

一、设计需求

本题要求写一个简单的shell程序——MyShell。要求如下：

1. 支持以下命令：

命令	解释
bg	将挂起的作业移至后台运行
cd	改变当前目录，同时更新PWD环境变量
clr	清屏
dir	列出指定目录下的内容
echo	在屏幕上显示指定内容并换行
exec	执行外部程序。该命令会替换掉MyShell的代码段，执行成功后直接退出
exit	退出MyShell
fg	将挂起或后台运行的作业转移至前台
help	显示用户手册
jobs	显示当前被挂起和后台运行的作业
pwd	显示当前目录
set	列出环境变量
shift	左移参数。\$0保持不变，\$1、\$2等变量被左移
test	进行文件、数值、字符串测试
time	显示当前时间
umask	显示或改变umask的值
unset	删除环境变量

- 2. MyShell的环境变量应包含SHELL = <PathName>/MyShell，表示可执行程序MyShell的完整路径。
- 3. 其他命令行输入表示程序调用，程序的执行环境应包含环境变量PARENT = <PathName>/MyShell。
- 4. MyShell能够从文件中提取命令并执行，例如命令MyShell BatchFile表示从BatchFile中获取命令并执行。
- 5. 支持I/O重定向。且>表示创建输出文件，>>表示追加到文件末尾。
- 6. 支持后台运行。若命令末尾是&，则表示该命令在后台运行。MyShell在加载完该命令后必须立刻返回命令行提示符。
- 7. 支持管道符“|”。
- 8. 命令行提示符包含当前路径（即模仿bash的命令行提示符）。

二、扩展功能

除了题目要求的内容以外，笔者还额外实现了以下功能：

1. `cd`命令若不加参数，则表示进入到用户主目录。
2. `echo`命令支持变量引用。例如`echo $0`，`echo ${11}`，`echo $PWD $HOME`等等。还可以用`'$#'`显示全局参数个数。
3. `set`命令无参时表示列出所有环境变量，若有参数，则表示给`$1`、`$2`等变量赋值。
例如执行`set 0 1 2 3`后，全局变量更新为：`$0`不变，`$1=0`，`$2=1`，`$3=2`，`$4=3`。且`set`命令后面不一定必须加常量，也可以使用`'$'`代表变量。例如`set $1 $PWD $#`。
4. `test`命令支持变量引用。例如`test $# -ge 3`，`test $PWD = /home`，`test -x $0`等等。
5. 支持快捷键操作。按`Ctrl+C`可以终止当前正在前台运行的命令。按`Ctrl+Z`可以挂起当前正在前台运行的命令。
6. 支持更多重定向。`"0<"`和`"<"`表示输入重定向。`"1>"`和`">"`表示覆盖模式的输出重定向，`"1>>"`和`">>"`表示追加模式的输出重定向。`"2>"`表示覆盖模式的错误信息重定向，`"2>>"`表示追加模式的错误信息重定向。
7. `MyShell`能够判断输入输出是否来自终端。当输入不是来自终端时，则不会输出命令行提示符，也不会即时打印已完成的后台作业信息。若输入来自终端，但输出被重定向到非终端（例如普通文件），`MyShell`仍会把已完成的后台作业信息即时输出到终端（而不是输出到重定向的文件）。

三、实验原理

在笔者的设计中，命令以换行符`'\n'`分隔：

```
1 Command1
2 Command2
3 Command3
4 .....
```

每个命令有若干参数，参数以空格`' '`或`tab`符`'\t'`分隔。每次读入一行命令，保存到`string CMD`变量中，然后再根据空格和`tab`符将其分割成若干参数，保存到数组`string cmd[]`中，并设定一个变量`int ParaNum`表示参数的总个数。然后再进行后续的处理。

我们首先从最简单的单条命令（不含管道符和重定向，不含作业控制指令`bg`、`fg`、`jobs`）开始：

1. 单条命令（不含管道符和重定向，不含作业控制指令`bg`、`fg`、`jobs`）

1. `cd`

◦ 命令格式：

`cd`

`cd 目录`

◦ 实现原理：

函数`chdir`可以改变当前目录，返回为0时表示成功，非0时表示失败。

无参执行`cd`命令时表示进入用户的主目录，主目录可以通过`getenv`函数得到`HOME`变量的值，就是主目录的路径。

每次改变路径后需要更新`PWD`环境变量的值，实现方式是调用`getcwd`函数获得当前路径的字符串，然后调用`setenv`更新`PWD`的值。

```
1 //cd命令，若无参数则进入主目录，否则进入参数对应的目录
```

```

2 void _cd(string cmd[], int ParaNum){
3     if (ParaNum == 0 || ParaNum == 1 && cmd[0] == "~"){
4         进入主目录
5         更新PWD
6         成功信息处理
7     }else if (ParaNum == 1){//参数个数为1
8         进入cmd[0]对应的目录
9         if(调用失败){
10            错误信息处理
11        }else{//chdir调用成功
12            更新PWD
13            成功信息处理
14        }
15    }else{//参数过多, 报错
16        错误信息处理
17    }
18 }

```

2. clr

- 命令格式:

clr

- 实现原理:

直接调用system("clear")就能实现清屏。

3. dir

- 命令格式:

dir

dir 目录

- 实现原理:

无参执行dir指令时表示列出当前目录下的文件。否则dir列出参数指定的目录下面的文件。

首先使用opendir函数打开指定目录，若函数返回NULL时表示打开错误，否则返回一个DIR *类型的指针。再利用readdir函数读取目录内容，readdir每调用一次，就返回一个struct dirent *类型的指针，存储文件的信息。当readdir返回NULL时，说明目录下的所有文件都已经被遍历一遍。

```

1 //dir列出指定文件夹内的文件
2 void _dir(string cmd[], int ParaNum){
3     if (ParaNum >= 2){
4         参数过多, 报错
5     }else{
6         //若无参, 则列出当前目录下的文件
7         if (ParaNum == 0) Directory = PWD;
8         //否则, 列出参数指定的目录下的文件
9         else Directory = cmd[0];
10        DIR *pDir;
11        struct dirent* ptr;
12        //打开目录
13        if((pDir = opendir(cmd[0].c_str())) == NULL){
14            打开文件夹失败, 处理报错信息
15            return;
16        }
17        //读取文件信息

```

```

18         while((ptr = readdir(pDir)) != NULL)
19             输出ptr->d_name;
20         closedir(pDir);
21     }
22 }

```

4. echo

- 命令格式:

echo arg₁ arg₂ arg₃ ... arg_n

- 实现原理:

对于每个待输出的参数，首先判断arg_i是否是以'\$'开头的字符串。如果是形如"\$0""\$1""\${10}"这样的字符串，则替换对应的值（根据main函数的参数char argv[]获得对应的值）；如果是"\$#"则替换成参数个数（等于main函数的参数int argc减一）；其余情况，根据getenv函数取得对应的环境变量值，若环境变量不存在则替换成空字符串。

若不以'\$'开头，则原样输出原字符串。

```

1 //根据字符串var，取得对应变量的值
2 string GetValue(const string &var){
3     if (var[0] == '$'){//以$开头，替换成变量的值
4         if (var是形如"$0"、"$1"、"${10}"这样的字符串)
5             return argv[对应的编号];
6         }else if (var == "$#"){//$#表示参数个数
7             return to_string(argc - 1);
8         }else{//否则，利用getenv获得变量的值
9             char * str = getenv(var.substr(1).c_str());
10            if (str != NULL)//环境变量存在
11                return str;
12            else return "";
13        }
14    } else//否则，不以$开头，则直接输出字符串
15        return var;
16 }

```

然后利用GetValue函数实现echo指令：

```

1 //echo打印内容到屏幕
2 void _echo(string cmd[], int ParaNum){
3     //遍历每个参数
4     for (int i = 0; i < ParaNum; i++){
5         string Value = GetValue(cmd[i]);
6         if (Value != "") Message1 += Value + " ";
7     }
8     //换行
9     if (Message1.length() > 0)
10        Message1[Message1.length() - 1] = '\n';
11    else Message1 = "\n";
12 }

```

5. exec

- 命令格式:

exec 程序名 参数1 参数2 参数3 ... 参数n

- 实现原理:

unistd.h提供了很多能够实现exec指令的函数：execl、execv、execle、execve、execlp、execvp、fexecve。具体参考《UNIX环境高级编程》第199页。笔者使用execvp函数，函数原型如下：

```
1 | int execvp(const char * filename, char * const argv[]);
```

filename是要执行的程序名（路径）。argv是参数列表，以NULL结尾。

exec并不创建新进程，而是用新的程序完全替换原来的程序，并从头开始执行。所以execvp调用成功后就会直接退出程序。若程序执行到execvp后面的语句，则一定说明execvp调用失败。

```
1 | //执行exec指令
2 | void _exec(string cmd[], int ParaNum){
3 |     if (ParaNum == 0){
4 |         参数太少，处理报错信息
5 |     }
6 |     准备execvp的参数列表
7 |     execvp(参数列表);
8 |     //execvp执行成功后会退出源程序。如果能执行到下面的语句，说明execvp出错
9 |     处理报错信息
10 | }
```

6. exit

- 命令格式：

exit

- 实现原理：

直接调用exit(int)函数即可。

7. help

- 命令格式：

help

help 指令

- 实现原理：

提前编写好用户手册，放在MyShell同目录下。然后从文本中读取内容再显示到屏幕即可。

8. pwd

- 命令格式：

pwd

- 实现原理：

使用getcwd函数可以得到当前目录路径，然后直接输出即可。

9. set

- 命令格式：

set

set value₁ value₂ value₃ ... value_n

- 实现原理：

首先考虑无参执行set的情况，需要输出所有环境变量的值。在unix环境中，可以使用extern char ** environ声明一个外部变量，这个变量可以视为字符串数组，以NULL结尾，NULL之前每一个字符串分别对应一个环境变量（以及它的值）。关于environ变量的更多说明，参考《UNIX环境高级编程》第163页。

其次，有参执行set，表示把\$0，\$1，\$2等赋值成对应的值。而且这些参数也可以用'\$'引用已有变量。利用前文所述的GetValue函数就可以提取出变量的值，然后把它们一一赋值给argv[1], argv[2], ..., argv[n]，然后更新argc即可。

需要注意的是，这些赋值需要“同步”进行。也就是说先进行的赋值不能影响后进行的赋值。考虑如下命令：

```
1 | set $2 $1
```

其效果是把\$1和\$2的值交换。如果我们先进行\$1=\$2，再进行\$2=\$1，就会导致原来\$1的值丢失。所以，需要开辟一块临时空间，实现“同步赋值”的效果。

```
1 //set命令列出当前所有环境变量，或设置全局变量的值
2 void _set(string cmd[], int ParaNum){
3     extern char** environ;//环境变量表
4     if (ParaNum > 0){
5         //参数不为0，则改变全局变量
6         string TempArgv[ParaNum + 1];//参数的值
7         //根据string cmd[], 确定参数的值，存入TempArgv
8         for (int i = 1; i <= ParaNum; i++)
9             TempArgv[i] = GetValue(cmd[i - 1]);
10        //更新argc
11        argc = ParaNum + 1;
12        //更新argv
13        for (int i = 1; i < TempArgc; i++)
14            argv[i] = TempArgv[i];
15    } else {
16        //参数个数为0，则输出所有环境变量
17        for(int i = 0; environ[i] != NULL; i++)
18            输出environ[i]
19            换行
20    }
21 }
```

10. shift

- 命令格式：

shift

shift 非负整数

- 实验原理：

shift用于左移参数。若shift无参，则表示左移一次。否则根据参数确定左移位数。

用一重循环就能实现：

```
1 //shift命令，左移全局变量
2 void _shift(string cmd[], int ParaNum){
3     if (ParaNum >= 2){
4         参数过多，报错
5     }else{
6         int cnt;
```

```

7         if (ParaNum == 0) cnt = 1; //无参, 默认左移1位
8         else if (参数不是合法的非负整数){
9             处理报错信息
10            return;
11        } else cnt = 参数对应的非负整数;
12        //更新argc
13        if (argc <= cnt + 1)
14            //若变量个数 <= 左移次数, 则直接把argc置1即可
15            argc = 1;
16        else
17            //否则, argc = argc - 左移位数
18            argc -= cnt;
19        //用一重循环实现左移
20        for (int i = 1; i < argc; i++)
21            argv[i] = argv[i + cnt];
22    }
23 }

```

11. test

- 命令格式:

一元运算:

