TEC WORKSHOP

Link: https://www.tecchannel.de/a/shell-scripting-tipps-und-tricks-fuer-admins,2033904

Linux-Workshop
Shell Scripting - Tipps und Tricks für Admins

Datum: 31.08.2011
Autor(en): Thomas Steudten

Befehle auf der Kommandozeile sind praktisch, die Stärke zeigt die Shell aber bei der Ausführung von Shell-Skripts. Die Abarbeitung in Schleifen erlaubt wiederholende Tasks und durch bedingte Verzweigungen ist das Maß der Automation recht hoch. Wir zeigen Ihnen die entsprechenden Tipps und Tricks für erweiterte Scripts.

Administratoren von Linux-Systemen greifen gerne auf Skripts zurück. Die Arbeitserleichterung ist schnell sehr groß, wenn das Skript erstmal erstellt ist. Statt mühsam mit einzelnen Befehlen auf der Kommandozeile zu arbeiten, lassen sich mit den Scripts ganze Abläufe automatisieren. Backups, Monitoring, Dateioperationen und vieles mehr wird durch den simplen Aufruf der entsprechenden Scripts erledigt.

In insgesamt drei Artikeln des Shell Scriptings beschäftigen wir uns mit alltäglichen Machenschaften des Systemadministrators in Interaktion mit der Bourne-Shell ("sh") und dessen Nachfolger Bourne-Again-Shell ("bash").

Im ersten Linux-Workshop **Shell Scripting - Abläufe automatisieren**[1] geht es um die grundlegende Funktionalität der Shell. Der zweite Teil **Shell Scripting im Admin-Alltag**[2] befasst sich mit alltäglichen Aufgaben des Admins. Im dritten und letzten Teil zeigen wir Ihnen weitere Script-Beispiele, die für Linux-Administratoren nützlich sind. Beispielsweise erläutern wir hier die Erstellung von Schleifen und erklären das Script-Debugging.

Artikelserie Shell-Scripting unter Linux

Teil 1: **Shell Scripting - Abläufe automatisieren**[3]
Teil 2: **Shell Scripting im Admin-Alltag**[4]
Teil 3: Shell Scripting - Tipps und Tricks für Admins

1. Script-Einstieg

Ein Script lässt sich unter Linux sehr einfach erstellen. Wir beginnen mit einem simplen Beispiel:

```
date
whoami
uptime
```

Ausgeführt sieht es so aus:

```
> tom.sh
tom.sh: command not found
> ls -l tom.sh
-rw-r--r-- 1 thomas thomas 19 2010-12-07 12:17 tom.sh
```

Wir haben unsere Kommandos einfach in eine Datei gepackt - woher sollte die Shell wissen, dass dort ausführbare Befehle enthalten sind?

```
> bash tom.sh
Tue Dec 7 12:20:17 CET 2010
thomas
12:20:17 up 6 days, 3:17, 8 users, load average: 0.34, 0.37, 0.44
```

Nun haben wir der bash explizit mitgeteilt, unsere Datei als Skript anzusehen und die Befehle darin zeilenweise auszuführen.

Die Bash kann uns auch anzeigen, welche Befehle als nächstes im Skript ausgeführt werden. Dazu dient die Option "-v" für verbose oder "-x" für debug:

```
> bash -x tom.sh
+ date
Tue Dec 7 12:22:56 CET 2010
+ whoami
thomas
+ uptime
12:22:56 up 6 days, 3:19, 8 users, load average: 0.43, 0.36, 0.42
```

Wir teilen der Shell mit, dass unsere Datei Befehle enthält und somit für die Shell ein ausführbares Skript darstellt. Zusätzlich setzen wir den Suchpfad für Befehle:

```
> chmod 755 tom.sh; export PATH=.:$PATH
> tom.sh
Tue Dec 7 12:29:10 CET 2010
thomas
12:29:10 up 6 days, 3:26, 8 users, load average: 0.27, 0.30, 0.36
```

2. Schleifen

Befehle können in Schleifen verpackt werden und so wiederholt zur Ausführung gebracht werden.

