参数获取

1. 获取用户命令行输入：

```
#!/bin/bash

echo "Please enter your name:"
read name
echo "Hello, $name!"
```

2. 获取参数个数：

```
#!/bin/bash

echo "There are $# arguments."
```

3. 获取所有参数：

```
#!/bin/bash

for arg in "$@"
do
  echo $arg
done
```

4. 获取脚本名：

```
#!/bin/bash

echo "This script is called $0."
```

5. 获取当前用户：

```
#!/bin/bash

echo "The current user is $(whoami)."
```

6. 获取当前时间戳：

```
#!/bin/bash

echo "The current timestamp is $(date +%s)."
```

7. 获取当前路径：

```
#!/bin/bash

echo "The current path is $(pwd)."
```

8. 获取文件名：

```
#!/bin/bash

filename="/path/to/file.txt"

echo "The filename is $(basename $filename)."
```

9. 获取文件扩展名：

```
#!/bin/bash

filename="/path/to/file.txt"

echo "The file extension is $(echo $filename | awk -F . '{print $NF}')."
```

10. 获取文件大小：

```
#!/bin/bash

filename="/path/to/file.txt"

echo "The file size is $(stat -c %s $filename) bytes."
```

算数运算

1. 加法运算：

```
a=10
b=5
c=$((a + b))
echo $c
```

输出结果为：15

2. 减法运算：

```
a=10
b=5
c=$((a - b))
echo $c
```

输出结果为：5

3. 乘法运算：

```
a=10
b=5
c=$((a * b))
echo $c
```

输出结果为：50

4. 除法运算：

```
a=10
b=5
c=$((a / b))
echo $c
```

输出结果为：2

5. 取余运算：

```
a=10
b=3
c=$((a % b))
echo $c
```

输出结果为：1

6. 自增运算：

```
a=10
((a++))
echo $a
```

输出结果为：11

7. 自减运算：

```
a=10
((a--))
echo $a
```

输出结果为：9

8. 浮点数加法运算：

```
a=3.14
b=2.5
c=$(echo "$a + $b" | bc)
echo $c
```

输出结果为：5.64

9. 浮点数减法运算：

```
a=3.14
b=2.5
c=$(echo "$a - $b" | bc)
echo $c
```

输出结果为：0.64

10. 浮点数乘法运算：

```
a=3.14
b=2.5
c=$(echo "$a * $b" | bc)
echo $c
```

输出结果为：7.85

注意：在进行浮点数运算时，需要使用 `bc` 命令。

11. 浮点数除法

```
#!/bin/bash

# 浮点数除法
a=3.14
b=2.0
result=$(echo "scale=2; $a / $b" | bc)
echo $result   # 输出 1.57
```

逻辑判断

在 Shell 脚本中，可以使用多种方式进行逻辑判断。以下是常用的逻辑判断条件的总结：

1. 使用 `[ ]` 进行条件判断

可以使用 `[ ]` 来进行条件判断，其中条件表达式可以使用比较运算符、字符串比较运算符、文件判断运算符等。例如：

```
if [ $num -gt 10 ]
then
    echo "The number is greater than 10."
fi
```

2. 使用 `[[ ]]` 进行条件判断

`[[ ]]` 是 Bash 中的一种条件判断语法，它类似于 `[ ]`，但是提供了更多的功能。例如，`[[ ]]` 支持模式匹配和正则表达式，还支持逻辑运算符 `&&` 和 `||`，以及数值比较运算符 `<`、`>`、`<=`、`>=` 等。例如：

```
if [[ $str == hello* && $num -lt 10 ]]
then
    echo "The string starts with hello and the number is less than 10."
fi
```

3. 使用 `(( ))` 进行数值比较

`(( ))` 是 Bash 中用于进行数值比较的语法，支持数值比较运算符 `<`、`>`、`<=`、`>=` 等。例如：

```
if (( $num > 10 ))
then
    echo "The number is greater than 10."
fi
```

下面是常用例子：`-eq` 表示等于，`-ne` 表示不等于，`-lt` 表示小于，`-le` 表示小于等于，`-gt` 表示大于，`-ge` 表示大于等于。`==` 表示字符串相等，`!=` 表示字符串不相等，`-e` 表示文件存在，`-r` 表示文件可读，`-f` 表示文件存在且是一个普通文件，`-d` 表示文件存在且是一个目录，`-w` 表示文件存在且可写，-z`表示字符串为空，`-n` ` 表示字符串不为空，

1. 等于

```
#!/bin/bash
a=1
b=2
if [ $a -eq $b ]
then
    echo "a is equal to b"
else
    echo "a is not equal to b"
fi
```

2. 不等于

```
#!/bin/bash
a=1
b=2
if [ $a -ne $b ]
then
    echo "a is not equal to b"
else
    echo "a is equal to b"
fi
```

3. 小于

```
#!/bin/bash
a=1
b=2
if [ $a -lt $b ]
then
    echo "a is less than b"
else
    echo "a is greater than or equal to b"
fi
```

4. 小于等于

```
#!/bin/bash
a=1
b=2
if [ $a -le $b ]
then
    echo "a is less than or equal to b"
else
    echo "a is greater than b"
fi
```

5. 大于

```
#!/bin/bash
a=1
b=2
if [ $a -gt $b ]
then
    echo "a is greater than b"
else
    echo "a is less than or equal to b"
fi
```

6. 大于等于

```
#!/bin/bash
a=1
b=2
if [ $a -ge $b ]
then
    echo "a is greater than or equal to b"
else
    echo "a is less than b"
fi
```

7. 字符串相等

```
#!/bin/bash
str1="hello"
str2="world"
if [[ $str1 == $str2 ]]
then
    echo "str1 is equal to str2"
else
    echo "str1 is not equal to str2"
fi
```

8. 字符串不相等

```
#!/bin/bash
str1="hello"
str2="world"
if [[ $str1 != $str2 ]]
then
    echo "str1 is not equal to str2"
else
    echo "str1 is equal to str2"
fi
```

9. 文件存在

```
#!/bin/bash
if [ -e "/etc/passwd" ]
then
    echo "/etc/passwd exists"
else
    echo "/etc/passwd does not exist"
fi
```

10. 文件可读

```
#!/bin/bash
if [ -r "/etc/passwd" ]
then
    echo "/etc/passwd is readable"
else
    echo "/etc/passwd is not readable"
fi
```

11. 字符串为空

```
#!/bin/bash
str=""
if [ -z "$str" ]
then
    echo "str is empty"
else
    echo "str is not empty"
fi
```

12. 字符串不为空

```
#!/bin/bash
str="hello"
if [ -n "$str" ]
then
    echo "str is not empty"
else
    echo "str is empty"
fi
```

13. 文件存在且是一个普通文件

```
#!/bin/bash
file="/etc/passwd"
if [ -f "$file" ]
then
    echo "$file exists and is a regular file"
else
    echo "$file does not exist or is not a regular file"
fi
```

14. 文件存在且是一个目录

```
#!/bin/bash
dir="/etc"
if [ -d "$dir" ]
then
    echo "$dir exists and is a directory"
else
    echo "$dir does not exist or is not a directory"
fi
```

