基本的动态脚本执行引擎

王思成

Nov 05, 2025

动态脚本在现代软件开发中无处不在。从游戏中的技能脚本到网页中的 JavaScript，再到办公软件中的宏和持续集成中的 Pipeline 脚本，这些功能都依赖于动态脚本执行引擎。动态脚本的核心价值在于其灵活性：它允许程序在运行时改变逻辑，无需重新编译整个应用程序。这带来了可扩展性，使用户或开发者能够定制和扩展功能；支持热更新，可以在线修复问题或更新玩法；以及促进了领域特定语言（DSL）的发展，为特定场景创造更高效的工具。本文旨在带领读者从理论到实践，深入理解动态脚本引擎的内部机制。在理论层面，我们将探讨脚本引擎的三大核心阶段：词法分析、语法分析和解释执行。在实践层面，我们将使用 Python 语言实现一个微型脚本引擎，能够执行四则运算、变量赋值和条件判断，例如处理像 `x = 10 + 2 * (3 - 1); if (x > 15) { y = 1; } else { y = 0; }` 这样的脚本。

1　理论基石：脚本引擎的三大核心阶段

一个脚本引擎的工作流程可以概括为三个核心阶段：词法分析、语法分析和解释执行。词法分析负责将源代码字符串转换为一系列有意义的「单词」或 Token。每个 Token 包含类型（如标识符、数字、运算符、关键字）和值（如变量名「x」、数字「10」）。这一过程类似于人类阅读时识别单词，它使用有限自动机（DFA）作为理论模型来识别 Token 序列。例如，源代码 `position = 10 + 20` 会被转换为 Token 列表：`[IDENTIFIER(position), OPERATOR(=), NUMBER(10), OPERATOR(+), NUMBER(20)]`。

语法分析阶段则根据预定义的语法规则，将 Token 流组织成抽象语法树（AST）。AST 是一种树形数据结构，它忽略源代码中的标点等细节，只保留程序的逻辑结构。上下文无关文法（CFG）常用于定义语言的语法规则，而递归下降法是一种简单直观的语法分析方法，适合手动实现。例如，对于表达式 `10 + 2 * 3`，AST 会确保乘法运算符的优先级高于加法，通过树结构正确表示计算顺序。

解释执行是最终阶段，它遍历 AST 并执行其中蕴含的操作。有两种主要策略：遍历解释和编译到字节码。遍历解释直接访问 AST 节点并执行相应操作，简单易实现；编译到字节码则先将 AST 编译成中间指令，由虚拟机执行，效率更高。在本文中，我们将采用遍历解释的方式，为每个节点类型定义执行逻辑，同时维护一个作用域来管理变量。

2　动手实现：构建微型脚本引擎

我们将使用 Python 语言实现一个微型脚本引擎，支持数字、字符串、布尔值数据类型，以及四则运算、逻辑比较、变量赋值、if-else 条件语句和语句块。实现过程分为五个步骤：定义语言语法、实现词法分析器、实现语法分析器、实现解释器，以及组装测试。

首先，我们定义迷你脚本语言的语法。它支持基本的表达式和语句，例如变量赋值使用 = 运算符，条件语句使用

if-else 结构，表达式可以包含加减乘除和比较操作。数据类型包括数字（如 10）、字符串（如 hello）和布尔值（如 true 和 false）。

接下来，实现词法分析器（Lexer）。词法分析器的输入是源代码字符串，输出是 Token 列表。我们定义 Token 类型枚举，包括 IDENTIFIER、NUMBER、OPERATOR、KEYWORD 等。在实现中，我们使用循环逐个字符扫描源代码，识别标识符、数字、运算符和关键字，并跳过空白字符。例如，以下 Python 代码展示了词法分析器的核心逻辑：

```
class TokenType:
    IDENTIFIER = 'IDENTIFIER'
    NUMBER = 'NUMBER'
    OPERATOR = 'OPERATOR'
    KEYWORD = 'KEYWORD'

class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

def lexer(source):
    tokens = []
    pos = 0
    while pos < len(source):
        char = source[pos]
        if char.isspace():
            pos += 1
            continue
        elif char.isalpha():
            start = pos
            while pos < len(source) and source[pos].isalnum():
                pos += 1
            value = source[start:pos]
            if value in ['if', 'else', 'true', 'false']:
                tokens.append(Token(TokenType.KEYWORD, value))
            else:
                tokens.append(Token(TokenType.IDENTIFIER, value))
        elif char.isdigit():
            start = pos
            while pos < len(source) and source[pos].isdigit():
                pos += 1
            value = int(source[start:pos])
```

```
            tokens.append(Token(TokenType.NUMBER, value))
        elif char in '=+-*/(){}<>!=':
            if char == '=' and pos + 1 < len(source) and source[pos+1] == '=':
                tokens.append(Token(TokenType.OPERATOR, '=='))
                pos += 2
            else:
                tokens.append(Token(TokenType.OPERATOR, char))
                pos += 1
        else:
            raise SyntaxError(f"未知字符␣:␣{char}")
    return tokens
```

