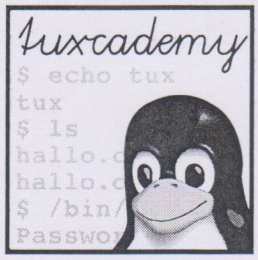

2

Shellskripte

Inhalt

Lernziele

- Einsatzgebiete und grundlegende Syntax von Shellskripten kennen
- Shellskripte aufrufen können
- `#!`-Zeilen verstehen und formulieren können

Vorkenntnisse

- Vertrautheit mit der Kommandooberfläche von Linux
- Umgang mit Dateien und einem Texteditor
- Shell-Vorkenntnisse (etwa aus Kapitel 1)

2.1 Einleitung

Vorteil

Der unschlagbare Vorteil von Shellskripten ist: Wenn Sie mit der Shell umgehen können, dann können Sie auch programmieren! Shellskripte bieten sich immer dann an, wenn es darum geht, eine Aufgabe zu automatisieren, die Sie auch interaktiv »von Hand« hätten erledigen können. Umgekehrt können Sie die Kontrollstrukturen für Shellskripte auch jederzeit auf der Kommandozeile benutzen. Dies vereinfacht nicht nur das Testen von Konstruktionen immens, es macht oft ein Skript komplett überflüssig.

Einsatzgebiete

Die Einsatzgebiete von Shellskripten sind vielfältig. Überall, wo dauernd die gleichen Kommandos eingegeben werden müssten, lohnt es sich, ein Shellskript zu schreiben (etwa bei der regelmäßigen Suche nach Dateien mit bestimmten Eigenschaften). Meist werden Shellskripte aber verwendet, um komplexe Aufgaben zu vereinfachen (zum Beispiel automatisches Starten von Netzwerkdiensten). In der Regel geht es dabei nicht darum, ausgefeilte bzw. schöne Programme zu schreiben, sondern die eigene Arbeit zu vereinfachen. Was nicht heißt, dass Sie einen schlechten Programmierstil entwickeln sollten – Kommentare und Dokumentation sind spätestens dann sinnvoll, wenn auch andere Personen Ihre Skripte nutzen und verstehen sollen.

Nachteile

Allerdings sollten Sie Shellskripte auch nicht überstrapazieren: Überall da, wo komplexere Datenstrukturen benötigt werden oder es auf Effizienz oder Sicherheit ankommt, sind Sie mit echten Programmiersprachen meist besser beraten. Neben den »klassischen« Sprachen wie C, C++ und Java kommen hier auch die modernen »Skriptsprachen« Tcl, Perl oder Python in Frage.

2.2 Aufruf von Shellskripten

Ein Shellskript können Sie auf verschiedene Art und Weise starten. Am einfachsten übergeben Sie seinen Namen einer Shell als Aufrufparameter:

```
$ cat skript.sh
echo Hallo Welt
$ bash skript.sh
Hallo Welt
```

Das ist natürlich unbefriedigend, da Benutzer Ihres Skripts zum einen wissen müssen, *dass* es sich um ein Shellskript handelt (und nicht etwa ein ausführbares Maschinenprogramm oder ein Skript für die Programmiersprache Perl), und zum anderen, dass es mit der Bash ausgeführt werden muss. Es wäre schöner, wenn der Aufruf Ihres Skripts aussähe wie der jedes anderen Programms. Dazu müssen Sie das Skript mit `chmod` für ausführbar erklären:

```
$ chmod u+x skript.sh
```

Anschließend können Sie es mit

```
$ ./skript.sh
```

direkt starten.

Shellskript-Dateien müssen nicht nur ausführbar sein, sondern auch lesbar (im Sinne des r-Rechts). Bei Maschinenprogrammen, die im Binärcode vorliegen, genügt Ausführbarkeit.

Wollen Sie auf das unschöne »./« verzichten, so müssen Sie dafür sorgen, dass die Shell Ihr Skript finden kann. Zum Beispiel können Sie ».« – das aktuelle Arbeitsverzeichnis – in Ihren Suchpfad für Kommandos (Umgebungsvariable `PATH`) aufnehmen. Das ist aber einerseits aus Sicherheitsgründen keine gute Idee (erst

recht nicht für root) und andererseits unpraktisch, weil es ja sein könnte, dass Sie Ihr Skript mal aus einem anderen Verzeichnis heraus aufrufen wollen. Am besten legen Sie sich für oft gebrauchte Shellskripte ein Verzeichnis $HOME/bin an und nehmen das explizit in Ihren PATH auf.

