Отладка скрипта

Эти несколько строк предназначены не как полноценное руководство по отладке, а как подсказки и комментарии по отладке скрипта Bash.

Используйте уникальное имя для вашего скрипта

Например, **не** называйте свой скрипт `test` ! *Почему?* `test` это имя UNIX®-команды, и, скорее всего, оно встроено в вашу оболочку (оно встроено в Bash), поэтому вы не сможете запустить скрипт с этим именем `test` обычным способом.

Не смейтесь!Это классическая ошибка

Прочитайте сообщения об ошибках

Многие люди заходят в IRC () и спрашивают что-то вроде *"Почему мой скрипт терпит неудачу? Я получаю сообщение об ошибке! "*. И когда вы спрашиваете их, что это за сообщение об ошибке, они даже не знают. Красивые.

Чтение и интерпретация сообщений об ошибках - это 50% вашей работы в качестве отладчика! Сообщения об ошибках на самом деле что-то **значат**. По крайней мере, они могут дать вам подсказки относительно того, с чего начать отладку. **ПРОЧИТАЙТЕ ВАШИ СООБЩЕНИЯ ОБ ОШИБКАХ!**

Вы можете спросить себя, почему это упоминается как совет по отладке? Ну, вы были бы удивлены, сколько пользователей оболочки игнорируют текст сообщений об ошибках!Когда я найду время, я вставлю сюда 2 или 3 фрагмента журнала IRC (), просто чтобы показать вам этот досадный факт.

Используйте хороший редактор

Выбор редактора зависит от личных предпочтений, но настоятельно рекомендуется редактор с подсветкой **синтаксиса Bash**! Подсветка синтаксиса помогает увидеть (как вы уже догадались) синтаксические ошибки, такие как незакрытые кавычки и фигурные скобки, опечатки и т. Д.

Исходя из моего личного опыта, я могу предложить `vim` или `GNU emacs` .

Запись файлов журнала

Для более сложных скриптов полезно записывать в файл журнала или в системный журнал. Никто не сможет отладить ваш скрипт, не зная, что на самом деле произошло и что пошло не так.

Доступный интерфейс системного `logger` журнала (интерактивная страница руководства (http://unixhelp.ed.ac.uk/CGI/man-cgi?logger+1)).

Внедрить отладочный код

Вставляйте **эхо**-сигналы везде, где только можете, и печатайте в `stderr` :

```
echo "DEBUG: текущий i = $i" > &2
```

Если вы читаете ввод из **любого** места, например, из файла или подстановки команд, распечатайте вывод отладки с буквальными кавычками, чтобы увидеть начальные и конечные пробелы!

```
pid=$(< fooservice.pid)
echo "ОТЛАДКА: чтение из файла: pid= \"$ pid \"" > &2
```

Команда printf в Bash имеет `%q` формат, который удобен для проверки того, являются ли строки тем, чем они кажутся.

```
foo=$(< входной файл)
printf "DEBUG: foo is |%q|\n" "$foo" >&2
# предоставляет пробелы (например, ссылки, см. Ниже) и непечатаемые с
имволы
```

Используйте вывод отладки оболочки

Для этой задачи есть два полезных вывода отладки (оба записываются в `stderr`):

- `set -v` режим (`set -o verbose`)
  - выводите команды для выполнения `stderr` так, как если бы они были прочитаны из ввода (файла скрипта или клавиатуры)
  - распечатайте все, **прежде** чем будут применены какие-либо (замена и расширение, ...)
- `set -x` режим (`set -o xtrace`)
  - распечатайте все, как если бы оно было выполнено, после применения замены и расширения
  - укажите уровень глубины подоболочки (по умолчанию `+` , добавляя знак (плюс) к отображаемой команде)
  - укажите распознанные слова после разделения слов, пометив их как `'x y'`
  - в версии оболочки 4.1 этот отладочный вывод можно распечатать в настраиваемый файловый дескриптор, а не в sdtout, установив переменную BASH_XTRACEFD.

