定义

给分析对象定义一个语言，定义它的文法的一种表示，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。

为了解释一种语言，而为语言创建的解释器

适用场景

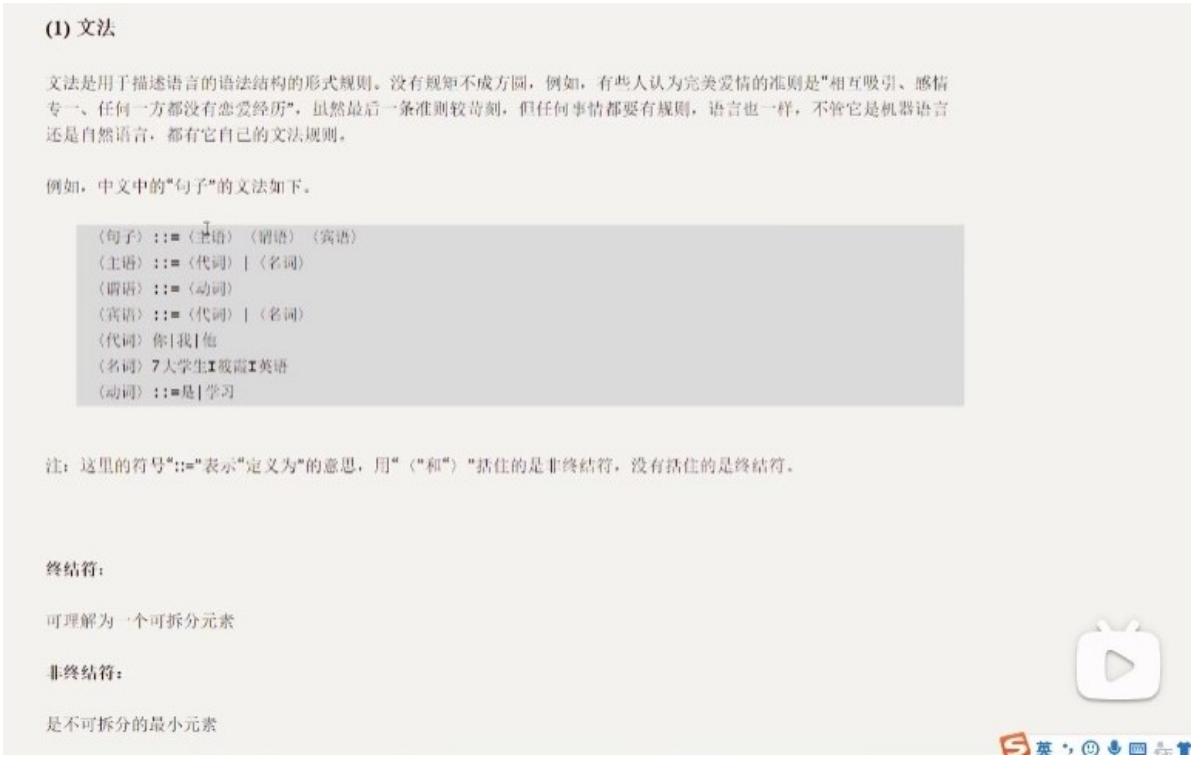
某个特定类型问题发生频率足够高

优点

语法由很多类表示，容易改变及拓展此”语言“

缺点

当语法规则数目太多时，增加了相同的复杂度

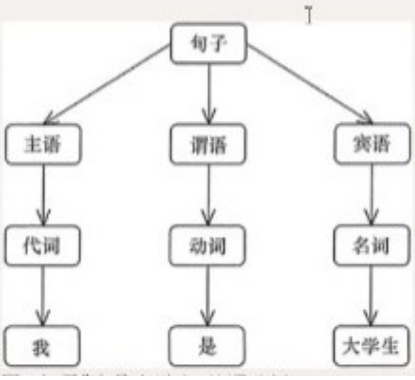


(2) 句子

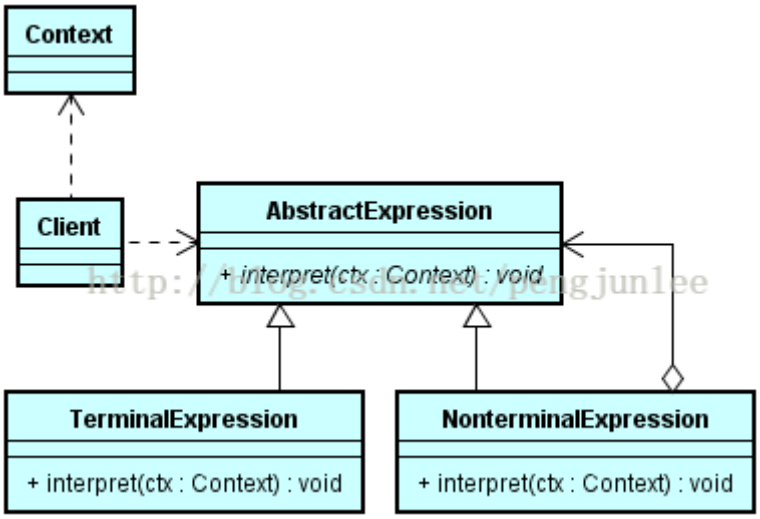
句子是语言的基本单位，是语言集中的一个元素，它由终结符构成，能由"文法"推导出。例如，上述文法可以推出"我是大学生"，所以它是句子。

