CVS

Concurrent Versions System

Artikel von Matthias Kranz
im Linux Magazin 10/1998

überarbeitet von Lars Oergel

14. November 1999

Inhaltsverzeichnis

1 Einleitung

Das CVS Versions–Kontroll–System ist ein Programm zum Bearbeiten eines Software–Projekts oder überhaupt eines aus Textdateien bestehenden Verzeichnis–Baums. Aus dieser Sicht betrachtet ist es auch überaus interessant für Web–Admins – mehr dazu später.

2 Was ist CVS?

Zunächst etwas zum Konzept von CVS (Concurrent Versions System). CVS verwaltet die Entwicklungsgeschichte (neudeutsch History) eines Verzeichnisbaums, der Quelldateien beinhaltet. Konkret werden zu jeder bearbeiteten Datei die sukzessiven Veränderungen gespeichert. Dabei wird jede Änderung eines Files mit der aktuellen Zeit und einem Hinweis auf den Urheber der Änderung versehen. Zusätzlich kann die Datei mit beschreibenden Hinweisen versehen werden. Denn grundsätzlich stellt sich jedem Entwickler, zumindest in einer größeren Gruppe die Frage, wer wie zuletzt ein File bearbeitet hat und warum.

Zentraler Bestandteil von CVS ist das *Repository*, in dem die Quelldateien liegen. Diese werden nicht direkt editiert. Stattdessen legt sich jeder beteiligte Entwickler ein eigenes Arbeitsverzeichnis an. In diesem kann er nun die Files beliebig editieren, Files hinzufügen oder löschen, usw. Entscheidend ist, daß diese Veränderungen immer nur die lokalen Kopien betreffen. Möchte man seine Version der Dateien in das Repository einspielen und damit jedem anderen Entwickler seine Veränderungen bekannt machen, müssen diese *eingecheckt* werden. Vor diesem Vorgang muß ein Update der eigenen Dateien erfolgen. Bei diesem Vorgang werden nur die Dateien aus dem Repository übernommen, die jüngeren Datums als die eigenen sind. Damit werden Inkonsistenzen vermieden. Schließlich könnten in der Zwischenzeit auch mehrere Leute an der gleichen Datei gearbeitet haben. Nach dem erfolgreichen Update werden die Veränderungen in das Repository übertragen. Jedes künftige *Auschecken* oder Updaten erfolgt also dann mit der aktualisierten Version.

Natürlich ist dieses Arbeiten nicht nur für große Software–Entwicklergruppen interessant. Man findet sehr schnell weitere Anwendungsgebiete, in denen eine Versionskontrolle und ein Versionsmanagement sinnvoll sein können, z.B. beim Erstellen und Verwalten von vielen Web–Seiten. Nach der schnöden einführenden Theorie nun die praktische Anwendung.

3 Einrichten des Repositories

Wie schon oben erwähnt geht beim Concurrent Versions System nichts ohne das Repository. Bevor wir allerdings zur Einrichtung eines solchen kommen, setzen wir noch eine Environment–Variable – das erleichtert die Arbeit ungemein: `CVSROOT` heißt sie, und sie enthält den (vorzugsweise absoluten) Pfadnamen des Verzeichnisses, in dem das Repository gehalten werden soll. Alternativ kann man auch mit der `-d`–Option des `cvs`–Befehls den Repository–Pfad explizit setzen. Dieser überschreibt auch eine möglicherweise gesetzte Environment–Variable.

Zu Testzwecken könnte man jetzt also ein Verzeichnis anlegen, in dem unser Repository beheimatet sein soll. Vor der ersten Benutzung von CVS müssen wir natürlich zunächst festlegen, was überhaupt verwaltet werden soll. Dazu initialisieren wir in einem ersten Schritt das Repository: `cd $CVSROOT; cvs init`. Im Verzeichnis ist nun ein Unterverzeichnis `CVSROOT` erzeugt worden. Es ist under anderem Heimat für diverse generierte Hilfsdateien. Im nächsten Schritt können evtl. bereits vorhandene Sourcecode–Dateien unter CVS–Kontrolle gebracht werden:

```
# cd <alter-source-code-baum>
# cvs import -m 'Kommentar' src_import vendor_tag release_tag
```

