

管道相关命令

目标

- `cut`
- `sort`
- `wc`
- `uniq`
- `tee`
- `tr`
- `split`
- `awk`
- `sed`

- 准备工作

`vim score.txt`

```
zhangsan 68 99 26
lisi 98 66 96
wangwu 38 33 86
zhaoliu 78 44 36
maq 88 22 66
zhouba 98 44 46
```

- 以上是成绩表
- 使用 逗号 分割, 第一列 是 姓名, 第二列是 语文成绩, 第三列是 数学成绩, 第四列是 英语成绩
- 需求1: 按照 数学成绩排名, 取出前三名
- 需求2: 显示 学生的数学成绩
- 需求3: 显示 数学平均分
- 需求4: 如何将大文件 切割成 若干小文件

- 准备工作

`vim 1.txt`

```
111:aaa:bbb:ccc
222:ddd:eee:fff
333:ggg:hhh
444:iii
```

1 cut

1.1 目标

- `cut` 根据条件 从命令结果中 **提取** 对应内容

1.2 路径

- 第一步: 截取出1.txt文件中前2行的第5个字符
- 第二步: 截取出1.txt文件中前2行以":"进行分割的第1,2段内容(方式一)
- 第三步: 截取出1.txt文件中前2行以":"进行分割的第1,2,3段内容(方式二)

1.3 实现

第一步: 截取出1.txt文件中前2行的第5个字符

命令	含义
cut 动作 文件	从指定文件 截取内容

- 参数

参数	英文	含义
-c	characters	按字符选取内容

答案:

```
head -2 1.txt | cut -c 5
```

第二步: 截取出1.txt文件中前2行以":"进行分割的第1,2段内容

参数	英文	含义
<code>-d '分隔符'</code>	delimiter	指定分隔符
<code>-f n1,n2</code>	fields	分割以后显示第几段内容, 使用 <code>,</code> 分割

范围控制

范围	含义
<code>n</code>	只显示第n项
<code>n-</code>	显示 从第n项 一直到行尾
<code>n-m</code>	显示 从第n项 到 第m项(包括m)

答案:

```
head -2 1.txt | cut -d ':' -f 1,2
```

或者

```
head -2 1.txt | cut -d ':' -f 1-2
```

第三步: 截取出1.txt文件中前2行以":"进行分割的第1,2,3段内容

答案:

```
head -2 1.txt | cut -d ':' -f 1,2,3
```

或者

```
head -2 1.txt | cut -d ':' -f 1-3
```

1.4 小结

- 通过 `cat` 动作 目标文件 可以根据条件 提取对应内容

- 准备工作

vim score.txt

```
zhangsan 68 99 26
lisi 98 66 96
wangwu 38 33 86
zhaoliu 78 44 36
maq 88 22 66
zhouba 98 44 46
```

2 sort 的工作原理

2.1 目标

- sort可针对文本文件的内容，以行为单位来排序。

2.2 路径

- 第一步: 对字符串排序
- 第二步: 去重排序
- 第三步: 对数值排序
- 第四步: 对成绩排序

2.3 实现

第一步: 对字符串排序

```
[root@node01 tmp]# cat 2.txt
banana
apple
pear
orange
pear

[root@node01 tmp]# sort 2.txt
apple
banana
orange
pear
pear
```

第二步: 去重排序

参数	英文	含义
<code>-u</code>	unique	去掉重复的

它的作用很简单，就是在输出行中去除重复行。

```
[root@node01 tmp]# sort -u 2.txt
apple
banana
orange
pear
```

第三步：对数值排序

参数	英文	含义
<code>-n</code>	numeric-sort	按照数值大小排序
<code>-r</code>	reverse	使次序颠倒

- 准备数据

```
[root@node01 tmp]# cat 3.txt
1
3
5
7
11
2
4
6
10
8
9
```

- 默认按照 `字符串` 排序

```
[root@node01 tmp]# sort 2.txt
1
10
11
2
3
4
5
6
7
8
9
```

- 升序

```
[root@node01 tmp]# sort -n 2.txt
1
2
3
4
5
6
7
8
9
10
11
```

- 倒序

```
[root@node01 tmp]# sort -n -r 2.txt
11
10
9
8
7
6
5
4
3
2
1
```

- 合并式

```
[root@node01 tmp]# sort -nr 2.txt
11
10
9
8
7
6
5
4
3
2
1
```

第四步: 对成绩排序

参数	英文	含义
<code>-t</code>	field-separator	指定字段分隔符
<code>-k</code>	key	根据那一列排序

```
# 根据第二段成绩 进行倒序显示 所有内容
sort -t ',' -k2nr score.txt
```

2.4 小结

- 通过 `sort` 选项 文件 可以对文件内容进行排序

3 wc命令

3.1 目标

- 显示指定文件 字节数, 单词数, 行数 信息.

3.2 路径

- 第一步: 显示指定文件 字节数, 单词数, 行数 信息.
- 第二步: 只显示 文件 的行数
- 第三步: 统计多个文件的 行数 单词数 字节数
- 第四步: 查看 `/etc` 目录下 有多少个 子内容

3.3 实现

第一步: 显示指定文件 字节数, 单词数, 行数 信息.

命令	含义
wc 文件名	显示指定文件 字节数, 单词数, 行数 信息

```
[root@hadoop01 export]# cat 4.txt
111
222 bbb
333 aaa bbb
444 aaa bbb ccc
555 aaa bbb ccc ddd
666 aaa bbb ccc ddd eee

[root@hadoop01 export]# wc 4.txt
 6 21 85 3.txt
```

第二步: 只显示 文件 的行数

参数	英文	含义
<code>-c</code>	bytes	字节数
<code>-w</code>	words	单词数
<code>-l</code>	lines	行数

```
[root@hadoop01 export]# wc 4.txt
 6 21 85 3.txt
```

第三步: 统计多个文件的 行数 单词数 字节数


```
[root@hadoop01 export]# wc 1.txt 2.txt 3.txt
 4   4  52 1.txt
11  11  24 2.txt
 6  21  85 3.txt
21  36 161 总用量

[root@hadoop01 export]# wc *.txt
 4   4  52 1.txt
11  11  24 2.txt
 6  21  85 3.txt
 6   6  95 score.txt
27  42 256 总用量
```

