

## Тема 4. Сценарии командного процессора

### Введение

Сценарий (script) командного процессора (shell) может быть определен как группа команд выполняемых в последовательности. Давайте начнем с описания шагов необходимых для написания и исполнения сценария командного процессора:

**Шаг 1:** Откройте файл используя текстовый редактор (например "gedit" или "nano").

```
nano firstshellscript.sh
```

**Шаг 2:** Все сценарии должны начинаться со строки "#!/bin/bash" или с указания любого другого командного процессора, который вы предпочитаете. Эта строка называется *shebang*, и хотя она выглядит как комментарий, это не так: она уведомляет оболочку о том, какой командный процессор должен использоваться для этого сценария. Указанный путь должен быть абсолютным (вы не можете просто написать "bash", к примеру), а *shebang* должен находиться на первой строке скрипта без символов перед ним.

**Шаг 3:** Напишите код, который вы хотите выполнить. Наш первый сценарий будет обычной программой "Hello World", которую мы разместим в файле названном 'firstshellscript.sh'.

```
#!/bin/sh  
echo "Hello World"
```

**Шаг 4:** Следующий шаг - сделать сценарий исполняемым, используя команду "chmod".

```
chmod +x firstshellscript.sh
```

**Шаг 5:** Выполнить сценарий. Это может быть сделано вводом имени сценария в командную строку с указанием его пути. Если сценарий находится в текущей директории, это очень просто:

```
bash$ ./firstshellscript.sh  
Hello World
```

Если вы хотите увидеть выполнение пошагово, что очень полезно для отладки - тогда выполните сценарий с опцией '-x' (что означает 'раскрыть аргументы'):

```
sh -x firstshellscript.sh  
+ echo 'Hello World'  
Hello World
```

Чтобы увидеть содержание сценария вы можете использовать команду 'cat' или просто открыть сценарий в любом текстовом редакторе:

```
bash$ cat firstshellscript.sh  
#!/bin/sh  
echo "Hello World"
```

### Комментарии в сценарии

В сценариях командного процессора все строки, начинающиеся с #, являются комментариями.

```
# Это строка комментария.  
# Это еще одна строка комментария.
```

Вы также можете написать многострочные комментарии, используя двоеточие и одинарные кавычки:

```
: 'Это комментарий.  
Это снова комментарий.  
Кто бы мог подумать, что это еще один комментарий.'
```

Примечание: Это не будет работать, если в комментариях будут символы одинарной кавычки.

## Переменные

Как вы знаете (или не знаете), переменные являются самой значительной частью любого языка программирования, будь это Perl, C, или сценарии. В сценариях переменные разделяются на системные и пользовательские.

### Системные переменные

Системные переменные определяются и хранятся в окружении *родительского командного процессора* (командного процессора, из которого ваш сценарий выполняется). Они также называются переменными окружения. Имена этих переменных состоят из заглавных букв, и могут быть показаны командой 'set'. Примерами системных переменных являются PWD, HOME, USER. Значения этих системных переменных могут быть показаны по отдельности командой 'echo'. Например 'echo \$HOME' выведет значение, хранящееся в системной переменной HOME.

Когда устанавливаете системную переменную, не забудьте использовать команду 'export', чтобы сделать ее доступной *дочерним командным процессорам* (любые командные процессоры, которые запущены из текущего, включая сценарии):

```
bash$ SCRIPT_PATH=/home/blessen/shellscript  
bash$ export SCRIPT_PATH
```

Современные командные процессоры также позволяют сделать это одной командой:

```
bash$ export SCRIPT_PATH=/home/blessen/shellscript
```

### Пользовательские переменные

Эти переменные часто используются в сценариях, когда нет необходимости или желания делать переменные доступными другим программам. Их имена не могут начинаться с цифр, записываются в нижнем регистре и вместо пробела используется знак подчеркивания - например "define\_tempval".