在这段代码中，我们定义了 Token 类和 TokenType 枚举来表示 Token 的类型和值。lexer 函数通过遍历源代码字符串，使用条件分支识别不同字符类型。例如，当遇到字母时，它读取连续的字母数字字符，判断是否为关键字或标识符；遇到数字时，读取连续数字并转换为整数；遇到运算符时，处理单字符或多字符情况（如 ==）。错误处理部分在遇到未知字符时抛出异常，为后续完善错误信息奠定基础。

语法分析器（Parser）的输入是 Token 列表，输出是 AST 根节点。我们定义 AST 节点类型，如 NumberLiteral、BinaryExpression、AssignmentStatement、IfStatement 等。使用递归下降法，我们编写解析函数来处理表达式和语句。例如，parse_expression 函数处理表达式，考虑运算符优先级；parse_statement 处理语句；parse_program 解析整个程序。以下代码展示了部分实现：

```
class ASTNode:
    pass

class NumberLiteral(ASTNode):
    def __init__(self, value):
        self.value = value

class BinaryExpression(ASTNode):
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right

class AssignmentStatement(ASTNode):
    def __init__(self, identifier, expression):
        self.identifier = identifier
        self.expression = expression

class IfStatement(ASTNode):
    def __init__(self, condition, then_branch, else_branch):
```

```
        self.condition = condition
        self.then_branch = then_branch
        self.else_branch = else_branch

class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def current_token(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos]
        return None

    def eat(self, token_type):
        token = self.current_token()
        if token and token.type == token_type:
            self.pos += 1
            return token
        else:
            raise SyntaxError(f"期望 {token_type}，但得到 {token.type if token else 'EOF'}
                ↪ ")

    def parse_expression(self):
        left = self.parse_term()
        while self.current_token() and self.current_token().type == TokenType.OPERATOR
            ↪ and self.current_token().value in ['+', '-']:
            operator = self.eat(TokenType.OPERATOR).value
            right = self.parse_term()
            left = BinaryExpression(left, operator, right)
        return left

    def parse_term(self):
        left = self.parse_factor()
        while self.current_token() and self.current_token().type == TokenType.OPERATOR
            ↪ and self.current_token().value in ['*', '/']:
            operator = self.eat(TokenType.OPERATOR).value
            right = self.parse_factor()
            left = BinaryExpression(left, operator, right)
        return left
```

```
58
    def parse_factor(self):
60      token = self.current_token()
        if token.type == TokenType.NUMBER:
62          self.eat(TokenType.NUMBER)
            return NumberLiteral(token.value)
64      elif token.type == TokenType.IDENTIFIER:
            self.eat(TokenType.IDENTIFIER)
66          return Identifier(token.value)
        elif token.value == '(':
68          self.eat(TokenType.OPERATOR)
            expr = self.parse_expression()
70          self.eat(TokenType.OPERATOR)
            return expr
72      else:
            raise SyntaxError("期望表达式")
74
    def parse_statement(self):
76      token = self.current_token()
        if token.type == TokenType.KEYWORD and token.value == 'if':
78          return self.parse_if_statement()
        elif token.type == TokenType.IDENTIFIER and self.pos + 1 < len(self.tokens) and
            ↪ self.tokens[self.pos+1].value == '=':
80          return self.parse_assignment()
        else:
82          raise SyntaxError("未知语句")

84  def parse_if_statement(self):
        self.eat(TokenType.KEYWORD)
86      self.eat(TokenType.OPERATOR)
        condition = self.parse_expression()
88      self.eat(TokenType.OPERATOR)
        self.eat(TokenType.OPERATOR)
90      then_branch = self.parse_statement()
        self.eat(TokenType.OPERATOR)
92      else_branch = None
        if self.current_token() and self.current_token().type == TokenType.KEYWORD and
            ↪ self.current_token().value == 'else':
94          self.eat(TokenType.KEYWORD)
            self.eat(TokenType.OPERATOR)
```

```
            else_branch = self.parse_statement()
            self.eat(TokenType.OPERATOR)
        return IfStatement(condition, then_branch, else_branch)

    def parse_assignment(self):
        identifier = self.eat(TokenType.IDENTIFIER).value
        self.eat(TokenType.OPERATOR)
        expression = self.parse_expression()
        return AssignmentStatement(identifier, expression)

    def parse_program(self):
        statements = []
        while self.current_token():
            statements.append(self.parse_statement())
        return statements
```

在这段代码中，我们定义了多种 AST 节点类来表示程序结构。Parser 类使用递归下降法解析 Token 流，通过 parse_expression、parse_term 和 parse_factor 函数确保运算符优先级正确。例如，乘除法在 parse_term 中处理，优先级高于加减法；parse_if_statement 函数解析条件语句，处理 if 关键字、条件表达式和分支语句。eat 方法用于验证和消耗 Token，提供基本错误检查。

