

Промяна по системата и писане на малки скриптове.

Thursday, August 27, 2020 4:19 PM

Environmental variables

Такива които са дефинирани за текущата сесия в интерпретатора. Наследени са от всички процеси "деца".

Shell variables

Такива съдържащи се само в определената среда в която са дефинирани.

Команди:

`printenv` - ще покаже всички `environmental variables`.

`set` - показва `SHELL` функции и `Env. Variables`.

`set -o posix`

`set | less` ще покаже всички `shell variables` без функциите

`set -o` ще покаже всички опции

Login Shell Process:

<code>/etc/profile</code>	Systemwide environment and shell variables
<code>/etc/profile.d/*</code> <code>.sh</code>	Systemwide environment and shell variables
<code>~/ .bash_profile</code>	User environment and shell variables
<code>~/ .bashrc</code>	Executes <code>/etc/bashrc</code>
<code>/etc/bashrc</code>	Systemwide aliases and shell functions
<code>~/ .bashrc</code>	User aliases and shell functions

*Login shell - такава която и трябва име и парола

Ако правим нещо много често можем да го добавим в `startup` скрипт.

Bash ред на екзекуция

1. Aliases
2. Bash Functions
3. Bash build-in commands
4. On-disk commands

Ако напишем `command` във интерпретатора `SHELL` ще потърси всички директории за `$PATH` променливата

```
vi ~/.bashrc
```

```
mkcd () {  
    mkdir -p $1  
    cd $1  
}
```

```
source ~/.bashrc
```

Можем да проверим дали е добавена със set.

Тази функция добавена в bashrc ще прави папка и ще влезе в нея директно. Където \$1 е името на нашата папка командата ще изглежда реално `mkdir test cd test`.

Можем да видим една команда какво е със `type -a mkcd`

Ще направим една папка която ще съдържа нашите скриптове и ще бъде в PATH

```
mkcd scripts  
vim ~/.bashrc
```

```
PATH=$PATH:~/scripts  
export PATH
```

Това ще добави папката към PATH.

По принцип за да може да се запази трябва да се разлогнете, но бързото е

```
source ~/.bashrc
```

```
echo $PATH | grep scripts
```

За бърза проверка, че всичко е точно.

Когато нов потребител е създаден той използва skel директорията.

Skel се знае още като skeleton директорията.

```
cd /etc/skel
```

Ще видите :

```
ls -lah
```

```
total 8.0K
```

```
drwxr-xr-x 1 root root 4.0K Aug  1 16:29 .
```

```
drwxr-xr-x 1 root root 4.0K Aug 29 11:29 ..
```

```
-rw-r--r-- 1 root root 220 Feb 25  2020 .bash_logout
```

```
-rw-r--r-- 1 root root 3.7K Feb 25  2020 .bashrc
```

```
-rw-r--r-- 1 root root 807 Feb 25  2020 .profile
```

По принцип мислете тази папка като гръбнака на вашият user.

И когато копирате вашите .bashrc и .profile в skel на нов потребител от примерно вашият потребител той ще има вашите настройки на

.bashrc .Съдържанието на директорията осигурява среда по подразбиране на новосъздадените потребители.

BASH е обикновената черупка в Enterprise Linux 7.
Можем също да го наречем и обикновеното возило за създаване на скриптове.

BASH като бивайки версия 4.2 съдържа следните способности:1

- if,then,else conditionals
- case statements
- conditional tests for:

Собственик на файл.

Права на файл.

Цифрово изравняване.

Текстово.

Файловия тип е файл

Файловия тип е директория.

Файловия тип е блок устройство или знак.

Файла се чете, пише и зарежда.

Файла съществува или папката..

- for loops.
- C style for loops
- While/Until Loops with conditions
- Brace expansion
- Arithmetic Expansion
- Command substitution
- Integer Math
- Mathematical conditions
- Indexed Arrays
- Associative Arrays
- Extended Globs
- Extended Regular expressions.
- Functions
- Substring operations
- Parameter substitution
- Increment/decrement operators
- Debug Traps
- Positional arguments
- Subshells
- Co-process

BASH е също добър за автоматизация на рутинни задължения или необходимост.

И приказва с операционната система.

BASH не става за:

- Справяне с XML или JSON файл.

Скриптинг:

В началото на всяка линия .sh баш скрипт трябва да се съдържа следното:

```
#!/bin/bash
```

"При изчисленията shebang е символната последователност, състояща се от знака с номер на знака и удивителен знак (!) В началото на скрипта.

