Написание сценариев со стилем

Fix Me! продолжить

Это некоторые рекомендации по кодированию, которые помогли мне читать и понимать мой собственный код на протяжении многих лет. Они также помогут создать код, который будет немного более надежным, чем "если что-то сломается, я знаю, как это исправить".

Конечно, это не библия. Но я видел так много уродливого и ужасного кода (не только в оболочке) за все эти годы, что я на 100% убежден, что должен быть *какой*-то макет кода и стиль. Независимо от того, какой из них вы используете, используйте его во всем коде (по крайней мере, не меняйте его в одном файле shellscript); не меняйте макет кода в зависимости от вашего настроения.

Хорошая компоновка кода поможет вам через некоторое время прочитать ваш собственный код. И, конечно, это помогает другим читать код.

Рекомендации по отступам

Отступы - это не то, что технически влияет на сценарий, это только для нас, людей.

Я привык видеть / использовать отступ из *двух пробелов* (хотя многие могут предпочесть 4 пробела, см. Ниже В разделе обсуждения):

- его легко и быстро печатать
- это не жесткая вкладка, которая отображается по-разному в разных средах
- он достаточно широкий, чтобы создать визуальный разрыв, и достаточно маленький, чтобы не тратить слишком много места на строке

Говоря о жестких вкладках: избегайте их, если это возможно. Они только создают проблемы. Я могу представить один случай, когда они полезны: отступы здесь-documents .

Разбиение строк

Всякий раз, когда вам нужно разбить строки длинного кода, вы должны следовать одному из этих двух правил:

Отступ с использованием ширины команды:

```
активируйте some_very_long_option \
some_other_option
```

Отступ с использованием двух пробелов:

```
активируйте some_very_long_option \
some_other_option
```

Лично я, за некоторыми исключениями, предпочитаю первую форму, потому что она поддерживает визуальное впечатление "они принадлежат друг другу".

Взлом составных команд

Составные команды формируют структуры, которые отличают сценарий оболочки от глупого перечисления команд. Обычно они содержат своего рода "голову" и "тело", которые содержат списки команд. Этот тип составных команд относительно легко сделать отступом.

Я привык (не все пункты применимы ко всем составным командам, просто выберите основную идею):

- поместите вводное ключевое слово и начальный список команд или параметров в одну строку ("head")
- поместите ключевое слово "body-introducing" в ту же строку
- список команд "тела" в отдельных строках с отступом в два пробела
- поместите ключевое слово закрытия в разделенную строку с отступом, как у начального вводного ключевого слова

Что?! Ну, вот опять:

Символический

```
Параметры HEAD_KEYWORD; BODY_BEGIN
 BODY_COMMANDS
BODY_END
```

если/то/elif/else

Эта конструкция немного особенная, потому что в ней есть ключевые слова (`elif` , `else`) "посередине". Визуально привлекательный способ - сделать отступы следующим образом:

```
если ...; тогда
 ...
elif ...; затем
 ...
ещё
 ...
fi
```

для

```
для f в /etc/*; выполните
 ...
Выполнено
```

пока / пока

```
в то время как [[ $answer != [YyNn] ]]; делать
 ...
Выполнено
```

Конструкция case

`case` Возможно, здесь потребуется более подробное обсуждение конструкции, поскольку ее структура немного сложнее.

В общем, каждый новый "слой" получает новый уровень отступа:

```
случай ввода $ в
привет)
 эхо "Ты поздоровался"
 ;;
пока)
 эхо "Ты сказал пока"
 если foo; тогда
bar
fi
 ;;
 *)
эхо "Ты сказал что-то странное ..."
 ;;
esac
```

Некоторые заметки:

- если не требуется 100%, необязательная левая скобка в шаблоне не используется
- шаблоны (`hello)`) и соответствующий ограничитель действия (`;;`) имеют отступы на одном уровне
- списки команд действий имеют отступы еще на один уровень (и продолжают иметь свои собственные отступы, если это необходимо).
- хотя это необязательно, дается самый последний терминатор действия

Рекомендации по синтаксису и кодированию

Загадочные конструкции

Загадочные конструкции, мы все их знаем, мы все их любим. Если они не нужны на 100%, избегайте их, поскольку никто, кроме вас, не сможет их расшифровать.

Это - как и в C - золотая середина между умным, эффективным и читабельным.

Если вам нужно использовать загадочную конструкцию, включите комментарий, объясняющий, что делает ваш "монстр".

