目录

1. 系统概述 2

2. 系统结构 2

3. 核心流程与组件 2

3.1. 命令解析 2

3.2. 命令执行 2

3.3. 错误处理 2

4. UML 图与详细设计 3

4.1. 用例图 3

4.2. 类图 3

4.3. 时序图 4

4.4. 流程图 5

5. 错误处理机制 6

6. 安装与使用指南 6

My_Shell 开发文档

1. 系统概述

My_Shell 是一个轻量级的命令行解释器，旨在提供基础的命令执行、管道操作、以及 I/O 重定向功能，模仿了类 Unix 系统 shell 的基本特性。它能够接受用户输入，解析复杂的命令结构，并通过系统调用执行这些命令，同时支持错误处理机制，以增强用户体验。

2. 系统结构

My_Shell 的核心架构围绕三大模块设计：

- 命令解析模块：负责将用户输入的命令行字符串解析为结构化的命令对象，包括命令名称、参数、以及 I/O 重定向信息。
- 命令执行模块：利用`fork()`和`execvp()`等系统调用创建子进程执行命令，同时处理命令间的管道通信。
- 错误处理模块：提供详尽的错误信息反馈，覆盖从命令解析到执行过程中的各种异常情况，帮助用户快速识别并解决错误。

3. 核心流程与组件

3.1. 命令解析

- 入口点：`main()`函数首先输出欢迎信息，随后根据命令行参数决定是直接进入脚本模式（通过`runShellScript`处理脚本文件）还是进入交互模式。
- 交互模式流程：在交互模式下，程序持续等待用户输入。用户输入的每一条命令通过`splitCommands`函数被解析成一系列`Command`对象，每个对象封装了一条待执行的命令及其相关参数和重定向信息。

3.2. 命令执行

- 执行策略：解析后的命令列表通过`executeCommands`函数执行。根据命令数量，执行逻辑有所不同：
- 单命令执行直接调用`forkToExecute`。
- 多命令情况下，通过创建管道（`pipe()`系统调用）连接命令，实现数据流的传递。

3.3. 错误处理

- 错误类型：涵盖了命令格式错误、I/O 文件格式错误、字符串引用不匹配、进程创建失败、命令执行失败以及进程异常退出等多种错误场景。
- 错误反馈：通过预定义的宏（如`ERROR_EMPTY`, `ERROR_IOFILE`, 等）提供即时、明确的错误信息，便于调试和用户理解。

4. UML 图与详细设计

4.1. 用例图

- 用例图描述：用例图展示了用户与 Simple Shell 系统之间的交互，主要包括用户输入命令、Simple Shell 解析命令、执行命令并返回结果。
- 文字描述：用户通过命令行界面与 Simple Shell 交互，输入命令并期望得到结果。My_ Shell 接收用户输入的命令，解析并执行命令，并将结果返回给用户。
- 核心用例：

  输入命令：用户向 Shell 提供命令行输入。
  解析命令：Shell 解析命令行，生成 Command 对象。
  执行命令：基于 Command 对象执行相应的系统命令。
  结果反馈：命令执行结果被输出到终端。

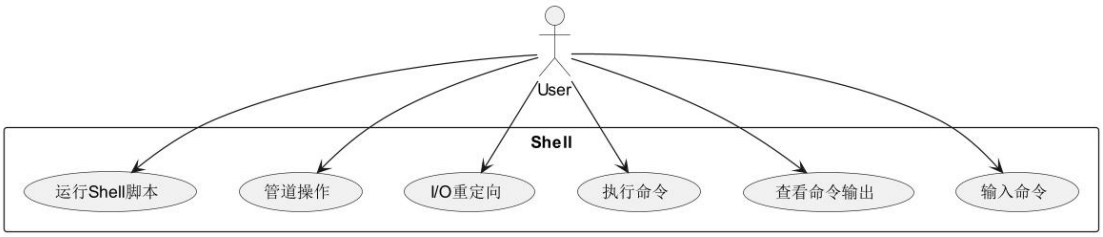

图 1 My_shell 程序用例图

4.2. 类图

- 类图描述：类图展示了 My_Shell 系统中的主要类及其属性和方法。
- 文字描述：Shell 类负责整体流程控制，包括命令的解析和执行。Command 类：表示一个命令及其参数和重定向信息。
- 关键类：

Shell：管理整个应用程序流程，包括命令的解析和执行。

属性：

- prompt：命令提示符字符串。

方法：

- run()：主运行函数，循环接收和处理用户命令。
- parseCommand(input)：解析用户输入的命令。
- executeCommand(cmd)：执行解析后的命令。

Command：表示一个命令及其参数、重定向信息，是命令执行的基础单元。

属性：

- command：命令名称字符串。
- args：命令参数列表。
- inputRedirect：输入重定向文件。
- outputRedirect：输出重定向文件。