Durch diese Operation wurde im Repository ein Verzeichnis `src_import` erzeugt, in dem nun sämtliche Dateien abgelegt sind. Sie haben alle die Endung `,v` bekommen, und bei näherer Betrachtung mit einem Editor fällt auf, daß sie mit Verwaltungs–Informationen erweitert worden sind. Verzichtet man auf obige `-m`–Option und den dazugehörigen Kommentar, wird automatisch ein Editor geöffnet. Dieser kann über die Environment–Variable `CVSEDITOR` gesetzt werden, ansonsten wird `EDITOR` ausgewertet. Beim `vendor_tag` muß ein Hersteller bzw. der eigene Name angegeben werden (allerdings sind hier keine Leerzeichen erlaubt), das `release_tag` bezeichnet symbolisch die erste Version (z.B. `start`).

4 Auschecken eines Verzeichnisses

Zum Anlegen eines lokalen Arbeitsverzeichnisses benutzt man das `checkout`–Kommando. Auf diese Art und Weise kann man beliebig oft den Inhalt des Repositories reproduzieren. Man legt in seinem Heimatverzeichnis beispielsweise das Verzeichnis `cvs-project` an und führt das folgende aus:

```
# cd ~/cvs-project
# cvs checkout src_import
cvs checkout: Updating src_import
U src_import/datei1.h
U src_import/datei1.c
U src_import/datei2.h
U src_import/datei2.c
```

Dann wird im Verzeichnis `cvs-project` das Unterverzeichnis `src_import` mit den entsprechenden Dateien angelegt. Diesen Vorgang kann man auch genauer spezifizieren. Bei Bedarf ist es z.B. über die Angabe eines *Tags* möglich, die beim Einchecken entsprechend markierte Datei auszulesen.

```
# cvs checkout -r release_tag
```

Oder es sollen beispielsweise alle Dateien von gestern ausgecheckt werden:

```
# cvs checkout -D yesterday src_import
```


5 Dateien verändern

Jetzt arbeitet man also mit seinen und an seinen Dateien, entwickelt vielleicht sogar innerhalb einer größeren Gruppe und nun stellt sich natürlich die Frage: Wie werden die Dateien konsistent gehalten bzw. wie befördert man seine veränderten Files wieder in das Repository? Im schwierigsten Fall haben ja zwei oder mehr Programmierer am gleichen Source–File gearbeitet. Wessen Veränderungen haben die höchste Priorität? Nun, CVS begegnet dieser Problematik zunächst einmal damit, daß man gezwungen wird, vor dem Einchecken, dem sog. *committen*, ein Update seines Arbeitsverzeichnisses durchzuführen. Versucht man eine Datei einzuchecken, die zwischenzeitlich von jemandem erneuert worden ist, erhält man eine Fehlermeldung:

```
# cvs commit datei2.h
cvs commit datei2.h
cvs commit: Up-to-date check failed for `datei2.h`
cvs [commit aborted]: correct above errors first!
```

Man wird also aufgefordert, zunächst ein Update vorzunehmen. Nun denn, nichts leichter als das.

```
# cvs update
cvs update: Updating .
RCS file: /home/developer/repository/src_import/datei2.h,v
retrieving revision 1.1.1.1
retrieving revision 1.2
Merging differences between 1.1.1.1 and 1.2 into datei2.h
M datei2.h
```

Dieses Updaten verlief ohne Probleme, d.h. die veränderte Datei `datei2.h` aus dem Repository konnte ohne Konflikte mit der Arbeitskopie verschmolzen werden. Dieses Verfahren hat sich in der Praxis als relativ zuverlässig erwiesen. Man sollte natürlich an dieser Stelle Vorsicht walten lassen und nach Merging–Meldungen überprüfen, ob sich die neu entstandenen Files noch kompilieren lassen (im Falle der Verwaltung von Source–Codes). Verläuft das Verschmelzen allerdings nicht reibungslos, wird man CVS durch Konflikt–Meldungen darauf aufmerksam gemacht.

```
# cvs update
cvs update: Updating .
RCS file: /home/developer/repository/src_import/datei2.h,v
retrieving revision 1.1.1.1
retrieving revision 1.2
Merging differences between 1.1.1.1 and 1.2 into datei2.h
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in datei2.h
C datei2.h
```