- 文件存在
test -e File
- 文件存在且可读
test -r File
- 文件存在且可写
test -w File
- 文件存在且可执行
test -x File
- 文件存在且至少有一个字符
test -s File
- 文件为目录文件
test -d File
- 文件为普通文件
test -f File
- 文件为字符型特殊文件
test -c File
- 文件为块特殊文件
test -b File
- 文件为符号链接
test -h File
test -L File
- 文件为命名管道
test -p File
- 文件为嵌套字

test -S File

- 文件被当前实际组拥有

test -G File

- 文件被当前实际用户拥有

test -O File

- 文件设置了setgid bit

test -g File

- 文件设置了setuid bit

test -u File

- 文件设置了sticky bit

test -k File

- 字符串长度非0

test -n String

- 字符串长度为0

test -z String

二元运算：

- 文件1和文件2的设备号和inode相同

test File1 -ef File2

- 文件1比文件2新

test File1 -nt File2

- 文件1比文件2旧

test File1 -ot File2

- 字符串相等

test String1 = String2

- 字符串不等

test String1 != String2

- 整数相等

test Integer1 -eq Integer2

- 整数大于等于

test Integer1 -ge Integer2

- 整数大于

test Integer1 -gt Integer2

- 整数小于等于

test Integer1 -le Integer2

- 整数小于

test Integer1 -lt Integer2

- 整数不相等

test Integer1 -ne Integer2

- 实现原理：

test指令支持的测试一共有3类，分别是文件测试、整数测试、字符串测试。整数测试和字符串测试很简单，这里不再赘述。

对于文件测试，sys/stat.h提供了四个用于读取文件信息的函数：stat，fstat，lstat，fstatat。stat函数返回指定文件的信息，lstat与stat相似，但当文件是一个符号链接时，lstat返回该符号链接的信息，而不是该符号链接所引用的文件的信息。具体参考《UNIX环境高级编程》第74页。

笔者在代码中使用lstat函数，函数原型为

```
1 int lstat(const char * restrict pathname,
2          struct stat * restrict buf);
```

给定pathname，表示文件的路径。若函数调用失败，则返回-1。若函数调用成功，则返回0，同时把文件的信息存入*buf。

struct stat结构的成员如下所示，MyShell代码中需要用到变量后面已经加了注释：

```
1 struct stat{
2     mode_t      st_mode;    //存储文件类型、文件权限等信息
3     ino_t       st_ino;     //i-node
4     dev_t       st_dev;
5     dev_t       st_rdev;
6     nlink_t     st_nlink;
7     uid_t       st_uid;     //文件所有者的uid
8     gid_t       st_gid;     //文件所有者的gid
9     off_t       st_size;    //普通文件的字节大小
10    struct timespec st_atime;
11    struct timespec st_mtime; //最后一次修改时间
12    struct timespec st_ctime;
13    blksize_t     st_blksize;
14    blkcnt_t      st_blocks;
15 };
```

欲判断文件类型，可以使用如下宏定义，返回布尔值：

```
1 S_ISREG(buf.st_mode)    //普通文件
2 S_ISDIR(buf.st_mode)    //目录文件
3 S_ISCHR(buf.st_mode)    //字符特殊文件
4 S_ISBLK(buf.st_mode)    //块特殊文件
5 S_ISFIFO(buf.st_mode)   //命名管道
6 S_ISLNK(buf.st_mode)    //符号链接
7 S_ISSOCK(buf.st_mode)   //套接字
```

欲判断文件权限，一种方法是用如下宏定义和st_mode进行按位与：

```
1 S_IRUSR    //用户读
2 S_IWUSR    //用户写
3 S_IXUSR    //用户执行
4 S_IRGRP    //组读
5 S_IWGRP    //组写
6 S_IXGRP    //组执行
7 S_IROTH    //其他读
8 S_IWOTH    //其他写
9 S_IXOTH    //其他执行
```

这种办法较为繁琐，因为还需要另外判断当前用户是否是文件的所有者、所有组。unistd.h提供了access函数，可以验证实际用户能否以指定的模式访问指定文件。函数返回0表示成功，返回-1表示失败。具体参考《UNIX环境高级编程》第81页：

```
1 | int access(const char * pathname, int mode);
```

mode有四种取值：

```
1 | F_OK      //文件存在
2 | R_OK      //文件可读
3 | W_OK      //文件可写
4 | X_OK      //文件可执行
```

欲判断文件的3个特殊权限位，可以用以下3个常数和st_mode按位与：

```
1 | S_ISUID //setuid bit
2 | S_ISGID //setgid bit
3 | S_ISVTX //sticky bit
```

有了以上介绍，test指令就很容易实现了。代码比较简单，不再赘述。

12. time

- 命令格式：

time

- 实现原理：

time指令用于显示当前的系统时间。具体可以参考《UNIX环境高级编程》第151页。

执行以下代码可以获取时间信息：

```
1 | time_t tt = time(NULL);
2 | struct tm * t = localtime(&tt);
```

struct tm类型的成员变量如下所示：

```
1 | struct tm{
2 |     int tm_sec;      //秒，取值0~60
3 |     int tm_min;      //分，取值0~59
4 |     int tm_hour;      //时，取值0~23
5 |     int tm_mday;      //月份中的第几天，取值1~31
6 |     int tm_mon;       //月份，取值0~11
7 |     int tm_year;      //从1900年开始的年份数
8 |     int tm_wday;      //从周日开始的天数，取值0~6
9 |     int tm_yday;      //从1月1日开始的天数，取值0~365
10 |    int tm_isdst;      //不懂
11 | };
```

将上述信息转换成合适的字符串，然后直接输出即可。

13. umask

- 命令格式：

umask

umask o
umask oo
umask ooo

- 实现原理:

umask无参时，表示输出当前的umask值。否则，umask接受至多3位8进制数，然后设置新的umask值。当数值不足3位时，默认向右对齐（高位补0）。

sys/stat.h提供了umask函数，可以直接读取或设置新的umask值：

```
1 mode_t umask(mode_t cmask);
```

umask函数将屏蔽字设置为cmask，并返回之前的屏蔽字值。具体参考《UNIX环境高级编程》第83页。

该命令的实现非常简单：

```
1 //输出umask或更改umask的值
2 void _umask(string cmd[], int ParaNum){
3     if (ParaNum >= 2){
4         参数过多，处理报错信息；
5     } else if (ParaNum == 1){
6         //只有一个参数，表示设置umask的值
7         if (参数不合法){
8             处理报错信息；
9             return;
10        }
11        mode_t cmask = 参数对应的3位八进制数；
12        umask(cmask);
13    }else{
14        //无参，表示显示umask的值
15        mode_t currentmode = umask(0);
16        umask(currentmode);
17        输出currentmode；
18    }
19 }
```

14. unset

- 命令格式:

unset 变量名

- 实现原理:

直接调用unsetenv函数即可。

15. 外部命令

- 命令格式:

ProgramName arg₁ arg₂ arg₃ ... arg_n

- 实现原理:

当输入了不属于前15种的指令时，MyShell自动将其识别为外部程序调用。在Bash中，外部程序调用分为两步：1. 调用vfork将进程拷贝一份，共享数据段，子进程优先执行，父进程等待子进程执行完毕；2. 子进程调用exec执行外部程序。

但在笔者的设计中，笔者使用fork而不是vfork，这样是为了防止vfork生成的子进程在退出时冲刷了输入输出流，从而影响父进程；同时也为了方便下文要介绍的作业控制。fork和vfork的区别在于fork生成的子进程和父进程不共享数据段，且父进程和子进程的执行顺序不确定。关于这两个函数，具体参考《UNIX环境高级编程》第182页和第187页。

为了让子进程仍然优先于父进程执行，我们可以让父进程调用waitpid函数，等待子进程结束。

```
1 SubPID = fork();
2 if (SubPID == 0){//子进程
3     //设置PARENT环境变量
4     setenv("PARENT", ShellPath.c_str(), 1);
5     //调用前文实现的_exec函数
6     _exec(cmd, ParaNum);
7     //如果能执行到这一步，说明exec出错
8     处理报错信息;
9     exit(1);
10 }
11 //父进程等待子进程完成
12 waitpid(SubPID, NULL, 0);
```

2. 重定向

通常情况下，一条指令从标准输入（stdin）读取数据，把结果输出到标准输出（stdout）。如果执行出错，则把错误信息输出到标准错误输出（stderr）。

MyShell允许对这三个标准输入输出进行重定向，格式为：

符号	解释
<	输入重定向
0<	输入重定向
>	输出重定向（覆盖）
1>	输出重定向（覆盖）
>>	输出重定向（追加）
1>>	输出重定向（追加）
2>	错误信息重定向（覆盖）
2>>	错误信息重定向（追加）

重定向有两种实现途径。第一种是利用输入输出流。利用freopen函数重定向，然后利用fscanf和fprintf实现输入和输出。此外fopen和fclose函数可以打开或关闭文件。这种办法基于FILE类型的变量，具体参考《UNIX环境高级编程》第115页：

```

1 FILE * fopen(const char * restrict pathname,
2             const char * restrict type);
3 FILE * freopen(const char * restrict pathname,
4               const char * restrict type,
5               FILE * restrict fp);
6 FILE * fdopen(int fd, const char * type);
7 int fclose(FILE * fp);
8 int fscanf(FILE * restrict fp, const char * restrict format, ...);
9 int fprintf(FILE * restrict fp, const char * restrict format, ...);

```

第二种方法是利用文件描述符。利用dup2函数重定向，然后利用read和write实现输入和输出。此外open和close函数可以打开或关闭文件。函数的定义在fcntl.h中。具体参考《UNIX环境高级编程》第49页。

```

1 int open(const char * path, int oflag, ...);
2 int close(int fd);
3 ssize_t read(int fd, void * buf, size_t nbytes);
4 ssize_t write(int fd, const void * buf, size_t nbytes);
5 int dup(int fd);
6 int dup2(int fd, int fd2);

```

第二种方法的好处在于1. open函数可以指定以何种方式打开外部文件。此外，dup函数可以用来复制（备份）一个现有的文件描述符。对于一条包含重定向的指令，MyShell在执行完它后，需要把三个标准输入输出恢复到原来的状态，这个时候就需要用到dup函数。2. 涉及到管道符“|”时，必须要调用pipe函数创建管道（见下文），而管道是以文件描述符的形式存在的，在这个方法中就不需要考虑文件描述符（int类型）和FILE变量的相互转换。

第二种方法的坏处在于1. 无法进行格式化输入输出。但这一点可以用sscanf和sprintf函数弥补。2. 因为没有考虑缓冲区，输入输出效率较低。

笔者选择第二种方法。重定向的伪代码如下：

```

1 //备份原来的标准输入输出
2 int InputFD = dup(STDIN_FILENO);
3 int OutputFD = dup(STDOUT_FILENO);
4 int ErrorFD = dup(STDERR_FILENO);
5 //搜索重定向符号
6 for(扫描指令字符串){
7     if (发现"<"和"0<"){//输入重定向
8         int NewFD = open(文件名, O_RDONLY);
9         if (NewFD < 0){
10             处理报错信息;
11             return;
12         }
13         dup2(NewFD, STDIN_FILENO);
14         close(NewFD);
15     }else
16     if (发现">"和"1>"){//输出重定向, 覆盖
17         int NewFD = open(文件名, O_WRONLY | O_CREAT | O_TRUNC, 0666);
18         if (NewFD < 0){
19             处理报错信息;
20             return;
21         }
22         dup2(NewFD, STDOUT_FILENO);
23         close(NewFD);
24     }else

```

```

25     if (发现">>"和"1>>"){//输出重定向, 追加
26         int NewFD = open(文件名, O_WRONLY | O_CREAT | O_APPEND, 0666);
27         if (NewFD < 0){
28             处理报错信息;
29             return;
30         }
31         dup2(NewFD, STDOUT_FILENO);
32         close(NewFD);
33     }else
34     if (发现"2>"){//错误信息重定向, 覆盖
35         int NewFD = open(文件名, O_WRONLY | O_CREAT | O_TRUNC, 0666);
36         if (NewFD < 0){
37             处理报错信息;
38             return;
39         }
40         dup2(NewFD, STDERR_FILENO);
41         close(NewFD);
42     }else
43     if (发现"2>>"){//错误信息重定向, 追加
44         int NewFD = open(文件名, O_WRONLY | O_CREAT | O_APPEND, 0666);
45         if (NewFD < 0){
46             处理报错信息;
47             return;
48         }
49         dup2(NewFD, STDERR_FILENO);
50         close(NewFD);
51     }
52 }
53 //执行指令
54 .....;
55 //恢复标准输入输出
56 dup2(InputFD, STDIN_FILENO);    close(InputFD);
57 dup2(OutputFD, STDOUT_FILENO);  close(OutputFD);
58 dup2(ErrorFD, STDERR_FILENO);   close(ErrorFD);

```

3. 管道符

MyShell支持管道符, 前一个指令的输出是下一个指令的输入; 第一条指令的输入是标准输入; 最后一条指令的输出是标准输出。

unistd.h提供了pipe函数, 可以生成管道, 经参数int fd[2]返回两个文件描述符, 其中fd[0]用于读, fd[1]用于写。具体参考《UNIX环境高级编程》第430页:

```

1 | int pipe(int fd[2]);

