BASH Programming - Introduction HOW-TO

Автор - Mike G `mikkey at dynamo.com.ar`
Переводчик - Дмитрий А. Смирнов, `das@cabel.net`

Thu Jul 27 09:36:18 ART 2000

Данная статья предназначена для оказания Вам помощи в программировании shell-скриптов начального и среднего уровня. Она не претендует на то, чтобы быть совершенным руководством (см. заглавие). Автор НЕ ЯВЛЯЕТСЯ ни экспертом, ни гуру в shell-программировании. Он решил написать эту статью по причине изучения им большого количества вопросов, которые могут быть полезны другим людям. Приветствуются любые предложения, рекомендации и комментарии относительно этого документа, особенно, в patch-форме :-)

---

1. Введение

1.1 Получение последних версий

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

1.2 Что требуется?

Необходимо ознакомиться с командной строкой, а также с основными концепциями программирования. Несмотря на то, что это не является учебником по программированию, здесь объясняются (или, по крайней мере, осуществляется попытка объяснить) многие основные концепции.

1.3 Использование данного документа

Данный документ может быть необходим в следующих ситуациях:

- У Вас имеются идеи, связанные с программированием, и существует необходимость в выполнении процесса кодирования каких-нибудь shell-скриптов.
- Ваши идеи, связанные с программированием, недостаточно конкретны и требуют дополнительных указаний.
- Вы желаете взглянуть на какие-нибудь shell-скрипты и комментарии в качестве образца для создания своих собственных.
- Вы мигрируете из DOS/Windows (или уже выполнили это) и хотите создавать файлы пакетной обработки ("batch").
- Вы - полный нерд и читаете любой попавший под руку how-to.

2. Простейшие скрипты

В данном HOW-TO осуществляется попытка предоставить Вам некоторые рекомендации по shell-программированию, основанные только на примерах.

В данном разделе Вы обнаружите небольшие скрипты, которые, вероятно, будут Вам полезны при освоении некоторых приёмов.

2.1 Традиционный скрипт "hello world"

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

Данный скрипт содержит только две строки. Первая сообщает системе о том, какая программа используется для запуска файла.

Вторая строка - это единственное действие, выполняемое данным скриптом, печатающее 'Hello world' на терминале.

Если Вы получите что-то типа *./hello.sh: Command not found.* , то, возможно, первая строка '#!/bin/bash' неправильная; запустите `whereis bash` или посмотрите `finding bash`, чтобы выяснить, какой должна быть эта строка.

2.2 Простой скрипт резервного копирования

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

В данном скрипте вместо печати сообщения на терминале мы создаём tar-архив пользовательского домашнего каталога. Скрипт НЕ предназначен для практического применения. Далее в данном документе будет представлен более эффективный скрипт резервного копирования.

3. Всё о перенаправлении

3.1 Теория и быстрый просмотр

Существуют 3 файловых дескриптора: stdin - стандартный ввод, stdout - стандартный вывод и stderr - стандартный поток ошибок.

Ваши основные возможности:

1. перенаправлять stdout в файл
2. перенаправлять stderr в файл
3. перенаправлять stdout в stderr
4. перенаправлять stderr в stdout
5. перенаправлять stderr и stdout в файл
6. перенаправлять stderr и stdout в stdout
7. перенаправлять stderr и stdout в stderr

1 означает stdout, а 2 - stderr.

Небольшое примечание для более глубокого понимания: с помощью команды less Вы можете просмотреть как stdout, который остаётся в буфере, так и stderr, который печатается на экране. Однако он стирается, когда Вы предпринимаете попытки "просмотреть" буфер.

3.2 Пример: stdout в файл

Это действие записывает стандартный вывод программы в файл.

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

Здесь создаётся файл с именем 'ls-l.txt'. В нём будет содержаться всё то, что Вы бы увидели, если бы

просто выполнили команду 'ls -l'.

3.3 Пример: stderr в файл

Это действие записывает стандартный поток ошибок программы в файл.

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

Здесь создаётся файл, названный 'grep-errors.txt'. В нём будет содержаться часть вывода команды 'grep da *', относящаяся к стандартному потоку ошибок.

