

课程目标

内容定位

1. 中介者模式

- 1.1. 中介者模式的应用场景
- 1.2. 使用中介者模式设计群聊场景
- 1.3. 中介者模式在源码中的体现
- 1.4. 中介者模式的优缺点

2. 解释器模式

- 2.1. 解释器模式的应用场景
- 2.2. 使用解释器模式解析数学表达式
- 2.3. 解释器模式在源码中的体现
- 2.4. 解释器模式的优缺点

课程目标

- 1、掌握中介者模式和解释器模式的应用场景。
- 2、了解设计群聊的底层逻辑。
- 3、掌握解析表达式的基本原理。
- 4、理解中介者模式和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。

1. 中介者模式

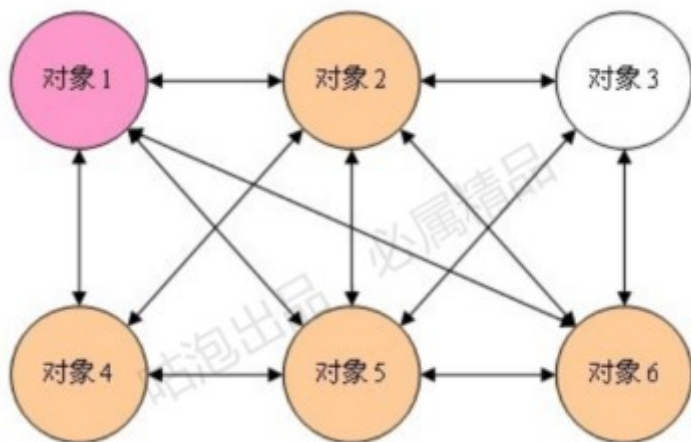
中介者模式 (Mediator Pattern) 又称为调解者模式或调停者模式。用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。属于行为型模式。

原文： Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly , and it lets you vary their interaction independently.

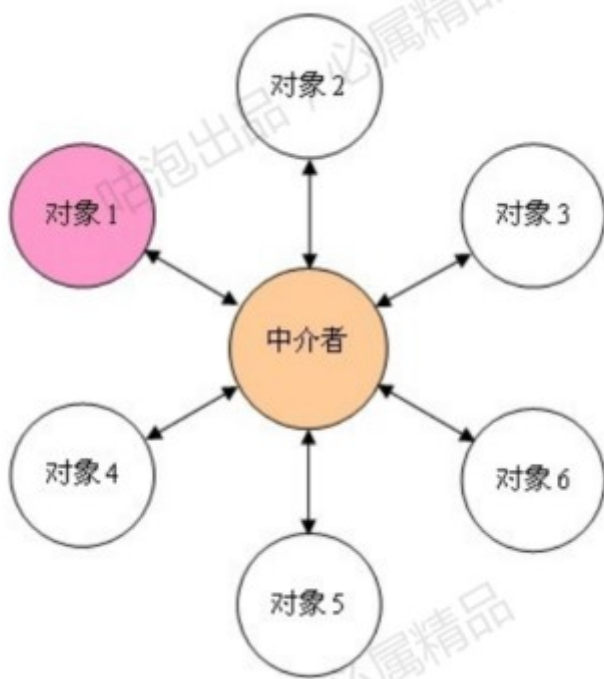
中介者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使它们可以松散耦合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。其核心思想是，通过中介者解耦系统各层次对象的直接耦合，层次对象的对外依赖通信统统交由中介者转发。

1.1. 中介者模式的应用场景

在现实生活中，中介者的存在是不可缺少的，如果没有了中介者，我们就不能与远方的朋友进行交流了。各个同事对象将会相互进行引用，如果每个对象都与多个对象进行交互时，将会形成如下图所示的网状结构。