15. 文件存在且可写

```
#!/bin/bash
file="/etc/passwd"
if [ -w "$file" ]
then
    echo "$file exists and is writable"
else
    echo "$file does not exist or is not writable"
fi
```

16. 多个条件的逻辑与

```
#!/bin/bash
a=1
b=2
c=3
if [ $a -lt $b ] && [ $b -lt $c ]
then
    echo "a is less than b and b is less than c"
else
    echo "condition is not satisfied"
fi
```

17. 多个条件的逻辑或

```
#!/bin/bash
a=1
b=2
c=3
if [ $a -lt $b ] || [ $b -gt $c ]
then
    echo "a is less than b or b is greater than c"
else
    echo "condition is not satisfied"
fi
```

循环

for循环

1. 使用 `for var in list` 循环

这是Shell脚本中最常用的 `for` 循环语法，用于遍历数组或者列表中的元素。语法如下：

```
for var in list
do
    # 循环体
done
```

其中 `list` 可以是一个数组或者一个由空格分隔的字符串列表。在循环体中，变量 `$var` 表示当前循环到的元素。

示例代码：

```
#!/bin/bash

# 定义一个数组
fruits=(apple banana orange)

# 使用 for var in list 循环输出数组中的元素
for fruit in "${fruits[@]}"
do
    echo "The current fruit is: $fruit"
done
```

运行脚本，可以看到如下输出：

```
The current fruit is: apple
The current fruit is: banana
The current fruit is: orange
```

2. 使用 `for (( expr1; expr2; expr3 ))` 循环

这种语法常用于需要精确控制循环次数的场合，或者需要逆序遍历数组的情况。语法如下：

```
for (( expr1; expr2; expr3 ))
do
    # 循环体
done
```

其中 `expr1` 表示循环变量的初始化表达式，`expr2` 表示循环条件表达式，`expr3` 表示循环变量的变化表达式。在循环体中可以使用 `$i` 表示当前循环变量的值。

示例代码：

```
#!/bin/bash

# 使用 for (( expr1; expr2; expr3 )) 循环输出变量 i 的值
for (( i=0; i<5; i++ ))
do
    echo "The value of i is: $i"
done
```

运行脚本，可以看到如下输出：

```
The value of i is: 0
The value of i is: 1
The value of i is: 2
The value of i is: 3
The value of i is: 4
```

3. 使用 `for var in $(command)` 循环

这种语法用于执行一个命令，然后将命令的输出作为列表进行循环。语法如下：

```
for var in $(command)
do
    # 循环体
done
```

其中 `command` 是一个Shell命令，它的输出会作为列表进行循环，变量 `$var` 表示当前循环到的元素。

示例代码：

```
#!/bin/bash

# 使用 for var in $(command) 循环读取当前目录下的所有文件
for file in $(ls)
do
    echo "The current file is: $file"
done
```

运行脚本，可以看到如下输出：

```
The current file is: file1.txt
The current file is: file2.txt
The current file is: script.sh
```

4. 使用 `for var in {start..end}` 循环

这种语法用于生成一个从 `start` 到 `end` 的整数序列进行循环。语法如下：

```
for var in {start..end}
do
    # 循环体
done
```

其中 `start` 和 `end` 分别表示序列的起始值和结束值，变量 `$var` 表示当前循环到的元素。

示例代码：

```
#!/bin/bash

# 使用 for var in {start..end} 循环输出整数序列
for i in {1..5}
do
    echo "The value of i is: $i"
done
```

运行脚本，可以看到如下输出：

```
The value of i is: 1
The value of i is: 2
The value of i is: 3
The value of i is: 4
The value of i is: 5
```

5. 使用for遍历文件

```
#!/bin/bash

# 指定要遍历的目录
dir="/path/to/directory"

# 使用通配符 * 遍历目录下的所有文件，并使用 for 循环遍历文件列表
for file in "$dir"/*
do
    echo "The current file is: $file"
done
```

6. 使用for遍历字符串

```
string="hello world"

for (( i=0; i<${#string}; i++ ))
do
    char="${string:$i:1}"
    echo "The current character is: $char"
done
```

判断最大:

```
#!/bin/bash
max=$1
for arg in "$@"; do
  if [ "$arg" -gt "$max" ]; then
    max=$arg
  fi
done
echo "最大值为：$max"
```

while循环

1. 遍历目录中的所有文件

```
#!/bin/bash

# 遍历指定目录下的所有文件，并输出文件列表
# read -r: 禁止对反斜杠字符进行转义
while IFS= read -r file
do
    echo "The current file is: $file"
done < <(find /path/to/dir -type f)
```

2. 逐行读取命令的输出，并处理输出结果

```
#!/bin/bash

# 打印当前系统中所有正在运行的进程的PID和名称
ps -ef | while read pid pname
do
    echo "PID: $pid, Name: $pname"
done
```

在处理命令输出时，可以使用管道来将一个命令的输出作为另一个命令的输入，例如：

```
command1 | command2
```

这个命令会将 `command1` 命令的输出作为 `command2` 命令的输入。因此，在(1)用法中，也可以使用管道来将 `find` 命令的输出作为 `while` 循环的输入，例如：

```
find /path/to/dir -type f | while IFS= read -r file
do
    echo "The current file is: $file"
done
```

然而，这种写法存在一个问题，就是 `while` 循环会在子 Shell 中执行，而不是在当前 Shell 中执行。具体来说，当使用管道时，左边的命令会在一个子 Shell 中执行，而右边的命令也会在一个子 Shell 中执行，因此，`while` 循环也会在一个子 Shell 中执行。这意味着，在循环体中定义的变量只能在子 Shell 中使用，而无法传递到父 Shell 中。

为了避免这个问题，可以使用进程替换语法，将 `find` 命令的输出作为文件传递给 `while` 循环，而不是使用管道。具体来说，这段代码使用了进程替换语法：

```
while IFS= read -r file
do
    echo "The current file is: $file"
done < <(find /path/to/dir -type f)
```

这种写法可以保证 `while` 循环在当前 Shell 中执行，可以正确传递变量。因此，在处理命令输出时，如果需要在循环体中使用变量，建议使用进程替换语法，而不是管道。

3. 从文件中读取多个参数，并执行命令

```
#!/bin/bash

# 从参数文件中读取多个参数，并执行命令
while read arg1 arg2 arg3
do
    echo "arg1: $arg1, arg2: $arg2, arg3: $arg3"
    # 执行命令，使用读取的参数
    command $arg1 $arg2 $arg3
done < args.txt
```

从文件 `args.txt` 中读取多个参数，并使用 `while` 循环逐个处理参数。在循环体中，使用 `$arg1`、`$arg2`、`$arg3` 变量分别表示读取到的每个参数，然后执行命令，使用读取到的参数。

4. 循环等待用户输入

```
#!/bin/bash

while true
do
  read -p "Enter op: " op
  if [ -z "$op" ]; then
    echo "Name is required."
  elif [ $op = 'n' ]
    break
  else
    read -p "Enter your name: " name
    "$name" >> file
    echo "Hello, $name!"
  fi
done
```

在循环体中，使用 `read -p` 命令提示用户输入姓名，并使用 `$name` 变量保存用户输入的姓名。然后使用 `if` 语句判断用户输入的姓名是否为空，如果不为空，则输出欢迎信息并退出循环。

