



984,18

Рейтинг

RUVDS.com

RUVDS – хостинг VDS/VPS серверов



ru_vds 3 апреля 2017 в 14:38

Bash-скрипты: начало

<https://likegeeks.com/bash-script-easy-guide/>

Серверное администрирование, Настройка Linux, Блог компании RUVDS.com

Перевод

Bash-скрипты: начало

Bash-скрипты, часть 2: циклы

Bash-скрипты, часть 3: параметры и ключи командной строки

Bash-скрипты, часть 4: ввод и вывод

Bash-скрипты, часть 5: сигналы, фоновые задачи, управление сценариями

Bash-скрипты, часть 6: функции и разработка библиотек

Bash-скрипты, часть 7: sed и обработка текстов

Bash-скрипты, часть 8: язык обработки данных awk

Bash-скрипты, часть 9: регулярные выражения

Bash-скрипты, часть 10: практические примеры

Bash-скрипты, часть 11: expect и автоматизация интерактивных утилит

Сегодня поговорим о bash-скриптах. Это — [сценарии командной строки](#), написанные для оболочки bash. Существуют и другие оболочки, например — zsh, tcsh, ksh, но мы сосредоточимся на bash. Этот материал предназначен для всех желающих, единственное условие — умение работать в командной строке Linux.



Сценарии командной строки — это наборы тех же самых команд, которые можно вводить с клавиатуры, собранные в файлы и объединённые некоей общей целью. При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. Сценарии — это мощный способ автоматизации часто выполняемых действий.

Habrahabr10

Промо-код для скидки в 10% на наши виртуальные сервера

Итак, если говорить о командной строке, она позволяет выполнить несколько команд за один раз, введя их через точку с запятой:

```
pwd ; whoami
```

На самом деле, если вы опробовали это в своём терминале, ваш первый bash-скрипт, в котором задействованы две команды, уже написан. Работает он так. Сначала команда `pwd` выводит на экран сведения о текущей рабочей директории, потом команда `whoami` показывает данные о пользователе, под которым вы вошли в систему.

Используя подобный подход, вы можете совмещать сколько угодно 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
```

Командная строка — отличный инструмент, но команды в неё приходится вводить каждый раз, когда в них возникает необходимость. Что если записать набор команд в файл и просто вызывать этот файл для их выполнения? Собственно говоря, тот файл, о котором мы говорим, и называется сценарием командной строки.

Как устроены bash-скрипты

Создайте пустой файл с использованием команды `touch`. В его первой строке нужно указать, какую именно оболочку мы собираемся использовать. Нас интересует `bash`, поэтому первая строка файла будет такой:

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

В других строках этого файла символ решётки используется для обозначения комментариев, которые оболочка не обрабатывает. Однако, первая строка — это особый случай, здесь решётка, за которой следует восклицательный знак (эту последовательность называют **шебанг**) и путь к `bash`, указывают системе на то, что сценарий создан именно для `bash`.

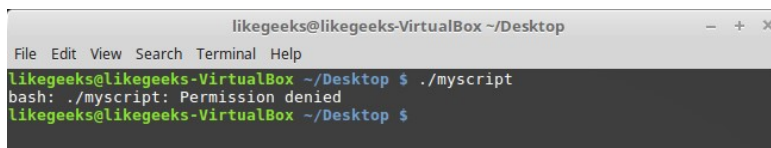
Команды оболочки отделяются знаком перевода строки, комментарии выделяют знаком решётки. Вот как это выглядит:

```
#!/bin/bash
# This is a comment
pwd
whoami
```

Тут, так же, как и в командной строке, можно записывать команды в одной строке, разделяя точкой с запятой. Однако, если писать команды на разных строках, файл легче читать. В любом случае оболочка их обработает.

Установка разрешений для файла сценария

Сохраните файл, дав ему имя `myscript`, и работа по созданию `bash`-скрипта почти закончена. Сейчас осталось лишь сделать этот файл исполняемым, иначе, попытавшись его запустить, вы столкнётесь с ошибкой `Permission denied`.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
bash: ./myscript: Permission denied
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Попытка запуска файла сценария с неправильно настроенными разрешениями

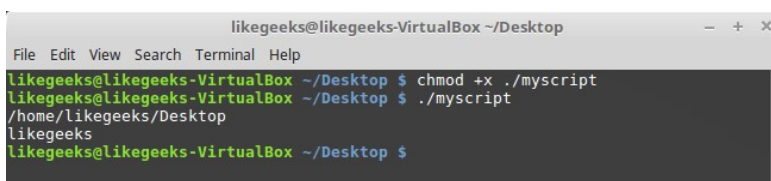
Сделаем файл исполняемым:

```
chmod +x ./myscript
```

Теперь попытаемся его выполнить:

```
./myscript
```

После настройки разрешений всё работает как надо.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ chmod +x ./myscript
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
/home/likegeeks/Desktop
likegeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

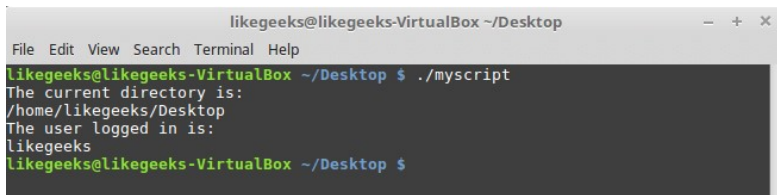
Успешный запуск `bash`-скрипта

Вывод сообщений

Для вывода текста в консоль Linux применяется команда `echo`. Воспользуемся знанием этого факта и отредактируем наш скрипт, добавив пояснения к данным, которые выводят уже имеющиеся в нём команды:

```
#!/bin/bash
# our comment is here
echo "The current directory is:"
pwd
echo "The user logged in is:"
whoami
```

Вот что получится после запуска обновлённого скрипта.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script named 'myscript'. The output of the script is: 'The current directory is: /home/likegeeks/Desktop' and 'The user logged in is: likegeeks'. The prompt returns to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

Вывод сообщений из скрипта

Теперь мы можем выводить поясняющие надписи, используя команду `echo`. Если вы не знаете, как отредактировать файл, пользуясь средствами Linux, или раньше не встречались с командой `echo`, взгляните на [этот](#) материал.

Использование переменных

Переменные позволяют хранить в файле сценария информацию, например — результаты работы команд для использования их другими командами.

Нет ничего плохого в исполнении отдельных команд без хранения результатов их работы, но возможности такого подхода весьма ограничены.

Существуют два типа переменных, которые можно использовать в bash-скриптах:

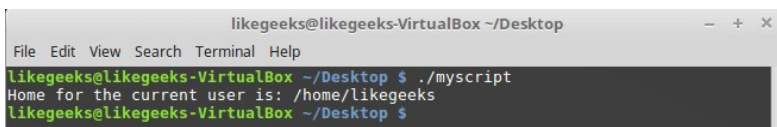
- Переменные среды
- Пользовательские переменные

Переменные среды

Иногда в командах оболочки нужно работать с некими системными данными. Вот, например, как вывести домашнюю директорию текущего пользователя:

```
#!/bin/bash
# display user home
echo "Home for the current user is: $HOME"
```

Обратите внимание на то, что мы можем использовать системную переменную `$HOME` в двойных кавычках, это не мешает системе её распознать. Вот что получится, если выполнить вышеприведённый сценарий.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script named 'myscript'. The output of the script is: 'Home for the current user is: /home/likegeeks'. The prompt returns to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

Использование переменной среды в сценарии

А что если надо вывести на экран значок доллара? Попробуем так:

```
echo "I have $1 in my pocket"
```

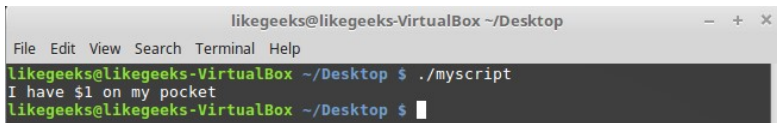
Система обнаружит знак доллара в строке, ограниченной кавычками, и решит, что мы сослались на переменную. Скрипт попытается вывести на

экран значение неопределённой переменной \$1. Это не то, что нам нужно. Что делать?

В подобной ситуации поможет использование управляющего символа, обратной косой черты, перед знаком доллара:

```
echo "I have \$1 in my pocket"
```

Теперь сценарий выведет именно то, что ожидается.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
I have $1 on my pocket
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Использование управляющей последовательности для вывода знака доллара

Пользовательские переменные

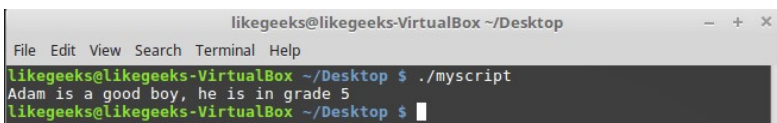
В дополнение к переменным среды, bash-скрипты позволяют задавать и использовать в сценарии собственные переменные. Подобные переменные хранят значение до тех пор, пока не завершится выполнение сценария.

Как и в случае с системными переменными, к пользовательским переменным можно обращаться, используя знак доллара:

TNW-CUS-FMP — промо-код на 10% скидку на наши услуги, доступен для активации в течение 7 дней

