Расширение параметров

Введение

Одной из основных функций Bash является управление **параметрами**. Параметр - это объект, который хранит значения и на который ссылается **имя**, **число** или **специальный символ**.

- параметры, на которые ссылается имя, называются **переменными** (это также относится к массивам).
- параметры, на которые ссылается число, называются **позиционными параметрами** и отражают аргументы, переданные оболочке
- параметры, на которые ссылается **специальный символ**, - это автоматически устанавливаемые параметры, которые имеют разные специальные значения и способы использования

Расширение параметров - это процедура получения значения из объекта, на который ссылается объект, например, расширение переменной для печати ее значения. Во время расширения вы можете делать очень неприятные вещи с параметром или его значением. Эти вещи описаны здесь .

Если вы где-то видели какой-то синтаксис расширения параметров и вам нужно проверить, что это может быть, попробуйте ознакомиться с обзорным разделом ниже!

Массивы могут быть частными случаями для расширения параметров, каждое применимое описание упоминает массивы ниже. Пожалуйста, также ознакомьтесь со статьей о массивах.

Для более технического представления о том, что такое параметр и какие типы существуют, см. словарную статью для "параметра".

Обзор

Ищете конкретный синтаксис, который вы видели, не зная названия?

- Простое использование
  - `$PARAMETER`
  - `${PARAMETER}`
- Косвенность
  - `${!PARAMETER}`
- Изменение регистра
  - `${PARAMETER^}`
  - `${PARAMETER^^}`
  - `${PARAMETER,}`

    - `${PARAMETER,,}`
    - `${PARAMETER~}`
    - `${PARAMETER~~}`
- Расширение имени переменной
    - `${!PREFIX*}`
    - `${!PREFIX@}`
- Удаление подстроки (также для **манипулирования именем** файла!)
    - `${PARAMETER#PATTERN}`
    - `${PARAMETER##PATTERN}`
    - `${PARAMETER%PATTERN}`
    - `${PARAMETER%%PATTERN}`
- Поиск и замена
    - `${PARAMETER/PATTERN/STRING}`
    - `${PARAMETER//PATTERN/STRING}`
    - `${PARAMETER/PATTERN}`
    - `${PARAMETER//PATTERN}`
- Длина строки
    - `${#PARAMETER}`
- Расширение подстроки
    - `${PARAMETER:OFFSET}`
    - `${PARAMETER:OFFSET:LENGTH}`
- Используйте значение по умолчанию
    - `${PARAMETER:-WORD}`
    - `${PARAMETER-WORD}`
- Присвоить значение по умолчанию
    - `${PARAMETER:=WORD}`
    - `${PARAMETER=WORD}`
- Используйте альтернативное значение
    - `${PARAMETER:+WORD}`
    - `${PARAMETER+WORD}`
- Ошибка отображения, если значение null или unset
    - `${PARAMETER:?WORD}`
    - `${PARAMETER?WORD}`

Простое использование

```
$PARAMETER

${PARAMETER}
```

Самая простая форма - просто использовать имя параметра в фигурных скобках. Это идентично использованию `$FOO` like, которое вы видите повсюду, но имеет то преимущество, что за ним могут сразу следовать символы, которые в противном случае интерпретировались бы как часть имени параметра. Сравните эти два выражения (`WORD="car"` например), где мы хотим напечатать слово с завершающим "s":

```
echo "Множественное число $ WORD, скорее всего, $WORDs"
echo "Множественное число $ WORD, скорее всего, ${WORD} s"
```

Почему первый сбой?Он ничего не печатает, потому что параметр (переменная) с именем " `WORDs` " не определен и, следовательно, печатается как "" (*ничего*). Без использования фигурных скобок для расширения параметров, Bash интерпретирует последовательность всех допустимых символов от вводного " `$` " до последнего допустимого символа как имя параметра. При использовании фигурных скобок вы просто заставляете Bash **интерпретировать только имя внутри ваших фигурных скобок**.

Кроме того, пожалуйста, помните, что **имена параметров** (как и почти все в UNIX®) чувствительны к **регистру!**

Вторая форма с фигурными скобками также необходима для доступа к позиционным параметрам (аргументам скрипта) за пределами `$9` :

```
echo "Аргумент 1 равен: $ 1"
echo "Аргумент 10 равен: ${10}"
```

Простое использование: массивы

См. Также статью об общем синтаксисе массива

Для массивов вам всегда нужны фигурные скобки. Массивы расширяются за счет отдельных индексов или массовых аргументов. Индивидуальный индекс ведет себя как обычный параметр, для массового расширения, пожалуйста, прочитайте статью о массивах, на которую дана ссылка выше.

- ${массив[5]}
- ${массив[*]}
- ${массив[@]}

Косвенность

`${!PARAMETER}`

В некоторых случаях, например, например

```
${ПАРАМЕТР}

${ПАРАМЕТР:0:3}
```

вместо этого вы можете использовать форму

```
${!ПАРАМЕТР}
```

чтобы ввести уровень косвенности. Параметр, на который ссылается, - это не `PARAMETER` он сам, а параметр, имя которого хранится как значение `PARAMETER` . Если параметр `PARAMETER` имеет значение " `TEMP` ", то `${!PARAMETER}` будет

расширяться до значения параметра с именем TEMP :

```
read -rep 'Какую переменную вы хотите проверить? ' look_var

printf 'Значение "%s" равно: "%s"\n' "$look_var" "${!look_var}"
```

Конечно, косвенное обращение также работает со специальными переменными:

```
# установите некоторые поддельные позиционные параметры
, установите один, два, три, четыре

# получить ПОСЛЕДНИЙ аргумент ("#" хранит количество аргументов, так
что "!#" будет ссылаться на ПОСЛЕДНИЙ аргумент)
echo ${!#}
```

Вы можете представить себе этот механизм как примерно эквивалентный принятию любого расширения параметра, которое начинается с имени параметра, и замене !PARAMETER части значением ПАРАМЕТРА.

```
echo "${!var^^}"
# ...
эквивалентно вычислению 'echo "${'"$var"'^^}"'
```

Это было неудачное дизайнерское решение использовать ! префикс для косвенного обращения, поскольку он вводит двусмысленность синтаксического анализа с другими расширениями параметров, которые начинаются с. ! Косвенное обращение невозможно в сочетании с любым расширением параметров, модификатор которого требует префикса к имени параметра. В частности, косвенное обращение невозможно в формах ${!var@} , ${!var*} , ${!var[@]} , ${!var[*]} , и ${#var} . Это означает ! , что префикс не может быть использован для получения индексов массива, длины строки или количества элементов в массиве косвенно (см. Косвенные обходные пути). Кроме того, ! расширение параметров с префиксом -конфликтует с ksh-подобными оболочками, которые имеют более мощную форму косвенной ссылки "имя-ссылка", где для расширения до имени переменной, на которую ссылаются, используется точно такой же синтаксис.