5. 获取命令行参数

```
#!/bin/bash
```

```
while [[ $# -gt 0 ]]
do
  key="$1"
  case $key in
    -f|--file)
      file="$2"
      shift
      shift
      ;;
    -d|--directory)
      dir="$2"
      shift
      shift
      ;;
    *)
      shift
      ;;
  esac
done
```

分支

if

1. 简单的比较运算符

使用比较运算符可以对两个值进行比较，并根据比较结果执行不同的代码块。以下是常用的比较运算符，以及它们的含义：

- `-eq`：等于
- `-ne`：不等于
- `-gt`：大于
- `-ge`：大于等于
- `-lt`：小于
- `-le`：小于等于

例如：

```
if [ $num -gt 10 ]
then
    echo "The number is greater than 10."
fi
```

2. 字符串比较运算符

使用字符串比较运算符可以对两个字符串进行比较，并根据比较结果执行不同的代码块。以下是常用的字符串比较运算符，以及它们的含义：

- `=`：相等
- `!=`：不相等

- `-z`：为空字符串
- `-n`：非空字符串

例如：

```
if [ "$str1" = "$str2" ]
then
    echo "The two strings are equal."
fi
```

3. **文件判断运算符**

使用文件判断运算符可以对文件进行判断，并根据判断结果执行不同的代码块。以下是常用的文件判断运算符，以及它们的含义：

- `-e`：文件存在
- `-f`：普通文件存在
- `-d`：目录存在
- `-r`：文件可读
- `-w`：文件可写
- `-x`：文件可执行
- -c：**是否为字符设备**

例如：

```
if [ -d "$dir" ]
then
    echo "The directory exists."
fi
```

4. 逻辑运算符

使用逻辑运算符可以将多个条件进行组合，并根据组合结果执行不同的代码块。以下是常用的逻辑运算符，以及它们的含义：

- `&&`：逻辑与
- `||`：逻辑或
- `!`：逻辑非

例如：

```
if [ -f "$file" ] && [ -r "$file" ]
then
    echo "The file is readable."
fi
```

5. 复合条件判断

可以使用复合条件判断来组合多个条件，并根据条件组合结果执行不同的代码块。以下是常用的复合条件判断，以及它们的含义：

- `if ... elif ... else ... fi`：多个条件判断

- `case ... esac`：多个条件判断，类似于 switch 语句

例如：

```
if [ "$num" -eq 1 ]
then
    echo "The number is 1."
elif [ "$num" -eq 2 ]
then
    echo "The number is 2."
else
    echo "The number is neither 1 nor 2."
fi
```

```
case "$str" in
    "hello")
        echo "The string is hello."
        ;;
    "world")
        echo "The string is world."
        ;;
    *)
        echo "The string is neither hello nor world."
        ;;
esac
```

6. 如果条件是一个命令

Bash 会根据命令的退出状态来判断条件的真假。如果命令的退出状态是 0，表示命令执行成功，条件为真；否则命令执行失败，条件为假。

```
#!/bin/bash

if grep "hello" file.txt &> /dev/null
then
  echo "hello is found in file.txt"
else
  echo "hello is not found in file.txt"
fi

#!/bin/bash
# 判断 grep 是否找到了匹配的字符串
result=$(grep "pattern" file.txt)
if [ -n "$result" ]; then
    echo "Found: $result"
else
    echo "Not found"
fi
```

`&> /dev/null`是一种重定向语法，在Linux shell中用于将命令的输出和错误信息都重定向到/dev/null设备文件中，从而达到忽略命令的输出和错误信息的目的。具体解释如下：

1. `>`符号用于将命令的输出重定向到指定文件中，例如`command > file.txt`表示将command命令的输出写入到file.txt文件中。

2. `2>` 符号用于将命令的错误信息重定向到指定文件中，例如 `command 2> error.txt` 表示将command命令的错误信息写入到error.txt文件中。
3. `&>` 符号则是将命令的输出和错误信息都重定向到指定文件中，例如 `command &> output.txt` 表示将command的输出和错误信息都写入到output.txt文件中。
4. `/dev/null` 是一个特殊的设备文件，它是一个黑洞，任何写入该文件的数据都会被丢弃，而不会被存储。因此，将命令的输出和错误信息重定向到/dev/null中，就可以达到忽略命令的输出和错误信息的效果。

判断文件是否是字符设备并复制文件

```
#!/bin/bash

echo "Please enter the file path:"
read FILE_PATH

if [[ ! -e "$FILE_PATH" ]]; then
    echo "This file does not exist."
    exit 1
fi

if [[ ! -c "$FILE_PATH" ]]; then
    echo "This file is not a character device file."
    exit 1
fi

cp "$FILE_PATH" /dev/
echo "The file has been copied to /dev/"
```

case

成绩判断

- ?|[1-5]?：小于60的
- 9?|100：90-100的

```
echo -e "Please enter the score:"
while read SCORE
do
    case $SCORE in
        ?|[1-5]? ) echo "Failed!"
        echo "Please enter the next score:";;
        6?) echo "Passed!"
        echo "Please enter the next score:";;
        7?) echo "Medium!"
        echo "Please enter the next score:";;
        8?) echo "Good!"
        echo "Please enter the next score:";;
        9?|100) echo "Great!"
        echo "Please enter the next score:";;
        *) exit;;
```

```
    esac
done
```

```
#!/bin/bash

echo "Please enter a string:"
while read STRING
do
case $STRING in
"hello" | "world" ) echo "Matched 'hello' or 'world'!"
echo "Please enter the next string:";;
[0-9][0-9][0-9]) echo "Matched a three-digit number!"
echo "Please enter the next string:";;
[a-zA-Z]+@[a-zA-Z]+\.[a-zA-Z]{2,3}) echo "Matched an email address!"
echo "Please enter the next string:";;
[A-Z][a-z]*) echo "Matched a capitalized word!"
echo "Please enter the next string:";;
*) echo "No match found."
exit;;
esac
done
```

字符串操作

1. 获取字符串长度：

```
#!/bin/bash

str="hello world"

echo ${#str}  # 输出 11
```

2. 截取子串：

```
#!/bin/bash

str="hello world"

echo ${str:0:5}  # 输出 hello
```

3. 查找子串：

```
#!/bin/bash

str="hello world"

if [[ $str == *"world"* ]]
then
  echo "world found in str"
else
  echo "world not found in str"
fi
```

4. 替换子串：

```
#!/bin/bash

str="hello world"

echo ${str/world/earth}  # 输出 hello earth
```

5. 转换为大写：

```
#!/bin/bash

str="hello world"

echo ${str^^}  # 输出 HELLO WORLD
```

6. 转换为小写：

```
#!/bin/bash

str="HELLO WORLD"

echo ${str,,}  # 输出 hello world
```

7. 去掉前缀：

```
#!/bin/bash

str="hello world"

echo ${str#hello}  # 输出 world
```

8. 去掉后缀：

```
#!/bin/bash

str="hello.txt"

echo ${str%.txt}  # 输出 hello
```

9. 拼接字符串：

```
#!/bin/bash

str1="hello"
str2="world"

echo $str1$str2  # 输出 helloworld
```

10. 去掉空格：

```
#!/bin/bash

str="  hello world  "

echo ${str}  # 输出 "  hello world  "
echo ${str// /}  # 输出 helloworld
```