Ein einfaches Beispiel dafür:

```
> while true; do date; sleep 1; done
> until false; do date; sleep 1; done
```

Diese Zeilen schreiben jede Sekunde die Ausgabe von "date" auf stdout.

Weitere Beispiele:

```
> for i in 1 2 3; do echo "$i: `date`"; done
> for i in `seq 10`; do echo $i; done
> for arch in *.gz; do gzip -v $arch; done
```

Die Shell-Variable IFS (Internal Field Separator) kann durch umsetzen genutzt werden, um Felder durch andere Trennzeichen zu separieren.

ifs_script:

```
for ff in $(date); do echo $ff; done
set -- $(date)
OLDIFS=$IFS; IFS=:
for ff in $4; do echo $ff; done
IFS=$OLDIFS
```

```
> bash ifs_script
Wed
Dec
8
11:01:34 <= enspricht $4
CET
2010
11
01
34
```

job.sh:

```
#!/bin/bash
myjob()
{
echo $1
..
}
for arg; do
myjob $arg
done
```

> job.sh test haus ball

3. Housekeeping

Mittels "trap" können wir in unserem Skript auf Signale, wie "Abbruch durch Benutzer CTRL-C" reagieren. Für ein "sauberes" Beenden und Löschen von temporär angelegten Dateien bietet die bash die Signalnummer 0, das heißt Skript-Beendigung.

Nehmen wir exemplarisch das nachfolgende Skript, welches eine Liste aller Dateien in /bin in ein temporäres File generiert und aus dieser Liste nur die erste Datei nach "https://www.computerwoche.de/tmp/<name>.bak" kopiert. Bei normaler oder signalbedingter Beendigung wird sowohl die Liste, als auch die kopierte Datei wieder gelöscht und somit der wünschenswerte Status-Quo vor der Skriptausführung wieder hergestellt. Bei fehlerfreier Ausführung erhalten Sie als Exit-Wert 0. Sollte das Skript durch ein Signal beendet worden sein, so wird der Exit-Wert um 128 inkrementiert.

```
#!/bin/bash
TMP=$(mktemp)
TMPLIST=""
clean()
{
/bin/rm -f $TMP $TMPLIST && echo "$TMP $TMPLIST removed"
}
trap 'clean; echo exit reached.' 0
### MAIN
echo "TMP file created as $TMP .."
find /bin -type f > $TMP
while read line; do
mybase=$(basename $line).bak
cp $line /tmp/$mybase && { echo "cp $line to /tmp/$mybase"; TMPLIST=/tmp/$mybase; }
break
done < $TMP
echo "our pid is $$"
sleep 10
# exit value from last commands..
```

Fehlerfreie Ausführung:

```
> clean.sh; echo $?
clean.sh
TMP file created as /home/thomas/tmp/tmp.fbu0DfjtX7 ..
cp /bin/setfont to /tmp/setfont.bak
our pid is 21883
/home/thomas/tmp/tmp.fbu0DfjtX7 /tmp/setfont.bak removed
exit reached.
0
```

Senden wir ein "kill <pid>" an das Skript, so zeigt der Exit-Wert 128+15=143 an:

```
..
exit reached.
Terminated <= Hinweis auf Signal 15 (TERM)
143
```

So implementierte Shell-Skripts hinterlassen keine unnötigen Dateileichen im Dateisystem.

4. Script-Debugging

Wenn das Skript nicht wie erwartet abläuft, hilft es, sich der Debugging-Möglichkeiten der Bash zu bedienen:

```
#!/bin/bash
mytrap()
{
echo "Trap catched..$?"
}
trap 'mytrap' 0
trap 'echo debug[$LINENO]: $BASH_COMMAND' DEBUG
#set -e
echo starting.. $$
false
sleep 10
echo finish.
```

Ausführen des Skripts "run.sh":

```
> ./run.sh
debug[17]: echo starting.. $$
starting.. 3242
debug[18]: false
debug[19]: sleep 10
debug[20]: echo finish.
finish.
debug[1]: echo finish.
Trap catched..0
```

Das DEBUG-Signal beim trap-Handler ruft den Handler vor jeder Befehlsausführung auf und der Handler gibt uns die Zeilennummer im Skript beziehungsweise der Funktion und den auszuführenden Befehl dazu aus.

