DSL语言客服系统

项目概述

> 🥇 使用DSL脚本解释语言，描述客服机器人，使能够达到应对客户询问的结果

运行环境

1. IntelliJ IDEA 2021.2.3 (Ultimate Edition)
2. VSCODE: 1.84.2 (user setup)

使用框架

1. 前端 : vue
2. 后端 : springboot

应答机器人功能描述

匹配回应

> 可以根据设计好的语法返回对应的相应

交互响应

> 根据用户输入返回交互式响应数据

问好响应

> 自动触发的响应

建议给予

> 根据当下用户所处状态给出输入建议

默认处理

> 找不到匹配的响应结果返回时的响应

超时处理

> 用户长时间处于一个状态下，机器人返回的响应

应答机器人语法设计

范式表示

```
<开始> ::= [<变量> | <状态>]
<变量> ::= <IDENTIFIER>=<结果>
<状态> ::= [KEYWORD]<IDENTIFIER>{<逻辑>}
<逻辑> ::= <MATCH>|<输入>:return <结果> [goto <IDENTIFIER>]
<逻辑> ::= KEYWORD [<NUMBER>] :return <结果> [goto <IDENTIFIER>]
<输入> ::= [[<STRING>|<IDENTIFIER>]....]
<结果> ::= [<STRING>|<NUMBER>|<FUNCTION>][+<结果>]
<函数> ::= <FUNCTION>([<参数>])
<参数> ::= $<IDENTIFIER>=<STRING> | $<IDENTIFIER>=<NUMBER> | $<IDENTIFIER>=
<IDENTIFIER>
```

结果

什么是结果

在该系统中，结果被定义为，基础信息元单位，不可再拆分的最小信息元，比如一段字符串，一个数字，一个函数求解值等！结果中包含字符串等静态结果，也有函数结果等需要外部输入的结果！

结果的运算

结果具有可连接性，通过运算符 + 可以使得两个求解后的结果链接形成更大的结果!

函数结果

系统提供了内置函数实现复杂功能的实现，函数结果可以用来实现复杂结果的求解！函数名是系统中的关键字，参数的开头以$引导，其他的参数可以使用输入匹配得到的结果!

```
Query($tb="telephone",$id=input_params)
```

变量

变量是对于一个结果的代称，您可以使用如下的方式来定义一个变量，来减少重复的结果定义!

```
标识符 = 字符串 | 数字 | 函数值 | 变量
```

PS:

1. 变量的标识符和状态的标识符是可区分的，也就是说，同样的标识符可以同时用于变量名和状态名，但这样不被推荐，会触发警告!

```
z = 1
z {
    xxx
}
```

2. 变量允许多次赋值，即允许同一个变量在不同的位置被重新赋值，但是这样也不被推荐，您可以重新使用别的标识符来赋值!

状态

状态定义了智能机器人的工作状态，在不同的状态下，机器人可以根据外部的输入返回响应的结果!状态拥有一个关键词global进行修饰，拥有该关键词的将会拥有更高的应答权限!

状态的基本定义如下

```
global s {
    逻辑一
    逻辑二

    。。。
}

main 包含 新东 123 next

main 默认 我没听懂 next

next 包含 新东 456 NULL

mian:
    包含 新东 123 > next
    准确 新东 567
    默认 456
next

a {
    逻辑一
    逻辑二
    。。。
}
```

PS:

1. 系统允许不可达状态的使用，但是会产生警告提醒使用者，同时这种行为是不建议的！
2. 在某种情况下，一种状态可能会成为僵尸状态，即到达后不可返回，系统会对这种状态进行提示并且产生警告，这种行为不建议出现

逻辑

在状态中，其主体通过一条一条逻辑构成，逻辑也是系统的核心部分，智能客服根据逻辑的定义来对用户的询问作出应答，是非常重要的部分！

基本定义如下

```
Match : return Result goto State
Match : return Result
```

Begin 逻辑

begin逻辑是在状态到达时的触发语句，无需客户询问，自动发送，通常用来问好！其中begin是定义的关键字！

```
begin :  return Result
```

PS:

begin逻辑后不建议使用goto语句，造成不必要的状态定义！

Default 逻辑

default逻辑是在当前状态没有其他可以匹配逻辑时触发的兜底逻辑，通常用于表示无法处理请求的回应，其中default是关键字！

```
default : return Result
```

PS:

default逻辑后不建议使用goto语句，造成不必要的状态定义！

Wait 逻辑

Wait逻辑是在当前状态长时间无应答的情况下，系统自动发送的相应，主要用于询问客户是否还在服务中等功能，wait是关键字！其中wait后的数字是以秒为单位的！

```
wait 100 : return Result
```

匹配逻辑

匹配逻辑是通过对客户输入的文字进行正则匹配，返回相应的回答，匹配模式分为精确，模糊，正则等三种方式，用不同符号表示！

```
'精确匹配' : return Result
<模糊匹配> : reutrn Result
`正则匹配` : return Result
```

PS `正则匹配` 提供了让用户自己定义匹配方式的机会，比如 `.*你.*干什么` 就可以匹配 '你会干什么' 和 '你能干什么'这多种情况，具体利弊需要进一步权衡

输入逻辑

系统提供输入机制，不再局限于听取用户的请求做出固定的回答，可以使用输入逻辑来提取用户输入中的信息，做出更丰富的相应！

提取出的参数params1可以用于结果中的输入变量，用于函数参数的赋值以及结果的表示！

```
1 ["Regex1".params1."Regex2"] : return Result
```

PS:

1. 尽管["Regex1".params1]的输入模式也可以提取出params1，但是不建议，因为这样不满足两个字符串夹逼的方式，提取结果肯存在误差

PS

1. 同一种状态中可能存在多个相同的匹配逻辑，这种情况下一般后一种不可达，优先返回前面的匹配结果，所以系统会给出警告！
2. default，begin，wait这三种关键字开头的逻辑，不允许重复出现，重复的逻辑会被认为是一种错误！
3. Goto 后是本身状态会产生警告，因为这是无意义的！
4. 如果非全局状态中存在default逻辑，但是不存在goto跳转，可能会产生僵尸状态，无法跳转到其他状态！

注释

指出单行注释

```
1 // 单行注释!
```

应答机器人模块设计

响应生成器

> 依据当下状态以及输入字符串生成响应字符串

```
public static Result ResponseGenerator(String inputStr) {
    // 异常判断
    List<TransferNode>  TransferList = robotDependency.getTransMap().get(state).getTransferList();
    // 遍历转移列表，找到符合匹配的转移节点
    for(int i = 0;i < TransferList.size();i++){
        // 比对Condition条件
        Result res = null;
        if((res = ConditionProcessor(condition,inputStr)) != null){
            // 如果匹配成功，那么就返回对应的响应
            // 判断是否存在状态转移
            if(TransferList.get(i).getTargetState() != -1){
                StateChangeProcessor(TransferList.get(i).getTargetState());//状态转移器
            }
            // 返回响应结果
            return 响应结果; // 调用结果解析器
        }
    }

    // 查看默认响应
    if(defaultResultMap.containsKey(state)){
        // 如果存在默认响应 切换状态
        if(defaultResultMap.get(state).getTargetState() != -1){
            StateChangeProcessor(defaultResultMap.get(state).getTargetState());
        }
        return 默认响应结果;
    }

    // 没有相应成功的 从全局状态中寻找
    List<Integer> globalState = robotDependency.getGlobalState();
    for(int i = 0;i < globalState.size();i++){
        // 查找全局状态
    }
    return null;
}
```

状态转移器

> 根据逻辑进行状态转移,同时对问好响应等附加逻辑做处理

```java
private static Result StateChangeProcessor(Integer targetState){
        if(targetState != -1){
            // 更改状态
            state = targetState;
            resetWaitResponse(); // 状态更新也要更新等待响应模块
            // 建议相关
            try {
                // 发送 suggestion
            } catch (IOException e) {
                e.printStackTrace();
            }
            // Say Hello 相关
            if(robotDependency.getHelloMap().containsKey(state)){
                try {
                    // 主动发送Hello
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return Result.success();
        }else{
            return Result.error();
        }
    }
```

建议生成器

> 根据现在的状态生成建议的语句，提供更友好的交互方式

1. 针对精准匹配：提供建议为匹配字符本身
2. 针对模糊匹配：返回"您可以输入xxx"
3. 针对正则匹配：返回正则表达式供参考
4. 针对交互匹配：返回输入范例,帮助输入

```java
public static List<Suggestion> SuggestionGenerator() {
    // 遍历转移列表，生成建议
    for(int i = 0;i < TransferList.size();i++){
        // 比对Condition条件
        Condition condition = TransferList.get(i).getCondition();
        Suggestion suggestion = new Suggestion();
        switch (condition.getType()){
            case ConditionConstant.JUDGE_EXACT -> {
                // 对精确匹配的建议
                break;
```

```
            }
            case ConditionConstant.JUDGE_CONTAIN -> {
                // 对模糊匹配的建议
                break;
            }
            case ConditionConstant.JUDGE_REGEX -> {
                // 对正则的建议
                break;
            }
            case ConditionConstant.INPUT -> {
                // 对输入的建议
                break;
            }
        }
    }
    return suggestionList;
}
```

