# **Shell函数详解（函数定义、函数调**用）

Shell 函数定义的语法格式如下：

function name() {  
    statements  
    [return value]  
}

对各个部分的说明：

* function是 Shell 中的关键字，专门用来定义函数；
* name是函数名；
* statements是函数要执行的代码，也就是一组语句；
* return value表示函数的返回值，其中 return 是 Shell 关键字，专门用在函数中返回一个值；这一部分可以写也可以不写。

由{ }包围的部分称为函数体，调用一个函数，实际上就是执行函数体中的代码。

## 函数定义的简化写法

如果你嫌麻烦，函数定义时也可以不写 function 关键字：

name() {  
    statements  
    [return value]  
}

如果写了 function 关键字，也可以省略函数名后面的小括号：

function name {  
    statements  
    [return value]  
}

我建议使用标准的写法，这样能够做到“见名知意”，一看就懂。

## 函数调用

调用 Shell 函数时可以给它传递参数，也可以不传递。如果不传递参数，直接给出函数名字即可：

name

如果传递参数，那么多个参数之间以空格分隔：

name param1 param2 param3

不管是哪种形式，函数名字后面都不需要带括号。  
  
和其它编程语言不同的是，Shell 函数在定义时不能指明参数，但是在调用时却可以传递参数，并且给它传递什么参数它就接收什么参数。  
  
Shell 也不限制定义和调用的顺序，你可以将定义放在调用的前面，也可以反过来，将定义放在调用的后面。

## 实例演示

1) 定义一个函数，输出 Shell 教程的地址：

1. #!/bin/bash
2. #函数定义
3. function url(){
4. echo "http://c.biancheng.net/shell/"
5. }
6. #函数调用
7. url

运行结果：  
http://c.biancheng.net/shell/  
  
你可以将调用放在定义的前面，也就是写成下面的形式：

1. #!/bin/bash
2. #函数调用
3. url
4. #函数定义
5. function url(){
6. echo "http://c.biancheng.net/shell/"
7. }

2) 定义一个函数，计算所有参数的和：

1. #!/bin/bash
2. function getsum(){
3. **local** sum=0
4. **for** n **in** $@
5. **do**
6. ((sum+=n))
7. **done**
8. **return** $sum
9. }
10. getsum 10 20 55 15 #调用函数并传递参数
11. echo $?

运行结果：  
100  
  
$@表示函数的所有参数，$?表示函数的退出状态（返回值）。关于如何获取函数的参数，我们将在《[Shell函数参数](http://c.biancheng.net/view/2860.html" \t "http://c.biancheng.net/view/_blank)》一节中详细讲解。  
  
此处我们借助 return 关键字将所有数字的和返回，并使用$?得到这个值，这种处理方案在其它编程语言中没有任何问题，但是在 Shell 中是非常错误的，Shell 函数的返回值和其它编程语言大有不同，我们将在《[Shell函数返回值](http://c.biancheng.net/view/vip_3239.html" \t "http://c.biancheng.net/view/_blank)》中展开讨论。

# **Shell函数参数**

函数参数是 [Shell 位置参数](http://c.biancheng.net/view/789.html" \t "http://c.biancheng.net/view/_blank)的一种，在函数内部可以使用$n来接收，例如，$1 表示第一个参数，$2 表示第二个参数，依次类推。  
  
除了$n，还有另外三个比较重要的变量：

* $#可以获取传递的参数的个数；
* $@或者$\*可以一次性获取所有的参数（猛击《[Shell $\*和$@的区别](http://c.biancheng.net/view/807.html" \t "http://c.biancheng.net/view/_blank)》可以了解更多内容）。

$n、$#、$@、$\* 都属于特殊变量，不了解的读者请转到《[Shell特殊变量](http://c.biancheng.net/view/806.html" \t "http://c.biancheng.net/view/_blank)》。  
  
【实例1】使用 $n 来接收函数参数。

1. #!/bin/bash
2. #定义函数
3. function show(){
4. echo "Tutorial: $1"
5. echo "URL: $2"
6. echo "Author: "$3
7. echo "Total $# parameters"
8. }
9. #调用函数
10. show C# http://c.biancheng.net/csharp/ Tom

运行结果：  
Tutorial: C#  
URL: http://c.biancheng.net/csharp/  
Author: Tom  
Total 3 parameters  
  
注意，第 7 行代码的写法有点不同，这里使用了 [Shell 字符串拼接](http://c.biancheng.net/view/1114.html" \t "http://c.biancheng.net/view/_blank)技巧。  
  
【实例2】使用 $@ 来遍历函数参数。  
  
定义一个函数，计算所有参数的和：

1. #!/bin/bash
2. function getsum(){
3. **local** sum=0
4. **for** n **in** $@
5. **do**
6. ((sum+=n))
7. **done**
8. echo $sum
9. **return** 0
10. }
11. #调用函数并传递参数，最后将结果赋值给一个变量
12. total=$(getsum 10 20 55 15)
13. echo $total
14. #也可以将变量省略
15. echo $(getsum 10 20 55 15)

运行结果：  
100  
100

[Shell $\*和$@的区别](http://c.biancheng.net/view/807.html" \t "http://c.biancheng.net/view/_blank)

