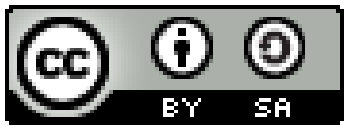
Командный интерпретатор **bash**



Предназначение

Командные интерпретаторы — обязательный компонент операционной системы.
Основные задачи:

- выполнение команд, полученных от пользователя на терминале;
- выполнение программ (скриптов), написанных на языке конкретного командного интерпретатора;
- выполнение задач из программ на компилируемых языках, когда требуется нечто большее, чем выполнение просто одной другой программы.

Здесь будет рассматриваться **bash** — наследник командного интерпретатора **sh**. Существуют также csh, tcsh, ksh, zsh ...

Комментарии и специальный комментарий

Для начала однострочного комментария используется специальный символ **#** . Пример:

```
# This is a comment
ls -l # это комментарий после команды
# а в следующей строке нет комментария
a="Primus␣#1"
```

Если комментарий в первой строке файла начинается с **#!**, то это — указание интерпретатора для последующей программы.

```
#!/bin/bash
# здесь программа на bash
files=‘ls -a ~/mydir/*.txt‘
```

```
#!/usr/bin/perl -w
# здесь программа на языке perl
```

Простая команда

Основа построения программ (и сложных команд) на bash — **простая команда**.
Синтаксис:

[name=value...] [command [arg ...]] [redirections]

[] — необязательная часть;
name=value — работа с переменными;
command — имя команды;
arg — аргументы;
redirections — перенаправления ввода-вывода.

Простая команда

Только команда с аргументами

Если дана только команда с необязательными аргументами — эта команда выполняется.
Если аргументы (позиционные и ключевые) заданы — они передаются программе. Имя программы передаётся как нулевой аргумент. Примеры

```
echo aaa          bbb     ccc
#> aaa bb ccc

ls -l / -d
#> dr-xr-xr-x. 23 root root 4096 Oct  9 19:44 /

./my_prog -x data.txt
#> Segmentation fault

/sbin/ifconfig -a
```

Код возврата = код завершения команды (0=Good), или $128 + n$ — если команда убита сигналом n.

Командой может быть:

- внутренней командой bash, такой как **cd**, **echo**;
- алиасом (синонимом), например **ll**;
- функцией, написанной на языке bash;
- внешней программой (бинарной или скриптовой).

Если в имени команды есть символ “/”, то это — только внешняя команда, причём путь к ней указан, и поиск производится только в указанном месте.
Если путь не указан — поиск производится только в каталогах, указанных в переменной окружения **PATH**.
Команда и аргументы разделяются пробельными символами — любым количеством.

Простая команда
Присваивание переменным

Если в простой команде задана только первая часть — это присваивание переменной bash.

```
a=abcde
b=123
c='String with special chars:  $ < > { } ( ) $'

echo $a $b "$c" $d
# abcde 123 String with special chars:  $ < > { } ( ) $
```

Команда **echo** — выводит на стандартный вывод свои аргументы
В bash все переменные по умолчанию содержат строки.
Если переменная не существовала — будет создана.
Пробелы вокруг “=” недопустимы.

```
ls=5 ; echo $ls
# 5
ls =7 ; echo $ls
# ls: cannot access =5: No such file or directory
# 5
```

Простая команда

Установка переменных окружения для команды

Если в команде присутствует и первая (присваивания), и вторая (команда) части, то это — запуск команды с установкой для неё переменной окружения. Примеры:

```
LC_TIME=uk_UA.UTF-8 cal -m -3

TERM=dumb ls

LANG=C man open

PATH=/usr/bin:/usr/sbin:~/bin bash
```

Если надо установить переменную окружения для всех последующих команд, следует использовать команду **export**:

```
export LC_NUMERIC=C
```

Перенаправления ввода-вывода

Обычно, у программы изначально открыто 3 файла. Им назначены специальные файловые дескрипторы (по умолчанию — текущий терминал).

Специальные дескрипторы файлов

0 — стандартный ввод (stdin, cin, getchar, scanf, read, . . .)

1 — стандартный вывод (stdout, cout, putchar, printf, echo)

2 — стандартный вывод ошибок (stderr, cerr)

Перенаправление вывода

> file — stdout -> в файл (создать при необходимости, старое содержимое — выбросить).

