Version control with Git: command summary

Robin Engler

March 13, 2022

Git command summary

Configuring Git

Depending on their scope, Git configurations apply to all Git repositories of a user or only to a specific repository. The main 3 scopes are:

- **Global (user wide):** settings apply to all Git repositories controlled by the user. Stored in `/home/<username>/.gitconfig` (UNIX) or `C:\Users\<username>\.gitconfig` (Windows). To save a setting as part of the global scope, add the `--global` flag to the `git config` command.
- **Local (repo specific):** settings apply only to a specific Git repo. Stored in the `.git/config` file of the repository.
- **System (system wide):** settings apply to all users and all repos on a given machine. This can only be modified by a system administrator.

To show the list of all your Git configurations, along with their scope and the location of the file they are stored-in:

```
git config --list --show-origin
git config --list --show-origin --show-scope  # only works with Git versions >= 2.26
```

User name and email

```
# Set user name/email globally (for all Git projects of current user).
git config --global user.name <user name>
git config --global user.email <email>

# Set a user name/email only for the current Git repo.
git config user.name <user name>
git config user.email <user email>

# Retrieve the value set for user name/email.
git config --get user.name
git config --get user.email
```

Change default editor

The default editor in Git in `vim`. But this can be changed.

```
git config --global core.editor nano   # Set "nano" as default editor.
git config --global core.editor vim    # Set "vim" as default editor.

# Retrieve the value set for default editor.
git config --get core.editor
```

Creating Git command aliases

Create a new git command alias named `<alias name>`, which can then be called with `git <alias name>`.

```
git config --global alias.<alias name> "git command/options to alias"

# Example: create a "git adog" alias
git config --global alias.adog "log --all --decorate --oneline --graph"
```

Existing aliases are stored in the `~/.gitconfig` file. One possibility to see the list of existing aliases is using:
`git config --list | grep ^alias`

First steps commands

Create a new git repository

```
git init                        # Create a new repo from scratch.
git clone <repo URL>            # Clone an existing remote repository.
```

Add and remove content from the index (i.e, staging and unstaging)

```
git add <file or directory>    # Stage a file or directory.
git add -u                      # Stage all already tracked files.
git add --all                   # Stage all files in the project's directory.

git restore --staged <file>    # Unstage changes to a file.
git reset HEAD                  # Unstage all newly staged content from index.
git reset HEAD <file>           # Unstage newly staged content of a specific file from
                                # the index. Same as git restore --staged <file>.

git rm <file>                   # Delete entire file from both working dir and index.
git rm --cached <file>          # Delete entire file from index.

git mv <file> <new location>   # Move, rename or move+rename a file.
```

Discard changes in a file in the working directory and reset it to the state it has in the index. **Warning:** any local changes to the file will be lost, this operation cannot be undone.

```
git restore <file>
```

Ignoring files

To ignore files, add their name or pattern to `.gitignore` (shared) or `.git/info/exclude` (private - for files to ignore only in your local repo). The following are commands add content to these files directly from the command line:

```
echo "file name or pattern" >> .gitignore          # Add file name or pattern to the
                                                    # shared .gitignore list.
echo "file name or pattern" >> .git/info/exclude   # Add file name or pattern to the
                                                    # private ignore list.
```

Creating and amending commits

Make a new commit

```
git commit                                 # Make a new commit (interactive message).
git commit -m "commit message"             # Make a new commit with message "commit message".
git commit -m "commit message" <file>      # Shortcut: add <file> to index + commit.
git commit -am "commit message"            # Shortcut: add all tracked files to index and commit.
```

Amend the latest commit of the current branch

```
git commit --amend             # Re-write latest commit with the currently staged content.
git commit --amend --no-edit   # Same as above but keeping the current commit message.
```

