

Работа в командной оболочке `bash`

BASH — инструмент

Использование BASH поможет сократить время на решение административных задач.

Терминал — это программа, дающая доступ к ОС. Одновременно можно открывать несколько окон терминала.

Оболочка (shell) — это программа, которая позволяет отправлять операционной системе команды, написанные на особом языке.

Интерфейс командной строки (Command Line Interface, CLI) — это способ взаимодействия человека и компьютера, (пользователь вводит команды с клавиатуры, а компьютер выводит в текстовом виде сообщения для пользователя).

Графический пользовательский интерфейс (Graphical User Interface, GUI) - при работе с которым, в основном, используется мышь.

BASH — инструмент

Bash (Bourne Again SHell) — самый распространённый язык командной оболочки, используемый для взаимодействия с операционной системой. Умение эффективно пользоваться этим инструментом позволяет экономить немало времени.

Bash - это командный процессор, который, работает в текстовом окне, что позволяет пользователю вводить команды вызывающие определенные действия. Bash также может читать команды из файла, который называется **скриптом**.

Какие командные процессоры зарегистрированы в системе можно посмотреть командой

```
cat /etc/shells
```

Переменные

Переменные, определенные оболочкой и используемые программами во время выполнения. Они могут определяться **системой** и **пользователем**. В bash **переменные не имеют типа**

Переменная окружения может быть трех типов

- 1. ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ** (Эти переменные определены только для текущей сессии)
- 2. ПОЛЬЗОВАТЕЛЬСКИЕ ПЕРЕМЕННЫЕ** (Эти переменные оболочки в Linux определяются для конкретного пользователя и загружаются каждый раз когда он входит в систему)
- 3. СИСТЕМНЫЕ ПЕРЕМЕННЫЕ** (Эти переменные доступны во всей системе, для всех пользователей)

Конфигурационные файлы

~/.bashrc - файл переменных конкретного пользователя.

~/.bash_profile - файл переменных переменного подключения.

/etc/environment - тот файл для создания, редактирования и удаления каких-либо переменных окружения на системном уровне. Доступны для всей системы, для каждого пользователя.

/etc/bashrc - Системный bashrc для локальных пользователей.

/etc/profile - Системный файл profile. Переменные доступны пользователю если он вошел удаленно.

Переменные окружения

\$EDITOR - текстовый редактор по умолчанию

\$UID - ...содержит реальный идентификатор, который устанавливается при логине.

\$HOME - домашний каталог пользователя

\$HOSTTYPE - архитектура машины.

\$LC_CTYPE - внутренняя переменная, которая определяет кодировку символов

\$OSTYPE - тип ОС

\$PATH - путь поиска программ

\$PPID - идентификатор родительского процесса

\$SECONDS - время работы скрипта(в сек.)

\$# - общее количество параметров переданных скрипту

\$* - все аргументы переданные скрипту(выводятся в строку)

\$@ - тоже самое, что и предыдущий, но параметры выводятся в столбик

\$_ - PID последнего запущенного в фоне процесса

\$\$ - PID самого скрипта

Команды Bash

Безусловно, BASH не только командная оболочка, это еще и превосходный скриптовый язык программирования.

Основная задача оболочки - **выполнять команды**, но кроме утилит, расположенных в файловой системе, Bash имеет **свой набор команд**, многими из которых вы пользуетесь каждый день сами этого не понимая. Эти команды Bash находятся не на диске, а **встроены в саму оболочку**.

И конечно же кроме встроенных команд используются целая куча внешних, отдельных команд-программ.

Комментарии

Комментарии начинаются с символа # (кроме первой строки).

Любой bash-скрипт должен начинаться со строки:

#!/bin/bash

в этой строке после #! указывается путь к bash-интерпретатору, вспомнить где установлен bash можно, набрав **whereis bash**

```
Файл  Правка  Вид  Поиск  Терминал  Помощь
[july@localhost ~]$ whereis bash
bash: /usr/bin/bash /usr/share/man/man1/bash.1.gz /usr/share/info/bash.info.gz
[july@localhost ~]$ |
```

Выполнение команд

Командная строка позволяет выполнить несколько команд за один раз, введя их через точку с запятой.

