

# 浙江大学实验报告

课程名称: 课程综合实践2 实验类型: 验证型

实验项目名称: 实验2 Shell命令

学生姓名: xxx 专业: 计算机科学与技术 学号: xxxxxxxxxx

电子邮件地址: xxxxxxxxxx 手机(可选): 嘻嘻嘻嘻嘻嘻嘻嘻

实验日期: 2022年7月3日

## 一、实验环境

计算机系统: windows 10

虚拟机版本: VMware Workstation Pro

虚拟机系统: Ubuntu 22.04 LTS

虚拟机内存配置: 6G+40G

## 二、实验内容和结果及分析

1. 在操作系统课程实验中, 要用make工具编译内核, 要掌握make和makefile。makfile文件中的每一行是描述文件间依赖关系的make规则。本实验是关于makefile内容的, 您不需要在计算机上进行编程运行, 只要书面回答下面这些问题。

对于下面的makefile:

```
CC = gcc
OPTIONS = -O3 -o
OBJECTS = main.o stack.o misc.o
SOURCES = main.c stack.c misc.c
HEADERS = main.h stack.h misc.h
polish: main.c $(OBJECTS)
        $(CC) $(OPTIONS) power $(OBJECTS) -lm
main.o: main.c main.h misc.h
stack.o: stack.c stack.h misc.h
misc.o: misc.c misc.h
```

回答下列问题

### 1. 所有宏定义的名字

- 在Makefile中, 宏定义的名字即为变量名, 可见在上述的makefile代码中, 我们有如下的宏定义: CC、OPTIONS、OBJECTS、SOURCES、HEADERS

### 2. 所有目标文件的名字

- 目标文件: power、main.o、stack.o、misc.o是这一个文件中的目标文件名称

### 3. 每个目标的依赖文件

- power依赖: main.c

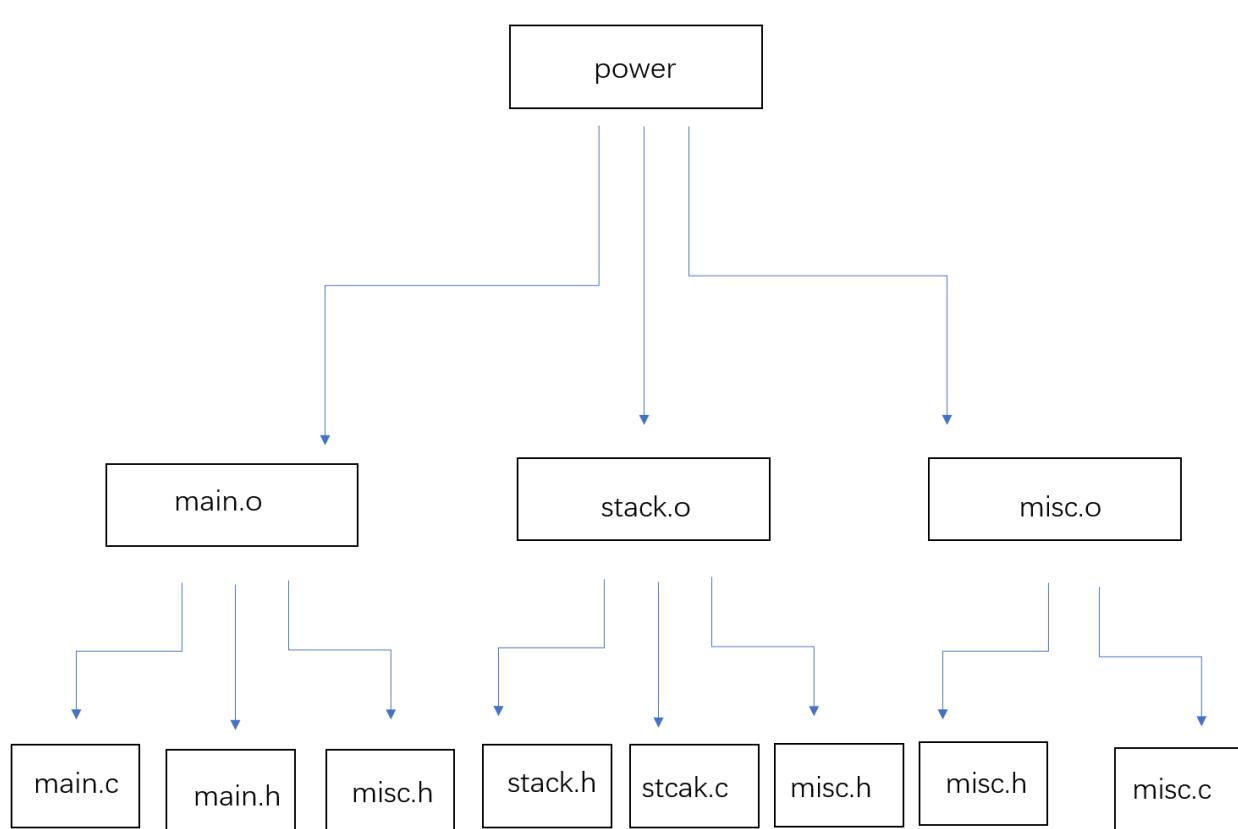
- main.o依赖main.c main.h misc.h
- stack.o依赖stack.c stack.h misc.h
- misc.o依赖misc.c misc.h

#### 4. 生成每个目标文件所需执行的命令

- power: gcc -O3 -o power main.o stack.o misc.o -lm
- main.o: gcc -o main.o main.c main.h misc.h
- stack.o :gcc -o stack.o stack.c stack.h misc.h
- misc.o: gcc -o misc.o misc.c misc.h

#### 5. 画出makefile对应的依赖关系树

•



#### 6. 生成main.o stack.o和misc.o时会执行哪些命令，为什么？

- 生成这些文件时，会执行如下的命令：
 

```
main.o: gcc -o main.o main.c main.h misc.h
stack.o :gcc -o stack.o stack.c stack.h misc.h
misc.o: gcc -o misc.o misc.c misc.h
```

2.用编辑器创建main.c、compute.c、input.c、compute.h、input.h和main.h文件。下面是它们的内容。注意compute.h和input.h文件仅包含了compute和input函数的声明但没有定义。定义部分是在compute.c和input.c文件中。main.c包含的是两条显示给用户的提示信息。

```
$ cat compute.h
/* compute函数的声明原形 */
```

```
double compute(double, double);
$ cat input.h
/* input函数的声明原形 */
double input(char *);
$ cat main.h
/* 声明用户提示 */
#define PROMPT1 "请输入x的值: "
#define PROMPT2 "请输入y的值: "
$ cat compute.c
#include <math.h>
#include <stdio.h>
#include "compute.h"
double compute(double x, double y)
{
    return (pow ((double)x, (double)y));
}
$ cat input.c
#include <stdio.h>
#include "input.h"
double input(char *s)
{
    float x;
    printf("%s", s);
    scanf("%f", &x);
    return (x);
}
$ cat main.c
#include <stdio.h>
#include "main.h"
#include "compute.h"
#include "input.h"

main()
{
    double x, y;
    printf("本程序从标准输入获取x和y的值并显示x的y次方.\n");
    x = input(PROMPT1);
    y = input(PROMPT2);
    printf("x的y次方是:%.3f\n", compute(x,y));
}
```

为了得到可执行文件power，我们必须首先从三个源文件编译得到目标文件，并把它们连接在一起。下面的命令将完成这一任务。注意，在生成可执行代码时不要忘了连接上数学库。

```
$ gcc -c main.c input.c compute.c
$ gcc main.o input.o compute.o -o power -lm
```

1. 创建上述三个源文件和相应头文件，用gcc编译器，生成power可执行文件，并运行power程序。给出完成上述工作的步骤和程序运行结果。

- 创建源文件：使用cat命令从终端直接输入源代码即可

```
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ cat >compute.h
/* compute函数的声明原形 */
double compute(double, double);
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ cat > input.h
/* input函数的声明原形 */
double input(char *);
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ cat > main.h
/* 声明用户提示 */
#define PROMPT1 "请输入x的值: "
#define PROMPT2 "请输入y的值: "
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ cat >compute.c
#include <math.h>
#include <stdio.h>
#include "compute.h"
double compute(double x, double y)
{
    return (pow ((double)x, (double)y));
}
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ cat > input.c
#include <stdio.h>
#include "input.h"
double input(char *s)
{
    float x;
    printf("%s", s);
    scanf("%f", &x);
    return (x);
}

handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ cat > main.c
#include <stdio.h>
#include "main.h"
#include "compute.h"
#include "input.h"

main()
{
    double x, y;
    printf("本程序从标准输入获取x和y的值并显示x的y次方.\n");
    x = input(PROMPT1);
    y = input(PROMPT2);
    printf("x的y次方是:%.3f\n",compute(x,y));
}
```

然后对代码进行编译和链接：

```
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ gcc -c main.c input.c compute.c
main.c:6:1: warning: return type defaults to 'int' [-Wimplicit-int]
  6 | main()
    | ^~~~
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ gcc main.o input.o compute.o -o power -lm
```

使用ls命令查看我们的文件夹中生成了哪些文件：

```
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ ls -l
总用量 52
-rw-rw-r-- 1 handsomehh handsomehh 135 7月 18 11:02 compute.c
-rw-rw-r-- 1 handsomehh handsomehh 67 7月 18 11:01 compute.h
-rw-rw-r-- 1 handsomehh handsomehh 1384 7月 18 11:04 compute.o
-rw-rw-r-- 1 handsomehh handsomehh 122 7月 18 11:02 input.c
-rw-rw-r-- 1 handsomehh handsomehh 55 7月 18 11:01 input.h
-rw-rw-r-- 1 handsomehh handsomehh 1736 7月 18 11:04 input.o
-rw-rw-r-- 1 handsomehh handsomehh 270 7月 18 11:02 main.c
-rw-rw-r-- 1 handsomehh handsomehh 101 7月 18 11:01 main.h
-rw-rw-r-- 1 handsomehh handsomehh 1968 7月 18 11:04 main.o
-rwxrwxr-x 1 handsomehh handsomehh 16264 7月 18 11:05 power
```

运行我们的power程序：

```
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ ./power
本程序从标准输入获取x和y的值并显示x的y次方。
请输入x的值：2
请输入y的值：3
x的y次方是：8.000
```

## 2. 创建Makefile文件，使用make命令，生成power可执行文件，并运行power程序。给出完成上述工作的步骤和程序运行结果。

- 首先删除第一问中生成的不必要的中间文件：

```
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ rm -rf *.o
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ ls -l
总用量 40
-rw-rw-r-- 1 handsomehh handsomehh 135 7月 18 11:02 compute.c
-rw-rw-r-- 1 handsomehh handsomehh 67 7月 18 11:01 compute.h
-rw-rw-r-- 1 handsomehh handsomehh 122 7月 18 11:02 input.c
-rw-rw-r-- 1 handsomehh handsomehh 55 7月 18 11:01 input.h
-rw-rw-r-- 1 handsomehh handsomehh 270 7月 18 11:02 main.c
-rw-rw-r-- 1 handsomehh handsomehh 101 7月 18 11:01 main.h
-rwxrwxr-x 1 handsomehh handsomehh 16264 7月 18 11:05 power
```

然后用vim编辑器直接输入和编写我们的MakeFile文件：

```

CC = gcc
OPTIONS = -O3 -o
OBJS = main.o compute.o input.o
power: $(OBJS)
    $(CC) $(OPTIONS) $@ $^ -lm
main.o: main.h input.h compute.h
input.o: input.h
compute.o: compute.h
.PHONY : clean
clean :
    -rm -f *.o

```

保存后退出，并在该界面下使用make

命令：

```

handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ make clean
rm -f *.o
handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ make
gcc     -c -o main.o main.c
gcc     -c -o compute.o compute.c
gcc     -c -o input.o input.c
gcc -O3 -o power main.o compute.o input.o -lm

```

然后以运行我们所生成的可执行文件：

```

handsomehh@handsomehh-virtual-machine:~/professional/courses/major/cs381/labs$ ./power
本程序从标准输入获取x和y的值并显示x的y次方。
请输入x的值：4
请输入y的值：2
x的y次方是：16.000

```

发现结果是正确的。

### 3. 编写shell 脚本，统计指定目录下的普通文件、子目录及可执行文件的数目，统计该目录下所有普通文件字节数总和，目录的路径名字由参数传入。

- 代码如下：

```

#!/bin/bash
#保存输入的目录路径
dir=$1
#当输入的参数多于1个时，报错并返回错误代码
if [ $# -ne 1 ];then
    echo "每次只能输入一个目录"
    exit 1
#当输入的参数恰好是一个并且是目录文件时
elif [ -d $1 ];then
#数组num存储的是不同种类的文件数
    declare -a num
    num=(0 0 0 )

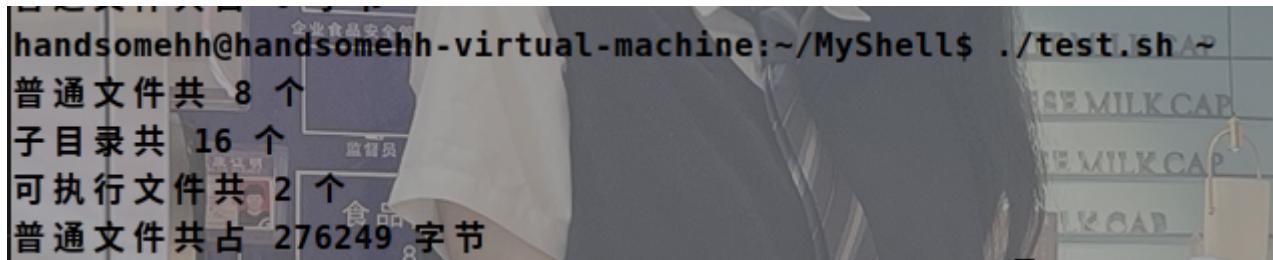
```

```

#num存储的是所有文件的字节总数
declare -i sum
sum=0
#使用set命令，使dir目录下的所有名称得到遍历
set $(ls $1 )
#for循环便利文件
for i in $@
do
#当这个文件是普通文件时
if [ -f $dir/$i ];then
(( num[1]++ ))
if [ -x $dir/$i ];then
(( num[0]++ ))
fi
#统计所有的文件字节数
(( sum += $(cat $dir/$i | wc -c) ))
#当这个文件是目录文件时
elif [ -d $dir/$i ];then
(( num[2]++ ))
fi
done
#打印结果
echo "普通文件共 ${num[1]} 个"
echo "子目录共 ${num[2]} 个"
echo "可执行文件共 ${num[0]} 个"
echo "普通文件共占 $sum 字节"
exit 0
else
#当这个路径不是目录文件
echo "您输入的并非目录名"
exit 1
fi

```

- 运行效果如图所示：



4. 编写shell 脚本，输入一个字符串，忽略（删除）非字母后，检测该字符串是否为回文(palindrome)。对于一个字符串，如果从前向后读和从后向前读都是同一个字符串，则称之为回文串。例如，单词“mom”，“dad”和“noon”都是回文串

- 代码如下：

```

#!/bin/bash
if [ $# -eq 1 ];then

```

```

string=$1
length=0
declare -i flag=0
declare -a str
for((i=0;i<#${string};i++))
do
    if [[ ${string:$i:1} = [a-zA-Z] ]];then
        str[$length]=${string:$i:1}
        ((length++))
    fi
done
for((i=0;i<($length/2);i++))
do
    if [[ ${str[$i]} != ${str[$((length-$i-1))]} ]];then
        echo "${str[$i]} != ${str[$((length-$i-1))]}"
        flag=1
        break
    fi
done
if [ $flag -eq 0 ];then
    echo "$string (${str[@]})是回文串"
else
    echo "$string 不是回文串"
fi
exit 0
else
    echo "输入的参数太多"
    exit 1
fi

```

效果如图：

```

handsomehh@handsomehh-virtual-machine:~/MyShell$ ./test.sh abc11ba
abc11ba (a b c b a)是回文串
handsomehh@handsomehh-virtual-machine:~/MyShell$ ./test.sh Aba
A != a
Aba 不是回文串
handsomehh@handsomehh-virtual-machine:~/MyShell$ ./test.sh aba
aba (a b a)是回文串

```

其中会删除非数字后再次判断

5. 编写一个实现文件备份和同步的shell脚本程序dirsync。程序的参数是两个需要备份同步的目录，如：dirsync ~\dir1 ~\dir2 # ~\dir1为源目录，~\dir2为目标目录。dirsync程序实现两个目录内的所有文件和子目录（递归所有的子目录）内容保持一致。程序基本功能如下。

