

# DSL 软件开发手册

DSL-程序设计实践

姓名：陈朴炎      学号：2021211138

目录

- 1 设计和实现..... 3
  - 1.1 功能需求说明和分析 ..... 3
    - 1.1.1 需求说明..... 3
    - 1.1.2 功能需求..... 3
    - 1.1.4 功能说明..... 4
  - 1.2 模块划分 ..... 6
    - 1.2.1 模块划分总览..... 6
    - 1.2.2 语法树模块 ..... 7
    - 1.2.3 解释器模块 ..... 7
    - 1.2.4 服务器模块 ..... 8
    - 1.2.5 登陆界面模块..... 8
    - 1.2.6 会话界面模块..... 9
    - 1.2.7 测试模块..... 9
  - 1.3 数据结构设计 ..... 10
    - 1.3.1 语法树 ScriptTree ..... 10
    - 1.3.2 解释器 ScriptParser ..... 12
    - 1.3.3 异常处理 ScriptError..... 14
    - 1.3.4 服务器 Server..... 15
    - 1.3.5 存放订单的数据结构 Order ..... 16
    - 1.3.6 存放环境变量信息 Vars..... 17

1.3.7 自动测试类 AutoTest .....	19
2 接口说明.....	21
2.1 人机接口 .....	21
2.1.1 登录界面人机接口 .....	21
2.1.2 对话界面人机接口 .....	22
2.2 前后端通信接口 .....	23
2.2.1 消息协议.....	23
2.2.2 消息工作过程.....	24
2.3 程序间接口—类与 API .....	25
2.3.1 ScriptTree 类的接口.....	25
2.3.2 ScriptParser 类的接口 .....	28
2.3.3 Order 类的接口 .....	29
2.3.4 Vars 类的接口 .....	30

# 1 设计和实现

## 1.1 功能需求说明和分析

### 1.1.1 需求说明

领域特定语言（Domain Specific Language, DSL）可以提供一种相对简单的文法，用于特定领域的业务流程定制。本作业要求定义一个领域特定脚本语言，这个语言能够描述在线客服机器人（机器人客服是目前提升客服效率的重要技术，在银行、通信和商务等领域的复杂信息系统中有广泛的应用）的自动应答逻辑，并设计实现一个解释器解释执行这个脚本，可以根据用户的不同输入，根据脚本的逻辑设计给出相应的应答。

### 1.1.2 功能需求

(1) 脚本语言的设计与定义：定义领域特定语言的语法和关键词，用来描述在线客服机器人的自动应答逻辑。支持对话流程、条件判断、变量定义和操作等基本语法元素。

(2) 变量和状态的管理：支持定义和操作变量，以便在对话过程中保存和更新信息。能够管理机器人的内部状态，以便根据历史对话做出更智能的应答。支持条件判断，根据用户的输入和机器人内部状态决定不同的应答路径。

(3) 基本应答逻辑：定义默认的应答逻辑，以应对未在脚本中明确处理的情况。支持预定义的常用应答，例如问候、感谢、再见等。

(4) 用户交互需求：能够接收用户输入，包括文本、按钮等；能够根据脚本定

义生成机器人的回应，包括文本和可能的其他输出形式。支持动态生成应答，考虑到变量和条件的影响；能够维护和管理对话状态，确保机器人能够理解上下文并做出合适的应答。支持多轮对话，能够处理用户的连续输入和多步操作。

(5) 多线程及安全需求：确保同时进行的不同会话不会互相影响，并且能够合理处理敏感信息，确保数据不冲突。

#### 1.1.4 功能说明

基于以上功能需求分析，可以总结出该项目要完成以下功能：

##### (1) 脚本语言定义

定义一款专为客服机器人设计的简洁明了的脚本语言。用户可以通过该语言轻松定义客服机器人的对话逻辑，包括对话流程、条件判断、变量操作、基本应答逻辑以及状态转移。这个脚本语言的设计旨在让用户能够简便地完成一个逻辑完善的客服机器人脚本，使其能够智能地响应用户输入、处理不同情境下的对话流程，并提供个性化和有效的回应，帮助用户在不需要繁杂编程知识的情况下实现智能的在线客服。

##### (2) 脚本解释器

脚本解释器的设计要求包括对已实现脚本文件进行逐行解析，通过判断每行关键词的排列顺序来检测语法和语义错误，提供详细的错误信息以协助用户修复问题。此外，解释器需要具备构建脚本语法树的能力，以反映脚本的结构和逻辑，并支持对语法树的遍历、添加、更新等。对于每个状态，解释器应当能够识别不同的触发条件，并执行相应的操作，包括状态转移和回应生成。解释器必须处理

不同触发条件下的多种结果，包括自定义操作，如调用外部函数或模块。具备容错机制以处理脚本中可能存在的错误，并支持动态条件判断，使得在执行过程中能够灵活调整触发条件。最终，解释器需要具备扩展性，允许用户自定义关键词、操作和触发条件，以适应不同领域和应用场景的需求。这样设计的解释器将为在线客服机器人提供高效、灵活、可扩展的自动应答逻辑执行环境。

### (3) 服务器

服务器的功能注重于环境变量的灵活操作，通过有效的解释机制将脚本中的语言占位符映射到相应的环境变量信息。为用户提供了高度的可定制性，使得脚本能够根据不同的环境变量状态生成个性化的响应。同时，服务器的架构充分支持多用户信息的处理，确保每个用户的状态独立管理，从而避免不同用户之间的信息冲突。每个用户在与服务器交互时，服务器能够根据其特定状态解释相关的脚本，提供个性化的服务。

服务器具备强大的消息处理机制，能够同时处理多个用户的信息流，确保不同用户之间的通信互不干扰。同时，服务器和前端之间建立了一套完善的通信协议体系，以确保数据的传输和解释的准确性。这种通信协议体系不仅有助于数据的有效传递，还为后续的功能扩展提供了稳定的基础。

该服务器要满足复杂多用户场景下的需求，通过一个脚本解释器，一颗语法树、多个环境变量的操作和通信协议体系的完备性，为客户提供高度个性化、实时响应的服务。这种设计理念使服务器成为一个强大而灵活的信息处理引擎，能够应对不同行业和应用场景的需求。

### (4) 客户端设计

客户端设计注重简洁与清晰，提供用户友好的交互体验。用户可以通过简单的登录过程，输入用户名和密码，并通过回车键轻松发送消息和请求。在登录失败时，系统将给予提示，而成功登录后将自动跳转至客服界面。

在客服界面，用户可以清晰地区分客户和客服的信息，确保对话的可读性。提供了聊天输入栏和发送按钮，使用户能够方便地输入和发送消息。同时，提供了退出按钮，以方便用户随时退出应用。界面大小可调整，以适应不同用户的个人偏好和设备屏幕大小。

