Классическая тестовая команда

```
test <EXPRESSION>

[ <EXPRESSION> ]
```

Общий синтаксис

Эта команда позволяет выполнять различные тесты и устанавливает ее код выхода в 0 (*TRUE*) или 1 (*FALSE*) всякий раз, когда такой тест проходит успешно или нет. Используя этот код выхода, можно позволить Bash реагировать на результат такого теста, здесь с помощью команды в if-инструкции:

```
#!/bin/bash
# проверить, существует ли /etc/passwd

если test -e /etc/passwd; затем
повторите "Хорошо, чувак ..." > & 2
остальные
повторяют "Фу! Где это ??" > & 2
 выход 1
fi
```

Синтаксис тестовой команды относительно прост. Обычно это имя команды " `test` ", за которым следует тип теста (здесь " `-e` " для "file exists"), за которым следуют значения, зависящие от типа теста (здесь имя файла для проверки, " `/etc/passwd` ").

Есть вторая стандартизированная команда, которая делает то же самое: команда " `[` " - разница только в том, что она называется " `[` ", а последним аргументом команды должен быть " `]` ": он формирует " **`[ <EXPRESSION> ]`** "

Давайте перепишем приведенный выше пример, чтобы использовать его:

```
#!/bin/bash
# проверить, существует ли /etc/passwd

если [ -e /etc/passwd ]; затем
повторите "Хорошо, чувак ..." > & 2
остальные
повторяют "Фу! Где это ??" > & 2
 выход 1
fi
```

Теперь можно **подумать**, что эти "[" и "]" принадлежат синтаксису if-предложения Bash: **нет, это не так! Тем не менее, это простая, обычная команда!**

Еще одна вещь, которую вы должны помнить, это то, что если команде test требуется один параметр для теста, вы должны предоставить ей один параметр. Давайте проверим наличие некоторых ваших музыкальных файлов:

```
#!/bin/bash

mymusic="/data/music/Van Halen/Ван Хален - Прямо сейчас.mp3"

если [ -e "$mymusic" ]; затем
повторите "Let's rock" > &2
еще
эхо "Сегодня нет музыки, извините ..." > & 2
 выход 1
fi
```

Как вы определенно отметили, имя файла содержит пробелы. Поскольку мы вызываем обычную обычную команду ("test" или "["), оболочка разделит расширение переменной на слова `mymusic` : вам нужно заключить ее в кавычки, если вы не хотите `test` , чтобы команда -жаловалась на слишком много аргументов для этого тестового типа! Если вы этого не поняли, пожалуйста, прочтите статью о словах ...

Пожалуйста, также обратите внимание, что файловые тесты требуют для проверки **одного имени** файла. Не указывайте глобус (имя файла-подстановочные знаки), так как он может расширяться до многих имен файлов ⇒ **слишком много аргументов!**

Еще одна распространенная ошибка - слишком **мало** аргументов:

```
[ "$mystring"!="test" ]
```

Это предоставляет ровно **один** тестовый аргумент команде. С одним параметром по умолчанию `-n` используется тест: он проверяет, является ли предоставленная строка пустой (`FALSE`) или нет (`TRUE`) - из-за отсутствия **пробелов для разделения аргументов** показанная команда всегда заканчивается `TRUE` !

Что ж, я рассмотрел несколько основных правил, теперь давайте посмотрим, что тестовая команда может сделать для вас. Типы тестов Bash можно разделить на несколько разделов: **файловые тесты**, **строковые тесты**, **арифметические тесты**, **разные тесты**. Ниже тесты, отмеченные знаком(!), являются нестандартными тестами (т. Е. Не в SUS / POSIX / etc ..).

Файловые тесты

В этом разделе, вероятно, содержится большинство тестов, я перечислю их в некотором логическом порядке. Начиная с версии Bash 4.1, все тесты, связанные с разрешениями, учитывают списки управления доступом, если их поддерживает базовая файловая система / ОС ().