- (1) 备份功能：目标目录将使用来自源目录的最新文件，新文件和新子目录进行升级，源目录将保持不变。dirsync程序能够实现增量备份。
- (2) 同步功能：两个方向上的旧文件都将被最新文件替换，新文件都将被双向复制。源目录被删除的文件和子目录，目标目录也要对应删除。
- (3) 其它功能自行添加设计。

- 代码如下：

```
#!/bin/bash
#定义函数，该函数用来判断源目录中的文件是否在目标文件
function in_tar(){
    #保存传入的参数和变量
    local x=$1
    local n=$2
    local i
    local dir_t=$3
    #遍历传入的文件
    for i in ${dir_t[*]}
    do
        #一旦找到就退出
        if [ "$x" = "$i" ];then
            return 1
        fi
    done
    #否则返回0
    return 0
}
#该函数用于判断给定文件是否在源文件目录中
function in_sou(){
    #保存变量
    local x=$1
    local n=$2
    local i
    dir_t=$3
    #遍历目录
    for i in ${dir_t[*]}
    do
        if [ "$x" = "$i" ];then
            #找到后返回1
            return 1
        fi
    done
    #没找到返回0
    return 0
}
#该函数用于比较时间，t1和t3来自一组，t2和t4来自一组
function comptime(){
    local t1=$1
    local t2=$2
    local t3=$3
    local t4=$4
    #下面用于比较两个时间哪个在前
    if (( $t1 < $t3 ));then
        #"返回2"
        return 2
    elif (( $t1 > $t3 ));then
        #"返回1"
        return 1
    else

```

```
if (($t2 == $t4));then
    # "返回0"
    return 0
elif (($t2 > $t4));then
    # "返回1"
    return 1
else
    # "返回2"
    return 2
fi
fi
}

#执行主体
function dirsync(){
#定义一些有用的变量
local -a dir_tar
local -a dir_sou
local -i tar_len
local -i sou_len
local -a tar_time
local -a sou_time
local i
local tar=$2
local sour=$1
#查看参数传入的个数是否正确
if [ $# -ne 2 ];then
    echo "参数的个数不是两个"
else
    echo "$1 $2"
    if [[ ! -d $1 || ! -d $2 ]];then
        echo "输入的不是目录"
        exit 1
    fi
    if [[ $1 = $2 ]];then
        echo "备份两个相同的目录"
        exit 0
    fi
#设置变量
set $(ls $tar)
#目录内容与长度
dir_tar=("$@")
tar_len=$#
set $(ls $sour)
#目录内容与长度
dir_sou=("$@")
sou_len=$#
#遍历目标文件夹中的文件
for i in ${dir_tar[@]}
do
    #看他是否在源目录中
    in_sou $i $sou_len "${dir_sou[*]}"
    temp=$?
    if [ $temp -eq 0 ];then
        # "这个没有找到, 直接删除该目录
        rm -rf $i
    else
        # "这个存在, 将其替换为新的
        mv $i ${dir_sou[*]}
    fi
done
}
#解压缩
function decompress(){
#解压缩
}
```

```
if rm -rf $tar/$i;then
    echo "删除$tar/$i"
fi
fi
done
#重新设置变量和长度
set $(ls $tar)
dir_tar="$@"
tar_len=$#
#遍历源目录
for i in ${dir_sou[@]}
do
    in_tar $i $tar_len "${dir_tar[*]}"
    temp=$?
    #查看当前文件是否出现在目标目录中
    if [ $temp -eq 0 ];then
        #这个没有找到，则直接复制拷贝过去
        if cp -r -p $sour/$i $tar;then
            echo "备份$sour/$i至$tar"
        fi
    fi
done
#重新遍历源目录
for i in ${dir_sou[@]}
do
    #对于源目录中的文件夹
    if [ -d $sour/$i ];then
        #获取创造时间
        set -- $(stat --format=%y $tar/$i)
        tar_time[0]="${1:0:4}" "${1:5:2}" "${1:8:2}"
        tar_time[1]="${2:0:2}" "${2:3:2}" "${2:6:2}"
        set -- $(stat --format=%y $sour/$i)
        sou_time[0]="${1:0:4}" "${1:5:2}" "${1:8:2}"
        sou_time[1]="${2:0:2}" "${2:3:2}" "${2:6:2}"
        #比较这个文件在两个文件夹中的事件
        comptime ${tar_time[0]} ${tar_time[1]} ${sou_time[0]} ${sou_time[1]}
        if [ $? -eq 0 ];then
            #"时间相同不处理"
            continue
        else
            #否则递归调用这两个文件夹中的内容，对其进行备份处理
            #echo "递归调用dirsync $sour/$i $tar/$i"
            dirsync $sour/$i $tar/$i
            #echo "结束调用dirsync $sour/$i $tar/$i"
        fi
    else
        #对于非目录而言，获取时间
        set -- $(stat --format=%y $tar/$i)
        tar_time[0]="${1:0:4}" "${1:5:2}" "${1:8:2}"
        tar_time[1]="${2:0:2}" "${2:3:2}" "${2:6:2}"
        set -- $(stat --format=%y $sour/$i)
        sou_time[0]="${1:0:4}" "${1:5:2}" "${1:8:2}"
        sou_time[1]="${2:0:2}" "${2:3:2}" "${2:6:2}"
        #比较时间
    fi
done
```

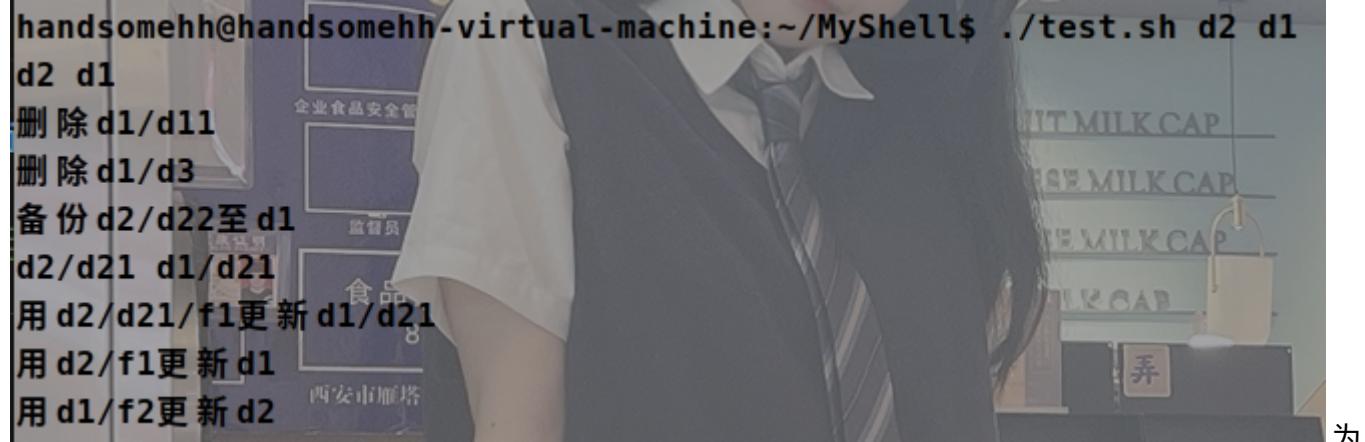
```

comptime ${tar_time[0]} ${tar_time[1]} ${sou_time[0]} ${sou_time[1]}
temp=$?
#根据时间做处理
if [ $temp -eq 0 ];then
    #时间相同不处理
    continue
elif [ $temp -eq 2 ];then
    #时间不同则相互更新
    if cp -r -p $sour/$i $tar;then
        echo "用$sour/$i更新$tar"
    fi
else
    #时间不同相互更新
    if cp -r -p $tar/$i $sour;then
        echo "用$tar/$i更新$sour"
    fi
fi
fi
done

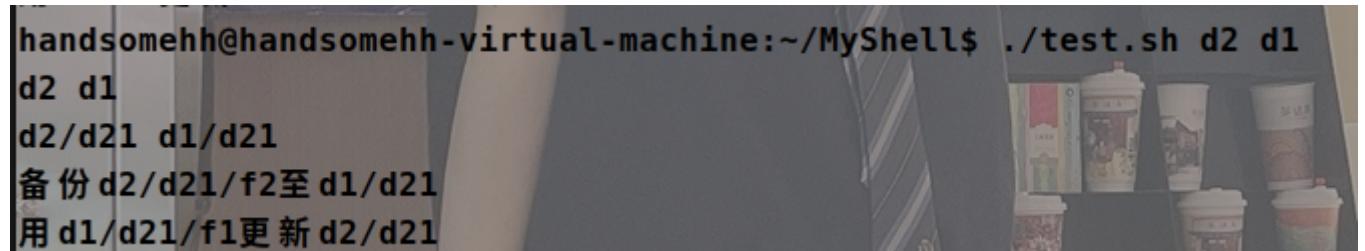
fi
}
#调用该程序
dirsync $1 $2

```

结果如图：我们首先随机创建两个目录，并在里面添加各种目录和文件，然后运行备份，需要注意的是，对于运行参数，d2和d1都需要是创建好的目录，且第一个参数是源目录（d2），第二个参数是目标目录（d1）



为了测试实验结果，我们在d2中创建一个新文件d2/d21/f2，然后在d1中更新d1/d21/f1，我们发现，结果中表明：



显然，我们用源目录(d2)中的f2拷贝到了目标目录d1，并且我们用目标目录中的最新文件，双向备份了源目录中的f1，我们紧接着验证删除功能，我们在d2中删除一个目录d22，然后运行：

```
handsomehh@handsomehh-virtual-machine:~/MyShell$ ./test.sh d2 d1  
d2 d1  
删除 d1/d22  
d2/d21 d1/d21
```

显

然，我们成功在目标目录中删除了源目录中已经被删除的文件。其他题目中要求的功能本实验均尽数实现

## 6. 使用bash，编写一个命令行编辑器程序，编辑器的功能参考vim/vi，实现vim部分功能即可。不能直接调用现成的编辑器。

要求：不能使用开源代码，自己编写所有的代码。鼓励使用图形界面。本实验题要求提供以下文档： a. 有功能描述文档。

- b. 设计文档，包括设计思想、功能模块、数据结构、算法等。
- c. 源程序。有详细的注释和良好编程风格。

### 1. 实验文档

- 设计语言：本实验采用bash作为实验的设计语言，实现了简易版的Myvim，但是要着重说明的是，为了保证文本显示的格式准确，本程序是处理**纯英文文本文件**的，虽然读者可以编辑和输入中文，但是由于中文占位符是两个，所以会导致格式的偏差，此外，本程序输入制表符后，格式也会发生偏差，所以，**使用者不要输入中文和制表符**，该MyVim支持三种模式，分别为：normal模式，visual (cmd) 模式和insert模式，三种模式下的界面如下所示（注：背景是我的终端壁纸，不是本程序实现的内容）：

normal模式



insert模式



cmd模式



- 设计框架与功能：为了模仿vim的操作和界面，本程序实现了如下功能：

三种模式下的转化：在normal模式下，输入"::"可以切换到cmd模式，输入"i"切换到insert模式，而在

insert和cmd模式下，均可以使用esc键切换到normal模式。

界面风格与特点：本程序保证在界面中显示出恰好一整页的文本，读者使用光标的移动可以浏览全部的文本。但是，注意在使用时**最好不要改变当前界面的大小**，否则文本将会串行，串行后，你需要再次输入文本或者执行任何一个操作，才可以恢复改变大小后的恰好整页显示，造成这种情况的原因在于无法在改变界面大小后及时的刷新界面，且为了适配整页的文本显示我对于文本本身增加了许多判断和操作

normal模式：在此模式下，使用者可以使用光标的移动来浏览所有的文本内容，而光标的移动可以通过上下左右或hjkl键实现，normal模式下，界面下方会显示">>> <打开的文本名称>： normal X: <光标的x坐标> Y: <光标的Y坐标> scale: <当前打开已经浏览到的百分比>%",如" >>"1.txt" : normal X: 1 Y:1 scale: 37%”。此外，在这一模式下，同样支持使用"^"快捷键操作，可以回到行首

insert模式：此模式下，可以对文本进行编辑，当然也支持两种快捷键："^" 和 "HOME"键，前者可以快速切换到一行的行首，后者可以切换当当前显示页的行首，然后使用者直接输入自己想要输入的文本就可以实现对文本的编辑了

cmd(visual)模式：此模式下，读者可以使用一些快捷键，并对文本进行保存，或者是退出文件，具体的使用命令如下：

w:保存

wq: 保存并退出

q:退出

gg: 返回第一行第一列

G: 到达当前屏幕的最后一行的第一个

HOME键:到达当页文本的开头

w+空格+路径:将文件另存为一份，且该份的路径为空格后输入的路径

- 使用样例：

在界面输入一些文字

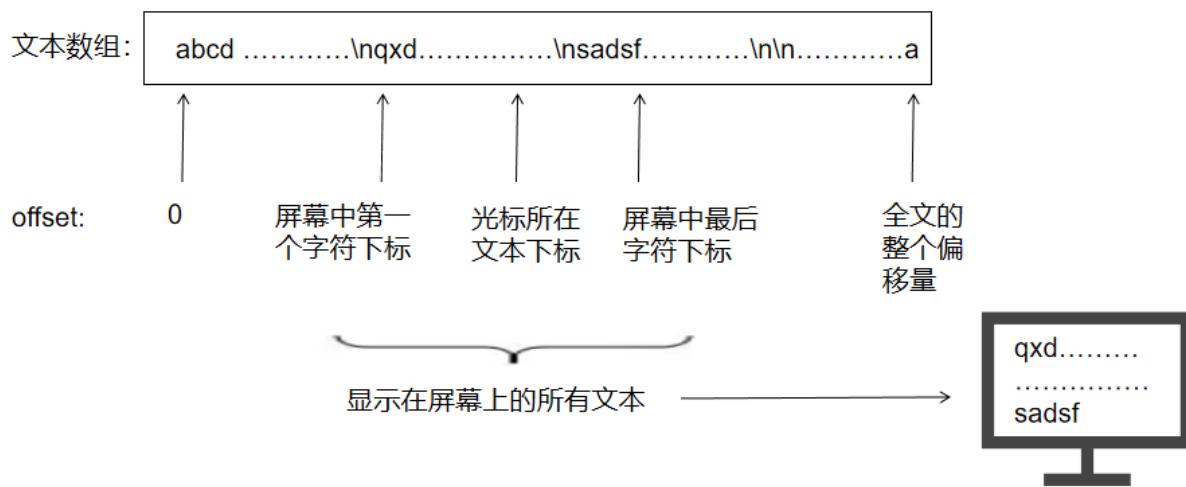


然后保存并退出



## 2. 实验设计思路和数据结构

- 本实验中，为了实现文本的显示和编辑功能，主要的数据结构为shell中的数组，我们可以将文本视为一段长信息，我们可以把信息看作一个数组，其中换行等操作不过是在这一段信息中的不同位置上，存储的字符不同，有些地方存储的是回车，而有些地方存储着普通字符，而区别这些数据的方式，则是下标的不同，不同位置上的字符有着不同的数组下标，基于这一点，我们可以定义比较重要的三个变量，分别为显示在屏幕上的第一个字符的数组下标，显示在屏幕上最后一个字符的数组下标，以及整篇文章所有的字符数。在此思路下，我们便可以实现界面的显示控制，利用stty函数提取出的屏幕大小和数组中对于回车的字符特殊控制，实现整张页面的文本的精准显示，而该程序的大部分代码都是在处理不同的字符在显示时，光标究竟如何变化，以及光标在变化后如何对应到整个数组特定字符的下标偏移量。



- 光标移动和处理：为了方便光标的移动和处理，我设置了一个数组用来记录显示在屏幕上的每一行的文本长度，这样我们就可以控制光标在屏幕中上下左右移动时，始终能与显示的文本保持一致了，当然，在处理的过程中可能存在许多特殊情况需要考虑，比如每一行之后的回车，再例如当最后一行无法完全显示时我们要如何处理文本，这些特殊处理的代码也是程序的主要部分