Подсказка: эти режимы можно вводить при вызове Bash:

- из командной строки: `bash -vx ./myscript`
- от shebang (зависит от операционной () системы): `#!/bin/bash -vx`

Простой пример того, как интерпретировать вывод xtrace

Вот простая команда (сравнение строк с использованием классической тестовой команды), выполняемая в `set -x` режиме:

```
set -x
foo="bar baz"
[ $foo = тест]
```

Это не удается. Почему? Давайте посмотрим на `xtrace` результат:

```
+ '[' bar baz = test ']'
```

И теперь вы видите, что он ("bar" и "baz") распознается как два отдельных слова (что вы бы поняли, если БЫ ПРОЧИТАЛИ СООБЩЕНИЯ ОБ ОШИБКАХ ;)). Давайте проверим это…

```
# следующая попытка
[ "$foo" = тест]
```

`xtrace` теперь дает

```
+ '[' 'bar baz' = тест ']'
 ^ ^
маркеры слов!
```

Как сделать xtrace более полезным

(автор: AnMaster)

`xtrace` вывод был бы более полезным, если бы он содержал исходный файл и номер строки. Добавьте это назначение PS4 в начало вашего скрипта, чтобы включить эту информацию:

```
экспорт PS4='+(${BASH_SOURCE}:${LINENO}): ${FUNCNAME[0]:+${FUNCNAME
[0]}(): }'
```

Обязательно используйте здесь одинарные кавычки!

Вывод будет выглядеть так, когда вы отслеживаете код *вне функции*:

```
+(somefile.bash:412): echo 'Привет, мир'
```

... и вот так, когда вы отслеживаете код *внутри функции*:

```
+(somefile.bash:412): myfunc(): echo 'Привет, мир'
```

Это очень помогает, когда скрипт длинный или когда основной скрипт использует много других файлов.

Установите переменные флага с описательными словами

Если вы тестируете переменные, которые помечают состояние параметров, например with `if [[ -n $option ]];` , рассмотрите возможность использования описательных слов, а не коротких кодов, таких как 0, 1, Y, N , потому что xtrace будет отображаться `[[ -n word ]]` , а не `[[ -n 1 ]]` когда параметр установлен.

Команды отладки в зависимости от заданной переменной

Для общих целей отладки вы также можете определить функцию и переменную для использования:

```
debugme() {
 [[ $script_debug = 1 ]] && "$@" || :
 # обязательно добавьте || : или || true здесь или используйте return
0 , поскольку код возврата
 # этой функции всегда должно быть 0, чтобы не влиять ни на что друго
е с нежелательным
 # код возврата "false" (например, код выхода скрипта, если используе
тся эта функция
 # как самая последняя команда в скрипте)
}
```

Эта функция ничего не делает, когда `script_debug` она не установлена или пуста, но она выполняет заданные параметры как команды, когда `script_debug` она установлена. Используйте его следующим образом:

```
script_debug=1
# чтобы отключить его, установите script_debug=0

debugme logger "Сортировка базы данных"
database_sort
debugme logger "Закончил сортировку базы данных, код выхода $?"
```

Конечно, это можно использовать для выполнения чего-то другого, кроме echo, во время отладки:

```
debugme set -x
# ... некоторый код ...
debugme set +x
```

Команды, управляемые STDIN, запускаются без ошибок

Представьте, что у вас есть скрипт, который запускает команды FTP () с использованием стандартного FTP ()-клиента:

```
пользователь ftp@host <журнал удаления текущего.log FTP
```

Метод сухого запуска этого с выводом отладки:

```
$DRY_RUN [[ if = yes ]]; затем
   sed /^/DRY_RUN FTP: /'
else
   ftp user@host
fi <журнал удаления текущего.log FTP
```

Это может быть обернуто в функцию оболочки для более читаемого кода.

