

Bash: Как писать и запускать bash-скрипты?

Category: bash & sh, bash программирование, GNU/Linux
2024-09-17

Что такое Shell.

Вся сила **GNU/Linux** в использовании терминала. Это такая командная оболочка, где вы можете выполнять различные команды, которые будут быстро и эффективно выполнять различные действия.

Для **GNU/Linux** было создано множество скриптов, которые выполняются в различных командных оболочках. Это очень удобно, вы просто объединяете несколько команд, которые выполняют определенное действие, а затем выполняете их одной командой или даже с помощью ярлыка.

Shell — это один из командных интерпретаторов, поставляемых вместе с **GNU/Linux**. Программа на языке называется скриптом. **Shell** скриптам указывают расширение *.sh и работают они в **GNU/Linux** системах.

Bash (Bourne-Again SHell) — это командный интерпретатор в **GNU/Linux**, который позволяет автоматизировать задачи на уровне командной строки.

Bash-скрипты — это файлы, содержащие последовательность команд, которые могут быть выполнены с помощью интерпретатора **bash**.

Bash-скрипты могут быть использованы для автоматизации повторяющихся задач. Например, если вам нужно каждый день создавать отчет и отправлять его по электронной почте, вы можете написать скрипт, который будет выполнять эти действия автоматически. Это позволит сэкономить много времени и уменьшить вероятность ошибок.

Bash-скрипты могут быть написаны в любом текстовом редакторе и должны иметь права на выполнение.

В начале каждого скрипта должна быть строка, называемая **шебанг (shebang)**, которая сообщает операционной системе, какой интерпретатор использовать для выполнения скрипта. Шебанг должен начинаться с символа решетки # и следующим за ним символ восклицания !, после которых указывается путь к интерпретатору.

Для использования интерпретатора **bash**, шебанг будет выглядеть следующим образом:

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

Также во время написания скрипта можно оставлять комментарии, которые начинаются с символа решетки и продолжаются до конца строки. Комментарии не будут выполнены интерпретатором и используются для описания функциональности скрипта.

```
# Это комментарий
```

Когда вы видите:

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

или

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

Просто имейте в виду, что командная оболочка **sh** старше, чем командная оболочка **bash**.

Так же командная оболочка **bash** является надмножеством командной оболочки **sh**, то есть все, что вы можете сделать **sh** вы можете также сделать и в **bash**. Во многих дистрибутивах **GNU\Linux**, ссылку `/bin/sh` можно интерпретировать как ссылку на `/bin/bash`.

Например, **bash** имеет больше функций (ветвление, встроенные массивы), что упрощает написание сценариев, а в `/bin/sh` нет истории или редактирования командной строки и еще нет контроля задания.

Существует множество других командных оболочек для **GNU\Linux**, которые мы не будем рассматривать в рамках данной инструкции.

Как работают скрипты.

В **GNU\Linux** почти не используется расширение файла для опережения его типа на системном уровне. Это могут делать файловые менеджеры и то не всегда. Вместо этого, используются сигнатуры начала файла и специальные флаги.

Система считает исполняемыми только те файлы, которым присвоен атрибут исполняемости.

Теперь о том, как работают скрипты. Это обычные файлы, которые содержат текст, но если для них установлен атрибут исполняемости, то

для их открытия используется специальная программа – интерпретатор, например, оболочки **sh** или **bash**, а уже интерпретатор читает последовательно строку за строкой и выполняет все команды, которые содержатся в файле.

У нас есть несколько способов выполнить запуск скрипта **GNU\Linux**. Мы можем запустить его как любую другую программу через терминал или же запустить оболочку и сообщить ей какой файл нужно выполнять. В этом случае не нужно даже флага исполняемости.

Как запустить sh скрипт из командной строки?

Допустим у вас есть скрипт `hello.sh` состоящий из одной команды.

```
# cd ~  
# mcedit hello.sh
```

Содержимое:

```
echo "Hello World"
```

Чтобы его запустить, надо зайти в каталог, где расположен скрипт, набрать название интерпретатора `sh` и первым параметром указать ваш файл `hello.sh`.

```
# sh hello.sh
```

Ответ:



Указать интерпретатор внутри файла.

