

BASH 编程 — 介绍如何编程

Miek . G (mikkey@dynamo.com.ar)
李峰(中文) (lifeng@telebyte.com.cn)

2007年7月27日 09:36:18 (奥地利时间)
2007年10月21日 08:17:28 (北京时间)

这是一篇帮助你学习基本的 bash 交互式脚本编程的文章。本文不想变成一个介绍 bash 高级功能的文章(请看章节标题)。我不是 bash 的专家或权威。我决定写这篇文章是因为可以通过这种方式学习很多有用的知识，并且我认为其它人可以通过这篇文章的内容受益。在这里我欢迎大家对本文提出指正意见，尤其是以 patch 格式的更正更受欢迎。

目 录

索 引	1
1 介 绍	4
1.1 最新版本	4
1.2 需掌握知识点	4
1.3 文档的使用	4
2 非常简单的脚本	4
2.1 传统的 hello world 脚本	4
2.2 一个非常简单的备份脚本	5
3 重定向	5
3.1 理论和快速参考	5
3.2 示例: stdout 重定向到文件	5
3.3 示例: stderr 重定向到文件	5
3.4 示例: stdout 重定向到 stderr	6
3.5 示例: stderr 重定向到 stdout	6
3.6 示例: stderr 和 stdout 重定向到文件	6

目 录	2
4 管道	6
4.1 管道是什么？你为什么想用它	6
4.2 示例：同 sed 一齐使用的管道示例	6
4.3 示例：另一个示例查询 <code>ls -l *.txt</code>	7
5 变量	7
5.1 示例：使用变量编写的 Hello World! 程序	7
5.2 示例：非常简单的备份脚本(后面会有更好的版本)	7
5.3 本地变量	8
6 条件	8
6.1 枯燥的理论知识	8
6.2 示例：基本的 <code>if...then...else</code>	8
6.3 示例：有变量的条件语句	9
7 for,while 和 until 循环	9
7.1 示例	9
7.2 类 C 的 for	10
7.3 while 示例	10
8 函数	10
8.1 函数示例	10
8.2 有参数的函数的示例	11
9 使用 select 来产生一个简单的菜单	11
9.1 使用命令行	12
10 杂 录	12
10.1 用 read 读用户输入信息	12
10.2 算术计算	13
10.3 查找 bash	13
10.4 从程序中得到返回值	14
10.5 捕捉一个命令的输出	14
10.6 操作多个文件	14
10.7 串比较示例	15
10.8 算术操作符	15
10.9 算术关系操作	16
10.10 有用的命令	16

目 录	3
11 更多的脚本	19
11.1 应用一个命令到所有的文件和目录	19
11.2 示例：一个非常简单的备份脚本(比以前的版本更好些)	19
11.3 文件重命名	20
11.4 文件重命名(简单)	22
12 当有错误时(调试)	23
12.1 调用 BASH 的方法	23
13 关于这篇文档	23
13.1 不担保条款	23
13.2 翻 译	23
13.3 鸣 谢	23
13.4 本文历史	24
13.5 更多资源	24

1 介绍

1.1 最新版本

<http://www.linuxdoc.org/HOWTO/Bash-Prog-Intro-HOWTO.html>

1.2 需掌握知识点

熟悉 GNU/Linux 命令行，如果能掌握基本的编程概念将对于理解本文更有帮助。然而这不是一篇编程指导，它解释(或至少试图)许多基本概念。

1.3 文档的使用

这篇文档将尝试在下列情况下被使用:

- 你有编程的想法，并且你想开始用 bash 来编写代码。
- 你想查看一些 shell 程序代码和注释来开始你自己的代码编写工作。
- 你正从 DOS/Windows 系统(或已迁移到新系统上)迁到新系统中，并想进行“批处理”工作。
- 你完全就是一个书呆子，并读每一篇可得到的 how-to 文档。