3.4 Пример: stdout в stderr

Это действие записывает стандартный вывод программы в тот же файл, что и стандартный поток ошибок.

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

Здесь стандартный вывод команды отправляется в стандартный поток ошибок. Вы можете увидеть это разными способами.

3.5 Sample: stderr 2 stdout

Это действие записывает стандартный поток ошибок программы туда же, куда и стандартный вывод.

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

Здесь стандартный поток ошибок команды отправляется на стандартный вывод; если Вы перешлёте результат через конвейер (|) в less, то увидите, что строки, которые обычно пропадают (как записанные в стандартный поток ошибок), в этом случае сохраняются (так как они находятся на стандартном выводе).

3.6 Пример: stderr и stdout в файл

Это действие помещает весь вывод программы в файл. Это является подходящим вариантом для заданий cron: если Вы хотите, чтобы команда выполнялась абсолютно незаметно.

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

Это (предположим, для cron) удаляет любой файл с названием 'core' в любом каталоге. Помните, что Вам следует полностью быть уверенным в том, что выполняет команда, если возникает желание затереть её вывод.

4. Конвейеры

Данный раздел объясняет достаточно простым и практичным способом, каким образом следует использовать конвейеры и для чего Вам это может потребоваться.

4.1 Что это такое и зачем Вам это использовать?

Конвейеры предоставляют Вам возможность использовать (автор убеждён, что это достаточно просто) вывод одной программы в качестве входа другой.

4.2 Пример: простой конвейер с sed

Это очень простой способ использования конвейеров.

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

Здесь происходит следующее: первоначально выполняется команда ls -l, и её вывод, вместо отображения на экране, отправляется в программу sed, которая, в свою очередь, выводит на экран то, что должна.

4.3 Пример: альтернатива для ls -l *.txt

Возможно, это значительно более сложный способ, чем ls -l *.txt, но он приводится здесь только для того, чтобы проиллюстрировать работу с конвейерами, а не для решения вопроса выбора из этих двух способов листинга.

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

Здесь вывод программы ls -l отправляется в программу grep, которая, в свою очередь, выводит на экран строки, соответствующие регулярному выражению "\.txt$".

5. Переменные

Вы можете использовать переменные таким же образом, что и в любом языке программирования. Типы данных отсутствуют. Переменная в bash может представлять собой число, символ или строку символов.

Вам не следует объявлять переменную. В действительности, присвоение значения на её указатель уже создаёт её.

5.1 Пример: "Hello World!", использующий переменные

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

Вторая строка создаёт переменную, которая называется STR, и присваивает ей строчное значение "Hello World!". Затем ЗНАЧЕНИЕ этой переменной извлекается добавлением в начале знака '$'. Пожалуйста, запомните (постарайтесь), что если Вы не используете знак '$', вывод программы может быть другим. Вероятно, не таким, который Вам требуется.

5.2 Пример: очень простой скрипт резервного копирования (более эффективный)

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

Данный скрипт вводит ещё одно понятие. Прежде всего, Вам следует разобраться со второй строкой. Обратите внимание на выражение '$(date +%Y%m%d)'. Если Вы запустите этот скрипт, то заметите, что он выполняет команду внутри скобок, перехватывая её вывод.

Обратите внимание, что в этом скрипте имя выходного файла будет ежедневно изменяться, исходя из

формата ключа к команде date (+%Y%m%d). Вы можете поменять это заданием другого формата.

Другие примеры:

echo ls

echo $(ls)

5.3 Локальные переменные

Локальные переменные могут быть созданы при использовании ключевого слова *local*.

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

Данного примера должно быть достаточно для отображения способов использования локальных переменных.

6. Условные операторы

Условные операторы предоставляют Вам возможность решить, выполнять действие или нет; решение принимается при вычислении значения выражения.

6.1 Просто теория

Существует большое количество форм условных операторов. Элементарная форма - это **if** *выражение* **then** *оператор*, где 'оператор' выполняется только в том случае, если 'выражение' имеет значение "истина". '2<1' - это выражение, имеющее значение "ложь", в то время как '2>1' - "истина".