N> file — дескриптор номер N -> в файл.

>> file — stdout -> дозапись в файл.

N>> file — дескриптор номер N -> дозапись в файл.

```
ls > a.txt

date >> a.txt

mkdir /yyy/xxx 2> err.txt

mkdir /yyy/xxx 2> /dev/null
# перенаправление в спец. файл
```

Перенаправления ввода-вывода

Перенаправления ввода, объединение, . . .

Перенаправление ввода

< file — stdin <- из файла

N< file — дескриптор N <- из файла

```
sort < a.txt
```

Объединение дескрипторов

M>&N

M<&N

```
cat /etc/shadow > x.txt 2>&1
cat /etc/shadow &> x.txt
```

Перенаправления ввода-вывода
Встроенные документы

Часто надо переназначить ввод, но использовать для этого отдельный файл неудобно. Тогда можно использовать встроенный документ:

<<WORD
data1
data2
data3
....
WORD

Пример:

```
cat > out.txt <<EOF
aaaaaaaaaa
bbbbbbbbbb
cccccccc
EOF
```

Трубопроводы (pipeline)

Предназначены для того, чтобы одна команда обеспечивала данными другую, причём без участия файловой системы.

Синтаксис:
[time] [!] cmd1 [| cmd2 . . .]

Стандартный вывод команды cmd1 перенаправляется на стандартный ввод cmd2 и т.д.

Примеры:

```
head -5 /etc/passwd | cut -f6 -d:

cut -f1 -d: /etc/group | sort | head -4
```

Статус завершения трубопровода определяется последней командой. При необходимости инвертируется символом ”!”
time — вывод затраченного времени.
|& — трубопровод с объединением.

Подстановки (expansions)

Перед тем, как решить, что из себя представляет входная строка, и что с ней делать, bash выполняет

подстановки (expansions, замены, расширения) —

- убирает сам текст, который привёл к срабатыванию подстановок;
- и чем-то заменяет (по определённым правилам).

Подстановки выполняются в определённом порядке.

Подстановка фигурных скобок { }

Если во входном потоке встречается конструкция вида
prefix{word1,word2,...}suffix
то она заменяется на
prefixword1suffix prefixword2suffix ...
При этом prefix и суффикс — необязательны.
Исключение: **${**
Примеры:

```
echo it{s14,s15,p14,p15}_home
#> its14_home its15_home itp14_home itp15_home
```

Можно использовать диапазоны

```
echo x{a..h}
#> xa xb xc xd xe xf xg xh

echo a{0..1}{t..y}
#> a0t a0u a0v a0w a0x a0y a1t a1u a1v a1w a1x a1y

echo {003..12..2}
#> 003 005 007 009 011
```

Подстановка тильды ~

Если слово начинается с ~, а потом — имя пользователя, то делается попытка подставить имя домашнего каталога этого пользователя (если не указано — текущего).
Примеры:

```
echo ~ ~/ ~jj ~root ~xxx x~atu
#> /home/atu /home/atu/ /home/jj /root ~xxx x~atu
```

~+ заменяется на имя текущего каталога ($PWD), а **~-** на имя предыдущего ($OLDPWD)

```
cd /usr/bin ; cd / ; echo ~+ ~-
#> / /usr/bin
```

Подстановка параметров (переменных)

Конструкция вида
$name или **${name}** — подставляет значение переменной или параметра.

```
f=AAAA ; g=BBBB ; ma[4]='Fifth elem' ; ma[7]='Apache'
ft='CCCC    DD'
ia=(aa bb cc dd ee)
echo $f $g $ft ${f}t ${ma[4]} ${ma[*]} ${#ma[*]}
echo ${ia[0]} ${ia[4]} ${#ia[@]}
#> AAAA BBBB CCCC DD AAAAt Fifth elem Fifth elem Apache 2
#> aa ee 5
```

Специальные параметры:
$0 — имя программы; $1 — первый аргумент $2 — второй.
$* или $@ — все параметры, $# — их число;
$$ — PID(bash); $RANDOM — случайное число ...