第四步: 查看 `/etc` 目录下有多少个子内容

```
[root@hadoop01 export]# ls /etc | wc -w
240
```

3.4 小结

- 通过 `wc 文件` 就可以统计文件的字节数、单词数、行数。

4 uniq

`uniq` 命令用于检查及删除文本文件中重复出现的行，一般与 `sort` 命令结合使用。

4.1 目标

- `uniq` 命令用于检查及删除文本文件中重复出现的行，一般与 `sort` 命令结合使用。

4.2 路径

- 第一步：实现去重效果
- 第二步：不但去重，还要统计出现的次数

4.3 实现

第一步：实现去重效果

命令	英文	含义
<code>uniq [参数] 文件</code>	unique 唯一	去除重复行

```
# 准备内容
[root@hadoop01 export]# cat 5.txt
张三    98
李四    100
王五    90
赵六    95
麻七    70
李四    100
王五    90
赵六    95
麻七    70

# 排序
[root@hadoop01 export]# cat 5.txt | sort
李四    100
李四    100
麻七    70
麻七    70
王五    90
王五    90
张三    98
赵六    95
赵六    95

# 去重
[root@hadoop01 export]# cat 5.txt | sort | uniq
李四    100
麻七    70
王五    90
张三    98
赵六    95
```

第二步：不但去重，还要 统计出现的次数

参数	英文	含义
<code>-c</code>	count	统计每行内容出现的次数

```
[root@hadoop01 export]# cat 5.txt | sort | uniq -c
      2 李四      100
      2 麻七      70
      2 王五      90
      1 张三      98
      2 赵六      95
```

4.4 小结

- 通过 `uniq [选项] 文件` 就可以完成 去重行 和 统计次数

5 tee

5.1 目标

- 通过 `tee` 可以将命令结果 **通过管道** 输出到 **多个文件**中

5.2 实现

命令	含义
命令结果 tee 文件1 文件2 文件3	通过 <code>tee</code> 可以将命令结果 通过管道 输出到 多个文件 中

- 将去重统计的结果 放到 a.txt、b.txt、c.txt 文件中**

```
cat 5.txt | sort | uniq -c | tee a.txt b.txt c.txt
```

5.3 小结

- 通过 `tee` 可以将命令结果 **通过管道** 输出到 **多个文件**中

6 tr

6.1 目标

- 通过 `tr` 命令用于 **替换** 或 **删除** 文件中的字符。

6.2 路径

- 第一步: 实现 **替换** 效果
- 第二步: 实现 **删除** 效果
- 第三步: 完成 **单词计数** 案例

6.3 实现

第一步: 实现 替换效果

命令	英文	含义
命令结果 tr 被替换的字符 新字符	translate	实现 替换效果

```
# 将 小写i 替换成 大写 I
# 把itheima的转换为大写
# 把 HELLO 转成 小写
```

```
# 将 小写i 替换成 大写 I
echo "itheima" | tr 'i' 'I'

# 把itheima的转换为大写
echo "itheima" |tr '[a-z]' '[A-Z]'

# 把 HELLO 转成 小写
echo "HELLO" |tr '[A-Z]' '[a-z]'
```

第二步: 实现删除效果

命令	英文	含义
命令结果 tr -d 被删除的字符	delete	删除指定的字符

- 需求: 删除abc1d4e5f中的数字

```
echo 'abc1d4e5f' | tr -d '[0-9]'
```

第三步: 单词计数

准备工作

```
[root@hadoop01 export]# cat words.txt
hello,world,hadoop
hive,sqoop,flume,hello
kitty,tom,jerry,world
hadoop
```

1 将, 换成 换行

2 排序

3 去重

4 计数

```
# 统计每个单词出现的次数
[root@hadoop01 export]# cat words.txt | tr ',' '\n' | sort | uniq -c
    1 flume
    2 hadoop
    2 hello
    1 hive
    1 jerry
    1 kitty
    1 sqoop
    1 tom
    2 world
```

6.4 小结

- 通过 `tr [选项] 字符1 字符2` 可以实现 **替换** 和 **删除** 效果

- 准备工作

```
# 查看 /etc目录下 以.conf以结尾的文件的内容
cat -n /etc/*.conf

# 将命令结果 追加到 /export/v.txt 文件中
cat -n /etc/*.conf >> /export/v.txt
```

7 split

7.1 目标

- 通过 `split` 命令将**大文件** 切分成 若干**小文件**

7.2 路径

- 第一步: 按 **字节** 将 大文件 切分成 若干小文件
- 第二步: 按 **行数** 将 大文件 切分成 若干小文件

7.3 实现

第一步: 按 字节 将 大文件 切分成 若干小文件

命令	英文	含义
<code>split -b 10k 文件</code>	byte	将大文件切分成若干 10KB 的小文件

第二步: 按 行数 将 大文件 切分成 若干小文件

命令	英文	含义
<code>split -l 1000 文件</code>	lines	将大文件切分成若干 1000行 的小文件

7.4 小结

- 通过 `split` 选项 文件名 命令将大文件 切分成 若干小文件

- 准备工作1:

vim score.txt

```
zhangsan 68 99 26
lisi 98 66 96
wangwu 38 33 86
zhaoliu 78 44 36
maq 88 22 66
zhouba 98 44 46
```

8 awk

8.1 目标

- 通过 `awk` 实现 模糊查询, 按需提取字段, 还可以进行 判断 和 简单的运算等.

