Bash Grundlagen

Wird ein Script ausgeführt, sorgt die Shebang Zeile (#!) dass die jeweilige Shell aufgerufen wird. Variablen, die vorher nicht exportiert wurden, werden in einer neuen Shell auch nicht mehr zur verfügung stehen. Beendet man das Script kann man die Variablen ebenfalls nicht mehr benutzen. Die „export“ Funktion steht auch nur für Subshells und nicht für übergeordnete Shells zur Verfügung.

Skripte, die für jeden Nutzer zugänglich sein sollen kann man in `/usr/bin/` oder `/usr/local/bin/` speichern. Skripte, die administrative Eingriffe ausführen, können in `/usr/sbin/` gespeichert werden.

Basis Knowhow und Aufbau eines Skriptes

Skripte ausführen

Ein Shellskript ist eine ausführbare Textdatei. Befindet sich das Skript in einem in der $PATH Variablen beschriebenen Verzeichnis, ist zum Ausführen des Skripts nur der Skriptname notwendig. Es gibt verschiedene Methoden ein Shellskript auszuführen.

Syntax:

- `/bin/bash [/pfad/zum/script.sh]`
- `source [/pfad/zum/script.sh]`
- `./script.sh`

Beispiel:

```
michael@home:~/workspace$ ./add.sh 7 7
Die Summe von 7 + 7 = 14
```

Die Datei muss natürlich ausführbar sein.

```
michael@home:~/workspace$ ls -lsa
insgesamt 4
-rwxr-xr-- 1 michael michael 104  7. Nov 16:27 add.sh
```

Die Ausgabe eines Skripts kann auch direkt in eine Mail umgeleitet werden.

```
michael@home:~$ ./bak | mail -s Backups root
```

Aufbau eines Skripts

```
#!/bin/bash
# Name: add.sh
# Addiere zwei Werte

let summe=$1+$2
echo "Die Summe von $1 + $2 = $summe"
```

In der ersten Zeile befindet sich das She Bang Zeichen, das zeigt, dass es sich um ein ausführbares Programm handelt und definiert mit welchem Interpreter das Skript ausgeführt wird. Danach folgt der Dateiname (Optional) und eine Kurzbeschreibung (Optional). Ab Zeile vier folgen die Anweisungen.

Bei „init“ Skripte sollte zur besseren Übersicht ein kompletter „head“ erstellt werden.

Beispiel:

```
#!/bin/bash

# xxx-Script

### BEGIN INIT INFO
# Provides:          xxx
# Required-Start:    $local_fs
# Required-Stop:     $local_fs
# Should-Start:
# Should-Stop:
# Default-Start:     S
# Default-Stop:      0 6
# Short-Description: xxx script
### END INIT INFO

# Anweisungen
...
```

Übergabevariablen und Rückgabewerte

Wenn beim Start eines Skripts Werte übergeben, oder ein Rückgabewert ausgegeben werden soll, können diese im Skript als Variablen aufgefangen werden. Dafür gibt es Automatische Variablen in

der Shell.

Übergabevariablen:

- $1 - $x ⇒ Werte die dem Skript übergeben wurden. Wird von $1 hochnummeriert.

Rückgabewerte:

- $0 ⇒ Gibt Pfad und Namen des ausgeführten Skripts aus.
- $# ⇒ Anzahl der übergebenen Parameter.
- $* ⇒ Zeigt alle Arumente aus der Kommandozeile in einer Zeichenkette.
- $@ ⇒ Zeigt alle Arumente aus der Kommandozeile als Arrays von Zeichenketten.
- $? ⇒ Errorlevel. Wird das Skript ohne Fehler ausgeführt ist der Errorlevel 0. Meisst wird bei einem Fehler 127 ausgegeben.
- $$ ⇒ Variable gibt die Prozessnummer aus.
- $! ⇒ Prozessnummer des zuletzt gestarteten Hintergrundprozesses.
- $_ ⇒ Letztes Argument in der Kommandozeile des zuletzt aufgerufenen Kommandos (nur Bash).

