解释器模式

2020年2月18日　　12:06

1. 概念

定义一个语言，并定义该语言的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。其中"文法"指的是语言的语法规则，“句子”是语言集中的元素，可以用一颗语法树来直观地描述语言中的句子。

为了解释一种语言，而为语言创建的解释器。例如对正则表达式的解释，需要解释器对其进行解释。

2. 扩展

1. 文法

文法是用于描述语言的语法结构的形式规则。例如，中文中的“句子”的文法如下。

〈句子〉::=〈主语〉〈谓语〉〈宾语〉
〈主语〉::=〈代词〉|〈名词〉
〈谓语〉::=〈动词〉
〈宾语〉::=〈代词〉|〈名词〉
〈代词〉你|我|他
〈名词〉大学生I晚霞I英语
〈动词〉::= 是|学习

注：这里的符号“::=”表示“定义为”的意思，用“〈”和“〉”括住的是非终结符，没有括住的是终结符。

2. 句子

句子是语言的基本单位，是语言集中的一个元素，它由终结符构成，能由“文法”推导出。例如，上述文法可以推出“我是大学生”，所以它是句子。

3. 语法树

语法树是句子结构的一种树型表示，它代表了句子的推导结果，它有利于理解句子语法结构的层次。如下所示是“我是大学生”的语法树。

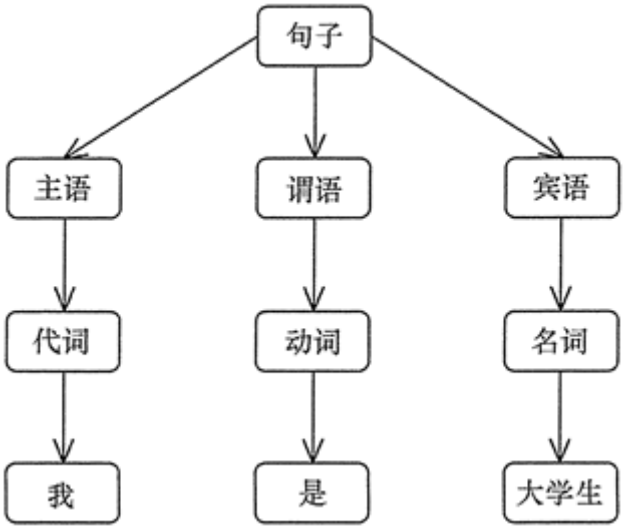

图2.1 语法树

3. UML结构图

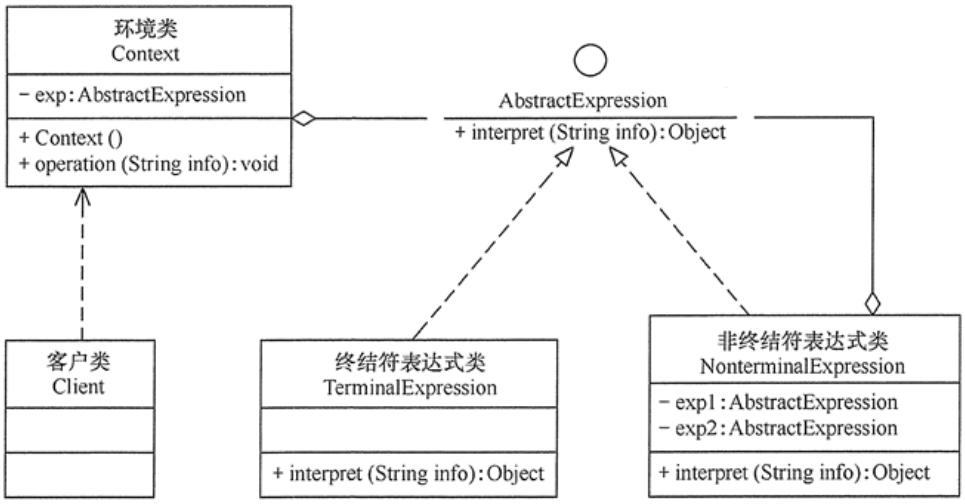

图2.1 解释器模式UML结构图

1. 抽象表达式（Abstract Expression）角色：定义解释器的接口，约定解释器的解释操作，主要包含解释方法 interpret()。
2. 终结符表达式（Terminal　Expression）角色：是抽象表达式的子类，用来实现文法中与终结符相关的操作，文法中的每一个终结符都有一个具体终结表达式与之相对应。
3. 非终结符表达式（Nonterminal Expression）角色：也是抽象表达式的子类，用来实现文法中与非终结符相关的操作，文法中的每条规则都对应于一个非终结符表达式。
4. 环境（Context）角色：通常包含各个解释器需要的数据或是公共的功能，一般用来传递被所有解释器共享的数据，后面的解释器可以从这里获取这些值。
5. 客户端（Client）：主要任务是将需要分析的句子或表达式转换成使用解释器对象描述的抽象语法树，然后调用解释器的解释方法，当然也可以通过环境角色间接访问解释器的解释方法。

