Бесплатная электронная книга

# УЧУСЬ

# Bash

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#bash

# Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: bash

It is an unofficial and free Bash ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Bash.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# глава 1: Начало работы с Bash

## Версии

| Версия | Дата выхода |
|---|---|
| 0,99 | 1989-06-08 |
| 1,01 | 1989-06-23 |
| 2,0 | 1996-12-31 |
| 2,02 | 1998-04-20 |
| 2,03 | 1999-02-19 |
| 2,04 | 2001-03-21 |
| 2.05b | 2002-07-17 |
| 3.0 | 2004-08-03 |
| 3,1 | 2005-12-08 |
| 3,2 | 2006-10-11 |
| 4,0 | 2009-02-20 |
| 4,1 | 2009-12-31 |
| 4,2 | 2011-02-13 |
| 4,3 | 2014-02-26 |
| 4,4 | 2016-09-15 |

## Examples

### Привет, мир, используя переменные

Создайте новый файл `hello.sh` со следующим содержимым и дайте ему исполняемые разрешения с помощью `chmod +x hello.sh` .

Выполнить / `./hello.sh` через: `./hello.sh`

```
#!/usr/bin/env bash
```

```
# Note that spaces cannot be used around the `=` assignment operator
whom_variable="World"

# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

Это приведет к печати `Hello, World` до стандартного вывода при его выполнении.

Чтобы сообщить bash, где сценарий вам нужен, нужно указать его в содержащую директорию, обычно с `./` если это ваш рабочий каталог, где `.` является псевдонимом текущего каталога. Если вы не укажете каталог, `bash` попытается найти скрипт в одном из каталогов, содержащихся в `$PATH` среды `$PATH` .

---

Следующий код принимает аргумент `$1` , который является первым аргументом командной строки, и выводит его в отформатированной строке, следующей за `Hello,` .

> Выполнение / выполнение через: `./hello.sh World`

```
#!/usr/bin/env bash
printf "Hello, %s\n" "$1"
#> Hello, World
```

Важно отметить, что `$1` должен быть указан в двойной кавычки, а не одинарной кавычки. `"$1"` по желанию расшифровывается до первого аргумента командной строки, а `'$1'` вычисляется до литеральной строки `$1` .

> **Примечание по безопасности:**
> Прочтите **последствия для безопасности, забывая процитировать переменную в оболочках bash,** чтобы понять важность размещения текста переменной в двойных кавычках.

### Привет, мир

# Интерактивная оболочка

Оболочка Bash обычно используется в **интерактивном режиме:** она позволяет вводить и редактировать команды, а затем выполняет их при нажатии клавиши `Return` . Многие Unix-подобные и Unix-подобные операционные системы используют Bash в качестве своей оболочки по умолчанию (особенно Linux и macOS). Терминал автоматически запускает интерактивный процесс оболочки Bash при запуске.

Выведите `Hello World` , введя следующее:

```
echo "Hello World"
#> Hello World  # Output Example
```

## Заметки

- Вы можете изменить оболочку, просто набрав имя оболочки в терминале. Например: `sh` , `bash` и т. Д.
- `echo` - это встроенная команда Bash, которая записывает полученные аргументы в стандартный вывод. По умолчанию он добавляет новую строку к выводу.

---

# Неинтерактивная оболочка

Оболочка Bash также может быть запущена **неинтерактивно** из сценария, в результате чего оболочка не требует взаимодействия с человеком. Интерактивное поведение и сценарий поведения должны быть одинаковыми - важное рассмотрение дизайна оболочки Unix V7 Bourne и транзитно Bash. Поэтому все, что можно сделать в командной строке, можно поместить в файл сценария для повторного использования.

Выполните следующие шаги, чтобы создать сценарий `Hello World` :

1. Создайте новый файл `hello-world.sh`

   ```
   touch hello-world.sh
   ```

2. Сделайте исполняемый файл скрипта, запустив `chmod +x hello-world.sh`

3. Добавьте этот код:

   ```
   #!/bin/bash
   echo "Hello World"
   ```

   **Строка 1** : первая строка скрипта должна начинаться с символьной последовательности `#!` , называемый *shebang* [1] . Shebang поручает операционной системе запустить `/bin/bash` , оболочку Bash, передав ей путь сценария в качестве аргумента.

   Eg `/bin/bash hello-world.sh`

   **Строка 2** : используется команда `echo` для записи `Hello World` на стандартный вывод.

4. Выполните скрипт `hello-world.sh` из командной строки, используя одно из следующих:

   - `./hello-world.sh` - наиболее часто используемые и рекомендуемые

- `/bin/bash hello-world.sh`
- `bash hello-world.sh` - предполагая, что `/bin` находится в вашей `$PATH`
- `sh hello-world.sh`

Для реального использования в производстве вы бы `.sh` расширение `.sh` (что вводит в заблуждение в любом случае, поскольку это скрипт Bash, а не сценарий `sh` ) и, возможно, переместите файл в каталог в вашей `PATH` чтобы он был доступен вам независимо от того, ваш текущий рабочий каталог, как и системная команда, например `cat` или `ls` .

К числу распространенных ошибок относятся:

1. Забыв применить к файлу разрешение на выполнение, то есть `chmod +x hello-world.sh` , в результате `./hello-world.sh: Permission denied` выход `./hello-world.sh: Permission denied` .

2. Редактирование сценария в Windows, который создает неправильные символы окончания строки, которые Bash не может обрабатывать.

   Общим симптомом является `: command not found` где возврат каретки заставил курсор к началу строки, переписав текст перед двоеточием в сообщении об ошибке.

   Сценарий можно исправить с `dos2unix` программы `dos2unix` .

   Пример использования: `dos2unix hello-world.sh`

   *`dos2unix` редактирует файл inline.*

3. Используя `sh ./hello-world.sh` , не понимая, что `bash` и `sh` представляют собой отдельные оболочки с различными функциями (хотя, поскольку Bash совместим с обратной связью, противоположная ошибка безвредна).

   Во всяком случае, просто полагаясь на строку shebang скрипта, гораздо предпочтительнее явно писать `bash` или `sh` (или `python` или `perl` или `awk` или `ruby` или ...) перед именем файла каждого скрипта.

   Обычная строка shebang, используемая для того, чтобы сделать ваш скрипт более переносимым, заключается в использовании `#!/usr/bin/env bash` вместо жесткого кодирования пути к Bash. Таким образом, `/usr/bin/env` должен существовать, но помимо этого, `bash` просто должен быть на вашем `PATH` . На многих системах `/bin/bash` не существует, и вы должны использовать `/usr/local/bin/bash` или какой-либо другой абсолютный путь; это изменение позволяет избежать необходимости выяснять детали этого.

---

[1] *Также упоминается как sha-bang, hashbang, pound-bang, hash-pling.*

## Просмотр информации для встроенных модулей Bash

```
help <command>
```

Это отобразит страницу справки Bash (ручная) для указанного встроенного устройства.

Например, `help unset` покажет:

```
unset: unset [-f] [-v] [-n] [name ...]
   Unset values and attributes of shell variables and functions.

   For each NAME, remove the corresponding variable or function.

   Options:
     -f    treat each NAME as a shell function
     -v    treat each NAME as a shell variable
     -n    treat each NAME as a name reference and unset the variable itself
       rather than the variable it references

   Without options, unset first tries to unset a variable, and if that fails,
   tries to unset a function.

   Some variables cannot be unset; also see `readonly'.

   Exit Status:
   Returns success unless an invalid option is given or a NAME is read-only.
```

Чтобы просмотреть список всех встроенных модулей с кратким описанием, используйте

```
help -d
```

## Hello World с пользовательским вводом

Следующее предложит пользователю ввести, а затем сохранит этот ввод в виде строки (текста) в переменной. Затем переменная используется для предоставления пользователю сообщения.

```
#!/usr/bin/env bash
echo  "Who are you?"
read name
echo "Hello, $name."
```

Команда `read` здесь читает одну строки данных из стандартного ввода в переменное `name` . Затем это делается с помощью `$name` и печатается по стандарту, используя `echo` .

Пример вывода:

```
$ ./hello_world.sh
Who are you?
Matt
Hello, Matt.
```

Здесь пользователь ввел имя «Мэтт», и этот код использовался, чтобы сказать « `Hello, Matt.`

,

И если вы хотите добавить что-то к значению переменной во время его печати, используйте фигурные скобки вокруг имени переменной, как показано в следующем примере:

```
#!/usr/bin/env bash
echo  "What are you doing?"
read action
echo "You are ${action}ing."
```

Пример вывода:

```
$ ./hello_world.sh
What are you doing?
Sleep
You are Sleeping.
```

Здесь, когда пользователь вводит действие, во время печати к этому действию добавляется «ing».

## Обработка именованных аргументов

```
#!/bin/bash

deploy=false
uglify=false

while (( $# > 1 )); do case $1 in
   --deploy) deploy="$2";;
   --uglify) uglify="$2";;
   *) break;
 esac; shift 2
done

$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"

# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

## Hello World в режиме «Отладка»

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

Аргумент `-x` позволяет вам пройти через каждую строку в скрипте. Один хороший пример:

```
$ cat hello.sh
#!/bin/bash
echo "Hello World\n"
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)

$ ./hello.sh
Hello World

expr: non-integer argument
```

Вышеприведенная ошибка не достаточна для отслеживания сценария; однако, используя следующий способ, вы можете лучше понять, где искать ошибку в скрипте.

```
$ bash -x hello.sh
+ echo Hello World\n
Hello World

+ adding_string_to_number=s
+ expr 5 + s
expr: non-integer argument
+ v=
```

### Важность цитаты в строках

Цитирование важно для расширения строки в bash. С их помощью вы можете контролировать, как bash анализирует и расширяет ваши строки.

## Существует два типа цитирования:

- **Слабый** : *использует двойные кавычки:*
- **Сильный** : *использует одинарные кавычки:*

Если вы хотите использовать bash, чтобы расширить свой аргумент, вы можете использовать **Weak Quoting** :

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

Если вы не хотите использовать bash для расширения своего аргумента, вы можете использовать **Strong Quoting** :

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
#> Hello $world
```

Вы также можете использовать escape для предотвращения расширения:

```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

---

Для получения более подробной информации, кроме деталей новичка, вы можете продолжить читать ее здесь .

Прочитайте Начало работы с Bash онлайн: https://riptutorial.com/ru/bash/topic/300/начало-работы-с-bash

# глава 2: Aliasing

## Вступление

Shell aliases - простой способ создать новые команды или обернуть существующие команды с помощью собственного кода. Они несколько перекрываются с функциями оболочки, которые, однако, более универсальны и поэтому часто предпочтительнее.

## замечания

Псевдоним будет доступен только в оболочке, где была выдана команда alias.

Чтобы сохранить псевдоним, положите его в свой `.bashrc`

## Examples

### Создать псевдоним

```
alias word='command'
```

Вызов `word` будет выполнять `command` . Любые аргументы, предоставленные псевдониму, просто добавляются к цели псевдонима:

```
alias myAlias='some command --with --options'
myAlias foo bar baz
```

Затем оболочка выполнит:

```
some command --with --options foo bar baz
```

Чтобы включить несколько команд в один и тот же псевдоним, вы можете связать их вместе с `&&` . Например:

```
alias print_things='echo "foo" && echo "bar" && echo "baz"'
```

### Список всех псевдонимов

```
alias -p
```

будут перечислены все текущие псевдонимы.

### Развернуть псевдоним

Предположим, что `bar` является псевдонимом для `someCommand -flag1` .

Введите `bar` в командной строке, а затем нажмите `Ctrl` + `alt` + `e`

вы получите `someCommand -flag1` где стоял `bar` .

### Удалить псевдоним

Чтобы удалить существующий псевдоним, используйте:

```
unalias {alias_name}
```

Пример:

```
# create an alias
$ alias now='date'

# preview the alias
$ now
Thu Jul 21 17:11:25 CEST 2016

# remove the alias
$ unalias now

# test if removed
$ now
-bash: now: command not found
```

### Обход псевдонима

Иногда вы можете временно обойти псевдоним, не отключая его. Чтобы работать с конкретным примером, рассмотрите этот псевдоним:

```
alias ls='ls --color=auto'
```

И, допустим, вы хотите использовать команду `ls` не отключая псевдоним. У вас есть несколько вариантов:

- Используйте `command` builtin: `command ls`
- Используйте полный путь к команде: `/bin/ls`
- Добавьте имя `\` любом месте в имени команды, например: `\ls` или `l\s`
- Отправьте команду: `"ls"` или `'ls'`

### BASH_ALIASES - это внутренний байт-массив

Псевдонимы называются ярлыками команд, которые можно определить и использовать в интерактивных экземплярах bash. Они хранятся в ассоциативном массиве с именем BASH_ALIASES. Чтобы использовать этот var в скрипте, он должен запускаться в

интерактивной оболочке

```
#!/bin/bash -li
# note the -li above! -l makes this behave like a login shell
# -i makes it behave like an interactive shell
#
# shopt -s expand_aliases will not work in most cases

echo There are ${#BASH_ALIASES[*]} aliases defined.

for ali in "${!BASH_ALIASES[@]}"; do
   printf "alias: %-10s triggers: %s\n" "$ali" "${BASH_ALIASES[$ali]}"
done
```

Прочитайте Aliasing онлайн: https://riptutorial.com/ru/bash/topic/368/aliasing

# глава 3: Bash на Windows 10

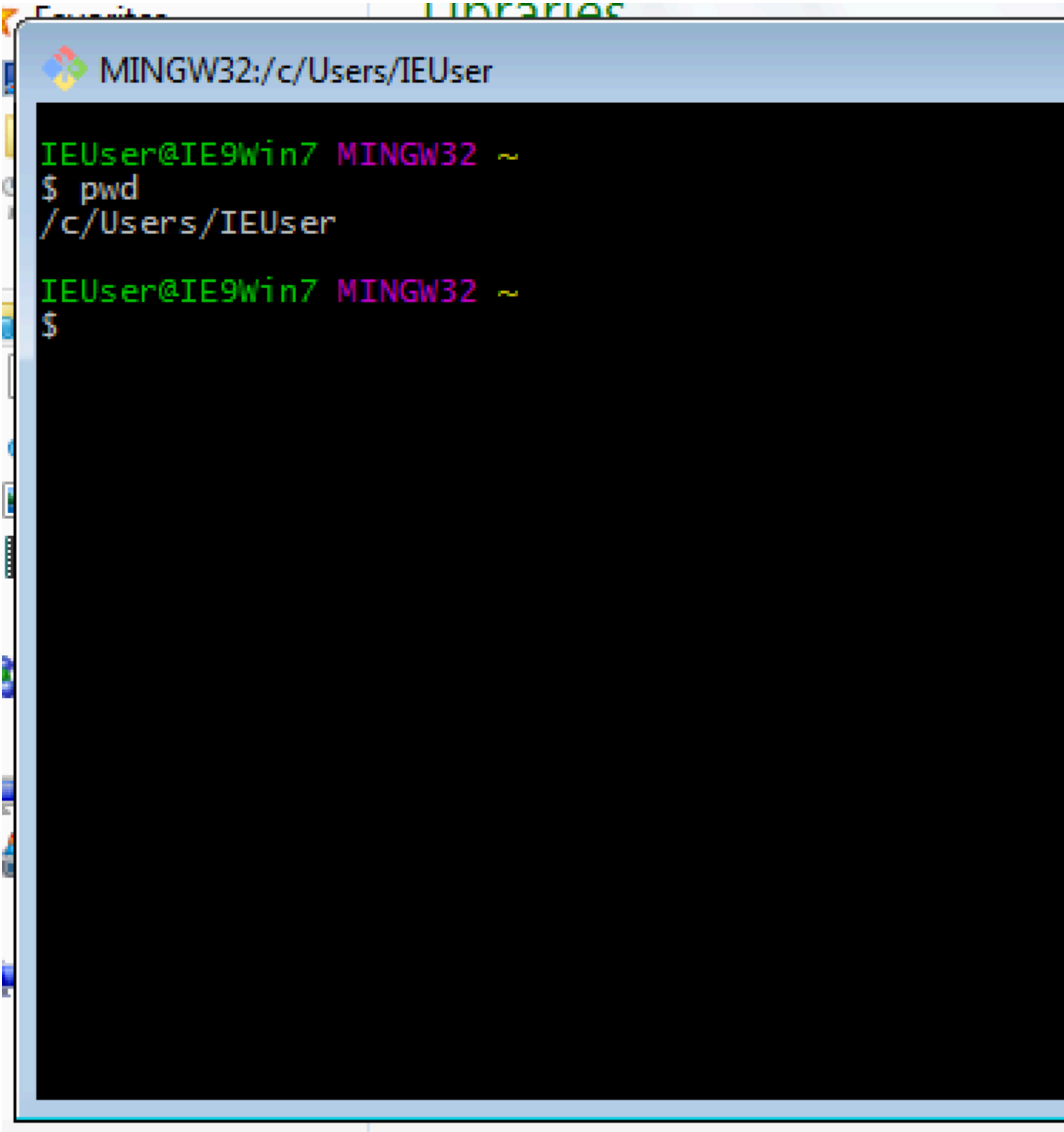
## Examples

### Прочти меня

Более простой способ использования Bash в Windows - установить Git для Windows. Он поставляется с Git Bash, который является настоящим Bash. Вы можете получить доступ к нему с помощью ярлыка в:

```
Start > All Programs > Git > Git Bash
```

Работают команды `grep` , `ls` , `find` , `sed` , `vi` т. Д.




Прочитайте Bash на Windows 10 онлайн: https://riptutorial.com/ru/bash/topic/9114/bash-на-windows-10

# глава 4: getopts: интеллектуальный анализ позиционных параметров

## Синтаксис

- getopts optstring name [args]

## параметры

| параметр | подробность |
|---|---|
| строка_опций | Символы опций, подлежащие распознаванию |
| название | Затем имя, где хранится опция синтаксического анализа |

## замечания

# Опции

`optstring` : символы опций, подлежащие распознаванию

> Если за символом следует двоеточие, ожидается, что параметр будет иметь аргумент, который должен быть отделен от него пробелом. Двоеточие ( `:` ) (и знак вопроса `?` ) Не могут быть использованы в качестве опций символов.

Каждый раз, когда он вызывается, `getopts` помещает следующий параметр в имя переменной оболочки, инициализирует имя, если оно не существует, и индекс следующего аргумента, который должен быть обработан в переменной `OPTIND` . `OPTIND` инициализируется до `1` каждом `OPTIND` оболочки или скрипта оболочки.

Когда опция требует аргумента, `getopts` помещает этот аргумент в переменную `OPTARG` . Оболочка не сбрасывает `OPTIND` автоматически; он должен быть сброшен вручную между несколькими вызовами в `getopts` в рамках одного и того же вызова оболочки, если будет использоваться новый набор параметров.

Когда встречается конец опций, `getopts` выходит с возвратным значением, большим нуля.

`OPTIND` устанавливается в индекс первого аргумента без аргумента, а для имени установлено значение `?` , `getopts` обычно анализирует позиционные параметры, но если в

`args` указано больше аргументов, `getopts` анализирует их.

`getopts` может сообщать об ошибках двумя способами. Если первый символ `optstring` является двоеточием ( `:` ), используются бесшумные сообщения об ошибках. При нормальной работе диагностические сообщения печатаются, когда встречаются недопустимые параметры или отсутствующие аргументы параметров.

Если для переменной `OPTERR` установлено значение `0` , сообщения об ошибках не будут отображаться, даже если первый символ `optstring` не является двоеточием.

Если отображается недопустимая опция, `getopts` места `?` на `name` и, если не молчать, выводит сообщение об ошибке и отключает `OPTARG` . Если `getopts` работает, найденный символ опции помещается в `OPTARG` и никакое диагностическое сообщение не печатается.

Если требуемый аргумент не найден, а `getopts` не является тихим, знак вопроса ( `?` ) Помещается в `name` , `OPTARG` , и `OPTARG` диагностическое сообщение. Если `getopts` молчит, затем двоеточие ( `:` ) помещаются в имени и `OPTARG` устанавливаются на символ опции.

# Examples

## pingnmap

```
#!/bin/bash
# Script name : pingnmap
# Scenario : The systems admin in company X is tired of the monotonous job
# of pinging and nmapping, so he decided to simplify the job using a script.
# The tasks he wish to achieve is
# 1. Ping - with a max count of 5 -the given IP address/domain. AND/OR
# 2. Check if a particular port is open with a given IP address/domain.
# And getopts is for her rescue.
# A brief overview of the options
# n : meant for nmap
# t : meant for ping
# i : The option to enter the IP address
# p : The option to enter the port
# v : The option to get the script version


while getopts ':nti:p:v' opt
#putting : in the beginnnig suppresses the errors for invalid options
do
case "$opt" in
   'i')ip="${OPTARG}"
       ;;
   'p')port="${OPTARG}"
       ;;
   'n')nmap_yes=1;
       ;;
   't')ping_yes=1;
       ;;
   'v')echo "pingnmap version 1.0.0"
       ;;
    *) echo "Invalid option $opt"
```

```
       echo "Usage : "
       echo "pingmap -[n|t[i|p]|v]"
       ;;
esac
done
if  [ ! -z "$nmap_yes" ] && [ "$nmap_yes" -eq "1" ]
then
   if [ ! -z "$ip" ] && [ ! -z "$port" ]
   then
     nmap -p "$port" "$ip"
   fi
fi

if  [ ! -z "$ping_yes" ] && [ "$ping_yes" -eq "1" ]
then
   if [ ! -z "$ip" ]
   then
     ping -c 5 "$ip"
   fi
fi
shift $(( OPTIND - 1 )) # Processing additional arguments
if [ ! -z "$@" ]
then
  echo "Bogus arguments at the end : $@"
fi
```

### Выход

```
$ ./pingnmap -nt -i google.com -p 80

Starting Nmap 6.40 ( http://nmap.org ) at 2016-07-23 14:31 IST
Nmap scan report for google.com (216.58.197.78)
Host is up (0.034s latency).
rDNS record for 216.58.197.78: maa03s21-in-f14.1e100.net
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.22 seconds
PING google.com (216.58.197.78) 56(84) bytes of data.
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=1 ttl=57 time=29.3 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=2 ttl=57 time=30.9 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=3 ttl=57 time=34.7 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=4 ttl=57 time=39.6 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=5 ttl=57 time=32.7 ms

--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 29.342/33.481/39.631/3.576 ms
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
```

Прочитайте getopts: интеллектуальный анализ позиционных параметров онлайн: https://riptutorial.com/ru/bash/topic/3654/getopts--интеллектуальный-анализ-позиционных-параметров

# глава 5: Grep

## Синтаксис

- grep [ОПЦИИ] PATTERN [FILE ...]

## Examples

### Как искать файл для шаблона

Чтобы найти слово **foo** в *строке* файла:

```
grep foo ~/Desktop/bar
```

Чтобы найти все строки, которые **не** содержат foo в *панели* файлов:

```
grep -v foo ~/Desktop/bar
```

Чтобы использовать все слова, содержащие foo в конце (расширение WIldcard):

```
grep "*foo" ~/Desktop/bar
```

Прочитайте Grep онлайн: https://riptutorial.com/ru/bash/topic/10852/grep

# глава 6: Sourcing

## Examples

### Поиск файла

Поиск файла отличается от исполнения, поскольку все команды оцениваются в контексте текущего сеанса bash - это означает, что любые определенные переменные, функции или псевдонимы будут сохраняться на протяжении всего сеанса.

Создайте файл, который вы хотите использовать `sourceme.sh`

```
#!/bin/bash

export A="hello_world"
alias sayHi="echo Hi"
sayHello() {
    echo Hello
}
```

Из вашей сессии укажите файл

```
$ source sourceme.sh
```

С этого момента у вас есть все ресурсы доступного файла

```
$ echo $A
hello_world

$ sayHi
Hi

$ sayHello
Hello
```

Обратите внимание, что команда `.` является синонимом `source` , так что вы можете просто использовать

```
$ . sourceme.sh
```

### Поиск виртуальной среды

При разработке нескольких приложений на одной машине становится целесообразным разграничение зависимостей в виртуальных средах.

С помощью `virtualenv` эти среды поставляются в вашу оболочку, поэтому при запуске команды она исходит из этой виртуальной среды.

Это чаще всего устанавливается с помощью `pip` .

```
pip install https://github.com/pypa/virtualenv/tarball/15.0.2
```

Создать новую среду

```
virtualenv --python=python3.5 my_env
```

Активировать среду

```
source my_env/bin/activate
```

Прочитайте Sourcing онлайн: https://riptutorial.com/ru/bash/topic/564/sourcing

# глава 7: true, false и: команды

## Синтаксис

- true,: - всегда возвращает 0 в качестве кода выхода.
- false - всегда возвращает 1 в качестве кода выхода.

## Examples

### Бесконечный цикл

```
while true; do
    echo ok
done
```

или же

```
while :; do
   echo ok
done
```

или же

```
until false; do
    echo ok
done
```

### Возврат функции

```
function positive() {
    return 0
}

function negative() {
    return 1
}
```

### Код, который будет / никогда не выполняться

```
if true; then
    echo Always executed
fi
if false; then
    echo Never executed
fi
```

Прочитайте true, false и: команды онлайн: https://riptutorial.com/ru/bash/topic/6655/true--false-и--команды

# глава 8: Арифметика Bash

## Синтаксис

- $ ((EXPRESSION)) - оценивает выражение и возвращает его результат.
- expr EXPRESSION - выводит результат EXPRESSION на stdout.

## параметры

| параметр | подробности |
| --- | --- |
| ЭКСПРЕССИЯ | Выражение для оценки |

## замечания

Между каждым термином (или знаком) выражения требуется пространство (""). «1 + 2» не будет работать, но «1 + 2» будет работать.

## Examples

### Арифметическая команда

- `let`

```
let num=1+2
let num="1+2"
let 'num= 1 + 2'
let num=1 num+=2
```

Вам нужны кавычки, если есть пробелы или символы подстановки. Так получится ошибка:

```
let num = 1 + 2    #wrong
let 'num = 1 + 2'  #right
let a[1] = 1 + 1   #wrong
let 'a[1] = 1 + 1' #right
```

- `(( ))`

```
((a=$a+1))     #add 1 to a
((a = a + 1))  #like above
((a += 1))     #like above
```

Мы можем использовать `(())` в `if` . Пример:

```
if (( a > 1 )); then echo "a is greater than 1"; fi
```

Вывод функции `(())` может быть присвоен переменной:

```
result=$((a + 1))
```

Или используется непосредственно в выводе:

```
echo "The result of a + 1 is $((a + 1))"
```

## Простая арифметика с (())

```
#!/bin/bash
echo $(( 1 + 2 ))
```

Выход: 3

```
# Using variables
#!/bin/bash
var1=4
var2=5
((output=$var1 * $var2))
printf "%d\n" "$output"
```

Выход: 20

## Простая арифметика с expr

```
#!/bin/bash
expr 1 + 2
```

Выход: 3

Прочитайте Арифметика Bash онлайн: https://riptutorial.com/ru/bash/topic/3652/арифметика-bash

# глава 9: Ассоциативные массивы

## Синтаксис

- declare -A_com_rray # без инициализации
- declare -A assoc_array = ([key] = "value" [еще один ключ] = "продумать пробелы" [три пробела] = "все пробелы суммируются")
- echo $ {assoc_array [@]} # значения
- echo $ {! assoc_array [@]} # ключи

## Examples

### Изучение массивов

Все необходимое использование показано с помощью этого фрагмента:

```
#!/usr/bin/env bash

declare -A assoc_array=([key_string]=value                                  \
                        [one]="something"                                   \
                        [two]="another thing"                               \
                        [ three ]='mind the blanks!'                        \
                        [ " four" ]='count the blanks of this key later!'   \
                                            [IMPORTANT]='SPACES DO ADD UP!!!'
\
                        [1]='there are no integers!'                        \
                        [info]="to avoid history expansion "                \
                        [info2]="quote exclamation mark with single quotes" \
                       )
echo # just a blank line
echo now here are the values of assoc_array:
echo ${assoc_array[@]}
echo not that useful,
echo # just a blank line
echo this is better:

declare -p assoc_array    # -p == print

echo have a close look at the spaces above\!\!\!
echo # just a blank line

echo accessing the keys
echo the keys in assoc_array are ${!assoc_array[*]}
echo mind the use of indirection operator \!
echo # just a blank line


echo now we loop over the assoc_array line by line
echo note the \! indirection operator which works differently,
echo if used with assoc_array.
echo # just a blank line
```

```
for key in "${!assoc_array[@]}"; do # accessing keys using ! indirection!!!!
   printf "key: \"%s\"\nvalue: \"%s\"\n\n" "$key" "${assoc_array[$key]}"
done


echo have a close look at the spaces in entries with keys two, three and four above\!\!\!
echo # just a blank line
echo # just another blank line

echo there is a difference using integers as keys\!\!\!
i=1
echo declaring an integer var i=1
echo # just a blank line
echo Within an integer_array bash recognizes artithmetic context.
echo Within an assoc_array bash DOES NOT recognize artithmetic context.
echo # just a blank line
echo this works: \${assoc_array[\$i]}:  ${assoc_array[$i]}
echo this NOT!!: \${assoc_array[i]}:  ${assoc_array[i]}
echo # just a blank line
echo # just a blank line
echo an \${assoc_array[i]} has a string context within braces in contrast to an integer_array
declare -i integer_array=( one two three )
echo "doing a: declare -i integer_array=( one two three )"
echo # just a blank line

echo both forms do work: \${integer_array[i]} : ${integer_array[i]}
echo and this too:  \${integer_array[\$i]} : ${integer_array[$i]}
```

Прочитайте Ассоциативные массивы онлайн: https://riptutorial.com/ru/bash/topic/7536/ассоциативные-массивы

# глава 10: Ввод переменных

## Examples

### объявлять слабо типизированные переменные

**declare** - внутренняя команда bash. (внутреннее использование команды **помощь** для отображения «страницы руководства»). Он используется для отображения и определения переменных или отображения функций.

Синтаксис: **declare [options] [name [= value]] ...**

```
# options are used to define
# an integer
declare -i myInteger
declare -i anotherInt=10
# an array with values
declare -a anArray=( one two three)
# an assoc Array
declare -A assocArray=( [element1]="something" [second]=anotherthing )
# note that bash recognizes the string context within []

# some modifiers exist
# uppercase content
declare -u big='this will be uppercase'
# same for lower case
declare -l small='THIS WILL BE LOWERCASE'

# readonly array
declare -ra constarray=( eternal true and unchangeable )

# export integer to environment
declare -xi importantInt=42
```

Вы также можете использовать +, который убирает данный атрибут. В основном бесполезно, просто для полноты.

Для отображения переменных и / или функций есть некоторые опции

```
# printing definded vars and functions
declare -f
# restrict output to functions only
declare -F # if debugging prints line number and filename defined in too
```

Прочитайте Ввод переменных онлайн: https://riptutorial.com/ru/bash/topic/7195/ввод-переменных

# глава 11: Внутренние переменные

## Вступление

Обзор внутренних переменных Bash, где, как и когда их использовать.

## Examples

### Внутренние переменные Bash с первого взгляда

| переменная | подробности |
|---|---|
| `$*` / `$@` | Позиционные параметры функции / скрипта (аргументы). Развернитесь следующим образом:<br><br>`$*` и `$@` такие же, как `$1 $2 ...` (обратите внимание, что вообще не имеет смысла оставлять эти неупорядоченные)<br>`"$*"` - это то же самое, что `"$1 $2 ..."` [1]<br>`"$@"` - это то же самое, что `"$1" "$2" ...`<br>1. Аргументы разделяются первым символом $ IFS, который не должен быть пробелом. |
| `$#` | Количество позиционных параметров, переданных скрипту или функции |
| `$!` | Идентификатор процесса последней команды (самый высокий для конвейеров) в последнем задании, помещенном в фоновом режиме (обратите внимание, что это не обязательно совпадает с идентификатором группы задач задания, когда включено управление заданием) |
| `$$` | ID процесса, который выполнил `bash` |
| `$?` | Выйти из последней команды |
| `$n` | Позиционные параметры, где n = 1, 2, 3, ..., 9 |
| `${n}` | Позиционные параметры (такие же, как указано выше), но n может быть> 9 |
| `$0` | В скриптах, путь, с которым был вызван скрипт; с `bash -c 'printf "%s\n" "$0"' name args` : `name` (первый аргумент после встроенного скрипта), в противном случае `argv[0]` который получил `bash` . |

| переменная | подробности |
|---|---|
| `$_` | Последнее поле последней команды |
| `$IFS` | Внутренний разделитель полей |
| `$PATH` | Переменная среды PATH, используемая для поиска исполняемых файлов |
| `$OLDPWD` | Предыдущий рабочий каталог |
| `$PWD` | Настоящий рабочий каталог |
| `$FUNCNAME` | Массив имен функций в стеке вызовов выполнения |
| `$BASH_SOURCE` | Массив, содержащий исходные пути для элементов в массиве `FUNCNAME` . Может использоваться для получения пути к скрипту. |
| `$BASH_ALIASES` | Ассоциативный массив, содержащий все указанные в настоящее время псевдонимы |
| `$BASH_REMATCH` | Массив совпадений из последнего совпадения регулярных выражений |
| `$BASH_VERSION` | Строка версии Bash |
| `$BASH_VERSINFO` | Массив из 6 элементов с информацией о версии Bash |
| `$BASH` | Абсолютный путь к текущей исполняемой оболочке Bash (эвристически определяемой `bash` на основе `argv[0]` и значением `$PATH` , может быть неправильным в случае с углами) |
| `$BASH_SUBSHELL` | Уровень подошвы Bash |
| `$UID` | Реальный (не эффективный, если отличается) Идентификатор пользователя процесса, запускающего `bash` |
| `$PS1` | Запрос основной командной строки; см. Использование переменных PS * |
| `$PS2` | Запрос дополнительной командной строки (используется для дополнительного ввода) |
| `$PS3` | Запрос третичной командной строки (используется в цикле выбора) |
| `$PS4` | Quaternary командной строки (используется для добавления информации с подробным выходом) |
| `$RANDOM` | Псевдослучайное целое число от 0 до 32767 |

| переменная | подробности |
| --- | --- |
| `$REPLY` | Переменная, используемая при `read` по умолчанию, если не указана переменная. Также используется для `select` возвращаемого пользователем значения |
| `$PIPESTATUS` | Переменная массива, которая содержит значения статуса выхода для каждой команды в последнем запущенном конвейере переднего плана. |

> Назначение переменной не должно иметь места до и после. `a=123` не `a = 123` . Последний (знак равенства, окруженный пробелами) изолированно означает выполнение команды `a` с аргументами `=` и `123` , хотя это также видно в операторе сравнения строк (который синтаксически является аргументом для `[` или `[[` или любого теста, который вы используете с помощью).

### $ BASHPID

Идентификатор процесса (pid) текущего экземпляра Bash. Это не то же самое, что и переменная `$$` , но часто дает тот же результат. Это новое в Bash 4 и не работает в Bash 3.

```
~> $ echo "\$\$ pid = $$  BASHPID = $BASHPID"
$$ pid = 9265  BASHPID = 9265
```

### $ BASH_ENV

Переменная среды, указывающая на файл запуска Bash, который считывается при вызове сценария.

### $ BASH_VERSINFO

Массив, содержащий полную информацию о версии, разбивается на элементы, гораздо удобнее, чем `$BASH_VERSION` если вы просто ищете основную версию:

```
~> $ for ((i=0; i<=5; i++)); do echo "BASH_VERSINFO[$i] = ${BASH_VERSINFO[$i]}"; done
BASH_VERSINFO[0] = 3
BASH_VERSINFO[1] = 2
BASH_VERSINFO[2] = 25
BASH_VERSINFO[3] = 1
BASH_VERSINFO[4] = release
BASH_VERSINFO[5] = x86_64-redhat-linux-gnu
```

### $ BASH_VERSION

Показывает версию bash, которая работает, это позволяет вам решить, можете ли вы использовать любые дополнительные функции:

```
~> $ echo $BASH_VERSION
```

```
4.1.2(1)-release
```

### $ EDITOR

Редактор по умолчанию, который будет запущен любыми сценариями или программами, обычно vi или emacs.

```
~> $ echo $EDITOR
vi
```

### $ имя_функции

Чтобы получить имя текущей функции - введите:

```
my_function()
{
    echo "This function is $FUNCNAME"    # This will output "This function is my_function"
}
```

Эта инструкция ничего не вернет, если вы внесете ее вне функции:

```
my_function

echo "This function is $FUNCNAME"    # This will output "This function is"
```

### $ HOME

Домашний каталог пользователя

```
~> $ echo $HOME
/home/user
```

### $ HOSTNAME

Имя хоста, назначенное системе во время запуска.

```
~> $ echo $HOSTNAME
mybox.mydomain.com
```

### $ HOSTTYPE

Эта переменная идентифицирует аппаратное обеспечение, может быть полезно определить, какие исполняемые файлы выполнить:

```
~> $ echo $HOSTTYPE
x86_64
```

### $ ГРУППЫ

Массив, содержащий числа групп, в которых находится пользователь:

```
#!/usr/bin/env bash
echo You are assigned to the following groups:
for group in ${GROUPS[@]}; do
   IFS=: read -r name dummy number members < <(getent group $group )
   printf "name: %-10s number: %-15s members: %s\n" "$name" "$number" "$members"
done
```

### $ IFS

Содержит строку внутреннего разделителя полей, которую bash использует для разделения строк при циклировании и т. Д. По умолчанию используются символы пробела: `\n` (новая строка), `\t` (вкладка) и пробел. Изменение этого на что-то другое позволяет разделить строки с использованием разных символов:

```
IFS=","
INPUTSTR="a,b,c,d"
for field in ${INPUTSTR}; do
    echo $field
done
```

Вышеуказанный результат:

```
a
b
c
d
```

**Заметки:**

- Это отвечает за явление, известное как расщепление слов .