```

例如, 假设有两个用管道符连接的命令 `cmd1 | cmd2`, 则cmd1应该从stdin中读, 向fd[1]中写; cmd2应该从fd[0]中读, 向stdout里写。

遇到含有管道符的指令时, MyShell首先调用fork生成自己的一个拷贝。父进程A什么都不干; 子进程B调用pipe生成一系列管道, 并调用fork进一步生成一系列子进程C₁, C₂, ..., C_n。这些生成的子进程首先对自己的标准输入输出重定向, 然后执行单条指令, 最后退出。进程B等待进程C₁, C₂, ..., C_n全部执行完毕后再退出。

之所以不直接让进程A生成子进程C₁, C₂, ..., C_n, 是为了下文要实现的作业控制。当MyShell要挂起一个作业时, 是把整个含管道符的指令挂起, 而不是把这条指令中的n个单个指令分别挂起。

```

1  int SubPID = fork();//生成子进程
2  if (SubPID){
3      //父进程A
4      waitpid(SubPID, NULL, 0);
5  }else{
6      //子进程B
7      int FD1[2], FD2[2]; //前一个管道和后一个管道的文件描述符
8      int PID[N];          //N个指令子进程的id
9      int cnt = 0;          //子进程的个数
10     在指令串的末尾加一个管道符'|';
11     for(遍历每一个管道符'|'){
12         //生成管道
13         if (是第一个遇到的管道符){
14             FD1[0] = STDIN_FILENO; FD1[1] = -1;
15             pipe(FD2);
16         }else if (是最后一个管道符){
17             if (FD1 != STDIN_FILENO) close(FD1[0]);
18             FD1[0] = FD2[0]; FD1[1] = FD2[1]; close(FD1[1]);
19             FD2[0] = -1; FD2[1] = STDOUT_FILENO;
20         }else{
21             if (FD1 != STDIN_FILENO) close(FD1[0]);
22             FD1[0] = FD2[0]; FD1[1] = FD2[1]; close(FD1[1]);
23             pipe(FD2);
24         }
25         //调用fork生成子进程Ci
26         PID[cnt] = fork();
27         if (PID[cnt++] == 0){//子进程
28             //输入输出重定向
29             dup2(FD1[0], STDIN_FILENO);
30             dup2(FD2[1], STDOUT_FILENO);
31             close(FD1[1]);close(FD2[0]);
32             执行单条指令;
33             exit(0);
34         }
35     }
36     //进程B等待进程C1~Cn全部完成
37     for (int i = 0; i < cnt; i++)
38         waitpid(PID[i], NULL, 0);
39 }

```

4. 作业控制

和bash一样，MyShell应该支持作业控制。具体体现为：

1. 在指令后面加上&符号，指令就在后台运行。MyShell无需等待这条指令执行完成，可以立刻输出终端提示符并接收用户的下一条指令。
2. 若前台有作业正在执行，按Ctrl+C可以终止当前作业。
3. 若前台有作业正在执行，按Ctrl+Z可以挂起当前作业。
4. fg指令可以将被挂起或后台运行的作业转移至前台。
5. bg指令可以将被挂起的作业转移至后台运行。
6. jobs指令可以查看当前所有被挂起或后台运行的作业。

为了实现作业控制功能，MyShell需要自己维护一个作业表，把被挂起、后台运行的作业信息存储起来。若某个作业被终止或移到前台，则从表中删除其信息。若某个指令最后带有“&”字符，则将其加入到作业表。

“1”的实现很简单，只要把实际工作移交给子进程完成即可。父进程每次执行新指令之前，先检查子进程是否完成。若完成则告知用户，若未完成，父进程并不等待子进程，而是继续执行自己的任务。**此外，还要把子进程单独设置到一个进程组里（利用setpgid函数），使子进程加入到后台进程组，这样，Ctrl+Z、Ctrl+Z产生的信号就不会发送给该进程。**

“2”和“3”的实现需要用到“信号”。具体参考《UNIX环境高级编程》第249页。按Ctrl+Z时，终端会向前台进程组发送SIGINT信号，在默认情况下，进程接收到SIGINT信号后就会被挂起；按Ctrl+C时，终端会向前台进程组发送SIGTSTP信号，在默认情况下，进程接收到SIGTSTP信号后就会被终止；如果被挂起的进程接收到SIGCONT信号，在默认情况下，进程就可以恢复运行。

我们期望MyShell接收到这些信号时并不是被挂起、被终止，而是让MyShell当前正在执行的任务被挂起、被终止，同时MyShell记录下这个被挂起或被终止的作业信息，同时MyShell把作业进程设置为后台进程。因此，需要改变MyShell接收信号后的行为。可以使用signal.h提供的signal函数：

```
1 void (*signal(int signo, void (*func)(int)))(int)
```

另一种更易理解的定义为：

```
1 typedef void Sigfunc(int);  
2 Sigfunc * signal(int, Sigfunc);
```

第一个形参表示我们要改变行为的信号值（在本文中，可以是SIGINT、SIGTSTP），第二个形参表示接收到该信号后要执行的函数（也叫信号捕捉函数）。若signal执行成功，则返回之前的信号捕捉函数，若执行失败则返回SIG_ERR。

如果忽略某个信号或把信号捕捉函数恢复到默认，可以把第二个形参设为SIG_IGN或SIG_DFL。

“4”的实现只要让MyShell向指定的进程发送SIGCONT信号即可，然后MyShell一直等待直至子进程结束。此外MyShell还要调用setpgid函数，把子进程移到前台进程组，使其接受Ctrl+Z、Ctrl+Z产生的信号。

可以用kill函数发送信号：

```
1 int kill(pid_t pid, int signo);
```

“5”和“4”的区别在于，在“5”的情况下MyShell不需要等待子进程，也不需要调用setpgid函数把子进程移到前台进程组。

“6”的实现就是让MyShell打印作业表。

1. bg

o 命令格式：

```
bg  
bg WorkID
```

o 实现原理：

无参调用bg时输出所有正在后台运行的作业信息。只要扫描整个作业表并输出信息即可。
有参数时，表示把指定ID的作业（被挂起）转到后台运行。向其发送SIGCONT信号即可。

2. fg

o 命令格式：

```
fg WorkID
```

o 实现原理：

fg命令把指定ID的作业（被挂起或正在后台运行）转移到前台运行。若作业被挂起，向其发送SIGCONT信号并调用waitpid等待其结束；若作业在后台运行，直接调用waitpid等待其结束。

3. jobs

- 命令格式：

jobs

- 实现原理：

扫描整个作业表并输出作业信息。

四、运行截图

笔者的Ubuntu系统安装了anaconda3，所以在bash的提示字符前面可以看到“(base)”。这一点可以用来区分bash和MyShell。

1. 启动MyShell

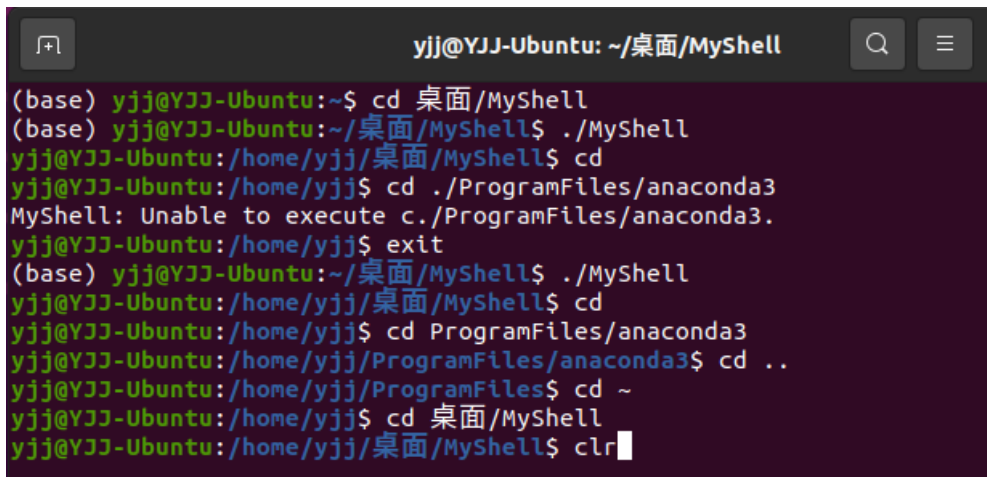
```
(base) yjj@YJJ-Ubuntu:~$ cd 桌面/MyShell
(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$ ./MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

2. cd

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ cd
yjj@YJJ-Ubuntu:/home/yjj$ cd ProgramFiles/anaconda3
yjj@YJJ-Ubuntu:/home/yjj/ProgramFiles/anaconda3$ cd ..
yjj@YJJ-Ubuntu:/home/yjj/ProgramFiles$ cd ~
yjj@YJJ-Ubuntu:/home/yjj$ cd 桌面/MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

3. clr

清屏前：



```
yjj@YJJ-Ubuntu: ~/桌面/MyShell
(base) yjj@YJJ-Ubuntu:~$ cd 桌面/MyShell
(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$ ./MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ cd
yjj@YJJ-Ubuntu:/home/yjj$ cd ./ProgramFiles/anaconda3
MyShell: Unable to execute c./ProgramFiles/anaconda3.
yjj@YJJ-Ubuntu:/home/yjj$ exit
(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$ ./MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ cd
yjj@YJJ-Ubuntu:/home/yjj$ cd ProgramFiles/anaconda3
yjj@YJJ-Ubuntu:/home/yjj/ProgramFiles/anaconda3$ cd ..
yjj@YJJ-Ubuntu:/home/yjj/ProgramFiles$ cd ~
yjj@YJJ-Ubuntu:/home/yjj$ cd 桌面/MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ clr
```

清屏后：

```
yjj@YJJ-Ubuntu: ~/桌面/MyShell
yjj@YJJ-Ubuntu: /home/yjj/桌面/MyShell$
```

4. dir

```
yjj@YJJ-Ubuntu: ~/桌面/MyShell
yjj@YJJ-Ubuntu: /home/yjj/桌面/MyShell$ dir
.vscode
EndlessLoop
EndlessLoop.cpp
MyShell
MyShell.cpp
TestProgram
TestProgram.cpp
help
test
testbash
yjj@YJJ-Ubuntu: /home/yjj/桌面/MyShell$ dir /home/yjj
.anaconda
.bash_history
.bash_logout
.bashrc
.cache
.conda
.condarc
.config
.continuum
.gnupg
.ipython
.local
.mozilla
.nv
.pki
.profile
.ssh
.sudo_as_admin_successful
.vscode
.wget-hsts
ProgramFiles
Workspace
config.yaml
python[version=>=2.7,<2.8.0a0|>=3.8,<3.9.0a0|>=3.6,<3.7.0a0|>=3.7,<3.8.0a0|>=3.5,<3.6.0a0]
snap
下载
公共的
图片
文档
桌面
模板
视频
音乐
yjj@YJJ-Ubuntu: /home/yjj/桌面/MyShell$
```

5. echo

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo string
string
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo $0 $PWD $SHELL
./MyShell /home/yjj/桌面/MyShell /home/yjj/桌面/MyShell/MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo $HOME comment $1 $2 $0
/home/yjj comment ./MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

6. exec

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ exec bash
(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$ ps
  PID TTY          TIME CMD
 12197 pts/0        00:00:00 bash
 12812 pts/0        00:00:00 bash
 12825 pts/0        00:00:00 ps
(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$
```

通过ps指令可以看到原来的MyShell进程被替换成了bash。

7. help

```
(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$ ./MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ help
MyShell用户手册:
支持的指令: bg, cd, clr, dir, echo, exec, exit, fg, help, jobs, pwd, set, shift, test, time, unask, unset, more. 其余指令被解释为外部程序调用.
支持重定向: "<", "o<"表示输入重定向; ">", "1>"表示输出重定向(覆盖); ">>", "1>>"表示输出重定向(追加); "2>"表示错误重定向(覆盖); "2>>"表示错误重定向(追加).
支持管道符: 使用"|"分隔多个指令, 前一个指令的输出是下一个指令的输入. 第一个指令的输入是标准输入, 最后一个指令的输出是标准输出.
支持作业控制: 快捷键Ctrl+z能挂起前台作业; 快捷键Ctrl+c能终止前台作业. bg、fg、jobs指令的用法请查询对应的帮助手册.
支持执行外部脚本: 命令MyShell BatchFile表示从BatchFile中获取命令并执行.
更多指令帮助, 请使用"help 指令名"获取对应帮助.

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ help fg
命令格式:
fg WorkID
指令解释:
fg命令把指定Io的作业(被挂起或正在后台运行)转移到前台运行.

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ help cd
命令格式:
cd
cd pathname
指令解释:
无参数执行cd时表示进入用户的主目录.
有参数时, 表示进入参数指定的目录.
每次执行成功会更新PWD环境变量. 可以用echo $PWD查看.

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ help help
命令格式:
help
help cmd
指令解释:
不加参数时输出MyShell用户帮助手册, 加参数时输出对应的指令帮助手册.

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ help man
help: There's no help manual for man.
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

使用管道符连接more命令的效果:

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ help test | more
```

```
命令格式:
文件存在
test -e File

文件存在且可读
test -r File

文件存在且可写
test -w File

文件存在且可执行
test -x File

文件存在且至少有一个字符
test -s File

文件为目录文件
test -d File

文件为普通文件
test -f File

文件为字符型特殊文件
test -c File

文件为块特殊文件
test -b File

文件为符号链接
test -h File
test -L File

文件为命名管道
test -p File

文件为嵌套字
test -S File

文件被当前实际组拥有
test -G File

文件被当前实际用户拥有
test -O File

文件设置了setgid bit
test -g File

文件设置了setuid bit
test -u File

文件设置了sticky bit
test -k File

字符串长度非0
--更多--
```

