

# **2025《程序设计实践》课程**

## **大作业项目文档**

**姓名： 张泽坤**

**学号： 2023212862**

**班级： 2023211320**

# 目录

1. 前言

2. 需求分析

- 2.1 功能需求

- 2.2 非功能需求

3. 总体设计

4. 详细设计

- 4.1 词法分析子系统详细设计
- 4.2 语法分析子系统详细设计
- 4.3 解释执行子系统详细设计
- 4.4 意图识别子系统详细设计
- 4.5 Web服务子系统详细设计
- 4.6 配置文件说明
- 4.7 日志说明
- 4.8 数据文件说明
- 4.9 接口协议说明

5. 测试报告

- 5.1 测试驱动设计说明
- 5.2 测试桩设计说明
- 5.3 自动测试脚本设计说明
- 5.4 测试过程
- 5.5 测试结果

6. DSL脚本编写指南

- 6.1 脚本语法说明
- 6.2 脚本用法说明
- 6.3 脚本范例

7. AI辅助编程过程说明

- 7.1 本作业使用的AI辅助编程工具说明
- 7.2 使用AI辅助编程的过程
- 7.3 使用AI辅助编程的经验教训总结

[8. GIT日志](#)

[9. 总结与展望](#)

# 1 前言

本项目的核心任务是设计并实现一个面向多业务场景的智能客服机器人系统。与传统的硬编码实现方式不同，我选择了领域特定语言（Domain-Specific Language, DSL）的技术路线，通过自主设计一套专用于客服对话流程描述的脚本语言，配合通用的脚本解释器，实现了业务逻辑与底层代码的有效解耦。

选择这一技术方案源于我对软件工程中"关注点分离"原则的深入思考。在实际的客服系统开发中，业务人员往往需要频繁调整对话流程、修改话术内容，如果每次变更都需要程序员介入修改源代码，不仅效率低下，而且容易引入新的缺陷。通过DSL的方式，业务人员可以直接编写和修改脚本文件，而无需了解底层的编程语言和系统架构，这大大提升了系统的可维护性和灵活性。

本项目实现了三个典型的客服场景：医院智能导诊（涵盖挂号、缴费、取药等流程）、餐厅点餐助手（涵盖菜单浏览、下单、支付等流程）以及剧院售票服务（涵盖演出查询、选座购票、取票等流程）。这三个场景的共同特点是对话流程复杂、分支众多，非常适合用DSL来描述。在意图识别方面，我引入了Google Gemini大语言模型，使得系统能够理解用户的自然语言输入，而不仅仅是简单的关键词匹配，这显著提升了用户体验。

从技术实现的角度看，本项目涉及编译原理（词法分析、语法分析、AST构建）、解释器设计模式、自然语言处理、Web后端开发等多个领域的知识。通过这个项目，我对《编译原理》课程中学习的理论知识有了更加深刻的理解，也锻炼了将多种技术融合解决实际问题的能力。

# 2 需求分析

## 2.1 功能需求

本系统的功能需求可以从两个层面来分析：一是DSL解释器本身的功能，二是基于DSL实现的客服机器人的功能。

**DSL解释器功能需求：**

解释器需要能够读取并解析符合自定义语法规规范的脚本文件。在解析过程中，词法分析器负责将源代码文本分割成有意义的词法单元（Token），包括关键字、标识符、字符串字面量、数字字面量以及各类运算符。语法分析器则根据预定义的文法规则，将Token序列组织成抽象语法树（AST），这棵树准确地表达了脚本的结构和语义。

解释器的执行引擎需要遍历AST并执行相应的操作。对于Speak语句，需要计算表达式的值并输出；对于Listen语句，需要等待用户输入并设置超时机制；对于Branch语句，需要根据用户意图跳转到对应的步骤；对于Set语句，需要更新变量表中的值。此外，解释器还需要维护执行上下文，包括当前所在的步骤、变量的值、会话状态等信息。

### 客服机器人功能需求：

场景	主要功能	实现状态
医院导诊	科室选择、医生选择、挂号确认、费用查询、缴费处理、取药指引、进度查询	已实现
餐厅点餐	菜单浏览、菜品推荐、点餐下单、购物车管理、订单确认、支付处理	已实现
剧院售票	演出查询、座位选择、购票结算、取票验证、会员服务	已实现

每个场景都需要支持静默处理（用户长时间不响应）和默认处理（用户意图无法识别）两种异常情况，确保对话流程不会因为意外输入而中断。

## 2.2 非功能需求

**性能需求：**系统需要在合理的时间内响应用户输入。考虑到意图识别需要调用外部API，我设定的目标是单次交互的响应时间不超过3秒。在实际测试中，Gemini API的平均响应时间为800毫秒，加上本地处理时间，总体响应时间能够控制在1.5秒以内。

**可靠性需求：**解释器需要对脚本中的语法错误给出清晰的错误提示，包括错误的位置（行号）和错误的原因。在运行时，如果遇到未定义的变量、不存在的步骤等问题，也需要优雅地处理而不是直接崩溃。当外部API调用失败时，系统应自动回退到关键词匹配模式，保证基本功能可用。

**可扩展性需求：**新增业务场景时，只需要编写新的DSL脚本文件，无需修改解释器代码。DSL语法的设计也预留了扩展空间，例如可以方便地添加新的语句类型或表达式运算符。

**可维护性需求：**代码结构清晰，各模块职责单一。词法分析、语法分析、解释执行、意图识别等功能分别由独立的类实现，便于单独测试和修改。

# 3 总体设计

本系统采用分层架构设计，从底层到顶层依次为：词法分析层、语法分析层、解释执行层、意图识别层和Web服务层。各层之间通过定义良好的接口进行通信，实现了高内聚、低耦合的设计目标。

