robot-DSL

毛子恒

2021 年 12 月 09 日

Contents

CHAPTER 1

简介

robot-DSL 定义了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的自动应答逻辑，并设计实现了一个解释器，可以解释执行这个脚本。该解释器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。

robot-DSL 将输入和应答逻辑封装为 Restful API 以供调用，并且实现了一个有美观 GUI 的客户端。

CHAPTER 2

模块介绍

2.1 客服机器人逻辑

2.1.1 概述

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，其输入可能是用户字符串，或者用户未执行操作的秒数；其输出通常是一个字符串序列。

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，用于简单存储相关的信息，用户变量持久保存在数据库中。

2.1.2 用户变量

在客服机器人中，可以自定义整型、浮点数型和字符串型的用户变量，用户变量名是以 `$` 开头的字母/数字串。顾名思义，用户变量值是每个用户独有的，但是每个用户都拥有相同种类和数量的用户变量。每个用户变量有一个默认值，在注册新用户时自动赋予，在脚本中可以使用 `Update` 动作触发用户变量的修改，也可以使用 `Speak` 动作向用户输出用户变量的值。

开始一个新会话时，用户默认分配到一个访客账户，用户之后可以注册或者登录自己的账户，由于访客账户是所有用户共有的，所以不能更改属于访客账户的变量。更多信息请参考*Update 动作*。

用户变量保存在 SQLite 数据库中，通过 Storm 库进行 ORM 访问。在分析脚本语言的过程中，会根据脚本中对于用户变量的定义建立数据库，每个用户关联到数据库中的一行，每个属性为数据库中的一列。

Storm 库不是线程安全的，因此每次数据库访问都需要互斥锁。

2.1.3 转移逻辑

如前所述，对于输入是用户字符串的情况，状态机中的每个状态会保存一个转移条件 列表，状态机依次检查列表中的每一个条件，如果用户的输入满足一个条件，则执行该条件下的所有动作，并且忽略之后的所有条件。每个状态必须有一个 `Default` 转移，表示当条件列表中的条件都不满足时，执行默认的动作。简而言之，条件的检查类似于 `if-elif-else` 逻辑。

对于输入是用户未执行操作的秒数，状态机中的每个状态会保存一个超时转移 字典，客户端应当每隔一段时间返回用户未操作的秒数，状态机检查字典中是否包含当前时间间隔中的时刻，如果包含，就执行相应的动作。

2.1.4 转移条件

对于用户输入的字符串，可以执行以下几种条件判断：

- 判断其输入长度是否满足限制，参考*server.state_machine.LengthCondition*。
- 判断其输入是否包含某个字符串，参考*server.state_machine.ContainCondition*。
- 判断其输入字面值是否是某种类型，参考*server.state_machine.TypeCondition*。
- 判断其输入是否和某个串相同，参考*server.state_machine.EqualCondition*。

2.1.5 动作

Update 动作

更新一个用户变量，更新的操作有以下三种：

- `Add`，适用于整型或浮点型的变量。
- `Sub`，适用于整型或浮点型的变量。
- `Set`，适用于任何变量。

更新的值有两种：

- 字符串或数字字面值。
- Copy，表示用户的输入。

编译器会检查字面值和用户变量的类型是否匹配，由于无法直接检查用户的输入，因此当更新的值为 Copy 时，Update 动作需要包括在判断用户输入类型的子句中，保证用户输入和用户变量的类型兼容。

为了保证不能更新访客用户的属性，或者避免用户访问到访客用户的某些变量，在定义状态时可选指定 `Verified` ，标识该状态需要登录才可以进入，访客用户进入该状态的请求会被拒绝。Update 子句只能定义在指定了 `Verified` 的状态中。