Распространенные сообщения об ошибках

Неожиданный конец файла

```
script.sh : строка 100: синтаксическая ошибка: неожиданный конец файл
a
```

Обычно указывает именно то, что он говорит: неожиданный конец файла. Это неожиданно, потому что Bash ожидает закрытия составной команды:

- вы закрыли свой `do` с помощью а `done` ?
- вы закрыли свой `if` с помощью а `fi` ?
- вы закрыли свой `case` с помощью а `esac` ?
- вы закрыли свой `{` с помощью а `}` ?
- вы закрыли свой `(` с помощью а `)` ?

Примечание: Похоже, что here-documents (проверенные на версиях `1.14.7` , `2.05b` , `3.1.17` и `4.0`) корректно завершаются, когда перед тегом end-of-here-document есть EOF () (см. Перенаправление). Причина неизвестна, но, похоже, она преднамеренная. В Bash 4.0 добавлено дополнительное сообщение для этого: `warning: here-document at line <N> delimited by end-of-file (wanted` `<MARKER>')`

Неожиданный конец файла при поиске соответствия…

```
script.sh : строка 50: неожиданный EOF при поиске совпадающего `"'
script.sh : строка 100: синтаксическая ошибка: неожиданный конец файл
a
```

Это указывает на то, что двойная кавычка, открытая в строке 50, не имеет соответствующей закрывающей кавычки.

Эти *несопоставимые ошибки* возникают при:

- пары двойных кавычек
- пары одинарных кавычек (также `$'string'` !)
- отсутствует `}` синтаксис закрытия с расширением параметра

Слишком много аргументов

```
bash: тест: слишком много аргументов
```

Скорее всего, вы где-то забыли указать расширение переменной. См. Пример `xtrace` вывода выше. Внешние команды могут отображать такое сообщение об ошибке, хотя в нашем примере ошибка была вызвана **внутренней** тестовой командой.

!": событие не найдено

```
$ echo "Привет, мир!"
bash: !": событие не найдено
```

Это не ошибка как таковая. Это происходит в интерактивных оболочках, когда включено расширение истории в стиле C-Shell (" `!searchword` "). Это значение по умолчанию. Отключите его следующим образом:

```
установить +H
# или
set +o гистограмма расширения
```

синтаксическая ошибка рядом с неожиданным токеном `('

Когда это происходит во время **определения функции** скрипта или в командной строке, например

```
$ foo () { echo "Привет, мир"; }
bash: синтаксическая ошибка возле неожиданного токена `('
```

скорее всего, у вас есть псевдоним, определенный с тем же именем, что и у функции (здесь : `foo`). Расширение псевдонима происходит до интерпретации реального языка, поэтому псевдоним расширяется и делает ваше определение функции недействительным.

Проблема CRLF

В чем проблема с CRLF?

Существует большая разница в том, как UNIX® и Microsoft® (и, возможно, другие) обрабатывают **окончания строк** обычных текстовых файлов. Разница заключается в использовании символов CR (возврат каретки) и LF (перевод строки).

- MSDOS использует: `\r\n` (ASCII () `CR` # 13 `^M` , ASCII () LF # 10)
- UNIX® использует: `\n` (ASCII () `LF` #10)

Имейте в виду, что ваш скрипт представляет собой **обычный текстовый файл**, и `CR` символ не означает ничего особенного для UNIX® - он обрабатывается как любой другой символ. Если он напечатан на вашем терминале, возврат каретки эффективно поместит курсор в начало *текущей* строки. Это может вызвать много путаницы и много головной боли, поскольку строки, содержащие CRS, не являются тем, чем они кажутся при печати. В целом, CRS - это боль.

Как CR оказался в моем файле?

Некоторые возможные источники CRS:

- текстовый редактор DOS / Windows
- текстовый редактор UNIX®, который "слишком умен" при определении типа содержимого файла (и думает, что "*это текстовый файл DOS*")
- прямое копирование и вставка с определенных веб-страниц (некоторые pastebins известны этим)

Почему CRS вредят?

