深入理解并实现基本的脚本语言解释器

杨子凡

Oct 26, 2025

1 从零开始，用几百行代码构建你的迷你「Python」或「JavaScript」

在编程世界中，我们经常使用像 Python 或 JavaScript 这样的脚本语言来执行动态任务。想象一下，当我们写下 `a = 10 + 2 * 3; print(a);` 这样一行简单的代码时，计算机是如何一步步理解并执行它的呢？这背后隐藏着解释器的神奇工作原理。本文将带你从零开始，深入探讨并亲手实现一个基本的脚本语言解释器。通过这个过程，你不仅能深化对编程语言内部机制的理解，还能掌握构建一个支持变量、算术运算、条件语句、循环和函数的动态类型语言的核心技能。我们称之为 `MiniScript`，并使用 Python 作为实现语言，因为其语法简洁，易于表达思想，但所涉及的原理是通用的。

造一个解释器可能听起来像在重复发明轮子，但这个过程却能极大地提升我们对编程语言本质的认识。以一个简单的代码片段为例：`a = 10 + 2 * 3; print(a);`。这行代码涉及赋值、算术运算和输出，计算机需要将其从文本转化为实际行动。通过实现解释器，我们可以直观地看到代码是如何被解析和执行的，这就像学习一门新语言时，理解其语法规则一样重要。解释器无处不在，从 Python 到 JavaScript，许多流行语言都依赖解释执行。更重要的是，一个基础的解释器并不复杂，其核心逻辑可以用几百行代码实现，这让它成为学习编译器设计和语言实现的绝佳起点。本文的目标是构建一个支持变量、算术运算、比较运算、条件语句、循环语句和函数的动态类型语言 `MiniScript`。最终，你将获得一个可运行的迷你解释器，并能通过动手实践加深对关键概念的理解。

2 背景知识：解释器与编译器的区别

在深入实现之前，我们需要先理解解释器和编译器的核心区别。编译器将源代码一次性翻译成另一种语言，通常是机器码，生成独立的可执行文件。这就像将一本中文书完整翻译成英文出版，整个过程是离线的。而解释器则直接读取源代码，边解析边执行，类似于同声传译，一边听一边现场翻译。对于 `MiniScript`，我们将采用解释器的方式，其工作流程可以概括为：源代码首先经过词法分析，被拆分成令牌流；然后通过语法分析构建抽象语法树；最后解释执行这棵树。这个流程确保了代码的逐步理解和执行，为后续实现奠定了基础。

3 第一步：词法分析 —— 从字符流到令牌流

词法分析是解释器的第一个阶段，目标是将源代码字符串拆分成一个个有意义的「单词」，即令牌。例如，代码 `a = 10 + 2;` 会被转换为令牌序列：`[Identifier('a'), Assign(), Number(10), Plus(), Number(2), Semicolon()]`。这个过程类似于人类阅读时识别单词和标点。实现词法分析器时，我们使用正则表达式来高

效匹配字符序列。首先，定义令牌类型，如 IDENTIFIER、NUMBER、PLUS 等，这些类型代表了代码中的基本元素。然后，构建一个 Lexer 类，该类包含属性如 input_text（存储源代码字符串）和 position（跟踪当前处理位置），以及方法如 next_token()（用于获取下一个令牌）、skip_whitespace()（跳过空白字符）、read_number()（读取数字）和 read_identifier()（读取标识符）。这些方法协同工作，逐步扫描输入文本并生成令牌流。

以下是一个简单的 Lexer 类核心代码示例，我们使用 Python 实现。代码中，Token 类用于表示令牌的类型和值，Lexer 类通过遍历字符来生成令牌。例如，next_token 方法会检查当前字符，如果是数字，则调用 read_number 来解析整数；如果是字母，则调用 read_identifier 来解析变量名。这个过程确保了代码的每个部分都被正确分类。

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

class Lexer:
    def __init__(self, input_text):
        self.input_text = input_text
        self.position = 0
        self.current_char = self.input_text[self.position] if self.input_text else None

    def advance(self):
        self.position += 1
        if self.position >= len(self.input_text):
            self.current_char = None
        else:
            self.current_char = self.input_text[self.position]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def read_number(self):
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def read_identifier(self):
```