### 3. 代码与实现

```
#!/bin/bash

# 定义全局变量

declare -x filename # 打开的目标文本名称
declare -x row # 打开的目标文本行数
declare -x content # 内存存储在该变量

declare -x width # 屏幕的宽度
declare -x height # 屏幕的高度
declare -x mode # 当前的模式, 0是normal, 1是insert, 2是 visual (cmd模式)

declare -x begin_offset # 屏幕中的文本开头偏移量
declare -x end_offset # 屏幕文本结束位置
declare -x offset # 文本总的偏移
declare -x cur_offset # 当前光标所在的偏移量

declare -x mouse_x # 光标的x坐标
declare -x mouse_y # 光标的y坐标

declare -x tty_row # 屏幕的行数
```

```
declare -x tty_clo #屏幕的列数

declare -xai screen_rowlen #屏幕中的每一行的字符数
declare -x cmd_input #cmd模式下的输入
declare -x row_occupy #屏幕中显示了多少行
declare -x delete_mask #删除时的目标

#-----
#remind函数打印屏幕中的提示信息
#-----
function remind(){
    #保存传入参数
    local re="$1"
    #提取屏幕的大小
    width=$(stty size | awk '{print $2}')
    height=$(stty size | awk '{print $1}')
    #清屏
    clear
    #打印鼠标位置
    printf '\033[%d;%dH' $height 1
    #判断mode
    case $mode in
        0) #normal
            string="\\"$filename\\"" : normal"
            Cal_cur_offset

            local percent
            if ((offset==0));then
                percent=0
            else
                percent=$((100*cur_offset/offset))
            fi
            #打印鼠标位置, 以及当前阅读的百分比
            printf ">>$string X:$mouse_x Y:$mouse_y scale:$percent %"
        ;;
        1) #insert
            string="\\"$filename\\"" : insert"
            #打印鼠标位置
            printf ">>$string X:$mouse_x Y:$mouse_y"
        ;;

        2) #visual
            string="\\"$filename\\":visual"
            printf ">> :$re"
        ;;

    esac
}

#-----
#Cal_cur_offset函数计算当前鼠标位置所在的文本偏移量
#-----
function Cal_cur_offset(){
    sum=0
    #遍历每一行
```

```
for ((i=1;i<=$mouse_y-1;i++))
do
    ((sum+=${screen_rowlen[$i]}))
done
#加上当前位置
cur_offset=$((sum+$mouse_x+$begin_offset-1))
}

#-----
#is_n函数判断当前位置的前一个字符是不是回车
#-----
function is_n(){
Cal_cur_offset
#当是回车时，返回值为1
if [[ ${content:$((cur_offset-1)):1} = '\n' ]];then
    return 1
else
#否则返回0
    return 0
fi
}

#-----
#Back_n函数，用于文本回退时，判断是否需要
#重开一行，即如果当前光标在 (1, 1) ，假设
#前面还有文本，在回退时要将前面的文本回滚
#回来，但此时的鼠标要跟在上一行的末尾，因
#此该函数是保证格式正确的必要函数
#-----
function Back_n(){
#当开始的偏移量是0时，什么都不做
if (($begin_offset<=0));then
    return 0
fi
printf "$begin_offset"
#否则去找他上一个行的开始字符
local pos=$((begin_offset-1))
while true
do
    #直到找到上一行的开始标志
    if (($pos<=0)) || [[ ${content:$((pos-1)):1} = '\n' ]];then
        break
    fi
    #每次位置向前推移一个
    ((pos--))
done
printf "mouse_x=$begin_offset-$pos yu $width -1"
#此时的pos恰好指向一行的最后一个字符
if (((($begin_offset-$pos))%$width == 0));then
#当中间的字符数量能被宽度整除
    if [[ ${content:$((begin_offset-1)):1} = '\n' ]];then
#上一个字符恰好是回车时
        mouse_x=$((width-1))
        begin_offset=$((begin_offset-$width))
    fi
fi
}
```

```
else
#当上一个字符不是回车时
    mouse_x=$width
    begin_offset=$((begin_offset-$width))
fi
elif (((begin_offset-$pos))%$width == 1));then
#当余1时
    if (( $((begin_offset-$pos)) == 1 ));then
        #有可能前面的字符只有一个
        begin_offset=$((begin_offset-1))
        mouse_x=1
    else
        #前面的字符有多个
        mouse_x=$width
        begin_offset=$((begin_offset-$width-1))
    fi
else
#对于一般情况
    mouse_x=$((($begin_offset-$pos)%$width-1))
    begin_offset=$((begin_offset-$mouse_x-1))
fi
#保证光标的位置非0
if (($mouse_x == 0));then
    mouse_x=1
fi
}

#-----
#normal_deal函数，处理normal模式下的键盘事件
#-----
function normal_deal(){
#保存参数
input="$1"
#计算当前便宜
Cal_cur_offset

case "$input" in
i)
#输入i，跳转插入模式
mode=1
;;
^)
#输入^，回到行首
mouse_x=1
;;
h|$'\E[D')
#光标左移
if (($mouse_x > 1));then
#一般情况下，直接x--
((mouse_x--))
else
#当到达每一行的第一个字符时，光标左移需要考虑换行问题
if (($mouse_y == 1));then
    Back_n
```

```
elif (($mouse_y>1));then
#当是在屏幕中间的行时
    if (( ${screen_rowlen[$mouse_y-1]} == $width));then
#当他的上一行满了时
        if Is_n ;then
            #如果上一行最后一个回车
            mouse_x=$(${screen_rowlen[$mouse_y-1]}-1)
            #则mouse_x=边界宽减1
            if (($mouse_x==0));then
                mouse_x=1
            fi
            ((mouse_y--))
        else
            #如果不是回车
            mouse_x=$(${screen_rowlen[$mouse_y-1]})
            #等于边界宽
            if (($mouse_x==0));then
                mouse_x=1
            fi
            #y--
            ((mouse_y--))
        fi
    else
#当上一行没满时
        mouse_x=$(${screen_rowlen[$mouse_y-1]}-1)
        #之间等于边界宽减1
        if (($mouse_x==0));then
            mouse_x=1
        fi
        #y--
        ((mouse_y--))
    fi
fi
#保证光标的位置始终大于0
if (($mouse_x==0));then
    mouse_x=1
fi
fi
;;
j|$'\E[B'|$'\n')
#当向下一移动光标时
if ((mouse_y<$height-2))&&((mouse_y<row_occupy));then
#如果在屏幕中间或者是在小于结尾处
    ((mouse_y++))
#判断当前的光标是否在下一行末尾的后面
    if (( $mouse_x >= ${screen_rowlen[$mouse_y]}-1 ));then
        mouse_x=$(( ${screen_rowlen[$mouse_y]}-1 ))
    fi
#如果达到了最后一行
elif ((mouse_y==$height-2));then
    num=0
    flag=0
#从当前位置一直排查到最后一个位置
    for ((i=1;i<=$(offset-end_offset-1));i++))
```

```
do
#每排查一个位置num++
((num++))
#当找到有回车或者位置的数量大于屏幕的宽度时
if [[ ${content:$((end_offset+i)):1} = '$\n' ]] || (($num >=
$width));then
    flag=1
    break
fi
done
#如果可以参与显示
if ((flag == 1));then
    ((begin_offset+=$screen_rowlen[1]))
    remind ""
    display_content $begin_offset
    printf '\033[%d;%dH' $mouse_y $mouse_x
    mouse_x=1
fi
else
#如果到了屏幕中的非最后一行的末尾位置
:
fi
#保证mouse_x始终大于0
if (($mouse_x==0));then
    mouse_x=1
fi
;;
k|$'\E[A')
#向上翻页
if ((mouse_y>1));then
#屏幕中间时，直接y--
((mouse_y--))
#保证格子不会不溢出
if (( $mouse_x >= ${screen_rowlen[$mouse_y]}-1 ));then
    mouse_x=$(( ${screen_rowlen[$mouse_y]}-1 ))
fi
else
#在第一行时，调用back_n
Back_n
fi
#保证x大于等于1
if (($mouse_x==0));then
    mouse_x=1
fi
;;
1| $'\E[C')
#计算当前的位移量
Cal_cur_offset
#当前位移量大于或等于总偏移量时，无视本次操作
if (($cur_offset>=$offset));then
    return 0
fi
#当鼠标x坐标小于当前行的最大字母数时
if (($mouse_x < ${screen_rowlen[$mouse_y]}-1));then
```

```

#向右偏移一位
((mouse_x++))

else
#当鼠标大于等于最大字母数时
if (($mouse_y == $height - 2));then
#当y坐标达到底部时，判断是否需要换行，处理结尾偏移量
if (($mouse_x == ${screen_rowlen[$mouse_y]}-1));then
if [[ ${content:$((cur_offset+1)):1} = '\n' ]];then
#换行时保证begin offset的正确
((begin_offset+=$screen_rowlen[1]))
mouse_x=1
else
#x++
((mouse_x++))
fi
else
if ((cur_offset == offset - 1));then
#当达到末尾，需要实现如下的情况：字符1 字符2 光标，以实现结尾的追加
if ((mouse_x < $width));then
#不需要换行时
((mouse_x++))
else
#需要换行时
((begin_offset+=$screen_rowlen[1]))
mouse_x=1
#((mouse_y++))
fi
else
((begin_offset+=$screen_rowlen[1]))
mouse_x=1
fi
fi
else
#当还在屏幕中间时，且达到文本的最后一行
if (( mouse_y == row Occupy));then
#若在屏幕的最后一行，因为最后一行的单词个数不用考虑最后的换行问题

if (($mouse_x == ${screen_rowlen[$mouse_y]}-1));then
#所以当x坐标是最大位移量-1时直接++
((mouse_x++))
elif (($mouse_x == ${screen_rowlen[$mouse_y]}));then
#否则判断是否另起一行
if ((mouse_x < $width));then
((mouse_x++))
else
mouse_x=1
((mouse_y++))
fi
#处理后达到offset位置
fi

else
#当还在屏幕中间时的一般情况

```

```
if (($mouse_x == ${screen_rowlen[$mouse_y]}-1));then
    if [[ ${content:$($cur_offset+1)}:1 = '$\n' ]];then
        #当遇到回车
        mouse_x=1
        ((mouse_y++))
    else
        #当一般情况
        ((mouse_x++))
    fi
else
    #当到达最大偏移量
    mouse_x=1
    ((mouse_y++))
fi
fi
#显示文本
remind ""
#更新每一行的状态
display_content $begin_offset
#鼠标更新
printf '\033[%d;%dH' $mouse_y $mouse_x
;;
:)
#输入: 时, 进入visual (cmd) 模式
mode=2
;;
esac
#更新实时位置
Cal_cur_offset
}

#-----
#display_content函数, 显示所有文本, 并跟新
#一些状态
#-----
function display_content(){
    #将光标指向1,1
    printf '\033[%d;%dH' 1 1
    #保存offset
    local store=$1
    local begin_offset=$1
    #默认行初始值为1
    row_occupy=1
    local clo_occupy=0
    #默认列初始值为0
    #初始化每行的存储列表
    for ((i=1;i<=$height - 2;i++))
    do
        screen_rowlen[$i]=0
    done
    #记录回车数
    num=0
```

```
local flag=0
#开始显示与渲染
while true
do
    #当前展示的数为最后一个数时
    if (( $begin_offset == $offset));then
        #结束符号是最后一个数
        end_offset=$((offset-1))
        #记录当前行的列数
        screen_rowlen[$row_occupy]=$clo_occupy
        flag=1
        break
    fi
    #当前展示为回车时
    if [[ ${content:$begin_offset:1} = '$\n' ]];then
        ((num++))
        #该行的字符数加一
        screen_rowlen[$row_occupy]=$(($clo_occupy+1))
        clo_occupy=0
        #行数加一
        ((row_occupy++))
        end_offset=$begin_offset
    elif (( $clo_occupy == $width));then
        #当前行满时
        screen_rowlen[$row_occupy]=$clo_occupy
        clo_occupy=1
        ((row_occupy++))
        end_offset=$((begin_offset-1))
    else
        #普通字符的展示直接加加
        ((clo_occupy++))
    fi
    #如果屏幕已经占满
    if (($row_occupy > $height - 2));then
        ((row_occupy--))
        break
    fi
    #printf "${content:$begin_offset:1}"
    #展示下一个字符
    ((begin_offset++))
done
if ((begin_offset==0));then
    #当文本头部时
    end_offset=0
else
    #否则不对之前得到的offset操作
    :
    #end_offset=$((begin_offset-1))
fi
#打印文本
printf "${content:$store:$($begin_offset-$store)}"
#下面的循环是debug时使用的信息，此处无影响
for ((i=1;i<=$row_occupy;i++))
do
```

```
:  
#printf "\n$i:${screen_rowlen[$i]}"  
done  
}  
  
#-----  
#Init函数, 初始化一些状态  
#-----  
function Init(){  
    #光标默认为 (1, 1)  
    mouse_x=1  
    mouse_y=1  
    #计算当前的屏幕大小  
    height=$(stty size | awk '{print $2}')  
    width=$(stty size | awk '{print $1}')  
    #计算偏移量  
    Cal_cur_offset  
}  
  
#-----  
#visual_deal函数, 处理cmd模式下的命令  
#-----  
function visual_deal(){  
    #存储传入的参数  
    cmd_input="$1"  
    num=0  
    #当输入的是HOME键  
    if [[ $cmd_input = '$\e[H' ]];then  
        #打印提示信息  
        printf "Home"  
        sleep 2  
        #更新光标位置  
        mouse_x=1  
        mouse_y=1  
        #更新mode和offset  
        begin_offset=0  
        mode=0  
        #退出函数  
        return 0  
    fi  
    #否则一直循环  
    while true  
    do  
        #光标位置控制  
        printf '\033[%d;%dH' $height 5  
        printf ""  
        printf '\033[%d;%dH' $height 5  
        printf "$cmd_input"  
        #读取输入  
        read -sN1 text  
        read -sN1 -t 0.0001 k1  
        read -sN1 -t 0.0001 k2  
        read -sN1 -t 0.0001 k3  
        #拼接字符串
```

```
text+="${k1}${k2}${k3}
#输入回车时
if [[ $text = '$\n' ]];then
#准备处理并返回normal模式
    mode=0
    break
elif [[ $text = '$\E' ]];then
#输入esc, 直接返回
    cmd_input=""
    mode=0
    break
elif [[ $text = '$\E[3~' || $text = '$\177' ]];then
#输入delete/backspace时
    if ((#${cmd_input}>0));then
#输入的cmd命令回退一个
        cmd_input=${cmd_input:0:${#${cmd_input}-1}}
        ((num--))
    fi
elif [[ $text = '$\e[H' ]];then
#打印提示信息
    printf "Home"
    sleep 2
#更新光标位置
    mouse_x=1
    mouse_y=1
#更新mode和offset
    begin_offset=0
    mode=0
#退出函数
    return 0
else
    if ((num<=10));then
#cmd命令字符最多不超过10个
        cmd_input=${cmd_input}"$text"
        ((num++))
    fi
fi
done
#处理输入的cmd命令
case "$cmd_input" in
"q" )
#q代表直接退出
    clear
    exit 1
    ;;
"wq" )
#wq保存并退出
    printf "$content" > $filename
    clear
    exit 1
    ;;
"w" )
#w保存
    printf "$content" > $filename
```

```
    return 0
    ;;
"gg")
#gg返回第一行第一列
mouse_x=1
mouse_y=1
begin_offset=0
mode=0
return 0
;;
"G")
#G到达当前屏幕的最后一行的第一个
mouse_y=$row Occupy
mouse_x=1
mode=0
return 0
;;
esac
#另存为功能w+空格+其他
if [[ "${cmd_input:0:1}" = 'w' && ${#cmd_input} > 1 ]];then
    local target=""
    #目标路径是“其他”
    for ((i=1;i<${#cmd_input};i++))
    do
        #解析输入的字符串，忽视其中的空格，即“w 3 2.t xt”也会处理为“32.txt”
        if [[ ${cmd_input:$i:1} != ' ' ]];then
            target=$target"${cmd_input:$i:1}"
        fi
    done
    #将内容另存为target文件
    printf "$content" > $target
fi
}
#-----
#insert_deal函数，处理insert模式下的命令
#-----
function insert_deal(){
    input="$1"
    #状态机参数flag
    local flag=0
    #始终循环
    while true
    do
        #更新当前位置
        Cal_cur_offset
        case "$input" in
        $'\E')
            #当输入的是esc时
            mode=0
            return 0
            ;;
        ^)
            #当输入是^时，回到行首
            mouse_x=1
```

```
;;
$'\E[3~|$'\177')
#当输入delete和backspace时
if (( cur_offset > 0))&&((offset>0));then
#记录当前的删除位置，并准备进入状态机循环
    delete_mask=$cur_offset
    if (( cur_offset == 1 ));then
        #删除第一个字符
        flag=2
    else
        #删除普通字符
        flag=3
    fi
fi
;;
$'\E[D'
#当向左移动光标
if (($mouse_x > 1));then
#一般情况下直接移动
    ((mouse_x--))
else
#当到达每一行的行首时
    if (($mouse_y == 1));then
        #需要向前翻页时，调用该函数
        Back_n
    elif (($mouse_y>1));then
#当是在屏幕中间的行时
        if (${screen_rowlen[$mouse_y-1]} == $width));then
            #当他的上一行满了时
            if Is_n ;then
                #如果上一行最后一个回车
                mouse_x=$(( ${screen_rowlen[$mouse_y-1]}-1 ))
                #则mouse_x=边界宽减1
                if (($mouse_x==0));then
                    mouse_x=1
                fi
                ((mouse_y--))
            else
                #如果不是回车
                mouse_x=$(( ${screen_rowlen[$mouse_y-1]}))
                #等于边界宽
                if (($mouse_x==0));then
                    mouse_x=1
                fi
                ((mouse_y--))
            fi
        else
#当上一行没满时
            mouse_x=$(( ${screen_rowlen[$mouse_y-1]}-1 ))
            #之间等于边界宽减1
            if (($mouse_x==0));then
                mouse_x=1
            fi
            #y--
        fi
    else
#当上一行没满时
        mouse_x=$(( ${screen_rowlen[$mouse_y-1]}-1 ))
        #之间等于边界宽减1
        if (($mouse_x==0));then
            mouse_x=1
        fi
        #y--
```

```
((mouse_y--))
fi
fi
#保证x始终大于0
if (($mouse_x==0));then
    mouse_x=1
fi
fi
;;
$'\E[B'
#当输入的是下键时
if ((mouse_y<$height-2))&&((mouse_y<row_occupy));then
#判断当前的鼠标位置, 屏幕中间时
((mouse_y++))
if (( $mouse_x >= ${screen_rowlen[$mouse_y]}-1 ));then
#当鼠标的位置大于下一行的长度时
    mouse_x=$(${screen_rowlen[$mouse_y]}-1)
fi
elif ((mouse_y==$height-2));then
#当是最后一行时
num=0
flag=0
#循环判断是否需要换行
for ((i=1;${(offset-end_offset-1)};i++))
do
    ((num++))
    #当遇到回车时直接break, 当num大于一行的长度也要break
    if [[ ${content:$((end_offset+i)):1} = '$\n' ]] || ((num >=
$width));then
        flag=1
        break
    fi
done
#flag=1表示要换行
if ((flag == 1));then
#始终保证begin_offset的正确
    ((begin_offset+=$screen_rowlen[1]))
    mouse_x=1
fi
else
#否则什末都不干
:
fi
#保证mouse_x始终大于0
if (($mouse_x==0));then
    mouse_x=1
fi
;;
$'\E[A'
#当输入的是上键时
if ((mouse_y>1));then
#一般情况下直接y--即可
    ((mouse_y--))
    if (( $mouse_x >= ${screen_rowlen[$mouse_y]}-1 ));then
```

```

#当上一行的长度小于光标位置时
    mouse_x=$(${screen_rowlen[$mouse_y]}-1))
fi
else
#否则要回退和翻页
    Back_n
fi
#保证mouse_x始终大于0
if (($mouse_x==0));then
    mouse_x=1
fi
;;
\$'\E[C'
#计算当前的位移量
Cal_cur_offset
#当前位移量大于或等于总偏移量时，无视本次操作
if (($cur_offset>=$offset));then
    return 0
fi
#当鼠标x坐标小于当前行的最大字母数时
if (($mouse_x < ${screen_rowlen[$mouse_y]}-1));then
    #向右偏移一位
    ((mouse_x++))
else
#当鼠标大于等于最大字母数时
    if (($mouse_y == $height - 2));then
        #当y坐标达到底部时，判断是否需要换行，处理结尾偏移量
        if (($mouse_x == ${screen_rowlen[$mouse_y]}-1));then
            #当下一个字符是回车时
            if [[ ${content:$((cur_offset+1)):1} = '$\n' ]];||[[ ${content:$((cur_offset+1)):1} = '$\r' ]];then
                #保证begin offset始终正确
                ((begin_offset+=${screen_rowlen[1]}))
                mouse_x=1
            else
                #x++
                ((mouse_x++))
            fi
        else
#当到达最大偏移量
            if ((cur_offset == offset - 1));then
                #考虑是否要换行，大于width要换行
                if (($mouse_x < $width));then
                    ((mouse_x++))
                else
#这是要换行时的操作
                    ((begin_offset+=${screen_rowlen[1]}))
                    mouse_x=1
                    #((mouse_y++))
                fi
            else
#保证begin正确
                ((begin_offset+=${screen_rowlen[1]}))
                mouse_x=1
            fi
        else
#保证begin正确
            ((begin_offset+=${screen_rowlen[1]}))
            mouse_x=1
        fi
    fi
fi