```
#!/bin/bash
# testing variables
grade=5
person="Adam"
echo "$person is a good boy, he is in grade $grade"
```

Вот что получится после запуска такого сценария.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Adam is a good boy, he is in grade 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Пользовательские переменные в сценарии

Подстановка команд

Одна из самых полезных возможностей bash-скриптов — это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария.

Сделать это можно двумя способами.

- С помощью значка обратного апострофа «`»
- С помощью конструкции \$()

Используя первый подход, проследите за тем, чтобы вместо обратного апострофа не ввести одиночную кавычку. Команду нужно заключить в два таких значка:

```
mydir=`pwd`
```

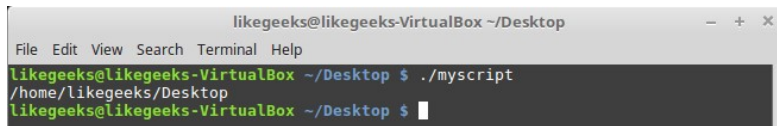
При втором подходе то же самое записывают так:

```
mydir=$(pwd)
```

А скрипт, в итоге, может выглядеть так:

```
#!/bin/bash
mydir=$(pwd)
echo $mydir
```

В ходе его работы вывод команды `pwd` будет сохранён в переменной `mydir`, содержимое которой, с помощью команды `echo`, попадёт в консоль.



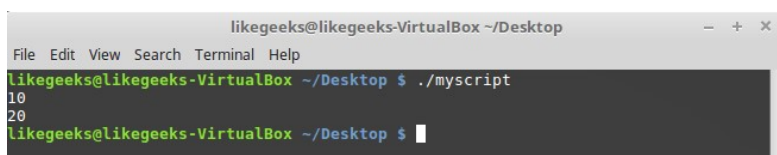
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
/home/likegeeks/Desktop
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Скрипт, сохраняющий результаты работы команды в переменной

Математические операции

Для выполнения математических операций в файле скрипта можно использовать конструкцию вида `$((a+b))`:

```
#!/bin/bash
var1=$(( 5 + 5 ))
echo $var1
var2=$(( $var1 * 2 ))
echo $var2
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
10
20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Математические операции в сценарии

Управляющая конструкция if-then

В некоторых сценариях требуется управлять потоком исполнения команд. Например, если некое значение больше пяти, нужно выполнить одно действие, в противном случае — другое. Подобное применимо в очень многих ситуациях, и здесь нам поможет управляющая конструкция `if-then`. В наиболее простом виде она выглядит так:

```
if команда
then
команды
fi
```

А вот рабочий пример:

```
#!/bin/bash
if pwd
then
echo "It works"
fi
```

В данном случае, если выполнение команды `pwd` завершится успешно, в консоль будет выведен текст «it works».

Воспользуемся имеющимися у нас знаниями и напишем более сложный сценарий. Скажем, надо найти некоего пользователя в `/etc/passwd`, и если найти его удалось, сообщить о том, что он существует.

```
#!/bin/bash
user=likegeeks
if grep $user /etc/passwd
then
echo "The user $user Exists"
fi
```

Вот что получается после запуска этого скрипта.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
likegeeks:x:1000:1000:likegeeks,,,:/home/likegeeks:/bin/bash
The user likegeeks Exists
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Поиск пользователя

Здесь мы воспользовались командой `grep` для поиска пользователя в файле `/etc/passwd`. Если команда `grep` вам незнакома, её описание можно найти [здесь](#).

В этом примере, если пользователь найден, скрипт выведет соответствующее сообщение. А если найти пользователя не удалось? В данном случае скрипт просто завершит выполнение, ничего нам не сообщив. Хотелось бы, чтобы он сказал нам и об этом, поэтому усовершенствуем код.

Управляющая конструкция `if-then-else`

Для того, чтобы программа смогла сообщить и о результатах успешного поиска, и о неудаче, воспользуемся конструкцией `if-then-else`. Вот как она устроена:

```
if команда
then
команды
else
команды
fi
```

Если первая команда возвратит ноль, что означает её успешное выполнение, условие окажется истинным и выполнение не пойдёт по ветке `else`. В противном случае, если будет возвращено что-то, отличающееся от нуля, что будет означать неудачу, или ложный результат, будут выполнены команды, расположенные после `else`.

Напишем такой скрипт:

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
else
echo "The user $user doesn't exist"
fi
```

Его исполнение пошло по ветке `else`.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The user anotherUser doesn't exist
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Запуск скрипта с конструкцией `if-then-else`

Ну что же, продолжаем двигаться дальше и зададимся вопросом о более сложных условиях. Что если надо проверить не одно условие, а несколько? Например, если нужный пользователь найден, надо вывести одно сообщение, если выполняется ещё какое-то условие — ещё одно сообщение, и так далее. В подобной ситуации нам помогут вложенные условия. Выглядит это так:

```
if команда1
then
команды
elif команда2
then
команды
fi
```

Если первая команда вернёт ноль, что говорит о её успешном выполнении, выполнятся команды в первом блоке `then`, иначе, если первое условие окажется ложным, и если вторая команда вернёт ноль, выполнится второй блок кода.

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
elif ls /home
then
echo "The user doesn't exist but anyway there is a directory under /home"
fi
```

В подобном скрипте можно, например, создавать нового пользователя с помощью команды `useradd`, если поиск не дал результатов, или делать ещё что-нибудь полезное.

Сравнение чисел

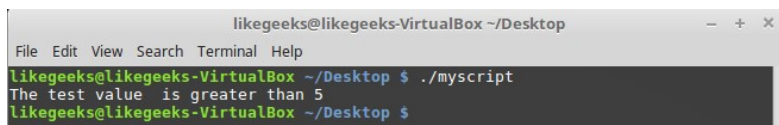
В скриптах можно сравнивать числовые значения. Ниже приведён список соответствующих команд.

```
n1 -eq n2 Возвращает истинное значение, если n1 равно n2.
n1 -ge n2 Возвращает истинное значение, если n1 больше или равно n2.
n1 -gt n2 Возвращает истинное значение, если n1 больше n2.
n1 -le n2 Возвращает истинное значение, если n1 меньше или равно n2.
n1 -lt n2 Возвращает истинное значение, если n1 меньше n2.
n1 -ne n2 Возвращает истинное значение, если n1 не равно n2.
```

В качестве примера опробуем один из операторов сравнения. Обратите внимание на то, что выражение заключено в квадратные скобки.

```
#!/bin/bash
vall=6
if [ $vall -gt 5 ]
then
echo "The test value $vall is greater than 5"
else
echo "The test value $vall is not greater than 5"
fi
```

Вот что выведет эта команда.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The test value is greater than 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Сравнение чисел в скриптах

Значение переменной `vall` больше чем 5, в итоге выполняется ветвь `then` оператора сравнения и в консоль выводится соответствующее сообщение.

Сравнение строк

В сценариях можно сравнивать и строковые значения. Операторы сравнения выглядят довольно просто, однако у операций сравнения строк есть определённые особенности, которых мы коснёмся ниже. Вот список операторов.

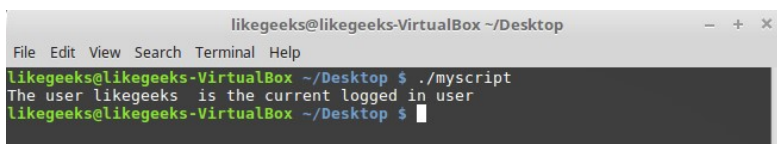
```
str1 = str2 Проверяет строки на равенство, возвращает истину, если строки идентичны.
str1 != str2 Возвращает истину, если строки не идентичны.
str1 < str2 Возвращает истину, если str1 меньше, чем str2.
str1 > str2 Возвращает истину, если str1 больше, чем str2.
-n str1 Возвращает истину, если длина str1 больше нуля.
-z str1 Возвращает истину, если длина str1 равна нулю.
```

Вот пример сравнения строк в сценарии:

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

```
user ="likegeeks"
if [$user = $USER]
then
echo "The user $user is the current logged in user"
fi
```

В результате выполнения скрипта получим следующее.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The user likegeeks is the current logged in user
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

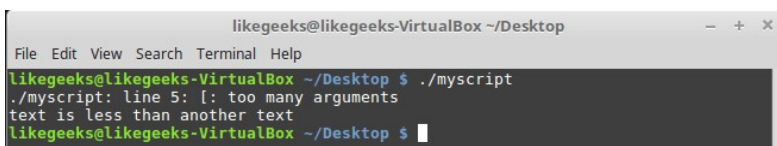
Сравнение строк в скриптах

Вот одна особенность сравнения строк, о которой стоит упомянуть. А именно, операторы «>» и «<» необходимо экранировать с помощью обратной косой черты, иначе скрипт будет работать неправильно, хотя сообщений об ошибках и не появится. Скрипт интерпретирует знак «>» как команду перенаправления вывода.