Если указать интерпретатор внутри файла, чтобы постоянно не указывать интерпретатор в терминале, можно сделать скрипт исполняемым.

На первой строчке после `#!` прописываем путь к `bash`-интерпретатору, по умолчанию это `/usr/bin/bash` (посмотреть можно командой `whereis bash`).

```
# whereis bash
```

Ответ:



Вторая строка – это действие, которое выполняет скрипт, но нас больше всего интересует первая строка – это оболочка, с помощью которого скрипт нужно выполнить.

Это может быть не только `/usr/bin/bash`, но и `/bin/bash` или `/bin/sh`, и даже `/usr/bin/python` или `/usr/bin/php`. Также часто встречается ситуация, что путь к исполняемому файлу оболочки получают с помощью утилиты `env`, например, `/usr/bin/env`, `php` и так далее.

```
# cd ~  
# mcedit script.sh
```

Содержимое скрипта `script.sh` получается таким:

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

Сделать наш файл исполняемым.

Чтобы каждый раз не указывать интерпретатор в терминале, можно сделать скрипт исполняемым.

Нужно просто осознать, что в **GNU\Linux** не существует ассоциаций файлов по расширениям. Расширение для **GNU\Linux** – просто часть файла после последней точки, система никаких действий к этому не привязывает. Поэтому единственный способ делать `.sh` файлы исполняемыми – ставить на них флаг `executable`.

Чтобы выполнить скрипт в указанной оболочке, нужно установить для него флаг исполняемости.

Чтобы сделать файл исполняемым, можно воспользоваться командой **chmod** (от англ. **change mode** – *изменить режим*).

Команда **chmod** имеет следующий синтаксис:

```
# chmod [опции] права_доступа файл
```

- **права_доступа** – это специальный код, который задает права доступа к файлу или каталогу, а файл – это путь к файлу или каталогу, права доступа которой нужно изменить.

Для того чтобы сделать файл исполняемым, нужно добавить к его правам доступа бит исполнения `x`.

Для этого используется команда `chmod +x` и имя файла скрипта.

Например, запустим от **суперпользователя**:

```
$ sudo chmod +x script.sh
```

Эта команда `chmod +x` добавит права на выполнение для текущего пользователя.

Или вот так:

```
$ sudo chmod ugo+x script.sh
```

При установке прав с помощью символов используются группы символов:

- Кому будет предоставляться или запрещаться доступ: **владельцу** (u), **группе** (g), **остальным** (o) или **всем** (a). Можно указать как одну, так сразу несколько категорий.
- Какие действия: **чтение** (r), **запись** (w), **исполнение** (x).

Можно указывать сразу несколько прав.

Для обозначения запрещения, разрешения или назначения права используется один из символов: «-», «+» или «=».

Мы разрешаем выполнять запуск `sh` операционной системе **GNU\Linux** всем категориям пользователей – **владельцу**, **группе** файла и **остальным**.

Следующий важный момент – это то место где находится скрипт, если вы просто наберете `script.sh`, то поиск будет выполнен только глобально, в каталогах, которые записаны в переменную `PATH` и даже если вы находитесь сейчас в том же каталоге где находится скрипт, то он не будет найден. К нему нужно указывать полный путь, например, для того же текущего каталога.

Запуск скрипта `sh` в **GNU\Linux**:

```
# ./script.sh
```

Ответ:



Перед названием скрипта надо ставить точку и слэш `./`, сделано для усложнения работы вирусов.

Точка означает «текущий каталог», а слэш «/» – «разделитель между именем каталога и именем скрипта».

Или полный путь от корня:

```
# /root/script.sh
```

Ответ:



Если вы не хотите писать полный путь к скрипту, это можно сделать, достаточно переместить скрипт в один из каталогов, которые указаны в переменной PATH. Одна из них, которая предназначена для ручной установки программ – /usr/local/bin.

```
# ln -s /root/script.sh /usr/local/bin/script.sh
```

Теперь вы можете выполнить запуск скрипта просто с упоминания названия, на которое ссылается на файл скрипта:

```
# cd /home/  
# script.sh
```

Ответ:



Это был первый способ вызвать скрипт, но есть еще один – мы можем запустить оболочку и сразу же передать ей скрипт, который нужно выполнить. Вы могли редко видеть такой способ с **bash**, но он довольно часто используется для скриптов **php** или **python**.