Синтаксис оператора	**Описание**
-a <ФАЙЛ>	True, если <FILE> существует. (!)(не рекомендуется, может столкнуться с `-a` for `AND` , см. Ниже)

Синтаксис оператора	Описание
-e <ФАЙЛ>	True, если <FILE> существует.
-f <ФАЙЛ>	Верно, если <FILE> существует и является **обычным** файлом.
-d <ФАЙЛ>	Верно, если <FILE> существует и является **каталогом**.
-c <ФАЙЛ>	Верно, если <FILE> существует и является **символьным специальным** файлом.
-b <ФАЙЛ>	Верно, если <FILE> существует и является **специальным файлом блока**.
-p <ФАЙЛ>	Верно, если <FILE> существует и является **именованным каналом** (FIFO).
-S <ФАЙЛ>	Верно, если <FILE> существует и является файлом **сокета**.
-L <ФАЙЛ>	Верно, если <FILE> существует и является **символической ссылкой**.
-h <ФАЙЛ>	Верно, если <FILE> существует и является **символической ссылкой**.
-g <ФАЙЛ>	Верно, если <FILE> существует и имеет установленный **бит sgid**.
-u <ФАЙЛ>	Верно, если <FILE> существует и имеет установленный **бит suid**.
-r <ФАЙЛ>	Верно, если <FILE> существует и доступен для **чтения**.
-w <ФАЙЛ>	Верно, если <FILE> существует и доступен для **записи**.
-x <ФАЙЛ>	Верно, если <FILE> существует и является **исполняемым**.
-s <ФАЙЛ>	Верно, если <FILE> существует и имеет размер больше 0 (**не пустой**).
-t <fd>	Верно, если дескриптор файла <fd> открыт и ссылается на терминал.
<FILE1> **-nt** <FILE2>	Верно, если <FILE1> **новее, чем** <FILE2> (mtime). !
<FILE1> **-ot** <FILE2>	Верно, если <FILE1> **старше, чем** <FILE2> (mtime). !
<FILE1> **-ef** <FILE2>	Верно, если <FILE1> и <FILE2> относятся к одному и тому **же устройству и номерам индексов**. !

Строковые тесты

Синтаксис оператора	Описание

Синтаксис оператора	Описание
-z <СТРОКА>	Верно, если <СТРОКА> **пуста**.
-n <СТРОКА>	Верно, если <STRING> **не является пустым** (это операция по умолчанию).
<STRING1> **=** <STRING2>	Верно, если строки **равны**.
<STRING1> **!=** <STRING2>	Верно, если строки **не равны**.
<STRING1> **<** <STRING2>	Верно, если <STRING1> сортируется **перед** <STRING2> лексикографически (чистый ASCII (), а не текущая локаль!). Не забудьте сбежать! Используйте `\<`
<STRING1> **>** <STRING2>	Верно, если <STRING1> сортируется **после** <STRING2> лексикографически (чистый ASCII (), а не текущая локаль!). Не забудьте сбежать! Используйте `\>`

Арифметические тесты

Синтаксис оператора	Описание
<INTEGER1> **-эквалайзер** <INTEGER2>	Верно, если целые числа **равны**.
<INTEGER1> **-ne** <INTEGER2>	Верно, если целые числа **НЕ равны**.
<INTEGER1> **-файл** <INTEGER2>	Верно, если первое целое число **меньше или равно** второму.
<INTEGER1> **-ge** <INTEGER2>	Верно, если первое целое число **больше или равно** второму.
<INTEGER1> **-lt** <INTEGER2>	Верно, если первое целое число **меньше** второго.
<INTEGER1> **-gt** <INTEGER2>	Верно, если первое целое число **больше** второго.

