解释器模式

题目链接

解释器模式-数学表达式

基本概念

解释器模式（Interpreter Pattern）是一种行为型设计模式，它定义了一个语言的文法，并且建立一个【解释器】来解释该语言中的句子。

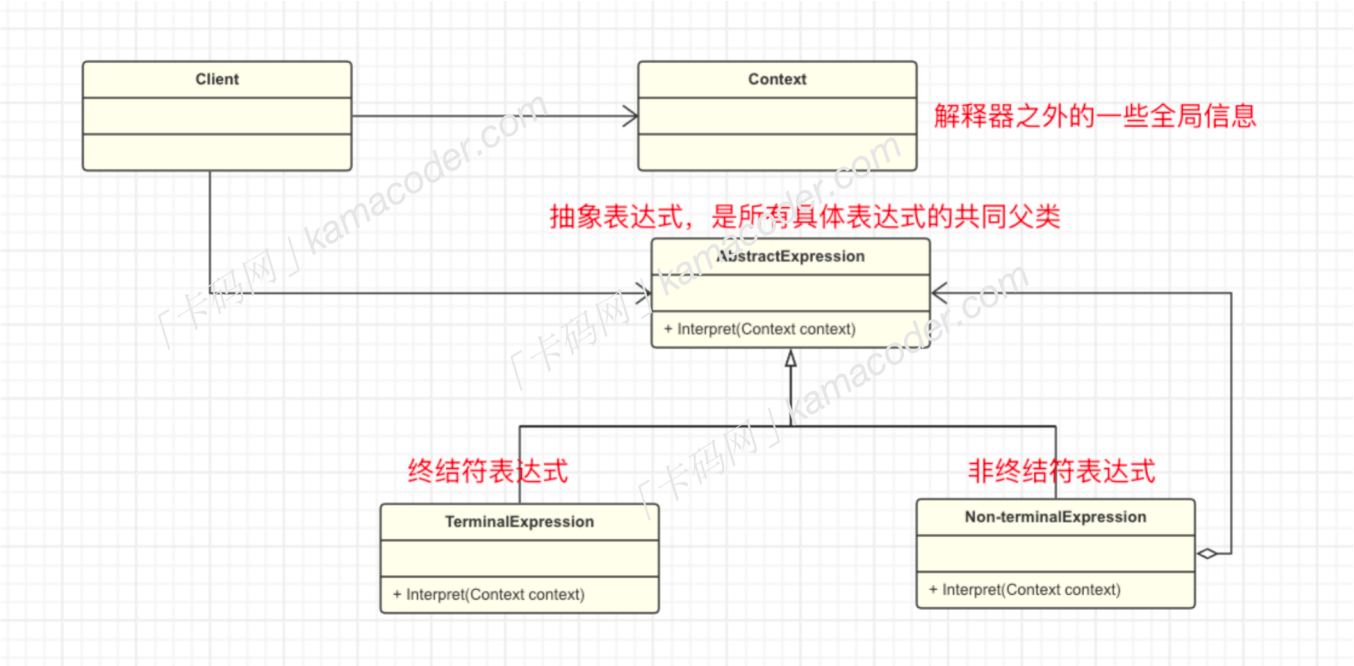
比如说SQL语法、正则表达式，这些内容比较简短，但是表达的内容可不仅仅是字面上的那些符号，计算机想要理解这些语法，就需要解释这个语法规则，因此解释器模式常用于实现编程语言解释器、正则表达式处理等场景。

组成结构

解释器模式主要包含以下几个角色：

1. **抽象表达式（Abstract Expression）：** 定义了解释器的接口，包含了解释器的方法 `interpret`。
2. **终结符表达式（Terminal Expression）：** 在语法中不能再分解为更小单元的符号。
3. **非终结符表达式（Non-terminal Expression）：** 文法中的复杂表达式，它由终结符和其他非终结符组成。
4. **上下文（Context）：** 包含解释器之外的一些全局信息，可以存储解释器中间结果，也可以用于向解释器传递信息。

