

# 实验2 程序设计

## 实验目的

1. 学习Bourne shell的shell脚本的基本概念
2. 学会shell程序如何执行
3. 通过写脚本，学会编写Bourne shell 脚本程序的方法
4. 理解并掌握shell

## 实验内容

### 1. makefile理解

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。makefile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。

```
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
```

a. 所有宏定义的名字

```
CC OPTIONS OBJECTS SOURCES HEADERS
```

b. 所有目标文件的名字

最终生成的目标文件名为power，不过题干里该规则的目标叫polish

```
power(polish) main.o stack.o misc.o
```

c. 每个目标的依赖文件

文件中每一个规则的名字后面即是每个目标的依赖文件

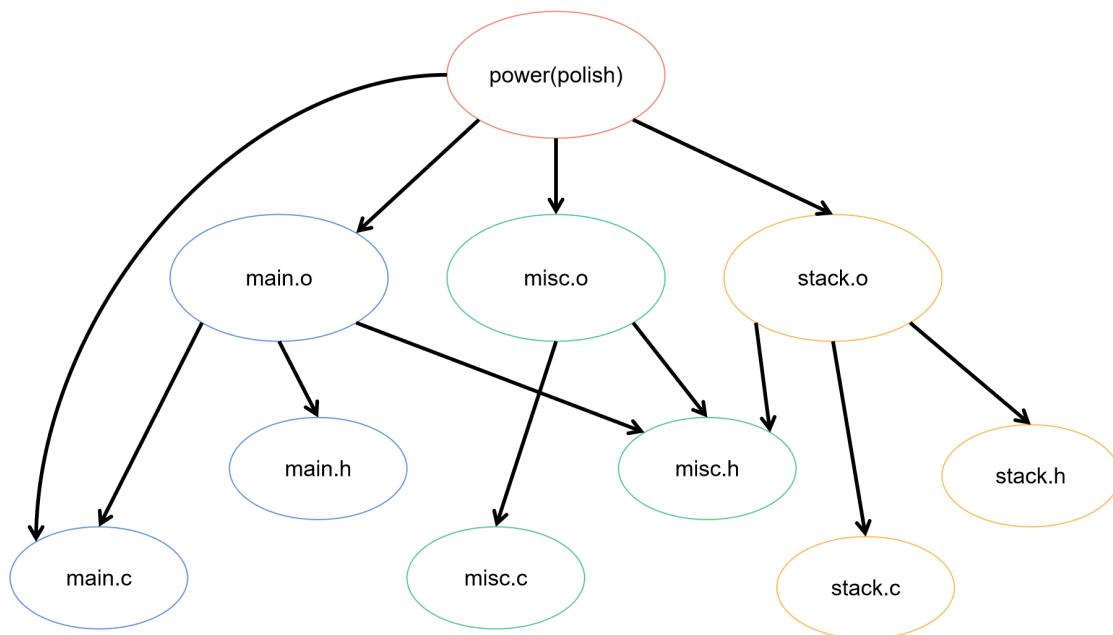
```
power(polish): main.c main.o stack.o misc.o  
main.o: main.c main.h misc.h  
stack.o: stack.c stack.h misc.h  
misc.o: misc.c misc.h
```

d. 生成每个目标文件所需执行的命令

其中第一条命令题干给出，后三条命令题干没给，但是实践表明后三条命令不能省略

```
power(polish): gcc -O3 -o power main.o stack.o misc.o -lm  
main.o: gcc -c main.c  
stack.o: gcc -c stack.c  
misc.o: gcc -c misc.c
```

e. 画出makefile对应的依赖关系树。



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

即为d问题中提到的题干省略的命令

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

## 2. makefile实践

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));
}
$
```

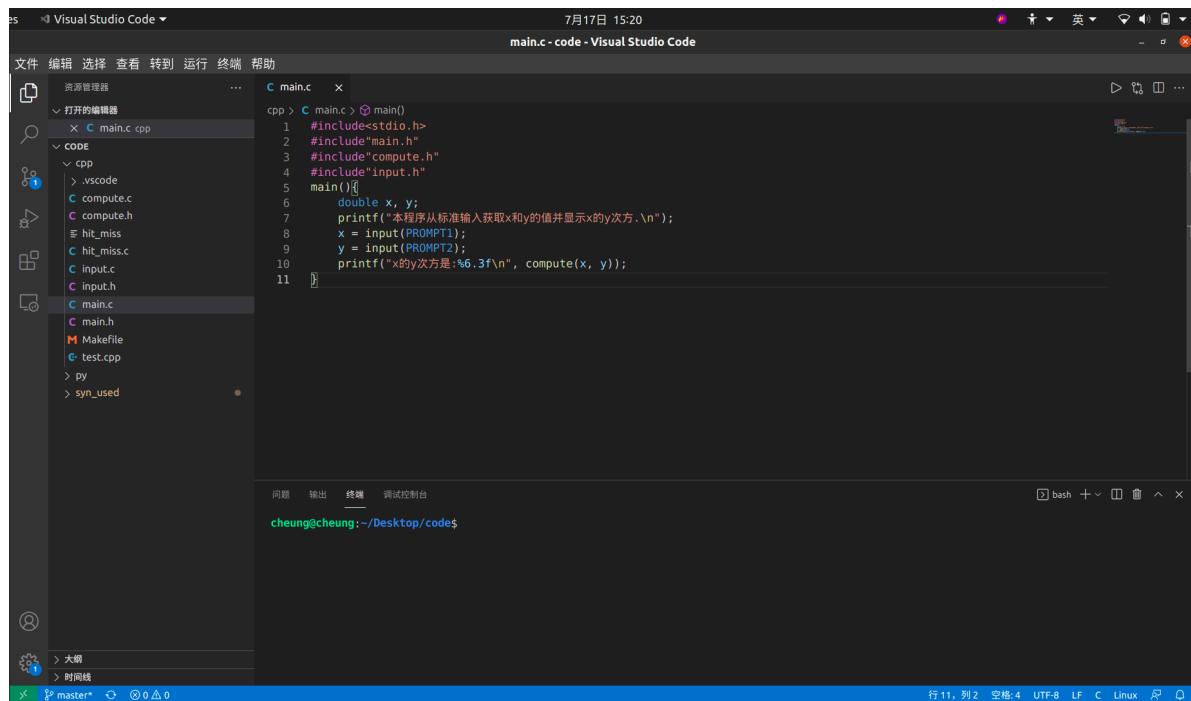
为了得到可执行文件power，我们必须首先从三个源文件编译得到目标文件，并把它们连接在一起。下面的命令将完成这一任务。注意，在生成可执行代码时不要忘了连接上数学库。

