基于 C++和 Linux 的 Shell 程序

计算机与信息科学学院　2021 级　软件工程　王新源、蒲雯

指导教师　龚伟

1 项目概述

本项目使用 Ubuntu 系统的 QT creator，并使用 Cmake 工具进行管理，旨在开发一个简易的 Shell 程序，结合 C++标准库和 Linux 的系统调用。它能够解析用户输入的命令行，并根据命令行中的指令执行相应的 Linux 系统调用。这包括执行基本命令、能够执行简单的 shell 脚本、能够提供 I/O 重定向和管道的功能。

2 项目小组成员分工

(1). 王新源

工作内容：

a. 共同完成了 Shell 程序的基本框架，处理用户输入并执行命令

b. 实现了 I/O 重定向和管道功能，处理命令的重定向和管道操作

c. 编写项目开发文档、录制答辩视频进行答辩

d. 上传项目到 GitHub 仓库

(2). 蒲雯

工作内容：

a. 共同完成了命令输入与执行功能，处理用户输入并执行命令

b. 实现了 Shell 脚本的解析和执行功能，支持简单的 Shell 脚本

c. 编写项目构建文档、答辩 PPT

3 项目开发状态

3.1 完成度

(1). 已完成基本的命令输入、执行以及结果显示的功能。

(2). 已完成简单的 shell 编程功能，能够执行简单的 shell 脚本。

(3). 已完成 I/O 重定向和管道功能。

3.2 质量

(1). 代码结构清晰：使用头文件和实现文件分离的方式组织代码，易于维护和扩展。

(2). 使用 CMake 进行项目管理：项目采用 CMake 构建系统，简化了编译和链接过程，提升了跨平台兼容性。

(3). 错误处理完善：对系统调用的返回值进行了严格检查，并在出错时提供详细的错误信息，方便调试。

(4). 资源管理规范：合理使用文件描述符和进程管理，确保没有资源泄露，保证了程序的稳定性。

(5). 多功能实现：不仅能够执行单一命令，还支持 I/O 重定向和管道功能，具备较高的实用性和灵活性。

3.3 核心内容及技术重难点

(1). 命令输入、执行与显示结果

a.使用 C++标准库实现命令输入，结合 Linux 系统调用 fork 和 execvp 实现命令执行，

b.通过 waitpid 等待子进程结束，并捕获其退出状态。

技术难点：正确处理子进程的创建与终止，防止僵尸进程的产生。

(2). Shell 编程功能

解析简单的 shell 脚本并逐行执行。

技术难点：设计一个高效的脚本解析器，处理不同类型的命令及其参数。

(3). I/O 重定向和管道功能

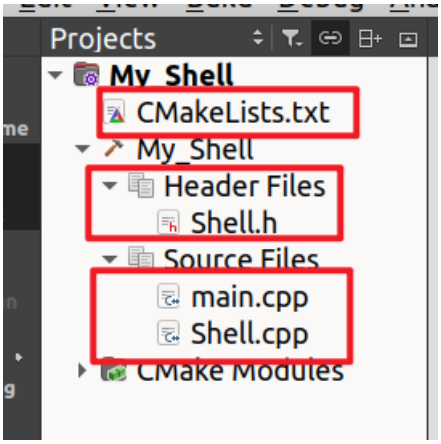
a.通过 dup2 实现文件描述符的重定向，支持输出重定向>和输入重定向<。

