

## 72 | 解释器模式：如何设计实现一个自定义接口告警规则功能？

2020-04-17 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 10:00 大小 9.17M



上一节课，我们学习了命令模式。命令模式将请求封装成对象，方便作为函数参数传递和赋值给变量。它主要的应用场景是给命令的执行附加功能，换句话说，就是控制命令的执行，比如，排队、异步、延迟执行命令、给命令执行记录日志、撤销重做命令等等。总体上来讲，命令模式的应用范围并不广。

今天，我们来学习解释器模式，它用来描述如何构建一个简单的“语言”解释器。比起命令模式，解释器模式更加小众，只在一些特定的领域会被用到，比如编译器、规则引擎、正则表达式。所以，解释器模式也不是我们学习的重点，你稍微了解一下就可以了。



话不多说，让我们正式开始今天的学习吧！

## 解释器模式的原理和实现

解释器模式的英文翻译是 Interpreter Design Pattern。在 GoF 的《设计模式》一书中，它是这样定义的：

Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

翻译成中文就是：解释器模式为某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法。

看了定义，你估计会一头雾水，因为这里面有很多我们平时开发中很少接触的概念，比如“语言”“语法”“解释器”。实际上，这里的“语言”不仅仅指我们平时说的中、英、日、法等各种语言。从广义上来讲，只要是能承载信息的载体，我们都可以称之为“语言”，比如，古代的结绳记事、盲文、哑语、摩斯密码等。

要想了解“语言”表达的信息，我们就必须定义相应的语法规则。这样，书写者就可以根据语法规则来书写“句子”（专业点的叫法应该是“表达式”），阅读者根据语法规则来阅读“句子”，这样才能做到信息的正确传递。而我们要讲的解释器模式，其实就是用来实现根据语法规则解读“句子”的解释器。

为了让你更好地理解定义，我举一个比较贴近生活的例子来解释一下。

实际上，理解这个概念，我们可以类比中英文翻译。我们知道，把英文翻译成中文是有一定规则的。这个规则就是定义中的“语法”。我们开发一个类似 Google Translate 这样的翻译器，这个翻译器能够根据语法规则，将输入的中文翻译成英文。这里的翻译器就是解释器模式定义中的“解释器”。

刚刚翻译器这个例子比较贴近生活，现在，我们再举个更加贴近编程的例子。

假设我们定义了一个新的加减乘除计算“语言”，语法规则如下：

运算符只包含加、减、乘、除，并且没有优先级的概念；


表达式（也就是前面提到的“句子”）中，先书写数字，后书写运算符，空格隔开；

按照先后顺序，取出两个数字和一个运算符计算结果，结果重新放入数字的最头部位  
置，循环上述过程，直到只剩下一个数字，这个数字就是表达式最终的计算结果。

我们举个例子来解释一下上面的语法规则。

比如 “8 3 2 4 - + \* ” 这样一个表达式，我们按照上面的语法规则来处理，取出数字 “8  
3” 和 “-” 运算符，计算得到 5，于是表达式就变成了 “5 2 4 + \* ”。然后，我们再取  
出 “5 2” 和 “+” 运算符，计算得到 7，表达式就变成了 “7 4 \* ”。最后，我们取  
出 “7 4” 和 “\*” 运算符，最终得到的结果就是 28。

看懂了上面的语法规则，我们将它用代码实现出来，如下所示。代码非常简单，用户按照上  
面的规则书写表达式，传递给 interpret() 函数，就可以得到最终的计算结果。