超时响应器

> 超时响应需要websocket的支持,通过websocket而不是请求获得消息和返回消息

```
public static Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        while(true) {
            try {
                for(tick = 0;tick <waitTime;tick++){
                    // 睡眠一秒
                    Thread.sleep(1000);
                }
                // 使用websocket发送超时响应,主动连接
                Thread.sleep(SystemConstant.WAIT_TIME);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
});
```

默认响应器

> 作为一个普通的额外结构,对于没有响应的输入生成响应

结果解析器

> 查询结果字典，结合输入字符，调用各种内置参数，以及运算符号得到字符串结果！

```
public static String ResultGenerator(Integer resultID, Map inputs) {
    CompileResult result = resultDictionary.getCompileResult(resultID);
    if(Objects.equals(result.getType(),
SemanticAnalysisConstant.CONSTANT_RESULT)){
        // 常量结果 直接返回
        return result.getValue();
    }else if(Objects.equals(result.getType(),
SemanticAnalysisConstant.COMPLEX_RESULT)) { // 复合型结果 需要遍历增加
        // 复合型结果需要递归调用
        for (int i = 0; i < resultIDList.size(); i++) {
            resultStr += " " + ResultGenerator(resultIDList.get(i), inputs);
        }
        return  resultStr;
    }else if(Objects.equals(result.getType(),
SemanticAnalysisConstant.FUNCTION_RESULT)){
        // 需要导入函数参数进行赋值 并且调用 data指的都是参数和结果指向列表
        Map<String,Integer> resultParams = (Map<String, Integer>)
result.getData();
        // 提取参数 调用函数
        return FunctionCaller.Caller(result.getValue(),params);
    }else if(Objects.equals(result.getType(),
SemanticAnalysisConstant.INPUT_VARIABLE_RESULT)){
        // 需要导入的参数进行赋值处理，参数在params中，这个节点的vaule是变量名称
    }
    else{
        LOG.WARNING("结果解析出现未知类型");
    }
    return "";
}
```

条件判断器

> 针对一个condition 和 input 判断是否匹配

```java
public static Result ConditionProcessor(Condition condition, String inputStr) {
    // 判断异常
    if(非交互式匹配){
        // 判断模式串
        if(inputStr.matches(condition.getPattern().pattern())){
            return Result.success();
        }else{
            return null;
        }
    }else if(交互式匹配){
        Matcher matcher = condition.getPattern().matcher(inputStr);
        // 匹配 提取参数
        if(matcher.find()){
            Map<String,String> params = new HashMap<>();
            for(int i = 0;i < condition.getParams().size();i++){
                params.put(condition.getParams().get(i),matcher.group(i+1));
            }
            return Result.success(params);
        }else{
            return null;
        }
    }else{
        LOG.WARNING("条件类型错误");
    }
    return null;
}
```

客服机器人编译处理

1. 词法分析

📌 词法分析的目的是将输入的源代码转换为词法单元（tokens），并为后续的语法分析和语义分析提供有组织的输入。词法分析器扫描源代码，识别和提取出具有独立含义的词法单元，如标识符、关键字、运算符、常量等，并将它们组织成有意义的结构。

词法分析器生成的结构通常是一个包含词法单元及其相关信息的数据结构，通常是一个抽象语法树（Abstract Syntax Tree，AST）或标记流（Token Stream）。这些结构可以被后续的解释器、编译器或其他语义分析阶段使用。

词法分析的结构可以根据具体的实现方式和需求而有所不同。一种常见的结构是标记流，它是一个按顺序排列的词法单元序列，每个词法单元包含词法单元类型（如标识符、关键字、运算符等）和对应的文本值。标记流提供了一个线性的、有序的视图，便于后续处理阶段对代码进行逐个词法单元的处理。

另一种常见的结构是抽象语法树（AST），它是一个树状的结构，以语法规则为基础组织起来。抽象语法树反映了源代码的结构和层次关系，每个节点表示一个语法单元，如表达式、语句或声明。通过构建抽象语法树，可以更方便地进行语法分析、语义分析和生成中间代码等操作。

总而言之，词法分析的目标是将源代码转换为有意义的词法单元，生成相应的数据结构，以便后续的语法分析和语义分析能够对代码进行进一步处理和分析。

词法定义

关键字 - KEYWORD

```
global
return
then
goto
其他内置函数
```

标识符 - IDENTIFIER

分隔符 - DELIMITER

```
{}
.
```

字符串 - STRING

```
“”
```

数字 - NUMBER

```
整数
```

匹配符 - MATCHMAKER（尽可能保证不一致）

```
''
```

```
``
<>
```

输入符 - INPUTFLAGER

```
[]
```

运算符 - OPERATOR

```
+
=
```

默认分隔符 - DEFAULT

```
\n
space
```

词法错误 - CHARERROR

> 对于错误的词法输入

分析函数架构 - 自动机

```
while(pointer < codeLength) {
    // 识别token
    if(stateMap.get(code.charAt(pointer)) == null){
        // 判定非法词法
    }

    switch (stateMap.get(code.charAt(pointer))) {
        case TokensConstant.IDENTIFIER:
            // 判定关键词和标识符
            break;
        case TokensConstant.NUMBER:
            // number
            break;
        case TokensConstant.STRING:
```

```
            // string
            break;
        case TokensConstant.OPERATOR:
            // operator
            break;
        case TokensConstant.DELIMITER:
            // delimiter
            break;
        case TokensConstant.MATCHMAKER:
            // matchmaker
            break;
        case TokensConstant.PARAMETER:
            // parameter
            break;
        case TokensConstant.INPUT_FLAG:
            break;
        case TokensConstant.COMMENT:
            // comment 识别到换行符
            break;
        case TokensConstant.DEFAULT:
            if(code.charAt(pointer) == '\n'){
                // 统计行数
            }
            pointer++;
            break;
        case TokensConstant.ERROR:
            throw new
LexiAnalyseException(CompileErrorConstant.ILLEGAL_CHAR,lineIndex);
        default:
            throw new
LexiAnalyseException(CompileErrorConstant.ILLEGAL_CODE,lineIndex);
    }
}
```

生成结构

```
public class Tokens {
    private List<Token> stream;
    private Map<Integer,List<String>> map;
}
```

Token Stream

> 提供有序的输入，可视化如图

```
Token{type=IDENTIFIER, token='my_telephone', lineIndex=2}
Token{type=OPERATOR, token='=', lineIndex=2}
Token{type=NUMBER, token='15665777392', lineIndex=2}
Token{type=IDENTIFIER, token='What_I_can_do', lineIndex=3}
Token{type=OPERATOR, token='=', lineIndex=3}
```

Token Map

> 提供Token 各种种类的映射关系

2. 语法分析

🎨 语法分析树的好处如下：

1. 结构清晰：语法分析树以树状结构的形式展示了程序的语法组织和嵌套关系，使得程序的结构更加清晰可见。通过遍历和分析语法分析树，可以深入了解程序的语法细节和组织结构。
2. 错误检测：语法分析树可以用于检测和报告代码中的语法错误。通过对语法规则的应用，可以在构建语法分析树的过程中发现不符合规则的语法结构，如类型不匹配、未定义的变量等。这有助于及早发现错误并提供更准确的错误提示。
3. 语义分析和优化：语法分析树是进行语义分析和优化的重要基础。在语法分析树的基础上，可以进行类型检查、符号表管理、语义转换和优化等操作。这些操作可以提高程序的性能、可读性和可维护性。
4. 代码生成：语法分析树可以用作生成中间代码或目标代码的基础。通过遍历语法分析树，可以将程序的语法结构转化为具体的代码表示形式，为后续的代码生成阶段提供便利。

语法分析单元类型

```java
public class AbstractSyntaxConstant {
    public static final int UNKNOWN_TYPE = 101;
    public static final int BEGIN_NODE = 102;
    public static final int VARIABLE_DEFINE = 103;
    public static final int STATE_DEFINE = 104;
    public static final int LOGIC_DEFINE = 105;
    public static final int RESULT_DEFINE = 106;
    public static final int FUNCTION_DEFINE = 107;
    public static final int PARAMETER_DEFINE = 108;
    public static final int INPUT_TEMPLATE = 109;
    public static final Integer INPUT_DEFINE = 110;
```

```
    public static final Integer WAIT_DEFINE = 111;
}
```

语法分析结果树 - 节选

```
|__type:VARIABLE_DEFINE value: What_I_can_do =  "我是一款智能客服机器人,支持查询电话等功能,欢迎使用!" line:3 children:3
        |__type:IDENTIFIER value:What_I_can_do line:3 children:0
        |__type:OPERATOR value:= line:3 children:0
        |__type:RESULT_DEFINE value: "我是一款智能客服机器人,支持查询电话等功能,欢迎使用!" line:3 children:1
                |__type:STRING value:"我是一款智能客服机器人,支持查询电话等功能,欢迎使用!" line:3 children:0
```