b.使用 pipe 系统调用实现管道操作，支持命令之间的管道|。

技术难点：正确管理文件描述符，防止资源泄露和死锁情况。

4 项目的基本情况

4.1 目录结构



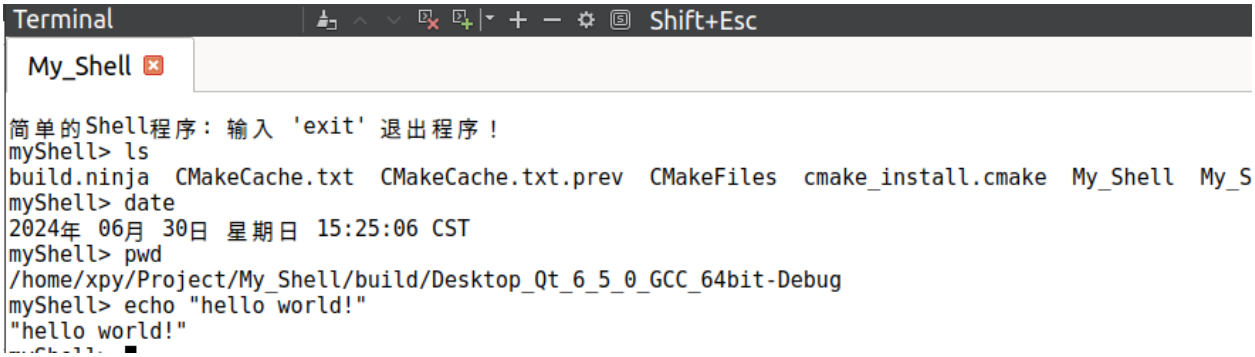
4.2 基本命令输入、执行与显示结果

核心代码：

```
// 分割字符串函数，用于处理输入的命令行
vector<string> split(const string& input, char delimiter) {
    vector<string> tokens;
    string token;
    istringstream tokenStream(input); // 使用输入字符串创建一个字符串流
    while (getline(tokenStream, token, delimiter)) { // 从字符串流中按分割符读取字符串到token
        tokens.push_back(token);
    }
    return tokens;
}

// 执行具体命令的函数
bool executeCommand(const vector<string>& args) {
    vector<char*> argv;
    for (const auto& arg : args) {
        argv.push_back(const_cast<char*>(arg.c_str())); // 将string转换为char*并添加到数组
    }
    argv.push_back(nullptr); // 参数列表以nullptr结束

    pid_t pid = fork(); // 创建子进程
    if (pid == 0) {
        // 子进程中
        execvp(argv[0], argv.data()); // 执行命令
        perror("execvp"); // 如果execvp返回，说明出错了，输出错误信息
        exit(EXIT_FAILURE); // 出错则退出该子进程
    } else if (pid > 0) {
        // 父进程中
        int status;
        waitpid(pid, &status, 0); // 等待子进程结束
        return WIFEXITED(status); // 返回子进程的退出状态
```

```
    } else {
        // fork 失败
        perror("fork");
        return false;
    }
}
```