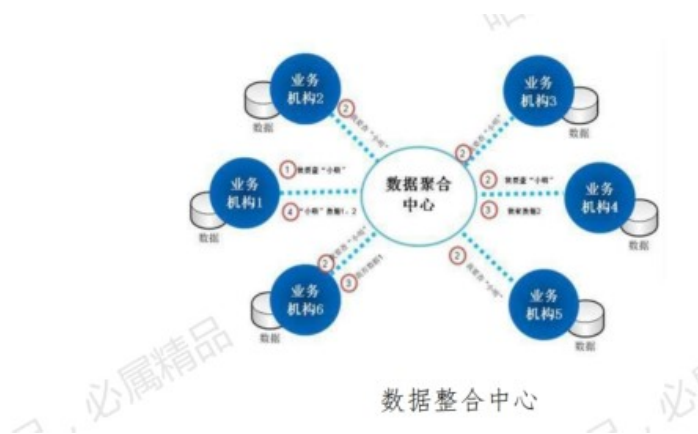


从上图可以发现，每个对象之间过度耦合，这样的既不利于信息的复用也不利于扩展。如果引入了中介者模式，那么对象之间的关系将变成星型结构，采用中介者模式之后会形成如下图所示的结构：



从上图可以发现，使用中介者模式之后，任何一个类的变化，只会影响中介者和类本身，不像之前的设计，任何一个类的变化都会引起其关联所有类的变化。这样的设计大大减少了系统的耦合度。

其实我们日常生活中每天在刷的朋友圈，就是一个中介者。还有我们所见的信息交易平台，也是中介者模式的体现。



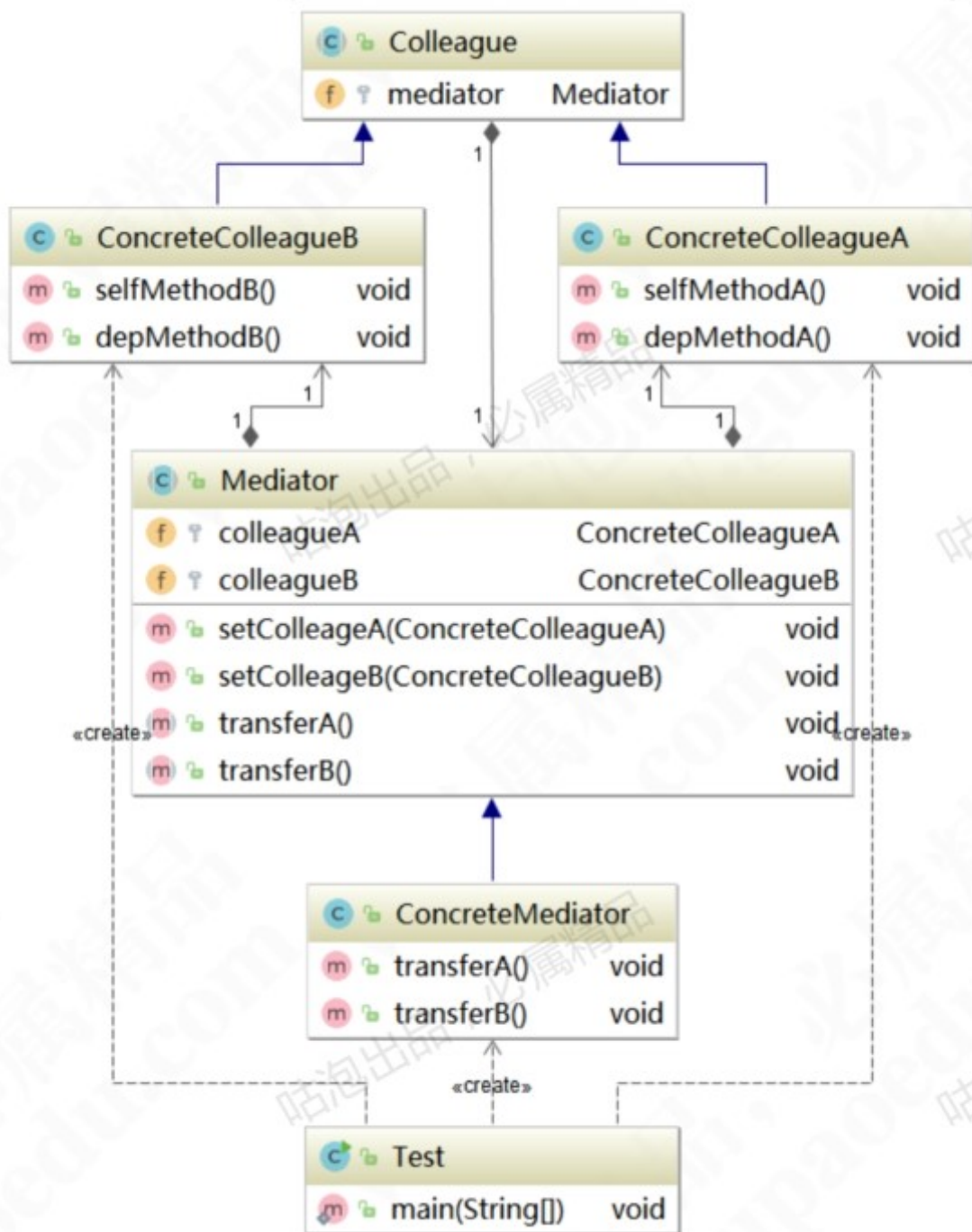
中介者模式是用来降低多个对象和类之间的通信复杂性。这种模式通过提供一个中介类，将系统各层次对象间的多对多关系变成一对多关系，中介者对象可以将复杂的网状结构变成以调停者为中心的星形结构，达到降低系统的复杂性，提高可扩展性的作用。