Подстановка параметров (переменных)

Модификация подставляемого значения

${name:-word} ${name-word} — значение по умолчанию

```
f='AAAA' ; unset a
echo ${f:-xxx} ${a:-zzz}
echo $a
#> AAAA zzz
#>
```

${name:=word} ${name=word} — значение по умолчанию с присваиванием

```
f='AAAA' ; unset a ; echo ${f:=xxx} ${a:=zzz}
echo $a
#> AAAA zzz
#> zzz
```

${#name} — длина строки или размер массива

```
f='AAAA'
echo $f ${#f}
#> AAAA 4
```

Подстановка параметров (переменных)

Модификация подставляемого значения (2)

${name#prefix} — убрать префикс (если есть)

```
x='file.txt' ; y='data.tar.gz'
echo ${x#file} ${x#word} "${y##*.}" "${y#*.}"
#> .txt file.txt gz tar.gz
```

${name%suffix} — убрать суффикс

```
x='data.file.txt'
echo ${x%.txt}.html ${x%.*} ${x%%.*}
#> data.file.html data.file data
echo ${x%.png}.html
#> data.file.txt.html
```

${name/pattern/string} — замена первого совпадения
${name//pattern/string} — замена всех совпадений

```
a='aaaxcombbbcxcomcccxcomddd'
echo ${a/xcom/ZZZZ}
echo ${a//xcom/ZZZZ} ${a//[d-z]/T}
#> aaaZZZZbbbcxcomcccxcomddd
#> aaaZZZZbbbcZZZZcccZZZZddd  aaaTcTTbbbcTcTTcccTcTTTTT
```

Подстановка параметров (переменных)
Модификация подставляемого значения (3)

${name:offset}
— подстрока от заданного смещения offset
${name:offset:length}
— подстрока от заданного смещения offset длины length

```
a='abcdefghijklmn' ; n=4
echo ${a:2}  ${a:2:5} ${a:1:n}
echo ${a:(-7):5} ${a:(-7):(-4)}
#> cdefghijklmn cdefg bcde
#> hijkl hij
```

${!prefix*}
– подставляются **имена** переменных, которые начинаются с prefix.

```
username='its1205' ; usergroup='stud' ; usedir='unknown'
echo ${!user*}
#> usergroup username
```

More: ^ ^^ , ,, :? :+

Подстановка вывода команды

Конструкции вида

```
`cmd`  $(cmd)
```

выполняют заданную команду, и подставляют во входной поток то, что команда вывела на стандартный вывод

Пример:

```
sz=$( stat -c '%s' /etc/fstab )
echo $sz
#> 1033

xhome=$( sort /etc/passwd | tail -1 | cut -f1 -d: )
echo "$xhome"
#> zabbixsrv

w=`wc -l /etc/group`
echo $w
#> 201 /etc/group
```

Подстановка арифметических выражений

Конструкции вида

```
$(( expr ))
```

вычисляет заданное выражение, и подставляют во входной поток текстовое представление результата. Пример:

```
a=17
b=4
echo $((2*4+a)) $((1<<10)) $(( 0xFFFF & (1<<b) ))
#> 25 1024 16
```

Внутри двойных скобок подстановка значений переменных производится автоматически, и приводится к целому числу.

Подстановка процессов

```
<(cmd)

>(cmd)
```

Выполняется команда, стандартный вывод/ввод перенаправляются в файл специального вида (/dev/fd/...), и подставляется имя данного файла.

Разбиение на слова

Следующий этап — разбиение входной строки (потока) на слова. Символы-разделители — в переменной IFS.

```
echo          aaa         bbb                  c
#> aaa bbb c
echo   "␣␣␣␣␣␣aaa␣␣␣␣␣␣␣bbb␣␣␣␣␣␣␣␣␣␣␣␣␣␣c"
#>       aaa          bbb                  c
```

Если изменяете переменную IFS (например, для команды read) – не забывайте восстанавливать.