Когда мы присваиваем значение переменной, мы пишем имя переменной перед знаком равенства '=', за которым немедленно идет значение, например 'define\_tempval=blessen' (заметьте, не должно быть пробелов перед или за знаком равенства.). А чтобы использовать или показать значение "define\_tempval", мы должны использовать команду 'echo' со знаком '\$' перед именем переменной, например:

```
bash$ echo $define_tempval  
blessen
```

Следующий сценарий устанавливает переменную "username" и показывает ее значение.

```
#!/bin/sh
username=blessen
echo "username - $username"
```

### Аргументы командной строки

Это переменные, которые содержат аргументы выполняемого сценария. Доступ к этим переменным может быть получен через имена \$1, \$2, ... \$n, где \$1 первый аргумент командной строки, \$2 второй, и так далее. Аргументы располагаются после имени сценария и разделены пробелами. Переменная \$0 - это имя сценария. Переменная \$# хранит количество аргументов командной строки, это количество ограничено 9 аргументами в старых шеллах и практически неограниченно в современных.

Рассмотрим сценарий, который возьмет два аргумента командной строки и покажет их. Мы назовем его 'commandline.sh':

```
#!/bin/sh
echo "Первый аргумент - $1"
echo "Второй аргумент - $2"
```

Если запустить 'commandline.sh' с аргументами командной строки "blessen" и "lijoe" результат будет таким:

```
bash$ ./commandline.sh blessen lijoe
Первый аргумент - blessen
Второй аргумент - lijoe
```

### Переменная кода возврата

Эта переменная сообщает нам, была ли последняя команда успешно выполнена. Она обозначается \$? . Нулевое значение означает, что команда была успешно выполнена. Любые другие числа означают, что команда была выполнена с ошибкой (также некоторые программы, такие как 'mail', используют ненулевое значение возврата для отображения состояния, а не ошибки). Таким образом, это очень полезное свойство при написании сценариев.

Чтобы проверить работу переменной кода возврата, создайте файл с именем "test", выполнив команду `touch test` . Затем покажите содержание файла:

```
bash$ cat test
```

Проверьте значение \$? .

```
bash$ echo $?
0
```

Значение равно нулю, потому что команда была успешно выполнена. Теперь попробуйте команду 'cat' с несуществующим файлом:

```
bash$ cat xyz1
bash$ echo $?
1
```

Значение 1 показывает, что предыдущая команда была выполнена с ошибкой.

## Область видимости переменной

При написании сценариев командного процессора видимость переменных используется для различных задач. Несмотря на то, что использование видимости редко бывает необходимо, это может быть полезным инструментом. В командных процессорах есть два типа видимости: глобальная и локальная. Локальные переменные определяются, используя ключевое слово "local" перед именем переменной, все остальные переменные, кроме связанных с аргументами функции, - глобальные, и поэтому доступны из любого места сценария. Сценарий, приведенный ниже демонстрирует различные видимости локальной и глобальной переменной:

```
#!/bin/sh
display()
{
    local local_var=100
    global_var=blessen
    echo "локальная переменная внутри функции равна $local_var"
    echo "глобальная переменная внутри функции равна $global_var"
}
echo "====внутри функции===="
display
echo "=====вне функции=====
echo "локальная переменная вне функции равна $local_var"
echo "глобальная переменная вне функции равна $global_var"
```

Запущенный сценарий выводит следующий результат:

```
====внутри функции====
локальная переменная внутри функции равна 100
глобальная переменная внутри функции равна blessen
=====вне функции=====
локальная переменная вне функции равна
глобальная переменная вне функции равна blessen
```

Заметьте отсутствие значения для локальной переменной вне функции.

## Ввод-вывод в сценариях

Для ввода с клавиатуры используется команда 'read'. Эта команда считывает значения, набранные на клавиатуре, и присваивает каждое определенной переменной.

```
read <имя_переменной>
```

Для вывода используется команда 'echo'.

```
echo <имя_переменной>
```

## Арифметические операции

Как и другие сценарные языки, сценарии командного процессора также позволяют нам использовать арифметические операции, такие как сложение, вычитание, умножение, и деление. Для их использования используется функция "expr"; например, "expr a + b" означает 'сложить a и b'.

