***.git subdirectory***

Under the new Git repository directory, a .git subdirectory at /Users/mike/GitIn-PracticeRedux/.git/ (for example) is created with various files and directories under it.

running the find command.

$ find .git



Git is a version control system built on top of an *object store*. Git creates and stores a collection of objects when you commit. The object store is stored inside the Git *repository*. In figure 1.3, you can see the main Git objects we’re concerned with: *commits*, *blobs*, and *trees*. There’s also a *tag* object, but don’t worry about tags until they’re introduced in technique 36. Figure 1.2 showed an example of a commit object and how it stores

metadata and referenced file contents. The file-contents reference is actually a reference to a *tree object*. A tree object stores a reference to all the *blob objects* at a particular point in time and other tree objects if there are any subdirectories. A blob object stores the contents of a particular version of a particular single file in the Git repository.

Add a remote repository to the current repository

$ git remote add origin https://github.com/XXXXXX

You can verify that this remote has been successfully created by running

$ git remote -v

Pushing changes to a remote repository

$ git push --set-upstream origin master

By passing this option, you tell Git that you want the local master branch you’ve just pushed to *track* the origin remote’s branch master. The master branch on the origin remote (which is often abbreviated origin/master) is now known as the *tracking branch* (or *upstream*) for your local master branch.

The git push --set-upstream (or -u) flag and explicit specification of origin and master are only required the first time you push to create a remote branch (without

them, some versions of Git may output fatal: The current branch master has no upstream branch.). After that, a git push with no arguments will default to running the equivalent of git push origin master. This is set up by default by git clone when you clone a repository.

git push can take an --all flag, which pushes all branches and tags (introduced later

in technique 36) at once. Be careful when doing this: you may push some branches with work in progress.

git push can take a --force flag, which disables some checks on the remote repository to allow rewriting of history

A *tracking branch* is the default push or fetch location for a branch. This means in future you can run git push with no arguments on this branch, and it will do the same thing as running git push origin master: push the current branch to the origin remote’s master branch.

git diff origin/master shows the differences between the current working tree state and the origin remote’s master branch

git pull can take a --rebase flag that performs a rebase rather than a merge.

I prefer to use git fetch over git pull. This means I can continue to fetch regularly in the background and only include these changes in my local branches when it’s convenient and using the method I find most appropriate, which may be merging or rebasing (or resetting, which you will see later in technique 42). Additionally, I sometimes work in situations where I have no internet connection (such as on planes),

and using git fetch is superior in these cases; it can fetch changes without requiring any human interaction in the case of a merge conflict, for example.

git branch can take a --track flag, which, combined with a start point, sets the upstream for the branch (similar to git push --set-upstream but without pushing

anything remotely yet).

Make sure you’ve committed any changes on the current branch before checking out a new branch. If you don’t do this, git checkout will refuse to check out the new branch if there are changes in that branch to a file with uncommitted changes. If you wish to overwrite these uncommitted changes anyway, you can force this with git checkout --force. Another solution is git stash, which allows temporary storage of changes and will be covered later in technique 23.

The --recurse-submodules (or --recursive) flag initializes all the Git submodules in the repository. This will be covered more later in technique 54.

git pull can take a --rebase flag that performs a rebase rather than a merge

***Technique 11 Creating a new local branch from the current branch: git branch***

git branch can take a second argument with the *start point* for the branch. This defaults to the current branch you’re on; for example, git branch chapter-two is the equivalent of git branch chapter-two master if you’re already on the master branch. This can be used to create branches from previous commits, which is sometimes useful if, say, the current master branch state has broken unit tests that you need to be working.

git branch can take a --track flag, which, combined with a start point, sets the upstream for the branch (similar to git push --set-upstream but without pushing anything remotely yet).

#### Technique 17 renaming or moving afile:git mv

Moving and renaming files in version control systems rather than deleting and recreating them is done to preserve their history. Git auomatically detect that the file was moved and git mv isn’t necessary. Despite this handy feature it is a good practice to use git mv.

If the filename you move to already exits , you ll need to use the git mv –f

#### Technique 18 removing a file git rm

Removing files from version control requires not just performing the filesystem operation as usual but also notifying Git.

If git add fails, use git add –f (your file is .gitignore file)

$ git rm –r remove the directory and all the unignored files and subdirectories within it

If a file has uncommitted changes , you need to use $ git rm –f

If you want to see a simulated run of git rm without actually removing the requested file, you can use git rm -n (or --dry-run). This will print the output of the command as if it were running normally and indicate success or failure, but without removing the file.

To remove a directory and all the unignored files and subdirectories within it, you need to use git rm -r (where the -r stands for *recursive*). When run, this deletes the directory and all unignored files under it. This combines well with --dry-run if you want to see what would be removed before removing it.

### Technique 19 resetting files to the last commit

The --hard argument reset the both index staging area and the working directory to the state of the previous commit on this branch. If run without an argument, it defaults to git reset –mixed which resets the index staging area but not the contents of the working directory.In short , git reset –mixed only undoes the git add but git reset –hard undoes git add and all file modifications.