客户端功能简单易用，注重用户体验，通过清晰的界面和简便的操作，使用户能够轻松地登录、发送消息和请求，并实现方便的退出功能。

## **1.2 模块划分**

### **1.2.1 模块划分总览**

基于功能需求说明分析，进行以下模块划分：

- (1) 语法树模块
- (2) 解释器模块
- (3) 服务器模块
- (4) 登录界面模块
- (5) 会话界面模块
- (6) 测试模块

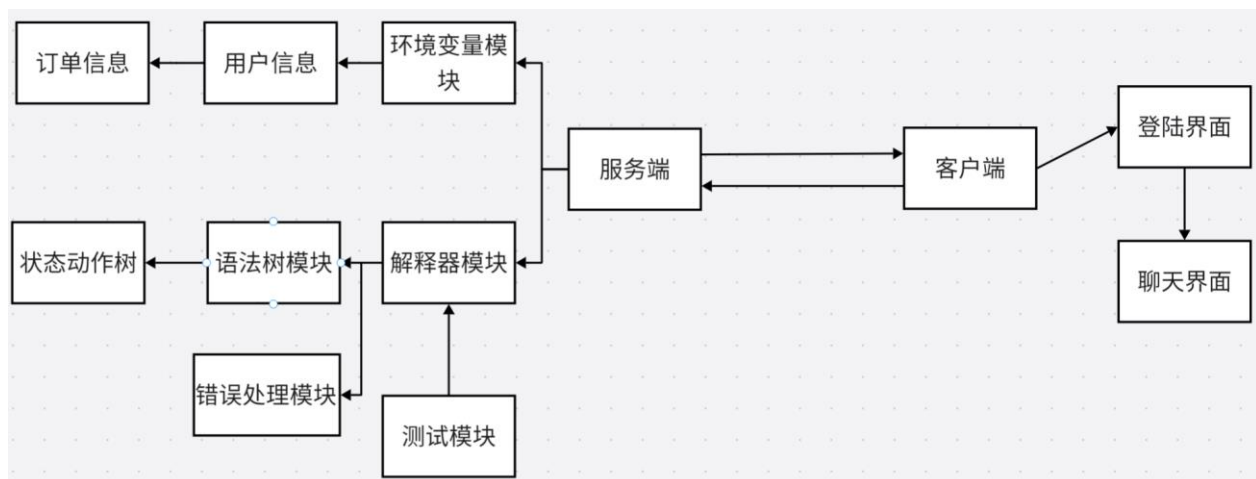


图 1-1 模块划分图

下面将介绍这些模块

### 1.2.2 语法树模块

**职责：**语法树模块是解释器模块的一部分，其主要职责是存储解释器解析出的脚本文件的语法树。语法树包括状态及动作树、各个状态的初始回应、语法入口、语法出口、各个状态的等待时间等关键信息。

**依赖关系：**依赖解释器模块，通过解释器模块解析脚本文件并构建语法树。

**作用：**这个模块通过封装了脚本解释过程中的关键信息，提供了易于访问和使用的接口，有助于解释器在运行时根据用户输入进行状态转移和回应生成。

### 1.2.3 解释器模块

**职责：**解释器模块负责解析脚本文件，更新语法树，并处理脚本文件中可能出现的语法错误。包含了语法树模块和错误处理模块，提供了从语法树获取回应的接口。

**依赖关系：**依赖语法树模块，用于存储解释器解析出的脚本文件的语法树。依赖



错误处理模块，在解释脚本文件时，会自动检查、更新语法树里的内容，当发现脚本文件有语法错误时，会引发 `ScriptError` 异常。被服务器模块包含，服务器在接收消息时需要使用解释器来获取相应的回应

作用：这个模块通过提供接口，实现了从脚本文件到语法树的解析过程。在解释脚本文件的过程中，会自动检查语法错误，并将解析出的信息更新到语法树中，以便后续使用。

#### 1.2.4 服务器模块

职责：服务器模块负责管理多个用户的会话，处理用户的输入和输出，以及调用解释器模块进行脚本解释。该模块包含了解释器模块，用于实现客户端与服务器之间的通信和服务逻辑。

依赖关系：依赖解释器模块，用于实现脚本解释和生成回应的功能。依赖环境变量管理模块，用于处理解释器解释出的语言占位符。依赖用户管理模块，确保每个用户的会话状态得到独立管理。

作用：这个模块通过整合解释器模块和其他关键功能模块，实现了一个完整的服务器功能，能够处理多用户的请求，确保各用户的状态独立，提供安全、稳定的服务。

#### 1.2.5 登陆界面模块

职责：登陆界面模块负责用户的登录交互，获取用户输入的用户名和密码，并通过 `Socket` 与服务器通信进行登录验证。在登录成功后打开聊天窗口。

依赖关系：依赖 Socket 通信模块 (SocketManager)：用于与服务器进行通信。

依赖聊天窗口模块 (ChatWindow)：登录成功后打开聊天窗口。

作用：这个模块通过提供用户友好的界面和与服务器的通信，实现了用户的登录交互功能。在登录成功后，用户可以进入聊天窗口进行更多的操作。

### 1.2.6 会话界面模块

职责：聊天窗口模块负责展示用户与机器人的聊天记录，并提供输入框和发送按钮，实现用户与机器人之间的实时交互。

依赖关系：依赖 Socket 通信模块，用于与服务器进行通信。

作用：这个模块通过提供一个友好的聊天界面，实现了用户与机器人之间的实时交互。用户可以通过输入框输入消息，通过点击按钮或按下回车键发送消息，并在聊天记录框中看到机器人的回应。

