



9. Shell scripting

Разделы:

- Зачем разрабатывать скрипты?
- Что такое bash?
- Первые скрипты
- Операторы bash
- Подстановка команды
- Константы
- Массивы
- Функции

Зачем разрабатывать скрипты?

- Знание языка командной оболочки является залогом успешного решения задач администрирования системы
- Даже если мы не собираемся заниматься написанием своих сценариев, должны иметь в виду, что во время загрузки Linux выполняется целый ряд сценариев из */etc/rc.d*
- Они настраивают конфигурацию ОС и запускают различные сервисы, поэтому очень важно четко понимать эти скрипты и иметь достаточно знаний, чтобы вносить в них какие-либо изменения

Зачем разрабатывать скрипты?

- Shell-скрипты очень хорошо подходят для быстрого создания прототипов сложных приложений, несмотря на ограниченный набор языковых конструкций и определенную «медлительность»
- Это позволяет детально проработать структуру будущего приложения, обнаружить возможные «ловушки» и лишь затем приступить к кодированию, например, с использованием C/C++ или Perl

Зачем разрабатывать скрипты?

- Для каких задач неприменимы скрипты:
 - для ресурсоемких задач;
 - для задач, связанных с выполнением математических вычислений;
 - для кросс-платформенного системного программирования;
 - для сложных приложений;
 - когда главное - безопасность системы;
 - для задач, выполняющих огромный объем работ с файлами;
 - для задач, работающих с многомерными массивами;
 - когда необходимо предоставить графический интерфейс с пользователем (GUI);
 - когда необходим прямой доступ к аппаратуре компьютера;
 - для проприетарных программ;
 - и др.

Что такое BASH?

- Название BASH – это аббревиатура от "Bourne-Again Shell" и игра слов от ставшего уже классикой "Bourne Shell" Стивена Бурна (*Stephen Bourne*)
- В последние годы BASH достиг такой популярности, что *de facto* стал стандартной командной оболочкой для многих UNIX-подобных систем
- Большинство принципов программирования на BASH одинаково хорошо применимы и к другим командным оболочкам - *Korn Shell* (*ksh* и его потомок *zsh*), от которой Bash позаимствовал некоторые особенности, и *C Shell* и его производных

Что такое BASH?

- Когда программа запускается, создается новый процесс, потому что оболочка создает полную свою копию
- У этого дочернего процесса то же самое окружение, что и у родителя, отличаются только идентификаторы процессов (PID)
- Это порождение процесса, или его **форк**
- После порождения процесса адресное пространство дочернего процесса заполняется новыми данными
- Это делается с помощью системного вызова *call*

Что такое BASH?

- Bash поддерживает 3 типа встроенных команд:
 1. встроенные команды языка *Bourne shell* (`:`, `.`, *break*, *cd*, *continue*, *eval*, *exec*, *exit*, *export*, *getopts*, *hash*, *pwd*, *readonly*, *return*, *set*, *shift*, *test*, `[`, *times*, *trap*, *umask* и *unset*);
 2. встроенные команды *Bash* (*alias*, *bind*, *builtin*, *command*, *declare*, *echo*, *enable*, *help*, *let*, *local*, *logout*, *printf*, *read*, *shopt*, *type*, *typeset*, *ulimit* и *unalias*);
 3. специальные встроенные команды POSIX-режима (`:`, `.`, *break*, *continue*, *eval*, *exec*, *exit*, *export*, *readonly*, *return*, *set*, *shift*, *trap* и *unset*).
 - Несмотря на похожие названия, их поведение отличается от типов 1 и 2



Что такое BASH?

- Схема *fork-and-exec* используется только после того, как оболочка проанализировала вход следующим способом:
 - Оболочка считает своим входом файл, строку или пользовательский терминал
 - Вход разбивается на слова и операторы с использованием правил цитирования
 - Оболочка разбирает лексемы на простые и составные команды
 - Bash выполняет расширение оболочки
 - Если необходимо, выполняется перенаправление
 - Выполняются команды
 - В некоторых случаях Bash ожидает окончания команды с сохранением ее кода завершения



Что такое BASH?

- Выполняются следующие подстановки:
 - Подстановки фигурными скобками
 - Подстановки тильдой
 - Подстановки параметра и переменной
 - Подстановки команды
 - Арифметические подстановки
 - Разбиение на слова
 - Подстановки в имени файла

Что такое BASH?

