

实验2 程序设计——myshell

使用任何一种程序设计语言实现一个shell 程序的基本功能。

shell 或者命令行解释器是操作系统中最基本的用户接口。写一个简单的shell 程序——**myshell**，它具有以下属性：

(一) 这个shell 程序必须支持以下内部命令：bg、cd、clr、dir、echo、exec、exit、fg、help、jobs、pwd、set、shift、test、time、umask、unset。部分命令解释如下：

cd <directory> ——把当前默认目录改变为<directory>。如果没有<directory>参数，则显示当前目录。如该目录不存在，会出现合适的错误信息。这个命令也可以改变PWD 环境变量。

pwd ——显示当前目录。

time ——显示当前时间

clr ——清屏。

dir <directory> ——列出目录<directory>的内容。

set ——列出所有的环境变量。

echo <comment> ——在屏幕上显示<comment>并换行（多个空格和制表符可能被缩减为一个空格）。

help ——显示用户手册，并且使用more 命令过滤。

exit ——退出shell。

shell 的环境变量应该包含shell=<pathname>/myshell，其中<pathname>/myshell 是可执行程序shell 的完整路径（不是你的目录下的路径，而是它执行程序的路径）。

(二) 其他的命令行输入被解释为程序调用，shell 创建并执行这个程序，并作为自己的子进程。程序的执行的环境变量包含一下条目：

parent=<pathname>/myshell。

(三) shell 必须能够从文件中提取命令行输入，例如shell 使用以下命令行被调用：

myshell batchfile

这个批处理文件应该包含一组命令集，当到达文件结尾时shell 退出。很明显，如果shell 被调用时没有使用参数，它会在屏幕上显示提示符请求用户输入。

(四) shell 必须支持I/O 重定向，stdin 和stdout，或者其中之一，例如命令行为：

programname arg1 arg2 < inputfile > outputfile

使用arg1 和arg2 执行程序programname，输入文件流被替换为inputfile，输出文件流被替换为outputfile。

stdout 重定向应该支持以下内部命令：dir、environ、echo、help。

使用输出重定向时，如果重定向字符是>，则创建输出文件，如果存在则覆盖之；如果重定向字符为>>，也会创建输出文件，如果存在则添加到文件尾。

(五) shell 必须支持后台程序执行。如果在命令行后添加&字符，在加载完程序后需要立刻返回命令行提示符。

(六) 必须支持管道 (“|”) 操作。

(七) 命令行提示符必须包含当前路径。

设计

设计思路

开发中我采用C++进行开发，在Ubuntu 20.04实体机上运行。使用C系列的语言是考虑到Linux有许多库函数可以调用，也是Linux开发者常用的语言；而使用C++而不是C主要是因为C++有许多对象及方法，可以减少许多不太重要的工作，方便开发。

开始编程之前，仔细阅读题目后，参考Linux的shell的实现，我讲题目要求实现的功能主要分为两种：一种是内建命令，即 `dir`、`time` 等命令的具体实现；一种是shell管理功能，如控制命令后台执行、管道操作等功能。因此对应地建立 `commands.cpp` 和 `myshell.cpp` 还有对应的头文件。开发中发现还有一部分宏定义是可以从二者中抽取出来的，又专门建立了一个 `base.h` 文件来存放这一部分宏定义。开发中还发现有一些进程管理和信号处理的函数需要单独处理，但是为了方便就放在了 `commands.cpp` 中。因此用到的文件有六个：`Makefile`、`base.h`、`commands.h`、`myshell.h`、`commands.cpp` 与 `myshell.cpp`。

数据结构

- `job`
 - 为了方便管理进程，自定义了一种结构体来记录进程的信息。这个结构体包括的成员有：进程的pid，进程的名字，进程的类型（前台或后台）和进程的状态（运行中、挂起或结束）。
- `utsname`
 - Linux库函数定义的一个结构体，结合函数 `uname()` 可以获取当前内核名称和其它信息。
- `sigaction`
 - Linux库函数定义的一个结构体，记录了信号处理的各种信息。其中比较重要的是：
`sa_handler` 是函数指针，指向新的信号处理函数；`sa_mask` 是处理信号时要阻塞的其他信号的集合；`sa_flags` 用来设置信号处理的其他动作，其值通过或运算来设置。`sa_flags` 被设置了`SA_RESTART`时，被信号中断的系统调用会在信号处理完成后重新启动；`sa_flags` 被设置了`SA_SIGINFO`时，信号处理函数可以有三个参数，接受系统发来的信号的各种信息。

base.h

存放一些myshell与内建命令都要用到的宏定义，没有特别的内容。

commands

包含所有实现内建命令所需要的函数，还有进程管理及信号处理函数。在此选择几个函数讲述实现原理。

- `signal_handler()`

- 这一个函数需要处理进程中收到的SIGCHLD信号和SIGTSTP信号。子进程的状态发生改变时会发出前者，子进程被Ctrl+Z中断时会发出后者。由于两个信号的处理用一个函数完成，因此接收到信号后首先需要判断接受的是什么信号。
- 对于SIGCHLD信号，在设计上我只希望函数在进程结束时处理这个信号。因此不仅要判断信号是否为SIGCHLD，还要借助`waitpid()`函数来判断子进程是否结束，如果子进程结束了，那么返回值为-1。发现进程结束时，如果进程在前台则其必定正在运行，将其状态设置为结束并转为后台；如果程序在后台且正在运行，则将其状态设置为结束。
- 对于SIGTSTP信号，需要做的比较简单。根据系统发来的信号信息，找出发出SIGTSTP的子进程，将其状态设置为挂起并转入后台即可。
- 如何建立信号与信号处理函数之间的联系等初始化工作将在[重难点](#)中说明。

- `init_jobs()`

- 该函数进行进程管理的初始化。
- 为了保证每个进程都能访问到进程表，需要先利用`shmget()`函数获取共享内存，然后利用`shmat()`函数将共享内存连接到指针上。完成后，直接将主进程myshell添加进进程表即可。

- `rtoa_path()`

- 该函数将相对路径转为绝对路径，主要用于配合`f_cd()`函数用来实现cd命令。切换目录函数`chdir()`可以直接接受相对路径为参数，但是cd命令还需要手动修改环境变量，该环境变量需要设置为绝对路径。
- 该函数主要是对`~`、`..`和`.`三个特殊符号进行处理，以及当发现输入的参数不是绝对路径时，利用环境变量将相对路径转化为绝对路径。

- `f_fg()` 和 `f_bg()`

- 这两个函数接受一个pid作为参数，将该pid指定的进程转为前台运行或后台运行，实现fg与bg命令。
- 实现fg命令时，首先在进程表中寻找目标进程，找到了将其状态转为前台运行中。实现该目标的关键是，通过`kill()`函数向进程发送SIGCONT信号使其继续运行，紧接着调用`waitpid()`函数的最后一个参数要设为0，表示阻塞当前进程等到目标进程结束。这样才可以实现前台运行的效果。
- 相对地，实现bg命令时，发送SIGCONT信号后不必调用`waitpid()`，直接让进程继续运行而不占用前台资源即可。

- `f_set()`

- 该函数实现set命令。
- 当命令没有参数时，命令需要列出所有环境变量。我们可以利用Linux系统中的`environ`指针，对所有环境变量进行遍历，输出环境变量即可。

```

extern char **environ;
for(int i=0; environ[i]; i++){
    std::cout << environ[i] << std::endl;
}

```

- 命令没有参数时，还可能是对环境变量进行赋值。通过检测命令是否带有等号，可以判断是否为该情况。如果是，则调用`setenv()`函数对环境变量进行赋值或添加。
- 命令有一个参数时，我将其设置为查看特定环境变量的值。如果没有找到参数指定的环境变量，则报错；否则直接输出值，这可以通过`getenv()`函数实现。

- `f_umask()`

- 该函数实现umask命令。
- 利用`umask()`函数可以很方便地实现该命令。该函数较为特殊，其接受一个不可缺省的输入作为新的掩码，同时返回旧的掩码。因此如果想要获取旧的掩码，就需要先设置一个新的掩码，读取到旧的掩码后再将其还原回去。

- 需要注意的是，设置新掩码时系统会自动按照八进制数处理输入值。因此，在调用之前需要将用户的输入从十进制转为八进制。如下即为先检查、进制转换再调用函数：

```

int illegal = 0, mask = 0;
for(int i=0; i<cmdlist[index+1].length(); i++){
    if(!(cmdlist[index+1][i]>='0' && cmdlist[index+1][i]<='8')){
        // 输入了非法字符
        illegal = 1;
        break;
    }
}
else{
    // 将十进制处理成八进制
    mask <=> 3;
    mask |= cmdlist[index+1][i] - '0';
}
if(illegal){
    fprintf(stderr, RED "myshell: umask: %s: Octal number out of range\n",
            cmdlist[index+1].c_str());
}
else{
    umask(mask);
}

```

- `f_cd()`

- 该函数实现cd命令。
- 利用`chdir()`函数可以实现工作目录的切换，支持直接输入相对路径，但是该函数不会设置环境变量。因此，调用该函数后，需要手动设置环境变量，使得PWD变量值与当前工作目录一致。
- 设置PWD之前需要将相对路径转为绝对路径，否则PWD变量的值会出错。

myshell

包含了主函数，实现了题目所要求的myshell的各种管理功能。myshell需要完成的功能可分为：

- 信号处理初始化
 - 该功能由`init()`完成。
 - 只需要声明一个`struct sigaction`结构体，定义好信号处理的各种设置，利用`sigaction()`函数将自定义的信号处理函数与信号链接即可。
- 进程管理
 - 在`init()`函数内需要调用进程表初始化函数`init_jobs()`。
 - 创建进程时，需要将进程的信息添加进进程表。进程的状态变更时，由信号处理函数对其信息进行更新。
- 打印提示符
 - 该功能由`init()`与`showprompt()`完成。
 - 提示符初始化需要获取用户名、机器域名以及当前的工作目录。用户名可以直接通过环境变量`LOGNAME`获取，而工作目录也因为cd命令维护了PWD环境变量而可以直接获取。机器域名的获取则需要通过`uname()`获取内核信息，得到域名后保存下来（用户名也可以通过该方式获取）。
 - 为了与shell的提示符作区分，我将myshell的提示符颜色设为了黄色+青色。

- 为了模仿shell的提示符，该功能还要检查工作目录是否包含主目录，包含的话将其替换为`~`。
- 获取输入并翻译
 - 该功能由 `interpreter()` 完成。
 - 考虑到支持执行batchfile的需求，获取输入之前需要先判断调用myshell时有没有输入参数。如果有，则打开相应的文件，执行文件内的命令；如果没有，则输出提示符，等待用户输入命令。
 - 设置一个 `flag` 变量，采用One-Hot编码，指示输入中是否有重定向、管道或后台等操作。
 - 利用 `fgets()` 或 `cin.getline()` 函数，获取一整行输入。
 - 利用 `istringstream` 对象，可以很方便地利用输入中的空白符，将输入拆分成单个的单词，保存在一个类型为向量的全局变量 `cmdlist` 中。进行一次遍历，以重定向和管道符号为间隔，记录下每个独立命令的开始下标以及长度。根据这些信息，可以开始执行命令。
- 执行命令
 - 该功能主要由 `executepipe()` 和 `execute()` 完成，前者处理有管道的输入，后者处理没有管道的输入。
 - 在编写实现内部命令的函数时，我将每个函数的输入都设置为了一个下标一个长度。一个类型为向量的全局变量保存翻译后的输入，因此只需要这两个信息函数即可获取命令的全部输入。
 - `execute()` 首先判断命令是否为内部命令，如果是则直接调用函数；否则利用 `fork()` 创建子进程，调用 `system()` 函数执行外部命令。
 - `executepipe()` 采用递归的方式。将前一个命令的标准输出重定向至管道后，将后一个命令的输入重定向至管道并递归调用该函数，即可利用递归实现多重管道操作。
- 重定向
 - 该功能由 `interpreter()` 完成。
 - 如果遍历命令时发现用户输入了重定向操作符，则直接将标准输入、标准输出或者标准错误重定向，并对 `flag` 作相应的编码。
 - 在实践中发现，一直不能利用 `dup2()` 函数重定向至文件，但是可以用该函数重定向至管道。因此，重定向至文件时，使用了 `freopen()` 函数；使用该函数时，如果要还原，标准输入、标准输出和标准错误都还原至 `/dev/tty`。
 - 每次执行完一行命令，一定要将标准输入、标准输出和标准错误还原。
- 管道
 - 进行管道操作时，需要先声明一个长度为2的整型数组，记录管道的文件描述符。
 - 在shell中，管道操作是让前一个命令的输出作为后一个命令的输入。因此，在程序中将前一个命令的进程的标准输出重定向至管道的输入端，将后一个命令的进程的标准输出重定向至管道的输出端。

重难点

进程管理

- 共享内存的释放
 - 在每一处需要直接退出myshell的异常，直接退出之前一定要先调用 `free_jobs()` 函数将共享内存释放掉。在执行exit命令时也要释放掉共享内存；否则下一次启动myshell的时候，会提示段错误，这就是因为之前申请的共享内存没有被释放掉。
- 进程的暂停与继续
 - 进程的管理是本程序值得加以改进的地方。在执行内部命令时，不必创建新的子进程；但在执行外部命令时，需要创建新的子进程。由于编程时对函数的进程特点不甚清楚，我在myshell中调用了一次 `fork()` 函数，然后才在子进程中调用 `system()` 函数执行外部命令。这么做的结果是，实际上系统多出了三个pid连续的进程：名为myshell的子进程，然后是名为sh的子

进程，然后是名为外部命令的进程。myshell子进程是因为 `fork()` 而出现，sh进程是 `system()` 用来执行命令的，而最后一个进程就是命令本身。

- 而记录子进程pid时，记录的是 `fork()` 返回的pid，这个实际上只是myshell子进程的pid。在挂起进程时，它作为另外两个进程的父进程，向他发出SIGTSTP信号则让整个命令挂起；但是在继续进程的时候，如果只向myshell子进程发出SIGCONT，则因为命令仍在挂起，该进程不会结束，如果只向命令进程发出SIGCONT，则因为父进程仍在挂起，命令进程会变成僵尸进程。
- 因此，发出SIGCONT时，需要向myshell子进程、sh和命令本身三个进程都发出SIGCONT信号。
- 现有的实现方式可以满足题干要求。后来我发现更好的方式是，不需要 `fork()`，直接获取命令本身的pid，这样可以让三个进程减少到两个。而如果我选择用 `execvp()` 而不是 `system()` 来执行外部命令，则可以再减少一个sh进程，将两个进程减少到一个。

- 进程的创建与执行

- 创建一个子进程并执行代码，有一个大致可以利用的框架如下：

```
pid_t pid = fork();
if(pid < 0){
    // 出错
}
else if(!pid){
    // 子进程
}
else{
    // 主进程
}
```

