***.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***

Git has a command named git rerere (which stands for “Reuse Recorded Resolution”)

that integrates with the normal git merge workflow to record the resolution of

merge conflicts for later replay.

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.

Sometimes you may want to make git rerere forget a resolution for a particular file because you resolved it incorrectly. In this case, you can use git rerere with a path to forget any resolutions for that file or directory. For example, to forget the resolution on 01-IntroducingGitInPractice.asciidoc, you’d run git

rerere forget 01-IntroducingGitInPractice.asciidoc. There will be no output

***Technique 36 Creating a tag: git tag***

tag is another *ref* (or pointer) for a single commit. Tags differ from branches in that they’re (usually) permanent. Rather than pointing to the work in progress on a feature, they’re generally used to describe a version of a software project

Run git tag v0.1

You’ve created a v0.1 tag in the GitInPracticeRedux repository.

All tags in the current repository (not just the current branch) are listed by git tag.

Note that, unlike branches, when new commits are made on the master branch the v0.1 tag won’t change. This is why tags are useful for versioning; they can record the significance of a particular commit without changing it.

git tag can take various flags:

* The --list (or -l) flag lists all the tags that match a given pattern. For example, the tag v0.1 will be matched and listed by git tag list --v0.\*.
* The --force (or -f) flag updates a tag to point to the new commit. This is usefulfor occasions when you realize you’ve tagged the wrong commit.
* The --delete (or -d) flag deletes a tag. This is useful if you’ve created a tag with the wrong name rather than just pointing to the wrong commi

Run git push to push the master branch to origin/master. You may notice that it doesn’t push any of the tags. After you’ve tagged a version and verified that it’s pointing to the correct commit and has the correct name, you can push it using git push -tags. This pushes all the tags you’ve created in the local repository to the remote repository. These tags will then be fetched by anyone using git fetch on the same repository in future.

HOW CAN YOU UPDATE REMOTE TAGS? You’ve seen that by using git tag --delete or git tag --force, it’s possible to delete or modify tags locally. It’s also possible to push these changes to the remote repository with git push --tags --force, but doing so is not advised. If other users of the repository want to have their tags updated, they will need to delete them locally and refetch. This is intentionally cumbersome, because Git intends tags to be static and so doesn’t change them locally without users’ explicit intervention.

If you realize you’ve tagged the wrong commit and wish to update it after pushing, it’s generally a better idea to tag a new version and push that instead. This complexity is why git push requires the --tags argument to push tags.

***Technique 37 Generating a version number based on previous tags: git describe***

***Technique 37 Generating a version number based on previous tags: git describe***

You want to generate a version number for a software project based on existing tags in the repository

# git describe --tags

v0.1-1-g0a5e328

Shows the version generated from the state based on existing tags. It’s hyphenated into three parts

* v0.1 is the most recent tag on the current branch.
* 1 indicates that one commit has been made since the most recent tag (v0.1) on the current branch.
* g0a5e328 is the current commit SHA-1 prepended with a g (which stands for git).

If you wish to use a longer or shorter SHA-1 ref, you can configure this using the --abbrev flag. For example, git describe --tags --abbrev=5 outputs v0.1-1-g0a5e3. Note that if you use very low values (such as --abbrev=1), git describe may use more than you’ve requested if it requires more to uniquely identify a commit.

***Technique 38 Adding a single commit to the current branch:git cherry-pick***

git cherry-pick can take various flags

* If the --edit flag is passed to git cherry-pick, it prompts you for a commit message before committing.
* If you’re cherry-picking from a public branch (one you’ll push remotely) to another public branch, you can use the -x flag to append a line to the cherrypicked commit’s message saying which commit this change was picked from. For example, if this flag had been used in the last example, the commit message would have had

(cherry picked from commit dfe2377f00bb58b0f4ba5200b8f4299d0bfeeb5d)

appended to it.

* When you want to indicate in the commit message which person cherry-picked a particular change more explicitly than the Committer metadata set by default, you can use the --signoff (or -s) flag. This appends a Signed-off-by line to the end of the commit message. For example, if this flag had been used in the last example, the commit message would have had Signed-off-by: Mike McQuaid <mike@mikemcquaid.com> appended to it.
* If there’s a merge conflict on a cherry-pick, you need to resolve it in a fashion similar to a git merge (or in the same fashion as git rebase, which you’ll see later in technique 43). This involves resolving the conflict and running git add, but then using git cherry-pick --continue instead of git commit to commit the changes. If you want to abort the current cherry-pick, perhaps because you’ve realized the merge conflict is too complex, you can do this using git cherry-pick --abort.