Beispiel:

```
#!/bin/bash
# Uerbergabevariablen ausgeben.

echo "Das Skript wurde mit dem Kommando $0 gestartet"
echo "Die Prozessnummer des Skripts ($0) lautet: $$"
echo "Es wurden $# Parameter uebergeben"
echo "Der erste Parameter lautet: $1"
echo "Der zweite Parameter lautet: $2"
Huchhh...
echo "Der aktuelle Errorlevel lautet: $?"
exit 0
```

```
michael@home:~$ ./uvar.sh 3 7
Das Script wurde mit dem Kommando ./uvar.sh gestartet
Die Prozessnummer des Scripts (./uvar.sh) lautet: 3332
Es wurden 2 Parameter uebergeben
Der erste Parameter lautet: 3
Der zweite Parameter lautet: 7
./uvar.sh: Zeile 8: Huchhh...: Kommando nicht gefunden.
Der aktuelle Errorlevel lautet: 127
```

Werte und Dateitypen Vergleichen mit "test"

„test“ ist ein Programm zum Dateitypen überprüfen und Werte vergleichen. Das Programm wird mit einem Exit-Status gemäss „AUSDRUCK“ beendet.

Syntax:

- `test [ AUSDRUCK ]`

Ausdruck:

- AUSDRUCK ⇒ AUSDRUCK ist wahr.
- `! AUSDRUCK` ⇒ AUSDRUCK ist falsch.
- `AUSDRUCK1 -a AUSDRUCK2` ⇒ AUSDRUCK1 und AUSDRUCK2 sind wahr.
- `AUSDRUCK1 -o AUSDRUCK2` ⇒ AUSDRUCK1 oder AUSDRUCK2 sind wahr.
- `-n ZEICHENKETTE` ⇒ Die Länge von ZEICHENKETTE ist ungleich Null.
- `ZEICHENKETTE` ⇒ Äquivalent zu -n ZEICHENKETTE.
- `-z ZEICHENKETTE` ⇒ Die Länge von ZEICHENKETTE ist Null.
- `ZEICHENKETTE1 = ZEICHENKETTE2` ⇒ Die Zeichenketten sind identisch.
- `ZEICHENKETTE1 != ZEICHENKETTE2` ⇒ Die Zeichenketten sind nicht identisch.
- `GANZZAHL1 -eq GANZZAHL2` ⇒ GANZZAHL1 ist gleich GANZZAHL2.
- `GANZZAHL1 -ge GANZZAHL2` ⇒ GANZZAHL1 ist grösser als oder gleich GANZZAHL2.
- `GANZZAHL1 -gt GANZZAHL2` ⇒ GANZZAHL1 ist grösser als GANZZAHL2.
- `GANZZAHL1 -le GANZZAHL2` ⇒ GANZZAHL1 ist kleiner als oder gleich GANZZAHL2.
- `GANZZAHL1 -lt GANZZAHL2` ⇒ GANZZAHL1 ist kleiner als GANZZAHL2.
- `GANZZAHL1 -ne GANZZAHL2` ⇒ GANZZAHL1 ist nicht gleich GANZZAHL2.
- `DATEI1 -ef DATEI2` ⇒ DATEI1 und DATEI2 haben dieselbe Geräte- und Inode-Nummern.
- `DATEI1 -nt DATEI2` ⇒ DATEI1 ist neuer (Änderungsdatum) als DATEI2.
- `DATEI1 -ot DATEI2` ⇒ DATEI1 ist älter als DATEI2.
- `-b DATEI` ⇒ DATEI existiert und ist ein »block special«.
- `-c DATEI` ⇒ DATEI existiert und ist ein »character special«.
- `-d DATEI` ⇒ DATEI existiert und ist ein Verzeichnis.
- `-e DATEI` ⇒ DATEI existiert.
- `-f DATEI` ⇒ DATEI existiert und ist eine reguläre DATEI.
- `-g DATEI` ⇒ DATEI existiert und das »Set-Group-ID«-Bit ist gesetzt.
- `-G DATEI` ⇒ DATEI existiert und gehört der effektiven Gruppen-ID.
- `-h DATEI` ⇒ DATEI existiert und ist ein symbolischer Link (identisch zu -L).
- `-k DATEI` ⇒ DATEI existiert und hat das »Sticky«-Bit gesetzt.
- `-L DATEI` ⇒ DATEI existiert und ist ein symbolischer Link (identisch zu -h).
- `-O DATEI` ⇒ DATEI existiert und gehört der effektiven Benutzer-ID.
- `-p DATEI` ⇒ DATEI existiert und ist eine benannte Pipe.
- `-r DATEI` ⇒ DATEI existiert und ist lesbar.
- `-s DATEI` ⇒ DATEI existiert und hat eine Grösse grösser Null.
- `-S DATEI` ⇒ DATEI existiert und ist ein »Socket«.
- `-t FD` ⇒ Datei-Deskriptor FD ist auf einem Terminal geöffnet.
- `-u DATEI` ⇒ DATEI existiert und das »Set-User-ID«-Bit ist gesetzt.
- `-w DATEI` ⇒ DATEI existiert und ist schreibbar.
- `-x DATEI` ⇒ DATEI existiert und ist ausführbar (oder darf gesucht werden).