若系统各层次对象之间存在大量的关联关系，即层次对象呈复杂的网状结构，如果直接让它们紧耦合通信，会造成系统结构变得异常复杂，且其中某个层次对象发生改变，则与其紧耦合的相应层次对象也需进行修改，系统很难进行维护。而通过为该系统增加一个中介者层次对象，让其他各层次需对外通信的行为统统交由中介者进行转发，系统呈现以中介者为中心进行通讯的星形结构，系统的复杂性大大降低。

简单的说就是多个类相互耦合，形成了网状结构，则考虑使用中介者模式进行优化。总结中介者模式适用于以下场景：

- 1、系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解；
- 2、交互的公共行为，如果需要改变行为则可以增加新的中介者类。

首先来看下中介者模式的通用UML类图：



从UML类图中，我们可以看到，中介者模式主要包含4个角色：

抽象中介者（Mediator）：定义统一的接口，用于各同事角色之间的通信；

具体中介者（ConcreteMediator）：从具体的同事对象接收消息，向具体同事对象发出命令，协调各同事间的协作；

抽象同事类（Colleague）：每一个同事对象均需要依赖中介者角色，与其他同事间通信时，交由中介者进行转发协作；

具体同事类（ConcreteColleague）：负责实现自发行为（Self-Method），转发依赖方法（Dep-Method）交由中介者进行协调。

1.2. 使用中介者模式设计群聊场景

假设我们要构建一个聊天室系统，用户可以向聊天室发送消息，聊天室会向所有的用户显示消息。实际上就是用户发信息与聊天室显示的通信过程，不过用户无法直接将信息发给聊天室，而是需要将信

息先到服务器上，然后服务器再将该消息发给聊天室进行显示。具体代码如下。

创建 User类：

```
1 package cn.sitedev.chatroom;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
8 @AllArgsConstructor
9 @NoArgsConstructor
10 public class User {
11     private String name;
12     private ChatRoom chatRoom;
13
14     public void sendMsg(String msg) {
15         this.chatRoom.showMsg(this, msg);
16     }
17 }
```