Косвенные ссылки на имена массивов также возможны, начиная с серии Bash 3 (точная версия неизвестна), но недокументированы. Подробности см. в разделе Косвенность.

Компания Chet добавила начальную реализацию nameref команды объявления ksh в ветку git devel. (declare -n , local -n , и т.д. будут поддерживаться). Это, наконец, решит многие проблемы, связанные с передачей и возвратом сложных типов данных в / из функций.

Изменение регистра

```
${PARAMETER^}

${PARAMETER^^}

${PARAMETER,}
```

```
${PARAMETER,,}

${PARAMETER~}

${PARAMETER~~}
```

Эти операторы расширения изменяют регистр букв в развернутом тексте.

`^` Оператор изменяет первый символ на верхний регистр, `,` оператор на нижний регистр. При использовании двойной формы (`^^` и `,,`) все символы преобразуются.

> Операторы (в **настоящее время недокументированные**) `~` и `~~` обратный регистр данного текста (in `PARAMETER`). `~` изменяет регистр первых букв слов в переменной, в то время `~~` как для всех регистров меняется на противоположные. Спасибо `Bushmills geirha` IRC ()-каналу Freenode и на нем за это открытие.

Пример: переименовать все `*.txt` имена файлов в нижний регистр

```
для файла в *.txt; сделайте
mv "$file" "${file,,}"
готово
```

Примечание: Изменение регистра - это удобная функция, которую вы можете применить к имени или названию. Или это? Модификация случая была важным аспектом выпуска Bash 4. В версии Bash 4, RC1 будет выполнять разделение слов, а затем изменение регистра, что приведет к регистру заголовка (где каждое слово пишется с большой буквы). Было решено применить изменение регистра к значениям, а не к словам, для выпуска Bash 4. Спасибо, Чет.

Изменение регистра: массивы

Изменение регистра может быть использовано для создания правильной заглавной буквы для имен или названий. Просто назначьте его массиву:

```
declare -a title=(my hello world john smith)
```

Для расширения массива изменение регистра применяется к **каждому расширенному элементу, независимо от того, расширяете ли вы отдельный индекс или массово расширяете** весь массив, используя `@` `*` индексы или. Некоторые примеры:

Предположим,: `array=(This is some Text)`

- `echo "${array[@],}"`
  - ⇒ `this is some text`
- `echo "${array[@],,}"`
  - ⇒ `this is some text`
- `echo "${array[@]^}"`
  - ⇒ `This Is Some Text`
- `echo "${array[@]^^}"`

    - ⇒ THIS IS SOME TEXT
- `echo "${array[2]^^}"`
    - ⇒ SOME

Расширение имени переменной

`${!PREFIX*}`

`${!PREFIX@}`

Это расширяется до списка всех заданных **имен переменных**, начинающихся со строки `PREFIX` . Элементы списка разделяются первым символом в `IFS` переменной - (<пробел> по умолчанию).

Это покажет все определенные имена переменных (не значения!) начиная с "BASH":

```
$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_LINENO BASH_SOURCE BASH_SU
BSHELL BASH_VERSINFO BASH_VERSION
```

Этот список также будет включать имена массивов.

Удаление подстроки

`${PARAMETER#PATTERN}`

`${PARAMETER##PATTERN}`

`${PARAMETER%PATTERN}`

`${PARAMETER%%PATTERN}`

Это может **расширить только часть** значения параметра, **учитывая шаблон для описания того, что нужно удалить** из строки. Шаблон интерпретируется точно так же, как шаблон для описания соответствующего имени файла (глобализация). Подробнее см. в разделе Сопоставление шаблонов.

Пример строки (*просто цитата большого человека*):

```
MYSTRING="Будьте либеральны в том, что вы принимаете, и консервативны
в том, что вы отправляете"
```

С самого начала

`${PARAMETER#PATTERN}` и `${PARAMETER##PATTERN}`

Эта форма предназначена для удаления описанного шаблона, пытающегося **соответствовать ему с начала строки**. Оператор " `#` " попытается удалить самый короткий текст, соответствующий шаблону, в то время как " `##` " пытается сделать это с

самым длинным текстом, соответствующим шаблону. Посмотрите на следующие примеры, чтобы понять идею (соответствующий текст ~~помечен зачеркнутым~~, помните, что он будет удален!):

Синтаксис	Результат
`${MYSTRING#*in}`	~~Будьте либеральны в~~ том, что вы принимаете, и консервативны в том, что вы отправляете
`${MYSTRING##*in}`	~~Будьте либеральны в том, что вы принимаете, и консервативны в~~ том, что вы отправляете

С конца

`${PARAMETER%PATTERN}` и `${PARAMETER%%PATTERN}`

Во второй форме все будет то же самое, за исключением того, что Bash теперь пытается сопоставить шаблон с конца строки:

Синтаксис	Результат
`${MYSTRING%in*}`	Будьте либеральны в том, что вы принимаете, и консервативны ~~в том, что вы отправляете~~
`${MYSTRING%%in*}`	Будьте либеральны ~~в том, что вы принимаете, и консервативны в том, что вы отправляете~~

Вторая форма обнуляет переменные, которые начинаются с `in` , работая с конца.

Общее использование

Как, черт возьми, это помогает облегчить мою жизнь?

Ну, возможно, наиболее распространенным его использованием является **извлечение частей имени** файла. Просто посмотрите на следующий список с примерами:

- **Получить имя без расширения**
  - `${FILENAME%.*}`
  - ⇒ `bash_hackers`~~`.txt`~~
- **Получить расширение**
  - `${FILENAME##*.}`
  - ⇒ ~~`bash_hackers.`~~`txt`
- **Получить имя каталога**
  - `${PATHNAME%/*}`
  - ⇒ `/home/bash`~~`/bash_hackers.txt`~~
- **Получить имя файла**
  - `${PATHNAME##*/}`
  - ⇒ ~~`/home/bash/`~~`bash_hackers.txt`

Это синтаксис для имен файлов с одним расширением. В зависимости от ваших потребностей вам может потребоваться настроить кратчайшее / самое длинное совпадение.

Удаление подстроки: массивы

Как и для большинства функций расширения параметров, работа с массивами **будет обрабатывать каждый расширенный элемент**, как для индивидуального расширения, так и для массового расширения.

Простой пример, удаление завершающего `is` элемента из всех элементов массива (при расширении):

Предположим,: `array=(This is a text)`

- `echo "${array[@]%is}"`
  - ⇒ `Th a text`
  - (это было: `Th~~is~~ ~~is~~ a text`)

Все остальные варианты этого расширения ведут себя одинаково.