- 在主进程中，如果子进程前台执行，要在 `waitpid()` 的第三个参数声明WUNTRACED，其会使 `waitpid()` 报告已经被停止的未报告的子进程的状态，否则中断挂起无法触发。而如果希望子进程的执行不会阻塞主进程，则要在第三个参数声明WNOHANG。
- 参考shell的实现，内部命令的执行时不需要创建子进程的，外部命令的执行需要创建子进程。

管道

- 双子进程与递归

- 在关于管道的实现时，我曾设想了三种实现方式。
- 第一个是双子进程的方式。利用 `pipe()` 函数建立管道，为管道前后两个命令都建立子进程，将前一个的标准输出重定向至管道的输入端，将后一个的标准输入重定向至管道的输出端。这种方法最为简单直观，但是很难用起实现多重管道。
- 第二个是直接使用公共的文件。这个做法可谓借鉴了管道的思想，管道本身就是一个不同进程都可访问的临时文件。在 `/tmp` 目录下建立一个临时文件，前一个命令将输出输出至文件内，后一个命令从文件内读取输入。这种方法可以方便地用来实现多重管道，但是没有用到 `pipe()` 等函数，有违反题干要求的风险。
- 第三个利用递归。在处理每一个管道时，利用 `pipe()` 函数建立管道，只为前一个命令建立子进程，将子进程的标准输出重定向至管道的输入端，将主进程的标准输入重定向至管道的输出端，然后主进程递归调用函数。该方法可以实现多重管道，也用到了系统提供的 `pipe()` 函数等工具。

- 标准输入的还原

- 采用递归的方法时，由于主进程的标准输入被修改了，因此一定要在管道关闭后将主进程的标准输入重定向至 `/dev/tty`。否则主进程的标准输入还在某个被关闭的管道中，程序会发生错

误。

信号处理

- 结构体的设置
 - 如下几行代码即为信号处理结构体的设置。第三行是设置信号处理函数，将结构体的函数指针指向自定义的信号处理函数。四五行是设置信号处理的动作。第六行的意义是清空信号集，允许信号处理函数被其他信号中断。最后三行即是建立连接，意味着系统收到这三个信号时，由自定义的信号处理结构体来处理这三个信号。
 - 该部分工作在 `init()` 函数中实现。
- ```
static struct sigaction siga;
// 初始化信号处理函数
siga.sa_sigaction = signal_handler;
siga.sa_flags |= SA_RESTART;
siga.sa_flags |= SA_SIGINFO;
sigemptyset(&siga.sa_mask);
sigaction(SIGCHLD, &siga, NULL);
sigaction(SIGTSTP, &siga, NULL);
sigaction(SIGSTOP, &siga, NULL);
```

- 多个信号的处理
  - 由于上述提到的进程的问题，命令运行结束时会产生两个SIGCHLD信号。一个是命令进程发送给myshell子进程的，一个是myshell子进程发送给myshell主进程的。因此，为了避免重复，信号处理函数中还要判断当前处在哪一个进程，避免本该处理一次的信号被处理两次。
  - 而SIGCHLD信号，在处理时一定要判断子进程是否结束。否则每一次子进程状态改变都会触发该信号处理函数，造成系统错误。

## 用户手册

### 运行myshell前请先执行make命令

支持的内部命令及其用法如下：

#### 1. 变量赋值

作用：对环境变量重新赋值或者创建新的环境变量（等号前后不能有空格）

参数：无参数

示例：SCHOOL=zju

#### 2. bg

作用：对环境变量重新赋值或者创建新的环境变量（等号前后不能有空格）

参数：无参数

示例：SCHOOL=zju

#### 3. cd

作用：切换工作目录（支持..、.、~）

参数：无参数时显示当前工作目录，有一个参数时切换工作目录至参数目录下

示例：cd ~

#### 4. clr

作用：清空终端

参数：无参数

示例：clr

## 5. dir

作用：显示目录下的内容（支持..、.、~）

参数：无参数则显示当前目录下的内容，有参数则显示参数所指目录下的内容；指定参数为-l时输出详细信息，类似于ls -l

示例：dir -l ~

## 6. echo

作用：显示命令后的内容

参数：无参数则显示空，有参数则显示命令后的内容

示例：echo test

## 7. exec

作用：执行参数代表的命令替换当前进程

参数：参数即为要执行的命令，可以是内部命令也可以是外部命令

示例：exec ls -l

## 8. exit

作用：退出myshell

参数：无参数

示例：exit

## 9. fg

作用：将参数指定的进程转为前台执行

参数：一个参数，指定进程的pid

示例：fg 2466

## 10. help

作用：显示帮助

参数：无参数

示例：help

## 11. jobs

作用：显示所有的后台进程

参数：无参数

示例：jobs

## 12. pwd

作用：显示当前的工作目录

参数：无参数

示例：pwd

## 13. set

作用：显示环境变量（此命令功能与environ重合，因此没有单独写environ命令）

参数：无参数时显示所有环境变量及其值，一个参数时显示参数代表的环境变量的值

示例：set SHELL

## 14. set

作用：显示环境变量（此命令功能与environ重合，因此没有单独写environ命令）；对环境变量赋值可以用等式完成

参数：无参数时显示所有环境变量及其值，一个参数时显示参数代表的环境变量的值

示例：set SHELL

## 15. shift

作用：输入命令后从标准输入读取变量，将输入变量左移后输出

参数：无参数时默认左移1位，有参数时参数制定移动多少位

示例：shift 2

## 16. test

作用：进行字符串或者整数的比较

参数：参数代表要进行比较的表达式

示例：test zju = thu

示例：test 123 -eq 321

## 17. time

作用：显示当前的日期、时间与时区

参数：无参数

示例：time

## 18. umask

作用：显示掩码或者修改掩码

参数：无参数时显示掩码，有一个参数时修改掩码

示例：umask 7777

## 19. unset

作用：删除环境变量

参数：有一个参数指定环境变量名

示例：unset SCHOOL

---

本程序支持的其他功能如下：

### 1. 执行外部命令

作用：除去上述提到的内部命令，本程序还可以执行shell的其余不同名指令，执行时会自动为其创建子进程

示例：ls -l

### 2. 执行脚本文件

作用：通过命令行执行该程序时，可以通过参数指定程序将要执行的脚本文件；执行完所有命令后，会自动退出myshell

示例：./myshell test.sh

### 3. 重定向

作用：在myshell内输入命令时，支持进行标准输入、标准输出和标准错误重定向；重定向即将从键盘读取输入改为从文件读取输入（标准输入），将在屏幕上打印输出改为在文件中保存输出（标准输出）和在屏幕上即时打印无缓冲区的错误信息改为在文件中保存错误信息（标准错误）。其中标准输出和标准错误重定向支持以覆盖的方式 (>, 1>, 2>) 和追加的方式 (>>, 1>>, 2>>) 写入文件

示例：grep "zju" < in

示例：ls -l 1> out

示例：ls -l 2> err

### 4. 管道

作用：在myshell内输入命令时，支持进行多层管道操作，如shell一般；管道的符号为|，输入后系统会将前一个命令的输出作为后一个命令的输入来执行一整行的命令

示例：ls -l | grep "rwx" | grep "test"

### 5. 后台执行

作用：在myshell内输入命令时，支持进行后台执行，在命令的最后输入&即可；后台执行命令时，命令的执行不会阻塞主进程的运行

示例：sleep 5 &

### 6. 中断执行

作用：在myshell内外部命令执行时，支持进行中断，在命令执行时按下Ctrl+Z即可将进程挂起；此后可以通过bg或者fg命令继续执行

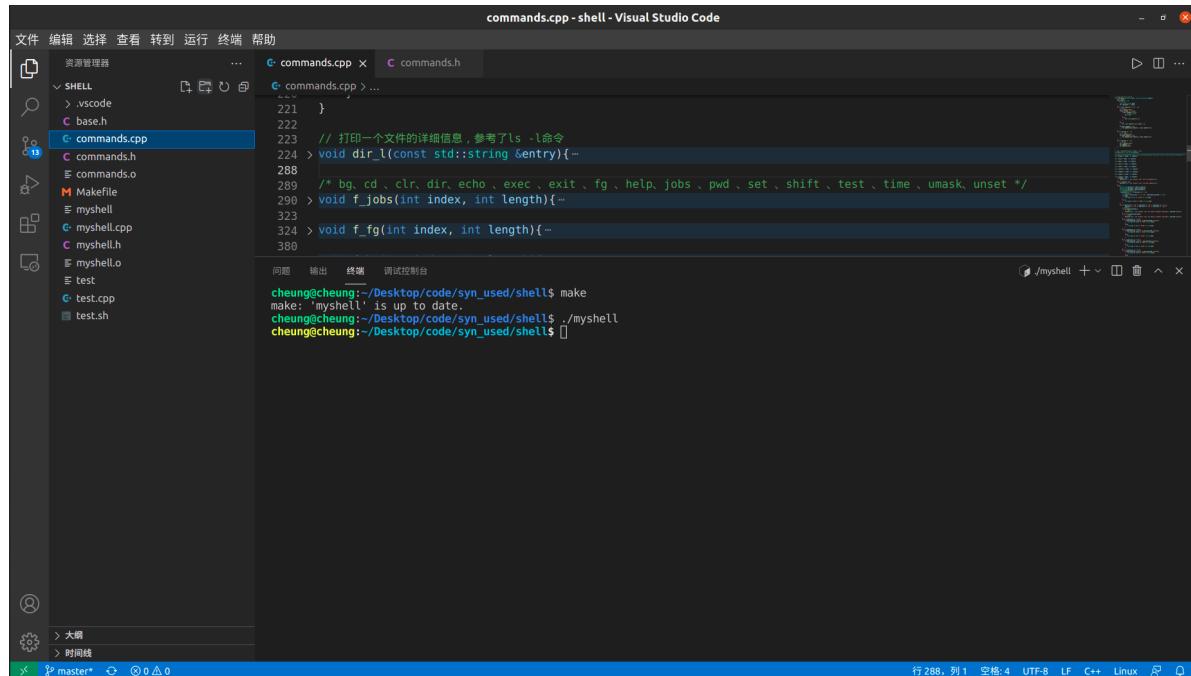
示例：Ctrl+Z

## 运行结果

## myshell

以下是myshell各个功能的运行结果：

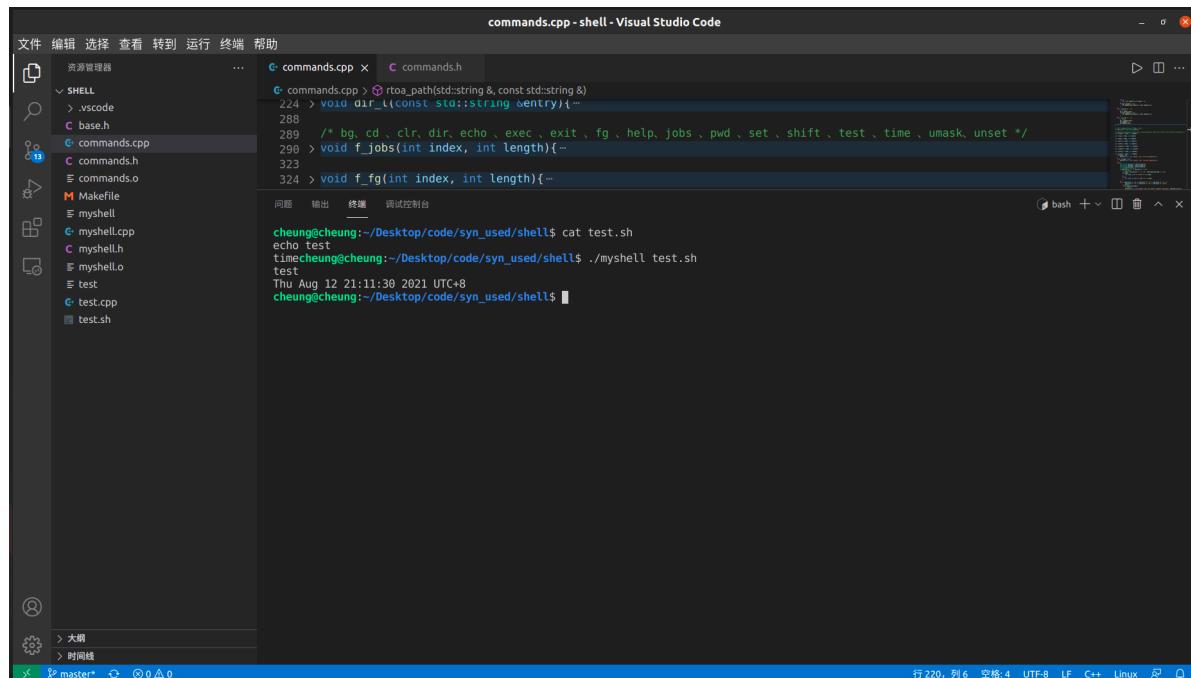
如下图，首先执行make命令，然后运行myshell。可以看到输出的提示符内容与shell一致，提示符中显示的路径也是当前的工作目录，但是颜色不一样。



The screenshot shows the Visual Studio Code interface with the title bar "commands.cpp - shell - Visual Studio Code". The left sidebar shows a file tree with files like commands.cpp, commands.h, base.h, Makefile, myshell, myshell.cpp, myshell.h, myshell.o, myshell.lo, test, test.cpp, and test.sh. The main editor area displays C++ code for a shell implementation. The bottom status bar shows "行 288, 列 1 空格:4 UTF-8 LF C++ Linux". The terminal tab is active, showing the command line output:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ make
make: 'myshell' is up to date.
cheung@cheung:~/Desktop/code/syn_used/shell$./myshell
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，执行batchfile，可以看到test.sh内有两条命令，运行myshell时传入参数，可以看到正确输出了命令执行的结果。myshell执行batchfile的功能正常。



The screenshot shows the Visual Studio Code interface with the title bar "commands.cpp - shell - Visual Studio Code". The left sidebar shows a file tree with files like commands.cpp, commands.h, base.h, Makefile, myshell, myshell.cpp, myshell.h, myshell.o, myshell.lo, test, test.cpp, and test.sh. The main editor area displays C++ code for a shell implementation. The bottom status bar shows "行 220, 列 6 空格:4 UTF-8 LF C++ Linux". The terminal tab is active, showing the command line output:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ cat test.sh
echo test
time
cheung@cheung:~/Desktop/code/syn_used/shell$./myshell test.sh
test
Thu Aug 12 21:11:30 2021 UTC+8
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，在命令的最后输入&让命令后台运行。可以看到屏幕上立刻打印出了该进程为进程表中的第几个后台进程及其pid。10秒中查看一次进程表，可以看到jobs显示该进程正在运行，该命令的执行说明后台运行的进程没有阻塞主进程。时间到后再次通过jobs命令查看进程表，可以看到显示该进程已经结束，这说明后台进程正常执行到结束。再次查看进程表，结束的后台进程被查看一次后即被删去，可以看到此处显示没有后台进程。由于我认为在后台进程结束运行后输出提示符不太美观，因此不为后台运行的进程打印结束提示符。一切结果符合预期，后台功能正常。

commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... C: commands.cpp x C: commands.h

