Customer Service DSL

Developed by Carl Shen

1. 项目概述

1.1 项目简介

本项目设计了一个用于描述在线客服机器人自动应答逻辑的领域特定脚本语言（DSL），以及配套的解释器与UI界面系统。让客服程序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。

1.2 主要内容

- 用于描述客服机器人逻辑的DSL语法
- 基于`PyParsing`库的DSL语法分析器
- 基于有限状态自动机模型的DSL解释器
- 基于`Flask`框架的服务端
- 基于RESTful风格的API
- 基于`PyWebIO`框架的客户端及网页页面

2. 安装和配置

2.1 安装依赖

对于希望自行部署的用户，建议使用项目文件中的requirements.txt来安装依赖：

```
conda create -n DSL python=3.12
pip install -r requirements.txt
```

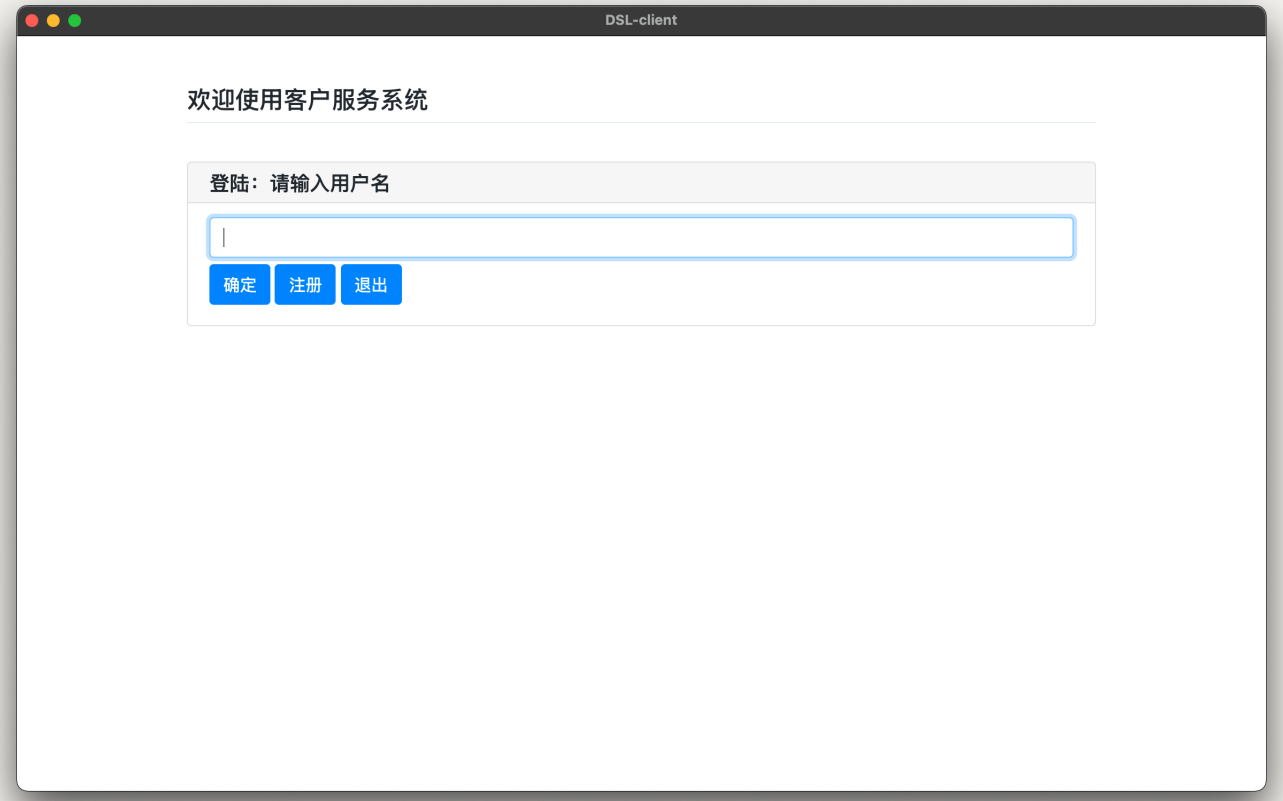
注：作者基于macOS系统进行开发，Windows环境下PyQt框架依赖的版本号可能与macOS版本不一致，建议Windows用户在创建虚拟环境时使用更低的Python版本并手动安装依赖

2.2 客户端

客户端使用PyWebIO框架搭建，使用asyncio库实现了消息的异步刷新。

2.1.1 客户端程序

使用`PyQt`的`QtWebEngine`组件封装为一个独立的应用程序。用户下载即用，无需特殊配置。



如果开发者想使用自己的服务器地址，可以在WebUI路径下找到`DSL_webui.py`和`web.py`两个文件，将`web.py`文件中`URL`修改为希望使用的服务器地址，然后在项目文件中找到`DSL_webUI.spec`文件，使用以下命令打包：

```
pyinstaller DSL_webUI.spec
```

2.1.2 网页版客户端

可以把PyWebIO实例直接部署在服务器上，用户通过浏览器可以以网页形式直接访问。

使用者也可以在本地的DSL-client路径下，使用以下命令在本地部署：

```
python webUI.py
```

如果开发者想在服务端开启页面访问，可以在服务器的DSL-server路径下通过以下命令开启网页客户端：

```
cd WebUI
python webUI_server.py
```