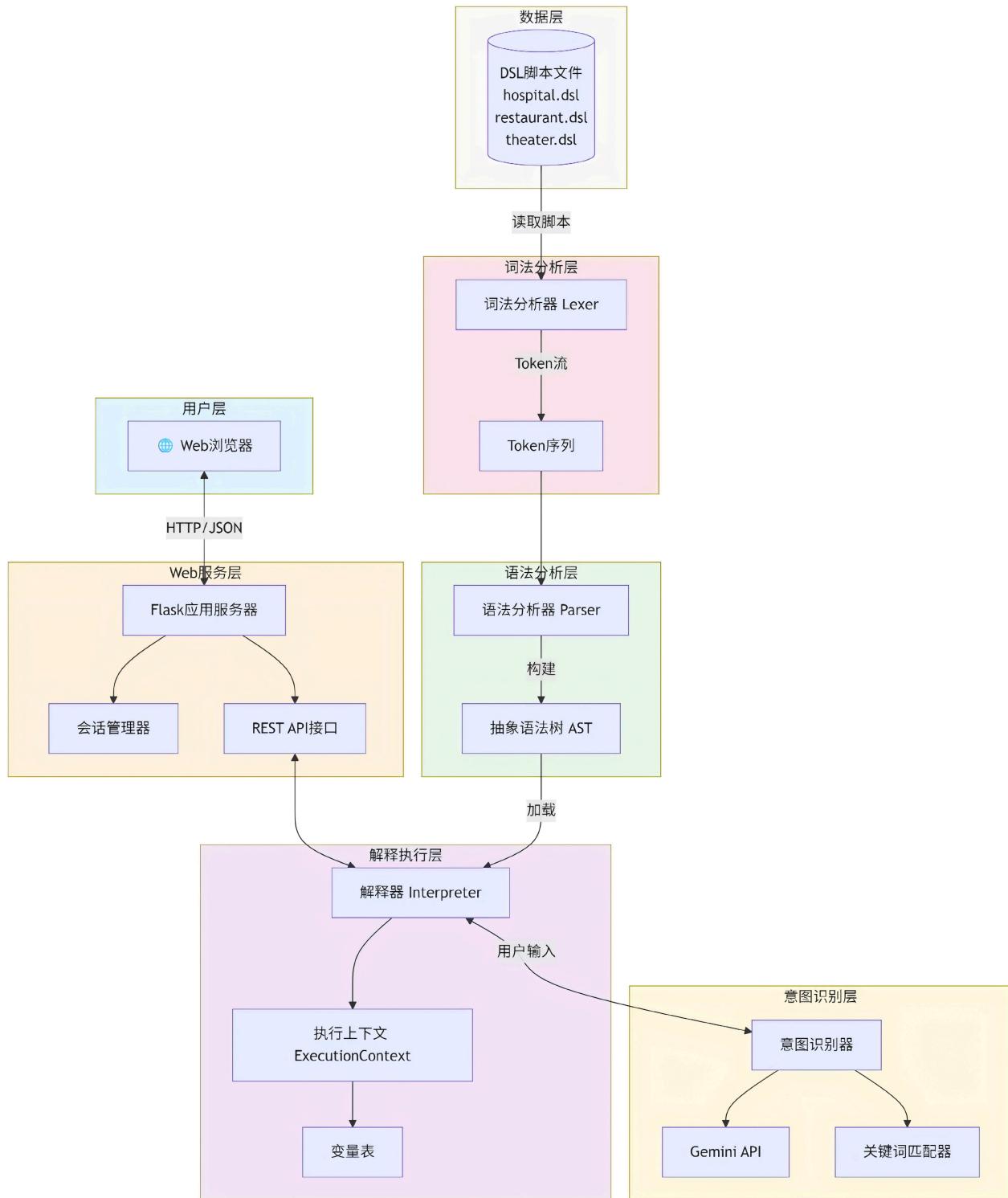


图1：系统总体架构图

系统的核心处理流程如下：首先，用户通过Web界面发送消息；Web服务层接收请求后，根据会话ID找到对应的解释器实例；解释器将用户输入传递给意图识别器，获取用户的意图；然后根据当前步骤