SHELL

- > .vscode
- C: base.h
- G: commands.cpp
- C: commands.h
- E: commands.o
- M: Makefile
- E: myshell
- G: myshell.cpp
- C: myshell.h
- E: myshell.o
- E: test
- G: test.cpp
- test.sh

命令：输出 终端 调试控制台

```
cheung@cheung:~/Desktop/code/syn_used/shell$ sleep 10 &
[1] 10293
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10293 RUNNING sleep 10
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10293 DONE sleep 10
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
cheung@cheung:~/Desktop/code/syn_used/shell$
```

行 220, 列 6 空格:4 UTF-8 LF C++ Linux

如下图，测试管道功能。第一行命令中的dir命令为内建命令，grep为外部命令，可以看到正确输出了 myshell.cpp 文件的详细信息（我设计dir -l命令时参考了ls -l，尽力使得其输出一致）。为了测试多重管道操作的功能，在上述命令的基础上再加上一条内部命令shift，可以看到输出正常，与预期相符。myshell的（多重）管道功能正常。

commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... C: commands.cpp M x C: myshell.cpp C: commands.h

SHELL

- > .vscode
- C: base.h
- G: commands.cpp
- C: commands.h
- E: commands.o
- M: Makefile
- E: myshell
- G: myshell.cpp
- C: myshell.h
- E: myshell.o
- E: test
- G: test.cpp
- test.sh

命令：输出 终端 调试控制台

```
cheung@cheung:~/Desktop/code/syn_used/shell$ dir -l | grep "myshell.cpp"
-rw-rw-r-- 1 cheung cheung 10845 Thu Aug 12 16:08:20 2021 myshell.cpp
cheung@cheung:~/Desktop/code/syn_used/shell$ dir -l | grep "myshell.cpp" | shift 2
cheung cheung 10845 Thu Aug 12 16:08:20 2021 myshell.cpp
cheung@cheung:~/Desktop/code/syn_used/shell$
```

行 749, 列 39 空格:4 UTF-8 LF C++ Linux

如下图，测试重定向功能。首先查看test.sh的内容，然后调用shift命令，将标准输入重定向为test.sh，将标准输出（1缺省时默认为重定向标准输出）重定向至out文件，out文件不存在将被创建。可以看到图中out为绿色，这是因为它是新建的文件，不在git的本地仓库中。文件内第一行即为第一条shift命令的输出。紧接着，再调用一次命令，改变左移的位数，标准输入还是test.sh文件，标准输出追加至out文件。执行后可以看到out文件内第二行即为第二条命令的执行结果。

The screenshot shows a Visual Studio Code interface with a dark theme. The terminal tab is active, displaying the following command-line session:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ cat test.sh
1 2 3 4 5 6
cheung@cheung:~/Desktop/code/syn_used/shell$ shift < test.sh > out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 2 < test.sh >> out
cheung@cheung:~/Desktop/code/syn_used/shell$
```

再重复一次类似的操作，此时的标准输出重定向命令中1不缺省。可以看到out文件中，前两条命令的结果丢失，这是因为第三条命令的重定向方式为覆盖，文件中只有后两条命令的执行结果。

The screenshot shows a Visual Studio Code interface with a dark theme. The terminal tab is active, displaying the following command-line session:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ cat test.sh
1 4 5 6
2 5 6
3
cheung@cheung:~/Desktop/code/syn_used/shell$ shift < test.sh > out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 2 < test.sh >> out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 3 < test.sh 1> out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 4 < test.sh 1>> out
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，将管道与重定向结合起来进行测试，可以看到命令从文件中获取了输入并在屏幕上正确输出了结果。

out - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器

SHELL

- > .vscode
- C base.h
- C commands.cpp M
- C commands.h
- commands.o M
- myshell M
- myshell.cpp M
- myshell.h
- myshell.o M
- out U
- test M
- test.cpp M
- test.sh M

问题 输出 终端 调试控制台

```
cheung@cheung:~/Desktop/code/syn_used/shell$ cat test.sh
1 2 3 4 5 6
cheung@cheung:~/Desktop/code/syn_used/shell$ shift < test.sh > out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 2 < test.sh >> out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 3 < test.sh 1> out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 4 < test.sh 1>> out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 2 | grep "6" < test.sh
3 4 5 6
cheung@cheung:~/Desktop/code/syn_used/shell$
```

行 3, 列 1 空格:4 UTF-8 LF 纯文本 ⌂ ⌂

如下图，类似地，接下来再测试标准错误重定向。可以看到err文件中有两个标准错误的输出而屏幕上没有，符合预期（头两行“myshell”前的字符是代表颜色的字符，系程序中的正常现象）。至此，可以看出重定向功能一切正常。

err - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器

SHELL

- > .vscode
- C base.h
- C commands.cpp M
- C commands.h
- commands.o M
- myshell M
- myshell.cpp M
- myshell.h
- myshell.o M
- out U
- test M
- test.cpp M
- test.sh M
- err U

问题 输出 终端 调试控制台

```
cheung@cheung:~/Desktop/code/syn_used/shell$ cat test.sh
1 2 3 4 5 6
cheung@cheung:~/Desktop/code/syn_used/shell$ shift < test.sh > out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 2 < test.sh >> err
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 3 < test.sh 1> out
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 4 < test.sh 1>> err
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 2 | grep "6" < test.sh
3 4 5 6
cheung@cheung:~/Desktop/code/syn_used/shell$ pwd 2 > err
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs 2 >> err
cheung@cheung:~/Desktop/code/syn_used/shell$
```

行 1, 列 1 空格:4 UTF-8 LF 纯文本 ⌂ ⌂

如下图，测试中断挂起的功能。在命令执行时按下Ctrl+Z，可以看到屏幕输出^Z。并且马上显示，这是后台进程中的第一个后台进程，10381的pid及其名字。再通过jobs命令查看后台进程，可以看到该进程被挂起。结合后台功能的测试，可以知道信号处理功能正常。

```
commands.cpp - shell - Visual Studio Code
文件 编辑 选择 查看 转到 运行 终端 帮助
资源管理器 ... C commands.cpp x C commands.h
SHELL
> .vscode
C base.h
C commands.cpp
C commands.h
E commands.o
Makefile
E myshell
E myshell.cpp
C myshell.h
E myshell.o
E test
C test.cpp
test.sh

命令 问题 输出 终端 调试控制台
cheung@cheung:~/Desktop/code/syn_used/shell$ sleep 20
^Z [1] Stopped 10381 sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10381 PID status name
[1] 10381 SUSPEND sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ fg 10333
myshell: fg: 10333: No such process
cheung@cheung:~/Desktop/code/syn_used/shell$ fg 10381
sleep 20
行 220, 列 6 空格:4 UTF-8 LF C++ Linux
```

至此，myshell的功能测试完毕，一切正常。

## commands

以下是各个命令的运行结果，所有命令都对参数进行了检查，但是截图并不包括每一个命令的参数测试结果：

如下两图，测试jobs和fg命令。可以看到，程序被挂起后，jobs正确显示了此时的后台进程。再调用fg命令，如果输入的pid不存在与后台进程表，则会提示不存在该进程。利用fg将命令转入前台，可以看到在一定时间内myshell在休眠，无法进行操作。时间到后，myshell恢复运行，再次通过jobs命令，可以看到显示该进程已经结束。这说明jobs与fg命令执行正常。

```
commands.cpp - shell - Visual Studio Code
文件 编辑 选择 查看 转到 运行 终端 帮助
资源管理器 ... C commands.cpp x C commands.h
SHELL
> .vscode
C base.h
C commands.cpp
C commands.h
E commands.o
Makefile
E myshell
E myshell.cpp
C myshell.h
E myshell.o
E test
C test.cpp
test.sh

命令 问题 输出 终端 调试控制台
cheung@cheung:~/Desktop/code/syn_used/shell$ sleep 20
^Z [1] Stopped 10381 sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10381 PID status name
[1] 10381 SUSPEND sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ fg 10333
myshell: fg: 10333: No such process
cheung@cheung:~/Desktop/code/syn_used/shell$ fg 10381
sleep 20
行 220, 列 6 空格:4 UTF-8 LF C++ Linux
```

The screenshot shows the Visual Studio Code interface with the title bar "commands.cpp - shell - Visual Studio Code". The left sidebar displays a file tree with files like commands.cpp, commands.h, base.h, and test.cpp. The main editor area shows C++ code for a sleep command. The terminal tab is active, displaying the following session:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ sleep 20
^Z
[1] Stopped 10381 sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10381 PID status name
[1] 10381 SUSPEND sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ fg 10381
myshell: fg: 10381: No such process
cheung@cheung:~/Desktop/code/syn_used/shell$ fg 10381
sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10381 PID status name
[1] 10381 DONE sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$
```

At the bottom, the status bar shows "行 220, 列 6 空格:4 UTF-8 LF C++ Linux R".

如下图，类似地进行bg命令的测试。如果输入的pid不存在，则会提示进程不存在。通过bg命令让进程后台运行后，在时间内查看进程，可以看到显示该进程正在运行。而时间到后，可以看到显示进程已结束。这说明bg命令执行正常。

The screenshot shows the Visual Studio Code interface with the title bar "commands.cpp - shell - Visual Studio Code". The left sidebar displays a file tree with files like commands.cpp, commands.h, base.h, and test.cpp. The main editor area shows C++ code for a sleep command. The terminal tab is active, displaying the following session:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ sleep 20
^Z
[1] Stopped 10640 sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10640 PID status name
[1] 10640 SUSPEND sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ bg 10640
myshell: bg: 10640: No such process
cheung@cheung:~/Desktop/code/syn_used/shell$ bg 10640
[1] 10640 Stopped sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10640 PID status name
[1] 10640 RUNNING sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$ jobs
[1] 10640 PID status name
[1] 10640 DONE sleep 20
cheung@cheung:~/Desktop/code/syn_used/shell$
```

At the bottom, the status bar shows "行 220, 列 6 空格:4 UTF-8 LF C++ Linux R".

如下图，进行cd命令的测试。可以看到提示符中的工作路径能正确显示，命令也执行相对路径的切换。如果输入的路径不存在，myshell会报错。这说明cd命令执行正常。

The screenshot shows the Visual Studio Code interface with the title bar "commands.cpp - shell - Visual Studio Code". The left sidebar displays a file tree with files like commands.cpp, commands.h, base.h, and myshell.cpp. The main area shows a terminal window with the following text:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ make
make: "myshell" is up to date.
cheung@cheung:~/Desktop/code/syn_used/shell$./myshell
cheung@cheung:~/Desktop/code/syn_used/shell$ cd ..
cheung@cheung:~/Desktop/code/syn_used/shell$ cd ..
cheung@cheung:~/Desktop$ cd ..
cheung@cheung:~/Desktop$ cd ..
cheung@cheung:~/home$ cd ..
cheung@cheung:~/$ cd ..
cheung@cheung:~/$ cd ~/Desktop/code
cheung@cheung:~/Desktop/code$ cd nonexistent
myshell: cd: nonexistent: No such file or directory or permission denied
cheung@cheung:~/Desktop/code$ cd syn used/shell
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，进行clr命令的测试。输入命令执行后，可以看到历史输出被清除。说明clr命令执行正常。

The screenshot shows the Visual Studio Code interface with the title bar "commands.cpp - shell - Visual Studio Code". The left sidebar displays a file tree with files like commands.cpp, commands.h, base.h, and myshell.cpp. The main area shows a terminal window with the following text:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ make
make: "myshell" is up to date.
cheung@cheung:~/Desktop/code/syn_used/shell$./myshell
cheung@cheung:~/Desktop/code/syn_used/shell$ cd ..
cheung@cheung:~/Desktop/code/syn_used/shell$ cd ..
cheung@cheung:~/Desktop$ cd ..
cheung@cheung:~/Desktop$ cd ..
cheung@cheung:~/home$ cd ..
cheung@cheung:~/$ cd ..
cheung@cheung:~/$ cd ~/Desktop/code
cheung@cheung:~/Desktop/code$ cd nonexistent
myshell: cd: nonexistent: No such file or directory or permission denied
cheung@cheung:~/Desktop/code$ cd syn used/shell
cheung@cheung:~/Desktop/code/syn_used/shell$ clr$
```

```
commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... C commands.cpp x C commands.h
SHELL
> vscode
C base.h
commands.cpp
commands.h
Makefile
myshell
myshell.cpp
myshell.h
myshell.o
test
test.cpp
test.sh

221 }
222
223 // 打印一个文件的详细信息，参考了ls -l命令
224 > void dir_l(const std::string entry){-
288 /* bg、cd、clr、dir、echo、exec、exit、fg、help、jobs、pwd、set、shift、test、time、umask、unset */
290 > void f_jobs(int index, int length){-
323
324 > void f_fg(int index, int length){-
380

问题 输出 终端 调试控制台

cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，进行dir命令的测试。该命令的输出参考了ls命令，没有参数时，可以看到文件夹下的文件名被简单列出。输入-l参数与多个目录，可以看到多个目录下的文件的详细信息被列出。说明dir命令执行正常。

```
commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... C commands.cpp x C commands.h
SHELL
> vscode
C base.h
commands.cpp
commands.h
Makefile
myshell
myshell.cpp
myshell.h
myshell.o
test
test.cpp
test.sh

224 > void dir_l(const std::string entry){-
288 /* bg、cd、clr、dir、echo、exec、exit、fg、help、jobs、pwd、set、shift、test、time、umask、unset */
290 > void f_jobs(int index, int length){-
323
324 > void f_fg(int index, int length){-
380

问题 输出 终端 调试控制台

cheung@cheung:~/Desktop/code/syn_used/shell$ dir
commands.h test.cpp myshell.cpp Makefile myshell.h test.sh commands.cpp commands.o myshell.o base.
h myshell .vscode test
cheung@cheung:~/Desktop/code/syn_used/shell$ dir -l .. ~/Desktop/code
.:
-rw-rw-r-- 1 cheung cheung 552 Mon Jul 19 15:39:56 2021 palindrome.sh
drwxrwxr-x 2 cheung cheung 4096 Fri Aug 6 22:34:03 2021 www
-rw-rw-r-- 1 cheung cheung 201601 Thu Aug 12 20:43:07 2021 stat2.png
drwxrwxr-x 2 cheung cheung 4096 Fri Aug 6 20:37:48 2021 dirsync1.png
drwxrwxr-x 2 cheung cheung 4096 Fri Aug 6 20:47:48 2021 lab2_1_1.png
-rw-rw-r-- 1 cheung cheung 149799 Sat Jul 17 15:20:50 2021 lab2_2_1.png
-rw-rw-r-- 1 cheung cheung 252148 Sat Jul 17 15:47:44 2021 lab2_2_4.png
-rw-rw-r-- 1 cheung cheung 3488 Mon Jul 19 20:11:16 2021 dirsync.sh
-rw-rw-r-- 1 cheung cheung 166927 Sat Jul 17 15:47:16 2021 lab2_2_3.png
-rw-rw-r-- 1 cheung cheung 861 Mon Jul 19 15:39:56 2021 stat.sh
-rw-rw-r-- 1 cheung cheung 861 Thu Jul 1 08:00:01 1970 .git
-rw-rw-r-- 1 cheung cheung 228677 Thu Aug 12 20:44:04 2021 stat3.png
-rw-rw-r-- 1 cheung cheung 931 Tue Aug 3 14:50:32 2021 test.sh
drwxrwxr-x 3 cheung cheung 4096 Thu Aug 12 16:20:12 2021 shell
-rw-rw-r-- 1 cheung cheung 8698 Tue Aug 3 14:50:32 2021 poker.sh
-rw-rw-r-- 1 cheung cheung 652422 Sat Jul 17 15:28:25 2021 lab2_2_2.png
-rw-rw-r-- 1 cheung cheung 5218 Sun Jul 25 17:55:36 2021 linux.md
-rw-rw-r-- 1 cheung cheung 292736 Thu Aug 12 20:42:34 2021 stati.png