CVS meldet in diesem Fall also, daß es nicht in der Lage war, die beiden verschiedenen Dateien zusammenzufügen und legt beide Varianten entsprechend markiert in der betreffenden Datei ab. Nun bleibt es dem Programmierer überlassen, den Inhalt zu überprüfen, den Konflikt aufzulösen (evtl. in Absprache mit den anderen beteiligten Programmierern) und anschließend die Datei einzuchecken.

Wurde eine Datei, die man nur importiert, selbst aber gar nicht editiert hatte, inzwischen verändert, wird sie einfach aktualisiert. CVS macht uns durch verschiedene Meldungen darauf aufmerksam. Wie oben zu sehen, steht am Anfang der Zeile vor jeder Datei ein einzelner Buchstabe. Dabei steht `U` für Update, `M` für Modified und `C` für Conflict. Durch das `M` wird angezeigt, daß man selbst als letzter die Datei bearbeitet hat und etwaige Änderungen allen anderen noch nicht bekannt sind.

6 Veränderungen bestätigen

Nachdem das Updaten keine Fehler beim Verschmelzen vermeldet hat, sollte man seine bearbeiteten Sourcen in das Repository einchecken. Dazu dient der Befehl `commit`.

```
# cvs commit datei2.h
Checking in datei2.h:
/home/devloper/repository/src_import/datei2.h,v <- datei2.h
new revision: 1.2; previous revision: 1.1
done
```

Dabei wird automatisch der Editor geöffnet, und man wird aufgefordert, eine *Log–Message* einzugeben, eine kurze Beschreibung der Änderungen. Nun sind die Veränderungen für jedermann im Repository sichtbar, und jeder andere wird beim Auschecken bzw. Updaten seiner Sourcen die neue Version erhalten. Wie man sehen kann, wird bei obigem Vorgang automatisch die Revisionsnummer[1] erhöht.

7 Dateien hinzufügen oder löschen

Für CVS bedeutet das Löschen oder Hinzufügen von Dateien nichts anderes, als eine Änderung am bestehenden Datenbestand. Das hat auch einen einleuchtenden Grund. Wie schon erähnt, ist es ja unter CVS möglich, auf beliebige Revisionen einzelner Dateien zurückzugreifen, d.h. man kann z.B. nach Belieben eine lauffähige Variante seines Programms auschecken ohne auf aktuellere, aber möglicherweise fehlerbehaftete Teile Rücksicht nehmen zu müssen. Nun kann es ja durchaus vorkommen, daß in dieser Version Dateien vorhanden sind, die in der weiteren Entwicklung wieder verschwunden sind. Sie werden aber weiterhin mitverwaltet und existieren immer noch im Repository, auch wenn sie beim jetzigen Checkout nicht mehr auftauchen würden. Ebenso weiß CVS nichts von neuen Dateien, die lediglich im Arbeitsverzeichnis, aber noch nicht im Repository auftauchen. Diese müssen zunächst mit einem `add`–Kommand hinzugefügt und dann mit dem schon bekannten `commit`–Befehl eingecheckt werden.

```
# cvs add datei3.c
cvs add: scheduling file ‘datei3.c‘ for addition
cvs add: use ‘cvs commit‘ to add this file permanently
# cvs commit
/home/developer/repository/src_import/datei3.c,v <- datei3.c
initial revision: 1.1
done
```

[1] Zur Vermeidung von Verwechslungen verwendet die CVS–Terminologie nicht den Begriff *Version*. Einzelne Dateien haben eine *Release–Nummer*, einem komplex aufgebauten Software–Produkt, das aus mehreren hundert oder tausend Dateien bestehen kann, kann man *Release–Nummern* oder *–Tags* zuordnen.

Wie man erkennen kann, beginnt jede Datei mit der initialen Revisionsnummer 1.1.