数组操作

- 输入
- 遍历
- 长度

```
#!/bin/bash

# 读取用户输入的10个数字
echo "请输入10个数字，以空格分隔："
read -ra nums

# 计算最大值和平均值
max=${nums[0]}
sum=0
for num in "${nums[@]}"; do
  if ((num > max)); then
    max=$num
  fi
  ((sum += num))
done
avg=$(bc -l <<<"$sum/${#nums[@]}")

# 输出结果
echo "最大值为：$max"
echo "平均值为：$avg"
```

在Shell脚本中，数组是一种特殊类型的变量，可以容纳多个值。以下是Shell数组的常见操作：

1. 定义数组

定义数组的语法如下：

```
array_name=(value1 value2 ... valuen)
```

其中，`array_name`表示数组的名称，`value1`、`value2`、...、`valuen`表示数组中的元素。可以使用空格或制表符将元素分隔开。

示例：

```
fruits=(apple banana orange)
```

2. 读取数组中的元素

读取数组中的元素可以使用如下语法：

```
${array_name[index]}
```

其中，`array_name`表示数组的名称，`index`表示元素的下标。下标从0开始。读取数组中的元素要加上花括号。

示例：

```
echo ${fruits[0]}   # 输出：apple
echo ${fruits[1]}   # 输出：banana
echo ${fruits[2]}   # 输出：orange
```

3. 获取数组中的所有元素

获取数组中的所有元素可以使用如下语法：

```
${array_name[@]}
```

其中，`array_name`表示数组的名称。

示例：

```
echo ${fruits[@]}   # 输出：apple banana orange
```

4. 获取数组中的元素个数

获取数组中的元素个数可以使用如下语法：

```
${#array_name[@]}
```

其中，`array_name`表示数组的名称。

示例：

```
echo ${#fruits[@]}   # 输出：3
```

5. 添加元素到数组中

添加元素到数组中可以使用如下语法：

```
array_name+=(value)
```

其中，`array_name`表示数组的名称，`value`表示要添加的元素。

示例：

```
fruits+=(grape)
echo ${fruits[@]}   # 输出：apple banana orange grape
```

6. 删除数组中的元素

删除数组中的元素可以使用如下语法：

```
unset array_name[index]
```

其中，`array_name` 表示数组的名称，`index` 表示要删除的元素的下标。

示例：

```
unset fruits[1]
echo ${fruits[@]}   # 输出：apple orange
```

7. 遍历数组

遍历数组可以使用 `for` 循环，如下所示：

```
for i in ${array_name[@]}
do
    echo $i
done
```

其中，`array_name` 表示数组的名称，`$i` 表示循环变量，表示数组中的元素。

示例：

```
for i in ${fruits[@]}
do
    echo $i
done
# 输出：
# apple
# orange
```

函数操作

1. 创建函数：

```
#!/bin/bash

my_function() {
  echo "Hello from my_function"
}

my_function  # 调用函数
```

2. 带参数的函数：

```
#!/bin/bash

my_function() {
  echo "Hello, $1 and $2"
}

my_function Alice Bob  # 调用函数并传递两个参数
```

3. 函数返回值：

```
#!/bin/bash

my_function() {
  return 42
}

my_function
echo "The return value is $?"
```

4. 获取函数参数个数：

```
#!/bin/bash

my_function() {
  echo "There are $# arguments"
}

my_function Alice Bob Charlie
```

5. 参数数组：

```
#!/bin/bash

my_function() {
  args=("$@")
  echo "The second argument is ${args[1]}"
}

my_function Alice Bob Charlie
```

6. 全局变量：

```
#!/bin/bash

my_function() {
  global_var="Hello from my_function"
}

global_var="Hello from main script"
echo $global_var
my_function
echo $global_var
```

7. 局部变量：

```
#!/bin/bash

my_function() {
  local local_var="Hello from my_function"
  echo $local_var
}

local_var="Hello from main script"
echo $local_var
my_function
echo $local_var
```

8. 函数嵌套：

```
#!/bin/bash

outer_function() {
  inner_function() {
    echo "Hello from inner_function"
  }
  inner_function
}

outer_function
```

```
#!/bin/bash

my_function() {
  for arg in "$@"
  do
    echo $arg
  done
}

caller_function() {
  func=$1
  shift # 参数向左移动一位，去除my_function，把Alice Bob Charlie传给$func
  $func "$@"
}
```

```
caller_function my_function Alice Bob Charlie
```

9. 函数作为参数：

```
#!/bin/bash

my_function() {
  echo "Hello, $1"
}

caller_function() {
  func=$1
  name=$2
  $func $name
}

caller_function my_function Alice
```

10. 函数作为返回值：

```
#!/bin/bash

my_function() {
  echo "Hello, $1"
}

caller_function() {
  echo my_function
}

func=$(caller_function)
$name="Alice"
$func $name
```

Shell文件统计综合

对指定目录下所有文件的大小进行排序，并输出前 10 大的文件名和大小，同时将结果保存到文件中。

```
#!/bin/bash

# 从命令行参数中读取目录名和输出文件名
directory=$1
output_file=$2

# 定义函数，使用 du 命令计算文件大小，并输出文件名和大小
calculate_size() {
  for file in $1/*
  do
    if [ -f "$file" ]
    then
      size=$(du -h "$file" | cut -f 1)
```

```
      echo "$size $file"
    elif [ -d "$file" ]
    then
      calculate_size "$file"
    fi
  done
}

# 调用函数 calculate_size，计算目录下所有文件的大小，并按大小排序
result=$(calculate_size "$directory" | sort -hr)

# 输出前 10 大的文件名和大小
echo "$result" | head -n 10

# 将结果保存到文件中
echo "$result" > "$output_file"
```

统计指定目录下所有文件的行数，并输出行数最多的前五个文件名和行数：

```
#!/bin/bash

# 从命令行参数中读取目录名称
directory=$1

# 定义函数，统计文件的行数，并输出文件名和行数
count_lines() {
  for file in $1/*
  do
    if [ -f "$file" ]
    then
      count=$(wc -l < "$file")
      echo "$count $file"
    elif [ -d "$file" ]
    then
      count_lines "$file"
    fi
  done
}

# 调用函数 count_lines，计算目录下所有文件的行数
result=$(count_lines "$directory")

# 使用 sort 命令按行数排序，并输出前五个文件名和行数
echo "$result" | sort -hr | head -n 5
```

从命令行参数中读取两个文件名，比较它们的内容是否相同，并输出比较结果：

```
#!/bin/bash

# 从命令行参数中读取两个文件名称
file1=$1
file2=$2

# 比较文件内容是否相同
if cmp -s "$file1" "$file2"
then
  echo "The files are the same"
else
  echo "The files are different"
fi
```

统计指定目录下所有文件的单词数，并输出单词数最多的前五个文件名和单词数：

```
#!/bin/bash

# 从命令行参数中读取目录名称
directory=$1

# 定义函数，统计文件的单词数，并输出文件名和单词数
count_words() {
  for file in $1/*
  do
    if [ -f "$file" ]
    then
      count=$(wc -w < "$file")
      echo "$count $file"
    elif [ -d "$file" ]
    then
      count_words "$file"
    fi
  done
}

# 调用函数 count_words，计算目录下所有文件的单词数
result=$(count_words "$directory")

# 使用 sort 命令按单词数排序，并输出前五个文件名和单词数
echo "$result" | sort -hr | head -n 5
```