Существуют другие формы условных операторов, такие как: **if** *выражение* **then** *оператор1* **else** *оператор2*. Здесь 'оператор1' выполняется, если 'выражение'- истина; в противном случае, выполняется 'оператор2'.

Ещё одной формой условных операторов является: **if** *выражение1* **then** *оператор1* **else if** *выражение2* **then** *оператор2* **else** *оператор3*. В данной форме добавляется только последовательность "ELSE IF 'выражение2' THEN 'оператор2'", заставляющая 'оператор2' выполняться, если 'выражение2' имеет значение "истина". Всё остальное соответствует Вашему представлению об этом (см. предыдущие формы).

Несколько слов о синтаксисе:

Элементарная конструкция оператора 'if' в bash выглядит следующим образом:

if [выражение];

then

code if 'выражение' is true.

fi

6.2 Пример: элементарный образец условного оператора if .. then

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo-выражение вычислилось как истина
fi
```

Если выражением внутри квадратных скобок является истина, то выполняемый код находится после слова 'then' и перед словом 'fi', которое обозначает конец исполняемого при выполнении условия кода.

6.3 Пример: элементарный пример условного оператора if .. then ... else

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo-выражение вычислилось как истина
else
   echo-выражение вычислилось как ложь
fi
```

6.4 Пример: условные операторы с переменными

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
    echo-выражение вычислилось как истина
else
    echo-выражение вычислилось как ложь
fi
```

7. Циклы for, while и until

В этом разделе Вы познакомитесь с циклами for, while и until.

Цикл **for** немного отличается от аналогов в других языках программирования. Прежде всего, он предоставляет Вам возможность выполнять последовательные действия над "словами" в строке.

Цикл **while** выполняет кусок кода, если тестируемым выражением является истина; и останавливается при условии, если им является ложь (или внутри исполняемого кода встречается явно заданное прерывание цикла).

Цикл **until** практически идентичен циклу while. Отличие заключается только в том, что код выполняется при условии, если проверяемым выражением является ложь.

Если Вы предполагаете, что while и until очень похожи, Вы правы.

7.1 Пример цикла for

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

Во второй строке мы представляем i в качестве переменной, которая получает различные значения, содержащиеся в $(ls).

При необходимости третья строка могла бы быть длиннее; или там могло бы находиться несколько строк перед done (4-я строка).

'done' (4-я строка) показывает, что код, в котором используется значение $i, заканчивается и $i получает новое значение.

Данный скрипт не предполагает большой важности. Более полезным применением цикла for было бы использование его для отбора только каких-то определённых файлов в предыдущем примере.

7.2 C-подобный for

fiesh предложил добавить эту форму цикла. Это цикл for, наиболее похожий на for в языках C, Perl и т.п.

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

7.3 Пример цикла while:

```
#!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

Данный скрипт "эмулирует" широко известную (в языках C, Pascal, perl и т.д.) структуру 'for'.

7.4 Пример цикла until:

```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

8. Функции

Аналогично любому другому языку программирования, Вы можете использовать функции для группировки кусков кода более логичным способом, а также для практического применения волшебного искусства рекурсии.

Объявление функции - это только лишь запись function my_func { my_code }.

Вызов функции осуществляется аналогичным образом, что и вызов других программ. Вы просто пишете её имя.

8.1 Пример функций

```
#!/bin/bash
function quit {
    exit
}
function hello {
    echo Hello!
}
```

```
hello
quit
echo foo
```

В строках 2-4 содержится функция 'quit'. В строках 5-7 - функция 'hello'. Если Вам недостаточно понятен процесс, выполняемый данным скриптом, испытайте его!

Следует заметить, что совсем необязательно объявлять функции в каком-то определённом порядке.

Если Вы запустите скрипт, обратите внимание, что сначала вызывается функция 'hello', а затем функция 'quit'. Что касается программы, она никогда не достигает 10-й строки.

8.2 Пример функций с параметрами

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

Данный скрипт практически идентичен предыдущему. Главное отличие - это функция 'e'. Она выводит самый первый получаемый аргумент. Аргументы в функциях обрабатываются таким же образом, что и аргументы, переданные скрипту.

9. Интерфейсы пользователя

9.1 Использование select для создания простых меню