核心分析方法 - LL分析

总分析函数

```
public AbstractSyntaxTree analyze(List<Token> tokens) {
    ast = new AbstractSyntaxTree();
    tokensStream = new TokensStream();
    tokensStream.setStream(tokens);

    pointer = 0;// 顺序识别tokens的指针
    tempPointer = 0;// 临时指针
    int length = tokens.size();// tokens的长度
    Result res = null;

    while(pointer < length) {
        // 先尝试识别下一条大语句是是什么类型的
        if(pointer + 1 >= length){
            // 判断异常
        }
        if(变量定义) {
            // 变量声明
            variableDeclaration(List<Token> tokens);
        }else{
            // 状态声明
            stateDeclaration(List<Token> tokens);
        }
        pointer += size;//更新指针
    }
    return ast;
}
```

状态分析函数

```
public Result stateDeclaration(List<Token> tokens) {
```

```
    // 判断语法异常
    int son_length = 0;
    if(tokens.get(0).type == TokensConstant.KEYWORD &&
tokens.get(0).token.equals("global")){
        // 识别关键字
    }

    if(tokens.get(son_length).type == TokensConstant.IDENTIFIER){
        // 识别标识符
        if(tokens.get(son_length).type == TokensConstant.DELIMITER &&
tokens.get(son_length).token.equals("{")){
            // 识别逻辑，可能存在很多条逻辑
            Result res = null;
            // 存在链式递归的情况 对逻辑递归识别
            while((res =
logicDefinition(tokens.subList(son_length,tokens.size()))) != null){
                // 处理指针
            }

            if(son_length > 3){ // 识别了不止一条逻辑
                if(son_length >= tokens.size()){
                    // 抛出异常
                }
                if(tokens.get(son_length).type == TokensConstant.DELIMITER
&& tokens.get(son_length).token.equals("}")){
                    // 成功返回
                }else{
                    // 抛出语法异常
                }
            }else{
                // 抛出语法异常
            }
        }else {
            // 抛出语法异常
        }
    }else{
        // 抛出语法异常
    }
}
```

其他分析函数

> 其他分析函数，均依照上方所述的结构，凭借互相调用，构造语法分析树！

3. 语义分析

标识符扫描 - 预处理

形成状态映射表，和变量映射表，形成索引。

```
Map<String, Variable> variableMap;
Map<String,State> stateMap;

public class State {
    private Integer index;
    private Boolean isGlobal;
    private Integer id;
}

public class Variable {
    private int length = 0;
    List<VariableSlice> variableSliceList;
}

public class VariableSlice {
    private Integer index; // 在ast树中的位置
    private Integer type; // 定义
    private Integer resultId;// 指向结果id，默认是未知的结果，随着对变量的解析，才能赋值
}
```

变量分析

> 针对变量定义，解析出结果ID，填充到变量切片中去

状态分析

填充State属性

> 全局变量等状态属性

生成 TransferNode

```
public class TransferNode implements Serializable{
    private Condition condition; // 对应着 <>
    private Integer resultID;
    private Integer targetState; // 对应着 goto
}
```

Condition 条件分析

```java
public Map<Integer, TransferList> conditionProcessor(Map<Integer, TransferList> transMap) {
    // 遍历所有的转移列表
    for(Map.Entry<Integer,TransferList> entry : transMap.entrySet()){
        // 遍历每一个转移列表
        List<TransferNode> transferNodeList = entry.getValue().getTransferList();
        for(TransferNode transferNode : transferNodeList){
            // 遍历每一个转移节点
            Condition condition = transferNode.getCondition();
            // 遍历每一个条件
            List<String> regex = condition.getREGEX();
            // 抛出异常
            if(非交互匹配){
                // 生成模式串
                Pattern pattern = Pattern.compile(regex.get(0));
                condition.setPattern(pattern);
            }else if(condition.getType() == ConditionConstant.INPUT){
                //交互匹配处理
            }else{
                // 抛出异常
            }
        }
    }
    return transMap;
}
```

Result 结果分析

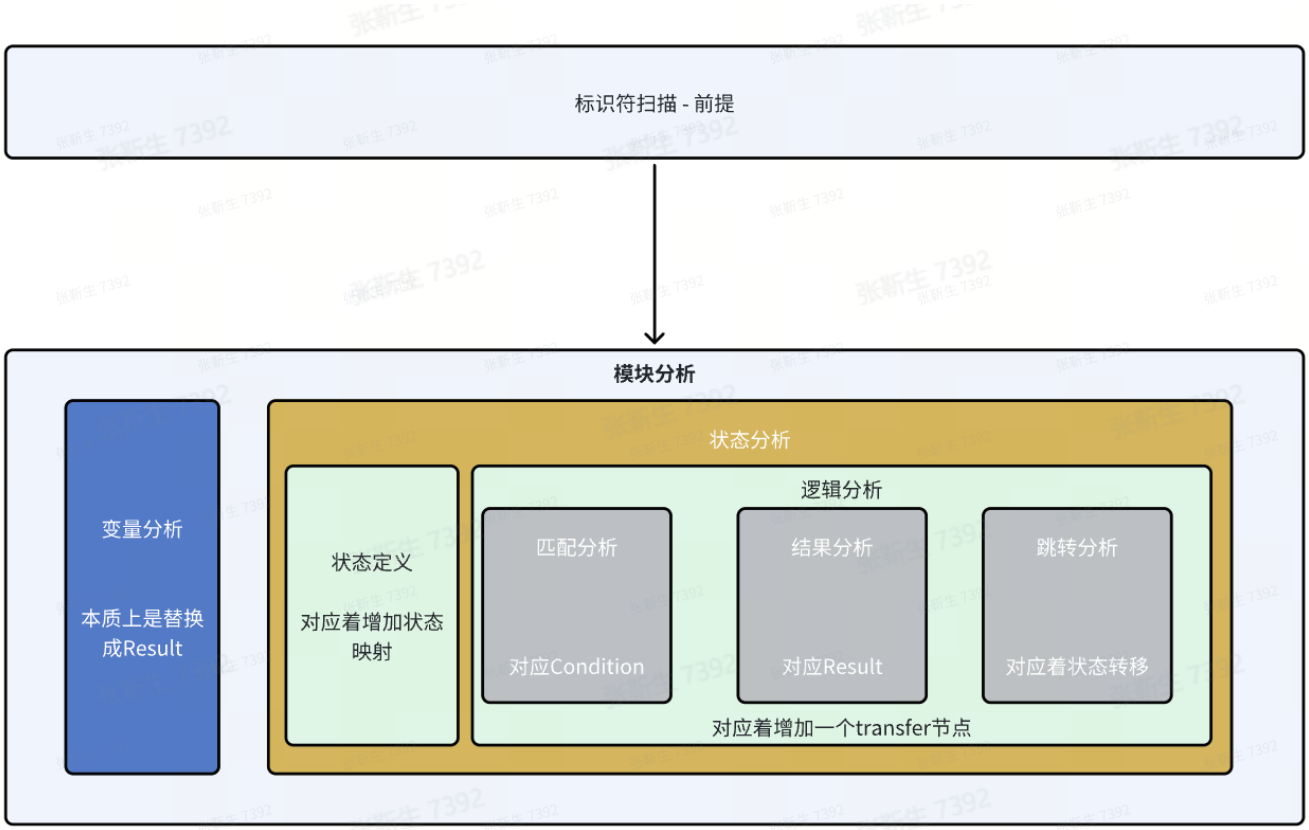
> 针对所有的结果，构建结果字典，生成可以相互引用的结果ID，作为TransferNode的结果

```java
public Result resultAnalysis(AbstractSyntaxNode node, Integer index, List<String> params) {
    // 处理异常

    // 结果列表
    List<Integer> resultList = new java.util.ArrayList<>();
    for(int i = 0;i < children.size();i++){
        AbstractSyntaxNode child = children.get(i);
        if(child.getType() == TokensConstant.IDENTIFIER){
            // IDENTIFIER 涉及递归分析，并且提防成环
        }else if(child.getType() == TokensConstant.STRING){
            // String 查看结果字典，如果没有加入
```

```
        }else if(child.getType() == TokensConstant.NUMBER){
            // Number 同String
        }else if(child.getType() == AbstractSyntaxConstant.FUNCTION_DEFINE){
            // 函数定义 调用 functionanalysos
        } else if(child.getType() == TokensConstant.OPERATOR &&
child.getValue().equals("+")) {
            // 继续处理子结果
        }else{
            // 抛出异常
        }
    }
    // 整合resultList
    if(resultList.size() == 0){
        // 异常
    }else if(resultList.size() != 1){ // 创建复合结果 不添加到map映射中
        // 多个结果合成复合结果
        return 复合结果
    }else{
        // 单个结果
        return 单个结果
    }
}
```

HelloMap,defaultMap,waitMap 等结构生成

> 在逻辑分析时，对begin，default，wait等关键字做处理，更新相关结构！



数据结构设计

实体类

依赖实体 - 响应的逻辑

```java
public class RobotDependency implements Serializable {
    private String dependencyId; // 依赖id
    private List<Integer> globalState; // 全局状态集合
    private Integer defaultState; // 默认状态
    private Map<Integer, TransferList> transMap; // 转移函数
    private ResultDictionary resultDictionary; // 结果字典
    private Map<String,Integer> stateMap; // 状态映射
    private Map<Integer,WaitResult> waitResultMap; // 等待结果
    private Map<Integer,TransferNode> defaultResultMap; // 默认结果
    private Map<Integer,TransferNode> HelloMap; // 问好映射
    private Boolean suggestion_when_check; // 是否在check的时候给出建议
    private Boolean suggestion_when_pass; // 是否在pass的时候给出建议
    public RobotDependency() {
        transMap = new java.util.HashMap<>(); // 初始化一下
        defaultState = -1; // 默认没有init
    }
}
```