Поиск и замена

```
${PARAMETER/PATTERN/STRING}

${PARAMETER//PATTERN/STRING}

${PARAMETER/PATTERN}

${PARAMETER//PATTERN}
```

Этот может заменить (*заменить*) подстроку, соответствующую шаблону, во время расширения. Соответствующая подстрока будет полностью удалена, и будет вставлена заданная строка. Опять какая-нибудь примерная строка для тестов:

```
MYSTRING="Будьте либеральны в том, что вы принимаете, и консервативны в том, что вы отправляете"
```

Две основные формы отличаются только **количеством косых** черт после имени параметра: `${PARAMETER/PATTERN/STRING}` и `${PARAMETER//PATTERN/STRING}`

Первая (*одна косая* черта) предназначена для замены только **первого вхождения** данного шаблона, вторая (*две косые* черты) предназначена для замены **всех вхождений** шаблона.

Во-первых, давайте попробуем сказать "счастливый" вместо "консервативный" в нашей строке примера:

```
${MYSTRING//консервативный/счастливый}
```

⇒ `Be liberal in what you accept, and ~~conservative~~happy in what you send`

Поскольку в этом примере есть только один "консервативный", на самом деле не имеет значения, какую из двух форм мы используем.

Давайте поиграем со словом "in", я не знаю, имеет ли это какой-либо смысл, но давайте заменим его на "by".

Первая форма: заменить первое вхождение

```
${MYSTRING/in/by}
```

⇒ `Be liberal ~~in~~by what you accept, and conservative in what you send`

Вторая форма: заменить все вхождения

```
${MYSTRING//in/by}
```

⇒ `Be liberal ~~in~~by what you accept, and conservative ~~in~~by what you send`

Привязка Дополнительно вы можете "привязать" выражение: A `#` (хэш-метка) будет указывать, что ваше выражение сопоставляется с начальной частью строки, а `%` (знак процента) сделает это для конечной части.

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING/#x/y} # РЕЗУЛЬТАТ: yxxxxxxxxx
echo ${MYSTRING/%x/y} # РЕЗУЛЬТАТ: xxxxxxxxxy
```

Если заменяющая часть полностью опущена, совпадения заменяются nullstring , т. е. Они удаляются. Это эквивалентно указанию пустой замены:

```
echo ${MYSTRING//консервативный/}
# эквивалентно
echo ${MYSTRING//conservative}
```

Поиск и замена: массивы

Этот тип расширения параметров, применяемый к массивам **, применяется ко всем расширенным элементам**, независимо от того, расширяется ли отдельный элемент или все элементы, использующие синтаксис массового расширения.

Простой пример, изменение (строчной) буквы `t` на `d` :

Предположим,: `array=(This is a text)`

- `echo "${array[@]/t/d}"`
  - ⇒ `This is a dext`
- `echo "${array[@]//t/d}"`
  - ⇒ `This is a dexd`

Длина строки

`${#PARAMETER}`

При использовании этой формы длина значения параметра увеличивается. Опять же, цитата большого человека, чтобы иметь тестовый текст:

```
MYSTRING="Будьте либеральны в том, что вы принимаете, и консервативны в том, что вы отправляете"
```

Использование echo `${#MYSTRING}` ...

```
⇒ 64
```

Длина указывается в символах, а не в байтах. В зависимости от вашей среды это может быть не всегда одинаково (многобайтовые символы, например, в кодировке UTF8).

Об этом особо нечего сказать, mh?

(Строка) длина: массивы

Для массивов этот тип расширения имеет два значения:

- Для **отдельных** элементов он сообщает длину строки элемента (как и для каждого "обычного" параметра)
- Для **массовых индексов** `@` и `*` сообщает о количестве заданных элементов в массиве

Пример:

Предположим,: `array=(This is a text)`

- `echo ${#array[1]}`
  - ⇒ 2 (слово "is" имеет длину 2)
- `echo ${#array[@]}`
  - ⇒ 4 (массив содержит 4 элемента)

Внимание: количество используемых элементов не обязательно должно соответствовать наивысшему индексу. В Bash возможны разреженные массивы, это означает, что у вас может быть 4 элемента, но с индексами 1, 7, 20, 31. **Вы не можете перебирать такой массив с помощью цикла счетчика, основанного на количестве элементов!**

Расширение подстроки

`${PARAMETER:OFFSET}`

`${PARAMETER:OFFSET:LENGTH}`

Это может расширить только **часть** значения параметра, учитывая **начальную позицию** и, возможно, **длину**. Если `LENGTH` параметр опущен, параметр будет расширен до конца строки. Если `LENGTH` значение отрицательное, оно принимается как второе смещение в строке, считая от конца строки.

`OFFSET` и `LENGTH` может быть **любым** арифметическим выражением. **Будьте осторожны:** `OFFSET` начинается с 0, а не с 1!

Пример строки (цитата большого человека): `MYSTRING="Be liberal in what you accept, and conservative in what you send"`

Использование только смещения

В первой форме расширение используется без значения длины, обратите внимание, что смещение 0 является первым символом:

```
echo ${MYSTRING:35}
```

⇒ ~~Be liberal in what you accept, and~~ conservative in what you send

Использование смещения и длины

Во второй форме мы также задаем значение длины:

```
echo ${MYSTRING:35:12}
```

⇒ ~~Be liberal in what you accept, and~~ conservative~~ in what you send~~

Отрицательное значение смещения

Если заданное смещение отрицательное, оно отсчитывается от конца строки, т.е. смещение, равное -1, является последним символом. В этом случае, конечно, длина по-прежнему считается вперед. При использовании отрицательного смещения необходимо сделать одну особую вещь: вам нужно отделить (отрицательное) число от двоеточия:

```
${MYSTRING: -10:5}
${MYSTRING:(-10):5}
```

Почему? Потому что это интерпретируется как синтаксис расширения параметров для **использования значения по умолчанию**.

Отрицательное значение длины

Если `LENGTH` значение отрицательное, оно используется как смещение от конца строки. Расширение происходит от первого до второго смещения, затем:

```
echo "${MYSTRING:11:-17}"
```

⇒ ~~Be liberal~~ in what you accept, and conservative~~ in what you send~~

Это работает с версии Bash 4.2-alpha, см. Также Изменения в Bash.

Расширение подстроки / элемента: массивы

Для массивов этот тип расширения снова имеет 2 значения:

- Для **отдельных** элементов он расширяется до указанной подстроки (как и для каждого “обычного” параметра)
- Для **массовых индексов** `@` и `*` массового расширения отдельных элементов массива, обозначаемых 2 заданными числами (*начальный элемент*, *количество элементов*)

Пример:

Предположим,: `array=(This is a text)`

- `echo ${array[0]:2:2}`
  - ⇒ `is` ("есть" в "Этом", элемент массива 0)
- `echo ${array[@]:1:2}`
  - ⇒ `is a` (начиная с элемента 1 включительно, расширяются 2 элемента, то есть элементы 1 и 2)

Используйте значение по умолчанию

`${PARAMETER:-WORD}`

`${PARAMETER-WORD}`

Если параметр `PARAMETER` не задан (никогда не был определен) или равен нулю (пустой), этот параметр расширяется до `WORD` , в противном случае он расширяется до значения `PARAMETER` , как если бы он только что был `${PARAMETER}` . Если вы опустите `:` (двоеточие), как показано во второй форме, значение по умолчанию используется только тогда, когда параметр не был **установлен**, а не когда он был пустым.