Различный синтаксис

Синтаксис оператора	Описание

Синтаксис оператора	Описание
<TEST1> **-a** <TEST2>	Верно, если <TEST1> **и** <TEST2> являются истинными (И). Обратите внимание, что это `-a` также может использоваться в качестве теста файла (см. Выше)
<TEST1> **-o** <TEST2>	True, если либо <TEST1>, **либо** <TEST2> имеет значение true (ИЛИ).
! <ТЕСТ>	True, если <TEST> имеет значение **false** (НЕТ).
(<ТЕСТ> **)**	Сгруппируйте тест (для приоритета). **Внимание:** при обычном использовании оболочки "(" и ")" должны быть экранированы; используйте "\(" и "\)"!
-o <ИМЯ_ОПЕРАТОРА>	Верно, если установлен параметр оболочки <ИМЯ_ОПЕРАТОРА>.
-v <ИМЯ ПЕРЕМЕННОЙ>	Верно, если установлена переменная <ИМЯ_ПЕРЕМЕННОЙ>. Используется `var[n]` для элементов массива.
-R <ИМЯ ПЕРЕМЕННОЙ>	Верно, если переменная <ИМЯ_ПЕРЕМЕННОЙ> была установлена и является переменной nameref (начиная с 4.3-alpha)

Правила количества аргументов

`test` Встроенный компонент, особенно скрытый под его `[` именем, может показаться простым, но на самом деле **иногда вызывает много проблем**. Одна из трудностей заключается в том, что поведение `test` не только зависит от его аргументов, но и от **количества его аргументов**.

Вот правила, взятые из руководства (**Примечание:** это для команды `test` , поскольку `[` количество аргументов вычисляется без окончательного `]` , например `[ ]` , следует правилу "нулевых аргументов"):

- **0 аргументов**
  - Выражение равно false.
- **1 аргумент**
  - Выражение истинно тогда и только тогда, когда аргумент не равен null
- **2 аргумента**
  - Если первый аргумент равен `!` (восклицательный знак), выражение истинно тогда и только тогда, когда второй аргумент равен null
  - Если первый аргумент является одним из унарных условных операторов, перечисленных выше в соответствии с правилами синтаксиса, выражение имеет значение true, если унарный тест имеет значение true
  - Если первый аргумент не является допустимым унарным условным оператором, выражение равно false
- **3 аргумента**
  - Если второй аргумент является одним из двоичных условных операторов, перечисленных выше в соответствии с правилами синтаксиса,

результатом выражения является результат двоичного теста с использованием первого и третьего аргументов в качестве операндов
    - Если первый аргумент равен `!`, значение является отрицанием теста с двумя аргументами, использующего второй и третий аргументы
    - Если первый аргумент равен точно `(`, а третий аргумент равен точно `)`, результатом является проверка второго аргумента с одним аргументом. В противном случае выражение равно false. В `-a -o` этом случае операторы and считаются бинарными операторами (**внимание:** это означает, что в данном случае оператор `-a` не является файловым оператором!)
- **4 аргумента**
    - Если первый аргумент равен `!`, результатом является отрицание выражения с тремя аргументами, состоящего из оставшихся аргументов. В противном случае выражение анализируется и оценивается в соответствии с приоритетом, используя правила, перечисленные выше
- **5 или более аргументов**
    - Выражение анализируется и оценивается в соответствии с приоритетом, используя правила, перечисленные выше

Эти правила могут показаться сложными, но на практике это не так уж плохо. Знание их может помочь вам объяснить некоторые "необъяснимые" поведения, с которыми вы можете столкнуться:

```
var=""
если [ -n $ var ]; тогда echo "var не пуст"; fi
```

Этот код выводит "var not empty", хотя `-n something` должен быть true, если `$var` не пусто - **почему?**

Здесь, как `$var` **не указано в кавычках**, происходит разделение слов и `$var` фактически ничего не приводит (Bash удаляет его из списка аргументов команды!). Таким образом, тест на самом деле `[ -n ]` **и подпадает под правило "один аргумент"**, единственным аргументом является "-n", который не равен null, и поэтому тест возвращает true. Решение, как обычно, заключается в том, чтобы **указать расширение параметра**: `[ -n "$var" ]` чтобы у теста всегда было 2 аргумента, даже если второй является нулевой строкой.