- Важной функцией оболочки Bash является следующий порядок поиска команд:
 - Проверяет, содержит ли команд символы косой черты
 - Если команда - не функция, то проверяется список встроенных команд
 - Если команда – это ни функция, ни встроенная команда, анализируются каталоги из переменной окружения PATH
 - Если поиск завершился неудачей, то Bash печатает сообщение об ошибке и возвращает код 127
 - Если же поиск успешен или команда содержит символы косой черты, то оболочка запускает команду на выполнение в отдельном рабочем окружении
 - Если выполнение происходит с ошибкой из-за того, что файл не является исполняемым или каталогом, то он трактуется как скрипт оболочки
 - Если команда не началась асинхронно, оболочка ожидает завершения команды и получения кода завершения



Первый скрипт

```
#!/bin/sh
```

```
# This is a comment!
```

```
echo Hello World
```

```
# This is a comment, too!
```

```
chmod u+x first.sh
```

```
./first.sh
```

```
#!/bin/sh
```

```
# This is a comment!
```

```
echo "Hello World"
```

```
# This is a comment, too!
```



Первый скрипт

```
#!/bin/sh
# This is a comment!
echo "Hello      World"           # This is a comment, too!
echo "Hello World"
echo "Hello * World"
echo Hello * World
echo Hello      World
echo "Hello" World
echo Hello "      " World
echo "Hello \\"*\" World"
echo `hello` world
echo 'hello' world
```



Первый скрипт

```
#!/bin/sh  
MY_MESSAGE="Hello World"  
echo $MY_MESSAGE
```

```
$ x="hello"  
$ y=`expr $x + 1`  
expr: non-numeric argument  
$
```



Первый скрипт

```
MY_MESSAGE="Hello World"
```

```
MY_SHORT_MESSAGE=hi
```

```
MY_NUMBER=1
```

```
MY_PI=3.142
```

```
MY_OTHER_PI="3.142"
```

```
MY_MIXED=123abc
```

```
#!/bin/sh
```

```
echo What is your name?
```

```
read MY_NAME
```

```
echo "Hello $MY_NAME - hope you're well."
```



Первый скрипт

```
#!/bin/sh  
echo "MYVAR is: $MYVAR"  
MYVAR="hi there"  
echo "MYVAR is: $MYVAR"
```

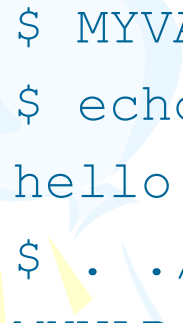
```
$ ./myvar2.sh  
MYVAR is:  
MYVAR is: hi there
```

```
$ MYVAR=hello  
$ ./myvar2.sh  
MYVAR is:  
MYVAR is: hi there
```




Первый скрипт

```
$ export MYVAR  
$ ./myvar2.sh  
MYVAR is: hello  
MYVAR is: hi there
```



```
$ MYVAR=hello  
$ echo $MYVAR  
hello  
$ . ./myvar2.sh  
MYVAR is: hello  
MYVAR is: hi there  
$ echo $MYVAR  
hi there
```





Первый скрипт

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called $USER_NAME_file"
touch $USER_NAME_file
```

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called ${USER_NAME}_file"
touch "${USER_NAME}_file"
```

Первый скрипт

- Здесь мы использовали кавычки ("\${USER_NAME}_file")
- Если пользователь введет "*Dima Medvedev*" (с пробелом в середине), забудет указать кавычки, то аргументы посылаемые команде *touch* будут *Dima* и *Medvedev_file*
- Вместо создания *Dima Medvedev_file*, будут созданы два файла
- Кавычки позволяют избежать этого

Оператор if

#неправильно!

```
if [$foo == "bar" ]
```

- **Студентам:** в чем здесь проблема?

#Покажем обязательные пробелы с помощью слова 'SPACE'

#в скриптах придется заменить 'SPACE' на реальные пробелы

```
if SPACE [ SPACE "$foo" SPACE == SPACE "bar"  
SPACE ]
```

Оператор if

```
if [ ... ]  
then  
    # if-code  
else  
    # else-code  
fi
```

```
if [ ... ]; then  
    # do something  
fi
```

```
if [ something ]; then  
    echo "Something"  
elif [ something_else ]; then  
    echo "Something else"  
else  
    echo "None of the above"  
fi
```

Оператор if

```
#!/bin/sh
if [ "$X" -lt "0" ]
then
    echo "X is less than zero"
fi
if [ "$X" -gt "0" ]; then
    echo "X is more than zero"
fi
[ "$X" -le "0" ] && \
    echo "X is less than or equal to zero"
[ "$X" -ge "0" ] && \
    echo "X is more than or equal to zero"
[ "$X" = "0" ] && \
    echo "X is the string or number \"0\""
[ "$X" = "hello" ] && \
    echo "X matches the string \"hello\""
[ "$X" != "hello" ] && \
    echo "X is not the string \"hello\""
[ -n "$X" ] && \
    echo "X is of nonzero length"
[ -f "$X" ] && \
    echo "X is the path of a real file" || \
    echo "No such file: $X"
[ -x "$X" ] && \
    echo "X is the path of an executable file"
[ "$X" -nt "/etc/passwd" ] && \
    echo "X is a file which is newer than /etc/passwd"
```