8.2 步骤

- 第一步: 模糊查询
- 第二步: 指定分割符, 根据下标显示内容
- 第三步: 指定输出字段的分割符
- 第四步: 调用 awk 提供的函数
- 第五步: 通过if语句判断\$4是否及格
- 第六步: 段内容 求和

8.3 实现

第一步: 搜索 zhangsan 和 lisi 的成绩

命令	含义
awk '/zh li/' score.txt	模糊查询

第二步: 指定分割符, 根据下标显示内容

命令	含义
awk -F ' ' '{print \$1, \$2, \$3}' 1.txt	操作1.txt文件, 根据 逗号 分割, 打印 第一段 第二段 第三段 内容

选项

选项	英文	含义
<code>-F ' , '</code>	field-separator	使用 指定字符 分割
<code>\$ + 数字</code>		获取 第几段 内容
<code>\$0</code>		获取 当前行 内容
<code>NF</code>		表示当前行共有多少个字段
<code>\$NF</code>		代表 最后一个字段
<code>\$(NF-1)</code>		代表 倒数第二个字段
<code>NR</code>		代表 处理的是第几行

第三步: 指定分割符, 根据下标显示内容

命令	含义
<code>awk -F ' ' '{OFS="==="}{print \$1, \$2, \$3}' 1.txt</code>	操作1.txt文件, 根据 逗号 分割, 打印 第一段 第二段 第三段 内容

选项

选项	英文	含义
<code>OFS="字符"</code>	output field separator	向外输出时的段分割字符串

第四步: 调用 awk 提供的函数

命令	含义
<code>awk -F ' ' '{print toupper(\$2)}' 1.txt</code>	操作1.txt文件, 根据 逗号 分割, 打印 第一段 第二段 第三段 内容

常用函数如下:

函数名	含义	作用
<code>toupper()</code>	upper	字符 转成 大写
<code>tolower()</code>	lower	字符 转成小写
<code>length()</code>	length	返回 字符长度

第五步: if语句 查询及格的学生信息

命令	含义
awk -F ',' '{if(\$4>60) print \$1, \$4 }' score.txt	如果及格,就显示 \$1, \$4
awk -F ',' '{if(\$4>60) print \$1, \$4, "及格"; else print \$1, \$4, "不及格"}' score.txt	显示 姓名, \$4, 是否及格

选项

参数	含义
if(\$0 ~ "aa") print \$0	如果这一行包含 "aa", 就打印这一行内容
if(\$1 ~ "aa") print \$0	如果 第一段 包含 "aa", 就打印这一行内容
if(\$1 == "lisi") print \$0	如果 第一段 等于 "lisi", 就打印这一行内容

第六步: 段内容 求学科平均分

命令	含义
awk 'BEGIN{初始化操作}{每行都执行} END{结束时操作}' 文件名	BEGIN{ 这里面放的是执行前的语句 } {这里面放的是处理每一行时要执行的语句} END {这里面放的是处理完所有的行后要执行的语句 }

```
awk -F ',' 'BEGIN{}{total=total+$4}END{print total, NR, (total/NR)}' score.txt
```

8.4 小结

- 通过 `awk 动作 文件名` 更加灵活的解析文件.
- 准备工作
vim 1.txt

```
aaa java root
bbb hello
ccc rt
ddd root nologin
eee rtt
fff ROOT nologin
ggg rttt
```

9 sed

9.1 目标

- 通过 sed 可以实现 **过滤** 和 **替换** 的功能.

9.2 路径

- 第一步: 实现 查询 功能
- 第二步: 实现 删除 功能
- 第三步: 实现 修改 功能
- 第四步: 实现 替换 功能
- 第五步: 对 原文件 进行操作
- 第六步: 综合 练习

9.3 实现

第一步: 实现 查询 功能

命令	含义
sed 可选项 目标文件	对目标文件 进行 过滤查询 或 替换

可选参数

可选项	英文	含义
p	print	打印
\$		代表 最后一行
<code>-n</code>		仅显示处理后的结果
<code>-e</code>	expression	根据表达式 进行处理

- 练习1** 列出 1.txt的 1~5行 的数据

```
sed -n -e '1,5p' 1.txt
```

- **练习2 列出01.txt的所有数据**

```
sed -n -e '1,$p' 1.txt
```

- **练习3 列出01.txt的所有数据 且 显示行号**

可选项	含义
=	打印当前行号

```
sed -n -e '1,$=' -e '1,$p' 1.txt
```

简化版

```
cat -n 1.txt
```

```
cat -b 1.txt
```

```
nl 1.txt
```

- **练习4: 查找01.txt中包含root行**

答案:

```
sed -n -e '/root/p' 1.txt
```

- **练习5 列出01.txt中包含root的内容, root不区分大小写,并显示行号**

可选项	英文	含义
	ignore	忽略大小写

答案:

```
nl 1.txt | sed -n -e '/root/Ip'

nl 01.txt | grep -i root

cat -n 01.txt | grep -i root
```

• 练习6 查找出1.txt中 字母 **r** 后面是多个t的行，并显示行号

可选项	英文	含义
-r	regex-extended	识别正则

答案:

```
nl 01.txt | sed -nr -e '/r+t/p'
```

或者

```
sed -nr -e '/r+t/p' -e '/r+t/= ' 01.txt
```

第二步: 实现 删除 功能

• 练习1 删除01.txt中前3行数据，并显示行号

可选项	英文	含义
d	delete	删除指定内容

答案:

```
nl 01.txt | sed -e '1,3d'
```

- **练习2 保留1.txt中前4行数据，并显示行号**

答案:

```
nl 01.txt | sed -e '5,$d'

nl 1.txt | sed -n -e '1,4p'
```

第三步: 实现 修改 功能

- **练习1: 在01.txt的第二行后添加aaaaa,并显示行号**

参数	英文	含义
i	insert	目标 前面 插入内容
a	append	目标 后面 追加内容

答案:

```
nl 01.txt | sed -e '2a aaaaa'
```

- **练习2 在1.txt的第1行前添加bbbbbb，并显示行号**

答案:

```
nl 01.txt | sed -e '1i bbbbbb'
```

第四步: 实现 替换 功能

- 练习1 把1.txt中的nologin替换成为huawei,并显示行号

	英文	含义
s/oldString/newString/	replace	替换

答案:

```
nl 1.txt | sed -e 's/nologin/huawei/'
```

- 练习2 把01.txt中的1,2行替换为aaa,并显示行号

选项	英文	
2c 新字符串	replace	使用新字符串 替换 选中的行

答案:

```
nl passwd | sed -e '1,2c aaa'
```

第五步: 对 原文件 进行操作

- 练习1 在01.txt中把nologin替换为 huawei

参数	英文	含义
-i	in-place	替换原有文件内容

答案:

```
sed -i -e 's/nologin/huawei/' 01.txt
```

- **练习2 在01.txt文件中第2、3行替换为aaaaaa**

答案:

```
sed -i -e '2,3c aaa' 01.txt
```

注意: 在进行操作之前, 最好是对数据进行备份, 放置操作失误, 数据无法恢复!