```
        result = ''
        while self.current_char is not None and (self.current_char.isalnum() or self.
            ↪ current_char == '_'):
            result += self.current_char
            self.advance()
        return result

    def next_token(self):
        while self.current_char is not None:
            if self.current_char.isspace():
                self.skip_whitespace()
                continue
            if self.current_char.isdigit():
                return Token('NUMBER', self.read_number())
            if self.current_char.isalpha() or self.current_char == '_':
                identifier = self.read_identifier()
                if identifier in KEYWORDS: # KEYWORDS 是预定义的关键字字典，如 'if', 'while'
                    return Token(identifier.upper(), identifier)
                return Token('IDENTIFIER', identifier)
            if self.current_char == '=':
                self.advance()
                return Token('ASSIGN', '=')
            if self.current_char == '+':
                self.advance()
                return Token('PLUS', '+')
            # 类似处理其他操作符和符号
            self.advance()
        return Token('EOF', None)
```

在这段代码中，Lexer 类通过 advance 方法移动当前位置，并利用辅助函数如 read_number 和 read_identifier 来提取数字和标识符。next_token 方法是核心，它循环处理字符，跳过空白后，根据字符类型返回相应的令牌。例如，遇到数字时，它会累积所有连续数字字符并转换为整数；遇到字母时，它会读取整个标识符，并检查是否为关键字。这种设计确保了词法分析的高效性和准确性，为后续语法分析提供了干净的输入。

4 第二步：语法分析 —— 构建抽象语法树

语法分析是解释器的第二个阶段，目标是将令牌流转换为抽象语法树，这是一种树形数据结构，能体现代码的层次结构和语义。以表达式 a = 10 + 2 * 3 为例，其 AST 可能包含一个赋值节点，左子节点是变量 a，右子节点是一个二元操作节点，表示加法，其中左操作数是数字 10，右操作数是另一个二元操作节点，表示乘法（左

操作数 2，右操作数 3)。这种结构清晰地反映了运算符的优先级和结合性，例如乘法在加法之前执行。AST 是后续解释执行的基础，因为它将线性代码转换为可遍历的树形形式。

为了定义 MiniScript 的语法，我们使用类似 BNF 的表示法来描述规则。例如，程序由多个语句组成，语句可以是表达式语句、if 语句或 while 语句；表达式可以进一步分解为赋值、相等比较、项、因子等。具体规则包括：程序是零个或多个语句的序列；语句包括表达式语句（以分号结尾）、if 语句和 while 语句；表达式从赋值开始，赋值可以是标识符后跟等号和表达式，或者相等比较；相等比较又由比较操作构成，比较操作由项和关系运算符组成；项由因子和加减运算符组成；因子由一元操作和乘除运算符组成；一元操作可以是逻辑非或负号，后跟一元操作或基本元素；基本元素包括数字、字符串、标识符或括号内的表达式。这些规则确保了语法的递归和层次性。

实现语法分析器时，我们采用递归下降分析法，这是一种直观且适合手写的方法。Parser 类包含属性如 tokens（存储令牌列表）和 current（当前令牌索引），以及一系列方法对应每条语法规则，如 parse_program、parse_statement 和 parse_expression。核心技巧是使用前瞻令牌来决定解析路径，例如在解析表达式时，查看下一个令牌类型以选择正确的规则。

以下是一个 Parser 类的核心代码示例，展示如何解析表达式和赋值语句。代码中，parse_expression 方法从赋值开始处理，而 parse_assignment 检查是否为标识符后跟等号，否则递归处理相等比较。

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

    def peek(self):
        return self.tokens[self.current] if self.current < len(self.tokens) else None

    def advance(self):
        if self.current < len(self.tokens):
            self.current += 1

    def parse_program(self):
        statements = []
        while self.peek() and self.peek().type != 'EOF':
            statements.append(self.parse_statement())
        return Program(statements) # Program 是 AST 根节点类

    def parse_statement(self):
        token = self.peek()
        if token.type == 'IF':
            return self.parse_if_statement()
        elif token.type == 'WHILE':
            return self.parse_while_statement()
```

```
        else:
            expr = self.parse_expression()
            if self.peek() and self.peek().type == 'SEMICOLON':
                self.advance()
            return ExpressionStatement(expr) # ExpressionStatement 是语句节点类

    def parse_expression(self):
        return self.parse_assignment()

    def parse_assignment(self):
        left = self.parse_equality()
        if self.peek() and self.peek().type == 'ASSIGN':
            self.advance()
            value = self.parse_assignment()
            return Assign(left, value) # Assign 是赋值节点类
        return left

    def parse_equality(self):
        left = self.parse_comparison()
        while self.peek() and self.peek().type in ('EQ', 'NEQ'): # EQ 和 NEQ 表示 == 和
            ↪ !=
            operator = self.peek().type
            self.advance()
            right = self.parse_comparison()
            left = BinaryOp(left, operator, right) # BinaryOp 是二元操作节点类
        return left

    # 类似实现 parse_comparison、parse_term 等方法
```