Имена переменных

Поскольку все зарезервированные переменные являются `UPPERCASE` , самый безопасный способ - использовать только `lowercase` имена переменных. Это верно для чтения пользовательского ввода, переменных подсчета циклов и т. Д., ... (В примере: `file`)

- предпочитаю `lowercase` переменные
- если вы используете `UPPERCASE` имена, **не используйте зарезервированные имена переменных** (неполный список см. в SUS (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html#tag_(
- если вы используете `UPPERCASE` имена, добавляйте к имени уникальный префикс (`MY_` в примере ниже)

```
#!/bin/bash

# префикс 'MY_'
MY_LOG_DIRECTORY=/var/adm/

для файла в "$MY_LOG_DIRECTORY" /*; выполнить
echo "Найден файл журнала: $file"
готово
```

Инициализация переменной

Как и в C, всегда полезно инициализировать ваши переменные, однако оболочка сама инициализирует новые переменные (лучше: неустановленные переменные обычно ведут себя как переменные, содержащие нулевую строку).

Передать **переменную среды** скрипту не проблема. Если вы слепо предполагаете, что все переменные, которые вы используете в первый раз, **пусты**, любой может **ввести** содержимое в переменную, передав его через среду.

Решение простое и эффективное: **инициализируйте их**

```
my_input=""
my_array=()
my_number=0
```

Если вы делаете это для каждой используемой переменной, у вас также есть некоторая документация в коде для них.

Расширение параметров

Если вы действительно не уверены, что делаете, **цитируйте каждое расширение параметра**.

В некоторых случаях это не требуется с технической точки зрения, например

- внутри `[[ ... ]]` (кроме RHS операторов `==` , `!=` , и `=~`)
- параметр (`WORD`) в `case $WORD in ….`
- присвоение переменной: `VAR=$WORD`

Но цитирование их никогда не является ошибкой. Если вы укажете все расширения параметров, вы будете в безопасности.

Если вам нужно проанализировать параметр как список слов, вы, конечно, не можете заключать его в кавычки, например

```
список= "раз, два, три"

# вы НЕ ДОЛЖНЫ заключать в кавычки $list здесь
для слова в $list; сделайте
 ...
Выполнено
```

Имена функций

Имена функций должны быть полными `lowercase` и значимыми. Имена функций должны быть удобочитаемыми для человека. Функция с именем `f1` может быть простой и быстрой для записи, но для отладки и особенно для других людей она ничего не показывает. Хорошие имена помогают документировать ваш код без использования дополнительных комментариев.

не используйте имена команд для своих функций. например, присвоение имени скрипту или функции `test` будет конфликтовать с командой UNIX `test` .

Без крайней необходимости используйте только буквенно-цифровые символы и подчеркивание для имен функций. `/bin/ls` это допустимое имя функции в Bash, но это не очень хорошая идея.

Замена команд

Как отмечалось в статье о подстановке команд, вы должны использовать `$( … )` форму.

Если вас беспокоит переносимость, используйте форму `` ` … ` `` с обратными кавычками .

В любом случае, если другие расширения и разделение слов нежелательны, вам следует указать команду substitution !

Оценка

Ну, как говорит Грег: **"Если eval - это ответ, вы, конечно, задаете неправильный вопрос".**

Избегайте этого, если это не является абсолютно необходимым:

- `eval` может быть вашим выстрелом в шею
- скорее всего, есть и другие способы добиться того, чего вы хотите
- если возможно, переосмыслите способ работы вашего скрипта, если кажется, что вы не можете избежать `eval` этого с помощью вашего текущего метода

- если вам действительно, действительно нужно это использовать, тогда будьте осторожны и будьте уверены в том, что вы делаете

Базовая структура

Базовая структура скрипта просто гласит:

```
#!SHEBANG

CONFIGURATION_VARIABLES

FUNCTION_DEFINITIONS

ОСНОВНОЙ КОД
```

The shebang

Если возможно (я знаю, что это не всегда возможно!), Используйте shebang.

Будьте осторожны `/bin/sh` : аргумент о том, что "в Linux `/bin/sh` есть Bash", **является ложью** (и технически не имеет значения)

Для меня shebang служит двум целям:

- он определяет интерпретатор, который будет использоваться при непосредственном вызове файла сценария: если вы кодируете для Bash, укажите `bash` !
- он документирует желаемый интерпретатор (итак: используйте `bash` при написании Bash-скрипта, используйте `sh` при написании общего скрипта Bourne / POSIX, ...)

Переменные конфигурации

Здесь я называю переменные, которые должны быть изменены пользователем, "переменными конфигурации".

Сделайте так, чтобы их было легко найти (прямо в верхней части сценария), дайте им значимые имена и, возможно, короткий комментарий. Как отмечалось выше, используйте `UPPERCASE` for them только тогда, когда вы уверены в том, что делаете. `lowercase` будет самым безопасным.

