Bash-скрипты, часть 4: ввод и вывод

<https://likegeeks.com/shell-scripting-awesome-guide-part4/>

* [Блог компании RUVDS.com](https://habr.com/company/ruvds/),
* [Настройка Linux](https://habr.com/hub/linux/),
* [Серверное администрирование](https://habr.com/hub/s_admin/)
* Перевод

[Bash-скрипты: начало](https://habrahabr.ru/company/ruvds/blog/325522/)  
[Bash-скрипты, часть 2: циклы](https://habrahabr.ru/company/ruvds/blog/325928/)  
[Bash-скрипты, часть 3: параметры и ключи командной строки](https://habrahabr.ru/company/ruvds/blog/326328/)   
[Bash-скрипты, часть 4: ввод и вывод](https://habrahabr.ru/company/ruvds/blog/326594/)  
[Bash-скрипты, часть 5: сигналы, фоновые задачи, управление сценариями](https://habrahabr.ru/company/ruvds/blog/326826/)  
[Bash-скрипты, часть 6: функции и разработка библиотек](https://habrahabr.ru/company/ruvds/blog/327248/)  
[Bash-скрипты, часть 7: sed и обработка текстов](https://habrahabr.ru/company/ruvds/blog/327530/)  
[Bash-скрипты, часть 8: язык обработки данных awk](https://habrahabr.ru/company/ruvds/blog/327754/)   
[Bash-скрипты, часть 9: регулярные выражения](https://habrahabr.ru/company/ruvds/blog/327896/)  
[Bash-скрипты, часть 10: практические примеры](https://habrahabr.ru/company/ruvds/blog/328346/)  
[Bash-скрипты, часть 11: expect и автоматизация интерактивных утилит](https://habrahabr.ru/company/ruvds/blog/328436/)

В прошлый раз, в третьей части этой серии материалов по bash-скриптам, мы говорили о параметрах командной строки и ключах. Наша сегодняшняя тема — ввод, вывод, и всё, что с этим связано.

[[](https://habrahabr.ru/company/ruvds/blog/326594/)](https://habrahabr.ru/company/ruvds/blog/326594/)

[https://habrastorage.org/files/1ba/550/d25/1ba550d25e8846ce8805de564da6aa63.png](https://ruvds.com/ru-rub/#order)  
  
Вы уже знакомы с двумя методами работы с тем, что выводят сценарии командной строки:

* Отображение выводимых данных на экране.
* Перенаправление вывода в файл.

Иногда что-то надо показать на экране, а что-то — записать в файл, поэтому нужно разобраться с тем, как в Linux обрабатывается ввод и вывод, а значит — научиться отправлять результаты работы сценариев туда, куда нужно. Начнём с разговора о стандартных дескрипторах файлов.

Стандартные дескрипторы файлов

Всё в Linux — это файлы, в том числе — ввод и вывод. Операционная система идентифицирует файлы с использованием дескрипторов.  
  
Каждому процессу позволено иметь до девяти открытых дескрипторов файлов. Оболочка bash резервирует первые три дескриптора с идентификаторами 0, 1 и 2. Вот что они означают.

* 0, STDIN — стандартный поток ввода.
* 1, STDOUT — стандартный поток вывода.
* 2, STDERR — стандартный поток ошибок.

Эти три специальных дескриптора обрабатывают ввод и вывод данных в сценарии.  
Вам нужно как следует разобраться в стандартных потоках. Их можно сравнить с фундаментом, на котором строится взаимодействие скриптов с внешним миром. Рассмотрим подробности о них.

STDIN

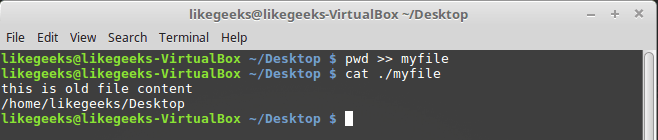
STDIN — это стандартный поток ввода оболочки. Для терминала стандартный ввод — это клавиатура. Когда в сценариях используют символ перенаправления ввода — <, Linux заменяет дескриптор файла стандартного ввода на тот, который указан в команде. Система читает файл и обрабатывает данные так, будто они введены с клавиатуры.  
  
Многие команды bash принимают ввод из STDIN, если в командной строке не указан файл, из которого надо брать данные. Например, это справедливо для команды cat.  
  
Когда вы вводите команду cat в командной строке, не задавая параметров, она принимает ввод из STDIN. После того, как вы вводите очередную строку, cat просто выводит её на экран.