词法分析结构

Token

```java
public class Token {
    public int type; // 类型
    public String token; // 值
    public int lineIndex; // 行号
}
```

Tokens

```java
public class Tokens {
    private List<Token> stream;
```

```
    private Map<Integer,List<String>> map;
}
```

语法分析结构

语法分析树节点

```
public class AbstractSyntaxNode {
    private Integer tokenIndex; // 标识这个的顺序
    private Integer length; // 这个节点的长度
    private Integer type; // 类型
    private String value; // 值
    private Integer line; // 行数
    private List<AbstractSyntaxNode> children; // 子代
    public void addChild(AbstractSyntaxNode node){
        // 增加孩子
    }
    public void insertChild(AbstractSyntaxNode node) {
        // 插入节点
    }
    public void logPrint(FileOutputStream fileOutputStream,int level){
        // 打印树结构
    }
}
```

语法分析树结构

```
public class AbstractSyntaxTree {
    public AbstractSyntaxNode root;
    public void logPrint(){
        // 打印日志
    }
}
```

词法分析相应逻辑结构

条件

```
public class Condition implements java.io.Serializable{
    Integer type; // type
    List<String> REGEX; // 正则，参数列表
```

```
    Integer input_num; // input_num 应该理论上需要多少个input
    Pattern pattern; // 正则表达式
    List<String> params;    // 参数列表
    public Boolean equals (Condition condition){
        // 判定条件是否相同
    }
}
```

结果

```
public class CompileResult implements java.io.Serializable{
    private Integer type; // 类型
    private String value; // 值
    private Object data; // 数据
}
```

状态

```
public class State {
    private Integer index; // 索引
    private Boolean isGlobal; // 是否是全局
    private Integer id; // 状态id
}
```

转移

```
public class TransferNode implements Serializable{
    private Condition condition; // 对应着 <>
    private Integer resultID; // 响应结果ID
    private Integer targetState; // 对应着 goto
}
```

建议

```
public class Suggestion {
    String suggestion; // 显示的建议
    String inputTemplate; // 输入模板
}
```

编译信息结构

编译信息

```
public class CompileInfo {

    private List<CompileWarning> compileWarnings;
    private List<CompileError> compileError;
}
```

编译警告

```
public class CompileWarning {
    private String msg; // 哪一行出现问题
    private int warningLine;

    public CompileWarning(String msg, Integer warningLine) {
        this.msg = msg;
        this.warningLine = warningLine;
    }
}
```

编译错误

```
public class CompileError {
    private int location; // 哪一行出现错误
    private int errorType;

    public CompileError(Integer location, Integer errorType) {
        this.location = location;
        this.errorType = errorType;
    }
}
```

测试数据结构

```
public class TestUnitEntity implements java.io.Serializable{
    private Integer state; // 状态ID
    private String name; // 状态名
    private String input; // 输入
```

```
    private String response; // 响应
    private Integer targetId; // 跳转ID
    private String targetName; // 跳转状态名
    private Integer checkState; // 是否检查状态
}
```

常量类

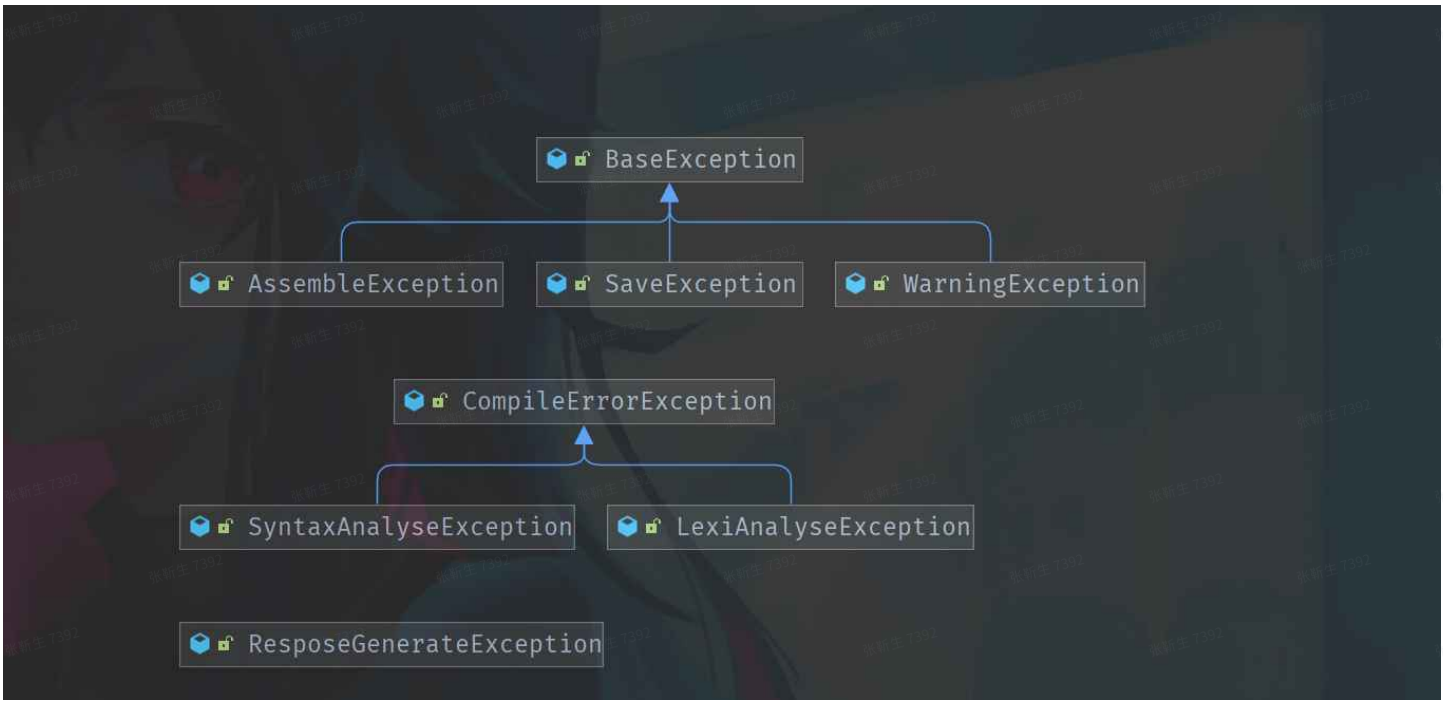
> 包含所有的系统常量，结构常量，类型常量

例如： 系统常量

```
public class SystemConstant {
    // 本项目中的resources文件夹下的RobotDependency文件夹
    public static final String ROBOT_DEPENDENCY_PATH =
"src/main/resources/RobotDependency/";
    public static final String LEXICAL_ANALYSIS_PATH =
"src/main/resources/Logs/lexicalAnalysis.txt";
    public static final String SYNTAX_ANALYSIS_PATH =
"src/main/resources/Logs/AbstractSyntaxTree.txt";
    public static final String LOG_PATH =
"src/main/resources/Logs/RUNTIME_LOG.txt";
    public static final String INPUT_PATH =
"src/main/resources/CodeInputFile/";
    public static final String SEMANTIC_ANAYSIS_PATH =
"src/main/resources/Logs/SemanticAnalysis.txt";

    public static final String DEFAULT_DEPENDENCY_NAME = "NewDependency";
    public static final long WAIT_TIME = 100; // 进程等待时间 单位毫秒
}
```

