Seminarvortrag PG571

Versionsverwaltung mit GIT

Bianca Patro

1. Oktober 2012

Inhaltsverzeichnis

Was ist Versionsverwaltung?

Versionsverwaltung

- VCS = Version Control System
- System zur Erfassung von Änderungen an Dateien und Dokumenten
- jede Version wird mit Änderungsdatum und Nutzer abgespeichert

→ es ist nachvollziehbar, wer wann was geändert hat

→ alle Versionen des Projekts werden archiviert, dadurch lassen sich Änderungen rückgängig machen

→ gleichzeitiges Arbeiten mit mehreren Personen wird ermöglicht

Nachteile von Subversion (SVN)

Zentrale Versionsverwaltung

SVN ist ein zentrales Versionsverwaltungssystem, d. h. es gibt einen zentralen Server, der alle Versionen enthält:

- totale Abhängigkeit vom Server $\rightarrow$ *Single Point of Failure*
- langsam, da für alle Operationen ein Netzzugriff notwendig ist

Konzeptionelle Schwächen

- Arbeitsverzeichnisse können sehr groß werden, da es für jede Datei eine Kopie im `.svn`-Ordner gibt
- Branches sind komplette Kopien

Neuerungen in GIT

Verteilte Versionsverwaltung

GIT ist ein verteiltes Versionsverwaltungssystem:

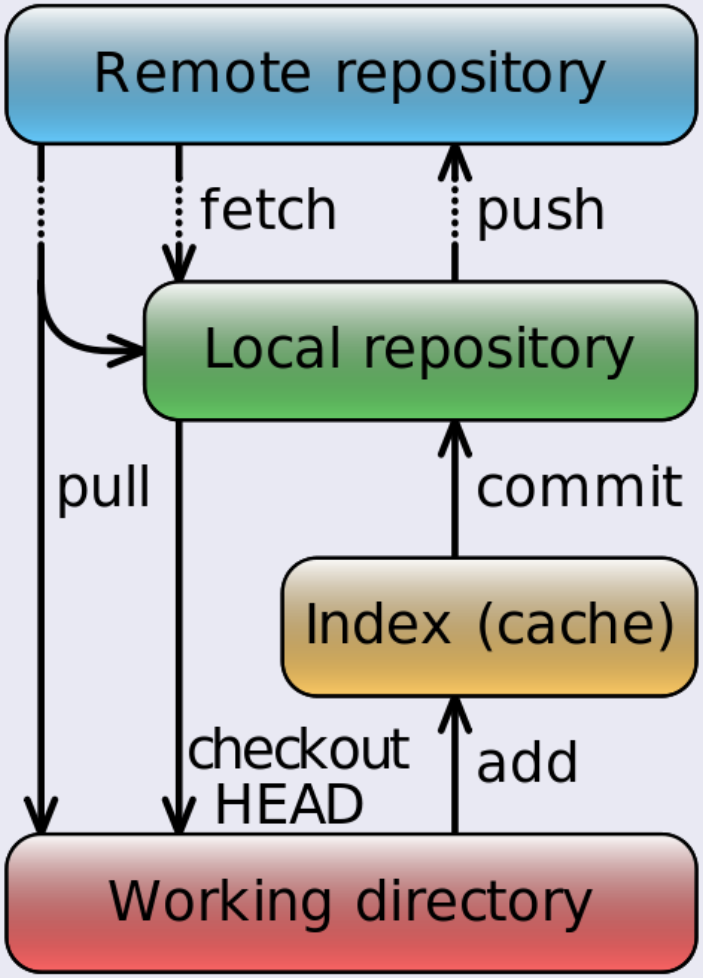
- es wird kein zentraler Server benötigt
- jeder Nutzer hat eine komplette Repository-Kopie (Clone)
  $\rightarrow$ dadurch ist das Backup inklusive
- parallele Entwicklungsstränge sind möglich
  $\rightarrow$ Branches als Verzweigungen des Entwicklungsprozesses
- alle Operationen werden lokal ausgeführt $\rightarrow$ schneller als SVN
- Synchronisation mit einem entfernten Repository über `pull/push`