- **练习3 删除01.txt中前2行数据, 并且删除原文件中的数据**

答案:

```
sed -i -e '1,2d' 01.txt
```

```
nl passwd 查看数据
```

第六步: 综合 练习

- **练习1 获取ip地址**

答案:

```
ifconfig eth0 | grep "inet addr" | sed -e 's/^.*inet addr:/' | sed -e 's/Bcast:.*$//'
```

- 练习2 从1.txt中提出数据，匹配出包含root的内容，再把nologin替换为itheima

答案：

```
nl 01.txt | grep 'root' | sed -e 's/nologin/itheima/'
```

或者

```
nl 01.txt | sed -n -e '/root/p' | sed -e 's/nologin/itheima/'
```

或者

```
nl 01.txt | sed -n -e '/root/{s/nologin/itheima/p}' #只显示替换内容的行
```

- 练习3 从1.txt中提出数据，删除前5行，并把nologin替换为itheima,并显示行号

答案：

```
nl 01.txt | sed -e '1,5d' | sed -e 's/nologin/itheima/'
```

2. Shell 编程

2.1 简介

Shell 是一个用 C 语言编写的程序，通过 Shell 用户可以访问操作系统内核服务。

Shell 既是一种命令语言，又是一种程序设计语言。

Shell script 是一种为 shell 编写的脚本程序。Shell 编程一般指 shell脚本编程，不是指开发 shell 自身。

Shell 编程跟 java、php 编程一样，只要有一个能编写代码的文本编辑器和一个能解释执行的脚本解释器就可以了。

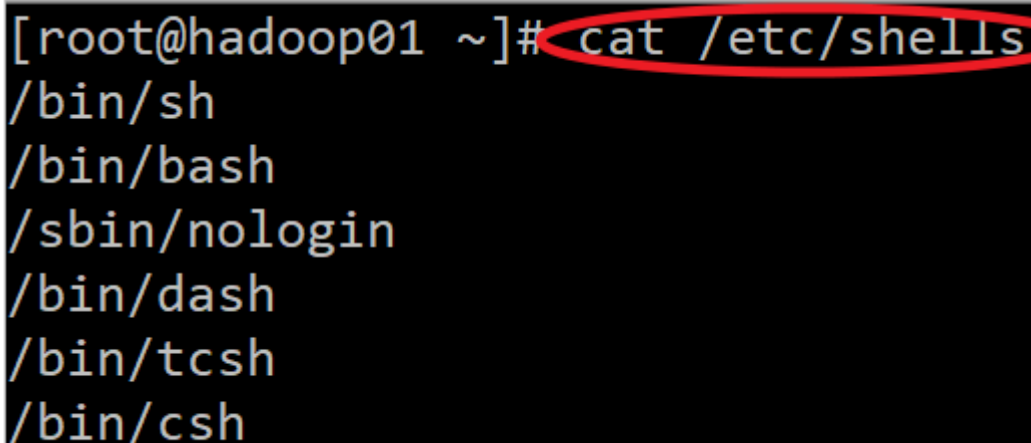
Linux 的 **Shell 解释器** 种类众多，一个系统可以存在多个 shell，可以通过 `cat /etc/shells` 命令查看系统中安装的 shell 解释器。

Bash 由于易用和免费，在日常工作中被广泛使用。同时，Bash 也是大多数 Linux 系统默认的 Shell。

shell 解释器

java 需要 虚拟机解释器, 同理 shell脚本也需要 解析器

```
[root@node04 shells]# cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/dash
/bin/tcsh
/bin/csh
```



```
[root@hadoop01 ~]# cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/dash
/bin/tcsh
/bin/csh
```

2.2快速入门

1 编写脚本

新建 `/export/hello.sh` 文件

```
#!/bin/bash

echo 'hello world'
```

`#!`是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种 Shell。

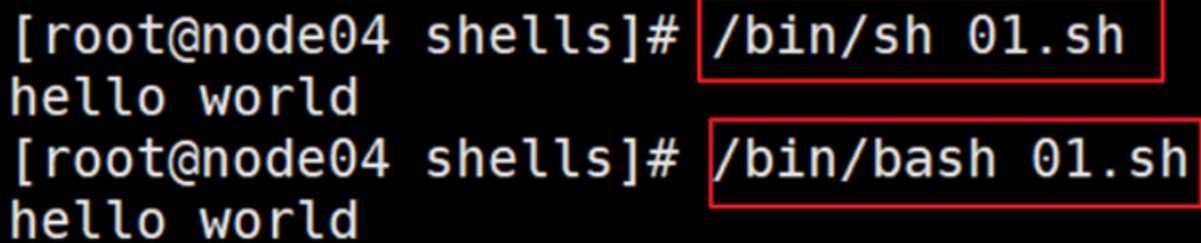
`echo` 命令用于向窗口输出文本。

2 执行shell脚本

- 执行方式一

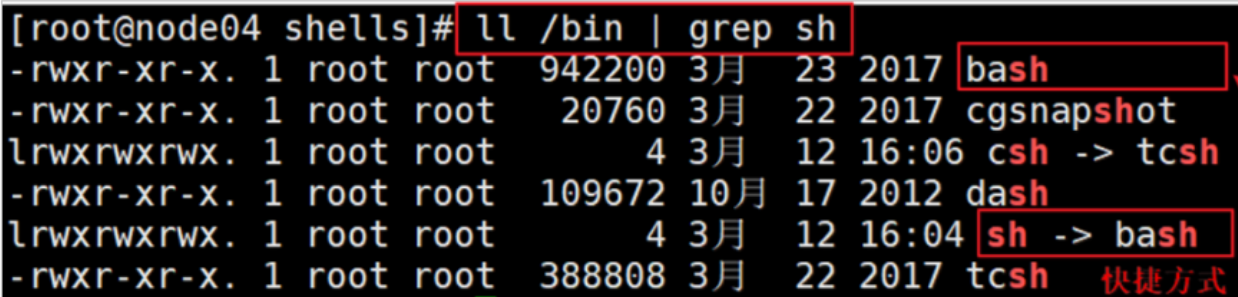
```
[root@node04 shells]# /bin/sh 01.sh
hello world

[root@node04 shells]# /bin/bash 01.sh
hello world
```



```
[root@node04 shells]# /bin/sh 01.sh
hello world
[root@node04 shells]# /bin/bash 01.sh
hello world
```