### 1.2.7 测试模块

职责：用于自动化测试脚本解释器，检验脚本的输出是否符合预期。

依赖关系：依赖脚本解释器模块 (ScriptParser)：用于模拟运行环境，获取脚本解释器的输出

作用：这个测试模块用于验证脚本解释器在给定输入条件下是否生成了正确的输出。通过加载输入文件和验证文件，执行脚本解释器，比对生成的输出与预期输出，输出测试结果。

## 1.3 数据结构设计

基于需求分析和模块划分，我们便能设计出项目所需的数据结构，以下是项目主要的数据结构。

### 1.3.1 语法树 ScriptTree

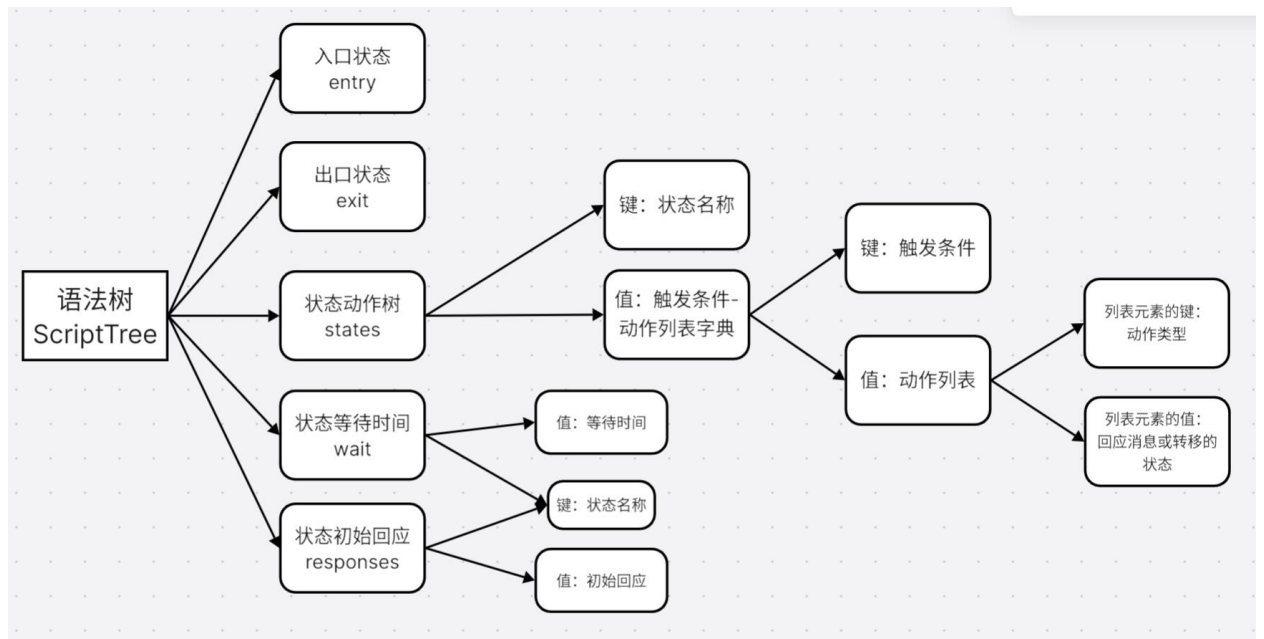


图 1-2 语法树数据结构示意图

将脚本元素抽象成树形结构：

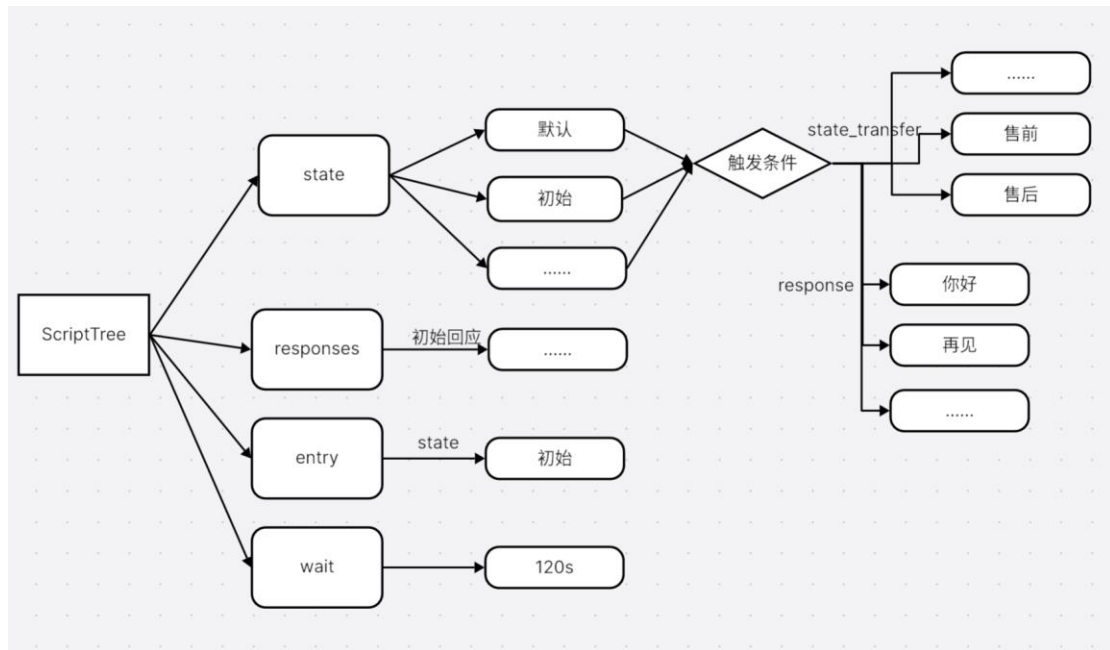


图 1-3 脚本元素转为树形结构

将以上数据结构示意图转成伪代码如下：

```

1. class ScriptTree:
2.     states: Dictionary of Dictionaries # 状态和动作树<状态名:[触发条件-动作列表]>
3.     responses: Dictionary # 初始化回应
4.     entry: String # 脚本入口状态
5.     exit: String
6.     wait: Dictionary # 等待时间表: <状态名: 等待时间>
  