当 $\* 和 $@ 不被双引号" "包围时，它们之间没有任何区别，都是将接收到的每个参数看做一份数据，彼此之间以空格来分隔。  
  
但是当它们被双引号" "包含时，就会有区别了：

* "$\*"会将所有的参数从整体上看做一份数据，而不是把每个参数都看做一份数据。
* "$@"仍然将每个参数都看作一份数据，彼此之间是独立的。

比如传递了 5 个参数，那么对于"$\*"来说，这 5 个参数会合并到一起形成一份数据，它们之间是无法分割的；而对于"$@"来说，这 5 个参数是相互独立的，它们是 5 份数据。  
  
如果使用 echo 直接输出"$\*"和"$@"做对比，是看不出区别的；但如果使用 for 循环来逐个输出数据，立即就能看出区别来。

关于 for 循环的用法请猛击：[Shell for循环和for int循环详解](http://c.biancheng.net/view/2804.html" \t "http://c.biancheng.net/view/_blank)

编写下面的代码，并保存为 test.sh：

[纯文本复制](http://c.biancheng.net/view/807.html)

1. #!/bin/bash
2. echo "print each param from \"\$\*\""
3. **for** var **in** "$\*"
4. **do**
5. echo "$var"
6. **done**
7. echo "print each param from \"\$@\""
8. **for** var **in** "$@"
9. **do**
10. echo "$var"
11. **done**

运行 test.sh，并附带参数：

[mozhiyan@localhost demo]$ . ./test.sh a b c d  
print each param from "$\*"  
a b c d  
print each param from "$@"  
a  
b  
c  
d

从运行结果可以发现，对于"$\*"，只循环了 1 次，因为它只有 1 分数据；对于"$@"，循环了 5 次，因为它有 5 份数据。

# **Shell特殊变量：Shell $#、$\*、$@、$?、$$**

|  |  |
| --- | --- |
| [Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) 特殊变量及其含义 | |
| **变量** | **含义** |
| $0 | 当前脚本的文件名。 |
| $n（n≥1） | 传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是 $1，第二个参数是 $2。 |
| $# | 传递给脚本或函数的参数个数。 |
| $\* | 传递给脚本或函数的所有参数。 |
| $@ | 传递给脚本或函数的所有参数。当被双引号" "包含时，$@ 与 $\* 稍有不同，我们将在《[Shell $\*和$@的区别](http://c.biancheng.net/view/807.html)》一节中详细讲解。 |
| $? | 上个命令的退出状态，或函数的返回值，我们将在《[Shell $?](http://c.biancheng.net/view/808.html)》一节中详细讲解。 |
| $$ | 当前 Shell 进程 ID。对于 [Shell 脚本](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank)，就是这些脚本所在的进程 ID。 |

下面我们通过两个例子来演示。

## 1) 给脚本文件传递参数

编写下面的代码，并保存为 test.sh：

1. #!/bin/bash
2. echo "Process ID: $$"
3. echo "File Name: $0"
4. echo "First Parameter : $1"
5. echo "Second Parameter : $2"
6. echo "All parameters 1: $@"
7. echo "All parameters 2: $\*"
8. echo "Total: $#"

运行 test.sh，并附带参数：

[mozhiyan@localhost demo]$ . ./test.sh Shell [Linux](http://c.biancheng.net/linux_tutorial/" \t "http://c.biancheng.net/view/_blank)

Process ID: 5943

File Name: bash

First Parameter : Shell

Second Parameter : Linux

All parameters 1: Shell Linux

All parameters 2: Shell Linux

Total: 2

## 2) 给函数传递参数

编写下面的代码，并保存为 test.sh：

[纯文本复制](http://c.biancheng.net/view/806.html)

1. #!/bin/bash
2. #定义函数
3. function func(){
4. echo "Language: $1"
5. echo "URL: $2"
6. echo "First Parameter : $1"
7. echo "Second Parameter : $2"
8. echo "All parameters 1: $@"
9. echo "All parameters 2: $\*"
10. echo "Total: $#"
11. }
12. #调用函数
13. func [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) http://c.biancheng.net/java/

运行结果为：  
Language: Java  
URL: http://c.biancheng.net/java/  
First Parameter : Java  
Second Parameter : http://c.biancheng.net/java/  
All parameters 1: Java http://c.biancheng.net/java/  
All parameters 2: Java http://c.biancheng.net/java/  
Total: 2

# **Shell函数返回值精讲**

# **Linux Shell重定向（输入输出重定向）精讲**

[Linux Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) 重定向分为两种，一种输入重定向，一种是输出重定向；从字面上理解，输入输出重定向就是「改变输入与输出的方向」的意思。

Linux 中一切皆文件，包括标准输入设备（键盘）和标准输出设备（显示器）在内的所有计算机硬件都是文件。  
  
为了表示和区分已经打开的文件，Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。

|  |  |  |  |
| --- | --- | --- | --- |
| 表1：与输入输出有关的文件描述符 | | | |
| **文件描述符** | **文件名** | **类型** | **硬件** |
| 0 | stdin | 标准输入文件 | 键盘 |
| 1 | stdout | 标准输出文件 | 显示器 |
| 2 | stderr | 标准错误输出文件 | 显示器 |