Вот как работа с этими операторами выглядит в коде:

```
#!/bin/bash
val1=text
val2="another text"
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

Вот результаты работы скрипта.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
./myscript: line 5: [: too many arguments
text is less than another text
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Сравнение строк, выведенное предупреждение

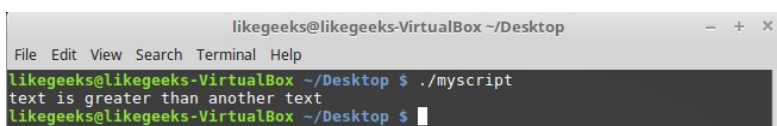
Обратите внимание на то, что скрипт, хотя и выполняется, выдаёт предупреждение:

```
./myscript: line 5: [: too many arguments
```

Для того, чтобы избавиться от этого предупреждения, заключим \$val2 в двойные кавычки:

```
#!/bin/bash
val1=text
val2="another text"
if [ $val1 \> "$val2" ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

Теперь всё работает как надо.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
text is greater than another text
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Сравнение строк

Ещё одна особенность операторов «>» и «<» заключается в том, как они работают с символами в верхнем и нижнем регистрах. Для того, чтобы понять эту особенность, подготовим текстовый файл с таким содержимым:

```
Likegeeks
likegeeks
```

Сохраним его, дав имя `myfile`, после чего выполним в терминале такую команду:

```
sort myfile
```

Она отсортирует строки из файла так:

```
likegeeks
Likegeeks
```

Команда `sort`, по умолчанию, сортирует строки по возрастанию, то есть строчная буква в нашем примере меньше прописной. Теперь подготовим скрипт, который будет сравнивать те же строки:

```
#!/bin/bash
val1=Likegeeks
val2=likegeeks
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

Если его запустить, окажется, что всё наоборот — строчная буква теперь больше прописной.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Likegeeks is less than likegeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $ sort ./myfile
likegeeks
Likegeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Команда `sort` и сравнение строк в файле сценария

В командах сравнения прописные буквы меньше строчных. Сравнение строк здесь выполняется путём сравнения ASCII-кодов символов, порядок сортировки, таким образом, зависит от кодов символов.

Команда `sort`, в свою очередь, использует порядок сортировки, заданный в настройках системного языка.

Проверки файлов

Пожалуй, нижеприведённые команды используются в `bash`-скриптах чаще всего. Они позволяют проверять различные условия, касающиеся файлов. Вот список этих команд.

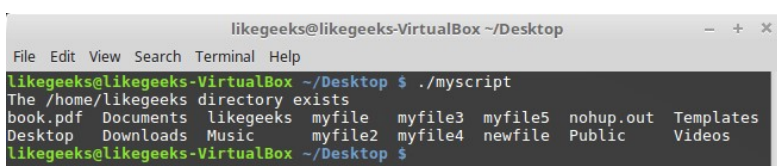
- d fileПроверяет, существует ли файл, и является ли он директорией.
- e fileПроверяет, существует ли файл.
- f fileПроверяет, существует ли файл, и является ли он файлом.
- r fileПроверяет, существует ли файл, и доступен ли он для чтения.
- s fileПроверяет, существует ли файл, и не является ли он пустым.
- w fileПроверяет, существует ли файл, и доступен ли он для записи.
- x fileПроверяет, существует ли файл, и является ли он исполняемым.
- file1 -nt file2Проверяет, новее ли file1, чем file2.
- file1 -ot file2Проверяет, старше ли file1, чем file2.
- O fileПроверяет, существует ли файл, и является ли его владельцем текущий пользователь.
- G fileПроверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Эти команды, как впрочем, и многие другие рассмотренные сегодня, несложно запомнить. Их имена, являясь сокращениями от различных слов, прямо указывают на выполняемые ими проверки.

Опробуем одну из команд на практике:

```
#!/bin/bash
mydir=/home/likegeeks
if [ -d $mydir ]
then
echo "The $mydir directory exists"
cd $ mydir
ls
else
echo "The $mydir directory does not exist"
fi
```

Этот скрипт, для существующей директории, выведет её содержимое.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The /home/likegeeks directory exists
book.pdf Documents likegeeks myfile myfile3 myfile5 nohup.out Templates
Desktop Downloads Music myfile2 myfile4 newfile Public Videos
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Вывод содержимого директории

Полагаем, с остальными командами вы сможете поэкспериментировать самостоятельно, все они применяются по тому же принципу.

Итоги

Сегодня мы рассказали о том, как приступить к написанию bash-скриптов и рассмотрели некоторые базовые вещи. На самом деле, тема bash-программирования огромна. Эта статья является переводом первой части большой серии из 11 материалов. Если вы хотите продолжения прямо сейчас — вот список оригиналов этих материалов. Для удобства сюда включён и тот, перевод которого вы только что прочли.

1. [Bash Script Step By Step](#) — здесь речь идёт о том, как начать создание bash-скриптов, рассмотрено использование переменных, описаны условные конструкции, вычисления, сравнения чисел, строк, выяснение сведений о файлах.
2. [Bash Scripting Part 2, Bash the awesome](#) — тут раскрываются особенности работы с циклами `for` и `while`.
3. [Bash Scripting Part 3, Parameters & options](#) — этот материал посвящён параметрам командной строки и ключам, которые можно передавать скриптам, работе с данными, которые вводит пользователь, и которые можно читать из файлов.
4. [Bash Scripting Part 4, Input & Output](#) — здесь речь идёт о дескрипторах файлов и о работе с ними, о потоках ввода, вывода, ошибок, о перенаправлении вывода.
5. [Bash Scripting Part 5, Signals & Jobs](#) — этот материал посвящён сигналам Linux, их обработке в скриптах, запуску сценариев по расписанию.
6. [Bash Scripting Part 6, Functions](#) — тут можно узнать о создании и использовании функций в скриптах, о разработке библиотек.
7. [Bash Scripting Part 7, Using sed](#) — эта статья посвящена работе с потоковым текстовым редактором `sed`.
8. [Bash Scripting Part 8, Using awk](#) — данный материал посвящён программированию на языке обработки данных `awk`.
9. [Bash Scripting Part 9, Regular Expressions](#) — тут можно почитать об использовании регулярных выражений в bash-скриптах.
10. [Bash Scripting Part 10, Practical Examples](#) — здесь приведены приёмы работы с сообщениями, которые можно отправлять пользователям, а так же методика мониторинга диска.
11. [Bash Scripting Part 11, Expect Command](#) — этот материал посвящён средству `Expect`, с помощью которого можно автоматизировать взаимодействие с интерактивными утилитами. В частности, здесь идёт речь об expect-скриптах и об их взаимодействии с bash-скриптами и

другими программами.

Полагаем, одно из ценных свойств этой серии статей заключается в том, что она, начинаясь с самого простого, подходящего для пользователей любого уровня, постепенно ведёт к довольно серьёзным темам, давая шанс всем желающим продвинуться в деле создания сценариев командной строки Linux.

Уважаемые читатели! Просим гуру bash-программирования рассказать о том, как они добрались до вершин мастерства, поделиться секретами, а от тех, кто только что написал свой первый скрипт, ждём впечатлений.

Habrahabr10

Промо-код для скидки в 10% на наши виртуальные сервера

Только зарегистрированные пользователи могут участвовать в опросе. Войдите, пожалуйста.

Переводить остальные части цикла статей?

- ☐ Да!
- ☐ Нет, не нужно

Проголосовали 963 пользователя. Воздержались 104 пользователя.

Метки: Linux, Bash, сценарий командной строки, Bash-скрипт

+53

1416

311k

119



RUVDS.com 984,18

RUVDS – хостинг VDS/VPS серверов



105,5

Карма

1416,2

Рейтинг

321

Подписчики

@ru_vds

Пользователь

Facebook

Twitter

Вконтакте

Google+

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

17 апреля 2017 в 12:59

Bash-скрипты, часть 4: ввод и вывод

+22

74,6k

335

14

12 апреля 2017 в 15:15

Bash-скрипты, часть 3: параметры и ключи командной строки

+35

89,9k

439

23

7 апреля 2017 в 15:58

Bash-скрипты, часть 2: циклы

+23

133k

527


35

Комментарии 119

iig 03.04.17 в 15:11

+1


С одной стороны, map bash я до конца так и не дочитал.
С другой стороны, даже в переводе его изучать не особо хочется.

 andreymal 03.04.17 в 15:15

+3


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

Позанудствую: во FreeBSD баш пихают в `/usr/local/bin/bash`, поэтому такой скрипт там не запустится. Так что теперь я пишу `#!/usr/bin/env bash` (правда, где-то читал, что это тоже где-то может не работать, но я уже забыл где)

 POS_troi 03.04.17 в 18:28


0

по хорошему симлинк делается, если сам не сделался то сделать и все проблемы исчезнут :)

 andreymal 03.04.17 в 19:34

+1

Думается мне, лезть в системные файлы, которые для лазания не предназначены, не очень хорошо

 Proxaber 03.04.17 в 21:43

-1

В Linux все системные файлы для того и нужны — чтоб было что ковырять)

Erelecano 03.04.17 в 22:42

-2

Вас обманули. В системах на базе ядра Linux системные файлы нужны, что бы все нормально работало, именно по этой причине GNU/Linux лидирует на серверном рынке(а в TOP500 просто занимает 99.9% мест). А детишкам, которые руками лезут туда, куда писать должен пакетный менеджер место на своих локалхостах. Мамкиным какирам мамкино какерство!

groaner 03.04.17 в 23:22

+1

Ага, именно поэтому видимо большинство системных настроек хранится в обычных текстовых файлах — очень уж удобно программам с ними работать. А главное, как производительно!

Erelecano 03.04.17 в 21:19

+1

Делать симлинки в /bin это по плохому. Правильно, как вам уже выше написали `#!/usr/bin/env bash`, аналогично и `#!/usr/bin/env python`, `#!/usr/bin/env perl` и так далее.

Twindo 03.04.17 в 15:54


+1

Было бы интересно почитать про развертывание приложений при помощи bash-скриптов

grieverrr 04.04.17 в 01:05


+2

а вот сейчас подленько было

 Artem_zin 04.04.17 в 03:32

0

1. Баш
2. Уводишь выполнение в докер как можно скорее
3. ???
4. Профит!

 grossws 04.04.17 в 13:00

0