### Technique 20 Deleting untraked files git clean

You can view the files that currently tracked by running git ls-files. You can run git ls-files –o to show the currently untracked files.

$ git clean –f

To preview what will be removed use the argument n

$ git clean –n

To remove untracked directories as well as untracked files you can use the –d parameter

git clean requires the --force argument because this command is potentially dangerous

To make git clean a bit safer, you can preview what will be removed before doing

so by using git clean -n (or --dry-run). This behaves like git rm --dry-run in that it

prints the output of the removals that would be performed but doesn’t actually do so.

To remove untracked directories as well as untracked files, you can use the -d

(“directory”) parameter.

==================================================================

### Technique 23 Temporarily stashing some changes

git stash save creates a temporary commit with a prepopulated commit message and

then returns your current branch to the state before the temporary commit was made.

$ git stash save

Creates a temporary commit with a prepopulated commit message. Save argument isn’t needed

git stash stashes your changes regardless of whether they’ve been added to the index staging area by git add.

If git stash is run with no save argument, it performs the same operation; the save argument isn’t needed. I’ve used it in the examples because it’s more explicit and easier to remember.

$git stash list

You can see all the stashes that have been made

$ git diff stash@{0}

Will show you the difference between the working directory and the contents of that statsh

$ git stash pop

When running git stash pop, the top stash on the stack is applied to the working directory and removed from the stack

$ git stash apply

If you wish to apply an item from the stack militple times, you can instead use git stash apply

$ git stash clear

Clearing stashed changes

***Technique 26 Assuming files are unchanged***

Sometimes you may wish to make changes to files but have Git ignore the specific changes you’ve made so that operations such as git stash and git diff ignore these changes.

$git update-index --assume-unchanged 01-IntroducingGitInPractice.asciidoc

When you run git update-index --assume-unchanged, Git sets a special flag on the file to indicate that it shouldn’t be checked for any changes. This can be useful to temporarily ignore changes made to a particular file when looking at git status or git diff, but also to tell Git to avoid checking a file that is particularly huge and/or slow to read.

git update-index --assume-unchanged takes only files as arguments, rather than directories. If you assume multiple files are unchanged, you need to specify them as multiple arguments

***Technique 27 Listing assumed-unchanged files***

Run git ls-files -v. The output should resemble the following

# git ls-files -v

H .gitignore

h 01-IntroducingGitInPractice.asciidoc

Shows that committed files are indicated by an uppercase H at the beginning of the line.

Shows that an assumed-unchanged file is indicated by a lowercase h tag.

Rather than reading through the output for this command, you can instead run git ls-files -v | grep '^[hsmrck?]' | cut -c 3-. This uses Unix pipes, where the output of each command is passed into the next and modified.

grep '^[hsmrck?]' filters the output filenames to show only those that begin with any of the lowercase hsmrck? characters (the valid prefixes output by git ls-files).

It’s not important to understand the meanings of any prefixes other than H and h, but you can read more about them by running git ls-files --help.

cut -c 3- filters the first two characters of each of the output lines: h followed by a space, in the example.

With these combined, the output should resemble the following.

# git ls-files -v | grep '^[hsmrck?]' | cut -c 3-

***Technique 28 Stopping assuming files are unchanged***

so I had to make Git stop ignoring this particular change before I could make a new commit

Run git update-index --no-assume-unchanged 01-IntroducingGitInPractice.asciidoc.

You can verify that Git has stopped assuming there were no changes made to 01-IntroducingGitInPractice.asciidoc by running git ls-files -v | grep 01-Introducing-GitInPractice.asciidoc. The output should resemble the following.

# git ls-files -v | grep 01-IntroducingGitInPractice.asciidoc

H 01-IntroducingGitInPractice.asciidoc

Once you tell Git to stop ignoring changes made to a particular file, all commands such as git add and git diff will start behaving normally on this file again.

*History visualization*

When working with a Git repository on large, long-running software projects, you’ll sometimes want to dig through the history to identify old versions of code, work out why and by whom changes were made, or analyze the changes to identify why a bug is occurring. You can do this to a limited extent using the commands you’ve already learned (git log and git diff) and extend this with two more we’ll cover in this chapter: git blame and git bisect.

the git log command has various flags and arguments that you can use to filter which commits are shown in its output

$git log --author "Mike McQuaid" --after "Nov 10 2013" --grep 'file\.'

The arguments provided to the log command indicate the following:

**Technique 33 Always creating a merge commit**

Force the creation of a merge commit it is useful for history visualization for this feature merge to be more explicit

$ git merge –no-ff branch1

You have merge branch1 into the master branch and forced a merge commit to be created. But on larger features, this explicit indication of branches can aid history visualization

git merge can also take a --ff-only flag, which does the opposite of no-off. It ensures that a merge commit is never created. If the merge can only be made with a merge commit, the merge isn’t performed.

Merge strategies

A merge stra

Git cheat sheet

$ git diff

Shows file differences not yet staged

$ git diff --staged

Shows file differences between staging and the last file version

$ git reset file

Unstages the file but preserve its contents

$ git rm file

Deletes the file from the working directory and stages the deletion

$ git rm –cached file

Removes the file from version control but preserves the file locally

$