2.3 服务端

服务端使用YAML脚本进行配置，使用者需要现在与`DSL-server`目录下创建一个`config.yaml`文件，填入将要部署的目标主机地址、端口号，以及希望使用的DSL脚本路径。

```yaml
server:
  host: 'your.ip.address.0'
  port: 5000
script:
  current_script: 'scripts/example_script.txt'
```

然后在命令行通过以下命令启动：

```
python app.py
```

3. 项目模块介绍

3.1 DSL语法

3.1.1 语法定义

本DSL由以下EBNF范式定义：

```
<DSL_language>      ::= <version_info> {<state_definition> |
<var_definition>} <exit_operation>
<version_info>      ::= "VERSION" <digit>+ "." <digit>+ "."
<digit>+
<digit>             ::= "0" | "1" | ... | "9"
<state_definition>  ::= "STATE" <state_id> <response>
{<response> | <match_clause>} <default_clause>
```

```
<state_id>          ::= <letter>+
<letter>            ::= "A" | "B" | ..."Z" | "a" | "b" | ... | "z"
<response>          ::= "RESPONSE" (<string> | <var_id>) {"+" (<string> | <var_id>)}
<match_clause>      ::= "MATCH" <string> {<response> | <var_assignment>} <goto_clause>
<default_clause>    ::= "DEFAULT" {<response> | <var_assignment>} <goto_clause>
<string>            ::= """ (<letter> | <digit>)+ """
<integer>           ::= ["+" | "-"] <digit>+
<float>             ::= ["+" | "-"] <digit>+ "." <digit>+
<goto_clause>       ::= "GOTO" (<state_id> | "EXIT")
<var_definition>    ::= "VAR" ("INT" <var_id> <integer>) | ("FLOAT" <var_id> <float>) | ("STR" <var_id> <string>)
<var_id>            ::= "$" (<letter> | <digit>)+ "$"
<var_assignment>    ::= "ASSIGN" ("INT" <var_id> <expr>) | ("FLOAT" <var_id> <expr>) | ("STR" <var_id> <string>)
<expr>              ::= <term> { ("+" | "-") <term>}
<term>              ::= <factor> { ("*" | "/") <factor> }
<factor>            ::= <number> | "(" <expr> ")"
<number>            ::= <integer> | <float> | <var_id>
<exit_operation>    ::= "EXIT" {<response>}
```

3.1.2 语法描述

注：下列描述中‘’标记表示元素的数量是可选的(0 - n)*

1. **语法概述**

   一个完整的DSL脚本由以下三部分组成：

   - VERSION字段
   - 脚本主体（由变量定义和状态定义构成）
   - EXIT字段

2. **VERSION字段**

   用于标记当前DSL脚本的版本，为后续开发的兼容性做准备

3. 变量定义

   变量定义语句的语法结构如下：

   - `VAR`关键字
   - 变量类型：`STR`、`INT`、`FLOAT`之一
   - 变量标识符：由'$'包围的任意字母、数字、下划线的组合
   - 初始值

   示例：

```
VAR STR $name$ "Carl"
VAR INT $age$ 80
```

4. 状态定义

   状态定义语句的语法结构如下：

   - `STATE`关键字
   - 状态标识符：任意字母组合
   - 回复操作语句：用于产生回应内容
   - *匹配操作语句：用于匹配用户输入内容，若匹配成功则执行一系列操作
   - 默认操作语句：在匹配操作语句匹配不成功的情况下执行的内容

   示例:

```
STATE main
    RESPONSE "您好，请问您有什么需要帮助的吗"
    RESPONSE "现支持以下功能：退休金查询"
    MATCH "退休金查询"
        GOTO query
    MATCH "退出"
        GOTO EXIT
    DEFAULT
        RESPONSE "抱歉，我没有理解您的意思"
        GOTO main
```

提示：在本DSL当中，为确保脚本能够正常运行，必须声明一个main状态用于初始状态！

5. 回复操作语句

回复操作语句的语法结构如下：

- `RESPONSE`关键字
- 一个变长的字符串：支持多字符串以及变量的拼接，用'+'连接

示例：

```
RESPONSE "您已成功提取" + $money$ + "元"
```

6. 匹配操作语句

匹配操作语句的语法结构如下：

- `MATCH`关键字
- 一个常量模式串：用于匹配用户的输入
- *回复操作语句：输出指定信息
- *变量赋值语句：给指定变量赋值
- 转移操作语句：转移到指定状态

