DSL设计

设计和实现

设计场景

- 本DSL主要针对于服务器端解析程序，用户与程序之间主要通过通过文件或者网络传输信息，程序不下放到客户端
- 本程序针对客服机器人的常见场景，设计了main function 和 final function 两个语法糖，提高了开发效率。
- 针对多用户场景，设置了var_vault 机制，可以根据用户名称等key方便的保存不同用户的不同数据。
- 针对单程序多进程，设置了 `--save_parsed` 和 `--load_parsed` 两个参数来保存和加载处理之后的程序，提高执行效率，同时可以通过shell脚本实现参数切换，实现同一程序多个进程。
- 本程序高度内聚，可以通过文件，web或者stdio与其他模块进行组合，可以通过shell脚本实现通过pipe的联通，或者通过文件保存实现信息交换，或者通过web实现语音转文字等拓展功能。
- 本程序一定程度上可以通过shell调用其他程序，进一步增强了DSL语言的功能与扩展性

主要数据结构

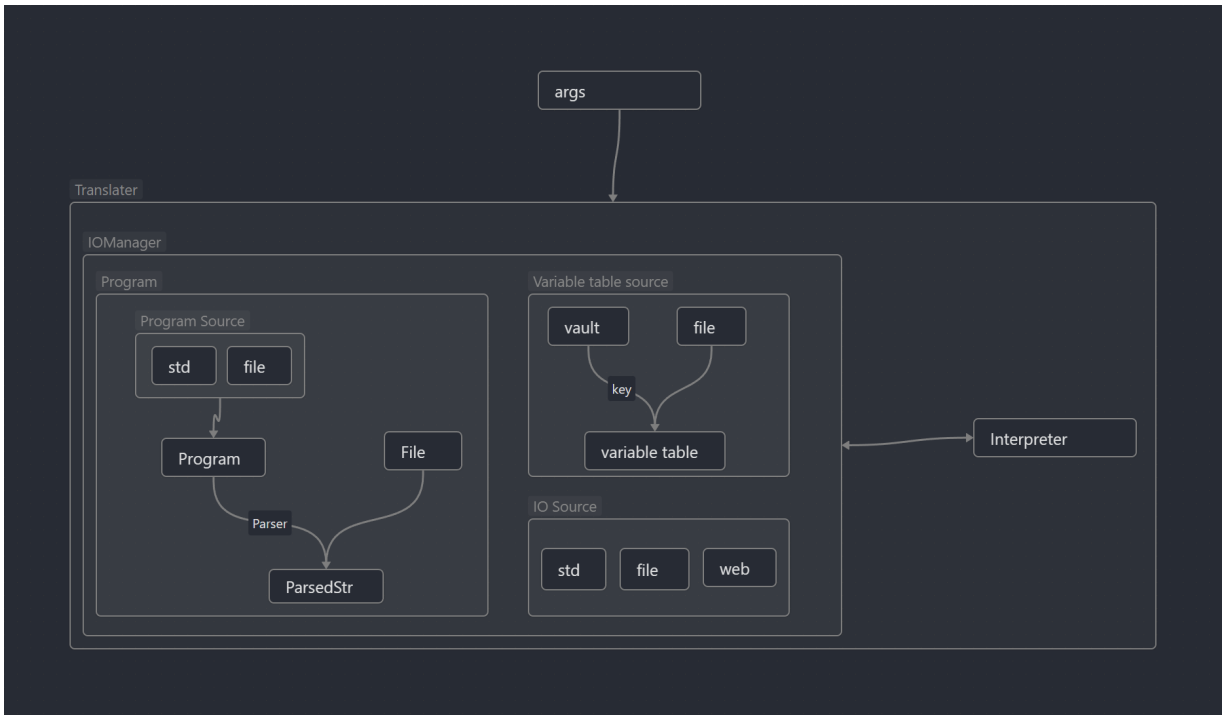
- args
    - 用于初始化时传递设置参数
- ParserElement

    - 为读取到的 `string` 处理后结果
    - 传递给 `Interpreter` 进行进一步处理
- Interpreter
    - 解释器部分
    - varList
        - 保存所有声明过的变量
        - 以字典形式保存
    - funcList
        - 保存所有声明过的函数
        - 以字典形式保存
    - exitflag
        - 用于跳过后续语句执行
        - 具体功能见 功能 章节