解释器（Interpreter）负责执行 AST，它使用作用域来管理变量。作用域是一个字典，存储变量名和值的映射。我们编写 evaluate 函数，递归遍历 AST 节点并执行操作。例如：

```
class Scope:
    def __init__(self, parent=None):
        self.variables = {}
        self.parent = parent

    def get(self, name):
        if name in self.variables:
            return self.variables[name]
        elif self.parent:
            return self.parent.get(name)
        else:
            raise NameError(f"未定义变量␣:␣{name}")

    def set(self, name, value):
        self.variables[name] = value

def evaluate(node, scope):
    if isinstance(node, NumberLiteral):
```

```
        return node.value
    elif isinstance(node, Identifier):
        return scope.get(node.value)
    elif isinstance(node, BinaryExpression):
        left_val = evaluate(node.left, scope)
        right_val = evaluate(node.right, scope)
        if node.operator == '+':
            return left_val + right_val
        elif node.operator == '-':
            return left_val - right_val
        elif node.operator == '*':
            return left_val * right_val
        elif node.operator == '/':
            return left_val / right_val
        elif node.operator == '>':
            return left_val > right_val
        elif node.operator == '<':
            return left_val < right_val
        elif node.operator == '==':
            return left_val == right_val
        else:
            raise RuntimeError(f"未知运算符：{node.operator}")
    elif isinstance(node, AssignmentStatement):
        value = evaluate(node.expression, scope)
        scope.set(node.identifier, value)
        return value
    elif isinstance(node, IfStatement):
        condition_val = evaluate(node.condition, scope)
        if condition_val:
            return evaluate(node.then_branch, scope)
        else:
            if node.else_branch:
                return evaluate(node.else_branch, scope)
            else:
                return None
    else:
        raise RuntimeError(f"未知节点类型：{type(node)}")
```

evaluate 函数根据节点类型执行相应操作。对于 BinaryExpression，它递归计算左右子树的值，然后应用运算符；对于 AssignmentStatement，它计算表达式值并存入作用域；对于 IfStatement，它评估条件并决

定执行哪个分支。作用域支持简单的嵌套，通过 parent 属性实现变量查找链。
最后，我们组装整个引擎，编写 run 函数串联所有步骤：

```
def run(source):
    tokens = lexer(source)
    parser = Parser(tokens)
    program = parser.parse_program()
    scope = Scope()
    for statement in program:
        result = evaluate(statement, scope)
    return scope
```

测试时，我们可以执行脚本如 `x = 10 + 2 * (3 - 1); if (x > 15) { y = 1; } else { y = 0; }`，并检查变量 x 和 y 的值。例如，计算 x 时，先计算 `3 - 1` 得 2，再乘以 2 得 4，然后加 10 得 14，由于 14 不大于 15，因此 y 被赋值为 0。

3 进阶与优化

在基本实现基础上，我们可以添加错误处理来提升用户体验。例如，在词法分析、语法分析和运行时阶段，记录行号和列号，提供详细的错误信息。这有助于开发者快速定位问题，例如在语法错误时指出具体位置。
性能优化方面，可以考虑编译到字节码。字节码是一种紧凑的中间表示，由虚拟机执行，比直接解释 AST 更高效。实现时，我们需要定义字节码指令集（如 LOAD_CONST、STORE_NAME、BINARY_ADD 等），并将 AST 编译成字节码序列。虚拟机使用栈来执行指令，例如对于表达式 `a + b`，字节码可能是 LOAD a、LOAD b、ADD，这减少了递归开销。
Just-In-Time（JIT）编译是更高级的优化，它将热点代码直接编译成本地机器码，大幅提升执行速度。但这涉及复杂的技术，如动态代码生成和优化，通常用于生产级引擎。
功能扩展上，我们可以添加循环语句（如 while）、函数定义与调用、内置函数（如 print）和复杂数据类型（如数组和对象）。这些扩展需要修改词法分析、语法分析和解释器，以支持新语法和语义。例如，添加函数调用时，需引入调用栈和作用域链。
通过本文的旅程，我们深入理解了动态脚本引擎的核心组件：词法分析将源代码转换为 Token 流，语法分析构建 AST，解释执行遍历 AST 实现功能。我们的微型引擎虽然简单，但揭示了真实引擎如 V8（JavaScript）、CPython 和 Lua 的基本原理。这些成熟引擎处理了更多边界情况，并集成了高级优化。
鼓励读者在此基础上继续探索，例如实现字节码虚拟机或添加新功能。进一步学习资源包括《编译原理》（龙书）和《Crafting Interpreters》书籍。现在，您已经拥有了理解更复杂脚本引擎的基石，去构建属于自己的「微缩宇宙」吧！