Эти правила также объясняют, почему, например, -a и -o могут иметь несколько значений.

И и ИЛИ

Предпочтительный способ

Часто рекомендуемый способ логического соединения нескольких тестов с AND и OR заключается в использовании **нескольких отдельных тестовых команд** и **объединении** их с `&&` операторами управления оболочкой `||` **и списком**.

Посмотрите на это:

```
если [ -n "$var"] && [ -e "$var"]; то
echo "\$var не равен нулю и файл с именем $var существует!"
fi
```

Статус возврата списков AND и OR - это статус завершения последней команды, выполненной в списке

- With `command1 && command2`, `command2` выполняется тогда и только тогда, `command1` когда возвращает нулевой статус выхода (true)
- With `command1 || command2`, `command2` выполняется тогда и только тогда, `command1` когда возвращает ненулевой статус выхода (false)

Другой способ: -a и -o

Логическими операторами AND и OR для самой тестовой команды являются `-a` and `-o`, таким образом:

```
если [ -n "$var" -a -e "$var" ] ; тогда
echo "\ $var не равно нулю и файл с именем $var существует"
fi
```

Они **не** `&&` являются или `||`:

```
$ if [ -n "/tmp" && -d "/tmp"]; тогда echo true; fi # НЕ РАБОТАЕТ
bash: [: отсутствует `]'
```

Вы можете обнаружить, что сообщение об ошибке сбивает с толку, `[` не находит требуемый финал `]`, потому что, как видно выше `&&`, используется для записи **списка команд**. `if` Оператор фактически **видит две команды**:

- `[ -n "/tmp"`
- `-d "/tmp" ]`

... которая **должна** завершиться неудачей.

Почему вам следует избегать использования -a и -o

Если проблема с переносимостью

POSIX® / SUSv3 **не** определяет поведение `test` в случаях, когда имеется более 4 аргументов. Если вы пишете скрипт, который может не выполняться с помощью Bash, поведение может быть другим! [1)]

Если вы хотите, чтобы поведение cut

Допустим, мы хотим проверить следующие две вещи (И):

1. если строка равна нулю (пуста)
2. если команда произвела вывод

Давайте посмотрим:

```
если [ -z "false" -a -z "$(echo I выполняется >&2)" ] ; тогда ...
```

⇒ Все аргументы расширяются **перед** `test` запуском, таким образом, **выполняется** эхо-команда.

```
если [ -z "false" ] && [ -z "$(echo I не выполняется> &2)" ]; тогд
а...
```

⇒ Из-за природы оператора `&&` списка вторая тестовая команда выполняется только в том случае, если первая тестовая команда возвращает true, наша эхо-команда **не выполняется**.

Примечание: На мой взгляд, `-a` и `-o` также менее читаемы `[pgas]`

Приоритет и скобки

Будьте осторожны, если вы конвертируете свои скрипты из использования `-a` и `-o` в использование способа списка (`&&` и `||`):

- в правилах тестовой команды `-a` имеет **приоритет над** `-o`
- в правилах грамматики оболочки `&&` и `||` имеют **равный приоритет**

Это означает, что **вы можете получить разные результаты** в зависимости от способа использования:

```
$ if [ "true" ] || [ -e /does/not/exist ] && [ -e /does/not/exist ];
тогда echo true; иначе echo false; fi
false

$ если [ "true" -o -e /does/ not/exist -a -e /does/not/exist ]; тогда
echo true; иначе echo false;fi
true
```

В результате вам придется немного подумать или добавить элемент управления приоритетом (скобки).