CRS может быть неприятным по-разному. Они особенно плохи, когда присутствуют в shebang / интерпретаторе, указанном `#!` в самой первой строке скрипта. Рассмотрим следующий скрипт, написанный с помощью текстового редактора Windows® (`^M` это символическое представление `CR` символа возврата каретки!):

```
#!/bin/bash^M
^M
эхо "Привет, мир" ^M
...
```

Вот что происходит из-за `#!/bin/bash^M` в нашем shebang:

- файл `/bin/bash^M` не существует (надеюсь)
- Итак, Bash выводит сообщение об ошибке, которое (в зависимости от терминала, версии Bash или пользовательских исправлений!) Может или не может выявить проблему.
- сценарий не может быть выполнен

Сообщение об ошибке может быть разным. Если вам повезет, вы получите:

```
bash: ./testing.sh : /bin/bash^M: плохой интерпретатор: нет такого фа
йла или каталога
```

который предупреждает вас о CR. Но вы также можете получить следующее:

```
: плохой интерпретатор: нет такого файла или каталога
```

Почему? Потому что при буквальном `^M` вводе курсор возвращается к началу строки. *Выводится* все сообщение об ошибке, но вы *видите* только его часть!

Легко представить `^M` , что это плохо и в других местах. Если вы получаете странные и нелогичные сообщения от вашего скрипта, исключите такую возможность `^M` . Найдите и устраните его!

Как я могу найти и устранить их?

Для отображения CRS (это всего лишь несколько примеров)

- в VI / VIM: `:set list`
- с помощью `cat(1)` : `cat -v FILE`

Чтобы устранить их (только несколько примеров)

- слепо с `tr(1)` : `tr -d '\r' <FILE >FILE.new`
- управляется с помощью `recode(1)` : `recode MSDOS..latin1 FILE`
- управляется с помощью `dos2unix(1)` : `dos2unix FILE`

Смотрите также

- встроенная команда set (для `-v` и `-x`)

Fix Me!

- Ловушка ОТЛАДКИ
- Отладчик BASH http://bashdb.sourceforge.net / (http://bashdb.sourceforge.net/)

Обсуждение

авкосинский (http://sourceforge.net/projects/basheclipse/), 2011/10/18 13:20 ()

Отладчик для Bash версии 3 (оболочка Bourne again). Плагин для Eclipse. http://sourceforge.net/projects/basheclipse / (http://sourceforge.net/projects/basheclipse/)

Paolo Supino, 2012/01/02 09:20 (), 2012/01/02 19:13 ()

не зная о существовании отладчика bash, я написал небольшой скрипт (я вызвал debug.sh), который устанавливает -x, -xv или -xvn (в зависимости от переданного параметра debug.sh). Отладка.сценарий sh (не стесняйтесь копировать, использовать и развивать его по своему усмотрению):

```
PS4

#!/bin/bash='+(${BASH_SOURCE}:${LINENO}): ${FUNCNAME[0]:+${FUNCNA
ME[0]}(): }'

экспортировать PS4

использование()
{
cat <<'EOF' использование: отладка [опция] аргументы скрипта возм
ожные варианты: - справка |использование: распечатать этот экран
 - подробный: устанавливает -xv флагов - noexec: устанавливает -x
vn флагов - нет наборов параметров -x флагов EOF

 fmt << 'EOF', еслискрипт принимает аргументы, не забудьте вложит
ь скрипт и исправления в "" EOF
}

debug_cmd()
{
   /bin/bash $FLAGS $SCRIPT
}

если [ $# -gt  0 ]; тогда
    регистр  "$1"  в
          "подробном")
            ФЛАГИ=-xv
 SCRIPT=$2
         ;;
         "noexec")
            ФЛАГИ=-xvn
 SCRIPT= $2
         ;;
         "справка" |"использование")

выход  использования 3
         ;;
         *)
            ФЛАГИ=-x
 SCRIPT=$1
         ;;
   esac
debug_cmd
другое
использование
fi
```