8. pwd

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ pwd
/home/yjj/桌面/MyShell
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ cd
yjj@YJJ-Ubuntu:/home/yjj$ pwd
/home/yjj
```

9. set

不含参set输出环境变量:

```
LS_COLORSrs=0;dl=01;34;lnl=01;36;mh=00;ptl=40;33;sos=01;35;d=01;35;bd=40;33;01;cdd=40;33;01;or=40;31;01;nl=00;su=37;41;sg=30;43;ca=30;41;t=30;42;ow=34;  
;31*;arj=01;31*;taz=01;31*;lha=01;31*;lz4=01;31*;lzh=01;31*;lzn=01;31*;tlz=01;31*;txz=01;31*;tzo=01;31*;t7z=01;31*;zip=01;31*;z=01;31;*;  
vz=01;31*;xz=01;31*;zt=01;31*;ztgz=01;31*;bz2=01;31*;bz=01;31*;bzr=01;31*;ht2=01;31*;ht=01;31*;dhb=01;31*;rpm=01;31*;jar=01;31*;war=01;31*;ear=
```

含参set改变全局参数:

```
yjj@YJJ-Ubuntu: /home/yjj$
```

10. shift

```
yjj@YJJ-Ubuntu: /home/yjj$
```

11. test

```
yjj@YJJ-Ubuntu: /home/yjj$
```

12. time

```
yjj@YJJ-Ubuntu:/home/yjj$ time
2021.8.1. Sunday 10:1:21
yjj@YJJ-Ubuntu:/home/yjj$ time aaa
time: Too many parameters.
yjj@YJJ-Ubuntu:/home/yjj$
```

13. umask

```
yjj@YJJ-Ubuntu:/home/yjj$ umask
0002
yjj@YJJ-Ubuntu:/home/yjj$ umask 3
yjj@YJJ-Ubuntu:/home/yjj$ umask
0003
yjj@YJJ-Ubuntu:/home/yjj$ umask 66
yjj@YJJ-Ubuntu:/home/yjj$ umask
0066
yjj@YJJ-Ubuntu:/home/yjj$ umask 777
yjj@YJJ-Ubuntu:/home/yjj$ umask
0777
yjj@YJJ-Ubuntu:/home/yjj$ umask 7777
yjj@YJJ-Ubuntu:/home/yjj$ umask
0777
yjj@YJJ-Ubuntu:/home/yjj$ umask 77777
umask: Expected at most 4 octonary digits: 77777
yjj@YJJ-Ubuntu:/home/yjj$ umask 8
8 is not an octonary digit.
yjj@YJJ-Ubuntu:/home/yjj$
```

14. unset

```
yjj@YJJ-Ubuntu:/home/yjj$ echo $HOME
/home/yjj
yjj@YJJ-Ubuntu:/home/yjj$ unset HOME
yjj@YJJ-Ubuntu:/home/yjj$ echo $HOME
yjj@YJJ-Ubuntu:/home/yjj$
```

15. 调用外部程序

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ ./helloworld
Hello world!
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

16. 后台运行sleep，用ps指令查看到sleep进程确实存在：

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ sleep 10 &
[1] 15560 Running sleep 10
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ ps
  PID TTY          TIME CMD
 12976 pts/0    00:00:00 bash
  15515 pts/0    00:00:00 MyShell
  15560 pts/0    00:00:00 sleep
  15571 pts/0    00:00:00 ps
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
[1] 15560 Finish sleep 10
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ ps
  PID TTY          TIME CMD
 12976 pts/0    00:00:00 bash
  15515 pts/0    00:00:00 MyShell
  15573 pts/0    00:00:00 ps
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

前台和后台运行echo指令的对比：

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo $HOME
/home/yjj
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo $HOME &
[1] 15645 Running echo $HOME
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ /home/yjj
```

17. jobs

```

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ sleep 10
^Z
[1]      15698      Hanging      sleep 10
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ sleep 20
^Z
[2]      15699      Hanging      sleep 20
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ jobs
[1]      15698      Hanging      sleep 10
[2]      15699      Hanging      sleep 20
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$

```

18. fg

```

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ jobs
[1]      15698      Hanging      sleep 10
[2]      15699      Hanging      sleep 20
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ fg 2
sleep 20
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ jobs
[1]      15698      Hanging      sleep 10
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$

```

19. bg

```

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ jobs
[1]      15698      Hanging      sleep 10
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ bg 1
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
[1]      15698      Finish       sleep 10
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ jobs
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$

```

20. 管道符。RepeatCharacter程序的作用是把接收到的每个字符重复输出两遍：

```

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo a | ./RepeatCharacter
aa
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo abc | ./RepeatCharacter | ./RepeatCharacter
aaaabbbbcccc
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$

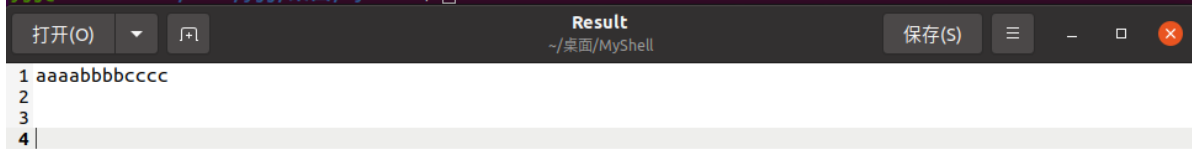
```

21. 重定向。在含管道符的指令中也可以使用重定向符号：

```

yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo abc | ./RepeatCharacter | ./RepeatCharacter 1> Result
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$

```



单个dir指令的重定向：

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ echo abc | ./RepeatCharacter | ./RepeatCharacter 1> Result
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ dir 1>> Result
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

打开(O)

Result

保存(S)

~/桌面/MyShell

```
1 aaaabbbbcccc
2
3
4
5 .vscode
6 EndlessLoop
7 EndlessLoop.cpp
8 MyShell
9 MyShell.cpp
10 RepeatCharacter
11 RepeatCharacter.cpp
12 Result
13 helloworld
14 helloworld.cpp
15 help
16 test
17 testbash
```

错误信息重定向:

```
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$ dir ddd 2> Result
yjj@YJJ-Ubuntu:/home/yjj/桌面/MyShell$
```

打开(O)

Result

~/桌面/MyShell

```
1 dir: Folder ddd doesn't Exist.
```

22. 含参调用MyShell，表示从文件中读取并执行指令。此时MyShell不会向终端输出提示字符:

```
(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$ ./MyShell testbash
a
./MyShell
ttrruuee

(base) yjj@YJJ-Ubuntu:~/桌面/MyShell$
```

打开(O)

testbash

保存(S)

~/桌面/MyShell

```
1 echo a
2 echo $0
3 test -e help | ./RepeatCharacter
4
```

附录：代码

```
1 //程序名: MyShell
2 //作者学号: 尤锦江 3190102352
3
4
5 #include <iostream>
6 #include <string>
7 #include <sstream>
8 #include <ctime>
9 #include <unistd.h>
10 #include <dirent.h>
11 #include <pwd.h>
12 #include <sys/wait.h>
13 #include <sys/stat.h>
14 #include <sys/types.h>
15 #include <fcntl.h>
16 using namespace std;
17
18
19 //-----全局变量-----
-----
```



```

20
21 //存储当前指令
22 char buffer[1024] = {0};
23 string CurrentCMD;
24 //输入输出控制相关
25 bool InIsTerminal; //输入来自终端. 只有为真时, 才会输出子进程表和提示字符串
26 bool OutIsTerminal; //输出送至终端.
27 int TerminalIn; //终端输入的文件描述符, 未必等于STDIN_FILENO, 因为标准输入有可能被
    重定向
28 int TerminalOut; //终端输出的文件描述符, 未必等于STDOUT_FILENO, 因为标准输出有可能
    被重定向
29 //返回信息变量
30 int State; //0: 执行成功; 1: 执行错误
31 string Message1; //输出内容
32 string Message2; //错误信息
33 //部分环境变量
34 string HostName; //主机名
35 string UserName; //用户名
36 string HomeDir; //用户主目录
37 string ShellPath; //MyShell的路径
38 string HelpPath; //帮助手册的路径
39 string PWD; //当前路径
40 int PID; //进程号
41 int SubPID; //子进程进程号, 若当前无子进程则设为-1
42 int argc; string argv[1024]; //参数个数和参数值, 分别对应$, $0, $1, $2, ...
43 //子进程表
44 const int MaxWork = 1024;
45 int Jobs[MaxWork]; //子进程的pid
46 int States[MaxWork]; //子进程的状态. 0: 空. 1: 后台运行. 2: 被挂起
47 string CMDInfo[MaxWork]; //子进程执行的命令
48 int Front, Rear; //表头指针和表尾指针
49
50
51 //-----辅助函数-----
    -----
52
53 //快排模板
54 template <class T>
55 static void QuickSort(T Arr[], int l, int r){
56     if (l >= r) return;
57     T mid = Arr[(l+r)/2];
58     int i = l, j = r;
59     while (i <= j){
60         while (Arr[i] < mid) i++;
61         while (Arr[j] > mid) j--;
62         if (i <= j){
63             T temp = Arr[i];
64             Arr[i] = Arr[j];
65             Arr[j] = temp;
66             i++; j--;
67         }
68     }
69     QuickSort(Arr, l, j);
70     QuickSort(Arr, i, r);
71 }
72
73 //将字符串转换为整数. 若无法转换则抛出异常
74 int StringToInt(const string& n);

```

```

75
76 //解析"$0", "$1", "${9}", "${31}"这种字符串, 返回其所代表的数值。例如输入"$9"则返回9。 若不合法则返回-1
77 int GetParamIndex(const string& cmd);
78
79 //根据字符串var, 取得对应变量的值。例如var="$HOME"时, 返回变量HOME对应的值
80 string GetValue(const string &var);
81
82 //比较两个timespec类型的变量。-1: 小于; 0: 等于; 1: 大于
83 int TimeCMP(const timespec& t1, const timespec& t2);
84
85 //返回子进程信息连接而成的字符串
86 string JobString(int JobIndex, int Finished = 0);
87
88
89 //-----单个指令实现-----
90
91 //bg命令, 将被挂起的工作转至后台运行
92 void _bg(string cmd[], int ParaNum);
93
94 //cd命令。若无参数则进入主目录, 否则进入参数对应的目录
95 void _cd(string cmd[], int ParaNum);
96
97 //clr清屏
98 void _clr(string cmd[], int ParaNum);
99
100 //dir列出指定文件夹内的文件
101 void _dir(string cmd[], int ParaNum);
102
103 //echo打印内容到屏幕
104 void _echo(string cmd[], int ParaNum);
105
106 //执行exec指令
107 void _exec(string cmd[], int ParaNum);
108
109 //exit退出程序
110 void _exit(string cmd[], int ParaNum);
111
112 //fg将后台作业移至前台执行
113 void _fg(string cmd[], int ParaNum);
114
115 //help输出帮助信息
116 void _help(string cmd[], int ParaNum);
117
118 //jobs输出当前正在执行或被挂起的任务
119 void _jobs(string cmd[], int ParaNum);
120
121 //pwd列出当前目录
122 void _pwd(string cmd[], int ParaNum);
123
124 //set命令列出当前所有环境变量, 或设置全局变量的值
125 void _set(string cmd[], int ParaNum);
126
127 //shift命令, 左移全局变量
128 void _shift(string cmd[], int ParaNum);
129
130 //test命令