Die dritte Methode, ein Shellskript aufzurufen, besteht darin, das Skript statt in einem Kindprozess in der aktuellen Shell auszuführen. Dazu können Sie das Kommando »source« bzw. seine Kurzschreibweise ».« verwenden:

```
$ source skript.sh
Hallo Welt
$ . skript.sh
Hallo Welt
```

(Bitte beachten Sie, dass bei der Kurzschreibweise hinter dem Punkt ein Leerzeichen folgen muss.) Einem so gestarteten Skript steht der komplette Kontext der aktuellen Shell zur Verfügung. Während ein in einem Kindprozess gestartetes Skript zum Beispiel das aktuelle Verzeichnis oder die *umask* Ihrer interaktiven Shell nicht direkt verändern kann, können Sie das über ein mit source gestartetes Skript durchaus. (Mit anderen Worten: Ein mit source aufgerufenes Skript wird so ausgeführt, als würden Sie die Zeilen in der Datei direkt in Ihre interaktive Shell eintippen.)

Wichtig ist diese Methode beispielsweise dann, wenn Sie aus einem Shellskript heraus auf Shellfunktionen, -variable oder Aliase zugreifen wollen, die in einem anderen Shellskript definiert werden – etwa im Sinne einer »Bibliothek«. Würde jenes Shellskript wie üblich in einem Kindprozess ausgeführt, dann hätten Sie nichts von den Definitionen darin!

Namensgebung

Bei der Namensgebung Ihrer Shellskripte sollten Sie aussagekräftige Bezeichnungen wählen. Es bietet sich zusätzlich an, Datei-Endungen wie ».sh« oder ».bash« zu verwenden, damit man auf den ersten Blick erkennen kann, dass es sich um Shellskripte handelt. Natürlich können Sie auch auf die Endung verzichten. Insbesondere in Experimentierphasen ersparen Sie sich durch die Endung aber viel Ärger, denn so naheliegende Namen wie test oder script werden schon von Systemkommandos benutzt.

Wie schon erwähnt, sollten Sie die Endung ».sh« für Skripte reservieren, die auch mit der Bourne-Shell oder Kompatiblen laufen, also keine speziellen Bash-Eigenschaften voraussetzen.

Übungen

2.1 [!2] Erstellen Sie eine Textdatei namens meinskript.sh, das zum Beispiel mit »echo« eine Meldung ausgibt, und machen Sie diese Datei ausführbar. Überzeugen Sie sich, dass die drei Aufrufmöglichkeiten

```
$ bash meinskript.sh
$ ./meinskript.sh
$ source meinskript.sh
```

im Sinne der obigen Beschreibung funktionieren.

2.2 [!1] Mit welcher Methode liest die Login-Shell die Dateien /etc/profile und $HOME/.bash_profile – Kindprozess oder »source«?

2.3 [2] Ein Benutzer kommt mit der folgenden Beschwerde zu Ihnen: »Ich habe ein Shellskript geschrieben und wenn ich es aufrufe, passiert gar nichts. Dabei gebe ich als allererstes eine Meldung auf die Standardfehlerausgabe aus! Linux ist Mist!« Bei einer peinlichen Befragung Ihrerseits kommt heraus, dass der Benutzer sein Skript test genannt hat. Was geht hier vor?

2.4 [3] (Trickreich.) Wie würden Sie dafür sorgen, dass ein Shellskript in einem Kindprozess ausgeführt wird und trotzdem das aktuelle Verzeichnis der aufrufenden Shell ändern kann?

2.3 Aufbau von Shellskripten

Shellskripte sind eigentlich nur Folgen von Shellkommandos, die in einer Textdatei abgespeichert sind. Die Shell kann ihre Eingabe wahlweise von der Tastatur (Standardeingabe) oder aus einer anderen Quelle, etwa einer Shellskript-Datei lesen – bei der Ausführung der Kommandos besteht da eigentlich kein Unterschied.

Zeilenumbrüche

Zeilenumbrüche dienen zum Beispiel als Kommandoseparatoren, ganz wie auf der »echten« Kommandozeile.

Einige Lesbarkeits-Tipps: Was Sie auf der Kommandozeile der Bequemlichkeit halber in der Form

```
⟨Kommando1⟩ ; ⟨Kommando2⟩
```

eingeben würden, sollten Sie im Skript der Übersichtlichkeit wegen untereinander schreiben:

```
⟨Kommando1⟩
⟨Kommando2⟩
```

Für die Ausführung bedeutet das keinen Unterschied.