```

$ 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。

首先打开编译器，创建需要的源文件与头文件，如下图所示：



创建好文件并输入内容保存后，用 `cat` 命令检查一次几个文件的内容。然后输入两个 `gcc` 命令进行编译。如下图所示。其中第一个 `gcc` 命令编译提示的警告是主函数的返回类型没有显式定义，不必理会。第二个 `gcc` 命令中的 `-o` 选项后接生成的文件名，如不指定则默认为 `a.out`；`-lm` 选项代表编译时要连接到数学库，这是由于源文件中调用了 `sqrt()` 函数。

```

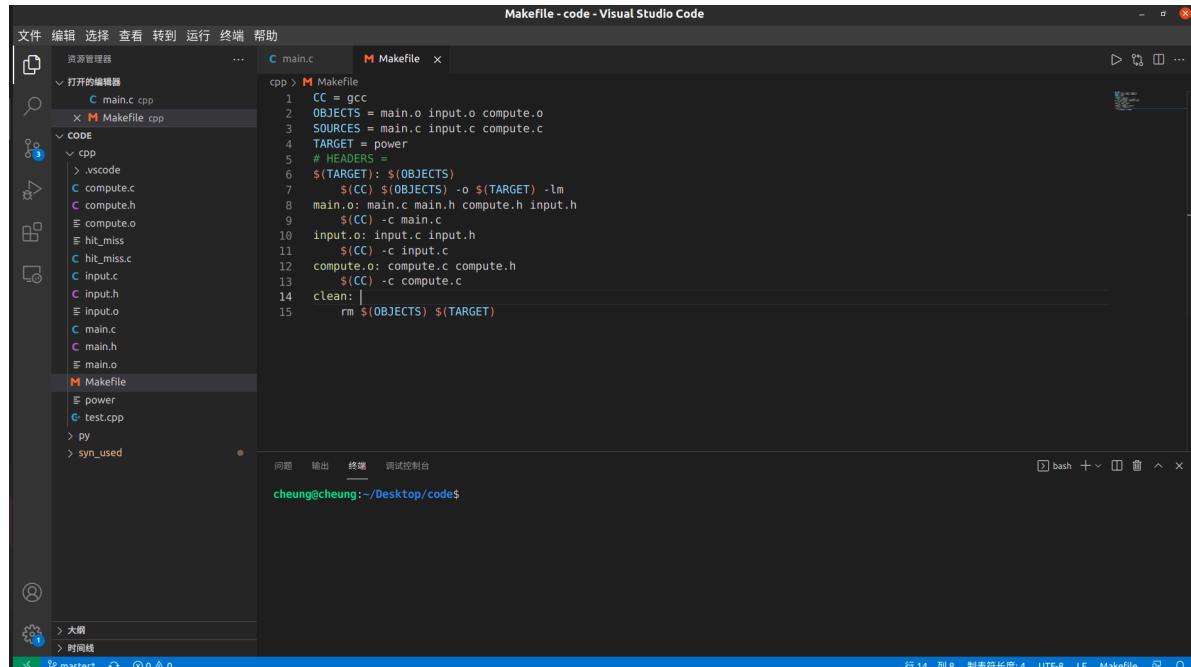
    cheung@cheung:~/Desktop/code/cpp$ cd ..
    cheung@cheung:~/Desktop/code$ code .
    cheung@cheung:~/Desktop/code$ cd cpp/
    cheung@cheung:~/Desktop/code/cpp$ cat compute.h
    double compute(double, double); cheung@cheung:~/Desktop/code/cpp$ cat input.h
    double input(char *); cheung@cheung:~/Desktop/code/cpp$ cat main.h
    #define PROMPT1 "请输入x的值："
    #define PROMPT2 "请输入y的值：" cheung@cheung:~/Desktop/code/cpp$ cat compute.c
    #include<math.h>
    #include<stdio.h>
    #include"compute.h"
    double compute(double x, double y){
        return (pow ((double) x, (double) y));
    } cheung@cheung:~/Desktop/code/cpp$ cat input.c
    #include<stdio.h>
    #include"input.h"
    double input(char *s){
        float x;
        printf("%s", s);
        scanf("%f", &x);
        return (x);
    } cheung@cheung:~/Desktop/code/cpp$ 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("%d的y次方是:%.3f\n", compute(x, y));
    } cheung@cheung:~/Desktop/code/cpp$ gcc -c main.c input.c compute.c
    main.c:5:1: warning: return type defaults to 'int' [-Wimplicit-int]
    5 | main(){
    | ^
    cheung@cheung:~/Desktop/code/cpp$ gcc main.o input.o compute.o -o power -lm
    cheung@cheung:~/Desktop/code/cpp$ ./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：2.2
    请输入y的值：3.3
    x的y次方是:13.489
    cheung@cheung:~/Desktop/code/cpp$ |