STDOUT

STDOUT — стандартный поток вывода оболочки. По умолчанию это — экран. Большинство bash-команд выводят данные в STDOUT, что приводит к их появлению в консоли. Данные можно перенаправить в файл, присоединяя их к его содержимому, для этого служит команда >>.  
  
Итак, у нас есть некий файл с данными, к которому мы можем добавить другие данные с помощью этой команды:

pwd >> myfile

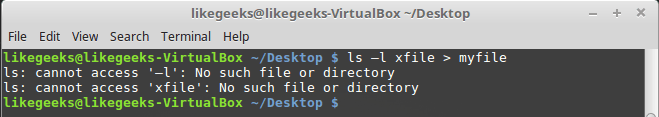
То, что выведет pwd, будет добавлено к файлу myfile, при этом уже имеющиеся в нём данные никуда не денутся.



*Перенаправление вывода команды в файл*  
  
Пока всё хорошо, но что если попытаться выполнить что-то вроде показанного ниже, обратившись к несуществующему файлу xfile, задумывая всё это для того, чтобы в файл myfile попало сообщение об ошибке.

ls –l xfile > myfile

После выполнения этой команды мы увидим сообщения об ошибках на экране.



*Попытка обращения к несуществующему файлу*  
  
При попытке обращения к несуществующему файлу генерируется ошибка, но оболочка не перенаправила сообщения об ошибках в файл, выведя их на экран. Но мы-то хотели, чтобы сообщения об ошибках попали в файл. Что делать? Ответ прост — воспользоваться третьим стандартным дескриптором.

STDERR

STDERR представляет собой стандартный поток ошибок оболочки. По умолчанию этот дескриптор указывает на то же самое, на что указывает STDOUT, именно поэтому при возникновении ошибки мы видим сообщение на экране.  
  
Итак, предположим, что надо перенаправить сообщения об ошибках, скажем, в лог-файл, или куда-нибудь ещё, вместо того, чтобы выводить их на экран.

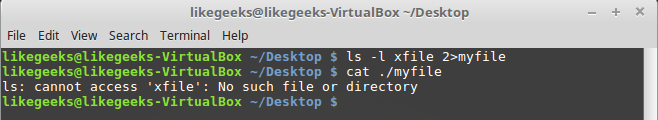
▍Перенаправление потока ошибок

Как вы уже знаете, дескриптор файла STDERR — 2. Мы можем перенаправить ошибки, разместив этот дескриптор перед командой перенаправления:

ls -l xfile 2>myfile

cat ./myfile

Сообщение об ошибке теперь попадёт в файл myfile.

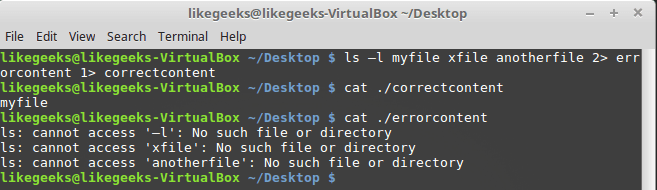


*Перенаправление сообщения об ошибке в файл*

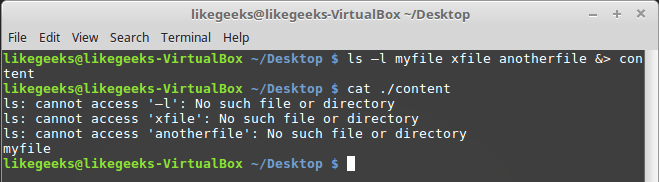
▍Перенаправление потоков ошибок и вывода

При написании сценариев командной строки может возникнуть ситуация, когда нужно организовать и перенаправление сообщений об ошибках, и перенаправление стандартного вывода. Для того, чтобы этого добиться, нужно использовать команды перенаправления для соответствующих дескрипторов с указанием файлов, куда должны попадать ошибки и стандартный вывод:

ls –l myfile xfile anotherfile 2> errorcontent 1> correctcontent



*Перенаправление ошибок и стандартного вывода*  
  
Оболочка перенаправит то, что команда ls обычно отправляет в STDOUT, в файл correctcontentблагодаря конструкции 1>. Сообщения об ошибках, которые попали бы в STDERR, оказываются в файле errorcontent из-за команды перенаправления 2>.  
  
Если надо, и STDERR, и STDOUT можно перенаправить в один и тот же файл, воспользовавшись командой &>:



*Перенаправление STDERR и STDOUT в один и тот же файл*  
  
После выполнения команды то, что предназначено для STDERR и STDOUT, оказывается в файле content.