Оператор if

```
#!/bin/sh
[ $X -ne 0 ] && echo "X isn't zero" || echo "X is zero"
[ -f $X ] && echo "X is a file" || echo "X is not a file"
[ -n $X ] && echo "X is of non-zero length" || \
    echo "X is of zero length"
```

```
echo -en "Please guess the magic number: "
read X
echo $X | grep "[^0-9]" > /dev/null 2>&1
if [ "$?" -eq "0" ]; then
    # If the grep found something other than 0-9
    # then it's not an integer.
    echo "Sorry, wanted a number"
else
    # The grep found only 0-9, so it's an integer.
    # We can safely do a test on it.
    if [ "$X" == "7" ]; then
        echo "You entered the magic number!"
    fi
fi
```



Подстановка команды

- Выполняется, когда команда заключается в круглые скобки:

`$(command)`

- или используются обратные апострофы:

``command``

```
$ echo `date`
```

```
Thu Mar 21 10:06:20 KrasTime 2013
```

Циклы

- *for, while, until, select*

for NAME [in LIST]; do COMMANDS; done

```
#!/bin/sh
for i in 1 2 3 5 7
do
    echo $i
done

exit 0
```


ЦИКЛЫ

```
#!/bin/sh
for i in hello 2 * 3 5 7 goodbye and farewell
do
    echo $i
done
exit 0
```

Циклы

```
while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done
```

```
#!/bin/sh
```

```
INPUT_STRING=hello
```

```
while [ "$INPUT_STRING" != "bye" ]
```

```
do
```

```
    echo "Type something here [bye to quit]"
```

```
    read INPUT_STRING
```

```
    echo "You've typed $INPUT_STRING"
```

```
done
```

```
exit 0
```

Циклы

```
#!/bin/sh
INPUT_STRING=hello
while :
do
    echo "Type something here [Ctrl-C to quit]"
    read INPUT_STRING
    echo "You've typed $INPUT_STRING"
done
exit 0
```

ЦИКЛЫ

```
#!/bin/sh
```

```
while read f  
do
```

```
    case $f in
```

```
        hello)
```

```
        howdy)
```

```
        gday)
```

```
        bonjour)
```

```
        "guten tag")
```

```
        *)
```

```
    esac
```

```
done < myfile
```

```
exit 0
```

```
    echo "English" ;;
```

```
    echo "American" ;;
```

```
    echo "Australian" ;;
```

```
    echo "French" ;;
```

```
    echo "German" ;;
```

```
    echo Unknown language: $f
```

```
;;
```

Циклы

```
until TEST-COMMAND; do CONSEQUENT-COMMANDS; done
```

```
#!/bin/sh
```

```
INPUT_STRING=hello
```

```
until [ "$INPUT_STRING" = "bye" ]
```

```
do
```

```
    echo "Type something here [bye to quit]"
```

```
    read INPUT_STRING
```

```
    echo "You've typed $INPUT_STRING"
```

```
done
```

```
exit 0
```

- Сравните с кодом на слайде 26

Циклы

```
select WORD [in LIST]; do RESPECTIVE-COMMANDS; done
```

```
#!/bin/sh
```

```
echo "This script can make any of the files in this directory private."
```

```
echo "Enter the number of the file you want to protect:"
```

```
select FILENAME in *;
```

```
do
```

```
    echo "You picked $FILENAME ($REPLY), it is now only accessible to you."
```

```
    chmod go-rwx "$FILENAME"
```

```
done
```

```
exit 0
```

Циклы

```
#!/bin/sh
```

```
echo "This script can make any of the files in this directory private."  
echo "Enter the number of the file you want to protect:"
```

```
PS3="Your choice: "
```

```
QUIT="QUIT THIS PROGRAM - I feel safe now."
```

```
touch "$QUIT"
```

```
select FILENAME in *;
```

```
do
```

```
    case $FILENAME in
```

```
        "$QUIT")
```

```
            echo "Exiting."
```

```
            break
```

```
            ;;
```

```
        *)
```

```
            echo "You picked $FILENAME ($REPLY)"
```

```
            chmod go-rwx "$FILENAME"
```

```
            ;;
```

```
    esac
```

```
done
```

```
rm "$QUIT"
```

```
exit 0
```

Циклы

- *break* или *break N* – прерывание выполнения цикла
- *continue* или *continue N* – досрочный выполнение очередной итерации

```
#!/bin/sh

LIMIT=19  # Upper limit

echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)."

a=0

while [ $a -le "$LIMIT" ]
do
    a=$((a+1))

    if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # Excludes 3 and 11.
    then
        continue      # Skip rest of this particular loop iteration.
    fi

    echo -n "$a "      # This will not execute for 3 and 11.
done

echo; echo

echo Printing Numbers 1 through 20, but something happens after 2.
```