```
echo "Ваш домашний каталог: ${HOME:-/home/$USER}".
echo "${HOME:-/home/$USER} будет использоваться для хранения ваших пе
рсональных данных".
```

Если `HOME` значение не задано или пусто, каждый раз, когда вы хотите напечатать что-то полезное, вам нужно ввести синтаксис этого параметра.

```
#!/bin/bash

read -p "Введите свой пол (просто нажмите ENTER, чтобы не сообщать на
м): "GENDER
echo "Ваш пол - $ {GENDER:-секрет}".
```

Он напечатает "Ваш пол является секретом". когда вы не вводите пол. Обратите внимание, что значение по умолчанию **используется для времени расширения**, оно **не присваивается параметру**.

Используйте значение по умолчанию: массивы

Для массивов поведение очень похоже. Опять же, вы должны сделать разницу между расширением отдельного элемента по заданному индексу и массовым расширением массива с использованием индексов `@` и `*`.

- Для отдельных элементов это то же самое: если расширенный элемент `NULL` установлен или не установлен (смотрите `:-` `-` варианты и), текст по умолчанию расширяется
- Для синтаксиса массового расширения текст по умолчанию расширяется, если массив
  - не содержит элемента или не задан (`:-` `-` варианты и здесь означают то **же** самое)
  - содержит только элементы, которые являются nullstring (`:-` вариант)

Другими словами: основное значение этого типа расширения применяется как можно более последовательно к массивам.

Пример кода (пожалуйста, попробуйте примеры сами):

```
####
# Примеры для неустановленных / пустых массивов и элементов nullstrin
g
####


### СЛУЧАЙ 1: сбросить массив (нет массива)

# убедитесь, что у нас вообще нет массива
unset array

echo ${array[@]: -этот массив имеет значение NULL или unset}
echo ${array[@]-этот массив имеет значение NULL или unset}

### СЛУЧАЙ 2: набор, но пустой массив (без элементов)

# объявить пустой массив
array=()

echo ${array[@]: -этот массив имеет значение NULL или unset}
echo ${array[@]-этот массив имеет значение NULL или unset}


### СЛУЧАЙ 3: массив, содержащий только один элемент,
массив nullstring=("")

echo ${array[@]: -этот массив имеет значение NULL или unset}
echo ${array[@]-этот массив имеет значение NULL или unset}


### СЛУЧАЙ 4: массив, содержащий только два элемента, нулевую строку
и обычное слово
array=("" word)

echo ${array[@]: -этот массив имеет значение NULL или unset}
echo ${array[@]-этот массив имеет значение NULL или unset}
```

Присвоить значение по умолчанию

```
${PARAMETER:=WORD}
```

```
${PARAMETER=WORD}
```

Это работает так же, как **использование значений по умолчанию**, но текст по умолчанию, который вы предоставляете, не только расширяется, но и **присваивается** параметру, если он не установлен или равен нулю. Эквивалентно использованию значения по умолчанию, когда вы опускаете `:` (двоеточие), как показано во второй форме, значение по умолчанию будет присвоено только тогда, когда параметр не был **установлен**.

```
echo "Ваш домашний каталог: ${HOME:=/home/$USER}".
echo "$HOME будет использоваться для хранения ваших личных данных".
```

После первого расширения здесь (`${HOME:=/home/$USER}`) `HOME` устанавливается и может использоваться.

Давайте изменим наш пример кода из приведенного выше:

```
#!/bin/bash

read -p "Введите свой пол (просто нажмите ENTER, чтобы не сообщать на
м): "GENDER
echo "Ваш пол - $ {GENDER:= секрет}".
эхо "Ах, если вы забыли, ваш пол на самом деле: $GENDER"
```

Присвоите значение по умолчанию: массивы

Для массивов этот тип расширения ограничен. Для отдельного индекса он ведет себя так же, как для "обычного" параметра, этому одному элементу присваивается значение по умолчанию. Индексы массового расширения `@` и `*` **не могут быть использованы здесь**, потому что им невозможно присвоить!

Используйте альтернативное значение

```
${PARAMETER:+WORD}
```

```
${PARAMETER+WORD}
```

Эта форма не расширяется до нуля, если параметр не задан или пуст. Если оно установлено, оно расширяется не до значения параметра, **а до некоторого текста, который вы можете указать**:

```
echo "Приложение Java установлено и может быть запущено.${JAVAPATH:+
ПРИМЕЧАНИЕ: JAVAPATH, похоже, установлен}"
```

Приведенный выше код просто добавит предупреждение, если `JAVAPATH` оно установлено (потому что это может повлиять на поведение при запуске этого

воображаемого приложения).

Еще один нереалистичный пример… Запросите некоторые флаги (по любой причине), а затем, если они были установлены, выведите предупреждение, а также выведите флаги:

```
#!/bin/bash

read -p "Если вы хотите использовать специальные флаги, введите их се
йчас: " SPECIAL_FLAGS
echo "Установка приложения завершена ${SPECIAL_FLAGS:+ (ПРИМЕЧАНИЕ: у
становлены специальные флаги: $SPECIAL_FLAGS)}".
```

Если вы опустите двоеточие, как показано во второй форме (`${PARAMETER+WORD}`), будет использоваться альтернативное значение, если параметр установлен (и он может быть пустым)! Вы можете использовать его, например, для подачи жалобы, если нужные вам переменные (и они могут быть пустыми) не определены:

```
# протестируйте это с помощью трех этапов:

# сбросить foo
# foo=""
# foo="что-то"

если [[ ${foo+isset} = isset ]]; тогда
echo "foo установлен ..."
иначе
эхо-сигнал "foo не установлен ..."
fi
```

Используйте альтернативное значение: массивы

Аналогично случаям расширения массивов до значения по умолчанию, это расширение ведет себя как для "обычного" параметра при использовании отдельных элементов массива по индексу, но реагирует по-разному при использовании индексов массового расширения `@` и `*` :

- Для отдельных элементов это то же самое: если расширенный элемент **не** равен нулю или не установлен (смотрите варианты:+ и +), альтернативный текст расширяется
- Для синтаксиса массового расширения альтернативный текст расширяется, если массив
  - содержит элементы, где min. один элемент **не** является nullstring (варианты :+ и + означают здесь то же самое)
  - содержит **только** элементы, которые **не** являются nullstring (вариант :+)

Для некоторых случаев, с которыми можно поиграть, пожалуйста, ознакомьтесь с примерами кода в описании для использования значения по умолчанию.

Ошибка отображения, если значение null или unset

`${PARAMETER:?WORD}`

`${PARAMETER?WORD}`

Если параметр `PARAMETER` установлен / не равен нулю, эта форма просто расширит его. В противном случае расширение `WORD` будет использоваться в качестве приложения для сообщения об ошибке:

