Обработка позиционных параметров

Вступление

Настанет день, когда вы захотите предоставить аргументы своим скриптам. Эти аргументы известны как **позиционные параметры**. Некоторые соответствующие специальные параметры описаны ниже:

Параметр(ы)	Описание
`$0`	первый позиционный параметр, эквивалентный `argv[0]` в C, см. **Первый аргумент**
`$FUNCNAME`	имя функции (**внимание**: внутри функции по- `$0` прежнему `$0` является именем оболочки, а **не** именем функции)
`$1 … $9`	список аргументов содержит элементы от 1 до 9
`${10} … ${N}`	список аргументов превышает 9 (обратите внимание на синтаксис расширения параметров!)
`$*`	все позиционные параметры, кроме `$0` , см. **Массовое использование**
`$@`	все позиционные параметры, кроме `$0` , см. **Массовое использование**
`$#`	количество аргументов, не считая `$0`

Эти позиционные параметры точно отражают то, что было задано скрипту при его вызове.

Синтаксический анализ с переключением параметров (например `-h` , для отображения справки) на данном этапе не выполняется.

См. Также словарную статью для "параметра".

Первый аргумент

Самый первый аргумент, к которому вы можете получить доступ, упоминается как `$0` . Обычно он устанавливается на имя скрипта точно так же, как вызывается, и устанавливается при инициализации оболочки:

Testscript - это просто эхо `$0` :

```
#!/bin/bash
echo "$ 0"
```

Видите ли, `$0` всегда устанавливается на имя, с которым вызывается скрипт (`>` это приглашение ...):

```
> ./testscript
./testscript
```

```
> /usr/bin/testscript
/usr/bin/testscript
```

Однако это не относится к оболочкам входа:

```
> echo "$ 0"
-bash
```

Другими словами, `$0` это не позиционный параметр, это специальный параметр, независимый от списка позиционных параметров. Он может быть установлен на что угодно. В **идеальном** случае это путь к скрипту, но поскольку он устанавливается при вызове, вызывающая программа может легко повлиять на него (`login` программа делает это для оболочек входа, например, добавляя тире).

Внутри функции по- `$0` прежнему работает так, как описано выше. Чтобы получить имя функции, используйте `$FUNCNAME` .

Смещение

Встроенная команда `shift` используется для изменения значений позиционных параметров:

- `$1` будут отброшены
- `$2` станет `$1`
- `$3` станет `$2`
- ...
- в общем: `$N` станет `$N-1`

Команда может принимать число в качестве аргумента: количество позиций для сдвига. например `shift 4` , сдвигается `$5` на `$1` .

Их использование

Хватит теории, вы хотите получить доступ к аргументам вашего скрипта. Ну, вот и все.

Один за другим

Один из способов - получить доступ к определенным параметрам:

```
#!/bin/bash
echo "Общее количество аргументов: $#"
echo "Аргумент 1: $ 1"
echo "Аргумент 2: 2 доллара"
echo "Аргумент 3: 3 доллара"
эхо "Аргумент 4: 4 доллара"
эхо "Аргумент 5: 5 долларов"
```

Хотя этот способ полезен в другой ситуации, ему не хватает гибкости. Максимальное количество аргументов - это фиксированное значение, что является плохой идеей, если вы пишете скрипт, который принимает много имен файлов в качестве аргументов.

⇒ забудьте об этом

Циклы

Существует несколько способов перебора позиционных параметров.

Вы можете закодировать цикл for в стиле C, используя `$#` его в качестве конечного значения. На каждой итерации `shift` команда -используется для сдвига списка аргументов:

```
numargs=$#
for ((i=1 ; i <= numargs ; i++))

повторять "$ 1"
 сдвиг
выполнен
```

Не очень стильно, но удобно. `numargs` Переменная используется для хранения начального значения `$#` , поскольку команда shift изменяет его по мере выполнения скрипта.