ЦИКЛЫ

Same loop, but substituting 'break' for 'continue'.

```
a=0
```

```
while [ "$a" -le "$LIMIT" ]  
do
```

```
    a=$((a+1))
```

```
    if [ "$a" -gt 2 ]  
    then
```

```
        break # Skip entire rest of loop.
```

```
    fi
```

```
    echo -n "$a "
```

```
done
```

```
echo; echo
```

```
exit 0
```

Циклы

```
#!/bin/sh
```

```
# "break N" breaks out of N level loops.
```

```
for outerloop in 1 2 3 4 5
```

```
do
```

```
    echo -n "Group $outerloop:  "
```

```
    # -----
```

```
    for innerloop in 1 2 3 4 5
```

```
    do
```

```
        echo -n "$innerloop "
```

```
        if [ "$innerloop" -eq 3 ]
```

```
        then
```

```
            break # Try break 2 to see what happens.
```

```
            # ("Breaks" out of both inner and outer loops.)
```

```
        fi
```

```
    done
```

```
    # -----
```

```
    echo
```

```
done
```

```
echo
```

```
exit 0
```

Циклы

```
#!/bin/sh
```

```
# The "continue N" command, continuing at the Nth level loop.
```

```
for outer in I II III IV V           # outer loop
```

```
do
```

```
    echo; echo -n "Group $outer: "
```

```
    # -----
```

```
    for inner in 1 2 3 4 5 6 7 8 9 10 # inner loop
```

```
    do
```

```
        if [ "$inner" -eq 7 ]
```

```
        then
```

```
            continue 2 # Continue at loop on 2nd level, that is "outer
loop".
```

```
            # Replace above line with a simple "continue"
```

```
            # to see normal loop behavior.
```

```
        fi
```

```
        echo -n "$inner " # 7 8 9 10 will never echo.
```

```
    done
```

```
    # -----
```

```
done
```

```
echo; echo
```

```
exit 0
```



Operator case

```
case EXPRESSION in  
CASE1)  COMMAND-LIST;;  
CASE2)  COMMAND-LIST;;  
...  
CASEN)  COMMAND-LIST;;  
esac
```

Operator case

```
#!/bin/sh

echo "Please talk to me ..."
while :
do
    read INPUT_STRING
    case $INPUT_STRING in
        hello)
            echo "Hello yourself!"
            ;;
        bye)
            echo "See you again!"
            break
            ;;
        *)
            echo "Sorry, I don't understand"
            ;;
    esac
done
echo
echo "That's all folks!"

exit 0
```

Предопределенные переменные

- Переменная \$0 – это базовое имя программы, по которому она была вызвана
- Переменные \$1 .. \$9 – это первые 9 дополнительных параметров, которые получил скрипт при вызове
- Переменная @\$ – это список всех параметров с \$1 и до «бесконечности»
- Переменная \$# – это количество параметров, переданных скрипту

Предопределенные переменные

```
#!/bin/sh
```

```
echo "I was called with $# parameters"
```

```
echo "My name is $0"
```

```
echo "My first parameter is $1"
```

```
echo "My second parameter is $2"
```

```
echo "All parameters are $@"
```

```
exit 0
```

```
echo "My name is `basename $0`"
```

Предопределенные переменные

```
#!/bin/sh
```

```
while [ "$#" -gt "0" ]
```

```
do
```

```
    echo "\$1 is $1"
```

```
    shift
```

```
done
```

```
exit 0
```


Предопределенные переменные

- Еще одна специальная переменная – \$?

```
#!/bin/sh

/usr/local/bin/my-command
if [ "$?" -ne "0" ]; then
    echo "Sorry, we had a problem there!"
fi

exit 0
```

Предопределенные переменные

- Еще две переменные, значения которых устанавливаются оболочкой – это \$\$ и \$!
- Они представляют собой идентификаторы процессов
- IFS (*Internal Field Separator*)

```
#!/bin/sh
```

```
old_IFS="$IFS"
```

```
IFS=:
```

```
echo "Please input three data separated by colons ..."
```

```
read x y z
```

```
IFS=$old_IFS
```

```
echo "x is $x y is $y z is $z"
```

```
exit 0
```

Предопределенные переменные

```
foo=sun
```

```
echo $fooshine      # $fooshine is undefined
```

```
echo ${foo}shine    # displays the word "sunshine"
```

Предопределенные переменные

```
#!/bin/sh
```

```
echo -en "What is your name [ `whoami` ] "
```

```
read myname
```

```
if [ -z "$myname" ]; then
```

```
    myname=`whoami`
```

```
fi
```

```
echo "Your name is : $myname"
```

```
exit 0
```

Предопределенные переменные

```
#!/bin/sh
```

```
echo -en "What is your name [ `whoami` ] "
```

```
read myname
```

```
echo "Your name is : ${myname:-`whoami`}"
```

```
exit 0
```