方法：

- __init__(command, args, inputRedirect, outputRedirect)：构造函数，初始化命令对象。

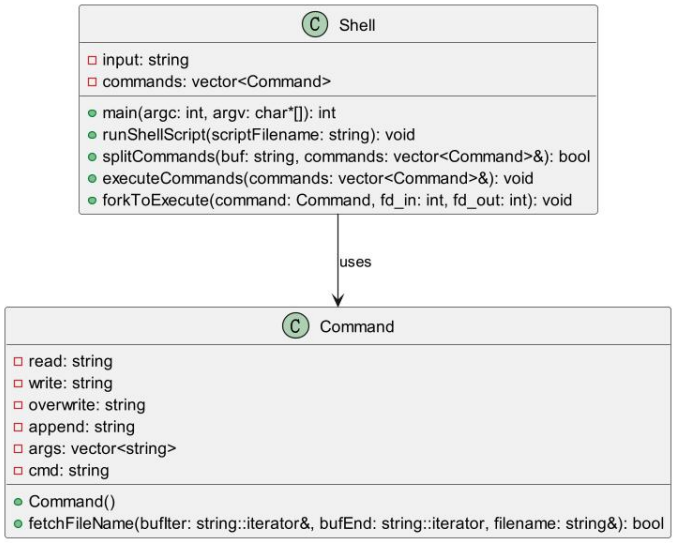

图 2 My_shell 程序类图

4.3. 时序图

- 时序图描述：描述了从用户输入到命令执行结果输出的完整时序，展现了各方法调用的顺序，以及 Shell 与操作系统（通过系统调用）的交互过程。
- 文字描述：
  - 用户：输入命令到命令行界面。
  - Shell 类的 run()方法接收用户输入，并调用 parseCommand(input) 方法解析命令。
  - Shell 类调用 executeCommand(cmd) 方法执行解析后的命令。
  - Shell 创建子进程并使用 execvp 执行命令。
  - 命令执行完成后，Shell 返回结果到命令行界面。
  - 用户看到命令的执行结果。

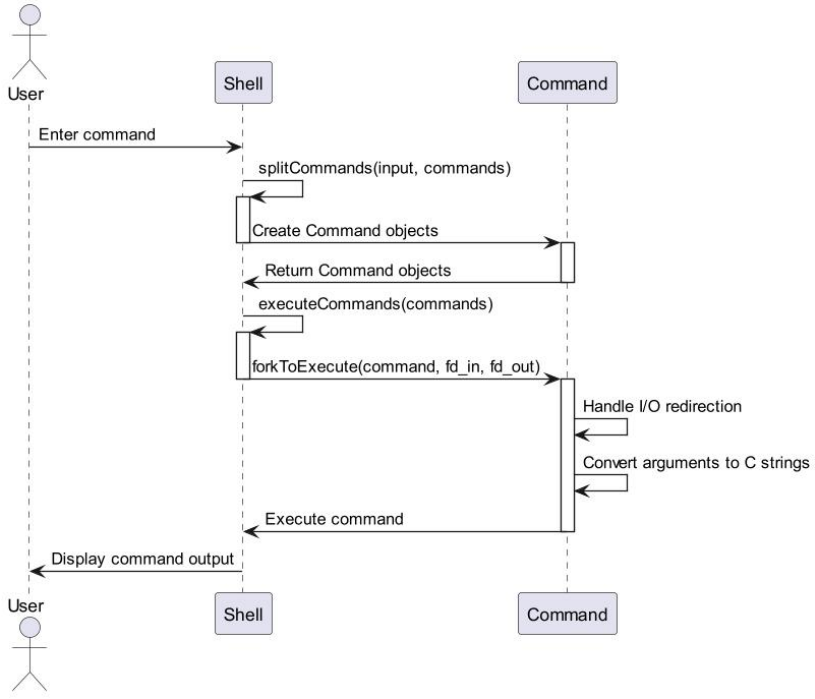

图 3 My_shell 程序时序图

4.4. 流程图

- 流程图描述：细致地展示了命令处理的每一步，从接收输入到命令解析，再到执行与结果反馈，直观反映了数据流动和控制转移过程。
- 文字描述：
  - 用户输入命令：用户在命令行界面输入命令。
  - Shell 接收输入：Shell 类的 run() 方法接收用户输入，并调用 parseCommand(input) 方法解析命令。
  - 解析命令：Shell 类的 parseCommand(input) 方法将输入的命令字符串解析成命令和参数。
  - 检查命令类型：Shell 检查命令是否包含 I/O 重定向或管道操作。
  - 创建子进程：Shell 创建一个子进程。
  - 执行命令：使用 execvp 在子进程中执行命令。
  - 等待子进程完成：Shell 等待子进程执行完成。
  - 返回结果：Shell 将命令执行结果返回到命令行界面。
  - 用户查看结果：用户在命令行界面查看命令的执行结果。