```

编译完成后运行 `power` 程序，可以看到程序正确输出了结果，与计算器验证的结果一致。

(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。

如下图创建一个Makefile文件并输入内容。



打开终端，首先把第一步中生成的文件清除，然后输入make命令。出现的警告与第一步中一致，然后运行编译生成的程序，可以看出程序运行的答案正确。

```
cheung@cheung:~/Desktop/code/cpp$ make clean
rm main.o input.o compute.o power
cheung@cheung:~/Desktop/code/cpp$ make
gcc -c main.c
main.c:5:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
  5 | main(){}
     | ^~~~
gcc -c input.c
gcc -c compute.c
gcc main.o input.o compute.o -o power -lm
cheung@cheung:~/Desktop/code/cpp$ ./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：2.2
请输入y的值：3.3
x的y次方是：13.489
cheung@cheung:~/Desktop/code/cpp$
```

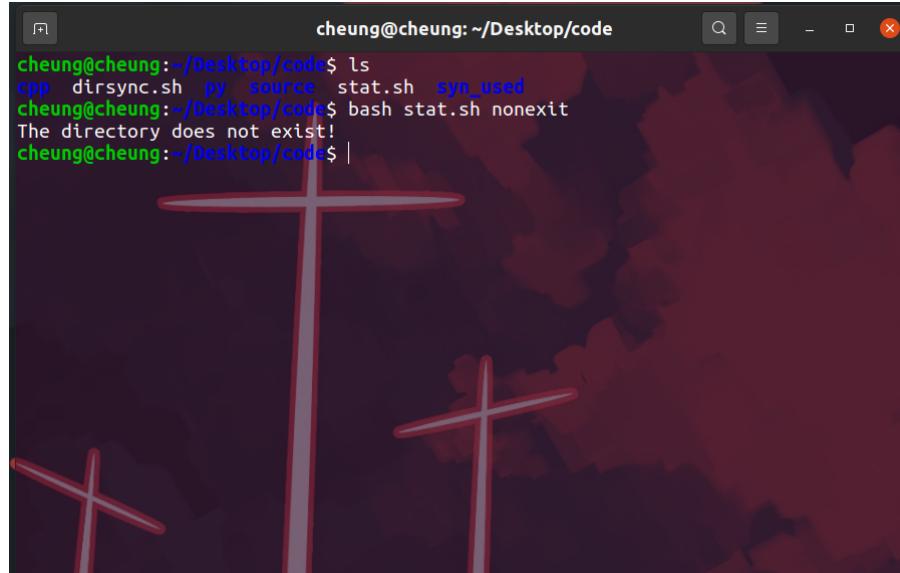
源代码：

```
CC = gcc
OBJECTS = main.o input.o compute.o
SOURCES = main.c input.c compute.c
TARGET = power
# HEADERS =
$(TARGET): $(OBJECTS)
    $(CC) $(OBJECTS) -o $(TARGET) -lm
main.o: main.c main.h compute.h input.h
    $(CC) -c main.c
input.o: input.c input.h
    $(CC) -c input.c
compute.o: compute.c compute.h
    $(CC) -c compute.c
clean:
    rm $(OBJECTS) $(TARGET)
```

### 3. 统计脚本

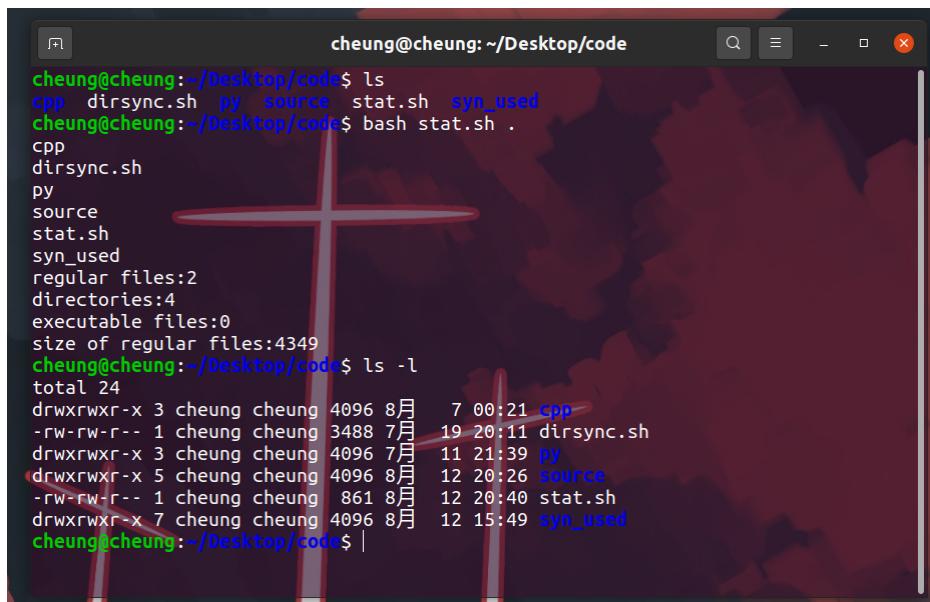
编写shell脚本，统计指定目录下的普通文件、子目录及可执行文件的数目，统计该目录下所有普通文件字节数总和，目录的路径名字由参数传入。

脚本的开始，我首先**检查了输入的参数个数**（题目未要求，我设定为一次只能输入一个目录名）以及**输入的参数是否为一个存在的目录**。如下图，如果输入的目录不存在，则脚本会报错并退出。**与预期相符**。



```
cheung@cheung:~/Desktop/code$ ls
cpp  dirsync.sh  py  source  stat.sh  syn_used
cheung@cheung:~/Desktop/code$ bash stat.sh nonexit
The directory does not exist!
cheung@cheung:~/Desktop/code$ |
```

