

ReD-DSL开发文档

严怡彬

2022 年 11月 20日

ReD-DSL开发文档

- 1 简介
- 2 程序风格
 - 2.1 模块命名风格
 - 2.2 注释风格
 - 2.2.1 模块注释风格
 - 2.2.2 函数注释风格
 - 2.2.3 类注释风格
 - 2.2.4 块注释和行注释
- 3 模块介绍
 - 3.1 脚本语言语法
 - 3.1.1 脚本语言语法的BNF定义
 - 3.1.2 脚本语言语法规则说明示例
 - 变量定义
 - 状态定义
 - switch语句定义
 - timeout语句定义
 - 动作语句
 - 其他语法规则
 - 3.1.3 抽象语法树
 - 3.1.4 模块接口(API)
 - 3.2 解释器主模块
 - 3.2.1 概述
 - 3.2.2 用户变量(UserVariableSet)
 - 3.2.3 状态机(StateMachine)
 - 3.2.4 状态转移表(StateMachine.action_dict)
 - 3.2.5 控制器(Controller)
 - 3.2.6 模块接口(API)
 - 3.3 RESTful API
 - GET /send
 - GET /register
 - GET /login
 - 3.4 客户端
 - 3.4.1 用户注册与登录
 - 3.4.2 机器人客服功能
 - 3.4.3 自然语言大模型 (Large Language Model)
- 4 测试
 - 4.1 自动测试脚本
 - 4.2 测试桩
 - 4.2.1 状态机(StateMachine)测试桩
 - 4.2.2 控制器(Controller)测试桩
 - 4.2.3 前端测试桩
 - 4.2.4 后端测试桩
 - 4.3 API测试
 - 4.3.1 API测试设置

- 4.3.2 API测试结果
- 4.4 压力测试
- 5 开发日志与版本管理
- 6 附录：文件结构
- 7 附录：接口（API）查询表
 - 7.1 词法分析器(Lexer) API
 - 7.1.1 类定义与初始化API
 - 7.1.2 获取扫描的下一个记号API
 - 7.1.3 从地址读取脚本API
 - 7.1.4 从字符串读取脚本API
 - 7.1.5 返回词法分析器API
 - 7.2 语法分析器(Parser) API
 - 7.2.1 类定义与初始化API
 - 7.2.2 对脚本进行分析API
 - 7.3 抽象语法树(ASTNode) API
 - 7.3.1 类定义与初始化API
 - 7.3.2 打印语法树API
 - 7.4 回调函数(CallBack) API
 - 7.4.1 类定义与初始化API
 - 7.4.2 执行回调函数API
 - 7.5 用户变量集(UserVariableSet) API
 - 7.5.1 类定义与初始化API
 - 7.6 状态机(StateMachine) API
 - 7.6.1 类定义与初始化API
 - 7.6.2 解释脚本API
 - 7.6.3 判断测试条件API
 - 7.6.4 更新用户返回值API
 - 7.6.5 用户注册API
 - 7.6.6 用户登录API
 - 7.7 控制器(Controller) API
 - 7.7.1 类定义与初始化API
 - 7.7.2 用户注册API
 - 7.7.3 用户登录API
 - 7.7.4 接受用户状态API

1 简介

Robotic efficiency Driven - Domain Specific Language(ReD-DSL)定义了一个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着广泛的应用）的自动应答逻辑，并设计实现一个解释器，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。

ReD-DSL主要有以下几个特点：

- ReD-DSL使用ply完备地描述了一个文法，可以进行用户输入值或变量的比较，并存在错误检出与恢复功能。
- ReD-DSL按照Google开源项目风格指南进行代码编写，各模块代码可读性强。
- ReD-DSL采用面向对象的编程方法，各个模块之间耦合度低，并且每个模块皆可独立运行。
- ReD-DSL将输入和应答逻辑封装为Restful API以供调用，接口作用明确，定义清晰。
- ReD-DSL实现了一个Web应用，因此可以将需要的机器人客服作为插件，嵌入在任意的网页中以满足不同的客户需求。
- ReD-DSL提供了各个模块的完整测试和测试桩，以及自动测试脚本。
- ReD-DSL支持多用户多并发处理，使用ORM的方式进行不同用户的数据管理，并且保证了线程的安全。
- ReD-DSL的演示程序中支持调用一个基于深度学习的自然语言预训练模型([blenderbot-400M-distill](#))，从而使得用户可以与机器人进行任意自然语言的交流，从而优化了程序的人机接口。

2 程序风格

ReD-DSL的后端代码使用Python实现，按照Google开源项目风格指南的风格进行编写([Python风格指南](#))。所有的注释内容均使用英文进行描述。

2.1 模块命名风格

- 模块名写法: `module_name` ;包名写法: `package_name` ;类名: `ClassName` ;方法名: `method_name` (类中私有方法: `_method_name`) ;异常名: `ExceptionName` ;函数名: `function_name` ;全局常量名: `GLOBAL_CONSTANT_NAME` ;全局变量名: `global_var_name` ;实例名: `instance_var_name` ;函数参数名: `function_parameter_name` ;局部变量名: `local_var_name` 。
- 函数名,变量名和文件名应该是描述性的,尽量避免缩写. 始终使用 `.py` 作为文件后缀名。

不同类型的对象命名格式表格如下：

Type	Public	Internal
Modules	lower_with_under	_lower_with_under
Packages	lower_with_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected) or __lower_with_under (private)
Method Names	lower_with_under()	_lower_with_under() (protected) or __lower_with_under() (private)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

2.2 注释风格

2.2.1 模块注释风格

每个文件应该包含一个许可样板。根据项目使用的许可(Apache 2.0), 选择合适的样板。其开头应是对模块内容和用法的描述。

如ReD-DSL controller模块的注释：

```

1  """ Controller class is a hanlde to operate the state machine for the user
2
3  Controller has the methods to register a user, accept a condition, and get the
   current state of the user.
4  And for every return value from the user, it will record it in the database.
5  It is a wrapper of the state machine, being used to handle the user's input and
   output.
6
7  Typical usage example:
8
9  controller = Controller(lexer, parser, script, debug=True)
10 controller.register("test", "test")
11 """

```

2.2.2 函数注释风格

每节应该以一个标题行开始。标题行以冒号结尾。除标题行外, 节的其他内容应被缩进2个空格。

- Args:

列出每个参数的名字, 并在名字后使用一个冒号和一个空格, 分隔对该参数的描述。如果描述太长超过了单行80字符, 使用2或者4个空格的悬挂缩进(与文件其他部分保持一致)。描述应该包括所需的类型和含义。如果一个函数接受`foo`(可变长度参数列表)或者`**bar`(任意关键字参数), 应该详细列出`foo`和`**bar`。

- Returns: (或者 Yields: 用于生成器)

描述返回值的类型和语义。如果函数返回None, 这一部分可以省略。

- Raises:

列出与接口有关的所有异常。

如ReD-DSL interpreter模块的 `_login` 函数

```

1      """logs in a user
2
3      Args:
4          username: the username of the user
5          passwd: the password of the user
6
7      Returns:
8          True if the user is logged in, Exception otherwise
9
10     Exceptions:
11         Exception: if the user does not exist
12         Exception: if the password is incorrect
13     """

```

2.2.3 类注释风格

类应该在其定义下有一个用于描述该类的文档字符串。如果有公共属性(Attributes), 那么文档中应该有一个属性(Attributes)段。并且应该遵守和函数参数相同的格式。