Display file status, show content differences and history

```
git status                  # Show status of files in the working directory.
git show                    # Show last commit on current branch. Same as "git show HEAD"
git show <commit ref>       # Show content of specified commit. E.g. "git show HEAD~1"

git diff                    # Show differences between working tree and Git index.
git diff --cached           # Show differences between Git index and the current HEAD.
git diff <ref 1> <ref 2>    # Show differences between commit reference 1 and 2.
                            # E.g., "git diff HEAD~3 HEAD~1"

git log                                     # Display history of current branch.
git log --oneline                           # Condensed history of current branch.
git log --all --decorate --oneline --graph  # Display entire history of repo.

git ls-files      # List files present in the Git index, i.e. the currently tracked files.
                  # This does not list files that are part of an earlier commit but are
                  # no longer in the index, i.e. files that are part of the repo's history
                  # but no longer in index.
```

Branches

Create and switch branches

```
git branch                      # List local branches.
git branch -a                   # List both local and remote branches.
git branch <new branch>         # Create a branch named <new branch>.

git switch <branch name>        # Switch to a different branch. E.g. "git switch master"
git switch -c <new branch>      # Shortcut: create <new branch> and switch to it.

# For Git versions < 2.23, the switch command is not available. "checkout" can be used instead.
git checkout <branch name>      # Switch to a different branch. E.g. "git checkout master"
git checkout -b <new branch>    # Shortcut: create <new branch> and switch to it.

# Crating a local branch from a remote branch: in this case, a simple switch/checkout command
# is sufficient (no need for the -c/-b options).
git switch <branch name>
git checkout <branch name>
```

Create a new branch rooted at a specific commit.`<commit ref>` can be an absolute hash or a relative reference.

```
git switch -c <new branch> <commit ref>
git checkout -b <new branch> <commit ref>
```

Delete branches

```
git branch -d <branch name>         # Delete local branch <branch name>.
git branch -D <branch name>         # Force-delete local branch with un-merged changes.
git push origin --delete <branch>   # Delete the specified branch on the "origin" remote.
```

Merge, rebase and cherry-pick

- To **merge** branch B into branch A, one must be on branch A.
- To **rebase** branch B onto branch A, one must be on branch B.

```
git merge <branch name>             # Merge <branch name> into the current branch.
git rebase <branch to rebase on>    # Rebase the current branch onto the latest commit of
                                    # <branch to rebase on>
git cherry-pick <commit ref>        # Cherry-pick commit <commit ref> onto the current
                                    # branch or HEAD.
```

Retrieve content from Git repository database

```
git checkout <commit ref> <file>  # Retrieve the specified <file> from commit <commit ref>.
                                  # Warning: this will overwrite the current version of
                                  # <file> in the working tree.

git checkout <commit ref>   # Set entire working tree content to its state as recorded in
                            # <commit ref>. After this command you will be in "detached
                            # HEAD" state, which you can exit again by switching back to
                            # one of your branches.
```

Working with remotes

- The `fetch` command does not modify anything in the working tree, it only retrieves new commits or references (branches, tags) from the remote.
- By default, `fetch`, `pull`, `push` retrieve/push content from/to the **default remote** (generally named `origin`). If you have more than one remote, a non-default remote can be specified.

```
git fetch          # Fetch all new changes from the remote that are not yet in your local repo.
git fetch --prune  # In addition of fetching, delete local references to remote branches.

git pull           # Fetch + merge all changes into the current branch.
git pull --prune   # In addition of pulling, delete local references to remote branches.

git push                        # Push new content on current branch to default remote.
git push -u origin <branch>     # Set origin/<branch> as upstream for the current branch, and push
                                # new content on the current branch to the remote. Only needed the
                                # first time a new branch is pushed.
                                # Note: -u is the short option for "--set-upstream"

# Fetch/pull/push content from a specific remote. Useful if there are more than 1 remote.
git fetch <remote name>
git pull <remote name>
git push <remote name>
```

Associate a remote (in this example, from GitHub) with an existing repo. In the examples below, we name the remote `origin` (the default name used by Git), but a remote can be named anything. * `git remote add`: used to add an online repository as a remote. * `git remote set-url`: used to modify the URL of remote when a remote is already set.

```
git remote add origin https://github.com/<GitHub user name>/<repo name>.git      # Add repo.
git remote set-url origin https://github.com/<GitHub user name>/<repo name>.git  # Update remote URL.
git remote -v   # Show name and URL of all remotes associated to the repo.
```

Relative references to commits

Here the examples are given with `HEAD`, but an absolute hash, a tag or branch name can also be used.

