Git: a brief introduction

Randal Schwartz
merlyn@stonehenge.com
http://www.stonehenge.com/merlyn/
Version 1.05 on 19-Dec-2006

Disclaimer

- I've been tracking git since it was created
- I've used git on small projects
- I've used other systems on small and large projects
- I've read a lot of people talk about git on the mailing list
- I've even provided some patches to git, and suggestions for user interface changes
- I do not have any experience working in a team with git

What is git?

- Git manages changes to a tree of files over time
- Git is optimized for:
  - Distributed development
  - Large file sets
  - Complex merges
  - Making trial branches
  - Being very fast
  - Being robust
- Git is not optimized for:
  - Tracking file permissions and ownership
  - Tracking individual files with separate history
  - Making things painful

Why git?

- Essential to Linux kernel development
- Created as a replacement when BitKeeper suddenly became “unavailable”
- Everybody has a “commit bit”
- Anyone can:
  - Clone the tree
  - Make and test local changes
  - Submit the changes as patches via mail
  - OR submit them as a published repository
  - Track the upstream to revise if needed
- Kernel.org is just a convenient public watering hole

How does git do it?

- Universal public identifiers
  - None of the SVK “my @245 is your @992”
- Multi-protocol transport
  - HTTP
  - SSH
  - GIT
- Efficient object storage
- Everyone has entire tree (disk is cheap)
- Easy branching and merging
  - Common ancestors are computable
- Patches can be transported or mailed
- Binary “patches” are supported

The SHA1 is King

- Every “object” has a SHA1 to uniquely identify it
- “objects” consist of:
  - Blobs (the contents of a file)
  - Trees (directories of blobs or other trees)
  - Commits:
    - A tree
    - Plus zero or more parent commits
    - Plus a message about why
  - Tags:
    - An object (usually a commit)
    - Plus a subject
    - Plus an optional payload to sign it off (like a gpg signature)

Objects live in the repo

- Git efficiently creates new objects
- Objects are generally added, not destroyed
- Unreferenced objects (deleted branch, added but not committed, etc) can be garbage collected
- Objects start “loose”, but can be “packed” for filesystem efficiency
- “Packs” often represent objects as deltas for space efficiency
- “Packs” are also created for inter-repo transfer

Commits rule the repo

- One or more commits form the head of object chains
- Typically one head called “master”
- Others can be made at will (“branches”)
- All reachable repo objects are accessed from these head commits
  - Chase down the tree object to get to directories and files as they existed at this commit time
  - Chase down the parent objects to get to earlier commits and their respective trees
- Usually one commit in the repo that has no parent commit

Mapping objects to a file tree

- A “working tree” has a “.git” dir at the top level
- Unlike CVS, SVN: no pollution of deeper directories
- The .git dir contains:
  - config – Configuration file (.ini style)
  - objects – The object repository
  - refs/heads/* – branches (like “master”)
  - refs/tags/* - tags
  - refs/heads/remotes/* - tracking others
  - index – the “index cache” (described shortly)
  - HEAD – points to one of the branches (the “current branch”, where commits go)

The index (or “cache”)

- Contains the mapping between a tree object and the filesystem
- Understand this, and you understand commits and merges
- Some commands move entries between the file tree and the index
- Some commands move entries between the index and the object repository
- Some higher-level commands do both at once

The great renaming controversy

- Git doesn’t record explicit renaming
- Nor expect you to declare it
- Exact renaming can be determined by SHA1 comparisons
- Copy-paste-edits can be detected by ad-hoc “similarity” rules for logs
- In general the computer can find these far easier than you can
- If you make it explicit, it will be wrong sometimes
- Being wrong breaks merges
- Even being right will break some merges!

Git speaks and listens

- Many protocols to transfer git data between repositories
  - rsync, http, https, git, ssh, local files
- In the core, git also has:
  - import/export with CVS
  - import/export with SVN
  - import from Arch
- I use CVS/SVN import to have entire history of a project at 30K feet
- Third party solutions handle others
- Git core also includes cvs-server
  - A git repository can act like a CVS repository for legacy clients or humans

Getting git gotten to your gut gantry

- Get the latest “git-*.tar.gz” from http://www.kernel.org/pub/software/scm/git/
- RPMs and Debian packages also exist
- Once you’ve bootstrapped git, git-clone the git-developer archive
  - git-clone git://git.kernel.org/pub/scm/git/git.git
- Rebuild once a week or so
- Maintenance releases are very stable
- I install mine “prefix=/opt/git”, and add /opt/git/bin to my PATH

Git commands

- All git commands start with “git-”
- “git-MUMBLE-FOO bar” can also be written as “git MUMBLE-FOO bar”
- This allows a single entry “git” to be added to the /usr/local/bin path
- But then it’s harder to auto-complete on a git-command
- The manpages are still under “git-MUMBLE-FOO”, so you have to know the convention either way

“The Internet is just a series of tubes”

