1.介绍和定义

1.介绍

1)栈的英文为(stack)

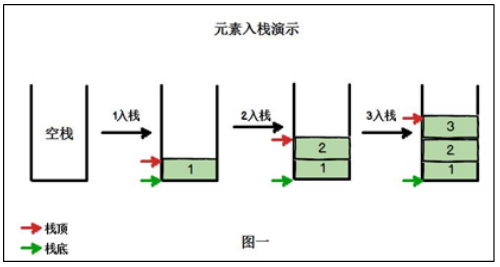
2)栈是一个**先入后出**(FILO-First In Last Out)的有序列表。

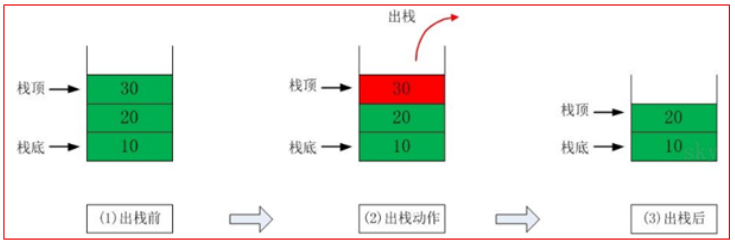
3)栈(stack)是限制线性表中元素的插入和删除只能在线性表的同一端进行的一种特殊线性表。允许插入和删除的一端，为变化的一端，称为栈顶(Top)，另一端为固定的一端，称为栈底(Bottom)。

4)根据栈的定义可知，最先放入栈中元素在栈底，最后放入的元素在栈顶，而删除元素刚好相反，最后放入的元素最先删除，最先放入的元素最后删除.

2.图示

入栈(pop) /出栈（push）





2.栈的应用场景

1)子程序的调用：在跳往子程序前，会先将下个指令的地址存到堆栈中，直到子程序执行完后再将地址取出，以回到原来的程序中。

2)处理递归调用：和子程序的调用类似，只是除了储存下一个指令的地址外，也将参数、区域变量等数据存入堆栈中。

3)表达式的转换[中缀表达式转后缀表达式]与求值(实际解决)。

4)二叉树的遍历。

5)图形的深度优先(depth一first)搜索法。

3.数组模拟栈

几个常用方法:

pop()

peek()

push()

isEmpty()

isFull()

代码实现：

```
/**
 * @author jndeng
 * 数组模拟栈
 * @create 2019-12-16 9:39
 */

public class MyStack {
    private int maxSize;
    private int[] stack;
    int top = -1;

    public MyStack(int maxSize) {
        this.maxSize = maxSize;
        stack = new int[maxSize];
    }

    public void push(int data) {
        if (isFull()) {
            top++;
            stack[top] = data;
        } else {
            throw new RuntimeException("栈已满");
        }
    }

    public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("栈为空");
        } else {
            return stack[top--];
        }
    }

    public int peek(){
        if (isEmpty()) {
            throw new RuntimeException("栈为空");
        } else {
            return stack[top];
        }
    }


    public boolean isEmpty() {
        return top==-1;
```

```
    }

    public boolean isFull() {
        return top < maxSize;
    }

    public void list(){
        if (isEmpty())
        {
            throw new RuntimeException("栈为空");
        }else {
            for (int i=top;i>-1;i--)
            {
                System.out.printf("stack[%d]=%d\n",i,stack[i]);
            }
        }
    }
}
```

4.栈应用

1.栈实现综合计算器(中缀表达式)

实现原理

1,扫描表达式,逐个判断是数字数字或者运算符

1. 是运算符

1. 判断运算符栈是否为空.

