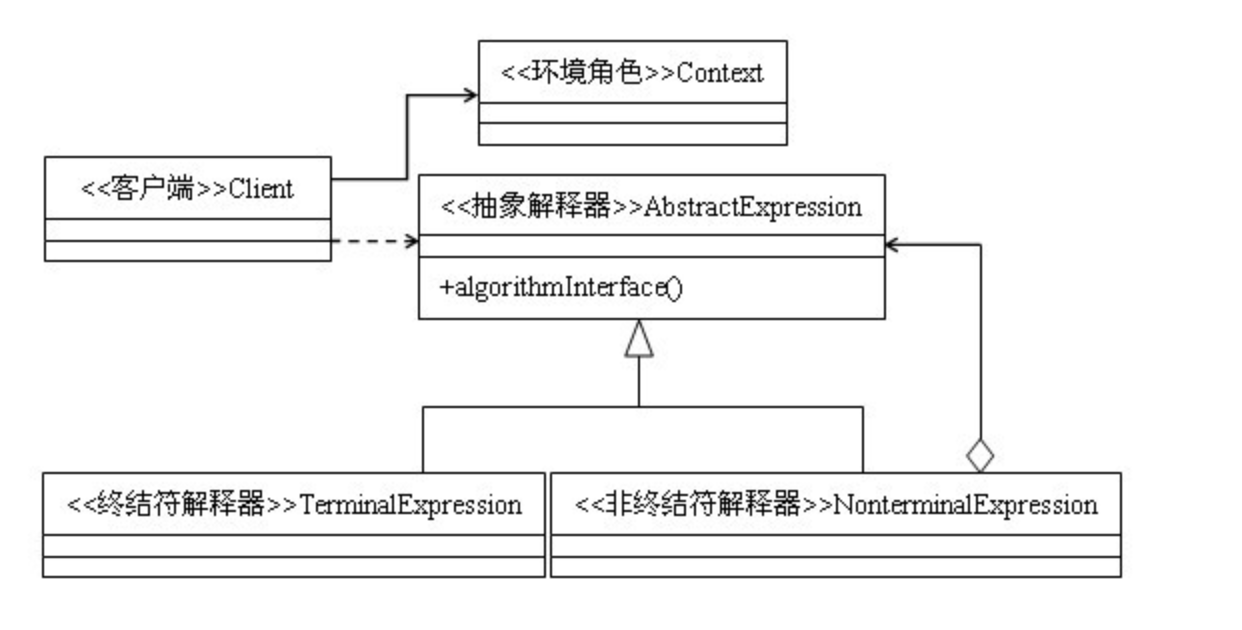
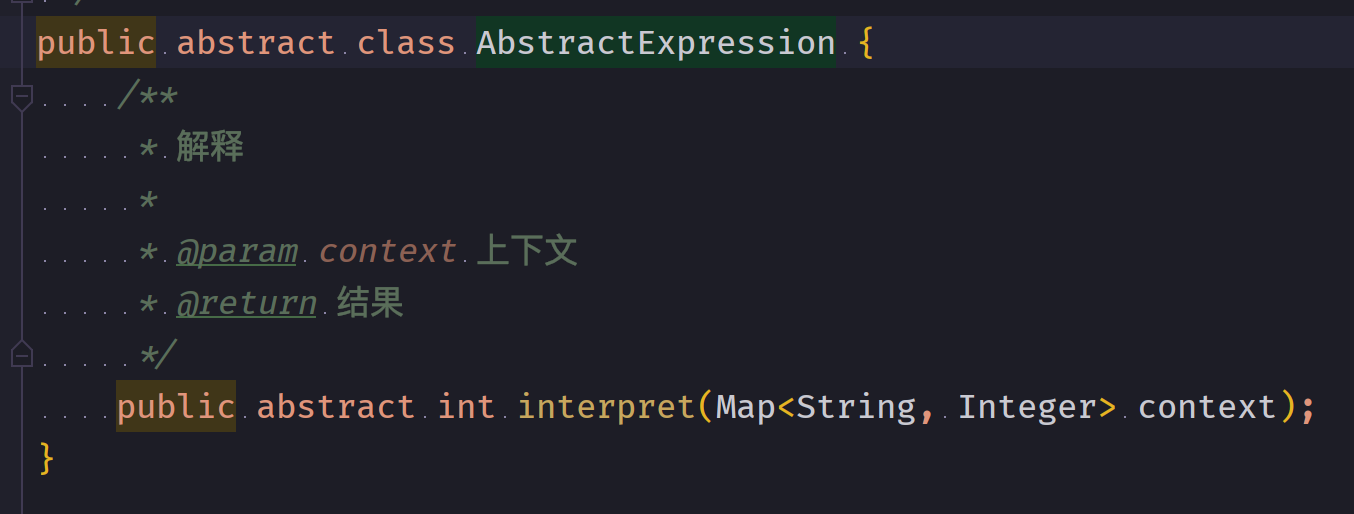
1. 解释器模式：给定一种语言，定义他的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中句子。
2. 应用场景
   1. 应用可以将一个需要解释执行的语言中的句子表示为一个抽象语法树
   2. 一些重复出现的问题可以用一种简单的语言来表达。比如加减乘除四则运算，但是公式每次都不同，有时是a+b-c\*d，有时是a\*b+c-d，等等等等个，公式千变万化，但是都是由加减乘除四个非终结符来连接的，这时我们就可以使用解释器模式
   3. 一个简单语法需要解释的场景
3. 类图