随后，进入参数所指示的目录，**遍历该目录下的所有文件**。我写的脚本会首先把文件的**名字打印出来**，然后利用 `-f`、`-d` 等测试，**判断文件的类型**，并对不同类型的文件数量进行统计。如果是普通文件，再利用 `wc -c` 命令获得其大小，进行累加。遍历完成后输出统计信息。第一个测试如下，统计当前目录的信息，可以看到脚本输出的文件名与 `ls` 命令输出的一致。再对比 `ls -l` 命令的输出，可以看到**三个类型的文件总数统计正确，普通文件的总大小也正确** ( $4349=3488+861$ )，**与预期相符**。



```
cheung@cheung:~/Desktop/code$ ls
cpp  dirsync.sh  py  source  stat.sh  syn_used
cheung@cheung:~/Desktop/code$ bash stat.sh .
cpp
dirsync.sh
py
source
stat.sh
syn_used
regular files:2
directories:4
executable files:0
size of regular files:4349
cheung@cheung:~/Desktop/code$ ls -l
total 24
drwxrwxr-x 3 cheung cheung 4096 8月   7 00:21 cpp
-rw-rw-r-- 1 cheung cheung 3488 7月  19 20:11 dirsync.sh
drwxrwxr-x 3 cheung cheung 4096 7月  11 21:39 py
drwxrwxr-x 5 cheung cheung 4096 8月  12 20:26 source
-rw-rw-r-- 1 cheung cheung  861 8月  12 20:40 stat.sh
drwxrwxr-x 7 cheung cheung 4096 8月  12 15:49 syn_used
cheung@cheung:~/Desktop/code$ |
```

第二个测试如下，统计桌面下的文件信息，这次输入绝对路径。与上述分析一样，**不难看出统计结果正确。与预期相符**。

The screenshot shows a terminal window with the following session:

```
cheung@cheung:~/Desktop/code$ bash stat.sh ~/Desktop/
code
regular files:0
directories:1
executable files:0
size of regular files:0
cheung@cheung:~/Desktop/code$ ls -l ~/Desktop/
total 4
drwxrwxr-x 6 cheung cheung 4096 8月 12 20:40 code
cheung@cheung:~/Desktop/code$ |
```

源代码：

```
#!/usr/bin/bash
# 检查传入参数的数量
if [ $# -ne 1 ]
then
    echo "The number of variables is wrong!"
    exit 1
fi

# 检查输入的目录是否存在
if [ ! -d "$1" ]
then
    echo "The directory does not exist!"
    exit 1
fi

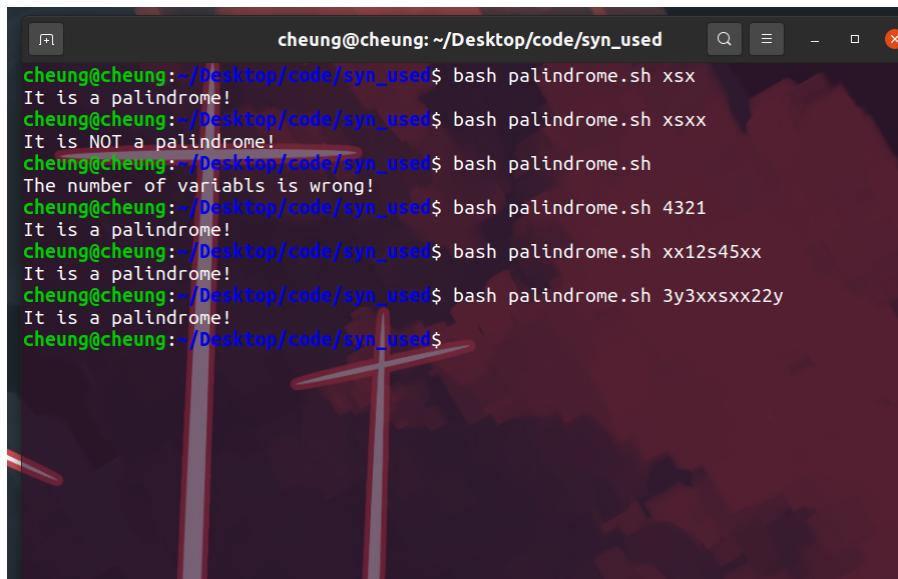
# 初始化数据
rf=0
df=0
xf=0
size=0
# 跳至目标目录下，方便进行遍历，同时也支持输入相对路径和绝对路径
cd $1
for filee in `ls $1`
do
echo $filee
if [ -f "$filee" ]; then      # 普通文件
    rf=`expr $rf + 1`
    info=`wc -c $filee` # 得到大小
    info=(${info[@]}) 
    size=`expr $size + ${info[0]}`
fi
if [ -d "$filee" ]; then      # 目录文件
    df=`expr $df + 1`
fi
if [ -f "$filee" ] && [ -x "$filee" ]; then # 可执行文件
    xf=`expr $xf + 1`
```

```
fi
done
echo "regular files:$rf"
echo "directories:$df"
echo "executable files:$xf"
echo "size of regular files:$size"
exit 0
```