Weiterer Vorteil

- effiziente Speicherung der lokalen Arbeitsverzeichnisse

Arbeitsweise von GIT

Lokales Arbeiten, Synchronisation mit Server



Server

Lokales Repository

Working Directory

Grundlegendes

Die Befehle für die Kommandozeile und für TortoiseGit unterscheiden sich zum Teil. Deswegen gilt im Folgenden:

- `Befehl` steht für den Kommandozeilenbefehl
- `Befehl` steht für den entsprechenden Aufruf unter TortoiseGit

Erste Schritte

Initialisierung eines neuen GIT-Repositories

Erstelle einen neuen Ordner. In diesem Ordner in TortoiseGIT auf `Git Create repository here...` klicken, oder in der Kommandozeile `git init` aufrufen.
Dies erzeugt ein ausgechecktes Repository als Arbeitsverzeichnis. Das Unterverzeichnis `.git` enthält alle Git-Verwaltungsinformationen.

Auf einem Server würde man ein nicht-ausgechecktes GIT erstellen:

```
git init --bare
```

oder

Mit einem vorhandenen GIT-Repository arbeiten

`Git Clone ...` → http://smAccount@holmes ... naodevils2010.git und einen Ordner angeben

```
git clone http://smAccount@holmes ... naodevils2010.git
```

Einstellungen

Globale User-Einstellungen

Es lassen sich globale User-Einstellungen setzen und speichern. `Settings` → `Git` → `Config` → `Save as Global`. Dort lassen sich *Name* und *Email* eintragen.

```
git config --global user.name "Vorname Nachname"
git config --global user.email Vor.Nachname@tu-dortmund.de
```

Diese Einstellung muss nur einmal vorgenommen werden. Die Nutzernamen tauchen in den Logs auf.

End of Line

Je nach Betriebssystem gibt es unterschiedliche Formate (Windows: *CR LF*, Unix: *LF*), deshalb sollten alle die gleichen Einstellungen verwenden: `git config core.autocrlf`

Dateien hinzufügen und committen

Eine neue Datei ins GIT hinzufügen

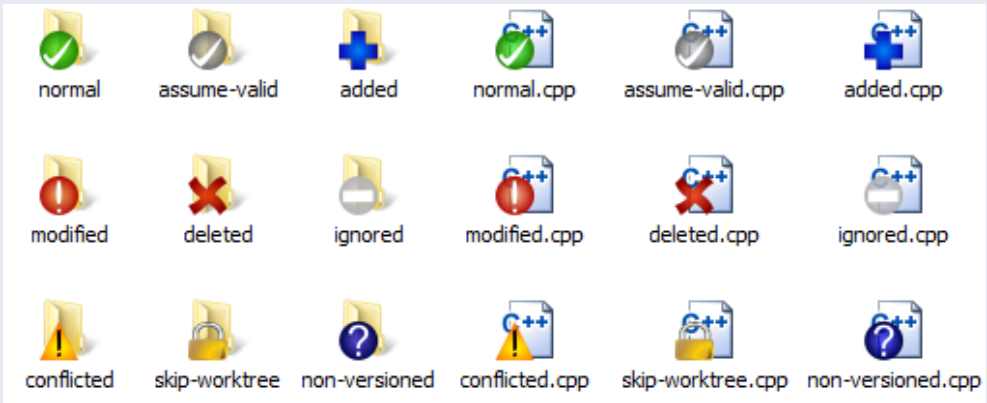
Die angegebene Datei wird mit `Add...` oder `git add myFile` der GIT-Kontrolle hinzugefügt.

Commit

In der ersten Zeile steht `Working dir changes`, also die lokalen Änderungen. Mit Rechtsklick auf diese wird ein `Commit...` gemacht.
In der Kommandozeile nutzt man `git commit -a` , ein zusätzliches `-m` ermöglicht direktes Eingeben einer Commit-Nachricht.
Mit dem Befehl werden alle geänderten und gelöschten Dateien committet, aber keine Dateien, die noch nicht unter Versionskontrolle stehen.