在这段代码中，Parser 类通过 peek 方法查看当前令牌，而不消耗它，advance 方法移动到下一个令牌。parse_expression 从顶层开始解析，逐步下降到更具体的规则。例如，在 parse_assignment 中，如果遇到等号令牌，则构建赋值节点；否则，返回解析的表达式。这种方法确保了语法规则的正确应用，并构建出完整的 AST。通过递归下降，我们可以高效地处理嵌套结构，如表达式中的括号，从而为解释执行阶段提供结构化的输入。

5 第三步：解释执行 —— 让 AST「活」起来

解释执行是解释器的核心阶段，我们遍历 AST 并执行计算。为了清晰处理不同类型的节点，我们使用访问者模式，这允许为每种 AST 节点定义特定的执行逻辑。例如，数字节点直接返回值，而二元操作节点需要递归计算左右子树后再执行运算。同时，执行环境用于存储变量和函数定义，本质上是一个字典，支持作用域链，通过维

护环境栈或让环境持有父环境引用来实现嵌套作用域。

Interpreter 类是解释执行的主体，包含属性如 environment（当前执行环境），以及一系列 visit_XXX 方法，如 visit_BinaryOp、visit_Assign 和 visit_Variable。这些方法对应 AST 节点类型，确保每个节点都能被正确解释。例如，在访问字面量节点时，visit_Number 直接返回数字值；在访问二元操作节点时，visit_BinaryOp 先递归访问左子树和右子树获取值，然后根据操作符执行相应运算，如加法或比较。对于变量赋值，visit_Assign 将值写入环境；对于变量引用，visit_Variable 从环境中读取值。控制流节点如 if 语句和 while 循环，则通过计算条件表达式值来决定执行路径。

以下是一个 Interpreter 类的核心代码示例，展示如何实现基本节点的访问方法。代码中，我们假设环境是一个字典，支持设置和获取变量值。

```
class Environment:
    def __init__(self, parent=None):
        self.vars = {}
        self.parent = parent

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

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

class Interpreter:
    def __init__(self):
        self.environment = Environment()

    def visit(self, node):
        method_name = f'visit_{type(node).__name__}'
        method = getattr(self, method_name, self.generic_visit)
        return method(node)

    def generic_visit(self, node):
        raise Exception(f 没有 visit_{type(node).__name__} 方法)

    def visit_Number(self, node):
        return node.value
```

```
31
    def visit_BinaryOp(self, node):
33      left = self.visit(node.left)
        right = self.visit(node.right)
35      if node.operator == 'PLUS':
            return left + right
37      elif node.operator == 'MINUS':
            return left - right
39      elif node.operator == 'MULTIPLY':
            return left * right
41      elif node.operator == 'DIVIDE':
            return left / right
43      elif node.operator == 'EQ':
            return left == right
45      # 类似处理其他操作符
        else:
47          raise Exception(f"未知操作符␣:␣{node.operator}")

49  def visit_Assign(self, node):
        value = self.visit(node.value)
51      self.environment.set(node.name, value)
        return value
53
    def visit_Variable(self, node):
55      return self.environment.get(node.name)

57  def visit_If(self, node):
        condition = self.visit(node.condition)
59      if condition:
            return self.visit(node.then_branch)
61      elif node.else_branch:
            return self.visit(node.else_branch)
63      return None

65  def visit_While(self, node):
        result = None
67      while self.visit(node.condition):
            result = self.visit(node.body)
69      return result
```

