

Основы сценариев оболочки

Файлы сценариев

Сценарий оболочки обычно находится внутри файла. Файл может быть исполняемым, но вы можете вызвать скрипт Bash с этим именем файла в качестве параметра:

```
bash ./myfile
```

Нет необходимости добавлять скучное расширение имени файла, например `.bash`, или `.sh`. Это пережиток UNIX®, где исполняемые файлы помечаются не расширением, а **разрешениями** (filemode). Имя файла может представлять собой любую комбинацию допустимых символов имени файла. Добавление правильного расширения имени файла - это соглашение, и ничего больше.

```
chmod +x ./myfile
```

Если файл является исполняемым, и вы хотите использовать его, вызывая только имя скрипта, shebang должен быть включен в файл.

The Shebang

Например, спецификация интерпретатора этого файла в файле:

```
#!/bin/bash
echo "Привет, мир ..."
```

Это интерпретируется ядром ¹⁾ вашей системы. В общем случае, если файл является исполняемым, но не является исполняемой (двоичной) программой, и такая строка присутствует, программа, указанная после `#!`, запускается с именем скрипта и всеми его аргументами. Эти два символа `#` и `!` должны быть **первыми двумя байтами** в файле!

Вы можете следить за процессом, используя `echo` в качестве поддельного интерпретатора:

```
#!/bin/echo
```

Здесь нам не нужно тело сценария, так как файл никогда не будет интерпретирован и выполнен "echo". Вы можете видеть, что делает операционная система, она

вызывает `/bin/echo` с именем исполняемого файла и следующими аргументами.

```
$ /home/bash/bin/test тестовое слово привет
/home/bash/bin/test тестовое слово привет
```

Точно так же, с `#!/bin/bash` оболочкой `/bin/bash` вызывается с именем файла скрипта в качестве аргумента. Это то же самое, что выполнение `/bin/bash /home/bash/bin/test testword hello`

Если интерпретатор может быть указан с аргументами и как долго это может быть зависит от системы (см. `#!/-magic` (<http://www.in-ulm.de/~mascheck/various/shebang/>)). Когда Bash выполняет файл с `#!/bin/bash` шаблоном, сам шаблон игнорируется, поскольку первым символом является хэш-метка `"#"`, которая указывает на комментарий. Shebang предназначен для операционной системы, а не для оболочки. Программы, которые не игнорируют такие строки, могут не работать как интерпретаторы, управляемые shebang.

Внимание:

Когда указанный интерпретатор недоступен или не выполняется (разрешения), вы обычно получаете `bad interpreter` сообщение об ошибке `""`. Если вы ничего не получаете, и это не удастся, проверьте shebang. Более старые версии Bash будут отвечать `no such file or directory` ошибкой `""` для несуществующего интерпретатора, указанного в shebang.

Дополнительное примечание: если вы укажете `#!/bin/sh as` shebang и это ссылка на Bash, то Bash будет выполняться в режиме POSIX®! См .:

- Поведение Bash.

Распространенным методом является указание shebang, например

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

... который просто перемещает местоположение потенциальной проблемы в

- `env` утилита должна быть расположена в `/usr/bin/`
- необходимый `bash` двоичный файл должен быть расположен в PATH

Какой из них вам нужен, или считаете ли вы, какой из них хороший или плохой, зависит от вас. Не существует пуленепробиваемого переносимого способа указать интерпретатор. **Распространенное заблуждение, что это решает все проблемы. Точка.**

Стандартные описания файлов

После инициализации каждая обычная UNIX®-программа имеет как *минимум 3 открытых файла*:

- **стандартный ввод:** стандартный ввод
- **стандартный вывод:** стандартный вывод
- **stderr:** стандартный вывод ошибки

Обычно все они подключены к вашему терминалу, `stdin` в качестве входного файла (клавиатура), `stdout` и `stderr` в качестве выходных файлов (экран). При вызове такой программы вызывающая оболочка может изменить эти соединения `filedescriptor` с терминала на любой другой файл (см. Перенаправление). Почему два разных выходных файла описывают? Принято отправлять сообщения об ошибках и предупреждения в `stderr` и только программировать вывод в стандартный вывод. Это позволяет пользователю решить, хочет ли он ничего не видеть, только данные, только ошибки или и то, и другое - и где они хотят их видеть.