***Technique 39 Reverting a previous commit: git revert***

# git revert c18c9ef

git revert can take a --signoff (or -s) flag, which behaves similarly to that of git cherry-pick; it appends a Signed-off-by line to the end of the commit message. For example, if this flag had been used in the last example, the commit message would have had Signed-off-by: Mike McQuaid <mike@mikemcquaid.com> appended to it.

***Technique 40 Listing what branches contain a commit: git cherry***

You wish to see what commits remain unmerged to the master branch from the v0.1-release branch

# git cherry --verbose master

- dfe2377f00bb58b0f4ba5200b8f4299d0bfeeb5d Advanced practice technique.

+ a8200e1407d49e37baad47da04c0981f43d7c7ff Add release preface

* is prefixed with - and shows a commit that has been already included into the master branch.
* is prefixed with + and shows a commit that hasn’t yet been included into the master branch.

If you omit the --verbose (or -v) flag from git cherry, it shows just the -/+ and the full SHA-1 but not the commit subject

*Rewriting history and disaster recovery*

***Technique 41 Listing all changes including history rewrites: git reflog***

Git’s *reflog* (or *reference log*) is updated whenever a commit pointer is updated (like a HEAD pointer or branch pointer).

Like git log, git reflog can be passed a ref as the final argument. If this isn’t specified, it defaults to HEAD. For example, you can view how the master branch has changed over time by using git reflog master.

The main rule to avoid data loss therefore is *commit early and commit often*.

Commit whenever you’ve written anything useful that you don’t want to lose, and then rewrite

your history later into small, readable commits.

If things ever go really badly and you suffer disk corruption with important but

unpushed commits in your repository, you can run the git fsck tool. It verifies the

integrity of the repository and prints out any missing or corrupt objects that it finds.

***Technique 42 Resetting a branch to a previous commit: git reset***

But what if the commit hasn’t been pushed yet? In this case you can use a command you first saw in technique 19: git reset.

When you used git reset previously, you used it either with no arguments (which implies --mixed) or with --hard. Remember, --hard resets the index and the working directory, and --mixed resets the index but not the working directory. In short, --hard discards any uncommitted work, whereas --mixed unstages it (effectively reversing a git add).

git reset can also take a ref as an argument. Rather than just resetting to the last commit, this allows you to reset a branch to any other commit in the repository.

WHAT’S THE DIFFERENCE BETWEEN GIT RESET AND GIT CHECKOUT? git reset modifies the current branch pointer so it points to another commit. Git checkout modifies the HEAD pointer so it points to another branch (or, rarely, commit). If you’re on the master branch, git reset --hard v0.1-release

sets the master branch to point to the top of the v0.1-release branch, whereas git checkout v0.1-release changes the current branch (the HEAD pointer) to point to the v0.1-release branch.

git reset can also take a list of paths as the last arguments to the command. These can be separated using -- between the ref and the list of paths. The -- is optional but makes more explicit the separation between the ref and paths. After all, it’s possible (if unlikely) that you could have a file and path with the same name. For example, to reset the contents of the 00-Preface.asciidoc file to the previous commit, you’d run

git reset HEAD^ -- 00-Preface.asciidoc.

In addition to --hard and --mixed, git reset can also take a --soft argument. The --soft argument can be compared to --mixed and --hard, as shown earlier. Whereas --hard resets the index staging area and working tree (discards all the changes) and --mixed resets the staging area but not the working tree (leaves the changes but removes them from the staging area), --soft resets neither the staging area nor the working tree but just changes the HEAD pointer to point to the previous commit. This means if you run git commit (with no other arguments) after a git reset --soft HEAD^, the contents of the index staging area (and therefore the commit) will be the same as the commit that was just reset.

You can also perform a combined reset and commit operation to modify the previous commit using git commit --amend. git commit --amend resets to the previous commit and then creates a new commit with the same commit message as the commit that was just reset. It uses git reset --soft HEAD^ and then runs git commit --reedit-message with the previous (now reset) commit as an argument. This means it adds anything you have currently added to the index staging area to the changes from the previous commit and prompts for a new commit message. I most commonly use this to adjust the previous commit message if I realize I’ve made a typo or omitted useful information.