Linux 程序在执行任何形式的 I/O 操作时，都是在读取或者写入一个文件描述符。一个文件描述符只是一个和打开的文件相关联的整数，它的背后可能是一个硬盘上的普通文件、FIFO、管道、终端、键盘、显示器，甚至是一个网络连接。  
  
stdin、stdout、stderr 默认都是打开的，在重定向的过程中，0、1、2 这三个文件描述符可以直接使用。

## Linux [Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) 输出重定向

输出重定向是指命令的结果不再输出到显示器上，而是输出到其它地方，一般是文件中。这样做的最大好处就是把命令的结果保存起来，当我们需要的时候可以随时查询。Bash 支持的输出重定向符号如下表所示。

|  |  |  |
| --- | --- | --- |
| 表2：Bash 支持的输出重定向符号 | | |
| **类 型** | **符 号** | **作 用** |
| 标准输出重定向 | command >file | 以覆盖的方式，把 command 的正确输出结果输出到 file 文件中。 |
| command >>file | 以追加的方式，把 command 的正确输出结果输出到 file 文件中。 |
| 标准错误输出重定向 | command 2>file | 以覆盖的方式，把 command 的错误信息输出到 file 文件中。 |
| command 2>>file | 以追加的方式，把 command 的错误信息输出到 file 文件中。 |
| 正确输出和错误信息同时保存 | command >file 2>&1 | 以覆盖的方式，把正确输出和错误信息同时保存到同一个文件（file）中。 |
| command >>file 2>&1 | 以追加的方式，把正确输出和错误信息同时保存到同一个文件（file）中。 |
| command >file1 2>file2 | 以覆盖的方式，把正确的输出结果输出到 file1 文件中，把错误信息输出到 file2 文件中。 |
| command >>file1  2>>file2 | 以追加的方式，把正确的输出结果输出到 file1 文件中，把错误信息输出到 file2 文件中。 |
| command >file 2>file | 【**不推荐**】这两种写法会导致 file 被打开两次，引起资源竞争，所以 stdout 和 stderr 会互相覆盖，我们将在《[结合Linux文件描述符谈重定向，彻底理解重定向的本质](http://c.biancheng.net/view/vip_3241.html" \t "http://c.biancheng.net/view/_blank)》一节中深入剖析。 |
| command >>file 2>>file |

在输出重定向中，>代表的是覆盖，>>代表的是追加。

#### 注意

输出重定向的完整写法其实是fd>file或者fd>>file，其中 fd 表示文件描述符，如果不写，默认为 1，也就是标准输出文件。  
  
当文件描述符为 1 时，一般都省略不写，如上表所示；当然，如果你愿意，也可以将command >file写作command 1>file，但这样做是多此一举。  
  
当文件描述符为大于 1 的值时，比如 2，就必须写上。  
  
需要重点说明的是，fd和>之间不能有空格，否则 Shell 会解析失败；>和file之间的空格可有可无。为了保持一致，我习惯在>两边都不加空格。  
  
下面的语句是一个反面教材：

echo "c.biancheng.net" 1 >log.txt

注意1和>之间的空格。echo 命令的输出结果是c.biancheng.net，我们的初衷是将输出结果重定向到 log.txt，但是当你打开 log.txt 文件后，发现文件的内容为c.biancheng.net 1，这就是多余的空格导致的解析错误。也就是说，Shell 将该条语句理解成了下面的形式：

echo "c.biancheng.net" 1 1>log.txt

#### 输出重定向举例

【实例1】将 echo 命令的输出结果以追加的方式写入到 demo.txt 文件中。

1. #!/bin/bash
2. **for** str **in** "C语言中文网" "http://c.biancheng.net/" "成立7年了" "日IP数万"
3. **do**
4. echo $str >>demo.txt #将输入结果以追加的方式重定向到文件
5. **done**

运行以上脚本，使用cat demo.txt查看文件内容，显示如下：  
C语言中文网  
http://c.biancheng.net/  
成立7年了  
日IP数万  
  