### $ LINENO

Выводит номер строки в текущем скрипте. В основном полезно при отладке скриптов.

```
#!/bin/bash
# this is line 2
echo something  # this is line 3
echo $LINENO # Will output 4
```

### $ MACHTYPE

Подобно `$HOSTTYPE` выше, это также включает информацию об ОС, а также аппаратное обеспечение

```
~> $ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

### $ OLDPWD

**OLDPWD (OLDP** ечати **Рабо D** irectory) содержит каталог до последнего `cd` команды:

```
~> $ cd directory
directory> $ echo $OLDPWD
/home/user
```

### $ OSTYPE

Возвращает информацию о типе ОС, запущенной на машине, например.

```
~> $ echo $OSTYPE
linux-gnu
```

### $ PATH

Путь поиска для поиска двоичных файлов для команд. Общие примеры включают `/usr/bin` и `/usr/local/bin` .

Когда пользователь или сценарий пытается выполнить команду, поиск путей в `$PATH` выполняется, чтобы найти соответствующий файл с разрешением на выполнение.

Каталоги в `$PATH` разделяются символом `:`

```
~> $ echo "$PATH"
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin
```

Так, например, с учетом вышеуказанного `$PATH` , если вы `lss` в приглашении, оболочка будет искать `/usr/kerberos/bin/lss` , затем `/usr/local/bin/lss` , then `/bin/lss` , затем `/usr/bin/lss` , в этом порядке, прежде чем заключить, что такой команды нет.

### $ PPID

Идентификатор процесса (pid) сценария или родителя оболочки, что означает процесс, чем вызов текущего скрипта или оболочки.

```
~> $ echo $$
13016
~> $ echo $PPID
13015
```

### $ PWD

**PWD** ( **P** rint **W** orking **D-** directory) Текущий рабочий каталог, в котором вы находитесь сейчас:

```
~> $ echo $PWD
/home/user
~> $ cd directory
directory> $ echo $PWD
/home/user/directory
```

### $ SECONDS

Количество секунд, в течение которых скрипт запущен. Это может стать довольно большим, если отобразить в оболочке:

```
~> $ echo $SECONDS
98834
```

### $ SHELLOPTS

В начало запускается список readonly параметров bash для управления его поведением:

```
~> $ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

### $ SHLVL

Когда команда bash выполняется, открывается новая оболочка. Переменная среды $ SHLVL содержит количество уровней оболочки, на которых работает *текущая* оболочка.

В *новом* окне терминала выполнение следующей команды приведет к различным результатам на основе используемого дистрибутива Linux.

```
echo $SHLVL
```

Используя *Fedora 25* , выход «3». Это указывает на то, что при открытии новой оболочки исходная команда bash выполняет и выполняет задачу. Начальная команда bash выполняет дочерний процесс (другая команда bash), которая, в свою очередь, выполняет окончательную команду bash, чтобы открыть новую оболочку. Когда новая оболочка открывается, она запускается как дочерний процесс из двух других процессов оболочки, следовательно, выход «3».

В следующем примере (учитывая, что пользователь запускает Fedora 25), вывод $ SHLVL в новой оболочке будет установлен на «3». Поскольку каждая команда bash выполняется, $ SHLVL увеличивается на единицу.

```
~> $ echo $SHLVL
```

```
3
~> $ bash
~> $ echo $SHLVL
4
~> $ bash
~> $ echo $SHLVL
5
```

Можно видеть, что выполнение команды «bash» (или выполнение сценария bash) открывает новую оболочку. Для сравнения, поиск сценария запускает код в текущей оболочке.

**test1.sh**

```
#!/usr/bin/env bash
echo "Hello from test1.sh. My shell level is $SHLVL"
source "test2.sh"
```

**test2.sh**

```
#!/usr/bin/env bash
echo "Hello from test2.sh. My shell level is $SHLVL"
```

**run.sh**

```
#!/usr/bin/env bash
echo "Hello from run.sh. My shell level is $SHLVL"
./test1.sh
```

**Выполнение:**

```
chmod +x test1.sh && chmod +x run.sh
./run.sh
```

**Выход:**

```
Hello from run.sh. My shell level is 4
Hello from test1.sh. My shell level is 5
Hello from test2.sh. My shell level is 5
```

### $ UID

Только переменная для чтения, которая хранит идентификационный номер пользователя:

```
~> $ echo $UID
12345
```

### $ 1 $ 2 $ 3 и т. Д. ...

Позиционные параметры передаются скрипту из командной строки или из функции:

```
#!/bin/bash
# $n is the n'th positional parameter
echo "$1"
echo "$2"
echo "$3"
```

Вышеуказанный результат:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
firstarg
secondarg
thirdarg
```

Если число позиционных аргументов больше девяти, необходимо использовать фигурные скобки.

```
#  "set -- " sets positional parameters
set -- 1 2 3 4 5 6 7 8 nine ten eleven twelve
# the following line will output 10 not 1 as the value of $1 the digit 1
# will be concatenated with the following 0
echo $10   # outputs 1
echo ${10} # outputs ten
# to show this clearly:
set -- arg{1..12}
echo $10
echo ${10}
```

### $ #

Чтобы получить количество аргументов командной строки или позиционных параметров - введите:

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

При запуске с тремя аргументами приведенный выше пример приведет к выводу:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
3
```

### $ *

Вернет все позиционные параметры в одну строку.

**testscript.sh:**

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

Запустите скрипт с несколькими аргументами:

```
./testscript.sh firstarg secondarg thirdarg
```

Выход:

```
firstarg secondarg thirdarg
```

### $!

Идентификатор процесса (pid) последнего задания запускается в фоновом режиме:

```
~> $ ls &
testfile1 testfile2
[1]+  Done                    ls
~> $ echo $!
21715
```

### $ _

Выводит последнее поле из последней выполненной команды, полезно, чтобы передать что-то другому в другую команду:

```
 ~> $ ls *.sh;echo $_
testscript1.sh  testscript2.sh
testscript2.sh
```

---

Он дает путь к скрипту, если он используется перед любыми другими командами:

**test.sh:**

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

Выход:

```
~> $ ./test.sh # running test.sh
./test.sh
```

**Примечание.** Это не надежный способ получить путь к сценарию

### $?

Статус выхода последней выполненной функции или команды. Обычно 0 будет означать OK. Все остальное указывает на сбой:

```
~> $ ls *.blah;echo $?
```

```
ls: cannot access *.blah: No such file or directory
2
~> $ ls;echo $?
testfile1 testfile2
0
```

### $$

Идентификатор процесса (pid) текущего процесса:

```
~> $ echo $$
13246
```

### $ @

`"$@"` расширяет все аргументы командной строки в виде отдельных слов. Он отличается от `"$*"` , который расширяется до всех аргументов как одно слово.

`"$@"` особенно полезен для циклического использования аргументов и обработки аргументов с пробелами.

Рассмотрим, что мы находимся в скрипте, который мы вызываем с двумя аргументами, например:

```
$ ./script.sh "␣1␣2␣" "␣3␣␣4␣"
```

---

Переменные `$*` или `$@` будут расширяться до `$1␣$2` , которые, в свою очередь, расширятся на `1␣2␣3␣4` поэтому цикл ниже:

```
for var in $*; do # same for var in $@; do
    echo \<"$var"\>
done
```

будет печатать для обоих

```
<1>
<2>
<3>
<4>
```

---

В то время как `"$*"` будет расширен до `"$1␣$2"` который, в свою очередь, будет расширяться до `"␣1␣2␣␣␣3␣␣4␣"` и поэтому цикл:

```
for var in "$*"; do
    echo \<"$var"\>
done
```

будет только один раз вызывать `echo` и распечатать

```
<␣1␣2␣␣␣3␣␣4␣>
```

---

И, наконец, `"$@"` расширится до `"$1" "$2"` , который будет расширяться до `"␣1␣2␣" "␣3␣␣4␣"` и поэтому цикл

```
for var in "$@"; do
    echo \<"$var"\>
done
```

распечатает

```
<␣1␣2␣>
<␣3␣␣4␣>
```

тем самым сохраняя как внутреннее расстояние в аргументах, так и разделение аргументов. Заметим, что конструкция `for var in "$@"; do ...` настолько распространен и идиоматичен, что он является значением по умолчанию для цикла for и может быть сокращен до `for var; do ...`

## $ HISTSIZE

Максимальное количество запоминаемых команд:

```
~> $ echo $HISTSIZE
1000
```

## $ RANDOM

Каждый раз, когда этот параметр ссылается, генерируется случайное целое число от 0 до 32767. Присвоение значения этой переменной сгенерирует генератор случайных чисел ( источник ).

```
~> $ echo $RANDOM
27119
~> $ echo $RANDOM
1349
```

Прочитайте Внутренние переменные онлайн: https://riptutorial.com/ru/bash/topic/4797/внутренние-переменные

# глава 12: Выбрать ключевое слово

## Вступление

Выберите ключевое слово, которое можно использовать для ввода входного аргумента в формате меню.

## Examples

### Выбрать ключевое слово можно для ввода входного аргумента в формате меню

Предположим, вы хотите, чтобы `user select` ключевые слова из меню, мы можем создать скрипт, похожий на

```
#!/usr/bin/env bash

select os in "linux" "windows" "mac"
do
    echo "${os}"
    break
done
```

Объяснение: Здесь ключевое слово `select` используется для циклического перебора списка элементов, которые будут представлены в командной строке для пользователя. Обратите внимание на ключевое слово `break` для выхода из цикла после выбора пользователем. В противном случае цикл будет бесконечным!

Результаты. После запуска этого скрипта будет отображено меню этих элементов, и пользователю будет предложено выбрать его. После выбора значение будет отображаться, возвращаясь к командной строке.

```
>bash select_menu.sh
1) linux
2) windows
3) mac
#? 3
mac
>
```

Прочитайте Выбрать ключевое слово онлайн: https://riptutorial.com/ru/bash/topic/10104/выбрать-ключевое-слово

# глава 13: Вывод цветного скрипта (кросс-платформенный)

## замечания

`tput` запрашивает базу данных terminfo для информации, зависящей от терминала.

Материал из tput в Википедии :

> При вычислении `tput` - стандартная команда операционной системы Unix, которая использует возможности терминала.
>
> В зависимости от системы, `tput` использует базу данных terminfo или termcap, а также просматривает среду для типа терминала.

из Bash Prompt HOWTO: Глава 6. Последовательности выхода ANSI: цвета и курсор Движение :

- **tput setab [1-7]**
  - Установите цвет фона с помощью ANSI-escape
- **tput setb [1-7]**
  - Установите цвет фона
- **tput setaf [1-7]**
  - Установите цвет переднего плана с помощью ANSI escape
- **tput setf [1-7]**
  - Установите цвет переднего плана
- **tput bold**
  - Установите жирный режим
- **tput sgr0**
  - Отключите все атрибуты (не работает так, как ожидалось)

## Examples

### color-output.sh

В первом разделе сценария bash можно определить некоторые переменные, которые служат помощниками для цветного или иного форматирования вывода терминала во время запуска скрипта.

Различные платформы используют разные последовательности символов для выражения цвета. Тем не менее, есть утилита `tput` которая работает на всех системах * nix и возвращает терминальные строки окраски конкретной платформы через согласованный межплатформенный API.

Например, чтобы сохранить последовательность символов, которая превращает текст терминала красным или зеленым:

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

Или, чтобы сохранить последовательность символов, которая сбрасывает текст по умолчанию:

```
reset=$(tput sgr0)
```

Затем, если скрипт BASH должен отображать разные цветные выходы, это может быть достигнуто с помощью:

```
echo "${green}Success!${reset}"
echo "${red}Failure.${reset}"
```

Прочитайте Вывод цветного скрипта (кросс-платформенный) онлайн: https://riptutorial.com/ru/bash/topic/6670/вывод-цветного-скрипта--кросс-платформенный-

# глава 14: Вырезать команду

## Вступление

В Bash команда `cut` полезна для деления файла на несколько меньших частей.

## Синтаксис

- cut [option] file

## параметры

| вариант | Описание |
|---|---|
| `-b LIST` , `--bytes=LIST` | Распечатайте байты, перечисленные в параметре LIST |
| `-c LIST` , `--characters=LIST` | Печатать символы в позициях, указанных в параметре LIST |
| `-f LIST` , `--fields=LIST` | Печать полей или столбцов |
| `-d DELIMITER` | Используется для разделения столбцов или полей |

## Examples

### Показать первый столбец файла

Предположим, у вас есть файл, который выглядит так:

```
John Smith 31
Robert Jones 27
...
```

Этот файл имеет 3 столбца, разделенных пробелами. Чтобы выбрать только первый столбец, сделайте следующее.

```
cut -d ' ' -f1 filename
```

Здесь `-d` флаг, указывает разделитель или разделяет записи. Флаг `-f` указывает номер поля или столбца. На этом отобразится следующий вывод

```
John
Robert
...
```

## Показать столбцы от x до y файла

Иногда полезно отображать ряд столбцов в файле. Предположим, у вас есть этот файл

```
Apple California 2017 1.00 47
Mango Oregon 2015 2.30 33
```

Чтобы выбрать первые 3 столбца, сделайте

```
cut -d ' ' -f1-3 filename
```

На этом отобразится следующий вывод

```
Apple California 2017
Mango Oregon 2015
```

Прочитайте Вырезать команду онлайн: https://riptutorial.com/ru/bash/topic/9138/вырезать-команду

# глава 15: глобальные и локальные переменные

## Вступление

По умолчанию каждая переменная в bash является **глобальной** для каждой функции, скрипта и даже внешней оболочки, если вы объявляете свои переменные внутри скрипта.

Если вы хотите, чтобы ваша переменная была локальной для функции, вы можете использовать `local` для этой переменной новую переменную, независимую от глобальной области и значение которой будет доступно только внутри этой функции.

## Examples

### Глобальные переменные

```
var="hello"

function foo(){
    echo $var
}

foo
```

Очевидно, будет выводиться «привет», но это тоже работает наоборот:

```
function foo()  {
    var="hello"
}

foo
echo $var
```

Будет также выводить `"hello"`

### Местные переменные

```
function foo() {
    local var
    var="hello"
}

foo
echo $var
```

Не будет выводить ничего, поскольку var является переменной, локальной для функции

foo, и ее значение не видно из-за пределов.

### Смешивание двух

```
var="hello"

function foo(){
    local var="sup?"
    echo "inside function, var=$var"
}

foo
echo "outside function, var=$var"
```

Выйдет

```
inside function, var=sup?
outside function, var=hello
```

Прочитайте глобальные и локальные переменные онлайн:
https://riptutorial.com/ru/bash/topic/9256/глобальные-и-локальные-переменные

# глава 16: Горячие клавиши

## замечания

`bind -P` показать все настроенные ярлыки.

## Examples

### Вспомогательные ссылки

| кратчайший путь | Описание |
|---|---|
| `Ctrl` + `r` | искать историю назад |
| `Ctrl` + `p` | предыдущая команда в истории |
| `Ctrl` + `n` | следующая команда в истории |
| `Ctrl` + `g` | выйти из режима поиска истории |
| `Alt` + `.` | используйте последнее слово предыдущей команды |
| | повторите, чтобы получить последнее слово предыдущей команды + 1 |
| `Alt` + `n` `Alt` + `.` | используйте n-ое слово предыдущей команды |
| `!!` + `Возвращение` | снова выполнить последнюю команду (полезно, когда вы забыли sudo: `sudo !!` ) |

### Редактирование ярлыков

| кратчайший путь | Описание |
|---|---|
| `Ctrl` + `a` | перейти к началу строки |
| `Ctrl` + `e` | переместиться в конец строки |
| `Ctrl` + `k` | Убейте текст от текущей позиции курсора до конца строки. |
| `Ctrl` + `u` | Убейте текст от текущей позиции курсора до начала строки |

| кратчайший путь | Описание |
|---|---|
| `Ctrl` + `w` | Убейте слово за текущей позицией курсора |
| `Alt` + `b` | переместить назад одно слово |
| `Alt` + `f` | переместить одно слово |
| `Ctrl` + `Alt` + `e` | линия расширения оболочки |
| `Ctrl` + `y` | Загрузите последний удаленный текст обратно в буфер под курсором. |
| `Alt` + `y` | Повернитесь через убитый текст. Вы можете сделать это, только если предыдущая команда - `Ctrl` + `y` или `Alt` + `y` . |

Текст Killing удаляет текст, но сохраняет его, чтобы пользователь мог повторно вставить его, потянув. Подобно вырезанию и вставке, за исключением того, что текст помещается на кольцо уничтожения, которое позволяет хранить более одного набора текста, который нужно вернуть в командную строку.

Вы можете узнать больше в руководстве emacs .

### Управление заданиями

| кратчайший путь | Описание |
|---|---|
| `Ctrl` + `c` | Остановить текущую работу |
| `Ctrl` + `z` | Приостановить текущее задание (отправить сигнал SIGTSTP) |

### макрос

| кратчайший путь | Описание |
|---|---|
| `Ctrl` + `x` , `(` | начать запись макроса |
| `Ctrl` + `x` , `)` | прекратить запись макроса |
| `Ctrl` + `x` , `e` | выполнить последний записанный макрос |

### Custome Key Bindings

С помощью команды `bind` можно определить пользовательские привязки клавиш.

Следующий пример привязывает `Alt` + `w` к `>/dev/null 2>&1` :

```
bind '"\ew":"\" >/dev/null 2>&1\""
```

Если вы хотите выполнить строку, немедленно добавьте в нее `\Cm` ( `Enter` ):

```
bind '"\ew":"\" >/dev/null 2>&1\C-m\""
```

Прочитайте Горячие клавиши онлайн: https://riptutorial.com/ru/bash/topic/3949/горячие-клавиши

# глава 17: Замена процесса

## замечания

Подстановка процесса - это форма перенаправления, когда вход или выход процесса (некоторая последовательность команд) отображаются как временный файл.

## Examples

### Сравните два файла из Интернета

Ниже сравниваются два файла с `diff` использующие замену процессов вместо создания временных файлов.

```
diff <(curl http://www.example.com/page1) <(curl http://www.example.com/page2)
```

### Подайте цикл while с выходом команды

Это питает `while` цикл с выходом `grep` команды:

```
while IFS=":" read -r user _
do
    # "$user" holds the username in /etc/passwd
done < <(grep "hello" /etc/passwd)
```

### С помощью команды paste

```
# Process substitution with paste command is common
# To compare the contents of two directories
paste <( ls /path/to/directory1 ) <( ls /path/to/directory1 )
```

### Объединение файлов

Хорошо известно, что вы не можете использовать тот же файл для ввода и вывода в той же команде. Например,

```
$ cat header.txt body.txt >body.txt
```

не делает то, что вы хотите. К моменту, когда `cat` считывает `body.txt` , он уже усечен перенаправлением и пуст. Конечным результатом будет то, что `body.txt` будет содержать только содержимое `header.txt` .

Можно подумать, чтобы избежать этого с заменой процесса, то есть, что команда

```
$ cat header.txt <(cat body.txt) > body.txt
```

заставит исходное содержимое `body.txt` быть каким-то образом сохранено в некотором буфере где-то до того, как файл будет усечен перенаправлением. Это не работает. `cat` в круглых скобках начинает читать файл только после того, как все дескрипторы файлов настроены, как и внешний. В этом случае нет смысла пытаться использовать замену процесса.

Единственный способ добавить файл к другому файлу - создать промежуточное:

```
$ cat header.txt body.txt >body.txt.new
$ mv body.txt.new body.txt
```

это то, что `sed` или `perl` или подобные программы выполняются под ковром при вызове с *параметром edit-in-place* (обычно `-i` ).

### Поток файла через несколько программ одновременно

Это подсчитывает количество строк в большом файле с `wc -l` , одновременно сжимая его с помощью `gzip` . Оба запускаются одновременно.

```
tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

Обычно `tee` записывает свой ввод в один или несколько файлов (и stdout). Мы можем писать команды вместо файлов с `tee >(command)` .

Здесь команда `wc -l >&2` подсчитывает строки, считанные из `tee` (которые, в свою очередь, `bigfile` из `bigfile` ). (Счетчик строк отправляется на stderr ( `>&2` ), чтобы избежать смешивания с входом в `gzip` .) Одновременное подавление `tee` подается в `gzip` .

### Чтобы избежать использования суб-оболочки

Одним из основных аспектов замещения процесса является то, что он позволяет нам избегать использования под-оболочки при выполнении команд трубопровода из оболочки.

Это можно продемонстрировать с помощью простого примера ниже. У меня есть следующие файлы в моей текущей папке:

```
$ find . -maxdepth 1 -type f -print
foo bar zoo foobar foozoo barzoo
```

Если я подключаюсь к циклу `while` / `read` который увеличивает счетчик следующим образом:

```
count=0
find . -maxdepth 1 -type f -print | while IFS= read -r _; do
    ((count++))
```

```
done
```

`$count` now *не* содержит `6` , потому что он был изменен в контексте подкласса. Любая из приведенных ниже команд запускается в контексте под-оболочки, а область переменных, используемых внутри, теряется после завершения суб-оболочки.

```
command &
command | command
( command )
```

Замена процесса решит проблему, избегая использования трубы `|` оператора в

```
count=0
while IFS= read -r _; do
    ((count++))
done < <(find . -maxdepth 1 -type f -print)
```

Это сохранит `count` переменной `count` как никакие под-оболочки не будут вызваны.

Прочитайте Замена процесса онлайн: https://riptutorial.com/ru/bash/topic/2647/замена-процесса

# глава 18: Замены истории Баша

## Examples

### Использование! $

Вы можете использовать `!$` Для уменьшения повторения при использовании командной строки:

```
$ echo ping
ping
$ echo !$
ping
```

Вы также можете опираться на повторение

```
$ echo !$ pong
ping pong
$ echo !$, a great game
pong, a great game
```

Обратите внимание, что в последнем примере мы не получили `ping pong, a great game` потому что последний аргумент, переданный предыдущей команде, был `pong` , мы можем избежать такой проблемы, добавив цитаты. Продолжая пример, наш последний аргумент был `game` :

```
$ echo "it is !$ time"
it is game time
$ echo "hooray, !$!"
hooray, it is game time!
```

### Краткий справочник

### Взаимодействие с историей

```
# List all previous commands
history

# Clear the history, useful if you entered a password by accident
history -c
```

### Обозначения событий

```
# Expands to line n of bash history
!n
```

```
# Expands to last command
!!

# Expands to last command starting with "text"
!text

# Expands to last command containing "text"
!?text

# Expands to command n lines ago
!-n

# Expands to last command with first occurrence of "foo" replaced by "bar"
^foo^bar^

# Expands to the current command
!#
```

## Обозначения слов

Они разделяются `:` от обозначенного события, на который они ссылаются. Двоеточие можно опустить, если указатель слова не начинается с числа `!^` То же самое, что и `!:^` .

```
# Expands to the first argument of the most recent command
!^

# Expands to the last argument of the most recent command (short for !!:$)
!$

# Expands to the third argument of the most recent command
!:3

# Expands to arguments x through y (inclusive) of the last command
# x and y can be numbers or the anchor characters ^ $
!:x-y

# Expands to all words of the last command except the 0th
# Equivalent to :^-$
!*
```

## Модификаторы

Они изменяют предыдущее событие или обозначение слова.

```
# Replacement in the expansion using sed syntax
# Allows flags before the s and alternate separators
:s/foo/bar/ #substitutes bar for first occurrence of foo
:gs|foo|bar| #substitutes bar for all foo

# Remove leading path from last argument ("tail")
:t

# Remove trailing path from last argument ("head")
:h

# Remove file extension from last argument
```

```
:r
```

Если переменная Bash `HISTCONTROL` содержит либо `ignorespace` либо `ignoreboth` (или, в качестве альтернативы, `HISTIGNORE` содержит шаблон `[ ]*` ), вы можете запретить хранить ваши команды в истории Bash, добавив их в пробел:

```
# This command won't be saved in the history
 foo

# This command will be saved
bar
```

## Поиск в истории команд по шаблону

Нажмите кнопку `управления r` и введите шаблон.

Например, если вы недавно исполнили `man 5 crontab` , вы можете быстро найти его, *начав вводить* «crontab». Приглашение изменится следующим образом:

```
(reverse-i-search)`cr': man 5 crontab
```

`` `cr' `` есть строка, которую я набрал до сих пор. Это инкрементный поиск, так как вы продолжаете вводить текст, результат поиска обновляется в соответствии с самой последней командой, содержащей шаблон.

Нажмите клавиши со стрелками влево или вправо, чтобы отредактировать согласованную команду перед запуском, или клавишу `ввода` для запуска команды.

По умолчанию поиск находит самую последнюю выполненную команду, соответствующую шаблону. Чтобы снова вернуться в историю, нажмите кнопку `управления r` снова. Вы можете нажимать его несколько раз, пока не найдете нужную команду.

## Переключитесь на новый каталог с! #: N

```
$ mkdir backup_download_directory && cd !#:1
mkdir backup_download_directory && cd backup_download_directory
```

Это заменит N-й аргумент текущей команды. В примере `!#:1` заменяется первым аргументом, то есть backup_download_directory.

## Повторите предыдущую команду с заменой

```
$ mplayer Lecture_video_part1.mkv
$ ^1^2^
mplayer Lecture_video_part2.mkv
```

Эта команда заменит `1` на `2` в ранее выполненной команде. Он заменит только первое вхождение строки и эквивалентен `!!:s/1/2/` .

Если вы хотите заменить *все* вхождения, вам нужно использовать `!!:gs/1/2/` или `!!:as/1/2/` .

### Повторить предыдущую команду с помощью sudo

```
$ apt-get install r-base
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
$ sudo !!
sudo apt-get install r-base
[sudo] password for <user>:
```

Прочитайте Замены истории Баша онлайн: https://riptutorial.com/ru/bash/topic/1519/замены-истории-баша

# глава 19: Заявление о ситуации

## Examples

### Простой регистр

В простейшей форме, поддерживаемой всеми версиями bash, case statement выполняет случай, соответствующий шаблону. `;;` оператор прерывается после первого матча, если таковой имеется.

```
#!/bin/bash

var=1
case $var in
1)
  echo "Antartica"
 ;;
2)
  echo "Brazil"
 ;;
3)
  echo "Cat"
 ;;
esac
```

Выходы:

```
Antartica
```

### Заявление о случаях с провалом

4,0

Начиная с bash 4.0 был введен новый оператор `;&` который обеспечивает провал механизма.

#! / Bin / Баш

```
var=1
case $var in
1)
  echo "Antartica"
  ;&
2)
  echo "Brazil"
  ;&
3)
  echo "Cat"
  ;&
esac
```

Выходы:

```
Antartica
Brazil
Cat
```

## Проваливайте, только если последующие шаблоны

4,0

Начиная с Bash 4.0, был введен еще один оператор `;;&` который также обеспечивает провал *только в том* случае, если совпадают шаблоны в последующих операторных утверждениях (если они есть).

```
#!/bin/bash

var=abc
case $var in
a*)
  echo "Antartica"
  ;;&
xyz)
  echo "Brazil"
  ;;&
*b*)
  echo "Cat"
  ;;&
esac
```

Выходы:

```
Antartica
Cat
```

В приведенном ниже примере `abc` соответствует первому и третьему случаям, но не второму случаю. Итак, второй случай не выполняется.

Прочитайте Заявление о ситуации онлайн: https://riptutorial.com/ru/bash/topic/5237/заявление-о-ситуации

# глава 20: Здесь документы и здесь строки

## Examples

### Отступы здесь документы

Вы можете отпечатать текст внутри документов с помощью вкладок, вам нужно использовать оператор `<<-` redirection вместо `<<` :

```
$ cat <<- EOF
    This is some content indented with tabs `\t`.
    You cannot indent with spaces you __have__ to use tabs.
    Bash will remove empty space before these lines.
    __Note__: Be sure to replace spaces with tabs when copying this example.
EOF

This is some content indented with tabs _\t_.
You cannot indent with spaces you __have__ to use tabs.
Bash will remove empty space before these lines.
__Note__: Be sure to replace spaces with tabs when copying this example.
```

Одно практическое применение этого (как упоминалось в `man bash` ) - в сценариях оболочки, например:

```
if cond; then
    cat <<- EOF
    hello
    there
    EOF
fi
```

Обычно для отрисовки строк в кодовых блоках, как и в этом операторе `if` , требуется улучшенная читаемость. Без синтаксиса оператора `<<-` мы будем вынуждены написать приведенный выше код следующим образом:

```
if cond; then
    cat << EOF
hello
there
EOF
fi
```

Это очень неприятно для чтения, и это становится намного хуже в более сложном реалистическом сценарии.

### Здесь строки

2.05b

Вы можете подать команду, используя следующие строки:

```
$ awk '{print $2}' <<< "hello world - how are you?"
world

$ awk '{print $1}' <<< "hello how are you
> she is fine"
hello
she
```

Вы также можете кормить `while` цикл с здесь строками:

```
$ while IFS=" " read -r word1 word2 rest
> do
> echo "$word1"
> done <<< "hello how are you - i am fine"
hello
```

## Лимитные строки

Heredoc использует *limitstring,* чтобы определить, когда прекратить потребление ввода. Конечная конечная строка **должна**

- Будьте в начале линии.
- Будьте единственным текстом в строке. **Примечание.** Если вы используете `<<-` limitstring может быть префикс с вкладками `\t`

Правильный:

```
cat <<limitstring
line 1
line 2
limitstring
```

Это приведет к выводу:

```
line 1
line 2
```

Неправильное использование:

```
cat <<limitstring
line 1
line 2
 limitstring
```

Поскольку `limitstring` в последней строке не находится точно в начале строки, оболочка будет продолжать ждать ввода, пока не увидит строку, которая начинается с `limitstring` и не содержит ничего другого. Только тогда он перестанет ждать ввода и продолжит передачу этого документа команде `cat` .

Обратите внимание, что при префиксе initial limitstring с дефисом любые табуляции в начале строки удаляются перед разбором, поэтому данные и предельная строка могут быть отступом с вкладками (для удобства чтения в сценариях оболочки).

```
cat <<-limitstring
        line 1    has a tab each before the words line and has
            line 2 has two leading tabs
        limitstring
```

будет производить

```
line 1    has a tab each before the words line and has
line 2 has two leading tabs
```

с удалением ведущих вкладок (но не внутренних вкладок).

## Создать файл

Классическое использование здесь документов - это создать файл, набрав его содержимое:

```
cat > fruits.txt << EOF
apple
orange
lemon
EOF
```

Здесь-документ представляет собой линии между `<< EOF` и `EOF` .

Этот документ становится входом команды `cat` . Команда `cat` просто выводит свой вход, и с помощью оператора перенаправления вывода `>` мы перенаправляем файл `fruits.txt` .

В результате файл `fruits.txt` будет содержать строки:

```
apple
orange
lemon
```

Применяются обычные правила перенаправления вывода: если `fruits.txt` ранее не существовал, он будет создан. Если он существовал раньше, он будет усечен.

## Выполнить команду здесь

```
ssh -p 21 example@example.com <<EOF
  echo 'printing pwd'
  echo "\$(pwd)"
  ls -a
  find '*.txt'
EOF
```

`$` экранируется, потому что мы не хотим, чтобы он расширялся текущей оболочкой, то есть `$(pwd)` должен быть выполнен на удаленной оболочке.

По-другому:

```
ssh -p 21 example@example.com <<'EOF'
  echo 'printing pwd'
  echo "$(pwd)"
  ls -a
  find '*.txt'
EOF
```

*Примечание* : Закрытие EOF **должно** быть в начале строки (перед пробелом не было пробелов). Если требуется отступ, вкладки могут использоваться, если вы запустите свой heredoc с помощью `<<-` . Дополнительную информацию см. В документах « Отступы» и « Лимитные строки» .

## Запустите несколько команд с помощью sudo

```
sudo -s <<EOF
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "\$a"
EOF
```

- `$a` необходимо экранировать, чтобы предотвратить его расширение текущей оболочкой

Или же

```
sudo -s <<'EOF'
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "$a"
EOF
```

Прочитайте Здесь документы и здесь строки онлайн: https://riptutorial.com/ru/bash/topic/655/здесь-документы-и-здесь-строки

# глава 21: Избегание даты с помощью printf

## Вступление

В Bash 4.2 было введено встроенное преобразование времени для `printf` : спецификация формата `%(datefmt)T` делает `printf` выводит строку даты, соответствующую строке формата `datefmt` как это понимается `strftime` .

## Синтаксис

- printf '% (dateFmt) T' # dateFmt может быть любой строкой формата, которая strftime распознает
- printf '% (dateFmt) T' -1 # -1 представляет текущее время (по умолчанию для отсутствия аргумента)
- printf '% (dateFmt) T' -2 # -2 - время, в которое была вызвана оболочка

## замечания

Использование `printf -v foo '%(...)T'` идентично `foo=$(date +'...')` и сохраняет fork для вызова внешней `date` программы.

## Examples

### Получить текущую дату

```
$ printf '%(%F)T\n'
2016-08-17
```

### Установите переменную на текущее время

```
$ printf -v now '%(%T)T'
$ echo "$now"
12:42:47
```

Прочитайте Избегание даты с помощью printf онлайн:
https://riptutorial.com/ru/bash/topic/5522/избегание-даты-с-помощью-printf

# глава 22: Изменить оболочку

## Синтаксис

- эхо $ 0
- ps -p $$
- echo $ SHELL
- экспорт SHELL = / bin / bash
- exec / bin / bash
- cat / etc / shells

## Examples

### Найти текущую оболочку

Существует несколько способов определить текущую оболочку

```
echo $0
ps -p $$
echo $SHELL
```

### Изменение оболочки

Чтобы изменить текущий баш, выполните эти команды

```
export SHELL=/bin/bash
exec /bin/bash
```

изменить bash, который открывается при запуске, отредактировать `.profile` и добавить эти строки

### Список доступных оболочек

Чтобы просмотреть список доступных оболочек входа:

```
cat /etc/shells
```

Пример:

```
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

Прочитайте Изменить оболочку онлайн: https://riptutorial.com/ru/bash/topic/3951/изменить-оболочку

# глава 23: Использование «ловушки» для реагирования на сигналы и системные события

## Синтаксис

- trap action sigspec ... # Запустить «действие» в списке сигналов
- trap sigspec ... # Опуская действие сбрасывает ловушки для сигналов

## параметры

| параметр | Имея в виду |
|---|---|
| -п | Список установленных ловушек |
| -l | Имена списков и соответствующие номера |

## замечания

Утилита `trap` - специальная встроенная оболочка. Он определен в POSIX , но bash добавляет некоторые полезные расширения.

Примеры, совместимые с POSIX, начинаются с `#!/bin/sh` , а примеры, начинающиеся с `#!/bin/bash` используют расширение bash.

Сигналами могут быть либо номер сигнала, имя сигнала (без префикса SIG), либо специальное ключевое слово `EXIT` .

Гарантируются POSIX:

| Число | название | Заметки |
|---|---|---|
| 0 | ВЫХОД | Всегда запускать на выходе оболочки, независимо от кода выхода |
| 1 | SIGHUP | |
| 2 | SIGINT | Это то, что посылает `^C` |
| 3 | SIGQUIT | |
| 6 | SIGABRT | |

| Число | название | Заметки |
|---|---|---|
| 9 | SIGKILL | |
| 14 | SIGALRM | |
| 15 | SIGTERM | Это то, что `kill` по умолчанию |

# Examples

## Захват SIGINT или Ctl + C

Ловушка сбрасывается для подоболочек, поэтому `sleep` будет по-прежнему действовать на сигнал `SIGINT` отправленный `^C` (обычно путем выхода из игры), но родительский процесс (то есть сценарий оболочки) не будет.

```
#!/bin/sh

# Run a command on signal 2 (SIGINT, which is what ^C sends)
sigint() {
    echo "Killed subshell!"
}
trap sigint INT

# Or use the no-op command for no output
#trap : INT

# This will be killed on the first ^C
echo "Sleeping..."
sleep 500

echo "Sleeping..."
sleep 500
```

И вариант, который по-прежнему позволяет вам выйти из основной программы, дважды нажав `^C` дважды:

```
last=0
allow_quit() {
    [ $(date +%s) -lt $(( $last + 1 )) ] && exit
    echo "Press ^C twice in a row to quit"
    last=$(date +%s)
}
trap allow_quit INT
```

## Введение: очистка временных файлов

Вы можете использовать команду `trap` для «ловушки» сигналов; это эквивалент оболочки `signal()` или `sigaction()` в C и большинстве других языков программирования для улавливания сигналов.

Одним из наиболее распространенных способов использования `trap` является очистка временных файлов как от ожидаемого, так и от неожиданного выхода.

К сожалению, недостаточно скриптов оболочки делают следующее :-(

```
#!/bin/sh

# Make a cleanup function
cleanup() {
  rm --force -- "${tmp}"
}

# Trap the special "EXIT" group, which is always run when the shell exits.
trap cleanup EXIT

# Create a temporary file
tmp="$(mktemp -p /tmp tmpfileXXXXXXX)"

echo "Hello, world!" >> "${tmp}"

# No rm -f "$tmp" needed. The advantage of using EXIT is that it still works
# even if there was an error or if you used exit.
```

## Накопите список ловушек для работы при выходе.

Вы когда-нибудь забыли добавить `trap` для очистки временного файла или выполнить другую работу при выходе?

Вы когда-нибудь устанавливали одну ловушку, которая отменяла другую?

Этот код упрощает добавление вещей, которые нужно выполнять при выходе из одного элемента за раз, вместо того, чтобы иметь один большой оператор `trap` где-то в вашем коде, что может быть легко забыть.

```
# on_exit and add_on_exit
# Usage:
#   add_on_exit rm -f /tmp/foo
#   add_on_exit echo "I am exiting"
#   tempfile=$(mktemp)
#   add_on_exit rm -f "$tempfile"
# Based on http://www.linuxjournal.com/content/use-bash-trap-statement-cleanup-temporary-files
function on_exit()
{
    for i in "${on_exit_items[@]}"
    do
        eval $i
    done
}
function add_on_exit()
{
    local n=${#on_exit_items[*]}
    on_exit_items[$n]="$*"
    if [[ $n -eq 0 ]]; then
        trap on_exit EXIT
    fi
```

```
}
```

### Убийство детских процессов на выходе

Выражения Trap не должны быть отдельными функциями или программами, они также могут быть более сложными выражениями.