如ReD-DSL interpreter模块的StateMachine类

```

1      """This class creates a state machine transition table
2
3      StateMachine will not create one instance for each state, but provides a
dictionary of states transition table.
4      Thus, the memory usage is reduced. And for all users, only one instance of
StateMachine is created. They share the same state machine.
5      Only the user variables are different. They are stored in the database. Users
can only read and write their own variables.
6      Users will query the transition table of the state machine to get the next
state and the actions of the next state.
7
8      Attributes:
9          AST (ASTNode): the abstract syntax tree of the DSL code
10         states_dict (dict): the states of the state machine
11         states_content (list): the content of the states
12         initial_state (str): the initial state of the state machine
13         debug (bool): the debug mode
14         variable_dict (dict): the variables of the state machine
15         action_dict (dict): the actions of the state machine
16         compare_list (list): the list of the comparison operators
17     """

```

2.2.4 块注释和行注释

最需要写注释的是代码中那些技巧性的部分。对于复杂的操作, 应该在其操作开始前写上若干行注释。对于不是一目了然的代码, 应在其行尾添加注释。为了提高可读性, 注释应该至少离开代码2个空格。

如ReD-DSL lexer模块中的113行注释

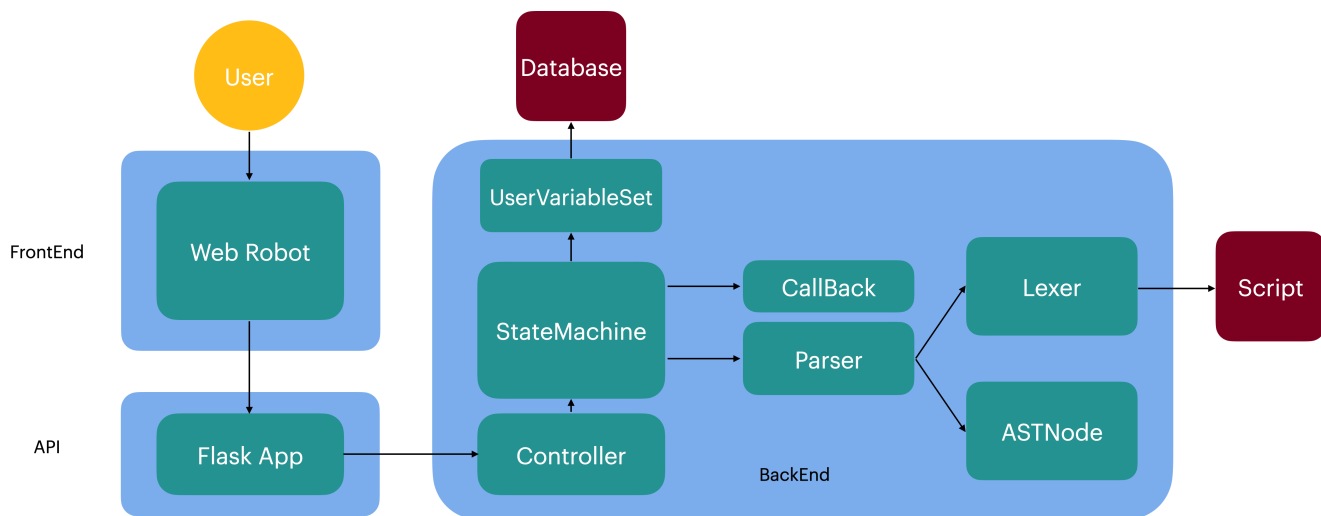
```

1  t.value = t.value[1:]    # remove the $

```

3 模块介绍

ReD-DSL的各个模块关系图如下：



- User代表用户，可以通过Web端与机器人发起对话，进行特定业务服务。
- Web Robot是ReD-DSL的前端模块，使用React实现了一个聊天对话框。
- Flask App是ReD-DSL的API接口，前端通过HTTP GET方法向Flask App发起请求，Flask App则将处理完的信息发送回前端。
- Controller是ReD-DSL后端的控制器，可以通过该模块进行数据库对象的修改、状态转移表的查询等业务处理。
- StateMachine是ReD-DSL运行时对应的状态机，负责解释抽象语法树，该模块主要由一张状态转移表 and 用户管理模块构成。
- UserVariableSet是ReD-DSL的用户变量到数据库的对象关系映射(ORM)，使用Python Storm库实现。
- Database是ReD-DSL存储用户变量的数据库实例，程序通过ORM的方式间接操作数据库对象。
- CallBack是ReD-DSL状态机状态转移表中存储的回调函数对象，代表状态机在收到不同的条件时需要执行动作的抽象类。
- Parser是ReD-DSL脚本语言的语法分析器，负责将执行脚本(Script)中标记的记号转化为一颗抽象语法树。
- Lexer是ReD-DSL脚本语言的词法分析器，负责将执行脚本(Script)中的记号提取出来，转化为标记提供给后续的语法分析模块。
- Script是ReD-DSL需要解释执行的脚本。

3.1 脚本语言语法

该模块主要由Parser、Lexer、ASTNode和Script构成。

3.1.1 脚本语言语法的BNF定义

使用C语言标准BNF(*The C programming language*, 2nd edition, by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1988.)进行定义

```

1  <script>                ::= "script" <id> <variables_defination>
   <states_defination>
2  <variables_defination>  ::= "variable" <var_clauses> "endVariable"
3  <var_clauses>           ::= <var_clause> | <var_clauses> <var_clause>
4  <var_clause>           ::= <identifier> "real" <variable> | <identifier> "integer"
   <variable> | <identifier> "text" <variable>
5  <states_defination>    ::= <state> | <states> <state>
6  <state>                ::= "state" <identifier> <expressions> "endState"
7  <expressions>          ::= <expression> | <expressions> <expression>
8  <expression>           ::= <speak> | <switch> | <goto> | <timeout> | <exit> |
   <update>
9  <speak>                ::= "speak" <terms>
10 <terms>                ::= <term> | <terms> "+" <term> | <term> "PLUS" <term> |
   <term> "MINUS" <term>
11 <term>                 ::= <string> | <variable> | <identifier> | <return>
12 <switch>               ::= "switch" <cases> <default> "endSwitch" | "switch"
   <cases> "endSwitch"
13 <cases>                ::= <case> | <cases> <case>
14 <case>                 ::= "case" <string> <expressions> | "case" <return>
   <expressions> | "case" <compare> <expressions>
15 <default>              ::= "default" <expressions>
16 <compare>              ::= <identifier> '>' <term> | <identifier> '<' <term> |
   <identifier> '>=' <term> | <identifier> '<=' <term>
   | <return> '>' <term> | <return> '<' <term> | <return> '>=' <term> | <return> '<='
   <term>
17 <goto>                 ::= "goto" <identifier>
18 <timeout>              ::= "timeout" <variable> <expressions> "endTimeout"
19 <exit>                 ::= "exit"
20 <update>               ::= "update" <identifier> "=" <terms>
21 <identifier>           ::= <letter_> | <identifier> <letter_> | <identifier>
   <number>
22 <variable>             ::= "$" <letter_> | "$" <number> | <variable> <letter_> |
   <variable> <number>
23 <string>               ::= <letter_> | <number> | <string> <letter_> | <string>
   <number>
24 <letter_>              ::= "_" | "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
25 <number>               ::= "0" | "1" | "2" | ... | "9"

```

3.1.2 脚本语言语法规则说明示例

本章节对脚本语言的语法规则按不同的模块和表达式进行举例的简单说明。

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、变量类型和变量初始值。变量名是由大小写字母和数字组成的字符串，实数和整数变量值则均为 \$ 开头，字符串变量则需要在头尾使用双引号注明。变量类型为 integer、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：

```
1 variable
2     x real $100
3     y integer $100
4     z text "hello"
5 endVariable
```