```
$ echo "Параметр unset равен: ${p_unset?не задано}"
bash: p_unset: не установлен
```

После печати этого сообщения,

- интерактивная оболочка имеет `$?` ненулевое значение
- неинтерактивная оболочка завершает работу с ненулевым кодом выхода

Значение двоеточия (`:`) такое же, как и для других синтаксисов расширения параметров: оно указывает, если

- только отменить или
- сбросить и очистить параметры

принимаются во внимание.

Примеры кода

Удаление подстроки

Удаление первых 6 символов из текстовой строки:

```
СТРОКА="Привет, мир"

# выводить только 'Hello'
echo "${STRING%??????}"

# выводить только 'world'
echo "${STRING#??????}"

# сохранить его в той же переменной
STRING=${СТРОКА#??????}
```

Проблемы с ошибками и переносимостью

- **Исправлено в 4.2.36** (патч (ftp://ftp.cwru.edu/pub/bash/bash-4.2-patches/bash42-036)). Bash не следует ни POSIX, ни своей собственной документации при расширении либо цитируемого `"$@"` , либо `"${arr[@]}"` смежного расширения. `"$@$x"` расширяется так же, как `"$*$x"` - т. е. Все параметры

плюс смежное расширение объединяются в один аргумент. В качестве обходного пути каждое расширение должно быть указано отдельно. К сожалению, на обнаружение этой ошибки ушло очень много времени.

```
~ $ set -- a b c; x=foo; printf '<%s> ' "$@$x" "$*""$x" "$@""$x"
<a b cfoo> <a b cfoo> <a> <b> <cfoo>
```

- Почти все оболочки расходятся во мнениях относительно обработки не заключенного в кавычки `$@` , `${arr[@]}` , `$*` , и `${arr[*]}` когда IFS (http://mywiki.wooledge.org/IFS) имеет значение null . POSIX неясен в отношении ожидаемого поведения. Нулевое значение IFS приводит к случайному разбиению слов и расширению пути. Поскольку существует несколько веских причин оставлять `IFS` значение set равным null более чем на время выполнения одной или двух команд, и еще меньше - для расширения `$@` и `$*` без кавычек, это должно быть редкой проблемой. **Всегда цитируйте их**!

```
коснитесь x 'y z'
для sh в bb {{d,b}a,{m,}k, z} sh;
повторите "$sh"
 "$sh" -s a 'b c' d \*

${ZSH_VERSION+:} false && эмулировать sh
IFS=
printf '<%s> ' $*
echo
printf "<%s> " $@
echo
EOF
```

```
bb
<ab cd*>
<ab cd*>
dash
<ab cd*>
<ab cd*>
bash
<a> <b c> <d> <x> <y z>
<a> <b c> <d> <x> <y z>
mksh
<a b c d *>
<a b c d *>
ksh
<a> <b c> <d> <x> <y z>
<a> <b c> <d> <x> <y z>
zsh
<a> <b c> <d> <x> <y z>
<a> <b c> <d> <x> <y z>
```

Когда `IFS` установлено значение, отличное от null , или не установлено, все оболочки ведут себя одинаково - сначала расширяются на отдельные аргументы, затем применяют расширение имени пути и разделение слов к результатам, за исключением zsh , который не выполняет расширение имени пути в режиме по умолчанию.

- Кроме того, оболочки расходятся во мнениях относительно различных способов разделения слов, поведения вставки символов-разделителей из IFS `$*` и способа объединения смежных аргументов, когда IFS изменяется в середине расширения из-за побочных эффектов.

```
для sh в bb {{d,b}a,po,{m,}k,z}sh; выполните
printf '%-4s: ' "$sh"
 "$sh"

${ZSH_VERSION+:} false && эмулировать sh
set -f -- a b c
unset -v
ЕСЛИ printf '<%s> ' ${*}${IFS=}${*}${IFS:=-}"${*}"
echo
EOF
```

```
bb: <a b cabc> <a-b-c>
dash: <a b cabc> <a-b-c>
bash: <a> <b> <ca> <b> <c-a b c>
posh: <a> <b> <ca b c> <a-b-c>
mksh: <a> <b> <ca b c> <a-b-c>
ksh: <a> <b> <ca> <b> <c> <a b c>
zsh: <a> <b> <ca> <b> <c> <a-b-c>
```

  ksh93 и mksh могут дополнительно достичь этого побочного эффекта (и других) с помощью `${ cmds;}` расширения. Я еще не проверил все возможные побочные эффекты, которые могут повлиять на расширение на полпути расширения таким образом.

- Как упоминалось ранее, косвенная форма Bash путем добавления префикса к расширению параметра с помощью a `!` конфликтует с тем же синтаксисом, который используется mksh, zsh и ksh93 для разных целей. Bash "слегка" изменит это расширение в следующей версии, добавив ссылки на имена.
- Bash (и большинство других оболочек) не допускают .'s в идентификаторах. В ksh93 точки в именах переменных используются для ссылки на методы (например, "Функции дисциплины"), атрибуты, специальные переменные оболочки и для определения "реального значения" экземпляра класса.
- В ksh93 `_` параметр имеет еще больше применений. Он используется так же, как `self` и в некоторых объектно-ориентированных языках; как заполнитель для некоторых данных, локальных для класса; а также как механизм наследования классов. В большинстве других контекстов `_` совместим с Bash.
- Bash оценивает индексы расширения среза (`${x:y:z}`) только в том случае, если параметр установлен (как для вложенных расширений, так и для арифметики). Для диапазонов Bash оценивает как можно меньше, т. Е., Если первая часть находится вне диапазона, вторая не будет оцениваться. ksh93 и mksh всегда оценивают части индекса, даже если параметр не задан.

```
 $ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=(); echo "${x[@]:n,6:m}"' # Нет вывода
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=([5]=hi); echo "${x[@]:n,6:m}"'
yo
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=([6]=привет); echo "${x[@]:n,6:m}"'
yojo
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x= 12345; echo "${x:n,5:m}"'
yojo
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x= 12345; echo "${x: n,6:m}"'
yo
```

Вложенность цитат

- В большинстве оболочек при работе с "альтернативным" расширением параметров, которое расширяется до нескольких слов, и при вложении таких расширений возможны не все комбинации вложенных кавычек.