Предопределенные переменные

```
#!/bin/sh
```

```
echo -en "What is your name [ `whoami` ] "
```

```
read myname
```

```
echo "Your name is : ${myname:=Dima  
Medvedev}"
```

```
exit 0
```

Константы

```
readonly OPTION VARIABLE(s)
```

```
readonly DATA=/home/sales/data/feb09.dat
```

```
echo $DATA
```

```
/home/sales/data/feb09.dat
```

```
DATA=/tmp/foo
```

```
# Error ... readonly variable
```

```
unset DATA
```

```
# Error ... readonly variable
```

Массивы

`ARRAY [INDEXNR]=value`

- или

`declare -a ARRAYNAME`

- или

`ARRAY=(value1 value2 ... valueN)`

- Доступ

`ARRAYNAME[indexnumber]=value`

- «Деструктор» элемента массива

`$unset ARRAY[1]`

- Нельзя удалить константу (см. предыдущий слайд)

Функции

```
function FUNCTION { COMMANDS; }
```

```
FUNCTION () { COMMANDS; }
```

```
#!/bin/sh
```

```
hello () { echo "hello, $1"; }
```

```
hello abrakadabra
```

```
exit 0
```

ФУНКЦИИ

```
#!/bin/sh
```

```
echo "This script demonstrates function arguments."
```

```
echo
```

```
echo "Positional parameter 1 for the script is $1."
```

```
echo
```

```
test()
```

```
{
```

```
    echo "Positional parameter 1 in the function is $1."
```

```
    RETURN_VALUE=$?
```

```
    echo "The exit code of this function is $RETURN_VALUE."
```

```
}
```

```
test other_param
```

```
exit 0
```

ФУНКЦИИ

```
#!/bin/sh
```

```
myfunc()
```

```
{
```

```
    echo "I was called as : $@"
```

```
    x=2
```

```
}
```

```
### Main script starts here
```

```
echo "Script was called with $@"
```

```
x=1
```

```
echo "x is $x"
```

```
myfunc 1 2 3
```

```
echo "x is $x"
```

```
exit 0
```

ФУНКЦИИ

```
#!/bin/sh

myfunc()
{
    echo "\$1 is $1"
    echo "\$2 is $2"
    # cannot change $1 - we'd have to say:
    # 1="Goodbye Cruel"
    # which is not a valid syntax. However, we can
    # change $a:
    a="Goodbye Cruel"
}

### Main script starts here

a=Hello
b=World
myfunc $a $b
echo "a is $a"
echo "b is $b"

exit 0
```

Функции

```
#!/bin/sh

myfunc()
{
    a="Goodbye Cruel"
}

### Main script starts here

echo "before myfunc a is $a"

myfunc

echo "after myfunc a is $a"

exit 0
```

Функции

```
local var=value
```

```
local varName
```

```
#!/bin/sh
```

```
func()
```

```
{
```

```
    local loc_var=23           # local variable declaration
```

```
    echo
```

```
    echo "\"loc_var\" inside the function = $loc_var"
```

```
    global_var=999             # global variable declaration
```

```
    echo "\"global_var\" inside the function =  
    $global_var"
```

```
}
```

```
func
```

ФУНКЦИИ

```
# Check if local variable is visible outside the function
```

```
echo
```

```
echo "\"loc_var\" outside the function = $loc_var"
```

```
# "loc_var" outside the function =
```

```
# So, $loc_var invisible in global context
```

```
echo "\"global_var\" outside the function = $global_var"
```

```
# "global_var" outside the function = 999
```

```
# $global_var has global scope
```

```
echo
```

```
exit 0
```

Функции

- Функции могут возвращать значения одним из четырех способов:
 - Изменение состояния одной или нескольких переменных
 - Использование команды *exit* в конце скрипта
 - Использованием команды *return* для досрочного переход в конец функции и возврата необходимого значения вызвавшей стороне
 - *echo*-печать в *stdout*, который будет перехвачен вызвавшей стороной
`c= `expr $a + $b``

ФУНКЦИИ

```
exit [N]
```

```
return [N]
```

```
#!/bin/sh
```

```
factorial()
```

```
{
```

```
    if [ "$1" -gt "1" ]; then
```

```
        i=`expr $1 - 1`
```

```
        j=`factorial $i`
```

```
        k=`expr $1 \* $j`
```

```
        echo $k
```

```
    else
```

```
        echo 1
```

```
    fi
```

```
}
```

```
while :
```

```
do
```

```
    echo "Enter a number:"
```

```
    read x
```

```
    [ $x -lt 0 ] && echo "Wrong number! Repeat, please!" && continue
```

```
    factorial $x
```

```
done
```

```
echo
```

```
exit 0
```