Объединив `jobs -p` и `kill` , мы можем убить все порожденные дочерние процессы оболочки при выходе:

```
trap 'jobs -p | xargs kill' EXIT
```

### реагировать на изменение размера окна терминала

Существует сигнал WINCH (WINdowCHange), который запускается при изменении размера окна терминала.

```
declare -x rows cols

update_size(){
  rows=$(tput lines) # get actual lines of term
  cols=$(tput cols)  # get actual columns of term
  echo DEBUG terminal window has no $rows lines and is $cols characters wide
}

trap update_size WINCH
```

Прочитайте Использование «ловушки» для реагирования на сигналы и системные события онлайн: https://riptutorial.com/ru/bash/topic/363/использование--ловушки--для-реагирования-на-сигналы-и-системные-события

# глава 24: Использование кота

## Синтаксис

- cat [OPTIONS] ... [FILE] ...

## параметры

| вариант | подробности |
|---|---|
| -n | Номера строк печати |
| -v | Показывать непечатаемые символы с использованием ^ и M-нотации, кроме LFD и TAB |
| -T | Показывать символы TAB как ^ I |
| -E | Показывать символы перевода строки (LF) в виде $ |
| -e | То же, что и -vE |
| -b | Число непустых выходных строк, переопределяет -n |
| -A | эквивалентно -vET |
| -s | подавлять повторяющиеся пустые выходные строки, s означает сжатие |

## замечания

`cat` может читать как файлы, так и стандартные входы и объединяет их со стандартным выходом

## Examples

### Печать содержимого файла

```
cat file.txt
```

будет печатать содержимое файла.

Если файл содержит символы, отличные от ASCII, вы можете символически отображать эти символы с `cat -v` . Это может быть весьма полезно для ситуаций, когда контрольные

символы в противном случае были бы невидимыми.

```
cat -v unicode.txt
```

Очень часто для интерактивного использования вам лучше использовать интерактивный пейджер, например, `less` или `more` . ( `less` чем `more` и рекомендуется использовать `less` часто, чем `more` ).

```
less file.txt
```

Чтобы передать содержимое файла в качестве ввода команды. Подход, обычно рассматриваемый как лучший ( UUOC ), заключается в использовании перенаправления.

```
tr A-Z a-z <file.txt   # as an alternative to cat file.txt | tr A-Z a-z
```

В случае, если содержимое нужно перечислить назад с его конца, можно использовать команду `tac` :

```
tac file.txt
```

Если вы хотите распечатать содержимое с номерами строк, используйте `-n` с `cat` :

```
cat -n file.txt
```

Чтобы отобразить содержимое файла в абсолютно однозначной побайтовой форме, шестнадцатеричный дамп является стандартным решением. Это полезно для очень коротких фрагментов файла, например, когда вы не знаете точное кодирование. Стандартная утилита hex dump - `od -cH` , хотя представление немного громоздко; общие замены включают `xxd` и `hexdump` .

```
$ printf 'Hëllö wörld' | xxd
0000000: 48c3 ab6c 6cc3 b620 77c3 b672 6c64       H..ll.. w..rld
```

## Отображать номера строк с выводом

Используйте флаг `--number` для печати номеров строк перед каждой строкой. Альтернативно, `-n` делает то же самое.

```
$ cat --number file

 1  line 1
 2  line 2
 3
 4  line 4
 5  line 5
```

Чтобы пропустить пустые строки при подсчете строк, используйте `--number-nonblank` или просто `-b` .

```
$ cat -b file

 1  line 1
 2  line 2

 3  line 4
 4  line 5
```

### Чтение со стандартного ввода

```
cat < file.txt
```

Результат такой же, как `cat file.txt` , но он считывает содержимое файла со стандартного ввода, а не непосредственно из файла.

```
printf "first line\nSecond line\n" | cat -n
```

Команда эха перед `|` выводит две строки. Команда cat действует на выходе для добавления номеров строк.

### Объединить файлы

Это основная цель `cat` .

```
cat file1 file2 file3 > file_all
```

`cat` также можно использовать аналогично файлам конкатенации в составе конвейера, например

```
cat file1 file2 file3 | grep foo
```

### Запись в файл

```
cat >file
```

Это позволит вам написать текст на терминале, который будет сохранен в файле с именем *file* .

```
cat >>file
```

будет делать то же самое, за исключением того, что он добавит текст в конец файла.

NB: `Ctrl + D` для завершения написания текста на терминале (Linux)

Этот документ можно использовать для встраивания содержимого файла в командную строку или скрипт:

```
cat <<END >file
Hello, World.
END
```

Токен после символа `<<` перенаправления - это произвольная строка, которая должна встречаться одна в строке (без начального или конечного пробела), чтобы указать конец документа здесь. Вы можете добавить цитирование, чтобы предотвратить выполнение командной строки и интерполяцию переменных:

```
cat <<'fnord'
Nothing in `here` will be $changed
fnord
```

(Без кавычек `here` будет выполняться как команда, а `$changed` будет заменено значением `changed` переменной - или ничего, если оно не определено).

## Показывать непечатаемые символы

Это полезно, чтобы увидеть, есть ли какие-либо непечатаемые символы или символы, отличные от ASCII.

например, если вы скопировали код из Интернета, у вас могут быть такие цитаты, как ″ вместо стандартного " .

```
$ cat -v file.txt
$ cat -vE file.txt # Useful in detecting trailing spaces.
```

например

```
$ echo '″     ' | cat -vE # echo | will be replaced by actual file.
M-bM-^@M-^]     $
```

Вы также можете использовать `cat -A` (A для всех), который эквивалентен `cat -vET` . Он отобразит символы TAB (отображается как `^I` ), непечатаемые символы и конец каждой строки:

```
$ echo '″ `' | cat -A
M-bM-^@M-^]^I`$
```

## Конкатенация файлов gzipped

Файлы, сжатые `gzip` могут быть напрямую объединены в файлы большего размера.

```
cat file1.gz file2.gz file3.gz > combined.gz
```

Это свойство `gzip` , которое менее эффективно, чем конкатенация входных файлов и gzipping результат:

```
cat file1 file2 file3 | gzip > combined.gz
```

Полная демонстрация:

```
echo 'Hello world!' > hello.txt
echo 'Howdy world!' > howdy.txt
gzip hello.txt
gzip howdy.txt

cat hello.txt.gz howdy.txt.gz > greetings.txt.gz

gunzip greetings.txt.gz

cat greetings.txt
```

В результате

```
Hello world!
Howdy world!
```

Обратите внимание, что `greetings.txt.gz` является ***одним файлом*** и распаковывается как ***один файл*** `greeting.txt` . Сравните это с `tar -czf hello.txt howdy.txt > greetings.tar.gz` , который хранит файлы отдельно в tarball.

Прочитайте Использование кота онлайн: https://riptutorial.com/ru/bash/topic/441/использование-кота

# глава 25: Использование сортировки

## Вступление

sort - это команда Unix для упорядочивания данных в файлах (-ax) в последовательности.

## Синтаксис

- sort [option] filename

## параметры

| вариант | Имея в виду |
| --- | --- |
| -u | Сделайте уникальные уникальные строки |

## замечания

Полное руководство пользователя по `sort` читающей онлайн

## Examples

### Вывод команды сортировки

`sort` используется для сортировки списка строк.

#### Вход из файла

```
sort file.txt
```

#### Вход из команды

Вы можете сортировать любую команду вывода. В примере приведен список файлов по шаблону.

```
find * -name pattern | sort
```

### Сделать вывод уникальным

Если каждая строка вывода должна быть уникальной, добавьте опцию `-u` .

Отобразить владельца файлов в папке

```
ls -l | awk '{print $3}' | sort -u
```

## Числовая сортировка

Предположим, что у нас есть этот файл:

```
test>>cat file
10.Gryffindor
4.Hogwarts
2.Harry
3.Dumbledore
1.The sorting hat
```

Чтобы отсортировать этот файл в численном порядке, используйте сортировку с опцией -n:

```
test>>sort -n file
```

Это должно сортировать файл, как показано ниже:

```
1.The sorting hat
2.Harry
3.Dumbledore
4.Hogwarts
10.Gryffindor
```

Обратный порядок сортировки: Чтобы изменить порядок сортировки, используйте параметр -r

Чтобы изменить порядок сортировки вышеуказанного файла, используйте:

```
sort -rn file
```

Это должно сортировать файл, как показано ниже:

```
10.Gryffindor
4.Hogwarts
3.Dumbledore
2.Harry
1.The sorting hat
```

## Сортировать по ключам

Предположим, что у нас есть этот файл:

```
test>>cat Hogwarts
  Harry        Malfoy        Rowena        Helga
  Gryffindor   Slytherin     Ravenclaw     Hufflepuff
```

```
  Hermione     Goyle         Lockhart      Tonks
  Ron          Snape         Olivander     Newt
  Ron          Goyle         Flitwick      Sprout
```

Для сортировки этого файла с использованием столбца в качестве ключа используйте параметр k:

```
test>>sort -k 2 Hogwarts
```

Это сортирует файл со столбцом 2 в качестве ключа:

```
  Ron        Goyle        Flitwick        Sprout
  Hermione   Goyle        Lockhart        Tonks
  Harry      Malfoy       Rowena          Helga
  Gryffindor Slytherin    Ravenclaw       Hufflepuff
  Ron        Snape        Olivander       Newt
```

Теперь, если нам нужно отсортировать файл с дополнительным ключом вместе с использованием первичного ключа:

```
sort -k 2,2 -k 1,1 Hogwarts
```

Сначала он сортирует файл со столбцом 2 в качестве первичного ключа, а затем сортирует файл со столбцом 1 в качестве вторичного ключа:

```
  Hermione        Goyle        Lockhart        Tonks
  Ron             Goyle        Flitwick        Sprout
  Harry           Malfoy       Rowena          Helga
  Gryffindor      Slytherin    Ravenclaw       Hufflepuff
  Ron             Snape        Olivander       Newt
```

Если нам нужно отсортировать файл с более чем одним ключом, то для каждой опции -k нам нужно указать, где заканчивается сортировка. Итак, -k1,1 означает начало сортировки в первом столбце и конец сортировки в первом столбце.

**-t вариант**

В предыдущем примере файл имел дельта-разделитель по умолчанию. В случае сортировки файла, который имеет нестандартный делиметр, нам нужна опция -t, чтобы указать делиметр. Предположим, у нас есть файл, как показано ниже:

```
test>>cat file
5.|Gryffindor
4.|Hogwarts
2.|Harry
3.|Dumbledore
1.|The sorting hat
```

Чтобы отсортировать этот файл в соответствии со вторым столбцом, используйте:

```
test>>sort -t "|" -k 2 file
```

Это отсортирует файл следующим образом:

```
3.|Dumbledore
5.|Gryffindor
2.|Harry
4.|Hogwarts
1.|The sorting hat
```

Прочитайте Использование сортировки онлайн: https://riptutorial.com/ru/bash/topic/6834/использование-сортировки

# глава 26: квотирование

## Синтаксис

- \ C (любой символ, кроме новой строки)
- «все буквальные, кроме одинарных кавычек»; 'this:' \ " - это одиночная цитата '
- $ 'only \\ and \' являются особыми; \ n = новая строка и т. д.
- "$ variable и другой текст; \" \\\ $ \ `являются специальными"

## Examples

### Новые строки и контрольные символы

Новая строка может быть включена в строку с одной строкой или с двойными кавычками. Обратите внимание, что обратная косая черта-новая строка не приводит к новой строке, разрыв строки игнорируется.

```
newline1='
'
newline2="
"
newline3=$'\n'
empty=\

echo "Line${newline1}break"
echo "Line${newline2}break"
echo "Line${newline3}break"
echo "No line break${empty} here"
```

Внутри строк долларовой котировки, обратная косая черта или обратная косая черта-восьмеричность могут использоваться для вставки управляющих символов, как и во многих других языках программирования.

```
echo $'Tab: [\t]'
echo $'Tab again: [\009]'
echo $'Form feed: [\f]'
echo $'Line\nbreak'
```

### Двойные кавычки для замены переменных и команд

Переменные замены должны использоваться только в двойных кавычках.

```
calculation='2 * 3'
echo "$calculation"         # prints 2 * 3
echo $calculation           # prints 2, the list of files in the current directory, and 3
echo "$(($calculation))"    # prints 6
```

Вне двойных кавычек, `$var` принимает значение `var` , разбивает его на части, разделенные пробелами, и интерпретирует каждую часть как шаблон glob (wildcard). Если вы не хотите этого поведения, всегда кладите `$var` внутри двойных кавычек: `"$var"` .

То же самое относится к подстановкам команд: `"$(mycommand)"` - это вывод `mycommand` , `$(mycommand)` - результат split + glob на выходе.

```
echo "$var"             # good
echo "$(mycommand)"     # good
another=$var            # also works, assignment is implicitly double-quoted
make -D THING=$var      # BAD! This is not a bash assignment.
make -D THING="$var"    # good
make -D "THING=$var"    # also good
```

Командные подстановки получают свои собственные контексты цитирования. Запись произвольно вложенные замен легко , потому что синтаксический анализатор будет следить за глубину вложенности , а не жадность поиска первого `"` символ Синтаксиса фломастер StackOverflow разбирает это неправильно, однако , например..:

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # “formatted text: a + b = 0000”
```

Переменные аргументы для подстановки команд должны быть также заключены в двойные кавычки внутри расширений:

```
echo "$(mycommand "$arg1" "$arg2")"
```

## Цитирование литералов

Все примеры в этом абзаце распечатывают строку

```
!"#$&'()*;<=>?  @[\]^`{|}~
```

Обратная косая черта цитирует следующий символ, т. Е. Следующий символ интерпретируется буквально. Единственным исключением является новая строка: backslash-newline расширяется до пустой строки.

```
echo \!\"\#\$\&\'\(\)\*\;\<\=\>\?\ \ \@\[\\\]\^\`\{\|\}\~
```

Весь текст между одинарными кавычками (форвардные кавычки `'` , также известный как апостроф, печатается буквально. Даже обратная косая черта выступает сама по себе, и невозможно включить одну цитату; вместо этого вы можете остановить литеральную строку, включить буквенную одинарную цитату с обратной косой чертой и снова запустить литерала. Таким образом, 4-символьная последовательность `'\''` эффективно позволяет включать одну кавычку в строковой литературе.

```
echo '!"#$&'\''()*;<=>?  @[\]^`{|}~'
```

```
#          ^^^^
```

Доллар-одиночная кавычка запускает строковый литерал `$'…'` как и многие другие языки программирования, где обратная косая черта цитирует следующий символ.

```
echo $'!"#$&\'()*;<=>?  @[\\]^`{|}~'
#           ^^            ^^
```

Двойные кавычки `"` разграничить пол-буквенные строки , где только символы `" \ $` и `` ` `` сохранить их особое значение. Этим символам нужна обратная косая черта перед ними (обратите внимание, что если обратная косая черта сопровождается каким-то другим символом, обратная косая черта остается). Двойные кавычки в основном полезны при включении переменной или подстановки команд.

```
echo "!\"#\$&'()*;<=>?  @[\\]^\`{|}~"
#      ^^                  ^^  ^^
echo "!\"#\$&'()*;<=>?  @[\]^\`{|}~"
#      ^^                  ^  ^^      \[ prints \[
```

Взаимодействуйте, будьте осторожны `!` запускает расширение истории внутри двойных кавычек: `"!oops"` ищет более старую команду, содержащую `oops` ; `"\!oops"` не выполняет расширение истории, но сохраняет обратную косую черту. Это не происходит в сценариях.

## Разница между двойной кавычкой и одиночной кавычкой

| Двойная цена | Единая котировка |
|---|---|
| Позволяет изменять переменную | Предотвращает изменение переменной |
| Позволяет расширение истории, если включено | Предотвращает расширение истории |
| Позволяет заменить команду | Предотвращает подстановку команд |
| `*` и `@` могут иметь особое значение | `*` и `@` всегда являются литералами |
| Может содержать как одинарные, так и двойные кавычки | Одинарная кавычка не допускается внутри одной кавычки |
| `$` , `` ` `` , `"` , `\` могут быть экранированы с `\` чтобы предотвратить их особое значение | Все они являются литералами |

**Свойства, которые являются общими для обоих:**

- Предотвращает зависание
- Предотвращает разбиение слов

Примеры:

```
$ echo "!cat"
echo "cat file"
cat file
$ echo '!cat'
!cat
echo "\"'\""
"'"
$ a='var'
$ echo '$a'
$a
$ echo "$a"
var
```

Прочитайте квотирование онлайн: https://riptutorial.com/ru/bash/topic/729/квотирование

# глава 27: Когда использовать eval

## Вступление

Прежде всего, знайте, что вы делаете! Во-вторых, в то время как вам следует избегать использования `eval` , если его использование делает более чистый код, продолжайте.

## Examples

### Использование Eval

Например, рассмотрим следующее, которое задает содержимое `$@` содержимому данной переменной:

```
a=(1 2 3)
eval set -- "${a[@]}"
```

Этот код часто сопровождается `getopt` или `getopts` чтобы установить `$@` на вывод вышеупомянутых парсеров параметров, однако вы также можете использовать его для создания простой функции `pop` которая может работать с переменными молча и напрямую, не сохраняя результат исходная переменная:

```
isnum()
{
    # is argument an integer?
    local re='^[0-9]+$'
    if [[ -n $1 ]]; then
        [[ $1 =~ $re ]] && return 0
        return 1
    else
        return 2
    fi
}

isvar()
{
    if isnum "$1"; then
        return 1
    fi
    local arr="$(eval eval -- echo -n "\$$1")"
    if [[ -n ${arr[@]} ]]; then
        return 0
    fi
    return 1
}

pop()
{
    if [[ -z $@ ]]; then
        return 1
```

```
    fi

    local var=
    local isvar=0
    local arr=()

    if isvar "$1"; then # let's check to see if this is a variable or just a bare array
        var="$1"
        isvar=1
        arr=($(eval eval -- echo -n "\${$1[@]}")) # if it is a var, get its contents
    else
        arr=($@)
    fi

    # we need to reverse the contents of $@ so that we can shift
    # the last element into nothingness
    arr=($(awk <<<"${arr[@]}" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # set $@ to ${arr[@]} so that we can run shift against it.
    eval set -- "${arr[@]}"

    shift # remove the last element

    # put the array back to its original order
    arr=($(awk <<<"$@" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # echo the contents for the benefit of users and for bare arrays
    echo "${arr[@]}"

    if ((isvar)); then
        # set the contents of the original var to the new modified array
        eval -- "$var=(${arr[@]})"
    fi
}
```

## Использование Eval с Getopt

В то время как eval может не понадобиться для функции `pop` , это необходимо, когда вы используете `getopt` :

Рассмотрим следующую функцию, которая принимает `-h` как вариант:

```
f()
{
    local __me__="${FUNCNAME[0]}"
    local argv="$(getopt -o 'h' -n $__me__ -- "$@")"

    eval set -- "$argv"

    while :; do
        case "$1" in
            -h)
              echo "LOLOLOLOL"
              return 0
              ;;
            --)
              shift
              break
```

```
            ;;
    done

    echo "$@"
}
```

Без `eval set -- "$argv"` генерирует `-h --` вместо желаемого `(-h --)` и затем вводит бесконечный цикл, потому что `-h --` не соответствует `--` или `-h` .

Прочитайте Когда использовать eval онлайн: https://riptutorial.com/ru/bash/topic/10113/когда-использовать-eval

# глава 28: Команда cut

## Вступление

Команда `cut` - это быстрый способ извлечения частей строк текстовых файлов. Он относится к самым старым командам Unix. Его наиболее популярными реализациями является версия GNU, найденная в Linux, и версия FreeBSD, найденная на MacOS, но каждый аромат Unix имеет свои собственные. См. Ниже различия. Строки ввода считываются либо из `stdin` либо из файлов, перечисленных в качестве аргументов в командной строке.

## Синтаксис

- cut -f1,3 # извлечь первое *и* третье **поля** с *разделителями табуляции* (из stdin)
- cut -f1-3 # экстракт *от* первого *до* третьего поля (в конце включены)
- cut -f-3 # -3 интерпретируется как 1-3
- cut -f2- # 2- интерпретируется как *от второго до последнего*
- cut -c1-5,10 # extract from stdin **символы** в позициях 1,2,3,4,5,10
- cut -s -f1 # подавлять строки, не содержащие разделителей
- cut --complement -f3 # (только разрезание GNU) извлекает *все* поля, *кроме* третьего

## параметры

| параметр | подробности |
|---|---|
| -f, --fields | Выбор на местах |
| -d, --delimiter | Разделитель для выбора на местах |
| -c, --characters | Выбор на основе символов, игнорируемый разделитель или ошибка |
| -s, - только разграниченные | Удерживать строки без разделительных символов (напечатано как - иначе) |
| --complement | Перевернутый выбор (извлекать все, *кроме* указанных полей / символов |

| параметр | подробности |
| --- | --- |
| --output-разделитель | Укажите, когда он должен отличаться от входного разделителя |

# замечания

### 1. Синтаксические различия

Длинные параметры в приведенной выше таблице поддерживаются только версией GNU.

### 2. Персонаж не получает специального лечения

Например, FreeBSD `cut` (который поставляется с MacOS) не имеет ключа `--complement` , а в случае диапазонов `colrm` вместо него можно использовать команду `colrm` :

```
$ cut --complement -c3-5 <<<"123456789"
126789

$ colrm 3 5 <<<"123456789"
126789
```

Однако существует большая разница, поскольку `colrm` обрабатывает символы TAB (ASCII 9) как реальные таблицы до следующего кратного восьми, а обратные пространства (ASCII 8) равны -1; напротив, `cut` обрабатывает всех персонажей как одну ширину столбца.

```
$ colrm  3 8 <<<$'12\tABCDEF' # Input string has an embedded TAB
12ABCDEF

$ cut --complement -c3-8 <<<$'12\tABCDEF'
12F
```

### 3. (Все еще нет) Интернационализация

Когда был разработан `cut` , все символы были одним байтом, а интернационализация не была проблемой. При написании систем с более широкими символами стало популярным решение, принятое POSIX, заключалось в том, чтобы подгонять между старым ключом `-c` , который должен сохранять смысл выбора *символов,* независимо от того, сколько байтов в ширину, и ввести новый переключатель `-b` который должен выберите *байты,* независимо от текущей кодировки символов. В большинстве популярных реализаций `-b` был введен и работает, но `-c` все еще работает точно так же, как `-b` но не так, как должно. Например, при `cut` GNU:

Кажется, что спам-фильтр SE черным списком английских текстов с изолированными иероглифами кандзи в них. Я не мог преодолеть это ограничение, поэтому следующие примеры менее выразительны, чем могли бы быть.

```
# In an encoding where each character in the input string is three bytes wide,
```

```
# Selecting bytes 1-6 yields the first two characters (correct)
$ LC_ALL=ja_JP.UTF-8 cut -b1-6 kanji.utf-8.txt
...first two characters of each line...


# Selecting all three characters with the -c switch doesn't work.
# It behaves like -b, contrary to documentation.
$ LC_ALL=ja_JP.UTF-8 cut -c1-3 kanji.utf-8.txt
...first character of each line...

# In this case, an illegal UTF-8 string is produced.
# The -n switch would prevent this, if implemented.
$ LC_ALL=ja_JP.UTF-8 cut -n -c2 kanji.utf-8.txt
...second byte, which is an illegal UTF-8 sequence...
```

Если ваши символы находятся за пределами диапазона ASCII, и вы хотите использовать `cut` , вы всегда должны знать ширину символов в кодировке и использовать `-b` соответственно. Если и когда `-c` начинает работать как задокументированный, вам не придется менять свои скрипты.

### 4. Сравнение скорости

`cut` ограничения «s есть люди , сомневающихся свою полезность. Фактически, та же функциональность может быть достигнута благодаря более мощным, более популярным утилитам. Однако преимуществом `cut` является его *производительность* . Ниже приведены некоторые сравнения скорости. `test.txt` имеет три миллиона строк с пятью полями, разделенными пробелами. Для `awk` теста использовался `mawk` , потому что он быстрее, чем GNU `awk` . Сама оболочка (последняя строка) является наихудшим исполнителем. Временные значения (в секундах) - это то, что команда `time` дает как *реальное время* .

(Чтобы избежать недоразумений: все тестируемые команды дали одинаковый результат с данным вводом, но они, конечно, не эквивалентны и будут давать разные результаты в разных ситуациях, в частности, если поля были разделены переменным числом пробелов)

| команда | Время |
|---|---|
| `cut -d ' ' -f1,2 test.txt` | 1.138s |
| `awk '{print $1 $2}' test.txt` | 1.688s |
| `join -a1 -o1.1,1.2 test.txt /dev/null` | 1.767s |
| `perl -lane 'print "@F[1,2]"' test.txt` | 11.390s |
| `grep -o '^\([^ ]*\) \([^ ]*\)' test.txt` | 22.925s |
| `sed -e 's/^\([^ ]*\) \([^ ]*\).*$/\1 \2/' test.txt` | 52.122s |
| `while read ab _; do echo $a $b; done <test.txt` | 55.582s |

**5. Справочные справочные страницы**

- OpenGroup
- GNU
- FreeBSD

# Examples

## Основное использование

Типичное использование файлов CSV-типа, где каждая строка состоит из полей, разделенных разделителем, указанным опцией `-d` . Разделителем по умолчанию является символ TAB. Предположим, у вас есть файл данных `data.txt` с такими строками, как

```
0 0 755 1482941948.8024
102 33 4755 1240562224.3205
1003 1 644 1219943831.2367
```

затем

```
# extract the third space-delimited field
$ cut -d ' ' -f3 data.txt
755
4755
644

# extract the second dot-delimited field
$ cut -d. -f2 data.txt
8024
3205
2367

# extract the character range from the 20th through the 25th character
$ cut -c20-25 data.txt
948.80
056222
943831
```

Как обычно, между переключателем и его параметром могут быть дополнительные пробелы: `-d,` совпадает с `-d ,`

GNU `cut` позволяет указать параметр `--output-delimiter` : (независимая функция этого примера заключается в том, что точку с запятой в качестве входного разделителя необходимо экранировать, чтобы избежать его специальной обработки оболочкой)

```
$ cut --output-delimiter=, -d\; -f1,2 <<<"a;b;c;d"
a,b
```

## Только один разделитель

У вас не может быть более одного разделителя: если вы укажете что-то вроде `-d ",;:"` , некоторые реализации будут использовать только первый символ в качестве разделителя (в данном случае - запятую). Другие реализации (например, GNU `cut` ) будут давать вы получите сообщение об ошибке.

```
$ cut -d ",;:" -f2 <<<"J.Smith,1 Main Road,cell:1234567890;land:4081234567"
cut: the delimiter must be a single character
Try `cut --help' for more information.
```

### Повторные разделители интерпретируются как пустые поля

```
$ cut -d, -f1,3 <<<"a,,b,c,d,e"
a,b
```

довольно очевидна, но с помощью строк с разделителями по пространству это может быть менее очевидным для некоторых

```
$ cut -d ' ' -f1,3 <<<"a  b c d e"
a b
```

`cut` не может использоваться для анализа аргументов, как это делают оболочки и другие программы.

### Нет цитирования

Невозможно защитить разделитель. Таблицы и аналогичное программное обеспечение для обработки CSV обычно могут распознавать символ цитирования текста, который позволяет определять строки, содержащие разделитель. С `cut` вы не можете.

```
$ cut -d, -f3 <<<'John,Smith,"1, Main Street"'
"1
```

### Извлечение, а не манипулирование

Вы можете извлекать только части строк, а не изменять порядок или повторять поля.

```
$ cut -d, -f2,1 <<<'John,Smith,USA' ## Just like -f1,2
John,Smith
$ cut -d, -f2,2 <<<'John,Smith,USA' ## Just like -f2
Smith
```

Прочитайте Команда cut онлайн: https://riptutorial.com/ru/bash/topic/8762/команда-cut

# глава 29: Контрольные структуры

## Синтаксис

- ["$ 1" = "$ 2"] #A "[" Скобка - это фактически команда. Из-за этого требуется пространство для него и после него.
- test "$ 1" = "$ 2" #Test является синонимом команды "["

## параметры

| Параметр [или тест | подробности |
|---|---|
| **Операторы файлов** | подробности |
| `-e "$file"` | Возвращает true, если файл существует. |
| `-d "$file"` | Возвращает true, если файл существует и является каталогом |
| `-f "$file"` | Возвращает true, если файл существует и является обычным файлом |
| `-h "$file"` | Возвращает true, если файл существует и является символической ссылкой |
| **Компоненты строк** | подробности |
| `-z "$str"` | Истинно, если длина строки равна нулю |
| `-n "$str` | Истинно, если длина строки отлична от нуля |
| `"$str" = "$str2"` | Истинно, если строка $ str равна строке $ str2. Не лучше для целых чисел. Он может работать, но будет несовместимым |
| `"$str" != "$str2"` | Истинно, если строки не равны |
| **Целочисленные компараторы** | подробности |
| `"$int1" -eq "$int2"` | Истинно, если целые числа равны |
| `"$int1" -ne "$int2"` | Истинно, если целые числа не равны |
| `"$int1" -gt "$int2"` | Истинно, если int1 больше, чем int 2 |

| Параметр [или тест | подробности |
|---|---|
| `"$int1" -ge "$int2"` | Истинно, если int1 больше или равно int2 |
| `"$int1" -lt "$int2"` | Истинно, если int1 меньше, чем int 2 |
| `"$int1" -le "$int2"` | Истинно, если int1 меньше или равно int2 |

## замечания

В bash имеется множество параметров компаратора. Не все еще перечислены здесь.

# Examples

### Если утверждение

```
if [[ $1 -eq 1 ]]; then
    echo "1 was passed in the first parameter"
elif [[ $1 -gt 2 ]]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

Замыкание `fi` необходимо, но предложения `elif` и / или `else` могут быть опущены.

Точки с запятой до `then` являются стандартным синтаксисом для объединения двух команд в одну строку; они могут быть опущены , если только `then` перемещается на следующую строку.

Важно понимать, что скобки `[[` не являются частью синтаксиса, но рассматриваются как команда; это код выхода из этой команды, которая тестируется. Поэтому вы всегда должны включать пробелы вокруг скобок.

Это также означает, что результат любой команды может быть протестирован. Если код выхода из команды равен нулю, утверждение считается истинным.

```
if grep "foo" bar.txt; then
    echo "foo was found"
else
    echo "foo was not found"
fi
```

Математические выражения, размещенные внутри двойных скобок, также возвращают 0 или 1 таким же образом, а также могут быть протестированы:

```
if (( $1 + 5 > 91 )); then
    echo "$1 is greater than 86"
```

```
fi
```

Вы также можете столкнуться , `if` заявления с одиночными скобками. Они определены в стандарте POSIX и гарантированно работают во всех POSIX-совместимых оболочках, включая Bash. Синтаксис очень похож на синтаксис в Bash:

```
if [ "$1" -eq 1 ]; then
    echo "1 was passed in the first parameter"
elif [ "$1" -gt 2 ]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

## Пока цикл

```
#! /bin/bash

i=0

while [ $i -lt 5 ] #While i is less than 5
do
    echo "i is currently $i"
    i=$[$i+1] #Not the lack of spaces around the brackets. This makes it a not a test
expression
done #ends the loop
```

Посмотрите, что во время теста есть пробелы вокруг скобок (после утверждения while). Эти пространства необходимы.

Эта петля выводит:

```
i is currently 0
i is currently 1
i is currently 2
i is currently 3
i is currently 4
```

## Для цикла

```
#! /bin/bash

for i in 1 "test" 3; do #Each space separated statement is assigned to i
    echo $i
done
```

Другие команды могут генерировать операторы для перебора. См. Пример «Использование для цикла для ввода чисел».

Эти результаты:

```
1
test
3
```

### Использование цикла для перебора Iterate Over Numbers

```
#! /bin/bash

for i in {1..10}; do # {1..10} expands to "1 2 3 4 5 6 7 8 9 10"
    echo $i
done
```

Это обеспечивает следующее:

```
1
2
3
4
5
6
7
8
8
10
```

### Для цикла с синтаксисом C-стиля

Основной формат C-стиль `for` петли:

```
for (( variable assignment; condition; iteration process ))
```

Заметки:

- Назначение переменной внутри цикла C-стиля `for` цикла может содержать пробелы, в отличие от обычного назначения
- Переменным внутри цикла C `for` цикла не предшествует `$` .

Пример:

```
for (( i = 0; i < 10; i++ ))
do
    echo "The iteration number is $i"
done
```

Также мы можем обрабатывать несколько переменных внутри цикла C `for` цикла:

```
for (( i = 0, j = 0; i < 10; i++, j = i * i ))
do
    echo "The square of $i is equal to $j"
done
```

## До цикла

Пока цикл не будет выполняться до тех пор, пока условие не будет истинным

```
i=5
until [[ i -eq 10 ]]; do #Checks if i=10
    echo "i=$i" #Print the value of i
    i=$((i+1)) #Increment i by 1
done
```

Выход:

```
i=5
i=6
i=7
i=8
i=9
```

Когда `i` достигает 10, условие до тех пор, пока цикл не станет истинным, и цикл закончится.

## продолжить и разбить

Пример продолжения

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
            continue # skip to the next value in "series"
    fi
    command 3
done
```

Пример разрыва

```
for i in [series]
do
    command 4
    if (condition) # Condition to break the loop
    then
            command 5 # Command if the loop needs to be broken
            break
    fi
    command 6 # Command to run if the "condition" is never true
done
```

## Зацикливание по массиву

`for` цикла:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
done
```

Или же

```
for ((i=0;i<${#arr[@]};i++));do
    echo "${arr[$i]}"
done
```

`while` loop:

```
i=0
while [ $i -lt ${#arr[@]} ];do
    echo "${arr[$i]}"
    i=$(expr $i + 1)
done
```

Или же

```
i=0
while (( $i < ${#arr[@]} ));do
    echo "${arr[$i]}"
    ((i++))
done
```

## Перерыв петли

Разрыв нескольких циклов:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break 2
    done
done
```

Выход:

```
a
a
```

Разрыв одного цикла:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
```

```
        break
    done
done
```

Выход:

```
a
a
b
a
c
a
d
a
e
a
f
a
```

## Оператор switch с футляром

С помощью оператора `case` вы можете сопоставлять значения с одной переменной.

Аргумент, переданный `case` , расширяется и пытается сопоставляться с каждым шаблоном.

Если совпадение найдено, команды upto `;;` выполняются.

```
case "$BASH_VERSION" in
 [34]*)
    echo {1..4}
    ;;
  *)
    seq -s" " 1 4
esac
```

Шаблон - это не регулярные выражения, а совпадение шаблонов (aka globs).

## Для Loop без параметра списка слов

```
for arg; do
    echo arg=$arg
done
```

В цикле `for` без параметра списка слов вместо этого будут выполняться итерация по позиционным параметрам. Другими словами, приведенный выше пример эквивалентен этому коду:

```
for arg in "$@"; do
    echo arg=$arg
done
```

Другими словами, если вы поймете, что пишете `for i in "$@"; do ...; done` , просто перенесите `in` части, а просто писать `for i; do ...; done` .

## Условное выполнение списков команд

## Как использовать условное выполнение списков команд

Любая встроенная команда, выражение или функция, а также любая внешняя команда или скрипт могут выполняться условно с использованием `&&` *(и)* и `||` *(или)* операторов.

Например, это приведет только к печати текущего каталога, если команда `cd` была успешной.

```
cd my_directory && pwd
```

Аналогично, это приведет к выходу из строя команды `cd` , предотвращая катастрофу:

```
cd my_directory || exit
rm -rf *
```

При объединении нескольких операторов таким образом важно помнить, что (в отличие от многих языков C-стиля) эти операторы не имеют приоритета и являются лево-ассоциативными .

Таким образом, это заявление будет работать как ожидалось ...

```
cd my_directory && pwd || echo "No such directory"
```

- Если `cd` завершается успешно, выполняется `&& pwd` и печатается текущее имя рабочего каталога. Если `pwd` не сработает (редкость) `|| echo ...` не будет выполняться.
- Если `cd` не удастся, `&& pwd` будет пропущен, а значение `|| echo ...` будет работать.

Но это не будет (если вы думаете, `if...then...else` ) ...

```
cd my_directory && ls || echo "No such directory"
```

- Если `cd` терпит неудачу, `&& ls` пропущен и `|| echo ...` выполняется.
- Если `cd` завершается успешно, выполняется `&& ls` .
    - Если `ls` преуспеет, `|| echo ...` игнорируется. *(Все идет нормально)*
    - ***НО ... если `ls` терпит неудачу, то `|| echo ...` также будет выполнено.***

***Это команда `ls`*** , а не `cd` , ***это предыдущая команда*** .

## Зачем использовать условное выполнение списков команд

Условное исполнение - это волосы быстрее, чем `if...then` но его главное преимущество -

позволить функциям и сценариям выходить рано или «короткое замыкание».

В отличие от многих языков, таких как `C` где ядро явно выделяется для структур и переменных и таких (и, следовательно, должно быть освобождено), `bash` обрабатывает это под обложками. В большинстве случаев нам не нужно ничего убирать, прежде чем покидать эту функцию. Оператор `return` освобождает все локальные функции и выполнение записи на обратном адресе в стеке.

Возвращение из функций или выход из сценариев как можно скорее может значительно повысить производительность и снизить нагрузку на систему, избегая ненужного выполнения кода. Например...