示例：

```
MATCH "退休金提取"
    RESPONSE "正在提取..."
    GOTO fetch
```

7. 默认操作语句

默认操作语句的语法结构如下：

- `DEFAULT`关键字
- *回复操作语句：输出指定信息
- *变量赋值语句：给指定变量赋值
- 转移操作语句：转移到指定状态

示例：

```
DEFAULT
    RESPONSE "抱歉，我没有理解您的意思"
    GOTO main
```

8. **转移操作语句**

   默认操作语句的语法结构如下：

   - `GOTO`关键字
   - 一个状态名或者"EXIT"关键字

   示例：

   ```
   GOTO fetch
   GOTO EXIT
   ```

9. **变量赋值语句**

   变量赋值语句的语法结构如下：

   - `ASSIGN`关键字
   - 变量类型
   - 变量标识符
   - 一个算术表达式：支持加减乘除计算，操作数可以是常量或变量

   示例：

   ```
   ASSIGN FLOAT $money$ $money$ * 1.25 + 100
   ```

10. **EXIT字段**

    EXIT字段包含：

    - `EXIT`关键字
    - *回复操作语句：输出指定信息

    示例：

    ```
    EXIT
        RESPONSE "感谢您的使用，祝您生活愉快！"
    ```

3.2 语法分析器

本项目使用PyParsing库实现了一个能够解析此DSL的语法分析器。分析器会接受使用者提供的DSL脚本，如果脚本合法，那么将输出一个嵌套列表形式的语法树。

对以下示例脚本进行解析：

```
VERSION 1.0.0

VAR STR $book$ "《操作系统》"
VAR INT $available_copies$ 3

STATE main
    RESPONSE "欢迎来到图书馆，请选择服务："
    RESPONSE "【借阅图书】【退出】"
    MATCH "借阅图书"
        GOTO borrowbook
    MATCH "退出"
        GOTO EXIT
    DEFAULT
        RESPONSE "抱歉，我没有理解您的意思"
        GOTO main

STATE borrowbook
    RESPONSE "借阅成功！您已借阅：" + $book$
    RESPONSE "【确定】【返回】"
    MATCH "确定"
        RESPONSE "借阅成功！您已借阅：" + $book$
        ASSIGN INT $available_copies$ $available_copies$ - 1
        GOTO main
    MATCH "退出"
        GOTO main
    DEFAULT
        RESPONSE "抱歉，我没有理解您的意思"
        GOTO main

EXIT
    RESPONSE "感谢使用图书馆服务，再见！"
```

调用parser类的parse_file()方法解析后，分析器输出以下嵌套列表：

```
[['VERSION', '1.0.0'], ['VAR', 'STR', '$book$', '《操作系统》'], ['VAR', 'INT', '$available_copies$', 3], ['STATE', 'main', ['RESPONSE', '欢迎来到图书馆，请选择服务：'], ['RESPONSE', '【借阅图书】【退出】'], ['MATCH', '借阅图书', ['GOTO', 'borrowbook']], ['MATCH', '退出', ['GOTO', 'EXIT']], ['DEFAULT', ['RESPONSE', '抱歉，我没有理解您的意思'], ['GOTO', 'main']]], ['STATE', 'borrowbook', ['RESPONSE', '借阅成功！您已借阅：', '$book$'], ['RESPONSE', '【确定】【返回】'], ['MATCH', '确定', ['RESPONSE', '借阅成功！您已借阅：', '$book$'], ['ASSIGN', 'INT', '$available_copies$', [['$available_copies$']], '-', [[1]]]], ['GOTO', 'main']], ['MATCH', '退出', ['GOTO', 'main']], ['DEFAULT', ['RESPONSE', '抱歉，我没有理解您的意思'], ['GOTO', 'main']]], ['EXIT', ['RESPONSE', '感谢使用图书馆服务，再见！']]]
```

以树状格式输出：

```
├──┐
│  ├── VERSION
│  └── 1.0.0
├──┐
│  ├── VAR
│  ├── STR
│  ├── $book$
│  └── 《操作系统》
├──┐
│  ├── VAR
│  ├── INT
│  ├── $available_copies$
│  └── 3
├──┐
│  ├── STATE
│  ├── main
│  ├──┐
│  │  ├── RESPONSE
│  │  └── 欢迎来到图书馆，请选择服务：
│  ├──┐
│  │  ├── RESPONSE
│  │  └── 【借阅图书】【退出】
```

```
23  │   ├──┐
24  │   │  ├── MATCH
25  │   │  ├── 借阅图书
26  │   │  └──┐
27  │   │     ├── GOTO
28  │   │     └── borrowbook
29  │   ├──┐
30  │   │  ├── MATCH
31  │   │  ├── 退出
32  │   │  └──┐
33  │   │     ├── GOTO
34  │   │     └── EXIT
35  │   └──┐
36  │      ├── DEFAULT
37  │      ├──┐
38  │      │  ├── RESPONSE
39  │      │  └── 抱歉，我没有理解您的意思
40  │      └──┐
41  │         ├── GOTO
42  │         └── main
43  ├──┐
44  │  ├── STATE
45  │  ├── borrowbook
46  │  ├──┐
47  │  │  ├── RESPONSE
48  │  │  ├── 借阅成功！您已借阅：
49  │  │  └── $book$
50  │  ├──┐
51  │  │  ├── RESPONSE
52  │  │  └── 【确定】【返回】
53  │  ├──┐
54  │  │  ├── MATCH
55  │  │  ├── 确定
56  │  │  ├──┐
57  │  │  │  ├── RESPONSE
58  │  │  │  ├── 借阅成功！您已借阅：
59  │  │  │  └── $book$
60  │  │  ├──┐
61  │  │  │  ├── ASSIGN
```

```
62  │   │   │   ├── INT
63  │   │   │   ├── $available_copies$
64  │   │   │   └──┐
65  │   │   │      ├──┐
66  │   │   │      │  └──┐
67  │   │   │      │     └── $available_copies$
68  │   │   │      ├── -
69  │   │   │      └──┐
70  │   │   │         └──┐
71  │   │   │            └── 1
72  │   │   └──┐
73  │   │      ├── GOTO
74  │   │      └── main
75  │   ├──┐
76  │   │   ├── MATCH
77  │   │   ├── 退出
78  │   │   └──┐
79  │   │      ├── GOTO
80  │   │      └── main
81  │   └──┐
82  │      ├── DEFAULT
83  │      ├──┐
84  │      │   ├── RESPONSE
85  │      │   └── 抱歉，我没有理解您的意思
86  │      └──┐
87  │         ├── GOTO
88  │         └── main
89  └──┐
90     ├── EXIT
91     └──┐
92        ├── RESPONSE
93        └── 感谢使用图书馆服务，再见！
```