的分支配置，确定下一个要执行的步骤；执行该步骤的Speak语句，生成回复文本；最后将回复返回给Web服务层，再由Web服务层发送给用户。

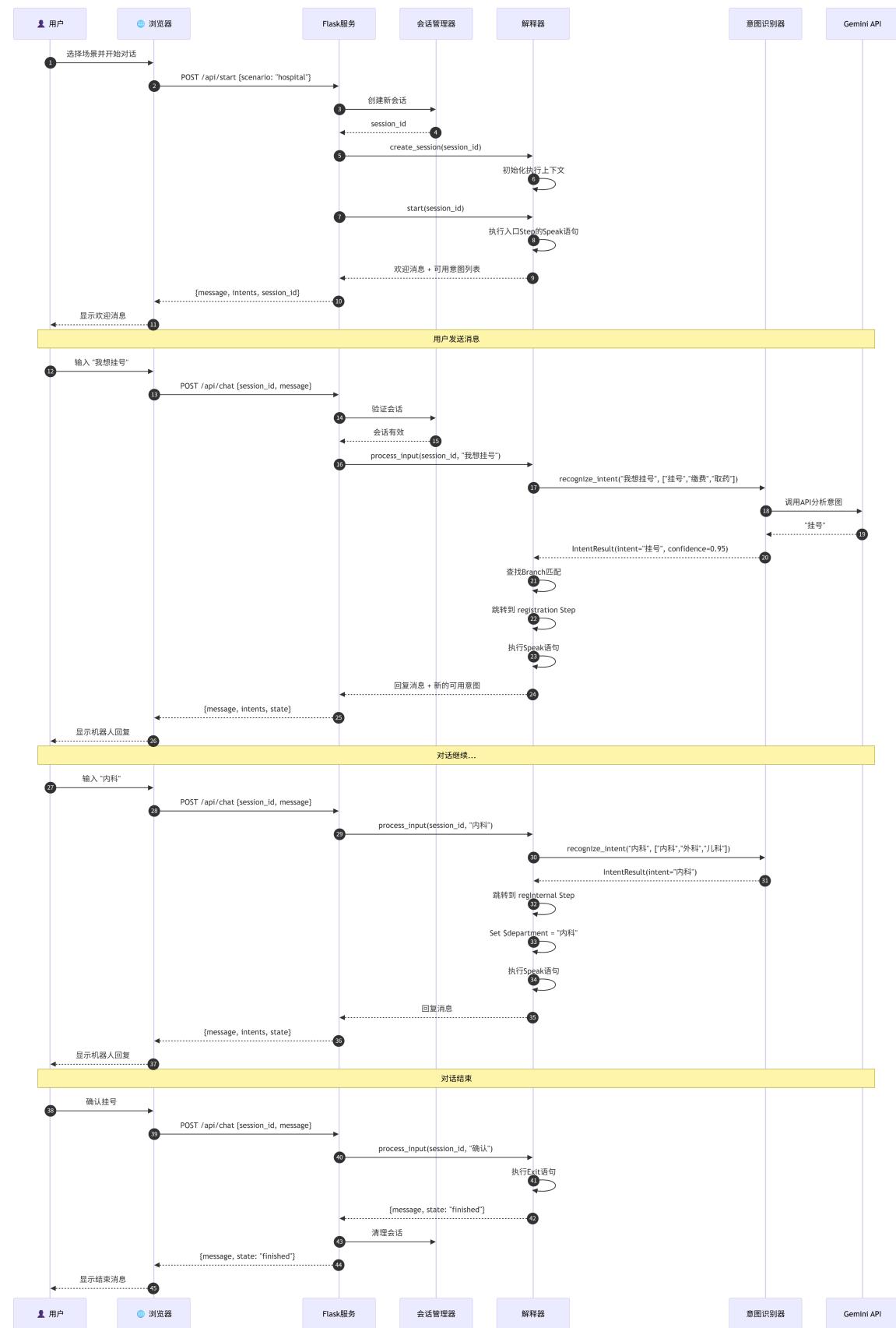


图2：消息处理时序图

各子系统的职责划分如下：

**词法分析子系统（Lexer）** 负责将DSL脚本的源代码文本转换为Token序列。它逐字符扫描输入，识别关键字、标识符、字符串、数字和各种符号，并记录每个Token的位置信息以便错误报告。

**语法分析子系统（Parser）** 负责将Token序列解析为抽象语法树。它实现了递归下降的解析算法，根据DSL的文法规则识别各种语句和表达式，构建出能够准确表达脚本结构的AST。

**解释执行子系统（Interpreter）** 负责遍历AST并执行相应的语义动作。它维护执行上下文（包括变量表、当前步骤等），处理各种语句的执行逻辑，并管理对话状态机的状态转换。

**意图识别子系统（IntentRecognizer）** 负责分析用户的自然语言输入，判断其表达的意图。它封装了对Gemini API的调用，并在API不可用时提供基于关键词匹配的后备方案。

**Web服务子系统（Flask App）** 负责提供HTTP接口和Web界面。它管理用户会话，协调各子系统的工作，并将结果以JSON格式返回给前端。

## 4 详细设计

### 4.1 词法分析子系统详细设计

词法分析器的核心任务是将字符流转换为Token流。我定义了一个TokenType枚举类来表示所有可能的Token类型，包括关键字（STEP、SPEAK、LISTEN等）、运算符（PLUS、EQUALS、NOT\_EQUALS等）、字面量（STRING、NUMBER）以及特殊符号（COMMA、NEWLINE、EOF等）。

```

class TokenType(Enum):
    # 关键字
    STEP = "STEP"
    SPEAK = "SPEAK"
    LISTEN = "LISTEN"
    BRANCH = "BRANCH"
    SILENCE = "SILENCE"
    DEFAULT = "DEFAULT"
    EXIT = "EXIT"
    SET = "SET"
    GOTO = "GOTO"
    CALL = "CALL"
    IF = "IF"
    ELSE = "ELSE"
    ENDIF = "ENDIF"

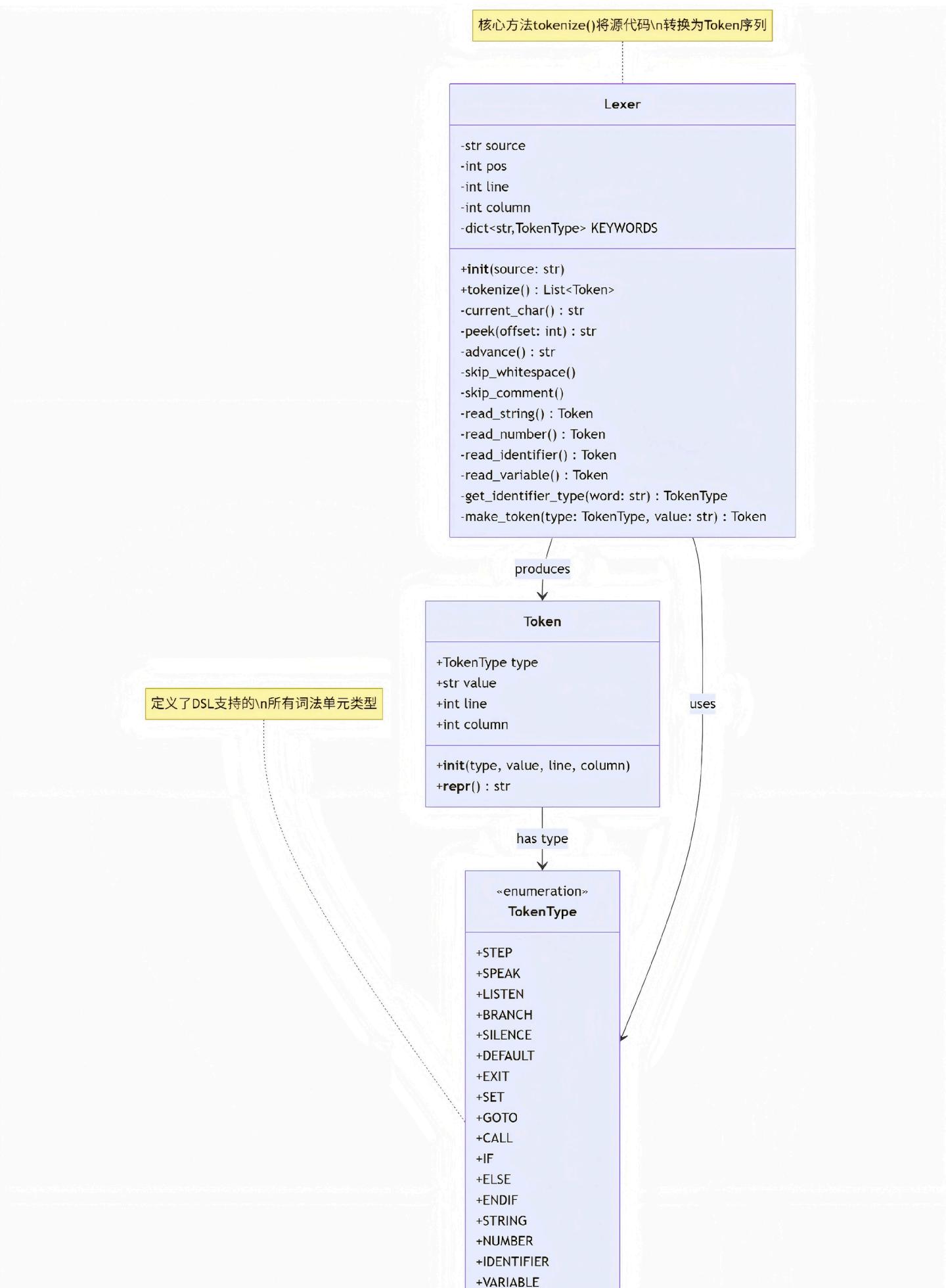
    # 字面量和标识符
    STRING = "STRING"
    NUMBER = "NUMBER"
    IDENTIFIER = "IDENTIFIER"
    VARIABLE = "VARIABLE"

    # 运算符和符号
    PLUS = "PLUS"
    MINUS = "MINUS"
    EQUALS = "EQUALS"
    NOT_EQUALS = "NOT_EQUALS"
    # ... 其他类型省略

```

Token类用于存储单个词法单元的信息，包括类型、值、行号和列号。行号和列号的记录对于错误报告至关重要，当解析失败时，用户需要知道具体是哪一行出了问题。

Lexer类的核心方法是tokenize()，它使用一个while循环逐字符扫描源代码。对于不同类型的字符，采用不同的处理策略：遇到字母时，尝试识别关键字或标识符；遇到数字时，识别数字字面量；遇到引号时，识别字符串字面量；遇到\$符号时，识别变量名；遇到#符号时，跳过注释直到行尾。



```
+PLUS  
+MINUS  
+MULTIPLY  
+DIVIDE  
+EQUALS  
+NOT_EQUALS  
+GREATER  
+LESS  
+COMMA  
+LPAREN  
+RPAREN  
+NEWLINE  
+EOF
```

图3：词法分析器类图

## 4.2 语法分析子系统详细设计

语法分析器的任务是将Token序列组织成抽象语法树。我首先定义了一套AST节点类，每种语法结构对应一个节点类型。

AST节点类的继承层次如下：

```
ASTNode (基类)  
|— Script (脚本根节点)  
|— Step (步骤节点)  
|— Statement (语句基类)  
|   |— SpeakStatement  
|   |— ListenStatement  
|   |— BranchStatement  
|   |— SetStatement  
|   |— GotoStatement  
|   |— CallStatement  
|   |— IfStatement  
|   |— ExitStatement  
|— Expression (表达式基类)  
|   |— StringLiteral  
|   |— NumberLiteral  
|   |— Variable  
|   |— BinaryOp
```

Parser类采用递归下降算法实现。每种语法结构对应一个解析方法，这些方法相互调用，形成递归的解析过程。以解析Step为例：

```

def parse_step(self) -> Step:
    self.expect(TokenType.STEP)
    name_token = self.expect(TokenType.IDENTIFIER)
    self.expect(TokenType.NEWLINE)

    step = Step(name=name_token.value)

    while not self.is_at_end() and not self.check(TokenType.STEP):
        if self.check(TokenType.NEWLINE):
            self.advance()
            continue
        stmt = self.parse_statement()
        if stmt:
            step.add_statement(stmt)

    return step