Другой вариант – запускать сам **bash** с передачей ему файла как аргумент, потому что бинарник **bash**'а уже с флагом исполняемости **executable**.

```
# bash script.sh
```

Ответ:



А если нам нужно запустить скрипт на **php**, то выполните:

```
# php script.php
```

Запуск скрипта в фоновом режиме.

Так можно запустить скрипт как фоновый процесс, используйте символ **&**:

```
# script.sh &
```

Ответ:



Выйти из программы можно как с помощью **Ctrl + C**, так и **Ctrl + D**.

Ctrl + C в **GNU/Linux** используется для прерывания текущего выполняющегося процесса в терминале. Если какая-либо команда начинает вести себя не так, как ожидается, или нужно прервать выполнение, достаточно нажать **Ctrl + C**.

Ctrl + D используется для выхода из терминала или завершения ввода данных. Когда пользователь находится в процессе ввода команд или текста, нажатие **Ctrl + D** сигнализирует терминалу о том, что ввод завершён. Если терминал пустой, это приведёт к завершению текущей сессии.

Простой скрипт.

Допустим, мы хотим создать скрипт в **GNU/Linux**, который приветствует пользователя и выводит текущую дату и время на экран.

```
# cd ~  
# mcedit hello_user_date.sh
```

Содержимое скрипта `script.sh` получается таким:

```
#!/bin/bash  
echo "Hello, $USER!"  
echo "Today is $(date)"
```

Первая строка указывает на то, что это скрипт на **bash**. Следующая строка `echo "Hello, $USER!"` выводит приветствие с именем текущего пользователя. `$USER` является системной переменной, которая содержит **имя текущего пользователя**. Третья строка `echo "Today is $(date)"` выводит текущую дату и время. `$(date)` используется для вызова команды `date`, которая возвращает текущую дату и время в формате, указанном в настройках системы.

Запустим скрипт:

```
# bash hello_user_date.sh
```

Ответ:

```
root@pbs:~# bash hello_user_date.sh  
Hello, hamster!  
Today is Fri Jul  4 00:07:58 +05 2025  
root@pbs:~#
```

Параметры командной строки.

Параметры командной строки позволяют передавать аргументы в скрипты GNU\Linux при его запуске.

Параметры командной строки могут быть доступны в скрипте как \$1, \$2, \$3 и так далее, где \$1 – это первый параметр, \$2 – второй параметр и так далее.

Перепишем скрипт для приветствия пользователя:

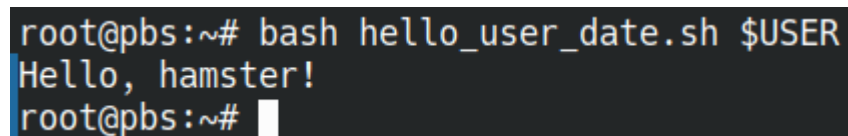
```
# cd ~  
# mcedit hello_user_date.sh
```

```
#!/bin/bash  
echo "Hello, $1!"
```

Запустим скрипт, передав ему аргумент \$USER:

```
# bash hello_user_date.sh $USER
```

Ответ:



```
root@pbs:~# bash hello_user_date.sh $USER  
Hello, hamster!  
root@pbs:~#
```

Кроме того, возможно использовать специальные параметры командной строки:

- \$0 – имя скрипта (то есть, имя файла, который был запущен);
- \$# – количество переданных параметров;
- \$* или @\$ – список всех переданных параметров (в виде одной строки или массива соответственно);
- \$? – код возврата последней выполненной команды.

Например, чтобы вывести на экран количество переданных параметров, можно использовать следующий код:

```
# cd ~  
# mcedit hello_user_date.sh  
  
#!/bin/bash  
echo "Hello, " $1 "!"  
echo "Number of passed parameters: $#"
```


Запустим скрипт, передав ему аргумент \$USER:

```
# bash hello_user_date.sh $USER
```

Ответ:

```
root@pbs:~# bash hello_user_date.sh $USER
Hello, hamster !
Number of passed parameters: 1
root@pbs:~#
```

Переменные.

Присвоить переменную.

