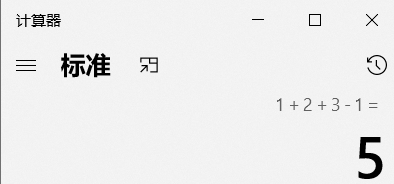
# 解释器模式简介

解释器模式用于描述如何构成一个简单的语言解释器，主要应用于使用面向对象语言开发的解释器的设计。当需要开发一个新的语言时可以考虑使用解释器模式。

## 概述

1. 虽然目前计算机编程语言有几百种，但有时候人们还是希望能用一些简单的语言来实现一些特定的操作，用户只要向计算机输入一个句子或文件，它就能按照先定义的文法规则对句子或文件进行解释，从而实现相应的功能。例如提供一个简单的加法/减法解释器，只要输入一个加法/减法表达式，它就能计算出表达式结果。如输入字符串亦表达式“1+2+3-1”时将输出计算结果为5。



众所周知，像Java、C++等语言无法直接解释类似“1+2+3-1”这样的字符串（如果直接作为数字表达式则可以解释），必须自己定义一套文法规则来实现对这些语句的解释，即设计一个自定义语言。在实际开发中，这些简单的自定义语言可以基于现有的编程语言来设计，如果所基于的编程语言是面向对象语言，此时可以适应解释器模式实现自定义语言。

1. 解释器模式是一种使用频率相对较低且学习难度相对较大的设计模式，它用于描述如何使用面向对象语言构成一个简单的语言解释器。在某些情况下，为了更好地描述某些特定类型的问题可以创建一种新的语言，这种语言拥有自己的表达式和结构，即文法规则，这些问题的实例将对应为该语言中的句子，此时可以适应解释器模式来设计这种新的语言。对解释器模式的学习能够加深对面向对象思想的理解，并且理解编程语言中文法规则的解释过程。
2. 解释器模式（Interpreter Pattern）的定义如下：给定一个语言，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。

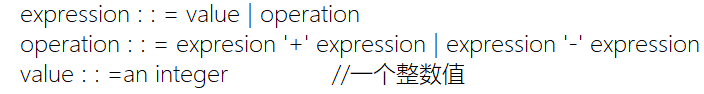
在解释器模式的定义中所指的“语言”是使用规定格式和语法的代码，解释器模式是一种类行为型模式。

## 文法规则和抽象语法树

解释器模式描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。在正式分析解释器模式结构之前先来学习如何表示一个语言的文法规则以及如何构造一棵抽象语法树。

**1、文法规则**

在前面提到的加法/减法解释器中，每一个输入表达式（例如“1+2+3-1”）都包含了3个语言单位，可以适应一下文法规则来定义：

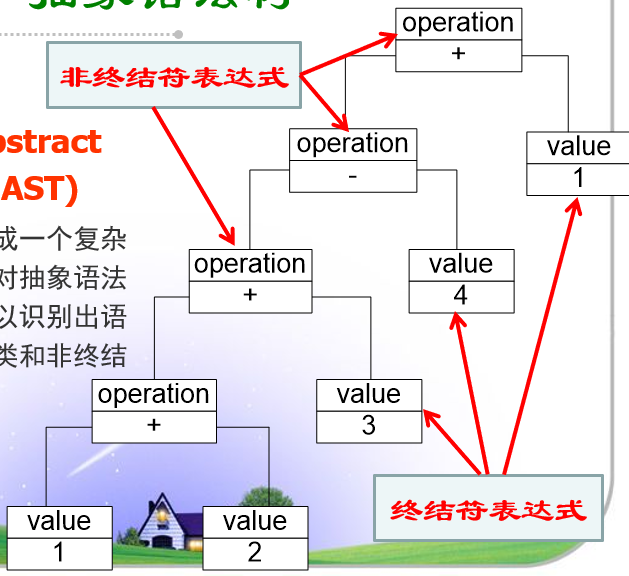


该文法规则包含3条语句，第一条表示表达式的组成方式，其中value和operation是后面两个语言单位的定义，每一条语言所定义的字符串（如operation和value）称为语言构造成分或语言单位，符号“::=”是“定位为”的意思，其左边的语言单位通过右边来进行说明和定义，语言单位对应终结符表达式和非终结符表达式。例如本规则中的opeartion是非非终结符表达式，它的组成元素仍然可以是表达式，可以进一步分解，而value是终结符表达式，它的组成元素时最基本的语言单位，不能再进行分解。