- 问题: bash 和 sh 是什么关系?



```
[root@node04 shells]# ll /bin | grep sh
-rwxr-xr-x. 1 root root 942200 3月 23 2017 bash
-rwxr-xr-x. 1 root root 20760 3月 22 2017 cgsnapshot
lrwxrwxrwx. 1 root root 4 3月 12 16:06 csh -> tcsh
-rwxr-xr-x. 1 root root 109672 10月 17 2012 dash
lrwxrwxrwx. 1 root root 4 3月 12 16:04 sh -> bash
-rwxr-xr-x. 1 root root 388808 3月 22 2017 tcsh 快捷方式
```

sh 是 bash 的 快捷方式

3.2 执行方式二

方式一的简化方式

```
[root@node04 shells]# bash hello.sh
hello world

[root@node04 shells]# sh hello.sh
hello world
```

```
[root@node04 shells]# bash 01.sh
hello world
[root@node04 shells]# sh 01.sh
hello world
```

3.2.1 问题: 请思考 为什么可以省略 /bin/

```
[root@node04 shells]# echo $PATH
/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
```

因为 PATH环境变量中增加了 /bin/目录, 所以 使用/bin/sh等类似指令时, 可以省略 /bin

3.3 执行方式三

./文件名

```
[root@node04 shells]# ll
总用量 4
-rw-r--r--. 1 root root 32 3月 14 00:20 01.sh
```

```
[root@node04 shells]# ./hello.sh
-bash: ./01.sh: 权限不够
```

3.3.1 权限不够怎么办?

```
[root@node04 shells]# chmod 755 hello.sh

[root@node04 shells]# ll
总用量 4
-rwxr-xr-x. 1 root root 32 3月 14 00:20 hello.sh

# 再次执行:
[root@node04 shells]# ./hello.sh
hello world!
```

2.3 shell变量

1 简介

在shell脚本中, 定义变量时, 变量名不加美元符号 (\$), 如:

```
your_name="runoob.com"
```

注意：**变量名和等号之间不能有空格**，这可能和你熟悉的所有编程语言都不一样。

同时，变量名的命名须遵循如下规则：

- 命名只能使用英文字母，数字和下划线，首个字符不能以数字开头。
- 中间不能有空格，可以使用下划线（`_`）。
- 不能使用标点符号。
- 不能使用bash里的关键字（可用help命令查看保留关键字）。

有效的 Shell 变量名示例如下：

```
RUNOOB
LD_LIBRARY_PATH
_var
var2
```

无效的变量命名：

```
?var=123
user*name=runoob
```

除了显式地直接赋值，还可以用语句给变量赋值，如：



```
for file in `ls /etc`
```

或

```
for file in $(ls /etc)
```

以上语句将 /etc 下目录的文件名循环出来。

2 使用变量

使用一个定义过的变量，只要在变量名前面加美元符号即可，如：

```
your_name="zhangsan"

echo $your_name

echo ${your_name}
```

变量名外面的花括号是可选的，加不加都行，加花括号是为了帮助解释器识别变量的边界，比如下面这种情况：

```
for skill in java php python; do
    echo "I am good at ${skill}Script"
done
```

如果不给skill变量加花括号，写成echo "I am good at \$skillScript"，解释器就会把\$skillScript当成一个变量（其值为空），代码执行结果就不是我们期望的样子了。

推荐给所有变量加上花括号，这是个好的编程习惯。

已定义的变量，可以被重新定义，如：

```
your_name="tom"
echo $your_name
your_name="alibaba"
echo $your_name
```

这样写是合法的，但注意，第二次赋值的时候不能写\$your_name="alibaba"，使用变量的时候才加美元符（\$）。

3 删除变量

使用 unset 命令可以删除变量。语法：

```
unset variable_name
```

变量被删除后不能再次使用。unset 命令不能删除只读变量。

实例

```
#!/bin/sh
myUrl="http://www.runoob.com"
unset myUrl
echo $myUrl
```

以上实例执行将没有任何输出。

4 只读变量

使用 `readonly` 命令可以将变量定义为只读变量，只读变量的值不能被改变。

下面的例子尝试更改只读变量，结果报错：

```
#!/bin/bash

myUrl="http://www.google.com"
readonly myUrl
myUrl="http://www.runoob.com"
```

运行脚本，结果如下：

```
/bin/sh: NAME: This variable is read only.
```

2.4 字符串

字符串是shell编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。

1 单引号

```
skill='java'

str='I am goot at $skill'

echo $str
```

输出结果为：

```
I am goot at $skill
```

单引号字符串的限制：

- 单引号里的任何字符都会原样输出，单引号字符串中的**变量是无效的**；
- 单引号字符串中不能出现单独一个的单引号（对单引号使用转义符后也不行），但可成对出现，作为字符串拼接使用。

2 双引号

```
skill='java'

str="I am goot at $skill"

echo $str
```

输出结果为：

```
I am goot at java
```

双引号的优点：

- 双引号里可以有变量
- 双引号里可以出现转义字符

3 获取字符串长度

```
skill='java'

echo ${skill}      # 输出结果： java

echo ${#skill}     # 输出结果： 4
```

4 提取子字符串

以下实例从字符串第 2 个字符开始截取 4 个字符：

```
str="I am goot at $skill"

echo ${str:2}      # 输出结果为： am goot at java

echo ${str:2:2}    # 输出结果为： am
```