异常类

> 对错误，警告等异常的自定义类



可能的错误

```
"AST语法树错误，无法继续进行语义分析（正常情况下看不到该条错误）";
"状态定义重复";
"出现未标识变量";
"变量有标识，但变量位置错误,请尽量不要使用一个变量赋值多次的情况!";
"变量依赖循环,请重新检查代码逻辑!";
"出现了逻辑上不存在的错误";
"依赖变量未求解，请检查是否依赖顺序存在问题";
"依赖变量求解失败，请检查是否依赖顺序存在问题";
"变量求解失败，请检查逻辑等问题";
"初始化状态时，找不到对应的状态";
"匹配符解析失败";
"输入符解析失败";
"输入时，同一个标识符出现两次，错误";
"结果分析时，发现未定义的标识符";
"结果分析时，发现未求解的标识符";
"函数使用参数重复";
"解析结果时，出现嵌套的函数调用解析出错";
"函数编译出错，建议检查函数是否正确";
"解析结果时，函数调用解析出错";
"逻辑求解时，结果分析失败";
"逻辑分析失败";
"条件解析失败";
"状态映射结构错误";
"不只存在一个默认状态";
"不只存在一个超时响应";
"时间定义错误";
"代码中存在非法字符";
```

```
"在尝试将词法识别为标识符时，发现错误";
"在尝试将词法识别为数字时，发现错误";
"在尝试将词法识别为字符串时，发现错误";
"在尝试将词法识别为匹配符时，发现错误";
"非法编码，请检查代码编码格式";
"未知语法分析错误";
"在识别变量和状态时，未能成功";
"语句不完整，请重新检查";
"识别变量时，第一个词法单元不是标识符";
"识别变量时，第二个词法单元不是等号";
"识别变量时，变量的值分析失败";
"识别状态时，未识别到标识符";
"识别状态时，未识别到左大括号,请检查是否有遗漏";
"识别状态时，未识别到逻辑表达式,或者逻辑表达式不完整";
"识别状态时，未识别到右大括号,状态定义不完整";
"识别结果时，未识别到合法的结果表达式";
"识别结果时，未知错误";
"识别结果时，未识别到加号后的表达式";
"识别逻辑表达式时，未识别到合法的逻辑表达式";
"识别逻辑表达式时，未识别到正确的输入表达式";
"识别逻辑表达式时，未识别到正确的等待表达方式";
"识别逻辑表达式时，识别到非法关键字";
"识别逻辑表达式时，不能识别到完整的逻辑表达式";
"识别结果时，不能识别到完整的结果表达式";
"识别函数时，不能识别到完整的函数表达式";
"识别函数时，不能识别到函数名";
"识别函数时，不能识别到左括号";
"识别函数时，不能识别到函数参数";
"识别函数时，不能识别到右括号";
"识别函数参数时，不能识别到完整的参数表达式";
"识别函数参数时，不能识别到参数符号";
"识别函数参数时，不能识别到参数标识符";
"识别函数参数时，不能识别到参数赋值符号";
"识别函数参数时，不能识别到参数赋值结果";
"识别函数参数时，存在参数分隔符,但是参数不完整";
"识别输入表达式时，不能识别到完整的输入表达式";
"识别输入表达式时，不能识别到输入分隔";
"识别输入表达式时，不能识别到输入定义";
"识别输入表达式时，不能识别到合法的输入表达式";
"识别输入表达式时，输入格式错误";
"识别逻辑表达式时，不能识别到合法的跳转表达式";
"识别逻辑表达式时，不能识别到冒号";
"识别逻辑表达式时，不能识别到返回表达式";
"识别逻辑表达式时，不能识别到合法的返回表达式";
"存在多个欢迎语句";
"语义分析时，扫描标识符失败";
"语义分析时，分析该状态，发现一个状态下的逻辑表达式中存在多个歧义匹配符！";
```

可能的警告

```
"该变量标识符已经用作状态码,请减少重复使用,以免造成混淆!";
"已经存在全局状态,请尽量使用一个全局关键字";
"该标识符已经用作变量名,请减少重复使用";
"输入符号连续，可能导致无法读取输入，导致状态不可达";
"输入字符串无效，可能导致无法读取输入，导致状态不可达";
"没有按照推荐的交叉方式，进行输入，可能导致无法读取输入成功！ ";
"输入符号没有使用完毕，请移除冗余的输入符号";
"该标识符已经用作变量名,请尽量减少这样的容易造成混淆的用法!";
"该转移会跳转到本身状态，该转移无效!";
"该状态存在default,但是没有设置转移状态，可能导致成为终止僵尸状态！ ";
"该状态不可达，可能导致成为终止僵尸状态！ ";
```

程序接口

程序接口

通过Springboot项目架构构造，使用autowired注入！

Controller 层 调用 Service层 相互调用！

前后端接口

通过http请求，websocket实现！

程序间接口

public Result variableDeclaration(List<Token> tokens);

