Git Pocket Guide

What Is Git?

Git is a tool for tracking changes made to a set of files over time, a task traditionally known as “version control.” Although it is most often used by programmers to coordinate changes to software

source code, and it is especially good at that, you can use Git to track any kind of content at all. Any body of related files evolving over time, which we’ll call a “project,” is a candidate for

using Git. With Git, you can:

* Examine the state of your project at earlier points in time
* Show the differences among various states of the project
* Split the project development into multiple independent

lines, called “branches,” which can evolve separately

* Periodically recombine branches in a process called “merging,”

reconciling the changes made in two or more branches

* Allow many people to work on a project simultaneously,

sharing and combining their work as needed

Git is the technology behind the enormously popular “social coding” website GitHub, which includes many wellknown open source projects.

This is called a “root commit,” and most often, there is only one root commit in a repository—the

initial one created when the repository was started. However, you can introduce multiple root commits if you want; the command git checkout --orphan does this. This incorporates multiple independent

histories into a repository, perhaps in order to collect the contents of previously separate projects (see “Importing Disconnected History” on page 154).

Author versus Committer

Cherry-picking carries forward the author information from the original commit, while adding new committer information. This preserves the identification and origin date of the changes, while indicating that they were applied at another point in the repository at a later date, possibly by a different person.

**$ git log --format=fuller**

Other operations that do this are git rebase and git filterbranch; like git cherry-pick, they too create new commits based on existing ones.

CHAPTER 13

Miscellaneous

Chapiter 14:

Fix the Last Commit I Made

**$ git commit --amend**

Edit the Previous n Commits

**$ git rebase -i HEAD~*n***

Undo My Last n Commits

**$ git reset HEAD~*n***

This removes the last *n* commits of a linear history from the current branch, leaving the corresponding changes in your working files. You can add --hard to make the working tree reflect the

new branch tip, but beware: this will also discard any current uncommitted changes, which you will lose with no recourse

Reuse the Message from an Existing Commit?

**$ git commit --reset-author -C *rev***

Add --edit to edit the message before committing

Reapply an Existing Commit from Another Branch?

**$ git cherry-pick *rev***

List Files with Conflicts when Merging

git status shows these as part of its report, but to just list their names

**$ git diff --name-only --diff-filter=U**

Get a Summary of My Branches?

* List local branches: git branch
* List all branches: git branch –a
* Get a compact summary of local branches and status with respect to their upstream counterparts: git branch –vv
* Get detail about the remote as well: git remote show origin (or other named remote)

Get a Summary of My Working Tree and Index State?

**$ git status**

Add -sb for a more compact listing; see the “Short Format” section of *git-status(1)* on how to interpret this.

Stage All the Current Changes to My Working Files?

**$ git add -A**

This does git add for every changed, new, and deleted file in your working tree. Add --force to include normally ignored files; you might do this when adding a new release to a “vendor branch,” which tracks updates to other projects you obtain by means other than Git (e.g., tarballs).

Show the Changes to My Working Files?

git diff shows unstaged changes; add --stage to see staged changes instead. Add --name-only or --name-status for a more compact listing.

List the Files in a Specific Commit?

**$ git ls-tree -r --name-only *rev***

This listing is restricted to the current directory; add –-fulltree for a complete list

git show *rev* is easier that git diff *rev*~ *rev*, and shows the author, timestamp, commit ID, and message as well. Add -s to suppress the diff and just see the latter information; use --name-status or --stat to summarize the changes. It also works for merge commits, showing conflicts from the merge as with git

log --cc (see “Showing Diffs” on page 142). The default for rev is HEAD.

List All Remotes?

git remote does this; add -v to see the corresponding URLs configured for push and pull (ordinarily the same):

Change the URL for a Remote?

**$ git remote set-url *remote URL***

Remove Old Remote-Tracking Branches?

**$ git remote prune origin**

This removes tracking for remote branches that have been deleted upstream

Find Commits I Made but Lost?

Perhaps after editing history with git rebase -i or git reset, or deleting a branch:

**$ git log –g**

Show the Changes for Each Commit?

git log -p shows the complete patch for each commit it lists, while these options summarize the changes in different ways:

**$ git log --name-status**

**$ git log –stat**

Show the Committer as well as the Author?

**$ git log --format=fuller**