创建 ChatRoom类：

```
1 package cn.sitedev.chatroom;
2
3 public class ChatRoom {
4     public void showMsg(User user, String msg) {
5         System.out.println "[" + user.getName() + "]: " + msg);
6     }
7 }
```

编写客户端测试代码：

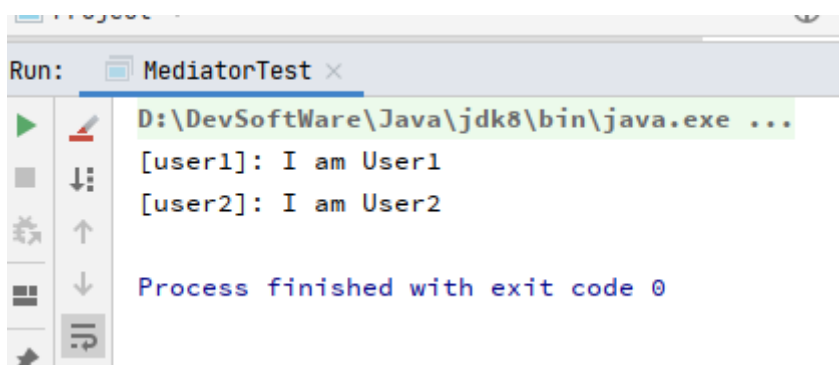
```
1 package cn.sitedev.chatroom;
2
3 public class MediatorTest {
4     public static void main(String[] args) {
5         ChatRoom chatRoom = new ChatRoom();
```

```

6
7     User user = new User("user1", chatRoom);
8     User user2 = new User("user2", chatRoom);
9
10    user.sendMsg("I am User1");
11    user2.sendMsg("I am User2");
12 }
13 }

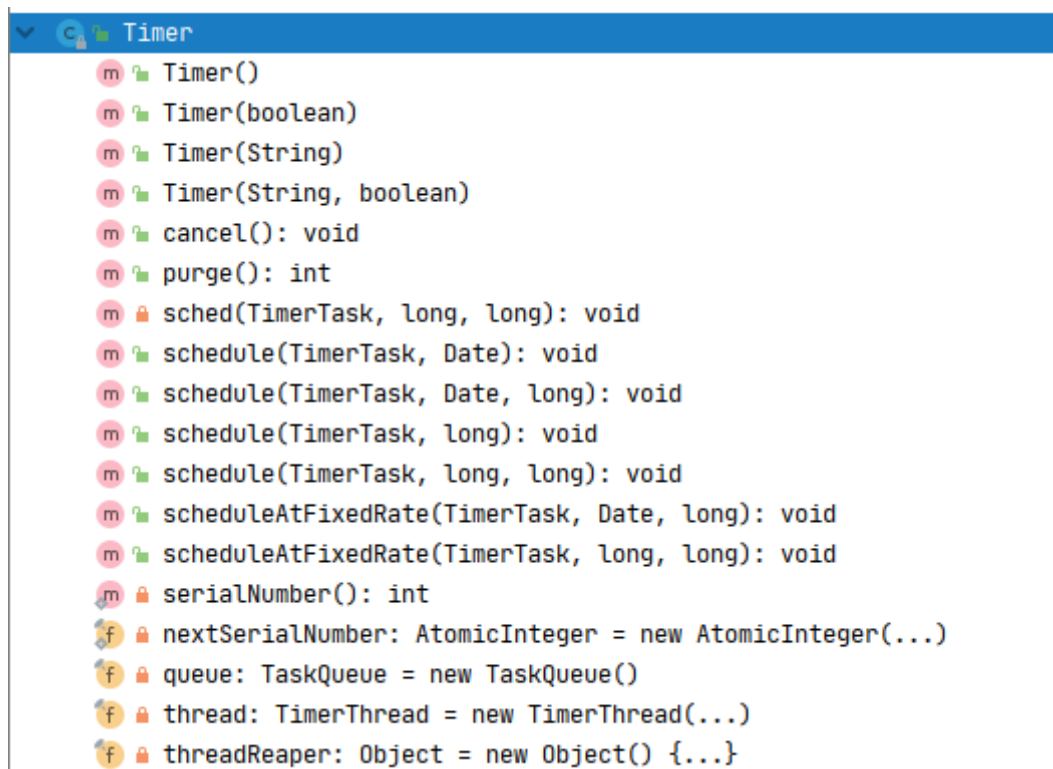
```