在文法规则中可以适应一些符号来表示不同的含义，例如使用“|”表示或、使用“{”和“}”表示组合，使用“\*”表示出现0次或多次等，其中使用频率最高的符号时表示“或”关系的“|”，例如文法规则“boolValue::=0|1”表示终结符表达式boolValue的取值可以为0或者1.

**2、抽象语法树**

除了使用文法规则来定义一个语言外，在解释器模式中还可以通过一种称为抽象语法树（Abstract Syntax Tree，AST）的图形方式来直观地表示语言的构成，每一棵抽象语法树对应一个语言实例，例如加法/加法表达式中的语句“1+2+3-4+1”可以通过如下图所示的抽象语法树来表示：



在该抽象语法树中可以通过终结符表达式value和非终结符表达式operation组成复杂的语句，每个文法规则的语言实例都可以表示为一个抽象语法树，即每一条具体的语句都可以用类似上图所示的抽象语法树来表示，在图中终结符表达式类的实例作为树的叶子节点，而非终结符表达式类的实例作为非叶子节点，它们可以将终结符表达式类的实例以及包含终结符和非终结符实例的子表达式作为其子结点。抽象语法树描述了如何构成一个复杂的句子，通过对抽象语法树的分析可以识别出语言中的终结符类和非终结符类。

## 结构

由于表达式可分为终结符表达式和非终结符表达式，因此解释器模式的结构与组合模式的结构有些类似，但在解释器模式中包含更多的组成元素，它的结构如下图所示：



有结构图可知，解释器模式包含以下4个角色。

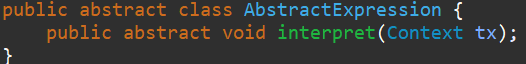
1. AbstractExpression（抽象表达式）：在抽象表达式中声明了抽象的解释器操作，它是所有终结符表达式和非终结符表达式的公共父类。
2. TerminalExpression（终结符表达式）：终结符表达式是抽象表达式的子类，它实现了与文法中的终结符相关联的解释操作，在句子中每一个终结符都是该类的一个实例。通常在一个解释器模式中只有少数几个终结符表达式类，它们的实例可以通过非终结符表达式组成较为复杂的句子。
3. NonterminalExpression（非终结符表达式）：非终结符表达式也是抽象表达式的子类，它实现了文法中的非终结符的解释操作，由于在非终结符表达式中可以包含终结符表达式，也可以继续包含非终结符表达式，因此其解释一般通过递归的方法完成。
4. Context（环境类）：环境类又称为上下文类，它用于存储解释器之外的一些全局信息，通常它临时存储了需要解释的语句。

# 实现

## 实现原理

在解释器模式中每一种终结符和非终结符都有一个具体类与之对应，正因为使用类来表示每一条文法规则，所以系统将具有较好的灵活性和可扩展性。

1. 对于所有的终结符和非终结符，首先需要抽象出一个公共父类，即抽象表达式类。其典型代码如下：



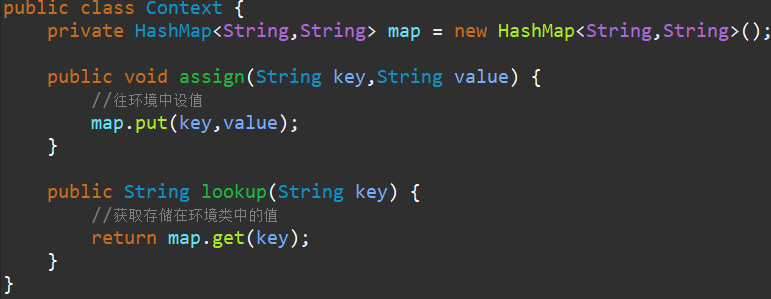
1. 终结符表达式和非终结符表达式类都是抽象表达式的子类，对于终结符表达式类，其代码很简单，主要是对终结符元素的处理。其典型代码如下：