```

```

131 void _test(string cmd[], int ParaNum);
132
133 //time列出当前时间
134 void _time(string cmd[], int ParaNum);
135
136 //umask输出屏蔽字或更改屏蔽字的值
137 void _umask(string cmd[], int ParaNum);
138
139 //unset命令删除环境变量
140 void _unset(string cmd[], int ParaNum);
141
142
143 //-----重要函数-----
144 -----
145
146 //初始化
147 void Init(int Argc, char * Argv[]);
148
149 //信号处理函数，用于处理Ctrl+C、Ctrl+Z等组合键
150 void SignalProcess(int Signal);
151
152 //第一层解析指令：处理"&"后台运行符号，其余交给ExecMultipCMD函数
153 void Exec(string CMD);
154
155 //第二层解析指令：执行多条用管道符分隔的指令。单条指令交给ExecSingleCMD函数
156 void ExecMultipCMD(string cmd[], int ParaNum, bool WithFork = true);
157
158 //第三层解析指令：执行单条指令，可能包含重定向
159 void ExecSingleCMD(string cmd[], int ParaNum, bool WithFork = true);
160
161 //-----main函数-----
162 -----
163
164 int main(int Argc, char * Argv[]){
165     //初始化
166     Init(Argc, Argv);
167     //循环处理指令系统默认
168     int Flag = 1;
169     while(Flag){
170         //如果是在控制台界面下，则需要输出提示字符
171         if (InIsTerminal){
172             sprintf(buffer, "\e[1;32m%s@%s\e[0m:\e[1;34m%s\e[0m$ ",
173                     UserName.c_str(),
174                     HostName.c_str(),
175                     PWD.c_str());
176             string TempStr = buffer;
177             write(TerminalOut, TempStr.c_str(), TempStr.length());//注意：
提示字符串只输出到终端，不能输出到STDOUT!!!!
178         }
179         //读取一行，遇到换行符或EOF则结束
180         int i;
181         for (i = 0; ; i++){
182             if (read(STDIN_FILENO, buffer + i, 1) <= 0){//读到EOF
183                 Flag = 0;
184                 break;
185             }
186             if (buffer[i] == '\n')

```

```

186         break;
187     }
188     buffer[i] = '\0';
189     //执行指令
190     Exec(buffer);
191 }
192 }
193
194
195 //-----函数实现-----
196
197 //将字符串转换为整数。若无法转换则抛出异常
198 int StringToInt(const string& n){
199     int Sign = 1, Magn = 0, i = 0;
200     while (n[i] == '+' || n[i] == '-'){
201         if (n[i] == '-') Sign = -Sign;
202         i++;
203     }
204     if (n[i] == '\0') throw invalid_argument("Not a integer.");
205     for (; n[i]; i++){
206         if (n[i] <= '9' && n[i] >= '0')
207             Magn = Magn * 10 + n[i] - '0';
208         else throw invalid_argument("Not a integer.");
209     }
210     return Sign * Magn;
211 }
212
213 //解析"$0", "$1", "${9}", "${31}"这种字符串，返回其所代表的数值。例如输入"$9"则返回9。若不合格则返回-1
214 int GetParamIndex(const string& cmd){
215     int ans = 0;
216     if (cmd.length() <= 1 || cmd[0] != '$') return -1;
217     if (cmd[1] == '{'){
218         if (cmd.length() <= 3 || cmd[cmd.length() - 1] != '}') return -1;
219         for (int i = 2; i < cmd.length() - 1; i++){
220             if (cmd[i] >= '0' && cmd[i] <= '9')
221                 ans = ans * 10 + cmd[i] - '0';
222             else return -1;
223         }
224     }else{
225         for (int i = 1; i < cmd.length(); i++){
226             if (cmd[i] >= '0' && cmd[i] <= '9')
227                 ans = ans * 10 + cmd[i] - '0';
228             else return -1;
229         }
230     }
231     return ans;
232 }
233
234 //根据字符串var，取得对应变量的值。例如var="$HOME"时，返回变量HOME对应的值
235 string GetValue(const string &var){
236     if (var[0] == '$'){//以$开头，替换成变量的值
237         int ParaIndex;
238         if ((ParaIndex = GetParamIndex(var)) >= 0){//进一步判断变量名是否是
239             $0, $1这种名字
240             if (ParaIndex < argc)
241                 return argv[ParaIndex];
242             else return "";
243         }else if (var == "$#"){//$#表示参数个数
244             return to_string(argc - 1);

```

```

241         else{//否则, 利用getenv获得变量的值
242             char * str = getenv(var.substr(1).c_str());
243             if (str != NULL)//环境变量存在
244                 return str;
245             else return "";
246         }
247     } else //否则, 不以$开头, 则直接返回原字符串
248         return var;
249 }
250
251 //比较两个timespec类型的变量。-1: 小于; 0: 等于; 1: 大于
252 int TimeCMP(const timespec& t1, const timespec& t2){
253     if (t1.tv_sec < t2.tv_sec)
254         return -1;
255     if (t1.tv_sec > t2.tv_sec)
256         return 1;
257     if (t1.tv_nsec < t2.tv_nsec)
258         return -1;
259     if (t1.tv_nsec > t2.tv_nsec)
260         return 1;
261     return 0;
262 }
263
264 //输出子进程信息
265 string JobString(int JobIndex, int Finished){
266     if (JobIndex < Front || JobIndex >= Rear || States[JobIndex] != 1 &&
States[JobIndex] != 2) return "";
267     return "[" + to_string(JobIndex + 1) + "]\t" +
to_string(Jobs[JobIndex]) + "\t\t" + ((Finished) ? "Finish" :
(States[JobIndex] == 1) ? "Running" : "Hanging") + "\t\t" +
CMDInfo[JobIndex] + "\n";
268 }
269
270 //bg命令, 将被挂起的工作转至后台运行
271 void _bg(string cmd[], int ParaNum){
272     //无参, 列出所有正在后台运行的进程
273     if (ParaNum == 0){
274         Message1 = "";
275         Message2 = "";
276         for (int i = Front; i < Rear; i++){
277             if (States[i] == 1)
278                 Message1 += JobString(i);
279             if (Message1 == "") Message1 = "bg: No mission is running at the
background.\n";
280             State = 0;
281             return;
282         }
283         //根据参数列表, 把对应的任务移至后台运行
284         Message1 = "";
285         Message2 = "";
286         State = 0;
287         for (int i = 0; i < ParaNum; i++){
288             int WorkID;
289             //将字符串转换为整数
290             try{
291                 WorkID = StringToInt(cmd[i]);
292             }catch(...){

```

```

293     Message1 += "bg: " + cmd[i] + ": Not a valid positive
integer.\n";
294     continue;
295 }
296 //若不存在指定作业号的任务
297 if (WorkID > Rear || WorkID <= Front || States[WorkID - 1] == 0){
298     Message1 += "bg: " + cmd[i] + ": No mission.\n";
299     continue;
300 }
301 //若已经在后台运行
302 if (States[WorkID - 1] == 1){
303     Message1 += "bg: " + cmd[i] + ": Already running at the
background.\n";
304     continue;
305 }
306 //更新Jobs表并发送信号
307 States[WorkID - 1] = 1;
308 kill(Jobs[WorkID - 1], SIGCONT);
309 }
310 }
311
312 //cd命令. 若无参数则进入主目录, 否则进入参数对应的目录
313 void _cd(string cmd[], int ParaNum){
314     if (ParaNum == 0 || ParaNum == 1 && cmd[0] == "~"){//chdir不支持 "~"作为
路径, 所以需要单独判断
315         //进入用户主目录
316         chdir(HomeDir.c_str());
317         PWD = HomeDir;
318         //更新PWD环境变量
319         setenv("PWD", PWD.c_str(), 1);
320         Message1 = "";
321         Message2 = "";
322         State = 0;
323     }else if (ParaNum == 1){//参数为1, 直接调用chdir改变路径
324         if(chdir(cmd[0].c_str())){//chdir调用失败
325             Message1 = "";
326             Message2 = "cd: Unable to change directory to " + cmd[0] +
"\n";
327             State = 1;
328         }else{//chdir调用成功
329             char buffer[1024];
330             getcwd(buffer, 1024);
331             PWD = buffer;
332             //更新PWD环境变量
333             setenv("PWD", buffer, 1);
334             Message1 = "";
335             Message2 = "";
336             State = 0;
337         }
338     }else{//参数过多, 报错
339         Message1 = "";
340         Message2 = "cd: Too many parameters.\n";
341         State = 1;
342     }
343 }
344
345 //clr清屏
346 void _clr(string cmd[], int ParaNum){

```

```

347     if (ParaNum > 0){//参数过多, 报错
348         Message1 = "";
349         Message2 = "clr: Too many parameters.\n";
350         State = 1;
351         return;
352     }
353     system("clear");
354     Message1 = "";
355     Message2 = "";
356     State = 0;
357 }
358
359 //dir列出指定文件夹内的文件
360 void _dir(string cmd[], int ParaNum){
361     if (ParaNum >= 2){//参数过多, 报错
362         Message1 = "";
363         Message2 = "dir: Too many parameters.\n";
364         State = 1;
365     }else{
366         if (ParaNum == 0) cmd[0] = PWD;//若无参, 则列出当前目录下的文件。否则,
列出参数指定的目录下的文件
367         DIR *pDir;
368         struct dirent* ptr;
369         //打开目录
370         if(!(pDir = opendir(cmd[0].c_str()))){
371             Message1 = "";
372             Message2 = "dir: Folder " + cmd[0] + " doesn't Exist.\n";
373             State = 1;
374             return;
375         }
376         string Files[1024]; int Num = 0;
377         //读取文件信息
378         while((ptr = readdir(pDir)) != NULL)
379             Files[Num++] = ptr->d_name;
380         closedir(pDir);
381         //对结果排序
382         QuickSort(Files, 0, Num - 1);
383         Message1 = "";
384         Message2 = "";
385         State = 0;
386         //前两个文件分别是.和.., 不用输出
387         for (int i = 2; i < Num; i++)
388             Message1 = Message1 + Files[i] + "\n";
389     }
390 }
391
392 //echo打印内容到屏幕
393 void _echo(string cmd[], int ParaNum){
394     Message1 = "";
395     Message2 = "";
396     State = 0;
397     //遍历每个参数
398     for (int i = 0; i < ParaNum; i++){
399         string Value = GetValue(cmd[i]);
400         if (Value != "") Message1 += Value + " ";
401     }
402     //换行
403     if (Message1.length()) Message1[Message1.length() - 1] = '\n';

```

```

404     else Message1 = "\n";
405 }
406
407 //执行exec指令
408 void _exec(string cmd[], int ParaNum){
409     if (ParaNum == 0){
410         Message1 = "";
411         Message2 = "exec: Expected at least a parameter.\n";
412         State = 1;
413         return;
414     }
415     //设置参数
416     char * arg[ParaNum + 1];
417     for (int i = 0; i < ParaNum; i++)
418         arg[i] = const_cast<char *>(cmd[i].c_str());
419     arg[ParaNum] = NULL;
420     Message1 = "";
421     Message2 = "";
422     State = 0;
423     //调用execvp函数
424     execvp(cmd[0].c_str(), arg);
425     //exec执行成功后会退出源程序。如果能执行到下面的语句，说明exec出错
426     Message1 = "";
427     Message2 = "exec: Fail to execute " + cmd[0] + ".\n";
428     State = 1;
429 }
430
431 //exit退出程序
432 void _exit(string cmd[], int ParaNum){
433     Message1 = "";
434     Message2 = "";
435     State = 0;
436     exit(0);
437 }
438
439 //fg将后台作业移至前台执行
440 void _fg(string cmd[], int ParaNum){
441     Message1 = "";
442     Message2 = "";
443     State = 0;
444     //无参
445     if (ParaNum == 0){
446         Message1 = "";
447         Message2 = "fg: Please input the mission ID.\n";
448         State = 1;
449         return;
450     }
451     //多于一个参数
452     if (ParaNum >= 2){
453         Message1 = "";
454         Message2 = "fg: Too many parameters.\n";
455         State = 1;
456         return;
457     }
458     //根据参数列表，把对应的任务移至前台运行
459     int WorkID;
460     try{
461         WorkID = StringToInt(cmd[0]);

```



```

462     }catch(...){
463         Message1 = "";
464         Message2 = "fg: " + cmd[0] + ": Not a valid positive integer.\n";
465         State = 1;
466         return;
467     }
468     if (WorkID > Rear || WorkID <= Front || States[WorkID - 1] == 0){
469         Message1 = "";
470         Message2 = "fg: " + cmd[0] + ": No mission.\n";
471         State = 1;
472         return;
473     }
474     Message1 = "";
475     Message2 = "";
476     State = 0;
477     CurrentCMD = CMDInfo[WorkID - 1];
478     if (InIsTerminal){//把指令内容输出到屏幕, 告知用户前台作业的信息
479         write(TerminalOut, CMDInfo[WorkID - 1].c_str(), CMDInfo[WorkID -
1].length());
480         write(TerminalOut, "\n", 1);
481     }
482     //更新Jobs表
483     States[WorkID - 1] = 0;
484     SubPID = Jobs[WorkID - 1];
485     if (Rear == WorkID && Front == WorkID - 1) Front = Rear = 0;
486     else if (Front == WorkID - 1) Front++;
487     else if (Rear == WorkID) Rear--;
488     //发送信号
489     setpgid(SubPID, getgid());//设置进程组, 使子进程进入前台进程组
490     kill(SubPID, SIGCONT);
491     //等待子进程完成
492     while (SubPID != -1 && !waitpid(SubPID, NULL, WNOHANG));
493     SubPID = -1;
494 }
495
496 //help输出帮助信息
497 void _help(string cmd[], int ParaNum){
498     if (ParaNum >= 2){
499         Message1 = "";
500         Message2 = "help: Too many parameters.\n";
501         State = 1;
502         return;
503     }
504     string Target;
505     if (ParaNum == 0) Target = "global";//若无参, 输出全局帮助手册
506     else Target = cmd[0];//否则输出对应指令的帮助手册
507
508     FILE * fp = fopen(HelpPath.c_str(), "r");
509     if (fp == NULL){
510         Message1 = "";
511         Message2 = "help: Help manual file is not found.\n";
512         State = 1;
513         return;
514     }
515     while(1){
516         char buf[1024];
517         fgets(buf, 1024, fp); for(int i = 0; buf[i] || (buf[i - 2] =
'\0'); i++);

