程序设计实践实验报告

1 概述

1.1 项目功能

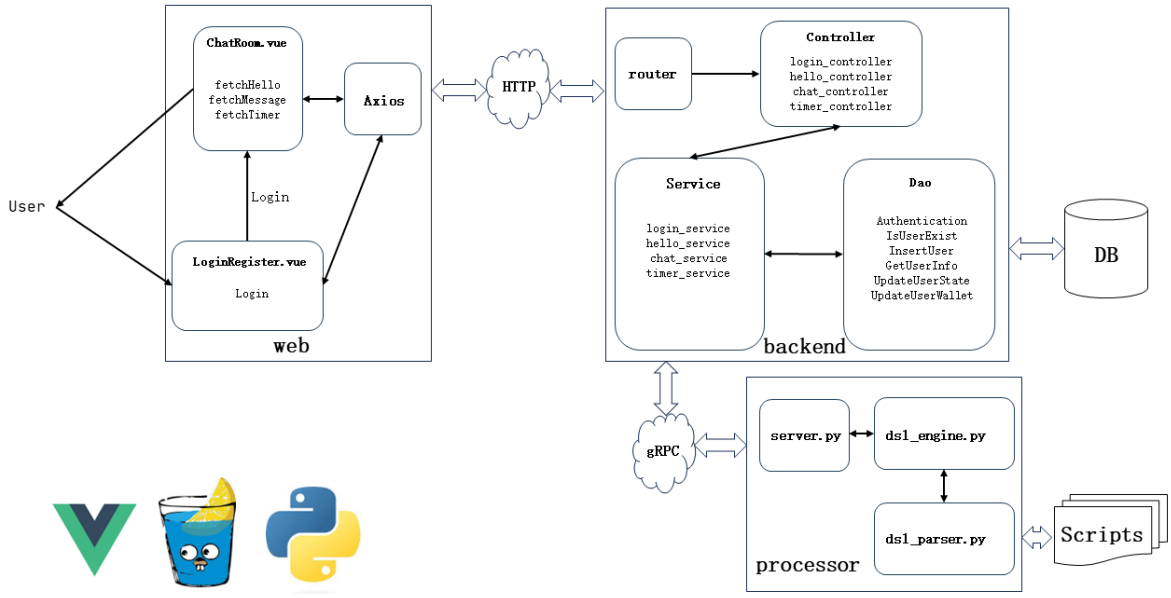
- 支持解析DSL脚本进行不同的自动客服服务
- 支持用户注册登录与数据存储
- 支持图形化界面交互

1.2 项目环境

- Windows 10
- WebStorm 2021.3.2
- GoLand 2022.1
- Pycharm 2023.2.1

2 项目架构

- 前端采用 Vue 框架实现
- 后端采用 Gin 框架实现
- 脚本处理器采用 Python 实现
- 数据库采用 MySQL
- 通信方式有 HTTP 与 gRPC 两种



3 processor 模块介绍——实现客服机器人

客服机器人与用户的交互一般为一问一答或者一问多答，因此客服机器人的底层逻辑被设计为一个拓广的 `Mealy` 状态机，其输入可能是用户信息类，或者用户未执行操作的秒数；其输出是一个用户信息类。

