Встроенная команда чтения

прочитайте кое-что о read здесь!

Краткое описание

```
читать [-ers] [-u <FD>] [-t <ТАЙМ-АУТ>] [-p <ПРИГЛАШЕНИЕ>] [-a <МАССИ
В>] [-n <NCHARS>] [-N <NCHARS>] [-d <DELIM>] [-i <ТЕКСТ>] [<ИМЯ ...>]
```

Описание

`read` Встроенная команда считывает **одну строку** данных (текст, пользовательский ввод, ...) из стандартного ввода или предоставленного номера filedescriptor в одну или несколько переменных с именем by `<NAME…>` .

Начиная с версии Bash 4.3-alpha, `read` пропускает любые `NUL` (код ASCII () 0) символы при вводе.

Если `<NAME…>` задано, строка разбивается на слова с использованием переменной IFS, и каждому слову присваивается одно `<NAME>` . Все оставшиеся слова присваиваются последнему `<NAME>` , если присутствует больше слов, чем имен переменных.

> Если `<NAME>` задано значение no, вся прочитанная строка (без выполнения разделения слов!) присваивается переменной оболочки REPLY . Тогда `REPLY` действительно содержит строку в том виде, в каком она была прочитана, без удаления предфиксных и постфиксных пробелов и прочего!
>
> ```
> во время чтения -r; выполните
> printf '"%s"\n' "$REPLY"
> готово <<<" строка с префиксным и постфиксным пробелом "
> ```

Если задан тайм-аут или установлена переменная оболочки TMOUT, он считается с момента первоначального ожидания ввода до завершения ввода (т. Е. До тех пор, пока не будет прочитана полная строка). Это означает, что тайм-аут может возникать и во время ввода.

Опции

Опция	Описание
`-a <ARRAY>`	считайте данные дословно в указанный массив `<ARRAY>` вместо обычных переменных
`-d <DELIM>`	распознать `<DELIM>` как конец данных, а не `<newline>`
`-e`	в интерактивных оболочках: используйте интерфейс чтения строк Bash для чтения данных. Начиная с версии 5.1-alpha, ее также можно использовать для указанных файловых дескрипторов с помощью `-u`
`-i <STRING>`	предварительно загружает входной буфер текстом из `<STRING>` , работает только при использовании Readline (`-e`)
`-n <NCHARS>`	считывает `<NCHARS>` символы ввода, затем завершает работу
`-N <NCHARS>`	считывает `<NCHARS>` символы ввода, *игнорируя любые разделители*, затем завершает работу
`-p <PROMPT>`	строка запроса `<PROMPT>` выводится (без завершающего автоматического перевода строки) перед выполнением чтения
`-r`	Необработанный ввод - **отключает** интерпретацию **экранирования обратной косой** черты и **продолжения строки** в прочитанных данных
`-s`	безопасный ввод - не повторяйте ввод, если на терминале (пароли!)
`-t <TIMEOUT>`	подождите несколько `<TIMEOUT>` секунд, затем завершите работу (код выхода 1). Начиная с Bash 4, разрешены доли секунды ("5.33"). Немедленно возвращается значение 0 и указывает, ожидают ли данные в коде выхода. Тайм-аут обозначается кодом выхода, превышающим 128. Если время ожидания истекает до того, как данные будут прочитаны полностью (до окончания строки), частичные данные сохраняются.
`-u <FD>`	используйте номер filedescriptor `<FD>` вместо `stdin` (0)

Когда заданы оба, `-a <ARRAY>` и имя переменной `<NAME>` , то задается массив, но не переменная.

Конечно, допустимо устанавливать отдельные элементы массива без использования `-a` :

```
прочитайте MYARRAY[5]
```

> Чтение элементов массива с использованием приведенного выше синтаксиса **может привести к расширению имени пути**.
>
> Пример: вы находитесь в каталоге с именем файла `x1` и хотите выполнить чтение в массив `x` , индексировать `1` с помощью

```
прочитайте x[1]
```

затем расширение имени пути расширится до имени `x1` файла и прервет вашу обработку!