2 非常简单的脚本

这篇 HOW-TO 文档将尝试由示例代码开始给你一些关于 shell 脚本程序方面的提示。在这节中你将发现一些脚本将帮助你理解一些技术。

2.1 传统的 hello world 脚本

```
#!/bin/bash
echo Hello World
```

这个脚本仅有两行。第一行指示操作系统用什么程序来运行这个文件。第二行是这个脚本所执行的唯一的动作，它打印‘Hello World’到终端。

如果你在终端中看到一些提示如：`./hello.sh: Command not found.` 或许代码的第一行‘#!/bin/bash’指示的 bash 程序的位置错误，你可以用 `whereis bash` 来得到 bash 的位置或在你的操作系统中查找其位置。

2.2 一个非常简单的备份脚本

```
#!/bin/bash
tar -cZf /var/my-backup.tgz /home/me/
```

在这个脚本中，做为将信息打印到终端的替换，我们将在用户的家目录中创建一个 tar 压缩包。这个脚本的目的不是打算被实际使用，更为实用的备份脚本将在这篇文章后面的章节中出现。

3 重定向

3.1 理论和快速参考

这里有 3 个文件说明符，stdin, stdout 和 stderr (标准的)基本上你可以：

1. 重定向 stdout 到一个文件
2. 重定向 stderr 到一个文件
3. 重定向 stdout 到一个 stderr
4. 重定向 stderr 到一个 stdout
5. 重定向 stderr 和 stdout 到一个文件
6. 重定向 stderr 和 stdout 到 stdout
7. 重定向 stderr 和 stdout 到 stderr

1‘代表’ stdout 并且 2‘代表’ stderr 。

关于重定向有一些需要注意的问题：使用 less 命令你可以查看将打印在屏幕上的(存在于缓存中) stdout 和 stderr，但当你试图‘浏览’时相关信息会被擦除。

3.2 示例: stdout 重定向到文件

这将使一个程序的输出信息写到一个文件中。

```
ls -l > ls-l.txt
```

在这里，一个被命名为‘ls-l.txt’的文件将被创建，并且其内容如同你在屏幕上看到命令‘ls -l’执行时显示的内容一样。

3.3 示例: stderr 重定向到文件

下面将引起一个程序的错误输出到一个文件中

```
grep da * 2> grep-errors.txt
```

在这里，一个被称作‘grep-errors.txt’的文件将被创建并且内容中命令‘grep da *’向标准终端输出的错误信息。

3.4 示例: `stdout` 重定向到 `stderr`

下面将引起一个程序的 '`stderr`' 被写到与 '`stdout`' 相同的文件说明符中。

```
grep da * 1>&2
```

这里, 一个程序的 `stdout` 的输出将被发送到 `stderr` 中, 你或许发现了这是用不同的方式来实现的。

3.5 示例: `stderr` 重定向到 `stdout`

这里的示例将使一个程序的 `stderr` 的输出发送到与 `stdout` 相同的文件说明符中。

```
grep * 2>&1
```

这里, 一个命令的错误信息(`stderr`)被发送到标准输出(`stdout`)文件说明符中。如果你使用一个管道将信息发送到 `less` 命令中。你将看到有些行'消失'了。(因为有些信息被写到 `stderr` 中)但在这里, 这些行被显示出来(因为现在它们被输出到了 `stdout` 中)。

3.6 示例: `stderr` 和 `stdout` 重定向到文件

下面的示例将一个程序的所有输出写到一个文件中。这对于列在 `cron` 条目中的程序来说是非常合适的。如果你想让一个命令以非常安静的方式运行。

```
rm -f $(find / -name core ) &> /dev/null
```

在这里(考虑在一个 `cron` 条目中)将删除所有目录下, 被命名为 '`core`' 的文件。注意你想将这个命令的输出清除, 请确保该命令被运行。

4 管道

本节将用非常简单和实用的方法来介绍如何使用管道, 以及为什么你想用它。

4.1 管道是什么? 你为什么想用它

管道(非常简单, 我坚持认为)是将一个程序的输出当做另一个程序的输入的一种手段。

4.2 示例: 同 `sed` 一齐使用的管道示例

这里是一个非常简单的使用管道的示例。

```
ls -l | sed -e 's/[aeio]/u/g'
```

在这里, 下列的事情按顺序发生: 首先命令 `ls -l` 被执行, 并且它的输出, 不是被打印出来, 而是(通过管道)被发送到 `sed` 程序, 依次处理, 并打印处理后的结果。