Другой способ повторения одного аргумента за раз - это `for` цикл без заданного списка слов. Цикл использует позиционные параметры в качестве списка слов:

```
для
аргумента выполните
эхо "$ arg"
```

Преимущество: позиционные параметры будут сохранены

Следующий метод похож на первый пример (`for` цикл), но он не проверяет достижение `$#` . Он сдвигается и проверяет, все ли `$1` еще расширяется до чего-либо, используя команду test:

```
в то время как [ "$ 1" ]

повторять "$ 1"
 сдвиг
выполнен
```

Выглядит красиво, но имеет недостаток остановки, когда `$1` пусто (нулевая строка). Давайте изменим его, чтобы он выполнялся до тех пор, пока `$1` он определен (но может быть нулевым), используя расширение параметра для альтернативного значения:

```
в то время как [ "${1+определено}" ]; выполнить
эхо "$1"
 сдвиг
выполнен
```

Getopts

Существует небольшое руководство, посвященное "getopts" (в *стадии разработки*).

Массовое использование

Все позиционные параметры

Иногда необходимо просто "передать" или "передать" заданные аргументы другой программе. Это очень неэффективно делать в одном из этих циклов, так как вы, скорее всего, нарушите целостность (пробелы!).

Разработчики оболочки создали `$*` and `$@` для этой цели.

В качестве обзора:

Синтаксис	Эффективный результат
`$*`	`$1 $2 $3 … ${N}`
`$@`	`$1 $2 $3 … ${N}`
`"$*"`	`"$1c$2c$3c…c${N}"`
`"$@"`	`"$1" "$2" "$3" … "${N}"`

Без кавычек (двойные кавычки) оба имеют одинаковый эффект: все позиционные параметры от `$1` последнего используемого расширяются без какой-либо специальной обработки.

Когда `$*` специальный параметр заключен в двойные кавычки, он расширяется до эквивалента: `"$1c$2c$3c$4c……..$N"` , где 'c' - первый символ `IFS` .

Но когда `$@` специальный параметр используется внутри двойных кавычек, он расширяется до эквивалента…

```
"$1" "$2" "$3" "$4" …. "$N"
```

... который **отражает все позиционные параметры в том виде, в каком они были установлены изначально** и переданы скрипту или функции. Если вы хотите повторно использовать свои позиционные параметры для **вызова другой программы** (например, в скрипте-оболочке), то это выбор за вами, используйте двойные кавычки `"$@"` .

Ну, давайте просто скажем: **вам почти всегда нужны кавычки `"$@"` !**

Диапазон позиционных параметров

Другой способ массового расширения позиционных параметров аналогичен тому, который возможен для диапазона символов, используя расширение подстроки по обычным параметрам и диапазон массового расширения массивов.

```
${@:START:COUNT}
```

```
${*:START:COUNT}
```

```
"${@:START:COUNT}"
```

```
"${*:START:COUNT}"
```

Правила использования `@` или `*` и цитирования такие же, как указано выше. Это увеличит `COUNT` количество позиционных параметров, начиная с `START` . `COUNT` может быть опущен (`${@:START}`) , и в этом случае все позиционные параметры, начинающиеся с `START` , раскрываются.

Если `START` значение отрицательное, позиционные параметры нумеруются в обратном порядке, начиная с последнего.

`COUNT` не может быть отрицательным, т. Е. Количество элементов не может быть уменьшено.

Пример: НАЧНИТЕ с последнего позиционного параметра:

```
echo "${@: -1}"
```

Внимание: начиная с Bash 4, а `START` of `0` включает специальный параметр `$0` , то есть имя оболочки или любое значение, равное $0, когда используются позиционные параметры. А `START` of `1` начинается с `$1` . В Bash 3 и старше оба `0` и `1` начинались с `$1` .