Status abfragen

Bei TortoiseGIT wird der Status einzelner Dateien und Ordner durch Icons angezeigt:



Alle veränderten, gelöschten, oder nicht getrackten Dateien auflisten:
`git status`
Änderungen in Dateien lassen sich mit Rechtsklick und
`Compare with Base` oder `Show differences as unified diff`
bzw. mit `git diff` anzeigen.

Synchronisation

fetch

Der Befehl `fetch` zieht erstmal nur die Änderungen vom Server herunter: `Fetch...` bzw. `git fetch`

pull

Durch `Pull...` bzw. `git pull` werden alle Änderungen seitens des Servers zum Client eingearbeitet. Intern führt der `pull` einen `fetch`, sowie - falls notwendig - einen `auto-merge` durch.

push

Mit dem Befehl `git push` bzw. `Push...` kann der aktuelle Stand zum Server gebracht werden.

Branches

Einen neuen Branch erstellen

`Create Branch at this version...`

```
git branch newBranch master
```

Dadurch wird noch nicht in den neuen Branch gewechselt.

In einen Branch wechseln

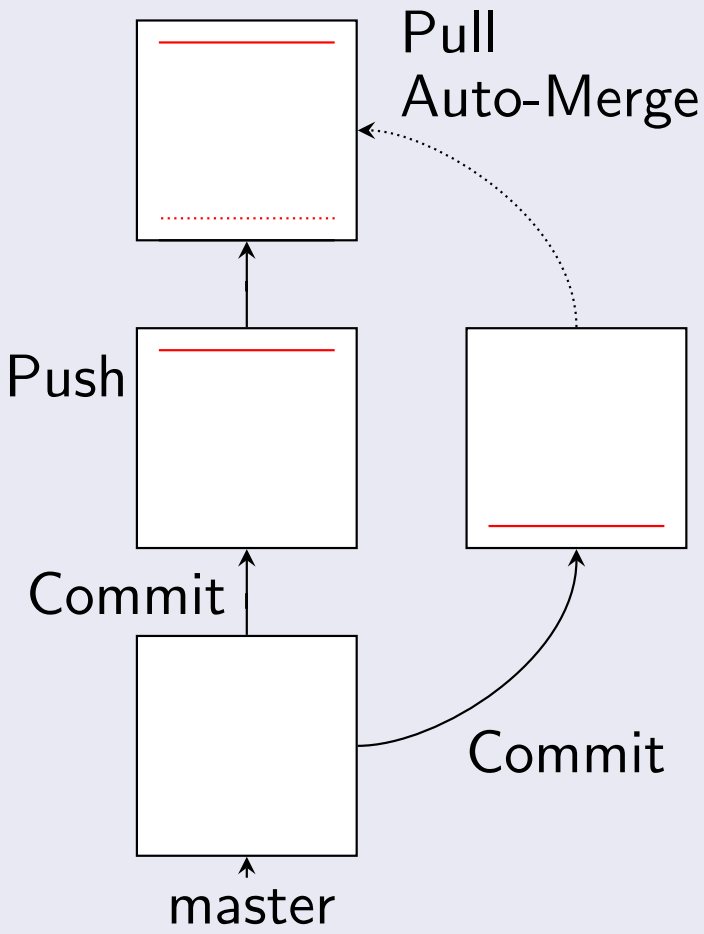
`Switch/Checkout to this...` `git checkout otherBranch`

Branch für alle Verfügbar

Um einen Branch für alle verfügbar zu machen, muss dieser beim `Push`, als `Local:`, sowie als `Remote:` eingetragen sein:

```
git push origin localBranch
```

Gemeinsames Arbeiten



- A arbeitet auf dem aktuellen master und committet seine Änderungen.
- A pusht seine Änderungen auf den Server.
- B hat aber ebenfalls etwas committet.
- Jetzt führt B einen Pull durch. Hierbei wird neben dem Fetch noch ein Auto-Merge durchgeführt.
- (Hinweis: Das ist jetzt erstmal nur lokal. Muss noch gepusht werden.)