参考*server.state_machine.UpdateAction*。

Speak 动作

向用户返回一个字符串，字符串可以由多个部分组成，各个部分之间用 + 连接。每个部分可以是以下三种之一：

- 由半角双引号包括的字符串常量。
- 变量名。
- Copy，表示用户的输入。

参考*server.state_machine.SpeakAction*。

Exit 动作

结束一个会话，简单地将用户的状态设为-1 表示会话结束。

参考*server.state_machine.ExitAction*。

Goto 动作

转移到另一个状态。

参考*server.state_machine.GotoAction*。

2.1.6 状态机

server.state_machine.StateMachine 是对上述状态机的实现。

构建状态机时，首先调用语法分析模块，在返回的分析树的基础上进行语义分析，并且构建模型。

状态机提供条件转移和超时转移两个接口，可以根据给定的用户状态和用户输入进行状态转移，并且返回需要输出给用户的字符串列表。

2.1.7 API

class server.state_machine.**UserVariableSet**(*username: str*, *passwd: str*)
用户变量集，与数据库关联。

采用 storm 作为 ORM，除了 `column_type` 之外各个类属性关联到数据库表中的一列，实例中的相关属性关联到元组的对应属性。

用户的基础属性包括用户名和密码，其他属性则通过 `setattr` 动态添加。

变量 column_type –表中各列的类型。

class server.state_machine.**Condition**
条件判断抽象基类。

abstract check(*check_string: str*) → bool
判断是否满足条件。

参数 check_string –需要判断的字符串。

返回 如果满足条件，返回 True；否则返回 False。

class server.state_machine.**LengthCondition**(*op: str*, *length: int*)
长度判断条件，判断用户输入是否满足长度限制。

变量

- **op** –判断运算符，可以为 <、>、<=、>=、= 其中之一。
- **length** –长度。

check(*check_string: str*) → bool
参考：*Condition.check()*

class server.state_machine.**ContainCondition**(*string: str*)
字符串包含判断条件，判断用户输入是否包含给定串。

变量 string –包含的字符串。

check(*check_string: str*) → bool
参考：*Condition.check()*

class server.state_machine.**TypeCondition**(*type_: str*)
字符串字面值类型判断，判断用户输入是否是某种类型。

变量 type –类型，可以为 Int、Real 之一。

check(*check_string: str*) → bool
参考：*Condition.check()*

class server.state_machine.**EqualCondition**(*string: str*)
字符串相等判断，判断用户输入是否和某一个串相等。

变量 string –字符串。

check(*check_string: str*) → bool
参考：*Condition.check()*

class server.state_machine.**Action**
动作抽象基类。

abstract exec(*user_state: server.state_machine.UserState*, *response: list[str]*, *request: Any*) → None
执行一个动作。

参数

- **user_state** –用户状态。
- **response** –产生回复字符串列表。
- **request** –用户请求字符串。

class server.state_machine.**ExitAction**
退出动作，结束一个会话。

exec(*user_state: server.state_machine.UserState*, *response: list[str]*, *request: str*) → None
参考：*Action.exec()*

class server.state_machine.**GotoAction**(*next_state: int*, *verified: bool*)
状态转移动作，转移到一个新状态。

变量

- **next** –转移到的状态。
- **verified** –新状态是否需要登录验证。

exec(*user_state: server.state_machine.UserState*, *response: list[str]*, *request: str*) → None
参考：*Action.exec()*

class server.state_machine.**UpdateAction**(*variable: str*, *op: str*, *value: Union[str, int, float]*, *value_check: Optional[str]*)

更新用户变量动作。

变量

- **variable** –变量名。
- **op** –更新操作类型，可以是 Add、Sub、Set 之一。
- **value** –更新的值，可以是以双引号开头和结尾的字符串、”Copy”或者一个数字。
- **value_check** –该动作是否处于什么样的类型检查环境，可以是 Int、Real、Text 或者 None。

exec(*user_state: server.state_machine.UserState*, *response: list[str]*, *request: str*) → None

参考：*Action.exec()*

class server.state_machine.**SpeakAction**(*contents: list[str]*)

产生回复动作。

变量 contents –回复内容列表。

exec(*user_state: server.state_machine.UserState*, *response: list[str]*, *request: str*) → None

参考：*Action.exec()*

class server.state_machine.**CaseClause**(*condition:* server.state_machine.Condition)