```

```
        fi
    fi
else
#当还在屏幕中间时,且达到文本的最后一行
if (( mouse_y == row_occupy));then
#若在屏幕的最后一行,因为最后一行的单词个数不用考虑最后的换行问题

    if (($mouse_x == ${screen_rowlen[$mouse_y]}-1));then
#所以当x坐标是最大位移量-1时直接++
        ((mouse_x++))
    elif (($mouse_x == ${screen_rowlen[$mouse_y]}));then
#否则判断是否另起一行
        if (($mouse_x < $width));then
            ((mouse_x++))
        else
            mouse_x=1
            ((mouse_y++))
        fi
#处理后达到offset位置
fi

else
#当还在屏幕中间时的一般情况
if (($mouse_x == ${screen_rowlen[$mouse_y]}-1));then
    if [[ ${content:$($cur_offset+1)}:1 = '\n' ]];then
#当遇到回车
        mouse_x=1
        ((mouse_y++))
    else
#当一般情况
        ((mouse_x++))
    fi
else
#当到达最大偏移量
    mouse_x=1
    ((mouse_y++))
fi
fi
fi
;;
*)
#当输入其他字符时
if [[ ${input:0:1} = '\n' ]] && ((cur_offset==offset )) &&((mouse_y ==
height-2))
|| ((${screen_rowlen[$mouse_y]}==width)) && ((mouse_y == height-2));then
#当输入的是回车
    ((begin_offset+${screen_rowlen[1]}))
    ((mouse_y--))
fi
#将所输入的字符串入我们的content变量
content=${content:0:$cur_offset}${input:0:1}${content:$cur_offset}
#更新offset参数
offset=$(printf "$content" | wc -m)
```

```
flag=1
;;
esac
#计算偏移量
Cal_cur_offset
if (( $flag==0 ));then
#当是0时，对应一般的插入模式
    remind ""
    display_content $begin_offset
    #更新屏幕并更新鼠标位置
    printf '\033[%d;%dH' $mouse_y $mouse_x
#读取下一个将要输入的内容
    read -sN1 input
    read -sN1 -t 0.0001 k1
    read -sN1 -t 0.0001 k2
    read -sN1 -t 0.0001 k3
    input+="${k1}${k2}${k3}"
elif ((flag==1));then
#当插入一个字符时，执行的实际上是先插入字符，然后向右移动光标
    input=$'\E[C'
    remind ""
    display_content $begin_offset
    #更新屏幕并更新鼠标位置
    printf '\033[%d;%dH' $mouse_y $mouse_x
    #更新flag
    flag=0
elif ((flag==2));then
#更新content
    content=${content:1}
    #鼠标的移动
    mouse_x=1
    mouse_y=1
    remind ""
    display_content $begin_offset
    #更新屏幕并更新鼠标位置
    printf '\033[%d;%dH' $mouse_y $mouse_x
    #更新flag
    flag=0
#读取下一个将要输入的内容
    read -sN1 input
    read -sN1 -t 0.0001 k1
    read -sN1 -t 0.0001 k2
    read -sN1 -t 0.0001 k3
    input+="${k1}${k2}${k3}"
elif ((flag==3));then
#删除普通字符，执行的实际上是向左移动两次，删除一个，再向右移动一次，以适应各种不同的
情况
    input=$'\E[D'
    #模拟一次左移光标
    flag=4
    remind ""
    display_content $begin_offset
    #更新屏幕并更新鼠标位置
    printf '\033[%d;%dH' $mouse_y $mouse_x
```

```
elif ((flag==4));then
    input=$'\E[D'
    #模拟一次左移光标
    flag=5
    remind ""
    display_content $begin_offset
    #更新屏幕并更新鼠标位置
    printf '\033[%d;%dH' $mouse_y $mouse_x
elif ((flag==5));then
#删除并之前的标记数
    content=${content:0:$((delete_mask-1))}${content:$delete_mask}
    #默认右移
    offset=$(printf "$content" | wc -m )
    input=$'\E[C'
    remind ""
    display_content $begin_offset
    #更新屏幕并更新鼠标位置
    printf '\033[%d;%dH' $mouse_y $mouse_x
    flag=0
fi
done
}

#-----
#main函数，程序的生命哦
#-----

function main(){
    #保证输入的参数是一个
    if [ $# -ne 1 ];then
        echo "输入的参数个数有误"
        exit 1
    fi
    #存储输入的参数
    filename="$1"
    #没有这个文件的时候，新建一个这样的文件
    if ! [ -f $filename ];then
        touch $filename
    fi
    #执行初始化函数
    Init
    #计算row和content等参数
    row=$( cat $filename | wc -l)
    content=$(cat $filename)
    #当内容为空，offset置位0
    if [[ $content = "" ]];then
        offset=0
    else
        offset=$(printf "$content" | wc -m )
    fi
    #初始化mode为0
    mode=0
    #打印提示信息
    remind=""
    #初始化begin_offset为0
```

```
begin_offset=0
#打印文本信息
remind "$remind"
display_content $begin_offset
printf '\033[%d;%dH' $mouse_y $mouse_x
#读取输入的内容
while read -sN1 text
do
#读取输入
    read -sN1 -t 0.0001 k1
    read -sN1 -t 0.0001 k2
    read -sN1 -t 0.0001 k3
#拼接字符串
text+="${k1}${k2}${k3}"
case $mode in
0) #noraml mode
    normal_deal "$text"
    #隐藏光标
    printf '\33[?25l'
    #打印文本
    remind ""
    display_content $begin_offset
    #当下一个模式是2时，特殊处理光标的位置
    if ((mode != 2));then
        printf '\033[%d;%dH' $mouse_y $mouse_x
    else
        printf '\033[%d;%dH' $height 5
    fi
    #显示光标
    printf '\33[?25h'
    ;;
1) #insert mode
    insert_deal $text
    #inset处理输入
    remind "$remind"
    display_content $begin_offset
    printf '\033[%d;%dH' $mouse_y $mouse_x
    #打印文本并更新光标位置
    ;;
2) #visual mode
    visual_deal $text
    #visual处理
    remind ""
    display_content $begin_offset
    #更新光标位置，并打印文本
    printf '\033[%d;%dH' $mouse_y $mouse_x
    ;;
esac

done

}
#调用一次main函数
```

```
main $1
```

## 7. 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、test、time、umask。部分命令解释如下：

1. cd ——把当前默认目录改变为。如果没有参数，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。
2. pwd ——显示当前目录。
3. time ——显示当前时间
4. clr ——清屏。
5. dir ——列出目录的内容。
6. set ——列出所有的环境变量。
7. echo ——在屏幕上显示并换行（多个空格和制表符可能被缩减为一个空格）。
8. help ——显示用户手册，并且使用more 命令过滤。
9. exit ——退出shell。

10. shell 的环境变量应该包含shell=/myshell，其中/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=/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。

### 实验设计文档：

#### (1) 设计语言

在实现Unix环境编程方面上，C语言已经有大量可以使用的库和函数，方便直观，而C++不仅完整兼容C，还有大量的STL特性方便程序设计，且C++是我比较熟练的一门语言，综合考虑下我选择了使用C语言Unix环境编程库的C++进行此次MyShell设计。

## (2) 运行过程

shell程序都是读入命令->命令语法分析->执行命令->读入命令这个循环流程进行操作的。

## (3) 功能与模块

本程序主要分为：读取输入模块，分割与语法分析模块，执行命令模块和信号处理模块。

## (4) 读取输入模块

在本程序中，如果在执行时就带有脚本名作为参数，那么程序将直接读入参数文件中的命令并执行，且支持多个文件同时输入，如果你输入了三个文件作为参数，那么该程序就会按照输入的顺序，依次执行三哥文件中包含的shell命令，执行到文件最后退出程序，不会接受用户输入。而如果在调用本程序时没有参数，那么本程序会输出命令提示符等待并提示用户输入，而输出的提示符包含有当前的路径信息。

## (5) 语句分割与分析模块

在本程序中，读取命令是一次读取缓冲区的一行内容，读取后使用strok函数对读入命令进行简单的分割处理，并对一些具有特殊标记的命令，执行不一样的处理策略，以标记该命令的特殊性。例如，当输入的命令，含有&时，就会对指定的命令打上后台运行的标记（一定要在管道符、“&”、“>”“>>”“<”前后保留空格！！！）。

对于使用者输入的命令，解析器将首先按照“;”分割，然后用空格、回车和制表符分割，其中;分割出命令组，而后者分割出具体的命令 对于分割出的命令，我使用 | 对命令再次分割，注意，在使用者输入 | 时，需要在 “|” 之前和之后保留空格，否则将解析命令错误。

在上述模块的作用下，会发生如下行为：假设输入 "ls -l | head -2 | wc -l ; time; echo a",会被解析为：

第一组：“ls -l、head -2、wc -l”（一定要在管道符和“&”前后保留空格！！！）

第二组：time

第三组：echo a

然后依次执行三组命令

## (6) 执行命令模块

拿到一个语法分析结果之后，首先判断拿到的命令数量有多少，对于分割出的每一组命令，依次执行。当一组的命令中只有一个命令时，该条命令一定是和管道无关的，直接运行即可，如果是多条命令，那么是和管道有关的，需要拿出之后再采用其他方式特殊执行，需要特别说明的是，管道的中间文件名称是"MYSELL\_pipe.txt"和"MYSELL\_pipeo.txt"

对于具有重定向或者后台的命令，我们会根据需要创建进程，保证执行的正确性

在创建进程执行命令时，父进程负责对进程表进行操作，而子进程负责执行具体的命令，对于我自己实现的命令，将采用自定义函数直接执行，而对于一些不被我实现的外部命令，我会采用execvp函数执行命令

管道命令的执行，在本程序中的原理也很简单，实际上在实现单条命令的执行后，管道命令相当于对一组命令每一个都单条执行，无非在执行的过程中，增加了重定向的细节处理，相当于将每一条的单条命令都重定向到了我指定的中间文件夹，第一条命令写入，第二条命令再读取，如果是多级管道，假设有x条命令，无非是第1条命令写入，第2~x-1条命令读取并写入，当然这里的读取写入是对两个文件操作后再整合的结果，第x条命令只读取，关于重定向，则采用open、close、dup2、dup这些系统调用组合实现的

需要额外强调的是，当你使用help命令时，会采用more方式读取工作文件夹下的man.txt，所以你需要保证的是，文件夹下存在man.txt，当然我也在下面给出了我的man.txt内容

## (7) 信号处理模块

信号处理模块主要是负责对挂起当前进程的STGSTP信号与子进程结束的SIGCHLD信号做出响应，挂起进程需要更改编程表的状态并且暂停进程，而子进程结束信号也是为了更新进程表，防止僵尸进程的产生，不过需要特别说明的是，CHLD信号在进程再次启动(接受到SIGCONT信号)是也会被触发，这就需要我们对这种情况进行特殊的处理，否则进程刚被CONT信号启动就会被我们的CHLD信号处理函数杀死并移除出进程表，从而导致进程出错，与这个处理有关的两个变量是BG\_FLAG和Z\_FLAG。

## (8) 数据结构与算法

在本次程序中，主要有两个类需要说明，一个类负责管理进程表，一个类负责实现我们的命令容器，需要说明的是，为了使得子进程也能正确的处理进程表，我们将进程表放置在了共享内存这一IPC资源里，而有关的代码被放置在了头文件里。

