基于领域特定语言的客服机器人设计与实现实验报告

LaTeX by 王子轩（学号：2022211210 班级:2022211301）

2024 年 11 月 23 日

目录

1　实验内容和实验背景

1.1　实验描述与基本要求

描述：领域特定语言（Domain Specific Language，DSL）可以提供一种相对简单的文法，用于特定领域的业务流程定制。本作业要求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广泛的应用）的自动应答逻辑，并设计实现一个解释器解释执行这个脚本，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。

基本要求：

脚本语言的语法可以自由定义，只要语义上满足描述客服机器人自动应答逻辑的要求。

程序输入输出形式不限，可以简化为纯命令行界面。

应该给出几种不同的脚本范例，对不同脚本范例解释器执行之后会有不同的行为表现

1.2　实验背景

本实验使用 Rust 语言完成了从词法分析器, 语法分析器到最终解释器的所有代码实现

有必要在此声明 Rust 语言用于此类工程化项目的优势, 以及简要阐明实验要求是如何在程序中体现的

1.2.1　Rust 语言在工程项目中的优势

1. 拥有强大的**包管理**工具 Cargo, 可以用于依赖包的下载、编译、更新、分发等，在编译各种优质第三方库时，这种支持简洁且安全

2. 通过 RAII, 所有权与生命周期的编译安全机制，实现了无 GC(垃圾回收) 的自动内存管理，**避免了绝大多数内存泄漏以及数据竞争问题**，同时提高了**访存效率**

3. 编译提示信息友好，**编译器管理严格**, 对于各个水平的程序员都十分友好

4. 自带**测试**框架, 可进行单元测试和集成测试以及 Rust 独有的文档测试, 且可以通过 cargo 工具**自动运行各个测试桩**, 无需重写自动测试脚本

5. **模块化**解耦合程度高，一个 Rust 项目的典型结构包括 lib.rs(声明对外的公开 API 且作为整个项目的 lib 根目录) 与一个 main.rs(用于集成各个模块中的函数生成最终可执行文件)

6. Rust 提供**极其优秀的文档化支持**，**甚至你可以在文档注释中写测试用例**, 对于文档注释, 还可以通过 cargo doc –open 在浏览器中查看整个项目结构, 从而进一步分析整个项目**各个程序之间的接口**

7. **风格方面**, Rust 对于不符合 Rust 规范的变量或函数等命名会进行**编译期检查并提供解决方案**，我们可以通过 cargo check 进一步检查代码风格是否合适，也可以利用 vscode 中对 rust-analyzer 支持的代码提示插件提前发现问题

2　实验过程

2.1　定义 DSL 文法

DSL 专为特定问题领域设计，解决该领域问题的能力强，通过特定领域的高层语法，让用户可以快速描述问题，减少代码量，故无需拥有 C++ 之类高级语言的复杂语法。

文法的**核心**是定义状态，每个状态对应了机器人的一**句输出**，在每个状态中定义了**迁移关系**，通过正则表达式**匹配用户的输入**来迁移到不同的状态。即该文法为**用户输入 (事件) 驱动型**，只有当用户**满足文法构造出的有限状态机的迁移条件**，机器人才会给出对应的输出。

本实验中我为 DSL 定义了 STAGE、SPEAK、MATCH、NEXT、DEFAULT、INPUT 六个**指令关键字**

为 STAGE **定义保留字 initial**, 标注状态机的开始;

为 NEXT **定义保留字 EXIT**, 标注状态机的结束;

为 MATCH **定义保留字 EMPTY**, 标注状态机进行空转移 (无需用户输入)。

2.1.1　指令关键字作用

1. STAGE [String]: 定义一个状态的名称, 状态默认从 initial 状态开始迁移
2. SPEAK [String]: 表示该状态的输出内容
3. MATCH [regex String]: 匹配用户输入的正则表达式或双引号包裹的字符串字面量
4. NEXT [String]: 表示对应 MATCH 匹配成功后迁移到的下一状态名
5. DEFAULT: 默认匹配当前状态前述 MATCH 匹配均未匹配成功的情形
6. INPUT [varname]: 接收用户输入，同时定义并赋值变量 varname 值为用户输入