```
my_function () {

    ### ALWAYS CHECK THE RETURN CODE

    # one argument required. "" evaluates to false(1)
    [[ "$1" ]]              || return 1

    # work with the argument. exit on failure
    do_something_with "$1" || return 1
    do_something_else      || return 1

    # Success! no failures detected, or we wouldn't be here
    return 0
}
```

Прочитайте Контрольные структуры онлайн: https://riptutorial.com/ru/bash/topic/420/контрольные-структуры

# глава 30: Копирование (cp)

## Синтаксис

- cp [опции] источник назначения

## параметры

| вариант | Описание |
|---|---|
| `-a` , `-archive` | Объединяет параметры `d` , `p` и `r` |
| `-b` , `-backup` | Перед удалением делает резервную копию |
| `-d` , `--no-deference` | Сохраняет ссылки |
| `-f` , `--force` | Удалить существующие адресаты без запроса пользователя |
| `-i` , `--interactive` | Показать приглашение перед перезаписью |
| `-l` , `--link` | Вместо копирования вместо этого вместо |
| `-p` , `--preserve` | Сохранять атрибуты файлов, когда это возможно |
| `-R` , `--recursive` | Рекурсивно копировать каталоги |

## Examples

### Скопируйте один файл

Скопируйте `foo.txt` из `/path/to/source/` в `/path/to/target/folder/`

```
cp /path/to/source/foo.txt /path/to/target/folder/
```

Скопируйте `foo.txt` из `/path/to/source/` в `/path/to/target/folder/` в файл с именем `bar.txt`

```
cp /path/to/source/foo.txt /path/to/target/folder/bar.txt
```

### Скопировать папки

скопировать папку `foo` в `bar` папок

```
cp -r /path/to/foo /path/to/bar
```

если перед выдачей команды существует папка, тогда `foo` и ее содержимое будут скопированы в `bar` папок. Однако, если `bar` не существует до выдачи команды, тогда будет создана `bar` папок, и содержимое `foo` будет помещено в `bar`

Прочитайте Копирование (cp) онлайн: https://riptutorial.com/ru/bash/topic/4030/копирование--cp-

# глава 31: Ловушки

## Examples

### Пробелы при назначении переменных

Пробелы имеют значение при назначении переменных.

```
foo = 'bar' # incorrect
foo= 'bar'  # incorrect
foo='bar'   # correct
```

Первые два будут приводить к синтаксическим ошибкам (или, что еще хуже, выполнению неправильной команды). Последний пример корректно установит переменную `$foo` в текст « bar».

### Отсутствует последняя строка в файле

В стандарте C говорится, что файлы должны заканчиваться новой строкой, поэтому, если EOF находится в конце строки, эта строка не может быть пропущена некоторыми командами. В качестве примера:

```
$ echo 'one\ntwo\nthree\c' > file.txt

$ cat file.txt
one
two
three

$ while read line ; do echo "line $line" ; done < file.txt
one
two
```

Чтобы убедиться, что это правильно работает в приведенном выше примере, добавьте тест, чтобы он продолжал цикл, если последняя строка не пуста.

```
$ while read line || [ -n "$line" ] ; do echo "line $line" ; done < file.txt
one
two
three
```

### Неудачные команды не останавливают выполнение сценария

В большинстве языков сценариев, если вызов функции завершается неудачно, он может вызывать исключение и останавливать выполнение программы. Команды Bash не имеют исключений, но у них есть коды выхода. Однако отказ от ненужных сигналов кода выхода,

однако, отличный от нуля код выхода не остановит выполнение программы.

Это может привести к опасным (хотя, предположительно, надуманным) ситуациям:

```
#!/bin/bash
cd ~/non/existent/directory
rm -rf *
```

Если `cd` -ing в этом каталоге не удастся, Bash проигнорирует сбой и перейдет к следующей команде, очистив каталог, из которого вы запускали скрипт.

Лучшим способом решения этой проблемы является использование команды set :

```
#!/bin/bash
set -e
cd ~/non/existent/directory
rm -rf *
```

`set -e` сообщает Bash немедленно выйти из сценария, если какая-либо команда возвращает ненулевой статус.

Прочитайте Ловушки онлайн: https://riptutorial.com/ru/bash/topic/3656/ловушки

# глава 32: Массивы

## Examples

### Назначение массива

#### Назначение списка

Если вы знакомы с Perl, C или Java, вы можете подумать, что Bash будет использовать запятые для разделения элементов массива, однако это не так; вместо этого Bash использует пробелы:

```
# Array in Perl
my @array = (1, 2, 3, 4);
```

```
# Array in Bash
array=(1 2 3 4)
```

Создайте массив с новыми элементами:

```
array=('first element' 'second element' 'third element')
```

#### Назначение подзаголовка

Создайте массив с явными индексами элементов:

```
array=([3]='fourth element' [4]='fifth element')
```

#### Назначение по индексу

```
array[0]='first element'
array[1]='second element'
```

#### Назначение по имени (ассоциативный массив)

4,0

```
declare -A array
array[first]='First element'
array[second]='Second element'
```

#### Динамическое присвоение

Создайте массив из вывода другой команды, например, используйте **seq,** чтобы получить диапазон от 1 до 10:

```
array=(`seq 1 10`)
```

Назначение входных аргументов скрипта:

```
array=("$@")
```

Назначение в циклах:

```
while read -r; do
    #array+=("$REPLY")      # Array append
    array[$i]="$REPLY"      # Assignment by index
    let i++                 # Increment index
done < <(seq 1 10)  # command substitution
echo ${array[@]}    # output: 1 2 3 4 5 6 7 8 9 10
```

где `$REPLY` всегда является текущим

## Доступ к элементам массива

Элемент печати с индексом 0

```
echo "${array[0]}"
```

4,3

Печать последнего элемента с использованием синтаксиса расширения подстроки

```
echo "${arr[@]: -1 }"
```

4,3

Печать последнего элемента с использованием синтаксиса подстроки

```
echo "${array[-1]}"
```

Распечатайте все элементы, каждый из которых указан отдельно

```
echo "${array[@]}"
```

Печать всех элементов в виде одной строки с кавычками

```
echo "${array[*]}"
```

Распечатайте все элементы из индекса 1, каждый из которых указан отдельно

```
echo "${array[@]:1}"
```

Распечатайте 3 элемента из индекса 1, каждый из которых указан отдельно

```
echo "${array[@]:1:3}"
```

### Строковые операции

Если обратиться к одному элементу, разрешены строковые операции:

```
array=(zero one two)
echo "${array[0]:0:3}" # gives out zer (chars at position 0, 1 and 2 in the string zero)
echo "${array[0]:1:3}" # gives out ero (chars at position 1, 2 and 3 in the string zero)
```

поэтому `${array[$i]:N:M}` выдает строку из `N` й позиции (начиная с 0) в строке `${array[$i]}` с `M` следующими символами.

## Длина массива

`${#array[@]}` дает длину массива `${array[@]}` :

```
array=('first element' 'second element' 'third element')
echo "${#array[@]}" # gives out a length of 3
```

Это также работает со строками в отдельных элементах:

```
echo "${#array[0]}"    # gives out the lenght of the string at element 0: 13
```

## Модификация массива

### Изменить индекс

Инициализировать или обновить определенный элемент в массиве

```
array[10]="elevenths element"    # because it's starting with 0
```

3,1

### присоединять

Измените массив, добавив элементы в конец, если индекс не указан.

```
array+=('fourth element' 'fifth element')
```

Замените весь массив новым списком параметров.

```
array=("${array[@]}" "fourth element" "fifth element")
```

Добавьте элемент в начале:

```
array=("new element" "${array[@]}")
```

**Вставить**

Вставьте элемент с заданным индексом:

```
arr=(a b c d)
# insert an element at index 2
i=2
arr=("${arr[@]:0:$i}" 'new' "${arr[@]:$i}")
echo "${arr[2]}" #output: new
```

**удалять**

Удалите индексы массива с помощью `unset` builtin:

```
arr=(a b c)
echo "${arr[@]}"   # outputs: a b c
echo "${!arr[@]}"  # outputs: 0 1 2
unset -v 'arr[1]'
echo "${arr[@]}"   # outputs: a c
echo "${!arr[@]}"  # outputs: 0 2
```

**сливаться**

```
array3=("${array1[@]}" "${array2[@]}")
```

Это работает и для разреженных массивов.

**Повторная индексация массива**

Это может быть полезно, если элементы были удалены из массива или если вы не уверены в наличии пробелов в массиве. Чтобы воссоздать индексы без пробелов:

```
array=("${array[@]}")
```

## Итерация массива

Итерация массива поставляется в двух вариантах: foreach и classic for-loop:

```
a=(1 2 3 4)
# foreach loop
for y in "${a[@]}"; do
    # act on $y
    echo "$y"
done
# classic for-loop
for ((idx=0; idx < ${#a[@]}; ++idx)); do
    # act on ${a[$idx]}
    echo "${a[$idx]}"
done
```

## Вы также можете перебирать вывод команды:

```
a=($(tr ',' ' ' <<<"a,b,c,d")) # tr can transform one character to another
for y in "${a[@]}"; do
    echo "$y"
done
```

### Уничтожить, удалить или удалить массив

Чтобы уничтожить, удалить или удалить массив:

```
unset array
```

Чтобы уничтожить, удалить или удалить один элемент массива:

```
unset array[10]
```

### Ассоциативные массивы

4,0

#### Объявить ассоциативный массив

```
declare -A aa
```

Объявление ассоциативного массива перед инициализацией или использованием является обязательным.

#### Инициализация элементов

Вы можете инициализировать элементы по одному в следующем порядке:

```
aa[hello]=world
aa[ab]=cd
aa["key with space"]="hello world"
```

Вы также можете инициализировать весь ассоциативный массив в одном выражении:

```
aa=([hello]=world [ab]=cd ["key with space"]="hello world")
```

#### Доступ к элементу ассоциативного массива

```
echo ${aa[hello]}
# Out: world
```

#### Список ассоциативных ключей массива

```
echo "${!aa[@]}"
#Out: hello ab key with space
```

**Отображение значений ассоциативного массива**

```
echo "${aa[@]}"
#Out: world cd hello world
```

**Итерация по ключам ассоциативных массивов и значениям**

```
for key in "${!aa[@]}"; do
    echo "Key:   ${key}"
    echo "Value: ${array[$key]}"
done

# Out:
# Key:   hello
# Value: world
# Key:   ab
# Value: cd
# Key:   key with space
# Value: hello world
```

**Элементы ассоциативного массива графа**

```
echo "${#aa[@]}"
# Out: 3
```

## Список инициализированных индексов

Получить список индексированных индексов в массиве

```
$ arr[2]='second'
$ arr[10]='tenth'
$ arr[25]='twenty five'
$ echo ${!arr[@]}
2 10 25
```

## Цитирование через массив

Наш массив примеров:

```
arr=(a b c d e f)
```

Использование цикла `for..in` :

```
for i in "${arr[@]}"; do
    echo "$i"
done
```

2,04

Использование C-стиля `for` цикла:

```
for ((i=0;i<${#arr[@]};i++)); do
    echo "${arr[$i]}"
done
```

Использование во `while` цикла:

```
i=0
while [ $i -lt ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

2,04

Использование во `while` цикла с числовым условно:

```
i=0
while (( $i < ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Не спользование `until` цикла:

```
i=0
until [ $i -ge ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

2,04

Не спользование `until` контура с числовым условным:

```
i=0
until (( $i >= ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

## Массив из строки

```
stringVar="Apple Orange Banana Mango"
arrayVar=(${stringVar// / })
```

Каждое пространство в строке обозначает новый элемент в результирующем массиве.

```
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[3]} # will print Mango
```

Аналогично, для разделителя могут использоваться другие символы.

```
stringVar="Apple+Orange+Banana+Mango"
arrayVar=(${stringVar//+/ })
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[2]} # will print Banana
```

## Функция вставки массива

Эта функция будет вставлять элемент в массив по заданному индексу:

```
insert(){
    h='
################## insert ########################
# Usage:
#   insert arr_name index element
#
#   Parameters:
#       arr_name    : Name of the array variable
#       index       : Index to insert at
#       element     : Element to insert
##################################################
    '
    [[ $1 = -h ]] && { echo "$h" >/dev/stderr; return 1; }
    declare -n __arr__=$1   # reference to the array variable
    i=$2                    # index to insert at
    el="$3"                 # element to insert
    # handle errors
    [[ ! "$i" =~ ^[0-9]+$ ]] && { echo "E: insert: index must be a valid integer"
>/dev/stderr; return 1; }
    (( $1 < 0 )) && { echo "E: insert: index can not be negative" >/dev/stderr; return 1; }
    # Now insert $el at $i
    __arr__=("${__arr__[@]:0:$i}" "$el" "${__arr__[@]:$i}")
}
```

Использование:

```
insert array_variable_name index element
```

Пример:

```
arr=(a b c d)
echo "${arr[2]}" # output: c
# Now call the insert function and pass the array variable name,
# index to insert at
# and the element to insert
insert arr 2 'New Element'
# 'New Element' was inserted at index 2 in arr, now print them
echo "${arr[2]}" # output: New Element
echo "${arr[3]}" # output: c
```

## Чтение всего файла в массив

Чтение за один шаг:

```
IFS=$'\n' read -r -a arr < file
```

Чтение в цикле:

```
arr=()
while IFS= read -r line; do
  arr+=("$line")
done
```

4,0

Использование `mapfile` или `readarray` (которые являются синонимами):

```
mapfile -t arr < file
readarray -t arr < file
```

Прочитайте Массивы онлайн: https://riptutorial.com/ru/bash/topic/471/массивы

# глава 33: математический

## Examples

### Математика с использованием постоянного тока

`dc` является одним из старейших языков в Unix.

Он использует обратную полировку , означающую, что вы являетесь первым номером штабелирования, а затем выполняете операции. Например, `1+1` записывается как `1 1+` .

Чтобы напечатать элемент из верхней части стека, используйте команду `p`

```
echo '2 3 + p' | dc
5

or

dc <<< '2 3 + p'
5
```

Вы можете печатать верхний элемент много раз

```
dc <<< '1 1 + p 2 + p'
2
4
```

Для отрицательных чисел используйте префикс _

```
dc <<< '_1 p'
-1
```

Вы также можете использовать заглавные буквы от `A to F` для чисел от `10 and 15` и . как десятичная точка

```
dc <<< 'A.4 p'
10.4
```

`dc` использует произвольную точность, что означает, что точность ограничена только доступной памятью. По умолчанию точность установлена на 0 десятичных знаков

```
dc <<< '4 3 / p'
1
```

Мы можем увеличить точность с помощью команды `k` . `2k` будет использовать

```
dc <<< '2k 4 3 / p'
1.33

dc <<< '4k 4 3 / p'
1.3333
```

Вы также можете использовать его на нескольких строках

```
dc << EOF
1 1 +
3 *
p
EOF
6
```

`bc` - препроцессор для `dc` .

## Математика с использованием bc

`bc` - произвольный язык калькулятора точности. Его можно использовать в интерактивном режиме или выполнять из командной строки.

Например, он может распечатать результат выражения:

```
echo '2 + 3' | bc
5

echo '12 / 5' | bc
2
```

Для арифметики с плавающей точкой вы можете импортировать стандартную библиотеку `bc -l` :

```
echo '12 / 5' | bc -l
2.40000000000000000000
```

Его можно использовать для сравнения выражений:

```
echo '8 > 5' | bc
1

echo '10 == 11' | bc
0

echo '10 == 10 && 8 > 3' | bc
1
```

## Математика с использованием возможностей bash

Арифметическое вычисление также может быть выполнено без привлечения каких-либо других программ, таких как:

Умножение:

```
echo $((5 * 2))
10
```

Раздел:

```
echo $((5 / 2))
2
```

Модульное:

```
echo $((5 % 2))
1
```

Возведение:

```
echo $((5 ** 2))
25
```

## Математика, использующая expr

`expr` или `Evaluate expressions` оценивают выражение и записывают результат на стандартный вывод

Основная арифметика

```
expr 2 + 3
5
```

При умножении вам нужно избежать знака `*`

```
expr 2 \* 3
6
```

Вы также можете использовать переменные

```
a=2
expr $a + 3
5
```

Имейте в виду, что он поддерживает только целые числа, поэтому выражение, подобное этому

```
expr 3.0 / 2
```

**будет вызывать ошибку** `expr: not a decimal number: '3.0'` .

Он поддерживает регулярное выражение для соответствия шаблонам

```
expr 'Hello World' : 'Hell\(.*\)rld'
o Wo
```

Или найдите индекс первого символа в строке поиска

> Это заставит `expr: syntax error` в **Mac OS X** , потому что она использует **BSD expr,** которая не имеет команды индекса, в то время как expr в Linux обычно **GNU expr**

```
expr index hello l
3

expr index 'hello' 'lo'
3
```

Прочитайте математический онлайн: https://riptutorial.com/ru/bash/topic/2086/математический

# глава 34: Навигация по каталогам

## Examples

### Изменить на последний каталог

Для текущей оболочки это приведет вас к предыдущей директории, в которой вы были, независимо от того, где она была.

```
cd -
```

Выполняя его несколько раз эффективно «переключает», вы находитесь в текущем каталоге или в предыдущем.

### Переход в домашний каталог

Каталог по умолчанию - это домашний каталог ( `$HOME` , обычно `/home/username` ), поэтому `cd` без какой-либо директории отправляет вас туда

```
cd
```

Или вы можете быть более явным:

```
cd $HOME
```

Ярлык для домашнего каталога `~` , поэтому он также может быть использован.

```
cd ~
```

### Абсолютные и относительные каталоги

Чтобы перейти в абсолютно определенный каталог, используйте полное имя, начиная с обратного слэша `\` , таким образом:

```
cd /home/username/project/abc
```

Если вы хотите изменить каталог рядом с текущим, вы можете указать относительное местоположение. Например, если вы уже находитесь в `/home/username/project` , вы можете ввести подкаталог `abc` таким образом:

```
cd abc
```

Если вы хотите перейти в каталог выше текущего каталога, вы можете использовать

псевдоним `..` Например, если вы были в `/home/username/project/abc` и хотели перейти в `/home/username/project` , то вы сделали бы следующее:

```
cd ..
```

Это также можно назвать «вверх» в каталоге.

### Перейдите в каталог сценария

В общем, существует два типа **сценариев** Bash:

1. Системные инструменты, которые работают из текущего рабочего каталога
2. Инструменты проекта, которые изменяют файлы относительно их собственного места в файловой системе

Для второго типа скриптов полезно изменить каталог, в котором хранится скрипт. Это можно сделать с помощью следующей команды:

```
cd "$(dirname "$(readlink -f "$0")")"
```

Эта команда запускает 3 команды:

1. `readlink -f "$0"` определяет путь к текущему скрипту ( `$0` )
2. `dirname` преобразует путь к скрипту в путь к его каталогу
3. `cd` изменяет текущий рабочий каталог на каталог, который он получает от `dirname`

Прочитайте Навигация по каталогам онлайн: https://riptutorial.com/ru/bash/topic/6784/навигация-по-каталогам

# глава 35: найти

## Вступление

find - это команда для рекурсивного поиска каталога для файлов (или каталогов), соответствующих критериям, а затем выполнения некоторых действий над выбранными файлами.

find search_path selection_criteria action

## Синтаксис

- find [-H] [-L] [-P] [-D debugopts] [-Olevel] [путь ...] [выражение]

## Examples

### Поиск файла по имени или расширению

Чтобы найти файлы / каталоги с определенным именем, относительно `pwd` :

```
$ find . -name "myFile.txt"
./myFile.txt
```

Чтобы найти файлы / каталоги с определенным расширением, используйте подстановочный знак:

```
$ find . -name "*.txt"
./myFile.txt
./myFile2.txt
```

Чтобы найти файлы / каталоги, соответствующие одному из многих расширений, используйте флаг `or` :

```
$ find . -name "*.txt" -o -name "*.sh"
```

Чтобы найти файлы / каталоги, имена которых начинаются с abc и заканчиваются одним альфа-символом, следующим за одной цифрой:

```
$ find . -name "abc[a-z][0-9]"
```

Чтобы найти все файлы / каталоги, расположенные в определенном каталоге

```
$ find /opt
```

Чтобы искать только файлы (не каталоги), используйте `-type f` :

```
find /opt -type f
```

Чтобы искать только каталоги (не обычные файлы), используйте `-type d` :

```
find /opt -type d
```

## Поиск файлов по типу

Чтобы найти файлы, используйте флаг `-type f`

```
$ find . -type f
```

Чтобы найти каталоги, используйте флаг `-type d`

```
$ find . -type d
```

Чтобы найти блокирующие устройства, используйте флаг `-type b`

```
$ find /dev -type b
```

Чтобы найти символические ссылки, используйте флаг `-type l`

```
$ find . -type l
```

## Выполнение команд по найденному файлу

Иногда нам нужно запускать команды против большого количества файлов. Это можно сделать с помощью `xargs` .

```
find . -type d -print | xargs -r chmod 770
```

Вышеуказанная команда будет рекурсивно находить все каталоги ( `-type d` ) относительно `.` (который является вашим текущим рабочим каталогом) и выполнить `chmod 770` на них. Параметр `-r` указывает на `xargs` чтобы не запускать `chmod` если `find` не нашел файлов.

Если ваши имена файлов или каталоги имеют в них пробельный символ, эта команда может задохнуться; решение заключается в использовании следующих

```
find . -type d -print0 | xargs -r -0 chmod 770
```

В приведенном выше примере флаги `-print0` и `-0` указывают, что имена файлов будут разделены с использованием `null` байта и позволят использовать специальные символы, например пробелы, в именах файлов. Это расширение GNU и может не работать в других

версиях `find` и `xargs` .

---

Предпочтительный способ сделать это - пропустить команду `xargs` и позволить `find` вызов самого подпроцесса:

```
find . -type d -exec chmod 770 {} \;
```

Здесь `{}` является заполнителем, указывающим, что вы хотите использовать имя файла в этой точке. `find` будет выполнять `chmod` для каждого файла отдельно.

Вы также можете передавать все имена файлов на *один* вызов `chmod` , используя

```
find . -type d -exec chmod 770 {} +
```

Это также поведение приведенных выше фрагментов `xargs` . (Для вызова каждого файла по отдельности вы можете использовать `xargs -n1` ).

---

Третий вариант - позволить bash loop над списком имен файлов `find` выходы:

```
find . -type d | while read -r d; do chmod 770 "$d"; done
```

Это синтаксически самый неуклюжий, но удобный, когда вы хотите запускать несколько команд для каждого найденного файла. Однако это **небезопасно** перед именами файлов с нечетными именами.

```
find . -type f | while read -r d; do mv "$d" "${d// /_}"; done
```

который заменит все пробелы в именах файлов символами подчеркивания. (Этот пример также не будет работать, если в именах ведущих *каталогов* есть пробелы.)

Проблема с вышесказанным заключается в том, что в `while read -r` ожидает одну запись в строке, но имена файлов могут содержать символы новой строки (а также `read -r` будет потерять любое конечное пустое пространство). Вы можете исправить это, повернув все вокруг:

```
find . -type d -exec bash -c 'for f; do mv "$f" "${f// /_}"; done' _ {} +
```

Таким образом, `-exec` получает имена файлов в форме, которая полностью корректна и переносима; `bash -c` принимает их как число аргументов, которые будут найдены в `$@` , правильно цитируются и т. д. (сценарий должен обрабатывать эти имена правильно, конечно, каждая переменная, которая содержит имя файла, должна быть в двойном цитаты.)

Таинственный _ необходим, потому что первый аргумент `bash -c 'script'` используется для

заполнения `$0` .

## Поиск файла по времени доступа / модификации

В файловой системе `ext` каждый файл имеет сохраненное время доступа, модификации и (состояние), связанное с ним - для просмотра этой информации вы можете использовать `stat myFile.txt` ; используя флаги внутри *find* , мы можем искать файлы, которые были изменены в течение определенного временного диапазона.

Чтобы найти файлы, *которые* были изменены за последние 2 часа:

```
$ find . -mmin -120
```

Чтобы найти файлы, *которые не* были изменены за последние 2 часа:

```
$ find . -mmin +120
```

В приведенном выше примере являются поиск только по времени *изменения* - чтобы произвести поиск по ступа раза, или **c** повешенным раз, используйте или `a c` соответственно.

```
$ find . -amin -120
$ find . -cmin +120
```

Общий формат:

`-mmin n` : Файл был изменен *n* минут назад
`-mmin -n` : Файл был изменен менее чем за *n* минут назад
`-mmin +n` : Файл был изменен более чем *n* минут назад

---

Найдите файлы, *которые* были изменены за последние 2 дня:

```
find . -mtime -2
```

Поиск файлов, *которые не* были изменены за последние 2 дня

```
find . -mtime +2
```

Используйте `-atime` и `-ctime` для времени доступа и времени изменения статуса соответственно.

Общий формат:

`-mtime n` : Файл был изменен *nx24* часов назад
`-mtime -n` : Файл был изменен менее *nx24* часов назад

`-mtime +n` : Файл был изменен более *nx24* часов назад

Найти файлы, измененные в ***диапазоне дат*** , с 2007-06-07 по 2007-06-08:

```
find . -type f -newermt 2007-06-07 ! -newermt 2007-06-08
```

Найти файлы, доступные в ***диапазоне временных меток*** (используя файлы как временную метку), от 1 часа до 10 минут назад:

```
touch -t $(date -d '1 HOUR AGO' +%Y%m%d%H%M.%S) start_date
touch -t $(date -d '10 MINUTE AGO' +%Y%m%d%H%M.%S) end_date
timeout 10 find "$LOCAL_FOLDER" -newerat "start_date" ! -newerat "end_date" -print
```

Общий формат:

`-newerXY reference` : сравнивает `-newerXY reference` метку текущего файла со ссылкой. `XY` может иметь одно из следующих значений: `at` (время доступа), `mt` (время модификации), `ct` (время изменения) и т. Д. `reference` - это *имя файла,* для которого требуется сравнить указанную временную метку (доступ, изменение, изменение) или *строку,* описывающую абсолютное время.

## Поиск файлов по определенному расширению

Чтобы найти все файлы определенного расширения в текущем пути, вы можете использовать следующий синтаксис `find` . Он работает за счет использования в `bash's` встроенного `glob` конструкции , чтобы соответствовать все имена , имеющие `.extension` .

```
find /directory/to/search -maxdepth 1 -type f -name "*.extension"
```

Чтобы найти все файлы типа `.txt` из текущего каталога, выполните

```
find . -maxdepth 1 -type f -name "*.txt"
```

## Поиск файлов по размеру

*Найти файлы размером более 15 МБ:*

```
find -type f -size +15M
```

*Найдите файлы размером менее 12 КБ:*

```
find -type f -size -12k
```

*Поиск файлов размером от 12 КБ:*

```
find -type f -size 12k
```

Или же

```
find -type f -size 12288c
```

Или же

```
find -type f -size 24b
```

Или же

```
find -type f -size 24
```

**Общий формат:**

```
find [options] -size n[cwbkMG]
```

> Найти файлы размером n-блока, где + n означает больше n-блока, -n означает меньше n-блока и n (без знака) означает ровно n-блок

*Размер блока:*

1. `c` : байты
2. `w` : 2 байта
3. `b` : 512 байт (по умолчанию)
4. `k` : 1 КБ
5. `M` : 1 МБ
6. `G` : 1 ГБ

## Фильтрация пути

Параметр `-path` позволяет указать шаблон, соответствующий пути результата. Шаблон может совпадать и с самим именем.

Чтобы найти только файлы, содержащие `log` любом месте своего пути (папка или имя):

```
find . -type f -path '*log*'
```

Чтобы найти только файлы в папке с именем `log` (на любом уровне):

```
find . -type f -path '*/log/*'
```

Чтобы найти только файлы в папке с именем `log` или `data` :

```
find . -type f -path '*/log/*' -o -path '*/data/*'
```

---

Чтобы найти все файлы, **кроме** тех, которые содержатся в папке `bin` :

```
find . -type f -not -path '*/bin/*'
```

Чтобы найти все файлы всех файлов, **кроме** тех, которые содержатся в папке с именем `bin` или log files:

```
find . -type f -not -path '*log' -not -path '*/bin/*'
```

Прочитайте найти онлайн: https://riptutorial.com/ru/bash/topic/566/найти

# глава 36: Настройка PS1

## Examples

### Изменить приглашение PS1

Чтобы изменить PS1, вам просто нужно изменить значение переменной оболочки PS1. Значение может быть установлено в файле `~/.bashrc` или `/etc/bashrc` , в зависимости от дистрибутива. PS1 можно изменить на любой простой текст, например:

```
PS1="hello "
```

Помимо обычного текста поддерживается несколько специальных символов с обратным слэшем:

| Формат | действие |
|---|---|
| `\a` | символ колокола ASCII (07) |
| `\d` | дата в формате «День недели месяца» (например, «Вт май 26») |
| `\D{format}` | формат передается в strftime (3), и результат вставляется в строку приглашения; пустой формат приводит к представлению времени, специфичному для локали. Требуются скобки |
| `\e` | ASCII escape-символ (033) |
| `\h` | имя хоста до первого '.' |
| `\H` | имя хоста |
| `\j` | количество рабочих мест, которыми в настоящее время управляет оболочка |
| `\l` | базовое имя имени терминального устройства оболочки |
| `\n` | новая линия |
| `\r` | возврат каретки |
| `\s` | имя оболочки, базовое имя $ 0 (часть, следующая за последней косой чертой) |
| `\t` | текущее время в 24-часовом формате HH: MM: SS |

| Формат | действие |
| --- | --- |
| `\T` | текущее время в 12-часовом формате HH: MM: SS |
| `\@` | текущее время в 12-часовом формате am / pm |
| `\A` | текущее время в 24-часовом формате HH: MM |
| `\u` | имя пользователя текущего пользователя |
| `\v` | версия bash (например, 2.00) |
| `\V` | выпуск bash, версия + уровень патча (например, 2.00.0) |
| `\w` | текущий рабочий каталог, с $ HOME сокращенно с тильдой |
| `\W` | basename текущего рабочего каталога, с $ HOME сокращенно с тильдой |
| `\!` | номер истории этой команды |
| `\#` | номер команды этой команды |
| `\$` | если эффективный UID равен 0, а #, в противном случае $ |
| `\nnn*` | символ, соответствующий восьмеричному числу nnn |
| `\` | обратная косая черта |
| `\[` | начните последовательность непечатаемых символов, которые могут быть использованы для встраивания управляющей последовательности терминала в приглашение |
| `\]` | завершение последовательности непечатаемых символов |

Так, например, мы можем установить PS1 в:

```
PS1="\u@\h:\w\$ "
```

И он будет выводить:

    пользователь @ машина: ~ $

## Показать ветку git с помощью PROMPT_COMMAND

Если вы находитесь в папке git-репозитория, может быть приятно показать текущую ветку, в которой вы находитесь. В `~/.bashrc` или `/etc/bashrc` добавьте следующее (для этого требуется git):

```
function prompt_command {
    # Check if we are inside a git repository
    if git status > /dev/null 2>&1; then
        # Only get the name of the branch
        export GIT_STATUS=$(git status | grep 'On branch' | cut -b 10-)
    else
        export GIT_STATUS=""
    fi
}
# This function gets called every time PS1 is shown
PROMPT_COMMAND=prompt_command

PS1="\$GIT_STATUS \u@\h:\w\$ "
```

Если мы находимся в папке внутри репозитория git, это будет выводить:

    branch user @ machine: ~ $

И если мы находимся внутри обычной папки:

    пользователь @ машина: ~ $

## Показать имя ветки git в приглашении терминала

У вас могут быть функции в переменной PS1, просто убедитесь, что она одинарная, или используйте escape для специальных символов:

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}

PS1='\u@\h:\w$(gitPS1)$ '
```

Он даст вам следующее:

```
user@Host:/path (master)$
```

**Заметки:**

- Внесите изменения в файлы `~/.bashrc` или `/etc/bashrc` или `~/.bash_profile` или `~./profile` (в зависимости от ОС) и сохраните их.
- Запустите `source ~/.bashrc` (distro specific) после сохранения файла.

## Показывать время в командной строке терминала

```
timeNow(){
    echo "$(date +%r)"
}
PS1='[$(timeNow)] \u@\h:\w$ '
```

Он даст вам следующее:

```
[05:34:37 PM] user@Host:/path$
```

**Заметки:**

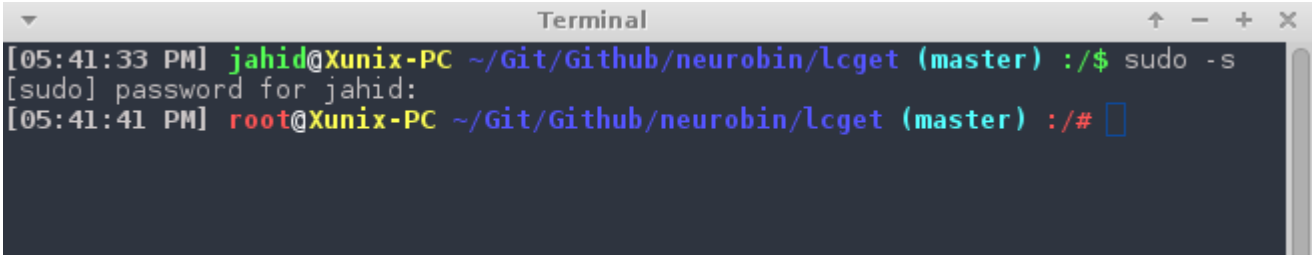
- Внесите изменения в файлы `~/.bashrc` или `/etc/bashrc` или `~/.bash_profile` или `~./profile` (в зависимости от ОС) и сохраните их.
- Запустите `source ~/.bashrc` (distro specific) после сохранения файла.

## Раскрасить и настроить подсказку терминала

Так автор устанавливает свою личную переменную `PS1` :

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}
#Please use the below function if you are a mac user
gitPS1ForMac(){
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
timeNow(){
    echo "$(date +%r)"
}
if [ "$color_prompt" = yes ]; then
  if [ x$EUID = x0 ]; then
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;31m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;31m\]:/#\[\033[00m\]
'
  else
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;32m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;32m\]:/$\[\033[00m\]
'
  fi
else
    PS1='[$(timeNow)] \u@\h \w$(gitPS1) :/$ '
fi
```

И вот как выглядит мое приглашение:




Ссылка цвета:

```
# Colors
txtblk='\e[0;30m' # Black - Regular
txtred='\e[0;31m' # Red
txtgrn='\e[0;32m' # Green
txtylw='\e[0;33m' # Yellow
txtblu='\e[0;34m' # Blue
txtpur='\e[0;35m' # Purple
txtcyn='\e[0;36m' # Cyan
txtwht='\e[0;37m' # White
bldblk='\e[1;30m' # Black - Bold
bldred='\e[1;31m' # Red
bldgrn='\e[1;32m' # Green
bldylw='\e[1;33m' # Yellow
bldblu='\e[1;34m' # Blue
bldpur='\e[1;35m' # Purple
bldcyn='\e[1;36m' # Cyan
bldwht='\e[1;37m' # White
unkblk='\e[4;30m' # Black - Underline
undred='\e[4;31m' # Red
undgrn='\e[4;32m' # Green
undylw='\e[4;33m' # Yellow
undblu='\e[4;34m' # Blue
undpur='\e[4;35m' # Purple
undcyn='\e[4;36m' # Cyan
undwht='\e[4;37m' # White
bakblk='\e[40m'   # Black - Background
bakred='\e[41m'   # Red
badgrn='\e[42m'   # Green
bakylw='\e[43m'   # Yellow
bakblu='\e[44m'   # Blue
bakpur='\e[45m'   # Purple
bakcyn='\e[46m'   # Cyan
bakwht='\e[47m'   # White
txtrst='\e[0m'    # Text Reset
```

**Заметки:**

- Внесите изменения в файлы `~/.bashrc` или `/etc/bashrc` или `~/.bash_profile` или `~./profile` (в зависимости от ОС) и сохраните их.
- Для `root` вам также может потребоваться отредактировать файл `/etc/bash.bashrc` или `/root/.bashrc`
- Запустите `source ~/.bashrc` (distro specific) после сохранения файла.
- Примечание: если вы сохранили изменения в `~/.bashrc` , не забудьте добавить `source ~/.bashrc` в свой файл `~/.bash_profile` чтобы это изменение в `PS1` записывалось каждый раз, когда запускается приложение терминала.

## Показать предыдущий статус и время возврата команды

Иногда нам нужен визуальный намек, указывающий статус возврата предыдущей команды. Следующий фрагмент делает его во главе PS1.

Обратите внимание, что функцию __stat () следует вызывать каждый раз, когда

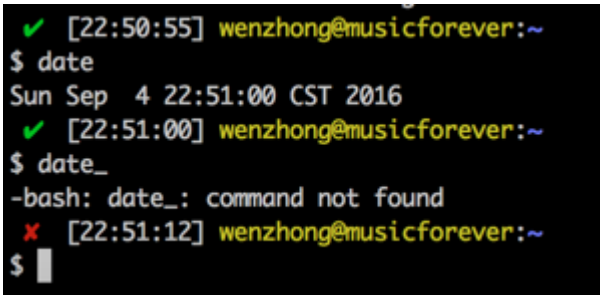
генерируется новый PS1, иначе он будет придерживаться статуса возврата последней команды вашего .bashrc или .bash_profile.