- Low-level git operations are called “plumbing”
- Higher level actions are called “porcelain”
- The git distro includes git plumbing and git porcelain
- We’ll stick with the git porcelain in the rest of this talk, unless otherwise noted
- Other porcelain exists:
  - cogito
  - stacked git

Common steps to add objects

- git-add file1 file2 file3 dir1-and-all-kids
  - Add new or changed files to the index
- git-status
  - See working tree relative to index
- git-commit
  - Commit the index as a commit
  - You'll be asked for a commit message
  - Make sure the first line stands alone (for short logs)
- If you change something after you add it, you need to add it again!
- “git-commit -a” is your friend

But before you can do that…

- git-init-db
  - Creates a .git in the current dir
  - Optional: edit .gitignore
  - “git-add .” to add all files (except .git!)
  - Then “git commit” for the initial commit
  - Creates a branch named “master”
  - HEAD points at “master”
- git-clone REMOTESPEC
  - Creates a git repo from an existing repo
  - Generally creates a subdirectory
  - Your workfiles and .git are in there
  - Remote branch defaults to “origin”
  - Checked out as your “master”
  - (But this is changing even today)

Linear History

- You could keep editing, adding, and committing
- You'd end up with a linear history of all your changes
- But efficient, even if you rename
- Read the history with git-log:
  - git-log
    - print the changes
  - git-log -p
    - print the changes, including a diff between revisions
  - git-log file1 file2 dir3
    - Show changes only for listed files or subdirs

Branching

- Git encourages branches for speculation
  - Branches are lightweight (41 bytes!)
  - When you toss them, they really disappear
  - You can share a branch with others
- Create a branch
  - git-checkout -b topic1
- The next commit affects topic1, not master (and HEAD tracks topic1)
- At any time, “git-checkout BRANCH” to switch
  - Requires the working tree not be dirty
- Thus, many branches can be active at once, each with an independent history

Merging

- When a branch is good:
  - git-checkout MERGETO
  - git-pull . MERGEFROM
- If there are no conflicts, you're done!
- If there are conflicts, fix them, and "git commit -a"
  - Git knows you're in the middle of a merge
- You can then delete the old topic:
  - git-branch -d MERGEFROM
- Or you can sync the other way:
  - git-checkout MERGEFROM
  - git-pull . MERGETO
- There should be no conflicts now!

What's the difference?

- git-diff
  - Diff between index and working tree
  - These are things you should "git-add"
  - "git-commit -a" will also make this list empty
- git-diff HEAD
  - Difference between HEAD and working tree
  - "git-commit -a" will make this empty
- git-diff --cached
  - between HEAD and index
  - "git-commit" (without -a) makes this empty
- git-diff OTHERBRANCH
  - Other branch and working tree
- git-diff BRANCH1 BRANCH2
  - Difference between two branch heads

Barking up the tree

- Most commands take “tree-ish” args
- SHA1 picks something absolutely
  - Can be abbreviated if not ambiguous
- HEAD, some-branch-name, some-tag-name, some-origin-name
- Any of those followed by:
  - ^n - “the n-th parent of an item” (default 1)
  - ~n - n ^1’s (so ~3 is ^1^1^1)
  - :path - pick the object from the tree
- Examples:
  - git-diff HEAD^ HEAD
    - most recent change on current branch
  - git-diff HEAD~3 HEAD
    - What damage did last three edits do?

Seeing the changes side by side

- Use git-show-branches to get a display of all heads back the common ancestor
- git-show-branches mytopic origin
  - Shows history from mytopic and origin back to common ancestor
- Use “gitk” to get the same thing visually (using Tk widgets)
- And “qgit” (which I’ve never used)

Playing well with others

- “origin” from a git-clone is just another branch
  - Although you generally can’t “git-checkout”
- First, get up to date:
  - git-fetch origin
  - git-checkout master
  - Merge all your topics into master
  - git-merge origin (or reverse these two steps)
  - Resolve any conflicts, and test again
- To push upstream:
  - git-push (args)
- To send it in as patches:
  - git-format-patch origin
  - Then email the patches in the current dir

Repacking and pruning

- git-repack -a -d
  - Deletes existing packs
  - Gathers all items reachable from all refs
  - Creates one big fat pack
- git-prune
  - Deletes any remaining loose items
  - These are not referenced from any head/branch/tag/remote
- Great for ensuring that you have everything you need...
  - ... as efficiently as possible ...
  - ... but nothing more

Resetting

- git-reset --soft
  - Makes all files “updated but not checked in”
- git-reset --hard # DANGER
  - Forces working dir to look like last commit
- git-reset --hard HEAD~3
  - Tosses most recent 3 commits, and updates the work dir to match that
  - Don’t do this if you’ve published: use git-revert instead

Rebasing

- Problem: you submit patches
- And only some of them are accepted
- And other changes are made in the meanwhile
- You have worthwhile changes that still want to be submitted
- Solution: rebase
  - git-checkout yourtopicbranch
  - git-rebase origin
  - Might need to resolve conflicts
- Now yourtopicbranch appears to be a delta to the new origin, not the ancestor
- And you can resubmit the patches!

