***.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 22 Deleting ignored files***

When files have been successfully ignored by the addition of a .gitignore file, you’ll

sometimes want to delete them all.

You wish to delete all ignored files from a Git working directory

# git clean --force –X

The -X argument specifies that git clean should remove *only* ignored files from the

working directory. If you wish to remove ignored files *and* all the untracked files (as

git clean --force would do), you can instead use git clean -x (note that the -x is

lowercase rather than uppercase).

The specified arguments can be combined with the others discussed in

technique 20. For example, git clean -xdf removes all untracked or ignored files

(-x) and directories (-d) from a working directory. This removes all files and directories

for a Git repository that weren’t previously committed. Take care when running

this; there will be no prompt, and all the files will be quickly deleted.

Often git clean -xdf is run after git reset --hard; this means you’ll have to reset

all files to their last-committed state and remove all uncommitted files. This gets you a

clean working directory: no added files or changes to any of those files.

### Technique 23 Temporarily stashing some changes

Instead you can *stash* your uncommitted changes to store them temporarily and then be able to change

branches, pull changes, and so on without needing to worry about these changes getting

in the way.

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

You can see all the stashes that have been made by running git stash list.

$ 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. This applies the stash to the working tree as git stash pop does but keeps the top stack stash on the stack so it can be run again to reapply.

$ 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:

--author specifies a regular expression that matches the contents of the author. In the previous case, it was searching for the author string Mike McQuaid

--after (or --since) specifies that the only commits shown should be those that were made after the specified date. These dates can be in any format that Git recognizes, such as Nov 10 2013, 2014-01-30, midnight, or yesterday.

--grep specifies a regular expression that matches the contents of the commit message. file\. was used rather than file. to escape the . character.

git log can take the following arguments:

A --max-count (or -n) argument to limit the number of commits shown in the log output. I tend to use this often when I only care about something in, say, the last 10 commits and don’t want to scroll through more output than that.

A --reverse argument to show the commits in ascending chronological order (oldest commit first).

A --before (or --until) argument, which will only show commits before the given date. This is the reverse of --after.

A --merges flag (or --min-parents=2), which will only show *merge commits*— commits that have at least two parents. If you adopt a branch-heavy workflow with Git, this will be useful in identifying which branches were merged and when.

git show is a command similar to git log, but it shows a single commit.

# git show HEAD^

* shows all the same information expected in git log output, but it only ever shows a single commit
* shows the changes made in that commit. It’s the equivalent of typing git diff HEAD^^..HEAD^—the difference between the previous commit and the one before it.

The git show HEAD^ output is equivalent to git log --max-count=1 --patch HEAD^.

***Technique 30 Listing commits with different formatting***

Fortunately, git log has some powerful formatting features with varied, sensible supplied options that give you the ability to completely customize the output to meet your needs.

You want to list the last two commits in an email format with the oldest displayed first.

# git log --format=email --reverse --max-count 2

If you specify the --patch (or -p) flag to git log, you can also format the diff output by specifying flags for git diff. Recall the discussion of word diffs in section 1.7. git log --patch --word-diff shows the word diff (rather than the unified diff) for each log entry.

git log can take a --date flag, which takes various parameters to display the output dates in different formats. For example, --date=relative displays all dates relative to the current date; 6 weeks ago and --date-short display only the date, such as 2013-11-28. iso (or iso8601), rfc (or rfc2822), raw, local, and default formats are also available, but I won’t detail them in this book.

The --format (or --pretty) flag can take various parameters, such as email, which you’ve seen in this example; medium, which is the default if no format was specified; and oneline, short, full, fuller, and raw. I won’t show every format in this book, but please compare and contrast them on your local machine. Different formats are better used in different situations depending on how much of their displayed information you care about at that time.

You may have noticed that the *full* output contains details about an author and a committer, and the *fuller* output additionally contains details of the author date and commit date.

# git log --format=fuller

If none of the git log output formats meets your needs, you can create your own custom format using a *format string*. The format string uses placeholders to fill in various attributes per commit.

# git log --format="%ar %an did: %s"

Here I’ve specified the format string with %ar %an did: %s. In this format string

* %ar is the relative format date on which the commit was authored.
* %an is the name of the author of the commit.
* %s is the commit message subject (the first line).

git shortlog displays commits grouped by author with one commit subject per line.

# git shortlog HEAD~6..HEAD

***Technique 31 Showing who last changed each line of a file: git blame***

As long as the file is stored in a Git repository, it’s easy to query who made a change as well as when and why (given a good commit message was used) a certain change was made. Instead, let’s see how to use a command designed specifically for this use case: git blame.

# git blame --date=short 01-IntroducingGitInPractice.asciidoc

* --date=short is used to display only the date (not the time). This accepts the same formats as the --date flag for git log. It was used in the preceding listing to make it more readable, because git blame lines tend to be very long.