Paolo Supino, 2012/06/22 09:12 (), 2012/06/22 17:51 ()

Я обновил отладку.sh опубликовал выше, чтобы отображать цвета в выводе при использовании только + x (делает все более читаемым). новый скрипт:

```
#!/bin/bash

color_def="~/.colorrc"

if [[ -f $color_def ]]; then
   . $color_def
else
   # color definitions
   black="$(tput setaf 0)"
   darkgrey="$(tput bold ; tput setaf 0)"
   lightgrey="$(tput setaf 7)"
   white="$(tput bold ; tput setaf 7)"
   red="$(tput setaf 1)"
   lightred="$(tput bold ; tput setaf 1)"
   green="$(tput setaf 2)"
   lightgreen="$(tput bold ; tput setaf 2)"
   yellow="$(tput setaf 3)"
   blue="$(tput setaf 4)"
   lightblue="$(tput bold ; tput setaf 4)"
   purple="$(tput setaf 5)"
   pink="$(tput bold ; tput setaf 5)"
   cyan="$(tput setaf 6)"
   lightcyan="$(tput bold ; tput setaf 6)"
   nc="$(tput sgr0)" # no color
fi
export darkgrey lightgreywhite red lightred green lightgreen yell
ow blue
export lightblue purple pink cyan lightcyan nc
if [[ ! $level_color ]]; then
   level_color=$cyan
fi
if [[ ! $script_color ]]; then
   script_color=$yellow
fi
if [[ ! $linenum_color ]]; then
   linenum_color=$red
fi
if [[ ! $funcname_color ]]; then
   funcname_color=$green
fi
if [[ ! $command_color ]]; then
   command_color=$white
fi
export script_color linenum_color funcname_color

reset_screen() {

   echo $nc
}
reset_screen

usage()
{
cat }
<: белый '$ nc EOF  cat
```

```
< Различные части приглашения скрипта напечатаны в цвете. Если цв
ета по умолчанию не подходят, вы можете задать для переменных сре
ды script_color linenum_color funcname_color любой из следующих ц
ветов: ${darkgrey}darkgrey $ nc, ${lightgrey}светло-серый $ nc,
 ${white}белый, ${red}красный, ${lightred}светло-красный,${green}
зеленый, ${lightgreen}светло-зеленый, ${yellow}желтый, $ {blue}си
ний, $ {lightblue} светло-голубой, $ {purple}фиолетовый, $ {pink}
розовый, $ {cyan}голубой, $ {lightcyan} светло-голубой $nc. EOF
 fmt

<
флаги - нет наборов параметров -x флагов EOF


debug_cmd()
{
   trap reset_screen INT
/bin/bash $FLAGS $SCRIPT
}

если [ $# -gt  0 ]; ФЛАГИ
   )  "подробный"  в
          случае "$ 1"
            , тогда =-xv
 SCRIPT=$2
         ;;
         "noexec")
            ФЛАГИ=-xvn
 SCRIPT= $2
         ;;
         "справка" |"использование")

выход  использования 3
         ;;
         *)
            ФЛАГИ=-x
 PS4="${level_color}+${script_color}"'(${BASH_SOURCE##*/}:${linen
um_color}${LINENO}${script_color}):'" $ {funcname_color}"
            экспорт PS4
 SCRIPT=$1
         ;;
   esac
debug_cmd
другое
использование
fi

reset_screen
```

Paolo Supino, 2012/07/04 07:29 ()

My last debug.sh version had a couple of bugs: 1: it didn't print the function name when run with only -x flag. 2: The first line of prompt had a different coloring scheme than the rest of the lines… In the version below I fixed those problems. Please delete my previous version and post this one instead. Thanx.