解析生成的语法树将会被提供给DFA模块，解释生成一个对应脚本逻辑的状态机模型。

3.3 自动机模型

现实生活中，客服的逻辑大多都是问-答逻辑。将对话过程抽象成状态机进行处理。在我的程序中使用有限状态自动机（DFA）来描述机器人客服的对话逻辑。

3.3.1 DFA类

用于描述客服机器人对话逻辑的有限状态自动机

类定义

```python
class DFA():
    def __init__(self, file: str):
        self.states: list[str] = []
        self.state_operations: dict[str, list] = {}
        self.current = CurrentState()
        try:
            grammar_tree = parser.parse_script(file)
        except ParseException as err:
            raise GrammarException(err.__str__(), [err.line])
        # 分析脚本逻辑
```

如果成功解析脚本得到语法树后，`DFA`的构造函数首先会遍历一遍语法树，从中找出变量定义并将变量信息添加到`self.current`属性的`var_table`当中。同时在此次遍历的过程中，还会提取出所有状态语句中声明的状态名。如果状态名存在重复或者不包含main状态，则认为不符合语法，抛出异常。第一遍遍历还会构造一个`Exit_operation`实例用于表示退出相关操作。

随后进行第二次遍历，对所有状态定义进行分析。对于每一个STATE语句下的操作语句，转化为对应的`Operation`实例并添加到`self.state_operations`对应的位置。自此完成了对DFA的构造

属性

- `states` (list[str]): #用于标记DFA的所有状态
- `state_operations` (dict[str, list]): 用于记录对应状态下的一系列操作序列。
- `current` (CurrentState): 一个CurrentState类的实例，用于记录DFA 的当前状态。

方法

`__init__(file: str)`

初始化 DFA 类，使用给定的脚本文件。

- 参数：
    - `file` (str)：脚本文件的路径。

`get_state_hint()`

获取直到下一个MATCH操作之前的所有信息。

`state_transition(message: str)`

根据给定的消息使 DFA 转换到下一个状态。

- 参数：
    - `message` (str): 状态转换的输入消息。

3.3.2 CurrentState 类

`CurrentState` 类用于记录当前状态，每个 DFA 都拥有一个该类的实例作为自身属性。

类定义

```python
class CurrentState:
    def __init__(self):
        self.state = 'main'
        self.var_table: dict[str, dict] = {}
        self.operation_index = 0
        self.is_running = True
```

属性

- `state` (str): 当前状态的状态名`。
- `var_table` (dict[str, dict]): 变量表，用于存储变量的类型和值。
- `operation_index` (int): 当前状态下执行操作的索引。
- `is_running` (bool): DFA 是否正在运行。

方法

`__init__()`

初始化 `CurrentState` 类。

3.3.3 Operation 类

`Operation` 类是所有操作类的父类。

类定义

```
class Operation:
    def __init__(self):
        pass
```

方法

`__init__()`

初始化 `Operation` 类。

3.3.4 Response_operation 类

`Response_operation` 类用于向用户返回当前状态下的提示信息。

类定义

```
class Response_operation(Operation):
    def __init__(self, phrases: list[str], c: CurrentState):
        self.phrases = phrases
        for phrase in self.phrases:
            if phrase.startswith("$") and phrase.endswith("$"):
                if phrase not in c.var_table:
                    raise GrammarException("Inexplicit
variable:", phrase)
```

属性

- `phrases` (list[str]): 提示信息的短语列表。

方法

`__init__(phrases: list[str], c: CurrentState)`

初始化 `Response_operation` 类。

- 参数:
    - `phrases` (list[str]): 提示信息的短语列表。
    - `c` (CurrentState): 当前状态实例。

`exec(c: CurrentState)`

执行操作，返回提示信息。

- 参数:
    - `c` (CurrentState): 当前状态实例。
- 返回:
    - (list[str]): 提示信息列表。

3.3.5 Goto_operation 类

`Goto_operation` 类用于实施状态的转移以及标记 DFA 的运作状态。

类定义

```
class Goto_operation(Operation):
    def __init__(self, next_state: str, c: CurrentState):
        self.next = next_state
```

属性

- `next` (str): 下一个状态。

方法

`__init__(next_state: str, c: CurrentState)`

初始化 `Goto_operation` 类。

- 参数:
    - `next_state` (str): 下一个状态。
    - `c` (CurrentState): 当前状态实例。

`exec(c: CurrentState)`

执行状态转移操作。

- 参数:
    - `c` (CurrentState): 当前状态实例。

3.3.6 Default_operation 类

`Default_operation` 类用于每一个 `STATE` 语句下的默认操作。

类定义

```
class Default_operation(Operation):
    def __init__(self, ops: list, c: CurrentState):
        self.operations: list[Operation] = []
        for op in ops:
            if op[0] == "RESPONSE":

 self.operations.append(Response_operation(op[1:], c))
            elif op[0] == "ASSIGN":
                self.operations.append(Var_assignment(op[1:],
c))
            elif op[0] == "GOTO":
                self.operations.append(Goto_operation(op[1], c))
```