【实例2】将ls -l命令的输出结果重定向到文件中。

[c.biancheng.net]$ ls -l #先预览一下输出结果

总用量 16

drwxr-xr-x. 2 root root 21 7月 1 2016 abc

-rw-r--r--. 1 mozhiyan mozhiyan 399 3月 11 17:12 demo.sh

-rw-rw-r--. 1 mozhiyan mozhiyan 67 3月 22 17:16 demo.txt

-rw-rw-r--. 1 mozhiyan mozhiyan 278 3月 16 17:17 main.c

-rwxr-xr-x. 1 mozhiyan mozhiyan 187 3月 22 17:16 test.sh

[c.biancheng.net]$ ls -l >demo.txt #重定向

[c.biancheng.net]$ cat demo.txt #查看文件内容

总用量 12

drwxr-xr-x. 2 root root 21 7月 1 2016 abc

-rw-r--r--. 1 mozhiyan mozhiyan 399 3月 11 17:12 demo.sh

-rw-rw-r--. 1 mozhiyan mozhiyan 0 3月 22 17:21 demo.txt

-rw-rw-r--. 1 mozhiyan mozhiyan 278 3月 16 17:17 main.c

-rwxr-xr-x. 1 mozhiyan mozhiyan 187 3月 22 17:16 test.sh

#### 错误输出重定向举例

命令正确执行是没有错误信息的，我们必须刻意地让命令执行出错，如下所示：

[c.biancheng.net]$ ls java #先预览一下错误信息

ls: 无法访问java: 没有那个文件或目录

[c.biancheng.net]$ ls java 2>err.log #重定向

[c.biancheng.net]$ cat err.log #查看文件

ls: 无法访问java: 没有那个文件或目录

#### 正确输出和错误信息同时保存

【实例1】把正确结果和错误信息都保存到一个文件中，例如：

[c.biancheng.net]$ ls -l >out.log 2>&1

[c.biancheng.net]$ ls java >>out.log 2>&1

[c.biancheng.net]$ cat out.log

总用量 12

drwxr-xr-x. 2 root root 21 7月 1 2016 abc

-rw-r--r--. 1 mozhiyan mozhiyan 399 3月 11 17:12 demo.sh

-rw-rw-r--. 1 mozhiyan mozhiyan 278 3月 16 17:17 main.c

-rw-rw-r--. 1 mozhiyan mozhiyan 0 3月 22 17:39 out.log

-rwxr-xr-x. 1 mozhiyan mozhiyan 187 3月 22 17:16 test.sh

ls: 无法访问java: 没有那个文件或目录

out.log 的最后一行是错误信息，其它行都是正确的输出结果。  
  
【实例2】上面的实例将正确结果和错误信息都写入同一个文件中，这样会导致视觉上的混乱，不利于以后的检索，所以我建议把正确结果和错误信息分开保存到不同的文件中，也即写成下面的形式：

ls -l >>out.log 2>>err.log

这样一来，正确的输出结果会写入到 out.log，而错误的信息则会写入到 err.log。

#### /dev/null 文件

如果你既不想把命令的输出结果保存到文件，也不想把命令的输出结果显示到屏幕上，干扰命令的执行，那么可以把命令的所有结果重定向到 /dev/null 文件中。如下所示：

ls -l &>/dev/null

大家可以把 /dev/null 当成 Linux 系统的垃圾箱，任何放入垃圾箱的数据都会被丢弃，不能恢复。

## Linux Shell 输入重定向

输入重定向就是改变输入的方向，不再使用键盘作为命令输入的来源，而是使用文件作为命令的输入。

|  |  |
| --- | --- |
| 表3：Bash 支持的输出重定向符号 | |
| **符号** | **说明** |
| command <file | 将 file 文件中的内容作为 command 的输入。 |
| command <<END | 从标准输入（键盘）中读取数据，直到遇见分界符 END 才停止（分界符可以是任意的字符串，用户自己定义）。 |
| command <file1 >file2 | 将 file1 作为 command 的输入，并将 command 的处理结果输出到 file2。 |

和输出重定向类似，输入重定向的完整写法是fd<file，其中 fd 表示文件描述符，如果不写，默认为 0，也就是标准输入文件。

#### 输入重定向举例

【示例1】统计文档中有多少行文字。  
  
Linux wc 命令可以用来对文本进行统计，包括单词个数、行数、字节数，它的用法如下：

wc  [选项]  [文件名]

其中，-c选项统计字节数，-w选项统计单词数，-l选项统计行数。  
  
统计 readme.txt 文件中有多少行文本：

[c.biancheng.net]$ cat readme.txt #预览一下文件内容

C语言中文网

http://c.biancheng.net/

成立7年了

日IP数万

[c.biancheng.net]$ wc -l <readme.txt #输入重定向

4

【实例2】逐行读取文件内容。

#!/bin/bash

while read str; do

echo $str

done <readme.txt

运行结果：  
C语言中文网  
http://c.biancheng.net/  
成立7年了  
日IP数万  
  
这种写法叫做代码块重定向，也就是把一组命令同时重定向到一个文件，我们将在《[Shell代码块重定向](http://c.biancheng.net/view/vip_3243.html" \t "http://c.biancheng.net/view/_blank)》一节中详细讲解  
  
【实例3】统计用户在终端输入的文本的行数。  
  
此处我们使用输入重定向符号<<，这个符号的作用是使用特定的分界符作为命令输入的结束标志，而不使用 Ctrl+D 键。

[c.biancheng.net]$ wc -l <<END

> 123

> 789

> abc

> xyz

> END

4

wc 命令会一直等待用输入，直到遇见分界符 END 才结束读取。  
  
<<之后的分界符可以自由定义，只要再碰到相同的分界符，两个分界符之间的内容将作为命令的输入（不包括分界符本身）。

# **Linux Shell管道详解**

[Linux](http://c.biancheng.net/linux_tutorial/" \t "http://c.biancheng.net/view/_blank) 管道使用竖线|连接多个命令，这被称为管道符。Linux 管道的具体语法格式如下：

command1 | command2  
command1 | command2 [ | commandN... ]

当在两个命令之间设置管道时，管道符|左边命令的输出就变成了右边命令的输入。只要第一个命令向标准输出写入，而第二个命令是从标准输入读取，那么这两个命令就可以形成一个管道。大部分的 [Linux 命令](http://c.biancheng.net/linux/" \t "http://c.biancheng.net/view/_blank)都可以用来形成管道。

这里需要注意，command1 必须有正确输出，而 command2 必须可以处理 command2 的输出结果；而且 command2 只能处理 command1 的正确输出结果，不能处理 command1 的错误信息。

## 为什么使用管道？

我们先看下面一组命令，使用 mysqldump（一个数据库备份程序）来备份一个叫做 wiki 的数据库：

mysqldump -u root -p '123456' wiki > /tmp/wikidb.backup  
gzip -9 /tmp/wikidb.backup  
scp /tmp/wikidb.backup username@remote\_ip:/backup/mysql/