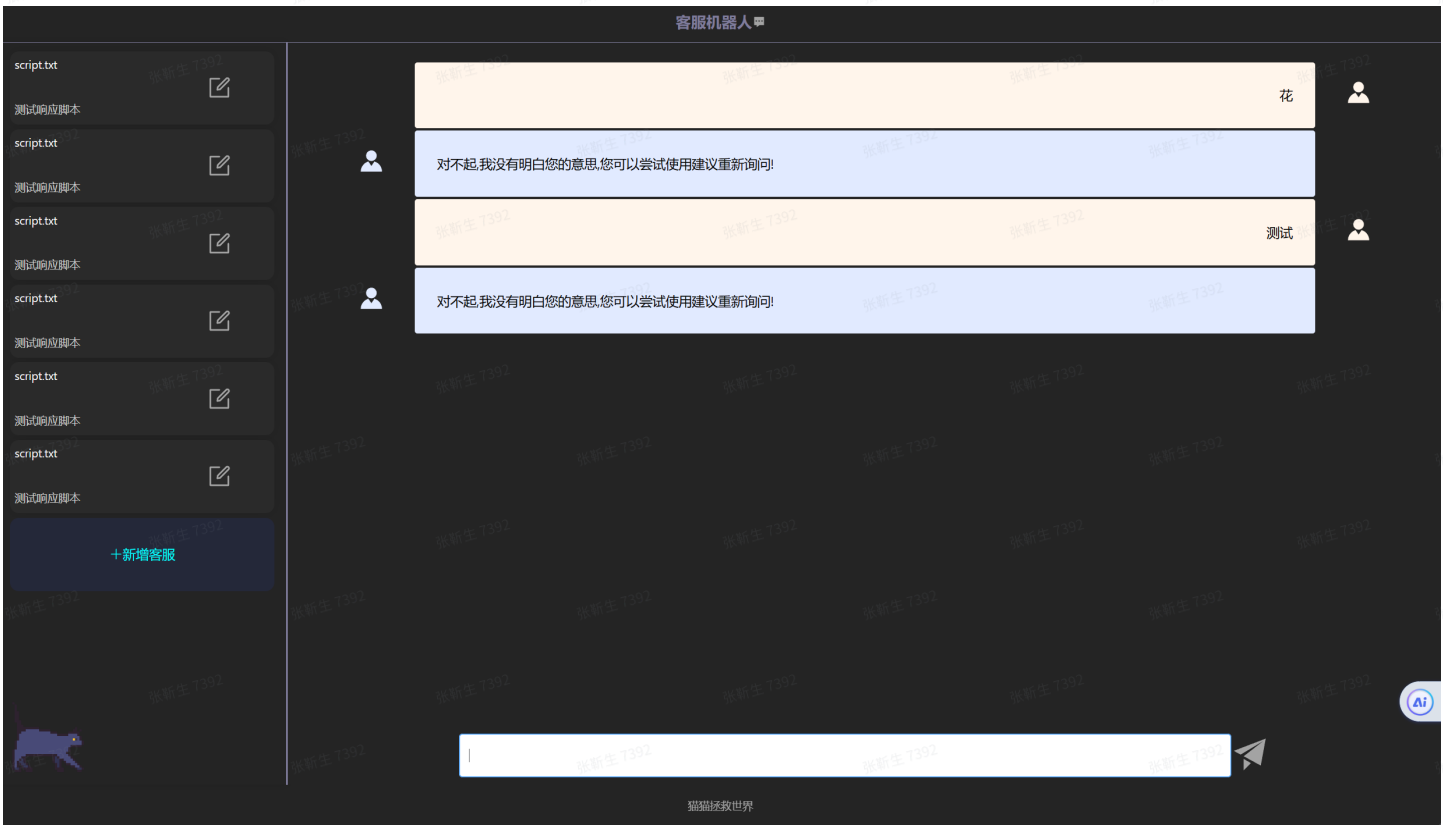
通过Result相互传递，Result可以注入各种对象类型，相互传递数据！

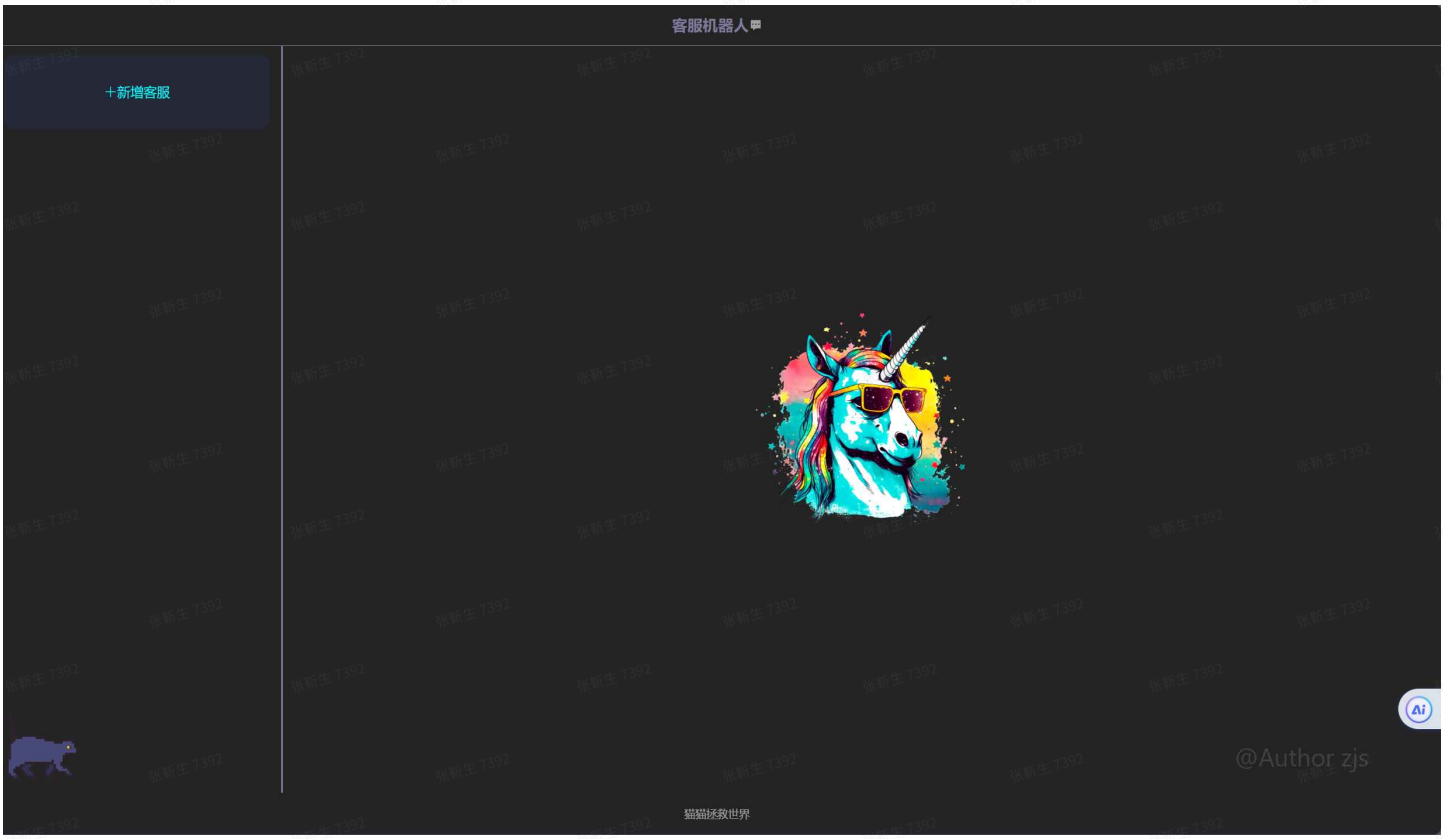
```java
package com.example.javaservice.Result;

import com.example.javaservice.Constant.ResultConstant;
import com.example.javaservice.Pojo.Entity.TransferNode;
import lombok.Data;

@Data
public class Result<T> {
    private int state;
    private T data;
```

```
11      private String msg;
12  }
```

人机接口

通过前端页面和配置文件实现.





日志配置

```yaml
auto-test-script:
  totalScriptsPath: src\main\resources\TestScripts\
  # 脚本路径
  errorScriptsPath: src\main\resources\TestScripts\ErrorScripts\
  # 含有错误的脚本
  warningScriptsPath: src\main\resources\TestScripts\WarningScripts\
  # 含有警告的脚本
  successScriptsPath: src\main\resources\TestScripts\SuccessScripts\
  # 没有错误的脚本

  # 含有警告的脚本 含有错误的脚本 没有错误的脚本
  compileOutputPath: src\main\resources\TestResults\TestCompileResults\
  responseOutputPath: src\main\resources\TestResults\TestResponseResults\

  # 测试后清除脚本
  clearScriptAfterCompile: true

  # 测试偏向

  # 1. 频繁的状态转移
  # 2. 频繁字符串匹配
  # 3. 访问查询数据库
  # 4. 测试default
  # 5. 测试wait连接

  responseScriptPath: src\main\resources\TestScripts\ResponseTestScripts

  testBias: false

  #  输入匹配 2;
  #  精准匹配 3;
  #  模糊匹配 4;
  #  正则匹配 5;
  testBiasFor: 2

  # 测试编译 对于每个脚本的测试次数

  # 测试解释
  checkBookPath: src\main\resources\TestScripts\CheckBook\ # 响应检查路径

  testRound: 3 # 测试所有状态为一轮
  testNumber: 100 # 测试每个状态中的语句

```

```
  randomTest: false # 随机测试
  randomTestSkip: 10 # 随机测试跳跃数量

  # 随机生成 针对输入是否随机生成组合 开启后会出现大量随机测试，需要多次检验
  randomGenerate: false
```

系统测试

测试参数

```
auto-test-script:
  # 懒人模式
  totalScriptsPath: src\main\resources\TestScripts\
  # 脚本路径
  errorScriptsPath: src\main\resources\TestScripts\ErrorScripts\
  # 含有错误的脚本
  warningScriptsPath: src\main\resources\TestScripts\WarningScripts\
  # 含有警告的脚本
  successScriptsPath: src\main\resources\TestScripts\SuccessScripts\
  # 没有错误的脚本

  # 含有警告的脚本 含有错误的脚本 没有错误的脚本
  compileOutputPath: src\main\resources\TestResults\TestCompileResults\
  responseOutputPath: src\main\resources\TestResults\TestResponseResults\

  # 测试后清除脚本
  clearScriptAfterCompile: true

  # 测试偏向

  # 1. 频繁的状态转移
  # 2. 频繁字符串匹配
  # 3. 访问查询数据库
  # 4. 测试default
  # 5. 测试wait连接

  responseScriptPath: src\main\resources\TestScripts\ResponseTestScripts

  testBias: false

  #  输入匹配 2;
  #  精准匹配 3;
```

```
#  模糊匹配 4;
#  正则匹配 5;
testBiasFor: 2

# 测试编译 对于每个脚本的测试次数

# 测试解释
checkBookPath: src\main\resources\TestScripts\CheckBook\ # 响应检查路径

testRound: 3 # 测试所有状态为一轮
testNumber: 100 # 测试每个状态中的语句

randomTest: false # 随机测试
randomTestSkip: 10 # 随机测试跳跃数量

# 随机生成 针对输入是否随机生成组合 开启后会出现大量随机测试，需要多次检验
randomGenerate: false
```