针对用户的输入，机器人可以对各个可行的转移条件进行判断，如果满足某个条件，则执行该分支下的所有动作。

此外，可以自定义一些用户变量，其中 `name`，`balance` 与 `bill` 变量由数据库存储，其他变量默认为静态变量。

3.1 dsl_parser.py——解析并构建DSL语法树

3.1.1 DSL 定义

DSL脚本语言的 BNF 定义如下：

```
<number>              ::= "0" | "1" | ... | "9"
<letter>              ::= "A" | "B" | ... "Z" | "a" | "b" | ... | "z"
<identifier>          ::= <letter>+

<float_type>          ::= {"-" | "+"} {<number>} {"."} <number>+ {("e" | "E") {"-" | "+"} <number>+}
<integer_type>        ::= {"-" | "+"} <number>+
<string_type>         ::= < {character} >

<variable_name>       ::= "$" (<letter> | "_")(<letter> | <number> | "_")*
<variable_type>       ::= <variable> ("Int" <integer_type> | "Float" <real_type> | "Text" <string_type>)
<variable_definition> ::= "Variable" <variable_type>+

<conditions>          ::= <length_condition> | <contains_condition> | <type_condition> | <string_condition>
<length_condition>    ::= "Length" ("<" | ">" | "<=" | ">=" | "==") <integer_type>
<contain_condition>   ::= "Contains" <string_type>
<type_condition>      ::= "Type" ("Int" | "Float")
<string_condition>    ::= <string_type>

<goto_action>         ::= "Goto" <identifier>
<update_float>        ::= ("Add" | "Sub" | "Set") (<float_type> | "Input")
<update_string>       ::= "Set" (<string_type> | "Input")
<update_action>       ::= "Update" <variable_name> (<update_float> | <update_string>)
<speak_content>       ::= <variable_name> | <string_type>
<speak_action>        ::= "Speak" <speak_content> {"+" <speak_content>}
<speak_action_Input>  ::= "Speak" (<speak_content> | "Input") {"+" (<speak_content> | "Input")}
<exit_action>         ::= "Exit"

<case_clause>         ::= "Case" <conditions> {<update_action> | <speak_action_input>} [<exit_action> <goto_action>]
<default_clause>      ::= "Default" {<update_action> | <speak_action_input>} [<exit_action> <goto_action>]
<timer_clause>        ::= "Timer" <integer_type> {<update_action> | <speak_action>} [<exit_action> <goto_action>]
<state_definition>    ::= "State" <identifier> ["Verified"] {<speak_action>} {<case_clause>} <default_clause> {<timer_clause>}
<language>            ::= {<state_definition> | <variable_definition>}
```

3.1.2 语法规则说明与示例

变量定义

变量定义由至少一个变量子句构成。变量子句为变量名、变量类型和默认值。变量名均为 `$` 开头的、由大小写字母和数字组成的字符串。变量类型为 `Int`、`Float`、`Text` 之一。默认值必须和变量类型匹配。

示例如下：

```
Variable
    $a Int 0
    $b Float 0
    $name Text "用户"
```

状态定义

状态定义包括标识符以及一个可选的需要登录验证的标识，之后 依次包含数个 `Speak` 动作、数个 `Case` 子句、一个 `Default` 子句、数个 `Timeout` 子句。

示例如下：

```
State Query Verified
    Speak <你好，> + $name + <! 你的余额是> + $balance + <元，你的账单是> + $bill + <元。>
    Speak <还有什么疑问吗？输入“退出”就可以退出服务啦>
    Case Contains <退出>
        Speak <查询服务先给您结束了，祝您生活愉快！>
        Goto Hello
    Default
        Speak <不好意思我没太懂您的意思，可以再说一遍嘛？输入“退出”就可以退出服务啦>
        Goto Hello
    Timer 10
        Speak <亲，请问您还在吗？查询服务会在10s后自动终止>
    Timer 20
        Speak <您已超时，查询服务先给您结束了，祝您生活愉快>
        Goto Hello
```

条件判断

条件判断有长度条件、子串条件、类型条件、字符串相等条件四种，示例如下：

```
Case Length < 10
Case Length > 10
Case Length == 5
Case Length >= 0
Case Length <= 5
Case Contains "hi"
Case Type Int
Case Type Float
Case "no"
```

动作

Speak 动作中包含数个由 + 连接的 Speak 内容，在状态定义中直接包含的、以及在 Timeout 子句中包含的 Speak 动作不能出现 Input 内容，在其他子句中包含的 Speak 动作可以包含 Input，表示以用户输入替换此处。

Update 动作包含三个部分：`<op, name, value>`。操作可以是 Add、Sub、Set 之一，值可以是字符串或者数字常量，或者 Input（代表用户输入），示例如下：

```
Update $balance Add Input
Update $bill Set 0
```

Goto 动作后跟随一个状态名称。Exit 动作没有参数。

3.1.3 解析示例

```
Variable
    $name Text <Ayu>
    $billing Float 1.1
    $balance Float 10.0

State Welcome
    Speak <你好，> + $name
    Case Contains <查询>
        Goto Hello
    Default
        Goto Hello

State Hello Verified
    Speak <test1: > + <Hello, > + $name + <!>
    Case Length >= 50
        Speak <Your question is too long, ask again>
    Default
        Goto Welcome
    Timer 10
        Exit
```