上述这组命令主要做了如下任务：

* mysqldump 命令用于将名为 wike 的数据库备份到文件 /tmp/wikidb.backup；其中-u和-p选项分别指出数据库的用户名和密码。
* gzip 命令用于压缩较大的数据库文件以节省磁盘空间；其中-9表示最慢的压缩速度最好的压缩效果。
* scp 命令（secure copy，安全拷贝）用于将数据库备份文件复制到 IP 地址为 remote\_ip 的备份服务器的 /backup/mysql/ 目录下。其中username是登录远程服务器的用户名，命令执行后需要输入密码。

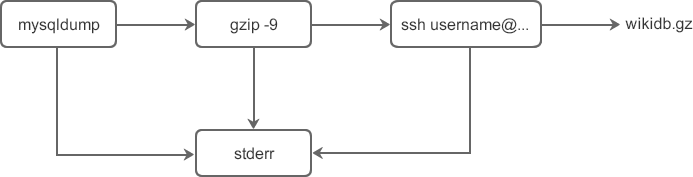
上述三个命令依次执行。然而，如果使用管道的话，你就可以将 mysqldump、gzip、ssh 命令相连接，这样就避免了创建临时文件 /tmp/wikidb.backup，而且可以同时执行这些命令并达到相同的效果。  
  
使用管道后的命令如下所示：

mysqldump -u root -p '123456' wiki | gzip -9 | ssh username@remote\_ip "cat > /backup/wikidb.gz"

这些使用了管道的命令有如下特点：

* 命令的语法紧凑并且使用简单。
* 通过使用管道，将三个命令串联到一起就完成了远程 mysql 备份的复杂任务。
* 从管道输出的标准错误会混合到一起。

上述命令的数据流如下图所示：



## 重定向和管道的区别

乍看起来，管道也有重定向的作用，它也改变了数据输入输出的方向，那么，管道和重定向之间到底有什么不同呢？  
  
简单地说，重定向操作符>将命令与文件连接起来，用文件来接收命令的输出；而管道符 | 将命令与命令连接起来，用第二个命令来接收第一个命令的输出。如下所示：

command > file  
command1 | command1

有些读者在学习管道时会尝试如下的命令，我们来看一下会发生什么：

command1 > command2

答案是，有时尝试的结果将会很糟糕。这是一个实际的例子，一个 Linux 系统管理员以超级用户（root 用户）的身份执行了如下命令：

cd /usr/bin  
ls > less

第一条命令将当前目录切换到了大多数程序所存放的目录，第二条命令是告诉 Shell 用 ls 命令的输出重写文件 less。因为 /usr/bin 目录已经包含了名称为 less（less 程序）的文件，第二条命令用 ls 输出的文本重写了 less 程序，因此破坏了文件系统中的 less 程序。  
  
这是使用重定向操作符错误重写文件的一个教训，所以在使用它时要谨慎。

## Linux管道实例

【实例1】将 ls 命令的输出发送到 grep 命令：

[c.biancheng.net]$ ls | grep log.txtlog.txt

上述命令是查看文件 log.txt 是否存在于当前目录下。  
  
我们可以在命令的后面使用选项，例如使用-al选项：

[c.biancheng.net]$ ls -al | grep log.txt

-rw-rw-r--. 1 mozhiyan mozhiyan 0 4月 15 17:26 log.txt

管道符|与两侧的命令之间也可以不存在空格，例如将上述命令写作ls -al|grep log.txt；然而我还是推荐在管道符|和两侧的命令之间使用空格，以增加代码的可读性。  
  
我们也可以重定向管道的输出到一个文件，比如将上述管道命令的输出结果发送到文件 output.txt 中：

[c.biancheng.net]$ ls -al | grep log.txt >output.txt

[c.biancheng.net]$ cat output.txt

-rw-rw-r--. 1 mozhiyan mozhiyan 0 4月 15 17:26 log.txt

【实例2】使用管道将 cat 命令的输出作为 less 命令的输入，这样就可以将 cat 命令的输出每次按照一个屏幕的长度显示，这对于查看长度大于一个屏幕的文件内容很有帮助。

cat /var/log/message | less

【实例3】查看指定程序的进程运行状态，并将输出重定向到文件中。

[c.biancheng.net]$ ps aux | grep httpd > /tmp/ps.output

[c.biancheng.net]$ cat /tem/ps.output

mozhiyan 4101 13776 0 10:11 pts/3 00:00:00 grep httpd

root 4578 1 0 Dec09 ? 00:00:00 /usr/sbin/httpd

apache 19984 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

apache 19985 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

apache 19986 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

apache 19987 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

apache 19988 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

apache 19989 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

apache 19990 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

apache 19991 4578 0 Dec29 ? 00:00:00 /usr/sbin/httpd

【实例4】显示按用户名排序后的当前登录系统的用户的信息。

[c.biancheng.net]$ who | sort

mozhiyan :0 2019-04-16 12:55 (:0)

mozhiyan pts/0 2019-04-16 13:16 (:0)

who 命令的输出将作为 sort 命令的输入，所以这两个命令通过管道连接后会显示按照用户名排序的已登录用户的信息。  
  
【实例5】统计系统中当前登录的用户数。

[c.biancheng.net]$ who | wc -l

5

## 管道与输入重定向

输入重定向操作符<可以在管道中使用，以用来从文件中获取输入，其语法类似下面这样：

command1 < input.txt | command2  
command1 < input.txt | command2 -option | command3