状态定义

一个状态定义由状态标识开始符"state"和结束符"endState"进行标记，内部可以包含speak动作、switch动作、timeout动作。

一个状态的定义示例如下：

```
1 state withdraw
2     speak "How much would you like to withdraw?"
3     switch
4         case _return <= x
5             speak "You have withdrawn " + _return + " dollars"
6             update x = x MINUS _return
7             goto welcome
8         case _return > x
9             speak "You do not have enough money in your account!"
10            goto welcome
11    endSwitch
12    timeout $30
13        speak "You have been idle for 30 seconds. Restarting service ..."
14        goto welcome
15    endTimeout
16 endState
```

switch语句定义

一个switch语句定义由语句标志开始符"switch"和结束符"endSwitch"进行标记，内部可以包含数个case语句，case语句可以选择有无default状态。

一个switch语句的定义示例如下：

```
1     switch
2         case "hello"
```

```

3         speak "hello"
4         goto welcome
5     case "balance"
6         speak "Your balance is " + x
7         goto welcome
8     case "topup"
9         goto topup
10    case "withdraw"
11        goto withdraw
12    case "exit"
13        goto goodbye
14    default
15        speak "Unknown command, please try again"
16        goto welcome
17    endSwitch

```

timeout语句定义

一个timeout语句定义由语句标志开始符"timeout"和结束符"endTimeout"进行标记，内部可以包含多个动作语句。

一个timeout语句的定义示例如下：

```

1    timeout $30
2        speak "You have been idle for 30 seconds. Restarting service ..."
3        goto welcome
4    endTimeout

```

动作语句

动作语句有speak、switch、goto、timeout、update和exit等类型。

上文已经介绍了块结构定义的语句switch和timeout，接下来介绍行结构定义的动作：

- speak动作代表机器人进行输出，speak可以引导一串字符串和变量标志符，支持使用[+]进行模块连接。
- goto动作代表机器人转换不同的状态，goto后必须紧跟一个存在的状态。
- update动作代表机器人更新用户指定的变量，操作可以为直接赋值，或者是ADD或MINUS操作，示例：`update x = x MINUS y`。
- exit动作由关键词exit单独引导，表示退出该机器人实例。

其他语法规则

为了获取用户的输入值，ReD-DSL语法定义了一个保留字_return，代表用户输入的内容，该内容会在每个状态开始后用户输入对应的内容时进行更新。

switch动作中的case语句支持条件比较，可以判断一个_return目标或者变量标识符目标是否满足某个条件，示例如下：

```

1      switch
2          case _return <= x
3              speak "You have withdrawn " + _return + " dollars"
4              update x = x MINUS _return
5              goto welcome
6          case _return > x
7              speak "You do not have enough money in your account!"
8              goto welcome
9      endSwitch

```

3.1.3 抽象语法树

ReD-DSL的词法分析器(lexer)将脚本语言进行分词后，由parser模块对脚本语言进行语法分析，并将script抽象为一颗语法树。

语法树类的定义如下：

```

1  class ASTNode:
2      """Abstract syntax tree node
3
4      Attributes:
5          type: the type of the node
6          childs: the child nodes
7
8      """
9      def __init__(self, type, *child):
10         """init the AST node"""
11         self.type = type
12         self.childs = list(child)

```

语法树的每个节点都有一个属性type，用于指明该节点的类型，同时还有一个列表属性childs，记录了语法树不同的孩子节点。

例如，对于一段测试脚本：

```

1  script Test2
2  variable
3      name text "Guest"
4  endVariable
5
6  state welcome
7      speak "Welcome, " + name + "!"
8      speak "Please Input [exit] to exit."
9      switch
10         case "exit"
11             exit
12         default
13             speak "I didn't understand that."
14     endSwitch

```

```

15     timeout $30
16         speak "No Respond in 30 seconds, exit."
17         exit
18     endTimeout
19 endState

```

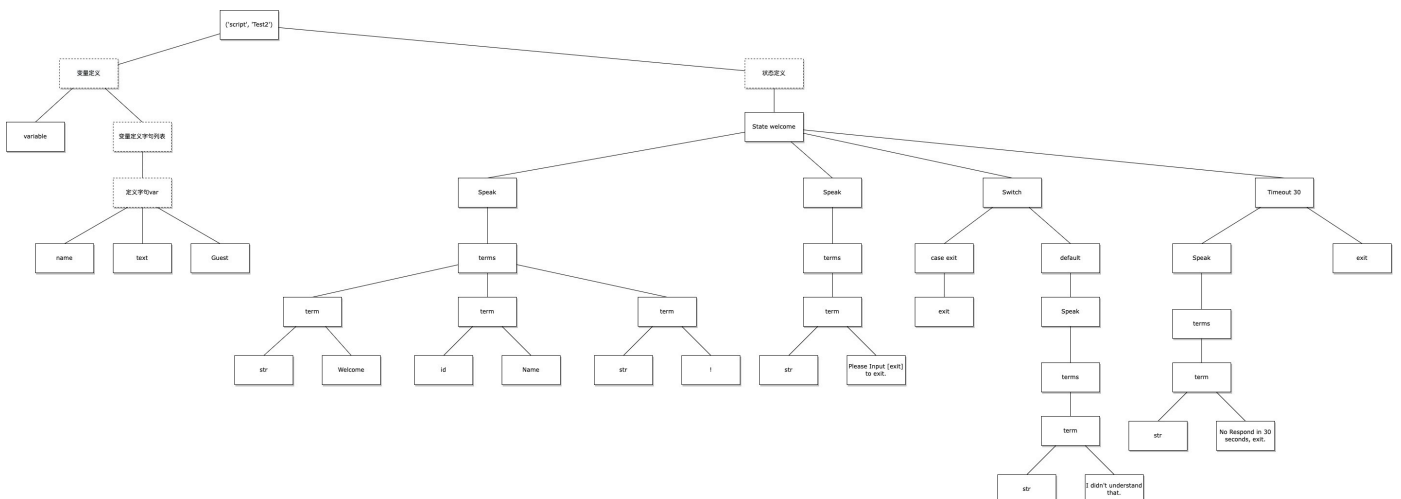
解析得到的抽象语法树输出为：

```

1  .
2  └─ ('script', 'Test2')
3      └─ variables
4          └─ ('var', 'name', 'text', 'Guest')
5      └─ states
6          └─ ('state', 'welcome')
7              └─ speak
8                  └─ terms
9                      └─ ('str', 'Welcome, ')
10                     └─ ('id', 'name')
11                     └─ ('str', '!')
12              └─ speak
13                  └─ terms
14                      └─ ('str', 'Please Input [exit] to exit.')
15              └─ switch
16                  └─ ('case', 'exit')
17                      └─ exit
18                  └─ default
19                      └─ speak
20                          └─ terms
21                              └─ ('str', "I didn't understand that.")
22          └─ ('timeout', '30')
23              └─ speak
24                  └─ terms
25                      └─ ('str', 'No Respond in 30 seconds, exit.')
26              └─ exit

```

该语法树的形态如下图所示：



3.1.4 模块接口(API)

该模块的各个子模块接口(API)定义如下（链接至[API查询表](#)）：

- [词法分析器\(Lexer\)API](#)
- [语法分析器\(Parser\)API](#)
- [抽象语法树\(ASTNode\)API](#)

3.2 解释器主模块

该模块主要由Controller、StateMachine、UserVariableSet和CallBack构成。

3.2.1 概述

该模块主要实现了一个解释器的功能，同时负责用户变量的管理。客服机器人与用户的交互一般为一问一答或者一问多答，因此解释器的底层逻辑是一个状态机，在一个状态下，给定用户的输入，或是用户未执行操作的秒数，状态机就会进行相应的条件转移判断，并执行存储在该转移判断中的所有动作，其输出是一个字符串序列。

为了实现多用户支持与减少内存占用，ReD-DSL不会为每个用户新建一个状态机实例，而是将解释得到的状态机按照状态和状态转移条件作为下标，使用Python字典的方法，存储对应的动作序列在字典的值中。因为采用了哈希表的实现，最快能在 $O(1)$ 时间内完成搜索，因此保证了状态机查询的高效。