```
#!/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
```

Если Вы запустите этот скрипт, то увидите, что он является мечтой программиста о меню на текстовой основе. Вероятно, Вы заметите, что это очень похоже на конструкцию 'for', только вместо циклической обработки каждого "слова" в $OPTIONS программа опрашивает пользователя.

9.2 Использование командной строки

```
#!/bin/bash
if [ -z "$1" ]; then
    echo используйте: $0 каталог
    exit
fi
SRCD=$1                       #SRCD - SouRCe Directory - исходный каталог
TGTD="/var/backups/"          #TGTD - TarGeT Directory - конечный каталог
OF=home-$(date +%Y%m%d).tgz   #OF - Output File - выходной файл
tar -cZf $TGTD$OF $SRCD
```

Вам должно быть понятно, что выполняет этот скрипт. Выражение в первом условном операторе проверяет, получила ли программа аргумент ($1). Если - нет, оно завершает работу программы, предоставляя пользователю небольшое сообщение об ошибке. Оставшаяся на данный момент часть скрипта, очевидно, является понятной.

10. Разное

10.1 Чтение пользовательского ввода с помощью read

В некоторых случаях, возможно, возникнет необходимость попросить пользователя что-нибудь ввести. Существуют различные способы выполнения этого. Одним из способов является следующий:

```
#!/bin/bash
echo Введите, пожалуйста, Ваше имя
read NAME
echo "Привет, $NAME!"
```

В качестве варианта Вы можете получать сразу несколько значений с помощью read. Следующий пример поясняет это:

```
#!/bin/bash
echo "Введите, пожалуйста, Ваше имя и фамилию"
read FN LN   #FN - First Name - имя; LN - Last Name - фамилия
echo "Hi! $LN, $FN !"
```

10.2 Арифметические вычисления

В командной строке (или оболочке) попробуйте ввести следующее:

echo 1 + 1

Если Вы рассчитываете увидеть '2', то будете разочарованы. Что следует выполнить, если возникает необходимость, чтобы BASH произвёл вычисления над Вашими числами? Решение заключается в следующем:

echo $((1+1))

В результате этого вывод будет более "логичным". Такая запись используется для вычисления арифметических выражений. Вы также можете выполнить это следующим образом:

echo $[1+1]

Если Вам необходимо использовать дроби или более сложную математику, то можно использовать bc для вычисления арифметических выражений.

Когда автор запустил "echo $[3/4]" в командной оболочке, она вернула значение 0. Это связано с тем, что если bash отвечает, он использует только целые значения. Если Вы запустите "echo 3/4|bc -l", оболочка вернёт правильное значение 0.75.

10.3 Поиск bash

Из сообщения от mike (смотрите раздел "Благодарность"):

Вы всегда используете #!/bin/bash .. Вы могли бы привести пример, каким образом можно обнаружить, где расположен bash.

Предпочтительнее использовать 'locate bash', но locate имеется не на всех машинах.

'find ./ -name bash' из корневого каталога обычно срабатывает.

Можно проверить следующие расположения:

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

(автор не способен сразу придумать какой-либо другой каталог... Он находил bash в большинстве этих мест на различных системах).

Вы также можете попробовать 'which bash'.

10.4 Получение возвратного значения программы

В bash возвратное значение программы сохраняется в специальной переменной $?.

Данный пример иллюстрирует, как перехватить возвратное значение программы; автор предположил, что каталог *dada* не существует (это также предложил mike).

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

10.5 Перехват вывода команды

Этот небольшой скрипт представляет все таблицы из всех баз данных (предполагается, что у Вас установлен MySQL). Кроме того, следует подумать о способах преобразования команды 'mysql' для использования подходящего имени пользователя и пароля.

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

10.6 Несколько исходных файлов

Вы можете запускать несколько файлов с помощью команды source.

__TO-DO__

11. Таблицы

11.1 Операторы сравнения строк

(1) s1 = s2

(2) s1 != s2

(3) s1 < s2

(4) s1 > s2

(5) -n s1

(6) -z s1

(1) s1 совпадает с s2

(2) s1 не совпадает с s2

(3) s1 в алфавитном порядке предшествует s2 (в соответствии с текущей локалью)

(4) s1 в алфавитном порядке следует после s2 (в соответствии с текущей локалью)

(5) s1 имеет ненулевое значение (содержит один символ или более)

(6) s1 имеет нулевое значение

11.2 Примеры сравнения строк

Сравнение двух строк.