Merging 1

Im Beispiel waren die Änderungen lokal unabhängig voneinander → Idealfall, der nicht immer auftritt.

Auto-Merge schlägt fehl

Falls Konflikte auftreten, muss manuell gemerget werden.
Hierbei insbesondere beachten:

- Es gibt die andere Version, den eigenen Commit und die „neue Version“. Die neue Version vereint beide Pfade.
- Änderungen von beiden Seiten sollten nachher enthalten sein.
  - `Resolve conflict using theirs` übernimmt komplett die Änderungen der anderen Version; `git checkout` (siehe später)
  - `Resolve conflict using mine` übernimmt komplett die Änderungen des eigenen Commits; `git merge -s ours`
  - oder die Datei manuell bearbeiten, beides einarbeiten und `Resolved` auswählen bzw. die Datei dann mit `git add` hinzufügen

Merging 2

Branches in den Master einpflegen

Auch zum Einpflegen von Branches in den Hauptzweig wird Merging benötigt. Dabei gibt es verschiedene Merge-Strategien, die sich angeben lassen:

- `git merge` verwendet standardmäßig den rekursiven Algorithmus (`-s recursive`), welcher weitere Parameter erhalten kann:
  - `-Xours` nimmt nur von konfliktbehafteten Stellen unsere Version
  - `-Xtheirs` nimmt nur von konfliktbehafteten Stellen die andere Version
  - `-Xignore-space-change`, `-Xignore-all-space`, `-Xignore-space-at-eol` ignoriert Änderungen die nur Leerzeichen betreffen
- `git merge -s resolve` verwendet den gleichen 3-Wege-Algorithmus, geht aber nicht rekursiv in die Tiefe
- `git merge -s octupus` zum Mergen mehrerer Branches in einen Branch, führt keine komplexe Konfliktbehandlung durch

Änderungen rückgängig machen 1

Hier muss man **Unterscheiden**: checkout, reset, revert

checkout - (TortoiseGIT: revert)

Bei checkout lassen sich Dateien einer beliebigen Version auschecken und damit aktuelle Änderungen rückgängig machen:
Unten im *Log* bei *Path* auf die entsprechende Datei klicken und `Revert...` auswählen bzw. `git checkout HEAD myFile` .

Änderungen rückgängig machen 2

reset

Beim reset sollen ganze Commits rückgängig gemacht werden. Man geht auf eine Version im *Log* und macht `Reset \master\ to this...` und ist damit wieder auf diesem Stand. `git reset --hard HEAD`

revert

Beim revert sollen nur die Änderungen, die durch einen Commit entstanden sind, rückgängig gemacht werden.
`Revert change by this commit` bzw. `git revert 38f6` .
Die angegebene Nummer entspricht der eindeutigen Nummer des Commits.

Änderungen rückgängig machen 3

Es gibt also drei verschiedene Möglichkeiten Dateien auf einen vorherigen Stand zurückzusetzen. Aber es sollte zwischen den Zielen unterschieden werden:

checkout Dabei holt man einen alten Stand wieder hervor.
Der HEAD wird hierbei nicht verrückt.

reset Setzt den HEAD auf eine bestimmte Version, um diese nun als HEAD zu verwenden.
In Folge dessen sollen die Versionen darüber komplett verworfen werden. Sie tauchen gar nicht mehr auf!

revert Wirklich nur die Änderungen eines speziellen Commits rückgängig machen.
Eine Anwendung wäre, einen Patch rückgängig zu machen.
Der HEAD wird hierbei nicht verrückt.

History

Show log

Mit `Show log` bzw. `git log` hat man eine Übersicht über die History.
Oder mit grafischer Oberfläche unter Linux: `gitk`, `qgit`

- Die Farbe gelb kennzeichnet existierende Branches und die **rot-farbige fette** Markierung den Branch auf dem wir gerade arbeiten.
- Ein „origin/“ vor dem Branchnamen bedeutet, dass der Branch auf dem Server verfügbar ist.
- Unten aktiviert `All Branches`, dass auch alle Branches angezeigt werden.