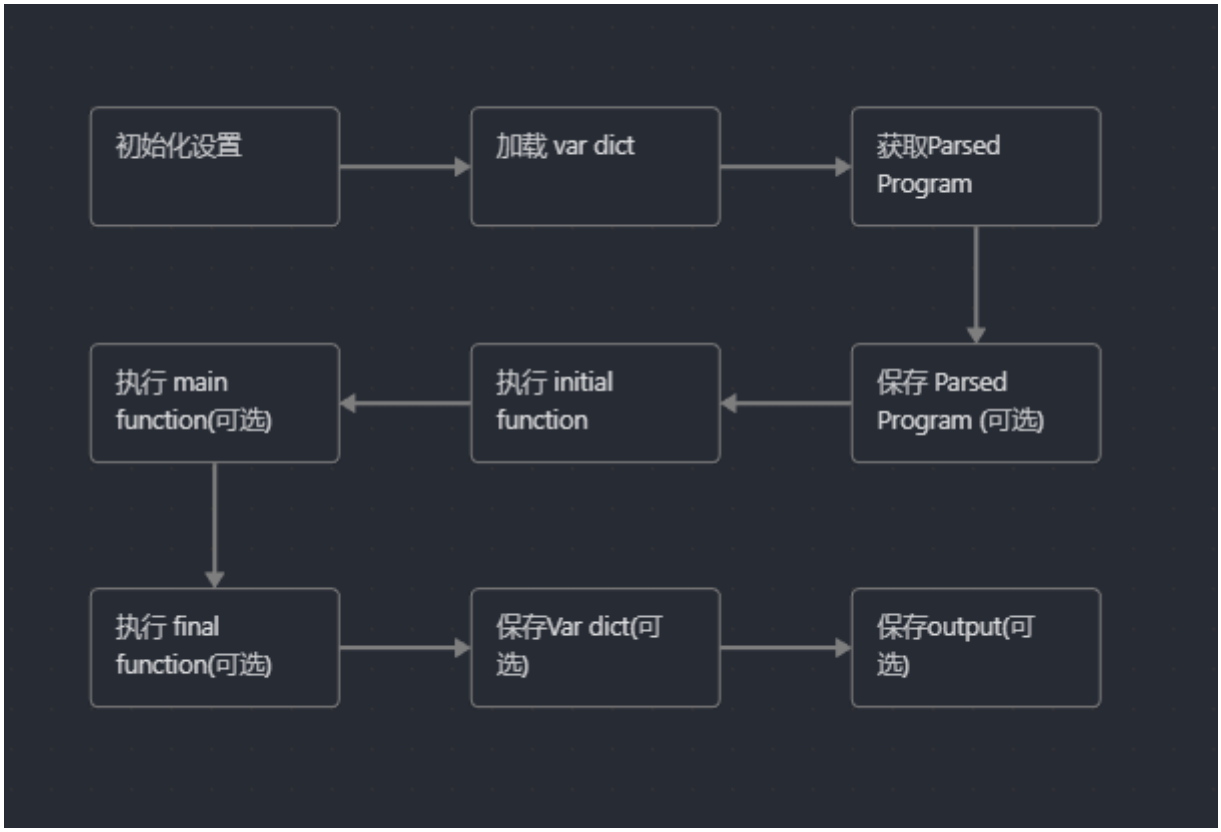
模块划分

- 主要模块分为
    - Translater
        - 用于连接其余三个模块，并作为与用户之间的接口
        - 接受用户输入的参数，并将其传给其余三个模块用于初始化配置
    - Parser
        - 用于对输入的原始Program进行处理，将其按照语法规则分为不同层次的list，为Interpreter进行语义解析做准备
    - Interpreter

        - 用于将Parser处理过的程序语义解析并执行
        - 与IOManager交互获取程序，变量表，IO信息等
    - IOManager
        - 用于管理相关数据输入输出，如程序，数据，交互过程的输入输出等，用于实现从不同来源读取IO和程序数据
- 总架构图



执行流程



参数

- 具体参数使用说明见 功能 章节
- 支持长指令与短指令
- 支持输入 `--help` 查询参数说明

```
'--program_file', '-pf', type=str, default=None, help
'--save_parsed', '-sp', type=str, default=None, help=
'--load_parsed', '-lp', type=str, default=None, help=
'--save_file', '-sf', type=str, default=None, help='s
'--save_from_key', '-sk', type=str, default=None, hel
'--load_file', '-lf', type=str, default=None, help='l
'--load_from_key', '-lk', type=str, default=None, hel
'--var_vault', '-vv', type=str, default='./var_vault/
'--debug', '-d', action='store_true', help='debug moc
'--mode', '-m', type=str, default='std', choices=['st
'--input', '-i', type=str, default=None, help='input
```

```
'--output', '-o', type=str, default=None, help='outpu
'--url', '-u', type=str, default=None, help='set web
```

语法

- 语句相应 BNF 范式见本节末尾
- block
  - 用 `{ }` 括起的若干sentences
  - 在最外层`{ }` 外不允许有内容
- sentence
  - 以 `;` 结尾的若干符合语法规则的字符串
  - 具体语法规则见下文
- variable
  - 变量名开头需加 $
  - 支持整数，浮点，字符串三种数据类型
  - 变量全局有效，不支持作用域
  - 输入字符串类型时，需要`"` 将相应字符串两端括起来
    - 当前字符集包含
      - 字母
      - 数字
      - 键盘上除 `"` 外可输出符号
        - 半角字符
    - 不支持汉字与全角字符
    - 字符串内容中不允许有 `"`
  - eg
    - `$a`
- expression

- 整数，浮点数支持`+ - * / ==` 5种运算
  - 分别对应四则运算与 比较运算符
  - `+ - * /`
    - 返回相应运算结果
  - `==`
    - 相同时返回1，不同返回0
- 字符串支持 `in == &` 3种运算
  - `a in b` 运算 用于判断 变量a是否位于 b中
    - 当 a 或 b不为字符串格式时会将其转换为字符串格式之后再进行运算
  - `a == b` 运算用于判断 变量 a和 b 是否相同
    - 相同时返回1，不同返回0
  - `a & b` 运算用于连接 变量 a和 b
    - 如果 a 和 b 不为字符串类型，则会将其转换为字符串之后进行连接
    - 不会自动补充空格
- 当 a 与 b 数据类型不同时，比较运算符会返回 0
- 运算优先级
  - $(* /) > (+ -) > (==) > (in\ \&)$
  - 支持 括号改变优先级
  - 相同优先级从左往右计算
- eg
  - `1 + 1`

- function
  - 函数名开头需为 _
  - 不支持传参，变量全局有效

    - `_main` 与 `_final` 函数有特殊功能，见 功能 章节
    - 声明函数
        - 需以 `func` 字段开头
        - 在后续block中书写调用该函数时需要执行的sentence
    - 执行函数
        - 函数名 + ;
    - eg
        - 函数声明
            - `func _test { send "test" ;};`
        - 函数执行
            - `_test ;`
- loop
    - 对指定func或block重复执行n次
        - $n \geq 0$
    - 只支持手动输入整数次数，不支持变量以及浮点数
    - eg
        - `loop 3 times _test ;`
        - `loop 3 times { send "test" ;};
- condition
    - 条件执行语句
    - 支持多分支
    - 支持else
    - 语句执行部分支持放入函数名或block
    - 条件判断时需为数值类变量
        - 当前值大于1即视为True

