课程目标

1、掌握解释器模式的应用场景。

2、掌握解析表达式的基本原理。

3、理解解释器模式的优缺点。

内容定位

适合参与软件框架设计开发的人群。

解释器模式

解释器模式（Interpreter Pattern）是指给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。是一种按照规定的语法（文法）进行解析的模式，属于行为型模式。

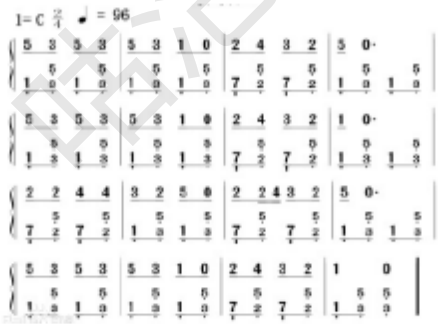
```
原文：Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
```

就比如编译器可以将源码编译解释为机器码，让 CPU 能进行识别并运行。解释器模式的作用其实与编译器一样，都是将一些固定的文法（即语法）进行解释，构建出一个解释句子的解释器。简单理解，解释器是一个简单语法分析工具，它可以识别句子语义，分离终结符号和非终结符号，提取出需要的信息，让我们能针对不同的信息做出相应的处理。其核心思想是识别文法，构建解释。

解释器模式的应用场景

其实我们每天都生活在解释器模式中，我们平时所听到的音乐都可以通过简谱记录下来；还

有战争年代发明的摩尔斯电码（又译为摩斯密码，Morse code），其实也是一种解释器。



音乐简谱

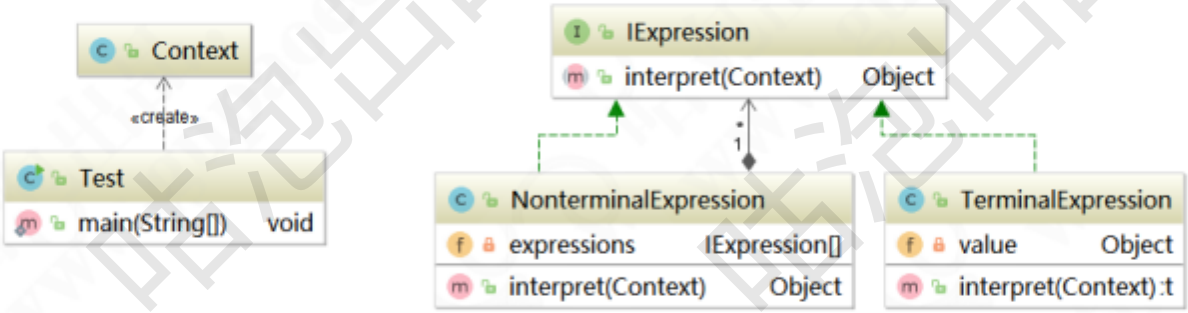
摩斯密码

我们的程序中，如果存在一种特定类型的问题，该类型问题涉及多个不同实例，但是具备固定文法描述，那么可以使用解释器模式对该类型问题进行解释，分离出需要的信息，根据获取的信息做出相应的处理。简而言之，对于一些固定文法构建一个解释句子的解释器。解释器模式适用于以下应用场景：

1、一些重复出现的问题可以用一种简单的语言来进行表达；

2、一个简单语法需要解释的场景。

首先来看下解释器模式的通用 UML 类图：



从 UML 类图中，我们可以看到，解释器模式 主要包含四种角色：

抽象表达式（Expression）：负责定义一个解释方法 interpret，交由具体子类进行具体解释；

终结符表达式（TerminalExpression）：实现文法中与终结符有关的解释操作。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，比如公式 R=R1+R2，R1 和 R2 就是终结符，

对应的解析 R1 和 R2 的解释器就是终结符表达式。通常一个解释器模式中只有一个终结符表达式，但有多个实例，对应不同的终结符（R1，R2）；

非终结符表达式（NonterminalExpression）：实现文法中与非终结符有关的解释操作。文法中的每条规则都对应于一个非终结符表达式。非终结符表达式一般是文法中的运算符或者其他关键字，比如公式 R=R1+R2 中，“+"就是非终结符，解析“+”的解释器就是一个非终结符表达式。非终结符表达式根据逻辑的复杂程度而增加，原则上每个文法规则都对应一个非终结符表达式；

上下文环境类（Context）：包含解释器之外的全局信息。它的任务一般是用来存放文法中各个终结符所对应的具体值，比如 R=R1+R2，给 R1 赋值 100，给 R2 赋值 200，这些信息需要存放到环境中。

使用解释器模式解析数学表达式

下面我们用解释器模式来实现一个数学表达式计算器，包含加减乘除运算。

首先定义抽象表达式角色 IArithmeticInterpreter 接口：

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

创建终结表达式角色 Interpreter 抽象类：

```
public abstract class Interpreter implements IArithmeticInterpreter {

    protected IArithmeticInterpreter left;
    protected IArithmeticInterpreter right;

    public Interpreter(IArithmeticInterpreter left, IArithmeticInterpreter right) {
        this.left = left;
        this.right = right;
    }
}
```