***Technique 43 Rebasing commits on top of another branch: git rebase***

If you wanted to undo this operation, you could run git branch --force inspiration 88e8b4b to reset the inspiration branch pointer to point back to the existing commit, essentially undoing the rebase.

***Technique 44 Rebasing commits interactively: git rebase - - interactive***

Run git commit --allow-empty --message "Empty commit" to create an empty commit. The output should resemble the following

***Technique 46 Rewriting history on a remote branch: git push - -force***

***Technique 47 Rewriting the entire history of a branch: git filter-branch***

Git provides a tool called git filter-branch for these cases. It iterates through the entire history of a branch and lets you rewrite every commit as it does so. This can be used to rewrite all the commits in an entire repository.

You wish to remove all references to the file 00-Preface.asciidoc on the master branch

Run

$ git filter-branch --prune-empty --index-filter "git rm –-cached --ignore-unmatch 00-Preface.asciidoc" master

(git filter-branch) takes the following flags:

* The --prune-empty flag discards any now-empty commits (those that only changed 00-Preface.asciidoc) because they’re no longer needed.
* The --index-filter flag rewrites the index of each commit, given a command to run on each commit.
* The master branch argument specifies which branch should be traversed and rewritten.

is passed as a string (surrounded by ") and is the command that’s run by git filterbranch on each commit. Here git rm takes the following flags:

* The --cached flag removes the file from the index. Because this is an index filter, this is all that’s necessary to remove the file from the commit; it doesn’t have to be removed from disk (which would be slower).
* The --ignore-unmatch flag specifies that the command should be successful even if the specified file (00-Preface.asciidoc) doesn’t exist on the current commit.

All commits that referenced the file have been changed, and all those that only changed this file have been pruned, because they were empty

filter-branch is a relatively niche command that is used only in fairly dramatic circumstances such as killing a project but extracting parts of it into a library, filtering history before open-sourcing a repository, or removing confidential information that was accidentally committed over a long period of time. git filter-branch can take the following arguments:

* --all runs on every branch rather than just the named one. This can be used to rewrite entire repositories rather than just single branches.
* --force (or -f) performs a second filter-branch on the same branch in a repository (which overwrites the backup). Otherwise the following message is output:

Cannot create a new backup.

A previous backup already exists in refs/original/

Force overwriting the backup with –f

* --env-filter lets you change environment variables to change the metadata for each commit. For example, you could set GIT\_AUTHOR\_EMAIL for each commit to change the email for every commit or change it conditionally to change it for a particular author.
* --tree-filter lets you rewrite the contents of the working directory tree. It checks out every revision and then allows modification of it. The command in this example could instead be done with a tree filter (git filter-branch --tree-filter "rm 00-Preface.asciidoc" master), but that would check out and delete the files from disk every time, which would be slower than the--index-filter used here.
* --msg-filter lets you rewrite commit messages. This can be useful in removing confidential information (or swear words) from commit messages.
* --subdirectory-filter filters the history to those commits that touch a particular subdirectory. It also makes that subdirectory the new project root. This can be useful when splitting a large repository into multiple smaller repositories based on existing subdirectories.
* --parent-filter changes commit parents. --commit-filter can vary commit commands, and --tag-name-filter modifies tag names, but these tend to be less widely used.

In this chapter you learned the following:

* How to use git reflog to reference logs and see how HEAD and branch pointers have changed over time
* How to use git reset to reset a branch to point to any other commit
* How to use git rebase to reparent commits, the --interactive flag to rearrange history on demand, and git pull --rebase to avoid merge conflicts when pulling changes
* How to use git push origin +branchname to force-push commits and rewrite history on remote branches
* How to use git filter-branch to rewrite the entire history of one or more branches
* How to avoid disaster by committing regularly

*7 Personalizing Git*

***Technique 48 Setting the configuration for all repositories***

# git config --global user.name

When you run git config --global user.name "Mike McQuaid", a file named .gitconfig

is created (or modified if it exists) in your $HOME directory.