    - eg
        - `if "enquiry" in $cmd then _find elif "complain" in $cmd then _complain else { send "sorry, we don't support this function yet"; };`
- assignment
    - 将等号右侧的值赋给等号左边变量
    - 等号右侧可为任意表达式，左侧需为变量名
    - 若左侧变量未被赋值过，则创建该变量并存入值
    - 若左侧变量已被赋值过，则更新该变量值
    - eg
        - `$a = 1;`
- get
    - 从数据源读取数据至给定变量中
        - 数据源设置见 参数 章节
        - 数据源说明见 功能 章节
    - 如果给定变量未声明，则会创建一个该名称变量并将读取到的值存入
    - 如果给定变量已声明，则会覆盖已有值
    - 输入 数字时直接输入即可
    - 输入 字符串需用`"` 括起
    - 如果读取时出错，会返回空字符串
    - eg
        - `get $a;`
- send
    - 将给定表达式的值输出至指定数据源中

        - 数据源设置见 参数 章节
        - 数据源说明见 功能 章节
    - eg
        - `send "hello" & " world";`
- load
    - 用于在程序中以给定key装载变量表
        - 装载后的变量可以直接在后续程序中使用
    - key 可以为变量或直接提供值
    - 同名变量会选择装载表中数据作为结果数据
    - eg
        - `load "Ben";`
- save
    - 用于在程序中把当前变量表存储至以key为标记的文件中
    - key 可以为变量或直接提供值
    - eg
        - `save "Ben";`
- exit
    - 用于设置 exit flag 为 1
    - 具体用法介绍见 功能 章节
    - eg
        - `exit;`
- exec
    - 用于执行给定 shell 指令
    - 指令执行输出会输出至指定数据源
    - 不支持与 shell 指令进行后续交互

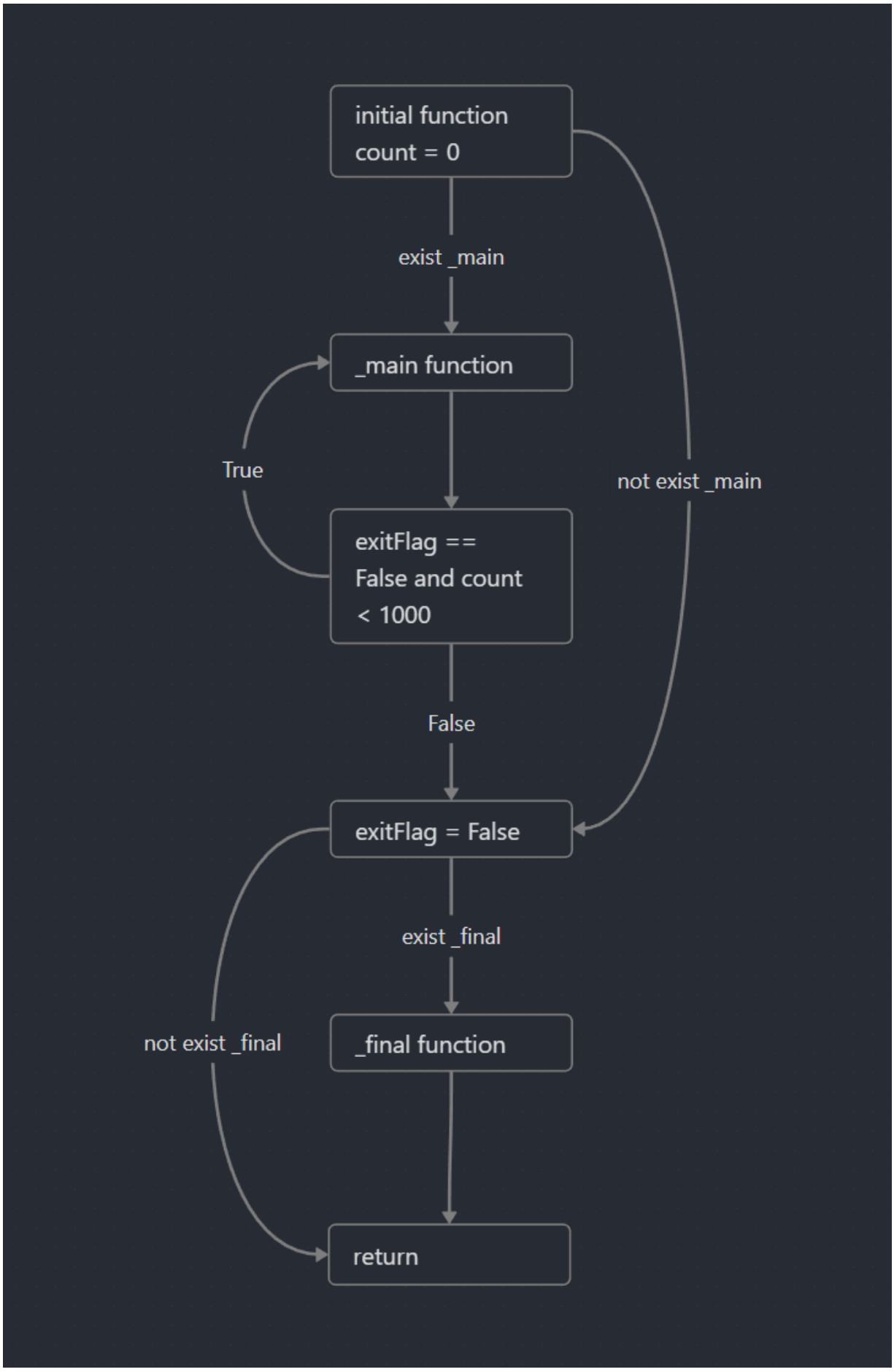
    - eg
        - `exec "echo hello";`
- 注释
    - C语言风格多行注释
    - 注释只允许在 block 之中，不支持在block外
    - 允许在sentence前，sentence后，不允许在sentence中
    - 允许多行
    - eg
        - `{ /*comment*/ send "hello"; }`
- 各语句 BNF 范式如下

```
<funcName> ::= "_" <alphanums>
<varName> ::= "$" <alphanums>
<string> ::= '"' <chars> '"'
<number> ::= <nums> ["." <nums>]
<var> ::= <number> | <varName> | <string>
<operators> ::= "*" | "/" | "+" | "-" | "==" | "in" |
<exp> ::= <var> <operators> <var> | <var>
<sentence> ::= <loop> | <condition> | <assignment> |
<block> ::= "{" <sentence>* "}"
<execute> ::= "exec" <string> ";"
<func> ::= "func" <funcName> <block> ";"
<loop> ::= "loop" <nums> "times" (<funcName> | <block
<condition> ::= "if" <exp> "then" (<funcName> | <bloc
<assignment> ::= <varName> "=" <exp> ";" | <varName>
<get> ::= "get" <varName> ";"
<send> ::= "send" <exp> ";"
<load> ::= "load" <var> ";"
<exitSentence> ::= "exit" ";"
```

```
<comment> ::= "/*" <any>* "*/"
<sentence> ::= [<comment>] <sentence> [<comment>]
<grammar> ::= <block>
```