进程管理相关类：

```
// job类，用于定义不同的工作进程
class job
{
public:
    string name; //进程名称
    int pid; //进程id
    bool fg; //是否为前台程序
    bool run; //是否正在运行
    bool over; //是否已经结束运行
    int bg_id; //后台进程号
};

// process类，用于管理进程
class Process
{
public:
    int sum_jobs; //所有的工作进程数量
    int sum_bgs; //后台运行的进程数量
    job jobs[MAX_JOB_SIZE]; // jobs数组，定义每一个进程的状态
    Process() : sum_jobs(0), sum_bgs(0)
    { //缺省初始化函数
    }
    Process(int sum_cnt_, int bg_cnt_) : sum_jobs(sum_cnt_), sum_bgs(bg_cnt_) //含参的初始化函数
    {
    }
    void Init()
    { //重新初始化赋值
        sum_jobs = 0;
        sum_bgs = 0;
    }
}
```

```

    int Addjob(int pid_, string name_, bool fg_, bool run_, bool over_); //添加一个进程
    Pr_State Delete(int pid_); //删除一个进程
    Pr_State Update(); //更新进程表，并删除所有已经结束的进程
    int FindJobs(int pid_); //查找进程id，失败时返回-1
    int Findbgs(int pid_); //查找后台id，失败时返回-1
};

//全局变量，Pro_manager用于管理所有的进程
Process *Pro_manager;

```

命令容器相关类：

```

// item类。用于定义一个执行命令的容器
class item
{
public:
    vector<string> content; //命令及参数
    bool IsBG; //命令是否为后台运行
    bool pipIn; //命令是否有管道输入
    bool pipOut; //命令是否有管道输出
    bool reIn; //重定向输入标记
    bool reOut; //重定向输出(覆盖)
    bool reApp; //重定向输出(覆盖)
    string InPath; //重定向输入路径
    string OutPath; //重定向输出路径
    item() //初始化函数
    {
        vector<string> tmp;
        IsBG = false;
        pipIn = false;
        pipOut = false;
        reIn = false;
        reOut = false;
        reApp = false;
        InPath = "";
        OutPath = "";
        content = tmp;
    }
    ~item() {} //析构函数
};

//命令的类
class Command
{
public:
    vector<vector<item>> cmd_container; //命令及参数
    Command() {} //构造函数
    Command(vector<vector<item>> &cmd) : cmd_container(cmd) {} //带参数的构造

```

```
void Create(vector<vector<item>> &cmd)
{ //构造后的赋值
    cmd_container = cmd;
}
~Command()
{ //析构函数
    for (auto i : cmd_container)
    {
        for (auto j : i)
        {
            j.~item();
        }
    }
};
}
```

## (9) 项目的运行方式:

想要运行myshell，你需要在linux系统下，编译运行三个文件，可以使用vscode运行该项目。

为了配置vscode的运行环境，你需要配置如下三个文件（这是一种可行的文件，你可以选择使用其他的可行配置）：

1、launch.json:

```
{
    "configurations": [
        {
            "name": "C/C++: g++ 生成和调试活动文件",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}/${fileBasenameNoExtension}",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "为 gdb 启用整齐打印",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "将反汇编风格设置为 Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "C/C++: g++ 生成活动文件",
        }
    ]
}
```

```
        "miDebuggerPath": "/usr/bin/gdb"
    }
],
"version": "2.0.0"
}
```

## 2、settings.json：

```
{
  "files.associations": {
    "string": "cpp",
    "*.tcc": "cpp",
    "compare": "cpp",
    "type_traits": "cpp",
    "iosfwd": "cpp",
    "iostream": "cpp",
    "vector": "cpp",
    "new": "cpp",
    "ostream": "cpp",
    "iomanip": "cpp"
  }
}
```

## 3、tasks.json：

```
{
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: g++ 生成活动文件",
      "command": "/usr/bin/g++",
      "args": [
        "-fdiagnostics-color=always",
        "-g",
        "${file}",
        "-o",
        "${fileDirname}/${fileBasenameNoExtension}"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "detail": "调试器生成的任务。"
    }
  ]
}
```

```
        }
    ],
    "version": "2.0.0"
}
```

当然如果你使用其他的IDE也是可以的，如果你为了方便起见，也可以直接在命令行下运行。

```
g++ -o <你想要的名字> main.cpp
```

当然这需要你提前有g++的编译器，以及main.cpp Command.cpp 和Myshell.h在同一个文件夹中，以便文件相互能找到彼此的位置

下面是我的代码部分：

```
/*
 * @Author: 韩恺荣
 * @Date: 2022-08-13 10:06:18
 * @LastEditTime: 2022-08-13 11:00:40
 * @LastEditors: 韩恺荣
 * @Description:文件名: main.cpp, 实现myshell的主体架构
 */

#include "Myshell.h"
#include "Command.cpp"

//定义debug模式, 当置位1时启动并帮助作者debug
#define _DEBUG_ 0

//输入函数的解析器
void Prase(string &tmp, Command *ret_com)
{
    //首先对输入的参数中含有的;进行分割, 将命令分割为命令组
    char delim[5];
    strcpy(delim, ";");
    vector<char *> container;
    //调用strtok函数分割
    char *tmp_ch = strtok(const_cast<char *>(tmp.c_str()), delim);
    while (tmp_ch != nullptr)
    {
        container.push_back(tmp_ch);
        tmp_ch = strtok(NULL, delim);
    }
    //再使用空格/制表符和回车对命令分割
    strcpy(delim, "\n\t");
    vector<vector<string>> cmd_delim;
    for (auto i : container)
    {
        vector<string> tmp_str;
        tmp_ch = strtok(i, delim);
```

```
//一直分割直到字符串结束
while (tmp_ch != nullptr)
{
    tmp_str.push_back(tmp_ch);
    tmp_ch = strtok(NULL, delim);
}
//将分割好的字符串，进入容器
if (!tmp_str.empty())
{
    cmd_delim.push_back(tmp_str);
}
}
//准备将要返回的命令容器
vector<vector<item>> ret;
//遍历每一个字符串
for (auto i : cmd_delim)
{
    //每一个命令组对应一个vector
    vector<item> tmp_item;
    item *tmp_ele = new item();
    int conunt = 0;
    int inpath_flag = 0;
    int outpath_flag = 0;
    //遍历每一个string
    for (auto j : i)
    {
        if (inpath_flag)
        { //当要获取输入的路径时
            inpath_flag = 0;
            tmp_ele->InPath = j;
        }
        else if (outpath_flag)
        { //当要获取输出的路径时
            outpath_flag = 0;
            tmp_ele->OutPath = j;
        }
        else
        {
            if (j == "<")
            { //当要重新向输入时
                tmp_ele->reIn = true;
                inpath_flag = 1;
            }
            else if (j == ">")
            { //当重定向输出时
                tmp_ele->reOut = true;
                outpath_flag = 1;
            }
            else if (j == ">>")
            { //当重定向输出时
                tmp_ele->reApp = true;
                outpath_flag = 1;
            }
            else if (j == "|")
            {
                if (inpath_flag)
                    tmp_ele->reIn = true;
                if (outpath_flag)
                    tmp_ele->reOut = true;
                if (outpath_flag)
                    tmp_ele->reApp = true;
            }
        }
    }
    tmp_item.push_back(*tmp_ele);
    tmp_ele = new item();
    tmp_ele->reIn = false;
    tmp_ele->reOut = false;
    tmp_ele->reApp = false;
    tmp_ele->InPath = "";
    tmp_ele->OutPath = "";
    tmp_ele->AppPath = "";
}
ret.push_back(tmp_item);
}
```

```
{ //当有管道操作时
    tmp_ele->pipOut = true;
    tmp_item.push_back(*tmp_ele);
    tmp_ele = new item(); //新建下一个命令容器
    // tmp_ele->content.clear();
    tmp_ele->pipIn = true;
}
else if (j == "&")
{ //当有后台运行标志时
    tmp_ele->IsBG = true;
}
else
{
    tmp_ele->content.push_back(j);
}
}

//将切割的命令装入容器
tmp_item.push_back(*tmp_ele);
if (!tmp_item.empty())
{
    ret.push_back(tmp_item);
}
}

//用容器初始化命令
ret_com->Create(ret);
}

//查看，命令是否来自于我们自己实现的命令
bool is_in_cmd(string cmd_name)
{
    return cmd_name == "umask" ||
           cmd_name == "cd" ||
           cmd_name == "exec" ||
           cmd_name == "jobs" ||
           cmd_name == "fg" ||
           cmd_name == "clr" ||
           cmd_name == "bg" ||
           cmd_name == "exit" ||
           cmd_name == "set" ||
           cmd_name == "dir" ||
           cmd_name == "pwd" ||
           cmd_name == "test" ||
           cmd_name == "echo" ||
           cmd_name == "time";
}

//运行我们自己实现的命令
bool run_in(item &tmp)
{
    //如果不在列表中，我们返回false
    if (!is_in_cmd(tmp.content[0]))
    {
        return false;
    }
}
```

```
//否则找打对应的函数并执行函数
if (tmp.content[0] == "bg")
    BG(tmp);
else if (tmp.content[0] == "fg")
    FG(tmp);
else if (tmp.content[0] == "jobs")
    JOBS(tmp);
else if (tmp.content[0] == "cd")
    CD(tmp);
else if (tmp.content[0] == "pwd")
    PWD();
else if (tmp.content[0] == "clr")
    CLR();
else if (tmp.content[0] == "time")
    TIME();
else if (tmp.content[0] == "umask")
    UMASK(tmp);
else if (tmp.content[0] == "dir")
    dir(tmp);
else if (tmp.content[0] == "set")
    SET();
else if (tmp.content[0] == "echo")
    ECHO(tmp);
else if (tmp.content[0] == "exec")
    EXEC(tmp);
else if (tmp.content[0] == "test")
    TEST(tmp);
else if (tmp.content[0] == "exit")
    EFUN(tmp);
return true;
}

/**
 * @description: 命令执行器
 * @param {Command} *cur_cmd 输入将要执行的命令
 * @return {*}
 */
void ExecuteCommand(Command *cur_cmd)
{
    for (auto i : cur_cmd->cmd_container)
    {
        //获取每一个命令组
        if (i.size() == 1)
        {
            //如果只有一个命令，证明和管道运算符无关，直接执行即可
            if (is_in_cmd(i[0].content[0]))
            {
                run_in(i[0]);
            }
            else
            {
                //当输入的是help命令时，特殊处理，我们用more的方式预览我们事先提供好滴
                if (i[0].content[0] == "help")