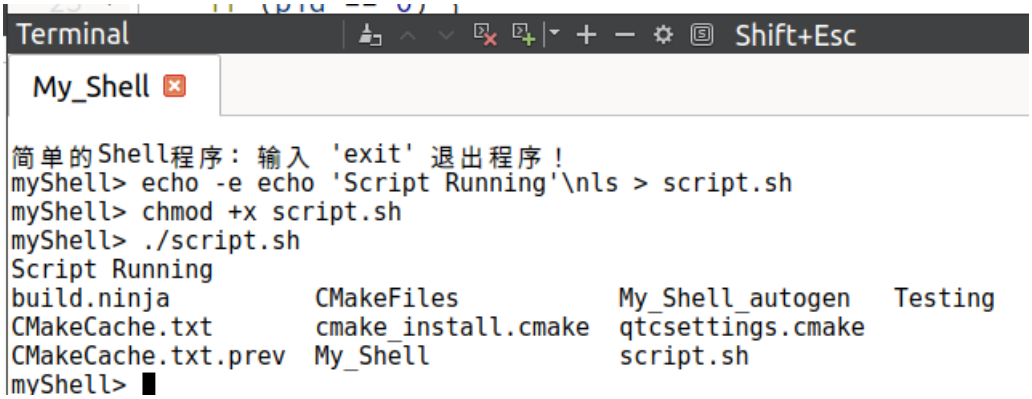
测试结果展示：



4.3 简单的 shell 编程功能，能够执行简单的 shell 脚本

代码以上已展示

测试结果展示：



上图，第一行创建一个脚本

第二行给脚本文件赋予权限，以便下面执行该脚本文件

用./来执行脚本文件，输出结果显示，正确执行

4.4 I/O 重定向和管道功能

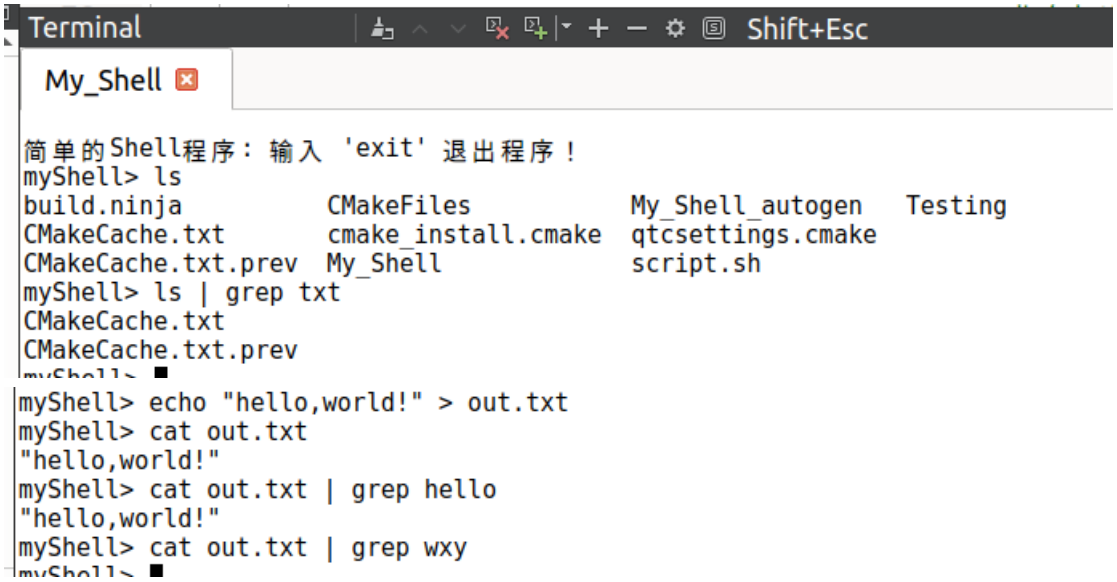
核心代码展示：

```
// 处理输入命令的函数，识别重定向和管道操作
void processCommand(const string& input) {
    auto tokens = split(input, ' '); // 使用空格来分割输入的命令
    if (tokens.empty()) return; // 命令为空则直接返回

    vector<string> command;
    for (size_t i = 0; i < tokens.size(); ++i) {
        if (tokens[i] == ">") {
            // 处理输出重定向
            command.resize(i);  // 重新调整命令的长度，只保留重定向前的命令部分
            int fd = open(tokens[i+1].c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0666);
            if (fd == -1) {
                perror("open");
                return;
            }
```

```
            int saved_stdout = dup(STDOUT_FILENO);
            dup2(fd, STDOUT_FILENO); // 将标准输出重定向到文件
            executeCommand(command); // 执行命令
            dup2(saved_stdout, STDOUT_FILENO); // 恢复标准输出
            close(saved_stdout);  // 确保关闭保存的stdout文件描述符
            close(fd);
            break;
        } else if (tokens[i] == "|") {
            // 处理管道
            command.resize(i);  // 将管道符号前的部分作为要执行的命令
            int pipefd[2];
            pipe(pipefd); // 创建管道，pipefd[0]为读端，pipefd[1]为写端

            int saved_stdout = dup(STDOUT_FILENO);
            dup2(pipefd[1], STDOUT_FILENO); // 将标准输出重定向到管道的写端
            close(pipefd[1]);  // 关闭写端的原始文件描述符，防止额外的引用计数

            executeCommand(command); // 执行管道左侧的命令，其输出会写入管道

            dup2(saved_stdout, STDOUT_FILENO); // 恢复标准输出到原来的文件描述符
            close(saved_stdout);

            int saved_stdin = dup(STDIN_FILENO);
            dup2(pipefd[0], STDIN_FILENO);
            close(pipefd[0]);  // 关闭读端描述符的额外副本，防止额外的引用计数

            command = vector<string>(tokens.begin() + i + 1, tokens.end()); // 从管道符号后的第一个命令开始作为新的命令
            executeCommand(command); // 执行管道右侧的命令，其输入来自管道
```

```
            dup2(saved_stdin, STDIN_FILENO);
            close(saved_stdin);
            break;
        } else {
            command.push_back(tokens[i]);
        }
    }

    if (command.size() == tokens.size()) {
        executeCommand(command);  // 没有重定向或管道，直接执行完整命令
    }
}
```