Переменные в **bash** используются для хранения данных, таких как строки и числа. Они могут быть определены явно, путем присвоения значения, или неявно, путем автоматического присвоения значения при выполнении определенных операций. Для создания переменной в **bash** необходимо присвоить ей значение, используя знак равенства =.

Например:

```
company="Timeweb"
```

Внимание! Обратите внимание, что не должно быть пробелов между именем переменной, знаком равенства и значением.

Значение переменной можно получить, указав ее имя после команды echo и знака \$.

Например:

```
echo $company
```

Также возможно присвоить значение переменной через ввод с клавиатуры с помощью команды read.

Запросить переменную.

Например, следующий скрипт запрашивает у пользователя имя и сохраняет его в переменной.

```
# cd ~
```

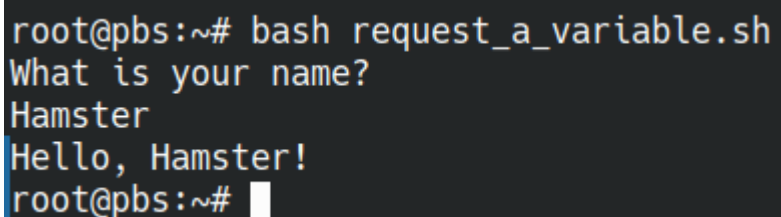
```
# mcedit request_a_variable.sh
```

```
#!/bin/bash
echo "What is your name?"
read name
echo "Hello, $name!"
```

Запустим скрипт:

```
# bash request_a_variable.sh
```

Ответ:

A terminal window with a dark background. The prompt is 'root@pbs:~#'. The user enters 'bash request_a_variable.sh'. The script outputs 'What is your name?'. The user enters 'Hamster'. The script outputs 'Hello, Hamster!'. The prompt returns to 'root@pbs:~#'.

```
root@pbs:~# bash request_a_variable.sh
What is your name?
Hamster
Hello, Hamster!
root@pbs:~#
```

Специальные переменные.

В **bash** доступны несколько специальных переменных, которые автоматически определяются и заполняются системой.

Например, переменная `$HOME` содержит путь к домашнему каталогу пользователя, а `$PWD` содержит путь к текущему рабочему каталогу.

Кроме этого, доступны переменные окружения, которые определяются системой и могут использоваться в скриптах.

Например, `$PATH` содержит список каталогов, в которых **bash** ищет исполняемые файлы.

Переменные также могут использоваться для передачи значений между различными командами и скриптами.

Например, для передачи значения переменной из одного скрипта в другой, необходимо использовать команду `export`:

```
export название_переменной
```

Подробнее про автодополнение `export` описано в разделе моего сайта: [«Автодополнение — Файл ~/.bashrc»](#).

Условные операторы.

Условные операторы — это конструкции, которые позволяют выполнить определенный набор действий в зависимости от истинности или ложности какого-то условия. В **bash-скриптах** условия записываются в скобках и передаются в команду `if`.

Синтаксис оператора `if` выглядит следующим образом:

```
if [ условие ]
then
    команды, которые нужно выполнить, если условие верно
fi
```

Здесь в квадратных скобках указывается условие, которое необходимо проверить. Если условие истинно, то будут выполнены команды, указанные между `then` и `fi`.

Например, напишем скрипт в **GNU/Linux** с названием `chetnechet.sh`, который будет проверять, является ли число, введенное пользователем, четным или нечетным.

```
# cd ~
# mcedit chetnechet.sh

#!/bin/bash
echo "Введите число: "
read n
if (( $n % 2 == 0 ))
then
    echo "Число $n четное"
else
    echo "Число $n нечетное"
fi
```

В этом примере мы использовали оператор `%`, который вычисляет остаток от деления на 2. Если остаток равен 0, то число четное, иначе — нечетное.

Запустим скрипт:

```
# bash chetnechet.sh
```

Ответы:

```
root@pbs:~# bash chetnechet.sh
Введите число:
7
Число 7 нечетное
root@pbs:~#
```

```
root@pbs:~# bash chetnechet.sh
Введите число:
4
Число 4 четное
root@pbs:~#
```

Операторы сравнения.