Можно совмещать много команд в одной строке, ограничение — лишь в максимальном количестве аргументов, которое можно передать программе. Определить это ограничение можно с помощью такой команды

getconf ARG_MAX

```
[july@localhost dir]$ ls
[july@localhost dir]$ pwd
/home/july/dir
[july@localhost dir]$ ps
  PID TTY          TIME CMD
 20992 pts/0    00:00:00 bash
 21371 pts/0    00:00:00 ps
[july@localhost dir]$ |
```

```
[july@localhost dir]$ ls; pwd; ps
/home/july/dir
  PID TTY          TIME CMD
 20992 pts/0    00:00:00 bash
 21465 pts/0    00:00:00 ps
[july@localhost dir]$ |
```

```
[july@localhost dir]$ getconf ARG_MAX
2097152
```

Запуск файла сценария

Скрипты **желательно** располагать в папке `~/bin`, `~/.local/bin`, `/bin`

Если при запуске появляется ошибка **Permission denied**

Необходимо сделать этот файл исполняемым

`chmod +x ./myscript`

После настройки разрешений всё будет работать как надо

Исполнение скрипта без изменений прав доступа.

`bash myscript`
`. myscript`

Обработка ошибок и завершение

exit — может использоваться для завершения работы сценария

Типовые ошибки:

- Синтаксические ошибки:
 - Отсутствующие кавычки
 - Отсутствующие или неожиданные лексемы
- Непредвиденная подстановка
- Логические ошибки

Первый сценарий

```
#!/bin/bash
```

```
#указываем где у нас хранится bash-интерпретатор
```

```
parametr1=$1 #первый параметр переданный скрипту
```

```
script_name=$0 #имя скрипта
```

```
echo "Вы запустили скрипт с именем $script_name и параметром  
$parametr1"
```

```
echo "Вы запустили скрипт с именем \script_name и параметром  
\parametr1"
```

```
echo 'Вы запустили скрипт с именем $script_name и параметром $parametr1'
```

```
exit 0 #Выход с кодом 0 (удачное завершение работы скрипта)
```

```
Вы запустили скрипт с именем 11111.txt и параметром sssss  
Вы запустили скрипт с именем $script_name и параметром $parametr1  
Вы запустили скрипт с именем $script_name и параметром $parametr1
```


Подстановка команд

Можно извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария. Сделать это можно двумя способами.

`mydir=`pwd`` или `mydir=$(reboot)`

Иногда срабатывает простая конструкция, но с ней нужно быть внимательнее. Такую конструкцию лучше избегать.

`mydir=reboot`

Математические операции

Для математических операций используют конструкцию **`$((a+b))`**:

```
var1=$(( 5 + 5 )); echo $var1  
var2=$(( $var1 * 2 )); echo $var2
```

let это встроенная функция bash.

```
let c=$a+$b; let c++; let c=c%2
```

expr похож на **let**, но по умолчанию выводится ответ.

```
x = $(expr 2 + 2); x = $(expr 5 \* $A);
```

В bash нет родной поддержки деления чисел с плавающей точкой. Для получения результата в дробном виде необходимо использовать команду **bc**

```
D=$(bc<<<"scale=3;4/3"); echo "D: $D"  
D=$( echo "scale=3;4/3" | bc -l ); echo "D: $D"
```

Алгоритмические конструкции

- Следование
- Условие
- Выбор
- Цикл
- Подпрограмма (функция)

Следование

- Следование — последовательное выполнение команд по очереди
- Последовательность команд, каждая на своей строке

cmd1

cmd2

cmd3

cmd1

- Последовательность команд, разделенных символом «;»

cmd1; cmd2; cmd3

- Группу команд можно объединять помощью { } или ()
- В случае использования () команды будут запущены в дочернем процессе

Условный оператор if. Условия.

Сравнение чисел

В скриптах можно сравнивать числовые значения

n1 -eq n2	истинное значение, если n1 = n2.
n1 -ge n2	истинное значение, если n1 >= n2.
n1 -gt n2	истинное значение, если n1 > n2.
n1 -le n2	истинное значение, если n1 <= n2.
n1 -lt n2	истинное значение, если n1 < n2.
n1 -ne n2	истинное значение, если n1 != n2.