Подстановка имён файлов

Если в слове есть символы:
* — любое кол-во любых символов
? — 1 любой символ
[] — диапазон
то делается попытка подставить подходящие имена файлов

```
echo *.sh
#> bash_subst.sh bash1k.sh bash2.sh bash2k.sh

echo *.c
#> *.c

echo /usr/include/stdi*.h
#> /usr/include/stdint.h /usr/include/stdio_ext.h
#>> usr/include/stdio.h
```

Экранирование (закавычивание)

\char — отмена специального значения 1 символа

```
mkdir my\ directory ; ls -ld m* ; rmdir my\ directory
#> drwxr-xr-x 2 atu atu 4096 Oct 26 23:15 my directory
```

'string ' — отмена специального значения всех символов

```
a='xxx'
echo 'abc $a *.sh  /* comment */ $( ls ) \\ '
#> abc $a *.sh  /* comment */ $( ls ) \\
```

"string␣with␣\$var␣\$(cmd)␣" – сохраняется специальное значение символов $, ‘, \, !

```
echo "abc␣$a␣*.sh␣␣/*␣comment␣*/␣$(␣ls␣)␣\\␣"
#> abc xxx *.sh  /* comment */ 1
#> bash_subst.sh
#>  bash1k.sh .......
```

Существует еще $'string\nwith\tcontorl chars' и $"Translate␣me!" .

Логические выражения

Для проверки условий можно использовать любую команду, но для проверки наиболее часто встречающихся условий есть команды **test** , **[** и специальная конструкция bash **[[ ]]** .

```
test -e /etc/passwd -a -f /etc/shadow ; echo $?
[ -e /etc/passwd -a -f /etc/nofile ] ; echo $?
[[ -e /etc/passwd && -d /run ]] ; echo $?
# > 0
# > 1
# > 0
```

Результат проверки — возвращаемое значение команды.
0 — true, $\neq 0$ — false **!!!**

Логические выражения

Проверки, связанные с файлами (1)

–e file — файл существует; (–a in [[ ]])
–f file — файл существует и является обычным файлом;
–d file — файл существует и является каталогом;
–b file — файл существует и является блочным устройством;
–c file — файл существует и является символьным устройством;
–S file — файл существует и является сокетом;
–p file — файл существует и является именованным каналом;
–L file и **-h file** — файл существует и является символической ссылкой;

–r file — файл существует и доступен для чтения;
–w file — файл существует и доступен для записи;
–x file — файл существует и доступен для выполнения;
–s file — файл существует и размер > 0;

Логические выражения

Проверки, связанные с файлами (2)

–g file — файл существует и установлен SGID бит;
–u file — файл существует и установлен SUID бит;
–k file — файл существует и установлен sticky бит;
–O file — файл существует и эффективный пользователь — его владелец;
–G file — файл существует и эффективный пользователь входит в группу файла;

file1 –ef file2 — файлы file1 и file2 — один и тот же файл;
file1 –nt file2 — файл file1 новее, чем file2;
file1 –ot file2 — файл file1 старее, чем file2
. . . . . .

Логические выражения

Проверки, связанные со строками (1)

string — строка не пустая ;
–z string — строка пустая ("");

string1 == string2 — строки равны;
string1 != string2 — строки не равны;
string1 < string2 ,
string1 <= string2 ,
string1 > string2 ,
string1 >= string2 — сравнение строк.

Для того, что бы сравнивать строки как числа, надо использовать операторы сравнения **–eq, –ne, –lt, -le, –gt, –ge** .

Списки команд (list)

Трубопроводы можно объединять в списки, разделяя их символами: **; & && ||**

; — просто разделитель команд в строке. Выполняются последовательно и независимо.

```
a=12; cd / ; ls -l -a | sort -u | head -4
```

Если трубопроводы разделены &&, то второй выполняется только тогда, если первый завершился успешно (rc=0)

```
[[ -f new_mail.txt ]] && \
 mail -s Report webmaster@my.host < new_mail.txt
```

Если трубопроводы разделены ||, то второй выполняется только тогда, если первый завершился с ошибкой (rc$\neq$ 0)

```
[[ "$USER" == 'xen' || "$USER" == 'nobody' ]] || exit 1
```

Объединение команд в одну

Несколько команд можно объединить в одну:
(list) — список выполняется в отдельном интерпретаторе. Изменения переменных, текущего каталога и т.д. не влияют на исходный интерпретатор.
{ list ; } — выполняется в том же интерпрететоре. Изменения переменных, текущего каталога и т.д. сохраняются.

```
cd / ; a='AAA'
( a='XXX' ; echo "$a" ; cd /home )
echo "$a" ; pwd
# AAA
# /

{ a='YYY' ; echo "$a" ; cd /home }
echo "$a" ; pwd
# YYY
# /home
```