* 1. AbstractExpression抽象表达式类，声明一个所有具体表达式都要实现的抽象接口（或者抽象类），接口中主要是一个interpret()方法，称为解释操作。具体解释任务由它的各个实现类来完成，具体的解释器分别由终结符解释器TerminalExpression和非终结符解释器NonterminalExpression完成
  2. TerminalExpression终结符表达式：实现与文法中的元素相关联的解释操作，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。终结符一半是文法中的运算单元，比如有一个简单的公式R=R1+R2，在里面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
  3. NonTerminalExpression非终结符表达式：文法中的每条规则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
  4. Context环境角色：这个角色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，我们给R1赋值100，给R2赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了

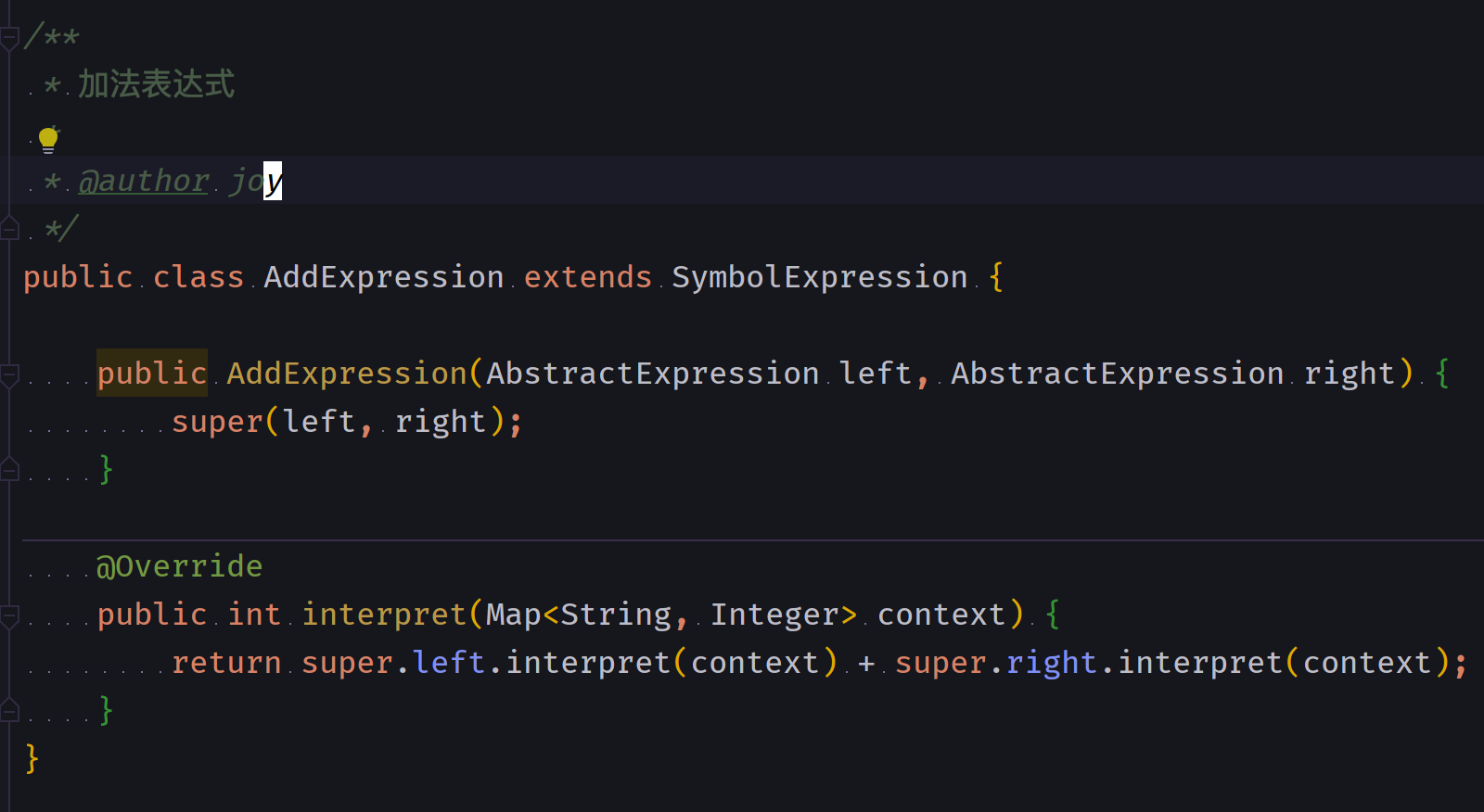
1. 举例
   1. 抽象表达式类



