BASH конспект

Дата создания: 16.12.97
Источники:

1. bash man page
2. /gnu/docs/bash1147/document/featu_*.htm (bash features overview)
3. /gnudocs/sh-util1.12/doc/sh_ut?.htm (sh-utils documentation)
4. news://comp.unix.shell/faq.shell? (shells FAQ)
5. news://comp.unix.shell/bash.faq (bash FAQ)
6. csh.whynot (Tom Christiansen . Cch Programming Considered Harmful)

ЗАМЕЧАНИЯ ОБ ЭТОМ ДОКУМЕНТЕ

В помощь читающим цветную версию

(в данный момент форматирование документа не завершено; скоро все будет выглядеть чуть более прилично; :))
Примеры, в которых метаобозначения типа [[команда 1]] или [[аргументы]] нужно заменить конкретными названиями, набраны

темно-синим цветом.

Примеры отдельных команд или скриптов, готовых к выполнению as is, набраны

коричневым цветом.

Если при выполнении скриптов на экран выводятся какие-то результаты, то эти результаты набраны

бирюзовым цветом.

другие особенности

Мне не удалось подобрать адекватных терминов (однословных) для перевода слов [[expansion]] и [[shell]], и далее по тексту для избежания [[несклоняемости]] вместо английских слов используются слова [[экспансия]] и [[шелл]] (то же самое можно сказать о [[баше]]). Если вы борец за чистоту русского языка, можете смотреть на фразы типа [[система сообщает шеллу]] как на [[система сообщает shell'у]], но лично мне использование русских падежных окончаний после слов, записанных латинскими буквами, нравится еще меньше, чем введение варварских неологизмов.
Если у вас есть какие-то идеи, напишите, пожалуйста, по адресу grg@philol.msu.ru

ЕЩЕ НЕ ОПИСАНЫ:

filename generation when using output redirection (command >a*)
test -o optname/s1 == s2/s1 < s2/s1 > s2/-nt/-ot/-ef/-O/-G/-S
История команд и команды истории (раздел начат);

READLINE library functions

BASH И ЕГО МЕСТО СРЕДИ ДРУГИХ SHELLS

BOURNE_SHELL=sh, {ba,k,z}sh
C_SHELL={c,tc}sh
OTHER_SHELL=rc ssh
Различия между bash и sh (а также особенности новейшей версии bash) описаны в конце документа.

ЗАПУСК BASH И ВЫХОД ИЗ НЕГО. КОНФИГУРАЦИОННЫЕ ФАЙЛЫ

Чтобы запустить какой-то скрипт на исполнение, пишут

```
bash filename
```

Если нужно выполнить встроенную команду, надо писать

```
bash -c 'command'
```

При запуске login-shell

Для всех пользователей: если есть файл /etc/profile , выполнить его.
Для данного пользователя: выполнить первый из существующих файлов: ~/.bash_profile, ~/.bash_login или ~/.profile (~ -- директория пользователя)

При выходе из login-shell

Выполняется ~/.bash_logout , если есть.

При запуске non-login shell:

Интерактивный -- выполнить ~/.bashrc.
Неинтерактивный -- исполнить файлы с именами $BASH_ENV и $ENV

УСТРОЙСТВО КОМАНДНОЙ СТРОКИ

Под "командой" далее будет пониматься или отдельная команда, например:

```
ls
```

-- или так называемый конвейер (pipeline) -- последовательность двух или более команд, в которой станартный вывод предшествующей команды передаются на стандартный ввод последующей:

```
ls | more
```

В bash синтаксис pipeline несколько расширен по сравнению с sh, и имеет следующий вид:

```
[time [-p] ] [ ! ] command [ | command2 ... ]
```

-- где в факультативную часть входит "time" -- зарезервированное слово, используемое для измерения времени, затраченного на выполнение команды, а также "!" -- знак отрицания, меняющий код завершения команды на противоположный.
Обычно pipeline находится на одной строке. Если требуется разместить ее на нескольких строках, несколько строк можно объединять в одну логическую строку, используя знак переноса "\". На одной строке можно записывать несколько команд, разделяя их специальными знаками (см. ниже).
Командная строка может состоять из одной или нескольких команд (точнее, pipelines). Для группирования команд применяются следующие символы:

символы ";" и "&" -- выполнение нескольких команд

Выполнить обе команды:

```
команда1; команда2  # последовательное выполнение
```

```
команда1& команда2  # параллельное выполнение
```
Символ ";" используется для последовательного выполнения нескольких команд: после завершения одной команды начинается выполение другой. Если вместо ";" используется "&", то команда, сопровождаемая этим знаком, выполняется в фоновом режиме, а выполнение следующей команды начинается немедленно, и, таким образом, две или более команд выполняются параллельно.
Note that wherever you see a `;' in the description of a command's syntax, it may be replaced indiscriminately with one or more newlines.

Символы "&&" и "||" -- условное выполнение команд

Выполнить команду1, и если она выполнилась удачно, выполнить команду2:

```
команда1 && команда2
```
Выполнить команду1, и если она выполнилась неудачно, выполнить команду2:

```
команда1 || команда2
```
Как известно, каждая команда возвращает некоторый код завершения, по которому можно судить об "успешности" выполнения. Если на командную строку смотреть как на логическое выражение, значение которого требуется вычислить, где отдельные команды -- переменные, принимающие значение "истина" или "ложь", то знаки "&&" и "||" можно рассматривать как операции логического И и ИЛИ. Отсюда понятна и логика действия этих связок в командой строке: если первая из двух команд, соединенных связкой И, возвращает "ложь", то результат всего выражения -- заведомо "ложь", поэтому вторая команда не выполняется; аналогично, если первая из двух команд, соединенных связкой ИЛИ, первая выполнилась успешно, то уже известно, что результат всего выражения -- "истина", и вторая команда не выполняется.
В отличие от подобных операций в Си, связки "||" и "&&" в bash имеют **одинаковый приоритет**, то есть выражение "A || B && C" аналогично "(A || B) && C", но не "A || (B && C)". Для изменения приоритета (порядка выполнения) можно использовать круглые или фигурные скобки (см. ниже)