```

属性说明：

states: 存储状态及其对应的动作树。每个状态关联一个触发条件-动作表。

responses: 字典，存储各个状态的初始回应。键为状态名，值为初始回应

entry: 脚本的入口状态。

exit: 脚本的退出状态。

wait: 字典，存储各个状态的等待时间

### 1.3.2 解释器 ScriptParser

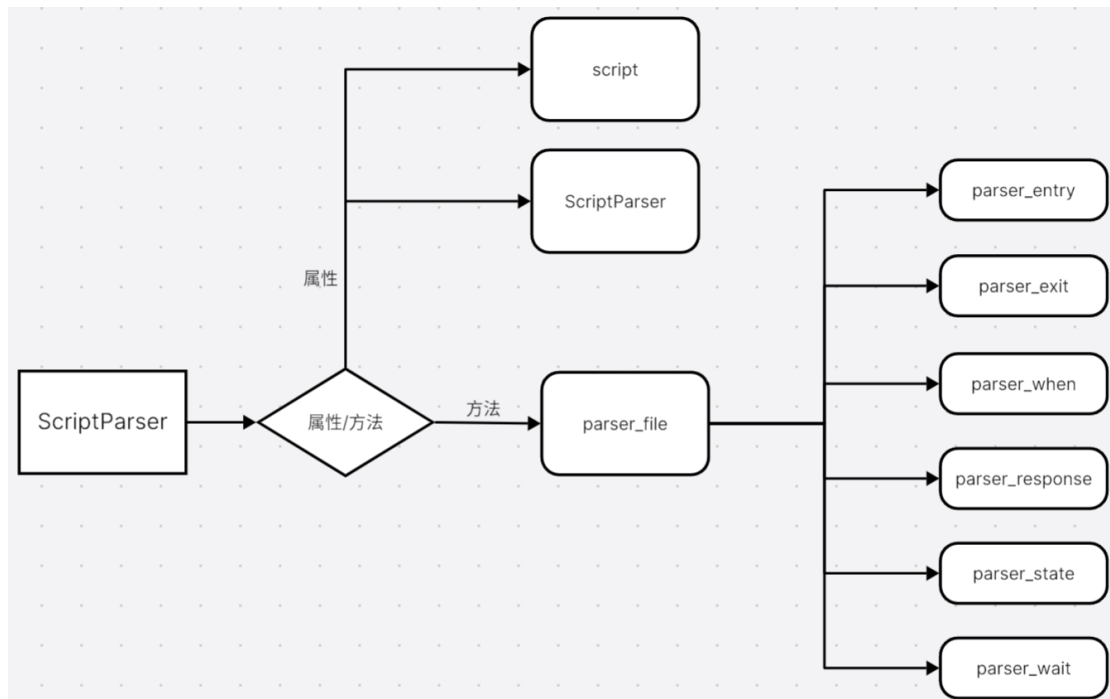


图 1-4 解释器数据结构示意图

将上述示意图换成数据结构伪代码如下：

```
1. Class: ScriptParser
2.     - script: 存储脚本内容的字符串
3.     - script_tree: ScriptTree 实例，用于存储和处理脚本解析后的语法树
4.
5.     Method: __init__(script_path: String)
6.         - 初始化方法，从给定路径加载脚本内容，创建 ScriptTree 实例
7.
8.     Method: get_entry() -> String
9.         - 获取脚本语法树的入口状态
10.
11.    Method: get_response(user_input: String, state_name: String) -> Tuple of
        List of String, String
12.        - 根据用户输入和当前状态，获取脚本的回应消息列表和新的状态
13.
14.    Method: parser_file()
15.        - 解析整个脚本文件，构建语法树
16.
17.    Method: parser_entry(line: String, row: Integer)
18.        - 解析脚本中的 entry 语句，设置语法树的入口状态
```

```

19.
20.     Method: parser_exit(line: String, row: Integer)
21.         - 解析脚本中的 exit 语句, 设置语法树的退出状态
22.
23.     Method: parser_wait(line: String, row: Integer)
24.         - 解析脚本中的 wait 语句, 设置各个状态的等待时间
25.
26.     Method: parser_when(line: String, row: Integer) -> Tuple of String, Dicti
        onary
27.         - 解析脚本中的 when 语句, 返回触发条件和动作字典的元组
28.
29.     Method: parser_response(current_state: String, line: String, row: Integer
        )
30.         - 解析脚本中的 response 语句, 设置状态的初始回应
31.
32.     Method: parser_state(current_state: String, cases: Dictionary, line: Stri
        ng, row: Integer) -> String
33.         - 解析脚本中的 state 语句, 将上一个状态的状态树加入语法树中, 并初始化新的状
        态树

```

## 属性说明:

script: String (存储脚本内容的字符串)

script\_tree: ScriptTree (存储和处理脚本解析后的语法树)

## 方法说明:

Method: \_\_init\_\_(script\_path: String)初始化方法, 从给定路径加载

脚本内容, 创建 ScriptTree 实例

Method: get\_entry() -> String 获取脚本语法树的入口状态

Method: get\_response(user\_input: String, state\_name: String)  
-> Tuple of List of String, String 根据用户输入和当前状态, 获取脚  
本的回应消息列表和新的状态

Method: parser\_file()解析整个脚本文件, 构建语法树

Method: parser\_entry(line: String, row: Integer)解析脚本中的

entry 语句，设置语法树的入口状态

Method: `parser_exit(line: String, row: Integer)`解析脚本中的  
exit 语句，设置语法树的退出状态

Method: `parser_wait(line: String, row: Integer)`解析脚本中的  
wait 语句，设置各个状态的等待时间

Method: `parser_when(line: String, row: Integer) -> Tuple of  
String, Dictionary` 解析脚本中的 when 语句，返回触发条件和动作字典的  
元组

Method: `parser_response(current_state: String, line: String,  
row: Integer)`解析脚本中的 response 语句，设置状态的初始回应

Method: `parser_state(current_state: String, cases:  
Dictionary, line: String, row: Integer) -> String` 解析脚本中的  
state 语句，将上一个状态的状态树加入语法树中，并初始化新的状态树

### 1.3.3 异常处理 ScriptError

```
1. Class: ScriptError
2.     - message: String (错误信息)
3.
4.     Method: __init__(message: String = "脚本文件有错")
5.         初始化方法，设置错误信息为给定的消息
```

当脚本解释器解释脚本时，会创建一个异常处理类，发出错误信息

### 1.3.4 服务器 Server

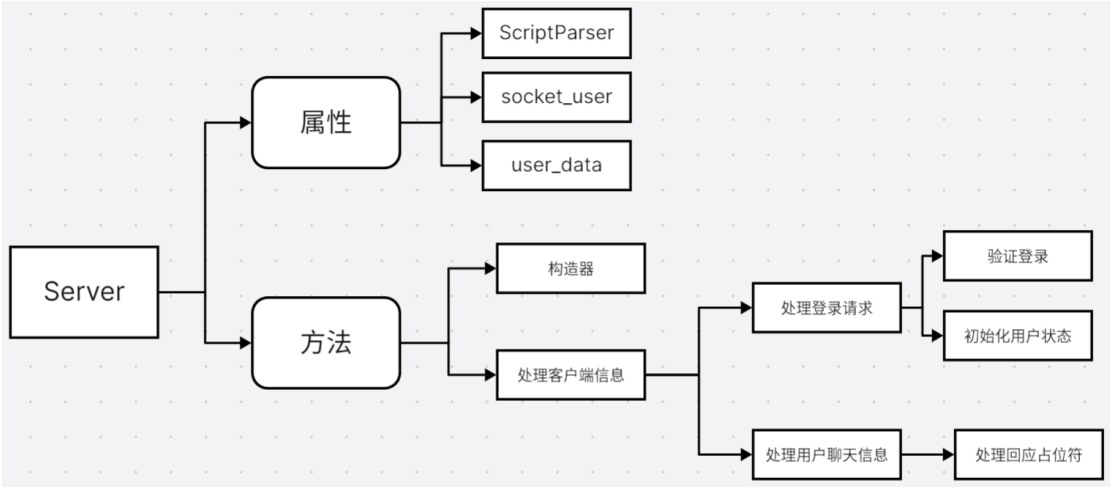


图 1-5 服务器数据结构示意图

将示意图转为伪代码如下：

```
1. Class: Server(BaseRequestHandler)
2.     Attribute: chat_robot
3.         类型: ScriptParser
4.         描述: 处理聊天消息的脚本解析器
5.
6.     Attribute: user_data
7.         类型: Dict[str, Vars]
8.         描述: 存储用户数据的字典, 键为用户名, 值为用户数据对象
9.
10.    Attribute: sockets_user
11.        类型: Dict[Socket, str]
12.        描述: 存储已连接的客户端套接字和对应的用户名的字典
13.
14.    Method: __init__(*args, **kwargs)
15.        初始化方法, 调用父类的初始化方法
16.
17.    Method: handle()
18.        处理客户端请求的方法: 出连接信息, 循环接收客户端消息, 解析并处理接收到的消息,
        处理异常, 关闭连接, 输出连接关闭信息
19.
20.    Method: handle_login(recv_data: Dict)
21.        处理登录请求消息的方法, 从接收到的消息中获取用户名和密码, 验证登录信息, 发送登
        录成功/失败消息, 存储登录成功信息
22.
```