对于这样的DSL脚本，解析之后的语法树如下所示：

```
[
    ['Variable',
        [
            ['$name', 'Text', 'Ayu'],
            ['$billing', 'Float', 1.1],
            ['$balance', 'Float', 10.0]
        ]
    ],
    ['State',
        'Welcome',
        [],
        [['Speak', ['你好，', '$name']]],
        [['Case', 'Contains', '查询', [['Goto', 'Hello']]]],
        ['Default', [['Goto', 'Hello']]],
        []
```

```
    ],
    ['State',
        'Hello',
        ['Verified'],
        [['Speak', ['test1: ', 'Hello, ', '$name', '!']]],
        [['Case', 'Length', '>=', 50, [['Speak', ['Your question is too long, ask again']]]]],
        ['Default', [['Goto', 'Welcome']]],
        [['Timer', 10, [['Exit']]]]
    ]
```

3.2 DSL 引擎模块——构建自动机执行服务

3.2.1 自动机类

DSL 脚本解析成为语法树之后，会由 StateMachine 类构建出一个自动机：

```python
class StateMachine(object):
    """ StateMachine

    :ivar states: 状态列表
    :ivar variable_set: 变量集合
    :ivar speak: 说话动作列表
    :ivar case: 条件列表
    :ivar default: 默认动作列表
    :ivar timer: 超时动作列表
    """
    def __init__(self, files: list[str]) -> None:
        try:
            parse_results = ChatDSL.parse_scripts(files)
        except Exception as e:
            raise e
        self.states: list[str] = []
        self.variable_set: dict[str, Any] = {}
        verified: list[bool] = []
        self.speak: list[list[Action]] = []
        self.case: list[list[CaseClause]] = []
        self.default: list[list[Action]] = []
        self.timer: list[dict[int, list[Action]]] = []
        ...
```

这个自动机类提供三个接口：

- `hello` 接口，负责输出一个状态最前面的欢迎语句
- `condition_transform` 接口，负责检查是否满足状态内的条件，以及满足条件后如何做动作
- `timeout_transform` 接口，负责检查当前状态下是否超时，已经超时之后如何做动作

3.2.1 用户信息类 UserInfo

在自动机中，用户信息是由用户信息类统一提供，后端发来的请求会首先包装成用户信息类再进行处理：

```
class UserInfo(object):
    """用户信息类

    :ivar state: 用户状态
    :ivar name: 用户名
    :ivar input: 用户输入
    :ivar wallet: 用户钱包
    :ivar lock: 互斥锁
    :ivar answer: 机器人回复
    """

    def __init__(self, user_state: int, user_name, user_input, user_wallet: dict[str, Any] = None) -> None:
        if user_wallet is None:
            user_wallet = {}
        self.state = user_state
        self.name = user_name
        self.input = user_input
        self.wallet = user_wallet
        self.answer: list[str] = []
        self.lock = Lock()
```

3.2.2 转移条件

转移条件由一个抽象类 `Condition` 为基础实现：

```
class Condition(metaclass=ABCMeta):

    @abstractmethod
    def check(self, check_str: str) -> bool:
        pass
```

分为四种：

- LengthCondition
- ContainsCondition
- TypeCondition
- EqualCondition

3.2.3 动作

动作由一个抽象类 `Action` 为基础实现：

```
class Action(metaclass=ABCMeta):
    @abstractmethod
    def exec(self, user_info: UserInfo = None, variable_set: dict[str, Any] = None) -> None:
        pass
```