功能

- Interpret流程



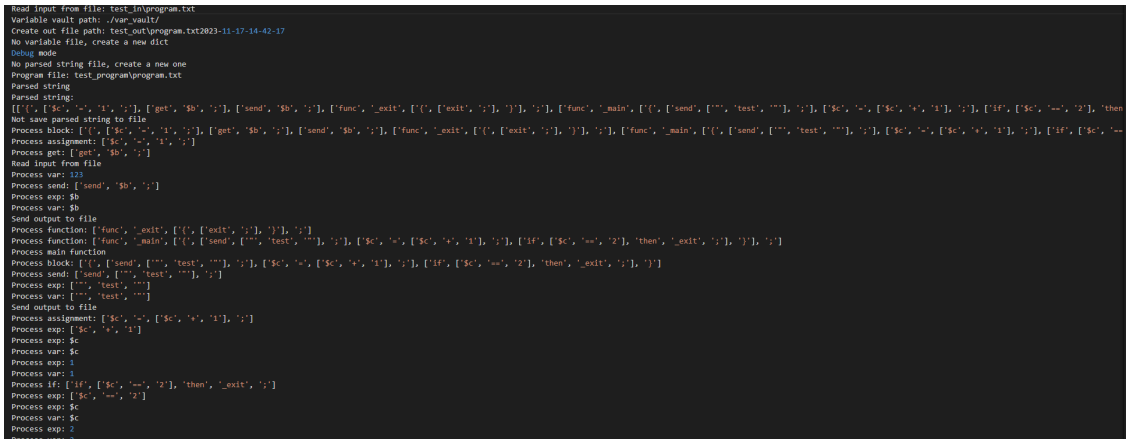
- initial function

    - program中所写最外层block
    - 不保存至funclist中
    - 在执行过程中必定会执行
    - 执行完成之后进行 `_main` 和 `_final` 部分处理
- exitFlag
    - 初始化时为False，执行完成循环`_main` 部分后重置为 False
    - 当exitFlag为True时，跳过后续所有sentence执行
- main
    - 需要在initial function 中显式声明该函数，名称需要为 `_main`
    - 如果在initial function中没有声明, 则跳过该部分
    - 在执行完initial function后会重复执行, 最多1000次
        - 执行1000次后程序会报错退出
    - 当exitFlag为1时跳过后续main中所有代码的执行，并停止循环
    - exit; 语句实现对exit Flag置1
        - exit; 语句没有位置限制，等同于普通sentence
- final
    - 需要在initial function 中显式声明该函数，名称需要为 `_final`
    - 如果在initial function中没有声明, 则跳过该部分
    - 在执行完initial function 和 main function 之后执行
    - 在执行final function 前会重置exit Flag
    - final function 中语句可以通过 exit; 跳过后续代码执行
- 存储与加载parse后的string

    - 需要在调用translator 时填写参数 `--save_parsed`
    - 文件格式为 .pkl, 保存的内容为通过pyparsing处理过的ParsedElement, 通过pickle保存为二进制文件
    - 当 `--load_parsed` 参数不为空时，IOManager 会优先从指明文件中读取ParsedStr 而不是读取Program 后再进行parse
- 获取原始程序
    - 通过设置 `--program_file` 指定程序文件夹
    - 当 `--program_file` 不为空时从文件加载程序，否则从stdin读取程序
    - 当 `--save_parsed` 不为空时直接加载 ParsedElement，跳过加载原始程序步骤
- 存储与加载变量表
    - 从文件直接读取
        - 需要在调用translator 时填写参数 `--save_file` 或 `--load_file`指明保存或加载的文件名
        - 需要为 .json 格式，变量名命名需要符合语法规则
        - 在程序进入Interpret 阶段前装载
    - 从key读取
        - 需要在调用translator 时填写参数 `--save_key`
        - 依照key保存的变量表均存储于`./var_vault/` 文件夹中，命名格式为`var_{key}.json`
            - 如指定参数为 `--load_key Peter` ,则从 `./var_vault/var_Peter.json`中读取数据
        - 可以通过`--var_vault` 参数指定变量表仓库位置
        - 在程序进入Interpret 阶段前装载

    - 在程序中通过 load 语句装载
        - 需要指明 key
        - 依照key保存的变量表均存储于`./{var_vault}/`文件夹中，命名格式为`var_{key}.json`
            - 如指定参数为 `--load_key Peter` ,则从`./var_vault/var_Peter.json`中读取数据
        - 可以通过`--var_vault` 参数指定变量表仓库位置
        - 在程序执行到load语句时装载
        - 如有同名变量，会覆盖已有属性
    - 保存变量表
        - 在程序执行完 final function 之后执行
        - `--load_key`指明保存的key 或 `--load_parsed`指明保存的文件名
        - 如有同名文件，会覆盖已有文件
- 数据源
    - 通过`--mode` 设置web，std，file 三种模式
    - 使用std 模式时不需要设置额外参数
    - 使用 web 模式时需要使用 `--url` 设置目的url
        - 本程序中使用 get 从目的 url 获取输入
        - 使用 post 将输出发送至 目的 url
    - 使用 file模式时需要设置 输入文件夹与 输出文件夹
        - `--input` 设置input的文件名
        - `--output` 设置 output 的文件名
    - 输入和输出模式一致，不允许输入和输出使用不同模式
- debug模式