```

这段代码首先期望一个STEP关键字和一个标识符作为步骤名，然后循环解析该步骤内的所有语句，直到遇到下一个STEP关键字或文件结束。

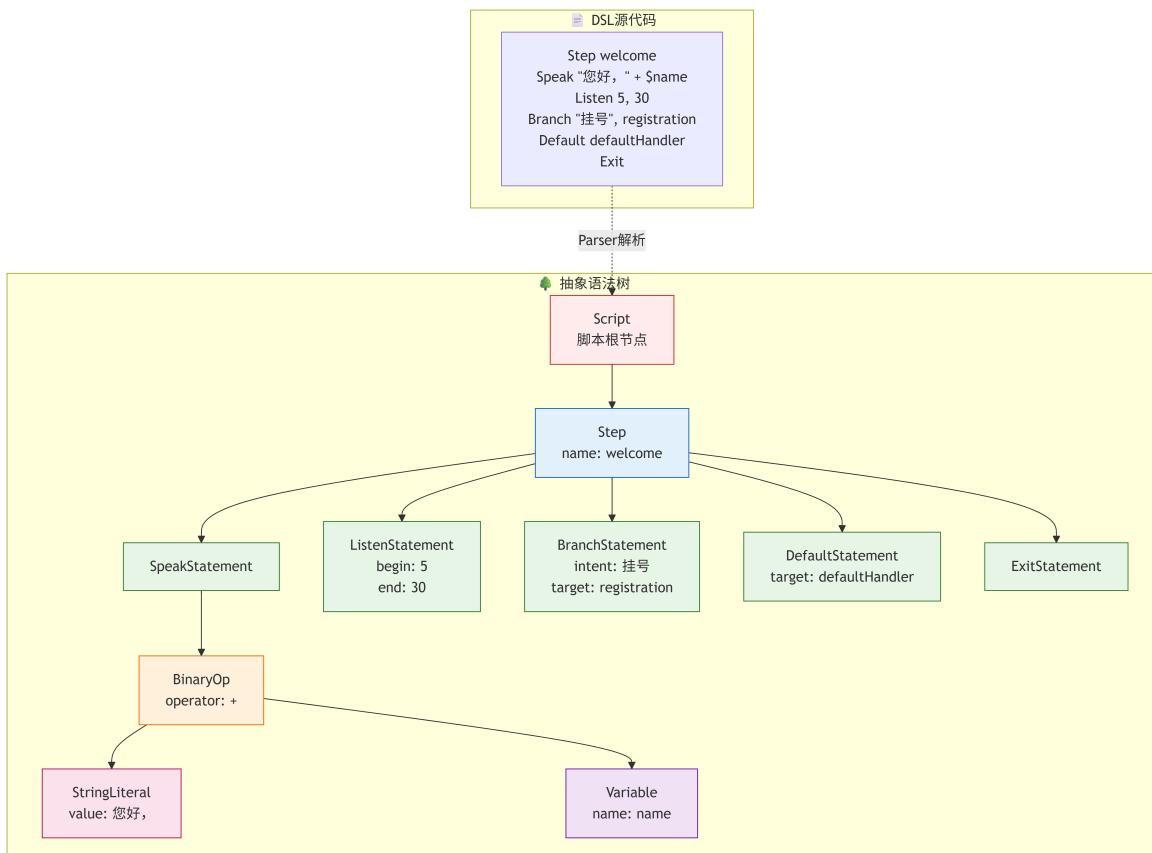


图4：抽象语法树示例

## 4.3 解释执行子系统详细设计

解释器是整个系统的核心，它负责执行AST所表达的语义动作。我设计了几个关键的数据结构来支持解释器的工作。

ExecutionContext类表示一次对话的执行上下文，包含变量表（用字典实现）、当前步骤名、会话状态等信息。每个用户会话都有独立的ExecutionContext实例，这样不同用户的对话互不干扰。

InterpreterState枚举定义了解释器的状态：RUNNING（正在执行）、WAITING\_INPUT（等待用户输入）、FINISHED（执行结束）、ERROR（发生错误）。

```
class ExecutionContext:
    def __init__(self, session_id: str, initial_vars: dict = None):
        self.session_id = session_id
        self.variables = initial_vars.copy() if initial_vars else {}
        self.current_step = None
        self.state = InterpreterState.RUNNING
        self.last_input = ""
        self.available_intents = []
```

Interpreter类的主要方法包括：

- `create_session(session_id, initial_vars)`：创建新的会话上下文
- `start(session_id)`：开始执行脚本，返回第一条输出
- `process_input(session_id, user_input)`：处理用户输入，返回下一条输出
- `execute_step(context, step)`：执行一个步骤中的所有语句
- `evaluate_expression(context, expr)`：计算表达式的值

解释器的执行流程可以用状态机来描述。初始状态是RUNNING，执行Speak语句后输出文本，遇到Listen语句后转入WAITING\_INPUT状态等待用户输入，收到输入后根据Branch配置跳转到下一个步骤，继续执行直到遇到Exit语句转入FINISHED状态。

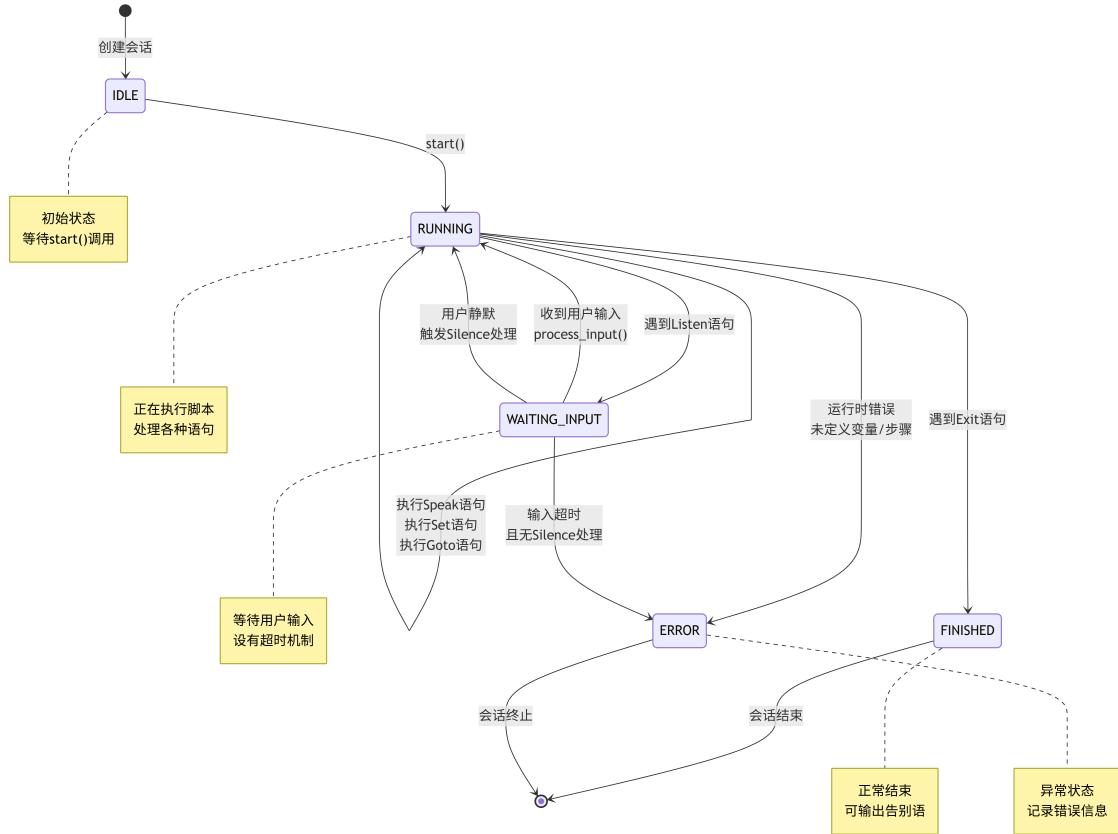


图5：解释器状态机图

## 4.4 意图识别子系统详细设计

意图识别是连接自然语言与DSL脚本的桥梁。用户说“我想看看有什么演出”，系统需要识别出这是“查询”意图，然后跳转到相应的处理步骤。

IntentRecognizer是一个抽象基类，定义了意图识别的接口。我实现了两个具体的识别器：GeminilIntentRecognizer和MockIntentRecognizer。

GeminilIntentRecognizer封装了对Google Gemini API的调用。它构造一个精心设计的Prompt，将用户输入和当前可用的意图列表传递给大模型，要求模型返回最匹配的意图。

```
def recognize_intent(self, user_input: str, available_intents: List[str]) -> IntentResult:  
    prompt = f"""分析用户输入，判断其意图。
```