Pay no attention to the man behind the curtain

- Every directory can contain a .gitignore
  - lines starting with “!” mean “not”
  - lines without “/” are checked against basename
  - otherwise, shell glob via fnmatch(3)
  - Leading / means “the current directory”
- Checked into the repository and tracked
- Every repository can contain a .git/info/exclude
- Both of these work together
- But .git/info/exclude won’t be cloned

Configuration

- Many commands have configurations
- git-repo-config name value
  - set name to value
  - name can contain periods for sub-items
- git-repo-config name
  - get current value
- git-repo-config -l
  - get all values
- git-repo-config --global name [value]
  - Same, but with ~/.gitconfig
  - This applies to all git repos from a user
- At a minimum, set user.email and user.name for commits

Other useful porcelain

- git-archive: export a tree as a tar/zip
- git-bisect: find the offensive commit
- git-cherry-pick: selective merging
- git-mv: rename a file/dir with the right index manipulations
- git-rm: ditto for delete
- git-push: write to an upstream
- git-revert: add a commit that undoes a previous commit
- git-blame: who wrote this?

Time to “git” dirty

- Make a git repository:
  - mkdir git-tutorial
  - cd git-tutorial
  - git-init-db
- Add some content:
  - echo "Hello World" >hello
  - echo "Silly example" >example
- What’s up?
  - git-status
  - git-add example hello
  - git-status

Beware the timing of “git-add”

- Change the content of “hello”
  - echo "It's a new day for git" >>hello
  - git-status
  - git-diff
- Now commit the index (with old hello)
  - git-commit -m initial
  - git-status
  - git-diff
  - git-diff HEAD
- Note that we committed the version of “hello” at the time we added it!
- Fix this by adding -a nearly always:
  - git-commit -a -m update
  - git-status

What hath Randal wrought?

- Ask for logs:
  - git-log
  - git-log -p
- Tag, you’re it:
  - git-tag my-first-tag
- Now we can always get back to that version later

Suddenly, a new developer arrives

- Create the clone:
  - cd ..
  - git-clone git-tutorial my-git
  - cd my-git
- The git-clone will often have some sort of transport path, like git: or rsync: or http:
- See what we've got:
  - git-log -p
- Note that we have the entire history
- And that the SHA1s are identical
- This is essential to remerging later

Branching out into new territory

- Create branch “my-branch”
  - git-checkout -b my-branch
  - git-status
- Make some changes:
  - echo "Work, work, work" >>hello
  - git-commit -a -m 'Some work.'
- Switch back, and make other changes:
  - git-checkout master
  - echo "Play, play, play" >>hello
  - echo "Lots of fun" >>example
  - git-commit -a -m 'Some fun.'
- We now have conflicting commits

Seeing the damage

- In an X11 display:
  - gitk --all
- The --all means “all heads, branches, tags”
- For the X11 challenged:
  - git-show-branch --all
- This is handy because you can drop it into a mail message
- With these displays, we can see how branching and merging work

I feel the urge, the urge to merge

- We’re on “master”, and we want to merge in the changes from my-branch
- Select the merge:
  - git-merge “Merge work in my-branch” HEAD my-branch
- Yes, “HEAD” is required there, and it’s required to be “HEAD” :)
- This fails, because we have a conflict in “hello”
- See this with:
  - git-status
- Edit “hello”, and commit:
  - git-commit -a -m “Merge work in my-branch”

What did the merge do?

- Verify the merge with:
  - gitk --all
  - git-show-branch --all
- See changes back to the common ancestor:
  - gitk master my-branch
  - git-show-branch master my-branch
- Note that master is only one edit from my-branch now (the merge patch-up)

Merging the upstream

- Master is now updated with my-branch changes
- But my-branch is now lagging
- We can merge back the other way:
  - git-checkout my-branch
  - git-merge “merge upstream changes.” HEAD master
- This will succeed
- In fact, it’s a “fast forward”
- This means that the merge-from branch already has all of our change history
- So it’s just adding linear history to the end

Meanwhile, back at the ranch

- The changes can be copied back to the other repo:
  - cd ../git-tutorial
  - git-branch
  - git-pull ../my-git master
- This pulls from the other repo's master into our current (and only) branch
- Now both repos have the same history:
  - gitk --all
  - git-show-branch --all
- The SHA1s are identical
- No "centralized naming" scheme was needed

For further info

- See “Git (software)” in Wikipedia
- And the git homepage http://git.or.cz
- See the Git wiki at http://git.or.cz/gitwiki
- Get on the mailing list
  - Helpful people there
  - You can submit bugs, patches, ideas
  - Find out who else is using git
- Now “git” to it!