Кроме того, существует несколько операторов сравнения, которые можно использовать в условных конструкциях:

- `-eq` — равно;
- `-ne` — не равно;
- `-gt` — больше;
- `-lt` — меньше;
- `-ge` — больше или равно;
- `-le` — меньше или равно.

Например, чтобы проверить, является ли переменная `$a` больше переменной `$b`, можно написать следующее:

```
if [ $a -gt $b ]
then
    echo "$a больше $b"
fi
```

Важно помнить, что в условных конструкциях необходимо использовать пробелы вокруг операторов сравнения. Если пробелов нет, то **bash-скрипт** будет считать это одной большой строкой, а не операцией сравнения.

Конструкция Case.

Кроме `if`, в **bash-скриптах** также используется конструкция `case`. Эта конструкция позволяет проверять значение переменной на соответствие нескольким вариантам.

Конструкция `case` в **bash-скриптах** позволяет упростить написание условных операторов для сравнения переменных с несколькими

возможными значениями.

Синтаксис конструкции case выглядит следующим образом:

```
case variable in
    pattern1)
        command1
        ;;
    pattern2)
        command2
        ;;
    pattern3)
        command3
        ;;
    *)
        default command
        ;;
esac
```

- variable – это переменная, которую нужно проверить;
- pattern1, pattern2, pattern3 – это возможные значения, которые нужно проверить;
- command1, command2, command3 – это команды, которые нужно выполнить в зависимости от значения переменной.

Символ * в конце списка значений используется как обработчик по умолчанию, если ни одно из значений не соответствует переменной.

Скрипт, который проверяет день недели и выполняет соответствующее действие:

```
# cd ~
# mcedit case_day.sh
```

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

```
day=$(date +%u)
```

```
case $day in
    1)
        echo "Сегодня понедельник"
        ;;
    2)
        echo "Сегодня вторник"
        ;;
    3)
        echo "Сегодня среда"
```

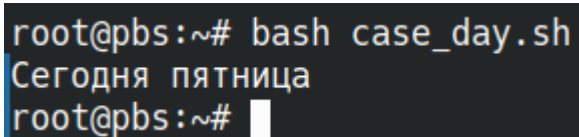
```
;;
4)
    echo "Сегодня четверг"
    ;;
5)
    echo "Сегодня пятница"
    ;;
6)
    echo "Сегодня суббота"
    ;;
7)
    echo "Сегодня воскресенье"
    ;;
*)
    echo "Некорректное значение дня недели"
    ;;
esac
```

В этом примере мы используем переменную `day`, которую мы определяем с помощью команды `date +%u`. В данном случае, `%u` используется для получения числового значения дня недели от 1 (понедельник) до 7 (воскресенье). Затем мы сравниваем эту переменную с днями недели, используя конструкцию `case`. Если ее значение соответствует определенному значению дня недели, то мы выводим соответствующее сообщение. Если значение не соответствует перечисленным дням недели, мы выводим сообщение об ошибке.

Запустим скрипт:

```
# bash case_day.sh
```

Ответ:

A terminal window with a dark background. The prompt is 'root@pbs:~#'. The user enters 'bash case_day.sh'. The output is 'Сегодня пятница'. The prompt returns to 'root@pbs:~#'.

```
root@pbs:~# bash case_day.sh
Сегодня пятница
root@pbs:~#
```

Циклы.

Циклы в **bash** используются для выполнения повторяющихся действий.

Существуют два типа циклов: цикл `for` и цикл `while`.

Цикл for.

Цикл `for` используется для выполнения команд для каждого элемента из списка.

Синтаксис цикла `for` выглядит следующим образом:

```
for переменная in список
do
    команды
done
```

Здесь переменная принимает значение элемента из списка, после чего для каждого из них выполняются команды, указанные между `do` и `done`.

Пример:

```
# cd ~
# mcedit cycle_for.sh
```

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

```
for i in {1..10}; do
    echo "Number: $i"
done
```

В этом примере `i` принимает значения от 1 до 10, и для каждого из них будет выполнена команда `echo "Number: $i"`.

Запустим:

```
# bash cycle_for.sh
```

Ответ:

```
root@pbs:~# bash cycle_for.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Number: 10
root@pbs:~#
```

Цикл while.