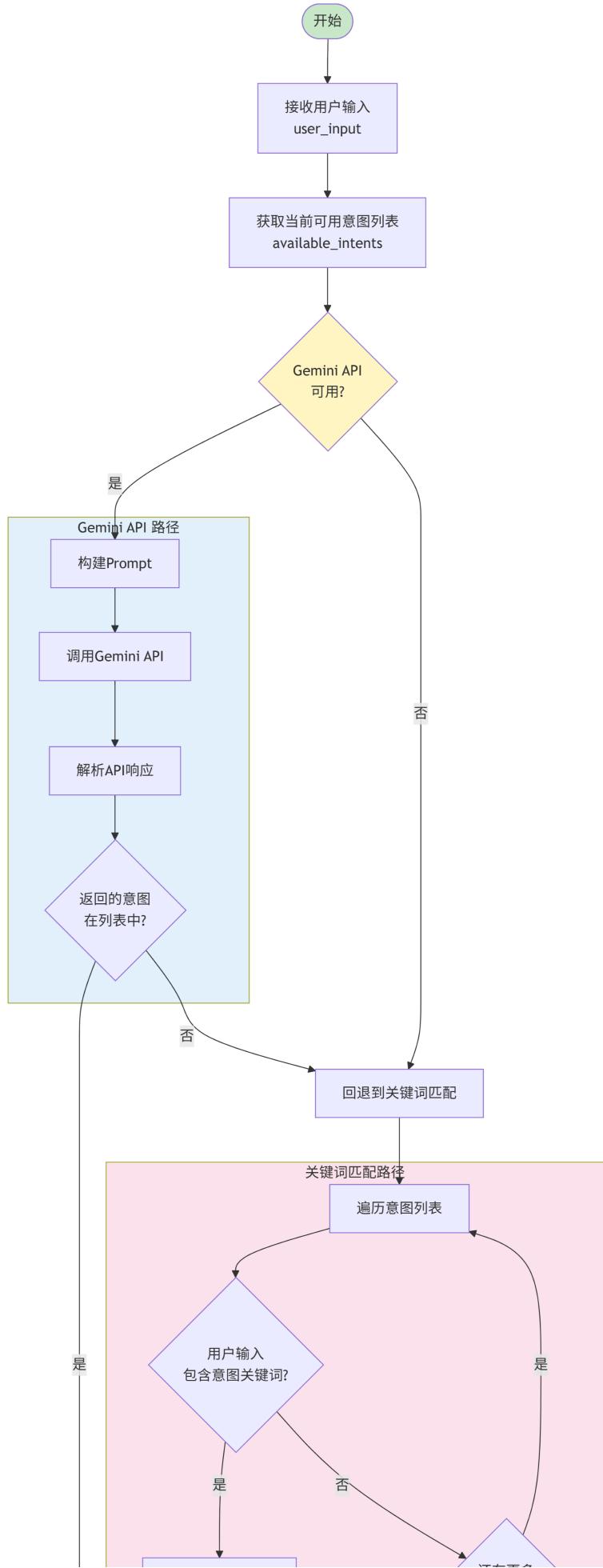
用户输入: {user\_input}  
可选意图: {', '.join(available\_intents)}

请从可选意图中选择最匹配的一个。如果都不匹配，返回空字符串。

只返回意图名称，不要其他内容。"""

```
response = self._call_gemini_api(prompt)  
intent = response.strip()  
  
if intent in available_intents:  
    return IntentResult(intent=intent, confidence=0.9)  
else:  
    return IntentResult(intent="", confidence=0.0)
```

MockIntentRecognizer用于测试和API不可用时的后备方案，它通过简单的关键词匹配来识别意图。



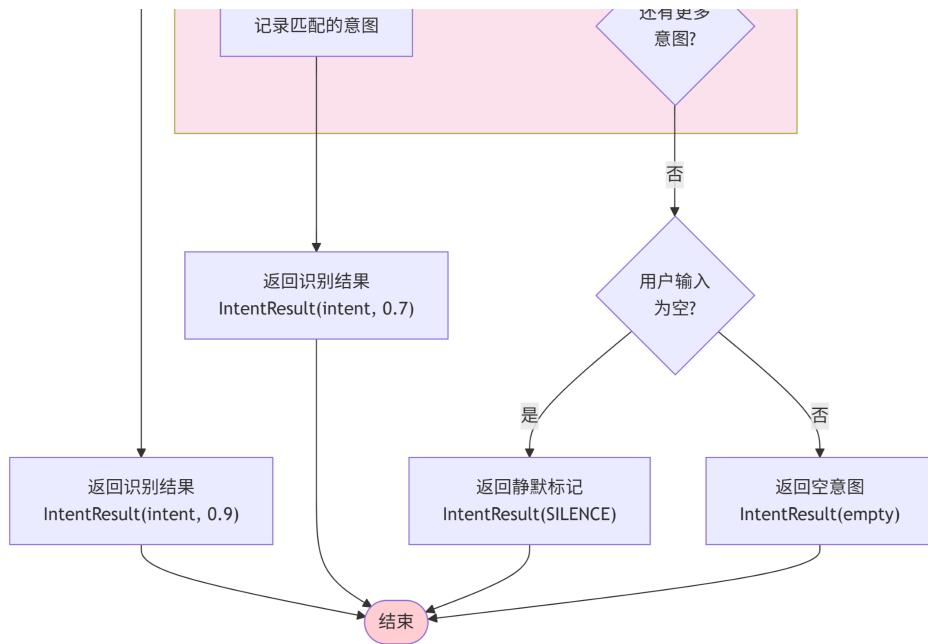


图6：意图识别流程图

## 4.5 Web服务子系统详细设计

Web服务层使用Flask框架实现，提供RESTful API和Web界面。

主要的API端点如下：

端点	方法	功能
/	GET	首页，展示场景选择
/chat/<scenario>	GET	聊天界面
/api/start	POST	创建会话，返回欢迎消息
/api/chat	POST	发送消息，返回机器人回复
/api/end	POST	结束会话

会话管理是Web服务层的重要职责。我使用字典来存储活跃的会话，键是会话ID（UUID），值是对应的解释器上下文。为了避免内存泄漏，会话在结束或超时后会被清理。

```

# 全局会话存储
sessions = {} # session_id -> ExecutionContext
interpreters = {} # scenario -> Interpreter

@app.route('/api/start', methods=['POST'])
def start_session():
    data = request.get_json()
    scenario = data.get('scenario')

    session_id = str(uuid.uuid4())
    interpreter = get_interpreter(scenario)
    context = interpreter.create_session(session_id)

    sessions[session_id] = context
    output = interpreter.start(session_id)

    return jsonify({
        'session_id': session_id,
        'message': output.message,
        'intents': output.available_intents
    })

```

前端使用原生JavaScript实现，通过AJAX与后端通信。聊天界面采用了响应式设计，在不同尺寸的屏幕上都能良好显示。

## 4.6 配置文件说明

本项目的配置主要通过环境变量和Python模块来管理。

API密钥配置存储在环境变量中：

```
export GEMINI_API_KEY="your-api-key-here"
```

脚本文件路径配置在代码中定义：

```

SCRIPTS_DIR = os.path.join(os.path.dirname(__file__), 'scripts')
SCRIPT_FILES = {
    'hospital': 'hospital.dsl',
    'restaurant': 'restaurant.dsl',
    'theater': 'theater.dsl'
}

```

## 4.7 日志说明

系统使用Python标准库的logging模块记录日志。日志分为以下几个级别：

- DEBUG: 详细的调试信息，如Token序列、AST结构等
- INFO: 常规运行信息，如会话创建、步骤跳转等
- WARNING: 警告信息，如API调用超时、使用后备方案等
- ERROR: 错误信息，如脚本解析失败、运行时异常等

日志格式示例：

```
2024-12-25 14:30:45 INFO [Interpreter] Session abc123 started, entry step: welcome
2024-12-25 14:30:46 INFO [IntentRecognizer] Input: "我要挂号" -> Intent: "挂号" (confidence: 0.9
2024-12-25 14:30:46 INFO [Interpreter] Jumping to step: registration
```