例如，使用 tr 命令从 os.txt 文件中获取输入，然后通过管道将输出发送给 sort 或 uniq 等命令：

[c.biancheng.net]$ cat os.txt

redhat

suse

centos

ubuntu

solaris

hp-ux

fedora

centos

redhat

hp-ux

[c.biancheng.net]$ tr a-z A-Z <os.txt | sort

CENTOS

CENTOS

FEDORA

HP-UX

HP-UX

REDHAT

REDHAT

SOLARIS

SUSE

UBUNTU

[c.biancheng.net]$ tr a-z A-Z <os.txt | sort | uniq

CENTOS

FEDORA

HP-UX

REDHAT

SOLARIS

SUSE

UBUNTU

## 管道与输出重定向

你也可以使用重定向操作符>或>>将管道中的最后一个命令的标准输出进行重定向，其语法如下所示：

command1 | command2 | ... | commandN > output.txt  
command1 < input.txt | command2 | ... | commandN > output.txt

【实例1】使用 mount 命令显示当前挂载的文件系统的信息，并使用 column 命令格式化列的输出，最后将输出结果保存到一个文件中。

[c.biancheng.net]$ mount | column -t >mounted.txt

[c.biancheng.net]$ cat mounted.txt

proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)

sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)

devtmpfs on /dev type devtmpfs (rw,nosuid,seclabel,size=496136k,nr\_inodes=124034,mode=755)

securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)

tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,seclabel)

devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,seclabel,gid=5,mode=620,ptmxmode=000)

tmpfs on /run type tmpfs (rw,nosuid,nodev,seclabel,mode=755)

tmpfs on /sys/fs/cgroup type tmpfs (rw,nosuid,nodev,noexec,seclabel,mode=755)

#####此处省略部分内容#####

【实例2】使用 tr 命令将 os.txt 文件中的内容转化为大写，并使用 sort 命令将内容排序，使用 uniq 命令去除重复的行，最后将输出重定向到文件 ox.txt.new。

[c.biancheng.net]$ cat os.txt

redhat

suse

centos

ubuntu

solaris

hp-ux

fedora

centos

redhat

hp-ux

[c.biancheng.net]$ tr a-z A-Z <os.txt | sort | uniq >os.txt.new

[c.biancheng.net]$ cat os.txt.new

CENTOS

FEDORA

HP-UX

REDHAT

SOLARIS

SUSE

UBUNTU

# **Linux Shell管道详解**

简单地说，过滤器可以概括为以下两点：

* 如果一个 [Linux 命令](http://c.biancheng.net/linux/" \t "http://c.biancheng.net/view/_blank)是从标准输入接收它的输入数据，并在标准输出上产生它的输出数据（结果），那么这个命令就被称为过滤器。
* 过滤器通常与 [Linux](http://c.biancheng.net/linux_tutorial/" \t "http://c.biancheng.net/view/_blank) 管道一起使用。

常用的被作为过滤器使用的命令如下所示：

|  |  |
| --- | --- |
| **命令** | **说明** |
| awk | 用于文本处理的解释性程序设计语言，通常被作为数据提取和报告的工具。 |
| cut | 用于将每个输入文件（如果没有指定文件则为标准输入）的每行的指定部分输出到标准输出。 |
| grep | 用于搜索一个或多个文件中匹配指定模式的行。 |
| tar | 用于归档文件的应用程序。 |
| head | 用于读取文件的开头部分（默认是 10 行）。如果没有指定文件，则从标准输入读取。 |
| paste | 用于合并文件的行。 |
| sed | 用于过滤和转换文本的流编辑器。 |
| sort | 用于对文本文件的行进行排序。 |
| split | 用于将文件分割成块。 |
| strings | 用于打印文件中可打印的字符串。 |
| tac | 与 cat 命令的功能相反，用于倒序地显示文件或连接文件。 |
| tail | 用于显示文件的结尾部分。 |
| tee | 用于从标准输入读取内容并写入到标准输出和文件。 |
| tr | 用于转换或删除字符。 |
| uniq | 用于报告或忽略重复的行。 |
| wc | 用于打印文件中的总行数、单词数或字节数。 |

接下来，我们通过几个实例来演示一下过滤器的使用。

## 在管道中使用 awk 命令

关于 awk 命令的具体用法，请大家自行学习，本节我们我们仅通过几个简单的实例来了解一下 awk 命令在管道中的使用。

#### 实例1

查看系统中的所有的账号名称，并按名称的字母顺序排序。

[c.biancheng.net]$ awk -F: '{print $1}' /etc/passwd | sort

adm

apache

avahi

avahi-autoipd

bin

daemon

dbus

ftp

games

...

在上例中，使用冒号 : 作为列分隔符，将文件 /etc/passwd 的内容分为了多列，并打印了第一列的信息（即用户名），然后将输出通过管道发送到了 sort 命令。

#### 实例2

列出当前账号最常使用的 10 个命令。

[c.biancheng.net]$ history | awk '{print $2}' | sort | uniq -c | sort -rn | head

140 echo

75 man

71 cat

63 su

53 ls

50 vi

47 cd

40 date

26 let

25 paste

在上例中，history 命令将输出通过管道发送到 awk 命令，awk 命令默认使用空格作为列分隔符，将 history 的输出分为了两列，并把第二列内容作为输出通过管道发送到了 sort 命令，使用 sort 命令进行排序后，再将输出通过管道发送到了 uniq 命令，使用 uniq 命令 统计了历史命令重复出现的次数，再用 sort 命令将 uniq 命令的输出按照重复次数从高到低排序，最后使用 head 命令默认列出前 10 个的信息。