Следующий цикл while используется для выполнения команд до тех пор, пока условие не станет ложным.

Синтаксис цикла while выглядит следующим образом:

```
while [ условие ]
do
    команды
done
```

Здесь в квадратных скобках указывается условие, которое проверяется перед каждой итерацией цикла.

Команды, указанные между do и done, будут выполнены до тех пор, пока условие остается истинным.

Пример:

```
# cd ~
# mcedit cycle_while.sh

#!/bin/bash

count=1
while [ $count -le 10 ]; do
    echo "Count: $count"
    count=$((count+1))
done
```

В этом примере count увеличивается на 1 после каждой итерации цикла.

Когда значение `count` достигает 10, цикл завершается.

Запустим:

```
# bash cycle_while.sh
```

Ответ:

```
root@pbs:~# bash cycle_while.sh
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Count: 6
Count: 7
Count: 8
Count: 9
Count: 10
root@pbs:~#
```

Функции.

Функции в **bash** используются для группировки команд в логически связанные блоки. Функции могут быть вызваны из скрипта с помощью имени функции.

Синтаксис функции выглядит следующим образом:

```
название_функции () {
    команды_и_выражения
}
```

- Название функции должно начинаться с буквы или символа подчеркивания и может содержать только буквы, цифры и символы подчеркивания.
- За названием функции следует список аргументов в скобках.
- Команды и выражения, которые должны быть выполнены при вызове функции, должны быть внутри фигурных скобок.

Имя функции, вызов функции.

Вот пример функции, которая выводит текущее время и дату:

```
# cd ~
```

```
# mcedit today_date.sh
```

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

```
print_date () {  
    echo "Сегодняшняя дата: $(date)"  
}
```

```
print_date #Вызов функции
```

Запустим:

```
# bash today_date.sh
```

Ответ:

```
root@pbs:~# bash today_date.sh  
Сегодняшняя дата: Пт 04 июл 2025 01:57:11 +05  
root@pbs:~#
```

Передача аргументов в функцию.

Функции также могут иметь аргументы, которые передаются в качестве параметров внутри скобок при их вызове.

Вот пример функции, которая принимает два аргумента и выводит их сумму:

```
# cd ~
```

```
# mcedit sum_of_arguments.sh
```

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

```
sum_numbers () {  
    result=$(( $1 + $2 ))  
    echo "Сумма чисел $1 и $2 равна $result"  
}
```

```
sum_numbers 10 20 #Вызов функции
```

- \$1 и \$2 – это переменные, которые содержат значения первого и второго аргументов соответственно;
- sum_numbers 10 20 вызовет функцию sum_numbers с аргументами 10 и 20 и выведет результат.

Запустим:

```
# bash sum_of_arguments.sh
```

Ответ:

```
root@pbs:~# bash sum_of_arguments.sh
Сумма чисел 10 и 20 равна 30
root@pbs:~#
```

Возврат результата из функции.

Функции также могут возвращать значения при помощи ключевого слова `return`.

```
# cd ~
# mcedit return_values.sh
```

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

```
sum_numbers () {
    result=$(( $1 + $2 ))
    return $result
}
```

```
sum_numbers 12 24 #Вызов функции
echo "Сумма чисел равна $?" #Вывод
```

Здесь результат сохраняется в переменную `result` и возвращается из функции с помощью команды `return`.

Переменная `$?` содержит код возврата функции, то есть в данном случае — результат вычисления суммы.

Запустим:

```
# bash return_values.sh
```

Ответ:

```
root@pbs:~# bash return_values.sh
Сумма чисел равна 36
root@pbs:~#
```

Сохранение результата вызова функции в

переменную.

Есть и другой способ взаимодействия с результатом вызова функции, помимо return.

```
# cd ~
# mcedit other_return_values.sh

#!/bin/bash

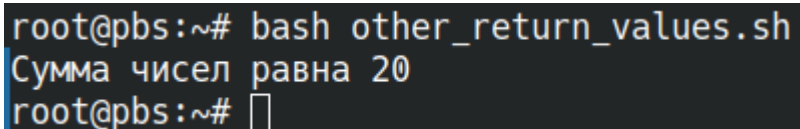
sum_numbers () {
    result=$(( $1 + $2 ))
    echo $result
}
sum=$(sum_numbers 9 11)
echo "Сумма чисел равна $sum" #Вывод
```

Здесь, вместо использования \$? и return, мы сохраняем результат вызова функции в переменную sum и затем выводим ее значение на экран.