1. 对于非终结符表达式，其代码相对比较复杂，因为可以通过非终结符将表达式组合成更加复杂的结构，对于包含两个操作元素的非终结符表达式，其典型代码如下：



1. 除了上述用于表示表达式的类以外，通常在解释器模式中还提供了一个环境类Context，用于存储一些全局信息，在环境类中一般包含一个HashMap或ArrayList等类型的集合对象（也可以直接由HashMap等集合类充当环境类），存储一系列公共信息，例如变量名与值的映射关系（key/value）等，用于在执行具体的解释操作时从中获取相关信息。其典型代码片段如下：

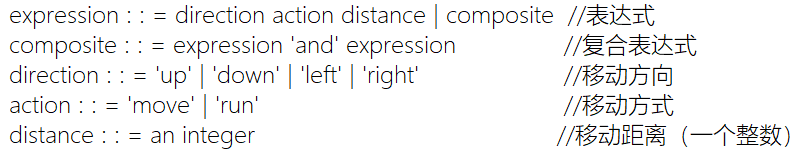


环境类Context的对象通常作为参数被传递到所有表达式的解释方法interpret( )方法中，可以在环境类对象中存储和访问表达式解释器的状态，向表达式解释器提供一些全局的、公共的数据，此外还可以在环境类中增加一些所有表达式解释器共有的功能，以减轻解释器的职责。当系统无须提供全局公共信息时可以省略环境类，根据实际情况决定是否需要环境类。

## 实例

实例说明：某软件公司要开发一套机器人控制程序，在该机器人控制程序中包含一些简单的英文控制指令，每一个指令对应一个表达式（expression），该表达式可以是简单表达式也可以是复合表达式，每一个简单表达式由移动方向（direction），移动方式（action）和移动距离（distance）3部门组成，其中移动方向包括上（up）、下（down）、左（left）、右（right）；移动方式包括移动（move）和快速移动（run）；移动距离为一个正整数。两个表达式事件可以通过与（and）连接形成复合（composite）表达式。用户通过对图形化的设置界面进行操作可以创建一个机器人控制指令，机器人在收到指令后将按照指令的设置进行移动，例如输入“up move 5”，则“向上移动5个单位”；输入控制指令“down run 10 and left move 20”，则“向下快速移动10个单位再向左移动20个单位”。现使用解释器模式来设计该程序并模拟实现。

根据上述需求描述用形式化语言表示该简单语言的文法规则如下：



该语言定义了5条文法规则，对应5个语言单位可以分为两类，一类为终结符（也称终结符表达式），例如direction、action和distance，它们是语言的最小组成单位，不能再进行拆分；另一类为非终结符（也称为非终结符表达式），例如expression和composite，它们都是一个完整的句子，包含一系列终结符或非终结符。针对这5条文法规则分别提高5个类来实现，其中终结符表达式direction、action和distance对应DirectionNode类、ActionNode类和DistanceNode类，非终结符表达式expression和composite对应SentenceNode类和AndNode类。可以通过抽象语法树来表示具体解释过程，例如机器人控制指令“down run 10 and left move 20”对应的抽象语法树如下图所示：

