计算机系统原理

大作业报告

经56 王思萍 2015012527

MY SHELL

My Shell 是一个基于 `MAC` 及 `Linux` 系统的命令行解析器。

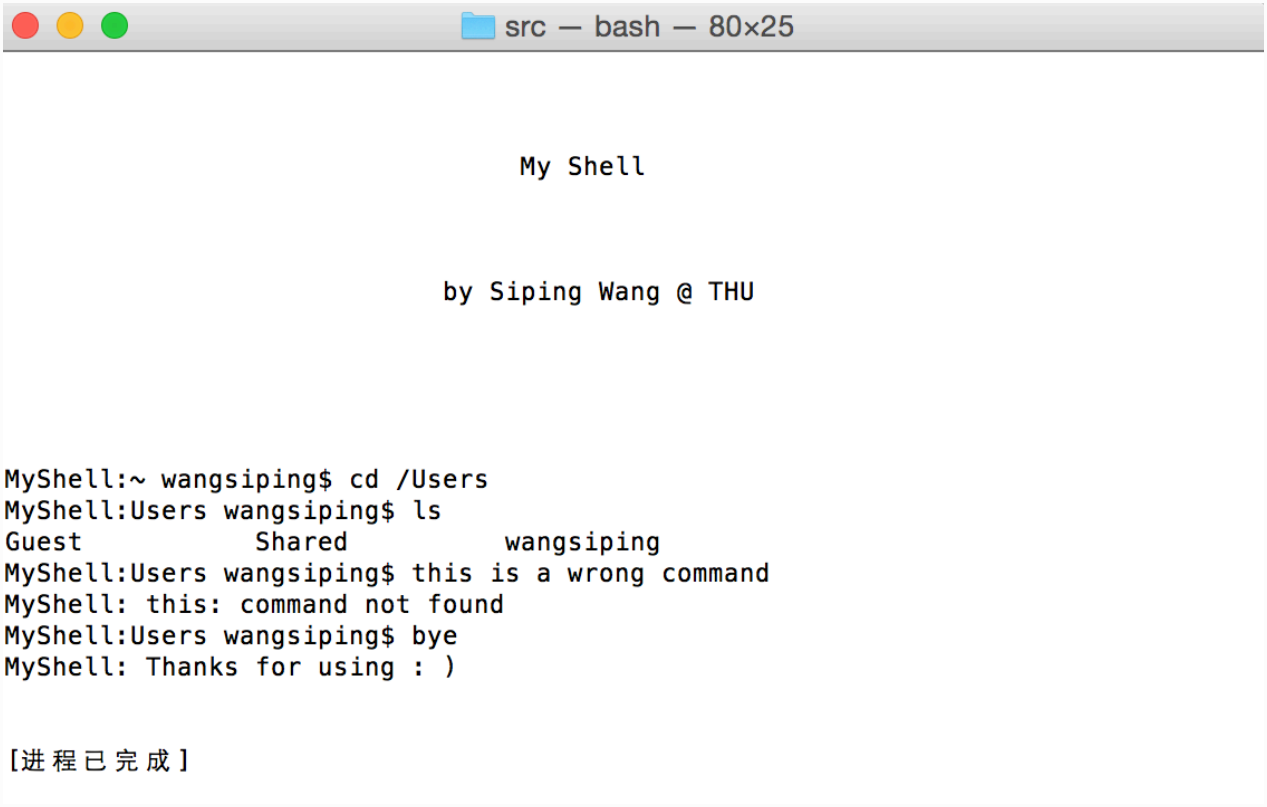
本文将从以下几个方面介绍这个程序的设计思路和整体架构：

- 主要功能和特色
- 算法设计
    - 整体架构
    - 模块介绍
- 主要技术难点及实现方案
- 文件结构
- 不足与改进

1 主要功能和特色

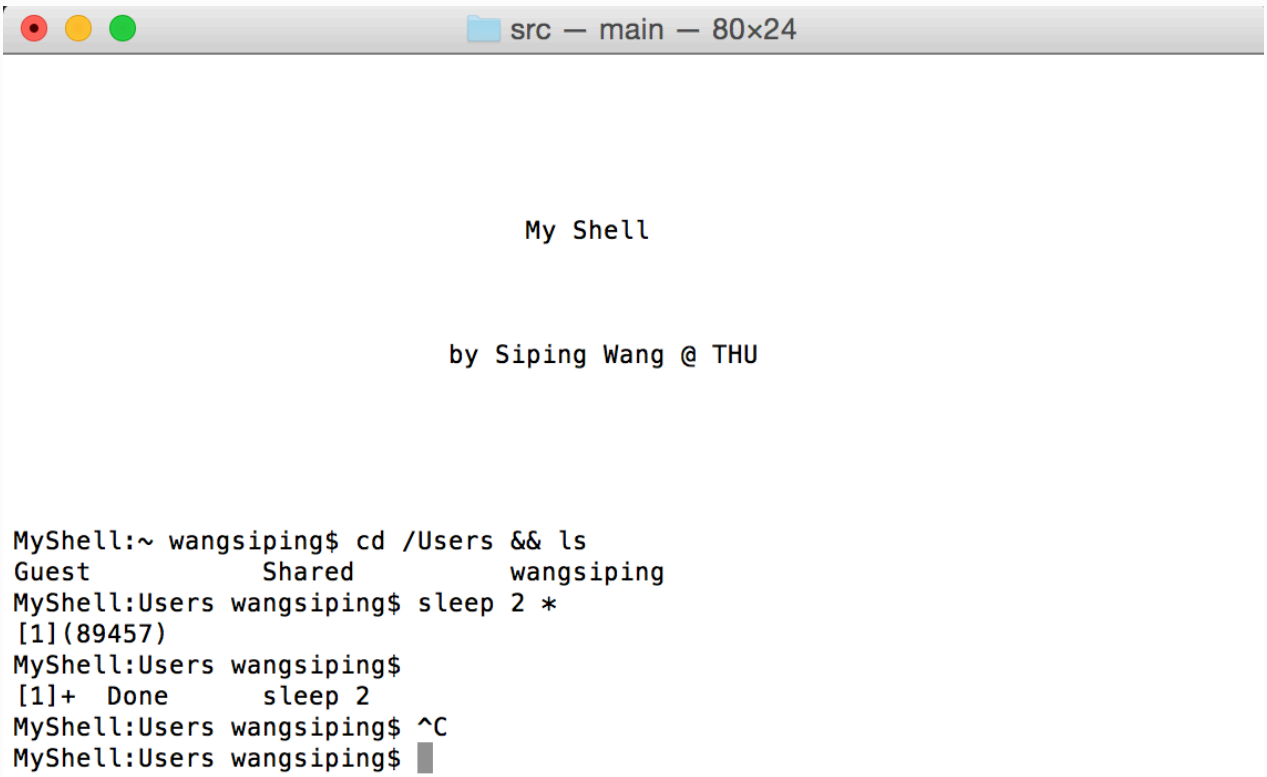
1.1 基本功能

- 从键盘读取输入的命令行语句
- 执行命令（如输入指令不合法则报错）
- 重复输入
- 输入 `bye` 时退出程序



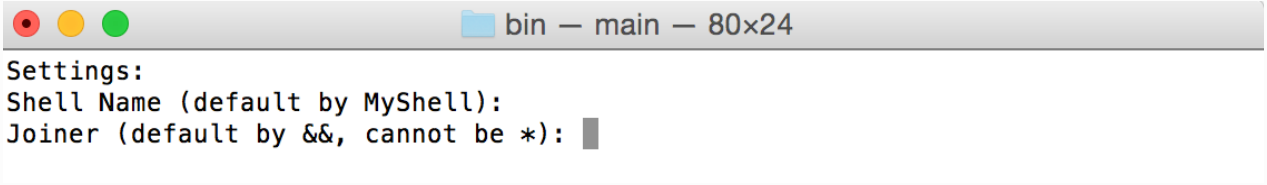
1.2 高级功能

- 能够一次执行多条指令，用 `&&` 连接
- 结尾为 `*` 时，将指令放入后台执行
- 从键盘输入 `ctrl - c` 时，不终止程序



1.3 增设功能

- 能够让用户自定义设置 Shell 名称与命令连接符



- 程序启动时，首先将工作路径定位到用户的根目录下
- 程序启动时，能够自动识别用户名并在每次输入时打印出来
- 执行 `cd` 指令更换工作路径后，将新的目录名称显示在用户名之前
- 命令进入后台运行时，输出 (任务序号)(进程号)；命令在后台运行结束后，输出 (任务序号)(进程号) Done + `命令内容`

2 算法设计

2.1 整体架构

实现这个 shell 程序的包含了以下三个部分的功能：

- 程序从键盘读取一行输入的指令字符串
- 处理读取的指令字符串，形成能够通过调用函数执行的指令集合
- 按顺序执行这些指令

第二部分 (指令处理) 中，需要判断该指令是为前台 (foreground) 还是后台 (background) 运行，进入第三部分 (指令执行) 中将会分为两种情况讨论。