~/Desktop/code:
drwxrwxr-x 3 cheung cheung 4096 Sun Jul 11 21:39:37 2021 py
drwxrwxr-x 5 cheung cheung 4096 Thu Aug 12 20:26:03 2021 source
-rw-rw-r-- 1 cheung cheung 3488 Mon Jul 19 20:11:16 2021 dirsync.sh
-rw-rw-r-- 1 cheung cheung 861 Thu Aug 12 20:40:57 2021 stat.sh
drwxrwxr-x 3 cheung cheung 4096 Sat Aug 7 00:21:25 2021 cpp
drwxrwxr-x 7 cheung cheung 4096 Thu Aug 12 20:44:04 2021 syn_used

问题 输出 终端 调试控制台

cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，进行echo命令的测试。没有参数时，输出空白行；有参数时，将参数全部输出到标准输出。这说明echo命令执行正常。

commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... G commands.cpp C commands.h

SHELL < .vscode < G commands.cpp < C commands.h

> base.h < G commands.cpp < C commands.h

commands.h < G commands.cpp < C commands.h

Makefile < G commands.cpp < C commands.h

myshell < G commands.cpp < C commands.h

myshell.cpp < G commands.cpp < C commands.h

myshell.h < G commands.cpp < C commands.h

myshell.o < G commands.cpp < C commands.h

test < G commands.cpp < C commands.h

test.cpp < G commands.cpp < C commands.h

test.sh < G commands.cpp < C commands.h

问题 输出 终端 调试控制台

cheung@cheung:~/Desktop/code/syn\_used/shell\$ echo  
cheung@cheung:~/Desktop/code/syn\_used/shell\$ echo test zju  
test zju  
cheung@cheung:~/Desktop/code/syn\_used/shell\$

行 288, 列 1 空格:4 UTF-8 LF C++ Linux ⚙️ 0

如下图，进行exec命令的测试。两次测试，可以看到myshell均输出了正确的结果，并且在执行完exec后退出。这说明exec命令执行正常。

commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... G commands.cpp C commands.h

SHELL < .vscode < G commands.cpp < C commands.h

> base.h < G commands.cpp < C commands.h

commands.h < G commands.cpp < C commands.h

Makefile < G commands.cpp < C commands.h

myshell < G commands.cpp < C commands.h

myshell.cpp < G commands.cpp < C commands.h

myshell.h < G commands.cpp < C commands.h

myshell.o < G commands.cpp < C commands.h

test < G commands.cpp < C commands.h

test.cpp < G commands.cpp < C commands.h

test.sh < G commands.cpp < C commands.h

问题 输出 终端 调试控制台

cheung@cheung:~/Desktop/code/syn\_used/shell\$ echo  
cheung@cheung:~/Desktop/code/syn\_used/shell\$ echo test zju  
test zju  
cheung@cheung:~/Desktop/code/syn\_used/shell\$ exec echo test  
test  
Goodbye  
cheung@cheung:~/Desktop/code/syn\_used/shell\$ ./myshell  
cheung@cheung:~/Desktop/code/syn\_used/shell\$ exec dir -l  
-rw-rw-r-- 1 cheung cheung 1751 Thu Aug 12 15:56:27 2021 commands.h  
-rw-rw-r-- 1 cheung cheung 2087 Thu Aug 12 14:26:16 2021 test.cpp  
-rw-rw-r-- 1 cheung cheung 10845 Thu Aug 12 16:08:36 2021 myshell.cpp  
-rw-rw-r-- 1 cheung cheung 333 Tue Aug 3 15:24:21 2021 Makefile  
-rw-rw-r-- 1 cheung cheung 744 Thu Aug 12 15:59:28 2021 myshell.h  
-rw-rw-r-- 1 cheung cheung 14 Thu Aug 12 15:03:32 2021 test.sh  
-rw-rw-r-- 1 cheung cheung 37683 Thu Aug 12 16:18:55 2021 commands.o  
-rw-rw-r-- 1 cheung cheung 166672 Thu Aug 12 16:18:59 2021 test.o  
-rw-rw-r-- 1 cheung cheung 112792 Thu Aug 12 16:18:58 2021 myshell.o  
-rwxrwxr-x 1 cheung cheung 561 Thu Aug 12 15:51:51 2021 base.h  
-rwxrwxr-x 1 cheung cheung 117648 Thu Aug 12 16:18:59 2021 myshell  
-rwxrwxr-x 1 cheung cheung 18912 Thu Aug 12 14:53:34 2021 test  
Goodbye  
cheung@cheung:~/Desktop/code/syn\_used/shell\$

行 288, 列 1 空格:4 UTF-8 LF C++ Linux ⚙️ 0

如下图，进行exit命令的测试。在输入exit命令后，myshell输出Goodbye并退出。如果输入了参数，myshell会提示参数太多。这说明exit命令执行正常。

The screenshot shows a Visual Studio Code interface with a terminal window open. The terminal shows the user testing a shell application named 'myshell'. The user types 'exit' and receives a 'Goodbye' response. Then they type 'exit 2' and receive an error message: 'myshell: exit: Too many arguments'. The terminal window has tabs for '/myshell' and other sessions.

如下图，进行help命令的测试。执行help命令，可以看到帮助信息被输出。如果要结合more命令使用，利用管道，可以看到命令也可以正确执行。这说明help命令执行正常。

The screenshot shows a Visual Studio Code interface with a terminal window open. The user has run the 'help' command in the shell. The terminal displays a detailed help menu for the 'myshell' commands. It includes sections for 'myshell', 'myshell - developed by Cheung', and descriptions for commands like 'bg', 'cd', 'clr', 'dir', 'echo', 'exec', 'fg', 'help', 'jobs', 'pwd', 'set', 'shift', 'test', 'time', and 'umask'. Each command entry provides a brief description, usage, parameters, and examples. The terminal window has tabs for '/myshell' and other sessions.

```
commands.cpp - shell - Visual Studio Code
文件 编辑 选择 查看 转到 运行 终端 帮助
资源管理器 ... C: commands.cpp x C: commands.h
SHELL
> .vscode
C: base.h
C: commands.cpp
C: commands.h
M: Makefile
E: myshell
E: myshell.cpp
C: myshell.h
E: myshell.o
E: test
C: test.cpp
E: test.sh
... C: commands.cpp > ...
224 > void dir_l(const std::string &entry){-
288 /* bg、cd、clr、dir、echo、exec、exit、fg、help、jobs、pwd、set、shift、test、time、umask、unset */
示例： umask 7777
-----myshell-----
--developed by Cheung--
本程序尽力还原了一部分shell的功能(具体命令可能不同)，支持的内部命令及其用法如下：
1. 赋值操作符
作用：对环境变量重新赋值或者创建新的环境变量(等号前后不能有空格)
参数：无参数
示例： SCHOOL=zju

2. bg
作用：对环境变量重新赋值或者创建新的环境变量(等号前后不能有空格)
参数：无参数
示例： SCHOOL=zju

3. cd
作用：切换工作目录(支持...、~)
参数：无参数时显示当前工作目录，有一个参数时切换工作目录至参数目录下
示例： cd ~

4. clr
作用：清空终端
参数：无参数
示例： clr

5. dir
作用：显示目录下的内容(支持...、~)
参数：无参数则显示当前目录下的内容，有参数则显示参数所指目录下的内容；指定参数为-l时输出详细信息，类似于ls -l
示例： dir -l

6. echo
作用：显示命令后的内容
参数：无参数则显示空，有参数则显示命令后的内容
示例： echo test

--More-->
cheung@cheung:~/Desktop/code/syn_used/shell$ help | more
```

如下图，进行外部命令的测试。执行ls -l命令，可以看到文件的详细信息被输出，与在shell中执行该命令的结果一致。这说明外部命令执行正常。

```
commands.cpp - shell - Visual Studio Code
文件 编辑 选择 查看 转到 运行 终端 帮助
资源管理器 ... C: commands.cpp x C: commands.h
SHELL
> .vscode
C: base.h
C: commands.cpp
C: commands.h
M: Makefile
E: myshell
E: myshell.cpp
C: myshell.h
E: myshell.o
E: test
C: test.cpp
E: test.sh
... C: commands.cpp > ...
985 std::cout << "1. 执行外部命令" << std::endl;
986 std::cout << "作用： 除去上述提到的内部命令，本程序还可以执行shell的其余不同名指令，执行时会自动为其创建子进程" << std::endl;
987 std::cout << "示例： ls -l" << std::endl;
988 std::cout << std::endl;
989 std::cout << "2. 执行外部文件" << std::endl;
cheung@cheung:~/Desktop/code/syn_used/shell$ ls -l
total 488
-rw-rw-r-- 1 cheung cheung 561 8月 12 15:51 base.h
-rw-rw-r-- 1 cheung cheung 37683 8月 12 16:18 commands.cpp
-rw-rw-r-- 1 cheung cheung 1759 8月 12 15:56 commands.h
-rw-rw-r-- 1 cheung cheung 166672 8月 12 16:18 commands.o
-rw-rw-r-- 1 cheung cheung 333 8月 3 15:24 Makefile
-rwxrwxr-x 1 cheung cheung 11764 8月 12 16:18 myshell
-rw-rw-r-- 1 cheung cheung 18845 8月 12 16:08 myshell.cpp
-rw-rw-r-- 1 cheung cheung 744 8月 12 16:59 myshell.h
-rwxrwxr-x 1 cheung cheung 11225 8月 12 14:53 test
-rw-rw-r-- 1 cheung cheung 2897 8月 12 14:26 test.cpp
-rw-rw-r-- 1 cheung cheung 14 8月 12 15:03 test.sh
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，进行pwd命令的测试。可以看到pwd会对参数进行检查。在多次切换工作目录后，pwd一直都正确输出当前工作目录的绝对路径形式。这说明pwd命令执行正常。

The screenshot shows a Visual Studio Code interface. On the left is a sidebar with project files: .vscode, base.h, commands.cpp, commands.h, Makefile, myshell, myshell.cpp, myshell.h, test, test.cpp, and test.sh. The main area has two tabs: commands.cpp and commands.h. The commands.cpp tab is active, displaying C++ code for a shell. The terminal tab shows a shell session:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ pwd
/home/cheung/Desktop/code/syn_used/shell
cheung@cheung:~/Desktop/code/syn_used/shell$ cd ..
cheung@cheung:~/Desktop/code/syn_used$ pwd
/home/cheung/Desktop/code/syn_used
cheung@cheung:~/Desktop/code/syn_used$ myshell
myshell: pwd: Too many arguments
cheung@cheung:~/Desktop/code/syn_used$ cd shell
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，进行set命令的测试。可以看到无参数时，命令会打印所有环境变量及其值。有一个参数时，命令会输出参数指定的环境变量的值，环境变量不存在则报错。如果直接采用等式，可以修改环境变量或新建环境变量。可以看到通过等式，新建了一个名为SCHOOL的环境变量。而再次用等式给它赋值后，它的值变为新的值。这说明set命令执行正常。

The screenshot shows a Visual Studio Code interface. On the left is a sidebar with project files: .vscode, base.h, commands.cpp, commands.h, Makefile, myshell, myshell.cpp, myshell.h, test, test.cpp, and test.sh. The main area has two tabs: commands.cpp and commands.h. The commands.cpp tab is active, displaying C++ code for a shell. The terminal tab shows a shell session with environment variable testing:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ set
SHELL=/bin/bash
SESSION_MANAGER=local:cheung:@/tmp/.ICE-unix/1776,unix/cheung:/tmp/.ICE-unix/1776
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
TERM_PROGRAM=GNOME-1.59.0
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
APP_LAUNCHER=GNOME-DESKTOP SESSION NO_DIAGNOSTIC_CHANNEL=true
LC_ADDRESSING=zh_CN.UTF-8
GNOME_SHELL_SESSION_MODE=ubuntu
LC_NAME=zh_CN.UTF-8
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
BREAKPAD_DUMP_LOCATION=/home/cheung/.config/Code/exthost Crash Reports
XMODIFIERS=@imibus
DESKTOP_SESSION=ubuntu
LC_MONETARY=zh_CN.UTF-8
SSH_AGENCY_PID=1738
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PULSEAUDIO_SOCKET=/tmp/pulse/cheung@cheung:~/Desktop/code/syn_used/shell
XDG_SESSION_DESKTOP=ubuntu
LOGNAME=cheung
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
VS_CODE_GIT_ASKPASS_NODE=/usr/share/code/code
WINDOWPATH=2
HOME=/home/cheung
USERNAME=cheung
IM_CONFIG_PHASE=1
LC_PAPER=zh_CN.UTF-8
```

commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... C commands.cpp x C commands.h

SHELL > .vscode

C base.h

C commands.cpp

E commands.o

M Makefile

E myshell

C myshell.cpp

C myshell.h

E myshell.o

E test

C test.cpp

test.sh

commands.cpp > ...

```
224 > void dir_l(const std::string &entry){-
288 /* bg, cd, clr, dir, echo, exec, exit, fg, help, jobs, pwd, set, shift, test, time, umask, unset */
290 > void f_jobs(int index, int length){-
323
324 > void f_fq(int index, int length){-
```

