

表达式计算

1. 从控制台中，读取要计算的表达式，并可以适当的对其进行预处理，比如可以提前将 16 进制的数字转化为十进制的数字，便于后面计算。要确保化简后的中缀表达式只有 () , + , - , * , \ 运算符。
2. 计算一个表达式的值，需要将我们常规状态中缀表达式修改为后缀表达式。
3. 碰到单元运算符我们要将其 () 里面的表达式，重复上面 1 2 操作，
4. 计算化简后的后缀表达式。

表达式计算的具体实现

需要注意的核心就是，high low 和移位是单元运算符，+ - * / 是二元运算符，两种不同类型的符号需要进行不同的处理。

首先对于只有二元运算符的式子来说，我们需要将中缀表达式变成逆波兰表达式。变成逆波兰表达式后就可以进行计算了。

```
public class InfixToSuffix {  
    public String[] translation(String[] expression){...}  
  
    public double calculateSuffix(String[] expression){...}  
}
```

InfixToSuffix 类中主要有两个方法

方法一 translation 是将表达式变成逆波兰表达式：

使用 map 设置运算符的优先级

```
int index = 0; // 后续表达式的下标  
Deque<String> stack = new LinkedList<>();  
HashMap<String, Integer> map = new HashMap(){  
    {  
        put("+", 0);  
        put("-", 0);  
        put("*", 1);  
        put("/", 1);  
        put("(", -100);  
        put(")", -100);  
    }  
};
```

根据二元符号的优先级进行转变

```

if(expression[i].equals("+") || expression[i].equals("-") || expression[i].equals("*")){
    if(stack.size() == 0){ //当栈为空时，直接将计算符号压入栈中
        stack.push(expression[i]);
    }else if(map.get(expression[i]) <= map.get(stack.peek())){
        ret[index++] = expression[i]; // 当运算符优先级较高时，直接将运算符放入ret数组
    }else{
        stack.push(expression[i]);
    }
}

}else if(expression[i].equals("(") || expression[i].equals(")")){
    if(expression[i].equals("(")){
        stack.push(expression[i]);
    }else if(expression[i].equals(")")){ // 将栈中括号内的运算符都弹出
        while(stack.peek() != "("){
            ret[index++] = stack.pop();
        }
        stack.pop(); // 将 ( 弹出，后缀表达式中可以没有 (
    }
}

}else { // 当是一个数字时，就直接放入ret数组

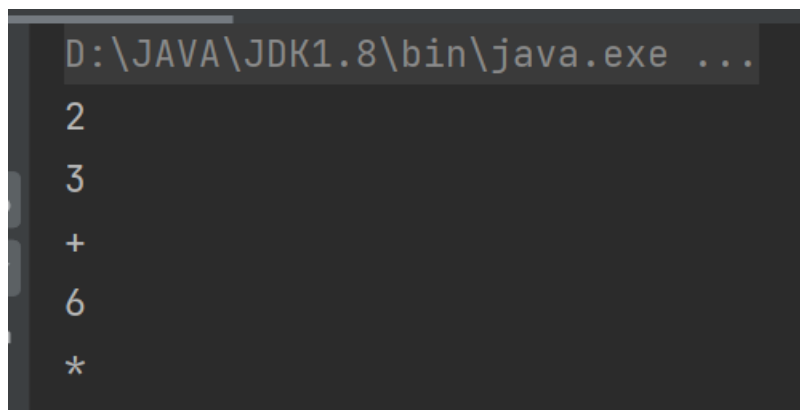
```

简单测试结果：

输入 (2 + 3) * 6

```
String[] expression = new String[]{"(", "2", "+", "3", ")", "*", "6"}; // 2 + 3 * 6
```

其输出结果就是



```

D:\JAVA\JDK1.8\bin\java.exe ...
2
3
+
6
*

```

2 3 + 6 *

同样自己在多次输入测试结果，比较输出，发现本方法能正确将转换。

方法二 calculateSuffix 是计算逆波兰表达式：

比较简单，数据结构课已经学习过，利用栈即可实现。

```

public double calculateSuffix(String[] expression){
    int len = expression.length;
    Deque<Double> stack = new LinkedList<>();

    for (int i = 0; i < len; i++) {
        if(stack.size() == 0){
            stack.push(Double.parseDouble(expression[i]));
        }else if(expression[i].equals("+")){
            double a = stack.pop();
            double b = stack.pop();
            double c = a + b;
            stack.push(c);
        }else if(expression[i].equals("-")){
            double a = stack.pop();
            double b = stack.pop();
            double c = b - a;
            stack.push(c);
        }
    }
}

```

注意将输出设置为 double，注意精度即可。

对于单元运算符来说，我们需要单独处理单元运算符的式子，比如 high (2 + 3 * 4) 这个式子，我们需要用上一步双元运算符的规则将括号里的值计算出来，然后在进行 high 运算。简而言之，就是我们需要将一个同时有双元运算符和单元运算符的表达式化简为一个只有双元运算符的式子。

可以利用正则表达式来找到括号里的内容

```

public class FindBracket { // 找到括号里的表达式
    public String findExpressionInBracket(String expression){
        Matcher mat = Pattern.compile("(?<=\\ (\\S+)(?=\\ )").matcher(expression);
        System.out.println(mat.group());
        return mat.group();
    }
}

```

在主函数中一步一步将复杂表达式转换成逆波兰表达式，最后计算出答案即可。