Reference to an ancestor commit.

```
HEAD~     # Parent of current HEAD.
HEAD~2    # Grand-parent of current HEAD.
HEAD~3    # Great grand-parent of current HEAD.
...
HEAD~X    # Xth generation before current HEAD (general form).
```

Reference to the first or second direct parent.

```
HEAD^1    # First direct parent of HEAD.
HEAD^2    # Second direct parent of HEAD.
```

Detached HEAD state

Detached HEAD state is when the HEAD pointer no longer points to a branch, but instead points directly at a commit.

- Detached HEAD mode is entered after running `git checkout <commit ref>`, where `<commit ref>` is a reference to a commit that is *not* a branch.
- To exit detached HEAD state, simply switch back to a regular branch with `git switch <branch>` or `<git checkout branch`.
- Adding commits in a detached HEAD mode can result in them getting lost when moving back to a regular branch. To preserve these commits, a new branch rooted at the current position of the HEAD should be created (`git switch -c <new branch>` or `git checkout -b <new branch>`).

Advanced topics commands

Interactive rebase

Same as a regular rebase but with the possibility to re-order, merge, and delete commits.

```
git rebase -i <commit ref>   # Reference of commit, branch or tag, on which to rebase.
```

Git reset

Moves the **HEAD** pointer to the selected reference (e.g. commit, branch, tag) and, depending on the selected option, also resets the Git index and worktree:

- `--mixed` is the default option (passing no option is the same as `--mixed`).
- using `--hard` **will make you lose any uncommitted change** in your worktree. Use it carefully.

```
git reset --soft  <commit ref>   # --soft: resets HEAD, but not the index nor the worktree.
git reset --mixed <commit ref>   # --mixed: resets HEAD and index, but not the worktree.
git reset --hard  <commit ref>   # --hard: resets HEAD, index and worktree.
```

Examples (frequent use cases):

```
# Clear staging area (git index) from newly added material.
git reset HEAD

# Merge the 2 latest commits on the current branch.
git reset --soft HEAD~2
git commit -m "the last two commits are now merged"

# Reset a merge or another reset:
git reset --hard HEAD@{1}

# Reset a rebase:
git reset --hard <ref of latest commit of the rebased branch before the rebase>

# Reset a branch so that it matches the state of its upstream branch. Note that in this
# example the remote is named "origin" (default name for remotes).
git fetch
git reset --hard origin/<branch name>
```

Git stash

the Git stash allows to temporarily clean the working tree by stashing away uncommitted changes (both staged and unstaged), which can then be restored at a later time.

```
git stash          # Stash all uncommitted changes (staged and unstaged).
git stash pop      # Restore stashed changes on the current HEAD commit.
```

Git reflog

The reflog is a log of all operations (and associated `HEAD` positions) performed in a repository. The `HEAD@{x}` notation can be used to access the position of HEAD relative to the reflog.

```
git reflog      # Print the git reflog.
HEAD@{0}        # Current position of HEAD.
HEAD@{1}        # Position of HEAD one reflog record ago.
HEAD@{2}        # Position of HEAD two reflog records ago.
HEAD@{x}        # Position of HEAD x reflog records ago.
```

Git tags

```
git tag <tag name>                # Create a new lightweight tag (no associated message).
git tag -a <tag name> -m "msg"    # Create a new annotated tag with a tag message.

git tag               # List tags present in the Git repo.
git tag -l "v1.*"     # List only the tags matching a specific pattern, e.g. tags starting
                      # with "v1." in this case.
git tag -n            # Also display tag annotations. For lightweight tags, the commit
                      # message of the commit to which the tag is pointing is shown.

git show <tag name>   # Display changes introduced at the commit referenced by the tag.
                      # For annotated tags, the tagger and tag message are also shown.

git push origin <tag name>   # Push the specified tag to the "origin" remote.
git push --tags              # Push all tags to the default remote.

git tag -d <tag name>                  # Delete tag in local repo.
git push origin --delete <tag name>    # Delete a tag on the "origin" remote.

git checkout <tag name>   # Revert the working tree to the specified tag.
                          # This will enter "detached HEAD" mode.
```

Note: a commit referenced by a tag will not be garbage-collected, even if it is not part of a branch.