The $HOME directory is often signified with the tilde (~) character, as it is in the rest of this chapter. If your username is mike, the $HOME directory typically resides in C:\Users\mike on Windows,

# cat ~/.gitconfig

The git config command takes arguments in the format git config --global section.key value. If you ran this command again with the same section.key but a different value, it would alter the current value rather than creating a new line.

This ~/.gitconfig file is used to set your preferred configuration settings to be shared among all your repositories on your current machine.

Options can also be unset by using the unset flag. For example, to unset the git rerere setting, you would run

# git config --global --unset rerere.enabled

But you can use four different flags to affect the location of the configuration file that’s used:

 --global—Uses the ~/.gitconfig file in your $HOME directory. For example, if your $HOME was /Users/mike, then the global file would be at /Users/mike/.gitconfig.

 --system—Uses the etc/gitconfig file under wherever Git was installed. For example, if Git was installed into /usr/local/, the system file would be at /usr/local/etc/gitconfig; or if installed into /usr/, the system file would be at /etc/gitconfig.

 --local—Uses the .git/config file in a Git repository. For example, if a Git repository was at /Users/mike/GitInPracticeRedux/.git, then the local file would be at /Users/mike/GitInPracticeRedux/.git/config. .git/config is the default if no other configuration location flags are provided.

 --file (or -f)—Takes another argument to specify a file path to write to. For example, you could specify a file using git config --file /Users/mike/Documents/git.cfg.

If you used --global, you’d instead see the value that was set in the global configuration file. If you omit --local and --global, Git uses the same default priority as when reading configuration settings for its own use. The priority for deciding which configuration file to read from is as follows:

1 The argument following --file (if provided)

2 The local configuration file (.git/config)

3 The global configuration file (~/.gitconfig)

4 The system configuration file (etc/gitconfig under where Git was installed)

If a value is set for a key in a higher-priority file, Git’s commands use that instead. This lets you override the individual configuration among different repositories, users, and systems.

Although the global ~/.gitconfig file isn’t created until you set some values, on creation every repository contains a ~/.git/config file.

.

***7.1 Useful configuration settings***

Git 2.0 (which was released on May 28, 2014) defaulted to a new push strategy: the simple push strategy. This means branches are pushed to their upstream branch (set the first time with git push --set-upstream). Also, with the simple strategy, Git refuses to push if the remote branch name is different than the local branch name unless you specify it with an option such as git push origin remotebranchname

***Pruning branches automatically***

In Git, if multiple people are using the same repository and someone else deletes a remote branch, the remote branch reference (such as origin/remote-branch-name) won’t be deleted from your repository without running the git remote prune command. This is the same behavior as with tags; Git tries to avoid removing refs that may be useful to you unless you specifically request it. To prune the origin remote

branches, you would run git remote prune origin.

DOES PRUNING AFFECT LOCAL OR REMOTE BRANCHES?

Pruning doesn’t delete local branches, only references to remote branches. For example, suppose you had an inspiration branch that you had pushed to origin/inspiration. Later, someone deleted origin/inspiration. The origin/inspiration remote branch reference would only be deleted from your local repository after you ran git remote prune. But both before and after the prune, your local inspiration branch would remain unchanged.

I find it tedious to run this every time I want to remove a branch, and I would prefer it happened on every git fetch or git pull operation. To enable this behavior, you can run the following:

# git config --global fetch.prune 1

This means all remote branches will be pruned whenever you fetch or pull from a remote repository. This is particularly useful when you’re working on a repository where remote branches are created and deleted very regularly. This can occur in some workflows where direct commits to the master branch are discouraged, so branches are created for every change that needs to be made

***Ignoring files across all repositories: global ignore file***

For this reason, Git allows you to set a global ignore file in which you can put your

personal ignore rules (useful if others don’t want them in a repository). To tell Git you

wish to use a ~/.gitignore file, run the following:

# git config --global core.excludesfile ~/.gitignore

This global file behaves like any other .gitignore file, but you can put entries in it to be shared among all repositories.

***Technique 50 Aliasing commands***

Aliases allow you to create your own Git commands from combinations of other Git commands or by renaming them. This may be useful for making commands that are more memorableor easier to type. These are set as configuration values in the alias section

$ git config --global alias.ultimate-log "log --graph –oneline --decorate".

