Was ist Shell?

Autor: Joel Garske (*jg-soft@web.de*)
Layout: Alexander Fischer (*Selflinux@tbanus.org*)

In diesem Kapitel geht es um eine der wichtigsten Schnittstellen zwischen Betriebssystem und Nutzer: Um die *Shell*. Der Leser soll hier mit den grundlegendsten Begriffen und Funktionsweisen vertraut gemacht werden. Es wird eine Einführung bis hin zu einigen praktischen Tips gegeben.

Inhaltsverzeichnis

1 Allgemeines zum Thema Shell

1.1 Was ist eine Shell?

Eine *Shell* (engl. Muschel) ist ein Kommandozeileninterpreter. Sie stellt einen elementaren Bestandteil eines Betriebssystems dar. Das, was man von einer *Shell* sieht, ist das Prompt. In Linux sieht es meist folgendermaßen aus (je nach System und Shell kann es verschiedene Formen des Layouts geben):

```
benutzername@rechner:pfad$
```

Hierbei steht benutzername für den gerade angemeldeten Benutzer. rechner ist der Name des Rechners (meist der Domain Name des Rechners). pfad ist der Pfad, in dem sich der Benutzer gerade befindet.
Steht anstelle von pfad ein `~`, so befindet sich der Benutzer in seinem Heimatverzeichnis. In dieses kann übrigens auch mit `cd` gewechselt werden.
Meist identifiziert das `$` Zeichen den gerade angemeldeten Benutzer als normalen Benutzer, ohne Administratorrechte. Ist der Administrator angemeldet (Benutzer root), so wird standardmäßig anstelle des `$` ein `#` angezeigt. Diese Einstellung der Zeichen wird in den meisten Fällen bei der Installation der Shell automatisch gemacht. Es ist aber ohne Probleme möglich, das in der Konfigurationdatei der Shell zu verändern und es kann auch vorkommen, das eine Distribution andere Zeichen einsetzt.

Das Prompt nimmt die Befehle des Benutzers entgegen und führt sie aus (z.B. einen Verzeichniswechsel oder die Ausführung eines Programmes) und erscheint nach Beendigung des Befehls wieder, um Befehle entgegen zu nehmen.

Unter Linux gibt es mehrere *Shells*. Die gängigste und meist verwendete *Shell* ist die *bash*. Des Weiteren existieren *bsh*, *ash*, *zsh* und andere...

Die *Shell* nimmt Befehle des Benutzers entgegen, um deren Ausführung zu veranlassen. Obwohl sie nicht direkt zum Betriebssystem gehört, sind beide eng verknüpft. Die *Shell* und die Arbeit mit der *Shell* spielt unter Linux eine große Rolle, da es durch sie ermöglicht wird, praktisch alle Systemvorgänge mit ihr kontrollieren und verwalten zu können.

Notiz für Windows-Umsteiger: Die *Shell* ist das, was Windows- oder DOS-Benutzer als `command.com` kennen (z.B. mit `C:\>` als Prompt).

2 Praxistips

2.1 Kleine Praxis-FAQ

Q: Wie finde ich heraus, welche *Shell* ich gerade benutze?
A: Das zeigt die Eingabe von "`echo $0`" (ohne die ")

Q: Kann ich meinen Prompt auch ändern?
A: Ja, durch die Eingabe von `export PS1="Mein Prompt: "`

2.2 Die Befehlshistory

Oftmals ist es nötig, bestimmte Systemvorgänge zu vereinfachen, beziehungsweise Vorgänge zu automatisieren, um die Arbeit mit dem System zu vereinfachen. Welcher DOS-User war z.B. nicht mit den Nerven am Ende, als es beim Kopieren eines weit entfernten Ordners um die Eingabe des endlos langen Pfades ging... Hier griff man schnell zu Programmen, die einem diese Arbeit abnehmen sollten. Das Programm, das die DOS-Benutzer unter dem Namen *doskey* kennen, erleichterte auch nur geringfügig die Arbeit, denn es speicherte nur bereits eingegebene Befehle (und auch nur bis zum Neustart). Diese Befehlshistory ist in den meisten Linux-*Shells* bereits integriert. Man kann auf sie sogar nach einem Neustart noch zugreifen.