测试桩单元测试

```
class UnitTests {

    @Autowired
    DependencyMapMapper dependencyMapMapper;

    @Autowired
    TelephoneMapMapper telephoneMapMapper;

    @Test
    void contextLoads() {
    }

    /**
     * 自动测试脚本
     */
    @Test
    void TestAutoScript(){
        // 读取文件
        File file = new File("C:\\Users\\Administrator\\Desktop\\test.txt");
        Scanner scanner = null;
        try {
            scanner = new Scanner(file, StandardCharsets.UTF_8.name());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
```

```
26          String code = "";
27          while (scanner.hasNextLine()) {
28              code += scanner.nextLine();
29          }
30          // 词法分析
31          LexicalAnalyzer lexicalAnalyzer = new LexicalAnalyzerImpl();
32          Tokens tokens = lexicalAnalyzer.analyze(code);
33          // 语法分析
34          SyntaxAnalyzer syntaxAnalyzer = new SyntaxAnalyzerIpml();
35          AbstractSyntaxTree abstractSyntaxTree =
    syntaxAnalyzer.analyze(tokens.getStream());
36          // 语义分析
37          SemanticAnalyzer semanticAnalyzer = new SemanticAnalyzerImpl();
38  //          semanticAnalyzer.analyze(abstractSyntaxTree);
39          // 生成DSL
40  //          DSLCompiler dslCompiler = new DSLCompilerImpl();
41  //          String dsl = dslCompiler.compile(abstractSyntaxTree);
42  //          System.out.println(dsl);
43      }
44
45
46      /**
47       * 正则匹配测试
48       */
49      @Test
50      void TestStrMatch () {}
51      /**
52       * 测试数据库存储
53       */
54      @Test
55      void TestDBTransfer(){}
56      /**
57       * 测试词法分析器
58       */
59      @Test
60      void TestLexicalAnalyzer(){
61          LexicalAnalyzer lexicalAnalyzer = new LexicalAnalyzerImpl();
62          // 把文件转换成字符串
63          FileInputStream fileInputStream = null;
64          try {
65              fileInputStream = new FileInputStream( SystemConstant.INPUT_PATH +
    "test1.txt");
66          } catch (FileNotFoundException e) {
67              e.printStackTrace();
68          }
69          byte[] bytes = null;
70          // 字符流转换
```

```
71          try {
72              bytes = fileInputStream.readAllBytes();
73          } catch (IOException e) {
74              e.printStackTrace();
75          }
76          bytes = new String(bytes).getBytes(StandardCharsets.UTF_8);
77          String code = new String(bytes);
78          Tokens tokens = lexicalAnalyzer.analyze(code);
79          lexicalAnalyzer.logPrint(tokens);
80      }
81
82      /**
83       * 测试语法分析器
84       */
85      @Test
86      void TestSyntaxAnalyzer(){
87          LexicalAnalyzer lexicalAnalyzer = new LexicalAnalyzerImpl();
88          // 把文件转换成字符串
89          FileInputStream fileInputStream = null;
90          try {
91              fileInputStream = new FileInputStream( SystemConstant.INPUT_PATH +
    "test1.txt");
92          } catch (FileNotFoundException e) {
93              e.printStackTrace();
94          }
95          byte[] bytes = null;
96          // 字符流转换
97          try {
98              bytes = fileInputStream.readAllBytes();
99          } catch (IOException e) {
100             e.printStackTrace();
101         }
102
103         bytes = new String(bytes).getBytes(StandardCharsets.UTF_8);
104         String code = new String(bytes);
105         Tokens tokens = lexicalAnalyzer.analyze(code);
106         lexicalAnalyzer.logPrint(tokens);
107         SyntaxAnalyzer syntaxAnalyzer = new SyntaxAnalyzerIpml();
108         AbstractSyntaxTree ast = syntaxAnalyzer.analyze(tokens.getStream());
109         ast.logPrint();// 打印日志
110     }
111     /**
112      * 测试语义分析器
113      */
114     @Test
115     RobotDependency TestSemanticAnalyzer(){
116         LexicalAnalyzer lexicalAnalyzer = new LexicalAnalyzerImpl();
```

```
117         // 把文件转换成字符串
118         FileInputStream fileInputStream = null;
119         try {
120             fileInputStream = new FileInputStream( SystemConstant.INPUT_PATH +
    "test1.txt");
121         } catch (FileNotFoundException e) {
122             e.printStackTrace();
123         }
124         byte[] bytes = null;
125         // 字符流转换
126         try {
127             bytes = fileInputStream.readAllBytes();
128         } catch (IOException e) {
129             e.printStackTrace();
130         }
131 
132         bytes = new String(bytes).getBytes(StandardCharsets.UTF_8);
133         String code = new String(bytes);
134 
135         Tokens tokens = lexicalAnalyzer.analyze(code);
136         lexicalAnalyzer.logPrint(tokens);
137         SyntaxAnalyzer syntaxAnalyzer = new SyntaxAnalyzerIpml();
138         AbstractSyntaxTree ast = syntaxAnalyzer.analyze(tokens.getStream());
139         ast.logPrint();
140 
141         SemanticAnalyzer semanticAnalyzer = new SemanticAnalyzerImpl();
142         Map<String, Object> analyse = semanticAnalyzer.analyse(ast);
143 
144         ((SemanticAnalyzerImpl) semanticAnalyzer).logPrint();
145 
146         return (RobotDependency) analyse.get("robotDependency");
147     }
148 
149     /**
150      * 测试CS系统
151      */
152     @Test
153     public void testCS(){
154         RobotDependency robotDependency = TestSemanticAnalyzer();
155         robotDependency.setDependencyId("TestDependency");
156         DSLCompiler dslCompiler = new DSLCompilerImpl();
157         dslCompiler.SaveDependency(robotDependency,null);
158         // 保存
159         CustomerService.testAssembly(robotDependency);
160 
161         // 从文件读取输入
162         FileInputStream fileInputStream = null;
```

```
163         try {
164             fileInputStream = new FileInputStream( SystemConstant.INPUT_PATH +
    "input1.txt");
165         } catch (FileNotFoundException e) {
166             e.printStackTrace();
167         }
168         // 按照行读取
169         Scanner scanner = new Scanner(fileInputStream);
170         while (scanner.hasNextLine()){
171             String input = scanner.nextLine();
172         }
173     }
174 
175     /**
176      * 测试依赖存储
177      */
178     @Test
179     void testDB(){
180         DependencyMap dependencyMap = new DependencyMap();
181         dependencyMap.setCode("test");
182         dependencyMap.setPath("test");
183         dependencyMap.setName("test");
184         dependencyMapMapper.insert(dependencyMap);
185         DependencyMap dependencyMap1 = new DependencyMap();
186         dependencyMap1.setCode("test1");
187         dependencyMap1.setPath("test1");
188         dependencyMap1.setName("test1");
189         dependencyMapMapper.insert(dependencyMap1);
190     }
191 
192     /**
193      * 清除数据库中的数据
194      */
195     @Test
196     void clearDataSource(){
197         List<DependencyMap> dependencyMaps =
    dependencyMapMapper.selectList(null);
198         for(DependencyMap dependencyMap : dependencyMaps){
199             // 清除在数据库中的数据
200             dependencyMapMapper.deleteById(dependencyMap.getId());
201             // 清除在本地的数据
202             String path = SystemConstant.ROBOT_DEPENDENCY_PATH +
    dependencyMap.getPath();
203             File file = new File(path);
204             if(file.exists()){
205                 file.delete();
206             }
```

```
            path = SystemConstant.INPUT_PATH + dependencyMap.getCode();
            file = new File(path);
            if(file.exists()){
                file.delete();
            }
        }
    }

    /**
     * 进行电话号码存储
     */
    @Test
    void addTelePhone(){
        telephoneMapMapper.insert(new TelephoneMap("淘宝",123435445));
    }

}
```

编译器系统测试

> 通过提前写好一个脚本中的错误到注释中，然后批量编译脚本查看是否一致！

```
void TestCompile(){
    LOG.INFO("开始测试");
    // 输出结果 用日期开头
    String outputString = "";

    // 获取所有的脚本路径
    String ErrorScriptsPath = autoTestScriptConfig.getErrorScriptsPath();
    String SuccessScriptsPath = autoTestScriptConfig.getSuccessScriptsPath();
    String WarningScriptsPath = autoTestScriptConfig.getWarningScriptsPath();

    // 编译随机部分脚本 查看是否和预期一致
    // 读取文件
    File ErrorScriptsDir = new File(ErrorScriptsPath);
    File SuccessScriptsDir = new File(SuccessScriptsPath);
    File WarningScriptsDir = new File(WarningScriptsPath);

    // 获取所有文件
    File[] ErrorScriptsFiles = ErrorScriptsDir.listFiles();
    File[] SuccessScriptsFiles = SuccessScriptsDir.listFiles();
    File[] WarningScriptsFiles = WarningScriptsDir.listFiles();

    // 统计测试文件数量 以及 本次测试文件
    outputString += "本次测试文件数量: \n";
```

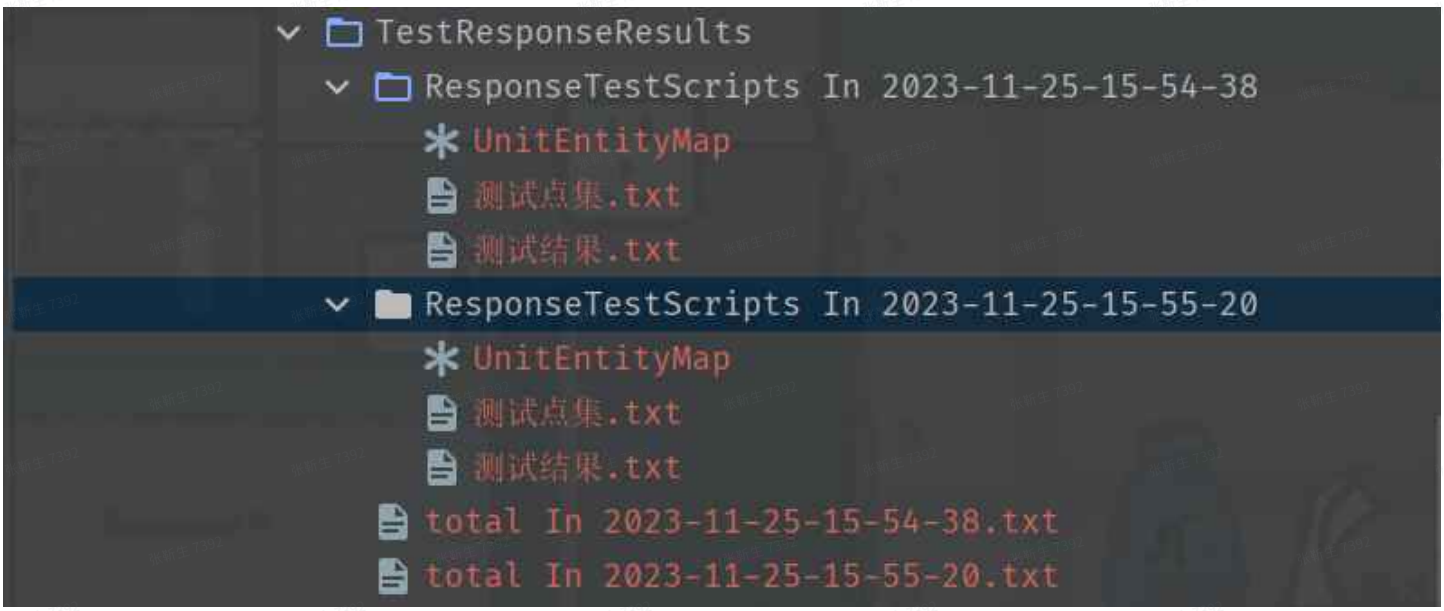
```
    。。。

    // 定义数目
    Integer ErrorOkTest = 0;
    。。。

    LOG.INFO("共有" + ScriptCount + "个脚本需要测试");

    // 读取文件内容
    for(File ErrorScript : ErrorScriptsFiles){
        String code = "";
        String annotion = "";
        try {
            // 读取脚本 编译
            Result result = dslCompiler.compileByCode(newDependencyDto);
            // 得到编译结果 放到注释中
            annotion = ParseResultFromResult(result);
        } catch (Exception e) {
            annotion = ParseResultFromException(e);
        }

        // 和代码中的结果比对
        if(code.startsWith(annotion)){
            LOG.INFO(ErrorScript.getName() + "测试错误脚本成功");
        }else{
            LOG.ERROR(ErrorScript.getName() + "测试错误脚本失败");
        }
    }
    // 成功脚本

    // 警告脚本


    // 打印总结
    LOG.INFO("测试完成");
    LOG.INFO("错误脚本测试完成" + ErrorOkTest + "/" + ErrorScriptCount);
    LOG.INFO("正确脚本测试完成" + SuccessOkTest + "/" + SuccessScriptCount);
    LOG.INFO("警告脚本测试完成" + WarningOkTest + "/" + WarningScriptCount);
    outputString += "错误脚本共" + ErrorScriptCount + "个，测试通过" +
ErrorOkTest + "个\n";
    outputString += "正确脚本共" + SuccessScriptCount + "个，测试通过" +
SuccessOkTest + "个\n";
    outputString += "警告脚本共" + WarningScriptCount + "个，测试通过" +
WarningOkTest + "个\n";

    if(满足测试数目) {
        LOG.INFO("测试通过");
```

```
68          outputString += "测试通过\n";
69      }else{
70          LOG.ERROR("测试失败");
71          outputString += "测试失败\n";
72      }
73      LOG.INFO("测试结束");
74      outputString += "测试结束\n";
75
76      // 写入文件
77
78      // 是否清除脚本
79      if(autoTestScriptConfig.getClearScriptAfterCompile()){
80          clearDataSource();
81      }
82  }
```