在这段代码中，Interpreter 类使用 visit 方法动态调用相应的节点访问方法。例如，visit_Number 直接返回节点的数值；visit_BinaryOp 先递归计算左右操作数的值，然后根据操作符执行运算，如加法或相等比较。对于赋值节点，visit_Assign 计算右侧表达式的值并存储到环境中；对于变量节点，visit_Variable 从环境中检索值。控制流节点如 If 和 While 通过条件判断来执行分支或循环体。这种设计确保了 AST 的逐步执行，并处理了语句和表达式的区别：表达式产生值，而语句执行操作但不一定返回值。通过环境管理，我们实现了变量的动态存储和检索，为更复杂的特性如函数打下了基础。

6 进阶特性：实现函数

函数是编程语言中的核心抽象，允许我们将代码封装为可重用的单元。在 MiniScript 中，函数被视为一种可调用对象，包含参数列表和函数体（一个代码块 AST）。实现函数涉及两个主要步骤：定义和调用。在定义时，我们解析 function 关键字，创建一个 Function 对象，该对象存储参数、函数体和声明时的环境（用于实现闭包），并将其作为值存入当前环境。在调用时，我们计算实参的值，创建新的函数作用域（以函数定义时的环境为父环境），绑定形参和实参，然后在新环境中执行函数体。

以下是一个函数实现的核心代码示例。我们扩展 Interpreter 类来处理函数定义和调用，并引入 Function 类来表示函数对象。

```
class Function:
    def __init__(self, params, body, env):
        self.params = params
        self.body = body
        self.env = env # 定义时的环境，用于闭包

class Interpreter:
    # 之前的方法保持不变

    def visit_FunctionDef(self, node):
        func = Function(node.params, node.body, self.environment)
        self.environment.set(node.name, func)
        return func

    def visit_Call(self, node):
        func = self.visit(node.func)
        args = [self.visit(arg) for arg in node.args]
        if isinstance(func, Function):
            new_env = Environment(parent=func.env) # 创建新环境，父环境为定义时的环境
            for param, arg in zip(func.params, args):
                new_env.set(param, arg)
            old_env = self.environment
            self.environment = new_env
```

```
            try:
                result = self.visit(func.body)
            finally:
                self.environment = old_env
            return result
        else:
            raise Exception(f"{func} 不是可调用对象")
```

在这段代码中，visit_FunctionDef 方法处理函数定义，它创建一个 Function 实例并存储到当前环境中。visit_Call 方法处理函数调用：首先计算函数表达式和参数值，然后检查函数是否为 Function 实例。如果是，它创建一个新环境，其父环境设置为函数定义时的环境，这实现了闭包 —— 函数可以访问定义时的变量。接着，将形参和实参绑定到新环境，临时切换解释器的环境来执行函数体，最后恢复原环境。这种设计确保了函数的作用域隔离和参数传递，同时通过闭包支持了高级特性如嵌套函数。例如，如果一个函数内部定义了另一个函数，内部函数可以访问外部函数的变量，因为这保存在定义时的环境中。

通过本文的旅程，我们从字符流开始，逐步实现了词法分析、语法分析和解释执行，构建了一个完整的 MiniScript 解释器。这个过程不仅让你理解了代码如何从文本转化为行动，还深化了对 AST、作用域和访问者模式等概念的认识。现在，你应该能够运行自己的迷你解释器，并扩展其功能。未来，你可以进一步优化性能，例如引入字节码和虚拟机；添加新特性如数组、字典和类；改进错误处理；或构建标准库。我鼓励你动手实践，分享代码，并参考资源如《编程语言实现模式》来继续探索。编程语言的实现是一个充满挑战和乐趣的领域，希望本文为你打开了这扇大门。