- 使用 `--debug` 参数设置为debug模式
- 会输出程序执行时的详细信息
- 不开启 debug模式只会输出必要提示信息以及输入输出相关信息
- 开启debug模式



- 关闭debug模式

```
Hint: You may forget to set the mode. If you want to use the file mode, please set the mode to file. Current mode is  std .
Please input the code, end with Ctrl+Z on Windows or Ctrl-D on Unix-like system:
{
    exec "echo Hello";
    func _load {
        send "please input your name";
        get $name;
        load $name;
    };

    _load;
    send "Welcome Mr/Miss " & $name & ", what can I help you?";

    func _main {
        get $cmd;
        if "enquiry" in $cmd then _find
        elif "complain" in $cmd then _complain
        elif "exit" in $cmd then _exit
        else {
            send "sorry, we don't support this function yet";
        };
    };

    func _final {
        send "thank you for your use, goodbye";
    };

    func _exit {
        exit;
    };

    func _find {
        send "The fee in the next 3 months are";
        loop 3 times {
            send $fee & " dollars";
            $fee = $fee + 1.5 ;
        };
    };

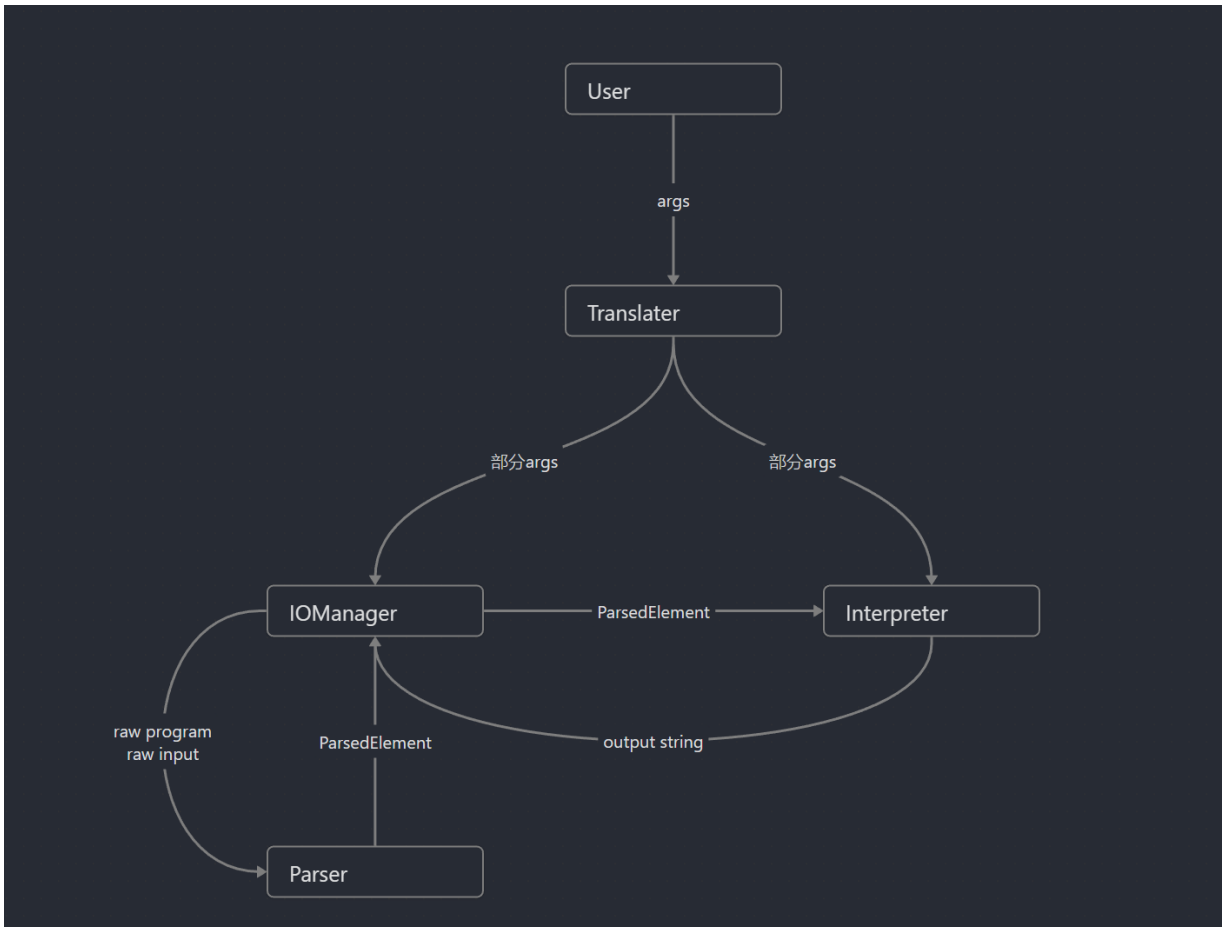
    func _complain {
        get $complain;
        send "Sorry, your complain has been saved.";
    };
}
^Z
Hello
please input your name
<<< "ben"
Welcome Mr/Miss ben, what can I help you?
<<< "exit"
thank you for your use, goodbye
```

- 异常处理
  - 本程序依靠自定义异常和使用库大风自带异常提供了丰富的报错信息以便于查询出错
  - 自定义异常见 `Exception.py`
  - 程序语法错误异常 由pyparsing自带异常信息给出提示，可以指明问题所在行列
  - 程序语义错误异常 由自定义异常实现，在`Interpreter.py`模块中会输出相应异常信息以及问题语句
  - IO设置错误 由自带文件异常结合额外信息辅助定位问题所在位置