ФУНКЦИИ

```
#!/bin/sh

factorial()
{
    local k
    if [ "$1" -gt "1" ]; then
        i=`expr $1 - 1`
        factorial $i
        j=$?
        k=`expr $1 \* $j`
    else
        k=1
    fi
    return $k
}

while :
do
    echo "Enter a number:"
    read x
    [ $x -lt 0 ] && echo "Wrong number! Repeat, please!" && continue
    factorial $x
    echo $?
done
echo

exit 0
```

ФУНКЦИИ

```
./library.sh
source ./library.sh

-----

# common.lib
# Note no #!/bin/sh as this should not spawn
# an extra shell. It's not the end of the world
# to have one, but clearer not to.
#
STD_MSG="About to rename some files..."
rename()
{
    # expects to be called as: rename .txt .bak
    FROM=$1
    TO=$2
    for i in *$FROM
    do
        j=`basename $i $FROM`
        mv $i ${j}$TO
    done
}
```

ФУНКЦИИ

```
#!/bin/sh
# function1.sh
. ./common.lib
echo $STD_MSG
rename txt bak
```

```
exit 0
```

```
#!/bin/sh
# function2.sh
. ./common.lib
echo $STD_MSG
rename html html-bak
```

```
exit 0
```

Команда shift

- Принимает один числовой аргумент
- Позиционные параметры сдвигаются влево на N
- Позиционные параметры от $N+1$ до $\$ \#$ переименовываются и становятся списком от $\$1$ до $\$ \# - N + 1$
- Аргумент $\$0$ не затрагивается
- Если $N == 0$ или больше, чем $\$ \#$, то позиционные параметры не меняются
- Если N не представлена, она принимается равной 1
- Возвращаемый статус равен нулю, если N больше, чем $\$ \#$ или меньше нуля; в противном случае он не равен 0

Команда shift

```
#!/bin/sh
echo "$@"
shift 4
echo "$@"
exit 0
```

```
-----
#!/bin/sh
# This script can clean up files that were last accessed over 365
# days ago.
USAGE="Usage: $0 dir1 dir2 dir3 ... dirN"
if [ "$#" == "0" ]; then
    echo "$USAGE"
    exit 1
fi
while (( "$#" )); do
    if [[ $(ls "$1") == "" ]]; then
        echo "Empty directory, nothing to be done."
    else
        find "$1" -type f -a -atime +365 -exec rm -i {} \;
    fi
    shift
done
exit 0
```

Ловушки

- Все сигналы можно просмотреть командой *kill*:

```
$ kill -l
```

- Аналогично по умолчанию команда *kill* отправляет сигнал *TERM*
- Отправим этот сигнал процессу с идентификатором 1234:

```
$ kill 1234
```

Signal name	Signal value	Effect
SIGHUP	1	Hangup
SIGINT	2	Interrupt from keyboard
SIGKILL	9	Kill signal
SIGTERM	15	Termination signal
SIGSTOP	17,19,23	Stop the process



Ловушки

```
trap [COMMANDS] [SIGNALS]
```

```
trap arg signal
```

```
trap command signal
```

```
trap 'action' signal1 signal2 signalN
```

```
trap 'action' SIGINT
```

```
trap 'action' SIGTERM SIGINT SIGFPE SIGSTP
```

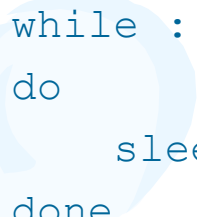
```
trap 'action' 15 2 8 20
```


Ловушки




```
#!/bin/sh
```

```
trap "echo Booms!" SIGINT SIGTERM  
echo "pid is $$"
```



```
while :                # This is the same as "while true".  
do  
    sleep 60    # This script is not really doing anything.  
done  
  
exit 0
```



Ловушки

```
trap - signal
```

```
trap - signal1 signal2
```

```
file=/tmp/test4563.txt
```

```
trap 'rm $file' 1 2 3 15
```

```
trap
```

```
trap - SIGINT
```

```
trap
```

```
trap - 1 2 3 15
```

```
trap
```

ЛОВУШКИ

```
# Shell script to find out odd or even number
# provided by the user
# ----
# set variables to an integer attribute
times=0
n=0

# capture CTRL+C, CTRL+Z and quit singles using the
# trap
trap 'echo " disabled"' SIGINT SIGQUIT SIGTSTP

# set an infinite while loop
# user need to enter -9999 to exit the loop
while :
do
...