条件分支。

变量

- **condition** –条件。
- **action** –满足条件后执行的动作列表。

class server.state_machine.**StateMachine**(*files: list[str]*)

状态机。

变量

- **states** –状态集合。
- **speak** –状态默认的 speak 语句集合。
- **case** –状态的条件分支集合。
- **default** –状态的默认分支。
- **timeout** –状态的超时转移分支。

_action_constructor(*language_list: list*, *target_list: list[*server.state_machine.Action*]*, *index: int*, *verified: list[bool]*, *value_check: Optional[str]*) → None

构建一个动作列表。

参数

- **language_list** –语法树的子树，包含一系列动作。
- **target_list** –构建的动作列表存储到此。
- **index** –状态编号。
- **verified** –状态是否需要登录验证。
- **value_check** –参考:py:class:*UpdateAction*。

condition_transform(*user_state: server.state_machine.UserState*, *msg: str*) → list[str]

条件转移。

参数

- **user_state** –用户状态
- **msg** –用户输入。

返回 输出的字符串列表。

hello(*user_state: server.state_machine.UserState*) → list[str]
输出某个状态的默认 speak 动作。

参数 **user_state** –用户状态。

返回 输出的字符串列表。

timeout_transform(*user_state: server.state_machine.UserState, now_seconds: int) -> (list[str], <class 'bool'>, <class 'bool'>*)
超时转移。

参数

- **user_state** –用户状态。
- **now_seconds** –用户未执行操作的秒数。

返回 输出的字符串列表、是否需要结束会话、是否转移到新的状态。

2.1.8 异常

class server.state_machine.**LoginError**
表示用户未登录的错误。

class server.state_machine.**GrammarError**(*msg: str*, *context: list[str]*)
表示脚本语言的语法错误。

变量

- **msg** –错误消息。
- **context** –错误上下文。

2.2 脚本语言语法

2.2.1 脚本语言语法的定义

该脚本语言语法的 BNF 定义如下：

```
<language>            ::= {<state_definition> | <variable_definition>}
<state_definition>    ::= "State" <identifier> ["Verified"] {<speak_action>} {<case_
↪clause>} <default_clause> {<timeout_clause>}
<identifier>          ::= <letter>+
<letter>              ::= "A" | "B" | ... "Z" | "a" | "b" | ... | "z"
<speak_action>        ::= "Speak" <speak_content> {"+" <speak_content>}
<speak_content>       ::= <variable> | <string_constant>
<variable>            ::= "$" (<letter> | <number> | "_")+
<number>              ::= "0" | "1" | ... | "9"
<string_constant>     ::= double_quote {character} double_quote
<case_clause>         ::= "Case" <conditions> {<update_action> | <speak_action_copy>}␣
↪[<exit_action> <goto_action>]
```

(下页继续)

(续上页)

```
<conditions>          ::= <length_condition> | <contain_condition> | <type_condition>␣
↪| <equal_condition>
<length_condition>    ::= "Length" ("<" | ">" | "<=" | ">=" | "=") <integer_constant>
<integer_constant>    ::= {"-" | "+"} <number>+
<contain_condition>   ::= "Contain" <string_constant>
<type_condition>      ::= "Type" ("Int" | "Real")
<equal_condition>     ::= <string_constant>
<update_action>       ::= "Update" <variable> (<update_real> | <update_string>)
<update_real>         ::= ("Add" | "Sub" | "Set") (<real_constant> | "Copy")
<update_string>       ::= "Set" (<string_constant> | "Copy")
<real_constant>       ::= {"-" | "+"} {<number>} {"."} <number>+ {("e" | "E") {"-" |
↪"+"} <number>+}
<speak_action_copy>   ::= "Speak" (<speak_content> | "Copy") {"+" (<speak_content> |
↪"Copy")}
<exit_action>         ::= "Exit"
<goto_action>         ::= "Goto" <identifier>
<default_clause>      ::= "Default" {<update_action> | <speak_action_copy>} [<exit_
↪action> <goto_action>]
<timeout_clause>      ::= "Timeout" <integer_constant> {<update_action> | <speak_
↪action>} [<exit_action> <goto_action>]
<variable_definition> ::= "Variable" <variable_clause>+
<variable_clause>     ::= <variable> ("Int" <integer_constant> | "Real" <real_
↪constant> | "Text" <string_constant>)
```