> *举例来说，表达式"3 + 5 * 2"，数字"3"和"5"，"2"是终结符，而运算符"+","*"都需要两个操作数，属于非终结符。*



简易实现

1. 创建抽象表达式接口： 定义解释器的接口，声明一个 `interpret` 方法，用于解释语言中的表达式。

```
// 抽象表达式接口
public interface Expression {
    int interpret();
}
```

2. 创建具体的表达式类： 实现抽象表达式接口，用于表示语言中的具体表达式。

```
public class TerminalExpression implements Expression {
    private int value;

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

    @Override
    public int interpret() {
        return value;
    }
}
```

3. 非终结符表达式：抽象表达式的一种，用于表示语言中的非终结符表达式，通常包含其他表达式。

```
public class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return left.interpret() + right.interpret();
    }
}
```

4. 上下文：包含解释器需要的一些全局信息或状态。

```
public class Context {
    // 可以在上下文中存储一些全局信息或状态
}
```

5. 客户端：构建并组合表达式，然后解释表达式。

```
public class Main {
    public static void main(String[] args) {
        Context context = new Context();

        Expression expression = new AddExpression(
                new TerminalExpression(1),
                new TerminalExpression(2)
        );

        int result = expression.interpret();
        System.out.println("Result: " + result);
    }
}
```

使用场景

当需要解释和执行特定领域或业务规则的语言时，可以使用解释器模式。例如，SQL解释器、正则表达式解释器等。但是需要注意的是解释器模式可能会导致类的层次结构较为复杂，同时也可能不够灵活，使用要慎重。

本题代码

```
import java.util.Scanner;
import java.util.Stack;

// 抽象表达式接口
interface Expression {
    int interpret();
}

// 终结符表达式类 - 数字
class NumberExpression implements Expression {
    private int number;

    public NumberExpression(int number) {
        this.number = number;
    }

    @Override
    public int interpret() {
        return number;
    }
}

// 非终结符表达式类 - 加法
class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return left.interpret() + right.interpret();
    }
}

// 非终结符表达式类 - 乘法
class MultiplyExpression implements Expression {
    private Expression left;
    private Expression right;

    public MultiplyExpression(Expression left, Expression right) {
```

```
        this.left = left;
        this.right = right;
    }

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

// 上下文类
class Context {
    private Stack<Expression> expressionStack = new Stack<>();

    public void pushExpression(Expression expression) {
        expressionStack.push(expression);
    }

    public Expression popExpression() {
        return expressionStack.pop();
    }
}

public class Main{
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Context context = new Context();

        // 处理用户输入的数学表达式
        while (scanner.hasNextLine()) {
            String userInput = scanner.nextLine();
            Expression expression = parseExpression(userInput);
            if (expression != null) {
                context.pushExpression(expression);
                System.out.println(expression.interpret());
            } else {
                System.out.println("Invalid expression.");
            }
        }

        scanner.close();
    }

    // 解析用户输入的数学表达式并返回相应的抽象表达式类
    private static Expression parseExpression(String userInput) {
        try {
```

```
            Stack<Expression> expressionStack = new Stack<>();
            char[] tokens = userInput.toCharArray();

            for (int i = 0; i < tokens.length; i++) {
                char token = tokens[i];

                if (Character.isDigit(token)) {
                    expressionStack.push(new
NumberExpression(Character.getNumericValue(token)));

                    // 如果下一个字符不是数字，且栈中有两个以上的元素，说明可以进行运
算
                    if (i + 1 < tokens.length &&
!Character.isDigit(tokens[i + 1]) && expressionStack.size() >= 2) {
                        Expression right = expressionStack.pop();
                        Expression left = expressionStack.pop();
                        char operator = tokens[i + 1];

                        if (operator == '+') {
                            expressionStack.push(new AddExpression(left,
right));
                        } else if (operator == '*') {
                            expressionStack.push(new
MultiplyExpression(left, right));
                        }

                        i++; // 跳过下一个字符，因为已经处理过了
                    }
                } else {
                    return null;
                }
            }

            return expressionStack.pop();
        } catch (Exception e) {
            return null;
        }
    }
}
```