- while循环
- if分支
- switch分支
- 接收输入参数（read）
- 文件输入（>>）
- 从文件中匹配满足要求的一行（grep）
- 获取一行字符串中的某几项（cut/awk）
- 指定分割字符，选定分割字段，从大到小排序（sort）
- 取前n条结构（head）

- 替换某个文件中的某个字段，或者删除某个值（sed）

```
#!/bin/bash

# 定义学生信息文件路径
students_file="./students.txt"

# 定义函数，添加学生信息
add_student() {
  read -p "Please enter the student's name: " name
  read -p "Please enter the student's grade: " grade
  # 创建文件/尾加
  echo "$name,$grade" >> "$students_file"
  echo "Student added successfully"
}

add_students(){
    while true
    do
        read -p "y/n" flag
        [[ $flag == "n" ]] && break
        read -p "Please enter the student's name: " name
        if [ -z "$name" ]
        then
            echo "name is empty"
            continue
        fi
        exist=$(grep "^$name," "$students_file")
        if [ -z "$exist" ]
        then
            echo "exist"
            continue
        fi
        read -p "Please enter the student's grade: " grade
        if [ -z "$grade" ]
        then
            echo "grade is empty"
            continue
        fi
        echo "$name,$grade" >> "$students_file"
        echo "add $name successfully!"
    done
}

# 定义函数，查询学生信息
query_student() {
  read -p "Please enter the student's name: " name
  grade=$(grep "^$name," "$students_file" | cut -d ',' -f 2)
  if [ -n "$grade" ]
  then
    echo "Student found: $name, $grade"
  else
    echo "Student not found"
  fi
}
```

```
# 定义函数，查询所有学生信息
query_all_students() {
  # 使用 awk 命令格式化输出
  local result=$(awk -F ',' '{printf "%-20s %-10s\n", $1, $2}' "$students_file")
  echo "$result"
}

# 定义函数，计算成绩均值
get_avg_scores() {
  # 使用 awk 命令计算指定文件中指定字段的均值，并将结果赋值给 avg 变量
  local avg=$(awk -F ',' '{sum += $2} END {if (NF > 0) print sum/NF}'
"$students_file")
  # 输出均值结果
  echo "AVG SCORE: $avg"
}


get_max_scores() {
  # 初始化最大值和学生信息变量
  max=0
  stu=""

  # 使用重定向符号将文件内容作为输入传递给循环
  while IFS=',' read -r -a fields; do
    # 获取第 1 和第 3 个字段
    name="${fields[0]}"
    grade="${fields[2]}"

    # 比较成绩大小，更新最大值和对应的学生信息
    if [ "$grade" -gt "$max" ]; then
      max="$grade"
      stu="$name"
    fi
  done < "$students_file"

  # 输出最大值和学生信息
  echo "MAX: $max STU: $stu"
}


# 定义函数，输出分数最大的十个学生的姓名和成绩
get_top10_stu() {
    // 指定分割字符，选定分割字段，从大到小排序
    sort -t ',' -k 2 -r "$students_file" | head -n 10
}

# 定义函数，修改学生信息
update_student() {
  read -p "Please enter the student's name: " name
  grade=$(grep "^$name," "$students_file" | cut -d ',' -f 2)
  if [ -n "$grade" ]
  then
    read -p "Please enter the new grade for $name: " new_grade
    sed -i "s/^$name,$grade$/$name,$new_grade/" "$students_file"
    echo "Student updated successfully"
```

```
  else
    echo "Student not found"
  fi
}

# 定义函数，删除学生信息
delete_student() {
  read -p "Please enter the student's name: " name
  grade=$(grep "^$name," "$students_file" | cut -d ',' -f 2)
  if [ -n "$grade" ]
  then
    sed -i "/^$name,$grade$/d" "$students_file"
    echo "Student deleted successfully"
  else
    echo "Student not found"
  fi
}

# 主程序循环
while :
do
  echo "Please select an option:"
  echo "1. Add a student"
  echo "2. Query a student"
  echo "3. Update a student"
  echo "4. Delete a student"
  echo "5. Exit"
  read -p "Your choice: " choice
  case $choice in
    1) add_student;;
    2) query_student;;
    3) update_student;;
    4) delete_student;;
    5) exit;;
    *) echo "Invalid choice";;
  esac
done
```

1. `read` 指令

`read` 指令用于从标准输入读取用户输入的值。它的基本语法如下：

```
read [options] [variable...]
```

其中 `options` 是可选的参数，可以用于指定读取值的一些选项，如 `-p` 用于显示提示信息，`-r` 用于保留输入值中的转义字符等。`variable` 是变量名，用于存储读取的值。

下面是一个读取用户输入并将其存储到变量中的例子：

```
read -p "Please enter your name: " name
echo "Your name is: $name"
```

该例子中，`-p` 选项用于显示提示信息，`name` 变量用于存储用户输入的值。程序将提示用户输入姓名，并将其存储到 `name` 变量中，然后输出变量的值。

2. `echo` 指令

`echo` 指令用于输出文本。它的基本语法如下：

```
echo [options] [string(s)]
```

其中 `options` 是可选的参数，可以用于指定输出文本的一些选项，如 `-e` 用于解析特殊字符等。`string(s)` 是需要输出的文本，可以是一个或多个字符串。如果有多个字符串，它们会被空格分隔。

下面是一个输出文本的例子：

```
echo "Hello, world!"
```

该例子中，程序将输出字符串 "Hello, world!"。

3. `grep` 指令

`grep` 指令用于在文件中查找匹配的文本。它的基本语法如下：

```
grep [options] pattern [file(s)]
```

其中 `options` 是可选的参数，可以用于指定一些选项，如 `-i` 用于忽略大小写，`-v` 用于反向匹配等。`pattern` 是需要匹配的文本，可以是一个字符串或正则表达式。`file(s)` 是需要查找的文件名，可以是一个或多个文件。如果没有指定文件名，则默认从标准输入中查找文本。

下面是一个在文件中查找匹配文本的例子：

```
grep "apple" fruits.txt
```

该例子中，程序将在 `fruits.txt` 文件中查找匹配字符串 "apple" 的行，并将其输出。

4. `cut` 指令

`cut` 指令用于从文本中提取字段。它的基本语法如下：

```
cut [options] [file(s)]
```

其中 `options` 是可选的参数，可以用于指定一些选项，如 `-d` 用于指定字段分隔符，`-f` 用于指定需要提取的字段等。`file(s)` 是需要提取字段的文件名，可以是一个或多个文件。如果没有指定文件名，则默认从标准输入中读取文本。

下面是一个从文本中提取字段的例子：

```
cut -d ',' -f 2 student.txt
```

该例子中，程序将从 `student.txt` 文件中提取以逗号为分隔符的第 2 个字段，并将其输出。

5. `sed` 指令

`sed` 指令用于流编辑器，可用于对文件或输出流进行编辑操作，如替换、删除、添加或者插入等操作。它的基本语法如下：

```
sed [options] 'command' [file(s)]
```