属性

- `operations` (list[Operation]): 操作列表。

方法

`__init__(ops: list, c: CurrentState)`

初始化 `Default_operation` 类。

- 参数:
    - `ops` (list): 操作列表，是语法树的子树。
    - `c` (CurrentState): 当前状态实例。

`exec(c: CurrentState)`

执行默认操作。

- 参数:
    - `c` (CurrentState): 当前状态实例。
- 返回:
    - (list[str]): 响应信息列表。

3.3.7 Var_assignment 类

`Var_assignment` 类用于表示变量赋值操作。

类定义

```
class Var_assignment(Operation):
    def __init__(self, info: list, c: CurrentState):
        self.var_type = info[0]
        self.var_id = info[1]
        self.expr = info[2]
        if self.var_id not in c.var_table:
            raise GrammarException("Undefined variable.",
info[1])
        if self.var_type != c.var_table[self.var_id]["type"]:
            raise TypeException("Conflicted variable type:", f"
{self.var_type} and {c.var_table[self.var_id]["type"]}")
        if self.var_type != "STR" and isinstance(self.expr,
str):
            raise TypeException("Conflicted variable type:", f"
{self.var_type} and {c.var_table[self.var_id]["type"]}")
```

在`Var_assignment` 的实例构造时会对语句的内容进行语义分析，检查变量类型是否合法，否则会抛出`TypeException`异常。

属性

- `var_type` (str): 目标变量类型。
- `var_id` (str): 目标变量标识符。
- `expr` (list): 表达式，是语法树的子树，提供给`evaluate_parsed_expression`方法来获取当前值。

方法

`__init__(info: list, c: CurrentState)`

初始化 `Var_assignment` 类。

- 参数:
    - `info` (list): 包含变量类型、标识符和表达式的信息列表。
    - `c` (CurrentState): 当前状态实例。

`evaluate_parsed_expression(parsed_expr, c: CurrentState)`

计算表达式在当前状态的值。

- 参数:
    - `parsed_expr` (list): 解析后的表达式。
    - `c` (CurrentState): 当前状态实例。
- 返回:
    - (int/float): 表达式的计算结果。

`exec(c: CurrentState)`

执行赋值操作。

- 参数:
    - `c` (CurrentState): 当前状态实例。

3.3.8 Exit_operation 类

`Exit_operation` 类用于 DFA 结束运行时执行的操作。

类定义

```
class Exit_operation(Operation):
    def __init__(self, ops: list, c: CurrentState):
        self.operations: list[Operation] = []
        for op in ops:
            if op[0] == "RESPONSE":
                self.operations.append(Response_operation(op[1:],
c))
```

属性

- `operations` (list[Operation]): 操作列表。

方法

`__init__(ops: list, c: CurrentState)`

初始化 `Exit_operation` 类。

- 参数:
    - `ops` (list): 操作列表。
    - `c` (CurrentState): 当前状态实例。

`exec(c: CurrentState)`

执行退出操作。

- 参数:
    - `c` (CurrentState): 当前状态实例。
- 返回:
    - (list[str]): 响应信息列表。

3.3.9 Match_operation 类

`Match_operation` 类用于判断用户输入是否与模式相同。

类定义

```
class Match_operation(Operation):
    def __init__(self, pattern: str, ops: list, c:
CurrentState):
        self.pattern = pattern
        self.operations: list[Operation] = []
        for op in ops:
            if op[0] == "RESPONSE":

 self.operations.append(Response_operation(op[1:], c))
            elif op[0] == "ASSIGN":
                self.operations.append(Var_assignment(op[1:],
c))
            elif op[0] == "GOTO":
                self.operations.append(Goto_operation(op[1], c))
```

属性

- `pattern` (str): 匹配模式。
- `operations` (list[Operation]): 操作列表。

方法

`__init__(pattern: str, ops: list, c: CurrentState)`

初始化 `Match_operation` 类。

- 参数:
    - `pattern` (str): 匹配模式。
    - `ops` (list): 操作列表，是语法树的子树。
    - `c` (CurrentState): 当前状态实例。

`exec(c: CurrentState, input: str)`

执行匹配操作。

- 参数:

    - `c` (CurrentState): 当前状态实例。
    - `input` (str): 用户输入。
- 返回:
    - (list[str]/None): 响应信息列表或 `None`，返回`None`用于标记匹配不成功。

3.4 服务端

3.4.1 服务端功能概述

服务端使用Flask框架作为主体进行开发，主要实现了用户管理以及客服对话功能。用户信息被存放在服务器的MySQL数据库上，使用SqlAlchemy库作为ORM模型来操作数据库。

服务端启动时首先从config.yaml读取配置信息，然后创建以下变量：

- `lock`：一个`Lock`类的实例，用于维护线程安全，确保共享变量的互斥访问。
- `dfa_group`:一个字典，以用户名为键，记录每个用户对应的DFA实例。
- `user_timers`:一个字典，以用户名为键，保存每个用户的Timer实例。一旦计时器到时则设置用户状态为离线。

3.4.2 功能函数

`reset_timer`函数

`reset_timer`函数用于重置用户的计时器。当用户上线或有操作时，刷新计时器的 5 分钟倒计时。

函数定义

```python
def reset_timer(username):
    """
    重置用户的计时器。对于每一个用户，上线或有操作时刷新timer的5min倒计时

    :param username: 用户名
    :type username: str
    """
    if username in user_timers:
        user_timers[username].cancel()
    timer = Timer(300, set_user_offline, args=[username])  # 5分钟（300秒）计时器
    user_timers[username] = timer
    timer.start()
```

