Thursday, December 15, 2022 11:11 AM

Git commands you might need

This section lists some of the commands that All developers have needed to use.

Changes and conflicts

Pulling changes

1. If you have local changes, first do:
    - **git stash**
2. **git pull** or **git fetch** & **git merge**
3. If you did stash your local changes, then do:
    - **git stash pop** or **git stash apply**

You can also first commit your changes before pulling. With the proper global options set, Git will pull changes and then attempt to apply your commits on top of those changes (i.e. rebase).
Refer to git-fetch-and-merge for the benefit of using git-fetch & git-merge instead of git-pull

Sharing changes

To quickly expose your local repository for another team member to pull (bypassing the server repository), run the following:

```
cd <git project to share>
git daemon --reuseaddr --base-path=. --export-all --verbose
```

For those familiar with Mercurial, the above is similar to running hg serve.
Team members can then clone:

```
git clone git://<host name of sharing developer>/.git <target directory>
```

Or pull changes:

```
git pull git://<host name of sharing developer>/.git <branch to pull>
```

Undo changes

Undo all uncommitted changes:

```
git reset --hard
```

Un-commit your last commit that you have *not* pushed yet:

```
git reset --soft HEAD~1
```

Notes:

- HEAD~1 is a shorthand for the commit before head. Replace the number with how many commits you want to undo. Alternatively you can refer to the hash that you want to reset to.
- --soft: will undo the commit and preserve the changes in those commits in a "Changes to be committed" state.
- --hard: will undo the commit and lose all the changes in those commits

Undo a git rm:

```
git checkout HEAD <the deleted file>
```

Clean all untracked files

If your source directory contains a lot of untracked files and you want to remove them all, run the following:

```
git clean -df
```

The -d option is for directories. Add -x to also remove ignored files, or -X to remove *only* ignored files.

Resolving conflicts

If you get conflicts while merging, simply edit and fix those files that Git reports as having conflicts.

Then run **git add** on those conflicted files; that would mark them as resolved.

Run **git commit -a** to commit the whole merge.

If you want to select a particular revision of the conflicted file (instead of having to manually merge), you can also do **one** of the following:

```
git checkout HEAD -- <filename> # Replace with HEAD revision
git checkout --ours -- <filename> # Use latest revision from your current local branch
git checkout --theirs -- <filename> # Use latest revision from the branch you merged from
git checkout <branch> -- <filename> # Use latest revision from the specified branch
```

Merge and ignore whitespace

If you are having difficulty performing a merge, or if the resultant merge does not seem correct, you can rollback and try this instead:

```
git merge -s recursive -Xignore-space-change <local branch name>
```

Stashing changes you do not want to commit

Stash normally stashes only changes to tracked files and ignored untracked files. To also stash untracked files:

```
git stash -u
```

To selectively stash changes for a commit:

```
git add <all files that you _want_ to commit>
git stash --keep-index
git commit -m "Your message; no need to do -a for this command"
git pop
```

Stashing changes with a custom description

To associate a particular set of stashed changes with a description

```
git stash save "my description" # "save" is the action that happens when you do a plain "git stash"
git stash list # To see your description
```

Recovering stashed changes that's not on top of the stack

To recover a stash by index

```
git stash list # To see all your stashes
git stash pop stash@{1} # To pop the 2nd set of stashed changes; or you can use "apply" instead
```

Recovering only one file from a stash

```
git checkout stash@{0} -- <file> # Recover the stashed <file> from the topmost stash
```

You can recover files from earlier stashes by changing "stash@{0}" appropriately. Use **git stash show** to display all the available stashes. The left column displays the index (i.e. name) of the stash.

Hiding local changes

```
 git update-index --skip-worktree <file>
 git update-index --no-skip-worktree <file>
```

You can use these commands to hide changes from Git, such as when modifying **ivy.properties** locally.

Branches

List all remote branches only

git branch -r

List all local and remote branches

git branch -a

Create a branch

git branch <new branch name>

Create a branch and immediately switch to it:

git checkout -b <new branch name>

Checkout a branch and track it:

git checkout -t origin/<remote branch name>

Rename a remote branch

git push origin <old_branch>:<new_branch> // create a new branch based on the old

git checkout <new_branch> // switch branch if still on the old branch

git push origin :<old_branch> // delete the old branch

Delete a remote branch

git push origin :<branch_to_delete> // delete remote branch - IMPORTANT: Make sure you know what you're doing!

git branch -d <branch_to_delete> // remove local copy of the branch

Set upstream branch tracking if a branch was initially pushed without tracking

git branch --set-upstream-to origin/<remote branch name>

git branch --set-upstream <local branch name> origin/<remote branch name> // older versions of Git

Set upstream branch tracking on initial push

git push -u

Push a local branch to a remote branch of a different name

git checkout <local branch name>

git push origin <remote branch name>

Delete all branches where the remote branch was already deleted

This is a command for cleaning up your local list of branches against the remote repository.

git remote prune origin

Tags

List all tags

git tag -l

Create a tag
git tag -a <tag identifier> -m <tag description>

Example:

git tag -a v0.1 -m "version 1.0 release 1"

git push --tags

View tag commit details
git show v0.1

Checkout a tag
git fetch

git checkout tags/<tag identifier> // This will put you in a no-branch state

Find tags that contain a specific commit
git tag --contains <SHA>

Delete a remote tag
git tag -d <tag identifier>

git push origin :refs/tags/<tag identifier> // IMPORTANT: Make sure you know what you're doing!

Logs and information

Show detailed log history as a graph
git log --oneline --graph --decorate --all

Show verbose information on tracked remote repositories
git remote -v

Amend an incorrect log message
git commit --amend -m "your new message"

Important: This command only works on the tip, not an older revision. For more info, see this. Also, do **not** modify commit logs once you have pushed the commit, because if someone else has pulled your changes, rewriting the log is rewriting history, causing merge conflicts.

Note: This option can also be used to add additional changes to a previous commit by performing:

git commit --amend -a

Find when a file was deleted
Sometimes, it's useful to figure out when a file was removed or renamed.

  git log --all --full-history -- <path-to-file>   (path can be relative or absolute)

  git log --all --full-history -- **/filename.*   (if you don't know the path, you can use file cards)

This will show all the commits that touched the named file. After that:

  git show <SHA> -- <path-to-file>   (this will show you the file)

  git checkout <SHA>^ -- <path-to-file>   (this will restore the file to your working directory - note the '^' symbol, which indicates "1 commit earlier", i.e. before the file got deleted)

Others

Disable automatic git garbage collection

Some operations, particularly git merge, may sometimes trigger a garbage collection. To disable it, run:

```
git config gc.auto 0
```

If disabled, remember to manually run garbage collection once in a while, to prune dangling objects and commits; both to save space and to speed up git.

```
git gc --aggressive
```

Troubleshooting

Too many files, skipping inexact rename detection

A warning like thie following appeared when attempting a merge:

```
warning: too many files (created: 1669 deleted: 632), skipping inexact rename detection
```

This means that in order to complete faster, Git skipped rename detection since the detection is O(n^2). This happens when you're merging two branches that has a large number of file differences.

To increase the rename detection limit, use the following global configuration:

```
git config --global diff.renamelimit 2400
```

The value of 2500 was used because the warning message mentioned 1669 + 632 = 2301 files; meaning at the limit needs to be more than 2301.