For `&&` и `||` скобки означают (оболочечно) группировку команд, и поскольку `( … )` вводится подоболочка, мы будем использовать `{ … }` вместо:

```
$ if [ "true" ] || { [ -e /does/not/exist ] && [ -e /does/not/exist ]
;} ; тогда echo true; иначе echo false; fi
true
```

Для тестовой команды скобки приоритета также являются, `( )` , но вам нужно экранировать или заключать их в кавычки, чтобы оболочка не пыталась их интерпретировать:

```
$ if [ \( "true" -o -e /does/not/exist \) -a -e /does/not/exist ]; то
гда echo true; иначе echo false; fi
false

# эквивалентно, но менее читаемо ИМХО:
$ if [ '(' "true" -o -e /does/not/exist ')' -a -e /does/not/exist ];
тогда echo true; иначе echo false; fi
false
```

НЕ

Что касается AND и OR, есть 2 способа отменить тест с помощью ключевого слова shell `!` или передать `!` в качестве аргумента `test` .

Здесь `!` отрицается статус завершения команды `test` , который равен 0 (true), и выполняется часть else:

```
если ! [ -d '/tmp' ]; тогда echo "/tmp не существует"; иначе echo "/t
mp существует"; fi
```

Здесь сама `test` команда завершается со статусом 1 (false), и также выполняется else:

```
если [ ! -d '/tmp' ]; тогда echo "/tmp не существует"; иначе echo "/t
mp существует"; fi
```

В отличие от for И и OR , оба метода for НЕ имеют идентичного поведения, по крайней мере, для выполнения одного теста.

Подводные камни обобщены

В этом разделе вы получите все упомянутые (и, возможно, больше) возможные подводные камни и проблемы в кратком изложении.

Общая информация

Вот копия письма в списке ошибок. Пользователь, задающий вопрос об использовании тестовой команды в Bash, **говорит о проблеме, с которой вы, возможно, уже сталкивались**:

```
От: (ЗАЩИЩЕНО)
Тема: опция -d не работает . . .?
Дата: Вт, 11 сен 2007 21:51:59 -0400
Чтобы: bug-bash@gnu.org

Привет всем,

У меня есть скрипт, который я пытаюсь настроить, но он продолжает соо
бщать мне
, что "[-d команда не найдена". Может кто-нибудь, пожалуйста, объясни
ть
, что с этим не так?:




#!/bin/sh

для i в $*
выполнить
{
 если [-d $ i]
 затем
повторите "$ i - это каталог! Ура!"
 else
повторяет "$i не является каталогом!"
 фи
}
Выполнено



С уважением
```

См. Проблему, связанную с используемой тестовой командой (другие потенциальные проблемы здесь не интересны)?

```
[-d $ i]
```