Цикл for

```
for var in wordlist ; do list ; done
```

var — переменная цикла, list — тело цикла, wordlist — список слов. Указанная переменная последовательно перебирает значения всех слов из списка.

```
li='file1 file2 file3'
for x in $li ; do
  touch "$x"
  echo "file␣$x"
  stat "$x"
done

for z in /usr/lib/?x* ; do
  ls -ld "$z"
done

ar=(aaa bbb ccc)
for a in "${ar[*]}" ; do echo "$a" ; done
for a in "${ar[@]}" ; do echo "$a" ; done
```

Цикл for арифметический

```
for(( expr1; expr2; expr3 )) do list ; done
```

expr1, expr2, expr3 — арифметические выражения, аналогичные таковым в языке C.

```
for((a=1;a<100000; a<<=1)) ; do
  echo $a
  printf '%4X %6o %6d \n' "$a" "$a" "$a"
done

for((a=0;a<10; a++)) ; do
  echo -n " $a "
done
```

вычисление арифметического выражения

```
(( expr ))
```

Вычисляет значения арифметического выражения, но в отличие от арифметической подстановки не подставляет результат, а просто вычисляет.

```
a=12
b=4
(( b=a*7+b/2 ))
(( a++ ))
echo "$a␣$b"
```

Вычисление логического выражения [[ ]]

```
[[ logical_expression ]]
```

Используется для проверки логических выражений. Часто удобнее, чем test или “[” при вычислении сложных выражений.

```
[[ -r /etc/shadow || -r /etc/gshadow ]] && echo 'Bad!!!'
```

Выбор из меню (select)

```
select var in wordlist ; do list ; done
```

var — переменная меню
list — тело меню
wordlist — список слов.

```
select z in /usr/lib/?x* ; do
  if [[ -f "$z" ]] ; then
    ls -ld "$z"
  fi
done
```

Ctrl-D — выход

цикл while (с предусловием)

```
while list1 ; do list2 ; done
```

list1 — условие продолжения цикла
list2 — тело цикла.

```
head -5 /etc/passwd | {
while read a; do
  b=$(echo $a | cut -d: -f1 )
  c=$(echo $a | cut -d: -f7 )
  echo "$b␣$c"
done
}
```

if — условие

```
if list1 ; then list2 ; elif list3 ; then list4
 ... else listn ;
fi
if list1 ; then list2 ;  else listn ; fi
if list1 ; then list2 ; fi
```

Условием служат списки команд list1, list3 . . . . При выполнении условия — выполняются соответствующие списки команд list2 . . .

```
if [ -f /etc/passwd ] ; then
  echo "File!"
else
  echo "Other"
fi

if grep -v 'some string' *.c ; then
  echo 'Found'
fi
```

Выбор из набора вариантов

```
case word in pattern) list ;; ... esac
```

```
read a
case "$a" in
  *.gif) echo 'GIF image' "$a" ;;
  *.png) echo 'PNG image' ;;
  *) echo 'Other';
esac
```

Функции

```
funcname() { list }
```

funcname — имя функции. Тело функции — list. Аргументы передаются как $1, $2, ... После определения можно вызывать как команду.

```
myfun()
{
  local f="$1"
  echo "arg1: $f"
  if [[ -z "$f" ]] ; then
    return 5
  fi
  return 0
}

echo 'myfun asdf zzz'
myfun asdf zzz ; echo $?
echo 'myfun'
myfun ; echo $?
```

Синонимы (алиасы)

Для часто используемых команд (можно с аргументами) можно вводить синонимы. После определения можно вызывать как команду.

```
alias lsld='ls -l -d'
lsld
# drwxr-xr-x. 5 atu atu 4096 Oct  6 19:39 .
lsld /
# drwxr-xr-x. 23 root root 4096 Oct  6 16:29 /
```

Параметры синонимам не передаются. Если надо — используйте функции. Убрать — unalias. Команда *alias* без параметров выводит список существующих.