```
#!/usr/bin/env bash

exec ansible-playbook -i prod site.yml
```

как-то так

A1estro 03.04.17 в 15:54

+2

Вместо

```
$((${1}+1))
```

намного читабельнее использовать

```
$(1+1)
```

```
if grep $user /etc/passwd
```

надо либо -q, либо вывод перенаправить.

iig 03.04.17 в 17:09

0

if grep \$user /etc/passwd

надо либо -q, либо вывод перенаправить

Все 3 способа неправильные ;)

myrslok 04.04.17 в 17:35

0

А какой правильный?

iig 04.04.17 в 17:43

0

В passwd есть структура из нескольких полей, поэтому искать что-то grep'ом неправильно. Либо резать на поля и искать (awk), либо взять готовую утилиту id

```
if id -u "$user" 2>&1 > /dev/null
then
```

, но это уже не поиск в passwd, а нечто большее. Зависит от того, что хотели получить.

 Hissing_Bridge 06.04.17 в 16:17

0

искать что-то grep'ом неправильно.

С чего это? Как раз правильно, просто нужно воспользоваться регуляркой


```
grep -qE "^$user:"
```

iig 06.04.17 в 16:38

+2

А это уже зависит от того, что именно нужно найти. ;)

Если строку в файле — grep, если пользователя в системе — id. Авторизация через passwd — не единственный способ.

 grossws 06.04.17 в 17:24

0

Можно ещё getent passwd, оно притаскивает из всего сконфигурированного в /etc/nsswitch.conf

 saboteur_kiev 08.04.17 в 11:28

0

Нужно!

tatalame 03.04.17 в 17:30

0

`\${1+1}


Читабельно, но deprecated, см. <http://stackoverflow.com/a/2415777/1221759>

A1estro 03.04.17 в 17:39

0

Хм. Видимо собирались выпилить, но не решились, посту уже 7 лет. Мельком посмотрел в ченжлогах — ничего не нашёл тоже. Debian Jessie. bash 4.3.30.

НЛО прилетело и опубликовало эту надпись здесь

 Ordinatus 03.04.17 в 17:44

+5


Просим гуру bash-программирования рассказать о том, как они добрались до вершин мастерства, поделиться секретами, а от тех, кто только что написал свой первый скрипт, ждём впечатлений.

Не читайте про bash на хабре, читайте man, --help и книги!

Ugrum 04.04.17 в 11:32

0

del/здесь была неудачная шутка.

 zirf 25.04.17 в 10:41

0

В общем да, есть Mendel Cooper "Advanced Bash Scripting Guide", есть кстати, и переводы, хотя и не последних версий.

lorc 03.04.17 в 19:02

0

Для меня когда-то давно было открытием, что `[` — это программа и лежит она обычно в `/usr/bin`.

Кстати, я думал что `[` — это симлинк на `test`, но в моей системе это разные программы:

```
lorc:work/ $ ls -l /usr/bin/[
-rwxr-xr-x 1 root root 51920 лют 18  2016 /usr/bin/[

lorc:work/ $ ls -l /usr/bin/test
-rwxr-xr-x 1 root root 47824 лют 18  2016 /usr/bin/test
```

khim 03.04.17 в 19:43

0

А теперь — открытие «второго порядка»: ни `/usr/bin/[`, ни `/usr/bin/test` не используются в вышеприведённых скриптах.

Нужно либо задавать полное имя, либо использовать другой shell (например `/bin/sh`), где эти команды не устроены...

lorc 03.04.17 в 19:46

0

да, прямо в `man [` написано

NOTE: your shell may have its own version of `test` and/or `[`, which usually supersedes the version described here. Please refer to your shell's documentation for details about the options it supports.



Artem_zin 03.04.17 в 19:30

+4

Очень рекомендую использовать <https://github.com/koalaman/shellcheck> для проверки шелл скриптов как часть CI, ловит как простые, так и сложные штуки, и помогает не забывать про corner кейсы на разных ОС, офигенный тул.

vmSPIKE 03.04.17 в 20:49

+1

Действительно классная штука, но всегда полагаться на её предупреждения не стоит, ибо она не достаточно проницательна, чтобы понять какое поведение тебе нужно или некоторые сложные случаи.

Кстати, в некоторых дистрибутивах она доступна в репе (например в Ubuntu: `sudo apt install shellcheck`).

Есть плагин для SublimeText: `SublimeLinter-shellcheck`



Artem_zin 04.04.17 в 03:39

0

Стоит отметить, что часто со второго-третьего раза, перечитывая описания предупреждений в их вики, таки находится более правильный вариант, который успокаивает и автора скрипта и `shellcheck` :)

Ну и у `shellcheck` есть формат комментариев для сапреса ворнингов.

А, кстати, на шелл скрипты можно и тесты писать <https://github.com/gojuno/mainframer> (см папку `test` и `travis.yml`), выглядит примерно так:

```
mainframer — artem@artems-mbp — .om/mainframer — -zsh — 130x40
artem at artems-mbp in ~/projects/gojuno.com/mainframer (tech/az/ci)
$ bash test.sh

----- TEST STARTED test_keeps_files_on_local_machine.sh -----
mainframer v1.1.2
Start time: Sat Jan 7 04:28:10 +03 2017
+ cd /Users/artem/run/
noop
+ echo noop
End time: Sat Jan 7 04:28:10 +03 2017
Whole process took 0 seconds.

----- TEST ENDED test_keeps_files_on_local_machine.sh -----

----- TEST STARTED test_moves_files_to_local_machine.sh -----
mainframer v1.1.2
Start time: Sat Jan 7 04:28:10 +03 2017
+ cd /Users/artem/run/
+ mkdir build
+ touch build/buildresult.txt
End time: Sat Jan 7 04:28:11 +03 2017
Whole process took 1 seconds.

----- TEST ENDED test_moves_files_to_local_machine.sh -----

----- TEST STARTED test_moves_files_to_remote_machine.sh -----
mainframer v1.1.2
Start time: Sat Jan 7 04:28:11 +03 2017
noop
+ cd /Users/artem/run/
+ echo noop
End time: Sat Jan 7 04:28:11 +03 2017
Whole process took 0 seconds.

----- TEST ENDED test_moves_files_to_remote_machine.sh -----

Test run SUCCESS, 3 test(s).

artem at artems-mbp in ~/projects/gojuno.com/mainframer (tech/az/ci)
$
```



greybox 03.04.17 в 19:41

+3

я просто оставлю это здесь <https://github.com/denysdovhan/bash-handbook>



overmind88 03.04.17 в 19:48

0

> Уважаемые читатели! Просим гуру bash-программирования рассказать о том, как они добрались до вершин мастерства, поделиться секретами, а от тех, кто только что написал свой первый скрипт, ждём впечатлений.

Прочитал ABS Guide. Периодически использую его как справку. Всё.

AVX 03.04.17 в 23:05

0

Соглашусь. Advanced Bash-Scripting Guide и man с интернетами для сложных случаев.

khim 04.04.17 в 19:25

0

В таком случае вам не составит труда скопировать переменную A в переменную B?

P.S. Сам факт, что в bash'e эта задача весьма нетривиальна вызывает лёгкую грусть, конечно...

iig 04.04.17 в 22:28

0

В чем нетривиальность?

khim 05.04.17 в 00:03

+1

```
declare A=([a]='x' [b]='y')
```

Копируйте.

P.S. Эта проблема в bash4 появилась. В bash2 было так:

```
$ declare -a A=('a' 'b' c 'd e')
$ B=("${A[@]}")
$ declare -p B
declare -a B=([0]="a" [1]="b" c [2]="d e")
```

Просто, логично, понятно. А теперь, в bash4, чего делать? Есть несколько решений, но красивого я не знаю... что грустно: что это за язык такой, в котором переменную скопировать — проблема?

iig 05.04.17 в 09:14

0

В C, чтобы скопировать массив, тоже надо пару раз присесть ;)

В bash операции со строками имеют особенности, а массивов, по возможности, лучше избегать. Простл запомнить это. Если нужны какие-то структуры данных — лучше взять python.

khim 05.04.17 в 16:17

0

В С, чтобы скопировать массив, тоже надо пару раз присесть ;)

То что массивы в С сделаны неудобно и криво — это всем известно. И оправданием для bash являться никак не может.

А массивы в bash нужны. Как без них параметры командной строки обрабатывать?

iig 05.04.17 в 16:29

0

```
while [ -n "$1" ]
do
    _param="$1"
    shift
    # флажки без параметров
    [ "$_param" = "-v" ] && verbose=1 && continue
    [ "$_param" = "-d" ] && debug=1 && continue
    # с параметрами
    if [ "$_param" = "-i" ]; then
        input="$1"
        shift
        continue
    fi
done
```

Где-то так. Например.

khim 05.04.17 в 19:28

0

Эты вы разобрали параметры, которые вам пришли обработали. А я про те, которые от вас уйти должны.

Напишите, скажем, скрипт с названием gss, например, который получает аргументы, находит среди них -с, и, если получается, то вызывает gss.geal дважды: один раз «как есть», а второй — заменяя в командной в имени файла после -о расширение с .о на .ii и вместо -с вставляя -Е.

С использованием массивов это делается. Без них — будут проблемы если имена файлов или каталогов будут с пробелами, возвратами кареток, etc.

iig 05.04.17 в 20:00

0

Да, это тот случай, когда массив в bash нужен. Хотя можно обойтись, но будет некрасиво...

Замечу, в программистских исходниках очень редко попадают имена файлов с возвратами кареток ;)

khim 05.04.17 в 20:07

0

Замечу, в программистских исходниках очень редко попадают имена файлов с возвратами кареток ;)

Собственно это самая большая проблема с bash'ем: то, что «сходу» приходит на ум — как правило работает в 99% случаев. А потом в результате чьей-нибудь идиотской ошибки происходит "mkdir -p `cat some-crazy-file`" (что создаёт, вроде бы, безобидный пустой каталог и всё) и скрипты, «работавшие годами» вдруг взрываются...

neit_kas 08.04.17 в 00:02

0

В С, чтобы скопировать массив, тоже надо пару раз присесть ;)

А в чём проблема то? метсру же.

iig 08.04.17 в 07:51

0

И malloc. И рассчитать размер массива. И не забыть free. А если массив указателей, и данные тоже нужно скопировать?

khim 08.04.17 в 12:50

+1

Всё правильно. Но С, насколько мне известно, никогда не позиционирования как «простой в использовании язык для написания скриптов». Это — быстрый, но сложный в использовании — и относительно низкоуровневый язык программирования. Удивляться тому, что в нём сложно скопировать массив не приходится — это реально сложно сделать на уровне машинных кодов!

Но скриптовые языки, как бы, призваны сделать эти задачи простыми — пусть и ценой существенной потери производительности. Так вот bash тут — уникален: потери в производительности есть, ещё какие — а выигрыша в простоте использования нет!

neit_kas 09.04.17 в 05:28

0

И malloc. И рассчитать размер массива. И не забыть free

Это не только в C. Плюса, делфи, да много их таких.

А если массив указателей, и данные тоже нужно скопировать?

А это уже «отсебятинская» (по отношению к языку) структура данных. Опять таки, в большинстве языков их копирование вызовет аналогичные проблемы. Единственное, в C мало «не отсебятинских» структур.

iig 09.04.17 в 07:45

0

И я о том. В обработке структур данных везде есть *особенности*. В bash тоже можно сделать поэлементное копирование, и все будет ясно и предсказуемо. Я бы так делал. А то, что есть способ копировать в 1 строчку, но зависимый от диалекта версии bash — это нехорошо.

MikhailBag 12.01.18 в 20:04

0

Плюсовые массивы (ака векторы) копируются при присваивании.

neit_kas 13.01.18 в 11:21

0

Плюсовые массивы (ака векторы) — это не массивы, а контейнер собранный на их основе. С таким же успехом можно и в C написать функцию, которая будет копировать массив с выделением памяти под новый.

khim 13.01.18 в 11:35

-1

Это вы с чем-то перепутали. Плюсовые массивы — это та ещё кака. Их даже в функцию-то передать и вернуть из неё нельзя!

Но речь не о C.

neit_kas 13.01.18 в 11:38

0

С чего их нельзя в функцию передать и вернуть от туда?

khim 13.01.18 в 14:31

-1

Так C устроен, однако. При попытке передать в функцию — вы, на самом деле, передадите указатель на массив. Если попытаетесь вернуть — просто синтаксическая ошибка.

neit_kas 13.01.18 в 21:35

0

Я не совсем понимаю, о каких массивах речь. Если вы о C++ обертках вроде `std::vector` и подобных, то они без проблем туда-сюда тащатся по ссылкам или по значению (да и по указателю, если очень надо), если вы о чисто C-шных, то, на сколько известно, тягают их по указателю. Стоит отметить, во многих языках сложные конструкции (в т.ч. и массивы) тягают по ссылке (суть указателю). Не понимаю, в чём проблема и в чём недостаток по сравнению с другими языками.

khim 14.01.18 в 12:01

0

Не понимаю, в чём проблема и в чём недостаток по сравнению с другими языками.

Во временах жизни. Если вы внутри функции заведёте массив и вернете его «по ссылке» — то это добром не кончится.

А в bash и передачи по ссылке нету...

neit_kas 15.01.18 в 01:59

0

Во временах жизни. Если вы внутри функции заведёте массив и вернете его «по ссылке» — то это добром не кончится.

Ну, правильно. Потому что массив статический. Сделайте динамическим и возвращайте как хотите. При этом, говоря о плюсах, векторы нормально таскаются туда-сюда. При этом в чём неудобство передачи по ссылке в функцию мне до сих пор не ясно.

А в bash и передачи по ссылке нету...

На счёт bash'a не знаю, но не редко скриптовые языки делают массивы не на стеке, а в куче. При этом в функции и из функций таскается указатель на них. Так что в них просто немного дополнительного синтаксического сахара, ничего принципиально отличного нет.

MikhailBag 13.01.18 в 18:52

+1

Типичные массивы в C — T[], не поддерживают копирование при присваивании просто так.

В C++ типичные массивы — `std::vector`, копируются при присваивании. Если вас не устраивает термин массив, давайте назовем списком.

Если писать на C++, то использовать стоит вещи из C++, имхо.

neit_kas 13.01.18 в 21:31

0

Но отличия C'шных T[] от C++ std::vector в том, что T[] — это конструкция языка, а std::vector — это конструкция библиотеки (и в принципе к языку имеет довольно опосредованное отношение). Никто не мешает обзавестись какой-нибудь C'шной библиотекой для более удобной работы с массивами.



saboteur_kiev 08.04.17 в 16:16

0

в оригинальном баш нет ассоциативных массивов, следовательно нет и проблемы)

khim 08.04.17 в 16:56

0

Конкретно этой проблемы нет. Есть другие.



saboteur_kiev 08.04.17 в 17:05

0

Не очень понятно, что вы хотите сказать. Проблемы есть везде — и в программировании и в жизни.

Но функционал, который доступен в стандартном bash (posix), вполне интуитивен, и нетривиальные задачи, часто имеют более тривиальное решение. Но для этого нужно разбираться по существу.

khim 08.04.17 в 19:44

+1

Но функционал, который доступен в стандартном bash (posix), вполне интуитивен

В том-то и дело, что нет. Одну [задачку](#) мы уже обсуждали. Рассмотрим более простую: напомним скрипт, который мы, опять-таки, назовём gcc и хотим «подсунуть» в PATH. Мы хотим просто вызвать настоящий gcc добавив в командную строку опцию -V4.4.3 (предположим что система сборки у нас такая хитрая, что это в ней сделать сложно). Как нам нужно писать?