 复制代码

```
1 public class ExpressionInterpreter {
2     private Deque<Long> numbers = new LinkedList<>();
3
4     public long interpret(String expression) {
5         String[] elements = expression.split(" ");
6         int length = elements.length;
7         for (int i = 0; i < (length+1)/2; ++i) {
8             numbers.addLast(Long.parseLong(elements[i]));
9         }
10
11         for (int i = (length+1)/2; i < length; ++i) {
12             String operator = elements[i];
13             boolean isValid = "+".equals(operator) || "-".equals(operator)
14                 || "*".equals(operator) || "/".equals(operator);
15             if (!isValid) {
16                 throw new RuntimeException("Expression is invalid: " + expression);
17             }
18
19             long number1 = numbers.pollFirst();
20             long number2 = numbers.pollFirst();
21             long result = 0;
22             if (operator.equals("+")) {
23                 result = number1 + number2;
24             } else if (operator.equals("-")) {
25                 result = number1 - number2;
26             } else if (operator.equals("*")) {
27                 result = number1 * number2;
28             } else if (operator.equals("/")) {
29                 result = number1 / number2;
30             }
31             numbers.addFirst(result);
32         }
33     }
34 }
```

```

32     }
33
34     if (numbers.size() != 1) {
35         throw new RuntimeException("Expression is invalid: " + expression);
36     }
37
38     return numbers.pop();
39 }
40 }

```

在上面的代码实现中，语法规则的解析逻辑（第 23、25、27、29 行）都集中在一个函数中，对于简单的语法规则的解析，这样的设计就足够了。但是，对于复杂的语法规则的解析，逻辑复杂，代码量多，所有的解析逻辑都耦合在一个函数中，这样显然是不合适的。这个时候，我们就要考虑拆分代码，将解析逻辑拆分到独立的小类中。

该怎么拆分呢？我们可以借助解释器模式。

解释器模式的代码实现比较灵活，没有固定的模板。我们前面也说过，应用设计模式主要是应对代码的复杂性，实际上，解释器模式也不例外。它的代码实现的核心思想，就是将语法解析的工作拆分到各个小类中，以此来避免大而全的解析类。一般的做法是，将语法规则拆分成一些小的独立的单元，然后对每个单元进行解析，最终合并为对整个语法规则的解析。

前面定义的语法规则有两类表达式，一类是数字，一类是运算符，运算符又包括加减乘除。利用解释器模式，我们把解析的工作拆分到 `NumberExpression`、`AdditionExpression`、`SubstractionExpression`、`MultiplicationExpression`、`DivisionExpression` 这样五个解析类中。

按照这个思路，我们对代码进行重构，重构之后的代码如下所示。当然，因为加减乘除表达式的解析比较简单，利用解释器模式的设计思路，看起来有点过度设计。不过呢，这里我主要是为了解释原理，你明白意思就好，不用过度细究这个例子。

 复制代码

```

1  public interface Expression {
2      long interpret();
3  }
4
5  public class NumberExpression implements Expression {
6      private long number;
7

```

```

8     public NumberExpression(long number) {
9         this.number = number;
10    }
11
12    public NumberExpression(String number) {
13        this.number = Long.parseLong(number);
14    }
15
16    @Override
17    public long interpret() {
18        return this.number;
19    }
20 }
21
22 public class AdditionExpression implements Expression {
23     private Expression exp1;
24     private Expression exp2;
25
26     public AdditionExpression(Expression exp1, Expression exp2) {
27         this.exp1 = exp1;
28         this.exp2 = exp2;
29     }
30
31     @Override
32     public long interpret() {
33         return exp1.interpret() + exp2.interpret();
34     }
35 }
36 // SubstractionExpression/MultiplicationExpression/DivisionExpression与Addition
37
38 public class ExpressionInterpreter {
39     private Deque<Expression> numbers = new LinkedList<>();
40
41     public long interpret(String expression) {
42         String[] elements = expression.split(" ");
43         int length = elements.length;
44         for (int i = 0; i < (length+1)/2; ++i) {
45             numbers.addLast(new NumberExpression(elements[i]));
46         }
47
48         for (int i = (length+1)/2; i < length; ++i) {
49             String operator = elements[i];
50             boolean isValid = "+".equals(operator) || "-".equals(operator)
51                 || "*".equals(operator) || "/".equals(operator);
52             if (!isValid) {
53                 throw new RuntimeException("Expression is invalid: " + expression);
54             }
55
56             Expression exp1 = numbers.pollFirst();
57             Expression exp2 = numbers.pollFirst();
58             Expression combinedExp = null;
59             if (operator.equals("+")) {