```

```

518         if (buf[0] == '#' && buf[1] == '#' && buf[2] == '#'){//"###"是分隔
符
519             if (buf[3] == '\\0'){
520                 Message1 = "";
521                 Message2 = "help: There's no help manual for " + Target +
".\\n";
522                 State = 1;
523                 fclose(fp);
524                 return;
525             }
526             if (Target == buf + 3) break;
527         }
528     }
529     Message1 = "";
530     Message2 = "";
531     State = 0;
532     while(1){
533         char buf[1024];
534         fgets(buf, 1024, fp); for(int i = 0; buf[i] || (buf[i - 2] =
'\\0'); i++);
535         if (buf[0] == '#' && buf[1] == '#' && buf[2] == '#')
536             break;
537         else
538             Message1 = Message1 + buf + "\\n";
539     }
540     fclose(fp);
541 }
542
543 //jobs输出当前正在执行或被挂起的任务
544 void _jobs(string cmd[], int ParaNum){
545     if (ParaNum > 0){
546         Message1 = "";
547         Message2 = "jobs: Too many parameters.\\n";
548         State = 1;
549         return;
550     }
551     Message1 = "";
552     for (int i = Front; i < Rear; i++)
553         Message1 += JobString(i);
554     Message2 = "";
555     State = 0;
556 }
557
558 //pwd列出当前目录
559 void _pwd(string cmd[], int ParaNum){
560     if (ParaNum > 0){//参数过多, 报错
561         Message1 = "";
562         Message2 = "pwd: Too many parameters.\\n";
563         State = 1;
564         return;
565     }
566     Message1 = PWD + "\\n";
567     Message2 = "";
568     State = 0;
569 }
570
571 //set命令列出当前所有环境变量, 或设置全局变量的值
572 void _set(string cmd[], int ParaNum){

```

```

573     Message1 = "";
574     Message2 = "";
575     State = 0;
576     extern char** environ; //环境变量表
577     if (ParaNum > 0) { //参数不为0, 则改变全局变量
578         string TempArgv[1024];
579         int TempArgc;
580         //根据string cmd[], 确定参数的值和个数, 存入TempArgv和TempArgc
581         for (TempArgc = 1; TempArgc <= 1023 && TempArgc <= ParaNum;
TempArgc++)
582             TempArgv[TempArgc] = GetValue(cmd[TempArgc - 1]);
583         //更新argc
584         argc = TempArgc;
585         //更新argv
586         for (int i = 1; i < TempArgc; i++)
587             argv[i] = TempArgv[i];
588     } else { //参数个数为0, 则输出所有环境变量
589         for (int i = 0; environ[i] != NULL; i++)
590             Message1 = Message1 + environ[i] + "\n";
591     }
592 }
593
594 //shift命令, 左移全局变量
595 void _shift(string cmd[], int ParaNum){
596     if (ParaNum >= 2){
597         Message1 = "";
598         Message2 = "shift: Too many parameters.\n";
599         State = 1;
600     }else{
601         if (ParaNum == 0) cmd[0] = "1"; //若无参, 则默认左移1次
602         int cnt;
603         try{
604             cnt = StringToInt(cmd[0]);
605             if (cnt < 0) throw invalid_argument("");
606         }catch(...){
607             Message1 = "";
608             Message2 = "shift: " + cmd[0] + " is not a valid nonnegative
integer.\n";
609             State = 1;
610             return;
611         }
612         if (argc <= cnt + 1) //若变量个数 <= 左移次数, 则直接把argc置1即可
613             argc = 1;
614         else
615             argc -= cnt;
616         //进行左移
617         for (int i = 1; i < argc; i++)
618             argv[i] = argv[i + cnt];
619         Message1 = "";
620         Message2 = "";
621         State = 0;
622     }
623 }
624
625 //test命令
626 void _test(string cmd[], int ParaNum){
627     if (ParaNum <= 1){
628         Message1 = "";

```

```

629     Message2 = "test: Too few parameters.\n";
630     State = 1;
631     return;
632 }
633 if (ParaNum >= 4){
634     Message1 = "";
635     Message2 = "test: Too many parameters.\n";
636     State = 1;
637     return;
638 }
639 if (ParaNum == 2){//一元运算符, 有且仅有2个参数
640     string ValueStr = GetValue(cmd[1]);
641     State = 0; Message1 = Message2 = "";
642     if (cmd[0] == "-e"){//文件存在
643         struct stat buf;
644         int ret = lstat(ValueStr.c_str(), &buf);
645         if (ret == 0) Message1 = "true\n";
646         else Message1 = "false\n";
647     }else if (cmd[0] == "-r"){//文件可读
648         struct stat buf;
649         int ret = lstat(ValueStr.c_str(), &buf);
650         if (ret == 0 && access(ValueStr.c_str(), R_OK)) Message1 =
"true\n";
651         else Message1 = "false\n";
652     }else if (cmd[0] == "-w"){//文件可写
653         struct stat buf;
654         int ret = lstat(ValueStr.c_str(), &buf);
655         if (ret == 0 && access(ValueStr.c_str(), W_OK)) Message1 =
"true\n";
656         else Message1 = "false\n";
657     }else if (cmd[0] == "-x"){//文件可执行
658         struct stat buf;
659         int ret = lstat(ValueStr.c_str(), &buf);
660         if (ret == 0 && access(ValueStr.c_str(), X_OK)) Message1 =
"true\n";
661         else Message1 = "false\n";
662     }else if (cmd[0] == "-s"){//文件至少有一个字符
663         struct stat buf;
664         int ret = lstat(ValueStr.c_str(), &buf);
665         if (ret == 0 && buf.st_size) Message1 = "true\n";
666         else Message1 = "false\n";
667     }else if (cmd[0] == "-d"){//文件为目录
668         struct stat buf;
669         int ret = lstat(ValueStr.c_str(), &buf);
670         if (ret == 0 && S_ISDIR(buf.st_mode)) Message1 = "true\n";
671         else Message1 = "false\n";
672     }else if (cmd[0] == "-f"){//文件为普通文件
673         struct stat buf;
674         int ret = lstat(ValueStr.c_str(), &buf);
675         if (ret == 0 && S_ISREG(buf.st_mode)) Message1 = "true\n";
676         else Message1 = "false\n";
677     }else if (cmd[0] == "-c"){//文件为字符型特殊文件
678         struct stat buf;
679         int ret = lstat(ValueStr.c_str(), &buf);
680         if (ret == 0 && S_ISCHR(buf.st_mode)) Message1 = "true\n";
681         else Message1 = "false\n";
682     }else if (cmd[0] == "-b"){//文件为块特殊文件
683         struct stat buf;

```

```

684         int ret = lstat(ValueStr.c_str(), &buf);
685         if (ret == 0 && S_ISBLK(buf.st_mode)) Message1 = "true\n";
686         else Message1 = "false\n";
687     }else if (cmd[0] == "-h" || cmd[0] == "-L"){//文件为符号链接
688         struct stat buf;
689         int ret = lstat(ValueStr.c_str(), &buf);
690         if (ret == 0 && S_ISLNK(buf.st_mode)) Message1 = "true\n";
691         else Message1 = "false\n";
692     }else if (cmd[0] == "-p"){//文件为命名管道
693         struct stat buf;
694         int ret = lstat(ValueStr.c_str(), &buf);
695         if (ret == 0 && S_ISFIFO(buf.st_mode)) Message1 = "true\n";
696         else Message1 = "false\n";
697     }else if (cmd[0] == "-S"){//文件为嵌套字
698         struct stat buf;
699         int ret = lstat(ValueStr.c_str(), &buf);
700         if (ret == 0 && S_ISSOCK(buf.st_mode)) Message1 = "true\n";
701         else Message1 = "false\n";
702     }else if (cmd[0] == "-G"){//文件被实际组拥有
703         struct stat buf;
704         int ret = lstat(ValueStr.c_str(), &buf);
705         if (ret == 0 && buf.st_gid == getgid()) Message1 = "true\n";
706         else Message1 = "false\n";
707     }else if (cmd[0] == "-O"){//文件被实际用户拥有
708         struct stat buf;
709         int ret = lstat(ValueStr.c_str(), &buf);
710         if (ret == 0 && buf.st_uid == getuid()) Message1 = "true\n";
711         else Message1 = "false\n";
712     }else if (cmd[0] == "-g"){//文件有设置组位
713         struct stat buf;
714         int ret = lstat(ValueStr.c_str(), &buf);
715         if (ret == 0 && (S_ISGID & buf.st_mode)) Message1 = "true\n";
716         else Message1 = "false\n";
717     }else if (cmd[0] == "-u"){//文件有设置用户位
718         struct stat buf;
719         int ret = lstat(ValueStr.c_str(), &buf);
720         if (ret == 0 && (S_ISUID & buf.st_mode)) Message1 = "true\n";
721         else Message1 = "false\n";
722     }else if (cmd[0] == "-k"){//文件有设置粘滞位
723         struct stat buf;
724         int ret = lstat(ValueStr.c_str(), &buf);
725         if (ret == 0 && (S_ISVTX & buf.st_mode)) Message1 = "true\n";
726         else Message1 = "false\n";
727     }else if (cmd[0] == "-n"){//字符串长度非0
728         if (ValueStr.length()) Message1 = "true\n";
729         else Message1 = "false\n";
730     }else if (cmd[0] == "-z"){//字符串长度为0
731         if (ValueStr.length() == 0) Message1 = "true\n";
732         else Message1 = "false\n";
733     }else{
734         State = 1;
735         Message2 = "test: Unknown command " + cmd[0] + ".\n";
736     }
737 }
738 if (ParaNum == 3){//二元运算符, 有且仅有3个参数
739     string ValueStr1 = GetValue(cmd[0]);
740     string ValueStr2 = GetValue(cmd[2]);
741     State = 0; Message1 = Message2 = "";

```

```

742         if (cmd[1] == "-ef"){//文件1和文件2的设备和inode相同
743             struct stat buf1, buf2;
744             int ret1 = lstat(ValueStr1.c_str(), &buf1);
745             int ret2 = lstat(ValueStr2.c_str(), &buf2);
746             if (ret1 == 0 && ret2 == 0 && buf1.st_dev == buf2.st_dev &&
buf1.st_ino == buf2.st_ino) Message1 = "true\n";
747             else Message1 = "false\n";
748         }else if (cmd[1] == "-nt"){//文件1比文件2更新
749             struct stat buf1, buf2;
750             int ret1 = lstat(ValueStr1.c_str(), &buf1);
751             int ret2 = lstat(ValueStr2.c_str(), &buf2);
752             if (ret1 == 0 && ret2 == 0 && TimeCMP(buf1.st_mtim,
buf2.st_mtim) == 1) Message1 = "true\n";
753             else Message1 = "false\n";
754         }else if (cmd[1] == "-ot"){//文件1比文件2更旧
755             struct stat buf1, buf2;
756             int ret1 = lstat(ValueStr1.c_str(), &buf1);
757             int ret2 = lstat(ValueStr2.c_str(), &buf2);
758             if (ret1 == 0 && ret2 == 0 && TimeCMP(buf1.st_mtim,
buf2.st_mtim) == -1) Message1 = "true\n";
759             else Message1 = "false\n";
760         }else if (cmd[1] == "="){//字符串相等
761             if (ValueStr1 == ValueStr2) Message1 = "true\n";
762             else Message1 = "false\n";
763         }else if (cmd[1] == "!="){//字符串不等
764             if (ValueStr1 != ValueStr2) Message1 = "true\n";
765             else Message1 = "false\n";
766         }else if (cmd[1] == "-eq"){//整数==
767             int Value1, Value2;
768             try{
769                 Value1 = StringToInt(ValueStr1);
770             }catch(...){
771                 State = 1;
772                 Message2 = "test: " + ValueStr1 + " is not a valid
integer.\n";
773                 return;
774             }
775             try{
776                 Value2 = StringToInt(ValueStr2);
777             }catch(...){
778                 State = 1;
779                 Message2 = "test: " + ValueStr2 + " is not a valid
integer.\n";
780                 return;
781             }
782             if (Value1 == Value2) Message1 = "true\n";
783             else Message1 = "false\n";
784         }else if (cmd[1] == "-ge"){//整数>=
785             int Value1, Value2;
786             try{
787                 Value1 = StringToInt(ValueStr1);
788             }catch(...){
789                 State = 1;
790                 Message2 = "test: " + ValueStr1 + " is not a valid
integer.\n";
791                 return;
792             }
793             try{