Определения функций

Если нет причин не делать этого, все определения функций должны быть объявлены до запуска основного кода скрипта. Это дает гораздо лучший обзор и гарантирует, что все имена функций известны до их использования.

Поскольку функция не анализируется перед ее выполнением, вам обычно не нужно следить за тем, чтобы они были в определенном порядке.

Следует использовать переносимую форму определения функции без `function` ключевого слова (здесь используется составная команда группировки):

```
getargs() {
 ...
}
```

Говоря о группировке команд в определениях функций с помощью `{ …; }` : Если у вас нет веской причины использовать другую составную команду напрямую, вы всегда должны использовать эту.

Поведение и надежность

Сбой на ранней стадии

Сбой на ранней стадии, это звучит плохо, но обычно это хорошо. Сбой на ранней стадии означает ошибку как можно раньше, когда проверки указывают на ошибку или невыполненное условие. Сбой на ранней стадии означает ошибку **до** того, как ваш скрипт начнет свою работу в потенциально неисправном состоянии.

Доступность команд

Если вы используете внешние команды, которые могут отсутствовать в пути или не установлены, проверьте их доступность, а затем сообщите пользователю, что они отсутствуют.

Пример:

```
my_needed_commands="sed awk lsof кто"

missing_counter=0
для нужной команды в $my_needed_commands; выполняйте
, если ! хэш "$needed_command" >/dev/null 2> &1; затем
printf "Команда не найдена в PATH: %s \ n" "$needed_command" > &2
 ((missing_counter++))
 фи
готово

if ((missing_counter > 0)); тогда
printf "Минимальные команды % d отсутствуют в PATH, прерывая \n" "$mi
ssing_counter" >&2
 выход 1
fi
```

Завершите осмысленно

Код выхода - это ваш единственный способ напрямую взаимодействовать с вызывающим процессом без каких-либо специальных условий.

Если ваш скрипт завершает работу, укажите значимый код выхода. Это минимально означает:

- `exit 0` (ноль), если все в порядке
- `exit 1` - в общем случае ненулевой - если произошла ошибка

Это, **и только это**, позволит вызывающему компоненту проверять состояние работы вашего скрипта.

Вы знаете: **"Одной из главных причин падения Римской империи было то, что из–за отсутствия нуля у них не было возможности указать успешное завершение своих программ на Си". -** *Роберт Ферт*

Разное

Вывод и внешний вид

- если скрипт интерактивный, если он работает для вас, и если вы считаете, что это хорошая функция, вы можете попытаться сохранить содержимое терминала и восстановить его после выполнения
- вывод чистых и понятных экранных сообщений
- если применимо, вы можете использовать цвета или определенные префиксы для обозначения сообщений об ошибках и предупреждений
  - упростите для пользователя идентификацию этих сообщений
- напишите обычный вывод на `STDOUT` . запись сообщений об ошибках, предупреждений и диагностических сообщений в `STDERR`
  - включает фильтрацию сообщений
  - предотвращает смешивание выходных данных скрипта с диагностическими данными или сообщениями об ошибках
  - если скрипт предоставляет синтаксическую справку (`-?` или `-h` или `–help` аргументы), он должен перейти к `STDOUT`
- если применимо, запишите сообщения об ошибках / диагностики в файл журнала
  - позволяет избежать загромождения экрана
  - сообщения доступны для диагностического использования

Ввод

- никогда не принимайте что-либо слепо. Если вы хотите, чтобы пользователь вводил число, **проверьте наличие числового ввода, начальных нулей** и т. Д. Если у вас есть определенные потребности в формате или контенте, **всегда проверяйте ввод!**

Инструментарий

- некоторые из этих рекомендаций, такие как отступы, расположение ключевых слов, вводящих текст, и объявления переносимых функций, могут быть

реализованы shfmt (https://github.com/mvdan/sh)

Обсуждение

Jari Aalto (http://wiki.debian.org/JariAalto), 2010/11/09 18:33 ()

О количестве отступов: отраслевой стандарт в значительной степени равен 4, что переводится как *половина табуляции*.

(1) Попробуйте настроить разрешение экрана на более высокое, вы все еще чувствуете себя комфортно с отступом? (2) Попробуйте настроить размер шрифта вашего редактора на меньшие шрифты, чтобы увеличить видимость и количество строк, отступы все еще четкие? (3) попробуйте напечатать код размером 4 страницы на одном листе, вы все еще читаете отступ кода?

Для всех этих пунктов 4 работает, у любого из них обычно возникают проблемы, когда они вырваны из контекста отдельного разработчика. С другой стороны, *половинная вкладка* без проблем адаптируется к различным средам.