分别创建非终结表达式角色加、减、乘、除解释器，加法运算表达式 AddInterpreter 类：

```
public class AddInterpreter extends Interpreter {

    public AddInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right) {
        super(left, right);
    }

    public int interpret() {
```

```
        return this.left.interpret() + this.right.interpret();
    }
}
```

减法运算表达式 SubInterpreter 类：

```
public class SubInterpreter extends Interpreter {
    public SubInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right) {
        super(left, right);
    }

    public int interpret() {
        return this.left.interpret() - this.right.interpret();
    }
}
```

乘法运算表达式 MultiInterpreter 类：

```
public class MultiInterpreter extends Interpreter {

    public MultiInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right){
        super(left,right);
    }

    public int interpret() {
        return this.left.interpret() * this.right.interpret();
    }

}
```

除法运算表达式 DivInterpreter 类：

```
public class DivInterpreter extends Interpreter {

    public DivInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right){
        super(left,right);
    }

    public int interpret() {
        return this.left.interpret() / this.right.interpret();
    }

}
```

数字表达式 NumInterpreter 类：

```
public class NumInterpreter implements IArithmeticInterpreter {
    private int value;

    public NumInterpreter(int value) {
        this.value = value;
    }


    public int interpret() {
        return this.value;
    }
}
```

创建计算器 GPCalculator 类：

```
public class GPCalculator {
    private Stack<IArithmeticInterpreter> stack = new Stack<IArithmeticInterpreter>();
```

```
    public GPCalculator(String expression) {
        this.parse(expression);
    }

    private void parse(String expression) {
        String [] elements = expression.split(" ");
        IArithmeticInterpreter left,right;

        for (int i = 0; i < elements.length ; i++) {
            String operator = elements[i];
            if(OperatorUtil.ifOperator(operator)){
                left = this.stack.pop();
                right = new NumInterpreter(Integer.valueOf(elements[++i]));
                System.out.println("出栈" + left.interpret() + "和" + right.interpret());
                this.stack.push(OperatorUtil.getInterpreter(left,right,operator));
                System.out.println("应用运算符：" + operator);
            }else {
                NumInterpreter numInterpreter = new NumInterpreter(Integer.valueOf(elements[i]));
                this.stack.push(numInterpreter);
                System.out.println("入栈：" + numInterpreter.interpret());
            }

        }
    }

    public int calculate() {
        return this.stack.pop().interpret();
    }
}
```

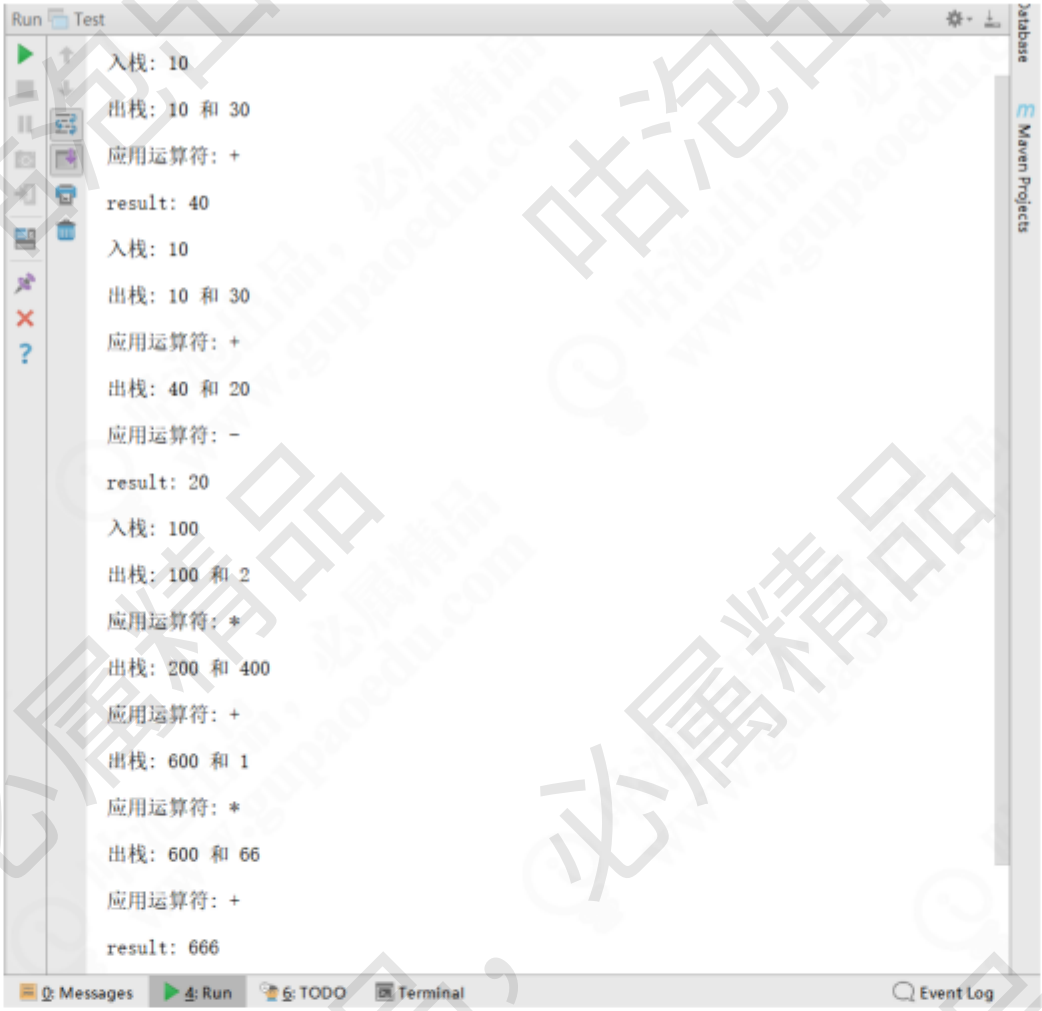
工具类 OperatorUtil 具体代码：

```
public class OperatorUtil {

    public static boolean isOperator(String symbol) {
        return (symbol.equals("+") || symbol.equals("-") || symbol.equals("*"));
    }

    public static Interpreter getInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right, String symbol)
{
        if (symbol.equals("+")) {
            return new AddInterpreter(left, right);
        } else if (symbol.equals("-")) {
            return new SubInterpreter(left, right);
        } else if (symbol.equals("*")) {
            return new MultiInterpreter(left, right);
        } else if (symbol.equals("/")) {
            return new DivInterpreter(left, right);
        }
        return null;
    }
}
```