символы {} -- группирование команд в один pipeline

Чтобы сделать перенаправление ввода-вывода для нескольких команд, вместо
```
"cmd1 >a; cmd2 >>a"
```
-- можно писать
```
"{ cmd1 ; cmd2; } >a"
```

символы ( )

( ) -- команды, после выполнения которых сделанные ими изменения environment vars восстанавливаются
Напр. строка:

```
a="global"; (a="local"; echo now a is $a); echo and now $a
```
выведет:

```
now a is local
and now global
```
И фигурные, и круглые скобки можно использовать для группирования команд с целью изменения порядка выполнения команд со связками "||" и "&&".

Проверка значения арифметического выражения с помощью (( ))

Выражение, заключенное в двойные круглые скобки, будет вычислено по правилам вычисления арифметических выражений, действующих, кроме данного случая, еще и для конструкции $(()) и команды let. После вычисления выражения команда считается выполненной успешно (и возвращается код 0) в том случае, если результат отличен от нуля, и неуспешно в случае равенства результата нулю.

символ !

```
! команда
```
-- отрицание: ставится перед командой для отрицания кода ее выхода

ЭКСПАНСИЯ

Экспансия -- процесс анализа командной строки с целью нахождения в ней специальных обозначений (выражений) и подстановки на их место соответствующих значений.
Если надо посмотреть результат экспансии набранной в данный момент строки, можно нажать M-C-e (Esc, Ctrl-e) , а затем можно вернуть на место прежнюю строку через undo (Ctrl-"-").
Порядок экспансии таков:

1. {}-экспансия (bash specific)

Создает преамбулы и постамбулы :
{prefix1,prefix2,...}radix{suffix1,suffix2,...}
Напр. создать структуру директорий:

```
mkdir /usr/{bin,lib,libexec,local,share{,/man,/doc,/misc,/locale},X11R6{,/bin,/lib,/man,/share}}
```

создаст:

```
/usr/lib
/usr/libexec
/usr/local
/usr/share
/usr/share
/usr/share/man
/usr/share/doc
/usr/share/misc
/usr/share/locale
/usr/X11R6
/usr/X11R6/bin
/usr/X11R6/lib
/usr/X11R6/man
/usr/X11R6/share
```

2. ~-экспансия

~/ -- директория текущего пользователя
~user1/ -- директория пользователя user1
~+ -- $PWD
~- -- $OLDPWD

3. $-экспансия

${n} -- Если цифр больше одной, номер параметра надо брать в {}
${n:-default} -- если параметр или переменная пусты, подставить default
${var:=default} -- только для переменных: если переменная пуста, присвоить ей default и подставить это значение
${n:?message} -- если пусто, выдать сообщение message и прекратить работу
${n:+value} -- если пусто, оставить пустым, а иначе подставить value; аналог в Си: 'flag ? string : """
$# -- количество аргументов, в отличие от Си $0 не считается
${#n} -- длина параметра или переменной
${n#pattern} отрезать начало, совпадающее с паттерн (shortest matching)
${n##pattern} отрезать начало, совпадающее с паттерн (longest matching)
${n%pattern} отрезать конец, совпадающий с паттерн (shortest matching)
${n%%pattern} отрезать конец, совпадающий с паттерн (longest matching)
Пример:

```
set $1="--file=a.txt"
echo ${1#--file=}
```

выведет

```
a.txt
```

$*, $@

Если надо при передаче всех аргументов сохранить исходные кавычки,

используется "$@" (именно в двойных кавычках!).

А "$*" передают все аргументы как один, тоже в кавычках.

Без кавычек $* $@ эквивалентны и не годятся для передачи

аргументов в кавычках.

$? -- код выхода последней команды, в т. ч. встроенной (ср. DOS errorlevel)
$$ -- PID, очень полезно для создания временных файлов с уникальными именами:

```
tmpfile1=tmp1$$
tmpfile2=tmp2$$
```

$! ???
Expands to the process ID of the most recently executed background (asynchronous) command.
$_ Последний аргумент предыдущей команды.
$- Флаги для команды set.

4. командная подстановка

$( ) и ` ` (обратные кавычки). Заменяются на вывод команды без символов новой строки
`команда`
-- старый стиль, обрабатываются внутренние backslashes . чтобы сделать вложение, внутри надо писать:
`команда1 \`команда2\``
$(команда)
-- новый стиль (bash specific) , все , что внутри , без обработки рассматривается как команда. Чтобы сделать вложение, надо писать:

```
$(command1 $(command2))
```

5. арифметическая подстановка

$(( )) или $[ ]
Есть, среди прочих, операции << и >>
Переменные можно использовать как с " $" , так и без.
Есть все виды присваивания:

```
= *= /= %= += -= <<= >>= &= ^= |=
```

Понимает префиксы 0 и 0x.
n # -- префикс, указывающий любую [позиционную] систему счисления

6. подстановка процесса (bash specific)

Возможна на системах, поддерживающих named pipelines(FIFOs)

```
>(команды)
```

-- отправить вывод команды в некоторый файл и передать имя этого файла как аргумент другой команде.

```
<(команды)
```

-- отправить вывод команд в файл и подставить имя этого файла
Пример. Сравнить выводы программ newprogram и oldprogram:

```
diff <(newprogram) <(oldprogram)
```

Еще пример. Загрузить вывод команды ls в редактор vi:

```
vi <(ls)
```

Следует заметить, что выполнение именно этой команды соответствует ожиданиям, тогда как команда:

```
ls | vi
```

-- приведет или к сообщению об ошибке, или к другим неожиданным последствиям.