其中 `options` 是可选的参数，可以用于指定一些选项，如 `-i` 用于直接修改文件等。`command` 是需要执行的编辑命令，可以是一个或多个命令。`file(s)` 是需要编辑的文件名，可以是一个或多个文件。下面是一个用 sed 命令进行替换的例子：

```
sed 's/apple/orange/g' fruits.txt
```

该例子中，程序将在 `fruits.txt` 文件中将所有的 "apple" 替换为 "orange" 并输出结果。

6. `awk` 指令

`awk` 指令用于文本处理和数据分析，可用于格式化输出、计算统计值等操作。它的基本语法如下：

```
awk [options] 'pattern { action }' [file(s)]
```

其中 `options` 是可选的参数，可以用于指定一些选项，如 `-F` 用于指定字段分隔符等。`pattern` 是需要匹配的模式，可以是一个正则表达式，也可以是一个条件语句。`action` 是需要执行的操作，可以是输出、计算统计值等操作。`file(s)` 是需要处理的文件名，可以是一个或多个文件。如果没有指定文件名，则默认从标准输入中读取文本。

下面是一个使用 awk 计算平均值的例子：

```
awk '{sum+=$2} END {print "Average: " sum/NR}' scores.txt
```

该例子中，程序将从 `scores.txt` 文件中读取每行的第 2 个字段，并计算它们的平均值并输出结果。

7. `printf` 指令

`printf` 指令用于格式化输出。它的基本语法如下：

```
printf [format] [arguments...]
```

其中 `format` 是格式化字符串，可以包含特定的格式占位符，如 `%s` 用于字符串，`%d` 用于整数等。`arguments` 是需要输出的参数，可以是一个或多个参数。

下面是一个使用 printf 进行格式化输出的例子：

```
printf "Name: %s, Age: %d\n" "Tom" 18
```

该例子中，程序将格式化输出字符串 "Name: Tom, Age: 18"。

以上是程序中用到的一些重点指令的简单讲解和例子示范，希望能够帮助理解。需要注意的是，这些指令的具体用法和选项较多，可以参考相应的帮助文档或者其他相关资源进行深入学习。

```
# 获取参数
filename="$1"

add_student() {
    read -p "input sid: " sid
    read -p "input score: " score
    exist=$(grep "^$sid," "$filename")
    if [[ -z "${exist}" ]]
    then
```

```
        echo "$sid,$exist" >> "$filename"
        echo "add successful !"
    else
        echo "has exist!"
    fi
}

query_student() {
    read -p "input sid: " sid
    score=$(grep "^$sid," "$filename")
    if [[ -z "${score}" ]]
    then
        echo "wrong"
    else
        echo "query $sid with score $score"
    fi
}

update_student() {
    read -p "input sid: " sid
    read -p "input new score: " nscore
    score=$(grep "^$sid," "$filename" | cut -d ',' -f 2)
    sed -i "s/^$sid,$score$/$sid,$nscore/" "$filename"
    echo "update successful!"
}

delete_student() {
    read -p "input sid: " sid
    score=$(grep "^$sid," "$filename" | cut -d ',' -f 2)
    sed -i "/^$sid,$score$/d" "$filename"
    echo "delete successful!"
}

# 查询操作
while true
do
    read -p "input mode: " mode
    case $mode in
        1) add_student;;
        2) query_student;;
        3) update_student;;
        4) delete_student;;
        5) exit;;
        *) echo "Invalid choice";;
    esac
done
```

常用括号

- (): 在另一个shell中执行指令，返回结果加$
    - $(ls -l)

- (()): 四则运算；整数比较；里面使用和c++一样的语法, 有返回值时外部才能加$, 里面变量不能加$
- - var3=$((var1+var2))
- []: 和(())一样四则运算；测试数字/字符串/文件时必须左右加空格，一般和if一起用
- - var3=$[var1+var2]
  - if [var1 -ge var2 -a var1 -l var3]
- [[]]: 和[]类似，但是除了-ne和-ge其他比较都能用数学符号替代
- - if [[var1 -ge var2 && var1 < var3]]
- ``: 命令执行
- - `ls -l`
  - `expr $a + $b`

变量调用：

- 方法(1)：${var}
- 方法(2)：$var

命令调用：

- 方法(1)：`COMMAND`
- 方法(2)：$(COMMAND)

测试表达式: **=两侧有空格**

- 方法(1)：[expression]
- 方法(2)：[[expression]]
- 方法(3)：test expression

算术运算: **=两侧没有空格**

- 方法(1)：let 算术运算表达式
  - let C=A+B
  - let C=$A+$B
- 方法(2)：$[算术运算表达式]
  - C=$[A+B]
  - C=$[A+$B]
- 方法(3)：$((算术运算表达式))
  - C=$((A+B))
  - C=$((A+$B))
- 方法(4)： **必须有空格，必须有$**，而且要使用命令引用,遇到乘法的*号还要转义.
  - C=`expr $A + $B` #参考单反引号的用法

方法(5)：expr $[算术运算表达式]，遇到乘法*号不用转义。

文件操作（复制，移动，创建，删除，权限，用户组）

1. 复制文件或目录

```
# 复制文件
cp file1.txt file2.txt

# 复制目录
cp -r dir1 dir2
```

2. 移动或重命名文件或目录

```
# 移动文件或目录
mv file1.txt /path/to/new/location
mv dir1 /path/to/new/location

# 重命名文件或目录
mv file1.txt file2.txt
mv dir1 dir2
```

3. 删除文件或目录

```
# 删除文件
rm file1.txt

# 删除目录及其所有内容
rm -r dir1
```

4. 创建新文件或目录

```
# 创建新文件
touch file1.txt

# 创建新目录
mkdir dir1
```

5. 查看文件内容

```
# 查看文件内容
cat file1.txt

# 查看文件前几行
head file1.txt

# 查看文件后几行
tail file1.txt
```

6. 改变文件所有者和所属组

```
# 改变文件所有者
sudo chown user1 file1.txt

# 改变文件所属组
sudo chgrp group1 file1.txt

# 同时改变文件所有者和所属组
sudo chown user1:group1 file1.txt
```

7. 修改文件权限

```
# 添加用户读权限
chmod u+r file1.txt

# 添加用户写权限
chmod u+w file1.txt

# 添加用户执行权限
chmod u+x file1.txt

# 添加所有用户读权限
chmod a+r file1.txt

# 添加所有用户写权限
chmod a+w file1.txt

# 添加所有用户执行权限
chmod a+x file1.txt

# 移除所有用户执行权限
chmod a-x file1.txt
```

如果您想将文件的权限授予文件所属组的用户，可以使用 `chmod` 命令的 `g` 和 `+/-/=` 标志来更改文件的组权限。下面是一些示例：

1. 添加组读权限

```
chmod g+r file1.txt
```

2. 添加组写权限

```
chmod g+w file1.txt
```

3. 添加组执行权限

```
chmod g+x file1.txt
```

4. 同时添加所有用户的组权限

```
chmod a+rwx file1.txt
```

5. 移除组执行权限

```
chmod g-x file1.txt
```

注意，这些命令将更改文件的组权限。如果您不知道文件所属组的名称，可以使用 `ls -l` 命令查看文件的详细信息，第四列将显示文件的所属组名称。例如：

```
ls -l file1.txt
-rw-r--r-- 1 user1 group1 0 Jun 14 09:00 file1.txt
```