```
# -ANSI-COLOR-CODES- #
Color_Off="\033[0m"
###-Regular-###
Red="\033[0;31m"
Green="\033[0;32m"
Yellow="\033[0;33m"
####-Bold-####

function __stat() {
    if [ $? -eq 0 ]; then
        echo -en "$Green ✔ $Color_Off "
  else
    echo -en "$Red ✘ $Color_Off "
  fi
}

PS1='$(__stat)'
PS1+="[\t] "
PS1+="\e[0;33m\u@\h\e[0m:\e[1;34m\w\e[0m \n$ "

export PS1
```




Прочитайте Настройка PS1 онлайн: https://riptutorial.com/ru/bash/topic/3340/настройка-ps1

# глава 37: Обзорный

## Examples

### Динамическое определение масштаба в действии

Динамические обзорный означает , что переменные поиски происходят в объеме , когда функция *называется,* не там , где она *определена.*

```
$ x=3
$ func1 () { echo "in func1: $x"; }
$ func2 () { local x=9; func1; }
$ func2
in func1: 9
$ func1
in func1: 3
```

На лексическом языке `func1` *всегда* будет искать в глобальной области значение `x` , потому что `func1` *определяется* в локальной области.

На динамически ограниченном языке `func1` просматривается в области, где он *называется* . Когда он вызывается из `func2` , он сначала смотрит в тело `func2` на значение `x` . Если он не был определен там, он будет выглядеть в глобальной области, откуда был вызван `func2` .

Прочитайте Обзорный онлайн: https://riptutorial.com/ru/bash/topic/2452/обзорный

# глава 38: Обработка приглашения системы

## Синтаксис

- export PS1 = «something» # отображается, когда bash ожидает команду, которую нужно ввести в
- экспорт PS2 = "anotherthing" # dsplayed, когда оператор распространяется на большее количество строк
- export PS3 = «запрос вопроса для оператора выбора» # редко используется приглашение для выбора. Сначала установите PS3 в соответствии с вашими потребностями, затем **выберите select** . См. **Help select**
- экспорт PS4 = «в основном полезен для отладки, номер строки и т. д.» # используется для отладки сценариев bash.

## параметры

| Побег | подробности |
|---|---|
| \ a | Символ колокола. |
| \ d | Дата в формате «День недели» (например, «Вторник 26 мая»). |
| \ D {FORMAT} | FORMAT передается в `strftime '(3), и результат вставляется в строку приглашения; пустой FORMAT приводит к представлению времени, специфичному для локали. Требуются скобки. |
| \ e | Управляющий символ. \ 033 тоже конечно. |
| \час | Имя хоста, до первого `. '. (т. е. не доменная часть) |
| \ЧАС | Имя хоста в конечном итоге с доменной частью |
| \ J | Количество рабочих мест, которыми в настоящее время управляет оболочка. |
| \ л | Базовое имя имени терминала терминала. |
| \ п | Новая строка. |
| \p | Возврат кареты. |
| \ s | Имя оболочки, базовое имя `$ 0 '(часть, следующая за последней косой чертой). |

| Побег | подробности |
|---|---|
| \ т | Время в 24-часовом формате HH: MM: SS. |
| \ T | Время в 12-часовом формате HH: MM: SS. |
| @ | Время, в 12-часовом формате am / pm. |
| \ A | Время в 24-часовом формате HH: MM. |
| \ и | Имя пользователя текущего пользователя. |
| \ v | Версия Bash (например, 2.00) |
| \ V | Выпуск Bash, версия + patchlevel (например, 2.00.0) |
| \ ш | Текущая рабочая директория с $ HOME сокращена тильдой (используется переменная $ PROMPT_DIRTRIM). |
| \ W | Базисное имя $ PWD, с $ HOME сокращенно с тильдой. |
| ! | Номер истории этой команды. |
| # | Номер команды этой команды. |
| $ | Если эффективный uid равен 0, # , в противном случае **$** . |
| \ NNN | Символ, код ASCII которого является восьмеричным значением NNN. |
| \ | Обратная косая черта. |
| \ [ | Начните последовательность непечатаемых символов. Это можно использовать для встраивания управляющей последовательности терминала в приглашение. |
| \] | Завершите последовательность непечатаемых символов. |

# Examples

### Использование переменной PROMPT_COMMAND envrionment

Когда выполняется последняя команда в экземпляре интерактивного баха, отображается оценочная переменная PS1. Перед тем, как фактически показать PS1 bash, будет ли установлен PROMPT_COMMAND. Это значение этого var должно быть вызываемой программой или скриптом. Если этот var задан, эта программа / сценарий вызывается до того, как отобразится приглашение PS1.

```
# just a stupid function, we will use to demonstrate
# we check the date if Hour is 12 and Minute is lower than 59
lunchbreak(){
   if (( $(date +%H) == 12 && $(date +%M) < 59 )); then
      # and print colored \033[ starts the escape sequence
      # 5; is blinking attribute
      # 2; means bold
      # 31 says red
      printf "\033[5;1;31mmind the lunch break\033[0m\n";
   else
      printf "\033[33mstill working...\033[0m\n";
   fi;
}

# activating it
export PROMPT_COMMAND=lunchbreak
```

## Использование PS2

PS2 отображается, когда команда распространяется на несколько строк, и bash ожидает больше нажатий клавиш. Он также отображается, когда вводится составная команда, например **, ... do..done** и **аналогично** .

```
export PS2="would you please complete this command?\n"
# now enter a command extending to at least two lines to see PS2
```

## Использование PS3

Когда оператор select выполняется, он отображает данные, предваряемые номером, и затем отображает приглашение PS3:

```
export PS3="  To choose your language type the preceding number : "
select lang in EN CA FR DE; do
   # check input here until valid.
   break
done
```

## Использование PS4

PS4 отображается, когда bash находится в режиме отладки.

```
#!/usr/bin/env bash

# switch on debugging
set -x

# define a stupid_func
stupid_func(){
   echo I am line 1 of stupid_func
   echo I am line 2 of stupid_func
}
```

```
# setting the PS4 "DEBUG" prompt
export PS4='\nDEBUG level:$SHLVL subshell-level: $BASH_SUBSHELL \nsource-file:${BASH_SOURCE}
line#:${LINENO} function:${FUNCNAME[0]:+${FUNCNAME[0]}(): }\nstatement: '

# a normal statement
echo something

# function call
stupid_func

# a pipeline of commands running in a subshell
( ls -l | grep 'x' )
```

## Использование PS1

PS1 - это обычное системное приглашение, указывающее, что bash ожидает ввода команд. Он понимает некоторые escape-последовательности и может выполнять функции или прогаммы. Поскольку bash должен позиционировать курсор после подсказки displayes, он должен знать, как вычислить эффективную длину строки подсказки. Для указания непечатаемых последовательностей символов в переменной PS1 используются экранированные скобки: **\ [** *непечатная последовательность символов* **]** . Все сказанное верно для всех PS * vars.

(Черная каретка указывает курсор)

```
#everything not being an escape sequence will be literally printed
export PS1="literal sequence "  # Prompt is now:
literal sequence ▉

# \u == user \h == host \w == actual working directory
# mind the single quotes avoiding interpretation by shell
export PS1='\u@\h:\w > ' # \u == user, \h == host, \w actual working dir
looser@host:/some/path > ▉

# executing some commands within PS1
# following line will set foreground color to red, if user==root,
# else it resets attributes to default
# $( (($EUID == 0)) &&  tput setaf 1)
# later we do reset attributes to default with
# $(  tput sgr0 )
# assuming being root:
PS1="\[$( (($EUID == 0)) &&  tput setaf 1 \]\u\[$(tput sgr0)\]@\w:\w \$ "
looser@host:/some/path > ▉  # if not root else <red>root<default>@host....
```

Прочитайте Обработка приглашения системы онлайн:
https://riptutorial.com/ru/bash/topic/7541/обработка-приглашения-системы

# глава 39: отладка

## Examples

### Отладка сценария bash с помощью «-x»

Используйте «-x» для включения отладочного вывода выполненных строк. Его можно запускать на весь сеанс или сценарий или включать программно в сценарий.

Запустите сценарий с включенным отладочным выходом:

```
$ bash -x myscript.sh
```

Или же

```
$ bash --debug myscript.sh
```

Включите отладку в сценарии bash. Он может быть дополнительно включен, хотя отладочный вывод автоматически сбрасывается при выходе сценария.

```
#!/bin/bash
set -x    # Enable debugging
# some code here
set +x    # Disable debugging output.
```

### Проверка синтаксиса скрипта с «-n»

Флаг -n позволяет проверить синтаксис скрипта без его выполнения:

```
~> $ bash -n testscript.sh
testscript.sh: line 128: unexpected EOF while looking for matching `"'
testscript.sh: line 130: syntax error: unexpected end of file
```

### Отладка usigh bashdb

Bashdb - это утилита, которая похожа на gdb, поскольку вы можете делать такие вещи, как заданные контрольные точки в строке или в функции, печатать содержимое переменных, вы можете перезапустить выполнение сценария и многое другое.

Обычно вы можете установить его через менеджер пакетов, например, в Fedora:

```
sudo dnf install bashdb
```

Или получить его с главной страницы . Затем вы можете запустить его со своим скриптом

в качестве параметра:

```
bashdb <YOUR SCRIPT>
```

Вот несколько команд, которые помогут вам начать:

```
l - show local lines, press l again to scroll down
s - step to next line
print $VAR - echo out content of variable
restart - reruns bashscript, it re-loads it prior to execution.
eval - evaluate some custom command, ex: eval echo hi

b <line num> set breakpoint on some line
c - continue till some breakpoint
i b - info on break points
d <line #> - delete breakpoint at line #

shell - launch a sub-shell in the middle of execution, this is handy for manipulating
variables
```

Для получения дополнительной информации я рекомендую обратиться к руководству:
http://www.rodericksmith.plus.com/outlines/manuals/bashdbOutline.html

См. Также домашнюю страницу:
http://bashdb.sourceforge.net/

Прочитайте отладка онлайн: https://riptutorial.com/ru/bash/topic/3655/отладка

# глава 40: Параллельно

## Вступление

Работа в GNU Linux может быть распараллелирована с использованием GNU-параллелизма. Задание может быть одной командой или маленьким скриптом, который должен быть запущен для каждой из строк на входе. Типичным входом является список файлов, список хостов, список пользователей, список URL-адресов или список таблиц. Задание также может быть командой, которая считывает из трубы.

## Синтаксис

1. parallel [options] [команда [аргументы]] <list_of_arguments>

## параметры

| вариант | Описание |
|---|---|
| `-jn` | Запуск n заданий параллельно |
| `-k` | Сохраняйте тот же порядок |
| `-X` | Множество аргументов с заменой контекста |
| `--colsep regexp` | Разделить ввод на регулярное выражение для позиционных замещений |
| `{} {.} {/} {/.} {#}` | Запасные струны |
| `{3} {3.} {3/} {3/.}` | Позиционные строки замены |
| `-S sshlogin` | `Example: foo@server.example.com` |
| `--trc {}.bar` | Сокращение для --transfer --return {} .bar --cleanup |
| `--onall` | Запустите заданную команду с аргументом во всех sshlogins |
| `--nonall` | Запустите заданную команду без аргументов во всех sshlogins |
| `--pipe` | Сплит stdin (стандартный ввод) для нескольких заданий. |
| `--recend str` | Запишите концевой разделитель для --pipe. |
| `--recstart str` | Секунда начала записи для --pipe. |

# Examples

## Параллелизировать повторяющиеся задачи в списке файлов

Многие повторяющиеся задания могут выполняться более эффективно, если вы используете больше ресурсов своего компьютера (например, ЦП и ОЗУ). Ниже приведен пример параллельной работы нескольких заданий.

Предположим, что у вас есть `< list of files >` , например вывод из `ls` . Кроме того, пусть эти файлы сжаты bz2, и на них должен работать следующий порядок задач.

1. Декомпрессируйте файлы `bzcat` с помощью `bzcat` в stdout
2. Grep (например, фильтр) с определенным ключевым словом (-ами) с использованием `grep <some key word>`
3. Выполните вывод, который будет объединен в один файл `gzip` используя `gzip`

Выполнение этого с помощью цикла while может выглядеть так:

```
filenames="file_list.txt"
while read -r line
do
name="$line"
     ## grab lines with puppies in them
     bzcat $line | grep puppies | gzip >> output.gz
done < "$filenames"
```

Используя GNU Parallel, мы можем запускать сразу три параллельных задания, просто делая

```
parallel -j 3 "bzcat {} | grep puppies" ::: $( cat filelist.txt ) | gzip > output.gz
```

Эта команда проста, кратки и эффективна, когда количество файлов и размер файла являются большими. `-j 3` запускаются `parallel` , опция `-j 3` запускает 3 параллельных задания, а вход в параллельные задания берется в `:::` . Выход в конечном итоге подается на `gzip > output.gz`

## Параллелизировать STDIN

Теперь давайте представим, что у нас есть 1 большой файл (например, 30 ГБ), который нужно преобразовать, построчно. Скажем, у нас есть скрипт, `convert.sh` , который выполняет эту `<task>` . Мы можем передать содержимое этого файла в stdin для параллельного ввода и работы с такими *кусками* , как

```
<stdin> | parallel --pipe --block <block size> -k <task> > output.txt
```

где `<stdin>` может происходить из чего-либо, такого как `cat <file>` .

В качестве воспроизводимого примера наша задача будет `nl -n rz` . Возьмите любой файл, мой будет `data.bz2` и передайте его в `<stdin>`

```
bzcat data.bz2 | nl | parallel --pipe --block 10M -k nl -n rz | gzip > ouptput.gz
```

В приведенном выше примере используется `<stdin>` из `bzcat data.bz2 | nl` , где я включил `nl` как доказательство того, что окончательный выходной `output.gz` будет сохранен в том порядке, в котором он был получен. Затем `parallel` делит `<stdin>` на куски размером 10 МБ, и для каждого фрагмента он передает его через `nl -n rz` где он просто добавляет числа, правильно оправданные (подробнее см. `nl --help` ). Опции `--pipe parallel` split `<stdin>` на несколько заданий, а `-- block` определяет размер блоков. Опция `-k` указывает, что упорядочение должно поддерживаться.

Ваш конечный результат должен выглядеть примерно так:

```
000001       1  <data>
000002       2  <data>
000003       3  <data>
000004       4  <data>
000005       5  <data>
 ...
000587  552409  <data>
000588  552410  <data>
000589  552411  <data>
000590  552412  <data>
000591  552413  <data>
```

В моем исходном файле было 552 413 строк. Первый столбец представляет собой параллельные задания, а второй столбец представляет собой исходную нумерацию строк, которая была передана `parallel` в кусках. Вы должны заметить, что порядок во втором столбце (и остальной части файла) сохраняется.

Прочитайте Параллельно онлайн: https://riptutorial.com/ru/bash/topic/10778/параллельно

# глава 41: Передача файлов с помощью scp

## Синтаксис

- scp / some / local / directory / file_name имя_пользователя @ host_name: destination_file_path
- scp имя_пользователя @ имя_хоста: origin_file_path / some / local / directory

## Examples

### scp transfering file

Чтобы безопасно передать файл другому устройству - введите:

```
scp file1.txt tom@server2:$HOME
```

Этот пример представляет перенос `file1.txt` от нашего хозяина к `server2` «пользователю с `tom` домашней директории» s.

### scp передача нескольких файлов

`scp` также может использоваться для передачи нескольких файлов с одного сервера на другой. Ниже приведен пример переноса всех файлов из `my_folder` с расширением `.txt` на `server2` . В приведенном ниже примере все файлы будут переданы в домашний каталог пользователя `tom` .

```
scp /my_folder/*.txt tom@server2:$HOME
```

### Загрузка файла с помощью scp

Чтобы загрузить файл с удаленного сервера на локальный компьютер, введите:

```
scp tom@server2:$HOME/file.txt /local/machine/path/
```

В этом примере показано, как загрузить файл с именем `file.txt` из домашнего каталога пользователя `tom` в текущий каталог нашего локального компьютера.

Прочитайте Передача файлов с помощью scp онлайн:
https://riptutorial.com/ru/bash/topic/5484/передача-файлов-с-помощью-scp

# глава 42: Перенаправление

## Синтаксис

- команда </ path / to / file # Перенаправить стандартный ввод в файл
- command> / path / to / file # Перенаправить стандартный вывод на flie
- command file_descriptor> / path / to / file # Перенаправить вывод файла file_descriptor в файл
- command> & file_descriptor # Перенаправить вывод в file_descriptor
- command file_descriptor> & another_file_descriptor # Перенаправить file_descriptor в another_file_descriptor
- команда <& file_descriptor # Перенаправить file_descriptor в стандартный ввод
- command &> / path / to / file # Перенаправить стандартный вывод и стандартную ошибку в файл

## параметры

| параметр | подробности |
| --- | --- |
| внутренний файловый дескриптор | Целое число. |
| направление | Один из `>` , `<` или `<>` |
| внешний файловый дескриптор или путь | `&` за ним следует целое число для файлового дескриптора или пути. |

## замечания

Консольные программы UNIX имеют входной файл и два выходных файла (входные и выходные потоки, а также устройства) рассматриваются как файлы операционной системы.) Обычно это клавиатура и экран, но любой или все из них могут быть перенаправлены из - или перейти к файлу или другой программе.

`STDIN` является стандартным входом, и как программа получает интерактивный вход. `STDIN` обычно назначается файловым дескриптором 0.

`STDOUT` является стандартным выходом. Все, что испускается в `STDOUT` , считается «результатом» программы. `STDOUT` обычно назначается файловым дескриптором 1.

`STDERR` - это то, где отображаются сообщения об ошибках. Как правило, при запуске программы с консоли `STDERR` выводится на экран и неотличим от `STDOUT` . Обычно для `STDERR`

назначается файловый дескриптор 2.

**Порядок перенаправления важен**

```
command > file 2>&1
```

Перенаправляет ( `STDOUT` и `STDERR` ) в файл.

```
command 2>&1 > file
```

Перенаправляет только `STDOUT` , поскольку дескриптор файла 2 перенаправляется в файл, на который указывает файловый дескриптор 1 (который еще не является файловым `file` при оценке оператора).

Каждая команда в конвейере имеет свой собственный `STDERR` (и `STDOUT` ), потому что каждый из них является новым процессом. Это может создать неожиданные результаты, если вы ожидаете, что перенаправление повлияет на весь конвейер. Например, эта команда (завернутая для удобочитаемости):

```
$ python -c 'import sys;print >> sys.stderr, "Python error!"' \
| cut -f1 2>> error.log
```

будет печатать "Python error!" на консоль, а не на файл журнала. Вместо этого прикрепите ошибку к команде, которую вы хотите захватить:

```
$ python -c 'import sys;print >> sys.stderr, "Python error!"' 2>> error.log \
| cut -f1
```

# Examples

## Перенаправление стандартного вывода

`>` перенаправить стандартный вывод (иначе `STDOUT` ) текущей команды в файл или другой дескриптор.

Эти примеры записывают вывод команды `ls` в файл `file.txt`

```
ls >file.txt
> file.txt ls
```

Целевой файл создается, если он не существует, иначе этот файл усекается.

Дескриптор перенаправления по умолчанию - стандартный вывод или `1` если ни один не указан. Эта команда эквивалентна предыдущим примерам со стандартным выходом, явно указанным:

```
ls 1>file.txt
```

Примечание: перенаправление инициализируется исполняемой оболочкой, а не выполненной командой, поэтому она выполняется перед выполнением команды.

## Перенаправление STDIN

`<` читает его правый аргумент и записывает его левый аргумент.

Чтобы записать файл в `STDIN` мы должны *прочитать* `/tmp/a_file` и *записать* в `STDIN` т. `STDIN 0</tmp/a_file`

Примечание. Внутренний дескриптор файла по умолчанию равен `0` ( `STDIN` ) для `<`

```
$ echo "b" > /tmp/list.txt
$ echo "a" >> /tmp/list.txt
$ echo "c" >> /tmp/list.txt
$ sort < /tmp/list.txt
a
b
c
```

## Перенаправление STDOUT и STDERR

Файловые дескрипторы, такие как `0` и `1` являются указателями. Мы изменим, на что указывают дескрипторы файлов с перенаправлением. `>/dev/null` означает, что `1` указывает на `/dev/null` .

Сначала мы укажем `1` ( `STDOUT` ) на `/dev/null` затем на точку `2` ( `STDERR` ) на `1` пункт.

```
# STDERR is redirect to STDOUT: redirected to /dev/null,
# effectually redirecting both STDERR and STDOUT to /dev/null
echo 'hello' > /dev/null 2>&1
```

4,0

Это *может* быть дополнительно сокращено до следующего:

```
echo 'hello' &> /dev/null
```

Однако эта форма может быть нежелательной в производстве, если совместимость с оболочкой является проблемой, поскольку она конфликтует с POSIX, вводит разбор синтаксического разбора и оболочки без этой функции, будет неверно истолковывать ее:

```
# Actual code
echo 'hello' &> /dev/null
echo 'hello' &> /dev/null 'goodbye'

# Desired behavior
```

```
echo 'hello' > /dev/null 2>&1
echo 'hello' 'goodbye' > /dev/null 2>&1

# Actual behavior
echo 'hello' &
echo 'hello' & goodbye > /dev/null
```

ПРИМЕЧАНИЕ. `&>` , Как известно, работает как в Bash, так и Zsh.

## Перенаправление STDERR

`2` - `STDERR` .

```
$ echo_to_stderr 2>/dev/null # echos nothing
```

Определения:

`echo_to_stderr` - это команда, которая записывает `"stderr"` в `STDERR`

```
echo_to_stderr () {
    echo stderr >&2
}

$ echo_to_stderr
stderr
```

## Добавить vs Truncate

## Усекать `>`

1. Создайте указанный файл, если он не существует.
2. Truncate (удалить содержимое файла)
3. Написать в файл

```
$ echo "first line" > /tmp/lines
$ echo "second line" > /tmp/lines

$ cat /tmp/lines
second line
```

## Добавить `>>`

1. Создайте указанный файл, если он не существует.
2. Добавить файл (запись в конце файла).

```
# Overwrite existing file
$ echo "first line" > /tmp/lines

# Append a second line
```

```
$ echo "second line" >> /tmp/lines

$ cat /tmp/lines
first line
second line
```

## STDIN, STDOUT и STDERR объяснили

Команды имеют один вход (STDIN) и два вида выходов, стандартный вывод (STDOUT) и стандартную ошибку (STDERR).

Например:

**STDIN**

```
root@server~# read
Type some text here
```

Стандартный ввод используется для ввода ввода в программу. (Здесь мы используем для `read` предопределённого для чтения строки из STDIN.)

**STDOUT**

```
root@server~# ls file
file
```

Стандартный вывод обычно используется для «нормального» вывода из команды. Например, `ls` перечисляет файлы, поэтому файлы отправляются в STDOUT.

**STDERR**

```
root@server~# ls anotherfile
ls: cannot access 'anotherfile': No such file or directory
```

Стандартная ошибка (как следует из названия) используется для сообщений об ошибках. Поскольку это сообщение не является списком файлов, оно отправляется в STDERR.

STDIN, STDOUT и STDERR являются тремя *стандартными потоками.* Они идентифицируются с оболочкой числом, а не именем:

0 = Стандарт в
1 = стандартный
2 = стандартная ошибка

По умолчанию STDIN подключается к клавиатуре, и на терминале появляются STDOUT и STDERR. Однако мы можем перенаправить STDOUT или STDERR на все, что нам нужно. Например, допустим, что вам нужен только стандарт, и все сообщения об ошибках, напечатанные на стандартной ошибке, должны быть подавлены. Именно тогда мы

используем дескрипторы `1` и `2` .

**Перенаправление STDERR в / dev / null**
Взяв предыдущий пример,

```
root@server~# ls anotherfile 2>/dev/null
root@server~#
```

В этом случае, если есть какой-либо STDERR, он будет перенаправлен на / dev / null ( специальный файл, который игнорирует что-либо в нем), поэтому вы не получите никаких ошибок в оболочке.

## Перенаправление нескольких команд в один и тот же файл

```
{
  echo "contents of home directory"
  ls ~
} > output.txt
```

## Использование именованных каналов

Иногда вы можете что-то выводить по одной программе и вводить ее в другую программу, но не можете использовать стандартный канал.

```
ls -l | grep ".log"
```

Вы можете просто записать во временный файл:

```
touch tempFile.txt
ls -l > tempFile.txt
grep ".log" < tempFile.txt
```

Это отлично `tempFile` для большинства приложений, однако никто не будет знать, что делает `tempFile` , и кто-то может удалить его, если он содержит вывод `ls -l` в этом каталоге. Здесь вызывается именованная труба:

```
mkfifo myPipe
ls -l > myPipe
grep ".log" < myPipe
```

`myPipe` - это технически файл (все в Linux), поэтому давайте сделаем `ls -l` в пустой директории, в которой мы только что создали канал:

```
mkdir pipeFolder
cd pipeFolder
mkfifo myPipe
ls -l
```

Выход:

```
prw-r--r-- 1 root root 0 Jul 25 11:20 myPipe
```

Обратите внимание на первый символ в разрешениях, он указан как канал, а не файл.

Теперь давайте сделаем что-нибудь классное.

Откройте один терминал и обратите внимание на каталог (или создайте его так, чтобы очистка была простой), и создайте канал.

```
mkfifo myPipe
```

Теперь давайте поместим что-то в трубку.

```
echo "Hello from the other side" > myPipe
```

Вы заметите, что это зависает, другая сторона трубы все еще закрыта. Давайте откроем другую сторону трубы и пропустим это.

Откройте другой терминал и перейдите в каталог, в котором находится труба (или, если вы знаете, добавьте его в трубу):

```
cat < myPipe
```

Вы заметите, что после получения `hello from the other side` программа в первом терминале заканчивается, как и во втором терминале.

Теперь запустите команды в обратном порядке. Начните с `cat < myPipe` а затем повторите что-нибудь в нем. Он по-прежнему работает, потому что программа будет ждать, пока что-то не будет помещено в трубку до завершения, потому что он знает, что он должен что-то получить.

Именованные каналы могут быть полезны для перемещения информации между терминалами или между программами.

Трубы маленькие. После полной записи автор блокирует, пока какой-либо читатель не прочитает содержимое, поэтому вам нужно либо запустить считыватель и запись на разных терминалах, либо запустить один или другой в фоновом режиме:

```
ls -l /tmp > myPipe &
cat < myPipe
```

**Дополнительные примеры с использованием именованных каналов:**

- Пример 1 - все команды на одном и том же терминале / той же оболочке

```
$ { ls -l && cat file3; } >mypipe &
$ cat <mypipe
# Output: Prints ls -l data and then prints file3 contents on screen
```

- Пример 2 - все команды на одном и том же терминале / той же оболочке

```
$ ls -l >mypipe &
$ cat file3 >mypipe &
$ cat <mypipe
#Output: This prints on screen the contents of mypipe.
```

Имейте в виду, что первое содержимое `file3` отображается, а затем отображаются `ls -l` данные (конфигурация LIFO).

- Пример 3 - все команды на одном и том же терминале / той же оболочке

```
$ { pipedata=$(<mypipe) && echo "$pipedata"; } &
$ ls >mypipe
# Output: Prints the output of ls directly on screen
```

Имейте в виду, что переменная `$pipedata` недоступна для использования в главном терминале / основной оболочке, так как использование `&` вызывает подоболочку, а `$pipedata` доступно только в этой подоболочке.

- Пример 4 - все команды на одном и том же терминале / той же оболочке

```
$ export pipedata
$ pipedata=$(<mypipe) &
$ ls -l *.sh >mypipe
$ echo "$pipedata"
#Output : Prints correctly the contents of mypipe
```

Это правильно печатает значение переменной `$pipedata` в основной оболочке из-за объявления экспорта переменной. Основной терминал / основная оболочка не висит из-за вызова фоновой оболочки ( `&` ).

## Печатать сообщения об ошибках в stderr

Сообщения об ошибках обычно включаются в сценарий для целей отладки или для обеспечения богатого пользовательского опыта. Просто напишите сообщение об ошибке следующим образом:

```
cmd || echo 'cmd failed'
```

может работать для простых случаев, но это не обычный способ. В этом примере сообщение об ошибке будет загрязнять фактический вывод сценария путем смешивания как ошибок, так и успешного вывода в `stdout` .

Короче говоря, сообщение об ошибке должно идти в `stderr` не `stdout` . Это довольно просто:

```
cmd || echo 'cmd failed' >/dev/stderr
```

Другой пример:

```
if cmd; then
    echo 'success'
else
    echo 'cmd failed' >/dev/stderr
fi
```

В приведенном выше примере сообщение об успешном завершении будет напечатано на `stdout` пока сообщение об ошибке будет напечатано на `stderr` .

Лучшим способом печати сообщения об ошибке является определение функции:

```
err(){
    echo "E: $*" >>/dev/stderr
}
```

Теперь, когда вам нужно напечатать сообщение об ошибке:

```
err "My error message"
```

## Перенаправление на сетевые адреса

2,04

Bash рассматривает некоторые пути как специальные и может выполнять некоторую сетевую связь, записывая в `/dev/{udp|tcp}/host/port` . Bash не может настроить сервер прослушивания, но может инициировать соединение, а TCP может читать результаты как минимум.

Например, чтобы отправить простой веб-запрос, который можно было бы сделать:

```
exec 3</dev/tcp/www.google.com/80
printf 'GET / HTTP/1.0\r\n\r\n' >&3
cat <&3
```

и результаты веб-страницы `www.google.com` умолчанию будут напечатаны на стандартный `stdout` .

так же

```
printf 'HI\n' >/dev/udp/192.168.1.1/6666
```

отправит сообщение UDP, содержащее `HI\n` слушателю на `192.168.1.1:6666`

Прочитайте Перенаправление онлайн: https://riptutorial.com/ru/bash/topic/399/перенаправление

# глава 43: Последовательность выполнения файла

## Вступление

`.bash_profile` , `.bash_login` , `.bashrc` и `.profile` все делают почти то же самое: настраивают и определяют функции, переменные и сортировки.

Основное отличие состоит в том, что `.bashrc` вызывается при открытии не-login, но интерактивного окна, а `.bash_profile` а остальные вызываются для оболочки входа. Многие люди имеют свой `.bash_profile` или аналогичный вызов `.bashrc` любом случае.

## замечания

Другие примечания:

- `/etc/profile` для системного (не пользовательского) кода инициализации.
- `.bash_logout` , запускается при выходе из системы (думаю, что `.bash_logout` )
- `.inputrc` , аналогично `.bashrc` но для readline.

## Examples

### .profile vs .bash_profile (и .bash_login)

`.profile` читается большинством оболочек при запуске, включая bash. Однако `.bash_profile` используется для конфигураций, специфичных для bash. Для общего кода инициализации поместите его в `.profile` . Если это специфично для bash, используйте `.bash_profile` .

`.profile` самом деле не предназначен специально для bash, но вместо этого `.bash_profile` . ( `.profile` для Bourne и других подобных оболочек, bash основан на) Bash будет возвращаться к `.profile` если `.bash_profile` не найден.

`.bash_login` - это `.bash_login` для `.bash_profile` , если он не найден. Обычно лучше использовать `.bash_profile` или `.profile` .

Прочитайте Последовательность выполнения файла онлайн:
https://riptutorial.com/ru/bash/topic/8626/последовательность-выполнения-файла

# глава 44: Программируемое завершение

## Examples

### Простое завершение с использованием функции

```
_mycompletion() {
    local command_name="$1" # not used in this example
    local current_word="$2"
    local previous_word="$3" # not used in this example
    # COMPREPLY is an array which has to be filled with the possible completions
    # compgen is used to filter matching completions
    COMPREPLY=( $(compgen -W 'hello world' -- "$current_word") )
}
complete -F _mycompletion mycommand
```

Пример использования:

```
$ mycommand [TAB][TAB]
hello world
$ mycommand h[TAB][TAB]
$ mycommand hello
```

### Простое заполнение опций и имен файлов

```
# The following shell function will be used to generate completions for
# the "nuance_tune" command.
_nuance_tune_opts ()
{
  local curr_arg prev_arg
  curr_arg=${COMP_WORDS[COMP_CWORD]}
  prev_arg=${COMP_WORDS[COMP_CWORD-1]}

  # The "config" option takes a file arg, so get a list of the files in the
  # current dir.  A case statement is probably unnecessary here, but leaves
  # room to customize the parameters for other flags.
  case "$prev_arg" in
    -config)
      COMPREPLY=( $( /bin/ls -1 ) )
      return 0
      ;;
    esac

    # Use compgen to provide completions for all known options.
    COMPREPLY=( $(compgen -W '-analyze -experiment -generate_groups -compute_thresh -config -
output -help -usage -force -lang -grammar_overrides -begin_date -end_date -group -dataset -
multiparses -dump_records -no_index -confidencelevel -nrecs -dry_run -rec_scripts_only -
save_temp -full_trc -single_session -verbose -ep -unsupervised -write_manifest -remap -
noreparse -upload -reference -target -use_only_matching -histogram -stepsize' -- $curr_arg )
);
}

# The -o parameter tells Bash to process completions as filenames, where applicable.
```

```
complete -o filenames -F _nuance_tune_opts nuance_tune
```

Прочитайте Программируемое завершение онлайн: https://riptutorial.com/ru/bash/topic/3162/программируемое-завершение

# глава 45: Пространство имен

## Examples

### Нет таких вещей, как пространства имен

```
myfunc(){
    echo "I will never be executed."
}
another_func(){
    # this "redeclare" overwrites original function
    myfunc(){ echo "I am the one and only"; }
}
# myfunc will print "I will never be executed"
myfunc
# but if we call another_func first
another_func
# it gets overwritten and
myfunc
# no prints "I am the one and only"
```

Побеждает последняя декларация. Нет таких вещей, как пространства имен! Однако функции могут содержать другие функции.

Прочитайте Пространство имен онлайн: https://riptutorial.com/ru/bash/topic/6835/пространство-имен

# глава 46: Прочитать файл (поток данных, переменную) по очереди (и / или поле за полем)?

## параметры

| параметр | подробности |
|---|---|
| IFS | Внутренний разделитель полей |
| файл | Имя файла / путь |
| `-r` | Запрещает интерпретацию обратной косой черты при использовании с чтением |
| `-t` | Удаляет `readarray` новую строку из каждой строки, считанной `readarray` |
| `-d DELIM` | Продолжайте, пока не будет прочитан первый символ DELIM (с `read` ), а не символ новой строки |

## Examples

### Считывает файл (/ etc / passwd) по строкам и по полям

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS=: read -r username password userid groupid comment homedir cmdshell
do
  echo "$username, $userid, $comment $homedir"
done < $FILENAME
```

В файле unix password информация пользователя хранится по очереди, каждая строка состоит из информации для пользователя, разделенного символом двоеточия (:). В этом примере при чтении файла строки за строкой строка также разделяется на поля с использованием символа двоеточия в качестве разделителя, который обозначается значением, заданным для IFS.

**Пример ввода**

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
```

```
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

**Образец вывода**

```
mysql, 27, MySQL Server /var/lib/mysql
pulse, 497, PulseAudio System Daemon /var/run/pulse
sshd, 74, Privilege-separated SSH /var/empty/sshd
tomcat, 91, Apache Tomcat /usr/share/tomcat6
webalizer, 67, Webalizer /var/www/usage
```

Чтобы читать строки за строкой и иметь всю строку, назначенную переменной, ниже приведена модифицированная версия примера. Обратите внимание, что у нас есть только одна переменная по названию.

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS= read -r line
do
  echo "$line"
done < $FILENAME
```

**Пример ввода**

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

**Образец вывода**

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

## Чтение строк файла в массив

```
readarray -t arr <file
```

Или с петлей:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <file
```

## Зацикливание через файл по строкам

```
while IFS= read -r line; do
   echo "$line"
done <file
```

Если файл не может содержать новую строку в конце, тогда:

```
while IFS= read -r line || [ -n "$line" ]; do
   echo "$line"
done <file
```

## Чтение строк строки в массив

```
var='line 1
line 2
line3'
readarray -t arr <<< "$var"
```

или с петлей:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <<< "$var"
```

## Зацикливание строк по строкам

```
var='line 1
line 2
line3'
while IFS= read -r line; do
   echo "-$line-"
done <<< "$var"
```

или же

```
readarray -t arr <<< "$var"
for i in "${arr[@]}";do
    echo "-$i-"
done
```

## Зацикливание через вывод командной строки за строкой

```
while IFS= read -r line;do
    echo "**$line**"
done < <(ping google.com)
```

или с трубой:

```
ping google.com |
```

```
while IFS= read -r line;do
    echo "**$line**"
done
```

## Чтение поля файла по полю

Предположим, что разделитель полей : (двоеточие) в файле *файла* .

```
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "$field"
done <file
```

Для контента:

```
first : se
con
d:
    Thi rd:
    Fourth
```

Выход:

```
**first **
** se
con
d**
**
    Thi rd**
**
    Fourth
**
```

## Чтение поля строки по полю

Предположим, что разделитель полей :

```
var='line: 1
line: 2
line3'
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "-$field-"
done <<< "$var"
```

Выход:

```
-line-
- 1
line-
- 2
line3
-
```

## Чтение полей файла в массиве

Предположим, что разделитель полей :

```
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <file
```

## Чтение полей строки в массив

Предположим, что разделитель полей :

```
var='1:2:3:4:
newline'
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <<< "$var"
echo "${arr[4]}"
```

Выход:

```
newline
```

## Зацикливание через вывод поля команды по полю

Предположим, что разделитель полей :

```
while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done < <(ping google.com)
```

Или с трубой:

```
ping google.com | while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done
```

Прочитайте Прочитать файл (поток данных, переменную) по очереди (и / или поле за полем)? онлайн: https://riptutorial.com/ru/bash/topic/5473/прочитать-файл--поток-данных--переменную--по-очереди--и---или-поле-за-полем--

# глава 47: Работа в определенное время

## Examples

### Выполнять задание один раз в определенное время

*Примечание:* **at** не устанавливается по умолчанию в большинстве современных дистрибутивов.

Чтобы выполнить задание один раз в другое время, чем сейчас, в этом примере 17:00 вы можете использовать

```
echo "somecommand &" | at 5pm
```

Если вы хотите поймать вывод, вы можете сделать это обычным способом:

```
echo "somecommand > out.txt 2>err.txt &" | at 5pm
```

`at` понимании многих временных форматов, поэтому вы также можете сказать

```
echo "somecommand &" | at now + 2 minutes
echo "somecommand &" | at 17:00
echo "somecommand &" | at 17:00 Jul 7
echo "somecommand &" | at 4pm 12.03.17
```

Если год или дата не указаны, он принимает следующий раз, когда будет указано указанное вами время. Поэтому, если вы дадите час, который уже прошел сегодня, это займет завтра, и если вы дадите месяц, который уже прошел в этом году, он будет принят в следующем году.

Это также работает вместе с nohup, как вы ожидали.