5 查找子字符串

查找字符 **i** 或 **o** 的位置(哪个字母先出现就计算哪个)：

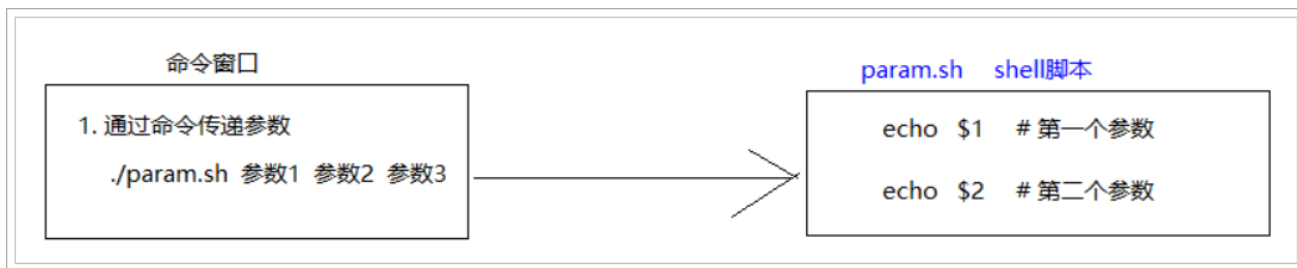
```
str="I am goot at $skill"

echo `expr index "$str" am`  # 输出是： 3
```

注意： 以上脚本中 ` 是反引号(Esc下面的)，而不是单引号 '，不要看错了哦。



2.5 传递参数



我们可以在执行 Shell 脚本时，向脚本传递参数，脚本内获取参数的格式为：**\$n**。

n 代表一个数字，1 为执行脚本的第一个参数，2 为执行脚本的第二个参数，以此类推.....

实例

以下实例我们向脚本传递三个参数，并分别输出，其中 **\$0** 为执行的文件名：

```
vim /export/sh/param.sh
```

```
#!/bin/bash

echo "Shell 传递参数实例! ";

echo "执行的文件名: $0";

echo "第一个参数为: $1";

echo "第二个参数为: $2";

echo "第三个参数为: $3";
```

为脚本设置可执行权限，并执行脚本，输出结果如下所示：

```
$ chmod 755 param.sh

$ ./param.sh 1 2 3
```


Shell 传递参数实例！

```
执行的文件名: ./param.sh

第一个参数为: 1

第二个参数为: 2

第三个参数为: 3
```

另外，还有几个特殊字符用来处理参数：

参数处理	说明
\$#	传递到脚本的参数个数
\$*	以一个单字符串显示所有向脚本传递的参数。 如"\$*"用「」括起来的情况、以"\$1 \$2 ... \$n"的形式输出所有参数。
\$\$	脚本运行的当前进程ID号
\$_	后台运行的最后一个进程的ID号
\$@	与\$*相同，但是使用时加引号，并在引号中返回每个参数。 如"\$@"用「」括起来的情况、以"\$1" "\$2" ... "\$n" 的形式输出所有参数。
\$-	显示Shell使用的当前选项，与 set命令 功能相同。
\$?	显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误。

```
#!/bin/bash

echo "Shell 传递参数实例！";

echo "第一个参数为: $1";

echo "参数个数为: $#";

echo "传递的参数作为一个字符串显示: $*";
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
```

```
$ ./test.sh 1 2 3
```

Shell 传递参数实例!

第一个参数为: 1

参数个数为: 3

传递的参数作为一个字符串显示: 1 2 3

`$*` 与 `$@` 区别:

- 相同点: 都是引用所有参数。
- 不同点: 只有在双引号中体现出来。假设在脚本运行时写了三个参数 1、2、3, , 则 "`*`" 等价于 "1 2 3" (传递了一个参数), 而 "`@`" 等价于 "1" "2" "3" (传递了三个参数)。

```
#!/bin/bash

echo "-- $* 演示 ---"
for i in "$*"; do
    echo $i
done

echo "-- @$ 演示 ---"
for i in "$@"; do
    echo $i
done
```

执行脚本, 输出结果如下所示:

```
$ chmod +x test.sh
```

```
$ ./test.sh 1 2 3
```

```
-- $* 演示 ---
```

```
1 2 3
```

```
-- @$ 演示 ---
```

```
1
```

```
2
```

```
3
```

2.6 Shell算术运算符

1 简介

Shell 和其他编程一样，**支持**包括：算术、关系、布尔、字符串等运算符。

原生 bash **不支持** 简单的数学运算，但是可以通过其他命令来实现，例如expr。

expr 是一款表达式计算工具，使用它能完成表达式的求值操作。

例如，两个数相加：

```
val=`expr 2 + 2`
echo $val
```

注意：

表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2。

完整的表达式要被 ` 包含，注意不是单引号，在 Esc 键下边。

下表列出了常用的算术运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明	举例
+	加法	<code>expr \$a + \$b</code> 结果为 30。
-	减法	<code>expr \$a - \$b</code> 结果为 -10。
*	乘法	<code>expr \$a * \$b</code> 结果为 200。
/	除法	<code>expr \$b / \$a</code> 结果为 2。
%	取余	<code>expr \$b % \$a</code> 结果为 0。
=	赋值	<code>a=\$b</code> 将把变量 b 的值赋给 a。
==	相等。用于比较两个数字，相同则返回 true。	<code>[\$a == \$b]</code> 返回 false。
!=	不相等。用于比较两个数字，不相同则返回 true。	<code>[\$a != \$b]</code> 返回 true。

注意：条件表达式要放在方括号之间，并且要有空格，例如: `[$a==$b]` 是错误的，必须写成 `[$a == $b]`。

2 例子

```
#!/bin/bash

a=4

b=20
```

#加法运算

```
each expr $a + $b
```

#减法运算

```
echo expr $a - $b
```

#乘法运算，注意*号前面需要反斜杠

```
echo expr $a \* $b
```

#除法运算

```
echo $a / $b
```

此外，还可以通过`(())`、`$(())`、`$[]`进行算术运算。

```
((a++))
```

```
echo "a = $a"
```

```
c=$((a + b))
```

```
d=$((a + b))
```

```
echo "c = $c"
```

```
echo "d = $d"
```