Запустим:

```
# bash other_return_values.sh
```

Ответ:



```
root@pbs:~# bash other_return_values.sh
Сумма чисел равна 20
root@pbs:~#
```

Работа с файлами и каталогами.

Bash-скрипты могут использоваться для выполнения различных операций с файлами и каталогам в **GNU/Linux**.

Например, чтобы проверить существование файла, можно использовать команду:

```
test -e filename
```

или

```
test -e filename && echo "Файл существует" || echo "Файл не"
```

существует"

Если файл существует, то команда вернет значение 0, в противном случае – ненулевое значение.

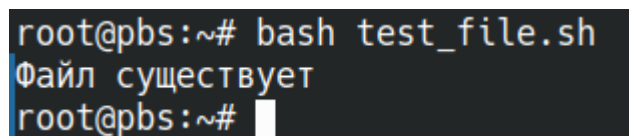
```
# cd ~
# mcedit test_file.sh

if test -e /root/test_file.sh; then
    echo "Файл существует"
else
    echo "Файл не существует"
fi
```

Запустим:

```
# bash test_file.sh
```

Ответ:

A terminal window with a dark background. The prompt is root@pbs:~#. The user enters 'bash test_file.sh'. The output is 'Файл существует'. The prompt returns to root@pbs:~#.

```
root@pbs:~# bash test_file.sh
Файл существует
root@pbs:~#
```

Для работы с каталогами в **bash-скриптах** можно использовать команды **cd**, **mkdir**, **rmdir**, **ls** и так далее.

Отладка скриптов.

Отладка **bash-скриптов** может оказаться трудной задачей, поскольку проблемы могут быть вызваны различными факторами, такими как ошибки синтаксиса, неправильное использование переменных или функций и так далее.

Для отладки **bash-скриптов** можно использовать такие инструменты, как **set -x**, **set -v** и **set -e**.

Команда **set -x** позволяет выводить на экран команды перед их выполнением, команда **set -v** выводит на экран значения переменных перед их использованием, а **set -e** прерывает выполнение скрипта в случае ошибки.

Пример использования **set -x**.

Команда **set -x** включает режим отладки, который выводит каждую команду перед её выполнением.

```
# cd ~
# mcedit set_x.sh

#!/bin/bash

set -x # Включаем отладку

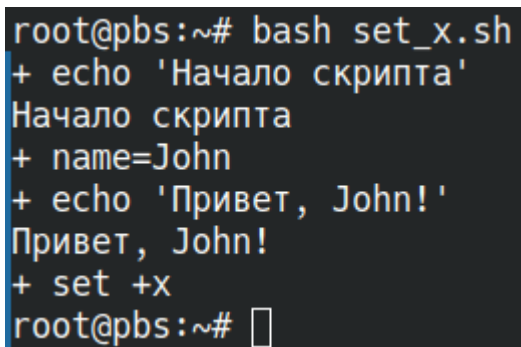
echo "Начало скрипта"
name="John"
echo "Привет, $name!"

set +x # Выключаем отладку
```

Запустим:

```
# bash set_x.sh
```

Ответ:



```
root@pbs:~# bash set_x.sh
+ echo 'Начало скрипта'
Начало скрипта
+ name=John
+ echo 'Привет, John!'
Привет, John!
+ set +x
root@pbs:~#
```

Пример использования set -v.

Команда `set -v` выводит строки скрипта по мере их чтения интерпретатором.

```
# cd ~
# mcedit set_v.sh

#!/bin/bash

set -v # Включаем вывод строк

echo "Начало скрипта"
name="John"
echo "Привет, $name!"

set +v # Выключаем вывод строк
```

Запустим:

```
# bash set_v.sh
```

Ответ:

```
root@pbs:~# bash set_v.sh
echo "Начало скрипта"
Начало скрипта
name="John"
echo "Привет, $name!"
Привет, John!

set +v # Выключаем вывод строк
root@pbs:~#
```

Пример использования set -e.