2.1.2　文法规则

由于在匹配输入输出内容时，对应的字符串形式较为复杂，故本文法中我们要求**每条指令独占一行**

这样的**优势**是进一步简化文法设计，同时使得脚本语言更加规范化，有可读性

```
program -> stageBlock+;
stageBlock -> stageCommand speakCommand Transition
stageCommand -> STAGE [ String | initial ]
speakCommand -> SPEAK [String]
Transition -> matchBlock+ | inputBlock
matchBlock -> matchCommand nextCommand
matchCommand -> MATCH [String | pattern | EMPTY]
inputBlock -> inputCommand nextCommand
nextCommand -> NEXT [String | EXIT]
inputCommand -> INPUT [varname]
```

进一步解释:

程序从多个不同的状态块构成, 每个状态块拥有 stage 和 speak 属性, 分别表示状态名和输出内容

同时状态块中包含下一状态的迁移条件和迁移对象，可能是对应一个 InputBlock 输入块, 接收用户输入存入全局变量中，可能是多个 matchBlock 匹配块组成的数组，用于匹配用户的不同输入, 迁移到不同状态。

语法中同时规定了 matchBlock 中 DEFAULT 匹配关键字只能作为最后一个匹配块而存在

2.1.3 示例说明

```
STAGE initial
SPEAK "请问你有什么需要帮忙的"
MATCH "打个招呼"
    NEXT get-name
DEFAULT
    NEXT unknown

STAGE get-name
SPEAK "你叫什么名字"
INPUT name
    NEXT hello

STAGE hello
SPEAK "你好" + name
MATCH EMPTY
    NEXT initial

STAGE unknown
SPEAK "听不懂命令"
MATCH EMPTY
    NEXT initial
```

1. 脚本从初始状态 initial 出发，首先输出” 请问你有什么需要帮忙的”

2. 当用户输入” 打个招呼”, 状态机会迁移到 get-name 状态, 输出” 你叫什么名字”

3. 在 get-name 状态下，用户输入自己的姓名, 状态迁移到 hello 状态，输出” 你好”+ 用户输入的姓名，然后进行空转移，返回到 initial 状态