#!/bin/bash посочва че ще екзекутираме bash скрипт.

```
#!/usr/bin/env python3
```

Тук се посочва usr папката, бинарката (от там ще зареди) env за да се посочи environmental променливата.

Compound commands / Съставни команди :

Това е сбор от команди които извършват едно цяло нещо.

```
mkdir newfolder && cd new folder
```

По време на скрипт командите ще бъдат последователно изпълнени.

Ако искате да изпълните втората команда дори да има провал това което трябва да направите е следното:

```
mkdir newfolder && cd newfolder ( Представете си, че в тази ситуация нямате достатъчно права и се налага да пуснете и втората команда)
```

В ситуацията трябва да ползвате || което посочва, че въпреки неуспеха на първата втората команда ще трябва да се състои.

Пример:

Ако по някаква причина вие сте попаднали в директория без права и не го знаете можете да си направите следното, ако използвате скрипт:

```
mkdir newfolder && cd newfolder || echo "Directory creation failed"
```

Може да ползвате ; за да се изпълняват команди една след друга, но те трябва да са в кърдрави скоби:

```
{ echo "hi" ; echo "there"; }
```

Вероятно е да си помислите, че това прави същото като примерите по горе, но не е така:

Изпълнете:

```
echo "hi" ; echo "there" > output.txt
```

Ще видите, че командата която първо правите ще се появи в конзолата, а втората ще влезе във файла. Така, че е жизнено важно да се знае - точката и запетаята са като един вид отделения между двете команди. **Идеята тук е с цел да доказателство, че не може да се вземе изхода на командите и да се насочи в текстови файл.**

По интересен пример:

```
Ако имаме променливи-  
a=0  
(a=10; echo "in=$a") ;  
echo "out=$a"
```

Тук казваме на конзолата, че нашата променлива е 0 и след това стартираме, краткото изказване, че а може да е 10 и да питаме колко е а . Виждаме, че а 10 въпреки, че казахме на сесията, че е 0. Тоест скобите правят един вид малък затвор в който е отделен от сесията ни. Защото след като викнем отново а получаваме, че е 0.

Използване на команди и променливо заменяне:

Можем да вмъкнем променлива в текстови низ, или командния ред и да замени променливата стойност заместена с името на променливата.

Примерно пишем:

```
echo "My workspace is $USER"
```

Това ще ни покаже \$USER е environmental променлива и е "закачена" за инстанцията и тя винаги ще показва всичко в дадената login сесия.

Можем да извършим и следната дейност:

```
pdir="/tmp/files/september"  
fname="report"  
mkdir -p $pdir  
touch $pdir/$fname  
ls -l $pdir/$fname
```

Това може да се използва в един редовен скрипт - ако трябва всеки ден да се прави файл в тази папка.

Нека го направим по адекватен с цел редовната работа:

```
#!/bin/bash  
echo "Please enter the month"  
read VAR  
pdir="/tmp/files/$VAR"  
echo "please enter the date of your report in the following format: 00-dayofweek"  
read REPORT  
fname="$REPORT"  
mkdir -p $pdir  
touch $pdir/$fname
```

read е такава команда която ще прочете аргумента написан веднага след нея.

Това може да се нагласи точно в определено време с cron job и да се пуска например в началото на работната ви смяна.

Друг интересен пример е следното:

```
echo "Permissions for find are $(ls -l $(which find))"
```

Сега си го представете с един read и определени променливи и докато се усетите ще може да си проверявате определени сервиси по системата.

```
#!/bin/bash
```

```
echo "Enter tool you want to check permissions for"
read COMMAND
echo "Permissions for $COMMAND are $(ls -l $(which ${COMMAND}))"
```

Условни оператори

Синтаксис:

```
=====
If    <condition>
then
    <run code>
fi
=====
```

```
if <condition>; then
<run code>
Else
<run code>
fi
```

```
=====

if <condition> ; then
<run code>
elif <condition>; then
<run code>
fi
```

```
=====
```

Условните могат да бъдат един ви тест за нещо или дали една команда се изпълнила правилно.

```
if grep root /etc/passwd ; then
echo "Runs"
else
echo "does not run"
fi
```

Нека го направим малко по адаптивно спрямо нуждите ни:

```
#!/bin/bash
echo "Enter user that you are looking for"
```

```
read NAME
```

```
if grep $NAME /etc/passwd; then
    echo "User exists and are there"
```

```
else
    echo "User does not exist do you want to add user"
```

```
fi
```

```
if [ "$VAR" = 5 ]; then
    <run code>
fi
```

```
if [[ "$VAR" = 5 ]]; then
    <run code>
fi
```

[] (четат команди) единичните квадратни скоби са POSIX четливи (POSIX е съвкупност от стандарти на IEEE, проектирани да поддържат съвместимост между операционните системи, особено Unix-подобни системи.)