#!/bin/bash

color_def="~/.colorrc"

if -f $color_def; then

```
. $color_def
```

else

```
# color definitions
black="$(tput setaf 0)"
darkgrey="$(tput bold ; tput setaf 0)"
lightgrey="$(tput setaf 7)"
white="$(tput bold ; tput setaf 7)"
red="$(tput setaf 1)"
lightred="$(tput bold ; tput setaf 1)"
green="$(tput setaf 2)"
lightgreen="$(tput bold ; tput setaf 2)"
yellow="$(tput setaf 3)"
blue="$(tput setaf 4)"
lightblue="$(tput bold ; tput setaf 4)"
purple="$(tput setaf 5)"
pink="$(tput bold ; tput setaf 5)"
cyan="$(tput setaf 6)"
lightcyan="$(tput bold ; tput setaf 6)"
nc="$(tput sgr0)" # no color
```

fi export darkgrey lightgreywhite red lightred green lightgreen yellow blue export lightblue purple pink cyan lightcyan nc if ! $lc; then

```
lc=$cyan
```

fi if ! $sc; then

```
sc=$yellow
```

fi if ! $lnc; then

```
lnc=$red
```

fi if ! $fc; then

```
fc=$green
```

fi if ! $cc; then

```
cc=$white
```

fi export sc lnc fc

reset_screen() {

```
echo $nc
```

} reset_screen

usage() { cat «'EOF'

usage: debug [option] script arguments

possilbe options are: - help|usage: print this screen - test|compile: sets -n flag - verbose: sets -xv flags - noexec: sets -xvn flags - no parameter sets -x flag

EOF fmt «EOF if the script takes arguments remember to enclose the script and arugments in "" EOF

fmt «EOF

The script prints the script name, script line number and function name as it executes the script. The various parts of the script prompt are printed in color. If the default colors are not suitable than you can set the environment varialbes sc lnc fc to any of the following colors: ${darkgrey}darkgrey$nc, ${lightgrey}light grey$nc, ${white}white, ${red}red, ${lightred}light red, ${green}green, ${lightgreen}light green, ${yellow}yellow, ${blue}blue, ${lightblue}light blue, ${purple}purple, ${pink}pink, ${cyan}cyan, ${lightcyan}light cyan$nc. EOF

cat «EOF

default colors are: ${lc}- shell level color: cyan ${sc}- script name: yellow ${lnc}- line number: red ${fc}- function name: green ${cc}- command executed: white EOF }

debug_cmd() {

```
trap reset_screen INT
/bin/bash $FLAGS $SCRIPT
```

}

if [$# -gt 0]; then

```
 case "$1" in
       "test"|"compile")
          FLAGS=-n
          SCRIPT=$2
       ;;
       "verbose")
          FLAGS=-xv
          SCRIPT=$2
       ;;
       "noexec")
          FLAGS=-xvn
          SCRIPT=$2
       ;;
       "help"|"usage")
          usage
          exit 3
       ;;
       *)
          FLAGS=-x
          PS4="${white}${lc}+${sc}"'(${BASH_SOURCE##*/}'":${ln
c}"'${LINENO}'"${sc}): ${fc}"'${FUNCNAME[0]}'"(): ${cc}"
          export PS4
          SCRIPT=$1
       ;;
 esac
 debug_cmd
```

else

```
 usage
```

fi

reset_screen

Robert Wlaschin, 2012/09/27 06:08 ()

This is a great article. I have a suggestion for putting in DEBUG switches.

Putting a line like the following:

debug switch [-z "$DEBUG"] && DEBUG=0 || :

…

[$DEBUG = 0] || echo "Debug output"

Will allow passing in a value through environment variables, meaning that code does not need to be changed or put in unnecessary command-line arguments to enable debugging

This is how it could be used

turn on debugging DEBUG=1 ./runtest

regular no debug ./runtest

Thoughts?

scripting/debuggingtips.txt Last modified: 2017/06/07 02:42 by fgrose

Bash Hackers Wiki