Ausser den Tests `[-h]` und `[-L]` werden symbolische Links von allen DATEI-Tests dereferenziert. Zu bedenken ist, dass runde Klammern für Shells mit Back-Slashes maskiert werden müssen \(...\).

Escape Sequenzen

Der Parameter `echo -e` aktiviert Escape Zeichen bei der Ausgabe.

Escape Zeichen:

- `\\` ⇒ Soll ein Backslash angezeigt werden, muss er mit einem \ Backslash escaped werden.
- `\a` ⇒ Alert. Erzeugt einen Ton.
- `\b` ⇒ Backspace.
- `\c` ⇒ Unterdrückt die Ausgabe des Newline Zeichens.
- `\f` ⇒ form feed. Leerseite.
- `\n` ⇒ Newline.
- `\r` ⇒ Carriage Return.
- `\t` ⇒ Horizontaler Tabulator (acht Zeichen).
- `\v` ⇒ Vertikaler Tabulator (acht Leerzeilen).

Schleifen / Bedingungen mit Bash

If-Elif-Else-Schleifen

```
#!/bin/bash
# Grundrechenarten
clear;
echo -e "Welche Operation möchten Sie ausführen?\nBitte den ersten Buchstabe
n eingeben und mit Enter bestätigen.\n"
echo -e "[a]ddieren\n[s]ubtrahieren\n[m]ultiplizieren\n[d]ividieren\n"
read operator
clear;
echo -e "\nGeben Sie bitte den ersten Wert ein "
read a
clear;
echo -e "\nGeben Sie bitte den zweiten Wert ein "
read b

if [[ $operator = a ]]; then
         let c=a+b
         clear;
         echo $a + $b = $c

elif [[ $operator = s ]]; then
         let c=a-b
         clear;
         echo $a - $b = $c

elif [[ $operator = m ]]; then
         let c=a*b
         clear;
```

```
         echo $a "*" $b = $c

elif [[ $operator = d ]]; then
         let c=a/b
         clear;
         echo $a / $b = $c

else
clear;
echo -e "\nSie haben einen Ungültigen Operator angegeben!"
exit 1
fi
exit 0
```

Skript Ausgabe:

```
michael@home:~$ ./rechne

Welche Operation möchten Sie ausführen?
Bitte den ersten Buchstaben eingeben und mit Enter bestätigen.

[a]ddieren
[s]ubtrahieren
[m]ultiplizieren
[d]ividieren

m


Geben Sie bitte den ersten Wert ein
6


Geben Sie bitte den zweiten Wert ein
7


6 * 7 = 42
```

Case-Schleifen

Hier wird das obige Skript Beispiel mit einer „case“ Fallunterscheidung verwendet.

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

```
# Grundrechenarten
clear;
echo -e "Welche Operation möchten Sie ausführen?\nBitte den ersten Buchstabe
n eingeben und mit Enter bestätigen.\n"
echo -e "[a]ddieren\n[s]ubtrahieren\n[m]ultiplizieren\n[d]ividieren\n"
read operator
clear;
echo -e "\nGeben Sie bitte den ersten Wert ein "
read a
clear;
echo -e "\nGeben Sie bitte den zweiten Wert ein "
read b

case "$operator" in
         a)
         let c=a+b
         clear;
         echo $a + $b = $c;;

         s)
         let c=a-b
         clear;
         echo $a - $b = $c;;

         m)
         let c=a*b
         clear;
         echo $a "*" $b = $c;;

         d)
         let c=a/b
         clear;
         echo $a / $b = $c;;

         *)
         clear;
         echo -e "\nSie haben einen Ungültigen Operator angegeben!"
         exit 1;;
esac
exit 0
```

For-Schleifen

```
# !/bin/bash
# Sicherungskopien im Homeverzeichnis suchen.

for a in $(find /home -name *.old)
```

```
do
        echo -e "\n$a ist eine Sicherungskopie"
done

unset a
exit 0
```

Das „find“ Kommando in Zeile 4 kann auch in Backquotes geschrieben werden.

```
for a in `find /home -name *.old`
```

```
user@home:~$ ./bak
...
/home/user/.xsession-errors.old ist eine Sicherungskopie
...
```

Die Ausgabe des „find“ Befehls wird der Variablen (a) übergeben. Nach jeder Übergabe wird die „for“ Schleife durchlaufen. „echo -e“ aktiviert die Escape Sequenzen, dadurch kann mit „\n“ ein Zeilenumbruch durchgeführt werden.