4. 解释器模式实例

对于形如"6 100 11 + *"的字符串，定义若干解释器，将该字符串按照一定的文法规则进行计算。

〈Expression〉::=〈NumberExpression〉〈OperationExpression〉
〈NumberExpression〉::= 6 | 100 | 11
〈OperationExpression〉::= + | *

1. 抽象表示式

```
public interface Interpreter {
    int interpret();
}
```

2. 终结表达式

```
public class NumberInterpreter implements Interpreter {
    private int number;

    public NumberInterpreter(int number) {
        this.number = number;
    }
    public NumberInterpreter(String number){
        this.number = Integer.parseInt(number);
    }

    @Override
    public int interpret() {
        return this.number;
    }
```

```
}
```

3. 非终结表达式

```
public class AddInterpreter implements Interpreter{
    private Interpreter firstExpression,secondExpression;

    public AddInterpreter(Interpreter firstExpression, Interpreter secondExpression) {
        this.firstExpression = firstExpression;
        this.secondExpression = secondExpression;
    }

    @Override
    public int interpret() {
        return this.firstExpression.interpret() + secondExpression.interpret();
    }

    @Override
    public String toString() {
        return "+";
    }
}
```

```
public class MultiInterpreter implements Interpreter {
    private Interpreter firstExpression,secondExpression;

    public MultiInterpreter(Interpreter firstExpression, Interpreter secondExpression) {
        this.firstExpression = firstExpression;
        this.secondExpression = secondExpression;
    }

    @Override
    public int interpret() {
        return this.firstExpression.interpret() * secondExpression.interpret();
    }

    @Override
    public String toString() {
        return "*";
    }
}
```

4. 环境

```
public class ExpressionParser {
    private Stack<Interpreter> stack = new Stack<Interpreter>();

    public int parse(String str){
        String[] strItemArray = str.split(" ");
        for (String symbol : strItemArray){
            if (!OperatorUtil.isOperator(symbol)){
                Interpreter numberExpression = new NumberInterpreter(symbol);
                stack.push(numberExpression);
                System.out.println(String.format("入栈: %d",numberExpression.interpret()));
            }else{
                // 是运算符号就进行计算
                Interpreter fistExpression = stack.pop();
                Interpreter secondExpression = stack.pop();
                System.out.println(String.format("出栈: %d, %
d",fistExpression.interpret(),secondExpression.interpret()));
                Interpreter operator =
OperatorUtil.getExpressionObject(fistExpression,secondExpression,symbol);
                System.out.println(String.format("应用运算符: %s",symbol));
                int result = operator.interpret();
                // 结果入栈
                NumberInterpreter resultExpression = new NumberInterpreter(result);
                stack.push(resultExpression);
                System.out.println(String.format("阶段结果入栈: %d",result));
            }
        }
        int result = stack.pop().interpret();
        return result;
    }
}
```

5. 工具类

```
public class OperatorUtil {
    // 判断是否时操作符
    public static boolean isOperator(String symbol){
        return (symbol.equals("+") || symbol.equals("*"));
    }

    // 返回解释对象
    public static Interpreter getExpressionObject(Interpreter firstExpression,
                                                  Interpreter secondExpreesion,String symbol){
        if (symbol.equals("+")){
            return new AddInterpreter(firstExpression,secondExpreesion);
        }else if (symbol.equals("*")){
```

```
            return new MultiInterpreter(firstExpression,secondExpreesion);
        }
        return null;
    }
}
```

6. 客户端

```
  @Test
    public void interpreterTest(){
        String strInput = "6 100 11 + *";
        HBExpressionParser hbExpressionParser = new HBExpressionParser();
        int result = hbExpressionParser.parse(strInput);
        System.out.println("解释器最终结果: " + result);
    }
```

7. 结果

```
入栈: 6
入栈: 100
入栈: 11
出栈: 11, 100
应用运算符: +
阶段结果入栈: 111
出栈: 111, 6
应用运算符: *
阶段结果入栈: 666
解释器最终结果: 666
```

5. 优/缺点

1. 优点

- 语法由很多类表示，容易改变及扩展此“语言”。
- 容易实现，在语法树上的每个表达式节点类都是相似的，所以实现其文法比较容易。

2. 缺点

- 当语法规则数目太多时，增加了系统的复杂度。
- 执行效率低。解释器模式中通常使用大量的循环和递归调用，当要解释的句子较复杂时，运行速度慢且代码调试麻烦。
- 可应用场景较少。

6. 适用场景

4. 语言的文法较为简单，且执行效率不是关键问题时。
5. 当问题重复出现，且可以用一种简单的语言来进行表达时。
6. 当一个语言需要解释执行，并且语言中的句子可以表示为一个抽象语法树的时候，如 XML 文档解释。
7. 解释器模式在实际的软件开发中使用比较少，因为它会引起效率、性能以及维护等问题。JAVA已经拥有与很多强大的公式解析器：Expression4J、MESP(Math Expression String Parser) 和 Jep 等，它们可以解释一些复杂的文法，功能强大，使用简单。

7. 参考资料

[1] http://c.biancheng.net/view/1402.html