在上面的示例中，文件 `file1.txt` 的所属组是 `group1`。如果想将文件的组权限授予该组的所有用户，可以使用上面的 `chmod` 命令。

常用指令

ls -l输出

`ls -l` 命令用于列出当前目录下的所有文件和子目录，并显示它们的详细信息。下面是 `ls -l` 命令输出内容的详解：

```
-rw-r--r--  1 user1 group1  4096 Jun 13 10:23 example.txt
drwxr-xr-x  2 user1 group1  4096 Jun 13 10:23 subdirectory
```

输出内容分为几个部分，每个部分代表一个文件或子目录的详细信息，各部分之间用换行符分隔。下面是每个部分的详细解释：

1. 文件类型和权限标识符：

第一个字符表示文件的类型，常见的文件类型包括：

- `-`：普通文件
- `d`：目录
- `l`：符号链接
- `c`：字符设备文件
- `b`：块设备文件
- `s`：套接字（socket）文件
- `p`：命名管道（FIFO）

接下来的九个字符表示文件的权限，分为三组，每组三个字符，分别代表文件所有者、文件所属组和其他用户的权限。每个字符可以是以下字符之一：

- `r`：读权限
- `w`：写权限
- `x`：执行权限
- `-`：没有相应的权限

例如，`-rw-r--r--` 表示文件所有者有读写权限，文件所属组和其他用户只有读权限。

2. 硬链接数：

第二列数字表示这个文件的硬链接数。硬链接是一种文件系统中的链接方式，它允许一个文件拥有多个文件名。当创建一个硬链接时，新链接与原文件具有相同的 inode 号，即它们实际上是同一个文件。

3. 文件所有者和文件所属组：

第三列和第四列分别表示文件的所有者和文件所属组。例如，`user1` 是文件所有者，`group1` 是文件所属组。

4. 文件大小：

第五列数字表示文件的大小，以字节为单位。

5. 修改时间：

第六列字符串表示文件的最后修改时间。在上面的例子中，`Jun 13 10:23` 表示文件最后修改时间是在 6 月 13 日上午 10 点 23 分。

6. 文件名：

最后一列是文件名或子目录名。例如，`example.txt` 是一个文件，`subdirectory` 是一个子目录。

```
ls -l | cut -d ' ' -f 1,2,5,9
```

find文件操作

`find` 常用操作：

- `find [path] -name [pattern]`：在指定路径 `[path]` 下查找文件名匹配 `[pattern]` 的文件。
- `find [path] -type [type]`：在指定路径 `[path]` 下查找类型为 `[type]` 的文件，可以是 `f`（普通文件）、`d`（目录）、`l`（符号链接）等。
- `find [path] -mtime [n]`：在指定路径 `[path]` 下查找修改时间在 `[n]` 天前的文件。
- `find [path] -size [n]`：在指定路径 `[path]` 下查找文件大小为 `[n]` 的文件。
- `find [path] -exec [command] {} \;`：在指定路径 `[path]` 下执行指定命令 `[command]`，将匹配的文件名替换 `{}` 占位符。

`grep` 常用操作：

- `grep [pattern] [file]`：在指定文件 `[file]` 中查找匹配 `[pattern]` 的行。
- `grep -r [pattern] [path]`：在指定路径 `[path]` 下递归查找匹配 `[pattern]` 的行。
- `grep -i [pattern] [file]`：在指定文件 `[file]` 中查找不区分大小写的匹配 `[pattern]` 的行。
- `grep -v [pattern] [file]`：在指定文件 `[file]` 中查找不匹配 `[pattern]` 的行。
- `grep -c [pattern] [file]`：在指定文件 `[file]` 中统计匹配 `[pattern]` 的行数。

`find` 和 `grep` 混合操作：

- `find [path] -name [pattern] -exec grep [pattern] {} \;`：在指定路径 `[path]` 下查找文件名匹配 `[pattern]` 的文件，并在每个文件中查找匹配 `[pattern]` 的行。
- `find [path] -type [type] -mtime [n] -exec grep [pattern] {} \;`：在指定路径 `[path]` 下查找类型为 `[type]` 且修改时间在 `[n]` 天前的文件，并在每个文件中查找匹配 `[pattern]` 的行。

查找目录\a下所有包含字符串“b”的c语言程序

```
find /a -name "*.c" -o -name "*.h" -type f -exec grep -q "b" {} \; -print
```

- -type f: 只查找文件，过滤文件夹
- -q：安静过滤，不输出
- -iq：不区分大小写+安静过滤
- {}：占位符，把find的文件名传入其中，当做是grep的参数
- \; 指令结束
- -printf：find命令的结果输出

路径包含空格，需要使用引号

```
find "/path/to/dir with spaces" -name "*.txt" -type f
```

wc统计文件信息

```
#!/bin/bash

if [ $# -ne 1 ]; then
  echo "Usage: $0 filename"
  exit 1
fi

if [ ! -f "$1" ]; then
  echo "Error: $1 is not a file"
  exit 1
fi

lines=$(wc -l "$1")
words=$(wc -w "$1")

echo "The file $1 has $lines lines and $words words."
```

mount挂载设备

1. 查看可用的设备

```
sudo fdisk -l
```

2. 挂载 U 盘

```
sudo mount -t vfat /dev/sdb1 /mnt/usb
```

这里假设 U 盘的设备名称为 `/dev/sdb1`，挂载点为 `/mnt/usb`。如果挂载点不存在，则需要先创建挂载点：

```
sudo mkdir /mnt/usb
```

3. 查看已挂载的文件系统

```
mount
```

4. 卸载 U 盘

```
sudo umount /mnt/usb
```

如果 U 盘正在使用，则需要先关闭所有正在使用 U 盘的应用程序，然后再执行卸载操作。如果无法卸载，则可以使用 `-l` 选项强制卸载：

```
sudo umount -l /mnt/usb
```

注意，以上命令中的 `sudo` 是为了获得足够的权限来执行挂载和卸载操作。如果您没有管理员权限，则需要使用管理员账户或者 `su` 命令来切换到管理员账户。

grep查找子串

以下是一些常用的正则表达式用法和示例：

1. 匹配指定字符串：

```
grep "string" file.txt
```

2. 匹配以指定字符串开头的行：

```
grep "^string" file.txt
```

3. 匹配以指定字符串结尾的行：

```
grep "string$" file.txt
```

4. 匹配包含指定字符串的行：

```
grep ".*string.*" file.txt
```

5. 匹配以指定字符集开头的行：

```
grep "^[a-z]" file.txt
```

6. 匹配以指定字符集结尾的行：

```
grep "[0-9]$" file.txt
```

7. 匹配指定字符集中的任意一个字符：

```
grep "[aeiou]" file.txt
```

8. 匹配指定字符集中的任意一个字符之外的字符：

```
grep "[^aeiou]" file.txt
```

9. 匹配指定字符集中的一个或多个字符重复出现：

```
grep "a+" file.txt
```

10. 匹配指定字符集中的零个或多个字符重复出现：

```
grep "a*" file.txt
```

在学生管理系统的 `query_student`、`update_student` 和 `delete_student` 函数中，使用了 `grep` 命令来查找包含指定学生姓名的行。具体而言，使用了以下命令：

```
grep "^$name," "$students_file"
```