23.     Method: `handle_chat(recv_data: Dict)`
24.         处理聊天消息的方法,获取用户名和消息数据,获取机器人回应和新状态,发送机器人回应消息,重置用户状态
- 25.
26.     Method: `handle_message(data: Dict)`
27.         处理接收到的消息的方法,获取消息类型,根据消息类型调用相应处理方法
- 28.
29.     Method: `validate_login(username: str, password: str) -> bool`
30.         验证登录信息的方法,检查用户名是否存在且密码正确
- 31.
32.     Method: `set_init_state(username: str)`
33.         设置登录用户的初始状态的方法,根据机器人的入口信息设置用户状态
- 34.
35.     Method: `explain_mark(response: str, username: str) -> str`
36.         解释回应信息中的占位符的方法,替换占位符为相应字符串

### 1.3.5 存放订单的数据结构 Order

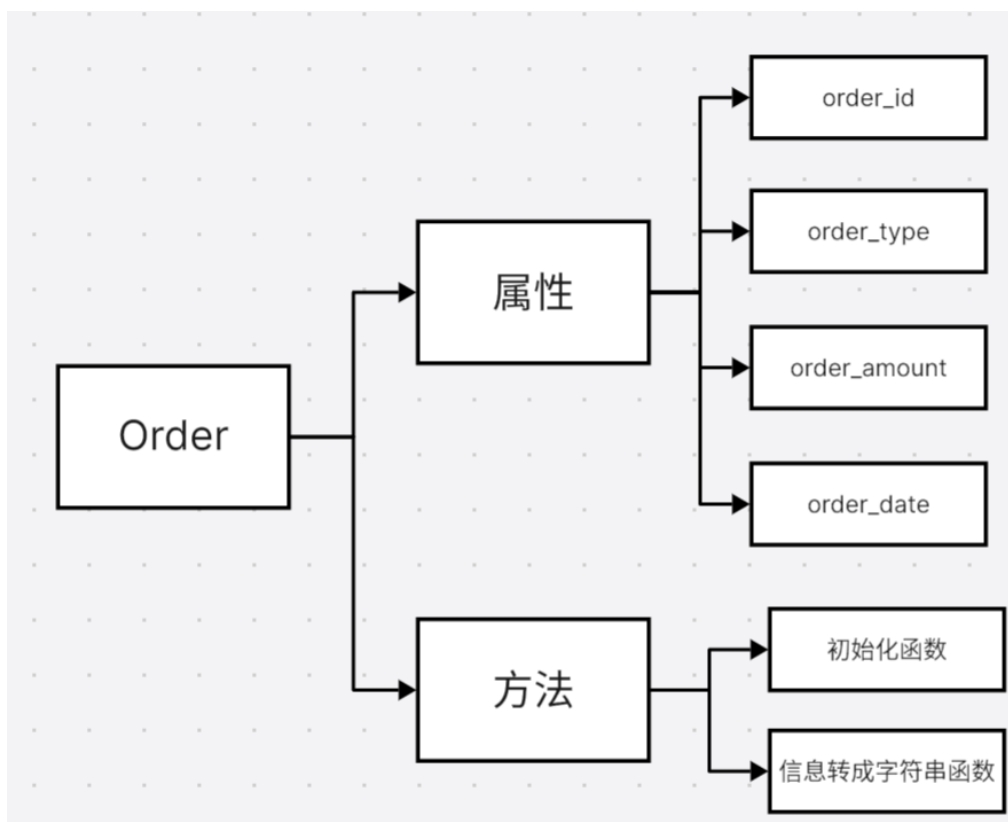


图 1-6 订单数据结构示意图

1. Class: `Order`
2.     Attribute: `order_id`
3.         类型: `str`

4.	描述: 订单号
5.	
6.	Attribute: order_type
7.	类型: str
8.	描述: 订单类型
9.	
10.	Attribute: order_amount
11.	类型: double
12.	描述: 订单金额
13.	
14.	Attribute: order_date
15.	类型: str
16.	描述: 订单时间
17.	
18.	Method: __init__(order_id: str, order_name: str, order_amount: double, order_date: str) -> None
19.	描述: 构造函数, 初始化订单属性
20.	
21.	Method: __str__() -> str
22.	描述: 返回订单信息的字符串表示

### 1.3.6 存放环境变量信息 Vars

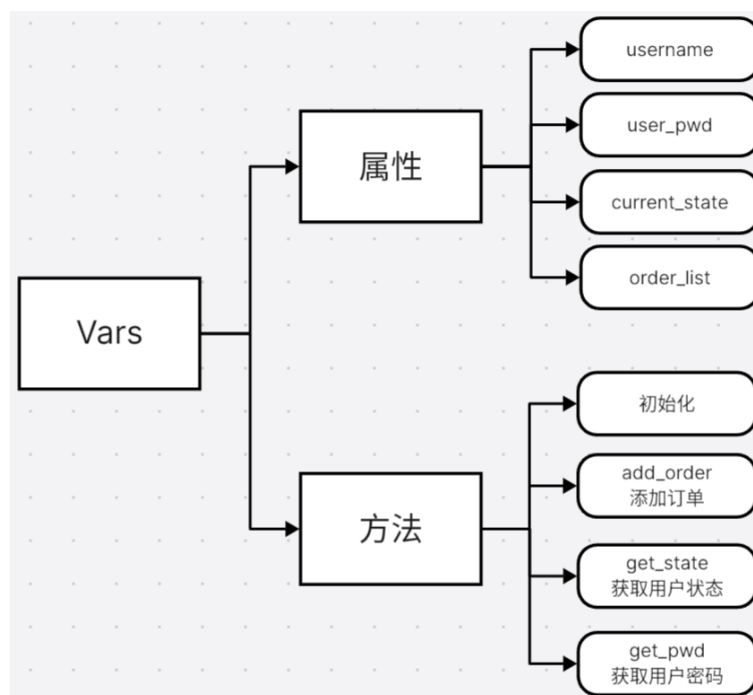


图 1-7 环境变量数据结构示意图

将上图转为伪代码及属性、方法说明，如下：

```
1. Class: Vars
2.     Attribute: user_name
3.         类型: String
4.         描述: 用户名
5.
6.     Attribute: user_pwd
7.         类型: String
8.         描述: 用户密码
9.
10.    Attribute: order_list
11.        类型: List[Order]
12.        描述: 订单列表
13.
14.    Attribute: current_state
15.        类型: String
16.        描述: 用户的当前状态
17.
18.    Method: __init__(user_name: String, user_pwd: String, order_list: List[Order], current_state: String = None) -> None
19.        描述: 构造函数，初始化用户环境变量
20.
21.    Method: add_order(order: String) -> None
22.        描述: 向订单列表中添加订单
23.
24.    Method: get_order_list() -> List[String]
25.        描述: 获取订单列表
26.
27.    Method: get_state() -> String
28.        描述: 获取用户当前状态
29.
30.    Method: set_state(state: String) -> None
31.        描述: 设置用户当前状态
32.
33.    Method: get_user_name() -> String
34.        描述: 获取用户名
35.
36.    Method: get_password() -> String
37.        描述: 获取用户密码
```