#### 实例3

显示当前系统的总内存大小，单位为 KB。

[c.biancheng.net]$ free | grep Mem | awk '{print $2}'

2029860

## 在管道中使用 cut 命令

cut 命令被用于文本处理。你可以使用这个命令来提取文件中指定列的内容。

#### 实例1

查看系统中登录 [Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) 是“/bin/bash”的用户名和对应的用户主目录的信息：

[c.biancheng.net]$ grep "bin/bash" /etc/passwd | cut -d: -f1,6

root:/root

mozhiyan:/home/mozhiyan

如果你对 Linux 系统有所了解，你会知道，/etc/passwd 文件被用来存放用户账号的信息，此文件中的每一行会记录一个账号的信息，每个字段之间用冒号分隔，第一个字段即是账号的账户名，而第六个字段就是账号的主目录的路径。

#### 实例2

查看当前机器的CPU类型。

[c.biancheng.net]$ cat /proc/cpuinfo | grep name | cut -d: -f2 | uniq

Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz

上例中，执行命令cat /proc/cpuinfo | grep name得到的内容如下所示：

[c.biancheng.net]$ cat /proc/cpuinfo | grep name

model name : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz

model name : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz

model name : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz

model name : Intel(R) Core(TM) i5-2520M CPU 0 2.50GHz

然后，我们使用 cut 命令将上述输出内容以冒号作为分隔符，将内容分为了两列， 并显示第二列的内容，最后使用 uniq 命令去掉了重复的行。

#### 实例3

查看当前目录下的子目录数。

[c.biancheng.net]$ ls -l | cut -c 1 | grep d | wc -l

5

上述管道命令主要做了如下操作：

* 命令ls -l输出的内容中，每行的第一个字符表示文件的类型，如果第一个字符是d，就表示文件的类型是目录。
* 命令cut -c 1是截取每行的第一个字符。
* 命令grep d来获取文件类型是目录的行。
* 命令wc -l用来获得 grep 命令输出结果的行数，即目录个数。

## 在管道中使用grep命令

grep 命令是在管道中比较常用的一个命令。

#### 实例1

查看系统日志文件中的错误信息。

[c.biancheng.net]$ grep -i "error:" /var/log/messages | less

#### 实例2

查看系统中 HTTP 服务的进程信息。

[c.biancheng.net]$ ps auxwww | grep httpd

apache 18968 0.0 0.0 26472 10404 ? S Dec15 0:01 /usr/sbin/httpd

apache 18969 0.0 0.0 25528 8308 ? S Dec15 0:01 /usr/sbin/httpd

apache 18970 0.0 0.0 26596 10524 ? S Dec15 0:01 /usr/sbin/httpd

#### 实例3

查找我们的程序列表中所有命令名中包含关键字 zip 的命令。

[c.biancheng.net]$ ls /bin /usr/bin | sort | uniq | grep zip

bunzip2

bzip2

bzip2recover

gunzip

gzip

#### 实例4

查看系统安装的 kernel 版本及相关的 kernel 软件包。

[c.biancheng.net]$ rpm -qa | grep kernel

kernel-2.6.18-92.e15

kernel-debuginfo-2.6.18-92.e15

kernel-debuginfo-common-2.6.18-92.e15

kernel-devel-2.6.18-92.e15

#### 实例5

查找 /etc 目录下所有包含 IP 地址的文件。

[c.biancheng.net]$ find /etc -type f -exec grep '[0-9][0-9]\*[.][0-9][0-9]\*[.][0-9][0-9]\*[.][0-9][0-9]\*' {} \;

## 在管道中使用 tar 命令

tar 命令是 Linux 系统中最常用的打包文件的程序。

#### 实例1

你可以使用 tar 命令复制一个目录的整体结构。

[c.biancheng.net]$ tar cf - /home/mozhiyan | ( cd /backup/; tar xf - )

#### 实例2

跨网络地复制一个目录的整体结构。

[c.biancheng.net]$ tar cf - /home/mozhiyan | ssh remote\_host "( cd /backup/; tar xf - )"

#### 实例3

跨网络地压缩复制一个目录的整体结构。

[c.biancheng.net]$ tar czf - /home/mozhiyan | ssh remote\_host "( cd /backup/; tar xzf - )"

#### 实例4

检査 tar 归档文件的大小，单位为字节。

[c.biancheng.net]$ cd /; tar cf - etc | wc -c

215040

#### 实例5

检查 tar 归档文件压缩为 tar.gz 归裆文件后所占的大小。

[c.biancheng.net]$ tar czf - etc.tar | wc -c

58006

#### 实例6

检查 tar 归档文件压缩为 tar.bz2 归裆文件后所占的大小。

[c.biancheng.net]$ tar cjf - etc.tar | wc -c

50708

## 在管道中使用 head 命令

有时，你不需要一个命令的全部输出，可能只需要命令的前几行输出。这时，就可以使用 head 命令，它只打印命令的前几行输出。默认的输出行数为 10 行。

#### 实例1

显示 ls 命令的前 10 行输出。

[c.biancheng.net]$ ls /usr/bin | head

addftinfo

afmtodit

apropos

arch

ash

awk

base64

basename

bash

bashbug

#### 实例2

显示 ls 命令的前 5 行内容。

[c.biancheng.net]$ ls / | head -n 5

bin

cygdrive

Cygwin.bat

Cygwin.ico

Cygwin-Terminal.ico

## 在管道中使用 uniq 命令

uniq 命令用于报告或删除重复的行。我们将使用一个测试文件进行管道中使用 uniq 命令的实例讲解，其内容如下所示：

[c.biancheng.net]$ cat testfile

This line occurs only once.