Die Lesbarkeit eines Skripts können Sie ferner durch gezielte Verwendung von Leerzeilen steigern, die von den Shells ignoriert werden – ganz wie auf der Kommandozeile. Ebenfalls ignoriert wird alles, was einem Rautenzeichen (»#«) folgt.

Kommentare

Damit können Sie Ihre Skripte kommentieren:

```
# Zuerst kommt Kommando1
⟨Kommando1⟩
⟨Kommando2⟩ # Das ist Kommando2
```

Kommentarblock

Größere Skripte sollten mit einem Kommentarblock anfangen, der beschreibt, wie das Skript heißt, was es tun soll und wie es das macht, wie es aufgerufen werden muss und so weiter. Auch der Name des Autors und eine Versionsgeschichte könnten dort erscheinen.

Mit `chmod` als ausführbar gekennzeichnete Textdateien werden vom System als Skripte für die Shell `/bin/sh` angesehen – auf Linux-Systemen ist das oft (aber nicht immer) ein Link auf die Bash. Um sicherzugehen, sollten Shellskripte daher mit einer Zeile beginnen, die die Zeichen »`#!`« und den Namen der gewünschten Shell als absoluten Pfad enthält. Zum Beispiel:

```
$ cat skript
#!/bin/bash
<<<<<
```

Dann wird zur Ausführung des Skripts die benannte Shell herangezogen, indem der Dateiname des Skripts als Parameter an den angegebenen Shellnamen angehängt wird. Unter dem Strich startet der Linux-Kern in unserem Beispiel also das Kommando

```
/bin/bash skript
```

Das Programm, das das Skript ausführt, muss keine Shell im engeren Sinne sein – jedes Programm, das im Binärcode vorliegt, kommt in Frage. Auf diese Weise können Sie zum Beispiel »`awk`-Skripte« schreiben (Kapitel 7).

Die »#!«-Zeile darf bei Linux maximal 127 Zeichen lang sein und außer dem Programmnamen auch Parameter enthalten; der Skriptname wird in jedem Fall ans Ende angehängt. Beachten Sie, dass proprietäre Unix-Systeme oft wesentlich engere Grenzen setzen: Zum Beispiel ist nur eine Gesamtlänge von 32 Zeichen mit höchstens einem Optionsargument erlaubt. Dies kann zu Schwierigkeiten führen, wenn Sie ein Linux-Shellskript auf einem proprietären Unix-System ausführen wollen.

Übungen

2.5 [!1] Mit welcher Ausgabe rechnen Sie, wenn das ausführbare Skript `blubb` mit dem folgenden Inhalt

```
#!/bin/echo bla fasel
echo Hallo Welt
```

über das Kommando »`./blubb`« aufgerufen wird?

2.6 [2] Was ist als erste Zeile für Shellskripte besser – »`#!/bin/sh`« oder »`#!/bin/bash`«?

2.4 Shellskripte planen

Wer auch nur ein bisschen Programmiererfahrung hat, hat die leidvolle Erfahrung gemacht: Programme sind selten auf Anhieb korrekt. Das *Debuggen* größerer Programme kostet meist sehr viel Zeit und Mühe. Der bekannte Programmierer und Autor Brian W. Kernighan formuliert das so:

> Jeder weiss, dass Fehlersuche zweimal so schwierig ist wie Programme erst mal hinzuschreiben. Wenn Sie also schon beim Programmieren so clever sind, wie Sie sein können – wie wollen Sie dann je die Fehler finden?

Es empfehlen sich also sorgfältige Planung, schrittweises Vorgehen und die Kenntnis der gängigsten Fehlerquellen.