ReD-DSL为存储机器人执行的动作序列，专门设计了CallBack类，用于将执行的某个动作抽象为CallBack类，保存在状态转移表中，从而可以在后续查询时再执行相应的动作。

此外，对于不同用户的变量，解释器通过Storm库建立用户变量集对象与数据库之间的ORM关系，使用状态机模块预留的API接口进行用户变量的增删改查等操作。

3.2.2 用户变量(UserVariableSet)

在ReD-DSL可以定义整型、浮点型、字符串型的用户变量。用户变量是每个用户私有的，每个用户只能访问自己的用户变量。每个用户变量都有一个默认值，在注册新用户时自动赋予，在脚本中可以使用Update动作进行用户变量的修改，也可以通过Speak动作向用户输出用户变量的值。

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。为了保证线程安全，每次数据库访问都需要使用互斥锁。

3.2.3 状态机(StateMachine)

在构建状态机转移表时，首先调用语法分析模块，在返回分析的基础上进行语义分析，并构建模型。状态机提供更新返回值、用户注册、用户登录、语义分析、测试比较条件接口。

3.2.4 状态转移表(StateMachine.action_dict)

为了实现多用户支持与减少内存占用，ReD-DSL不会为每个用户新建一个状态机实例，而是将解释得到的状态机按照状态和状态转移条件作为下标，使用Python字典的方法，存储对应的动作序列在字典的值中。

在解释器解释抽象语法树的过程中，会根据语法树的内容和结构，产生对应的状态转移表。状态转移表是一个二维字典，第一维的键值代表状态名，第二维的键值代表转移条件。转移条件可以是用户输入的内容，也可以是超时处理(<on_timeout>)、状态进入处理(<on_enter>)、比较处理(<compare>)或默认转移(<default>)。

状态转移表存储的内容是机器人执行的动作序列，使用CallBack类列表进行存储。每个CallBack类都对应了一个需要机器人进行执行的动作，在解释器解释过程中创建对应的CallBack对象，并保存需要进行的动作函数、函数参数、动作类型等内容，以方便后续用户查询时的调用。

由于使用了哈希表的实现，状态转移表的搜索时间复杂度为 $O(1)$ ，保证了状态机状态转移查询的高效。

根据用户输入内容的转移条件直接存储对应内容的字符串值，并在用户响应后，进行该状态下的比较，若满足该条件，则进行用户输入内容的转移处理。

每个状态可以指定状态进入处理，若需要使用状态进入处理，则状态转移表的状态转移条件会存储 `<on_enter>` 转移条件，若后续的Controller模块判断为状态进入，则会进行状态进入转移处理。

每个状态可以指定比较处理，若需要使用比较处理，则状态转移表会存储一个(`<compare>`, 比较ID)元组，并将需要进行的比较操作保存在compare_list中，若当前的状态有比较转移状态，就会在用户进行输入后检查每个比较条件的比较ID，在compare_list中通过指定下标查询的方法获取对应的比较条件，再进行比较的判断，若满足比较条件，就会进行比较处理。

每个状态可以指定超时处理，若需要使用超时处理，则状态转移表的状态转移条件会存储 `<on_timeout>` 转移条件，同时并将超时的时间值存储在 `<timeout_value>` 中，若当前用户未执行操作秒数达到了 `<timeout_value>`，就会进行超时转移处理。

3.2.5 控制器(Controller)

控制器是用户进行数据查询、状态转移等动作的中介模块，用于在状态转移表、用户变量等内部数据结构中进行查询、回调函数执行等操作。

因为状态转移表的每个单元存储了对应的CallBack类列表，而不执行相应的动作。因此只有在控制器层面执行对应的CallBack类存储的动作方法，机器人才能对用户的输入做出响应。在接受用户状态和状态转移条件时，控制器同时更新数据库中的用户输入值，并且记录是否发生了状态转移，若发生状态转移，则需要进行对应状态的状态进入处理。

控制器提供接受用户条件、用户登录、用户注册等接口。

3.2.6 模块接口(API)

该模块的各个子模块接口(API)定义如下（链接至[API查询表](#)）：

- [回调函数\(CallBack\)API](#)
- [用户变量集\(UserVariableSet\)API](#)
- [状态机\(StateMachine\)API](#)
- [控制器\(Controller\) API](#)

3.3 RESTful API

服务端采用Flask封装了RESTful API。

Flask原生支持多线程，由于在用户变量管理以及相关数据库修改等操作时已经设计了线程安全的方法，因此ReD-DSL获得了较好的并发性。

此外，对于每个用户，服务器均使用JWT鉴权，并在用户登录或注册时下发对应的Token。

GET /send

客户端发送一条新消息，服务器返回相应。

Param 客户端发送一条消息，用户的当前状态和token，格式为：`{"msg": "some message", "state": "current_state", "token": "some token"}`。

Return 返回一条消息，用户的下一个状态，超时时间值和是否退出标志，以及状态值200，格式为：`{"msg": "some message", "state": "next_state", "timeout": "timeout_value", "exit": "True/False"}` and HTTP status code 200。

Raises 如果状态转移出现异常，则返回报错信息和错误状态值500。

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，格式为：`{"username": "some username", "password": "some password"}`。

Return 返回一个新的token和欢迎消息，以及状态值200，格式为：`{"token": "some token", "msg": "welcome message"}` and HTTP status code 200。

Raises 如果注册的用户名已经存在，则返回错误信息和错误状态值403。

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，格式为：`{"username": "some username", "password": "some password"}`。

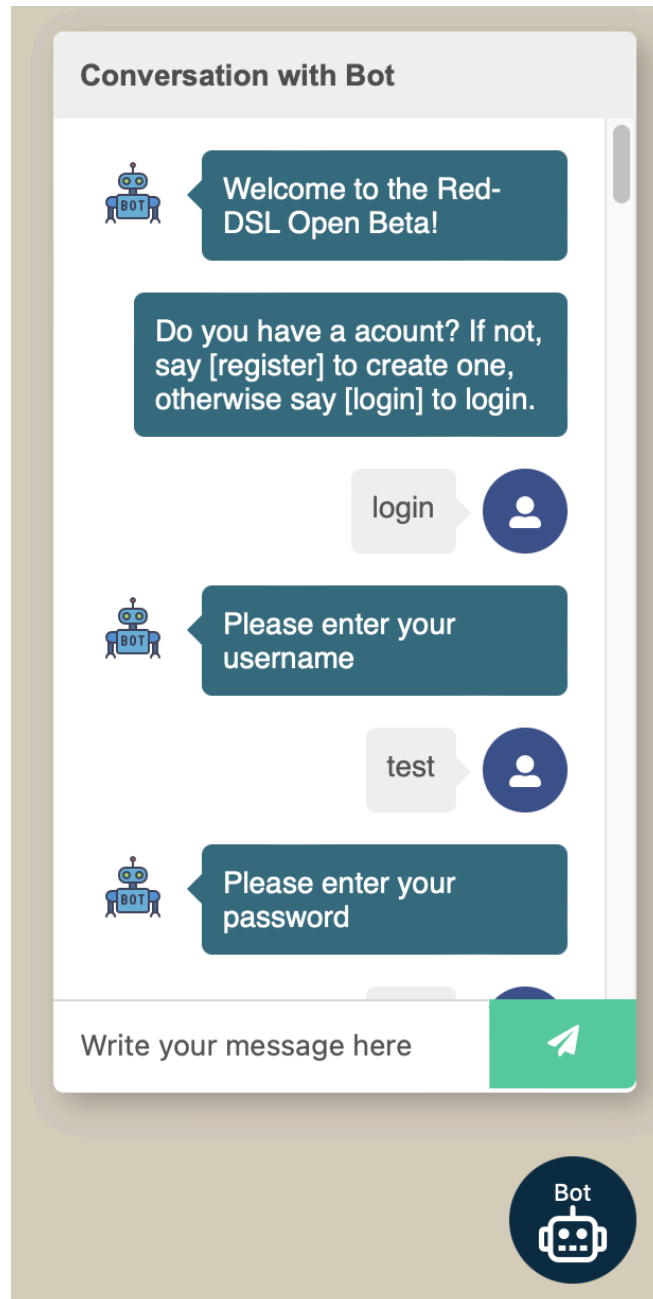
Return 返回返回一个新的token和欢迎消息，以及状态值200，格式为：`{"token": "some token", "msg": "welcome message"}` and HTTP status code 200。