7. разбиение на слова

8. path-экспансия

Обрабатывает ?, *, [list] и [^ list] в именах файлов
* -- все файлы, кроме начинающихся на "."
.* -- файлы, начинающиеся на "."
Чтобы сделать шаблон, которому бы соответствовали все файлы, в том числе начинающиеся на ".", но исключая файлы "." и ".." , надо использовать сразу три шаблона:

```
.[^.]*  .??*  *
```

-- потому что:

- ".[^.]*" включает все файлы с одной точкой в начале (но не с двумя), в именах которых на втором месте не стоит "."
- ".??" включает все файлы, состоящие из точки и еще двух знаков, что позволит исключить файл ".."
- "*" включает все файлы, не начинающиеся с "."

9. удаление кавычек

В конце экспансии все кавычки, кроме экранированных (\") удаляются

Кавычки и экранирование специальных символов

Метасимволы и их экранирование

Метасимволы -- это специальные символы, выполняющие роль разделителей слов (таких, как аргументы командной строки и названия команд). Некоторые метасимволы играют также роль знаков препинания. К метасимволам относятся знаки:

```
| & ; ( ) < > пробел  табулятор
```

Иногда требуется включить метасимвол в состав слова, избежав его использования в качестве разделителя слов. Для этого перед метасимволом ставится обратная косая черта, которая и означает, что непосредственно следующий за ней специальный символ должен быть лишен своего специального значения и воспринят "буквально". Операция постановки косой черты назвается экранированием. Примеры будут приведены далее, после рассмотрения еще двух способов включения специальных символов в состав слов.

Двойные и одинарные кавычки.

Другие виды кавычек

Для записи управляющих символов с помощью эскейп-последовательностей, используется следующая конструкция:

```
$'escape-sequence'
```

Например:

```
echo Backspaces inserted in this string will remove a whole$'\b\b\b\b\b'
word
```

ПЕРЕНАПРАВЛЕНИЕ

Самые интересные сведения о перенаправлении, изложенные ниже, взяты из [[csh.whynot]]. Этот документ слишком интересен и информативен для того, чтобы полностью пересказывать его здесь, поэтому ниже приводятся лишь некоторые сведения.
Можно запретить перенаправление вывода ">" в уже существующий файл(см. про переменную noglobber), но ">|" работает всегда
При нескольких перенаправлениях существенен порядок:

```
command 2>&1 1>a.txt
```

перенаправит sdderr на stdout , а stdout на a.txt, а вот

```
command 1>a.txt 2>&1
```

-- перенаправит и то и другое в a.txt, так как во время обработки 2>&1 / dev /fd1 будет уже перенаправлен.
Мощные возможности для [[динамического]] перенаправления внутри скрипта дает команда exec (см.). Собственно запуск процесса exec при данном применении не осуществляется, поэтому опишем это применение здесь.

перенаправление "здесь документы"

Пример из news://comp.unix.shell:

```
#!/bin/bash
ftp -n << EOF

open somehost.somedomain
user Anonymous aaa@bbb.com

cd /pub/upload

put /home/me/myfile.txt

quit

EOF
```

Опция - n В этом скрипте заставляет ftp не спрашивать имя пользователя и пароль, предоставляя возможность воспользоваться командой user в следующей форме:

```
user USER_NAME PASSWORD
```

динамическое перенаправление

Если аргумент для команды exec не указан, но указаны символы перенаправления, exec устанавливает эти перенаправления, и выполнение текущего скрипта продолжается. Таким образом достигается [[динамическое]] перенаправление:

```
#!/bin/bash
echo
Эта строка выводится на экран
exec >file.txt
echo
А эта строка в файл
echo И эта в файл
```

-- выведет на экран только одну строку, а две других запишет в file.txt

дескрипторы

открытие новых дескрипторов

Кроме возможности дублировать существующие дескрипторы (0, 1, 2) другими, также уже существующими, (напр. 2 >&1) или перенаправлять существующие дескрипторы в файл (напр. 1>log.txt), имеется возможность открывать новые, прежде закрытые дескрипторы с номерами 3 и более.
В следующем примере сначала создается (с применением перенаправления here documents]]) файл os.txt, затем с этим файлом связывается дескриптор 4, из файла читается одна строка и записывается в файл, связанный с дескриптором 3 (docname.txt):

```
#!/bin/bash
#создать файл-пример для последующего чтения
cat << EOF >os.txt
ОПЕРАЦИОННЫЕ СИСТЕМЫ
UNIX
DOS
OS/2
Windows NT
EOF
echo Открываем дескрипторы 3 и 4
exec 3>docname.txt
exec 4<os.txt
echo Читаем одну строку через дескриптор 4
read d <&4
echo Записываем строку через дескриптор 3
echo $d >&3
```

-- запишет в файл docname.txt строку [[ОПЕРАЦИОННЫЕ СИСТЕМЫ]].