например:

```
sum=`expr 12 + 20`
```

Подобный синтаксис может быть использован для вычитания, умножения, и деления. Есть и другой путь совершать арифметические операции - заключить

переменные и оператор в квадратные скобки выражения начинающегося на знак '\$'. Синтаксис такой:

```
$[арифметические операции]
```

например:

```
echo $[12 + 10]
```

Заметьте, что подобное написание не универсально. К примеру, оно не будет работать в Korn shell. Написание '\$((...))' более независимо от командного процессора, еще лучше следовать принципу "позволь командному процессору делать то, что он умеет делать лучше всех и оставь остальное стандартным программам", использовать программы для вычислений такие как 'bc' или 'dc' и команды замещения. Также учтите, что арифметические операции командного процессора только **целочисленные**, а два вышеуказанных метода не имеют такой проблемы.

## Оператор условия "if"

Давайте развлечемся с условным выражением "if" (если). Большую часть времени программисты сталкиваются с ситуациями, когда должны сравниваться две переменные, и затем исполняются целые блоки команд в зависимости от верности или ложности условия. Поэтому в подобных случаях мы должны использовать выражение "if". Синтаксис показан ниже:

```
if [ <условие> ]
then
  <любые выражения или операторы>
fi
```

Следующий сценарий запросит имя пользователя, и если было введено "blessen", отобразит сообщение, показывающее, что вы успешно зашли. Иначе покажет сообщение о неправильном имени пользователя.

```
#!/bin/sh
echo "Введите имя пользователя:"
read username
if [ "$username" = "blessen" ]
then
  echo 'Успешно!!! Вы зашли.'
else
  echo 'Извините, неправильное имя пользователя.'
fi
```

***Не забывайте всегда закрывать переменную в условии в двойные кавычки; если этого не сделать ваш сценарий вызовет ошибку синтаксиса, когда переменная будет пуста. Также квадратные скобки должны иметь пробел за открывающей скобкой и пробел перед закрывающей.***

## Сравнение переменных

В сценариях командного процессора можно выполнять различные сравнения. Если значения сравниваемых переменных являются числами, вы должны использовать следующие опции:

-eq Равно

-ne Не равно

- lt Меньше
- le Меньше или равно
- gt Больше
- ge Больше или равно

При работе со строками:

- z string Верно, если строка пуста.
- n string Верно, если строка не пуста.
- string1 = string2 Верно, если строка string1 равна строке string2.
- string1 != string2 Верно, если строка string1 не равна строке string2.
- < Первая строка отсортирована перед второй
- > Первая строка отсортирована после второй

При работе с файлами:

- d file Верно, если файл является директорией.
- e file Верно, если файл существует.
- f file Верно, если файл существует и является обычным файлом.
- L file Верно, если файл является символической ссылкой.
- r file Верно, если файл можно читать.
- w file Верно, если в файл можно писать.
- x file Верно, если файл можно исполнять..
- file1 -nt file2 Верно, если file1 новее file2.
- file1 -ot file2 Верно, если file1 старше file2.

## Циклы

### Цикл "for"

Самый часто используемый цикл - цикл "for". В сценариях существует два типа: первый аналогичен циклу "for" в языке программирования Си, а второй является циклом итерации (обработки списков).