2.3 Der Completer

Die History hilft einem allerdings auch nicht bei der Eingabe eines Pfades... Nehmen wir an, der Administrator des Systems "Mond" möchte gerne die Datei

`/home/milchstrasse/Dhc412/PldE/text.dat` nach

`/usr/share/planetare_datenbank/` kopieren.

Gibt er die Pfade manuell ein, dauert dies ggf. sehr lange. Hier greift die Completer-Funktion der meisten (modernen) Linux-*Shells*. Diese vervollständigt durch Druck auf die TAB- (Tabulator-) Taste den Pfad nach bestem Wissen. So reicht die Eingabe von

```
user@linux / # cp /ho TAB mil TAB Dh TAB Pl TAB te TAB /usr/sh TAB pl TAB
ENTER
```

im Normalfall aus, um die o. g. Aktion auszuführen. TAB steht hier für das Drücken der Tabulator-Taste. Die Leerzeichen müssen (und dürfen) nicht eingegeben werden. Was hier im Beispiel kompliziert aussieht, ist meist von sehr hohem Nutzen in der täglichen Arbeit des Linux-Anwenders.

Machen wir das Beispiel etwas komplizierter... Nehmen wir an, im Ordner

`/usr/share/`

existiert zusätzlich zum Verzeichnis `planetare_datenbank/` noch eine Datei mit Namen `planeten.dat`.

Woher soll der Completer nun wissen, dass wir den Ordner `planetare_datenbank` meinen? Ebensogut könnten wir die Datei `planeten.dat` überschreiben wollen. Um sich beim Benutzer rückzuversichern, gibt der Completer einen Beep aus (Piepton) und vervollständigt nur so weit, wie beide Datei/Ordnernamen übereinstimmen (in diesem Fall bis "`planet`"). Ein zweimaliges Drücken der TAB-Taste gibt als Hilfestellung alle Möglichkeiten auf dem Bildschirm aus, ohne die Eingabe in der Kommandozeile abzubrechen. Nun kann man durch die Eingabe des Buchstabens `a` und drücken der TAB-Taste sagen, dass man den Ordner `planetare_datenbank` meint.

(Ordnernamen werden generell mit abschliessendem / vervollständigt).

Mit dieser Funktion lassen sich jedoch nicht nur Pfade vervollständigen, sondern es lässt sich auch feststellen, ob ein bestimmter Befehl existiert. Als Beispiel soll der Befehl `cdrecord` dienen... Je nach installierten Programmen kann die Ausgabe etwas variieren.

Vergass man unter DOS einen Befehl, so war nur durch eine Suche oder "blindes" Testen die Feststellung des Befehlsnamens möglich. Unser Beispielbenutzer hat vergessen, wie dieses verflixte Brennprogramm heisst... Mutige Benutzer könnten ohne vorherige Eingabe am Prompt `TAB + TAB` drücken. Dadurch werden (nach kurzer Arbeit und meist auch nach Nachfrage) alle in den für den Nutzer sichtbaren Pfade vorhandenen Befehle ausgegeben (alphabetisch sortiert). Wem das zu viel war, der gebe den Anfang des Befehls ein.

Unser Beispielbenutzer konnte in den Unmengen Befehlen (unter Umständen über 1000) seinen gewünschten Befehl nicht finden. Er erinnert sich jedoch schwach an das Programm cd.... cd.... wie war es doch gleich? Die Eingabe von `cd` und anschliessendes `TAB + TAB` bringen wieder einige Befehle zum Vorschein, die mit `cd` beginnen...

Noch immer nicht? Gut. er findet das Programm wider Erwarten immernoch nicht und versucht es mit `cdr TAB + TAB`. In den wenigen Befehlen hat er schnell das Programm `cdrecord` gefunden.

2.4 Konsolenpuffer / Output History

Sicherlich war man auch unter DOS genervt, wenn mal ein Programm mehr Ausgabe erzeugte, als auf einem Bildschirm zu fassen war. Hier löste man das Problem mit einer *pipe* (| - engl. für Rohr) zu einem Befehl, der die Ausgabe anpasste. Auch das ist unter Linux möglich (dazu später mehr), jedoch nicht immer nötig. Die Arbeit erledigt der sog. Konsolenpuffer, der einige Bildschirme der letzten Ausgabe zwischenspeichert. ACHTUNG! Dieser Puffer wird jedoch gelöscht, sobald man die Konsole wechselt!!! (Dies gilt nicht für X-Terminals). Der Speicher lässt sich durch drücken der Tastenkombinationen `SHIFT + BILD-AUF` und `SHIFT + BILD-AB` durchscrollen.