参数

- `username`(str): 用户名。

功能

- 检查用户是否已有计时器，如果有则取消当前计时器。
- 创建一个新的 5 分钟（300 秒）计时器，当计时器超时时调用 `set_user_offline` 函数。
- 将新的计时器存储在`user_timer`字典中，并启动计时器。

`set_user_offline` 函数

`set_user_offline`函数用于在用户登出或者计时器超时的情况下将用户设置为离线状态，删除计时器以及此用户对应的 DFA。

函数定义

```
def set_user_offline(username):
    """
    在用户登出或者计时器超时的情况下将用户设置为离线状态，删除计时器以及此用户对应的DFA

    :param username: 用户名
    :type username: str
    """
    with lock:
        if username in user_timers:
            print(f'User {username} is set to offline due to inactivity.')
            user_timers.pop(username, None)
            if username in dfa_group:
                del dfa_group[username]
```

参数

- `username`(str): 用户名。

功能

- 使用互斥锁确保线程安全。
- 检查用户是否有计时器，如果有则删除计时器。
- 打印用户离线的日志信息。
- 检查用户是否有对应的 DFA 实例，如果有则删除该实例，释放资源。

3.4.3 API

本服务端采用Flask框架提供的方法封装了RESTful风格的API。

注册用户

- **URL**: `/register`
- 方法: `POST`

- 请求体:

```
{
  "username": "newuser",
  "password": "newpassword"
}
```

- 响应:
  - 成功:

```
{
  "status": "success"
}
```

    状态码：200

  - 失败:

```
{
  "status": "注册失败：用户名已存在"
}
```

    状态码：400

- 功能描述

  当客户端发送注册请求时，服务端会先从数据库进行查询。如果用户名已经存在那么返回失败的状态信息，否则将用户信息写入数据库并返回成功。

用户登录

- **URL**: `/login`
- 方法: `POST`
- 请求体:

```
{
  "username": "newuser",
  "password": "newpassword"
}
```

- 响应:
    - 成功:

      ```
      {
        "status": "success",
        "hello_info": ["欢迎信息"]
      }
      ```

      状态码：200
    - 用户名密码不匹配:

      ```
      {
        "status": "用户不存在或密码错误"
      }
      ```

      状态码：401
    - 用户已在线:

      ```
      {
        "status": "用户已在线"
      }
      ```

      状态码：409
- 功能描述

  当客户端发送登陆请求时，首先服务端会通过`user_timers`检查此用户是否已经在线。如果已经在线返回409，否则把信息提供给数据库。如果用户信息不存在则返回401。如果存在对应信息则认定成功登陆，通过`reset_timer`创建计时器，并为这个用户创建一个DFA实例保存在`dfa_group`用于后续的对话逻辑，同时调用DFA的`get_state_hint`方法获取初始状态的提示信息，放置在`hello_info`字段。

用户登出

- **URL**: `/logout`
- 方法: `POST`
- 请求体:

```
{
  "username": "newuser"
}
```

- 响应:

```
{
  "message": "Logout successful"
}
```

状态码：200

- 功能描述

接受客户端的登出请求，将对应的用户状态通过`set_user_offline`函数设置为离线，释放资源。

聊天

- **URL**: `/chat`
- 方法: `POST`
- 请求体:

```
{
  "username": "newuser",
  "message": "Hello"
}
```

- 响应:
  - 成功:

```
{
  "status": "success",
  "running": true,
  "reply": ["回复信息"],
  "bye": 0
}
```

状态码：200

- 失败：

```
{
  "status": "用户未登录或会话已超时"
}
```

状态码：401

- **功能描述**

  接受来自用户的输入信息。如果请求者用户名不在已登陆的用户当中，则返回401。否则将信息提交给DFA，使状态发生转移并获取相应的输出。`running`字段用于判断DFA当前是否结束运行，`reply`字段放置DFA的输出内容，`bye`在DFA运行时始终为0。如果DFA结束运行，`bye`字段将会表示`reply`字段中第几个字符串开始是由DFA退出操作产生的，便于客户端显示退出信息。

3.5 客户端

3.5.1 网页客户端

网页客户端基于PyWebIO框架进行开发，用户可以在服务器开启PyWebIO服务的情况下通过域名直接访问，或者在本地部署一个PyWebIO实例通过浏览器访问。

网页客户端的核心是以`chat_page()`为主的一系列异步函数，主导了用户和客服机器人的对话过程。其中定义了一个滚动的显示区域用于显示对话信息，以及一个输入区域用于接受用户的输入。此外，消息通过另一个异步函数`refresh_msg()`异步刷新，从而避免了网络问题导致的页面卡顿。