```
/path/to/gcc -V4.4.3 $*
```

Нет, так не годится. Нужно вот так:

```
/path/to/gcc -V4.4.3 "$@"
```

Это — один из первых «костылей», которые были добавлены в bash для того, чтобы простое, понятное, но **не работающее** решение превратить в хитрое и странное — но работающее!

В современном bash'e таких костылей — достаточно, и ассоциативные массивы — один из них. Они создают свои проблемы, но в старом bash'e жизнь, увы, не легче. Ибо этих костылей у вас нет и приходится извращаться. Там даже регулярных выражений нет у [[...]]!

alexur 09.04.17 в 09:58

0

напишем скрипт, который мы, опять-таки, назовём gcc

А не проще в таком случае создать alias?

iig 09.04.17 в 11:26

0

Вряд ли. Обычно враппер к gcc вкручивают в кросс-компиляторные тулчейны, чтобы передавать компилятору дополнительные флаги или переменные окружения. А такая вот неявная зависимость кросс-компилятора от особенностей bash — это очень, очень плохо.

khim 09.04.17 в 13:23

0

Ну тут как бы если всё сделать правильно и использовать "\$@" , то никаких зависимостей не будет. Проблема в том, что «интуитивный» и «естественный» код не работает.

Впрочем скажу одну вещь и в защиту bash'a сказать: пусть и с «некрасивым» синтаксисом, пусть и «странно» — но в нём эта задача делается. А вот в Windows мы не нашли **ни одного** встроенного инструмента, позволяющего это сделать. На cmd, ни powershell решить эту задачу не позволяют. Пришлось враппера на C писать в своё время...

neit_kas 09.04.17 в 19:28

0

На cmd, ни powershell решить эту задачу не позволяют.

Хм, заинтересовали. Если речь про [эту](#), то вроде на cmd решается при включении режима расширенной обработки команд (при желании, там не сложно и массивы поднять). Ну, или я задачку не совсем понял. Попробую на досуге.

khim 10.04.17 в 01:47

0

Ну, или я задачку не совсем понял.

Задачку вы, я думаю, поняли — вы пробелы не поняли.

Мне нужен «псевдо-gcc», который вызывает «настоящий» gcc с добавлением в командную строку пары аргументов.

Всего-навсего. Чтобы можно было подsunуть его в произвольную систему сборки.

Кажется можно сделать просто `gcc.cmd` и всё? А вот и нет: что будет если система сборки вызывает какой-нибудь `helper.cmd`:

```
gcc %*
@copy /B %1.done+., %1.done
```

Что будет если у вас в первой строке вызовется `gcc.cmd`? Правильно — `touch` уже не отработает.

Та же самая проблема с PowerShell: если система сборки использует не `ShellExecute`, а `WinExec` — то ваш скрипт работать не будет!

P.S. Конечно дальше выясняется, что и просто взять и написать программу на C тоже нельзя, но это — другая история.

khim 09.04.17 в 13:19

0

О! Спасибо что напомнили ещё про один костыль.

А не проще в таком случае создать `alias`?

Нет, не проще. Алиасы работают только «внутри» `bash`'а, программы, вызванные из него их «не видят».



saboteur_kiev 10.04.17 в 00:29

0

Так а вы внутри баша алиас и вызываете, в чем же проблема сделать алиас, который будет работать только внутри вашего скрипта на БАШЕ?

khim 10.04.17 в 01:34

0

Зачем бы я таким странным способом вызывал `gcc` внутри моего собственно скрипта???

Нет, конечно — разумеется я этот скрипт хочу подsunуть в систему сборки вместо «настоящего» `gcc`.

А систем сборки на `bash`'е я, слава богу, давно не видел...



saboteur_kiev 10.04.17 в 00:27

0

Не понимаю, с чего вы взяли что нужно `$*`?

Из документации, как раз следует, что нужно `$@`, и это не костыль а как раз правильное использование.
<https://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters>

Что же касается кавычек, то это и понятно — вы хотите, чтобы встреченные `wildcards` были раскрыты в вашем скрипте, или во время выполнения команды?

ВСЕ логично. Просто после вдумчивого чтения документации, интуиция работает значительно лучше.

khim 10.04.17 в 01:32

+1

Не понимаю, с чего вы взяли что нужно `$*`?

Потому что только так и можно было использовать переменные в `V6`. Внутри кавычек они не раскрывались, а для передачи аргументов (до 9) нужно было писать `$1 $2 $3 $4 $5 $6 $7 $8 $9` — почти как в DOS.

А в `V7` вспомнили про то, что в именах файлов бывают пробелы и сделали так, что позиционные аргументы стали раскрываться не только вне кавычек, но и внутри. Но при этом `"$*"` вместо DOS-стиля `"$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9"` — использовать оказалось нельзя, так как все аргументы сливались в кучу. Появился первый костыль — в добавление к простому, понятному, но не всегда работающему `$*` — сделали ещё и `"$@"`.

Что же касается кавычек, то это и понятно — вы хотите, чтобы встреченные `wildcards` были раскрыты в вашем скрипте, или во время выполнения команды?

Во время исполнения команды `wildcards` не раскрываются в Unix.

ВСЕ логично

Нелогично то, что любая задача решается 5-10 способами, причём первые 3-7 самых простых — работают не всегда. Такая, немножко иезуитская логика: «а вы не забыли, что в именах файлов могут быть пробелы? ага — а тут у нас костылик надо использовать! а про то, что имена файлов могут начинаться с дефиса не забыли? ну как же — ещё один костылик нужен! а про перевод каретки в имени файла? ну как же без 3го костылика-то?»

Просто после вдумчивого чтения документации, интуиция работает значительно лучше.

Вдумчивое чтение документации позволяет вам достаточно уверенно решать ребусы, которыми являются `bash`-скрипты. Но не делает написание этих скриптов простым занятием всё равно.

Простейший пример: `$ (...)` — это всего лишь замена на ``...``? И вроде как их можно просто заменять друг на друга? Да? Но

ведь нет:

```
$ echo "`echo "\\\\"`"
\
echo "$ (echo "\\\\" )"
\\
```

«Вдумчивое чтение документации», конечно, объяснит в чём разница — но логичней всю эту коллекцию костылей не сделает...



saboteur_kiev 11.04.17 в 17:38

0

Простите, серьезно? Вы ругаетесь на проблемы, которые были в V6 (1975 год) и V7(1979 год)?

Простейший пример: `$(...)` — это всего лишь замена на ``...``?

Не очень удачный пример — конструкция `$(...)` появилась именно для того, чтобы быть нагляднее и решать конкретно этот случай, когда вам нужно вложить подстановку внутри подстановки, в остальном они равноправны.

Проблемы в вашей билд системе, ИМНО заключается в том, что вы хотите своими баш скриптами запускать чужие баш скрипты, а про совместимость никто не думал.

Я могу ошибаться, но можно переходить на современные системы типа maven и всех проблем избежать. То есть ИМНО зря наезжаете на баш.

Если брать именно os-related язык, то bash — гораздо лучше чем cmd/powershell и другие подобные языки.

khim 11.04.17 в 19:20

+1

Вы ругаетесь на проблемы, которые были в V6 (1975 год) и V7(1979 год)?

Я ругаюсь на язык, который содержит в себе большую коллекцию проблем и большую коллекцию костылей, предназначенных для обхода этих проблем. И в котором нужно «решать ребусы» при написании программ, так как простые решения — как правило не на 100% работоспособны.

Не очень удачный пример — конструкция `$(...)` появилась именно для того, чтобы быть нагляднее и решать конкретно этот случай, когда вам нужно вложить подстановку внутри подстановки, в остальном они равноправны.

Точно так же как `$@` появилась чтобы решить проблему с использованием позиционных аргументов в кавычках, собственно. И также как `[[...]]` появился чтобы решить проблемы с `[...]`. Но при этом старые, «проблемные» механизмы никуда не делись, и, собственно, узнать о том, как нужно делать «правильно» зачастую неоткуда. Потому что в том же мануале описана разница между ``...`` и `$ (...)`, но вот **зачем** эта разница нужна — ничего не сказано.

Проблемы в вашей билд системе, ИМНО заключается в том, что вы хотите своими баш скриптами запускать чужие баш скрипты, а про совместимость никто не думал.

Проблема не билд-системе.

Я могу ошибаться, но можно переходить на современные системы типа maven и всех проблем избежать.

Интересная идея. У тебя спрашивают — как пользоваться вашим компилятором, а ты и отвечаешь «да никуда не годятся все ваши GNU Make, Ninja и прочие SCONS'ы, если вы хотите воспользоваться нашим чудо-компилятором — то maven и только maven, там всё будет работать».

Я боюсь начальство такое решение проблемы не одобрит, однако. И будет право.

Если брать именно os-related язык, то bash — гораздо лучше чем cmd/powershell и другие подобные языки.

Однако python (если его можно использовать) — ещё лучше!



saboteur_kiev 13.04.17 в 15:47

0

Плюс вам за питон.

vmspike 03.04.17 в 20:59

0

[Shell Style Guide от Google](#)

[Тот самый Advanced Bash-Scripting Guide](#)

[Учебное пособие на eddnet.org](#)

[Тред на StackOverflow о скрытых фишках bash](#)

[Полезные одно-строчные скрипты sed](#)

Ну и `man bash` периодически покуривать имея ввиду, что на удалённом хосте версия bash может отличаться и некоторые функции могут быть [не]доступны.



nsemikov 03.04.17 в 21:44

0

Всегда пишу расширение для скриптов. Может немного старомодно, но хотя бы нет путаницы: увидел `myscript.sh` и сразу понял какой интерпретатор используется.

Erelecano 03.04.17 в 23:47

0

Это не старомодно, это у вас привычки из винды. И какой же интерпретатор используется, если `.sh?` `dash?` `bash?` `sh?` `kcsh?` `tcsh?` `zsh?` Интерпретатор нужно указывать в начале скрипта и оно прекрасно будет запускаться, а ваше «расширение» ничего не делает и ничего не дает.

grieverrr 04.04.17 в 01:03

0

/bin/sh очевидно же!

Erelecano 04.04.17 в 01:17

0

Ага, очевидно человек для bash-скриптов тогда использует расширение `.bash`, при чем так как у `bash` от версии к версии были изменения еще и наверное расширения `.bash2`, `bash3`, `bash4`... Слабо верится.

ShadoWalkeR30 04.04.17 в 13:22

0

Скрипты так то можно и на Ruby и на Lua если что писать. И интерпретатор там будет `#!/usr/bin/env ruby` и тд. если что.

iig 04.04.17 в 14:40

0

Если расширение указывает на особенности того что внутри — почему бы и нет?

`.sh` — внутри скрипт на shell, скорее всего совместим с `bash/ash/dash`

`.bash` — внутри башизмы, может не запуститься на произвольном shell без напильника.

 nsemikov 04.04.17 в 18:02

0

Именно это и имел ввиду. Добавлю еще, что скрипты могут быть и на PHP | python | ruby | etc...

 saboteur_kiev 07.04.17 в 19:12

0

От расширения вообще ничего не зависит.

Если не указан `shabang` с интерпретатором, запустится в том, откуда вызываете.

groaner 03.04.17 в 22:51

+1

Упражнение на дом: что выведет последний скрипт если в переменной `$mydir` будет начинающееся с пробела имя существующей директории?

iig 03.04.17 в 23:27

0

То же самое, если в `$mydir` пробел в середине. Выведет, но не то, что хотелось.

groaner 04.04.17 в 00:03

+2


Именно. Убеден, что изучение командной оболочки *nix надо начинать с осознания таких вот драматических несоответствий в её дизайне. Типа:

— О, давай у нас оболочка сама будет шаблоны разворачивать!

— Круто!... Эй, подожди, у нас же имена файлов могут содержать почти любые символы, включая `*` и `-`!!!

— Ну и что? В 99% случаев будет работать, да и ладно!

Чтобы потом не было мучительно больно за скрипт, выполнивший немного не то, что ты рассчитывал.

 Lelik13a 04.04.17 в 05:12


0

По нынешнем временам в баше предпочтительно использовать `"[[]]"` вместо `"[]"`. На эту тему даже отдельный пункт есть в [подводные камни Bash](#).

hagent 04.04.17 в 07:24

+1

я понимаю что ко многим серверам нет доступа и нельзя поставить какой то скриптовый язык, но мне кажется стоит упомянуть что сложные задачи лучше получатся на чемнибудь другом не на баше

 Alukardd 04.04.17 в 14:27

+2

Ну сколько можно писать статьи о Bash и упорно лепить sh-совместимые команды?!

Bash гораздо функциональнее чем описанные здесь примеры.

Я уже не говорю о том, что вы пишете ужасный код, и некоторые привычки могут рано или поздно вылиться Вам боком, например использование `[`, вместо `[[`, или использование ``$a``, вместо ``a`` внутри `$()`. Конечно всё это рабочие варианты, если помнить о разных нюансах, но тогда не надо статью озаглавливать как «Bash-скрипты».

vd1 04.04.17 в 14:52


0

имхо лучшие материалы по bash это:

<http://mywiki.woledge.org/BashFAQ>

<http://mywiki.woledge.org/BashGuide>

<http://wiki.bash-hackers.org/>

 Openmsk 04.04.17 в 17:35

-5

powershell реально мощнее bash, ждем полноценного прихода на linux/mac

НЛО прилетело и опубликовало эту надпись здесь

fireSparrow 05.04.17 в 09:30

0

Вряд ли powershell придёт на линукс, потому что почти в любом дистрибутиве из коробки присутствует python. А он гораздо практичнее.

emanation 04.04.17 в 17:53

0

Если уж начали с азов про bash, то с самого начала надо объяснить почему так происходит

```
echo -e '#!/bin/bash\nnecho $$\n' > tryit.sh
chmod 750 !$
./tryit.sh
./tryit.sh
./tryit.sh
source ./tryit.sh
source ./tryit.sh
source ./tryit.sh
```

ну а во второй части уже пора бы узнать что такое \$\$, \$! и т.д.
И почему моя echo команда не будет работать с двойными кавычками...



Himura 04.04.17 в 18:43

0

str1 < str2Возвращает истину, если str1меньше, чем str2.

А какая строка меньше "десять" или "тыща"? Совершенно непонятно ЧТО сравнивается в строках, а потом статья говорит что можно еще и сортировать с помощью этого оператора. WAT??? Сравниваются ASCII-коды символов?? Как можно сравнить два массива чисел разной длины? Может всё-таки по размеру сначала, а при равенстве размера какая-то еще логика? Вот этот вопрос не ясен...

Обратите внимание на то, что скрипт, хотя и выполняется, выдаёт предупреждение:

./myscript: line 5: [: too many arguments

Для того, чтобы избавиться от этого предупреждения, заключим \$val2 в двойные кавычки:

Здесь пропущен архи-важный ответ на вопрос "зачем?". Я не понимаю как bash интерпретировал выражение и ПОЧЕМУ кавычки эту интерпритацию меняют. Вот эти все сравнения строк для меня всегда были какой-то черной магией с десятком разных подходов и Ваша статья еще сильнее убеждает меня в том что так и есть.



Himura 04.04.17 в 18:49

0

Я даже проверить не смог после этой статьи... Не хватает хороших мануалов по bash, не хватает...

```
root@W10:~# "десять" \> "тыща"
десять: command not found
root@W10:~# if ["десять" \> "тыща"] echo true
>
> fi
bash: syntax error near unexpected token `fi'
root@W10:~# if ["десять" \> "тыща"]; echo true; fi
bash: syntax error near unexpected token `fi'
root@W10:~# if ["десять" \> "тыща"] then echo true; fi
bash: syntax error near unexpected token `fi'
root@W10:~# if ["десять" \> "тыща"] then echo true fi
> ;
bash: syntax error near unexpected token `;'
root@W10:~# if ["десять" \> "тыща"] then; echo true; fi
bash: syntax error near unexpected token `fi'
root@W10:~# if ["десять" \> "тыща"] then; echo true; fi
bash: syntax error near unexpected token `fi'
root@W10:~#
```

iig 04.04.17 в 19:09

+1

```
if ["десять" \> "тыща"] then; echo true; fi
```


```
if [ "десять" \> "тыща" ]; then echo true; fi
```

Найдите 3 отличия ;)

 Himura 04.04.17 в 23:15 0
ну да, я быстро сдался, спасибо. Надеюсь, запомню где там обязательные разрывы а где необязательные...

iig 04.04.17 в 23:22 +2

Все обязательные ;)
[— это команда с параметрами. Параметры разделяют пробелами.

 Himura 05.04.17 в 09:10 0

Вот теперь я вижу логику языка, спасибо. А когда после then вместо пробела разрыв строки, это больше похож на какую-то особую конструкцию языка чем на три команды с параметрами.

khim 05.04.17 в 16:18 0

Разрыв строки должен быть перед then, а не после, однако.

 Himura 05.04.17 в 16:40 0

По [коменту](#) это очевидно, а по исходной статье — нет. Я бы придерживался the one true brace style:

```
if [ "десять" \> "тыща"]; then
    echo true;
fi
```

Этот semicolon слишком подозрительно выглядит чтобы его игнорировать, сразу понятно всё.

khim 05.04.17 в 20:12 +1

А зачем после `echo true` ставить semicolon?

В bash'e semicolon — просто синоним перевода строки, зачем вам ещё одна пустая строка там?

iig 04.04.17 в 19:01 0

Как можно сравнить два массива чисел разной длины?

strcmp, strcmpi... Можно.

Зачем? Хотя бы чтобы был критерий для сортировки :)

Для того, чтобы избавиться от этого предупреждения, заключим \$val2 в двойные кавычки:

Это особенная, bash'евская магия! Если в \$val2 пустая строка — в оператор сравнения не передается один из параметров. А если в \$val2 строка с пробелами — передается N параметров. Если пустая строка, но в кавычках — передается пустая строка. А если в одинарных кавычках — переменные внутри кавычек не преобразуются в значения.

КМК очень, очень плохая идея — обрабатывать строки в bash. Разве что от безысходности. perl создан для этого ;).

khim 04.04.17 в 19:21 -1

Совершенно непонятно ЧТО сравнивается в строках, а потом статья говорит что можно еще и сортировать с помощью этого оператора.

А как строки сравниваются в других языках — вам понятно? Pascal, Python, C? [stroll](#)? Толковый словарь Ожегова?

Я не уверен что задачей статьи было обучить писать скрипты человека, который о программировании не знает ничего вообще — и гордится этим.

 Himura 04.04.17 в 23:10 0

Лексикографический порядок не является единственным возможным вариантом. Сходу нагуглился как минимум Kleene–Brouwer order. Кроме того, статья позиционируется как обучающая и совершенно точно не должна оставлять таких вопросов (ссылки на википедию было бы более чем достаточно). Я радикально не согласен с тем что задачей статьи не является "обучить писать скрипты человека, который о программировании не знает ничего вообще", потому что для автоматизации простых задач совершенно не требуется знать что такое лексикографический порядок и что на самом деле "десять" > "тыща".

khim 05.04.17 в 00:42 +1

Тут есть некоторая проблема: судя по вашему тону вы считаете, что писать скрипты на bash'e — относительно легко и этому полезно обучать новичков.

К огромному сожалению оба посыла ложны: писать скрипты на bash **сложно** и, как правило, **не нужно**: есть много других языков, где решение простейших задач не превращается в ребус.

А если вас **необходимо** писать скрипты по той или иной причине, то вы, скорее всего уже не один язык программирования знаете и рассказывать вам про лексикографический порядок — не нужно от слова «совсем»...

P.S. Если же ваша задача не «написать скрипт, который работает независимо от того, есть ли в имени файла, с которым он работает, возврат каретки или нет», а «написать скрипт, который, как правило, работает — если звёзды стоят правильно», то тут и вообще никакой серии статей не нужно: SODD вполне работает, если вас устраивает не вполне гарантированный результат — зачем ещё и статьи какие-то?



Himura 05.04.17 в 09:06

0

Вот сейчас Вы правы, я считал что bash позиционируется как "простой" язык. Если в реальности bash-скриптинг — это действительно более ребусы, чем продуктивность, то да, лучше на том же питоне писать (лично я так и делаю). Но все-таки, мне всегда казалось что я что-то упускаю... Ведь именно bash является стандартной оболочкой всех современных линуксов. Создаётся впечатление, что это стандарт отрасли и вообще говоря у него большие шансы стать первым языком у юзера хотя бы потому что ему поневоле приходится писать `apt-get` и `dpkg -i`.

Если **необходимо** писать и есть конкретная задача, то решить её при помощи Гугла или даже `man` — никакого труда не составит. А статья нужна чтобы узнать как **правильно, надёжно** и без хаков писать рутинные вещи типа `if`. И перестать тратить время на выяснение этого каждый раз. Хотя, про то что это именно та статья, которая будет находится по запросам в Гугл, я не подумал. Но судя по количеству неточностей в статье, которые раскрыты коментами, не уверен что это хорошо.

iig 05.04.17 в 09:34

0

как правильно, надёжно и без хаков писать рутинные вещи типа `if`

`man` же. Кроме `if`, есть и другие конструкции.



Himura 05.04.17 в 09:46

+1

Ну, давайте не будем писать обучающие статьи вообще. И переводить `man` тоже не нужно, каждый кто общается с консолью обязан знать английский. И `nano` выпилить из всех дистрибутивов, только `sed` и `vim`, пусть как хотят так и колушаются, `nano` не тру. Повысим порог вхождения до небес, nobody needs lamers.

iig 05.04.17 в 10:19

0

Синтаксис языковых конструкций лучше смотреть в первоисточнике. А в обучающей статье лучше бы рассказать, зачем вообще (не)нужно писать скрипты на bash. С реальными use case.

khim 06.04.17 в 03:00

0

Я вот знаю только одно применения для bash'a: на нём необходимо писать скрипты для сборки всяких `rpm`'ов и `ebuild`'ов. По историческим причинам.

Также полезно его знать если вы используете `make` и тому подобные вещи.

В обоих случаях у вас, во-первых, не так много выбора, а во-вторых, так как вы работаете с чем-то, что заслуживает доверия, то для вас не так важно писать скрипты, умеющие работать с файлами, названия которых содержат странные символы,

Во всех остальных случаях, увы, bash использовать не стоит.

khim 05.04.17 в 20:02

+1

Создаётся впечатление, что это стандарт отрасли и вообще говоря у него большие шансы стать первым языком у юзера хотя бы потому что ему поневоле приходится писать `apt-get` и `dpkg -i`.

Это «стандарт отрасли», потому что почти полвека назад его древний предок был стандартным shell'ом — и больше ничему.

Если **необходимо** писать и есть конкретная задача, то решить её при помощи Гугла или даже `man` — никакого труда не составит.

Составит, к сожалению.

Вот сейчас Вы правы, я считал что bash позиционируется как «простой» язык.

Простой язык — это `sh`. Но многие вещи в нём, к сожалению, не делаются от слова «никак». А bash — это попытка добавить костылей, чтобы они так делались. В результате — да, всё делается, всё как бы хорошо... вот только одна беда: из-за пресловутой «обратной совместимости» новые, работающие «костыльные» решения — неочевидны ни разу. А старые, «простые и понятные» — не работают!

Вот как, например, прочитать список строк, сохранённых в файле `filelist.zero-delim` и разделённых там символом с кодом 0 (потому что, блин, все остальные символы могут встречаться в именах файлов) и передать их в командной строке в вызове одной команды?

Ну, например, так:

```
declare -a file_list
while IFS='' read -r -d '' file_name; do
    file_list+=("${file_list[@]}:${file_list[@]}") "$file_name"
done < filelist.zero-delim
process "${file_list[@]}:${file_list[@]}"
```

Офигительно просто, не так ли? Это вообще — выглядит как программа? Нет — это ребус.

А 99% ответов, которые вы найдёте на просторах интернета будут решать эту задачу неправильно либо небезопасно (будут «взрываться» при использовании `set -o nounset`, например).

В общем мой вам совет: используйте bash-скрипты тогда, когда вы можете быть уверены что «враг» не передаст вам плохих данных (например если вы сами и пишете и используете их), либо если у вас нет выбора. Если выбор есть, то... лучше что-нибудь другое...

iig 05.04.17 в 22:21

0

прочитать список строк, сохранённых в файле `filelist.zero-delim` и разделённых там символом с кодом 0 и передать их в командной строке в вызове одной команды?

1. `xargs`?
2. Кто-то ведь зачем-то создал этот файл? Можно было бы, наверное, передавать список и без файла, через конвейер?
3. `tar`, например, умеет получать список файлов из файла. Может, и не нужно решать эту задачу?

khim 05.04.17 в 22:30

+1

1. `xargs`?

`xargs` позволит вам решить ровно эту задачу — и ничего более. Ни фильтрации, ни обработки, ничего.

2. Кто-то ведь зачем-то создал этот файл?

Ну допустим его создали, скажем, использованием `find` с какими-то там параметрами — вам легче стало?

Можно было бы, наверное, передавать список и без файла, через конвейер?

От этого задача стала бы только сложнее, увы.

3. `tar`, например, умеет получать список файлов из файла. Может, и не нужно решать эту задачу?

Во-первых он требует списка файлов разделённых `'\n'` — то есть с произвольными именами файлов работать не может. А во-вторых так мы дойдём до того, что будем писать программу на `bash`'е, которая выглядит как `perl -e '...'`.

Я весьма неплохо знаю `bash`, `tar` и прочие штуки, но давайте посмотрим правде в глаза: это корявые инструменты. Очень корявые. То, что мы научились с ними как-то жить — этой истины, увы, не отменяет...

iig 05.04.17 в 23:01

0

Идеальных инструментов не бывает. Вы знаете о недостатках `bash` — это не делает его плохим. Просто не надо делать на нем то, к чему он не приспособлен.

Да, пример с `find` мне кажется натянутым. :)

khim 05.04.17 в 23:33

0

Просто не надо делать на нем то, к чему он не приспособлен.

Он «не приспособлен» работать со списками файлов, среди которых может встретиться «нечто странное» (пробелы, хотя бы). Что, в общем, делает его мало пригодным для чего-либо вообще: в современном мире инструмент, который может, внезапно, сломаться, если у вас в каталоге обнаружится файл с «неправильным» именем — просто слишком опасен, чтобы рекомендовать его использовать где-бы-то-ни-было... Слишком велик риск...

Только [полноправные пользователи](#) могут оставлять комментарии. [Войдите](#), пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

В чём важность $196\,884 = 196\,883 + 1$? Как это объяснить на пальцах?

+39

14,9k

93

56

Для устранения Spectre и Meltdown, возможно, придётся создать процессор совершенно нового типа

+19

8,9k

27

62

Новые техпроцессы для производства микросхем все чаще откладывают — почему?

+57 29,3k 40 99

Раскрываем номера пользователей Telegram

+52 14,5k 88 92

Как создавался World of Warcraft: взгляд изнутри на 20 лет разработки

+45 27,5k 150 51

Аккаунт

Войти

Регистрация

Разделы

Публикации

Хабы

Компании

Пользователи

Песочница

Информация

Правила

Помощь

Документация

Соглашение

Конфиденциальность

Услуги

Реклама

Тарифы

Контент

Семинары

Приложения

Загрузите в App Store

доступно в Google Play

TM

© 2006 – 2018 «TM»

О сайте

Служба поддержки

Мобильная версия