响应器系统测试

> 通过随机测试得到测试点集和UnitEntityMap文件，检验无误后，UnitEntityMap文件可以作为输入再次加入到脚本的测试之中！
>
> 测试点：
>
> 1. 测试结果是否和测试依赖相同
> 2. 相同的输入是否响应结果相同

测试生成文件



UnitEntityMap

> 映射序列化文件，可以读取作为测试依赖！

测试点集

```
{状态ID=1, 状态名称='main', 输入='', 响应='', 跳转ID=1, 跳转状态名称='main', 检查状态=等待核验}
```

测试核心代码

```
void TestResponse(){
    Integer testPoint = 0;
    String totalOutputString = "";
    totalOutputString += TestINFO("开始测试响应\n");
    // 打印测试参数
    。。。
    // 读取相应测试脚本
    。。。
    // 得到所有脚本文件夹

    for(File ResponseScriptDir : ResponseScriptsFiles){
        outputString = "";
        File ResponseScript = new File(ResponseScriptDir,"script.txt");
        // 是否随机抽取测试
        outputString += TestINFO("开始测试" + ResponseScriptDir.getName());
        // 测试参数
        Integer totalTestPoint = 0;
        Integer passTestPoint = 0;
        Integer failTestPoint = 0;
        Integer unknownTestPoint = 0;

        String code = "";
        Integer robotId = -1;
        try {
            // 编译
        } catch (Exception e) {
            LOG.ERROR(ResponseScript.getName() + "编译响应脚本失败");
        }
        // 装配相应脚本依赖
        dependencyManager.checkOutDependency(robotId,0);
        // 响应映射 状态 - 输入映射 - 定义
        Map<Integer, Map<String,TestUnitEntity>> testUnitEntityMap = new HashMap<>();

        // 序列化映射测试单元
        File testUnitEntityMapFile = new File(ResponseScriptDir,"UnitEntityMap");
        if(testUnitEntityMapFile.exists()) {
            try {
```

```
38                  FileInputStream fileInputStream = new
FileInputStream(testUnitEntityMapFile);
39                  // 反序列化
40                  ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
41                  testUnitEntityMap = (Map<Integer, Map<String,
TestUnitEntity>>) objectInputStream.readObject();
42                  objectInputStream.close();
43                  fileInputStream.close();
44
45              } catch (Exception e) {
46                  LOG.ERROR(ResponseScript.getName() + "序列化映射失败");
47              }
48          }
49
50          Map<Integer, Map<String,TestUnitEntity>> tempTestUnitEntityMap = new
HashMap<>();
51          // 得到 Customer中Robot 所有状态
52          RobotDependency robotDependency = CustomerService.getRobotDependency();
53          Map<String, Integer> stateMap = robotDependency.getStateMap();
54          // 根据ID映射成状态名
55          Map<Integer, String> stateIdMap = new HashMap<>();
56          for(String stateName : stateMap.keySet()){
57              Integer stateId = stateMap.get(stateName);
58              stateIdMap.put(stateId,stateName);
59          }
60
61          Map<Integer, TransferList> transMap = robotDependency.getTransMap();
62
63          // 得到所有状态的测试单元
64          for(int i = 0;i < testRound; i++){
65              // 测试轮数 每一轮应该相同
66              for(String stateName : stateMap.keySet()){
67                  Integer stateId = stateMap.get(stateName);
68                  // 得到该状态对应的TransferMap
69                  TransferList transferList = transMap.get(stateId);
70                  List<TransferNode> transferNodes =
transferList.getTransferList();
71
72                  Integer tempState = stateId; // 防止状态被修改
73
74                  // 遍历所有的转移表
75                  for(TransferNode transferNode : transferNodes){
76                      // 得到测试状态 给Customer装配状态
77                      CustomerService.setStateInTest(stateId);
78                      // 更新测试点 查看是否需要跳过
79
```

```
80                  // 开始测试
81                  totalTestPoint++;
82                  testPoint++;
83                  String response;
84                  Integer resState;
85                  Result result;
86                  TestUnitEntity testUnitEntity = new TestUnitEntity();
87
88                  // 判断类型 得到输出
89                  String inputStr = getInputFromCondition(Condition);
90
91                  // 得到响应结果
92                  result = CustomerService.ResponseGenerator(inputStr);
93                  response = (String) result.getData();
94                  resState = (Integer) result.getState();
95                  // 填充测试单元
96                  testUnitEntity.setState(tempState);
97                  testUnitEntity.setName(stateIdMap.get(tempState));
98                  。。。
99
100                 // 查看模板 Map映射中是否存在该单元 并且检测
101
102
103                 // 查看临时Map中是否存在该单元 检查多次相应是否相同 不同为错误
104
105                 // 更新到Map中
106                 tempTestUnitEntityMap =
    updateTestUnitMap(stateId,inputStr,testUnitEntity,tempTestUnitEntityMap);
107
108                 // 检查testUnitEntity状态 更新统计数据
109                 if(testUnitEntity.getCheckState() ==
    TestUnitConstant.TEST_RIGHT){
110                     passTestPoint++;
111                 }else if(testUnitEntity.getCheckState() ==
    TestUnitConstant.TEST_WRONG){
112                     failTestPoint++;
113                 }else{
114                     unknownTestPoint++;
115                 }
116             }
117             outputString += TestINFO("测试状态 " + stateIdMap.get(stateId)
    + " 测试完毕");
118         }
119         outputString += TestINFO("测试轮数 " + i + " 测试完毕");
120     }
121
122     // 测试一个脚本结束 统计测试数据
```

```
123         outputString += TestINFO("测试脚本 " + ResponseScriptsDir.getName() + "
    测试完毕");
124         outputString += TestINFO("测试点数 " + totalTestPoint);
125         outputString += TestINFO("测试通过点数 " + passTestPoint );
126         outputString += TestINFO("测试失败点数 " + failTestPoint );
127         outputString += TestINFO("测试未知点数 " + unknownTestPoint );
128 
129         // 保存测试结果
130 
    SaveTestResponseData(ResponseScriptsDir.getName(),outputString,tempTestUnitEnti
    tyMap);
131         totalOutputString += outputString;
132     }
133     // 保存 totalOutputString 测试结果 到文件中
134     SaveTestResponseData("total",totalOutputString,null);
135 }
```

项目总结和亮点

1. 从零开始,整理词法分析,语法分析,语义分析,通过异常抛出对脚本语言编译的检查,从词法分析 到 语法分析 和 语义分析 都有对应的错误和警告!
2. 错误和警告精确,贴近实际运行需要,比如僵尸状态,不可达状态等的检测!
3. 清晰的项目架构,采用清晰的BS架构,可拓展性强,代码风格好!
4. 日志系统完善 , 包含调用函数,级别,信息等,针对异常无法捕获的错误可以快速定位错误信息!
5. 清晰的测试环节,针对通过日志打印语法树,词法Token流,等可以更加针对性的找到错误，灵活的测试机制，可以针对某种错误进行检测，或者设置随机检测的方式减少测试时间!
6. 灵活的语法定义,通过设定 结果的语法 实现了内置函数,字符串,变量,等基本元单位的串联,能够实现更加灵活的定义,给予灵活的拓展性!
7. 更好的面向用户,脚本解释器不仅仅实现了应答的基本功能,同时,也提供了建议的提供等更友好的方式!
8. 良好的代码管理意识 使用github来进行代码管理!

本次实验中，我深刻地理解了一个领域语言是怎么从一串字符，转变成Tokens流，语法树，最后到可执行逻辑的全过程，并且对于命名规则，注释详解等代码规范等有了更深刻的认识，也明白了代码规范在项目构造中的重要性。此外，比较重要的一点是，我深刻地理解了，建立日志系统的必要性，在编码过程中，通过打印日志，我找了了识别参数的错误，这大大节省了差错时间。通过这学期的学习，我也明白了测试的重要性，这次实验中，我使用了三种测试方法，包含测试桩，编译测试，响应测试等，这在一定程度上证明了程序的正确性，健壮性。