第三部分 (指令执行) 中，大部分指令能够由 `<unistd>` 库中的 `exec族函数` 执行，需要自己添加两个命令的执行函数，一个是 `cd` 系列的命令，另一个是退出程序的命令 `bye` 。

基于以上设计，采用 C++ 语言实现这个程序。程序中有一个 `Shell` 类，负责 My Shell 的各项功能及运行。2.2 中将对 `Shell` 类进行相似介绍。

2.2 模块介绍

Shell 类

- 数据成员

类型	变量名	含义	用途
istream	in	输入流	输入
ostream	out	输出流	输出
string	name	根目录文件夹名称	用户名
string	cmdstring	命令字符串	以字符串方式储存从键盘读取的命令
vector<string>	vcmd	以动态数组为基础	分段进行命令的顺序执行

		的命令**队列**	
string	workPath	工作目录	定位程序的工作目录，输出文件夹名称
string	defaultPath	默认目录 (用户根目录)	存储程序的默认目录
bool	background	任务是否在后台执行	判断用户输入的命令是否要在后台执行
int	count	参与后台运行的任务编号	执行后台命令时与完成后输出任务编号
string	shell_name	命令解析器名称	用于输出输入提示信息，默认为 MyShell
string	joiner	命令连接符	用于命令的连接，默认为 &&
string	fpath	文件路径	该文件用于储存已经完成但尚未输出的后台运行任务信息

- 函数成员

Shell (istream&, ostream&)

- 构造函数，分别传入一个输入、输出流参数。在本题中，输入流用 `std::cin` 表示，输出流用 `std::cout` 表示。同时进行一些类成员的初始化：将 `background` 设为 false，将 `count` 设为 0，将 `name` 设为默认的 user，并设置工作路径 `work path` ；同时打开 `output` 文件，清空该文件。

void init ()

- My Shell 的初始化函数。首先输出欢迎界面的 UI，而后设置初始工作路径 (即用户根目录) 并根据路径确定用户名称。将用户根目录存储至 `defaultPath` 中，将用户名存储至 `name` 中。
- 之所以将这个函数的功能与构造函数分开写，是因为如果不执行这个操作 shell 依旧可以正常运行，只是初始工作路径为程序所在文件夹，无默认路径。

string getDirName ()

- 返回程序工作路径的文件夹名称。用于打印输入提示信息使用。

void run ()

- 运行 Shell 的函数。负责从键盘读取用户输入的命令字符串，并调用 `split` 函数与 `parseCommand` 函数预处理这个字符串。每次检测 `parseCommand` 函数的返回值，若为 true，则让用户继续输入；若为 false，则停止输入，并退出程序。
- 同时该函数还将打开 `output` 文件，查看是否有尚未输出的进程结束信息，若有，则全部输出，并清空该文件。

void split (string&, char)

- 处理命令的函数之一。传入两个参数，第一个为 string 类的引用，为用户从键盘输入的指令字符串，第二个为 char 类型参数，为分隔符。函数将 `参数1` 以 `参数2` 分割成若干的短字符串，并按顺序存放进动态数组 `vcmd` 中。

bool parseCommand ()

- 处理命令的函数之二。返回值为 bool 类型，若为 true，则用户还可以继续输入命令；若为 false，则退出程序，用户不可再输入命令。该函数负责对短命令集 `vcmd` 进行初步处理：

  1）判断 `vcmd` 是否为空，若是则直接返回 true

  2）判断命令是否以 `bye` 或 `exit` 开始，若是则返回 false

  3）判断命令是否以 `joiner` 或 `*` 开始，若是则输出报错信息并返回 true

  4）判断命令是否以 `*` 结束，若是则将 `background` 设为 true，修改 `count` 值，并执行指令，返回true

string rebuildCommand ()

- 处理命令的函数之三。这个函数用于检测 `vcmd` 中可以调用 `exec_command` 函数执行多少个短命令。主要有以下功能：

  1）检测初始命令是否是 `joiner`，若是则跳过

  2）依次将 `vcmd` 中能够执行的命令合并成字符串，用空格连接，同时将 `vcmd` 中被合并了的短命令出队，直到命令结束，或遇到 `cd` 或 `joiner` 为止。

void bye ()

- 自定义的 bye 函数，输出结束语。

int cd (vector<string>&)

- 自定义的 cd 函数，用于切换父/子进程的工作路径。传入参数为一个 string 类的 vector 的引用，若 vector 中只有 `cd` 而没有下一个参数，则切换至默认路径（`defaultPath`）；若有参数，则切换至该参数对应的路径。
- 若该命令不是在后台执行的，则更新数据成员 `workPath`；反之则不更新，因为父进程的工作路径没有被切换。

bool exec_cd ()

