.push(s2.pop());
        }
//          将s2逆袭结果依次输出，组成字符串
        StringBuilder sbu = new StringBuilder();
        while(!s1.isEmpty()){
            sbu.append(s1.pop());
        }
        return sbu.toString();
    }


    public  static int priority(char ch){
        if(ch=='*'||ch=='/'){
            return 1;
        }
        if(ch=='+'||ch=='-'){
            return 0;
        }
        return  -1;
    }
}
```

## leetcode 150逆波兰表达式求值

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。

```
输入: ["2", "1", "+", "3", "*"]
输出: 9
解释: ((2 + 1) * 3) = 9
```

## 解决代码

```
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();
        for (int i = 0 ;i < tokens.length;i++){
            String str = tokens[i];
            if (str.length() == 1){
                char ch = str.charAt(0);
                if (ch-'0' >= 0 && ch - '0' <= 9 ){
                    Integer a = Integer.valueOf(str);
                    stack.push(a);
                }
                else{
                    if (stack.size() < 2)
                        return 0;
                    int num2 = stack.pop();
                    int num1 = stack.pop();
                    switch(ch){
                        case '+':
                            stack.push(num1 + num2);
                            break;
                        case '-':
                            stack.push(num1 - num2);
                            break;
                        case '*':
                            stack.push(num1 * num2);
                            break;
                        case '/':
                            stack.push(num1 / num2);
                            break;
                    }

                }
            }else{
                int n = Integer.valueOf(str);
                stack.push(n);
            }
        }
        return stack.pop();
    }
}
```