Настройка позиционных параметров

Установка позиционных параметров с помощью аргументов командной строки - не единственный способ их установки. Встроенная команда set может использоваться для "искусственного" изменения позиционных параметров изнутри скрипта или функции:

```
установите "Это" мой новый "набор" позиционных параметров

# РЕЗУЛЬТАТЫ В
# $ 1: это
# $ 2: мой
# $ 3: новое
# $ 4: набор
# $ 5: позиционный
# $6: параметры
```

При настройке позиционных параметров таким образом разумно сигнализировать "конец опций". В противном случае тире могут быть интерпретированы как переключение параметров само по `set` себе:

```
# оба способа работают, но ведут себя по-разному. Смотрите статью о к
оманде set!
установите -- ...
установить - ...
```

Кроме того, это также сохранит любые подробные (-v) или трассировочные (-x) флаги, которые в противном случае могут быть сброшены `set`

```
set -$- ...
```

Fix Me! продолжить

Примеры производства

Использование цикла while

Чтобы заставить вашу программу принимать параметры в качестве стандартного синтаксиса команды:

`COMMAND [options] <params>` # Нравится 'cat -A file.txt '

Смотрите простой код синтаксического анализа параметров ниже. Это не настолько гибко. Он не автоматически интерпретирует комбинированные параметры (-fu USER), но он работает и является хорошим элементарным способом анализа ваших аргументов.

```
#!/bin/sh
# Сохранение параметров в алфавитном порядке упрощает добавление допо
лнительных.

в то время как:

укажите значение "$ 1" в
 -f | --file)
 file="$2" # Возможно, вы захотите проверить действительность
сдвига $ 2 на 2
 ;;
-h | --help)
 display_help # Вызов вашей функции
 # здесь не требуется никаких изменений, мы закончили.
 выход 0
 ;;
-u | --user)
 username="$ 2" # Возможно, вы захотите проверить действительность $
2
shift 2
 ;;
-v | --подробный)
 # Лучше назначить строку, чем число типа "verbose = 1"
 # потому что, если вы отлаживаете скрипт с помощью кода "bash -x", п
одобного этому:
 #
 # if [ "$verbose" ] ...
 #
 # Вы увидите:
 #
 # if [ "подробный" ] ...
 #
 # Вместо загадочного
 #
 # если [ "1" ] ...
 #
verbose="подробный"
 сдвиг
 ;;
 --) # Конец всех параметров

перерыв сдвига;
 -*)
 echo "Ошибка: неизвестный параметр: $ 1" > & 2
 выход 1
 ;;
 *) # Больше никаких
разрывов параметров
;;
esac
выполнено

# Конец файла
```

Фильтруйте нежелательные параметры с помощью скрипта-оболочки

Эта простая оболочка позволяет фильтровать нежелательные параметры (здесь: `-a` и `–all` для `ls`) из командной строки. Он считывает позиционные параметры и создает отфильтрованный массив, состоящий из них, затем вызывает `ls` с новым набором опций. Он также учитывает `–` как "конец параметров" для `ls` и ничего не меняет после этого:

```
#!/bin/bash

# простая оболочка ls(1), которая не допускает опции -a

options=() # буферный массив для параметров
eoo=0 # достигнут конец параметров

в то время как [[ $ 1 ]]
делай
, если ! ((eoo)); затем
регистр "$ 1" в
 -a)
сдвиг
;;
--все)
 сдвиг
;;
-[^-]*a*|-a?*)
 параметры+=("${1//a}")
 сдвиг
 ;;
 --)
 eoo=1
options+=("$ 1")
 сдвиг
 ;;
 *)
параметры+=("$1")
 shift
;;
esac
else
options+=("$1")

 # Другой (худший) способ сделать то же самое:
 # options=("${параметры[@]}" "$1")
 сдвиг
fi
выполнен

/bin/ls "${options[@]}"
```

Использование getopts

Существует небольшое руководство, посвященное "getopts" (в *стадии разработки*).