注：除了 <variable>、<string_constant>、<real_constant>、<integer_constant> 以外，所有产生式右部的各个属性之间应该以至少一个空白字符分隔，为了增加可读性，定义中略去。

2.2.2 语法规则说明及示例

本节对脚本语言的语法规则进行简单说明，所有的语法细节无法完全顾及，可以参考上一节的 BNF 定义。

该脚本语言有数个状态定义和变量定义构成。

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、Real、Text 之一。默认值必须和变量类型匹配。

一个变量定义的示例如下：

```
Variable
    $trans Int 0
    $billing Real 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。

一个状态定义的示例如下：

```
State Rename Verified
    Speak "请输入您的新名字，不超过30个字符"
    Case "返回"
        Goto Welcome
    Case Length <= 30
        Speak "您的新名字为" + Copy
        Update $name Set Copy
        Goto Welcome
    Default
        Speak "输入过长"
    Timeout 60
        Speak "您已经很久没有操作了，即将返回主菜单"
        Goto Welcome
```

Case 子句包含一个条件判断，之后跟随数个 Update 动作或 Speak 动作，再跟随一个可选的 Goto 动作或者 Exit 动作。

Default 子句包含数个动作，与 Case 子句类似。

Timeout 子句包含一个整数，之后跟随的动作和 Case 子句类似。

条件判断

条件判断有长度条件、子串条件、类型条件、串相等条件四种，几种条件判断语句的示例如下：

```
Case Length < 20
Case Length > 10
Case Length = 5
Case Length >= 0
Case Length <= 200
Case Contain "hi"
Case Type Int
Case Type Real
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 `Copy` 内容，在其他子句中包含的 Speak 动作可以包含 `Copy`，表示以用户输入的字符串替换此处。

Update 动作包含三个部分：操作、变量名、值。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 `Copy`，几个示例如下：

```
Update $billing Sub -1
Update $name Set Copy
```

Goto 动作后跟随一个状态名称。Exit 动作没有参数。

更多语法规则

即使严格遵循上述语法规则也不能保证写出一个正确的脚本，在语法的定义中无法包含变量名和状态名冲突、Update 的变量和值匹配等问题，因此仍然需要参考*动作* 章节的说明编写脚本。

2.2.3 语法树的形态

server.parser.RobotLanguage 类内置了上述语法规则，可以实现从文件中读入脚本并且解析，解析出的语法树以嵌套列表的形式返回。因此仅有叶节点上存储语法元素，非叶节点均为空。假设根节点在第 0 层。