Работят с със стари интерпретатори като например Bourne и са команди които след това тестват условието.

&&, |, <, и > оператори биват интерпретирани от интерпретатора.

[] двойните квадратни скоби не са като единичните. Те се четат като ключови думи и падат под формата на POSIX. Специфични са за bash и ksh. Не работят със стари "черупки".

Поддържат &&, |, < и > оператори. Поддържат автоматичното пресмятане на октални и шестнадесетичен. И поддържат разширен вид на REGEX с цел намиране.

```
#!/bin/bash
```

```
echo -n "Enter a number: "
```

```
read VAR
```

```
if [[ $VAR -gt 10 ]]
then
    echo "The variable is greater than 10."
```

```
fi
```

```
#!/bin/bash
```

```
read MENUCHOICE
```

```
case $MENUCHOICE in
```

```
    [1-4]) echo "You are quite young" ;;
```

```
    [5-9]) echo "Time for elementary school" ;;
```

```
    1[0-9]) echo "Time for middle school" ;;
```

```
    2[0-9]) echo " You are an adult" ;;
```

```
    3[0-9]) echo " You are an adult" ;;
```

```
    [4-9][0-9]) echo "This is a complex short way of displaying it" ;;
```

```
esac
```

*преглед на

```
nick@fly:/mnt/c/Users/nykos/Documents/Scripts$
./casesample.sh
https://tldp.org/LDP/Bash-Beginners-Guide/html/sect\_07\_03.html
```

for loop или for цикъл/оператор

Когато искате да минете през лист от неща.

```
for item in <list>; do
    <work on $item>
done
```

Листа за този тип цикъл може да идва от каквито и да е източници.
Може да е статичен лист със имена или с последователност от числа.

```
for VARIABLE in 1 2 3 4 5 .. N
do
command1
command2
commandN
done
```

```
for VARIABLE in file1 file2 file3
do
command1 on $VARIABLE
command2
commandN
done
```

```
for OUTPUT in $(Linux-Or-Unix-Command-Here)
do
    command1 on $OUTPUT
    command2 on $OUTPUT
    commandN
done
```

```
#!/bin/bash
for file in /etc/*
do
if [ "${file}" == "/etc/resolv.conf" ]
then
countNameservers=$(grep -c nameserver /etc/resolv.conf)
echo "Total ${countNameservers} nameservers defined in ${file}"
break
fi
done
```

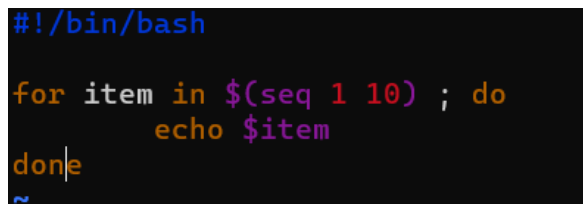
```
PKGS="php7-openssl-7.3.19-r0 php7-common-7.3.19-r0 php7-fpm-7.3.19-r0 php7-opcache-7.3.19-r0
php7-7.3.19-r0"
for p in $PKGS
do
    echo "Installing $p package"
```



```
sudo apt install "$p"
done
```

```
#!/bin/bash
```

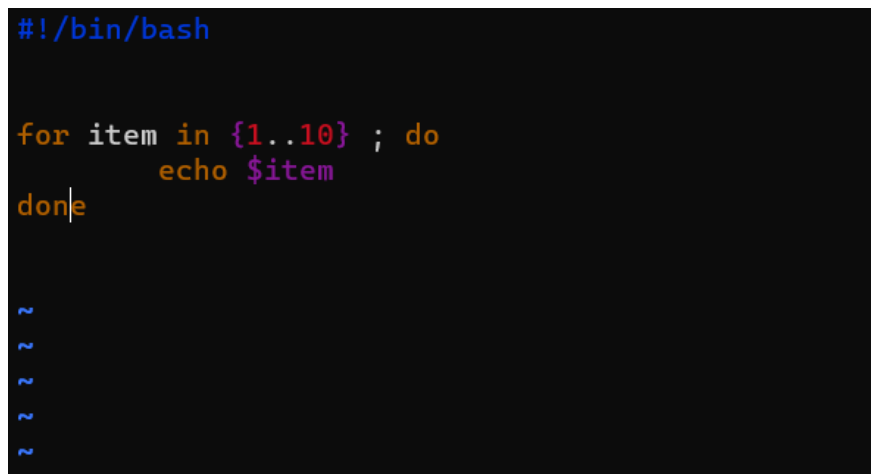
```
for item in $(find /etc);
do
    echo "$item"
done
```



```
#!/bin/bash

for item in $(seq 1 10) ; do
    echo $item
done
```