```

Ловушки

```
...  
# get date  
read -p "Enter number (-9999 to exit) : " n  
# if it is -9999 die  
[ $n -eq -9999 ] && { echo "Bye!"; break; }  
# find out if $n is odd or even  
ans=$(( n % 2 ))  
# display result  
[ $ans -eq 0 ] && echo "$n is an even number." || echo "$n is  
an odd number."  
# increase counter by 1  
times=$(( ++times ))  
done  
  
# reset all traps  
trap - SIGINT SIGQUIT SIGTSTP  
  
# display counter  
echo "You played $times times."  
  
exit 0
```

Ловушки

```
# define die()
die()
{
    echo "... "
}
# set trap and call die()
trap 'die' 1 2 3 15
....
...
```

Ловушки

```
# Shell script to find out odd or even number provided by the user
# ----
# set variables to an integer attribute
times=0
n=0

# define function
warning()
{
    echo -e "\n*** CTRL+C and CTRL+Z keys are disabled. Please enter
    number only. Hit [Enter] key to continue..."
}

# capture CTRL+C, CTRL+Z and quit singles using the trap
trap 'warning' SIGINT SIGQUIT SIGTSTP
...
```

Ловушки

...

```
# set an infinite while loop
# user need to enter -9999 to exit the loop
while :
do
    # get date
    read -p "Enter number (-9999 to exit) : " n
    # if it is -9999 die
    [ $n -eq -9999 ] && { echo "Bye!"; break; }
    # find out if $n is odd or even
    ans=$(( n % 2 ))
    # display result
    [ $ans -eq 0 ] && echo "$n is an even number." || echo
"$n is an odd number."
    # increase counter by 1
    times=$(( ++times ))
done
```

...

Ловушки



...

```
# reset all traps
```

```
trap - SIGINT SIGQUIT SIGTSTP
```

```
# display counter
```

```
echo "You played $times times."
```

```
exit 0
```


«Полезности»

```
#!/bin/sh
MIN=200
MAX=500
let "scope = $MAX - $MIN"
if [ "$scope" -le "0" ]; then
echo "Error - MAX is less than MIN!"
fi
for i in `seq 1 10`
do
let result="$RANDOM % $scope + $MIN"
echo "A random number between $MIN and $MAX is
    $result"
done
exit 0
```

«Полезности»

```
eval var1=\$$var2
```

```
#!/bin/sh
```

```
a=letter
```

```
letter=z
```

```
echo
```

```
# Прямое обращение
```

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

```
# Косвенное обращение
```

```
eval a=\$$a
```

```
echo "А теперь a = $a"
```

```
echo
```

```
exit 0
```

«Полезности»

```
`${!variable}
```

```
#!/bin/sh
```

```
a=letter
```

```
letter=z
```

```
echo "a = $a" # Direct reference.
```

```
echo "Now a = `${!a}`" # Indirect reference.
```

```
# The `${!variable}` notation is more intuitive than  
the old eval var1=\$$var2
```

```
echo
```

```
exit 0
```

«Полезности»

```
a=$(( 5 + 3 ))
```

```
#!/bin/sh
```

```
echo
```

```
(( a = 23 ))
```

```
echo "a (начальное значение) = $a"
```

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

```
echo "a (после a++) = $a"
```

```
(( a-- ))
```

```
echo "a (после a--) = $a"
```

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

```
echo "a (после ++a) = $a"
```

```
(( --a ))
```

```
echo "a (после --a) = $a"
```

```
echo
```

```
(( t = a<45?7:11 ))
```

```
echo "If a < 45, then t = 7, else t = 11."
```

```
echo "t = $t "
```

```
echo
```

```
exit 0
```

«Полезности»

```
COMMAND_OUTPUT >
# Redirect stdout to a file.
# Creates the file if not present, otherwise overwrites it.
ls -lR > dir-tree.list
# Creates a file containing a listing of the directory tree.
: > filename
# The > truncates file "filename" to zero length.
# If file not present, creates zero-length file (same effect as 'touch').
# The : serves as a dummy placeholder, producing no output.
> filename
# The > truncates file "filename" to zero length.
# If file not present, creates zero-length file (same effect as 'touch').
# (Same result as ": >", above, but this does not work with some shells.)
COMMAND_OUTPUT >>
# Redirect stdout to a file.
# Creates the file if not present, otherwise appends to it.
# Single-line redirection commands (affect only the line they are on):
# -----1>filename
# Redirect stdout to file "filename."
1>>filename
# Redirect and append stdout to file "filename."
2>filename
# Redirect stderr to file "filename."
2>>filename
# Redirect and append stderr to file "filename."
&>filename
# Redirect both stdout and stderr to file "filename."
# This operator is now functional, as of Bash 4, final release.
```

«Полезности»

```
M>N
# "M" is a file descriptor, which defaults to 1, if not explicitly set.
# "N" is a filename.
# File descriptor "M" is redirect to file "N."
M>&N
# "M" is a file descriptor, which defaults to 1, if not set.
# "N" is another file descriptor.
#=====
# Redirecting stdout, one line at a time.
LOGFILE=script.log
echo "This statement is sent to the log file, \"$LOGFILE\"." 1>$LOGFILE
echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE
echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE
echo "This statement is echoed to stdout, and will not appear in \"$LOGFILE\"."
# These redirection commands automatically "reset" after each line.
# Redirecting stderr, one line at a time.
ERRORFILE=script.errors
bad_command1 2>$ERRORFILE # Error message sent to $ERRORFILE.
bad_command2 2>>$ERRORFILE # Error message appended to $ERRORFILE.
bad_command3 # Error message echoed to stderr,
#+ and does not appear in $ERRORFILE.
# These redirection commands also automatically "reset" after each line.
#=====
```