$ git config --global alias.l '!git ultimate-log'

Note the use of single quotes when setting the alias. These are required in this case: otherwise the Unix shell might not write the !,

Let’s create a command that does a fetch and then an interactive rebase

Run git config --global alias.fetch-and-rebase '!git fetch && git rebase

-i origin/master'.

This tells Git to go to the root of the working directory (the directory containing the .git directory), run git fetch, and, if it succeeds, run git rebase -i origin/master.

*Vendoring dependencies as submodules*

You wish to add a the GitInPracticeReduxSubmodule repository as a submodule of the GitInPracticeRedux repository in the master branch.

Run git submodule add https://github.com/GitInPractice/GitInPractice-

ReduxSubmodule.git submodule.

Commit the new submodule changes to the repository by running git commit

--message "Add submodule.".

Git uses the .gitmodules file and special metadata for the directory named submodule to reference the submodule and the current submodule commit. This ensures that anyone else cloning this repository can access the same submodules at the same version after initializing the submodule(s).

Initializing all submodules can be done by running git submodule init, which copies all the submodule names and URLs from the .gitmodules file to the local repository Git configuration file (in .git/config).

***Technique 53 Showing the status of submodules: git submodule status***

You wish to show the current states of all submodules of a repository.

git submodule status can take a --recursive flag, which runs git submodule status in each of the submodules directories too. This is useful because submodules can themselves contain submodules, and you may wish to query the status of the submodules within the submodules.

***Technique 54 Updating and initializing all submodules: git submodule update - - init***

You initialized a submodule (copied the submodule names and URLs in .gitmodules to .git/config) when you ran git submodule add earlier. But initialization won’t be done automatically for anyone else with a clone of this repository: they must run git submodule init.

Let’s simulate this situation by removing the current clone of the submodule in this repository:

* Run git submodule deinit.
* Run rm -rf .git/modules/ to remove the directory where Git caches submodules outside of the working directory.

Although the submodule is cloned into the submodule directory, it’s initially cloned into .git/modules too and then cloned from there to the submodule directory. Deleting this ensures that there’s no copy of the submodule’s repository in the current repository

You wish to initialize all submodules in your repository and update them to the latest revision

Run git submodule update --init

the git submodule update --init command would initialize the submodule in the local repository and then update the stored submodule revision to the latest revision in the local repository. This would then require another commit and push to update the remote repository. This should only be done after testing that the changes made to the GitInPracticeReduxSubmodule repository remain compatible with the GitIn-

PracticeRedux project

git clone can also take a --recurse-submodules (or --recursive) flag to automatically run git submodule update --init on any submodules in the repository. Typically, if you’re cloning a repository you know contains submodules, you’ll use git clone --recursive-submodules to clone it and all the necessary submodules (and the submodules of the submodules, if they exist).

***Technique 55 Running a command in every submodule: git submodule foreach***

Sometimes you may wish to perform a command or query in every submodule. Git provides the git submodule foreach command for this case: it takes a command (or commands) as an argument and then iterates through each Git submodule (and their submodules) and runs the same command.

You wish to output some status information for every submodule in the GitInPractice-Redux repository

# git submodule foreach 'echo $name: $toplevel:$path [$sha1]'

*12 Creating a clean history*

***Writing a good commit message***

It’s strongly influenced by a guide written by Tim Pope, which is now at http://mng.bz/hMe2

git add has a --patch (or -p) flag that provides an interactive menu in which you can select what parts of files you want to add.

git commit also has a --patch (or -i but, confusingly, not -p) flag. It provides the same interactive menu.

git commit --patch is equivalent to git add --patch && git commit

Git provides two GUI applications for this purpose: gitk (first seen in technique 4) and Git Gui

***Avoiding whitespace issues: git diff - -check***

Git expects certain whitespace usage in files. Many Git users (and almost all Git-based open source projects) try to avoid Git’s whitespace warnings. To do so, it’s always a good idea to try to ensure that your whitespace follows good Git practice:

* No lines in files end with whitespace (trailing tab or space characters).
* No lines in files start with one or more space characters followed immediately by one or more tab characters.
* All files end with one or more newline character(s): a line-feed character on Unix or a carriage-return and a line-feed character on Windows.

You can check that you haven’t violated any of these rules by running git diff --check.

git diff displays whitespace errors with a red background

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

$