4.3 示例：另一个示例查询 `ls -l *.txt`

或许，这是一个更为困难的方式来运行 `ls -l *.txt`。但这里为了说明如何使用管道，而不是解决如列表困境。

```
ls -l |grep '\.txt$'
```

在这里，程序 `ls -l` 的输出被发送到程序 `grep` 中，之后，依次打印匹配规则表达式 `'\.txt$'` 的行。

5 变量

在 `bash` 中，你可以象其它任何程序语言一样使用变量。在 `bash` 中，没有数据类型这个概念。在 `bash` 中的一个变量可以存放一个数字、字符、字符串。

你不需声明一个变量，仅需要为变量分配一个值，这时就可创建变量。

5.1 示例：使用变量编写的 **Hello World!** 程序

```
#!/bin/bash
STR='Hello World!'
```

```
echo $STR
```

第二行创建一个被称作 `STR` 的变量并分配一个字符串“Hello World!”到这个变量中。之后，这个变量的值可以通过在变量名前加一个 `$` 来取得。请注意(试一下!)如果你不用 `$` 符号，程序输出将不一样，并将不是你想得到的结果。

5.2 示例：非常简单的备份脚本(后面会有更好的版本)

```
#!/bin/bash
OF=/var/my-backup-$(date +%Y%m%d).tgz
tar -czf $OF /home/me/
```

这个脚本介绍了另外一些事，首先，你将在代码的第二行熟悉变量的创建和变量值的分配。如果你运行这段代码，请注意表达式。你将注意到在圆括号内运行的命令，并找出其输出。

在这个脚本中请注意，输出的文件名每天都将不同，由于 `date` 命令格式切换的原因(`+%Y%m%d`)，你可以通过指定不同的格式来改变其值。

另外的一些示例：

```
echo ls
echo $(ls)
```

5.3 本地变量

本地变量(Local variables)可以通过使用关键字 `local` 来创建。

```
#!/bin/bash
HELLO=Hello
function hello {
  local HELLO=World
  echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

这个示例将帮助你理解如何使用本地变量。

6 条件

条件(Conditionals)将允许你来决定是否执行一个动作。是否执行是通过判断一个表达式来决定的。

6.1 枯燥的理论知识

条件有许多种格式。最基本的格式是：`if` 表达式 `then` 语句。当 `if` ‘表达式’ 被评估为 `true` 时，语句被执行。

条件还有其它的格式如：`if` 表达式 `then` 语句1 `else` 语句2。这种情况，如果‘表达式’值为 `true` 时，语句1被执行，否则，语句2被执行。

这里还有其它的形式：`if` 表达式1 `then` 语句1 `else if` 表达式2 `then` 语句2 `else` 语句3。在这个格式中，添加了“`ELSE IF` ‘表达式2’ `THEN` ‘语句2’”，这时如果表达式2的值被测试为 `true`，然后语句2将会被执行。其余的事你可以想象一下(看前面的格式)。

关于语法的一句话总结是：

在 `bash` 中其本的‘`if`’结构如下：

```
if [表达式];
then
  当 if 表达式为真时的代码
fi
```

6.2 示例：基本的 `if...then...else`

```
#!/bin/bash
if [ ‘foo’ = ‘foo’ ]; then
```

```
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

6.3 示例：有变量的条件语句

```
#!/bin/bash
T1='foo'
T2='bar'
if [ '$T1' = 'T2' ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

7 for,while 和 until 循环

在这节，你将看到 for,while 和 until 循环的示例。

相对于其它的程序语言来说，for 循环的语法格式有些不一样，这需要你牢牢记住。

while 是一段当控制表达式为 true 时所执行的代码段，并且当表达式值为 false 时(或当执行的代码意外中断时)停止执行。

until 循环几乎与 while 循环一样，不同之处是表达式的值为 false 时语句被执行。

如果你对于 while 和 until 语句如此相同感到奇怪，那你的感觉是对的。

7.1 示例