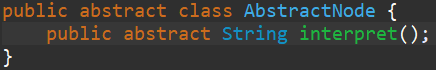


机器人控制程序结构图如下：

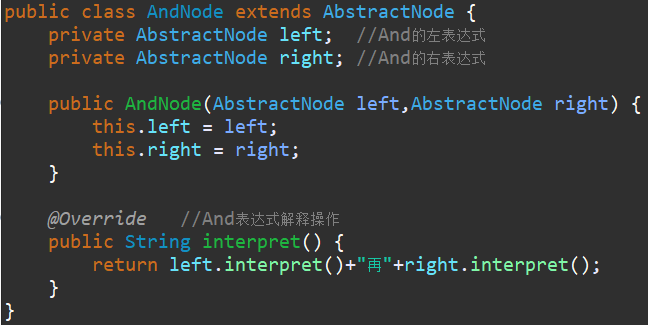


结构图中，AbstractNode充当抽象表达式角色，DirectionNode、ActionNode和DistanceNode充当终结符表达式角色，AndNode和SentenceNode充当非终结符表达式角色。

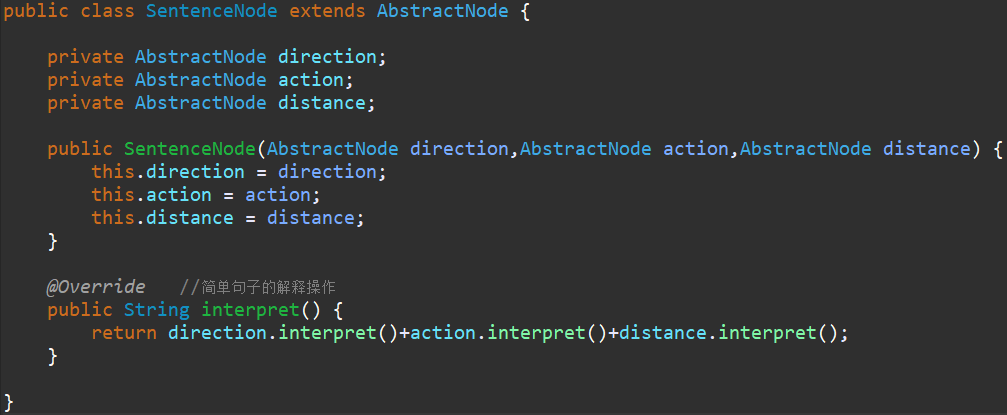
1. AbstractNode：抽象节点类，充当抽象表达式角色



1. AndNode：And节点类，充当非终结符表达式角色



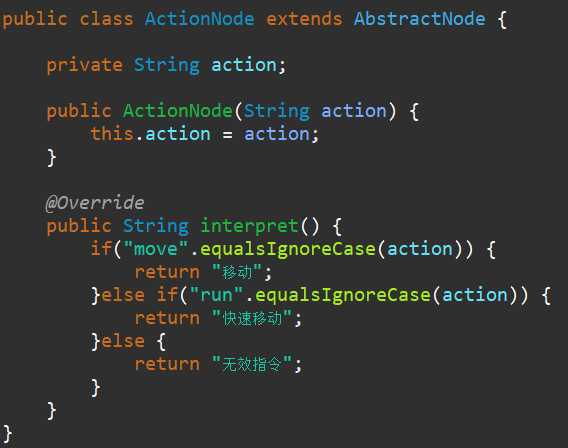
1. SentenceNode：简单句子节点，充当非终结符表达式角色



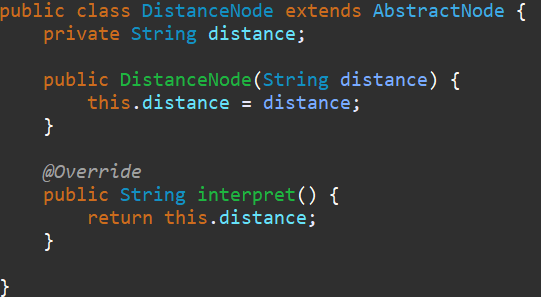
1. DirectionNode：方向节点类，充当终结符表达式角色



1. ActionNode：动作节点类，充当终结符表达式角色

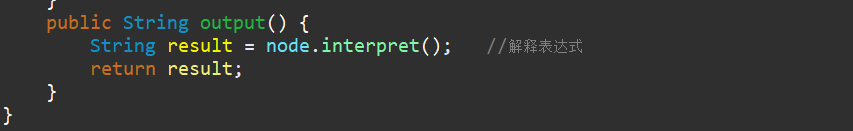


1. DistanceNode：距离节点类，充当终结符表达式角色

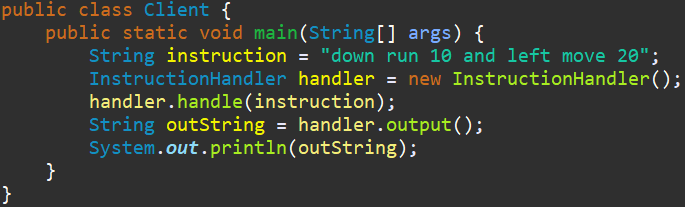


1. InstructionHandler：指令处理类，工具类，提供相应的方法对输入的执行进行处理。它将输入的指令分割为字符串数组，将第一个、第二个和第三个单词合成一个句子，并存入栈中；如果发现单词“and”，则将“and”后的第一个、第二个和第三个单词组成合成一个新的句子作为“and”的右表达式，并从栈中取出原先所存的句子作为左表达式，然后组合成一个And节点存入栈中。依此类推，知道整个指令解析结束。





1. Client：客户端测试类



运行结果如下：



在本实例中对机器人控制指令的输出结果进行模拟，将英文指令翻译为中文指令，在真实情况下系统将调用不同的控制程序对机器人进行控制，包括对移动方向、方式和距离的控制等。