问题 输出 终端 调试控制台

输出

```
USER=cheung
VSCODE_GIT_IPC_HANDLE=/run/user/1000/vscode-git-30e9c05322.sock
GNOME_TERMINAL_SERVICE=:1.208
DISPLAY=:0
SHLVL=2
LC_TELEPHONE=zh_CN.UTF-8
QT_IM_MODULE=xibus
LC_MEASUREMENT=zh_CN.UTF-8
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=93ffd442740eb0ea500609eb61150cde
XDG_RUNTIME_DIR=/run/user/1000
LC_TIME=zh_CN.UTF-8
VSCODE_GIT_ASKPASS_MAIN=/usr/share/code/resources/app/extensions/git/dist/askpass-main.js
JOURNAL_STREAM=8:56205
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/var/lib/snapd/desktop
GDMSESSION=ubuntu
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/bin:/usr/games:/usr/local/games:/snap/bin
GDMSESSION=ubuntu
ORIGINAL_XDG_CURRENT_DESKTOP=gnome
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=93ffd442740eb0ea500609eb61150cde
LC_NUMERIC=zh_CN.UTF-8
TERM_PROGRAM=vscod
= ./myshell
sheung@cheung:~/Desktop/code/myshell$ set shell
sheung@cheung:~/Desktop/code/myshell$ set shell
sheung@cheung:~/Desktop/code/syn_used/shell$ set shell
myshell: set: environment variable not found
sheung@cheung:~/Desktop/code/syn_used/shell$ SCHOOL=zju
sheung@cheung:~/Desktop/code/syn_used/shell$ set SCHOOL
zjui
sheung@cheung:~/Desktop/code/syn_used/shell$ SCHOOL=zjui
sheung@cheung:~/Desktop/code/syn_used/shell$ set SCHOOL
zjui
sheung@cheung:~/Desktop/code/syn_used/shell$
```

终端

命令行输入: ./myshell

命令行输出: sheung@cheung:~/Desktop/code/myshell\$ set shell  
sheung@cheung:~/Desktop/code/myshell\$ set shell  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ set shell  
myshell: set: environment variable not found  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ SCHOOL=zju  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ set SCHOOL  
zjui  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ SCHOOL=zjui  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ set SCHOOL  
zjui  
sheung@cheung:~/Desktop/code/syn\_used/shell\$

状态栏 行 288, 列 1 空格:4 UTF-8 LF C++ Linux ⚙ 0

如下图，进行unset命令的测试。可以看到一开始存在一个名为SCHOOL的环境变量。如果unset命令的参数过多或过少，或者参数指定的环境变量不存在，myshell都会报错。通过unset删除掉SCHOOL这个环境变量后，通过set再次查看该环境变量，提示该环境变量不存在，说明其已被删去。这说明unset命令执行正常。

commands.cpp - shell - Visual Studio Code

文件 编辑 选择 查看 转到 运行 终端 帮助

资源管理器 ... C commands.cpp x C commands.h

SHELL > .vscode

C base.h

C commands.cpp

E commands.o

M Makefile

E myshell

C myshell.cpp

C myshell.h

E myshell.o

E test

C test.cpp

test.sh

commands.cpp > ...

```
224 > void dir_l(const std::string &entry){-
288 /* bg, cd, clr, dir, echo, exec, exit, fg, help, jobs, pwd, set, shift, test, time, umask, unset */
290 > void f_jobs(int index, int length){-
323
324 > void f_fq(int index, int length){-
```

问题 输出 终端 调试控制台

输出

```
sheung@cheung:~/Desktop/code/syn_used/shell$ set SCHOOL
zjui
sheung@cheung:~/Desktop/code/syn_used/shell$ unset
myshell: unset: wrong number of arguments
sheung@cheung:~/Desktop/code/syn_used/shell$ unset SCH
myshell: unset: environment variable doesn't exist
sheung@cheung:~/Desktop/code/syn_used/shell$ unset SCHOOL
sheung@cheung:~/Desktop/code/syn_used/shell$ set SCHOOL
myshell: set: environment variable not found
sheung@cheung:~/Desktop/code/syn_used/shell$
```

终端

命令行输入: ./myshell

命令行输出: sheung@cheung:~/Desktop/code/syn\_used/shell\$ set SCHOOL  
zjui  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ unset  
myshell: unset: wrong number of arguments  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ unset SCH  
myshell: unset: environment variable doesn't exist  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ unset SCHOOL  
sheung@cheung:~/Desktop/code/syn\_used/shell\$ set SCHOOL  
myshell: set: environment variable not found  
sheung@cheung:~/Desktop/code/syn\_used/shell\$

状态栏 行 288, 列 1 空格:4 UTF-8 LF C++ Linux ⚙ 0

```

命令行输出:
GIT_ASKPASS=/usr/share/code/resources/app/extensions/git/dist/askpass.sh
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/f487b29b_e20d_4c0a_bela_17e1515d8009
INVOCATION_ID=6c118f6fd6940ce834ac27362e5721
MANAGER_ID=1546
CHROMELESS=0
CODE_URL_HANDLER=code-url-handler.desktop
LESSCLOSE=/usr/bin/lesspipe %s %
XDG_SESSION_CLASS=user
LC_IDENTIFICATION=zh_CN.UTF-8
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %
USER=cheung
VSCODE_GIT_IPC_HANDLE=/run/user/1000/vscode-git-30e9c05322.sock
GNOME_TERMINAL_SERVICE=:1.208
DISPLAY=:0
SHLVL=1
LC_TELEPHONE=zh_CN.UTF-8
QT_JM_MODULE=1805
LC_MEASUREMENT=zh_CN.UTF-8
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=93ffd442740eb0ea500609eb61150cde
XDG_RUNTIME_DIR=/run/user/1000
LC_TIME=zh_CN.UTF-8
VSCODE_GIT_ASKPASS_MAIN=/usr/share/code/resources/app/extensions/git/dist/askpass-main.js
JOURNAL_STREAM=8:50205
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/var/lib/snap/desktop
GDK_BACKEND=x11
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/local/games:/snap/bin
GDMSESSION=ubuntu
ORIGINAL_XDG_CURRENT_DESKTOP=ubuntu:GNOME
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=93ffd442740eb0ea500609eb61150cde
LC_NUMERIC=zh_CN.UTF-8
TERM_PROGRAM=vscodium
= ./myshell
sheung@sheung:~/Desktop/code/myshell$ cheung@sheung:~/Desktop/code/syn_used/shell$ 行 288, 列 1 空格:4 UTF-8 LF C++ Linux

```

如下图，进行shift命令的测试。可以看到命令从终端读取输入。无参数时，将空格分隔的输入左移一位；有参数时，会左移参数指定的位数。如果参数过多，myshell会报错。事实上myshell还会检查输入的参数是否为整数，截图没有包含测试结果。这说明shift命令执行正常。

```

命令行输出:
cheung@cheung:~/Desktop/code/syn_used/shell$ shift
a b c d
b c d
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 2
a b c d e
c d e
cheung@cheung:~/Desktop/code/syn_used/shell$ shift 3 4
myshell: shift: Too many arguments
cheung@cheung:~/Desktop/code/syn_used/shell$ 行 288, 列 1 空格:4 UTF-8 LF C++ Linux

```

如下图，进行test命令的测试。截图包含了两种字符串测试以及六种整数测试的结果。如果输入的参数数量不对或者整数测试时输入了非整数，myshell都会报错。这说明test命令执行正常。

The screenshot shows the Visual Studio Code interface with the title "commands.cpp - shell - Visual Studio Code". The terminal tab is active, displaying the following command-line session:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ test zju = zjui
FALSE
cheung@cheung:~/Desktop/code/syn_used/shell$ test zju != zjui
TRUE
cheung@cheung:~/Desktop/code/syn_used/shell$ test 1 -eq 2
FALSE
cheung@cheung:~/Desktop/code/syn_used/shell$ test 1 -ne 2
TRUE
cheung@cheung:~/Desktop/code/syn_used/shell$ test 123 -lt 321
TRUE
cheung@cheung:~/Desktop/code/syn_used/shell$ test 123 -gt 321
FALSE
cheung@cheung:~/Desktop/code/syn_used/shell$ test 123 -le 123
TRUE
cheung@cheung:~/Desktop/code/syn_used/shell$ test 123 -ge 123
TRUE
cheung@cheung:~/Desktop/code/syn_used/shell$ myshell: test: zju: Numeric argument required
myshell: test: zju: Numeric argument required
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，进行time命令的测试。可以看到日期等时间信息被输出，其中最后一个值代表了当前处于的时区。这说明time命令执行正常。

The screenshot shows the Visual Studio Code interface with the title "commands.cpp - shell - Visual Studio Code". The terminal tab is active, displaying the following command-line session:

```
cheung@cheung:~/Desktop/code/syn_used/shell$ time
Thu Aug 12 20:57:49 2021 UTC+8
cheung@cheung:~/Desktop/code/syn_used/shell$
```

如下图，进行umask命令的测试。可以看到myshell正确输出了当前的掩码。设置掩码为777后再查看，可以看到其值确实为777。如果输入的值有非数字或者非八进制或者位数过多， myshell会报错。这说明umask命令执行正常。

```
commands.cpp - shell - Visual Studio Code
文件 编辑 选择 查看 转到 运行 终端 帮助
资源管理器 ... C commands.cpp X C commands.h
SHELL
> .vscode
C base.h
C commands.cpp
commands.h
I commands.o
M Makefile
E myshell
C myshell.cpp
C myshell.h
E myshell.o
I test
C test.cpp
E test.sh

224 > void dir_l(const std::string &entry){...
288
289 /* bg, cd, clr, dir, echo, exec, exit, fg, help, jobs, pwd, set, shift, test, time, umask, unset */
290 > void f_jobs(int index, int length){...
323
324 > void f_fg(int index, int length){...
cheung@cheung:~/Desktop/code/syn_used/shell$ umask
0022
cheung@cheung:~/Desktop/code/syn_used/shell$ umask 7777
cheung@cheung:~/Desktop/code/syn_used/shell$ umask
0777
cheung@cheung:~/Desktop/code/syn_used/shell$ umask 2
cheung@cheung:~/Desktop/code/syn_used/shell$ umask
0002
cheung@cheung:~/Desktop/code/syn_used/shell$ umask 22222
myshell: umask: 22222: Octal number out of range
cheung@cheung:~/Desktop/code/syn_used/shell$ umask 9999
sh: 1: umsk: not found
cheung@cheung:~/Desktop/code/syn_used/shell$ umask 9999
myshell: umask: 9999: Octal number out of range
cheung@cheung:~/Desktop/code/syn_used/shell$
```

## 源代码

### Makefile

```
CC = g++
OBJECTS = myshell.o commands.o
SOURCES = myshell.cpp commands.cpp
TARGET = myshell
HEADERS =
$(TARGET): $(OBJECTS)
 $(CC) $(OBJECTS) -o $(TARGET) -lm
commands.o: commands.cpp base.h commands.h
 $(CC) -c commands.cpp
myshell.o: myshell.cpp myshell.h base.h commands.h
 $(CC) -c myshell.cpp
clean:
 rm $(OBJECTS) $(TARGET)
```

### bash.h

```
#ifndef BASE_H
#define BASE_H

// 颜色 特殊值 命令的标志位
#define GOON 0
#define LOGOUT 1
#define FG 0
#define BG 1
#define RUNNING 0
#define SUSPEND 1
#define DONE 2
#define MAX_JOB 128
#define IN_RD 1
#define OUT_RD 2
```

```
#define OUT_RD_APP 4
#define ERR_RD 8
#define ERR_RD_APP 16
#define IS_PIPE 32
#define IS_BG 64
#define CANCEL "\e[0m"
#define RED "\e[1;31m"
#define GREEN "\e[0;32m"
#define YELLOW "\e[1;33m"
#define BLUE "\e[1;34m"
#define PURPLE "\e[1;35m"
#define CYAN "\e[1;36m"
#define WHITE "\e[1;37m"
#define CLEAR "\e[1;1H\e[2J"
#endif
```

## commands.h

```
#ifndef COMMANDS_H
#define COMMANDS_H

#include<iostream>
#include<sstream>
#include<unistd.h>
#include<stdlib.h>
#include<time.h>
#include<cstring>
#include<vector>
#include<dirent.h>
#include<sys/stat.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<pwd.h>
#include<grp.h>
#include"base.h"

// 进程信息结构体
typedef struct job
{
 pid_t pid;
 std::string name;
 int type;
 int status;
}job;

/* backup functions */
// 信号处理函数，处理SIGCHLD与SIGTSTP
void signal_handler(int signum, siginfo_t *siginfo, void *context);
// 添加一个进程
job* addjob(pid_t pid, std::string &jobname, int type, int status);
// 删除一个进程
void removejob(pid_t pid);
// 初始化进程表
void init_jobs();
```

```
// 释放进程表
void free_jobs();
// 判断一个字符串是否为纯数字
int is_digit(const std::string &str);
// 将相对路径转化为绝对路径
void rtoa_path(std::string &dir, const std::string &input);
// 打印一个文件的详细信息，参考了ls -l命令
void dir_l(const std::string &entry);
/* built-in commands */
void f_jobs(int index, int length);
void f_fg(int index, int length);
void f_bg(int index, int length);
void f_dir(int index, int length);
void f_set(int index, int length);
void f_unset(int index, int length);
void f_umask(int index, int length);
void f_exec(int index, int length);
void f_test(int index, int length);
void f_shift(int index, int length);
void f_time(int index, int length);
void f_pwd(int index, int length);
void f_echo(int index, int length);
void f_cd(int index, int length);
void f_exit(int index, int length);
void f_clr(int index, int length);
void f_system(int index, int length);
void f_help(int index, int length);
#endif
```

## myshell.h

```
#ifndef MYSHELL_H
#define MYSHELL_H

#include<iostream>
#include<sstream>
#include<string>
#include<unistd.h>
#include<stdlib.h>
#include<algorithm>
#include<vector>
#include<pwd.h>
#include<string.h>
#include<sys/utsname.h>
#include<sys/wait.h>
#include<signal.h>
#include<stddef.h>
#include"base.h"

// 打印提示信息
void showprompt();
// 翻译输入的命令并选择执行
int interpreter(char *cmd_char);
// 初始化函数
void init();
```

```
// 执行内部命令并返回0，当命令为外部命令时返回-1
int execute_builtin(int index, int length);
// 执行命令
void execute(int index, int length);
// 递归执行带有管道的命令
void executepipe(int flag, std::vector<std::pair<int, int>>&cmdinfo, int level);
#endif
```