восстановление перенаправленных дескрипторов

Если требуется перенаправить ввод или вывод временно, а затем отменить это перенаправление, информацию, ассоциируемую перераправляемую с дескриптором, можно запомнить в одном из неиспользуемых дескрипторов, а затем восстановить его. Следующий пример представляет собой усовершенствование одного из примеров, данных выше:

```
#!/bin/bash
echo Эта строка выводится на экран
# запомнить дескриптор 1 в дескрипторе 3, а 1 перенапавить в file.txt:
exec 3>&1 1>file.txt
echo А эта строка в файл
echo И эта в файл
# восстановить предыдущее значение дескриптора 1:
exec 1>&3
echo Эта строка опять на экран
```

-- выведет:

```
Эта строка выводится на экран
Эта строка опять на экран
```

-- а две другие строки запишет в файл file.txt
Известно, как передать через pipe и stdout, и stderr:

```
command1 2>&1 | command2
```

А вот как можно передать информацию со stderr программы через pipe другой программе, оставив при этом stdout:

```
command1 3>&1 2>&1 1>&3 | command2
```

Здесь stdout (1) запоминается в дескрипторе 3, затем stderr (2) перенаправляется в stdout (1), а сам stdout перенаправляется в 3 (запомненное значение дескриптора 1). Поскольку через pipe передается только информация, выводимая через дескриптор 1, дескриптор 3 (которому ранее было присвоено значение stdout) останется нетронутым.
Вот пример из [[csh.whynot]] , иллюстрирующий этот прием:

```
grep yyy xxx 3>&1 2>&1 1>&3 3>&- | sed
s/file/foobar/
```

-- выведет:

```
grep: xxx: No such foobar or directory (ENOENT)
```

закрытие дескриптора

Закрытие дескриптора осуществляется дублированием его псевдодескриптором " &-":

```
#!/bin/bash
exec 1>&-
echo
Этот текст никто не увидит
```

КОМАНДЫ ИЗ НАБОРА SHELL-UTILS

Некоторые из перечисленных здесь команд обычно являются одновременно внешними командами из набора SHELL-UTILS и встроенными командами bash.
Подробное описание команд из этого набора см. в документации к GNU SHELL-UTILS.

команды echo, pwd, test

Внешние команды, echo, pwd и test дублируют встроенные команды bash и немного от них отличаются. Например, встроенная команда BASH echo не воспринимает escape- последовательности как таковые, если не указать специальную опцию "-e".
То есть, если написать 'echo "message1\nmessage2"', bash не заменит "\n" на новую строку, а просто выведет эти символы на экран. Внешняя же команда echo всегда обрабатывает escape- последовательности и не требует ключа "-e".
Чтобы вызвать внешнюю команду вместо встроенной, нужно или указать путь, например "/bin/echo" вместо "echo", или запретить использование некоторой внутренней команды (enable -n echo) , и тогда bash будет искать внешнюю.
Проверить, дублируется ли некоторая внутренняя команда внешней, можно с помощью " type -a команда".

let

Вычисляет арифметическое выражение и возвращает 0, если оно не равно нулю, и не-0 -- иначе. Среди прочих операций есть присваивание (=) можно присваивать значение переменной. В этом случае имя переменной должно быть без префикса "$", в прочих случаях можно и с "$" и без него.

[ ], test & /usr/bin/test

Проверяет истинность условия (тестирует условие).
Существует в виде как встроенной, так и внешней команды. Если вместо слова test используются квадратные скобки, они обязательно должны быть отделены от аргументов пробелом, потому что на самом деле "[" ≈ это название команды, а "]" ≈ обязательный последний аргумент этой команды.
Можно комбинировать и отрицать условия:
[условие1 - a условие2] Логическое И
[условие1 - o условие2] Логическое ИЛИ
[! условие] Логическое НЕ
Основные тесты:
-t [fd] истина, если file descriptor fd (по умолчанию 0) не перенаправлен
Напр. '! - t 0 ' -- осуществляется ли ввод с клавиатуры
Файлы:
-e file Истина, если существует
-s file Истина, если имеет ненулевой размер
Пустая директория также может иметь ненулевой размер. Поэтому проверять, пуста ли директория, надо примерно так (пример из comp.unix.shell):

```
dir=${1:-.} # for a standalone script
set -- $dir/.??* $dir/.[!.] $dir/*
case $#$* in
  "3$dir/.??* $dir/.[!.] $dir/*")
    echo empty ;;
  *)
    echo not empty ;;
esac
```

-d file Директория
-f file Обычный файл (не директория)
file1 -ot file2 Старее (older than)
file1 -nt file2 Новее (newer than)

Строки:
-z string Пустая
-n string Не пустая
string1 = string2
string1 != string2
Числа:
arg1 -eq arg2 Равно
также -ne, -lt, -le, -gt, -ge
(not equal, less than, less or equal, greater than etc.)

expr (внешняя команда)

Выводит результат выражения
A == B
A + B
A && B
A || B
length string
substr string index length
match string regexp или string : regexp
Выводит номер символа, с которого начинается совпадение. Если в выражении есть "\(\)", вместо номера печатается часть строки, совпавшая с выражением в скобках.
В отличие от test, expr обычно не является встроенной командой, а имеется только в отдельном исполняемом файле. **Часто лучше не использовать!** Может замедлить выполнение скрипта в десятки раз. Везде, где это возможно, лучше применять test или let.
Например, вместо:

```
expr $i + 1 == $n && break
```

лучше писать:

```
let i+1==n && break
```

или:

```
[ $[$i+1] = "$n" ] && break
```

tee

Название английской буквы " T". T-shaped pipe splitter.
Перенаправляет вывод стразу во много мест (файлов)
Ключ " -a" заставляет дополнять существующие файлы, а не затирать.