2.7 流程控制

1 if else

1.1 if

if 语句语法格式：

```
if condition; then
    command1
    command2
    ...
    commandN
fi
```

demo

```
[root@hadoop01 export]# cat if_test.sh
#!/bin/bash

a=20

if [ $a -gt 10 ]; then
    echo "a 大于 10"
fi
```

末尾的fi就是if倒过来拼写，后面还会遇到类似的。

1.2 if else

if else 语法格式：

```
if condition; then
    command1
    command2
    ...
    commandN
else
    command
fi
```

1.3 if else-if else

if else-if else 语法格式：

```
if condition1; then
    command1
elif condition2; then
    command2
else
    commandN
fi
```

以下实例判断两个变量是否相等：

关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

下表列出了常用的关系运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明	英文	举例
-eq	检测两个数是否相等，相等返回 true。	equal	[\$a -eq \$b] 返回 false。
-ne	检测两个数是否不相等，不相等返回 true。	not equal	[\$a -ne \$b] 返回 true。
-gt	检测左边的数是否大于右边的，如果是，则返回 true。	greater than	[\$a -gt \$b] 返回 false。
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	less than	[\$a -lt \$b] 返回 true。
-ge	检测左边的数是否大于等于右边的，如果是，则返回 true。	Greater than or equal to	[\$a -ge \$b] 返回 false。
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。	Less than or equal to	[\$a -le \$b] 返回 true。

案例:

```
[root@hadoop01 export]# cat if_test.sh
#!/bin/bash

a=20
b=10

# 需求1: 判断 a 是否 100
if [ $a > 100 ]; then
    echo "$a 大于 100"
fi

# 需求2: 判断 a 是否等于 b
if [ $a -eq $b ]; then
    echo "$a 等于 $b"
else
    echo "$a 不等于 $b"
fi

# 需求3: 判断 a 与 b 比较
if [ $a -lt $b ]; then
    echo "$a 小于 $b"
elif [ $a -eq $b ]; then
    echo "$a 等于 $b"
else
    echo "$a 大于 $b"
fi
```

```
# 需求4: 判断 (a + 10) 和 (b * b) 比较大小
if test $[ a + 10 ] -gt $[ b * b ]; then
    echo "(a+10) 大于 (b * b)"
else
    echo "(a+10) 小于或等于 (b*b)"
fi
```

2 for 循环

2.1 格式

```
for variable in (list); do
    command
    command
    ...
done
```

2.2 随堂练习

```
# 需求1: 遍历 1~5
# 需求2: 遍历 1~100
# 需求3: 遍历 1~100之间的奇数
# 需求4: 遍历 根目录 下的内容
```

代码如下:

```
[root@hadoop01 export]# cat for_test.sh
#!/bin/bash

# 需求1: 遍历 1~5
for i in 1 2 3 4 5; do
    echo $i;
done
# 需求2: 遍历 1~100
for i in {1..100}; do
    echo $i
done
# 需求3: 遍历 1~100之间的奇数
for i in {1..100..2}; do
    echo $i
done
# 需求4: 遍历 根目录 下的内容
for f in `ls /`; do
    echo $f
done
```

3 while 语句

while循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。其格式为：

```
while condition; do
    command
done
```

需求: 计算 1~100 的和

```
#!/bin/bash

sum=0
i=1
while [ $i -le 100 ]; do
    sum=$(( sum + i ))
    i=$(( i + 1 ))
done

echo $sum
```

运行脚本，输出：

```
1
2
3
4
5
```

使用中使用了 Bash let 命令，它用于执行一个或多个表达式，变量计算中不需要加上 \$ 来表示变量，具体可查阅：[Bash let 命令](#)。

4 无限循环

无限循环语法格式：


```
while true; do
    command
done
```

需求：每隔1秒 打印一次当前时间

5 case

Shell case语句为多选择语句。可以用case语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。case语句格式如下：

```
case 值 in
    模式1)
        command1
        command2
        ...
        commandN
        ;;
    模式2)
        command1
        command2
        ...
        commandN
        ;;
esac
```

case工作方式如上所示。取值后面必须为单词in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至;;。

取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号*捕获该值，再执行后面的命令。

下面的脚本提示输入1到4，与每一种模式进行匹配：

```
echo '输入 1 到 4 之间的数字:'

read aNum

case $aNum in
    1) echo '你选择了 1'
        ;;
    2) echo '你选择了 2'
```

```
;;

3) echo '你选择了 3'
;;

4) echo '你选择了 4'
;;

*) echo '你没有输入 1 到 4 之间的数字'
;;
esac
```

输入不同的内容，会有不同的结果，例如：

输入 1 到 4 之间的数字：

你输入的数字为：

3

你选择了 3

6 跳出循环

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，Shell使用两个命令来实现该功能：break和continue。

6.1 break命令

break命令允许跳出所有循环（终止执行后面的所有循环）。

需求：执行死循环 每隔1秒打印当前时间，执行10次停止

```
#!/bin/bash
# 需求: 执行死循环 每隔1秒打印当前时间, 执行10次停止
i=0;
while true; do
    sleep 1
    echo $i `date +%Y-%m-%d %H:%M:%S`

    i=$(( i + 1 ))
    if [ $i -eq 10 ]; then
        break
    fi
done
```

6.2 continue

continue命令与break命令类似, 只有一点差别, 它不会跳出所有循环, 仅仅跳出当前循环。

需求: 打印 1~30, 注意 跳过3的倍数

```
#!/bin/bash

# 需求: 打印 1~30, 注意 跳过3的倍数

for i in {1..30}; do
    if test $( i % 3 ) -eq 0; then
        continue
    fi
    echo $i
done
```

2.8 函数使用

1 函数的快速入门

- 格式

```
[ function ] funname()
{
    action;
    [return int;]
}
```

- 可以带function fun() 定义，也可以直接fun() 定义,不带任何参数。
 - 参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。return后跟数值n(0-255)
- 快速入门

```
#!/bin/bash

demoFun(){

    echo "这是我的第一个 shell 函数!"

}

echo "-----函数开始执行-----"
demoFun
echo "-----函数执行完毕-----"
```

2 传递参数给函数

在Shell中，调用函数时可以向其传递参数。在函数体内部，通过 \$n 的形式来获取参数的值，例如，\$1表示第一个参数，\$2表示第二个参数...