运行结果如下：



1.3. 中介者模式在源码中的体现

首先来看JDK中的Timer类，打开Timer的结构我们发现Timer类中有很多的schedule()重载方法，如下图：



我们任意点开其中一个方法，发现所有方法最终都是调用了私有的sched()方法，我们来看一下它的源码：

```

1 public class Timer {
2     ...
3     public void schedule(TimerTask task, Date time) {
4         sched(task, time.getTime(), 0);
5     }
6     ...
7     private void sched(TimerTask task, long time, long period) {
8         if (time < 0)
9             throw new IllegalArgumentException("Illegal execution time.");
10
11         // Constrain value of period sufficiently to prevent numeric
12         // overflow while still being effectively infinitely large.
13         if (Math.abs(period) > (Long.MAX_VALUE >> 1))
14             period >>= 1;
15
16         synchronized(queue) {
17             if (!thread.newTasksMayBeScheduled)
18                 throw new IllegalStateException("Timer already cancelled.");
19
20             synchronized(task.lock) {
21                 if (task.state != TimerTask.VIRGIN)
22                     throw new IllegalStateException(
23                         "Task already scheduled or cancelled");
24                 task.nextExecutionTime = time;
25                 task.period = period;
26                 task.state = TimerTask.SCHEDULED;
27             }
28
29             queue.add(task);
30             if (queue.getMin() == task)
31                 queue.notify();
32         }
33     }
34     ...

```

而且，我们发现，不管是什么样的任务都被加入到一个队列中顺序执行。我们把这个队列中的所有对象称之为“同事”。同事之间通信都是通过Timer来协调完成的，Timer就承担了中介者的角色。

1.4. 中介者模式的优缺点

优点：

- 1、减少类间依赖，将多对多依赖转化成了一对多，降低了类间耦合；
- 2、类间各司其职，符合迪米特法则。

缺点：

中介者模式中将原本多个对象直接的相互依赖变成了中介者和多个同事类的依赖关系。当同事类越多时，中介者就会越臃肿，变得复杂且难以维护。

2. 解释器模式

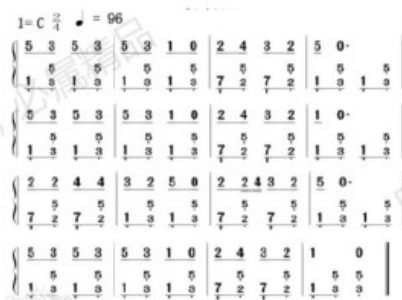
解释器模式 (Interpreter Pattern) 是指给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。是一种按照规定的语法 (文法) 进行解析的模式，属于行为型模式。

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.