```
echo "nohup somecommand > out.txt 2>err.txt &" | at 5pm
```

Есть еще несколько команд для управления заданиями по времени:

- **atq** перечисляет все заданные по времени задания ( **atq** ueue)
- **atrm** удаляет **заданное** время ( **atr** e **m** ove)
- **пакет** работает в основном так же, как и у, но выполняет задания только тогда, когда загрузка системы ниже 0,8

Все команды применяются к заданиям пользователя, вошедшего в систему. Если вы вошли в систему с правами администратора, все системные задания обрабатываются, конечно.

## Повторное выполнение заданий в указанное время с помощью systemd.timer

**systemd** обеспечивает современную реализацию **cron** . Для выполнения периодического сценария необходима служба и файл таймера. Файлы службы и таймера должны быть помещены в / etc / systemd / {system, user}. Файл службы:

```
[Unit]
Description=my script or programm does the very best and this is the description

[Service]
# type is important!
Type=simple
# program|script to call. Always use absolute pathes
# and redirect STDIN and STDERR as there is no terminal while being executed
ExecStart=/absolute/path/to/someCommand >>/path/to/output 2>/path/to/STDERRoutput
#NO install section!!!! Is handled by the timer facitlities itself.
#[Install]
#WantedBy=multi-user.target
```

Затем файл таймера:

```
[Unit]
Description=my very first systemd timer
[Timer]
# Syntax for date/time specifications is  Y-m-d H:M:S
# a * means "each", and a comma separated list of items can be given too
# *-*-* *,15,30,45:00  says every year, every month, every day, each hour,
# at minute 15,30,45 and zero seconds

OnCalendar=*-*-* *:01:00
# this one runs each hour at one minute zero second e.g. 13:01:00
```

Прочитайте Работа в определенное время онлайн: https://riptutorial.com/ru/bash/topic/7283/работа-в-определенное-время

# глава 48: Работа и процессы

## Examples

### Список текущих заданий

```
$ tail -f /var/log/syslog > log.txt
[1]+  Stopped                 tail -f /var/log/syslog > log.txt

$ sleep 10 &

$ jobs
[1]+  Stopped                 tail -f /var/log/syslog > log.txt
[2]-  Running                 sleep 10 &
```

### Обработка работы

# Создание рабочих мест

Чтобы создать задание, просто добавьте один `&` после команды:

```
$ sleep 10 &
[1] 20024
```

Вы также можете сделать выполняемый процесс заданием, нажав `Ctrl` + `z` :

```
$ sleep 10
^Z
[1]+  Stopped                 sleep 10
```

# Фоновый и передний план процесса

Чтобы перенести процесс на передний план, команда `fg` используется вместе с `%`

```
$ sleep 10 &
[1] 20024

$ fg %1
sleep 10
```

Теперь вы можете взаимодействовать с процессом. Чтобы вернуть его на задний план, вы можете использовать команду `bg` . Из-за занятой сессии терминала вам нужно сначала остановить процесс, нажав `Ctrl` + `z` .

```
$ sleep 10
^Z
[1]+  Stopped                 sleep 10

$ bg %1
[1]+ sleep 10 &
```

Из-за лени некоторых программистов все эти команды также работают с одним `%` если есть только один процесс или первый процесс в списке. Например:

```
$ sleep 10 &
[1] 20024

$ fg %        # to bring a process to foreground 'fg %' is also working.
sleep 10
```

или просто

```
$ %           # laziness knows no boundaries, '%' is also working.
sleep 10
```

Кроме того, просто набрав `fg` или `bg` без какого-либо аргумента, обработает последнее задание:

```
$ sleep 20 &
$ sleep 10 &
$ fg
sleep 10
^C
$ fg
sleep 20
```

# Убийство рабочих мест

```
$ sleep 10 &
[1] 20024

$ kill %1
[1]+  Terminated              sleep 10
```

Процесс сна выполняется в фоновом режиме с идентификатором процесса (pid) `20024` и номером задания `1` . Чтобы ссылаться на этот процесс, вы можете использовать либо pid, либо номер задания. Если вы используете номер задания, вы должны указать его `%` . Сигнал уничтожения по умолчанию, отправленный `kill` - это `SIGTERM` , который позволяет целевому процессу выйти изящно.

Ниже приведены некоторые общие сигналы об уничтожении. Чтобы просмотреть полный список, запустите `kill -l` .

| Название сигнала | Значение сигнала | эффект |
|---|---|---|
| SIGHUP | 1 | Вешать трубку |
| SIGINT | 2 | Прерывание с клавиатуры |
| SIGKILL | 9 | Убить сигнал |
| SIGTERM | 15 | Сигнал о завершении |

# Запуск и уничтожение определенных процессов

Вероятно, самый простой способ убить `pkill` процесс - это выбрать его через имя процесса, как в следующем примере, используя команду `pkill` как

```
pkill -f test.py
```

(или) более безопасный способ, используя `pgrep` для поиска фактического идентификатора процесса

```
kill $(pgrep -f 'python test.py')
```

Тот же результат можно получить, используя `grep` над `ps -ef | grep name_of_process` затем `ps -ef | grep name_of_process` процесс, связанный с результатом pid (id процесса). Выбор процесса с использованием его имени удобен в тестовой среде, но может быть действительно опасным, когда сценарий используется в производстве: практически невозможно быть уверенным, что имя будет соответствовать процессу, который вы на самом деле хотите убить. В этих случаях на самом деле такой подход безопасен.

Запустите скрипт, который в конечном итоге будет убит с помощью следующего подхода. Предположим, что команда, которую вы хотите выполнить и в конечном итоге убить, это `python test.py`

```
#!/bin/bash

if [[ ! -e /tmp/test.py.pid ]]; then   # Check if the file already exists
    python test.py &                   #+and if so do not run another process.
    echo $! > /tmp/test.py.pid
else
    echo -n "ERROR: The process is already running with pid "
    cat /tmp/test.py.pid
    echo
fi
```

Это создаст файл в каталоге `/tmp` содержащий pid процесса `python test.py` Если файл уже

существует, мы предполагаем, что команда уже запущена, и скрипт возвращает ошибку.

Затем, когда вы хотите его убить, используйте следующий скрипт:

```
#!/bin/bash

if [[ -e /tmp/test.py.pid ]]; then   # If the file do not exists, then the
    kill `cat /tmp/test.py.pid`      #+the process is not running. Useless
    rm /tmp/test.py.pid              #+trying to kill it.
else
    echo "test.py is not running"
fi
```

который будет точно уничтожать процесс, связанный с вашей командой, не полагаясь на любую изменчивую информацию (например, строку, используемую для запуска команды). Даже в этом случае, если файл не существует, сценарий предполагает, что вы хотите убить неиспользуемый процесс.

Этот последний пример может быть легко улучшен для выполнения одной и той же команды несколько раз (например, для добавления файла pid вместо его перезаписи) и для управления случаями, когда процесс умирает до его уничтожения.

## Список всех процессов

Существует два распространенных способа перечислить все процессы в системе. Оба перечисляют все процессы, выполняемые всеми пользователями, хотя они отличаются в том формате, который они выдают (причина различий историческая).

```
ps -ef   # lists all processes
ps aux   # lists all processes in alternative format (BSD)
```

Это можно использовать, чтобы проверить, работает ли данное приложение. Например, чтобы проверить, запущен ли SSH-сервер (sshd):

```
ps -ef | grep sshd
```

## Проверьте, какой процесс выполняется на определенном порту

Чтобы проверить, какой процесс выполняется на порту 8080

```
lsof -i :8080
```

## Поиск информации о запущенном процессе

`ps aux | grep <search-term>` показывает процессы, соответствующие *поисковому термину*

Пример:

```
root@server7:~# ps aux | grep nginx
root       315  0.0  0.3 144392  1020 ?        Ss   May28   0:00 nginx: master process
/usr/sbin/nginx
www-data  5647  0.0  1.1 145124  3048 ?        S    Jul18   2:53 nginx: worker process
www-data  5648  0.0  0.1 144392   376 ?        S    Jul18   0:00 nginx: cache manager process
root     13134  0.0  0.3   4960   920 pts/0    S+   14:33   0:00 grep --color=auto nginx
root@server7:~#
```

Здесь второй столбец - это идентификатор процесса. Например, если вы хотите убить процесс nginx, вы можете использовать команду `kill 5647` . Всегда рекомендуется использовать команду `kill` с `SIGTERM` а не `SIGKILL` .

## Отключение фоновой работы

```
$ gzip extremelylargefile.txt &
$ bg
$ disown %1
```

Это позволяет продолжить процесс, как только ваша оболочка (терминал, ssh и т. Д.) Будет закрыта.

Прочитайте Работа и процессы онлайн: https://riptutorial.com/ru/bash/topic/398/работа-и-процессы

# глава 49: Разделение слов

## Синтаксис

- Установите IFS в новую строку: IFS = $ '\ n'
- Установите IFS на nullstring: IFS =
- Установите IFS в / символ: IFS = /

## параметры

| параметр | подробности |
|---|---|
| IFS | Внутренний разделитель полей |
| -Икс | Команды печати и их аргументы по мере их выполнения (опция Shell) |

## замечания

- Разделение слов не выполняется во время присвоений, например `newvar=$var`
- Разделение слов не выполняется в конструкции `[[ ... ]]`
- Используйте двойные кавычки для переменных, чтобы предотвратить разделение слов

## Examples

### Разделение с IFS

Чтобы быть более понятным, давайте создадим скрипт с именем `showarg` :

```
#!/usr/bin/env bash
printf "%d args:" $#
printf " <%s>" "$@"
echo
```

Теперь посмотрим различия:

```
$ var="This is an example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

> `$var` делится на 4 аргумента. `IFS` - это символы пробела и, следовательно, разбиение слов происходит в пробелах

```
$ var="This/is/an/example"
$ showarg $var
1 args: <This/is/an/example>
```

В предыдущем слове расщепление не произошло, потому что символы `IFS` не были найдены.

Теперь давайте установим `IFS=/`

```
$ IFS=/
$ var="This/is/an/example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

`$var` делится на 4 аргумента, а не один аргумент.

## Что, когда и почему?

Когда оболочка выполняет *расширение параметра* , *подмену команды* , *переменное или арифметическое расширение* , он сканирует границы слова в результате. Если какая-либо граница слова найдена, результат разбивается на несколько слов в этой позиции. Граница слова определяется переменной оболочки `IFS` (Internal Field Separator). Значением по умолчанию для IFS являются пробел, табуляция и новая строка, то есть разделение слов будет происходить на этих трех символах пробела, если это не будет предотвращено явно.

```
set -x
var='I am
a
multiline string'
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

В приведенном выше примере это то, как выполняется функция `fun` :

```
fun I am a multiline string
```

`$var` делится на 5 аргументов, будет напечатан только `I` , `am` и `a` .

## IFS & разбиение слов

*Посмотрите, что, когда и почему, если вы не знаете о присоединении IFS к разбиению слов*

**давайте зададим IFS только символу пробела:**

```
set -x
```

```
var='I am
a
multiline string'
IFS=' '
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

Это расщепление слова будет работать только на пространствах. Функция `fun` будет выполнена следующим образом:

```
fun I 'am
a
multiline' string
```

> `$var` делится на 3 аргумента. `I` , `am\na\nmultiline` и `string` будет напечатана

**Давайте установим IFS только для новой строки:**

```
IFS=$'\n'
...
```

Теперь `fun` будет выполнена как:

```
fun 'I am' a 'multiline string'
```

> `$var` делится на 3 аргумента. `I am` , `a` , `multiline string` будет напечатана

**Давайте посмотрим, что произойдет, если мы установим IFS в nullstring:**

```
IFS=
...
```

На этот раз `fun` будет выполнена следующим образом:

```
fun 'I am
a
multiline string'
```

> `$var` не разделяется, т. е. остается одним аргументом.

**Вы можете предотвратить разбиение слов, установив IFS в nullstring**

**Общий способ предотвращения разделения слов - использовать двойную кавычку:**

```
fun "$var"
```

будет предотвращать расщепление слов во всех рассмотренных выше случаях, т. е. функция `fun` будет выполняться только с одним аргументом.

## Плохие эффекты расщепления слов

```
$ a='I am a string with spaces'
$ [ $a = $a ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = $a ]` интерпретируется как `[ I am a string with spaces = I am a string with spaces ]` . `[` это `test` команда, для которой `I am a string with spaces` - это не один аргумент, а **6** аргументов!

```
$ [ $a = something ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = something ]` интерпретируется как `[ I am a string with spaces = something ]`

```
$ [ $(grep . file) = 'something' ]
bash: [: too many arguments
```

> Команда `grep` возвращает многострочную строку с пробелами, поэтому вы можете просто представить, сколько аргументов существует ...: D

*Посмотрите, что, когда и зачем основы.*

## Полезность расщепления слов

Существуют случаи, когда разбиение слов может быть полезным:

*Заполнение массива:*

```
arr=($(grep -o '[0-9]\+' file))
```

> Это заполнит `arr` всеми численными значениями, найденными в *файле*

*Циклическое выделение пробелов:*

```
words='foo bar baz'
for w in $words;do
    echo "W: $w"
done
```

Выход:

```
W: foo
```

```
W: bar
W: baz
```

*Передача разделенных пробелов параметров, которые не содержат пробелов:*

```
packs='apache2 php php-mbstring php-mysql'
sudo apt-get install $packs
```

или же

```
packs='
apache2
php
php-mbstring
php-mysql
'
sudo apt-get install $packs
```

> Это установит пакеты. Если вы удвоите кавычки `$packs` то это вызовет ошибку.
>
> Unquoetd `$packs` отправляет все имена разделяемых пространств в качестве аргументов `apt-get` , в то время как цитирование будет отправлять строку `$packs` как один аргумент, а затем `apt-get` попытается установить пакет с именем `apache2 php php-mbstring php-mysql` (для первого), который, очевидно, не существует

*Посмотрите, что, когда и зачем основы.*

## Разделение по разделительным изменениям

Мы можем просто выполнить простую замену разделителей из пространства на новую, как показано ниже.

```
echo $sentence | tr " " "\n"
```

Он разделит значение `sentence` переменной и покажет его по строке соответственно.

Прочитайте Разделение слов онлайн: https://riptutorial.com/ru/bash/topic/5472/разделение-слов

# глава 50: Разделение файлов

## Вступление

Иногда полезно разбить файл на несколько отдельных файлов. Если у вас большие файлы, неплохо было бы разбить его на мелкие куски

## Examples

### Разделить файл

Запуск команды split без каких-либо параметров разбивает файл на 1 или более отдельных файлов, содержащих до 1000 строк каждый.

```
split file
```

Это будет создавать файлы с именем `xaa` , `xab` , `xac` и т.д., каждый из которых содержит до 1000 строк. Как вы можете видеть, все они имеют префикс с буквой `x` по умолчанию. Если исходный файл был меньше 1000 строк, будет создан только один такой файл.

Чтобы изменить префикс, добавьте нужный префикс в конец командной строки

```
split file customprefix
```

Теперь будут созданы файлы с именем `customprefixaa` , `customprefixab` , `customprefixac` и т. Д.

Чтобы указать количество строк для вывода на файл, используйте параметр `-l` . Ниже будет разбит файл на 5000 строк

```
split -l5000 file
```

ИЛИ ЖЕ

```
split --lines=5000 file
```

Кроме того, вы можете указать максимальное количество байтов вместо строк. Это делается с использованием опций `-b` или `--bytes` . Например, чтобы разрешить максимум 1 МБ

```
split --bytes=1MB file
```

### Мы можем использовать sed с параметром w, чтобы разделить файл на

**несколько файлов. Файлы можно разделить, указав адрес или шаблон строки.**

Предположим, что у нас есть этот исходный файл, который мы хотели бы разделить:

```
cat –n sourcefile
```

*1 На Нин Нанг Нонг*
*2 Куда идут коровы Бонг!*
*3 и обезьяны все говорят BOO!*
*4 Существует Nong Nang Ning*
*5 Где деревья идут в Пинг!*
*6 И чайный горшок jibber jabber joo.*
*7 На Нонг Нин Нанг*

Команда для разделения файла по номеру строки:

```
sed '1,3w f1
> 4,7w f2' sourcefile
```

Это записывает строки1 в строку3 в файл f1 и line4 в строку7 в файл f2 из исходного файла.

```
cat –n f1
```

*1 На Нин Нанг Нонг*
*2 Куда идут коровы Бонг!*
*3 и обезьяны все говорят BOO!*

```
cat –n f2
```

1 Существует Nong Nang Ning
2 Где идут деревья Пинг!
3 И чайный горшок jibber jabber joo.
4 На Нонг Нин Нанг

Команда для разделения файла по контексту / шаблону:

```
sed '/Ning/w file1
> /Ping/w file2' sourcefile
```

Это разделяет исходный файл на file1 и file2. file1 содержит все строки, соответствующие Ning, file2 содержит строки, соответствующие Ping.

```
cat file1
```

*О Нин Нанг Нонг*
*Там Нонг Нанг Нин*
*На Нонг Нин Нанг*

```
cat file2
```

*Где деревья идут в Пинг!*

Прочитайте Разделение файлов онлайн: https://riptutorial.com/ru/bash/topic/9151/разделение-файлов

# глава 51: Расширение параметра Bash

## Вступление

Символ `$` вводит расширение параметра, замену команды или арифметическое расширение. Имя параметра или символ, который нужно развернуть, могут быть заключены в фигурные скобки, которые являются необязательными, но служат для защиты переменной, которая должна быть расширена из символов, непосредственно следующих за ней, которые могут быть интерпретированы как часть имени.

Подробнее читайте в руководстве пользователя Bash .

## Синтаксис

- $ {parameter: offset} # Подстрока, начинающаяся со смещения
- $ {parameter: offset: length} # Подстрока длины «длина», начинающаяся со смещения
- $ {# parameter} # Длина параметра
- $ {parameter / pattern / string} # Заменить первое вхождение шаблона на строку
- $ {parameter // pattern / string} # Заменить все вхождения шаблона в строку
- $ {parameter / # pattern / string} # Заменить шаблон на строку, если шаблон находится в начале
- $ {параметр /% pattern / string} # Заменить шаблон на строку, если шаблон находится в конце
- $ {parameter # pattern} # Удалить кратчайшее соответствие шаблона с начала параметра
- $ {parameter ## pattern} # Удалить наибольшее совпадение шаблона с начала параметра
- $ {parameter% pattern} # Удалить кратчайшее совпадение шаблона с конца параметра
- $ {parameter %% pattern} # Удалить наибольшее совпадение шаблона с конца параметра
- $ {parameter: -word} # Развернуть до слова, если параметр unset / undefined
- $ {parameter: = word} # Развернуть до слова, если параметр unset / undefined и установить параметр
- $ {parameter: + word} # Развернуть до слова, если параметр установлен / определен

## Examples

### Подстроки и подмассивы

```
var='0123456789abcdef'
```

```
# Define a zero-based offset
$ printf '%s\n' "${var:3}"
3456789abcdef

# Offset and length of substring
$ printf '%s\n' "${var:3:4}"
3456
```

4,2

```
# Negative length counts from the end of the string
$ printf '%s\n' "${var:3:-5}"
3456789a

# Negative offset counts from the end
# Needs a space to avoid confusion with ${var:-6}
$ printf '%s\n' "${var: -6}"
abcdef

# Alternative: parentheses
$ printf '%s\n' "${var:(-6)}"
abcdef

# Negative offset and negative length
$ printf '%s\n' "${var: -6:-5}"
a
```

Те же расширения применяются, если параметр является **позиционным параметром** или **элементом подстрочного массива** :

```
# Set positional parameter $1
set -- 0123456789abcdef

# Define offset
$ printf '%s\n' "${1:5}"
56789abcdef

# Assign to array element
myarr[0]='0123456789abcdef'

# Define offset and length
$ printf '%s\n' "${myarr[0]:7:3}"
789
```

Аналогичные расширения применяются к **позиционным параметрам** , где смещения однонаправлены:

```
# Set positional parameters $1, $2, ...
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f

# Define an offset (beware $0 (not a positional parameter)
# is being considered here as well)
$ printf '%s\n' "${@:10}"
0
a
b
c
```

```
d
e
f

# Define an offset and a length
$ printf '%s\n' "${@:10:3}"
0
a
b

# No negative lengths allowed for positional parameters
$ printf '%s\n' "${@:10:-2}"
bash: -2: substring expression < 0

# Negative offset counts from the end
# Needs a space to avoid confusion with ${@:-10:2}
$ printf '%s\n' "${@: -10:2}"
7
8

# ${@:0} is $0 which is not otherwise a positional parameters or part
# of $@
$ printf '%s\n' "${@:0:2}"
/usr/bin/bash
1
```

Расширение подстроки может использоваться с **индексированными массивами** :

```
# Create array (zero-based indices)
$ myarr=(0 1 2 3 4 5 6 7 8 9 a b c d e f)

# Elements with index 5 and higher
$ printf '%s\n' "${myarr[@]:12}"
c
d
e
f

# 3 elements, starting with index 5
$ printf '%s\n' "${myarr[@]:5:3}"
5
6
7

# The last element of the array
$ printf '%s\n' "${myarr[@]: -1}"
f
```

## Длина параметра

```
# Length of a string
$ var='12345'
$ echo "${#var}"
5
```

Обратите внимание, что это длина числа *символов,* которая не обязательно совпадает с количеством *байтов* (например, в UTF-8, где большинство символов кодируются более чем

в одном байте), а также количество *глифов / графем* (некоторые из которых комбинации символов), и не обязательно совпадает с шириной отображения.

```
# Number of array elements
$ myarr=(1 2 3)
$ echo "${#myarr[@]}"
3

# Works for positional parameters as well
$ set -- 1 2 3 4
$ echo "${#@}"
4

# But more commonly (and portably to other shells), one would use
$ echo "$#"
4
```

## Изменение случая буквенных символов

4,0

В верхний регистр

```
$ v="hello"
# Just the first character
$ printf '%s\n' "${v^}"
Hello
# All characters
$ printf '%s\n' "${v^^}"
HELLO
# Alternative
$ v="hello world"
$ declare -u string="$v"
$ echo "$string"
HELLO WORLD
```

В нижнем регистре

```
$ v="BYE"
# Just the first character
$ printf '%s\n' "${v,}"
bYE
# All characters
$ printf '%s\n' "${v,,}"
bye
# Alternative
$ v="HELLO WORLD"
$ declare -l string="$v"
$ echo "$string"
hello world
```

Toggle Case

```
$ v="Hello World"
# All chars
```

```
$ echo "${v~~}"
hELLO wORLD
$ echo "${v~}"
# Just the first char
hello World
```

## Неверный параметр

`Bash` позволяет получить значение переменной, имя которой содержится в другой переменной. Пример переменных:

```
$ red="the color red"
$ green="the color green"

$ color=red
$ echo "${!color}"
the color red
$ color=green
$ echo "${!color}"
the color green
```

Еще несколько примеров, демонстрирующих использование косвенного расширения:

```
 $ foo=10
 $ x=foo
 $ echo ${x}      #Classic variable print
 foo

 $ foo=10
 $ x=foo
 $ echo ${!x}     #Indirect expansion
 10
```

Еще один пример:

```
$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${i}";done; }; argtester -ab -cd -ef
1   #i expanded to 1
2   #i expanded to 2
3   #i expanded to 3

$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${!i}";done; }; argtester -ab -cd -ef
-ab     # i=1 --> expanded to $1 ---> expanded to first argument sent to function
-cd     # i=2 --> expanded to $2 ---> expanded to second argument sent to function
-ef     # i=3 --> expanded to $3 ---> expanded to third argument sent to function
```

## Замена значения по умолчанию

```
${parameter:-word}
```

Если параметр не задан или нулевым, заменяется слово. В противном случае значение параметра заменяется.

```
$ unset var
$ echo "${var:-XX}"     # Parameter is unset -> expansion XX occurs
XX
$ var=""                # Parameter is null -> expansion XX occurs
$ echo "${var:-XX}"
XX
$ var=23                # Parameter is not null -> original expansion occurs
$ echo "${var:-XX}"
23
```

```
${parameter:=word}
```

Если параметр не задан или null, расширение слова присваивается параметру. Затем значение параметра заменяется. Таким образом, нельзя назначать позиционные параметры и специальные параметры.

```
$ unset var
$ echo "${var:=XX}"     # Parameter is unset -> word is assigned to XX
XX
$ echo "$var"
XX
$ var=""                # Parameter is null -> word is assigned to XX
$ echo "${var:=XX}"
XX
$ echo "$var"
XX
$ var=23                # Parameter is not null -> no assignment occurs
$ echo "${var:=XX}"
23
$ echo "$var"
23
```

## Ошибка, если переменная пуста или не установлена

Семантика для этого аналогична семантике замещения по умолчанию, но вместо того, чтобы подставлять значение по умолчанию, она выдает ошибку с предоставленным сообщением об ошибке. Формы: `${VARNAME?ERRMSG}` и `${VARNAME:?ERRMSG}` . Форма с `:` будет ошибкой нашей, если переменная не **установлена** *или* **пуста** , тогда как форма без ошибок будет только выходить из строя, если переменная не *установлена* . Если `ERRMSG` ошибка, `ERRMSG` а код выхода - `1` .

```
#!/bin/bash
FOO=
# ./script.sh: line 4: FOO: EMPTY
echo "FOO is ${FOO:?EMPTY}"
# FOO is
echo "FOO is ${FOO?UNSET}"
# ./script.sh: line 8: BAR: EMPTY
echo "BAR is ${BAR:?EMPTY}"
# ./script.sh: line 10: BAR: UNSET
echo "BAR is ${BAR?UNSET}"
```

Запустите полный пример выше каждого из операторов эхо-сообщений об ошибке, которые

необходимо прокомментировать, чтобы продолжить.

## Удаление шаблона с начала строки

Наименьшее совпадение:

```
$ a='I am a string'
$ echo "${a#*a}"
m a string
```

Наибольшее совпадение:

```
$ echo "${a##*a}"
 string
```

## Удаление шаблона с конца строки

Наименьшее совпадение:

```
$ a='I am a string'
$ echo "${a%a*}"
I am
```

Наибольшее совпадение:

```
$ echo "${a%%a*}"
I
```

## Заменить шаблон в строке

Первый матч:

```
$ a='I am a string'
$ echo "${a/a/A}"
I Am a string
```

Все совпадения:

```
$ echo "${a//a/A}"
I Am A string
```

Матч в начале:

```
$ echo "${a/#I/y}"
y am a string
```

Матч в конце:

```
$ echo "${a/%g/N}"
I am a strinN
```

Заменить шаблон ничем:

```
$ echo "${a/g/}"
I am a strin
```

Добавить префикс в элементы массива:

```
$ A=(hello world)
$ echo "${A[@]/#/R}"
Rhello Rworld
```

## Мутирование во время расширения

Переменные не обязательно должны расширяться до их значений - подстроки могут быть извлечены во время расширения, что может быть полезно для извлечения расширений файлов или частей путей. Глобущие символы сохраняют свои обычные значения, поэтому `.*` Относится к буквальной точке, за которой следует любая последовательность символов; это не регулярное выражение.

```
$ v=foo-bar-baz
$ echo ${v%%-*}
foo
$ echo ${v%-*}
foo-bar
$ echo ${v##*-}
baz
$ echo ${v#*-}
bar-baz
```

Также возможно расширить переменную с использованием значения по умолчанию - например, я хочу вызвать редактор пользователя, но если они не установили его, я бы хотел дать им `vim` .

```
$ EDITOR=nano
$ ${EDITOR:-vim} /tmp/some_file
# opens nano
$ unset EDITOR
$ $ ${EDITOR:-vim} /tmp/some_file
# opens vim
```

Существует два разных способа выполнения этого расширения, которые отличаются тем, что соответствующая переменная пуста или не установлена. Использование `:-` будет использовать значение по умолчанию, если переменная является либо незамкнутой, либо пустой, тогда как `-` используется только по умолчанию, если переменная не установлена, но будет использовать переменную, если она установлена в пустую строку:

```
$ a="set"
$ b=""
$ unset c
$ echo ${a:-default_a} ${b:-default_b} ${c:-default_c}
set default_b default_c
$ echo ${a-default_a} ${b-default_b} ${c-default_c}
set default_c
```

Подобно значениям по умолчанию, могут быть предоставлены альтернативы; где используется значение по умолчанию, если конкретная переменная недоступна, используется альтернатива, если переменная доступна.

```
$ a="set"
$ b=""
$ echo ${a:+alternative_a} ${b:+alternative_b}
alternative_a
```

Отмечая, что эти расширения могут быть вложенными, использование альтернатив становится особенно полезным при подаче аргументов в флаги командной строки;

```
$ output_file=/tmp/foo
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget -o /tmp/foo www.stackexchange.com
$ unset output_file
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget  www.stackexchange.com
```

## Расширение параметров и имена файлов

Вы можете использовать Bash Parameter Expansion для эмуляции общих операций обработки имен файлов, таких как `basename` и `dirname` .

Мы будем использовать это как наш пример:

```
FILENAME="/tmp/example/myfile.txt"
```

Чтобы эмулировать имя `dirname` и вернуть имя каталога пути к файлу:

```
echo "${FILENAME%/*}"
#Out: /tmp/example
```

Чтобы эмулировать `basename $FILENAME` и вернуть имя файла пути к файлу:

```
echo "${FILENAME##*/}"
#Out: myfile.txt
```

Чтобы эмулировать `basename $FILENAME .txt` и вернуть имя файла без `.txt`. расширение:

```
BASENAME="${FILENAME##*/}"
```

```
echo "${BASENAME%%.txt}"
#Out: myfile
```

Прочитайте Расширение параметра Bash онлайн: https://riptutorial.com/ru/bash/topic/502/расширение-параметра-bash

# глава 52: Расширение скобы

## замечания

Справочное руководство Bash: расширение брекета

## Examples

### Создание каталогов для группировки файлов по месяцам и годам

```
$ mkdir 20{09..11}-{01..12}
```

Ввод команды `ls` покажет, что были созданы следующие каталоги:

```
2009-01 2009-04 2009-07 2009-10 2010-01 2010-04 2010-07 2010-10 2011-01 2011-04 2011-07 2011-
10
2009-02 2009-05 2009-08 2009-11 2010-02 2010-05 2010-08 2010-11 2011-02 2011-05 2011-08 2011-
11
2009-03 2009-06 2009-09 2009-12 2010-03 2010-06 2010-09 2010-12 2011-03 2011-06 2011-09 2011-
12
```

Помещение `0` перед `9` в примере гарантирует, что цифры дополняются одним `0` . Вы также можете вводить числа с несколькими нулями, например:

```
$ echo {001..10}
001 002 003 004 005 006 007 008 009 010
```

### Создать резервную копию dotfiles

```
$ cp .vimrc{,.bak}
```

Это расширяет команду `cp .vimrc .vimrc.bak` .

### Изменение расширения имени файла

```
$ mv filename.{jar,zip}
```

Это расширится на `mv filename.jar filename.zip` .

### Использовать приращения

```
$ echo {0..10..2}
0 2 4 6 8 10
```

Третий параметр для указания приращения, т. `{start..end..increment}`

Использование приращений не ограничено просто цифрами

```
$ for c in {a..z..5}; do echo -n $c; done
afkpuz
```

## Использование расширения скобок для создания списков

Bash может легко создавать списки из буквенно-цифровых символов.

```
# list from a to z
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# reverse from z to a
$ echo {z..a}
z y x w v u t s r q p o n m l k j i h g f e d c b a

# digits
$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# with leading zeros
$ echo {01..20}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20

# reverse digit
$ echo {20..1}
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

# reversed with leading zeros
$ echo {20..01}
20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01

# combining multiple braces
$ echo {a..d}{1..3}
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3
```

Расширение скобки является самым первым расширением, которое имеет место, поэтому его нельзя комбинировать с другими расширениями.

Могут использоваться только символы и цифры.

Это не будет работать: `echo {$(date +$H)..24}`

## Создание нескольких каталогов с суб-каталогами

```
mkdir -p toplevel/sublevel_{01..09}/{child1,child2,child3}
```

Это позволит создать папку верхнего уровня называется `toplevel` , девять папок внутри от `toplevel` с именем `sublevel_01` , `sublevel_02` и т.д. Тогда внутри этих подуровней: `child1` , `child2`

, `child3` папки, давая вам:

```
toplevel/sublevel_01/child1
toplevel/sublevel_01/child2
toplevel/sublevel_01/child3
toplevel/sublevel_02/child1
```

и так далее. Я считаю, что это очень полезно для создания нескольких папок и подпапок для моих конкретных целей, с одной командой bash. Заменяйте переменные, чтобы помочь автоматизировать / проанализировать информацию, предоставленную скрипту.

Прочитайте Расширение скобы онлайн: https://riptutorial.com/ru/bash/topic/3351/расширение-скобы

# глава 53: Расшифровка URL

## Examples

### Простой пример

#### Кодированный URL

HTTP% 3A% 2F% 2Fwww.foo.com% 2Findex.php% 3Fid% 3Dqwerty

#### Используйте эту команду для декодирования URL-адреса

```
echo "http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty" | sed -e "s/%\([0-9A-F][0-9A-F]\)/\\\\\x\1/g" | xargs -0 echo -e
```

#### Декодированный URL (результат команды)

http://www.foo.com/index.php?id=qwerty

### Использование printf для декодирования строки

```
#!bin/bash

$ string='Question%20-
%20%22how%20do%20I%20decode%20a%20percent%20encoded%20string%3F%22%0AAnswer%20%20%20-
%20Use%20printf%20%3A)'
$ printf '%b\n' "${string//%/\\x}"

# the result
Question - "how do I decode a percent encoded string?"
Answer   - Use printf :)
```

Прочитайте Расшифровка URL онлайн: https://riptutorial.com/ru/bash/topic/10895/расшифровка-url

# глава 54: Сеть с Bash

## Вступление

Bash часто используется в управлении и обслуживании серверов и кластеров. Информация, относящаяся к типичным командам, используемым сетевыми операциями, когда использовать какую команду для этой цели, и примеры / образцы уникальных и / или интересных приложений из нее должны быть включены

## Examples

### Сетевые команды

```
ifconfig
```

Вышеприведенная команда покажет весь активный интерфейс машины, а также предоставит информацию о

1. Назначение IP-адреса для интерфейса
2. MAC-адрес интерфейса
3. Адрес широковещания
4. Передача и получение байтов

Пример

```
ifconfig -a
```

Вышеупомянутая команда также показывает интерфейс отключения

```
ifconfig eth0
```

Вышеупомянутая команда будет показывать только интерфейс eth0

```
ifconfig eth0 192.168.1.100 netmask 255.255.255.0
```

Вышеупомянутая команда назначит статический IP-адрес eth0-интерфейсу

```
ifup eth0
```

Вышеупомянутая команда позволит использовать интерфейс eth0

```
ifdown eth0
```

Команда ниже отключит интерфейс eth0

```
ping
```

Вышеупомянутая команда (Packet Internet Grouper) предназначена для проверки возможности подключения между двумя узлами

```
ping -c2 8.8.8.8
```

Вышеупомянутая команда проверит или проверит подключение к серверу google в течение 2 секунд.

```
traceroute
```

Вышеприведенная команда предназначена для устранения неполадок, чтобы узнать количество перелетов, которые были достигнуты для достижения цели.

```
netstat
```

Вышеуказанная команда (статистика сети) предоставляет информацию о подключении и их состояние

```
dig www.google.com
```

Вышеприведенная команда (группа данных домена) запрашивает информацию, связанную с DNS

```
nslookup www.google.com
```

Вышеуказанная команда запрашивает DNS и узнает IP-адрес соответствующего имени веб-сайта.

```
route
```

Вышеупомянутая команда используется для проверки информации о маршруте сети Netwrok. В основном это показывает таблицу маршрутизации

```
router add default gw 192.168.1.1 eth0
```

Вышеприведенная команда добавит по умолчанию маршрут сети eth0 Interface к 192.168.1.1 в таблицу маршрутизации.

```
route del default
```

Вышеупомянутая команда удалит маршрут по умолчанию из таблицы маршрутизации

Прочитайте Сеть с Bash онлайн: https://riptutorial.com/ru/bash/topic/10737/сеть-с-bash

# глава 55: Скрипт shebang

## Синтаксис

- Используйте `/bin/bash` в качестве интерпретатора bash:

  #! / Bin / Баш

- Найдите интерпретатор bash в `PATH` среды `PATH` с помощью `env` :

  #! / usr / bin / env bash

## замечания

Распространенная ошибка заключается в попытке выполнить Windows-файлы с отформатированными файлами `\r\n` системах UNIX / Linux, в этом случае используемый интерпретатор скриптов в shebang:

```
/bin/bash\r
```

И не замечается, но его трудно понять.

## Examples

### Прямая shebang

Чтобы выполнить файл сценария с помощью интерпретатора `bash` , **первая строка** файла сценария должна указывать абсолютный путь к исполняемому файлу `bash` :

```
#!/bin/bash
```

Путь `bash` в shebang разрешен и используется только в том случае, если скрипт запускается следующим образом:

```
./script.sh
```

Сценарий должен иметь разрешение на выполнение.

Shebang игнорируется, когда интерпретатор `bash` явно указывается для выполнения скрипта:

```
bash script.sh
```

### Env shebang

Чтобы выполнить файл сценария с исполняемым файлом `bash` найденным в `PATH` среды `PATH` с помощью исполняемого `env` , **первая строка** файла сценария должна указывать абсолютный путь к исполняемому файлу `env` с аргументом `bash` :

```
#!/usr/bin/env bash
```

Путь `env` в shebang разрешен и используется только в том случае, если скрипт запускается следующим образом:

```
script.sh
```

Сценарий должен иметь разрешение на выполнение.

Shebang игнорируется, когда интерпретатор `bash` явно указывается для выполнения скрипта:

```
bash script.sh
```

### Другие шебанги

Ядро знает о двух типах программ. Бинарная программа идентифицируется заголовком ELF ( **E** xtenable **L** oadable **F** ormat), который обычно создается компилятором. Второй - это скрипты любого типа.

Если файл начинается в самой первой строке с последовательностью **#!** то следующей строкой должен быть путь к интерпретатору. Если ядро читает эту строку, оно вызывает интерпретатор, названный этим именем пути, и дает все следующие слова в этой строке в качестве аргументов интерпретатору. Если нет файла с именем «something» или «wrong»:

```
#!/bin/bash something wrong
echo "This line never gets printed"
```

bash пытается выполнить свой аргумент «что-то неправильно», которого не существует. Также добавляется имя файла сценария. Чтобы увидеть это, используйте **эхо-** шебанг:

```
#"/bin/echo something wrong
# and now call this script named "thisscript" like so:
# thisscript one two
# the output will be:
something wrong ./thisscript one two
```

Некоторые программы, такие как **awk,** используют этот метод для запуска более длинных скриптов, находящихся в файле на диске.

Прочитайте Скрипт shebang онлайн: https://riptutorial.com/ru/bash/topic/3658/скрипт-shebang

# глава 56: Скрипты CGI

## Examples

### Метод запроса: GET

CGI-скрипт довольно легко вызвать через `GET` .
Сначала вам понадобится `encoded url` -адрес скрипта.

Затем вы добавляете знак вопроса `?` за которыми следуют переменные.

- Каждая переменная должна иметь две секции, разделенные на **=** .
  Первый раздел должен всегда быть уникальным именем для каждой переменной,
  а вторая часть имеет в ней только значения
- Переменные разделяются символом **&**
- Общая длина строки не должна превышать **255** символов
- Имена и значения должны быть закодированы в html (replace: **<, /?: @ & = + $** )
  **Подсказка:**
  При использовании **html-форм** метод запроса может быть сгенерирован им самостоятельно.
  С помощью **Ajax** вы можете кодировать все через **encodeURI** и **encodeURIComponent**

**Пример:**

```
http://www.example.com/cgi-bin/script.sh?var1=Hello%20World!&var2=This%20is%20a%20Test.&
```

Сервер должен взаимодействовать только через **общий доступ к ресурсам Cross-Origin** (CORS), чтобы сделать запрос более безопасным. В этой витрине мы используем **CORS** для определения типа `Data-Type` мы хотим использовать.

Есть много `Data-Types` мы можем выбрать, наиболее распространенными являются ...

- **текст** / **html**
- **текст** / **обычный**
- **Применение** / **JSON**

При отправке запроса сервер также создает множество переменных среды. На данный момент наиболее важными переменными среды являются `$REQUEST_METHOD` и `$QUERY_STRING` .

Метод **запроса** должен `GET` !
**Строка запроса** включает все `html-endoded data` .

**Сценарий**

```
#!/bin/bash

# CORS is the way to communicate, so lets response to the server first
echo "Content-type: text/html"    # set the data-type we want to use
echo ""    # we dont need more rules, the empty line initiate this.