Перенаправление вывода в скриптах

Существует два метода перенаправления вывода в сценариях командной строки:

* Временное перенаправление, или перенаправление вывода одной строки.
* Постоянное перенаправление, или перенаправление всего вывода в скрипте либо в какой-то его части.

▍Временное перенаправление вывода

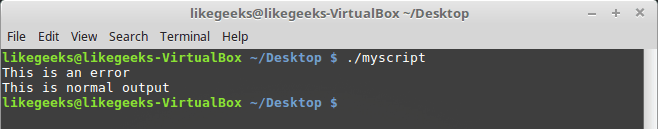
В скрипте можно перенаправить вывод отдельной строки в STDERR. Для того, чтобы это сделать, достаточно использовать команду перенаправления, указав дескриптор STDERR, при этом перед номером дескриптора надо поставить символ амперсанда (&):

#!/bin/bash

echo "This is an error" >&2

echo "This is normal output"

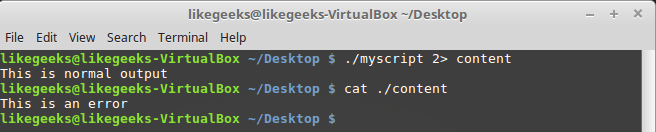
Если запустить скрипт, обе строки попадут на экран, так как, как вы уже знаете, по умолчанию ошибки выводятся туда же, куда и обычные данные.



*Временное перенаправление*  
  
Запустим скрипт так, чтобы вывод STDERR попадал в файл.

./myscript 2> myfile

Как видно, теперь обычный вывод делается в консоль, а сообщения об ошибках попадают в файл.



*Сообщения об ошибках записываются в файл*

▍Постоянное перенаправление вывода

Если в скрипте нужно перенаправлять много выводимых на экран данных, добавлять соответствующую команду к каждому вызову echo неудобно. Вместо этого можно задать перенаправление вывода в определённый дескриптор на время выполнения скрипта, воспользовавшись командой exec:

#!/bin/bash

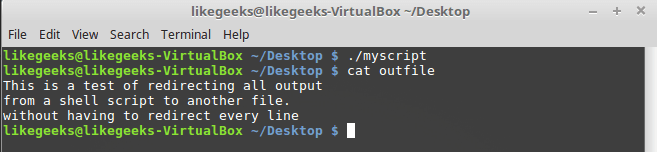
exec 1>outfile

echo "This is a test of redirecting all output"

echo "from a shell script to another file."

echo "without having to redirect every line"

Запустим скрипт.



*Перенаправление всего вывода в файл*  
  
Если просмотреть файл, указанный в команде перенаправления вывода, окажется, что всё, что выводилось командами echo, попало в этот файл.  
  
Команду exec можно использовать не только в начале скрипта, но и в других местах:

#!/bin/bash

exec 2>myerror

echo "This is the start of the script"

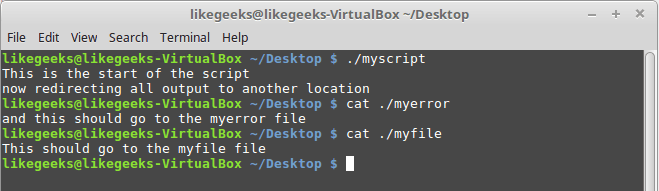
echo "now redirecting all output to another location"

exec 1>myfile

echo "This should go to the myfile file"

echo "and this should go to the myerror file" >&2

Вот что получится после запуска скрипта и просмотра файлов, в которые мы перенаправляли вывод.



*Перенаправление вывода в разные файлы*  
  
Сначала команда exec задаёт перенаправление вывода из STDERR в файл myerror. Затем вывод нескольких команд echo отправляется в STDOUT и выводится на экран. После этого команда exec задаёт отправку того, что попадает в STDOUT, в файл myfile, и, наконец, мы пользуемся командой перенаправления в STDERR в команде echo, что приводит к записи соответствующей строки в файл myerror.  
  
Освоив это, вы сможете перенаправлять вывод туда, куда нужно. Теперь поговорим о перенаправлении ввода.

Перенаправление ввода в скриптах

Для перенаправления ввода можно воспользоваться той же методикой, которую мы применяли для перенаправления вывода. Например, команда exec позволяет сделать источником данных для STDINкакой-нибудь файл:

exec 0< myfile

Эта команда указывает оболочке на то, что источником вводимых данных должен стать файл myfile, а не обычный STDIN. Посмотрим на перенаправление ввода в действии:

#!/bin/bash

exec 0< testfile

count=1

while read line

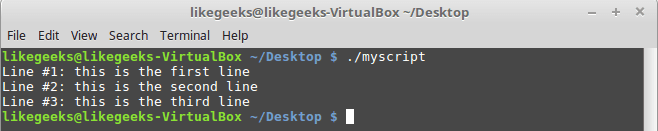
do

echo "Line #$count: $line"

count=$(( $count + 1 ))

done

Вот что появится на экране после запуска скрипта.