```
#!/bin/bash
S1='string'
S2='String'
if [ $S1=$S2 ];
then
        echo "S1('$S1') не равна to S2('$S2')"
fi
if [ $S1=$S1 ];
then
        echo "S1('$S1') равна to S1('$S1')"
fi
```

На данный момент, автор считает необходимым процитировать замечание из письма, полученного от Андреаса Бека, которое связано с использованием *if [$1 = $2]*.

Это не является хорошей идеей, так как если либо $S1, либо $S2 - пустая строка, Вы получите синтаксическую ошибку. Более приемлимым будет использование x$1 = x$2 или "$1" = "$2" .

11.3 Arithmetic operators

+

-

*

/

% (remainder)

11.4 Арифметические операторы сравнения

-lt (<)

-gt (>)

-le (<=)

-ge (>=)

-eq (==)

-ne (!=)

Программистам на C необходимо просто выбрать оператор, соответствующий выбранному оператору в скобках.

11.5 Полезные команды

Этот раздел переписал Kees (смотрите раздел "Благодарность").

Некоторые из этих команд практически содержат полноценные командные языки. Здесь объясняются только основы таких команд. Для более подробной информации внимательно просмотрите man-страницы каждой команды.

sed (потоковый редактор)

Sed - это неинтерактивный редактор. Вместо того, чтобы изменять файл движением курсора на экране, следует использовать сценарий инструкций по редактированию для sed, а также имя редактируемого файла. Вы также можете рассматривать sed в качестве фильтра. Посмотрите на некоторые примеры:

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

Sed заменяет строку 'to_be_replaced' строкой 'replaced', читая файл /tmp/dummy . Результат отправляется на стандартный вывод (обычно, на консоль), но Вы также можете добавить '> capture' в вышеуказанную строку, чтобы sed отправлял вывод в файл 'capture'.

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

Sed отображает все строки, за исключением строк с 12 по 18. Исходный файл этой командой не изменяется.

awk (манипулирование файлами данных, выборка и обработка текста)

Существует большое количество реализаций языка программирования AWK (наиболее распространенными интерпретаторами являются gawk из проекта GNU и "новый awk" mawk.) Принцип достаточно прост: AWK находится в поиске шаблона; для каждого подходящего шаблона выполняется какое-нибудь действие.

Автор повторно создал файл dummy, содержащий следующие строки:

"test123

test

tteesstt"

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

test123

test

Шаблон, искомый AWK, это 'test', а действие, выполняемое AWK при обнаружении строки в /tmp/dummy с подстрокой 'test', это 'print'.

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

3

Если Вы находитесь в поиске нескольких шаблонов, замените текст между кавычками на '-f file.awk'. В этом случае, Вы можете записать все шаблоны и действия в файле 'file.awk'.

grep (выводит строки, соответствующие искомому шаблону)

Мы рассматривали несколько команд grep в предыдущих главах, которые отображали строки, соответствующие шаблону. Однако grep способен выполнять значительно большее.

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

12

Строка "look for this" была обнаружена 12 раз в файле /var/log/messages.

[ok, данный пример был фикцией, /var/log/messages был переделан :-)]

wc (считает строки, слова и байты)

В следующем примере можно заметить, что выводится не то, что мы ожидаем. В этом случае, файл dummy содержит следующий текст:

"bash introduction

howto test file"