```
# Bash
 $ typeset -a a=(meh bleh blerg) b
 $ IFS=e
 $ printf "<% s> " "${b[@]-"${a[@]}""${a[@]}"}"; echo # Весь PE заклю
чен в кавычки, поэтому Bash считает внутренние кавычки избыточными.
<meh> <bleh> <blerg meh> <bleh> <blerg>
 $ printf "<%s> " "${b[@]-${a[@]} ${a[@]}}"; echo # Внешние кавычки п
риводят к тому, что внутренние расширения считаются заключенными в ка
вычки.
<meh> <bleh> <blerg meh> <bleh> <blerg>
 $ b=(звуковой сигнал)
 $ printf "<%s> " "${b[@]-"${a[@]}" "${a[@]}"}" "${ b[@]-${a[@]} ${a
[@]}}"; echo # Снова никаких сюрпризов. Внешние кавычки цитируют все
рекурсивно.
<мип> <бип> <мип> <бип>
```

Теперь давайте посмотрим, что может произойти, если мы оставим внешнее без кавычек.

```
# Bash
 $ typeset -a a=(meh bleh blerg) b
 $ IFS=e
 $ printf "<%s> " ${b[@]-"${a[@]}" "${a[@]}"}; echo # Внутренние кавы
чки заключают внутренние расширения в кавычки.
<meh> <bleh> <blerg meh> <bleh> <blerg>
 $ printf "<% s> " $ {b[@]- $ {a[@]} ${a[@]}}; echo' # Никаких кавыче
к во всех словосочетаниях / глобусах, как и следовало ожидать.
<m> <h> <bl> <h> <bl> <rg m> <h> <bl> <h> <bl> <rg>
```

Все это может быть интуитивно понятным и является наиболее распространенной реализацией, но этот дизайн отстой по ряду причин. Во-первых, это означает, что Bash делает абсолютно невозможным расширение какой-либо части внутренней области без *кавычек*, оставляя внешнюю область в кавычках. Цитирование внешних сил цитирование внутренних областей рекурсивно (за исключением вложенных командных замен, конечно). Разделение слов необходимо для разделения слов внутренней области, что невозможно сделать вместе с внешним цитированием. Рассмотрим следующий (лишь слегка притянутый за уши) код:

```
# Bash (нерабочий пример)

unset -v IFS # убедитесь, что у нас есть IFS по умолчанию

если какая-то хрень; тогда
typeset -a someCmd=(myCmd arg1 'arg2 ура!' 'третий * аргумент *' 4)
фи

someOtherCmd=
набор типов моей команды -a otherArgs=(arg3 arg4)

# Как вы думаете, чего ожидал здесь программист?
# Как вы думаете, что произойдет на самом деле...

"${someCmd[@] -"$someOtherCmd" arg2 "${otherArgs[@]}"}" arg5
```

Эта последняя строка, возможно, не самая очевидная, но я сталкивался со случаями, когда такой тип логики может быть желательным и реалистичным. Мы можем сделать вывод, что было задумано:

- Если `someCmd` установлено, то результирующее расширение должно выполнить команду: `"myCmd" "arg1" "arg2 yay!" "third*arg*" "4" "arg5"`
- В противном случае, если `someCmd` не задано, разверните `$someOtherCmd` и внутренние аргументы, чтобы выполнить другую команду : `"mycommand" "arg2" "arg3" "arg4" "arg5"` .

К сожалению, невозможно получить желаемый результат в Bash (и большинстве других оболочек), не применяя существенно иной подход. Единственный способ разделить литеральные внутренние части - это разделение слов, для чего PE должен быть без кавычек. Но единственный способ правильно развернуть внешнее расширение без разделения слов или глобулирования - это процитировать его. Bash фактически расширит команду как одну из следующих:

```
# Приведенный PE выдает здесь правильный результат...
 $ bash -c 'typeset -a someCmd=(myCmd arg1 "arg2 ура!" "третий * аргу
мент *" 4); printf "<%s> " "${someCmd[@]-"$someOtherCmd" arg2 "${othe
rArgs[@]}"}" arg5;эхо '
<myCmd> <arg1> <arg2 ура!> <третий * аргумент *> <4> <arg5>

# ... но в обратном случае первые 3 аргумента склеиваются вместе. Обх
одных путей нет.
 $ bash -c 'typeset -a otherArgs=(arg3 arg4); someOtherCmd=mycommand;
printf "<%s> " "${someCmd[@]-"$someOtherCmd" arg2 "${otherArgs[@]}"}"
arg5; echo'
<моя команда arg2 arg3> <arg4> <arg5>

# ЕСЛИ ТОЛЬКО! мы снимаем кавычки с внешнего расширения, позволяя вну
тренним кавычкам
# влияет на необходимые части, позволяя разделение слов для разделени
я литералов:
 $ bash -c 'typeset -a otherArgs=(arg3 arg4); someOtherCmd=mycommand;
printf "<%s> " ${someCmd[@]- "$someOtherCmd" arg2 "${otherArgs[@]}"}
arg5; echo'
<mycommand> <arg2>> <arg4> <arg5>

# Успех!!!
 $ bash -c 'typeset -a someCmd=(myCmd arg1 "arg2 ура!" "третий * аргу
мент *" 4); printf "<%s> " ${someCmd[@]- "$someOtherCmd" arg2 "${othe
rArgs[@]}"} arg5; echo'
<myCmd> <arg1> <arg2> <ура!> <третий * аргумент *> <4> <arg5>

# ...Ах, черт возьми. (опять же, обходной путь невозможен.)
```

Исключение ksh93

Насколько мне известно, ksh93 - единственная оболочка, которая действует по-другому. Вместо принудительного ввода вложенных расширений в кавычки, кавычки в начале и конце вложенной области приведут к изменению состояния кавычек внутри вложенной части. Я понятия не имею, является ли это преднамеренным или документированным эффектом, но это решает проблему и, следовательно, добавляет много потенциальной мощности этим расширениям.

Все, что нам нужно сделать, это добавить две дополнительные двойные кавычки:

```
# ksh93 прохождение двух неудачных тестов сверху:

 $ ksh -c 'otherArgs=(arg3 arg4); someOtherCmd="mycommand"; printf "
<%s> " "${someCmd[@]-""$someOtherCmd" arg2 "${otherArgs[@]}""}" arg5;
echo'
<mycommand>> <arg3> <arg4> <arg5>