## 4. 回文字符串脚本

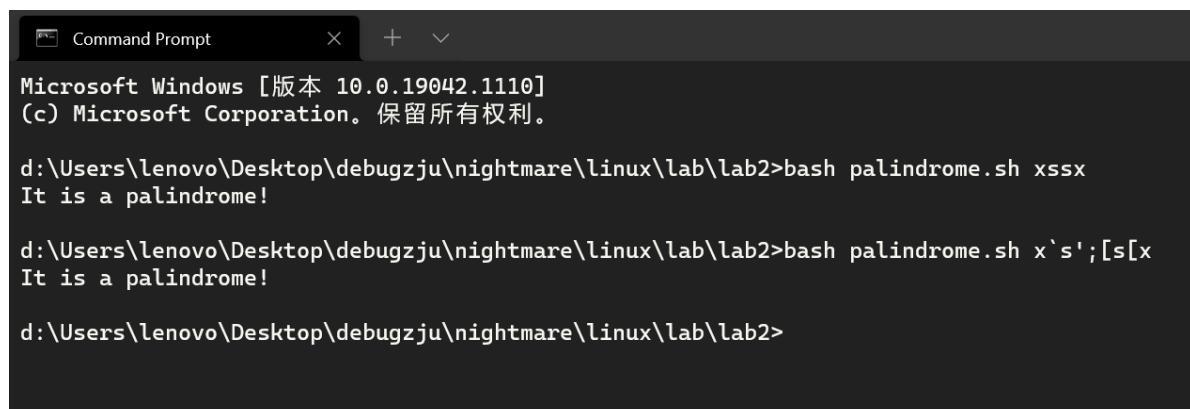
编写shell脚本，输入一个字符串，忽略（删除）非字母后，检测该字符串是否为回文(palindrome)。对于一个字符串，如果从前向后读和从后向前读都是同一个字符串，则称之为回文串。例如，单词“mom”，“dad”和“noon”都是回文串。

脚本的开始，我首先**检查输入的参数个数**，输入的参数少于或多于一个，脚本都会报错并退出。然后脚本获取输入，利用脚本的字符串处理功能以及正则表达式，**去除参数中的非字母**。最后开始判断得到的字符串是否为回文字符串。由回文字符串的性质不难知道，回文字符串的第一个字母与最后一个字母一定一致；去除首尾两个字母后的子串也肯定还是回文字符串，还是符合这个性质。因此，我设置left和right两个变量表示下标，逐对字母进行判断，如果循环能正常退出，则说明输入的字符串是回文字符串，否则不是。测试结果如下图所示。



```
cheung@cheung:~/Desktop/code/syn_used$ bash palindrome.sh xsx
It is a palindrome!
cheung@cheung:~/Desktop/code/syn_used$ bash palindrome.sh xsxx
It is NOT a palindrome!
cheung@cheung:~/Desktop/code/syn_used$ bash palindrome.sh
The number of variabls is wrong!
cheung@cheung:~/Desktop/code/syn_used$ bash palindrome.sh 4321
It is a palindrome!
cheung@cheung:~/Desktop/code/syn_used$ bash palindrome.sh xx12s45xx
It is a palindrome!
cheung@cheung:~/Desktop/code/syn_used$ bash palindrome.sh 3y3xxsxx22y
It is a palindrome!
cheung@cheung:~/Desktop/code/syn_used$
```

其中输入4321时，去除掉非字母后得到空串。此处没有对这一种输入进行特殊处理，因为我觉得空串也满足回文串的定义，因此让程序直接判断其为回文字符串。从截图可以看出，**程序测试与预期相符**。其中当回文字符串的长度为偶数时的测试结果如下。（在Linux下截图时忘记把这一种情况的测试结果也截图下来，因此直接在Windows下进行截图。在Linux的运行结果和再Windows下的运行结果都是一致且正确的。**与预期相符**。）



```
Microsoft Windows [版本 10.0.19042.1110]
(c) Microsoft Corporation。保留所有权利。

d:\Users\lenovo\Desktop\debugzju\nightmare\linux\lab\lab2>bash palindrome.sh xsss
It is a palindrome!

d:\Users\lenovo\Desktop\debugzju\nightmare\linux\lab\lab2>bash palindrome.sh x`s';[s[x
It is a palindrome!

d:\Users\lenovo\Desktop\debugzju\nightmare\linux\lab\lab2>
```

源代码：

```
#!/usr/bin/bash

# 检查传入参数的数量
if [ $# -ne 1 ]
then
    echo "The number of variables is wrong!"
    exit 1
fi

origin=$1    # 取得原输入
str=${origin//[^a-zA-Z]/}    # 去除非字母
# echo $str
left=0
right=${#str}-1    # 计算长度
# let "right--"
while [ $left -lt $right ]; do
    if [ ${str:$left:1} != ${str:$right:1} ]; then    # 不是回文
        echo "It is NOT a palindrome!"
        exit 1
    fi
    let "left++"
    let "right--"
done
echo "It is a palindrome!"    # 能执行到这一步的一定是回文
exit 0
```

## 5. 备份同步

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。

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。

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。

其它功能自行添加设计。

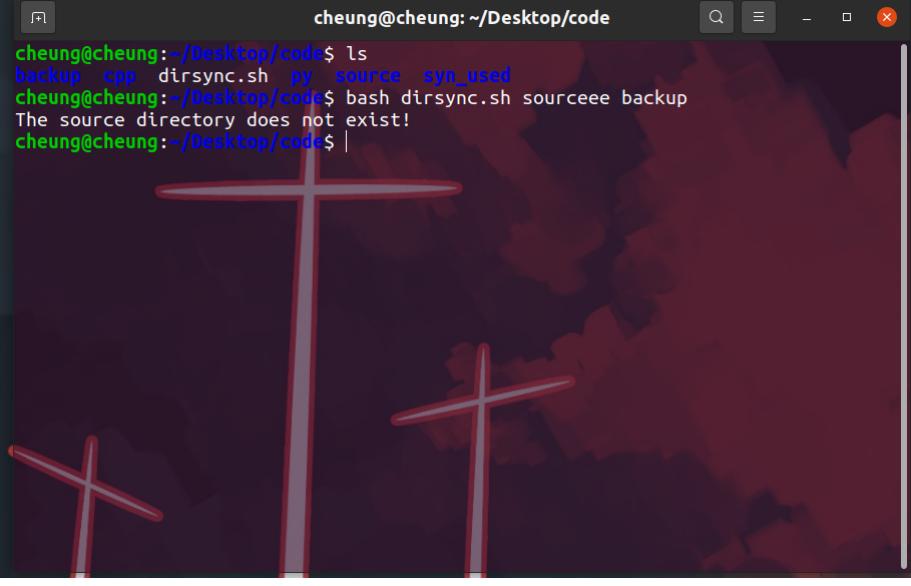
脚本首先会对输入的参数进行检查。**检查输入的参数个数**，如果不为2，则报错并退出；**检查第一个参数（源目录）是否存在**，如果不存在则报错并退出；还要**检查第二个参数（目标目录）是否存在**，如果不存在则脚本会创建该文件夹。此后脚本即开始**递归调用 dirsync() 函数**进行同步备份。