```

```

60         combinedExp = new AdditionExpression(exp1, exp2);
61     } else if (operator.equals("-")) {
62         combinedExp = new AdditionExpression(exp1, exp2);
63     } else if (operator.equals("*")) {
64         combinedExp = new AdditionExpression(exp1, exp2);
65     } else if (operator.equals("/")) {
66         combinedExp = new AdditionExpression(exp1, exp2);
67     }
68     long result = combinedExp.interpret();
69     numbers.addFirst(new NumberExpression(result));
70 }
71
72 if (numbers.size() != 1) {
73     throw new RuntimeException("Expression is invalid: " + expression);
74 }
75
76 return numbers.pop().interpret();
77 }
78 }


```

## 解释器模式实战举例

接下来，我们再来看一个更加接近实战的例子，也就是咱们今天标题中的问题：如何实现一个自定义接口告警规则功能？

在我们平时的项目开发中，监控系统非常重要，它可以时刻监控业务系统的运行情况，及时将异常报告给开发者。比如，如果每分钟接口出错数超过 100，监控系统就通过短信、微信、邮件等方式发送告警给开发者。

一般来讲，监控系统支持开发者自定义告警规则，比如我们可以用下面这样一个表达式，来表示一个告警规则，它表达的意思是：每分钟 API 总出错数超过 100 或者每分钟 API 总调用数超过 10000 就触发告警。

 复制代码

```
1 api_error_per_minute > 100 || api_count_per_minute > 10000
```

在监控系统中，告警模块只负责根据统计数据 and 告警规则，判断是否触发告警。至于每分钟 API 接口出错数、每分钟接口调用数等统计数据的计算，是由其他模块来负责的。其他模块将统计数据放到一个 Map 中（数据的格式如下所示），发送给告警模块。接下来，我们只关注告警模块。




```
1 Map<String, Long> apiStat = new HashMap<>();
2 apiStat.put("api_error_per_minute", 103);
3 apiStat.put("api_count_per_minute", 987);
```

为了简化讲解和代码实现，我们假设自定义的告警规则只包含 “||、&&、>、<、==” 这五个运算符，其中，“>、<、==” 运算符的优先级高于 “||、&&” 运算符，“&&” 运算符优先级高于 “||”。在表达式中，任意元素之间需要通过空格来分隔。除此之外，用户可以自定义要监控的 key，比如前面的 api\_error\_per\_minute、api\_count\_per\_minute。

那如何实现上面的需求呢？我写了一个骨架代码，如下所示，其中的核心的实现我没有给出，你可以当作面试题，自己试着去补全一下，然后再看我的讲解。

```
1 public class AlertRuleInterpreter {
2
3     // key1 > 100 && key2 < 1000 || key3 == 200
4     public AlertRuleInterpreter(String ruleExpression) {
5         //TODO:由你来完善
6     }
7
8     //<String, Long> apiStat = new HashMap<>();
9     //apiStat.put("key1", 103);
10    //apiStat.put("key2", 987);
11    public boolean interpret(Map<String, Long> stats) {
12        //TODO:由你来完善
13    }
14
15 }
16
17 public class DemoTest {
18     public static void main(String[] args) {
19         String rule = "key1 > 100 && key2 < 30 || key3 < 100 || key4 == 88";
20         AlertRuleInterpreter interpreter = new AlertRuleInterpreter(rule);
21         Map<String, Long> stats = new HashMap<>();
22         stats.put("key1", 101l);
23         stats.put("key3", 121l);
24         stats.put("key4", 88l);
25         boolean alert = interpreter.interpret(stats);
26         System.out.println(alert);
27     }
28 }
```