## commands.cpp

```
#include"base.h"
#include"commands.h"

// 声明myshell.cpp中的变量
extern job* jobs;
extern int* jobnum;
extern int shmid;
extern std::vector<std::string> cmdlist;

/* backup functions */
// 信号处理函数，处理SIGCHLD与SIGTSTP
void signal_handler(int signum, siginfo_t *siginfo, void *context){
 const int job_num = *jobnum;
 int i, bgid = 0;
 pid_t pid = siginfo->si_pid;
 pid_t mainpid = jobs[0].pid;
 if(signum == SIGCHLD && waitpid(-1, NULL, WNOHANG|WUNTRACED) && getpid() == mainpid){
 // 处理结束信号，一定要判断子进程的状态
 // 为什么判断getpid()的值：因为实际上的调用是创建了一个新的myshell进程，myshell进程又创建了一个sh进程，sh进程还要再创建一个进程来调用命令
 // 如果不判断是否为主进程收到信号，这个函数会被多次触发
 for(i=0; i<job_num; i++){
 if(jobs[i].type == BG){
 bgid++;
 }
 if(pid == jobs[i].pid){
 break;
 }
 }
 if(i<job_num){
 // 找到
 if(jobs[i].type == BG){
 if(jobs[i].status == RUNNING){
 jobs[i].status = DONE;
 // std::cout << WHITE << "\n[" << bgid+1 << "] DONE\t\t" <<
jobs[i].name << std::endl;
 }
 else if(jobs[i].status == SUSPEND){
 ;
 }
 else{
 removejob(pid);
 ;
 }
 }
 }
 }
}
```

```

 }
 else{
 if(jobs[i].status == RUNNING){
 jobs[i].status = DONE;
 jobs[i].type = BG;
 }
 }
 }

// 处理中断挂起 ^Z
else if(signum == SIGTSTP && getpid() == mainpid){
 int lastFG = 0, id, bgid = 0;
 std::string name;
 pid_t pid;
 // 寻找该进程
 for(i=0; i<job_num; i++){
 if(jobs[i].type == BG){
 bgid++;
 }
 else{
 if(jobs[i].name != "myshell"){
 id = bgid + 1;
 lastFG = i;
 name.append(jobs[i].name);
 pid = jobs[i].pid;
 break;
 }
 }
 }
}

if(lastFG>0){
 // 找到了该进程
 std::cout << WHITE << "\n[" << id << "] stopped\t\t" << pid << "\t"
<< name << std::endl;
 jobs[lastFG].type = BG;
 jobs[lastFG].status = SUSPEND;
 kill(pid, SIGSTOP);
}
}

// 添加一个进程
job* addjob(pid_t pid, std::string &jobname, int type, int status){
 int job_num = *jobnum;
 if(job_num == MAX_JOB){
 fprintf(stderr, RED "myshell: Too many jobs\n");
 free_jobs();
 exit(1);
 }
 jobs[job_num].pid = pid;
 jobs[job_num].name = jobname;
 // strcpy(jobs[job_num].name, jobname.c_str());
 jobs[job_num].type = type;
 jobs[job_num].status = status;
 *jobnum = job_num + 1;
 return &jobs[job_num];
}

```

```

// 删除一个进程
void removejob(pid_t pid){
 int i, job_num = *jobnum;
 // 在表中寻找该进程
 for(i=0; i<job_num; i++){
 if(pid == jobs[i].pid){
 break;
 }
 }

 if(i < job_num - 1){
 // 找到
 for(; i<job_num-1; i++){
 jobs[i].pid = jobs[i+1].pid;
 jobs[i].name = jobs[i+1].name;
 jobs[i].type = jobs[i+1].type;
 jobs[i].status = jobs[i+1].status;
 }
 }
 // 修改信息
 jobs[i].pid = -1;
 *jobnum = job_num - 1;
}

// 初始化进程表
void init_jobs(){
 // 创建共享内存，返回值为-1则创建失败
 if((shmID = shmget((key_t)4399, sizeof(job)*MAX_JOB+sizeof(int),
0666|IPC_CREAT)) == -1){
 fprintf(stderr, RED "myshell: Fail to create shared memory\n");
 exit(1);
 }
 // 连接共享内存
 void *mem = shmat(shmID, 0, 0);
 if(mem == (void *)-1){
 fprintf(stderr, RED "myshell: Fail to connect to shared memory\n");
 exit(1);
 }
 else{
 jobs = (job*)mem;
 }

 // 初始化
 for(int i=0; i<MAX_JOB; i++){
 jobs[i].pid = -1;
 }

 // 把当前进程myshell加进去
 std::string myshell = "myshell";
 jobnum = (int *)((char *)mem + sizeof(jobs)*MAX_JOB);
 *jobnum = 0;
 addjob(getpid(), myshell, FG, RUNNING);
}

// 释放进程表
void free_jobs(){
 // 断开连接
 if(shmdt((void *)jobs) == -1){

```

```

 fprintf(stderr, RED "myshell: Fail to detach shared memory segment\n");
 exit(1);
 }

 // 释放共享内存
 if(shmctl(shmID, IPC_RMID, 0) == -1){
 fprintf(stderr, RED "myshell: Fail to free shared memory segment\n");
 exit(1);
 }
}

// 判断一个字符串是否为纯数字
int is_digit(const std::string &str){
 int length = str.length();
 for(int i=0; i<length; i++){
 if(str[i]<'0' || str[i]>'9'){
 // 发现非数字则返回0
 return 0;
 }
 }
 return 1;
}

// 将相对路径转化为绝对路径
void rtoa_path(std::string &dir, const std::string &input){
 dir = input;
 if(dir[0] == '~'){
 // 主目录
 extern std::string home;
 dir.replace(0, 1, home);
 }
 else if(dir.substr(0, 2) == ".."){
 // 父目录
 dir = getenv("PWD");
 if(dir.rfind('/') == 0){
 if(dir.length() == 0){
 // 已经到达根目录
 dir = dir;
 }
 else {
 dir = dir.substr(0, 1);
 }
 }
 else {
 dir = dir.substr(0, dir.rfind('/'));
 }
 if(input.length() > 2){
 dir.append(input.substr(2, input.length()-2));
 }
 }
 else if(dir[0] == '.'){
 // 当前目录
 dir = getenv("PWD");
 if(input.length() > 1){
 dir.append(input.substr(1, input.length()-1));
 }
 }
 else if(dir[0] != '/'){
 // 根目录
 }
}

```

```
 dir = getenv("PWD");
 dir.append("/");
 dir.append(input);
}

}

// 打印一个文件的详细信息，参考了ls -l命令
void dir_l(const std::string &entry){
 struct stat entrystatus;
 std::string path;
 rtoa_path(path, entry);
 stat(path.c_str(), &entrystatus);
 // 判断文件类型
 std::string mask;
 switch (entrystatus.st_mode & S_IFMT){
 case S_IFREG:
 // 普通文件
 mask.append("-");
 break;
 case S_IFDIR:
 // 目录文件
 mask.append("d");
 break;
 case S_IFIFO:
 // 管道文件
 mask.append("p");
 break;
 case S_IFLNK:
 // 软连接
 mask.append("l");
 break;
 case S_IFSOCK:
 // 套接字文件
 mask.append("s");
 break;
 case S_IFBLK:
 // 块设备文件
 mask.append("b");
 break;
 case S_IFCHR:
 // 字符设备文件
 mask.append("c");
 break;
 default:
 mask.append("-");
 break;
 }
 // 判断文件权限
 mask.append((entrystatus.st_mode & S_IRUSR) ? "r" : "-");
 mask.append((entrystatus.st_mode & S_IWUSR) ? "w" : "-");
 mask.append((entrystatus.st_mode & S_IXUSR) ? "x" : "-");
 mask.append((entrystatus.st_mode & S_IRGRP) ? "r" : "-");
 mask.append((entrystatus.st_mode & S_IWGRP) ? "w" : "-");
 mask.append((entrystatus.st_mode & S_IXGRP) ? "x" : "-");
 mask.append((entrystatus.st_mode & S_IROTH) ? "r" : "-");
 mask.append((entrystatus.st_mode & S_IWOTH) ? "w" : "-");
 mask.append((entrystatus.st_mode & S_IXOTH) ? "x" : "-");
 // 硬连接
}
```

```

int nlink = entrystatus.st_nlink;
// 用户与所属组
std::string user = getpwuid(entrystatus.st_uid)->pw_name;
std::string group = getgrgid(entrystatus.st_uid)->gr_name;
// 大小
int size = (int)entrystatus.st_size;
// 修改时间
std::string mo_time = ctime(&entrystatus.st_mtime);
mo_time.replace(mo_time.find('\n'), 1, "");

std::cout << mask << " " << nlink << "\t\t" << user << "\t";
std::cout << group << "\t" << size << "\t" << mo_time << " " << entry <<
std::endl;
}

/*
bg、cd、clr、dir、echo、exec、exit、fg、help、jobs、pwd、set、shift、test
、time、umask、unset */
void f_jobs(int index, int length){
 const int job_num = *jobnum;
 if(length>1){
 fprintf(stderr, RED "myshell: jobs: Too many arguments\n");
 }
 else {
 int bg_id = 0;
 // 要删除的进程的表
 std::vector<pid_t> remove_list;
 std::cout << "\tPID" << "\tstatus" << "\tname" << std::endl;
 for (int i = 0; i < job_num; i++)
 {
 if(jobs[i].type == BG){
 std::cout << '[' << ++bg_id << "] \t" << jobs[i].pid << "\t";
 if(jobs[i].status == RUNNING){
 std::cout << "RUNNING\t";
 }
 else if(jobs[i].status == SUSPEND){
 std::cout << "SUSPEND\t";
 }
 else{
 remove_list.push_back(jobs[i].pid);
 std::cout << "DONE\t";
 }
 std::cout << jobs[i].name << std::endl;
 }
 }
 // 状态为DONE的进程，输出一次后删除
 for (int i = 0; i < remove_list.size(); i++){
 removejob(remove_list[i]);
 }
 }
}

void f_fg(int index, int length){
 const int job_num = *jobnum;
 if(length == 1){
 fprintf(stderr, RED "myshell: fg: Too few arguments\n");
 }
 else if(length > 2){
 fprintf(stderr, RED "myshell: fg: Too many arguments\n");
 }
}

```

```

 }
 else{
 if(!is_digit(cmdlist[index+1])){
 fprintf(stderr, RED "myshell: fg: %s: Numeric pid required\n",
cmdlist[index+1].c_str());
 }
 else{
 int i, ret;
 pid_t target = atoi(cmdlist[index+1].c_str());
 // 寻找该进程
 for(i=0; i<job_num; i++){
 if(jobs[i].pid == target){
 break;
 }
 }
 if(i<job_num){
 // 找到了
 if(jobs[i].type == BG){
 std::cout << WHITE << jobs[i].name << std::endl;
 if(jobs[i].status == RUNNING){
 // 正在后台运行
 jobs[i].type = FG;
 waitpid(jobs[i].pid, &ret, 0);
 }
 else if(jobs[i].status == SUSPEND){
 // 正在后台挂起
 jobs[i].type = FG;
 jobs[i].status = RUNNING;
 // 发出信号让进程继续运行
 // 实际这里存的是myshell的pid, 恢复时先继续myshell, 再继续sh, 最
后继续命令
 kill(jobs[i].pid, SIGCONT);
 kill(jobs[i].pid+1, SIGCONT);
 kill(jobs[i].pid+2, SIGCONT);
 waitpid(jobs[i].pid, &ret, 0);
 }
 else{
 // 已经完成
 fprintf(stderr, RED "myshell: fg: %d: Process already
done\n", target);
 }
 }
 else{
 fprintf(stderr, RED "myshell: fg: %d: Foreground
process\n", target);
 }
 }
 else{
 fprintf(stderr, RED "myshell: fg: %d: No such process\n",
target);
 }
 }
 }

void f_bg(int index, int length){
 const int job_num = *jobnum;
 if(length == 1){

```

```

 fprintf(stderr, RED "myshell: bg: Too few arguments\n");
 }
 else if(length > 2){
 fprintf(stderr, RED "myshell: bg: Too many arguments\n");
 }
 else{
 if(!is_digit(cmdlist[index+1])){
 fprintf(stderr, RED "myshell: bg: %s: Numeric pid required\n",
cmdlist[index+1].c_str());
 }
 else{
 int i, ret, bgid = 0;
 pid_t target = atoi(cmdlist[index+1].c_str());
 // 寻找该进程
 for(i=0; i<job_num; i++){
 if(jobs[i].type == BG){
 bgid++;
 }
 if(jobs[i].pid == target){
 break;
 }
 }
 if(i<job_num){
 // 找到了
 if(jobs[i].type == BG){
 if(jobs[i].status == RUNNING){
 // 正在后台运行
 fprintf(stderr, RED "myshell: bg: %d: Running
process\n", target);
 }
 else if(jobs[i].status == SUSPEND){
 // 正在后台挂起
 std::cout << WHITE << '[' << bgid << "]\t" <<
jobs[i].pid;
 std::cout << "\tstopped\t\t" << jobs[i].name <<
std::endl;
 jobs[i].status = RUNNING;
 // 发出信号让进程继续运行
 // 实际这里存的是myshell的pid, 恢复时先继续myshell, 再继续sh, 最
后继续命令
 kill(jobs[i].pid, SIGCONT);
 kill(jobs[i].pid+1, SIGCONT);
 kill(jobs[i].pid+2, SIGCONT);
 }
 else{
 // 已经完成
 fprintf(stderr, RED "myshell: bg: %d: Process already
done\n", target);
 }
 }
 else{
 fprintf(stderr, RED "myshell: bg: %d: Foreground
process\n", target);
 }
 }
 else{
 fprintf(stderr, RED "myshell: bg: %d: No such process\n",
target);
 }
 }
 }
}

```

```

 }
 }
}

void f_dir(int index, int length){
 DIR *dp;
 struct dirent *entry;
 std::string directory;

 if(length < 3){
 if(length == 1){
 directory = getenv("PWD");
 }
 else {
 if(cmdlist[index+1] == "-l"){
 // 参数中含有-l
 directory = getenv("PWD");
 dp = opendir(directory.c_str());
 while (entry = readdir(dp)){
 if(std::string(entry->d_name) != "." && std::string(entry-
>d_name) != ".."){
 // 输出详细信息
 dir_l(std::string(entry->d_name));
 }
 }
 return ;
 }
 else{
 // 输入的一个参数为一个路径
 directory = cmdlist[index+1];
 }
 }
 if(dp = opendir(directory.c_str())){
 // 目录存在且可以打开
 while (entry = readdir(dp)){
 if(std::string(entry->d_name) != "." && std::string(entry-
>d_name) != ".."){
 std::cout << entry->d_name << "\t";
 }
 }
 std::cout << std::endl;
 closedir(dp);
 }
 else{
 fprintf(stderr, RED "myshell: dir: %s: No such directory or
permission denied\n", cmdlist[index+1].c_str());
 }
 }
 else{
 // 多个参数
 // 检查是否输入了-l
 int lflag = 0;
 for(int i=1; i<length; i++){
 if(cmdlist[index+i] == "-l"){
 lflag = 1;
 break;
 }
 }
 }
}