2.5 Ausgabeumleitung

Wie oben schon angedeutet, kann es mitunter sinnvoll sein, die Ausgabe eines Programms umzuleiten. Dies kann z. B. zu einem anderen Befehl, der diese weiterverarbeitet, oder in eine Datei (oder ein Gerät) erfolgen. Diese zwei verschiedenen Formen der Umleitung sollen in diesem Kapitel erklärt werden.

2.5.1 Umleiten in eine Datei

Ein typisches Problem: Wir haben ein Verzeichnis, dessen Inhalt katalogisiert werden soll. Um Rechte und Grösse der einzelnen Dateien festzuhalten, eignet sich das Tool `ls` (welches den Befehl `dir` aus DOS ersetzt) gut, wenn man es mit dem Parameter `-l` kombiniert. (Mitunter ist dieser Schritt nicht notwendig, weil je nach Distribution dieser Parameter als Standard eingestellt ist). Leider (oder in den meisten Fällen glücklicherweise) gibt `ls -l` seine Ausgabe auf dem Bildschirm aus. Wie bekommen wir die Ausgabe in eine Datei? Der Operator `>` hilft uns weiter. Er sorgt dafür, dass die Ausgabe des Programms nicht auf den Bildschirm, sondern auf die angegebene Datei erfolgt.

Ein Beispiel:

```
user@linux / # ls -l > ~/listing.txt
```

gibt den Verzeichnisinhalt des aktuellen Verzeichnisses in die Datei `listing.txt` im Heimatverzeichnis des Benutzers.

Die Umleitung kann auch auf Geräte erfolgen, was jedoch nur bedingt Sinn macht. Soll z. B. die Ausgabe eines Programms direkt (nicht als Datei) auf Diskette geschrieben werden (jeglicher bisheriger Inhalt der Diskette geht verloren!), so wird die Ausgabe an das Diskettenlaufwerk (im Regelfall `/dev/fd0`) weitergeleitet.

Ein sinnloses Beispiel:

```
user@linux / # ls -l > /dev/fd0
```

gibt die Liste der Dateien im aktuellen Verzeichnis an das Diskettenlaufwerk... Dieses Beispiel mach wenig Sinn, da diese Liste praktisch nicht mehr auslesbar ist und den Inhalt der Diskette zerstört.

2.5.2 Umleitung an Befehle (via Pipe)

Wie bereits oben beschrieben kann es nötig sein, die Ausgabe einer Datei umzuleiten, um sie besser verarbeiten zu können. Die häufigste Anwendung ist die Weiterleitung an die Programme `less` oder `more`. Diese beiden Programme sorgen dafür, dass die Ausgabe eines Programms auch dann vollständig zu lesen ist, wenn sie sowohl grösser als der Bildschirm, als auch grösser als der Ausgabepuffer ist. Dies kommt besonders häufig beim Listing langer Verzeichnisse oder ähnlichen Vorgängen vor. Ein ausführliches Listing des Geräteordners `/dev` zeigt dies anschaulich, da dieser Ordner mitunter weit über 1500 "Dateien" enthält.

Folgendes Beispiel soll die Funktionsweise der *Pipe* verdeutlichen, macht jedoch wiederum wenig Sinn:

```
user@linux / # ls -l /dev | less
```

Dieser Befehl gibt eine Liste der Geräte aus, die sich mit den Bildlauf-Tasten `BILD-AUF` und `BILD-AB` beliebig scrollen lässt. Die Ausgabe lässt sich mit der Taste `q` beenden.