4. 在 initial 状态下输入非” 打个招呼” 等未定义匹配的字符串，通过 DEFAULT 匹配分支，到达 unknown 状态

5. unknown 状态下, 输出” 听不懂命令”, 进行空转移, 回到 initial 状态

6. **对于这种无限循环的状态，程序提供 ESC 键方式退出**

效果如图

亮点: 本 DSL 支持 MATCH 过程中的**正则表达式**的模糊匹配, 支持**变量的持久化存储**，支持**空匹配模式**，支持**中文输入**

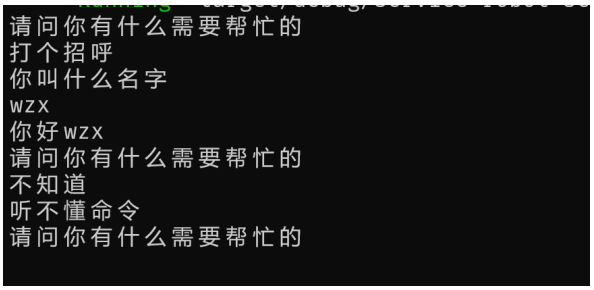

图 1: 示例

2.2　项目框架设计

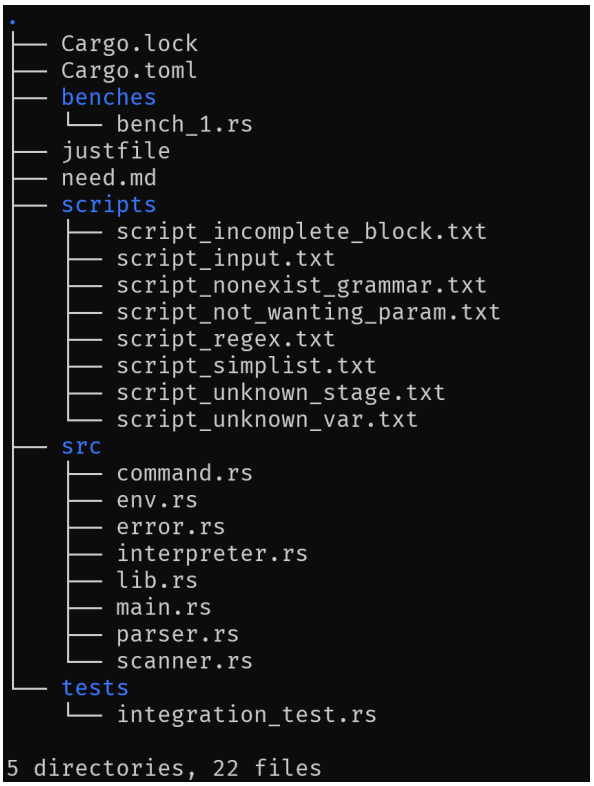

图 2: 项目框架

scripts 文件夹下存放脚本样例
tests 文件夹下存放集成测试代码
benches 文件夹下存放微基准测试代码
Cargo.toml 为 cargo 项目管理脚本，维护项目信息和依赖包
src 文件夹中 lib.rs 记录对外公开的不同 API 模块
justfile 为自动测试脚本，用于自动化支持不同测试样例的运行和解析

2.2.1　整体模块引用

下图由 cargo modules dependencies –lib –no-externs –no-fns –no-sysroot –no-traits –no-uses > mods.dot 命令生成