```

```
{  
    i[0].content.clear();  
    i[0].content.push_back("more");  
    //对命令修改和切割，得到我们想要的命令  
    string path_tmp = getcwd(nullptr, 0);  
    path_tmp += "/man.txt";  
    i[0].content.push_back(path_tmp);  
}  
//新建进程  
pid_t pid = fork();  
//进程创建失败时  
if (pid < 0)  
{  
    fprintf(stderr, "create process failed\n");  
    exit(1);  
}  
else if (pid == 0)  
{  
// debug模式下输出一些有用的信息帮助我们debug  
#if _DEBUG_ == 1  
    cout << getpid() << endl;  
    cout << Pro_manager->FindJobs(getpid()) << endl;  
    char *arg[i[0].content.size()];  
    int cnt = 0;  
    for (auto j : i[0].content)  
    {  
        arg[cnt++] = (char *)j.c_str();  
    }  
    arg[cnt] = nullptr;  
    cout << 2 << endl;  
    if (!execvp(i[0].content[0].c_str(), arg))  
        fprintf(stderr, "MYSHELL:NOT FOUND COMMAND");  
#endif  
    //等待父进程对进程列表操作完成后，再执行我们的操作  
    usleep(1);  
    int cnt = 0;  
    while (true)  
    {  
        if (Pro_manager->FindJobs(getpid()) == -1)  
        { //没有找打，证明还未完成进程列表的添加  
            usleep(1);  
        }  
        else  
        {  
            //找到之后，并且发现找到的进程状态被修改成为true，开始执行  
命令  
            if (Pro_manager->jobs[Pro_manager->FindJobs(getpid())].run)  
                break;  
        }  
        usleep(1);  
    }  
    //当有重定向时  
    if (i[0].reIn)
```

```
{  
    //关闭标准输入  
    close(0);  
    int Fd = open(i[0].InPath.c_str(), O_RDONLY);  
}  
//重定向输出时  
if (i[0].reOut)  
{  
    //关闭标准输出  
    close(1);  
    int Fd = open(i[0].OutPath.c_str(), O_WRONLY | O_CREAT |  
O_TRUNC, 0666);  
}  
//重定向追加  
if (i[0].reApp)  
{  
    //关闭标准输出  
    close(1);  
    int Fd = open(i[0].OutPath.c_str(), O_RDWR | O_CREAT |  
O_APPEND, 0666);  
}  
//执行命令  
if (!is_in_cmd(i[0].content[0]))  
{  
    //要执行的命令不在我们的列表中，调用execvp方法  
    // cout<<"size:"<<i[0].content.size()<<endl;  
    char *arg[i[0].content.size()];  
    int cnt = 0;  
    //对命令进行切割，以满足execvp方法对命令的要求  
    for (auto j : i[0].content)  
    {  
        arg[cnt++] = new char[strlen(const_cast<char *>  
(j.c_str()))];  
        strcpy(arg[cnt - 1], const_cast<char *>(j.c_str()));  
    }  
    cout << cnt - 1 << " " << j << " " << arg[cnt - 1] <<  
    endl;  
    #endif  
    #if _DEBUG_ == 1  
        cout << i[0].content[0].c_str() << endl;  
        for (int k = 0; k < cnt; k++)  
        {  
            cout << k << " " << arg[k] << endl;  
        }  
    #endif  
    //执行失败，证明命令不存在  
    if (!execvp(i[0].content[0].c_str(), arg))  
        fprintf(stderr, "MYSHELL:NOT FOUND COMMAND");  
}  
else  
{  
    //否则，简单调用我们的命令执行函数  
}
```

```
        run_in(i[0]);
    }
    //恢复我们之前的输入和输出到标准流
    close(0);
    close(1);
}
else
{
    //父进程负责管理进程表的添加
    int index;
    if (!i[0].IsBG)
    {
        //当是后台进程时
        index = Pro_manager->Addjob(pid, i[0].content[0], true,
false, false);
    }
    else
    {
        //当是前台进程时
        index = Pro_manager->Addjob(pid, i[0].content[0], false,
false, false);
    }
    //获取当前所在的文件夹
    string rootpath = getcwd(NULL, 0);
    //添加环境变量
    setenv("PARENT", rootpath.c_str(), 1);
    Pro_manager->jobs[index].run = true;
    //如果不是后台命令
    if (!i[0].IsBG)
    {
        //使用阻塞的方式等待命令的结束
        waitpid(pid, NULL, WUNTRACED);
        //执行完成后，修改我们的进程表
        if (!Pro_manager->jobs[index].run && Pro_manager-
>jobs[index].fg)
            Pro_manager->Delete(pid);
    }
    else
    {
        //后台命令使用非阻塞的方式执行
        waitpid(pid, NULL, WNOHANG);
    }
}
}
else if (i.size() > 1)
{
    //当要执行的命令个数超过一个时，证明有管道
    int cnt_outer = 0;
    string pipe_txt = "MYSHELL_pipe.txt";
    //指定管道命令的中间文件
    for (auto cur_cmd : i)
    {
        //对于每一个命令，更改输入和输出流
    }
}
```

```
int fd_o, fd_i, store_o, store_i;
//当是中间的命令时，输入输出都要改变
if (cnt_outer != 0 && cnt_outer != i.size() - 1)
{
    //打开新文件作为输入和输出流
    fd_i = open(pipe_txt.c_str(), O_RDONLY);
    fd_o = open("MYSHELL_pipeo.txt", O_WRONLY | O_CREAT | O_TRUNC
, 0777);
    //保存原来的输入和输出
    store_i = dup(0);
    store_o = dup(1);
    //更改输入和输出
    dup2(fd_o, 1);
    dup2(fd_i, 0);
}
else if (cnt_outer == 0)
{
    //对于第一个命令只更改输出流
    fd_o = open(pipe_txt.c_str(), O_WRONLY | O_CREAT | O_TRUNC,
0666);
    store_o = dup(1);
    dup2(fd_o, 1);
}
else
{
    //最后一个命令只更改输入流
    fd_i = open(pipe_txt.c_str(), O_RDONLY);
    store_i = dup(0);
    dup2(fd_i, 0);
}
//当是列表中自定义函数时，直接执行
if (is_in_cmd(cur_cmd.content[0]))
{
    run_in(cur_cmd);
}
else
{
    //当时help命令
    if (cur_cmd.content[0] == "help")
    {
        cur_cmd.content.clear();
        cur_cmd.content.push_back("more");
        //修改help命令的参数为我自创的手册路径
        string path_tmp = getcwd(nullptr, 0);
        path_tmp += "/man.txt";
        cur_cmd.content.push_back(path_tmp);
    }
    //新建一个命令
    pid_t pid = fork();
    if (pid < 0)
    {
        //新建失败
        fprintf(stderr, "create process failed\n");
        exit(1);
    }
}
```

```
        }
        //子进程负责执行命令，和单条命令类似
        else if (pid == 0)
        {
            // debug模式下的输出一些调试中有用的信息
#if _DEBUG_ == 1
            cout << getpid() << endl;
            cout << Pro_manager->FindJobs(getpid()) << endl;
            char *arg[cur_cmd.content.size()];
            int cnt = 0;
            for (auto j : cur_cmd.content)
            {
                arg[cnt++] = (char *)j.c_str();
            }
            arg[cnt] = nullptr;
            cout << 2 << endl;
            if (!execvp(cur_cmd.content[0].c_str(), arg))
                fprintf(stderr, "MYSHELL:NOT FOUND COMMAND");
#endif
            usleep(1);
            //等待父进程执行进程表插入完毕
            int cnt = 0;
            while (true)
            {
                if (Pro_manager->FindJobs(getpid()) == -1)
                {
                    //等待插入完毕
                    usleep(1);
                }
                else
                {
                    if (Pro_manager->jobs[Pro_manager->FindJobs(getpid())].run)
                        break;
                }
                usleep(1);
            }
            //当重定向输入
            if (cur_cmd.reIn)
            {
                //更改标准输入流
                close(0);
                int Fd = open(cur_cmd.InPath.c_str(), O_RDONLY);
            }
            //重定向输出时，更改标准输出流
            if (cur_cmd.reOut)
            {
                close(1);
                int Fd = open(cur_cmd.OutPath.c_str(), O_WRONLY |
O_CREAT | O_TRUNC, 0666);
            }
            //重定向追加
            if (cur_cmd.reApp)
            {
```

```
        close(1);
        int Fd = open(cur_cmd.OutPath.c_str(), O_RDWR |
O_CREAT | O_APPEND, 0666);
    }
    //当是自定义命令的执行时
    if (!is_in_cmd(cur_cmd.content[0]))
    {
        // cout<<"size:"<<cur_cmd.content.size()<<endl;
        //更改格式以便于适配execvp方法的执行
        char *arg[cur_cmd.content.size()];
        int cnt = 0;
        for (auto j : cur_cmd.content)
        {
            //将命令重新转化为字符串
            arg[cnt++] = new char[strlen(const_cast<char *>
(j.c_str()))];
            strcpy(arg[cnt - 1], const_cast<char *>
(j.c_str()));
        }
        #if _DEBUG_ == 1
            cout << cnt - 1 << " " << j << " " << arg[cnt - 1]
<< endl;
        #endif
    }
    arg[cnt] = nullptr;
    #if _DEBUG_ == 1
        cout << cur_cmd.content[0].c_str() << endl;
        for (int k = 0; k < cnt; k++)
        {
            cout << k << " " << arg[k] << endl;
        }
    #endif
    if (!execvp(cur_cmd.content[0].c_str(), arg)) //执行失败时，输出报错信息
        fprintf(stderr, "MYSHELL:NOT FOUND COMMAND");
    }
    else
    {
        //当时自定义命令，直接执行就好
        run_in(cur_cmd);
    }
}
else
{
    //父进程负责进程表的添加和控制
    int index;
    if (!cur_cmd.IsBG)
    {
        //当是后台进程是
        index = Pro_manager->Addjob(pid, cur_cmd.content[0],
true, false, false);
    }
    else
    {
        //当不是后台进程
    }
}
```

```
        index = Pro_manager->Addjob(pid, cur_cmd.content[0],
false, false, false);
    }
    //添加父进程路径到环境变量
    string rootpath = getcwd(NULL, 0);
    setenv("PARENT", rootpath.c_str(), 1);
    Pro_manager->jobs[index].run = true;
    //当不是后台命令
    if (!cur_cmd.IsBG)
    {
        //阻塞等待
        waitpid(pid, NULL, WUNTRACED);
        //删除进程表中的数据
        if (!Pro_manager->jobs[index].run && Pro_manager-
>jobs[index].fg)
            Pro_manager->Delete(pid);
    }
    else
    {
        //后台命令我们非阻塞
        waitpid(pid, NULL, WNOHANG);
    }
}
//恢复标准输出和输入流
if (cnt_outer != 0 && cnt_outer != i.size() - 1)
{
    //用保存好的输入和输出再次更改我们的输入和输出流
    dup2(store_i, 0);
    dup2(store_o, 1);
    close(fd_i);
    close(fd_o);
    fd_i = open(pipe_txt.c_str(), O_WRONLY | O_CREAT | O_TRUNC ,
0777);
    fd_o = open("MYSHELL_pipeo.txt", O_RDONLY);
    char c;
    while(read(fd_o,&c,1) == 1)
        write(fd_i,&c,1);
    close(fd_i);
    close(fd_o);
    remove("MYSHELL_pipeo.txt");
    cnt_outer++;
}
else if (cnt_outer == 0)
{
    //同理，恢复输出流
    dup2(store_o, 1);
    close(fd_o);
    cnt_outer++;
}
else
{
    //同理，恢复输入流
    dup2(store_i, 0);
```

```
        close(fd_i);
        remove("MYSHELL_pipe.txt");
        cnt_outer++;
    }
}
}
}
return;
}
/***
 * @description: main函数, 负责执行我们的命令
 */
int main(int argc, char **argv)
{
    Init(); //初始化
    //显示当前路径
    string cur_path = getcwd(NULL, 0);
    setenv("SHELL", cur_path.c_str(), 1);
    //当输入多个参数, 证明要执行文件
    if (argc != 1)
    {
        //执行文件, 有多个的话循环依次执行
        for (int i = 1; i < argc; i++)
        {
            ifstream fin(argv[i]);
            string tmp;
            //每次读取一行
            while (getline(fin, tmp))
            {
                //显示提示信息
                cur_path = getcwd(NULL, 0);
                cout << "ROOT@" << cur_path << "$" << tmp << endl;
                Command *cur_cmd = new Command;
                //解析后执行
                Prase(tmp, cur_cmd);
                ExecuteCommand(cur_cmd);
            }
        }
        return 0;
    }
    //当从控制台输入时
    string tmp;
    cur_path = getcwd(NULL, 0);
    //打印提示信息
    cout << "ROOT@" << cur_path << "$";
    getline(cin, tmp);
    while (true)
    {
        //循环读取
        Command *cur_cmd = new Command;
        //解析后执行命令
        Prase(tmp, cur_cmd);
        ExecuteCommand(cur_cmd);
        cur_path = getcwd(NULL, 0);
    }
}
```

```
        cout << "ROOT@" << cur_path << "$";
        getline(cin, tmp);
    }
}

/***
 * @description: 处理CHLD信号
 * @param {int} sign 信号的辨识标志
 * @param {siginfo_t} *catcher 捕获信号额外的信息
 * @param {void} *pointer
 * @return {*}
*/
void Handler_S(int sign, siginfo_t *catcher, void *pointer)
{
#if _DEBUG_ == 1
    cout << catcher->si_pid << " over" << endl;
#endif
    //使用flag以避免错误的删除
    if (BG_FLAG)
    {
        BG_FLAG = 0;
        return;
    }
    else if (Z_FLAG)
    {
        Z_FLAG = 0;
        return;
    }
    //确认捕获的信号为CHLD
    if (sign == SIGCHLD)
    {
        //找不到进程表中的对应命令
        int index = Pro_manager->FindJobs(catcher->si_pid);
        if (index == -1)
        {
            cout << "fatal error!" << endl;
            exit(1);
        }
        else
        {
            //找到后, 当是前台命令, 证明这个命令执行完毕
            if (Pro_manager->jobs[index].fg)
            {
                //打印信息
                //cout << "[" << Pro_manager->jobs[index].pid << "]" <<
Pro_manager->jobs[index].name << " 已完成" << endl;
                //修改进程表
                Pro_manager->jobs[index].run = false;
                Pro_manager->jobs[index].over = true;
            }
            else if (!Pro_manager->jobs[index].fg && Pro_manager->jobs[index].run)
            {
                //当是后台命令已完成
                cout << "[" << Pro_manager->jobs[index].pid << "]" << Pro_manager-
>jobs[index].name << " 已完成" << endl;
            }
        }
    }
}
```

```

        Pro_manager->Delete(Pro_manager->jobs[index].pid);
        //确保命令完成
        kill(Pro_manager->jobs[index].pid, 9);
        Pro_manager->Delete(Pro_manager->jobs[index].pid);
    }
}
}

/**
 * @description: 处理挂起命令
 * @param {int} sign 信号标志
 * @return {*}
 */
void Handler_Z(int sign)
{
    //确认是挂起信号
    if (sign == SIGTSTP || sign == SIGSTOP)
    {
#ifndef _DEBUG_
        Output();
#endif
        // cout<<"1"<<endl<<Pro_manager->jobs[Pro_manager->sum_jobs-1].run<<Pro_manager->jobs[Pro_manager->sum_jobs-1].fg<<endl;
        //保证将要处理的命令是一个正在运行的命令, 找到最近的正在运行命令, 执行如下操作
        if (Pro_manager->jobs[Pro_manager->sum_jobs - 1].run && Pro_manager->jobs[Pro_manager->sum_jobs - 1].fg && Pro_manager->jobs[Pro_manager->sum_jobs - 1].pid != 0)
        {
            //在对命令挂起前, 首先处理run和flag标志
            Z_FLAG = 1;
            Pro_manager->jobs[Pro_manager->sum_jobs - 1].run = false;
            Pro_manager->jobs[Pro_manager->sum_jobs - 1].fg = false;
            //发送挂起信号
            kill(Pro_manager->jobs[Pro_manager->sum_jobs - 1].pid, SIGTSTP);
            Pro_manager->jobs[Pro_manager->sum_jobs - 1].bg_id = (++Pro_manager->sum_bgs);
            Pro_manager->jobs[Pro_manager->sum_jobs - 1].over = false;
            //输出提示信息
            cout << endl;
            cout << "[" << Pro_manager->jobs[Pro_manager->sum_jobs - 1].bg_id << "]"
                << Pro_manager->jobs[Pro_manager->sum_jobs - 1].pid << " + is already stop" << endl;
        }
    }
}