3.4 客户端

客户端采用React设计，作为一个Web App可以被挂载在任何一个需要ReD-DSL承担客户机器人的网站。

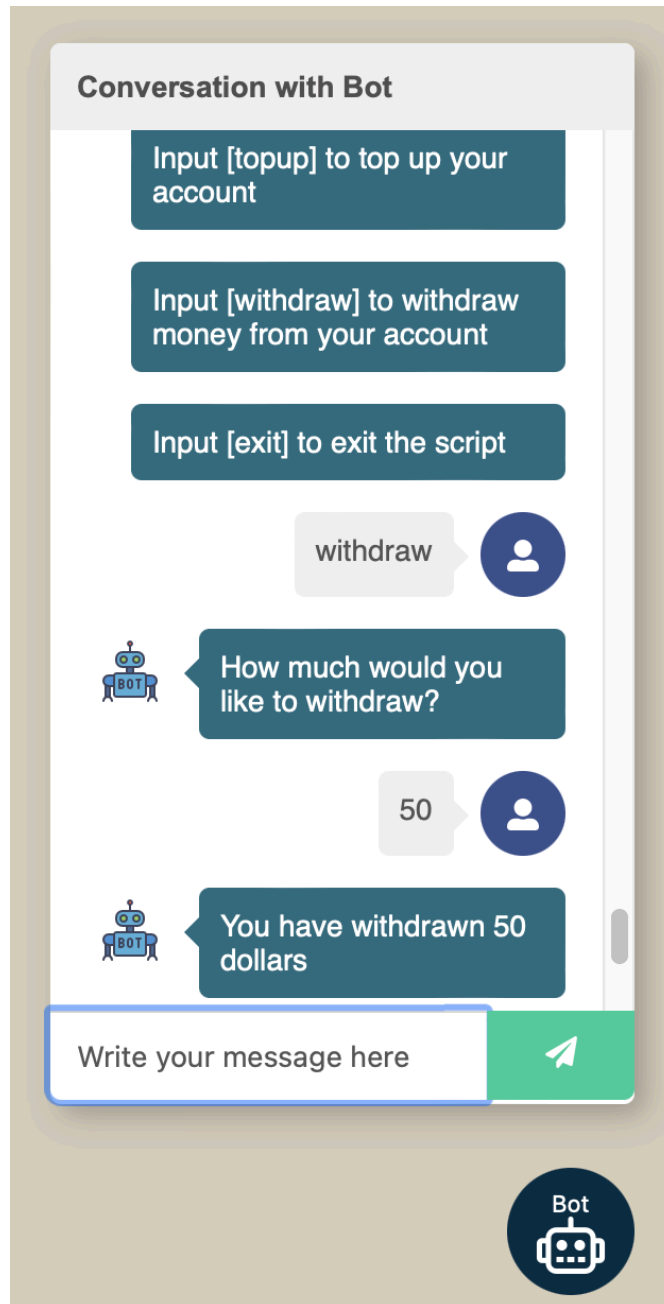
3.4.1 用户注册与登录

用户的注册与登录同样使用对话框方式进行，根据机器人的提示用户进行相应的登录或注册操作。



3.4.2 机器人客服功能

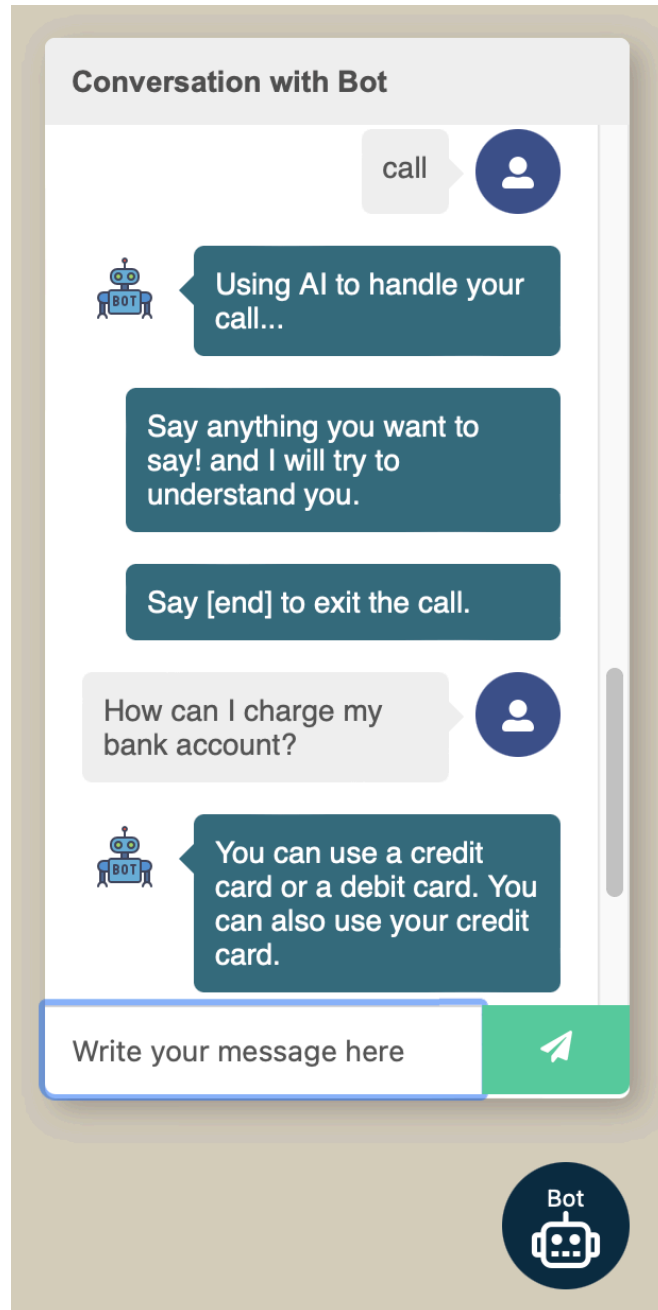
用户根据机器人提示输入对应的语句，机器人进行相关语句的回应。



3.4.3 自然语言大模型 (Large Language Model)

ReD-DSL的客户端支持调用一个基于深度学习的自然语言预训练模型([blenderbot-400M-distill](#)), 从而使得用户可以与机器人进行任意自然语言的交流。

在演示程序中, 该模型用以模拟人工服务, 用户在登录后输入call呼叫人工客服, 随后可与人工客服进行任意自然语言对话, 输入end结束呼叫客服。



4 测试

4.1 自动测试脚本

对于后端的各个模块，都有对应的自动测试脚本，每个脚本都可以独立运行，以测试每个模块的功能是否正确。在开发过程中，当某个模块被修改时，就运行对应模块的测试脚本，已检测模块在修改后是否依然正确。

ReD-DSL提供以下自动测试脚本

```
1 test.test_app
2 test.test_controller
3 test.test_parser
4 test.test_state_machine
```