date

Команда date не является командой bash, а входит в набор SHELL-UTILS. Однако представляется нелишним описать ее здесь, ввиду ее исключительной полезности для некоторых задач, традиционно решаемых написанием shell-скриптов, а также потому, что по причине обилия возможностей она имеет довольно нетривиальный синтаксис. Кроме описания, ниже будут даны примеры решения практических задач, связанным с обращением к данным о дате и времени, которые взяты из сообщений в comp.unix.shell
К сожалению, помимо сложного синтаксиса работу с date затрудняет еще и то, что разные реализации date могут иметь опции с одинаковым названием и совершенно разными функциями. Приводимые ниже примеры тестировались на /bin/date из FreeBSD 2.2.5. В некоторых случаях в примерах указаны также отличия этой реализации от date из набора GNU SHELL-UTILS
Формат вывода даты задается например так

```
date +%d.%m.%y
```

Ключ -v (-d в GNU date) позволяет напечатать не сегодняшнюю дату, а, например, завтрашнюю:

```
date -v +1d          # /bin/date
date -d "+1 day"  # то же для GNU date
```

или какая была месяц назад :

```
date -v -1m           # /bin/date
date -d "-1 month"  # GNU date
```

Вот такая команда, записанная в скрипте /etc/daily будет копировать каждый день выпуск сегодняшних новостей newsDDMMYY.txt в файл newstoday.txt:

```
cp `date +news%d%m%y.txt` newstoday.txt
```

Работа с именами файлов: basename, dirname, pathchk

basename -- имя файла без пути; если в качестве второго документа задать расширение, оно тоже отрезается
dirname -- пусть к файлу без самого имени файла
pathchk -- проверить, допустимо ли имя файла или пути в данной системе
Внимание! Так же как при работе с expr (см.) и другими внешними командами, следует избегать частого применения данных утилит по причине замедления ими работы. dirname и basename без каких либо трудностей можно реализовать с помощью операций над переменными. В следующем примере определяются функции dirname и basename, идентичные по своему поведению одноименным утилитам dirname и basename, и демонстрируется их применение:

```
basename()
{
 local name="${1##*/}"
 echo "${name%$2}"
}

dirname()
{
 local dir="${1%${1##*/}}"
 [ "${dir:=./}" != "/" ] && dir="${dir%?}"
 echo "$dir"
}

fullpath=${1:-/dir/file.ext}
name=`basename "$fullpath"`
dir=`dirname "$fullpath"`
echo "file '$name' is located in $dir"
```

-- выведет:

```
file 'file.ext' is located in /dir
```

Для того чтобы всегда вызывать эти функции вместо соответствующих внешних утилит, можно переименовать их в basename и dirname и записать в отдельный файл, например path.sh, а в файлы с использующими эти функции скрипты, добавить ". path.sh" (см. команду "." и source).
Полезными могут также оказаться функции для определения расширения имени файла (назовем ее ext) и имени файла без расширения (namename):

```
function ext
{
 local name=${1##*/}
 local name0="${name%.*}"
 local ext=${name0:+${name#$name0}}
 echo "${ext:-.}"
}

function namename
{
 local name=${1##*/}
 local name0="${name%.*}"
 echo "${name0:-$name}"
}
```

Обратите внимание на то, что ни одна из показанных выше функций не вызывает других функций из этого же набора, хотя, например, dirname могла бы вызвать basename и отрезать от полного пути полученное из basename значение, однако такой вызов не осуществляется из соображений оптимизации скорости работы этих функций.

Другие внешние команды

true
false
yes
printf
tty -- чем является стандартный ввод (tty / not tty)
sleep -- заснуть на столько-то секунд

nohup -- запуск программы, продолжающей работать после logoff

Информация о пользователях:

id, logname, whoami, groups, users, who

Информация о системе:

uname, hostname

ENVIRONMENT И АТРИБУТЫ БАША

Environment ≈ массив переменных, который наследуется вызываемыми башем программами. Это не все переменные, а только те, которые унаследовал сам баш, и те созданные в баше переменные, которым был присвоен признак "экспортируемых".
Атрибуты баша, или флаги, включают особые режимы функционирования баша. Устанавливаются командой set (см.).

export

Команда export и используется добавления переменных к environment.
Переменные, созданные в баше, не входят автоматически в environment (массив переменных, наследуемый запускаемыми программами). Однако это можно изменить с помощью команды set (см.).
Экспортировать можно как переменные, так и функции (-f).
В принципе, команда export не нужна, так как вместо нее можно использовать declare -x.
Чтобы изъять переменную из environment, нужно использовать

```
export -n var
```

declare, typeset

typeset -- устаревшее, но разрешенное название declare. Используйте это название, если хотите обеспечить с совместимоть с Korn shell.
Без аргументов выводит значения всех переменных. Если вызвана таким образом из скрипта, ведет себя довольно странно: сначала выводит текст скрипта, предшествующий команде set, затем все переменные, а затем ≈ текст скрипта, следующий за командой.
Если нет аргументов, но указаны опции, выводит определения переменных, имеющих такие опции, а именно:

-a -- массивы

-f -- функции

-i -- целые числа

-r -- только для чтения

-x --
экспортируемые

Если есть аргументы, то создаются переменные с указанными опциями или изменяются опции существующих. Если опция указана со знаком " +" вместо "-", она удаляется. Переменным попутно можно присвоить значения.
Пример:

```
declare -irx MAGIC_NUMBER=666
```

-- создает переменную MAGIC_NUMBER, которую нельзя изменять, которая всегда будет рассматриваться как целое число и которая автоматически добавляется к environment , как если бы была выполнена команда export. Кроме того, переменной присваивается значение.
Экспортировать и делать только для чтения можно не только переменные, но и функции (-f).
Внутри функций declare создает локальные переменные, аналогично local. В ksh (и в нек. реализациях sh?), не имеющем команды local, для создания локальных переменных можно использовать синоним declare typeset.
Опция опция -p показывает атрибуты переменной или всех переменных, если переменная не указана.

```
declare -r
```