# 优缺点及适用环境

解释器模式为自定义语言的设计和实现提供了一种解决方案，用于定义一组文法规则并通过这组文法规则来解释语言中的句子。虽然解释器模式的使用频率不是特别高，但是它在正则表达式、XML文档解释等领域还是得到了广泛使用。与解释器模式类似，目前还诞生了很多基于抽象语法树的源代码处理工具，例如Eclipse中的Eclipse AST，它可以用于表示和处理Java语言的语法结构，用户可以通过扩展其功能创建自己的文法规则，实现对源代码的分析。

## 优点

解释器模式的优点主要如下：

1. 易于改变和扩展文法。由于解释器模式中使用类表示语言的文法规则，因此可以通过继承等机制来改变或扩展文法。
2. 每一条文法规则都可以表示为一个类，因此可以方便地实现一个简单的语言。
3. 实现文法较为容易。在抽象语法树中每一个表达式结点类的实现方式都是相似的，这些类的代码编写都不会特别复杂，还可以通过一些工具自动生成节点类代码。
4. 增加新的解释表达式较为方便。如果用户要增加新的解释表达式只需要对应增加一个新的终结符表达式或非终结符表达式类，原有表达式类代码无须修改，符合开闭原则。

## 缺点

解释器模式的缺点主要如下：

1. 对于复杂文法难以维护。在解释器模式中每一条规则至少需要定义一个类，因此如果一个语言包含太多文法规则，类的个数会急剧增加，导致系统难以管理和维护，此时可以考虑使用语法分析程序等方式来取代解释器模式。
2. 执行效率较低。由于在解释器模式中使用了大量的循环和递归调用，因此在解释较为复杂的句子时其速度很慢，而且代码的调试过程也比较复杂。

## 适用环境

在以下情况下可以考虑使用解释器模式：

1. 可以将一个需要解释执行的语言中的句子表示为一棵抽象语法树。
2. 一些重复出现的问题可以用一种简单的语言进行表达。
3. 一个语言的文法较为简单。对于复杂的文法，解释器模式中的文法类层次结构将变得很庞大而无法管理，此时最好使用语法分析程序生成器。
4. 执行效率不是关键问题。高效的解释器通常不是通过直接解释抽象语法树来实现的，而是需要将它们转换成其他形式，使用解释器模式的执行效率并不高。