Синтаксис первого типа цикла "for" (этот тип доступен только в современных командных процессорах):

```
for (( <начальное значение>; <условие>; <инкремент/декремент> ))
do
    <любые выражения или операторы>
done
```

Пример:

```
#!/bin/sh
for (( i=1; $i <= 10; i++ ))
do
    echo $i
done
```

Выведет список чисел от 1 до 10. Синтаксис второго, более распространенного типа цикла "for":

```
for <переменная> in <список>
do
    <любые выражения или операторы>
done
```

Этот сценарий прочтет содержимое '/etc/group' и покажет каждую строку:

```
#!/bin/sh
count=0
for i in `cat /etc/group`
do
    count=`expr "$count" + 1`
    echo "Строка номер $count :"
    echo $i
done
echo "Конец файла"
```

Другой пример цикла "for" использует оператор "seq" чтобы сформировать последовательность:

```
#!/bin/sh
for i in `seq 1 5`
do
    echo $i
done
```

## Цикл "while"

Цикл "while" - еще один полезный цикл, используемый во всех языках программирования. Цикл продолжает выполняться до тех пор, пока условие перестанет быть верным.

```
while [ <условие> ]
do
    <любые выражения или операторы>
done
```

Следующий сценарий присваивает значение "1" переменной "num" и прибавляет единицу к "num" каждый раз, когда "num" меньше 5.

```
#!/bin/sh
num=1
while [ $num -lt 5 ]
do
    num=$((num + 1));
    echo $num;
done
```

## Операторы "select" и "case"

Подобно конструкции "switch/case" в языке программирования Си комбинация операторов "select" и "case" обеспечивает сценариям такую же функциональность. Оператор "select" не является частью конструкции "case", но оба приведены для иллюстрации того, как они могут быть использованы в сценариях.

Синтаксис оператора select:

```
select <переменная> in <список>
```

```
do
  <любые выражения или операторы>
done
```

### Синтаксис оператора case:

```
case $<переменная> in
  <выбор1>) <любые выражения или операторы> ;;
  <выбор2>) <любые выражения или операторы> ;;
  *) echo "Извините, неправильный выбор" ;;
esac
```

Пример, приведенный ниже, покажет пример совместного использования операторов "select" и "case". Когда пользователь выберет действие, сценарий перезапустит соответствующий сервис.

```
#!/bin/bash
echo "*****"
select opt in apache named sendmail
do
  case $opt in
    apache) /etc/rc.d/init.d/httpd restart;;
    named) /etc/rc.d/init.d/named restart;;
    sendmail) /etc/rc.d/init.d/sendmail restart;;
    *) echo "Ничего не будет перезапущено"
  esac
  echo "*****"
  # Если здесь не поставить "break", тогда мы не выйдем в процессор после выбора.
  break
done
```

*Предпочтительнее вместо оператора 'break', который будет мешать если вы захотите выполнить больше чем одно из предоставленных действий, добавить действие 'Quit' как последнюю опцию в списке с соответствующим оператором.*

## Функции

В современном мире, где все программисты используют объектно-ориентированную модель программирования, даже программисты командных процессоров не остались позади. Можно разбить код на маленькие части - функции, и присвоить им имена в главном коде программы. Это помогает при отладке и повторном использовании кода.

### Синтаксис функции:

```
<имя_функции> ()
{ # начало функции
  <операторы>
} # конец функции
```

Функции вызываются, когда их имя встречается в коде программы, возможно с аргументами, располагающимися за именем функции. Например:

```
#!/bin/sh
sumcalc ()
{
  sum=$(( $1 + $2 ))
}
echo "Введите первое число:"
read num1
echo "Введите второе число:"
read num2
```



```
sumcalc $num1 $num2  
echo "Сумма чисел: $sum"
```

## **Отладка сценариев командных процессоров**

Теперь нам нужно отлаживать наши программы. Для этого мы используем опции '-x' и '-v' шелла. Опция '-v' выводит больше информации о ходе выполнения программы. Опция '-x' подробно распишет каждую простую команду, цикл "for", оператор "case", оператор "select" или арифметический оператор, показывая значение PS4, за каждой командой и ее аргументы или список слов. Попробуйте их - они могут быть очень полезны, когда вы не можете понять, где находится проблема в вашем сценарии.

**Задание**

— , .