1,空的话直接压入栈

2. 不为空,和栈顶的运算符比较优先级

1. 优先级高于栈顶,取出数字栈的2个数字,进行运算,将结果压入数字栈

2. 优先级不高于栈顶,将运算符压入运算符栈

2. 是数字,进行多位数判断,压入数字栈

2. 一次取出2个数字和运算符进行运算,然后将结果压入数字栈

3. 结果是数字栈有唯一元素是结果,运算符栈为空

```
package 线性结构.stack;

/**
 * @author jndeng
 * @create 2019-12-16 9:57
 */
public class StackTest {
```

```
public static void main(String[] args) {
    String ex = "30+2*6-2";
    char[] array = ex.toCharArray();
    MyStack numStack = new MyStack(10);
    MyStack opStack = new MyStack(10);
    int index = 0;
    StringBuilder sb = new StringBuilder();
    while (true) {

        char ch = array[index];
        if (isOp(ch)) {
            if (opStack.isEmpty()) {
                opStack.push(ch);
            } else {
                if (property(ch) > property(opStack.peek())) {
                    opStack.push(ch);
                } else {
                    int num2 = numStack.pop();
                    int num1 = numStack.pop();
                    int ch1 = opStack.pop();
                    int cal = cal(num1, num2, ch1);
                    opStack.push(ch);
                    numStack.push(cal);
                }
            }
        } else {
         //   numStack.push(ch - 48);

            sb.append(ch);
            while (index+1 < ex.length() && !isOp(array[index+1])) {
                sb.append(array[++index]);
            }

            numStack.push(Integer.parseInt(sb.toString()));
            sb.delete(0, sb.length());
        }
        index++;
        if (index == ex.length()) {
            while (true) {
                int num2 = numStack.pop();
                int num1 = numStack.pop();
                int ch1 = opStack.pop();
                int cal = cal(num1, num2, ch1);
                numStack.push(cal);
                if (opStack.isEmpty()) {
                    break;
                }
```

```
            }
            break;
        }
    }

    System.out.println(numStack.pop());


}


public static boolean isOp(char ch) {
    return ch == '/' || ch == '+' || ch == '-' || ch == '*';
}

public static int property(char ch) {
    if (ch == '*' || ch == '/') {
        return 1;
    } else if (ch == '+' || ch == '-') {
        return 0;
    } else {
        throw new RuntimeException("运算符错误");
    }
}

public static int property(int ch) {
    if (ch == '*' || ch == '/') {
        return 1;
    } else if (ch == '+' || ch == '-') {
        return 0;
    } else {
        throw new RuntimeException("运算符错误");
    }
}

public static int cal(int num1, int num2, int ch1) {
    switch (ch1) {
        case '+':
            return (num1 + num2);
        case '-':
            return (num1 - num2);
        case '*':
            return (num1 * num2);
        case '/':
            return (num1 / num2);
        default:
            throw new RuntimeException("运算符错误");
```

```
        }
    }
}
```

加入括号,我的思考是先按照括号进行字符串拆分,先单独对于括号运算,然后在重新拼接表达式进行运算.

5.前缀,中缀,后缀表达式

1.前缀表达式（波兰表达式）

1)前缀表达式又称波兰式，前缀表达式的运算符位于操作数之前

2)举例说明： (3+4)×5-6 对应的前缀表达式就是 - × + 3 4 5 6

前缀表达式的计算机求值

从右至左扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（栈顶元素 和 次顶元素），并将结果入栈；重复上述过程直到表达式最左端，最后运算得出的值即为表达式的结果

例如: (3+4)×5-6 对应的前缀表达式就是 - × + 3 4 5 6 ，针对前缀表达式求值步骤如下:

1)从右至左扫描，将6、5、4、3压入堆栈

2)遇到+运算符，因此弹出3和4（3为栈顶元素，4为次顶元素），计算出3+4的值，得7，再将7入栈

3)接下来是×运算符，因此弹出7和5，计算出7×5=35，将35入栈

4)最后是-运算符，计算出35-6的值，即29，由此得出最终结果

2.中缀表达式

1)中缀表达式就是常见的运算表达式，如(3+4)×5-6

2)中缀表达式的求值是我们人最熟悉的，但是对计算机来说却不好操作(前面我们讲的案例就能看的这个问题)，因此，在计算结果时，往往会将中缀表达式转成其它表达式来操作(一般转成后缀表达式.)

3.后缀表达式（逆波兰表达式）

1)后缀表达式又称逆波兰表达式,与前缀表达式相似，只是运算符位于操作数之后

2)中举例说明： (3+4)×5-6 对应的后缀表达式就是 3 4 + 5 × 6 -

正常的表达式	逆波兰表达式
a+b	a b +
a+(b-c)	a b c - +
a+(b-c)*d	a b c – d * +
a+d*(b-c)	a d b c - * +
a=1+3	a 1 3 + =

4.后缀表达式来实现计算

```
package 线性结构.stack;

import java.util.*;

/**
 * @author jndeng
 * @create 2019-12-16 13:27
 */
public class PolandNotion {

    public static void main(String[] args) {
      String suffixExpression = "4 5 * 8 - 60 + 8 2 / +";
      List<String> list = getListString(suffixExpression);
      System.out.println("rpnList=" + list);
      int res = calculate(list);
      System.out.println("计算的结果是=" + res);
    }
    public static List<String> getListString(String suffixExpression) {
        //将 suffixExpression 分割
        String[] split = suffixExpression.split(" ");
        List<String> list = new ArrayList<String>();
        for(String ele: split) {
            list.add(ele);
        }
        return list;
    }
    public static int calculate(List<String> ls) {
        ArrayDeque<String> stack = new ArrayDeque<String>();
        for (String item : ls) {
            if (item.matches("\\d+")) {
                stack.push(item);
            } else {
                // pop出两个数，并运算， 再入栈
                int num2 = Integer.parseInt(stack.pop());
                int num1 = Integer.parseInt(stack.pop());
                int res = 0;
```

```
            if (item.equals("+")) {
                res = num1 + num2;
            } else if (item.equals("-")) {
                res = num1 - num2;
            } else if (item.equals("*")) {
                res = num1 * num2;
            } else if (item.equals("/")) {
                res = num1 / num2;
            } else {
                throw new RuntimeException("运算符有误");
            }
            //把res 入栈
            stack.push("" + res);
        }
    }
    //最后留在stack中的数据是运算结果
    return Integer.parseInt(stack.pop());
  }
}
```