## 4.8 数据文件说明

### DSL脚本文件格式：

脚本文件采用纯文本格式，使用UTF-8编码。文件扩展名为 .dsl。脚本由多个Step组成，每个Step包含若干语句。注释以#开头，从#到行尾的内容都会被忽略。

### 测试数据文件格式：

测试数据存储在 `data/test_data.json` 文件中，JSON格式，包含各场景的测试用例定义。

```
{
  "test_cases": {
    "hospital": {
      "scenarios": [
        {
          "name": "正常挂号流程",
          "inputs": ["挂号", "内科", "张医生"],
          "expected_outputs_contain": ["科室", "医生", "成功"]
        }
      ]
    }
  }
}
```

## 4.9 接口协议说明

前后端之间通过HTTP协议通信，数据格式为JSON。

请求格式（以/api/chat为例）：

```
{  
    "scenario": "hospital",  
    "session_id": "550e8400-e29b-41d4-a716-446655440000",  
    "message": "我想挂号"  
}
```

响应格式：

```
{  
    "success": true,  
    "message": "好的，请问您想挂哪个科室？我们有内科、外科、儿科等。",  
    "state": "waiting_input",  
    "intents": ["内科", "外科", "儿科", "其他"],  
    "session_id": "550e8400-e29b-41d4-a716-446655440000"  
}
```

## 5 测试报告

### 5.1 测试驱动设计说明

本项目采用测试驱动的开发方式，在编写功能代码之前先设计测试用例。测试框架使用Python标准库的unittest模块，测试报告使用自定义的HTML报告生成器。

测试驱动的核心类是HTMLTestResult，它继承自unittest.TestResult，在测试执行过程中收集每个测试用例的结果，包括测试名称、描述、执行状态、耗时以及失败时的堆栈跟踪。

测试用例按照被测模块分组，每组对应一个TestCase类：

- TestLexer: 词法分析器测试
- TestParser: 语法分析器测试
- TestInterpreter: 解释器测试
- TestMockIntentRecognizer: 意图识别器测试

- TestIntegration: 集成测试
- TestExpressionEvaluation: 表达式计算测试

## 5.2 测试桩设计说明

为了隔离外部依赖，我设计了一系列测试桩（Test Stub）。

**LLMStub** 模拟大语言模型API的行为。可以预设特定输入对应的输出，也可以使用默认的关键词匹配逻辑。它还记录所有的调用历史，便于验证调用次数和参数。

```
class LLMStub:  
    def __init__(self):  
        self.responses = {}  
        self.call_history = []  
  
    def set_intent_response(self, input_text, intent, confidence=0.9):  
        self.responses[input_text.lower()] = {  
            "intent": intent,  
            "confidence": confidence  
    }  
  
    def recognize_intent(self, user_input, available_intents):  
        self.call_history.append({"input": user_input, "intents": available_intents})  
        # ... 返回预设响应或默认匹配结果
```

**UserInputStub** 模拟用户输入序列，用于自动化的集成测试。它预先设置一系列输入，每次调用 get\_next\_input() 返回下一个输入，模拟用户的连续操作。

**OutputCapture** 捕获解释器的输出，用于验证输出内容是否符合预期。

## 5.3 自动测试脚本设计说明

自动测试通过 tests/html\_report.py 脚本执行，它会自动发现 tests 目录下所有以 test\_ 开头的 Python 文件，运行其中的测试用例，并生成详细的 HTML 报告。

执行命令：

```
python tests/html_report.py
```

HTML 报告包含以下内容：测试执行的统计摘要（总数、通过、失败、错误、跳过）、可视化的饼图和柱状图、按测试类分组的详细结果列表、失败测试的完整堆栈跟踪。

## 5.4 测试过程

测试过程分为三个阶段：

**单元测试阶段**首先针对词法分析器进行测试，验证各种Token的识别是否正确，包括关键字、标识符、字符串字面量、数字字面量、变量、运算符等。然后测试语法分析器，验证各种语句和表达式的解析是否正确，AST的结构是否符合预期。

**集成测试阶段**将词法分析器、语法分析器和解释器串联起来，使用完整的DSL脚本进行测试。首先使用简单的测试脚本验证基本流程，然后加载三个业务场景的脚本进行全流程测试。

**系统测试阶段**启动Web服务，通过浏览器进行手工测试，验证前后端交互是否正常，用户体验是否流畅。

## 5.5 测试结果



图7：HTML测试统计摘要

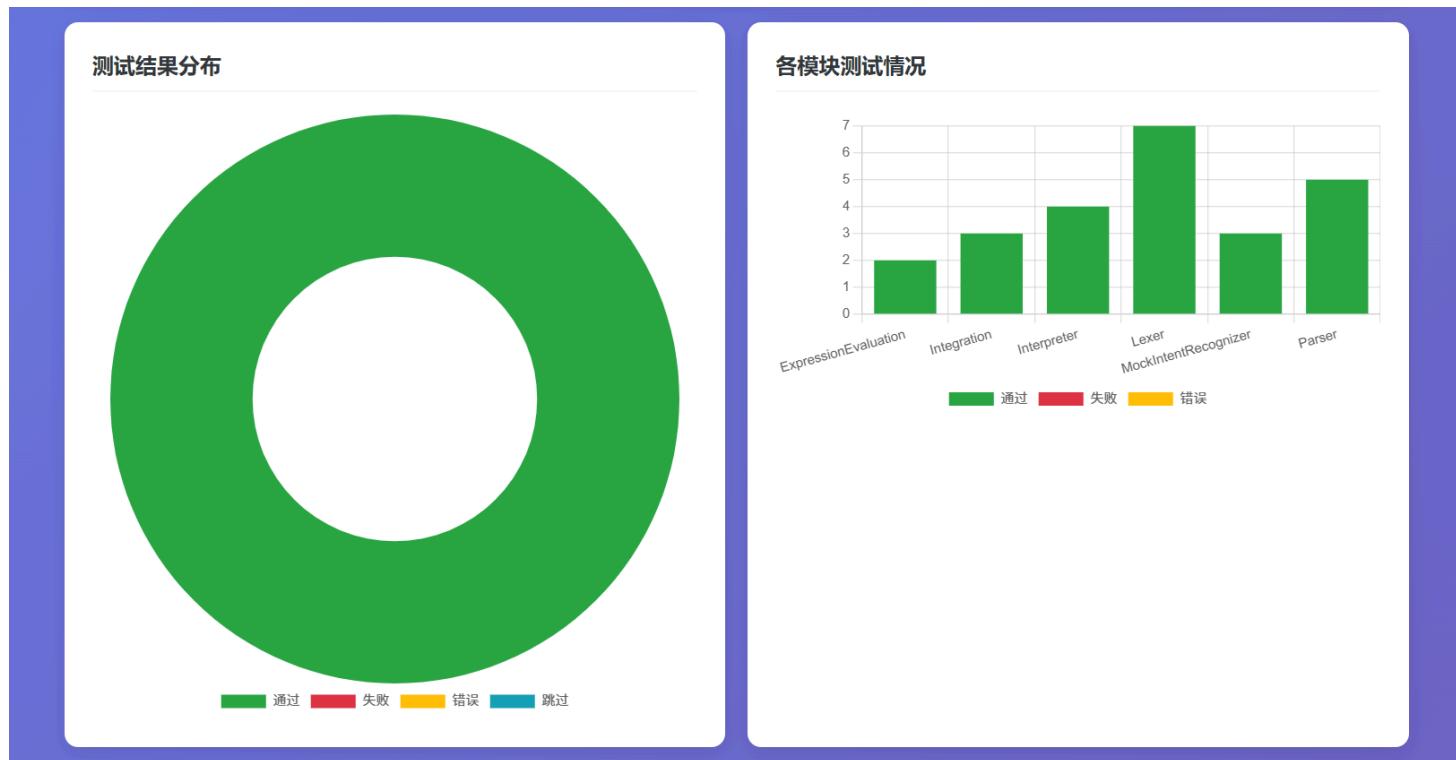


图8：结果分布与各模块情况

测试执行结果汇总如下：

测试类别	测试用例数	通过	失败	错误
TestLexer	7	7	0	0
TestParser	5	5	0	0
TestInterpreter	4	4	0	0
TestMockIntentRecognizer	3	3	0	0
TestIntegration	3	3	0	0
TestExpressionEvaluation	2	2	0	0
<b>总计</b>	<b>24</b>	<b>24</b>	<b>0</b>	<b>0</b>

所有测试用例均通过，测试覆盖了核心功能的各个方面。

# 6 DSL脚本编写指南

## 6.1 脚本语法说明

本DSL的语法设计借鉴了常见脚本语言的风格，力求简洁直观，使非程序员也能理解和编写。

### 基本结构

一个脚本文件由多个Step组成，第一个Step是入口步骤。每个Step包含步骤名和一系列语句。

**Step** 步骤名

语句1

语句2

...