You can use the -w flag to ignore whitespace changes when finding where the line changes came from. Sometimes people fix stuff like indentation or whitespace on a line, which makes no functional difference to the code in most programming languages.

The -s flag hides the author name and date in the output (and takes precedence over --show-email/-e). This can be useful for displaying a more concise output format and looking up this information by passing the SHA-1 to git show at a later point.

If the -L flag is specified and followed with a line range—for example, -L 40,60— then only the lines in that range are shown. This can be useful if you know already what subset of the file you care about and don’t want to have to search through it again in the git blame output.

*Advanced branching*

***Technique 33 Merging branches and 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

This explicit indication of a merge through the creation of a merge commit can show all the metadata present in any other commit, such as who performed the merge, when, and why. In software projects, merging a new

feature is usually done by merging a branch, and it’s useful for regression testing

and history visualization for this feature merge to be more explicit

Recall that a *fast-forward merge* means the incoming branch has the current branch as an ancestor

$ 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

You’ve now merged the chapter-spacing branch 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.

git merge can also take a --ff-only flag, which does the opposite of no-ff: it ensures that a merge commit is never created. If the merge can only be made with a merge commit (there are conflicts that need to be resolved and marked in a merge commit), the merge isn’t performed.

***5.1 Merge strategies***

A *merge strategy* is an algorithm that Git uses to decide how to perform a merge. The previous merge output stated that it was using the recursive merge strategy.

You can select a strategy by passing the --strategy (or -s) flag to git merge, followed by the name of the strategy. For example, to select the default, recursive strategy, you could also call git merge --strategy=recursive.

Certain strategies (such as recursive) can also take options by passing the --strategy-option (or -X) flag. For example, to set the patience diff option for the recursive strategy, you’d call git merge --strategy-option=patience.

The following are some useful merge strategies:

* recursive—Merges one branch into another and automatically detects renames. This strategy is the default if you try to merge a single branch into another.
* octopus—Merges multiple branches at once but fails on a merge conflict. This strategy is the default if you try to merge two or more branches into another by running a command like git merge branch1 branch2 branch3. You’ll never set it explicitly, but it’s worth remembering that you can’t manually resolve merge conflicts if you merge multiple branches at once. In my experience, this means it’s worth always merging branches one at a time.
* ours—Performs a normal merge but ignores all the changes from the incoming branch. This means the resulting tree is the same as it was before the merge. This can be useful when you wish to merge a branch and indicate this in the history without wanting to include any of its changes. For example, you could use this to merge the results of a failed experiment and then delete the experimental branch afterward. In this case, the experiment would remain in the history without being in the current code.
* subtree—A modified version of the recursive strategy that detects whether the tree structures are at different levels and adjusts them if needed. For example,if one branch had all the files in the directory A/B/C and the other had all the same files in the directory A/B, then the subtree strategy would handle this case; A/B/C/README.md and A/B/README.md could be merged despite their different tree locations.

Some useful merge strategy options for a recursive merge (currently the only strategy with options) are as follows:

* ours—Automatically solves any merge conflicts by always selecting the previous version from the current branch (instead of the version from the incoming branch).
* theirs—The reverse of ours. This option automatically solves any merge conflicts by always selecting the version from the incoming branch (instead of the previous version from the current branch).
* patience—Uses a slightly more expensive git diff algorithm to try to decrease the chance of a merge conflict.
* ignore-all-space—Ignores whitespace when selecting which version should be chosen in case of a merge conflict. If the incoming branch has made only whitespace changes to a line, the change is ignored. If the current branch has introduced whitespace changes but the incoming branch has made nonwhitespace changes, then that version is used.

Neither of these lists is exhaustive, but these are the strategies and options I’ve found are most commonly used. You can examine all the merge strategies and options by running git help merge.

***Technique 34 Resolving a merge conflict***

you can use the git show command with a branchname:filename argument to show the current state of the 01-IntroducingGitInPractice.asciidoc file on each branch.

# git show master:01-IntroducingGitInPractice.asciidoc

In this diff there are two columns (rather than the usual one) allocated for - and + markers. This is because whereas a normal diff indicates insertions into and deletions from a file, this *merge diff* shows file insertions and deletions and the branch in which they were inserted or removed. For example, in the preceding listing, the first column indicates a line inserted into or deleted from the incoming branch (separatefiles),

and the second column indicates a line inserted into or deleted from the current branch (master). Don’t worry about identifying which column is which; it’s not very important but provides more context for changes.

***Technique 35 Resolving each merge conflict only once: git rerere***

Run git config --global --add rerere.enabled 1. There will be no output.

You’ve enabled git rerere to automatically save and retrieve merge-conflict resolutions in all repositories

You don’t need to run git rerere manually for it to store and retrieve merge conflicts.

After enabling git rerere, you’ll see some slightly different output the next

time you run git commit after resolving a merge conflict.

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

$