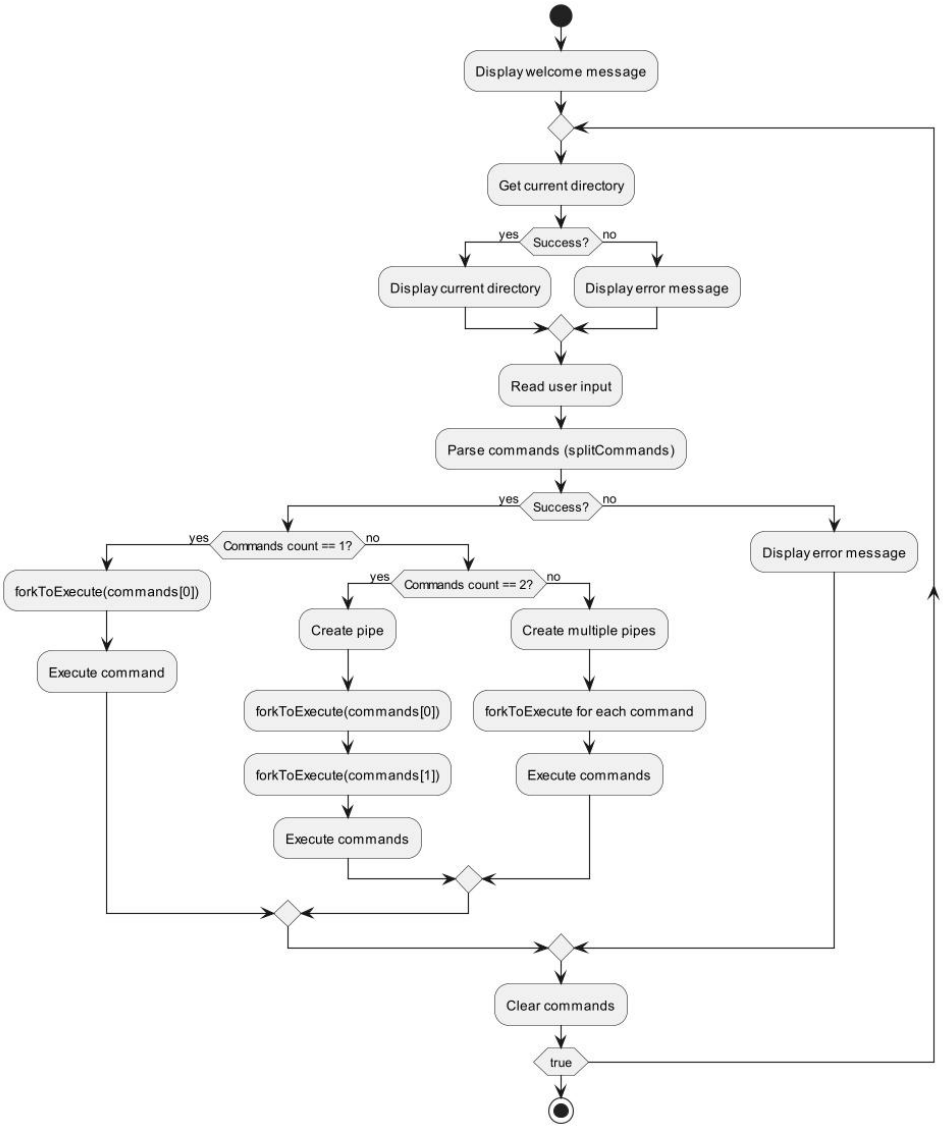

图 4 My_shell 程序顺序图

5. 错误处理机制

错误处理机制是软件开发中一个至关重要的环节，它直接影响到程序的健壮性、用户体验以及维护效率。针对“详细错误处理策略”，对以下几个方面进行更详细的说明：

- 错误分类与识别
  - 输入验证错误：对用户输入或外部数据进行严格校验，如格式不正确等。
  - 逻辑错误：程序执行逻辑中可能出现的问题，例如算法错误、状态不符合预期等。
  - 资源访问错误：文件找不到、网络连接失败等。

- 错误处理逻辑设计
  - 即时反馈：一旦检测到错误，应立即停止当前操作，并向用户反馈错误信息。
  - 精确错误码：为每种已知错误定义唯一的错误码，便于识别和后续处理。
  - 资源清理：确保在错误处理流程中释放或关闭已分配的资源，避免资源泄露。

- 用户友好的错误信息输出
  - 清晰简洁：错误信息应该简明扼要，直接指出问题所在。
  - 指导性建议：提供可能的解决方案或下一步操作建议，帮助用户快速解决问题。
  - 安全考量：避免在错误信息中泄露敏感信息，如密码、个人数据等。

6. 安装与使用指南

- 编译：确保环境中安装了 C++编译器（如 g++或 clang++），然后在命令行中运行类似 `g++ -o my_shell my_shell.cpp`的命令进行编译。
- 运行：直接执行编译后的二进制文件`./my_shell`进入交互模式，或通过`./my_shell script.sh`运行脚本文件。