Что еще хуже, если `nullglob` установлено значение, ваш массив / индекс исчезнет.

Чтобы избежать этого, либо **отключите расширение имени пути**, либо заключите имя и индекс массива в **кавычки**:

```
прочитайте 'x[1]'
```

Возвращает статус

Статус	Причина
0	ошибки нет
0	ошибка при присвоении переменной, доступной только для чтения [1)]
2	недопустимый параметр
>128	Тайм-аут (см. `-t`)
!=0	неверный файловый дескриптор, предоставленный `-u`
!=0	Достигнут конец файла

чтение без -r

По сути, все, что вам нужно знать, `-r` это **ВСЕГДА** использовать ее. Точное поведение, которое вы получаете, `-r` совершенно бесполезно даже для странных целей. Это в основном позволяет экранировать ввод, который соответствует чему-то в IFS, а также экранирует продолжения строк. Это довольно хорошо объяснено в спецификации чтения POSIX (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/read.html#tag_20_109) ().

```
2012-05-23 13:48:31 гейра, она должна удалять только обратную косую ч
ерту, а не изменять \n и \t и тому подобное на новые строки и вкладки
2012-05-23 13:49:00 ormaaj так вот что делает чтение без -r?
2012-05-23 13:49:16 гейра нет, -r не удаляет обратную косую черту
2012-05-23 13:49:34 ormaaj Я думал, что read <<<'str' эквивалентно re
ad -r <<<$'str'
2012-05-23 13:49:38 гейра # читать x y <<< 'foo\ bar baz'; echo "<$x>
<$y>"
2012-05-23 13:49:40 шбот гейра: <foo bar> <baz>
2012-05-23 13:50:32 гейра нет, чтение без -r в основном бессмысленно.
Проклятый Борн
2012-05-23 13:51:08 ormaaj Так что в основном (полностью) используетс
я для экранирования пробелов
2012-05-23 13:51:24 ormaaj и вставьте новые строки
2012-05-23 13:51:47 гейра ормаадж: в основном вы получаете тот же эфф
ект, что и при использовании \ в командной строке
2012-05-23 13:52:04 geirha echo \" выводит ", считывает x <<< '\"' сч
итывает "
2012-05-23 13:52:32 ormaaj o, странно
2012-05-23 13:52:46 * ормаадж изо всех сил пытается придумать, что к
этому добавить...
2012-05-23 13:53:01 гейра ормаадж: спросите Борна: P
2012-05-23 13:53:20 гейра (не Джейсон)
2012-05-23 13:53:56 ormaaj хм, все равно спасибо :)
```

Примеры

Элементарная замена cat

Элементарная замена `cat` команды: считывание строк ввода из файла и печать их на терминале.

```
opossum() {
при чтении -r; выполните
printf "%s\n" "$REPLY"
 выполнено <"$ 1"
}
```

Примечание: здесь `read -r` и `REPLY` используется значение по умолчанию, потому что мы хотим иметь реальную буквальную строку без каких-либо искажений. `printf` используется, потому что (в зависимости от настроек) `echo` может интерпретировать некоторые экранирования basks lash или переключатели (например `-n`).

Нажмите любую клавишу...

Помните команду MSDOS `pause` ? Вот что-то похожее:

```
pause() {
локальное фиктивное
чтение -s -r -p "Нажмите любую клавишу, чтобы продолжить ..." -n 1 фи
ктивный
}
```

Примечания:

- `-s` для подавления эха терминала (печать)
- `-r` чтобы не интерпретировать специальные символы (например, ожидание второго символа, если кто-то нажмет обратную косую черту)

Чтение столбцов

Простое разделение

Чтение может использоваться для разделения строки:

```
var="один, два, три"
read -r col1 col2 col3 <<< "$var"
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$ col3"
```

Позаботьтесь о том, чтобы вы не могли использовать канал:

```
echo "$var" | read col1 col2 col3 # не работает!
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$ col3"
```

Почему? потому что команды канала выполняются в подоболочках, которые не могут изменять родительскую оболочку. В результате переменные `col1 col2` и `col3` родительской оболочки не изменяются (см. Статью: Bash и дерево процессов).