其中，`^` 表示行的开头，`$name` 表示要查找的学生姓名，`,` 表示分隔符。因此，`^$name,` 表示以指定学生姓名开头、后面跟随一个逗号的行。`"$students_file"` 则表示要查找的文本文件。

这个命令的作用是在文本文件中查找包含指定学生姓名的行，并输出这些行。例如，假设我们有一个文本文件 `students.txt`，其中包含以下内容：

```
Alice,90
Bob,85
Charlie,95
```

如果我们使用 `grep "^Bob," "students.txt"` 命令，则可以输出包含 Bob 的行：

```
Bob,85
```

在学生管理系统中，我们使用类似的命令来查找包含指定学生姓名的行，并提取对应的学生成绩。如果找到了对应学生的成绩，则可以进行修改或删除操作，否则输出“Student not found”。

cut和awk获取某几项

如果要从一个字符串中匹配某几个项，并将它们赋值为新的变量，可以使用 `cut` 或 `awk` 命令将字符串分割成字段，然后使用变量来存储每个字段的值。

例如，如果要从一个使用逗号分隔的字符串中匹配第 1 和第 3 个字段，可以使用 `cut` 命令：

```
# 原始字符串
str="apple,banana,orange,grape"

# 使用 cut 命令获取第 1 和第 3 个字段
field1=$(echo "$str" | cut -d ',' -f 1)
field3=$(echo "$str" | cut -d ',' -f 3)

# 输出结果
echo "Field 1: $field1"
echo "Field 3: $field3"
```

在这个例子中，使用 `cut` 命令将字符串分割成逗号分隔的字段，并使用 `-f` 选项指定要获取的字段编号。然后，将每个字段的值存储在变量 `$field1` 和 `$field3` 中，并输出这些变量的值。

类似地，如果要从一个使用逗号分隔的字符串中匹配多个字段，并将它们赋值为数组，可以使用 `awk` 命令：

```
# 原始字符串
str="apple,banana,orange,grape"

# 使用 awk 命令获取多个字段并存储到数组中
fields=($(echo "$str" | awk -F ',' '{print $1, $3}'))

# 输出结果
echo "Field 1: ${fields[0]}"
echo "Field 2: ${fields[1]}"

# 使用 awk 命令获取多个字段，并存储到数组中
fields=$(awk -F ',' '{print $1, $3}' < "$filename")

# 输出结果
echo "Field 1: ${fields[0]}"
echo "Field 3: ${fields[1]}"
```

在这个例子中，使用 `awk` 命令将字符串分割成逗号分隔的字段，并使用大括号 `{}` 中的命令来指定要获取的字段编号，然后将这些字段的值存储在名为 `$fields` 的数组中。最后，使用 `${fields[0]}` 和 `${fields[1]}` 访问数组中的值，并输出它们。

好的，下面我会详细讲解一些在程序中用到的重点指令，并提供一些例子来帮助理解。

tr替换输入流

正则表达式的使用

1. `.`：匹配任意一个字符，除了换行符。
2. `*`：匹配前面的字符出现零次或多次。例如，`a*` 表示匹配所有以零个或多个 `a` 开头的字符串。
3. `+`：匹配前面的字符出现一次或多次。例如，`a+` 表示匹配所有以一个或多个 `a` 开头的字符串。
4. `?`：匹配前面的字符出现零次或一次。例如，`a?` 表示匹配所有以零个或一个 `a` 开头的字符串。

5. `[]`：表示字符集，匹配方括号中的任意一个字符。例如，`[abc]` 表示匹配所有包含字符 `a`、`b` 或 `c` 的字符串。
6. `[^]`：表示字符集的补集，匹配除了方括号中的字符以外的任意一个字符。例如，`[^abc]` 表示匹配所有不包含字符 `a`、`b` 或 `c` 的字符串。
7. `-`：表示字符集的范围，匹配方括号中指定的字符范围内的任意一个字符。例如，`[a-z]` 表示匹配所有小写字母。
8. `()`：表示分组，在正则表达式中可以使用分组来改变优先级、重复次数等属性。例如，`(ab)+` 表示匹配所有由一个或多个 `ab` 组成的字符串。
9. `|`：表示逻辑或，匹配两个或多个表达式中的任意一个。例如，`a|b` 表示匹配所有包含 `a` 或 `b` 的字符串。
10. `^`：表示字符串的开头，匹配以指定字符或字符集开头的字符串。例如，`^a` 表示匹配所有以字母 `a` 开头的字符串。
11. `$`：表示字符串的结尾，匹配以指定字符或字符集结尾的字符串。例如，`a$` 表示匹配所有以字母 `a` 结尾的字符串。

以下是在Shell中使用正则表达式的一些例子：

1. `grep` 命令：
   - 匹配包含字符串 "hello" 的行：`grep 'hello' file.txt`
   - 匹配以字母 "a" 开头的行：`grep '^a' file.txt`
   - 匹配以数字结尾的行：`grep '[0-9]$' file.txt`
   - 匹配包含两个连续的数字的行：`grep '[0-9][0-9]' file.txt`
   - 匹配包含单词 "world" 的行，忽略大小写：`grep -i 'world' file.txt`
2. `sed` 命令：
   - 将所有的数字替换为 "X"：`sed 's/[0-9]/X/g' file.txt`
   - 将所有以字母 "a" 开头的行删除：`sed '/^a/d' file.txt`
   - 将所有行的第一个数字替换为 "X"：`sed 's/[0-9]/X/' file.txt`
   - 将所有行的最后一个数字替换为 "X"：`sed 's/[0-9]$/X/' file.txt`
   - 将所有包含单词 "hello" 的行替换为 "world"：`sed 's/hello/world/' file.txt`
3. 双括号表达式 `[[ ... ]]`：
   - 判断变量 `$str` 是否包含字符串 "hello"：`[[ "$str" == *hello* ]]`
   - 判断变量 `$num` 是否为一个正整数：`[[ "$num" =~ ^[1-9][0-9]*$ ]]`
   - 判断变量 `$path` 是否为一个绝对路径：`[[ "$path" =~ ^/ ]]`
   - 判断变量 `$url` 是否为一个合法的URL：`[[ "$url" =~ ^https?://[a-z0-9.-]+/[a-z0-9./-]*$ ]]`
   - 判断变量 `$email` 是否为一个合法的邮箱地址：`[[ "$email" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]`
4. `expr` 命令：
   - 判断变量 `$num` 是否为一个正整数：`expr "$num" : '^[1-9][0-9]*$' > /dev/null`
   - 判断变量 `$path` 是否为一个绝对路径：`expr "$path" : '^/' > /dev/null`

- 判断变量 `$url` 是否为一个合法的URL：`expr "$url" : '^https\?://[a-z0-9.-]\+/[a-z0-9./-]\+$' > /dev/null`
- 判断变量 `$email` 是否为一个合法的邮箱地址：`expr "$email" : '^[a-zA-Z0-9._%+-]\+@[a-zA-Z0-9.-]\+\.[a-zA-Z]\{2,\}$' > /dev/null`