```
#!/bin/bash
for i in $( ls ); do
    echo item $i
done
```

在第二行中，我们声明 i 是一个变量，并且这个变量分别保存 ls 命令执行后的查询信息的不同值。

第三行如果需可，可以写的更长些，或在 done 行前，可以有更多的代码行。

之后结束第三行的语句，并且 \$i 取一个新值。

这个脚本有些奇怪，但一个更加有用的方式来使用 for 循环将用它来匹配前面的示例中的部分文件。

7.2 类 C 的 for

fish 建议添加这个格式的循环。它是一个更象 C/perl...for 循环。

```
#!/bin/bash
for i in `seq 1 10`;
do
    echo $i
done
```

7.3 while 示例

下面是 while 示例

```
#!/bin/bash
COUNTER=20
until [ $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
let COUNTER--
done
```

8 函数

象几乎所有的程序语言一样，你可以使用函数将一组代码用更加有逻辑的方式组织起来，或实践神奇的循环功能。

声名一个函数的方式仅是写如下的代码：

```
function my_func {
    my_code
}
```

调用函数如同调用其它程序一样，你只需写函数的名字。

8.1 函数示例

```
#!/bin/bash
function quit {
    exit
}
```

```
function hello {
    echo Hello!
}
hello
quit
echo foo
```

行 2-4 存在一个‘quit’函数。行 5-7 存在一个‘hello’函数。如果你不是十分确定这个脚本的执行情况，请试着运行它一下！

注意函数不需要以特殊顺序声明。

当运行这个脚本，你将首先将注意到：函数‘hello’被调用，其次被调用的是‘quit’函数，并且程序不会运行到第 10 行。

8.2 有参数的函数的示例

```
#!/bin/bash
function quit {
    exit
}
function e {
    echo $1
}
e Hello
e World
quit
echo foo
```

这个程序脚本几乎和前一个程序一样。主要的不同是函数‘e’。这个函数打印出它收到的第一个参数，在函数内，函数如同脚本参数一样的方式来对待函数参数。

9 使用 select 来产生一个简单的菜单

```
#!/bin/bash
OPTIONS='Hello Quit'
select opt in $OPTIONS; do
    if [ "$opt" = "Quit" ]; then
echo done
exit
elif [ "$opt" = "Hello" ]; then
```

```
        echo Hello World
    else
        clear
        echo bad option
    fi
done
```

如果你运行这个脚本，你将看到程序员们所梦想的文本界面的菜单。你也可能注意到相对于‘for’结构来说这两段代码的结构非常相似，唯一的不同于for循环的是每个在\$OPTIONS变量中的单词。

9.1 使用命令行

```
#!/bin/bash
if [ -z "$1" ]; then
    echo usage: $0 directory
exit
fi
SRCD=$1
TGTD=''/var/backups/'
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

首要的一点是你必需明白这个脚本在做什么。第一个条件测试的表达式是看你的程序是否收到一个参数(\$1)，如果没有收到，程序将停止执行并关闭，并且向用户显示一些有用的信息。程序的其余部分将从这个语句开始停止。

10 杂录

10.1 用 read 读用户输入信息

有许多时候，你想提示用户输入一些你想要的信息，并且有几种方法来实现读用户输入的需求。下面是一个示例：

```
#!/bin/bash
echo Please enter your name
read NAME
echo 'Hi $NAME!'
```

作为一种变体方式，你可以用 `read` 来读多个值，这个例子可以澄清你的疑问。

```
#!/bin/bash
echo Please,enter your firstname and lastname
read FN LN
echo ‘‘Hi! $LN, $FN !’’
```

10.2 算术计算

在命令行模式下(或 shell)模式下，试着运行：

```
echo 1 + 1
```

从算术的角度来看，你希望能看到‘2’。但事实将令你失望。如何让 BASH 来计算你手头的一些数字？解决的方法是：

```
echo $((1+1))
```

这将产生一个更为‘逻辑’的输出。它将计算表达式的值。你也可以通过下面的方式来构建算术表达式。

```
echo ${1+1}
```

如果你需要使用函数，或需要更多的数学计算，你可以使用 `bc` 来计算数学表达式。

如果我在命令行上输入 `echo ${3/4}`，它将返回值 0。因为 `bash` 此时只使用整形数据。如果你运行 `echo 3/4bc -l`，些时将返回值 0.75。