```
$wc --words --lines --bytes /tmp/dummy
```

2 5 34 /tmp/dummy

wc не заботится о порядке параметров. Он всегда выводит их в стандартном порядке: <число строк><число слов><число байтов><имя файла>.

sort (сортирует строки текстового файла)

В этом случае, файл dummy содержит следующий текст:

"b

c

a"

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

Вывод выглядит следующим образом:

a

b

c

Команды не должны быть такими простыми :-)

bc (вычислительный язык программирования)

bc производит вычисления с командной строки (ввод из файла, но не через перенаправление или конвейер), а также из пользовательского интерфейса. Следующий пример показывает некоторые команды. Обратите внимание, что автор использовал bc с параметром -q, чтобы отказаться от вывода сообщения с приглашением.

```
$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 (инициализирует терминал или запрашивает базу данных terminfo)

Небольшая иллюстрация возможностей tput:

```
$tput cup 10 4
```

Приглашение командной строки появится в координатах (y10,x4).

```
$tput reset
```

Экран очищается и приглашение появляется в (y1,x1). Обратите внимание, что (y0,x0) - это левый верхний угол.

```
$tput cols
```

80

Отображает возможное количество символов в направлении по оси x.

Настоятельно рекомендуется быть с этими программами на "ты" (как минимум). Существует огромное количество небольших программ, которые предоставляют Вам возможность заняться настоящей магией в командной строке.

[Некоторые примеры были заимствованы из man-страниц или FAQ.]

12. Ещё скрипты

12.1 Применение команды ко всем файлам в каталоге.

12.2 Пример: очень простой скрипт резервного копирования (более эффективный)

```
#!/bin/bash
SRCD="/home/"                 #SRCD - SouRCe Directory - исходный каталог
TGTD="/var/backups/"          #TGTD - TarGeT Directory - конечный каталог
OF=home-$(date +%Y%m%d).tgz  #OF - Output File - выходной файл
tar -cZf $TGTD$OF $SRCD
```

12.3 Программа переименования файлов

```
#!/bin/sh
# renna: переименование нескольких файлов по специальным правилам
# Автор - felix hudson Jan - 2000

#Прежде всего, посмотрите на различные "режимы", которые имеются у этой программы.
#Если первый аргумент ($1) является подходящим, мы выполняем эту часть
#программы и выходим.

# Проверка на возможность добавления префикса.
if [ $1 = p ]; then

#Теперь переходим от переменной режима ($1) и префикса ($2)
prefix=$2 ; shift ; shift

# Необходимо проверить, задан ли, по крайней мере, хотя бы один файл.
# В противном случае, лучше ничего не предпринимать, чем переименовывать несуществующие
# файлы!!

  if [$1 = ]; then
     echo "не задано ни одного файла"
     exit 0
  fi

# Этот цикл for обрабатывает все файлы, которые мы задали
# программе.
# Он осуществляет одно переименование на файл.
  for file in $*
    do
    mv ${file} $prefix$file
  done

#После этого выполняется выход из программы.
  exit 0
fi
```

```
# Проверка на условие добавления суффикса.
# В остальном, данная часть фактически идентична предыдущему разделу;
# пожалуйста, смотрите комментарии, содержащиеся в нем.
if [ $1 = s ]; then
  suffix=$2 ; shift ; shift
  if [$1 = ]; then
   echo "не задано ни одного файла"
   exit 0
  fi

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

 exit 0
fi

# Проверка на условие переименования с заменой.
if [ $1 = r ]; then

  shift

# Из соображений безопасности автор включил эту часть, чтобы не повредить ни один файл, если пользователь
# не определил, что следует выполнить:

  if [ $# -lt 3 ] ; then
    echo "Ошибка; правильный ввод: renna r [выражение] [замена] файлы... "
    exit 0
  fi

# Рассмотрим другую информацию
  OLD=$1 ; NEW=$2 ; shift ; shift

# Данный цикл for последовательно проходит через все файлы, которые мы
# задали программе.
# Он совершает одно переименование на файл, используя программу 'sed'.
# Это простая программа с командной строки, которая анализирует стандартный
# ввод и заменяет регулярное выражение на заданную строку.
# Здесь мы задаём для sed имя файла (в качестве стандартного ввода) и заменяем
# необходимый текст.

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

# Если мы достигли этой строки, это означает, что программе были заданы
# неправильные параметры. В связи с этим, следует объяснить пользователю, как её
# использовать
echo "используйте:"
echo " renna p [префикс] файлы.."
echo " renna s [суффикс] файлы.."
echo " renna r [выражение] [замена] файлы.."
exit 0

# done!
```