- 执行字符串命令的函数之一，只负责执行上述 cd 函数。当 `vcmd` 中的第一个元素为 `cd` 的时候调用 cd 函数并将执行后的 `cd` 及其参数 (若有) 出队。`vcmd` 被更新。

int exec_command (string&)

- 执行字符串命令的函数之二，只负责执行除 cd 系列命令以外的命令。创建一个子进程并在其中调用 `execl` 函数执行命令。在父进程中，使用 `waitpid` 函数来等待进程的状态发生变化，使用 `waitpid` 提供的宏来等待进程退出或终止。函数返回一个 `status` 的值，若为 0，则说明命令执行成功，若为 -1，则说明命令未成功执行。

void execute ()

- 执行字符串命令的函数之三，将 exec_cd 和 exec_command 两个函数整合，即可执行全部输入的命令。首先根据已经被设置过的 `background` 值判断命令是否需要在后台执行。若是在后台执行，则创建一个子进程并在子进程中执行命令；执行结束后，将结束信息追加写入到 `output` 文件中。
- 执行命令的过程为：顺序执行，碰到 `cd` 系列命令则调用 exec_cd 函数执行，若执行失败直接返回；执行 `cd` 命令后，再调用 exec_command 函数执行之后的命令；以此类推直至 `vcmd` 为空 (所有短命令全部出队/被执行完) 为止。

3 主要技术难点及实现方案

3.1 命令中包含多次 `cd` 系列命令

如果命令中没有 `cd` 系列命令，单纯调用 execl 函数，将从键盘读取的命令字符串传入即可执行。但由于 `cd` 系列的命令必须自己实现，那么就只能按顺序读取每一段命令，判断即将被执行的命令是否为 `cd` 系列的命令，直到所有命令被执行完为止。

于是将用户输入的命令字符串按照空格截断分割后，按顺序存入一个**队列**里。对于队列中的第一个指令，判断它是否为 `cd` 或 连接符（`joiner`），再进一步判断后面是否有 `cd` 的参数；对于每一段被执行完或者被跳过（如连接符）的字符串，让其出队，于是**当所有的元素均成功出队后，说明全部命令已被执行完。**

对于这个指令队列的创建，可以选择 `c++ STL模板库` 中的 `queue<string>` 来实现，但由于程序的其他函数中涉及到对指令队列的随机访问、删去最后一个元素以及清空等操作，仅仅用队列不方便实现，因此我最后还是选择了用 `vector<string>` 为基础，实现队列的功能。例如入队操作可以用 push_back() 实现，而出队操作可以用 erase(vcmd.begin()) 实现。

3.2 命令的后台运行

对于这个功能的实现，我增加了 Shell 类的数据成员 `background` 和 `count` 。对于传入的命令字符串，将其分割并按顺序入队后，首先判断队尾元素是否为 `*` 。若是，则把 `background` 设为 true，并删去队尾元素（`*`），反之则把 `background` 设为 false。这样能够保证在之后的执行过程中，命令是否需要在后台执行并不影响命令的内容（队列元素），因此 exec_cd 和 exec_command 就不需要对两种情况分别来实现了，只需要在整合二者的 execute 函数中对 `background` 进行分情况讨论即可。

对于子进程结束后的信息输出，选择了使用文件来解决进程间通信的困难。将进程结束的信息直接写入文件中，每次控制台输出输入提示信息之前均先检查一遍 `output` 文件中是否有未输出的结束信息，若有则输出，并清空该文件。

3.3 对 `ctrl-c` 的屏蔽

在查阅相关资料后得知，可以使用 signal 函数屏蔽 `ctrl-c` 发出的 `SIGINT` 信号。具体实现方法为：

```
#include <signal.h>
signal(SIGINT, SIG_IGN);
```

4 文件结构

根目录

- `bin`：可执行文件
  - `main`：Shell 可执行文件
  - `output`：用于存储尚未输出的进程结束信息的文件
- `src`：所有源代码
  - `main.cpp`：主函数文件
  - `myshell.cpp`：`Shell` 类的实现文件
  - `myshell.h`：`Shell` 类的头文件
- `doc`：所有文档
  - `report.md`：大作业报告的 md 格式文件
  - `report.pdf`：大作业报告的 pdf 格式文件
  - `requirement.md`：大作业需求文件
  - `1.png`：基本功能展示图
  - `2.png`：高级功能展示图
  - `3.png`：增设功能部分展示图

5 不足与改进

- 没有实现通过上、下键切换历史命令的功能
- 没有实现 `tab` 键自动补全的功能