Он просто не знал этого `test` или `[` это обычная, простая команда. Ну, вот ответ, который он получил. Я цитирую ее здесь, потому что это хорошо написанный текст, в котором рассматриваются большинство распространенных проблем с "классической" тестовой командой:

```
От: Боб Проулкс (EMAIL PROTECTED)
Тема: Опция Re: -d не работает . . .?
Дата: Ср, 12 сен 2007 10:32:35 -0600
Чтобы: bug-bash@gnu.org

> (ТЕКСТ В КАВЫЧКАХ БЫЛ УДАЛЕН)

Оболочка - это прежде всего способ запуска других команд.
Синтаксис - это просто "if", за которым следует список команд (наприм
ер, if /some/foo;
или даже если cmd1; cmd2; cmd3; тогда). Кроме того, синтаксис '( ...
)'
уже используется при запуске подоболочки.

Насколько я помню, в оригинальном языке оболочки оператор проверки фа
йлов
не был встроен. Она была предоставлена автономной командой '/bin/tes
t'.
Результат был фактически таким:

 if /bin/test -d somedir

Хотя полный путь /bin/test никогда не использовался. Я показал это
здесь, чтобы подчеркнуть, что после инструкции "if" следует список ко
манд.
Обычно это было бы просто:

 если test -d somedir

Конечно, это нормально, и для лучшей переносимости этот стиль по-
прежнему является рекомендуемым способом использования команды test.
Но многие
люди считают, что она отличается от других языков программирования.
Чтобы создать оператор тестирования (обратите внимание, что я упомина
ю оператор тестирования, а не
язык оболочки, это локализованное изменение, не влияющее на
язык в целом) больше похож на другие языки программирования
программа 'test' была закодирована так, чтобы игнорировать последний
 аргумент, если это был ']'.
Затем копия тестовой программы может быть использована в качестве про
граммы '['.

 ...измените /bin/test, чтобы игнорировать ']' в качестве последнего
 аргумента...
 cp /bin/test /bin/[

Это позволяет:

 если [ -d somedir ]

Разве это не выглядит более нормально? Людям это понравилось, и это п
рижилось. Она
была настолько популярна, что и 'test', и '[' теперь являются встроен
ными в оболочку. Они
больше не запускают внешнюю программу '/ bin / test'. Но они * исполь
```

```
зовались *
для запуска внешних программ. Поэтому синтаксический анализ аргументо
в такой же
, как если бы они все еще запускали внешнюю программу. Это влияет
на синтаксический анализ аргументов.

 ит-тест -f * .txt
-тест: слишком много аргументов

Упс. У меня есть двадцать файлов .txt, и поэтому test получил один -
f, за которым следует
первый файл, за которым следуют остальные файлы. (например, test -f
 1.txt 2.txt
3.txt 4.txt )

 если test -d $file
test: ожидаемый аргумент

Упс. Я имел в виду установить file.

 file=/path/some/file
если тест -d $file

Если такие переменные не заданы, они будут расширены
оболочкой перед передачей их (возможно, внешней) команде и
полностью исчезнуть. Вот почему тестовые аргументы всегда следует зак
лючать в кавычки.

 если test -d "$file"
 если [ -d "$file" ]

На самом деле сегодня определено, что если задан только один аргумен
т, как
в данном случае "test FOO", то тогда test возвращает true, если аргум
ент
не равен нулю по длине текста. Поскольку "-d" имеет ненулевую длину,
 "test -d" является
истинным. Количество аргументов влияет на то, как тест анализирует ар
гументы. Это
позволяет избежать случая, когда в зависимости от данных может выгляд
еть как
оператор тестирования.

 ДАННЫЕ = "что-то"
 если тест "$DATA" # true, $DATA имеет ненулевую длину

 ДАННЫЕ =""
 если тест "$DATA" # false, $DATA имеет нулевую длину

Но проблема в том, как тест должен обрабатывать аргумент, который выг
лядит
как оператор? Раньше это приводило к возникновению ошибок, но теперь,
поскольку это
только один аргумент, он определяется как такой же, как test -n $DATA
.
```

```
 DATA="-d"
 если тест "$DATA" # true, $DATA имеет ненулевую длину
, если test -d # true, как и в предыдущем случае.

Поскольку test и [, возможно, являются внешними командами, все
их части выбраны так, чтобы избежать метасимволов оболочки. Имена опе
раторов Fortran
были хорошо известны в то время (например, .gt., .eq. и т. Д.) И были
введена в эксплуатацию и для оператора тестирования оболочки. Исходящ
ая из
Фортран, использующий -gt, -eq и т. Д. выглядело очень нормально.

Неправильное использование, приводящее к маловероятным результатам:

 если тест 5> 2 # true, "5" имеет ненулевую длину, создается файл с и
менем "2"

Предполагаемое использование:

 если тест 5 -gt 2 # true (и метасимволы оболочки не нуждаются в кавы
чках)

Затем, намного позже, где-то в середине 1980-х, Korn sh решили
улучшить эту ситуацию. Был введен новый оператор тестирования.
Эта команда всегда была встроенной в оболочку и, следовательно, могла
напрямую воздействовать на аргументы
оболочки. Это '[[', которое является ключевым словом оболочки.
(Ключевое слово, метасимволы, встроенные функции - все разные.) Поско
льку
оболочка обрабатывает [[ внутренне, все аргументы известны и их не ну
жно
заключать в кавычки.

 если [[ -d $file ]] # окей
, если [[ 5 > 2 ]] # окей

Я уверен, что я неправильно помню детали, но, надеюсь, это
полезно в качестве мягкого введения и в любом случае интересно.

Боб
```

Я надеюсь, что этот текст немного защитит вас от попадания из одной ловушки в другую.