运行某个模块的自动测试脚本，如：

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

检查得到的结果，如果输出OK则说明自动测试完成且无误，如：

```
1 .
2 -----
3 Ran 1 test in 0.004s
4
5 OK
```

4.2 测试桩

后端各个存在联调过程的模块，都可以直接独立运行，调用测试桩——模拟需要联调的模块，测试的模块/方法所调用的某个模块或系统，模拟返回值。

使用测试桩可以有以下优点：

- 在测试时避免上游模块修改导致的错误
- 测试桩数据避免了人为干预
- 保证了上游输入的正确性
- 减少测试成本

4.2.1 状态机(StateMachine)测试桩

使用Python pickle模块，读取预存完毕的抽象语法树，该抽象语法树即为测试桩。将读取完的抽象语法树作为状态机初始化函数的参数进行初始化，从而可以在没有语法分析模块的前提下完成状态机模块的测试。

运行状态机测试桩：

```
1 python -m server.interpreter
```

4.2.2 控制器(Controller)测试桩

同样使用Python pickle模块，读取预存完毕的状态机，该状态机即为测试桩。同时将控制器的初始化参数 `stub` 设置为 `True`，说明使用测试桩模式进行初始化，随后将读取完成的测试桩状态机输入给控制器，从而可以在没有状态机模块的前提下完成控制器模块的测试。

运行控制器测试桩：

```
1 python -m server.controller
```

4.2.3 前端测试桩

为了在没有后端模块的情况下对前端进行测试，ReD-DSL使用Flask测试模块实现了一个简单功能的后端，可以测试前端的逻辑与功能是否正确。

运行前端测试桩：

```
1 python -m test.test_client
2 cd client
3 yarn start
```

在前端网页上根据机器人输入交互信息即可进行测试。

4.2.4 后端测试桩

为了在没有前端模块的情况下对后端进行测试，ReD-DSL拥有一个后端测试桩，可以测试后端的逻辑与功能是否正确。

运行后端测试桩：

```
1 python -m test.test_server
```

该测试运行后使用命令行(CLI)进行交互，根据屏幕打印的提示信息进行输入。

4.3 API测试

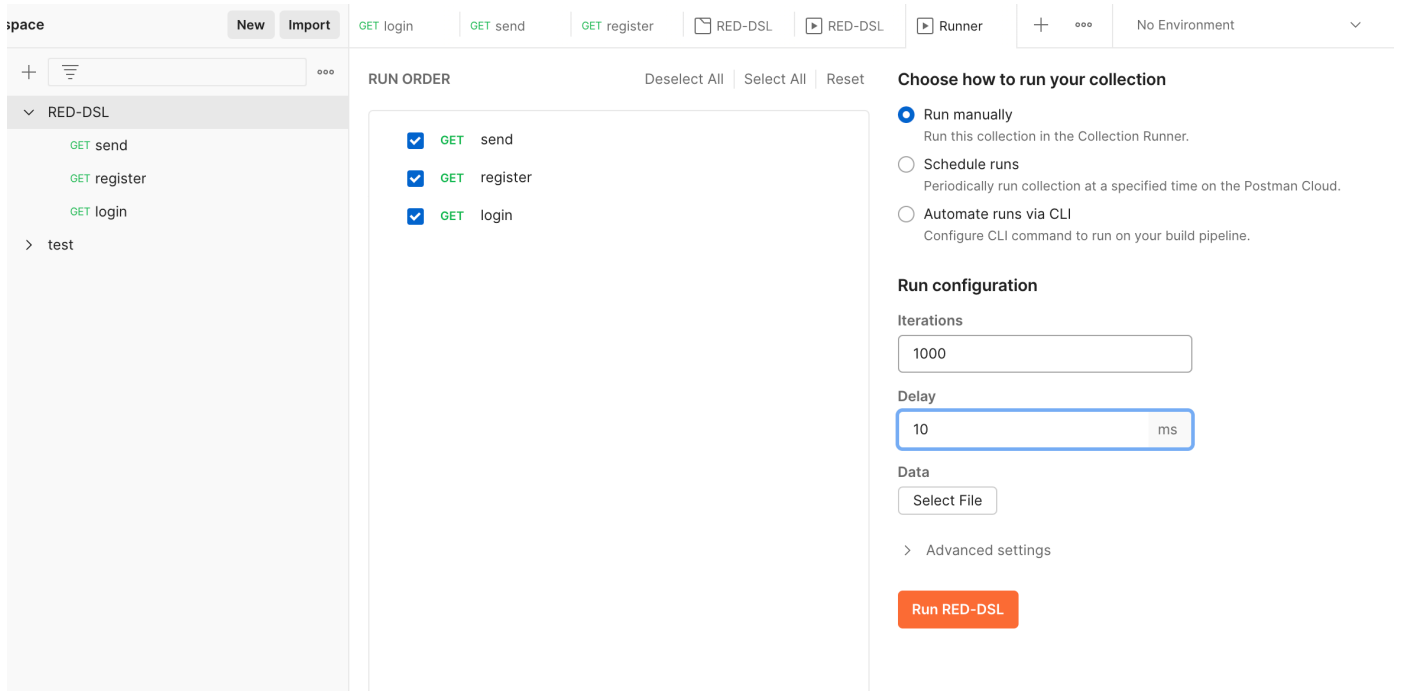
4.3.1 API测试设置

API测试除了在test_app中模拟了各个API的响应功能外，ReD-DSL还使用了Postman Tests的API自动测试工具进行测试。

使用Postman Tests Runner功能，将ReD-DSL后端的三个API进行测试：

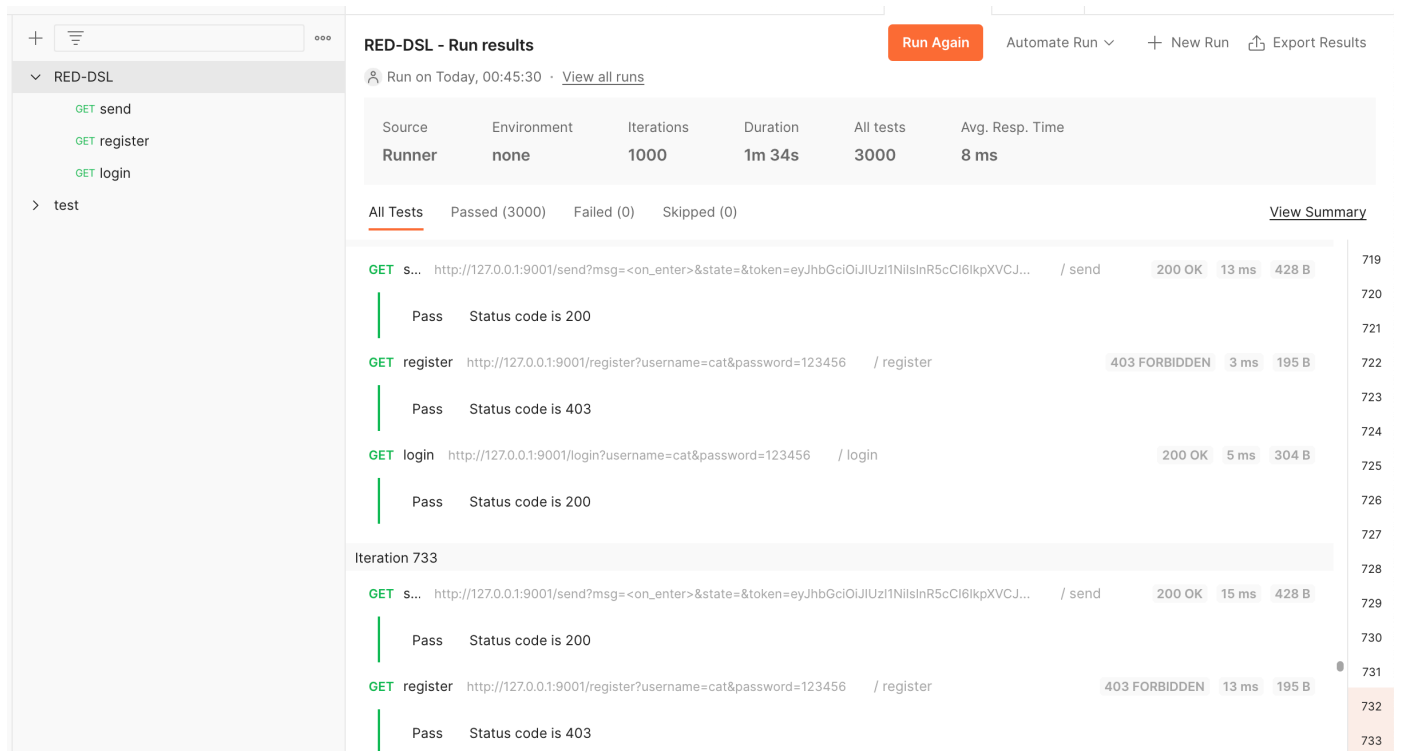
- **/send** 重复发送一条进入状态机的消息，期望得到服务器状态码正常响应200
- **/register** 重复注册一个已经注册的账号，期望得到服务器状态码报错403
- **/login** 重复登录某个账号，期望得到服务器状态码正常响应200