Lioba, 2013/11/29 19:07 ()

Хорошо, Яри, ты, должно быть, работаешь в другой отрасли, я не могу чувствовать себя более несогласным. Похоже, вы говорите о другом другом языке программирования или сценариев. Вы должны попытаться прочитать и написать много скриптов в реальных боевых песках терминалов, прежде чем понять, что вы ошибаетесь, но, возможно, вы это сделаете. Вам действительно нужно распечатать 4 страницы сценариев bash, подходящих для одного формата A4, тогда у вас возникнут проблемы посерьезнее, чем ваш отступ, который вы ошибочно выбрали в своей работе. Почему вы так уверены, что все работают с такими редакторами, где вы можете "изменить размер моего шрифта на меньший" или "настроить разрешение экрана"? Вы используете для работы терминалы или эмуляторы терминалов? Похоже, что нет, и если да, то почему вы игнорируете, что типы шрифтов обычно моноширинные только для более четкого чтения?

Аластер Эндрю, 2011/03/28 23:26 ()

Как насчет функции Bash 4 command_not_found_handle? Реализация этого, по-видимому, является гораздо более чистым способом решения ситуации, когда команда, на которую полагаются, не может быть найдена.

Ян Шампера, 2011/03/29 05:19 (), 2011/03/29 05:29 ()

Несколько причин, которые мне не нравятся `command_not_found_handle()` в этой ситуации

- основная идея хороша, фактический способ - это уродливый взлом, ИМХО (хотя обсуждение лучшего способа, которое я имел с сопровождающим Bash, было не очень успешным)
- что касается способа его реализации и того, как он действительно работает (см. Ниже), это функция, предназначенная в первую очередь для интерактивного использования
- это означало бы сбой при использовании команды, а не в начале предполетной проверки (плохой код, ИМХО)
- сложно увидеть неудачные команды:

```
$ command_not_found_handle() { echo "Не найдено: "$@""; }
$ echa foo | grep bar | grep baz # "echa" - он не выводит
ся - он проглатывается трубой
$ echa foo | grep bar | greb baz # теперь он выводит толь
ко для "greb"
Не найдено: "greb baz"
```

  Основная причина здесь в том, что функция выполняется в той же среде, что и команда, которая должна была быть выполнена (она выполняется в канале вместо "echa"), что может потребоваться не во всех случаях

Конечно, вы можете написать сценарий вокруг третьей проблемы, используя несколько `BASH_*` переменных и правильные filedescriptors и прочее, но вы можете поймать все это в небольшом цикле, повторяя необходимые команды и сообщая пользователю, какая именно команда отсутствует.

Мартин Кили, 2012/08/06 06:46 ()

Предложенный вами макет для составных команд, хотя и соответствует другим языкам программирования (и, действительно, из-за этого обычно используется для написания сценариев оболочки), скорее упускает из виду грамматику оболочки.

В частности, две связанные точки: точка с запятой и конец строки являются терминаторами, а "головная" часть (в большинстве случаев) представляет собой список.

Поэтому, по крайней мере, в некоторых случаях имеет смысл поместить промежуточное слово в отдельную строку.

Во-первых, в цикле "while" иногда вы хотите что-то сделать * до * теста:

```
в то время
как do_this do_that
test_something
do_something_else

сделано
```

В крайнем случае, "тело" может быть даже пустым, обеспечивая эквивалент конструкции "do ... while" на языке Си:

```
в то время
как do_something
test_something
сделать :; готово
```

(Я считаю, что было бы неплохо расширить, чтобы полностью исключить "СПИСОК дел".)

Во-вторых, в цикле "for" у вас могут быть длинные элементы, которые будут более четкими, если выровнять их по вертикали:

```
для x в path/to/one/thing \
 путь / к / другой / вещи \
 путь / к / еще / другой / вещи \
 a/path/to/somewhere else
do
do_something $x
сделано
```

В этих случаях я думаю, что наличие "do" в отдельной строке делает более понятным, где заканчивается список "head" и начинается список "body".

"Head" в операторе "if" аналогично представляет собой список, и опять же имеет смысл визуально отделить составную "head" от "body" (хотя это имеет значение, только если разделено чем-то другим, кроме точки с запятой, или если используется в конвейере):

```
если X=$( generate_list |
 grep -qs "$quick_filter" ) &&
[[ $ X = "$precise_thing_to_check" ]]
затем
echo НАШЕЛ "$ X"
else
echo НЕ НАЙДЕНО
fi
```

(и да, большую часть времени я предпочитаю 4 пробела, а не 2)

Ян Шампера, 15.02.2012 17:33 ()

Я согласен.