* 1. 抽象运算符表达式



* 1. 加法运算符解释



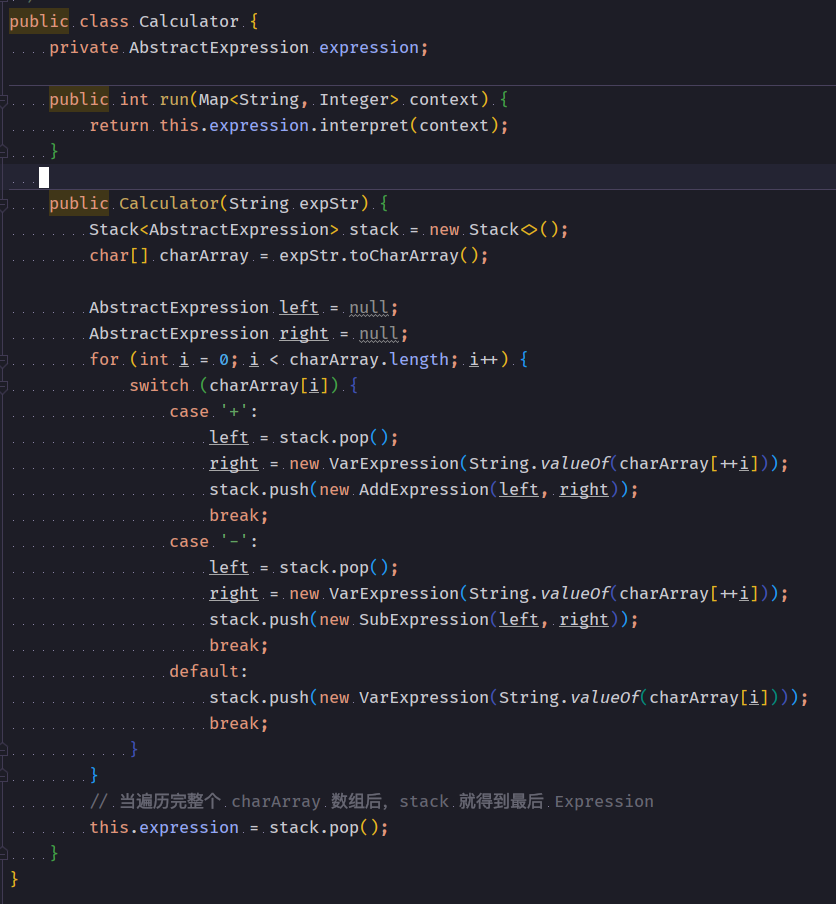
* 1. 减法运算符解释



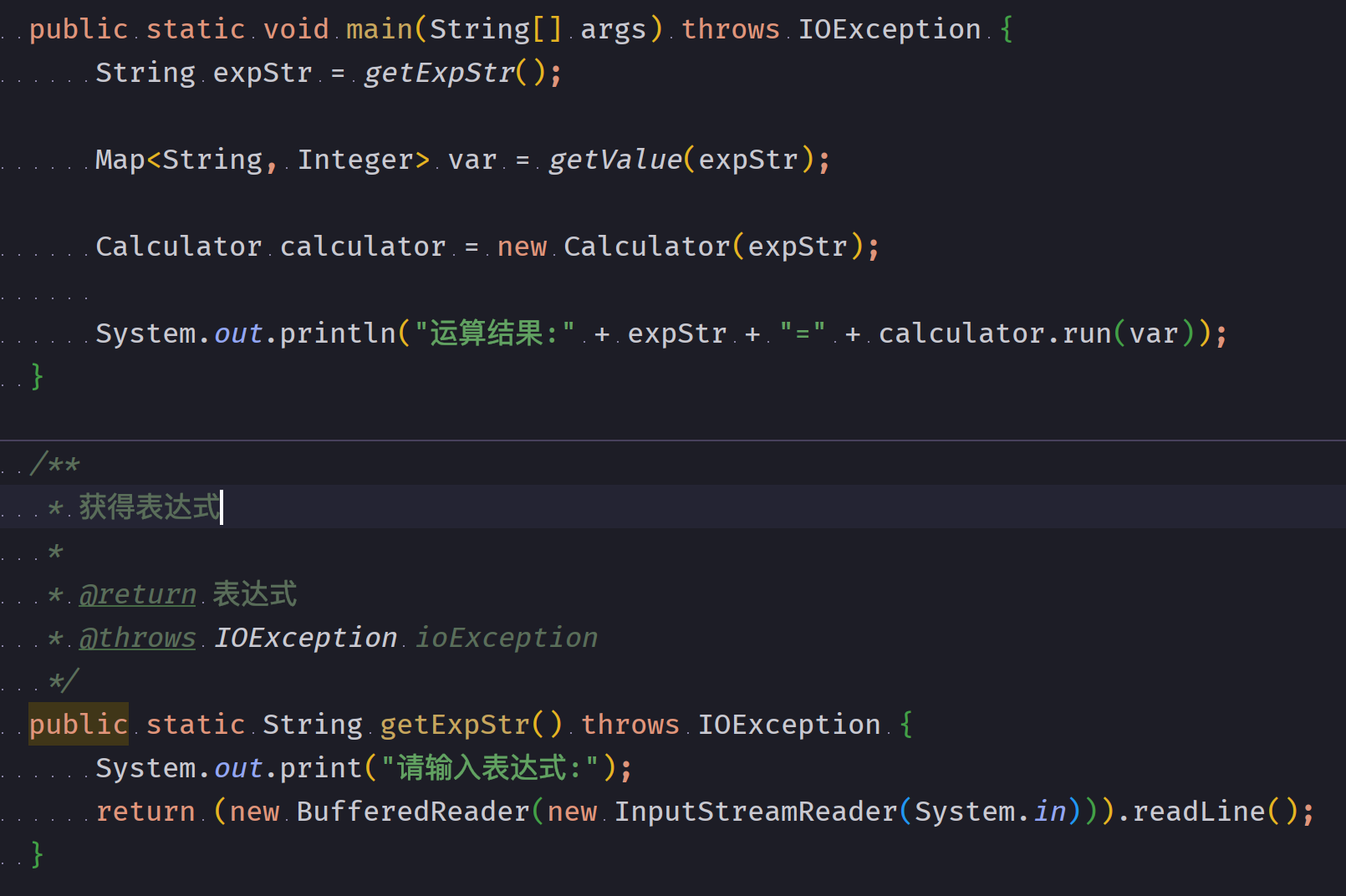
* 1. 变量解释器



* 1. 计算器

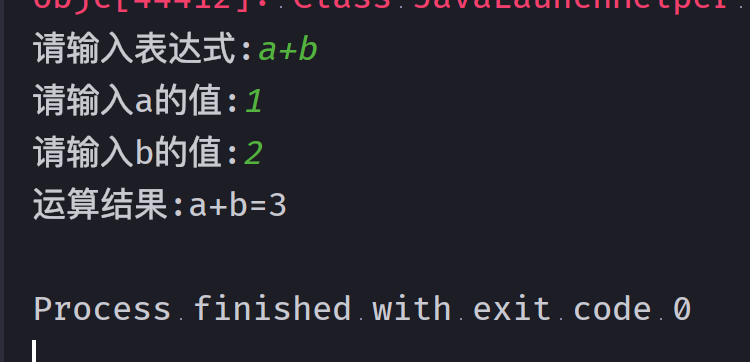


* 1. 测试Client

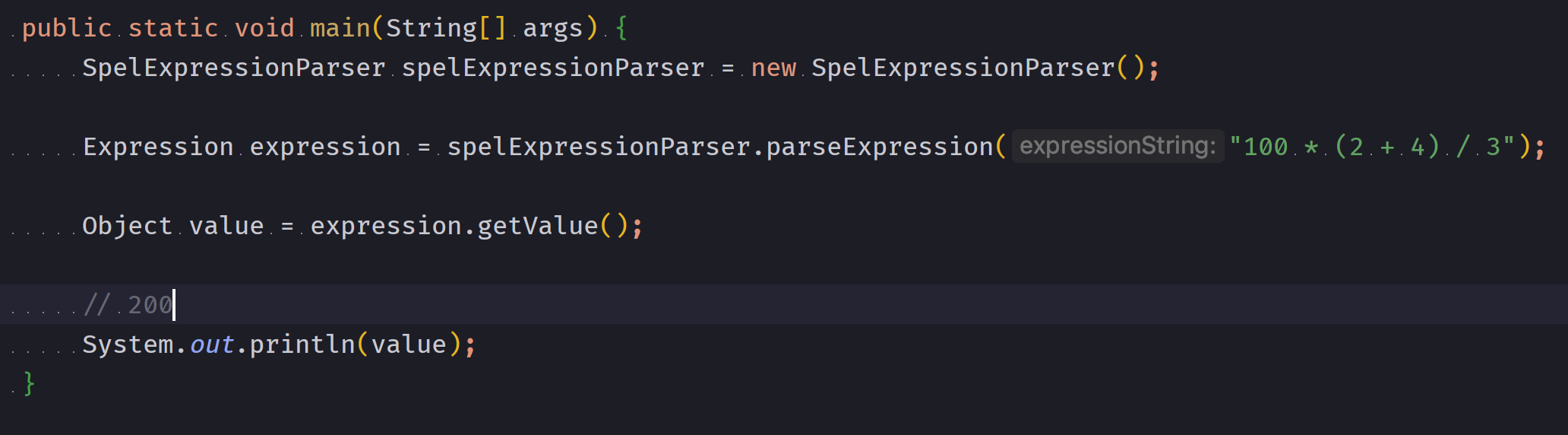




* 1. 运行



1. 解释器模式在Spring中org.springframework.expression.spel.standard.SpelExpressionParser中有使用到



1. 解释器模式的注意事项和细节
   1. 当有一个语言需要解释执行，可将该语言中的句子表示为一个抽象语法树，就可以考虑使用解释器模式，让程 序具有良好的扩展性
   2. 应用场景:编译器、运算表达式计算、正则表达式、机器人等
   3. 使用解释器可能带来的问题:解释器模式会引起类膨胀、解释器模式采用递归调用方法，将会导致调试非常复杂、效率可能降低.
   4. 解释器模式真的是一个比较少用的模式，因为对它的维护实在是太麻烦了，想象一下，一坨一坨的非终结符解释器，假如不是事先对文法的规则了如指掌，或者是文法特别简单，则很难读懂它的逻辑。解释器模式在实际的系统开发中使用的很少，因为他会引起效率、性能以及维护等问题