 $ ksh -c 'typeset -a someCmd=(myCmd arg1 "arg2 ура!" "третий * аргум
ент *" 4); printf "<%s> " "${someCmd[@]-""$someOtherCmd" arg2 "${othe
rArgs[@]}""}"arg5; echo'
<myCmd> <arg1> <arg2 ура!> <третий * аргумент *> <4> <arg5>
```

Это может использоваться для управления состоянием кавычек любой части любого расширения на произвольную глубину. К сожалению, это единственная оболочка, которая делает это, и разница может привести к возможной проблеме совместимости.

Смотрите также

- Внутренний: введение в расширение и замену
- Внутренние: массивы
- Словарь, внутренний: Параметр

Обсуждение

Стэн Р., 2010/06/19 23:48 ()

Обнаружена ошибка в подзаголовке "Поиск и замена", рядом с началом раздела "Привязка", где есть пример фрагмента:

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING//#x/y} # РЕЗУЛЬТАТ: yxxxxxxxxx
echo ${MYSTRING//%x/y} # РЕЗУЛЬТАТ: xxxxxxxxxy
```

Это должно быть:

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING/#x/y} # РЕЗУЛЬТАТ: yxxxxxxxxx
echo ${MYSTRING/%x/y} # РЕЗУЛЬТАТ: xxxxxxxxxy
```

Разница в том, что MYSTRING/ вместо MYSTRING// , поскольку у вас не может быть одновременно привязки и .

Ян Шампера, 2010/06/20 12:56 ()

Спасибо за эту находку. Может быть, поздняя ночная опечатка :)

Стэн Р., 2010/06/21 01:11 ()

Рад, что смог помочь.

Шмерл, 2010/10/20 16:37 ()

Я обнаружил ошибку. Вы пишете:

${ПАРАМЕТР:-WORD} ${ПАРАМЕТР-WORD} Если параметр параметра не задан (никогда не был определен) или равен нулю (пустой), он расширяется до WORD , в противном случае он расширяется до значения ПАРАМЕТРА, как если бы он был просто ${PARAMETER} . ***Если вы опустите : (двоеточие), как показано во второй форме, значение по умолчанию используется только тогда, когда параметр не был установлен, а не когда он был пустым.***

Запуск простого теста

```
str=""; echo ${str-test_result}
```

Производит:

тест_результат

Теперь выполняется:

```
unset str; echo ${str-test_result}
```

Ничего не дает. Итак, вышесказанное прямо противоположно. Это должно быть: если вы опускаете : (двоеточие), как показано во второй форме, значение по умолчанию используется только тогда, когда ***параметр был пустым, а не когда он был usnet****.*

Ян Шампера, 2010/10/20 21:13 ()

Это работает (и всегда работало) для меня:

```
bonsai@core:~$ str=""; echo ${str-test_result}

bonsai@core:~$ unset str; echo ${str-test_result}
тест_результат
bonsai@core:~$
```

Я не уверен, что может быть причиной того, что ваши тесты делают прямо противоположное. Это действительно Bash? Хотя это поведение также задается POSIX ***(R): "В расширениях параметров [показано ранее] использование <двоеточия> в формате должно приводить к проверке параметра, который не задан или равен нулю; пропуск <двоеточия> должен приводить к проверкепараметр, который только не установлен ".***

Есть ли у вас какой-либо способ узнать, в чем заключается ваша (или моя) проблема?

Шмерл, 2010/10/21 15:59 ()

Извините, я действительно имел в виду случай с plus .

Запустите этот тест:

str=""; echo ${str+'test_result'}

выдает: ***test_result***

сбросить значение str; echo ${str+'test_result'}

выдает: (ничего)

Я ошибался в других случаях, это различие относится только к регистру + (альтернативное значение).

Шмерл, 2010/10/21 16:03 ()

Убедитесь, что там есть плюс - предварительный просмотр съел его.

Ян Шампера, 2010/10/23 09:36 (), 2010/10/23 09:37 ()

Он делает именно то, что должен делать. + выдает альтернативное значение, означает, что оно расширяется до чего-то ***вместо*** *реального значения параметра, если параметр "установлен" (+) или "установлен или нулевой" (:+).*

Шмерл, 2010/10/24 04:20 ()

Да, но ваш текст в вики выше вводит в заблуждение. Он пишет:

Если вы опустите двоеточие, как показано во второй форме (${PARAMETER+WORD}), альтернативное значение не будет использоваться, когда параметр пуст, только если он не ***установлен!***

В то время как он должен сказать:

Если вы опустите двоеточие, как показано во второй форме (${PARAMETER+WORD}), альтернативное значение не будет использоваться, когда параметр не ***задан****, только если он* ***пустой****!*

Ян Шампера, 2010/10/24 06:09 ()

О, да! Извините, потребовалось некоторое время, чтобы понять, что вы имеете в виду : (

Я немного перефразировал это. Если вас это не устраивает, не стесняйтесь изменить его. Спасибо, что указали на это.

Шмерл, 2010/10/20 16:38 ()

На самом деле предыдущее примечание относится ко всем подобным примерам с :-, := и т.д.

мой, 2011/01/25 10:55 (), 2011/02/01 05:31 ()

у `${ ^^ }` and friends есть параметр шаблона. Но, похоже, ничего сверх `[ ]` этого не работает.

```
TEST="abcABCxyz"

echo ${TEST^[abc]} # измените первый символ, если это ||b||c
echo ${TEST^![abc]} # измените первый символ, если он НЕ является a||b|| c

TEST="привет, мир"
echo ${TEST^^[aeiou]} # измените ЛЮБОЙ символ, который находится в aeiou
```

```
AbcABCxyz
abcABCxyz
Привет, мир
```

Ян Шампера, 2011/02/01 05:30 ()

Я не вижу, что здесь не работает. Вы имели в виду

```
${TECT^[!abc]}
# (вместо ${TEST^![abc]})
```

здесь?

Фредерик Гроуз, 2011/02/11 22:43 ()

В GNU bash версия 4.1.7(1)-release (x86_64-redhat-linux-gnu),

синтаксис "отрицательное значение длины",

```
echo "${MYSTRING:11:-17}"
```

приводит к следующему:

```
bash: -17: выражение подстроки < 0
```

ДЛИНА, похоже, ведет себя как количество элементов массива.

Ян Шампера, 2011/02/12 08:44 ()

Привет, Фредерик,

вы абсолютно правы, это функция Bash 4.2. Я добавил примечание.

Альтаир IV, 2012/01/12 13:02 ()

Хм? [b]${!массив[*]}[/b]/[b]${!шаблоны array[@]}[/b] здесь не перечислены? Они распечатывают список всех существующих индексов массива и могут использоваться для перебора разреженных и ассоциативных массивов.

Ян Шампера, 2012/06/20 21:24 ()

Это указано на странице массивов, но может принадлежать и здесь, да. Это неоднозначная тема, поскольку все расширения, связанные с массивом, являются расширением параметров.

Дэвид Си, 2012/04/27 00:01 ()

В вашем примере кода удаления подстроки вы удаляете только 1 символ на "?". Поэтому пример необходимо обновить, чтобы он отображал "${STRING#??????}" (6 вопросительные знаки). Другим примером контекста может быть echo "${STRING%??????}", которое выдаст вам "Привет".

Ян Шампера, 2012/06/20 21:18 ()

Поздно, но исправлено. Спасибо.

Р.У. Эмерсон II, 2012/12/17 09:58 (), 2014/10/06 04:34 ()

Как мы определяем, содержит ли одна строка другую? Как нам найти положение встроенной подстроки?

Удаление подстроки предлагает решение. Если подстрока присутствует, удаление подстроки изменит длину заключающей строки. Позиция $ tFind может быть получена из разницы в длине.