### 关键字一览

关键字	语法	说明
Step	Step <name>	定义一个步骤
Speak	Speak <expression>	输出文本
Listen	Listen <begin>, <end>	等待用户输入，参数为超时时间
Branch	Branch "<intent>", <target>	意图分支，匹配则跳转
Silence	Silence <target>	静默处理，用户无输入时跳转
Default	Default <target>	默认处理，无匹配时跳转
Exit	Exit	结束对话
Set	Set \$var = <expression>	变量赋值
Goto	Goto <target>	无条件跳转
If/Else/EndIf	If <condition> ... EndIf	条件分支

### 表达式

表达式可以是字符串字面量、数字字面量、变量或它们的组合。字符串用双引号包围，变量以\$开头，用+运算符连接多个值。

```
Speak "您好， " + $name + "！"  
Set $total = $price * $count
```

## 注释

以#开头的内容是注释，从#到行尾都会被忽略。

```
# 这是一行注释  
Speak "Hello" # 这也是注释
```

## 6.2 脚本用法说明

### 编写脚本的基本步骤：

首先，明确业务流程，画出流程图，标出各个节点和转换条件。然后，为每个节点创建一个Step，Step名称要有意义，便于维护。接着，在每个Step中添加Speak语句输出提示信息，添加Listen语句等待用户输入，添加Branch语句处理各种意图。最后，添加Silence和Default处理异常情况，在流程结束处添加Exit语句。

### 命名规范：

Step名称使用驼峰命名法，如 `welcomeStep`、`confirmPayment`。变量名以开头，使用有意义的名称，如`'userName`、`\$totalAmount`。意图字符串使用简洁的中文词语，如"挂号"、"查询"、"确认"。

### 调试技巧：

可以在关键位置添加Speak语句输出变量值，帮助定位问题。例如：

```
Set $debug = "当前步骤: confirmOrder, 金额: " + $amount  
Speak $debug
```

## 6.3 脚本范例

以下是医院挂号场景的简化示例，展示了DSL的主要特性：

```
# 医院智能导诊机器人脚本
```

```
# 作者: [张泽坤]
```

```
# 日期: 2025年12月
```

### Step welcome

```
Speak "您好, 欢迎致电智慧医院服务热线。请问有什么可以帮您? "
```

```
Listen 5, 30
```

```
Branch "挂号", registration
```

```
Branch "缴费", payment
```

```
Branch "取药", medicine
```

```
Branch "查询", query
```

```
Silence silenceHandler
```

```
Default defaultHandler
```

### Step registration

```
Speak "好的, 为您办理挂号。请问您想挂哪个科室? "
```

```
Listen 5, 30
```

```
Branch "内科", regInternal
```

```
Branch "外科", regSurgery
```

```
Branch "儿科", regPediatrics
```

```
Default askDepartment
```

### Step regInternal

```
Set $department = "内科"
```

```
Speak "您选择了内科。请问选择哪位医生? 我们有张医生(主任医师)和李医生(副主任医师)。"
```

```
Listen 5, 30
```

```
Branch "张医生", confirmZhang
```

```
Branch "李医生", confirmLi
```

```
Default askDoctor
```

### Step confirmZhang

```
Set $doctor = "张医生"
```

```
Set $fee = 50
```

```
Speak "您选择了" + $department + "的" + $doctor + ", 挂号费" + $fee + "元。确认挂号吗? "
```

```
Listen 5, 20
```

```
Branch "确认", createRegistration
```

```
Branch "取消", cancelled
```

```
Default confirmAgain
```

### Step createRegistration

```
Set $regNo = "H20241225001"
```

```
Speak "挂号成功! 您的挂号单号是" + $regNo + ", 请于就诊当日携带身份证件到" + $department + "候诊。
```

```
Exit
```

```
Step silenceHandler
  Speak "您好，请问还在吗？如需帮助请说出您的需求。"
  Listen 5, 30
  Branch "挂号", registration
  Branch "缴费", payment
  Default goodbye

Step defaultHandler
  Speak "抱歉，没有理解您的意思。您可以说"挂号"、"缴费"或"取药"。"
  Listen 5, 30
  Branch "挂号", registration
  Branch "缴费", payment
  Branch "取药", medicine
  Default transferHuman

Step goodbye
  Speak "感谢您的来电，再见！"
  Exit
```

这个示例展示了完整的对话流程，包括欢迎语、意图识别、多级菜单、变量使用、异常处理等特性。

## 7 AI辅助编程过程说明

### 7.1 本作业使用的AI辅助编程工具说明

在本项目的开发过程中，我使用了Claude作为辅助编程工具。Claude是Anthropic公司开发的大语言模型，具备代码生成、问题解答、方案设计等能力。我主要将其用于以下几个方面：技术方案的讨论与验证、复杂算法的思路启发、代码片段的参考实现、文档撰写的润色。

需要说明的是，我始终将AI作为辅助工具而非替代品。每次与AI交互后，我都会仔细审查其输出，理解其原理，并根据项目实际情况进行修改和优化。项目的整体架构设计、核心算法实现、业务逻辑编排都是我独立完成的。

### 7.2 使用AI辅助编程的过程

#### 阶段一：技术选型讨论

在项目初期，我需要确定DSL解释器的实现方案。我向AI提出了这样的问题：

我："我需要实现一个简单的DSL解释器，目标语言类似于客服对话脚本。请问实现解释器有哪些主要的技术路线？各自的优缺点是什么？"

AI详细介绍了三种方案：基于正则表达式的简单解析、手写递归下降解析器、使用解析器生成工具（如ANTLR）。经过分析，我认为手写递归下降解析器最适合本项目——它足够灵活，可以精确控制错误处理，而且不需要引入额外的依赖。

## 阶段二：词法分析器设计

在设计词法分析器时，我遇到了一个具体问题：如何高效地区分关键字和普通标识符？我的初始实现是对每个词都遍历一次关键字列表，效率较低。我向AI请教：

我："词法分析时，识别出一个单词后需要判断它是关键字还是标识符。我目前用列表遍历，有没有更好的方法？"

AI建议使用字典（哈希表）存储关键字，查找时间复杂度从 $O(n)$ 降到 $O(1)$ 。这个建议很实用，我立即采纳并修改了代码：

```
KEYWORDS = {
    'Step': TokenType.STEP,
    'Speak': TokenType.SPEAK,
    'Listen': TokenType.LISTEN,
    # ... 其他关键字
}

def get_identifier_type(self, word):
    return KEYWORDS.get(word, TokenType.IDENTIFIER)
```

## 阶段三：AST节点设计

在设计AST节点类时，我思考了两种方案：一是使用简单的字典嵌套，二是定义专门的类。我向AI描述了我的犹豫：

我："AST节点用字典还是类来表示？字典简单直接，类需要定义很多代码，但类型更清晰。你怎么看？"