接口

程序间接口

- 数据流图



- 为最小化程序间接口，大部分配置在初始化时通过args进行配置，后续调用过程中依照配置进行数据交互，仅传递必要数据，实现高内聚，低耦合。
- IOManager读取到的字符串均需要通过Parser处理之后传递给Interpreter
    - 包括程序与后续的input均需处理
- Interpreter 输出的字符串构建完成后传递给IOManager, IOManager 根据设置选择相应的输出位置

人机接口

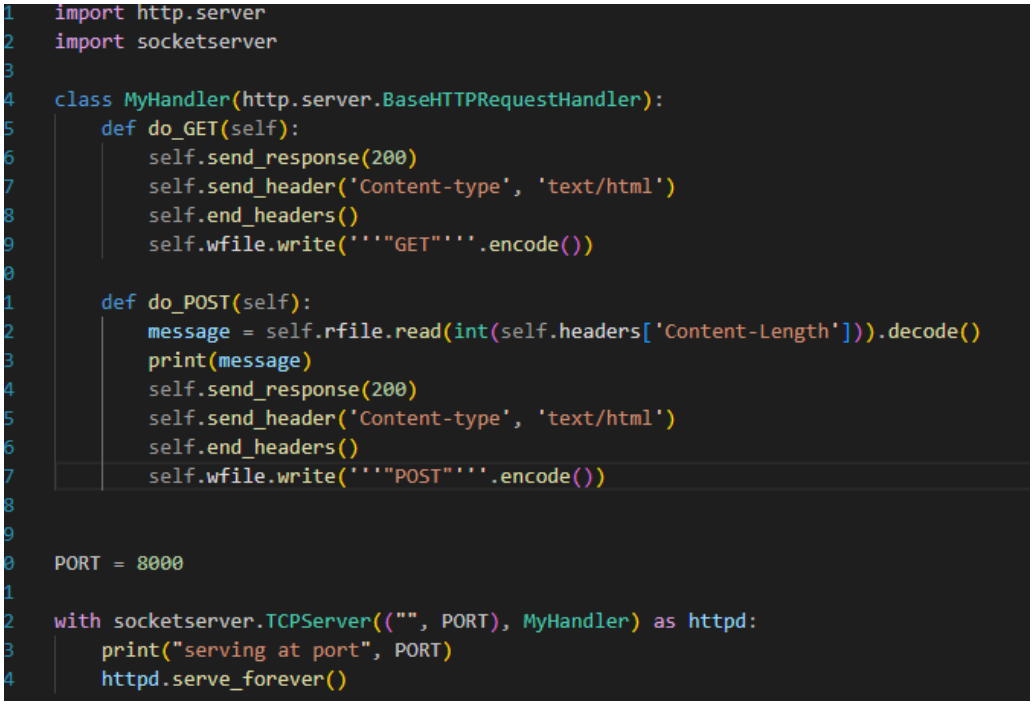
- 程序和用户之间通过调用程序时填写的的args作为配置，实现对IO模式，程序来源，变量表初始化等进行设置，同时便于用脚本对程序进行调用，实现批量处理与自动化
    - 在调用程序时输入 --help可以查看可用args和解释

- 为实现最小化接口，用户与程序的交互接口仅存在于初始化设置阶段与后续IO数据交换阶段，对用户屏蔽了内部实现细节。
- 用户可直接接触到的类只有 `Translater`，最大程度进行了封装
- 程序提供了普通模式和Debug模式两种模式。
  - 普通模式仅输出IO 数据
    - 输入数据前会显示 `<<<`
    - ```
      Hello
      please input your name
      <<<"ben"
      Welcome Mr/Miss ben, what can I help you?
      <<<"abc enquiry def"
      The fee in the next 3 months are
      5dollars
      6.5dollars
      8.0dollars
      <<<"complain"
      <<<"complain 123"
      Sorry, your complain has been saved.
      <<<"exit"
      thank you for your use, goodbye
      ```
  - Debug模式会详细输出程序执行步骤及重要中间结果
    - ```
      Process loop: ['loop', '3', 'times', ['{', ['send', ['$fee', '&', ['"', 'dollars', '"']], ';'], ['$fee', '=', ['$fee', '+', '1.5'], ';'], '}'], ';']
      Process block: ['{', ['send', ['$fee', '&', ['"', 'dollars', '"']], ';'], ['$fee', '=', ['$fee', '+', '1.5'], ';'], '}']
      Process send: ['send', ['$fee', '&', ['"', 'dollars', '"']], ';']
      Process exp: ['$fee', '&', ['"', 'dollars', '"']]
      Process exp: $fee
      Process var: $fee
      Process exp: ['"', 'dollars', '"']
      Process var: ['"', 'dollars', '"']
      Send output to file
      Process assignment: ['$fee', '=', ['$fee', '+', '1.5'], ';']
      Process exp: ['$fee', '+', '1.5']
      Process exp: $fee
      Process var: $fee
      Process exp: 1.5
      Process var: 1.5
      ```
- 程序提供了异常信息，在捕获到异常信息之后会指明出错模块，输出异常信息类型并退出程序
  - ```
    Process assignment: ['$c', '=', '1', ';']
    Process send: ['send', '$b', ';']
    Error: Variable $b does not exist
    Not save variable to file
    Not save output to file
    ```
  - Parser模块使用了Pyparsing自带的错误信息提示
  - Interpreter模块的Exception具体见Exception.py
    ```

    - IOManager模块结合File和HTTP自带的 Exception和额外信息提示指明哪个模块出错
- 可以通过shell脚本实现log保存

测试

- Parser，IOManager，Interpreter， Web均有stub
    - 分别模拟 Parser模块，IOManager模块， Interpreter模块和io mode中的 Web服务器
    - 由于本程序IO接口除web外为std与file，故不单独设置stub
    - webstub示意



```python
import http.server
import socketserver

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write('''"GET"'''.encode())

    def do_POST(self):
        message = self.rfile.read(int(self.headers['Content-Length'])).decode()
        print(message)
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write('''"POST"'''.encode())