While-Schleifen

```
#!/bin/bash
# Name: add_1.sh
# Beliebig viele Zahlen addieren.
summe=0

while test $# -gt 0
        do
        let summe=summe+$1
        shift
done

echo "Die Summe der Werte = $summe"
unset summe
exit 0
```

```
michael@home:~$ ./add_1.sh 29 21
Die Summe der Werte = 50
```

In dieser Schleife wird, sofern die Numerischen Argumente „greater than“ (-gt) [0] und mit der Variablen „summe“ addiert. „shift“ sorgt dafür, dass die Arumente beim auslesen un eins nach links verschoben werden. Aus $2 wird $1 usw.

Bash Rules

Big Rules

- Always double quote variables, including subshells. No naked $ signs
  - This rule gets you pretty far. Read http://mywiki.wooledge.org/Quotes for details
- All code goes in a function. Even if it's one function, `main`.
  - Unless a library script, you can do global script settings and call `main`. That's it.
  - Avoid global variables. Though when defining constants use `readonly`
- Always have a `main` function for runnable scripts, called with `main` or `main “$@”`
  - If script is also usable as library, call it using `"$0" == "$BASH_SOURCE" && main “$@”`
- Always use `local` when setting variables, unless there is reason to use `declare`
  - Exception being rare cases when you are intentionally setting a variable in an outer scope.
- Variable names should be lowercase unless exported to environment.
- Always use `set -eo pipefail`. Fail fast and be aware of exit codes.
  - Use `|| true` on programs that you intentionally let exit non-zero.
- Never use deprecated style. Most notably:
  - Define functions as `myfunc() { … }`, not `function myfunc { … }`
  - Always use [[instead of [or test
  - Never use backticks, use `$( … )`
  - See http://wiki.bash-hackers.org/scripting/obsolete for more
- Prefer absolute paths (leverage $PWD), always qualify relative paths with `./`.
- Always use `declare` and name variable arguments at the top of functions that are more than 2-lines
  - Example: `declare arg1=“$1” arg2=“$2”`
  - The exception is when defining variadic functions. See below.
- Use `mktemp` for temporary files, always cleanup with a `trap`.
- Warnings and errors should go to STDERR, anything parsable should go to STDOUT.
- Try to localize `shopt` usage and disable option when finished.

If you know what you're doing, you can bend or break some of these rules, but generally they will be right and be extremely helpful.

Best Practices and Tips

- Use Bash variable substitution if possible before awk/sed.
- Generally use double quotes unless it makes more sense to use single quotes.
- For simple conditionals, try using && and ||.
- Don't be afraid of `printf`, it's more powerful than `echo`.
- Put `then`, `do`, etc on same line, not newline.
- Skip `...` in your if-expression if you can test for exit code instead.
- Use `.sh` or `.bash` extension if file is meant to be included/sourced. Never on executable script.

- Put complex one-liners of `sed`, `perl`, etc in a standalone function with a descriptive name.
- Good idea to include `"$TRACE" && set -x`
- Design for simplicity and obvious usage.
    - Avoid option flags and parsing, try optional environment variables instead.
    - Use subcommands for necessary different “modes”.
- In large systems or for any CLI commands, add a description to functions.
    - Use `declare desc=“description”` at the top of functions, even above argument declaration.
    - This can be queried/extracted with a simple function using reflection.
- Be conscious of the need for portability. Bash to run in a container can make more assumptions than Bash made to run on multiple platforms.
- When expecting or exporting environment, consider namespacing variables when subshells may be involved.
- Use hard tabs. Heredocs ignore leading tabs, allowing better indentation.

Good References and Help:

- http://wiki.bash-hackers.org/scripting/start
    - Especially http://wiki.bash-hackers.org/scripting/newbie_traps
- http://tldp.org/LDP/abs/html/
- Tips for interactive Bash: http://samrowe.com/wordpress/advancing-in-the-bash-shell/
- For reference, Google's Bash styleguide

Weiteres

Substring extraction in bash

To extract substrings from variables in bash, we can use the following syntax

```
${variable_name:offset:length}
```

Caveats:

- If offset or length is negative, count from the end
- Negative values must be separated by a space away from the colon (:) to avoid being interpreted by bash as default value substitution. (See below for examples)

Get the file extension of a filename

```
$ filename=foobar.txt
$ echo ${filename: -4}
.txt
```

Get the file name without extension

```
$ filename=foobar.txt
```

```
$ echo ${filename:0: -4}
foobar
```

Last update: **2019/10/22 08:45**