例如，对于以下脚本：

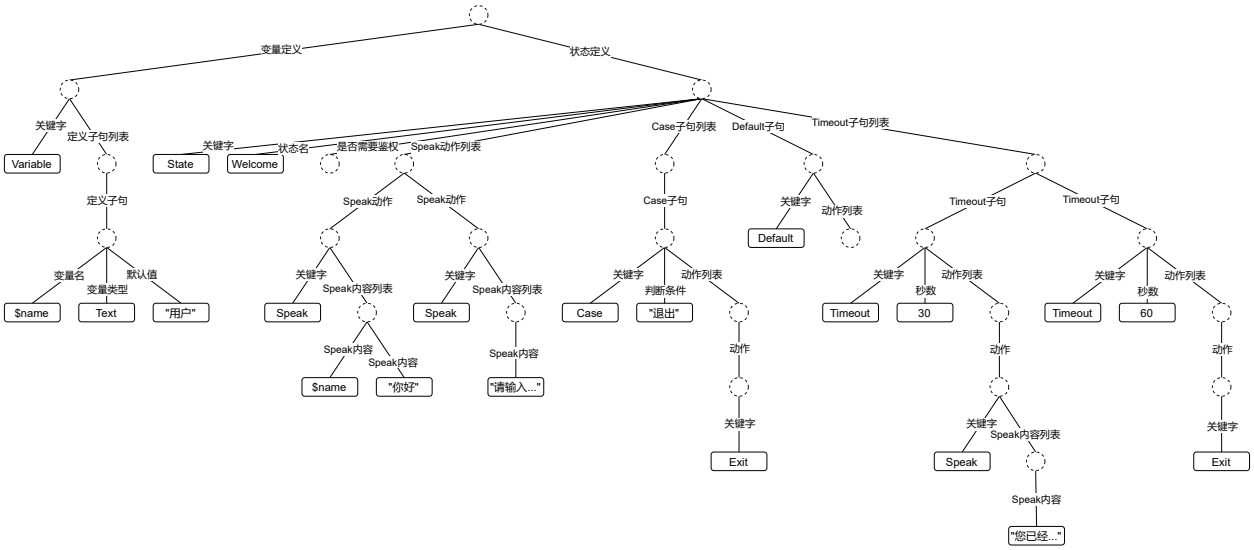
```
Variable
    $name Text "用户"

State Welcome
    Speak $name + "你好"
    Speak "请输入 退出 以退出"
    Case "退出"
        Exit
    Default
    Timeout 30
        Speak "您已经很久没有操作了，即将于30秒后退出"
    Timeout 60
        Exit
```

解析得到的文本如下：

```
[['Variable', [['$name', 'Text', '"用户"']]], ['State', 'Welcome', [], [['Speak', ['
↪$name', '"你好"']], ['Speak', ['"请输入 退出 以退出"']]], [['Case', '"退出"', [[
↪'Exit']]]], ['Default', []], [['Timeout', 30, [['Speak', ['
↪"您已经很久没有操作了，即将于30秒后退出"']]]], ['Timeout', 60, [['Exit']]]]]]
```

生成的语法树形态如下图所示：



注：此图中各边上的文字仅用作说明子树的含义，在实际语法树中不出现。

2.2.4 API

class server.parser.**RobotLanguage**
脚本语言对象。

定义脚本语言的文法，以及对一个文件列表进行分析的方法。

static parse_files(*files: list[str]*) → list[pyparsing.results.ParseResults]
解析一个脚本，脚本存储在一系列文件中。

参数 **files** –文件名列表。

返回 解析脚本得到的语法树。

2.3 用户管理

2.3.1 概述

系统采用用户名唯一标识每个用户，对于访客用户，连接时会给用户名后连接一个串以确保不同客户端的用户名不重复。

系统采用字典将用户名映射到每个用户对象，每个用户对象对应一个正在或者曾经连接到服务器的客户端，其中包含有用户状态和超时计时器等信息。

用户采用 JWT 鉴权，首次连接时用户会获取到唯一的 JWT 令牌，该 JWT 令牌永久有效，但是当用户登录或者注册成功时，会获取到新的令牌，原有的令牌立即作废。

由于需要记录用户闲置的时间，所以客户端需要定期向服务器发送 echo 消息，其中包含用户闲置的时间。对于每个用户有一个计时器，计时器超时会在用户一段时间内 *没有任何请求时触发*，注意此处的概念与用户 *一段时间内闲置*不同。

登录、注册导致用户名变化、用户到达结束状态，或者触发超时后会释放用户对象。

2.3.2 API

class server.user_manage.**User**(*username: str*)
用户类。

变量

- **timer** –计时器，当用户很久没有发送请求时，认为用户已经离线，调用超时处理函数，释放用户对象。
- **state** –用户状态。
- **username** –用户名。

class server.user_manage.**UserManage**(*key: str*)
用户管理类。

变量

- **users** –从用户名映射到*server.user_manage.User* 对象的字典。
- **lock** –互斥访问 *users* 字典的锁。
- **key** –JWT 加密密钥。

connect() -> (<class 'server.user_manage.User'>, <class 'str'>)
处理新客户端连接到服务器的请求。

返回 *User* 对象和 JWT 令牌。

jwt_decode(*token: str*) → *server.user_manage.User*
JWT 令牌解码。

参数 **token** –JWT 令牌。

返回 如果解码成功，并且用户存在，则返回对应的 *User* 对象。

引发 **jwt.InvalidTokenError** –当解码失败或者用户名不存在时触发。

jwt_encode(*username: str*) → str
JWT 令牌编码。

参数 **username** –用户名。

返回 JWT 令牌。

login(*user:* server.user_manage.User, *username: str*, *passwd: str*) → Optional[str]
处理登录请求。