10.3 查找 `bash`

从 mike 的信息(查看 Thanks)

你通常使用 `#!/bin/bash` 你可能需要一个如何发现 `bash` 位置的示例。

建议从下列的位置查找：

```
ls -l /bin/bash
ls -l /sbin/bash

ls -l /usr/local/bin/bash
ls -l /usr/bin/bash
ls -l /usr/sbin/bash
ls -l /usr/local/sbin/bash
```

(我不能想象出还有其它的位置了，我已在不同的系统中常用的位置都找过了 bash 程序)
你可以用 ‘which bash’ 查找 bash 在当前系统中的位置。

10.4 从程序中得到返回值

在 bash 中，一个程序的返回值被存储在一个被称做 \$? 的特殊变量中。

为了说明如何捕一个程序的返回值。我假设目录数据不存在。(这同时也是 mike 建议的)

```
#!/bin/bash
cd /data $? /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

10.5 捕捉一个命令的输出

这个小的脚本显示数据库中的所有表(假设你安装了 MySQL 数据库系统)。另外，请考虑改变 ‘mysql’ 命令使用一个有效的用户名和密码。

```
#!/bin/bash
DBS='mysql -uroot -e''show databases''
for b in $DBS
do
    mysql -uroot -e''show tables from $b''
done
```

10.6 操作多个文件

你可以用命令 source 来使用多个文件。

```
__TO-DO__

11

Tables
11.1
```

串比较操作:

```
(1) s1 = s2
(2) s1 != s2
(3) s1 < s2
(4) s1 > s2
(5) -n s1
(6) -z s1
```

```
(1) s1 matches s2
(2) s1 does not match s2
(3) __TO-DO__
(4) __TO-DO__
(5) s1 is not null (存在一或多个字符)
(6) s1 is null
```

10.7 串比较示例

比较两个串：

```
#!/bin/bash
S1='string'
S2='String'
if [ $S1=$S2 ];
then
    echo "$1('$S1') is not equal to $2('$S2')"
fi
if [ $S1=$S1 ];
then
    echo "$1('$S1') is equal to $1('$S1')"
fi
```

我从一封邮件中引用一点需要注意的地方，这封邮件发自 Andreas Beck,是关于使用 `if [$1 = $2]`。

这种方式不是一个好的主意，当如果 `$S1` 或 `$S2` 为空时，你将得到一个语法错误的提示。`x$1=x$2` 或 `"$1"="$2"` 是更好的方式。

10.8 算术操作符

+

```
-
*
/
% (取余)
```

10.9 算术关系操作

```
-lt (<)
-gt (>)
-le (<=)
-ge (>=)
-eq (==)
-ne (!=)
```

对于C程序员来说，只需简单地将相应的操作符与括号中的符号对应就可以了。

10.10 有用的命令

这节被 Keeps 重写。(查看鸣谢一节)

这些命令中的一部分几乎存在于完整的程序语言语法中。这些命令的格式仅基本上被解释。对于更详细的说明，请阅相应的 man 手册页。

sed (流编辑器)

Sed 是一个非交互式编辑器。相对于将光标从屏幕上移动的方式来修改文件，你可以使用脚本来编辑 sed 指令，另外加上要修改的文件。你可以将 sed 描述成一个过滤器。让我们看一些示例：

```
$sed 's/to_be_replaced/replaced/g' /tmp/dummy
```

Sed 将替换串 'to_be_replaced' 到串 'replaced' 并且读文件 /tmp/dummy。最终的结果将发送到 stdout (标准输出文件说明符，通常这个文件说明符是 console)。但你也可以加上 '> capture' 到上面的代码的行最后，此时，sed 将输出到文件 'capture' 中。

```
$sed 12, 18d /tmp/dummy
```

Sed 将显示除了 12 和 18 行外的所有行。原文件将不被 sed 命令所修改。

awk (维护数据库文件，文本恢复和处理)

现在有许多对可以执行 AWK 语言的程序(最出名的是解释器是 GNU 的 gawk 和 'new awk')。其原则是由 AWK 简单对一种匹配规则进行扫描，对于每一个匹配执行一个动作。

另外，我先创建一个模形文件存在如下的行：