Когда вы пишете сценарий:

- всегда считывайте вводимые пользователем данные из `stdin`
- всегда записывайте диагностические / ошибки / предупреждающие сообщения в `stderr`

Чтобы узнать больше о стандартных файловых дескрипторах, особенно о перенаправлении и конвейере, см.:

- Иллюстрированное руководство по перенаправлению

Имена переменных

Рекомендуется использовать имена переменных в нижнем регистре, поскольку имена оболочек и системных переменных обычно пишутся в ВЕРХНЕМ РЕГИСТРЕ. Однако вам следует избегать именования ваших переменных любым из следующих (неполный список!):

BASH	BASH_ARGC	BASH_ARGV	BASH_LINENO	BASH_SOURCE	BASH
BASH_VERSION	COLUMNS	DIRSTACK	DISPLAY	EDITOR	EL
GROUPS	HISTFILE	HISTFILESIZE	HISTSIZE	HOME	HC
IFS	LANG	LANGUAGE	LC_ALL	LINES	LC
LS_COLORS	MACHTYPE	MAILCHECK	OLDPWD	OPTERR	OF
OSTYPE	PATH	PIPESTATUS	PPID	PROMPT_COMMAND	PS
PS2	PS4	PS3	PWD	SHELL	SH
SHLVL	TERM	UID	USER	USERNAME	XA



Этот список неполный. **Самый безопасный способ - использовать все имена переменных в нижнем регистре.**

Коды выхода

Каждая программа, которую вы запускаете, завершается кодом выхода и сообщает об этом операционной системе. Этот код выхода может быть использован Bash. Вы можете показать это, вы можете действовать на нем, вы можете управлять потоком сценариев с его помощью. Код представляет собой число от 0 до 255. Значения от 126 до 255 зарезервированы для использования непосредственно оболочкой или для специальных целей, таких как сообщение о завершении с помощью сигнала:

- **126**: запрошенная команда (файл) найдена, но не может быть выполнена
- **127**: команда (файл) не найдена
- **128**: согласно ABS, он используется для сообщения недопустимого аргумента встроенному выходу, но я не смог проверить это в исходном коде Bash (см. Код 255)
- **128 + N**: оболочка была завершена сигналом N
- **255**: неверный аргумент для встроенного выхода (см. Код 128)

Младшие коды от 0 до 125 не зарезервированы и могут использоваться для любых сообщений, которые программа хочет сообщить. Значение 0 означает **успешное** завершение, значение, отличное от 0, означает **неудачное** завершение. На это поведение (`== 0`, `!= 0`) также реагирует Bash в некоторых операторах управления потоком.

Пример использования кода выхода программы `grep` для проверки, присутствует ли конкретный пользователь в `/etc/passwd`:

```
grep  if ^root /etc/passwd; затем
      echo "Корень пользователя найден"
,  иначе
      echo "Корень пользователя не найден"
fi
```

Обычная команда для принятия решений - `" test "` или ее эквивалент `" ["`. Но обратите внимание, что при вызове `test` с именем `" ["` квадратные скобки не являются частью синтаксиса оболочки, левая скобка - **это** команда `test` !

```
" $mystring " [if = "Привет, мир" ]; затем
      повторите "Да, чувак, ты ввел правильные слова ... "
,  иначе
      повторите "Ииик - уходи ... "
фи
```

Подробнее о команде `test`

Обычный метод проверки кода выхода использует операторы `" || "` или `" && "`. Это позволяет вам выполнить команду в зависимости от того, успешно завершилась предыдущая команда или нет:

```
grep ^root: /etc/passwd >/dev/null || echo "корень не найден - провер
ьте паб на углу".
какой vi && echo "Установлен ваш любимый редактор".
```

Пожалуйста, когда ваш скрипт завершается с ошибками, укажите "ЛОЖНЫЙ" код выхода, чтобы другие могли проверить выполнение скрипта.

Комментарии

В более крупном или сложном сценарии целесообразно комментировать код. Комментарии могут помочь в отладке или тестировании. Комментарии начинаются с символа `#` (хэш-метки) и продолжаются до конца строки:

```
#!/bin/bash
# Это небольшой скрипт, чтобы что-то сказать.
echo "Будьте либеральны в том, что вы принимаете, и консервативны в т
ом, что вы отправляете" # скажи что-нибудь
```

Первое, что уже было объяснено, это так называемый shebang, для оболочки, **только комментарий**. Второй комментарий - это комментарий с начала строки, третий комментарий начинается после допустимой команды. Все три синтаксически корректны.