Ein weiteres Anwendungsgebiet der *Pipe* ist die Suche. So lässt sich z. B. über den Befehl `grep` mit einem Befehl eine Reihe von Dateien in einem Verzeichnis durchsuchen:

```
user@linux / # cat * | grep Blubberbläschen
```

Diese Befehlsreihe durchsucht alle Dateien im Verzeichnis nach dem Begriff Blubberbläschen. Anzumerken ist hier, dass `grep` die Suche unter Beachtung von Gross- und Kleinschreibung durchführt (was mit mit dem Schalter `-i` verhindern kann). Der Befehl `cat` gibt in diesem Fall den Inhalt aller Dateien nacheinander an die *Pipe* weiter (normalerweise auf den Bildschirm) die zum Befehl `grep` führt. Dieser durchsucht jede einzelne Zeile nach dem Wort Blubberbläschen und gibt die Zeilen aus, in denen das Wort vorkommt.

2.6 Der Ersetzer sed

Eins der nützlichsten Programme in einem Linux System ist der Ersetzer *sed*. Er stellt ein flexibles Tool zum Ersetzen von Ausdrücken in Dateien dar. Ein Praktisches Beispiel: Die Firma XY hat ihre Homepage vom Server *server1.xy.de* auf *server2.xy.de* verlegt. In den HTML-Dateien müssen nun die vielen Links geändert werden. Eine Möglichkeit wäre, die Dateien auf einer grafischen Oberfläche in den Lieblingseditoren und dessen "Ersetzen"-Funktion zu verwenden. Diese durchaus zweckdienliche Methode für "Mausverliebte" hat jedoch eine schlagkräftige Alternative. Ihr Name ist *sed*. Dieses einfach zu bedienende Programm erledigt die Arbeit fast von alleine. In unserem Beispiel müssen also alle Ausdrücke *server1* in *server2* umgeschrieben werden. Dies erledigt folgender Befehl:

```
user@linux / # sed 's/server1/server2/g' dateiname.html
```

Eine kurze Erklärung: `sed` ist der Befehlsaufruf. Auf ihn folgt ein Text, der in `''` gefasst ist. Das ist das Ersetzungsscript. Dieses Script enthält die Informationen, was wie ersetzt werden soll... Entschlüsseln wir es einmal:

```
s/server1/server2/g
```

Das erste `s` leitet die Ersetzung ein. Der darauffolgende `/` deutet auf den Beginn des zu ersetzenden Wortes hin. Nun kommt das zu ersetzende Wort, gefolgt von einem weiteren `/`, der das Wort angibt, durch das ersetzt werden soll (hier *server2*). Der dritte `/` beendet das zu einzufügende Wort. Im allgemeinen ersetzt *sed* immer nur den ersten Ausdruck, den es in einer Zeile findet. Das `g` am Ende sorgt dafür, dass auch folgende Ausdrücke, die dem Suchmuster entsprechen ersetzt werden. Es steht für *global*.

Wer diesen Befehl einmal getestet hat, wird enttäuscht feststellen, dass die Ausgabe der geänderten Datei auf der Konsole erfolgt. Leider können wir nicht die Ausgabe von *sed* nach dateiname.html umleiten, da die Datei ansonsten durch den Aufruf einer Endlosschleife vernichtet würde. Wir können jedoch, wie in "*Umleiten in eine Datei*" gelernt, die Ausgabe in eine andere Datei umleiten. Dann sieht der Befehl so aus:

```
user@linux / # sed 's/server1/server2/g' dateiname.html > dateiname2.html
```

Nun liegt die aktualisierte Fassung von *dateiname.html* in *dateiname2.html*. Nach einer Überprüfung des Ergebnisses kann die neue Version mittels

```
user@linux / # mv dateiname2.html dateiname.html
```

die alte Version ersetzten. Die alte Version geht hierbei jedoch verloren. Vorsicht ist angesagt. Besondere Vorsicht sollte geboten sein, wenn sich Sonderzeichen oder Operatoren in Quelle oder Ziel der Ersetzung befinden. Beispiele hierfür sind folgende Zeichen:

! " $ % & / ( ) = ? \ . ;

Diese müssen im *sed-Script* mit vorangestelltem `\` verwendet werden, also zum Beispiel `\\` für einen Backslash oder `\/` für einen Slash. Man sollte hier das Endergebnis besonders sorgfältig prüfen!
Ratsam ist auf jeden Fall das Lesen der Manual-Page zu sed (`man sed`).