(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。图 1 所示是"我是大学生"的语法树。



结构



解释器模式涉及的角色及其职责如下：

抽象表达式(AbstractExpression)角色：约定解释器的解释操作，主要是一个interpret()方法。

终结符表达式(TerminalExpression)角色：用来实现文法中和终结符相关的解释操作，不再包含其它的解释器

非终结符表达式(NonterminalExpression)角色：用来实现文法中和非终结符相关的解释操作，通常一个解释器对应一个语法规则，可以包含其它的解释器

环境(Context)角色：也称"上下文"，常用HashMap来代替，通常包含解释器之外的一些全局信息（解释器需要的数据，或是公共的功能）。

客户端(Client)角色：构建文法表示的抽象语法树（Abstract Syntax Tree，该抽象语法树由终结符表达式和非终结符表达式的实例装配而成），并调用解释操作interpret()方法。

解释器模式结构示意源代码如下：

首先定义一个抽象表达式(AbstractExpression)角色，并在其中定义一个执行解释操作的接口，示例代码如下。

```
/**
 * 抽象表达式
 */
   public abstract class AbstractExpression {

   /**

    * 解释的操作
    * @param ctx 上下文对象
      */
      public abstract void interpret(Context ctx);

}
```

抽象表达式(AbstractExpression) 的具体实现分两种：终结符表达式(TerminalExpression)和非终结符表达式

```
/**
 * 终结符表达式
 */
public class TerminalExpression extends AbstractExpression {

    @Override
    public void interpret(Context ctx) {
        // 实现与语法规则中的终结符相关联的解释操作

    }

}
```

```
/**
 * 非终结符表达式
 */
public class NonterminalExpression extends AbstractExpression {

    @Override
    public void interpret(Context ctx) {
        // 实现与语法规则中的非终结符相关联的解释操作

    }

}
```

再来看看环境(Context)的定义，示例代码如下。

```
/**
 * 上下文，包含解释器之外的一些全局信息
 */
public class Context {

}
```

最后来看看客户端的定义，示例代码如下。

```
/**
 * 使用解释器的客户
 */
public class Client {
    /**
     * 主要按照语法规格对特定的句子构建抽象语法树
     * 然后调用解释操作
     */

}
```

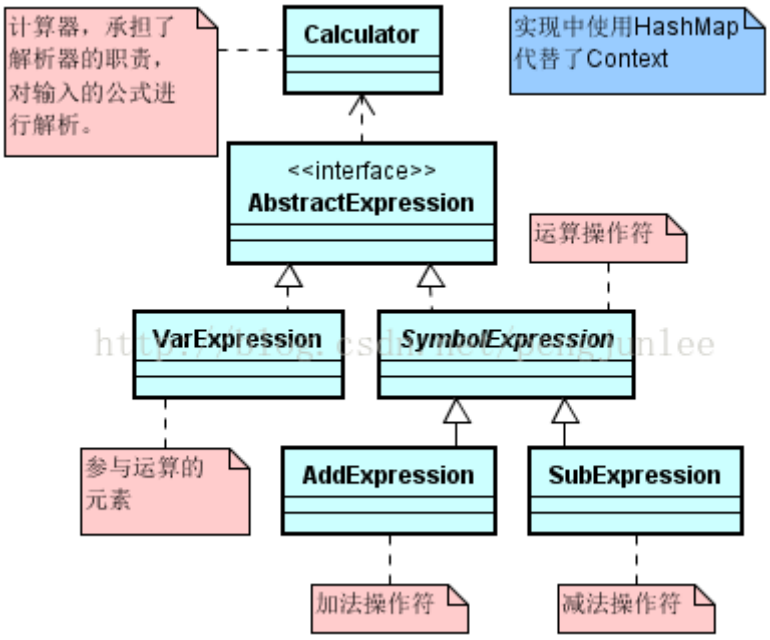
以上示例代码很简单， 只是为了说明解释器模式实现的基本结构和各个角色的功能， 实际的解释逻辑并未写出。