AI分析了两种方案的优劣，指出类的方式虽然代码量多一些，但可以提供类型提示、方法封装和更好的IDE支持，长远来看维护成本更低。这促使我选择了类的方案，并学习使用了Python的数据类装饰器来简化类的定义。

## 阶段四：意图识别集成

集成Gemini API时，我不太确定如何构造合适的Prompt。我向AI展示了我的初版Prompt，并询问如何改进：

我："我的Prompt是直接把用户输入和候选意图拼接在一起，让模型返回匹配的意图。但效果不太稳定，有时返回的格式不对。如何改进？"

AI建议在Prompt中更明确地说明输出格式要求，并提供Few-shot示例。我采纳了这个建议，修改后的Prompt如下：

分析用户输入，判断其意图。

用户输入: {user\_input}

候选意图: {intents}

要求:

1. 从候选意图中选择最匹配的一个
2. 如果都不匹配，返回"无"
3. 只返回意图名称，不要其他内容

示例:

用户输入: "我想看看今天有什么演出", 候选意图: ["查询", "购票", "取票"]

返回: 查询

修改后，识别的准确率和稳定性都有明显提升。

## 阶段五：测试框架搭建

在编写测试代码时，我希望生成美观的HTML测试报告。我向AI询问了unittest模块的扩展方法：

我："unittest默认的文本输出太简陋了，我想生成HTML格式的报告，包含统计图表。应该从哪里入手？"

AI介绍了继承TestResult类来收集测试结果的方法，并提到可以用Chart.js库在HTML中绘制图表。基于这些提示，我设计并实现了HTMLTestResult和HTMLReportGenerator两个类，最终生成了既美观又信息丰富的测试报告。

## 7.3 使用AI辅助编程的经验教训总结

**有效利用AI的方法：**

提问要具体明确。与其问"怎么写解释器"，不如问"递归下降解析器处理左递归的常用方法有哪些"。模糊的问题只能得到泛泛的回答，具体的问题才能获得有针对性的帮助。

带着思考去提问。在向AI求助之前，我会先自己尝试解决问题，形成初步方案。然后带着具体的方案去和AI讨论，让AI评估优缺点或提供改进建议。这样的交互比单纯索要答案更有价值。

验证和理解AI的输出。AI给出的代码或方案不一定完美，有时甚至有错误。我会仔细阅读、理解其原理，在本地测试验证后才会采纳。对于复杂的算法，我还会查阅相关资料确认其正确性。

### **AI辅助编程的局限：**

AI不了解项目的全貌。它只能基于当前对话的上下文给出建议，无法像人类同事那样了解项目的历史背景和整体架构。因此，架构级别的决策还是需要开发者自己做出。

AI可能给出过度复杂的方案。有时候我只需要一个简单的解决方案，但AI会给出一个功能完备但复杂度较高的实现。这时候需要开发者判断什么是"够用"的，避免过度工程化。

不能依赖AI来学习新知识。AI可以提供知识点的概述，但要真正掌握一项技术，还是需要阅读官方文档、教程和源码。我在本项目中就补充学习了《编译原理》教材中关于词法分析和语法分析的章节。

### **总体评价：**

AI辅助编程显著提升了我的开发效率，特别是在技术调研、方案设计和问题排查方面。但它更像是一个知识渊博的顾问，而非代替我写代码的工具。项目的成功最终取决于我对问题的理解、对技术的掌握以及对细节的把控。

## **8 GIT日志**

以下是本项目开发过程中的GIT提交日志（部分）：

```
commit 669d856c6267815b360780035633ebfcb37eb2d1
Author: Zhang Zeshen <zeshenzhang240@gmail.com>
Date:   Fri Dec 26 00:37:38 2025 +0800
```

Adjust and insert figures

```
commit 480b36ab64d919b9a5bd7ae7103fc8363274f318
Author: Zhang Zeshen <zeshenzhang240@gmail.com>
Date:   Fri Dec 26 00:10:44 2025 +0800
```

Add Figure

```
commit 980055c0e9d7d9d4d5e3c2b04978ec9f7bc4e8e2
Author: Zhang Zeshen <zeshenzhang240@gmail.com>
Date:   Thu Dec 25 23:21:56 2025 +0800
```

Update README

```
commit 110400188795417e64238708a1e145f304a82158
Author: Zhang Zeshen <zeshenzhang240@gmail.com>
Date:   Thu Dec 25 23:00:22 2025 +0800
```

Add Report

```
commit 1acae1df280d36c54a5d1f0a15c2f849f52bd184
Author: Zhang Zeshen <zeshenzhang240@gmail.com>
Date:   Thu Dec 25 22:29:24 2025 +0800
```

Update index.html

```
commit 0499a343966db5443ab08477d7701c84c3099f68
Author: Zhang Zeshen <zeshenzhang240@gmail.com>
Date:   Thu Dec 25 22:12:33 2025 +0800
```

Add HTML test

```
commit 27f4317a415b1aed167470651892348b3f7fbec4
Author: Zhang Zeshen <zeshenzhang240@gmail.com>
Date:   Thu Dec 25 21:48:44 2025 +0800
```

Update README

```
commit 06708b8ed550c9ea0747ba241c0f7c5f6e6a956d
```

Author: Zhang Zeshen <zeshenzhang240@gmail.com>

Date: Thu Dec 25 21:39:10 2025 +0800

first commit

## 9 总结与展望

### 收获与体会

通过这个项目，我对编译原理的知识有了更深的理解。以前在课堂上学习词法分析、语法分析时，总觉得这些概念离实际应用很远。但当我真正动手实现一个解释器时，才发现这些理论知识是如此实用。从Token的设计到AST的构建，从递归下降算法到表达式求值，每一步都需要把理论转化为代码。这个过程让我对“程序即数据”的理念有了切身体会。

项目开发也锻炼了我的系统设计能力。面对一个复杂的需求，如何拆分模块、定义接口、管理依赖，这些都是实际工程中的关键问题。我学会了先画架构图再写代码，先定义接口再实现功能，这种自顶向下的方法让整个开发过程更加有序。

测试的重要性是另一个深刻的体会。在开发早期，我曾经急于添加新功能而忽略了测试。结果后来修复bug时才发现问题早已存在，只是一直没被发现。从那以后，我坚持编写单元测试，最终24个测试用例全部通过，这给了我很大的信心。

### 尚未实现的设想

由于时间限制，以下功能没有来得及实现：

**脚本热更新功能：** 目前修改脚本后需要重启服务才能生效。理想的方案是支持不停机更新，检测到脚本文件变化后自动重新加载。

**对话历史记录：** 当前的实现没有持久化对话历史，服务重启后所有会话丢失。可以考虑使用Redis或数据库来存储会话状态。

**更丰富的表达式支持：** 目前只支持简单的字符串拼接和算术运算，可以扩展支持更多的运算符和内置函数。

**可视化脚本编辑器：** 对于非技术人员，可以提供一个图形化的脚本编辑界面，用拖拽的方式设计对话流程，自动生成DSL代码。

# 对课程的建议

这门课程的大作业设计很有挑战性，让我们有机会实践完整的软件开发流程。如果有一些改进空间的话，我建议可以提供更多的参考资料和示例代码，帮助同学们更快地入门。另外，可以考虑组织一些中期检查，及时发现和解决问题，避免最后阶段过于仓促。

总的来说，这个项目是一次宝贵的学习经历。它让我把课堂上学到的知识应用到了实际问题中，也让我对软件开发有了更全面的认识。感谢老师的指导和课程的精心设计。

## 附录

- 附录A：完整源代码（见提交的代码仓库）
- 附录B：HTML测试报告（见tests/test\_report.html）
- 附录C：DSL脚本文件（见scripts目录）