выведет что-нибудь вроде:

```
declare -ri EUID="42"
declare -ri PPID="1"
declare -ri UID="42"
```

-- однако просто declare выведет переменные без указания их признаков. Зато:

```
export -p
```

работает так, как можно было бы ожидать от declare -p

readonly

Делает переменную или функцию (-f) read only или выводит список таких переменных (-p) .
Аналогичную вещь делает 'declare -r', так что эта команда лишняя.

unset

Отменяет определение переменной или функции (-f).

set

Опции, переданные set , устанавливают атрибуты самого баша. Если вместо "-" перед атрибутом идет "+", атрибут сбрасывается. Если есть аргументы, set присваивает их позиционным параметрам $1, $2 и т. д.
Без опций и аргументов выводит все переменные. Если вызвана таким образом из скрипта, ведет себя довольно странно, как и команда declare (см.).
Атрибуты могут задаваться в короткой или длинной форме. В длинной форме ≈ через опцию -o.
Опция -o без параметров выводит значения всех атрибутов:

```
set -o
```

выведет что-нибудь вроде:

```
allexport       on
braceexpand     on
emacs                   on
errexit                 off
histexpand      on
ignoreeof       off
interactive-comments    on
monitor                 off
noclobber       off
noexec                  off
noglob                  off
nohash                  off
nounset                 off
physical        off
posix           off
privileged      off
verbose                 off
vi              off
xtrace          off
```

Интересные атрибуты:
emacs (нет короткого названия) -- расширенные возможности редактирования. В интерактивных шеллах установлена по умолчанию.
vi (нет короткого названия) -- возможности редактирования, аналогичные командам vi
a (allexport) -- автоматический экспорт всех переменных

e (errexit) -- прекращать работу, если команда, не входящая в группу, или группа команд возвратила ненулевое значение. При прекращении работы не выдается никаких сообщений.
f (noglob) -- запрещает обрабатывать маски (*, ? и т. д.).
h (hashall) -- запоминать пути к выполненным командам (по умолчанию)
n (noexec) -- не выполнять команды, только проверить их правильность
v (verbose) -- аналог echo on в ДОС
x (xtrace) -- расширенный аналог verbose
n (nounset) -- рассматривать использование неопределенной переменной как ошибку
C (noclobber) -- см. про переменную noclobber
P (physical) -- при смене текущей директории на линк, делать именем текущей директории не имя линка, а имя физической директории.
"--" -- если за этим ничего не следует, unset позиционные параметры.

ВСТРОЕННЫЕ КОМАНДЫ (SHELL BUILTINS)

команда ":" (двоеточие)

Осуществляет анализ командной строки, но не выполняет ее. Всегда завершается успешно.

```
#!/bin/bash
: $[a=2*2]
echo "a=$a"
```

-- выведет:

```
a=4
```

Значение переменной $a было присвоено в результате [[побочного эффекта]] анализа, выполненного командой ":".

source или "." (точка)

Выполняет команды из указанного файла в текущем баше, без запуска нового, как если бы содержимое файла набиралось на клавиатуре. Это позволяет, в отличие от выполнения скрипта, запомнить изменения в environment, определения функций и прозвищ. Используя эту команду, в программу можно включать "библиотечные" функции, определенные в других файлах.

enable

Запрещает (-n) или снова разрешает применение встроенной команды.
Когда запрещается, то вместо встроенной команды ищется файл, если его нет -- ошибка

exec

Если имеется аргумент, заменяет текущий процесс (bash) на выполняемую команду, и к башу возврата уже не происходит:

```
PerlProgram $scriptname && exec perl $scriptname
```

Если аргумент не указан, но указаны символы перенаправления, exec устанавливает эти перенаправления, и выполнение текущего скрипта продолжается. Подробнее см. [[Перенаправление]]
Таким образом достигается [[динамическое]] перенаправление.

```
echo Эта строка выводится на экран
exec >file.txt
echo А эта строка в файл
echo И эта в файл
```

Более того, можно открывать для ввода или вывода новые дескрипторы:

```
# создать файл-пример для последующего чтения
cat << EOF >user.rsp
```

```
29.06.77
30.01.85
EOF
exec 3>file.log
exec 4<user.rsp
echo Вводим дату через дескриптор 4
read d <&4
echo Дата получена и будет записана через дескриптор 3
echo $d >&3
```

getopts

getopts названия-опций переменная [альтернативные-аргументы]
Действует аналогично Си-функции getopt . Названия опций состоят из букв, соответствующих опциям. Если опция требует аргумент, после буквы ставится двоеточие.
Понимает "--" и дальше опции не смотрит Ни фига не понимает длинные . опции (--опция) .
! Если есть альтернативные аргументы, то анализируют их, а не $1-$n
Если опция имеет аргумент, он помещается в OPTARG.
OPTIND сначала равен 1, затем гетопт его увеличивает. Если надо запустить гетопт снова, надо снова присвоить единицу.
После первого неудачного гетопта ($? не равен 0) $OPTIND равен номеру первого аргумента, не являющегося опцией, если такой есть ($OPTIND <= $#). Чтобы обработать этот и последующие аргументы, нельзя писать ${$OPTIND}, поскольку номер параметра обязан быть константой. Вместо этого нужно сделать

```
shift $[$OPTIND-1]
```

и обращаться к $1 и т. д.
Если ошибочная опция, в переменной присваивается "?", и выдается сообщение об ошибке. Чтобы запретить вывод сообщений, надо или OPTERR=0 (по умолчанию 1), или первый символ в названиях опций ":".
Пример:

```
while getopts "qhe:v" arg
do
  echo "Opt: $arg ${OPTARG:+($OPTARG)}"
  case $arg in
    q) quiet=1;; # установить quiet mode
    h) echo Using: $0 [-q] [-h] [-e scriptfile] FILES...;
       exit;;
    e) scriptfile=OPTARG;; # задать файл с программой
  esac
done
shift $[$OPTIND-1]
while [ -n "$1" ]
do
  echo Processing file $1...
  # разные действия...
  shift
done
```

hash

Показывает таблицу запомненных расположений команд. Можно стереть (-r), вопрос только в том, зачем то и другое.

pushd, popd, dirs

dirs выводит стек запомненных директорий
popd удаляет или последнюю запомненную, или такую-то по счету сверху стека (+0..n) или снизу (-0..n).

type

Похожа на команду which.
Информация о том, как проинтерпретировалась бы команда, будь она набрана в командной строке: Полное имя файла или сообщение типа 'shell builtin' или 'aliased to...'.