Сравнение в условиях заключается в квадратные скобки

```
if [ n1 -eq n2 ]
```

Условный оператор if. Условия.

Сравнение строк

str1 = str2

str1 != str2

str1 < str2

str1 > str2

-n str1

-z str1

Проверяет строки на равенство

Возвращает истину, если строки не идентичны

Возвращает истину, если str1 меньше, чем str2.

Возвращает истину, если str1 больше, чем str2.

Возвращает истину, если длина str1 больше нуля.

Возвращает истину, если длина str1 равна нулю.

Сравнение в условиях заключается в квадратные скобки

```
if [ str1 != str2 ]
```

Условный оператор if. Условия.

Проверки условий для файлов

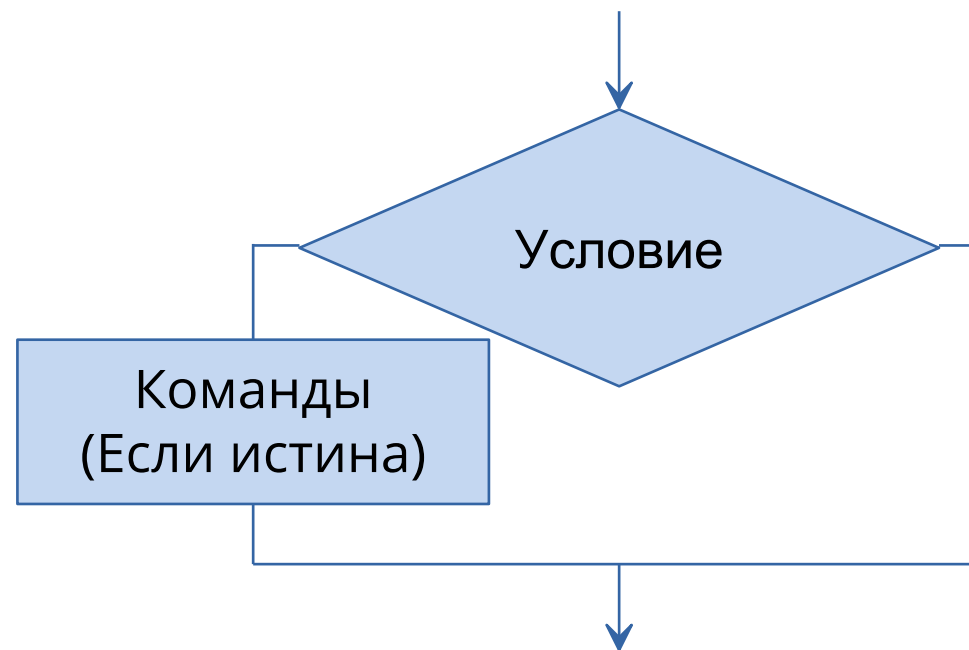
-d file	существует ли файл, и является ли он директорией.
-e file	существует ли файл.
-f file	существует ли файл, и является ли он файлом.
-r file	существует ли файл, и доступен ли он для чтения.
-s file	существует ли файл, и не является ли он пустым.
-w file	существует ли файл, и доступен ли он для записи.
-x file	существует ли файл, и является ли он исполняемым.
file1 -nt file2	новее ли file1, чем file2.
file1 -ot file2	старше ли file1, чем file2.

Сравнение в условиях заключается в квадратные скобки

```
if [ str1 != str2 ]
```