# CORS are set in stone and any communication from now on will be like reading a html-
document.
# Therefor we need to create any stdout in html format!

# create html scructure and send it to stdout
echo "<!DOCTYPE html>"
echo "<html><head>"

# The content will be created depending on the Request Method
if [ "$REQUEST_METHOD" = "GET" ]; then

    # Note that the environment variables $REQUEST_METHOD and $QUERY_STRING can be processed
by the shell directly.
    # One must filter the input to avoid cross site scripting.

    Var1=$(echo "$QUERY_STRING" | sed -n 's/^.*var1=\([^&]*\).*$/\1/p')    # read value of
"var1"
    Var1_Dec=$(echo -e $(echo "$Var1" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))    # html decode

    Var2=$(echo "$QUERY_STRING" | sed -n 's/^.*var2=\([^&]*\).*$/\1/p')
    Var2_Dec=$(echo -e $(echo "$Var2" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))

    # create content for stdout
    echo "<title>Bash-CGI Example 1</title>"
    echo "</head><body>"
    echo "<h1>Bash-CGI Example 1</h1>"
    echo "<p>QUERY_STRING: ${QUERY_STRING}<br>var1=${Var1_Dec}<br>var2=${Var2_Dec}</p>"    #
print the values to stdout

else

    echo "<title>456 Wrong Request Method</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Requesting data went wrong.<br>The Request method has to be \"GET\" only!</p>"

fi

echo "<hr>"
echo "$SERVER_SIGNATURE"    # an other environment variable
echo "</body></html>"    # close html

exit 0
```

**Html-документ** будет выглядеть так ...

```
<html><head>
<title>Bash-CGI Example 1</title>
</head><body>
<h1>Bash-CGI Example 1</h1>
<p>QUERY_STRING: var1=Hello%20World!&amp;var2=This%20is%20a%20Test.&amp;<br>var1=Hello
World!<br>var2=This is a Test.</p>
<hr>
<address>Apache/2.4.10 (Debian) Server at example.com Port 80</address>
```

```
</body></html>
```

**Вывод** переменных будет выглядеть следующим образом ...

```
var1=Hello%20World!&var2=This%20is%20a%20Test.&
Hello World!
This is a Test.
Apache/2.4.10 (Debian) Server at example.com Port 80
```

**Отрицательные побочные эффекты ...**

- Все кодирование и декодирование не выглядят хорошо, но необходимо
- Запрос будет общедоступным и оставить лоток
- Размер запроса ограничен
- Нуждается в защите от Cross-Side-Scripting (XSS)

## Метод запроса: POST / w JSON

Использование метода запроса `POST` в сочетании с `SSL` делает datatransfer более безопасным.

К тому же...

- Большая часть кодирования и декодирования больше не нужна
- URL-адрес будет видимым для любого и должен быть закодирован в URL-адресе. Данные будут отправляться отдельно, и поэтому они должны быть защищены через SSL
- Размер данных почти не сообщается
- По-прежнему требуется защита от Cross-Side-Scripting (XSS)

Чтобы сделать эту демонстрацию простой, мы хотим получить **данные JSON** и связь должна быть **связана с совместным использованием ресурсов Cross-Origin** (CORS).

Следующий сценарий также продемонстрирует два разных типа **содержимого** .

```
#!/bin/bash

exec 2>/dev/null    # We dont want any error messages be printed to stdout
trap "response_with_html && exit 0" ERR    # response with an html message when an error
occurred and close the script

function response_with_html(){
    echo "Content-type: text/html"
    echo ""
```

```
    echo "<!DOCTYPE html>"
    echo "<html><head>"
    echo "<title>456</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Attempt to communicate with the server went wrong.</p>"
    echo "<hr>"
    echo "$SERVER_SIGNATURE"
    echo "</body></html>"
}

function response_with_json(){
    echo "Content-type: application/json"
    echo ""
    echo "{\"message\": \"Hello World!\"}"
}

if [ "$REQUEST_METHOD" = "POST" ]; then

    # The environment variabe $CONTENT_TYPE describes the data-type received
    case "$CONTENT_TYPE" in
    application/json)
        # The environment variabe $CONTENT_LENGTH describes the size of the data
        read -n "$CONTENT_LENGTH" QUERY_STRING_POST        # read datastream

        # The following lines will prevent XSS and check for valide JSON-Data.
        # But these Symbols need to be encoded somehow before sending to this script
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed "s/'//g" | sed
's/\$//g;s/`//g;s/\*//g;s/\\//g' )        # removes some symbols (like \ * ` $ ') to prevent
XSS with Bash and SQL.
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed -e :a -e 's/<[^>]*>//g;/</N;//ba')
# removes most html declarations to prevent XSS within documents
        JSON=$(echo "$QUERY_STRING_POST" | jq .)        # json encode - This is a pretty save
way to check for valide json code
    ;;
    *)
        response_with_html
        exit 0
    ;;
    esac

else
    response_with_html
    exit 0
fi

# Some Commands ...

response_with_json

exit 0
```

Вы получите `{"message":"Hello World!"}` качестве ответа при отправке **JSON-Data** через `POST` в этот скрипт. Каждая вещь получит html-документ.

Важным является также varialbe `$JSON` . Эта переменная свободна от XSS, но все же может иметь неправильные значения в ней и должна быть проверена в первую очередь. Пожалуйста, имейте это в виду.

Этот код работает аналогично без JSON.
Вы можете получить любые данные таким образом.
Вам просто нужно изменить `Content-Type` для ваших нужд.

**Пример:**

```
if [ "$REQUEST_METHOD" = "POST" ]; then
    case "$CONTENT_TYPE" in
    application/x-www-form-urlencoded)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    text/plain)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    ;;
    esac
fi
```

И последнее, но не менее важное: не забудьте ответить на все запросы, иначе сторонние программы не узнают, если они преуспели

Прочитайте Скрипты CGI онлайн: https://riptutorial.com/ru/bash/topic/9603/скрипты-cgi

# глава 57: Скрипты с параметрами

## замечания

- `shift` сдвигает позиционные параметры влево, так что `$2` становится `$1` , `$3` становится `$2` и так далее.
- `"$@"` - это массив всех позиционных параметров, передаваемых скрипту / функции.
- `"$*"` - это строка, состоящая из всех позиционных параметров, переданных скрипту / функции.

## Examples

### Многопараметрический анализ

Чтобы разобрать множество параметров, Предпочитаемый способ сделать это с помощью цикла *WHILE, тематическое* заявление, и *сдвиг.*

`shift` используется для появления первого параметра в серии, что составляет *$ 2* , теперь составляет *$ 1* . Это полезно для обработки аргументов по одному.

```
#!/bin/bash

# Load the user defined parameters
while [[ $# > 0 ]]
do
        case "$1" in

                -a|--valueA)
                        valA="$2"
                        shift
                        ;;

                -b|--valueB)
                        valB="$2"
                        shift
                        ;;

                --help|*)
                        echo "Usage:"
                        echo "    --valueA \"value\""
                        echo "    --valueB \"value\""
                        echo "    --help"
                        exit 1
                        ;;
        esac
        shift
done

echo "A: $valA"
echo "B: $valB"
```

```
$ ./multipleParams.sh --help
Usage:
    --valueA "value"
    --valueB "value"
    --help

$ ./multipleParams.sh
A:
B:

$ ./multipleParams.sh --valueB 2
A:
B: 2

$ ./multipleParams.sh --valueB 2 --valueA "hello world"
A: hello world
B: 2
```

### Доступ к параметрам

При выполнении сценария Bash параметры, переданные в скрипт, называются в соответствии с их положением: `$1` - это имя первого параметра, `$2` - имя второго параметра и т. Д.

Отсутствующий параметр просто оценивает пустую строку. Проверка наличия параметра может быть выполнена следующим образом:

```
if [ -z "$1" ]; then
    echo "No argument supplied"
fi
```

## Получение всех параметров

`$@` и `$*` - это способы взаимодействия со всеми параметрами скрипта. Обращаясь к странице руководства Bash , мы видим, что:

- `$*` : Расширяется до позиционных параметров, начиная с одного. Когда расширение происходит в двойных кавычках, оно расширяется до одного слова со значением каждого параметра, разделенным первым символом специальной переменной IFS.
- `$@` : Расширяется до позиционных параметров, начиная с единицы. Когда расширение происходит в двойных кавычках, каждый параметр расширяется до отдельного слова.

## Получение количества параметров

`$#` получает количество параметров, переданных в скрипт. Типичным примером использования будет проверка того, передано ли соответствующее количество аргументов:

```
if [ $# -eq 0 ]; then
    echo "No arguments supplied"
fi
```

# Пример 1

Прокрутите все аргументы и проверьте, являются ли они файлами:

```
for item in "$@"
do
    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

# Пример 2.

Прокрутите все аргументы и проверьте, являются ли они файлами:

```
for (( i = 1; i <= $#; ++ i ))
do
    item=${@:$i:1}

    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

## Анализ аргументов с использованием цикла for

Простой пример, который предоставляет варианты:

| Опт | Чередующийся Опт | подробности |
|---|---|---|
| `-h` | `--help` | Показать справку |
| `-v` | `--version` | Показать информацию о версии |
| `-dr path` | `--doc-root path` | Опция, которая принимает вторичный параметр (путь) |
| `-i` | `--install` | Логическая опция (true / false) |
| `-*` | - | Неверная опция |

```
#!/bin/bash
dr=''
```

```
install=false

skip=false
for op in "$@";do
    if $skip;then skip=false;continue;fi
    case "$op" in
        -v|--version)
            echo "$ver_info"
            shift
            exit 0
            ;;
        -h|--help)
            echo "$help"
            shift
            exit 0
            ;;
        -dr|--doc-root)
            shift
            if [[ "$1" != "" ]]; then
                dr="${1/%\//}"
                shift
                skip=true
            else
                echo "E: Arg missing for -dr option"
                exit 1
            fi
            ;;
        -i|--install)
            install=true
            shift
            ;;
        -*)
            echo "E: Invalid option: $1"
            shift
            exit 1
            ;;
    esac
done
```

## Сценарий обертки

Скрипт Wrapper - это сценарий, который обертывает другой скрипт или команду, чтобы обеспечить дополнительные функциональные возможности или просто сделать что-то менее утомительное.

Например, фактический `egrep` в новой системе GNU / Linux заменяется скриптом-оболочкой с именем `egrep` . Вот как это выглядит:

```
#!/bin/sh
exec grep -E "$@"
```

Итак, когда вы запускаете `egrep` в таких системах, вы фактически запускаете `grep -E` со всеми перенаправленными аргументами.

В общем случае, если вы хотите запустить пример сценария / команды `exmp` с другим

сценарием `mexmp` то обертка `mexmp` сценарий будет выглядеть следующим образом :

```
#!/bin/sh
exmp "$@" # Add other options before "$@"
# or
#full/path/to/exmp "$@"
```

## Разделить строку на массив в Bash

Допустим, у нас есть параметр String, и мы хотим разбить его запятой

```
my_param="foo,bar,bash"
```

Чтобы разбить эту строку запятой, мы можем использовать;

```
IFS=',' read -r -a array <<< "$my_param"
```

Здесь IFS - это специальная переменная, называемая внутренним разделителем полей, которая определяет символ или символы, используемые для разделения шаблона на токены для некоторых операций.

Чтобы получить доступ к отдельному элементу:

```
echo "${array[0]}"
```

Чтобы перебрать элементы:

```
for element in "${array[@]}"
do
    echo "$element"
done
```

Чтобы получить как индекс, так и значение:

```
for index in "${!array[@]}"
do
    echo "$index ${array[index]}"
done
```

Прочитайте Скрипты с параметрами онлайн: https://riptutorial.com/ru/bash/topic/746/скрипты-с-параметрами

# глава 58: Совпадение шаблонов и регулярные выражения

## Синтаксис

- $ shopt -u option # Деактивировать встроенную опцию Bash '
- $ shopt -s option # Активировать встроенную опцию Bash '

## замечания

### Классы символов

Допустимые классы символов для `[]` glob определяются стандартом POSIX:

    alnum alpha ascii blank cntrl digit graph lower print punct space верхнее слово xdigit

Внутри `[]` может использоваться более одного класса или диапазона символов, например,

```
$ echo a[a-z[:blank:]0-9]*
```

будет соответствовать любому файлу, начинающемуся с `a` и за ним следует либо строчная буква, либо пробел или цифра.

Следует иметь в виду, однако, что `[]` glob может быть полностью отвергнут, а не только его части. Отрицающая символ *должен* быть первым символом после открытия `[` , например, это выражение соответствует всем файлам , которые **не** начинаются с `a`

```
$ echo [^a]*
```

Следующее соответствует всем файлам, начинающимся с цифры или `^`

```
$ echo [[:alpha:]^a]*
```

Он **не** соответствует файлу или папке, которая начинается с буквы, кроме `a` потому что `^` интерпретируется как литерал `^` .

### Экранирование символов глобуса

Возможно, что файл или папка содержит символ глобуса как часть его имени. В этом случае glob может быть экранирован с предыдущим `\` для того, чтобы выполнить литеральное совпадение. Другой подход заключается в использовании двойных `""` или одиночных `''` котировок для обращения к файлу. Bash не обрабатывает глобусы, которые

заключены в `""` или `''` .

**Отличие от регулярных выражений**

Наиболее существенное различие между глобусами и регулярными выражениями состоит в том, что для правильных регулярных выражений требуется определитель, а также квантификатор. Квалификатор определяет, *что* нужно совместить, и квантификатор сообщает, как часто соответствовать определителю. Эквивалентный RegEx для `*` glob есть `.*` Где `.` обозначает любой символ и `*` обозначает ноль или более совпадений предыдущего символа. Эквивалентный RegEx для `?` glob `.{1}` . Как и раньше, определитель `.` соответствует любому символу, а `{1}` указывает, что он соответствует предыдущему квалификатору ровно один раз. Это не следует путать с `?` квантификатор, который соответствует нулю или один раз в RegEx. Шаблон `[]` glob можно использовать точно так же в RegEx, если за ним следует обязательный квантификатор.

**Эквивалентные регулярные выражения**

| Glob | RegEx |
|---|---|
| `*` | `.*` |
| `?` | `.` |
| `[]` | `[]` |

# Examples

## Проверьте, соответствует ли строка регулярному выражению

3.0

Проверьте, состоит ли строка в точности 8 цифр:

```
$ date=20150624
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
yes
$ date=hello
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
no
```

## * Glob

**подготовка**

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
```

```
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Звездочка *, вероятно, является наиболее часто используемым шаром. Он просто соответствует любой строке

```
$ echo *acy
macy stacy tracy
```

Единый * не будет соответствовать файлам и папкам, которые находятся в подпапках

```
$ echo *
emptyfolder folder macy stacy tracy
$ echo folder/*
folder/anotherfolder folder/subfolder
```

## ** glob

4,0

**подготовка**

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -s globstar
```

Bash способен интерпретировать две соседние звездочки как единый глобус. При `globstar` опции `globstar` это можно использовать для соответствия папкам, которые находятся глубже в структуре каталогов

```
echo **
emptyfolder folder folder/anotherfolder folder/anotherfolder/content
folder/anotherfolder/content/deepfolder folder/anotherfolder/content/deepfolder/file
folder/subfolder folder/subfolder/content folder/subfolder/content/deepfolder
folder/subfolder/content/deepfolder/file macy stacy tracy
```

`**` можно думать о расширении пути, независимо от того, насколько глубока траектория. Этот пример соответствует любому файлу или папке, которая начинается с `deep` ,

независимо от того, насколько глубоко он вложен:

```
$ echo **/deep*
folder/anotherfolder/content/deepfolder folder/subfolder/content/deepfolder
```

## ? шарик

### подготовка

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

`?` просто соответствует одному символу

```
$ echo ?acy
macy
$ echo ??acy
stacy tracy
```

## [] Glob

### подготовка

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Если необходимо сопоставить определенные символы, тогда можно использовать «[]». Любой символ внутри '[]' будет сопоставляться ровно один раз.

```
$ echo [m]acy
macy
$ echo [st][tr]acy
stacy tracy
```

`[]` Glob, однако, более универсален, чем просто. Он также допускает отрицательное совпадение и даже совпадение диапазонов символов и характеристик. Отрицательный результат достигается при использовании `!` или `^` как первый символ, следующий за `[` . Мы можем соответствовать `stacy` по

```
$ echo [!t][^r]acy
stacy
```

Здесь мы говорим bash, что хотим сопоставлять только файлы, которые не начинаются с `t` а вторая буква не является `r` а файл заканчивается `acy` .

Диапазоны могут быть сопоставлены путем разделения пары символов с дефисом ( `-` ). Любой символ, который попадает между этими двумя охватывающими символами - включительно - будет соответствовать. Например, `[rt]` эквивалентно `[rst]`

```
$ echo [r-t][r-t]acy
stacy tracy
```

Классы символов могут быть сопоставлены с помощью `[:class:]` , например, чтобы соответствовать файлам, содержащим пробелы

```
$ echo *[[:blank:]]*
file with space
```

## Сопряжение скрытых файлов

### подготовка

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Встроенная опция *dotglob Bash* позволяет сопоставлять скрытые файлы и папки, то есть файлы и папки, начинающиеся с .

```
$ shopt -s dotglob
$ echo *
file with space folder .hiddenfile macy stacy tracy
```

## Нечувствительность к регистру

**подготовка**

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Установка параметра `nocaseglob` будет соответствовать `nocaseglob` нечувствительным к регистру

```
$ echo M*
M*
$ shopt -s nocaseglob
$ echo M*
macy
```

## Поведение, когда glob ничего не соответствует

**подготовка**

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Если glob ничего не соответствует, результат определяется опциями `nullglob` и `failglob` . Если ни один из них не задан, Bash вернет сам глобус, если ничего не согласовано

```
$ echo no*match
no*match
```

Если `nullglob` активирован, то ничего ( `null` ) не возвращается:

```
$ shopt -s nullglob
$ echo no*match

$
```

Если `failglob` активирован, возвращается сообщение об ошибке:

```
$ shopt -s failglob
$ echo no*match
bash: no match: no*match
$
```

Обратите внимание, что параметр `failglob` заменяет параметр `nullglob` , т. `nullglob` Если `nullglob` и `failglob` оба установлены, тогда - в случае отсутствия соответствия - возвращается ошибка.

## Расширенное подтягивание

2,02

**подготовка**

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

`extglob` опция `extglob` позволяет расширить возможности glob

```
shopt -s extglob
```

Следующие под-шаблоны содержат действительные расширенные глобусы:

- `?(pattern-list)` - Соответствует нулю или одному вхождению данных шаблонов
- `*(pattern-list)` - соответствует нулю или больше вхождений данных шаблонов
- `+(pattern-list)` - соответствует одному или нескольким вхождениям данных шаблонов
- `@(pattern-list)` - соответствует одному из заданных шаблонов
- `!(pattern-list)` - Совпадает со всем, кроме одного из заданных шаблонов

Список `pattern-list` - это список глобусов, разделенных символом `|` ,

```
$ echo *([r-t])acy
stacy tracy

$ echo *([r-t]|m)acy
macy stacy tracy

$ echo ?([a-z])acy
macy
```

Сам `pattern-list` может быть другим, вложенным расширенным глобусом. В приведенном выше примере мы видели, что мы можем сопоставить `tracy` и `stacy` с `*(rt)` . Этот расширенный сам глобус можно использовать внутри отрицаемого расширенного глоба `!(pattern-list)` , чтобы соответствовать `macy`

```
$ echo !(*([r-t]))acy
macy
```

Он соответствует любому, что **не** начинается с нуля или более вхождений букв `r` , `s` и `t` , что оставляет только `macy` как возможное совпадение.

## Соответствие регулярных выражений

```
pat='[^0-9]+([0-9]+)'
s='I am a string with some digits 1024'
[[ $s =~ $pat ]] # $pat must be unquoted
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
```

Выход:

```
I am a string with some digits 1024
1024
```

Вместо назначения регулярного выражения переменной ( `$pat` ) мы могли бы также:

```
[[ $s =~ [^0-9]+([0-9]+) ]]
```

**объяснение**

- Конструкция `[[ $s =~ $pat ]]` выполняет сопоставление регулярных выражений
- Захваченные группы, т. *E.* Результаты совпадения, доступны в массиве с именем *BASH_REMATCH*
- 0-й индекс в массиве *BASH_REMATCH* - это полное совпадение
- *I-й* индекс в массиве *BASH_REMATCH* - это *i-я* захваченная группа, где *i* = 1, 2, 3 ...

## Получить захваченные группы из регулярного выражения для строки

```
a='I am a simple string with digits 1234'
pat='(.*) ([0-9]+)'
[[ "$a" =~ $pat ]]
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
echo "${BASH_REMATCH[2]}"
```

Выход:

```
I am a simple string with digits 1234
I am a simple string with digits
1234
```

Прочитайте Совпадение шаблонов и регулярные выражения онлайн:
https://riptutorial.com/ru/bash/topic/3795/совпадение-шаблонов-и-регулярные-выражения

# глава 59: Создание каталогов

## Вступление

Манипулирование каталогами из командной строки

## Examples

### Переместить все файлы, которые уже не находятся в каталоге, в папку с именем пользователя

ll | grep ^ - | awk -F "." '{print $ 2 "." $ 3} '| awk -F ":" '{print $ 2}' | awk '{$ 1 = ""; print $ 0} '| cut -c2- | awk -F "." '{print "mkdir" "$ 1" "; mv" "$ 1". "$ 2" "" "$ 1" ""}'> tmp; source tmp

Прочитайте Создание каталогов онлайн: https://riptutorial.com/ru/bash/topic/8168/создание-каталогов

# глава 60: со-процессы

## Examples

### Привет, мир

```
# create the co-process
coproc bash

# send a command to it (echo a)
echo 'echo Hello World' >&"${COPROC[1]}"

# read a line from its output
read line <&"${COPROC[0]}"

# show the line
echo "$line"
```

Результатом является «Hello World».

Прочитайте со-процессы онлайн: https://riptutorial.com/ru/bash/topic/6933/со-процессы

# глава 61: Список файлов

## Синтаксис

- ls [OPTION] ... [FILE] ...

## параметры

| вариант | Описание |
|---|---|
| `-a` , `--all` | Перечислите все записи, включая те, которые начинаются с точки |
| `-A` , - `--almost-all` | Перечислите все записи, кроме `.` и `..` |
| `-c` | Сортировка файлов по времени изменения |
| `-d` , `--directory` | Список записей в каталоге |
| `-h` , `--human-readable` | Показывать размеры в формате, читаемом человеком (например, `K` , `M` ) |
| `-H` | То же, что и выше, только с мощностью 1000 вместо 1024 |
| `-l` | Показать содержимое в формате long-listing |
| `-o` | Формат длинного списка без информации о группе |
| `-r` , `--reverse` | Показать содержимое в обратном порядке |
| `-s` , `--size` | Размер печати каждого файла в блоках |
| `-S` | Сортировать по размеру файла |
| `--sort=WORD` | Сортировка содержимого по слову. (т.е. размер, версия, статус) |
| `-t` | Сортировать по времени модификации |
| `-u` | Сортировать по времени последнего доступа |
| `-v` | Сортировать по версии |
| `-1` | Перечислить один файл на строку |

## Examples

## Список файлов

Команда `ls` отображает содержимое указанного каталога, **исключая** dotfiles. Если каталог не указан, то по умолчанию указывается содержимое текущего каталога.

Перечисленные файлы сортируются по алфавиту по умолчанию и выравниваются по столбцам, если они не помещаются в одну строку.

```
$ ls
apt  configs  Documents  Fonts    Music      Programming  Templates  workspace
bin  Desktop  eclipse    git      Pictures   Public       Videos
```

## Список файлов в формате Long Listing

Параметр `-l` команды `ls` печатает содержимое указанного каталога в формате длинного списка. Если каталог не указан, то по умолчанию указывается содержимое текущего каталога.

```
ls -l /etc
```

Пример:

```
total 1204
drwxr-xr-x  3 root root    4096 Apr 21 03:44 acpi
-rw-r--r--  1 root root    3028 Apr 21 03:38 adduser.conf
drwxr-xr-x  2 root root    4096 Jun 11 20:42 alternatives
...
```

На выходе сначала отображается `total` , которое указывает общий размер в **блоках** всех файлов в указанном каталоге. Затем он отображает восемь столбцов информации для каждого файла в указанном каталоге. Ниже приведены данные для каждого столбца на выходе:

| Номер столбца | пример | Описание |
|---|---|---|
| 1,1 | `d` | Тип файла (см. Таблицу ниже) |
| 1.2 | `rwxr-xr-x` | Строка разрешения |
| 2 | `3` | Количество жестких ссылок |
| 3 | `root` | Имя владельца |
| 4 | `root` | Группа владельцев |
| 5 | `4096` | Размер файла в байтах |

| Номер столбца | пример | Описание |
|---|---|---|
| 6 | `Apr 21 03:44` | Время модификации |
| 7 | `acpi` | Имя файла |

# Тип файла

Тип файла может быть одним из следующих символов.

| символ | Тип файла |
|---|---|
| `-` | Обычный файл |
| `b` | Блокировать специальный файл |
| `c` | Специальный файл символов |
| `C` | Файл с высокой производительностью («смежные данные») |
| `d` | каталог |
| `D` | Дверь (специальный IPC-файл только в Solaris 2.5+) |
| `l` | Символическая ссылка |
| `M` | Офлайновый («перенесенный») файл (Cray DMF) |
| `n` | Сетевой специальный файл (HP-UX) |
| `p` | FIFO (названный канал) |
| `P` | Порт (специальный системный файл только в Solaris 10+) |
| `s` | Разъем |
| `?` | Некоторые другие типы файлов |

## Список файлов, отсортированных по размеру

Параметр `-S` команды `ls` сортирует файлы в порядке убывания размера файла.

```
$ ls -l -S ./Fruits
total 444
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
```

```
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
```

При использовании с параметром `-r` порядок сортировки меняется на противоположный.

```
$ ls -l -S -r /Fruits
total 444
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
```

## Список файлов без использования `ls`

Используйте расширения расширения и расширения файлов оболочки Bash для получения имен файлов:

```
# display the files and directories that are in the current directory
printf "%s\n" *

# display only the directories in the current directory
printf "%s\n" */

# display only (some) image files
printf "%s\n" *.{gif,jpg,png}
```

Чтобы захватить список файлов в переменную для обработки, обычно рекомендуется использовать массив bash :

```
files=( * )

# iterate over them
for file in "${files[@]}"; do
    echo "$file"
done
```

## Список десяти последних измененных файлов

Ниже перечислены до десяти последних измененных файлов в текущем каталоге, используя длинный формат списка ( `-l` ) и отсортированный по времени ( `-t` ).

```
ls -lt | head
```

## Список всех файлов, включая Dotfiles

**Dotfile** - это файл, имена которого начинаются с символа `.` , Они обычно скрыты `ls` и не перечисляются, если только не запрашиваются.

Например, следующий вывод `ls` :

```
$ ls
```

```
bin  pki
```

`-a` или `--all` вариант будет список всех файлов, в том числе составляют скрытые.

```
$ ls -a
.   .ansible       .bash_logout   .bashrc  .lesshst  .puppetlabs  .viminfo
..  .bash_history  .bash_profile  bin      pki       .ssh
```

Параметр `-A` или `--almost-all` будет отображать все файлы, включая dotfiles, но не указывать подразумеваемые `.` и `..` Обратите внимание, что `.` это текущий каталог и `..` является родительским каталогом.

```
$ ls -A
.ansible       .bash_logout   .bashrc  .lesshst  .puppetlabs  .viminfo
.bash_history  .bash_profile  bin      pki       .ssh
```

## Список файлов в древовидном формате

Команда `tree` выводит содержимое указанного каталога в древовидном формате. Если каталог не указан, то по умолчанию указывается содержимое текущего каталога.

Пример:

```
$ tree /tmp
/tmp
├── 5037
├── adb.log
└── evince-20965
    └── image.FPWTJY.png
```

Используйте параметр `-L` команды `tree` , чтобы ограничить глубину отображения и параметр `-d` только списком каталогов.

Пример:

```
$ tree -L 1 -d /tmp
/tmp
└── evince-20965
```

Прочитайте Список файлов онлайн: https://riptutorial.com/ru/bash/topic/366/список-файлов

# глава 62: Тип корпуса

## замечания

### Войти в Shell

Оболочка для входа - это тот, чей первый символ аргумента нуль - -, или один начинается с параметра -login. Инициализация более полная, чем в обычной интерактивной (суб) оболочке.

### Интерактивная оболочка

Интерактивная оболочка запускается без аргументов без опционов и без опции -c, стандартный ввод и ошибка которой соединены с терминалами (как определено isatty (3)), или один начинается с опции -i. PS1 установлен, а $ - включает i, если bash является интерактивным, позволяя сценарию оболочки или загрузочному файлу проверить это состояние.

### неинтерактивная оболочка

Неинтерактивная оболочка - это оболочка, в которой пользователь не может взаимодействовать с оболочкой. Например, оболочка, запускающая скрипт, всегда является неинтерактивной оболочкой. Тем не менее, скрипт может получить доступ к его tty.

### Настройка оболочки входа

При входе в систему:

```
If '/etc/profile' exists, then source it.
If '~/.bash_profile' exists, then source it,
else if '~/.bash_login' exists, then source it,
else if '~/.profile' exists, then source it.
```

### Для интерактивных оболочек без входа

При запуске:

```
If `~/.bashrc' exists, then source it.
```

### Для неинтерактивных оболочек

При запуске: Если переменная окружения ENV не равна null, разверните переменную и введите файл с именем по значению. Если Bash не запускается в режиме Posix, он ищет BASH_ENV до ENV.

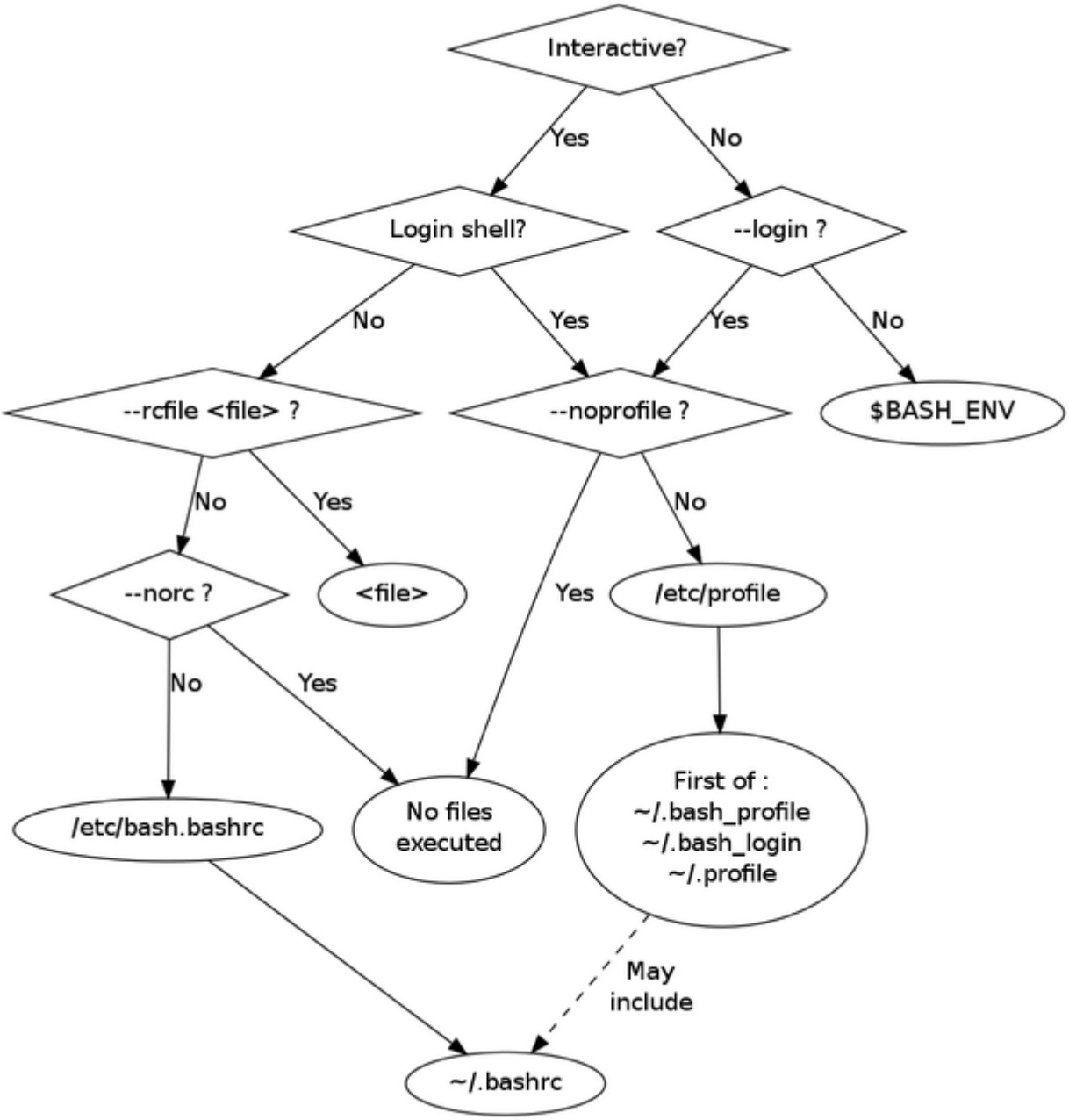
# Examples

## Введение в файлы точек

В Unix файлы и каталоги, начинающиеся с периода, обычно содержат настройки для конкретной программы / серии программ. Файлы Dot обычно скрыты от пользователя, поэтому вам нужно будет запустить `ls -a` чтобы увидеть их.

Примером точечного файла является `.bash_history` , который содержит последние выполненные команды, предполагая, что пользователь использует Bash.

Существуют различные файлы, которые получены при падении в оболочку Bash. На изображении ниже, взятом с этого сайта , показан процесс принятия решения о выборе файлов для запуска при запуске.



### Начать интерактивную оболочку

```
bash
```

### Определить тип оболочки

```
shopt -q login_shell && echo 'login' || echo 'not-login'
```

Прочитайте Тип корпуса онлайн: https://riptutorial.com/ru/bash/topic/6517/тип-корпуса

# глава 63: Трассирование

## Синтаксис

- strace -c [df] [-In] [-bexecve] [-eexpr] ... [-Ooverhead] [-Ssortby] -ppid ... / [-D] [-Evar [= val]] ... [-uusername] [args]

## Examples

### Как наблюдать за системными вызовами программы

Для *исполняемого файла или команды* exec, при запуске этого процесса будут перечислены все системные вызовы:

```
$ ptrace exec
```

Для отображения конкретных системных вызовов используйте параметр -e:

```
$ strace -e open exec
```

Для сохранения вывода в файл используйте опцию -o:

```
$ strace -o output exec
```

Чтобы найти системные вызовы, используемые активной программой, используйте параметр -p при указании pid [как получить pid] :

```
$ sudo strace -p 1115
```

Чтобы создать статистический отчет всех используемых системных вызовов, используйте параметр -c:

```
$ strace -c exec
```

Прочитайте Трассирование онлайн: https://riptutorial.com/ru/bash/topic/10855/трассирование

# глава 64: Трубопроводы

## Синтаксис

- [time [-p]] [!] command1 [| или | & command2] ...

## замечания

Конвейер - это последовательность простых команд, разделенных одним из управляющих операторов `|` или `|&` ( источник ).

`|` соединяет выход `command1` на вход `command2` .

`|&` соединяет стандартный вывод и стандартную ошибку `command1` со стандартным вводом `command2` .