实际上，我们可以把自定义的告警规则，看作一种特殊“语言”的语法规则。我们实现一个解释器，能够根据规则，针对用户输入的数据，判断是否触发告警。利用解释器模式，我们把解析表达式的逻辑拆分到各个小类中，避免大而复杂的大类的出现。按照这个实现思路，我把刚刚的代码补全，如下所示，你可以拿你写的代码跟我写的对比一下。

 复制代码

```
1 public interface Expression {
2     boolean interpret(Map<String, Long> stats);
3 }
4
5 public class GreaterExpression implements Expression {
6     private String key;
7     private long value;
8
9     public GreaterExpression(String strExpression) {
10         String[] elements = strExpression.trim().split("\\s+");
11         if (elements.length != 3 || !elements[1].trim().equals(">")) {
12             throw new RuntimeException("Expression is invalid: " + strExpression);
13         }
14         this.key = elements[0].trim();
15         this.value = Long.parseLong(elements[2].trim());
16     }
17
18     public GreaterExpression(String key, long value) {
19         this.key = key;
20         this.value = value;
21     }
22
23     @Override
24     public boolean interpret(Map<String, Long> stats) {
25         if (!stats.containsKey(key)) {
26             return false;
27         }
28         long statValue = stats.get(key);
29         return statValue > value;
30     }
31 }
32
33 // LessExpression/EqualExpression跟GreaterExpression代码类似，这里就省略了
34
35 public class AndExpression implements Expression {
36     private List<Expression> expressions = new ArrayList<>();
37
38     public AndExpression(String strAndExpression) {
39         String[] strExpressions = strAndExpression.split("&&");
40         for (String strExpr : strExpressions) {
41             if (strExpr.contains(">")) {
42                 expressions.add(new GreaterExpression(strExpr));
43             } else if (strExpr.contains("<")) {
```



```

44         expressions.add(new LessExpression(strExpr));
45     } else if (strExpr.contains("==")) {
46         expressions.add(new EqualExpression(strExpr));
47     } else {
48         throw new RuntimeException("Expression is invalid: " + strAndExpression);
49     }
50 }
51 }
52
53 public AndExpression(List<Expression> expressions) {
54     this.expressions.addAll(expressions);
55 }
56
57 @Override
58 public boolean interpret(Map<String, Long> stats) {
59     for (Expression expr : expressions) {
60         if (!expr.interpret(stats)) {
61             return false;
62         }
63     }
64     return true;
65 }
66
67 }
68
69 public class OrExpression implements Expression {
70     private List<Expression> expressions = new ArrayList<>();
71
72     public OrExpression(String strOrExpression) {
73         String[] andExpressions = strOrExpression.split("\\|\\\\|");
74         for (String andExpr : andExpressions) {
75             expressions.add(new AndExpression(andExpr));
76         }
77     }
78
79     public OrExpression(List<Expression> expressions) {
80         this.expressions.addAll(expressions);
81     }
82
83     @Override
84     public boolean interpret(Map<String, Long> stats) {
85         for (Expression expr : expressions) {
86             if (expr.interpret(stats)) {
87                 return true;
88             }
89         }
90         return false;
91     }
92 }
93
94 public class AlertRuleInterpreter {
95     private Expression expression;

```

```
96
97     public AlertRuleInterpreter(String ruleExpression) {
98         this.expression = new OrExpression(ruleExpression);
99     }
100
101     public boolean interpret(Map<String, Long> stats) {
102         return expression.interpret(stats);
103     }
104 }
```

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

解释器模式为某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法。实际上，这里的“语言”不仅仅指我们平时说的中、英、日、法等各种语言。从广义上来讲，只要是能承载信息的载体，我们都可以称之为“语言”，比如，古代的结绳记事、盲文、哑语、摩斯密码等。

要想了解“语言”要表达的信息，我们就必须定义相应的语法规则。这样，书写者就可以根据语法规则来书写“句子”（专业点的叫法应该是“表达式”），阅读者根据语法规则来阅读“句子”，这样才能做到信息的正确传递。而我们要讲的解释器模式，其实就是用来实现根据语法规则解读“句子”的解释器。

