Gitting started

Christian Neukirchen

14dec2011

Agenda

Wieso Versionskontrolle?

Erste Schritte

Branches

Zusammenarbeit

Nicht-trivale Features

GUI

Ausblick

Wieso Versionskontrolle?

Den Verlauf eines Projekts speichern

Zu alten Versionen zurückkehren

“Gestern ging das noch…”

Mehrere Versionen parallel pflegen

Gemeinsam an einem Projekt arbeiten

Zur Geschichte

Klassisch: SCCS (1972), RCS (1982),
CVS (1989), SVN (2000)

Verteilt: GNU Arch (2001), Monotone (2003),
Darcs (2003), Bazaar (jan2005),
Git (7apr2005), Mercurial (19apr2005)

Wieso verteile Versionskontrolle?

Jedes Repo hat die *gesamte* History.

Alle *repositories* sind a priori gleich.

Ideal für *open source*…

… aber auch für die Arbeit offline.

Wieso Git?

Populär: Linux, X.org, GCC, Android, Wine, Exherbo, Fedora, GNOME, Perl 5, Rails, Qt, VLC, Samba…

Schnell, Robust, Unixig

GPL

Jetzt gits los!

```
% packer -S git

% emerge -av dev-vcs/git

% brew install git-core

% apt-get install git-core
```

whoami

Um Commitbeschreibungen sinnvoll auszufüllen, braucht Git einmalig den Namen und die Mail-Adresse des Users.

```
% git config --global user.name \
    "Christian Neukirchen"
% git config --global user.email \
    chneukirchen@gmail.com
```

Editor kontrollieren:

```
% echo $EDITOR
```

Erste Schritte: Workshop

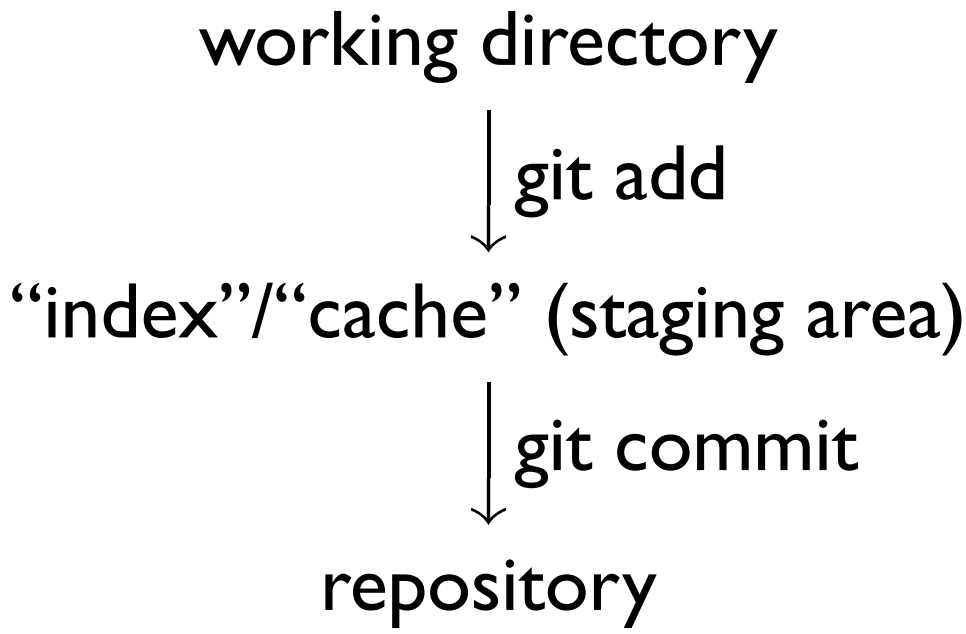
```
% git init myrepo
% cd myrepo

myrepo% vi hello-world.c
myrepo% git status
myrepo% git add hello-world.c
myrepo% git status
myrepo% git commit
```

Zweite Schritte: Workshop

```
myrepo% git log
myrepo% vi hello-world.c
myrepo% git status
myrepo% git diff
myrepo% git add hello-world.c  # !!!
myrepo% git diff --staged
myrepo% git commit
```

Der Index


working directory
git add
“index”/“cache” (staging area)
git commit
repository

Interaktiv committen: Workshop

```
myrepo% git add -p
myrepo% git commit
```

Schneller: Workshop

```
myrepo% vi hello-world.c
myrepo% git commit hello-world.c
```

Noch schneller: Workshop

```
myrepo% vi hello-world.c
myrepo% git commit -a
```

Mehrere Dateien: Workshop

```
% mv hello-world.c goodbye-world.c
% git add goodbye-world.c
% git rm hello-world.c
% git status
% git commit

% git mv hello-world.c goodbye-world.c
```

Git speichert das Umbenennen *nicht*, sondern erkennt es heuristisch. Funktioniert prima.