```
''test123
test
tteesstt''

$awk '/test/ {print}' /tmp/dummy

test123

test
```

对于 AWK 来说，需要处理的匹配参数是 test 并且对于这个匹配参数的处理是当在文件 /tmp/dummy 中发现串 'test' 则 '打印'。

```
$awk '/test/ {i=i+1} END {print i}' /tmp/dummy

2
```

当你搜索大量的匹配时，你可以将引号中的文本替换为 '-f file.awk' 以便你将所有的匹配条件和处理行为放入一个 'file.awk' 文件中。

grep (打印匹配的搜索条件的行)

我们在前面的章节中已看到了 grep 命令，它将显示匹配搜索条件的行。但是除此之外，grep 还可以干很多的事。

```
$grep 'look for this' /var/log/messages -c

12
```

这显示串 "look for this" 在文件 /var/log/messages 中出现了 12 次。

[好了，这里的示例是假的，其中文件 /var/log/messages 被节选了 :-)]

wc (计算行数，词和位数)

在下面的示例中，我们将看到输出不是我们想要的。文件 dummy 在这个示例中使用的文件，存在下面的文本：

```
''bash introduction
howto test file''
```

我们使用下面的命令来查看结果：

```
$wc --words --lines --bytes /tmp/dummy  
2 5 34 /tmp/dummy
```

我们不关心参数的顺序。`wc`总是按标准的顺序打印它们。这样，就如同你看到的一样。

`sort` (对文本文件进行排序)

这次，我们的测试文件存在如下的内容：

```
‘‘b  
c  
a’’
```

我们用下面的命令来查看结果：

```
$sort /tmp/dummy
```

通过上面的命令，我们可以得到如下的结果：

```
a  
b  
c
```

命令不是同`bc` (一个数学计算程序)一样容易 :-)

但是它从命令行接受计算 (从文件中接受输入，而不是从重定向(`redirector`)或管道(`pipe`)接受输入，但也可以从用户接口中接受。下面的演示显示了部分命令。注意，我在开始时，使用了`bc`的`-q`参数以避免显示欢迎信息。

```
$bc -q  
1 == 5  
0  
0.05 == 0.05  
1  
5 != 5  
0
```

```

2 ^ 8
256
sqrt(9)
3
while (i !=9) {
i = i+1
print i
}
123456789
quit

```

tput (初始化一个终端或查询终端信息数据库)
下面是一个小的显示 tput 能力的示例:

```
$tput cup 10 4
```

此时, 提示符将在 (y10,x4) 的坐标处显示。

```
$tput reset
```

清空屏幕并且提示符将在 (y1,x1) 处显示。注意 (y0,x0) 是屏幕的左上角。

```
$tput cols
80
```

显示在 x 轴方向上可能的字符数。

强烈推荐你(至少)熟悉这些程序。这里有大量的小程序将会让你在命令行模式下做许多神奇的工作。

[有些示例得自 man 手册或常见问题解答 (FAQs)]

11 更多的脚本

11.1 应用一个命令到所有的文件和目录

11.2 示例: 一个非常简单的备份脚本 (比以前的版本更好些)

```

#!/bin/bash
SRCD='' /home/''
TGTD='' /var/backup/''
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD

```

11.3 文件 重命名

```
#!/bin/sh
# renna: rename multiple files according to several rules
# written by felix hudson Jan - 2000

#first check for the various 'modes' that this program has
#if the first ($1) condition matches then we execute that portion of the
#program and then exit

# check for the prefix condition
if [ $1 = p ]; then

#we now get rid of the mode ($1) variable and prefix ($2)
prefix=$2 ; shift ; shift

# a quick check to see if any files were given
# if none then its better not to do anything than rename some non-existent
# files!!

if [ $1 = ]; then
echo "no files given"
exit 0
fi

# this for loop iterates through all of the files that we gave the program
# it does one rename per file given
for file in $*
do
mv ${file} $prefix$file
done