Текст нуждается в некоторой доработке. Я знаю эти конструкции и сам их использую. Я просто не имел их в виду здесь. Текст далек от завершения, да.

Кристофер Барри, 2012/09/30 21:30 ()

RE: Отступ. Всегда липкая калитка и определенно религиозная вещь, но я обнаружил, что всегда лучше избегать реальных символов табуляции, вместо этого настраивая ваш редактор на использование пробелов для табуляции. Лично я предпочитаю табуляции с 4 пробелами.

Причина, по которой я это говорю, заключается в том, что при написании кода в моем редакторе мне нравится вырезать и вставлять функции в оболочку и выполнять их по мере уточнения функции. Этот метод рабочего процесса хорошо работает для меня, у вас может быть свой собственный путь. Дело в том, что вставка вкладок может вызвать всевозможные странные ошибки, поэтому я избегаю их, как чумы. Я также использую emacs с sh-режимом, и мне нравится, как это работает.

Я думаю, что хороший способ форматирования вашего кода - это тот же способ, которым сам bash переформатирует ваш код. Попробуйте это в оболочке:

скопируйте довольно сложную функцию из одного из ваших скриптов и вставьте ее в оболочку, чтобы создать ее экземпляр. допустим, функция называется 'my_func'. Теперь выполните следующие действия (при условии, что вкладки не взорвали его :):

$ declare -f my_func

Если результат выглядит точно так, как вы его закодировали, у вас все хорошо. Если нет, попробуйте закодировать способ, которым сам bash выводит его. Вы заметите, что в выходных данных нет символов табуляции.

Мои 2 цента.

Арилд Дженсен, 08.03.01 21:58 ()

Ссылка http://www.opensolaris.org/os/project/shell/shellstyle / (http://www.opensolaris.org/os/project/shell/shellstyle/) мертв.

Алистер, 2013/10/19 02:52 ()

> Избегайте if, если это не является абсолютно необходимым…

Должно ли это говорить "избегайте этого ..."?

Хемал Пандья, 2014/08/29 18:49 ()

Было бы полезно, если бы вы объяснили, почему `eval` плох и при каких обстоятельствах он полезен.

Кевин Бриггс, 09.07.2016 13:17 ()

Похоже, что аргумент "eval is bad" - это просто слепо повторяемая мантра, которая действительно всплывает только потому, что она *опасна*. Как и во всем, я думаю, что если вы знаете, что делаете, это *хорошо*. Это просто

зависит от перспективы. С точки зрения написания переносимого, повторно используемого кода, который может быть подобран, отредактирован и использован новичками в bash, возможно, это плохо. С точки зрения использования его экспертом по bash для выполнения некоторых сложных косвенных действий, возможно, это хорошо. В любом случае, это уникальный инструмент, который потенциально может делать то, чего не может ни один другой инструмент bash.

алекс, 07.07.2017 19:52 ()

мне было интересно то же самое :)

Zibri (http://twitter.com/Zibri), 2016/12/15 01:16 ()

В bash я делаю это:

```
 источник <(echo "Zibri () {";cat script_to_be_reindented.sh ; ec
ho "}")
 объявить -f Zibri| cut -c 5-|head --lines=-1|tail --lines=+3
```

это устраняет комментарии и переиндентирует сценарий "bash way". это не сработает, если скрипт содержит HEREDOCS, но если вы сделаете это:

```
 источник <(echo "Zibri () {";cat script_to_be_reindented.sh ; ec
ho "}")
 объявить -f Zibri|head --lines=-1|tail --lines=+3
```

он будет работать с любым скриптом, но весь скрипт будет иметь отступ в 4 пробела. не стесняйтесь вносить изменения, но укажите мое имя в своем скрипте и опубликуйте его! : D

Дэвид, 2017/04/23 21:35 ()

Как и эта страница, это очень хорошая ссылка.

В части именования функций у меня есть несколько дополнительных соглашений, которые я использую.

Для функций библиотечного типа я использую следующую форму:

```
f_array.Получить
индекс f_array.Обратная
```

Для других имен функций, специфичных для скриптов, я использую соглашение, которое, возможно, полезно для других. Я пытаюсь отделить то, что мои функции делают с:

```
f_get_IpOfDomain возвращает содержимое.
f_set_NumberOfCalls изменяет глобальную переменную.
f_do_SaveSession вносит изменения в систему или хост, больше, чем запись в /tmp.
```

И я рассматриваю следующее соглашение об именовании для меню с бесконечными циклами while:

```
f_show_MainMenu
f_show_SelectionMenu
```