```

```

 }
 for (int i=1; i<length; i++){
 if(cmdlist[index+i] != "-l"){
 std::cout << cmdlist[index+i] << ":" << std::endl;
 if(!lflag){
 // 输出详细信息
 directory = cmdlist[index+i];
 rtoa_path(directory, cmdlist[index+i]);
 if(!chdir(directory.c_str())){
 // 目录存在且可以打开
 // 需要设置PWD环境变量才可以读出对应目录下的文件信息，怀疑是权限问题
 std::string pwd(getenv("PWD"));
 setenv("PWD", directory.c_str(), 1);
 dp = opendir(directory.c_str());
 while (entry = readdir(dp)){
 if(std::string(entry->d_name)!="." &&
std::string(entry->d_name)!=".."){
 dir_l(std::string(entry->d_name));
 }
 }
 std::cout << std::endl;
 closedir(dp);
 chdir(pwd.c_str());
 setenv("PWD", pwd.c_str(), 1);
 }
 }
 else{
 fprintf(stderr, RED "myshell: dir: %s: No such file or
directory or permission denied\n", cmdlist[index+1].c_str());
 }
 }
 else{
 // 多个参数全为路径
 directory = cmdlist[index+i];
 if(dp = opendir(directory.c_str())){
 // 目录存在且可以打开
 while (entry = readdir(dp)){
 if(std::string(entry->d_name)!="." &&
std::string(entry->d_name)!=".."){
 std::cout << entry->d_name << "\t";
 }
 }
 std::cout << std::endl;
 closedir(dp);
 }
 else{
 fprintf(stderr, RED "myshell: dir: %s: No such file or
directory or permission denied\n", cmdlist[index+1].c_str());
 }
 }
 if(i<length-1){
 std::cout << std::endl;
 }
 }
}
}

```

```

void f_set(int index, int length){
 if(length == 1){
 // 无参数，打印所有环境变量或者增加环境变量
 if((std::string::size_type)cmdlist[index].find('=') ==
 std::string::npos){
 // 找不到等号，说明是打印所有环境变量
 extern char **environ;
 for(int i=0; environ[i]; i++){
 std::cout << environ[i] << std::endl;
 }
 }
 else {
 // 赋值或者创建变量
 std::string name(cmdlist[index], 0, cmdlist[index].find('='));
 std::string value(cmdlist[index], cmdlist[index].find('=')+1,
 cmdlist[index].length()-cmdlist[index].find('='));
 // 该环境变量不存在，则添加；存在则重新赋值
 setenv(name.c_str(), value.c_str(), 1);
 }
 }
 else if(length == 2){
 // 打印特定环境变量的值
 if(!getenv(cmdlist[index+1].c_str())){
 // 该环境变量不存在
 fprintf(stderr, RED "myshell: set: Environment variable not
found\n");
 }
 else {
 std::cout << getenv(cmdlist[index+1].c_str()) << std::endl;
 }
 }
 else {
 fprintf(stderr, RED "myshell: set: wrong number of arguments\n");
 }
}

void f_unset(int index, int length){
 if(length == 2){
 // 删除环境变量
 if(!getenv(cmdlist[index+1].c_str())){
 // 该环境变量不存在
 fprintf(stderr, RED "myshell: unset: Environment variable doesn't
exist\n");
 }
 else{
 // 删除
 unsetenv(cmdlist[index+1].c_str());
 }
 }
 else {
 fprintf(stderr, RED "myshell: unset: wrong number of arguments\n");
 }
}

void f_umask(int index, int length){
 if(length == 1){
 // mask = 旧掩码
 mode_t mask = umask(0);

```

```

 umask(mask);
 fprintf(stdout, WHITE "%04o\n", mask);
 }
 else if(length == 2){
 // 判断输入是否合法再改
 if(cmdlist[index+1].length() > 4){
 // 输入过长
 fprintf(stderr, RED "myshell: umask: %s: octal number out of
range\n", cmdlist[index+1].c_str());
 }
 else{
 int illegal = 0, mask = 0;
 for(int i=0; i<cmdlist[index+1].length(); i++){
 if(!(cmdlist[index+1][i]>='0' && cmdlist[index+1][i]<='8')){
 // 输入了非法字符
 illegal = 1;
 break;
 }
 else{
 // 将十进制处理成八进制
 mask <=> 3;
 mask |= cmdlist[index+1][i]-'0';
 }
 }
 if(illegal){
 fprintf(stderr, RED "myshell: umask: %s: octal number out of
range\n", cmdlist[index+1].c_str());
 }
 else{
 umask(mask);
 }
 }
 }
 else{
 fprintf(stderr, RED "myshell: umask: Too many arguments\n");
 }
}

void f_exec(int index, int length){
 extern void execute(int index, int length);
 if(length == 1){
 fprintf(stderr, RED "myshell: exec: Too few arguments\n");
 }
 else{
 // 递归调用执行命令的函数
 execute(index+1, length-1);
 f_exit(0, 1);
 }
}

void f_test(int index, int length){
 if(length < 4){
 fprintf(stderr, RED "myshell: test: Too few arguments\n");
 }
 else if(length > 4){
 fprintf(stderr, RED "myshell: test: Too many arguments\n");
 }
 else{

```

```

 std::string operation = cmdlist[index+2];
 std::string operandA = cmdlist[index+1];
 std::string operandB = cmdlist[index+3];
 // 同时判断操作符以及结果
 if(operation == "=" || operation == "!="){
 // 字符串比较
 int flag = (operation=="=" ? 1 : 0) ^ (operandA==operandB ? 1 : 0);
 if(flag){
 std::cout << RED << "FALSE" << std::endl;
 }
 else{
 std::cout << GREEN << "TRUE" << std::endl;
 }
 }
 else if(operation == "-eq" || operation == "-ne" || operation == "-lt"
|| \
 operation == "-le" || operation == "-gt" || operation == "-ge"){
 // 整数比较
 if(!is_digit(operandA)){
 // 非整数
 fprintf(stderr, RED "myshell: test: %s: Numeric argument
required\n", operandA.c_str());
 }
 else if(!is_digit(operandB)){
 // 非整数
 fprintf(stderr, RED "myshell: test: %s: Numeric argument
required\n", operandB.c_str());
 }
 else if(operation == "-eq"){
 if(atoi(operandA.c_str()) == atoi(operandB.c_str())){
 std::cout << GREEN << "TRUE" << std::endl;
 }
 else{
 std::cout << RED << "FALSE" << std::endl;
 }
 }
 else if(operation == "-ne"){
 if(atoi(operandA.c_str()) != atoi(operandB.c_str())){
 std::cout << GREEN << "TRUE" << std::endl;
 }
 else{
 std::cout << RED << "FALSE" << std::endl;
 }
 }
 else if(operation == "-lt"){
 if(atoi(operandA.c_str()) < atoi(operandB.c_str())){
 std::cout << GREEN << "TRUE" << std::endl;
 }
 else{
 std::cout << RED << "FALSE" << std::endl;
 }
 }
 else if(operation == "-le"){
 if(atoi(operandA.c_str()) <= atoi(operandB.c_str())){
 std::cout << GREEN << "TRUE" << std::endl;
 }
 else{
 std::cout << RED << "FALSE" << std::endl;
 }
 }
 }
 }
}

```

```

 }
 }
 else if(operation == "-gt"){
 if(atoi(operandA.c_str()) > atoi(operandB.c_str())){
 std::cout << GREEN << "TRUE" << std::endl;
 }
 else{
 std::cout << RED << "FALSE" << std::endl;
 }
 }
 else {
 if(atoi(operandA.c_str()) >= atoi(operandB.c_str())){
 std::cout << GREEN << "TRUE" << std::endl;
 }
 else{
 std::cout << RED << "FALSE" << std::endl;
 }
 }
}
else{
 fprintf(stderr, RED "myshell: test: %s: unknown operator\n",
operation.c_str());
}
}

void f_shift(int index, int length){
if(length <= 2){
 if(length == 1 || is_digit(cmdlist[index+1])){
 // 默认一位
 int shift;
 if(length == 1){
 shift = 1;
 }
 else{
 shift = atoi(cmdlist[index+1].c_str());
 }
 char inputbuf[128] = {0};
 std::cin.getline(inputbuf, 123);
 // 处理输入
 std::istringstream inputbuffer(inputbuf);
 std::string tmp;
 std::vector<std::string> input;
 while(inputbuffer >> tmp){
 input.push_back(tmp);
 }
 // 输出
 if(shift > input.size()){
 std::cout << std::endl;
 }
 else{
 for(int i=shift; i<input.size(); i++){
 std::cout << input[i] << " ";
 }
 std::cout << std::endl;
 }
 }
}
else{

```

```

 fprintf(stderr, RED "myshell: shift: Numeric argument required\n");
 }
}
else{
 fprintf(stderr, RED "myshell: shift: Too many arguments\n");
}
}

void f_time(int index, int length){
if(length>1){
 fprintf(stderr, RED "myshell: time: Too many arguments\n");
}
else {
 time_t current, tmp;
 const char *week[7] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
 // 得到秒数
 time(¤t);
 // 将秒数转化为可读时间
 std::string current_time = asctime(localtime(¤t));
 current_time.replace(current_time.find("\n"), 1, " ");
 // 得到格林威治时间
 tmp = mktime(gmtime(¤t));
 // 最后一个输出的计算在计算时区
 std::cout << current_time << "UTC" << std::showpos << (current-tmp)/3600
<< std::endl;
 // 取消输出正负号
 std::cout.unsetf(std::ios::showpos);
}
}
}

void f_pwd(int index, int length){
if(length>1){
 fprintf(stderr, RED "myshell: pwd: Too many arguments\n");
}
else {
 // 获得环境变量的值并打印
 std::string path = getenv("PWD");
 std::cout << path << std::endl;
}
}
}

void f_echo(int index, int length){
for(int i=1; i<length; i++){
 std::cout << cmdlist[index+i] << " ";
}
std::cout << std::endl;
}

void f_cd(int index, int length){
if(length == 1){
 // 无参数, 显示当前目录
 std::string path = getenv("PWD");
 std::cout << path << std::endl;
}
else if(length > 2){
 // 参数过多
 fprintf(stderr, RED "myshell: cd: Too many arguments\n");
}
}

```

```

else {
 std::string dir;
 // 相对路径处理
 rtoa_path(dir, cmdlist[index+1]);
 // 切换目录
 if(!chdir(dir.c_str())){
 // 切换目录成功
 setenv("PWD", dir.c_str(), 1);
 }
 else{
 // 目录不存在
 fprintf(stderr, RED "myshell: cd: %s: No such file or directory or
permission denied\n", cmdlist[index+1].c_str());
 }
}

void f_exit(int index, int length){
 if(length>1){
 fprintf(stderr, RED "myshell: exit: Too many arguments\n");
 }
 else {
 free_jobs();
 std::cout << "Goodbye" << std::endl;
 exit(0);
 }
}

void f_clr(int index, int length){
 if(length>1){
 fprintf(stderr, RED "myshell: clr: Too many arguments\n");
 }
 else {
 std::cout << CLEAR << std::endl;
 }
}

void f_system(int index, int length){
 // 调用system()来执行外部命令
 std::string cmd = cmdlist[index];
 for(int i=1; i<length; i++){
 cmd.append(" ");
 cmd.append(cmdlist[index+i]);
 }
 system(cmd.c_str());
}

void f_help(int index, int length){
 if(length>1){
 fprintf(stderr, RED "myshell: help: Too many arguments\n");
 }
 else {
 std::cout << WHITE << std::endl;
 std::cout << "-----myshell-----" <<
std::endl;
 std::cout << "-----developed by " << CYAN << "Cheung" <<
WHITE << "-----" << std::endl;
 }
}