编写客户端代码：

```
    public static void main(String[] args) {
        System.out.println("result: " + new GPCalculator("10 + 30").calculate());
        System.out.println("result: " + new GPCalculator("10 + 30 - 20").calculate());
        System.out.println("result: " + new GPCalculator("100 * 2 + 400 * 1 + 66").calculate());
    }
```

运行结果如下：



解释器模式在源码中的体现

JDK 源码中的 Pattern 对正则表达式的编译和解析。

```
public final class Pattern implements java.io.Serializable {
    ...
    private Pattern(String p, int f) {
        pattern = p;
        flags = f;

        if ((flags & UNICODE_CHARACTER_CLASS) != 0)
            flags |= UNICODE_CASE;

        // Reset group index count
        capturingGroupCount = 1;
        localCount = 0;

        if (pattern.length() > 0) {
            compile();
        } else {
            root = new Start(lastAccept);
            matchRoot = lastAccept;
        }
    }
    ...
    public static Pattern compile(String regex) {
```

```
        return new Pattern(regex, 0);
    }
    public static Pattern compile(String regex, int flags) {
        return new Pattern(regex, flags);
    }

    ...
}
```

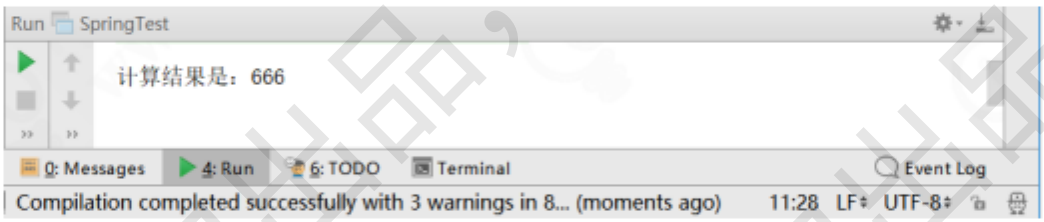
再来看 Spring 中的 ExpressionParser 接口。

```
public interface ExpressionParser {

    Expression parseExpression(String expressionString) throws ParseException;

    Expression parseExpression(String expressionString, ParserContext context) throws ParseException;

}
```

这里的源码我们不深入讲解，通过前面我们自己编写的案例大致能够清楚其原理。我们不妨来编写一段客户端代码验证一下功能。我们编写如下测试代码：

```
    public static void main(String[] args) {
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression = parser.parseExpression("100 * 2 + 400 * 1 + 66");
        int result = (Integer) expression.getValue();
        System.out.println("计算结果是：" + result);
    }
```

其运行结果是：



和我们所期望的是一致的。

解释器模式的优缺点

优点：

1、扩展性强：在解释器模式中由于语法是由很多类表示的，当语法规则更改时，只需修改相应的非终结符表达式即可；若扩展语法时，只需添加相应非终结符类即可；

2、增加了新的解释表达式的方式；

3、易于实现文法：解释器模式对应的文法应当是比较简单且易于实现的，过于复杂的语法并不适合使用解释器模式。

缺点

1、语法规则较复杂时，会引起类膨胀：解释器模式每个语法都要产生一个非终结符表达式，当语法规则比较复杂时，就会产生大量的解释类，增加系统维护困难；

2、执行效率比较低：解释器模式采用递归调用方法，每个非终结符表达式只关心与自己有关的表达式，每个表达式需要知道最终的结果，因此完整表达式的最终结果是通过从后往前递归调用的方式获取得到。当完整表达式层级较深时，解释效率下降，且出错时调试困难，因为递归迭代层级太深。