分为四种：

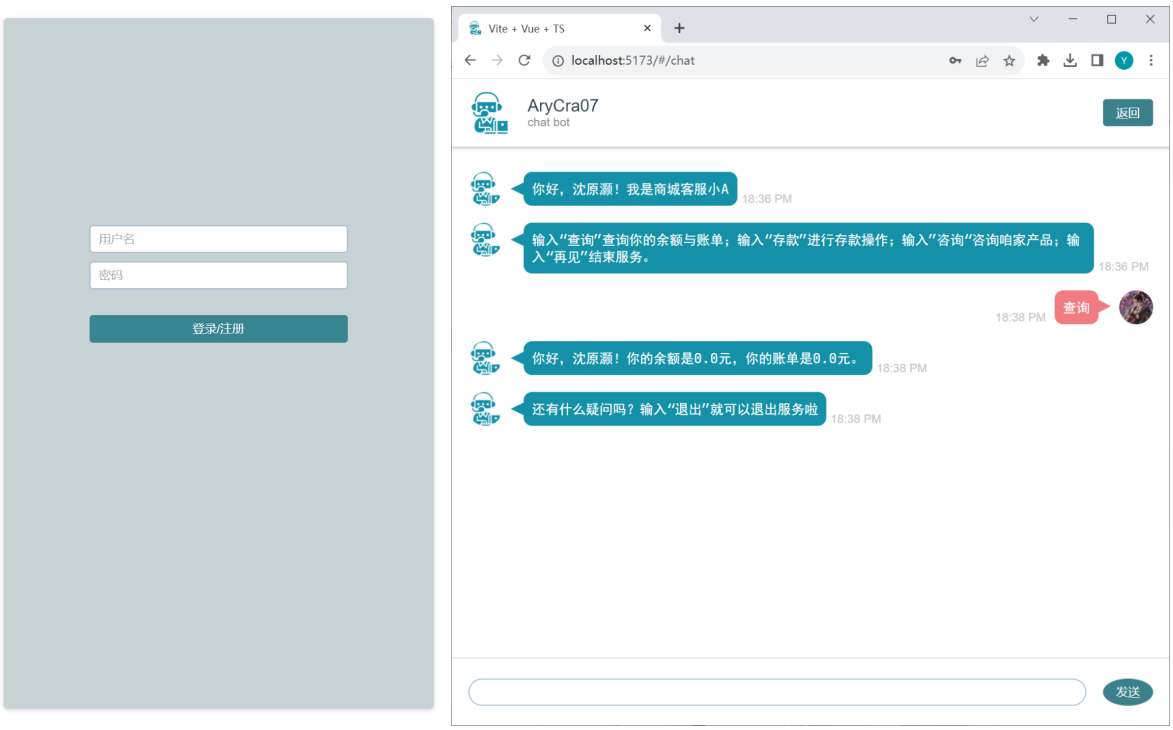
- SpeakAction
- UpdateAction
- GotoAction
- ExitAction

4 后端模块介绍

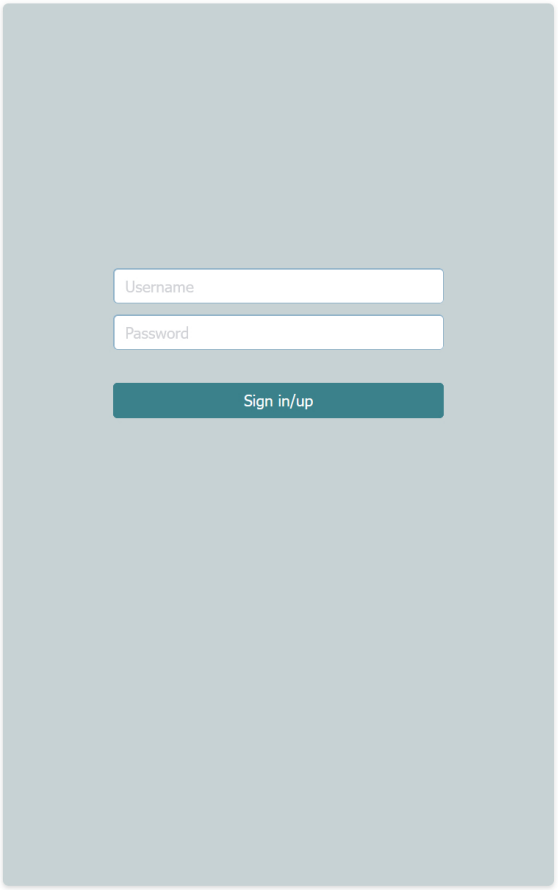
后端采用 Go 语言的 Gin 框架编写，采用 `Controller-Service-Dao` 的三层架构，与 processor 模块实现 gRPC 交互，与前端实现了 HTTP 交互与 jwt 鉴权。