Git submodules

Add/register a new submodule to a project. Submodules are added by using the URL of their online repository (they can also be added without an online remote but it's mode complicated). By default the submodule is named after the remote name, but a custom name can also be set.

```
git submodule add <repository URL>                # Add a new submodule to the git repo.
git submodule add <repository URL> <custom name>  # Add a new submodule under a custom name.

git mv <submodule name> <submodule new name>      # Rename a submodule.
```

Clone a repo with submodules.

```
git clone --recurse-submodules <repo URL>   # Clone a repository containing submodules.

# The above is a shortcut for:
git clone <repo URL>
git submodule init                  # Initialize/Activate the specified submodule.
git submodule update --recursive    # Download content for the specified submodule.
```

Work with submodules.

```
git submodule foreach "<command>"         # Execute the specified command in all submodules.
git submodule update --init --recursive   # Update the content of all local submodules by
                                          # performing a pull (fetch + merge) on their
                                          # default branch - "master" by default, but other
                                          # default branches can be specified in .gitmodules.

git push --recurse-submodules=check       # Only make a push on the main repo if all changes
                                          # in the submodules have been pushed.
git push --recurse-submodules=on-demand   # Try to push the main project together with all
                                          # submodules.
git submodule update --remote --merge     # Get the latest versions of your submodules and
                                          # merge them with your changes. You have to
                                          # specify the branch in .gitmodules in case you
                                          # don't want to use the "master" branch.
```

Git LFS

Track and untrack files with Git LFS

```
git lfs track <file name>       # Add file name/pattern to list of LFS-tracked files,
                                # i.e. add them to the .gitattributes file.
git lfs track "*.pattern"       # To track a pattern, it must be quoted.
git lfs track <directory/**>    # To track the content of entire directory (recursively),
                                # the directory name must be followed by "/**".

git lfs untrack <file name or pattern>    # Stop LFS-tracking on file name/pattern. The
                                          # same can be achieved by deleting a line in
                                          # the .gitattributes file.

git lfs ls-files               # List LFS-tracked files associated with the HEAD commit.
git lfs ls-files <commit ref>  # List LFS-tracked files associated with a specific
                               # commit hash or reference.
git lfs ls-files --all         # List all LFS-tracked files.
```

Download LFS-tracked files to local cache

```
git lfs fetch --recent  # Download all LFS-tracked files associated with "recent" commits.
git lfs fetch --all     # Download all LFS-tracked files to the local cache.
```

Clear local cache

To save space on the local machine, LFS-tracked files that are locally cached can be deleted. They will remain available on the remote LFS object store (provided they were pushed at some point).

```
git lfs prune                   # Delete non-needed files from local LFS cache.
git lfs prune --dry-run         # Same as above, but in "test" mode: only show what
                                # would be deleted, without actually deleting it.
git lfs prune --verify-remote   # Double check that LFS-tracked files are really present
                                # on the remote object store before deleting them.
```

Convert an existing repo to Git LFS

Warning: this will rewrite the entire history of the Git repository! It's strongly advised to make a copy of the repo before running this command.

```
git lfs migrate import --include=<file name or pattern> --everything

git lfs checkout   # After a "migrate import" command, the LFS-tracked files in the
                   # working tree have their content replaced by the pointer content.
                   # The "lfs checkout" command restores their original content.
```

Install Git LFS for current user

This must be performed only once for each user account on a given machine.

```
git lfs install
```