就好比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。

2.1. 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

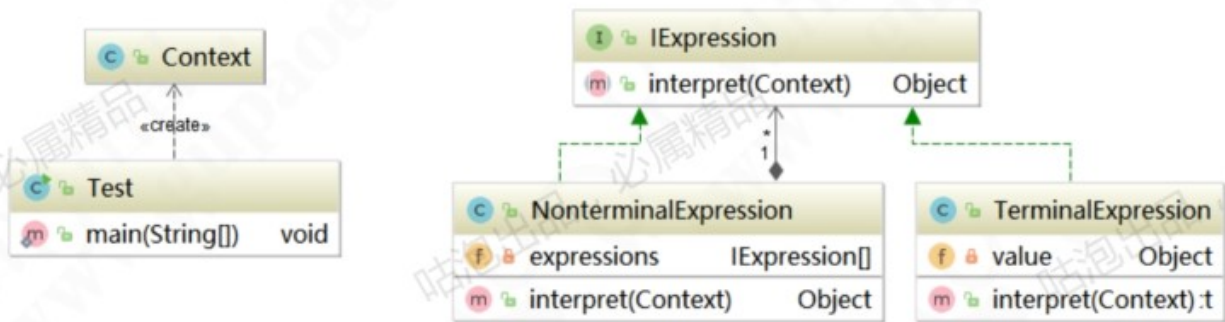


摩斯密码

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 $R=R_1+R_2$ ， R_1 和 R_2 就是终结符，对应的解析 R_1 和 R_2 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，但有多实例，对应不同的终结符（ R_1 ， R_2 ）；

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 $R=R_1+R_2$ 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 $R=R_1+R_2$ ，给 R_1 赋值100，给 R_2 赋值200，这些信息需要存放到环境中。

2.2. 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：

```

1 package cn.sitedev.calculate;
2
3 public interface IArithmeticInterpreter {
4     int interpret();
5 }
  
```

创建终结符表达式角色 Interpreter抽象类：

```

1 package cn.sitedev.calculate;
2
3 public abstract class Interpreter implements IArithmeticInterpreter {
4     protected IArithmeticInterpreter left;
5     protected IArithmeticInterpreter right;
  
```

```

6
7     public Interpreter(IArithmeticInterpreter left, IArithmeticInterpreter right) {
8         this.left = left;
9         this.right = right;
10    }
11 }

```

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：

```

1 package cn.sitedev.calculate;
2
3 public class AddInterpreter extends Interpreter {
4
5     public AddInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right) {
6         super(left, right);
7     }
8
9     @Override
10    public int interpret() {
11        return this.left.interpret() + this.right.interpret();
12    }
13 }

```

减法运算表达式 SubInterpreter类：

```

1 package cn.sitedev.calculate;
2
3 public class SubInterpreter extends Interpreter {
4
5     public SubInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right) {
6         super(left, right);
7     }
8
9     @Override
10    public int interpret() {
11        return this.left.interpret() - this.right.interpret();
12    }
13 }

```

乘法运算表达式 MultiInterpreter类：

```
1 package cn.sitedev.calculate;
2
3 public class MultiInterpreter extends Interpreter {
4
5     public MultiInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right)
6         super(left, right);
7     }
8
9     @Override
10    public int interpret() {
11        return this.left.interpret() * this.right.interpret();
12    }
13 }
```

除法运算表达式DivInterpreter类：

```
1 package cn.sitedev.calculate;
2
3 public class DivInterpreter extends Interpreter {
4     public DivInterpreter(IArithmeticInterpreter left, IArithmeticInterpreter right) {
5         super(left, right);
6     }
7
8     @Override
9     public int interpret() {
10        return this.left.interpret() / this.right.interpret();
11    }
12 }
```

数字表达式NumInterpreter类：

```
1 package cn.sitedev.calculate;
2
3 public class NumInterpreter implements IArithmeticInterpreter {
4     private int value;
5
6     public NumInterpreter(int value) {
```

```

7         this.value = value;
8     }
9
10    @Override
11    public int interpret() {
12        return this.value;
13    }
14 }

```

创建计算器MyCalculator类：

```

1 package cn.sitedev.calculate;
2
3 import java.util.Stack;
4
5 public class MyCalculator {
6     private Stack<IArithmeticInterpreter> stack = new Stack<>();
7
8     public MyCalculator(String expression) {
9         this.parse(expression);
10    }
11
12    private void parse(String expression) {
13        String[] elements = expression.split(" ");
14        IArithmeticInterpreter left, right;
15        for (int i = 0; i < elements.length; i++) {
16            String operator = elements[i];
17            if (OperatorUtil.isOperator(operator)) {
18                left = this.stack.pop();
19                right = new NumInterpreter(Integer.valueOf(elements[++i]));
20                System.out.println("出栈:" + left.interpret() + "和" + right.interpret());
21                this.stack.push(OperatorUtil.getInterpreter(left, right, operator));
22                System.out.println("应用运算符:" + operator);
23            } else {
24                NumInterpreter numInterpreter = new NumInterpreter(Integer.valueOf(elements[i]));
25                this.stack.push(numInterpreter);
26                System.out.println("入栈:" + numInterpreter.interpret());
27            }
28        }
29    }
30 }