测试结果：



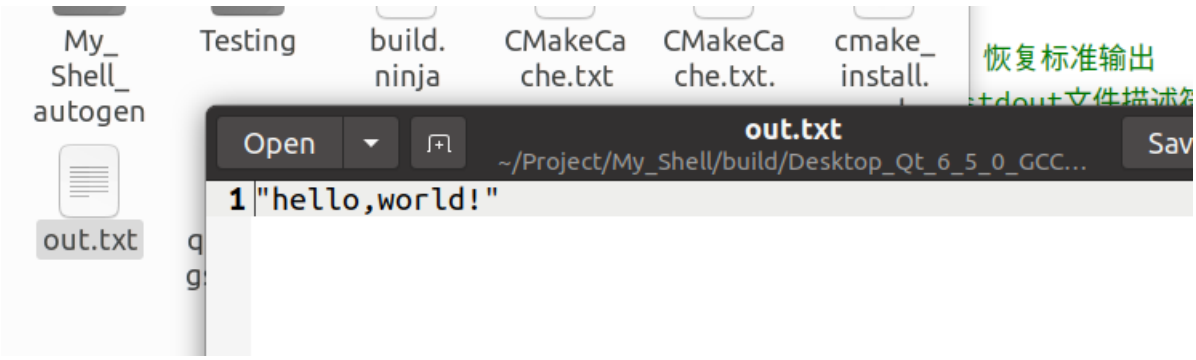
上图将“hello,world!”输出重定向到文件

再读取文件内容，可以正确显示

接下来通过管道过滤，将含“hello”的内容输出

下一句因为文件中无“wxy”内容，所以输出结果未展示任何内容

out.txt 文件内容展示：



5 收获及经验教训

5.1 收获

(1). Linux 系统调用：

深入理解并掌握了 fork、execvp、pipe、dup2 等系统调用的用法，特别是在进程创建与管理、文件描述符操作等方面的应用。

(2). C++标准库：

a.熟练使用 C++标准库进行字符串处理（如 std::istringstream 和 std::getline），提高了对 C++标准库的应用能力。

b.掌握了 C++的流操作和向量操作（如 std::vector），提高了编程效率。

(3). 项目构建和管理：

学习并使用了 CMake 进行项目构建和管理，简化了编译过程，提高了项目的可移植性和维护性。

(4). Shell 编程：

了解了 Shell 脚本的基本语法和执行流程，能够编写和解析简单的 Shell 脚本。

5.2 经验教训

(1). 技术提升

a.学习并掌握了 Linux 系统调用的使用，特别是进程管理和文件描述符操作。

b.加强了对 C++标准库的应用能力，尤其是在字符串处理和流操作方面。

(2). 实践能力

a.通过项目实践，发现并解决了多种实际编程问题，提高了代码调试和错误处理能力。

b.提高了设计和实现复杂功能模块的能力，增强了解决实际问题的能力。

(3). 团队合作

a.在项目中分工合作，相互配合，提高了团队合作和沟通能力。

b.通过相互代码评审和讨论，提升了代码质量和开发效率。

(4). 问题处理

a.在实现过程中遇到了多种技术难题，例如管道和重定向的实现，通过查阅资料和反复调试，最终成功解决。

b.锻炼了面对问题时的分析和解决能力，积累了丰富的编程经验。