```

## Command.cpp

```

/**
 * @Author: 韩恺荣
 * @Date: 2022-08-13 10:07:16
 * @LastEditTime: 2022-08-13 11:52:47

```

```
* @LastEditors: 韩恺荣
* @Description: 文件名: Command.cpp负责命令的实现
*/
#include "Myshell.h"
/**
 * @description: 将字符串转正整数, 失败时返回-1
 * @param {string} &a
 * @return {*} 返回转换的数
 */
int TurnToInt(string &a)
{
    int sum = 0;
    for (int i = 0; i < a.size(); i++)
    {
        //遍历字符串的每一个元素并转换
        if (a[i] >= '0' && a[i] <= '9')
        {
            sum *= 10;
            sum += (int)(a[i] - '0');
        }
        else
        {
            //失败返回-1
            return -1;
        }
    }
    return sum;
}
/**
 * @description: 将挂起的命令转到后台运行, 支持bg %n和不带参两种模式
 * @param {item} &cmd
 * @return {*}
 */
void BG(item &cmd)
{
    if (cmd.content.size() == 1)
    {
        //当只有一个参数时, 找到最后的后台挂起命令
        int i;
        for (i = 0; i < Pro_manager->sum_jobs; i++)
        {
            if (!Pro_manager->jobs[i].fg && !Pro_manager->jobs[i].run &&
!Pro_manager->jobs[i].over)
            {
                BG_FLAG = 1;
                //向进程发生继续信号
                kill(Pro_manager->jobs[i].pid, SIGCONT);
                Pro_manager->jobs[i].fg = false;
                Pro_manager->jobs[i].run = true;
                //打印提示信息
                cout << "[" << Pro_manager->jobs[i].bg_id << "]" << Pro_manager-
>jobs[i].name << " 转至后台运行" << endl;
                break;
            }
        }
    }
}
```

```
        }
        if (i == Pro_manager->sum_jobs)
        {
            cout << "There is no process to deal" << endl;
            return;
        }
    }
else
{
    //当输出多个参数时，确保格式正确
    if (cmd.content[1][0] == '%')
    {
        int id;
        string tmp = cmd.content[1].substr(1);
        if ((id = TurnToInt(tmp)) != -1)
        {
            //转换字符串为后台进程号
            int index = Pro_manager->Findbgs(id);
            if (index != -1)
            {
                //当找到时
                BG_FLAG = 1;
                //发送进程继续的信号
                kill(Pro_manager->jobs[index].pid, SIGCONT);
                Pro_manager->jobs[index].fg = false;
                Pro_manager->jobs[index].run = true;
                //打印提示信息
                cout << "[" << Pro_manager->jobs[index].bg_id << "]"
Pro_manager->jobs[index].name << " 转至后台运行" << endl;
            }
            else
            {
                cout << "MYSHELL:Not Found target" << endl;
            }
        }
        else
        {
            cout << "illegal input" << endl;
        }
    }
    else
    {
        cout << "Format is not allowed" << endl;
    }
}
}

/**
 * @description: 打印当前后台运行和挂起的进程，支持-1参数选项
 * @param {item} &cmd
 * @return {*}
 */
void JOBS(item &cmd)
{
    int cnt = 0;
```

```
if (cmd.content.size() == 1)
{
    //只输入一个参数
    for (; cnt < Pro_manager->sum_jobs; cnt++)
    {
        //遍历并打印
        if (!Pro_manager->jobs[cnt].fg && !Pro_manager->jobs[cnt].over)
        {
            //对于后台未over的进程
            cout << setw(5) << "[" << Pro_manager->jobs[cnt].bg_id << "]";
            if (Pro_manager->jobs[cnt].run)
            {
                //当该进程运行时
                cout << setw(3) << " " << setw(3) << Pro_manager-
>jobs[cnt].name << " 正在运行" << endl;
            }
            else
            {
                //当是挂起的进程时
                cout << setw(3) << "+" << setw(3) << Pro_manager-
>jobs[cnt].name << " 已挂起" << endl;
            }
        }
    }
}
else if (cmd.content.size() == 2)
{
    if (cmd.content[1] == "-1")
    {
        //对于-1参数的选项
        for (; cnt < Pro_manager->sum_jobs; cnt++)
        {
            //遍历进程表
            if (!Pro_manager->jobs[cnt].fg && !Pro_manager->jobs[cnt].over)
            {
                //打印详细的进程信息
                cout << setw(5) << "[" << Pro_manager->jobs[cnt].bg_id << "]"
<< Pro_manager->jobs[cnt].pid;
                if (Pro_manager->jobs[cnt].run)
                {
                    //运行的
                    cout << setw(3) << " " << setw(3) << Pro_manager-
>jobs[cnt].name << " 正在运行" << endl;
                }
                else
                {
                    //挂起的
                    cout << setw(3) << "+" << setw(3) << Pro_manager-
>jobs[cnt].name << " 已挂起" << endl;
                }
            }
        }
    }
}
```

```
        {
            cout << "MYSHELL: JOBS :invalid input" << endl;
        }
    }
else
{
    cout << "MYSHELL: JOBS :invalid input" << endl;
}
}

/**
 * @description: 将后台或挂起的进程转到前台，支持fg %n和不带参两种模式
 * @param {item} &cmd
 * @return {*}
 */
void FG(item &cmd)
{
    int cnt = Pro_manager->sum_jobs - 1;
    if (cmd.content.size() == 1)
    {
        //不含参数时，默认操作对象是最进的后台进程
        for (; cnt > 0; cnt--)
        {
            if (!Pro_manager->jobs[cnt].fg && !Pro_manager->jobs[cnt].over)
            {
                break;
            }
        }
        //没有找到后台进程
        if (cnt == 0)
        {
            cout << "MYSHELL : NOT BACKGROUND PROCESS" << endl;
        }
        else
        {
            //找到之后
            for (int i = cnt + 1; i < Pro_manager->sum_jobs - 1; i++)
            {
                //修改后台id总数
                if (!Pro_manager->jobs[i].fg)
                {
                    Pro_manager->jobs[i].bg_id--;
                }
            }
            //如果正在运行
            if (Pro_manager->jobs[cnt].run)
            {
                //转前台并阻塞等待
                Pro_manager->jobs[cnt].fg = true;
                waitpid(Pro_manager->jobs[cnt].pid, NULL, 0);
            }
            else
            {
                //如果挂起，转前台
                Pro_manager->jobs[cnt].fg = true;
            }
        }
    }
}
```

```
    Pro_manager->jobs[cnt].run = true;
    Pro_manager->jobs[cnt].over = false;
    BG_FLAG = 1;
    //阻塞运行，并发送继续的信号
    kill(Pro_manager->jobs[cnt].pid, SIGCONT);
    waitpid(Pro_manager->jobs[cnt].pid, NULL, 0);
}
}
}
else if (cmd.content.size() == 2)
{
    //输入多个参数
    if (cmd.content[1][0] == '%')
    {
        int id;
        string tmp = cmd.content[1].substr(1);
        if ((id = TurnToInt(tmp)) != -1)
        {
            //获取后台id
            int index = Pro_manager->Findbgs(id);
            if (index != -1)
            {
                //如果找到进程
                for (int i = index + 1; i < Pro_manager->sum_jobs - 1; i++)
                {
                    //修改后台id总数
                    if (!Pro_manager->jobs[i].fg)
                    {
                        Pro_manager->jobs[i].bg_id--;
                    }
                }
                //后台正在运行的进程
                if (Pro_manager->jobs[index].run)
                {
                    //改为前台并阻塞和运行
                    Pro_manager->jobs[index].fg = true;
                    waitpid(Pro_manager->jobs[index].pid, NULL, 0);
                }
                else
                {
                    //挂起的进程
                    Pro_manager->jobs[index].fg = true;
                    Pro_manager->jobs[index].run = true;
                    Pro_manager->jobs[index].over = false;
                    BG_FLAG = 1;
                    //发送继续的信号，并运行
                    kill(Pro_manager->jobs[index].pid, SIGCONT);
                    waitpid(Pro_manager->jobs[index].pid, NULL, 0);
                }
            }
            else
            {
                cout << "MYSHELL:Not Found target" << endl;
            }
        }
    }
}
```

```
        }
    else
    {
        cout << "illegal input" << endl;
    }
}
else
{
    cout << "MYSHELL: JOBS :invalid input" << endl;
}
//更新进程
Pro_manager->Update();
}
/**
 * @description: 输出当前文件夹
 * @return {*}
 */
void PWD()
{
    cout << getcwd(NULL, 0) << endl;
}
/**
 * @description: cd命令, 不带参简单输出当前文件夹, 带参时转移当前文件夹
 * @param {item} &cmd
 * @return {*}
 */
void CD(item &cmd)
{
    //不带参
    if (cmd.content.size() == 1)
        PWD();
    else if (cmd.content.size() == 2)
    {
        //带参时
        if (chdir(cmd.content[1].c_str()))
            fprintf(stderr, "Error:没有%s此目录!\n", cmd.content[1].c_str()); //改变目录失败, 没有此目录
        else
        {
            setenv("PWD", cmd.content[1].c_str(), 1); // CD指令要改变PWD环境变量
        }
    }
    else
        fprintf(stderr, "Error:参数过多\n");
}
/**
 * @description: 清屏
 * @return {*}
 */
```

```
void CLR()
{
    cout << "\e[1;1H\e[2J";
}
/***
 * @description: 输出当前时间
 * @return {*}
 */
void TIME()
{
    time_t now_time = time(NULL);
    tm *t_tm = localtime(&now_time);
    printf("%s\n", asctime(t_tm));
}
/***
 * @description: 改变掩码
 * @param {item} &cmd
 * @return {*}
 */
void UMASK(item &cmd)
{
    if (cmd.content.size() == 1)
    {
        //不带参输出当前掩码
        unsigned int old;
        old = umask(0); //改变掩码以获取当前掩码
        umask(old); //再改回去
        cout << "UMASK:" << old << endl;
    }
    else if (cmd.content.size() == 2)
    {
        //当输入多个参数
        if (TurnToInt(cmd.content[1]) != -1)
        {
            //简单改变掩码即可
            unsigned int tmp = stoi(cmd.content[1], nullptr, 8);
            umask(tmp);
        }
        else
            fprintf(stderr, "Error:不合法的输入\n");
    }
    else
    {
        //参数过多
        fprintf(stderr, "Error:参数过多\n");
    }
}
/***
 * @description: 输出dir信息，支持带参和不带参，且支持-A, -l, -a参数，但是这些选项不能同时使用
 * @param {item} &cmd
 * @return {*}
 */
void dir(item &cmd)
```

```
{  
    string option;  
    if (cmd.content.size() == 1)  
    {  
        //不带参，简单输出当前文件夹  
        cmd.content.push_back(getcwd(nullptr, 0));  
    }  
    else if (cmd.content.size() == 3)  
    {  
        //带参，将参数提取出来  
        option = cmd.content[1];  
        cmd.content[1] = cmd.content[2];  
    }  
    else if (cmd.content.size() == 2)  
    ;  
    else  
    {  
        fprintf(stderr, "Error:参数过多\n");  
        return;  
    }  
    //获取dir信息  
    DIR *dp = opendir(cmd.content[1].c_str());  
    if (dp == nullptr)  
    {  
        fprintf(stderr, "打开%s文件夹失败\n", cmd.content[1].c_str());  
    }  
    //提取信息  
    dirent *info;  
    try  
    {  
        if (cmd.content.size() == 2 || cmd.content.size() == 1)  
        {  
            //不带参数时  
            while ((info = readdir(dp)) != nullptr)  
            {  
                //不输出隐藏文件  
                if (info->d_name[0] != '.')  
                    cout << setw(10) << setiosflags(ios::left) << info->d_name;  
            }  
            cout << endl;  
        }  
        else  
        {  
            //带参数选项  
            while ((info = readdir(dp)) != nullptr)  
            {  
                if (option == "-A")  
                {  
                    //输出隐含但不包括'.','..'这两个文件  
                    if (info->d_name != "." || info->d_name != "..")  
                        cout << setw(10) << setiosflags(ios::left) << info->d_name;  
                }  
                else if (option == "-a")  
                {  
                    //输出所有文件  
                    cout << setw(10) << setiosflags(ios::left) << info->d_name;  
                }  
            }  
        }  
    }  
}
```

```
{  
    //输出隐含文件  
    cout << setw(10) << setiosflags(ios::left) << info->d_name;  
}  
else if (option == "-l")  
{  
    //长列表形式输出  
    cout << setw(8) << setiosflags(ios::left) << info->d_ino  
        << setw(10) << setiosflags(ios::left) << info->d_name  
        << setw(8) << setiosflags(ios::left) << info->d_type <<  
endl;  
}  
else  
{  
    //无用的参数，默认为一般输出  
    if (info->d_name[0] != '.')  
        cout << setw(10) << setiosflags(ios::left) << info-  
>d_name;  
}  
}  
}  
if (option != "-l")  
    cout << endl;  
}  
}  
catch (...)  
{  
    //处理未知异常  
    cout << "unexcepted error" << endl;  
}  
}  
}***  
* @description: 打印环境变量  
* @return {*}  
*/  
void SET()  
{  
    extern char **environ; //直接获取环境变量  
    for (int i = 0; environ[i]; i++)  
        cout << environ[i] << endl;  
}  
}***  
* @description: 打印命令  
* @param {item} &cmd  
* @return {*}  
*/  
void ECHO(item &cmd)  
{  
    //输入什末打印什末  
    for (auto i : cmd.content)  
    {  
        cout << i << " ";  
    }  
    cout << endl;  
}
```

```
/*
 * @description: 执行外部命令
 * @param {item} &cmd
 * @return {*}
 */
void EXEC(item &cmd)
{
    //提取字符串并调用execvp方法
    char *arg[cmd.content.size()];
    int cnt = 0;
    for (auto j : cmd.content)
    {
        arg[cnt++] = new char[strlen(const_cast<char *>(j.c_str()))];
        strcpy(arg[cnt - 1], const_cast<char *>(j.c_str()));
#if _DEBUG_ == 1
        cout << cnt - 1 << " " << j << " " << arg[cnt - 1] << endl;
#endif
    }
    arg[cnt] = nullptr;
#if _DEBUG_ == 1
    cout << i[0].content[0].c_str() << endl;
    for (int k = 0; k < cnt; k++)
    {
        cout << k << " " << arg[k] << endl;
    }
#endif
    //调用execvp方法
    if (!execvp(cmd.content[0].c_str(), arg))
        fprintf(stderr, "MYSHELL:NOT FOUND COMMAND");
}

/*
 * @description: test对比命令
 * @param {item} &cmd
 * @return {*}
 */
void TEST(item &cmd)
{
    int res = 0;
    //对比数字
    if (cmd.content[2] == "-eq")
        res = (TurnToInt(cmd.content[1]) == TurnToInt(cmd.content[3]));
    else if (cmd.content[2] == "-ge")
        res = (TurnToInt(cmd.content[1]) >= TurnToInt(cmd.content[3]));
    else if (cmd.content[2] == "-lt")
        res = (TurnToInt(cmd.content[1]) < TurnToInt(cmd.content[3]));
    else if (cmd.content[2] == "-ne")
        res = (TurnToInt(cmd.content[1]) != TurnToInt(cmd.content[3]));
    else if (cmd.content[2] == "-gt")
        res = (TurnToInt(cmd.content[1]) > TurnToInt(cmd.content[3]));
    else if (cmd.content[2] == "-le")
        res = (TurnToInt(cmd.content[1]) <= TurnToInt(cmd.content[3]));
    //字符串对比
    else if (cmd.content[2] == "=")
        res = (cmd.content[1] == cmd.content[3]);
}
```

```
else if (cmd.content[2] == "!=")
    res = (cmd.content[1] != cmd.content[3]);
    cout << res << endl;
}
/***
 * @description: exit命令
 * @param {item} &tmp
 * @return {*}
 */
void EFUN(item &tmp)
{
    exit(0);
}
```

## Myshell.h

```
/***
 * @Author: 韩恺荣
 * @Date: 2022-08-13 10:06:44
 * @LastEditTime: 2022-08-13 11:51:43
 * @LastEditors: 韩恺荣
 * @Description: Myshell.h 头文件引用和数据结构定义
 */

#ifndef _MYSHELL_H_
#define _MYSHELL_H_
/*-----头文件引用-----*/
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <unistd.h>
#include <stdlib.h>
#include <cctime>
#include <cstring>
#include <sys/types.h>
#include <sys/stat.h>
#include <iomanip>
#include <dirent.h>
#include <sys/shm.h>
#include <wait.h>
#include <fcntl.h>
#include <signal.h>
#include <iomanip>
#include <time.h>

using namespace std;
/*-----宏定义-----*/
#define MAX_JOB_SIZE 100
/*-----返回值定义-----*/
enum Pr_State
{
```

```
NOT_FOUND = -1,
SUCCEED

};

/*-----数据结构定义区-----*/

// job类, 用于定义不同的工作进程
class job
{
public:
    string name; //进程名称
    int pid; //进程id
    bool fg; //是否为前台程序
    bool run; //是否正在运行
    bool over; //是否已经结束运行
    int bg_id; //后台进程号
};

// process类, 用于管理进程
class Process
{
public:
    int sum_jobs; //所有的工作进程数量
    int sum_bgs; //后台运行的进程数量
    job jobs[MAX_JOB_SIZE]; // jobs数组, 定义每一个进程的状态
    Process() : sum_jobs(0), sum_bgs(0)
    { //缺省初始化函数
    }
    Process(int sum_cnt_, int bg_cnt_) : sum_jobs(sum_cnt_), sum_bgs(bg_cnt_) //含参的初始化函数
    {
    }
    void Init()
    { //重新初始化赋值
        sum_jobs = 0;
        sum_bgs = 0;
    }
    int Addjob(int pid_, string name_, bool fg_, bool run_, bool over_); //添加一个进程
    Pr_State Delete(int pid_); //删除一个进程
    Pr_State Update(); //更新进程表, 并删除所有已经结束的进程
    int FindJobs(int pid_); //查找进程id, 失败时返回-1
    int Findbgs(int pid_); //查找后台id, 失败时返回-1
};

//更新函数
Pr_State Process::Update()
{
    for (int i = 0; i < sum_jobs - 1; i++)
    {
        //遍历进程表, 对已经结束 的进程执行删除
        if (jobs[i].over)
```

```
        {
            Delete(jobs[i].pid);
        }
    }
    return SUCCEED;
}
//查找进程id
int Process::FindJobs(int pid_)
{
    for (int i = 0; i < sum_jobs; i++)
    {
        //遍历进程表，找到pid一致的进程时，返回下标，否则返回-1
        if (pid_ == jobs[i].pid)
            return i;
    }
    return -1;
}
//查找后台id
int Process::Findbgs(int pid_)
{
    for (int i = 0; i < sum_jobs; i++)
    {
        //遍历进程表，找pid一致的后台进程时，返回下标，否则返回-1
        if (pid_ == jobs[i].bg_id)
            return i;
    }
    return -1;
}
// item类。用于定义一个执行命令的容器
class item
{
public:
    vector<string> content; //命令及参数
    bool IsBG;           //命令是否为后台运行
    bool pipIn;          //命令是否有管道输入
    bool pipOut;         //命令是否有管道输出
    bool reIn;           //重定向输入标记
    bool reOut;          //重定向输出(覆盖)
    bool reApp;          //重定向输出(覆盖)
    string InPath;       //重定向输入路径
    string OutPath;      //重定向输出路径
    item()              //初始化函数
    {
        vector<string> tmp;
        IsBG = false;
        pipIn = false;
        pipOut = false;
        reIn = false;
        reOut = false;
        reApp = false;
        InPath = "";
        OutPath = "";
        content = tmp;
    }
}
```

```
~item() {} //析构函数
};

//全局变量, Pro_manager用于管理所有的进程
Process *Pro_manager;
//命令的类
class Command
{
public:
    vector<vector<item>> cmd_container; //命令及参数
    Command() {} //构造函数
    Command(vector<vector<item>> &cmd) : cmd_container(cmd) {} //带参数的构造
    void Create(vector<vector<item>> &cmd)
    { //构造后的赋值
        cmd_container = cmd;
    }
    ~Command()
    { //析构函数
        for (auto i : cmd_container)
        {
            for (auto j : i)
            {
                j.~item();
            }
        }
    }
};

// zflag全局变量, 用于标志是否按下了ctrl+z
int Z_FLAG = 0;
// bgflag, 用于标志是否发生了kill(pid,SIGCONT)行为
int BG_FLAG = 0;
//信号类, 用于定义新的信号处理以及保存旧的信号处理
struct sigaction *new_sig = new struct sigaction;
struct sigaction old_sig;
//该函数用于输出debug信息, 打印当前的命令表
void Output()
{
    cout << Pro_manager->sum_jobs << Pro_manager->sum_bgs << endl;
    cout << "name pid over run bg_id fg" << endl;
    for (int i = 0; i < Pro_manager->sum_jobs; i++)
    {
        //格式控制, 以便表格美观
        cout << setw(10) << Pro_manager->jobs[i].name << setw(10) << Pro_manager->jobs[i].pid;
        cout << setw(3) << Pro_manager->jobs[i].over << setw(3) << Pro_manager->jobs[i].run << setw(3) << Pro_manager->jobs[i].bg_id << setw(3) << Pro_manager->jobs[i].fg << endl;
    }
}
//新定义的信号处理函数
void Handler_S(int sign, siginfo_t *catcher, void *pointer);
void Handler_Z(int sign);
//信号处理的函数
void Sig_manager()
```

```
{  
    //绑定新函数处理CHDL信号  
    new_sig->sa_sigaction = Handler_S;  
    //处理过后恢复，并且保存处理时的相关信息  
    new_sig->sa_flags = SA_RESTART | SA_SIGINFO;  
    //置空忽略的信号集  
    sigemptyset(&new_sig->sa_mask);  
    //信号绑定  
    signal(SIGTSTP, Handler_Z);  
    signal(SIGSTOP, Handler_Z);  
    sigaction(SIGCHLD, new_sig, &old_sig);  
}  
//全局初始化  
void Init()  
{  
    // IPC资源处理，实现共享内存  
    int shmid;  
    if ((shmid = shmget(IPC_PRIVATE, sizeof(Process), 0666 | IPC_CREAT)) == -1)  
    {  
        fprintf(stderr, "共享内存创建失败");  
        exit(1);  
    }  
    //绑定共享内存当当前的进程  
    void *shm = NULL;  
    if ((shm = shmat(shmid, 0, 0)) == (void *)-1)  
    {  
        fprintf(stderr, "共享内存连接失败");  
        exit(1);  
    }  
    //用共享内存赋值给进程管理器  
    Pro_manager = (Process *)shm;  
    Pro_manager->Init();  
    //默认新添加一个进程，为MYSHELL  
    Pro_manager->Addjob(0, "MYSHELL", true, true, false);  
    //信号的初始化  
    Sig_manager();  
    // Output();  
}  
//新建一个job  
int Process::Addjob(int pid_, string name_, bool fg_, bool run_, bool over_)  
{  
    //用输入的参数赋值给当前的jobs  
    jobs[sum_jobs].pid = pid_;  
    jobs[sum_jobs].name = name_;  
    jobs[sum_jobs].fg = fg_;  
    jobs[sum_jobs].run = run_;  
    jobs[sum_jobs].over = over_;  
    //如果是后台进程，还要更新后台进程号  
    if (!fg_)  
    {  
        jobs[sum_jobs].bg_id = (++sum_bgs);  
    }  
    //进程的总数加1  
    sum_jobs++;
```

```

        return sum_jobs - 1;
    }
    //删除一个进程
Pr_State Process::Delete(int pid_)
{
    int i;
    for (i = 0; i < sum_jobs; i++)
    {
        //遍历进程表，找到要删除的进程
        if (pid_ == jobs[i].pid)
        {
            break;
        }
    }
    //如果没有找到，返回值为NOT_FOUND
    if (i == sum_jobs)
        return NOT_FOUND;
    bool flag = false;
    //如果要删除的进程是后台进程
    if (!jobs[i].fg)
    {
        flag = true;
        sum_bgs--;
    }
    //之后的每个后台进程，都要更新他们的后台进程号
    for (int j = i; j < sum_jobs - 1; j++)
    {
        jobs[j] = jobs[j + 1];
        //更新后台进程号
        if (flag && !jobs[j].fg)
        {
            jobs[j].bg_id--;
        }
    }
    sum_jobs--;
    //返回删除成功的标志
    return SUCCEED;
}

#endif