Einfache Skripte

In die Situation, ein Shellskript verfassen zu müssen, kommen Sie meist, weil Sie eine bestimmte Tätigkeit automatisieren wollen. In einem solchen Fall ist einerseits die Aufgabe des Skriptes klar umrissen, andererseits wissen Sie auch schon grob, welche Kommandos Sie in welcher Reihenfolge ausführen müssen. Zum Schreiben des Skriptes bietet sich hier eine »evolutionäre« Herangehensweise an. Was soviel bedeutet wie: Sie schreiben erst mal alle Kommandos in eine Datei, die Sie auch auf der Kommandozeile eingegeben hätten, und verbinden sie dann auf sinnvolle Weise. Zum Beispiel können Sie Alternativen oder Schleifen einführen, wenn diese das Skript übersichtlicher, fehlertoleranter oder universeller machen. Auch mehrfach benutzte Dateinamen (etwa für Protokolldateien) sollten Sie in Variable schreiben und dann nur noch die Variablennamen verwenden. Kommandozeilenparameter können angeschaut und verwendet werden, um von Aufruf zu Aufruf unterschiedliche Elemente einzusetzen; natürlich sollten Sie dann auch die Vollständigkeit und Plausibilität dieser Parameter prüfen und gegebenenfalls Warnungen oder Fehlermeldungen ausgeben.

Evolution

Größere Projekte

Auch bei größeren Programmen können Sie die »evolutionäre« Herangehensweise wählen, was bedeuten würde, Sie fangen intuitiv mit dem an, was Ihnen als erstes einfällt und entwickeln darauf aufbauend das Skript. Der große Nachteil: Es passiert schnell, dass man einen Fehler im Konzept begeht, und die anschließenden Umschreibarbeiten machen einen Haufen unnötige Arbeit.

Plan

Sie sollten also auf jeden Fall einen groben Plan mit allen voraussichtlichen Einzelschritten aufstellen – dann erkennen Sie schneller, ob schon das Konzept

logische Fehler aufweist. Es reicht dabei vollkommen aus, wenn Sie sich die Einzelschritte in Form einer »natürlichsprachigen« Liste aufschreibt, mit den genauen Kommandos und ihrer Syntax können Sie sich dann später befassen.

Ein weiterer Vorteil dieser planenden Herangehensweise ist, dass Sie für die Einzelschritte im Vorhinein überlegen können, welche Kommandos sich am besten eignen, beispielsweise ob Sie sich mit einfachen Filterkommandos herumschlagen, nicht doch lieber `awk` oder gar gleich ein Perl-Skript nehmen wollen ...

Ein guter Plan hilft Ihnen allerdings wenig, wenn Sie nach einigen Monaten merken, dass am Skript etwas zu verbessern ist und Sie das Skript nicht mehr verstehen. Wenn Sie sich schon die Mühe machen, einen Plan zu erstellen, verewigen Sie ihn am besten im Programm selbst – nicht (nur) in Form von Kommandos, sondern in Form von Kommentaren. Dabei sollten Sie darauf verzichten, jedes einzelne Kommando zu kommentieren. Geben Sie lieber die grobe Struktur wieder und kommentieren Sie vor allem den Datenfluss und insbesondere das Format der Ein- und Ausgabe: Meistens ergibt die Verarbeitung der Daten sich relativ zwingend aus den Definitionen der Ein- und Ausgabe, während der Umkehrschluss sich bei weitem nicht so sehr aufdrängt. Auch ist es nicht falsch, bei umfangreicheren Programmen, externe Dokumentation (z. B. Handbuchseiten) zu erstellen.

Dokumentation

Ein guter Plan hat eine Struktur. Es schadet nicht, diese Struktur auch im Programmtext zum Ausdruck zu bringen, indem Sie zum Beispiel mit Einrückungen arbeiten. Das bedeutet, dass Sie Kommandos, die logisch gesehen auf derselben Ebene liegen, den gleichen Abstand vom linken Textrand geben. Hierbei können Sie wahlweise Tabulator- oder Leerzeichen verwenden, sollten aber konsistent vorgehen.

Einrückungen

Übungen

2.7 [!2] Sie möchten ein Shellskript erstellen, das für jeden »echten« Benutzer des Systems (also keine administrativen Konten wie `root` oder `bin`) den Zeitpunkt des letzten Einloggens und den gerade von dessen Heimatverzeichnis belegten Plattenplatz anzeigt. Wie würden Sie die folgenden Schritte anordnen, um einen vernünftigen »Plan« für ein Shellskript zu erhalten?