在Postman Tests Runner中选择迭代次数1000次，每次请求的间隔为10ms，一共向发送3000条HTTP GET请求。



4.3.2 API测试结果

测试迭代1000次，每次请求间隔为10ms时，一共使用1m34s完成测试，服务器平均响应时间为8ms，证明服务器API可以正确运行。



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，ReD-DSL设计了压力测试，并行开启 100 个客户端对服务器发送请求，请求均为对数据库的访问。

压力测试使用银行服务脚本，在100个客户端中，每个客户端都会首先创建一个用户账号，再分别执行充值101元，再支取100元的操作，重复100次，最后检查每个用户账号中的余额是否等于200元（初始值为100元）。

运行压力测试的命令如下：

```
1 | python -m test.test_pressure
```

压力测试结果如下：

```
1 | .
2 | -----
3 | Ran 1 test in 231.698s
4 |
5 | OK
```



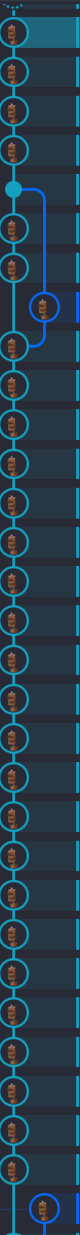

5 开发日志与版本管理

ReD-DSL的开发过程中使用了Git进行版本管理，并且采用自底向上的开发方式，同时也持续更新测试脚本与测试桩，保证模块的独立运行与功能准确无误：

- 词法分析器
- 语法分析器
- 解释器
- 状态机与状态转换表
- 控制器
- 后端
- API开发
- 前端

在开发过程中，测试步骤也同时更新，在commit当前的版本之前，确保测试结果无误。

Git Commit的部分记录如下：

BRANCH / TAG	GRAPH	COMMIT MESSAGE	AUTHOR	COMMIT DATE / TI...	SHA
✓   master		[Done] Refactor	You	2 days ago	67d4a62
		[Done] Change README	You	3 days ago	1d68dc3
		[Working] Add Unit Test	You	3 days ago	9cb9b5e
		[Working] Add Unit Test	You	3 days ago	461c9be
		[Working] Merge Confilcts	You	3 days ago	52b922c
		[Working] Add Unit Test	You	3 days ago	02a8281
		[Done] Directory Refactor	You	3 days ago	09d249f
		[Done] Directory Refactor	You	3 days ago	2460a55
		[Done] Modified Comments	You	4 days ago	51be973
		[Done] Change File Names	You	4 days ago	67b86f6
		[Done] Fix Comparing Bugs	You	5 days ago	1df2f6b
		[Done] Add Comparing	You	5 days ago	761bf28
		[Working] Add Comparing	You	5 days ago	46c8130
		[Working] Under Test	You	6 days ago	2146e52
		[Working] Timeout Change	You	last week	1c71d3d
		[Done] Interpreter	You	last week	89d77d1
		[Done] Usermanage	You	last week	293f222
		[Done] Naive Web	You	last week	b6a4992
		[Done] Register	You	last week	ddcc3e5
		[Done] Naive DB	You	last week	922cbaf
		[Done] Update Test File	You	2 weeks ago	f1aa14d
		[Done] Return Value	You	2 weeks ago	0b55f94
		[Done] Format	You	2 weeks ago	1400a1c
		Simple Timeout [Done]	You	2 weeks ago	d88faa4
		Finished Default	You	2 weeks ago	986f4f5
		Least Runnable	You	2 weeks ago	4db5676
		Finish Transition Table	You	2 weeks ago	1473aa6
		writing the transition table	You	2 weeks ago	a4bfab5
		No Collision	You	2 weeks ago	2442494
		test newline	You	2 weeks ago	83f09c6
 newline		modifying newlines	You	2 weeks ago	196f113

6 附录：文件结构

```

1  .
2  |— README.md //ReD-DSL用户手册
3  |— client //客户端文件夹
4  |   |— README.md //客户端用户手册
5  |   |   ...
6  |   |— yarn.lock
7  |— config.json //ReD-DSL配置文件
8  |— script //可运行的脚本
9  |   |— bank_service.txt //银行服务脚本
10 |   |— echo.txt //echo脚本
11 |   |— hello.txt //hello脚本
12 |   |— mobile_fee.txt //移动设备流量充值查询脚本
13 |— server //服务端文件夹
14 |   |— __init__.py //python init标识
15 |   |— app.py //后端flask API
16 |   |— controller.py //控制器
17 |   |— database.db //默认数据库
18 |   |— interpreter.py //解释器
19 |   |— lexer.py //词法分析器
20 |   |— parser.out //LALR1分析结果
21 |   |— parsetab.py //ply中间文件
22 |   |— yacc.py //语法分析器
23 |— test //测试文件夹
24 |   |— __init__.py //python init标识
25 |   |— script //测试脚本
26 |       |— app_test //测试api
27 |       |   |— output1.txt
28 |       |   |— output2.txt
29 |       |   |— output3.txt
30 |       |— controller_test //测试控制器
31 |       |   |— output1.txt
32 |       |   |— result1.txt
33 |       |   |— test1.txt
34 |       |— interpreter_test //测试解释器
35 |       |   |— result1.txt
36 |       |   |— result2.txt
37 |       |   |— test1.txt
38 |       |   |— test2.txt
39 |       |— parser_test //测试语法分析器
40 |       |   |— result1.txt
41 |       |   |— result2.txt
42 |       |   |— test1.txt
43 |       |   |— test2.txt
44 |       |   |— test3.txt
45 |       |   |— test4.txt
46 |       |— test.txt

```

```
47 |— stub //测试桩文件
48 |   |— ast.stub //语法树测试桩文件
49 |   └─ state_machine.stub //状态机测试桩文件
50 |— test_app.py //后端API自动测试脚本
51 |— test_client.py //测试客户端的测试桩
52 |— test_controller.py //控制器自动测试脚本
53 |— test_parser.py //语法分析器自动测试脚本
54 |— test_pressure.py //压力测试测试脚本
55 |— test_server.py //测试服务端的测试桩
56 |— test_state_machine.py //状态机自动测试脚本
```

7 附录：接口（API）查询表

7.1 词法分析器(Lexer) API

7.1.1 类定义与初始化API

```

1 class Lexer:
2     """Lexer is for tokenizing the input script
3
4     Attributes:
5         _lexer: the lexer object
6         input: the input target
7     """
8     def __init__(self):
9         """init the lexer
10
11     Attributes:
12         _lexer: the lexer object
13         input: the input target
14     """

```

7.1.2 获取扫描的下一个记号API

```

1     def token(self):
2         """get the next token
3
4     Returns:
5         the next token tokenized
6
7     Raises:
8         SyntaxError: if the input is illegal
9         RuntimeError: if the input is not loaded
10    """