其他语言版本

C++

```
#include <iostream>
#include <sstream>
#include <stack>
#include <vector>
#include <stdexcept>
#include <iterator>
#include <regex>

// 抽象表达式类
class Expression {
public:
    virtual int interpret() = 0;
    virtual ~Expression() {}
};

// 终结符表达式类 - 数字
class NumberExpression : public Expression {
private:
    int value;

public:
    NumberExpression(int val) : value(val) {}

    int interpret() override {
        return value;
    }
};

// 非终结符表达式类 - 加法操作
class AddExpression : public Expression {
private:
    Expression* left;
    Expression* right;

public:
    AddExpression(Expression* l, Expression* r) : left(l), right(r) {}

    int interpret() override {
        return left->interpret() + right->interpret();
    }
};
```

```
// 非终结符表达式类 - 乘法操作
class MultiplyExpression : public Expression {
private:
    Expression* left;
    Expression* right;

public:
    MultiplyExpression(Expression* l, Expression* r) : left(l), right(r) {}

    int interpret() override {
        return left->interpret() * right->interpret();
    }
};

// 非终结符表达式类 - 操作符
class OperatorExpression : public Expression {
private:
    std::string oper;

public:
    OperatorExpression(const std::string& op) : oper(op) {}

    int interpret() override {
        throw std::runtime_error("OperatorExpression does not support
interpretation");
    }

    std::string getOperator() const {
        return oper;
    }
};

// 解析表达式字符串
int parseExpression(const std::string& expressionStr) {
    std::istringstream iss(expressionStr);
    std::vector<std::string> elements(std::istream_iterator<std::string>
{iss}, std::istream_iterator<std::string>());

    std::stack<Expression*> stack;

    for (const auto& element : elements) {
        if (std::regex_match(element, std::regex("\\d+"))) {
            stack.push(new NumberExpression(std::stoi(element)));
        } else if (element == "+" || element == "*") {
            stack.push(new OperatorExpression(element));
```

```
        } else {
            throw std::invalid_argument("Invalid element in expression: " +
element);
        }
    }

    while (stack.size() > 1) {
        Expression* right = stack.top();
        stack.pop();
        Expression* operatorExp = stack.top();
        stack.pop();
        Expression* left = stack.top();
        stack.pop();

        if (auto* opExp = dynamic_cast<OperatorExpression*>(operatorExp)) {
            std::string op = opExp->getOperator();
            if (op == "+") {
                stack.push(new AddExpression(left, right));
            } else if (op == "*") {
                stack.push(new MultiplyExpression(left, right));
            }
        } else {
            throw std::invalid_argument("Invalid operator type in
expression");
        }
    }

    int result = stack.top()->interpret();
    delete stack.top();
    return result;
}

int main() {
    std::vector<std::string> input_lines;
    std::string line;

    while (std::getline(std::cin, line) && !line.empty()) {
        input_lines.push_back(line);
    }

    for (size_t i = 0; i < input_lines.size(); ++i) {
        try {
            int result = parseExpression(input_lines[i]);
            std::cout << result << std::endl;
        } catch (const std::exception& e) {
            std::cout << "Error - " << e.what() << std::endl;
```

```
        }
    }

    return 0;
}
```

Python

```python
# 抽象表达式类
class Expression:
    def interpret(self):
        pass

# 终结符表达式类 - 数字
class NumberExpression(Expression):
    def __init__(self, value):
        self.value = int(value)

    def interpret(self):
        return self.value

# 非终结符表达式类 - 加法操作
class AddExpression(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() + self.right.interpret()

# 非终结符表达式类 - 乘法操作
class MultiplyExpression(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() * self.right.interpret()

# 客户端代码
def parse_expression(expression_str):
    elements = expression_str.split()
    stack = []

    for element in elements:
        if element.isdigit():
```

```
            stack.append(NumberExpression(element))
        elif element == '+':
            if len(stack) < 2:
                raise ValueError("Invalid expression format")
            right = stack.pop()
            left = stack.pop()
            stack.append(AddExpression(left, right))
        elif element == '*':
            if len(stack) < 2:
                raise ValueError("Invalid expression format")
            right = stack.pop()
            left = stack.pop()
            stack.append(MultiplyExpression(left, right))
        else:
            raise ValueError(f"Invalid element in expression: {element}")

    if len(stack) != 1:
        raise ValueError("Invalid expression format")

    return str(stack.pop().interpret())

# 从标准输入读取输入
input_lines = []
while True:
    try:
        line = input().strip()
        if not line:
            break
        input_lines.append(line)
    except EOFError:
        break

# 输出计算结果到标准输出
for i, input_line in enumerate(input_lines, start=1):
    try:
        result = parse_expression(input_line)
        print(f"Case {i}: {result}")
    except ValueError as e:
        print(f"Case {i}: Error - {e}")
```