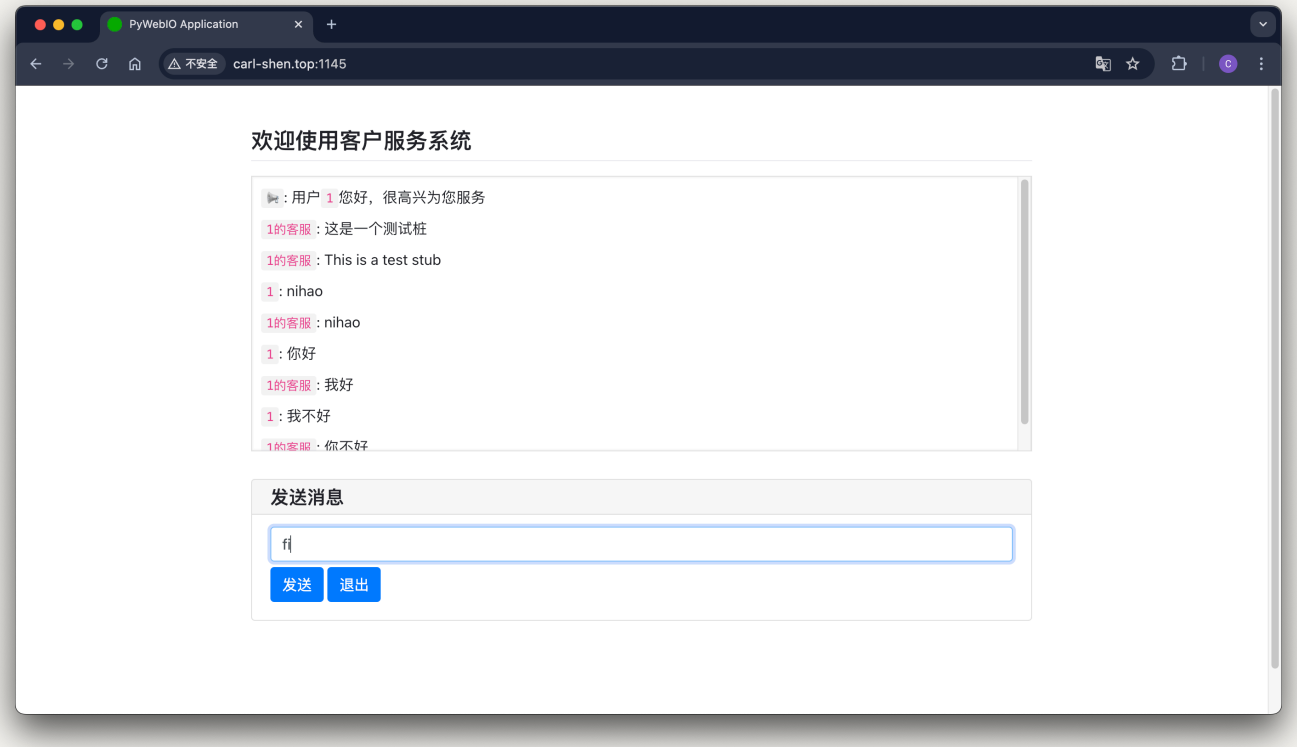
3.5.2 本地客户端

本地客户端实际上也是浏览器，是基于PyQt框架中的`PyWebEngine`组件实现的对PyWebIO应用的封装。在本地客户端中，PyWebIO应用作为一个线程被启动，监听本地8080端口，然后调用PyQt框架显示出一个`QWebEngineView`组件，显示http://localhost:8080的内容。

4. 测试

4.1 测试桩

在进行客户端开发时，服务端功能尚不完善，所以我编写了一个测试桩用于模拟服务端的功能。测试桩同样基于Flask框架，API与正常的服务端相同。但是`/login`，`/register`不具备处理逻辑，永远返回`success`状态。而`/chat`则会将客户端输入稍作修改直接返回如下图所示。