Если в переменной больше полей, чем переменных, последняя переменная получает оставшуюся часть строки:

```
прочитайте col1 col2 col3 <<< "один два три четыре"
printf "%s \ n" "$ col3" #печатает три четыре
```

Изменение разделителя

По умолчанию чтение разделяет строку в полях с помощью пробелов или табуляции. Вы можете изменить это, используя *специальную переменную* IFS, разделитель внутренних полей.

```
IFS=":" read -r col1 col2 <<< "привет: мир"
printf "col1: %s col2: %s \ n" "$col1" "$col2"
```

Здесь мы используем `var=value command` синтаксис для установки `read` временного окружения. Мы могли бы установить `IFS` нормально, но тогда нам пришлось бы позаботиться о том, чтобы сохранить его значение и восстановить его позже (`OLD=$IFS IFS=":"; read ….;IFS=$OLD`).

Значение по умолчанию `IFS` отличается тем, что 2 поля могут быть разделены одним или несколькими пробелами или табуляцией. Когда вы устанавливаете `IFS` что-то помимо пробела (пробел или табуляция), поля разделяются **ровно** одним символом:

```
IFS=":" read -r col1 col2 col3 <<< "hello::world"
printf "col1: %s col2: %s col3 %s\ n" "$col1" "$ col2" "$ col3"
```

Посмотрите, как `::` в середине фактически определяется дополнительное *пустое поле*.

Поля разделены ровно одним символом, но символ может отличаться для каждого поля:

```
IFS=":|@" read -r col1 col2 col3 col4 <<< "hello:world/in@bash "
printf "col1: %s col2: %s col3 %s col4 %s \n" "$col1" "$col2" "$col3"
"$ col4"
```

Вы уверены?

```
asksure() {
echo -n "Вы уверены (Y / N)?"
при чтении -r -n 1 -s ответ; выполнить
, если [[ $answer = [YyNn] ]]; затем
 [[ $answer = [Yy] ]] && retval=0
 [[ $answer = [Nn] ]] && retval=1
перерыв

выполнен

echo # просто последний перевод строки, оптика...

возвращает $retval
}

### использовать ее
, если спросить, уверен; затем
повторите: "Хорошо, выполняем rm -rf / затем, мастер ...."
еще
эхо "Пффф ..."
fi
```

Запрашивает путь со значением по умолчанию

Примечание: `-i` опция была введена с Bash 4

```
read -e -p "Введите путь к файлу: " -i "/usr/local/etc/" FILEPATH
```

Пользователю будет предложено, он может просто принять значение по умолчанию или отредактировать его.

Многозначные символы: Анализ простой строки даты / времени

Здесь `IFS` содержит двоеточие и пробел. Поля строки даты / времени распознаются правильно.

```
дата и время="2008:07:04 00:34:45"
IFS=": " read -r год месяц день час минута секунда <<< "$datetime"
```

Соображения о переносимости

- В POSIX® указана только `-r` опция (необработанное чтение); `-r` это не только POSIX, вы можете найти его в более раннем исходном коде Bourne
- POSIX® не поддерживает массивы
- `REPLY` это не POSIX®, вам нужно установить `IFS` пустую строку, чтобы получить всю строку для оболочек, которые не знают `REPLY` .

```
в то время как IFS= строка read -r; выполните
 ...
готово < text.txt
```

Смотрите также

- Внутренняя: встроенная команда printf

[1)] исправлено в 4.2-rc1

Обсуждение

Дэн Дуглас, 2011/09/13 06:07 ()

Что-то немного подозрительное происходит с тем, как анализируются аргументы чтения. На самом деле это произвольные расширения.

```
$ x='['; y=']'; z='+++'; в то время как read -r "a $ {x} v$ {z%+}
$y"; do :; сделано < <(printf '%s \ n' {a..d}); echo "${a[@]}";
a b c d
```

Строка расширяется, передается в read, а затем вычисляется. Таким образом, расширения здесь происходят дважды, предполагая, что это двойные, а не одинарные кавычки.