Go

```
package main

import (
    "bufio"
```

```
    "fmt"
    "os"
    "regexp"
    "strconv"
    "strings"
)

// 抽象表达式类
type Expression interface {
    interpret() int
}

// 终结符表达式类 - 数字
type NumberExpression struct {
    value int
}

func NewNumberExpression(val int) *NumberExpression {
    return &NumberExpression{value: val}
}

func (n *NumberExpression) interpret() int {
    return n.value
}

// 非终结符表达式类 - 加法操作
type AddExpression struct {
    left  Expression
    right Expression
}

func NewAddExpression(left, right Expression) *AddExpression {
    return &AddExpression{left, right}
}

func (a *AddExpression) interpret() int {
    return a.left.interpret() + a.right.interpret()
}

// 非终结符表达式类 - 乘法操作
type MultiplyExpression struct {
    left  Expression
    right Expression
}

func NewMultiplyExpression(left, right Expression) *MultiplyExpression {
```

```
    return &MultiplyExpression{left, right}
}

func (m *MultiplyExpression) interpret() int {
    return m.left.interpret() * m.right.interpret()
}

// 非终结符表达式类 - 操作符
type OperatorExpression struct {
    oper string
}

func NewOperatorExpression(op string) *OperatorExpression {
    return &OperatorExpression{oper: op}
}

func (o *OperatorExpression) interpret() int {
    panic("OperatorExpression does not support interpretation")
}

func (o *OperatorExpression) getOperator() string {
    return o.oper
}

// 解析表达式字符串
func parseExpression(expressionStr string) (int, error) {
    elements := strings.Fields(expressionStr)
    stack := make([]Expression, 0)

    for _, element := range elements {
        if regexp.MustCompile(`^\d+$`).MatchString(element) {
            val, _ := strconv.Atoi(element)
            stack = append(stack, NewNumberExpression(val))
        } else if element == "+" || element == "*" {
            stack = append(stack, NewOperatorExpression(element))
        } else {
            return 0, fmt.Errorf("Invalid element in expression: %s",
element)
        }
    }

    for len(stack) > 1 {
        right := stack[len(stack)-1]
        stack = stack[:len(stack)-1]
        operatorExp := stack[len(stack)-1]
        stack = stack[:len(stack)-1]
```

```
            left := stack[len(stack)-1]
            stack = stack[:len(stack)-1]

            if opExp, ok := operatorExp.(*OperatorExpression); ok {
                op := opExp.getOperator()
                if op == "+" {
                    stack = append(stack, NewAddExpression(left, right))
                } else if op == "*" {
                    stack = append(stack, NewMultiplyExpression(left, right))
                }
            } else {
                return 0, fmt.Errorf("Invalid operator type in expression")
            }
        }

        result := stack[0].interpret()
        return result, nil
    }

    func main() {
        scanner := bufio.NewScanner(os.Stdin)

        for scanner.Scan() {
            expression := scanner.Text()
            if expression == "" {
                continue
            }
            result, err := parseExpression(expression)
            if err != nil {
                fmt.Printf("Error - %s\n", err)
            } else {
                fmt.Println(result)
            }
        }

        if err := scanner.Err(); err != nil {
            fmt.Println("Error reading standard input:", err)
        }
    }
```