This line occurs twice.

This line occurs twice.

This line occurs three times.

This line occurs three times.

This line occurs three times.

#### 实例1

去掉输出中重复的行。

[c.biancheng.net]$ sort testfile | uniq

This line occurs only once.

This line occurs three times.

This line occurs twice.

#### 实例2

显示输出中各重复的行出现的次数，并按次数多少倒序显示。

[c.biancheng.net]$ sort testfile | uniq -c | sort -nr

3 This line occurs three times.

2 This line occurs twice.

1 This line occurs only once.

## 在管道中使用 wc 命令

wc 命令用于统计包含在文本流中的字符数、单同数和行数。

#### 实例1

统计当前登录到系统的用户数。

[c.biancheng.net]$ who | wc -l

#### 实例2

统计当前的 Linux 系统中的进程数。

[c.biancheng.net]$ ps -ef | wc -l

# **组命令和子shell**

组命令和子shell

在bash中，有两种方式可以将命令组合到一起，一种是组命令，另一种是子shell。格式如下：

# 组命令

{ commnad1; command2; [ command3.. ] }

# 子shell

(command1; command2; [command3..])

上面两种形式的差别在于：1.组命令使用花括号括起来，前花括号和后花括号与命令之间要有一个空格，并且闭合花括号前要用分号或换行结束命令。

2.对于子shell，只需要用圆括号包围即可。

执行重定向

那么，组命令和子shell有什么用途呢？尽管它们有一个主要的区别，但是它们都可以用来管理重定向。下面请看这个例子：

[fbap:/home/fbap/usertmp]>ls -l > output.txt

[fbap:/home/fbap/usertmp]>echo "hello world" >> output.txt

[fbap:/home/fbap/usertmp]>cat foo.txt >> output.txt

显然，上面的3条命令将输出重定向到output.txt。使用组命令，可以将它们合并成一行：

{ ls -l; echo "hello world"; cat foo.txt; } > output.txt // { } 要有空格

1

当然，你也可以使用子shell的方式：

(ls -l; echo "hello world"; cat foo.txt;) > output.txt

1

在上面的例子中，使用组命令和子shell可以减少一些输入。但是，组命令和子shell真正有价值的地方在于管道的使用。当创建命令管道时，通常将多条命令的结果输出到一条流中，这很有用。

{ ls -l; echo "hello world"; cat foo.txt; } | lpr

这里，我们将3个命令的输出进行合并，并通过管道输出到lpr的输入以产生一个打印报告。

**进程替换**

虽然组命令和子shell看起来很相似，都可以用来为重定向整合流，但是，它们有一处主要的不同。子shell（正如其名字）在当前shell的子拷贝中执行命令，而组命令则是直接在当前shell中执行所有命令。这以为子shell会复制当前环境变量以创建一个新的shell实例。当子shell退出时，复制的环境变量也就消失了，因此对任何子shell（包括变量赋值）的改变也同样丢失了。所以，大多数情况下，除非脚本需要子shell，否则使用组命令比子shell更快，占用内存也更少。

请看下面的例子：

echo 'foo' | read

echo $REPLY

在上面的例子中，REPLY的内容总是空的，因为read命令是在子shell中执行的，并且当子shell终止的时候，REPLY的拷贝也遭到了破坏。

很幸运的是，shell提供了一种叫做进程替换的外部扩展方式来解决这个问题。

实现进程替换的方式有两种：

# 产生标准输出的进程

<(list)

# 吸纳标准输入的命令

>(list)

为了解决上述read命令的问题，我们可以像这样使用进程替换：

read < <(echo "foo")

echo $REPLY

进程替换允许将子shell当成普通的文件，目的是为了重定向。事实上，这是一种扩展形式，我们可以查看它的真实值。

[fbap:/home/fbap]>echo <(echo "foo")

/dev/fd/63

通过使用echo查看扩展结果，可以看到文件/dev/fd/63正为子shell提供输出。

**以下是一个读循环的实例，该实例用循环处理子shell创建的目录列表的内容**：

#!/bin/bash

# 注：

# 1.如果将此处的内容另存为一个脚本如sub-proc.sh,但是如果像这样执行脚本：sh sub-proc.sh，shell将报错syntax error（语法错误）。

#

# 正确的执行方式是：

# 1)脚本添加可执行权限： chmod u+x sub-proc.sh

# 2)使用相对（绝对）路径： ./sub-proc.sh

while read attr links owner group size date\_1 date\_2 year filename

do

cat <<EOF

文件名: ${filename}

大小: ${size}

拥有者: ${owner}

属性: ${attr}

EOF

echo "======================="

done < <(ls -l | tail -n +2)

执行命令：./sub-proc.sh | head -n 10，得到如下输出：

[fbap:/home/fbap/usertmp]>./sub\*sh|head -n 10

文件名: AFA\_TEST\_NEWADM.ixf

大小: 4783

拥有者: fbap

属性: -rw-rw-r--.

=======================

文件名: all\_python\_files.log

大小: 1168088

拥有者: fbap

属性: -rw-rw-r--.

=======================