12.4 Программа переименования файлов (простая)

```
#!/bin/bash
# renames.sh
# простая программа переименования

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
```

13. Если происходящее отличается от ожидаемого (отладка)

13.1 Каким образом можно вызвать BASH?

Было бы неплохо добавить в первую строку

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

В результате этого будет выводиться некоторая интересная выходная информация.

14. О документе

Не следует стесняться вносить исправления, дополнения или что-либо ещё, если, по Вашему мнению, это должно присутствовать в этом документе. Автор постарается по возможности обновить его.

14.1 Гарантии (отсутствуют)

Данный документ поставляется без каких-либо гарантий и т.д.

14.2 Переводы

Итальянский: Вильям Гельфи (William Ghelfi, wizzy at tiscalinet.it), http://web.tiscalinet.it/penguin_rules.

Французский: Лорент Мартелли (Laurent Martelli), is missed.

Корейский: Минсок Парк (Minseok Park), http://kldp.org.

Корейский: Чхун Хе Чин (Chun Hye Jin), unknown.

Испанский: Габриэль Родригес Альберич (Gabriel Rodri'guez Alberich), http://www.insflug.org.

Нидерландский: Эллен Бокхорст (Ellen Bokhorst), http://nl.linux.org/.

Словенский: Андрей Лайовиц (Andrej Lajovic), http://www.lugos.si/.

По мнению автора, существуют другие переводы. Однако у него отсутствует какая-либо касающиеся их информация; если у Вас она имеется, отправьте её автору и он обновит этот раздел.

14.3 Благодарность

- Автор выражает признательность переводчикам этого документа на другие языки (предыдущий раздел):
- Нэйтану Херсту (Nathan Hurst), который предоставил большое количество исправлений;
- Джону Абботту (Jon Abbott), который прислал комментарии по поводу вычислений арифметических выражений;
- Феликсу Хадсону (Felix Hudson) за написание скрипта *renna*;
- Кесу ван ден Бруку (Kees van den Broek) (который предоставил большое количество исправлений и переписал полезный раздел по командам);
- Mike (pink), предоставившему ряд рекомендаций относительно местоположения bash и тестирования файлов;
- Fiesh, давшему неплохой совет для раздела по циклам;
- Lion, который предложил упомянуть о распространённой ошибке (./hello.sh: Command not found.);
- Андреасу Беку (Andreas Beck), который внес несколько исправлений и комментариев.

14.4 История

Добавлена информация о новых переводах и немного исправлений.

Добавлен переписанный Kess'ом раздел по полезным командам.

Внесен ряд исправлений и дополнений.

Добавлены примеры по сравнению строк.

Версия 0.8. Отказ от нумерации версий. По мнению автора, достаточно указания одной даты.

Версия 0.7. Внесены исправления и написаны некоторые прежние разделы TO-DO.

Версия 0.6. Внесены небольшие исправления.

Версия 0.5. Добавлен раздел по перенаправлению.

Версия 0.4. Документ изменил свое расположение (из-за эксбосса автора) и, в данный момент, находится на предназначенном для него сайте: www.linuxdoc.org.

Предыдущее: автор не помнит, он не использовал ни rcs, ни cvs :(

14.5 Другие источники

Введение в bash (под BE): http://org.laol.net/lamug/beforever/bashtut.htm.

Программирование на Bourne Shell: http://207.213.123.70/book/.

14.6 Примечание переводчика

Перевод распространяется на условиях лицензии GPL2.

Текущая версия перевода находится на Линукс-странице переводчика:

http://daslinux.da.ru/.

Высказывайте свои комментарии, замечания и предложения в гостевой книге на странице или по адресу: das@cabel.net.