```

```

794         Value2 = StringToInt(ValueStr2);
795     }catch(...){
796         State = 1;
797         Message2 = "test: " + ValueStr2 + " is not a valid
integer.\n";
798         return;
799     }
800     if (Value1 >= Value2) Message1 = "true\n";
801     else Message1 = "false\n";
802 }else if (cmd[1] == "-gt"){//整数>
803     int Value1, Value2;
804     try{
805         Value1 = StringToInt(ValueStr1);
806     }catch(...){
807         State = 1;
808         Message2 = "test: " + ValueStr1 + " is not a valid
integer.\n";
809         return;
810     }
811     try{
812         Value2 = StringToInt(ValueStr2);
813     }catch(...){
814         State = 1;
815         Message2 = "test: " + ValueStr2 + " is not a valid
integer.\n";
816         return;
817     }
818     if (Value1 > Value2) Message1 = "true\n";
819     else Message1 = "false\n";
820 }else if (cmd[1] == "-le"){//整数<=
821     int Value1, Value2;
822     try{
823         Value1 = StringToInt(ValueStr1);
824     }catch(...){
825         State = 1;
826         Message2 = "test: " + ValueStr1 + " is not a valid
integer.\n";
827         return;
828     }
829     try{
830         Value2 = StringToInt(ValueStr2);
831     }catch(...){
832         State = 1;
833         Message2 = "test: " + ValueStr2 + " is not a valid
integer.\n";
834         return;
835     }
836     if (Value1 <= Value2) Message1 = "true\n";
837     else Message1 = "false\n";
838 }else if (cmd[1] == "-lt"){//整数<
839     int Value1, Value2;
840     try{
841         Value1 = StringToInt(ValueStr1);
842     }catch(...){
843         State = 1;
844         Message2 = "test: " + ValueStr1 + " is not a valid
integer.\n";
845         return;

```

```

846     }
847     try{
848         Value2 = StringToInt(ValueStr2);
849     }catch(...){
850         State = 1;
851         Message2 = "test: " + ValueStr2 + " is not a valid
integer.\n";
852         return;
853     }
854     if (Value1 < Value2) Message1 = "true\n";
855     else Message1 = "false\n";
856 }else if (cmd[1] == "-ne"){//整数!=
857     int Value1, Value2;
858     try{
859         Value1 = StringToInt(ValueStr1);
860     }catch(...){
861         State = 1;
862         Message2 = "test: " + ValueStr1 + " is not a valid
integer.\n";
863         return;
864     }
865     try{
866         Value2 = StringToInt(ValueStr2);
867     }catch(...){
868         State = 1;
869         Message2 = "test: " + ValueStr2 + " is not a valid
integer.\n";
870         return;
871     }
872     if (Value1 != Value2) Message1 = "true\n";
873     else Message1 = "false\n";
874 }else{//无法识别的运算符
875     State = 1;
876     Message2 = "test: Unknown command " + cmd[0] + ".\n";
877 }
878 }
879 }
880
881 //time列出当前时间
882 void _time(string cmd[], int ParaNum){
883     if (ParaNum > 0){//参数过多, 报错
884         Message1 = "";
885         Message2 = "time: Too many parameters.\n";
886         State = 1;
887         return;
888     }
889     //得到当前时间
890     time_t tt = time(NULL);
891     struct tm * t = localtime(&tt);
892     //用stringstream生成返回信息
893     stringstream sstm;
894     const char * Week[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"};
895     sstm << t->tm_year + 1900 << "." << t->tm_mon + 1 << "." << t->tm_mday
<< "."
896         << " " << Week[t->tm_wday]
897         << " " << t->tm_hour << ":" << t->tm_min << ":" << t->tm_sec;
898     getline(sstm, Message1);

```



```

899     Message1 += "\n";
900     Message2 = "";
901     State = 0;
902 }
903
904 //输出umask或更改umask的值
905 void _umask(string cmd[], int ParaNum){
906     if (ParaNum > 1){//参数过多, 报错
907         Message1 = "";
908         Message2 = "umask: Too many parameters.\n";
909         State = 1;
910     } else if (ParaNum == 1){//只有一个参数, 表示设置umask的值
911         if (cmd[0].length() >= 5){//参数多于4位, 报错
912             Message1 = "";
913             Message2 = "umask: Expected at most 4 octonary digits: " +
cmd[0] + "\n";
914             State = 1;
915         }else{
916             while (cmd[0].length() < 4) cmd[0] = "0" + cmd[0];//补齐到恰好4
位
917             if (cmd[0][0] < '0' || cmd[0][0] > '7'){//判断是否为8进制数
918                 Message1 = "";
919                 Message2 = to_string(cmd[0][0] - '0') + " is not an
octonary digit.\n";
920                 State = 1;
921             }else if (cmd[0][1] < '0' || cmd[0][1] > '7'){//判断是否为8进制
数
922                 Message1 = "";
923                 Message2 = to_string(cmd[0][1] - '0') + " is not an
octonary digit.\n";
924                 State = 1;
925             }else if (cmd[0][2] < '0' || cmd[0][2] > '7'){//判断是否为8进制
数
926                 Message1 = "";
927                 Message2 = to_string(cmd[0][2] - '0') + " is not an
octonary digit.\n";
928                 State = 1;
929             }else if (cmd[0][3] < '0' || cmd[0][3] > '7'){//判断是否为8进制
数
930                 Message1 = "";
931                 Message2 = to_string(cmd[0][3] - '0') + " is not an
octonary digit.\n";
932                 State = 1;
933             }else{
934                 int newmode = ((cmd[0][0] - '0') << 9) | ((cmd[0][1] -
'0') << 6) | ((cmd[0][2] - '0') << 3) | (cmd[0][3] - '0');
935                 umask(newmode);
936                 Message1 = "";
937                 Message2 = "";
938                 State = 0;
939             }
940         }
941     }else{//无参, 表示显示umask的值
942         mode_t currentmode = umask(0);
943         umask(currentmode);
944         Message1 = to_string((currentmode >> 9) & 7) +
to_string((currentmode >> 6) & 7)

```

```

945         + to_string((currentmode >> 3) & 7) +
to_string(currentmode & 7) + "\n";
946     Message2 = "";
947     State = 0;
948 }
949 }
950
951 //unset命令删除环境变量
952 void _unset(string cmd[], int ParaNum){
953     if (ParaNum > 1){
954         Message1 = "";
955         Message2 = "unset: Too many parameters.\n";
956         State = 1;
957     } else if (ParaNum == 0){
958         Message1 = "";
959         Message2 = "unset: Input a variable's name.\n";
960         State = 1;
961     } else {
962         Message1 = "";
963         Message2 = "";
964         State = 0;
965         unsetenv(cmd[0].c_str());
966     }
967 }
968
969
970 //第三层解析指令：执行单条指令，可能包含重定向
971 void ExecSingleCMD(string cmd[], int ParaNum, bool WithFork){
972     //备份三个标准输入输出
973     int InputFD = dup(STDIN_FILENO), OutputFD = dup(STDOUT_FILENO),
ErrorFD = dup(STDERR_FILENO);
974     int InputFDNew = -1, OutputFDNew = -1, ErrorFDNew = -1;
975     //搜索重定向符号
976     State = 0;
977     string InputFile = "", OutputFile = "", ErrorFile = "";
978     for (int i = ParaNum - 2; i >= 0; i--) { //注意，i从ParaNum-2开始枚举，因
为重定向符号不可能是最后一个字符串
979         //输入重定向
980         if (cmd[i] == "<" || cmd[i] == "<"){
981             if (InputFile != ""){
982                 Message1 = "";
983                 Message2 = "MyShell: Expected at most 1 input
redirection.\n";
984                 State = 1;
985                 break;
986             }
987             InputFile = cmd[i + 1];
988             InputFDNew = open(InputFile.c_str(), O_RDONLY);
989             if (InputFDNew < 0){
990                 Message1 = "";
991                 Message2 = "MyShell: Unable to open " + InputFile + ".\n";
992                 State = 1;
993                 break;
994             }
995             dup2(InputFDNew, STDIN_FILENO);
996             close(InputFDNew);
997             ParaNum = i;
998         }

```

```

999         //输出重定向, 覆盖模式
1000         else if (cmd[i] == ">" || cmd[i] == "1>"){
1001             if (OutputFile != ""){
1002                 Message1 = "";
1003                 Message2 = "MyShell: Expected at most 1 output
redirection.\n";
1004                 State = 1;
1005                 break;
1006             }
1007             OutputFile = cmd[i + 1];
1008             OutputFDNew = open(OutputFile.c_str(), O_WRONLY | O_CREAT |
O_TRUNC, 0666);
1009             if(OutputFDNew < 0){
1010                 Message1 = "";
1011                 Message2 = "MyShell: Unable to open " + OutputFile +
".\n";
1012                 State = 1;
1013                 break;
1014             }
1015             dup2(OutputFDNew, STDOUT_FILENO);
1016             close(OutputFDNew);
1017             ParaNum = i;
1018         }
1019         //输出重定向, 追加模式
1020         else if (cmd[i] == ">>" || cmd[i] == "1>>"){
1021             if (OutputFile != ""){
1022                 Message1 = "";
1023                 Message2 = "MyShell: Expected at most 1 output
redirection.\n";
1024                 State = 1;
1025                 break;
1026             }
1027             OutputFile = cmd[i + 1];
1028             OutputFDNew = open(OutputFile.c_str(), O_WRONLY | O_CREAT |
O_APPEND, 0666);
1029             if(OutputFDNew < 0){
1030                 Message1 = "";
1031                 Message2 = "MyShell: Unable to open " + OutputFile +
".\n";
1032                 State = 1;
1033                 break;
1034             }
1035             dup2(OutputFDNew, STDOUT_FILENO);
1036             close(OutputFDNew);
1037             ParaNum = i;
1038         }
1039         //错误重定向, 覆盖模式
1040         else if (cmd[i] == "2>"){
1041             if (ErrorFile != ""){
1042                 Message1 = "";
1043                 Message2 = "MyShell: Expected at most 1 error
redirection.\n";
1044                 State = 1;
1045                 break;
1046             }
1047             ErrorFile = cmd[i + 1];
1048             ErrorFDNew = open(ErrorFile.c_str(), O_WRONLY | O_CREAT |
O_TRUNC, 0666);

```

```

1049         if(ErrorFDNew < 0){
1050             Message1 = "";
1051             Message2 = "MyShell: Unable to open " + ErrorFile + ".\n";
1052             State = 1;
1053             break;
1054         }
1055         dup2(ErrorFDNew, STDERR_FILENO);
1056         close(ErrorFDNew);
1057         ParaNum = i;
1058     }
1059     //错误重定向, 追加模式
1060     else if (cmd[i] == "2>>"){
1061         if (ErrorFile != ""){
1062             Message1 = "";
1063             Message2 = "MyShell: Expected at most 1 error
1064 redirection.\n";
1065             State = 1;
1066             break;
1067         }
1068         ErrorFile = cmd[i + 1];
1069         ErrorFDNew = open(ErrorFile.c_str(), O_WRONLY | O_CREAT |
O_APPEND, 0666);
1070         if(ErrorFDNew < 0){
1071             Message1 = "";
1072             Message2 = "MyShell: Unable to open " + ErrorFile + ".\n";
1073             State = 1;
1074             break;
1075         }
1076         dup2(ErrorFDNew, STDERR_FILENO);
1077         close(ErrorFDNew);
1078         ParaNum = i;
1079     }
1080     if (State == 0){//如果前面的重定向处理没有出错, 则继续
1081         //解析指令
1082         if (ParaNum == 0 || cmd[0][0] == '#') {
1083             Message1 = "";
1084             Message2 = "";
1085             State = 0;
1086         }else if (cmd[0] == "bg") {
1087             _bg(cmd + 1, ParaNum - 1);
1088         }else if (cmd[0] == "cd") {
1089             _cd(cmd + 1, ParaNum - 1);
1090         }else if (cmd[0] == "clr") {
1091             _clr(cmd + 1, ParaNum - 1);
1092         }else if (cmd[0] == "dir") {
1093             _dir(cmd + 1, ParaNum - 1);
1094         }else if (cmd[0] == "echo") {
1095             _echo(cmd + 1, ParaNum - 1);
1096         }else if (cmd[0] == "exec") {
1097             _exec(cmd + 1, ParaNum - 1);
1098         }else if (cmd[0] == "exit") {
1099             _exit(cmd + 1, ParaNum - 1);
1100         }else if (cmd[0] == "fg") {
1101             _fg(cmd + 1, ParaNum - 1);
1102         }else if (cmd[0] == "help") {
1103             _help(cmd + 1, ParaNum - 1);
1104         }else if (cmd[0] == "jobs") {