1. u, t und p ausgeben
2. Den Zeitpunkt t des letzten Einloggens von u bestimmen
3. Ende der Wiederholung
4. Den durch v belegten Plattenplatz p bestimmen
5. Eine Liste aller »echten« Benutzer aufstellen
6. Das Heimatverzeichnis v von u bestimmen
7. Wiederhole die folgenden Schritte für jeden Benutzer u in der Liste

2.8 [2] Entwerfen Sie einen Plan für die folgende Aufgabe: In einer großen Installation sind die Heimatverzeichnisse auf verschiedene Platten verteilt (stellen Sie sich zum Beispiel vor, dass die Heimatverzeichnisse heissen wie `/home/entwick/hugo` oder `/home/market/susi` für die Benutzer `hugo` aus der Entwicklungsabteilung und `susi` aus der Marketingabteilung). Sie möchten in periodischen Abständen prüfen, wie sehr die Platten mit den verschiedenen Heimatverzeichnissen ausgelastet sind; das Testskript soll Ihnen eine E-Mail-Nachricht schicken, wenn mindestens eine Platte zu mehr als 95% belegt ist. (Wir ignorieren hier die Existenz von LVM.)

2.5 Fehlertypen

Grundsätzlich können Sie zwei verschiedene Fehlertypen unterscheiden:

Konzeptuelle Fehler Hierbei handelt es sich um Fehler im logischen Ablauf des Programms. Es kann sehr aufwendig sein, solche Fehler zu erkennen und zu beheben. Sie sollten am Besten von vornherein vermieden werden – durch sorgfältige Planung.

Syntaktische Fehler Solche Fehler treten im Prinzip immer auf. Es reicht schon, einfache Tippfehler ins Programm einzubauen: Ein Zeichen vergessen und nichts läuft mehr. Viele syntaktische Fehler können Sie umschiffen, indem Sie sich beim Verfassen des Skriptes vom Einfachen zum Speziellen vortasten. So sollten Sie bei Klammerausdrücken erst *beide* Klammern eingeben und diese dann mit Inhalt füllen. Eine vergessene Klammer ist dann Schnee von gestern. Das gleiche gilt für Kontrollelemente: Schreiben Sie nie ein `if`, ohne direkt das `fi` eine Zeile dahinter zu setzen. Ein guter Editor, der Syntaxhervorhebung beherrscht, ist ein wichtiges Hilfsmittel zur frühzeitigen Vermeidung von solchen »strukturellen« Syntaxfehlern.

Natürlich können Sie sich immer noch bei Programmnamen und -optionen oder bei Verweisen auf Shellvariable vertippen. (Dabei hilft der Editor Ihnen auch nicht.) Die Shell hat hier einen eindeutigen Nachteil gegenüber »traditionellen« Programmiersprachen mit fester Syntax, die pingelig von einem Sprachübersetzer geprüft und gegebenenfalls angemeckert wird. Da dieser Übersetzungsschritt und die damit verbundenen Prüfungen Ihres Programms entfallen, müssen Sie besonderes Augenmerk darauf legen, Ihr Skript systematisch zu testen, damit möglichst alle Skriptzeilen, Zweige von Fallunterscheidungen usw. tatsächlich durchlaufen werden. Dieser Umstand legt den Schluss nahe, dass Shellskripte jenseits einer bestimmten »kritischen Masse« von einigen hundert bis tausend Zeilen nicht mehr wirklich beherrschbar sind – hier sollten Sie dann doch zumindest zu einer Skriptsprache mit besserer Syntaxprüfung, etwa Perl oder Python, greifen

Der Rest des Kapitels beschäftigt sich hauptsächlich mit der Diskussion der häufigsten syntaktischen Fehler und deren Behebung.

2.6 Fehlererkennung

Fehler erkennen Sie, wie erwähnt, erst, wenn das Programm abläuft. Darum sollten Sie Ihre Skripte schon während der Entwicklung so häufig wie möglich testen. Speziell wenn das Skript bestehende Dateien verändert, sollten Sie dabei auf eine »Testumgebung« zurückgreifen.

Testdurchlauf

Wenn Sie zum Beispiel automatisch Konfigurationsdateien in `/etc` editieren wollen, dann schreiben Sie Ihr Skript so, dass alle Verweise auf Dateien `/etc/...` als `$ROOT/etc/...` geschrieben werden. Zum Testen können Sie dann die Umgebungsvariable `ROOT` auf einen unverfänglichen Wert setzen und kommen dann vielleicht (hoffentlich!) sogar ohne Administratorrechte aus. Wenn Ihr Skript »`myscript`« heißt, können Sie es zum Testen aus einem »Testgerüst« aufrufen, das ungefähr so aussieht:

```
#!/bin/sh
# test-myscript -- Testskript für myscript
#
# Stelle die Testumgebung her
cd $HOME/myscript-dev
rm -rf test
cp -a files test    # Frische Kopie der Testdateien
# Rufe das Skript auf
ROOT=$HOME/myscript-dev/test $HOME/bin/myscript-dev "$@"
```