#we now exit the program
exit 0
fi

# check for a suffix rename
# the rest of this part is virtually identical to the previous section
# please see those notes
```

```
if [ $1 = s ]; then
suffix=$2 ; shift ; shift

if [ $1 = ]; then
echo "no files given"
exit 0
fi

for file in $*
do
mv ${file} $file$suffix
done

exit 0
fi

# check for the replacement rename
if [ $1 = r ]; then

shift

# i included this bit as to not damage any files if the user does not specify
# anything to be done
# just a safety measure

if [ $# -lt 3 ] ; then
echo "usage: renna r [expression] [replacement] files... "
exit 0
fi
# remove other information
OLD=$1 ; NEW=$2 ; shift ; shift

# this for loop iterates through all of the files that we give the program
# it does one rename per file given using the program 'sed'
# this is a single command line program that parses standard input and
# replaces a set expression with a give string
# here we pass it the file name ( as standard input) and replace the nessesary
# text

for file in $*
do
```

```
new='echo ${file} | sed s/${OLD}/${NEW}/g'
mv ${file} $new
done
exit 0
fi

# if we have reached here then nothing proper was passed to the program
# so we tell the user how to use it
echo "usage;"
echo " renna p [prefix] files.."
echo " renna s [suffix] files.."
echo " renna r [expression] [replacement] files.."
exit 0

# done!
```

11.4 文件重命名(简单)

```
#!/bin/bash
# renames.sh
# basic file renamer

criteria=$1
re_match=$2
replace=$3

for i in $( ls *$criteria* );
do
    src=$i
    tgt=$(echo $i | sed -e 's/$re_match/$replace/')
    mv $src $tgt
done
```

12 当有错误时(调试)

12.1 调用 BASH 的方法

```
#!/bin/bash -x
```

这将产生许多有意思的信息。

13 关于这篇文档

欢迎对这篇文档提出建议/更正，或当你看这篇文档时的灵感。我将尽可能快的更新这篇文档。

13.1 不担保条款

这篇文档不提供任何形式的担保。

13.2 翻译

意大利语：由 William Ghelfi (wizzy at tiscalinet.it) 翻译

法语：由 Laurent Martelli 完成

韩语：Minseok Park <http://kldp.org>

韩语：Chun Hye Jin 联系方式未知

西班牙语：作者未知 <http://www.insflug.org>

中文简体：李峰 <http://lifeng.telebyte.com.cn>

我猜想还会有更多语言的翻译版本，但我手头相关的信息不足，如果你有相关翻译的信息，请通过邮件通知我，以便于我更新这节的信息。

13.3 鸣谢

在这里感谢下列各位：

- 翻译这篇文档到其它语言的人(查看前面的小节)
- Nathan Hurst 对这篇文档提出了许多的更正意见
- Jon Abbott 对于计算数学表达式提出了评论
- Felix Hudson 编写了 rena 脚本
- Kees van den Broek (发来了许多的评论意见并重写了有用的命令一节)

- Mike (pink) 对于 bash 的位置和定位提出了一些建议
- Fiesh 对于循环一节提出了一些很好的建议
- Lion 建议注意通常错误的处理 (`./hello.sh:Command not found.`)
- Andress Beck 提出了一些更正和意见

13.4 本文历史

- 最新的翻译信息和小量的更正
- 由 Kess 重写的有用的命令一节被添加
- 包含更多的建议和更正
- 串比较示例被添加
- v0.8 删除了版本号, 我猜有日期就足够了
- v0.7 进行了许多更正, 并且老的 如何完成 一节被写完
- v0.6 小量的更正
- v0.5 添加了重定向小节
- v0.4 由于我前老板的原因, 文章从以前的位置上消失, 并且发现了新的合适的 [url:www.linuxdoc.org](http://www.linuxdoc.org)
- 更早期版本: 我记不清了并且我没有用 rcs 或 cvs :(

13.5 更多资源

介绍 bash (下面可替换资源)

- <http://org.laol.net/lamug/beforever/bashtut.htm>
- Bourne Shell Programming <http://207.213.123.70/book/>

索引

Thanks, 13

本地变量(Local variables), 8

条件(Conditionals), 8