Условный оператор if

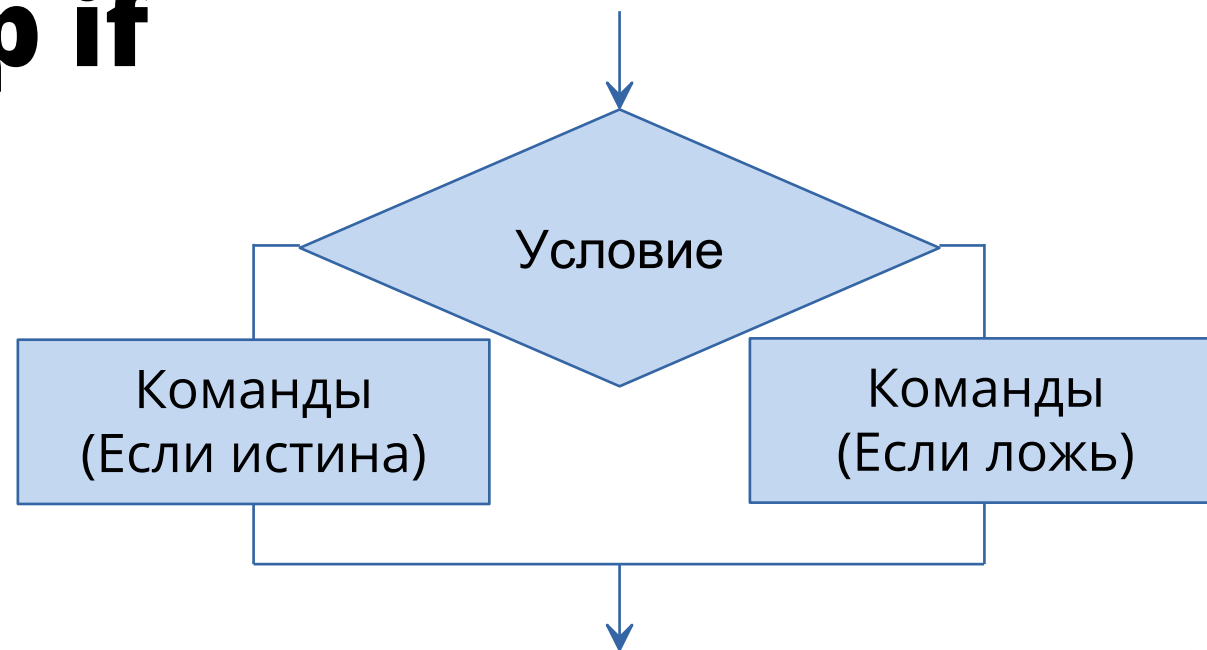
```
if условие
then
    команды
fi
```



```
if [ ! "$1" ] #Проверяем передачу первого параметра
then
    echo "Имя файла не задано. Ошибка 3"; exit 3
fi
if [ ! -f "$1" ] #Проверяем наличие файла
then
    echo "Имя файла $1 не найдено. Ошибка 2"; exit 2
fi
```