Я нахожу ее очень интересной и информативной, поэтому я процитировал ее здесь. Большое спасибо, Боб, также за разрешение скопировать текст сюда!

Примеры кода

Фрагменты

Далее следуют некоторые фрагменты кода, используются различные способы реагирования оболочки.

- **проверьте, определена ли переменная / ненулевая**
  - `test "$MYVAR"`
  - `[ "$MYVAR" ]`
  - **Примечание:** Есть возможности изменить ситуацию, если переменная не *определена* или равна *нулю* - см. Расширение параметра - использование альтернативного значения
- **проверьте, существует ли каталог, если нет, создайте его**
  - `test ! -d /home/user/foo && mkdir /home/user/foo`
  - `[ ! -d /home/user/foo ] && mkdir /home/user/foo`
  - `if [ ! -d /home/user/foo ]; then mkdir /home/user/foo; fi`
- **проверьте, был ли задан минимум один параметр, и этот параметр - "Привет"**
  - `test $# -ge 1 -a "$1" = "Hello" || exit 1`
  - `[ $# -ge 1 ] && [ "$1" = "Hello" ] || exit 1` (см. Описание списков)

Список каталогов

Использование цикла for для перебора всех записей каталога, если запись является directory (`[ -d "$fn" ]`), выведите ее имя:

```
для ввода fn в *; выполните
[ -d "$fn" ] && echo "$fn"
Выполнено
```

Смотрите также

- Внутреннее: условное выражение (оно же "новая тестовая команда")
- Внутренняя: предложение if

[1)] Конечно, можно задаться вопросом, какая польза от включения круглых скобок в спецификацию без определения поведения с более чем 4 аргументами или насколько полезны примеры с 7 или 9 аргументами, прикрепленными к спецификации.</rant>

Обсуждение

ПК Пит, 2012/10/06 00:06 ()

Спасибо за интересное и полезное объяснение источников и требований операторов тестирования. Даже спустя 20 лет я все еще учусь!

Что я хотел бы знать, так это как избежать одной из наиболее распространенных ошибок тестов файлов и каталогов (в частности, -f и -d). Это странное поведение, когда вы тестируете скрытый файл или файл, начинающийся с "." (а не просто файл, который не читается с помощью разрешений, примененных к нему).

В этом случае, хотя файл может быть указан и передан в качестве аргумента для обоих типов тестов ("if [[-d" и "if test -d"), оба теста завершаются неудачей при передаче файла с точкой.

До сих пор все обходные пути, которые я видел, довольно громоздки и отвлекают от "хороших" сценариев оболочки. Можете ли вы помочь с примером или объяснить, почему эти тесты, по-видимому, "проваливаются" так, как они это делают, и что мы можем сделать, чтобы заставить их работать со всеми файлами?

Спасибо!

Ян Шампера, 2012/10/06 08:30 ()

Привет, Пит,

можете ли вы подробнее объяснить, что вы подразумеваете под ошибкой с точечными файлами? Точечный файл "скрыт" соглашением о том, чтобы не отображать его (ls, file explorers, ...), Технически не скрыт (т. Е. Нет флага "скрыто").

Что касается разрешений, это относительно легко объяснить. Просто приведите мне пример того, что неясно.

Пит, 2012/10/06 14:16 (), 2012/10/13 09:31 ()

Спасибо, Ян, я ценю помощь!

Хорошо, вот реальный пример проблемы, которую я вижу.

У меня есть каталог в моем домашнем каталоге. В этом подкаталоге есть только один элемент, каталог с именем ".git", который содержит несколько файлов и папок, в которых я хочу выполнить поиск (без использования find). Этот поиск является частью общего поиска, но по какой-то причине он, похоже, никогда не выполнял поиск в папке .git!