```

## (10) man.txt内容

欢迎使用MyShell的使用手册和指南

本程序实现了在乌班图系统中的mini Shell，该shell支持如下功能： 1、通过自定义函数实现了部分的内建命令  
2、通过execvp函数实现了外部命令的调用

一、运行指南 想要运行myshell，你需要在linux系统下，编译运行三个文件，可以使用vscode运行该项目。为了配置vscode的运行环境，你需要配置如下三个文件（这是一种可行的文件，你可以选择使用其他的配置）：

1、launch.json： { "configurations": [ { "name": "C/C++: g++ 生成和调试活动文件", "type": "cppdbg", "request": "launch", "program": "fileDirname/{fileBasenameNoExtension}", "args": [], "stopAtEntry": false,

```

{fileDirname},
"environment": [],
"externalConsole": false,
"MIMode": "gdb",
"setupCommands": [
{
"description": "为 gdb 启用整齐打印",
"text": "-enable-pretty-printing",
"ignoreFailures": true
},
{
"description": "将反汇编风格设置为 Intel",
"text": "-gdb-set disassembly-flavor intel",
"ignoreFailures": true
}
],
"preLaunchTask": "C/C++: g++ 生成活动文件",
"miDebuggerPath": "/usr/bin/gdb"
}
],
"version": "2.0.0"
}
2、settings.json:
{
"files.associations": {
"string": "cpp",
"*.tcc": "cpp",
"compare": "cpp",
"type_traits": "cpp",
"iosfwd": "cpp",
"iostream": "cpp",
"vector": "cpp",
"new": "cpp",
"ostream": "cpp",
"iomanip": "cpp"
}
}
3、tasks.json:
{
"tasks": [
{
"type": "cppbuild",
"label": "C/C++: g++ 生成活动文件",
"command": "/usr/bin/g++",
"args": [
"-fdiagnostics-color=always",
"-g",
"c:\\Users\\Administrator\\Desktop\\test\\main.cpp"
],
"cwd": "{fileDirname}",
"options": { "cwd": "c:\\Users\\Administrator\\Desktop\\test\\main.cpp" }
}, "group": { "kind": "build", "isDefault": true }, "detail": "调试器生成的任务。 " },
"version": "2.0.0" }

```

## 二、MyShell的工程架构：

1、MyShell的主要模块： myshell是由一个语法解析器和一个命令执行器组成，语法解析器负责解析语法，而命令执行器负责执行解析出的命令。对于使用者输入的命令，解析器将首先按照“;”分割，然后用空格、回车和

制表符分割，其中;分割出命令组，而后者分割出具体的命令 对于分割出的命令，我使用 | 对命令再次分割，注意，在使用者输入 | 时，需要在 “|” 之前和之后保留空格，否则将解析命令错误。在上述模块的作用下，会发生如下行为：假设输入 “ls -l | head -2 | wc -l ; time; echo a”，会被解析为： 第一组：“ls -l head -2 wc -l”（一定要在管道符前后保留空格！！！） 第二组：time 第三组：echo a 然后依次执行三组命令

## 2、Myshell支持的功能：

在语法解析器和执行器下，Myshell可以支持后台运行操作（以&为命令标志），可以支持挂起操作（捕获SIGSTOP信号），可以支持进程的前后台切换，也可以支持将存有命令的文件作为输入

### 三、Myshell 的命令手册：(下面是自定义的命令执行，对于其他外部命令，参考官方的shell手册，如ls -l等)

1. bg命令 解释：后台执行一个挂起的后台进程，支持bg和bg %n两种格式 范例：bg bg %1
2. cd命令 解释：无参数显示当前目录，有参数切换文件目录并更改环境变量 范例：cd  
范例：cd
3. clr命令 解释：清理当前屏幕内容 范例：clr
4. dir命令 解释：无参数显示当前文件夹内容，有参数显示参数文件夹内容，支持 -A, -l, -a选项，但三种选项每次最多输入一个 范例：dir  
范例：dir  
范例：dir -l
5. echo命令 解释：显示参数内容 范例：echo Hello World
6. exec命令 解释：使用参数命令代替当前进程 范例：exec ls -l
7. exit 解释：退出shell 范例：exit
8. fg 解释：将后台进程转到前台执行,支持fg和fg %n两种格式 范例：fg
9. help 解释：输出用户手册并使用more进行控制 范例：help
10. jobs 解释：显示所有的后台进程及其状态 范例：jobs
11. pwd 解释：显示当前路径 范例：pwd
12. set 解释：显示所有的环境变量 范例：set
13. test 解释：比较字符串或数字之间的大小关系是否成立 范例：test aaa = aaa 范例：test bbb != aaa 范例：test 1 -eq 1 范例：test 1 -ne 2 范例：test 1 -lt 1 范例：test 1 -gt 1 范例：test 1 -le 1 范例：test 1 -ge 1
14. time 解释：输出当前系统时间 范例：time
15. umask 解释：无参数时显示当前掩码，有参数则设置掩码为参数 范例：umask 范例：umask 0666

## (11) 部分实验结果截图：

后台运行命令与jobs -l查看

```
ROOT@home/handsomehh/personal$ sleep 100 &
ROOT@home/handsomehh/personal$ jobs -l
[1] 2013038  sleep 正在运行
```

ls -l直接运行以及运行结束后显示的ls已完成标志

```
ROOT@/home/handsomehh/personal$ls -l &
ROOT@/home/handsomehh/personal$总用量 220
-rwxrwxr-x 1 handsomehh handsomehh 166744 8月 14 15:23 a.out
-rw-rw-r-- 1 handsomehh handsomehh 15847 8月 14 15:28 Command.cpp
-rw-rw-r-- 1 handsomehh handsomehh 26139 8月 13 14:57 main.cpp
-rw-rw-r-- 1 handsomehh handsomehh 8005 8月 14 15:17 Myshell.h
drwxrwxr-x 6 handsomehh handsomehh 4096 7月 20 20:30 taxes
[2014710]ls 已完成
```

sleep命令后台转前台，运行结束后jobs查看

```
ROOT@/home/handsomehh/personal$sleep 20 &
ROOT@/home/handsomehh/personal$fg
ROOT@/home/handsomehh/personal$jobs -l
```

sleep 命令的挂起

```
ROOT@/home/handsomehh/personal$sleep 20
^Z
[2]2020182 + is already stop
ROOT@/home/handsomehh/personal$jobs -l
[2]2020182 + sleep 已挂起
```

sleep命令挂起后转后台

```
ROOT@/home/handsomehh/personal$bg
[2]sleep 转至后台运行
ROOT@/home/handsomehh/personal$[2020182]sleep 已完成
```

重定向的一个例子

```
ROOT@/home/handsomehh/personal$ls -l > 2.txt
ROOT@/home/handsomehh/personal$cat 2.txt
总用量 220
-rw-rw-r-- 1 handsomehh handsomehh 0 8月 14 15:43 2.txt
-rwxrwxr-x 1 handsomehh handsomehh 166744 8月 14 15:23 a.out
-rw-rw-r-- 1 handsomehh handsomehh 15847 8月 14 15:28 Command.cpp
-rw-rw-r-- 1 handsomehh handsomehh 26139 8月 13 14:57 main.cpp
-rw-rw-r-- 1 handsomehh handsomehh 8005 8月 14 15:17 Myshell.h
drwxrwxr-x 6 handsomehh handsomehh 4096 7月 20 20:30 taxes
```

管道操作的一个例子

```
ROOT@/home/handsomehh/personal$ls -l | head -2 | wc -l  
2
```

文件作为参数输入的一个例子

```
handsomehh@handsomehh-virtual-machine:~/personal$ ./a.out 2.txt  
ROOT@/home/handsomehh/personal$time  
Sun Aug 14 15:48:07 2022
```

```
ROOT@/home/handsomehh/personal$pwd  
/home/handsomehh/personal  
ROOT@/home/handsomehh/personal$ls -l  
总用量 224  
-rw-rw-r-- 1 handsomehh handsomehh 16 8月 14 15:47 2.txt  
-rwxrwxr-x 1 handsomehh handsomehh 166744 8月 14 15:23 a.out  
-rw-rw-r-- 1 handsomehh handsomehh 15847 8月 14 15:28 Command.cpp  
-rw-rw-r-- 1 handsomehh handsomehh 26139 8月 13 14:57 main.cpp  
-rw-rw-r-- 1 handsomehh handsomehh 8005 8月 14 15:17 Myshell.h  
drwxrwxr-x 6 handsomehh handsomehh 4096 7月 20 20:30 taxes  
ROOT@/home/handsomehh/personal$
```

用;分割命令并执行的一个例子

```
ROOT@/home/handsomehh/personal$time;ls -l  
Sun Aug 14 15:48:40 2022  
  
总用量 224  
-rw-rw-r-- 1 handsomehh handsomehh 16 8月 14 15:47 2.txt  
-rwxrwxr-x 1 handsomehh handsomehh 166744 8月 14 15:23 a.out  
-rw-rw-r-- 1 handsomehh handsomehh 15847 8月 14 15:28 Command.cpp  
-rw-rw-r-- 1 handsomehh handsomehh 26139 8月 13 14:57 main.cpp  
-rw-rw-r-- 1 handsomehh handsomehh 8005 8月 14 15:17 Myshell.h  
drwxrwxr-x 6 handsomehh handsomehh 4096 7月 20 20:30 taxes
```

### 三、实验心得

bash脚本编程的学习，相比上一个作业而言困难了许多，尤其是最后两个作业的实现中，我遇到了许多问题，也在解决问题中学到了许多，以下是我的学习心得：

前两道题目中，让我对makefile的书写有了一定的概念，并且我在书写makefile的过程中，也通过上网搜索资料学到了一些技巧，比如我们可以使用一些简便的方式来书写，以及@^来指代生成项和依赖项，通过定义clean方法来删除中间文件，相信这会对我下学期操作系统的学产生积极作用与影响。

第三四道题，让我熟悉了bash脚本的基本语法，我也发现虽然学习语法感觉自己学的很快，但是到了真正使用的时候，反倒会想不起来该用什么，以及该如何使用，比如以为自己书写的if语句是对的，实际上自己甚至没发现else if要写成elif，结果找了半天都不知道自己错在哪里，不过虽然纠错的过程是痛苦的，但是成果是显著的，我也学习了一些调试方式，比如对bash增加编译时的-x参数选项，可以详细的看到程序的运行过程。这种不熟练的情况让一开始上手的过程会比较的慢，但是写了这两道题目之后，我对语法就基本上熟练了。

第五道题的备份问题，相比前两道题目的难度略有增加，但是难度还是比较简单的，这道题的关键在于理解题目的含义，在理解含义之后，编写代码就会很顺畅了。不过编写的过程我也遇到了一些困难，比如之前上课时我不理解@和\*的区别，直到编写这道题时，我要向函数传递参数时，在多次尝试后才彻底明白这两种表达在被引号引用时的区别和差异，以及在使用for i in array[@]和for i in array[\*]的区别。这道题中我也对eval函数的作用有了一定的理解，例如当i=1，如何用i表示1，如果不使用eval，那么\$会被提前解析，从而达不到我们想要的目的。

第六题的Myvim，主要遇到的困难是如何像vim一样每次控制界面中的显示内容恰好为一页，我之前使用head命令，但是发现这样的行是文本意义上的行，不是显示屏意义上的行，后来又尝试使用了awk等方式，但是效果都不怎么样，最后我决定从最基本的数组角度入手，逐字逐句按行标顺序显示直到这一个屏幕恰好适配即可，但是这也造成了一定的负面影响，那就是当输入的是中文时格式会混乱，因为一个中文占位两个格子且我们无法对中文做出简单高效的判断和特殊出路，在权衡之下我决定放弃对中文的处理以达到显示意义上的美观，所以我的MYvim是基于英文文本处理的。此外光标的控制是这个程序中最难和最复杂的一部分，我用大量的篇幅来处理各种各样的特殊情况，这其中的debug过程异常难受，甚至一开始我不知道屏幕的坐标原点（左上角）是(1, 1)而不是(0,0),因shell不像c++等语言我们可以单步调试，实时观察结果，且光标控制的处理中会出现各种各样奇怪的bug，例如光标突然跳转，例如文本突然消失，在我不断的更改下，好在最终效果还不错，努力也得到了一定的回报。

最后一题也是最难的一题，为了解决这一题，我看了我们的课件和视频后，对这一题还是没有太多的头绪，一开始对fork等函数的具体行为也没有什么概念，处于一种似是而非的状态。于是我在网上搜集查看了各种各样的文档和函数解析，也结合一本unix系统编程的书籍，不断加深理解，才逐渐的建立起对这个项目的基本框架理解。当然这个项目中遇到的bug也是这几道题中最多的，从一开始使用全局变量保存进程表，发现子进程的更改和父进程的更改是不同步在一个变量中的，通过资料我才知道fork后子进程和父进程会各自复制和继承什么，后来改用建立共享内存保存进程表后，也发现进程表的内容总是被莫名其妙的修改，在我将创建共享内存的第一个参数更改为每一次都创建时而不是某个确定值后，问题才被解决。在分割字符串以获取输入的命令时，我也遇到了c++容器方面的问题，我新创建的变量总是会自动初始化之前那个变量内的内容，不断尝试后，我改用指针new出内容来解决这一个问题，在信号处理控制中，我发现当后台运行结束后会出现循环打印出满屏幕的当前路径，后来我将信号处理的参数改为RESTART并保存old action才得以解决这个问题，在管道通信时，我发现cout如果最后没有endl居然不会被打印进入重定向的文件，此外，我也在多次尝试后采用dup和dup2函数来重定向我们的输入和输出，解决了一些不明所以的bug。

这次作业，花费了大量的精力，但也学到了许多linux里的知识点，在数千行代码的锻炼下，我的代码能力也有了进一步的提高，我也有了学好下学期操作系统课程的信心，感谢老师的辛勤付出，我也相信天道酬勤，功夫不负有心人~