```

```

31     public int calculate() {
32         return this.stack.pop().interpret();
33     }
34 }

```

工具类OperatorUtil具体代码：

```

1  package cn.sitedev.calculate;
2
3  public class OperatorUtil {
4      public static boolean isOperator(String symbol) {
5          return "+".equals(symbol) || "-".equals(symbol) || "*".equals(symbol) || "/".eq
6      }
7
8      public static Interpreter getInterpreter(IArithmeticInterpreter left,
9                                              IArithmeticInterpreter right, String symbol) {
10         Interpreter interpreter = null;
11         switch (symbol) {
12             case "+":
13                 interpreter = new AddInterpreter(left, right);
14                 break;
15             case "-":
16                 interpreter = new SubInterpreter(left, right);
17                 break;
18             case "*":
19                 interpreter = new MultiInterpreter(left, right);
20                 break;
21             case "/":
22                 interpreter = new DivInterpreter(left, right);
23                 break;
24             default:
25                 interpreter = null;
26                 break;
27         }
28         return interpreter;
29     }
30 }

```

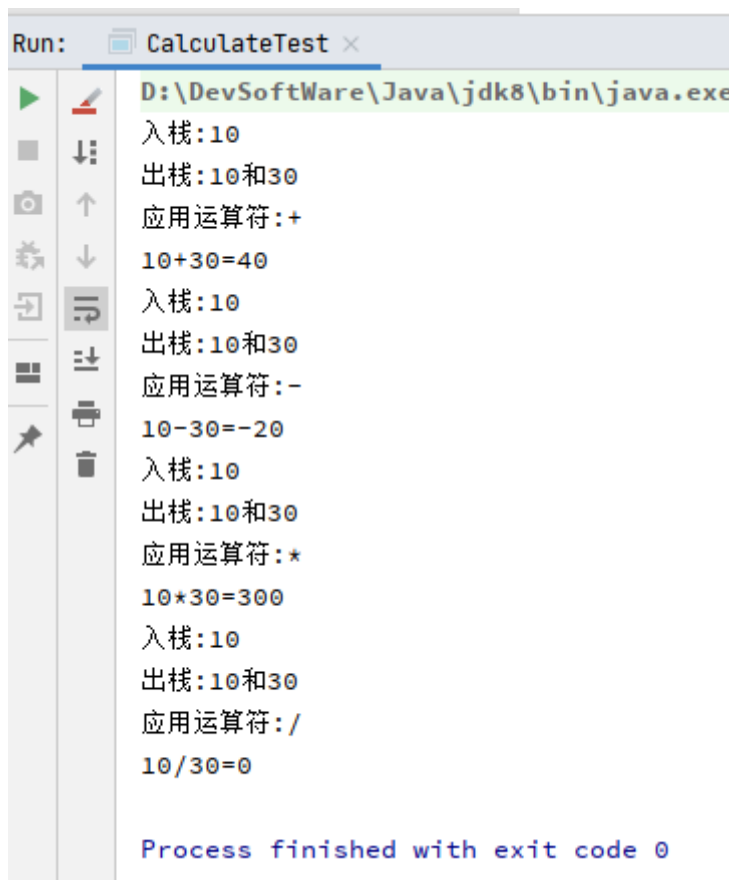
编写客户端代码：

```

1 package cn.sitedev.calculate;
2
3 public class CalculateTest {
4     public static void main(String[] args) {
5         System.out.println("10+30=" + new MyCalculator("10 + 30").calculate());
6         System.out.println("10-30=" + new MyCalculator("10 - 30").calculate());
7         System.out.println("10*30=" + new MyCalculator("10 * 30").calculate());
8         System.out.println("10/30=" + new MyCalculator("10 / 30").calculate());
9     }
10 }