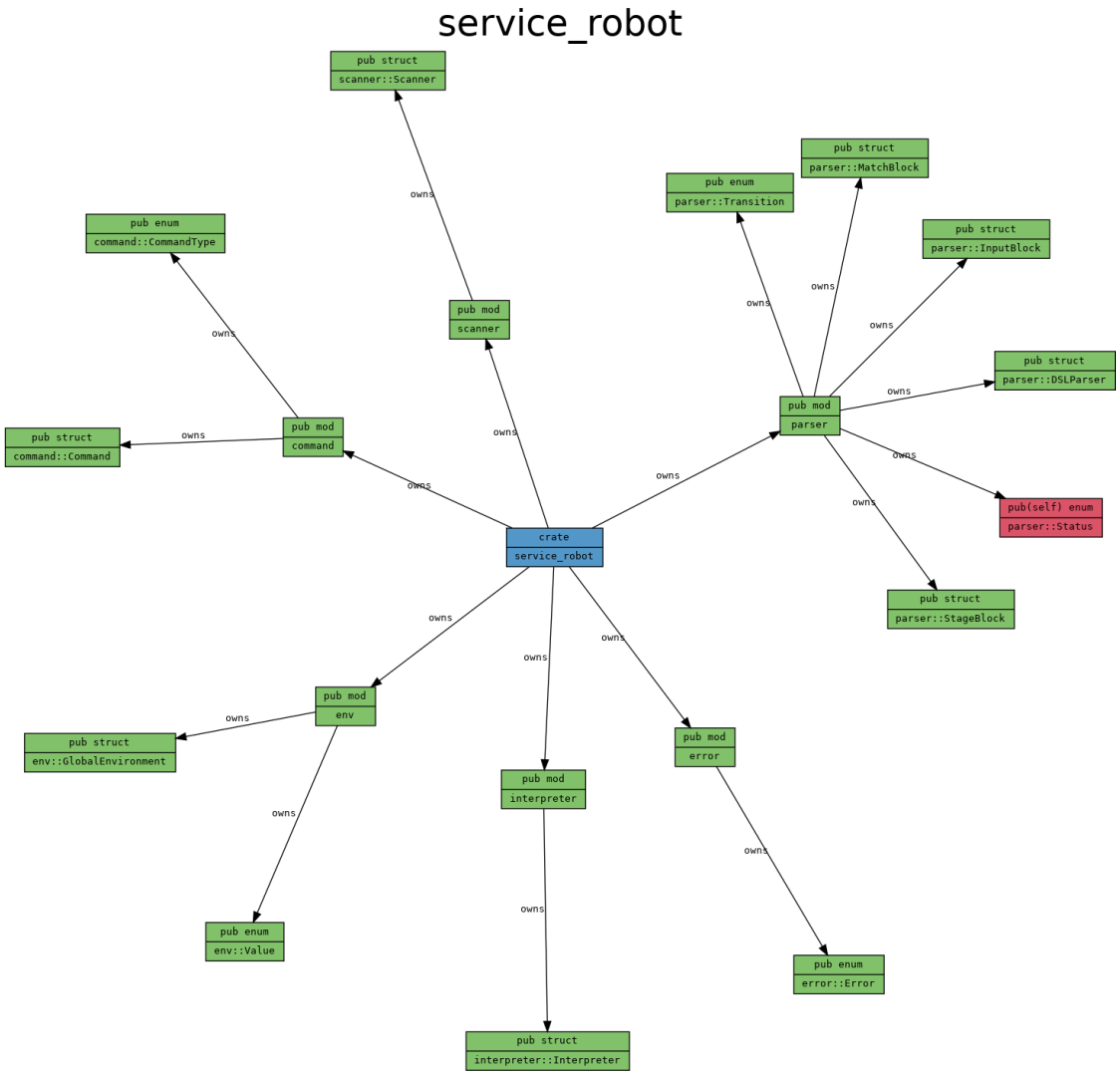

图 3: 模块图

2.2.2　将脚本语言转换为 DFA 状态迁移表

1. Command 模块中定义了 DSL 支持的命令类型与数据结构

```
#[derive(Debug, Clone, PartialEq)]
pub enum CommandType {
    MATCH(String),
    INPUT(String),
    SPEAK(String),
    NEXT(String),
    STAGE(String),
    DEFAULT,
}

///
/// Command类型, 包含命令类型和行号
```

```
///
#[derive(Debug, Clone, PartialEq)]
pub struct Command {
    /// 命令类型
    pub ctype: CommandType,
    /// 行号，在语法分析过程中适用于定位错误位置
    pub line: i32,
}
```

2. 先通过 Scanner 模块对源代码进行词法分析，得到 DSL 的 Command 命令向量

```
pub fn scan(&mut self) -> Result<Vec<Command>, Error> {
    let mut commands: Vec<Command> = Vec::new();
    for line in self.source.lines() {
        self.current += 1;
        if let Some(command) = self.scan_line(line) {
            match command {
                Ok(cmd) => commands.push(Command::new(cmd, self.current as i32)),
                Err(e) => return Err(e),
            }
        }
    }
    Ok(commands)
}
```

3. 再通过 Parser 模块将 DSL 命令向量解析为 (键为状态名，值为状态块) 的哈希表, 也即状态转移表

```
pub struct DSLParser {
    pub stages: HashMap<String, StageBlock>,
}
pub fn parse(&mut self, commands: Vec<Command>) -> Result<(), Error>
```

2.2.3 全局环境实现

若要定义一个变量, 首先要有支持该变量的运行时环境.