1.3.7 自动测试类 AutoTest

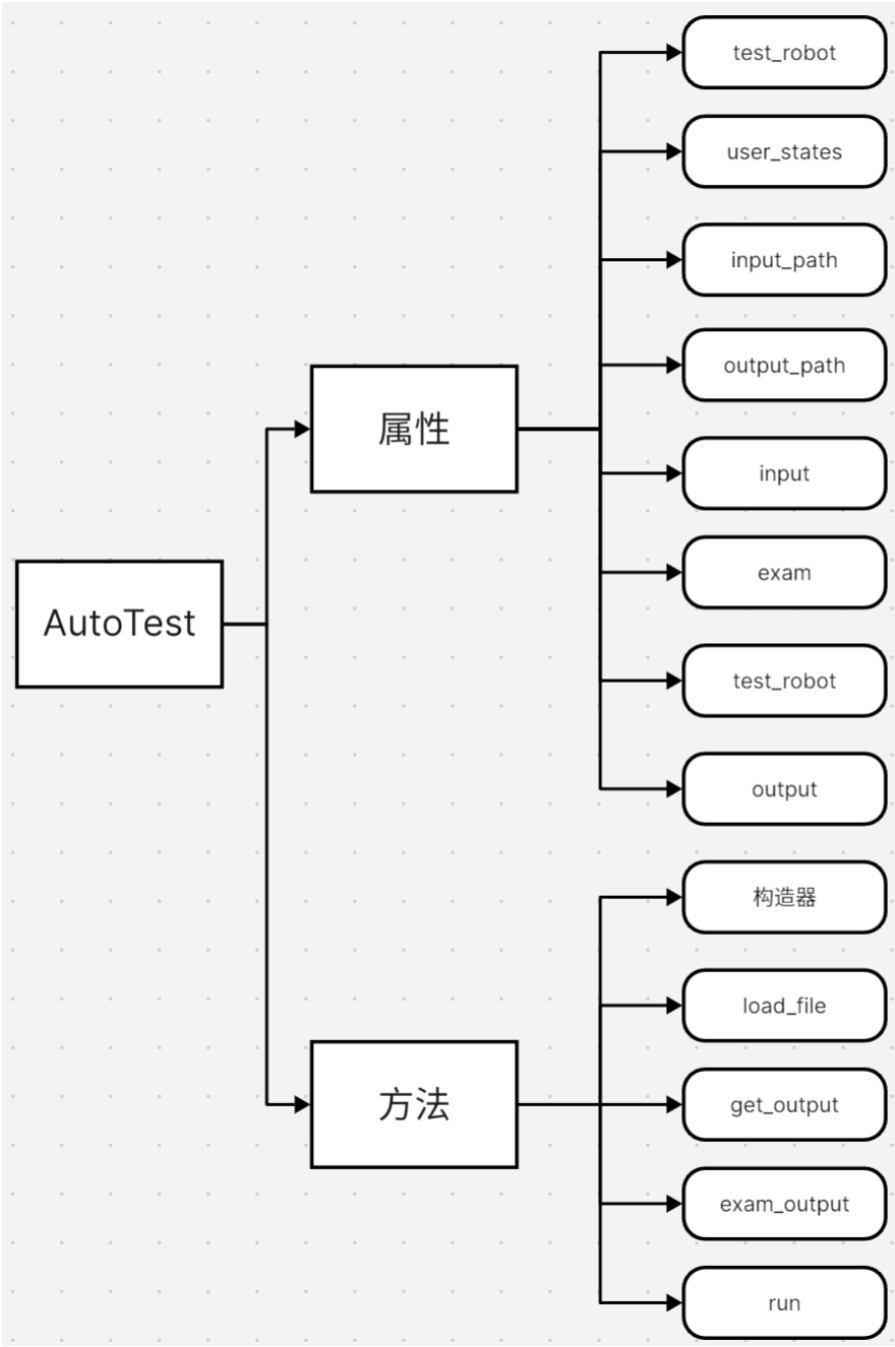


图 1-8 自动测试数据结构示意图

将上述数据结构示意图转为伪代码及说明，如下：

- 1. Class: AutoTest
- 2.     Attribute: test\_robot
- 3.         类型: ScriptParser
- 4.         描述: 脚本解析器对象
- 5.

6.	Attribute: user_states
7.	类型: String
8.	描述: 用户的当前状态
9.	
10.	Attribute: input_path
11.	类型: String
12.	描述: 输入文件路径
13.	
14.	Attribute: output_path
15.	类型: String
16.	描述: 输出文件路径
17.	
18.	Attribute: input
19.	类型: List[String]
20.	描述: 输入文本列表
21.	
22.	Attribute: exam
23.	类型: List[String]
24.	描述: 检验文本列表
25.	
26.	Attribute: output
27.	类型: List[String]
28.	描述: 生成的输出文本列表
29.	
30.	Method: __init__(input_path: String, output_path: String) -> None
31.	描述: 构造函数, 初始化 AutoTest 对象
32.	
33.	Method: load_file() -> None
34.	描述: 从输入文件和验证文件中加载数据到相应列表中
35.	
36.	Method: get_output() -> None
37.	描述: 获取生成的输出文本列表
38.	
39.	Method: exam_output() -> Boolean
40.	描述: 检查生成的输出是否与验证文本一致
41.	
42.	Method: run() -> None
43.	描述: 运行自动测试, 加载文件, 获取输出, 检查输出是否正确