Das Löschen funktioniert ganz ähnlich. Zunächst löscht man mit Hilfe des gewohnten Unix–Befehl `rm` die Datei aus seinem Arbeitsverzeichnis. Anschließend markiert man sie mit `cvs remove` zum Löschen. Durch Ausführen des `commit`–Befehls wird die Datei endgültig entfernt.

Komplizierter wird da schon ein Umbenennen von Files. Eine mögliche Lösung in diesem Fall ist mit den schon bekannten Kommandos zu erreichen. Zuerst gibt man der entsprechenden Datei einen neuen Namen. Im Anschluß muß man nun die alte Version aus dem Repository löschen. Natürlich darf man nicht vergessen, das neu benannte File mit Hilfe des `add`–Kommandos dem Repository hinzuzufügen. Nicht sehr elegant und auch mit einem Nachteil behaftet, denn die Log–Daten der Datei werden natürlich nicht übernommen[2].

8 Verwaltung mit Tags

Es wurde ja schon erwähnt, daß ein wichtiger Vorteil der Versionsverwaltung von CVS darin liegt, jederzeit Zugriff auf den älteren Stand einer Entwicklung zu haben. So kann man sehr einfach, sobald man beispielsweise einen stabilen Zustand seiner Software erreicht hat, das komplette Paket mit einem Release–Tag markieren und ist nun immer in der Lage, genau dieses Release wieder auszuchecken.

```
# cvs tag release-1-0
cvs tag: Tagging .
T datei1.h
T datei1.c
T datei2.h
T datei2.c
```

An strategisch wichtigen Punkten in der Entwicklung eines Projektes ist es sinnvoll, nicht nur einzelne Dateien, sondern wie in diesem Fall ein gesamtes Verzeichnis zu markieren. Hier lautet es `release-1-0`, und ab jetzt ist es möglich, über die Angabe dieses Release–Tags alle obigen Dateien im jetzigen Zustand zu erhalten. Der Befehl `status` zeigt einem ausführlich die entsprechenden Markierung einer Datei.

9 Dokumentieren ist alles

Die Kommentare, die die verschiedenen CVS–Befehle verlangen, sind in ihrer Bedeutung nicht zu unterschätzen. Ist man alleiniger Benutzer von CVS und verwaltet möglicherweise einen kleinen Baum von Web–Seiten mit Hilfe dieses Utilities, kann man natürlich durchaus auch über einen längeren Zeitraum den Überblick behalten. Arbeiten aber mehrere Menschen an einem Software–Projekt oder zieht sich ein umfangreiches womöglich über Jahre hinweg, ist ausführliche Dokumentationen unerläßlich.

Man hat ja im Prinzip zwei Möglichkeiten, um unter CVS Informationen über ältere Revisionen von Dateien zu bekommen. Zum einen gibt es ja das `log`–Kommando

[2] Eine andere Möglichkeit besteht in der direkten Manipulation des CVS–Repositories...

und zum anderen `diff`. Bei letzterem hat man dann sogar die Möglichkeit, sich die gesamte Veränderung im Detail zu betrachten. Aus diesem Grund ist es sinnvoll, aussagekräftige Formulierung in allgemein verständlicher Sprache als Log zu verwenden und sich nicht in die konkrete Implementierung zu vertiefen.

A CVS–Befehlsübersicht

`cvs checkout`	Checkt Dateien aus dem Repository aus
`cvs update`	Aktualisierung des Arbeitsverzeichnisses
`cvs add`	Hinzufügen neuer Dateien zum Repository
`cvs remove`	Entfernen von Dateien aus dem Repository
`cvs commit`	Bestätigen von Änderungen
`cvs status`	Anzeigen des Dateistatus‘
`cvs log`	Anzeigen der Log–Informationen
`cvs history`	Anzeigen der globalen CVS–Kommando–History
`cvs diff`	`diff` auf alle Dateien im aktuellen Verzeichnis
`cvs tag`	Vergabe eines Release–Tags

Literatur

[1] Cyclic Software.
http://www.cyclic.com.

[2] Pascal Molli's CVS Site.
http://www.loria.fr/˜molli/cvs–index.html.

[3] CVS Reference Manual.
http://www.loria.fr/cgi–bin/molli/wilma.cgi/doc.847210383.html.