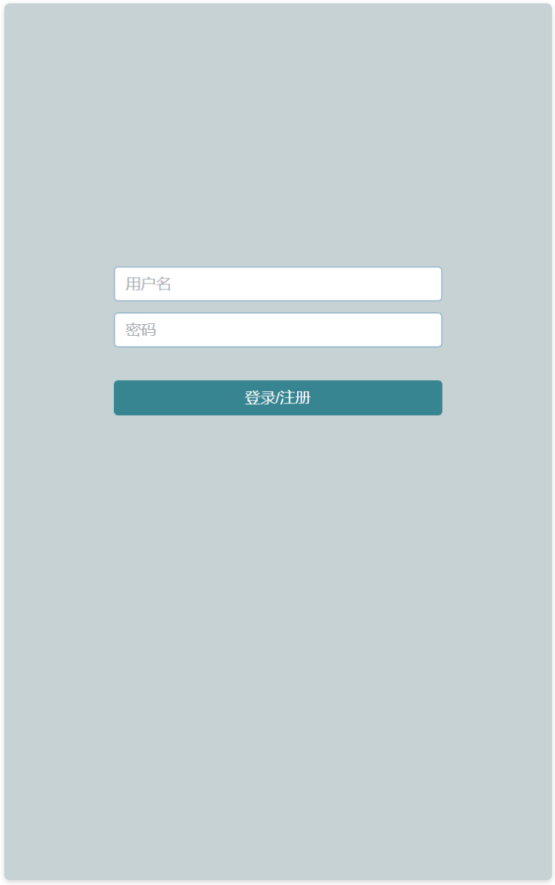
5 前端模块介绍

前端采用 Vue 与 TypeScript 编写，分为登录界面与聊天界面，如下图所示：

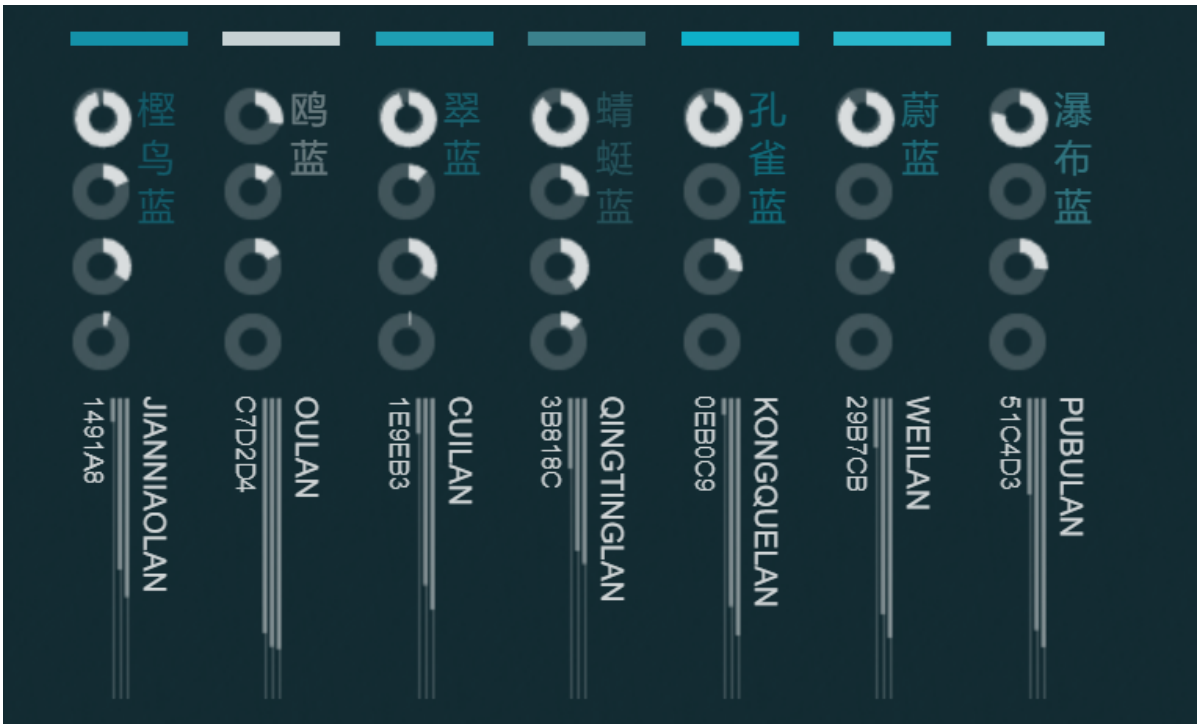


使用 i18n 库实现国际化





网站配色方案



4 接口设计

4.1 前端与后端——HTTP通信

POST /login

login 接口用于用户登录与注册

- **Param** 客户端发送用户名，密码

```
{
    "data": {
        "name": "xxx",
        "password": "xxx"
    },
}
```

- **Return** 返回相应的状态码、信息和一个 token，格式为

```
{
    "code": 0,
    "data": {
        "token": "xxx"
    },
    "msg": "xxx"
}
```