```

7.1.3 从地址读取脚本API

```

1     def load_script(self, path):
2         """load the script file
3
4     Args:
5         path: the script file path
6     """

```

7.1.4 从字符串读取脚本API

```

1  def load_str(self, input):
2      """load the input string
3
4      Args:
5          input: the input string
6      """

```

7.1.5 返回词法分析器API

```

1  def get_lexer(self):
2      """get the lexer object
3
4      Returns:
5          the lexer object
6      """

```

7.2 语法分析器(Parser) API

7.2.1 类定义与初始化API

```

1  class Parser:
2      """Parse the script and return the AST
3
4      Attributes:
5          _lexer: the lexer
6          tokens: the tokens
7          _yacc: the yacc parser
8          debug: the debug mode
9      """
10     def __init__(self, lexer: Lexer, debug=False):
11         """init the parser"""

```

7.2.2 对脚本进行分析API

```

1  def parse(self, script):
2      """parse a script
3
4      Args:
5          script: the script to parse
6
7      Returns:
8          the AST of the script
9
10     Raises:
11         SyntaxError: if the script is not valid
12     """

```

7.3 抽象语法树(ASTNode) API

7.3.1 类定义与初始化API

```

1  class ASTNode:
2      """Abstract syntax tree node
3
4      Attributes:
5          type: the type of the node
6          childs: the child nodes
7      """
8      def __init__(self, type, *child):
9          """init the AST node
10
11      Args:
12          type: the type of the node
13          childs: the child nodes
14      """

```

7.3.2 打印语法树API

```

1  def print(self, indent=1):
2      """print the AST
3
4      Args:
5          indent: the indent of the current node
6      """

```

7.4 回调函数(CallBack) API

7.4.1 类定义与初始化API

```

1 class CallBack(object):
2     """CallBack class used to perform the actions of the DSL code
3
4     Attributes:
5         callback (function): the callback function
6         args (tuple): the arguments of the callback function
7         type (str): the type of the callback function
8     """
9     def __init__(self, callback, *args):
10         """Inits CallBack with callback and args"""

```

7.4.2 执行回调函数API

```

1 def __call__(self, *args):
2     """calls the callback function"""

```

7.5 用户变量集(UserVariableSet) API

7.5.1 类定义与初始化API

```

1 class UserVariableSet(object):
2     """creates a new user variable set
3
4     Attributes:
5         username (str): the username of the user
6         passwd (str): the password of the user
7     """
8     def __init__(self, username: str, passwd: str):
9         """Inits UserVariableSet with username and passwd
10
11         Note that other attributes can be added dynamically
12
13         Args:
14             username: the username of the user
15             passwd: the password of the user
16     """

```

7.6 状态机(StateMachine) API

7.6.1 类定义与初始化API

```

1  class StateMachine:
2      """This class creates a state machine transition table
3
4      StateMachine will not create one instance for each state, but provides a
      dictionary of states transition table.
5
6      Thus, the memory usage is reduced. And for all users, only one instance of
      StateMachine is created. They share the same state machine.
7
8      Only the user variables are different. They are stored in the database. Users
      can only read and write their own variables.
9
10     Users will query the transition table of the state machine to get the next
      state and the actions of the next state.
11
12     Attributes:
13         AST (ASTNode): the abstract syntax tree of the DSL code
14         states_dict (dict): the states of the state machine
15         states_content (list): the content of the states
16         initial_state (str): the initial state of the state machine
17         debug (bool): the debug mode
18         variable_dict (dict): the variables of the state machine
19         action_dict (dict): the actions of the state machine
20         compare_list (list): the list of the comparison operators
21         db_path (str): the path of the database
22
23     """
24     def __init__(self, AST : ASTNode, db_path="./database.db", debug=False):
25         """Inits StateMachine with AST and debug
26
27         Args:
28             AST (ASTNode): the abstract syntax tree of the DSL code
29             db_path (str): the path of the database
30             debug (bool): the debug mode
31         """

```

7.6.2 解释脚本API

```

1  def interpret(self):
2      """interprets the DSL code and builds the state machine transition table
3
4      Raises:
5          Exception: if the DSL code is invalid
6      """

```

7.6.3 判断测试条件API

```

1  def test(self, condition, num, username="Guest"):
2      """tests the condition
3
4      Args:
5          condition: the condition to test
6          num: the entry to compare list
7
8      Returns:
9          True if the condition is true, False otherwise
10
11     Raises:
12         Exception: if the condition type is unknown
13     """

```

7.6.4 更新用户返回值API

```

1  def update_return_value(self, value, username="Guest"):
2      """updates the return value
3
4      Args:
5          value: value is the value to return
6          username: the username of the user
7      """

```

7.6.5 用户注册API

```

1  def register(self, username, passwd):
2      """registers a new user
3
4      Args:
5          username: the username of the new user
6          passwd: the password of the new user
7
8      Returns:
9          "Registered" if the user was registered successfully
10
11     Raises:
12         Exception: if the user already exists
13     """

```


7.6.6 用户登录API

```

1  def login(self, username, passwd):
2      """logs in a user
3
4      Args:
5          username: the username of the user
6          passwd: the password of the user
7
8      Returns:
9          True if the user is logged in, Exception otherwise
10
11     Exceptions:
12         Exception: if the user does not exist
13         Exception: if the password is incorrect
14     """

```

7.7 控制器(Controller) API

7.7.1 类定义与初始化API

```

1  class Controller:
2      """Controller class for the State machine, acts as the interface between the
3      state machine and the user
4
5      Controller has the methods to register a user, log in a user, accept a
6      condition of the user.
7
8      For every return value from the user, it will record it in the database.
9      User query with the current state of the user to get the next state and the
10     output as well as the timeout value.
11
12     Attributes:
13         _lexer: the lexer used to tokenize the script
14         _parser: the parser used to parse the script
15         debug: the debug mode
16         state_machine: the state machine
17         action_table: the action table of the state machine
18         _return: the return value of the user
19     """
20
21     def __init__(self, lexer: Lexer, parser: Parser, script,
22                  db_path="./database.db", debug=False, stub=False):
23         """init the controller with the script
24
25         Args:

```

```

21         lexer: the lexer used to tokenize the script
22         parser: the parser used to parse the script
23         script: the script of the state machine
24         db_path: the path of the database
25         debug: the debug mode
26         stub: the stub mode
27         """

```

7.7.2 用户注册API

```

1  def register(self, username: str, password: str):
2      """register a new user
3
4      Args:
5          username: the username of the user
6          password: the password of the user
7
8      Returns:
9          True if the user is registered successfully
10
11     Raises:
12         Exception: if the user is already registered
13     """

```

7.7.3 用户登录API

```

1  def login(self, username: str, password: str):
2      """login a user
3
4      Args:
5          username: the username of the user
6          password: the password of the user
7
8      Returns:
9          True if the user is logged in successfully
10
11     Raises:
12         Exception: if the user is not registered or the password is wrong
13     """

```

7.7.4 接受用户状态API

```

1  def accept_condition(self, current_state: str, condition: str, username="Guest"):
2      """accepts a condition, performs the required action and returns the next state,
3      output and timeout
4
5      The controller will check the database of the state machine to perform the
6      required action if there is any.

```

```
5      Or it will check the action table to perform the required action if there is
6      any.
7
8      Args:
9          current_state: the current state of the user
10         condition: the condition to check
11         username: the username of the use
12
13     Return:
14         next_state: the next state of the user
15         output: the output of the state machine
16         timeout: the timeout of the state machine
17         is_exit: whether the state machine is exited
18
19     Raises:
20         Exception: if the current state is not in the state machine
21         Exception: if the condition is not in the state machine
22
23     """
```