```

```
 std::cout << "本程序尽力还原了一部分shell的功能（具体命令可能不同），支持的内部命令
及其用法如下：" << std::endl;

 std::cout << "1. 变量赋值" << std::endl;
 std::cout << "作用： 对环境变量重新赋值或者创建新的环境变量（等号前后不能有空格）"
<< std::endl;
 std::cout << "参数： 无参数" << std::endl;
 std::cout << "示例： SCHOOL=zju" << std::endl;
 std::cout << std::endl;

 std::cout << "2. bg" << std::endl;
 std::cout << "作用： 对环境变量重新赋值或者创建新的环境变量（等号前后不能有空格）"
<< std::endl;
 std::cout << "参数： 无参数" << std::endl;
 std::cout << "示例： SCHOOL=zju" << std::endl;
 std::cout << std::endl;

 std::cout << "3. cd" << std::endl;
 std::cout << "作用： 切换工作目录（支持...、..、~）" << std::endl;
 std::cout << "参数： 无参数时显示当前工作目录，有一个参数时切换工作目录至参数目录下"
<< std::endl;
 std::cout << "示例： cd ~" << std::endl;
 std::cout << std::endl;

 std::cout << "4. clr" << std::endl;
 std::cout << "作用： 清空终端" << std::endl;
 std::cout << "参数： 无参数" << std::endl;
 std::cout << "示例： clr" << std::endl;
 std::cout << std::endl;

 std::cout << "5. dir" << std::endl;
 std::cout << "作用： 显示目录下的内容（支持...、..、~）" << std::endl;
 std::cout << "参数： 无参数则显示当前目录下的内容，有参数则显示参数所指目录下的内
容；指定参数为-l时输出详细信息，类似于ls -l" << std::endl;
 std::cout << "示例： dir -l ~" << std::endl;
 std::cout << std::endl;

 std::cout << "6. echo" << std::endl;
 std::cout << "作用： 显示命令后的内容" << std::endl;
 std::cout << "参数： 无参数则显示空，有参数则显示命令后的内容" << std::endl;
 std::cout << "示例： echo test" << std::endl;
 std::cout << std::endl;

 std::cout << "7. exec" << std::endl;
 std::cout << "作用： 执行参数代表的命令替换当前进程" << std::endl;
 std::cout << "参数： 参数即为要执行的命令，可以是内部命令也可以是外部命令" <<
std::endl;
 std::cout << "示例： exec ls -l" << std::endl;
 std::cout << std::endl;

 std::cout << "8. exit" << std::endl;
 std::cout << "作用： 退出myshell" << std::endl;
 std::cout << "参数： 无参数" << std::endl;
 std::cout << "示例： exit" << std::endl;
 std::cout << std::endl;

 std::cout << "9. fg" << std::endl;
 std::cout << "作用： 将参数指定的进程转为前台执行" << std::endl;
```

```
 std::cout << "参数: 一个参数, 指定进程的pid" << std::endl;
 std::cout << "示例: fg 2466" << std::endl;
 std::cout << std::endl;

 std::cout << "10. help" << std::endl;
 std::cout << "作用: 显示帮助" << std::endl;
 std::cout << "参数: 无参数" << std::endl;
 std::cout << "示例: help" << std::endl;
 std::cout << std::endl;

 std::cout << "11. jobs" << std::endl;
 std::cout << "作用: 显示所有的后台进程" << std::endl;
 std::cout << "参数: 无参数" << std::endl;
 std::cout << "示例: jobs" << std::endl;
 std::cout << std::endl;

 std::cout << "12. pwd" << std::endl;
 std::cout << "作用: 显示当前的工作目录" << std::endl;
 std::cout << "参数: 无参数" << std::endl;
 std::cout << "示例: pwd" << std::endl;
 std::cout << std::endl;

 std::cout << "13. set" << std::endl;
 std::cout << "作用: 显示环境变量 (此命令功能与environ重合, 因此没有单独写environ命令)" << std::endl;
 std::cout << "参数: 无参数时显示所有环境变量及其值, 一个参数时显示参数代表的环境变量的值" << std::endl;
 std::cout << "示例: set SHELL" << std::endl;
 std::cout << std::endl;

 std::cout << "13. set" << std::endl;
 std::cout << "作用: 显示环境变量 (此命令功能与environ重合, 因此没有单独写environ命令); 对环境变量赋值可以用等式完成" << std::endl;
 std::cout << "参数: 无参数时显示所有环境变量及其值, 一个参数时显示参数代表的环境变量的值" << std::endl;
 std::cout << "示例: set SHELL" << std::endl;
 std::cout << std::endl;

 std::cout << "14. shift" << std::endl;
 std::cout << "作用: 输入命令后从标准输入读取变量, 将输入变量左移后输出" << std::endl;
 std::cout << "参数: 无参数时默认左移1位, 有参数时参数制定移动多少位" << std::endl;
 std::cout << "示例: shift 2" << std::endl;
 std::cout << std::endl;

 std::cout << "15. test" << std::endl;
 std::cout << "作用: 进行字符串或者整数的比较" << std::endl;
 std::cout << "参数: 参数代表要进行比较的表达式" << std::endl;
 std::cout << "示例: test zju = thu" << std::endl;
 std::cout << "示例: test 123 -eq 321" << std::endl;
 std::cout << std::endl;

 std::cout << "16. time" << std::endl;
 std::cout << "作用: 显示当前的日期、时间与时区" << std::endl;
 std::cout << "参数: 无参数" << std::endl;
 std::cout << "示例: time" << std::endl;
 std::cout << std::endl;
```

```
 std::cout << "17. umask" << std::endl;
 std::cout << "作用: 显示掩码或者修改掩码" << std::endl;
 std::cout << "参数: 无参数时显示掩码, 有一个参数时修改掩码" << std::endl;
 std::cout << "示例: umask 7777" << std::endl;
 std::cout << std::endl;

 std::cout << "18. unset" << std::endl;
 std::cout << "作用: 删除环境变量" << std::endl;
 std::cout << "参数: 有一个参数指定环境变量名" << std::endl;
 std::cout << "示例: unset SCHOOL" << std::endl;

 std::cout << "-----" << std::endl;
 std::cout << "本程序支持的其他功能如下:" << std::endl;
 std::cout << "1. 执行外部命令" << std::endl;
 std::cout << "作用: 除去上述提到的内部命令, 本程序还可以执行shell的其余不同名指令, 执行时会自动为其创建子进程" << std::endl;
 std::cout << "示例: ls -l" << std::endl;
 std::cout << std::endl;

 std::cout << "2. 执行脚本文件" << std::endl;
 std::cout << "作用: 通过命令行执行该程序时, 可以通过参数指定程序将要执行的脚本文件" << std::endl;
 std::cout << "示例: ./myshell test.sh" << std::endl;
 std::cout << std::endl;

 std::cout << "3. 重定向" << std::endl;
 std::cout << "作用: 在myshell内输入命令时, 支持进行标准输入、标准输出和标准错误重定向" << std::endl;
 std::cout << "示例: grep \"zju\" < in" << std::endl;
 std::cout << "示例: ls -l 1> out" << std::endl;
 std::cout << "示例: ls -l 2> err" << std::endl;
 std::cout << std::endl;

 std::cout << "4. 管道" << std::endl;
 std::cout << "作用: 在myshell内输入命令时, 支持进行多层管道操作, 如shell一般" << std::endl;
 std::cout << "示例: ls -l | grep \"rwx\" | grep \"test\"" << std::endl;
 std::cout << std::endl;

 std::cout << "5. 后台执行" << std::endl;
 std::cout << "作用: 在myshell内输入命令时, 支持进行后台执行, 在命令的最后输入&即可" << std::endl;
 std::cout << "示例: sleep 5 &" << std::endl;
 std::cout << std::endl;

 std::cout << "6. 中断执行" << std::endl;
 std::cout << "作用: 在myshell内外部命令执行时, 支持进行中断, 在命令执行时按下Ctrl+Z即可; 此后可以通过bg或者fg命令继续执行" << std::endl;
 std::cout << "示例: Ctrl+Z" << std::endl;
 std::cout << CANCEL << std::endl;
}

}
```

```
#include"myshell.h"
#include"commands.h"

// 用户名 域名 主目录路径
std::string username, hostname, home;
// 翻译好的命令
std::vector<std::string> cmdlist;
// 进程表的地址、数量以及共享内存ID
job* jobs;
int* jobnum;
int shmid;
// 输入命令的标志位
int flag = 0;
// 信号处理函数结构体
static struct sigaction siga;

// 打印提示信息
void showprompt(){
 std::string path = getenv("PWD");
 if(path.find(home) == 0){
 path = path.replace(path.find(home), home.length(), "~");
 }
 std::cout << YELLOW << username << "@" << hostname << WHITE << ':' << CYAN
 << path << WHITE << "$ " << CANCEL;
 return ;
}

// 翻译输入的命令并选择执行
int interpreter(char *cmd_char){
 std::istringstream cmd(cmd_char);
 std::string tmp;

 // 忽略掉多余的空格
 while(cmd >> tmp){
 cmdlist.push_back(tmp);
 }

 // interpret
 int cmdlist_length = cmdlist.size(), cmdindex = 0, cmdlength = 0;
 std::vector<std::pair<int, int>> cmdinfo;
 for(int i=0; i<cmdlist_length; i++){
 if(cmdlist[i] == "<"){
 // 输入重定向
 flag |= IN_RD;
 i++;
 if(!freopen(cmdlist[i].c_str(), "r", stdin)){
 fprintf(stderr, RED "myshell: stdin: No such file\n");
 flag = 0;
 cmdlist.clear();
 return -1;
 }
 }
 if(cmdlength){
 // 保存单个命令的起始下标与命令长度
 cmdinfo.push_back(std::make_pair(cmdindex, cmdlength));
 cmdindex = i+1;
 cmdlength = 0;
 }
 }
}
```

```
 }
 else if(cmdlist[i] == "1>" || cmdlist[i] == ">"){
 // 输出重定向
 flag |= OUT_RD;
 i++;
 freopen(cmdlist[i].c_str(), "w+", stdout);
 if(cmdlength){
 // 保存单个命令的起始下标与命令长度
 cmdinfo.push_back(std::make_pair(cmdindex, cmdlength));
 cmdindex = i+1;
 cmdlength = 0;
 }
 }
 else if(cmdlist[i] == "1>>" || cmdlist[i] == ">>"){
 // 输出追加重定向
 flag |= OUT_RD_APP;
 i++;
 freopen(cmdlist[i].c_str(), "a+", stdout);
 if(cmdlength){
 // 保存单个命令的起始下标与命令长度
 cmdinfo.push_back(std::make_pair(cmdindex, cmdlength));
 cmdindex = i+1;
 cmdlength = 0;
 }
 }
 else if(cmdlist[i] == "2>"){
 // 错误重定向
 flag |= ERR_RD;
 i++;
 freopen(cmdlist[i].c_str(), "w+", stderr);
 if(cmdlength){
 // 保存单个命令的起始下标与命令长度
 cmdinfo.push_back(std::make_pair(cmdindex, cmdlength));
 cmdindex = i+1;
 cmdlength = 0;
 }
 }
 else if(cmdlist[i] == "2>>"){
 // 错误追加重定向
 flag |= ERR_RD_APP;
 i++;
 freopen(cmdlist[i].c_str(), "a+", stderr);
 if(cmdlength){
 // 保存单个命令的起始下标与命令长度
 cmdinfo.push_back(std::make_pair(cmdindex, cmdlength));
 cmdindex = i+1;
 cmdlength = 0;
 }
 }
 else if(cmdlist[i] == "|"){
 // 管道操作
 flag |= IS_PIPE;
 if(cmdlength){
 cmdinfo.push_back(std::make_pair(cmdindex, cmdlength));
 cmdindex = i+1;
 cmdlength = 0;
 }
 }
 }
```

```

 else if(i == cmdlist_length-1 && cmdlist[i] == "&"){
 // 后台执行
 flag |= IS_BG;
 }
 else{
 cmdlength++;
 }
 // 翻译到最后，要把未记录的单个命令的信息记录下来
 if(i == cmdlist_length - 1 && cmdlength){
 cmdinfo.push_back(std::make_pair(cmdindex, cmdlength));
 }
 }

 // 执行
 if(flag & IS_PIPE){
 // 有管道
 executepipe(flag, cmdinfo, 0);
 }
 else{
 // 无管道
 execute(cmdinfo[0].first, cmdinfo[0].second);
 }

 // 还原
 if(flag & IN_RD){
 fclose(stdin);
 std::cin.clear();
 std::cin.sync();
 freopen("/dev/tty", "r", stdin);
 }
 if(flag & OUT_RD || flag & OUT_RD_APP){
 fclose(stdout);
 freopen("/dev/tty", "w", stdout);
 }
 if(flag & ERR_RD || flag & ERR_RD_APP){
 fclose(stderr);
 freopen("/dev/tty", "w", stderr);
 }
 flag = 0;
 cmdlist.clear();
 return 0;
}

// 执行内部命令并返回0，当命令为外部命令时返回-1
int execute_builtin(int index, int length){
 if(cmdlist[index] == "bg"){
 f_bg(index, length);
 }
 else if(cmdlist[index] == "cd"){
 f_cd(index, length);
 }
 else if(cmdlist[index] == "clr"){
 f_clr(index, length);
 }
 else if(cmdlist[index] == "dir"){
 f_dir(index, length);
 }
 else if(cmdlist[index] == "echo"){
 f_echo(index, length);
 }
}

```

```

 else if(cmdlist[index] == "exec"){
 f_exec(index, length);
 }
 else if(cmdlist[index] == "exit"){
 f_exit(index, length);
 }
 else if(cmdlist[index] == "fg"){
 f_fg(index, length);
 }
 else if(cmdlist[index] == "help"){
 f_help(index, length);
 }
 else if(cmdlist[index] == "jobs"){
 f_jobs(index, length);
 }
 else if(cmdlist[index] == "pwd"){
 f_pwd(index, length);
 }
 else if(cmdlist[index] == "set" ||
 (std::string::size_type)cmdlist[index].find('=') != std::string::npos){
 f_set(index, length);
 }
 else if(cmdlist[index] == "shift"){
 f_shift(index, length);
 }
 else if(cmdlist[index] == "test"){
 f_test(index, length);
 }
 else if(cmdlist[index] == "time"){
 f_time(index, length);
 }
 else if(cmdlist[index] == "umask"){
 f_umask(index, length);
 }
 else if(cmdlist[index] == "unset"){
 f_unset(index, length);
 }
 else{
 return -1;
 }
 return 0;
 }

 // 执行命令
 void execute(int index, int length){
 // 无管道
 if(execute_builtin(index, length)){
 // 处理外部命令
 pid_t pid = fork();
 if(pid < 0){
 // 出错
 fprintf(stderr, RED "myshell: fail to create a child process\n");
 return ;
 }
 else if(!pid){
 // 子进程
 // 添加环境变量
 setenv("PARENT", "/home/cheung/Desktop/code/myshell", 1);
 }
 }
 }
}

```

```

 // 执行外部命令
 f_system(index, length);
 exit(0);
 }
 else{
 // 主进程
 std::string jobname = cmdlist[index];
 for(int i=1; i<length; i++){
 jobname.append(" ");
 jobname.append(cmdlist[index+i]);
 }
 if(flag & IS_BG){
 // 后台进程
 addjob(pid, jobname, BG, RUNNING);
 // kill(pid, SIGSTOP);
 // 不等待
 waitpid(pid, NULL, WNOHANG);
 const int job_num = *jobnum;
 int bgid = 0;
 for(int i=0; i<job_num; i++){
 if(jobs[i].type == BG){
 bgid++;
 }
 }
 std::cout << WHITE << '[' << bgid << "] " << pid << CANCEL <<
std::endl;
 }
 else{
 addjob(pid, jobname, FG, RUNNING);
 // 等待
 waitpid(pid, NULL, WUNTRACED);
 }
 }
}

// 递归执行带有管道的命令
void executepipe(int flag, std::vector<std::pair<int, int>>&cmdinfo, int level){
 // 最后一个单个命令，不再需要创建管道
 if(level == cmdinfo.size()-1){
 if(execute_builtin(cmdinfo[level].first, cmdinfo[level].second)){
 f_system(cmdinfo[level].first, cmdinfo[level].second);
 }
 return ;
 }
 int pipeFD[2];

 // 创建管道
 if(pipe(pipeFD) < 0){
 fprintf(stderr, RED "myshell: fail to create pipes\n");
 return ;
 }

 pid_t pid = fork();
 if(pid < 0){
 // 出错
 fprintf(stderr, RED "myshell: fail to create a child process\n");
 return ;
 }
}

```

```
}

else if(!pid){
 // 子进程，第一个子进程向管道输出数据，重定向标准输出
 close(pipeFD[0]);
 dup2(pipeFD[1], 1);
 close(pipeFD[1]);
 if(execute_builtin(cmdinfo[level].first, cmdinfo[level].second)){
 f_system(cmdinfo[level].first, cmdinfo[level].second);
 }
 exit(0);
}

else{
 // 主进程
 int status;
 waitpid(pid, &status, 0);
 // 从管道读取数据，重定向标准输入
 close(pipeFD[1]);
 dup2(pipeFD[0], 0);
 close(pipeFD[0]);
 // 递归执行
 executepipe(flag, cmdinfo, level+1);
 // 恢复标准输入
 std::cin.clear();
 std::cin.sync();
 freopen("/dev/tty", "r", stdin);
}

return ;
}

// 初始化函数
void init(){
 // 初始化部分环境变量
 username = getenv("LOGNAME");
 home = getenv("HOME");
 struct utsname tmp;
 uname(&tmp);
 hostname = tmp.nodename;
 setenv("shell", "/home/cheung/Desktop/code/myshell", 1);
 // 初始化进程表
 init_jobs();
 // 初始化信号处理函数
 siga.sa_sigaction = signal_handler;
 siga.sa_flags |= SA_RESTART;
 siga.sa_flags |= SA_SIGINFO;
 sigemptyset(&siga.sa_mask);
 sigaction(SIGCHLD, &siga, NULL);
 sigaction(SIGTSTP, &siga, NULL);
 sigaction(SIGSTOP, &siga, NULL);
}

int main(int argc, char **argv){
 init();
 while(true){
 if(argc == 1){
 char cmd[64];
 showprompt();
 // std::cin.getline(cmd, 64);
 }
 }
}
```

```
if(fgets(cmd, 63, stdin))
{
 interpreter(cmd);
}
std::cin.clear();
std::cin.sync();
}
else{
 // 执行外部文件
FILE *fp;
char cmd[64];
// 打开文件
if(!(fp = fopen(argv[1], "r"))){
 fprintf(stderr, RED "myshell: %s: Fail to open the file\n",
 argv[1]);
 free_jobs();
 exit(1);
}
while (fgets(cmd, 63, fp))
{
 interpreter(cmd);
}
fclose(fp);
free_jobs();
exit(0);
}

return 0;
}
```