Блокировать комментирование

Чтобы временно отключить полные блоки кода, вам обычно приходится ставить перед каждой строкой этого блока `#` (хэш-метку), чтобы сделать его комментарием. Есть небольшая хитрость, использующая псевдокоманду `:` (двоеточие) и перенаправление ввода. `:` Ничего не делает, это псевдокоманда, поэтому ее не волнует стандартный ввод. В следующем примере кода вы хотите протестировать почту и ведение журнала, но не дампы базы данных или выполнить завершение работы:

```
|
"Запрошена остановка системы"
echo # Напишите информационные письма, выполните некоторые задачи и б
езопасно отключите систему #!/bin / bash mail -s "Остановка систем
ы" netadmin@example.com
logger -t SYSHALT "Запрошена остановка системы"

##### Следующий "блок кода" фактически игнорируется
: <<"SOMEWORD"
/etc/init.d/mydatabase clean_stop
mydatabase_dump /var/db/db1 /mnt/fsrv0/backup/db1
logger -t СИСТЕМНЫЙ СБОЙ "Остановка системы: действия перед завершени
ем работы выполнены, теперь система завершает работу" shutdown
-h NOW
НЕКОТОРОЕ СЛОВО
##### Игнорируемый кодовый блок заканчивается здесь
```

Что произошло? : Псевдокоманде был предоставлен некоторый ввод путем перенаправления (здесь-document) - псевдокоманда не заботилась об этом, фактически, весь блок был проигнорирован.

Здесь приведен тег here-document-tag, **чтобы избежать подстановок** в "прокомментированном" тексте! Проверьте перенаправление с помощью here-documents для получения дополнительной информации

Область видимости переменной

В Bash область действия пользовательских переменных обычно является *глобальной*. Это означает, что **не** имеет значения, установлена ли переменная в "основной программе" или в "функции", переменная определяется везде.

Сравните следующие *эквивалентные* фрагменты кода:

```
myvariable=тестовое
echo $myvariable
```

```
myfunction() {
    myvariable=test
}

myfunction
echo $myvariable
```

В обоих случаях переменная `myvariable` устанавливается и доступна отовсюду в этом скрипте, как в функциях, так и в "основной программе".

Внимание: когда вы устанавливаете переменные в дочернем процессе, например, в *подоболочке*, они будут установлены там, но у вас никогда **не** будет доступа к ним за пределами этой подоболочки. Одним из способов создания подоболочки является канал. Все это упоминается в небольшой статье о Bash в *processtree*!

Локальные переменные

Bash предоставляет способы сделать область видимости переменной *локальной* для функции:

- Используя `local` ключевое слово или
- Использование `declare` (которое *определяет*, когда оно было вызвано из функции, и делает переменные локальными).

```
myfunc() {
    local var=ЗНАЧЕНИЕ

    # альтернатива, только при использовании ВНУТРИ функции
    объявляет var=VALUE

    ...
}
```

Ключевое слово *local* (или объявление переменной с помощью `declare` команды) помечает переменную, которая должна обрабатываться *полностью локально и отдельно* внутри функции, в которой она была объявлена:

```
foo= внешний

printvalue() {
  local foo=internal

  echo $foo
}

# это выведет "внешнее"
echo $foo

# это выведет "внутреннее"
значение printvalue

# это приведет к повторной печати "внешнего"
echo $foo
```

Переменные среды

Пространство среды напрямую не связано с темой о области видимости, но его стоит упомянуть.

Каждый процесс UNIX® имеет так называемую *среду*. Там сохраняются другие элементы, помимо переменных, так называемые *переменные окружения*. Когда создается дочерний процесс (в Bash, например, путем простого выполнения другой программы, скажем `ls`, для перечисления файлов), вся среда, *включая переменные среды*, копируется в новый процесс. Чтение этого с другой стороны означает: **в дочернем процессе доступны только переменные, которые являются частью среды**.

Переменная может быть помечена как часть среды с помощью `export` команды:

```
myvariable
# -> Это обычная переменная оболочки, а не переменная среды!
# создайте новую переменную и установите ее: "Привет, мир".

# сделайте переменную видимой для всех дочерних процессов:
# -> Сделайте ее переменной среды: "экспортируйте" она
экспортирует myvariable
```

Помните, что *экспортируемая* переменная является **копией**. Не предусмотрено "копировать его обратно в родительский файл". Смотрите статью о Bash в дереве процессов!

¹⁾ при определенных обстоятельствах, также самой оболочкой

📄 scripting/basics.txt 📅 Последнее изменение: 2019/08/30 09:07 автор ersen

Этот сайт поддерживается Performing Databases - вашими
экспертами по администрированию баз данных

Bash Hackers Wiki



Если не указано иное, содержимое этой вики лицензируется по следующей лицензии:
Лицензия GNU Free Documentation 1.3