PORT = 8000

with socketserver.TCPServer(("", PORT), MyHandler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

- unittest + python脚本实现自动测试
    - unitest 完成Parser和IOManager单个函数测试
        - 见`TestParser.py`与`TestIOManager.py`
        - `TestIOManager.py`需在vscode的debug模式下执行
        - `TestParser.py` 示意

```
import unittest
from Parser import Parser

class TestParser(unittest.TestCase):
    def setUp(self):
        self.parser = Parser()

    def test_execute(self):
        result = self.parser.grammarParse('{exec "echo hello world";}')
        expected = [['{', 'exec', ['"', 'echo hello world', '"'], ';'], '}']]
        self.assertEqual(result, expected)

    def test_func(self):
        result = self.parser.grammarParse('''{func _myFunc {exec "echo hello world";};}''')
        expected = [['{', 'func', '_myFunc', ['{', ['exec', ['"', 'echo hello world', '"'], ';'], '}'], ';'], '}']]
        self.assertEqual(result, expected)

    def test_loop(self):
        result = self.parser.grammarParse('{loop 3 times _myFunc;}')
        expected = [['{', ['loop', '3', 'times', '_myFunc', ';'], '}']]
        self.assertEqual(result, expected)

    def test_condition(self):
        result = self.parser.grammarParse('{if $x == 5 then _myFunc elif $x == 6 then _myOtherFunc else _myThirdFunc;}')
        expected = [['{', ['if', ['$x', '==', '5'], 'then', '_myFunc', 'elif', ['$x', '==', '6'], 'then', '_myOtherFunc', 'else', '_myThirdFunc', ';'], '}']]
        self.assertEqual(result, expected)

    def test_condition2(self):
        result = self.parser.grammarParse('{if $x == 5 then _myFunc;}')
        expected = [['{', ['if', ['$x', '==', '5'], 'then', '_myFunc', ';'], '}']]
        self.assertEqual(result, expected)

    def test_condition3(self):
        result = self.parser.grammarParse('{if $x == 5 then _myFunc elif $x == 6 then _myOtherFunc;}')
        expected = [['{', ['if', ['$x', '==', '5'], 'then', '_myFunc', 'elif', ['$x', '==', '6'], 'then', '_myOtherFunc', ';'], '}']]
        self.assertEqual(result, expected)

    def test_condition4(self):
        result = self.parser.grammarParse('{if $x == 5 then _myFunc else _myOtherFunc;}')
```

- 测试示意

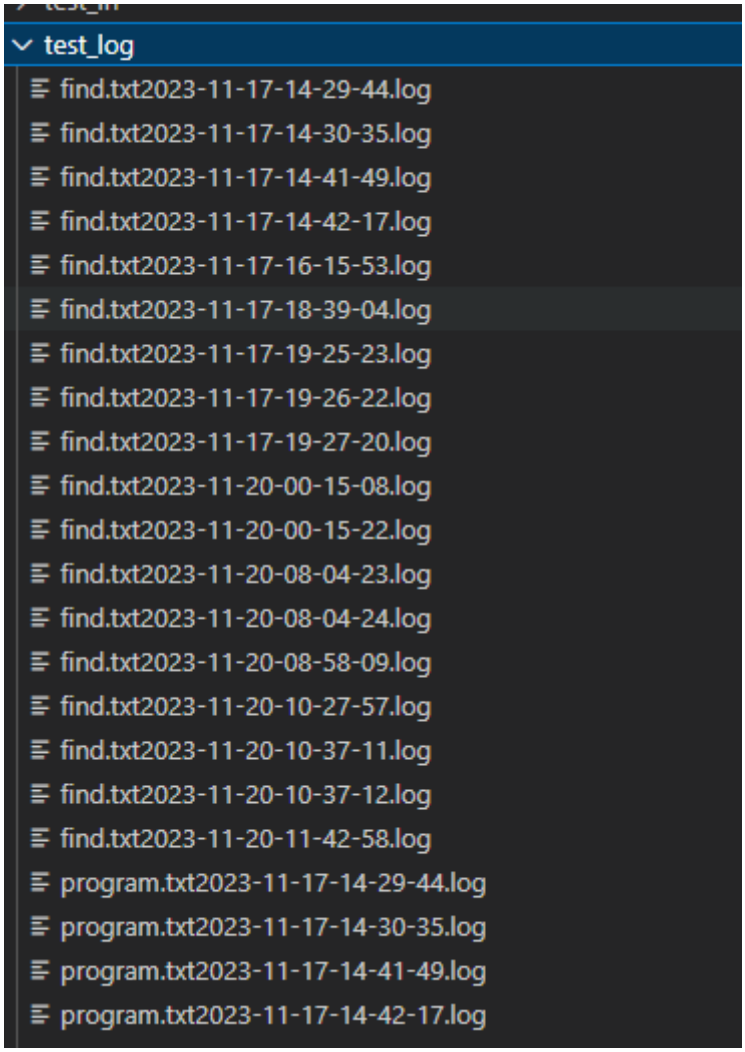
```
PS D:\Code\DSL> & C:/Users/15570/.conda/envs/DSL/python.exe d:/Code/DSL/TestParser.py
......................
----------------------------------------------------------------------
Ran 22 tests in 0.208s

OK
```