Dabei stehen in `~/myscript-dev/files` die ursprünglichen Dateien, die vor jedem Testlauf nach `~/myscript-dev/test` kopiert werden. Nach dem Programmlauf könnten Sie den Inhalt von `test` auch automatisch (etwa mit »`diff -r`«) mit einem weiteren Verzeichnis vergleichen, das Muster für die korrekte Ausgabe enthält. – `myscript` bekommt die Argumente von `test-myscript` übergeben; Sie könnten `test-myscript` also als Baustein in einer noch aufwendigeren Infrastruktur verwenden, die `myscript` mit verschiedenen vorgekochten Kommandoargumenten aufruft und jeweils automatisch testet, dass das Programm die erwarteten Ausgaben liefert.

Viele Shellskripte rufen externe Kommandos auf. In solchen Fällen können Sie bei der Fehlererkennung auf die eingebauten Fehlermeldungen des entsprechenden Programmes zurückgreifen – sofern es sich um syntaktische Fehler handelt.

Sollten die beschriebenen Fehlermeldungen nicht ausreichen, so können Sie unter anderem die Bash wesentlich geschwätziger machen. Häufig handelt es sich bei Syntaxfehlern nämlich um Fehler, die die Syntax der Shell betreffen – beispielsweise, wenn Substitutionen in einer nicht vorgesehenen Reihenfolge durchgeführt werden oder wenn Klammerfehler auftauchen.

Mit »`set -x`« können Sie sehen, was die Bash für Schritte unternimmt:

```
$ set -x
$ echo "My $HOME is my castle"
+ echo 'My /home/tux is my castle'
My /home/tux is my castle
```

Ablaufverfolgung

Das Ganze nennt sich auch Ablaufverfolgung (*tracing*).

Der Nachteil an »`set -x`« ist, dass der Befehl trotzdem ausgeführt wird. Im Falle von Substitutionen kann es besser sein, wenn die Kommandos nur so angezeigt werden, wie sie ausgeführt worden wären. Dazu steht die Option »`-n`« zur Verfügung. Allerdings funktioniert diese Option nur in Shellskripten, nicht aber in interaktiven Shells (warum wohl?).

Eine weitere nützliche Option ist »`-v`«. Hiermit wird das Kommando zwar ausgeführt, zusätzlich zeigt die Shell aber alle Kommandos an. Das heißt für ein Shellskript, Sie bekommen nicht nur seine Ausgaben, sondern auch das, was drin steht.

Mit der Option »`-u`« wird ein Fehler gemeldet, wenn Sie auf den Wert einer Variable zugreifen wollen und diese Variable gerade undefiniert ist. Auch das kann dabei helfen, obskure Probleme aufzudecken.

Alle vier Optionen lassen sich wieder abschalten, wenn Sie im `set`-Kommando statt des »`-`« bei den Optionen ein »`+`« setzen.

In der Praxis der Shellprogrammierung funktioniert das Ganze auch anders. Während der Entwicklung des Skriptes schreiben Sie einfach die entsprechende Option in die erste Zeile des Skripts:

```
#!/bin/bash -x
<<<<<
```

Als weitere Regel zur Fehlererkennung gilt: Bevor Sie ein Kommando, das »zweifelhafte« Substitutionen enthält, ausführen lassen, setzen Sie erstmal ein `echo`. Dadurch wird das Kommando komplett (das heißt, inklusive aller Substitutionen und Expansionen) ausgegeben, ohne ausgeführt zu werden. Für den schnellen Test reicht das allemal.

Kommandos in diesem Kapitel

`chmod` Setzt Rechte für Dateien und Verzeichnisse `chmod(1)` 24

Zusammenfassung

- Shellskripte bieten eine einfache Möglichkeit zur Automatisierung von Kommandofolgen.
- Sie können Shellskripte entweder explizit einer Shell als Dateiargument übergeben, sie ausführbar machen und direkt starten oder über `source` unmittelbar in die aktuelle Shell einlesen.
- Shellskripte sind Textdateien, die Folgen von Shellkommandos enthalten.
- Die erste Zeile eines ausführbaren Skripts kann ein Programm benennen (außer der Shell auch andere), das das Skript ausführen soll.
- Sorgfältige Planung beim Programmieren reduziert späteres Kopfzerbrechen.
- Die Bash verfügt über verschiedene Eigenschaften zur Fehlersuche.