## Examples

### Показать все процессы с разбивкой по страницам

```
ps -e | less
```

`ps -e` показывает все процессы, его выход подключен к входу больше через `|` , `less` paginates результатов.

### Используя | &

`|&` соединяет стандартный вывод и стандартную ошибку первой команды ко второй, а `|` только подключает стандартный вывод первой команды ко второй команде.

В этом примере страница загружается через `curl` . с `-v` опция `curl` записывает некоторую информацию о `stderr` включая загруженную страницу, записанную на `stdout` . Заголовок страницы можно найти между `<title>` и `</title>` .

```
curl -vs 'http://www.google.com/' |& awk '/Host:/{print}
/<title>/{match($0,/<title>(.*)<\/title>/,a);print a[1]}'
```

Выход:

```
> Host: www.google.com
Google
```

Но с `|` будет напечатано намного больше информации, то есть тех, которые отправляются

в `stderr` потому что только `stdout` передается по следующей команде. В этом примере все строки, кроме последней строки (Google), были отправлены в `stderr` помощью `curl` :

```
* Hostname was NOT found in DNS cache
*   Trying 172.217.20.228...
* Connected to www.google.com (172.217.20.228) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: www.google.com
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Date: Sun, 24 Jul 2016 19:04:59 GMT
< Expires: -1
< Cache-Control: private, max-age=0
< Content-Type: text/html; charset=ISO-8859-1
< P3P: CP="This is not a P3P policy! See
https://www.google.com/support/accounts/answer/151657?hl=en for more info."
< Server: gws
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: NID=82=jX0yZLPPUE7u13kKNevUCDg8yG9Ze_C03o0IM-
EopOSKL0mMITEagIE816G55L2wrTlQwgXkhq4ApFvvYEoaWF-
oEoq2T0sBTuQVdsIFULj9b2O8X35O0sAgUnc3a3JnTRBqelMcuS9QkQA; expires=Mon, 23-Jan-2017 19:04:59
GMT; path=/; domain=.google.com; HttpOnly
< Accept-Ranges: none
< Vary: Accept-Encoding
< X-Cache: MISS from jetsib_appliance
< X-Loop-Control: 5.202.190.157 81E4F9836653D5812995BA53992F8065
< Connection: close
<
{ [data not shown]
* Closing connection 0
Google
```

## Изменение непрерывного вывода команды

```
~$ ping -c 1 google.com # unmodified output
PING google.com (16.58.209.174) 56(84) bytes of data.
64 bytes from wk-in-f100.1e100.net (16.58.209.174): icmp_seq=1 ttl=53 time=47.4 ms
~$ ping google.com | grep -o '^[0-9]\+[^()]\+' # modified output
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
...
```

Труба ( `|` ) соединяет `stdout ping` с `stdin grep` , который обрабатывает его немедленно. Некоторые другие команды, такие как `sed` default, для буферизации их `stdin` , что означает, что он должен получать достаточно данных, прежде чем он что-то распечатает, что может

вызвать задержки при дальнейшей обработке.

Прочитайте Трубопроводы онлайн: https://riptutorial.com/ru/bash/topic/5485/трубопроводы

# глава 65: Управление заданиями

## Синтаксис

- long_cmd &
- работы
- fg% JOB_ID
- fg%? PATTERN
- fg% JOB_ID

## Examples

### Запустить команду в фоновом режиме

```
$ sleep 500 &
[1] 7582
```

Помещает команду sleep в фоновом режиме. 7582 - это идентификатор процесса фонового процесса.

### Список фоновых процессов

```
$ jobs
[1]   Running                 sleep 500 &   (wd: ~)
[2]-  Running                 sleep 600 &   (wd: ~)
[3]+  Running                 ./Fritzing &
```

В первом поле отображаются идентификаторы работы. Знак + и -, который следует за идентификатором задания для двух заданий, обозначает задание по умолчанию и следующее задание по умолчанию кандидата, когда текущее задание по умолчанию заканчивается соответственно. Задание по умолчанию используется, когда команды `fg` или `bg` используются без каких-либо аргументов.

Второе поле дает статус задания. Третье поле - это команда, используемая для запуска процесса.

В последнем поле (wd: ~) говорится, что команды сна были запущены из рабочего каталога ~ (Главная).

### Привести фоновый процесс на передний план

```
$ fg %2
sleep 600
```

% 2 указывает номер задания. 2. Если fg используется без каких-либо аргументов, если последний процесс помещается в фоновом режиме на передний план.

```
$ fg %?sle
sleep 500
```

`?sle` относится к команде процесса «baground», содержащей «sle». Если несколько фоновых команд содержат строку, это приведет к ошибке.

### Остановить процесс переднего плана

Нажмите Ctrl + Z, чтобы остановить процесс переднего плана и поместить его в фоновом режиме.

```
$ sleep 600
^Z
[8]+  Stopped                 sleep 600
```

### Перезапустить остановленный фоновый процесс

```
$ bg
[8]+ sleep 600 &
```

Прочитайте Управление заданиями онлайн: https://riptutorial.com/ru/bash/topic/5193/управление-заданиями

# глава 66: Управление переменной среды PATH

## Синтаксис

- Добавить путь: PATH = $ PATH: / new / path
- Добавить путь: PATH = / новый / путь: $ PATH

## параметры

| параметр | подробности |
| --- | --- |
| ДОРОЖКА | Переменная окружения |

## замечания

**Файл конфигурации Bash:**

Этот файл создается каждый раз, когда запускается новая интерактивная оболочка Bash.

В системах GNU / Linux это обычно файл ~ / .bashrc; в Mac это ~ / .bash_profile или ~ / .profile

**Экспорт:**

Переменная PATH должна быть экспортирована один раз (это делается по умолчанию). После его экспорта он будет экспортирован и любые внесенные в него изменения будут немедленно применены.

**Применять изменения:**

Чтобы применить изменения к файлу конфигурации Bash, вы должны перезагрузить этот файл в терминале ( `source /path/to/bash_config_file` )

## Examples

### Добавить путь к переменной среды PATH

Переменная среды PATH обычно определяется в ~ / .bashrc или ~ / .bash_profile или / etc / profile или ~ / .profile или /etc/bash.bashrc (файл конфигурации Bash для дистрибутива)

```
$ echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr
```

Теперь, если мы хотим добавить путь (например, `~/bin` ) к переменной PATH:

```
PATH=~/bin:$PATH
# or
PATH=$PATH:~/bin
```

Но это изменит PATH только в текущей оболочке (и ее подоболочке). Как только вы выйдете из оболочки, эта модификация исчезнет.

Чтобы сделать его постоянным, нам нужно добавить этот бит кода в файл ~ / .bashrc (или любой другой) и перезагрузить файл.

Если вы запустите следующий код (в терминале), он добавит `~/bin` в PATH навсегда:

```
echo 'PATH=~/bin:$PATH' >> ~/.bashrc && source ~/.bashrc
```

Объяснение:

- `echo 'PATH=~/bin:$PATH' >> ~/.bashrc` добавляет строку `PATH=~/bin:$PATH` в конце файла ~ / .bashrc (вы можете сделать это с помощью текстового редактора)
- `source ~/.bashrc` перезагружает файл ~ / .bashrc

---

Это немного кода (запускается в терминале), который проверяет, включен ли путь и добавляет путь, только если нет:

```
path=~/bin            # path to be included
bashrc=~/.bashrc      # bash file to be written and reloaded
# run the following code unmodified
echo $PATH | grep -q "\(^\|:\)$path\(:\|/\{0,1\}$\)" || echo "PATH=\$PATH:$path" >> "$bashrc";
source "$bashrc"
```

## Удалить путь из переменной среды PATH

Чтобы удалить PATH из переменной среды PATH, вам необходимо отредактировать файл ~ / .bashrc или ~ / .bash_profile или / etc / profile или ~ / .profile или /etc/bash.bashrc (файл с дистрибутивом) и удалить назначение для этот конкретный путь.

Вместо того, чтобы находить точное задание, вы можете просто сделать замену в `$PATH` на своем последнем этапе.

Следующее безопасно удалит `$path` из `$PATH` :

```
path=~/bin
PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
```

Чтобы сделать его постоянным, вам нужно добавить его в конец вашего конфигурационного файла bash.

---

Вы можете сделать это функционально:

```
rpath(){
    for path in "$@";do
        PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
    done
    echo "$PATH"
}

PATH="$(rpath ~/bin /usr/local/sbin /usr/local/bin)"
PATH="$(rpath /usr/games)"
# etc ...
```

Это упростит обработку нескольких путей.

**Заметки:**

- Вам нужно будет добавить эти коды в конфигурационный файл Bash (~ / .bashrc или что-то еще).
- Запустите `source ~/.bashrc` чтобы перезагрузить файл конфигурации Bash (~ / .bashrc).

Прочитайте Управление переменной среды PATH онлайн:
https://riptutorial.com/ru/bash/topic/5515/управление-переменной-среды-path

# глава 67: Условные выражения

## Синтаксис

- [[-OP $ filename]]
- [[$ file1 -OP $ file2]]
- [[-z $ string]]
- [[-n $ string]]
- [["$ string1" == "$ string2"]]
- [["$ string1" == $ pattern]]

## замечания

Синтаксис `[[ … ]]` окружает встроенные условные выражения bash. Обратите внимание, что с обеих сторон скобок требуются пробелы.

Условные выражения могут использовать унарные и двоичные операторы для проверки свойств строк, целых чисел и файлов. Они также могут использовать логические операторы `&&` , `||` и `!` ,

## Examples

### Сравнение файлов

```
if [[ $file1 -ef $file2 ]]; then
  echo "$file1 and $file2 are the same file"
fi
```

«Тот же файл» означает, что изменение одного из файлов на месте влияет на другое. Два файла могут быть одинаковыми, даже если они имеют разные имена, например, если они являются жесткими ссылками или являются символическими ссылками с одной и той же целью, или если это символическая ссылка, указывающая на другую.

Если два файла имеют один и тот же контент, но они представляют собой разные файлы (так что их изменение не влияет на другое), то `-ef` сообщает о них как о разных. Если вы хотите сравнить два байта по байтам, используйте утилиту `cmp` .

```
if cmp -s -- "$file1" "$file2"; then
  echo "$file1 and $file2 have identical contents"
else
  echo "$file1 and $file2 differ"
fi
```

Чтобы создать удобочитаемый список различий между текстовыми файлами, используйте

утилиту `diff` .

```
if diff -u "$file1" "$file2"; then
  echo "$file1 and $file2 have identical contents"
else
  : # the differences between the files have been listed
fi
```

## Тесты доступа к файлам

```
if [[ -r $filename ]]; then
  echo "$filename is a readable file"
fi
if [[ -w $filename ]]; then
  echo "$filename is a writable file"
fi
if [[ -x $filename ]]; then
  echo "$filename is an executable file"
fi
```

Эти тесты учитывают разрешения и права собственности, чтобы определить, может ли сценарий (или программы, запущенные из сценария) получить доступ к файлу.

Остерегайтесь условий гонки (TOCTOU) : только потому, что тест преуспевает, теперь не означает, что он все еще действует на следующей строке. Обычно лучше пытаться получить доступ к файлу и обрабатывать ошибку, а не сначала тестировать, а затем обрабатывать ошибку в любом случае, если файл изменился за это время.

## Числовые сравнения

В числовых сравнениях используются операторы `-eq` и друзья

```
if [[ $num1 -eq $num2 ]]; then
  echo "$num1 == $num2"
fi
if [[ $num1 -le $num2 ]]; then
  echo "$num1 <= $num2"
fi
```

Существует шесть числовых операторов:

- `-eq` равно
- `-ne`
- `-le` меньше или равно
- `-lt` меньше, чем
- `-ge` больше или равно
- `-gt` больше, чем

Обратите внимание, что операторы `<` и `>` внутри `[[ … ]]` сравнивают строки, а не числа.

```
if [[ 9 -lt 10 ]]; then
  echo "9 is before 10 in numeric order"
fi
if [[ 9 > 10 ]]; then
  echo "9 is after 10 in lexicographic order"
fi
```

Обе стороны должны быть числами, записанными в десятичной форме (или в восьмеричном с начальным нулем). В качестве альтернативы используйте синтаксис арифметических выражений `((…))` , который выполняет **целочисленные** вычисления в синтаксисе C / Java / ....

```
x=2
if ((2*x == 4)); then
  echo "2 times 2 is 4"
fi
((x += 1))
echo "2 plus 1 is $x"
```

## Сравнение строк и сопоставление

Сравнение строк использует оператор `==` между *цитируемыми* строками. Оператор `!=` Отрицает сравнение.

```
if [[ "$string1" == "$string2" ]]; then
  echo "\$string1 and \$string2 are identical"
fi
if [[ "$string1" != "$string2" ]]; then
  echo "\$string1 and \$string2 are not identical"
fi
```

Если правая часть не цитируется, то это шаблон подстановки, сопоставляемый с `$string1` .

```
string='abc'
pattern1='a*'
pattern2='x*'
if [[ "$string" == $pattern1 ]]; then
  # the test is true
  echo "The string $string matches the pattern $pattern"
fi
if [[ "$string" != $pattern2 ]]; then
  # the test is false
  echo "The string $string does not match the pattern $pattern"
fi
```

Операторы `<` и `>` сравнивают строки в лексикографическом порядке (для строк нет операторов с меньшим или равным или большим или равным).

Для пустой строки есть унарные тесты.

```
if [[ -n "$string" ]]; then
  echo "$string is non-empty"
```

```
fi
if [[ -z "${string// }" ]]; then
  echo "$string is empty or contains only spaces"
fi
if [[ -z "$string" ]]; then
  echo "$string is empty"
fi
```

Выше, проверка `-z` может означать, что `$string` не установлена или установлена пустая строка. Чтобы различать пустые и неустановленные, используйте:

```
if [[ -n "${string+x}" ]]; then
    echo "$string is set, possibly to the empty string"
fi
if [[ -n "${string-x}" ]]; then
    echo "$string is either unset or set to a non-empty string"
fi
if [[ -z "${string+x}" ]]; then
    echo "$string is unset"
fi
if [[ -z "${string-x}" ]]; then
    echo "$string is set to an empty string"
fi
```

где `x` произвольно. Или в виде таблицы :

```
                       +-------+-------+-----------+
            $string is: | unset | empty | non-empty |
+----------------------+-------+-------+-----------+
| [[ -z ${string} ]]   | true  | true  | false     |
| [[ -z ${string+x} ]] | true  | false | false     |
| [[ -z ${string-x} ]] | false | true  | false     |
| [[ -n ${string} ]]   | false | false | true      |
| [[ -n ${string+x} ]] | false | true  | true      |
| [[ -n ${string-x} ]] | true  | false | true      |
+----------------------+-------+-------+-----------+
```

В качестве альтернативы , состояние можно проверить в случае:

```
case ${var+x$var} in
  (x) echo empty;;
  ("") echo unset;;
  (x*[![:blank:]]*) echo non-blank;;
  (*) echo blank
esac
```

Где `[:blank:]` - это символы горизонтального интервала, специфичные для локали (вкладка, пробел и т. Д.).

## Тестирование типа файла

`-e` условный оператор проверяет, существует ли файл (включая все типы файлов: каталоги и т. Д.).

```
if [[ -e $filename ]]; then
  echo "$filename exists"
fi
```

Существуют также тесты для определенных типов файлов.

```
if [[ -f $filename ]]; then
  echo "$filename is a regular file"
elif [[ -d $filename ]]; then
  echo "$filename is a directory"
elif [[ -p $filename ]]; then
  echo "$filename is a named pipe"
elif [[ -S $filename ]]; then
  echo "$filename is a named socket"
elif [[ -b $filename ]]; then
  echo "$filename is a block device"
elif [[ -c $filename ]]; then
  echo "$filename is a character device"
fi
if [[ -L $filename ]]; then
  echo "$filename is a symbolic link (to any file type)"
fi
```

Для символической ссылки, кроме `-L` , эти тесты применяются к цели и возвращают false для неработающей ссылки.

```
if [[ -L $filename || -e $filename ]]; then
  echo "$filename exists (but may be a broken symbolic link)"
fi

if [[ -L $filename && ! -e $filename ]]; then
  echo "$filename is a broken symbolic link"
fi
```

## Проверка состояния выхода команды

Статус выхода 0: успех
Состояние выхода, отличное от 0: сбой

Чтобы проверить состояние выхода команды:

```
if command;then
    echo 'success'
else
    echo 'failure'
fi
```

## Один тест на гильзе

Вы можете делать такие вещи:

```
[[ $s = 'something' ]] && echo 'matched' || echo "didn't match"
```

```
[[ $s == 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s != 'something' ]] && echo "didn't match" || echo "matched"
[[ $s -eq 10 ]] && echo 'equal' || echo "not equal"
(( $s == 10 )) && echo 'equal' || echo 'not equal'
```

Один тест линейки для статуса выхода:

```
command && echo 'exited with 0' || echo 'non 0 exit'
cmd && cmd1 && echo 'previous cmds were successful' || echo 'one of them failed'
cmd || cmd1 #If cmd fails try cmd1
```

Прочитайте Условные выражения онлайн: https://riptutorial.com/ru/bash/topic/731/условные-выражения

# глава 68: Утилита сна

## Вступление

Команда Sleep может использоваться для паузы в течение заданного времени.

Если вы хотите использовать разные входные данные, используйте следующие секунды: $ sleep 1s (секунды по умолчанию) Минуты: $ sleep 1m Часы: $ sleep 1h days: $ sleep 1d

Если вы хотите спать менее одной секунды, используйте $ sleep 0.5. Вы можете использовать, как указано выше, в соответствии с вашими потребностями.

## Examples

### $ sleep 1

Здесь процесс, инициированный этим вызовом, будет спать в течение 1 секунды.

Прочитайте Утилита сна онлайн: https://riptutorial.com/ru/bash/topic/10879/утилита-сна

# глава 69: функции

## Синтаксис

- Определите функцию с ключевым словом `function` :

```
function f {
```

}

- Определить функцию с `()` :

```
f(){
```

}

- Определите функцию с ключевым словом `function` и `()` :

```
function f(){
```

}

## Examples

### Простая функция

`helloWorld.sh`

```
#!/bin/bash

# Define a function greet
greet ()
{
    echo "Hello World!"
}

# Call the function greet
greet
```

При запуске скрипта мы видим наше сообщение

```
$ bash helloWorld.sh
Hello World!
```

*Обратите внимание, что* поиск файла с функциями делает их доступными в текущем сеансе bash.

```
$ source helloWorld.sh   # or, more portably, ". helloWorld.sh"
$ greet
Hello World!
```

Вы можете `export` функцию в некоторые оболочки, чтобы она подвергалась дочерним процессам.

```
bash -c 'greet'  # fails
export -f greet  # export function; note -f
bash -c 'greet'  # success
```

### Функции с аргументами

В `helloJohn.sh` :

```
#!/bin/bash

greet() {
  local name="$1"
  echo "Hello, $name"
}

greet "John Doe"
```

```
# running above script
$ bash helloJohn.sh
Hello, John Doe
```

1. Если вы каким-либо образом не изменяете аргумент, нет необходимости копировать его в `local` переменную - просто `echo "Hello, $1"` .

2. Вы можете использовать `$1` , `$2` , `$3` и т. Д., Чтобы получить доступ к аргументам внутри функции.

   > **Примечание:** для аргументов более 9 `$10` не будет работать (bash будет читать его как **$ 1** 0), вам нужно сделать `${10}` , `${11}` и так далее.

3. `$@` ссылается на все аргументы функции:

   ```
   #!/bin/bash
   foo() {
     echo "$@"
   }

   foo 1 2 3 # output => 1 2 3
   ```

   > **Примечание.** Вы всегда должны использовать двойные кавычки вокруг `"$@"` , как здесь.

   Опускание кавычек заставит оболочку расширять подстановочные знаки (даже если

пользователь специально ссылался на них, чтобы избежать этого) и вообще вводит нежелательное поведение и потенциально даже проблемы с безопасностью.

```
foo "string with spaces;" '$HOME' "*"
# output => string with spaces; $HOME *
```

4. для аргументов по умолчанию используйте `${1:-default_val}` . Например:

```
#!/bin/bash
foo() {
  local val=${1:-25}
  echo "$val"
}

foo     # output => 25
foo 30  # output => 30
```

5. для запроса аргумента используйте сообщение `${var:?error message}`

```
foo() {
  local val=${1:?Must provide an argument}
  echo "$val"
}
```

## Возвращаемое значение из функции

Оператор `return` в Bash не возвращает значение, подобное C-функциям, вместо этого он выходит из функции с возвратом. Вы можете думать о нем как о статусе выхода этой функции.

Если вы хотите вернуть значение из функции, отправьте значение в `stdout` следующим образом:

```
fun() {
    local var="Sample value to be returned"
    echo "$var"
    #printf "%s\n" "$var"
}
```

Теперь, если вы это сделаете:

```
var="$(fun)"
```

выход `fun` будет храниться в `$var` .

## Обработка флагов и дополнительных параметров

*Встроенные* функции *getopts* могут использоваться внутри функций для записи функций, которые содержат флаги и необязательные параметры. Это не представляет особых

трудностей, но нужно должным образом обрабатывать ценности, затронутые *getopts* . В качестве примера мы определяем функцию *сбоя,* которая записывает сообщение на *stderr* и выходит с кодом 1 или произвольным кодом, поставляемым как параметр опции `-x` :

```
# failwith [-x STATUS] PRINTF-LIKE-ARGV
#  Fail with the given diagnostic message
#
# The -x flag can be used to convey a custom exit status, instead of
# the value 1.  A newline is automatically added to the output.

failwith()
{
    local OPTIND OPTION OPTARG status

    status=1
    OPTIND=1

    while getopts 'x:' OPTION; do
        case ${OPTION} in
            x)    status="${OPTARG}";;
            *)    1>&2 printf 'failwith: %s: Unsupported option.\n' "${OPTION}";;
        esac
    done

    shift $(( OPTIND - 1 ))
    {
        printf 'Failure: '
        printf "$@"
        printf '\n'
    } 1>&2
    exit "${status}"
}
```

Эту функцию можно использовать следующим образом:

```
failwith '%s: File not found.' "${filename}"
failwith -x 70 'General internal error.'
```

и так далее.

Обратите внимание, что как и для *printf* , переменные не должны использоваться в качестве первого аргумента. Если сообщение для печати состоит из содержимого переменной, следует использовать спецификатор `%s` для его печати, например, в

```
failwith '%s' "${message}"
```

## Код выхода функции - это код выхода последней команды

Рассмотрим эту примерную функцию, чтобы проверить, находится ли хост:

```
is_alive() {
    ping -c1 "$1" &> /dev/null
}
```

Эта функция отправляет одиночный пинг на хост, указанный первым параметром функции. Выходной сигнал и вывод ошибки `ping` оба перенаправлены на `/dev/null` , поэтому функция никогда ничего не выводит. Но команда `ping` будет иметь код выхода 0 при успешном завершении и не равна нулю при сбое. Поскольку это последняя (и в этом примере единственная) команда функции, код выхода `ping` будет повторно использован для кода выхода самой функции.

Этот факт очень полезен в условных утверждениях.

Например, если хост `graucho` встал, то подключитесь к нему с помощью `ssh` :

```
if is_alive graucho; then
    ssh graucho
fi
```

Другой пример: многократно проверяйте, пока хост `graucho` не встал, а затем подключитесь к нему с помощью `ssh` :

```
while ! is_alive graucho; do
    sleep 5
done
ssh graucho
```

## Распечатайте определение функции

```
getfunc() {
    declare -f "$@"
}

function func(){
    echo "I am a sample function"
}

funcd="$(getfunc func)"
getfunc func # or echo "$funcd"
```

Выход:

```
func ()
{
    echo "I am a sample function"
}
```

## Функция, которая принимает именованные параметры

```
foo() {
  while [[ "$#" -gt 0 ]]
  do
    case $1 in
      -f|--follow)
```

```
        local FOLLOW="following"
        ;;
      -t|--tail)
        local TAIL="tail=$2"
        ;;
    esac
    shift
  done

  echo "FOLLOW: $FOLLOW"
  echo "TAIL: $TAIL"
}
```

Пример использования:

```
foo -f
foo -t 10
foo -f --tail 10
foo --follow --tail 10
```

Прочитайте функции онлайн: https://riptutorial.com/ru/bash/topic/472/функции

# глава 70: Цепь команд и операций

## Вступление

Есть несколько способов объединить команды вместе. Простые, как просто; или более сложные, такие как логические цепи, которые работают в зависимости от некоторых условий. Третий - это команды трубопроводов, которые эффективно передают выходные данные следующей команде в цепочке.

## Examples

### Подсчет текста

Использование канала делает вывод команды входным для следующего.

```
ls -1 | grep -c ".conf"
```

В этом случае вывод команды ls используется как вход команды grep. Результатом будет количество файлов, содержащих «.conf» в их имени.

Это можно использовать для конструирования цепочек последующих команд до тех пор, пока это необходимо:

```
ls -1 | grep ".conf" | grep -c .
```

### передать выходной файл cmd в файл пользователя

Часто хочется показать результат команды, выполняемой root для других пользователей. Команда **tee** позволяет легко записывать файл с пользовательским perms из команды, выполняемой с правами root:

```
su -c ifconfig | tee ~/results-of-ifconfig.txt
```

Только **ifconfig** работает от имени root.

### логическая цепочка команд с && и ||

**&&** объединяет две команды. Второй работает только в том случае, если первый из них успешно завершает работу. **||** цепи две команды. Но второй работает только в том случае, если первый из них выходит с ошибкой.

```
[ a = b  ] && echo "yes" || echo "no"
```

```
# if you want to run more commands within a logical chain, use curly braces
# which designate a block of commands
# They do need a ; before closing bracket so bash can diffentiate from other uses
# of curly braces
[ a = b ] && { echo "let me see."
               echo "hmmm, yes, i think it is true" ; } \
          || { echo "as i am in the negation i think "
               echo "this is false. a is a not b."  ; }
# mind the use of line continuation sign \
# only needed to chain yes block with || ....
```

### последовательная цепочка команд с точкой с запятой

Точка с запятой разделяет только две команды.

```
echo "i am first" ; echo "i am second" ; echo " i am third"
```

### команды цепочки с |

**|** выводит результат левой команды и передает его в качестве входной правой команды. Разумеется, это делается в подоболочке. Следовательно, вы не можете устанавливать значения варов вызывающего процесса с трубой.

```
find . -type f -a -iname '*.mp3'  | \
       while read filename; do
             mute --noise "$filename"
       done
```

Прочитайте Цепь команд и операций онлайн: https://riptutorial.com/ru/bash/topic/5589/цепь-команд-и-операций

# глава 71: Шаблоны проектирования

## Вступление

Выполнение некоторых общих шаблонов проектирования в Bash

## Examples

### Шаблон публикации / подписки (паб / суб)

Когда проект Bash превращается в библиотеку, становится сложно добавлять новые функции. Названия функций, переменные и параметры обычно должны быть изменены в сценариях, которые их используют. В таких сценариях полезно отделить код и использовать шаблон проектирования, управляемый событиями. В указанном шаблоне внешний скрипт может подписаться на событие. Когда это событие запускается (публикуется), скрипт может выполнять код, который он зарегистрировал в событии.

**pubsub.sh:**

```
#!/usr/bin/env bash

#
# Save the path to this script's directory in a global env variable
#
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

#
# Array that will contain all registered events
#
EVENTS=()

function action1() {
    echo "Action #1 was performed ${2}"
}

function action2() {
    echo "Action #2 was performed"
}

#
# @desc   :: Registers an event
# @param  :: string $1 - The name of the event. Basically an alias for a function name
# @param  :: string $2 - The name of the function to be called
# @param  :: string $3 - Full path to script that includes the function being called
#
function subscribe() {
    EVENTS+=("${1};${2};${3}")
}

#
# @desc   :: Public an event
```

```
    # @param  :: string $1 - The name of the event being published
    #
    function publish() {
        for event in ${EVENTS[@]}; do
            local IFS=";"
            read -r -a event <<< "$event"
            if [[  "${event[0]}" ==  "${1}" ]]; then
                ${event[1]} "$@"
            fi
        done
    }

    #
    # Register our events and the functions that handle them
    #
    subscribe "/do/work"           "action1" "${DIR}"
    subscribe "/do/more/work"      "action2" "${DIR}"
    subscribe "/do/even/more/work" "action1" "${DIR}"

    #
    # Execute our events
    #
    publish "/do/work"
    publish "/do/more/work"
    publish "/do/even/more/work" "again"
```

**Бежать:**

```
chmod +x pubsub.sh
./pubsub.sh
```

Прочитайте Шаблоны проектирования онлайн: https://riptutorial.com/ru/bash/topic/9531/шаблоны-проектирования

# кредиты

| S. No | Главы | Contributors |
|---|---|---|
| 1 | Начало работы с Bash | 4444, Acey, Ajay Sangale, Alessandro Mascolo, Anil, Ashari, Benjamin W., Blachshma, Bob Bagwill, Bubblepop, Burkhard, Christopher Bottoms, Colin Yang, Community, CraftedCart, Danny, Diego Torres Milano, divyum, dotancohen, fedorqui, Franck Dernoncourt, Functino, Gavyn, glenn jackman, Inanc Gumus, Ingve, intboolstring, J F, Jahid, Jean-Baptiste Yunès, JHS, Jonny Henly, l0b0, lesmana, Marek Skiba, Matt, Matt Clark, mouviciel, Nathan Arthur, niglesias, rap-2-h, Richard Hamilton, Riker, satyanarayan rao, shloosh, Shubham, sjsam, Sundeep, Sylvain Bugat, Trevor Clarke, tripleee, user1336087, William Pursell, WMios, Yuki Inoue, Zaz, |
| 2 | Aliasing | Bostjan, Daniel Käfer, dingalapadum, glenn jackman, intboolstring, janos, JHS, Neui, tripleee, uhelp |
| 3 | Bash на Windows 10 | Thomas Champion |
| 4 | getopts: интеллектуальный анализ позиционных параметров | pepoluan, sjsam |
| 5 | Grep | Chandrahas Aroori |
| 6 | Sourcing | hgiesel, JHS, Matt Clark |
| 7 | true, false и: команды | Alessandro Mascolo |
| 8 | Арифметика Bash | Alessandro Mascolo, Ashkan, Gavyn, Jesse Chen, user1336087 |
| 9 | Ассоциативные массивы | Benjamin W., uhelp, UNagaswamy |
| 10 | Ввод переменных | uhelp |
| 11 | Внутренние переменные | Alexej Magura, Ashari, Ashkan, Benjamin W., codeforester, criw, Daniel Käfer, Dario, Dr Beco, fedorqui, Gavyn, Grisha Levit, Jahid, mattmc, Savan Morya, Stephane Chazelas, tripleee, uhelp, William Pursell, Wojciech Kazior |

| | | |
|---|---|---|
| 12 | Выбрать ключевое слово | UNagaswamy |
| 13 | Вывод цветного скрипта (кросс-платформенный) | charneykaye |
| 14 | Вырезать команду | Richard Hamilton |
| 15 | глобальные и локальные переменные | George Vasiliou, Ocab19 |
| 16 | Горячие клавиши | Daniel Käfer, JHS, Judd Rogers, m02ph3u5, Saqib Rokadia |
| 17 | Замена процесса | Benjamin W., cb0, Dario, Doctor J, fedorqui, Inian, Martin Lange, Мона_Сах |
| 18 | Замены истории Баша | Benjamin W., Bubblepop, Grexis, janos, jimsug, kalimba, Will Barnwell, zarak |
| 19 | Заявление о ситуации | P.P. |
| 20 | Здесь документы и здесь строки | Ajinkya, Benjamin W., Deepak K M, fedorqui, Iain, Jahid, janos, lanoxx, Stobor, uhelp |
| 21 | Избегание даты с помощью printf | Benjamin W., chepner, kojiro, RamenChef |
| 22 | Изменить оболочку | depperm, IamaTacos |
| 23 | Использование «ловушки» для реагирования на сигналы и системные события | Benjamin W., Carpetsmoker, dubek, jackhab, Jahid, jerblack, laconbass, Mike S, phs, Roman Piták, Sriharsha Kalluru, suvayu, TomOnTime, uhelp, Will, William Pursell |
| 24 | Использование кота | anishsane, Bubblepop, Christopher Bottoms, glenn jackman, intboolstring, Jahid, Matt Clark, Rafa Moyano, Root, Samik, Samuel, SLePort, tripleee, vielmetti, xhienne, |
| 25 | Использование сортировки | Flows, Mohima Chaudhuri |
| 26 | квотирование | Benjamin W., binki, Gilles, Jahid, Will |
| 27 | Когда использовать | Alexej Magura |

| | | |
|---|---|---|
| | eval | |
| 28 | Команда cut | Dario |
| 29 | Контрольные структуры | Dennis Williamson, DocSalvager, DonyorM, Edgar Rokyan, gzh, Jahid, janos, miken32, Samik, SLePort |
| 30 | Копирование (cp) | dingalapadum, Richard Hamilton |
| 31 | Ловушки | Cody, Scroff |
| 32 | Массивы | Alexej Magura, Arronical, Benjamin W., Bubblepop, chepner, Christopher Werby, codeforester, fedorqui, Grisha Levit, Jahid, janos, jerblack, John Kugelman, markjwill, Mateusz Piotrowski, NeilWang, ormaaj, RamenChef, Samik, Sk606, UNagaswamy, Will, |
| 33 | математический | deepmax, Pavel Kazhevets, Tim Rijavec |
| 34 | Навигация по каталогам | Christopher Bottoms, JepZ |
| 35 | найти | Batsu, Daniel Käfer, Echoes_86, fedorqui, GiannakopoulosJ, Inian, Jahid, John Bollinger, Laurel, leftaroundabout, Matt Clark , Michael Gorham, Mohima Chaudhuri, Peter, sjsam, tripleee, |
| 36 | Настройка PS1 | Blacksilver, Cows quack, Jahid, Jonny Henly, Josh de Kock, Kamal Soni, Misa Lazovic, tversteeg, Wenzhong |
| 37 | Обзорный | Benjamin W., chepner |
| 38 | Обработка приглашения системы | RamenChef, uhelp |
| 39 | отладка | Gavyn, Leo Ufimtsev, Sk606 |
| 40 | Параллельно | Jon |
| 41 | Передача файлов с помощью scp | Benjamin W., onur güngör, Pian0_M4n, Rafa Moyano, RamenChef, Reboot, Wojciech Kazior |
| 42 | Перенаправление | Alexej Magura, Antoine Bolvy, Archemar, Benjamin W., Brydon Gibson, chaos, David Grayson, Eric Renouf, fedorqui, Gavyn, George Vasiliou, hedgar2017, Jahid, Jon Ericson, Judd Rogers , lesmana, liborm, Matt Clark, miken32, Neui, Pooyan Khosravi , RamenChef, Root, Stephane Chazelas, Sven Schoenung, Sylvain Bugat, Warren Harper, William Pursell, Wolfgang, |

| | | |
|---|---|---|
| 43 | Последовательность выполнения файла | Riker |
| 44 | Программируемое завершение | Benjamin W., jandob |
| 45 | Пространство имен | meatspace, uhelp |
| 46 | Прочитать файл (поток данных, переменную) по очереди (и / или поле за полем)? | Jahid, vmaroli |
| 47 | Работа в определенное время | fifaltra, uhelp |
| 48 | Работа и процессы | Amir Rachum, Bubblepop, DonyorM, fifaltra, J F, Jouster500, Mike Metzger, Neui, Riccardo Petraglia, Root, Sameer Srivastava, Sk606, suvayu, u02sgb, WAF, WannaGetHigh, zarak |
| 49 | Разделение слов | Jahid, Jesse Chen, RamenChef, Skynet |
| 50 | Разделение файлов | Mohima Chaudhuri, Richard Hamilton |
| 51 | Расширение параметра Bash | Benjamin W., chepner, codeforester, fedorqui, George Vasiliou , Grexis, hedgar2017, J F, Jahid, Jesse Chen, Stephane Chazelas, Sylvain Bugat, uhelp, Will, WMios, ymbirtt |
| 52 | Расширение скобы | 4444, Benjamin W., Jamie Metzger, mnoronha, Peter, uhelp, zarak |
| 53 | Расшифровка URL | Crazy, l0_ol |
| 54 | Сеть с Bash | dhimanta |
| 55 | Скрипт shebang | DocSalvager, Sylvain Bugat, uhelp |
| 56 | Скрипты CGI | suleiman |
| 57 | Скрипты с параметрами | Jahid, James Taylor, Kelum Senanayake, Matt Clark, RamenChef |
| 58 | Совпадение шаблонов и регулярные выражения | Benjamin W., chepner, Chris Rasys, fedorqui, Grisha Levit, Jahid, nautical, RamenChef, suvayu |

| | | |
|---|---|---|
| 59 | Создание каталогов | Brendan Kelly |
| 60 | со-процессы | Dunatotatos |
| 61 | Список файлов | Benjamin W., cswl, depperm, glenn jackman, Holt Johnson, Iain, intboolstring, J F, Jonny Henly, Markus V., Misa Lazovic, mpromonet, Neui, Osaka, Richard Hamilton, RJHunter, Samik, Sylvain Bugat, teksisto |
| 62 | Тип корпуса | Jeffrey Lin, liborm, William Pursell |
| 63 | Трассирование | Chandrahas Aroori |
| 64 | Трубопроводы | Ashkan, Jahid, liborm, lynxlynxlynx |
| 65 | Управление заданиями | Samik |
| 66 | Управление переменной среды PATH | Jahid, kojiro, RamenChef |
| 67 | Условные выражения | BurnsBA, Gilles, hedgar2017, Jahid, Jonny Henly, mnoronha, RamenChef, |
| 68 | Утилита сна | Ranjit Mane |
| 69 | функции | BrunoLM, Christopher Bottoms, dimo414, divyum, edi9999, Jahid, janos, Matt Clark, Michael Le Barbier Grünewald, Neui, Reboot, Samik, Sergey, Sylvain Bugat, tripleee, |
| 70 | Цепь команд и операций | jordi, Mateusz Piotrowski, uhelp |
| 71 | Шаблоны проектирования | mattmc |