*Перенаправление ввода*  
  
В одном из предыдущих материалов вы узнали о том, как использовать команду read для чтения данных, вводимых пользователем с клавиатуры. Если перенаправить ввод, сделав источником данных файл, то команда read, при попытке прочитать данные из STDIN, будет читать их из файла, а не с клавиатуры.  
  
Некоторые администраторы Linux используют этот подход для чтения и последующей обработки лог-файлов.

Создание собственного перенаправления вывода

Перенаправляя ввод и вывод в сценариях, вы не ограничены тремя стандартными дескрипторами файлов. Как уже говорилось, можно иметь до девяти открытых дескрипторов. Остальные шесть, с номерами от 3 до 8, можно использовать для перенаправления ввода или вывода. Любой из них можно назначить файлу и использовать в коде скрипта.  
  
Назначить дескриптор для вывода данных можно, используя команду exec:

#!/bin/bash

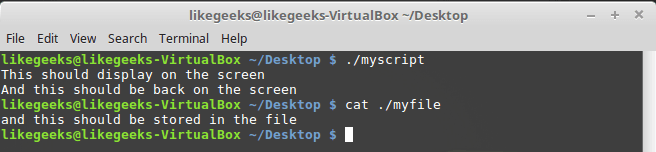
exec 3>myfile

echo "This should display on the screen"

echo "and this should be stored in the file" >&3

echo "And this should be back on the screen"

После запуска скрипта часть вывода попадёт на экран, часть — в файл с дескриптором 3.



*Перенаправление вывода, используя собственный дескриптор*

Создание дескрипторов файлов для ввода данных

Перенаправить ввод в скрипте можно точно так же, как и вывод. Сохраните STDIN в другом дескрипторе, прежде чем перенаправлять ввод данных.  
  
После окончания чтения файла можно восстановить STDIN и пользоваться им как обычно:

#!/bin/bash

exec 6<&0

exec 0< myfile

count=1

while read line

do

echo "Line #$count: $line"

count=$(( $count + 1 ))

done

exec 0<&6

read -p "Are you done now? " answer

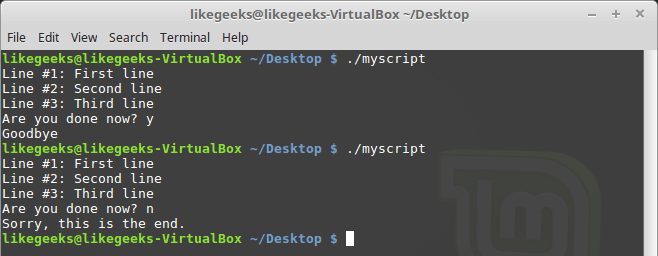
case $answer in

y) echo "Goodbye";;

n) echo "Sorry, this is the end.";;

esac

Испытаем сценарий.



*Перенаправление ввода*  
  
В этом примере дескриптор файла 6 использовался для хранения ссылки на STDIN. Затем было сделано перенаправление ввода, источником данных для STDIN стал файл. После этого входные данные для команды read поступали из перенаправленного STDIN, то есть из файла.  
  
После чтения файла мы возвращаем STDIN в исходное состояние, перенаправляя его в дескриптор 6. Теперь, для того, чтобы проверить, что всё работает правильно, скрипт задаёт пользователю вопрос, ожидает ввода с клавиатуры и обрабатывает то, что введено.

Закрытие дескрипторов файлов

Оболочка автоматически закрывает дескрипторы файлов после завершения работы скрипта. Однако, в некоторых случаях нужно закрывать дескрипторы вручную, до того, как скрипт закончит работу. Для того, чтобы закрыть дескриптор, его нужно перенаправить в &-. Выглядит это так:

#!/bin/bash

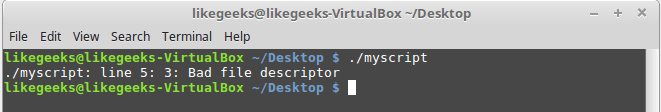
exec 3> myfile

echo "This is a test line of data" >&3

exec 3>&-

echo "This won't work" >&3

После исполнения скрипта мы получим сообщение об ошибке.