«Полезности»

```
2>&1
# Redirects stderr to stdout.
# Error messages get sent to same place as standard output.
>>filename 2>&1
bad_command >>filename 2>&1
# Appends both stdout and stderr to the file "filename" ...
2>&1 | [command(s)]
bad_command 2>&1 | awk '{print $5}' # found
# Sends stderr through a pipe.
# |& was added to Bash 4 as an abbreviation for 2>&1 |.
i>&j
# Redirects file descriptor i to j.
# All output of file pointed to by i gets sent to file pointed to by j.
>&j
# Redirects, by default, file descriptor 1(stdout) to j.
# All stdout gets sent to file pointed to by j.
```

«Полезности»

```
0< FILENAME
< FILENAME
# Accept input from a file.
# Companion command to ">", and often used in combination with it.
#
# grep search-word <filename
[j]<>filename
# Open file "filename" for reading and writing,
#+ and assign file descriptor "j" to it.
# If "filename" does not exist, create it.
# If file descriptor "j" is not specified, default to fd 0, stdin.
#
# An application of this is writing at a specified place in a file.
echo 1234567890 > File # Write string to "File".
exec 3<> File # Open "File" and assign fd 3 to it
read -n 4 <&3 # Read only 4 characters.
echo -n . >&3 # Write a decimal point there.
exec 3>&- # Close fd 3.
cat File # ==> 1234.67890
# Random access, by golly.
|
# Pipe.
# General purpose process and command chaining tool.
# Similar to ">", but more general in effect.
# Useful for chaining commands, scripts, files, and programs together.
cat *.txt | sort | uniq > result-file
# Sorts the output of all the .txt files and deletes duplicate lines,
# finally saves results to "result-file".
```


«Полезности»

```
command < input-file > output-file
```

```
# Or the equivalent:
```

```
< input-file command > output-file # Although this is non-standard.
```

```
command1 | command2 | command3 > output-file
```

```
ls -yz >> command.log 2>&1
```

```
# Capture result of illegal options "yz" in file "command.log."
```

```
# Because stderr is redirected to the file,
```

```
#+ any error messages will also be there.
```

```
# Note, however, that the following does *not* give the same result.
```

```
ls -yz 2>&1 >> command.log
```

```
# Outputs an error message, but does not write to file.
```

```
# More precisely, the command output (in this case, null)
```

```
#+ writes to the file, but the error message goes only to stdout.
```

```
# If redirecting both stdout and stderr,
```

```
#+ the order of the commands makes a difference.
```

«Полезности»

```
n<&-Close input file descriptor n.  
0<&-, <&-Close stdin.  
n>&-Close output file descriptor n.  
1>&-, >&-Close stdout.
```

```
# Redirecting only stderr to a pipe.  
exec 3>&1 # Save current "value" of stdout.  
ls -l 2>&1 >&3 3>&- | grep bad 3>&- # Close fd 3 for 'grep' (but not 'ls').  
# ^^^^ ^^^^  
exec 3>&- # Now close it for the remainder of the script.
```

«Полезности»

```
if [ -z "$1" ]
then
  Filename=names.data # Default, if no filename specified.
else
  Filename=$1
fi
#+ Filename=${1:-names.data}
# can replace the above test (parameter substitution).
count=0
echo
while [ "$name" != Smith ] # Why is variable $name in quotes?
do
  read name # Reads from $Filename, rather than stdin.
  echo $name
  let "count += 1"
done <"$Filename" # Redirects stdin to file $Filename.
# ^^^^^^^^^^^^^
echo; echo "$count names read"; echo
exit 0
```

See also

- Искусство программирования на языке сценариев командной оболочки- http://www.opennet.ru/docs/RUS/bash_scripting_guide/
- Береснев, А. Администрирование GNU/Linux с нуля (+ CD-ROM). – СПб.: БХВ-Петербург, 2010. – 576 с.
- Bash Guide for Beginners - <http://tille.garrels.be/training/bash/>
- GNU Bash - <http://www.gnu.org/software/bash/>
- Home Page For The KornShell Command And Programming Language - <http://www.kornshell.com/>
- Zsh - <http://www.zsh.org/>
- Tcsh Homepage - <http://www.tcsh.org>
- BUSYBOX - <http://www.busybox.net/>
- Bash Guide for Beginners - <http://tille.garrels.be/training/bash/>
- Linux Shell Scripting Tutorial (LSST) v2.0 - http://bash.cyberciti.biz/guide/Main_Page