```
объявить TText="ABCDEFG"
объявить tFind="CD"
объявить tTemp=${TText/"$tFind"/} # $tFind присутствующим в $TTex
t?
объявить -i tFPos # Вычислить позицию $tFind

если проверить "$Text" != "$tTemp"; тогда
echo "Найдено $tFind в $TText"
fi


tTemp=${TText#*"$ tFind"} # "EFG" (из исходных 7 символов осталос
ь 3, поэтому 7-3 = 4 были удалены)
tFPos= ${#TText}-${#tTemp}- ${#tFind} # 7-3-2=2 (Из 4 удаленных,
$ tFind составляет 2, поэтому 2 предшествуют $ tFind)
echo "Найдено $ tFind в позиции $ tFPos" # 2
```

Необходимо заключать в кавычки $tFind в поле шаблона, чтобы предотвратить посторонние эффекты, когда $tFind содержит скобки или другие специальные символы шаблона.

liungkejin, 2013/05/30 06:47 (), 2014/10/06 04:36 ()

только `"${!PREFIX*}"` разделяется первым символом в `IFS` переменной `${!PREFIX*}` - `${!PREFIX@}` и `"${!PREFIX@}"` разделяются пробелом

```
testone=; testtwo=; testthree=;
IFS=":";
echo "${!test*}"
 => testone:testtwo:testthree
echo ${!тест *}
 => testone testtwo testthree
echo ${!test@}
 => testone testtwo testthree
эхо "${!test@}"
 => testone testtwo testthree
```

Дик Гертин (https://sites.google.com/site/dickguertin/home), 2014/06/07 03:40 ()

В разделе "Длина строки" я бы добавил, что параметр может быть системной переменной, такой как позиционные переменные, но с # replacing $. Таким образом, допустимо что-то вроде этого: ${#1} для длины переменной $ 1.

Роман С., 2015/02/23 19:17 ()

Привет всем,

@1st спасибо, что поделились своими идеями и опытом

кто-нибудь знает, если это: `mkdir /Linux/Debian7/DVD{1..10}`

равно этому: `for i in {1..10}; do mkdir /Linux/Debian7/DVD$i; done`

или расширить его до: `mkdir /Linux/Debian7/DVD1 /Linux/Debian7/DVD2 …. /Linux/Debian7/DVD10` *?*

Ян Шампера, 2015/07/09 04:32 ()

Он расширяется до нескольких строк (но в конце дня имеет тот же результат)

Кип Сандерс, 2015/05/07 01:17 ()

Это лучший сайт, который я когда-либо находил с точки зрения демонстрации полезных примеров собственных манипуляций с параметрами bash. Тем не менее, я до сих пор не нашел встроенную замену / извлечение переменной bash (без обращения к sed / awk / grep / expr) для извлечения (вместо удаления) шаблона в середине строкового параметра, например, извлечения "123" из параметров ниже:

my_var="foo_123_bar"

my_var2="foo.123:bar"

Конечно, это можно сделать в два этапа:

$ x=${my_var2/:}*

$ echo ${x/.}*

123

Но есть ли способ добиться этого за один шаг?

Ян Шампера, 2015/07/09 04:35 ()

Вы правы, это можно сделать только в несколько этапов. Возможно, для этого можно написать универсальную функцию (с некоторыми отступами), чтобы объединить необходимые шаги в одну операцию высокого уровня.

Дэн Дуглас, 2015/07/20 04:36 ()

Приемы в этом направлении работают в ограниченных случаях.

```
 $ bash -O extglob -c 'myvar=foo.123:bar; echo "${myvar//@(*.|:*)}"'
123
```

Конечно, это довольно хрупко. Ksh лучше справляется с заменой шаблонов и не жадными кванторами для шаблонов регулярных выражений и оболочек.

```
 $ ksh -c 'myvar=foo.123:bar; echo "${myvar/#*-([!.]).*-([^:]):*/\2}"'
123
$ ksh -c 'myvar=foo.123:bar; echo "${myvar/#~(P)[^.]*?\.([^:]*?):.*/\1}"'
123
```

Обычно я предпочитаю это `BASH_REMATCH`.

```
 $ { bash /dev/fd/3 ; ksh /dev/fd/3; } 3<<\EOF
${KSH_VERSION+'false'} || набор -n BASH_REMATCH=.sh.соответствует
myvar=foo.123:bar
[[ $myvar =~ ^[^.]*\.([^:]+): ]] && echo "${BASH_REMATCH[1]}"
EOF

123
123
```

Борис Батинков, 25.07.2016 23:09 ()

За один шаг вы могли бы просто извлечь только числа:

$ my_var2="foo.123:bar"

$ echo ${my_var2[!0-9]/} 123

Кристофер ЛаФейв, 03.06.2016 01:57 ()

Большое вам спасибо за это. Мне очень нужны были примеры, чтобы лучше познакомиться с расширением параметров, и это сокровищница примеров. :)

Томас, 08.08.2016 03:55 ()

В разделе об удалении подстроки вы могли бы добавить пример "косой черты в конце строки" ${WORKDIR%/} для общего использования. Обратите внимание, что, поскольку подстановочный знак отсутствует, он удаляет косую черту только в том случае, если это на самом деле последний символ.

грац, 2016/12/01 05:27 ()

Вероятно, лучшая страница, которую я когда-либо видел, объясняющая специфику BASH. Отличные примеры, простые для чтения и понимания. Отличная работа !

Valentin Hilbig, 2017/01/15 22:16 ()

Кстати (): Спасибо за компиляцию всего этого. Очень полезно! Но, пожалуйста, исправьте:

`contains only elements that are not the nullstring (the :+ variant)`

должны быть удалены две части

`contains only elements that are ~~**not**~~ the nullstring (the ~~:~~+ variant)`

такой, что он читает

`contains only elements that are the nullstring (the + variant)`

или короче `contains elements (the + variant)`

Доказательство:

```
a=(); echo ${a[@]+this is not output};

a=(); echo ${a[@]:+this is not output};

a=(""); echo ${a[@]+this is output};

a=(""); echo ${a[@]:+this is not output};

a=(.); echo ${a[@]+this is output};

a=(.); echo ${a[@]:+this is output};
```

syntax/pe.txt Последнее редактирование: 2021/12/10 08:12 автор ajrou

Bash Hackers Wiki