Wie funktioniert das alles?

```
git show-ref
git show --format=raw $COMMITID
git ls-tree $TREEID
git show $BLOBID
```

Exkurs: Revisionsnamen angeben

`HEAD`	Stamm
`b130ec682`	Eindeutiges Hash-Präfix
`refs/tags/$NAME`	Tags
`refs/heads/$NAME`	Andere Stämme
`refs/remotes/$NAME`	Andere Repos
`$REF^`	Erster Vater
`$REF~n`	n-ter Vorfahre
`$REF:$PATH`	`$PATH` im Tree `$REF`
`$REF1..$REF2`	Alle Eltern von `$REF2` ohne Eltern von `$REF1`

Branching

```
% git branch
% git checkout -b chris2

% git branch
% git log
% git log master
% git log master..chris2
% git diff master..chris2
```

Merging

```
% git checkout master
% ...
% git diff chris2
% git merge chris2
% git diff
% git log --graph --oneline
```

Tagging

```
% git tag $TAGNAME
% git show-ref
```

fetch, pull und push

```
% git clone ssh.fs.lmu.de:/home/gaf/alle/Git/hello
hello% git fetch
hello% git pull

hello% git remote add goodbye \
          ssh.fs.lmu.de:/home/gaf/alle/Git/goodbye
hello% git fetch goodbye
hello% git log master..goodbye/master
hello% git log -p master..goodbye/master
hello% git merge goodbye/master

hello% git pull goodbye master

hello% git push
```

(Das Remote, auf das standardmäßig gepusht wird, kann man mit `git push -u ...` festlegen.)

Neue remote Repos erzeugen

Will man in ein Repo pushen, so sollte man es als bare repo anlegen:

```
% ssh hund.fs.lmu.de
hund% git init --bare myrepo.git
hund% ls myrepo.git
HEAD    branches/  config    description
hooks/  info/      objects/  refs/

myrepo% git remote add hund hund.fs.lmu.de:myrepo.git
myrepo% git push hund master
```

Git Repos publizieren

HTTP

Github

Gitorious

Google Code

gitosis/gitolite

blame

Wer hat das verbrochen?

```
% git blame hello-world.c
```

Patches erstellen

```
% git checkout catwell/master

% git diff master..

% git format-patch \
    --stdout origin/master

% git format-patch origin/master~3
```

Empfehlungen

Viele, kleine Commits

Spezifische Commits

Sinnvolle Commit-Messages

Feature-Branches

Nur einer (oder wenige) committed in den master.

Workflow

Commit-Zyklus (local):

git add

git commit

Publish-Zyklus (global):

git pull

git push

Upps, was vergessen/vertippt

```
% git show-ref master
% git commit --amend
% git show-ref master
```

Die Operation ist *destruktiv*, “history is rewritten”.
Nur Privat verwenden!

Upps, zurück…

```
% git add foo
# Upps, nee...
% git reset

% git commit -a
% git log
# Upps, nee...
% git reset --hard HEAD^
% git log
```

Obacht, kann zu Datenverlust führen!

Wie werd ich diese tausend Dateien los?

```
% find .git/objects
% git gc
% find .git/objects
```

Eigentlich nicht nötig, von Hand aufgerufen zu werden.

Dateien ignorieren

```
% latex foo.tex
% git status
% echo "*.aux" >>.gitignore
% echo "*.toc" >>.gitignore
% git status
```

Am besten committed man die `.gitignore` auch ins Repo.

Stashing

```
*werkel... oh mist, ich muss pullen*
% git pull
Updating 3e4833b..85ca454
error: Your local changes to the
following files would be
 overwritten by merge:
        lib/rack/request.rb
% git stash
% git pull
% git stash list
% git stash apply
*weiterwerkel*
```

Git GUIs

gitk (Visualisierung der History, nützlich)

git-gui (Grafisches Commit-Tool)

Mac OS X: GitX, Gity

Ausblick

```
% ls -l /usr/lib/git-core | wc -l
153
```