Другой пример предварительной инициализации переменной в арифметическом контексте:

```
$ while read -r "var[${x:=5}, x ++]"; сделать :; сделано < <(prin
tf '%s\n' {a..c}); для i в "${!var[@]}"; выполнить printf '(%d, %
s), ' "$i" "${var[$i]}"; выполнено;
(5, a), (6, b), (7, c), (8, ),
```

Однако, если это все, что нужно было сделать, вы бы не ожидали, что это сработает:

```
при чтении -r "var[$((x++))]"; выполняем ...
```

Здесь используется какой-то уникальный режим оценки, поскольку Bash знает, что не следует выполнять арифметическое расширение перед передачей результирующей строки в read, иначе результатом было бы просто выполнить все присвоения var[0] .

PS ... предварительный просмотр удаляет все "плюсы". Надеюсь, вышесказанное понятно.

Ян Шампера, 2011/09/20 17:38 ()

Привет,

извините за задержку

Само по себе это не "двойное расширение", поведение вполне понятно (но я думаю, вы это знаете).

Спасибо за этот очень интересный spotlight, я никогда так много об этом не думал!

Гриша, 2015/09/04 06:45 ()

> Bash знает, что не следует выполнять арифметическое расширение перед передачей результирующей строки в read

Это потому `read` , что является частью `while` команды, поэтому расширения не выполняются до тех пор, пока команда не будет выполнена для каждой итерации цикла.

Андор, 2012/06/06 14:46 ()

Прочитайте, есть некоторые ошибки.

http://www.redhat.com/mirrors/LDP/LDP/abs/html/gotchas.html (http://www.redhat.com/mirrors/LDP/LDP/abs/html/gotchas.html)

Ян Шампера, 07.07.2012 11:43 ()

Да, это связано не напрямую с `read` , а с дизайном Bash в отношении конвейерной обработки. Кстати, в списке рассылки было обсуждение, чтобы привести поведение в соответствие с тем, что делает Korn - запуск последнего элемента конвейера в текущей среде выполнения.

ABS - хорошая коллекция и хорошая работа, показывающая, как использовать оболочку, но не переоценивайте ее техническую сторону.

мусорщик, 2013/02/18 09:49 (), 2013/12/31 09:32 ()

спасибо, bash, за запуск вспомогательной оболочки при конвейере, теперь мы больше не можем читать несколько переменных одновременно!

```
grep -w regexp file | read a b c
```

нет решения для замены этой функциональности KSH. Команда: `read -r a b c «<$(command)` решение несовместимо с bourne и korn shell (88 и 93).

dannysauer, 2016/05/26 15:42 ()

И bash, и ksh93 могут выполнять подстановку процессов, например `read a b c < <(some command)` (обратите внимание, что между первым < , который перенаправляет STDIN , и вторым < , который указывает на выполнение команды и подключение ее к именованному каналу, предоставляя канал вместо `<(cmd)` выражения).

Вы можете использовать это вместо канала практически в любое время. Я использую ее все время для таких вещей, как `diff <(sort file1 | grep -v '^#') <(sort file2 | grep -v '^#')` , которая сравнивает два файла, игнорируя комментарии и сортируя строки в файлах перед сравнением. :)

BobD (http://ezilidanto.com), 2015/01/12 17:42 ()

мусорщик, как насчет?–

echo a b c | read -r -a a ; echo ${a[@]} ${#a[@]}

a b c 3