在解释函数的原理前，再解释一下我所理解的题干中的备份与同步的意思。结合题干要求与一般的对同步备份的理解，我将备份理解为**抛开内容不说，目标目录具有的文件、子目录应该与源目录的保持一致**。源目录具有而目标目录没有的文件，应该进行备份；源目录没有而目标目录有的文件，应该进行删除。备份关注的是**文件是否存在**的问题。而我将同步理解为**对于源目录和目标目录都含有的文件，内容应该与最新修改的内容保持一致**。二者都有的文件，如果源目录中的文件是更新的文件，则用其覆盖目标目录的文件；反过来则用目标目录的文件覆盖源目录的。同步关注的是**文件是否一致**的问题。

函数首先将源目录与目标目录中的文件名排序，并保存在临时文件中。对比两个文件的内容，即可以知道哪些文件需要备份哪些文件需要同步。此后的三个循环，**前两个循环用来实现备份功能，最后一个循环用来实现同步功能**。第一个循环进行增量备份，第二个循环删除多余的文件。通过 `comm -23` 可以得到源目录有而目标目录没有的文件；通过 `comm -13` 命令可以得到源目录没有而目标目录有的文件。第三个循环通过比较文件的修改时间进行双向同步，保证两个目录下的文件都是最新修改的。如果在上述三个循环中发现有子目录需要复制，则递归调用该函数。

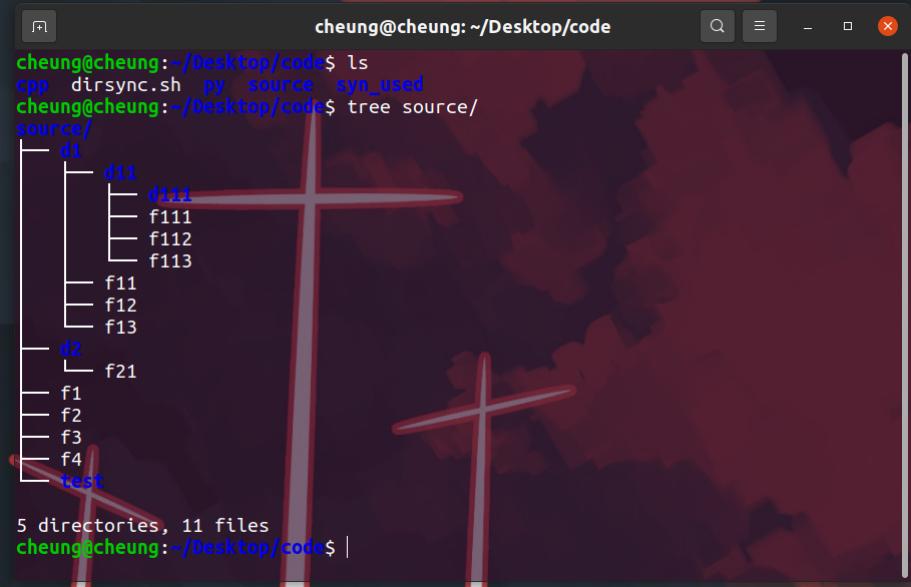
其中需要注意的是复制命令需要指定参数，使用 `cp -p` 命令，确保复制得到的新文件与原文件的修改时间一致；以及我在实践中发现，递归函数中使用同名变量的话，脚本不会为每一层调用保存各自的变量，因此我在调用递归函数时还要传入递归的层数，借助递归层数来区分每一层调用的变量。

验证的结果与截图如下。输入的源目录不存在时，脚本会报错并退出。



```
cheung@cheung:~/Desktop/code$ ls
backup cpp dirsync.sh py source syn_used
cheung@cheung:~/Desktop/code$ bash dirsync.sh sourceee backup
The source directory does not exist!
cheung@cheung:~/Desktop/code$ |
```

调用前首先准备一个多层次的源目录，其树状结果如下图所示：



```
cheung@cheung:~/Desktop/code$ ls
cpp dirsync.sh py source syn_used
cheung@cheung:~/Desktop/code$ tree source/
source/
├── d1
│   ├── d11
│   │   ├── d111
│   │   │   ├── f111
│   │   │   ├── f112
│   │   │   └── f113
│   ├── f11
│   │   ├── f12
│   │   └── f13
└── d2
    ├── f21
    ├── f1
    ├── f2
    ├── f3
    └── f4
    └── test
5 directories, 11 files
cheung@cheung:~/Desktop/code$ |
```

然后调用脚本，可以看到脚本创建了对应的目标目录，并且将源目录的多层次结构复制了下来，**与预期相符**。

```
cheung@cheung:~/Desktop/code$ bash dirsSync.sh source backup
The backup directory does not exist and is created!
Synchronization completes!
cheung@cheung:~/Desktop/code$ tree backup/
backup/
└── d1
    ├── d11
    │   ├── d111
    │   │   ├── f111
    │   │   ├── f112
    │   │   └── f113
    │   ├── f11
    │   ├── f12
    │   └── f13
    └── d2
        └── f21
            ├── f1
            ├── f2
            ├── f3
            └── f4
            └── test
5 directories, 11 files
cheung@cheung:~/Desktop/code$
```