参数

- **user** –客户端对应的 *User* 对象。
- **username** –登录的用户名。
- **passwd** –登录的密码。

返回 如果注册成功，返回新 JWT 令牌。否则返回 None。

register(*user:* server.user_manage.User, *username: str*, *passwd: str*) → Optional[str]
处理注册请求。

参数

- **user** –客户端对应的 *User* 对象。
- **username** –注册的用户名。
- **passwd** –注册的密码。

返回 如果注册成功，返回新 JWT 令牌。否则返回 None。

timeout_handler(*username: str*) → None
超时处理函数。

参数 **username** –超时的用户名。

2.4 Restful API

服务端采用 Flask 封装了 Restful API。

GET /register
客户端请求注册，服务器返回新的 token。

Param 客户端发送用户名、密码和 token，格式为：`{"username": "xxx", "passwd": xxx, "token": "xxx"}`。

Return 返回一个新的 token，格式为：`{"token": "xxx"}`。

Status Codes

- 200 OK –鉴权并注册成功。
- 400 Bad Request –客户端请求消息格式有误。
- 403 Forbidden –鉴权失败。

一个客户端通过此路由向服务器发送一个注册请求。

收到请求后，客户端首先对 token 进行鉴权，之后验证用户名是否合法，如果验证通过，则返回一个新的 token。原有的 token 立即过期，客户端需要使用新的 token 继续会话。

GET /login
客户端请求登录，服务器返回新的 token。

Param 客户端发送用户名、密码和 token，格式为：`{"username": "xxx", "passwd": xxx, "token": "xxx"}`。

Return 返回一个新的 token，格式为：`{"token": "xxx"}`。

Status Codes

- 200 OK –鉴权并登录成功。
- 400 Bad Request –客户端请求消息格式有误。
- 403 Forbidden –鉴权失败。

一个客户端通过此路由向服务器发送一个登录请求。

收到请求后，客户端首先对 token 进行鉴权，之后验证用户名和密码，如果验证通过，则返回一个新的 token。原有的 token 立即过期，客户端需要使用新的 token 继续会话。

GET /send
客户端发送一条新消息，服务器返回响应。

Param 客户端发送一条消息和 token，格式为：`{"msg": "xxx", "token": "xxx"}`。

Return 返回一个消息列表和是否结束会话的标志，格式为：`{"msg": ["xxx", "xxx"], "exit": false}`。

Status Codes

- 200 OK –鉴权成功，服务器产生响应。
- 400 Bad Request –客户端请求消息格式有误。
- 403 Forbidden –鉴权失败。
- 401 Unauthorized –用户是访客，需要登录。

一个客户端通过此路由向服务器发送一条消息。

收到消息后，客户端首先对 token 进行鉴权，之后对消息进行处理并产生响应，返回一个消息列表。如果服务器需要终止一个会话，则设 `exit` 为 1，该 token 立即过期，客户端需要重新开启一个会话。

GET /echo
客户端发送一条 echo，服务器返回响应。

Param 客户端发送闲置时间和 token，格式为：`{"seconds": 5, "token": "xxx"}`。

Return 返回一个消息列表、是否结束会话的标志和是否要求用户重置计时器的标志，格式为：`{"msg": ["xxx", "xxx"], "exit": false, reset: false}`。