5.中缀表达式转后缀表达式

思路(要处理括号)

1)初始化两个栈：运算符栈s1和储存中间结果的栈s2；

2)从左至右扫描中缀表达式；

3)遇到操作数时，将其压s2；

4)遇到运算符时，比较其与s1栈顶运算符的优先级：

(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；

(2)否则，若优先级比栈顶运算符的高，也将运算符压入s1；

(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)与s1中新的栈顶运算符相比较；

5)遇到括号时：

(1) 如果是左括号“(”，则直接压入s1

(2) 如果是右括号“)”，则依次弹出s1栈顶的运算符，并压入s2，直到遇到左括号为止，此时将这一对括号丢弃

6)重复步骤2至5，直到表达式的最右边

7)将s1中剩余的运算符依次弹出并压入s2

8)依次弹出s2中的元素并输出，结果的逆序即为中缀表达式对应的后缀表达式

```
package 线性结构.stack;

import java.util.*;
```

```
/**
 * @author jndeng
 * @create 2019-12-16 13:27
 */
public class PolandNotion {

    public static void main(String[] args) {
        String  s="1+((2+3)*4)-5";
        List<String> strings = toInfixExpressionList(s);
        //[1, +, (, (, 2, +, 3, ), *, 4, ), -, 5]
        System.out.println(strings);
        List<String> strings1 = expressionConvert(strings);
        //[1, 2, 3, +, 4, *, +, 5, -]
        System.out.println(strings1);
    }

    public static List<String> expressionConvert(List<String> list) {
        List<String> suffix = new ArrayList<>();
        ArrayDeque<String> op = new ArrayDeque<>();
        ArrayDeque<String> num = new ArrayDeque<>();
        for (String str :
                list) {
            if (str.matches("\\d+")) {
                num.push(str);
            } else {
                if (op.isEmpty() || "(".equals(str) || "(".equals(op.peek())) {
                    op.push(str);
                } else if (")".equals(str)) {
                    while (!"(".equals(op.peek())) {
                        num.push(op.pop());
                    }
                    op.pop();
                } else {
                    if (Operation.getValue(str) > Operation.getValue(op.peek())) {
                        op.push(str);
                    }else {
                        num.push(op.pop());
                        op.push(str);
                    }
                }
            }
        }
        num.push(op.pop());
        while (!num.isEmpty())
        {
            op.push(num.pop());
        }
```

```
    while (!op.isEmpty())
    {
        suffix.add(op.pop());
    }
    return suffix;
}
/**
 * 方法：将 中缀表达式转成对应的List
 *  s="1+((2+3)×4)-5";
 * @param s
 * @return
 */
public static List<String> toInfixExpressionList(String s) {
    //定义一个List,存放中缀表达式 对应的内容
    List<String> ls = new ArrayList<String>();
    int i = 0;
    String str;
    char c;
    do {

        if((c=s.charAt(i)) < 48 ||  (c=s.charAt(i)) > 57) {
            ls.add("" + c);
            i++;
        } else {
            str = "";
            while(i < s.length() && (c=s.charAt(i)) >= 48 && (c=s.charAt(i)) <= 57) {
                str += c;
                i++;
            }
            ls.add(str);
        }
    }while(i < s.length());
    return ls;
}

public static List<String> getListString(String suffixExpression) {
    //将 suffixExpression 分割
    String[] split = suffixExpression.split(" ");
    List<String> list = new ArrayList<String>();
    for (String ele : split) {
        list.add(ele);
    }
    return list;

}

public static int calculate(List<String> ls) {
```

```
        ArrayDeque<String> stack = new ArrayDeque<String>();

        for (String item : ls) {
            //正则表达式判断
            if (item.matches("\\d+")) {

                stack.push(item);
            } else {
                // pop出两个数，并运算， 再入栈
                int num2 = Integer.parseInt(stack.pop());
                int num1 = Integer.parseInt(stack.pop());
                int res = 0;
                if (item.equals("+")) {
                    res = num1 + num2;
                } else if (item.equals("-")) {
                    res = num1 - num2;
                } else if (item.equals("*")) {
                    res = num1 * num2;
                } else if (item.equals("/")) {
                    res = num1 / num2;
                } else {
                    throw new RuntimeException("运算符有误");
                }
                //把res 入栈
                stack.push("" + res);
            }
        }
        //最后留在stack中的数据是运算结果
        return Integer.parseInt(stack.pop());
    }

}

/**
 * 编写一个类 Operation 可以返回一个运算符 对应的优先级
 */
class Operation {
    private static int ADD = 1;
    private static int SUB = 1;
    private static int MUL = 2;
    private static int DIV = 2;

    /**
     *
     * @param operation
     * @return
     */
```

```
    public static int getValue(String operation) {
        int result = 0;
        switch (operation) {
            case "+":
                result = ADD;
                break;
            case "-":
                result = SUB;
                break;
            case "*":
                result = MUL;
                break;
            case "/":
                result = DIV;
                break;
            default:
                System.out.println("不存在该运算符" + operation);
                break;
        }
        return result;
    }
}
```