Смотрите также

- Внутреннее: небольшое руководство по getopts
- Внутренний: цикл while
- Внутренний: цикл for в стиле C
- Внутренние: массивы (для эквивалентного синтаксиса для массового расширения)
- Внутренний: расширение подстроки для параметра (для эквивалентного синтаксиса для массового расширения)
- Словарь, внутренний: Параметр

Обсуждение

skmdu, 2010/04/14 12:20 (), 2010/04/14 15:13 ()

> Разработчики оболочки изобрели $ * и $ @ для этой цели.

> Без кавычек (двойных кавычек) оба имеют одинаковый эффект: все позиционные параметры от $ 1 до последнего используемого> расширяются, разделяются первым символом IFS (здесь представлен "c", но обычно пробелом):
> $ 1c $ 2c $ 3c $ 4c.....$N

Без двойных кавычек $ * и $ @ расширяют позиционные параметры, разделенные только пробелом, а не IFS.

```
#!/bin/bash

экспорт IFS='-'

echo -e $*
echo -e $@
```

```
$./test "Это" 2 3
Это 2 3
Это 2 3
```

(Отредактировано: вставлены теги кода)

Ян Шампера, 2010/04/14 15:12 ()

Большое вам спасибо за эту находку. Я знаю, как `$*` это работает, поэтому я не могу понять, почему я описал это так неправильно. Я думаю, это было на какой-то поздней ночной сессии.

Еще раз спасибо.

gdh, 2011/02/18 15:11 ()

```
#!/bin/bash

OLDIFS="$IFS" IFS='-' #экспортировать IFS='-'

#echo -e $* #echo -e $ @ #должно быть echo -e "$*" echo -e "$ @"
IFS="$OLDIFS"
```

gdh, 2011/02/18 15:14 ()

```
#должно быть echo -e "$*"
```

Дэйв Карлтон (http://polymicrosystems.com), 2010/05/18 13:23 ()

Я бы предложил использовать другое приглашение, поскольку $ вводит новичков в заблуждение. В противном случае, отличный трактат об использовании позиционных параметров.

Ян Шампера, 2010/05/24 08:48 ()

Спасибо за предложение, я использую "> " здесь сейчас, и я изменю его в любом тексте, который я редактирую в будущем (вся вики). Давайте посмотрим, подходит ли "> ".

херб, 2012/04/20 08:32 ()

Вот еще один способ, не связанный с getopts.

http://bsdpants.blogspot.de/2007/02/option-ize-your-shell-scripts.html (http://bsdpants.blogspot.de/2007/02/option-ize-your-shell-scripts.html)

аборреро, 2012/07/16 12:48 (), 2012/08/12 07:06 ()

Всем привет!

Что, если я использую "$ @" в последующих вызовах функций, но аргументы являются строками?

Я имею в виду, имея:

```
#!/bin/bash
echo "$@"
echo n: $#
```

Если вы используете его

```
mypc $ script arg1 arg2 "asd asd" arg4
arg1 arg2 asd asd arg4
n: 4
```

Но имея

```
#!/bin/bash
myfunc()
{
 echo "$@"
 echo n: $#
}
ech "$@"
echo n: $#
myfunc "$@"
```

вы получаете:

```
mypc $ myscrpt arg1 arg2 "asd asd" arg4
arg1 arg2 asd asd arg4
4
arg1 arg2 asd asd arg4
5
```

Как вы можете видеть, нет способа сообщить функции, что параметр является строкой, а не списком аргументов, разделенных пробелом.

Есть идеи, как это решить? Я тестировал функции вызова и выполнял расширение почти всеми способами без каких-либо результатов.

Ян Шампера, 08.02.2012 07:11 ()

Я не знаю, почему у вас это не получается. Это должно сработать, если вы используете `"$@"` , конечно.

Смотрите пример, с которым я использовал ваш второй скрипт:

```
$ ./args1 a b c "d e" f
a b c d e f
n: 5
a b c d e f
n: 5
```

Jacek Puchta, 2015/06/10 08:00 ()

Большое спасибо за этот урок. Особенно первый пример очень полезен.

scripting/posparams.txt Последнее редактирование: 2018/05/12 18:04 by wayeoyuz

Bash Hackers Wiki