Log-Übersicht

`Show log` - Graph

Der angezeigte Graph stellt die Commits sowie Merges im Projekt dar.

- Jeder Knoten ist ein Commit.
  (Ein Merge ist auch ein Commit)
- Zweiggabelungen: signalisieren parallele Entwicklungsstränge.
  Dies sind nicht zwingend Branches. Meist wurde einfach nur parallel gearbeitet.
- Farbe und Linienposition haben keine Relevanz.

Sonstiges

Tags

Ein Tag ist eine Referenz auf einen Commit. Damit kann man z.B. stabile Versionen markieren. `Create Tag at this version...`
`git tag -a newTag <tag>`

Um Tags zu pushen muss man beim Push `Push...` `Include Tags` setzen. `git push --tags`

Wenn man sich nur die Tags anzeigen lassen will, geht man oben über `All Branches` auf `tags` oder gibt `git tag -l` ein.

Ansonsten lassen Tags sich normal auschecken:
`git checkout newTag`

Problemfälle

1. Pullen funktioniert nicht

Entweder: Es sind unbehandelte Änderungen vorhanden → Dateien zurücksetzen oder Änderungen comitten
Oder: Das Auto-Merge beim Pullen schlägt fehl → Konfliktbehandlung, Mergen von Hand, `git mergetool`
Oder: Etwas anderes, z.B. Dateien gelöscht, ohne `git rm` zu verwenden

2. Pushen funktioniert nicht

Vorm Pushen muss das Repository auf dem aktuellen Stand sein → pullen
Oder es sind noch Probleme vorhanden (siehe oben)

Häufige Anfängerprobleme

- Nach Änderungen vergessen auf `Refresh` zu drücken und auf einer alten Darstellung gearbeitet
- Commit und Merge nicht getrennt; Erst eigene Änderungen committen, dann Merge ausführen
- Beim Mergen werden zwei Pfade gemergt, deshalb werden auch die Änderungen durch den anderen Pfad angezeigt (Parent 1, Parent 2); Man entscheidet auch über Änderungen von anderen
- Beim *pull* kommt die Meldung: „Please, commit your changes or stash them before you can merge. Aborting“, was nicht heißt ihr sollt einfach einen *commit*/*push* durchführen. GIT will nur nicht die potentiell geleistete Arbeit wegwerfen und ruft deshalb auf diese zu commiten.

Konventionen

- Generell nur kompilierbares & ausführbares commiten.
  Falls nicht:
  - In einen eigenen Branch packen
- Am Besten: getätigte Commits nochmal kurz angucken
  - Falls notwendig: Erst im nächsten Schritt den Merge durchführen
  - Dann den fertigen Stand pushen.

Zusammenfassung

- GIT ist ein verteiltes Versionsverwaltungssystem, jeder Nutzer hat einen Clon des kompletten Repositories
- GIT weist einige Vorteile gegenüber SVN auf
- Neue Repositories werden initialisiert, oder bestehende geclont
- Mit `add` werden Dateien hinzugefügt, mit `commit` erfolgt ein Commit
- Der Status lässt sich abfragen und Änderungen lassen sich zeilenweise anzeigen
- Die Synchronisation mit dem Server erfolgt über `push`, `pull` und `fetch`
- Konflikte können über Merging behoben werden
- Änderungen können wieder rückgängig gemacht werden
- Die komplette History ist verfügbar

Quellen & Literatur

Der Vortrag wurde erstellt in Anlehnung an den Vortrag *PG560 - GIT (aus SVN-Sicht)* von Max Vallender.

GIT

- Start:
  - Tutorial: `http://schacon.github.com/git/gittutorial.html` `man gittutorial`
  - Buch Pro-GIT: `http://book.git-scm.com/index.html`
  - Weiteres Tutorial: `http://learn.github.com/p/setup.html`
- GIT-Manual-Seiten zu den einzelnen Befehlen:
  - `http://linux.die.net/man/1/git-...` `man git-...`
- `http://schacon.github.com/git/user-manual.html`

TortoiseGIT

- `http://code.google.com/p/tortoisegit`