Setzen wir die Option "-e", dann wird das Skript beendet, sobald ein Befehl darin einen Exit-Wert > 0 liefert:

```
./run.sh
debug[15]: set -e
debug[17]: echo starting.. $$
starting.. 3378
debug[18]: false
debug[1]: false
Trap catched..1
```

5. Here-Dokument

Wie bereits angesprochen, liest jeder Befehl über stdin seine Eingaben. Damit dies automatisiert funktioniert, kann man mittels "echo .. | <cmd>" dem nachfolgenden Befehl die Eingaben liefern.

```
> bc
..
ibase=10
obase=16
10
A <= Ausgabe 0Ah
quit
```

Automatisiert:

```
> echo -e "ibase=10\nobase=16\n10\nquit\n" | bc
A
```

Im Script gibt es dafür eine elegantere Lösung und wir sprechen von einem Here-Dokument für diesen Aufbau. Die Shell liest alle Zeilen zwischen zwei Markern und sendet diese (bearbeitet) über stdin an den Befehl. So lassen sich auch komplexe Abläufe automatisieren.

myscript:

```
[..]
bc << _EOF_
obase=16
ibase=10
10
quit
_EOF_
[..]
```

> bash myscript

A

Recht neu ist die Form des "Here Strings", die "word" auf stdin an den Befehl weiterreicht:

> cmd<<<word

6. Command-Wrapper

Wir können den Aufruf eines Tools im Dateisystem verfolgen, indem wir ein Wrapperskript implementieren und jeden Aufruf in eine Datei schreiben. Mittels "exec" überlagern wir die aktuelle Shell mit diesem Kommando, da anschließend keine weiteren Befehle mehr folgen. Anderenfalls wäre der Tool-Aufruf ein Kindprozess der Shell, die nur auf die Beendigung wartet.

Wir ersetzen "https://www.computerwoche.de/bin/tar" durch unser Skript. Das Vorgehen ist wie folgt:

- Umbennenen von /bin/tar nach /bin/.tar.bin
- Unser Wrapperskript "tar" nach /bin kopieren und die Rechte anpassen

Inhalt des Wrappers:

```
#!/bin/sh
LOG=/tmp/tar.log
echo "tar called from `id` on `date` `tty` with args $*" >> $LOG
exec /bin/.tar.bin $*
```

7. Zusätzliche Tipps

Immer nützlich sind die "help <cmd>" Seiten der Bash, die auch gleich die reservierten Befehle der Shell listet.

"type <cmd>" liefert eine Antwort auf "Was ist <cmd>?",

"which <cmd>" auf "Wo im Suchpfad (PATH) wird <cmd> gefunden?"

8. Fazit

Die Shells, insbesondere die verbreitete Bourne-Again Shell (bash), bieten im Terminal ein mächtiges Werkzeug für die Systemadministration.

Wir haben in den drei Workshops kurz und knapp einige Grundkenntnisse vermittelt und einige Stolpersteine aufgezeigt, ohne zu sehr in die Tiefe zu gehen. Für spezialisierte Aufgaben, wie Patternsuche und -Bearbeitung ist Perl oder eine Hochsprache die in der Ausführung schnellere Methode. (cvi)

Links im Artikel:

[1] **https://www.tecchannel.de/server/linux/2032466/workshop_shell_scripting_fuer_linux/**
[2] **https://www.tecchannel.de/server/linux/2033265/linux_workshop_shell_scripting_fuer_den_admin_alltag/**
[3] **https://www.tecchannel.de/server/linux/2032466/workshop_shell_scripting_fuer_linux/**
[4] **https://www.tecchannel.de/server/linux/2033265/linux_workshop_shell_scripting_fuer_den_admin_alltag/**