Status Codes

- 200 OK –鉴权成功，服务器产生响应。
- 400 Bad Request –客户端请求消息格式有误。

- 403 Forbidden –鉴权失败。
- 401 Unauthorized –用户是访客，需要登录。

一个客户端通过此路由向服务器发送一条 echo，表明自己仍然存活和用户闲置的时间。

收到 echo 后，客户端首先对 token 进行鉴权，之后依照闲置时间进行处理并产生响应，返回一个消息列表。如果服务器需要终止一个会话，则设 `exit` 为 1，该 token 立即过期，客户端需要重新开启一个会话。如果服务器要求客户端重置闲置时间计时器，则设 `reset` 为 1，客户端应当重启计时器。

GET /

一个新的客户端连接到服务器时，请求一个 token。

Return 返回一个消息列表和 token，格式为：`{"msg": ["xxx", "xxx"], "token": "xxx"}`。

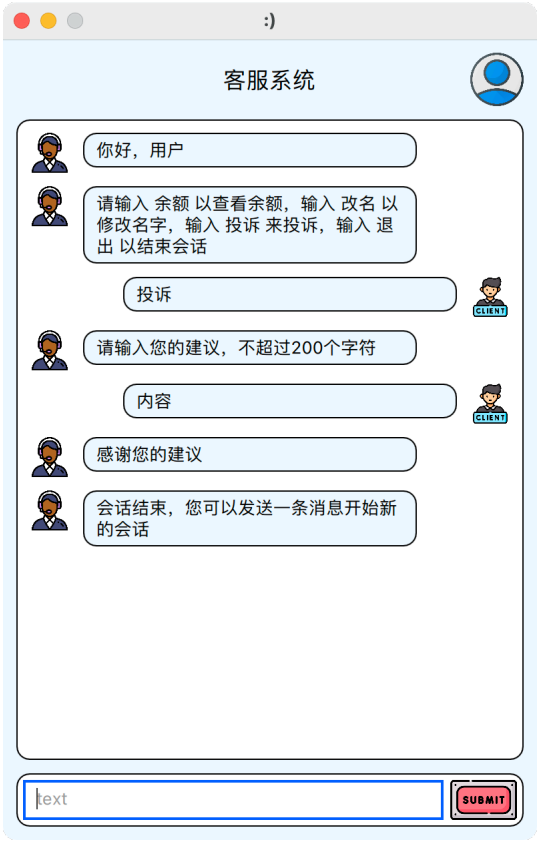
Status Codes

- 200 OK –成功建立会话。

一个客户端与服务器建立连接时，或者客户端开始一个新的会话时，从此路由获取一个 token。服务器默认分配一个访客账户，如果设置了默认的问候消息，还会返回消息列表。

2.5 客户端

客户端采用 PyQt5 + QtQuick 编写，有消息页面和登录页面，如下图所示。




:)
登录页
用户名
密码
LOG IN
SIGN UP

CHAPTER 3

用户指南

3.1 用户指南

安装依赖：

```
pip install -r requirement.txt
```

启动服务端：

```
python -m flask run
```

启动客户端：

```
cd client
python main.py
```

CHAPTER 4

测试

4.1 单元测试

4.2 测试脚本

对于后端的每个模块都有对应的单元测试，例如，运行 parser 模块的单元测试：

```
python -m test.test_parser
```

会得到如下结果：

```
.
----------------------------------------------------------------------
Ran 1 test in 0.014s

OK
```

可用的单元测试模块如下：

```
test.test_app
test.test_parser
test.test_speak_action
test.test_update_action
test.test_state_machine
test.test_user_state
```

4.3 测试桩

为了对客户端进行测试，stub.py 实现了一个具有简单功能的后端，仅有两个状态，支持登录和注册操作。

运行测试桩的命令如下：

```
python test/stub.py
```

4.4 压力测试

为了测试线程安全和多客户端访问服务器时服务器的承受能力，我设计了压力测试，并行开启 100 个客户端对数据库进行多次访问，运行压力测试的命令如下：

```
python -m test.test_pressure
```

CHAPTER 5

索引表

- genindex
- search

HTTP Routing Table

/

/echo

/login

/register

/send

索引

非字母

A

C

E

G

H

J

L

P

R

S

T

U