解释器模式的代码实现比较灵活，没有固定的模板。我们前面说过，应用设计模式主要是应对代码的复杂性，解释器模式也不例外。它的代码实现的核心思想，就是将语法解析的工作拆分到各个小类中，以此来避免大而全的解析类。一般的做法是，将语法规则拆分一些小的独立的单元，然后对每个单元进行解析，最终合并为对整个语法规则的解析。

## 课堂讨论

1. 在你过往的项目经历或阅读源码的时候，有没有用到或者见过解释器模式呢？
2. 在告警规则解析的例子中，如果我们要在表达式中支持括号“()”，那如何对代码进行重构呢？你可以把它当作练习，试着编写一下代码。

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

# 极客时间充值卡

— 充值享优惠，学习更高效 —



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 71 | 命令模式：如何利用命令模式实现一个手游后端架构？

下一篇 73 | 中介模式：什么时候用中介模式？什么时候用观察者模式？

## 精选留言 (14)

写留言



**Ken张云忠**

2020-04-17

Java中注解处理器做的就是解释的功能，以及前端编译时的语法分析、语义分析，后端编译时生成的中间表达式，用来触发更多优化，优化的处理可以理解为高效的解释，最终生成机器可以执行的汇编指令。

2

4



**Panmax**

2020-04-18

第二个代码示例中的代码段：

```
if (operator.equals("+")) {  
    combinedExp = new AdditionExpression(exp1, exp2);
```

```
} else if (operator.equals("-")) {  
    combinedExp = new AdditionExpression(exp1, exp2);...
```

展开 ▾



👍 3



**辣么大**

2020-04-17

关于问题一，使用过。偏向科研，自定义一门语言，然后通过语法解析器分析读入。例如使用RDDL(Relational Domain Definition Language)关系领域定义语言描述马尔可夫决策过程。

```
domain prop_dbn {
```

...

展开 ▾



👍 2



**Yang**

2020-04-18

因为做的是数据类型的项目，就是根据自定义的SQL来创建API以供可以通过http形式直接调用，项目中就是用Druid来解析SQL的，用的就是解释器模式，SQL语句中的每个字符对应一个表达式。

💬 1

👍 1



**test**

2020-04-17

加括号的话，要加一个ExpressionManager，在manager里面用括号把表达式划分为几段，再根据表达式间是 与 还是 或 来添加最上面那一层的表达式

展开 ▾



👍 1



**南山**

2020-04-21

还真有项目有这种场景，动态表单的值校验，可以自定义校验，校验之间还可以相互组合。但是用的是组合模式，后面思考一下解释器模式如何使用，貌似更适合~

展开 ▾



**makermade**

2020-04-19

最近整好要做告警规则相关的开发，，，醍醐灌顶

展开 ▾



++

2020-04-18

告警规则的代码 真是读了半天才看懂😓

展开



李稳

2020-04-17

输入规则，解析规则，输入数据，使用规则判断数据

展开



Heaven

2020-04-17

<https://github.com/HeavenXin/MonitorExpressionTest>

关于第二道题,趁着中午写了写,大家可以参考一下

展开



唐朝农民

2020-04-17

请问怎么生成复杂一点的表达式树呢

展开



Heaven

2020-04-17

对于一个Java程序员来说,应该知道从Java语言解释为JVM规范语言,是需要进行解释器解析的,从词法解析器,解析出对应的类定义属性等,到语法解析器,解析成对应的语法树,再使用语义解析器,进行判断规范和解析语法糖



守拙

2020-04-17

解释器模式符合单一职责原则. 在例子中, 为">", "<", "="分别封装成独立函数, 避免了处理函数过长导致的可读性, 可维护性问题.

解释器模式符合开闭原则. 在例子中, 如果要添加 "(" , ")" 解析功能, 封装 BracketExpression, AlertRuleInterpreter 添加 BracketExpression 就能实现新的需求.

展开 ▾



**liu\_liu**

2020-04-17

react 中的 jsx

展开 ▾