在 env 模块中定义了脚本支持的 Value 类型, 进一步定义了全局环境

后者的数据结构中包含当前所处状态以及当前全局变量的哈希表

```
 pub enum Value {
    /// 数值
    Number(f64),
    /// 字符串
    String(String),
}
```

```
    pub struct GlobalEnvironment {
    /// 全局变量
    pub values: HashMap<String, Value>,
    /// 当前阶段
```

```
    pub stage: String,
}
```

2.2.4 核心解释器实现

解释器通过与 Scanner、Parser 之间调用接口，对得到的状态迁移哈希表中进行处理
调用示例如下

```
let source = std::fs::read_to_string(path)?;
let mut scanner = Scanner::new(source);
let commands = scanner.scan()?;
let mut parser = DSLParser::new();
parser.parse(commands)?;
self.interpreter.interpret(&parser.stages)
```

解释器本身自带全局环境, 在状态迁移过程中随着用户输入的变化相应地对全局环境进行修改

```
pub struct Interpreter {
    /// 全局环境变量
    pub global_env: GlobalEnvironment,
}

pub fn interpret(&mut self, stages: &HashMap<String, StageBlock>) -> Result<(),
    Error> {
    loop {
        // 当stage get不到时, 输出error错误信息
        let stage = stages.get(&self.global_env.stage).ok_or_else(|| {
            self.error(&self.global_env.stage, "Runtime Error", "Stage not found
                ")
        })?;
        // 输出stage.speak,当speak内容中包含变量, 且变量未定义时, 返回运行时错误
        let speak = self.format_output(&stage.speak)?;
        // println!("DEBUG: the stage is {}", &stage.stage);
        println!("{}", speak);
        io::stdout().flush()?;
        // 判断迁移条件是输入块还是匹配块
        match &stage.transition {
            Transition::Input(input) => {
                // 输入块
                self.interpret_input_block(input)?;
                self.global_env.stage = input.next_stage.clone();
            }
            Transition::Match(match_) => {
                // 匹配块
                let match_block = self.interpret_match_blocks(match_)?;
                self.global_env.stage = match_block.next_stage.clone();
            }
        }
        if self.global_env.stage == "EXIT" {
```

```
            break;
        }
    }
    Ok(())
}
```

2.3 错误处理

针对这一特定领域, 我们自定义了程序执行过程中可能出现的几类错误, 同时提供更好的错误诊断信息输出到错误流中

```
#[derive(Debug)]
pub enum Error {
    /// 文件读取错误
    Io(io::Error),
    /// 词法错误
    Scan,
    /// 语法错误
    Parse,
    /// 运行时错误
    Runtime,
}
```

在执行过程中，程序不会单独在某一模块意外退出，而是会不断进行错误传播，将错误信息交给更高层模块进行管理，这样有助于全局化管理项目

针对不同的错误问题, 我们会提供相应错误阶段 (或错误行号) 及错误内容，帮助脚本编写者进行更一步修改完善

举例说明，下面的脚本中尝试输出未定义过的变量值

```
STAGE initial
SPEAK "尝试输出未定义的变量值" + name
MATCH EMPTY
NEXT next_stage
```

```
    Running `target/debug/service-robot scripts/script_unknown_var.txt`
[stage initial] Error (Runtime Error): Undefined variable 'name'
```

图 4: 运行结果

2.4 测试

2.4.1 测试桩

使用 Rust 语言自带的优秀测试框架

对于单元测试: 直接在模块文件底部撰写 test 测试桩, 以 scanner.rs 为例

```
#[test]
fn test_scan_line_to_unknown_command() {
    let placeholder = String::new();
```

```
    let scanr = Scanner::new(placeholder);
    println!();
    let ans = if let Some(Err(Error::Scan)) = scanr.scan_line("COMMAND THAT WE
        DON'T KNOW") {
        true
    } else {
        false
    };
    assert_eq!(ans, true);
}
```