- **Status Code**
  - 200 OK - 鉴权成功，服务器返回响应
  - 400 Bad Request - 客户端请求消息格式有误
- **Custom Code**
  - 0 SUCCESS - 鉴权并登录成功
  - 1 FAIL - 服务失败

POST /hello

hello 接口用于刚进入服务页面时从后端获取初始的欢迎语句

- **Param** 无，请求标头附上 `token`

```
header: {
   "Authorization": token
}
```

- **Return** 返回相应的状态码、信息与字符串列表，格式为

```
{
    "code": 0,
    "data": {
        "content": ["xxx", "xxx"]
    },
    "msg": "xxx"
}
```

- **Status Codes**
  - 200 OK - 鉴权成功，服务器返回响应
  - 400 Bad Request - 客户端请求消息格式有误
- **Custom Code**
  - 0 SUCCESS - 鉴权并登录成功
  - 1 FAIL - 服务失败

POST /message

message 接口用于在用户输入语句之后从后端获取回答

- **Param** 输入语句，token

```
{
    "data": {
        "input": "xxx",
        "token": "xxx"
    }
}
```

- **Return** 返回相应的状态码、信息与字符串列表，格式为

```
{
    "code": 0,
    "data": {
        "content": ["xxx", "xxx"]
    },
    "msg": "xxx"
}
```

- **Status Codes**
  - 200 OK - 鉴权成功，服务器返回响应
  - 400 Bad Request - 客户端请求消息格式有误
- **Custom Code**
  - 0 SUCCESS - 鉴权并登录成功
  - 1 FAIL - 服务失败

POST /timer

timer 接口在用户无响应的时候，每 5 秒向后端发送一次，用于获取超时后的答复和是否应该终止服务。

- **Param** 上次发送时的计时秒数与当前发送时的计时秒数

```
{
    "data": {
        "last_time": 0,
        "now_time": 5
    }
}
```

- **Return** 返回相应的状态码、信息与字符串列表，以及是否应该退出或重置计时器的 `bool` 值。格式为：

```
{
    "code": 0,
    "data": {
        "content": ["xxx", "xxx"],
        "exit": false,
        "reset": false
    },
    "msg": "xxx"
}
```

- **Status Codes**
  - 200 OK - 鉴权成功，服务器返回响应
  - 400 Bad Request - 客户端请求消息格式有误
- **Custom Code**
  - 0 SUCCESS - 鉴权并登录成功
  - 1 FAIL - 服务失败

4.2 后端与处理器——gRPC通信

```
syntax = "proto3";

option go_package = "backend/pb";

package pb;

// 欢迎服务
service Greet {
  rpc SayHelloService (HelloRequest) returns (HelloResponse) {}
}

// 聊天服务
service Chat {
  rpc AnswerService (ChatRequest) returns (ChatResponse) {}
}

```

```
// 计时器服务
service Timer {
  rpc TimerService (TimerRequest) returns (TimerResponse) {}
}

message HelloRequest {
  int32 state = 1;
  string name = 2;
}

message HelloResponse {
  repeated string words = 1; // 欢迎语句
}

message ChatRequest {
  int32 state = 1;
  string name = 2;
  string input = 3; // 用户输入
  map<string, float> wallet = 4; // 用户的存款、账单等信息
}

message ChatResponse {
  int32 state = 1;
  repeated string answer = 2;
  map<string, float> wallet = 3;
}

message TimerRequest {
  int32 state = 1;
  int32 last_time = 2; // 上次请求时间
  int32 now_time = 3; // 当前请求时间
}

message TimerResponse {
  int32 state = 1;
  bool is_exit = 2; // 是否终止服务
  bool reset = 3; // 是否重置计时器
  repeated string answer = 4;
}
```

5 测试

5.1 processor 部分

processor中实现了三个自动测试脚本：

- test_parser.py: 测试 DSL 脚本是否能正常解析，解析结果与测试桩比对；
- test_update.py: 测试 DSL 引擎是否能正常工作，工作结果与测试桩比对；
- test_server.py: 测试 gRPC 服务是否能正常工作，在 stub.py 中实现了一个简易版服务器，将服务响应与测试桩比对。

运行 main_test.py 可以执行以上所有测试

5.2 backend 部分

backend 实现了 jwt 鉴权的自动测试脚本，生成一个测试 token 并检查是否能解析成功