Условный оператор if

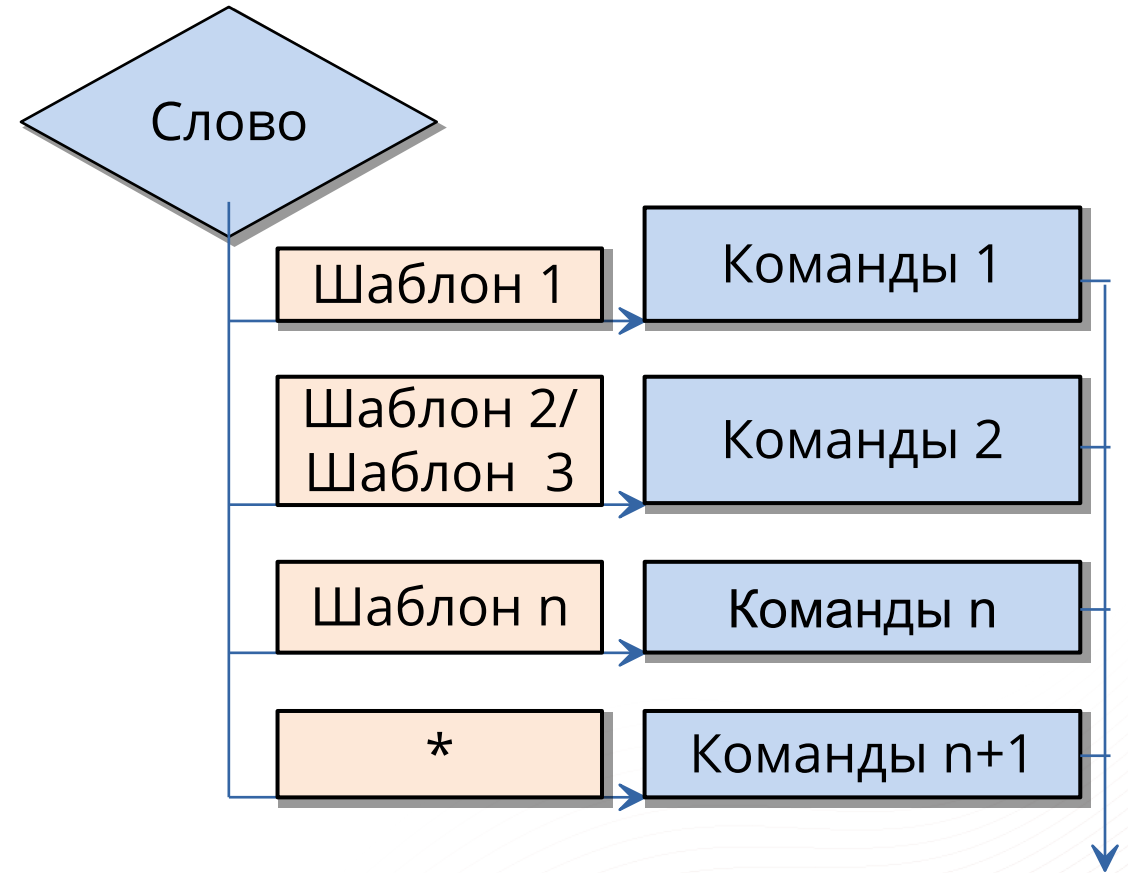
```
if условие
then
    команды «если истина»
else
    команды «если ложь»
fi
```



```
echo "Введен пользователь $1"
if grep $1 /etc/passwd
then
    echo "Такой пользователь найден"
else
    echo "Такого пользователя нет"
fi
```

Конструкция case

```
read СЛОВО
case $СЛОВО in
  ШАБЛОН1)
    КОМАНДЫ1
  ;;
  ШАБЛОН2 | ШАБЛОН3)
    КОМАНДЫ2
  ;;
  *)
    КОМАНДЫ3
  ;;
esac
```



Условная конструкция

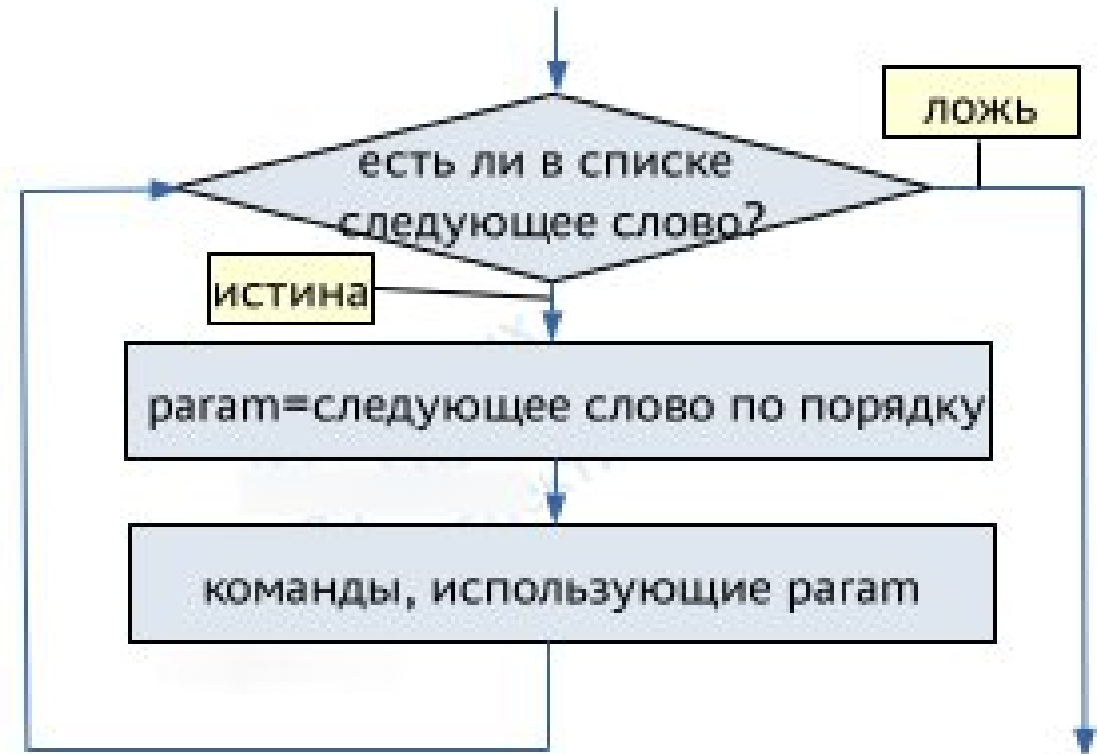
Комбинируя **&&** и **||** можно строить сложные варианты объединения команд.

```
true || echo "1" || echo "2" || false && echo "4" && echo "5" || echo "6"  
true || echo "1" && (echo "2"; echo "3" || echo "4") && echo "5" || echo "6"  
true || echo "1" && { echo "2"; echo "3" || echo "4"; } && echo "5" || echo "6"
```