можно только тип (-t): file, builtin, alias, keyword
Можно вывести все альтернативы (-a), например type -a ls" вернет "/bin/ls и ls is aliased to `gnuls --color=tty'"

ulimit

Показывает или устанавливает ограничения на что-либо. Можно показать все ограничения (-a)

umask

Атрибуты, с которыми создаются файлы данного пользователя. Без параметров показывает текущие в виде числа или букв (- S)

alias, unalias

создают и удаляют прозвища

ИСТОРИЯ КОМАНД И КОМАНДЫ ИСТОРИИ

history

Показывает всю историю, или последние n команд.

символ !

!! -- Предыдущая команда
!-n -- энная с конца команда
!n -- энная с начала команда
!строка -- одна из последних команд, начинающаяся строкой
!?строка -- -"- содержащая строку

^строка1^строка2 -- последняя команда, в которой строка1 заменена на строку2
команды fc и многие другие, пока лень разбираться.

ПЕРЕМЕННЫЕ

PS1, PS2, PS3, PS4, PROMPT_COMMAND

Подсказки, по умолчанию:
PS1 "bash\$"
Если пусто ([-z "$PS1"]) , шелл не интерактивный.

```
[ -n "$PS1" ] && echo interactive || echo batch job
```

PS2 ">" -- запрос продолжения ввода
PS3 "" -- подсказка при read
PS4 "+" -- команда, выполненная в режиме трассировки (set -x)
PROMPT_COMMAND -- команда, выполняемая всякий раз перед показом PS1
Есть динамически вычисляемые escape-последовательности:
\w -- текущий каталог (полный путь)
\W -- текущий подкаталог, без пути к нему
\u -- user
\h -- host
\$ -- "$" для обычного пользователя и "#" для root

и т. д.

Другие

RANDOM
PWD
OLDPWD
REPLY -- см. read и select
BASH -- путь к текущему башу
SHLVL -- номер копии баша ?
SECONDS -- сколько секунд запущен баш
LINENO -- номер строки в текущем скрипт-файле или функции
BASH_VERSION -- версия баша
HOSTTYPE -- тип машины, напр.: "PCAT" (IBM PC AT), "i586"
OSTYPE -- тип операционной системы, напр.: "MSDOS", "linux", "freebsd2.2.5"
MAIL -- файл, в который приходит почта, если не задано, что-нибудь вроде
/var/mail/user
MAILCHECK -- через сколько секунд проверять почту, по умолчанию 60

OPTARG, OPTIND, OPTERR

Результаты последнего getopts (см.)

булевые

nolinks -- если установлена, превращать symlinks в настоящие файлы
noclobber
выдавать ошибку при попытке перенаправить (">") в существующий файл, но можно использовать ">|"
И другие...

ОПЕРАТОРЫ

другие

until test-commands; do consequent-commands; done
while test-commands; do consequent-commands; done

if

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```

Вместо if можно также использовать символы группирования команд " &&" и " ||":

```
[ -e config.txt ] && echo configuring... || echo no config
```

for

for name [in words ...]; do commands; done
Пример.
Склеить текстовые файлы в один, указав для каждого файла его название:

```
echo "{directory $PWD}" > texts.lst
for fn in *.txt
do
  echo "{file $fn}" >> texts.lst
  cat $fn >> texts.lst
done
```

Если "in ..." не указано, проходит по всем параметрам (т. е. in "$@")
Пример.

```
for fn # список не указан, подразумевается "$@"
do
  echo Processing file $fn...
  # разные действия...
done
```

select, read

Делает меню для выбора:

```
select a in strings;
do if [ -n "$a" ] # строка непуста только при допустимом выборе
   then echo $a eto klass
   break
else
   echo $REPLY eto ne otvet  # то что было введено сохраняется в $REPLY
fi;
done
```

read без аргументов тоже запоминает строку в REPLY

case

```
case слово in
 [значение [| значение]...) команды;;]...
esac
```

значение может содержать ? и *
Пример.

```
for name in a.gz b.tar c.zip d.txt /kernel
do
  echo -n "$name: "
  case $name in
    /kernel) echo "Ядро";;
    *.txt) echo "Текстовый файл";;
    *.gz|*.tar|*.zip) echo "Архив";;
    *.c) echo "Программа на Си";;
    *.cc|*.C|*.cpp|*.cx) echo "Программа на C++";;
    *) echo "А это еще что?";;
  esac
done
```

ФУНКЦИИ

Да, есть такие.
Синтаксис вызова функции такой же, как у скрипта. Как и скрипт, функция может иметь аргументы, и, аналогично скрипту, обращаться к ним через ${n}. Функция не может обращаться к аргументам командной строки скрипта, в файле которого она сама находится, так как ${n} показывает на ее собственные аргументы.
Может возвращать значение через return. Отсутствие значения или всего оператора return эквивалентно наличию return 0.