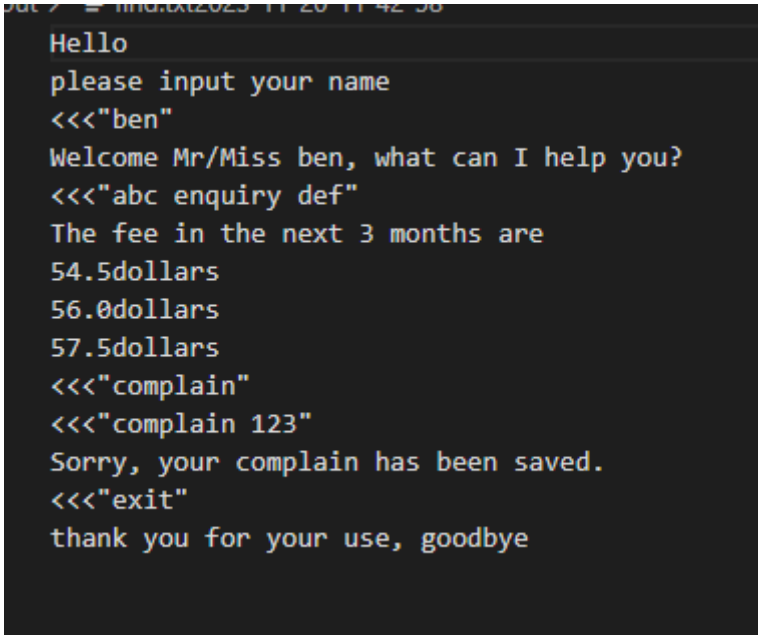
- python脚本实现全流程测试
  - 具体内容见`test.py`

```
import os
import subprocess
from datetime import datetime

if not os.path.isdir('test_log'):
    os.mkdir('test_log')

if not os.path.isdir('test_out'):
    os.mkdir('test_out')

if not os.path.isdir('test_in'):
    os.mkdir('test_in')
    print('Please put the test input file in test_in')
    exit()

if not os.path.isdir('test_program'):
    os.mkdir('test_program')
    print('Please put the test program file in test_program')
    exit()


for file_name in os.listdir('test_program'):
    file_path = os.path.join('test_program', file_name)
    in_path = os.path.join('test_in', file_name  )
    out_path = os.path.join('test_out', file_name + datetime.now().strftime('%Y-%m-%d-%H-%M-%S') )


    if os.path.isfile(file_path):
        log_file = os.path.join('test_log', (file_name + datetime.now().strftime('%Y-%m-%d-%H-%M-%S') + '.log' ))
        with open(log_file, 'w') as f:
            subprocess.run(['python', 'main.py', '--program_file', file_path, '-i', in_path, '-o', out_path, '-d', '-m', 'file'], stdout=f, stderr=f)
```

  - 基于file mode实现测试
  - 支持单次多个program测试
    - 设置了两个不同的program，其中`find.txt`涵盖了全部语句与语法糖，并且模拟了客服应答场景
  - 需要设置program和input数据
    - 将program放于test_program文件夹下
    - 将input放于test_in文件夹下

    - 对应的program和input名字需要相同
- input示意
    -
```
"ben"
"abc enquiry def"
"complain"
"complain 123"
"exit"
```
- program示意
    -
```
{
    exec "echo Hello";
    func _load {
        send "please input your name";
        get $name;
        load $name;
    };

    _load;
    send "Welcome Mr/Miss " & $name & ", what can I help you?";

    func _main {
        get $cmd;
        if "enquiry" in $cmd then _find
        elif "complain" in $cmd then _complain
        elif "exit" in $cmd then _exit
        else {
            send "sorry, we don't support this function yet";
        };
    };

    func _final {
        save $name;
        send "thank you for your use, goodbye";
    };

    func _exit {
        exit;
    };

    func _find {
        send "The fee in the next 3 months are";
        loop 3 times {
            send $fee & " dollars";
            $fee = $fee + 1.5 ;
        };
    };

    func _complain {
        get $complain;
        send "Sorry, your complain has been saved.";
    };
}
```
- log示意
    - 内容

- 
Read input from file: test_in\find.txt
Variable vault path: ./var_vault/
Create out file path: test_out\find.txt2023-11-20-11-42-58
No variable file, create a new dict
Debug mode
No parsed string file, create a new one
Program file: test_program\find.txt
Parsed string
Parsed string:
Not save parsed string to file
Process Sentence: ['exec', ['"', 'echo Hello', '"'], ';']
Process var: ['"', 'echo Hello', '"']
Process var: Hello
Send output to file
Process Sentence: ['get', '$name', ';']
Read input from file
Process var: ['"', 'ben', '"']
Process Sentence: ['load', '$name', ';']
Process var: $name
Find var dict from key: ben
- 目录
- 
test_log
find.txt2023-11-17-14-29-44.log
find.txt2023-11-17-14-30-35.log
find.txt2023-11-17-14-41-49.log
find.txt2023-11-17-14-42-17.log
find.txt2023-11-17-16-15-53.log
find.txt2023-11-17-18-39-04.log
find.txt2023-11-17-19-25-23.log
find.txt2023-11-17-19-26-22.log
find.txt2023-11-17-19-27-20.log
find.txt2023-11-20-00-15-08.log
find.txt2023-11-20-00-15-22.log
find.txt2023-11-20-08-04-23.log
find.txt2023-11-20-08-04-24.log
find.txt2023-11-20-08-58-09.log
find.txt2023-11-20-10-27-57.log
find.txt2023-11-20-10-37-11.log
find.txt2023-11-20-10-37-12.log
find.txt2023-11-20-11-42-58.log
program.txt2023-11-17-14-29-44.log
program.txt2023-11-17-14-30-35.log
program.txt2023-11-17-14-41-49.log
program.txt2023-11-17-14-42-17.log
- output示意

- 
```
Hello
please input your name
<<<"ben"
Welcome Mr/Miss ben, what can I help you?
<<<"abc enquiry def"
The fee in the next 3 months are
54.5dollars
56.0dollars
57.5dollars
<<<"complain"
<<<"complain 123"
Sorry, your complain has been saved.
<<<"exit"
thank you for your use, goodbye
```

结论

在软件开发过程中，除了关注模块设计、接口设计等功能性方面的工作外，还需要着重考虑测试脚本的编写。为了促进团队协作，制定清晰的接口规范、统一的代码风格以及详尽的文档至关重要。此外，为了支持各个模块的并行开发，测试桩也是不可或缺的一环。通过这些举措，我们可以更好地确保软件质量，提高开发效率，同时降低后续维护和调试的难度。