Цикл for

```
for параметр in слово1 слово2
do
    команды
done
```

```
#for var in {10..14}
#for var in `seq 10 1 14`
#for (( var=10; var<15; var++ ))
#for var in 1 two "три файл"
#for var in /home/dima/*
for var in $(ls /home/dima)
do
    echo "Это элемент: $var"; file $var
done
```



Разделитель полей

Оболочка bash считает разделителями полей: Пробел, Знак таб, Знак перевода строки.

Если bash встречает в данных любой из этих символов, он считает, что перед ним — следующее самостоятельное значение списка. Для того, чтобы решить проблему, можно временно изменить переменную среды IFS (Internal Field Separator - Внутренний Разделитель Полей).

IFS=\$'\n' #разделитель перевод строки

IFS=: #разделитель двоеточие

IFS=\$'\t' #разделитель табуляция

IFS=\$'\n\t' #разделитель и табуляция и перенос строки

```
IFS=:  
for var in `cat /etc/passwd`  
do  
echo "Это элемент: $var"  
done
```

Цикл `while` — пока условие верно

`while` условие
`do`
 команды
`done`

```
#!/bin/bash
var1=5
while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$(( $var1 - 1 ))
done
```

Как только **`$var1`** примет значение **0**, цикл прекращается



Цикл `until` — пока условие НЕ верно

`until` условие
`do`
 команды
`done`

```
#!/bin/bash
var1=5
until [ $var1 -gt 10 ]
do
    echo $var1
    var1=$(( $var1 + 1 ))
done
```

Как только **`$var1`** примет значение **10**, цикл прекращается



Функции

Функция (подпрограмма) – программный код внутри сценария, к которому можно обратиться из другого места сценария

- Функция имеет имя, принимает параметры, возвращает код возврата, который сохраняется в \$?
- Функция не порождает дочернего процесса, похожа на сценарий в сценарии
- Синтаксис:

```
function имя {  
    ... команды  
}  
или  
имя_функции () {  
    ... команды  
}  
  
#!/bin/bash  
is_alive_ping() {  
    ping -c 1 $1 > /dev/null  
    [ $? -eq 0 ] && echo Хост с IP: $i поднят.  
}  
  
for i in 192.168.88.{1..254}  
do  
    is_alive_ping $i & disown  
done
```


Практическая работа

1. Разработать скрипт, производящий копирование заданного в командной строке файла (передаётся скрипту как аргумент), в десять файлов, имена которых генерируются автоматически.
2. Перед каждым копированием делать паузу в 3 секунды командой `sleep`.
3. Сгенерированные имена файлов должны содержать текущее время, используя, например команду `date +%m-%d`
4. Если исходный файл не существует, пользователю должна выдаваться ошибка с кодом 2.
5. Если скрипту не передан аргумент с именем файла, пользователю должна выдаваться ошибка с кодом 3

Пример команды `date`:

`date +%Y-%m-%d-%H-%M-%S`



Спасибо за внимание!

www.red-soft.ru
redos@red-soft.ru