带参数的函数示例：

```
#!/bin/bash

funWithParam(){
    echo "第一个参数为 $1 !"
    echo "第二个参数为 $2 !"
    echo "第十个参数为 $10 !"
    echo "第十个参数为 ${10} !"
    echo "第十一个参数为 ${11} !"
    echo "参数总数有 $# 个!"
    echo "作为一个字符串输出所有参数 $* !"
}

funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

输出结果：

```
第一个参数为 1 !

第二个参数为 2 !

第十个参数为 10 !
```

第十个参数为 34 ！

第十一个参数为 73 ！

参数总数有 11 个！

作为一个字符串输出所有参数 1 2 3 4 5 6 7 8 9 34 73 ！

注意，\$10 不能获取第十个参数，获取第十个参数需要\${10}。当n>=10时，需要使用\${n}来获取参数。

另外，还有几个特殊字符用来处理参数：

参数处理	说明
\$#	传递到脚本的参数个数
\$*	以一个单字符串显示所有向脚本传递的参数
\$\$	脚本运行的当前进程ID号
\$_	后台运行的最后一个进程的ID号
@	与\$*相同，但是使用时加引号，并在引号中返回每个参数。
-	显示Shell使用的当前选项，与set命令功能相同。
?	显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误。

2.9 数组

1 定义数组

数组中可以存放多个值。Bash Shell **只支持一维数组**（不支持多维数组），初始化时不需要定义数组大小（。与大部分编程语言类似，数组元素的下标由0开始。

Shell 数组用括号来表示，元素用"空格"符号分割开，语法格式如下：

```
array_name=(value1 value2 value3 ... valuen)
```

1.1 实例

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

```
my_array=(A B "C" D)
```

我们也可以使用下标来定义数组：

```
array_name[0]=value0
```

```
array_name[1]=value1
```

```
array_name[2]=value2
```

2 读取数组

读取数组元素值的一般格式是：

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

2.1 实例

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

```
my_array=(A B "C" D)
```

```
echo "第一个元素为：${my_array[0]}"
```

```
echo "第二个元素为：${my_array[1]}"
```

```
echo "第三个元素为：${my_array[2]}"
```

```
echo "第四个元素为：${my_array[3]}"
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
```

```
$ ./test.sh
```

第一个元素为：A

第二个元素为：B

第三个元素为：C

第四个元素为：D

2.2 获取数组中的所有元素

使用@ 或 * 可以获取数组中的所有元素，例如：

```
#!/bin/bash

my_array[0]=A
my_array[1]=B
my_array[2]=C
my_array[3]=D

echo "数组的元素为: ${my_array[*]}"
echo "数组的元素为: ${my_array[@]}"
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh

数组的元素为: A B C D
数组的元素为: A B C D
```

2.3 获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同，例如：

```
#!/bin/bash
my_array[0]=A
my_array[1]=B
my_array[2]=C
my_array[3]=D

echo "数组元素个数为: ${#my_array[*]}"
echo "数组元素个数为: ${#my_array[@]}"
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh

数组元素个数为: 4
数组元素个数为: 4
```

3 遍历数组

3.1 方式一

```
#!/bin/bash

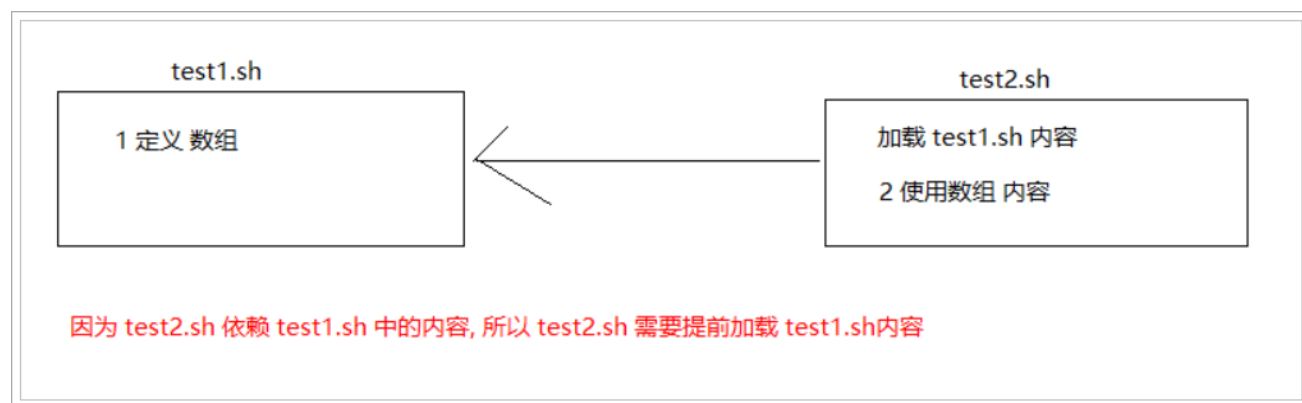
my_arr=(AA BB CC)

for var in ${my_arr[*]}
do
    echo $var
done
```

3.2 方式二

```
my_arr=(AA BB CC)
my_arr_num=${#my_arr[*]}
for((i=0;i<my_arr_num;i++));
do
    echo "-----"
    echo ${my_arr[$i]}
done
```

2.10 加载其它文件的变量



1 简介

和其他语言一样, Shell 也可以包含外部脚本。这样可以很方便的封装一些公用的代码作为一个独立的文件。

Shell 文件包含的语法格式如下:


```
. filename    # 注意点号(.)和文件名中间有一空格
```

或

```
source filename
```

2 练习

定义两个文件 test1.sh和test2.sh, 在test1中定义一个变量arr=(java c++ shell),在test2中对arr进行循环打印输出。

第一步: vim test1.sh

```
#!/bin/bash

my_arr=(AA BB CC)
```

第二步: vim test2.sh

```
#!/bin/bash

source ./test1.sh  # 加载test1.sh 的文件内容

for var in ${my_arr[*]}
do

    echo $var

done
```

第三步: 执行 test2.sh

```
sh test2.sh
```

好处：

1. 数据源 和 业务处理 分离
2. 复用 代码扩展性更强