解释器模式应用举例

假如我们要实现这么一个功能：加减法计算器。

计算器中参与运算的元素为a、b、c、d等...，这些元素都与具体的业务相关，我们不细追究其所代表的意义。现要实现：客户端指定一个公式，如“a+b-c”，“a+b+c-d”，计算器就可以计算出结果。

设计好的类图结构示意如下：



源代码如下。

```
import java.util.HashMap;

/**
 * 抽象表达式,声明解释操作
 */
public interface AbstractExpression {

    // 每个表达式都必须有一个解释操作
    public int interprete(HashMap<String, Integer> var);

}
```

```
import java.util.HashMap;
```

```
/**
 * 终结符表达式,代表参加运算的元素对象
 */
public class VarExpression implements AbstractExpression {

    private String key;

    public VarExpression(String key) {
        this.key = key;
    }

    public int interprete(HashMap<String, Integer> var) {
        return (Integer) var.get(this.key);
    }
}
```

```
/**
 * 非终结符表达式,运算符（此处为加法和减法）的抽象父类,真正的解释操作由其子类来实现
 */
public abstract class SymbolExpression implements AbstractExpression {

    protected AbstractExpression left;
    protected AbstractExpression right;

    // 非终结符表达式的解释操作只关心自己左右两个表达式的结果
    public SymbolExpression(AbstractExpression left, AbstractExpression right) {
        this.left = left;
        this.right = right;
    }

}
```

```
import java.util.HashMap;

/**
 * 加法表达式
 */
public class AddExpression extends SymbolExpression {

    public AddExpression(AbstractExpression left, AbstractExpression right) {
        super(left, right);
    }

    // 把左右两个表达式运算的结果加起来
    public int interprete(HashMap<String, Integer> var) {
        return super.left.interprete(var) + super.right.interprete(var);
    }
}
```

```
import java.util.HashMap;

/**
 * 减法表达式
 */
```

```
public class SubExpression extends SymbolExpression {

    public SubExpression(AbstractExpression left, AbstractExpression right) {
        super(left, right);
    }

    // 左右两个表达式相减
    public int interprete(HashMap<String, Integer> var) {
        return super.left.interprete(var) - super.right.interprete(var);
    }
}
```

```
import java.util.HashMap;
import java.util.Stack;

public class Calculator {

    private AbstractExpression expression;

    /**
     * 对公式进行解析操作
     *
     * @param expStr
     *             输入的公式
     */
    public Calculator(String expStr) {
        // 定义一个堆栈，安排运算的先后顺序
        Stack<AbstractExpression> stack = new Stack<AbstractExpression>();
        // 表达式拆分为字符数组
        char[] charArray = expStr.toCharArray();
        // 运算
        AbstractExpression left = null;
        AbstractExpression right = null;
        for (int i = 0; i < charArray.length; i++) {
            switch (charArray[i]) {
            case '+': // 加法
                left = stack.pop();
                right = new VarExpression(String.valueOf(charArray[++i]));
                stack.push(new AddExpression(left, right));
                break;

            case '-': // 减法
                left = stack.pop();
                right = new VarExpression(String.valueOf(charArray[++i]));
                stack.push(new SubExpression(left, right));
                break;
            default: // 公式中的变量
                stack.push(new VarExpression(String.valueOf(charArray[i])));
            }
        }
        // 把运算结果抛出来
        this.expression = stack.pop();
    }

    // 计算结果
    public int calculate(HashMap<String, Integer> var) {
        return this.expression.interprete(var);
```

```
    }
}
```

```
import java.util.HashMap;

public class Client {
    public static void main(String[] args) {
        // 构造运算元素的值列表
        HashMap<String, Integer> ctx = new HashMap<String, Integer>();
        ctx.put("a", 10);
        ctx.put("b", 20);
        ctx.put("c", 30);
        ctx.put("d", 40);
        ctx.put("e", 50);
        ctx.put("f", 60);
        Calculator calc = new Calculator("a+b-c");
        int result = calc.calculate(ctx);
        System.out.println("Result of a+b-c: " + result);
        calc = new Calculator("d-a-b+c");
        result = calc.calculate(ctx);
        System.out.println("Result of d-a-b+c: " + result);
    }
}
```

运行程序打印结果如下：

```
Result of a+b-c: 0
Result of d-a-b+c: 40
```

该加减法计算器示例与解释器模式组成元素对照如下：

①给定一种语言， 本例中就是一个简单的加减运算。

②定义一种文法表示，本例中就是指定的参与运算的元素(abcdef)以及运算符(+-)，以及由它们构造而成的公式，如 d-a-b+c。

③给定一个解释器来解释语言中的句子：本例中的解释器是多个类的组合,包括Calculator和AbstractExpression 。

④TerminalExpression表示终结符表达式，相当于本例中的VarExpression。

⑤NonterminalExpression是非终结符表达式，相当于本例中的加法、减法。