## 2 接口说明

### 2.1 人机接口

#### 2.1.1 登录界面人机接口

交互流程：

用户在用户名和密码输入框输入信息。

用户点击登录按钮或按下 Enter 键。

用户输入的信息发送给服务器进行验证。

服务器返回登录结果。

如果登录成功，隐藏登录界面，显示聊天界面。如果失败，弹出提示框，并清空用户名和密码的输入栏，请求用户重写输入

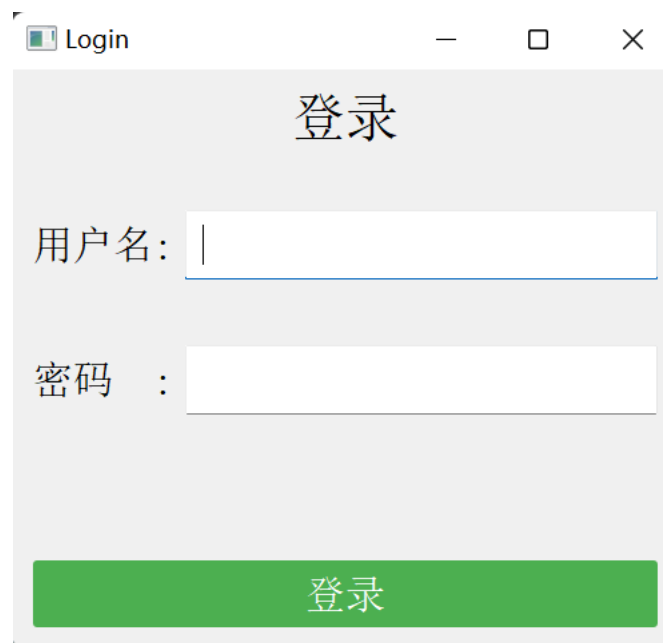


图 2-1 登录界面示意图

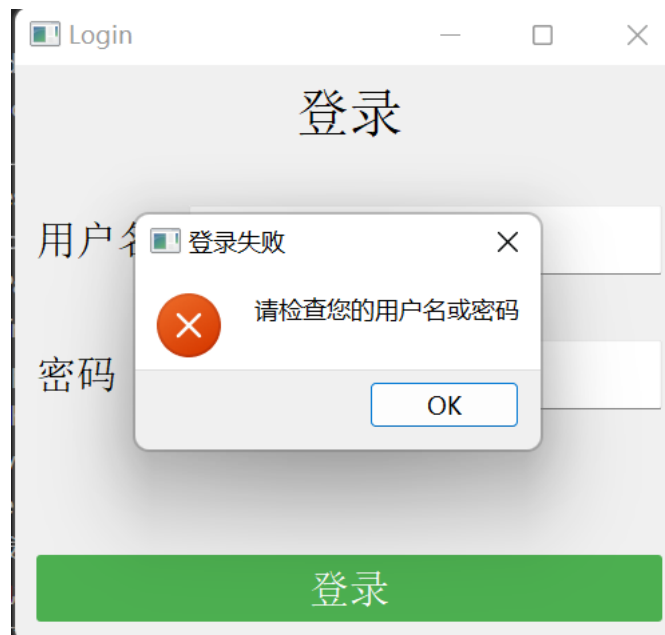


图 2-2 登录失败示意图

### 2.1.2 对话界面人机接口

交互流程：

用户在文本输入框输入消息。

用户点击“发送”按钮或按下 enter 键发送信息。

用户消息显示在界面上，并通过 socket\_manager 发送给服务器。

等待服务器的回应，接收机器人的消息。

机器人回应显示在界面上。

其中，用户的消息是白字绿底，机器人的回应则是黑字灰底

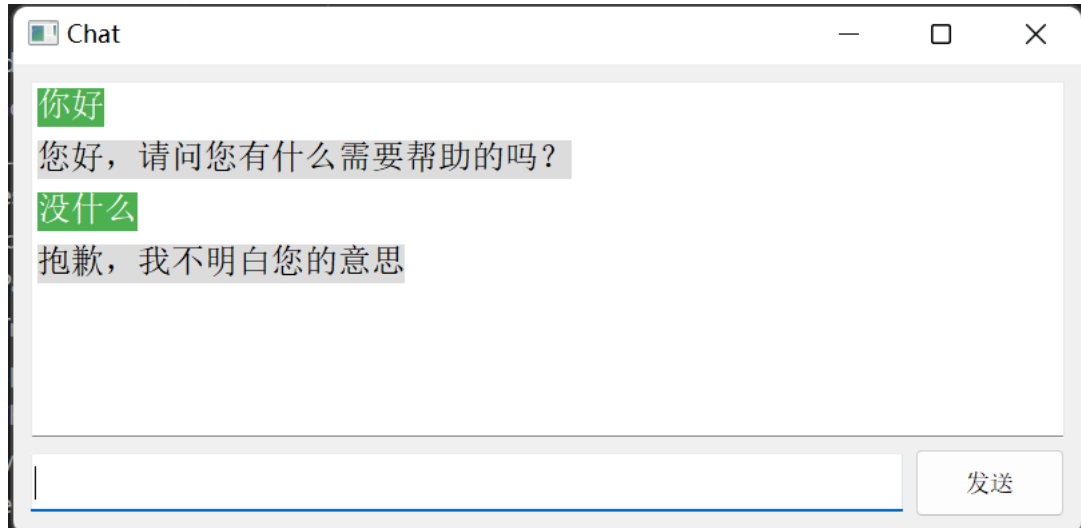


图 2-3 聊天界面示意图

## 2.2 前后端通信接口

### 2.2.1 消息协议

客户端：

登录消息：

```
{  
  "type": "login",  
  "username": "user",  
  "password": "pass"  
}
```

聊天消息：

```
{  
  "type": "chat",  
  "user-message": "Hello, bot!"  
}
```

服务端：

登录验证：

```
{  
  "login": True/False
```



```
}
```

聊天回应:

```
{  
  "bot-response": message  
}
```

本项目设计的消息类型明确,协议中定义了不同的消息类型(login和 chat),使得服务器能够根据消息类型执行相应的逻辑,提高了消息的可靠性。

字段结构简单明了,消息中的字段结构相对简单,容易解析。

协议可扩展性较强,协议的设计允许简单的扩展,可以根据需要添加新的消息类型或字段,以满足系统的不断演进。

使用 JSON 格式,协议中使用 JSON 格式进行消息的序列化和反序列化,这有助于减少由于数据格式错误导致的通信问题,提高了可靠性。

### 2.2.2 消息工作过程

用户登录流程:

客户端发送登录消息给服务器。

服务器验证用户名和密码。

服务器返回登录响应给客户端。

客户端根据响应结果进行相应操作,如果成功则隐藏登录界面,显示聊天界面。如果失败,则弹出弹窗,并帮助用户清空输入栏。

聊天流程:

客户端发送聊天消息给服务器。

服务器处理聊天消息,获取机器人的回应。

服务器返回聊天响应给客户端。

客户端显示机器人的回应聊天界面。

消息处理流程:

客户端接收到服务器发来的消息。

客户端根据消息类型分发消息处理函数。

如果是登录消息，调用 `handle_login` 处理登录响应。

如果是聊天消息，调用 `handle_chat` 处理聊天响应。

发送用户消息流程:

客户端获取用户在输入框中输入的消息。

客户端发送聊天消息给服务器。

服务器处理聊天消息，获取机器人的回应。

服务器返回聊天响应给客户端。

客户端显示用户和机器人的消息在聊天界面。

## 2.3 程序间接口—类与 API

### 2.3.1 ScriptTree 类的接口

**`get_response(user_input, state_name, ret_list=[ ])`**

概述：这个接口对于 `ScriptTree` 内部调用时可能需要传递参数 `ret_list`，用

于在 `ret_list` 中添加找到的回应信息。然而在外部的使用时，这个

`ret_list` 参数通常不需要传递进去，因为往往是还没获取到回应时调用

该函数来获取回应。

请注意: 返回的 `ret_list` 可能是个空列表, 表示并没有与用户输入匹配的回应, 这时候你需要在外部设置一个合适的默认回应。

参数说明:

@param user\_input 用户输入

@param state\_name 要查找回应的状态

@return ret\_list 回应列表

使用范例:

```
1. def get_response1(self, user_input, state_name):  
2.     return self.script_tree.get_response(user_input, state_name)
```

在本例中, 这个外部的 `get_response1` 函数调用了 `ScriptTree` 中的 `get_response` 函数, 并将获取到的回应列表返回。

## get\_entry()

概述: 该接口返回语法树的状态入口, 无参数

请注意: 当语法树中没有设置 `entry` 时, 会返回空值, 请在外部处理这个空值

使用范例:

```
1. # 设置登录的用户初始状态  
2. def set_init_state(self, username):  
3.     if chat_robot.get_entry() is None:  
4.         user_data[username].set_state("默认")  
5.         print("set state None")  
6.     else:  
7.         user_data[username].set_state(chat_robot.get_entry())  
8.         print("entry not none")
```

在本例中, 判断获取的 `entry` 是否为空, 如果为空, 则设置为默认值, 如果不为空, 则设置用户的初始状态为获取到的 `entry` 值。

## add\_state(name, cases)

概述：该接口用来给语法树添加状态动作树。

请注意：当 cases 格式不正确时会导致后续的解析出错，请检查您的 cases 格式是否正确

参数说明：

name：要添加的状态名称

cases：要添加的状态动作树，格式为：

```
{
  "trigger1":
    {"act_type1":"act_result1","act_type2":"act_result2"...}
  "trigger2":{"act_type1":"act_result3".....}
  "trigger3": {"act_type1":"act_result5".....}
  .....
}
```

使用范例：

```
1. # 遇到 state, 将上一个状态的状态树加入语法树中, 并初始化新的状态树
2. def parser_state(self, current_state, cases, line, row):
3.     pattern = r'state "(.*?)"'
4.     match = re.match(pattern, line)
5.     if match:
6.         # 将 cases 赋值给当前的 states, 并重置 cases
7.         self.script_tree.add_state(current_state, cases)
8.         current_state = match.group(1)
9.         self.script_tree.responses[current_state] = None
10.        return current_state
11.    else:
12.        raise ScriptError("请检查脚本文件的第{}行, state 后仅可跟随一个状态".format(row))
```

在本例中，先获取了 cases 字典，和状态名称，然后给语法树添加状态树。

## add\_init\_response(name, response)

概述：这个函数用来向语法树中添加一个状态的初始回应，即转到该状态的第一个回应。

参数说明：

name：状态名称，要添加初始回应的状态名称

response：初始回应，要添加的初始回应

使用范例：

```
1. def parser_response(self, current_state, line, row):
2.     pattern = r'response "(.*?)"'
3.     match = re.match(pattern, line)
4.     if match:
5.         response = match.group(1)
6.         if self.script_tree.responses[current_state] is not None:
7.             raise ScriptError("脚本文件第{}行,状态的初始 response 只能有一个".format(row))
8.         self.script_tree.add_init_response(current_state, response)
9.     else:
10.        raise ScriptError("脚本文件第{}行,状态的默认 response 后只能跟一个回应".format(row))
```

在本例中，将获取到的 response 添加到了语法树中。

## 2.3.2 ScriptParser 类的接口

### \_\_init\_\_(script\_path)

概述：本函数是 ScriptParser 类的构造器，用来创建解释器对象

参数说明：

script\_path 是待解释的脚本路径

使用范例

```
1. chat_robot = ScriptParser("../script_file/chatScript.bot")
```

## get\_entry()

概述：本函数用来获取语法树的 entry，调用了 ScriptTree 中的 get\_entry()函数

使用范例：

```
1. # 设置登录的用户初始状态
2. def set_init_state(self, username):
3.     if chat_robot.get_entry() is None:
4.         user_data[username].set_state("默认")
5.         print("set state None")
6.     else:
7.         user_data[username].set_state(chat_robot.get_entry())
8.         print("entry not none")
```

## 2.3.3 Order 类的接口

### \_\_init\_\_(order\_id,order\_name,order\_amount,order\_date)

概述：该函数为 Order 类的构造器，用来构造一个存储订单信息的数据结构

参数说明：

order\_id: 订单号，为字符串

order\_name: 购买商品的名称，为字符串

order\_amount: 该订单的金额，为浮点型

order\_date: 该订单的日期，为字符串

使用范例：

```
1. test_order4 = Order(4444444444, "平板电脑", 2688, "2023-11-27")
```

该范例构建了一个订单号为 4444444444，商品名称为平板电脑，金额为 2688，

日期为 2023-11-27 的订单

## **\_\_str\_\_()**

概述：该函数定义了打印这个类时的格式

使用范例：

```
2. print(test_order4)
```

直接打印即可。

## **2.3.4 Vars 类的接口**

### **\_\_init\_\_(user\_name, user\_pwd, order\_list)**

概述：该函数为环境变量的构造器，传入用户名、用户密码、用户的订单列表来创建一个用户变量。

使用范例：

```
1. test_order1 = Order(1111111111, "手机", 1688, "2023-11-24")
2. test_order2 = Order(2222222222, "电脑", 6888, "2023-11-05")
3.
4. test_order_list1 = [test_order1, test_order2]
5. test_user1 = Vars("1", "", test_order_list1)
```

### **add\_order(Order)**

概述：该函数用来为环境变量添加订单信息。

### **get\_order\_list()**

概述：该函数用来获取用户的订单列表

使用范例：

```
1. # 如果回应信息里有占位符，则要将占位符转换成相应字符串
2. def explain_mark(self, response, username):
3.     username_pattern = re.compile(r'\$username')
4.     order_pattern = re.compile(r'\$order')
5.     username_match = username_pattern.search(response)
6.     order_match = order_pattern.search(response)
7.     if username_match:
8.         response = response.replace('$username', username)
9.     if order_match:
10.         order = user_data[username].get_order_list()
11.         order_str = "".join(str(order) for order in order)
12.         response = response.replace('$order', order_str)
13.     return response
```

该示例中，获取了订单列表，并获取每个订单信息，放入回应的消息中

## get\_state()

概述：用来返回用户当前的状态。

使用范例：

```
1. bot_resp_list, state = chat_robot.get_response(data, user_data[username].get_state())
```

该范例中，获取了用户的信息并传递到另一个函数中

## set\_state()

概述：用来设置用户的状态。

```
1. if chat_robot.get_entry() is None:
2.     user_data[username].set_state("默认")
3.     print("set state None")
4. else:
5.     user_data[username].set_state(chat_robot.get_entry())
6.     print("entry not none")
```

本例中，使用 set\_state 用来设置用户的状态



## get\_password()

概述：用来获取用户的密码。

```
1. def validate_login(self, username, password):
2.     print("validating...")
3.     try:
4.         if username in user_data and user_data[username].get_password
           () == password:
5.             return True
6.     except Exception as e:
7.         print(f"处理客户端的时候出错: {self.client_address}: {e}")
8.     return False
```

本例中，获取了用户的密码并和登录的请求密码进行比对。