紧接着再抽取文件查看内容，也可以看到文件的内容是一致的，与预期相符。如下图所示：

```
cheung@cheung:~/Desktop/code$ tree backup/
backup/
└── d1
    ├── d11
    │   ├── d111
    │   │   ├── f111
    │   │   ├── f112
    │   │   └── f113
    │   ├── f11
    │   ├── f12
    │   └── f13
    └── d2
        └── f21
            ├── f1
            ├── f2
            ├── f3
            └── f4
            └── test
5 directories, 11 files
cheung@cheung:~/Desktop/code$ cat source/f4
test
cheung@cheung:~/Desktop/code$ cat backup/f4
test
cheung@cheung:~/Desktop/code$ |
```

将源目录中名为test的子目录删除，再创建名为test的文件。如下图，可以看到源目录下的test由目录变为了普通文件。

```
cheung@cheung:~/Desktop/code$ rmdir source/test
cheung@cheung:~/Desktop/code$ touch source/test
cheung@cheung:~/Desktop/code$ tree source/
source/
└── d1
    ├── d11
    │   ├── d111
    │   │   ├── f111
    │   │   ├── f112
    │   │   └── f113
    │   ├── f11
    │   ├── f12
    │   └── f13
    └── d2
        └── f21
            ├── f1
            ├── f2
            ├── f3
            └── f4
            └── test
4 directories, 12 files
cheung@cheung:~/Desktop/code$ |
```

再次调用脚本，可以看到目标目录中的test文件夹也变为了普通文件，与预期相符。

```
cheung@cheung:~/Desktop/code$ bash dirsSync.sh source backup
Synchronization completes!
cheung@cheung:~/Desktop/code$ tree backup/
backup/
├── d1
│   ├── d11
│   │   ├── d111
│   │   ├── f111
│   │   ├── f112
│   │   └── f113
│   ├── f11
│   ├── f12
│   └── f13
└── d2
    └── f21
        └── f1
        └── f2
        └── f3
        └── f4
        └── test
4 directories, 12 files
cheung@cheung:~/Desktop/code$ |
```

进行一个反向操作，把源目录中的test文件删掉，再创建一个名为test的文件夹。

```
cheung@cheung:~/Desktop/code$ rm source/test
cheung@cheung:~/Desktop/code$ mkdir source/test
cheung@cheung:~/Desktop/code$ tree source/
source/
├── d1
│   ├── d11
│   │   ├── d111
│   │   ├── f111
│   │   ├── f112
│   │   └── f113
│   ├── f11
│   ├── f12
│   └── f13
└── d2
    └── f21
        └── f1
        └── f2
        └── f3
        └── f4
        └── test
5 directories, 11 files
cheung@cheung:~/Desktop/code$ bash dirsSync.sh source backup
Synchronization completes!
```

调用脚本，可以看到目标目录中的test文件再次变回了文件夹，与预期相符。

```
cheung@cheung:~/Desktop/code$ bash dirsSync.sh source backup
Synchronization completes!
cheung@cheung:~/Desktop/code$ tree backup/
backup/
├── d1
│   ├── d11
│   │   ├── d111
│   │   ├── f111
│   │   ├── f112
│   │   └── f113
│   ├── f11
│   ├── f12
│   └── f13
└── d2
    └── f21
        └── f1
        └── f2
        └── f3
        └── f4
        └── test
5 directories, 11 files
cheung@cheung:~/Desktop/code$ |
```

最后着重测试一下同步功能，查看不同级目录下的几个文件内容，可以看出对应文件的内容都不一样，并且有的是源目录中的更新，有的是目标目录中的更新。

```
cheung@cheung:~/Desktop/code$ cat source/f1
f1
cheung@cheung:~/Desktop/code$ cat backup/f1
cheung@cheung:~/Desktop/code$ stat source/f1 | grep "Modify"
Modify: 2021-08-12 20:26:03.038519557 +0800
cheung@cheung:~/Desktop/code$ stat backup/f1 | grep "Modify"
Modify: 2021-08-12 20:23:39.287128614 +0800
cheung@cheung:~/Desktop/code$ cat source/f2
cheung@cheung:~/Desktop/code$ cat backup/f2
f2 backup
cheung@cheung:~/Desktop/code$ stat source/f2 | grep "Modify"
Modify: 2021-08-12 20:22:27.905162400 +0800
cheung@cheung:~/Desktop/code$ stat backup/f2 | grep "Modify"
Modify: 2021-08-12 20:27:27.374927607 +0800
cheung@cheung:~/Desktop/code$ cat source/d1/d11/f111
f111
cheung@cheung:~/Desktop/code$ cat backup/d1/d11/f111
f111 backup
cheung@cheung:~/Desktop/code$ stat source/d1/d11/f111 | grep "Modify"
Modify: 2021-08-12 20:27:04.565490908 +0800
cheung@cheung:~/Desktop/code$ stat backup/d1/d11/f111 | grep "Modify"
Modify: 2021-08-12 20:30:56.457408229 +0800
cheung@cheung:~/Desktop/code$ |
```

调用脚本，再次查看这几个文件的内容，可以看到源目录中的和目标目录中的文件内容都是最新的文件内容。**与预期相符。**

```
cheung@cheung:~/Desktop/code$ bash ./dirsync.sh source backup
Synchronization completes!
cheung@cheung:~/Desktop/code$ cat source/f1
f1
cheung@cheung:~/Desktop/code$ cat backup/f1
f1
cheung@cheung:~/Desktop/code$ cat source/f2
f2 backup
cheung@cheung:~/Desktop/code$ cat backup/f2
f2 backup
cheung@cheung:~/Desktop/code$ cat source/d1/d11/f111
f111 backup
cheung@cheung:~/Desktop/code$ cat backup/d1/d11/f111
f111 backup
cheung@cheung:~/Desktop/code$ |
```