Ако търсите списък с последователни числа, можем да го създадем динамично. В миналото бихме използвали команда за последователност за това и бихме използвали заместване на команди, за да го накараме да работи. Въпреки това, по-добре е да имате Bash. Въпреки това е по-добре BASH да създаде списъка с помощта на разширяване.



```
#!/bin/bash

for item in {1..10} ; do
    echo $item
done
```

По този начин ние не призоваваме нова черупка (SHELL) и не изпълняваме команди. Просто е по сигурно и по бързо.

for цъкъл ползва - Вътрешен разделител на полето IFS с цел разделяне на думите. Следи дали има дупка в думата или файла дали има разстояние в името. Тоест ако има място в някой от файловете цикъла ще се развали. Тоест ще спре. Можем да променим IFS така, че да не се счупи отдолу е пример с нагласен IFS с цел да няма счупване.

```
OLDIFS="$IFS"
IFS=$'\n'
for file in $(find /etc) ; do
    echo "$file"
done

IFS="$OLDIFS"
```

while loop

До сега гледахме for loop която е с дизайн да мине през лист от неща и да свърши когато стигне своя край. Отделно е възможно да имаме условие (if/else/elif/fi) което би прекъснало процеса.

Пример:

```
for item in {1..10000} ; do
if [[ $item = 100 ]] ; then
break ;
fi
done
```

Защото условието винаги ще спре цикъла е по добре да използвате "loop" със вградено условие в себе си.

```
while [ condition ] ; do
    <Do stuff>
done
```

Имаме два типа цикли от този формат while и until.

while ще работи докато условието е валидно.

until ще работи докато условието не е валидно.

Поради факта, че употребява единични прави скобки посочва, че може да ползва wildcards, но не и reg-exи

Пример за безкрайна затворена верига е следното, ако пуснете подобен скрипт като този по долу той ще работи докато не го спрете.

```
while true ; do
if [[<condition> ] ] ; then
Break;
fi
done
```

=====

Пример на верига с условие:

```
i='0'
while [ $i -lt 4 ] ; do
echo "$i is still less than 4"
((i++))
done
```

=====

```
#!/bin/bash
```

```
#Този скрипт копира файлове от home директорията в webserver директорията
PICSDIR=/home/username/pics
WEBDIR=/var/www/username/webcam
```

```

while true; do
    DATE=`date +%Y%m%d`
    HOUR=`date +%H`
    mkdir $WEBDIR/"$DATE"

    while [ $HOUR -ne "00" ]; do
        DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
        mkdir "$DESTDIR"
        mv $PICDIR/*.jpg "$DESTDIR"/
        sleep 3600
        HOUR=`date +%H`
    done
done

```

```

=====
#!/bin/bash

```

This script opens 4 terminal windows.

```
i="0"
```

```

while [ $i -lt 4 ]
do
xterm &
i=$((i+1))
done

```

while loop се справя добре със IFS (internal field separator) Което означава, че ако боравите с файлове ще можете да, четете празните им места.
Обратни кодове (можем да ги наречем възможни грешки)

Всяка команда в баш връща някакъв вид код.
Този код дефинира дали командата е била успешна или не.
За да проверите дали командата ви е била успешна просто напишете следното в конзолата си:

```
echo $?
```

Тази команда ще ви покаже едно число.

Например напишете top и го затворете:

Пуснете echo \$?

И ще видите кода за успех в конзолата. (0)

Успех	0
Неуспех	1-255

Причината защо неуспех=1 до 255 е, защото процеса (командата или скрипта) може да се провали поради много причини. Идеята на това нещо е с цел да може да се прави поправка на ситуацията.

Напишете в конзолата `ls -lah /etc/passwd`
Ще върне 0

`cat /etc/password`
Ще върне 1 защото няма такъв файл.