对于集成测试: 集成测试中调用对外公开的模块接口，模拟程序运行结果

```
use service_robot::{error::Error, interpreter::Interpreter, parser::DSLParser,
    scanner::Scanner};

struct DSL {
    interpreter: Interpreter,
}

impl DSL {
    fn new() -> Self {
        Self {
            interpreter: Interpreter::new(),
        }
    }

    ///
    /// 运行DSL
    /// 根据DSL脚本文件路径，解释DSL
    /// # 参数
    /// * path: DSL脚本文件路径
    ///
    /// # 返回值
    /// * 成功返回Ok，失败返回Error
    ///
    fn run(&mut self, path: &str) -> Result<(), Error> {
        let source = std::fs::read_to_string(path)?;
        let mut scanner = Scanner::new(source);
        let commands = scanner.scan()?;
        let mut parser = DSLParser::new();
        parser.parse(commands)?;
        self.interpreter.interpret(&parser.stages)
    }
}

#[test]
fn test_run() {
    let mut dsl = DSL::new();
    let path = "scripts/script_simplist.txt";
```

```
    assert!(dsl.run(path).is_ok());
}
```

2.4.2 自动测试脚本

使用 justfile, 类似于 Makefile，但更加简洁、现代和易用。

在其中定义一系列任务和规则，可进行单独模块的自动化测试, 也可进行整体项目的自动化测试

```
test: parser_test scanner_test env_test interpreter_subfunc_test integration_test

parser_test:
    cargo test parser_test -- --test-threads=1 --nocapture

scanner_test:
    cargo test scanner_test -- --test-threads=1 --nocapture

env_test:
    cargo test env_test -- --test-threads=1 --nocapture

interpreter_subfunc_test:
    cargo test interpreter_test_subfunction -- --test-threads=1 --nocapture

integration_test:
    cargo test --test integration_test -- --test-threads=1
```

通过执行 just test 命令进行自动化测试

```
cargo test interpreter_test_subfunction -- --test-threads=1 --nocapture
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.04s
     Running unittests src/lib.rs (target/debug/deps/service_robot-273b8647c32d3671)

running 2 tests
test interpreter::interpreter_test_subfunction::test_match_blocks_with_empty_pattern ... ok
test interpreter::interpreter_test_subfunction::test_match_blocks_with_more_than_one_empty_trans
untime Error): Match pattern 'EMPTY' must be the only pattern
ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 15 filtered out; finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/service_robot-10a8afd6cca410fb)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-1885940479511d53)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 4 filtered out; finished in 0.00s

cargo test --test integration_test -- --test-threads=1
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.04s
     Running tests/integration_test.rs (target/debug/deps/integration_test-1885940479511d53)

running 4 tests
test test_parse_error ... ok
test test_run ... ok
test test_run_error ... ok
test test_scan_error ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.02s
```

图 5: 测试结果

值得注意的是，默认测试过程是多线程且无标准输出的，但为了查看实际测试内容，通过–test-thread=1 –nocapture 参数实现了更可视化的测试效果

2.4.3 多脚本范例

在 scripts 文件夹下，包含了正确性测试和错误处理测试等多个脚本，基本做到了全覆盖

2.5 性能测试

由于本 DSL 是输入事件驱动的，故设计海量微基准测试脚本的意义不大，且实现复杂

故本性能测试仅作为参考，进行的是一次单阶段的空转移测试 (更多的是我对 Rust 性能测试模块的一次探索)

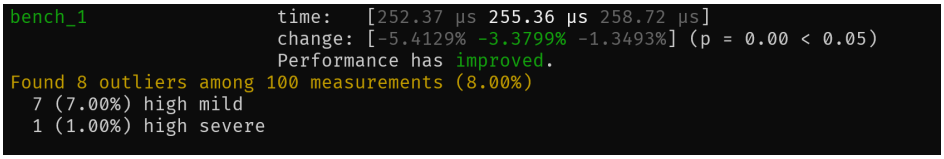

图 6: 性能

3 总结

本次实验让我认识了工程化项目中代码设计规范和编写测试框架的重要性

后续我需要进一步结合 DevOps 理念，通过 Docker 等现代化软件管理工具，实现跨平台可部署的软件项目