Я написал небольшой скрипт с рекурсивной функцией, который помещает компакт-диски в каждый каталог, затем для каждого файла в каталоге я использую следующий точный код (за вычетом некоторых функций печати, которые выполняются только вне теста папки):

```
#!/bin/bash

shopt -s dotglob

...

функция повторяется
{
 oldIFS=$IFS
 IFS=$ ' \ n'
для f в "$@"
 выполните
, если [[ -e "$PWD/$FilePattern" ]]; затем
 ## сделай что-нибудь, чтобы сообщить мне, что мы что-то нашли
... например, найденный
файл fi
echo "Тест: проверка, является ли $ f каталогом ..."
 если [[ -d "${f}" ]]; то
echo ищет в "$f"...
 cd "${f}"
 Повторяется их $(ls -A1 ".")
 cd ..
 фи
готово
 IFS=$oldIFS
}

...

Повторяется их $(ls -A1 "$StartPath")
```

Для начала я установил в $StartPath значение, скажем, '.' и запустил скрипт в моем домашнем каталоге.

Она работает, как и ожидалось, для всех папок, которые она находит, но когда она попадает в папку, содержащую папку .git, хотя первая команда echo повторяет папку dot (поэтому она не скрывается параметрами ls или чем-либо еще, поэтому я установил параметр оболочки dotglob), -общий тест всегда завершается неудачей для этой папки, а вторая команда echo никогда не выполняется, даже если это реальная папка, доступна для чтения и так далее.

Это происходит для любой ".folder" - за исключением того, что если я тестирую с помощью -d '.foldername' в командной строке, это работает!

Я уверен, что это что-то действительно глупое, что я неправильно понимаю, но будь я проклят, если смогу это понять. Есть идеи или предложения?

Я подумал, что это может быть использование "." в качестве параметра для ls в вызове функции ... но его удаление никак не повлияло на эту проблему, и я хочу иметь возможность расширить код и использовать его в качестве другого параметра позже. Это все, что я мог понять, может быть причиной проблемы.

Кстати, я также получаю очень странные ошибки в некоторых папках с этим скриптом, такие как "ls: опция 'A' недопустима", я не уверен, связаны ли они, но я не могу найти никакой информации ни в одном из документов оболочки об этих

сообщениях об ошибках или проблеме с папкой dot. Самое неприятное… Но по одному за раз!

Любая помощь очень ценится! Это сводит меня с ума, хорошо, что не так много файлов, которые я хочу найти, находятся внизу.папки!

Ян Шампера, 2012/10/13 09:41 ()

Для структуры каталогов, подобной

```
sub1/
sub1 / subsub1
sub1 / subsub2
sub2/
sub2 / subsub1
sub2 / subsub2
sub2 /.subsub3
sub2 /.subsub4
sub2 /.subsub4 / subsub1
sub2 / subsub5
sub2 / subsub6
```

она работает здесь:

```
bonsai@core:~/tests/firerecurs$ ./скрипт
Тест: проверка, является ли скрипт каталогом...
Тест: проверка, является ли sub1 каталогом...
Смотрю в sub1...
Тест: проверка, является ли subsub1 каталогом...
Смотрю в subsub1...
Тест: проверка, является ли subsub2 каталогом...
Смотрю в subsub2...
Тест: проверка, является ли sub2 каталогом...
Смотрю в sub2...
Тест: проверка, является ли subsub1 каталогом...
Смотрю в subsub1...
Тест: проверка, является ли subsub2 каталогом...
Смотрю в subsub2...
Тест: проверка, является ли .subsub3 каталогом...
Заглядываю в .subsub3...
Тест: проверка, является ли .subsub4 каталогом...
Заглядываю в .subsub4...
Тест: проверка, является ли subsubsub1 каталогом...
Ищем в subsubsub1...
Тест: проверка, является ли subsub5 каталогом...
Смотрю в subsub5...
Тест: проверка, является ли subsub6 каталогом...
Смотрю в subsub6...
```