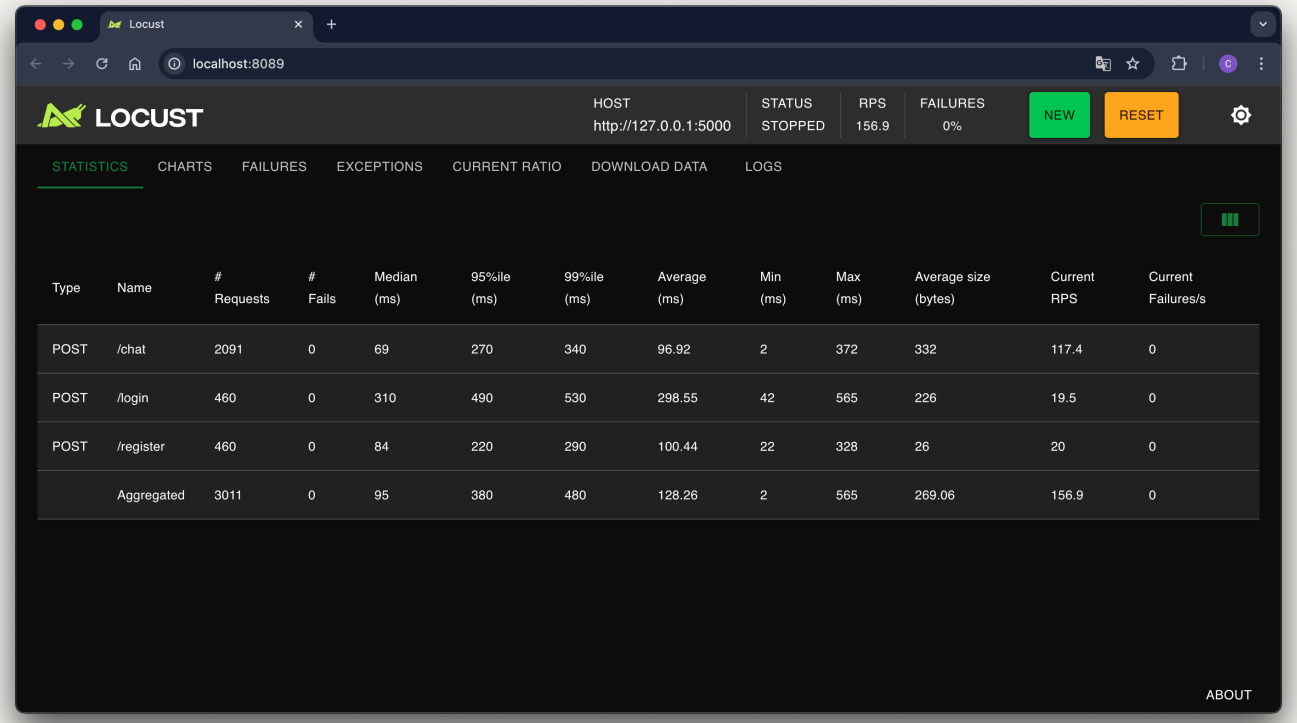
4.2 单元测试

我使用了Python提供的unittest库，针对DSL分析器，用户登陆，用户注册，用户对话都编写了相应的单元测试。对于我在scripts目录下提供的脚本和样例均通过了所有的单元测试。

其中登陆测试，注册测试和对话测试的记录均存放在log目录下。

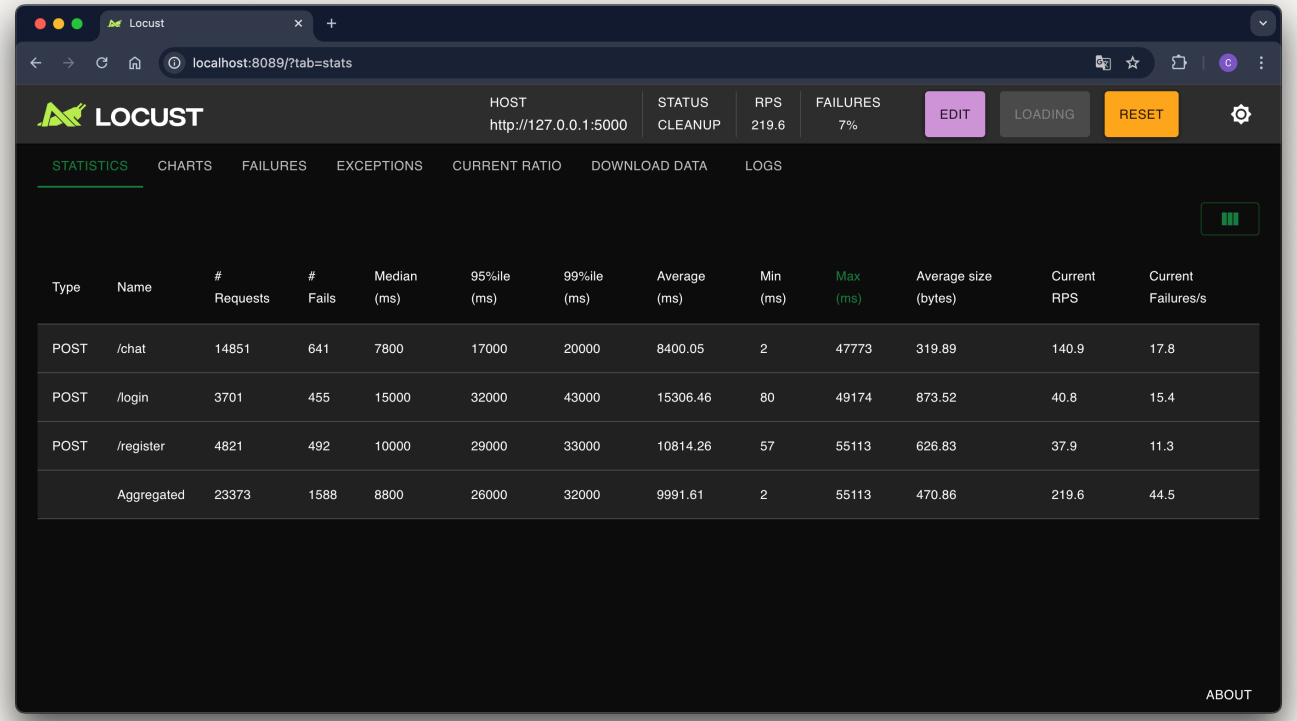
4.3 压力测试

使用了locust库来模拟了大量用户并发访问服务器的情景。当用户数量为2000，RPS约为160时，服务端能够正常提供响应，请求失败率为0%。



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/chat	2091	0	69	270	340	96.92	2	372	332	117.4	0
POST	/login	460	0	310	490	530	298.55	42	565	226	19.5	0
POST	/register	460	0	84	220	290	100.44	22	328	26	20	0
	Aggregated	3011	0	95	380	480	128.26	2	565	269.06	156.9	0

而当用户数量添加到5000时，RPS超过200，我的服务器不再能完全承载如此大量的访问，约有7%的请求失败。



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/chat	14851	641	7800	17000	20000	8400.05	2	47773	319.89	140.9	17.8
POST	/login	3701	455	15000	32000	43000	15306.46	80	49174	873.52	40.8	15.4
POST	/register	4821	492	10000	29000	33000	10814.26	57	55113	626.83	37.9	11.3
	Aggregated	23373	1588	8800	26000	32000	9991.61	2	55113	470.86	219.6	44.5

Flask作为一个轻量级的Web开发框架，在实际部署时建议使用Gunicorn这类的WSGI服务器来提高稳定性。