function, return

Пример:

```
function Warning()
{
  echo "Warning: $1"
  echo "continue(y/n)?"
```

```
 read answer
 if [ o"$answer" != o"y" ]; then
       return 1          # 1 Значит "ошибка, ложь"!!!
 else
     return 0  # 0 -- Удачное завершение, истина!!!
 fi
}
if Warning "the string is empty!"; then
       echo OK;
else
       echo Operation aborted
fi
```

Если поместить некоторую функцию А в теле функции Б, то реально А будет определена только после вызова Б. Но после этого А будет видна глобально, если в конце Б не сделать unset.

local

объявляет локальные переменные внутри функций, и может сразу задать им значение

```
local a
local b=kuku
```

ИОО: Иногда отвечаемые ответы

shells FAQ

Источник: faq.shell из comp.unix.shell
В источнике 79 ответов, разбитых на 7 разделов. Полезные ответы, имеющие отношение к башу, приводятся ниже.

1.2 Что значит "{некоторое странное название команды}"

cat = "catenate"
awk = "Aho, Weinberger and Kernighan"
grep = "Global Regular Expression Print" (аналог команды ed "g/re/p")
fgrep = "Fixed GREP"; fgrep searches for fixed strings only.
Perl = "Pathologically Eclectic Rubbish Lister"
Unics (Unix) = "UNiplexed Information and Computing System"
gecos = "General Electric Comprehensive Operating Supervisor"
biff = "BIFF"

> "... This command, which turns on asynchronous mail notification, was actually named after a dog at Berkeley."

rc (напр. ".bashrc" или "/etc/rc") = "RunCom"

2.5 Как считать со стандартного ввода один символ

Используя команды dd и stty:

```
stty -icanon
echo -n "Enter a character: "
readchar=`dd if=/dev/tty bs=1 count=1 2>/dev/null`
echo "Thank you for typing a $readchar ."
stty icanon
```

Про dd см. в документации к SHELL-UTILS

2.6 Как переименовать все файлы *.html в *.htm

мой ответ:

```
find . -name '*.html' | while read a; do mv $a ${a%.html}.htm; done
```

2.12 Как обратиться к параметру номер i , если i не константа, а переменная?

Можно подумать, что это легко и можно написать ${$i}, но это не сработает, и вам напишут bad substitution.

- Примечание. Начиная с bash 2.0 можно использовать конструкцию ${!i}.

Зато можно вот так:

```
eval "param=\${$i}"
echo "Parameter number $i is $param"
```

А вот программа, которая выводит все свои параметры в обратном порядке, (C) 1997 ГС:

```
i=$#
while let i
do
  eval "echo \${$i}"
  i=$[i-1]
done
```

BASH FAQ

Источник: bash.faq из comp.unix.shell

Последняя версия BASH

Самая последняя версия bash -- 2.01. Она почти не отличается от 2.0, зато довольно сильно отличается от версии 1.14.7, предшествующей 2.0.
Новые возможности по сравнению с 1.14.7:

- зарезервированное слово time для вычисления времени выполнения программы
- одномерные массивы
- новые виды экспансии: ${param:length:offset}, ${param/pat/replace-to}
- новые команды disown и shopt
- новые виды кавычек для locale-specific translations, обозначения $'...' и $"...".
- ссылки на переменные: {$! param_name } будет вычислена переменная с именем, хранящимся в $ param_name (в старом для этого нужен eval)

Чем BASH отличается от SH

В SH отсутствуют следующие возможности BASH:

- новые возможности BASH 2.0 (см. выше)

- long named options (для вызова bash)
- отрицание !
- слово function (по слухам, в некоторых реализациях sh это слово есть)
- команда select
- обращение к параметрам больше девятого ${10}
- длина параметра ${#param}, "редактирование" ${var#str}, ${var%str}
- нет переменных: BASH, BASH_VERSION, BASH_VERSINFO, UID, EUID, **REPLY** , TIMEFORMAT, PPID, **PWD** , OLDPWD, SHLVL, **RANDOM** , SECONDS, LINENO, HISTCMD, **HOSTTYPE** , **OSTYPE** , MACHTYPE, **HOSTNAME** , **ENV** , **PS3** , **PS4** , DIRSTACK, PIPESTATUS, HISTSIZE, HISTFILE, HISTFILESIZE, HISTCONTROL, HISTIGNORE, GLOBIGNORE, PROMPT_COMMAND, FCEDIT, FIGNORE, IGNOREEOF, INPUTRC, SHELLOPTS, **OPTERR** , HOSTFILE, TMOUT, histchars, auto_resume
- перенаправление <>, &>, >|
- echo -e (в некоторых реализациях имеется), hash -p (то же), type -apt,
- test -o optname/s1 == s2/s1 < s2/s1 > s2/-nt/-ot/-ef/-O/-G/-S
- разные конфигурационные файлы для интерактивного или неинтерактивного сеанса: ~/.bashrc и $ENV соотв .
- функции и переменные с одинаковыми именами
- { }- экспансия: {префиксы}корень{суффиксы}
- обозначение ~ (имеется в некоторых реализациях)
- let (в некоторых реализациях есть команда с таким же именем, но отличающимся поведением)
- $((expr)) -- только $[] или вообще нет ?
- вместо $(...) только `...`

- подстановки процессов >(cmd) <(cmd)
- alias и unalias (имеются в некоторых реализациях)
- local (в некоторых реализациях есть команда с таким же именем, но отличающимся поведением)
- история команд (в некоторых реализациях)
- команды command, builtin, declare, typeset, dirs, enable, help, logout, popd, pushd
- экспортируемые функции
- filename generation when using output redirection (command >a*)

16.12.1997-23.09.1998, Григорий Строкин (grg@philol.msu.ru)