*Попытка обращения к закрытому дескриптору файла*  
  
Всё дело в том, что мы попытались обратиться к несуществующему дескриптору.  
  
Будьте внимательны, закрывая дескрипторы файлов в сценариях. Если вы отправляли данные в файл, потом закрыли дескриптор, потом — открыли снова, оболочка заменит существующий файл новым. То есть всё то, что было записано в этот файл ранее, будет утеряно.

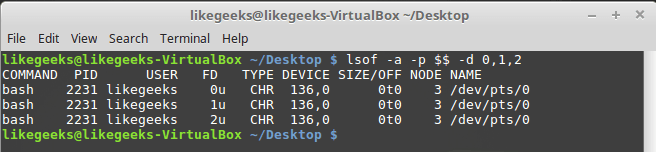
Получение сведений об открытых дескрипторах

Для того, чтобы получить список всех открытых в Linux дескрипторов, можно воспользоваться командой lsof. Во многих дистрибутивах, вроде Fedora, утилита lsof находится в /usr/sbin. Эта команда весьма полезна, так как она выводит сведения о каждом дескрипторе, открытом в системе. Сюда входит и то, что открыли процессы, выполняемые в фоне, и то, что открыто пользователями, вошедшими в систему.  
  
У этой команды есть множество ключей, рассмотрим самые важные.

* -p Позволяет указать ID процесса.
* -d Позволяет указать номер дескриптора, о котором надо получить сведения.

Для того, чтобы узнать PID текущего процесса, можно использовать специальную переменную окружения $$, в которую оболочка записывает текущий PID.  
  
Ключ -a используется для выполнения операции логического И над результатами, возвращёнными благодаря использованию двух других ключей:

lsof -a -p $$ -d 0,1,2



*Вывод сведений об открытых дескрипторах*  
  
Тип файлов, связанных с STDIN, STDOUT и STDERR — CHR (character mode, символьный режим). Так как все они указывают на терминал, имя файла соответствует имени устройства, назначенного терминалу. Все три стандартных файла доступны и для чтения, и для записи.  
  
Посмотрим на вызов команды lsof из скрипта, в котором открыты, в дополнение к стандартным, другие дескрипторы:

#!/bin/bash

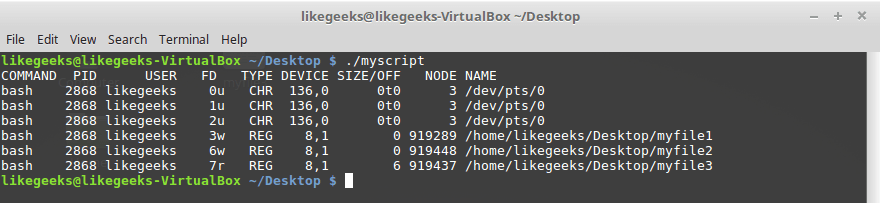
exec 3> myfile1

exec 6> myfile2

exec 7< myfile3

lsof -a -p $$ -d 0,1,2,3,6,7

Вот что получится, если этот скрипт запустить.



*Просмотр дескрипторов файлов, открытых скриптом*  
  
Скрипт открыл два дескриптора для вывода (3 и 6) и один — для ввода (7). Тут же показаны и пути к файлам, использованных для настройки дескрипторов.

Подавление вывода

Иногда надо сделать так, чтобы команды в скрипте, который, например, может исполняться как фоновый процесс, ничего не выводили на экран. Для этого можно перенаправить вывод в /dev/null. Это — что-то вроде «чёрной дыры».  
  
Вот, например, как подавить вывод сообщений об ошибках:

ls -al badfile anotherfile 2> /dev/null

Тот же подход используется, если, например, надо очистить файл, не удаляя его:

cat /dev/null > myfile

Итоги

Сегодня вы узнали о том, как в сценариях командной строки работают ввод и вывод. Теперь вы умеете обращаться с дескрипторами файлов, создавать, просматривать и закрывать их, знаете о перенаправлении потоков ввода, вывода и ошибок. Всё это очень важно в деле разработки bash-скриптов.  
  
В следующий раз поговорим о сигналах Linux, о том, как обрабатывать их в сценариях, о запуске заданий по расписанию и о фоновых задачах.  
  
Уважаемые читатели! В этом материале даны основы работы с потоками ввода, вывода и ошибок. Уверены, среди вас есть профессионалы, которые могут рассказать обо всём этом то, что приходит лишь с опытом. Если так — передаём слово вам.