```

运行结果如下：



```

Run: CalculateTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe
入栈:10
出栈:10和30
应用运算符:+
10+30=40
入栈:10
出栈:10和30
应用运算符:-
10-30=-20
入栈:10
出栈:10和30
应用运算符:*
10*30=300
入栈:10
出栈:10和30
应用运算符:/
10/30=0
Process finished with exit code 0

```

2.3. 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。

```

1 public final class Pattern
2     implements java.io.Serializable
3 {
4     ...
5     private Pattern(String p, int f) {

```

```

6      pattern = p;
7      flags = f;
8
9      // to use UNICODE_CASE if UNICODE_CHARACTER_CLASS present
10     if ((flags & UNICODE_CHARACTER_CLASS) != 0)
11         flags |= UNICODE_CASE;
12
13     // Reset group index count
14     capturingGroupCount = 1;
15     localCount = 0;
16
17     if (pattern.length() > 0) {
18         compile();
19     } else {
20         root = new Start(lastAccept);
21         matchRoot = lastAccept;
22     }
23 }
24 ...
25
26 public static Pattern compile(String regex) {
27     return new Pattern(regex, 0);
28 }
29
30 public static Pattern compile(String regex, int flags) {
31     return new Pattern(regex, flags);
32 }
33 ...

```

再来看Spring 中的ExpressionParser接口。

```

1 public interface ExpressionParser {
2     Expression parseExpression(String expressionString) throws ParseException;
3
4     Expression parseExpression(String expressionString, ParserContext context) throws P
5     ...

```

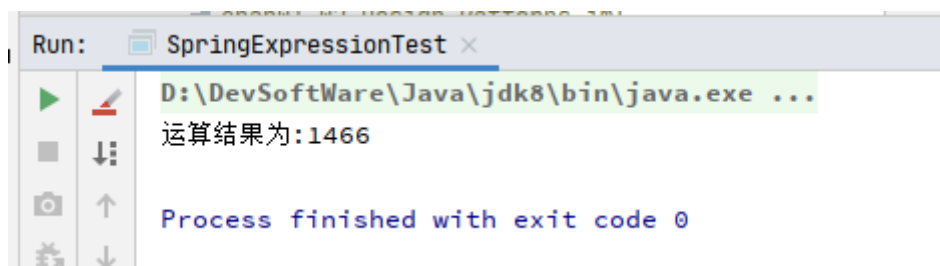
这里的源码我们不深入讲解，通过前面我们自己编写的案例大致能够清楚其原理。我们不妨来编写一段客户端代码验证一下功能。我们编写如下测试代码：


```

1 package cn.sitedev.calculate;
2
3 import org.springframework.expression.Expression;
4 import org.springframework.expression.ExpressionParser;
5 import org.springframework.expression.spel.standard.SpelExpressionParser;
6
7 public class SpringExpressionTest {
8
9     public static void main(String[] args) {
10         ExpressionParser parser = new SpelExpressionParser();
11         Expression expression = parser.parseExpression("100 * 2 + 400 * 3 + 66");
12         int result = (int) expression.getValue();
13         System.out.println("运算结果为:" + result);
14     }
15 }

```

其运行结果是：



和我们所期望的是一致的。

注意使用前需要引入依赖:

```

1     <dependency>
2         <groupId>org.springframework</groupId>
3         <artifactId>spring-expression</artifactId>
4         <version>5.0.2.RELEASE</version>
5     </dependency>

```

2.4. 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得。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。