```

```

1105     _jobs(cmd + 1, ParaNum - 1);
1106 }else if (cmd[0] == "pwd") {
1107     _pwd(cmd + 1, ParaNum - 1);
1108 }else if (cmd[0] == "set") {
1109     _set(cmd + 1, ParaNum - 1);
1110 }else if (cmd[0] == "shift") {
1111     _shift(cmd + 1, ParaNum - 1);
1112 }else if (cmd[0] == "test") {
1113     _test(cmd + 1, ParaNum - 1);
1114 }else if (cmd[0] == "time") {
1115     _time(cmd + 1, ParaNum - 1);
1116 }else if (cmd[0] == "umask") {
1117     _umask(cmd + 1, ParaNum - 1);
1118 }else if (cmd[0] == "unset") {
1119     _unset(cmd + 1, ParaNum - 1);
1120 }else if (cmd[0] == "exit") {
1121     _exit(cmd + 1, ParaNum - 1);
1122 }else{//其他命令，表示程序调用
1123     Message1 = "";
1124     Message2 = "";
1125     State = 0;
1126     if (WithFork){
1127         //fork将父进程拷贝一份变成子进程
1128         SubPID = fork();
1129         if (SubPID == 0){//子进程
1130             //设置PARENT环境变量
1131             setenv("PARENT", ShellPath.c_str(), 1);
1132             _exec(cmd, ParaNum);
1133             //如果能执行到这一步，说明exec出错
1134             Message2 = "MyShell: Unable to execute " + cmd[0] +
1135 ".\n";
1136             write(STDERR_FILENO, Message2.c_str(),
1137 Message2.length());
1138             exit(0);
1139         }
1140         //父进程等待子进程完成
1141         while (SubPID != -1 && !waitpid(SubPID, NULL, WNOHANG));
1142         SubPID = -1;
1143     }else{
1144         //设置PARENT环境变量
1145         setenv("PARENT", ShellPath.c_str(), 1);
1146         _exec(cmd, ParaNum);
1147         //如果能执行到这一步，说明exec出错
1148         Message2 = "MyShell: Unable to execute " + cmd[0] + ".\n";
1149     }
1150 }
1151 //输出结果
1152 write(STDOUT_FILENO, Message1.c_str(), Message1.length());
1153 write(STDERR_FILENO, Message2.c_str(), Message2.length());
1154 //恢复三个标准输入输出
1155 dup2(InputFD, STDIN_FILENO); dup2(OutputFD, STDOUT_FILENO);
1156 dup2(ErrorFD, STDERR_FILENO);
1157 close(InputFD); close(OutputFD); close(ErrorFD);
1158 }
1159 //第二层解析指令：执行多条用管道符分隔的指令。单条指令交给ExecSingleCMD函数
1160 void ExecMultiCMD(string cmd[], int ParaNum, bool WithFork){

```

```

1160 //如果整个指令串都没有管道符，则直接在当前进程中执行
1161 bool PipeFlag = false;
1162 for (int i = 0; i < ParaNum; i++)
1163     if (cmd[i] == "|"){
1164         PipeFlag = true;
1165         break;
1166     }
1167 if (PipeFlag == false){
1168     ExecSingleCMD(cmd, ParaNum, WithFork);
1169     return;
1170 }
1171 //否则，生成一个子进程，执行用管道符连接的多条指令
1172 if (WithFork) SubPID = fork();
1173 if (WithFork && SubPID){//父进程
1174     while (SubPID != -1 && !waitpid(SubPID, NULL, WNOHANG));
1175     SubPID = -1;
1176 }else{//子进程
1177     //将信号处理函数恢复至系统默认
1178     signal(SIGINT, SIG_DFL);
1179     signal(SIGTSTP, SIG_DFL);
1180     //标记上一个管道符的位置
1181     int LastPipe = -1;
1182     //管道的文件描述符，只需要记录两个管道即可（但整个过程可能会创建很多次管道）
1183     int FirstFD[2], SecondFD[2];
1184     //执行管道符指令时，所有子进程的pid。之所以要存储该信息，是为了在解析指令结束后使用waitpid等待所有子进程完成
1185     const int MaxPipe = 1024;
1186     int PipePid[MaxPipe];
1187     int PipeCNT;
1188     //在末尾临时添加一个管道符
1189     cmd[ParaNum++] = "|";
1190     //扫描，找出所有的管道符
1191     PipeCNT = 0;
1192     for (int i = 0; i < ParaNum; i++)
1193         if (cmd[i] == "|"){//遇到管道符，把LastPipe + 1到i - 1之间的指令提
//取出来，作为子进程执行
1194             //创建管道
1195             if (LastPipe == -1) {//第一个遇到的管道符
1196                 FirstFD[0] = STDIN_FILENO;
1197                 FirstFD[1] = -1;
1198                 pipe(SecondFD);
1199             }else if (i == ParaNum - 1){//最后一个管道符
1200                 if (FirstFD[0] != STDIN_FILENO) close(FirstFD[0]);//防
//止误关STDIN
1201                 FirstFD[0] = SecondFD[0]; FirstFD[1] = SecondFD[1];
1202                 close(FirstFD[1]);
1203                 SecondFD[0] = -1; SecondFD[1] = STDOUT_FILENO;
1204             }else{//既不是第一个管道符也不是最后一个管道符
1205                 if (FirstFD[0] != STDIN_FILENO) close(FirstFD[0]);//防
//止误关STDIN
1206                 FirstFD[0] = SecondFD[0]; FirstFD[1] = SecondFD[1];
1207                 close(FirstFD[1]);
1208                 pipe(SecondFD);
1209             }
1210             //拷贝进程
1211             PipePid[PipeCNT++] = fork();
1212             if (PipePid[PipeCNT - 1] == 0){ //子进程
1213                 //设置信号处理函数

```

```

1212         signal(SIGINT, SIG_IGN);
1213         signal(SIGTSTP, SIG_DFL);
1214         //重定向
1215         dup2(FirstFD[0], STDIN_FILENO);
1216         dup2(SecondFD[1], STDOUT_FILENO);
1217         close(FirstFD[1]); close(SecondFD[0]);
1218         //执行指令
1219         ExecSingleCMD(cmd + LastPipe + 1, i - LastPipe - 1,
false);
1220         exit(0);
1221     }
1222     LastPipe = i;
1223 }
1224 close(FirstFD[0]);
1225 //等待所有子进程完成
1226 for (int i = 0; i < PipeCNT; i++)
1227     waitpid(PipePid[i], NULL, 0);
1228 exit(0);
1229 }
1230 }
1231
1232 //第一层解析指令：处理"&"后台运行符号，其余交给ExecMultipCMD函数
1233 void Exec(string CMD){
1234     //执行之前，先扫描Jobs表，输出已完成的进程，并更新Jobs表
1235     for (int i = Front; i < Rear; i++){
1236         if (States[i] && waitpid(Jobs[i], NULL, WNOHANG) == Jobs[i]) {
1237             if (InIsTerminal){//只有输入来自终端的时候才要打印子进程表
1238                 string str = JobString(i, 1);
1239                 write(TerminalOut, str.c_str(), str.length());//注意：子进程
表只输出到终端，不能输出到STDOUT!!!!
1240             }
1241             States[i] = 0;
1242             if (Front == i) Front++;
1243         }
1244     }
1245     if (Front == Rear) Front = Rear = 0;
1246     //利用stringstream切割字符串
1247     stringstream sstm;
1248     int ParaNum = 0;//用空白字符分隔的参数个数
1249     string cmd[1024];//切割结果
1250     sstm << CMD;
1251     while(1){
1252         cmd[ParaNum] = "";
1253         sstm >> cmd[ParaNum];
1254         if (cmd[ParaNum] == "") break;
1255         ParaNum++;
1256     }
1257     //此时，指令串分隔完毕。检查末尾是否是&，表示后台运行
1258     if (ParaNum > 0 && cmd[ParaNum - 1] == "&") {
1259         ParaNum--;
1260         int pid = fork();
1261         if (pid){
1262             //父进程，存储子进程的id，指令等信息
1263             Jobs[Rear] = pid;
1264             States[Rear] = 1;
1265             CMDInfo[Rear] = "";
1266             for (int i = 0; i < ParaNum; i++) CMDInfo[Rear] += cmd[i] + "
";

```

```

1267         CurrentCMD = CMDInfo[Rear];
1268         Rear++;
1269         if (InIsTerminal){//只有输入来自终端的时候才要打印子进程表
1270             string str = JobString(Rear - 1);
1271             write(TerminalOut, str.c_str(), str.length());//注意：子进程
//只输出到终端，不能输出到STDOUT!!!!
1272         }
1273         }else{//子进程，执行指令
1274             setpgid(0, 0);//使子进程单独成为一个进程组。后台进程组自动忽略Ctrl+Z、
//Ctrl+C等信号
1275             ExecMultipCMD(cmd, ParaNum, false);
1276             exit(0);
1277         }
1278     }else{//没有&符号，前台运行
1279         CurrentCMD = CMD;
1280         ExecMultipCMD(cmd, ParaNum);
1281     }
1282 }
1283
1284 //信号处理函数，用于处理Ctrl+C、Ctrl+Z等组合键
1285 void SignalProcess(int Signal){
1286     switch (Signal){
1287         case SIGINT: //Ctrl+C，终止当前进程
1288             write(TerminalOut, "\n", 1);
1289             //什么都不需要做，因为子进程也会接收到SIGINT信号，子进程被终止
1290             break;
1291         case SIGTSTP: //Ctrl+Z，挂起当前进程
1292             write(TerminalOut, "\n", 1);
1293             if (SubPID != -1){
1294                 setpgid(SubPID, 0);
1295                 kill(SubPID, SIGTSTP);
1296                 Jobs[Rear] = SubPID;
1297                 States[Rear] = 2;
1298                 CMDInfo[Rear] = CurrentCMD;
1299                 Rear++;
1300                 if (InIsTerminal){//只有输入来自终端的时候才要打印子进程表
1301                     string str = JobString(Rear - 1);
1302                     write(TerminalOut, str.c_str(), str.length());//注意：
//子进程表只输出到终端，不能输出到STDOUT!!!!
1303                 }
1304                 SubPID = -1;
1305             }
1306             break;
1307         case SIGCONT: //继续执行任务信号
1308             break;
1309     }
1310 }
1311
1312 //初始化
1313 void Init(int Argc, char * Argv[]){
1314     char buffer[1024] = {0};
1315     //把参数赋值给全局变量
1316     argc = Argc;
1317     for (int i = 0; i < argc; i++) argv[i] = Argv[i];
1318     //若超过两个参数，则把标准输入重定向到argv[1]
1319     int InputFD = -1;
1320     if (argc >= 2){
1321         InputFD = open(argv[1].c_str(), O_RDONLY);

```



```

1322         if (InputFD < 0){
1323             sprintf(buffer, "MyShell: Unable to open %s.\n",
argv[1].c_str());
1324             write(STDERR_FILENO, buffer, 1024);
1325             _exit(0);
1326         }
1327         dup2(InputFD, STDIN_FILENO);
1328         close(InputFD);
1329     }
1330     //得到进程号
1331     PID = getpid();
1332     SubPID = -1;
1333     //得到当前主机名
1334     gethostname(buffer, 1024);
1335     HostName = buffer;
1336     //得到当前用户名
1337     UserName = getenv("USERNAME");
1338     //获取主目录地址
1339     HomeDir = getenv("HOME");
1340     //获取当前地址
1341     PWD = getenv("PWD");
1342     //帮助手册路径
1343     HelpPath = PWD + "/help";
1344     //获取程序自身的路径
1345     int len = readlink("/proc/self/exe", buffer, 1024);
1346     buffer[len] = '\0';
1347     ShellPath = buffer;
1348     setenv("SHELL", buffer, 1);
1349     //设置父进程的路径
1350     setenv("PARENT", "\\bin\\bash", 1);
1351     //初始化后台进程表
1352     Front = Rear = 0;
1353     //设置终端标准输入输出的文件描述符
1354     TerminalIn = open("/dev/tty", O_RDONLY);
1355     TerminalOut = open("/dev/tty", O_WRONLY);
1356     //判断标准输入输出是否来自终端
1357     struct stat FileInfo;
1358     fstat(STDIN_FILENO, &FileInfo);
1359     InIsTerminal = S_ISCHR(FileInfo.st_mode);
1360     fstat(STDOUT_FILENO, &FileInfo);
1361     OutIsTerminal = S_ISCHR(FileInfo.st_mode);
1362     //设置信号处理函数
1363     if (InIsTerminal){//只有输入来自终端时才对Ctrl+C、Ctrl+Z等快捷键进行作业控制
1364         signal(SIGINT, SignalProcess);
1365         signal(SIGTSTP, SignalProcess);
1366     }
1367 }

```