Команда set -e прерывает выполнение скрипта, если какая-либо команда завершится с ошибкой (ненулевым кодом возврата).

```
# cd ~
```

```
# mcedit set_e.sh
```

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

```
set -e # Прерываем выполнение при ошибке
```

```
echo "Начало скрипта"
```

```
ls /несуществующий_каталог # Эта команда завершится с ошибкой
```

```
echo "Эта строка не будет выполнена"
```

В этом примере выполнение скрипта прерывается после команды ls, так как она завершилась с ошибкой, и последняя строка echo не выполняется.

Запустим:

```
# bash set_e.sh
```

Ответ:

```
root@pbs:~# bash set_e.sh
Начало скрипта
ls: cannot access '/несуществующий_каталог': No such file or directory
root@pbs:~#
```

Комбинирование команд.

Вы можете комбинировать эти команды для более детальной отладки:

```
# cd ~  
# mcedit ko-ko-kombo.sh
```

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

```
set -exv # Включаем отладку, прерывание при ошибке и вывод строк
```

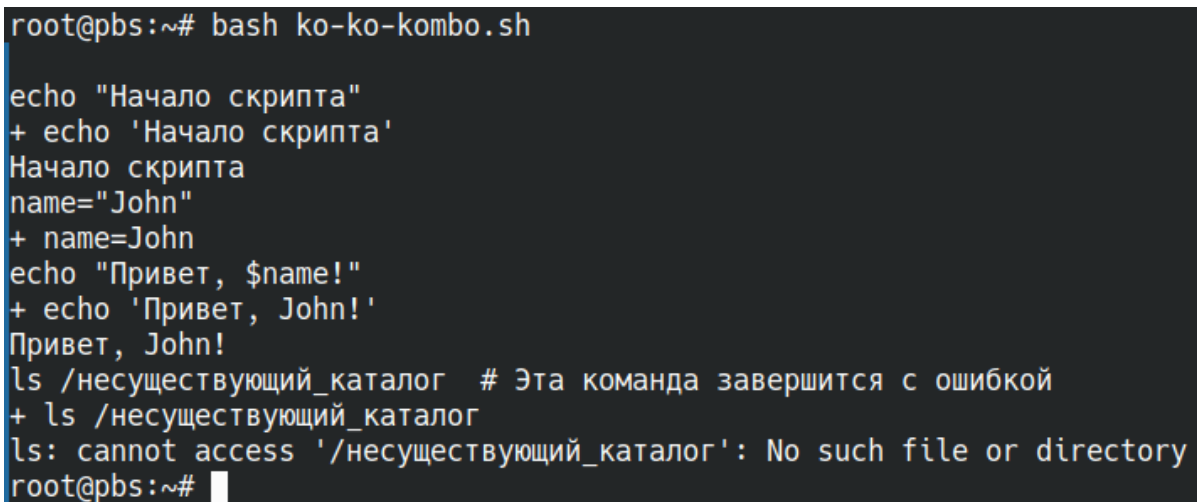
```
echo "Начало скрипта"  
name="John"  
echo "Привет, $name!"  
ls /несуществующий_каталог # Эта команда завершится с ошибкой  
echo "Эта строка не будет выполнена"
```

В этом примере скрипт прерывается после команды `ls`, и последняя строка `echo` не выполняется.

Запустим:

```
# bash ko-ko-kombo.sh
```

Ответ:



```
root@pbs:~# bash ko-ko-kombo.sh  
echo "Начало скрипта"  
+ echo 'Начало скрипта'  
Начало скрипта  
name="John"  
+ name=John  
echo "Привет, $name!"  
+ echo 'Привет, John!'  
Привет, John!  
ls /несуществующий_каталог # Эта команда завершится с ошибкой  
+ ls /несуществующий_каталог  
ls: cannot access '/несуществующий_каталог': No such file or directory  
root@pbs:~#
```

Эти команды помогают эффективно отлаживать **bash-скрипты**, выявляя проблемы на ранних этапах.

Оригиналы источников информации.

1. qna.habr.com «Как запустить sh скрипты в Linux?»
2. superuser.com «What is the difference between bash and sh?»
3. timeweb.cloud «Инструкция по написанию скриптов в Linux Bash».