Ако обаче напишете `ls -lah /etc/password`
Ще върне 2 защото не съществува файла.

Причината, защо техните кодове са различни е, че всяка команда има различна код към себе си за своята стандартна грешка.

Друг пример е ако напишем `catfile` - команда която не съществува.
Ще върне 127.

1	Обикновена грешка
2	Не правилно използване на команда
126	Командата не може да бъде изпълнена
127	Командата не може да бъде открита
128	Неправилен аргумент
128+n	Фатална грешка сигнал -n (n е сигнала)
130	Скрипт който е спрял със Ctrl-C
255+	Изходен статус извън обхват

Много от man страниците ще могат да ви кажат какви са обикновените кодове за изход.
Например `man ls`

```
Exit status:
  0      if OK,
  1      if minor problems (e.g., cannot access subdirectory),
  2      if serious trouble (e.g., cannot access command-line argument).
```

Нека сега направим един скрипт който употребява изхода в условие.
Ще го наречем `check.sh` и ще го направим в `~/bin/check.sh`.

```
#!/bin/bash
```

```
cat database.txt &> /dev/null
```

```
if [ [ $? -eq 0 ] ]; then
echo "Able to read the database"
else
    echo " Not able to read the database" >&2
fi
```

Пренасочването на края изпраща текста към обикновена грешка вместо стандартен изход.
За да можем да пренасочим грешният изход от нашият скрипт към друг лог.

Този скрипт се опитва да отвори файла database.txt и проверява дали е имало успех или не. Ако изхода е 0 ще каже, че имаме четене, а ако е друг ще каже - не.

Имайте в предвид, че би могло да е по полезно да използвате ситуационни условия вместо много if условия.

Ще изглежда грубо така:

```
if cat database.txt $> /dev/null ; then
echo "able to read database"
else
echo "unable to read database" >&2
fi
```

Може да използвате и функция:

```
#!/bin/bash

check_database () {
if cat database.txt &> /dev/null ; then
    echo "Able to read database"
    return 0
else
    echo "Unable to read database"
    return 1
fi
}
If ! check_database ; then
    exit 1
fi
```

В тази ситуация условието ни е само и само ако функцията е извършена. Ако не е успешна ще върне код 1.

На кратко, ако искате да върнете стойността на функция използвайте return, а ако искате да върнете стойността от скрипт използвайте exit.

Мониториращ скрипт който следи за провалени логини. Ако числото е по високо от прага зададен ще изпрати имейл.

```
#!/bin/bash
```

```
REPDATE=$(date --date='yesterday' +"%b-%d-%Y") ---> Взимаме датата от вчера в този формат Sep-04-2020
LOG=/var/log/secure* Специфицираме кой лог ще четем
THRESH="10" Прага на проваления логини които бихме позволили
```

```
REPORTEMAIL="report@localdomain.com" Имейла на който ще пращаме репорта всеки ден
ADMINMAIL="admin@localdomain.com" Имейла на администратора който се използва само и само ако прага минат
SUBJECT="User Authentication Rerport for $REPDATE" Тази линия е предметната линия която включва в себе си деня на анализа
MESSAGE="/tmp/message.txt" емейл файла на който ще добавим текста по късно в скрипта
```

```
ACOUNT=0 Броячите на проваления логини
RCOUNT=0 Броячите на проваления логини
```

DATE=\$(date --date='yesterday' +"%b %d") Получаваме деня отново, но без годината разделена без тирета (Правим го с цел да съвпадне лог формата на файловете)

ACOUNT=\$(grep -ic "^\$DAY.*authentication failure" \$LOG) търсим за провалени логини правиме не регистрирано търсене

RCOUNT=\$(grep -ic "\$DAY.*authentication failure.*euid=0" \$LOG) търсим за root логин провали

echo "Failed logins for \$REPDATE" >> \$MESSAGE Смятат се провалените опити и се изпращат във файла

echo "All failed logins attempts: \$ACOUNT" >> \$MESSAGE

echo "Root failed login attempts: \$RCOUNT" >> \$MESSAGE

if [[\$RCOUNT -ge \$THRESH]] ; then Проверка на колко провалени опити има през ден, ако е повече от прага се изпраща имейла.

mail -s "\$SUBJECT" "\$ADMINMAIL" < \$MESSAGE

fi

mail -s "\$SUBJECT" "\$REPORTMAIL" < \$MESSAGE