源代码：

```
#!/usr/bin/bash

dirsyn(){
    # 写一个递归的程序
    # 写的时候发现好像不能直接把两个文件名定义成变量，父进程的变量值居然是子进程的
    touch "/tmp/.source_files_$3"
    touch "/tmp/.target_files_$3"
    # 获取文件名并排序
    cd "$1"
    ls | sort >"/tmp/.source_files_$3"
    cd "$2"
    ls | sort >"/tmp/.target_files_$3"
    # 这一个循环处理应该添加到仓库里的文件
    for file in $(comm -23 "/tmp/.source_files_$3" "/tmp/.target_files_$3"); do
        if [ -d $1'/'$file ]
            then
                # 该文件是一个文件夹
```

```

        # echo "a directory"
        mkdir $2'/'$file
        # 既然该文件夹是新的，说明里面的内容也没有备份过，因此直接用cp命令
        dirsyn $1'/'$file $2'/'$file ${((3+1))}
    else
        # echo "a file"
        cp -p $1'/'$file $2'/'$file
    fi
done
# 这一个循环处理应该从仓库里删除的文件
for file in $(comm -13 "/tmp/.source_files_$3" "/tmp/.target_files_$3"); do
    # echo rm $2'/'$file
    rm -fr $2'/'$file
done
# 这一个循环处理应该双向同步的文件
for file in $(comm -12 "/tmp/.source_files_$3" "/tmp/.target_files_$3"); do
    # 获取修改时间
    sourcetime=$(stat $1'/'$file -c %Y)
    targettime=$(stat $2'/'$file -c %Y)
    if [ -d $1'/'$file ]
        then
            # 源文件是一个目录
            if [ ! -d $2'/'$file ]
                then
                    # 仓库中的对应文件不是目录，文件类型不一致时把仓库里面的更新掉
                    rm $2'/'$file
                    mkdir $2'/'$file
                    dirsyn $1'/'$file $2'/'$file ${((3+1))}
            elif [ $sourcetime -lt $targettime ]; then
                # 更新源文件
                dirsyn $2'/'$file $1'/'$file ${((3+1))}
            elif [ $sourcetime -gt $targettime ]; then
                # 更新仓库文件
                dirsyn $1'/'$file $2'/'$file ${((3+1))}
        fi
    else
        # 源文件是一个文件
        if [ -d $2'/'$file ]
            then
                # 仓库中的对应文件是目录，文件类型不一致时把仓库里面的更新掉
                rmdir $2'/'$file
                cp -p $1'/'$file $2'/'$file
            elif [ $sourcetime -lt $targettime ]; then
                # 更新源目录
                cp -p $2'/'$file $1'/'$file
            elif [ $sourcetime -gt $targettime ]; then
                # 更新仓库中的目录
                cp -p $1'/'$file $2'/'$file
        fi
    fi
done
rm "/tmp/.source_files_$3"
rm "/tmp/.target_files_$3"
}

# 检查传入参数的数量
if [ $# -ne 2 ]
then

```

```

        echo "The number of variables is wrong!"
        exit 1
    fi

# 检查输入的目录是否存在
if [ ! -d "$1" ]
then
    echo "The source directory does not exist!"
    exit 1
fi
if [ ! -e "$2" ]
then
    echo "The backup directory does not exist and is created!"
    mkdir $2
fi

# 获取路径
cpath=$(pwd)
cd $1
sourcepath=$(pwd)
cd $cpath
cd $2
targetpath=$(pwd)
cd $cpath
# 调用函数
dirsyn $sourcepath $targetpath 1
echo "Synchronization completes!"
exit 0

```

## 6. 扑克牌比大小游戏

由于内容较多，另外附文档，请见扑克牌.pdf

## 7. myshell

由于内容较多，另外附文档，请见myshell.pdf

## 讨论心得

编写bash脚本时：

- 风格
  - 由于是第一次写大量的bash脚本，因此在编写记到题目时候我能明显感觉到自己写代码的风格在变化。比如if...else...语句的写法，经过几次变换，最后我比较倾向于使用以下的写法：

```

if [ ... ]; then
...
else
...
fi

```

- 递归的变量问题

- 在用bash脚本写递归程序时，我发现脚本似乎并不会为每一层调用的函数创建变量。当我写的递归函数调用自身时，下层的函数执行完后，上层的函数变量竟然是下层函数的值。这是脚本与一般编程语言的一个不同之处，它似乎是一个脚本只保存一个同名变量，并且所有变量都是“全局变量”。

写程序：

- 进程
  - 进程的问题主要就是 `fork()` 与 `system()` 造成的。在写程序时，有相当一段时间我的信号处理函数进入了僵尸进程，无法正常返回主进程。后来仔细了解了更对进程相关的知识，又在多处利用 `ps -1` 命令查看进程的具体信息，这才发现了隐蔽的多个进程以及僵尸进程的问题。由此才解决了进程的问题。
  - 如果我的判断无误，则更好的方式是直接调用 `execvp()`，对这一个函数创建的一个进程进行管理，更加精确也更加快速。
- 库函数
  - Linux提供了很多有用的库函数，加以利用可以很方便地完成编程任务。如在实现`unset`命令时，当时我只知道有获取环境变量值的 `getenv()` 函数和设置环境变量值的 `setenv()` 函数，我一度想用将值设为空值的形式来实现`unset`。然后根据 `setenv()` 名字试着用了 `unsetenv()`，发现确实存在这么一个函数并且作用确实是删除环境变量。知道这个函数后，`unset`命令的实现变得更好更快。

总得来说，其实许多命令的实现，借助于Linux丰富的库函数并不难实现。我觉得在实现myshell时最难的地方在于要深刻理解重定向、进程、信号等概念，并且能够利用相关的函数实现编程。本次作业让我能够更熟练地编写Linux脚本，也让我对许多系统层面的概念有了更深入的理解。